A Toolbox for Mechanised First-Order Logic

Johannes Hostert, Mark Koch, and Dominik Kirst

The Coq Workshop
July 1, 2021

SAARLAND UNIVERSITY
COMPUTER SCIENCE
SIC Saarland Informatics Campus

Bad Example from Kirst and Hermes (2021)

```
x, y, a, b :  term'
B :  list form'
HB :  (∃ $0 ≡' x`[↑]
          ∧ (∃ (∃ $0 ≡' x`[↑]`[↑]`[↑]
                   ∧ (∃ $0 ≡' x`[↑]`[↑]`[↑]`[↑]
                         ∧ (∀ $0 ∈' ↑ 2 ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
               ∧ (∀ $0 ∈' a`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
      ∧ (∀ $0 ∈' y`[↑] ↔ (∃ $0 ∈' a`[↑]`[↑] ∧ ↑ 0 ∈' $0))
      :: (∃ (∃ $0 ≡' x`[↑]`[↑]
                 ∧ (∃ (∃ $0 ≡' x`[↑]`[↑]`[↑]`[↑]
                         ∧ (∃ $0 ≡' x`[↑]`[↑]`[↑]`[↑]`[↑]
                              ∧ (∀ $0 ∈' ↑ 2 ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
                      ∧ (∀ $0 ∈' ↑ 2 ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
             ∧ (∀ $0 ∈' y`[↑]`[↑] ↔ (∃ $0 ∈' ↑ 1 ∧ ↑ 0 ∈' $0)))
          ∧ (∃ $0 ≡' ↑ n ∧ y`[↑] ∈' $0) :: x ∈' $n :: A <<= B
z :  term'
H :  ((∃ $0 ≡' x`[↑]
          ∧ (∃ $0 ≡' x`[↑]`[↑]
               ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
      ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
     ⊢ ∀ $0 ∈' z`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' x`[↑]
c :  term'
H1 :  (c ∈' a ∧ b ∈' c
       :: (∃ $0 ≡' x`[↑]
              ∧ (∃ $0 ≡' x`[↑]`[↑]
                    ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
           ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
      ⊢ c ≡' x ∨ c ≡' z
H2 :  (c ∈' a ∧ b ∈' c
       :: (∃ $0 ≡' x`[↑]
              ∧ (∃ $0 ≡' x`[↑]`[↑]
                    ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
           ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
      ⊢ b ∈' c
```

```
(1/1)
(c ≡' x
 :: c ∈' a ∧ b ∈' c
    :: (∃ $0 ≡' x`[↑]
           ∧ (∃ $0 ≡' x`[↑]`[↑]
                 ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
        ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
⊢ b ∈' x ∨ b ≡' x
```

Bad Example from Kirst and Hermes (2021)

```
Lemma prv_to_min_inductive A n :
  minZFeq' <<= A -> A ⊢ rm_const_fm (inductive $n) -> A ⊢ is_inductive $n.
Proof.
  cbn. intros HA HI. apply CI.
  - apply CE1 in HI. use_exists' HI x. clear HI.
    apply (ExI x). cbn. assert1 H. apply CE in H as [H1 H2]. apply CI; trivial.
    change (∃ $0 ≡' ↑ n ∧ x `[↑] ∈' $0) with (∃ $0 ≡' $n`[↑] ∧ x`[↑] ∈' $0) in H2.
    now simpl_ex_in H2.
  - apply CE2 in HI. prv_all' x. apply (AllE x) in HI. cbn in HI. simpl_ex_in HI.
    change (∃ $0 ≡' ↑ n ∧ x`[↑] ∈' $0) with (∃ $0 ≡' $n`[↑] ∧ x`[↑] ∈' $0) in HI.
    simpl_ex_in HI. rewrite imps in *. use_exists' HI y. clear HI.
    assert1 H. apply (ExI y). cbn. subsimpl. apply CI.
    + apply CE1 in H. use_exists' H a. clear H. assert1 H. apply CE in H as [H1 H2].
      simpl_ex_in H1. prv_all' b. apply (AllE b) in H2. cbn in H2. subsimpl_in H2.
      eapply iff_equiv; try apply H2; try tauto.
      intros B HB. clear H2. eapply Weak in H1; try apply HB. split; intros H2.
      * use_exists' H1 z. clear H1. assert1 H. apply CE in H as [H H'].
        apply prv_ex_eq in H; try rewrite <- HB; auto. cbn in H. subsimpl_in H.
        apply prv_ex_eq in H; try rewrite <- HB; auto. cbn in H. subsimpl_in H.
        eapply Weak in H2. apply (DE H2). 3: auto.
        -- apply (ExI x). cbn. subsimpl. apply CI; auto. apply (AllE x) in H'. cbn in H'. subsimpl_in H'.
           apply CE2 in H'. eapply IE. apply (Weak H'); auto. apply DI1. apply minZF_refl. rewrite <- HB. auto 6.
        -- apply (ExI z). cbn. subsimpl. apply CI.
           ++ apply (AllE z) in H'. cbn in H'. subsimpl_in H'. apply CE2 in H'. eapply IE.
              apply (Weak H'); auto. apply DI2. apply minZF_refl. rewrite <- HB. auto 6.
           ++ apply (AllE b) in H. cbn in H. subsimpl_in H. apply CE2 in H. eapply IE.
              apply (Weak H); auto. apply DI2. auto.
      * use_exists' H1 z. clear H1. assert1 H. apply CE in H as [H H'].
        apply prv_ex_eq in H; try rewrite <- HB; auto. cbn in H. subsimpl_in H.
        apply prv_ex_eq in H; try rewrite <- HB; auto. cbn in H. subsimpl_in H.
        eapply Weak in H2. use_exists' H2 c. 2: auto. clear H2. assert1 H1. apply CE in H1 as [H1 H2].
        apply (AllE c) in H'. cbn in H'. subsimpl_in H'. apply CE1 in H'. eapply Weak in H'.
        apply (IE H') in H1. 2: auto. clear H'. apply (DE H1).
        -- apply DI1. eapply minZF_elem. rewrite <- HB, HA. auto 8. 3: apply (Weak H2); auto.
           2: auto. apply minZF_refl. rewrite <- HB, HA. auto 8.
        -- apply DI2. apply (AllE b) in H. cbn in H. subsimpl_in H. apply CE1 in H. eapply DE'.
           eapply IE. apply (Weak H). auto. eapply minZF_elem. rewrite <- HB, HA. auto 8.
           3: apply (Weak H2); auto. 2: auto. apply minZF_refl. rewrite <- HB, HA. auto 8.
    + apply CE2 in H. change (∃ $0 ≡' ↑ n ∧ y`[↑] ∈' $0) with (∃ $0 ≡' $n`[↑] ∧ y`[↑] ∈' $0) in H.
      now simpl_ex_in H.
Qed.
```

