Design of point-and-click user interfaces for proof assistants

Bohua Zhan[1], Zhenyan Ji[2], Wenfan Zhou[2], Chaozhu Xiang[2], Jie Hou[2] and Wenhui Sun[2]

[1]State Key Lab. of Computer Science, Institute of Software, CAS
[2]Beijing Jiaotong University

November 6, 2019

Motivation

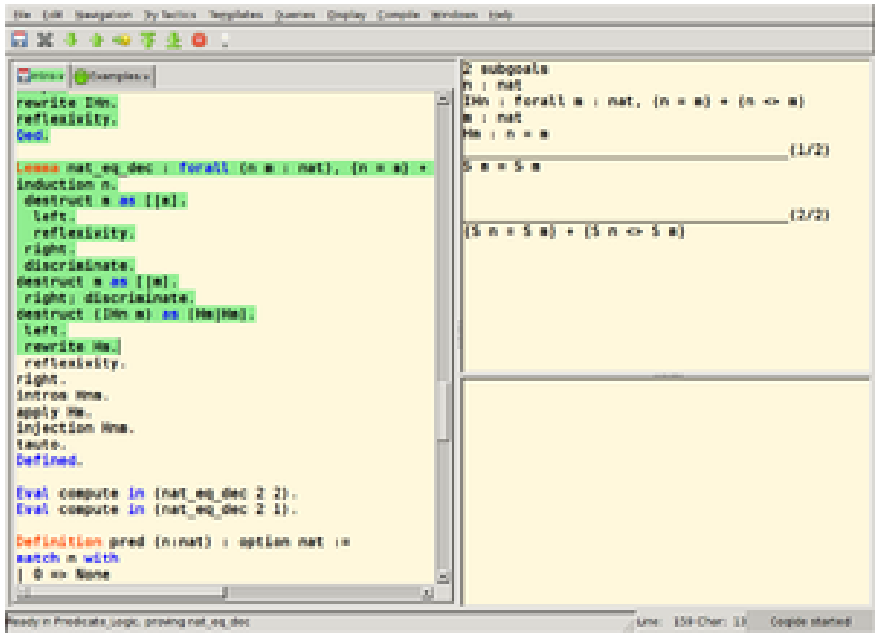
- Interactive Theorem Proving is concerned with *constructing* and *checking* formal proofs by interaction between the human user and the computer.

Motivation

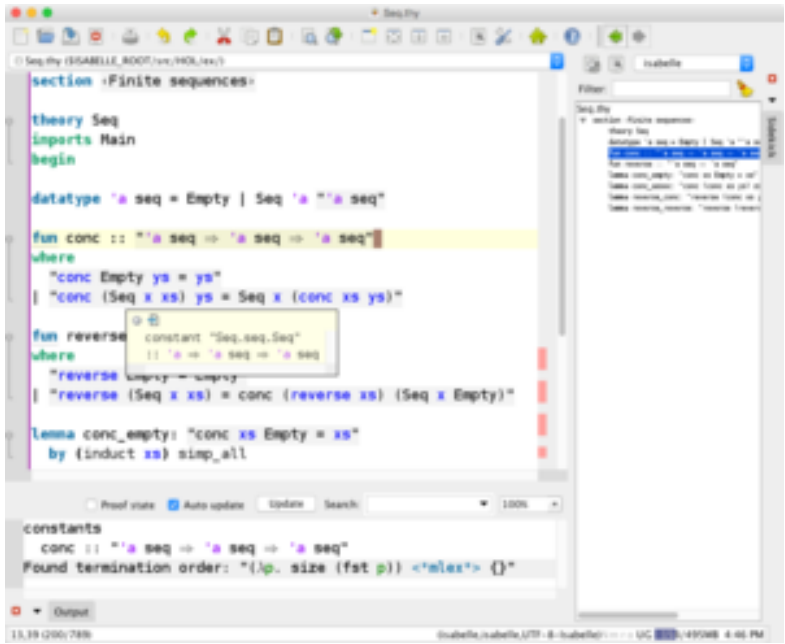
- Interactive Theorem Proving is concerned with *constructing* and *checking* formal proofs by interaction between the human user and the computer.
- The user interface receives user commands and displays the current status of the proof. As a lot of time is spent interacting with the user interface, it can be considered as a key part of the system.

Motivation

Existing user interfaces are mostly text-based.



Coq



Isabelle

Motivation

Generally has a steep learning curve because:

- Need to remember (and choose between) a large number of tactics.
- Need to remember names of theorems and labels in a proof.
- For structured language like Isabelle/Isar, need to learn another layer of syntax.

