AliveInLean: A Verified LLVM Peephole Optimization Verifier

Juneyoung Lee[1], Chung-Kil Hur[1], and Nuno P. Lopes[2]

[1] Seoul National University
[2] Microsoft Research

Abstract. Ensuring that compiler optimizations are correct is important for the reliability of the entire software ecosystem, since all software is compiled. Alive [12] is a tool for verifying LLVM's peephole optimizations. Since Alive was released, it has helped compiler developers proactively find dozens of bugs in LLVM, avoiding potentially hazardous miscompilations. Despite having verified many LLVM optimizations so far, Alive is itself not verified, which has led to at least once declaring an optimization correct when it was not.
We introduce AliveInLean, a formally verified peephole optimization verifier for LLVM. As the name suggests, AliveInLean is a reengineered version of Alive developed in the Lean theorem prover [14]. Assuming that the proof obligations are correctly discharged by an SMT solver, AliveInLean gives the same level of correctness guarantees as state-of-the-art formal frameworks such as CompCert [11], Peek [15], and Vellvm [26], while inheriting the advantages of Alive (significantly more automation and easy adoption by compiler developers).

Keywords: Compiler Verification · Peephole Optimization · LLVM · Lean · Alive.

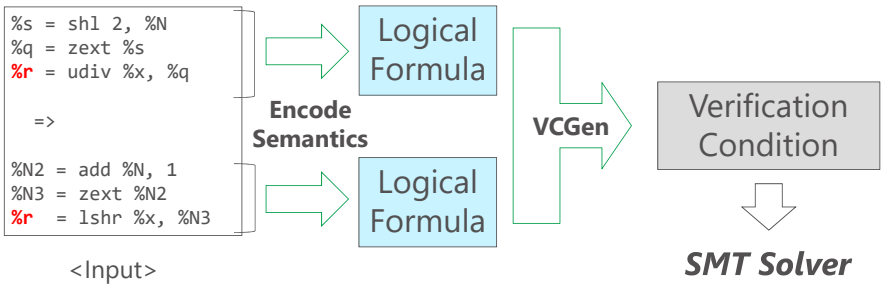
1 Introduction

Verifying compiler optimizations is important to ensure reliability of the software ecosystem. Various frameworks have been proposed to verify optimizations of industrial compilers. Among them, Alive [12] is a tool for verifying peephole optimizations of LLVM that has been successfully adopted by compiler developers. Since it was released, Alive has helped developers find dozens of bugs.

Fig. 1 shows the structure of Alive. An optimization pattern of interest written in a domain-specific language is given as input. Alive parses the input, and encodes the behavior of the source and target programs into logic formulas in the theory of quantified bit-vectors and arrays. Finally, several proof obligations are created from the encoded behavior, and then checked by an SMT solver.

Alive relies on the following three-fold trust base. Firstly, the semantics of LLVM's intermediate representation and SMT expressions. Secondly, Alive's verification condition generator. Finally, the SMT solver used to discharge proof obligations. None of these are formally verified, and thus an error in any of these may result in an incorrect answer.

Fig. 1. The structure of Alive and AliveInLean



To address this problem, we introduce AliveInLean, a formally verified peephole optimization verifier for LLVM. AliveInLean is written in Lean [14], an interactive theorem proving language. Its semantics of LLVM IR (Intermediate Representation) and SMT expressions are rigorously tested using Lean's metaprogramming language [5] and system library. AliveInLean's verification condition generator is formally verified in Lean.

Using AliveInLean requires less human effort than directly proving the optimizations on formal frameworks thanks to automation given by SMT solvers. For example, verifying the correctness of a peephole optimization on a formal framework requires more than a hundred lines of proofs [15]. However, the correctness of AliveInLean relies on the correctness of the used SMT solver. To counteract the dependency on SMT solvers, proof obligations can be cross-checked with multiple SMT solvers. Moreover, there is substantial work towards making SMT solvers generate proof certificates [2, 3, 6, 7].

