AUTOMATIC VERIFICATION OF MIR OPTIMIZATIONS IN THE RUST COMPILER

A Research Paper submitted to the Department of Computer Science
In Partial Fulfillment of the Requirements for the Degree
Bachelor of Science in Computer Science

By

Nathan Whitaker

April 26, 2021

On my honor as a University student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments.

ADVISOR
Worthy Martin, Department of Computer Science

Automatic Verification of MIR Optimizations in the Rust Compiler

Nathan Whitaker
University of Virginia
Charlottesville, Virginia, USA
nw7ct@virginia.edu

ABSTRACT

As compilers become increasingly advanced, the number and level at which optimizations are performed has risen as well. With optimizations, however, come the chance of miscompiling a program and introducing a bug or vulnerability. The Rust programming language, which aims to be a safe and performant systems programming language, introduced a mid-level intermediate representation (MIR) of programs in its compiler, with one goal being the ability to create optimizations that are applied at a higher level, and early on during compilation. This project aims to propose a tool for the automatic verification of MIR optimizations performed by the Rust compiler. Similar tools have been created for optimizations performed in the LLVM compiler infrastructure, but the semantics and model of LLVM programs differ from Rust, and existing tools work on a lower-level representation than MIR. This project aims to adapt the strategies used in existing tools to meet the requirements of MIR optimizations in the Rust compiler.

1 INTRODUCTION

Modern compilers perform a dizzying array of optimizations in the course of producing a program. At their best, compiler optimizations not only squeeze more performance out of existing code, but also empower programmers to express programs at higher levels of abstraction without fear of performance loss. However, at their worst, compiler optimizations can unintentionally change a program's semantics, introducing unexpected behavior or potential security vulnerabilities. Rust [11] [19] is a relatively novel, memory-safe systems programming language with a focus on helping developers write robust, performant software. Another important feature of Rust is its expressivity, which allows developers to create higher-level abstractions that have little to no runtime cost. In order to achieve these goals, the Rust compiler relies heavily on aggressive optimizations. The Rust compiler primarily uses the LLVM compiler framework [7] for code generation. This allows the compiler to tap into the wealth of optimizations and analysis that LLVM has already developed, and produces code that is on-par with state-of-the-art C/C++ compilers such as Clang [6] and GCC [18]. To become less reliant on LLVM's optimizations and improve performance further, there has been an interest in performing optimizations in the Rust compiler itself. As these optimizations are implemented, the risk of miscompilation rises, and unsound transformations have the potential to violate Rust's memory safety guarantees[15]. To remedy this, there is a need for improved verification tools to aid compiler developers in writing and maintaining these optimizations.

2 BACKGROUND

2.1 An Introduction to MIR

The Rust compiler originally performed little to no optimization, instead it fully relied on LLVM for optimization. Internally, the compiler used a high-level intermediate representation (HIR) which closely resembled the surface syntax, then lowered the HIR into LLVM's intermediate representation (LLVM IR) for optimization and code generation. In 2016, a mid-level intermediate representation (MIR) was introduced to the compiler, with one of its main goals being to facilitate performing optimization directly on MIR [10]. Since the introduction of MIR, the compiler now runs a series of optimization passes on a program's MIR before producing LLVM IR.

MIR is based on the control-flow graph of a program, and is fully typed [1]. The MIR for a given function consists of a series of basic blocks. A basic block is an atomic unit of the program's control-flow graph, and is made up of a sequence of statements followed, optionally, by a terminator, which corresponds to an edge in the control-flow graph.

MIR, being very much internal to the Rust compiler, does not have a stable definition and undergoes frequent changes. There is no surface syntax for MIR, much less a stable one, rather it is defined as a collection of data structures [1]. MIR has enough explicit information to make program analysis feasible, though its unstable nature makes the consumption of MIR challenging to maintain.

While the semantics of LLVM IR are documented, the semantics of MIR are very much undocumented and undecided [3]. There has been recent work to model the semantics of MIR, the most notable being the work on the MIR interpreter (MIRI), and the aliasing model it adopts, Stacked Borrows [5].

2.2 Translation Validation

Modern compiler optimizations can be complex and nontrivial to implement. This poses an issue, because even if the theoretical soundness of an optimization has been formally proven, this does not mean that the implementation of that optimization is necessarily sound. One approach that is able to verify the implementation itself is translation validation [13]. Translation validation compares the representation of a program or program fragment before and after an optimization has been applied. By checking that the resulting representation is semantically equivalent to the original, it can be ensured that the optimization has not altered the meaning of the program, and is therefore a valid transformation.

