Improving Side-Effect Analysis with Lazy Access Path Resolving

Ju Qian

jqian@nuaa.edu.cn

Nanjing University of Aeronautics and Astronautics

Side-Effect Analysis

- Side-Effect Analysis determines the memory locations **modified** or **used** by each program entity.
- We concentrate on **method-level** side-effect analysis
- Side-Effect Analysis — the state-of-the-art
  - Based on Pointer Analysis
  - Location-Based Fashion
    - representing side-effects as abstract locations

Side-Effect Analysis

- Pointer Analysis — What we prefer?
  - Inclusion-based + Context-Insensitive
  - Benefits: Practical + acceptable precision
  - Typical Example: Spark in Soot

- Widely Preferred Side-Effect Analysis
  - side-effect analysis based on inclusion-based context-insensitive pointer analysis

Side-Effect Analysis: Problems

- Problems of side-effect analysis based on *inclusion-based context-insensitive* pointer analysis

  Still no good enough precision due to the context-insensitive nature

  - Side-effects under different calling contexts are not distinguished
  - Thousands of abstract locations in a single modification set

Side-Effect Analysis: What we want?

- a **lightweight** approach to improve the precision of side-effect analysis

Special Requirements

- Keep Scalable
- Prefer to not redesign the background pointer analysis
  - A new context-sensitive pointer analysis may affect scalability
  - A pointer analysis is often shared to achieve many different goals

    building a separate pointer analysis merely for side-effect collecting may introduce redundant computation.

Our Solution

- Fix the background pointer analysis, just **improve the precision of side-effect analysis under inclusion-based context-insensitive pointer analysis**

- **Basic Idea**

  Inspired by the following observation

  – In inclusion-based context-insensitive points-to analysis, the points-to sets of variables in the callers tend to be smaller than the ones in the callees

Method: lazy access path resolving

- **Inside a procedure**: Partly represent the side-effects of a method as access paths (e.g., p.x, p.y) on formal parameters
  - – For a modification, if its effects can be described by a formal access path with the help of *interstatement must aliases*
    - Then: represent it as access path
    - Else: represent it as abstract locations

- **Inter-procedure**: Propagate side-effects from the callees to the callers
  - – For access path, map from formal to actual
  - – For abstract locations, just merge to the caller

Method: **lazy access path resolving**

- The meaning of LAZY
  - During the bottom-up phase, a mod/use access path will never be resolved to the accessed locations as long as it could be propagated in access path form.
- Source of precision improvement
  - access paths in the caller can often be resolved to smaller abstract location sets

    (introducing some level of context-sensitivity)

How lightweight?

- Do not demand a new pointer analysis
- Do not use exhaustive access path representation
- Use must alias instead of backward tracing to determine if a MOD/USE can be represented with access path
- Compute interstatement must alias based on global value numbering

Experimental Results

- Less improvement when no heuristics used. Why?
  - Some method in large call depth has huge side-effect sets (due to Java library) that are difficult to refine, but widely propagated

The improvement seems minor compared to these huge side-effect sets

Experimental Results

- Using Heuristics:
  - Treat *Integer*, *String* and some other immutable types as build-in types
  - Ignore class initializations and finalize calls

  This will still be safe for many applications, although not for all

- In this case, the improvement is more significant
  - > 26% more precision for MOD effect computation.
  - > 25% of methods with side-effect sets reduced by more than a half

  The new method would be more beneficial for the applications where safety is not critical.

For Discussion

- It seems hard to largely improve the precision of side-effect analysis in limited time. Can we use heuristics to help the analysis? What other heuristics can we use?
- Many people spend a lot of time in analyzing the same codes. Can we build a standard repository to share the analysis results?
- For Java programs, what kinds of pointer analysis and side-effect analysis would be the most practical ones for program slicing?

Thanks !