AliveInLean is a proof of concept. It currently does not support all operations that Alive does like, e.g., memory-related operations. However, AliveInLean supports all integer peephole optimizations, which is already useful in practice as most bugs found by Alive were in integer optimizations [12].

2 Overview

We give an overview of AliveInLean's features from a user's perspective.

Verifying Optimizations. AliveInLean reads optimization(s) from a file and checks their correctness. A user writes an optimization of interest in a DSL with similar syntax to that of LLVM IR:

```
Name: AddSub:1309
%lhs = and i4 %a, %b
%rhs = or i4 %a, %b
%r = add i4 %lhs, %rhs
  =>
%r = add i4 %a, %b
```

This example transformation corresponds to rewriting `(%a & %b) + (%a | %b)` to `%a + %b`, given 4-bits integers `%a` and `%b`. The last variable `%r`, or *root* variable, is assumed to be the return value of the programs. AliveInLean encodes the behavior of each program and generates verification conditions (VCs). Finally, AliveInLean calls Z3 to discharge the VCs.

Proving Useful Properties. AliveInLean can be used as a formal framework to prove lemmas using interactive theorem proving. This is helpful when a user wants to show a property of a program which is hard to represent as a transformation.

For example, one may want to prove that the divisor of `udiv` (unsigned division) is never `poison`[3] if it did not raise undefined behavior (UB). The lemma below states this in Lean. This lemma says that the divisor `val` is never poison if the state `st'` after executing the `udiv` instruction (`step`) has no UB.

```
lemma never_poison:
  forall .. (HSTEP: some st' = step st (udiv isz name op1 op2))
            (HNOUB: not (has_ub st'))
            (HVAL:  some val = get_value st op2 (ty.int isz)),
    not (is_poison val)
```

Testing Specifications. AliveInLean supports random testing of AliveInLean's specifications (for which no verification is possible). For example, the `step` function in the above example implements a specification of the LLVM IR, and it can be tested with respect to the behavior of the LLVM compiler. Another trust-base is the specification of SMT expressions, which defines a relation between expressions (with no free variable) and their corresponding concrete values.

These tests help build confidence in the validity of VC generation. Running tests is helpful when a user wants to use a different version of LLVM or modify AliveInLean's specifications (e.g., adding a new instruction to IR).

3 Verifying Optimizations

In this section we introduce the different components of AliveInLean that work together to verify an optimization.

3.1 Semantics Encoder

Given a program and an initial state, the semantics encoder produces the final state of the program as a set of SMT expressions. The IR interpreter is similar, but works over concrete values rather than symbolic ones. The semantics encoder and the IR interpreter share the same codebase (essentially the LLVM IR

[3] `poison` is a special value of LLVM representing a result of an erroneous computation.

semantics). The code is parametric on the type of the program state. For example, the type of undefined behavior can be either initialized as the `bool` type of Lean or the `Bool` SMT expression type. Given the type, Lean can automatically resolve which operations to use to update the state using typeclass resolution.

3.2 Refinement Encoder

Given a source program, a transformed program, and an initial state, the refinement encoder emits an SMT expression that encodes the refinement check between the final states of the two programs. To obtain the final states, the semantics encoder is used.

The refinement check proves that (1) the transformed program only triggers UB when the original program does (i.e., UB can only be removed), (2) the root variable of the transformed program is only `poison` when it is also `poison` in the original program, and (3) variables' values in the final states of the two programs are the same when no UB is triggered and the original value is not `poison`.

3.3 Parser and Z3 backend

The parser for Alive's DSL is implemented using Lean's parser monad and file I/O library. SMT expressions are processed with Z3 using Lean's SMT interface.

4 Correctness of AliveInLean

We describe how the correctness of AliveInLean is proved. First, we explain the correctness proof of the semantics encoder and the refinement encoder. We show that if the SMT expression encoded by refinement encoder is valid, the optimization is indeed correct. Next, we explain how the trust-base is tested.