Motivation

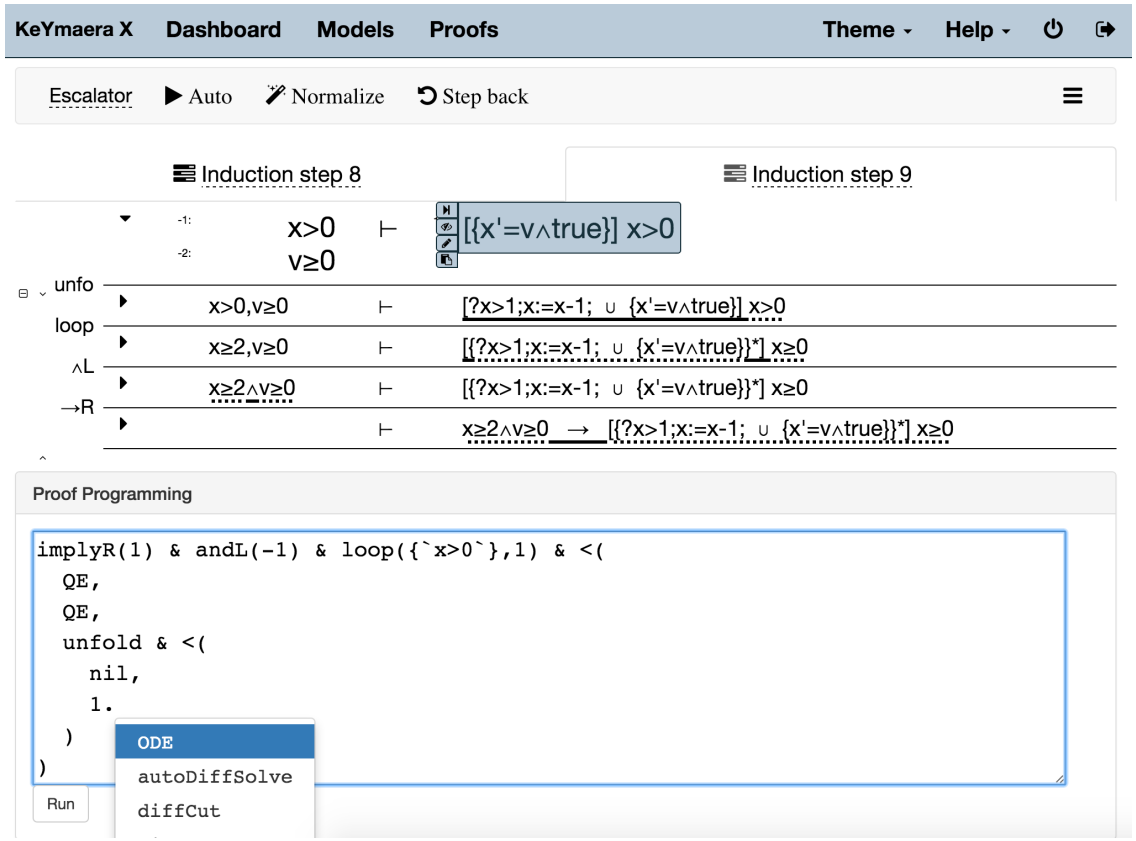
Generally has a steep learning curve because:

- Need to remember (and choose between) a large number of tactics.
- Need to remember names of theorems and labels in a proof.
- For structured language like Isabelle/Isar, need to learn another layer of syntax.

Is it possible to build user interfaces for proof assistants based on pointing-and-clicking?

Existing work

KeYmaera/KeYmaeraX[1]

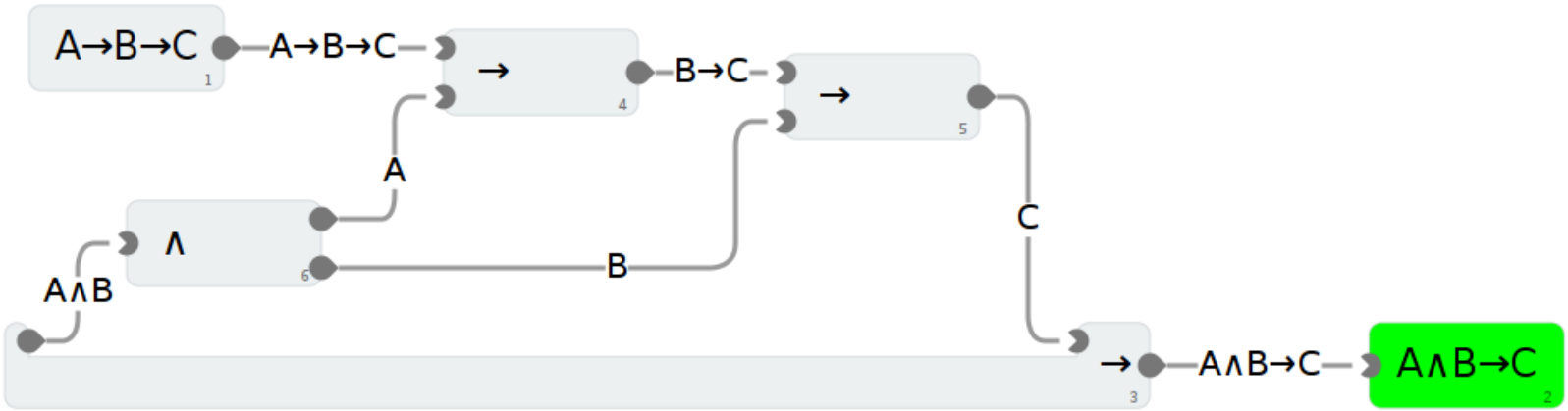


Focuses on verification of hybrid systems.

