Faster SMT Solving via Constraint Transformation

Benjamin Mikek, Qirun Zhang

Georgia Institute of Technology

14 October 2023

Background: SMT Constraints

- SMT constraints encode first-order logic problems
- SMTLIB: a standard language for expressing constraints
- Many solvers: Z3, CVC5, etc.
- Bitvector constraints represent program logic
- Benchmark example: does multiplication overflow?

SAT: There is a variable assignment such that every assertion evaluates to true

UNSAT: There is not variable assignment such that every assertion evaluates to true

Theories: Bitvectors (machine integers), floating-point numbers, integers (linear and nonlinear), real numbers (linear and nonlinear)

Background: SMT Constraints

https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/-/blob/master/challenge/multiplyOverflow.smt2

```
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (=
              ((_ extract 63 32)
               (bvmul ((_ zero_extend 32) a)
                      ((_ zero_extend 32) b)))
              #x00000000)))
(assert (bvuge (bvudiv #xffffffff a) b))
(check-sat)
```

Bitvectors: Can multiplication a * b overflow, subject to a division constraint?

Unbounded integers: Does an underlying program terminate?

```
(declare-fun a () Int)
(declare-fun a2 () Int)
(declare-fun a3 () Int)
(declare-fun a5 () Int)
(assert (and (>= (+ (* a4 a2) (* a4 a3) (- 0 1)) 0)
             (>= (+ (* a5 a2) (* (- 0 1) a2)) 0)
             (>= (+ (* a5 a3) (* (- 0 1) a3)) 0)))
(check-sat)
```

https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NIA/-/blob/master/AProVE/aproveSMT1002369120799378097.smt2

Example: Can multiplication overflow?

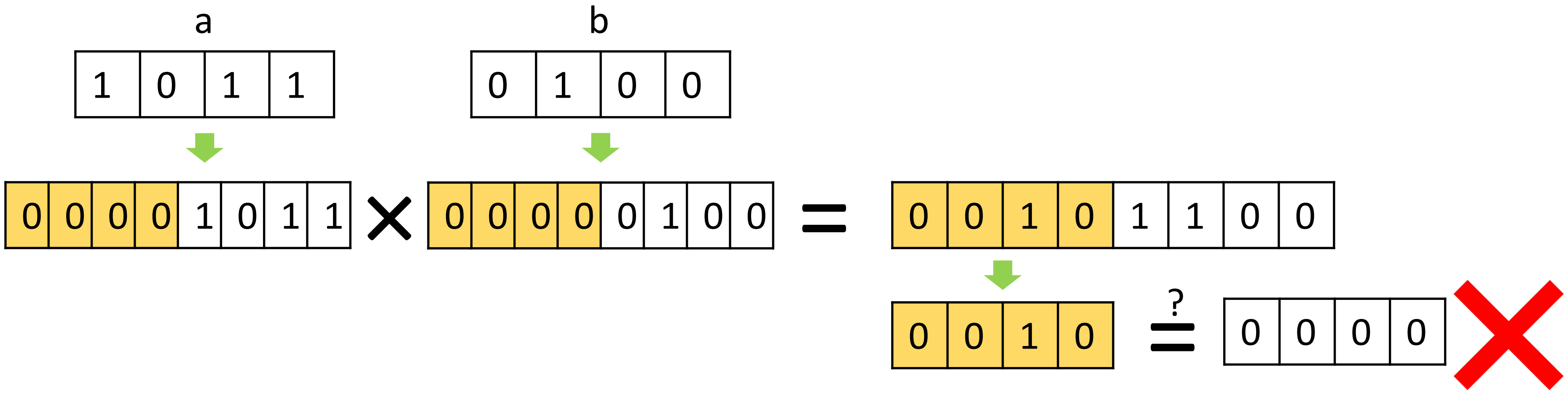
```
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (=
              ((_ extract 63 32)
               (bvmul ((_ zero_extend 32) a)
                      ((_ zero_extend 32) b)))
              #x00000000)))
(assert (bvuge (bvudiv #xffffffff a) b))
(check-sat)
```

https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/-/blob/master/challenge/multiplyOverflow.smt2

Example: Can multiplication overflow?

```
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (=
              ((_ extract 63 32)
               (bvmul ((_ zero_extend 32) a)
                      ((_ zero_extend 32) b)))
              #x00000000)))
(assert (bvuge (bvudiv #xffffffff a) b))
(check-sat)
```

