Verifying Compiler Optimization Passes

by
Brae Webb

School of Information Technology and Electrical Engineering,
The University of Queensland

Submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy
in the field of Computer Science

January 2023

Contents

Abstract

Optimizing compilers are notoriously difficult to implement correctly. Rigorous fuzzy testing has found that the optimization phase is most frequently the source of bugs within a compiler implementation. An incorrect compiler implementation can manifest bugs in any of the myriad of programs which the implementation compiles, including safety-critical software. Mechanized formal verification, as the most trusted approach to ensuring software is bug-free, has shown promising results as a solution to incorrect compiler implementations. Unfortunately, mechanized formal verification is a time consuming task requiring experts in the field and, as a consequence, it is not a standard practice for compiler implementations.

We introduce and outline a plan to apply mechanized formal verification techniques to the optimization phase of a state-of-the-art production compiler. The GraalVM compiler uses Futamura projections to implement a variety of hosted languages; subsequently, all hosted languages share the same optimization phase. The shared optimization phase allows all hosted languages to benefit from verifying the optimization phase with no additional verification effort. We propose research into verifying the correctness of the shared optimization phase in a manner that is accessible to the existing compiler developers.

1 Introduction

As software becomes more elaborate, developers seek higher levels of abstraction to enable them to manage the required complexity of modern projects. As a consequence, concerns of developers have, in general, shifted away from writing efficient code and towards writing code which can be maintained.

The ability of developers to focus on literate programming at higher levels of abstraction is in no small part a consequence of optimizing compilers. Software developers are reliant on compilers to optimize their code. Consequently, compiler optimizations have become more specialized and complex to keep pace with the growing needs of developers. These advances exacerbate the problem of incorrect optimizations for which compilers are already prone.

The optimization phase of a compiler has been shown to be prone to introducing translation bugs [1]. This compiler phase is notoriously difficult to get right as translations need to ensure that every subtle semantic rule is preserved.

Figure 1 illustrates how subtle language semantics can invalidate seemingly correct optimizations. At first glance, the unoptimized version of `lessThan` appears to be a poor implementation of the optimized version. In fact, testing would indicate that this is the case, Figure 2 shows passing tests which achieve path coverage. However, due to the semantics of the equality operator for the Java `Integer` class, these methods are not equivalent. For all equal integers, `a`, `b`, larger than 128, the call `lessThan(a, b)` returns `true` while `lessThanOpt(a, b)` returns `false`. The Java equality operator for reference types performs reference equality. Therefore, two distinct integer instances equal in value evaluates to `false`. However, since Java caches all `Integer` instances less than 128, their references are equal if their values are equal [2], allowing the test suite to pass. This small example serves to illustrate the challenge of implementing correct optimizations even when accompanied by a comprehensive test suite.

Testing can show presence of bugs but not their absence [3], formal verification is the gold standard for ensuring programs meet their specification. While traditional software testing may suffice for most software projects, for something as foundational, relied upon, and challenging as an optimizing compiler, it is often worth the overhead involved in the formal verification process. It is worth reiterating that a bug in a compiler has the potential to manifest in any program which it compiles. At best, compiler bugs manifest as a debugging challenge for the programmer and, at worst, may manifest in the failure of a deployed safety-critical system.

```
boolean lessThan(Integer a, Integer b) {
    if (a == b) {
        return false;
    }
    if (a < b) {
        return true;
    }
    if (a > b) {
        return false;
    }
    return true;
}
```

(a) A convoluted method to calculate if an integer is less than another

```
boolean lessThanOpt(Integer a, Integer b) {
    return a < b;
}
```

(b) Potential optimization of the `lessThan` method

Figure 1. An example optimization of an overly verbose method to calculate if one integer is less than another to a simpler computation

```
assert lessThan(0, 0) == lessThanOpt(0, 0)
assert lessThan(0, 1) == lessThanOpt(0, 1)
assert lessThan(1, 0) == lessThanOpt(1, 0)
assert lessThan(-1, 100) == lessThanOpt(-1, 100)
assert lessThan(50, 50) == lessThanOpt(50, 50)
```

Figure 2. Unit tests ensuring the optimized and unoptimized code in Figure 1 are equivalent

This thesis is part of a larger collaborative project between the University of Queensland and Oracle Labs.[1] The project aims to formally verify the optimization phase of the GraalVM compiler. The project is led by Professor Ian Hayes and Associate Professor Mark Utting. This PhD thesis is funded in part by a gift from Oracle Labs.

