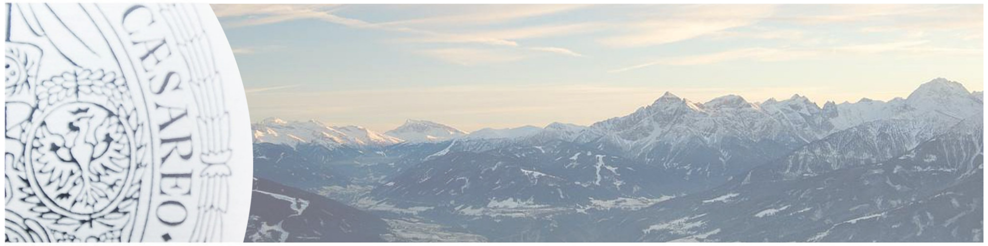
universität innsbruck



# SAT and SMT Solving


**Sarah Winkler**

KRDB
Department of Computer Science
Free University of Bozen-Bolzano


lecture 10
WS 2022

## Outline

- Summary of Last Week
- Bit Vectors



## Satisfiability in Linear Integer Arithmetic

**Fact**
for every LIA problem can compute bounds to get equisatisfiable bounded problem, so BranchAndBound terminates

**Definition (Cut)**
given solution $\alpha$ to problem over $\mathbb{Q}^n$, cut is inequality $a_1x_1 + \cdots + a_nx_n \leqslant b$ which is not satisfied by $\alpha$ but by every $\mathbb{Z}^n$-solution



## Gomory Cuts

**Gomory Cuts: Assumptions**

- DPLL($T$) Simplex returned solution $\alpha$ and final tableau $A$ such that

$$A\overline{x}_I = \overline{x}_D \qquad\qquad l_i \leqslant x_i \leqslant u_i$$

- for some $x_i \in D$ have $\alpha(x_i) \notin \mathbb{Z}$ and for all $x_j \in I$ value $\alpha(x_j)$ is $l_j$ or $u_j$

**Notation**

- write $c = \alpha(x_i) - \lfloor \alpha(x_i) \rfloor$
- split independent variables $I$ into $L = \{x_j \mid \alpha(x_j) = l_j\}$ and $U = \{x_j \mid \alpha(x_j) = u_j\}$
- $L^+ = \{\, x_j \in L \mid \alpha(x_j){=}l_j \text{ and } A_{ij} \geqslant 0 \,\}$ $\quad U^+ = \{\, x_j \in U \mid \alpha(x_j){=}u_j \text{ and } A_{ij} \geqslant 0 \,\}$
  $L^- = \{\, x_j \in L \mid \alpha(x_j){=}l_j \text{ and } A_{ij} < 0 \,\}$ $\quad U^- = \{\, x_j \in U \mid \alpha(x_j){=}u_j \text{ and } A_{ij} < 0 \,\}$

**Lemma (Gomory Cut)**

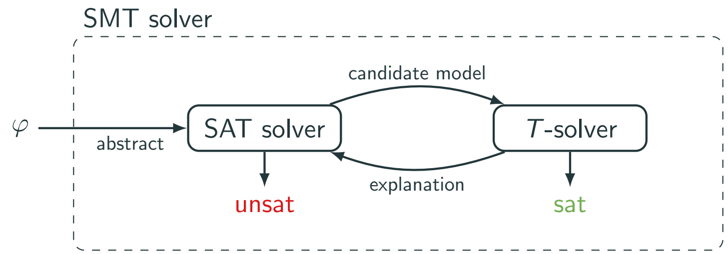
$$\sum_{x_j \in L^+} \frac{A_{ij}}{1-c}(x_j - l_j) - \sum_{x_j \in U^-} \frac{A_{ij}}{1-c}(u_j - x_j) - \sum_{x_j \in L^-} \frac{A_{ij}}{c}(x_j - l_j) + \sum_{x_j \in U^+} \frac{A_{ij}}{c}(u_j - x_j) \geqslant 1$$

