Toward A More Declarative InstCombine

Siddharth Bhat, *Léo Stefanesco, George Rennie, John Regehr, Tobias Grosser*

```
(x y : BV w), x + y = y + x
```

↓

```
(x' y' w' : BV 65), w' <= 64 ->
  let x := x' & (1 <<< w' - 1)
  let y := y' & (1 <<< w' - 1)
  x + y = y + x
```

THE UNIVERSITY of UTAH

UNIVERSITY OF CAMBRIDGE

EuroLLVM '26, 14 April 2026


Léo
Mog had a medal.
She also had an egg every day for breakfast.
Mr and Mrs Thomas told all their friends about her.
They said, "Mog is really remarkable."
And they never – (or almost never) – said, "Bother that cat!"


George


John


Tobias


Sid
Will My
Visa Arrive?

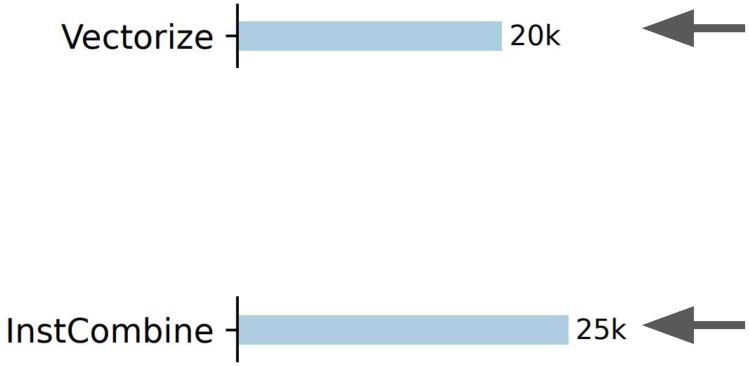
What's InstCombine?

LLVM’s Amazing Peephole Optimizer!

```
X + X     →   X <<< 1

-A + -B   →   - (A + B)

-A + B    →   B - A
```

… and thousands more!

Is InstCombine Code Common (in lines of code)?



InstCombine is larger than Vectorize?! 😱

Declarative Code Hidden in Plain ~~Sight~~ C++

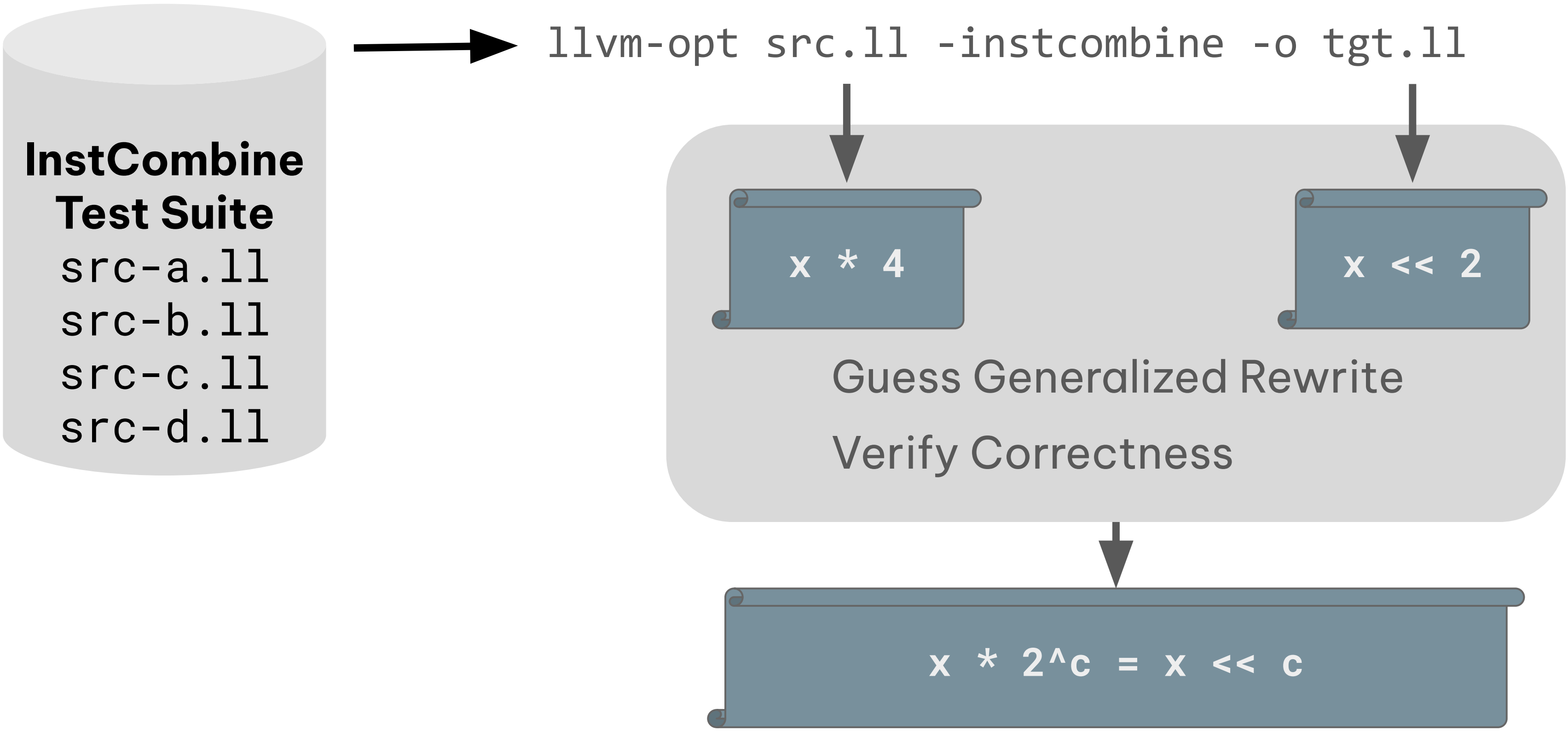
```
// X + X --> X << 1
if (LHS == RHS) {
  auto *Shl = BinaryOperator::CreateShl(LHS, ConstantInt::get(Ty, 1));
  Shl->setHasNoSignedWrap(I.hasNoSignedWrap());
  Shl->setHasNoUnsignedWrap(I.hasNoUnsignedWrap());
  return Shl;
}

Value *A, *B;
if (match(LHS, m_Neg(m_Value(A)))) {
  // -A + -B --> -(A + B)
  if (match(RHS, m_Neg(m_Value(B))))
    return BinaryOperator::CreateNeg(Builder.CreateAdd(A, B));

  // -A + B --> B - A
  auto *Sub = BinaryOperator::CreateSub(RHS, A);
  auto *OB0 = cast<OverflowingBinaryOperator>(LHS);
  Sub->setHasNoSignedWrap(I.hasNoSignedWrap() && OB0->hasNoSignedWrap());
```

InstCombine yearns to be declarative!

It facilitates

- **verification**
- maintenance
- optimization

Idealized Pipeline For Extracting Declarative Rewrites



LLVM Has "Verify" Already: 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!
```

LLVM Has “Verify” Already: Alive! – It’s used Daily!

[InstCombine] I
only if the and

XChy authored 3

Fixes #113123
Alive proof: https:

main (#113164)

1 parent 67ff5ba cc

Fold fma x,
(#100106)

dtcxzyw auth

Alive2 proof (Pl
`smt-to`):
https://alive2.l

main (#100

1 parent eb9bf18

[InstCombine] fold (a
|| (a != c && b =
== (b != c) (#949

zjaffal authored on Jun 23

resolves #92966

alive proof
https://alive2.llvm.org/ce/

main (#94915) · llvm
llvmorg-

1 parent 3ae6755 commit f7f

[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

LLVM Has "**Verify**" Already: 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!
```

LLVM Has **“Verify”** Already: 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
}
```

LLVM Has "Verify" Already: 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
}
```

```
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)
```

LLVM Has “Verify” Already: 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
}
```

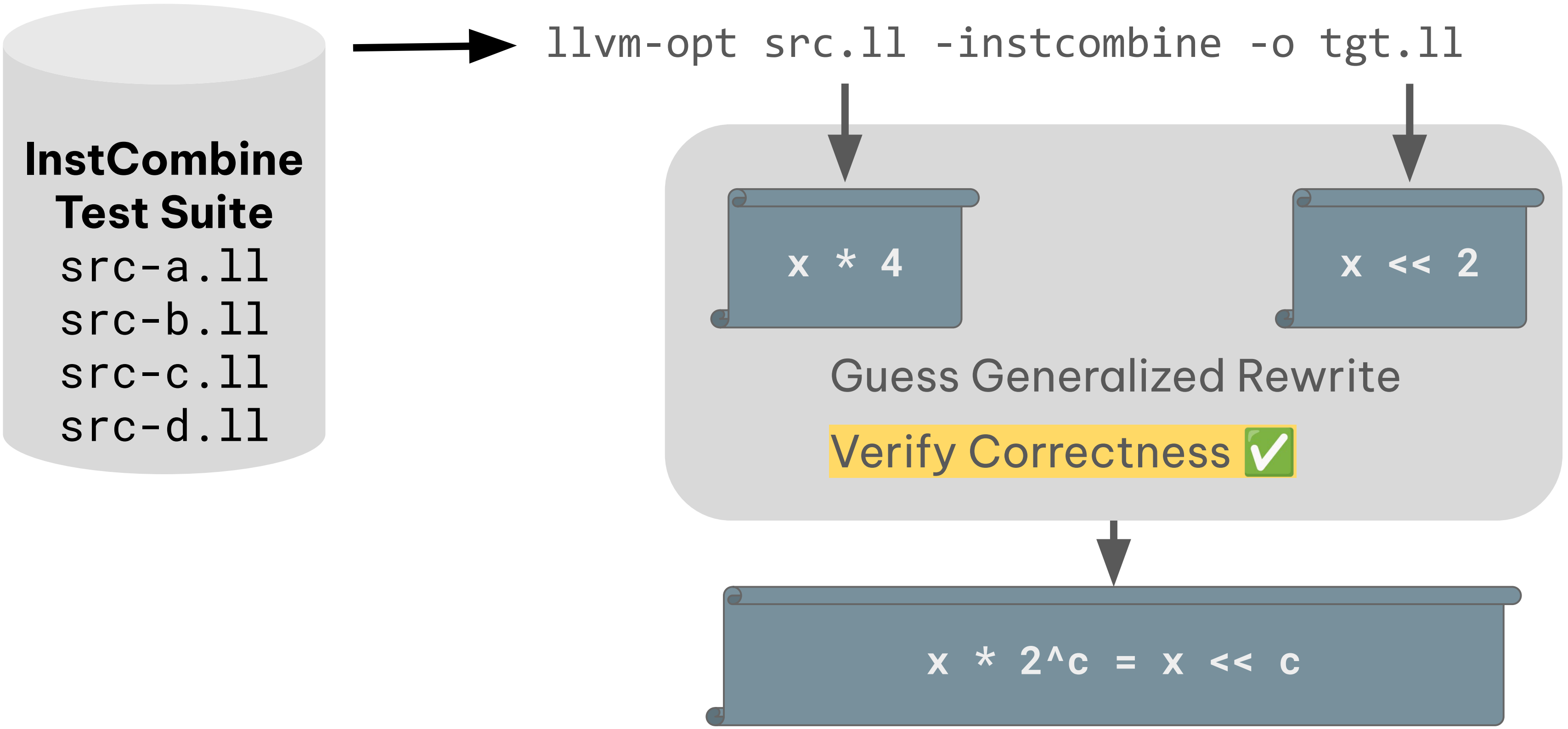
```
Transformation doesn't verify!

