Verifying SAT Encodings in Lean

Cayden R. Codel
Advised by Marijn J. H. Heule and Jeremy Avigad
Bryan Parno, committee member

Thesis at
http://crcodel.com/research/verified-encodings-thesis.pdf

Overview

An encoding transforms one problem to another.

Overview

An encoding transforms one problem to another.

The SAT toolchain is trustworthy—except for encodings.

Overview

An encoding transforms one problem to another.

The SAT toolchain is trustworthy—except for encodings.

We present a library of verified SAT encodings in Lean.

Overview

An encoding transforms one problem to another.

The SAT toolchain is trustworthy—except for encodings.

We present a library of verified SAT encodings in Lean.

Our correctness proofs improve confidence in solver results.

Overview

An encoding transforms one problem to another.

The SAT toolchain is trustworthy—except for encodings.

We present a library of verified SAT encodings in Lean.

Our correctness proofs improve confidence in solver results.

We develop general proof methods and data structures.

What is an encoding?

The Lean theorem prover

Encoding verification library

Encodings and proofs of correctness

Conclusion

What is an encoding?

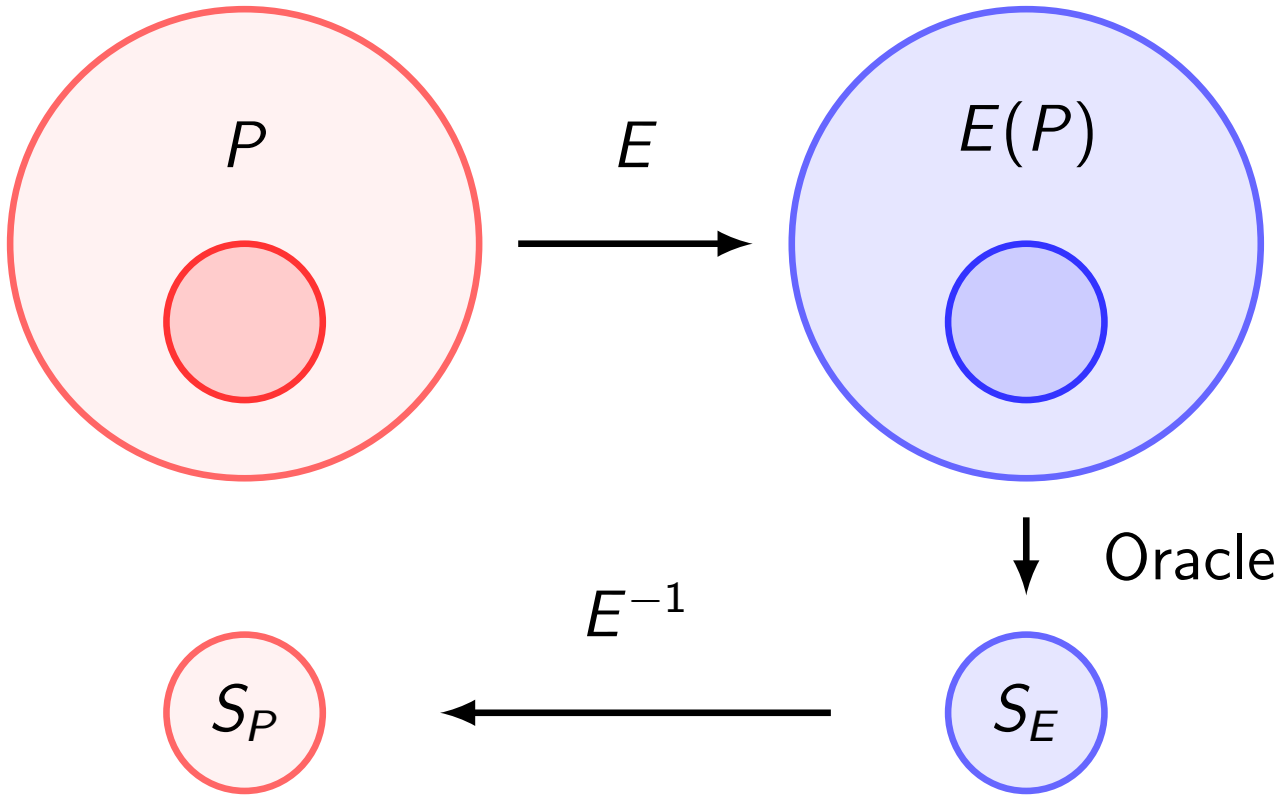
The Lean theorem prover

Encoding verification library

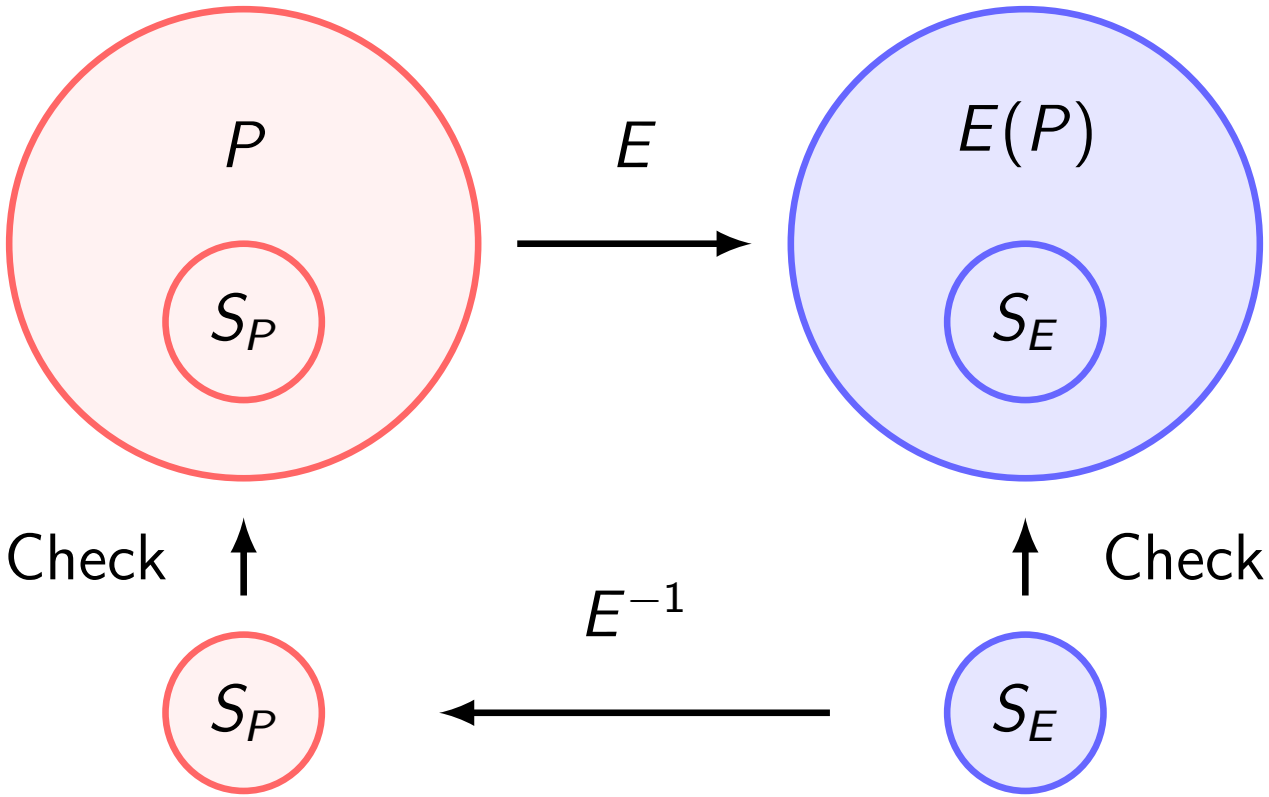
Encodings and proofs of correctness

Conclusion

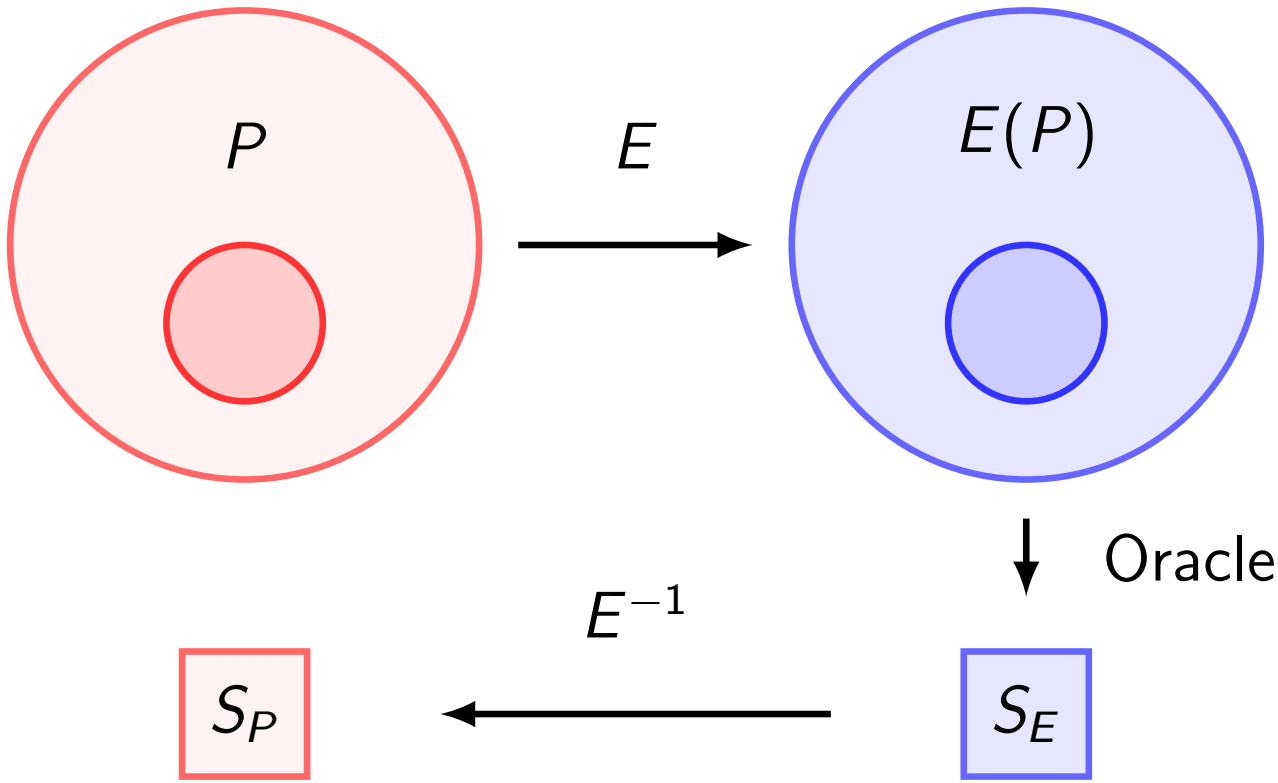
What is an encoding?



