Automated Reasoning

Week 2. Beyond the basics of SAT: integers and program verification

Cynthia Kop

Fall 2024

Arithmetic in proposition logic

Arithmetic in proposition logic

Many practical problems use **integers**. But SAT only has booleans...

Arithmetic in proposition logic

Many practical problems use **integers.** But SAT only has booleans…

Solution: use the binary representation!

Arithmetic in proposition logic

Many practical problems use **integers.** But SAT only has booleans. . .

Solution: use the binary representation!

$$a_n a_{n-1} \cdots a_1$$

Addition in primary school

Addition in primary school

Given a and b, find d satisfying $a + b = d$.

Addition in primary school

Given a and b, find d satisfying $a + b = d$.

In decimal:

Addition in primary school

Given a and b, find d satisfying $a + b = d$.

In decimal:

$$\begin{array}{cccccc} 1 & 3 & 7 & \leftarrow & & a \\ & 7 & 9 & \leftarrow & & b \\ \hline & & & & & + \end{array}$$

Addition in primary school

Given a and b, find d satisfying $a + b = d$.

In decimal:

$$\begin{array}{ccccc} 1 & 3 & 7 & \leftarrow & a \\ & 7 & 9 & \leftarrow & b \\ \hline & & & & + \end{array}$$

Addition in primary school

Given a and b, find d satisfying $a + b = d$.

In decimal:

$$\begin{array}{ccccc} 1 & 3 & 7 & \leftarrow & a \\ & 7 & 9 & \leftarrow & b \\ \hline & & \color{red}{6} & & + \end{array}$$

Addition in primary school

Given a and b, find d satisfying $a + b = d$.

In decimal:

$$
\begin{array}{ccccl}
 & \color{red}{1} & & \leftarrow & \text{carry} \\
1 & 3 & 7 & \leftarrow & a \\
 & 7 & 9 & \leftarrow & b \\
\hline
 & & 6 & & +
\end{array}
$$

Addition in primary school

Given a and b, find d satisfying $a + b = d$.

In decimal:

$$
\begin{array}{ccccc}
 & 1 & & \leftarrow & \text{carry} \\
1 & 3 & 7 & \leftarrow & a \\
 & 7 & 9 & \leftarrow & b \\
\hline
 & & 6 & & +
\end{array}
$$

Addition in primary school

Given a and b, find d satisfying $a + b = d$.

In decimal:

$$\begin{array}{cccccc} \color{red}{1} & 1 & & \leftarrow & \text{carry} \\ 1 & 3 & 7 & \leftarrow & a \\ & 7 & 9 & \leftarrow & b \\ \hline & \color{red}{1} & 6 & & + \end{array}$$

Addition in primary school

Given a and b, find d satisfying $a + b = d$.

In decimal:

$$
\begin{array}{rrrcl}
\color{red}{1} & 1 & & \leftarrow & \text{carry} \\
\color{red}{1} & 3 & 7 & \leftarrow & a \\
 & 7 & 9 & \leftarrow & b \\
\hline
 & 1 & 6 & & +
\end{array}
$$

Addition in primary school

Given a and b, find d satisfying $a + b = d$.

In decimal:

$$
\begin{array}{cccclc}
0 & 1 & 1 & & \leftarrow & \text{carry} \\
 & 1 & 3 & 7 & \leftarrow & a \\
 & & 7 & 9 & \leftarrow & b \\
\hline
 & 2 & 1 & 6 & & +
\end{array}
$$

Addition in primary school

Given a and b, find d satisfying $a + b = d$.

In decimal:

$$
\begin{array}{cccccc}
0 & 1 & 1 & & \leftarrow & \text{carry} \\
 & 1 & 3 & 7 & \leftarrow & a \\
 & & 7 & 9 & \leftarrow & b \\
\hline
 & 2 & 1 & 6 & & +
\end{array}
$$

In binary: just the same!

Addition in primary school

Given a and b, find d satisfying $a + b = d$.

In decimal:

$$
\begin{array}{cccccl}
0 & 1 & 1 & & \leftarrow & \text{carry} \\
 & 1 & 3 & 7 & \leftarrow & a \\
 & & 7 & 9 & \leftarrow & b \\
\hline
 & 2 & 1 & 6 & & +
\end{array}
$$

In binary: just the same!

$$
\begin{array}{cccccccl}
0 & 0 & 1 & 1 & 1 & & \leftarrow & \text{carry} \\
 & 0 & 0 & 1 & 1 & 1 & \leftarrow & a = 7 \\
 & 1 & 0 & 1 & 0 & 1 & \leftarrow & b = 21 \\
\hline
0 & 1 & 1 & 1 & 0 & 0 & \leftarrow & d = 7 + 21 = 28
\end{array}
$$

Adding two n-bit numbers

Goal: $a + b = d$, where a, b, d are all **5-bit numbers**.

Adding two n-bit numbers

Goal: $a + b = d$, where a, b, d are all **5-bit numbers.**

$$
\begin{array}{cccccc}
a_5 & a_4 & a_3 & a_2 & a_1 & \\
b_5 & b_4 & b_3 & b_2 & b_1 & \\
\hline
d_5 & d_4 & d_3 & d_2 & d_1 & +
\end{array}
$$

Adding two n-bit numbers

Goal: $a + b = d$, where a, b, d are all **5-bit numbers.**

$$
\begin{array}{ccccccc}
c_5 & c_4 & c_3 & c_2 & c_1 & & \\
 & a_5 & a_4 & a_3 & a_2 & a_1 & \\
 & b_5 & b_4 & b_3 & b_2 & b_1 & \\
\hline
 & d_5 & d_4 & d_3 & d_2 & d_1 & +
\end{array}
$$

Adding two n-bit numbers

Goal: $a + b = d$, where a, b, d are all **5-bit numbers.**

$$
\begin{array}{ccccccc}
c_5 & c_4 & c_3 & c_2 & c_1 & & \\
 & a_5 & a_4 & a_3 & a_2 & a_1 & \\
 & b_5 & b_4 & b_3 & b_2 & b_1 & \\
\hline
 & d_5 & d_4 & d_3 & d_2 & d_1 & +
\end{array}
$$

Requirements:

Adding two n-bit numbers

Goal: $a + b = d$, where a, b, d are all **5-bit numbers.**

$$
\begin{array}{cccccc}
c_5 & c_4 & c_3 & c_2 & c_1 & \\
 & a_5 & a_4 & a_3 & a_2 & a_1 \\
 & b_5 & b_4 & b_3 & b_2 & b_1 \\
\hline
 & d_5 & d_4 & d_3 & d_2 & d_1 \quad +
\end{array}
$$

Requirements:

- $d_1 = a_1$ XOR b_1
- for larger i: $d_i = (c_{i-1} + a_i + b_i)\%2$.

Adding two n-bit numbers

Goal: $a + b = d$, where a, b, d are all **5-bit numbers.**

$$
\begin{array}{cccccc}
c_5 & c_4 & c_3 & c_2 & c_1 & \color{red}{c_0} \\
 & a_5 & a_4 & a_3 & a_2 & a_1 \\
 & b_5 & b_4 & b_3 & b_2 & b_1 \\
\hline
 & d_5 & d_4 & d_3 & d_2 & d_1 & +
\end{array}
$$

Requirements:

- for all i: $d_i = (c_{i-1} + a_i + b_i)\%2$.

Adding two n-bit numbers

Goal: $a + b = d$, where a, b, d are all **5-bit numbers.**

$$
\begin{array}{cccccccc}
c_5 & c_4 & c_3 & c_2 & c_1 & c_0 & \\
 & a_5 & a_4 & a_3 & a_2 & a_1 & \\
 & b_5 & b_4 & b_3 & b_2 & b_1 & \\
\hline
 & d_5 & d_4 & d_3 & d_2 & d_1 & +
\end{array}
$$

Requirements:

- for all i: $d_i = (c_{i-1} + a_i + b_i)\%2$.

$$\bigwedge_{i=1}^{n-1} a_i \leftrightarrow b_i \leftrightarrow c_{i-1} \leftrightarrow d_i$$

Adding two n-bit numbers

Goal: $a + b = d$, where a, b, d are all **5-bit numbers.**

$$
\begin{array}{cccccc}
c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\
 & a_5 & a_4 & a_3 & a_2 & a_1 \\
 & b_5 & b_4 & b_3 & b_2 & b_1 \\
\hline
 & d_5 & d_4 & d_3 & d_2 & d_1 & +
\end{array}
$$

Requirements:

- for all i: $d_i = (c_{i-1} + a_i + b_i)\%2$.
- for all $i < n$: $c_i = 1$ if and only if $a_i + b_i + c_{i-1} > 1$.

