A Decade Verifying LLVM

HOW TO RETROFIT SOUNDNESS IN INDUSTRIAL SOFTWARE

Nuno P. Lopes
University of Lisbon

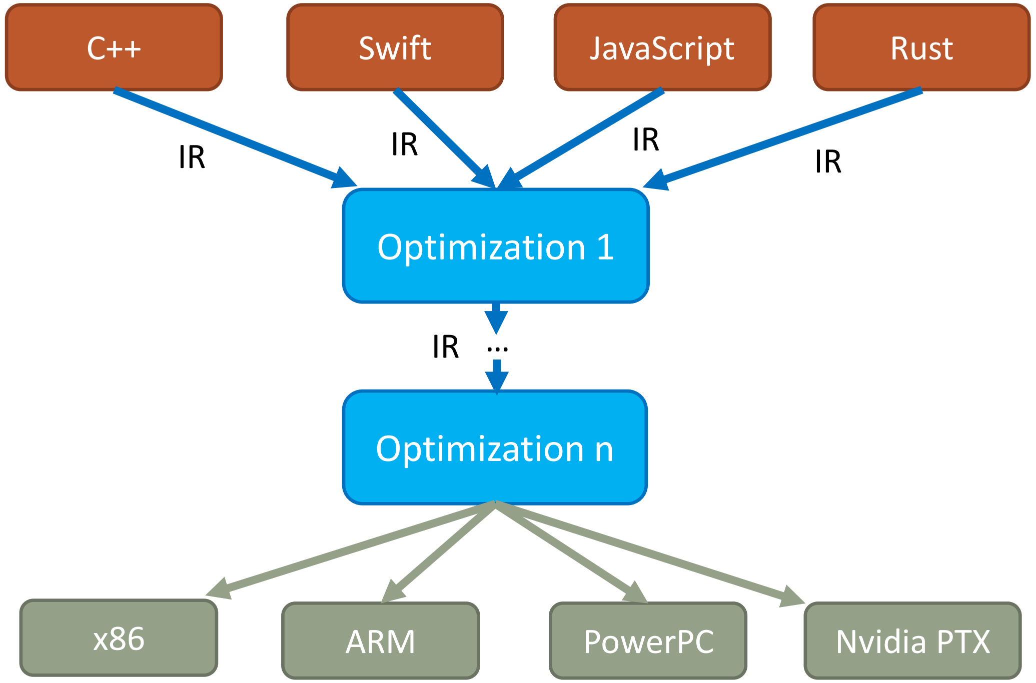
TÉCNICO LISBOA

LLVM

- Compiler used by Apple, Arm, Azul Systems, Cray, Google, Huawei, Imagination Technologies, Intel, Meta, Microsoft, PlayStation, Qualcomm, …

- Used for: C, C++, ObjC, Fortran, Rust, Swift, TensorFlow, PyTorch, DirectX, OpenGL, WebAssembly, …

Typical compiler



LLVM’s SSA-based IR

```c
int f(int a, int cond) {
  int b;

  if (cond)
    b = a + 1;
  else
    b = a << 2;

  return b;
}
```

```llvm
define i32 @f(i32 %a, i32 %cond) {
  %cmp = icmp ne i32 %cond, 0
  br i1 %cmp, label %then, label %else

then:
  %b1 = add i32 %a, 1
  br label %end

else:
  %b2 = shl i32 %a, 2
  br label %end

end:
  %b = phi i32 [ %b1, %then ], [ %b2, %else ]
  ret i32 %b
}
```

IR: The most important data-structure

- Used as input from frontends
- Used as input/output by optimizations

<u>IR must be:</u>

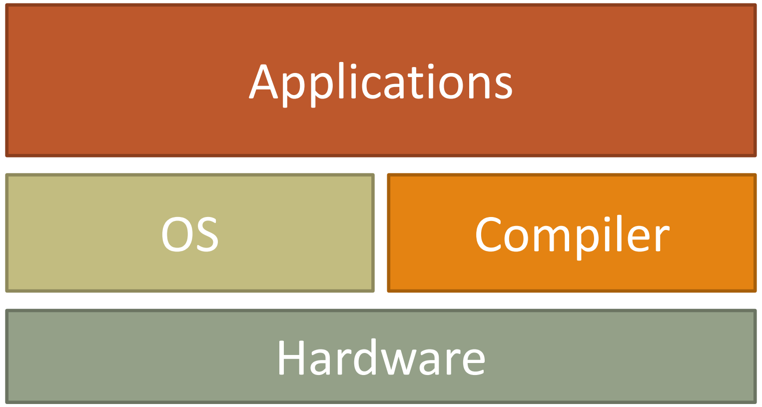
- Expressible
- Support desired optimizations
- Block wrong transformations
- Efficient transformations
- Efficient analyses
- Efficient encoding of assumptions from source language
- Efficiently cache derived facts
- Efficient lowering to ASM

??

UNSAT!

Why focus on compilers?

- Today’s software goes at least through one compiler (often more than one!)
- Correctness and safety depends on compilers



Do compilers have bugs?