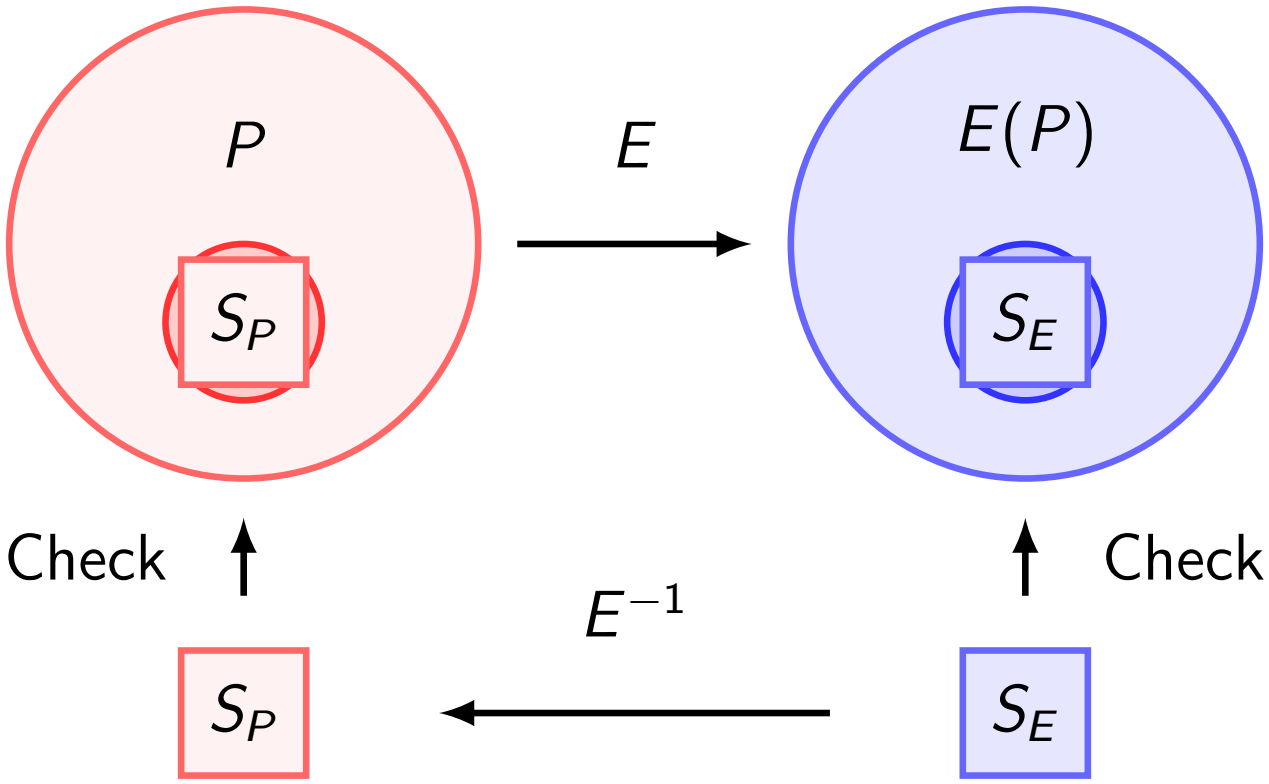
What is an encoding?



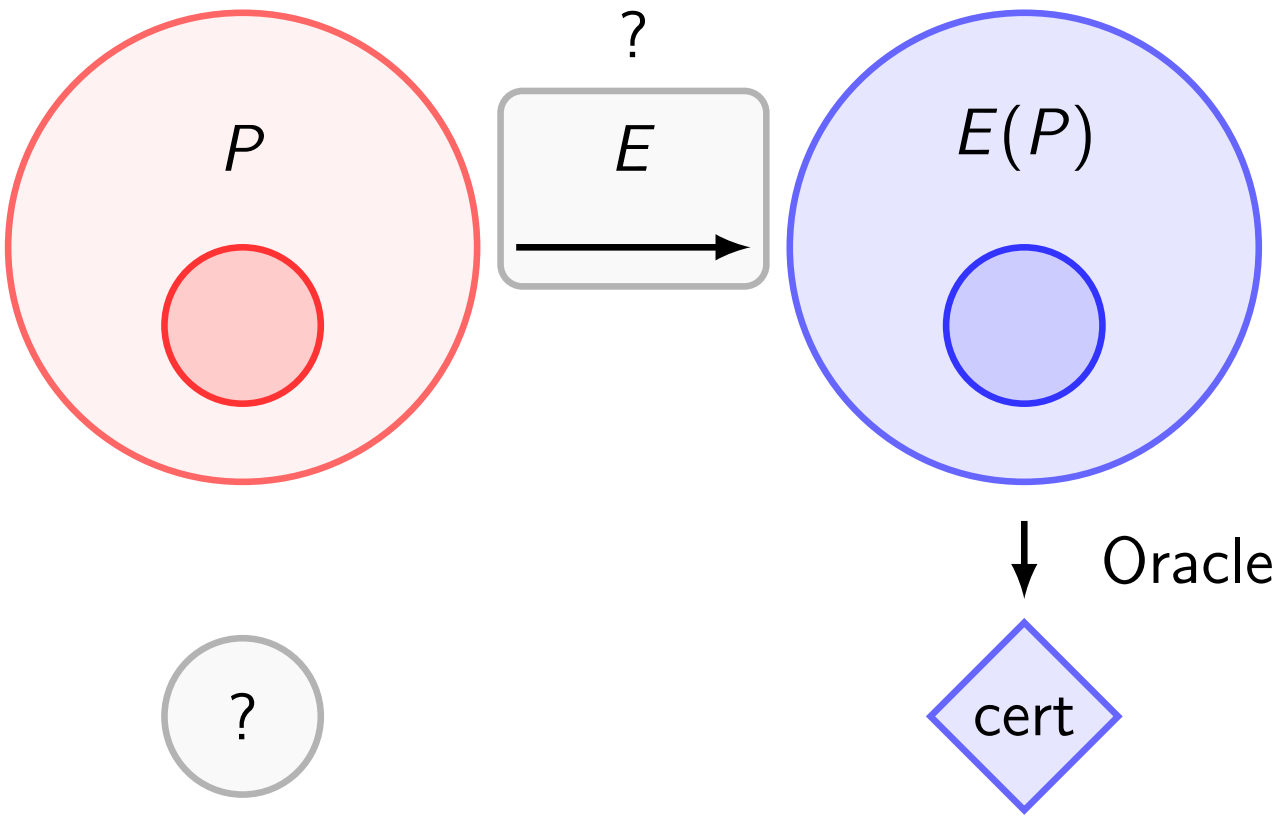
What is an encoding?



What is an encoding?



What is an encoding?



What is an encoding?

$$F = (x_1 \vee x_2) \wedge (\overline{x}_1 \vee x_3) \wedge (\overline{x}_2 \vee \overline{x}_3)$$

What is an encoding?

$$F = (x_1 \vee x_2) \wedge (\overline{x}_1 \vee x_3) \wedge (\overline{x}_2 \vee \overline{x}_3)$$

$\tau = \{x_1 = \top, x_2 = \bot, x_3 = \top\}$

Variable set $V(\tau) = \{x_1, x_2, x_3\}$

What is an encoding?

$$F = (x_1 \vee x_2) \wedge (\overline{x}_1 \vee x_3) \wedge (\overline{x}_2 \vee \overline{x}_3)$$

$\tau = \{x_1 = \top, x_2 = \bot, x_3 = \top\}$

Variable set $V(\tau) = \{x_1, x_2, x_3\}$

τ satisfies F (written $\tau(F) = \top$) and F is satisfiable.

What is an encoding?

$$F = (\mathbf{x}_1 \vee x_2) \wedge (\overline{x}_1 \vee \mathbf{x}_3) \wedge (\overline{\mathbf{x}}_2 \vee \overline{x}_3)$$

$\tau = \{x_1 = \top, x_2 = \bot, x_3 = \top\}$

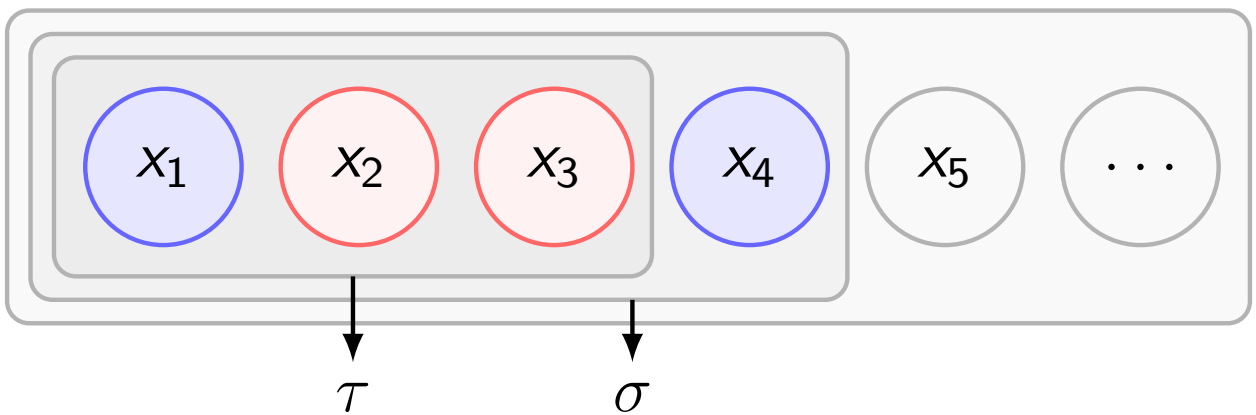
Variable set $V(\tau) = \{x_1, x_2, x_3\}$

τ satisfies F (written $\tau(F) = \top$) and F is satisfiable.