4.1 Semantics encoding

Given an IR interpreter `run`, a semantics encoder `encoder` is correct with respect to `run` if for any IR program and input state, the final program state generated by `run` and the symbolic state encoded by `encoder` are equivalent.

To formally define its correctness, an equivalence relation between SMT expressions and concrete values is defined. We say that an SMT expression e and a Lean value ν are equivalent, or $e \sim \nu$, if e has no free variables and it evaluates to ν. The equivalence relation is inductively defined with respect to the structure of an SMT expression. To deal with free variables, an environment η is defined, which is a set of pairs (x, ν) where x is a variable and ν is a concrete value. $\eta[\![e]\!]$ is an expression with all free variables x replaced with ν if $(x, \nu) \in \eta$.

Next, we define a program state. A state s is defined as (u, r) where u is an undefined behavior flag and r is a register file. r is a list of (x, v) where x is a variable and v is a value. v is defined as (sz, i, p) where sz is its size in bits, i is an integer value, and p is a poison flag.

There are two kinds of states: a symbolic state, and a concrete state. A symbolic state s_s is a state whose u, i, p are SMT expressions. A concrete state s_c is a state whose all attributes are concrete values. We say that s_s and s_c are equivalent, or $s_s \sim s_c$, if s_s has no free variable in its attributes and they are equivalent. $\eta[\![s_s]\!]$ is a symbolic state with the environment η applied to u, i, p.

Now, the correctness of `encoder` with respect to `run` is defined as follows. It states that the result of `encoder` is equivalent to the result of `run`.

Theorem 1. *For all initial states s_s, s_c, program p, and environment η s.t. $\eta[\![s_s]\!] \sim s_c$, we have that $\eta[\![\texttt{encoder}(p, s_s)]\!] \sim \texttt{run}(p, s_c)$.*

4.2 Refinement encoding

Function $\texttt{check}(p_{src}, p_{tgt}, s_s)$ generates an SMT expression that encodes refinement between the source and target programs, respectively, p_{src} and p_{tgt}.

We first define refinement between two concrete states. As Alive does, AliveInLean only checks the value of the root variable of a program. Given a root variable r, a concrete state s'_c refines s_c, or $s'_c \sqsubseteq s_c$, if (1) s_c has undefined behavior, or (2) both s_c and s'_c have values assigned to r, say v and v', and $v = \texttt{poison} \vee v' = v$. A target program p_{tgt} refines program p_{src} if $\texttt{run}(p_{tgt}, s_c) \sqsubseteq \texttt{run}(p_{src}, s_c)$ holds for any initial concrete state s_c,.

The correctness of `check` is stated as follows.

Theorem 2. *Given an initial symbolic state s_s, if $\eta_0[\![\texttt{check}(p_{src}, p_{tgt}, s_s)]\!] \sim$ true for any η_0, then for any environment η and initial state s_c s.t. $\eta[\![s_s]\!] \sim s_c$, we have that $\texttt{run}(p_{tgt}, s_c) \sqsubseteq \texttt{run}(p_{src}, s_c)$.*

This theorem says that if the returned expression of `check` evaluates to true in any environment, program p_{tgt} refines program p_{src}.

4.3 Validity of Trust-base

Testing specification of SMT expressions. Specifications of SMT expressions are traversed using Lean's metaprogramming language and tested. The testing we have done is different from QuickChick [4] because QuickChick evaluates expressions in Coq. The approach cannot be used here because SMT expressions need to be evaluated in an SMT solver (e.g., Z3). Example spec:

```
forall {sz : size} (s1 s2 : sbitvec sz) (b1 b2 : bitvector sz),
  bv_equiv s1 b1 -> bv_equiv s2 b2 ->
    bv_equiv (sbitvec.add s1 s2) (bitvector.add b1 b2)
```

