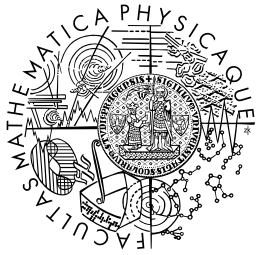

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Denis Leskovar

Automated Program Minimization With Preserving of Runtime Errors

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: doc. RNDr. Pavel Parízek, Ph.D.
Study programme: Computer Science
Study branch: System Programming

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

Dedication.

Title: Automated Program Minimization With Preserving of Runtime Errors

Author: Denis Leskovar

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Debugging of large programs is a difficult and time consuming task. Given a runtime error, the developer must first reproduce it. He then has to find the cause of the error and create a bugfix. This process can be made significantly more efficient by reducing the amount of code the developer has to look into. This paper introduces three different methodologies of automatically reducing a given program $\mathcal{P}$ into its minimal runnable subset $\mathcal{P}'$. The automatically generated program $\mathcal{P}'$ also has to result in the same runtime error as $\mathcal{P}$. The main focus of the reduction is on correctness when operating in a concrete application domain set by this study. Implementations of introduced methodologies written using the LLVM compiler infrastructure are then compared and classified. Performance is measured based on the statement count of the newly generated program and the speed at which the minimal variant was generated. Moreover, the limits of the three different approaches are investigated with respect to the general application domain. The paper concludes with an overview of the most general and efficient methodology.

Keywords: automated debugging, code analysis, syntax tree, statement reduction, clang

Contents

Introduction

TODO: Rewrite the introduction to include goals.

Automation of routine tasks tied with software development has resulted in a tremendous increase in the productivity of software engineers.

However, the task of debugging a program remains mostly manual chore. This is due to the difficulty of reliably encountering logic-based runtime errors in the code, a task that, to this day, requires the developer's attention and supervision.

Let program $\mathcal{P}$ contain a runtime error E that consistently occurs when $\mathcal{P}$ is run with arguments A. Since the error E is present at runtime and not compile-time, it can be assumed that syntax wise the code is mostly correct. Therefore, any syntax-based error can be ruled out. This, in turn, leaves us with a set of logical errors $\mathcal{E}_{\mathcal{L}}$. Those include wrongly indexed arrays and calculations that lead to either the incorrect result or an altered control flow of the program. Let $E \in \mathcal{E}_{\mathcal{L}}$. As the generality of errors in $\mathcal{E}_{\mathcal{L}}$ appears too complicated to be solved for all programming languages at once, it is necessary to break the problem down for each programming language. This article is concerned with the logical errors of C and C++. Although C is not a subset of C++, the logical errors made in C can be approached similarly to those in C++. Both languages share mostly comparable constructs. Finding the cause of a logical error in a concrete language requires knowing these constructs, their behavior, and their general handling.

To make finding the cause of an error a systematic approach, one might try removing unnecessary statements in the code, thus minimizing the program. Let $\mathcal{P}'$ be a minimal variant of $\mathcal{P}$ such that $\mathcal{P}'$ results in the same error E as $\mathcal{P}$ when run with the same arguments A. If done carefully and correctly, $\mathcal{P}'$ represents the smallest subset of $\mathcal{P}$ regarding code size, while preserving the cause of the error in that subset. Upon manual inspection, the developer is required to make less of an effort to find the cause in $\mathcal{P}'$ as opposed to $\mathcal{P}$.

The minimization of a program can be achieved in numerous ways. In further sections, the article describes and compares three different approaches. The first is based on naive statement removal and its consequences during runtime. The second removes major chunks of the code while periodically testing the generated program's correctness. The third deploys a sequence of code altering techniques, namely slicing and delta debugging.

1. Automated debugging techniques

TODO: Link relevant literature from Slicing of LLVM bitcode (muni.cz) and Bobox Runtime Optimization (cuni.cz)

Debugging can be described as the process of analyzing erroneous code to find the cause of those errors. Errors can also be of different natures. It can for example stem from poor design of the application. If that is not the case, then perhaps it comes from a rarely encountered input or a corner-case. The flaw might also be present in external code such as libraries or inappropriate usage of existing technologies.

It can be said with confidence that debugging is rarely an algorithmic approach. While the goal is clear, the process of debugging depends entirely on the programmer. It is typical that developers try to look for a root cause of an error by feeling what might be wrong. This works rather well in code the programmer is familiar with. However, in larger projects the developer did not create by himself, more sophisticated and reliable approaches are required. For example, one might add logging to the code being debugged, or perhaps create more tests that can narrow down the errorneous code.

All of the mentioned techniques require either the knowledge of the code or enough time to write supporting code. Additional time might be spent looking through the logs and executing tests. Therefore, it is rather hard to tell beforehand how much time and resources debugging will take.

While most developers see debugging as a manual chore, there were numerous attempts at automating at least some parts of it during the last few decades. The rise in popularity of program analysis resulted in automated error checks for popular programming languages.

SpotBugs [1], formerly known as FindBugs, is a free and platform-independent application for, as the name suggests, finding bugs. It works with the bytecode of JDK8 and newer, which indicates that source code is not required. SpotBugs uses static analysis to discover bug patterns. These patterns are sequences of code that might contain errors. They include misused language features, misused API methods, and changes to source code invariants created during code maintenance. Java developers can use SpotBugs's static analysis in its GUI form or as a plugin for build tools.

Clang static analyzer [2] provides similar functionality to C, C++, and Objective-C programmers. The code written in these languages is parsed by the analyzer. A collection of code analyzing techniques is then applied to it. This process results in an automatic bug finding, similar to compiler warnings. These warnings, however, include runtime bugs as well. The analyzer can uncover many bugs, from simple faulty array indexing to guarding the stack address scope. Due to its extensibility and integration in tools and IDEs alike, the Clang static analyzer is popular amongst developers working with the C family of languages.

The functionality of the previous tool was extended in CodeChecker [3]. CodeChecker serves as a wrapper for the Clang static analyzer and Clang-Tidy. Wrapping these two tools into a more sophisticated application helps with user-friendliness tremendously. Additionally, the wrapper also deals with false positives.

Furthermore, it allows the user to visualize the result as HTML or save time by analyzing only relevant files.

Facebook's Infer [4] translates both the C family of languages and Java into a common intermediate language. It also utilizes compilation information for additional accuracy. The intermediate code is then analyzed one function at a time. During the analysis, Infer can uncover tedious bugs such as invalid memory address access and thread-safety violation.

While the tools mentioned above mainly cover only specific cases of potential bugs, such as out-of-range array indexing, they have proven themselves valuable for the developer. In the context of this work, such checks provide a helping hand at a low cost when minimizing a program. The following sub-chapters will talk about the techniques behind such checks and how they deal with automated debugging.

Although these checks mostly cover only specific cases of potential bugs, such as out-of-range array indexing, they have proven themselves as a useful tool for the developer. Upon further developement, static analysis shifted from purely syntactic checks to analyzing the semantics code. In the context of this work, such checks provide a helping hand at a low cost when minimizing a program.

The following sections will talk about the techniques behind such checkers and how they deal with automated debugging. Notably, they describe the motivation and notation of Delta debugging and static and dynamic slicing.

1.1 Delta debugging

Delta debugging is an iterative approach described by Zeller [5]. It does not perform any static analysis of the debugged program, as it is not meant to find failures in the code.

Delta debugging instead intends to minimize the debugged program's incorrect input to isolate the input's failure-inducing part. Therefore, it requires the program in question and the specific input and the expected output. In other words, Delta debugging requires a set of test cases, which attempts to minimize and isolate the failure-inducing input. In order to understand where this approach came from, we need to define several important properties of programs and test cases presented by Zeller and Hildebrandt[6].

Definition 1 (Test case)**.** *Let $c_{\mathcal{F}}$ be a set of all changes $\delta_1, \ldots, \delta_n$ between a passing program run $r_{\mathcal{P}}$ and a failing program run $r_{\mathcal{F}}$ such that*

$$r_{\mathcal{F}} = (\delta_1(\delta_2(\ldots(\delta_n(r_{\mathcal{P}}))))).$$

We call a subset $c \subseteq c_{\mathcal{F}}$ a test case*.*

Test runs should differ based on program's code. The definition states that making changes to the program, i.e. applying a function that alters the code, enough times and well enough to transform a passing program to a failing one, results in a test case.

Definition 2 (Global minimum)**.** *a test case $c \subset c_{\mathcal{F}}$ is called a* global minimum *of $c_{\mathcal{F}}$ if $\forall c_i \subseteq c_{\mathcal{F}} : (|c_i| < |c| \implies c_i$ does not cause the program to fail.)*

Global minimum can be interpreted as the smallest set of changes able to make the program fail.