Adding two n-bit numbers

Goal: $a + b = d$, where a, b, d are all **5-bit numbers.**

$$
\begin{array}{cccccc}
c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\
 & a_5 & a_4 & a_3 & a_2 & a_1 \\
 & b_5 & b_4 & b_3 & b_2 & b_1 \\
\hline
 & d_5 & d_4 & d_3 & d_2 & d_1 & +
\end{array}
$$

Requirements:

- for all i: $d_i = (c_{i-1} + a_i + b_i)\%2$.
- for all $i < n$: $c_i = 1$ if and only if $a_i + b_i + c_{i-1} > 1$.

$$\bigwedge_{i=1}^{n-1} c_i \leftrightarrow ((a_i \wedge b_i) \vee (a_i \wedge c_{i-1}) \vee (b_i \wedge c_{i-1}))$$

Adding two n-bit numbers

Goal: $a + b = d$, where a, b, d are all **5-bit numbers.**

$$\begin{array}{cccccccc}
c_5 & c_4 & c_3 & c_2 & c_1 & c_0 & \\
 & a_5 & a_4 & a_3 & a_2 & a_1 & \\
 & b_5 & b_4 & b_3 & b_2 & b_1 & \\
\hline
 & d_5 & d_4 & d_3 & d_2 & d_1 & +
\end{array}$$

Requirements:

- for all i: $d_i = (c_{i-1} + a_i + b_i)\%2$.
- for all $i < n$: $c_i = 1$ if and only if $a_i + b_i + c_{i-1} > 1$.
- $c_0 = 0$ (no initial carry) and $c_n = 0$ (no overflow).

Adding two n-bit numbers

Goal: $a + b = d$, where a, b, d are all **5-bit numbers.**

$$
\begin{array}{cccccccc}
c_5 & c_4 & c_3 & c_2 & c_1 & c_0 & \\
 & a_5 & a_4 & a_3 & a_2 & a_1 & \\
 & b_5 & b_4 & b_3 & b_2 & b_1 & \\
\hline
 & d_5 & d_4 & d_3 & d_2 & d_1 & +
\end{array}
$$

Requirements:

- for all i: $d_i = (c_{i-1} + a_i + b_i)\%2$.
- for all $i < n$: $c_i = 1$ if and only if $a_i + b_i + c_{i-1} > 1$.
- $c_0 = 0$ (no initial carry) and $c_n = 0$ (no overflow).

$$\neg c_0 \wedge \neg c_n$$

Making a SAT-solver add

Challenge: make a SAT-solver compute 17 + 11

Making a SAT-solver add

Challenge: make a SAT-solver compute 17 + 11

$$
\begin{array}{rcccccc}
17 & = & 1 & 0 & 0 & 0 & 1 \\
11 & = & 0 & 1 & 0 & 1 & 1
\end{array}
$$

Making a SAT-solver add

Challenge: make a SAT-solver compute 17 + 11

$$\begin{array}{ccccccc} 17 & = & 1 & 0 & 0 & 0 & 1 \\ 11 & = & 0 & 1 & 0 & 1 & 1 \end{array}$$

Solution:

$$\phi \ \wedge \ \underbrace{a_5 \wedge \neg a_4 \wedge \neg a_3 \wedge \neg a_2 \wedge a_1}_{\vec{a}=17} \ \wedge \ \underbrace{\neg b_5 \wedge b_4 \wedge \neg b_3 \wedge b_2 \wedge b_1}_{\vec{b}=11}$$

Making a SAT-solver add

Challenge: make a SAT-solver compute 17 + 11

$$\begin{array}{rcccccc} 17 & = & 1 & 0 & 0 & 0 & 1 \\ 11 & = & 0 & 1 & 0 & 1 & 1 \end{array}$$

Solution:

$$\phi \wedge \underbrace{a_5 \wedge \neg a_4 \wedge \neg a_3 \wedge \neg a_2 \wedge a_1}_{\vec{a}=17} \wedge \underbrace{\neg b_5 \wedge b_4 \wedge \neg b_3 \wedge b_2 \wedge b_1}_{\vec{b}=11}$$

Challenge: make a SATs-solver compute 17 + 18

Making a SAT-solver add

Challenge: make a SAT-solver compute 17 + 11

$$\begin{array}{rcccccc} 17 & = & 1 & 0 & 0 & 0 & 1 \\ 11 & = & 0 & 1 & 0 & 1 & 1 \end{array}$$

Solution:

$$\phi \wedge \underbrace{a_5 \wedge \neg a_4 \wedge \neg a_3 \wedge \neg a_2 \wedge a_1}_{\vec{a}=17} \wedge \underbrace{\neg b_5 \wedge b_4 \wedge \neg b_3 \wedge b_2 \wedge b_1}_{\vec{b}=11}$$

Challenge: make a SATs-solver compute 17 + 18

Solution: just add a leading 0 to a and b and use 6-bit addition!

Making a SAT-solver add and subtract

Challenge: make a SAT-solver compute 17 + 11

$$\begin{array}{rcccccc} 17 & = & 1 & 0 & 0 & 0 & 1 \\ 11 & = & 0 & 1 & 0 & 1 & 1 \end{array}$$

Solution:

$$\phi \;\wedge\; \underbrace{a_5 \wedge \neg a_4 \wedge \neg a_3 \wedge \neg a_2 \wedge a_1}_{\vec{a}=17} \;\wedge\; \underbrace{\neg b_5 \wedge b_4 \wedge \neg b_3 \wedge b_2 \wedge b_1}_{\vec{b}=11}$$

Challenge: compute 17 - 11.

Making a SAT-solver add and subtract

Challenge: make a SAT-solver compute 17 + 11

$$\begin{array}{ccccccc} 17 & = & 1 & 0 & 0 & 0 & 1 \\ 11 & = & 0 & 1 & 0 & 1 & 1 \end{array}$$

Solution:

$$\phi \wedge \underbrace{a_5 \wedge \neg a_4 \wedge \neg a_3 \wedge \neg a_2 \wedge a_1}_{\vec{a}=17} \wedge \underbrace{\neg b_5 \wedge b_4 \wedge \neg b_3 \wedge b_2 \wedge b_1}_{\vec{b}=11}$$

Challenge: compute 17 - 11.

Solution:

$$\phi \wedge \underbrace{d_5 \wedge \neg d_4 \wedge \neg d_3 \wedge \neg d_2 \wedge d_1}_{\vec{d}=17} \wedge \underbrace{\neg a_5 \wedge a_4 \wedge \neg a_3 \wedge a_2 \wedge a_1}_{\vec{a}=11}$$

Multiplication in primary school

Given a and b, find d satisfying $a * b = d$.

Multiplication in primary school

Given a and b, find d satisfying $a * b = d$.

In decimal:

$$
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
\hline
\\
\hline
\end{array}
$$

Multiplication in primary school

Given a and b, find d satisfying $a * b = d$.

In decimal:

$$
\begin{array}{ccccl}
 & 1 & 2 & 3 & \\
 & 4 & 5 & \color{red}{6} & \\
\hline
 & 7 & 3 & 8 & = 6 * 123 \\
 & & & & \\
\hline
\end{array}
$$

Multiplication in primary school

Given a and b, find d satisfying $a * b = d$.

In decimal:

$$
\begin{array}{rrrrl}
 & 1 & 2 & 3 & \\
 & 4 & 5 & 6 & \\
\hline
 & 7 & 3 & 8 & = 6 * 123 \\
6 & 1 & 5 & & = 5 * 123 \\
 & & & & \\
\hline
\end{array}
$$

Multiplication in primary school

Given a and b, find d satisfying $a * b = d$.

In decimal:

$$
\begin{array}{cccccc}
 & & 1 & 2 & 3 & \\
 & & \color{red}{4} & 5 & 6 & \\
\hline
 & & 7 & 3 & 8 & = 6 * 123 \\
 & 6 & 1 & 5 & & = 5 * 123 \\
4 & 9 & 2 & & & = 4 * 123 \\
\hline
\end{array}
$$

Multiplication in primary school

Given a and b, find d satisfying $a * b = d$.

In decimal:

$$
\begin{array}{ccccccl}
 & & 1 & 2 & 3 & & \\
 & & 4 & 5 & 6 & & \\
\hline
 & & 7 & 3 & 8 & & = 6 * 123 \\
 & 6 & 1 & 5 & & & = 5 * 123 \\
4 & 9 & 2 & & & & = 4 * 123 \\
\hline
5 & 6 & 0 & 8 & 8 & + &
\end{array}
$$

Multiplication in primary school

Given a and b, find d satisfying $a * b = d$.

In decimal:

$$
\begin{array}{cccccl}
 & & 1 & 2 & 3 & \\
 & & 4 & 5 & 6 & \\
\hline
 & & 7 & 3 & 8 & = 6 * 123 \\
 & 6 & 1 & 5 & & = 5 * 123 \\
4 & 9 & 2 & & & = 4 * 123 \\
\hline
5 & 6 & 0 & 8 & 8 & +
\end{array}
$$

Proposed algorithm:

Multiplication in primary school

Given a and b, find d satisfying $a * b = d$.

In decimal:

$$
\begin{array}{ccccc|l}
 & & 1 & 2 & 3 & \\
 & & 4 & 5 & 6 & \\
\hline
 & & 7 & 3 & 8 & = 6 * 123 \\
 & 6 & 1 & 5 & & = 5 * 123 \\
4 & 9 & 2 & & & = 4 * 123 \\
\hline
5 & 6 & 0 & 8 & 8 & +
\end{array}
$$

Proposed algorithm:

d := 0

Multiplication in primary school

Given a and b, find d satisfying $a * b = d$.

