THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

VeriTest: Automatic Verification of Compiler Optimizations

by

Achmad Afriza Wibawa
47287888
a.wibawa@uqconnect.edu.au

School of Electrical Engineering and Computer Science,
University of Queensland

Submitted for the degree of Master of Computer Science
in the division of Computer Science

3 June 2024

Achmad Afriza Wibawa
a.wibawa@uqconnect.edu.au

June 3, 2024

Professor Michael Bruenig
Head of School
School of Electical Engineering and Computer Science
The University of Queensland
St Lucia, Queensland 4072

Dear Professor Bruenig,

In accordance with the requirements of the degree of Master of Computer Science in the School of Electical Engineering and Computer Science, I present the following thesis entitled

'VeriTest: Automatic Verification of Compiler Optimizations'

This thesis was performed under the supervision of Associate Professor Mark Utting, Emeritus Professor Ian J. Hayes, and Brae J. Webb. I declare that the work submitted in the thesis is my own, except as acknowledged in the text and footnotes, and that it has not previously been submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,

Achmad Afriza Wibawa

To you, saying "Just one more semester".
What could go wrong?

Acknowledgements

I would like to thank my supervisors, Prof. Mark Utting and Prof. Ian J. Hayes, for their continued support and advice. Your guidance is what allowed me to learn and grow over this past year. I am grateful for the opportunity to work on this topic that I love. I apologize if you had to learn latin throughout reviewing my drafts.

I would also extend my gratitude towards my supervisor, Brae J. Webb. You have been an immense help towards my thesis, and I really appreciate your constant feedback and assistance. Thank you for introducing me to this field in Computer Science. This topic sparked my interest in the field that I didn't know existed, and I am grateful.

Finally, thank you Nadhila for always being a constant in my life, through our ups and downs. Thank you for your enduring support throughout the year. You believed in me even when I had lost hope, when I didn't believe in myself. I haven't been the most loving person, but you certainly are. I am eternally grateful to have you in my life, and I certainly cannot do this without you and our pigs. It's your turn next year my love, and I certainly believe that you can do this as well.

Abstract

Ensuring compiler correctness is critical to mitigating the risks of errors, especially during the optimization phases of a compiler. Despite existing approaches to identifying incorrect optimizations in GraalVM's expression optimizations, none incorporate a solver-based tool that provides GraalVM developers with a sufficiently comprehensive analysis of the optimization rules without requiring expertise in formal proofs. Thus, we introduce VeriTest, a tool designed to automatically verify the existence of a simple proof of optimization correctness or identify counterexamples for incorrect optimizations. VeriTest leverages Isabelle's automated tools and Veriopt's representation of GraalVM's intermediate representation (IR) as a domain-specific language (DSL) within Isabelle. This combination enables a comprehensive analysis of optimization rules. The implementation abstracts Isabelle Client - Server interactions to ensure efficient and concurrent processing of optimization rules, while also assuring extensibility for future refinements. Our evaluation demonstrates the capabilities of VeriTest in automatically verifying optimization rules, though significant limitations were found in identifying counterexamples due to the inherent limitations of Isabelle and Veriopt's DSL. However, this also implies that the analysis is expected to improve as the theoretical foundations for GraalVM's IR is improved. Despite these challenges, VeriTest has shown significant potential as an automatic verification tool for GraalVM's expression optimizations, providing a solid foundation for future enhancements.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

While compilers are generally believed to be correct, unwanted errors do happen, especially in the optimization phases of a compiler. This is due to: the common pitfalls of the language it's written in are *just* accepted by the community; edge cases of the language that are not well considered; or simply making a mistake inside the implementation [1, Sec. 1.2]. Human mistakes are natural in human-made software. As such, it is critical to *try* to minimize the intrinsic risks of error in compilers.

Minimizing the risks of error is non-trivial. A suite of testing mechanisms is needed to ensure the reliability of software. There are several ways to do this. For software, regression testing in the form of Unit, Integration, and System level tests are the industry standard ways for mitigating risks [2].

Such testing suites are ideal for software with human-understandable behavior. However, the behaviors of compilers itself are not exactly human-readable. As such, manually defining the obscure behaviors of compilers is tedious and time-consuming [3]. Another way to verify the behavior of compilers is to **formally** verify their specifications [3].

1.1 Objectives & Scope

The primary purpose of this project is to provide a framework for answering the 2nd step of the compiler research thread proposed by Lopes and Regehr:

> "Implement solver-based tools for finding incorrect optimizations, not-weakest pre-conditions, groups of optimizations that either subsume each other or undo each other, etc." [1, p. 5].

Hence, this project introduces VeriTest: a software interface for automatic verification of GraalVM's expression optimizations. VeriTest aims to provide fast analysis over proposed optimization rules – written in Veriopt's GraalVM intermediate representation (IR) domain specific language (DSL) [4, Sec. 3] – to identify incorrect optimizations. GraalVM developers could integrate the analysis into their development workflow. This would make it easy for GraalVM developers to use the tool *as you go*, without being a *"proof expert"* on Isabelle.

To achieve this, we intend to leverage automation tools built in interactive theorem provers such as Isabelle [5], inspired by the previous works such as CompCert [3] and Alive [6], [7]; all of which integrate the theoretical aspects of formal verification into the practicality of using it in the industry. This project builds upon the previous works done to introduce formal semantics for GraalVM's [8] IR [4], [9] implemented in Isabelle [5].

1. GraalVM developers write a proposed optimization proof in Veriopt's DSL.
2. The optimization rule would be passed into VeriTest by the developers with the following goals:
   (a) Determine if it is verifiable, either a simple proof for correct rules or counterexample for incorrect rules;
   (b) If VeriTest can't determine, optimization rules would be passed to *"proof experts"*.

Figure 1.1: Proposed development workflow for GraalVM developers

However, verification that DSL matches the implementation of GraalVM would be out of the scope of this project. It would represent a thorough 2nd step of the compiler verification research thread [1, p. 5] – similar to Alive2's [7] solution on LLVM's formal verification. Furthermore, extending Veriopt's DSL and Isabelle's automated tools is out of the scope of this project, considering the time constraints of this project and the significant amount of work needed to undertake the endeavour. All of which could represent a future direction in the research.

Additionally, in relation to the evaluation of VeriTest, we will conduct benchmarks upon Veriopt's current theory base and GraalVM's source code annotations, which is elaborated in Section 5. However, the purpose of this evaluation is not to benchmark Isabelle, but rather to provide a starting point to understand the capabilities of VeriTest. There is a community-led benchmark spearheaded by Huch and Bode [10] which evaluates the performance of Isabelle for typical computations.

1.2 Thesis Structure

In relation to the goals, there would be several questions implied:

1. Is it possible to extend the current workings of Veriopt in Isabelle to a tool that could provide analysis for the developers?
2. Would the analysis be useful to GraalVM developers?
3. How fast could the analysis be provided to GraalVM developers?

In order to achieve the goals of this project, we conducted a literature review in Chapter 2 that explores why software testing is considered to be inadequate for eliminating errors inside a compiler, and the significancy of formally specifying a compiler – done by CompCert [3], Alive [6], [7], and Veriopt [4], [9]. Furthermore, the capabilities of Isabelle as an interactive theorem prover that can help provide analysis towards optimization rules are described in this chapter. Chapter 3 examines the possible approaches to extend the current workings of Veriopt in order to analyze optimization rules.

Furthermore, Chapter 4 describes VeriTest's current implementation, following what we consider to be the best possible approach to analyze optimization rules considering the time constraints of this project. The implementation of VeriTest considers several key requirements: (1) extensibility and modularity of the implementation for future work to be done; (2) comprehensiveness of the analysis; and (3) performance of VeriTest as a system. Chapter 5 evaluates

the key requirements of VeriTest, and examines our findings towards leveraging Isabelle as VeriTest's proof engine.

Finally, Chapter 6 concludes by discussing the projects contribution to the field of compiler research. This chapter also reviews our findings and provides recommendations for future work to be done.

Chapter 2

Background

2.1 Software Testing

Testing is a critical part of the software development life cycle [2]. It aims to minimize the risk of errors inside the software itself. In testing, we attempt to determine whether the semantics of a software follows their specifications. However, we quote Dijkstra's famous line [11]:

> "Testing can be used to show the presence of bugs, but never to show their absence".

2.1.1 Regression Testing

In software engineering, the most commonly used method of software testing is regression testing. Regression testing revolves around identifying parts of the program that are changed and verifying their behavior through a test suite; e.g. Unit, System, and Integration-level tests [2]. This only shows the presence of bugs inside the program.

In practice, regression testing is limited to the capabilities of humans to define the behavior of the program. Therefore, there is an inherent risk of errors happening due to human mistakes; it's possible that the defined behavior is not correct. Defining the complete set of the behavior of the intended program requires developers to spend a considerable amount of time writing tests manually [12].

Thus, **random testing** is introduced to substitute humans with defined systematic computer behaviors.

2.1.2 Random Testing

Random testing is a suite of random tests generated by the computer to determine the correctness of the program based on a predetermined set of rules [12]. For example, [13, Sec. 2] generates test cases and executes them on programs that have a predictable result. For instance, if a program deviates from the expected results, then the program has undefined behaviors.

The difficulty of random testing lies in determining the set of rules to generate test cases. In [12], the test cases are generated by substituting a subset of the input to a random input. While this allows test cases to be generated quickly, programs only need several *"interesting values"* or edge cases to consider in their behavior. As such, a stronger test suite such as an **inference-based test generation** will be preferable in this project.

2.1.3 Inference-based Exhaustive Testing

Exhaustive tests such as [14] take into account the possible variable bounds of a program and convert them to a set of inference rules. For a program to be correct, all of the premises *(P, Q)* in the inference rules *(P →Q)* must be correct. Hence, finding a counterexample is as simple as determining if the bounds of a variable result in a satisfiable ¬*(P →Q)* expression. This could be extended even further by checking the inference rules on an SAT solver to find premises where the inference rules are incorrect [15, Ch. 5]. Exhaustive searching allows developers to focus on defining the behavior of the program rather than defining tests to verify the behavior.

2.1.4 Mutation Testing

Mutation testing is a form of testing where faults (or mutations) are inserted into a program [16]. These mutations are based on the grammar of the program and represent programs that are invalid. A key aspect of mutation testing is to determine suitable mutations to inject into the program called *mutation operators* [16, Sec. 2], as inserting random values into the program would be akin to random testing. Specifically, Offutt et al. [16, Sec. 6] elaborates on how mutation testing can be used to generate valid or invalid input strings that will have erroneous responses.

1. **Nonterminal Replacement**

   Every nonterminal symbol in a production is replaced by other nonterminal symbols.

2. **Terminal Replacement**

   Every terminal symbol in a production is replaced by other terminal symbols.

3. **Terminal and Nonterminal Deletion**

   Every terminal and nonterminal symbol in a production is deleted.

4. **Terminal and Nonterminal Duplication**

   Every terminal and nonterminal symbol in a production is duplicated.

Figure 2.1: Mutation operators for input strings. Summarised from [16, Sec. 6.2]

2.2 Isabelle

Isabelle is an interactive theorem prover that utilizes a multitude of tools for automatic proving – similar to Coq, used by CompCert 2.3.1. Isabelle emphasizes breaking down a proof for a theorem toward multiple smaller goals that are achievable called tactics. Tactics are functions, written in the implementation of Isabelle, that work on a proof state [15]. Tactics either output a direct proof towards the goal or break them down into smaller sub-goals in a divide-and-conquer manner. These tactics work on the foundation that theory definitions can be modified into a set of inference rules that can be automatically reasoned with by the system.

Finding the right proving methods and arguments to use is one of the biggest challenges for proving a theorem [15]. Many tools in Isabelle's arsenal can help the user progress towards their proof [5]. However, the most notable ones are Sledgehammer, Quickcheck, and Nitpick.

2.2.1 Sledgehammer

Sledgehammer is one of the tools in Isabelle that *could* automatically prove a theorem. It utilizes the set of inference rules as conjectures that can be cross-referenced with relevant facts (lemmas, definitions, or axioms) from Isabelle [15, Sec. 3]. Afterward, Sledgehammer passes them into resolution prover and SMT solvers that try to solve it [15, Sec. 3.3]. If reasonable proof is found, Sledgehammer reconstructs the inference rules back into a *relatively* human-readable proof definition in the style of Isabelle/Isar [15, Sec. 3.4].

Utilizing Sledgehammer has proven to be an effective method of theorem proving. Sledgehammer, combined with external SMT solvers, can solve 60.1% of proof goals, with a 44.7% success rate for non-trivial goals [17, Sec. 6]. Despite their potential, as Blanchette et al. note [15, p. 2]:

> "...most automatic proof tools are helpless in the face of an invalid conjecture. Novices and experts alike can enter invalid formulas and find themselves wasting hours (or days) on an impossible proof; once they identify and correct the error, the proof is often easy."

To make it easier for users to avoid this trap, Isabelle complements the automatic theory proving with counterexample generators such as Quickcheck and Nitpick.

2.2.2 Quickcheck

Quickcheck is one of the counterexample generators in Isabelle. It works by utilizing the code generation features of Isabelle, translating conjectures into ML (or Haskell) code [14]. This allows Quickcheck to discover counterexamples quickly by assigning free variables on the code via random, exhaustive, or narrowing test data generators. However, this would also mean that Quickcheck is limited to *executable* and *some* well-defined unbounded proof definitions [14].

Random testing of a conjecture assigns free variables with pseudorandom values [14, Sec. 3.1]. This strategy tends to be fast, with the ability to generate millions of test cases within seconds. However, random testing could easily overlook obvious counterexamples. Furthermore, random testing is also limited to well-defined proof definitions [14]. As such, exhaustive and narrowing test data generators are more suitable for proof definitions that are non-trivial or have unbounded variables.

Exhaustive and narrowing test data generators systematically generate values up to their bounds [14]. However, exhaustive test data generators fail to find counterexamples of proofs that have unbounded variables. Narrowing test data generators improves it by evaluating proof definitions symbolically rather than taking variables at face value. This is possible due to term rewriting static analysis done on proof definitions [14, Sec. 5].

Based on observed results, Bulwahn notes that random, exhaustive, and narrowing testing are comparable in terms of performance; with exhaustive testing finding non-trivial counterexamples easily compared to random testing [14, Sec. 7]. As much of proof definitions are defined over unbounded variables, exhaustive testing is the default option for Quickcheck.

2.2.3 Nitpick

Nitpick is an alternative tool for finding counterexamples for proof obligations. Instead of enumerating the bounds of free variables inside the system of Isabelle, Nitpick passes the translated conjectures – translation of proof obligations into inference rules – into SMT solvers [15, Sec. 5]. SMT solvers search for premises that falsify the given conjecture. If a conjecture has bounds over finite domains, Nitpick will *eventually* find the counterexample. Conjectures with unbounded variables will be partially evaluated [15, Sec. 5.2]. However, it could not determine whether infinite bounds result in a satisfiable conjecture.

Nitpick and Quickcheck shouldn't be compared to one another. Instead, they are tools that complement each other to determine whether a conjecture will be possible to prove [14]. The performance of Nitpick is comparable to Quickcheck, with the addition that Nitpick can find counterexamples to proof definitions that are not executable within Isabelle's code generation [14, Sec. 7].

2.2.4 Limitations

It is worthy to note that Quickcheck has some limitations over arbitrary type definitions [14, Sec. 3.1] and conditional conjectures [14, Sec. 4]. For arbitrary type definitions, Quickcheck is unable to transform the conjectures into internal Isabelle datatypes. As such, users need to define a way to *construct* their datatype, which would be used by Quickcheck to build test generators. For conditional conjectures, Quickcheck would evaluate the given conjectures with no regards to their premises [14, Sec. 4]. As such, users would need to specify their own test generators that would take the premises into account [14, Sec. 4.1].

Furthermore, this limitations extends to Nitpick. For (co)inductive datatypes, Nitpick needs the user to properly define their encoding of (co)inductive datatypes, and manually provide selectors/discriminators if Nitpick cannot automatically infer one [15, Sec. 5.4]. Moreover, Nitpick has explicitly stated several known bugs and limitations related to the completeness of a datatype's encoding. If an encoding (i.e., Veriopt's DSL on Section 2.3.3) is underspecified, or it involves nested (co)recursion through non-(co)inductive datatypes (e.g., an Isabelle word), it might lead to spurious errors and errors related to unsupported semantics [18, Ch. 8].

2.3 Formal Verification of Compiler

If software code is the recipe for system behaviors, then a compiler would be the chef who puts it all together. Most people would assume that the behavior of compiled programs will match the input program exactly. However, this is usually not the case [3]. Chefs have their way of creating magical concoctions from a recipe, and so does a compiler. Not only does a compiler try to replicate system behaviors, but it will try to make them faster in their ways; i.e. adding optimizations or reducing unneeded behaviors. However, the original behavior of the program must be preserved in order to consider a compiler to be correct [3], [4], [6], [7].

Verifying compiler correctness is not easy. While the correctness of software *could* be verified by defining behaviors through regression testing, compiler bugs are infamous for being hard to spot [2], [3]. Hence, it will be time-consuming to do and not exactly productive. There are a multitude of ways that a compiler can go wrong [1, Sec. 1.2]; all of which have their specific way of verification.

For example, CompCert [3] tries to tackle all of the implementation and semantics errors inside a compiler (See Sec. 2.3.1) – creating a completely verified compiler for the C language

platform. Formally verifying compilers on the scale of CompCert will require vast amounts of time and resources, which projects often don't have.

As such, there are smaller-scale projects such as Alive [6] & Alive2 [7] that focus on behavior translation errors in LLVM's peephole optimizer (See Sec. 2.3.2). Veriopt attempts to work on the same steps as CompCert – by defining the IR of GraalVM and proving much of the side-effect-free data-flow behavior optimizations that occur in GraalVM [4], [9] (See Sec. 2.3.3).