This spec says that if SMT expressions `s1`, `s2` of a bit-vector type (`sbitvec`) are equivalent to two concrete bit-vector values `b1`, `b2` in Lean (`bitvector`), an add expression of `s1`, `s2` is equivalent to the result of adding `b1` and `b2`. Function `bitvector.add` must be called in Lean, so its operands (`b1`, `b2`) are assigned random values in Lean. `sbitvec.add` is translated to SMT's `bvadd` expression, and `s1` and `s2` are initialized as `BitVec` variables in an SMT solver. The testing function generates an SMT expression with random inputs like the following:

```
(assert (forall ((s1 (_ BitVec 4))) (forall ((s2 (_ BitVec 4)))
  (=> (= s1 #xA) (=> (= s2 #x2) (= (bvadd s1 s2) #xC))))))
```

The size of bitvector (`sz`) is initialized to 4, and `b1`, `b2` were randomly initialized to 10 (`#xA`) and 2 (`#x2`). A specification is incorrect if the generated SMT expression is not valid.

Testing specification of LLVM IR. Specification of LLVM IR is tested using randomly generated IR programs. IR programs of 5∼10 randomly chosen instructions are generated, compiled with LLVM, and ran. The result of the execution of the program is compared with the result of AliveInLean's IR interpreter.

5 Evaluation

For the evaluation, we used a computer with an Intel Core i5-6600 CPU and 8 GB of RAM, and Z3 [13] for SMT solving. To test whether AliveInLean and Alive give the same result, we used all of the 150 integer optimizations from Alive's test suite that are supported by AliveInLean. No mismatches were observed.

To test the SMT specification, we randomly generated 10,000 tests for each of the operations (18 bit-vector and 15 boolean). This test took 3 CPU hours.

The LLVM IR specification was tested by running 1,000,000 random IR programs in our interpreter and comparing the output with that of LLVM. This comparison needs to take into account that some programs may trigger UB or yield a poison value, which gives freedom to LLVM to produce a variety of results. These tests took 10 CPU hours overall. Four admitted arithmetic lemmas were tested as well. As a side-effect of the testing, we found several miscompilation bugs in LLVM.[4]

AliveInLean[5] consists of 11.9K lines of code. The optimization verifier consists of 2.2K LoC, the specification tester is 1.5K, and the proof has 8.1K lines. It took 3 person-months to implement the tool and prove its correctness.

6 Related Work

We introduce previous work on compiler verification and validation and compare it with AliveInLean. Also, we give an overview on previous work on semantics of compiler intermediate representations (IRs).

6.1 Compiler Verification

Proving correctness on formal semantics. The correctness of compilation can be proved on a formal semantics of a language that is written in a theorem proving language such as Coq. Vellvm [26] is a Coq formalization of the semantics

[4] https://llvm.org/PR40657
[5] https://github.com/Microsoft/AliveInLean

of LLVM IR. CompCert [11] is a verified C compiler written in Coq, and its compilation to assembly languages including x86, PowerPC is proved correct.

However, it is hard to apply this approach to existing industrial compilers because proving correctness of optimizations requires non-trivial effort. Peek [15] is a framework for implementing and verifying peephole optimizations for x86 on CompCert. They implemented 28 peephole optimizations which required 3.3k lines of code and 6.6k lines of proofs (∼350 LoC each). Even if this is small compared to the size of CompCert, the burden is non-trivial considering that LLVM has more than 1,000 peephole optimizations [12].

Another problem with this approach is that changing the semantics requires modification of the proof. The semantics of `poison` and `undef` value of LLVM is currently not consistent and thus it triggers miscompilations of some programs [10]. Therefore, compiler developers regularly test various `undef` semantics with existing optimizations, which would be a non-trivial task if correctness proofs had to be manually updated.

Translation validation and credible compilation. In translation validation [18], a pair of an original program and an optimized program is given to a validation tool at compile time to check the correctness of the optimization. Several such tools exist for LLVM [20,22,25]. Translation validation is, however, slow, and it cannot tell whether an optimization is correct in general. Consider this optimization:

```
z = 0 - (x / C)
  =>
z = x / -C
```

If `C` is a constant, `-C` can be computed at compile time. However, this optimization is wrong only if `C` is `INT_MIN`. To show that compilation is fully correct, translation validation would need to be run for every combination of inputs.

