Project Description

1. Motivation

Alive2 [1] is a formal-methods-based tool that—given a pair of functions in LLVM IR—determines whether one of them *refines* the other. Refinement is the top-level correctness criterion for a compiler like LLVM. Refinement is compositional: if every individual optimization performed by the compiler is a refinement, then so is the entire compilation process.

Alive2 has been used—primarily, but not exclusively, by my collaborators and me—to discover and report 97 miscompilation defects in the LLVM middle-end optimization passes. Furthermore, I have discovered and reported 40 miscompilation bugs in LLVM's AArch64 (64-bit ARM) backend, using Arm-tv: an experimental (as-yet-unpublished) fork of Alive2 that my group created.

However, **we did not create Alive2 primarily for our own use: our main goal was to give the LLVM community a reliable correctness oracle for LLVM optimizations.** Thus, in June 2020 I made a web site, `alive2.llvm.org`, based on a fork of Matt Godbolt's popular "Compiler Explorer" software. This site makes it easy for the LLVM community to use the latest version of Alive2, without having to compile it. Compiler Explorer also has a builtin "shortlink" generator, making it easy for people to share specific examples over the internet.

In the 4.5 years that the online Alive2 instance has been running, the LLVM developer community has come to rely on it as a core service. At present, more than 800 issues in the LLVM project's Github instance contain links to `alive2.llvm.org`! Furthermore, the LLVM project's guidelines for adding rewrites to its InstCombine pass (nearly 50,000 lines of C++ implementing local rewrites) strongly suggests that contributors use Alive2 to validate new optimizations. As far as I know, **the LLVM project's adoption of Alive2 represents the most widespread use of formal methods by a production compiler's development community to date.** In effect, Alive2 has become the accepted standard semantics for LLVM IR, alongside the (informal, English-language) "LLVM Language Reference Manual."

In this proposal, I request funding to build upon this existing success by substantially upgrading both the software and hardware for the online Alive2 instance.

2. More and Better Alive-Based Tools

So far, the only Alive2-based tool that is available for use online is Alive-tv: a command-line tool that performs translation validation by either proving that a "source" function is refined by a "target" function, or else it provides a counterexample demonstrating failure of refinement. However, there are several other tools that build upon the Alive2 core, that we (the Alive2 team) believe are useful and should be made available for easy, online use by the LLVM community.

2.1. **Alive-exec.** By design, Alive2 reasons about all possible inputs to the pair of functions that it is validating. This, of course, is the point of Alive2, but it's not always what we

want. Sometimes we simply want to examine a single execution of a piece of code, in order to understand or demonstrate some property of interest.

To fill this niche, we created Alive-exec, an interpreter for LLVM IR that basically just iteratively applies Alive2's semantics to a concrete program state. This tool is currently pretty rudimentary: since we have not yet implemented a parser for input values, the user must write a function in LLVM IR that calls the function of interest, supplying its input values.

Alive-exec's output is also very basic: it simply lists the instructions that it executes, and the resulting SSA and memory values. I believe that a better way to visualize its output would be similar to the GDB debugger's "split view," which shows both the program being debugged and also the program state, in separate windows. The split view mode in Alive-exec will show the LLVM IR that is being interpreted in one window, and the program state in a different window, and will provide a simple interface allowing the user to step forward and backwards in the program's execution. Since the Compiler Explorer software already supports multiple panes, and already supports execution of generated code (in a sandbox), I believe that it will not be too difficult to integrate Alive-exec.

Effort estimate: Three months (by the PhD student) to implement argument parsing and split view output. A few days (by the PI) to integrate Alive-exec into the Compiler Explorer site.

2.2. **Arm-tv.** By itself, Alive2 can only prove properties about LLVM IR. My research group created a fork of Alive2 that adds to it a formal semantics for AArch64 (64-bit ARM code) that is derived from ARM's "machine readable architecture," and then also adds a specification of the AArch64 ABI (application binary interface). The resulting tool, Arm-tv, then uses these semantics together with Alive2's existing LLVM semantics and refinement checking engine to check whether LLVM's AArch64 backend produces output that refines its input. Using this tool I have discovered and reported 40 miscompilation defects in this LLVM backend.

The Compiler Explorer software's core functionality is to illustrate the translation of a source language like C into assembly code. Thus, Arm-tv will fit naturally into this software.

Effort estimate: A few days (by the PI) to integrate Arm-tv into Compiler Explorer.

2.3. **Alivecc.** Traditionally, the C compiler on a Unix machine was invoked as "cc." We created Alivecc, a compiler driver that offers the same interface as LLVM's "Clang" front end, but then inserts Alive2 into the compilation pipeline, where it performs translation validation of every optimization pass in LLVM's middle end. Thus, Alivecc offers a convenient way to perform translation validation of an entire application. Since Alivecc is compatible with Clang, it will naturally fit into the Compiler Explorer software.