2.3.1 CompCert

CompCert verifies that Clight, a subset of the C programming language [3], is correct through several steps:

1. With deterministic programs, a compiler compiles a source program to the produced program – in which both of the programs must have the same behavior.

   This step is done by augmenting the compiler code with a *certificate* – code that carries proof that the behavior is exactly as intended [3, Sec. 2.2].

2. Compiler optimization rules must be accompanied by the formal definition of their intermediate representation (IR) semantics [3], [9].

3. Lastly, to formally verify each of the optimization rules, a compiler must either:

   (a) Prove that the code implementing the optimization is correct [3, Sec. 2.4].
   Veriopt uses this approach in verifying data-flow optimizations (See Sec. 2.3.3).

   (b) Prove that the unverified code produces the correct behavior in their translation [3, Sec. 2.4].
   Alive uses this approach in verifying LLVM (See Sec. 2.3.2).

CompCert formally verifies each step in the source, intermediate, and target languages that go through the compiler [3, Sec. 3.3]. Furthermore, each code translation and optimization are accompanied by *certificates* that prove the correctness of the semantics. This is done through Coq, a proof assistant similar to Isabelle. The workflow of Coq also remains closely related to Isabelle (See Sec. 2.2) [3, Sec. 3.3].

CompCert utilizes Coq not only to formally verify the semantics of Clight but also to generate the verified parts of the compiler code [3, Sec. 3.4]. The clever part of CompCert is that it uses the functional programming capabilities of Coq to automatically generate code. As such, it is able to write a compiler-compiler – which is the 3$^{\text{rd}}$ step of compiler verification research [1].

The formal verification of Clight results in 42000 lines of Coq – approximately equivalent to 3 years of man-hours of work [3, Sec. 3.3]. As you can see, replicating the results of CompCert will require an enormous amount of work. Formal verifications such as Alive (See Sec. 2.3.2) and Veriopt (See Sec. 2.3.3) undertake the smaller subset, namely the 1$^{\text{st}}$ and 2$^{\text{nd}}$ step, of the compiler verification research thread [1].

2.3.2 Alive

Alive takes on a subset of compiler verification by verifying that code optimizations inside LLVM are correct [6]. For example, a compiler will optimize:

$$(LHS = x * 2) \implies (RHS = x << 1) \tag{2.1}$$

(2.1) explains that LHS is transformed into RHS [6, Sec. 2.1]. While this may seem trivial, there would be a lot of edge cases where the behavior translation might be incorrect (e.g., buffer overflows).

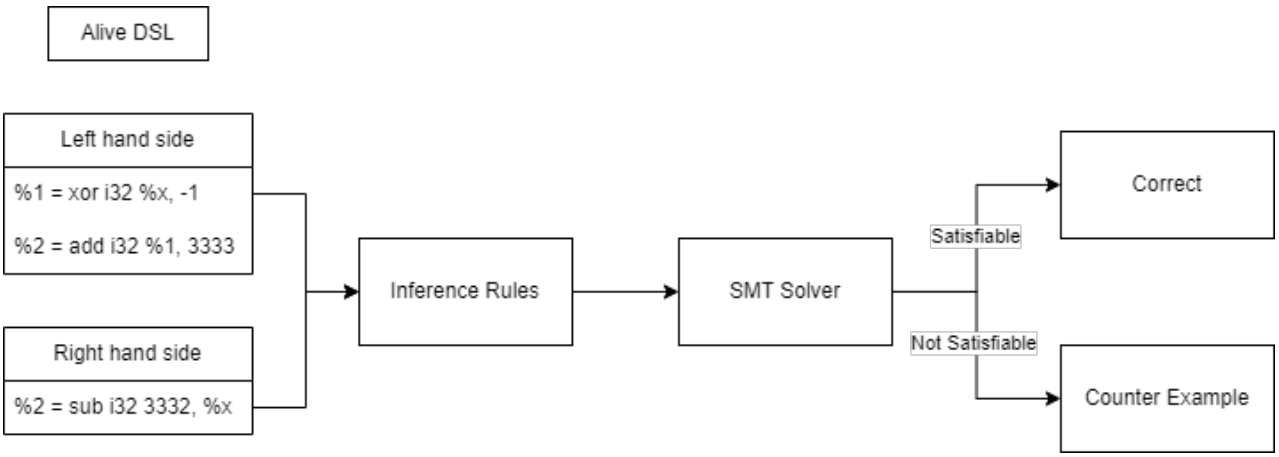

Figure 2.2: How Alive verifies $(LHS = (X \oplus -1) + C) \Longrightarrow (RHS = (C - 1) - X)$ [6, p. 1]

To verify that code optimizations are correct, Alive utilizes inference-based exhaustive testing (See Sec. 2.1.3) that allows the tool to encode machine code behaviors to inference rules [6, Sec. 3.1.1]. These inference rules are then passed to SMT solvers to check for their satisfiability (See Figure 2.2). If the translation is proven to be correct, then it means that the underlying optimization code is correct.

Alive encodes LLVM's [19] underlying IR semantics through their DSL [6, Fig. 1]. The DSL specification is made to be similar to LLVM's IR semantics to allow developers to easily integrate Alive with the development of DSL. This represents the *certificate* that the underlying optimization is formally verified to be correct.

Alive takes this further by creating Alive2: a system to translate LLVM IR into Alive's IR [7]. This allows the developers to entirely focus on developing LLVM, while completely ignoring the specifications of Alive. This is found to be effective, as differential testing of LLVM's unit tests and Alive2 discover multiple errors inside the unit test behaviors themselves [7, Sec. 8.2]. Alive & Alive2 covers the whole 1st and 2nd steps of the compiler optimization research thread [1, p. 5].

2.3.3 Veriopt

In comparison to CompCert and Alive, the theoretical aspects of compiler verification are similar. GraalVM's IR is made up of two components: control-flow nodes and data-flow nodes; which are combined as a sea-of-nodes data structure [9]. However, Veriopt's DSL only concerns the subset of GraalVM's IR, which is the side-effect-free data-flow nodes [4]. Side-effect-free data-flow nodes are comparatively easier to prove and optimize, as it is considered to be defined – as opposed to LLVM's undefined and poisoned variables [7].

Figure 2.3 defines the structure of the DSL for an optimization rule. Veriopt's DSL is implemented in Isabelle [5], which represents optimization rules as inductive datatypes that allows efficient reasoning within Isabelle [9, Sec. 3] [20]. The **optimization** keyword represents the proof definition that must be proven in Isabelle. Note that 2 proof obligations must be met to consider that the side-effect-free optimization is correct: (1) proof that the optimization

optimization *InverseLeftSub:* $(x\ -\ y)\ +\ y \longmapsto x$

1. $trm(x)\ <\ trm(BinaryExpr\ BinAdd\ (BinaryExpr\ BinSub\ x\ y)\ y)$
2. $BinaryExpr\ BinAdd\ (BinaryExpr\ BinSub\ x\ y)\ y \sqsupseteq x$

Figure 2.3: Sample of Veriopt's optimization rule and proof obligations DSL [4, Fig. 3]

rule will terminate; (2) proof that each pass of the optimization rule will result in a refinement of the expression [4]. Note that these proofs need to be provided by users.

2.4 Model-Driven Software Development

Formal verification of a compiler *can* be considered as a model-driven approach to software development. Model-driven development begins with high-level abstractions of an intermediate representation for the semantics of the software [21, p. 43]. Bünder [22] demonstrated how model-driven development by the use of a DSL can be integrated into your typical integrated development environment (IDE) through the use of a language server protocol (LSP) to enhance the developer's experience. An LSP enables separation of concerns and simplifies the complexity to integrate the analysis for a language by abstracting communications to a language analysis server [22, Sec. 3.1]. This practice is widely used to give the developers feedback [22, Sec. 3.2] by IDEs such as IntelliJ [23] to give code completion or static analysis tools such as SonarQube [24] to give metrics of code quality and bugs.

This closely resembles Veriopt's approach to *model* the semantics of GraalVM into their respective DSL (See Section 2.3.3), and the goals of VeriTest to provide feedback towards the proposed DSL. Currently, there are some tools that the developers of GraalVM could use to provide a *certificate* for the compiler code [4, Sec. 7]. A semi-automated approach exists in the form of source code annotations [4, Sec. 5.1]. However, this semi-automated approach is inadequate as it only matches the existence of optimization rules in Veriopt's current theory base without any logical reasoning done.

Providing proof for the proof obligations of an optimization rule (i.e., Figure 2.3) will be challenging for developers who are not experts in formal verification, as it requires the developers of GraalVM to be familiar with Isabelle – something that ideally Veriopt would like to avoid. Similar tools that provide feedback through an IDE (i.e., SonarQube [24]) are preferrable. Hence, that is where this project contributes: by providing a tool to automatically verify the existence of a simple proof that the optimization is correct, or a counterexample for incorrect optimizations.

Chapter 3

Possible Approaches

Veritest represents another tool that the developers of GraalVM could use to provide a *certificate* for their implementation of optimization rules. To assist in verifying the optimization rule, the tool would need to be able to classify each of the optimization rules (See Figure 3.1). Furthermore, there are several key non-functional requirements for the system that need to be satisfied (See Figure 3.2).

1. The optimization rule is false;

   This would require the tool to generate obvious counterexamples via Quickcheck (See 2.2.2) or Nitpick (See 2.2.3).

2. The optimization rule is true;

   This would require the tool to verify that Sledgehammer (See 2.2.2) can provide proof obligations for the optimization rule **without** dynamically defining proof tactics.

3. The optimization rule would require manual proving by *"proof experts"*.

   This means that the optimization is non-trivial: Isabelle is not able to find an obvious counterexample, and proving would require additional tactics or sub-goals to be defined.

Figure 3.1: The classification of an optimization rule

1. Developers of GraalVM need to be able to integrate this easily into their test suite;
2. Developers of GraalVM can easily use this without understanding Isabelle;
3. *If possible*, the system doesn't require enormous computing resources locally.

Figure 3.2: Non-functional requirements of VeriTest

Understanding Isabelle's implementation is crucial to realize the goals of this project. At a glance, there are four possible approaches to the project:

- Utilize Isabelle Client - Server interactions (See Sec. 3.2);
- Extend Isabelle/Scala (See Sec. 3.3);
- Utilize Isabelle CLI (See Sec. 3.4)
- Create an Interpreter for GraalVM's optimization DSL (See Sec. 3.5).

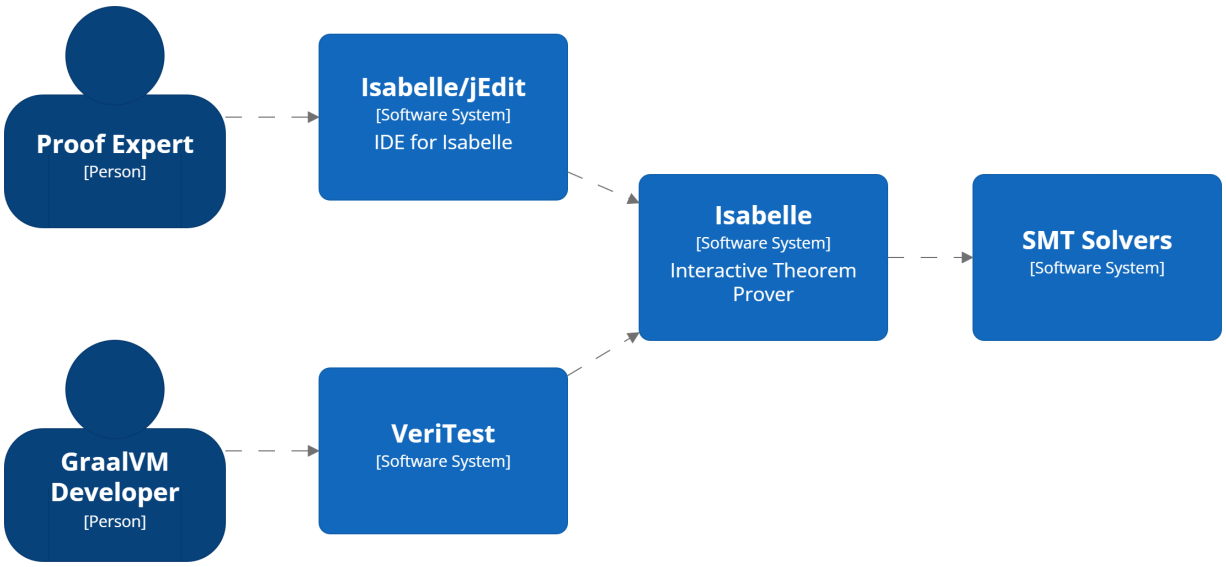

Figure 3.3: Proposed Solution

Figure 3.3 depicts the overview of the proposed solution's system landscape. In theory, it would utilize Isabelle in a similar manner as Isabelle/jEdit [25]. Therefore, it should be able to use the same Isabelle functionalities as Isabelle/jEdit does.

3.1 Isabelle System Overview

Isabelle is made up of two significant components: Isabelle/ML and Isabelle/Scala [25, Ch. 5] (See Figure 3.4). Isabelle/ML acts as the core functionality of Isabelle, harboring all the tools needed for proving theorems. Isabelle/Scala acts as the system infrastructure for Isabelle/ML – hiding all implementation details of Isabelle/ML.

3.2 Utilizing Isabelle Server

Isabelle Server acts as the core Isabelle process that allows theorems and all the required facts to be loaded up and processed by Isabelle/ML [25, Ch. 4]. Interactions to Isabelle Server require a duplex socket connection over TCP [25, Sec. 4.2]. To simplify the communication between the framework and Isabelle Server, we utilize Isabelle Client [25, Sec. 4.1.2] – a proxy for Isabelle/Server that handles all the communication protocols of Isabelle Server (See Figure 3.5).

Isabelle Server can load theorems and process requests in parallel [25, Sec. 4.2.6]. This implies that it would require a *facade* that implements a demultiplexer for asynchronous

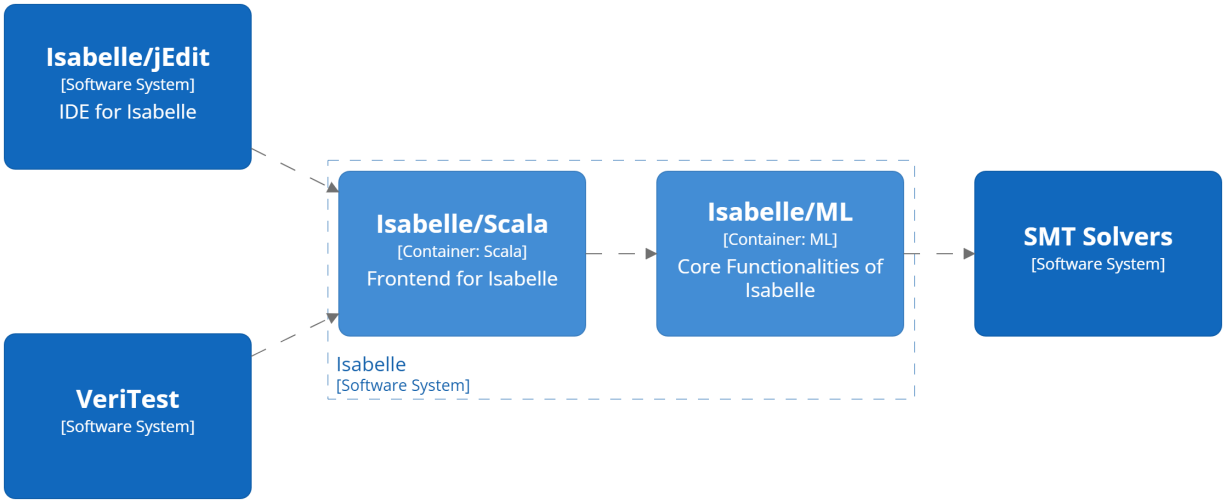

Figure 3.4: Isabelle System Overview

messages on Isabelle Client (See Figure 3.5). As such, this solution – *theoretically* – it would allow the framework to offload the computing resources of loading and processing optimization proofs at external sites (See Ch. 5). VeriTest currently implements this approach, which is elaborated in detail in Ch. 4.

3.3 Extending Isabelle/Scala

Isabelle can be extended by accessing Isabelle/Scala functions [25, Ch. 5]. Extending Isabelle/Scala would require the framework to utilize Isabelle's Scala compiler [25, Sec. 5.1.4]. Utilizing Isabelle/Scala would also enable VeriTest to leverage Isabelle/ML. Subsequently, this also implies that it is possible to extend and fine-tune existing automated tools to fit the requirements of this project. Figure 3.6 depicts the proposed solution for this option.

However, the sheer complexity of Isabelle implies that extending Isabelle/Scala would require much of the project's timeline to understand Isabelle/Scala, which the time constraints of this thesis project does not allow. Furthermore, extending Isabelle/Scala *could* mean that the framework would take up much of the computing resources to execute Isabelle/ML functions locally.

3.4 Utilizing Isabelle CLI

Veriopt's current semi-automated approach [4, Sec. 5.1] utilizes Isabelle CLI, a wrapper for Isabelle/Scala functions [5]. As such, this solution would represent an Isabelle management service, where each optimization rule will be passed into an Isabelle/Scala function inside an Isabelle process and managed accordingly. In essence, the solution is similar to Sec. 3.3. However, the solution will require additional processing overhead in the form of multiple local Isabelle processes for each of the optimization rules, instead of a single VeriTest and Isabelle instance.