Towards a Coq Library of First-Order Logic

Merge of developments presented at ITP/CPP/IJCAR/LFCS, including:

Towards a Coq Library of First-Order Logic

Merge of developments presented at ITP/CPP/IJCAR/LFCS, including:

Completeness:

- Several deduction systems and semantics
- Constructive analysis (relation to MP and WKL, exploding models)

Towards a Coq Library of First-Order Logic

Merge of developments presented at ITP/CPP/IJCAR/LFCS, including:

Completeness:

- Several deduction systems and semantics
- Constructive analysis (relation to MP and WKL, exploding models)

Incompleteness:

- (Fragments and extensions of) Peano arithmetic and ZF set theory
- Computational approach, no Gödel/Rosser sentences

Towards a Coq Library of First-Order Logic

Merge of developments presented at ITP/CPP/IJCAR/LFCS, including:

Completeness:

- Several deduction systems and semantics
- Constructive analysis (relation to MP and WKL, exploding models)

Incompleteness:

- (Fragments and extensions of) Peano arithmetic and ZF set theory
- Computational approach, no Gödel/Rosser sentences

Undecidability:

- Synthetic reductions from PCP and H10
- Validity, satisfiability, provability, finite satisfiability, PA, ZF

Towards a Coq Library of First-Order Logic

Merge of developments presented at ITP/CPP/IJCAR/LFCS, including:

Completeness:

- Several deduction systems and semantics
- Constructive analysis (relation to MP and WKL, exploding models)

Incompleteness:

- (Fragments and extensions of) Peano arithmetic and ZF set theory
- Computational approach, no Gödel/Rosser sentences

Undecidability:

- Synthetic reductions from PCP and H10
- Validity, satisfiability, provability, finite satisfiability, PA, ZF

Part of the Coq Library of Undecidability Proofs (Forster et al. (2020))

Representing First-Order Logic in Coq

Meta-theoretic results require a deep embedding:

Representing First-Order Logic in Coq

Meta-theoretic results require a deep embedding:

- De Bruijn encoding of syntax, parallel substitution

Representing First-Order Logic in Coq

Meta-theoretic results require a deep embedding:

- De Bruijn encoding of syntax, parallel substitution
- Model-theoretic semantics by embedding into Coq's logic

Representing First-Order Logic in Coq

Meta-theoretic results require a deep embedding:

- De Bruijn encoding of syntax, parallel substitution
- Model-theoretic semantics by embedding into Coq's logic
- Inductive predicates to represent deduction systems

Representing First-Order Logic in Coq

Meta-theoretic results require a deep embedding:

- De Bruijn encoding of syntax, parallel substitution
- Model-theoretic semantics by embedding into Coq's logic
- Inductive predicates to represent deduction systems

Nice meta-theory but challenging for more concrete results:

Representing First-Order Logic in Coq

Meta-theoretic results require a deep embedding:

- De Bruijn encoding of syntax, parallel substitution
- Model-theoretic semantics by embedding into Coq's logic
- Inductive predicates to represent deduction systems

Nice meta-theory but challenging for more concrete results:

- Writing down long first-order formulas φ in de Bruijn is unpleasant

Representing First-Order Logic in Coq

Meta-theoretic results require a deep embedding:

- De Bruijn encoding of syntax, parallel substitution
- Model-theoretic semantics by embedding into Coq's logic
- Inductive predicates to represent deduction systems

Nice meta-theory but challenging for more concrete results:

- Writing down long first-order formulas φ in de Bruijn is unpleasant
- Semantic derivations $\Gamma \vDash \varphi$ compute to statements in Coq but might still expose syntax e.g. via axiom schemes present in Γ

Representing First-Order Logic in Coq

Meta-theoretic results require a deep embedding:

- De Bruijn encoding of syntax, parallel substitution
- Model-theoretic semantics by embedding into Coq's logic
- Inductive predicates to represent deduction systems

Nice meta-theory but challenging for more concrete results:

- Writing down long first-order formulas φ in de Bruijn is unpleasant
- Semantic derivations $\Gamma \vDash \varphi$ compute to statements in Coq but might still expose syntax e.g. via axiom schemes present in Γ
- Deductions $\Gamma \vdash \varphi$ need to be done by hand, including substitution treatment for quantifier rules

Representing First-Order Logic in Coq

Meta-theoretic results require a deep embedding:

- De Bruijn encoding of syntax, parallel substitution
- Model-theoretic semantics by embedding into Coq's logic
- Inductive predicates to represent deduction systems

Nice meta-theory but challenging for more concrete results:

- Writing down long first-order formulas φ in de Bruijn is unpleasant
- Semantic derivations $\Gamma \vDash \varphi$ compute to statements in Coq but might still expose syntax e.g. via axiom schemes present in Γ
- Deductions $\Gamma \vdash \varphi$ need to be done by hand, including substitution treatment for quantifier rules

Partial solutions:

Representing First-Order Logic in Coq

Meta-theoretic results require a deep embedding:

- De Bruijn encoding of syntax, parallel substitution
- Model-theoretic semantics by embedding into Coq's logic
- Inductive predicates to represent deduction systems