1.1 GraalVM Compiler

The GraalVM compiler [4] is a modern compiler for the JVM, it has a focus on producing highly optimized code. The compiler features hot-spot and native compilation of JVM bytecode. It is of particular interest due to its approach to supporting multiple source languages. The execution of a hosted language is implemented via an interpreter written in annotated Java [5]. To overcome the inefficiency of interpretation, the interpreter code itself is subject to optimization at runtime using the scheme devised by Futamura [6]. In particular, it is subject to partial evaluation which allows for efficient execution comparable to that of existing tailored compilers [7]. The optimizer (which includes partial evaluation) is crucial to providing efficient execution of hosted languages. As hosted languages are implemented by a Java interpreter, the optimization and code generation phases are common to all the languages supported by GraalVM.

1.2 GraalVM Intermediate Representation

During the compilation process of the GraalVM compiler, the input program is modeled as an Intermediate Representation (IR) that is a graph structure. This graph is an implementation of

[1] https://www.eait.uq.edu.au/news/article/uq-and-oracle-team-develop-world-class-cyber-security-experts

```
if (cond) {
    result = value1 + value2;
} else {
    result = value2;
}
return result;
```

Figure 3. Control-flow dependent variable

the sea-of-nodes structure introduced by Cliff Click [8].

The sea-of-nodes structure is based on Static Single Assignment (SSA) form [9]. Programs are efficiently converted into SSA form during parsing [10]. In SSA form, each variable is assigned exactly once. For variables whose values are dependent on the control-flow path, as with `result` in Figure 3, ϕ variables are introduced. A ϕ variable can have multiple possible values that are resolved by the control-flow path taken.

Sea-of-nodes is designed for program optimizations and is particularly well suited for optimizations such as Conditional Constant Propagation [11], Global Value Numbering [9], and Global Code Motion [12]. The sea-of-nodes structure makes data-flow and control-flow dependencies explicit by representing these dependencies as edges of the graph.

The sea-of-nodes structure consists of two superimposed graph structures. The control-flow graph defines the execution flow of a program, and the data-flow graph manages value producing operations and data-flow dependencies. Each node of the graph can produce at most one value, as in traditional SSA. Control-flow edges specify the execution path. Data-flow edges specify data dependencies.

Demange *et al.* [13] have provided an initial formalization of the sea-of-nodes structure in a limited environment, excluding a model of the heap, interprocedural calls, and exception handling. They focus primarily on the formal semantics of ϕ nodes and regions.

As a first step towards optimization verification, a semantics of the GraalVM IR has been developed based on the sea-of-nodes semantics, but extended to include heap-based object allocation, interprocedural calls, exception handling, etc. This work has been accepted to the 19th International Symposium on Automated Technology for Verification and Analysis. A pre-print of the paper is available on arXiv [14].

1.3 Compiler Verification

The end-to-end verification of a compiler is a worthwhile effort and a requisite step for full-stack verification of a software project. However, it is an mammoth undertaking requiring large contributions from experts in the field. As a result, it is not common practice for most languages, however, there have been notable projects in the area.

Painter [15] was the first to explore the idea of compiler verification; the paper presents a collection of manual proofs for a simple compiler of arithmetic expressions to a low-level instruction set.

CompCert [16] is the most notable compiler verification project. CompCert utilizes the Coq interactive theorem prover to verify the compilation of a subset of the C Programming Language. CompCert verifies 20 transformation phases between 11 intermediate languages. For each intermediate language, a formal semantics was developed and each transformation phase was verified to be semantics preserving. CompCert covers verification from after parsing and type-checking through to the generation of assembly code. CompCert relies upon the formal semantics of all 11 intermediate languages to be specified accurately. Extensive fuzzy testing of the verified components of CompCert [1] found no bugs, which provides sufficient confidence that the formal specifications are indeed accurate.