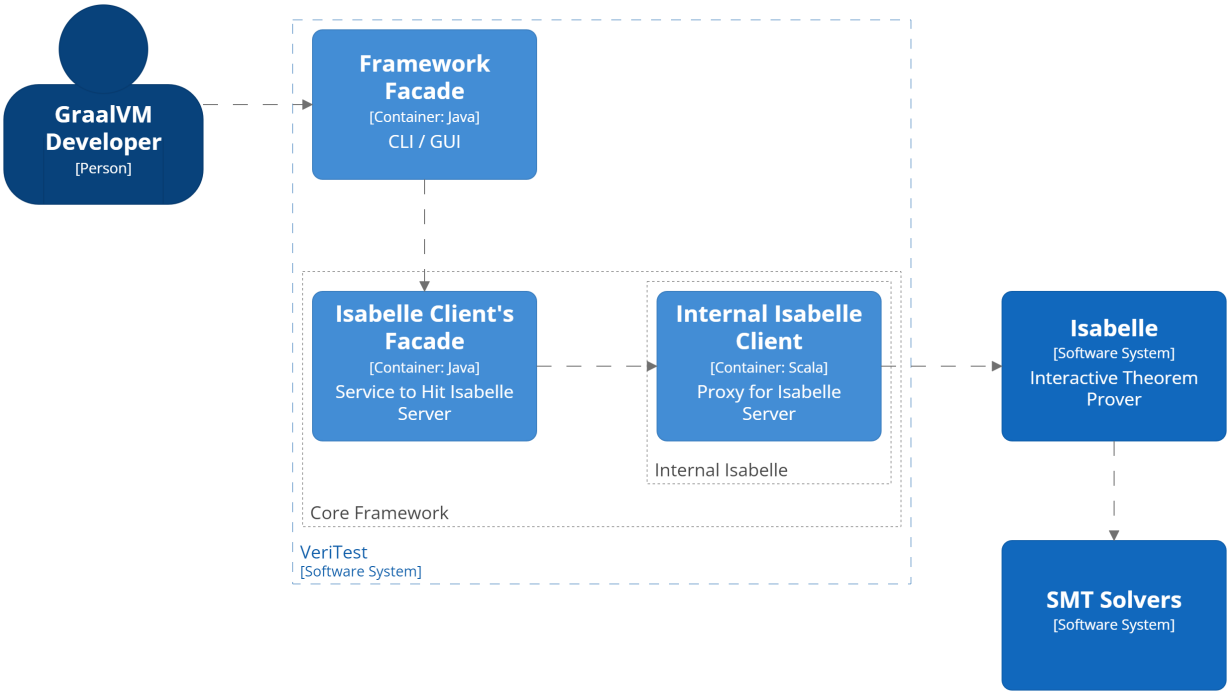

Figure 3.5: Utilizing Isabelle Client - Server interaction

3.5 Interpreter for DSL

Building an interpreter for GraalVM's optimization DSL acts as a last resort to the project. To implement this, it would require a significant amount of time to rework the DSL into the framework, and designing tools similar to Quickcheck (See Sec. 2.2.2) in order to satisfy the system requirements. Reinventing the wheel would not be productive for the project, and it would result in a tool that is far inferior to Isabelle. Therefore, this option should be avoided.

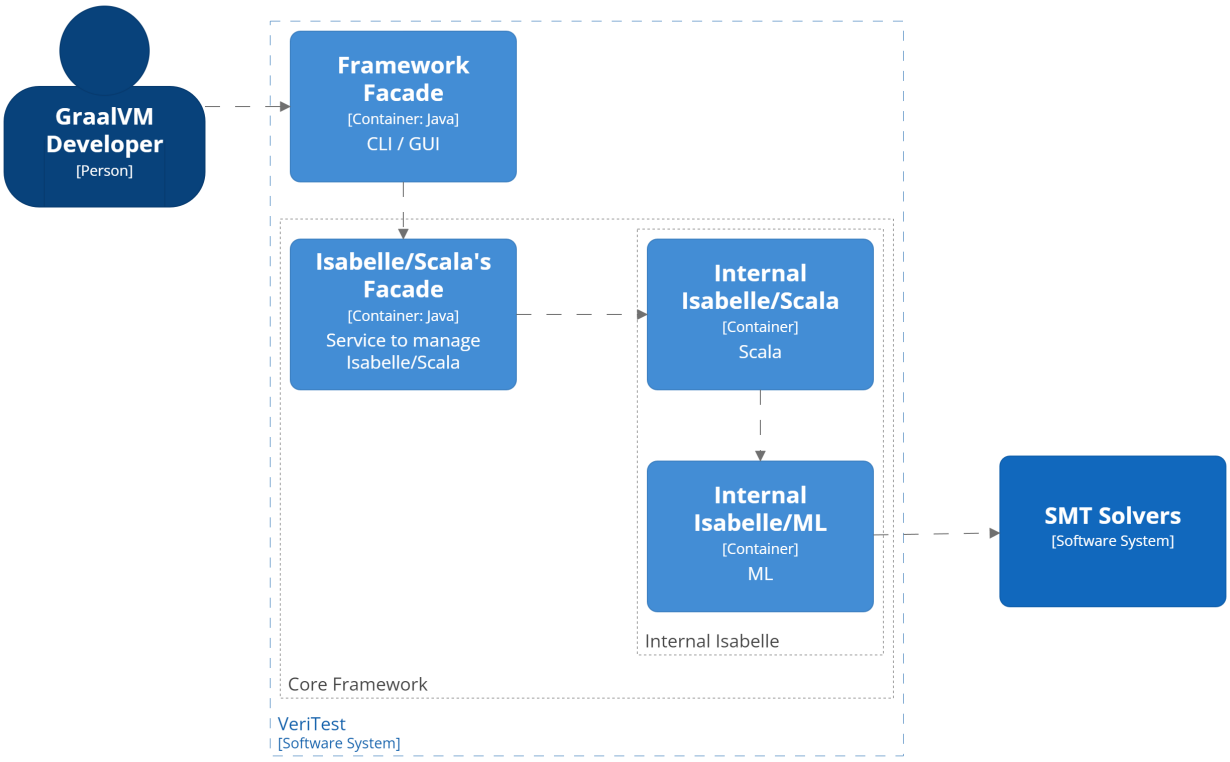

Figure 3.6: Extending Isabelle/Scala

Chapter 4

Implementation

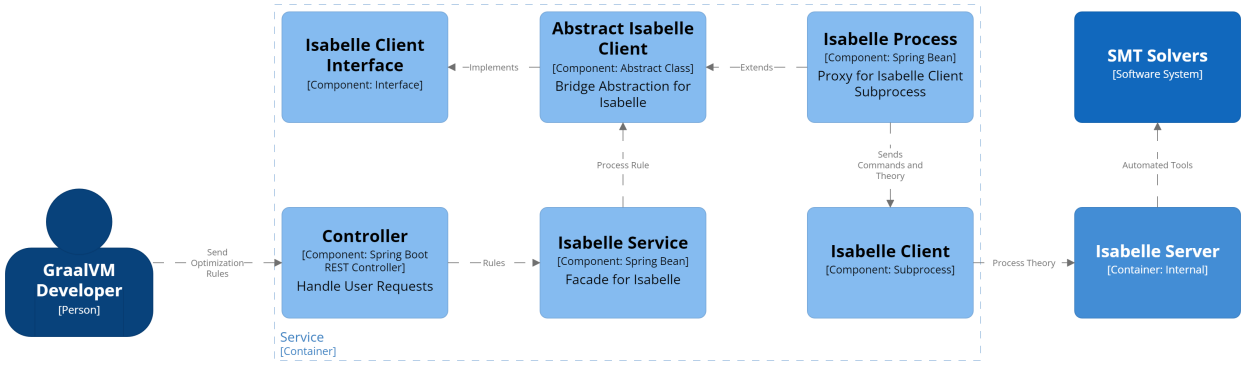

Figure 4.1: Software Architecture of Veritest

Figure 4.1 outlines the architecture implemented for VeriTest to implement a solver-based tool for finding incorrect optimizations, referred by Lopes and Regehr [1, p. 5]. VeriTest is available in online repository[1]. The API documentation for VeriTest can be accessed through Postman[2]. There are several key design decisions needed to achieve the functional and non-functional requirements of this project (See Sec. 3), all of which are outlined in the following sections.

4.1 Isabelle as Proof Engine

We leverage Isabelle's automated tools – Sledgehammer to find proofs; Nitpick and Quickcheck to find counterexamples – to carry out the analysis for each of the optimization rules. This is possible due to the intermediate representation (IR) of GraalVM represented as an inductive datatype inside Isabelle [9, Sec. 3]. This encoding allows Isabelle to automatically translate the encoding into datatypes internal to Isabelle [20]. The efficacy of this method is outlined in Section 5.

Furthermore, as the inductive datatype acts as the grammar of Veriopt's DSL, erroneous syntax can be detected automatically by Isabelle. This is due to the fact that the inductive datatype expects certain types to be present in order for the optimization to be syntactically correct.

[1]https://github.com/achmadafriza/veritest-dev
[2]https://documenter.getpostman.com/view/12004801/2sA35D64D8

optimization *WrongMultiplication:* $(x * 1) \longmapsto x$ (4.1)

optimization *CorrectMultiplication:* $(x * (ConstantExpr\ (IntVal\ b\ 1))) \longmapsto x$ (4.2)

In (4.1), we see that the optimization is intuitively correct. However, as $(x * 1)$ is translated into *BinaryExpr BinMul x 1*, Isabelle expects the syntax to be in the form of *BinaryExpr BinMul IRExpr IRExpr* [4, Definition 1]. With type unification, we can infer that x is of type *IRExpr*. However, 1 is internally encoded in Isabelle as an Isabelle word, not an IR expression [4, Definition 2]. We can amend the optimization rule by defining it as (4.2). As such, syntax errors can be detected.

4.2 Mutual Exclusion

optimization *WrongOptimization* : $x + y \longmapsto x - y$ (4.3)
sorry

optimization *ImpactedOptimization*: $x * 1 \longmapsto x$ (4.4)
nitpick

Figure 4.2: Example of conflicting optimization rules

As optimization rules are *proposed* in VeriTest, the proposed optimization rule might be incorrect due to counterexamples. In Figure 4.2, we see that optimization rule (4.3) is obviously false due to arithmetic rules. But as we can omit proof for such optimization rules, by including the keyword **sorry**, this optimization rule is regarded as proven by Isabelle. This can impact the verification of optimization rule (4.4), falsifying the optimization rule, by the following reasoning:

$$z * 1 \longmapsto z,\ (4.4) \tag{4.5}$$

$$Let\ z = (x + y) \tag{4.6}$$
$$(x + y) * 1 \longmapsto (x + y) \tag{4.7}$$
$$(x + y) * 1 \longmapsto (x + y) \longmapsto (x - y),\ \text{From (4.3)} \tag{4.8}$$
$$(x + y) * 1 \longmapsto (x - y) \tag{4.9}$$

$$Let\ x = 1,\ y = 1 \tag{4.10}$$
$$(1 + 1) * 1 \longmapsto (1 - 1) \tag{4.11}$$
$$2 * 1 \longmapsto 0 \tag{4.12}$$
$$2 \longmapsto 0 \tag{4.13}$$
$$False \tag{4.14}$$

As such, it is critical for each of the *proposed* optimization rule to be mutually exclusive with one another, only interacting with proven theories inside Veriopt's current theory base.

This is aided by Isabelle's session framework [25, Ch. 2], where each session only interacts with the imported theorems and lemmas inside of it, equivalent to Isabelle sessions defined for Isabelle/jEdit. As such, optimization rules are analyzed by spawning a session for each of the rules and commands, depending on the current state of the analysis.

4.3 Interfacing with Isabelle

Regarding the non-functional requirement:

> "Developers of GraalVM need to be able to integrate this easily into their test suite"

While developer experience is important with concerns of developer workflow, the only concern for this project is to demonstrate the full capabilities of the analysis done inside VeriTest. Providing capabilities of a language server protocol (LSP) would be challenging, as the grammar for Veriopt's DSL is not formalized yet (See Chapter 5). Thus, the extent of *how* GraalVM would integrate VeriTest into their test suite is out of the scope of this project. As such, it is imperative that VeriTest provides a *generic* interface towards analyzing optimization rules.

```
interface IsabelleProcessInterface extends Closeable:
    (BlockingQueue, LockCondition) open()
    TaskId submitTask(TaskType, args)
```

Figure 4.3: Pseudocode for Isabelle Process Interface

We utilize the Isabelle Client-Server interface to interact with Isabelle. Interacting with Isabelle Server can be done in several ways: through Isabelle Client, and through a TCP socket [25, Ch. 4]. For the purpose of VeriTest, we utilize Isabelle Client to simplify the implementation. Because of the possible alternatives, VeriTest abstracts the interactions through `IsabelleProcessInterface`, as outlined in Figure 4.3. Isabelle Process component outlined in Figure 4.1 implements this interface, and acts as a proxy for Isabelle Client.

```
interface IsabelleClient extends Closeable:
    Async<Task> startSession(request);
    Async<Task> stopSession(request);
    Async<Task> useTheory(request);
```

Figure 4.4: Pseudocode for Isabelle Client

Furthermore, as any analysis done would only concern *submitting* an optimization rule to analyze, VeriTest generalizes the interaction through `IsabelleClient` interface, as illustrated in Figure 4.4. This interface generalizes any type of commands that would be invoked in Isabelle Server [25, Sec. 4.4]. This allows other components to focus on the implementation of the analysis, rather than worrying about how to interface with Isabelle.

```
abstract class AbstractIsabelleClient implements IsabelleClient:
    var Isabelle: IsabelleProcessInterface;

    Async<Task> startSession(request):
        return Isabelle.submitTask(START_SESSION, request)
            then waitForCompletion()
            then (TaskId) -> getResult(TaskId)
            then switch (result):
                case SuccessTask response -> return response;
                case ErrorTask error -> throw error;

    ...
```

Figure 4.5: Pseudocode for Abstract Isabelle Client

As any interaction with Isabelle would involve the procedures of `IsabelleClient`, we utilize a bridge pattern to abstract Isabelle's interaction, as seen in Abstract Isabelle Client in Figure 4.1. The `AbstractIsabelleClient`[3] defines the procedures of how to interact with Isabelle, utilizing Isabelle Process, which can be seen in Figure 4.5. `stopSession()` and `useTheory()` procedures use the same algorithm as `startSession()` to define their interactions. Note that `then` keyword represent asynchronous function compositions. `waitForCompletion()` waits for the asynchronous functions to complete. `getResult()` is a procedure that handles demultiplexing tasks that allows for concurrent processes, which is elaborated in Section 4.4.

It is worthy to consider that Isabelle responses in the form of `Task` represents various types of responses from `startSession()`, `useTheory()`, `stopSession()`, and even miscellaneous error messages from Isabelle [25, Sec. 4.4]. Each task would contain at least three fields: (1) their task identifier; (2) the type of message being sent, whether the corresponding task is finished or have errors; and (3) a list of messages from Isabelle. Through pattern matching Isabelle's messages, we can classify the result of the analysis, as elaborated on Section 4.5. We parse Isabelle responses in the form of a string with a customized polymorphic deserializer called `TaskDeserializer`[4], which utilizes Jackson's `ObjectMapper` [26] – that transforms a string into a abstract JSON tree – with added functionality through Java Reflections API. This method of parsing is used extensively in Figure 4.7 that elaborates how to interface with Isabelle Client's subprocess.

Through this approach to interface with Isabelle, we provide modularity and extensibility for VeriTest, which would immensely assist GraalVM developers integration to their test suite and future work to be done.

4.4 Parallel Execution

To support VeriTest's goal of providing fast analysis towards optimization rules, VeriTest utilizes concurrent processing by using asynchronous functions. The implementation of Isabelle Process and Abstract Isabelle Client utilizes a producer-consumer pattern, ensuring that interactions towards Isabelle Client would not be a bottleneck for other optimization rules.

As Isabelle Client is a subprocess for VeriTest, we can split the input and output stream to efficiently process command invocations. The input stream would act as the producer towards

[3] AbstractIsabelleClient.java & IsabelleProcess.java extending `AbstractIsabelleClient`

[4] TaskDeserializer.java

```
class IsabelleProcess implements IsabelleProcessInterface:
    var processQueue : BlockingQueue;

    (BlockingQueue, LockCondition) open():
        startIsabelleSubprocess();
        queue := new BlockingQueue();

        var daemon := Daemon();
        daemon.processQueue := queue;
        daemon.notEmpty := new LockCondition();
        daemon.asyncQueue := new ReentrantLock();

        return (daemon.asyncQueue, daemon.notEmpty);

    synchronized TaskId submitTask(TaskType, args):
        writeToProcess(TaskType, args);

        var taskResponse := this.syncQueue.waitAndTake();

        return taskResponse.taskId;

    ...
```

Figure 4.6: Pseudocode for writing to process

```
class Daemon:
    var processQueue : BlockingQueue;

    var notEmpty : LockCondition;
    var asyncQueue : BlockingQueue;

    void run():
        while process isAlive:
            var string := readFromProcess();

            var response := TaskDeserializer.parse(string);
            switch (response):
                case ImmediateTask -> this.processQueue.waitAndPut(response);
                case AsyncTask -> {
                    this.asyncQueue.put(response);
                    signalAllCondition(this.notEmpty);
                }
```

Figure 4.7: Pseudocode for reading from process

the BlockingQueue, while the output stream would be continuously consumed by a separate thread – as illustrated by Figure 4.6 and 4.7. This ensures that command invocations are only IO-bound by the input stream.