Nice meta-theory but challenging for more concrete results:

- Writing down long first-order formulas φ in de Bruijn is unpleasant
- Semantic derivations $\Gamma \vDash \varphi$ compute to statements in Coq but might still expose syntax e.g. via axiom schemes present in Γ
- Deductions $\Gamma \vdash \varphi$ need to be done by hand, including substitution treatment for quantifier rules

Partial solutions:

- Come up with compromises (e.g. Laurent (2021))

Representing First-Order Logic in Coq

Meta-theoretic results require a deep embedding:

- De Bruijn encoding of syntax, parallel substitution
- Model-theoretic semantics by embedding into Coq's logic
- Inductive predicates to represent deduction systems

Nice meta-theory but challenging for more concrete results:

- Writing down long first-order formulas φ in de Bruijn is unpleasant
- Semantic derivations $\Gamma \vDash \varphi$ compute to statements in Coq but might still expose syntax e.g. via axiom schemes present in Γ
- Deductions $\Gamma \vdash \varphi$ need to be done by hand, including substitution treatment for quantifier rules

Partial solutions:

- Come up with compromises (e.g. Laurent (2021))
- Implement tools for each problem (our approach)

DEMO

DEMO Reification

Reification

Reification:

- Extraction of Coq formulas into FOL abstract syntax

Reification

Reification:

- Extraction of Coq formulas into FOL abstract syntax

Approach:

- Recursively match Coq AST for known constructs
- Construct environment beforehand

Reification

Reification:

- Extraction of Coq formulas into FOL abstract syntax

Approach:

- Recursively match Coq AST for known constructs
- Construct environment beforehand
- Use MetaCoq
  - We already had experience with it
  - Already required for other parts of the library
  - Allows deep introspection of Coq AST

Reification

Reification:

- Extraction of Coq formulas into FOL abstract syntax

Approach:

- Recursively match Coq AST for known constructs
- Construct environment beforehand
- Use MetaCoq
  - We already had experience with it
  - Already required for other parts of the library
  - Allows deep introspection of Coq AST
- Reification with MetaCoq already seen in Forster and Kunze (2019)
- Reification of FOL previously worked on by Rech (2020)

Reification

Reification:

- Extraction of Coq formulas into FOL abstract syntax

Approach:

- Recursively match Coq AST for known constructs
- Construct environment beforehand
- Use MetaCoq
  - We already had experience with it
  - Already required for other parts of the library
  - Allows deep introspection of Coq AST
- Reification with MetaCoq already seen in Forster and Kunze (2019)
- Reification of FOL previously worked on by Rech (2020)

Main difficulties:

- Transporting binders ($\forall, \exists$) into FOL
- Building environments

Reification

Reification:

- Extraction of Coq formulas into FOL abstract syntax

Approach:

- Recursively match Coq AST for known constructs
- Construct environment beforehand
- Use MetaCoq
  - We already had experience with it
  - Already required for other parts of the library
  - Allows deep introspection of Coq AST
- Reification with MetaCoq already seen in Forster and Kunze (2019)
- Reification of FOL previously worked on by Rech (2020)

Main difficulties:

- Transporting binders ($\forall, \exists$) into FOL
- Building environments

Analysis

Issues:

- Finding proofs is slow
  - Just looking for the term is way faster
  - Can you profile MetaCoq?

Analysis

Issues:

- Finding proofs is slow
  - Just looking for the term is way faster
  - Can you profile MetaCoq?
- Reification just looks at AST/syntax
  - Can not reify terms hidden behind definitions
  - Requires users to use proper notations and definitions

Analysis

Issues:

- Finding proofs is slow
  - Just looking for the term is way faster
  - Can you profile MetaCoq?
- Reification just looks at AST/syntax
  - Can not reify terms hidden behind definitions
  - Requires users to use proper notations and definitions
- Reification only knows about basic embedding of function/relation symbols
  - Reifying extensional equality does not work
    Framework does not know what eq is represented by in FOL

Extension points

Potential solution for previous problems: Extension points

- User-defined type class instance

Extension points

Potential solution for previous problems: Extension points

- User-defined type class instance

- Can be used to supply additional representations, like
  - Extensional equality
  - FOL terms for higher-order embeddings

Extension points

Potential solution for previous problems: Extension points

- User-defined type class instance
- Can be used to supply additional representations, like
  - Extensional equality
  - FOL terms for higher-order embeddings
- Requires understanding of framework internals

