1

Handout for lecture 2:

# Beyond the basics of SAT: integers and program verification

# 1. Binary Arithmethic

2

## Arithmetic in proposition logic

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

**Solution:** use the binary representation!

The binary representation

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

of an $n$-bit number $a$ means that each $a_i \in \{0, 1\}$ and:

$$a = \sum_{i=1}^{n} a_i * 2^{i-1}$$

3

## 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!

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

4

## Adding two $n$-bit numbers

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

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

So to add $a_n a_{n-1} \ldots a_1$ to $b_n b_{n-1} \ldots b_1$ we need **carries** $c_n c_{n-1} \cdots c_1$. We also include a carry $c_0$ (which will always be 0) because it makes it easier to specify the other requirements.

For this to be a correct computation, we must satisfy the following properties:

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

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

  *Note:* this should be read as $((a_i \leftrightarrow b_i) \leftrightarrow c_i) \leftrightarrow d_i$.

- for all $i < n$: $c_i = 1$ if and only if $a_i + b_i + c_{i-1} > 1$.

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

- $c_0 = 0$ (no initial carry) and $c_n = 0$ (no overflow).

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

---

## 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:** Let $\phi$ be the conjunction of all of these formulas.

Then the result of $d = 17 + 11$ follows from the unique satisfying assignment for:

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

In case $d$ does not fit in $n$ digits then $c_n$ would be forced to be 1, by which no satisfying assignment exists. This is solved by adding a leading 0 to $a$ and $b$, and using $(n+1)$-bit addition.

We get subtraction for free! Computing $b = d - a$ follows from satisfiability of:

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

---

## Multiplication in primary school

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

In decimal:

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

That is, we go through $b$, from right to left. We keep track of a list of numbers. In every step, we multiply $b_i$ by $a$, and place the result in a list, shifted to the left by the number of steps we have already taken.

There is not really any need to use this order. We can also go from left to right through the number, and shift one position to the right in each step. This would give roughly the following algorithm:

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

Note that this indeed gives us:

- $d := 0$
- $i := 3; d := 10 * 0 + 4 * 123 = 492$
- $i := 2; d := 10 * 492 + 5 * 123 = 4920 + 615$
- $i := 1; d := 10 * (4920 + 615) + 6 * 123 = 49200 + 6150 + 738 = 56088$

## Binary multiplication

Translating this algorithm to binary integers, we obtain the following:

```
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 $d = 99$ $(45 * 2 + 9)$

---

8

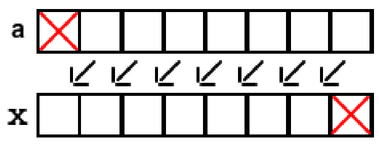
## Binary multiplication

```
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 $d = b * a$

As $a, b$ are numbers, represented as boolean vectors, we will write $\vec{a} = (a_n, \ldots, a_1)$, and so on.

For representing $\vec{x} = 2\vec{a}$ we need to encode the intuition:



This is done by the formula:

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

---

9

## 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!**

In SAT, there is no notion of changeable variables; each boolean variable is either set to true or to false, in one go. Hence, to model a "programming variable" that is updated over time, we need to use *multiple* variables. Hence, we 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 $r_{i,j}$.

The vector $(r_{in}, \ldots, r_{i1})$ indicates the value of $d$ at the start of the for loop for the given value of $i$.