The challenge of translation validation lies in how exactly to prove the semantics are equivalent. This is typically achieved by

tracking the invariants for the program, as well as tracking the state of the program through symbolic execution. Simplistically, if the invariants of the transformed program are a subset of the original, and the observable state at the end of execution is equivalent, then the transformation can be considered to preserve the semantics of the original program. It is important, however, to note that translation validation cannot prove that an optimization is always correct, rather it can only prove that a specific application of an optimization is correct. In this way it is a somewhat weaker form of verification, as its ability to detect errors is reliant upon the concrete cases it encounters.

3 RELATED WORK

3.1 MIR Validation in the Rust Compiler

A limited form of MIR validation [17] was introduced to the Rust compiler after it was found that in some cases MIR optimizations could produce invalid code and introduce memory unsafety [12]. The MIR validator in the current Rust compiler takes the form of an additional pass that visits transformed MIR and checks that certain invariants are upheld. The validation occurs at compile-time, and runs after MIR optimizations are applied. This turns potential miscompilations into compiler errors, making it easier for compiler developers to identify potentially incorrect optimizations. However, only a small number of invariants are checked by the validator, with additional checks being added [16] as new optimization bugs are identified [15]. Due to the ad-hoc nature of the MIR validator, it is useful for preventing the recurrence of potential bugs that have already been identified, but is not readily able to identify unforeseen bugs. Additionally, the MIR validator only checks that certain MIR invariants are upheld, and cannot ensure that an optimization correctly preserves the meaning of a program.

3.2 MIRI: Detection of Undefined Behavior at Runtime

MIRI is a tool that primarily serves to interpret MIR code directly. It additionally serves as a dynamic analysis tool [5] which detects undefined behavior at runtime. MIRI tracks extra information during interpretation in order to issue an error if the program would execute undefined behavior. It is a particularly useful tool for users who write unsafe code, as it is able to catch errors that the compiler cannot detect statically. It is an official project that is maintained by the Rust team, and also serves as a useful guide to creating analysis tools that consume MIR. Similar to the MIR validator, it can help to identify whether a given piece of MIR is correct, but it is instead intended for use by end users rather than compiler developers. This, combined with its inability to ensure that semantics of a program are maintained, mean it is inadequate for the purpose of verifying the soundness of MIR optimizations.

3.3 Alive: Verification of LLVM Peephole Optimizations

The original Alive project [9] focused on verifying only peephole optimizations in LLVM [7]. Its approach was to create a domain specific language (DSL) in which compiler developers could express a specific transformation and its accompanying preconditions. This provided Alive with enough information to test whether or not the transformation was valid, making it possible for compiler developers to check the validity of a given optimization. The downside was that it required optimizations to be re-expressed in a new DSL, which meant that LLVM developers would need to expend extra effort to reap the benefits. This meant that the process was not very automatic or easily integrated into the development process, which limited its usefulness. Additionally, its approach required the optimization to be simple enough to express in the DSL which ruled out the verification of complex optimizations.

3.4 Alive2: A Translation Verifier for LLVM IR

Alive2 [8] is a project that builds upon the original Alive project, and provides tools for verifying optimizations in the LLVM compiler infrastructure [7]. Most relevantly, Alive2 implements a translation verification tool that is easily integrated into LLVM's unit testing and overall development process [14]. It works, broadly, by lowering LLVM IR into logic supported by the Z3 SMT solver [4], then tracking the state of the program through the process of symbolic execution. Specifically, it checks that the transformed IR is a refinement of the original IR, meaning that the transformed program must not introduce additional non-determinism and the observable state must not differ between the two versions. Due to its ease of integration with LLVM's development process, Alive2 seems to be widely used by LLVM developers. Alive2 serves as the model for much of this research, as its approach is generally applicable to MIR and proves the feasibility and usefulness of tooling like it. In fact, Rust compiler developers have explicitly expressed a desire for tooling analogous to Alive2 that operates on MIR rather than LLVM IR.

4 SYSTEM DESIGN

4.1 Overview

This work proposes a tool, MIR-TV (MIR Translation Validator), which plugs directly into the Rust compiler to prove the validity of MIR optimizations. MIR-TV would run during the MIR optimization phase, where the compiler runs individual MIR optimization passes on the program's MIR. After each optimization pass, MIR-TV would accept the original MIR and the optimized MIR as input, then run the validation process, terminating compilation with an error if the optimization was found to be invalid. MIR-TV would ideally be integrated into the Rust compiler's unit testing, serving to automatically verify new optimizations as they are added, and would also be able to run optionally during the compilation of a user program. This would be primarily be used by Rust compiler developers to aid the implementation and testing of new MIR optimizations.

