COMP 621 Program Analysis and Transformations
Assignment #2

Using the McLab Framework for Analysis and Profiling

Due: Wednesday February 26, 2014

Overview:

The purpose of this assignment is to give you some practice designing flow analyses, and to familiarize you with the McLab analysis framework, so you will be comfortable with it for your course project. You have been given an introduction to the system during the lecture. Chapter 5 of Jesse Doherty's master's thesis (`http://www.sable.mcgill.ca/mclab/mcsaf/mcsafthesis.pdf`) contains a detailed description of the analysis framework.

Ismail (one of your TAs) has provided a framework of the code needed for each of the questions below. You can access that .tar.gz file at: `http://www.sable.mcgill.ca/~hendren/621/SVN/Assignments/2014/mclabtutorial.tar.gz`.

The individual files are in the directory `http://www.sable.mcgill.ca/~hendren/621/SVN/Assignments/2014/mclabtutorial/`.

Please feel free to ask questions of the graduate students in my research lab, the TAs during their office hours, and also feel free to share information with your classmates on the google group. The objective of this assignment is to get everyone comfortable with the environment - so don't spend a lot of time getting stuck on some small technical issue.

We will provide everyone in the class with some benchmarks which are known to run correctly with McLab.

Question 1 - *Measuring code coverage*

Code coverage is a metric which measures roughly how much (and which parts) of the source code of a program was exercised in a given execution of that program. There exist many different flavors of coverage data, for example describing which functions were called, which statements which executed, which branches or control flow paths were taken.

MATLAB's profiler, which you used in assignment 1, does provide some limited coverage information. It can tell you, for each function, which lines were executed. Here's an example. Suppose you have a function test:

```
function test()
  x = 1;
  y = 2;
  if x < y
    disp('then branch')
  else
    disp('else branch')
  end
end
```


Then you can get coverage information like this:

```
>> profile on
>> test()
>> profile off
>> stats = profile('info')
>> stats.FunctionTable(1).ExecutedLines

ans =

     2     1     0
     3     1     0
     4     1     0
     5     1     0
     9     1     0
```

Here the left column contains line numbers, the middle column corresponds to how many times that line was executed, and the third column corresponds to how much time was spent on that line (we don't care about this). Lines 2 and 3 correspond to the assignments, line 4 was the comparison, line 5 is the then branch, and line 9 is the end of the function.

Notice that lines that could have been executed but weren't, like line 7, aren't included in the table. This is bad. Typically you want to know which lines were executable, so that you have some meaningful measure of how much of the code was executed, e.g. 4/5 lines. You can't just take the number of lines in the file because you don't want to count blank lines, comments, and otherwise meaningless lines (like lines 6 and 8, which just consist of the 'else' and 'end' keywords). (There's an undocumented function, callstats, which among other things lets you get at an array of executable lines for a file, but it's brittle, and doesn't handle subfunctions and nested functions well.)

For this question, you should implement your own line coverage collection mechanism using McLab. Each node in the Natlab AST contains the line number of the first token corresponding to it in the original file (it's not perfect information, but in practice it tends to be good enough). You can get at the line number by calling the `getStartLine()` method, which is defined on all nodes.

This information can then be "injected" in the program by instrumenting each statement so that, if it's executed, the fact that that line was executed is recorded somewhere (perhaps in some global data structure). Before the program exits, it can use this information to

generate a coverage report. For our purposes, this can just be a plain text file, each line consisting of filename, line number, and 1 or 0 for executed or not executed.

For simplicity, you can use the same overall structure we saw in class for the profiler example – a driver function written in MATLAB, and a source to source transformation which is aware of what that driver function expects.

You should submit a short description of your approach, your source code, and some examples showing the pretty-printed IR before and after inserting the coverage collecting statements, as well as the coverage reports you generated for them. If possible, you could also try to produce a coverage report for your benchmark program (from assignment 1).

Question 2 - *Implementing an Analysis in McLab*

McLab provides a structured flow analysis framework that can be used to implement both forward and backward analyses. For this question you can choose `one` of the following analyses to implement. However, you are strongly encouraged to suggest your own analysis, just give a clear definition of it. The problems below are listed in increasing levels of difficulty.

- **IsScalar Analysis:** In MATLAB all variables are, by default, 2-dimensional arrays. However, many variables, for index variables like i, j or k are only ever used as scalar variables. Write an analysis, that determines at each program point, if a variable is: (1) scalar (this variable must be a scalar variable at this point), (2) non-scalar (this variable is definitely not a scalar at this program point), or (3) top (the variable could be a scalar or a non-scalar.
- **IsInteger Analysis:** The base type of variables in MATLAB is double, however often variables will only contain integer values (and hence can be stored more efficiently). Write an analysis, that determines at each program point, if a variable is: (1) integer (all values stored in this variable are integer at this program point) or (2) top (the types of the values stored in this variable are unknown).
- **Range Analysis:** For array-based programs it is very useful to know the range of values which summarizes the values a variable may have at a program point. Ranges are usually approximated as a pair [low, high], which indicates that the variable must have a value $\geq low$ and $\leq high$.

You should implement the analysis and use the pretty printing mechanism to print out the results of your program for several small illustrative examples. Choose the examples to illustrate the subtle points of your analysis and to show that it is correct for these cases.

For the purposes of this question you should submit a short description of your approach to implementing the analysis, the source code for your implementation (only those files you implemented), and several small example programs with the pretty-printed analysis results.

Question 3 - *Loop Invariant Code Motion*

Consider the following (contrived) code snippet:

```
x = input('x: ');
y = input('y: ');
for i = 1:100
  disp(i + x * y);
end
```

In this example, the expression `x * y` is constant across each iteration of the loop; such an expression is said to be *loop invariant*. A simple optimization, called *loop invariant code motion*, consists of taking all loop invariant expressions inside a loop and hoisting them up to be computed outside the loop and stored in a temporary variable. The loop above could be transformed to:

```
t = x * y;
for i = 1:100
  disp(i + t);
end
```

For this question, you will implement loop invariant code motion using McLab.

To determine whether an expression is loop invariant and therefore suitable for such a transformation, reaching definitions analysis can be used (it is up to you to determine precisely how). Rather than write your own reaching definitions analysis, you can simply use the implementation provided by the framework (`natlab.toolkits.analysis.core.ReachingDefs`). Using this, you should implement a source-to-source transformation that will hoist all loop invariant expressions identified in the input file.

You should submit a short description of your approach (including your algorithm for identifying whether an expression is loop invariant given reaching definitions information), your source code, and some examples showing the pretty-printed IR before and after performing your transformations.

Discuss any cases where you think your transformation might be unsafe.