## Outline

- Summary of Last Week
- Bit Vectors

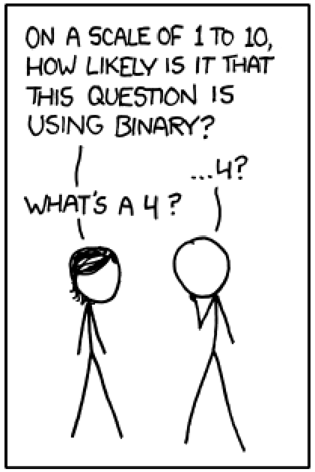


## Theories in SMT Solving




| Theory $T$ | $T$-solving method | |
|---|---|---|
| ▶ equality logic | equality graphs | ✓ |
| ▶ equality + uninterpreted functions (EUF) | congruence closure | ✓ |
| ▶ linear real arithmetic (LRA) | Simplex | ✓ |
| ▶ linear integer arithmetic (LIA) | Simplex + cuts + bounds | ✓ |
| ▶ bitvectors (BV) | bit-blasting | |
| ▶ arrays (A) | | |






**Disclaimer**
rest of lecture assumes brains in binary mode



## Flashback to the Binary World

### Binary representation

- $k$-bit representation of non-negative number $n$

| $n \bmod 2^{k-1}$ | $n \bmod 2^{k-2}$ | $\cdots$ | $n \bmod 2^1$ | $n \bmod 2^0$ |
|---|---|---|---|---|

denoted $n_k$

- …of negative number $-n$ is $(\sim n_k) + 1_k$ 2-complement

### Operations on binary numbers (for fixed bitwidth)

- $\&$, $|$, and $\sim$ are bitwise and, or, and negation
- $+$, $-$, $\times$ are addition, subtraction, and multiplication (with overflow)

### Example

| | |
|---|---|
| ▶ $5_4$ | 0 1 0 1 |
| ▶ $5_4 \ \& \ 13_4$ | 0 1 0 1 |
| ▶ $5_4 + 1_4$ | 0 1 1 0 |
| ▶ $-5_4$ | 1 0 1 1 |
| ▶ $13_4$ | 1 1 0 1 |
| ▶ $5_4 \mid 13_4$ | 1 1 0 1 |
| ▶ $5_4 + 13_4$ | 0 0 1 0 |
| ▶ $-1_4$ | 1 1 1 1 |
| ▶ $\sim 5_4$ | 1 0 1 0 |

## Comparison operators

- $<_u$, $\leqslant_u$, …considers operands as unsigned numbers
- $<_s$, $\leqslant_s$, …considers operands as signed numbers

### Example

- $3_4$ $\boxed{0\;0\;1\;1}$ $<_u$ $\boxed{1\;1\;0\;1}$ $13_4$
- $3_4$ $\boxed{0\;0\;1\;1}$ $\not<_s$ $\boxed{1\;1\;0\;1}$ $-3_4$

## Binary operations and sign

- $+$, $-$, $\times$ work independently of whether operands are considered signed
- division and modulo depend on signedness: distinguish $\div_u$, $\%_u$ and $\div_s$, $\%_s$

### Example

$$
\begin{array}{rccc}
3_4 & & \boxed{0\;0\;1\;1} & 3_4 \\
10_4 & + & \boxed{1\;0\;1\;0} & -6_4 \\
\hline
13_4 & & \boxed{1\;1\;0\;1} & -3_4
\end{array}
\qquad
\begin{array}{ccc}
 & \boxed{1\;0\;1\;0} & 10_4 \\
\div_u & \boxed{0\;0\;1\;1} & 3_4 \\
\hline
 & \boxed{0\;0\;1\;1} & 3_4
\end{array}
\qquad
\begin{array}{ccc}
 & \boxed{1\;0\;1\;0} & -6_4 \\
\div_s & \boxed{0\;0\;1\;1} & 3_4 \\
\hline
 & \boxed{1\;1\;1\;0} & -2_4
\end{array}
$$



