Static Analysis: Automated Bug Hunting and Beyond

Profiling & Tuning Large Functional Programs

Julian Erhard Michael Schwarz
{julian.erhard, m.schwarz}@tum.de

Chair for Formal Languages, Compiler Construction, Software Construction
Department of Informatics, Technical University of Munich

Winter Term 2022/2023

Writing programs is hard.

Writing correct programs is very hard.

Testing

- Widely successful
- Can be automated to some extent
- Can only show that there are bugs, not their absence

Machine-verified proof (e.g. Isabelle)

- Can show bugs & their absence
- A highly manual process requiring highly trained people
- Problem with proof and implementation diverging

Static Analysis

- Fully automated
- Can show absence of certain classes of bugs
- Runs directly on the input program
- Abstract Interpretation, Model Checking, ...

Static Analysis

- Fully automated
- Can show absence of certain classes of bugs
- Runs directly on the input program
- Abstract Interpretation, Model Checking, ...

Abstract Interpretation

- Widely used both in Academia & Industry
- Can scale to huge industry-scale codebases
- The technique covered in Program Optimization Course (IN2053)

Goblint

- Analysis of multi-threaded, real-world C
- Efficient solvers for computation of fixpoints
- `https://goblint.in.tum.de`

Example

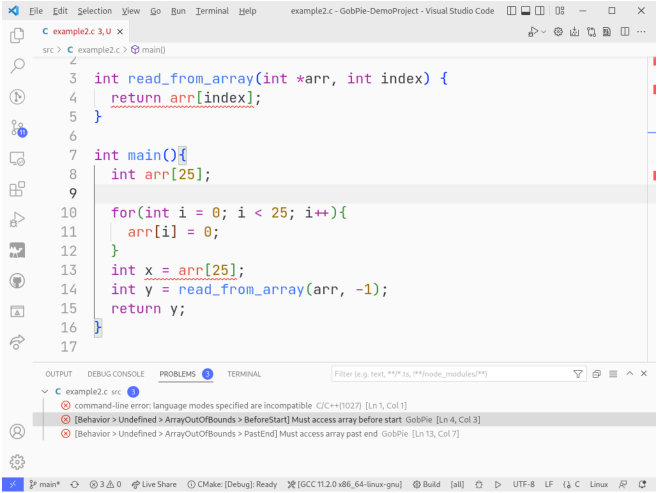

Figure: VS Code with the GobPie extension, showing warnings found by Goblint.

Profiling & Tuning
Large Functional Programs

Profiling & Tuning Large Functional Programs

- Large C programs contain hundreds of thousands of program points
- Computation can get expensive
- Where exactly are the bottlenecks?
  1. Use **profiler** to identify expensive and frequent operations
  2. Identify opportunities for improvements
  3. Implement and benchmark improvements
- Open topic, as it has not been deeply investigated yet.

One Possible Bottleneck?

During analysis of large code bases, we access vast amounts of program states, stored in a **large hashtable**, with hundreds of thousands of keys.

- How expensive are these lookups?
- Are cache-misses to blame?
- Can we do better?

Other Possible Points of Investigation?

- Are there places where naive **algorithms** can be replaced with more optimized ones?
- Do we benefit from **selectively abandoning immutability**?
- Could restricting the types of polymorphic functions increase performance?
- Would we benefit from `flambda`[1] optimizations?

[1] `https://v2.ocaml.org/manual/flambda.html`

Benefits

- Give you insights into profiling functional programs
- Deepen your skills in functional programming and writing performant code
- Help your understanding of the performance impact of high level design decisions
- Give you insights into developing a research prototype

Requirements

- Proficient knowledge of a functional programming language (we use OCaml)
- Program Optimization Course (IN2053) recommended, but not required
- Be an advanced Bachelor student or in your Master's

Static Analysis: Automated Bug Hunting and Beyond

Topics

- More expressive integer domains for **detection of overflows**
  - Integer overflow for signed types is undefined behavior in C
  - Mutually refining integer domains already implemented
  - Further enhance with e.g. Interval Sets

Topics

- More expressive integer domains for **detection of overflows**
  - Integer overflow for signed types is undefined behavior in C
  - Mutually refining integer domains already implemented
  - Further enhance with e.g. Interval Sets
- **Termination** analysis
  - Loops & recursion as sources of non-termination
  - Loops: Introduce ghost variables (c.f. ranking functions)
  - Recursion: Check abstract call graph for cycles

Topics

- More expressive integer domains for **detection of overflows**
  - Integer overflow for signed types is undefined behavior in C
  - Mutually refining integer domains already implemented
  - Further enhance with e.g. Interval Sets
- **Termination** analysis
  - Loops & recursion as sources of non-termination
  - Loops: Introduce ghost variables (c.f. ranking functions)
  - Recursion: Check abstract call graph for cycles
- Analyzing **C11** code: C11 finally gaining traction
  - New threading library
  - thread-local variables
  - `Noreturn` keyword

Benefits

- Deepen your understanding of
  - The Semantics of C and typical programming errors
  - Static Analysis by Abstract Interpretation
- Train your functional programming skills
- Give some insights into developing a research prototype

Requirements

- Program Optimization Course (IN2053)
- Knowledge of a functional programming language (we use OCaml)
- Be in your Master's (Advanced Bachelor's students welcome)

Profiling & Tuning
Large Functional Programs

&

Static Analysis:
Automated Bug Hunting and Beyond

Format

- Teams of 2-5 students
- Course will take place throughout the semester
- (Bi-)weekly meetings with us, default in person
- Presentation at the end (one day, all groups)
  - Attendance & Active Participation mandatory(!)

Questions?