Alive2: Bounded Translation Validation for LLVM

Nuno.P.Lopes, Juneyoung Lee, Chung-Kil Hur, Zhenyang Liu, John Regehr

Presented by JS. Kwon

IS661 Paper Review

Background: Compiler’s Optimization

😡

```
int foo(int X) {
  int result = X % 4;
  return result;
}
```

High-cost Program

😁

```
int foo(int X, int Y) {
  int result = X & 3;
  return result;
}
```

Low-cost Program

Background: Compiler’s Optimization

😡

```
int foo(int X) {
  int result = X % 4;
  return result;
}
```

High-cost Program

😁

```
int foo(int X, int Y) {
  int result = X & 3;
  return result;
}
```

Low-cost Program

😡

```
void swap(int* X, int* Y) {
  int tmp = *X;
  *X = *Y;
  *Y = tmp;
  return;
}
```

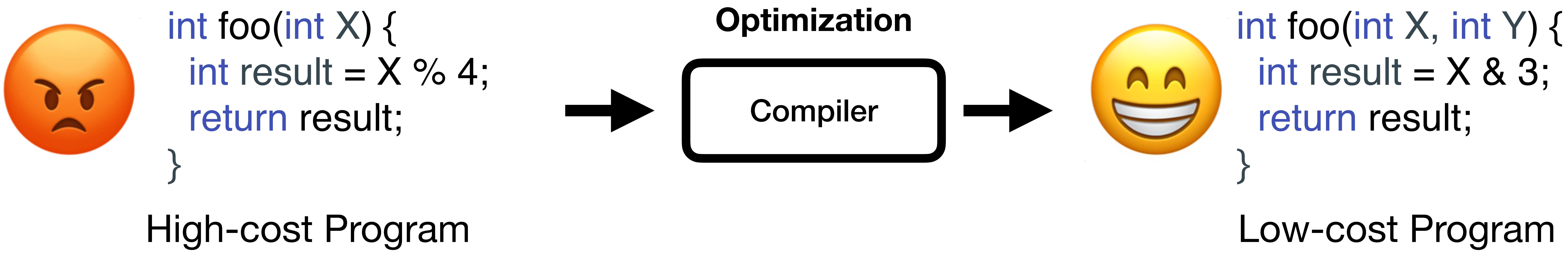
High-cost Program

😁

```
void swap(int* X, int* Y) {
  *X = *X ^ *Y;
  *Y = *X ^ *Y;
  *X = *X ^ *Y;
  return;
}
```

Low-cost Program

Background: Compiler's Optimization



😡

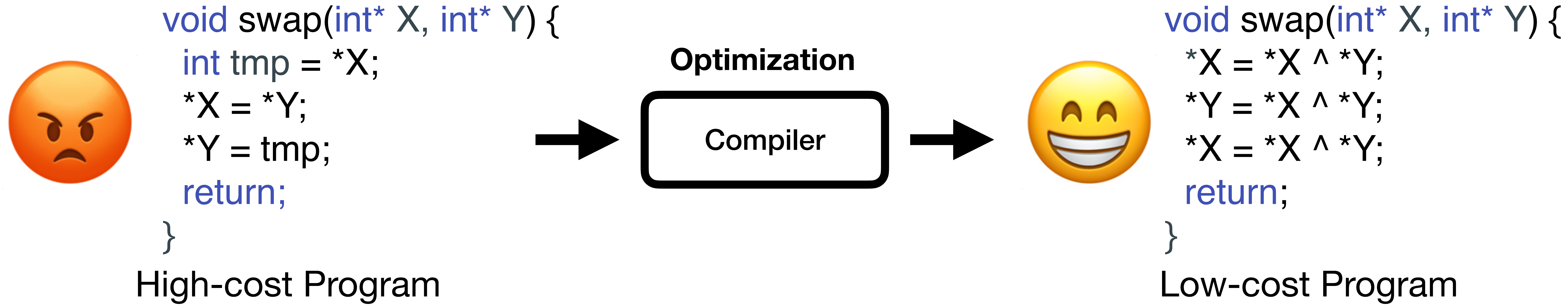
```
int foo(int X) {
  int result = X % 4;
  return result;
}
```

High-cost Program

😁

```
int foo(int X, int Y) {
  int result = X & 3;
  return result;
}
```

Low-cost Program



😡

```
void swap(int* X, int* Y) {
  int tmp = *X;
  *X = *Y;
  *Y = tmp;
  return;
}
```

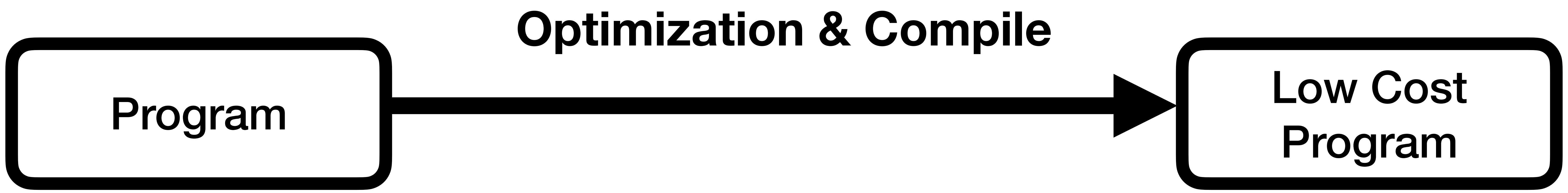
High-cost Program

😁

```
void swap(int* X, int* Y) {
  *X = *X ^ *Y;
  *Y = *X ^ *Y;
  *X = *X ^ *Y;
  return;
}
```