[1] http://www.ls.cs.cmu.edu/KeYmaeraX/tutorial/fm2016/tacticeditor.png

Existing work

Visual theorem proving with the incredible proof machine[2]



Conduct proofs in logic by connecting blocks.

[2] https://www.joachim-breitner.de/various/incredibe-proof.png

Existing work

Earlier works:

- Bertot, Y. and Théry, L. **A Generic Approach to Building User Interfaces for Theorem Provers**, J. of Symbolic Computation, 1998.
- Abrial, J. and Cansel, D. **Click'n Prove: Interactive Proofs within Set Theory**, TPHOLs, 2003.

Motivation

Existing work on user interfaces are either limited to proofs in logic and set theory, or to special problem domains.

Motivation

Existing work on user interfaces are either limited to proofs in logic and set theory, or to special problem domains.

In our work, we aim for the system to be useful for general purpose theorem proving. In particular, the system can

- work with newly added theorems.
- be extended with domain-specific proof methods.

holpy

higher-order logic in python

Proof assistant based on higher-order logic, implemented in Python.

holpy

higher-order logic in python

Proof assistant based on higher-order logic, implemented in Python.

Underlying logic is similar to that of Isabelle/HOL, HOL Light, and HOL4.

Proof object

holpy works with an explicit proof object:

- Proof is represented as a *list of sequents*.

Proof object

holpy works with an explicit proof object:

- Proof is represented as a *list of sequents*.
- Each sequent is derived from (zero or more) previous sequents using a *proof rule*.

Proof object

holpy works with an explicit proof object:

- Proof is represented as a *list of sequents*.
- Each sequent is derived from (zero or more) previous sequents using a *proof rule*.
- A proof rule can be either *primitive* or *composite* (also known as *macros*).

Proof object

holpy works with an explicit proof object:

- Proof is represented as a *list of sequents*.
- Each sequent is derived from (zero or more) previous sequents using a *proof rule*.
- A proof rule can be either *primitive* or *composite* (also known as *macros*).
- An incomplete proof has *holes* signifying subgoals.

Example

0. $A \wedge B \vdash A \wedge B$ **by** assume $A \wedge B$
1. $A \wedge B \vdash B$ **by** apply_theorem conjD2 **from** 0
2. $A \wedge B \vdash A$ **by** apply_theorem conjD1 **from** 0
3. $A \wedge B \vdash B \wedge A$ **by** apply_theorem conjI **from** 1, 2
4. $\vdash A \wedge B \to B \wedge A$ **by** implies_intr **from** 3

```
0    assume A ∧ B
1    have B by apply_theorem conjD2 from 0
2    have A by apply_theorem conjD1 from 0
3    show B ∧ A by apply_theorem conjI from 1, 2
```

Example

0. $\vdash 0 + 0 = 0$ **by** sorry
1.0. $\vdash$ _VAR n **by** variable n :: nat
1.1. $n + 0 = n \vdash n + 0 = n$ **by** assume $n + 0 = n$
1.2. $n + 0 = n \vdash \text{Suc}\ n + 0 = \text{Suc}\ n$ **by** sorry
1. $\vdash \forall n.\ n + 0 = n \longrightarrow \text{Suc}\ n + 0 = \text{Suc}\ n$ **by** intros **from** 1.0, 1.1, 1.2
2. $\vdash n + 0 = n$ **by** apply_theorem_for nat_induct, $\{P\text{: } \lambda n.\ n + 0 = n, x\text{: } n\}$ **from** 0, 1

```
0    have (0::nat) + 0 = 0 by sorry
1    have ∀n. n + 0 = n ⟶ Suc n + 0 = Suc n with
1.0    fix n :: nat
1.1    assume n + 0 = n
1.2    show Suc n + 0 = Suc n by sorry
2    show x + 0 = x by apply_theorem_for nat_induct, {}, {P: λx::nat. x + 0 = x, x: x} from 0, 1
```

Note: facts available for the current subgoal are always visible.

Overall design

- State of the proof is the current proof object.

Overall design

- State of the proof is the current proof object.
- At each step, the user selects a subgoal and a list of facts available for use at that subgoal.

Overall design

- State of the proof is the current proof object.
- At each step, the user selects a subgoal and a list of facts available for use at that subgoal.
- System provides a list of suggested actions.

