# CS 6340 Final
# Fall 2011

Tuesday, Dec 13, 2:50pm-5:20pm

- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straight-forward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary.
- Partial solutions will be graded for partial credit.
- In accordance with both the letter and spirit of the Georgia Tech Honor Code, you will neither give nor receive assistance on this exam.

NAME: ______________________________________________

GT ID: ______________________

| Problem | Points | |
|---|---|---|
| 1 | 4 | |
| 2 | 4 | |
| 3 | 8 | |
| 4 | 4 | |
| 5 | 2 | |
| 6 | 8 | |
| 7 | 6 | |
| 8 | 8 | |
| 9 | 8 | |
| 10 | 6 | |
| 11 | 10 | |
| 12 | 7 | |
| 13 | 6 | |
| 14 | 5 | |
| 15 | 14 | |
| Total | 100 | |

1. **[4 points] Static vs. Dynamic Analysis**

   The course discussed numerous dynamic analyses (analyses that run the program) and static analyses (analyses that do not run the program).

   (a) **[1 point]** State one advantage of static analysis over dynamic analysis.

   *Solution:* Static analysis can be sound (i.e., whenever it does not report any bugs, the program indeed does not have any bugs) whereas dynamic analysis is unsound (i.e., it has false negatives).

   (b) **[1 point]** State one advantage of dynamic analysis over static analysis.

   *Solution:* Dynamic analysis can be complete (i.e., whenever it reports a bug, the bug is indeed real) whereas static analysis is incomplete (i.e., it has false positives).

   (c) **[2 points]** Static and dynamic analysis are complementary. State one way by which they can be combined in a manner that gains their advantages without suffering their drawbacks.

   *Solution:* Static analysis can be used to reduce the run-time instrumentation overhead of dynamic analysis. For instance, the performance of a dynamic race detector can be improved by using a sound static race detector to prove the absence of races on certain memory-accessing statements, thereby obviating the need to instrument and dynamically track races involving those statements.

10. **[6 points] Type Qualifiers**

Flow-sensitive type qualifiers `locked` and `unlocked` can be used to check the *consistent locking property*, namely, that `acquireLock` is invoked only a lock in the unlocked state and that `releaseLock` is invoked only on a lock in the locked state. State whether or not each of the following program fragments is typable using these type qualifiers using the CQual system. If not, would it be possible to make the fragments typable by using the restrict construct (discussed in the flow-sensitive type qualifiers paper) to restrict some value(s)? Briefly justify your answer in each case.

(a) [**2 points**]

```
if (...) l = l1; else l = l2;
acquireLock(l);
releaseLock(l);
```

where `l1` and `l2` are distinct locks initially in the unlocked state.

*Solution:* Yes.

(b) [**2 points**]

```
if (x > 0) acquireLock(l);
if (x > 0) releaseLock(l);
```

where `l` is a lock initially in the unlocked state.

*Solution:* No, since CQual is not path-sensitive. It is not possible to make the program typable even using restrict.

(c) [**2 points**]

```
for (i = 0; i < N; i++) {
    acquireLock(L[i]);
    releaseLock(L[i]);
}
```

where `L` is an array of `N` distinct locks, each of which is initially in the unlocked state.

*Solution:* No, but it is possible to make the program typable by using restrict for the expression `L[i]` in the loop body.

11. **[10 points] Datalog**

Datalog is a succinct declarative language for expressing program analyses. Assume that you are given the following two program relations for Java:

- Relation `IM` $\subseteq (\mathbb{I} \times \mathbb{M})$, containing each pair $(i, m)$ such that call site $i$ may call method $m$. This is the call-graph relation. Note that the same method may be called from multiple different call sites, and, due to dynamic dispatching in Java, the same call site may call multiple different methods.
- Relation `MI` $\subseteq (\mathbb{M} \times \mathbb{I})$, containing each pair $(m, i)$ such that the body of method $m$ contains call site $i$. Note that the same method may contain multiple different call sites in its body, though each call site is contained in a unique method's body.

Write Datalog rules to compute the following program relations using only the above two relations, and basic relations such as `!=` and `==` (e.g., you can use the clause $m_1$ `!=` $m_2$ or the clause $m_1$ `==` $m_2$ in any of your rules). You are free to use additional intermediate relations as long as you define them yourself using Datalog rules.

(a) **[2 points]** Relation `mayReach` $\subseteq (\mathbb{M} \times \mathbb{M})$ containing each pair $(m_1, m_2)$ such that method $m_1$ may call method $m_2$ directly or transitively.