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

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

Let us update the algorithm with these variables which are only *set* rather than *updated*:

```
$\vec{r}_n = 0$

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

---

10

## Bringing it all together: multiplication

The requirement

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

is now described by the formula:

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

$$\bigwedge_{j=1}^{n} \neg r_{nj} \qquad\qquad \vec{r}_n := 0$$

$$\wedge$$

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

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

---

11

## Factorisation

In this way we can do all kinds of arithmetic by SAT, for instance factorise a number.

**Challenge:** Is 1234567891 prime? And 1234567897?

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

(We have $(a_n a_{n-1} \dots a_2 a_1) > 1$ if $a_n \vee \dots \vee a_2$ holds.)

$r$ is prime $\iff$ $\text{fac}(r)$ is unsatisfiable.

If satisfiable, then $a, b$ represent factors, thus giving us prime factorisation almost for free.

**Answers:**

- fac(1234567891) is unsatisfiable, so 1234567891 is prime.

  Found by `minisat` or `yices` within 1 minute.

- fac(1234567897) is satisfiable, yielding

$$1234567897 = 1241 \times 994817$$

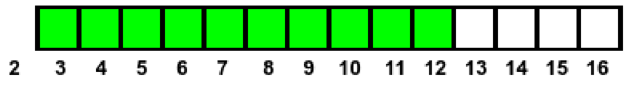
  found by `minisat` or `yices` within 1 second.

# 2. Unary arithmetic

12

## Unary arithmetic in proposition logic

**Idea:** instead of the binary encoding, implement a number in the range $i..j$ by $j-i$ booleans.



That is, to represent one number in this range we introduce the Boolean variables $x_{i+1},\dots,x_j$.

Here, the Boolean variable $x_k$ represents: $\vec{x} \geq k$. So if all these variables are false, $\vec{x} = i$ (since $\neg x_{i+1}$). If all are true, then $\vec{x} = j$ (since $x_j$ holds).

Unary integers must satisfy the following well-definedness condition:

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

This essentially states that $\vec{x} \geq k \to \vec{x} \geq k-1$, without which property reasoning about unary integers would not make sense.

13

## Unary addition

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

First, note that $c_{\min} = a_{\min} + b_{\min}$ and $c_{\max} = a_{\max} + b_{\max}$. We also have to introduce the requirements

$$\bigwedge_{i=c_{\min+1}}^{c_{\max}} c_i \to c_{i-1}$$

To impose that $\vec{c} = \vec{a} + \vec{b}$, it suffices to have $\vec{c} \geq \vec{a} + \vec{b}$ and $\vec{c} \leq \vec{a} + \vec{b}$. This is given by the following requirements:

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

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

Hence:

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

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

Or put differently:

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

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

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

---

14

## 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 |

All in all, while the binary representation is generally preferable when encoding integer variables that may take large values, there is also a place for the unary representation, especially when considering numbers within a small range. Experience shows that when we consider numbers whose ranges are at most 50 (or 100 if we do not add or multiply by more than a handful), the unary representation often works faster than the binary one.

Numbers like this occur in many practical problems.

# 3. Program verification

---

15

## Expressing programs in SAT

**Goal:** express *simple* integer programs in Boolean logic (so we can analyse their correctness).

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

That is, we can for instance have loops of the form `for i := 1 to 10 do P` (where `i` is only used as a counter and not as a variable that may be altered inside the loop), but we cannot have `while x > 0 do P` where it is not known beforehand how many steps the loop may take.

**Basic idea:**

- Express all integer variables `x` by sequences of boolean variables $\vec{x}_0$, in binary notation.
- For every step $i$ in the program, introduce a copy $x_i$ of every boolean variable $x$. Intuitively, $x_i$ represents the value of the boolean variable `x` after step $i$.
- $a := b$ 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` (where in each copy, `i` is replaced by the corresponding constant value).

---

16

## Program correctness by SAT

**Goal:** proving a property about a program.

Such a property is typically given by a **Hoare triple**:

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

Here

- $S$ is the program;
- $P$ is the **precondition**: the property assumed to hold initially;
- $Q$ is the **postcondition**: the property that should hold after the program has finished.

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

$$P_0 \;\wedge\; \neg Q_m$$

and prove that the resulting formula is unsatisfiable. Here, $P_0$ refers to the property $P$ using the variables at the start of the program, and $Q_m$ to the property $Q$ using the variables at the end of the program.

---

17

## Program correctness by SAT – example

For a very simple example demonstrating the core ideas, we consider a boolean array $a[1..m]$.

**CLAIM:** After doing

`for` $j := 1$ `to` $m-1$ `do` $a[j+1] := a[j]$

we have

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

In this case, we let the precondition be *true*, so this may be ignored.

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

Semantics of the $j$th iteration:

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

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

A SAT solver can show that the conjunction of all iterations, as well as the postcondition, is unsatisfiable.

---

18

## Expressing more elaborate programs in SAT

The same approach applies for somewhat more complicated programs. We still limit interest to programs with a bounded number of assignments, but also support:

- *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”, etc.
- *branching:* for an if-then-else statement `if cond then P else Q`:
  - add `skip` statements to `P` or `Q` to make them equally long
  - let `P` encode to $\varphi$ and `Q` to $\psi$;
    add requirements $(cond \to \varphi) \wedge (\neg cond \to \psi)$

---

## 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:

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

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

---

## A more complicated program correctness example

**Claim:** if $\underbrace{x < 20}_{\text{pre-condition}}$ at the start of the program, then $\underbrace{ret = 0}_{\text{post-condition}}$ at the end.

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

Let us first make the lengths of the branches equal, so the number of steps is fixed rather than merely bounded.

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

And then let's note down the step for each statement.