Skeletal Program Enumeration for Rigorous Compiler Testing

Qirun Zhang Chengnian Sun Zhendong Su
University of California, Davis, United States
{qrzhang, cnsun, su}@ucdavis.edu

Volatiles Are Miscompiled, and What to Do about It

Eric Eide
University of Utah, School of Computing
Salt Lake City, UT USA
eeide@cs.utah.edu

John Regehr
University of Utah, School of Computing
Salt Lake City, UT USA
regehr@cs.utah.edu

Finding Compiler Bugs via Live Code Mutation

Chengnian Sun Vu Le Zhendong Su
Department of Computer Science, University of California, Davis, USA
{cnsun, vmle, su}@ucdavis.edu

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr
University of Utah, School of Computing
{ jxyang, chenyang, eeide, regehr }@cs.utah.edu

Random Testing for C and C++ Compilers with YARPGen

VSEVOLOD LIVINSKII, University of Utah and Intel Corporation, USA
DMITRY BABOKIN, Intel Corporation, USA
JOHN REGEHR, University of Utah, USA

Random Testing of C Calling Conventions

Christian Lindig
Saarland University
Department of Computer Science
Saarbrücken, Germany
lindig@cs.uni-sb.de

Fuzzing tools found thousands of bugs in gcc and LLVM!

Compiler bugs can be nasty!

- Miscompilations can introduce security vulnerability in safe programs
- First documented case: CVE-2006-1902

- Academics have used a bug in LLVM to introduce a backdoor in “sudo” (2015)

Summary so far

- Compilers are crucial in software ecosystem
- But they have bugs, including security-sensitive ones
- Designing IRs is extremely complex

Optimizations are easy to get wrong

```
int a = x << c;
int b = a / d;
```

➡

```
int t = d / (1 << c);
int b = x / t;
```

$x * 2^c / d$

$x / (d / 2^c) = x / d * 2^c$

$= x * 2^c / d$

(c and d are constants)

Optimizations are easy to get wrong

```
int a = x << c;
int b = a / d;
```

➡

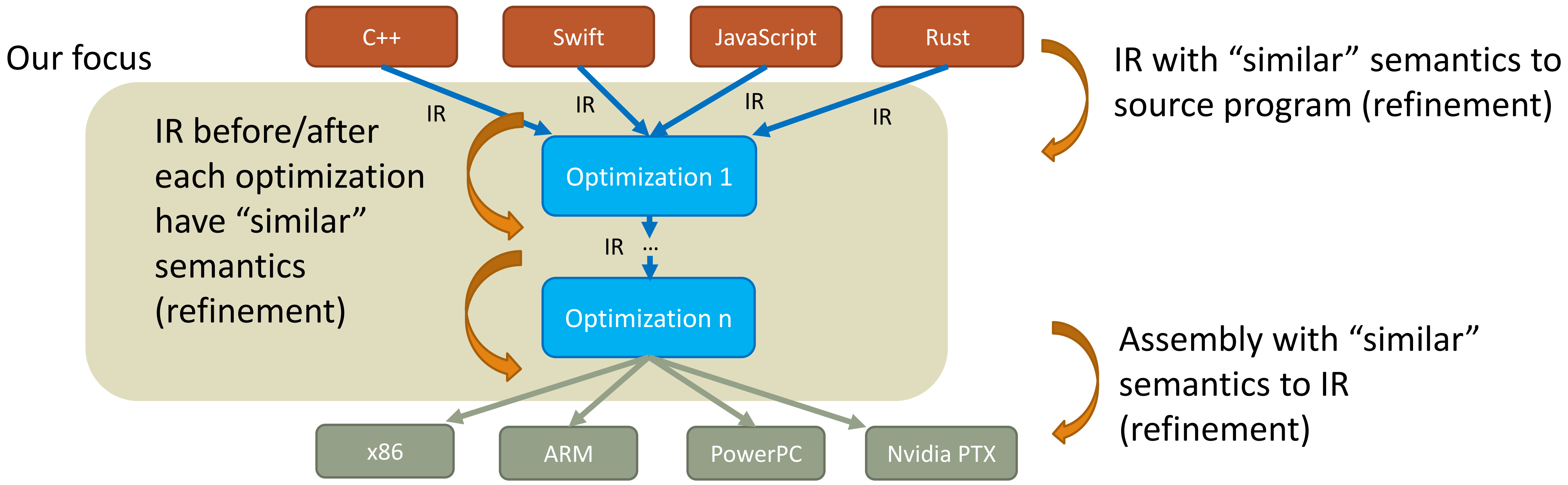
```
int t = d / (1 << c);
int b = x / t;
```

```
ERROR: Domain of definedness of Target is
smaller than Source's for i4 %b

Example:
%X i4 = 0x0 (0)
c  i4 = 0x3 (3)
d  i4 = 0x7 (7)
%a i4 = 0x0 (0)
(1 << c) i4 = 0x8 (8, -8)
%t i4 = 0x0 (0)
Source value: 0x0 (0)
Target value: UB
```

LLVM bug #21245

What's a correct compiler?



First attempt: Alive

- Fully automatic verification tool for peephole optimizations (SMT-based)
- Found dozens of bugs in LLVM
- Avoided many more bugs due to use before commit

- Released as open-source in Fall 2014
- Used by developers across 8+ companies

A new optimization, or how Alive was adopted

- A developer wrote a new optimization that improved benchmarks:
  - 3.8% perlbmk (SPEC CPU 2000)
  - 1% perlbench (SPEC CPU 2006)
  - 1.2% perlbench (SPEC CPU 2006) w/ LTO+PGO