What is an encoding?

Definition: Extended truth assignment

Let τ be an assignment on $V(\tau)$. Then σ extends τ if $V(\tau) \subseteq V(\sigma)$ and $\forall x \in V(\tau),\ \tau(x) = \sigma(x)$.



What is an encoding?

Definition: Extended truth assignment

Let τ be an assignment on $V(\tau)$. Then σ extends τ if $V(\tau) \subseteq V(\sigma)$ and $\forall x \in V(\tau),\ \tau(x) = \sigma(x)$.

Definition: Encoding of a Boolean relation

Let R be a Boolean relation, F be a formula, and $x_1, \ldots, x_n$ be variables representing the inputs of R. Then F encodes R iff: $\forall \tau$ on $x_1, \ldots, x_n$, $R(\tau(x_1), \ldots, \tau(x_n))$ iff F is satisfied by some assignment that extends τ.

What is an encoding?

Definition: Extended truth assignment

Let τ be an assignment on $V(\tau)$. Then σ extends τ if $V(\tau) \subseteq V(\sigma)$ and $\forall x \in V(\tau),\ \tau(x) = \sigma(x)$.

Definition: Encoding of a Boolean relation

Let R be a Boolean relation, F be a formula, and $x_1, \ldots, x_n$ be variables representing the inputs of R. Then F encodes R iff: $\forall \tau$ on $x_1, \ldots, x_n$, $R(\tau(x_1), \ldots, \tau(x_n))$ iff F is satisfied by some assignment that extends τ.

Special case $R := (f(\tau(x_1), \ldots, \tau(x_n)) = \top)$. Then R is defined by f.

What is an encoding?

The Lean theorem prover

Encoding verification library

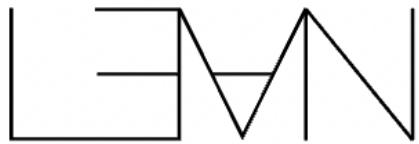
Encodings and proofs of correctness

Conclusion

The Lean theorem prover

Lean is an interactive theorem prover based on the CoIC.

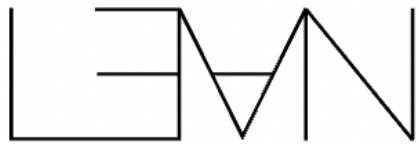
The Lean theorem prover



Lean is an interactive theorem prover based on the CoIC.

We used Lean 3; Lean 4 is currently in development.

The Lean theorem prover



Lean is an interactive theorem prover based on the CoIC.

We used Lean 3; Lean 4 is currently in development.

Quick demo!

What is an encoding?

The Lean theorem prover

Encoding verification library

Encodings and proofs of correctness

Conclusion

Encoding verification library

Open-source on Github

Encoding verification library

Open-source on Github

Contains:

- Data structures (CNF representations, `gensym`)
- Supporting lemmas and theorems
- Encoding proofs of correctness

Encoding verification library

Open-source on Github

Contains:

- Data structures (CNF representations, `gensym`)
- Supporting lemmas and theorems
- Encoding proofs of correctness

Basis for future verification efforts

Encoding verification library

Open-source on Github

Contains:

- Data structures (CNF representations, `gensym`)
- Supporting lemmas and theorems
- Encoding proofs of correctness

Basis for future verification efforts

Encoding verification library - CNF representations

```
inductive literal (V : Type*)
| Pos (v : V) : literal
| Neg (v : V) : literal

def clause (V : Type*) := list (literal V)
def cnf (V : Type*) := list (clause V)
def assignment (V : Type*) := V -> bool
```

Encoding verification library - CNF representations

```
def eval (τ : assignment V) (c : clause V) : bool :=
  c.foldr (λ l b, b || (l.eval τ)) ff
```

`l.eval` is syntactic sugar for `literal.eval l`

`tt` is true, `ff` is false

Encoding verification library - CNF representations

```
theorem eval_tt_iff_exists_literal_eval_tt
  {τ : assignment V} {c : clause V} :
  c.eval τ = tt ↔ ∃ (l : literal V), l ∈ c → l.eval τ = tt

theorem eval_ff_iff_forall_literal_eval_ff
  {τ : assignment V} {c : clause V} :
  c.eval τ = ff ↔ ∀ (l : literal V), l ∈ c → l.eval τ = ff
```

Encoding verification library - `gensym`

```
structure gensym (α : Type *) := (offset : nat)
  (f : nat -> α) (f_inj : injective f)
```

Inspired by de Bruijn indices, Hilbert's hotel, Lisp gensym

Encoding verification library - `gensym`

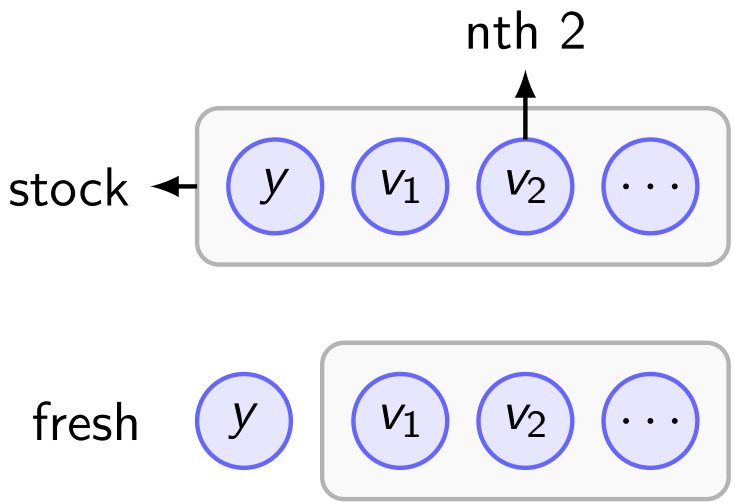
```
def stock : set α := { a | ∃ (n : nat), g.f (g.offset + n) = a }
```

Encoding verification library - gensym

```
def stock : set α := { a | ∃ (n : nat), g.f (g.offset + n) = a }

def fresh (g : gensym α) : (α × gensym α) :=
  ⟨g.f (g.offset), ⟨g.offset + 1, g.f, g.f_inj⟩⟩

def nth (g : gensym α) (n : nat) : α := g.f (g.offset + n)
```

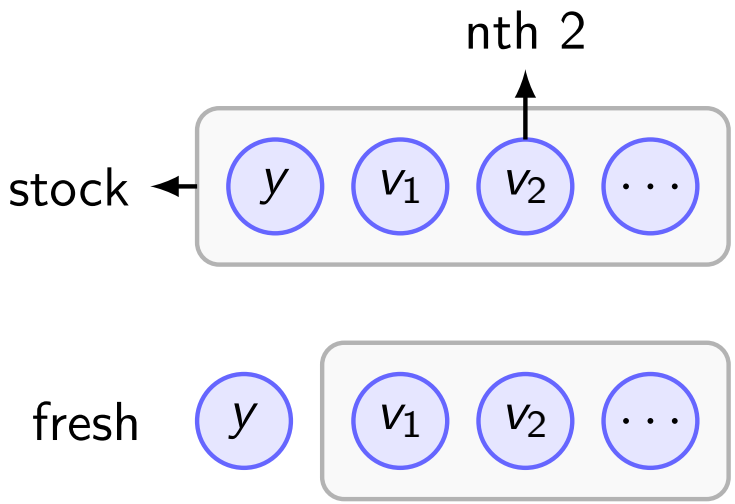
Encoding verification library - `gensym`

```
def stock : set α := { a | ∃ (n : nat), g.f (g.offset + n) = a }

def fresh (g : gensym α) : (α × gensym α) :=
  ⟨g.f (g.offset), ⟨g.offset + 1, g.f, g.f_inj⟩⟩

def nth (g : gensym α) (n : nat) : α := g.f (g.offset + n)
```



Encoding verification library - gensym

```
def stock : set α := { a | ∃ (n : nat), g.f (g.offset + n) = a }

def fresh (g : gensym α) : (α × gensym α) :=
  ⟨g.f (g.offset), ⟨g.offset + 1, g.f, g.f_inj⟩⟩

def nth (g : gensym α) (n : nat) : α := g.f (g.offset + n)
```



In practice, gensym is often a global int counter.

What is an encoding?

The Lean theorem prover

Encoding verification library

Encodings and proofs of correctness

Conclusion

Encodings and proofs of correctness

We proved correct:

- Two encodings of XOR (direct and Tseitin)
- Two encodings of AMO (direct and Sinz)
- One encoding of AMK (Sinz)

Encodings and proofs of correctness - XOR

Definition

$\mathrm{XOR}(x_1, \ldots, x_n) := \bigoplus_{i=1}^{n} x_i = \top$ iff an odd number of x_i are true

Encodings and proofs of correctness - XOR

Definition

$\mathrm{XOR}(x_1, \ldots, x_n) := \bigoplus_{i=1}^{n} x_i = \top$ iff an odd number of x_i are true

Important properties

$$\top \oplus x \equiv \overline{x} \qquad\qquad \bot \oplus x \equiv x$$

Encodings and proofs of correctness - XOR, DIRECTXOR

$$\textsc{directxor}(x_1, \ldots, x_n) = \bigwedge_{\text{even \# of negations}} \left(\pm x_1 \vee \cdots \vee \pm x_n \right)$$

Encodings and proofs of correctness - XOR, DIRECTXOR

$$\textsc{directxor}(x_1, \ldots, x_n) = \bigwedge_{\text{even \# of negations}} \left(\pm x_1 \vee \cdots \vee \pm x_n \right)$$

$$\textsc{directxor}(x_1, \overline{x}_2, x_3) =$$
$$(x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$$

Encodings and proofs of correctness - XOR, DIRECTXOR

$$\text{DIRECTXOR}(x_1, \ldots, x_n) = \bigwedge_{\text{even \# of negations}} \left(\pm x_1 \vee \cdots \vee \pm x_n \right)$$

$$\text{DIRECTXOR}(x_1, \overline{x}_2, x_3) =$$
$$(x_1 \vee \overline{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$$

Generates 2^{n-1} clauses on n input variables.