Definition 3 (Local minimum)**.** *a test case* $c \subset c_{\mathcal{F}}$ *is called a* local minimum *of* $c_{\mathcal{F}}$ *if* $\forall c_i \subseteq c : (c_i$ *does not cause the program to fail.*$)$

The aforementioned minimality is defined as follows.

Definition 4 (n-minimality)**.** *a test case* $c \subset c_{\mathcal{F}}$ *is* n-minimal *if* $\forall c_i \subseteq c : (|c| - |c_i| \leq n \implies c_i$ *does not cause the program to fail.*$)$

The minimizing Delta debugging algorithm attempts to find a 1-minimal test case.

Delta debugging seems to bet on the premise that large-scale applications are written with automated testing in mind. On the same note, it is the recommended practice to develop programs while at the same time dedicating resources to write tests for that program.

The defined minimality can be used to construct the minimizing algorithm. However, the delta debugging algorithm can be easily and more comprehensively explained without the definition as well.

TODO: Move comments more towards the center.

Algorithm 1 Minimizing Delta Debugging Algorithm.

1: $n \leftarrow 2$
2: Split a string σ into $\alpha_1, \ldots, \alpha_n$ of equal size. ▷ Where σ is test's input.
3: For each α_i, calculate its complement β_i.
4: Run tests on $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$.
5: **if** all tests passed **then**
6: $\quad n \leftarrow 2 * n$
7: $\quad$ **if** $n > |\sigma|$ **then return** the most recent failure causing substring.
8: $\quad$ **else**
9: $\quad\quad$ goto (2).
10: $\quad$ **end if**
11: **else if** $\alpha_i failed$ **then**
12: $\quad n \leftarrow 2.$
13: $\quad \sigma \leftarrow \alpha_i.$
14: $\quad$ **if** $|\sigma| == 1$ **then return** σ.
15: $\quad$ **else**
16: $\quad\quad$ goto (2).
17: $\quad$ **end if**
18: **else** ▷ β_i failed.
19: $\quad \sigma \leftarrow \beta_i.$
20: $\quad n \leftarrow n - 1.$
21: $\quad$ goto (2).
22: **end if**

The simplified algorithm seen in 1.1 splits the input into n even-sized parts and their respective complements. Tests are then applied to these snippets, which can result in three different outcomes. If all tests pass correctly, the granularity, i.e., n, is doubled, and the input is split into more even-sized parts. On the other hand,

if a snippet fails a test, the granularity is reset to its initial value. Furthermore, the snippet now becomes the input. If neither of the two mentioned scenarios happens, then a snippet's complement must have failed to pass a test. This case results in the granularity being decreased, and the input is set to the failure causing complement. These three actions repeat iteratively, updating the input and splitting it systematically with different granularities. Once the granularity is greater than the input's size, the most recent failure-inducing snippet is returned. The same case holds when the input is of size 1, i.e., it cannot be further divided.

Additionally, minimizing is not the only approach Delta debugging suggests. A more sophisticated one is isolation. Minimization can be described as removing parts while the failure persists, which means that the output changes are only made in failing iterations. Isolation extends this by adding failure-inducing differences while the program passes tests. This addition results in changes in both the passing and failing iterations.

One can quickly transform the input minimalization of Delta debugging into either source code minimalization or error isolation at both the compile-time and runtime. This transformation can be achieved for the compile-time by first setting the input as the debugged program's source code. Second, it is required to set the expected output to either 'compiled' or 'failed to compile'. Finally, the input is fed into a compiler, for example, GCC, which produces one of the two set outputs.

The runtime variant only differs in two points—first, changing the expected outputs. Second, changing the compiler to a compiler-debugger pipeline so that the source can be compiled and run.

1.2 Static slicing

Program slicing, formalized more than three decades ago, is a branch of program analysis that studies program semantics. It systematically observes and alters the program's control-flow and data-flow for a given statement and variable in the code. The goal of slicing is to create a slice of a program, i.e., a series of parts of the program that could potentially impact the control and data flow at some given point in that program. The direction from which the target statement is approached divides slicing methods into two groups. Firstly, forward slicing uncovers parts of the code that might be affected by the targeted statement and variable. Secondly, and much more common, backward slicing computes parts of the program that impacts the targeted statement.

The first introduced slicing method was static backward slicing. And with it came brand new formalism concerning program analysis. Specifically for static slicing methods, definitions for the target 4 statement and variable needed to be written. Weiser [7] defined a slice with respect to criterion C as a part of a program that potentially affects given variables in a given point.

Definition 5 (Static slicing criterion)**.** *Let $\mathcal{P}$ be a program consisting of program points $P = p_1, \ldots, p_n$ and variables $V = v_1, \ldots, v_m$. Any pair $C = (p_i, V')$, such that $p_i \in P$, $V' \subseteq V$, and $\forall v_i \in V' : v_i$ is present in p_i, is called a* slicing criterion*.*

Slicing is the process of finding such a part of a program. Suggested approaches neglected any execution information and focused solely on observations

made by analyzing the code.

One can imagine that the size of a static slice would be much smaller than the original program. That would be the case in modular code that rarely interacts between its components. An example of such code would be heavy parallel applications and computational tasks. However, in programs with aggressive use of branching, it is not so. Since static slicing considers statements that **might** impact the criterion, it leaves otherwise useless branches in the slice, thus negating the potential decrease in size.

In listing 1.1, we can see the code of a simple program. It loads a value a, which then alters the control-flow of the code. Meanwhile, it iterates through a printing loop. The intriguing part, however, is the output of the $write(x)$ command on line 42. Let the criterion be $C = (write(x)_{42}, \{x\})$. The value of x on that line is changed in the branching part of the program, which entirely depends on the value of a. Since a is unknown, no significant code reduction can be made. The static slice with respect to C, seen on 1.2, still contains all of the branching statements. Note that the independent printing loop is gone.

Later that year, K. J. Ottenstein and L. M. Ottenstein [8] restated the problem as a reachability search in the program dependence graph (PDG). PDG represents statements in the code as vertices and data and control dependencies as oriented edges. Additionally, edges induce a partial ordering on the vertices. In order to preserve the semantics of the program, statements must be executed according to this ordering.

Edges are, therefore, of two types. First, the control dependency edge specifies that an incoming vertex's execution depends on the outgoing one's execution. Second, the data flow dependence edge suggests that a variable appearing in both the outgoing and incoming edge share a variable, the value of which depends on the order of the vertices execution.

Once the PDG is built, slices can be extracted in linear time with respect to the number of vertices.

Figure 1.1 shows a PDG that was extracted using an AST Slicer. Nodes of the graph contain the same statements as seen in the code. Frameworks that achieve such mapping between the code and the internal control and data flows allow developers to create slicing tools much more easily. One such framework is the LLVM/Clang Tooling library, which will be talked about later. The tool is available at `https://github.com/dwat3r/slicer`.

However, one can find many potential issues and obstacles when performing data flow analysis. Omitting the interprocedural slicing, as it is not relevant in this paper's context, one is left with pointers and unstructured control flow. While the latter is rarely used in single-threaded modern programming, the same cannot be said about the former.

Pointers require us to extend the syntactic data flow analysis into a pointer or points-to analysis, which should be performed first. It is necessary to keep track of where pointers may point to (or must point to, in case their address is not reassigned) during the execution. From this knowledge, other data flow edges must be created or changed to accommodate the fact when the outgoing vertex mayhap writes into a memory location possibly used by the incoming vertex.

The analogical approach is then used for control dependency analysis since pointers might alter control flow as well. This change to control flow happens,

Listing 1.1: Simple branching program.

```
1  #include<iostream>
2
3  void write(int x)
4  {
5    std::cout << x << "\n";
6  }
7
8  int read()
9  {
10   int x;
11   std::cin >> x;
12
13   return x;
14 }
15
16 int main(void)
17 {
18   int x = 1;
19   int a = read();
20
21   for (int i = 0;
22   i < 0xffff; i++)
23   {
24     write(i);
25   }
26
27   if ((a % 2) == 0)
28   {
29     if (a != 0)
30     {
31       x *= -1;
32     }
33     else
34     {
35       x = 0;
36     }
37   }
38   else
39   {
40     x++;
41   }
42
43   write(x);
44
45   return 0;
46 }
```

Listing 1.2: Static slice of the simple branching program.

```
#include<iostream>                1
                                  2
void write(int x)                 3
{                                 4
  std::cout << x << "\n";         5
}                                 6
                                  7
int read()                        8
{                                 9
  int x;                          10
  std::cin >> x;                  11
                                  12
  return x;                       13
}                                 14
                                  15
int main(void)                    16
{                                 17
  int x = 1;                      18
  int a = read();                 19
                                  20
                                  21
                                  22
                                  23
                                  24
                                  25
                                  26
  if ((a % 2) == 0)               27
  {                               28
    if (a != 0)                   29
    {                             30
      x *= -1;                    31
    }                             32
    else                          33
    {                             34
      x = 0;                      35
    }                             36
  }                               37
  else                            38
  {                               39
    x++;                          40
  }                               41
                                  42
  write(x);                       43
                                  44
  return 0;                       45
}                                 46
```