DEMO

Deductive Proofs

Deductive Proofs

```
Lemma prv_to_min_inductive A n :
  minZFeq' <<= A -> A ⊢ rm_const_fm (inductive $n) -> A ⊢ is_inductive $n.
Proof.
  cbn. intros HA HI. apply CI.
  - apply CE1 in HI. use_exists' HI x. clear HI.
    apply (ExI x). cbn. assert1 H. apply CE in H as [H1 H2]. apply CI; trivial.
    change (∃ $0 ≡' ↑ n ∧ x `[↑] ∈' $0) with (∃ $0 ≡' $n`[↑] ∧ x`[↑] ∈' $0) in H2.
    now simpl_ex_in H2.
  - apply CE2 in HI. prv_all' x. apply (AllE x) in HI. cbn in HI. simpl_ex_in HI.
    change (∃ $0 ≡' ↑ n ∧ x`[↑] ∈' $0) with (∃ $0 ≡' $n`[↑] ∧ x`[↑] ∈' $0) in HI.
    simpl_ex_in HI. rewrite imps in *. use_exists' HI y. clear HI.
    assert1 H. apply (ExI y). cbn. subsimpl. apply CI.
    + apply CE1 in H. use_exists' H a. clear H. assert1 H. apply CE in H as [H1 H2].
      simpl_ex_in H1. prv_all' b. apply (AllE b) in H2. cbn in H2. subsimpl_in H2.
      eapply iff_equiv; try apply H2; try tauto.
      intros B HB. clear H2. eapply Weak in H1; try apply HB. split; intros H2.
      * use_exists' H1 z. clear H1. assert1 H. apply CE in H as [H H'].
        apply prv_ex_eq in H; try rewrite <- HB; auto. cbn in H. subsimpl_in H.
        apply prv_ex_eq in H; try rewrite <- HB; auto. cbn in H. subsimpl_in H.
        eapply Weak in H2. apply (DE H2). 3: auto.
        -- apply (ExI x). cbn. subsimpl. apply CI; auto. apply (AllE x) in H'. cbn in H'. subsimpl_in H'.
           apply CE2 in H'. eapply IE. apply (Weak H'); auto. apply DI1. apply minZF_refl. rewrite <- HB. auto 6.
        -- apply (ExI z). cbn. subsimpl. apply CI.
           ++ apply (AllE z) in H'. cbn in H'. subsimpl_in H'. apply CE2 in H'. eapply IE.
              apply (Weak H'); auto. apply DI2. apply minZF_refl. rewrite <- HB. auto 6.
           ++ apply (AllE b) in H. cbn in H. subsimpl_in H. apply CE2 in H. eapply IE.
              apply (Weak H); auto. apply DI2. auto.
      * use_exists' H1 z. clear H1. assert1 H. apply CE in H as [H H'].
        apply prv_ex_eq in H; try rewrite <- HB; auto. cbn in H. subsimpl_in H.
        apply prv_ex_eq in H; try rewrite <- HB; auto. cbn in H. subsimpl_in H.
        eapply Weak in H2. use_exists' H2 c. 2: auto. clear H2. assert1 H1. apply CE in H1 as [H1 H2].
        apply (AllE c) in H'. cbn in H'. subsimpl_in H'. apply CE1 in H'. eapply Weak in H'.
        apply (IE H') in H1. 2: auto. clear H'. apply (DE H1).
        -- apply DI1. eapply minZF_elem. rewrite <- HB, HA. auto 8. 3: apply (Weak H2); auto.
           2: auto. apply minZF_refl. rewrite <- HB, HA. auto 8.
        -- apply DI2. apply (AllE b) in H. cbn in H. subsimpl_in H. apply CE1 in H. eapply DE'.
           eapply IE. apply (Weak H). auto. eapply minZF_elem. rewrite <- HB, HA. auto 8.
           3: apply (Weak H2); auto. 2: auto. apply minZF_refl. rewrite <- HB, HA. auto 8.
    + apply CE2 in H. change (∃ $0 ≡' ↑ n ∧ y`[↑] ∈' $0) with (∃ $0 ≡' $n`[↑] ∧ y`[↑] ∈' $0) in H.
      now simpl_ex_in H.
Qed.
```

Deductive Proofs

```
Lemma prv_to_min_inductive A n :
  minZFeq' <<= A -> A ⊢ rm_const_fm (inductive $n) -> A ⊢ is_inductive $n.
Proof.
  cbn. intros HA HI. apply CI.
  - apply CE1 in HI. use_exists' HI x. clear HI.
    apply (ExI x). cbn. assert1 H. apply CE in H as [H1 H2]. apply CI; trivial.
    change (∃ $0 ≡' ↑ n ∧ x `[↑] ∈' $0) with (∃ $0 ≡' $n`[↑] ∧ x`[↑] ∈' $0) in H2.
    now simpl_ex_in H2.
  - apply CE2 in HI. prv_all' x. apply (AllE x) in HI. cbn in HI. simpl_ex_in HI.
    change (∃ $0 ≡' ↑ n ∧ x`[↑] ∈' $0) with (∃ $0 ≡' $n`[↑] ∧ x`[↑] ∈' $0) in HI.
    simpl_ex_in HI. rewrite imps in *. use_exists' HI y. clear HI.
    assert1 H. apply (ExI y). cbn. subsimpl. apply CI.
    + apply CE1 in H. use_exists' H a. clear H. assert1 H. apply CE in H as [H1 H2].
      simpl_ex_in H1. prv_all' b. apply (AllE b) in H2. cbn in H2. subsimpl_in H2.
      eapply iff_equiv; try apply H2; try tauto.
      intros B HB. clear H2. eapply Weak in H1; try apply HB. split; intros H2.
      * use_exists' H1 z. clear H1. assert1 H. apply CE in H as [H H'].
        apply prv_ex_eq in H; try rewrite <- HB; auto. cbn in H. subsimpl_in H.
        apply prv_ex_eq in H; try rewrite <- HB; auto. cbn in H. subsimpl_in H.
        eapply Weak in H2. apply (DE H2). 3: auto.
        -- apply (ExI x). cbn. subsimpl. apply CI; auto. apply (AllE x) in H'. cbn in H'. subsimpl_in H'.
           apply CE2 in H'. eapply IE. apply (Weak H'); auto. apply DI1. apply minZF_refl. rewrite <- HB. auto 6.
        -- apply (ExI z). cbn. subsimpl. apply CI.
           ++ apply (AllE z) in H'. cbn in H'. subsimpl_in H'. apply CE2 in H'. eapply IE.
              apply (Weak H'); auto. apply DI2. apply minZF_refl. rewrite <- HB. auto 6.
           ++ apply (AllE b) in H. cbn in H. subsimpl_in H. apply CE2 in H. eapply IE.
              apply (Weak H); auto. apply DI2. auto.
      * use_exists' H1 z. clear H1. assert1 H. apply CE in H as [H H'].
        apply prv_ex_eq in H; try rewrite <- HB; auto. cbn in H. subsimpl_in H.
        apply prv_ex_eq in H; try rewrite <- HB; auto. cbn in H. subsimpl_in H.
        eapply Weak in H2. use_exists' H2 c. 2: auto. clear H2. assert1 H1. apply CE in H1 as [H1 H2].
        apply (AllE c) in H'. cbn in H'. subsimpl_in H'. apply CE1 in H'. eapply Weak in H'.
        apply (IE H') in H1. 2: auto. clear H'. apply (DE H1).
        -- apply DI1. eapply minZF_elem. rewrite <- HB, HA. auto 8. 3: apply (Weak H2); auto.
           2: auto. apply minZF_refl. rewrite <- HB, HA. auto 8.
        -- apply DI2. apply (AllE b) in H. cbn in H. subsimpl_in H. apply CE1 in H. eapply DE'.
           eapply IE. apply (Weak H). auto. eapply minZF_elem. rewrite <- HB, HA. auto 8.
           3: apply (Weak H2); auto. 2: auto. apply minZF_refl. rewrite <- HB, HA. auto 8.
    + apply CE2 in H. change (∃ $0 ≡' ↑ n ∧ y`[↑] ∈' $0) with (∃ $0 ≡' $n`[↑] ∧ y`[↑] ∈' $0) in H.
      now simpl_ex_in H.
Qed.
```