40 lines of code

August 2014

A new optimization, or how Alive was adopted

- The first patch was wrong

```
Pre: isPowerOf2(C1 ^ C2)
%x = add %A, C1
%i = icmp ult %x, C3
%y = add %A, C2
%j = icmp ult %y, C3
%r = or %i, %j
  =>
%and = and %A, ~(C1 ^ C2)
%lhs = add %and, umax(C1, C2)
%r = icmp ult %lhs, C3
```

```
ERROR: Mismatch in values of %r

Example:
%A i4 = 0x0 (0)
C1 i4 = 0xA (10, -6)
C3 i4 = 0x5 (5)
C2 i4 = 0x2 (2)
%x i4 = 0xA (10, -6)
%i i1 = 0x0 (0)
%y i4 = 0x2 (2)
%j i1 = 0x1 (1, -1)
%and i4 = 0x0 (0)
%lhs i4 = 0xA (10, -6)
Source value: 0x1 (1, -1)
Target value: 0x0 (0)
```

A new optimization, or how Alive was adopted

- The second patch was wrong
- The third patch was correct!
- Still fired on the benchmarks!

```
Pre: C1 u> C3 &&
     C2 u> C3 &&
     isPowerOf2(C1 ^ C2) &&
     isPowerOf2(-C1 ^ -C2) &&
     (-C1 ^ -C2) == ((C3-C1) ^ (C3-C2)) &&
     abs(C1-C2) u> C3
%x = add %A, C1
%i = icmp ult %x, C3
%y = add %A, C2
%j = icmp ult %y, C3
%r = or %i, %j
  =>
%and = and %A, ~(C1^C2)
%lhs = add %and, umax(C1,C2)
%r = icmp ult %lhs, C3
```

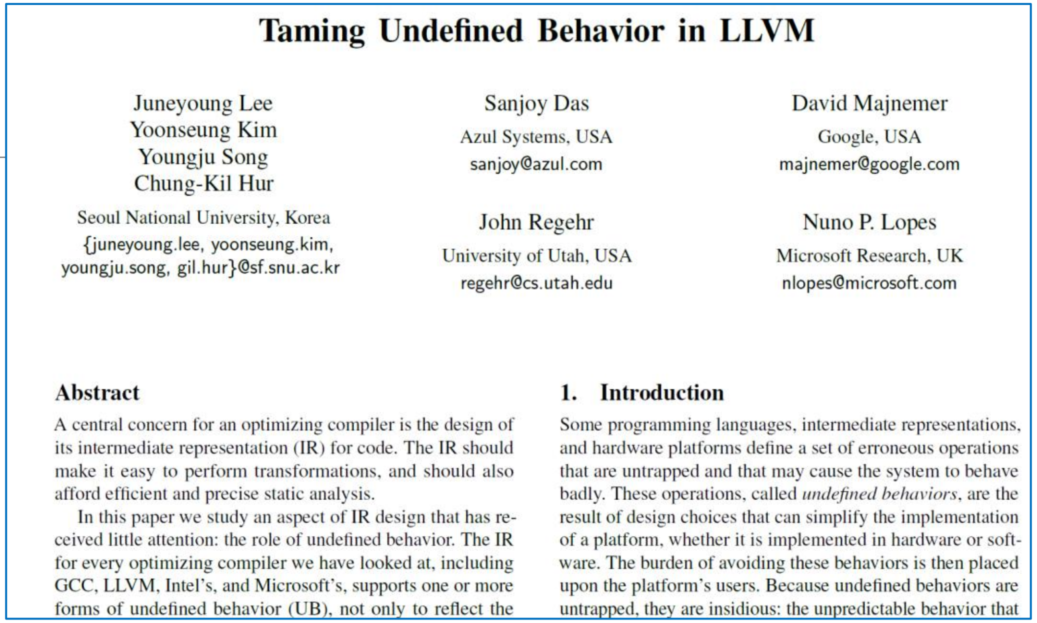
Alive couldn’t verify all LLVM optimizations

- They seemed wrong, but we weren’t sure
- Nobody we asked knew

- We started digging!

Study on UB semantics

- Published in 2017
- Showed that LLVM IR wasn't expressive enough for all optimizations that people cared about
- E.g. can't have GVN & Loop unswitching
- Proposed a fix: a new freeze instruction



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

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

David Majnemer
Google, USA
majnemer@google.com

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

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

Undefined Behavior in LLVM

- "Immediate UB" – this is like undefined behavior in C or C++, destroys the meaning of the program
  - Division by zero
  - Out of bounds memory accesses
- Undef – an arbitrary value
  - Mainly used to model uninitialized memory
  - Each read can return a different value!
- Poison – a contagious error value, similar to NaN
  - Things like integer overflow turn into poison

GVN vs Loop Unswitching

```
while (c) {
  if (c2) { foo }
  else    { bar }
}
```

```
if (c2) {
  while (c) { foo }
} else {
  while (c) { bar }
}
```

<u>Loop unswitch</u>

Branch on poison/undef **cannot** be UB

Otherwise, wrong if loop never executed

GVN vs Loop Unswitching

```
t = x + 1;
if (t == y) {
  w = x + 1;
  foo(w);
}
```

```
t = x + 1;
if (t == y) {
  foo(y);
}
```

Contradiction with loop unswitching!

GVN

Branch on poison/undef **must** be UB

Otherwise, wrong if y poison but not x

But.. no one listened!

- A compiler developer reaction (early 2017): “Paper is a nice read, but examples are academic. No one will ever write such code”.

- LLVM miscompiles itself (July 2017)

- Broken LLVM miscompiles internal code
  - The company’s devs waste a couple of weeks debugging

- What happened?

What was wrong?

"Every transformation above seems of no problem, but the composition result is wrong. It is still not clear which transformation to blame."
— LLVM developer

The transformations were…
GVN & loop unswitching!

Internal code miscompiling

- The compiler developer reaction: “Paper is a nice read, but examples are academic. No one will ever write such code.”