Jitawa [17] is a verified compiler for the Lisp programming language. The input language for Jitawa, Lisp, is straight-forward, however Jitawa has a focus on end-to-end verification which CompCert lacks. The Jitawa compiler assumes that every program terminates and does not offer any guarantees on the correctness of non-terminating programs.

CakeML [18] expands on prior compiler verification efforts, formally verifying the entire compilation process of a complex and practically used language, from lexing and parsing down to the machine code generation. CakeML also introduces a novel technique of bootstrapping a verified compiler. Through the use of bootstrapping, the CakeML project achieves a very minimal trusted computing base.

1.4 Optimization Phase Verification

Due to the inherent and increasing complexity of compiler optimizations, coupled with their natural cohesion with verification techniques, verifying the optimization phase of a compiler is a rich area of study. We survey the literature to provide an overview of the various approaches.

Kozen and Patron [19] are one of the first to give compiler optimizations special treatment in the field of formal verification. Kleene Algebra with Tests (KAT), introduced by Kozen [20], can be used to encode terminating programs. Kozen and Patron [19] show how this encoding naturally leads to optimization proofs using purely equational reasoning. The technique that they introduce does not give a treatment of non-terminating programs which cannot be expressed within KAT. Hand-written proofs are given with respect to the language-agnostic specification of optimizations in KAT. Their technique is unconcerned with a mechanized approach to optimization implementation or verification.

Lacey *et al.* [21] express optimizations as rewrites of a control-flow graph. Computation tree logic (CTL) is used to express side conditions of when the graph rewrites can be applied. They argue that using CTL to express side conditions allows mechanized checking of applicability conditions, and therefore mechanized application of transformations. The TRANS language is introduced for expressing conditional control-flow graph rewrites on a simple imperative language, L_0. It is argued through extensive example encodings that TRANS is a suitably expressive language for encoding both local and global transformations based on program path behaviour.

The specification of the TRANS language does not provide an optimization verification procedure. Mansky and Gunter [22] give the TRANS specification merit as a verification tool by mechanizing the semantics within the Isabelle/HOL interactive theorem prover. During the course of their mechanization effort, they discovered useful predicates and definitions which they introduced as an extension to the TRANS specification. The practical use of their mechanization is then demonstrated by verifying the correctness of an algorithm for transforming a control-flow graph into SSA form.

Mansky and Gunter [23] go on to investigate the application their earlier work of mechanizing TRANS to the problem of concurrent program optimizations. The existing formalization of TRANS which performs transformations with respect to a control-flow graph is lifted to PTRANS which specifies transformations of threaded control-flow graphs [24]. As in their initial TRANS mechanization, the authors provide evidence of the practical use of their PTRANS mechanization by verifying a redundant store elimination optimization for a language in the style of LLVM. The choice of optimization is appropriate as it is strongly influenced by a concurrent environment.

1.5 Domain Specific Languages

Domain-Specific Languages (DSLs) are a natural choice for compiler optimizations, because at their core, compiler optimizations comprise a class of conditional term rewriting rules. One of the first papers to explore the implementation of an optimization DSL was the Sharlit DSL [25], Sharlit primarily simplifies the implementation of data-flow analysis and transformations which are dependent on data-flow analysis results.

The Gospel [26] language was the first optimization DSL to introduce the idea of automated analysis of optimizations expressed in the language. The Gospel language is a formal notation for specifying compiler optimizations. The optimization phases which it generates can be used to analyse dynamic properties of each optimization, including analysing interactions

between optimizations, such as order of application.

Optimization specification languages such as Cobalt [27] and Rhodium [28] take the analysis of transformations one step further by automatically proving the soundness of optimizations expressed in the languages. Both of these languages use the approach introduced by Lacey *et al.* [21], i.e. control-flow graph rewriting based on CTL side conditions. To allow for automated soundness proofs these languages limit the expressibility of the original approach.

R. Tate *et. al* [29] introduce a novel technique which generates new optimizations by example. Concrete example programs are given before and after an optimization application. The transformation from the example is then generalized to a more broadly applicable optimization by ensuring that the generalized optimization is provably correct.

