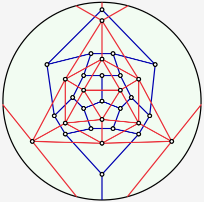
Verified encodings for SAT solvers

Cayden R. Codel
Advised by Marijn J. H. Heule and Jeremy Avigad



June 5, 2023

Repo at https://github.com/ccodel/verified-encodings

The SAT problem

SAT is an NP-hard problem in propositional logic

The SAT problem

SAT is an NP-hard problem in propositional logic

Q: Does there exist a satisfying assignment ($F \models \top$?)

The SAT problem

SAT is an NP-hard problem in propositional logic

Q: Does there exist a satisfying assignment ($F \vDash \top$?)

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

The SAT problem

SAT is an NP-hard problem in propositional logic

Q: Does there exist a satisfying assignment ($F \vDash \top$?)

$$F = (\mathbf{x_1} \vee x_2) \wedge (\overline{x}_1 \vee \mathbf{x_3}) \wedge (\overline{\mathbf{x}}_\mathbf{2} \vee \overline{x}_3)$$

$\tau = \{x_1,\ \overline{x}_2,\ x_3\}$

The SAT problem

SAT is an NP-hard problem in propositional logic

Q: Does there exist a satisfying assignment ($F \vDash \top$?)

$$F = (\mathbf{x_1} \lor x_2) \land (\overline{x}_1 \lor \mathbf{x_3}) \land (\overline{\mathbf{x}}_\mathbf{2} \lor \overline{x}_3)$$

$\tau = \{x_1,\ \overline{x}_2,\ x_3\}$

SAT solvers find a satisfying τ, or declare that none exists

The SAT problem

SAT solvers accept text input in conjunctive normal form

The SAT problem

SAT solvers accept text input in conjunctive normal form

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

```
p cnf 3 3
1 2 0
-1 3 0
-2 -3 0
```

SAT solvers at work

Hardware/software verification, optimization, SMT solvers

SAT solvers at work

Hardware/software verification, optimization, SMT solvers

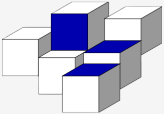
Resolve longstanding problems in mathematics:

SAT solvers at work

Hardware/software verification, optimization, SMT solvers

Resolve longstanding problems in mathematics:

Keller's Conjecture

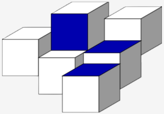


SAT solvers at work

Hardware/software verification, optimization, SMT solvers

Resolve longstanding problems in mathematics:

Keller's Conjecture



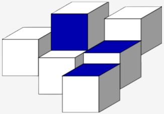
Pythagorean triples problem $a^2 + b^2 = c^2$

SAT solvers at work

Hardware/software verification, optimization, SMT solvers

Resolve longstanding problems in mathematics:

Keller's Conjecture



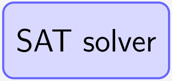
Pythagorean triples problem

$$a^2 + b^2 = c^2$$

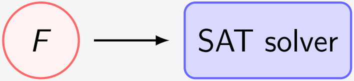
Lam's Problem

1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0
0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0
0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1