In decimal:

$$
\begin{array}{ccccccl}
 & & 1 & 2 & 3 & & \\
 & & 4 & 5 & 6 & & \\
\hline
 & & 7 & 3 & 8 & & = 6 * 123 \\
 & 6 & 1 & 5 & & & = 5 * 123 \\
4 & 9 & 2 & & & & = 4 * 123 \\
\hline
5 & 6 & 0 & 8 & 8 & + &
\end{array}
$$

Proposed algorithm:

d := 0

for i := n downto 1 do:

Multiplication in primary school

Given a and b, find d satisfying $a * b = d$.

In decimal:

$$\begin{array}{ccccc|l}
 & & 1 & 2 & 3 & \\
 & & 4 & 5 & 6 & \\
\hline
 & & 7 & 3 & 8 & = 6 * 123 \\
 & 6 & 1 & 5 & & = 5 * 123 \\
4 & 9 & 2 & & & = 4 * 123 \\
\hline
5 & 6 & 0 & 8 & 8 & +
\end{array}$$

Proposed algorithm:

d := 0

for i := n downto 1 do:

$d := 10 * d + b_i * a$

Binary multiplication

Decimal algorithm:

d := 0

for i := n downto 1 do:

$d := 10 * d + b_i * a$

Binary multiplication

Decimal algorithm:

d := 0
for i := n downto 1 do:
$$d := 10 * d + b_i * a$$

Binary algorithm:

d := 0
for i := n downto 1 do:

Binary multiplication

Decimal algorithm:

d := 0
for i := n downto 1 do:
$d := 10 * d + b_i * a$

Binary algorithm:

d := 0
for i := n downto 1 do:
if b_i **then** $d := 2 * d + a$
else $d := 2 * d$

Binary multiplication

Binary algorithm:

d := 0

for i := n downto 1 do:

if b_i **then** $d := 2 * d + a$

else $d := 2 * d$

Example: 9 * 11 (01011)

Binary multiplication

Binary algorithm:

d := 0

for i := n downto 1 do:

if b_i **then** $d := 2 * d + a$

else $d := 2 * d$

Example: 9 * 11 (01011)

- $d = 0$

Binary multiplication

Binary algorithm:

d := 0

for i := n downto 1 do:

if b_i **then** $d := 2 * d + a$

else $d := 2 * d$

Example: 9 * 11 (01011)

- $d = 0$
- $i = 5$ and $d = 0$ $(0 * 2)$

Binary multiplication

Binary algorithm:

d := 0

for i := n downto 1 do:

if b_i **then** $d := 2 * d + a$

else $d := 2 * d$

Example: 9 * 11 (01011)

- $d = 0$
- $i = 5$ and $d = 0$ $(0 * 2)$
- $i = 4$ and $d = 9$ $(0 * 2 + 9)$

Binary multiplication

Binary algorithm:

d := 0

for i := n downto 1 do:

if b_i **then** $d := 2 * d + a$

else $d := 2 * d$

Example: 9 * 11 (01011)

- $d = 0$
- $i = 5$ and $d = 0$ $(0 * 2)$
- $i = 4$ and $d = 9$ $(0 * 2 + 9)$
- $i = 3$ and $d = 18$ $(9 * 2)$

Binary multiplication

Binary algorithm:

d := 0

for i := n downto 1 do:

if b_i **then** $d := 2 * d + a$

else $d := 2 * d$

Example: 9 * 11 (01011)

- $d = 0$
- $i = 5$ and $d = 0$ $(0 * 2)$
- $i = 4$ and $d = 9$ $(0 * 2 + 9)$
- $i = 3$ and $d = 18$ $(9 * 2)$
- $i = 2$ and $d = 45$ $(18 * 2 + 9)$

Binary multiplication

Binary algorithm:

d := 0

for i := n downto 1 do:

if b_i **then** $d := 2 * d + a$

else $d := 2 * d$

Example: 9 * 11 (01011)

- $d = 0$
- $i = 5$ and $d = 0$ $(0 * 2)$
- $i = 4$ and $d = 9$ $(0 * 2 + 9)$
- $i = 3$ and $d = 18$ $(9 * 2)$
- $i = 2$ and $d = 45$ $(18 * 2 + 9)$
- $i = 1$ and $d = 99$ $(45 * 2 + 9)$

Binary multiplication

d := 0
for i := n downto 1 do:
 if b_i then $d := 2 * d + a$
 else $d := 2 * d$

Invariant: $d = [b_n \cdots b_i] * a$, so at the end $d = b * a$

Binary multiplication

d := 0
for i := n downto 1 do:
    if b_i then $d := 2 * d + a$
    else $d := 2 * d$

Invariant: $d = [b_n \cdots b_i] * a$, so at the end $d = b * a$

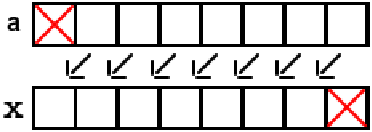
Goal: $\vec{x} = 2\vec{a}$

Binary multiplication

d := 0
for i := n downto 1 do:
    if b_i then $d := 2 * d + a$
    else $d := 2 * d$

Invariant: $d = [b_n \cdots b_i] * a$, so at the end $d = b * a$

Goal: $\vec{x} = 2\vec{a}$

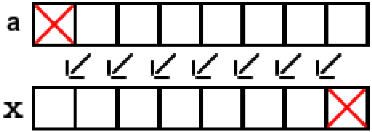


Binary multiplication

d := 0
for i := n downto 1 do:
  if b_i then $d := 2 * d + a$
  else $d := 2 * d$

Invariant: $d = [b_n \cdots b_i] * a$, so at the end $d = b * a$

Goal: $\vec{x} = 2\vec{a}$



Solution:

$$\mathrm{dup}(\vec{a}, \vec{x}) = \neg a_n \wedge \neg x_1 \wedge \bigwedge_{i=1}^{n-1} (x_{i+1} \leftrightarrow a_i)$$

Encoding a variable that changes over time

d := 0
for i := n downto 1 do:
    if b_i then $d := 2 * d + a$
    else $d := 2 * d$

In every step, d changes!

Encoding a variable that changes over time

d := 0
for i := n downto 1 do:
    if b_i then $d := 2 * d + a$
    else $d := 2 * d$

In every step, d changes!

Solution: introduce boolean variables $\vec{r}_i$ for $i \in \{0, \ldots, n\}$.

Encoding a variable that changes over time

d := 0
for i := n downto 1 do:
  if b_i then $d := 2 * d + a$
  else $d := 2 * d$

In every step, d changes!

Solution: introduce boolean variables $\vec{r}_i$ for $i \in \{0, \ldots, n\}$.

That is: for $i \in \{0, \ldots, n\}, j \in \{1, \ldots, n\}$, we introduce a boolean variable $r_{i,j}$.

Encoding a variable that changes over time

d := 0
for i := n downto 1 do:
    if b_i then $d := 2 * d + a$
    else $d := 2 * d$

In every step, d changes!

Solution: introduce boolean variables $\vec{r}_i$ for $i \in \{0, \ldots, n\}$.

That is: for $i \in \{0, \ldots, n\}, j \in \{1, \ldots, n\}$, we introduce a boolean variable $r_{i,j}$.

Also introduce $\vec{s}_i$ for $i \in \{1, \ldots, n\}$.

Encoding a variable that changes over time

d := 0
for i := n downto 1 do:
    if b_i then $d := 2 * d + a$
    else $d := 2 * d$

In every step, d changes!

Solution: introduce boolean variables $\vec{r}_i$ for $i \in \{0, \ldots, n\}$.

That is: for $i \in \{0, \ldots, n\}, j \in \{1, \ldots, n\}$, we introduce a boolean variable $r_{i,j}$.

Also introduce $\vec{s}_i$ for $i \in \{1, \ldots, n\}$.

We will use $\vec{s}$ to represent $2 * \vec{r}$.

Encoding a variable that changes over time

Solution: introduce boolean variables $\vec{r}_i$ for $i \in \{0, \ldots, n\}$.

That is: for $i \in \{0, \ldots, n\}, j \in \{1, \ldots, n\}$, we introduce a boolean variable $r_{i,j}$.

Also introduce $\vec{s}_i$ for $i \in \{1, \ldots, n\}$.

We will use $\vec{s}$ to represent $2 * \vec{r}$.