2 Methodology

2.1 Overall Approach

Our approach can be expressed as:

1. encode the IR semantics into Isabelle/HOL;
2. define a proof obligation for the correctness of optimizations in Isabelle/HOL;
3. write a DSL to express optimizations which can translate into Isabelle/HOL;
4. encode the optimizations in the DSL;
5. prove the optimizations satisfy the correctness obligation; and
6. integrate verified optimizations into the existing compiler infrastructure.

For this project, we have chosen to use the Isabelle/HOL interactive theorem prover as a tool to interactively prove optimizations.

As with any formal verification effort, software must be shown to be correct with respect to a specification. In the case of compiler optimizations, the optimizations must be semantics preserving. To show that optimizations are semantics preserving, we first need a formalized semantics of the compiler's IR (1).

Once we have defined the semantics of the IR, we need to define the conditions for IR transformations to be semantics preserving (2). This definition acts as our correctness proof obligation — if we can show that the semantics are always preserved by an optimization, then that optimization is verified.

In order to make the verification of new and existing optimizations as accessible as possible, we aim to be able to express optimizations in a DSL (3). The proposed DSL aims to be clear and suitably expressive for compiler developers. One output target of the DSL is the optimization expressed in Isabelle/HOL, allowing proof of correctness using an appropriate obligation. Java code to implement the optimization is another possible output target. This is discussed in more detail in Section 2.3.

Once we have established a suitably expressive DSL, we can encode each of the existing optimization rules implemented in GraalVM (4). This will allow us to ensure that the DSL is capable of handling arbitrary optimizations. The Isabelle/HOL DSL target will generate a proof obligation to show that the optimization is semantics preserving. For each optimization we encode, we will need to show the proof obligation is satisfied (5).

Finally, the verified optimization phases will need to be integrated back into the compiler (6). The exact approach to this will require further investigation. Section 2.3 describes the two approaches that are being considered.

2.2 Tasks

The proposed project exceeds the scope of a single PhD project. For completeness, we describe the approach, tasks, and deliverables of the whole project. Where tasks and deliverables are listed, a prefixing asterisk indicates the tasks/deliverables which are prioritized as the focus of this PhD.

We break our approach down into three types of tasks; each type of task has different deadline requirements. *Foundational tasks* build requisite infrastructure for further verification. Strict deadlines can be set for the completion of these

tasks with possible later stage modification. *Incremental tasks* progressively expand verification coverage. These efforts are ongoing with no fixed deadline, but often a metric can be derived to measure progress. *Validation tasks* help increase confidence that *incremental tasks* are correctly implemented. The requisite infrastructure for these tasks can have deadlines but they will be performed periodically throughout the project.

The following *foundational tasks* build infrastructure to support verification. In order to allow verification we must:

1. * develop a formal notation to represent the GraalVM sea-of-nodes based IR;
2. * encode the formal notation into the Isabelle/HOL theorem prover;
3. * develop an (executable) semantics framework which enables semantics to be specified for each node;
4. define and encode a semantics preservation proof obligation;
5. * design a DSL for expressing compiler optimizations;
6. * generate Isabelle/HOL encoded optimizations from the DSL;
7. * generate code to perform optimizations from the DSL; and
8. * integrate the generated code into the GraalVM compiler.

The following *incremental tasks* build upon the output of the *foundational tasks* to expand the set of verified optimizations:

1. encode a formal semantics for each node of the GraalVM IR in the Isabelle/HOL interactive theorem prover;
2. express each existing optimization phase in the DSL; and
3. * verify that each encoded optimization satisfies the proof obligation.

Finally, the *validation tasks* help ensure that each *incremental task* is implemented correctly. These tasks correspond with the preceding *incremental tasks*:

1. execute test programs in both GraalVM and the executable Isabelle/HOL semantics, ensuring both result in the same value; and
2. perform optimizations in both GraalVM and the executable Isabelle/HOL optimizations, ensuring the optimized graphs are structurally equivalent; and
3. attempt to verify optimizations which were known to have bugs, ensuring that our specification is able to identify these bugs.