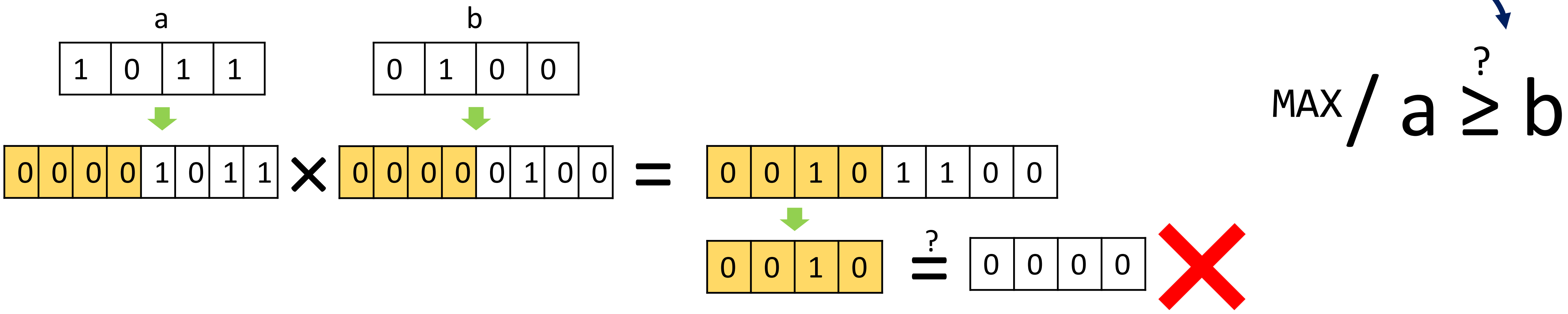
https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/-/blob/master/challenge/multiplyOverflow.smt2



Example: Can multiplication overflow?

```
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (=
                 ((_ extract 63 32)
                  (bvmul ((_ zero_extend 32) a)
                         ((_ zero_extend 32) b)))
                 #x00000000)))
(assert (bvuge (bvudiv #xffffffff a) b))
(check-sat)
```

https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/-/blob/master/challenge/multiplyOverflow.smt2



Example: Can multiplication overflow?

```
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (=
                  ((_ extract 63 32)
                   (bvmul ((_ zero_extend 32) a)
                          ((_ zero_extend 32) b)))
                  #x00000000)))
(assert (bvuge (bvudiv #xffffffff a) b))
(check-sat)
```

https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/-/blob/master/challenge/multiplyOverflow.smt2

The constraint is **UNSAT**

But Z3 takes ~5 minutes to solve it ☹

How can we speed up solving?

- Make a new solver – labor intensive, requires existing domain knowledge
- Extend an existing solver – requires detailed domain knowledge
- Key idea: Simplify constrains *before* applying a solver
- Two realizations:
  - Simplify bounded constraints using compiler optimization
  - Simplify unbounded constraints by making them bounded

Approach the First: SLOTting into Success

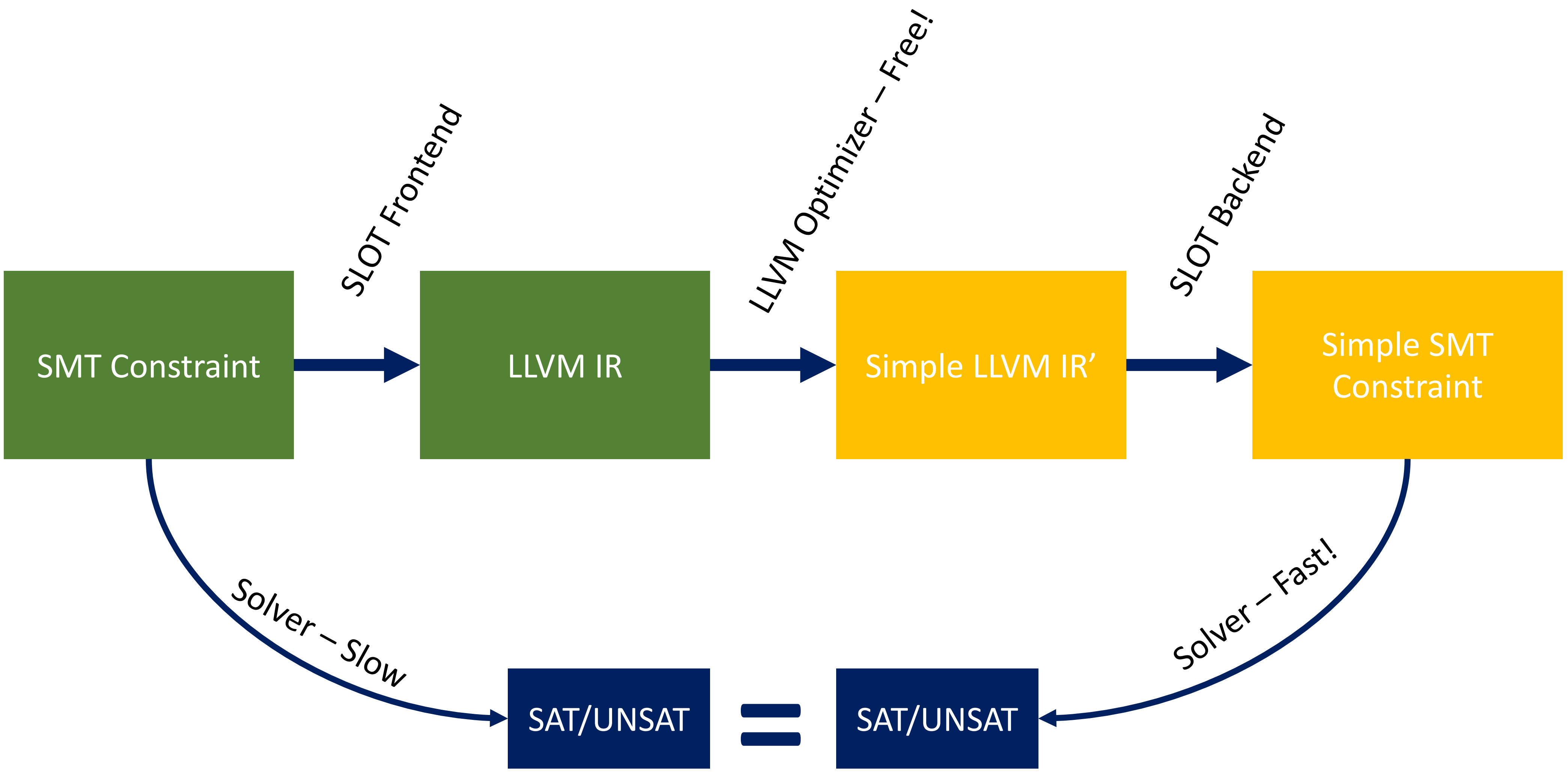
Simplifying bounded constraints with compiler optimization