Overall design

- State of the proof is the current proof object.
- At each step, the user selects a subgoal and a list of facts available for use at that subgoal.
- System provides a list of suggested actions.
- User chooses among the suggestions, or another action from the menu.

Methods

A method is a function that takes the following inputs:

1. The current theory environment and the current proof state.
2. The selected goal and list of selected facts.
3. Additional parameters depending on the method.

and (if successful) returns a new proof state.

Methods

A method is a function that takes the following inputs:

1. The current theory environment and the current proof state.
2. The selected goal and list of selected facts.
3. Additional parameters depending on the method.

and (if successful) returns a new proof state.

Abstracts the concept of action during a proof. In particular, any *tactic* can be packaged as a method.

Search function

In addition, a method provides a *search function*, that takes the following inputs:

1. The current theory environment and the current proof state.
2. The selected goal and list of selected facts.

and returns a (possibly empty) list of suggested parameters.

Search function

In addition, a method provides a *search function*, that takes the following inputs:

1. The current theory environment and the current proof state.
2. The selected goal and list of selected facts.

and returns a (possibly empty) list of suggested parameters.

The suggestions for all methods are collected together to form the suggestions by the system.

Searching and applying theorems

- Method `apply_backward_step` applies a theorem in the backward direction (corresponds to `apply rule` in Isabelle).

Searching and applying theorems

- Method `apply_backward_step` applies a theorem in the backward direction (corresponds to `apply rule` in Isabelle).
- Its *search function* searches for applicable theorems by matching the current subgoal with the conclusion.

Searching and applying theorems

- Method `apply_backward_step` applies a theorem in the backward direction (corresponds to `apply rule` in Isabelle).
- Its *search function* searches for applicable theorems by matching the current subgoal with the conclusion.
- The user may *annotate* a theorem to include it in the search.

Searching and applying theorems

- Method `apply_backward_step` applies a theorem in the backward direction (corresponds to `apply rule` in Isabelle).
- Its *search function* searches for applicable theorems by matching the current subgoal with the conclusion.
- The user may *annotate* a theorem to include it in the search.
- Likewise, there are methods for forward application, rewriting, etc.

Example 1

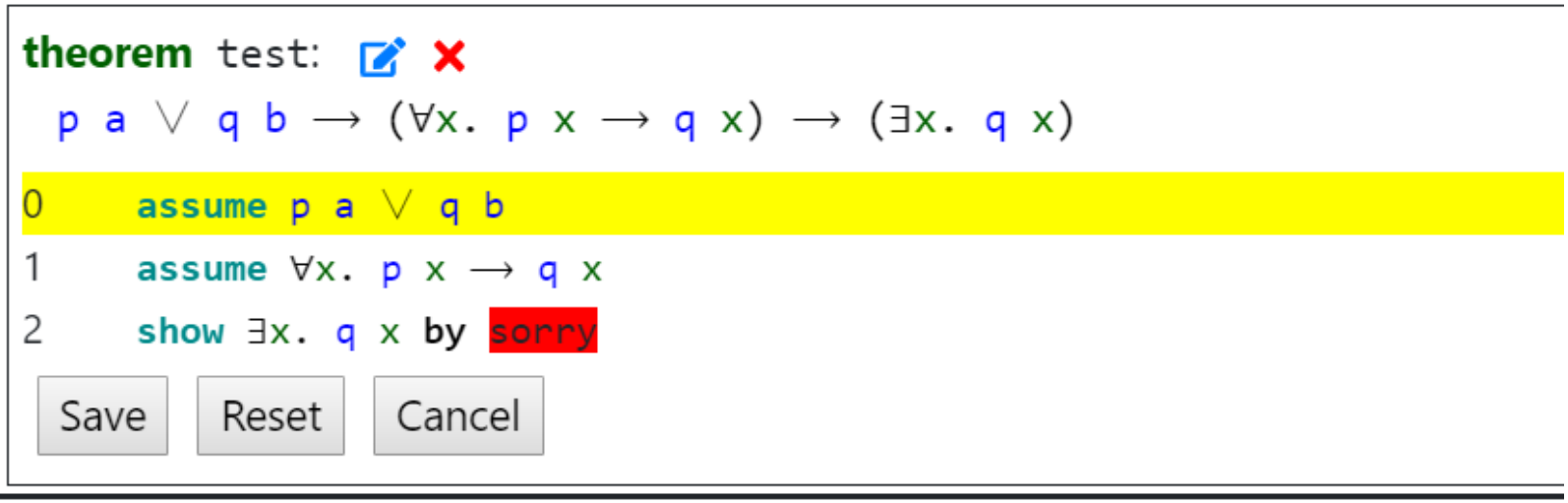
First example, from Bertot and Théry:

$$p\ a \lor q\ b \to (\forall x.\ p\ x \to q\ x) \to (\exists x.\ q\ x)$$

Illustrates:

- Propositional and predicate logic.