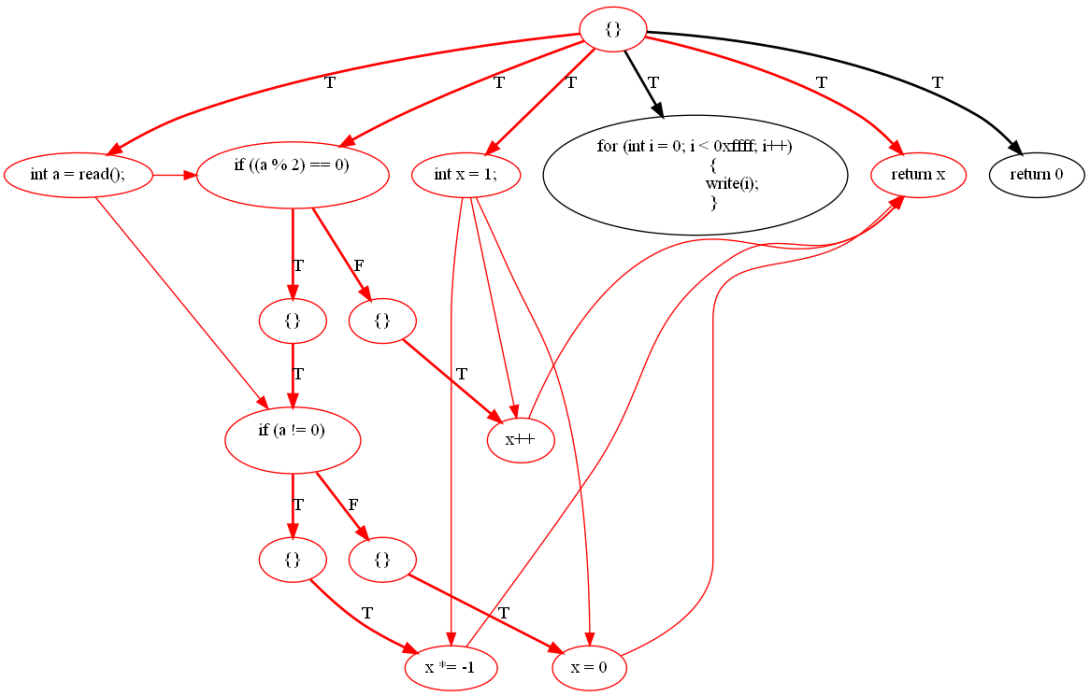

Figure 1.1: Sliced PDG. The graph was created from the source code of 1.1. Red edges indicate the sliced part of the program w.r.t. $C = (write(x)_{42}, \{x\})$.

namely when functions are called using function pointers.

The main advantage of static slicing is that it does not require any runtime information. As program execution can be expensive both time-wise and resource-wise, static slicing offers program comprehension at a low cost. Because static slicing discovers program statements that can affect certain variables, it can remove dead code and be used for program segmentation.

Furthermore, static slicing is used for testing software quality, maintenance, and test, all of which are relevant to this project.

1.3 Dynamic slicing

While the idea of building a program slice prevails, dynamic slicing drastically differs from static slicing in terms of input and the way it is processed.

Korel [9] described a slicing approach that took into consideration information regarding a program's concrete execution. As opposed to static slicing, which builds a slice for any execution, dynamic slicing builds a slice for a given execution of a program. Using information available during a run of the program results in a typically much smaller slice.

This decrease in size is mainly due to removing unnecessary branching of control statements and unexecuted statements in general. The slicing criterion now contains a set of the program's arguments in addition to the previous information. The location of the criterion's statement is also specified to avoid vagueness in the execution history.

The criterion is therefore defined as follows.

Listing 1.3: Dynamic slice of the simple branching program.

```
1  #include<iostream>
2
3  void write(int x)
4  {
5    std::cout << x << "\n";
6  }
7
8  int read()
9  {
10   int x;
11   std::cin >> x;
12
13   return x;
14 }
15
16 int main(void)
17 {
18   int x = 1;
19   int a = read();
20
21   x = 0;
22
23   write(x);
24
25   return 0;
26 }
```

Definition 6 (Dynamic slicing criterion)**.** *Let $\mathcal{H} = (s_{x1}, \ldots, s_{xn})$ be an execution history of a program $\mathcal{P} = (\{s_1, \ldots, s_m\}, V)$, where s_i denotes a statement and V is a set of variables $v_1, \ldots, v_k$. Any triple $C = (h_i, V', \{a_1, \ldots, a_j\})$, such that $h_i \in \mathcal{H}$, $V' \subseteq V$, $\forall v_i \in V' : v_i$ is present in h_i, and $\{a_1, \ldots, a_j\}$ is the input of the program, is called a* slicing criterion*.*

The example listing 1.3 was computed from the original listing 1.1. The criterion was set to $C = (write(x)_{42}, \{x\}, \{2\})$. Since the dynamic slicer witnessed the program's execution, it could precisely reduce the code to only those statements that were executed. the result is a significantly smaller slice than in the case of 1.2. Note that branching statements are gone.

Since dynamic slicing requires the user to run the program, it is typically used in cases where the execution with a fixed input happens regardless. Such cases include debugging and testing. For debugging, dynamic slices must reflect the subsequent restriction: a program and its slices must follow the same execution paths.

1.4 Summary

While the described program minimizing and debugging approaches have been formulated more than two decades ago, there have not been nearly enough successful attempts at implementing them.

With each approach having its clear positives and negatives, it would be interesting to see how they handle program minimization. When cleverly used, a combination of these methods might result in a reasonably fast and inexpensive algorithm for the reduction of program size.

2. Compilers and analysis tools

In the previous chapter, the reader was introduced to a branch of program analysis. The techniques discussed above focused on both the static and runtime side of program analysis.

Regardless of whether these approaches have been implemented, it was required to find a suitable tool for source code manipulation for two reasons. First, any external tool output might require altering the input source code based on its output. Second, if implementing any code reducing algorithm would have to occur, one would need a sophisticated code modifying framework.

Due to these reasons, an analysis of compilers and tools for C and C++ was conducted. The goal of the analysis is to pick the most practical tool available. Required criteria include frequent upkeep of the framework, an existing user base, and the ability to manipulate some abstract representation of the code.

The representation boiled down to an abstract syntax tree (AST). AST embodies the syntactic structure of the code, regardless of the code's language. A vertex of an AST represents a construct of the code while not being concrete with the programming language's details. This generality is perfect for C and C++'s chosen domain, as both languages only differ syntax-wise in minor details.

Below are the findings concerning the most important candidates.

2.1 GCC

A well-known C and C++ compiler, the GNU Compiler Collection [10] is an extensive open source project. As popular as GCC is, it does not provide the features an analysis-tool-building developer needs.

For the sake of building such tools, a compiler front end is used. Due to an old design, it is difficult to work with either the front end or the back end of GCC alone. Besides, the compiler implicitly makes optimizations that destroy any parallels between the source code and the AST. Therefore, the AST has to be treated as an entirely different object rather than an abstraction of the code. Most of the compiler's source code representation is unintuitive and hard to pick up for anyone not actively contributing to GCC. Figure 2.1 showcases the unfriendliness rather well. Compared to figure 1.1, which is an output of a tool built using LLVM and Clang, GCC's mapping between the source code and the internal representation does not hold up.

As far as AST manipulation is concerned, the compiler allows the user to dump the structure into a text representation. However, due to the difficulties mentioned above, it can hardly be used.

These issues result in a seldom-used variant that offers nearly no developer-friendly features. An upside is that GCC allows the user to visualize the AST. However, that is hardly a useful feature in the context of this paper.