- Compilers already implement many complex optimization passes
- Idea kernel: apply compiler optimizations to SMT problems
- Converting optimization passes into the SMT world would be complex
- Instead, we translate constraints to a compiler IR
- Implemented with LLVM: SMT-LLVM Optimizing Translation (SLOT) [FSE 2023]

SMTLIB and LLVM

	SMTLIB	LLVM
Bitvector types	One for each integer width n	One for each width up to 2^{23}
Floating point types	One for each integer pair eb, sb, but almost all in powers of 2	16-bit, 32-bit, 64-bit, 128-bit
Logic operations	and, or, xor, not, ite,=>, …	and, or, xor, select, …
Bitvector math operations	bvadd, bvsub, bvmul, bvsdiv, bvudiv, …	add, sub, mul, sdiv, udiv
Floating point math operations	fp.add, fp.sub, fp.div, fp.fma, …	fadd, fsub, fdiv, llvm.fma, …
Conversions	to_fp, to_fp_unsigned, fp.to_sbv, …	sitofp, uitofp, fptosi, …

Two SMT theories represent machine arithmetic: bitvectors ("integers") and floating-point numbers


SMT Constraint
SLOT Frontend
LLVM IR
LLVM Optimizer – Free!
Simple LLVM IR’
SLOT Backend
Simple SMT Constraint
Solver – Slow
SAT/UNSAT
SAT/UNSAT
Solver – Fast!

SLOT: Key Challenges

- SLOT has two parts: a front end and back end. Both have to preserve semantics
- LLVM is missing some SMT operations
- SMTLIB is missing some LLVM operations
- SMT constraints are *declarative*; LLVM is *imperative*
- Within assertions, order is dictated. But assertions are unordered

SLOT Translation

- Frontend: traverse the syntax tree of each SMT assertion
- Build an LLVM expression with the same semantics
- Most operations have 1-to-1 equivalents
- bvmul -> mul, bvadd -> add, fp.add -> fadd
- Some expressions are more complex and may involve undefined behavior handling

SLOT by Example: Checking for Overflow

```
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (=
              ((_ extract 63 32)
               (bvmul ((_ zero_extend 32) a)
                      ((_ zero_extend 32) b)))
              #x00000000)))
(assert (bvuge (bvudiv #xffffffff a) b))
(check-sat)
```

Applying SLOT's frontend

```
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (=
              ((_ extract 63 32)
               (bvmul ((_ zero_extend 32) a)
                      ((_ zero_extend 32) b)))
              #x00000000)))
(assert (bvuge (bvudiv #xffffffff a) b))
(check-sat)
```

```
define i1 @SMT(i32 %a, i32 %b) {
  %0 = zext i32 %b to i64
  %1 = zext i32 %a to i64
  %2 = mul i64 %1, %0
  %3 = lshr i64 %2, 32
  %4 = trunc i64 %3 to i32
  %5 = icmp eq i32 %4, 0
  %6 = xor i1 %5, true
  %7 = udiv i32 -1, %a
  %8 = icmp eq i32 %a, 0
  %9 = select i1 %8, i32 -1, i32 %7
  %10 = icmp uge i32 %9, %b
  %11 = and i1 %6, %10
  ret i1 %11
}
```

The LLVM function returns true if the inputs satisfy the underlying constraint.

Applying LLVM's optimizer

```
define i1 @SMT(i32 %a, i32 %b) {
  %0 = zext i32 %b to i64
  %1 = zext i32 %a to i64
  %2 = mul i64 %1, %0
  %3 = lshr i64 %2, 32
  %4 = trunc i64 %3 to i32
  %5 = icmp eq i32 %4, 0
  %6 = xor i1 %5, true
  %7 = udiv i32 -1, %a
  %8 = icmp eq i32 %a, 0
  %9 = select i1 %8, i32 -1, i32 %7
  %10 = icmp uge i32 %9, %b
  %11 = and i1 %6, %10
  ret i1 %11
}
```