## Definition (Bit Vector Theory)

for bitwidth $k$, theory $\mathrm{BV}_k$ is given by

- signature
  - constants $n_k$ for all $n < 2^k$
  - binary function symbols $+$, $-$, $\times$, $\div_u$, $\div_s$, $\%_u$, $\%_s$, $\ll$, $\gg_u$, $\gg_s$, $\&$, $|$, $\hat{}$
  - unary function symbols $-$ and $\sim$
  - predicates $=$, $\neq$, $\geqslant_u$, $\geqslant_s$, $>_u$, and $>_s$
- axioms are equality axioms plus all correct arithmetic, comparison, and bit operations on binary numbers with $k$ bits

## Remark

- theories $\mathrm{BV}_{k_1}, \ldots \mathrm{BV}_{k_m}$ of different bit widths can be combined
- can also use binary $::$ for concatenation and unary $(\cdot)[i{:}j]$ to extract bits

## Definitions

- variable $\mathbf{x}_k$ is list of length $k$ of propositional variables $x_{k-1} \ldots x_2 x_1 x_0$
- valuation $v$ assigns element in $\{\mathsf{T},\mathsf{F}\}^k$ to variable $\mathbf{x}_k$,
  (usually written as binary number with $k$ bits)



### Examples

- $\mathbf{x}_4 + \mathbf{y}_4 = \mathbf{7}_4$
  satisfiable: $v(\mathbf{x}_4) = \mathbf{4}_4$ and $v(\mathbf{y}_4) = \mathbf{3}_4$
- $\mathbf{x}_4 + \mathbf{2}_4 <_u \mathbf{x}_4$
  overflow semantics!
  satisfiable: $v(\mathbf{x}_4) = \mathbf{15}_4$
- $(\mathbf{x}_4 \times \mathbf{y}_4 = \mathbf{6}_4) \wedge (\mathbf{x}_4 \;\&\; \mathbf{y}_4 = \mathbf{2}_4)$
  satisfiable: $v(\mathbf{x}_4) = \mathbf{3}_4$, $v(\mathbf{y}_4) = \mathbf{2}_4$
- $(\mathbf{x}_4 \geqslant_u \mathbf{y}_4) \wedge \neg(\mathbf{x}_4 \geqslant_s \mathbf{y}_4)$
  satisfiable: $v(\mathbf{x}_4) = \mathbf{8}_4$, $v(\mathbf{y}_4) = \mathbf{0}_4$
- $(\mathbf{x}_4 \ll \mathbf{2}_4 = \mathbf{12}_4) \wedge (\mathbf{x}_4 + \mathbf{1}_4 = \mathbf{12}_4)$
  satisfiable: $v(\mathbf{x}_4) = \mathbf{11}_4$
- $(\mathbf{8}_4 \gg_u \mathbf{2}_4 = \mathbf{2}_4) \wedge (\mathbf{8}_4 \gg_s \mathbf{2}_4 = \mathbf{14}_4)$
  holds
  $\gg_u$ shifts in 0s,
  $\gg_s$ shifts in sign bits
- $(\mathbf{x}_4[1{:}0] :: \mathbf{x}_4[3{:}2] = \mathbf{2}_4) \wedge (\mathbf{y}_4[2{:}0] = \mathbf{7}_3)$
  $\mathbf{x}[i{:}j]$ denotes $x_i \ldots x_j$
  and $::$ is concatenation
  satisfiable: $v(\mathbf{x}_4) = \mathbf{8}_4$ and $v(\mathbf{y}_4) = \mathbf{15}_4$



## Notation for Constants

- $\mathbf{n}_k$ is binary representation of $n$ in $k$ bits
- $\mathbf{xn}_k$ is binary representation of hexadecimal $n$ in $k$ bits

### Example

- $\mathbf{0}_1$, $\mathbf{3}_2$, $\mathbf{10}_4$, $\mathbf{1024}_{32}$, …
- $\mathbf{x0}_4$, $\mathbf{xa}_4$, $\mathbf{xb0}_8$, $\mathbf{x11cf}_{16}$, $\mathbf{xffffffff}_{32}$, …