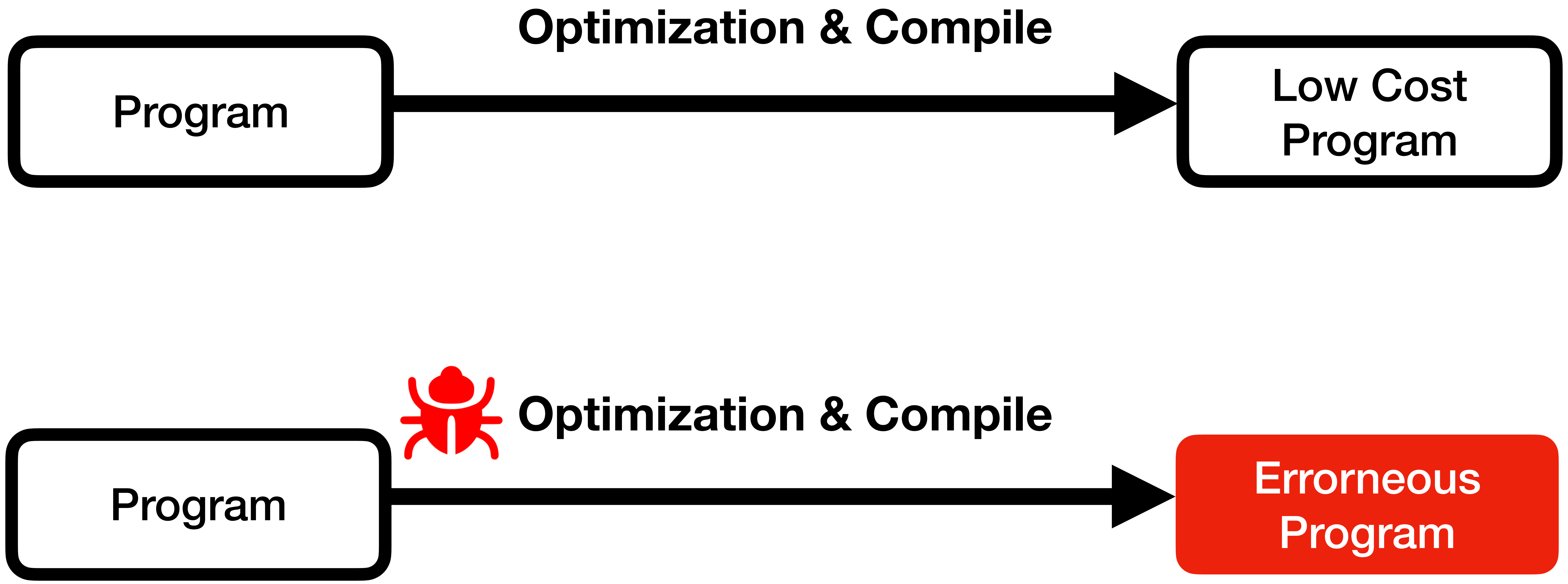
Low-cost Program

Motivation: Optimization Bugs



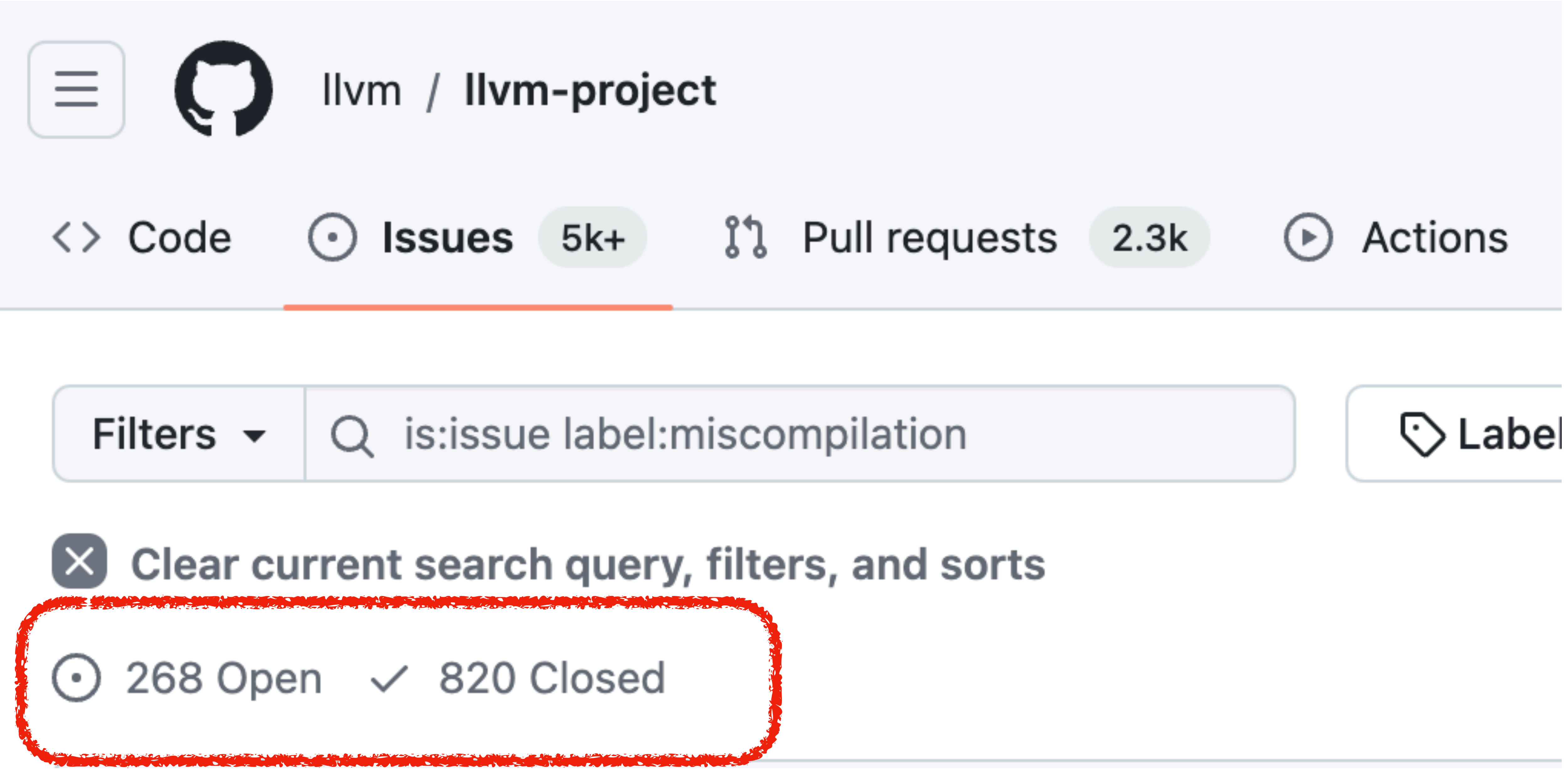
Motivation: Optimization Bugs

- Incorrect optimization can produce erroneous programs



LLVM's Bugs

- More than 1k bugs are reported



llvm / llvm-project

is:issue label:miscompilation

Clear current search query, filters, and sorts

268 Open ✓ 820 Closed

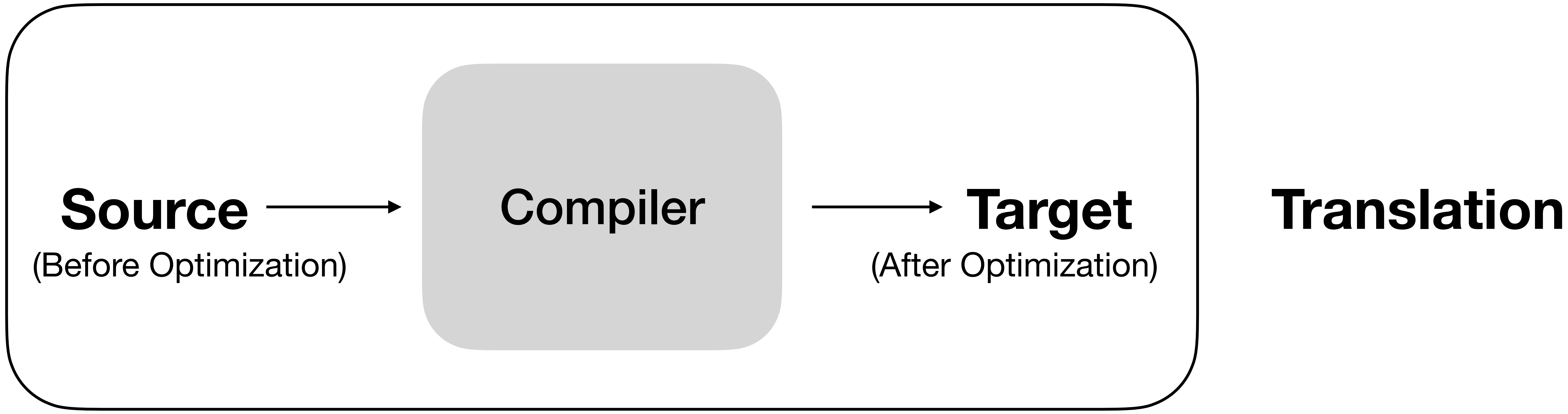
Challenge: LLVM’s Complexity

- More than **2.6 M** lines of code
- 3,525 commits in a month (2024.03)
- 56 Contributors (Developers) in a month

Hard to verify compiler in real-world!

