```
def p1 : PeepholeRewrite Op [.int] .int := {
      lhs := lhs,
      rhs := rhs,
      correct := by
        rw [lhs, rhs]; funext Γv; simp_peephole [add, cst] at Γv
        /- ⊢ ∀ (a : BitVec 32), a + BitVec.ofInt 32 0 = a -/
        intros a; simp_alive /- goals accomplished 🎉 -/
}
```

```
/-- x + 0 -/
def lhs : Com Op [.int] .int := [mlir_icom| {
 ^bb0(%x: int):
   %0 = const 0 : () -> int
   %1 = add (%x, %0) : (int, int) -> int
  return (%1) : (int) -> ()
}]
```

```
/-- x -/
def rhs : Com Op [.int] .int := [mlir_icom| {
 ^bb0(%x: int):
   %0 = const 0 : () -> int
   %1 = add (%x, %0) : (int, int) -> int
  return (%x) : (int) -> ()
}]
```

lean-mlir: Formally Verifying Peephole Optimizations for MLIR

Siddharth Bhat, *Alex Keizer*, Chris Hughes, Andres Goens, Tobias Grosser

University of Edinburgh · University of Amsterdam · University of Cambridge

Wait, What About Alive?

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

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

Wait, What About Alive?

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

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

Wait, What About Alive?

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

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

Wait, What About Alive?

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

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

```
Transformation seems to be correct!
```

Wait, What About Alive?

```
define i32 @src(i32) {
  %r = udiv i32 %0, 1
  ret i32 %r
}

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

Alive is Awesome!

```
define i32 @src(i32) {
  %r = udiv i32 %0, 1
  ret i32 %r
}

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

```
Transformation doesn't verify!

ERROR: Value mismatch

Example:
i32 %#0 = #x00000001 (1)

Source:
i32 %r = #x00000001 (1)

Target:
i32 %r = #x00000000 (0)
Source value: #x00000001 (1)
Target value: #x00000000 (0)
```

Alive is Awesome!

```
define i32 @src(i32) {
  %r = udiv i32 %0, 1
  ret i32 %r
}

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

```
Transformation doesn't verify!

ERROR: Value mismatch

Example:
i32 %#0 = #x00000001 (1)

Source:
i32 %r = #x00000001 (1)

Target:
i32 %r = #x00000000 (0)
Source value: #x00000001 (1)
Target value: #x00000000 (0)
```

Alive is Awesome!

```
define i32 @src(i32) {
  %r = udiv i32 %0, 1
  ret i32 %r
}

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

```
Transformation doesn't verify!

ERROR: Value mismatch

Example:
i32 %#0 = #x00000001 (1)

Source:
i32 %r = #x00000001 (1)

Target:
i32 %r = #x00000000 (0)
Source value: #x00000001 (1)
Target value: #x00000000 (0)
```

Alive is Awesome!

[Inst Fold fma
only (#10010
xc
Fixes
Alive
1 par

dtcxzyw
Alive2 proo
`smt-to`):
https://ali
main (
1 parent eb

[InstCombine] fo
|| (a != c &&
== (b != c) (#
zjaffal authored on
resolves #92966
alive proof
https://alive2.llvm.c
main (#94915)
1 parent 3ae6755 comm

[InstCombine] Extend Fold of Zero-extended Bit Test (#102100)

mskamp authored on Aug 21 · ✕ 51 / 56 · Verified

```
Previously, (zext (icmp ne (and X, (1 << ShAmt)), 0))
has only been
folded if the bit width of X and the result were equal.
Use a trunc or
zext instruction to also support other bit widths.

This is a follow-up to commit 533190a,
which introduced a regression: (zext (icmp ne (and (lshr
X ShAmt) 1) 0))
is not folded any longer to (zext/trunc (and (lshr X
ShAmt) 1)) since
the commit introduced the fold of (icmp ne (and (lshr X
ShAmt) 1) 0) to
(icmp ne (and X (1 << ShAmt)) 0). The change introduced
by this commit
restores this fold.

Alive proof: https://alive2.llvm.org/ce/z/MFkNXs

Relates to issue #86813 and pull request #101838.
```

main (#102100)

1 parent 4f07508 commit 170a21e

Alive is Awesome! What about MLIR?

'polynomial' Dialect

The Polynomial dialect defines single-variable polynomial types and operations.

The simplest use of `polynomial` is to represent mathematical operations in a polynomial ring `R[x]`, where `R` is another MLIR type like `i32`.

More generally, this dialect supports representing polynomial operations in a quotient ring `R[X]/(f(x))` for some statically fixed polynomial `f(x)`. Two polyomials `p(x), q(x)` are considered equal in this ring if they have the same remainder when dividing by `f(x)`. When a modulus is given, ring operations are performed with reductions modulo `f(x)` and relative to the coefficient ring `R`.

Alive is Awesome! How does it work?

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

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

Alive is Awesome! How does it work?

```
define i32 @src(i32) {
  %r = udiv i32 %0, 1
  ret i32 %r
}

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

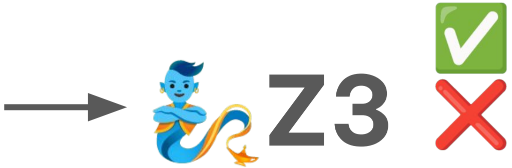
```
(set-logic QF_UFBV)

(define-fun src
  ((x (_ BitVec 32)))
  (_ BitVec 32)
  (bvudiv x (_ bv32 1)))

(define-fun tgt
  ((x (_ BitVec 32)))
  (_ BitVec 32)
  (bvlshr x (_ bv32 32)))
```

Alive is Awesome! How does it work?

```
define i32 @src(i32) {
  %r = udiv i32 %0, 1
  ret i32 %r
}

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

```
(set-logic QF_UFBV)

(define-fun src
  ((x (_ BitVec 32)))
  (_ BitVec 32)
  (bvudiv x (_ bv32 1)))

(define-fun tgt
  ((x (_ BitVec 32)))
  (_ BitVec 32)
  (bvlshr x (_ bv32 32)))
```

"does src equal tgt for all inputs?"



Z3

```
Transformation doesn't verify!

ERROR: Value mismatch

Example:
i32 %#0 = #x00000001 (1)

Source:
i32 %r = #x00000001 (1)

Target:
i32 %r = #x00000000 (0)
Source value: #x00000001 (1)
Target value: #x00000000 (0)
```

Alive is Awesome! How does it work?

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

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

```
(set-logic QF_UFBV)

(define-fun src
  ((x (_ BitVec 32)))
  (_ BitVec 32)
  (bvudiv x (_ bv32 8192)))
```

LLVM $\xrightarrow{\text{Alive}}$ **SMT-LIB** $\longrightarrow$ 🧞 **Z3** ✅ ❌

"does src equal tgt for all inputs?"

Provably Correct Peephole Optimizations with Alive

Nuno P. Lopes
Microsoft Research, UK
nlopes@microsoft.com

David Menendez Santosh Nagarakatte
Rutgers University, USA
{davemm,santosh.nagarakatte}@cs.rutgers.edu

John Regehr
University of Utah, U
regehr@cs.utah.ed

Abstract

Compilers should not miscompile. Our work addresses problems in developing peephole optimizations that perform local rewriting to improve the efficiency of LLVM code. These optimizations are individually difficult to get right, particularly in the presence of undefined behavior; taken together they represent a persistent source of bugs. This paper presents Alive, a domain-specific language for writing optimizations and for automatically either proving them correct or else generating counterexamples. Furthermore, Alive can be automatically translated into C++ code that is suitable for inclusion in an LLVM optimization pass. Alive is based on an attempt to balance usability and formal methods; for example, it captures—but largely hides—the detailed semantics of three different kinds of undefined behavior in LLVM. We have translated more than 300 LLVM optimizations into Alive and, in the process, found that eight of them were wrong.

Categories and Subject Descriptors D.2.4 [Programming Languages]: Software/Program Verification; D.3.4 [Programming

(compiler verification) or a proof that a particular com correct (translation validation). For example, CompC a hybrid of the two approaches. Unfortunately, creatin required several person-years of proof engineering and tool does not provide a good value proposition for man use cases: it implements a subset of C, optimizes only does not yet support x86-64 or the increasingly impo extensions to x86 and ARM. In contrast, production are constantly improved to support new language sta to obtain the best possible performance on emerging ar

This paper presents Alive: a new language and to oping correct LLVM optimizations. Alive aims for a that is both practical and formal; it allows compiler wri ify peephole optimizations for LLVM's intermediate re (IR), it automatically proves them correct with the hel bility modulo theory (SMT) solvers (or provides a coun and it automatically generates C++ code that is simi written peephole optimizations such as those found in struction combiner (InstCombine) pass. InstCombine

Alive2: Bounded Translation Validation for LLVM

Nuno P. Lopes
nlopes@microsoft.com
Microsoft Research
UK

Juneyoung Lee
juneyoung.lee@sf.snu.ac.k
Seoul National University
South Korea

Chung-Kil Hur
gil.hur@sf.snu.ac.k
Seoul National University
South Korea

Zhengyang Liu
liuz@cs.utah.edu
University of Utah
USA

John Regehr
regehr@cs.utah.edu
University of Utah
USA

Abstract

We designed, implemented, and deployed Alive2: a *bounded* translation validation tool for the LLVM compiler's intermediate representation (IR). It limits resource consumption by, for example, unrolling loops up to some bound, which means there are circumstances in which it misses bugs. Alive2 is designed to avoid false alarms, is fully automatic through the use of an SMT solver, and requires no changes to LLVM. By running Alive2 over LLVM's unit test suite, we discovered and reported 47 new bugs, 28 of which have been fixed already. Moreover, our work has led to eight patches to the LLVM Language Reference—the definitive description of the semantics of its IR—and we have participated in numerous discussions with the goal of clarifying ambiguities and fixing errors in these semantics. Alive2 is open source and we also made it available on the web, where it has active users from the LLVM community.