2.3 Challenges

Whilst the main challenge of the project will be the inherent complexity which accompanies any verification effort, our project is also accompanied by unique difficulties:

1. Establishing a semantics for the existing compiler implementation with sufficient confidence of a correct formalization.
2. Faithful translation of existing optimization implementations to an abstract notation for expressing optimizations.
3. Incrementally integrating a verified optimization phase into the existing compiler infrastructure.

This section attempts to address each of these challenges and the proposed solution to resolve the issue.

During the development of an executable semantics for the IR of the compiler, we want to ensure that our semantics corresponds with the GraalVM implementation. This is a non-trivial task. Fortunately, we are able to utilize the existing compiler test cases to increase our confidence in correctness. The existing GraalVM compiler test cases provide expected execution output for a host of representative Java programs. We can execute these programs using our executable Isabelle/HOL semantics and compare the generated output with the expected output. While this does not provide strong confidence, particularly as it only compares a single output value and not the whole execution state, it does provide sufficient confidence for our purposes at this stage.

Next, our encoding of the existing optimizations in our notation must remain as faithful as possible to the original implementation. Faithful implementation of the optimizations allows us

to evaluate the success of the project by highlighting any potential implementation bugs. We propose a validation approach based on Figure 4. We assert that given a correct *encoder* program and faithfully encoded optimizations, that the diagram commutes. The process for validating correct encodings of optimizations (and as a side-effect, the *encoder* program) is as follows. We take a collection of input programs and, for each input program graph, G

1. Encode G into the Isabelle/HOL representation, producing E.
2. Trigger an Isabelle/HOL optimization of the encoded unoptimized graph, E, to produce E'.
3. Trigger a GraalVM compiler optimization of the graph, producing G'.
4. Encode the GraalVM optimized graph, G', into Isabelle/HOL, this process should produce an equivalent E'.
5. We then show by structural equality that both approaches for reaching E' are equivalent, and therefore that the diagram commutes.

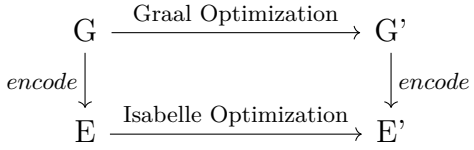

Figure 4. Commutative diagram for validating encoded optimizations

This process, for a sufficiently large collection of input programs, gives us a high degree of confidence that the encoded optimizations are faithful.

The final unique challenge of integrating verified optimizations into the compiler requires further research into suitable approaches, however we can outline two potential approaches.

The first approach uses the code generation facilities of Isabelle/HOL, which can translate Isabelle/HOL definitions into a range of target languages. Fortunately, one of the target languages for code generation is Scala which compiles to Java bytecode to run on the Java Virtual Machine (JVM). This property enables interoperability between Java and the compiled bytecode from Scala, enabling native calls to generated procedures which perform optimizations. Through compiling the generated Scala with a small translation layer to allow direct access and manipulation of the GraalVM IR, the compiler is then able to use the generated optimization code. The practicality of this approach still requires evaluation through metrics such as memory footprint and runtime efficiency. An initial proof-of-concept integrated optimization phase will need to be developed. We can then evaluate the efficiency and eliminate potential bottlenecks where it is practical to do so. As optimization phases are modular, implementing generated optimizations will be a process of incrementally replacing phases.

The second approach directly generates code to perform optimizations from the DSL. We would simply implement a secondary DSL compiler backend to generate Java code, in addition to Isabelle/HOL theories. This approach gives us more control over the efficiency of generated output, which makes it a more practical solution from an efficiency point of view. However, we then encounter the problem of correctness for our DSL translation. In order for us to claim that we have verified the optimization phase, we need to be confident that the optimization code generated from our DSL corresponds exactly with the Isabelle/HOL encoding. If this approach is found to be more practical, then we will need to investigate a way to verify that both DSL backends produce equivalent output, i.e. that the proof obligation is equivalent to the implementing code.

3 Outcomes

We partition the expected deliverables into two categories. Project deliverables consist of the deliverables that are essential to the success of the project. In addition to the project deliverables, we expect to produce investigation deliverables, which are tools to help our understanding of the system, or to otherwise assist our development.