The actual verification would consist of three main stages: lowering MIR to SMT logic , performing symbolic execution and state tracking, and checking that the optimized MIR is a refinement of the original MIR. This process would run for each function in the program, and due to the fact that the verification only needs to inspect the current function this process could be easily parallelized. If at the end of this process all functions were verified to be correct, then the entire MIR transformation would be considered correct.

4.1.1 MIR Lowering. The process of refinement checking requires using an SMT solver in order to determine whether or not the refinement relationship is satisfiable. In order to make this more tractable, MIR would be lowered into a representation that more closely resembles the SMT encoding for each type of value in MIR. The simplest values to encode in SMT logic are scalars. Scalars are naturally represented in SMT, as most SMT solvers support integer and floating point logics. Scalars, however, may also be uninitialized, so a flag must be stored indicating whether the value is initialized or not. An uninitialized value is propagated by most instructions and is used to keep track of whether or not an optimization introduces nondeterminism. Aggregate types such as structures and enumerations would be represented by a bitvector consisting of the concatenation of each of the aggregate type's fields. Pointers would be represented by an ID which corresponds to a specific allocation, as well as an offset indicating which byte of the allocation is pointed to.

Memory would be modeled as a mapping from an allocation ID to a specific memory block. A memory block would store extra information to track the size of the allocation, the alignment, and a flag indicating whether the allocation is alive. The actual value of the memory block would be represented by an array of bytes, along with a mask indicating which specific bytes were initialized or uninitialized. Instructions that store into the memory block would mark the bytes written to as initialized. A memory access that read bytes uninitialized memory would be considered UB and would set a flag indicating UB had occurred.

MIR-TV would inspect the MIR of a function's body, collecting a map of all of the local variables used, then use this to build a map from the variable name to its SMT representation. This information would then be used during symbolic execution to track program state.

4.1.2 Symbolic Execution. At this point, MIR-TV would begin executing the MIR of the function body statement by statement. The program's state would be represented by three major components: a mapping of all local variables to their value representation, a memory instance representing local allocations, and a flag indicating whether UB has been executed.

The implementation would most likely reuse much of the code from MIRI [2], but modify the interpreter to support symbolic execution. This would help to make sure that the semantics match how MIR would normally be executed, as well as reduce the amount of implementation work required. As each statement is executed, the program state would be updated accordingly. If at any point execution performs an action that is UB according to MIRI's semantics, the UB flag would be set. External function calls would be modeled as having an undefined return value, and if pointers are passed to the function call, then the memory pointed to by the pointer would be considered undefined. In general an undefined value would be modeled by a fresh variable in SMT, representing a specific but unknown value.

4.1.3 Refinement Checking. After executing both the original and optimized MIR for the function, MIR-TV would then use the final program states and values to determine if the optimized MIR refines the original MIR. For values, the condition for refinement would be that the optimized version of the value is at least as defined as the original version. In other words, given a value v in the original program and its corresponding value v' in the optimized program, v' refines v if $v' = v$, or if v is undefined and v' is defined. If the value v' is undefined while v is defined, then v' does not refine v. The final program state s' of the optimized program would be considered to refine the final program state s of the original program if the return value in s' refines the return value of s, or if the UB flag of s is set. In this case, the original program would have executed UB, and so the optimized program must be at least as defined as the original. If either of these conditions are met, then the optimized MIR is considered to refine the original MIR, and the transformation would be considered sound.

4.2 Challenges

4.2.1 MIR Semantics and Undefined Behavior. The semantics of MIR are not yet decided, and due to its novelty and instability, there has not been much work towards a formal specification of MIR. This poses a challenge for modeling MIR, as the best effort that can be made is to observe the current MIR semantics and aim to model that. In a similar vein, the question of what exactly is undefined behavior in MIR (and Rust in general) is not yet answered. There have been official efforts towards identifying what exactly is UB in Rust, as well as towards defining Rust's memory model [20], but this is a work in progress and far from being part of the language specification. As an approximation, MIR-TV would adopt the MIR semantics and forms of UB recognized in MIRI [5], which currently has the most comprehensive model of MIR and its behavior.

