VERIFIED VAMPIRE PROOFS IN $\lambda\Pi$ -CALCULUS MODULO

(A MACHINE CHECKABLE PROOF OUTPUT MODE)

ANJA PETKOVIĆ KOMEL, MICHAEL RAWSON, MARTIN SUDA

EUROPROOFNET SYMPOSIUM: WORKSHOP ON AUTOMATED REASONING AND PROOF LOGGING, SEPTEMBER 2025

TWO APPROACHES TO FORMAL PROOFS

TWO APPROACHES TO FORMAL PROOFS

Automated theorem prover

(ATP)

TWO APPROACHES TO FORMAL PROOFS

Automated theorem prover

(ATP)

Interactive theorem prover

(ITP)

AUTOMATED THEOREM PROVER (ATP)

AUTOMATED THEOREM PROVER (ATP)

INPUT

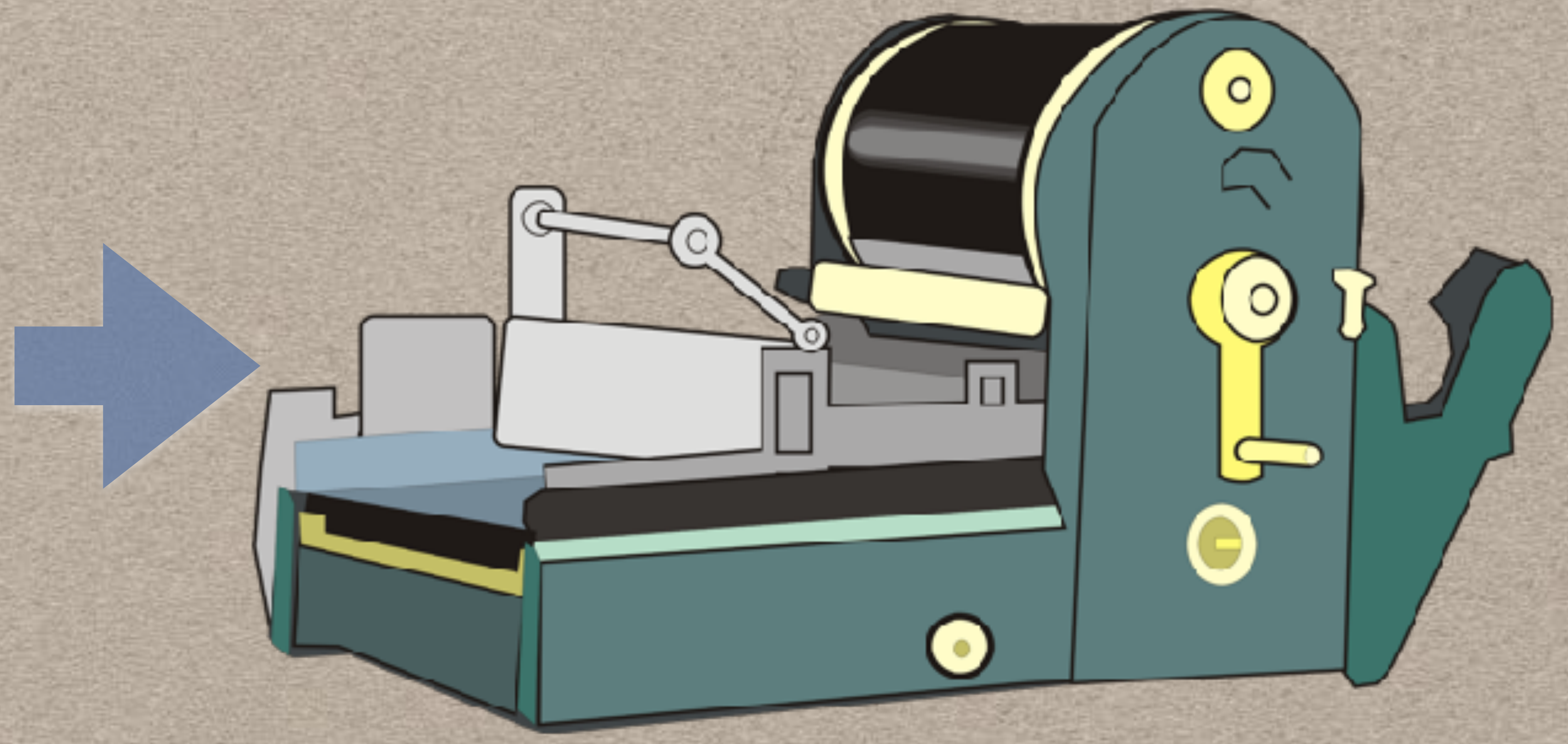
- Axioms
- (negated) conjecture

AUTOMATED THEOREM PROVER (ATP)

INPUT

- Axioms
- (negated) conjecture

ATP



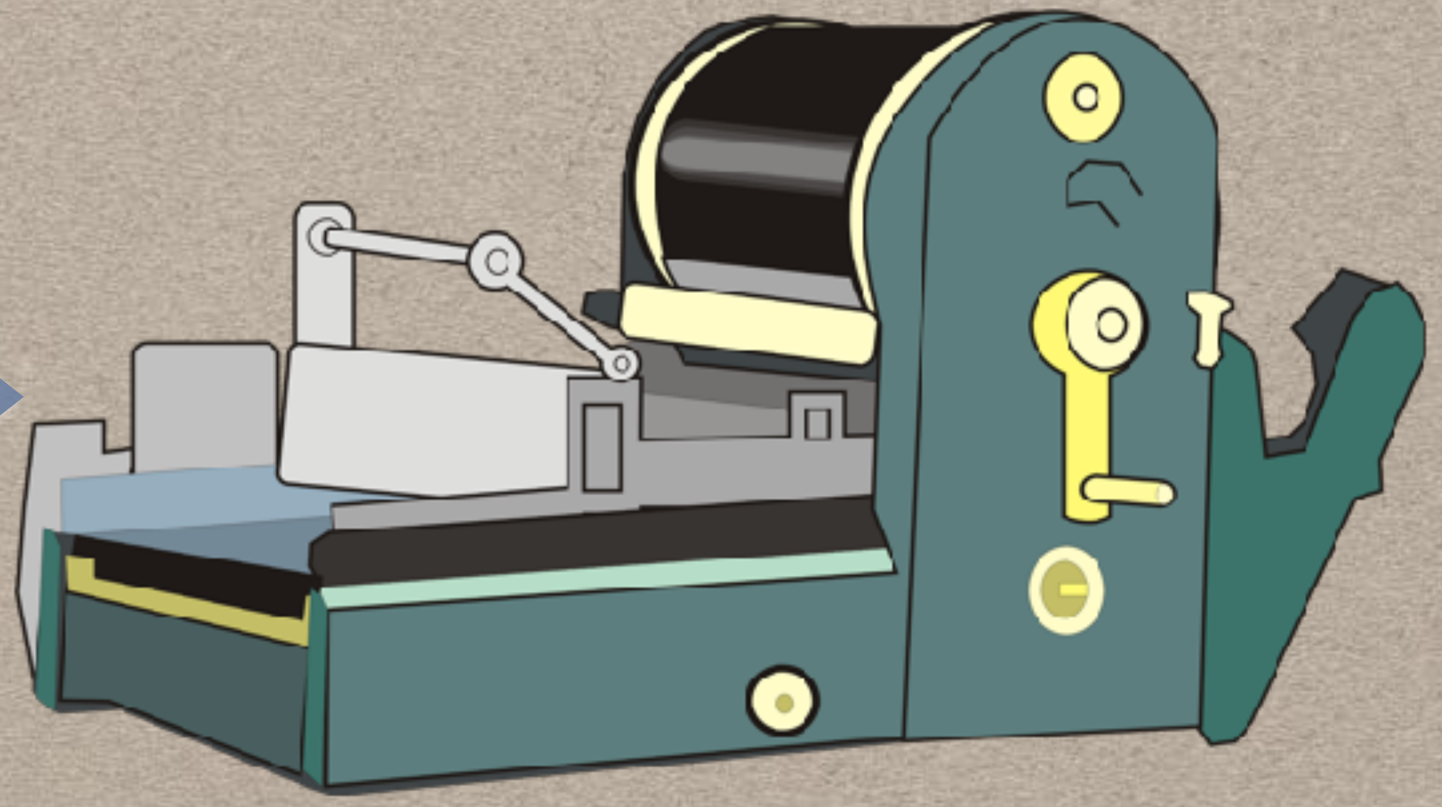
🙏 🤞 🍀

AUTOMATED THEOREM PROVER (ATP)

INPUT

- Axioms
- (negated) conjecture

ATP



OUTPUT

Proof trace

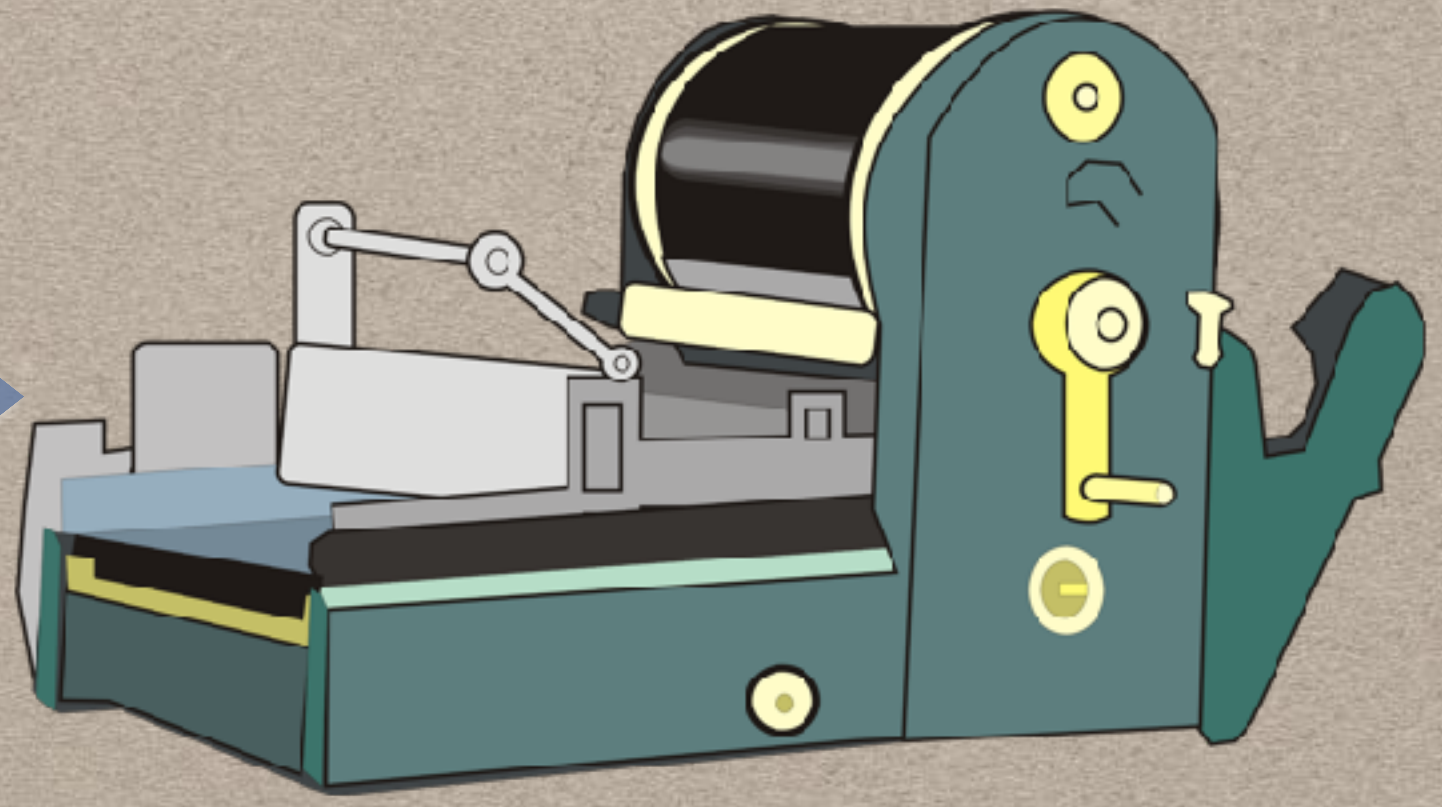
🙏🤞🍀

AUTOMATED THEOREM PROVER (ATP)

