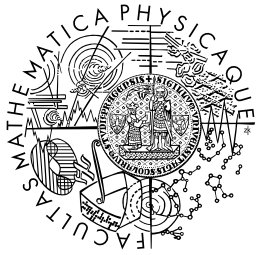

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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 P into its minimal runnable subset P'. The automatically generated program P' also has to result in the same runtime error as 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, fix math fonts.

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 P contain a runtime error E that consistently occurs when 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 $\mathbf{E_L}$. Those include wrongly indexed arrays and calculations that lead to either the incorrect result or an altered control flow of the program. Let $E \in \mathbf{E_L}$. As the generality of errors in $\mathbf{E_L}$ 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 P' be a minimal variant of P such that P' results in the same error E as P when run with the same arguments A. If done carefully and correctly, P' represents the smallest subset of 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 P' as opposed to 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.

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. 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 [1999]. It does not perform any static analysis of the debugged program, as it is not meant to find failures in the code.

TODO: Link the reference number in brackets, i.e. Zeller [X]

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 [2002].

TODO: Link the reference number in brackets, i.e. Zeller [X]

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.

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

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: $n \leftarrow 2 * n$
7: **if** $n > |\sigma|$ **then return** the most recent failure causing substring.
8: **else**
9: goto (2).
10: **end if**
11: **else if** $\alpha_i failed$ **then**
12: $n \leftarrow 2$.
13: $\sigma \leftarrow \alpha_i$.
14: **if** $|\sigma| == 1$ **then return** σ.
15: **else**
16: goto (2).
17: **end if**
18: **else** ▷ β_i failed.
19: $\sigma \leftarrow \beta_i$.
20: $n \leftarrow n - 1$.
21: goto (2).
22: **end if**

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 [1984] defined a slice with respect to criterion C as a part of a program that potentially affects given variables in a given point.

TODO: Link the reference number in brackets, i.e. Zeller [X]

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.

Listing 1.1: Simple branching program

```
#include<iostream>

void write(int x)
{
        std::cout << x << "\n";
}

int read()
{
        int x;
        std::cin >> x;

        return x;
}

int main(void)
{
        int x = 1;
        int a = read();

        for (int i = 0; i < 0xffff; i++)
        {
                write(i);
        }

        if ((a % 2) == 0)
        {
                if (a != 0)
                {
                        x *= -1;
                }
                else
                {
                        x = 0;
                }
        }
        else
        {
                x++;
        }

        write(x);

        return 0;
}
```

Listing 1.2: Static slice of the simple branching program

```
#include<iostream>

void write(int x)
{
        std::cout << x << "\n";
}

int read()
{
        int x;
        std::cin >> x;

        return x;
}

int main(void)
{
        int x = 1;
        int a = read();

        if ((a % 2) == 0)
        {
                if (a != 0)
                {
                        x *= -1;
                }
                else
                {
                        x = 0;
                }
        }
        else
        {
                x++;
        }

        write(x);

        return 0;
}
```

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 [Ottenstein and Ottenstein, 1984] 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.

TODO: Link the reference number in brackets, i.e. Zeller [X]

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.

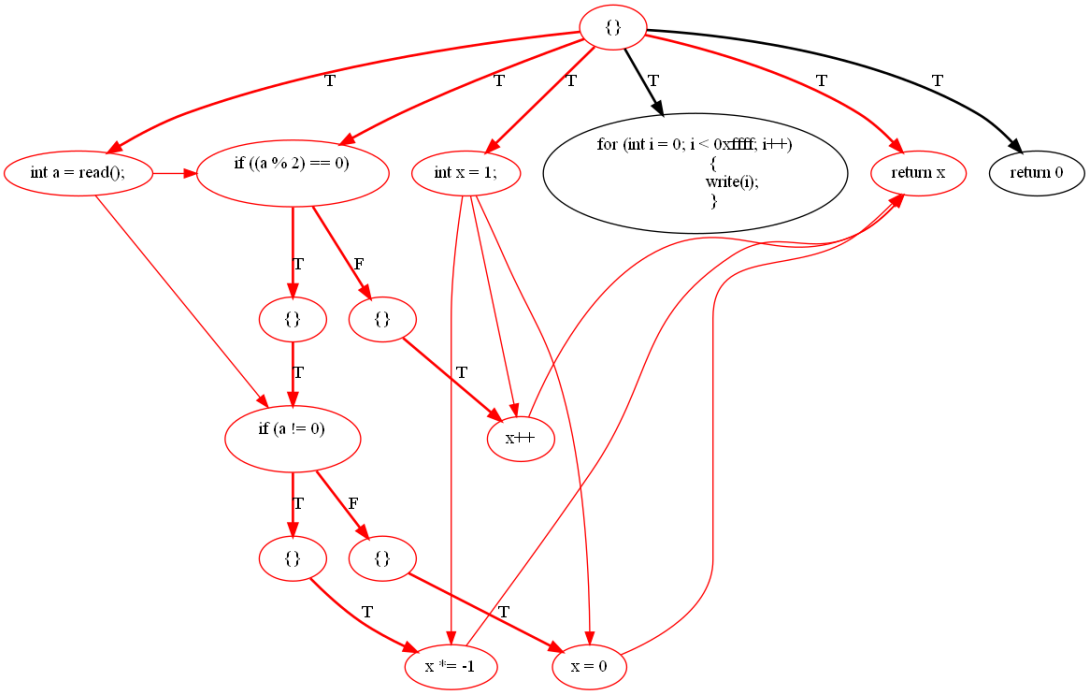

Figure 1.1: Sliced PDG. This figure showcases the PDG of the listing 1.1. Red edges indicate the sliced part of the program w.r.t. $C = (write(x)_{42}, \{x\})$.