Need to explicitly change the goal because of substitutions

Deductive Proofs

```
Lemma prv_to_min_inductive A n :
  minZFeq' <<= A -> A ⊢ rm_const_fm (inductive $n) -> A ⊢ is_inductive $n.
Proof.
  cbn. intros HA HI. apply CI.
  - apply CE1 in HI. use_exists' HI x. clear HI.
    apply (ExI x). cbn. assert1 H. apply CE in H as [H1 H2]. apply CI; trivial.
    change (∃ $0 ≡' ↑ n ∧ x `[↑] ∈' $0) with (∃ $0 ≡' $n`[↑] ∧ x`[↑] ∈' $0) in H2.
    now simpl_ex_in H2.
  - apply CE2 in HI. prv_all' x. apply (AllE x) in HI. cbn in HI. simpl_ex_in HI.
    change (∃ $0 ≡' ↑ n ∧ x`[↑] ∈' $0) with (∃ $0 ≡' $n`[↑] ∧ x`[↑] ∈' $0) in HI.
    simpl_ex_in HI. rewrite imps in *. use_exists' HI y. clear HI.
    assert1 H. apply (ExI y). cbn. subsimpl. apply CI.
    + apply CE1 in H. use_exists' H a. clear H. assert1 H. apply CE in H as [H1 H2].
      simpl_ex_in H1. prv_all' b. apply (AllE b) in H2. cbn in H2. subsimpl_in H2.
      eapply iff_equiv; try apply H2; try tauto.
      intros B HB. clear H2. eapply Weak in H1; try apply HB. split; intros H2.
      * use_exists' H1 z. clear H1. assert1 H. apply CE in H as [H H'].
        apply prv_ex_eq in H; try rewrite <- HB; auto. cbn in H. subsimpl_in H.
        apply prv_ex_eq in H; try rewrite <- HB; auto. cbn in H. subsimpl_in H.
        eapply Weak in H2. apply (DE H2). 3: auto.
        -- apply (ExI x). cbn. subsimpl. apply CI; auto. apply (AllE x) in H'. cbn in H'. subsimpl_in H'.
           apply CE2 in H'. eapply IE. apply (Weak H'); auto. apply DI1. apply minZF_refl. rewrite <- HB. auto 6.
        -- apply (ExI z). cbn. subsimpl. apply CI.
           ++ apply (AllE z) in H'. cbn in H'. subsimpl_in H'. apply CE2 in H'. eapply IE.
              apply (Weak H'); auto. apply DI2. apply minZF_refl. rewrite <- HB. auto 6.
           ++ apply (AllE b) in H. cbn in H. subsimpl_in H. apply CE2 in H. eapply IE.
              apply (Weak H); auto. apply DI2. auto.
      * use_exists' H1 z. clear H1. assert1 H. apply CE in H as [H H'].
        apply prv_ex_eq in H; try rewrite <- HB; auto. cbn in H. subsimpl_in H.
        apply prv_ex_eq in H; try rewrite <- HB; auto. cbn in H. subsimpl_in H.
        eapply Weak in H2. use_exists' H2 c. 2: auto. clear H2. assert1 H1. apply CE in H1 as [H1 H2].
        apply (AllE c) in H'. cbn in H'. subsimpl_in H'. apply CE1 in H'. eapply Weak in H'.
        apply (IE H') in H1. 2: auto. clear H'. apply (DE H1).
        -- apply DI1. eapply minZF_elem. rewrite <- HB, HA. auto 8. 3: apply (Weak H2); auto.
           2: auto. apply minZF_refl. rewrite <- HB, HA. auto 8.
        -- apply DI2. apply (AllE b) in H. cbn in H. subsimpl_in H. apply CE1 in H. eapply DE'.
           eapply IE. apply (Weak H). auto. eapply minZF_elem. rewrite <- HB, HA. auto 8.
           3: apply (Weak H2); auto. 2: auto. apply minZF_refl. rewrite <- HB, HA. auto 8.
    + apply CE2 in H. change (∃ $0 ≡' ↑ n ∧ y`[↑] ∈' $0) with (∃ $0 ≡' $n`[↑] ∧ y`[↑] ∈' $0) in H.
      now simpl_ex_in H.
Qed.
```