1 Introduction

LLVM is a popular open-source compiler that is used by numerous frontends (e.g., C, C++, Fortran, Rust, Swift), and that generates high-quality code for a variety of target architectures. We want LLVM to be correct but, like any large code base, it contains bugs. Proving functional correctness of about 2.6 million lines of C++ is still impractical, but a weaker formal technique—translation validation—can be used to certify that individual executions of the compiler respected its specification.

A key feature of LLVM that makes it a suitable platform for translation validation is its intermediate representation (IR), which provides a common point of interaction between frontends, backends, and middle-end transformation passes. LLVM IR has a specification document,[1] making it more amenable to formal methods than are most other compiler IRs. Even so, there have been numerous instances of ambiguity in the specification, and there have also been (and still

Alive is Awesome! What about MLIR?

'polynomial' Dialect

The Polynomial dialect defines single-variable polynomial types and operations.

The simplest use of `polynomial` is to represent mathematical operations in a polynomial ring `R[x]`, where `R` is another MLIR type like `i32`.

More generally, this dialect supports representing polynomial operations in a quotient ring `R[X]/(f(x))` for some statically fixed polynomial `f(x)`. Two polyomials `p(x), q(x)` are considered equal in this ring if they have the same remainder when dividing by `f(x)`. When a modulus is given, ring operations are performed with reductions modulo `f(x)` and relative to the coefficient ring `R`.

LLVM $\xrightarrow{\text{Alive}}$ **SMT-LIB** $\longrightarrow$ Z3 ✅ ❌

Alive is Awesome! What about MLIR?

'polynomial' Dialect

The Polynomial dialect defines single-variable polynomial types and operations.

The simplest use of `polynomial` is to represent mathematical operations in a polynomial ring `R[x]`, where `R` is another MLIR type like `i32`.

More generally, this dialect supports representing polynomial operations in a quotient ring `R[X]/(f(x))` for some statically fixed polynomial `f(x)`. Two polyomials `p(x), q(x)` are considered equal in this ring if they have the same remainder when dividing by `f(x)`. When a modulus is given, ring operations are performed with reductions modulo `f(x)` and relative to the coefficient ring `R`.

MLIR $\xrightarrow{\text{Alive}}$ SMT-LIB $\longrightarrow$ Z3 ✅ ❌

Alive is Awesome! What about MLIR?

'polynomial' Dialect

The Polynomial dialect defines single-variable polynomial types and operations.

The simplest use of `polynomial` is to represent mathematical operations in a polynomial ring `R[x]`, where `R` is another MLIR type like `i32`.

More generally, this dialect supports representing polynomial operations in a quotient ring `R[X]/(f(x))` for some statically fixed polynomial `f(x)`. Two polyomials `p(x), q(x)` are considered equal in this ring if they have the same remainder when dividing by `f(x)`. When a modulus is given, ring operations are performed with reductions modulo `f(x)` and relative to the coefficient ring `R`.

MLIR $\xrightarrow{\text{Alive}}$ SMT-LIB $\longrightarrow$ Z3 ✅ ❌

Alive is Awesome! What about MLIR?

'polynomial' Dialect

The Polynomial dialect defines single-variable polynomial types and operations.

The simplest use of `polynomial` is to represent mathematical operations in a polynomial ring `R[x]`, where `R` is another MLIR type like `i32`.

More generally, this dialect supports representing polynomial operations in a quotient ring `R[X]/(f(x))` for some statically fixed polynomial `f(x)`. Two polyomials `p(x), q(x)` are considered equal in this ring if they have the same remainder when dividing by `f(x)`. When a modulus is given, ring operations are performed with reductions modulo `f(x)` and relative to the coefficient ring `R`.

Just ask Nuno, June, John, ... to write Alive-MLIR?

MLIR ···Alive···▶ SMT-LIB ⟶ Z3 ✅ ❌


Z3
Problem
Outside
SMT-LIB

Alive is Awesome! What about MLIR?

'polynomial' Dialect

The Polynomial dialect defines single-variable polynomial types and operations.

The simplest use of `polynomial` is to represent mathematical operations in a polynomial ring `R[x]`, where `R` is another MLIR type like `i32`.

More generally, this dialect supports representing polynomial operations in a quotient ring `R[X]/(f(x))` for some statically fixed polynomial `f(x)`. Two polyomials `p(x), q(x)` are considered equal in this ring if they have the same remainder when dividing by `f(x)`. When a modulus is given, ring operations are performed with reductions modulo `f(x)` and relative to the coefficient ring `R`.

Need very clever encodings of concepts into SMT-LIB :(

Alive is Awesome! What about MLIR?

'polynomia

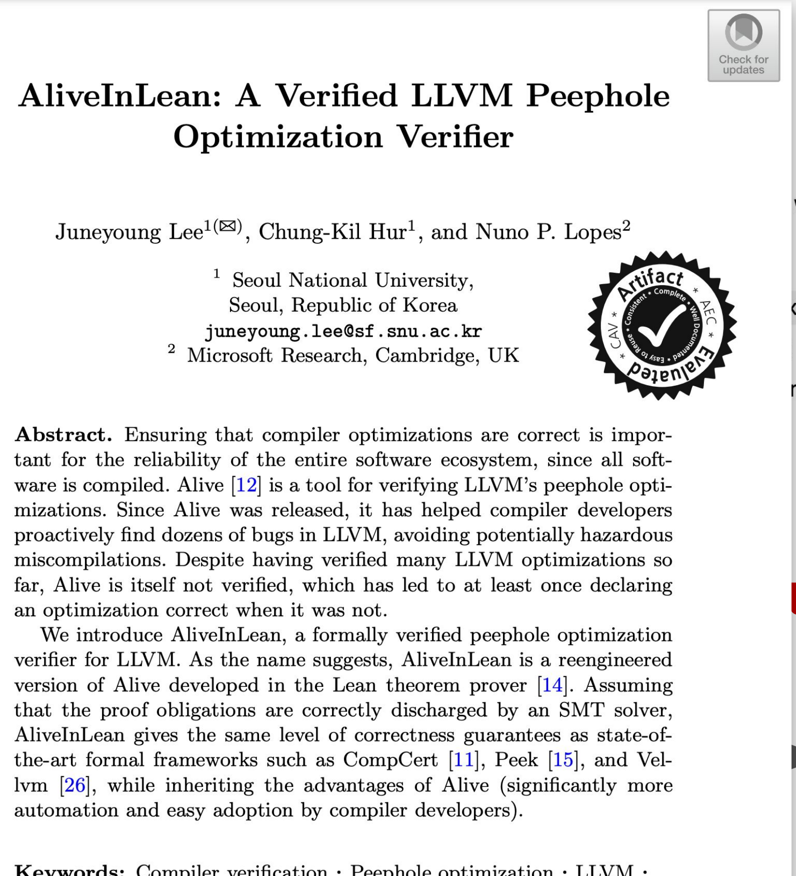
The Polynomial dialect c

The simplest use of `po` ... where `R`
is another MLIR type like

More generally, this dial ... `))` for
some statically fixed pol ... if they
have the same remainde ... rmed
with reductions modulo

Need very ... **to SMT-LIB :(**

MLIR ·· ▶ Z3 ✅ ❌


Check for updates

AliveInLean: A Verified LLVM Peephole Optimization Verifier

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

[1] Seoul National University, Seoul, Republic of Korea
juneyoung.lee@sf.snu.ac.kr
[2] Microsoft Research, Cambridge, UK

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 ·

Alive is Awesome! What about MLIR?

'polynomia

The Polynomial dialect

The simplest use of `po` ... where `R`

is another MLIR type like

More generally, this dial ... `))` for

some statically fixed pol ... if they

have the same remainde ... rmed

with reductions modulo

Check for updates

AliveInLean: A Verified LLVM Peephole Optimization Verifier

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

[1] Seoul National University, Seoul, Republic of Korea
juneyoung.lee@sf.snu.ac.kr

[2] Microsoft Research, Cambridge, UK

Artifact Evaluated · CAV · AEC · Consistent · Complete · Well Documented · Easy to Reuse

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 ·

Need very ... **to SMT-LIB :(**

MLIR ··· ▶ Z3 ✅ ❌

Alive is Awesome! What about MLIR?