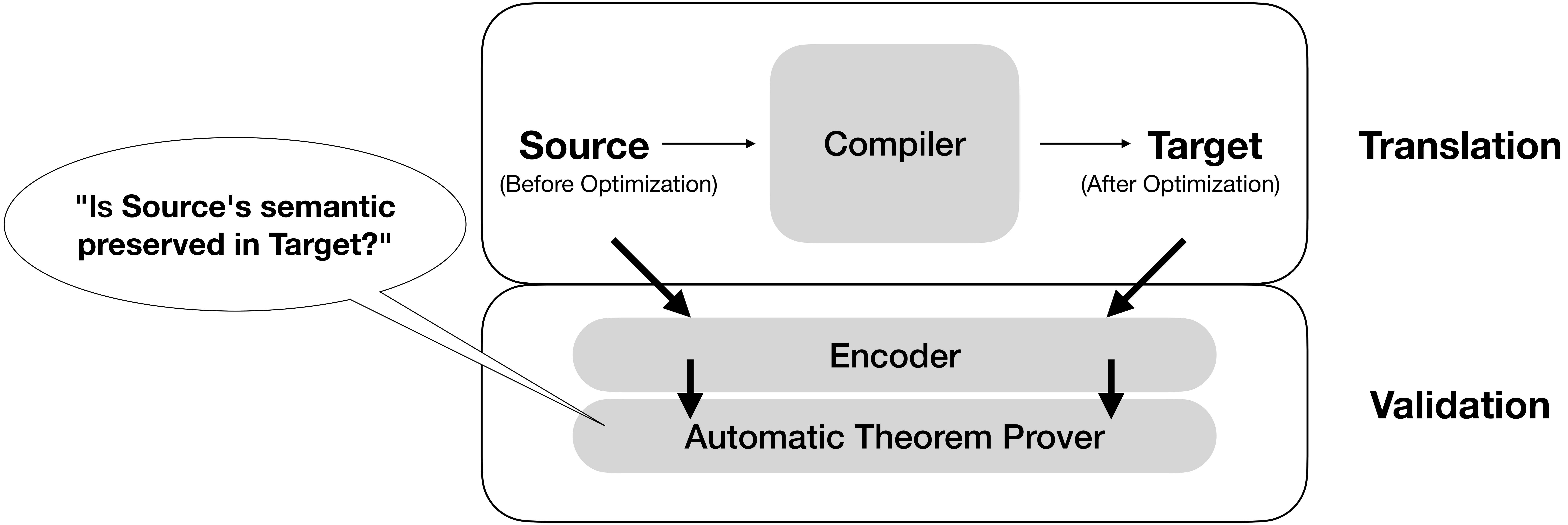
Idea: Translation Validation (TV)

- Check if translation preserves the program’s semantics
- Agnostic to compiler's implementation, focusing only on language’s semantics

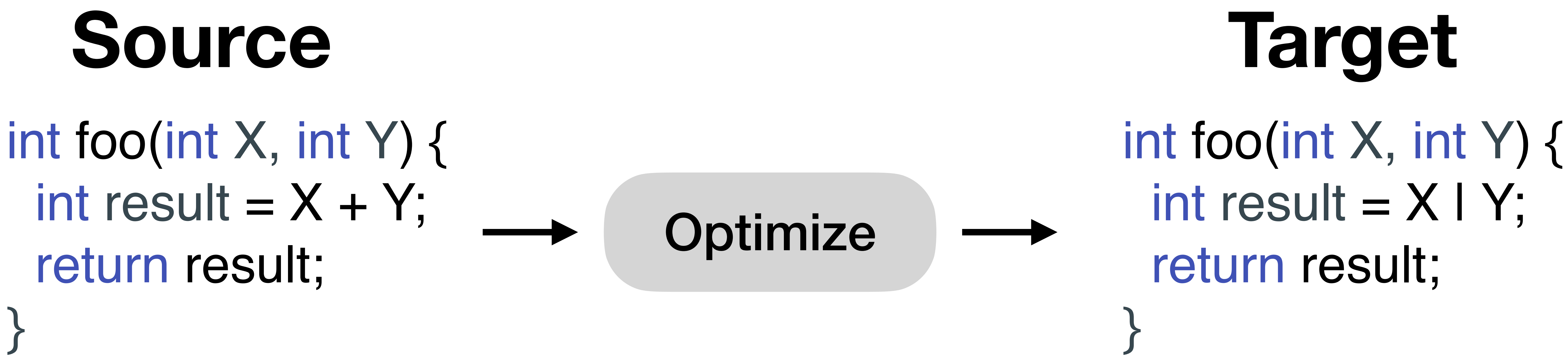


Idea: Translation Validation (TV)

- Check if translation preserves the program’s semantics
- Agnostic to compiler's implementation, focusing only on language’s semantics



Example: How Alive2 Works?



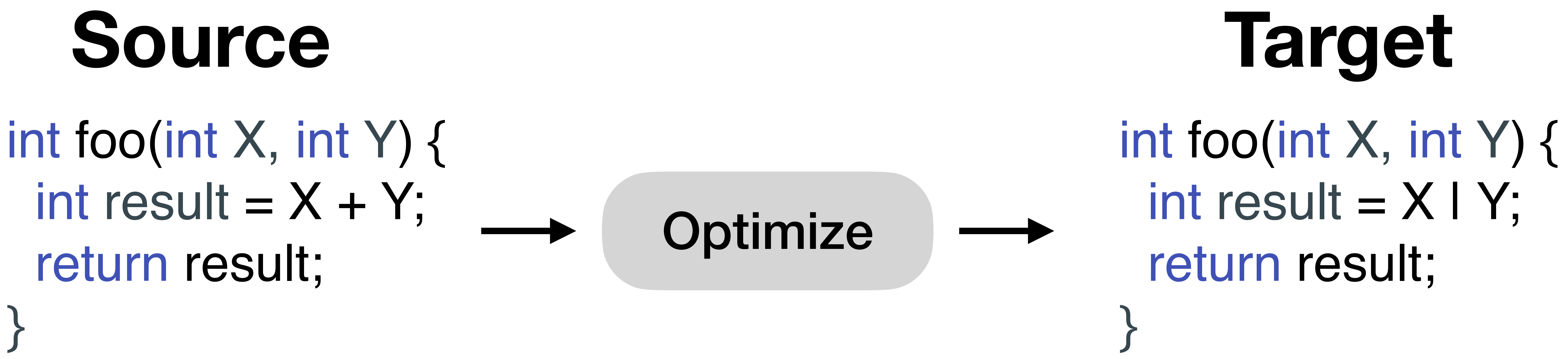
Source

```
int foo(int X, int Y) {
  int result = X + Y;
  return result;
}
```

Target

```
int foo(int X, int Y) {
  int result = X | Y;
  return result;
}
```

Example: How Alive2 Works?



Source

```
int foo(int X, int Y) {
  int result = X + Y;
  return result;
}
```

Target

```
int foo(int X, int Y) {
  int result = X | Y;
  return result;
}
```

Encode : $\forall X \forall Y . X + Y$

Example: How Alive2 Works?