4.2.2 Type Polymorphism in MIR. One problematic aspect of MIR for analysis is that MIR allows functions and instructions to be polymorphic. For example, it is perfectly valid for the return type of a function in MIR to be a type variable. The Rust compiler does not monomorphize code by instantiating type variables until the code generation stage, when it translates MIR to LLVM IR. LLVM IR, unlike MIR, only allows for polymorphism over integer bit width. In the case of Alive2, which operates on LLVM IR, this is easily resolved because there are a fixed number of integer bit widths, so testing all possible widths is feasible, albeit a bit computationally intensive. The polymorphism in MIR, however, is unrestricted; given only the MIR of a polymorphic function, there are an infinite number of types which could be accepted by that function.

In order to solve this issue, several options were considered. First, the monomorphization process could essentially be performed within MIR-TV. In theory, at the time of verification the MIR for the entire program is available. This means it would be possible to collect all usages of each function by inspecting every function body in the program, then enumerating the types that are actually used to instantiate each polymorphic function. Then, a monomorphized version of the function could be created for each combination of concrete types by instantiating the function's type variables with the actual types. At this point, the polymorphism would be eliminated and the issue would be solved. This has a few issues, however, the most important being that MIR-TV would lose the ability to reason locally about a given function. Instead of only needing to inspect the current function, the entire program would need to be inspected, which makes analysis much less composable. Additionally, this could drastically impact verification performance

if functions are instantiated with many different types. This would also mean that the verifier is synthesizing new MIR during the course of its analysis, which introduces room for error and makes the analysis less "pure," in that the verification would perform on MIR that wasn't directly emitted by the compiler. A second option is that type polymorphism could be modeled in the SMT encoding of MIR. The naive encoding would require both the value and type to be universally quantified, which would mean reasoning about a variable which can take on any value for any type. This would most likely make it impossible to perform any useful analysis because the SMT equation may be underspecified. It is certainly possible that there is a workable representation for polymorphic functions, but the complexity of determining this is beyond the scope of this project. The final option is simply to exclude polymorphic function bodies from analysis. This is clearly the simplest choice, though the downside is that it limits the usefulness and power of the verification performed by MIR-TV. For the sake of this project, however, this is the most reasonable solution, and figuring out a proper SMT encoding for polymorphic functions is left to future work. Instead of failing upon encountering a polymorphic function, MIR-TV would simply skip the verification of polymorphic functions and then issue a warning indicating the functions that were not verified.

4.2.3 SMT Encoding. The most challenging aspect of the project was designing a suitable SMT encoding for each aspect of MIR. This proved difficult due to a general lack of familiarity with SMT solvers, as well as the complexity of MIR. Without concrete experience with SMT solvers, it was difficult to tell whether the proposed design was acceptable. This combined with the fact that MIR is relatively high level (despite its name), and that SMT solvers support relatively low level representations, made coming up with a suitable translation between the two representations difficult. The SMT encoding described in this project is therefore a best effort. It is likely that the SMT encoding model would need to be refined and fleshed out in the process of implementing MIR-TV.

4.2.4 The Cutting Edge. One of the greatest challenges of this project was the fact that much of this work sits on the forefront of existing research. For example, Alive2, which is the inspiration for this project, was released in 2020, and the paper describing its methodology is not yet published. This challenge is compounded by the fact that the Rust programming language is a rather new language, with far fewer resources and prior research pertaining to it as compared to long-standing projects such as LLVM, and languages such as C and C++. The novelty of this area of research meant that there were not many clear references to look to in order to understand how this tool might be implemented. The exhaustive design and implementation of MIR-TV is most likely a large enough task that it could feasibly be graduate research. The complexity of the topic combined with the novelty of the area, as well as the lack of resources, made this project very challenging and limited what could reasonably be accomplished given the time constraints.

5 RESULTS

The design and groundwork for MIR-TV, a translation verifier for MIR optimizations, has been developed. Its design allows it to fit within the Rust compiler's unit testing suite, and provides a path for integration within the compiler's development process. The successful integration of tools such as Alive2 into LLVM's development bodes well for MIR-TV usefulness to compiler developers. The design meets the requirements of a modern compiler developer, and its automatic nature should lower the barrier to adoption. MIR-TV compares favorably to existing verification for MIR optimizations in that it is more complete, more automatic, and more able to identify new errors, instead of preventing the recurrence of known errors. Due to time constraints, as this was an individual project, as well as the complexity of the system, an implementation of MIR-TV was not produced. However, the design has been closely tailored to the needs and wants of Rust compiler developers, and the fact that it is modeled after other, successful tools strongly suggests that a concrete implementation would fulfill the goals of the tool well.