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, namely when functions are called using function pointers.

The main advantage of static slicing is that it does not require any run-time 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 and Laski [1988] 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.

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 }
```

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.

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.

TODO: Add more text about AST.

Below are the findings concerning the most important candidates.

2.1 GCC

A well known C and C++ compiler, the GNU Compiler Collection 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.

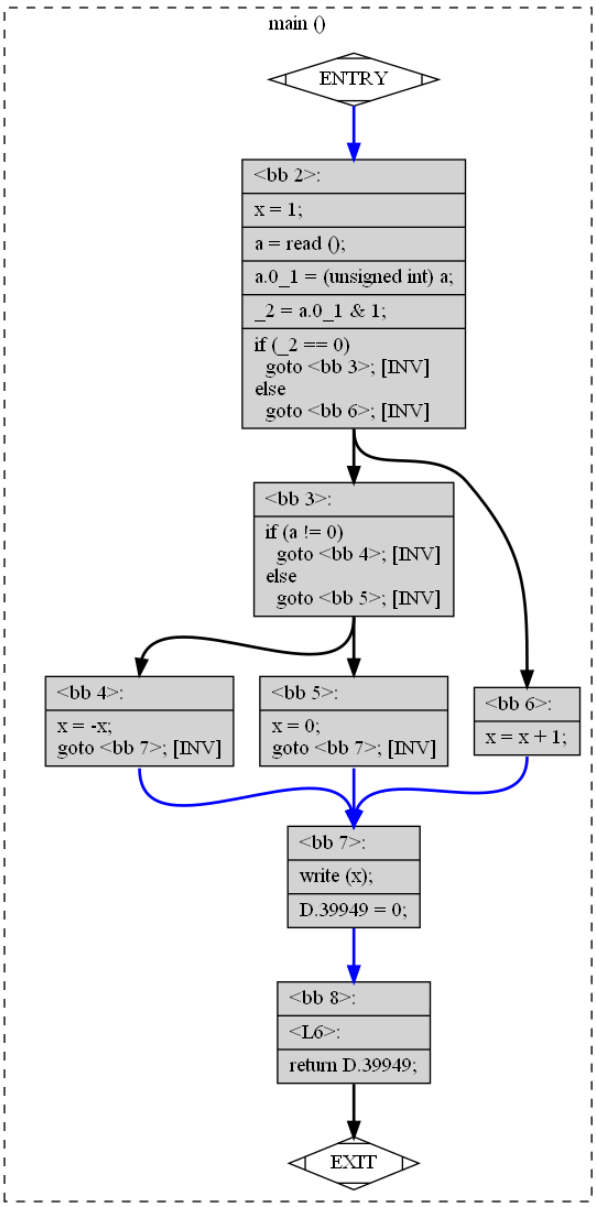

Figure 2.1: GCC AST Dump. This figure showcases the AST representation of 1.1 as dumped by GCC. Note that it is not easily comprehendable.

2.2 Clang

Thanks to LLVM, the widespread compiler infrastructure, the Clang project 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.

TODO: Add more general text based on what I write in the Clang chapter.

2.3 Summary

While the chapter only highlighted two significant candidates, the analysis looked at a plethora of tools. Those, however, were not able to compete feature-wise due to the sheer size and extent of GCC and Clang.

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

TODO: Convert links into references (https://llvm.org/ and https://clang.llvm.org/ and https://clang.llvm.org/docs/LibTooling.html and https://clang.llvm.org/docs/IntroductionToTheClangAST.html and https://eli.thegreenplace.net/2014/05/01/modern-source-to-source-transformation-with-clang-and-libtooling and https://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html)

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 floor and skill ceiling. 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.

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 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.

3.1 Clang AST

TODO: Add AST dump.

The abstract syntax tree used in the Clang front-end 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. 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 topmost node, the root, of Clang AST is called the translation unit declaration. Edges between nodes are simplified, as each node stores a container of its children.

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.

3.2 ASTVisitor

LibTooling offers a built-in curiously recurring template pattern (CRTP) visitor. The class `RecursiveASTVisitor` 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.

TODO: Add example Visit method override.

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.

3.3 Matchers

TODO: Finish matchers, add code examples.

3.4 Source-to-source transformation

To transform source code based on its AST, it 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. 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.

TODO: Talk about adding intrumentation code.

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.

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.

Conclusion

Bibliography

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

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.

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

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

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

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 First Attachment