INPUT

- Axioms
- (negated) conjecture

ATP



OUTPUT

Proof trace

- Sequence / directed graph of (first-order) formulas

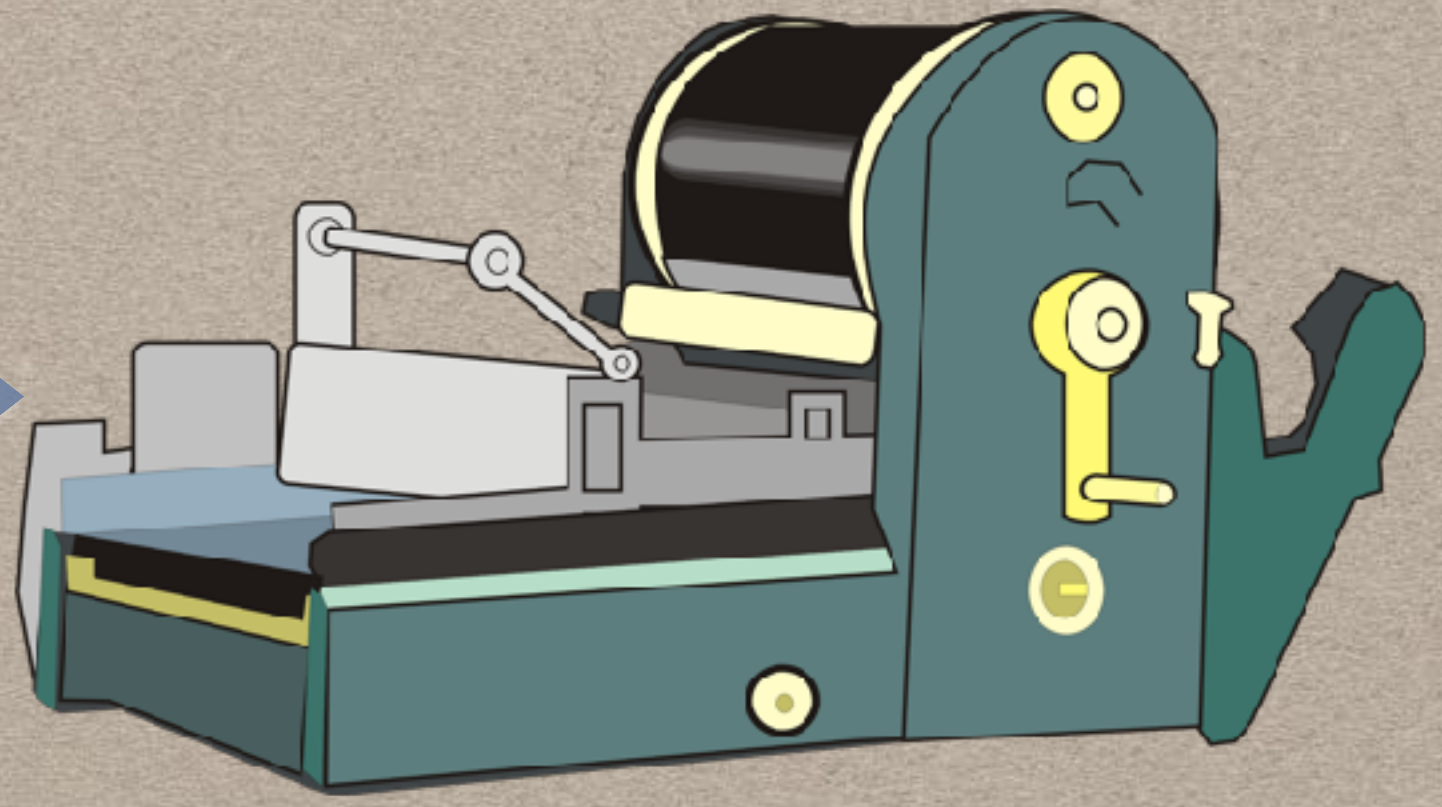
🙏🤞🍀

AUTOMATED THEOREM PROVER (ATP)

INPUT

- Axioms
- (negated) conjecture

ATP



OUTPUT

Proof trace

- Sequence / directed graph of (first-order) formulas
- Which premises and inference rules were used

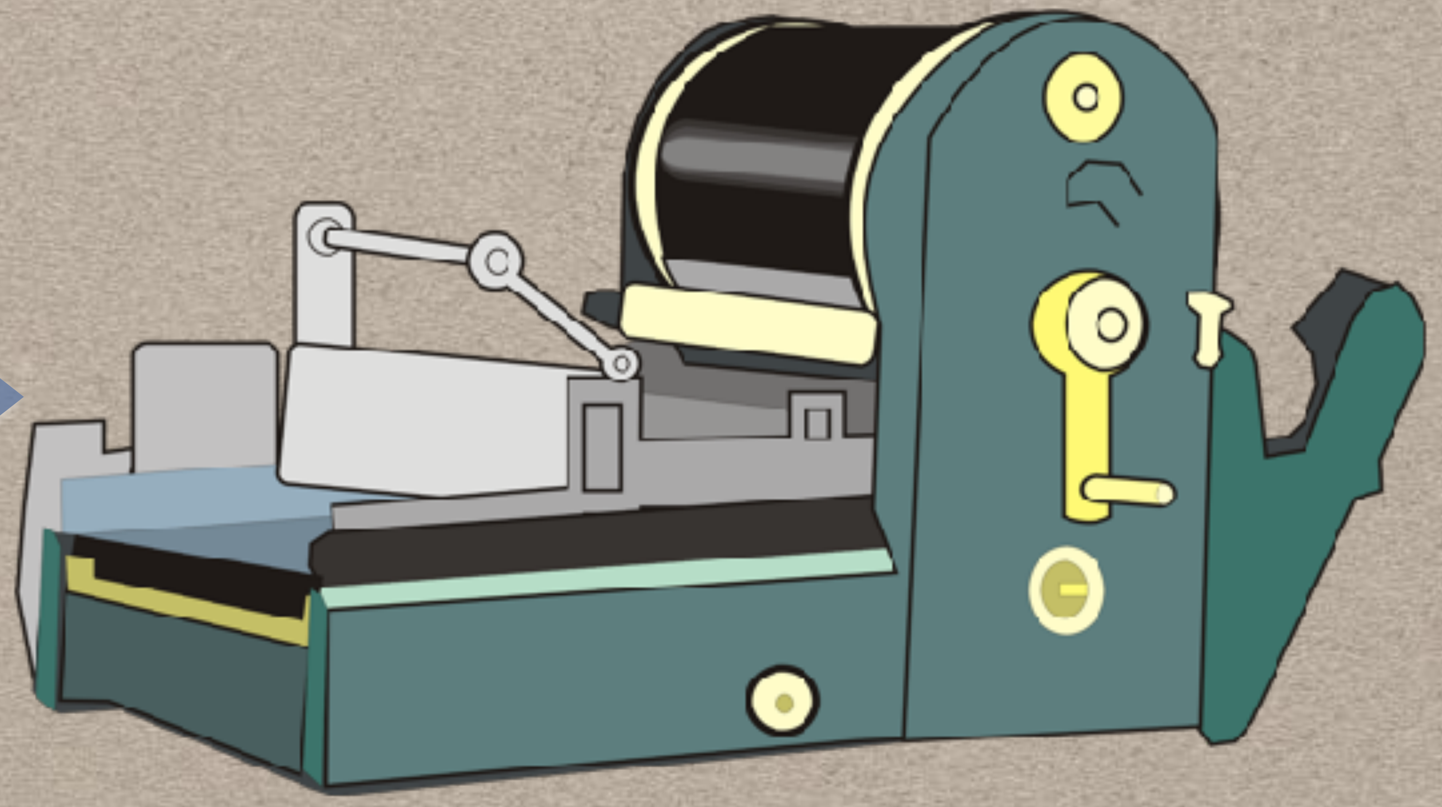
🙏 🤞 🍀

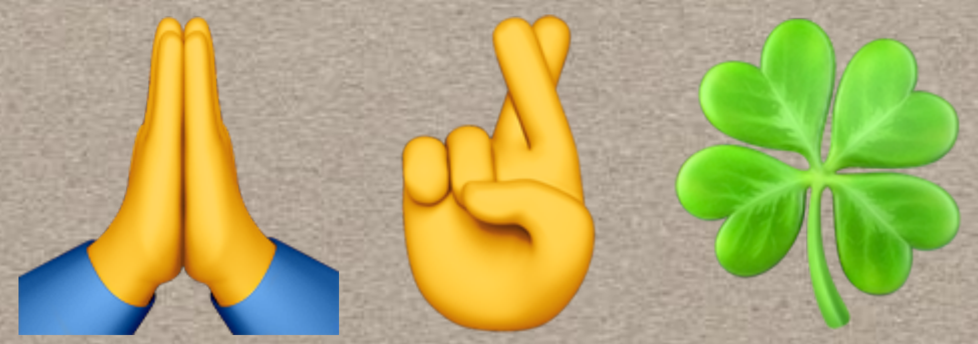
AUTOMATED THEOREM PROVER (ATP)

INPUT

- Axioms
- (negated) conjecture

ATP





OUTPUT

Proof trace

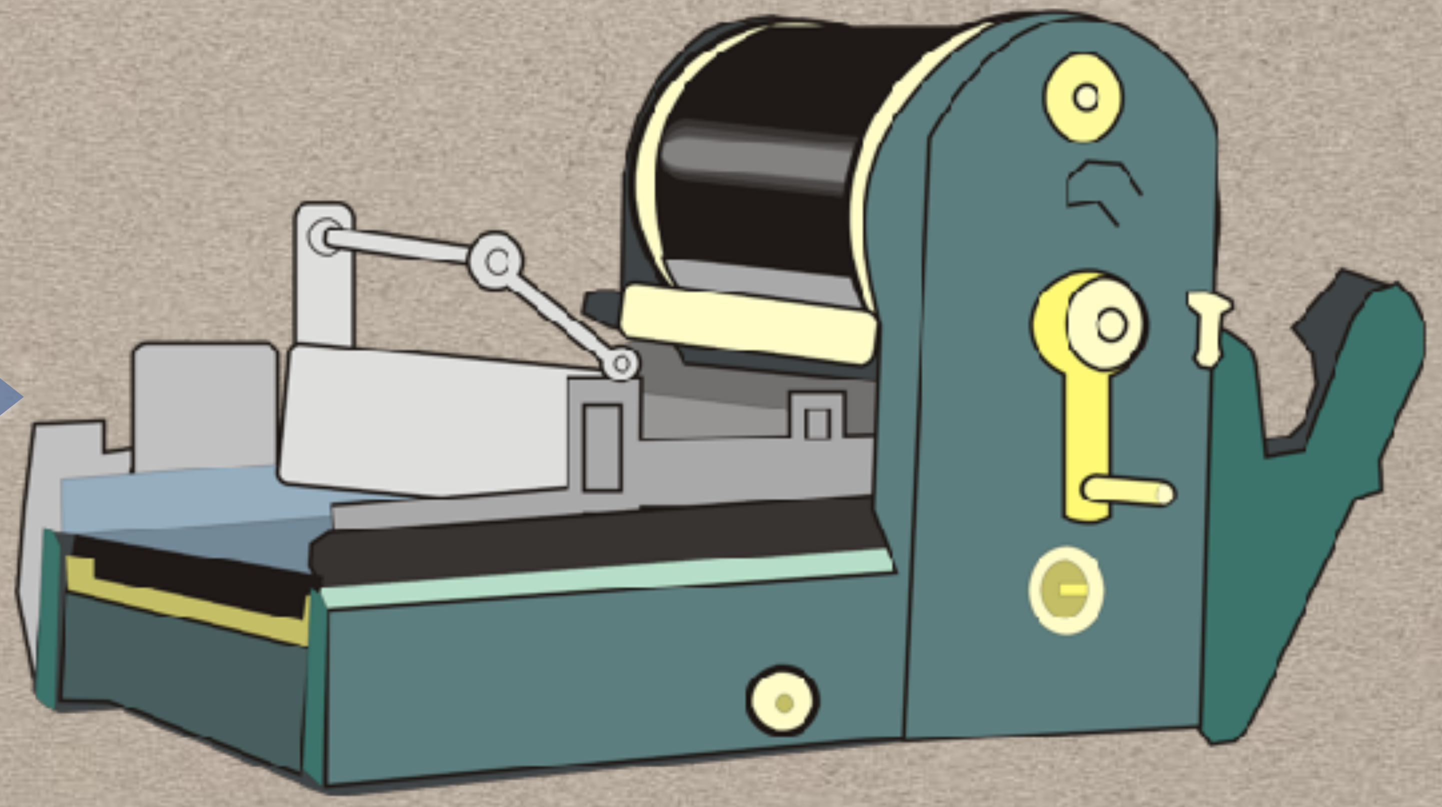
- Sequence / directed graph of (first-order) formulas
- Which premises and inference rules were used
- Should be enough to reconstruct the proof

AUTOMATED THEOREM PROVER (ATP)

INPUT

- Axioms
- (negated) conjecture

ATP



OUTPUT

Proof trace

- Can be used as a "black box"