6 CONCLUSIONS

A tool to automatically verify the soundness of MIR optimizations in the Rust compiler has been designed. MIR-TV is meant to aid compiler developers and improve the quality of the testing of MIR optimizations in Rust. By modeling the system after existing, successful automatic translation verifiers such as Alive2 [8], MIR-TV is positioned to integrate well within the compiler development process and serve as a powerful tool for identifying miscompilations. MIR-TV has been designed to model the semantics of MIR as closely as possible, and builds off of well-tested, official tooling from the Rust team. It is designed to be easy for developers to use and provides support for the implementation of robust, correct MIR optimizations that can be stabilized and released into production more confidently. The usage of MIR-TV as designed would help to secure the Rust compiler and give greater assurance to end users, improving developer's trust in the correctness of their final programs.

7 FUTURE WORK

The primary focus of future work would be to implement MIR-TV as it is described in this project. An implementation would be able to build off of some existing libraries and tools, as described in the report, but certain areas such as integration with SMT solvers may require extra work. The design of the SMT encoding and model for MIR semantics is not complete, as MIR covers a fairly large number of instructions, intrinsic functions, and operations that would need to be supported. This would not necessarily be challenging, but it would certainly require a large time investment. In addition, support for polymorphic functions would ideally be added. This would require more modeling work, and there does not seem to be prior formal research in this area to inform how this could be approached. Discussions with Rust compiler developers suggest that polymorphic functions may be quite challenging to support, so this would most likely require additional guidance from others. The semantics of MIR are also an estimation, because there is no formal specification for MIR, so an area of future work might also be to help the Rust community in formalizing MIR's semantics. After the implementation of MIR-TV, the final step would be to actually integrate it within the Rust compiler's testing. This would

require getting feedback from the Rust team and Rust compiler developer community in order to officially use MIR-TV in the compiler development process.

REFERENCES

[1] [n.d.]. The MIR (Mid-level IR). https://rustc-dev-guide.rust-lang.org/mir/index.html

[2] 2021. rust-lang/miri. https://github.com/rust-lang/miri original-date: 2015-11-12T21:51:25Z.

[3] Eduard-Mihai Burtescu. 2020. Semantics of MIR assignments, around aliasing, ordering, and primitives. · Issue #68364 · rust-lang/rust. https://github.com/rust-lang/rust/issues/68364

[4] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[5] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. Stacked borrows: an aliasing model for Rust. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–32. https://doi.org/10.1145/3371109

[6] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5.

[7] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California.

[8] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM *(PLDI '21)*. Association for Computing Machinery, 15. https://doi.org/10.1145/3453483.3454030

[9] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably Correct Peephole Optimizations with Alive. *SIGPLAN Not.* 50, 6 (June 2015), 22–32. https://doi.org/10.1145/2813885.2737965 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[10] Niko Matsakis. 2016. Introducing MIR. https://blog.rust-lang.org/2016/04/19/MIR.html

[11] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT '14)*. Association for Computing Machinery, New York, NY, USA, 103–104. https://doi.org/10.1145/2663171.2663188 event-place: Portland, Oregon, USA.

[12] Scott McMurray. 2017. MIR InstCombine introduces copies of mutable borrows · Issue #46420 · rust-lang/rust. https://github.com/rust-lang/rust/issues/46420

[13] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 83–94. https://doi.org/10.1145/349299.349314 event-place: Vancouver, British Columbia, Canada.

[14] John Regehr and Nuno P Lopes. 2020. Alive2 Part 2: Tracking miscompilations in LLVM using its own unit tests – Embedded in Academia. https://blog.regehr.org/archives/1737

[15] Cormac Relf. 2020. Segfaults/corruption when reading an enum in release mode · Issue #77359 · rust-lang/rust. https://github.com/rust-lang/rust/issues/77359

[16] Jonas Schievink. 2020. -Zvalidate-mir: Assert that storage is allocated on local use by jonas-schievink · Pull Request #77369 · rust-lang/rust. https://github.com/rust-lang/rust/pull/77369

[17] Jonas Schievink. 2020. Avoid 'Operand::Copy' with '&mut T' by jonas-schievink · Pull Request #72093 · rust-lang/rust. https://github.com/rust-lang/rust/pull/72093

[18] Richard M. Stallman and G.C.C. Developer Community. 2009. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Scotts Valley, CA.

[19] The Rust Team. 2021. Rust Programming Language. https://www.rust-lang.org/

[20] Unsafe Code Guidelines Working Group. 2021. rust-lang/unsafe-code-guidelines. https://github.com/rust-lang/unsafe-code-guidelines original-date: 2018-07-04T15:06:32Z.