Credible compilation [19], or witnessing compiler [16,17], is an approach to improve translation validation by accepting witnesses generated by a compiler. Crellvm [8] is a credible compilation framework for LLVM. It requires modifications to the compiler, which makes it harder to apply and maintain.

6.2 Solver-Aided Programming Languages

Proving correctness of optimizations can be represented as a search problem that finds a counter-example for the optimization. Tools like Z3, CVC4 can be used to solve the search problem. Translation of a high-level search problem to the external solver's input has been considered bug-prone, and frameworks like Rosette [21] and Smten [23] address this issue by providing higher-level languages for describing the search problem. SpaceSearch [24] helps programmers prove the correctness of the description by supporting Coq and Rosette backends from a single specification. AliveInLean provides a stronger guarantee of correctness because translation to SMT expressions is also written in Lean, leaving Lean as the sole trust-base.

6.3 Semantics of Compiler IR

Correctly encoding semantics of compiler IR is important for the validity of a tool. LLVM IR is an SSA-based intermediate representation which is used to represent a program being compiled. LLVM LangRef [1] has an informal definition of the LLVM IR, but there are a few known problems. [10] shows that the semantics of `poison` and `undef` values are inconsistent. [9] shows that the semantics of pointer↔integer casting is inconsistent. AliveInLean supports `poison` but not `undef`, following the suggestion from [10]. AliveInLean does not support memory-related operations including load, store, and pointer ↔ integer casting.

7 Discussion

AliveInLean has several limitations. As discussed before, AliveInLean does not support memory operations. Correctly encoding the memory model of LLVM IR is challenging because the memory model of LLVM IR is more complex than either a byte array or a set of memory objects [9]. Supporting branch instructions and floating point would help developers prove interesting optimizations. Supporting branches is a challenging job especially when loops are involved.

Maintainability of AliveInLean highly relies on one's proficiency in Lean. Changing the semantics of an IR instruction breaks the proof, and updating it requires proficiency in Lean. However, we believe that only relevant parts in the proof need to be updated as the proof is modularized.

Alive has features that are absent in AliveInLean. Alive supports defining a precondition for an optimization, inferring types of variables if not given, and showing counter-examples if the optimization is wrong. We leave this as future work.

8 Conclusion

AliveInLean is a formally verified compiler optimization verifier. Its verification condition generator is formally verified with a machine-checked proof. Using AliveInLean, developers can easily check the correctness of compiler optimizations with high reliability. Also, they can use AliveInLean as a formal framework like Vellvm to prove properties of interest in limited cases. The extensive random testing did not find problems in the trust base, increasing its trustworthiness. Moreover, as a side-effect of the IR semantics testing, we found several bugs in LLVM.

Acknowledgments. The authors thank Leonardo de Moura and Sebastian Ullrich for their help with Lean. This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (2017R1A2B2007512). The first author was supported by a Korea Foundation for Advanced Studies scholarship.