3.1 Project Deliverables

- * A formal specification of the GraalVM IR.
- * An executable semantics of the GraalVM IR.
- * A DSL for expressing optimizations, including the following features:
  - a language specification;
  - syntax highlighting using Language Server Protocol; and
  - debugging tools.
- * A DSL backend to transform the encoded optimizations into:
  - Isabelle/HOL theories including automating proof or counter-example generation through SMT solvers; and
  - code to perform optimization transformations.
- A sufficient collection of supporting theories to enable mostly automated verification.

3.2 Investigation Deliverables

- * A collection of well-formedness properties of the GraalVM IR. These properties will be used to prove that all well-formed IRs have a corresponding semantics, i.e. the semantics of the IR are total. The well-formedness properties can be integrated back into the compiler to ensure all constructed IR graphs are well-formed.
- An *encoder* program to transform GraalVM IR graphs to the Isabelle/HOL IR graph notation.
- * A GraalVM IR interpreter derived directly from the formal semantics.
- A fuzzy testing-based bug finder for Java compiler implementations using the CSmith approach [1].

4 Timeline

4.1 Proposed Publications

The following proposed publications serve to document the process of verifying an existing compiler implementation.

A Formal Semantics for the GraalVM IR addresses formalizing a sea-of-nodes based IR in the context of procedure calls and a heap (accepted for ATVA 2021).

Using Automated Testing to Validate a Compiler Optimization Verifier documents the techniques employed to validate that our semantics and optimization encodings are faithful to the original implementation.

Verification of Existing GraalVM Optimization Suites documents the challenges encountered and resolution methods involved in verifying at least one existing optimization suite.

An Expressive and Verifiable Compiler Optimization DSL documents all of the required design decisions to allow the DSL to optimize for both expressiveness and automated verification.

Incremental Compiler Optimization Verification documents the process and technical challenges of integrating verified optimization code incrementally into an existing compiler.

4.2 Proposed Venues

Here we enumerate the potential venues suitable for publishing the proposed research. The topics of the proposed venues include formal verification, testing, programming languages, and general software engineering.

Conferences

1. Automated Technology for Verification and Analysis (ATVA)
2. Interactive Theorem Proving (ITP)
3. International Symposium on Formal Methods (FM)
4. Automated Software Engineering (ASE)

5. Computer Aided Verification (CAV)
6. International Conference on Software Testing, Verification, and Validation (ICST)
7. Conference on Programming Language Design and Implementation (PLDI)
8. Symposium on Principles of Programming Languages (POPL)
9. Asia-Pacific Software Engineering Conference (APSEC)
10. Asian Symposium on Programming Languages and Systems (APLAS)
11. International Conference on Formal Engineering Methods (ICFEM)

Journals

1. ACM Transactions on Programming Languages and Systems (TOPLAS)
2. ACM Transactions on Computational Logic (TOCL)
3. Formal Methods in System Design
4. Logical Methods in Computer Science (LMCS)
5. Theoretical Computer Science (TCS)
6. Formal Aspects of Computing: applicable formal methods (FAOC)
7. Journal of Logic and Computation
8. Science of Computer Programming
9. ACM Transactions on Software Engineering and Methodology (TOSEM)
10. ACM Transactions on Architecture and Code Optimization (TACO)
11. Acta Informatica
12. Software Testing, Verification and Reliability (STVR)

4.3 Gantt Chart

The Gantt chart presented in Figure 5 lists the numbered *foundational tasks* from Section 2. It also lists the unnumbered *incremental tasks* as a broad time period wherein the task will be performed.