*Solution:*

```
mayReach(m1, m2) :- MI(m1, i), IM(i, m2).
mayReach(m1, m2) :- mayReach(m1, m), mayReach(m, m2).
```

(b) **[2 points]** Relation `mustNotReach` $\subseteq (\mathbb{M} \times \mathbb{M})$ containing each pair $(m_1, m_2)$ such that method $m_1$ does not call method $m_2$ directly or transitively.

*Solution:*

```
mustNotReach(m1, m2) :- !mayReach(m1, m2).
```

(c) **[2 points]** Relation `recursive` $\subseteq (\mathbb{M} \times \mathbb{M})$ containing each pair $(m_1, m_2)$ such that method $m_1$ and method $m_2$ may be mutually recursive, that is, $m_1$ may call $m_2$ directly or transitively, and vice versa. Ensure that you handle the self-recursive case as well, in which $m_1 = m_2$.

*Solution:*

```
recursive(m1, m2) :- mayReach(m1, m2), mayReach(m2, m1).
```

(d) [**4 points**] Relation `chain` $\subseteq (\mathbb{M} \times \mathbb{M})$ containing each pair $(m_1, m_2)$ such that there is exactly one path in the call graph from method $m_1$ to method $m_2$.

Intuitively, a chain is a series of methods $m_1, m_2, ..., m_n$ such that each $m_i$ is a "wrapper" around $m_{i+1}$, and the only way to get to $m_{i+1}$ is via a unique call site in the body of $m_i$. Then, $(m_i, m_j) \in$ `chain` for all $i \in [1..n-1], j \in [i+1..n]$.

For instance, if $m_1$ has two call sites in its body that both call $m_2$, then $(m_1, m_2) \notin$ `chain`, but if $m_1$ has only one call site in its body that calls $m_2$, and there is no other method $m_3$ that calls $m_2$, then $(m_1, m_2) \in$ `chain`.

*Solution:*

```
multipleCallers(m2) :- IM(i1, m2), IM(i2, m2), i1 != i2.
notMultipleCallers(m2) :- !multipleCallers(m2).
chain(m1, m2) :- MI(m1, i), IM(i, m2), notMultipleCallers(m2).
chain(m1, m2) :- chain(m1, m), chain(m, m2).
```

12. **[7 points] Pointer Analysis**

Various heap abstractions exist that are more precise than allocation sites (0-CFA). We discussed "$k$-CFA with heap cloning" where call strings of length $\leq k$ are used to make further distinctions between objects created at the same allocation site. For instance, 0-CFA cannot prove the below `mustNotAlias` query, but 1-CFA with heap cloning can:

```
C getNew() { return new C(); }
void main() {
    C v1 = getNew();
    C v2 = getNew();
    mustNotAlias(v1, v2)?
}
```

[Note: `mustNotAlias`$(v_1, v_2)$ is equivalent to $\neg$`mayAlias`$(v_1, v_2)$.]

Another popular heap abstraction, called $r$-recency, is capable of distinguishing between the $r$ most recently created objects at an allocation site, both from each other pairwise as well as from all other objects created at that site. Consider the following program:

```
for (int i = 0; i < n; i++) {
    C v = new C();  // h
}
```

0-recency can distinguish between objects created at different allocation sites but not between those created at the same site. Thus, for the above program, it uses a single abstract location `h`.

1-recency uses two abstract locations `(h,0)` and `(h,*)` for the above program, where `(h,0)` abstracts the lone object created at site `h` in the most recent iteration of the loop, and `(h,*)` abstracts all other objects created at site `h` (in all earlier iterations of the loop).

2-recency uses three abstract locations `(h,0)`, `(h,1)`, and `(h,*)`, for the above program, abstracting the most recently created object, the next-to-most-recently created object, and all other objects, respectively.

In general, $r$-recency uses $r + 1$ abstract locations, one each for abstracting each of the $r$ most recently created objects at a site, and one for all the remaining objects created at that site.

(a) **[2 points]** Write the simplest program and `mustNotAlias` query you can think of, similar to the one shown above for 1-CFA, such that 0-recency cannot prove the query but 1-recency can.

*Solution:*

```
C v1 = getNew();
C v2 = getNew();
mustNotAlias(v1, v2)?
```

**OR**

```
C v1 = null, v2;
for (int i = 0; i < n; i++) {
    v2 = v1;
    v1 = getNew();
    mustNotAlias(v1, v2)?
}
```

(b) [**2 points**] Write the simplest program and `mustNotAlias` query you can think of, similar to the one shown above for 1-CFA, such that 1-recency cannot prove the query but 2-recency can.