Poor performance for solvers above small n.

Encodings and proofs of correctness - XOR, DIRECTXOR

```
def direct_xor : list (literal V) → cnf V
| []        := [[]]
| (l :: ls) := (explode (map var ls)).map
    (λ c, ite (!bodd (c.count_flips ls)) (l :: c) (l.flip :: c))
```

Encodings and proofs of correctness - XOR, DIRECTXOR

```
def direct_xor : list (literal V) → cnf V
| []        := [[]]
| (l :: ls) := (explode (map var ls)).map
    (λ c, ite (!bodd (c.count_flips ls)) (l :: c) (l.flip :: c))
```

Encodings and proofs of correctness - XOR, DIRECTXOR

```
def direct_xor : list (literal V) → cnf V
| []        := [[]]
| (l :: ls) := (explode (map var ls)).map
    (λ c, ite (!bodd (c.count_flips ls)) (l :: c) (l.flip :: c))
```

`explode` generates all possible (ordered) positive and negative literals on input list of variables.

Encodings and proofs of correctness - XOR, DIRECTXOR

```
def direct_xor : list (literal V) → cnf V
| []        := [[]]
| (l :: ls) := (explode (map var ls)).map
    (λ c, ite (!bodd (c.count_flips ls)) (l :: c) (l.flip :: c))
```

`explode` generates all possible (ordered) positive and negative literals on input list of variables.

$$\texttt{explode}([x, y, z]) = [[x, y, z], [\overline{x}, y, z], [x, \overline{y}, z], [x, y, \overline{z}], [\overline{x}, \overline{y}, \overline{z}], [x, \overline{y}, \overline{z}], [\overline{x}, y, \overline{z}], [\overline{x}, \overline{y}, z]]$$

Encodings and proofs of correctness - XOR, DIRECTXOR

```
def direct_xor : list (literal V) → cnf V
| []        := [[]]
| (l :: ls) := (explode (map var ls)).map
    (λ c, ite (!bodd (c.count_flips ls)) (l :: c) (l.flip :: c))
```

Add in `l` according to parity of clause.

Encodings and proofs of correctness - XOR, DIRECTXOR

Theorem: DIRECTXOR encodes XOR.

Proof.
Parity argument.

Encodings and proofs of correctness - XOR, DIRECTXOR

Theorem: DIRECTXOR encodes XOR.

Proof.

Parity argument.

$\text{XOR}(\tau(x_1), \ldots, \tau(x_n)) = \top \Rightarrow$ all clauses evaluate to true.
$\text{XOR}(\tau(x_1), \ldots, \tau(x_n)) = \bot \Rightarrow$ there is a false clause.

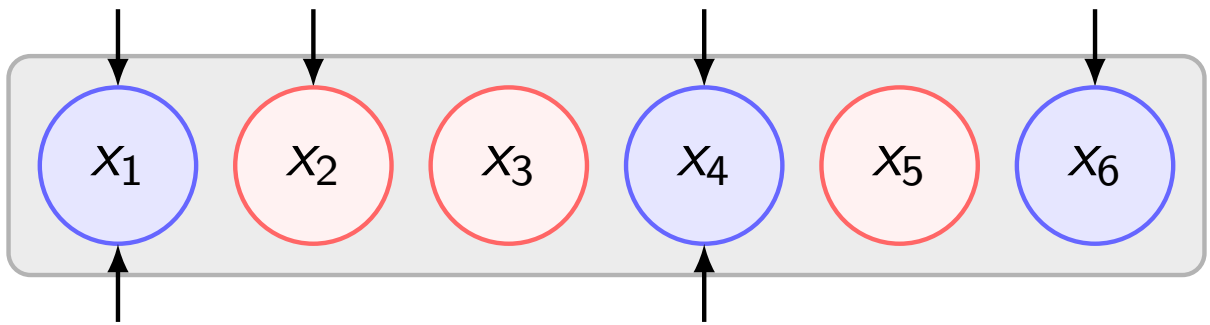
Encodings and proofs of correctness - XOR, DIRECTXOR

Theorem: DIRECTXOR encodes XOR.

Proof.

Parity argument.

$\text{XOR}(\tau(x_1), \ldots, \tau(x_n)) = \top \Rightarrow$ all clauses evaluate to true.
$\text{XOR}(\tau(x_1), \ldots, \tau(x_n)) = \bot \Rightarrow$ there is a false clause.



Odd number of the x_i true $\neq$ even number of negations. □

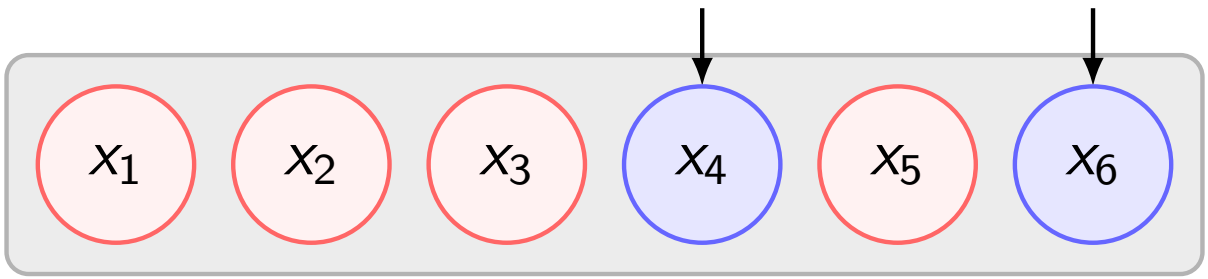
Encodings and proofs of correctness - XOR, DIRECTXOR

Theorem: DIRECTXOR encodes XOR.

Proof.

Parity argument.

$\text{XOR}(\tau(x_1), \ldots, \tau(x_n)) = \top \Rightarrow$ all clauses evaluate to true.
$\text{XOR}(\tau(x_1), \ldots, \tau(x_n)) = \bot \Rightarrow$ there is a false clause.



Identify the clause with exactly the true x_i negated. □

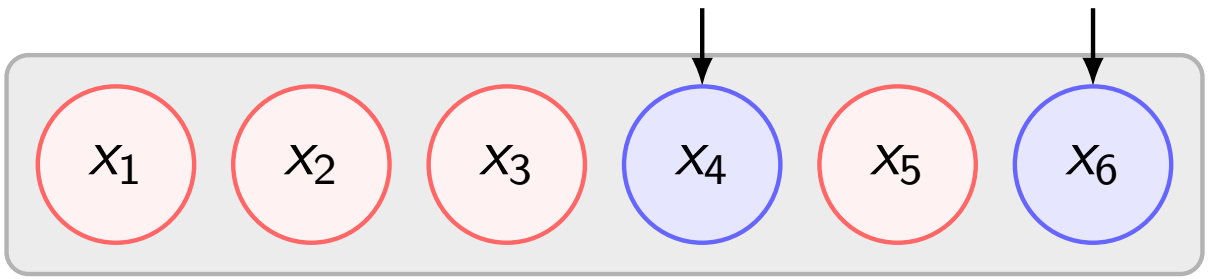
Encodings and proofs of correctness - XOR, DIRECTXOR

Theorem: DIRECTXOR encodes XOR.

Proof.

Parity argument.

$\text{XOR}(\tau(x_1), \ldots, \tau(x_n)) = \top \Rightarrow$ all clauses evaluate to true.
$\text{XOR}(\tau(x_1), \ldots, \tau(x_n)) = \bot \Rightarrow$ there is a false clause.



Identify the clause with exactly the true x_i negated. □

Takeaway: clauses disallow falsifying inputs—stick to the semantics.

Encodings and proofs of correctness - XOR, Tseitin

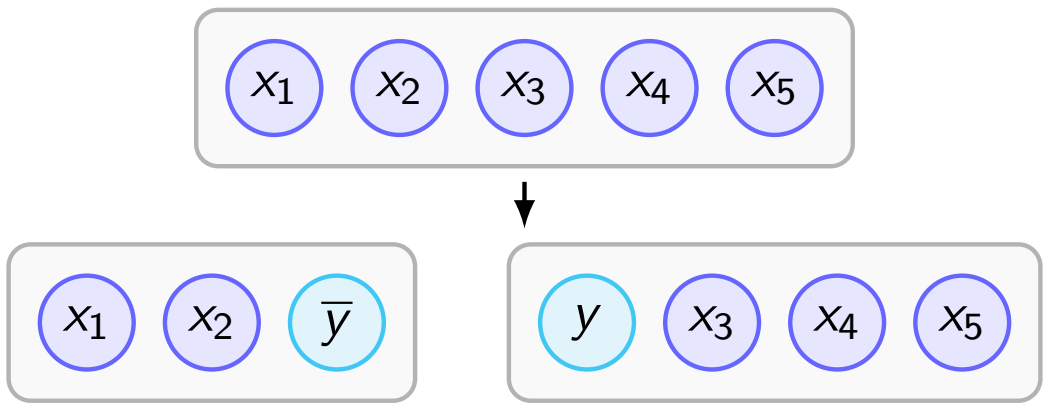
Tseitin encoding (linear)

$$\text{LINEAR}_k(x_1, \ldots, x_n) =$$
$$\begin{cases} \text{DIRECTXOR}(x_1, \ldots, x_n) & n \leq k \\ \text{DIRECTXOR}(x_1, \ldots, x_{k-1}, \overline{y}) \wedge \text{LINEAR}_k(y, x_k, \ldots, x_n) & \text{o.w.} \end{cases}$$

Encodings and proofs of correctness - XOR, Tseitin

Tseitin encoding (linear)