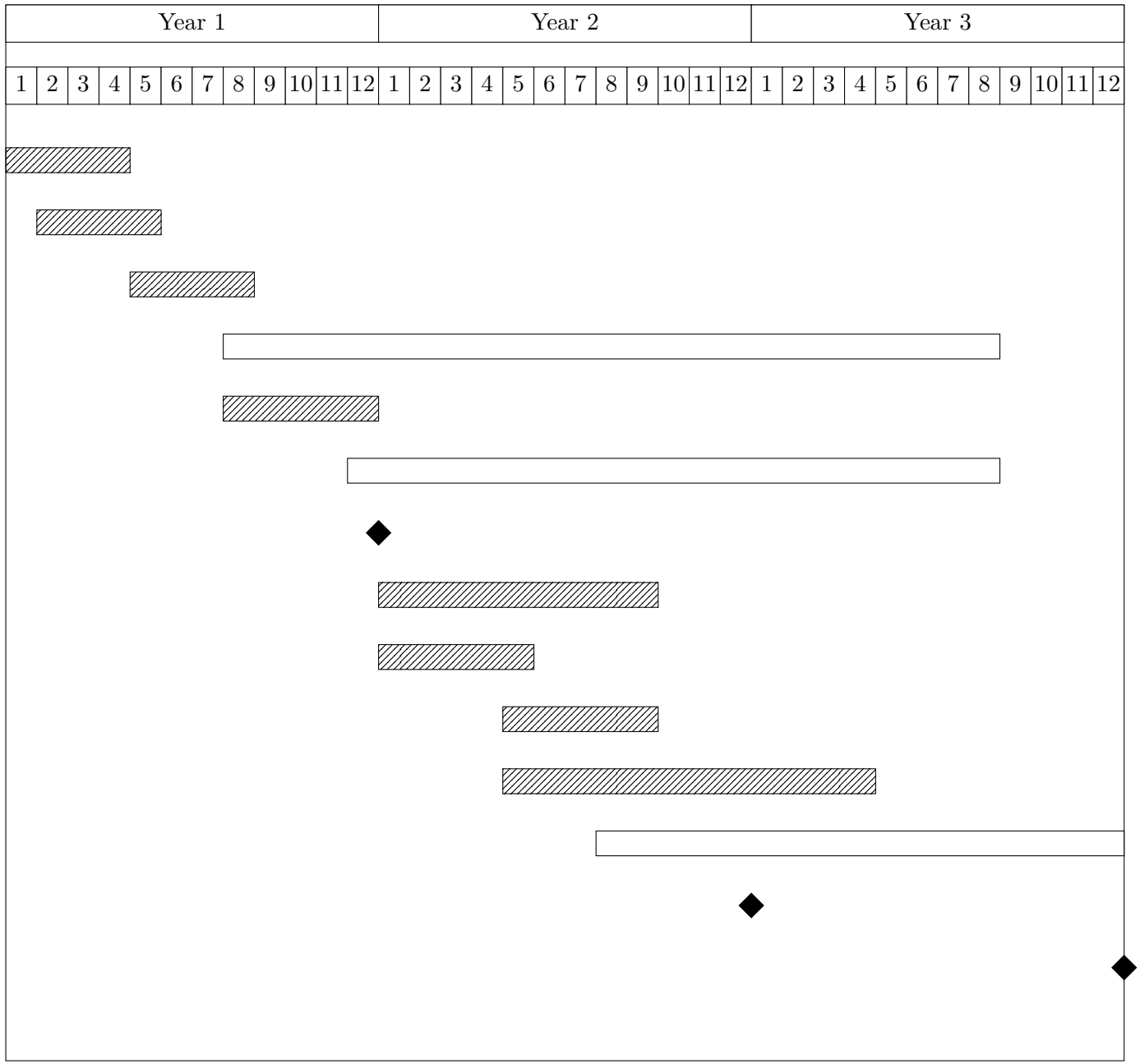

Figure 5. Project Gantt Chart

References

[1] 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, 2011, p. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[2] z, "Strangest language feature," https://stackoverflow.com/a/2001861/13430616, Jan 2010.

[3] E. W. Dijkstra, "On the reliability of programs," n.d., circulated privately. [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF

[4] Oracle, "GraalVM: Run programs faster anywhere," 2020. [Online]. Available: https://github.com/oracle/graal

[5] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko, "One VM to rule them all," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 187–204. [Online]. Available: https://doi.org/10.1145/2509578.2509581

[6] Y. Futamura, "Partial evaluation of computation process—an approach to a compiler-compiler," *Higher-Order and Symbolic Computation*, vol. 12, no. 4, pp. 381–391, Dec 1999.