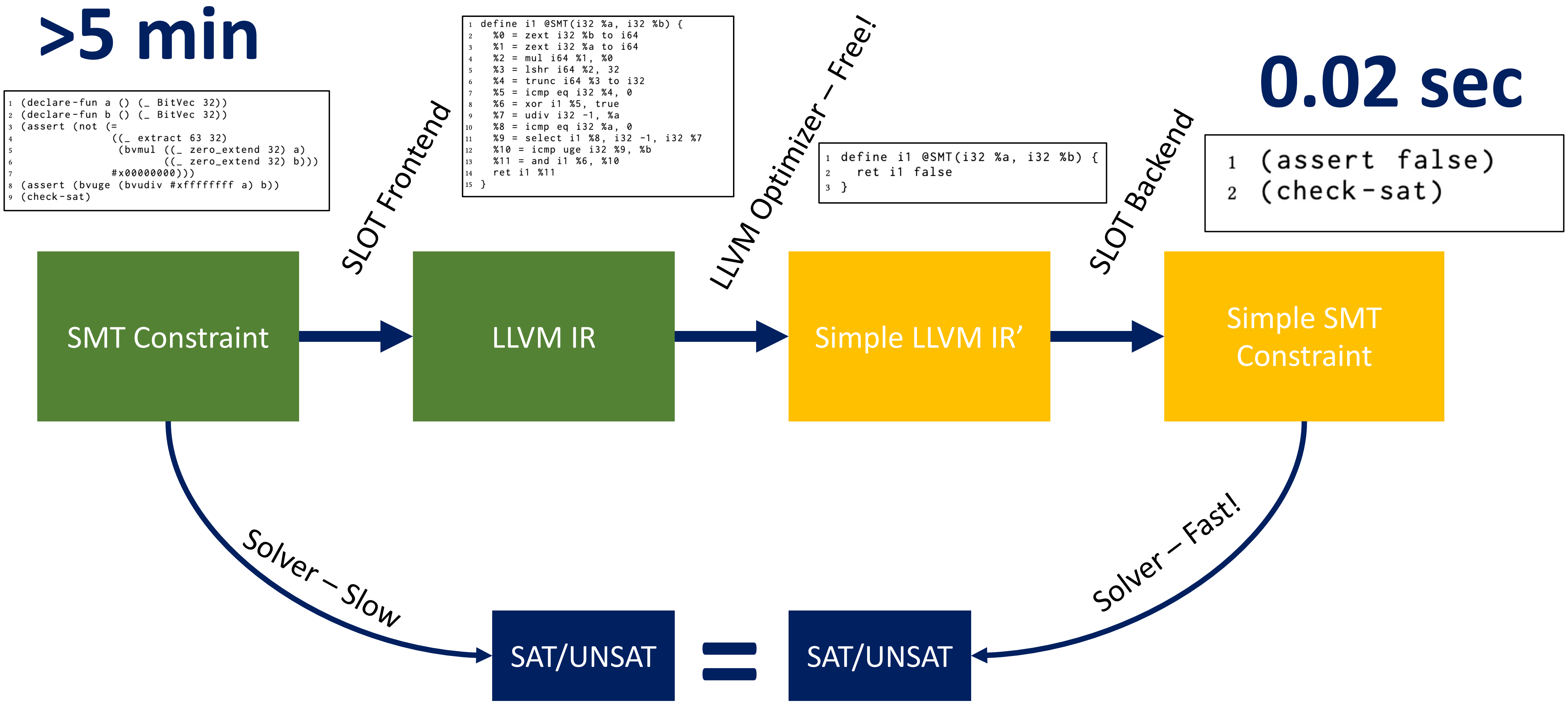
```
define i1 @SMT(i32 %a, i32 %b) {
  ret i1 false
}
```

Applying SLOT's backend

```
define i1 @SMT(i32 %a, i32 %b) {
  ret i1 false
}
```

```
(assert false)
(check-sat)
```

Can be solved almost instantly (0.02 seconds)!