🙏🤞🍀

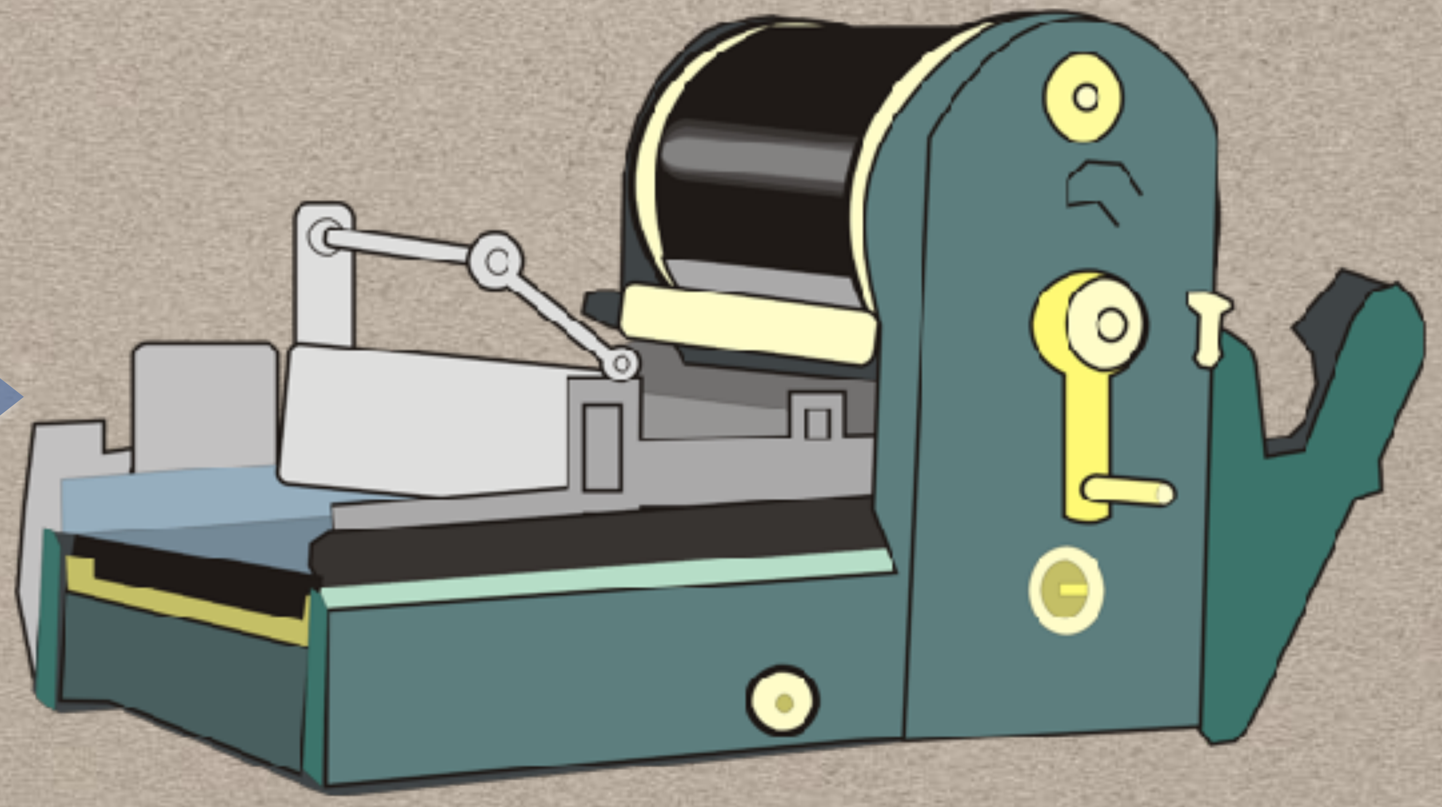
- Sequence / directed graph of (first-order) formulas
- Which premises and inference rules were used
- Should be enough to reconstruct the proof

AUTOMATED THEOREM PROVER (ATP)

INPUT

- Axioms
- (negated) conjecture

ATP



OUTPUT

Proof trace

- Can be used as a "black box"
- Complex piece of software

🙏 🤞 🍀

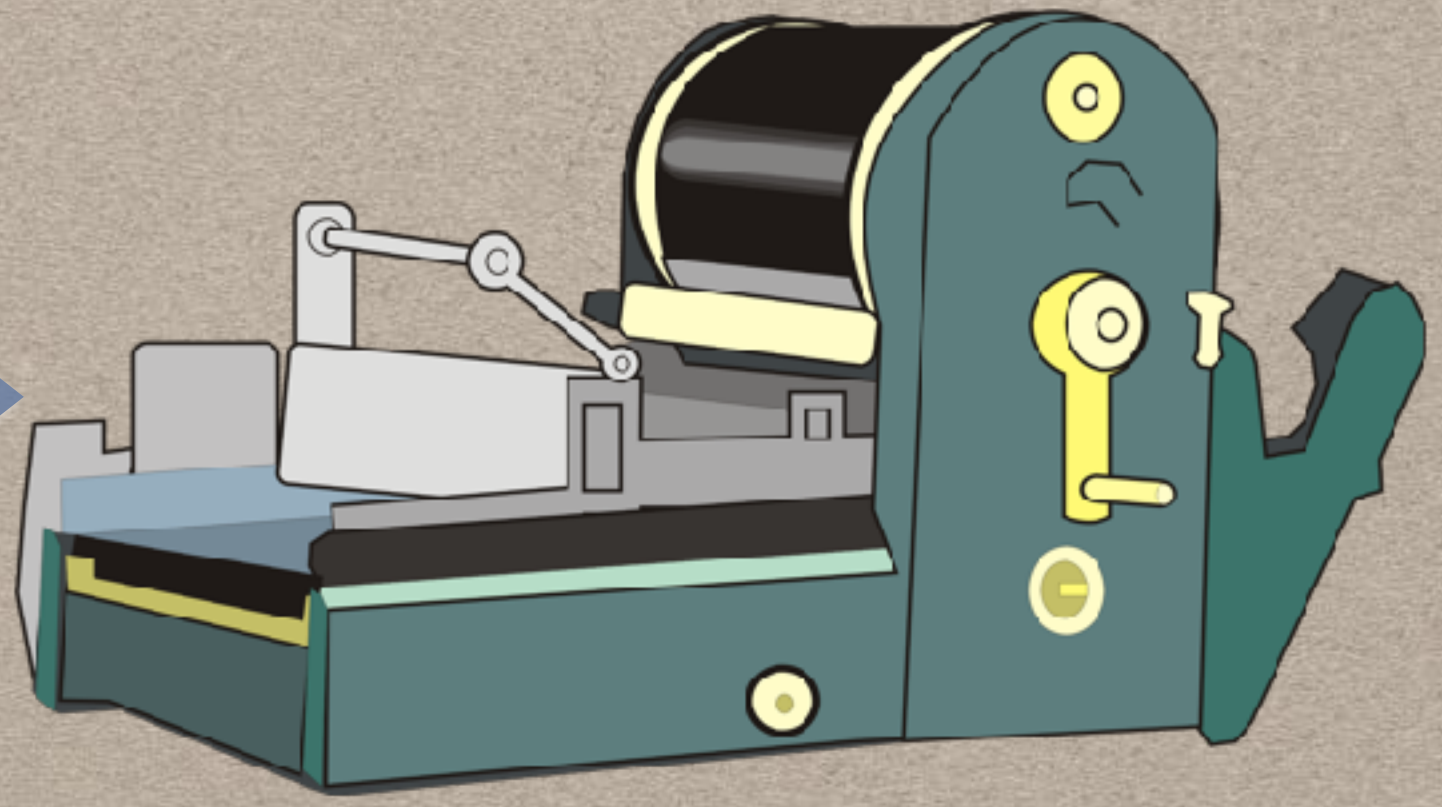
- Sequence / directed graph of (first-order) formulas
- Which premises and inference rules were used
- Should be enough to reconstruct the proof

AUTOMATED THEOREM PROVER (ATP)

INPUT

- Axioms
- (negated) conjecture

ATP



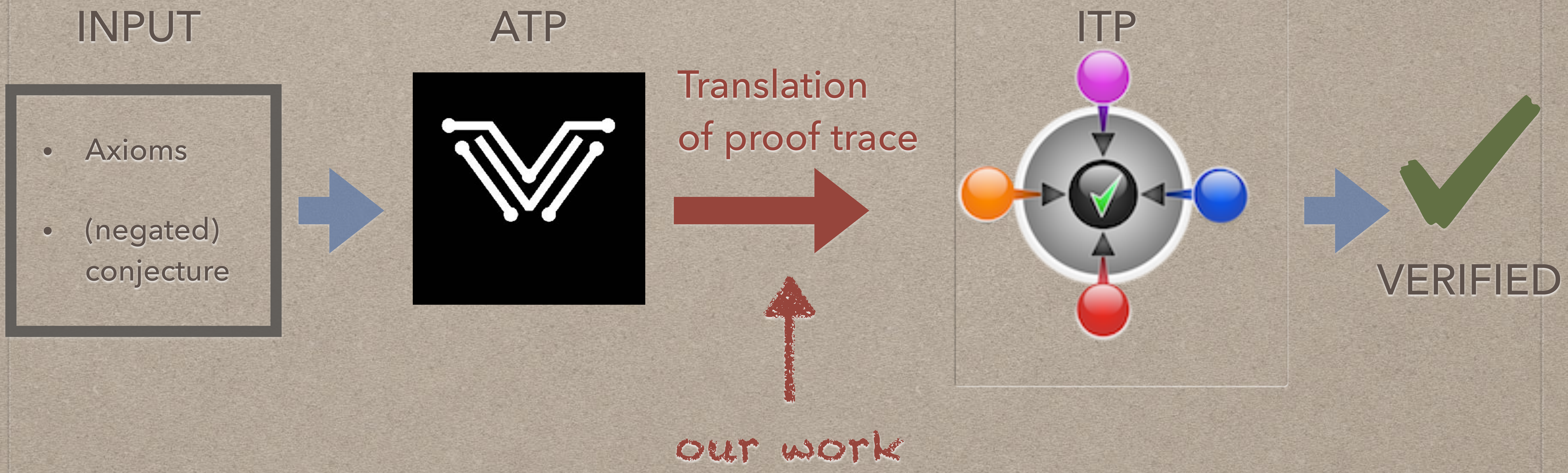
OUTPUT

Proof trace

- Can be used as a "black box"
- Complex piece of software
- Big trusted code base

- Sequence / directed graph of (first-order) formulas
- Which premises and inference rules were used
- Should be enough to reconstruct the proof

FROM VAMPIRE TO DEDUKTI



WHY OUTPUT VAMPIRE PROOFS TO DEDUKTI?

WHY OUTPUT VAMPIRE PROOFS TO DEDUKTI?

- Find unsoundness bugs

WHY OUTPUT VAMPIRE PROOFS TO DEDUKTI?

- Find unsoundness bugs

- Bugs have been found and some are likely still there.

WHY OUTPUT VAMPIRE PROOFS TO DEDUKTI?

- Find unsoundness bugs

- Bugs have been found and some are likely still there.
- Most recent unsoundess Vampire bug: January 2nd 2025, UWA + HO

WHY OUTPUT VAMPIRE PROOFS TO DEDUKTI?

- Find unsoundness bugs
- Have high degree of confidence

- Bugs have been found and some are likely still there.
- Most recent unsoundess Vampire bug: January 2nd 2025, UWA + HO

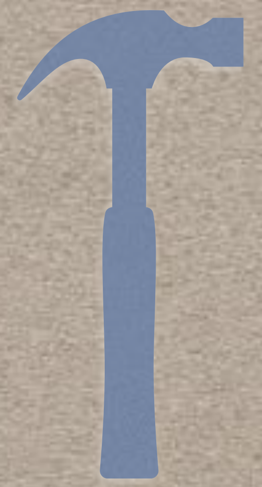
WHY OUTPUT VAMPIRE PROOFS TO DEDUKTI?

- Find unsoundness bugs
- Have high degree of confidence
- Interoperability of proofs

- Bugs have been found and some are likely still there.
- Most recent unsoundess Vampire bug: January 2nd 2025, UWA + HO

WHY OUTPUT VAMPIRE PROOFS TO DEDUKTI?

- Find unsoundness bugs
- Have high degree of confidence
- Interoperability of proofs
- Potential for hammers



- Bugs have been found and some are likely still there.
- Most recent unsoundess Vampire bug: January 2nd 2025, UWA + HO

VAMPIRE

- **First-order system**, extended with:
  - ★ reasoning with theories
  - ★ induction
  - ★ higher-order logic
- **Saturation based** theorem prover
- Employs **a number of techniques**: indexing, scheduling, ordered rewriting, AVATAR, heuristics, etc.

WHY DEDUKTI?

WHY DEDUKTI?

dedukti

WHY DEDUKTI?

dedukti

- Choir: Interoperability, has a small trusted kernel, powerful enough to express our language

WHY DEDUKTI?

- Choir: Interoperability, has a small trusted kernel, powerful enough to express our language
- Scales reasonably well: designed to machine check large proofs
(unlike lambdapi?)