Need to explicitly change the goal because of substitutions

Already uses custom tactics

Deductive Proofs

Assumption management
rather tedious:

```
x, y, a, b :  term'
B :  list form'
HB :  (∃ $0 ≡' x`[↑]
        ∧ (∃ (∃ $0 ≡' x`[↑]`[↑]`[↑]
                ∧ (∃ $0 ≡' x`[↑]`[↑]`[↑]`[↑]
                     ∧ (∀ $0 ∈' ↑ 2 ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
            ∧ (∀ $0 ∈' a`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
     ∧ (∀ $0 ∈' y`[↑] ↔ (∃ $0 ∈' a`[↑]`[↑] ∧ ↑ 0 ∈' $0))
     :: (∃ (∃ $0 ≡' x`[↑]`[↑]
              ∧ (∃ (∃ $0 ≡' x`[↑]`[↑]`[↑]`[↑]
                      ∧ (∃ $0 ≡' x`[↑]`[↑]`[↑]`[↑]`[↑]
                           ∧ (∀ $0 ∈' ↑ 2 ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
                  ∧ (∀ $0 ∈' ↑ 2 ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
          ∧ (∀ $0 ∈' y`[↑]`[↑] ↔ (∃ $0 ∈' ↑ 1 ∧ ↑ 0 ∈' $0)))
        ∧ (∃ $0 ≡' ↑ n ∧ y`[↑] ∈' $0) :: x ∈' $n :: A <<= B
z :  term'
H :  ((∃ $0 ≡' x`[↑]
        ∧ (∃ $0 ≡' x`[↑]`[↑]
            ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
     ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
    ⊢ ∀ $0 ∈' z`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' x`[↑]
c :  term'
H1 :  (c ∈' a ∧ b ∈' c
      :: (∃ $0 ≡' x`[↑]
            ∧ (∃ $0 ≡' x`[↑]`[↑]
                 ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
          ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
     ⊢ c ≡' x ∨ c ≡' z
H2 :  (c ∈' a ∧ b ∈' c
      :: (∃ $0 ≡' x`[↑]
            ∧ (∃ $0 ≡' x`[↑]`[↑]
                 ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
          ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
     ⊢ b ∈' c
```

```
(1/1)
(c ≡' x
 :: c ∈' a ∧ b ∈' c
    :: (∃ $0 ≡' x`[↑]
          ∧ (∃ $0 ≡' x`[↑]`[↑]
               ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
        ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
⊢ b ∈' x ∨ b ≡' x
```

Deductive Proofs

Assumption management rather tedious:

```
x, y, a, b :  term'
B :  list form'
HB :  (∃ $0 ≡' x`[↑]
        ∧ (∃ (∃ $0 ≡' x`[↑]`[↑]`[↑]
                ∧ (∃ $0 ≡' x`[↑]`[↑]`[↑]`[↑]
                     ∧ (∀ $0 ∈' ↑ 2 ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
              ∧ (∀ $0 ∈' a`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
     ∧ (∀ $0 ∈' y`[↑] ↔ (∃ $0 ∈' a`[↑]`[↑] ∧ ↑ 0 ∈' $0))
     :: (∃ (∃ $0 ≡' x`[↑]`[↑]
               ∧ (∃ (∃ $0 ≡' x`[↑]`[↑]`[↑]`[↑]
                       ∧ (∃ $0 ≡' x`[↑]`[↑]`[↑]`[↑]`[↑]
                            ∧ (∀ $0 ∈' ↑ 2 ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
                    ∧ (∀ $0 ∈' ↑ 2 ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
           ∧ (∀ $0 ∈' y`[↑]`[↑] ↔ (∃ $0 ∈' ↑ 1 ∧ ↑ 0 ∈' $0)))
        ∧ (∃ $0 ≡' ↑ n ∧ y`[↑] ∈' $0) :: x ∈' $n :: A <<= B
z :  term'
H :  ((∃ $0 ≡' x`[↑]
        ∧ (∃ $0 ≡' x`[↑]`[↑]
             ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
     ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
    ⊢ ∀ $0 ∈' z`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' x`[↑]
c :  term'
H1 :  (c ∈' a ∧ b ∈' c
      :: (∃ $0 ≡' x`[↑]
           ∧ (∃ $0 ≡' x`[↑]`[↑]
                ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
         ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
     ⊢ c ≡' x ∨ c ≡' z
H2 :  (c ∈' a ∧ b ∈' c
      :: (∃ $0 ≡' x`[↑]
           ∧ (∃ $0 ≡' x`[↑]`[↑]
                ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
         ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
     ⊢ b ∈' c
```

```
(1/1)
(c ≡' x
 :: c ∈' a ∧ b ∈' c
    :: (∃ $0 ≡' x`[↑]
          ∧ (∃ $0 ≡' x`[↑]`[↑]
               ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
       ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
⊢ b ∈' x ∨ b ≡' x
```

Our tool is inspired by the Iris Proof Mode (Krebbers et al. (2017)):

```
1 subgoal
M : ucmraT
A : Type
P, R : iProp
Ψ : A → iProp
x : A
______________________________________(1/1)
"HP" : P
"HΨ" : Ψ x
"HR" : R
--------------------------------------∗
∃ a : A, Ψ a ∗ P
```

DEMO

Deductive Proofs

Assumption management very tedious:

```
x, y, a, b :  term'
B :  list form'
HB :  (∃ $0 ≡' x`[↑]
          ∧ (∃ (∃ $0 ≡' x`[↑]`[↑]`[↑]
                 ∧ (∃ $0 ≡' x`[↑]`[↑]`[↑]`[↑]
                      ∧ (∀ $0 ∈' ↑ 2 ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
               ∧ (∀ $0 ∈' a`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
     ∧ (∀ $0 ∈' y`[↑] ↔ (∃ $0 ∈' a`[↑]`[↑] ∧ ↑ 0 ∈' $0))
     :: (∃ (∃ $0 ≡' x`[↑]`[↑]
                ∧ (∃ (∃ $0 ≡' x`[↑]`[↑]`[↑]`[↑]
                       ∧ (∃ $0 ≡' x`[↑]`[↑]`[↑]`[↑]`[↑]
                            ∧ (∀ $0 ∈' ↑ 2 ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
                     ∧ (∀ $0 ∈' ↑ 2 ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
             ∧ (∀ $0 ∈' y`[↑]`[↑] ↔ (∃ $0 ∈' ↑ 1 ∧ ↑ 0 ∈' $0)))
          ∧ (∃ $0 ≡' ↑ n ∧ y`[↑] ∈' $0) :: x ∈' $n :: A <<= B
z :  term'
H :  ((∃ $0 ≡' x`[↑]
          ∧ (∃ $0 ≡' x`[↑]`[↑]
               ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
     ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
    ⊢ ∀ $0 ∈' z`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' x`[↑]
c :  term'
H1 :  (c ∈' a ∧ b ∈' c
       :: (∃ $0 ≡' x`[↑]
             ∧ (∃ $0 ≡' x`[↑]`[↑]
                  ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
          ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
     ⊢ c ≡' x ∨ c ≡' z
H2 :  (c ∈' a ∧ b ∈' c
       :: (∃ $0 ≡' x`[↑]
             ∧ (∃ $0 ≡' x`[↑]`[↑]
                  ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
          ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
     ⊢ b ∈' c
```

```
(1/1)
(c ≡' x
 :: c ∈' a ∧ b ∈' c
    :: (∃ $0 ≡' x`[↑]
          ∧ (∃ $0 ≡' x`[↑]`[↑]
               ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
       ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
⊢ b ∈' x ∨ b ≡' x
```