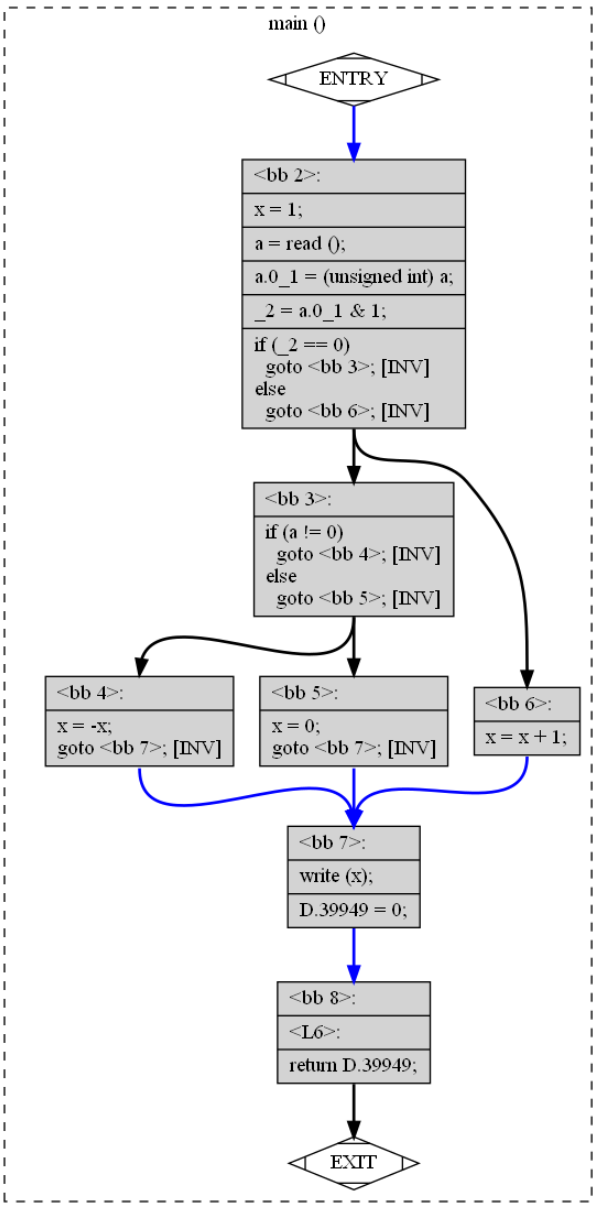

Figure 2.1: GCC AST Dump. This figure showcases the AST representation of 1.1 as dumped by GCC. Note that it is not easily comprehendable.

2.2 Clang

Thanks to LLVM [11], the widespread compiler infrastructure, the Clang project [12] has provided a compiler front end not only for C and C++ but also for CUDA, OpenCL, and other languages. The extend of Clang as a compiler front end is so vast that it covers both the C++ standard and the unofficial GNU++ dialect.

The project does not include just the front end but also a static analyzer and several code analysis tools, which are now commonly used in IDE's as syntax and semantic checks.

This description of Clang foreshadows its friendliness to analysis tool developers. The fact that the front end runs on a common intermediate language also indicates that openly working with abstract code representations is supported.

There are three most notable interfaces for customizing Clang. Firstly, the LibClang interface allows the users to write comprehend-able high-level code with limited functionality. On the other hand, LibTooling gives the user much more control at the cost of a steep learning curve. Lastly, the Plugins interface features similar difficulty as LibTooling with a more specific goal. Plugins are used with the Clang compiler and can be run as a front-end action when called during compilation.

2.3 ANTLR

A less typical way of extracting an AST from a source file is by using grammar recognition. ANTLR [13], which stands for Another Tool for Language Recognition, is a free parser generator that generates both a lexer and a parser based on a given grammar. Additionally, ANTLR can also generate a tree parser. Tree parsers are helpful in processing ASTs.

The tool is generally used to read data formats, process expressions of various query languages, and even parse programming languages. It can be used to generate a syntax tree and walk through it using a visitor. ANTLR is based on the LL parser, which parses the input from left to right, performing its leftmost derivation.

To create a parser or a syntax tree of a programming language, ANTLR requires the complete grammar of that language. Some programming languages, namely C and C++, have an ambiguous syntax that is hard to parse based solely on its grammar. Due to ANTLR's high popularity, many grammars have already been written for it. As far as C++ is concerned, its C++14 standard's grammar is the most recent one available.

Writing grammar for newer standards or creating a custom one for both C and C++ would be unnecessarily burdensome for this project. This statement holds, especially when considering other tools mentioned above.

The most recent release, ANTLR 4, added more options for grammar rules. Most notably, it supports direct left recursion. However, that still might not be enough to choose it over other tools.

2.4 DMS

Similar to ANTLR, the DMS Software Reengineering Toolkit [14] features a parser generator. The tool is proprietary software created by Semantic Designs. Besides the mentioned parser generator, it features an entire toolkit for creating custom software analysis. This toolkit is used mainly for reliable refactoring, duplicate code detection, and language migration.

The parser generator part takes a grammar and produces a parser. This parser then constructs abstract syntax trees for provided source code. Additionally, created ASTs can be converted back to source code using prettyprinters. The parser saves additional information about provided source files, such as comments and formatting. It can then recreate the file accurately.

DMS provides a grammar for a large number of languages, including C and C++. The language support, however, is not always up-to-date. The newest supported C++ standard is still the older C++17. These complicated grammars' ambiguity is avoided using a generalized left-to-right parser, which performs the rightmost derivation (GLR). Since DMS provides refactoring ability as well, it allows for transformation rules in the grammar.

Another helpful feature of the toolkit is control flow and data flow analysis. Analyzing control flow and data flow, generating their graphs, and performing the points-to analysis (also supported by DMS) is practical when considering static slicing (section 1.2).

It should be noted that some of the free, open-source tools mentioned above do a better job of being a so-called 'software analysis toolkit' than DMS does.

2.5 Summary

The chapter highlighted a spectrum of tools, ranging from language recognizers to compilers.

It would seem that parsing multiple programming languages into an abstract representation requires a common intermediate language, in which the representation is stored. Having an intermediate language is not always possible for several reasons, including licensing and old architecture. The compiler giant GCC seems to suffer from precisely that. Additionally, since the Clang project is being contributed to regularly, resulting in as many as five releases per year, it pulls in a more significant developer community.

Therefore, Clang is the favorite source code altering tool for this project. In the following chapter, the relevant parts of the Clang project will be broken down and explained.

3. Clang LibTooling

The previous chapter described tools and environments that were taken into consideration for this project. The utmost importance was given to the ease of use, availability, and active community. As the reader might have guessed from the summary, the LLVM/Clang suite stood out as the best candidate.

Clang is a language front-end. With high compilation performance, low memory footprint, and modifiable code base, it quickly and flexibly converts source code to LLVM intermediate code representation. The front-end supports languages and frameworks such as C/C++, Objective C/C++, CUDA, OpenCL, OpenMP, RenderScript, and HIP. This support is crucial for this thesis since the project aims to support both C and C++. The LLVM Core then handles the optimization and IR synthesis, supporting a plethora of popular CPUs.

Clang is widely used for its warnings and error checks, both very helpful and outstanding compared to competing compilers. Furthermore, Clang offers an extensive tooling infrastructure through which tools such as clang-tidy were developed. A relatively well-documented tooling API written in C++ helps programmers create their tools easily. However, not all developers share the same skill set. Some programmers require complicated additional features, while others prefer an easy-to-use interface. The tooling API has been split into multiple libraries and frameworks.

For plugin development, a library intuitively called Plugins is used. The library is linked dynamically, resulting in relatively small tools. Plugins are launched at compilation and offer compilation control as well as access to the AST.

More specifically, Plugins allow performing an extra custom front-end action during compilation. The functionality is generally similar to that of LibTooling, which will be talked about later. However, unlike a standalone tool, Plugins cannot do any tasks before and after the analysis (and compilation). When creating a plugin, one can choose from a selection of `FrontendAction` classes to inherit. If, for example, the plugin should work with the AST, the `ASTFrontendAction` can be inherited. Doing so also allows overriding the `ParseArgs` method, in which the plugin's command line handling is specified.

Due to dynamic loading, the wanted plugin must be added to a plugin registry inside the code. The plugin is then loaded from the registry by specifying the `-load` command or `-fplugin` on the command line when running clang. The plugin takes those arguments from the command line that are prefixed by `-Xclang`.

Another framework, LibClang, offers a simple C and Python API for quick tool writing. Unlike Plugins and LibTooling, which will be mentioned later, the code base of LibClang is stable. This stability implies that tools written using LibClang do not require upkeep with every new LLVM/Clang release. Overall, the framework and tools written using it are high-level and are easily readable.

The most feature woven set of libraries is LibTooling. Unlike Plugins, LibTooling [15] allows the developer to build standalone Clang tools. This robust framework is written in C++ and has an active community of contributors. One can find many manuals and tutorials online. However, with each contribution to LibTooling and each release of Clang, there is a chance that older tools will not

support the newer LibTooling API. That is the reason why countless tools written using this framework do not run in modern environments. Programmers who use LibTooling cannot expect compatibility in upcoming releases. On the bright side, the libraries of LibTooling allow a plethora of source code modifications, AST traversals, and access to the compiler's internals.

The set of features supplied by LibTooling is immense. The following sections describe notable features used during the implementation of this project. The reader should get a better idea of how a tool is built and what LibTooling offers during the development process. Important concepts, such as providing the correct input to the tool in the form of a compilation database, traversing the AST, and modifying source code inside the tooling environment, are described below. These concepts will be referenced further in the text.

3.1 Compilation databases

To accurately and faithfully recreate a compilation, tools created using LibTooling require a compilation database (CD) [16] for a given input project.