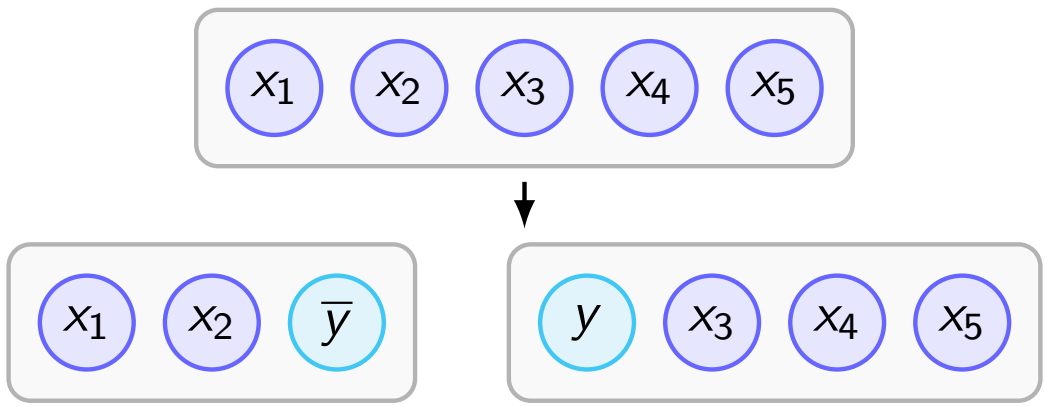
$$\text{LINEAR}_k(x_1, \ldots, x_n) =$$
$$\begin{cases} \text{DIRECTXOR}(x_1, \ldots, x_n) & n \leq k \\ \text{DIRECTXOR}(x_1, \ldots, x_{k-1}, \overline{y}) \wedge \text{LINEAR}_k(y, x_k, \ldots, x_n) & \text{o.w.} \end{cases}$$



Encodings and proofs of correctness - XOR, Tseitin

Tseitin encoding (linear)

$$\text{LINEAR}_k(x_1, \ldots, x_n) =$$
$$\begin{cases} \text{DIRECTXOR}(x_1, \ldots, x_n) & n \leq k \\ \text{DIRECTXOR}(x_1, \ldots, x_{k-1}, \overline{y}) \wedge \text{LINEAR}_k(y, x_k, \ldots, x_n) & \text{o.w.} \end{cases}$$

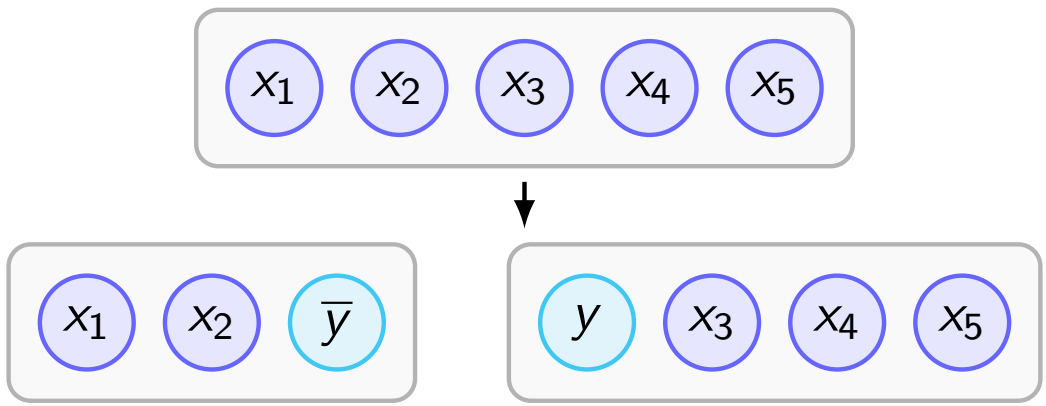


Fixed $k \geq 3$ is cutting number.

Encodings and proofs of correctness - XOR, Tseitin

Tseitin encoding (linear)

$$\text{LINEAR}_k(x_1, \ldots, x_n) =$$
$$\begin{cases} \text{DIRECTXOR}(x_1, \ldots, x_n) & n \leq k \\ \text{DIRECTXOR}(x_1, \ldots, x_{k-1}, \overline{y}) \wedge \text{LINEAR}_k(y, x_k, \ldots, x_n) & \text{o.w.} \end{cases}$$



Fixed $k \geq 3$ is cutting number.

Solver performance impacted by cutting number and linear vs. pooled.

Encodings and proofs of correctness - XOR, Tseitin

$$\text{LINEAR}_3(x_1, \ldots, x_6) =$$
$$D(x_1, x_2, \overline{y}_1) \wedge D(y_1, x_3, \overline{y}_2) \wedge D(y_2, x_4, \overline{y}_3) \wedge D(y_3, x_5, x_6)$$

$$\text{POOLED}_3(x_1, \ldots, x_6) =$$
$$D(x_1, x_2, \overline{y}_1) \wedge D(x_3, x_4, \overline{y}_2) \wedge D(x_5, x_6, \overline{y}_3) \wedge D(y_1, y_2, y_3)$$

Encodings and proofs of correctness - XOR, Tseitin

$$\text{LINEAR}_3(x_1, \ldots, x_6) =$$
$$D(x_1, x_2, \overline{y}_1) \wedge D(y_1, x_3, \overline{y}_2) \wedge D(y_2, x_4, \overline{y}_3) \wedge D(y_3, x_5, x_6)$$

$$\text{POOLED}_3(x_1, \ldots, x_6) =$$
$$D(x_1, x_2, \overline{y}_1) \wedge D(x_3, x_4, \overline{y}_2) \wedge D(x_5, x_6, \overline{y}_3) \wedge D(y_1, y_2, y_3)$$

Linear requires $O(n)$ updates to the y_i, pooled only $O(\log n)$.

Encodings and proofs of correctness - XOR, Tseitin

```
def linear_xor {k : nat} (hk : k ≥ 3) :
  list (literal V) → gensym V → cnf V
| l g := ite (length l ≤ k) (direct_xor l)
    direct_xor (l.take (k - 1) ++ [Neg g.fresh.1]) ++
    linear_xor (Pos g.fresh.1 :: l.drop (k - 1)) g.fresh.2)
```

Encodings and proofs of correctness - XOR, Tseitin

```
def linear_xor {k : nat} (hk : k ≥ 3) :
  list (literal V) → gensym V → cnf V
| l g := ite (length l ≤ k) (direct_xor l)
    direct_xor (l.take (k - 1) ++ [Neg g.fresh.1]) ++
    linear_xor (Pos g.fresh.1 :: l.drop (k - 1)) g.fresh.2)
```

Encodings and proofs of correctness - XOR, Tseitin

```
def linear_xor {k : nat} (hk : k ≥ 3) :
  list (literal V) → gensym V → cnf V
| l g := ite (length l ≤ k) (direct_xor l)
    direct_xor (l.take (k - 1) ++ [Neg g.fresh.1]) ++
    linear_xor (Pos g.fresh.1 :: l.drop (k - 1)) g.fresh.2)
```

Encodings and proofs of correctness - XOR, Tseitin

```
def linear_xor {k : nat} (hk : k ≥ 3) :
  list (literal V) → gensym V → cnf V
| l g := ite (length l ≤ k) (direct_xor l)
    direct_xor (l.take (k - 1) ++ [Neg g.fresh.1]) ++
    linear_xor (Pos g.fresh.1 :: l.drop (k - 1)) g.fresh.2)
```

Note the updated `gensym` object.

Encodings and proofs of correctness - XOR, Tseitin

```
def linear_xor {k : nat} (hk : k ≥ 3) :
  list (literal V) → gensym V → cnf V
| l g := ite (length l ≤ k) (direct_xor l)
    direct_xor (l.take (k - 1) ++ [Neg g.fresh.1]) ++
    linear_xor (Pos g.fresh.1 :: l.drop (k - 1)) g.fresh.2)
```

Note the updated `gensym` object.

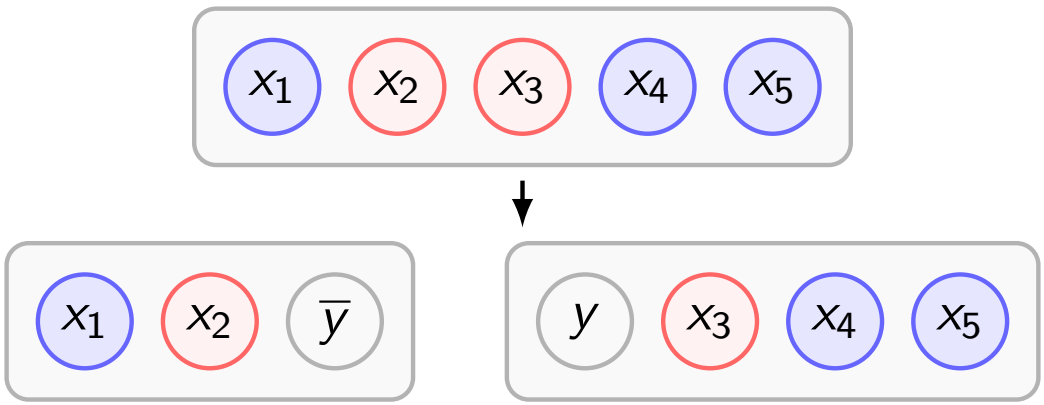
Challenge: generating fresh variables - use `fresh`.

Encodings and proofs of correctness - XOR, Tseitin

Theorem: The Tseitin transformation encodes XOR.

Proof.

By strong induction on n.



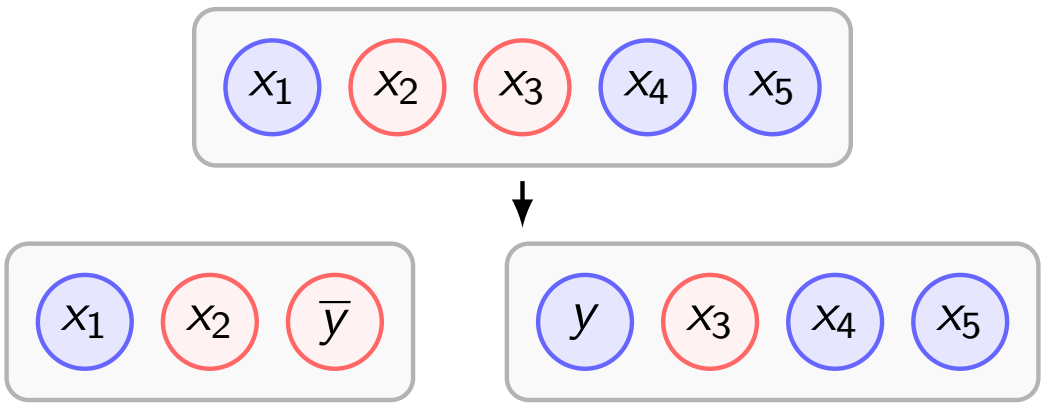
□

Encodings and proofs of correctness - XOR, Tseitin

Theorem: The Tseitin transformation encodes XOR.