Same proof done using Proof Mode:

```
Lemma prv_to_min_inductive n :
  minZFeq' ⊢ rm_const_fm (inductive $n) → is_inductive $n.
Proof.
  cbn. fstart. fintros "[[e [H [s [H0 H1]]]] H2]". fsplit.
  - fexists e. fsplit. fintros x; fapply "H". frewrite <- "H0"; ctx.
  - fintros. fdestruct ("H2" x) as "...".
    { fexists x. fsplit. fapply ax_refl'. fexists $n. fsplit.
      fapply ax_refl'. ctx. }
    fexists x0. fsplit.
    + fintros y. fsplit.
      * fintros "H11". fapply "H8" in "H11" as "[? [? ?]]".
        fapply "H7" in "H11" as "[|]".
        -- fleft. frewrite <- "H2". frewrite <- "H11". ctx.
        -- fright. fdestruct ("H6" y). fdestruct "H6".
           frewrite <- "H11". ctx. all: frewrite "H6"; ctx.
      * fintros "[|]".
        -- fapply "H8". fexists x2. fsplit. fapply "H7". fleft.
           fapply ax_refl'. frewrite "H2". ctx.
        -- frewrite "H11". fapply "H8". fexists x3. fsplit. fapply "H7".
           fright. fapply ax_refl'. fapply "H6". fleft. fapply ax_sym'. ctx.
    + frewrite <- "H9". ctx.
Qed.
```

Deductive Proofs

Assumption management very tedious:

```
x, y, a, b :  term'
B :  list form'
HB :  (∃ $0 ≡' x`[↑]
        ∧ (∃ (∃ $0 ≡' x`[↑]`[↑]`[↑]
                ∧ (∃ $0 ≡' x`[↑]`[↑]`[↑]`[↑]
                     ∧ (∀ $0 ∈' ↑ 2 ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
             ∧ (∀ $0 ∈' a`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
      ∧ (∀ $0 ∈' y`[↑] ↔ (∃ $0 ∈' a`[↑]`[↑] ∧ ↑ 0 ∈' $0))
      :: (∃ (∃ $0 ≡' x`[↑]`[↑]
               ∧ (∃ (∃ $0 ≡' x`[↑]`[↑]`[↑]`[↑]
                       ∧ (∃ $0 ≡' x`[↑]`[↑]`[↑]`[↑]`[↑]
                            ∧ (∀ $0 ∈' ↑ 2 ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
                    ∧ (∀ $0 ∈' ↑ 2 ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
            ∧ (∀ $0 ∈' y`[↑]`[↑] ↔ (∃ $0 ∈' ↑ 1 ∧ ↑ 0 ∈' $0)))
        ∧ (∃ $0 ≡' ↑ n ∧ y`[↑] ∈' $0) :: x ∈' $n :: A <<= B
z :  term'
H :  ((∃ $0 ≡' x`[↑]
        ∧ (∃ $0 ≡' x`[↑]`[↑]
             ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
      ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
     ⊢ ∀ $0 ∈' z`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' x`[↑]
c :  term'
H1 :  (c ∈' a ∧ b ∈' c
       :: (∃ $0 ≡' x`[↑]
             ∧ (∃ $0 ≡' x`[↑]`[↑]
                  ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
          ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
      ⊢ c ≡' x ∨ c ≡' z
H2 :  (c ∈' a ∧ b ∈' c
       :: (∃ $0 ≡' x`[↑]
             ∧ (∃ $0 ≡' x`[↑]`[↑]
                  ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
          ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
      ⊢ b ∈' c
```

```
(1/1)
(c ≡' x
 :: c ∈' a ∧ b ∈' c
    :: (∃ $0 ≡' x`[↑]
          ∧ (∃ $0 ≡' x`[↑]`[↑]
               ∧ (∀ $0 ∈' z`[↑]`[↑]`[↑] ↔ $0 ≡' ↑ 1 ∨ $0 ≡' ↑ 0)))
       ∧ (∀ $0 ∈' a`[↑] ↔ $0 ≡' x`[↑] ∨ $0 ≡' z`[↑]) :: B)
⊢ b ∈' x ∨ b ≡' x
```