Source

```
int foo(int X, int Y) {
  int result = X + Y;
  return result;
}
```

Optimize

Target

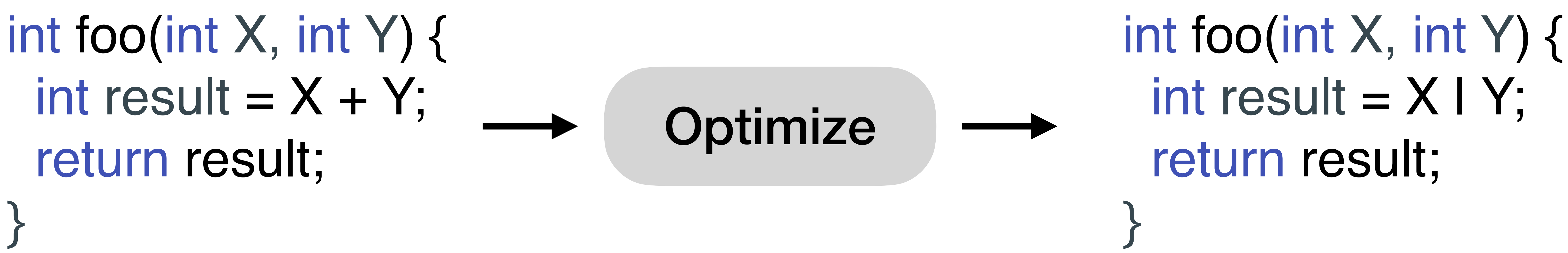
```
int foo(int X, int Y) {
  int result = X | Y;
  return result;
}
```

Encode : $\forall X \forall Y . X + Y$ $\forall X \forall Y . X | Y$

Example: How Alive2 Works?

Source

Target



```
int foo(int X, int Y) {
  int result = X + Y;
  return result;
}
```

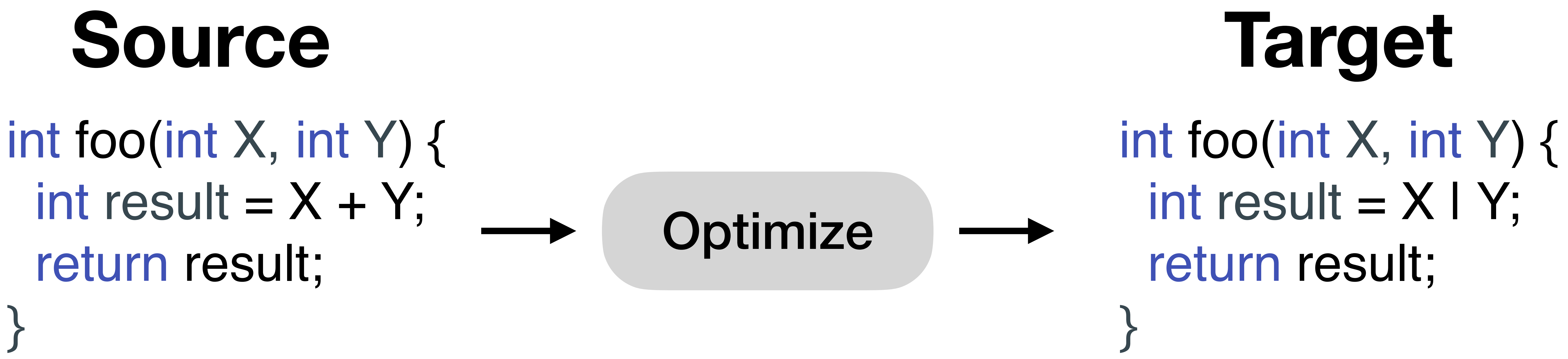
Optimize

```
int foo(int X, int Y) {
  int result = X | Y;
  return result;
}
```

Encode : $\forall X \forall Y . X + Y$ $\forall X \forall Y . X | Y$

Solve (SMT-Solver) : $\exists X \exists Y . X + Y \neq X | Y$ is **satisfiable?**

Example: How Alive2 Works?



Source

```
int foo(int X, int Y) {
  int result = X + Y;
  return result;
}
```

Optimize

Target

```
int foo(int X, int Y) {
  int result = X | Y;
  return result;
}
```

Encode : $\forall X \forall Y . X + Y$ $\forall X \forall Y . X | Y$

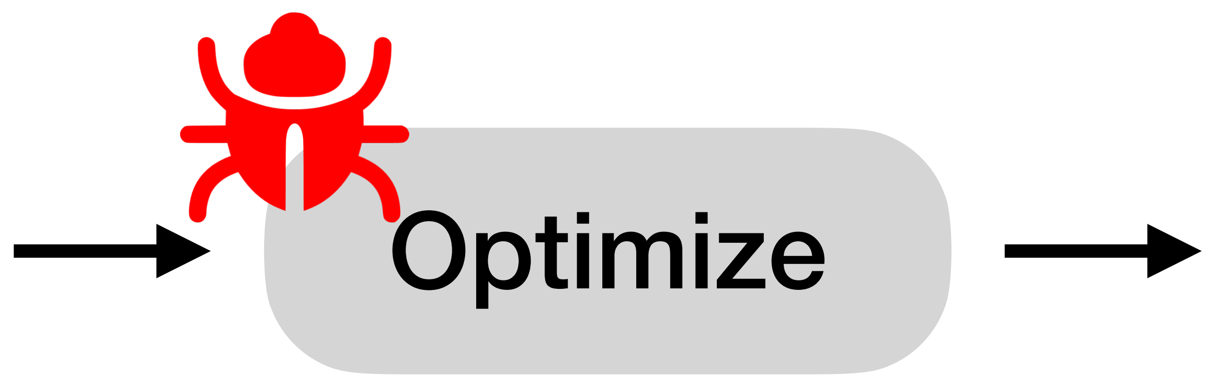
Solve (SMT-Solver) : $\exists X \exists Y . X + Y \neq X | Y$ is **satisfiable?**

Result : Satisfiable, Model: **X: 4, Y: 2**

Example: How Alive2 Works?

Source

```
int foo(int X, int Y) {
  int result = X + Y;
  return result;
}
```



Optimize

Target

```
int foo(int X, int Y) {
  int result = X | Y;
  return result;
}
```

Encode : $\forall X \forall Y . X + Y$ $\forall X \forall Y . X | Y$

Solve (SMT-Solver) : $\exists X \exists Y . X + Y \neq X | Y$ is **satisfiable?**

Result : Satisfiable, Model: **X: 4, Y: 2**

Wrong Optimization! Semantic are not equivalent When **X** is **4** & **Y** is **2**

How Is It Possible?

- LLVM has specification*
  - It is easy to encode semantic in a program

*https://llvm.org/docs/LangRef.html

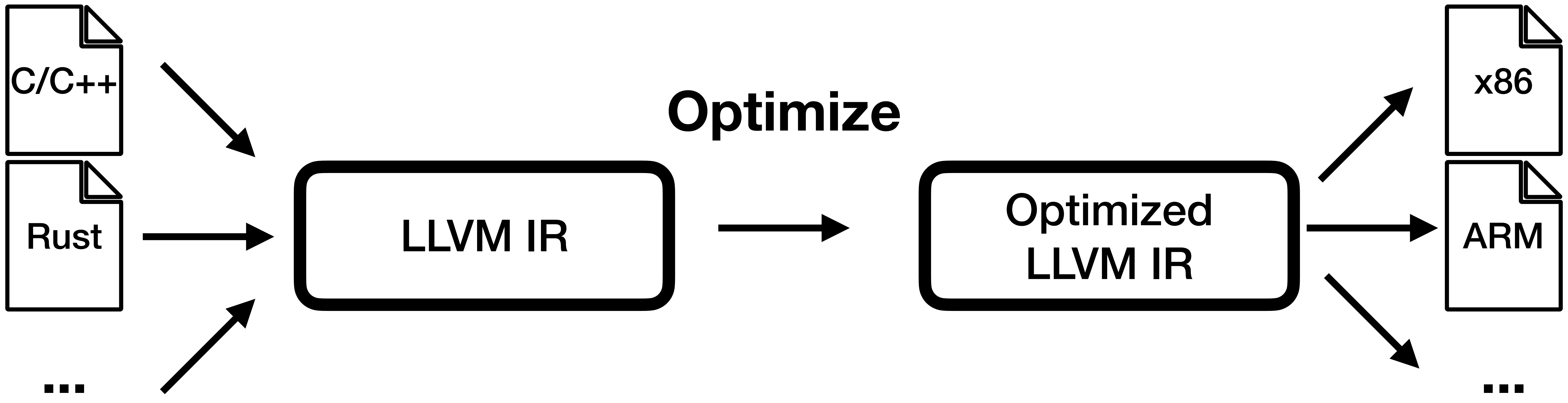
How Is It Possible?