'polynomia

The Polynomial dialect c

The simplest use of `po`
is another MLIR type like

More generally, this dial
some statically fixed pol
have the same remainde
with reductions modulo

Need very

MLIR ··

AliveInLean: A Verified LL
Optimization Ver

Juneyoung Lee[1(✉)], Chung-Kil Hur[1], an

[1] Seoul National Universi
Seoul, Republic of Korea
juneyoung.lee@sf.snu.ac
[2] Microsoft Research, Cambrid

Abstract. Ensuring that compiler optimizatio
tant for the reliability of the entire software e
ware is compiled. Alive [12] is a tool for verifyin
mizations. Since Alive was released, it has hel
proactively find dozens of bugs in LLVM, avoidi
miscompilations. Despite having verified many
far, Alive is itself not verified, which has led to
an optimization correct when it was not.

We introduce AliveInLean, a formally verifie
verifier for LLVM. As the name suggests, Alive
version of Alive developed in the Lean theoren
that the proof obligations are correctly discha
AliveInLean gives the same level of correctness
the-art formal frameworks such as CompCert |
lvm [26], while inheriting the advantages of A
automation and easy adoption by compiler deve

Keywords: Compiler verification · Peephole o

Verifying Peephole Rewriting In SSA Compiler IRs

Siddharth Bhat
Cambridge University, United Kingdom

Alex Keizer
Cambridge University, United Kingdom

Chris Hughes
University of Edinburgh, United Kingdom

Andrés Goens
University of Amsterdam, Netherlands

Tobias Grosser
Cambridge University, United Kingdom

Abstract

There is an increasing need for domain-specific reasoning in modern compilers. This has fueled the use of tailored intermediate representations (IRs) based on static single assignment (SSA), like in the MLIR compiler framework. Interactive theorem provers (ITPs) provide strong guarantees for the end-to-end verification of compilers (e.g., CompCert). However, modern compilers and their IRs evolve at a rate that makes proof engineering alongside them prohibitively expensive. Nevertheless, well-scoped push-button automated verification tools such as the Alive peephole verifier for LLVM-IR gained recognition in domains where SMT solvers offer efficient (semi) decision procedures. In this paper, we aim to combine the convenience of automation with the versatility of ITPs for verifying peephole rewrites across domain-specific IRs. We formalize a core calculus for SSA-based IRs that is generic over the IR and covers so-called regions (nested scoping used by many domain-specific IRs in the MLIR ecosystem). Our mechanization in the Lean proof assistant provides a user-friendly frontend for translating MLIR syntax into our calculus. We provide scaffolding for defining and verifying peephole rewrites, offering tactics to eliminate the abstraction overhead of our SSA calculus. We prove correctness theorems about peephole rewriting, as well as two classical program transformations. To evaluate our framework, we consider three use cases from the MLIR ecosystem that cover different levels of abstractions: (1) bitvector rewrites from LLVM, (2) structured control flow, and (3) fully homomorphic encryption. We envision that our mechanization provides a

Alive is Awesome! What about MLIR?

'polynomia

The Polynomial dialect d

The simplest use of po

is another MLIR type like

More generally, this dial

some statically fixed pol

have the same remainde

with reductions modulo

Need very

MLIR

AliveInLean: A Verified LL
Optimization Ver

Juneyoung Lee[1(✉)], Chung-Kil Hur[1], an

[1] Seoul National Universi
Seoul, Republic of Korea
juneyoung.lee@sf.snu.ac
[2] Microsoft Research, Cambric

Abstract. Ensuring that compiler optimizatio tant for the reliability of the entire software e ware is compiled. Alive [12] is a tool for verifyin mizations. Since Alive was released, it has hel proactively find dozens of bugs in LLVM, avoidi miscompilations. Despite having verified many far, Alive is itself not verified, which has led to an optimization correct when it was not.

We introduce AliveInLean, a formally verifie verifier for LLVM. As the name suggests, Alive version of Alive developed in the Lean theorem that the proof obligations are correctly discha AliveInLean gives the same level of correctness the-art formal frameworks such as CompCert lvm [26], while inheriting the advantages of A automation and easy adoption by compiler deve

Keywords: Compiler verification · Peephole o

Verifying Peephole Rewriting In SSA Compiler IRs

Siddharth Bhat ✉
Cambridge University, United Kingdom

Alex Keizer ✉
Cambridge University, United Kingdom

Chris Hughes ✉
University of Edinburgh, United Kingdom

Andrés Goens ✉
University of Amsterdam, Netherlands

Tobias Grosser ✉
Cambridge University, United Kingdom

Abstract

There is an increasing need for domain-specific reasoning in modern compilers. This has fueled the use of tailored intermediate representations (IRs) based on static single assignment (SSA), like in the MLIR compiler framework. Interactive theorem provers (ITPs) provide strong guarantees for the end-to-end verification of compilers (e.g., CompCert). However, modern compilers and their IRs evolve at a rate that makes proof engineering alongside them prohibitively expensive. Nevertheless, well-scoped push-button automated verification tools such as the Alive peephole verifier for LLVM-IR gained recognition in domains where SMT solvers offer efficient (semi) decision procedures. In this paper, we aim to combine the convenience of automation with the versatility of ITPs for verifying peephole rewrites across domain-specific IRs. We formalize a core calculus for SSA-based IRs that is generic over the IR and covers so-called regions (nested scoping used by many domain-specific IRs in the MLIR ecosystem). Our mechanization in the Lean proof assistant provides a user-friendly frontend for translating MLIR syntax into our calculus. We provide scaffolding for defining and verifying peephole rewrites, offering tactics to eliminate the abstraction overhead of our SSA calculus. We prove correctness theorems about peephole rewriting, as well as two classical program transformations. To evaluate our framework, we consider three use cases from the MLIR ecosystem that cover different levels of abstractions: (1) bitvector rewrites from LLVM, (2) structured control flow, and (3) fully homomorphic encryption. We envision that our mechanization provides a

A Three Minute Detour Into Lean

```
def max (a b : Nat) : Nat :=
 if a > b then a else b
```

A Three Minute Detour Into Lean

```
def max (a b : Nat) : Nat :=
 if a > b then a else b
#eval max 3 4     /- = 4 -/
```

A Three Minute Detour Into Lean

```
def max (a b : Nat) : Nat :=
 if a > b then a else b

#eval max 3 4      /- = 4 -/

theorem max_commutative (a b : Nat) : max a b = max b a
```

Three cases:
1. If a < b, then we know that (max a b) will take the else branch, and (max b a) will take the then branch, returning the value b in both cases.
1. if a = b, then we are done immediately, since left and right hand side become identical.
3. If a > b, then proof is same as (a > b) case.

A Three Minute Detour Into Lean

```lean
def max (a b : Nat) : Nat :=
 if a > b then a else b

#eval max 3 4     /- = 4 -/

theorem max_commutative (a b : Nat) : max a b = max b a := by
 simp [max]
 by_cases h : b < a
 · simp [h]
   have h₁ : ¬ (a < b) := by omega
   simp [h₁]
 · simp [h]
   by_cases h₁ : a = b
   · simp [h₁]
   · have h₂ : a < b := by omega
     simp [h₂]
```

✅

Alive is Awesome! What about MLIR?

'polynomial' Dialect

The Polynomial dialect defines single-variable polynomial types and operations.

The simplest use of `polynomial` is to represent mathematical operations in a polynomial ring `R[x]`, where `R` is another MLIR type like `i32`.

More generally, this dialect supports representing polynomial operations in a quotient ring `R[X]/(f(x))` for some statically fixed polynomial `f(x)`. Two polyomials `p(x), q(x)` are considered equal in this ring if they have the same remainder when dividing by `f(x)`. When a modulus is given, ring operations are performed with reductions modulo `f(x)` and relative to the coefficient ring `R`.

Need very clever encodings of concepts into SMT-LIB :(

Alive is Awesome! What about MLIR?

'polynomial' Dialect

The Polynomial dialect defines single-variable polynomial types and operations.

The simplest use of `polynomial` is to represent mathematical operations in a polynomial ring `R[x]`, where `R` is another MLIR type like `i32`.