Same proof done using Proof Mode:

```
Lemma prv_to_min_inductive n :
  minZFeq' ⊢ rm_const_fm (inductive $n) --> is_inductive $n.
Proof.
  cbn. fstart. fintros "[[e [H [s [H0 H1]]]] H2]". fsplit.
  - fexists e. fsplit. fintros x; fapply "H". frewrite <- "H0"; ctx.
  - fintros. fdestruct ("H2" x) as "...".
    { fexists x. fsplit. fapply ax_refl'. fexists $n. fsplit.
      fapply ax_refl'. ctx. }
    fexists x0. fsplit.
    + fintros y. fsplit.
      * fintros "H11". fapply "H8" in "H11" as "[? [? ?]]".
        fapply "H7" in "H11" as "[|]".
        -- fleft. frewrite <- "H2". frewrite <- "H11". ctx.
        -- fright. fdestruct ("H6" y). fdestruct "H6".
           frewrite <- "H11". ctx. all: frewrite "H6"; ctx.
      * fintros "[|]".
        -- fapply "H8". fexists x2. fsplit. fapply "H7". fleft.
           fapply ax_refl'. frewrite "H2". ctx.
        -- frewrite "H11". fapply "H8". fexists x3. fsplit. fapply "H7".
           fright. fapply ax_refl'. fapply "H6". fleft. fapply ax_sym'. ctx.
    + frewrite <- "H9". ctx.
Qed.
```

```
e, s, x, x0, x1, x2, x3, x4, x5, x6, y, x7 :  term'
```

```
(1/1)
 minZFeq'
 "H" : ∀ x8, ¬ x8 ∈' e`[↑]
 "H0" : s ≡' $n
 "H1" : e ∈' s
 "H3" : x ∈' $n
 "H2" : x2 ≡' x
 "H4" : x4 ≡' x
 "H5" : x5 ≡' x
 "H6" : ∀ x8, x8 ∈' x3`[↑] ↔ x8 ≡' x4`[↑] ∨ x8 ≡' x5`[↑]
 "H7" : ∀ x8, x8 ∈' x1`[↑] ↔ x8 ≡' x2`[↑] ∨ x8 ≡' x3`[↑]
 "H8" : ∀ x8, x8 ∈' x0`[↑] ↔ (∃ x9, x9 ∈' x1`[↑]`[↑] ∧ x8 ∈' x9)
 "H9" : x6 ≡' $n
 "H10" : x0 ∈' x6
 "H12" : y ∈' x7
 "H11" : x7 ≡' x2
______________________________
 (y ∈' x ∨ y ≡' x)
```

Implementation Details

- (Almost) completely implemented using Ltac

  Except for MetaCoq plugin to turn strings into Coq identifiers

Implementation Details

- (Almost) completely implemented using Ltac
  Except for MetaCoq plugin to turn strings into Coq identifiers

- Aliases to control notations. They also carry the hypothesis and variable names, e.g.

```
econs : string -> form -> list form
econs s phi E := phi :: E
```

Implementation Details

- (Almost) completely implemented using Ltac
  Except for MetaCoq plugin to turn strings into Coq identifiers
- Aliases to control notations. They also carry the hypothesis and variable names, e.g.

```
econs : string -> form -> list form
econs s phi E := phi :: E
```

- Rewriting:

Implementation Details

- (Almost) completely implemented using Ltac
  Except for MetaCoq plugin to turn strings into Coq identifiers

- Aliases to control notations. They also carry the hypothesis and variable names, e.g.

```
econs : string -> form -> list form
econs s phi E := phi :: E
```

- Rewriting:
  - Equality not built into our FOL. Instead user can provide custom equality symbol and congruence lemmas using type class.

Implementation Details

- (Almost) completely implemented using Ltac
  Except for MetaCoq plugin to turn strings into Coq identifiers

- Aliases to control notations. They also carry the hypothesis and variable names, e.g.

```
econs : string -> form -> list form
econs s phi E := phi :: E
```

- Rewriting:
  - Equality not built into our FOL. Instead user can provide custom equality symbol and congruence lemmas using type class.
  - We then rewrite by applying the following substitution rule:

$$\mathcal{T} \vdash x = y \ \rightarrow \ \forall \varphi. \mathcal{T} \vdash \varphi[x] = \varphi[y]$$

Remarks

- Across whole development overall reduction from 167 to 89 proof lines

Remarks

- Across whole development overall reduction from 167 to 89 proof lines
- Limitations:
  - Performance: For larger proofs noticeable delays (up to a few seconds for complex tactics).
  - Deduction on theories not fully supported yet

Conclusion

- 3 Tools: HOAS input language, reification tactic, proof mode

Conclusion

- 3 Tools: HOAS input language, reification tactic, proof mode
- Ideas all on the market, adapted to FOL library

Conclusion

- 3 Tools: HOAS input language, reification tactic, proof mode
- Ideas all on the market, adapted to FOL library
- Prototypes with limitations regarding performance and scale

Conclusion

- 3 Tools: HOAS input language, reification tactic, proof mode
- Ideas all on the market, adapted to FOL library
- Prototypes with limitations regarding performance and scale
- You can use our FOL library without caring about the implementation

Conclusion

- 3 Tools: HOAS input language, reification tactic, proof mode
- Ideas all on the market, adapted to FOL library
- Prototypes with limitations regarding performance and scale
- You can use our FOL library without caring about the implementation
- Tools in principle adjustable to similar object logics

Conclusion

- 3 Tools: HOAS input language, reification tactic, proof mode
- Ideas all on the market, adapted to FOL library
- Prototypes with limitations regarding performance and scale
- You can use our FOL library without caring about the implementation
- Tools in principle adjustable to similar object logics
- Demos and manuals: reification tactic, proof mode

Conclusion

- 3 Tools: HOAS input language, reification tactic, proof mode
- Ideas all on the market, adapted to FOL library
- Prototypes with limitations regarding performance and scale
- You can use our FOL library without caring about the implementation
- Tools in principle adjustable to similar object logics
- Demos and manuals: reification tactic, proof mode

Thank you!

Bibliography I

Forster, Y. and Kunze, F. (2019). A certifying extraction with time bounds from coq to call-by-value λ-calculus. In *Interactive Theorem Proving - 10th International Conference, ITP 2019, Portland, USA*, page 17:1–17:19. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. Also available as arXiv:1904.11818.

Forster, Y., Larchey-Wendling, D., Dudenhefner, A., Heiter, E., Kirst, D., Kunze, F., Smolka, G., Spies, S., Wehr, D., and Wuttke, M. (2020). A Coq library of undecidable problems. In *CoqPL 2020 The Sixth International Workshop on Coq for Programming Languages*.

Kirst, D. and Hermes, M. (2021). Synthetic Undecidability and Incompleteness of First-Order Axiom Systems in Coq. In Cohen, L. and Kaliszyk, C., editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:20, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Krebbers, R., Timany, A., and Birkedal, L. (2017). Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 205–217, New York, NY, USA. Association for Computing Machinery.

Laurent, O. (2021). An anti-locally-nameless approach to formalizing quantifiers. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 300–312.

Rech, F. (2020). Mechanising set theory in coq. Master's thesis, Saarland University.