WHAT ARE WE DOING

WHAT ARE WE DOING

- Proof format "-p dedukti" (on vampire branch dedukti)

vampire $problem -p dedukti - - proof_extra full | dk check

WHAT ARE WE DOING

- Proof format “-p dedukti” (on vampire branch dedukti)

vampire $problem -p dedukti - - proof_extra full | dk check

- Using **standard Dedukti encoding of FOL** and Dedukti semantics encode **concrete instances** of Vampire inferences

WHAT ARE WE DOING

- Proof format "-p dedukti" (on vampire branch dedukti)

vampire $problem -p dedukti - - proof_extra full | dk check

- Using **standard Dedukti encoding of FOL** and Dedukti semantics encode **concrete instances** of Vampire inferences

- Sometimes need to store extra information in the proof: "–proof_extra full"

More on that later

STANDARD ENCODING OF FOL IN DEDUKTI

```
(; Prop ;)
Prop : Type.
def Prf : (Prop -> Type).
true : Prop.
[] Prf true --> (r : Prop -> ((Prf r) -> (Prf r))).
false : Prop.
[] Prf false --> (r : Prop -> (Prf r)).
not : (Prop -> Prop).
[p] Prf (not p) --> ((Prf p) -> (r : Prop -> (Prf r))).
and : (Prop -> (Prop -> Prop)).
[p, q] Prf (and p q) --> (r : Prop -> (((Prf p) -> ((Prf q) -> (Prf r))) -> (Prf r))).
or : (Prop -> (Prop -> Prop)).
[p, q] Prf (or p q) --> (((Prf p) -> (Prf false)) -> (((Prf q) -> (Prf false)) -> (Prf false))).
imp : (Prop -> (Prop -> Prop)).
[p, q] Prf (imp p q) --> ((Prf p) -> (Prf q)).
iff : Prop -> Prop -> Prop.
[p, q] Prf (iff p q) --> (Prf (and (imp p q) (imp q p))).

(; Set ;)
Set : Type.
injective El : (Set -> Type).
iota : Set.
inhabit : A : Set -> El A.

(; Equality ;)
def eq : a : Set -> El a -> El a -> Prop.
[a, x, y] Prf (eq a x y) --> p : (El a -> Prop) -> Prf (p x) -> Prf (p y).
```

```
(; Prop ;)
Prop : Type.
def Prf : (Prop -> Type).
true : Prop.
[] Prf true --> (r : Prop -> ((Prf r) -> (Prf r))).
false : Prop.
[] Prf false --> (r : Prop -> (Prf r)).
not : (Prop -> Prop).
[p] Prf (not p) --> ((Prf p) -> (r : Prop -> (Prf r))).
and : (Prop -> (Prop -> Prop)).
[p, q] Prf (and p q) --> (r : Prop -> (((Prf p) -> ((Prf q) -> (Prf r))) -> (Prf r))).
or : (Prop -> (Prop -> Prop)).
[p, q] Prf (or p q) --> (((Prf p) -> (Prf false)) -> (((Prf q) -> (Prf false)) -> (Prf false))).
imp : (Prop -> (Prop -> Prop)).
[p, q] Prf (imp p q) --> ((Prf p) -> (Prf q)).
iff : Prop -> Prop -> Prop.
[p, q] Prf (iff p q) --> (Prf (and (imp p q) (imp q p))).

(; Set ;)
Set : Type.
injective El : (Set -> Type).
iota : Set.
inhabit : A : Set -> El A.

(; Equality ;)
def eq : a : Set -> El a -> El a -> Prop.
[a, x, y] Prf (eq a x y) --> p : (El a -> Prop) -> Prf (p x) -> Prf (p y).
```

General or instantiated with false.

```
(; Prop ;)
Prop : Type.
def Prf : (Prop -> Type).
true : Prop.
[] Prf true --> (r : Prop -> ((Prf r) -> (Prf r))).
false : Prop.
[] Prf false --> (r : Prop -> (Prf r)).
not : (Prop -> Prop).
[p] Prf (not p) --> ((Prf p) -> (r : Prop -> (Prf r))).
and : (Prop -> (Prop -> Prop)).
[p, q] Prf (and p q) --> (r : Prop -> (((Prf p) -> ((Prf q) -> (Prf r))) -> (Prf r))).
or : (Prop -> (Prop -> Prop)).
[p, q] Prf (or p q) --> (((Prf p) -> (Prf false)) -> (((Prf q) -> (Prf false)) -> (Prf false))).
imp : (Prop -> (Prop -> Prop)).
[p, q] Prf (imp p q) --> ((Prf p) -> (Prf q)).
iff : Prop -> Prop -> Prop.
[p, q] Prf (iff p q) --> (Prf (and (imp p q) (imp q p))).

(; Set ;)
Set : Type.
injective El : (Set -> Type).
iota : Set.
inhabit : A : Set -> El A.

(; Equality ;)
def eq : a : Set -> El a -> El a -> Prop.
[a, x, y] Prf (eq a x y) --> p : (El a -> Prop) -> Prf (p x) -> Prf (p y).
```

General or instantiated with false.

```
(; Prop ;)
Prop : Type.
def Prf : (Prop -> Type).
true : Prop.
[] Prf true --> (r : Prop -> ((Prf r) -> (Prf r))).
false : Prop.
[] Prf false --> (r : Prop -> (Prf r)).
not : (Prop -> Prop).
[p] Prf (not p) --> ((Prf p) -> (r : Prop -> (Prf r))).
and : (Prop -> (Prop -> Prop)).
[p, q] Prf (and p q) --> (r : Prop -> (((Prf p) -> ((Prf q) -> (Prf r))) -> (Prf r))).
or : (Prop -> (Prop -> Prop)).
[p, q] Prf (or p q) --> (((Prf p) -> (Prf false)) -> (((Prf q) -> (Prf false)) -> (Prf false))).
imp : (Prop -> (Prop -> Prop)).
[p, q] Prf (imp p q) --> ((Prf p) -> (Prf q)).
iff : Prop -> Prop -> Prop.
[p, q] Prf (iff p q) --> (Prf (and (imp p q) (imp q p))).

(; Set ;)
Set : Type.
injective El : (Set -> Type).
iota : Set.
inhabit : A : Set -> El A.

(; Equality ;)
def eq : a : Set -> El a -> El a -> Prop.
[a, x, y] Prf (eq a x y) --> p : (El a -> Prop) -> Prf (p x) -> Prf (p y).
```

General or instantiated with false.

Polymorphic Leibniz encoding of equality.

```
(; Prop ;)
Prop : Type.
def Prf : (Prop -> Type).
true : Prop.
[] Prf true --> (r : Prop -> ((Prf r) -> (Prf r))).
false : Prop.
[] Prf false --> (r : Prop -> (Prf r)).
not : (Prop -> Prop).
[p] Prf (not p) --> ((Prf p) -> (r : Prop -> (Prf r))).
and : (Prop -> (Prop -> Prop)).
[p, q] Prf (and p q) --> (r : Prop -> (((Prf p) -> ((Prf q) -> (Prf r))) -> (Prf r))).
or : (Prop -> (Prop -> Prop)).
[p, q] Prf (or p q) --> (((Prf p) -> (Prf false)) -> (((Prf q) -> (Prf false)) -> (Prf false))).
imp : (Prop -> (Prop -> Prop)).
[p, q] Prf (imp p q) --> ((Prf p) -> (Prf q)).
iff : Prop -> Prop -> Prop.
[p, q] Prf (iff p q) --> (Prf (and (imp p q) (imp q p))).

(; Set ;)
Set : Type.
injective El : (Set -> Type).
iota : Set.
inhabit : A : Set -> El A.

(; Equality ;)
def eq : a : Set -> El a -> El a -> Prop.
[a, x, y] Prf (eq a x y) --> p : (El a -> Prop) -> Prf (p x) -> Prf (p y).
```

General or instantiated with false.

```
(; Quant ;)
forall : (a : Set -> (((El a) -> Prop) -> Prop)).
[a, p] Prf (forall a p) --> (x : (El a) -> (Prf (p x))).
exists : (a : Set -> (((El a) -> Prop) -> Prop)).
[a, p] Prf (exists a p) --> (r : Prop -> ((x : (El a) -> ((Prf (p x)) -> (Prf r))) -> (Prf r))).

(; polymorphic quantifier ;)
forall_poly : (Set -> Prop) -> Prop.
[p] Prf (forall_poly p) --> a : Set -> Prf (p a).
```

Polymorphic Leibniz encoding of equality.

```
(; Clauses ;)
def prop_clause : Type.
def ec : prop_clause.
def cons : (Prop -> (prop_clause -> prop_clause)).
def clause : Type.
def cl : (prop_clause -> clause).
def bind : (A : Set -> (((El A) -> clause) -> clause)).
def bind_poly : (Set -> clause) -> clause.
def Prf_prop_clause : (prop_clause -> Type).

[] Prf_prop_clause ec --> (Prf false).
[p, c] Prf_prop_clause (cons p c) --> ((Prf p -> Prf false) -> (Prf_prop_clause c)).
def Prf_clause : (clause -> Type).
[c] Prf_clause (cl c) --> (Prf_prop_clause c).
[A, f] Prf_clause (bind A f) --> (x : (El A) -> (Prf_clause (f x))).
[f] Prf_clause (bind_poly f) --> (alpha : Set -> (Prf_clause (f alpha))).

def av_clause : Type.
def acl : clause -> av_clause.
def if : Prop -> av_clause -> av_clause.
def Prf_av_clause : av_clause -> Type.

[c] Prf_av_clause (acl c) --> Prf_clause c.
[sp, c] Prf_av_clause (if sp c) --> (Prf (not sp) -> Prf false) -> Prf_av_clause c.
```

```
(; Clauses ;)
def prop_clause : Type.
def ec : prop_clause.
def cons : (Prop -> (prop_clause -> prop_clause)).
def clause : Type.
def cl : (prop_clause -> clause).
def bind : (A : Set -> (((El A) -> clause) -> clause)).
def bind_poly : (Set -> clause) -> clause.
def Prf_prop_clause : (prop_clause -> Type).

[] Prf_prop_clause ec --> (Prf false).
[p, c] Prf_prop_clause (cons p c) --> ((Prf p -> Prf false) -> (Prf_prop_clause c)).
def Prf_clause : (clause -> Type).
[c] Prf_clause (cl c) --> (Prf_prop_clause c).
[A, f] Prf_clause (bind A f) --> (x : (El A) -> (Prf_clause (f x))).
[f] Prf_clause (bind_poly f) --> (alpha : Set -> (Prf_clause (f alpha))).

def av_clause : Type.
def acl : clause -> av_clause.
def if : Prop -> av_clause -> av_clause.
def Prf_av_clause : av_clause -> Type.

[c] Prf_av_clause (acl c) --> Prf_clause c.
[sp, c] Prf_av_clause (if sp c) --> (Prf (not sp) -> Prf false) -> Prf_av_clause c.
```

Encoding of disjunction in clauses.

```
(; Clauses ;)
def prop_clause : Type.
def ec : prop_clause.
def cons : (Prop -> (prop_clause -> prop_clause)).
def clause : Type.
def cl : (prop_clause -> clause).
def bind : (A : Set -> (((El A) -> clause) -> clause)).
def bind_poly : (Set -> clause) -> clause.
def Prf_prop_clause : (prop_clause -> Type).

[] Prf_prop_clause ec --> (Prf false).
[p, c] Prf_prop_clause (cons p c) --> ((Prf p -> Prf false) -> (Prf_prop_clause c)).
def Prf_clause : (clause -> Type).
[c] Prf_clause (cl c) --> (Prf_prop_clause c).
[A, f] Prf_clause (bind A f) --> (x : (El A) -> (Prf_clause (f x))).
[f] Prf_clause (bind_poly f) --> (alpha : Set -> (Prf_clause (f alpha))).

def av_clause : Type.
def acl : clause -> av_clause.
def if : Prop -> av_clause -> av_clause.
def Prf_av_clause : av_clause -> Type.

[c] Prf_av_clause (acl c) --> Prf_clause c.
[sp, c] Prf_av_clause (if sp c) --> (Prf (not sp) -> Prf false) -> Prf_av_clause c.
```

Encoding of disjunction in clauses.

"empty" AVATAR clause is regular clause.

```
(; Clauses ;)
def prop_clause : Type.
def ec : prop_clause.
def cons : (Prop -> (prop_clause -> prop_clause)).
def clause : Type.
def cl : (prop_clause -> clause).
def bind : (A : Set -> (((El A) -> clause) -> clause)).
def bind_poly : (Set -> clause) -> clause.
def Prf_prop_clause : (prop_clause -> Type).

[] Prf_prop_clause ec --> (Prf false)
[p, c] Prf_prop_clause (
def Prf_clau
[c
[A,                                    (Prf_clause (f x))).
[f]                             alpha : Set -> (Prf_clause (f alpha))).

def a          se : Type.
def acl : clause -> av_clause.
def if : Prop -> av_clause -> av_clause.
def Prf_av_clause : av_clause -> Type.

[c] Prf_av_clause (acl c) --> Prf_clause c.
[sp, c] Prf_av_clause (if sp c) --> (Prf (not sp) -> Prf false) -> Prf_av_clause c.
```

All this rewrites/normalizes to encoding of FOL.

We did not introduce any new axioms.

“empty” AVATAR clause is regular clause.

WHAT WE CAN CURRENTLY DO

WHAT WE CAN CURRENTLY DO

- **Parse input problem**: read TPTP/SMT-LIB, write Dedukti axioms

WHAT WE CAN CURRENTLY DO

- **Parse input problem**: read TPTP/SMT-LIB, write Dedukti axioms
- **Run vampire in default mode** and **fully check** reasoning steps:
  - Resolution
  - Forward/backward demodulation
  - Superposition
  - Subsumption resolution
  - Equality resolution
  - AVATAR
  - Trivial equality removal
  - Factoring
  - Remove duplicate literals
  - Some pre-processing steps
    - ★ Equality resolution with deletion
    - ★ Definiton unfolding

WHAT WE CAN CURRENTLY DO

- **Parse input problem**: read TPTP/SMT-LIB, write Dedukti axioms
- **Run vampire in default mode** and **fully check** reasoning steps:
  - Resolution
  - Forward/backward demodulation
  - Superposition
  - Subsumption resolution
  - Equality resolution
  - AVATAR ← Presented later
  - Trivial equality removal
  - Factoring
  - Remove duplicate literals
  - Some pre-processing steps
    - ★ Equality resolution with deletion
    - ★ Definiton unfolding
- **Many sorted logic and polymorphism** are supported

CURRENT (TEMPORARY) LIMITATIONS

CURRENT (TEMPORARY) LIMITATIONS

Full first-order formulas (FOF) can be parsed, but clausification steps are not checked.