Proof.

By strong induction on n.



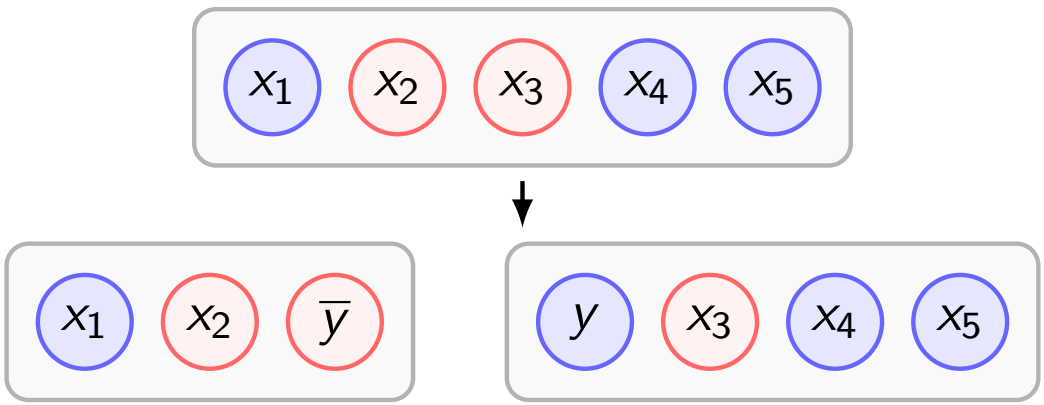
□

Encodings and proofs of correctness - XOR, Tseitin

Theorem: The Tseitin transformation encodes XOR.

Proof.

By strong induction on n.



□

Takeaway: recursive definitions require inductive proofs.

`fresh` used to generate variables, updated `gensym` passed to recursive call.

Encodings and proofs of correctness - AMO

Definition

$$\text{AMO}(x_1, \ldots, x_n) := \sum_{i=1}^{n} x_i \leq 1$$

Encodings and proofs of correctness - AMO

Definition

$$\text{AMO}(x_1, \ldots, x_n) := \sum_{i=1}^{n} x_i \leq 1$$

Part of general class of cardinality constraints.

$$\text{AMK}_k(x_1, \ldots, x_n) := \sum_{i=1}^{n} x_i \leq k.$$

Encodings and proofs of correctness - AMO, DIRECTAMO

$$\text{DIRECTAMO}(x_1, \ldots, x_n) = \bigwedge_{1 \leq i < j \leq n} (\overline{x}_i \vee \overline{x}_j)$$

Encodings and proofs of correctness - AMO, DIRECTAMO

$$\text{DIRECTAMO}(x_1, \ldots, x_n) = \bigwedge_{1 \leq i < j \leq n} (\overline{x}_i \vee \overline{x}_j)$$

Generates $\binom{n}{2} \in O(n^2)$ clauses.

Encodings and proofs of correctness - AMO, DIRECTAMO

```
def direct_amo : list (literal V) → cnf V
| []       := []
| (l::ls) := (ls.map (λ m, [l.flip, m.flip])) ++ (direct_amo ls)
```

Encodings and proofs of correctness - AMO, DIRECTAMO

```
def direct_amo : list (literal V) → cnf V
| []      := []
| (l::ls) := (ls.map (λ m, [l.flip, m.flip])) ++ (direct_amo ls)
```

Encodings and proofs of correctness - AMO, DIRECTAMO

```
def direct_amo : list (literal V) → cnf V
| []      := []
| (l::ls) := (ls.map (λ m, [l.flip, m.flip])) ++ (direct_amo ls)
```

Recursive in nature, but global definition possible too

Encodings and proofs of correctness - AMO, DIRECTAMO

```
def direct_amo : list (literal V) → cnf V
| []      := []
| (l::ls) := (ls.map (λ m, [l.flip, m.flip])) ++ (direct_amo ls)
```

Recursive in nature, but global definition possible too

Correctness proof analogous to DIRECTXOR.

Developed `distinct` object to relate to semantics - crucial for proof.

Encodings and proofs of correctness - AMO, SINZAMO

$$\text{SINZAMO}(x_1, \ldots, x_n) =$$
$$\left(\bigwedge_{i=1}^{n-1}(\overline{x}_i \vee s_i)\right) \wedge \left(\bigwedge_{i=1}^{n-1}(\overline{s}_i \vee \overline{x}_{i+1})\right) \wedge \left(\bigwedge_{i=1}^{n-2}(\overline{s}_i \vee s_{i+1})\right)$$

Encodings and proofs of correctness - AMO, SINZAMO

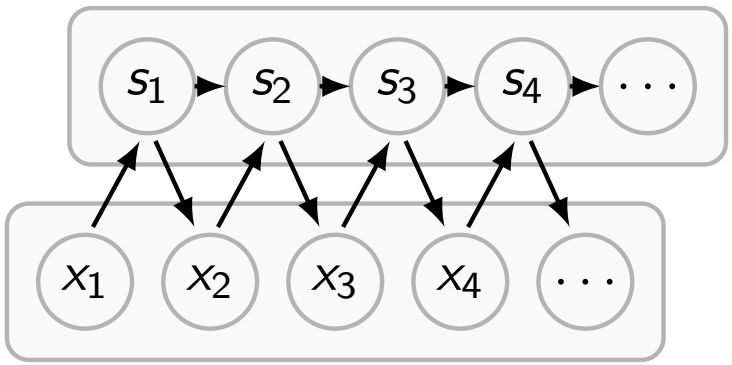
$$\text{SINZAMO}(x_1, \ldots, x_n) =$$
$$\left(\bigwedge_{i=1}^{n-1}(\overline{x}_i \vee s_i)\right) \wedge \left(\bigwedge_{i=1}^{n-1}(\overline{s}_i \vee \overline{x}_{i+1})\right) \wedge \left(\bigwedge_{i=1}^{n-2}(\overline{s}_i \vee s_{i+1})\right)$$

Generates $3n - 4$ clauses, requires $n - 2$ fresh variables.

Encodings and proofs of correctness - AMO, SINZAMO

$$\textsc{Sinzamo}(x_1, \ldots, x_n) =$$
$$\left(\bigwedge_{i=1}^{n-1}(\overline{x}_i \vee s_i)\right) \wedge \left(\bigwedge_{i=1}^{n-1}(\overline{s}_i \vee \overline{x}_{i+1})\right) \wedge \left(\bigwedge_{i=1}^{n-2}(\overline{s}_i \vee s_{i+1})\right)$$

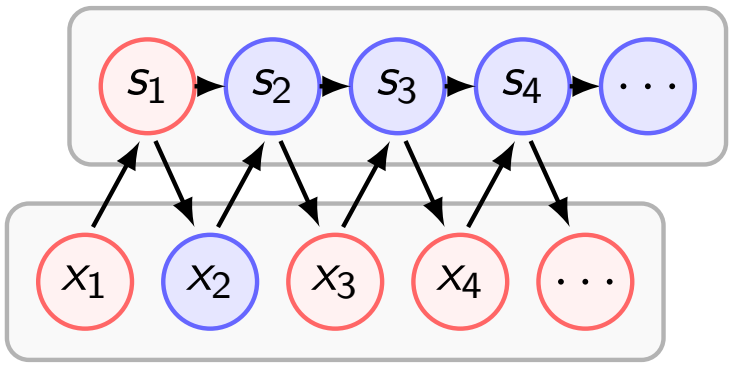
Generates $3n - 4$ clauses, requires $n - 2$ fresh variables.



Encodings and proofs of correctness - AMO, SINZAMO

$$\textsc{Sinzamo}(x_1, \ldots, x_n) =$$
$$\left(\bigwedge_{i=1}^{n-1}(\overline{x}_i \vee s_i)\right) \wedge \left(\bigwedge_{i=1}^{n-1}(\overline{s}_i \vee \overline{x}_{i+1})\right) \wedge \left(\bigwedge_{i=1}^{n-2}(\overline{s}_i \vee s_{i+1})\right)$$

Generates $3n - 4$ clauses, requires $n - 2$ fresh variables.