Updated algorithm:

$\vec{r}_n = 0$

for i := n downto 1 do:

$\vec{s}_i = 2 * \vec{r}_i$

if b_i then $\vec{r}_{i-1} = \vec{s}_i + \vec{a}$

else $\vec{r}_{i-1} = \vec{s}_i$

Bringing it all together: multiplication

The requirement

$$\vec{a} * \vec{b} = \vec{r}_0$$

is now described by the formula:

Bringing it all together: multiplication

The requirement

$$\vec{a} * \vec{b} = \vec{r}_0$$

is now described by the formula:

mul$(\vec{a}, \vec{b}, \vec{r}_0) \quad =$

$$\bigwedge_{j=1}^{n} \neg r_{nj}$$

$\vec{r}_n := 0$

$$\wedge$$

$$\bigwedge_{i=1}^{n} \left(\begin{array}{c} \text{dup}(\vec{r}_i, \vec{s}_i) \\ \wedge \\ b_i \rightarrow \text{plus}(\vec{a}, \vec{s}_i, \vec{r}_{i-1})) \\ \wedge \\ \neg b_i \rightarrow \bigwedge_{j=1}^{n} (s_{ij} \leftrightarrow r_{(i-1)j}) \end{array} \right)$$

for $i := n$ downto 1:
- $\vec{s}_i = 2 * \vec{r}_i$;
- if b_i then
- $\vec{r}_{i-1} = \vec{s}_i + \vec{a}$;
- else $\vec{r}_{i-1} = \vec{s}_i$;

Factorisation

Challenge: Is 1234567891 prime? And 1234567897?

Factorisation

Challenge: Is 1234567891 prime? And 1234567897?

Define

$$\mathsf{fac}(r) \;=\; \mathsf{mul}(a,b,r) \wedge a>1 \wedge b>1$$

Factorisation

Challenge: Is 1234567891 prime? And 1234567897?

Define

$$\text{fac}(r) \;=\; \text{mul}(a, b, r) \;\wedge\; a > 1 \;\wedge\; b > 1$$

Answers:

- fac(1234567891) is unsatisfiable, so 1234567891 is prime. Found by `minisat` or `yices` within 1 minute.

Factorisation

Challenge: Is 1234567891 prime? And 1234567897?

Define
fac(r) $=$ mul(a, b, r) $\wedge\ a > 1\ \wedge\ b > 1$

Answers:

- fac(1234567891) is unsatisfiable, so 1234567891 is prime. Found by `minisat` or `yices` within 1 minute.
- fac(1234567897) is satisfiable, yielding

$$1234567897 = 1241 \times 994817$$

found by `minisat` or `yices` within 1 second.

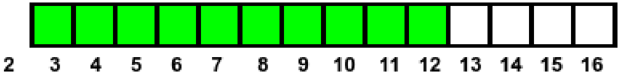
Unary arithmetic in proposition logic

Unary arithmetic in proposition logic

Idea: implement a number in the range $i..j$ by $j - i$ booleans.

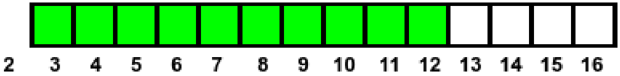
Unary arithmetic in proposition logic

Idea: implement a number in the range $i..j$ by $j - i$ booleans.



Unary arithmetic in proposition logic

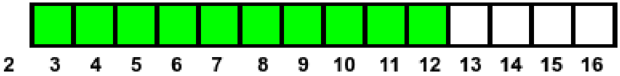
Idea: implement a number in the range $i..j$ by $j - i$ booleans.



Boolean variables $x_{i+1}, \ldots, x_j$.

Unary arithmetic in proposition logic

Idea: implement a number in the range $i..j$ by $j - i$ booleans.

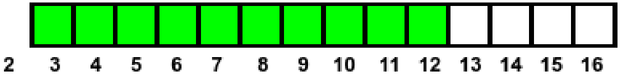


Boolean variables $x_{i+1}, \ldots, x_j$.

Boolean variable x_k represents: $\vec{x} \geq k$.

Unary arithmetic in proposition logic

Idea: implement a number in the range $i..j$ by $j - i$ booleans.



Boolean variables $x_{i+1}, \ldots, x_j$.

Boolean variable x_k represents: $\vec{x} \geq k$.

Well-definedness condition:

$$\bigwedge_{k=i+2}^{j} x_k \rightarrow x_{k-1}$$

Unary addition

Given: $\vec{a} \in \{a_{\text{min}}..a_{\text{max}}\}$ and $\vec{b} \in \{b_{\text{min}}..b_{\text{max}}\}$.
How to express $\vec{c} = \vec{a} + \vec{b}$?

Unary addition

Given: $\vec{a} \in \{a_{\text{min}}..a_{\text{max}}\}$ and $\vec{b} \in \{b_{\text{min}}..b_{\text{max}}\}$.
How to express $\vec{c} = \vec{a} + \vec{b}$?

Note: $c_{\text{min}} = a_{\text{min}} + b_{\text{min}}$ and $c_{\text{max}} = a_{\text{max}} + b_{\text{max}}$.

Unary addition

Given: $\vec{a} \in \{a_{\mathsf{min}}..a_{\mathsf{max}}\}$ and $\vec{b} \in \{b_{\mathsf{min}}..b_{\mathsf{max}}\}$.
How to express $\vec{c} = \vec{a} + \vec{b}$?

Note: $c_{\mathsf{min}} = a_{\mathsf{min}} + b_{\mathsf{min}}$ and $c_{\mathsf{max}} = a_{\mathsf{max}} + b_{\mathsf{max}}$.

$$\bigwedge_{i=a_{\mathsf{min}}}^{a_{\mathsf{max}}} \bigwedge_{j=b_{\mathsf{min}}}^{b_{\mathsf{max}}} \vec{a} \geq i \wedge \vec{b} \geq j \rightarrow \vec{c} \geq i + j$$

$$\bigwedge_{i=a_{\mathsf{min}}}^{a_{\mathsf{max}}} \bigwedge_{j=b_{\mathsf{min}}}^{b_{\mathsf{max}}} \vec{a} \leq i \wedge \vec{b} \leq j \rightarrow \vec{c} \leq i + j$$

Unary addition

Given: $\vec{a} \in \{a_{\text{min}}..a_{\text{max}}\}$ and $\vec{b} \in \{b_{\text{min}}..b_{\text{max}}\}$.
How to express $\vec{c} = \vec{a} + \vec{b}$?

Note: $c_{\text{min}} = a_{\text{min}} + b_{\text{min}}$ and $c_{\text{max}} = a_{\text{max}} + b_{\text{max}}$.

$$\bigwedge_{i=a_{\text{min}}}^{a_{\text{max}}} \bigwedge_{j=b_{\text{min}}}^{b_{\text{max}}} a_i \wedge b_i \rightarrow c_{i+j}$$

$$\bigwedge_{i=a_{\text{min}}}^{a_{\text{max}}} \bigwedge_{j=b_{\text{min}}}^{b_{\text{max}}} \neg a_{i+1} \wedge \neg b_{i+1} \rightarrow \neg c_{i+j+1}$$

(defining $a_{a_{\text{min}}} = \top$ and $a_{a_{\text{max}}+1} = \bot$; similar for b and c)

Unary addition

Given: $\vec{a} \in \{a_{\mathsf{min}}..a_{\mathsf{max}}\}$ and $\vec{b} \in \{b_{\mathsf{min}}..b_{\mathsf{max}}\}$.
How to express $\vec{c} = \vec{a} + \vec{b}$?

Note: $c_{\mathsf{min}} = a_{\mathsf{min}} + b_{\mathsf{min}}$ and $c_{\mathsf{max}} = a_{\mathsf{max}} + b_{\mathsf{max}}$.

$$\bigwedge_{i=a_{\mathsf{min}}}^{a_{\mathsf{max}}} \bigwedge_{j=b_{\mathsf{min}}}^{b_{\mathsf{max}}} \neg a_i \vee \neg b_i \vee c_{i+j}$$

$$\bigwedge_{i=a_{\mathsf{min}}}^{a_{\mathsf{max}}} \bigwedge_{j=b_{\mathsf{min}}}^{b_{\mathsf{max}}} a_{i+1} \vee b_{i+1} \vee \neg c_{i+j+1}$$

(defining $a_{a_{\mathsf{min}}} = \top$ and $a_{a_{\mathsf{max}}+1} = \bot$; similar for b and c)

Binary versus unary arithmetic

Binary versus unary arithmetic

Let $i \in \{0..n\}, j \in \{0..m\}$.

