DDC: a declarative debugger for C++

Roland Coeurjoly, directed by Adrián Riesco Rodríguez

July 7, 2022

Outline

1. Introduction
2. Goals
3. State of the art
4. Tool description
5. Formal verification
6. Conclusions and future work
7. Afterthoughts
8. Questions?

Introduction: Declarative debugging

Declarative debugging, also called algorithmic debugging, is a debugging technique. Steps:

- taking a program execution which the user deems incorrect,
- building an debugging tree (DT) abstracting the execution of this program, and
- asking questions about the correctness of computations to an oracle (usually the user, but other sources such as test cases can be used) until a certain computation is narrowed down as buggy.

Goals

Develop a declarative debugger for the C++ language

- Integrated in workflow
- No changes to the program being debugged
- Tree transformations
- Test cases as oracles

State of the art: Problems

- The Scalability Problem.
  - Time Needed to Generate the Debugging Tree
  - Memory Needed to Store the Debugging Tree
- User Experience and Effectiveness.
  - Amount and Difficulty of the Questions
  - Rigidity and Loss of Control During the Navigation Phase
  - Bug Granularity
- Completeness.
  - Termination
  - Concurrency
  - I/O

State of the art: Scalability issue for imperative programs

Quotes from pages 139 and 140 of Why programs fail, by Andreas Zeller

> *Algorithmic debugging has not been demonstrated to be effective for real-world applications.*
>
> *The process does not scale. In a (. . .) imperative program, (. . .) many (. . .) functions communicate via shared data structures, rather than simple arguments and returned values. (. . .) the data structures being accessed are far too huge to be checked manually. Imagine debugging a compiler: "Are these 4 megabytes of executable code correct (yes/no)?*
> *For these reasons, algorithmic debugging works best for functional and logical programming languages.*

State of the art: Declarative debuggers for many language paradigms

- Functional
- Declarative
- Imperative
- Object oriented

Demo

Possible demos:

- Normal debugging session
- Debugging a non terminating program
- Performing debugging tree transformations

Tool description

Use of GDB Python API and rr, a reverse debugging tool.

Formal verification done in Coq

- types
  - Correctness
  - Node
- functions
  - Weight
  - *generic_debugging_algorithm*
- lemmas
  - Proving *generic_debugging_algorithm* correct

Conclusions: all goals fulfilled

- Integrated in workflow: DDC uses GDB, the most common debugger for C++ under GNU/Linux
- No changes to the program being debugged: only debugging symbols needed
- Tree transformations: Simplified Tree Compression implemented (novel)
- Test cases as oracles: server sends correct nodes to client

Future work: Complete the formal verification of all algorithms

The generic debugging algorithm returns a node that was in the initial debugging tree.

Future work: Test programming languages other than C++

Possible with GDB Python API

Afterthoughts: Specifications of functions

> *With future tools, (. . .) programmers providing specifications of particular function properties-specifications that can then be reused for narrowing down the incorrect part.*

Contracts a la Dafny where proposed for C++20. Finally rejected, no hope until at least C++26.

Afterthoughts: All problems solved with external oracle

Specification Mining: New Formalisms, Algorithms and Applications, by Wenchao Li (PhD Thesis)

> *specification mining – the process of inferring likely specifications by observing a design's behaviors*

Questions?

Thank you! Questions?