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)
```

LLVM Has "Verify" Already: Alive!

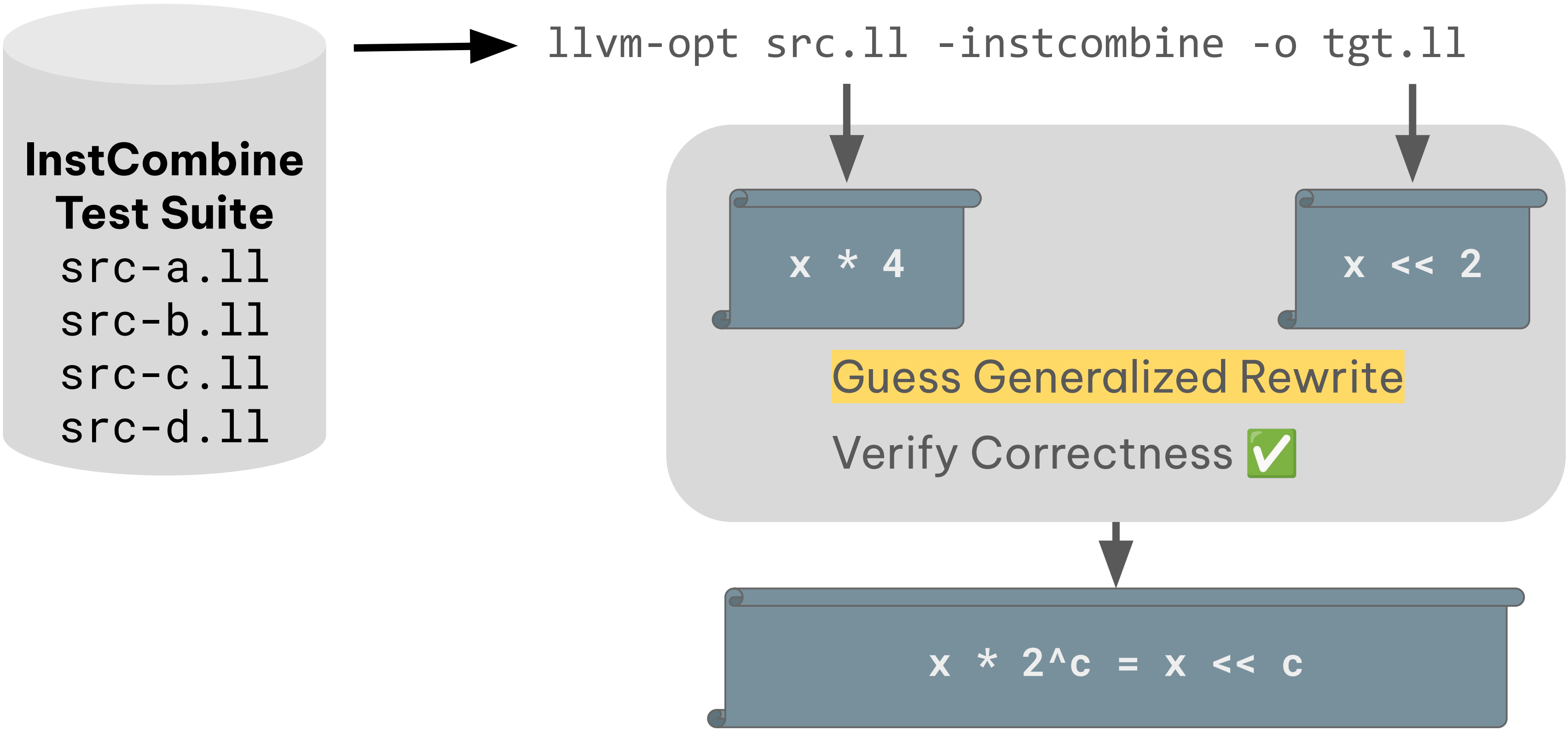
```
define i32 @src(i32) {
  %r = udiv i32 %0, 1
  ret i32 %r
}

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

```
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)
```

Idealized Pipeline For Extracting Declarative Rewrites



Idealized Pipeline For Extracting Declarative Rewrites



How To Guess Good Width Generic Rewrites?

```
(x y : BV 3), x + y = y + x
(x y : BV w), x + y = y + x
```

🧞 "Alive, is this rewrite true?"

Problem 1: Constant Generalization

```
(x : BV 5), (x << 4) >> 4 = x & 1
(x : BV w), (x << (w - 1)) >> (w - 1)) = x & 1
```

🤔 How to come up with (w - 1)?

Problem 2: Precondition Generation

```
(x : BV 3), (x.zext 4).sext 7 = x.zext 7
(x : BV w3), w3 < w4 <= w7 =>
  (x.zext w4).xsezt w7 = x.sext w7
```

🤔 How to come up with (w3 < w4 <= w7)?

Hydra: Generalizing Peephole Optimizations with Program Synthesis

MANASIJ MUKHERJEE, University of Utah, USA
JOHN REGEHR, University of Utah, USA

Optimizing compilers rely on peephole optimizations to simplify combinations of instructions and remove redundant instructions. Typically, a new peephole optimization is added when a compiler developer notices an optimization opportunity—a collection of dependent instructions that can be improved—and manually derives a more general rewrite rule that optimizes not only the original code, but also other, similar collections of instructions. In this paper, we present Hydra, a tool that automates the process of generalizing peephole optimizations using a collection of techniques centered on program synthesis. One of the most important problems we have solved is finding a version of each optimization that is independent of the bitwidths of the optimization's inputs (when this version exists). We show that Hydra can generalize 75% of the ungeneralized missed peephole optimizations that LLVM developers have posted to the LLVM project's issue tracker. All of Hydra's generalized peephole optimizations have been formally verified, and furthermore we can automatically turn them into C++ code that is suitable for inclusion in an LLVM pass.

CCS Concepts: • **Software and its engineering** → **Translator writing systems and compiler generators**; *Source code generation.*

Additional Key Words and Phrases: program synthesis, generalization, superoptimization, llvm, alive2, souper, hydra, peephole optimization

ACM Reference Format:
Manasij Mukherjee and John Regehr. 2024. Hydra: Generalizing Peephole Optimizations with Program Synthesis. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 120 (April 2024), 29 pages. https://doi.org/10.1145/3649837

Hydra Generates Potentially Likely Candidates 😊

Hydra: Generalizing Peephole Optimizations with Program Synthesis

MANASIJ MUKHERJEE, University of Utah, USA
JOHN REGEHR, University of Utah, USA

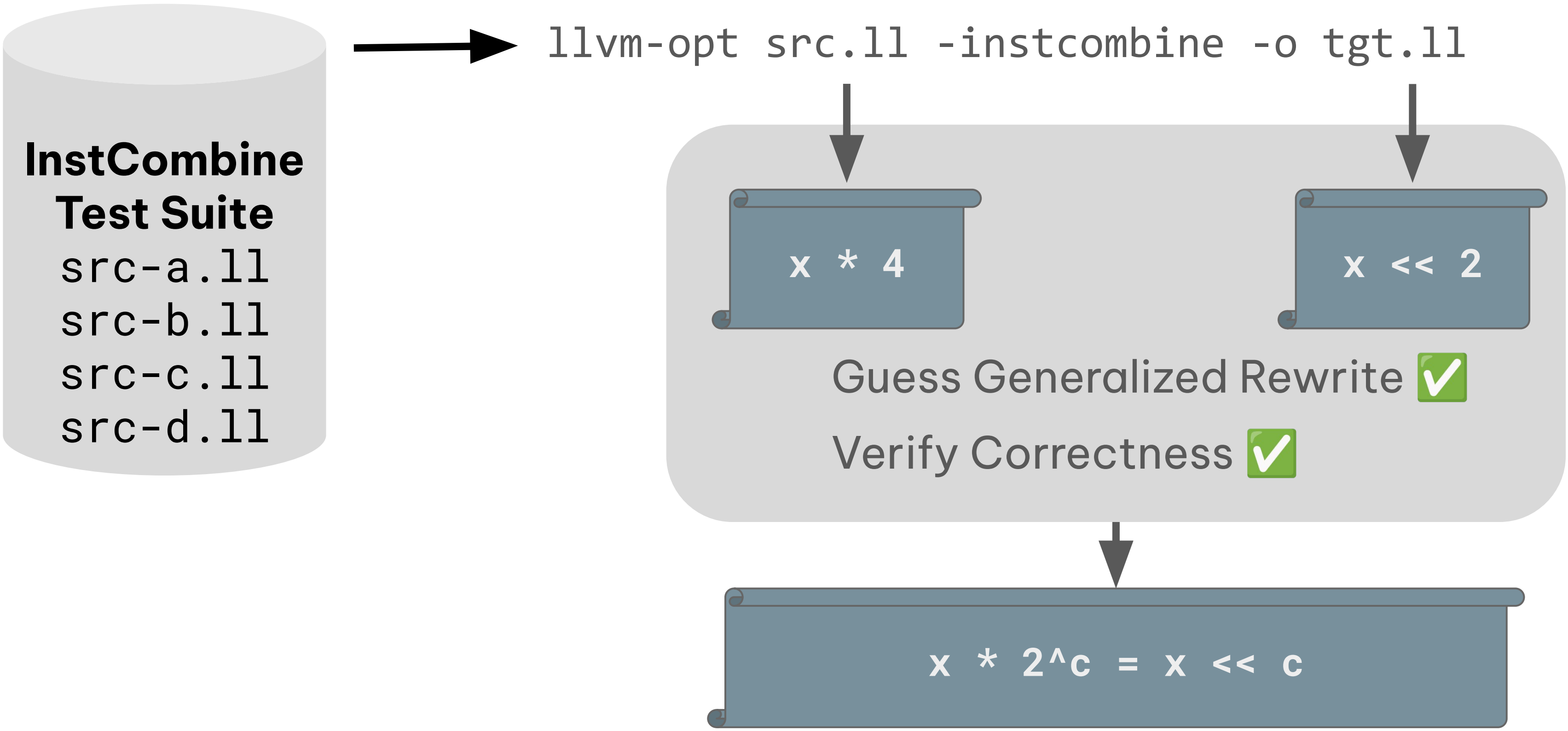
Optimizing compilers rely on peephole optimizations to simplify combinations of instructions and remove redundant instructions. Typically, a new peephole optimization is added when a compiler developer notices an optimization opportunity—a collection of dependent instructions that can be improved—and manually derives a more general rewrite rule that optimizes not only the original code, but also other, similar collections of instructions. In this paper, we present Hydra, a tool that automates the process of generalizing peephole optimizations using a collection of techniques centered on program synthesis. One of the most important problems we have solved is finding a version of each optimization that is independent of the bitwidths of the optimization's inputs (when this version exists). We show that Hydra can generalize 75% of the ungeneralized missed peephole optimizations that LLVM developers have posted to the LLVM project's issue tracker. All of Hydra's generalized peephole optimizations have been formally verified, and furthermore we can automatically turn them into C++ code that is suitable for inclusion in an LLVM pass.