Encodings and proofs of correctness - AMO, SINZAMO

```
def xi_to_si (g : gensym V) (n i : nat) (lit : literal V) :=
  ite (i < n - 1) [[lit.flip, Pos (g.nth i)]] []

def si_to_next_si (g : gensym V) (n i : nat) :=
  ite (i < n - 2) [[Neg (g.nth i), Pos (g.nth (i + 1))]] []

def si_to_next_xi (g : gensym V) (n i : nat) (lit : literal V) :=
  ite (0 < i) [[Neg (g.nth (i - 1)), lit]] []

def sinz (l : list (literal V)) (g : gensym V) :=
ite (hl : length l < 2) []
  join (map_with_index (λ (idx : nat) (lit : literal V),
    xi_to_si g (length l) idx lit ++
    si_to_next_si g (length l) idx ++
    si_to_next_xi g (length l) idx lit) l)
```

Encodings and proofs of correctness - AMO, SINZAMO

```
def xi_to_si (g : gensym V) (n i : nat) (l : literal V) :=
  ite (i < n - 1) [[l.flip, Pos (g.nth i)]] []

def si_to_next_si (g : gensym V) (n i : nat) :=
  ite (i < n - 2) [[Neg (g.nth i), Pos (g.nth (i + 1))]] []

def si_to_next_xi (g : gensym V) (n i : nat) (lit : literal V) :=
  ite (0 < i) [[Neg (g.nth (i - 1)), lit]] []

def sinz (l : list (literal V)) (g : gensym V) :=
ite (hl : length l < 2) []
  join (map_with_index (λ (idx : nat) (lit : literal V),
    xi_to_si g (length l) idx lit ++
    si_to_next_si g (length l) idx ++
    si_to_next_xi g (length l) idx lit) l)
```

Encodings and proofs of correctness - AMO, SINZAMO

```
def xi_to_si (g : gensym V) (n i : nat) (lit : literal V) :=
  ite (i < n - 1) [[lit.flip, Pos (g.nth i)]] []

def si_to_next_si (g : gensym V) (n i : nat) :=
  ite (i < n - 2) [[Neg (g.nth i), Pos (g.nth (i + 1))]] []

def si_to_next_xi (g : gensym V) (n i : nat) (lit : literal V) :=
  ite (0 < i) [[Neg (g.nth (i - 1)), lit]] []

def sinz (l : list (literal V)) (g : gensym V) :=
ite (hl : length l < 2) []
  join (map_with_index (λ (idx : nat) (lit : literal V),
    xi_to_si g (length l) idx lit ++
    si_to_next_si g (length l) idx ++
    si_to_next_xi g (length l) idx lit) l)
```

Encodings and proofs of correctness - AMO, SINZAMO

```
def xi_to_si (g : gensym V) (n i : nat) (lit : literal V) :=
  ite (i < n - 1) [[lit.flip, Pos (g.nth i)]] []

def si_to_next_si (g : gensym V) (n i : nat) :=
  ite (i < n - 2) [[Neg (g.nth i), Pos (g.nth (i + 1))]] []

def si_to_next_xi (g : gensym V) (n i : nat) (lit : literal V) :=
  ite (0 < i) [[Neg (g.nth (i - 1)), lit]] []

def sinz (l : list (literal V)) (g : gensym V) :=
ite (hl : length l < 2) []
  join (map_with_index (λ (idx : nat) (lit : literal V),
    xi_to_si g (length l) idx lit ++
    si_to_next_si g (length l) idx ++
    si_to_next_xi g (length l) idx lit) l)
```

Encodings and proofs of correctness - AMO, SINZAMO

```
def xi_to_si (g : gensym V) (n i : nat) (lit : literal V) :=
  ite (i < n - 1) [[lit.flip, Pos (g.nth i)]] []

def si_to_next_si (g : gensym V) (n i : nat) :=
  ite (i < n - 2) [[Neg (g.nth i), Pos (g.nth (i + 1))]] []

def si_to_next_xi (g : gensym V) (n i : nat) (lit : literal V) :=
  ite (0 < i) [[Neg (g.nth (i - 1)), lit]] []

def sinz (l : list (literal V)) (g : gensym V) :=
ite (hl : length l < 2) []
  join (map_with_index (λ (idx : nat) (lit : literal V),
    xi_to_si g (length l) idx lit ++
    si_to_next_si g (length l) idx ++
    si_to_next_xi g (length l) idx lit) l)
```

Encodings and proofs of correctness - AMO, SINZAMO

```
def xi_to_si (g : gensym V) (n i : nat) (lit : literal V) :=
  ite (i < n - 1) [[lit.flip, Pos (g.nth i)]] []

def si_to_next_si (g : gensym V) (n i : nat) :=
  ite (i < n - 2) [[Neg (g.nth i), Pos (g.nth (i + 1))]] []

def si_to_next_xi (g : gensym V) (n i : nat) (lit : literal V) :=
  ite (0 < i) [[Neg (g.nth (i - 1)), lit]] []

def sinz (l : list (literal V)) (g : gensym V) :=
ite (hl : length l < 2) []
  join (map_with_index (λ (idx : nat) (lit : literal V),
    xi_to_si g (length l) idx lit ++
    si_to_next_si g (length l) idx ++
    si_to_next_xi g (length l) idx lit) l)
```

Encodings and proofs of correctness - AMO, SINZAMO

```
def xi_to_si (g : gensym V) (n i : nat) (lit : literal V) :=
  ite (i < n - 1) [[lit.flip, Pos (g.nth i)]] []

def si_to_next_si (g : gensym V) (n i : nat) :=
  ite (i < n - 2) [[Neg (g.nth i), Pos (g.nth (i + 1))]] []

def si_to_next_xi (g : gensym V) (n i : nat) (lit : literal V) :=
  ite (0 < i) [[Neg (g.nth (i - 1)), lit]] []

def sinz (l : list (literal V)) (g : gensym V) :=
ite (hl : length l < 2) []
  join (map_with_index (λ (idx : nat) (lit : literal V),
    xi_to_si g (length l) idx lit ++
    si_to_next_si g (length l) idx ++
    si_to_next_xi g (length l) idx lit) l)
```

Encodings and proofs of correctness - AMO, SINZAMO

```
def xi_to_si (g : gensym V) (n i : nat) (lit : literal V) :=
  ite (i < n - 1) [[lit.flip, Pos (g.nth i)]] []

def si_to_next_si (g : gensym V) (n i : nat) :=
  ite (i < n - 2) [[Neg (g.nth i), Pos (g.nth (i + 1))]] []

def si_to_next_xi (g : gensym V) (n i : nat) (lit : literal V) :=
  ite (0 < i) [[Neg (g.nth (i - 1)), lit]] []

def sinz (l : list (literal V)) (g : gensym V) :=
ite (hl : length l < 2) []
  join (map_with_index (λ (idx : nat) (lit : literal V),
    xi_to_si g (length l) idx lit ++
    si_to_next_si g (length l) idx ++
    si_to_next_xi g (length l) idx lit) l)
```

Non-recursive definition - better for capturing global behavior.

Use `nth` instead of `fresh` to generate variables.

Encodings and proofs of correctness - AMO, SINZAMO

Theorem: SINZAMO encodes AMO

Proof.

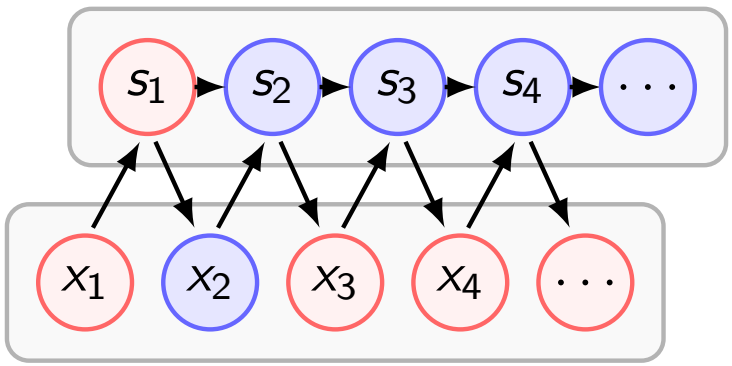
Provide explicit truth assignment. Give semantic meaning to each fresh variable.

Encodings and proofs of correctness - AMO, SINZAMO

Theorem: SINZAMO encodes AMO

Proof.

Provide explicit truth assignment. Give semantic meaning to each fresh variable.

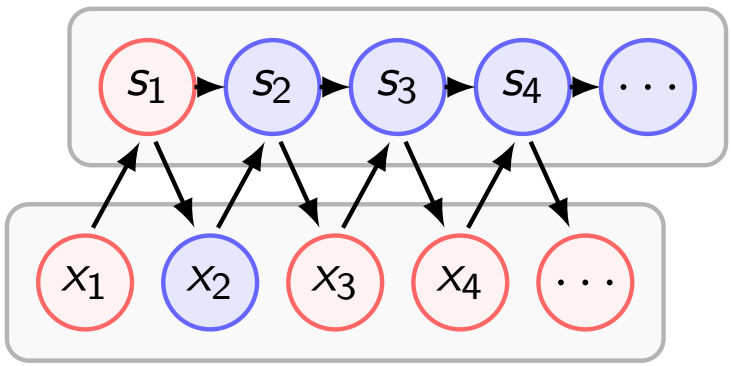


Encodings and proofs of correctness - AMO, SINZAMO

Theorem: SINZAMO encodes AMO

Proof.

Provide explicit truth assignment. Give semantic meaning to each fresh variable.



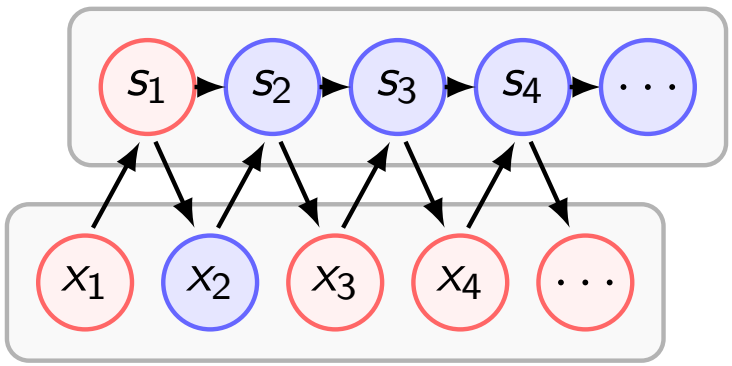
For reverse direction, take σ and show this explicit assignment works. $\square$

Encodings and proofs of correctness - AMO, SINZAMO

Theorem: SINZAMO encodes AMO

Proof.

Provide explicit truth assignment. Give semantic meaning to each fresh variable.