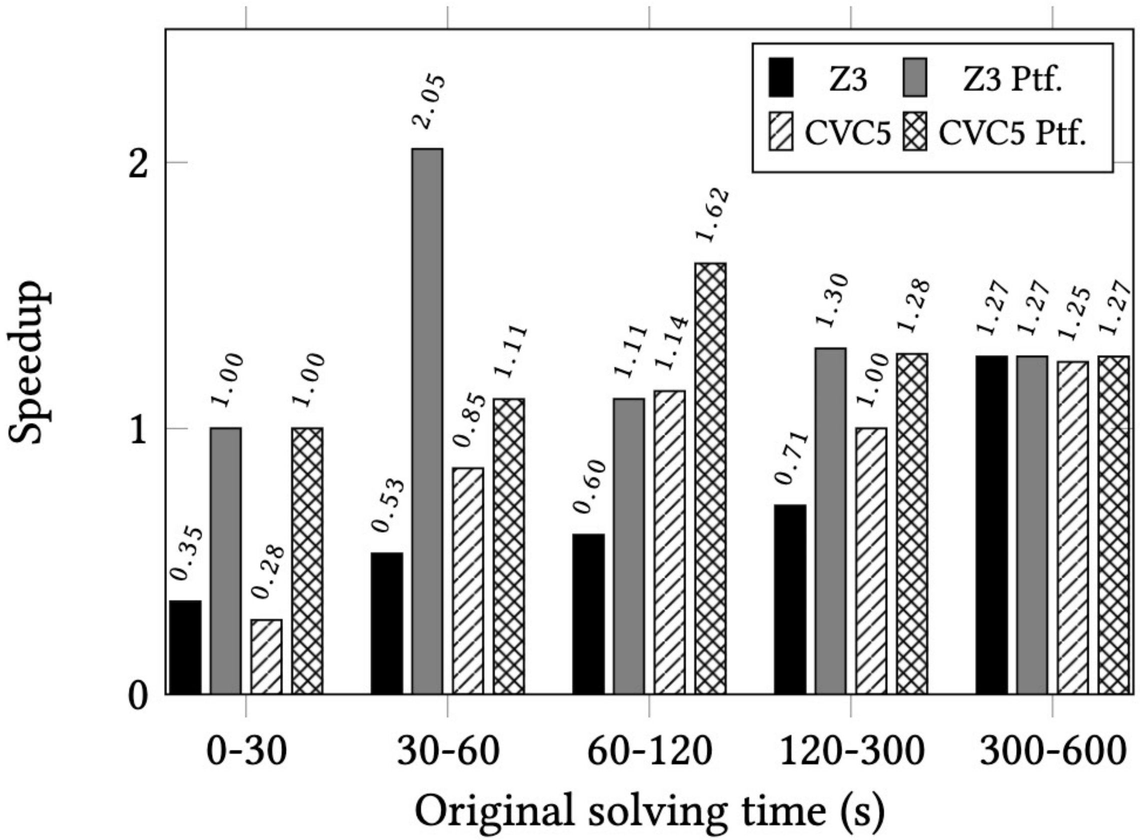


>5 min
(declare-fun a () (_ BitVec 32))
(declare-fun b () (_ BitVec 32))
(assert (not (=
((_ extract 63 32)
(bvmul ((_ zero_extend 32) a)
((_ zero_extend 32) b)))
#x00000000)))
(assert (bvuge (bvudiv #xffffffff a) b))
(check-sat)
SLOT Frontend
define i1 @SMT(i32 %a, i32 %b) {
%0 = zext i32 %b to i64
%1 = zext i32 %a to i64
%2 = mul i64 %1, %0
%3 = lshr i64 %2, 32
%4 = trunc i64 %3 to i32
%5 = icmp eq i32 %4, 0
%6 = xor i1 %5, true
%7 = udiv i32 -1, %a
%8 = icmp eq i32 %a, 0
%9 = select i1 %8, i32 -1, i32 %7
%10 = icmp uge i32 %9, %b
%11 = and i1 %6, %10
ret i1 %11
}
LLVM Optimizer – Free!
define i1 @SMT(i32 %a, i32 %b) {
ret i1 false
}
SLOT Backend
0.02 sec
(assert false)
(check-sat)
SMT Constraint
LLVM IR
Simple LLVM IR’
Simple SMT Constraint
Solver – Slow
Solver – Fast!
SAT/UNSAT
=
SAT/UNSAT