- Limited pre-processing: clausification is not checked

CURRENT (TEMPORARY) LIMITATIONS

Full first-order formulas (FOF) can be parsed, but clausification steps are not checked.

- Limited pre-processing: clausification is not checked
- No higher-order reasoning and no theories yet

CURRENT (TEMPORARY) LIMITATIONS

Full first-order formulas (FOF) can be parsed, but clausification steps are not checked.

- Limited pre-processing: clausification is not checked
- No higher-order reasoning and no theories yet
- Turn off all the fun bits (only inferences on the previous slide)

CURRENT (TEMPORARY) LIMITATIONS

Full first-order formulas (FOF) can be parsed, but clausification steps are not checked.

- Limited pre-processing: clausification is not checked
- No higher-order reasoning and no theories yet
- Turn off all the fun bits (only inferences on the previous slide)

★ But inferences are done **incrementally** :)

CURRENT (TEMPORARY) LIMITATIONS

Full first-order formulas (FOF) can be parsed, but clausification steps are not checked.

- Limited pre-processing: clausification is not checked
- No higher-order reasoning and no theories yet
- Turn off all the fun bits (only inferences on the previous slide)
  - ★ But inferences are done **incrementally** :)
  - ★ Unsupported inferences are handled by "**sorry**".

We emit a warning during type-checking when there is a sorry.

SOME LESSONS LEARNED DURING IMPLEMENTATION

SOME LESSONS LEARNED DURING IMPLEMENTATION

- Equality is symmetric:

SOME LESSONS LEARNED DURING IMPLEMENTATION

- **Equality is symmetric**:
  - Vampire will switch LHS and RHS when convenient.

SOME LESSONS LEARNED DURING IMPLEMENTATION

- **Equality is symmetric**:
  - Vampire will switch LHS and RHS when convenient.
  - Manually insert commutativity lemmas during proof-printing.

SOME LESSONS LEARNED DURING IMPLEMENTATION

- **Equality is symmetric**:
  - Vampire will switch LHS and RHS when convenient.
  - Manually insert commutativity lemmas during proof-printing.
- **Numbering deduction steps accordingly**: we give the deduction steps in Deduct script the same number as appears in the “default” Vampire proof trace, to ease uncovering bugs (if the elaborated proofs do not type check).

AVATAR

AVATAR

- Technique that greatly improves the efficiency of first-order reasoning

AVATAR

- Technique that greatly improves the efficiency of first-order reasoning
- **Splitting** clauses and **offloading** the disjunctive structure **to a SAT solver**

AVATAR

- Technique that greatly improves the efficiency of first-order reasoning
- **Splitting** clauses and **offloading** the disjunctive structure **to a SAT solver**
- For proof logging: **introducing propositional labels**

AVATAR INFERENCES

AVATAR INFERENCES

Definition: propositional label to a sub-clause

$$\mathsf{sp}_1 \equiv \forall xy.\ P(x, f(y)) \vee \neg Q(y)$$

$$\mathsf{sp}_2 \equiv \forall z.\ c = z$$

AVATAR INFERENCES

Definition: propositional label to a sub-clause

$$\mathsf{sp}_1 \equiv \forall xy.\ P(x, f(y)) \vee \neg Q(y)$$

$$\mathsf{sp}_2 \equiv \forall z.\ c = z$$

Split: clauses split into variable-disjoint components, deriving SAT clause

$$\frac{\neg Q(z) \vee c = y \vee P(x, f(z))}{\mathsf{sp}_1 \vee \mathsf{sp}_2}$$

AVATAR INFERENCES

Definition: propositional label to a sub-clause

$$\mathsf{sp}_1 \equiv \forall xy.\ P(x, f(y)) \vee \neg Q(y)$$
$$\mathsf{sp}_2 \equiv \forall z.\ c = z$$

Split: clauses split into variable-disjoint components, deriving SAT clause

$$\frac{\neg Q(z) \vee c = y \vee P(x, f(z))}{\mathsf{sp}_1 \vee \mathsf{sp}_2}$$

Component: injected into the search space, conditionally on the split label

$$P(x, f(y)) \vee \neg Q(y) \leftarrow \mathsf{sp}_1$$
$$c = z \leftarrow \mathsf{sp}_2$$

AVATAR INFERENCES

Definition: propositional label to a sub-clause

$$\mathsf{sp}_1 \equiv \forall xy.\ P(x, f(y)) \vee \neg Q(y)$$
$$\mathsf{sp}_2 \equiv \forall z.\ c = z$$

Split: clauses split into variable-disjoint components, deriving SAT clause

$$\frac{\neg Q(z) \vee c = y \vee P(x, f(z))}{\mathsf{sp}_1 \vee \mathsf{sp}_2}$$

Avatar clause: all existing inferences work conditionally on avatar splits (conjunction of splits of parents)

Component: injected into the search space, conditionally on the split label

$$P(x, f(y)) \vee \neg Q(y) \leftarrow \mathsf{sp}_1$$
$$c = z \leftarrow \mathsf{sp}_2$$

AVATAR INFERENCES

Definition: propositional label to a sub-clause

$$\mathsf{sp}_1 \equiv \forall xy.\ P(x, f(y)) \vee \neg Q(y)$$
$$\mathsf{sp}_2 \equiv \forall z.\ c = z$$

Split: clauses split into variable-disjoint components, deriving SAT clause

$$\frac{\neg Q(z) \vee c = y \vee P(x, f(z))}{\mathsf{sp}_1 \vee \mathsf{sp}_2}$$

Avatar clause: all existing inferences work conditionally on avatar splits (conjunction of splits of parents)

Component: injected into the search space, conditionally on the split label

$$P(x, f(y)) \vee \neg Q(y) \leftarrow \mathsf{sp}_1$$
$$c = z \leftarrow \mathsf{sp}_2$$

Contradiction: false conditionally on split derives SAT clause (the split)

$$\frac{\bot \leftarrow \mathsf{sp}_3 \wedge \neg \mathsf{sp}_5}{\neg \mathsf{sp}_3 \vee \mathsf{sp}_5}$$

AVATAR INFERENCES

Definition: propositional label to a sub-clause

$$\mathsf{sp}_1 \equiv \forall xy.\ P(x, f(y)) \vee \neg Q(y)$$
$$\mathsf{sp}_2 \equiv \forall z.\ c = z$$

Split: clauses split into variable-disjoint components, deriving SAT clause

$$\frac{\neg Q(z) \vee c = y \vee P(x, f(z))}{\mathsf{sp}_1 \vee \mathsf{sp}_2}$$

Avatar clause: all existing inferences work conditionally on avatar splits (conjunction of splits of parents)

Component: injected into the search space, conditionally on the split label

$$P(x, f(y)) \vee \neg Q(y) \leftarrow \mathsf{sp}_1$$
$$c = z \leftarrow \mathsf{sp}_2$$

Contradiction: false conditionally on split derives SAT clause (the split)

$$\frac{\bot \leftarrow \mathsf{sp}_3 \wedge \neg \mathsf{sp}_5}{\neg \mathsf{sp}_3 \vee \mathsf{sp}_5}$$

Refutation: last step of the proof, derives false, because SAT set is unsatisfiable.

AVATAR INFERENCES ENCODED

AVATAR INFERENCES ENCODED

Definition: just a Dedukti definition

$$\mathsf{sp}_1 : \mathsf{Prop}$$

$$\mathsf{sp}_1 \hookrightarrow \forall x, y : \mathsf{El}\ \iota.\ |\neg P(x, f(y))| \implies |\neg\neg Q(y)| \implies \bot$$

AVATAR INFERENCES ENCODED

Definition: just a Dedukti definition

$\mathsf{sp}_1 : \mathsf{Prop}$

$\mathsf{sp}_1 \hookrightarrow \forall x, y : \mathsf{El}\ \iota.\ |\neg P(x, f(y))| \implies |\neg\neg Q(y)| \implies \bot$

$$\frac{\neg Q(z) \lor c = y \lor P(x, f(z))}{\mathsf{sp}_1 \lor \mathsf{sp}_2}$$

Split: unpack with variable renaming, apply vars and literals to premise.

$C : \Pi x, y, z : \mathsf{El}\ \iota.\ [\![\neg Q(z)]\!] \to [\![c = y]\!] \to [\![P(x, f(z))]\!] \to \mathsf{Prf}\ \bot.$

Prf(c=Y) -> Prf(false)

$D : [\![\mathsf{sp}_1]\!] \to [\![\mathsf{sp}_2]\!] \to \mathsf{Prf}\ \bot:$

$$\begin{aligned} D \hookrightarrow\ & \lambda s_1 : [\![\mathsf{sp}_1]\!].\ \lambda s_2 : [\![\mathsf{sp}_2]\!].\\ & s_1\ (\lambda x, z : \mathsf{El}\ \iota.\ \lambda \ell_3 : [\![P(x, f(x))]\!].\ \lambda \ell_1 : [\![\neg Q(x)]\!].\\ & s_2\ (\lambda y : \mathsf{El}\ \iota.\ \lambda \ell_2 : [\![c = y]\!].\\ & \quad C\ x\ y\ z\ \ell_1\ \ell_2\ \ell_3)) \end{aligned}$$

AVATAR INFERENCES ENCODED

Definition: just a Dedukti definition

$$\mathsf{sp}_1 : \mathsf{Prop}$$
$$\mathsf{sp}_1 \hookrightarrow \forall x, y : \mathsf{El}\ \iota.\ |\neg P(x, f(y))| \implies |\neg\neg Q(y)| \implies \bot$$

Split: unpack with variable renaming, apply vars and literals to premise.

$$C : \Pi x, y, z : \mathsf{El}\ \iota.\ [\![\neg Q(z)]\!] \to [\![c = y]\!] \to [\![P(x, f(z))]\!] \to \mathsf{Prf}\ \bot.$$

Prf(c=Y) -> Prf(false)

$$D : [\![\mathsf{sp}_1]\!] \to [\![\mathsf{sp}_2]\!] \to \mathsf{Prf}\ \bot:$$

$$\begin{aligned} D \hookrightarrow \quad & \lambda s_1 : [\![\mathsf{sp}_1]\!].\ \lambda s_2 : [\![\mathsf{sp}_2]\!]. \\ & s_1\ (\lambda x, z : \mathsf{El}\ \iota.\ \lambda \ell_3 : [\![P(x, f(x))]\!].\ \lambda \ell_1 : [\![\neg Q(x)]\!]. \\ & s_2\ (\lambda y : \mathsf{El}\ \iota.\ \lambda \ell_2 : [\![c = y]\!]. \\ & \quad C\ x\ y\ z\ \ell_1\ \ell_2\ \ell_3)) \end{aligned}$$

AVATAR INFERENCES ENCODED

Definition: just a Dedukti definition

$$\mathsf{sp}_1 : \mathsf{Prop}$$
$$\mathsf{sp}_1 \hookrightarrow \forall x, y : \mathsf{El}\ \iota.\ |\neg P(x, f(y))| \implies |\neg\neg Q(y)| \implies \bot$$

Split: unpack with variable renaming, apply vars and literals to premise.

$$C : \Pi x, y, z : \mathsf{El}\ \iota.\ [\![\neg Q(z)]\!] \to [\![c = y]\!] \to [\![P(x, f(z))]\!] \to \mathsf{Prf}\ \bot.$$

Prf(c=Y) -> Prf(false)

$$D : [\![\mathsf{sp}_1]\!] \to [\![\mathsf{sp}_2]\!] \to \mathsf{Prf}\ \bot:$$

$$\begin{aligned} D \hookrightarrow\quad & \lambda s_1 : [\![\mathsf{sp}_1]\!].\ \lambda s_2 : [\![\mathsf{sp}_2]\!]. \\ & s_1\ (\lambda x, z : \mathsf{El}\ \iota.\ \lambda \ell_3 : [\![P(x, f(x))]\!].\ \lambda \ell_1 : [\![\neg Q(x)]\!]. \\ & s_2\ (\lambda y : \mathsf{El}\ \iota.\ \lambda \ell_2 : [\![c = y]\!]. \\ & \quad C\ x\ y\ z\ \ell_1\ \ell_2\ \ell_3)) \end{aligned}$$

Avatar clause: clauses tagged with split sets need to be handled (bound and applied to parents in the derivation)

$$C \leftarrow \mathsf{sp}_i \wedge \neg \mathsf{sp}_j$$

$$[\![\neg \mathsf{sp}_i]\!] \to [\![\neg\neg \mathsf{sp}_j]\!] \to C.$$

AVATAR INFERENCES ENCODED

Definition: just a Dedukti definition

$$\mathsf{sp}_1 : \mathsf{Prop}$$

$$\mathsf{sp}_1 \hookrightarrow \forall x, y : \mathsf{El}\ \iota.\ |\neg P(x, f(y))| \implies |\neg\neg Q(y)| \implies \bot$$