Binary versus unary arithmetic

Let $i \in \{0..n\}, j \in \{0..m\}$.

	binary	**unary**
number variables to represent i	$\log(n)$	n

Binary versus unary arithmetic

Let $i \in \{0..n\}, j \in \{0..m\}$.

	binary	**unary**
number variables to represent i	$\log(n)$	n
size of CNF formula for $i + j$	$\mathcal{O}(\log(\max(n, m)))$	$\mathcal{O}(n * m)$

Binary versus unary arithmetic

Let $i \in \{0..n\}, j \in \{0..m\}$.

	binary	**unary**
number variables to represent i	$\log(n)$	n
size of CNF formula for $i + j$	$\mathcal{O}(\log(\max(n, m)))$	$\mathcal{O}(n * m)$
size of CNF formula for $i + 1$	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$

Binary versus unary arithmetic

Let $i \in \{0..n\}, j \in \{0..m\}$.

	binary	**unary**
number variables to represent i	$\log(n)$	n
size of CNF formula for $i+j$	$\mathcal{O}(\log(\max(n,m)))$	$\mathcal{O}(n*m)$
size of CNF formula for $i+1$	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$
constant in $\mathcal{O}$ is roughly	20–30	2

Binary versus unary arithmetic

Let $i \in \{0..n\}, j \in \{0..m\}$.

	binary	**unary**
number variables to represent i	$\log(n)$	n
size of CNF formula for $i + j$	$\mathcal{O}(\log(\max(n, m)))$	$\mathcal{O}(n * m)$
size of CNF formula for $i + 1$	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$
constant in $\mathcal{O}$ is roughly	20–30	2
number extra variables for $i + j$	$\mathcal{O}(\log(\max(n, m)))$	0

Binary versus unary arithmetic

Let $i \in \{0..n\}, j \in \{0..m\}$.

	binary	**unary**
number variables to represent i	$\log(n)$	n
size of CNF formula for $i+j$	$\mathcal{O}(\log(\max(n,m)))$	$\mathcal{O}(n*m)$
size of CNF formula for $i+1$	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$
constant in $\mathcal{O}$ is roughly	20–30	2
number extra variables for $i+j$	$\mathcal{O}(\log(\max(n,m)))$	0
easy for SAT solvers	NO	YES

Binary versus unary arithmetic

Let $i \in \{0..n\}, j \in \{0..m\}$.

	binary	**unary**
number variables to represent i	$\log(n)$	n
size of CNF formula for $i+j$	$\mathcal{O}(\log(\max(n,m)))$	$\mathcal{O}(n*m)$
size of CNF formula for $i+1$	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$
constant in $\mathcal{O}$ is roughly	20–30	2
number extra variables for $i+j$	$\mathcal{O}(\log(\max(n,m)))$	0
easy for SAT solvers	NO	YES

Experience: unary is useful for **small** numbers! ($\leq 50 - 100$)

Binary versus unary arithmetic

Let $i \in \{0..n\}, j \in \{0..m\}$.

	binary	**unary**
number variables to represent i	$\log(n)$	n
size of CNF formula for $i + j$	$\mathcal{O}(\log(\max(n, m)))$	$\mathcal{O}(n * m)$
size of CNF formula for $i + 1$	$\mathcal{O}(\log(n))$	$\mathcal{O}(n)$
constant in $\mathcal{O}$ is roughly	20–30	2
number extra variables for $i + j$	$\mathcal{O}(\log(\max(n, m)))$	0
easy for SAT solvers	NO	YES

Experience: unary is useful for **small** numbers! ($\leq 50 - 100$)

Numbers like this occur in many practical problems.

Expressing programs in SAT

Goal: express *simple* integer programs in Boolean logic.

Expressing programs in SAT

Goal: express *simple* integer programs in Boolean logic.

Limitation: programs with a *fixed* (or *bounded*) number of assignments.

Expressing programs in SAT

Goal: express *simple* integer programs in Boolean logic.

Limitation: programs with a *fixed* (or *bounded*) number of assignments.

Basic idea:

- Integer variables: use binary notation

Expressing programs in SAT

Goal: express *simple* integer programs in Boolean logic.

Limitation: programs with a *fixed* (or *bounded*) number of assignments.

Basic idea:

- Integer variables: use binary notation
- For every step i in the program, introduce a copy x_i of every boolean variable x.

Expressing programs in SAT

Goal: express *simple* integer programs in Boolean logic.

Limitation: programs with a *fixed* (or *bounded*) number of assignments.

Basic idea:

- Integer variables: use binary notation
- For every step i in the program, introduce a copy x_i of every boolean variable x.
- $a := b$ in step i can be expressed as:

$$(a_{i+1} \leftrightarrow b_i) \land \bigwedge_c (c_{i+1} \leftrightarrow c_i)$$

where c ranges over all variables $\neq a$.

Expressing programs in SAT

Goal: express *simple* integer programs in Boolean logic.

Limitation: programs with a *fixed* (or *bounded*) number of assignments.

Basic idea:

- Integer variables: use binary notation
- For every step i in the program, introduce a copy x_i of every boolean variable x.
- $a := b$ in step i can be expressed as:

  $$(a_{i+1} \leftrightarrow b_i) \wedge \bigwedge_c (c_{i+1} \leftrightarrow c_i)$$

  where c ranges over all variables $\neq a$.
- For-loops `for i := 1 to m do X` are treated as m copies of `X`.

Program correctness by SAT

Goal: proving a property about a program.

Program correctness by SAT

Goal: proving a property about a program.

Typically given by a **Hoare triple**:

$$\{P\}S\{Q\}$$

Program correctness by SAT

Goal: proving a property about a program.

Typically given by a **Hoare triple**:

$$\{P\}S\{Q\}$$

Here

- S is the program;
- P is the **precondition**;
- Q is the **postcondition**.

Program correctness by SAT

Goal: proving a property about a program.

Typically given by a **Hoare triple**:

$$\{P\}S\{Q\}$$

Here

- S is the program;
- P is the **precondition**;
- Q is the **postcondition.**

For proving $\{P\}S\{Q\}$, add the formula

$$P_0 \quad \wedge \quad \neg Q_m$$

and prove that the resulting formula is unsatisfiable.

Program correctness by SAT – example

Given: a boolean array $a[1..m]$.

Program correctness by SAT – example

Given: a boolean array $a[1..m]$.

CLAIM: After doing

$\texttt{for } j := 1 \texttt{ to } m-1 \texttt{ do } a[j+1] := a[j]$

Program correctness by SAT – example

Given: a boolean array $a[1..m]$.

CLAIM: After doing

$\texttt{for}\ j := 1\ \texttt{to}\ m-1\ \texttt{do}\ a[j+1] := a[j]$

we have

$$\underbrace{a[1] = a[m]}_{\text{postcondition}}$$

Program correctness by SAT – example

Given: a boolean array $a[1..m]$.

CLAIM: After doing

$\texttt{for}\ j := 1\ \texttt{to}\ m-1\ \texttt{do}\ a[j+1] := a[j]$

we have

$$\underbrace{a[1] = a[m]}_{\text{postcondition}}$$

PROOF: let a_{ij} represent the value $a[i]$ after j iterations.

Program correctness by SAT – example

Given: a boolean array $a[1..m]$.

CLAIM: After doing

$\texttt{for } j := 1 \texttt{ to } m-1 \texttt{ do } a[j+1] := a[j]$

we have

$$\underbrace{a[1] = a[m]}_{\text{postcondition}}$$

PROOF: let a_{ij} represent the value $a[i]$ after j iterations.

Semantics of the jth iteration:

$$(a_{j+1,j} \leftrightarrow a_{j,j-1}) \wedge \bigwedge_{i \in \{1,\ldots,m\}, i \neq j+1} (a_{ij} \leftrightarrow a_{i,j-1})$$

Program correctness by SAT – example

Given: a boolean array $a[1..m]$.

CLAIM: After doing

$\texttt{for}\ j := 1\ \texttt{to}\ m-1\ \texttt{do}\ a[j+1] := a[j]$

we have

$$\underbrace{a[1] = a[m]}_{\text{postcondition}}$$

PROOF: let a_{ij} represent the value $a[i]$ after j iterations.

Semantics of the jth iteration:

$$(a_{j+1,j} \leftrightarrow a_{j,j-1}) \wedge \bigwedge_{i \in \{1,\dots,m\}, i \neq j+1} (a_{ij} \leftrightarrow a_{i,j-1})$$

Negation of the postcondition: $\neg(a_{1,m-1} \leftrightarrow a_{m,m-1})$

Expressing more elaborate programs in SAT

Slightly less basic idea:

Expressing more elaborate programs in SAT

Slightly less basic idea:

- *computations:* for a variable update `x := e` in step i:
  - if `e` is another variable: $\bigwedge_{j=1}^{n} x_{ij} \leftrightarrow e_{(i-1)j}$
  - if `e = y + z`: plus(y_{i-1}, z_{i-1}, x_i)
  - if `e = y * z`: mul(y_{i-1}, z_{i-1}, x_i)

Expressing more elaborate programs in SAT

Slightly less basic idea:

- *computations:* for a variable update `x := e` in step i:
  - if `e` is another variable: $\bigwedge_{j=1}^{n} x_{ij} \leftrightarrow e_{(i-1)j}$
  - if `e = y + z`: plus(y_{i-1}, z_{i-1}, x_i)
  - if `e = y * z`: mul(y_{i-1}, z_{i-1}, x_i)
  - if `e = a + b * c`: mul(b_{i-1}, c_{i-1}, tmp) $\wedge$ plus(a_{i-1}, tmp, x_i)
  - ...