Example 1




```
OK. 1 gap(s) remaining.
<0/0>  Current state

 disjE (b): q b ⟶ (∃x. q x), p a ⟶ (∃x. q x)
```

Example 1

theorem test:

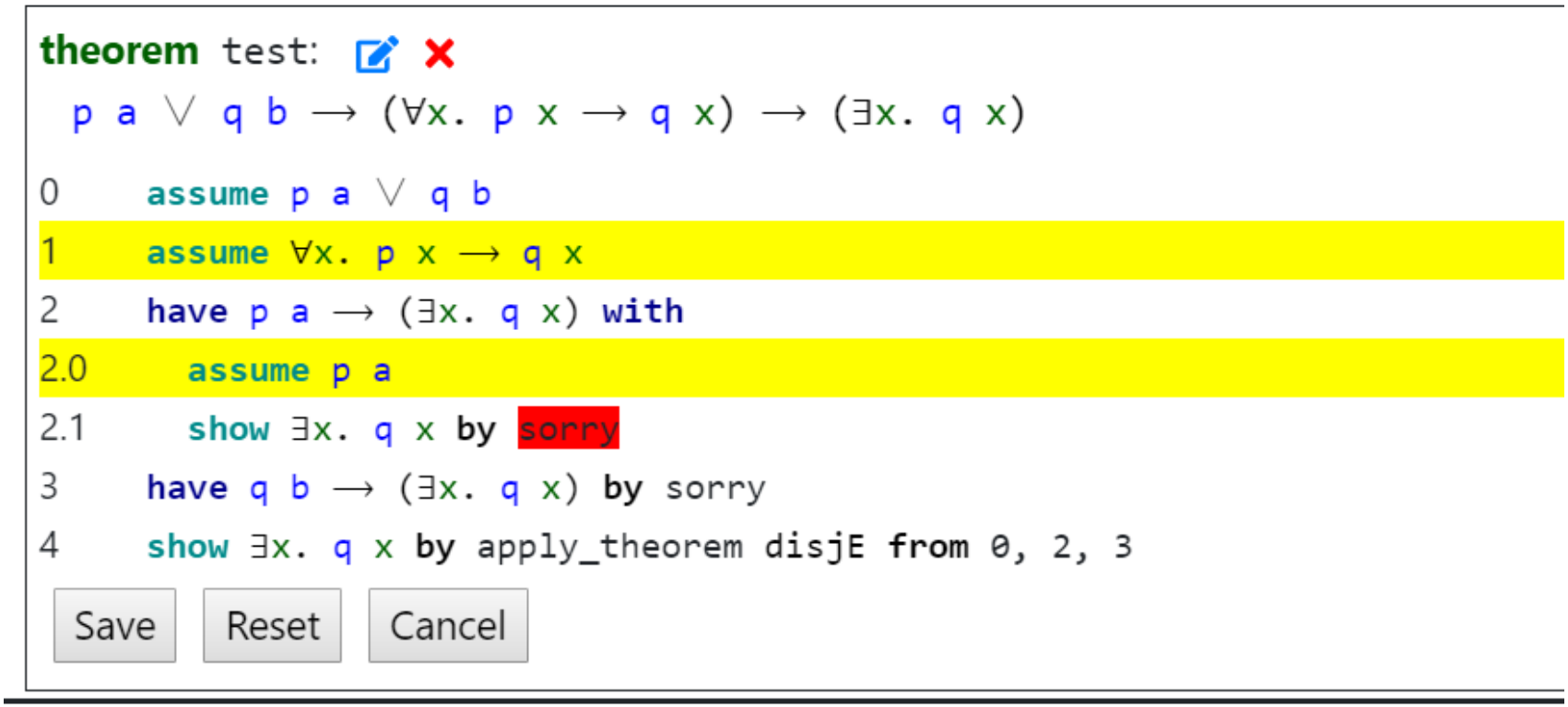
```
p a ∨ q b ⟶ (∀x. p x ⟶ q x) ⟶ (∃x. q x)
```

```
0    assume p a ∨ q b
1    assume ∀x. p x ⟶ q x
2    have p a ⟶ (∃x. q x) by sorry
3    have q b ⟶ (∃x. q x) by sorry
4    show ∃x. q x by apply_theorem disjE from 0, 2, 3
```