(Suspense

NOTE: Not blaming anyone. We weren’t sure ourselves of the extent of the issue. Just a funny story.

- He wrote the code 😂

- Bug “fixed”: `if (match_code(..)) dont_optimize();`

Freeze wasn't used until 2 years later

- We needed more pain & suffering:
  - Miscompilation in Android (2018)
  - Azul Java compiler broken (2019)

- In Oct 2019, people asked us to commit freeze to fix bugs once and for all

- Initial patches regressed performance: committed & rolled back!
  - Perf matters more sometimes

- Freeze + other fixes released in LLVM 10
  - Incl performance improvements due to additional expressivity

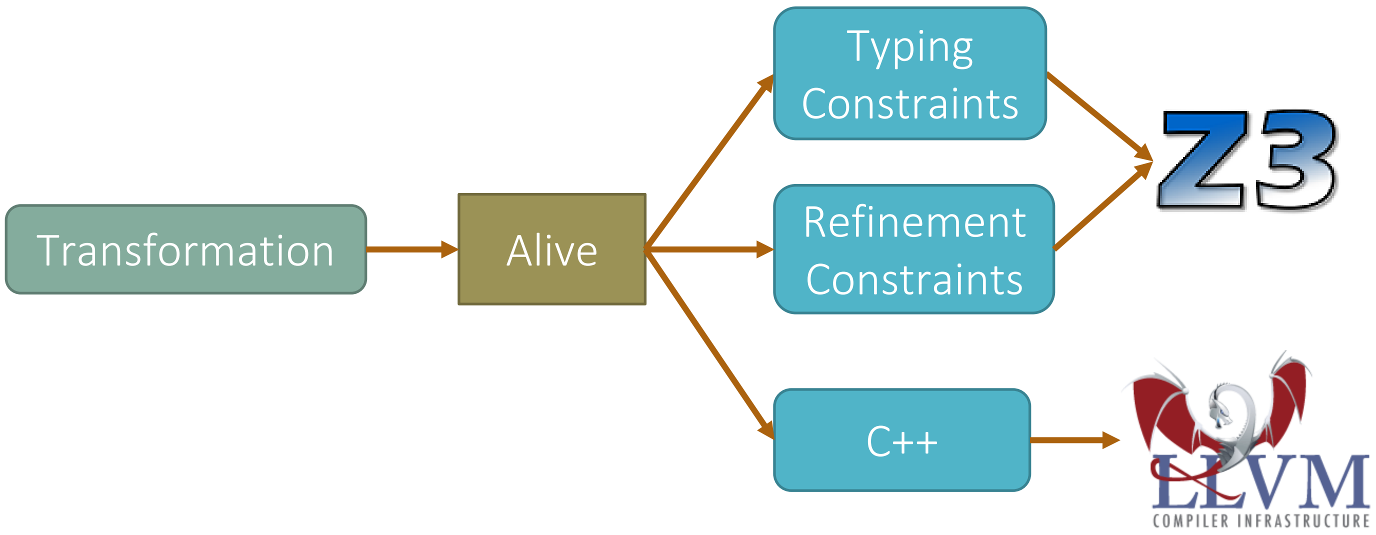
It’s hard to sell correctness

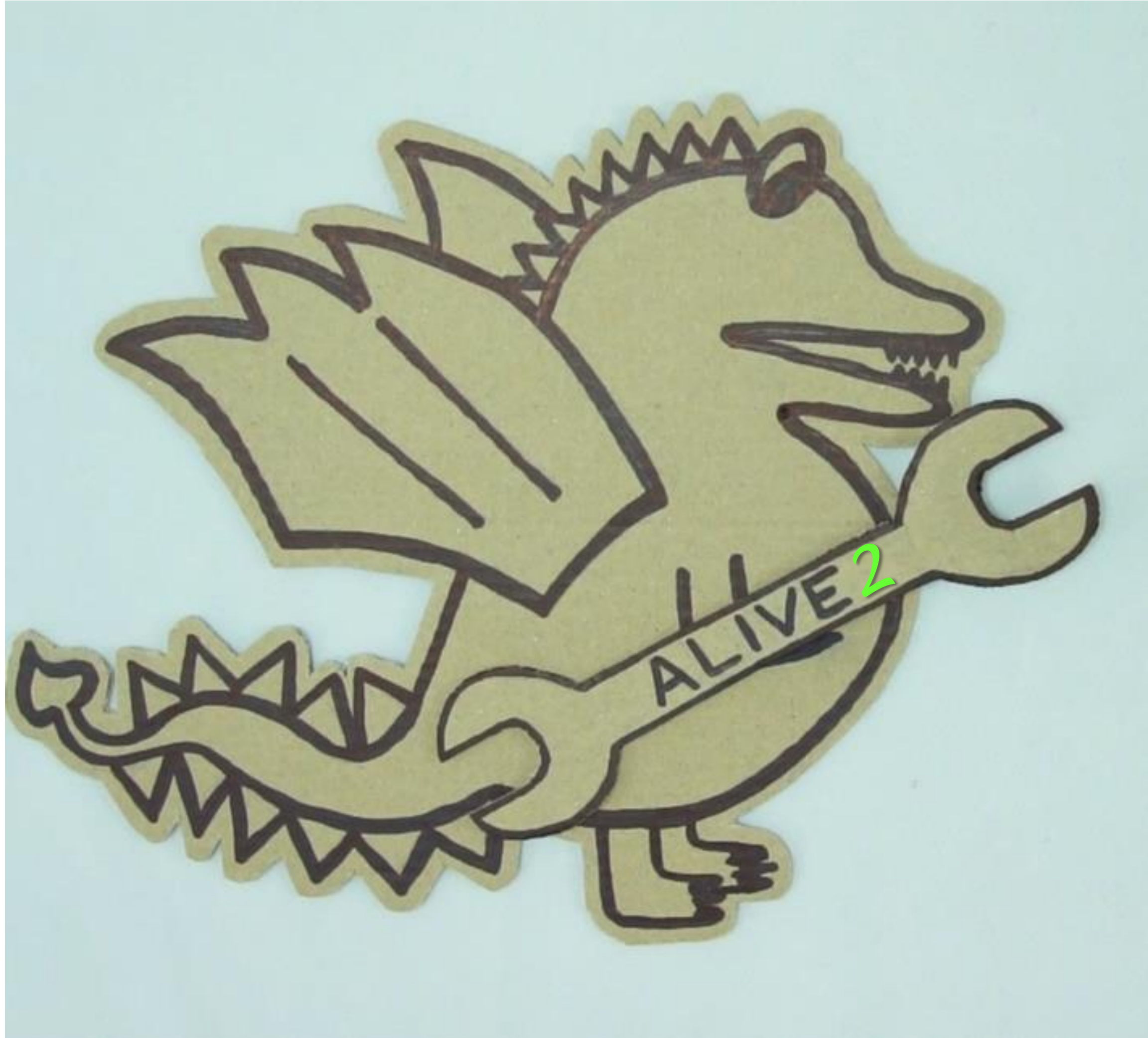
- Things are working fine; why bother?
- It looks expensive (it’s an investment for the long run)