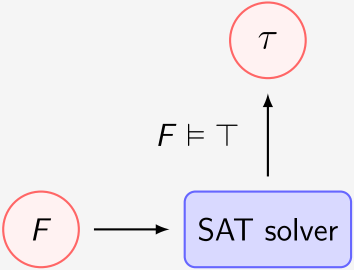
SAT toolchain



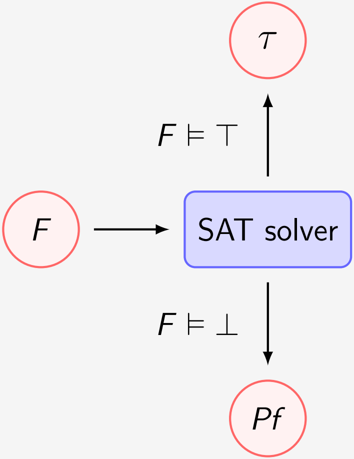
SAT toolchain



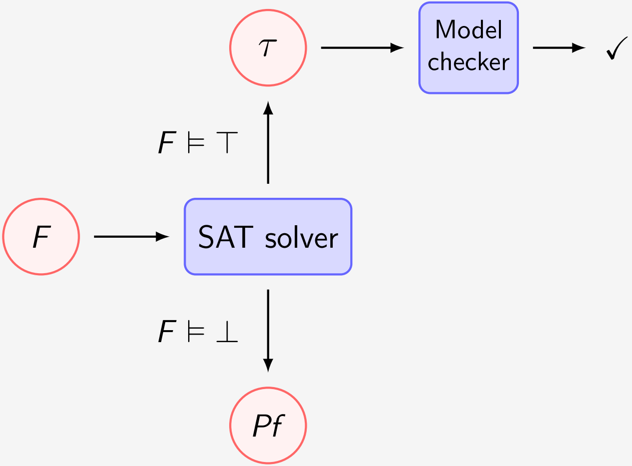
SAT toolchain



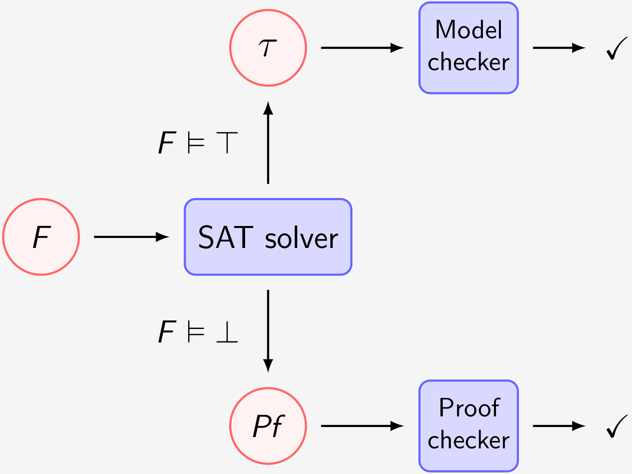
SAT toolchain



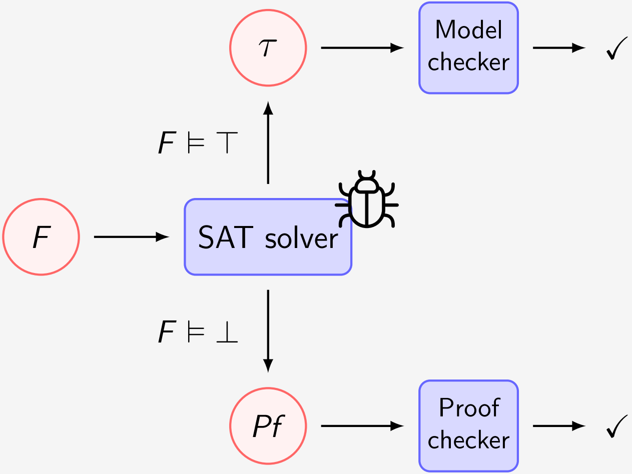
SAT toolchain



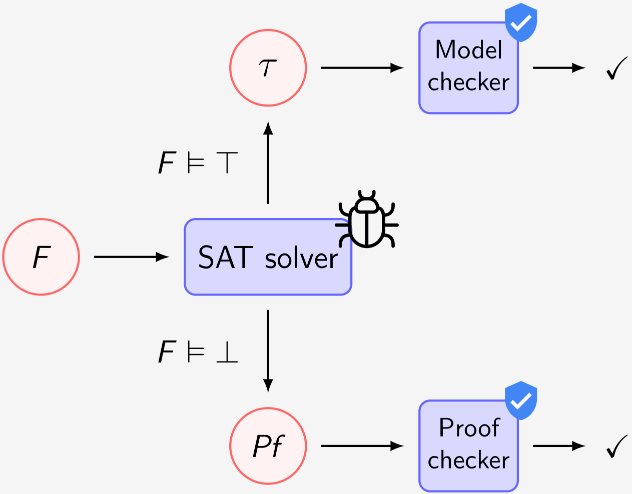
SAT toolchain



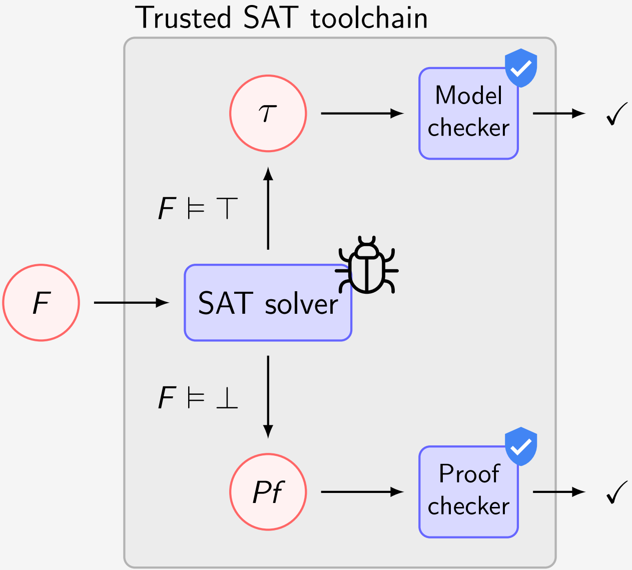
SAT toolchain



SAT toolchain

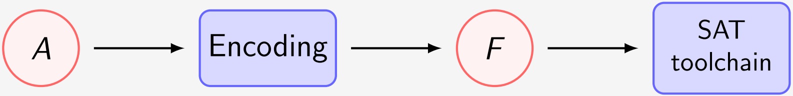


SAT toolchain

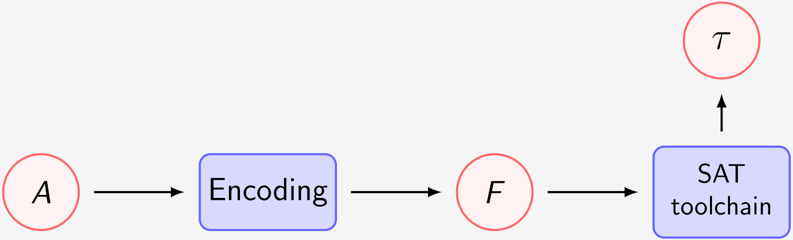


Using the SAT toolchain

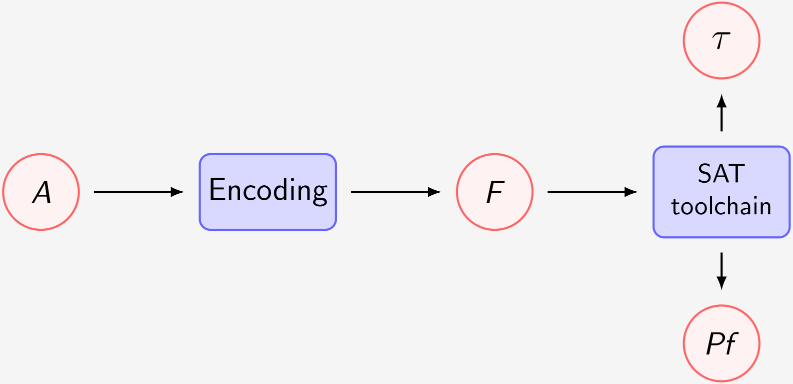
Using the SAT toolchain



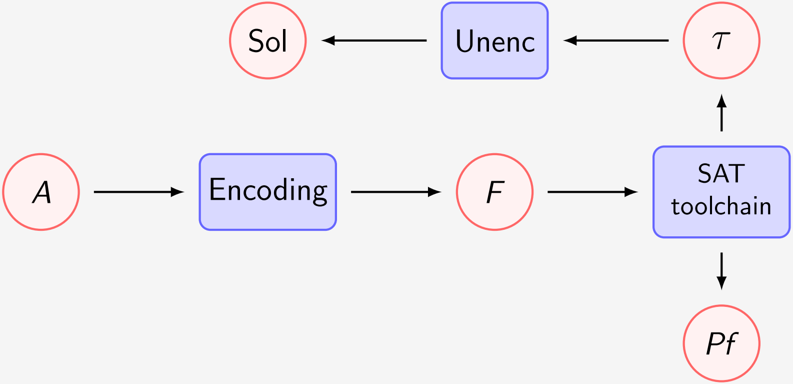
Using the SAT toolchain



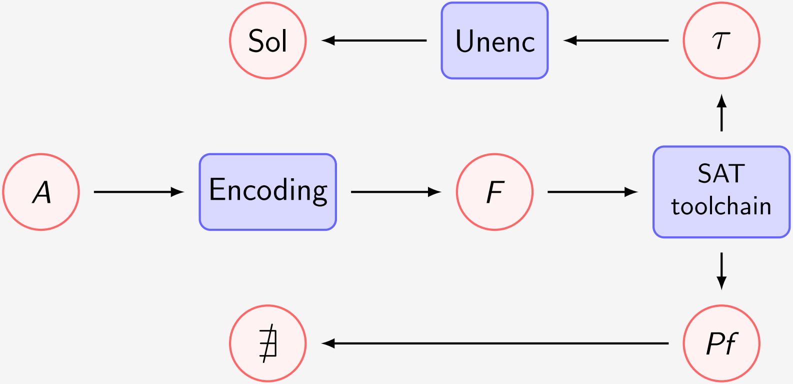
Using the SAT toolchain



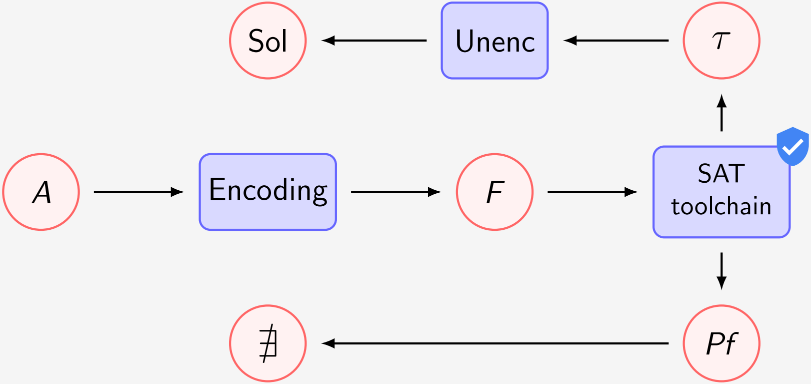