- LLVM has specification*
  - It is easy to encode semantic in a program
- All programming languages are represented by LLVM IR, then optimized
  - Only need to encode the LLVM IR program

*https://llvm.org/docs/LangRef.html

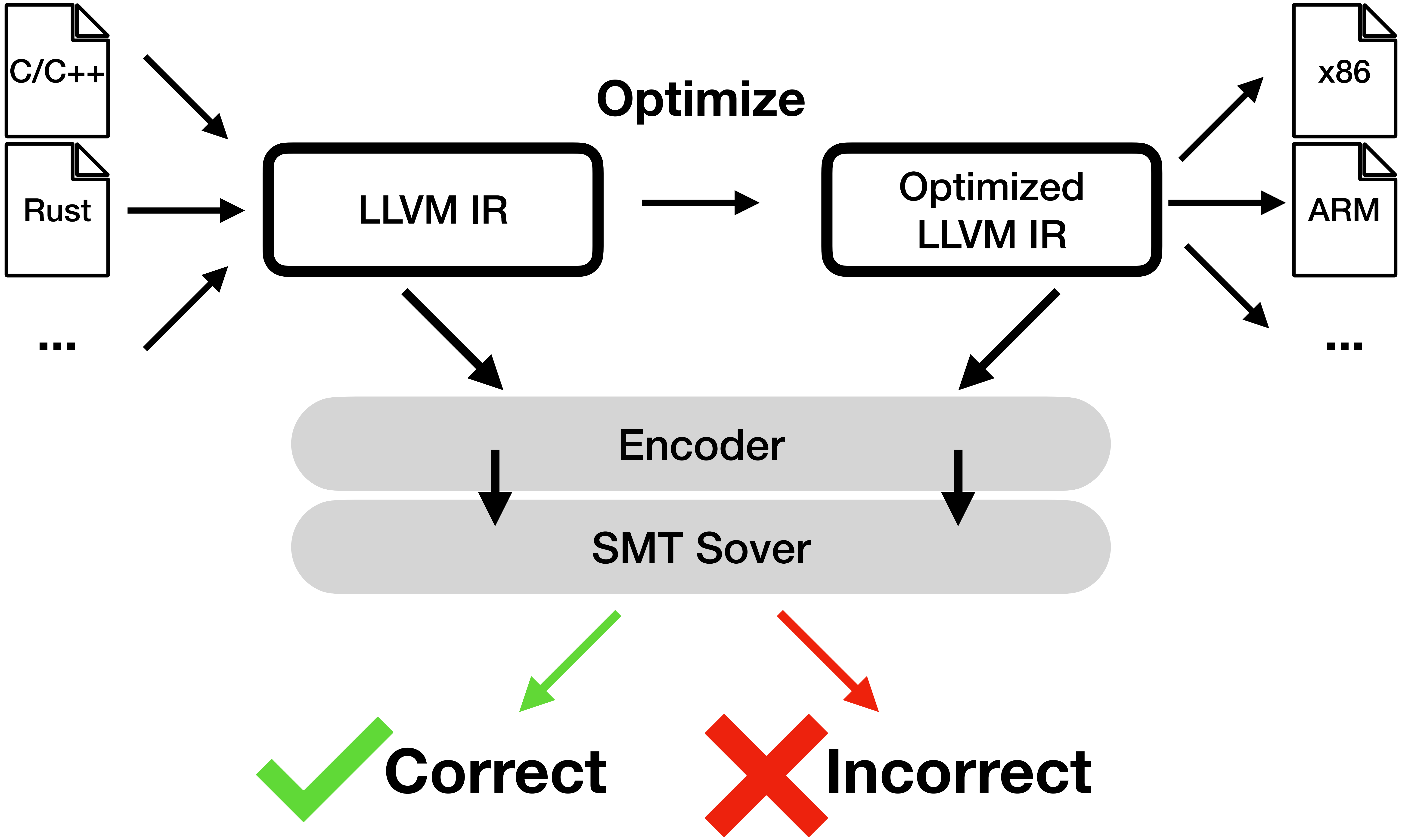
How Is It Possible?

- LLVM has specification*
  - It is easy to encode semantic in a program
- All programming languages are represented by LLVM IR, then optimized
  - Only need to encode the LLVM IR program



*https://llvm.org/docs/LangRef.html

Alive2



C/C++

Rust

...

LLVM IR

Optimize

Optimized LLVM IR

x86

ARM

...

Encoder

SMT Sover

✓ Correct

✗ Incorrect

Appealing Result

47 bugs in LLVM Test Suits

Appealing Result

47 bugs in LLVM Test Suits

8 patches in LLVM Specification

Appealing Result

47 bugs in LLVM Test Suits

8 patches in LLVM Specification

Only **<3%** false positives

Technical Details

- **2** key technical details
- What is correct optimization in LLVM?
    - Define and Check refinement of LLVM IR program
- How to improve precision of Alive2?
    - Unroll loops up to some bound

Technical Details

- **2** key technical details
- What is correct optimization in LLVM?
  - Define and Check refinement of LLVM IR program
- How to improve precision of Alive2?
  - Unroll loops up to some bound

Technical Details

- **2** key technical details
- What is correct optimization in LLVM?
    - Define and Check refinement of LLVM IR program
- How to improve precision of Alive2?
    - Unroll loops in program up to some bound

LLVM IR's Refinement

- **Undefined Behavior** is important
    - division by zero, shift past bitwidth ...

LLVM IR's Refinement

- **Undefined Behavior** is important
  - division by zero, shift past bitwidth ...

```
// division by zero
int foo(int X) {
  int result =  X / 0;
  return result;
}
```

↓

Undef

(... -2, -1, 0, 1, 2 ...)

```
// shift past bitwidth
int foo(int X) {
  int result =  X << 100;
  return result;
}
```

↓

Undef

(... -2, -1, 0, 1, 2 ...)

LLVM IR's Refinement

- **Undefined Behavior** is important

```
int foo(int X) {
  int result =  (X / 0) + 1;
  return result;
}
```

LLVM IR's Refinement

- **Undefined Behavior** is important

```
int foo(int X) {
  int result =  (X / 0) + 1;
  return result;
}
```

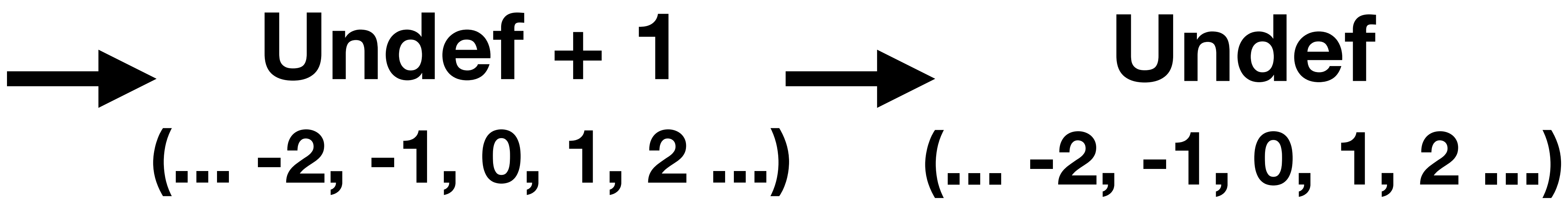
→ **Undef + 1**

(... -2, -1, 0, 1, 2 ...)