- Semantics of compiler IRs is tricky business
  - Not enough research
  - Not enough knowledge

Alive wasn't enough

- Optimizations had to be written in Alive's DSL
- Alive only supported peephole optimizations
- C++ code generation wasn't productized





Alive2

TRANSLATION VALIDATION FOR LLVM

Alive2

- Supports all intra-procedural optimizations
- Ensures LLVM adheres to **a** specification
- Actively used by LLVM developers

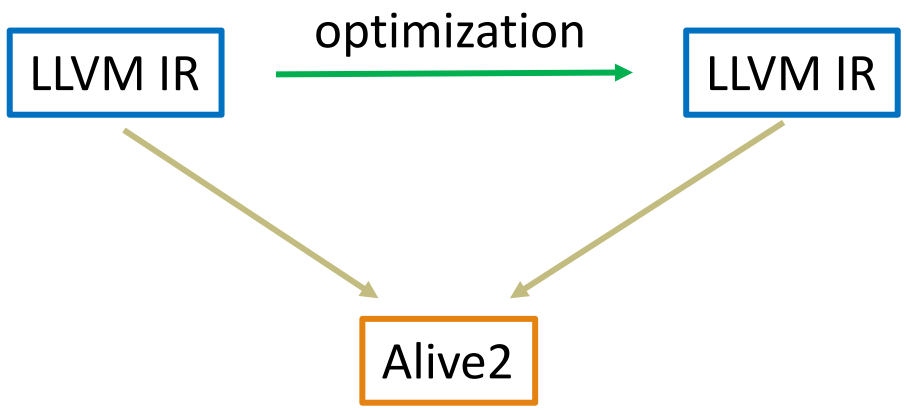
- Requires zero changes to LLVM
- Fully automatic
- Easy to use
- Identifies the optimization that miscompiled the code & produces minimal test case

https://alive2.llvm.org

https://github.com/AliveToolkit/alive2

Translation Validation

- *Was* the optimization correct?



- Correct
- Not correct + example
- Timeout

Validating LLVM with its own unit tests

- We found 100+ miscompilation bugs in LLVM through its own unit tests
  - Wait, what?

- The expected output of tests is generated automatically
  - Good for detecting regressions
  - Not so good to ensure developers read all of it!