[7] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer, "Practical partial evaluation for high-performance dynamic language runtimes," in *PLDI 2017*. New York, NY: ACM, 2017, pp. 662–676.

[8] C. Click and K. D. Cooper, "Combining analyses, combining optimizations," *TOPLAS*, vol. 17, no. 2, p. 181–196, Mar. 1995.

[9] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88. New York, NY, USA: Association for Computing Machinery, 1988, p. 12–27. [Online]. Available: https://doi.org/10.1145/73560.73562

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, p. 451–490, Oct. 1991. [Online]. Available: https://doi.org/10.1145/115372.115320

[11] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 2, p. 181–210, Apr. 1991. [Online]. Available: https://doi.org/10.1145/103135.103136

[12] C. Click, "Global code motion/global value numbering," in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, ser. PLDI '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 246–257. [Online]. Available: https://doi.org/10.1145/207110.207154

[13] D. Demange, Y. Fernández de Retana, and D. Pichardie, "Semantic reasoning about the sea of nodes," in *CC 2018*. New York, NY: ACM, 2018, p. 163–173.

[14] B. J. Webb, M. Utting, and I. J. Hayes, "A formal semantics of the GraalVM intermediate representation," 2021.

[15] J. McCarthy and J. Painter, "Correctness of a compiler for arithmetic expressions," in *Proceedings of a Symposium in Applied Mathematics*, vol. 19, 1967, pp. 33–41.

[16] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand, "CompCert - A Formally Verified Optimizing Compiler," in *ERTS 2016*. Toulouse, France: SEE, Jan. 2016. [Online]. Available: https://hal.inria.fr/hal-01238879

[17] M. O. Myreen and J. Davis, "A verified runtime for a verified theorem prover," in *International Conference on Interactive Theorem Proving*. Springer, 2011, pp. 265–280.

[18] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: A verified implementation of ML," in *POPL '14*. ACM Press, Jan. 2014, pp. 179–191.

[19] D. Kozen and M.-C. Patron, "Certification of compiler optimizations using Kleene algebra with tests," in *International Conference on Computational Logic*. Springer, 2000, pp. 568–582.

[20] D. Kozen, "Kleene algebra with tests," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 19, no. 3, pp. 427–443, 1997.

[21] D. Lacey, N. D. Jones, E. Van Wyk, and C. C. Frederiksen, "Proving correctness of compiler optimizations by temporal logic," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 283–294. [Online]. Available: https://doi.org/10.1145/503272.503299

[22] W. Mansky and E. Gunter, "A framework for formal verification of compiler optimizations," in *International Conference on Interactive Theorem Proving*. Springer, 2010, pp. 371–386.

[23] W. Mansky and E. L. Gunter, "Verifying optimizations for concurrent programs," in *First Int. Workshop on Rewriting Techniques for Program Transformations and Evaluation, WPTE 2014, July 13, 2014, Vienna, Austria*, 2014, pp. 15–26.

[24] J. Krinke, "Context-sensitive slicing of concurrent programs," in *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, 2003, pp. 178–187.

[25] S. W. K. Tjiang and J. L. Hennessy, "Sharlit—a tool for building optimizers," *SIGPLAN Not.*, vol. 27, no. 7, p. 82–93, Jul. 1992. [Online]. Available: https://doi.org/10.1145/143103.143120

[26] D. L. Whitfield and M. L. Soffa, "An approach for exploring code improving transformations," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 6, p. 1053–1084, Nov. 1997. [Online]. Available: https://doi.org/10.1145/267959.267960

[27] S. Lerner, T. Millstein, and C. Chambers, "Automatically proving the correctness of compiler optimizations," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 220–231. [Online]. Available: https://doi.org/10.1145/781131.781156

[28] S. Lerner, T. Millstein, E. Rice, and C. Chambers, "Automated soundness proofs for dataflow analyses and transformations via local rules," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 364–377. [Online]. Available: https://doi.org/10.1145/1040305.1040335

[29] R. Tate, M. Stepp, and S. Lerner, "Generating compiler optimizations from proofs," in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 389–402. [Online]. Available: https://doi.org/10.1145/1706299.1706345