For reverse direction, take σ and show this explicit assignment works. □

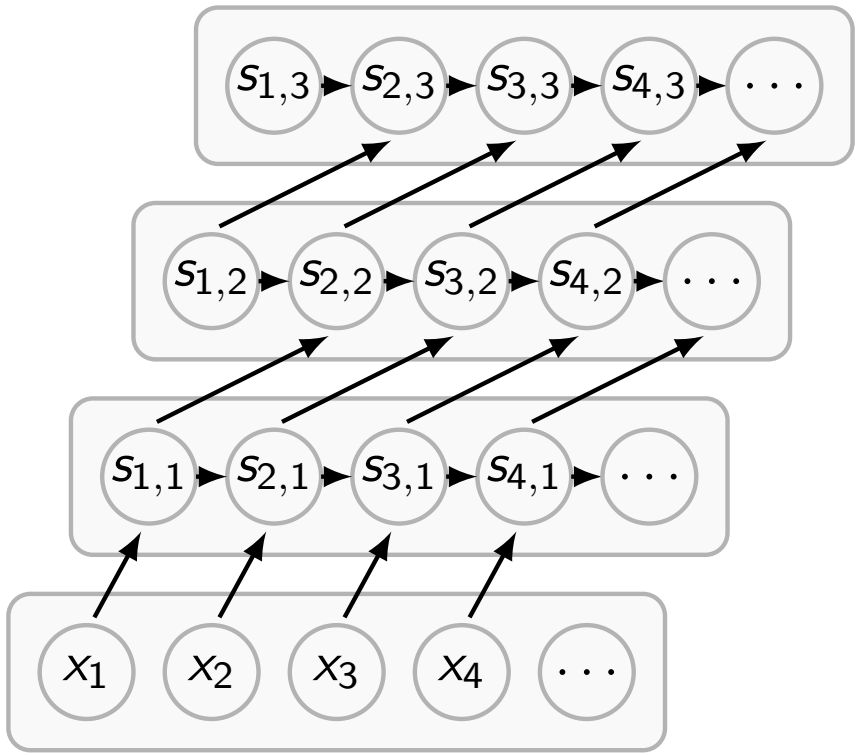
Takeaway: For encoding with global behavior, tie the introduced variables to a semantic definition, and hard-code in a satisfying assignment

Encodings and proofs of correctness - AMK, Sinz

$$\textsc{SinzAMK}(x_1, \ldots, x_n) = \left(\bigwedge_{i=1}^{n}(\overline{x}_i \vee s_{i,1})\right) \wedge \left(\bigwedge_{i=1}^{n-1}\bigwedge_{j=1}^{k+1}(\overline{s}_{i,j} \vee s_{i+1,j})\right) \wedge \left(\bigwedge_{i=1}^{n-1}\bigwedge_{j=1}^{k}(\overline{x}_{i+1} \vee \overline{s}_{i,j} \vee s_{i+1,j+1})\right) \wedge \overline{s}_{n,k+1}$$

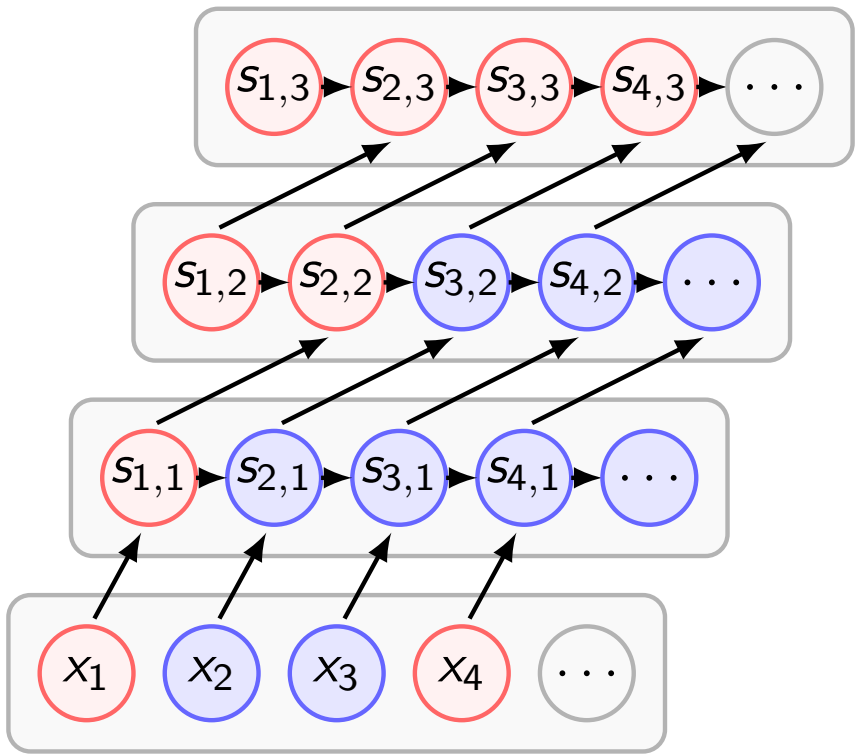
Encodings and proofs of correctness - AMK, Sinz

$$\textsc{SinzAMK}(x_1, \ldots, x_n) = \left(\bigwedge_{i=1}^{n}(\overline{x}_i \vee s_{i,1})\right) \wedge \left(\bigwedge_{i=1}^{n-1}\bigwedge_{j=1}^{k+1}(\overline{s}_{i,j} \vee s_{i+1,j})\right) \wedge \left(\bigwedge_{i=1}^{n-1}\bigwedge_{j=1}^{k}(\overline{x}_{i+1} \vee \overline{s}_{i,j} \vee s_{i+1,j+1})\right) \wedge \overline{s}_{n,k+1}$$



Encodings and proofs of correctness - AMK, Sinz

$$\textsc{SinzAMK}(x_1, \ldots, x_n) = \left(\bigwedge_{i=1}^{n}(\overline{x}_i \vee s_{i,1})\right) \wedge \left(\bigwedge_{i=1}^{n-1}\bigwedge_{j=1}^{k+1}(\overline{s}_{i,j} \vee s_{i+1,j})\right) \wedge \left(\bigwedge_{i=1}^{n-1}\bigwedge_{j=1}^{k}(\overline{x}_{i+1} \vee \overline{s}_{i,j} \vee s_{i+1,j+1})\right) \wedge \overline{s}_{n,k+1}$$



Encodings and proofs of correctness - AMK, Sinz

```
variables (l : list (literal V)) (g : gensym V) (r c : nat)

def S := g.nth ((r * length l) + c)

def inc_prop (hc : c < length l) := ite (c > 0 ∧ r > 0)
  [[(nth_le l c hc).flip,
     Neg (S g l (r - 1) (c - 1)),
     Pos (S g l r c) ]] []

def sinz_amk (k : nat) := ite (l = []) []
join ((range (k + 1)).map (λ i, join
  ((range (length l)).map (λ j,
      signal_row_prop g l i j ++
      first_prop g l i j ++
      inc_prop g l i j ++
      neg_unit g l k i j )) ))
```

Encodings and proofs of correctness - AMK, Sinz

```
variables (l : list (literal V)) (g : gensym V) (r c : nat)

def S := g.nth ((r * length l) + c)

def inc_prop (hc : c < length l) := ite (c > 0 ∧ r > 0)
  [[(nth_le l c hc).flip,
     Neg (S g l (r - 1) (c - 1)),
     Pos (S g l r c) ]] []

def sinz_amk (k : nat) := ite (l = []) []
join ((range (k + 1)).map (λ i, join
  ((range (length l)).map (λ j,
      signal_row_prop g l i j ++
      first_prop g l i j ++
      inc_prop g l i j ++
      neg_unit g l k i j )) ))
```

Encodings and proofs of correctness - AMK, Sinz

```
variables (l : list (literal V)) (g : gensym V) (r c : nat)

def S := g.nth ((r * length l) + c)

def inc_prop (hc : c < length l) := ite (c > 0 ∧ r > 0)
  [[(nth_le l c hc).flip,
     Neg (S g l (r - 1) (c - 1)),
     Pos (S g l r c) ]] []

def sinz_amk (k : nat) := ite (l = []) []
join ((range (k + 1)).map (λ i, join
  ((range (length l)).map (λ j,
      signal_row_prop g l i j ++
      first_prop g l i j ++
      inc_prop g l i j ++
      neg_unit g l k i j )) ))
```

Encodings and proofs of correctness - AMK, Sinz

```
variables (l : list (literal V)) (g : gensym V) (r c : nat)

def S := g.nth ((r * length l) + c)

def inc_prop (hc : c < length l) := ite (c > 0 ∧ r > 0)
  [[(nth_le l c hc).flip,
     Neg (S g l (r - 1) (c - 1)),
     Pos (S g l r c) ]] []

def sinz_amk (k : nat) := ite (l = []) []
join ((range (k + 1)).map (λ i, join
  ((range (length l)).map (λ j,
      signal_row_prop g l i j ++
      first_prop g l i j ++
      inc_prop g l i j ++
      neg_unit g l k i j )) ))
```

Encodings and proofs of correctness - AMK, Sinz

```
variables (l : list (literal V)) (g : gensym V) (r c : nat)

def S := g.nth ((r * length l) + c)

def inc_prop (hc : c < length l) := ite (c > 0 ∧ r > 0)
  [[(nth_le l c hc).flip,
     Neg (S g l (r - 1) (c - 1)),
     Pos (S g l r c) ]] []

def sinz_amk (k : nat) := ite (l = []) []
join ((range (k + 1)).map (λ i, join
  ((range (length l)).map (λ j,
      signal_row_prop g l i j ++
      first_prop g l i j ++
      inc_prop g l i j ++
      neg_unit g l k i j )) ))
```

```
variables (l : list (literal V)) (g : gensym V) (r c : nat)

def S := g.nth ((r * length l) + c)

def inc_prop (hc : c < length l) := ite (c > 0 ∧ r > 0)
  [[(nth_le l c hc).flip,
     Neg (S g l (r - 1) (c - 1)),
     Pos (S g l r c) ]] []

def sinz_amk (k : nat) := ite (l = []) []
join ((range (k + 1)).map (λ i, join
  ((range (length l)).map (λ j,
      signal_row_prop g l i j ++
      first_prop g l i j ++
      inc_prop g l i j ++
      neg_unit g l k i j )) ))
```

Encodings and proofs of correctness - AMK, Sinz

Theorem: SinzAMK encodes AMK

Proof.

Analogous to the AMO case, but with more bookkeeping. □

Encodings and proofs of correctness - AMK, Sinz

Theorem: SinzAMK encodes AMK

Proof.

Analogous to the AMO case, but with more bookkeeping. □

Takeaway: Again, tie fresh variables to semantic meaning.

Conclusion

We prove the correctness of several common SAT encodings.

Fresh variable generation and management were challenging—`gensym` is our solution to the problems posed.

The proof methods (recursive vs. non-recursive) are general and will aid future verification efforts.

Future work: verify more encodings.