References

1. LLVM language reference manual, https://llvm.org/docs/LangRef.html
2. Barbosa, H., Blanchette, J.C., Fontaine, P.: Scalable fine-grained proofs for formula processing. In: Automated Deduction – CADE 26. pp. 398–412 (2017)
3. Böhme, S., Fox, A.C.J., Sewell, T., Weber, T.: Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In: Certified Programs and Proofs. pp. 183–198 (2011)
4. Dénès, M., Hriţcu, C., Lampropoulos, L., Paraskevopoulou, Z., Pierce, B.C.: Quickchick : Property-based testing for Coq (2014)
5. Ebner, G., Ullrich, S., Roesch, J., Avigad, J., de Moura, L.: A metaprogramming framework for formal verification. Proc. ACM Program. Lang. **1**(ICFP), 34:1–34:29 (Aug 2017). https://doi.org/10.1145/3110278
6. Ekici, B., Mebsout, A., Tinelli, C., Keller, C., Katz, G., Reynolds, A., Barrett, C.: Smtcoq: A plug-in for integrating SMT solvers into Coq. In: Computer Aided Verification. pp. 126–133 (2017)
7. Hadarean, L., Barrett, C., Reynolds, A., Tinelli, C., Deters, M.: Fine grained SMT proofs for the theory of fixed-width bit-vectors. In: Logic for Programming, Artificial Intelligence, and Reasoning. pp. 340–355 (2015)
8. Kang, J., Kim, Y., Song, Y., Lee, J., Park, S., Shin, M.D., Kim, Y., Cho, S., Choi, J., Hur, C.K., Yi, K.: Crellvm: Verified credible compilation for LLVM. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 631–645. ACM (2018). https://doi.org/10.1145/3192366.3192377
9. Lee, J., Hur, C.K., Jung, R., Liu, Z., Regehr, J., Lopes, N.P.: Reconciling high-level optimizations and low-level code in LLVM. Proc. ACM Program. Lang. **2**(OOPSLA), 125:1–125:28 (Oct 2018). https://doi.org/10.1145/3276495
10. Lee, J., Kim, Y., Song, Y., Hur, C.K., Das, S., Majnemer, D., Regehr, J., Lopes, N.P.: Taming undefined behavior in LLVM. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 633–647. ACM (2017). https://doi.org/10.1145/3062341.3062343
11. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (Jul 2009). https://doi.org/10.1145/1538788.1538814
12. Lopes, N.P., Menendez, D., Nagarakatte, S., Regehr, J.: Provably correct peephole optimizations with Alive. In: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 22–32. ACM (2015). https://doi.org/10.1145/2737924.2737965
13. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems (2008)
14. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean theorem prover (system description). In: Automated Deduction - CADE-25. pp. 378–388. Springer International Publishing (2015)
15. Mullen, E., Zuniga, D., Tatlock, Z., Grossman, D.: Verified peephole optimizations for CompCert. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 448–461. ACM (2016). https://doi.org/10.1145/2908080.2908109
16. Namjoshi, K.S., Tagliabue, G., Zuck, L.D.: A witnessing compiler: A proof of concept. In: Runtime Verification. pp. 340–345. Springer Berlin Heidelberg (2013)
17. Namjoshi, K.S., Zuck, L.D.: Witnessing program transformations. In: Static Analysis. pp. 304–323. Springer Berlin Heidelberg (2013)

18. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (1998)
19. Rinard, M.C., Marinov, D.: Credible compilation with pointers. In: Proceedings of the Workshop on Run-Time Result Verification (1999)
20. Stepp, M., Tate, R., Lerner, S.: Equality-based translation validator for LLVM. In: Proceedings of the 23rd International Conference on Computer Aided Verification. pp. 737–742. Springer-Verlag (2011)
21. Torlak, E., Bodik, R.: Growing solver-aided languages with Rosette. In: Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. pp. 135–152. ACM (2013). https://doi.org/10.1145/2509578.2509586
22. Tristan, J.B., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for LLVM. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 295–305. ACM (2011). https://doi.org/10.1145/1993498.1993533
23. Uhler, R., Dave, N.: Smten: Automatic translation of high-level symbolic computations into SMT queries. In: Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044. pp. 678–683. Springer-Verlag New York, Inc. (2013). https://doi.org/10.1007/978-3-642-39799-8_45
24. Weitz, K., Lyubomirsky, S., Heule, S., Torlak, E., Ernst, M.D., Tatlock, Z.: Spacesearch: A library for building and verifying solver-aided tools. Proc. ACM Program. Lang. **1**(ICFP), 25:1–25:28 (Aug 2017). https://doi.org/10.1145/3110269
25. Zaks, A., Pnueli, A.: CoVaC: Compiler validation by program analysis of the cross-product. In: FM 2008: Formal Methods (2008)
26. Zhao, J., Nagarakatte, S., Martin, M.M., Zdancewic, S.: Formalizing the LLVM intermediate representation for verified program transformations. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 427–440. ACM (2012). https://doi.org/10.1145/2103656.2103709