Oral Exam Notes: Static Program Reduction via Type-Directed Slicing

Group 3: Specimin

Time Limit: 5 Minutes

1. The Problem (0:00 - 1:00)

- **Context:** Typecheckers (like `javac` or the Checker Framework) often crash or produce false positives on large, complex codebases.
- **The Conflict:** Users report bugs but provide massive codebases. Maintainers need a *minimal* reproduction case to fix the bug.
- **Why existing tools fail:** Existing reducers (like *Perses* or *C-Reduce*) are **dynamic**. They require the code to be runnable and run the typechecker repeatedly (thousands of times). This is:
  1. Too slow (hours/days).
  2. Impossible if the user cannot provide the full build environment/dependencies.

2. The Solution: Specimin (1:00 - 2:00)

- **Core Insight:** Typecheckers are **modular**. To typecheck a method, you don't need the implementation of its dependencies; you only need their **type signatures**.
- **Proposal:** A static tool called **Specimin** that performs *Type-Directed Slicing*.
- **Goal:** Create a standalone, minimal program that preserves the *compile-time behavior* (the bug) without preserving runtime semantics.

3. Methodology (2:00 - 3:00)

- **Type Rule Dependency Map:** They inverted the Java type rules. If a type rule says "To typecheck expression E, I need the type of variable X", Specimin maps $E \to X$.
- **Two Modes of Operation:**
  - **Exact Mode:** Requires full source code. Guarantees behavior preservation.
  - **Approximate Mode (The "Killer Feature"):** Works on **incomplete code**. If a class is missing (e.g., a library), Specimin *synthesizes* a dummy class/stub with the necessary structure to make the typechecker happy.
- **Algorithm:** It uses a worklist algorithm. It starts at the slicing criterion (the crash location) and iteratively adds dependencies based on the Type Rule Map.

4. Evaluation (3:00 - 4:00)

- **Dataset:** Tested on 28 historical bugs from *javac*, *NullAway*, and *Checker Framework*.
- **Effectiveness:**
  - Preserved the bug behavior in **89%** of cases using Approximate Mode (no classpath needed).
  - **93%** success in Exact Mode.
- **Performance:** Extremely fast. Average runtime was **15 seconds**, whereas dynamic reducers typically take much longer.
- **Reduction Size:** Reduced million-line programs to an average of **116 lines**.

5. Critique & Conclusion (4:00 - 5:00)

- **Strengths:**
  - **Handling Incomplete Code:** This is the biggest contribution. It removes the "it works on my machine" barrier for reporting bugs.
  - **Speed:** Static analysis is orders of magnitude faster than dynamic reduction.
- **Weaknesses (Crucial for exam):**
  - **Language Specific:** Relies on manually mapping Java type rules. Hard to port to C++ or Python.
  - **Oversizing:** The output (119 lines) is about 6x larger than human-minimized code (19 lines) because it is conservative.
  - **Scope:** Only works for typecheckers, not logic bugs or runtime crashes.