More generally, this dialect supports representing polynomial operations in a quotient ring `R[X]/(f(x))` for some statically fixed polynomial `f(x)`. Two polyomials `p(x), q(x)` are considered equal in this ring if they have the same remainder when dividing by `f(x)`. When a modulus is given, ring operations are performed with reductions modulo `f(x)` and relative to the coefficient ring `R`.

```
variable (q t : Nat) [Fact (q > 1)] (n : Nat)
noncomputable def f : (ZMod q)[X] := X^(2^n) + 1
abbrev R := (ZMod q)[X] / (Ideal.span {f q n})
```

Z3

Aliv

'poly

The Polyn

The simple
is another

More gene
some stati
have the s
with reduc

N

```
var
non
abb
```

```
/-  `x^(2^n) + a = a`, since we quotient the polynomial ring with x^(2^n) -/
open MLIR AST in
noncomputable def p1 : PeepholeRewrite (FHE q n) [.polynomialLike]
.polynomialLike :=
 { lhs := a_plus_generator_eq_a,
   rhs := rhs,
   correct := by
    ...
    have hgenerator :
       f q n - (1 : Polynomial (ZMod q)) =
         (Polynomial.monomial (R := ZMod q) (2^n : Nat) 1)  := by
           simp [f, Polynomial.X_pow_eq_monomial]
     rw [← hgenerator]
     have add_congr_quotient :
         ((Ideal.Quotient.mk (Ideal.span {f q n})) (f q n - 1) + 1)  =
           ((Ideal.Quotient.mk (Ideal.span {f q n})) (f q n )) := by
       simp
     rw [add_congr_quotient]
     apply Poly.add_f_eq
   }
```


Automation
Z3
Lean
Completeness


Automation
Z3
Lean
MLIR?
Lean
Completeness


Automation
Z3
Replicate Alive-style tactics
Lean
MLIR
Streamline
manual proof
Lean
Completeness


Automation
Z3
Lean
MLIR
Lean
Completeness


Automation
Z3
Lean
MLIR
Lean
Completeness


Automation
Z3
Replicate Alive-style tactics
Lean
MLIR
Streamline
manual proof
Lean
Completeness

Lean-MLIR: Goals

Evolving Semantics with MLIR 💙

Peephole Verification 💡
- Make easy things **trivial**
- Make hard things **possible**

Lean-MLIR: The Alive Experience™ for MLIR

Verifying Peephole Optimizations from LLVM

The Poly IR: Why Mathlib

Defining a Dialect in Lean-MLIR

The Alive Experience in Lean-MLIR

```
/-- %y = %x + 0 -/
def lhs : Com Op [.int] .int :=
```

The Alive Experience in Lean-MLIR

```
/-- %y = %x + 0 -/
def lhs : Com Op [.int] .int := [mlir_icom| {




}]
```

The Alive Experience in Lean-MLIR

```
/-- %y = %x + 0 -/
def lhs : Com Op [.int] .int := [mlir_icom| {
 ^bb0(%x: int):
 %0 = const 0 : () -> int
 %1 = add (%x, %0) : (int, int) -> int
 return (%1) : (int) -> ()
}]
```

The Alive Experience in Lean-MLIR

```
/-- %y = %x + 0 -/
def lhs : Com Op [.int] .int := [mlir_icon
 ^bb0(%x: int):
 %0 = const 0: () -> int
 %1 = add (%x, %0) : (int, int) -> int
 return (%1) : (int) -> ()
}]
```

Operations ¶

Syntax:

```
operation               ::= op-result-list? (generic-operation | custom-operation)
                            trailing-location?
generic-operation       ::= string-literal `(` value-use-list? `)`  successor-list?
                            dictionary-properties? region-list? dictionary-attribute?
                            `:` function-type
custom-operation        ::= bare-id custom-operation-format
op-result-list          ::= op-result (`,` op-result)* `=`
op-result               ::= value-id (`:` integer-literal)?
successor-list          ::= `[` successor (`,` successor)* `]`
successor               ::= caret-id (`:` block-arg-list)?
dictionary-properties   ::= `<` dictionary-attribute `>`
region-list             ::= `(` region (`,` region)* `)`
dictionary-attribute    ::= `{` (attribute-entry (`,` attribute-entry)*)? `}`
trailing-location       ::= `loc` `(` location `)`
```

The Alive Experience in Lean-MLIR

```
/-!
# MLIR OPS WITH REGIONS AND ATTRIBUTES AND BASIC BLOCK ARGS
-/

-- Op with potential result
syntax
  (mlir_op_operand "=")?
  str "(" mlir_op_operand,* ")"
          ( "(" mlir_region,* ")" )?
          (mlir_attr_dict)?
  ":" "(" mlir_type,* ")" "->" "("mlir_type,*")" : mlir_op

macro_rules
  | `([mlir_op| $x]) => `(mlir_op| $x)

macro_rules
  | `([mlir_op| $$($x)]) => return x

macro_rules
  | `(mlir_op|
        $[ $resName = ]?
        $name:str
        ( $operandsNames,* )
        $[ ( $rgns,* ) ]?
        $[ $attrDict ]?
        : ( $operandsTypes,* ) -> ( $resTypes,* ) ) => do
```

Operations ¶

Syntax:

```
operation             ::= op-result-list? (generic-operation | custom-operation)
                          trailing-location?
generic-operation     ::= string-literal `(` value-use-list? `)`  successor-list?
                          dictionary-properties? region-list? dictionary-attribute?
                          `:` function-type
custom-operation      ::= bare-id custom-operation-format
op-result-list        ::= op-result (`,` op-result)* `=`
op-result             ::= value-id (`:` integer-literal)?
successor-list        ::= `[` successor (`,` successor)* `]`
successor             ::= caret-id (`:` block-arg-list)?
dictionary-properties ::= `<` dictionary-attribute `>`
region-list           ::= `(` region (`,` region)* `)`
dictionary-attribute  ::= `{` (attribute-entry (`,` attribute-entry)*)? `}`
trailing-location     ::= `loc` `(` location `)`
```

The Alive Experience in Lean-MLIR

```
/-- %y = %x + 0 -/
def lhs : Com Op [.int] .int := [mlir_icom| {
 ^bb0(%x: int):
 %0 = const 0 : () -> int
 %1 = add (%x, %0) : (int, int) -> int
 return (%1) : (int) -> ()
}]
```

The Alive Experience in Lean-MLIR

```
/-- %y = %x + 0 -/
def lhs : Com Op [.int] .int := [mlir_icom| {
 ^bb0(%x: int):
 %0 = const 0 : () -> int
 %1 = add (%x, %0) : (int, int) -> int
 return (%1) : (int) -> ()
}]

/-- %y = %x -/
def rhs : Com Op [.int] .int :=
[mlir_icom| {
 ^bb0(%x: int):
 return (%x) : (int) -> ()
}]
```

The Alive Experience in Lean-MLIR

```
/-- %y = %x + 0 -/
def lhs : Com Op [.int] .int := [mlir_icom| {
 ^bb0(%x: int):
 %0 = const 0 : () -> int
 %1 = add (%x, %0) : (int, int) -> int
 return (%1) : (int) -> ()
}]

/-- %y = %x -/
def rhs : Com Op [.int] .int :=
[mlir_icom| {
 ^bb0(%x: int):
 return (%x) : (int) -> ()
}]
```

```
def p1 : PeepholeRewrite Op [.int] .int :=
```

The Alive Experience in Lean-MLIR

```
/-- %y = %x + 0 -/
def lhs : Com Op [.int] .int := [mlir_icom| {
 ^bb0(%x: int):
 %0 = const 0 : () -> int
 %1 = add (%x, %0) : (int, int) -> int
 return (%1) : (int) -> ()
}]

/-- %y = %x -/
def rhs : Com Op [.int] .int :=
[mlir_icom| {
 ^bb0(%x: int):
 return (%x) : (int) -> ()
}]
```

```
def p1 : PeepholeRewrite Op [.int] .int :=
  { lhs := lhs, rhs := rhs, correct :=
    by
```

The Alive Experience in Lean-MLIR

```
/-- %y = %x + 0 -/
def lhs : Com Op [.int] .int := [mlir_icom| {
 ^bb0(%x: int):
 %0 = const 0 : () -> int
 %1 = add (%
 return (%1)
}]

/-- %y = %x -/
def rhs : Com Op [.int] .int :=
[mlir_icom| {
 ^bb0(%x: int):
 return (%x) : (int) -> ()
}]
```

Eliminate SSA-boilerplate

```
def p1 : PeepholeRewrite Op [.int] .int :=
  { lhs := lhs, rhs := rhs, correct :=
    by
      rw [lhs, rhs]
      funext Γv
      simp_peephole [add, cst] at Γv
      /- ⊢ ∀ (a : BitVec 32),
         a + BitVec.ofInt 32 0 = a -/
```

The Alive Experience in Lean-MLIR

```
/-- %y = %x + 0 -/
def lhs : Com Op [.int] .int := [mlir_icom| {
 ^bb0(%x: int):
 %0 = const 0 : () -> int
 %1 = add (%
 return (%1)
}]

/-- %y = %x
def rhs : Co
[mlir_icom|
 ^bb0(%x: int):
 return (%x) : (int) -> ()
}]
```

Eliminate SSA-boilerplate

Try proof-hammer

```
def p1 : PeepholeRewrite Op [.int] .int :=
  { lhs := lhs, rhs := rhs, correct :=
    by
      rw [lhs, rhs]
      funext Γv
      simp_peephole [add, cst] at Γv
      /- ⊢ ∀ (a : BitVec 32),
         a + BitVec.ofInt 32 0 = a -/
      intros a
      simp_alive
      /- goals accomplished 🎉 -/
    }
```

LEAN

Lean-MLIR · Examples · Load

```lean
import SSA.Projects.InstCombine.LLVM.PrettyEDSL
import SSA.Projects.InstCombine.Refinement
import SSA.Projects.InstCombine.Tactic
import SSA.Projects.InstCombine.TacticAuto

/-- x + 0 -/
def lhs := [llvm| {
  ^bb0(%x : i32):
    %0 = llvm.mlir.constant 0 : i32
    %1 = llvm.add %x, %0 : i32
    llvm.return %1 :  i32
 }]

/-- x -/
def rhs := [llvm| {
  ^bb0(%x : i32):
    llvm.return %x : i32
 }]

def p1 :=
 { lhs := lhs, rhs := rhs, correct :=
  by
    rw [lhs, rhs]
    funext Γv; revert Γv
    simp_alive_peephole
    simp_alive_undef
    simp_alive_ops
    simp_alive_case_bash
    simp
    /- No goals -/
  : PeepholeRewrite ..
  }
```