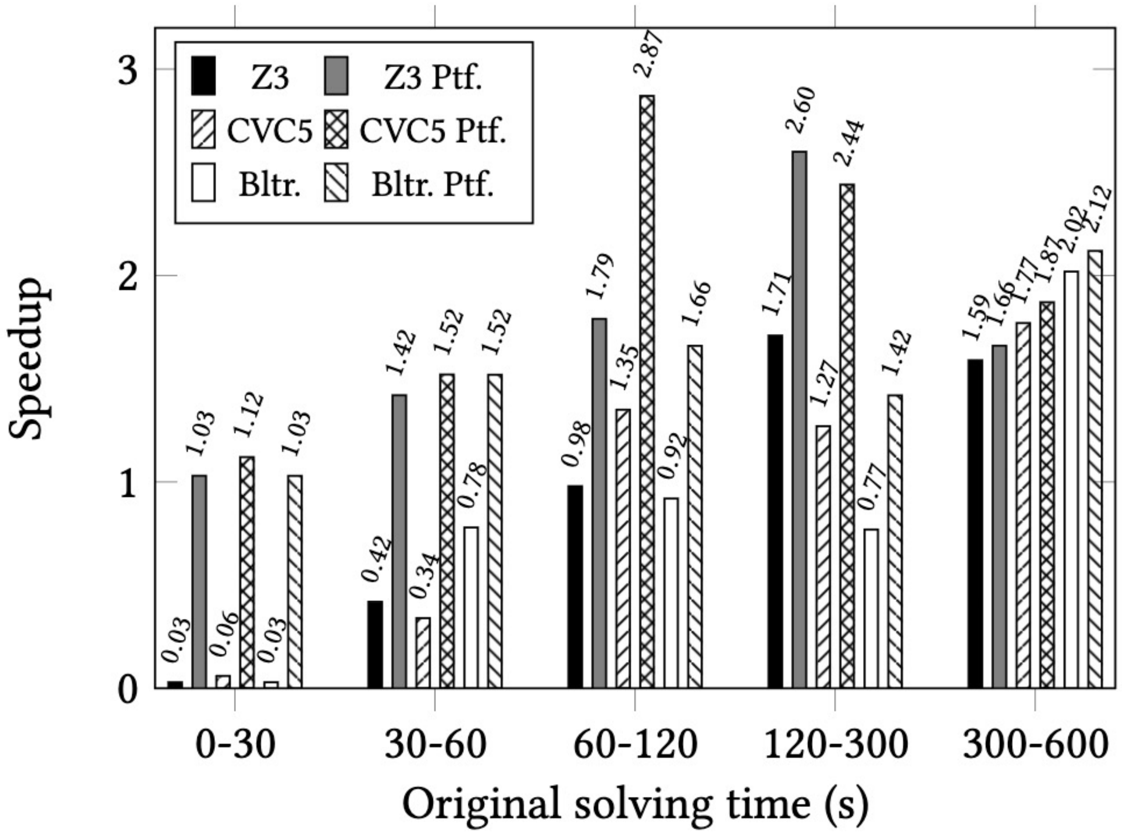
SLOT Results

- SLOT increases the number of solvable constraints at 600 second timeout by 10-20%
- SLOT can solve formulas for which three different solvers all fail (Z3, CVC5, Boolector)
- SLOT speeds up average solving time of large constraints by up to 3x
- SLOT and existing solvers act as a sieve

SLOT Results



(a) QF_FP



(c) QF_BV

SLOT Results

- The most effective optimizations passes are reassociate, instcombine, and global value numbering
- Solver developers can learn from these results about new optimizations to include in solvers

How many benchmarks does each pass change?

Pass	QF_FP	QF_BVFP	QF_BV
instcombine	99%	100%	78%
reassociate	78%	57%	26%
gvn	<1%	<1%	43%
sccp	0%	<1%	17%
dce	0%	<1%	17%
instsimplify	0%	<1%	16%
aggressive-instcombine	0%	0%	<1%
adce	0%	0%	0%

Which passes contribute the most speedup?

Pass	Count	Speedup without	Speedup with	Spread
reassociate	2,168	1.58×	2.02×	0.44
instcombine	4,031	1.49×	1.83×	0.34
gvn	3,816	1.51×	1.85×	0.34
instsimplify	1,562	1.74×	1.75×	0.22
sccp	1,360	1.71×	1.86×	0.15
dce	1,705	1.78×	1.68×	−0.10
agg-instcombine	8	1.75×	1.22×	−0.53

Approach the Second: From Unbounded to Bounded

Unbounded SMT Theories

- Unbounded theories are hard to solve
- Nonlinear integer arithmetic is undecidable
- Linear integer arithmetic and real arithmetic have no practical bounds on solutions
- In general, solvers perform better on bounded theories
- Idea kernel: transform unbounded constraints into equivalent bounded ones

Imposing Bounds

- Whenever we transform an unbounded variable into a bounded one, we loose information
- The transformation involves choosing sizes for the bounded variables (i.e., integer and floating-point widths)
- One option is to choose a constant size. But bigger widths slow down solving
- Therefore, we use abstract interpretation to estimate widths

Example: Imposing Bounds

```
1 (declare-fun a () Int)
2 (declare-fun b () Int)
3 (assert (>= a 15))
4 (assert (< (- a b) 0))
5 (check-sat)
```

Satisfying assignment:

a = 15

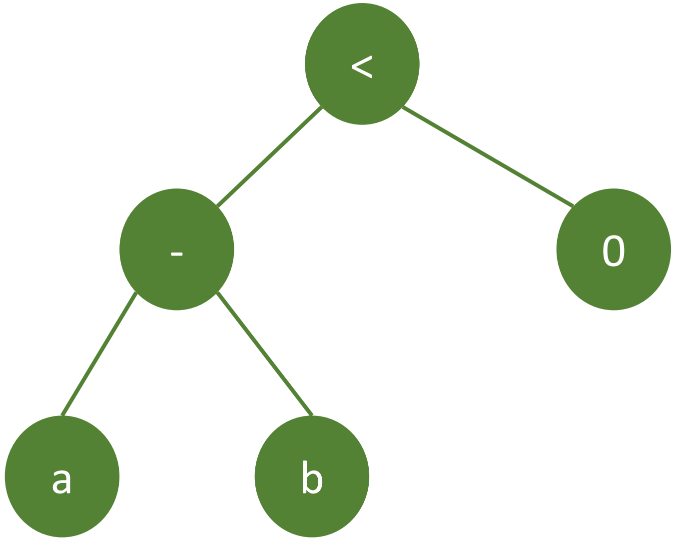
b = 16

- Choose a fixed width: 4 bits
- Maximum value for a variable is 15
- Line 3 is SAT
- Line 4 is UNSAT!

Example: Imposing Bounds

Use abstract interpretation. Maximum constant is 4 bits (15).