The motivation behind a CD is simple. If a source file uses unusual include paths that need to be provided using the `-I` compiler command, it cannot be reliably compiled. Similarly, if the file contains macros and lacks definitions, its content can drastically change when the definitions are present. In the latter case, definitions are provided to the compiler with the `-D` command. Such compiler commands, options, and flags are usually defined in a build system. At least, that is the recommended practice for larger projects. Having a build system is similar to having a CD. It is clear which file is compiled with which options.

Clang expects a CD in the JSON format and looks for the file specifically named `compile_commands.json` in the current or parent directories. The JSON file contains entries for source files. Each entry contains a directory, a file name, and a compilation command. Multiple entries for a single source file are also valid. Such a case can arise when performing repeated compilation.

As previously mentioned, having a build system helps. Build tools such as CMake and Ninja can be used to generate a CD. If the project is not using any of the compatible build tools, the user can either make a CD manually or use an external tool. One such tool is Build EAR available at `https://github.com/rizsotto/Bear`.

Tools created using LibTooling do not always require compilation databases to run. For simple projects, they can take the `--` argument that separates the tool's arguments from the project's compilation arguments. One can interpret the arguments following `--` as a temporary compilation database.

3.2 Clang AST

The abstract syntax tree used in the Clang front-end [17] is different from the typical AST. It saves and carries more data, namely context. For example, it contains additional information to map source code to nodes and capture semantics.

Its nodes belong to a vast class hierarchy. This hierarchy contains classes that represent every supported source code construct. Nodes are of four different types:

statements (`Stmt`), declarations (`Decl`), declaration context (`DeclContext`), and types (`Type`). However, in the APIs mentioned above, the nodes do not share a common ancestor.

The children of `Type` represent all available types. The goal is to give each type in the source code a canonical type, i.e., a type stripped of any typedef names. Canonical types are used for type comparison, while non-canonical types give complete information during diagnostics. The `Decl` hierarchy's goal is to have a class for each type of declaration or definition. These declarations vary, and the children cover specific cases such as function, structure, and enum declarations. Some declarations, such as function and namespace declarations, capture additional data in `DeclContext`'s children. The final node type, `Stmt`, represents a single statement. It has subclasses for loops, control statements, compound statements, and more. Additionally, expressions (`Expr`) also belong to the `Stmt` hierarchy.

Figure 3.1 shows a part of the class hierarchy. The entire class diagram cannot be shown as there are over a thousand different classes[1]. The topmost node, the root, of a concrete Clang AST is called the translation unit declaration (`TranslationUnitDecl`). Edges between nodes are simplified, as each node stores a container of its children.

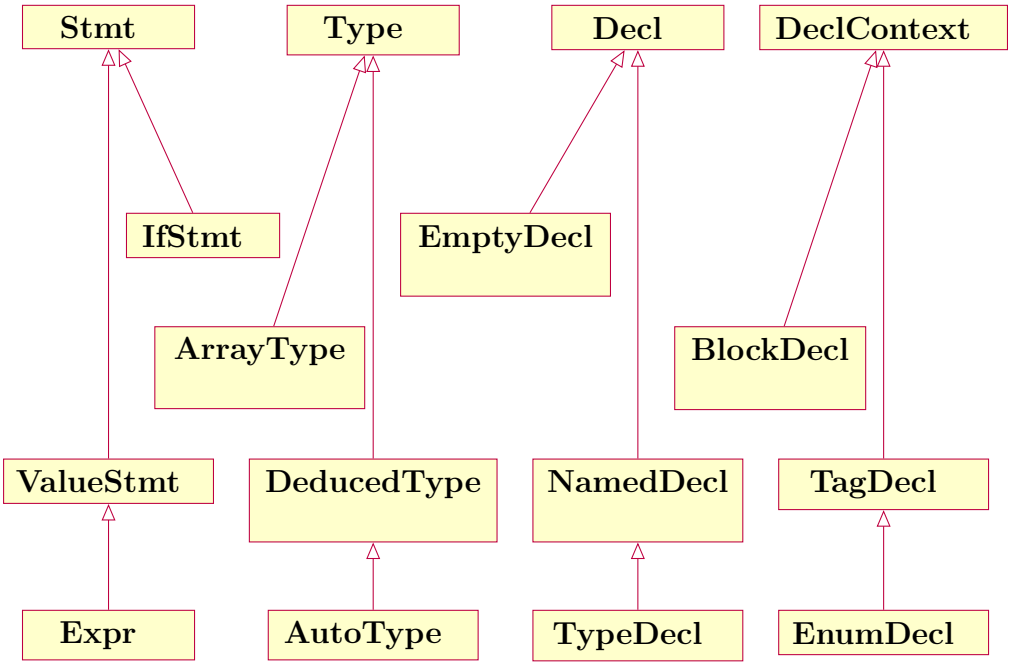

Figure 3.1: An example of the Clang AST class hierarchy. The figure contains only a handful of classes and their children. Note that the top most classes do not share a common ancestor.

Listing 3.1 contains a short program written in C++. The source code was provided to a Clang tool clang-check, which dumped the abstract syntax subtree of a given function. In this case, the filter was set to the `main` function. The AST dump visualizes the subtree using ASCII characters and node information. Nodes entries start with their type names. Each node also carries its

[1]The class hierarchy is shown in Clang's Doxygen documentation. An example of the Stmt hierarchy can be found at `https://clang.llvm.org/doxygen/classclang_1_1Stmt.html`.

address, source location, and description. Note that the root of the subtree is of type `FunctionDecl`. The usual root `TranslationUnitDecl` is absent due to the function filter being applied.

Listing 3.1: Clang AST Dump.

```
$ cat -n simple.cpp
    1 #include<iostream>
    2
    3 int main()
    4 {
    5     int x;
    6     std::cin >> x;
    7
    8     return (x / 42);
    9 }
$ clang-check -ast-dump -ast-dump-filter=main simple.cpp --
Dumping main:
FunctionDecl '...' </simple...> line:3:5 main 'int ()'
`-CompoundStmt 0x556041ab84a0 <line:4:1, line:9:1>
  |-DeclStmt 0x556041ab6900 <line:5:2, col:7>
  | `-VarDecl 0x556041ab6898 <col:2, col:6> col:6 used x 'int'
  |-CXXOperatorCallExpr '...' <line:6:2, col:14> 'std::bas...'
  | |-ImplicitCastExpr 0x556041ab83a0 <col:11> 'std::basic...'
  | | `-DeclRefExpr 0x556041ab8318 <col:11> 'std::basic...'
  | |-DeclRefExpr 0x556041ab6980 <col:2, col:7> 'std::istr...'
  | `-DeclRefExpr '...' <col:14> 'int' lvalue Var '...' 'x'
  `-ReturnStmt 0x556041ab8490 <line:8:2, col:16>
    `-ParenExpr 0x556041ab8470 <col:9, col:16> 'int'
      `-BinaryOperator '...' <col:10, col:14> 'int' '/'
        |-ImplicitCastExpr '...' <col:10> 'int' <LValueToRValue>
        | `-DeclRefExpr 0x556041ab83f8 <col:10> 'int...'
        `-IntegerLiteral 0x556041ab8418 <col:14> 'int' 42
```

The Clang AST attempts to represent the source code as faithfully as possible. It can be said that Clang's AST is closer to C, C++, and Objective-C code and grammar than other ASTs. To achieve the best accuracy in reproducing a source code file, it must save additional data besides the AST. This supplementary data makes information that would be lost otherwise, such as compile-time constants, available in the unreduced form.

For each parsed source code file, an instance of ASTContext is used to represent the AST. The ASTContext allows the programmer to use many valuable methods.

TODO: Show some of those functions from the documentation.

The ASTContext bundles Clang's AST for a translation unit and allows its traversal from the `getTranslationUnitDecl` point, which is the file's highest node. Additionally, the context has access to the identifier table and the source manager. The `SourceManager` class offloads some of the data from AST's nodes. Nodes store their `SourceLocation`. The location is not in its complete form since it is required to be small in size. Instead, the node's full location is referenced in

`SourceManager`.

Extracting Clang AST comes at the cost of compiling the program's source code. Usually, this is done using an instance of `FrontEndAction`, which specifies what and how should be compiled. The front-end compilation is essential to note because it can affect LibTooling's performance on large projects. In comparison, clang-format does not execute any compilations. Therefore, clang-format runs efficiently on large projects and correctly on incomplete ones. The compilation action also implies that LibTooling tools often do not support incomplete source codes. The same can be said for programs that contain compile-time errors.

An additional characteristic of Clang's AST is its immutability. The AST has strong invariants that might be broken upon changing its structure. Generally, changes to the Clang AST are strongly discouraged, although some changes happen internally. Those changes include template instantiation.