▼ lean-mlir.lean:30:9

▼ Tactic state

No goals

▶ Expected type

▶ All Messages (0)

```
import SSA.Projects.InstCombine.LLVM.PrettyEDSL
import SSA.Projects.InstCombine.Refinement
import SSA.Projects.InstCombine.Tactic
import SSA.Projects.InstCombine.TacticAuto

/-- x + 0 -/
def lhs := [llvm| {
  ^bb0(%x : i32):
    %0 = llvm.mlir.constant 0 : i32
    %1 = llvm.add %x, %0 : i32
    llvm.return %1 :  i32
}]

/-- x -/
def rhs := [llvm| {
  ^bb0(%x : i32):
    llvm.return %x
}]

def p1 :=
  { lhs := lhs, rhs := rhs, correct :=
    by
      rw [lhs, rhs]
      funext Γv; revert Γv
      simp_alive_peephole
      simp_alive_undef
      simp_alive_ops
      simp_alive_case_bash
      simp
      /- No goals -/
    : PeepholeRewrite ..
  }
```

Playground Link @ lean-mlir.grosser.es

▼ lean-mlir.lean:30:9

▼ Tactic state

No goals

▶ Expected type

▶ All Messages (0)

Lean-MLIR Examples Load

```
6 def src  := [llvm| {
7   ^bb0(%0 : i32):
8     %c1 = llvm.mlir.constant 1 : i32
9     %r = llvm.udiv %0, %c1 : i32
10    llvm.return %r : i32
11 }]
12
13 def tgt := [llvm| {
14   ^bb0(%0 : i32):
15   %c13 = llvm.mlir.constant 13 : i32
16   %r = llvm.lshr %0, %c13 : i32
17   llvm.return %r : i32
18 }]
19
20 theorem equiv?  : src ⊑ tgt := by
21   unfold src tgt
22   simp_alive_peephole
23   simp_alive_undef
24   simp_alive_ops
25   simp_alive_case_bash
26   intros x
27   simp
28   /- x = x >>> 13 -/
29   bv_decide
30   -- The prover found a counterexample, consider the following assignment:
31   -- x = 0xffffffff#32
32
```

▼lean-mlir.lean:29:11

▼Tactic state

No goals

▼Messages (1)

▼lean-mlir.lean:29:2

```
The prover found a potential
counterexample, consider the following
assignment:
x = 0xffffffff#32
```

▶ All Messages (1)

Lean-MLIR Examples Load

```lean
import SSA.Projects.InstCombine.LLVM.PrettyEDSL
import SSA.Projects.InstCombine.Refinement
import SSA.Projects.InstCombine.Tactic
import SSA.Projects.InstCombine.TacticAuto


def alive_AddSub_1043_src  (w : Nat)   :=
[llvm ( w )| {
^bb0(%C1 : _, %Z : _, %RHS : _):
  %v1 = llvm.and %Z, %C1
  %v2 = llvm.xor %v1, %C1
  %v3 = llvm.mlir.constant 1
  %v4 = llvm.add %v2, %v3
  %v5 = llvm.add %v4, %RHS
  llvm.return %v5
}]

def alive_AddSub_1043_tgt  (w : Nat)  :=
[llvm ( w )| {
^bb0(%C1 : _, %Z : _, %RHS : _):
  %v1 = llvm.not %C1
  %v2 = llvm.or %Z, %v1
  %v3 = llvm.and %Z, %C1
  %v4 = llvm.xor %v3, %C1
  %v5 = llvm.mlir.constant 1
  %v6 = llvm.add %v4, %v5
  %v7 = llvm.sub %RHS, %v2
  llvm.return %v7
}]
theorem alive_AddSub_1043  (w : Nat)   : alive_AddSub_1043_src w  ⊑ alive_AddSub_1043_tgt w  := by
  unfold alive_AddSub_1043_src alive_AddSub_1043_tgt
  simp_alive_peephole
  simp_alive_undef
  simp_alive_ops
  simp_alive_case_bash
  alive_auto
```

▼ lean-mlir.lean:36:12

▼ Tactic state

No goals

▶ All Messages (0)

Lean-MLIR Examples Load

```
import SSA.Projects.InstCombine.LLVM.PrettyEDSL
import SSA.Projects.InstCombine.Refinement
import SSA.Projects.InstCombine.Tactic
import SSA.Projects.InstCombine.TacticAuto


def alive_AddSub_1043_src  (w : Nat)   :=
[llvm ( w )| {
^bb0(%C1 : _, %Z : _, %RHS : _):
  %v1 = llvm.and %Z, %C1
  %v2 = llvm.xor %v1, %C1
  %v3 = llvm.mlir.constant 1
  %v4 = llvm.add %v2, %v3
  %v5 = llvm.add %v4, %RHS
  llvm.return %v5
}]

def alive_AddSub_1043_tgt  (w : Nat)  :=
[llvm ( w )| {
^bb0(%C1 : _, %Z : _, %RHS : _):
  %v1 = llvm.not %C1
  %v2 = llvm.or %Z, %v1
  %v3 = llvm.and %Z, %C1
  %v4 = llvm.xor %v3, %C1
  %v5 = llvm.mlir.constant 1
  %v6 = llvm.add %v4, %v5
  %v7 = llvm.sub %RHS, %v2
  llvm.return %v7
}]
theorem alive_AddSub_1043  (w : Nat)   : alive_AddSub_1043_src w  ⊑ alive_AddSub_1043_tgt w  := by
  unfold alive_AddSub_1043_src alive_AddSub_1043_tgt
  simp_alive_peephole
  simp_alive_undef
  simp_alive_ops
  simp_alive_case_bash
  alive_auto
```

Generic Width

▼ lean-mlir.lean:36:12
▼ Tactic state
No goals
▶ All Messages (0)

LEAN

Lean-MLIR · Examples · Load

```
import SSA.Projects.InstCombine.LLVM.PrettyEDSL
import SSA.Projects.InstCombine.Refinement
import SSA.Projects.InstCombine.Tactic
import SSA.Projects.InstCombine.TacticAuto


def alive_AddSub_1043_src  (w : Nat)   :=
[llvm ( w )| {
^bb0(%C1 : _, %Z : _, %RHS : _):
  %v1 = llvm.and %Z, %C1
  %v2 = llvm.xor %v1, %C1
  %v3 = llvm.mlir.constant 1
  %v4 = llvm.add %v2, %v3
  %v5 = llvm.add %v4, %RHS
  llvm.return %v5
}]

def alive_
[llvm ( w
^bb0(%C1 :
  %v1 = llvm.not %C1
  %v2 = llvm.or %Z, %v1
  %v3 = llvm.and %Z, %C1
  %v4 = llvm.xor %v3, %C1
  %v5 = llvm.mlir.constant 1
  %v6 = llvm.add %v4, %v5
  %v7 = llvm.sub %RHS, %v2
  llvm.return %v7
}]
theorem alive_AddSub_1043  (w : Nat)   : alive_AddSub_1043_src w  ⊑ alive_AddSub_1043_tgt w  := by
  unfold alive_AddSub_1043_src alive_AddSub_1043_tgt
  simp_alive_peephole
  simp_alive_undef
  simp_alive_ops
  simp_alive_case_bash
  alive_auto
```

[Playground Link @ lean-mlir.grosser.es](https://lean-mlir.grosser.es)

▼ lean-mlir.lean:36:12

▼ Tactic state

No goals

▶ All Messages (0)

Proof Automation for Push Button Verification*

Tactic	Hacker's Delight	Alive	InstCombine
Total	112	93	866
`bv_ring`	0	10	0
`bv_decide` (symbolic width)	3	3	N/A
`bv_decide` (concrete width $w = 64$)	34	51	581
`bv_automata`	27	49	399
`alive_auto`	32	67	567

Fig. 5. Comparison of the various tactics we have for automatically proving bitvector rewrites across three datasets. See that the bv_automata tactic, which proves results for arbitrary width, is competitive with bv_decide, a complete decision procedure that equations for finite width.

💖 **Lean FRO, mathlib community!**
Special thanks to Henrik & Kim.

Fully Homomorphic Encryption: Complex Proofs

'polynomial' Dialect

The Polynomial dialect defines single-variable polynomial types and operations.