```
(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

---

21

## A more complicated program correctness example

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

---

22

## Program correctness summary

**Overall:** in this way verification of a rich class of imperative programs can be expressed in SAT.

To do this for integers we have to fix a number of bits, and encode all arithmetical operations and relations ourselves.

We have also (briefly) discussed **SMT: satisfiability modulo theories**. There, we can express (in)equalities on linear expressions like $a < 3 * b + c$ directly, without bounding the integers.

(However, note that in programs we do not actually have unbounded integers!)

# 4. Tseitin transformation

23

## The need for CNF

**So far:**

- SAT solvers can handle **CNF** as input format. We also encoded both Sudoku and the $n$-queens problem straight as a CNF.

- However, in this lecture I have completely ignored this requirement: our implementation of binary arithmetic, and also program analysis, used much more complex formulas. Also in many other problems, we naturally encounter formulas that are hard to specify directly as a CNF.

Hence, we want to establish satisfiability of arbitrary propositions by first transforming the proposition to CNF.

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

This is always possible:

every 0 in the truth table yields a clause

proposition $\equiv$ conjunction of these clauses

(Often it can be done much more efficiently than this.)

**However:** this is worst-case exponential. And that is not a *rare* worst-case: it often happens that every CNF that we can find which is logically equivalent to a given proposition is unacceptably big. In some cases, this may just be because we use the wrong method to find a CNF – there exists a short CNF, but we have to use the right approach to find it. However, there are also cases where *every* CNF is guaranteed to have at least a certain size.

24

## Formulas whose CNF is always large

Consider the following proposition in $n$ variables:

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

This formula yields true if and only if 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 $p_i$ for some fixed variable $p_i$.

Give values to the remaining variables to make $C$ false. Then, $B$ is false. Hence, for this assignment of variables, $B$ is false regardless of the value assigned to $p_i$.

Now give a value to $p_i$ such that $A$ yields *true*. This is always possible, because either the number of *false*-variables among the rest is even or it is odd.

But now we have a contradiction with $A \equiv B$. □

**Observation:** The truth table of $A$ contains exactly $2^{n-1}$ zeroes: half of all entries.

Every clause containing $n$ literals yields exactly one zero in the truth table.

According to the claim all clauses of $B$ are of this shape.

Hence $B$ consists of $2^{n-1}$ clauses.

**Conclusion:** Any CNF $B$ equivalent to $A$ has size exponential in the size of $A$.

This is unacceptably large for practical purposes.

---

25

## Alternatives to an equivalent CNF

Hence we are looking for a way to transform any arbitrary propositional formula $A$ to a CNF $B$ such that:

- $A$ is satisfiable if and only if $B$ is satisfiable;
- the size of $B$ is **linear** in the size of $A$. (Actually, **polynomial** would suffice, but linear is better, and it is within reach!)

Note that we weaken the restriction that $A$ and $B$ are equivalent.

Such a construction is possible if we allow that $B$ contains a number of **fresh** variables.

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

---

26

## Tseitin Transformation: basics

For every formula $D$ on at most 3 variables there is a CNF $\mathit{cnf}(D)$ with $\mathit{cnf}(D) \equiv D$ and $\mathit{cnf}(D)$ contains at most 4 clauses; most pertinently:

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

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

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

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

---

## Tseitin Transformation

**Step 1:** introduce a new variable for every non-literal subformula of $A$ (including $A$ itself), the **name** of the subformula. Note that a *literal* is either a variable or a negated variable.