Isabelle Server has two types of responses that originate from command invocations: immediate responses and asynchronous responses [25, Sec 4.2.6]. Immediate responses denote the task identifier to differentiate between asynchronous responses, which is used by

`IsabelleProcess`[5]. Asynchronous responses denote the actual progress of the command invocation, identified by their task identifier. As we can see in Figure 4.7, the types of responses are differentiated and sent to different queues.

```
class AbstractIsabelleClient implements IsabelleClient:
    var Isabelle: IsabelleProcessInterface;

    ...

    var asyncQueue, notEmpty := Isabelle.open()

    Async<Task> getResult(taskId):
        while true:
            while asyncQueue is empty:
                waitCondition(notEmpty);

            if asyncQueue.peek().taskId == taskId:
                return asyncQueue.take();
```

Figure 4.8: Pseudocode for getting asynchronous results

Asynchronous responses are consumed by the `getResult()` method as illustrated by Figure 4.8. Sufficient locking mechanisms without spinning the locks are done to ensure that concurrent processes accessing the same queue would not lead to starvation. As the queue itself is a `BlockingQueue`, and the purpose of the procedure is only to demultiplex the asynchronous responses, merely checking if the task has the correct identifier is enough to ensure mutual exclusion of concurrent processes.

```
Async<Void> allOfReturnOnSuccess(List<Async<Task>> futures):
    var futureResult := allOf(futures);

    for future in futures:
        future()
            then (result) -> {
                /* If the future is finished, and result is not failed, then terminate
                    allOf() */
                if future is successful && result.status != FAILED:
                    futureResult.complete();
            }

    return futureResult;
```

Figure 4.9: Pseudocode for circuit-breaker asynchronous functions

Asynchronous functions are managed by a thread pool, separating IO bound processes into a different executor to ensure freedom from threadpool-induced deadlock. Furthermore, asynchronous functions implement a circuit-breaker pattern when a successful result is available from the function, which can be seen in Figure 4.9. Through this, VeriTest is able to provide

[5]IsabelleProcessFacade.java

a *comparatively* fast analysis for optimization rules with only Isabelle as a bottleneck (See chapter 5).

4.5 Generating the Analysis

VeriTest provides an API that encapsulates all the analysis that would be done towards the optimization rule in order to provide the functional requirements of VeriTest, and satisfy the non-functional requirement:

"Developers of GraalVM can easily use this without understanding Isabelle."

The analysis is done by invoking Isabelle's automated tools and pattern matching their response into their classification. The order of which the analysis is carried out can be seen in Figure 4.10. After significant evaluation and analysis, we extended their classification into multiple categories (See Figure 4.11).

4.5.1 Sledgehammer Invocations

It is interesting to note that there are multiple invocations of Sledgehammer in order to find proofs towards an optimization rule. This is due to the fact that Sledgehammer can both return a partial proof towards an optimization rule due to the two proof obligations needed to prove an optimization rule, and return several proof options that can prove an optimization rule. A recursive algorithm is used in order to comprehensively prove an optimization rule, as illustrated by Figure 4.12.

4.5.2 Finding Counterexamples through Nitpick & Quickcheck

Finding counterexamples are much more straightforward compared to Sledgehammer invocations. If any of the messages inside Isabelle's response contains a counterexample string, then a counterexample is found. This algorithm works with both Nitpick and Quickcheck invocations. As such, we can use an algorithm such as Figure 4.13 to find incorrect optimization rules.

4.5.3 Detecting Malformed Optimization Rules and Automatic Proof

While we utilize Isabelle's automated tools to analyze our optimization rules, invoking a time consuming tool to detect malformed optimization rules is unnecessary. For some of the optimization rules (e.g., Fig. 4.14), the rule is automatically verified by Isabelle due to it not having any necessary proof obligations to prove. As such, we can leverage this fact to invoke a quick and efficient automated tool in the form of a **dot** (.) that only does the reasoning internally. An algorithm such as Figure 4.15 allows us to check for malformed optimization rules, type unification errors, and other types of errors.

4.6 Dependencies and Containerization

VeriTest is available as a containerized application[6] ready for immediate use through Docker [27]. All of the required dependencies of VeriTest are built through multiple build stages in

[6]https://hub.docker.com/repository/docker/achmadafriza/veritest

order to reduce the time needed to build VeriTest through build caching. A critical component of VeriTest is utilizing Isabelle2023's Docker image to avoid rebuilding Isabelle from scratch [25, Sec. 7.1].

Furthermore, VeriTest pre-builts all of the necessary lemmas and theorems from Veriopt's theory base in order to reduce the time needed to start a session for analyzing an optimization rule. It is important to note that Isabelle sessions needs to be built in the *correct order*, beginning with Isabelle/HOL, dependencies of the Canonicalizations session from Veriopt's theory base, to the Canonicalizations session. The detail of the containerization can be seen in Appendix A.

4.7 Supporting Tool

VeriTest also improves Veriopt's current extraction tool [4, Sec. 7] by creating an extraction script capable of extracting source code annotations inside GraalVM's compiler and extracting Veriopt's existing optimization rules inside their theory base. This extraction tool also serves as a test generation script for VeriTest, converting extracted optimization rules into JSON files. The details of this tool can be seen in Appendix B.

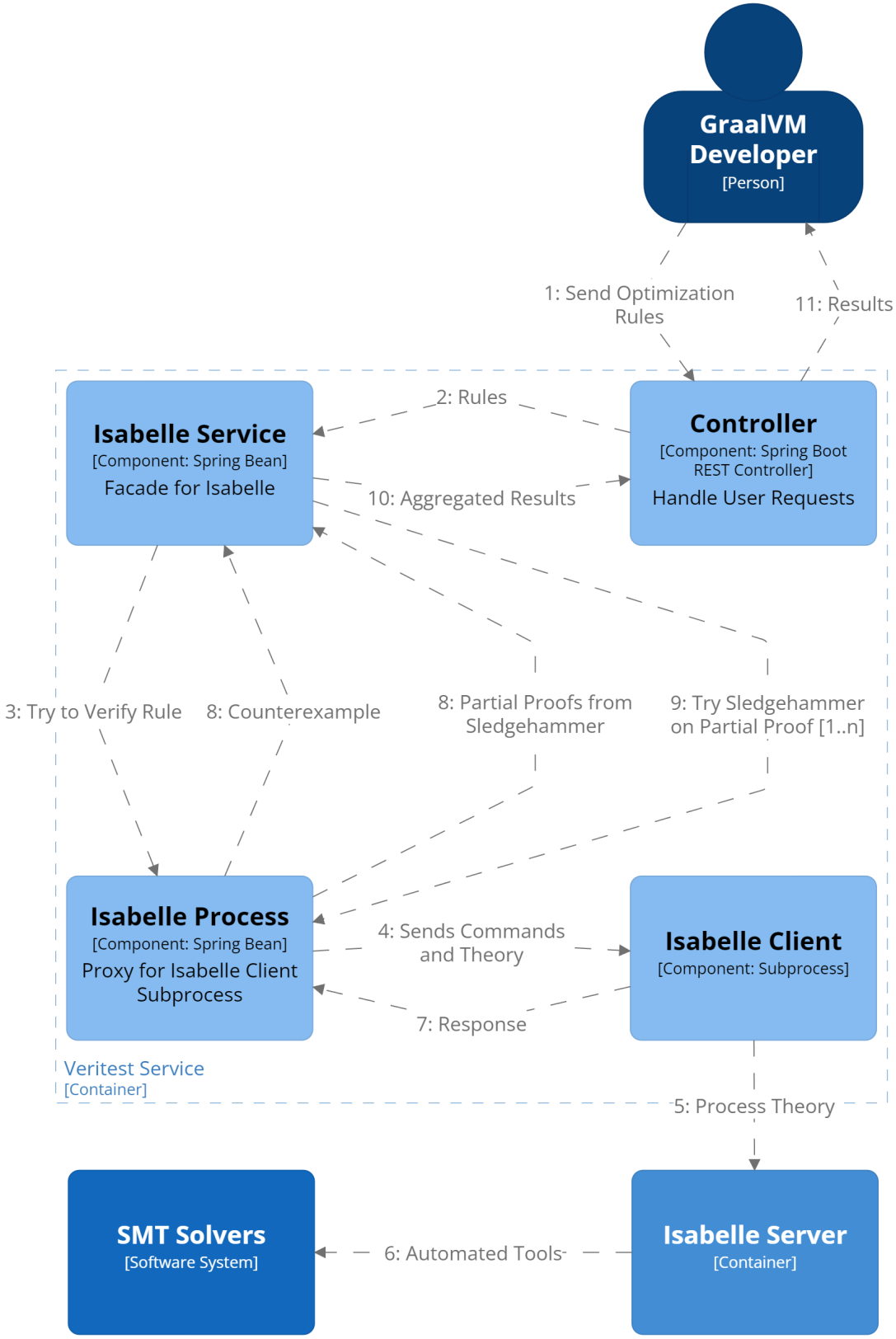

Figure 4.10: Sequence Diagram for VeriTest

1. The optimization rule is false;

   Quickcheck or Nitpick are able to find a counterexample.

2. The optimization rule is automatically proven;

   Based on the optimization rule, no proof obligation are necessary to proof that the optimization rule is true.

3. The optimization rule has no subgoals;

   The optimization rule should be able to be automatically proven, but Isabelle classified the optimization as an error. It is worthy to note that optimization rules that have no subgoals are specific to Isabelle2023, and should be amended by the next iteration of Isabelle.

4. The optimization rule is proven;

   Sledgehammer is able to prove the optimization rule.

5. The optimization rule is malformed due to syntax errors;

   The optimization rule does not follow the syntax of the DSL.

6. The optimization rule is malformed due to type errors;

   This means that the optimization rule provided has incompatible types.

7. The optimization rule cannot be classified.

   VeriTest is unable to verify the optimization rule, due to the complexity of said rule.

Figure 4.11: Extended classification of an optimization rule

```
class IsabelleService:
    var Client: IsabelleClient;

    Async<Result> trySledgehammer(Theory):
        return generateTheory(SLEDGEHAMMER, Theory)
            then Client.startSession(request)
            then Client.useTheory(generated)
            then recursiveSledgehammer(Theory, TaskResponse);

    Result recursiveSledgehammer(Theory, TaskResponse):
        /* Base Case: Errors returned from Isabelle */
        if TaskResponse contains errors:
            return FAILED;

        /* Base Case: Found Proof */
        if TaskResponse contains "No Proof State" or "No Subgoal":
            return (FOUND_PROOF, TaskResponse.proofs)

        var possibleProofs := filterProofs(TaskResponse);

        /* Base Case: Proof not found */
        if possibleProofs is empty:
            return FAILED;

        var futures : List;
        for proof in possibleProofs:
            var childRequest := generateChildRequest();
            childRequest.proofs := TaskResponse.proofs + [proof];

            futures.add(trySledgehammer(childRequest));

        allOfReturnOnSuccess(futures)
            then waitForCompletion();

        var result := filterAnySuccessfulFuture(futures);

        return result;
```

Figure 4.12: Pseudocode for Sledgehammer recursive invocations

```
class IsabelleService:
    var Client: IsabelleClient;

    Async<Result> tryNitpick(Theory):
        return generateTheory(NITPICK, Theory);
            then Client.startSession(request);
            then Client.useTheory(generated);
            then (Task) -> {
                if Task contains error:
                    return FAILED;

                if Task.messages contains counterexample:
                    return (FOUND_COUNTEREXAMPLE, counterexample);
                else:
                    return FAILED;
            };

    ...
```

Figure 4.13: Pseudocode for finding counterexamples

optimization *Auto:* $(true\ ?\ x\ :\ y) \longmapsto x$.

Figure 4.14: Example of optimization rule that can automatically be verified

```
class IsabelleService:
    var Client: IsabelleClient;

    Async<Result> tryAuto(Theory):
        return generateTheory(AUTO, Theory);
            then Client.startSession(request);
            then Client.useTheory(generated);
            then (Task) -> {
                if Task not contains error:
                    return FOUND_AUTO_PROOF;

                if Task.messages contains "syntax error" or "undefined type":
                    return MALFORMED;

                if Task.messages contains "type unification error":
                    return TYPE_ERROR;

                if Task.messages contains "no subgoal":
                    return NO_SUBGOAL;

                return (FAILED, Task.messages);
            };

    ...
```

Figure 4.15: Pseudocode for finding malformed optimization rules

Chapter 5

Results & Discussion

In order to determine the thoroughness of VeriTest's analysis, we evaluated VeriTest based on Veriopt's current, *proven*, collection of optimization rules inside their theory base [4]. Furthermore, as GraalVM's compiler source code is currently annotated with optimization rules following Veriopt's DSL [4, Sec. 5.1], we can evaluate VeriTest by evaluating the source code annotations through VeriTest.

Evaluation is done by exporting the existing Veriopt's expression canonicalization optimization rules using VeriTest's extraction tool (See Sec. 4.7) and iterating them several times as a benchmark on an AMD Ryzen 9 7900X processor with 64GB of RAM inside a Windows 10 WSL environment.

5.1 Evaluation of Veriopt's Current Theory Base

Result	#Rules	Mean ± *SD*
Failed	59	87.33 ± 49.80
Found Auto Proof	7	37.73 ± 3.92
Found Proof	32	82.91 ± 29.43
Found Counterexample	1	40.79 ± 4.02
Malformed	13	37.96 ± 4.02
No Subgoal	2	37.93 ± 3.38

Table 5.1: Evaluation of each existing Veriopt optimization rules based on runtime (in seconds)

Our findings suggest that there is a baseline runtime for every verification of an optimization rule. This is because, surprisingly, there is a key flaw inside Isabelle Server. Kobschätzki [28] notes that Isabelle Server triggers a race condition with multiple users and sessions, unless a delay of several seconds is placed in between subsequent commands. Isabelle Server's source code also confirms that each command invocation doesn't support concurrent use. Since every command invocation is preceded – and followed by – session invocations, this flaw inside Isabelle Server presents a bottleneck for processing optimization rules. On the condition that such flaw does not exist, the baseline runtime for verifying optimization rules could be reduced by approximately 24-36 seconds or possibly more – depending on the depth of the recursion tree.

Surprisingly, some of the existing optimization rules are malformed. While it is a commented out rule – and it is malformed due to a minor syntax error (i.e., `\mapsto` instead of

`\longmapsto`) – it is interesting to see that it has type unification errors over the IR expression. This modification is included inside our mutation tests, which is described in Section 5.2.

optimization *SubNegativeConstant:* $x - (const\ (val[-y])) \longmapsto x + (const\ y)$

1. $False$
2. $BinaryExpr\ BinSub\ x\ (ConstantExpr\ (intval\_negate\ y)) \sqsupseteq BinaryExpr\ BinAdd\ x\ (ConstantExpr\ y)$

Figure 5.1: Optimization rule that VeriTest found a counterexample

A counterexample is found for one of the existing optimization rule, as illustrated in Figure 5.1. While it is also on a commented out rule, Isabelle is able to reason that the termination proof obligation would be simplified to *False*. This is due to the fact that an expression termination is represented by a measure function, *trm* of type $IRExpr \Rightarrow nat$ [4, Sec. 3.3]. $val[e]$ effectively lowers an IR expression into an equivalent value. Isabelle can immediately reason that the size of an equivalent value of an expression $-y$ is not equal to the expression y. This implies that the optimization rule will not terminate on any context of an optimization phase.

The evaluated runtime suggests a significant variation of runtime for some results. This is due to VeriTest attempting to exhaust every possible proof and counterexample for the optimization rule, as illustrated by Section 4.5. As the complexity of optimization rules varies, the depth of the recursive tree depends on the proof obligations that Sledgehammer can prove. Failure to find a proof suggests that it *may* be able to be proven with adequate expertise in formal proofs.

Finally, we found that VeriTest is capable of utilizing existing lemmas inside Veriopt's current theory base to find proofs for the optimization rule. While we need to consider if the usage of such lemmas are proper – as it may use incomplete lemmas (See Section 5.1.1) – this also implies that, as the collection of theories grows, the percentage of optimization rules that can be automatically proven is expected to improve.

5.1.1 Incomplete Proofs

We have discovered that *some* of the lemmas are in fact only partially proven and are still on progress, due to the fact that Veriopt [4] are still in development of proving GraalVM's optimizations. This introduces a moderate drawback for the goals of VeriTest, as incorporating incomplete lemmas would effectively lead to an incorrect optimization rule.

Using incomplete proofs to prove an optimization rule can be detected by utilizing oracles through *thm_oracles* [29, Sec. 5.14]. Oracles works by leveraging external reasoners to produce arbitrary Isabelle theorems treated as an axiom. This is particularly useful as incomplete proofs would then invoke such oracles and mark the arbitrary Isabelle theorem used to skip proof obligations.

However, we found that the behaviors of the oracle invoked directly inside Isabelle and inside Isabelle Server has discrepancies between them. Verifying Figure 5.2 inside Isabelle would not invoke any external oracle, as it already proved all required proof obligations. Even so, when invoked inside Isabelle Server, the oracle would produce an arbitrary Isabelle theorem and skip the necessary proof obligation.

optimization *AddNeutral*: $(e\ +\ const\ (IntVal\ 32\ 0)) \longmapsto e$
using AddNeutral_Exp by presburger
thm_oracles *AddNeutral*

Figure 5.2: Example of an Isabelle Oracle

While the reasons for this is inconclusive, this is possibly a bug inside Isabelle, tracing the exact reasons inside Isabelle would prove to be a challenge. As such, we were unable to determine which of the existing theorems verified by VeriTest are incorporating incorrect lemmas.

A possible remedy to the limitations of Isabelle Server would be to have a list of such unproven lemmas inside VeriTest so that any proofs found from VeriTest can be matched and excluded. However, building such list only serves as a band-aid for the problem, and should not be treated as the answer. Enhancing VeriTest to consider usage of incorrect lemmas would represent a future refinement to the tool.