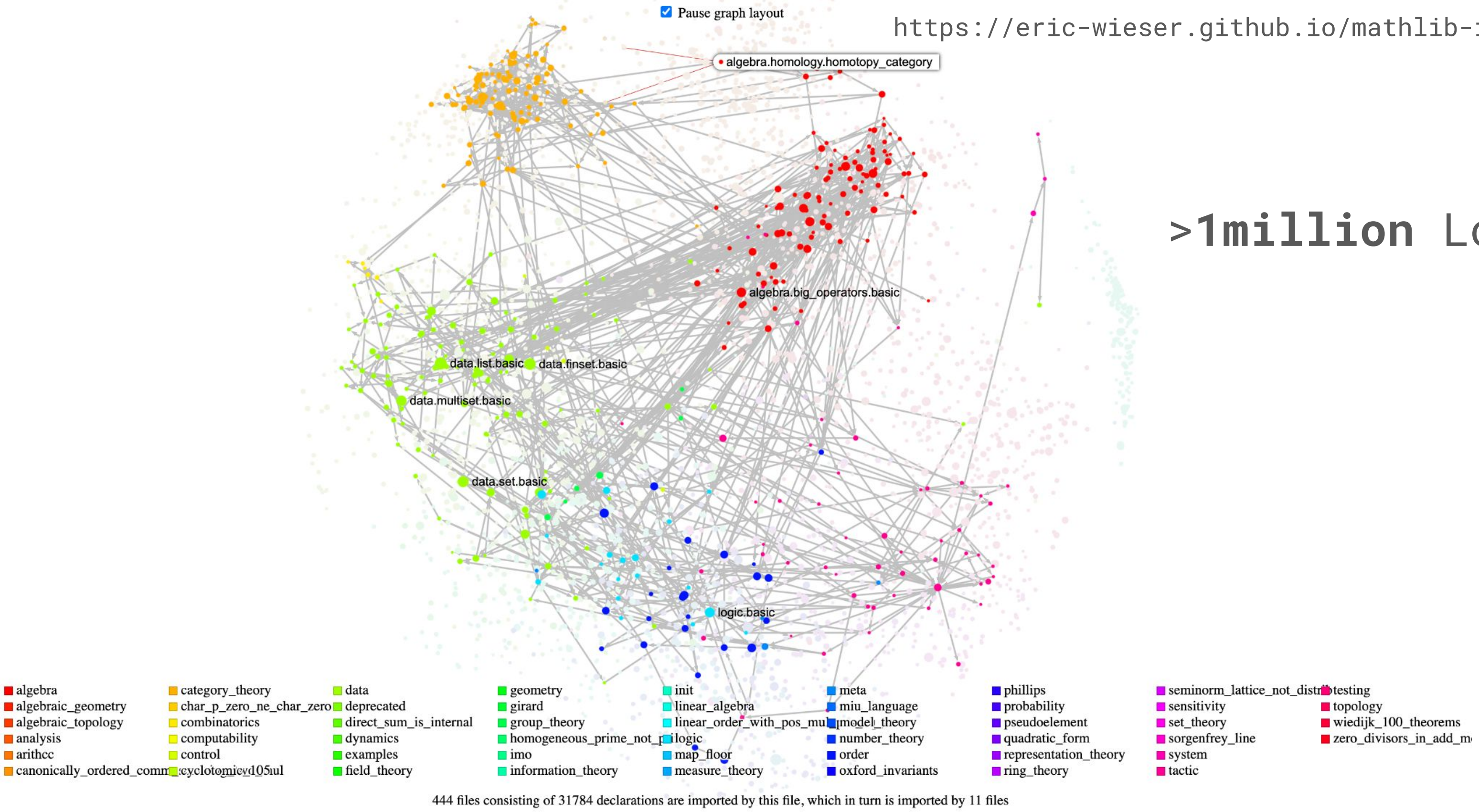
The simplest use of `polynomial` is to represent mathematical operations in a polynomial ring `R[x]`, where `R` is another MLIR type like `i32`.

More generally, this dialect supports representing polynomial operations in a quotient ring `R[X]/(f(x))` for some statically fixed polynomial `f(x)`. Two polyomials `p(x), q(x)` are considered equal in this ring if they have the same remainder when dividing by `f(x)`. When a modulus is given, ring operations are performed with reductions modulo `f(x)` and relative to the coefficient ring `R`.

```
variable (q t : Nat) [Fact (q > 1)] (n : Nat)
noncomputable def f : (ZMod q)[X] := X^(2^n) + 1
abbrev R := (ZMod q)[X] / (Ideal.span {f q n})
```

Mathlib: The World's Largest Formal Math Repo

https://eric-wieser.github.io/mathlib-import-graph/



>1million LoC

Fully Homomorphic Encryption: Complex Proofs

'polynomial' Dialect

The Polynomial dialect defines single-variable polynomial types and operations.

The simplest use of `polynomial` is to represent mathematical operations in a polynomial ring `R[x]`, where `R` is another MLIR type like `i32`.

More generally, this dialect supports representing polynomial operations in a quotient ring `R[X]/(f(x))` for some statically fixed polynomial `f(x)`. Two polyomials `p(x), q(x)` are considered equal in this ring if they have the same remainder when dividing by `f(x)`. When a modulus is given, ring operations are performed with reductions modulo `f(x)` and relative to the coefficient ring `R`.

```
variable (q t : Nat) [Fact (q > 1)] (n : Nat)
noncomputable def f : (ZMod q)[X] := X^(2^n) + 1
abbrev R := (ZMod q)[X] / (Ideal.span {f q n})
```

Fully I… ofs

'polynor

The Polynomial dia

The simplest use c

is another MLIR ty

More generally, thi

some statically fixe

have the same rem

with reductions mo

```
variable (
noncomputa
abbrev R :
```

```
/-  `x^(2^n) + a = a`, since we quotient the polynomial ring with x^(2^n) -/
open MLIR AST in
noncomputable def p1 : PeepholeRewrite (FHE q n) [.polynomialLike]
.polynomialLike :=
 { lhs := a_plus_generator_eq_a,
   rhs := rhs,
   correct := by
    ...
   have hgenerator :
      f q n - (1 : Polynomial (ZMod q)) =
        (Polynomial.monomial (R := ZMod q) (2^n : Nat) 1)  := by
          simp [f, Polynomial.X_pow_eq_monomial]
    rw [← hgenerator]
    have add_congr_quotient :
        ((Ideal.Quotient.mk (Ideal.span {f q n})) (f q n - 1) + 1)  =
          ((Ideal.Quotient.mk (Ideal.span {f q n})) (f q n )) := by
      simp
    rw [add_congr_quotient]
    apply Poly.add_f_eq
 }
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add     => ⟨[.int, .int], .int⟩
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add    => ⟨[.int, .int], .int⟩
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add     => ⟨[.int, .int], .int⟩
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add    => ⟨[.int, .int], .int⟩
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add   => ⟨[.int, .int], .int⟩
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add     => ⟨[.int, .int], .int⟩
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add    => ⟨[.int, .int], .int⟩
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add   => ⟨[.int, .int], .int⟩
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add   => ⟨[.int, .int], .int⟩
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add     => ⟨[.int, .int], .int⟩
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add    => ⟨[.int, .int], .int⟩
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add    => ⟨[.int, .int], .int⟩
```

declare operation semantics

```
instance : OpDenote Op Ty where
 denote -- (o : Op) -> denote (signature o).args -> denote o.retTy
   | .const n, [] => BitVec.ofInt 32 n
   | .add, [(a : BitVec 32), (b : BitVec 32)]ₕ => a + b
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add     => ⟨[.int, .int], .int⟩
```

declare operation semantics

```
instance : OpDenote Op Ty where
 denote -- (o : Op) -> denote (signature o).args -> denote o.retTy
    | .const n, [] => BitVec.ofInt 32 n
    | .add, [(a : BitVec 32), (b : BitVec 32)]ₕ => a + b
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add    => ⟨[.int, .int], .int⟩
```

declare operation semantics

```
instance : OpDenote Op Ty where
  denote -- (o : Op) -> denote (signature o).args -> denote o.retTy
    | .const n, [] => BitVec.ofInt 32 n
    | .add, [(a : BitVec 32), (b : BitVec 32)]□ => a + b
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add    => ⟨[.int, .int], .int⟩
```

declare operation semantics

```
instance : OpDenote Op Ty where
 denote -- (o : Op) -> denote (signature o).args -> denote o.retTy
   | .const n, [] => BitVec.ofInt 32 n
   | .add, [(a : BitVec 32), (b : BitVec 32)]□ => a + b
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add   => ⟨[.int, .int], .int⟩
```

declare operation semantics

```
instance : OpDenote Op Ty where
 denote -- (o : Op) -> denote (signature o).args -> denote o.retTy
   | .const n, [] => BitVec.ofInt 32 n
   | .add, [(a : BitVec 32), (b : BitVec 32)]□ => a + b
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add    => ⟨[.int, .int], .int⟩
```

declare operation semantics

```
instance : OpDenote Op Ty where
 denote -- (o : Op) -> denote (signature o).args -> denote o.retTy
   | .const n, [] => BitVec.ofInt 32 n
   | .add, [(a : BitVec 32), (b : BitVec 32)] => a + b
```

Lean-MLIR: Declaring an IR

declare types

```
inductive Ty
| int
```

declare operations

```
inductive Op : Type
| add : Op
| const : (val : ℤ) → Op
```

declare type semantics

```
instance : TyDenote Ty where
  toType  -- Ty -> Type
    | .int => BitVec 32
```

declare operation signature

```
instance : OpSignature Op Ty where
  signature -- Op -> Signature
    | .const _ => ⟨[], .int⟩
    | .add   => ⟨[.int, .int], .int⟩
```

declare operation semantics

```
instance : OpDenote Op Ty where
  denote -- (o : Op) -> denote (signature o).args -> denote o.retTy
    | .const n, [] => BitVec.ofInt 32 n
    | .add, [(a : BitVec 32), (b : BitVec 32)] => a + b
```