### More examples

negation uses two's complement

- $-\mathbf{a}_4 = \mathbf{a}_4$
  satisfiable: $v(\mathbf{a}_4) = -\mathbf{8}_4 = \mathbf{x8}_4$
- $\mathbf{a}_8 \div_u \mathbf{b}_8 = \mathbf{a}_8 \gg_u \mathbf{1}_8$
  satisfied by powers of 2 (and 0)
  satisfiable: $v(\mathbf{a}_8) = \mathbf{8}_8$ and $v(\mathbf{b}_8) = \mathbf{2}_8$
- $\mathbf{a}_8 \;\&\; (\mathbf{a}_8 - \mathbf{1}_8) = \mathbf{0}_8$
  satisfiable: $v(\mathbf{a}_8) = \mathbf{8}_8$ or $\mathbf{x0}_8$, $\mathbf{x1}_8$, $\mathbf{x2}_8$, $\mathbf{x4}_8$, $\mathbf{x8}_8$, $\mathbf{x10}_8$, $\mathbf{x20}_8$, $\mathbf{x40}_8$, $\mathbf{x80}_8$

## Remarks

- theory is decidable because carrier is finite
- common decision procedures use translation to SAT (bit blasting)
  - eager: no DPLL($T$), bit-blast entire formula to SAT problem
  - lazy: second SAT solver as BV theory solver, bit-blast only BV atoms
- solvers heavily rely on preprocessing via rewriting

## Example (Preprocessing)

$$\begin{aligned}\mathbf{x}_1 \neq \mathbf{0}_1 \wedge (\mathbf{y}_3 :: \mathbf{x}_1) \mathbin{\%_u} \mathbf{2}_4 = \mathbf{0}_4 &\to \mathbf{x}_1 = \mathbf{1}_1 \wedge (\mathbf{y}_3 :: \mathbf{x}_1) \mathbin{\%_u} \mathbf{2}_4 = \mathbf{0}_4 \\ &\to (\mathbf{y}_3 :: \mathbf{1}_1) \mathbin{\%_u} \mathbf{2}_4 = \mathbf{0}_4 \to \mathsf{F}\end{aligned}$$

## Definition (Bit Blasting: Formulas)

bit blasting transformation $\mathbf{B}$ transforms BV formula into propositional formula:

$$\mathbf{B}(\varphi \vee \psi) = \mathbf{B}(\varphi) \vee \mathbf{B}(\psi)$$

$$\mathbf{B}(\varphi \wedge \psi) = \mathbf{B}(\varphi) \wedge \mathbf{B}(\psi)$$

$$\mathbf{B}(\neg\varphi) = \neg\mathbf{B}(\varphi)$$

bit blasting $\mathbf{B}_t$ for term $t$ returns (result $u$, side condition $\varphi$)

$$\mathbf{B}(t_1 \; rel \; t_2) = \mathbf{B}_r(u_1 \; rel \; u_2) \wedge \varphi_1 \wedge \varphi_2 \qquad \text{if } \mathbf{B}_t(t_1) = (u_1, \varphi_1) \text{ and } \mathbf{B}_t(t_2) = (u_2, \varphi_2)$$

$\mathbf{B}_r$ transforms atom into propositional formula



## Definition (Bit Blasting: Atoms)

for bit vectors $\mathbf{x}_k$ and $\mathbf{y}_k$ set

- equality

$$\mathbf{B}_r(\mathbf{x}_{k+1} = \mathbf{y}_{k+1}) = (x_k \leftrightarrow y_k) \wedge \cdots \wedge (x_1 \leftrightarrow y_1) \wedge (x_0 \leftrightarrow y_0)$$

- inequality

$$\mathbf{B}_r(\mathbf{x}_{k+1} \neq \mathbf{y}_{k+1}) = (x_k \oplus y_k) \vee \cdots \vee (x_1 \oplus y_1) \vee (x_0 \oplus y_0)$$

- unsigned greater-than or equal

$$\mathbf{B}_r(\mathbf{x}_1 \geqslant_u \mathbf{y}_1) = y_0 \to x_0$$

$$\mathbf{B}_r(\mathbf{x}_{k+1} \geqslant_u \mathbf{y}_{k+1}) = (x_k \wedge \neg y_k) \vee ((x_k \leftrightarrow y_k) \wedge \mathbf{B}(\mathbf{x}[k-1{:}0] \geqslant \mathbf{y}[k-1{:}0]))$$