Using the SAT toolchain



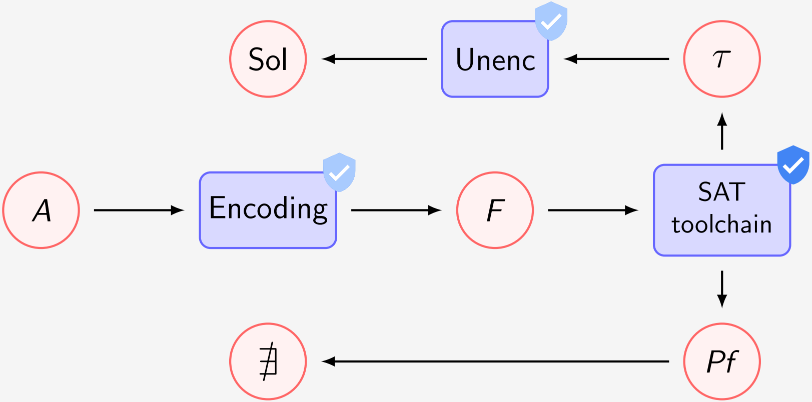
Using the SAT toolchain



Using the SAT toolchain



Using the SAT toolchain



My work: extend the trusted SAT toolchain to include encodings by using a theorem prover

The Lean theorem prover

LEAN

Lean is an interactive theorem prover based on the calculus of inductive constructions (constructive logic)

The Lean theorem prover

Lean is an interactive theorem prover based on the calculus of inductive constructions (constructive logic)

`mathlib` is the community mathematics library, with over a million lines of code

The Lean theorem prover

Lean is an interactive theorem prover based on the calculus of inductive constructions (constructive logic)

`mathlib` is the community mathematics library, with over a million lines of code

We used version 3; version 4 is under active development

The Lean theorem prover

Proofs are written in Lean declaratively or with tactics that manipulate proof state (similar to Coq, Isabelle, etc.)

The Lean theorem prover

Proofs are written in Lean declaratively or with tactics that manipulate proof state (similar to Coq, Isabelle, etc.)

```
theorem take_sublist_of_le {α : Type*} {i j : nat} : i ≤ j →
  ∀ (l : list α), l.take i <+ l.take j :=
begin
  intros hij l,
  induction l with a as ih generalizing i j,
  { rw [take_nil, take_nil] },
  { cases i,
    { rw take_zero,
      exact nil_sublist _ },
    { cases j,
      { exact absurd hij (not_le.mpr (succ_pos i)) },
      { rw [take, take],
        exact cons_sublist_cons_iff.mpr
          (ih (succ_le_succ_iff.mp hij)) } } }
end
```