Lean-MLIR: Where The Semantics Gets Used

```
/-- %y = %x + 0 -/
def lhs : Com Op [.int] .int := [mlir_icom| {
 ^bb0(%x: int):
 %0 = const 0 : () -> int
 %1 = add (%x, %0) : (int, int) -> int
 return (%1) : (int) -> ()
}]
```

declare operation semantics

```
instance : OpDenote Op Ty where
 denote -- (o : Op) -> denote (signature o).args -> denote o.retTy
    | .const n, [] => BitVec.ofInt 32 n
    | .add, [(a : BitVec 32), (b : BitVec 32)] => a + b
```

Lean-MLIR: Where The Semantics Gets Used

```
/-- %y = %x + 0 -/
def lhs : Com Op [.int] .int := [mlir_icom| {
 ^bb0(%x: int):
 %0 = const 0 : () -> int
 %1 = add (%x, %0) : (int, int) -> int
 return (%1) : (int) -> ()
}]
```

declare operation semantics

```
instance : OpDenote Op Ty where
 denote -- (o : Op) -> denote (signature o).args -> denote o.retTy
   | .const n, [] => BitVec.ofInt 32 n
   | .add, [(a : BitVec 32), (b : BitVec 32)] => a + b
```

Lean-MLIR: Where The Semantics Gets Used

```
/-- %y = %x + 0 -/
def lhs : Com Op [.int] .int := [mlir_icom| {
 ^bb0(%x: int):
 %0 = const 0 : () -> int
 %1 = add (%x, %0) : (int, int) -> int
 return (%1) : (int) -> ()
}]
```

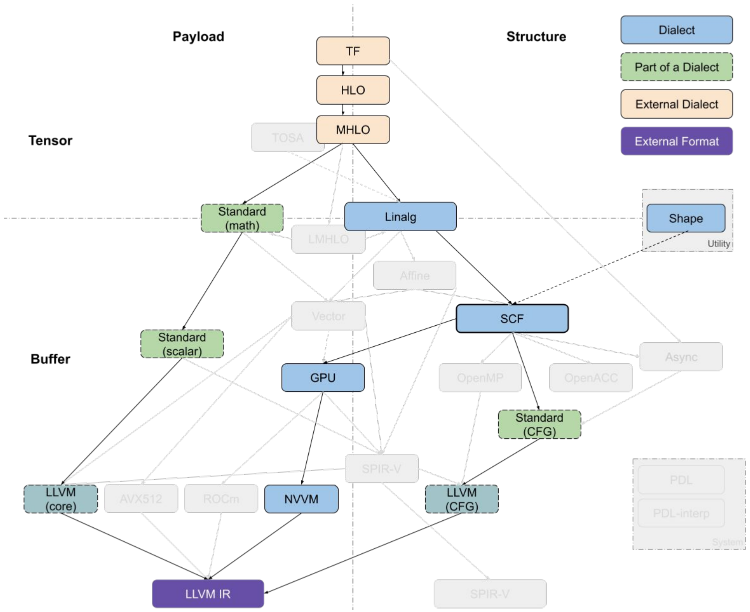
We need the semantics!
Please give us semantics 💛

declare operation semantics

```
instance : OpDenote Op Ty where
 denote -- (o : Op) -> denote (signature o).args -> denote o.retTy
   | .const n, [] => BitVec.ofInt 32 n
   | .add, [(a : BitVec 32), (b : BitVec 32)] => a + b
```

Formally Verifying Peephole Optimizations for MLIR!

github.com/opencompl/lean-mlir




```
def p1 : PeepholeRewrite Op [.int] .int := {
  lhs := lhs,
  rhs := rhs,
  correct := by
   rw [lhs, rhs]; funext Γv;
   simp_peephole [add, cst] at Γv
   /- ⊢ ∀ (a : BitVec 32), a + BitVec.ofInt 32 0 = a -/
   intros a; simp_alive /- goals accomplished 🎉 -/
}
```

Evolving Semantics with MLIR 💙

Ease-of-use 💙

Peephole Proofs 💡

UB versus Poison

Taming Undefined Behavior in LLVM

Juneyoung Lee
Yoonseung Kim
Youngju Song
Chung-Kil Hur
Seoul National University, Korea
{juneyoung.lee, yoonseung.kim, youngju.song, gil.hur}@sf.snu.ac.kr

Sanjoy Das
Azul Systems, USA
sanjoy@azul.com

David Majnemer
Google, USA
majnemer@google.com

John Regehr
University of Utah, USA
regehr@cs.utah.edu

Nuno P. Lopes
Microsoft Research, UK
nlopes@microsoft.com

Abstract

A central concern for an optimizing compiler is the design of its intermediate representation (IR) for code. The IR should make it easy to perform transformations, and should also afford efficient and precise static analysis.

In this paper we study an aspect of IR design that has received little attention: the role of undefined behavior. The IR for every optimizing compiler we have looked at, including GCC, LLVM, Intel's, and Microsoft's, supports one or more forms of undefined behavior (UB), not only to reflect the semantics of UB-heavy programming languages such as C and C++, but also to model inherently unsafe low-level operations such as memory stores and to avoid over-constraining IR semantics to the point that desirable transformations be-

1. Introduction

Some programming languages, intermediate representations, and hardware platforms define a set of erroneous operations that are untrapped and that may cause the system to behave badly. These operations, called *undefined behaviors*, are the result of design choices that can simplify the implementation of a platform, whether it is implemented in hardware or software. The burden of avoiding these behaviors is then placed upon the platform's users. Because undefined behaviors are untrapped, they are insidious: the unpredictable behavior that they trigger often only shows itself much later.

The AVR32 processor architecture document [2, p. 51] provides an example of hardware-level undefined behavior:

If the region has a size of 8 KB, the 13 lowest bits in

UB versus Poison

Taming Undefined Behavior in LLVM

Juneyoung Lee
Yoonseung Kim
Youngju Song
Chung-Kil Hur
Seoul National University, Korea
{juneyoung.lee, yoonseung.kin
youngju.song, gil.hur}@sf.snu.ac.

Sanjoy Das
Azul Systems, USA
sanjoy@azul.com

David Majnemer
Google, USA
majnemer@google.com

John Regehr

Nuno P. Lopes

Abstract

A central concern for an optimizing
its intermediate representation (IR
make it easy to perform transforn
afford efficient and precise static a
In this paper we study an aspec
ceived little attention: the role of u
for every optimizing compiler we
GCC, LLVM, Intel's, and Microso
forms of undefined behavior (UB
semantics of UB-heavy programm
and C++, but also to model inheren
ations such as memory stores and t
IR semantics to the point that desi

Alive2: Bounded Translation Validation for LLVM

Nuno P. Lopes
nlopes@microsoft.com
Microsoft Research
UK

Juneyoung Lee
juneyoung.lee@sf.snu.ac.k
Seoul National University
South Korea

Chung-Kil Hur
gil.hur@sf.snu.ac.k
Seoul National University
South Korea

Zhengyang Liu
liuz@cs.utah.edu
University of Utah
USA

John Regehr
regehr@cs.utah.edu
University of Utah
USA

Abstract

We designed, implemented, and deployed Alive2: a *bounded* translation validation tool for the LLVM compiler's intermediate representation (IR). It limits resource consumption by, for example, unrolling loops up to some bound, which means there are circumstances in which it misses bugs. Alive2 is designed to avoid false alarms, is fully automatic through the use of an SMT solver, and requires no changes to LLVM. By running Alive2 over LLVM's unit test suite, we discovered and reported 47 new bugs, 28 of which have been fixed already. Moreover, our work has led to eight patches to the LLVM Language Reference—the definitive description of the semantics of its IR—and we have participated in numerous discussions with the goal of clarifying ambiguities and fixing errors in these semantics. Alive2 is open source and we also

1 Introduction

LLVM is a popular open-source compiler that is used by numerous frontends (e.g., C, C++, Fortran, Rust, Swift), and that generates high-quality code for a variety of target architectures. We want LLVM to be correct but, like any large code base, it contains bugs. Proving functional correctness of about 2.6 million lines of C++ is still impractical, but a weaker formal technique—translation validation—can be used to certify that individual executions of the compiler respected its specification.

A key feature of LLVM that makes it a suitable platform for translation validation is its intermediate representation (IR), which provides a common point of interaction between frontends, backends, and middle-end transformation passes. LLVM IR has a specification document,[1] making it more

UB versus Poison

Taming Undefined Behavior in LLVM

Juneyoung Lee
Yoonseung Kim
Youngju Song
Chung-Kil Hur
Seoul National University, Korea
{juneyoung.lee, yoonseung.kim
youngju.song, gil.hur}@sf.snu.ac.