- Anecdote: every time we implement a feature in Alive2, we find a bug in LLVM

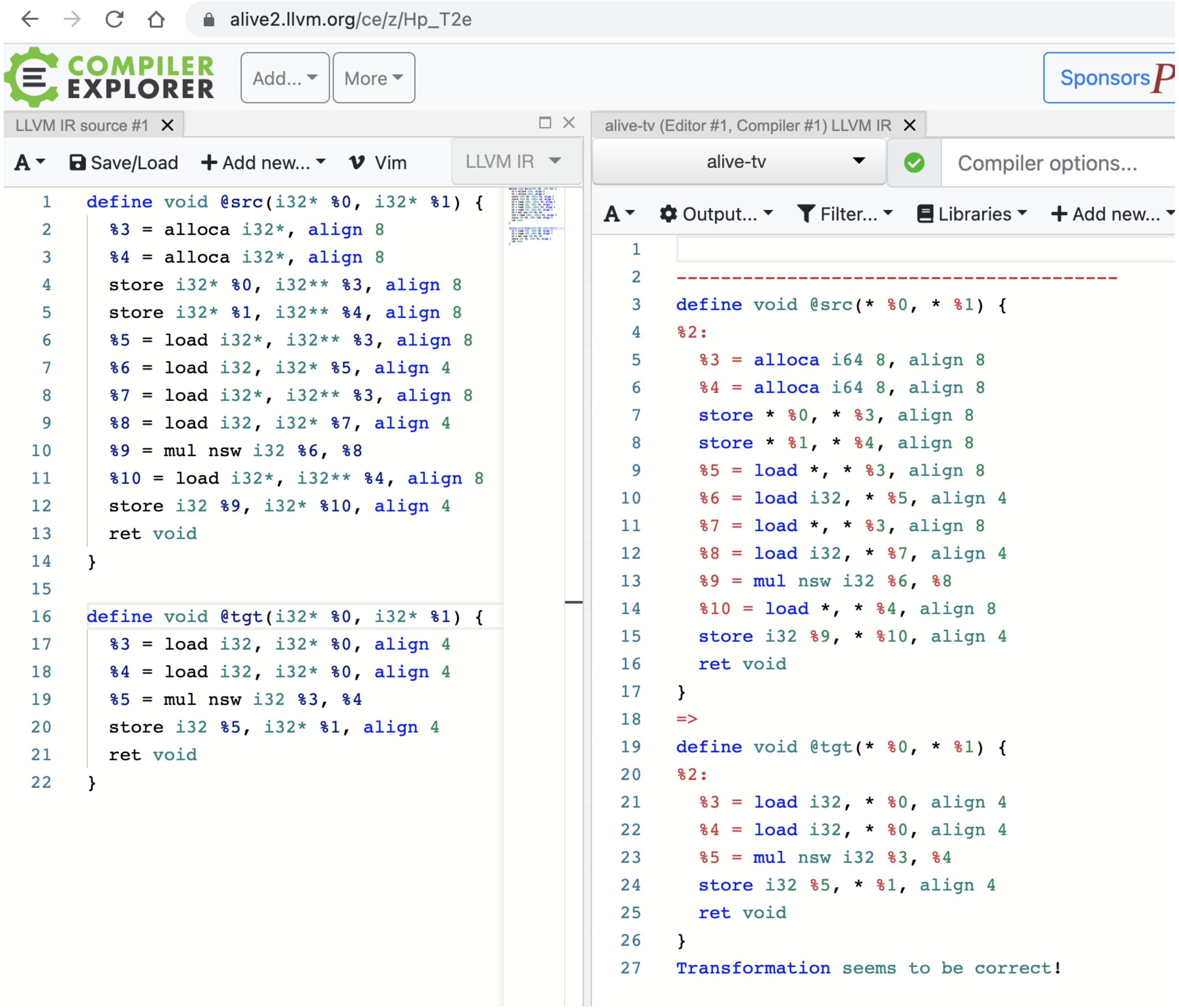
- Very important: allows us to validate our semantics of LLVM (aka “verifying the verifier”)
  - Plus experiment with different semantics

Validating LLVM by compiling C programs

- Found a lot of scalability issues in Alive2 & Z3

- Finds a lot of missing features in Alive2
  - Top 10 is very different from that of the unit tests!

- Finds extra bugs
  - The coverage of the test suite is very good for some optimizations, not great for others



Online tool is mandatory!

- Not everyone will spend time compiling the tool
- Easy share of inputs through permalinks
- Users educate each other

Alive2 in use

Phabricator

Differential > D91038

[LoopIdiom] Introduce 'left-shift until bittest' idiom

Accepted Public

Authored by **lebedev.ri** on Nov 9 2020, 4:49 AM.

The motivation here is the following inner loop in fp16/fp24 -> fp32 expander, that runs as part of the floating-point DNG decompression in RawSpeed library:

and we can prove that via alive2:

https://alive2.llvm.org/ce/z/7vQnji (ha nice, isn't it?)

```
while (!(fp32_fraction & (1 << 23))) {
  fp32_exponent -= 1;
  fp32_fraction <<= 1;
}
```

```
unsigned x = fp32_fraction;
unsigned bit = 23;
unsigned bitmask = 1U << bit;
unsigned mask = bitmask | (bitmask - 1);
unsigned x_masked = x & mask;
unsigned num_steps = __builtin_clz(x_masked) -
                     (CHAR_BIT*sizeof(x_masked)-bit-1);

fp32_exponent -= num_steps;
fp32_fraction <<= num_steps;
```