Split: unpack with variable renaming, apply vars and literals to premise.

$$C : \Pi x, y, z : \mathsf{El}\ \iota.\ [\![\neg Q(z)]\!] \to [\![c = y]\!] \to [\![P(x, f(z))]\!] \to \mathsf{Prf}\ \bot.$$

Prf(c=y) -> Prf(false)

$$D : [\![\mathsf{sp}_1]\!] \to [\![\mathsf{sp}_2]\!] \to \mathsf{Prf}\ \bot:$$

$$\begin{aligned} D \hookrightarrow\ & \lambda s_1 : [\![\mathsf{sp}_1]\!].\ \lambda s_2 : [\![\mathsf{sp}_2]\!]. \\ & s_1\ (\lambda x, z : \mathsf{El}\ \iota.\ \lambda \ell_3 : [\![P(x, f(x))]\!].\ \lambda \ell_1 : [\![\neg Q(x)]\!]. \\ & s_2\ (\lambda y : \mathsf{El}\ \iota.\ \lambda \ell_2 : [\![c = y]\!]. \\ & \quad C\ x\ y\ z\ \ell_1\ \ell_2\ \ell_3)) \end{aligned}$$

Avatar clause: clauses tagged with split sets need to be handled (bound and applied to parents in the derivation)

$$C \leftarrow \mathsf{sp}_i \wedge \neg \mathsf{sp}_j$$

$$[\![\neg \mathsf{sp}_i]\!] \to [\![\neg\neg \mathsf{sp}_j]\!] \to C.$$

Component: morally the id function

AVATAR INFERENCES ENCODED

Definition: just a Dedukti definition

$$\mathsf{sp}_1 : \mathsf{Prop}$$
$$\mathsf{sp}_1 \hookrightarrow \forall x, y : \mathsf{El}\ \iota.\ |\neg P(x, f(y))| \implies |\neg\neg Q(y)| \implies \bot$$

Split: unpack with variable renaming, apply vars and literals to premise.

$$C : \Pi x, y, z : \mathsf{El}\ \iota.\ [\![\neg Q(z)]\!] \to [\![c = y]\!] \to [\![P(x, f(z))]\!] \to \mathsf{Prf}\ \bot.$$

Prf(c=Y) -> Prf(false)

$$D : [\![\mathsf{sp}_1]\!] \to [\![\mathsf{sp}_2]\!] \to \mathsf{Prf}\ \bot:$$

$$\begin{aligned} D \hookrightarrow\ & \lambda s_1 : [\![\mathsf{sp}_1]\!].\ \lambda s_2 : [\![\mathsf{sp}_2]\!]. \\ & s_1\ (\lambda x, z : \mathsf{El}\ \iota.\ \lambda \ell_3 : [\![P(x, f(x))]\!].\ \lambda \ell_1 : [\![\neg Q(x)]\!]. \\ & s_2\ (\lambda y : \mathsf{El}\ \iota.\ \lambda \ell_2 : [\![c = y]\!]. \\ & \quad C\ x\ y\ z\ \ell_1\ \ell_2\ \ell_3)) \end{aligned}$$

Avatar clause: clauses tagged with split sets need to be handled (bound and applied to parents in the derivation)

$$C \leftarrow \mathsf{sp}_i \wedge \neg \mathsf{sp}_j$$
$$[\![\neg \mathsf{sp}_i]\!] \to [\![\neg\neg \mathsf{sp}_j]\!] \to C.$$

Component: morally the id function

Contradiction: just the premise

AVATAR INFERENCES ENCODED

Definition: just a Dedukti definition

$$\mathsf{sp}_1 : \mathsf{Prop}$$
$$\mathsf{sp}_1 \hookrightarrow \forall x, y : \mathsf{El}\ \iota.\ |\neg P(x, f(y))| \implies |\neg\neg Q(y)| \implies \bot$$

Split: unpack with variable renaming, apply vars and literals to premise.

$$C : \Pi x, y, z : \mathsf{El}\ \iota.\ [\![\neg Q(z)]\!] \to [\![c = y]\!] \to [\![P(x, f(z))]\!] \to \mathsf{Prf}\ \bot.$$

Prf(c=Y) -> Prf(false)

$$D : [\![\mathsf{sp}_1]\!] \to [\![\mathsf{sp}_2]\!] \to \mathsf{Prf}\ \bot:$$

$$\begin{aligned} D \hookrightarrow\ & \lambda s_1 : [\![\mathsf{sp}_1]\!].\ \lambda s_2 : [\![\mathsf{sp}_2]\!]. \\ & s_1\ (\lambda x, z : \mathsf{El}\ \iota.\ \lambda \ell_3 : [\![P(x, f(x))]\!].\ \lambda \ell_1 : [\![\neg Q(x)]\!]. \\ & s_2\ (\lambda y : \mathsf{El}\ \iota.\ \lambda \ell_2 : [\![c = y]\!]. \\ & \quad C\ x\ y\ z\ \ell_1\ \ell_2\ \ell_3)) \end{aligned}$$

Avatar clause: clauses tagged with split sets need to be handled (bound and applied to parents in the derivation)

$$C \leftarrow \mathsf{sp}_i \wedge \neg \mathsf{sp}_j$$
$$[\![\neg \mathsf{sp}_i]\!] \to [\![\neg\neg \mathsf{sp}_j]\!] \to C.$$

Component: morally the id function

Contradiction: just the premise

Refutation: involved, see next slide.

AVATAR REFUTATION

AVATAR REFUTATION

- Vampire ships with a copy of MiniSAT that does not emit proofs.

AVATAR REFUTATION

- Vampire ships with a copy of MiniSAT that does not emit proofs.
- New SAT solver added: **CaDiCaL** -> emits DRAT proofs, but only the reverse unit propagation **(RUP) proofs** are used

AVATAR REFUTATION

CaDiCaL computes again just the part where MiniSAT succeeded, to get the RUP proof out

- Vampire ships with a copy of MiniSAT that does not emit proofs.
- New SAT solver added: **CaDiCaL** -> emits DRAT proofs, but only the reverse unit propagation **(RUP) proofs** are used
- – proof_extra: intermediate SAT clauses

AVATAR REFUTATION

CaDiCaL computes again just the part where MiniSAT succeeded, to get the RUP proof out

- Vampire ships with a copy of MiniSAT that does not emit proofs.
- New SAT solver added: **CaDiCaL** -> emits DRAT proofs, but only the reverse unit propagation **(RUP) proofs** are used
- – proof_extra: intermediate SAT clauses
- re-play RUP steps (while proof printing) and encode them in Dedukti (morally a chain of resolutions)

AVATAR REFUTATION

CaDiCaL computes again just the part where MiniSAT succeeded, to get the RUP proof out

- Vampire ships with a copy of MiniSAT that does not emit proofs.
- New SAT solver added: **CaDiCaL** -> emits DRAT proofs, but only the reverse unit propagation **(RUP) proofs** are used
- – proof_extra: intermediate SAT clauses
- re-play RUP steps (while proof printing) and encode them in Dedukti (morally a chain of resolutions)
- must end in **Prf(false)**

WHAT DO WE TRUST

WHAT DO WE TRUST

- Manually check:

WHAT DO WE TRUST

- Manually check:
    - ★ (Correct standard encoding of FOL in Dedukti.)

WHAT DO WE TRUST

- Manually check:
  - ★ (Correct standard encoding of FOL in Dedukti.)
  - ★ Correct encoding of the problem axioms and negated conjecture.

WHAT DO WE TRUST

- Manually check:
  - ★ (Correct standard encoding of FOL in Dedukti.)
  - ★ Correct encoding of the problem axioms and negated conjecture.
  - ★ All symbols after the axioms are defined (using :=).

WHAT DO WE TRUST

- Manually check:
  - ★ (Correct standard encoding of FOL in Dedukti.)
  - ★ Correct encoding of the problem axioms and negated conjecture.
  - ★ All symbols after the axioms are defined (using :=).
  - ★ No “sorry”s.

WHAT DO WE TRUST

- Manually check:
  - ★ (Correct standard encoding of FOL in Dedukti.)
  - ★ Correct encoding of the problem axioms and negated conjecture.
  - ★ All symbols after the axioms are defined (using :=).
  - ★ No “sorry”s.
  - ★ No rewrite rules after the encoding of FOL.

WHAT DO WE TRUST

- Manually check:
  - ★ (Correct standard encoding of FOL in Dedukti.)
  - ★ Correct encoding of the problem axioms and negated conjecture.
  - ★ All symbols after the axioms are defined (using :=).
  - ★ No “sorry”s.
  - ★ No rewrite rules after the encoding of FOL.
- Trust:

WHAT DO WE TRUST

- Manually check:
  - ★ (Correct standard encoding of FOL in Dedukti.)
  - ★ Correct encoding of the problem axioms and negated conjecture.
  - ★ All symbols after the axioms are defined (using :=).
  - ★ No “sorry”s.
  - ★ No rewrite rules after the encoding of FOL.
- Trust:
  - ★ Dedukti type checker (dk check).

PROOF EXTRA

PROOF EXTRA

- **Minimise** what needs to be **carried around** during proof search:

PROOF EXTRA

- **Minimise** what needs to be **carried around** during proof search:
    - Resolution: selected literals

PROOF EXTRA

- **Minimise** what needs to be **carried around** during proof search:
  - Resolution: selected literals
  - Superposition: selected literals, which side of the equation, rewritten term

PROOF EXTRA

- **Minimise** what needs to be **carried around** during proof search:
  - Resolution: selected literals
  - Superposition: selected literals, which side of the equation, rewritten term
  - Subsumption resolution: selected literal

PROOF EXTRA

- **Minimise** what needs to be **carried around** during proof search:
  - Resolution: selected literals
  - Superposition: selected literals, which side of the equation, rewritten term
  - Subsumption resolution: selected literal
  - AVATAR: One final call to CaDiCaL to record intermediate clauses for RUP proofs

PROOF EXTRA

- **Minimise** what needs to be **carried around** during proof search:
  - Resolution: selected literals
  - Superposition: selected literals, which side of the equation, rewritten term
  - Subsumption resolution: selected literal
  - AVATAR: One final call to CaDiCaL to record intermediate clauses for RUP proofs
- Some things are re-computed at proof-output stage, but **only** for the steps that actually appear in the proof:

PROOF EXTRA

- **Minimise** what needs to be **carried around** during proof search:
  - Resolution: selected literals
  - Superposition: selected literals, which side of the equation, rewritten term
  - Subsumption resolution: selected literal
  - AVATAR: One final call to CaDiCaL to record intermediate clauses for RUP proofs
- Some things are re-computed at proof-output stage, but **only** for the steps that actually appear in the proof:
  - Substitution (superposition, resolution, subsumption resolution, avatar split clauses …)

PROOF EXTRA

- **Minimise** what needs to be **carried around** during proof search:
  - Resolution: selected literals
  - Superposition: selected literals, which side of the equation, rewritten term
  - Subsumption resolution: selected literal
  - AVATAR: One final call to CaDiCaL to record intermediate clauses for RUP proofs
- Some things are re-computed at proof-output stage, but **only** for the steps that actually appear in the proof:
  - Substitution (superposition, resolution, subsumption resolution, avatar split clauses …)
  - AVATAR: RUP steps for refutation

EXPERIMENTS

EXPERIMENTS

- Ran on **TPTP 9.0.0** in CNF, FOF, TF0 and TF1 fragments (no satisfiable problems or arithmetic).