The Lean theorem prover

Proofs are written in Lean declaratively or with tactics that manipulate proof state (similar to Coq, Isabelle, etc.)

```
theorem take_sublist_of_le {α : Type*} {i j : nat} : i ≤ j →
  ∀ (l : list α), l.take i <+ l.take j :=
begin
  intros hij l,
  induction l with a as ih generalizing i j,
  { rw [take_nil, take_nil] },
  { cases i,
    { rw take_zero,
      exact nil_sublist _ },
    { cases j,
      { exact absurd hij (not_le.mpr (succ_pos i)) },
      { rw [take, take],
        exact cons_sublist_cons_iff.mpr
          (ih (succ_le_succ_iff.mp hij)) } } }
end
```

The Lean theorem prover

Proofs are written in Lean declaratively or with tactics that manipulate proof state (similar to Coq, Isabelle, etc.)

```
theorem take_sublist_of_le {α : Type*} {i j : nat} : i ≤ j →
  ∀ (l : list α), l.take i <+ l.take j :=
begin
  intros hij l,
  induction l with a as ih generalizing i j,
  { rw [take_nil, take_nil] },
  { cases i,
    { rw take_zero,
      exact nil_sublist _ },
    { cases j,
      { exact absurd hij (not_le.mpr (succ_pos i)) },
      { rw [take, take],
        exact cons_sublist_cons_iff.mpr
          (ih (succ_le_succ_iff.mp hij)) } } }
end
```

The Lean theorem prover

Proofs are written in Lean declaratively or with tactics that manipulate proof state (similar to Coq, Isabelle, etc.)

```
theorem take_sublist_of_le {α : Type*} {i j : nat} : i ≤ j →
  ∀ (l : list α), l.take i <+ l.take j :=
begin
  intros hij l,
  induction l with a as ih generalizing i j,
  { rw [take_nil, take_nil] },
  { cases i,
    { rw take_zero,
      exact nil_sublist _ },
    { cases j,
      { exact absurd hij (not_le.mpr (succ_pos i)) },
      { rw [take, take],
        exact cons_sublist_cons_iff.mpr
          (ih (succ_le_succ_iff.mp hij)) } } }
end
```

The Lean theorem prover

Proofs are written in Lean declaratively or with tactics that manipulate proof state (similar to Coq, Isabelle, etc.)

```
theorem take_sublist_of_le {α : Type*} {i j : nat} : i ≤ j →
  ∀ (l : list α), l.take i <+ l.take j :=
begin
  intros hij l,
  induction l with a as ih generalizing i j,
  { rw [take_nil, take_nil] },
  { cases i,
    { rw take_zero,
      exact nil_sublist _ },
    { cases j,
      { exact absurd hij (not_le.mpr (succ_pos i)) },
      { rw [take, take],
        exact cons_sublist_cons_iff.mpr
          (ih (succ_le_succ_iff.mp hij)) } } }
end
```

Verified encodings library

Open-source on Github

Verified encodings library

Open-source on Github

Contains:

- Data structures (CNF representations, variable generation)
- Supporting lemmas and theorems
- Proofs of correctness for parity, at-most-one, at-most-k
- Support for combining encodings to form larger ones

Verified encodings library

Open-source on Github

Contains:

- Data structures (CNF representations, variable generation)
- Supporting lemmas and theorems
- Proofs of correctness for parity, at-most-one, at-most-k
- Support for combining encodings to form larger ones

Basis for future verification efforts

Library preliminaries

Goal: prove that an encoding is correct

Library preliminaries

Goal: prove that an encoding is correct

Q: What does it mean for an encoding to be correct?

Library preliminaries

F is a formula in propositional logic

C is a boolean constraint with inputs $X = x_1, \ldots, x_n$

Library preliminaries

F is a formula in propositional logic

C is a boolean constraint with inputs $X = x_1, \ldots, x_n$

F encodes C if for all truth assignments τ,

$$C(\tau(x_1), \ldots, \tau(x_n)) \leftrightarrow \exists \sigma,\ \sigma(F) = \top,$$

where σ agrees with τ on X (i.e. $\forall x \in X,\ \tau(x) = \sigma(x)$)

Library preliminaries

F is a formula in propositional logic

C is a boolean constraint with inputs $X = x_1, \ldots, x_n$

F encodes C if for all truth assignments τ,

$$C(\tau(x_1), \ldots, \tau(x_n)) \leftrightarrow \exists \sigma,\ \sigma(F) = \top,$$

where σ agrees with τ on X (i.e. $\forall x \in X,\ \tau(x) = \sigma(x)$)

