Verified encodings for SAT solvers

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

June 26 - 30, 2023

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

The problem with SAT encodings

The Lean theorem prover

Verified encodings library

Applications

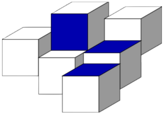
SAT solvers are great!

Hardware/software verification, optimization, SMT solvers, ...

SAT solvers are great!

Hardware/software verification, optimization, SMT solvers, ...

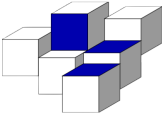
Keller's Conjecture [IJCAR'20]



SAT solvers are great!

Hardware/software verification, optimization, SMT solvers, ...

Keller's Conjecture [IJCAR'20]



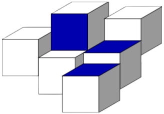
Pythagorean triples [SAT'16]

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

SAT solvers are great!

Hardware/software verification, optimization, SMT solvers, . . .

Keller's Conjecture [IJCAR'20]



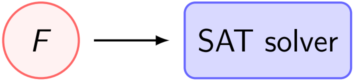
Pythagorean triples [SAT'16]

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

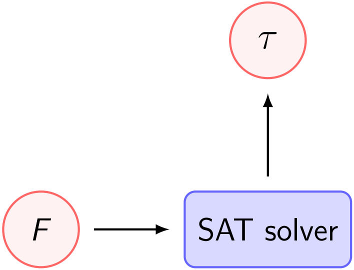
Lam's Problem [AAAI'21]