CCS Concepts: • **Software and its engineering → Translator writing systems and compiler generators**; *Source code generation.*

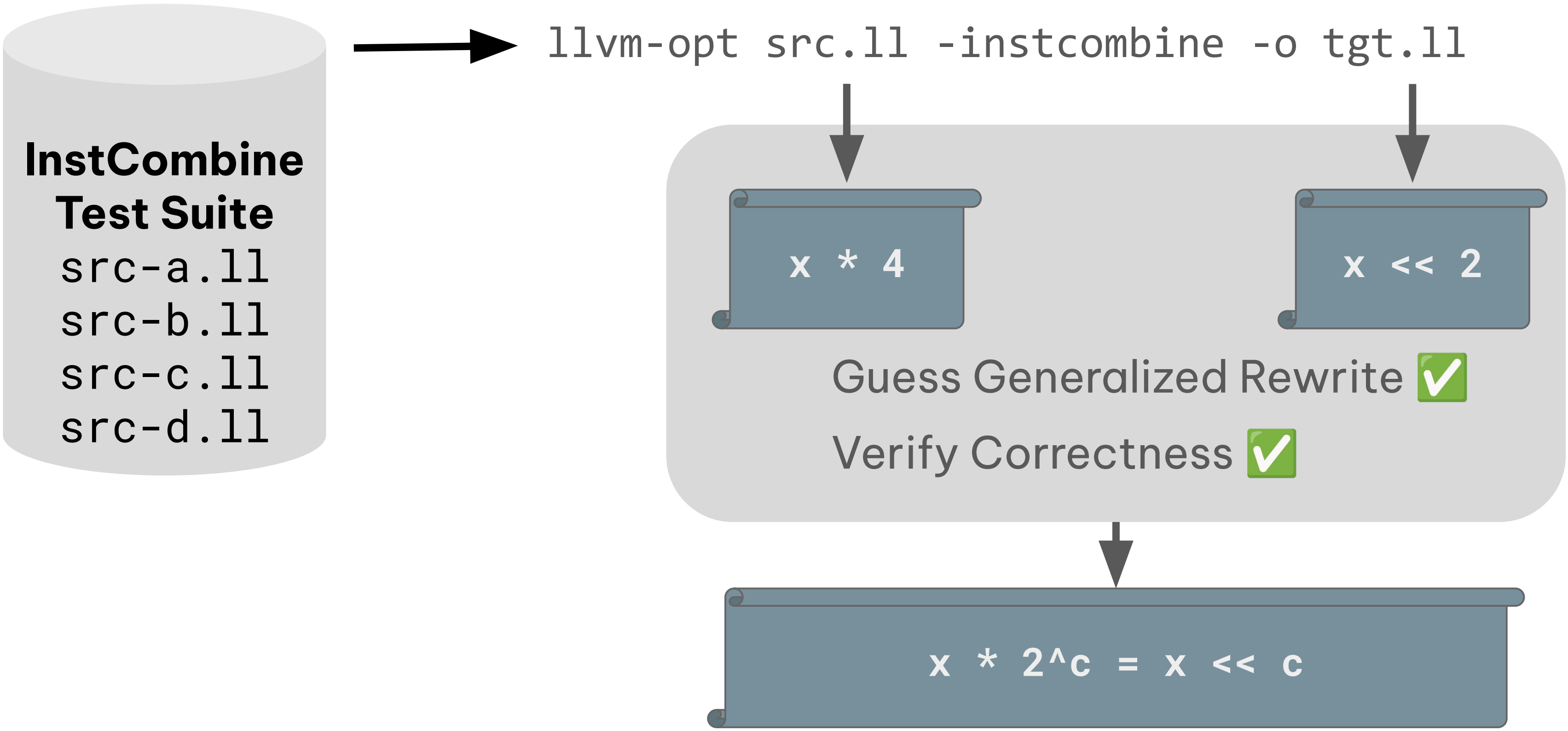
Additional Key Words and Phrases: program synthesis, generalization, superoptimization, llvm, alive2, souper, hydra, peephole optimization

ACM Reference Format:
Manasij Mukherjee and John Regehr. 2024. Hydra: Generalizing Peephole Optimizations with Program Synthesis. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 120 (April 2024), 29 pages. https://doi.org/10.1145/3649837

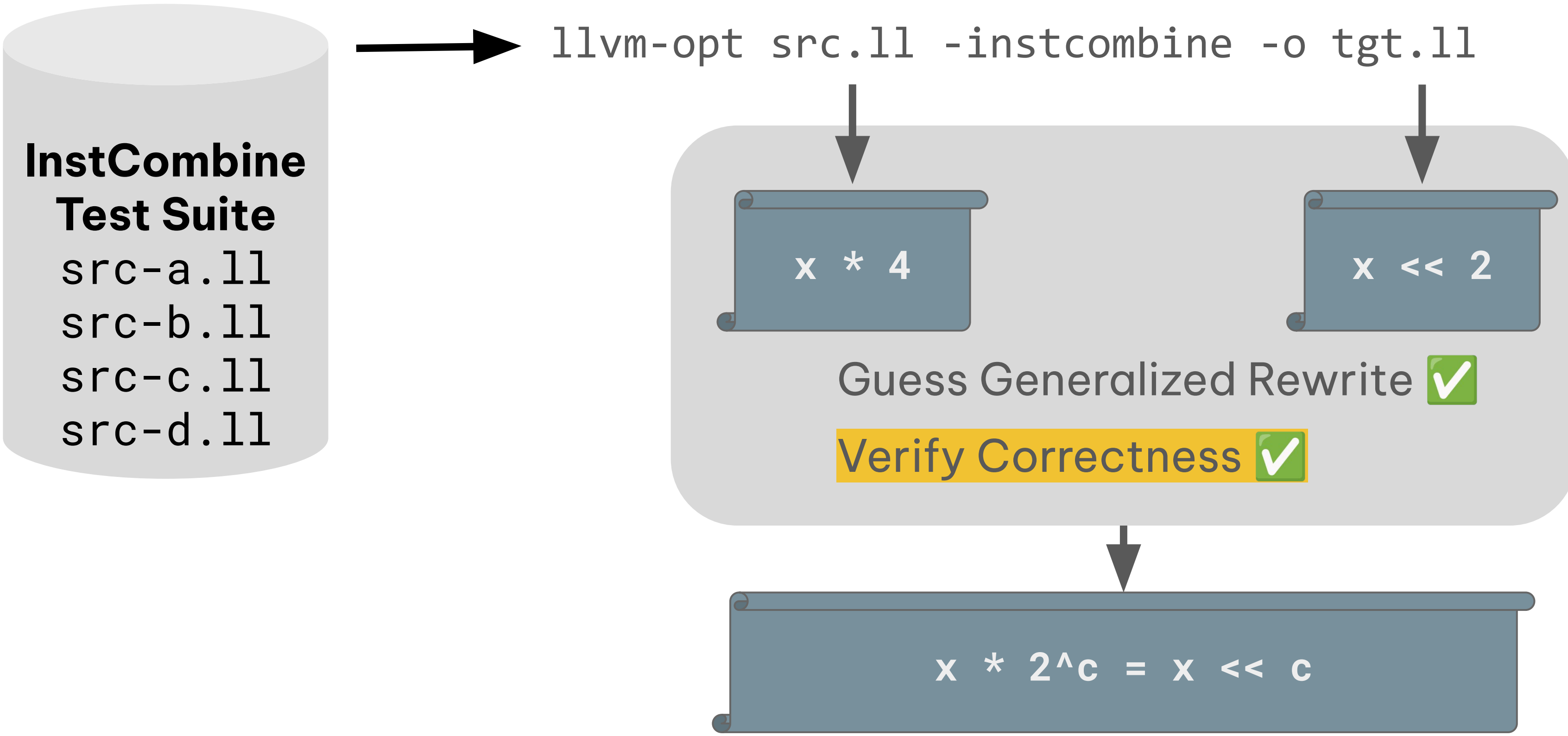
Idealized Pipeline For Extracting Declarative Rewrites



Did Sid Think This Is A Lightning Talk By Accident? 😨



Idealized Pipeline For Extracting Declarative Rewrites



Verifing The Correctness Of Candidates

`(x y : BV w), x + y = y + x`

"Alive, is this rewrite true?"

→

`(x y : BV 1), x + y = y + x`

"Bitwuzla, is this rewrite true?"

`∀ (x y : BV 1), x + y = y + x`

fixed size bitvectors — theory problem to decide true/false

About Bitwuzla

Bitwuzla is a Satisfiability Modulo Theories (SMT) solver for the theories of fixed-size bit-vectors, floating-point arithmetic, arrays and uninterpreted functions and their combinations.