Effort estimate: A few days (by the PI) to integrate Alivecc into the Compiler Explorer site.

2.4. **Better Counterexamples from Alive-tv.** When Alive2 signals a failure of refinement, it provides a counterexample: a collection of concrete inputs and memory states that demonstrates the problem. The specific values in the counterexample are chosen by Z3 and are effectively arbitrary, which makes counterexamples more difficult to understand than they could be. To compensate for this problem, Alive2 contains some logic for simplifying counterexamples, by issuing repeated Z3 queries that, for example, attempt to minimize the number of memory blocks and values of integer inputs. I believe that there is much more

that could be done in this area. For example, because undefined behavior at the level of LLVM IR can be very difficult to understand [2], the counterexample should never contain a "poison" or "undef" value that is incidental—these should only be present when they are actually necessary to trigger a failure of refinement.

Any single counterexample can leave a user wondering *why* it is a counterexample—what is the property of that particular input that triggers a miscompilation? A straightforward solution would be to provide a collection of 3–5 counterexamples that the user can look at, which might suggest a pattern. For example, if every counterexample shows a negative number being input to the code, the user might reasonably guess that this is a necessary precondition for the miscompilation. Although it is trivial to get Alive2 to generate multiple counterexamples, in practice—due to incidental details of Z3's search strategy—they are usually all very similar. I believe that we can do better by inducing Z3 to provide counterexamples that are different from each other, for example by asking it bias numerical inputs towards different special values (INT_MIN, INT_MAX, 0, UINT_MAX, etc.). This effort will necessarily be heuristical—the interesting question is whether we can do something that human users find to be useful.

Effort estimate: Four months (by the PhD student) to implement and evaluate improved counterexample simplification.

2.5. **llvm-reduce Integration.** llvm-reduce is a command line tool bundled with LLVM that attempts to minimize the size of a module (a collection of functions and data) of LLVM IR, while preserving a user-specified property of interest. A typical property would be "crashes the optimizer" or "triggers a miscompilation bug." The interface to llvm-reduce is styled after C-Reduce—which I wrote, but which does not work well for LLVM IR.

I am a heavy user of llvm-reduce (and have contributed code to it!), and I believe that easy access to it will be a useful addition to the online Alive2 web site. The workflow for a compiler developer would then be:

(1) Identify a transformation that appears to be mis-optimizing some LLVM IR.
(2) Paste the IR into the Alive2 web site.
(3) Wait for Alive2 to confirm that the transformation is indeed incorrect.
(4) Click the "reduce" button.
(5) Wait for llvm-reduce to produce a reduced version of the original LLVM IR that still triggers the issue.

Effort estimate: A few days (by the PI) to integrate llvm-reduce into the Compiler Explorer site.

3. Infrastructure Upgrades

I currently pay—out of my own pocket—for the cloud machine that runs the online Alive2 instance. This (virtual) machine has two cores, 8 GB of RAM, and 160 GB of disk space. The consequences of these limited resources are that:

- Each translation validation request is limited to a 10 second timeout
- Each translation validation request is limited to 1 GB of memory
- Only the latest version of Alive2 and LLVM are available (each LLVM installation requires about 15 GB of disk). I refresh this version every 2–3 weeks.
- Building LLVM and Alive2 takes just a bit under eight hours, and the web site is not very responsive while this is happening.

I would like to relax all of these restrictions! Thus, in this proposal I am requesting about $4,000 with which to purchase a rack-mounted server machine with 32 GB of RAM, 8 cores, and a 240 GB solid-state disk. To this machine I will add a 4 TB hard disk that I have sitting around, not currently being used. It will reside in the Kahlert School of Computing's machine room, which has reliable network, power, and cooling. I will back this machine up using our existing backup facilities. Of particular importance is backing up the stateful part of Compiler Explorer's "shortlinks" that allow short URLs to provide access to large amounts of code. If we lose these, then all of the shortlinks in the LLVM project's Github issues will become useless.

Effort estimate: One month (by the PI) to:

- Setup the new machine, get it properly configured as a web server and backed up.
- Bring my fork of the Compiler Explorer software up to the latest version (after 4.5 years, it is almost 5,000 commits behind) and get it running on the new machine.

4. Evaluation Plan

The proposed work will be successful if it results in more formal methods tools being used by more members of the LLVM community. To evaluate it, I will track the following metrics:

(1) The availability of the Compiler Explorer web site.
(2) The number of times each Alive2-based tool, such as Alive-tv and Alive-exec, is used.
(3) The number of errors (timeouts, malformed inputs, etc.) encountered by each Alive2-based tool.
(4) The number of links from the LLVM project's issue tracker, into the Alive2 Compiler Explorer site.