Save | Reset | Cancel

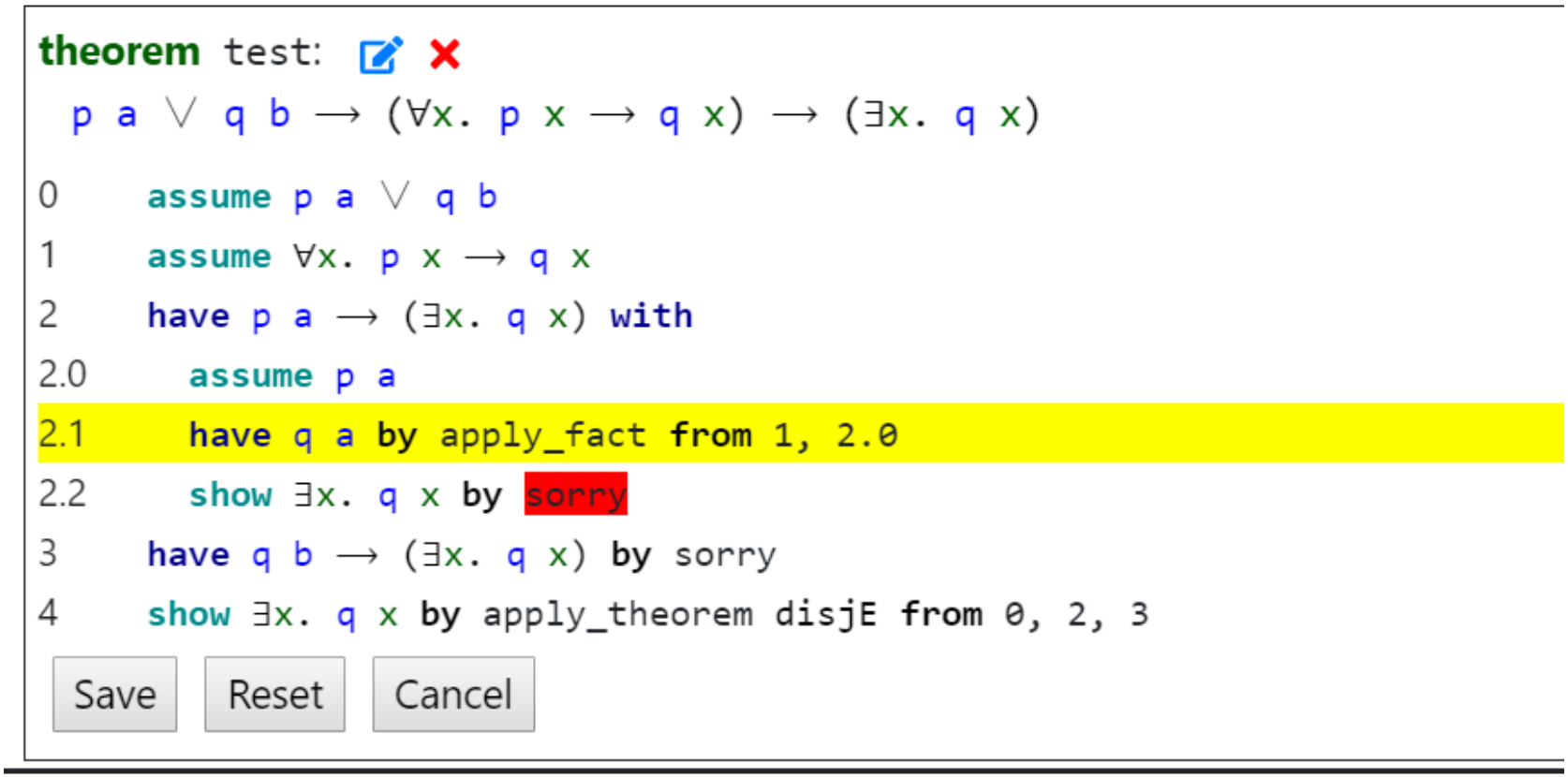
```
OK. 2 gap(s) remaining.
<1/1>  Current state

 introduction on 2
```

Example 1


theorem test:
p a ∨ q b ⟶ (∀x. p x ⟶ q x) ⟶ (∃x. q x)
0 assume p a ∨ q b
1 assume ∀x. p x ⟶ q x
2 have p a ⟶ (∃x. q x) with
2.0 assume p a
2.1 show ∃x. q x by sorry
3 have q b ⟶ (∃x. q x) by sorry
4 show ∃x. q x by apply_theorem disjE from 0, 2, 3
Save
Reset
Cancel


OK. 2 gap(s) remaining.
<2/2> Current state
Apply fact (f) 1 onto 2.0

Example 1



OK. 2 gap(s) remaining.