Verifing The Correctness Of Candidates

`(x y : BV w), x + y = y + x`

"Alive, is this rewrite true?"

→

`(x y : BV 1), x + y = y + x`

"Bitwuzla, is this rewrite true?"

`(x y : BV 2), x + y = y + x`

"Bitwuzla, is this rewrite true?"

`(x y : BV 3), x + y = y + x`

"Bitwuzla, is this rewrite true?"

...

All widths from 1 to 64

Verifing The Correctness Of Candidates Is Exponential!

```
(x : BV w3), w3 < w4 <= w7 =>
  (x.zext w4).xsezt w7 = x.sext w7
```

"Alive, is this rewrite true?"

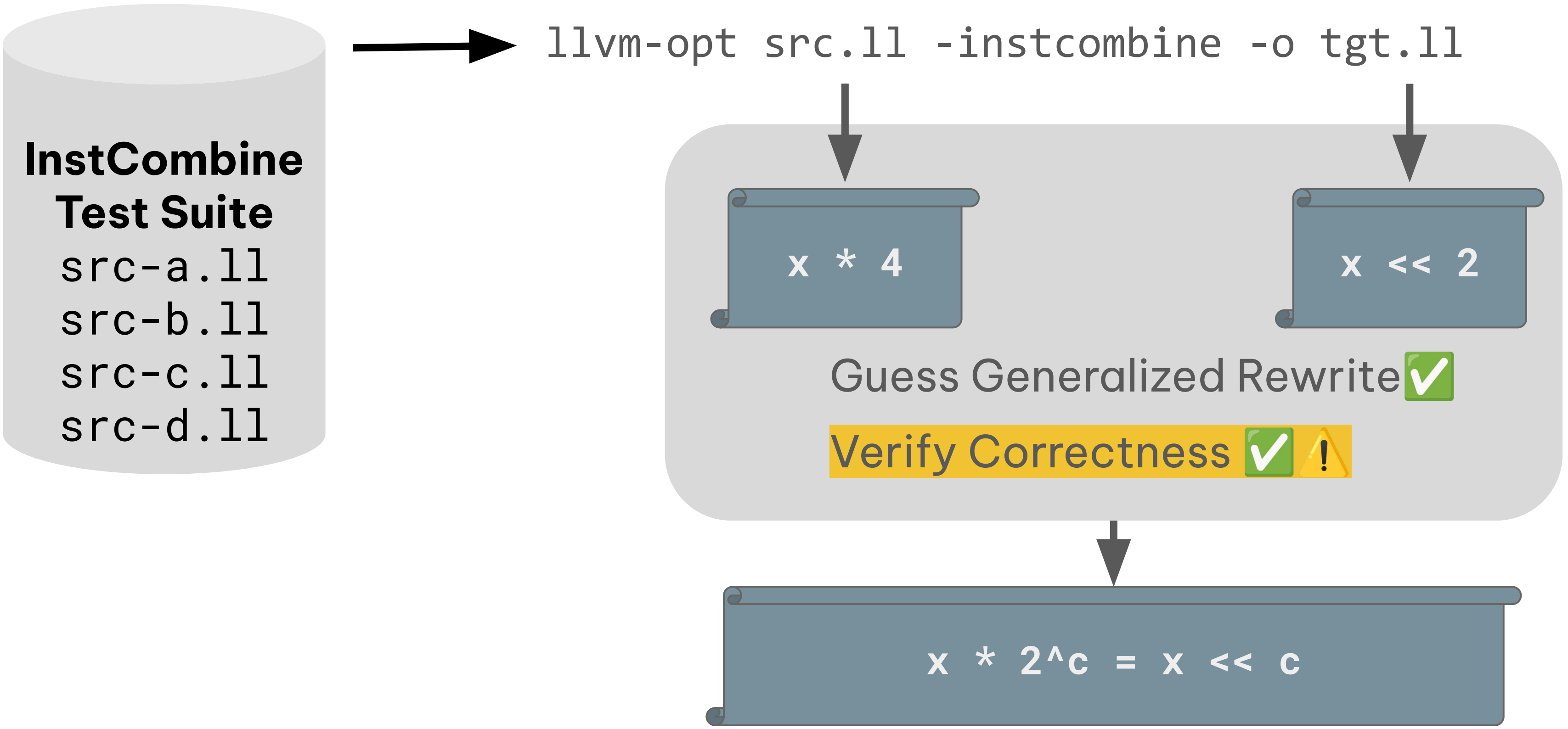
→

w3 : 64 choices
w4 : 64 choices
w7 : 64 choices

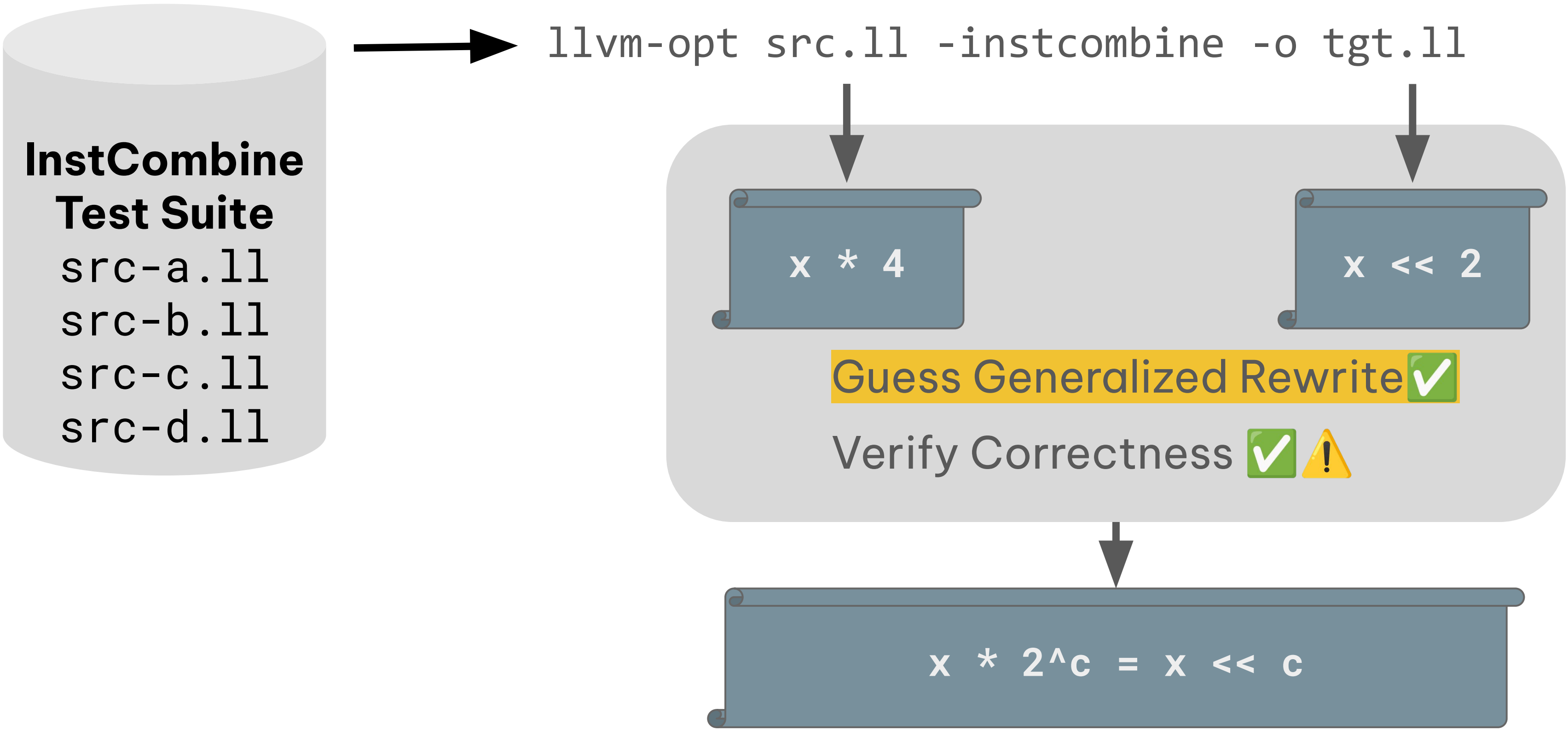
262144 combinations 😱
Alive needs **18.5 hours** to prove this!

We Need An Algorithm That's Much Better Than Exhaustive Enumeration!

Idealized Pipeline For Extracting Declarative Rewrites



Idealized Pipeline For Extracting Declarative Rewrites



Hydra: Generalizing Peephole Optimizations with Program Synthesis

MANASIJ MUKHERJEE, University of Utah, USA
JOHN REGEHR, University of Utah, USA

Optimizing compilers rely on peephole optimizations to simplify combinations of instructions and remove redundant instructions. Typically, a new peephole optimization is added when a compiler developer notices an optimization opportunity—a collection of dependent instructions that can be improved—and manually derives a more general rewrite rule that optimizes not only the original code, but also other, similar collections of instructions. In this paper, we present Hydra, a tool that automates the process of generalizing peephole optimizations using a collection of techniques centered on program synthesis. One of the most important problems we have solved is finding a version of each optimization that is independent of the bitwidths of the optimization's inputs (when this version exists). We show that Hydra can generalize 75% of the ungeneralized missed peephole optimizations that LLVM developers have posted to the LLVM project's issue tracker. All of Hydra's generalized peephole optimizations have been formally verified, and furthermore we can automatically turn them into C++ code that is suitable for inclusion in an LLVM pass.