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

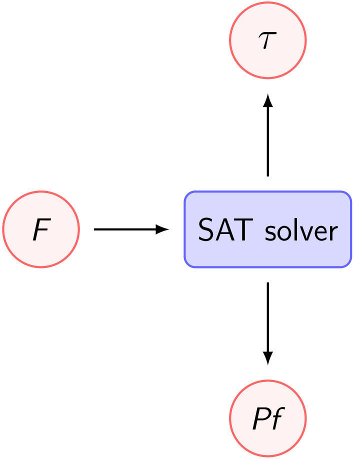
The SAT toolchain



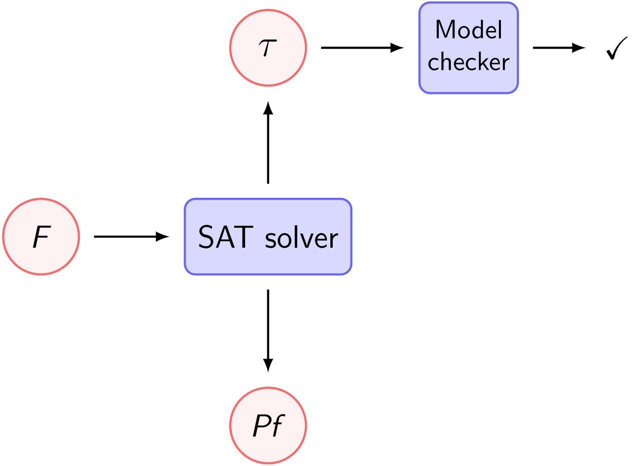
The SAT toolchain



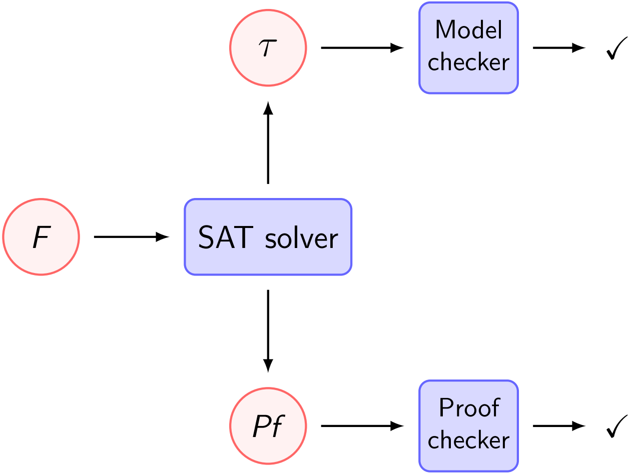
The SAT toolchain



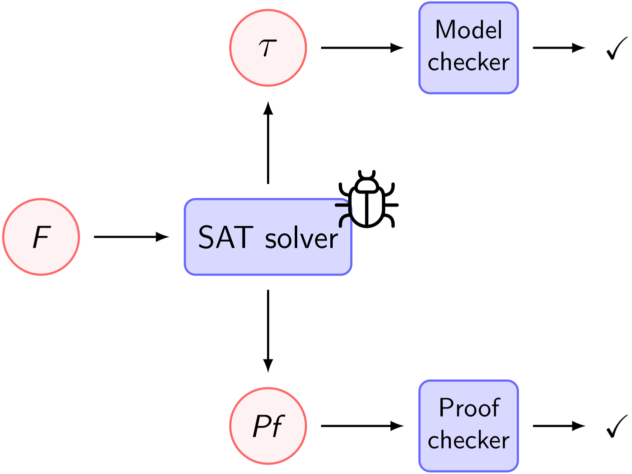
The SAT toolchain



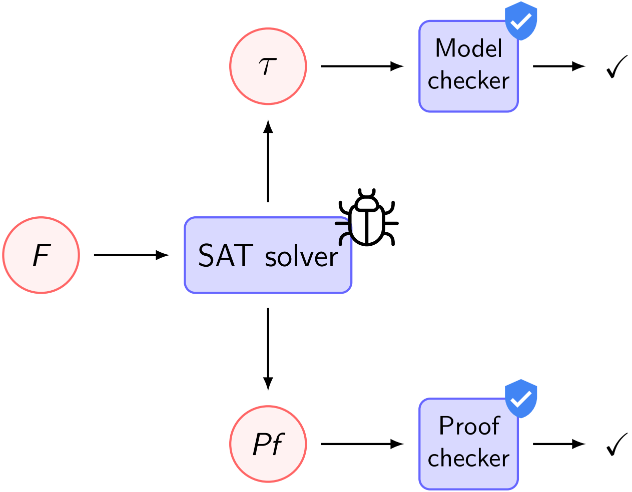
The SAT toolchain



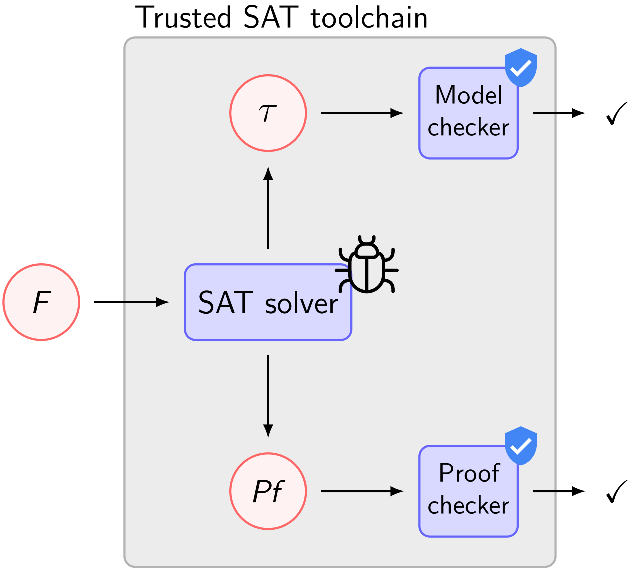
The SAT toolchain



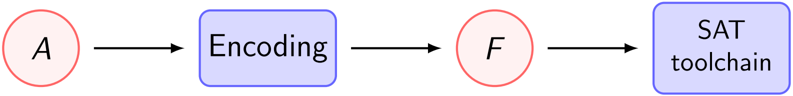
The SAT toolchain



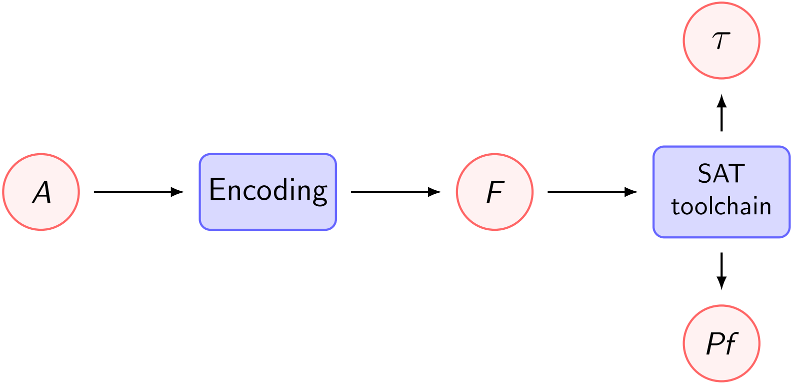
The SAT toolchain



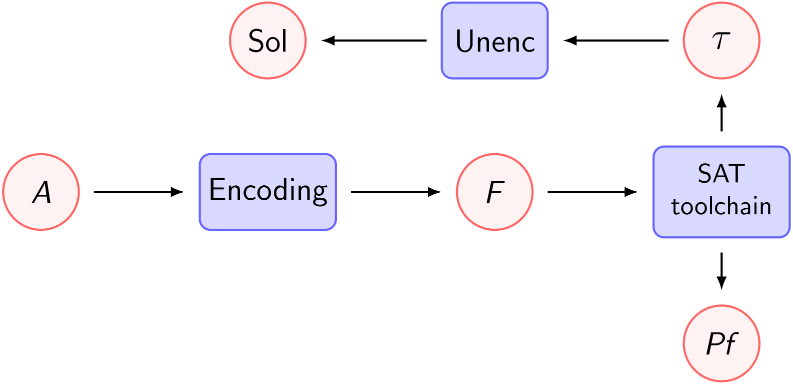
The problem with encodings