An encoding function E is correct for C if the formula it produces encodes C on all inputs

Library preliminaries

In Lean, the definitions look like:

```
def encodes (C : constraint) (l : list literal) (F : cnf) :=
  ∀ (τ : assignment),
  (C.eval τ l = tt) ↔
    ∃ σ, F.eval σ = tt ∧ agree_on τ σ (vars l)
```

Library preliminaries

In Lean, the definitions look like:

```
def encodes (C : constraint) (l : list literal) (F : cnf) :=
  ∀ (τ : assignment),
  (C.eval τ l = tt) ↔
    ∃ σ, F.eval σ = tt ∧ agree_on τ σ (vars l)
```

```
def is_correct (C) (enc : enc_fn) :=
  ∀ ⦃l : list literal⦄ ⦃g : gensym⦄, disjoint l g →
  encodes C (formula (enc l g)) l
```

Library preliminaries

In Lean, the definitions look like:

```
def encodes (C : constraint) (l : list literal) (F : cnf) :=
  ∀ (τ : assignment),
  (C.eval τ l = tt) ↔
    ∃ σ, F.eval σ = tt ∧ agree_on τ σ (vars l)
```

```
def is_correct (C) (enc : enc_fn) :=
  ∀ ⦃l : list literal⦄ ⦃g : gensym⦄, disjoint l g →
  encodes C (formula (enc l g)) l
```

We prove that the encodings in our library are correct and well-behaved (generate new variables in a reasonable manner)

Case study: at-most-one

The at-most-one encoding is true iff at most one of the boolean variables is true

Case study: at-most-one

The at-most-one encoding is true iff at most one of the boolean variables is true

The naive encoding produces $O(n^2)$ clauses and enumerates all pairs of variables:

$$\mathrm{Naive}(X) = \bigwedge_{1 \le i < j \le n} (\overline{x}_i \vee \overline{x}_j)$$

Case study: at-most-one

The at-most-one encoding is true iff at most one of the boolean variables is true

The naive encoding produces $O(n^2)$ clauses and enumerates all pairs of variables:

$$\mathrm{Naive}(X) = \bigwedge_{1 \leq i < j \leq n} (\overline{x}_i \vee \overline{x}_j)$$

The Sinz encoding produces $O(n)$ clauses and needs $n - 1$ new variables:

$$\mathrm{Sinz}(X) = \bigwedge_{i=1}^{n-1} \left((\overline{x}_i \vee s_i) \wedge (\overline{s}_i \vee s_{i+1}) \wedge (\overline{s}_i \vee \overline{x}_{i+1}) \right)$$

Case study: at-most-one

The at-most-one encoding is true iff at most one of the boolean variables is true

The naive encoding produces $O(n^2)$ clauses and enumerates all pairs of variables:

$$\mathrm{Naive}(X) = \bigwedge_{1 \le i < j \le n} (\overline{x}_i \vee \overline{x}_j)$$

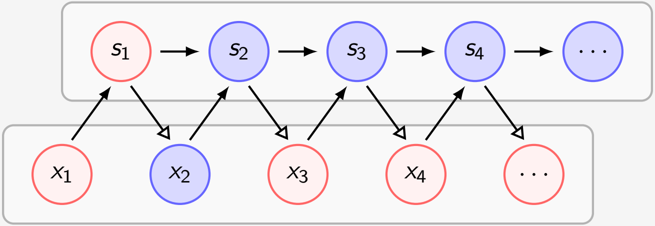
The Sinz encoding produces $O(n)$ clauses and needs $n - 1$ new variables:

$$\mathrm{Sinz}(X) = \bigwedge_{i=1}^{n-1} \left((\overline{x}_i \vee s_i) \wedge (\overline{s}_i \vee s_{i+1}) \wedge (\overline{s}_i \vee \overline{x}_{i+1}) \right)$$

The three clauses are logically equivalent to

$$(x_i \to s_i) \wedge (s_i \to s_{i+1}) \wedge (s_i \to \overline{x}_{i+1})$$

Case study: at-most-one



(Hollow arrow heads indicate negated implications)