CCS Concepts: • **Software and its engineering → Translator writing systems and compiler generators**; *Source code generation.*

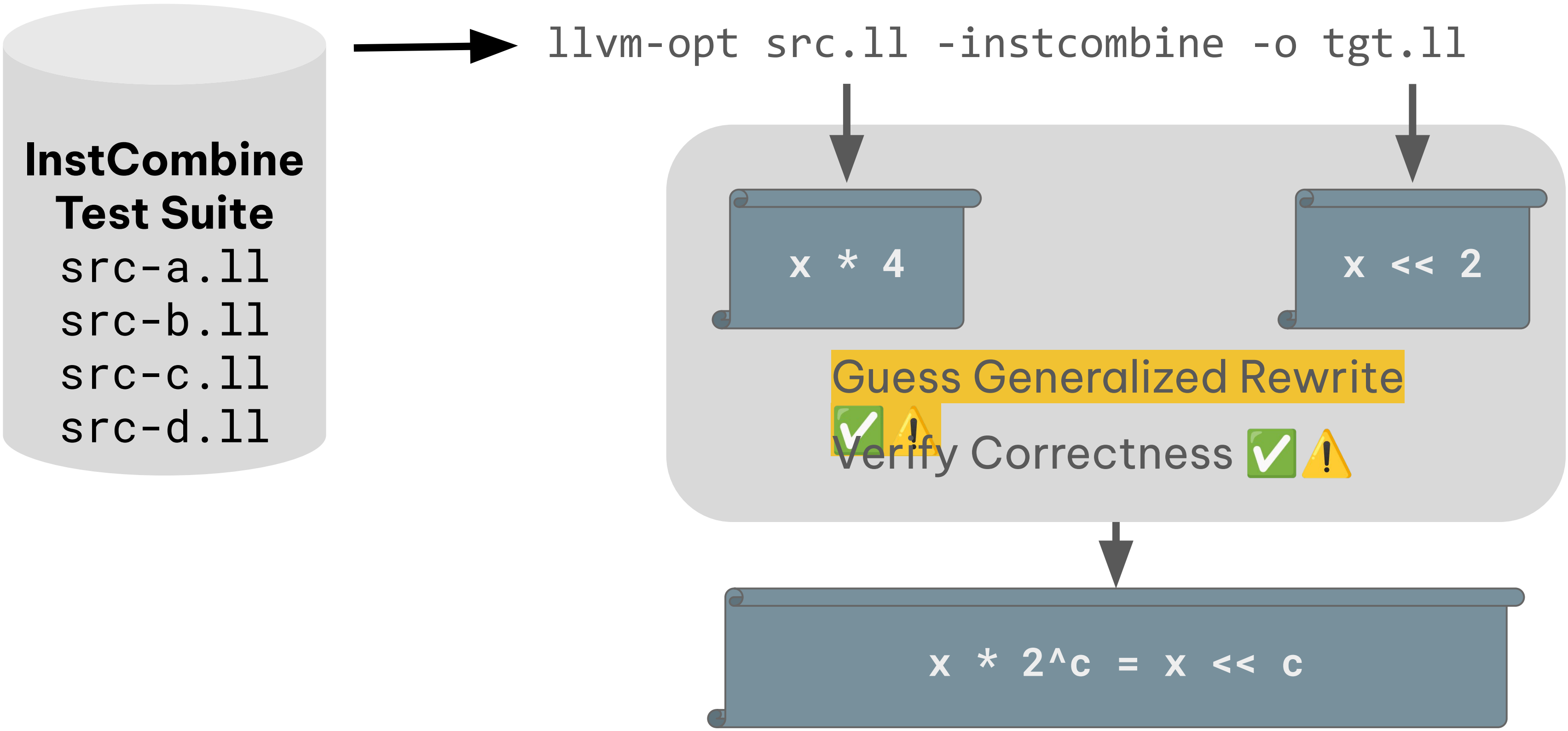
uper,

Cannot generalize on multiple widths

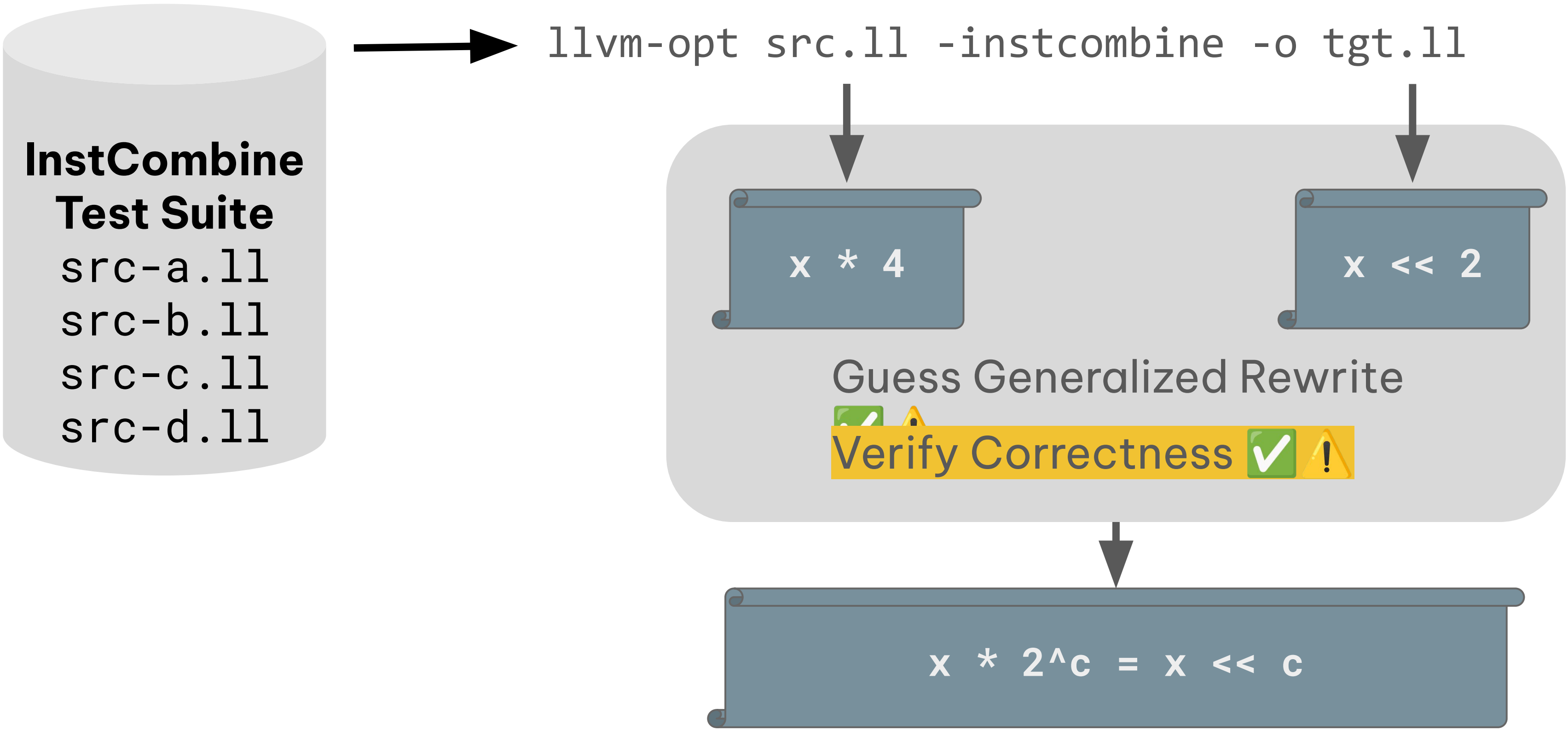
Can we have fewer human-tuned heuristics?

ram
145/
3649837

Idealized Pipeline For Extracting Declarative Rewrites



Idealized Pipeline For Extracting Declarative Rewrites



Verifing The Correctness Of Candidates

`(x y : BV w), x + y = y + x`

"Alive, is this rewrite true?"

`(x y : BV 1), x + y = y + x`

"Bitwuzla, is this rewrite true?"

`(x y : BV 2), x + y = y + x`

"Bitwuzla, is this rewrite true?"

`(x y : BV 3), x + y = y + x`

"Bitwuzla, is this rewrite true?"

...

All widths from 1 to 64

Toward A Faster Algorithm

`(x y : BV w), x + y = y + x`

`(x y : BV 1), x + y = y + x`

"Bitwuzla, is this rewrite true?"

`x y : BV 2), x + y = y + x`

"Bitwuzla, is this rewrite true?"

`x y : BV 3), x + y = y + x`

"Bitwuzla, is this rewrite true?"

...

All widths from 1 to 64

Toward A Faster Algorithm: Can We Eliminate Looping?

O Aristotle, can we have bitwuzla loop on the widths?

y + x

"Bitwuzla, is this rewrite true?"

x y : BV 2), x + y = y + x

"Bitwuzla, is this rewrite true?"

x y : BV 3), x + y = y + x

"Bitwuzla, is this rewrite true?"

...

All widths from 1 to 64

Toward A Faster Algorithm: Can We Eliminate Looping?

But Plato, Bitwuzla only knows about bitvectors, not natural numbers!

y + x

"Bitwuzla, is this rewrite true?"

x y : BV 2), x + y = y + x

"Bitwuzla, is this rewrite true?"

x y : BV 3), x + y = y + x

"Bitwuzla, is this rewrite true?"

...

All widths from 1 to 64

Toward A Faster Algorithm: Can We Eliminate Looping?

Aha, but what is a natural number, if not a unsigned bitvector?

y + x

"Bitwuzla, is this rewrite true?"

x y : BV 2), x + y = y + x

"Bitwuzla, is this rewrite true?"

x y : BV 3), x + y = y + x

"Bitwuzla, is this rewrite true?"

...

All widths from 1 to 64

Toward A Faster Algorithm: Widths As Bitvectors

```
x               : BV 3    -   -   x2  x1  x0
w3mask          : BV 5    0   0    1   1   1   (w3mask := 1 <<< 3 - 1)
x'              : BV 5    *   *   x2  x1  x0

x' & w3mask     : BV 5    0   0   x2  x1  x0
```

Widths Like '3' Can Be Encoded By **Bitvectors** Like 'w3mask'!

Toward A Faster Algorithm: Widths As Bitvectors

`(x y : BV w), x + y = y + x` →

`(x y : BV 1), x + y = y + x`

"Bitwuzla, is this rewrite true?"

`(x y : BV 2), x + y = y + x`

"Bitwuzla, is this rewrite true?"

`(x y : BV 3), x + y = y + x`

"Bitwuzla, is this rewrite true?"

...

upto width 64

Toward A Faster Algorithm: Widths As Bitvectors

`(x y : BV w), x + y = y + x`

`(x y : BV 1), x + y = y + x`

"Bitwuzla, is this rewrite true?"

`(x y : BV 2), x + y = y + x`

"Bitwuzla, is this rewrite true?"

`(x y : BV 3), x + y = y + x`

"Bitwuzla, is this rewrite true?"

...

upto width 64

`(           : BV 65)`

Toward A Faster Algorithm: Widths As Bitvectors

`(x y : BV w), x + y = y + x`

`(x y : BV 1), x + y = y + x`

"Bitwuzla, is this rewrite true?"