Traversing the Clang AST is possible through two different APIs. First, it is possible to invoke an `ASTFrontendAction`, which creates an `ASTConsumer`, who then calls the `ASTRecursiveVisitor`. The front-end action is invoked upon parsing a source file. The action can be overridden to create a consumer and pass any necessary data to it. For example, this data might include references to variables used for counting objects in the AST or more complicated constructs.

The `ASTConsumer`'s job is to read the Clang AST and handle actions on the tree's specific items. One such action is `HandleTopLevelDecl()`, which, as the name suggests, handles the highest priority declaration in a file. These handle functions are overridable. The consumer also keeps track of a visitor implemented by inhering from the ASTRecursiveVisitor class. The consumer dispatches the visitor from overridden handle methods. However, it is not always beneficial to override granular handle methods. Handling specific events in the consumer might lead to an intriguing case in which a part of the code is parsed while the rest is not. This unwanted behavior can be avoided by overriding just the `HandleTranslationUnit()` method. The translation unit is handled once the entire source file is parsed. Dispatching the visitor internally from a consumer is the preferred approach. Visit methods of the `ASTRecursiveVisitor` should not be called directly. Details concerning the visitor can be found in the following section.

Second, one can use AST Matchers. Matchers, unlike the visitor approach, do not require a complicated setup. Instead, they provide a query-like syntax for matching Clangs AST's nodes. Matchers will be talked about in detail later.

3.3 ASTVisitor

LibTooling offers a built-in curiously recurring template pattern (CRTP) visitor. The class `RecursiveASTVisitor` [18] offers `Visit` methods that can be overridden to the programmer's liking. Each override specifies the type of node on which the method triggers and the actions that should be performed.

The implementation seen on listing 3.2 illustrates the idea. a custom class with a strict dedication, i.e., counting program's statements, has two visit functions. Firstly, a `VisitStmt` method, which is triggered upon encountering a node of type `Stmt`, as seen in its parameters. Furthermore, since no additional visit functions for children of `Stmt` have been overridden, `VisitStmt` will trigger on every node

type inheriting from `Stmt` as well. Secondly, the method `VisitVarDecl` only accepts `VarDecl` and its inheriting types. Because `VarDecl` is a child of `Decl`, not the other way around, `Decl` will not trigger this visit function. Typically, when using less specific visit methods, a good way of differentiating node types is casting them dynamically.

Listing 3.2: CountASTVisitor.

```
1  /**
2   * Counts the number of statements.
3   */
4  class CountASTVisitor : public
       clang::RecursiveASTVisitor<CountASTVisitor>
5  {
6      clang::ASTContext* astContext;
7      int statementCount;
8
9  public:
10     explicit CountASTVisitor(clang::CompilerInstance* ci)
11         : astContext(&ci->getASTContext()),
       statementCount(0) { }
12
13     virtual bool VisitStmt(clang::Stmt* st)
14     {
15         outs() << "Found a statement.\n";
16         statementCount++;
17
18         return true;
19     }
20
21     virtual bool VisitVarDecl(clang::VarDecl* decl)
22     {
23         outs() << "Found a variable declaration.\n";
24
25         return true;
26     }
27 };
```

Visiting statements, expressions, declarations, and types is straightforward. The same applies to children of these classes. However, it is challenging to visit more complicated entities such as nested types, e.g., `int* const* x`. Such cases require fetching additional semantical context, utilizing `ASTMatchers` and nodes of type declaration context.

TODO: Show how complicated cases are handled.

The `RecursiveASTVisitor` is launched by visiting the root node using a `TraverseDecl` method. It then dispatches to other nodes and their children. For each node, the visitor searches the class hierarchy from the node's dynamic type up. Once the type is determined, the visitor calls the appropriate overridden

`Visit` method. Traversing the class hierarchy from the bottom up translates to calling specific visit functions for specific types rather than visit functions of their abstract types.

The tree traversal can be done in a preorder or postorder fashion. Preorder traversal is the default. the developer can also stop the traversal at any point by returning `false` from the visit function as opposed to `true`.

3.4 Matchers

Clang's ASTMatchers [19] is a domain-specific language (DSL) used for querying specified AST nodes. Each matcher represents a predicate on nodes. Together, they form a query-like expression that matches particular nodes. Like the rest of LibTooling, the DSL is written in C++ and is used from C++ as well. Matchers are useful for query tools and code transformations. In a query tool, one might want to extract a niche subset of Clang's AST, inspect it, and perhaps perform some action on it. Similarly, refactoring tools can use matchers to navigate and extract similar nodes, rewrite their source code, or add descriptive comments. A matcher will match on some adequate node. It might match multiple times if the AST has enough of these nodes. When combined, multiple matchers form a matcher expression. Such expression can be seen as a query for the Clang AST. The expression reads like an English sentence, from left to right, alternating several type-specifying and node-narrowing matchers.

All available matchers fall into three basic categories. The first one being node matchers. Node matcher's job is to match a specific type of AST node. An example of such a matcher could be the `binaryOperator(...)` matcher, whose purpose is to look for nodes of that exact type: `BinaryOperator`. Node matchers are the core of matcher expressions. Expressions start with them, and they specify which node type is expected. Node matchers also serve as arguments for other matcher types. Furthermore, they allow binding nodes. Binding nodes allows the programmer to retrieve matched nodes later and use them for code transformation tasks.

The second category, called narrowing matchers, serves a different purpose. By matching specific attributes on the current AST node, they narrow down the search range. Narrowing matchers allow specifying more granular demands for the searched node. A concrete example would be the `hasOperatorName("+")` matcher. As one might guess, this matcher narrows down the search to those nodes whose binary operator is the plus sign. Narrowing matchers also provide more general logical matchers. These include `allOf`, `anyOf`, `anything`, and `unless`.

The last category specifies the relationship between nodes. Traversal matchers are used for filtering reachable nodes based on the AST's structure. Most notably, they include matchers for specifying node's children, such as `has`, `hasDescendant`, and `forEachDescendant`. Traversal matchers take node matchers, the first category, as arguments. For example, the `hasLHS(integerLiteral(equals(0)))` matcher specifies the requirement for the current node to have the given child. In this case, it is an integer with the value 0 on the left hand side.

Together, these three examples form a matcher expression found in the AST Matcher tutorial [20]. Going by the mentioned rules of building an expression, it

would have the following form:

Listing 3.3: Matcher expression.

```
binaryOperator(hasOperatorName("+"),
    hasLHS(integerLiteral(equals(0)))).
```

The expression in 3.3 searches for a binary operator. The search is further narrowed to a plus sign with a zero left-hand side of the operation.

In the tool, expressions are build by calling a creator function. The expression is then represented as a tree of matchers. While the developer has access to a plethora of predefined matchers, as seen in the Matchers Reference [21], they can define custom ones as well. Creating a custom matcher can be done in two ways. First, a matcher can be created by inheriting an existing Matcher class and overriding it to one's liking. Second, one can use a matcher creation macro. These macros specify the type, the name, and the parameters of the matcher.

The default behavior, defined by the `AsIs` mode, is to traverse the entire AST and visit all nodes, including implicit ones. Implicit nodes might include constructs omitted in the source code, such as parentheses. Working with these nodes increases the difficulty of writing matcher expressions severely since it requires a deep knowledge of the AST's hierarchy and its corner-cases. The traversal mode can, however, be changed to ignore implicit nodes. One such traversal mode is `IgnoreUnlessSpelledInSource`, which conveniently only looks at nodes represented by the source code.

3.5 Source-to-source transformation

To transform source code based on its AST, the programmer must extract the AST from the code, alter the AST, and then translate it back to valid source code. LibTooling allows the programmer to extract the AST and examine it. Additional functionality also allows modifying the AST both directly and indirectly [22]. However, there are obstacles and limitations to both approaches.

Let us examine the pitfalls of direct AST transformation first. Before explaining the possibilities of direct modifications, it should be noted that these transformations are not recommended. Clang has powerful invariants about its AST, and changes might break them. Although it is not encouraged, the methods to change the AST are available.

Given an `ASTContext`, it is possible to create specific nodes using their `Create` method. Likewise, nodes with public constructors and destructors can combine keywords `placement new`, `delete` and the ASTContext to add or remove nodes. The job of `ASTContext` is then to manage the memory internally.

A more sophisticated approach is the one offered by the `TreeTransform` class. Although it is rarely used and no real examples can be found, the premise is simple. The `TreeTransform` class needs to be inherited from, and its `Rebuild` methods need to be overridden. The overrides then transform specified nodes of an input AST into a modified AST.

One additional dirty way of replacing nodes is by utilizing `std::replace`. The child container of the replaced node's immediate parent must be specified in parameters of `std::replace`, together with the node itself and the new node.

When attempting to modify the AST indirectly, which is how LibTooling intends it to, the developer can run into a couple of issues. First of all, the AST does not reference the source code entirely. The programmer has access to `SourceManager`, `Lexer`, `Rewriter`, and `Replacement` classes. When used individually or in combinations, they can map to and alter a given node's source code. It is then possible to add, remove, or replace the AST's underlying code with node-level precision.

