OMSCS 6340 - Fall 2016
Assignment 8 (100 points)
Due at: October 31, 8am EDT

Objective: The objective of this assignment is to implement two analyses using Datalog: a *mod-ref analysis* and a *method-escape analysis*. Both analyses are flow-insensitive, context-insensitive, may (as opposed to must), and inter-procedural (requiring reasoning across method boundaries).

Resources:

- Chord Project Webpage: http://www.cc.gatech.edu/~naik/chord.html
- Chord Repository: https://bitbucket.org/pag-lab/jchord
- Chord user guide: http://pag-www.gtisc.gatech.edu/chord/user_guide/ (especially the chapter on how to write an analysis in Datalog: http://pag-www.gtisc.gatech.edu/chord/user_guide/datalog.html).
- Chord tutorial: http://pag-www.gtisc.gatech.edu/chord/pldi11/tutorial.pptx
- Download the zip file `constraints.zip` from T-Square which when uncompressed should produce a directory named `constraints/`.

Setup: A Linux or MacOS machine is preferred for running Chord. You will need to install the following software on your machine if not already present:

- If the command `javac` is not found on your machine, then install a JVM, e.g. Oracle JVM from http://www.oracle.com/technetwork/java/javase/downloads/index.html. You will need a JVM that supports Java 6 or 7 (note that Chord does not support Java 8). You can find the version number by running `javac -version`.
- If the command `ant` is not found on your machine, then install Apache Ant (a Java build tool) from `http://ant.apache.org/`.

Build the given example Java program to be analyzed, called TestCase, by running `ant` in the directory `constraints/examples/test_case/`. This should produce a subdirectory `classes/`. You can create your own test cases similar to TestCase in the `examples/` directory to test the analyses you will write.

Write your mod-ref analysis in the file `constraints/src/modref.dlog` and your method-escape analysis in the file `constraints/src/escape.dlog`. More details on what the analyses must do are given below.

To run your analyses, run the following commands in `constraints/`:

```
ant -Dchord.work.dir=examples/test_case/ modref
ant -Dchord.work.dir=examples/test_case/ escape
```

You can find the output of the analyses under `constraints/examples/test_case/chord_output`. Sample outputs for `test_case` are provided under `constraints/sample_output`.

Both analyses will use a pre-existing, flow- and context-insensitive pointer analysis with on-the-fly call graph construction (henceforth called pointer/call-graph analysis). This analysis is provided in `constraints/cipa_0cfa.dlog`. You can study it to learn how to write your own analyses in Datalog. It also declares all the relations, both input and output, that you will need to write your analyses. For instance:

- reachable methods in relation `reachableM`
- the call graph in relation `IM`
- points-to information in relations `VH`, `FH`, and `HFH`
- various relations that describe basic program facts (e.g., relation `MI` that species the containing method of each call site).

Note that any changes you make to this file will not be reflected when you run your analyses; Chord uses an analogous file in `chord.jar` for that purpose.

Description of Static Mod-Ref Analysis: The goal of static mod-ref analysis (short for modified-referenced analysis) is to determine which memory locations *may* be modified and which memory locations *may* be referenced during the execution of a method. The execution of a method includes the execution of any methods called by that method, directly or transitively. You must do this analysis for each reachable method (i.e., for each method in relation `reachableM`). The output of your analysis must be in the form of the following four relations (below, $\mathbb{M}$ denotes domain M, $\mathbb{H}$ denotes domain H, etc.):

- Relation `refStatField` $\subseteq (\mathbb{M} \times \mathbb{F})$ containing each tuple (*m, f*) such that pointer-typed static field *f* may be read during the execution of method *m*.
- Relation `modStatField` $\subseteq (\mathbb{M} \times \mathbb{F})$ containing each tuple (*m, f*) such that pointer-typed static field *f* may be written during the execution of method *m*.
- Relation `modInstField` $\subseteq (\mathbb{M} \times \mathbb{H} \times \mathbb{F})$ containing each tuple (*m, h, f*) such that pointer-typed instance field *f* of an object allocated at site *h* may be written during the execution of method *m*. Note that *f* may be the element 0 in domain F, in which case it denotes some pointer-typed element of an array allocated at site *h*.
- Relation `refInstField` $\subseteq (\mathbb{M} \times \mathbb{H} \times \mathbb{F})$ containing each tuple (*m, h, f*) such that pointer-typed instance field *f* of an object allocated at site *h* may be read during the execution of method *m*. As in the preceding item, *f* may denote pointer-typed array elements.