I will also evaluate the success of the proposed work by engaging with the LLVM community. I have attended the LLVM Developers' Meeting almost every year for the past decade, and have good connections there (for example, I was a member of the LLVM Foundation's Board of Directors from 2016–2020). Specifically, I will:

(1) Organize a "birds of a feather" session about Alive2—and in particular its online version—at the LLVM Developers' Meeting in October 2025. My goal will be to solicit feedback and ideas.
(2) Give a technical talk about Alive2's online instance at the LLVM Developers' Meeting in October 2026, describing the progress made over the course of the proposed work.

Based on our work so far on Alive2, all indications are that the LLVM community is appreciative and supportive of our work. Even so, by directly talking to interested members of the community, I hope to learn about problems or requirements that the Alive2 team has not thought about.

5. Broader Impacts

5.1. **Improved Compiler Correctness.** LLVM is incredibly broadly used by the computer industry, and dozens of companies have compiler teams that contribute to it. For example, all of the systems-level software on Android and iOS phones is compiled using LLVM. We've used Alive2 to discover numerous miscompilation bugs in LLVM, and it has prevented an unknown number of additional bugs from being committed to the source tree in the first place. It is this second category of bugs—those that we prevent, rather than discover—that this proposal is concerned with.

5.2. **Changing industrial perception of formal methods.** We want to shift use of Alive2 out into the broader compiler community because:

- The software engineering literature has shown that bugs caught earlier are cheaper.
- We—the Alive2 team—do not scale, but the community does.
- We want to engender a shift in attitudes about formal methods. Rather than being an ivory tower technique, formal methods should simply be another tool in the working compiler engineer's toolbox.

It is this last point, changing industrial attitudes towards formal methods, that I believe might end up being the most significant broader impact due to the proposed work. Of course this requires providing industrial engineers with usable, high-quality tooling of the kind that our current academic system does not do a very good job of rewarding.

5.3. **Classroom Education.** I use Compiler Explorer heavily in my teaching. For example, in Spring 2025 I'm teaching an undergraduate Computer Systems course based on "Computer Systems: A Programmer's Perspective" by O'Hallaron and Bryant. In most of my lectures, I include multiple links into Compiler Explorer so that the students can not only see examples of code generation for themselves, but also play around with the examples on their own. My belief is that the interactivity is key to helping students build robust mental models of compilation and related phenomena.

When teaching compiler-related courses, whether undergraduate compilers or my grad-level Advanced Compilers class, I use the Alive2 compiler explorer site to illustrate refinement, as embodied in LLVM. I believe that other compiler class instructors would also benefit from this tool, and I proselytize it at every opportunity.

5.4. **Informal Education.** When a user first visits the Alive2 online web site, they see a greeting including this text on the left side of the browser window:

```
define i32 @src(i32) {
  %r = udiv i32 %0, 8192
  ret i32 %r
}

define i32 @tgt(i32) {
  %r = lshr i32 %0, 13
  ret i32 %r
}

; a good "getting started" exercise with alive2
; would be to change the udiv (unsigned divide)
; instruction in @src above into an sdiv (signed
; divide) and then fix @tgt so that the optimization
; is again correct
```

On the right side of the browser window, Alive2 indicates that the displayed transformation is indeed a refinement. When the user starts to change the code, Alive2 will continuously update its answer, reporting that their code either is or isn't a refinement (or that it is not syntactically valid LLVM IR).

Over the last few years I've watched many people try out this exercise. Usually, people can get the correct answer with a little coaching. (Hint: It's not enough to just change the logical right-shift to an arithmetic right shift. The code in `@tgt` also has to account for the fact that signed division in LLVM IR—as in C or C++—truncates towards zero.)

I propose creating a suite of similar exercises and making them publicly available. Some of these would be oriented towards illustrating specific concepts in LLVM IR, but most of them would simply illustrate the kinds of thinking that go into compiler middle-end optimizations. The great advantage of basing lessons of this kind on a formal methods tool is that it can perfectly differentiate between correct and incorrect answers, and furthermore it always provides a counterexample that concretely demonstrates why an incorrect answer is incorrect. For example, if a user naively changes the `lshr` instruction in the code above to an `ashr`, the counterexample provided by Alive2 will indicate that the optimized code is incorrect for a negative input.

Effort estimate: A week (by the PI) to create a suite of 20–30 exercises, spanning a range of difficulties.