- unsigned greater-than

$$\mathbf{B}(\mathbf{x}_k >_u \mathbf{y}_k) = \mathbf{B}(\mathbf{x}_k \geqslant \mathbf{y}_k) \wedge \mathbf{B}(\mathbf{x}_k \neq \mathbf{y}_k)$$



## Definition (Bit Blasting: Bitwise Operations)

for bit vectors $\mathbf{x}_k$ and $\mathbf{y}_k$ use fresh variable $\mathbf{z}_k$ and set

- bitwise and

$$\mathbf{B}_t(\mathbf{x}_k \mathbin{\&} \mathbf{y}_k) = (\mathbf{z}_k, \varphi) \qquad \varphi = \bigwedge_{i=0}^{k-1} z_i \leftrightarrow (x_i \wedge y_i)$$

- bitwise or

$$\mathbf{B}_t(\mathbf{x}_k | \mathbf{y}_k) = (\mathbf{z}_k, \varphi) \qquad \varphi = \bigwedge_{i=0}^{k-1} z_i \leftrightarrow (x_i \vee y_i)$$

- bitwise exclusive or

$$\mathbf{B}_t(\mathbf{x}_k \mathbin{\hat{}} \mathbf{y}_k) = (\mathbf{z}_k, \varphi) \qquad \varphi = \bigwedge_{i=0}^{k-1} z_i \leftrightarrow (x_i \oplus y_i)$$

- bitwise negation

$$\mathbf{B}_t(-\mathbf{x}_k) = (\mathbf{z}_k, \varphi) \qquad \varphi = \bigwedge_{i=0}^{k-1} z_i \leftrightarrow \neg x_i$$



## Definition (Bit Blasting: Concatenation, Extraction, If)

- concatenation

$$\mathbf{B}_t(\mathbf{x}_k :: \mathbf{y}_m) = (\mathbf{x}_k\mathbf{y}_m, \mathsf{T})$$

for bit vectors $\mathbf{x}_k$ and $\mathbf{y}_m$

- extraction

$$\mathbf{B}_t(\mathbf{x}[n{:}m]) = (\mathbf{z}_{n-m+1}, \varphi) \qquad \varphi = \bigwedge_{i=0}^{n-m} z_i \leftrightarrow x_{i+m}$$

for bit vector $\mathbf{x}_k$, $k > n \geqslant m \geqslant 0$ and fresh variable $\mathbf{z}_{n-m+1}$

- if-then-else

$$\mathbf{B}_t(p \;?\; \mathbf{x}_k : \mathbf{y}_k) = (\mathbf{z}_k, \varphi) \qquad \varphi = \bigwedge_{i=0}^{k-1} (p \to (z_i \leftrightarrow x_i)) \wedge (\neg p \to (z_i \leftrightarrow y_i))$$

for formula $p$ and bit vectors $\mathbf{x}_k$ and $\mathbf{y}_k$

### Definition (Bit Blasting: Addition and Subtraction)

- addition

$$\mathbf{B}_t(\mathbf{x}_k + \mathbf{y}_k) = (\mathbf{s}_k, \varphi)$$

ripple-carry adder: $\mathbf{c}_k$ are carry bits

where

$$\varphi = (c_0 \leftrightarrow x_0 \wedge y_0) \wedge (s_0 \leftrightarrow x_0 \oplus y_0) \wedge \bigwedge_{i=1}^{k-1} (c_i \leftrightarrow \mathrm{min2}(x_i, y_i, c_{i-1})) \wedge (s_i \leftrightarrow x_i \oplus y_i \oplus c_{i-1})$$

for fresh variables $\mathbf{s}_k$ and $\mathbf{c}_k$ and $\mathrm{min2}(a, b, d) = (a \wedge b) \vee (a \wedge d) \vee (b \wedge d)$