5.2 Evaluation of Malformed Rules

Result	#Rules	Mean ± *SD*
Failed	25	80.95 ± 18.53
Found Proof	3	78.17 ± 16.20
Found Counterexample	2	56.22 ± 25.16
Malformed	1	35.98 ± 0
Type Errors	14	33.47 ± 4.82

Table 5.2: Evaluation of each mutated optimization rules based on runtime (in seconds)

In order to determine the completeness of VeriTest's analysis in finding incorrect optimization rules, we devised a suite of test cases describing malformed optimization rules. These malformed rules are generated through mutation operators devised by Offutt et al. [16]. The mutated optimization rules should represent possible scenarios where a GraalVM developer might make mistakes in the syntax analogous to the Java syntax (i.e., using a short-circuiting logical or (`||`) instead of bitwise or (`|`)). The list of mutation operators applied and the mutated optimization rules can be seen in Appendix C. The results for the evaluation can be seen in Table 5.2.

optimization *condition_bounds_x_1:* $((u\ >\ v)\ ?\ x\ :\ y) \longmapsto x$ (5.1)

Peculiarly, only one mutated optimization rule is regarded as a malformed optimization rule, while others are blended into the classification of a complex rule – due to the limitations of Nitpick and Quickcheck – or rules that have type unification errors within them. An example of them are type errors caused by utilizing $>$ and $\geq$, as illustrated by (5.1). Both of the operators do not exist inside GraalVM's IR, but do exist in Isabelle as an operator that returns a *bool*, which cannot be transformed into an *IRExpr*.

$$\textbf{optimization}\ \textit{EliminateRedundantFalse\_1:}\ x \mid \mathit{true} \longmapsto x \tag{5.2}$$

$$\textbf{optimization}\ \textit{EliminateRedundantFalse\_2:}\ x \mid\mid \mathit{true} \longmapsto x \tag{5.3}$$

Furthermore, there are some ambiguity to the grammar of Veriopt's DSL, particularly with *BinShortCircuitOr* (||), *BinOr* (|) – as exemplified by (5.2). (5.3) should be regarded as a *syntactically* correct optimization rule, but type errors exist due to Isabelle regarding || as two *BinOr* nodes.

$$\begin{aligned}\textbf{optimization}\ \textit{flipX:}\ &((x\ \mathit{eq}\ (\mathit{const}\ (\mathit{IntVal}\ 32\ 1)))\ ?\\ &(\mathit{const}\ (\mathit{IntVal}\ 32\ 1))\ :\ (\mathit{const}\ (\mathit{IntVal}\ 32\ 0)))\\ &\quad\longmapsto x \oplus (\mathit{const}\ (\mathit{IntVal}\ 32\ 1))\\ &\mathit{when}\ (x = \mathit{const}\ (\mathit{IntVal}\ 32\ 0)) \mid\\ &\quad (x = \mathit{const}\ (\mathit{IntVal}\ 32\ 1)))\end{aligned} \tag{5.4}$$

This ambiguity is further exacerbated by the use of *const* keyword and *ConstantExpr* node, as seen in (5.4). A *const* keyword is just a syntactic sugar for a *ConstantExpr*. However, it cannot be used interchangeably inside a precondition of an optimization rule. A translation function *exp*[] needs to be used to mark the expression as an *IRExpr*. Furthermore, it is not clear when the translation function needs to be used, as opposed to just adding parenthesis.

$$\textbf{optimization}\ \textit{MulNeutral:}\ x * \mathit{ConstExpr}\ (\mathit{IntVal}\ b\ 1) \longmapsto x \tag{5.5}$$

The ambiguity extends to the definitions of an *IRExpr*. As *ConstExpr* is not included in the definition of *IRExpr*, Isabelle regarded *ConstExpr* as a function of type $Value \Rightarrow IRExpr$ as seen in (5.5). While this is syntactically correct, it would mean that the resulting IR expression would not conform to the semantics of GraalVM's IR.

These findings indicate that the syntax of Veriopt's DSL may be ambiguous and rather unclear. This represents an opportunity for future research to formalize the grammar of the DSL, possibly adding a parser for the DSL, to reduce the ambiguity so that it is easier for GraalVM's developers to adopt and integrate it into their development workflow.

5.2.1 Limitations for Finding Counterexamples

Although VeriTest is capable of finding syntax and type unification errors, our investigation discovered that VeriTest is unable to discover counterexamples for *specifically* the term refinement proof obligation of the mutated optimization rules. Even though it may seem trivial intuitively, it does not appear to be so for Isabelle. While the reasons for it are inconclusive, and are in fact beyond the scope of this project, we discuss several factors that might explain *why* this happens.

$$e_1 \sqsupseteq e_2 = (\forall\ m\ p\ v.\ [m,\ p] \vdash e_1 \mapsto v \longrightarrow [m,\ p] \vdash e_2 \mapsto v)$$

Figure 5.3: Term refinement proof obligation, Adapted from [4, Definition 6]

For termination proof obligation, it is trivial for Isabelle to reason with it due to working over natural numbers. However, for an optimization rule's term refinement proof obligation,

illustrated by Figure 5.3, it is significantly more complex. Isabelle needs to be able to transform the proof obligation into conjectures that it can reason with.

Revisiting Quickcheck's limitations (See Sec. 2.2.4), it is possible that Quickcheck's exhaustive test generator is unable to transform complex conjectures into datatypes that it can generate tests for, even when appropriate constructors are defined. For conditional conjectures for each optimization rule, such as Figure 5.3, Quickcheck exhausts all possible values of IR expressions that define the context of $[m,\ p]$ at the expression level, which it fails to do so.

$$val[e_1]\ \neq\ UndefVal \wedge val[e_2]\ \neq\ UndefVal \longrightarrow\ val[e_1]\ =\ val[e_2]$$

Figure 5.4: Conditional evaluation of e_1 and e_2

Instead of exhaustively enumerating all the possible contexts for an expression, it may be possible for Quickcheck to conditionally evaluate the given conjecture as a value, as illustrated in Figure 5.4. As long as e_1 and e_2 are not undefined, the values of e_1 and e_2 are compared to determine equality. This semi-automatic method of conditionally evaluating expression semantics would need VeriTest to parse GraalVM's IR DSL, statically analyze and construct appropriate and equivalent theorems to verify it, which may represent potential future work.

In the case of Nitpick, extensive debugging suggests that there may be an inherent flaw inside the encoding of GraalVM's expression semantics. Inside Veriopt's DSL, the encoding for GraalVM's IR expressions is defined as functions and datatypes that influence how Isabelle reasons about it. Such definitions are used to translate optimization rules into conjunctions for SMT solvers to find satisfiability.

However, we found that among the translated conjunctions, especially for the term refinement proof obligation, Nitpick is unable to support *representative functions* to properly map an Integer quotient type [18, Sec. 3.7] into the respective first-order relational logic [18, Ch. 8]. This is explicitly mentioned as a known bug inside Nitpick, and should be amended by defining a *term postprocessor* that converts such quotient type into a standard mathematical notation [18, Sec. 3.7] or correcting underspecified functions [18, Ch. 8]. To quote from Blanchette [18, Ch. 8]:

> "Axioms or definitions that restrict the possible values of the undefined constant or other partially specified built-in Isabelle constants (e.g., *Abs_* and *Rep_* constants) are in general ignored. Again, such nonconservative extensions are generally considered bad style."

5.3 Evaluation of GraalVM's Source Code Annotations

While the evaluation of existing GraalVM source code annotations (See Table 5.3) is in line with the evaluation of Veriopt's existing optimization rules, one thing that stood out is the presence of significantly more optimization rules that have type unification errors. This can be attributed to the factors discussed in Section 5.2, where it is noted that the grammar of Veriopt's DSL can be rather ambiguous. This would also prove to be a challenge for GraalVM's developers to effectively integrate the DSL into their development workflow.

Integration would mean that the capabilities of VeriTest are extended into a language server protocol (LSP) to provide feedback while processing source code annotations in the background (See Section 2.4). However, extending VeriTest into an LSP would require the DSL's grammar to be formalized such that a typical parser would be able to construct an abstract syntax tree (AST) for the DSL. Building such an AST would also mean that VeriTest can be extended to

Result	#Rules	Mean ± *SD*
Failed	12	73.78 ± 4.66
Found Auto Proof	1	29.22 ± 0
Found Proof	19	62.19 ± 25.20
Found Counterexample	3	40.79 ± 5.10
Malformed	21	38.02 ± 5.16
Type Error	40	37.85 ± 5.43

Table 5.3: Evaluation of each existing GraalVM source code annotation based on runtime (in seconds)

incorporate some metrics of static analysis, allowing for an intelligent evaluation by building equivalent theorems (i.e., Figure 5.4) that would allow Isabelle to reason better.

5.4 Challenges

File Extension	Total LoC
.java (Java classes)	1751
.md (Markdown documents)	1446
.thy (Isabelle theory files)	952
.pl (Perl scripts)	60
.py (Python scripts)	56
.sh (Bash scripts)	52

Table 5.4: Total lines of code (LoC) written for VeriTest

Although the volume of code written reflects the efforts invested in this project, as illustrated by Table 5.4, it does not adequately capture the full extent of the effort expended. Much of the effort is put into experimenting and capturing the full extent of the capabilities of Isabelle Server [25] and their viability as a solution to this project. As a result of our due diligence, we discovered several bugs in Isabelle and identified future considerations for Veriopt's DSL [4].

Furthermore, exploring the viability of Isabelle Server also meant that exploring the source code of Isabelle is necessary, as Isabelle's documentation is not always the most descriptive. Isabelle's source code is complex and challenging to read, which highlights the considerable effort invested in this project. This also proved to be a challenge for packaging Isabelle into a Docker container (See Section 4.6), as the dependencies of Isabelle are not explicitly stated anywhere in the documentation.

Moreover, careful considerations and analysis are made to ensure that VeriTest implements a mutually exclusive and concurrent analysis of the optimization rule, ensuring freedom from starvation for each invocation (See Section 4). An elegant and well-crafted implementation is not necessarily reflected in the sheer volume of code written, but rather in the simplicity and modularity of the solution.

Chapter 6

Conclusion

In order to provide solver-based tools for identifying incorrect optimizations, we introduced VeriTest: a software interface for automatic verification of GraalVM's expression optimizations. The main goal of this project is to integrate VeriTest into the GraalVM development workflow to assist developers in verifying proposed optimization rules without requiring expertise in formal proof systems. Through leveraging Isabelle and their automated tools such as Sledgehammer, Nitpick, and Quickcheck, VeriTest demonstrated its capability to detect malformed and incorrect optimization rules, and provide automatic verification of optimization rules.

Through the design patterns and algorithms that VeriTest has implemented, it has demonstrated its capability to provide fast and comprehensive analysis for verifying optimization rules. Producer-consumer pattern and sufficient mutual exclusion locks implemented inside ensured efficient processing for Isabelle invocations without bottlenecks. Furthermore, VeriTest displayed its extensibility for future refinement through its use of bridge and facade design patterns – abstracting Isabelle interactions and minimizing tight-coupling.

While the evaluation of VeriTest on Veriopt's current theory base and GraalVM's source code annotation suggests that VeriTest cannot replace manual verification, it has demonstrated its potential in automatically verifying optimization rules; implying that the percentage of optimization rules that *can* be automatically proven will improve as the collection of theories grows. Limitations were identified, particularly in VeriTest's ability to find counterexamples, due to the inherent limitations of Nitpick and Quickcheck. Despite these challenges, the tool provided valuable insights to previous works done and acts as a starting point for further refinement and expansion.

6.1 Possible Future Work

Due to the limitations of VeriTest in finding counterexamples, a potential direction for VeriTest's development is to enhance its capabilities and address the limitations found. To implement a more robust method of finding counterexamples, VeriTest can be extended with an interpreter capable of statically analyzing the semantics of Veriopt's domain specific language and transform it into a more robust theory to analyze – possibly lowering intermediate representation expressions into an equivalent value or binary operations – similar to Figure 5.4. Such interpreter would also benefit Veriopt to provide a less ambiguous grammar for the domain specific language. However, certain expressions might have complex side-conditions, which the analysis would need to consider.

Additionally, Isabelle's Quickcheck and Nitpick could be customized to suit the needs of

GraalVM's intermediate representation by extending Isabelle/Scala and Isabelle/ML. This would greatly enhance VeriTest's capabilities, while still utilizing the framework that VeriTest provided.

Furthermore, the theory base for GraalVM's intermediate representation should be extended to represent a more comprehensive baseline for verifying expression optimizations. Currently, theorems and lemmas inside Veriopt are created specifically for verifying one optimization rule. A more generalized and comprehensive theory base would immensely improve VeriTest's capabilities.

Analyzing the usage of incomplete and unproven lemmas could represent a future iteration for VeriTest. This could be done either by identifying the cause of bug for Isabelle's oracle inside Isabelle Server, or having a list of unproven lemmas that can be matched. This would allow VeriTest to enhance the comprehensiveness of its analysis.

Another key fact that VeriTest demonstrated is that it is a *generalized* interface for interacting with Isabelle. As such, VeriTest can be extended to automatically verify different compilers, given a *sufficient* formal specification for its intermediate representation. Furthermore, the algorithms used in VeriTest to ensure parallel processing and mutual exclusion can be used by Isabelle to enhance Isabelle Server's capabilities and alleviate its current limitations.

To support VeriTest's integration towards GraalVM's development workflow, it is necessary for VeriTest to be catered towards *how* the developers would interface with VeriTest. This can be in the form of syntax highlighting inside a code editor by extending VeriTest into a language server protocol, or through a web interface that can display the analysis for optimization rules in a more human-readable form.

Overall, while VeriTest has shown promising results, continued development is necessary to fully realize its potential as a comprehensive tool for automatic verification of compiler optimizations.

Chapter 7

Bibliography

[1] N. P. Lopes and J. Regehr, "Future directions for optimizing compilers," arXiv preprint, Sep. 2018. arXiv: 1809.02161 [cs.PL].

[2] N. J. Wahl, "An overview of regression testing," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 1, pp. 69–73, Jan. 1999, ISSN: 0163-5948. DOI: 10.1145/308769.308790.

[3] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/1538788.1538814.

[4] B. J. Webb, I. J. Hayes, and M. Utting, "Verifying term graph optimizations using Isabelle/HOL," in *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2023, pp. 320–333, ISBN: 9798400700262. DOI: 10.1145/3573105.3575673.

[5] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic* (Lecture Notes in Computer Science). Berlin, Heidelberg: Springer, 2002, vol. 2283, ISBN: 3-540-43376-7. DOI: 10.1007/3-540-45949-9.

[6] J. Lee, C.-K. Hur, and N. P. Lopes, "AliveInLean: A verified LLVM peephole optimization verifier," in *Computer Aided Verification*, I. Dillig and S. Tasiran, Eds., 2019, pp. 445–455, ISBN: 978-3-030-25543-5. DOI: 10.1007/978-3-030-25543-5_25.

[7] N. P. Lopes, J. Lee, C.-K. Hur, Z. Liu, and J. Regehr, "Alive2: Bounded translation validation for LLVM," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, Virtual Canada: ACM, Jun. 2021, pp. 65–79, ISBN: 978-1-4503-8391-2. DOI: 10.1145/3453483.3454030.

[8] Oracle, *GraalVM: Run programs faster anywhere*, 2020. [Online]. Available: https://github.com/oracle/graal (visited on 09/13/2023).

[9] B. J. Webb, M. Utting, and I. J. Hayes, "A formal semantics of the GraalVM intermediate representation," in *Automated Technology for Verification and Analysis*, Z. Hou and V. Ganesh, Eds., ser. LNCS, vol. 12971, Oct. 2021, pp. 111–126, ISBN: 978-3-030-88885-5. DOI: 10.1007/978-3-030-88885-5_8.

[10] F. Huch and V. Bode, *The Isabelle Community Benchmark*, arXiv:2209.13894 [cs], Sep. 2022. DOI: 10.48550/arXiv.2209.13894.

[11] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds., *Structured programming*. GBR: Academic Press Ltd., 1972, ISBN: 978-0-12-200550-3.

[12] W. M. McKeeman, "Differential Testing for Software," vol. 10, no. 1, pp. 100–107, 1998.

[13] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, New York, NY, USA: Association for Computing Machinery, Jun. 2011, pp. 283–294, ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993532.

[14] L. Bulwahn, "The New Quickcheck for Isabelle," in *Certified Programs and Proofs*, C. Hawblitzel and D. Miller, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2012, pp. 92–108, ISBN: 978-3-642-35308-6. DOI: 10.1007/978-3-642-35308-6_10.

[15] J. C. Blanchette, L. Bulwahn, and T. Nipkow, "Automatic Proof and Disproof in Isabelle/HOL," in *Frontiers of Combining Systems*, ser. LNCS, C. Tinelli and V. Sofronie-Stokkermans, Eds., vol. 6989, 2011, pp. 12–27, ISBN: 978-3-642-24363-9 978-3-642-24364-6. DOI: 10.1007/978-3-642-24364-6_2.

[16] J. Offutt, P. Ammann, and L. Liu, "Mutation Testing implements Grammar-Based Testing," in *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, Raleigh, NC, USA: IEEE, Nov. 2006, pp. 12–12, ISBN: 978-0-7695-2897-7. DOI: 10.1109/MUTATION.2006.11.

[17] J. C. Blanchette, S. Böhme, and L. C. Paulson, "Extending Sledgehammer with SMT Solvers," *Journal of Automated Reasoning*, vol. 51, no. 1, pp. 109–128, Jun. 2013, ISSN: 0168-7433, 1573-0670. DOI: 10.1007/s10817-013-9278-5.

[18] J. Blanchette, "A Users Guide to Nitpick for Isabelle/HOL," [Online]. Available: https://isabelle.in.tum.de/dist/Isabelle2024/doc/nitpick.pdf (visited on 06/01/2024).

[19] *The LLVM Compiler Infrastructure Project*. [Online]. Available: https://llvm.org/ (visited on 08/20/2023).