Expressing more elaborate programs in SAT

Slightly less basic idea:

- *computations:* for a variable update `x := e` in step i:
  - if `e` is another variable: $\bigwedge_{j=1}^{n} x_{ij} \leftrightarrow e_{(i-1)j}$
  - if `e = y + z`: plus(y_{i-1}, z_{i-1}, x_i)
  - if `e = y * z`: mul(y_{i-1}, z_{i-1}, x_i)
  - if `e = a + b * c`: mul(b_{i-1}, c_{i-1}, tmp) $\wedge$ plus(a_{i-1}, tmp, x_i)
  - . . .
- *conditions:* introduce other formulas like “equal”, “smaller”

Expressing more elaborate programs in SAT

Slightly less basic idea:

- *computations:* for a variable update `x := e` in step i:
  - if `e` is another variable: $\bigwedge_{j=1}^{n} x_{ij} \leftrightarrow e_{(i-1)j}$
  - if `e = y + z`: plus(y_{i-1}, z_{i-1}, x_i)
  - if `e = y * z`: mul(y_{i-1}, z_{i-1}, x_i)
  - if `e = a + b * c`: mul(b_{i-1}, c_{i-1}, tmp) $\wedge$ plus(a_{i-1}, tmp, x_i)
  - ...
- *conditions:* introduce other formulas like “equal”, “smaller”
- *branching:* for `if cond then P else Q`:

Expressing more elaborate programs in SAT

Slightly less basic idea:

- *computations:* for a variable update `x := e` in step i:
  - if `e` is another variable: $\bigwedge_{j=1}^{n} x_{ij} \leftrightarrow e_{(i-1)j}$
  - if `e = y + z`: plus(y_{i-1}, z_{i-1}, x_i)
  - if `e = y * z`: mul(y_{i-1}, z_{i-1}, x_i)
  - if `e = a + b * c`: mul(b_{i-1}, c_{i-1}, tmp) $\wedge$ plus(a_{i-1}, tmp, x_i)
  - . . .
- *conditions:* introduce other formulas like “equal”, “smaller”
- *branching:* for `if cond then P else Q`:
  - add `skip` statements to `P` or `Q` to make them equally long

Expressing more elaborate programs in SAT

Slightly less basic idea:

- *computations:* for a variable update `x := e` in step i:
  - if `e` is another variable: $\bigwedge_{j=1}^{n} x_{ij} \leftrightarrow e_{(i-1)j}$
  - if `e = y + z`: plus(y_{i-1}, z_{i-1}, x_i)
  - if `e = y * z`: mul(y_{i-1}, z_{i-1}, x_i)
  - if `e = a + b * c`: mul$(b_{i-1}, c_{i-1}$, tmp$)$ $\wedge$ plus$(a_{i-1}$, tmp, $x_i)$
  - ...
- *conditions:* introduce other formulas like “equal”, “smaller”
- *branching:* for `if cond then P else Q`:
  - add `skip` statements to `P` or `Q` to make them equally long
  - let `P` encode to φ and `Q` to ψ;
    add requirements $(cond \rightarrow \varphi) \wedge (\neg cond \rightarrow \psi)$

Program correctness by SAT – example

CLAIM: After doing

Program correctness by SAT – example

CLAIM: After doing

$a := 0;$
for $i := 1$ to m do $a := a + k$

Program correctness by SAT – example

CLAIM: After doing

$a := 0;$
for $i := 1$ to m do $a := a + k$

we have $a = m * k$.

Program correctness by SAT – example

CLAIM: After doing

$a := 0;$
for $i := 1$ to m do $a := a + k$

we have $a = m * k$.

PROOF: we need **unsatisfiability** of:

Program correctness by SAT – example

CLAIM: After doing

$a := 0;$
for $i := 1$ to m do $a := a + k$

we have $a = m * k$.

PROOF: we need **unsatisfiability** of:

$$\bigwedge_{j=1}^{n} \neg a_{0,j} \quad \wedge \quad \bigwedge_{i=0}^{m-1} \text{plus}(\vec{a}_i, \vec{k}, \vec{a}_{i+1}) \quad \wedge$$

$$\neg\text{mul}([\vec{m}], \vec{k}, \vec{a}_m)$$

where $[\vec{m}]$ is the binary encoding of number m.

A more complicated program correctness example

```
ret := 0
for i := 20 to 30 do
  if i < x then
    ret := ret + i
  x := x + 1
```

A more complicated program correctness example

Claim: if $\underbrace{x < 20}_{\text{pre-condition}}$ at the start of the program,

then $\underbrace{\text{ret} = 0}_{\text{post-condition}}$ at the end.

```
ret := 0
for i := 20 to 30 do
  if i < x then
    ret := ret + i
  x := x + 1
```

A more complicated program correctness example

Claim: if $\underbrace{x < 20}_{\text{pre-condition}}$ at the start of the program,

then $\underbrace{\text{ret} = 0}_{\text{post-condition}}$ at the end.

```
ret := 0
for i := 20 to 30 do
  if i < x then
    ret := ret + i
  else
    skip
  x := x + 1
```

A more complicated program correctness example

Claim: if $\underbrace{x < 20}_{\text{pre-condition}}$ at the start of the program,

then $\underbrace{\text{ret} = 0}_{\text{post-condition}}$ at the end.

```
(1)             ret := 0
                for i := 20 to 30 do
                  if i < x then
(2(i-19))           ret := ret + i
                  else
(2(i-19))           skip
(2(i-19)+1)       x := x + 1
```

A more complicated program correctness example

Claim: if $\underbrace{x < 20}_{\text{pre-condition}}$ at the start of the program,

then $\underbrace{\text{ret} = 0}_{\text{post-condition}}$ at the end.

```
(1)             ret := 0
                for i := 20 to 30 do
                  if i < x then
(2(i-19))           ret := ret + i
                  else
(2(i-19))           skip
(2(i-19)+1)       x := x + 1
```

Total steps: 23

A more complicated program correctness example

$\text{smaller}(\vec{x}_0, [\vec{20}])$	$\wedge$	pre-condition
$\neg\text{equal}(\vec{r}_{23}, [\vec{0}])$	$\wedge$	post-condition
$\text{equal}(\vec{r}_1, [\vec{0}])$	$\wedge$	(1) `ret := 0`
$\text{equal}(\vec{x}_1, \vec{x}_0)$	$\wedge$	
$\bigwedge_{i=20}^{30} \text{smaller}([\vec{i}], \vec{x}_{2(i-20)+1}) \rightarrow$		`(2(i-19)) ret :=`
$\text{plus}(r_{2(i-20)+1}, [\vec{i}], r_{2(i-19)})$	$\wedge$	`ret + i`
$\bigwedge_{i=20}^{30} \text{smaller}([\vec{i}], \vec{x}_{2(i-20)+1}) \rightarrow$		
$\text{equal}(x_{2(i-20)+1}, x_{2(i-19)})$	$\wedge$	
$\bigwedge_{i=20}^{30} \neg\text{smaller}([\vec{i}], \vec{x}_{2(i-20)+1}) \rightarrow$		`(2(i-19)) skip`
$\text{equal}(r_{2(i-20)+1}, r_{2(i-19)})$	$\wedge$	
$\bigwedge_{i=20}^{30} \neg\text{smaller}([\vec{i}], \vec{x}_{2(i-20)+1}) \rightarrow$		
$\text{equal}(x_{2(i-20)+1}, x_{2(i-19)})$	$\wedge$	
$\bigwedge_{i=20}^{30} \text{plus}(\vec{x}_{2(i-19)}, [\vec{1}], \vec{x}_{2(i-19)+1})$		`(2(i-19)+1) x := x + 1`

Program correctness summary

Overall: a rich class of imperative programs is covered.

Program correctness summary

Overall: a rich class of imperative programs is covered.

SAT versus SMT: bounded or unbounded integers.

The need for CNF

So far:

The need for CNF

So far:

- SAT solvers can handle **CNF** as input format.

The need for CNF

So far:

- SAT solvers can handle **CNF** as input format.
- However: a lot of $\leftrightarrow$ in this lecture...

The need for CNF

So far:

- SAT solvers can handle **CNF** as input format.
- However: a lot of $\leftrightarrow$ in this lecture...

Need: transform arbitrary boolean formulas to CNF.

The need for CNF

So far:

- SAT solvers can handle **CNF** as input format.
- However: a lot of $\leftrightarrow$ in this lecture...

Need: transform arbitrary boolean formulas to CNF.

Straightforward approach: transform the proposition to a logically equivalent CNF.

The need for CNF

So far:

- SAT solvers can handle **CNF** as input format.
- However: a lot of $\leftrightarrow$ in this lecture...

Need: transform arbitrary boolean formulas to CNF.

Straightforward approach: transform the proposition to a logically equivalent CNF.

every 0 in the truth table yields a clause
proposition $\equiv$ *conjunction of these clauses*

The need for CNF

So far:

- SAT solvers can handle **CNF** as input format.
- However: a lot of $\leftrightarrow$ in this lecture...

Need: transform arbitrary boolean formulas to CNF.

Straightforward approach: transform the proposition to a logically equivalent CNF.

every 0 in the truth table yields a clause
proposition $\equiv$ *conjunction of these clauses*

However: this is worst-case exponential.

Formulas whose CNF is always large

Consider:

Formulas whose CNF is always large

Consider:

$$A : (\cdots((p_1 \leftrightarrow p_2) \leftrightarrow p_3) \cdots \leftrightarrow p_n)$$

True $\leftrightarrow$ an even number of p_i's has the value *false*.