*Solution:*

```
C v1 = getNew();
C v2 = getNew();
C v3 = getNew();
mustNotAlias(v1, v3)?
```

**OR**

```
C v1 = null, v2 = null, v3;
for (int i = 0; i < n; i++) {
    v3 = v2;
    v2 = v1;
    v1 = getNew();
    mustNotAlias(v1, v3)?
}
```

(c) [**3 points**] Order the following heap abstractions by precision:

- Types: distinguishes between objects of different types (i.e., classes) but not between those of the same type (for simplicity, assume that there is no subtyping).
- 0-CFA with heap cloning (described above).
- 1-CFA with heap cloning (described above).
- $\infty$-CFA with heap cloning.
- 0-recency (described above).
- 1-recency (described above).
- $\infty$-recency.

The $\infty$-CFA and $\infty$-recency abstractions are hypothetical but they naturally generalize $k$-CFA and $r$-recency, respectively. Use notation $A \preceq B$ to denote that "$A$ is at least as precise as $B$".

*Solution:*

$\infty$-CFA $\preceq$ 1-CFA $\preceq$ 0-CFA $\preceq$ types

$\infty$-recency $\preceq$ 1-recency $\preceq$ 0-recency $\preceq$ types

0-CFA = 0-recency

15. **[14 points] Intra-procedural Dataflow Analysis**

Consider an imperative single-procedure language where each primitive statement (quad) is of the form $v = e$, where $v$ is an integer variable and $e$ is an expression of any of the following forms:

$$e \ ::= \ c \ | \ v \ | \ v_1 + v_2 \ | \ v_1 - v_2 \ | \ v_1 \times v_2$$

where $c$ is an integer constant, and $v_1$ and $v_2$ are integer variables. The goal of this problem is to design a precise forward dataflow analysis that, upon completion, provides the range of integer constants $[c_1..c_2]$ that each variable may take at each program point, where both $c_1$ and $c_2$ are included in the range and $c_1 \leq c_2$.

(a) **[2 points]** Describe the (partial) ordering between abstract values that each variable can possibly take. The top (least precise) abstract value is $[-\infty, \infty]$. Specifically, define the binary relation $a \preceq a'$, meaning abstract value $a$ is at least as precise as abstract value $a'$ (where both $a$ and $a'$ are of the form $[c_1..c_2]$).

$$[c_1..c_2] \preceq [c'_1..c'_2] \quad \text{if} \quad c_1 \geq c'_1 \ \wedge \ c_2 \leq c'_2$$

(b) **[7 points]** Write the transfer functions for each of the five kinds of assignment statements. (Note: they should be as precise as possible.)

*Solution:* Let $\Gamma$ denote the abstract state, that is, a map from each variable in the program to its abstract value. Let $\delta$ denote the transfer function, which takes the statement and the incoming $\Gamma$ and produces the outgoing $\Gamma'$. Then:

$$\begin{array}{rcl}
\delta(v = c, \Gamma) & = & \Gamma[v \mapsto [c..c]] \\
\delta(v = u, \Gamma) & = & \Gamma[v \mapsto \Gamma(u)] \\
\delta(v = v_1 + v_2, \Gamma) & = & \Gamma[v \mapsto [lo_1 + lo_2..hi_1 + hi_2] \\
 & & \text{where } \Gamma(v_1) = [lo_1..hi_1] \text{ and } \Gamma(v_2) = [lo_2..hi_2] \\
\delta(v = v_1 - v_2, \Gamma) & = & \Gamma[v \mapsto [lo_1 - hi_2..hi_1 - lo_2] \\
 & & \text{where } \Gamma(v_1) = [lo_1..hi_1] \text{ and } \Gamma(v_2) = [lo_2..hi_2] \\
\delta(v = v_1 \times v_2, \Gamma) & = & \Gamma[v \mapsto [-\infty..\infty]]
\end{array}$$

(The transfer function for $\times$ can be made even more precise.)

(c) **[2 points]** Write the join function $[c_1..c_2] \sqcup [c'_1..c'_2]$ which shows how to merge a pair of abstract values of a variable arising from different paths.

*Solution:* $[c_1..c_2] \sqcup [c'_1..c'_2] = [min(c_1, c'_1)..max(c_2, c'_2)]$

(d) **[3 points]** Java performs a run-time check at each program point $p$ at which there is an array dereference $a[v]$, to ensure that $0 \leq v < a.length$. Assuming that the value of $a.length$ is stored in another variable $v'$ at program point $p$, state the condition under which the result of the above analysis can be used to eliminate this run-time check soundly.

*Solution:* Suppose the abstract value of $v$ is $[c_1..c_2]$ and that of $v'$ is $[c'_1..c'_2]$. Then, the check can be eliminated if $c_1 \geq 0$ and $c_2 < c'_2$.