LLVM IR's Refinement

- **Undefined Behavior** is important

```
int foo(int X) {
  int result =  (X / 0) + 1;
  return result;
}
```



Undef + 1

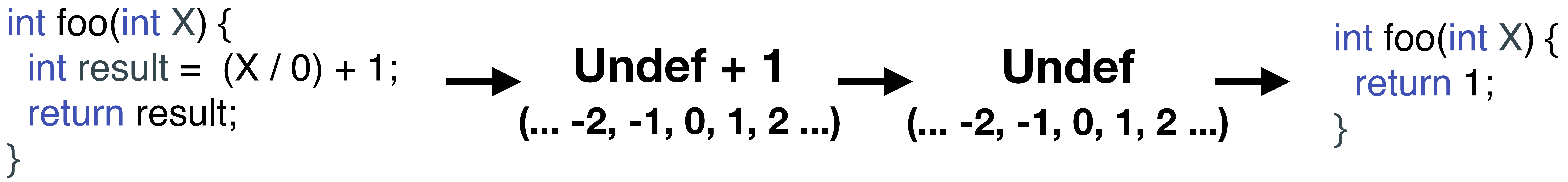
(... -2, -1, 0, 1, 2 ...)

Undef

(... -2, -1, 0, 1, 2 ...)

LLVM IR's Refinement

- **Undefined Behavior** is important



```
int foo(int X) {
  int result =  (X / 0) + 1;
  return result;
}
```

Undef + 1

(... -2, -1, 0, 1, 2 ...)

Undef

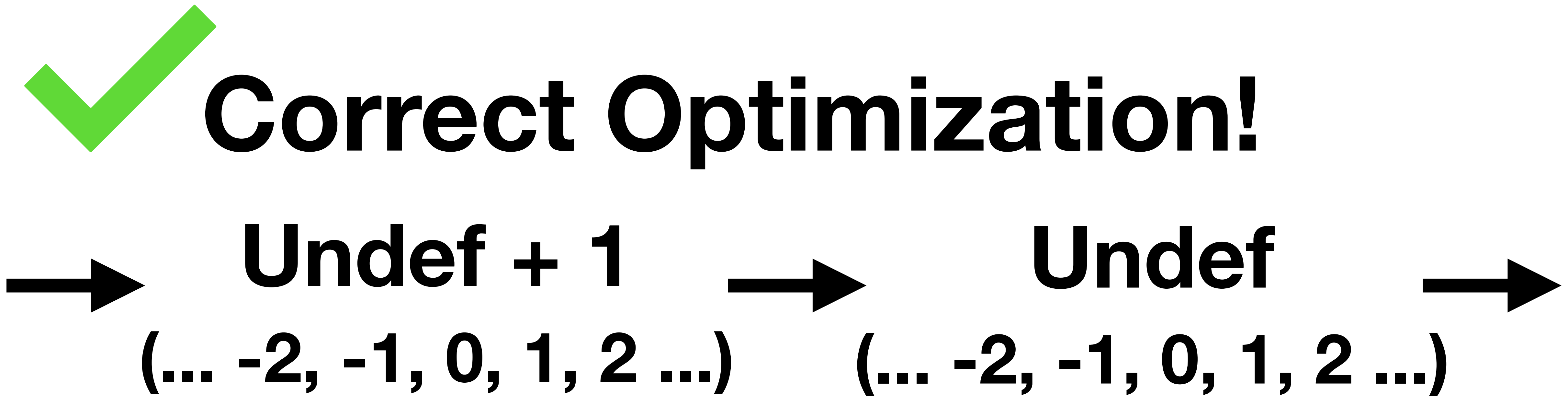
(... -2, -1, 0, 1, 2 ...)

```
int foo(int X) {
  return 1;
}
```

LLVM IR's Refinement

- **Undefined Behavior** is important

```
int foo(int X) {
  int result =  (X / 0) + 1;
  return result;
}
```



✓ **Correct Optimization!**

Undef + 1

(... -2, -1, 0, 1, 2 ...)

Undef

(... -2, -1, 0, 1, 2 ...)

```
int foo(int X) {
  return 1;
}
```

LLVM IR's Refinement

- **Undefined Behavior** is important

✓ **Correct Optimization!**

```
int foo(int X) {
  int result =  (X / 0) + 1;
  return result;
}
```

→ **Undef + 1** **(... -2, -1, 0, 1, 2 ...)** → **Undef** **(... -2, -1, 0, 1, 2 ...)** →

```
int foo(int X) {
  return 1;
}
```

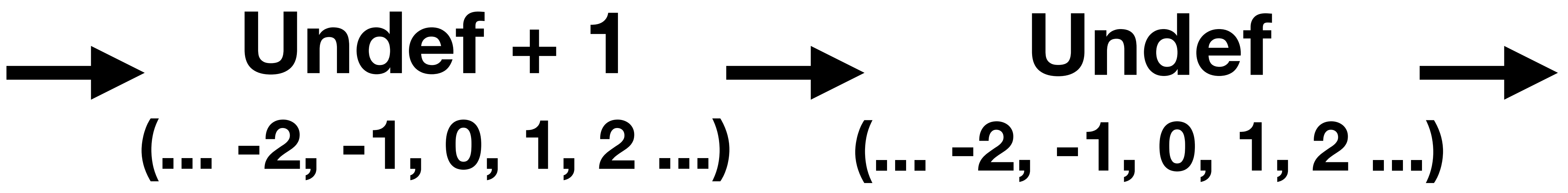
```
int foo(int X) {
  int result =  (X / 0) | 1;
  return result;
}
```

LLVM IR's Refinement

- **Undefined Behavior** is important

✓ **Correct Optimization!**

```
int foo(int X) {
  int result =  (X / 0) + 1;
  return result;
}
```



Undef + 1

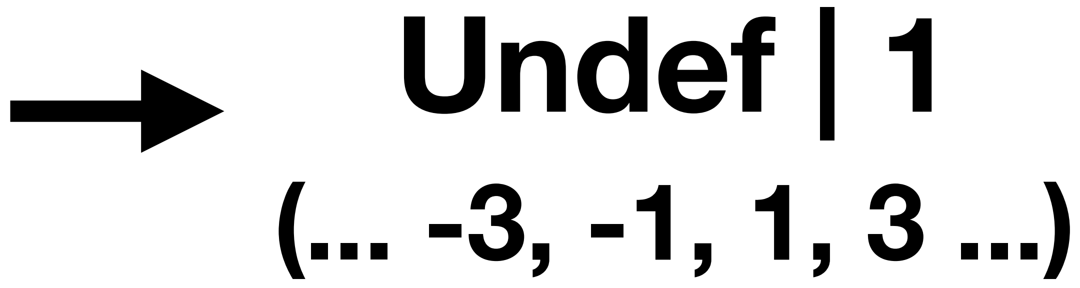
(... -2, -1, 0, 1, 2 ...)