Formulas whose CNF is always large

Consider:

$$A : (\cdots((p_1 \leftrightarrow p_2) \leftrightarrow p_3) \cdots \leftrightarrow p_n)$$

True $\leftrightarrow$ an even number of p_i's has the value *false*.

Claim: for every CNF B satisfying $A \equiv B$:
every clause C in B contains exactly n literals.

Formulas whose CNF is always large

Consider:

$$A : (\cdots((p_1 \leftrightarrow p_2) \leftrightarrow p_3) \cdots \leftrightarrow p_n)$$

True $\leftrightarrow$ an even number of p_i's has the value *false*.

Claim: for every CNF B satisfying $A \equiv B$:
every clause C in B contains exactly n literals.

Proof: if not, there is is some clause C in B missing p_i.

Formulas whose CNF is always large

Consider:

$$A : (\cdots((p_1 \leftrightarrow p_2) \leftrightarrow p_3) \cdots \leftrightarrow p_n)$$

True $\leftrightarrow$ an even number of p_i's has the value *false*.

Claim: for every CNF B satisfying $A \equiv B$:
every clause C in B contains exactly n literals.

Proof: if not, there is is some clause C in B missing p_i.

Give values to the remaining variables to make C false. Then, B is false.

Formulas whose CNF is always large

Consider:

$$A : (\cdots((p_1 \leftrightarrow p_2) \leftrightarrow p_3) \cdots \leftrightarrow p_n)$$

True $\leftrightarrow$ an even number of p_i's has the value *false*.

Claim: for every CNF B satisfying $A \equiv B$:
every clause C in B contains exactly n literals.

Proof: if not, there is is some clause C in B missing p_i.

Give values to the remaining variables to make C false. Then, B is false.

Now give a value to p_i such that A yields *true*.

Formulas whose CNF is always large

Consider:

$$A : (\cdots((p_1 \leftrightarrow p_2) \leftrightarrow p_3) \cdots \leftrightarrow p_n)$$

True $\leftrightarrow$ an even number of p_i's has the value *false*.

Claim: for every CNF B satisfying $A \equiv B$:
every clause C in B contains exactly n literals.

Proof: if not, there is is some clause C in B missing p_i.

Give values to the remaining variables to make C false. Then, B is false.

Now give a value to p_i such that A yields *true*.

Contradiction with $A \equiv B$. $\square$

Formulas whose CNF is always large

Consider:

$$A : (\cdots((p_1 \leftrightarrow p_2) \leftrightarrow p_3) \cdots \leftrightarrow p_n)$$

True $\leftrightarrow$ an even number of p_i's has the value *false*.

Claim: for every CNF B satisfying $A \equiv B$:
every clause C in B contains exactly n literals.

Proof: if not, there is is some clause C in B missing p_i.

Give values to the remaining variables to make C false. Then, B is false.

Now give a value to p_i such that A yields *true*.

Contradiction with $A \equiv B$. $\square$

#Clauses in B: number of 0 entries in the truth table.

Formulas whose CNF is always large

Consider:

$$A : (\cdots((p_1 \leftrightarrow p_2) \leftrightarrow p_3) \cdots \leftrightarrow p_n)$$

True $\leftrightarrow$ an even number of p_i's has the value *false*.

Claim: for every CNF B satisfying $A \equiv B$:
every clause C in B contains exactly n literals.

Proof: if not, there is is some clause C in B missing p_i.

Give values to the remaining variables to make C false. Then, B is false.

Now give a value to p_i such that A yields *true*.

Contradiction with $A \equiv B$. $\square$

#Clauses in B: number of 0 entries in the truth table. (2^{n-1})

Formulas whose CNF is always large

Consider:

$$A : (\cdots((p_1 \leftrightarrow p_2) \leftrightarrow p_3) \cdots \leftrightarrow p_n)$$

True $\leftrightarrow$ an even number of p_i's has the value *false*.

Claim: for every CNF B satisfying $A \equiv B$:
every clause C in B contains exactly n literals.

Proof: if not, there is is some clause C in B missing p_i.

Give values to the remaining variables to make C false. Then, B is false.

Now give a value to p_i such that A yields *true*.

Contradiction with $A \equiv B$. $\square$

#Clauses in B: number of 0 entries in the truth table. (2^{n-1})

$\Longrightarrow$ Any CNF B equivalent to A has size exponential in $|A|$.

Alternatives to an equivalent CNF

Goal: given proposition A, find CNF B such that:

Alternatives to an equivalent CNF

Goal: given proposition A, find CNF B such that:

- A and B are **equi-satisfiable**;

Alternatives to an equivalent CNF

Goal: given proposition A, find CNF B such that:

- A and B are **equi-satisfiable**;
- the size of B is **linear** in the size of A.

Alternatives to an equivalent CNF

Goal: given proposition A, find CNF B such that:

- A and B are **equi-satisfiable**;
- the size of B is **linear** in the size of A.

Difference: we allow B to contain **extra variables**.

Alternatives to an equivalent CNF

Goal: given proposition A, find CNF B such that:

- A and B are **equi-satisfiable**;
- the size of B is **linear** in the size of A.

Difference: we allow B to contain **extra variables**.

This will result in the **Tseitin transformation**.

Tseitin Transformation: basics

Define: for specific (small) D, let *cnf*$(D) \equiv D$:

Tseitin Transformation: basics

Define: for specific (small) D, let *cnf*$(D) \equiv D$:

$$\mathit{cnf}(p \leftrightarrow \neg q) \quad = \quad (p \vee q) \\ \wedge(\neg p \vee \neg q)$$

Tseitin Transformation: basics

Define: for specific (small) D, let *cnf*$(D) \equiv D$:

$$\begin{aligned}
\textit{cnf}(p \leftrightarrow \neg q) &= (p \vee q) \\
&\quad \wedge(\neg p \vee \neg q)
\end{aligned}$$

$$\begin{aligned}
\textit{cnf}(p \leftrightarrow (q \wedge r)) &= (p \vee \neg q \vee \neg r) \\
&\quad \wedge(\neg p \vee q) \\
&\quad \wedge(\neg p \vee r)
\end{aligned}$$

Tseitin Transformation: basics

Define: for specific (small) D, let *cnf*$(D) \equiv D$:

$$
\begin{aligned}
\mathit{cnf}(p \leftrightarrow \neg q) &= (p \vee q) \\
&\quad \wedge(\neg p \vee \neg q) \\[1em]
\mathit{cnf}(p \leftrightarrow (q \wedge r)) &= (p \vee \neg q \vee \neg r) \\
&\quad \wedge(\neg p \vee q) \\
&\quad \wedge(\neg p \vee r) \\[1em]
\mathit{cnf}(p \leftrightarrow (q \vee r)) &= (\neg p \vee q \vee r) \\
&\quad \wedge(p \vee \neg q) \\
&\quad \wedge(p \vee \neg r)
\end{aligned}
$$

Tseitin Transformation: basics

Define: for specific (small) D, let *cnf*$(D) \equiv D$:

$$\begin{aligned} \mathit{cnf}(p \leftrightarrow \neg q) \quad = \quad & (p \vee q) \\ & \wedge(\neg p \vee \neg q) \end{aligned}$$

$$\begin{aligned} \mathit{cnf}(p \leftrightarrow (q \wedge r)) \quad = \quad & (p \vee \neg q \vee \neg r) \\ & \wedge(\neg p \vee q) \\ & \wedge(\neg p \vee r) \end{aligned}$$

$$\begin{aligned} \mathit{cnf}(p \leftrightarrow (q \vee r)) \quad = \quad & (\neg p \vee q \vee r) \\ & \wedge(p \vee \neg q) \\ & \wedge(p \vee \neg r) \end{aligned}$$

$$\begin{aligned} \mathit{cnf}(p \leftrightarrow (q \leftrightarrow r)) \quad = \quad & (p \vee q \vee r) \\ & \wedge(p \vee \neg q \vee \neg r) \\ & \wedge(\neg p \vee q \vee \neg r) \\ & \wedge(\neg p \vee \neg q \vee r) \end{aligned}$$

Tseitin Transformation

Example: $\neg s \wedge p \leftrightarrow ((q \rightarrow r) \vee p)$

Tseitin Transformation

Example: $\neg s \wedge p \leftrightarrow ((q \rightarrow r) \vee p)$

Step 1: name every non-literal subformula of A (including A).

Tseitin Transformation

Example: $\neg s \wedge p \leftrightarrow ((q \rightarrow r) \vee p)$

Step 1: name every non-literal subformula of A (including A).

For a subformula D of A we define:

- $n_D = D$ if D is a literal
- $n_D =$ *the name of* D, otherwise

Tseitin Transformation

Example: $\neg s \wedge p \leftrightarrow ((q \rightarrow r) \vee p)$

Step 1: name every non-literal subformula of A (including A).

For a subformula D of A we define:

- $n_D = D$ if D is a literal
- $n_D =$ *the name of* D, otherwise

Step 2: take the conjunction of:

Tseitin Transformation

Example: $\neg s \wedge p \leftrightarrow ((q \rightarrow r) \vee p)$

Step 1: name every non-literal subformula of A (including A).