Sanjoy Das
Azul Systems, USA
sanjoy@azul.com

David Majnemer
Google, USA
majnemer@google.com

John Regehr

Nuno P. Lopes

Abstract

A central concern for an optimizing
its intermediate representation (IR)
make it easy to perform transforr
afford efficient and precise static a
In this paper we study an aspec
ceived little attention: the role of u
for every optimizing compiler we
GCC, LLVM, Intel's, and Microso
forms of undefined behavior (UB
semantics of UB-heavy programm
and C++, but also to model inheren
ations such as memory stores and t
IR semantics to the point that desi

Alive2: Bounded Translation Validation for LLVM

Nuno P. Lopes
nlopes@microsoft.com
Microsoft Research
UK

Juneyoung Lee
juneyoung.lee@sf.snu.ac.k
Seoul National University

Chung-Kil Hur
gil.hur@sf.snu.ac.k
Seoul National University

Abstract

We designed, implemented, a
translation validation tool for
diate representation (IR). It li
for example, unrolling loops u
there are circumstances in w
designed to avoid false alarm
the use of an SMT solver, and
By running Alive2 over LLV
ered and reported 47 new bug
already. Moreover, our work
LLVM Language Reference—t
semantics of its IR—and we h
discussions with the goal of cl
errors in these semantics. Aliv

Exploring C Semantics and Pointer Provenance

KAYVAN MEMARIAN, University of Cambridge, UK
VICTOR B. F. GOMES, University of Cambridge, UK
BROOKS DAVIS, SRI International, USA
STEPHEN KELL, University of Cambridge, UK
ALEXANDER RICHARDSON, University of Cambridge, UK
ROBERT N. M. WATSON, University of Cambridge, UK
PETER SEWELL, University of Cambridge, UK

67

The semantics of pointers and memory objects in C has been a vexed question for many years. C values cannot be treated as either purely abstract or purely concrete entities: the language exposes their representations, but compiler optimisations rely on analyses that reason about provenance and initialisation status, not just runtime representations. The ISO WG14 standard leaves much of this unclear, and in some respects differs with de facto standard usage — which itself is difficult to investigate.

In this paper we explore the possible source-language semantics for memory objects and pointers, in ISO C and in C as it is used and implemented in practice, focussing especially on pointer provenance. We aim to, as far as possible, reconcile the ISO C standard, mainstream compiler behaviour, and the semantics relied on by the corpus of existing C code. We present two coherent proposals, tracking provenance via integers and not; both address many design questions. We highlight some pros and cons and open questions, and illustrate the discussion with a library of test cases. We make our semantics executable as a test oracle, integrating it with the Cerberus semantics for much of the rest of C, which we have made substantially more complete and robust, and equipped with a web-interface GUI. This allows us to experimentally assess our proposals on those test cases. To assess their viability with respect to larger bodies of C code, we analyse the changes required and the resulting behaviour for a port of FreeBSD to CHERI, a research architecture supporting

Why Do We Trust Our LLVM Semantics?

- We model both UB and poison as poison. (details in paper)
- overapprox. **on purpose**, has taken experts years; out of scope.

Why Do We Trust Our LLVM Semantics?

- We model both UB and poison as poison. (details in paper)
- overapprox. **on purpose**, has taken experts years; out of scope.
- check correctness of semantics via cosim runs

```
Build completed successfully.
+ ../../.lake/build/bin/ssaLLVMEnumerator
+ diff generated-llvm-optimized-data.csv generated-ssa-llvm-semantics.csv
+ diff /dev/fd/63 /dev/fd/62
++ awk -F, '$2 == 4' generated-ssa-llvm-semantics.csv
++ sort -t, -k1,1
++ sort -t, -k1,1
++ awk -F, '$2 == 4' generated-ssa-llvm-syntax-and-semantics.csv
```

Alive Style Workflow for LLVM IR

∀ w, BitVec w

`(c ||| b) &&& a ||| c = c ||| a &&& b`	⟶	extensionality
`(c + b) * a = c * a + b * a`	⟶	ring
`(c &&& b ^^^ b) + 1 + a = a - (c ||| ~~~b)`	⟶	automata

BitVec 64

`(c ||| b) &&& a ||| c = c ||| a &&& b`	⟶	LeanSAT
`(c + b) * a = c * a + b * a`	⟶	LeanSAT
`(c &&& b ^^^ b) + 1 + a = a - (c ||| ~~~b)`	⟶	LeanSAT

Proof Automation for Push Button Verification*

Tactic	Hacker's Delight	Alive	InstCombine
Total	112	93	866
`bv_ring`	0	10	0
`bv_decide` (symbolic width)	3	3	N/A
`bv_decide` (concrete width $w = 64$)	34	51	581
`bv_automata`	27	49	399
`alive_auto`	32	67	567

Fig. 5. Comparison of the various tactics we have for automatically proving bitvector rewrites across three datasets. See that the `bv_automata` tactic, which proves results for arbitrary width, is competitive with `bv_decide`, a complete decision procedure that equations for finite width.

Manual Proofs for Complex Transformations

API Coverage for Manual Proof Writing

	add	sub	neg	abs	mul	udiv	sdiv	srem	smod	ofBool	fill	extractLsb'	zeroExtend	shiftLeftZeroExtend	zeroExtend'	signExtend	and	or	xor	not	shiftLeft	ushiftRight	sshiftRight	sshiftRight'	rotateLeft	rotateRight	append	replicate	concat	twoPow
toNat	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	-	-	✓	-	✓	-	✓	✓	✓	✓	✓	✓	-	-	-	-	✓	-	✓	✓
toInt	✓	-	✓	-	✓	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
toFin	✓	✓	✓	-	✓	-	-	-	-	-	-	-	-	-	-	-	✓	✓	✓	✓	✓	-	-	-	-	-	-	-	-	-
getElem	✓	-	-	-	✓	-	-	-	-	✓	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	✓	✓	✓	✓
getLsbD	✓	✓	-	-	✓	-	-	-	-	✓	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
getMsbD	✓	✓	-	-	-	-	-	-	-	-	-	-	-	✓	✓	-	✓	✓	✓	-	✓	✓	✓	✓	-	-	✓	-	-	-
msb	✓	✓	-	-	-	-	-	-	-	✓	-	-	-	✓	-	-	✓	✓	✓	✓	✓	✓	✓	✓	-	-	✓	-	-	-

Table 1. Our BitVector API for Lean implements all smtlib functions offering for each conversions to `Nat`, `Int`, and `Fin` as well as indexing for obtaining individual bits via `getElem`, `getLsbD`, `getMsbD` and `msb`.

Extras: Metatheoretic reasoning (CSE, DCE)

```
def cse [DecidableEq d.Ty] [DecidableEq d.Op]
 {α : d.Ty} {Γ : Ctxt d.Ty} (com: Com d Γ .pure α) :
 { com' : Com Op Γ α // com.denote = com'.denote }
```

```
def dce {Γ : Ctxt d.Ty} {t : d.Ty} (com : Com d Γ .pure t) :
 Σ (Γ' : Ctxt Ty) (hom : Ctxt.Hom Γ' Γ),
   { com' : Com Op Γ' t //  com.denote = com'.denote ∘ Valuation.comap hom }
```

Playground @ lean-mlir.grosser.es

LEAN

Lean-MLIR ⌄ ★ Examples ⇪ Load ☰

```
  simp only [ge_iff_le,
    EffectKind.return_impure_toMonad_eq, Option.pure_def, mul_eq,
    Option.bind_eq_bind, Option.none_bind, h, ↓reduceIte, Option.none_bind,
    Option.bind_none, Option.some_bind, Refinement.some_some, Refinement.refl]
  apply BitVec.eq_of_toNat_eq
  simp only [bv_toNat, Nat.mod_mul_mod]
  ring_nf

/--
info: 'AlivePaperExamples.shift_mul' depends on axioms: [propext, Classical.choice, Quot.sound]
-/
#guard_msgs in #print axioms shift_mul

-- Example proof of xor + sub, this is automatically closed by automation.
theorem xor_sub :
    [llvm (w)| {
  ^bb0(%X : _, %Y : _):
```

simp_alive_peephole extends simp_peephole to simplify goals about refinement of LLVM programs into statements about just bitvectors.
That is, the tactic expects a goal of the form: Com.Refinement com₁ com₂ That is, goals of the form Com.refine, com₁.denote Γv ⊑ com₂.denote Γv , where com₁ and com₂ are programs in the LLVM dialect.

busily processing...

```
    simp_alive_peephole
    alive_auto

/-- info: 'AlivePaperExamples.xor_sub' depends on axioms: [propext, Classical.choice, Quot.sound] -/
#guard_msgs in #print axioms xor_sub

theorem bitvec_AddSub_1309 :
  [llvm (w)| {
    ^bb0(%X : _, %Y : _):
      %v1 = llvm.and %X, %Y
      %v2 = llvm.or %X, %Y
      %v3 = llvm.add %v1, %v2
      llvm.return %v3
```

▼ lean-mlir.lean:61:17
▼ Tactic state
1 goal
w : ℕ
⊢ ∀ (e e_1 : LLVM.IntW w), LLVM.xor (LLVM.sub e_1 e_1) e ⊑ e
▶ All Messages (0)

Restart File