[20] J. Biendarra, J. Blanchette, M. Desharnais, *et al.*, "Dening (Co)datatypes and Primitively (Co)recursive Functions in Isabelle/HOL," May 2024. [Online]. Available: https://isabelle.in.tum.de/dist/doc/datatypes.pdf (visited on 05/26/2024).

[21] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *IEEE Software*, vol. 20, no. 5, pp. 42–45, Sep. 2003, Conference Name: IEEE Software, ISSN: 1937-4194. DOI: 10.1109/MS.2003.1231150.

[22] H. Bünder, "Decoupling Language and Editor - The Impact of the Language Server Protocol on Textual Domain-Specific Languages," in *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development*, ser. MODELSWARD 2019, Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda, Feb. 2019, pp. 129–140, ISBN: 978-989-758-358-2. DOI: 10.5220/0007556301290140.

[23] Jetbrains, *IntelliJ IDEA the Leading Java and Kotlin IDE*. [Online]. Available: https://www.jetbrains.com/idea/ (visited on 06/01/2024).

[24] SonarSource, *Code Quality Tool & Secure Analysis with SonarQube*. [Online]. Available: https://www.sonarsource.com/products/sonarqube/ (visited on 06/01/2024).

[25] M. Wenzel, "The Isabelle System Manual," [Online]. Available: https://isabelle.in.tum.de/dist/Isabelle2024/doc/system.pdf (visited on 05/06/2024).

[26] FasterXML, *ObjectMapper (jackson-databind 2.7.0 API)*. [Online]. Available: https://fasterxml.github.io/jackson-databind/javadoc/2.7/com/fasterxml/jackson/databind/ObjectMapper.html (visited on 05/28/2024).

[27] Docker Inc., *Docker: Accelerated Container Application Development*, May 2022. [Online]. Available: https://www.docker.com/ (visited on 05/28/2024).

[28] J. Kobschätzki, *Unexpected Behavior with Isabelle Server 2023*, Jan. 2024. [Online]. Available: https://lists.cam.ac.uk/sympa/arc/cl-isabelle-users/2024-01/msg00006.html (visited on 05/06/2024).

[29] M. Wenzel, "The Isabelle/Isar Reference Manual," [Online]. Available: https://isabelle.in.tum.de/dist/Isabelle2024/doc/isar-ref.pdf (visited on 05/31/2024).

Appendices

Appendix A

Containerization of VeriTest

```
FROM gradle:8.6.0-jdk21-graal AS GRADLE_CACHE

RUN mkdir -p /home/gradle/cache_home

ENV GRADLE_USER_HOME /home/gradle/cache_home
COPY build.gradle /home/gradle/java-code/
COPY settings.gradle /home/gradle/java-code/

WORKDIR /home/gradle/java-code
RUN gradle clean build -i --stacktrace
```

```
FROM gradle:8.6.0-jdk21-graal AS GRADLE_BUILD

COPY --from=GRADLE_CACHE /home/gradle/cache_home /home/gradle/.gradle
COPY . /usr/src/code
WORKDIR /usr/src/code

USER root
RUN gradle bootJar -i --stacktrace
```

```
FROM ubuntu:22.04 AS VERIOPT

WORKDIR /home
RUN apt update && \
    apt install -y wget unzip && \
    wget --no-check-certificate -O veriopt.zip
        https://github.com/uqcyber/veriopt-releases/archive/main.zip && \
    unzip veriopt.zip && \
    rm veriopt.zip && \
    sed -i 's/document\s*=\s*.*\s*,/document = false,/g'
        /home/veriopt-releases-main/ROOT && \
    apt remove -y wget unzip && \
    apt autoremove -y && \
    rm -rf /var/lib/apt/lists/* /var/cache/apt/archives/*
```

```
FROM makarius/isabelle:Isabelle2023 AS ISABELLE_BUILD

COPY --chown=isabelle:isabelle --from=VERIOPT /home/veriopt-releases-main
    /home/isabelle/source

WORKDIR /home/isabelle/source
RUN /home/isabelle/Isabelle/bin/isabelle build -v -o system_heaps -b HOL-Library && \
    /home/isabelle/Isabelle/bin/isabelle build -R -v -d . -o system_heaps -o
        show_question_marks=false -o document=false -o quick_and_dirty -b
        Canonicalizations && \
    /home/isabelle/Isabelle/bin/isabelle build -v -d . -o system_heaps -o
        show_question_marks=false -o document=false -o quick_and_dirty -b
        Canonicalizations
```

```
FROM ghcr.io/graalvm/graalvm-community:21 AS VERITEST

COPY --from=GRADLE_BUILD /usr/src/code/build/libs/*.jar /root/app/app.jar
COPY --from=GRADLE_BUILD /usr/src/code/cmd /root/cmd

COPY --from=ISABELLE_BUILD /home/isabelle/Isabelle /root/Isabelle

COPY --from=VERIOPT /home/veriopt-releases-main /root/source

WORKDIR /root/cmd

CMD ["./start.sh"]
```

Appendix B

Perl Script to Generate Test Cases

```
#!/usr/bin/perl
use strict;
use warnings;
use JSON;
use Digest::MD5 qw(md5_hex);

# Check for the correct number of arguments
unless (@ARGV == 2) {
    die "Usage: $0 <input_file> <output_directory>\n";
}

my $filename = $ARGV[0];
open(my $fh, '<', $filename) or die "Could not open file '$filename' $!";

my $json = JSON->new->allow_nonref;

# Read output directory from command line argument and ensure it ends with a slash
my $output_directory = $ARGV[1];
$output_directory .= '/' unless $output_directory =~ /\/$/;

# Hash to keep track of duplicate request_id counts and written theories
my %request_id_count;
my %written_theories;

my $current_request_id = '';
my $collecting = 0;
my $data = '';

my $optimizationRegex = qr/.*:\d+:optimization\s+([^:]+): "(.*)$/;
my $newOptimizationRegex = qr/^.*:\d+:optimization/;
```

```
while (my $line = <$fh>) {
    chomp $line;
    if ($line =~ $optimizationRegex) {
        if ($collecting) {
            # Save the previous record to its own file if it's unique
            save_to_file($current_request_id, $data);
            $data = '';
        }
        # Start a new record
        $current_request_id = $1;
        $data = $2;
        $collecting = 1; # Start collecting lines
    } elsif ($collecting) {
        # Continue collecting lines if they do not start a new record
        if ($line =~ $newOptimizationRegex) {
            # If a new record starts, save the previous if unique and reset
            save_to_file($current_request_id, $data);
            $data = '';
            $collecting = 0;
            redo; # Re-evaluate this line because it's a new record start
        } else {
            $data .= "\n" . $line;
        }
    }
}

# Save the last record if it's unique
if ($collecting) {
    save_to_file($current_request_id, $data);
}

close $fh;
```

```
sub save_to_file {
    my ($request_id, $theory) = @_;

    # Remove everything from the last escaped quote to the end of the string
    $theory =~ s/\".*$//s; # The 's' modifier allows '.' to match newline characters
    $theory =~ s/^\s+|\s+$//g;

    my $theory_hash = md5_hex($theory);
    # Check if this theory has already been written
    return if $written_theories{$theory_hash};

    # Mark this theory as written
    $written_theories{$theory_hash} = 1;

    # Increment the count for the current request_id or initialize it
    $request_id_count{lc($request_id)}++;
    # Prepend the count to the request_id for the filename
    my $unique_request_id = $request_id . "_" . $request_id_count{lc($request_id)};
    my $output_file = $output_directory . $unique_request_id . ".json";
    $output_file =~ s/\s+/_/g; # Replace spaces with underscores for filenames

    # Ensure the output directory exists or create it
    unless (-d $output_directory) {
        mkdir $output_directory or die "Unable to create directory '$output_directory':
            $!";
    }

    open(my $out_fh, '>', $output_file) or die "Could not open file '$output_file' $!";
    print $out_fh $json->encode({ request_id => $unique_request_id, theory => $theory });
    close $out_fh;
}
```

Appendix C

Mutated Optimization Rules

```
// mapsto to longmapsto
optimization AbsIdempotence: "abs(abs(Rep_int32 e)) ⟼ abs(Rep_int32 e)"

// mapsto to longmapsto
optimization AbsNegate: "abs(-e) ⟼ abs(e) when is_IntegerStamp (stamp_expr e)"

// 0 to 1
optimization AddNeutral: "(e + (const (IntVal 32 1))) ⟼ e"

// 0 to 1
optimization AddNeutral: "(e + (const (IntVal 32 1))) ⟼ e when (stamp_expr e =
    IntegerStamp 32 l u)"

// b to 128
optimization AndNeutral: "(x & ~(const (IntVal 128 0))) ⟼ x
   when (wf_stamp x ∧ stamp_expr x = IntegerStamp 128 lo hi)"


// add ~
optimization AndRightFallThrough: "(x & y) ⟼ y
                             when ~(((and (not (IRExpr_down x)) (IRExpr_up y)) = 0))"

// << to >>
optimization AndSignExtend: "BinaryExpr BinAnd (UnaryExpr (UnarySignExtend In Out) (x))
                                        (const (new_int b e))
                            ⟼ (UnaryExpr (UnaryZeroExtend In Out) (x))
                               when (e = (1 >> In) - 1)"

// remove precondition
optimization BinaryFoldConstant: "BinaryExpr op (const e1) (const e2) ⟼
                                ConstantExpr (bin_eval op e1 e2)"

// < to >
optimization condition_bounds_x: "((u > v) ? x : y) ⟼ x
    when (stamp_under (stamp_expr u) (stamp_expr v) ∧ wf_stamp u ∧ wf_stamp v)"

// < to >=
optimization condition_bounds_x: "((u ≥ v) ? x : y) ⟼ x
    when (stamp_under (stamp_expr u) (stamp_expr v) ∧ wf_stamp u ∧ wf_stamp v)"

// < to >
optimization ConditionalEliminateKnownLess: "((x > y) ? x : y) ⟼ x
```

```
                                when (stamp_under (stamp_expr x) (stamp_expr y)
                                    ∧ wf_stamp x ∧ wf_stamp y)"

// < to >=
optimization ConditionalEliminateKnownLess: "((x ≥ y) ? x : y) ⟼ x
                                when (stamp_under (stamp_expr x) (stamp_expr y)
                                    ∧ wf_stamp x ∧ wf_stamp y)"

// missing colon
optimization ConditionalEqualBranches: "(e ? x x) ⟼ x" .

// y to x
optimization ConditionalEqualIsRHS: "((x eq y) ? x : y) ⟼ x"

// y to x
optimization ConditionalEqualIsRHS: "((x eq x) ? x : y) ⟼ y"

// add !
optimization ConditionalEqualIsRHS: "(!(x eq y) ? x : y) ⟼ y"

// add ~
optimization ConditionalEqualIsRHS: "(~(x eq y) ? x : y) ⟼ y"

// missing isBoolean
optimization ConditionalExtractCondition2: "exp[(c ? false : true)] ⟼ !c"

// missing exp
optimization ConditionalExtractCondition2: "(c ? false : true) ⟼ !c
                                    when isBoolean c"

// ! to ~
optimization ConditionalExtractCondition2: "exp[(c ? false : true)] ⟼ ~c
                                    when isBoolean c"

// swap true and false
optimization ConditionalExtractCondition2: "exp[(c ? true : false)] ⟼ ~c
                                    when isBoolean c"

// true to false
optimization DefaultTrueBranch: "(false ? x : y) ⟼ x" .

// false to true
optimization EliminateRedundantFalse: "x | true ⟼ x"

// BinOr to ShortCircuitOr
optimization EliminateRedundantFalse: "x || false ⟼ x"

// false to true
optimization EliminateRedundantFalse: "(x ⊕ true) ⟼ x"

// 0 to 1
optimization flipX: "((x eq (const (IntVal 32 1))) ?
                        (const (IntVal 32 1)) : (const (IntVal 32 0))) ⟼ x ⊕ (const
                            (IntVal 32 1))
                         when (x = ConstantExpr (IntVal 32 0) |
                              (x = ConstantExpr (IntVal 32 1)))"

// ⊕ to ^ (java xor)
optimization flipX: "((x eq (const (IntVal 32 1))) ?
```

```
                        (const (IntVal 32 1)) : (const (IntVal 32 0))) ⟼ x ^ (const
                            (IntVal 32 1))
                         when (x = ConstantExpr (IntVal 32 0) |
                              (x = ConstantExpr (IntVal 32 1)))"

// ConstantExpr to const
optimization flipX: "((x eq (const (IntVal 32 1))) ?
                        (const (IntVal 32 1)) : (const (IntVal 32 0))) ⟼ x ⊕ (const
                            (IntVal 32 1))
                         when (x = const (IntVal 32 0) |
                              (x = const (IntVal 32 1)))"

// mapsto to longmapsto
optimization MulNegate: "(x * const (-1) ) ⟼ -x when (stamp_expr x = IntegerStamp 32
    l u)"

// ConstantExpr to ConstExpr
optimization MulNeutral: "x * ConstExpr (IntVal b 1) ⟼ x"

// ConstantExpr to ConstNode
optimization MulNeutral: "x * ConstNode (IntVal b 1) ⟼ x"

// ConstantExpr to ConstantNode
optimization MulNeutral: "x * ConstantNode (IntVal b 1) ⟼ x"

// IntVal b 1 to 1
optimization MulNeutral: "x * const (1) ⟼ x"

// 0 to 1
optimization normalizeX: "((x eq const (IntVal 32 1)) ?
                            (const (IntVal 32 0)) : (const (IntVal 32 1))) ⟼ x
                            when stamp_expr x = IntegerStamp 32 0 1 ∧ wf_stamp x ∧
                                 isBoolean x"

// ~ to !
optimization NotCancel: "exp[!(!a)] ⟼ a"

// missing exp[]
optimization NotCancel: "!(!a) ⟼ a"

// ~ to !
optimization NotCancel: "exp[~(!a)] ⟼ a"

// ~ to !
optimization NotCancel: "exp[!(~a)] ⟼ a"

// - to +
optimization RedundantAddSub: "(b + a) + b ⟼ a"

// < to >=
optimization SubNegativeConstant: "(x - (const (IntVal b y))) ⟼
                                x + (const (IntVal b y)) when (y ≥ 0)"

// < to >
optimization SubNegativeConstant: "(x - (const (IntVal b y))) ⟼
                                x + (const (IntVal b y)) when (y > 0)"

// remove precondition
optimization SubSelfIsZero: "(x - x) ⟼ const IntVal b 0"
```

```
// missing brackets
optimization SubThenAddLeft: "(x - x + y) ⟼ -y"

// remove precondition
optimization XorSelfIsFalse: "(x ⊕ x) ⟼ false"

// false to true
optimization XorSelfIsFalse: "(x ⊕ x) ⟼ true when
                    (wf_stamp x ∧ stamp_expr x = default_stamp)"
```

Appendix D

Evaluation on Veriopt's Optimization Rules

Optimization Rule	Result	Mean	N	Std. Dev	Min	Max
AbsIdempotence_1	MALFORMED	36.8608	5	4.04082	30.17	40.13
	Total	36.8608	5	4.04082	30.17	40.13
AbsNegate_1	MALFORMED	40.2000	5	5.01788	33.01	45.01
	Total	40.2000	5	5.01788	33.01	45.01
AddLeftNegateToSub_1	FOUND_PROOF	113.9996	5	6.96647	108.00	123.01
	Total	113.9996	5	6.96647	108.00	123.01
AddNeutral_1	FOUND_PROOF	71.8072	5	8.14791	66.08	85.97
	Total	71.8072	5	8.14791	66.08	85.97
AddNeutral_2	FOUND_PROOF	68.3940	5	1.25673	66.15	69.00
	Total	68.3940	5	1.25673	66.15	69.00
AddNot_1	FAILED	69.0100	5	2.96315	65.90	72.00
	Total	69.0100	5	2.96315	65.90	72.00
AddNot2_1	FAILED	123.3238	5	0.77323	122.49	124.49
	Total	123.3238	5	0.77323	122.49	124.49
AddRightNegateToSub_1	FAILED	69.1270	5	2.73379	66.06	71.89
	Total	69.1270	5	2.73379	66.06	71.89
AddShiftConstantRight_1	FAILED	69.0274	5	2.07543	66.05	71.92
	Total	69.0274	5	2.07543	66.05	71.92
AddShiftConstantRight_2	FAILED	70.7876	5	1.61937	68.99	72.01