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:** the CNF $T(A)$, the **Tseitin transformation** of $A$, is defined to be the CNF consisting of the clauses:

- $n_A$
- the clauses of $\mathit{cnf}(n_D \leftrightarrow \neg n_E)$ for every non-literal subformula $D$ of the shape $\neg E$
- the clauses of $\mathit{cnf}(n_D \leftrightarrow (n_E \diamond n_F))$ for every subformula $D$ of the shape $E \diamond F$ for $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$

**Example:** $A : \underbrace{(\neg s \wedge p)}_{B} \leftrightarrow \underbrace{(\underbrace{(q \rightarrow r)}_{D} \vee \neg p)}_{C}$

yields $T(A)$ consisting of the clauses $A$ and

$$\left.\begin{array}{l} A \vee B \vee C \\ A \vee \neg B \vee \neg C \\ \neg A \vee \neg B \vee C \\ \neg A \vee B \vee \neg C \end{array}\right\} \mathit{cnf}(A \leftrightarrow (B \leftrightarrow C))$$

$$\left.\begin{array}{l} B \vee s \vee \neg p \\ \neg B \vee \neg s \\ \neg B \vee p \end{array}\right\} \mathit{cnf}(B \leftrightarrow (\neg s \wedge p))$$

$$\left.\begin{array}{l} \neg C \vee D \vee \neg p \\ C \vee \neg D \\ C \vee p \end{array}\right\} \mathit{cnf}(C \leftrightarrow (D \vee \neg p))$$

$$\left.\begin{array}{l} \neg D \vee r \vee \neg q \\ D \vee \neg r \\ D \vee q \end{array}\right\} \mathit{cnf}(D \leftrightarrow (q \rightarrow r))$$

28

## 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 $A$. $\square$

29

## 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: variables occurring in $A$ and variables representing names of subformulas of $A$.
- The size of $T(A)$ is linear in the size of $A$.
- Every clause in $T(A)$ contains at most 3 literals: $T(A)$ is a **3-CNF**.
- A fruitful approach to investigate satisfiability of $A$ is applying a modern CNF based SAT solver on $T(A)$.

# 5. Pigeonhole formulas

30

## Some particularly difficult formulas

SAT solvers can often handle problems with thousands of variables without much difficulty.

Now we give an example of a formula (the **pigeon hole formula**) that can be concluded to be unsatisfiable, hence is logically equivalent to *false*, but for which it is hard to conclude this directly from the formula itself.

Choose an integer number $n > 0$ and $n(n+1)$ boolean variables $P_{yx}$ for $y = 1, \ldots, n$ and $x = 1, \ldots, n+1$.

Define

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

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

$$PF_n = C_n \wedge R_n$$

31

## Intuition for the pigeonhole formulas

Put the variables in a matrix as follows:

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

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

Validity of $C_n$ means that in every column at least one variable is true.

Hence if $C_n$ holds then at least $n+1$ variables are true.

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

Validity of $R_n$ means that in every row at most one variable is true.
Note that this exactly means that there are no two $P$s in the same row that are both true.

Hence if $R_n$ holds then at most $n$ variables are true.

Hence $C_n$ and $R_n$ cannot both be valid, and therefore $PF_n = C_n \wedge R_n$ is unsatisfiable.

---

## Pigeonhole formulas

This counting argument is closely related to 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.

The formula $PF_n$ is called the **pigeon hole formula** for $n$.

The formula is a conjunction of

$$(n+1) + n \times \frac{n(n+1)}{2}$$

disjunctions.

The disjunctions are of the shape $\bigvee_{y=1}^{n} P_{yx}$ and $\neg P_{yx} \vee \neg P_{yk}$.

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

**In summary:**

$PF_n$ is an artificial unsatisfiable formula of size polynomial in $n$.

Minor modifications of $PF_n$ are satisfiable.

For most methods proving unsatisfiability of $PF_n$ automatically is hard: it can be done, but for most methods the number of steps is exponential in $n$.

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

# 6. Other

33

## The practical assignment: SAT or SMT

SAT solvers require input in dimacs format. Dimacs format does not support integers.

There is no need to manually implement bitblasting, or to transform an arbitrary propositional formula to dimacs format:

Several modern SAT solvers like `Z3` also accept SMT format (satisfiability modulo theories) of which the most basic instance coincides with arbitrary propositional formulas.

Internally, these solvers apply a transformation to move formulas into a dimacs-like format. Depending on the solver, they may also encode integers as binary numbers or apply number-sensitive techniques. (Or both.)

34

## 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:

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

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