Undef

(... -2, -1, 0, 1, 2 ...)

```
int foo(int X) {
  return 1;
}
```

```
int foo(int X) {
  int result =  (X / 0) | 1;
  return result;
}
```

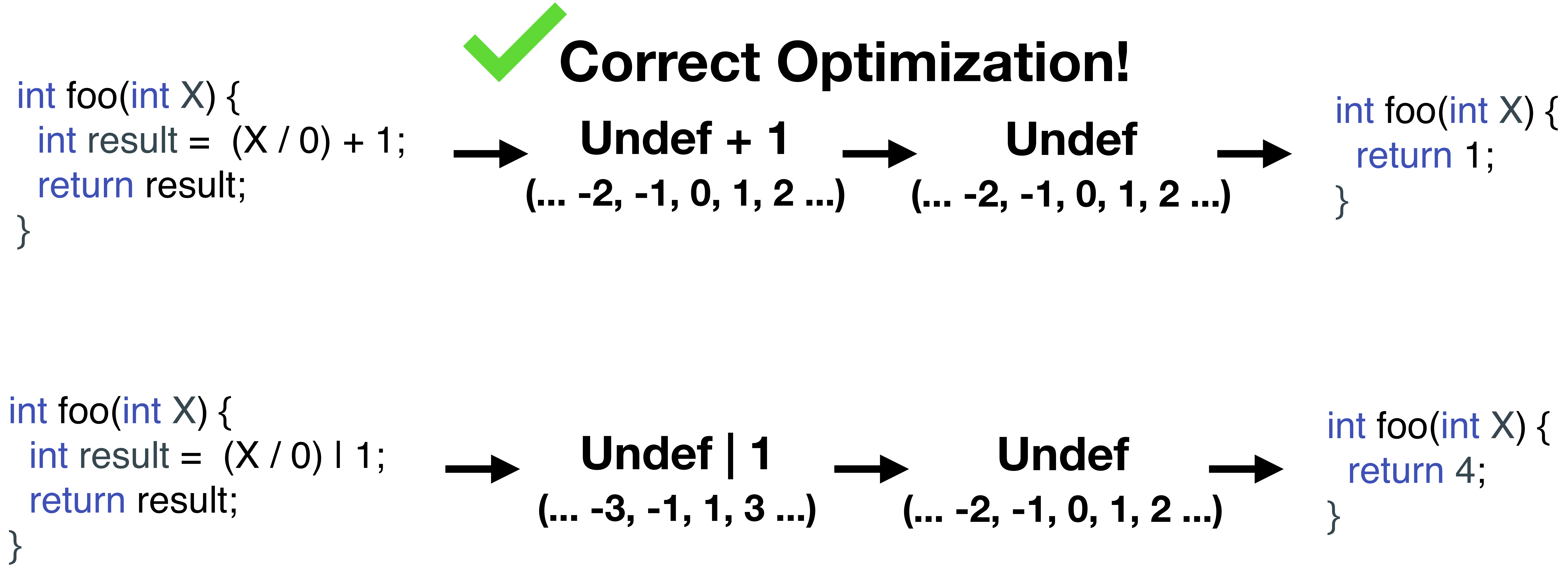


Undef | 1

(... -3, -1, 1, 3 ...)

LLVM IR's Refinement

- **Undefined Behavior** is important



✓ **Correct Optimization!**

```
int foo(int X) {
  int result =  (X / 0) + 1;
  return result;
}
```

Undef + 1

(... -2, -1, 0, 1, 2 ...)

Undef

(... -2, -1, 0, 1, 2 ...)

```
int foo(int X) {
  return 1;
}
```

```
int foo(int X) {
  int result =  (X / 0) | 1;
  return result;
}
```

Undef | 1

(... -3, -1, 1, 3 ...)

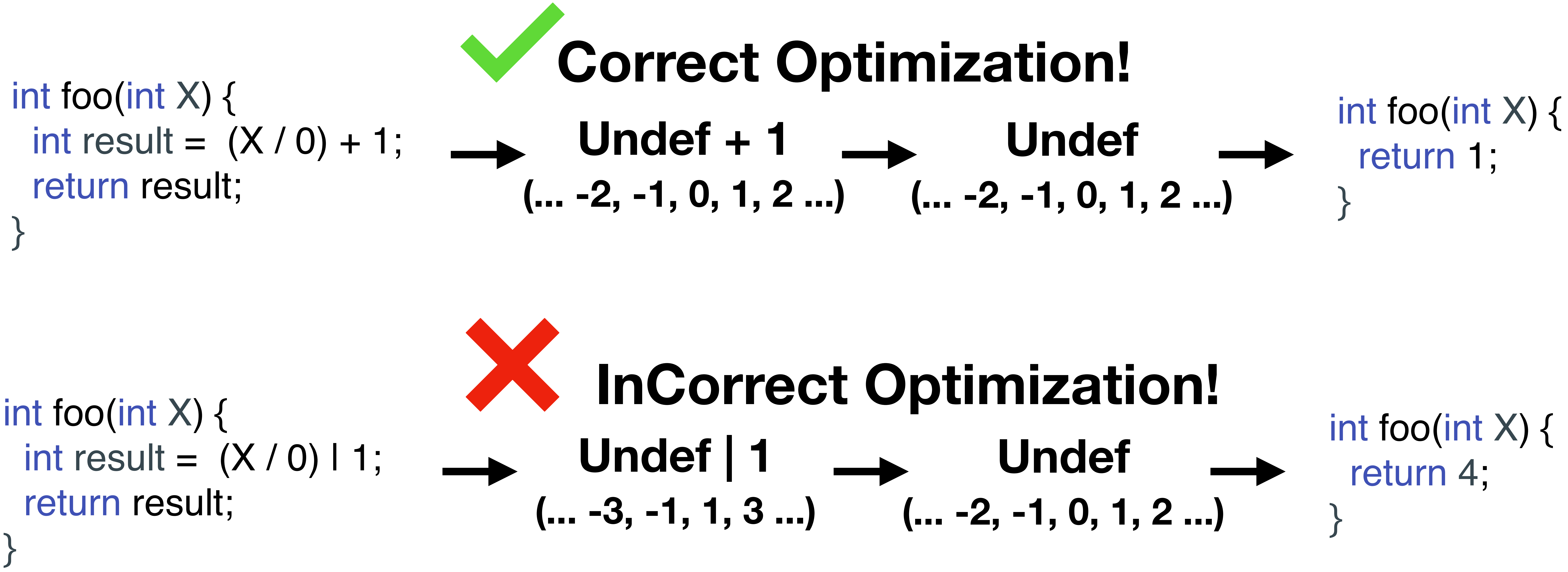
Undef

(... -2, -1, 0, 1, 2 ...)

```
int foo(int X) {
  return 4;
}
```

LLVM IR's Refinement

- **Undefined Behavior** is important



✓ **Correct Optimization!**

```
int foo(int X) {
  int result =  (X / 0) + 1;
  return result;
}
```

→ **Undef + 1** **(... -2, -1, 0, 1, 2 ...)** → **Undef** **(... -2, -1, 0, 1, 2 ...)** →

```
int foo(int X) {
  return 1;
}
```