Accessing this information through these classes can result in node-to-code mapping issues. Compound statements might mismatch parentheses and curly brackets. Similarly, declarations and statements might miss a reference to a semicolon. These and more obstacles could surface anytime a programmer attempts to debug their source-to-source transformation tool.

The programmer must be careful in managing object instances when transforming multiple files. Each source file creates a new FrontendAction, and with it, the developer needs a new instance of Rewriter.

While LibTooling intends most of the issues mentioned earlier, they are not as quickly comprehendible as the rest of the framework. Templates, the language feature of C++, further complicate the matter. In Clang AST, multiple types derived from a template might share some nodes. Having multiple parent nodes is also not uncommon for template types. Thankfully, templates are rarely used. A more common threat, macros, has a similar effect. Modifying a source code containing macros and comments results in losing both.

Doing source-to-source transformation is often accompanied by inserting instrumentation code. By performing so-called cross-checking, one can make sure that the transformation behaves as intended. Cross-checking works by inserting code with the same behavior into the original and the transformed source code. This insertion can be done in a sophisticated manner using the AST. If, for example, the transformation alters calls to functions in the code, the instrumentation code should be inserted inside the function's body—that way, the developer can check whether the transformation had its intended result. Cross-checking is a safe way of ensuring source-to-source transformations work as intended. While they might be excessive for small refactorings, they are beneficial when debugging source-to-source transformations of larger scales, such as translating one language's source code to another's.

4. Program minimization

As described in the first chapter, debugging is a time-consuming task. Any amount of help with debugging is always appreciated by developers. This paper attempts to help by providing means to minimize the debugged program for a given runtime error. The minimization's goal is to reduce the amount of source code programmers must go through when debugging, thus speeding up the process. The size reduction of the program should be fully automated and reasonably fast on simple inputs. Furthermore, it should correctly handle any source code from the program domain specified below. Great attention is given to the accuracy with which the minimizing algorithm works and its running performance.

TODO: Change the domain based on the implementation (e.g., UI applications might work, so extend to UI, same with threads).

The domain in which the algorithm operates can be described as more minor, simple projects. The approach takes into consideration code written in C and C++. More complicated concepts such as templates are omitted. Additionally, projects with non-deterministic features as such multithreading are also not taken into consideration. Instead, the program minimization described in this paper focuses on simple single-threaded console applications.

The problem of program minimization while preserving runtime errors can be described as follows. A developer has encountered a runtime error in his application. Using logging or debugging tools, he can extract the stack trace at that given point. The stack trace provides valuable information the algorithm takes into consideration during its execution. It notably requires a description of the error and the source code location at which the error was produced. Based on the described scenario, we can draw the following definitions.

Def: Let loc: S -¿ (file, line, col). We call the result of loc the location of statement S.

The source code's location is specified by a file name, the line number, and on that line, the number of characters from the left. The location could be described in further detail by including starting and ending points. However, in this simplified description, only the starting point is taken into consideration.

Def: Let E = (location, desc) be a runtime error specified by its location and description thrown by the program P = S1, S2, ..., Sn. We call the Si the failure-inducing statement of E iff loc(S) = E(location).

Failure-inducing statements are constructs in the code that directly contributed to the thrown error. That means the statements were present at the error's location when the error occurred.

Having found the source code location, the developer can now investigate the source code for a potential bug. In the process, he might consider the values of application arguments present at launch-time and change his debugging process accordingly. Nonetheless, the developer has to look through the source code to find the error's root cause. This exact point is where the source code size reduction comes into action. Using static and dynamic analysis, it is possible to effectively and safely minimize unnecessary source code. Such code includes statements, declarations, and expressions that do not affect the program's state at the point given by the error. With additional verification, it is also probable to remove code constructs that affect the state, but the error occurs regardless of

whether they are present or not.

Using code size reduction, one can look at the reduced program's source code to find the error. The newly generated program has to fulfill the following invariant.

Invariant: Every program P' created by reducing the original program P based on dynamic information given by P's execution with arguments A must result in the same runtime error E. The error's absolute location can differ; it must, however, occur in the same context.

The rule specifies that a program must end in the same runtime error as the original program to be considered a correct reduced program. Though, with the change in the program's size, the location of failure-inducing statements also changes. In P, the error's location should be further down compared to P' since P' has less code in general. Stress is placed on the location's context in which the error arises. As long as locations in P' are adjusted based on those in P, the location of the error does not matter.

TODO: Come up with an actual way to make sure a program is minimal.

TODO: Come up with an approximation to guess whether the program will terminate.

As far as minimality goes, there is no actual conclusion on recognizing whether a solution is minimal. Similarly, it has not been determined how to recognize programs that can run indefinitely.

Minimization of programs requires two steps—first, the removal of chunks of the given source code. Second, performing a validation to determine whether the result meets the required criteria, i.e., minimality and correctness. Following approaches for both steps have been considered.

4.1 Naive reduction

4.1.1 Description

The simplest approach examined in this project is the naive removal of each source code statement. This technique aims to try every possible variation of the code and find the smallest correct solution through trial and error. All possible variations, both valid and invalid at compile-time, can be generated by separating the source code into units of statements, declarations, and expressions and removing one code unit at a time.

Once the input source code is provided, it is split into n of these code units. Every unit is then taken into consideration, removing it and keeping every other unit in the code. This process results in n new variants, each complementing their respective code unit. These outputs are then fed back as input code and processed the same way one by one.

TODO: Add a definition for code units.

TODO: Show pseudocode.

Each iteration with the input size of n code units results in n new variants, those contribute to $n * (n - 1)$ new results. The naive time complexity is, therefore, the abysmal $O(n!)$. Moreover, the $n!$ variants require some verification and classification to determine whether they are minimal or not. The correctness of many of these variants can be determined at compile-time. The rest, however, must be executed and tested for runtime errors.

An efficient way of finding the smallest correct program variant is to rule out programs with compile-time errors, sort the remaining variants by size and verify them from the smallest to the largest. Depending on the input program's execution time, the verification might take more time than the variant generation step.

4.1.2 Takeaways

TODO: Draw parallels to the traveling salesman problem and reduce the complexity to exponential.

The algorithm can be sped up by using the following techniques. The input's size can be significantly reduced by performing static slicing of the input program as the first step. Compared to the naive approach, static slicing is a significantly less expensive operation. Furthermore, it does not require the program to run. With two main issues concerning the current discussed approach - the size of the input and its execution time - static slicing manages to ignore one of these factors. Both the input size and subsequent execution time could both be brought down by using dynamic slicing.

The usage of dynamic slices, as opposed to static, has its potential pros. On the other hand, it has definitive cons. One such con is the requirement to run the said program. This point will be discussed in more detail later; however, let us consider static slicing for this approach to minimize the number of program executions. Since static slicing does not handle branching and other control statements nearly as efficiently as dynamic slicing, we can employ a simple trick to help. Using the same additional input information as dynamic slicing, i.e., program arguments, we can provide more specific information to the static slicing algorithm. All that is required is to define the arguments with their respective values inside the code. Slices generated from this modified source code will be more precise since they will not contain unnecessary branching.

It is important to restate that this modification only affects control statements dependent on the program's arguments. If the arguments are not in the original, unmodified static slice, their values will not affect the slice's size. Such modified input is guaranteed to be smaller or equal in size. Removing only a single code unit, i.e., a statement, an expression, or a declaration, and leaving its complement sometimes generates an invalid and unnecessary program variant. This problem can arise when removing a variable declaration. All subsequent source code using this variable will be invalid. Therefore, the removal of some code units should also remove their potential usage. This case does not only concern the mentioned variable declarations but function declaration, function definitions, and structure, enum, and class declarations as well.

These proposed modifications can potentially reduce the output of each iteration. However, lower runtime complexity is not guaranteed. Static slicing may help in programs whose intent is to perform multiple independent tasks, but it might not contribute anything to the complexity decrease otherwise. The removal of dependent constructs described in the second point should be used regardless of the input's nature, though it still does not ensure the algorithms' quicker running time. The running time will especially be left unchanged in programs that do not employ structured or object-oriented programming.

4.2 Delta debugging

Zeller's Delta debugging [5, 23, 6] has been described in detail in section 1.1. Although it serves primarily to find failure-inducing parts of inputs when run on test cases, it can tremendously help with program minimization. For a given input, Delta debugging attempts to find and isolate its smallest failure-inducing subset. Other than the input, Delta debugging also requires the debugged program (in its executable form) and expected output.