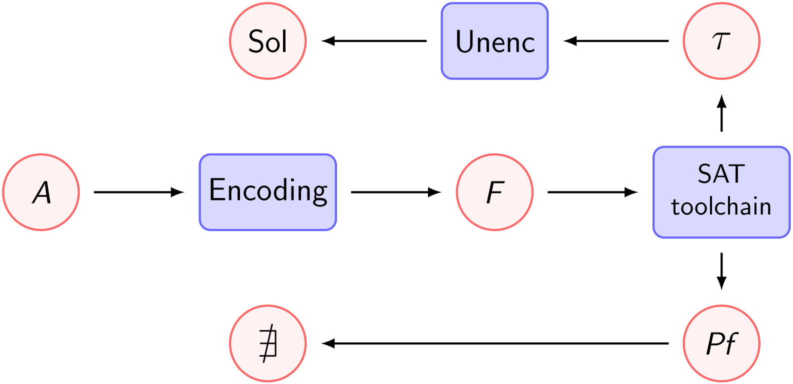
The problem with encodings



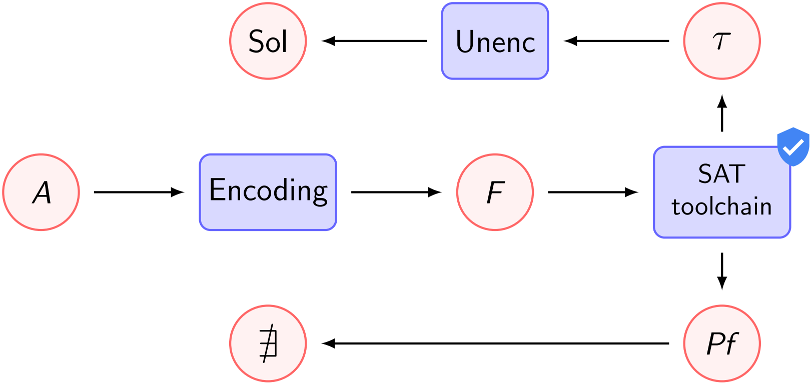
The problem with encodings



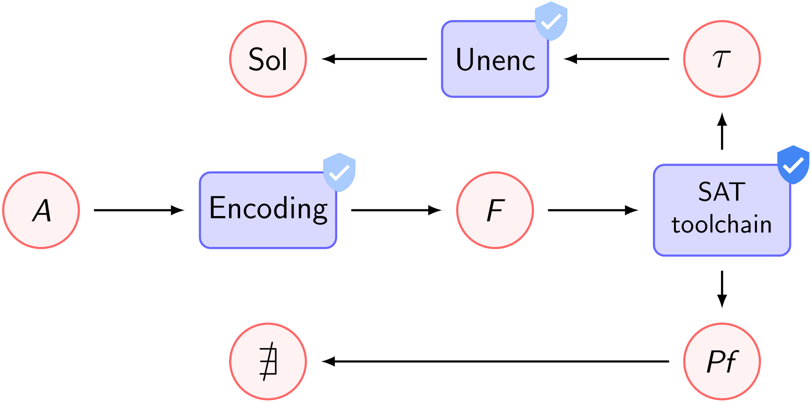
The problem with encodings



The problem with encodings



The problem with encodings



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

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

Prior work [Cruz-Filipe, Marques-Silva, Schneider-Kamp '19; Giljegård and Wennerbreck '21] verified specific encodings; our library is general

The Lean theorem prover

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 LoC, with theorems on lists, sets, natural numbers, ...