EXPERIMENTS

- Ran on **TPTP 9.0.0** in CNF, FOF, TF0 and TF1 fragments (no satisfiable problems or arithmetic).
- Vampire ran on default strategy (replacing nondeterministic LRS with a stable one)

EXPERIMENTS

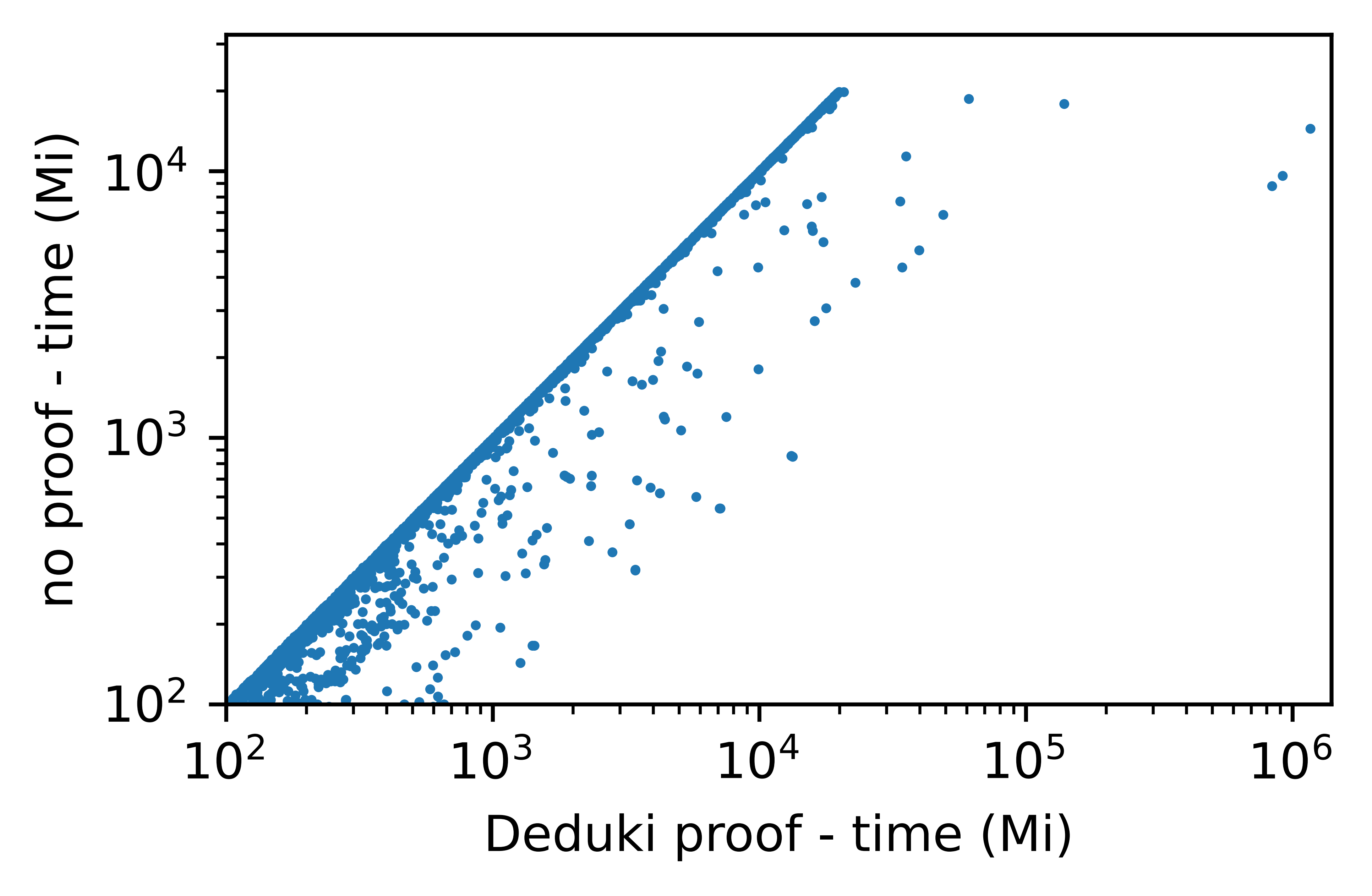
- Ran on **TPTP 9.0.0** in CNF, FOF, TF0 and TF1 fragments (no satisfiable problems or arithmetic).
- Vampire ran on default strategy (replacing nondeterministic LRS with a stable one)
- Ran without checking AVATAR refutation proofs (re-running benchmarks now again with recent AVATAR proof development)

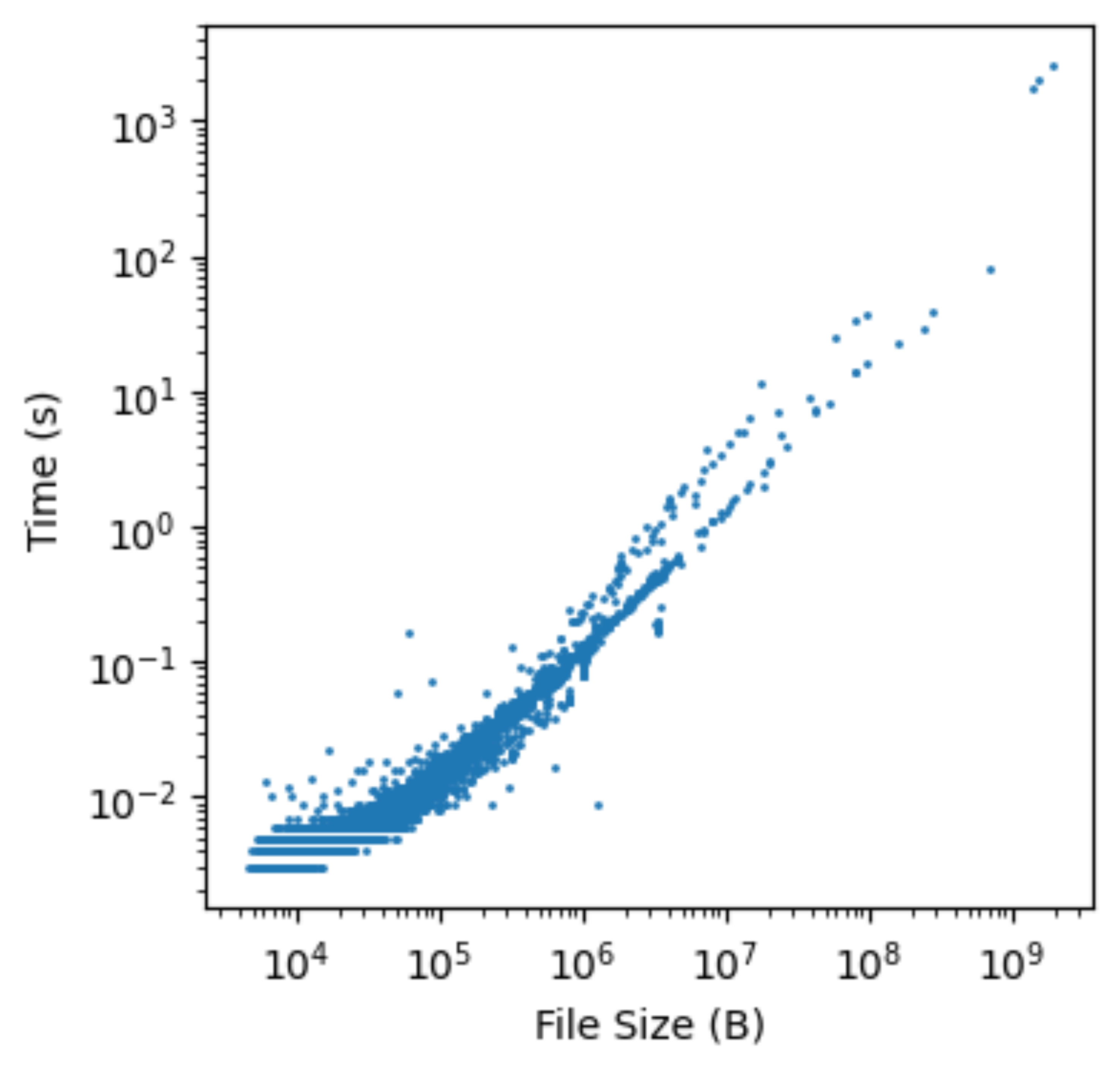
EXPERIMENTS

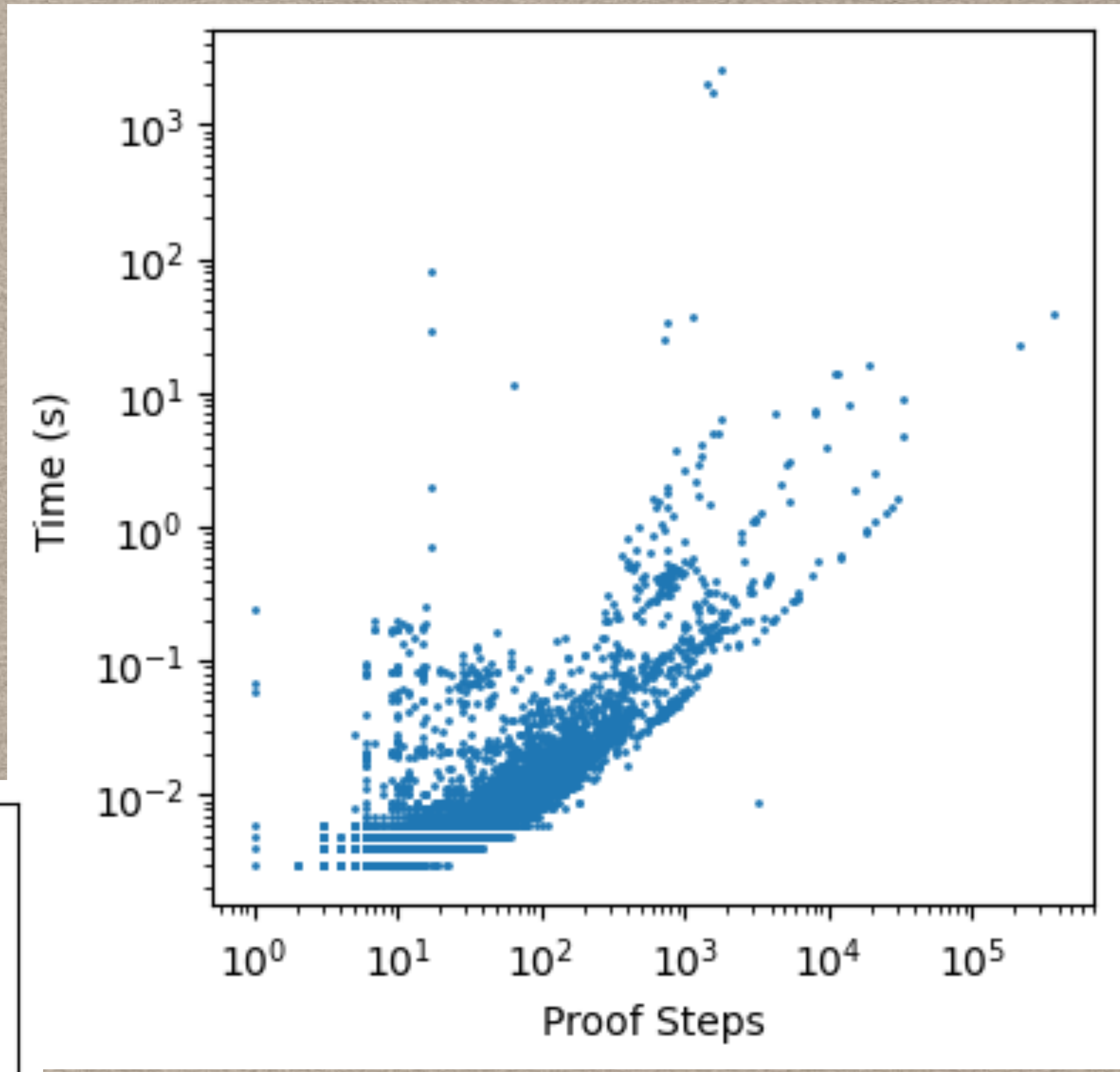
- Ran on **TPTP 9.0.0** in CNF, FOF, TF0 and TF1 fragments (no satisfiable problems or arithmetic).
- Vampire ran on default strategy (replacing nondeterministic LRS with a stable one)
- Ran without checking AVATAR refutation proofs (re-running benchmarks now again with recent AVATAR proof development)
- Instruction limit 20 000 Mi: –proof_extra manages to prove just 5 problems fewer than the total 7839 in the default mode.