6. Results from Previous NSF-Funded Research

My only current NSF award (currently in its last year, after a no-cost extension) is #1955688, "SHF: Medium: Formal Methods as a First-Class Citizen of a Mainstream Compiler Framework."

The motivation for this project was that as far as production compilers go, formal semantics are always an afterthought (including our own Alive2 work!). The proposed solution is to integrate formal semantics directly into compiler development, with the ultimate goal of deriving multiple formal-methods-based tools, and multiple parts of a compiler implementation, from those semantics. We split this project into three main thrusts; two of them, that we originally started working on in the MLIR ecosystem, got redirected into LLVM instead, due to the fact that MLIR was in worse technical shape than we had hoped, by the time the project started, and I did not want this fact to get in the way of PhD students graduating.

The first of these was Minotaur [3], a superoptimizer that we created in order to answer the question "Can we automatically synthesize optimizations that make effective use of SIMD instructions?" This project built upon many lessons learned from my earlier superoptimizer, Souper. Minotaur ended up being quite effective, and produced small but significant end-to-end speedups on the SPEC CPU 2017 benchmarks, when compared with `clang -O3`, which is a high bar for a new optimizer.

The second sub-project was Hydra [4], a project for taking a specific example of a local rewrite (which could come from a human looking at optimized code and noticing a missing optimization, or could come from a superoptimizer) and relaxing this rewrite into its most general form. The name "Hydra" (a monster that grows two heads when you chop one of them off) came from the fact that this problem ended up being much more difficult and multi-faceted than we had originally thought! The generalization process involves synthesizing a weakest precondition for the optimization and synthesizing functions to compute values of constants found in the optimized code that are not present in the unoptimized code. Perhaps the most interesting and difficult problem was synthesizing rewrites that were independent of the bitwidths of the inputs—there was very little related work here for us to draw on, and we had no direct support for bitwidth independence from Z3. Another interesting problem was figuring out how to make use of dataflow facts not as part of the precondition for an

optimization, but as inputs to the synthesis procedure that generates functions for computing constant values found in optimized code. A generalized optimization that is produced by Hydra is formally verified and can be automatically turned into C++ code suitable for inclusion in an optimizer. We founds that a collection of a few hundred generalized rules—after automated ranking and deduplication—was comparable in effectiveness with LLVM's enormous (and historically buggy) instruction combiner.

Both Hydra and Minotaur are fundamentally enabled by formal semantics! I believe that tools like this are a big part of the future of compilers. We simply need to learn how to derive more and more parts of a compiler from formal specifications, we can't keep implementing artifacts like LLVM in millions of lines of C++. The engineering cost, and the time to reach a mature, tuned, and tested code base are simply too large.

The third sub-project from this grant is the most ambitious; it is ongoing and while a paper summarizing it is in submission, it has not yet been published. In this work (which is in collaboration with Tobias Grosser's group from the University of Cambridge), we have created a family of MLIR dialects for encoding the semantics of MLIR dialects that encode programs. Our new dialects are at different levels of abstraction; for example, one of them supports encoding the semantics of memory operations, which are notoriously difficult to efficiently encode as SMT queries. The purpose of encoding semantics as dialects is that we can then write optimization passes over the dialects, leading to more efficient encodings. In contrast, Alive2 encodes all semantics and optimizations over these semantics monolithically, making it a very complex and difficult artifact to work with (problems inside Alive2 were one of the motivations for our MLIR work). Using our new dialects, we have encoded semantics for the MLIR "arith" and "comb" dialects and used these to derive translation validation tools that we have used to find bugs in MLIR optimization passes. In ongoing work, we are encoding the semantics of tensor-oriented MLIR dialects that are used for machine learning.

Finally, we have created an MLIR dialect that is intended for writing transfer functions for dataflow analyses. This dialect can be lowered to C++, in order to get executable dataflow analyses, and it can also be lowered alongside our semantics dialects in order to prove that the transfer functions are sound and (in some cases) maximally precise, with respect to the concrete semantics.

I'm very excited about this project and I believe that this effort represents the beginning of a new way to build compilers and compiler-adjacent tools. However, even in 2025, MLIR remains quite challenging to work with! It is plagued by problems where companies who invest in MLIR-based infrastructure often keep large chunks of their work internal and proprietary, instead of open-sourcing it.

References

[1] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for LLVM. In *PLDI*, 2021.

[2] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. Taming undefined behavior in LLVM. In *PLDI*, 2017.

[3] Zhengyang Liu, Stefan Mada, and John Regehr. Minotaur: A simd-oriented synthesizing superoptimizer. In *OOPSLA*, October 2024.

[4] Manasij Mukherjee and John Regehr. Hydra: Generalizing peephole optimizations with program synthesis. In *OOPSLA*, October 2024.