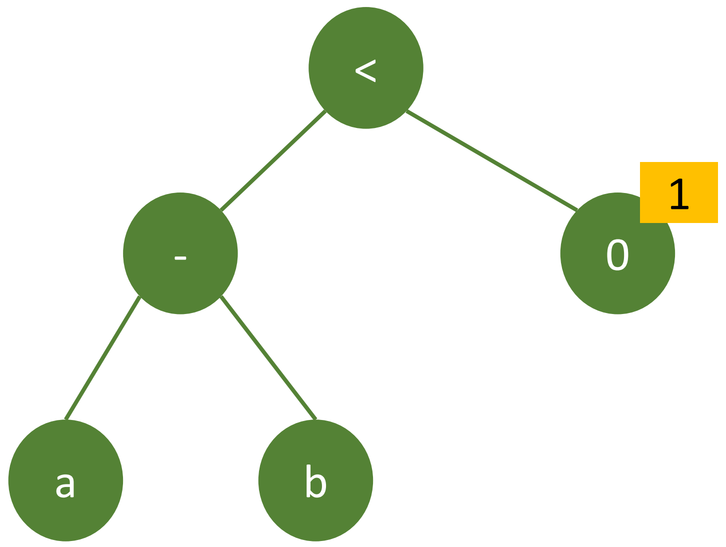
```
(declare-fun a () Int)
(declare-fun b () Int)
(assert (>= a 15))
(assert (< (- a b) 0))
(check-sat)
```



Example: Imposing Bounds

Use abstract interpretation. Maximum constant is 4 bits (15).

```
(declare-fun a () Int)
(declare-fun b () Int)
(assert (>= a 15))
(assert (< (- a b) 0))
(check-sat)
```

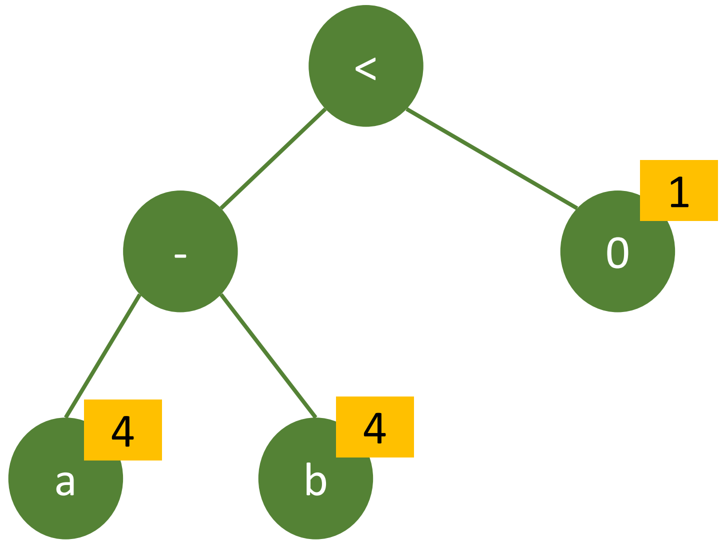


1: Constants are assigned their width

Example: Imposing Bounds

Use abstract interpretation. Maximum constant is 4 bits (15).

```
(declare-fun a () Int)
(declare-fun b () Int)
(assert (>= a 15))
(assert (< (- a b) 0))
(check-sat)
```

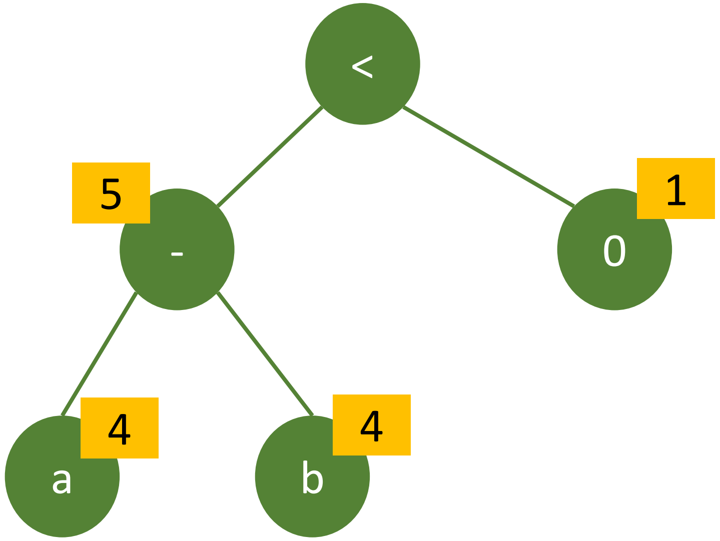


2: Variables are given the maximum constant width

Example: Imposing Bounds

Use abstract interpretation. Maximum constant is 4 bits (15).

```
(declare-fun a () Int)
(declare-fun b () Int)
(assert (>= a 15))
(assert (< (- a b) 0))
(check-sat)
```