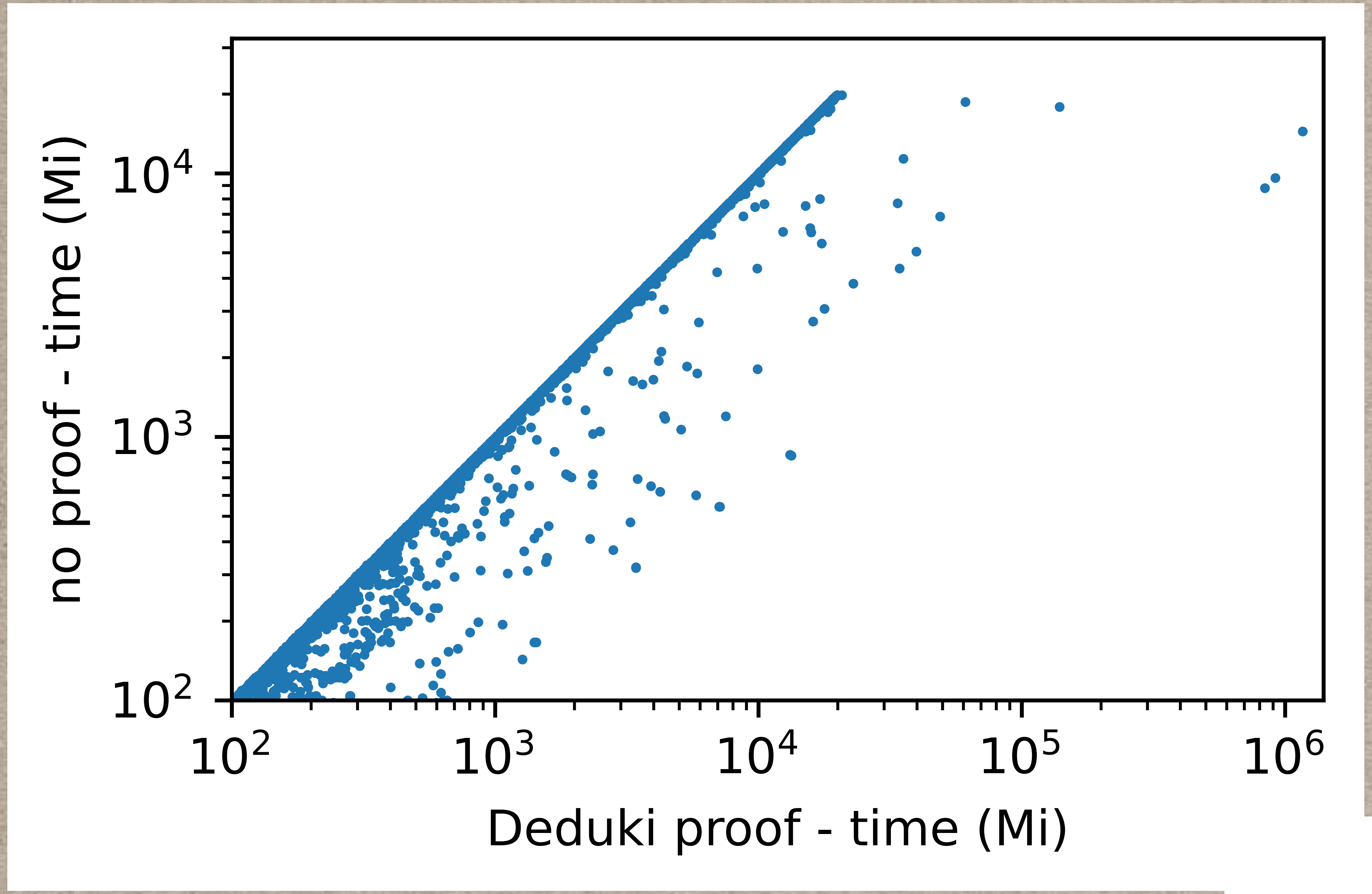
EXPERIMENTS

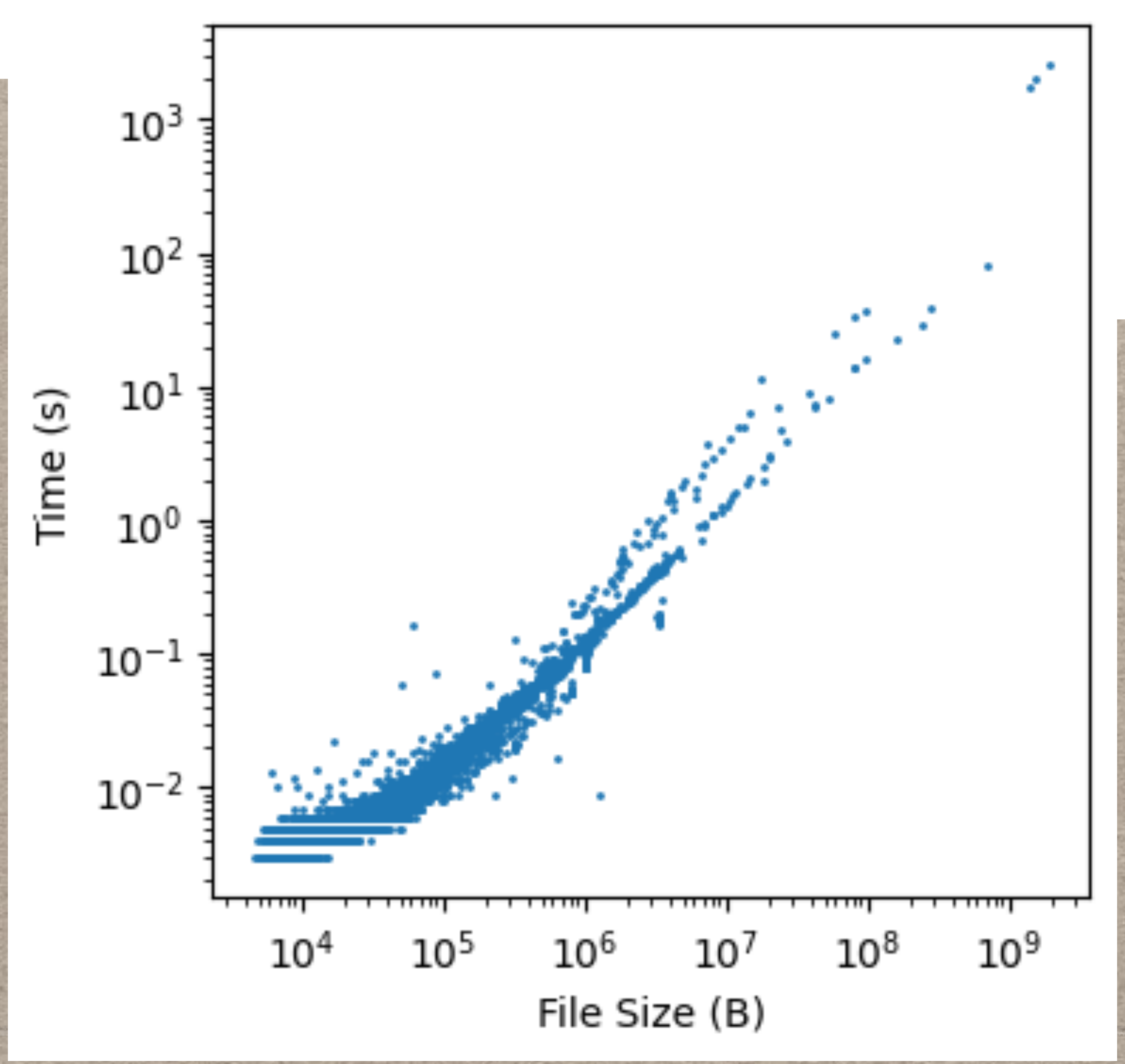
- Ran on **TPTP 9.0.0** in CNF, FOF, TF0 and TF1 fragments (no satisfiable problems or arithmetic).
- Vampire ran on default strategy (replacing nondeterministic LRS with a stable one)
- Ran without checking AVATAR refutation proofs (re-running benchmarks now again with recent AVATAR proof development)
- Instruction limit 20 000 Mi: –proof_extra manages to prove just 5 problems fewer than the total 7839 in the default mode.
- **All proofs were successfully checked by Dedukti** (increasing stack was necessary)

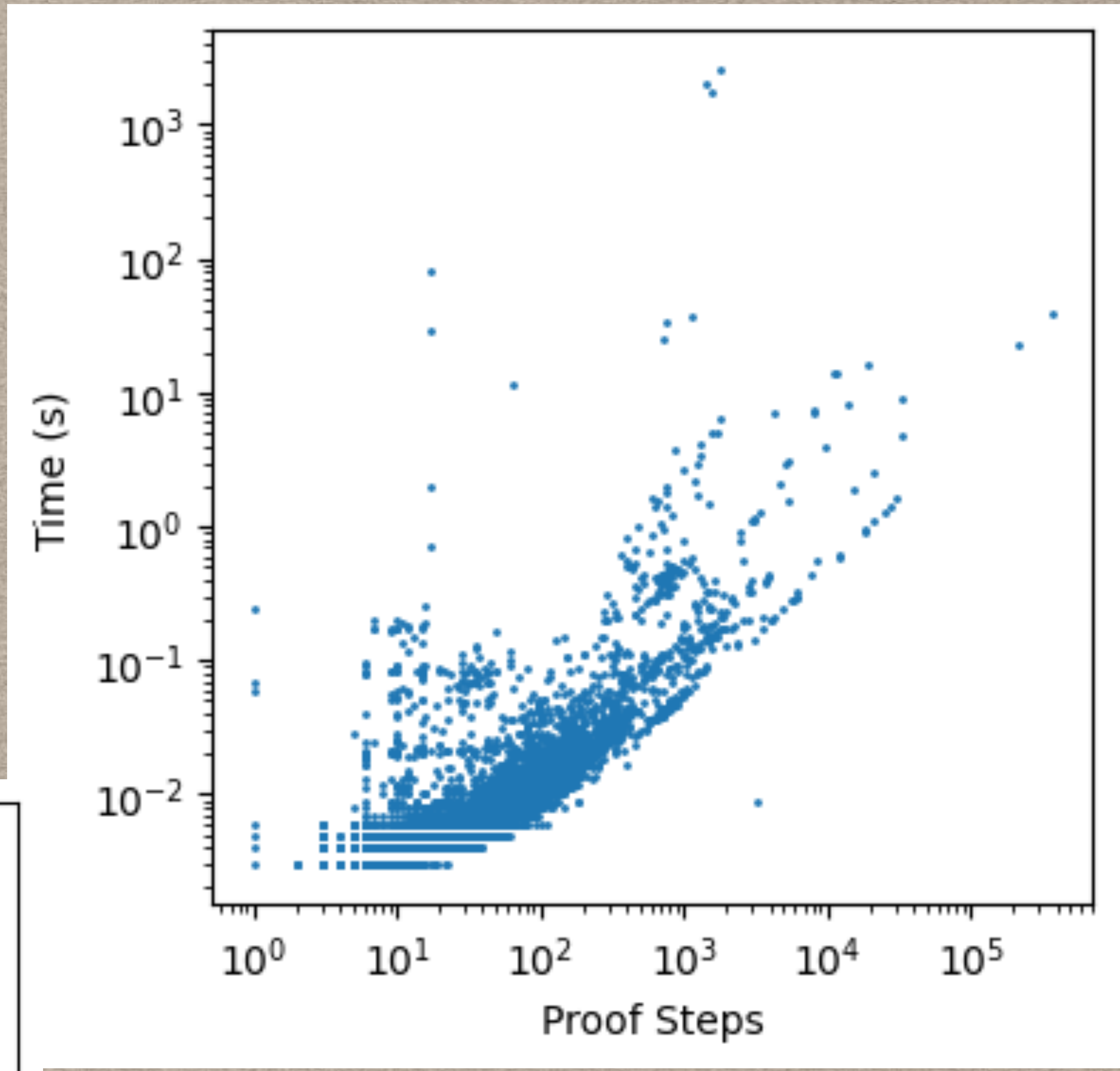

no proof - time (Mi)
Deduki proof - time (Mi)


Time (s)
File Size (B)


Time (s)
Proof Steps







Proof checking (dk check) time

Median time: 0.006s

Average time: 0.881s

Max time: 2567s (42min)

CONSTRUCTIVE PROOFS

CONSTRUCTIVE PROOFS

- Inferences (so far) are constructive : We don’t yet use classical axioms.

CONSTRUCTIVE PROOFS

- Inferences (so far) are constructive : We don’t yet use classical axioms.
- Vampire outputs the proof of double negation:
  negated conjecture –> false

CONSTRUCTIVE PROOFS

- Inferences (so far) are constructive : We don’t yet use classical axioms.
- Vampire outputs the proof of double negation:
  negated conjecture –> false
- The last step to go from double negation to asserting conjecture is classical, but we actually do not explicitly do that.

Note: Due to double-negation translation, it is always possible to have a classical proof, that is constructive all but for the one (last) step. This is an illustration of this fact.

SPECULATIONS ON CURRENTLY UNSUPPORTED INFERENCES

SPECULATIONS ON CURRENTLY UNSUPPORTED INFERENCES

- **Many classification rules are straightforward**, but we need to traverse the formula again and output more details in –proof_extra

SPECULATIONS ON CURRENTLY UNSUPPORTED INFERENCES

- **Many classification rules are straightforward**, but we need to traverse the formula again and output more details in –proof_extra
- **Skolemization**: there is no free lunch.

SPECULATIONS ON CURRENTLY UNSUPPORTED INFERENCES

- **Many classification rules are straightforward**, but we need to traverse the formula again and output more details in –proof_extra
- **Skolemization**: there is no free lunch.
  - Posing an axiom of choice in full:

```
choose : (a : Set -> (r : (El a -> Prop) -> Prf (exists a r) -> El a)).
axiom_of_choice : (a : Set -> (r : (El a -> Prop) -> (tex : Prf (exists a r)) -> Prf (r (choose a r tex)))).
```

SPECULATIONS ON CURRENTLY UNSUPPORTED INFERENCES

- **Many classification rules are straightforward**, but we need to traverse the formula again and output more details in –proof_extra
- **Skolemization**: there is no free lunch.
  - Posing an axiom of choice in full:
    choose : (a : Set -> (r : (El a -> Prop) -> Prf (exists a r) -> El a)).
    axiom_of_choice : (a : Set -> (r : (El a -> Prop) -> (tex : Prf (exists a r)) -> Prf (r (choose a r tex)))).
  - Posing axioms that are instances choice needed for skolemization steps in the proof

SPECULATIONS ON CURRENTLY UNSUPPORTED INFERENCES

- **Many classification rules are straightforward**, but we need to traverse the formula again and output more details in –proof_extra
- **Skolemization**: there is no free lunch.
  - Posing an axiom of choice in full:
    choose : (a : Set -> (r : (El a -> Prop) -> Prf (exists a r) -> El a)).
    axiom_of_choice : (a : Set -> (r : (El a -> Prop) -> (tex : Prf (exists a r)) -> Prf (r (choose a r tex)))).
  - Posing axioms that are instances choice needed for skolemization steps in the proof
- **Polarity flip**: exploiting Dedukti definitions
  def polarity_flip (p : Prop -> Prop) := (not p) and proceeding with polarity_flip(p).

QUESTIONS?

vampire $problem -p dedukti - - proof_extra full

| egrep -v ^%

| dk check /dev/stdin