- unary minus

$$\mathbf{B}_t(-\mathbf{x}_k) = \mathbf{B}_t(\sim \mathbf{x}_k + \mathbf{1}_k)$$

- subtraction

$$\mathbf{B}_t(\mathbf{x}_k - \mathbf{y}_k) = \mathbf{B}_t(\mathbf{x}_k + (-\mathbf{y}_k))$$



### Definition (Bit Blasting: Multiplication and Division)

for bit vectors $\mathbf{x}_k$ and $\mathbf{y}_k$ set

- multiplication

shift-and-add

$$\mathbf{B}_t(\mathbf{x}_k \times \mathbf{y}_k) = \mathbf{B}_t(\mathrm{mul}(\mathbf{x}_k, \mathbf{y}_k, 0))$$

where mul is defined by recursion on last argument:

$$\mathrm{mul}(\mathbf{x}_k, \mathbf{y}_k, k) = \mathbf{0}_k$$

$$\mathrm{mul}(\mathbf{x}_k, \mathbf{y}_k, i) = \mathrm{mul}(\mathbf{x}_k \ll \mathbf{1}_k, \mathbf{y}_k, i+1) + (y_i \ ? \ \mathbf{x}_k : \mathbf{0}_k) \qquad \text{if } i < k$$

- unsigned division

$$\mathbf{B}_t(\mathbf{x}_k \div_u \mathbf{y}_k) = (\mathbf{q}_k, \varphi)$$

$$\varphi = \mathbf{B}(\mathbf{y}_k \neq \mathbf{0}_k \to (\mathbf{q}_k \times \mathbf{y}_k + \mathbf{r}_k = \mathbf{x}_k \wedge \mathbf{r}_k < \mathbf{y}_k \wedge \mathbf{q}_k < \mathbf{x}_k))$$

for fresh variables $\mathbf{q}_k$ and $\mathbf{r}_k$



### Example (SMT-LIB 2 for BV)

$(\mathbf{a}_4 + \mathbf{b}_4 <_u \mathbf{b}_4) \wedge (\mathbf{a}_4 \neq \mathbf{10}_4) \wedge (\mathbf{a}_4 \ \& \ \mathbf{b}_4 = \mathbf{8}_4)$ is expressed as

```
(declare-const a (_ BitVec 4))
(declare-const b (_ BitVec 4))
(assert (bvult (bvadd a b) b))
(assert (not (= a #xa)))
(assert (= (bvand a b) #b1000))
(check-sat)
```

### Bit vectors in SMT-LIB 2

- `(_ BitVec k)` is sort of bitvectors of length $k$
- `#xa` is constant in hexadecimal
- `#b1000` is constant in binary
- `bvadd`, `bvsub`, `bvmul` are arithmetic operations,
  `bvudiv` and `bvsdiv` are unsigned and signed division
- `bvult` and `bvule` are unsigned, `bvslt` and `bvsle` are signed $<$ and $\leqslant$
- `bvshl`, `bvlshr`, `bvashr` are shifts
- `bvand`, `bvor` are bitwise logical operations



## Bit Vectors in python/z3

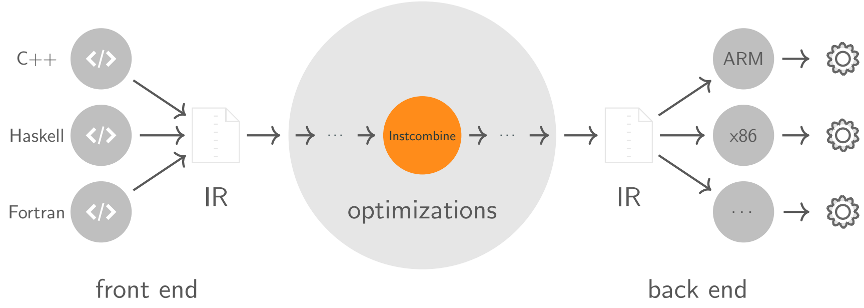
```python
from z3 import *
x = BitVec("x", 8)
y = BitVec("y", 8)
zero = BitVecVal(0, 8)
one = BitVecVal(1, 8)
r = y ^ ((x ^ y) & (zero -(If(x < y, one, zero))))
m = If(x<y, x, y)
solve(r != m) # shorthand for checking single formula
```