Case study: at-most-one

Encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]          g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[l₁.flip, Pos y], [Neg y, l₂.flip]], g₁⟩

| (l₁ :: l₂ :: ls) g :=
  let ⟨y, g₁⟩ := g.fresh in
  let ⟨z, _⟩ := g₁.fresh in
  let ⟨F_rec, g₂⟩ := sinz_rec (l₂ :: ls) g₁ in
  ⟨[[l₁.flip, Pos y], [Neg y, Pos z],
      [Neg y, l₂.flip]] ++ F_rec, g₂⟩
```

Case study: at-most-one

Encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]         g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[l₁.flip, Pos y], [Neg y, l₂.flip]], g₁⟩

| (l₁ :: l₂ :: ls) g :=
  let ⟨y, g₁⟩ := g.fresh in
  let ⟨z, _⟩ := g₁.fresh in
  let ⟨F_rec, g₂⟩ := sinz_rec (l₂ :: ls) g₁ in
  ⟨[[l₁.flip, Pos y], [Neg y, Pos z],
      [Neg y, l₂.flip]] ++ F_rec, g₂⟩
```

Case study: at-most-one

Encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]          g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[l₁.flip, Pos y], [Neg y, l₂.flip]], g₁⟩

| (l₁ :: l₂ :: ls) g :=
  let ⟨y, g₁⟩ := g.fresh in
  let ⟨z, _⟩ := g₁.fresh in
  let ⟨F_rec, g₂⟩ := sinz_rec (l₂ :: ls) g₁ in
  ⟨[[l₁.flip, Pos y], [Neg y, Pos z],
      [Neg y, l₂.flip]] ++ F_rec, g₂⟩
```

Case study: at-most-one

Encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]         g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[l₁.flip, Pos y], [Neg y, l₂.flip]], g₁⟩

| (l₁ :: l₂ :: ls) g :=
  let ⟨y, g₁⟩ := g.fresh in
  let ⟨z, _⟩ := g₁.fresh in
  let ⟨F_rec, g₂⟩ := sinz_rec (l₂ :: ls) g₁ in
  ⟨[[l₁.flip, Pos y], [Neg y, Pos z],
      [Neg y, l₂.flip]] ++ F_rec, g₂⟩
```

Case study: at-most-one

Encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]          g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[l₁.flip, Pos y], [Neg y, l₂.flip]], g₁⟩

| (l₁ :: l₂ :: ls) g :=
  let ⟨y, g₁⟩ := g.fresh in
  let ⟨z, _⟩ := g₁.fresh in
  let ⟨F_rec, g₂⟩ := sinz_rec (l₂ :: ls) g₁ in
  ⟨[[l₁.flip, Pos y], [Neg y, Pos z],
      [Neg y, l₂.flip]] ++ F_rec, g₂⟩
```

Case study: at-most-one

Encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]         g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[l₁.flip, Pos y], [Neg y, l₂.flip]], g₁⟩

| (l₁ :: l₂ :: ls) g :=
  let ⟨y, g₁⟩ := g.fresh in
  let ⟨z, _⟩ := g₁.fresh in
  let ⟨F_rec, g₂⟩ := sinz_rec (l₂ :: ls) g₁ in
  ⟨[[l₁.flip, Pos y], [Neg y, Pos z],
      [Neg y, l₂.flip]] ++ F_rec, g₂⟩
```

Case study: at-most-one

Encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]          g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[l₁.flip, Pos y], [Neg y, l₂.flip]], g₁⟩

| (l₁ :: l₂ :: ls) g :=
  let ⟨y, g₁⟩ := g.fresh in
  let ⟨z, _⟩ := g₁.fresh in
  let ⟨F_rec, g₂⟩ := sinz_rec (l₂ :: ls) g₁ in
  ⟨[[l₁.flip, Pos y], [Neg y, Pos z],
      [Neg y, l₂.flip]] ++ F_rec, g₂⟩
```

Case study: at-most-one

Encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]          g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[l₁.flip, Pos y], [Neg y, l₂.flip]], g₁⟩

| (l₁ :: l₂ :: ls) g :=
  let ⟨y, g₁⟩ := g.fresh in
  let ⟨z, _⟩ := g₁.fresh in
  let ⟨F_rec, g₂⟩ := sinz_rec (l₂ :: ls) g₁ in
  ⟨[[l₁.flip, Pos y], [Neg y, Pos z],
      [Neg y, l₂.flip]] ++ F_rec, g₂⟩
```