<3/3> Current state

exI (b): (solves)

Example 1

theorem test:

```
p a ∨ q b ⟶ (∀x. p x ⟶ q x) ⟶ (∃x. q x)
```

```
0     assume p a ∨ q b
1     assume ∀x. p x ⟶ q x
2     have p a ⟶ (∃x. q x) with
2.0     assume p a
2.1     have q a by apply_fact from 1, 2.0
2.2     show ∃a. q a by apply_theorem_for exI, {a: 'a}, {P: q, a: a} from 2.1
3     have q b ⟶ (∃x. q x) by sorry
4     show ∃x. q x by apply_theorem disjE from 0, 2, 3
```

Save Reset Cancel

```
OK. 1 gap(s) remaining.
<4/4>  Current state

 introduction on 3
```

Example 1

theorem test:

```
p a ∨ q b ⟶ (∀x. p x ⟶ q x) ⟶ (∃x. q x)
```

```
0     assume p a ∨ q b
1     assume ∀x. p x ⟶ q x
2     have p a ⟶ (∃x. q x) with
2.0     assume p a
2.1     have q a by apply_fact from 1, 2.0
2.2     show ∃a. q a by apply_theorem_for exI, {a: 'a}, {P: q, a: a} from 2.1
3     have q b ⟶ (∃x. q x) with
3.0     assume q b
3.1     show ∃x. q x by sorry
4     show ∃x. q x by apply_theorem disjE from 0, 2, 3
```

Save Reset Cancel

```
OK. 1 gap(s) remaining.
<5/5>  Current state

 exI (b): (solves)
```

Example 1

theorem test:

```
p a ∨ q b ⟶ (∀x. p x ⟶ q x) ⟶ (∃x. q x)
```

```
0     assume p a ∨ q b
1     assume ∀x. p x ⟶ q x
2     have p a ⟶ (∃x. q x) with
2.0     assume p a
2.1     have q a by apply_fact from 1, 2.0
2.2     show ∃a. q a by apply_theorem_for exI, {a: 'a}, {P: q, a: a} from 2.1
3     have q b ⟶ (∃x. q x) with
3.0     assume q b
3.1     show ∃a. q a by apply_theorem_for exI, {a: 'a}, {P: q, a: b} from 3.0
4     show ∃x. q x by apply_theorem disjE from 0, 2, 3
```

Save Reset Cancel

```
OK. Proof complete!
<6/6>  Current state
```

Example 2

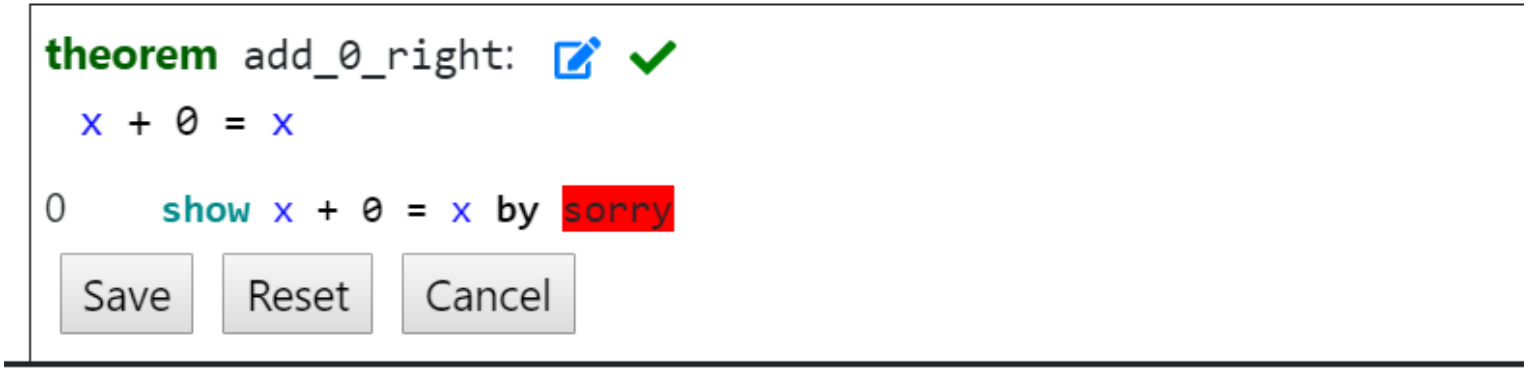
Second example, from Peano arithmetic:

$$x + 0 = x$$

Illustrates:

- Induction on natural numbers.
- Rewriting using a theorem / fact in proof.

Example 2




```
OK. 1 gap(s) remaining.
<0/0>  Current state

 induction nat_induct var: x
```