	Total	70.7876	5	1.61937	68.99	72.01
AndEqual_1	FOUND_PROOF	66.4082	5	4.38136	63.09	74.07
	Total	66.4082	5	4.38136	63.09	74.07
AndEqual_2	MALFORMED	37.1062	5	4.73613	29.53	42.00
	Total	37.1062	5	4.73613	29.53	42.00
AndLeftFallthrough_1	MALFORMED	38.9994	5	4.74326	33.00	45.00
	Total	38.9994	5	4.74326	33.00	45.00
AndNeutral_1	FOUND_PROOF	71.2144	5	3.78492	67.64	76.83
	Total	71.2144	5	3.78492	67.64	76.83
AndNeutral_2	MALFORMED	37.3844	5	7.61193	30.00	47.99
	Total	37.3844	5	7.61193	30.00	47.99
AndNots_1	FAILED	72.3512	5	6.26297	66.26	82.75
	Total	72.3512	5	6.26297	66.26	82.75
AndRight-Fallthrough_1	MALFORMED	36.8480	5	4.80699	28.25	39.01
	Total	36.8480	5	4.80699	28.25	39.01
AndSelf_1	FAILED	145.2112	5	7.94462	136.58	154.67
	Total	145.2112	5	7.94462	136.58	154.67
AndSelf2_1	FAILED	67.1934	5	6.64696	60.25	77.57
	Total	67.1934	5	6.64696	60.25	77.57
AndShiftConstantRight_1	FAILED	70.4710	5	2.05361	68.98	73.39
	Total	70.4710	5	2.05361	68.98	73.39
AndShiftConstantRight_2	FAILED	69.6100	5	2.71004	65.98	73.46
	Total	69.6100	5	2.71004	65.98	73.46
AndSignExtend_1	FOUND_PROOF	69.1106	5	1.98736	66.51	72.11
	Total	69.1106	5	1.98736	66.51	72.11
BinaryFoldConstant_1	FOUND_PROOF	96.6006	5	1.37825	95.88	99.06
	Total	96.6006	5	1.37825	95.88	99.06
BinaryFoldConstant_2	FOUND_PROOF	94.8050	5	2.64327	90.08	96.08
	Total	94.8050	5	2.64327	90.08	96.08
condition_bounds_x_1	FOUND_PROOF	69.0816	5	0.24518	68.69	69.31
	Total	69.0816	5	0.24518	68.69	69.31

condition_bounds_y_1	FOUND_PROOF	66.5922	5	1.15881	65.92	68.65
	Total	66.5922	5	1.15881	65.92	68.65
ConditionalEliminateKnownLess_1	FOUND_PROOF	71.3850	5	2.89079	66.97	73.94
	Total	71.3850	5	2.89079	66.97	73.94
ConditionalEliminateKnownLess_2	MALFORMED	38.6058	5	4.14825	34.03	44.99
	Total	38.6058	5	4.14825	34.03	44.99
ConditionalEliminateKnownLess_3	MALFORMED	38.3994	5	2.50771	36.00	42.00
	Total	38.3994	5	2.50771	36.00	42.00
ConditionalEqualBranches_1	FOUND_AUTO	39.0014	5	0.00744	38.99	39.01
	Total	39.0014	5	0.00744	38.99	39.01
ConditionalEqualBranches_2	FOUND_AUTO	38.9994	5	0.01128	38.99	39.01
	Total	38.9994	5	0.01128	38.99	39.01
ConditionalEqualIsRHS_1	FAILED	72.5772	5	4.77902	69.12	80.75
	Total	72.5772	5	4.77902	69.12	80.75
ConditionalEqualIsRHS_2	FAILED	69.6014	5	2.44543	66.02	71.93
	Total	69.6014	5	2.44543	66.02	71.93
ConditionalExtractCondition_1	FAILED	68.9022	5	2.10412	66.03	71.97
	Total	68.9022	5	2.10412	66.03	71.97
ConditionalExtractCondition2_1	FOUND_PROOF	66.6672	5	3.01269	64.18	70.25
	Total	66.6672	5	3.01269	64.18	70.25
ConditionalIntegerEquals_1_1	FAILED	74.4882	5	1.38439	72.03	75.38
	Total	74.4882	5	1.38439	72.03	75.38
ConditionalIntegerEquals_2_1	FAILED	73.2258	5	1.63229	71.88	75.01
	Total	73.2258	5	1.63229	71.88	75.01
DefaultFalseBranch_1	FOUND_AUTO	37.0574	5	5.69286	29.29	45.01
	Total	37.0574	5	5.69286	29.29	45.01

DefaultTrueBranch_1	FOUND_AUTO	39.6014	5	3.28671	35.99	42.01
	Total	39.6014	5	3.28671	35.99	42.01
distribute_sub_1	FAILED	70.1616	5	2.60717	65.96	71.95
	Total	70.1616	5	2.60717	65.96	71.95
DistributeSubtraction_1	FAILED	71.1220	5	5.06686	66.21	79.47
	Total	71.1220	5	5.06686	66.21	79.47
DivItself_1	FAILED	130.5130	5	6.14724	121.73	138.86
	Total	130.5130	5	6.14724	121.73	138.86
EliminateRedundantFalse_1	FOUND_PROOF	67.5126	5	2.09928	65.54	70.11
	Total	67.5126	5	2.09928	65.54	70.11
EliminateRedundantFalse_2	FOUND_PROOF	68.4922	5	4.04603	64.86	75.23
	Total	68.4922	5	4.04603	64.86	75.23
EliminateRedundantNegative_1	FAILED	71.2886	5	2.21591	69.10	74.36
	Total	71.2886	5	2.21591	69.10	74.36
EliminateRHS_64_1	FAILED	71.2576	5	1.24545	69.05	71.95
	Total	71.2576	5	1.24545	69.05	71.95
flipX_1	FOUND_AUTO	37.4988	4	3.00051	33.00	39.01
	FOUND_PROOF	44.9920	1		44.99	44.99
	Total	38.9974	5	4.24052	33.00	44.99
flipX2_1	FOUND_AUTO	37.0480	5	5.29949	29.24	42.01
	Total	37.0480	5	5.29949	29.24	42.01
MaskOutRHS_1	NO_SUBGOAL	38.9996	5	0.00627	38.99	39.01
	Total	38.9996	5	0.00627	38.99	39.01
MulEliminator_1	FOUND_PROOF	74.7916	5	3.48010	70.06	78.00
	Total	74.7916	5	3.48010	70.06	78.00
MulEliminator_2	MALFORMED	37.4070	5	4.14440	31.05	42.01
	Total	37.4070	5	4.14440	31.05	42.01
MulNegate_1	FOUND_PROOF	73.0076	5	2.53701	70.16	76.86
	Total	73.0076	5	2.53701	70.16	76.86
MulNegate_2	MALFORMED	37.9932	5	3.09781	33.96	42.00
	Total	37.9932	5	3.09781	33.96	42.00

MulNeutral_1	FOUND_PROOF	69.6430	5	2.93069	65.95	74.08
	Total	69.6430	5	2.93069	65.95	74.08
MulNeutral_2	FOUND_PROOF	66.9500	5	2.78694	62.61	70.19
	Total	66.9500	5	2.78694	62.61	70.19
MulPower2_1	FAILED	67.9712	5	1.46939	66.00	69.09
	Total	67.9712	5	1.46939	66.00	69.09
MulPower2Add1_1	FAILED	72.6552	5	1.46457	71.98	75.28
	Total	72.6552	5	1.46457	71.98	75.28
MulPower2Sub1_1	FAILED	68.9422	5	3.57582	62.88	71.69
	Total	68.9422	5	3.57582	62.88	71.69
NegateCancel_1	FAILED	70.3684	5	2.80912	65.92	72.43
	Total	70.3684	5	2.80912	65.92	72.43
NegateConditionFlip-Branches_1	FAILED	71.4162	5	1.35428	69.00	72.08
	Total	71.4162	5	1.35428	69.00	72.08
NegativeShift_1	FAILED	70.3488	5	2.34894	66.86	72.10
	Total	70.3488	5	2.34894	66.86	72.10
NeverEqNotSelf_1	FAILED	137.9394	5	1.09936	136.95	139.63
	Total	137.9394	5	1.09936	136.95	139.63
normalizeX_1	FOUND_PROOF	70.1996	5	4.77967	67.07	78.55
	Total	70.1996	5	4.77967	67.07	78.55
normalizeX2_1	FOUND_AUTO	34.1970	4	5.56249	28.80	39.00
	FOUND_PROOF	48.0020	1		48.00	48.00
	Total	36.9580	5	7.83081	28.80	48.00
NotCancel_1	FOUND_PROOF	67.6016	5	2.70756	62.82	69.18
	Total	67.6016	5	2.70756	62.82	69.18
NotXorToXor_1	FAILED	188.0008	5	21.88899	160.75	214.08
	Total	188.0008	5	21.88899	160.75	214.08
opt_conditional_eliminate_known_less_1	FAILED	70.3276	5	2.75181	66.11	72.73
	Total	70.3276	5	2.75181	66.11	72.73
opt_DivisionByOneIsSelf32_1	FAILED	69.5642	5	3.11374	66.09	71.97
	Total	69.5642	5	3.11374	66.09	71.97

opt_normalize_x_original_1	FAILED	68.8546	5	2.70214	66.07	71.83
	Total	68.8546	5	2.70214	66.07	71.83
OrEqual_1	FOUND_PROOF	66.7850	5	2.65995	63.18	70.38
	Total	66.7850	5	2.65995	63.18	70.38
OrInverse_1	FAILED	64.7408	5	2.65660	62.88	68.84
	Total	64.7408	5	2.65660	62.88	68.84
OrInverse2_1	FAILED	136.5728	5	0.90812	135.38	137.37
	Total	136.5728	5	0.90812	135.38	137.37
OrNotOperands_1	FAILED	69.5350	5	2.33398	66.18	72.00
	Total	69.5350	5	2.33398	66.18	72.00
OrShiftConstantRight_1	FAILED	68.3546	5	3.74285	63.09	71.99
	Total	68.3546	5	3.74285	63.09	71.99
redundant_lhs_add_1	FAILED	86.5026	5	5.34624	82.61	95.66
	Total	86.5026	5	5.34624	82.61	95.66
redundant_lhs_x_or_1	FAILED	81.9728	5	6.33849	72.67	88.96
	Total	81.9728	5	6.33849	72.67	88.96
redundant_lhs_y_or_1	FAILED	78.7754	5	3.13297	73.93	81.22
	Total	78.7754	5	3.13297	73.93	81.22
redundant_rhs_x_or_1	FAILED	80.2422	5	4.84560	76.24	87.58
	Total	80.2422	5	4.84560	76.24	87.58
redundant_rhs_y_or_1	FAILED	80.7394	5	2.30875	77.40	82.81
	Total	80.7394	5	2.30875	77.40	82.81
RedundantAddSub_1	FOUND_PROOF	70.4562	5	2.89715	66.19	73.95
	Total	70.4562	5	2.89715	66.19	73.95
RedundantAddSub_2	MALFORMED	38.9994	5	0.00114	39.00	39.00
	Total	38.9994	5	0.00114	39.00	39.00
RedundantSubAdd_1	FOUND_PROOF	71.2046	5	8.43466	65.93	86.04
	Total	71.2046	5	8.43466	65.93	86.04
RedundantSubAdd_2	MALFORMED	37.3924	5	3.59471	30.96	39.01
	Total	37.3924	5	3.59471	30.96	39.01

RedundantSubAdd2_1	FOUND_PROOF	140.4016	5	7.76315	132.00	147.00
	Total	140.4016	5	7.76315	132.00	147.00
ReturnXOnZeroShift_1	FAILED	71.3120	5	1.53021	68.58	72.05
	Total	71.3120	5	1.53021	68.58	72.05
SubAfterAddLeft_1	FOUND_PROOF	72.2480	5	4.09551	65.26	75.00
	Total	72.2480	5	4.09551	65.26	75.00
SubAfterAddRight_1	FOUND_PROOF	69.6252	5	2.77983	67.02	74.07
	Total	69.6252	5	2.77983	67.02	74.07
SubAfterSubLeft_1	FAILED	70.1160	5	2.58641	65.99	72.13
	Total	70.1160	5	2.58641	65.99	72.13
SubNegativeConstant_1	NO_SUBGOAL	36.8606	5	4.78497	28.30	39.01
	Total	36.8606	5	4.78497	28.30	39.01
SubNegativeConstant_2	COUNTEREXAMPLE	40.7992	5	4.02314	38.99	48.00
	Total	40.7992	5	4.02314	38.99	48.00
SubNegativeValue_1	FAILED	72.2814	5	0.62427	71.99	73.40
	Total	72.2814	5	0.62427	71.99	73.40
SubSelfIsZero_1	FAILED	68.0692	5	2.41427	66.03	71.90
	Total	68.0692	5	2.41427	66.03	71.90
SubSelfIsZero_2	FOUND_PROOF	177.6476	5	20.62850	147.24	200.99
	Total	177.6476	5	20.62850	147.24	200.99
SubThenAddLeft_1	FOUND_PROOF	81.0014	5	22.86855	68.91	121.84
	Total	81.0014	5	22.86855	68.91	121.84
SubThenAddRight_1	FOUND_PROOF	70.7994	5	2.50392	67.02	73.80
	Total	70.7994	5	2.50392	67.02	73.80
SubThenSubLeft_1	FOUND_PROOF	152.9996	5	18.61544	123.00	174.00
	Total	152.9996	5	18.61544	123.00	174.00
SubtractZero_1	FOUND_PROOF	74.2148	5	15.67855	65.94	101.99
	Total	74.2148	5	15.67855	65.94	101.99
UnaryConstantFold_1	MALFORMED	37.3824	5	3.61204	30.92	39.01
	Total	37.3824	5	3.61204	30.92	39.01
XorEqNeg1_64_1	FAILED	69.8170	5	2.80738	65.91	73.20
	Total	69.8170	5	2.80738	65.91	73.20

XorEqZero_64_1	FAILED	133.4868	5	2.80689	130.43	137.72
	Total	133.4868	5	2.80689	130.43	137.72
XorFallThrough1_1	FAILED	69.8928	5	2.43540	66.10	71.92
	Total	69.8928	5	2.43540	66.10	71.92
XorFallThrough2_1	FAILED	68.2800	5	3.24721	65.34	71.85
	Total	68.2800	5	3.24721	65.34	71.85
XorFallThrough3_1	FAILED	69.7236	5	2.64008	66.11	72.68
	Total	69.7236	5	2.64008	66.11	72.68
XorFallThrough4_1	FAILED	70.6492	5	1.38303	69.15	72.01
	Total	70.6492	5	1.38303	69.15	72.01
XorInverse_1	FAILED	68.1978	5	1.19642	66.13	69.01
	Total	68.1978	5	1.19642	66.13	69.01
XorInverse2_1	FAILED	145.0686	5	4.21575	137.85	148.58
	Total	145.0686	5	4.21575	137.85	148.58
XorIsEqual_64_1_1	FAILED	228.3936	5	34.23598	169.80	250.85
	Total	228.3936	5	34.23598	169.80	250.85
XorIsEqual_64_2_1	FAILED	72.0072	5	0.35231	71.44	72.30
	Total	72.0072	5	0.35231	71.44	72.30
XorIsEqual_64_3_1	FAILED	71.4846	5	1.34938	69.09	72.21
	Total	71.4846	5	1.34938	69.09	72.21
XorIsEqual_64_4_1	FAILED	270.2736	5	245.23649	146.10	708.20
	Total	270.2736	5	245.23649	146.10	708.20
XorSelfIsFalse_1	FOUND_PROOF	136.1898	5	14.33338	125.94	156.00
	Total	136.1898	5	14.33338	125.94	156.00
XorShiftConstantRight_1	FAILED	72.8956	5	5.88537	66.16	82.41
	Total	72.8956	5	5.88537	66.16	82.41
ZeroSubtractValue_1	FAILED	70.1394	5	2.76388	65.77	72.03
	Total	70.1394	5	2.76388	65.77	72.03
Total	FAILED	87.3304	290	49.80078	60.25	708.20
	FOUND_AUTO	37.7370	33	3.92665	28.80	45.01
	COUNTEREXAMPLE	40.7992	5	4.02314	38.99	48.00
	FOUND_PROOF	82.9147	162	29.43735	44.99	200.99
	MALFORMED	37.9676	65	4.02370	28.25	47.99

NO_SUBGOAL	37.9301	10	3.38333	28.30	39.01
Total	76.2027	565	43.47376	28.25	708.20

Table D.1: Full results for the evaluation of each existing Veriopt optimization rules based on runtime (in seconds)

Appendix E

Evaluation on Malformed Optimization Rules

Optimization Rule	Result	Mean	N	Std. Dev	Min	Max
AbsIdempotence_1	TYPE_ERROR	26.5800	1		26.58	26.58
	Total	26.5800	1		26.58	26.58
AbsNegate_1	TYPE_ERROR	37.3060	1		37.31	37.31
	Total	37.3060	1		37.31	37.31
AddNeutral_1	FAILED	82.0780	1		82.08	82.08
	Total	82.0780	1		82.08	82.08
AddNeutral_2	FAILED	72.4990	1		72.50	72.50
	Total	72.4990	1		72.50	72.50
AndNeutral_1	FOUND_PROOF	67.3350	1		67.34	67.34
	Total	67.3350	1		67.34	67.34
AndRight-FallThrough_1	FAILED	75.9740	1		75.97	75.97
	Total	75.9740	1		75.97	75.97
AndSignExtend_1	FAILED	73.8460	1		73.85	73.85
	Total	73.8460	1		73.85	73.85
BinaryFoldConstant_1	FOUND_PROOF	96.7980	1		96.80	96.80
	Total	96.7980	1		96.80	96.80
condi-tion_bounds_x_1	TYPE_ERROR	35.7200	1		35.72	35.72
	Total	35.7200	1		35.72	35.72
condi-tion_bounds_x_2	TYPE_ERROR	32.3080	1		32.31	32.31
	Total	32.3080	1		32.31	32.31

ConditionalEliminate-KnownLess_1	TYPE_ERROR	32.4500	1	32.45	32.45
	Total	32.4500	1	32.45	32.45
ConditionalEliminate-KnownLess_2	TYPE_ERROR	41.9960	1	42.00	42.00
	Total	41.9960	1	42.00	42.00
ConditionalEqual-Branches_1	MALFORMED	35.9850	1	35.99	35.99
	Total	35.9850	1	35.99	35.99
ConditionalEqualIs-RHS_1	FAILED	75.1520	1	75.15	75.15
	Total	75.1520	1	75.15	75.15
ConditionalEqualIs-RHS_2	FAILED	72.7400	1	72.74	72.74
	Total	72.7400	1	72.74	72.74
ConditionalEqualIs-RHS_3	TYPE_ERROR	29.5520	1	29.55	29.55
	Total	29.5520	1	29.55	29.55
ConditionalEqualIs-RHS_4	TYPE_ERROR	38.5330	1	38.53	38.53
	Total	38.5330	1	38.53	38.53
ConditionalExtract-Condition2_1	FAILED	81.5480	1	81.55	81.55
	Total	81.5480	1	81.55	81.55
ConditionalExtract-Condition2_2	FOUND_PROOF	70.3920	1	70.39	70.39
	Total	70.3920	1	70.39	70.39
ConditionalExtract-Condition2_3	FAILED	65.1260	1	65.13	65.13
	Total	65.1260	1	65.13	65.13
ConditionalExtract-Condition2_4	FAILED	68.8980	1	68.90	68.90
	Total	68.8980	1	68.90	68.90
DefaultTrueBranch_1	FAILED	78.9570	1	78.96	78.96
	Total	78.9570	1	78.96	78.96
EliminateRedundant-False_1	FAILED	72.1350	1	72.14	72.14
	Total	72.1350	1	72.14	72.14