Case study: at-most-one

Encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]          g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[l₁.flip, Pos y], [Neg y, l₂.flip]], g₁⟩

| (l₁ :: l₂ :: ls) g :=
  let ⟨y, g₁⟩ := g.fresh in
  let ⟨z, _⟩ := g₁.fresh in
  let ⟨F_rec, g₂⟩ := sinz_rec (l₂ :: ls) g₁ in
  ⟨[[l₁.flip, Pos y], [Neg y, Pos z],
      [Neg y, l₂.flip]] ++ F_rec, g₂⟩
```

Applications and future work

Combine sub-encodings to form more complex ones

Easily recover proofs of correctness

```
def append (enc₁ enc₂ : enc_fn) : enc_fn :=
  λ (l : list literal) (g : gensym),
  let (F₁, g₁) := enc₁ l g in
  let (F₂, g₂) := enc₂ l g₁ in
  (F₁ ++ F₂, g₂)
```

Applications and future work

Combine sub-encodings to form more complex ones

Easily recover proofs of correctness

```
def append (enc₁ enc₂ : enc_fn) : enc_fn :=
  λ (l : list literal) (g : gensym),
  let (f₁, g₁) := enc₁ l g in
  let (f₂, g₂) := enc₂ l g₁ in
  (f₁ ++ f₂, g₂)
```

Applications and future work

Combine sub-encodings to form more complex ones

Easily recover proofs of correctness

```
def append (enc₁ enc₂ : enc_fn) : enc_fn :=
  λ (l : list literal) (g : gensym),
  let (F₁, g₁) := enc₁ l g in
  let (F₂, g₂) := enc₂ l g₁ in
  (F₁ ++ F₂, g₂)
```

Applications and future work

Combine sub-encodings to form more complex ones

Easily recover proofs of correctness

```
def append (enc₁ enc₂ : enc_fn) : enc_fn :=
  λ (l : list literal) (g : gensym),
  let (F₁, g₁) := enc₁ l g in
  let (F₂, g₂) := enc₂ l g₁ in
  (F₁ ++ F₂, g₂)
```

Applications and future work

Combine sub-encodings to form more complex ones

Easily recover proofs of correctness

```
def append (enc₁ enc₂ : enc_fn) : enc_fn :=
  λ (l : list literal) (g : gensym),
  let (F₁, g₁) := enc₁ l g in
  let (F₂, g₂) := enc₂ l g₁ in
  (F₁ ++ F₂, g₂)
```

Applications and future work

Combine sub-encodings to form more complex ones

Easily recover proofs of correctness

```
def append (enc₁ enc₂ : enc_fn) : enc_fn :=
  λ (l : list literal) (g : gensym),
  let (F₁, g₁) := enc₁ l g in
  let (F₂, g₂) := enc₂ l g₁ in
  (F₁ ++ F₂, g₂)
```

```
theorem is_correct_append
  {c₁ c₂ : constraint} {enc₁ enc₂ : enc_fn V} :
  is_correct c₁ enc₁ → is_correct c₂ enc₂ →
  is_correct (c₁ ++ c₂) (enc₁ ++ enc₂) := ...
```

Applications and future work

Combine sub-encodings to form more complex ones

Easily recover proofs of correctness

```
def append (enc₁ enc₂ : enc_fn) : enc_fn :=
  λ (l : list literal) (g : gensym),
  let (F₁, g₁) := enc₁ l g in
  let (F₂, g₂) := enc₂ l g₁ in
  (F₁ ++ F₂, g₂)
```

```
theorem is_correct_append
  {c₁ c₂ : constraint} {enc₁ enc₂ : enc_fn V} :
  is_correct c₁ enc₁ → is_correct c₂ enc₂ →
  is_correct (c₁ ++ c₂) (enc₁ ++ enc₂) := ...
```

Already demonstrated by combining sub-encodings for Sudoku

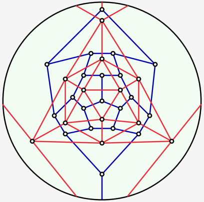
Applications and future work

- Prove more (sub-)encodings correct
- Prove the Keller reduction correct
- Write verified proof checkers for SAT proof systems

Applications and future work

- Prove more (sub-)encodings correct
- Prove the Keller reduction correct
- Write verified proof checkers for SAT proof systems

Overall, the goal is to make Lean the one-stop-shop for generating SAT queries in a trusted way



Thank you!
Any questions?