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 LoC, with theorems on lists, sets, natural numbers, ...

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

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 LoC, with theorems on lists, sets, natural numbers, ...

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

Quick demo!

Verified encodings library

The encodings library is open-source on Github

Verified encodings library

The encodings library is 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

The encodings library is 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

Goal: prove that an encoding is correct

Library preliminaries

Goal: prove that an encoding is correct

But what is a correct encoding?

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. $\forall x \in X,\, \tau(x) = \sigma(x)$)
(In other words, σ extends τ.)

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. $\forall x \in X,\, \tau(x) = \sigma(x)$)
(In other words, σ extends τ.)

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 : constraint) (enc : enc_fn) :=
  ∀ {l : list literal} {g : gensym}, disjoint l g →
  encodes C ((enc l g).formula) 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 : constraint) (enc : enc_fn) :=
  ∀ {l : list literal} {g : gensym}, disjoint l g →
  encodes C ((enc l g).formula) l
```

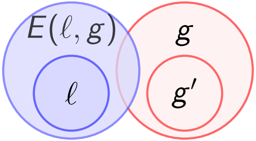
We prove that the encoding functions in our library are correct according to these definitions

Library preliminaries

Encodings must also be well-behaved (i.e. that they generate fresh variables in a reasonable way)

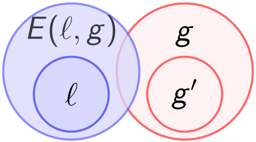
Library preliminaries

Encodings must also be well-behaved (i.e. that they generate fresh variables in a reasonable way)



Library preliminaries

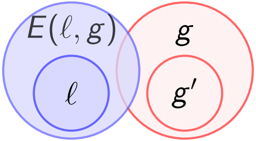
Encodings must also be well-behaved (i.e. that they generate fresh variables in a reasonable way)



$$g' \subseteq g \qquad \mathrm{vars}(E(\ell, g)) \subseteq \mathrm{vars}(\ell) \cup (g \setminus g')$$

Library preliminaries

Encodings must also be well-behaved (i.e. that they generate fresh variables in a reasonable way)



$$g' \subseteq g \qquad \mathrm{vars}(E(\ell, g)) \subseteq \mathrm{vars}(\ell) \cup (g \setminus g')$$

```
def is_wb (enc : enc_fn) :=
  ∀ {l : list literal} {g : gensym}, disjoint l g →
  (enc l g).gensym ⊆ g ∧
  vars (enc l g).formula ⊆ (vars l) ∪ (g \ (enc l g).gensym)
```

Case study: at-most-one

The Sinz at-most-one encoding produces $\sim 3n$ 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 Sinz at-most-one encoding produces $\sim 3n$ clauses and needs $n-1$ new variables:

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

The three clauses are logically equivalent to

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

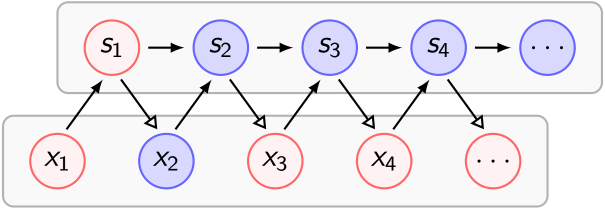
Case study: at-most-one

The Sinz at-most-one encoding produces $\sim 3n$ clauses and needs $n-1$ new variables:

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

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

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

We implement the encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]          g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[-l₁, y], [-y, -l₂]], 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, y], [-y, z], [-y, -l₂]] ++ F_rec, g₂⟩
```

Case study: at-most-one

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

We implement the encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]         g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[-l₁, y], [-y, -l₂]], 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, y], [-y, z], [-y, -l₂]] ++ F_rec, g₂⟩
```

Case study: at-most-one

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

We implement the encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]          g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[-l₁, y], [-y, -l₂]], 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, y], [-y, z], [-y, -l₂]] ++ F_rec, g₂⟩
```