For a subformula D of A we define:

- $n_D = D$ if D is a literal
- $n_D =$ *the name of* D, otherwise

Step 2: take the conjunction of:

- n_A

Tseitin Transformation

Example: $\neg s \land p \leftrightarrow ((q \to r) \lor p)$

Step 1: name every non-literal subformula of A (including A).

For a subformula D of A we define:

- $n_D = D$ if D is a literal
- $n_D =$ *the name of* D, otherwise

Step 2: take the conjunction of:

- n_A
- *cnf*$(n_D \leftrightarrow \neg n_E)$ for every non-literal subformula D of the shape $\neg E$

Tseitin Transformation

Example: $\neg s \wedge p \leftrightarrow ((q \rightarrow r) \vee p)$

Step 1: name every non-literal subformula of A (including A).

For a subformula D of A we define:

- $n_D = D$ if D is a literal
- $n_D =$ *the name of* D, otherwise

Step 2: take the conjunction of:

- n_A
- *cnf*$(n_D \leftrightarrow \neg n_E)$ for every non-literal subformula D of the shape $\neg E$
- *cnf*$(n_D \leftrightarrow (n_E \diamond n_F))$ for every subformula D of the shape $E \diamond F$

Tseitin Transformation

Theorem

For every propositional formula A we have:

Tseitin Transformation

Theorem

For every propositional formula A we have:

A is satisfiable if and only if $T(A)$ is satisfiable.

Tseitin Transformation

Theorem

For every propositional formula A we have:

A is satisfiable if and only if $T(A)$ is satisfiable.

Proof sketch:

Tseitin Transformation

Theorem

For every propositional formula A we have:

A is satisfiable if and only if $T(A)$ is satisfiable.

Proof sketch:

$\Leftarrow$ A satisfying assignment for $T(A)$ restricting to the variables from A yields a satisfying assignment for A.

$\Rightarrow$ A satisfying assignment for A is extended to a satisfying assignment for $T(A)$ by giving n_D the value of D obtained from the satisfying assignment for A. $\square$

Tseitin Transformation summary

For every propositional formula A we have:

Tseitin Transformation summary

For every propositional formula A we have:

- A is satisfiable if and only if $T(A)$ is satisfiable.

Tseitin Transformation summary

For every propositional formula A we have:

- A is satisfiable if and only if $T(A)$ is satisfiable.
- $T(A)$ contains two types of variables.

Tseitin Transformation summary

For every propositional formula A we have:

- A is satisfiable if and only if $T(A)$ is satisfiable.
- $T(A)$ contains two types of variables.
- The size of $T(A)$ is linear in the size of A.

Tseitin Transformation summary

For every propositional formula A we have:

- A is satisfiable if and only if $T(A)$ is satisfiable.
- $T(A)$ contains two types of variables.
- The size of $T(A)$ is linear in the size of A.
- $T(A)$ is a **3-CNF**.

Tseitin Transformation summary

For every propositional formula A we have:

- A is satisfiable if and only if $T(A)$ is satisfiable.
- $T(A)$ contains two types of variables.
- The size of $T(A)$ is linear in the size of A.
- $T(A)$ is a **3-CNF**.
- To determine satisfiability of A: run a SAT-solver on $T(A)$.

Some particularly difficult formulas

Some particularly difficult formulas

Variables: P_{yx} for $y \in \{1, \ldots, n\}$ and $x \in \{1, \ldots, n+1\}$

Some particularly difficult formulas

Variables: P_{yx} for $y \in \{1, \ldots, n\}$ and $x \in \{1, \ldots, n+1\}$

Define:

$$C_n = \bigwedge_{x=1}^{n+1} (\bigvee_{y=1}^{n} P_{yx})$$

Some particularly difficult formulas

Variables: P_{yx} for $y \in \{1, \ldots, n\}$ and $x \in \{1, \ldots, n+1\}$

Define:

$$C_n = \bigwedge_{x=1}^{n+1} (\bigvee_{y=1}^{n} P_{yx})$$

$$R_n = \bigwedge_{y=1}^{n} \bigwedge_{1 \leq j < k \leq n+1} (\neg P_{yj} \vee \neg P_{yk})$$

Some particularly difficult formulas

Variables: P_{yx} for $y \in \{1, \ldots, n\}$ and $x \in \{1, \ldots, n+1\}$

Define:

$$C_n = \bigwedge_{x=1}^{n+1} (\bigvee_{y=1}^{n} P_{yx})$$

$$R_n = \bigwedge_{y=1}^{n} \bigwedge_{1 \le j < k \le n+1} (\neg P_{yj} \vee \neg P_{yk})$$

$$PF_n = C_n \wedge R_n$$

Intuition for the pigeonhole formulas

$$\begin{matrix} P_{11} & P_{12} & \cdots & P_{1,n+1} \\ P_{21} & P_{22} & \cdots & P_{2,n+1} \\ \vdots & \vdots & & \vdots \\ P_{n1} & P_{n2} & \cdots & P_{n,n+1} \end{matrix}$$

Intuition for the pigeonhole formulas

$$\begin{array}{cccc}
P_{11} & P_{12} & \cdots & P_{1,n+1} \\
P_{21} & P_{22} & \cdots & P_{2,n+1} \\
\vdots & \vdots & & \vdots \\
P_{n1} & P_{n2} & \cdots & P_{n,n+1}
\end{array}$$

$$C_n = \bigwedge_{x=1}^{n+1} (\bigvee_{y=1}^{n} P_{yx})$$

Validity of C_n: in every column at least one variable is true.

Intuition for the pigeonhole formulas

$$\begin{array}{cccc} P_{11} & P_{12} & \cdots & P_{1,n+1} \\ P_{21} & P_{22} & \cdots & P_{2,n+1} \\ \vdots & \vdots & & \vdots \\ P_{n1} & P_{n2} & \cdots & P_{n,n+1} \end{array}$$

$$C_n = \bigwedge_{x=1}^{n+1} (\bigvee_{y=1}^{n} P_{yx})$$

Validity of C_n: in every column at least one variable is true.

$$R_n = \bigwedge_{y=1}^{n} \bigwedge_{1 \leq j < k \leq n+1} (\neg P_{yj} \vee \neg P_{yk})$$

Validity of R_n: in every row at most one variable is true.

Pigeonhole formulas

The pigeon hole principle:

> *If $n+1$ pigeons fly out of a cage having n holes, then there is at least one hole through which at least two pigeons fly.*

Pigeonhole formulas

The pigeon hole principle:

> *If $n+1$ pigeons fly out of a cage having n holes, then there is at least one hole through which at least two pigeons fly.*

If one arbitrary disjunction is removed from the big conjunction, then the resulting formula is always satisfiable.

Pigeonhole formulas

The pigeon hole principle:

> *If $n+1$ pigeons fly out of a cage having n holes, then there is at least one hole through which at least two pigeons fly.*

If one arbitrary disjunction is removed from the big conjunction, then the resulting formula is always satisfiable.

Proving unsatisfiability of PF_n automatically is **hard**: typically exponential in n.

Pigeonhole formulas

The pigeon hole principle:

> *If $n+1$ pigeons fly out of a cage having n holes, then there is at least one hole through which at least two pigeons fly.*

If one arbitrary disjunction is removed from the big conjunction, then the resulting formula is always satisfiable.

Proving unsatisfiability of PF_n automatically is **hard**: typically exponential in n.

PF_n and modifications are a good test case for implementations of methods for SAT.

The practical assignment: SAT or SMT

Remember that you are not limited to DIMACS format!

The practical assignment: SAT or SMT

Remember that you are not limited to DIMACS format!

No need to bitblast or implement the Tseitin Transformation.

The practical assignment: SAT or SMT

Remember that you are not limited to DIMACS format!

No need to bitblast or implement the Tseitin Transformation.

Z3 (and other SAT/SMT-solvers) typically also accept SMT format.

The practical assignment: SAT or SMT

Remember that you are not limited to DIMACS format!

No need to bitblast or implement the Tseitin Transformation.

Z3 (and other SAT/SMT-solvers) typically also accept SMT format.

You only need to use the basics.

The practical assignment: SAT or SMT

Remember that you are not limited to DIMACS format!

No need to bitblast or implement the Tseitin Transformation.

Z3 (and other SAT/SMT-solvers) typically also accept SMT format.

You only need to use the basics.

(Internally, Tseitin Transformation and perhaps bitblasting are done.)

Quiz

1. Provide a SAT encoding that expresses that $a + b = c$, where a, b, c are all two-bit binary numbers.
2. Provide a SAT encoding that expresses $a > b$ when $a, b \in \{0, \ldots, 3\}$ are encoded as unary numbers.
3. Why is it sometimes useful to use the unary encoding instead of binary?
4. Use Tseitin's Transformation to give a CNF whose satisfiability is equivalent to:

$$x \leftrightarrow ((y \wedge \neg x) \wedge (z \rightarrow w))$$

5. Why are pigeonhole formulas a good testcase for SAT solvers?