`(x y : BV 2), x + y = y + x`

"Bitwuzla, is this rewrite true?"

`(x y : BV 3), x + y = y + x`

"Bitwuzla, is this rewrite true?"

...

upto width 64

`(      w' : BV 65), w' <= 64 ->`

Toward A Faster Algorithm: Widths As Bitvectors

```
(x y : BV w), x + y = y + x
```

```
(x' w' : BV 65), w' <= 64 ->
  let x := x'
```

```
(x y : BV 1), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

```
(x y : BV 2), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

```
(x y : BV 3), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

...

upto width 64

Toward A Faster Algorithm: Widths As Bitvectors

```
(x y : BV w), x + y = y + x
```

```
(x'    w' : BV 65), w' <= 64 ->
  let x := x' & (1 <<< w' - 1)
```

```
(x y : BV 1), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

```
(x y : BV 2), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

```
(x y : BV 3), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

...

upto width 64

Toward A Faster Algorithm: Widths As Bitvectors

```
(x y : BV w), x + y = y + x
```

x' masked with w **is** x!

```
(x'    w' : BV 65), w' <= 64 ->
  let x := x' & (1 <<< w' - 1)
```

```
(x y : BV 1), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

```
(x y : BV 2), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

```
(x y : BV 3), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

...

upto width 64

Toward A Faster Algorithm: Widths As Bitvectors

```
(x y : BV w), x + y = y + x
```

x' masked with w **is** x!
y' masked with w **is** y!

```
(x' y' w' : BV 65), w' <= 64 ->
  let x := x' & (1 <<< w' - 1)
  let y := y' & (1 <<< w' - 1)
```

```
(x y : BV 1), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

```
(x y : BV 2), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

```
(x y : BV 3), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

...
upto width 64

Toward A Faster Algorithm: Widths As Bitvectors

```
(x y : BV w), x + y = y + x
```

```
(x y : BV 1), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

```
(x y : BV 2), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

```
(x y : BV 3), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

...

upto width 64

x' masked with w **is** x!
y' masked with w **is** y!

```
(x' y' w' : BV 65), w' <= 64 ->
  let x := x' & (1 <<< w' - 1)
  let y := y' & (1 <<< w' - 1)
  x + y = y + x
```

Toward A Faster Algorithm: Widths As Bitvectors

```
(x y : BV w), x + y = y + x
```

```
(x y : BV 1), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

```
(x y : BV 2), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

```
(x y : BV 3), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

...

upto width 64

x' masked with w **is** x!
y' masked with w **is** y!

```
(x' y' w' : BV 65), w' <= 64 ->
  let x := x' & (1 <<< w' - 1)
  let y := y' & (1 <<< w' - 1)
  x + y = y + x
```

"Bitwuzla, is this rewrite true?"

Upshot: One Solver Call For All Widths Simultaneously

```
(x y : BV w), x + y = y + x
```

```
(x y : BV 1), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

```
(x y : BV 2), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

```
(x y : BV 3), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

...
upto width 64

x' masked with w **is** x!
y' masked with w **is** y!

```
(x' y' w' : BV 65), w' <= 64 ->
  let x := x' & (1 <<< w' - 1)
  let y := y' & (1 <<< w' - 1)
  x + y = y + x
```

"Bitwuzla, is this rewrite true?"

💡 One call for **all** widths, **simultaneously**

⚡ Bitwuzla is fast for one call!

Upshot: One Solver Call For All Widths Simultaneously

```
(x y : BV w), x + y = y + x
```

→

```
(x y : BV 1), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

```
(x y : BV 2), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

```
(x y : BV 3), x + y = y + x
```

"Bitwuzla, is this rewrite true?"

...

upto width 64 **18.5h**

↓

x' masked with w **is** x!
y' masked with w **is** y!