Side-effects: stress-test SMT solvers

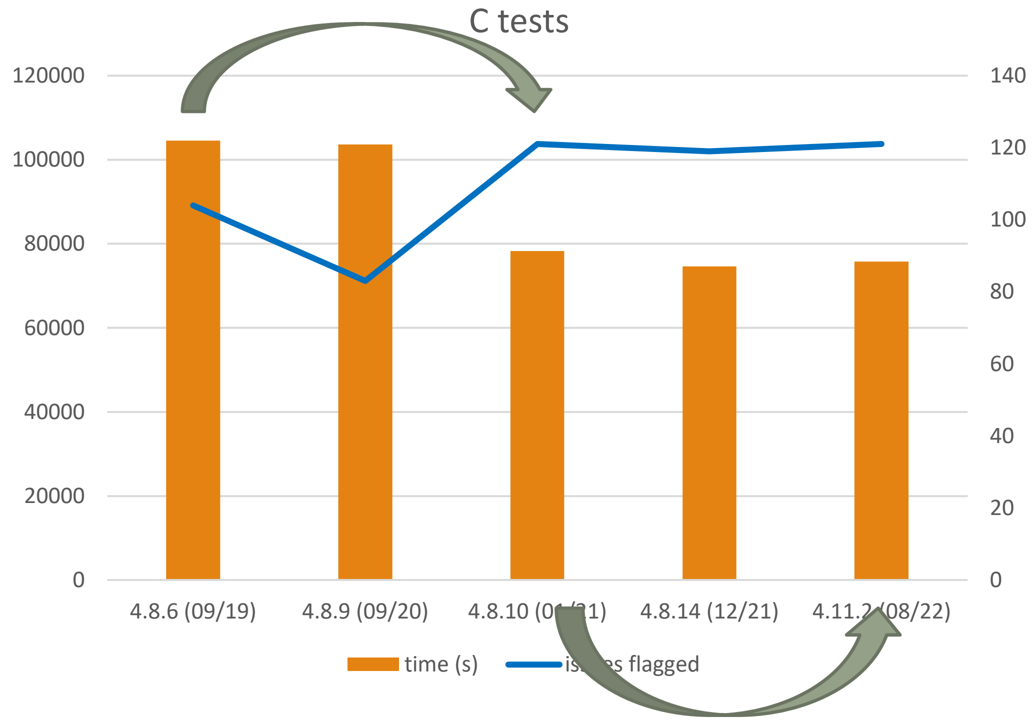
Bugs found in Z3

1. Incorrect bitblast for fprem (Z3Prover/z3#2369)
2. Bug in FPA w/ quantifiers (Z3Prover/z3#2596)
3. Bug in FPA w/ quantifiers (Z3Prover/z3#2631)
4. Crash in partial model mode (Z3Prover/z3#2652)
5. Crash when printing multi-precision integer (Z3Prover/z3#2761)
6. Bug with lambdas and quantified variables (Z3Prover/z3#2792)
7. Bug in MBQI (Z3Prover/z3#2822)
8. Bug with equality of arrays w/ lambdas (https://github.com/Z3Prover/z3/commit/0b14f1b6f6bb33b555bace93d644163987b0c5
9. Crash in FPA model construction (Z3Prover/z3#2865)
10. Crash in BV theory assertion (Z3Prover/z3#2878)
11. Assertion violation in SMT equality propagation (Z3Prover/z3#2879)
12. Assertion violation in qe_lite (https://github.com/Z3Prover/z3/commit/bb5edb7c653f93
13. SMT internalize doesn't respect the timeout (Z3Prover/z3#4192)
14. Unsoundness with smt.bv.size_reduce=true (Z3Prover/z3#6314)
15. Incorrect sort after lambda rewrite (Z3Prover/z3#6340)

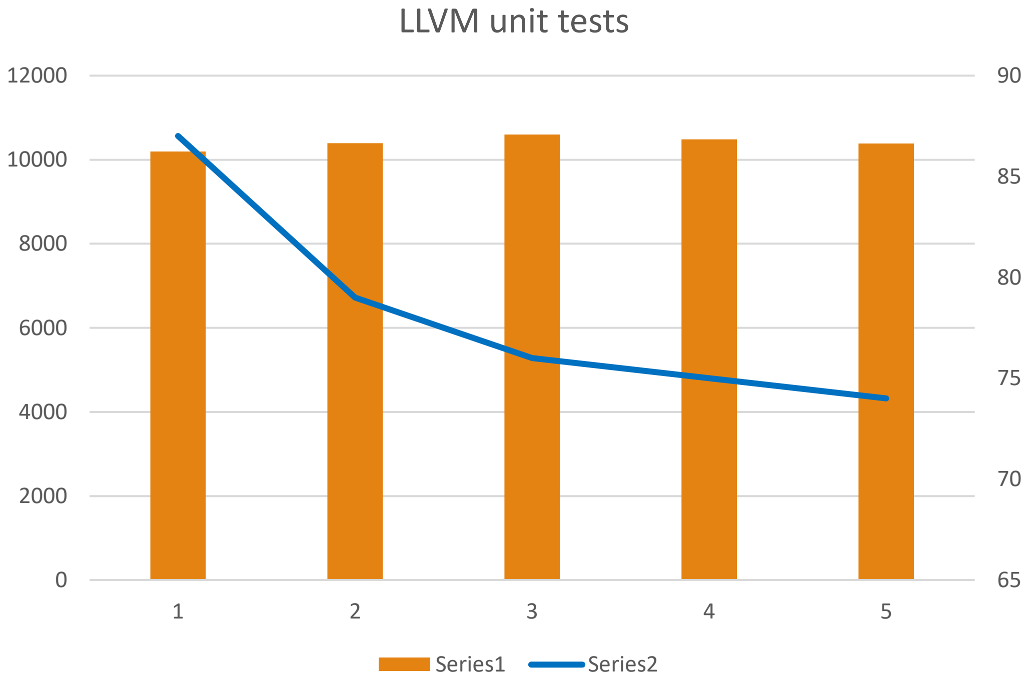
+ scalability issues in memory allocation, timeout mechanism, etc

SMT solvers improve all the time! Myth!

(We) Fixed exponential behavior with lambdas



3% speedup in 1.5 years



16% fewer bugs found 😟

Is LLVM correct already?

Is LLVM correct already?

- No!
- But it’s more correct than a decade ago*
- A few efforts ongoing:
  - Remove undef
  - Change semantics of load instructions (to remove undef)
  - Semantics of integer -> pointer cast
- Some theoretical issues still standing
  - Full semantics spec for LLVM doesn’t exist yet!