- `BitVec(name, k)` creates variable with `k` bits
- `BitVecVal(s, k)` is constant `c` in `k` bits
- `+`, `-`, `*` are arithmetic operations
- `&`, `|`, `~`, `^` are bitwise operations
- comparisons `<`, `<=`, `>`, `>=` are signed, use `ULT`, `ULE`, `UGT`, `UGE` for unsigned
- `<<` is left shift, `>>` is $\gg_s$, `LShR` is $\gg_u$
- division `/` and modulo `%` is signed, use `UDiv` and `URem` for unsigned
- for valuations, solver returns integers by default

## Application 1: Verifying Compiler Optimizations

### LLVM

- open-source umbrella project: set of reusable toolchain components: libraries, assemblers, compilers, debuggers, ...
- compilation toolchain includes peephole optimizations in Instcombine pass






## Application 1: Verifying Compiler Optimizations

### Instcombine Pass

- over 1000 algebraic simplifications of expressions
  - transform multiplies with constant power-of-two argument into shifts
  - bitwise operators with constant operands are always grouped so that shifts are performed first, then ors, then ands, then xors
  - changing bitwidth of variables
  - ...
- code is community maintained
- sometimes optimizations have errors—and compiler bugs are critical

### Example

```
int foo(int z) {
int x = 4 * (z | 1001);
return -256 & x;
}
```

```
define i32 @foo(i32) #0 {
  %2 = or i32 %0, 1001
  %3 = mul nsw i32 4, %2
  %4 = xor i32 -256, %3
  ret i32 %4
}
```

Instcombine →

```
define i32 @foo(i32) #0 {
  %2 = shl i32 %0, 2
  %3 = or i32 %2, 4004
  %4 = xor i32 %3, -256
  ret i32 %4
}
```



## Application 1: Verifying Compiler Optimizations

### Alive Project

- represent Instcombine optimizations in domain-specific language, e.g.

```
Name: PR20186
%a = sdiv %X, C
%r = sub 0, %a
=>
%r = sdiv %X, -C
```

- check correctness by means of SMT encoding

```
(declare-const x (_ BitVec 32))
(declare-const c (_ BitVec 32))
(declare-const before (_ BitVec 32))
(declare-const after (_ BitVec 32))
(assert (= before (bvsub #x00000000 (bvsdiv x c))))
(assert (= after (bvsdiv x (bvneg c))))
(assert (not (= before after)))
(assert (not (= c #x00000000)))
(check-sat)
```

- wrong for c = x = #x80000000



## Same in python/z3

```python
from z3 import *
x = BitVec('x', 32) # create variable named x with 32 bits
c = BitVec('c', 32)

before = BitVecVal(0, 32) - (x / c)
after = x / - c

solver = Solver()
solver.add(c != BitVecVal(0, 32)) # exclude case where c=0
solver.add(after != before)

result = solver.check()
if result == z3.sat:
  m = solver.model()
  print m[x], m[c] # 2147483648 2147483648
  print m.eval(before), m.eval(after) # 4294967295 1
```

## Application 2: Detecting Nontermination in Programs

```
int bsearch(int a[], int k, unsigned int lo, unsigned int hi) {
  unsigned int mid;
  while (lo < hi) {
    mid = (lo + hi)/2;
    if (a[mid] < k)
      lo = mid + 1;
    else if (a[mid] > k)
      hi = mid - 1;
    else
      return mid;
  }
  return -1;
}
```

- (former) implementation of binary search in Java library
- loops for inputs `lo=1` and `hi=UINT_MAX` if `a[0] < k`.
- SMT encoding can find values such that parameters stay the same in recursive call



## Bibliography


- Daniel Kroening and Ofer Strichman
  **Bit Vectors**
  Chapter 6 of Decision Procedures — An Algorithmic Point of View
  Springer, 2008
- Nuno Lopes, David Menendez, Sarantosh Nagarakatte, and John Regehr.
  **Provably Correct Peephole Optimizations with Alive.**
  Proc. 36th PLDI, pp. 22–32, 2013.