```
(x' y' w' : BV 65), w' <= 64 ->
  let x := x' & (1 <<< w' - 1)
  let y := y' & (1 <<< w' - 1)
  x + y = y + x
```

1sec "Bitwuzla, is this rewrite true?"

💡 One call for **all** widths, **simultaneously**

⚡ Bitwuzla is fast for one call!

Everything Can Be Translated From Many Widths to One

Multi-width PBV	**Mono-width PBV** $[\![\cdot]\!]$
Variable Declarations (preconditions asserted)	
$\mathsf{Var}(x_w) : \mathbb{N}$	$(x_w : \mathsf{BitVec}\ o)$
$\mathsf{Var}(x, w) : \mathsf{BitVec}\ e$	$(x : \mathsf{BitVec}\ o)$, $\quad$ assert $(x \mathbin{\&} [\![e]\!]^{\mathsf{mask}} = x)$
Width Expressions (widths stored directly as bitvectors)	
$\mathsf{Var}(x_w)$	x_w
$\mathsf{const}(c)$	c
$\min(u, v)$	$\mathsf{ite}([\![u]\!] <_u [\![v]\!],\ [\![u]\!],\ [\![v]\!])$
$\mathsf{wadd}(u, v)$	$[\![u]\!] + [\![v]\!]$
Bitvector Expressions (mask result to width via $[\![w]\!]^{\mathsf{mask}} = (1 \ll [\![w]\!]) - 1$)	
$\mathsf{Var}(a, w)$	a (pre-masked by variable declaration)
$\mathsf{Const}(x, w)$	$x \mathbin{\&} [\![w]\!]^{\mathsf{mask}}$
$\mathsf{add}(a, b, w)$	$([\![a]\!] + [\![b]\!]) \mathbin{\&} [\![w]\!]^{\mathsf{mask}}$
$\mathsf{nand}(a, b, w)$	$\sim([\![a]\!] \mathbin{\&} [\![b]\!]) \mathbin{\&} [\![w]\!]^{\mathsf{mask}}$
$\mathsf{zext}(a, v, w)$	$[\![a]\!] \mathbin{\&} [\![w]\!]^{\mathsf{mask}}$
$\mathsf{sext}(a, v, w)$	**let** $s = \mathsf{ite}(\mathsf{msb}_v([\![a]\!]),\ \sim 0,\ 0)$ (sign-extend fill) **let** $ext = s \mathbin{\&} \sim [\![v]\!]^{\mathsf{mask}}$ (bits above width v) $([\![a]\!] \mathbin{\Vert} ext) \mathbin{\&} [\![w]\!]^{\mathsf{mask}}$
$\mathsf{append}(a, v, b, w)$	$([\![a]\!] \ll [\![w]\!]) \mathbin{\Vert} [\![b]\!]$
Predicate Expressions	
$\mathsf{eq}(a, b, w)$	$[\![a]\!] = [\![b]\!]$ (inputs pre-masked)
$\mathsf{ult}(a, b, w)$	$[\![a]\!] <_u [\![b]\!]$ (inputs pre-masked)
$\mathsf{slt}(a, b, w)$	$\mathsf{ite}(\mathsf{msb}_w([\![a]\!]) = \mathsf{msb}_w([\![b]\!]),$ $\mathsf{msb}_w([\![a]\!]) \oplus \mathsf{ult}([\![a]\!], [\![b]\!]),$ $\mathsf{msb}_w([\![a]\!]))$
$\mathsf{and}(p, q)$	$[\![p]\!] \land [\![q]\!]$
$\mathsf{or}(p, q)$	$[\![p]\!] \lor [\![q]\!]$
$\mathsf{weq}(u, v)$	$[\![u]\!] = [\![v]\!]$
$\mathsf{wlt}(u, v)$	$[\![u]\!] <_u [\![v]\!]$

where $\mathsf{msb}_w(a) \triangleq (a \mathbin{\&} ((1 \ll [\![w]\!]) \gg 1)) \neq 0$.

The PBV_1 encoding is equisatisfiable with the PBV_n encoding:

Proof!

Theorem 4.1. *If p is a PBV_n formula, then $\vDash p$ iff $\vDash [\![p]\!]$.*

Proof. *Left to right direction.* Let ρ be a model of p. We begin by constructing a model $\bar{\rho}$ for $[\![p]\!]$. In particular, ρ associates each width variable $x_w^1, \ldots, x_w^n$ with a natural number $\rho(x_w^1), \ldots, \rho(x_w^n)$. We define $\bar{\rho}$ to map the unique width variable o of $[\![p]\!]$ to $\max\{\rho(x_w^1), \ldots, \rho(x_w^n)\}$, and map the bitvector variables of $[\![p]\!]$ as follows:

$$\begin{aligned} \bar{\rho}(x_w^i) &= \rho(x_w^i) \text{ as a bitvector of size } \bar{\rho}(o) \\ \bar{\rho}(x_i) &= \rho(x_i) \text{ zero-extended to size } \bar{\rho}(o). \end{aligned}$$

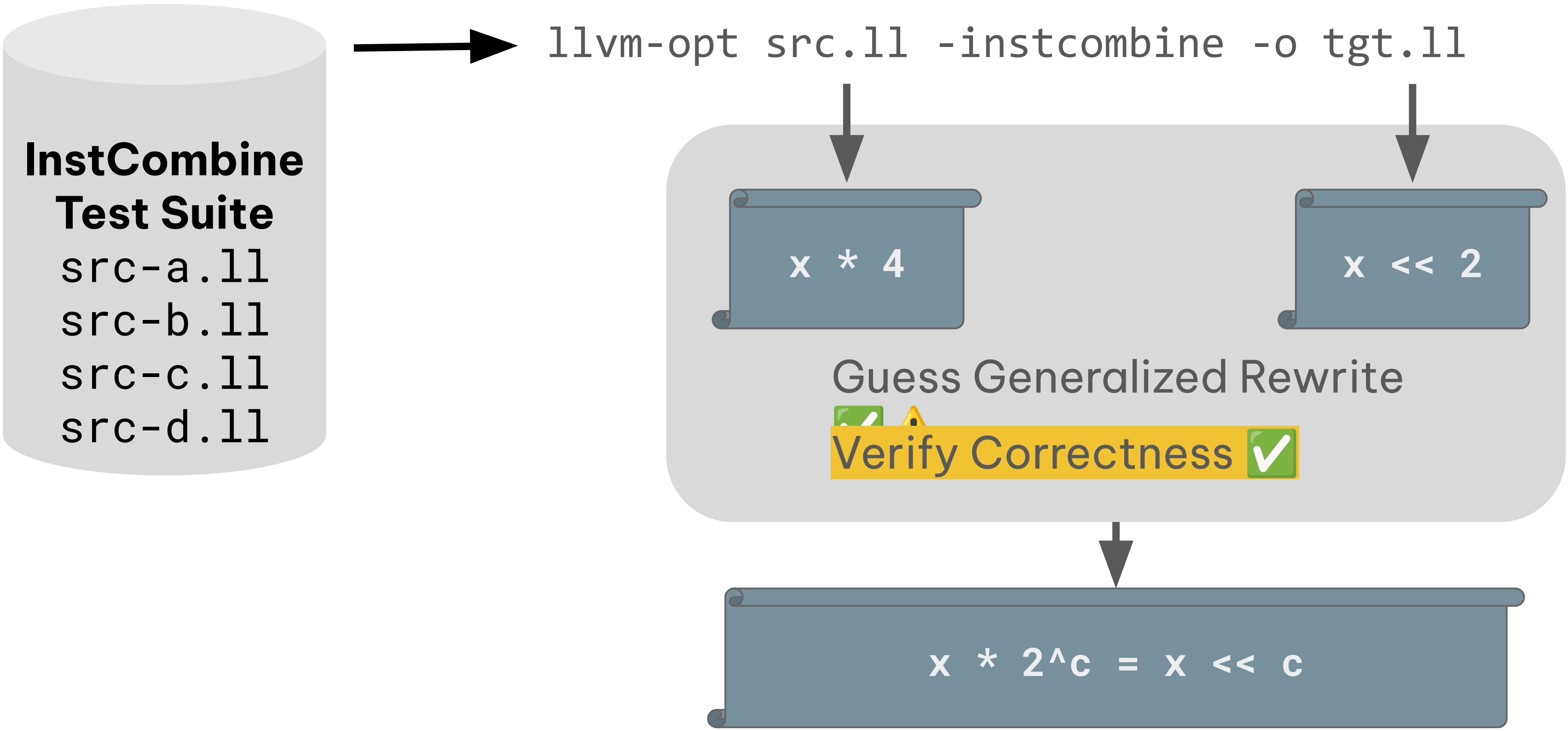
We check that all the variables x_i of width w satisfy the condition $\rho(x_i) \mathbin{\&} (2^{\rho(w)} - 1) = \rho(x_i)$ since they are obtained via zero extension. We then prove that for a well-typed width expression w, $\rho(w)$ is equal to the bitvector $\bar{\rho}([\![w]\!])$ interpreted as a natural number. This is a straightforward induction on the typing derivation.

The crux of the proof is to relate the bitvector expressions: for any bitvector expression a of

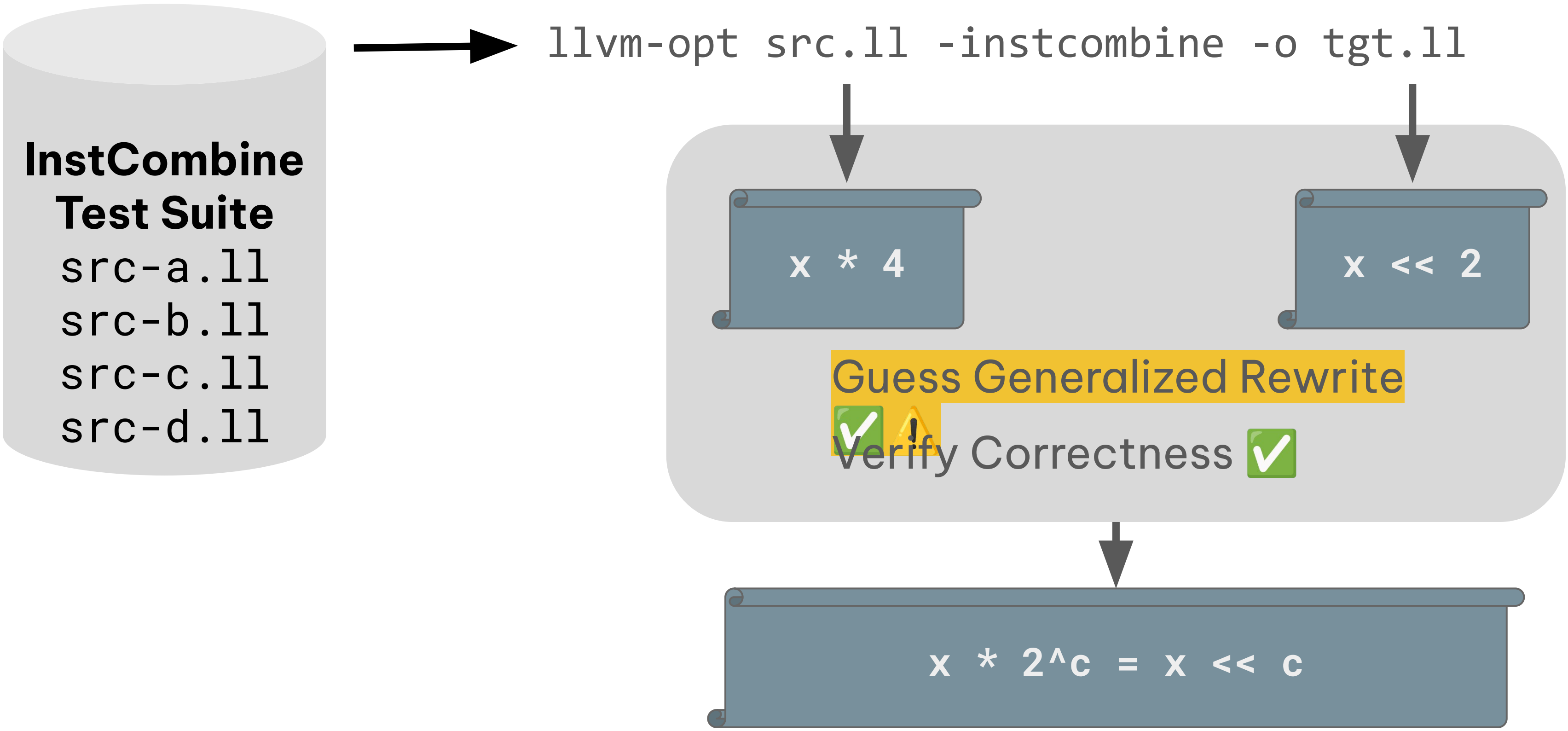
💡 One call for **all** widths, **simultaneously**

⚡ Bitwuzla is fast for one call!

Idealized Pipeline For Extracting Declarative Rewrites



Idealized Pipeline For Extracting Declarative Rewrites



Guess Generalization

Guess Generalization By Prayer To LLM

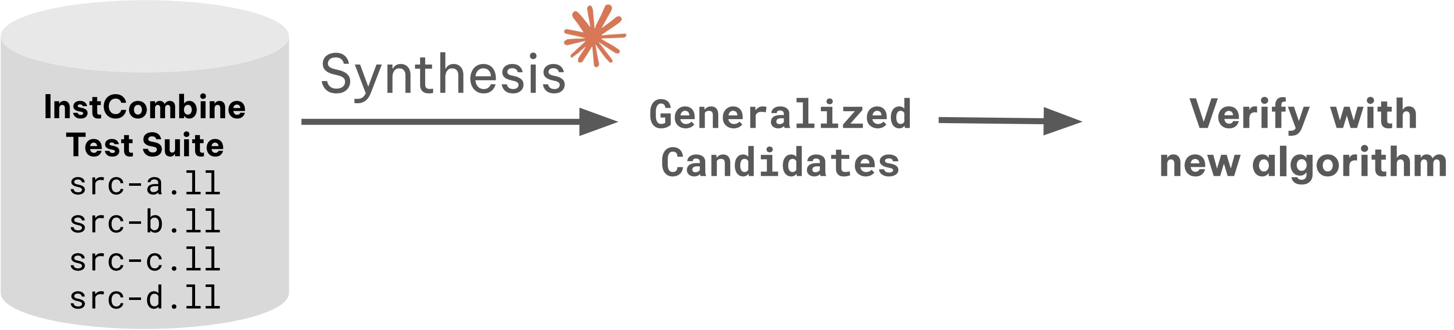
John Regehr 23:16

yeah, program synthesis as a research area is basically done-- our enumerative work is totally obsolete except in narrow circumstances

I think the verification side of synthesis is where we should focus all of our efforts now

✅ 1

LLM-Driven Synthesis, Algorithmic Verification 💖



Start with LLVM rewrite problems from lean-mlir (**1553** problems of the form LHS -> RHS)

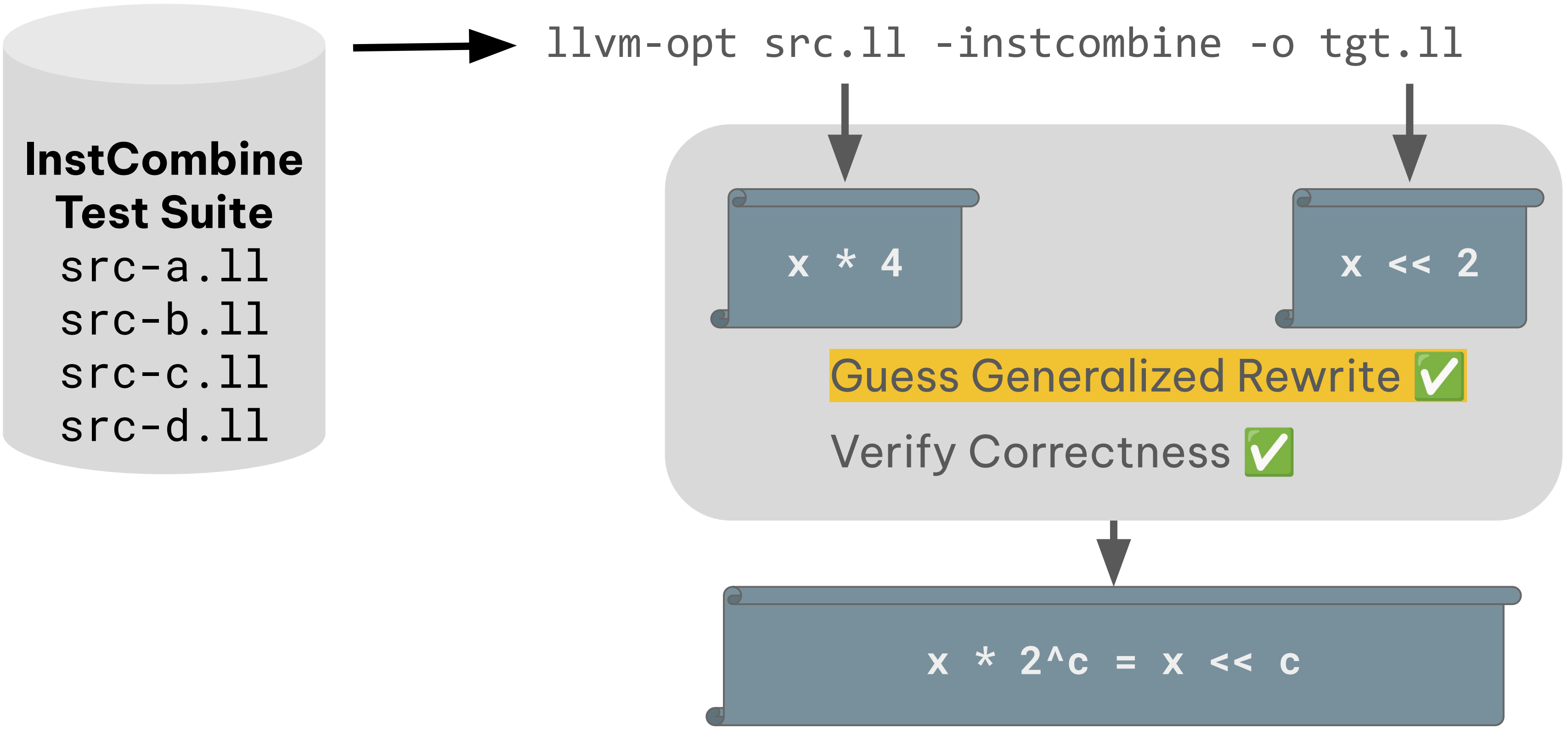
Ask the agentic harness (claude code, Opus 4.6) to sort these by "clean rewrites"

Ask the agentic harness to generalize these clean rewrites
+ tool use of our verifier given to agent, agent uses tool repeatedly
+ (⚡ fast => no bottleneck here, catches hallucinations quickly)

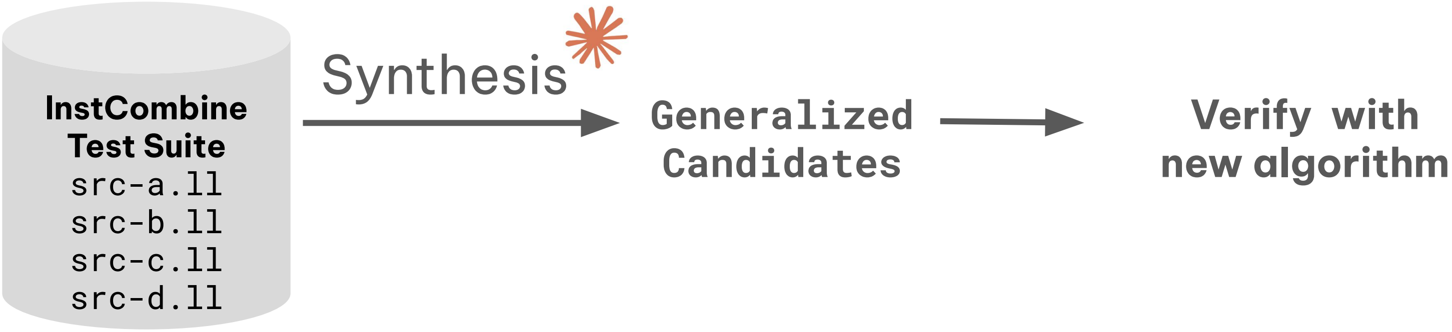
Produced 309 "candidate rewrites" after burning 1 week of CC subscription in 1 hr 😊

Check that 309 candidates are legal, giving **309** verified rewrites

Idealized Pipeline For Extracting Declarative Rewrites



What More Do We Need To Scale This Up?



1) Extracted Rewrites are in declarative SMT-LIB format.
How to teach LLVM to consume and execute these as part of the rewriter?

2) Need to run this pipeline at scale, I ran out of credits for thinking.
Given enough thinking tokens, maybe everything will be generalized?

3) Support for nsw/nuw, KnownBits, floating point, ...

LLM-Driven Synthesis, Algorithmic Verification 💖

New Algorithms To Move Toward Declarative Compilers!

InstCombine Test Suite
src-a.ll
src-b.ll
src-c.ll
src-d.ll

Synthesis → **Generalized Candidates** → **Verify with new algorithm**