Let us draw parallels between the mentioned requirements and this project's minimization task. The input for program minimization is the given program's source code. The code can be labeled as the input Delta debugging takes. It is essential to clarify that this Delta debugging usage does not utilize the source code as the debugged program. Instead, it considers it as a given input. Then, we must specify the expected output. It is required that the program terminates with a given runtime error. Let us label that runtime error and its location as the expected output. Lastly, we must set the debugged program. In our case, to get from the test's input (source code) to the expected output (a specific runtime error), the input must first be compiled and then executed. The fitting debugged program is, therefore, a pipeline of a compiler and an execution environment.

The result is what we need - a minimal program variant that fails with a particular error. As was already mentioned, the location of the error might differ based on the variant's structure. Nonetheless, the location could be aligned based on the source code in an additional step.

TODO: Take the following paragraph (equal sizes) with a grain of salt. Think about the implementation of DD for this project. Is it better to only consider leaf nodes or always look at full code constructs (functions, classes, ...)?

The minimizing Delta debugging algorithm works iteratively. Initially, during each iteration, the input is split into n code units of equal size. Equal sizes do not fit the nature of our input very well. A better solution would be to split the source code into n or fewer units based on the code constructs present in the code. For example, there is no point in splitting a function definition in half. Instead, it would make more sense to leave its head and body as a single code unit. The body could then be divided more gradually in further iterations. Each iteration runs a test on each of the n code units and its complements, resulting in $n * 2$ tests run per iteration. In our case, the test consists of compiling and executing a snippet of source code.

The requirement to run this many executions significantly hinders the performance of the algorithm. Delta debugging can also be done in another manner. The isolating algorithm can make an overall better use of iterations. When minimizing using Delta debugging, the result of each iteration only changes when the test fails. On the other hand, the isolating approach also contributes changes to the result when the test passes. However, passing cases are sporadic when working with source code. They only arise when every executed code unit ends in the same desired result. It is unlikely that those code units would compile on their own since they would probably represent only a snippet of code, not a stand-alone subprogram. Furthermore, it is even less probable that the code units would result in the same runtime error when executed.

TODO: Come up with a way to guess the time complexity of DD.

TODO: Verify that the following is true! Make sure to understand the DD approach correctly and its verification (compilation and execution of snippets).

4.3 Slicing-based solution

As mentioned in the naive approach, both static and dynamic slicing need to be analyzed further. The main focus of the discussion should be the running time. It is known that dynamic slices are the smallest they can be. However, they require information available at execution time. The question is whether running the program is necessary.

We know that the program that is being minimized has been run before. Hence the availability of the information about the encountered runtime error. If the program ran deterministically, it would have to terminate in future executions as well. That is considering it would run with the same arguments as previously. The time of the termination might vary depending on the purpose of the program. For server-like applications, it might take months to encounter an error at runtime. Static slicing does not suffer from the mentioned issue. It is inexpensive in terms of execution time regardless of the purpose of the sliced program. With the arguments trick mentioned in section 4.1, static slices can be as small as dynamic ones. Minimality of the slice is, however, not guaranteed.

It is safer to employ dynamic slicing to get the smallest slices possible. However, one can make modifications to help dynamic slicing run more effectively. Let us consider a program that performs multiple demanding tasks such as computations. These tasks are primarily independent, and their runtime is long. Using dynamic slicing alone would be inconvenient. However, by first employing static slicing to remove these long-running unnecessary tasks, the program's execution time can be significantly reduced. The reduced program could then be sliced dynamically. The result would be a minimal slice at a fraction of the original time compared to dynamic slicing alone.

This crafted ideal use case only concerns a very narrow range of existing programs. However, due to its low running time, static slicing could be used before just about any attempt at dynamic slicing.

TODO: Add references to the halting problem and Rice's theorem.

The improvement in the form of a static slice is genuinely convenient. However, checking whether the improvement has any effect before running dynamic slicing is not an easy task. The issue stems from the Halting problem and Rice's theorem. The halting problem states that it is undecidable whether a program terminates on its particular input. Rice expanded the thought further by stating that all interesting semantic properties of a program are undecidable. Without proper and accurate means to determine many wanted properties, we are required to approximate them.

Amongst such properties is the factor of how effective static slicing is. The approximation will be required in the following sections as well. In particular, the section concerning program validation will look at this issue in more detail. One way of guessing the effectiveness of static slicing in terms of size reduction is by analyzing the program's branching factor. By employing a metric for the number and density of control-flow altering statements, we can approximate static slicing's relative performance. It is assumed that programs with a high branching factor, i.e., with more control-flow-altering statements, are less likely to reduce their size during static slicing. Nonetheless, slicing statically before doing so dynamically has been a rule of thumb for this project.

The proposed systematical solution is described in the following algorithm. The input program is sliced statically w.r.t. every variable available at the failure-inducing line. The slices are then unified and given as the input to a dynamic slicer. Similarly, the dynamic slicer generates slices w.r.t. those potentially failure-inducing variables. Those dynamic slices are unified. The intermediate result extracted after performing the two slicing types should be significantly smaller than the original program. It is then fed to a more precise and less efficient algorithm for further reduction. Since the result so far contains slices for multiple variables, it might not be minimal yet. However, it can be assumed that it is valid, i.e., ends with the desired runtime error. We could implement the naive approach or Delta debugging as the more precise algorithm.

TODO: Add pseudo code.

TODO: If a more sophisticated final algorithm is created, mention it.

Another thought-about approach is hybrid slicing. The comparison of hybrid slicing and the combination of static and dynamic could yield exciting results. It can be assumed that hybrid slicing would be more effective on smaller programs with a short execution time. The static-dynamic combination could work better on larger-scale applications, where static slicing can remove unnecessarily long-running chunks of code.

TODO: Get more information on hybrid slicing.

4.4 Program verification

TODO: Add https://youtu.be/UcxF6CVueDM?t=177 as a reference.

TODO: Meditate at the thought of using slicing for systematic validation. For example, a slice of the original program and the reduced program's slice must not differ in some parts.

One way of achieving systematic validation is by inserting instrumentation code at compile-time.

TODO: Verify this statement and explain further.

Conclusion

Bibliography

[1] SpotBugs. https://spotbugs.github.io/index.html. [Online; accessed 14-March-2021].

[2] Clang Static Analyzer. https://clang-analyzer.llvm.org/. [Online; accessed 14-March-2021].

[3] CodeChecker. https://codechecker.readthedocs.io/en/latest/. [Online; accessed 14-March-2021].

[4] Infer Static Analyzer: Infer: Infer. https://fbinfer.com/. [Online; accessed 14-March-2021].

[5] A. Zeller. Yesterday, my program worked. Today, it does not. Why? *LNCS*, 1687:253–267, 1999.

[6] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[7] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[8] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, 1984.

[9] B. Korel and J. Laski. Dynamic program slicing. *Inform. Process.*, Letters 29(3):155–163, 1988.

[10] GCC, the GNU Compiler Collection. https://gcc.gnu.org/. [Online; accessed 14-March-2021].

[11] The LLVM Compiler Infrastructure Project. https://llvm.org/. [Online; accessed 14-March-2021].

[12] Clang: a C language family frontend for LLVM. https://clang.llvm.org/. [Online; accessed 14-March-2021].

[13] About The ANTLR Parser Generator. https://www.antlr.org/about.html. [Online; accessed 14-March-2021].

[14] Semantic Designs. Dms® software reengineering toolkit. http://www.semdesigns.com/Products/DMS/DMSToolkit.html. [Online; accessed 14-March-2021].

[15] LibTooling - Clang 12 documentation. https://clang.llvm.org/docs/LibTooling.html. [Online; accessed 15-March-2021].

[16] Eli Bendersky. Compilation databases for Clang-based tools. https://eli.thegreenplace.net/2014/05/21/compilation-databases-for-clang-based-tools. [Online; accessed 15-March-2021].

[17] Introduction to the Clang AST - Clang 12 documentation. https://clang.llvm.org/docs/IntroductionToTheClangAST.html. [Online; accessed 15-March-2021].

[18] RecursiveASTVisitor Class Template Reference. https://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html. [Online; accessed 15-March-2021].

[19] Matching the Clang AST - Clang 12 documentation. https://clang.llvm.org/docs/LibASTMatchers.html. [Online; accessed 16-March-2021].

[20] Tutorial for building tools using LibTooling and LibASTMatchers - Clang 12 documentation. https://clang.llvm.org/docs/LibASTMatchersTutorial.html. [Online; accessed 16-March-2021].

[21] AST Matcher Reference. https://clang.llvm.org/docs/LibASTMatchersReference.html. [Online; accessed 16-March-2021].

[22] Eli Bendersky. Modern source-to-source transformation with Clang and libTooling. https://eli.thegreenplace.net/2014/05/01/modern-source-to-source-transformation-with-clang-and-libtooling. [Online; accessed 16-March-2021].

[23] A. Zeller. Automated Debugging: Are We Close? *IEEE Computer*, 2001.

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 First Attachment