Case study: at-most-one

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

We implement the encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]          g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[-l₁, y], [-y, -l₂]], 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, y], [-y, z], [-y, -l₂]] ++ F_rec, g₂⟩
```

Case study: at-most-one

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

We implement the encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]          g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[-l₁, y], [-y, -l₂]], 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, y], [-y, z], [-y, -l₂]] ++ F_rec, g₂⟩
```

Case study: at-most-one

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

We implement the encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]          g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[-l₁, y], [-y, -l₂]], 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, y], [-y, z], [-y, -l₂]] ++ F_rec, g₂⟩
```

Case study: at-most-one

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

We implement the encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]          g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[-l₁, y], [-y, -l₂]], 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, y], [-y, z], [-y, -l₂]] ++ F_rec, g₂⟩
```

Case study: at-most-one

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

We implement the encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]          g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[-l₁, y], [-y, -l₂]], 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, y], [-y, z], [-y, -l₂]] ++ F_rec, g₂⟩
```

Case study: at-most-one

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

We implement the encodings in Lean's functional programming language:

```
def Sinz_amo : enc_fn
| [l₁, l₂]          g :=
  let ⟨y, g₁⟩ := g.fresh in
  ⟨[[-l₁, y], [-y, -l₂]], 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, y], [-y, z], [-y, -l₂]] ++ F_rec, g₂⟩
```

Proof methods

The method of correctness proof follows the form of encoding function (recursive $\Rightarrow$ induction, non-recursive $\Rightarrow$ "direct")

Proof methods

The method of correctness proof follows the form of encoding function (recursive $\Rightarrow$ induction, non-recursive $\Rightarrow$ "direct")

We must be careful with the fresh variables:

$$\text{XOR}(x_1, \ldots, x_n) \equiv \text{XOR}(x_1, \ldots, x_{k-1}, \overline{y}) \oplus \text{XOR}(y, x_k, \ldots, x_n)$$

Proof methods

The method of correctness proof follows the form of encoding function (recursive $\Rightarrow$ induction, non-recursive $\Rightarrow$ "direct")

We must be careful with the fresh variables:

$$\text{XOR}(x_1, \ldots, x_n) \equiv \text{XOR}(x_1, \ldots, x_{k-1}, \overline{y}) \oplus \text{XOR}(y, x_k, \ldots, x_n)$$

IH used on $S = \{y, x_k, \ldots, x_n\}$, so the assignment given back extends S. But τ is defined on $\{x_1, \ldots, x_n\}$.

Proof methods

The method of correctness proof follows the form of encoding function (recursive $\Rightarrow$ induction, non-recursive $\Rightarrow$ "direct")

We must be careful with the fresh variables:

$$\text{XOR}(x_1, \ldots, x_n) \equiv \text{XOR}(x_1, \ldots, x_{k-1}, \overline{y}) \oplus \text{XOR}(y, x_k, \ldots, x_n)$$

IH used on $S = \{y, x_k, \ldots, x_n\}$, so the assignment given back extends S. But τ is defined on $\{x_1, \ldots, x_n\}$.

In "direct" proofs, supply extended assignments explicitly

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} :
  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} :
  is_correct c₁ enc₁ → is_correct c₂ enc₂ →
  is_correct (c₁ ++ c₂) (enc₁ ++ enc₂) := ...
```

Toy example by combining sub-encodings for Sudoku (demo!)

Applications and future work

- Prove more (sub-)encodings correct

Applications and future work

- Prove more (sub-)encodings correct
- Re-write variable generation in terms of a monad

Applications and future work

- Prove more (sub-)encodings correct
- Re-write variable generation in terms of a monad
- Prove the Keller SAT reduction correct

Applications and future work

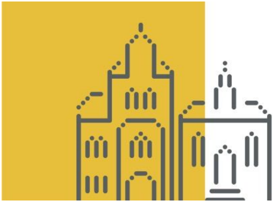
- Prove more (sub-)encodings correct
- Re-write variable generation in terms of a monad
- Prove the Keller SAT reduction correct
- Write verified proof checkers for SAT proof systems

Applications and future work

- Prove more (sub-)encodings correct
- Re-write variable generation in terms of a monad
- Prove the Keller SAT 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

Verified encodings for SAT solvers



Thank you for your attention!
Any questions?