* I take no responsibility for this statement

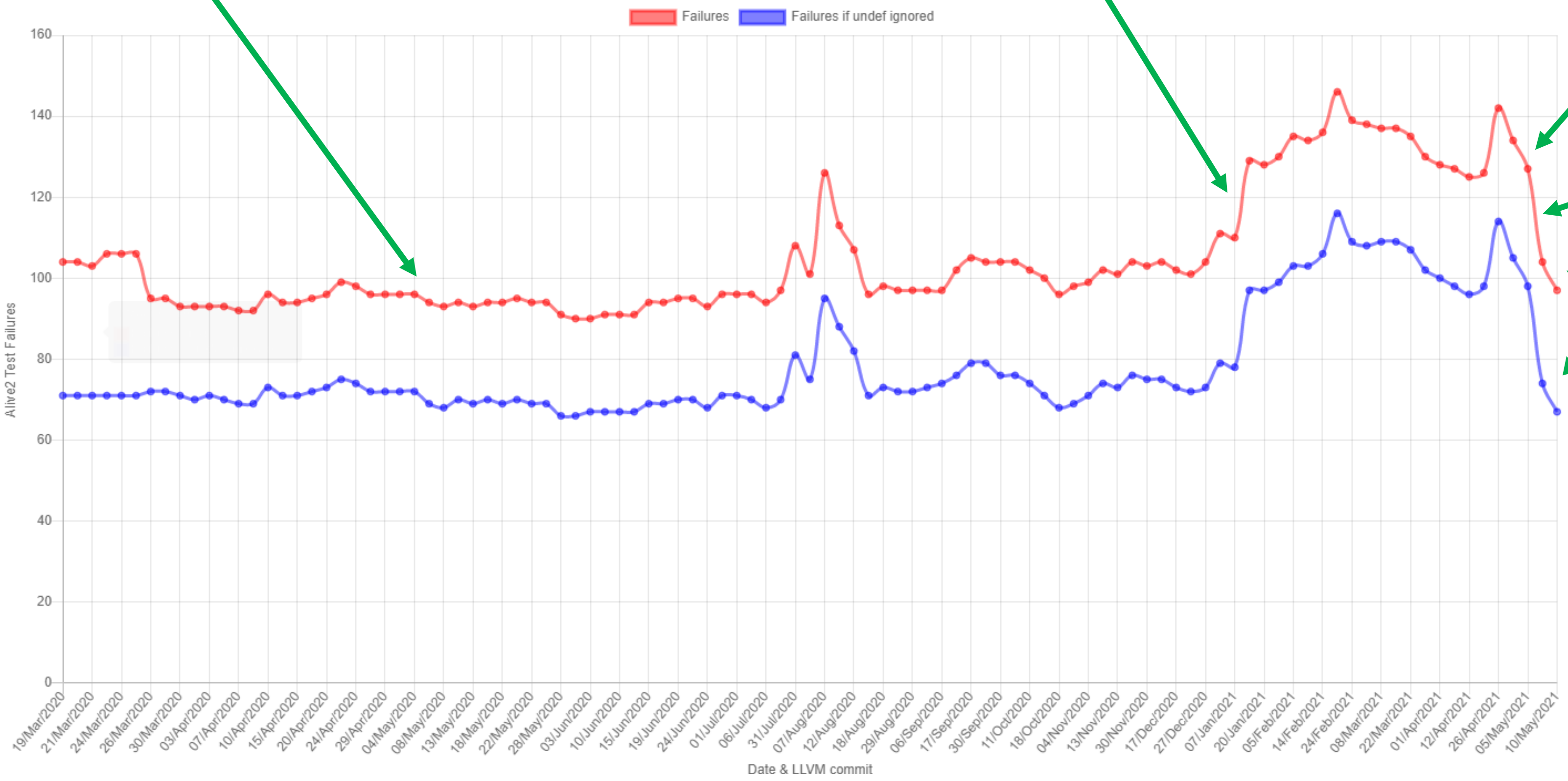
Continuous verification

Alive2 adds support for more LLVM features
Finds new bugs in LLVM; fixed at same pace

LLVM adds new unit tests
for select issues

Fix SimplifyCFG bug

Fix long-standing
InstCombine bug re
select instruction



Fix regression in Alive2
when passing null
pointers as arguments to
function calls

Still > 0 ☹

Conclusion

- Retrofitting soundness is very challenging
  - Requires patience, horror stories, education & marketing
  - Changing culture takes time
- Correctness is a never-ending job
  - Mandatory to have continuous validation
- Mandatory to have easy to use tools
  - Little or no change in developers' workflow
  - Web interfaces are fundamental to lower learning curve & increase adoption!
- Verifying a system requires fixing it first!

- Alive/Alive2 have been improving the correctness of LLVM for the past decade! ☺

Semantics of corner cases?

What's the result of:

and i8 %x, poison
and i1 false, poison
and i32 0, poison

'and' Instruction

Syntax:

```
<result> = and <ty> <op1>, <op2>   ; yields ty:result
```

Overview:

The 'and' instruction returns the bitwise logical and of its two operands.

Arguments:

The two arguments to the 'and' instruction must be integer or vector of integer values. Both arguments must have identical types.

Semantics:

The truth table used for the 'and' instruction is:

In0	In1	Out
0	0	0
0	1	0
1	0	0
1	1	1

Example:

```
<result> = and i32 4, %var         ; yields i32:result = 4 & %var
<result> = and i32 15, 40          ; yields i32:result = 8
<result> = and i32 4, 8            ; yields i32:result = 0
```

Semantics for select?

```
select %c, %a, %b
```

	UB if c poison + conditional poison	UB if c poison + poison if either a/b poison	Conditional poison + non-det choice if c poison	Conditional poison + poison if c poison	Poison if any of a/b/c poison
control-flow → select	✓		✓	✓	
select → control-flow	✓	✓			
select → arithmetic		✓			✓
select removal	✓	✓		✓	✓
select hoisting	✓	✓	✓		
easy movement			✓	✓	✓

Which one is the best and why?

Which one LLVM uses?