✗ **InCorrect Optimization!**

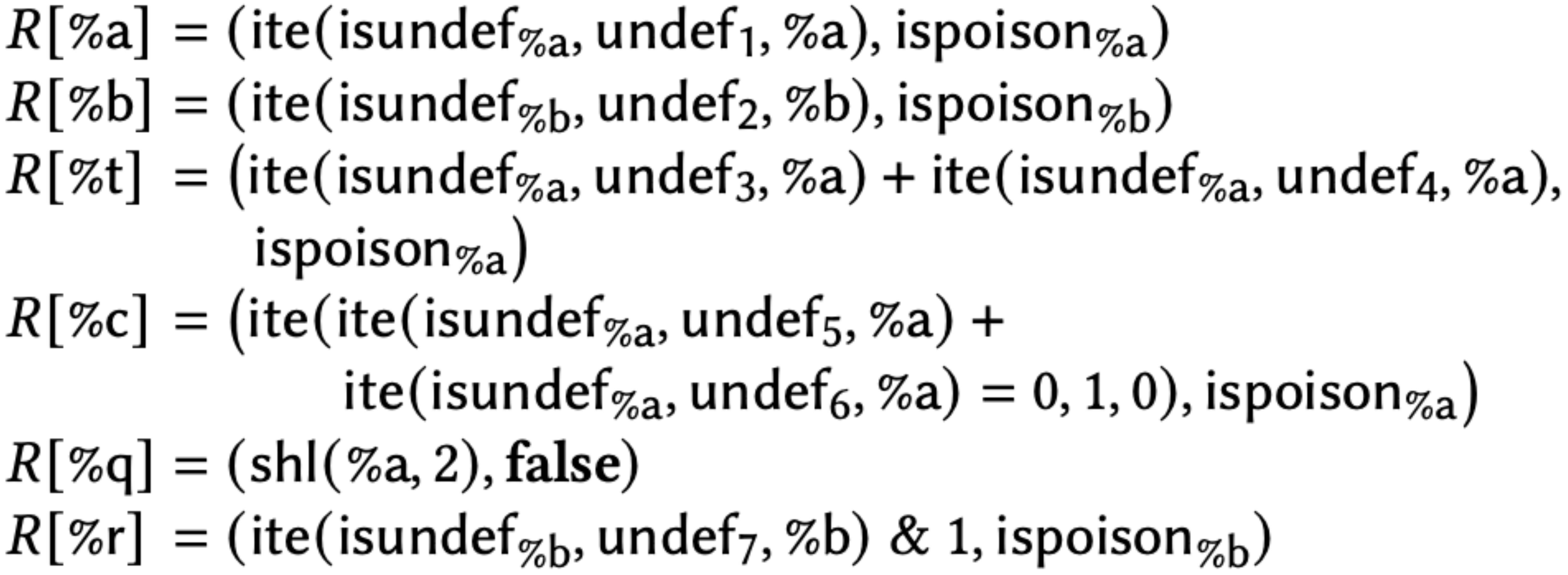
```
int foo(int X) {
  int result =  (X / 0) | 1;
  return result;
}
```

→ **Undef | 1** **(... -3, -1, 1, 3 ...)** → **Undef** **(... -2, -1, 0, 1, 2 ...)** →

```
int foo(int X) {
  return 4;
}
```

LLVM IR's Refinement

- **Alive2** defined every case of LLVM IR's refinement



$$
\begin{aligned}
R[\%\mathrm{a}] &= (\mathrm{ite}(\mathrm{isundef}_{\%\mathrm{a}}, \mathrm{undef}_1, \%\mathrm{a}), \mathrm{ispoison}_{\%\mathrm{a}}) \\
R[\%\mathrm{b}] &= (\mathrm{ite}(\mathrm{isundef}_{\%\mathrm{b}}, \mathrm{undef}_2, \%\mathrm{b}), \mathrm{ispoison}_{\%\mathrm{b}}) \\
R[\%\mathrm{t}] &= \big(\mathrm{ite}(\mathrm{isundef}_{\%\mathrm{a}}, \mathrm{undef}_3, \%\mathrm{a}) + \mathrm{ite}(\mathrm{isundef}_{\%\mathrm{a}}, \mathrm{undef}_4, \%\mathrm{a}), \\
&\qquad \mathrm{ispoison}_{\%\mathrm{a}}\big) \\
R[\%\mathrm{c}] &= \big(\mathrm{ite}(\mathrm{ite}(\mathrm{isundef}_{\%\mathrm{a}}, \mathrm{undef}_5, \%\mathrm{a}) + \\
&\qquad\qquad \mathrm{ite}(\mathrm{isundef}_{\%\mathrm{a}}, \mathrm{undef}_6, \%\mathrm{a}) = 0, 1, 0), \mathrm{ispoison}_{\%\mathrm{a}}\big) \\
R[\%\mathrm{q}] &= (\mathrm{shl}(\%\mathrm{a}, 2), \mathbf{false}) \\
R[\%\mathrm{r}] &= (\mathrm{ite}(\mathrm{isundef}_{\%\mathrm{b}}, \mathrm{undef}_7, \%\mathrm{b})\ \&\ 1, \mathrm{ispoison}_{\%\mathrm{b}})
\end{aligned}
$$

…

Alive2's Loop Unrolling

- Alive2 wants **"Bug means a bug"**
- Alive2 Unroll loops in program up to some bound

```
int foo(int X) {
  int j = 3;
  for (int i = 0; i < 3; i++)
  {
    j--;
  }
  return X + j;    // X + 0
```

→

```
int foo(int X) {
  return X;    // X + 0
```

Alive2's Loop Unrolling

- Alive2 wants **"Bug means a bug"**
- Alive2 Unroll loops in program up to some bound

```
int foo(int X) {
  int j = 3;
  for (int i = 0; i < 3; i++)
  {
    j--;
  }
  return X + j;    // X + 0
```

→

```
int foo(int X) {
  int j = 3;
    j--;              // unrolling
    j--;
    j--;
  return X + j;    // X + 0
```

→

```
int foo(int X) {
  return X;    // X + 0
```

Evaluation: Precision

- **Translation Validation of LLVM's Unit Tests**

All TestSuits	Supported	True Alarms	Reported
168,000	36,000	121	43

Evaluation: Scalability

- **Translation Validation for Applications**

Programs	Total	True Alarms	False Alarms	Failed	Unsupported
bzip2	2.2K	333	10	735	1,125
gzip	2.6K	884	4	965	754
oggenc	1.8K	440	4	660	663
ph7	5.6K	1,393	28	1,372	2,755
SQLite3	12.2K	2,314	38	2,202	7,543

Evaluation: Scalability

- **Translation Validation for Applications**

Only <3% False Positives!

Programs	Total	True Alarms	False Alarms	Failed	Unsupported
bzip2	2.2K	333	10	735	1,125
gzip	2.6K	884	4	965	754
oggenc	1.8K	440	4	660	663
ph7	5.6K	1,393	28	1,372	2,755
SQLite3	12.2K	2,314	38	2,202	7,543

Evaluation

- **Updates to the LLVM IR Semantics**
    - In encoding LLVM IR, Ambiguously written specifications were founded
    - After discussions with LLVM communities, 8 patches applied to spec

Conclusion

- Translation validator for LLVM middle-end optimization
  - TV is suitable for LLVM's optimization
- Formally defined refinement & Precise validation
  - Alive2 is suitable for real-world validation
- Found new bugs in a LLVM
  - 47 bugs in LLVM Test Suits
  - 8 patches in LLVM Specification

Review (My Opinion)

- Positive aspects
  - A useful testing system to find optimization bugs in LLVM
  - Fast, accurate, and able to identify the reasons for bugs

- Negative aspects
  - When new features are added to LLVM IR, new encodings need to be added
  - Since encoding is done manually by humans, problems can arise (Alive2's bug)