EliminateRedundant-False_2	TYPE_ERROR	29.2610	1	29.26	29.26
	Total	29.2610	1	29.26	29.26
EliminateRedundant-False_3	FAILED	81.3280	1	81.33	81.33
	Total	81.3280	1	81.33	81.33
flipX_1	FAILED	72.9960	1	73.00	73.00
	Total	72.9960	1	73.00	73.00
flipX_2	TYPE_ERROR	29.4190	1	29.42	29.42
	Total	29.4190	1	29.42	29.42
flipX_3	TYPE_ERROR	37.8720	1	37.87	37.87
	Total	37.8720	1	37.87	37.87
MulNegate_1	TYPE_ERROR	39.0040	1	39.00	39.00
	Total	39.0040	1	39.00	39.00
MulNeutral_1	FAILED	82.0560	1	82.06	82.06
	Total	82.0560	1	82.06	82.06
MulNeutral_2	FAILED	72.9560	1	72.96	72.96
	Total	72.9560	1	72.96	72.96
MulNeutral_3	FAILED	68.4410	1	68.44	68.44
	Total	68.4410	1	68.44	68.44
MulNeutral_4	TYPE_ERROR	29.4670	1	29.47	29.47
	Total	29.4670	1	29.47	29.47
normalizeX_1	FAILED	80.0690	1	80.07	80.07
	Total	80.0690	1	80.07	80.07
NotCancel_1	FAILED	73.2060	1	73.21	73.21
	Total	73.2060	1	73.21	73.21
NotCancel_2	TYPE_ERROR	29.1060	1	29.11	29.11
	Total	29.1060	1	29.11	29.11
NotCancel_3	FAILED	78.6620	1	78.66	78.66
	Total	78.6620	1	78.66	78.66
NotCancel_4	FAILED	72.2580	1	72.26	72.26
	Total	72.2580	1	72.26	72.26
RedundantAddSub_1	FAILED	66.6610	1	66.66	66.66
	Total	66.6610	1	66.66	66.66

SubNegativeConstant_1	COUNTEREXAMPLE	38.4230	1		38.42	38.42
	Total	38.4230	1		38.42	38.42
SubNegativeConstant_2	COUNTEREXAMPLE	74.0180	1		74.02	74.02
	Total	74.0180	1		74.02	74.02
SubSelfIsZero_1	FAILED	134.2790	1		134.28	134.28
	Total	134.2790	1		134.28	134.28
SubThenAddLeft_1	FAILED	72.8450	1		72.85	72.85
	Total	72.8450	1		72.85	72.85
XorSelfIsFalse_1	FAILED	134.1240	1		134.12	134.12
	Total	134.1240	1		134.12	134.12
XorSelfIsFalse_2	FAILED	114.9130	1		114.91	114.91
	Total	114.9130	1		114.91	114.91
Total	FAILED	80.9499	25	18.53822	65.13	134.28
	COUNTEREXAMPLE	56.2205	2	25.16947	38.42	74.02
	FOUND_PROOF	78.1750	3	16.20026	67.34	96.80
	MALFORMED	35.9850	1		35.99	35.99
	TYPE_ERROR	33.4696	14	4.82394	26.58	42.00
	Total	63.8949	45	26.73482	26.58	134.28

Table E.1: Full results for the evaluation of each mutated optimization rules based on runtime (in seconds)

Appendix F

Evaluation on GraalVM's Source Code Annotations

Optimization Rule	Result	Mean	N	Std. Dev	Min	Max
AbsIdempotence_1	TYPE_ERROR	34.6030	1		34.60	34.60
	Total	34.6030	1		34.60	34.60
AbsNegate_1	TYPE_ERROR	29.8490	1		29.85	29.85
	Total	29.8490	1		29.85	29.85
AddLeftNegateToSub_1	FOUND_PROOF	107.0670	1		107.07	107.07
	Total	107.0670	1		107.07	107.07
AddNeutral_1	TYPE_ERROR	33.9280	1		33.93	33.93
	Total	33.9280	1		33.93	33.93
AddRightNegateToSub_1	FAILED	82.4230	1		82.42	82.42
	Total	82.4230	1		82.42	82.42
AndEqual_1	FOUND_PROOF	61.5710	1		61.57	61.57
	Total	61.5710	1		61.57	61.57
AndNeutral_1	FAILED	75.7800	1		75.78	75.78
	Total	75.7800	1		75.78	75.78
ConditionalEqualBranches_1	FOUND_AUTO	29.2240	1		29.22	29.22
	Total	29.2240	1		29.22	29.22
ConditionalEqualIsRHS_1	TYPE_ERROR	39.0070	1		39.01	39.01
	Total	39.0070	1		39.01	39.01
ConvertTernaryIntoShift_1	MALFORMED	38.9860	1		38.99	38.99

	Total	38.9860	1	38.99	38.99
DefaultFalseBranch_1	TYPE_ERROR	39.0000	1	39.00	39.00
	Total	39.0000	1	39.00	39.00
DefaultTrueBranch_1	TYPE_ERROR	38.9940	1	38.99	38.99
	Total	38.9940	1	38.99	38.99
DistributeSubtraction_1	MALFORMED	38.9970	1	39.00	39.00
	Total	38.9970	1	39.00	39.00
DivisionByNegativeOneIsNegativeSelf_1	TYPE_ERROR	39.0070	1	39.01	39.01
	Total	39.0070	1	39.01	39.01
DivisionByOneIsSelf_1	TYPE_ERROR	38.9890	1	38.99	38.99
	Total	38.9890	1	38.99	38.99
DivisionByOneIsSelf_2	MALFORMED	38.9950	1	39.00	39.00
	Total	38.9950	1	39.00	39.00
DivisionByPower2_1	MALFORMED	47.9960	1	48.00	48.00
	Total	47.9960	1	48.00	48.00
EliminateConstantRHS_1	FAILED	72.5250	1	72.53	72.53
	Total	72.5250	1	72.53	72.53
EliminateOtherLeftShift_1	TYPE_ERROR	29.4590	1	29.46	29.46
	Total	29.4590	1	29.46	29.46
EliminateRedundantFalse_1	FOUND_PROOF	77.1580	1	77.16	77.16
	Total	77.1580	1	77.16	77.16
EliminateRedundantFalse_2	TYPE_ERROR	30.8360	1	30.84	30.84
	Total	30.8360	1	30.84	30.84
EliminateRedundantMod_1	TYPE_ERROR	47.9980	1	48.00	48.00
	Total	47.9980	1	48.00	48.00
EliminateRedundantNegative_1	FAILED	73.3410	1	73.34	73.34

	Total	73.3410	1	73.34	73.34
EliminateRHS_1	FAILED	68.2570	1	68.26	68.26
	Total	68.2570	1	68.26	68.26
EliminateRHS_2	FAILED	72.2740	1	72.27	72.27
	Total	72.2740	1	72.27	72.27
FalseCGeqZero_1	COUNTEREX-AMPLE	35.1160	1	35.12	35.12
	Total	35.1160	1	35.12	35.12
FalseCLessZero_1	COUNTEREX-AMPLE	42.2810	1	42.28	42.28
	Total	42.2810	1	42.28	42.28
flipX_1	MALFORMED	38.0480	1	38.05	38.05
	Total	38.0480	1	38.05	38.05
flipX2_1	MALFORMED	30.6570	1	30.66	30.66
	Total	30.6570	1	30.66	30.66
FloatDivByOneIs-Self_1	TYPE_ERROR	39.0070	1	39.01	39.01
	Total	39.0070	1	39.01	39.01
FloatDoubleLHSMax-NaNIsNaN_1	MALFORMED	38.9960	1	39.00	39.00
	Total	38.9960	1	39.00	39.00
FloatDoubleLHSMin-NaNIsNaN_1	MALFORMED	38.9990	1	39.00	39.00
	Total	38.9990	1	39.00	39.00
FloatDoubleRHSMax-NaNIsNaN_1	MALFORMED	38.9980	1	39.00	39.00
	Total	38.9980	1	39.00	39.00
FloatDoubleRHSMin-NaNIsNaN_1	MALFORMED	38.9970	1	39.00	39.00
	Total	38.9970	1	39.00	39.00
FloatLHSMax-OfNegInfinity_1	FOUND_PROOF	47.9970	1	48.00	48.00
	Total	47.9970	1	48.00	48.00
FloatLHSMinOfPosIn-finity_1	FOUND_PROOF	38.9980	1	39.00	39.00
	Total	38.9980	1	39.00	39.00

FloatRHSMax-OfNegInfinity_1	FOUND_PROOF	36.0060	1	36.01	36.01
	Total	36.0060	1	36.01	36.01
FloatRHSMinOfPosIn-finity_1	FOUND_PROOF	41.9980	1	42.00	42.00
	Total	41.9980	1	42.00	42.00
FloatTrun-cateTernary1_1	TYPE_ERROR	29.9910	1	29.99	29.99
	Total	29.9910	1	29.99	29.99
FloatTrun-cateTernary2_1	TYPE_ERROR	38.9950	1	39.00	39.00
	Total	38.9950	1	39.00	39.00
LeftShiftBe-comesZero_1	TYPE_ERROR	41.9880	1	41.99	41.99
	Total	41.9880	1	41.99	41.99
MaskOutRHS_1	TYPE_ERROR	42.0060	1	42.01	42.01
	Total	42.0060	1	42.01	42.01
MulEliminator_1	FOUND_PROOF	41.9880	1	41.99	41.99
	Total	41.9880	1	41.99	41.99
MulNegate_1	TYPE_ERROR	30.0080	1	30.01	30.01
	Total	30.0080	1	30.01	30.01
MulNegativeConst-Shift_1	TYPE_ERROR	38.9980	1	39.00	39.00
	Total	38.9980	1	39.00	39.00
MulNeutral2_1	TYPE_ERROR	38.9860	1	38.99	38.99
	Total	38.9860	1	38.99	38.99
MulPower2_1	TYPE_ERROR	38.9940	1	38.99	38.99
	Total	38.9940	1	38.99	38.99
MulPower2Add1_1	TYPE_ERROR	39.0110	1	39.01	39.01
	Total	39.0110	1	39.01	39.01
MulPower2Add1_2	TYPE_ERROR	38.9970	1	39.00	39.00
	Total	38.9970	1	39.00	39.00
MulPower2Ad-dPower2_1	TYPE_ERROR	38.9980	1	39.00	39.00
	Total	38.9980	1	39.00	39.00
MulPower2Sub1_1	TYPE_ERROR	38.9970	1	39.00	39.00

	Total	38.9970	1	39.00	39.00
NegateCancel_1	FAILED	81.2560	1	81.26	81.26
	Total	81.2560	1	81.26	81.26
NegateConditionFlip-Branches_1	FAILED	66.4960	1	66.50	66.50
	Total	66.4960	1	66.50	66.50
NegativeShift_1	MALFORMED	29.2310	1	29.23	29.23
	Total	29.2310	1	29.23	29.23
normalizeX_1	MALFORMED	38.9880	1	38.99	38.99
	Total	38.9880	1	38.99	38.99
normalizeX2_1	MALFORMED	39.0080	1	39.01	39.01
	Total	39.0080	1	39.01	39.01
NotCancel_1	FOUND_PROOF	74.1960	1	74.20	74.20
	Total	74.1960	1	74.20	74.20
OptimiseIntegerTest_1	TYPE_ERROR	30.8170	1	30.82	30.82
	Total	30.8170	1	30.82	30.82
OrEqual_1	FOUND_PROOF	77.2480	1	77.25	77.25
	Total	77.2480	1	77.25	77.25
OrLeftFallthrough_1	FOUND_PROOF	36.7470	1	36.75	36.75
	Total	36.7470	1	36.75	36.75
OrNotOperands_1	FAILED	73.5340	1	73.53	73.53
	Total	73.5340	1	73.53	73.53
OrRightFallthrough_1	FOUND_PROOF	34.4510	1	34.45	34.45
	Total	34.4510	1	34.45	34.45
OrShiftCon-stantRight_1	FAILED	76.4070	1	76.41	76.41
	Total	76.4070	1	76.41	76.41
RedundantAddSub_1	MALFORMED	28.5770	1	28.58	28.58
	Total	28.5770	1	28.58	28.58
RedundantSubAdd_1	MALFORMED	42.0080	1	42.01	42.01
	Total	42.0080	1	42.01	42.01
RemainderCompareZe-roEquivalent_1	TYPE_ERROR	35.9890	1	35.99	35.99
	Total	35.9890	1	35.99	35.99

RemainderWhenXNegative_1	MALFORMED	45.0070	1	45.01	45.01
	Total	45.0070	1	45.01	45.01
RemainderWhenXPositive_1	MALFORMED	32.9870	1	32.99	32.99
	Total	32.9870	1	32.99	32.99
ReturnXOnZeroShift_1	TYPE_ERROR	39.0070	1	39.01	39.01
	Total	39.0070	1	39.01	39.01
RightShiftsBecomeNegativeOne_1	TYPE_ERROR	38.9870	1	38.99	38.99
	Total	38.9870	1	38.99	38.99
RightShiftsBecomeZero_1	TYPE_ERROR	39.0090	1	39.01	39.01
	Total	39.0090	1	39.01	39.01
RightShiftsMerge1_1	TYPE_ERROR	41.9980	1	42.00	42.00
	Total	41.9980	1	42.00	42.00
RightShiftsMerge2_1	TYPE_ERROR	38.9850	1	38.99	38.99
	Total	38.9850	1	38.99	38.99
ShiftLeftByConstantAnd_1	TYPE_ERROR	35.9970	1	36.00	36.00
	Total	35.9970	1	36.00	36.00
ShiftRightByConstantAnd_1	TYPE_ERROR	39.0060	1	39.01	39.01
	Total	39.0060	1	39.01	39.01
SubAfterAddLeft_1	FOUND_PROOF	76.9990	1	77.00	77.00
	Total	76.9990	1	77.00	77.00
SubAfterAddRight_1	FOUND_PROOF	72.2130	1	72.21	72.21
	Total	72.2130	1	72.21	72.21
SubAfterSubLeft_1	FAILED	71.3620	1	71.36	71.36
	Total	71.3620	1	71.36	71.36
SubNegativeValue_1	FAILED	71.6750	1	71.68	71.68
	Total	71.6750	1	71.68	71.68
SubSelfIsZero_1	TYPE_ERROR	28.7390	1	28.74	28.74
	Total	28.7390	1	28.74	28.74
SubThenAddLeft_1	FOUND_PROOF	77.9880	1	77.99	77.99

	Total	77.9880	1	77.99	77.99
SubThenAddRight_1	FOUND_PROOF	77.2970	1	77.30	77.30
	Total	77.2970	1	77.30	77.30
SubThenSubLeft_1	FOUND_PROOF	120.6880	1	120.69	120.69
	Total	120.6880	1	120.69	120.69
SubtractZero_1	TYPE_ERROR	60.0070	1	60.01	60.01
	Total	60.0070	1	60.01	60.01
TransformRightShift-IntoConstantShift_1	TYPE_ERROR	38.9990	1	39.00	39.00
	Total	38.9990	1	39.00	39.00
TransformShiftInto-Mul_1	TYPE_ERROR	39.0060	1	39.01	39.01
	Total	39.0060	1	39.01	39.01
TransformToUnsigne-dOnPositiveX_1	COUNTEREX-AMPLE	44.9770	1	44.98	44.98
	Total	44.9770	1	44.98	44.98
TrueCGeqZero_1	FOUND_PROOF	40.8700	1	40.87	40.87
	Total	40.8700	1	40.87	40.87
TrueCLessZero_1	FOUND_PROOF	40.1240	1	40.12	40.12
	Total	40.1240	1	40.12	40.12
UnsignedMergeDivi-sion_1	MALFORMED	30.0080	1	30.01	30.01
	Total	30.0080	1	30.01	30.01
UnsignedRemWhenY-One_1	MALFORMED	38.9980	1	39.00	39.00
	Total	38.9980	1	39.00	39.00
UnsignedRemWhenY-Power2_1	MALFORMED	38.9970	1	39.00	39.00
	Total	38.9970	1	39.00	39.00
UnsignedRemXY-Const_1	MALFORMED	44.9860	1	44.99	44.99
	Total	44.9860	1	44.99	44.99
XorSelfIsFalse_1	TYPE_ERROR	35.9970	1	36.00	36.00
	Total	35.9970	1	36.00	36.00
XorShiftCon-stantRight_1	TYPE_ERROR	35.9980	1	36.00	36.00
	Total	35.9980	1	36.00	36.00

ZeroSubtractValue_1	TYPE_ERROR	39.0080	1		39.01	39.01
	Total	39.0080	1		39.01	39.01
Total	FAILED	73.7775	12	4.66542	66.50	82.42
	FOUND_AUTO	29.2240	1		29.22	29.22
	COUNTEREX-AMPLE	40.7913	3	5.09649	35.12	44.98
	FOUND_PROOF	62.1897	19	25.19682	34.45	120.69
	MALFORMED	38.0221	21	5.16433	28.58	48.00
	TYPE_ERROR	37.8549	40	5.43876	28.74	60.01
	Total	47.1999	96	18.31314	28.58	120.69

Table F.1: Full results for the evaluation of each existing GraalVM source code annotation based on runtime (in seconds)