Use the following relations to write your analysis. They are also used as input relations in the pointer/call-graph analysis provided to you:

- Relation `MgetStatFldInst` $\subseteq (\mathbb{M} \times \mathbb{V} \times \mathbb{F})$ contains each tuple (*m, v, f*) such that the body of method *m* contains the instruction *v = f* where *f* is a static field of pointer type.

- Relation `MputStatFldInst` $\subseteq (\mathbb{M} \times \mathbb{F} \times \mathbb{V})$ contains each tuple (*m, f, v*) such that the body of method *m* contains the instruction *f = v* where *f* is a static field of pointer type.
- Relation `MgetInstFldInst` $\subseteq (\mathbb{M} \times \mathbb{V} \times \mathbb{V} \times \mathbb{F})$ contains each tuple (*m, v, b, f*) such that the body of method *m* contains the instruction *v = b.f*, where *f* is either an instance field of pointer type or it is the distinguished hypothetical field (denoted by element 0 in domain F) that designates any array element.
- Relation `MputInstFldInst` $\subseteq (\mathbb{M} \times \mathbb{V} \times \mathbb{F} \times \mathbb{V})$ contains each tuple (*m, b, f, v*) such that the body of method *m* contains the instruction *b.f = v*, where *f* is either an instance field of pointer type or it is the distinguished hypothetical field (denoted by element 0 in domain F) that designates any array element.

Besides the above relations, use the output relations of the pointer/call-graph analysis provided to you and described above. You will need to use both call-graph information (to determine method reachability) and points-to information (to determine the sites that may allocate objects whose instance fields or array elements may be read/written by a particular method). You can find the descriptions of these relations at
http://pag-www.gtisc.gatech.edu/chord/user_guide/predefined.html#id3

Note that the execution of a method includes the execution of any methods that are called by it directly or transitively; use relations `MI` and `IM` **recursively** to determine such methods.
Note: For this assignment, we will ignore reporting static fields, instance fields, and array elements of *non-pointer type* (called primitive type in Java, e.g., bool, int, float, etc.) that may be read or written during method execution. The above relations such as `MgetStatFldInst`, etc. already ignore such fields.

Description of Static Method-Escape Analysis: The goal of static method-escape analysis is to determine, for each reachable method *m* and for each object allocation site *h* in the body of *m*, whether any object allocated at *h* in any invocation of *m may* out-live that invocation. If so, the site is called *method-escaping*; if not, it is called *method-local* and all objects created at that site at runtime can be allocated on the method invocation stack (instead of in the heap, which is more expensive).
An object may out-live the invocation of its allocating method if any of the following conditions hold:

- An object allocated at site *h* may become reachable from some global variable (i.e., a static field in Java).
- An object allocated at site *h* may become reachable from some argument of method *m*.
- An object allocated at site *h* may become reachable from some return variable of method *m*.

You must do this analysis for each reachable method (i.e., for each method in relation `reachableM`). The output of your analysis must be a relation `localMH` $\subseteq (\mathbb{M} \times \mathbb{H})$ containing each tuple (*m, h*) such that allocation site *h* contained in the body of reachable method *m* is method-local (i.e., does *not* method-escape). Use the following relations to write your analysis.

- Relation `MmethArg` $\subseteq (\mathbb{M} \times \mathbb{Z} \times \mathbb{V})$ contains each tuple (*m, z, v*) such that variable *v* is the *zth* actual argument of method *m*.
- Relation `MmethRet` $\subseteq (\mathbb{M} \times \mathbb{Z} \times \mathbb{V})$ contains each tuple (*m, z, v*) such that variable *v* is the return value of method *m* (you can ignore *z*).

In addition, you may use the output relations of the pointer/call-graph analysis provided to you and described above. You will need to use both call-graph information (to determine method reachability) and points-to information (to determine the sites that may allocate objects that may escape). You can find the descriptions of these relations at http://pag-www.gtisc.gatech.edu/chord/user_guide/predefined.html#id3.

Note that this analysis requires reasoning about *reachability in the heap* (as exemplified by the use of the word "reachable" in the above three items); use the relation `HFH` **recursively** for this purpose. The relations `VH` and `FH` will provide the base cases for the recursion.

Submission Instructions: Upload your completed files `modref.dlog` and `escape.dlog` to T-Square.

Assignment Notes and Tips:

- The provided cipa_0cfa.dlog file is most useful as a syntactic reference. Attempting to understand the implementation details of each relation is not recommended.
- In past, students with databases experience have found it useful to view datalog relations as similar to SQL queries.
- A relation which is used in its own definition is recursive. Recursive relations must also have a base case. For example, a recursive relation A defined in terms of other relations B and C, may look something like this:
  - `A(i) :- A(i), B(i).`
  - `A(i) :- C(i).`

  Here the second relation is the base case.