Lecture 3. Computability and Complexity of Program Analysis

Wei Le

2016.2
Computability and Complexity

- Problem: Computability
- Algorithm for a defined problem: Complexity
- Tool/implementation: Scalability (when the size of the program increases, does the performance increase dramatically?)

Some example papers that discuss complexity of program analysis
- The Complexity of Andersen’s Analysis in Practice
- Undecidability of Static Analysis
- New Results on the Computability and Complexity of Points-to Analysis
Problem Definition: Points-to Analysis [ChakaravarthyPOPL03]

One Example

Problem Definition: We are given a set of pointers, a program (or say, its control flow graph) and two pointers $p$ and $q$. Three types of assignment statements are allowed in the program: (1) $\ast\ast\ast\ldots\ast x = \&y$, (2) $\ast\ast\ast\ldots\ast x = \ast\ast\ast\ldots\ast y$ and (3) $\ast\ast\ast\ldots\ast x = \text{NEW}$. The third statement creates a new unnamed variable and makes $\ast\ast\ast\ldots\ast x$ point to it. Two types of control flow statements are allowed: $if(\ldots)then\ldotselse\ldots$ and $while(\ldots)\ldots$. Now the goal is to check if there is some path from the start node to the exit node in the control flow graph, such that, at the end of executing the statements along the path, $p$ points to $q$. 
Problem Definition

- Restrictions (boundary of the state space) on the program:
  - variables (scalar or structure)
  - dynamic memory (allocate memory at run-time) – the configuration space of pointers is infinite
  - procedures (single or multiple)
  - level of pointer dereference
  - restricted data types: int* only can points to int

- Types of program analysis, e.g., whether it is flow-sensitive or flow-insensitive points-to analysis
Complexity of Types of Points-to Analyses [ChakaravarthyPOPL03]

Points-to Analysis

- With dynamic memory
  - (Undecidable)

- Without dynamic memory
  - With well defined types
    - Levels $\geq 2$ (PSPACE-Complete)
    - Levels $< 2$ (Is in P)
  - Without well defined types (PSPACE-Complete)
Complexity of Types of Points-to Analysis [ChakaravarthyPOPL03]

- Undecidable: variables can be structure or even only scalars, dynamic allocated memory
- NP-hard: dynamic memory is not allowed, flow-insensitive analysis, variables are scalars
- P: dynamic memory is not allowed, flow-insensitive analysis, variables – well defined data types
- PSPACE-Complete: dynamic memory is not allowed, flow-sensitive analysis, variables are more restricted

- it is not known: $P \subseteq NP$ or $P = NP$
- it is known: $P \subseteq PSPACE$ and $NP \subseteq PSPACE$, but not known whether $P \neq PSPACE$
- $PSPACEComplete$: at least as hard as $PSPACE$
- $NPComplete$: at least as hard as $NP$
- $NPHard$: at least as hard as $NPComplete$, don’t have to be $NP$, e.g., it can be undecidable
Why Do We Need to Know the Complexity of Program Analysis?

- Compare program analysis algorithms: flow-insensitive analysis is easier than flow-sensitive analysis.
- Understand that since the program analysis is undecidable, the existing program analysis is either:
  - **Unsound**: always gives correct answer when it terminates, but may run forever, or
  - **Incomplete**: Defined on a problem that is easier than program analysis on the real-world programs.

- **Overapproximation** and **Underapproximation**
- **Soundness** and **completeness**
Soundness and Completeness

A sound jury will never send a guilty person free, while a complete jury will never send an innocent person to jail.

Used in formal logic to describe the properties of a logic system

- a logic system is sound if the formula is valid and the inferences rules can derive it is valid

Used in static analysis, programming language type theory

- A program analysis identifies some program information for a goal.
- Soundness and completeness describe the properties of the analysis by comparing the solution computed by the algorithm and the ground truth
  - a sound analysis will never exclude correct program information from the solution, e.g., a and b are aliasing, a and b will be derived to be aliased in program analysis
  - a complete analysis will never report incorrect program information in the solution
Bug Detection Example by John Rushby

- We’re trying to guarantee absence of errors in a certain class
- Equivalently, trying to verify properties of a certain class
- Terminology is in terms of finding errors
  - **TP True Positive**: found a real error
  - **FP False Positive**: false alarm
  - **TN True Negative**: no error, no alarm—OK
  - **FN False Negative**: missed error
- Then we have
  - **Sound**: no false negatives
  - **Recall**: $\frac{TP}{TP+FN}$ measures how (un)sound
    - $TP+FN$ is number of real errors
  - **Complete**: no false alarms
  - **Precision**: $\frac{TP}{TP+FP}$ measures how (in)complete
    - $TP+FP$ is number of alarms
Other Tradeoffs in Practice by John Rushby

**Testing** is complete but unsound

**Spark Ada with its Examiner** is sound but not fully automatic

**Abstract Interpretation** (e.g., PolySpace) is sound but incomplete, and may not terminate

- Astrée is pragmatically complete for its domain

**Pattern matchers** (e.g. Lint, Findbugs) are not based on semantics of program execution, neither sound nor complete

- But pragmatically effective for bug finding

**Commercial tools** (e.g., Coverity, Code Sonar, Fortify, KlocWork, LDRA) are neither sound nor complete

- Pragmatically effective
- Different tools use different methods, have different capabilities, make different tradeoffs
More Properties of Program Analysis

- Flow-sensitive vs flow-insensitive
- Context-sensitive vs context-insensitive
- Path-sensitive vs path-insensitive
- Object-sensitive vs object-insensitive
- Field-sensitive vs Field-insensitive
- Intraprocedural vs interprocedural