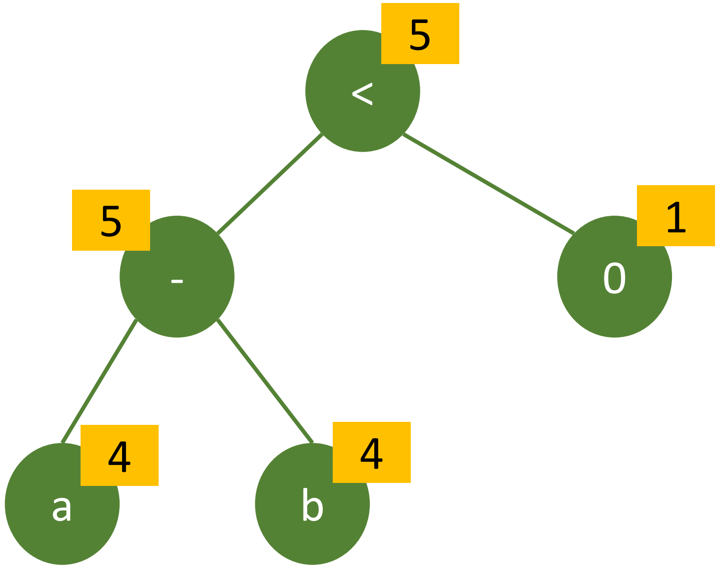


3: Subtraction: max(left, right) + 1

Example: Imposing Bounds

Use abstract interpretation. Maximum constant is 4 bits (15).

```
1 (declare-fun a () Int)
2 (declare-fun b () Int)
3 (assert (>= a 15))
4 (assert (< (- a b) 0))
5 (check-sat)
```

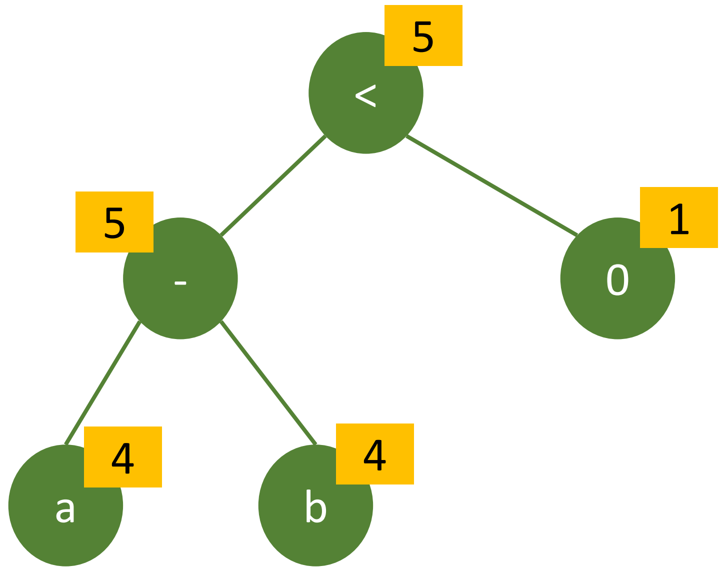


4: Comparison: max(left, right)

Example: Imposing Bounds

Use abstract interpretation. Maximum constant is 4 bits (15).

```
(declare-fun a () Int)
(declare-fun b () Int)
(assert (>= a 15))
(assert (< (- a b) 0))
(check-sat)
```



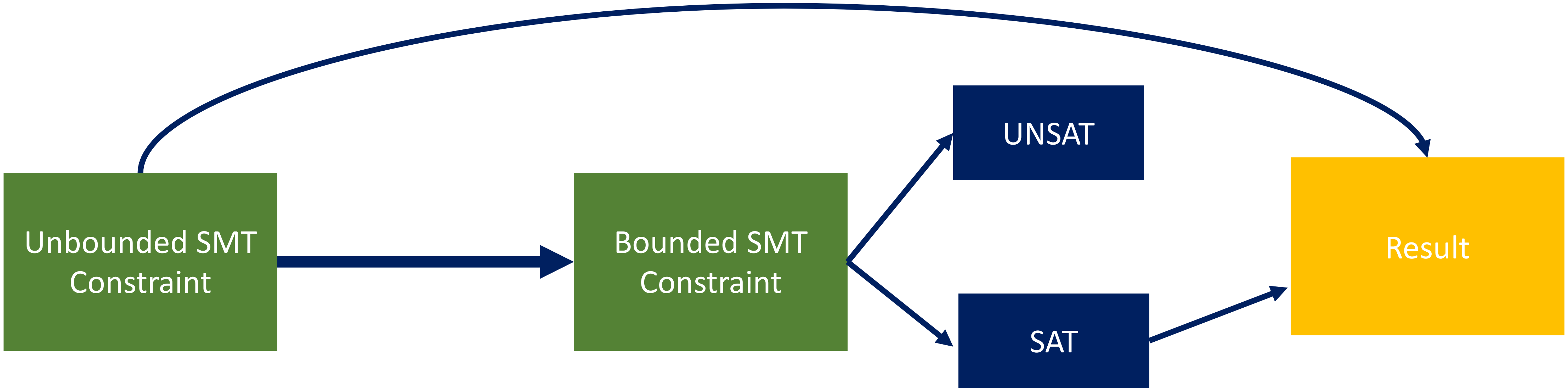
Satisfying assignment:

a = 15

b = 16

Choosing Bounds

- Even abstract interpretation cannot choose large enough bounds for all constraints
- If the final constraint is SAT, we are done
- If the final constraint is UNSAT, we must revert to the original



Unbounded theories: Results

- Z3 can perform 5x or more slower on unbounded constraints than similar bounded ones
- Speedups of 2.06x for NIA *on satisfiable cases*
- Speedups of 1.2x for NIA *on average*

Conclusion

- **Transforming SMT constraints *before applying a solver* reduces workload and can still simplify constraints**
- SLOT harnesses LLVM optimization to simplify constraints and speed up solving
- We transform unbounded constraints into bounded ones to improve performance
- Any questions?