```
(x y : BV w), x + y = y + x
```

→

```
(x' y' w' : BV 65), w' <= 64 ->
  let x := x' & (1 <<< w' - 1)
  let y := y' & (1 <<< w' - 1)
  x + y = y + x
```

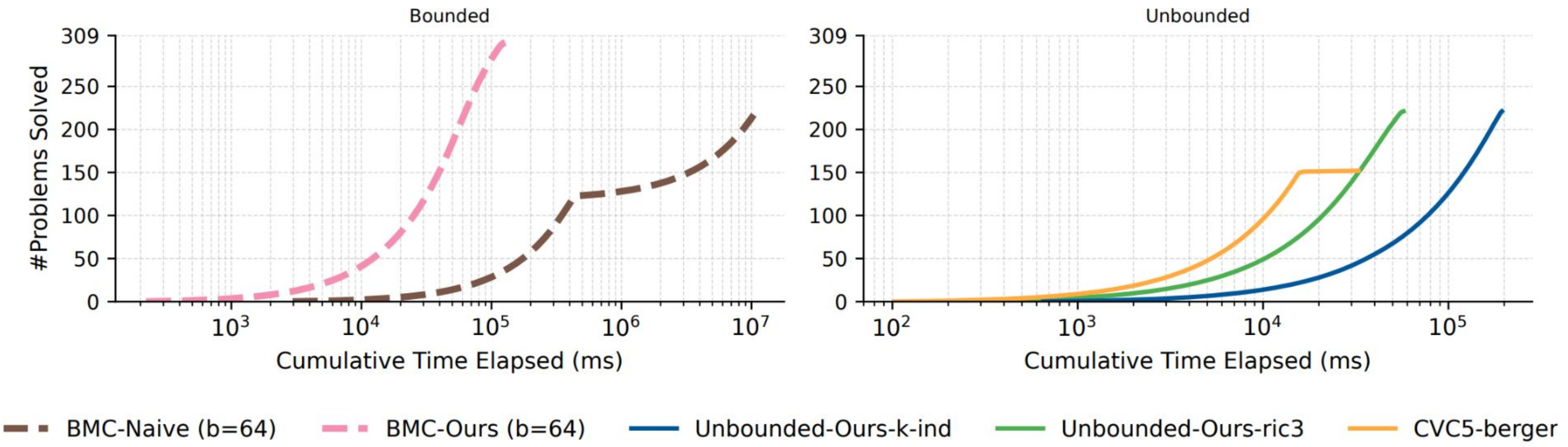
Blase - Blaster for StrEams

github.com/opencompl/lean-mlir/tree/main/Blase

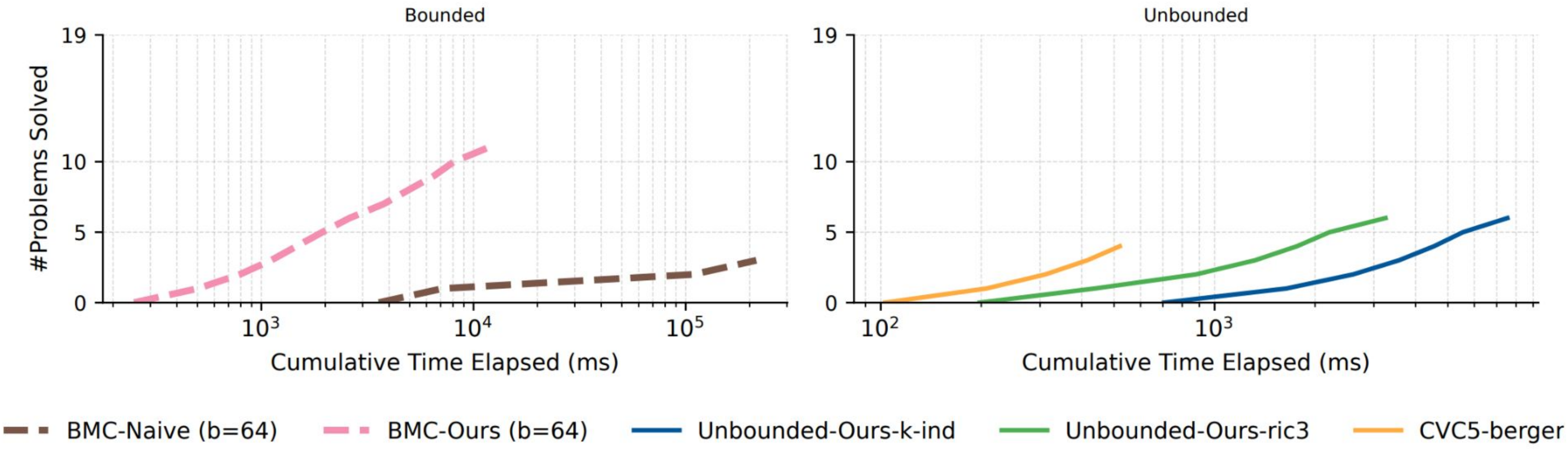
A decision procedure that is sound and complete for parametric bitvector expressions with linear arithmetic, bitwise operationrs, zeroExtend, signExtend, and bitwidth constraints.

To use the bleeding edge of `Blase` your project, add the following to your `lakefile.toml`:

```
[[require]]
name = "Blase"
git = { url = "https://github.com/opencompl/lean-mlir", subDir = "Blase/" }
rev = "main"
```



(b) InstCombine (309 multi-width problems). BMC-Ours saturates the dataset despite multiple widths; BMC-Naive collapses from enumeration blowup.



(c) ROVER (19 problems, up to 12 symbolic widths). Naive enumeration is completely impractical; BMC-Ours and the unbounded solvers remain effective.

But Why 32? Because Can Only Solve Concrete Width

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

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

But Why 32? Because Can Only Solve Concrete Width

```
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)))
```

"does src equal tgt for all inputs?"

LLVM —Alive→ **QF_BV** → 🧞 **Solver** ✅ ❌

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, CompCe 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 st to obtain the best possible performance on emerging a

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 wr 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

BV SMT Solver

```
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)
```