Example 2

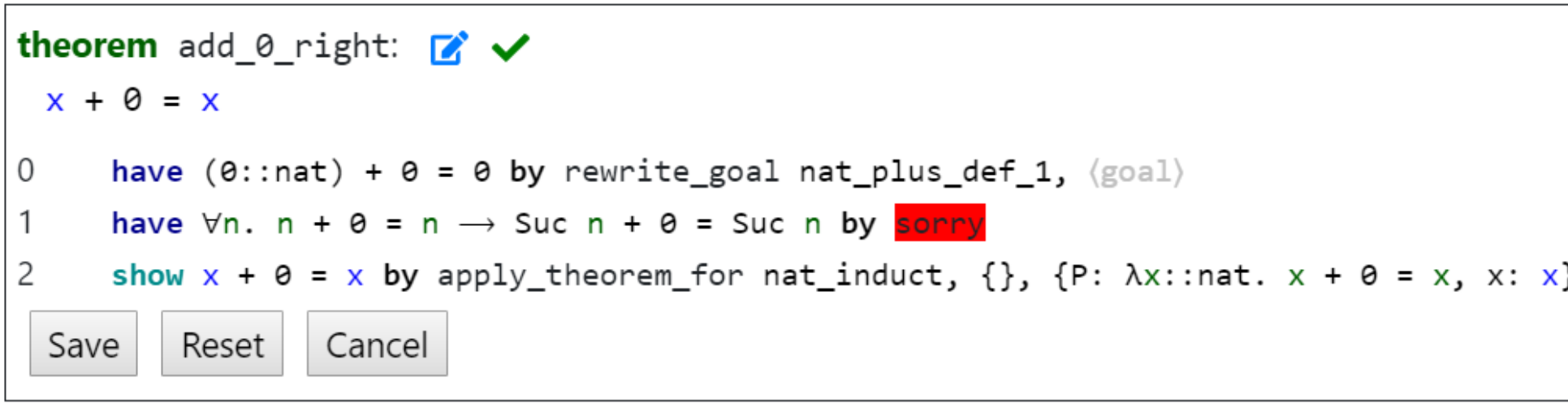
theorem add_0_right:

```
  x + 0 = x
0     have (0::nat) + 0 = 0 by sorry
1     have ∀n. n + 0 = n ⟶ Suc n + 0 = Suc n by sorry
2     show x + 0 = x by apply_theorem_for nat_induct, {}, {P: λx::nat. x + 0 = x, x: x
```

Save | Reset | Cancel

```
OK. 2 gap(s) remaining.
<1/1>  Current state

 nat_plus_def_1 (r): (solves)
```

Example 2




```
OK. 1 gap(s) remaining.
<2/2>  Current state

 introduction on 1
```

Example 2

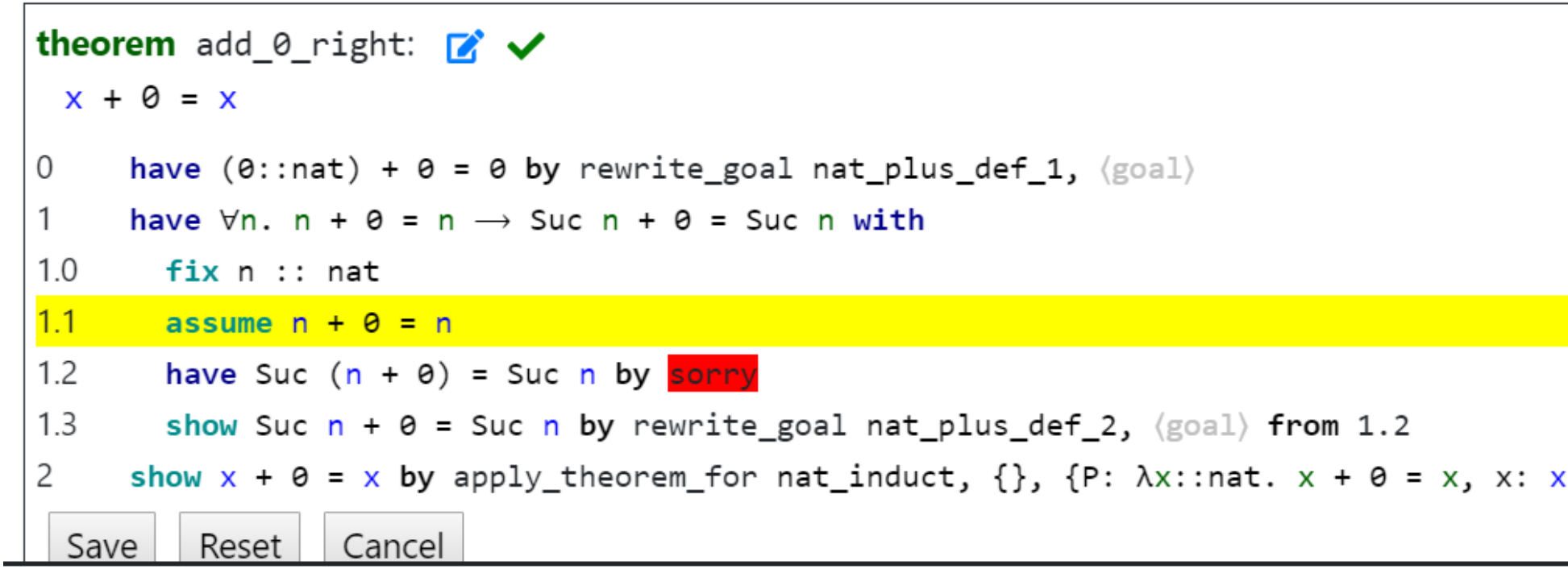
theorem add_0_right:

```
  x + 0 = x
0     have (0::nat) + 0 = 0 by rewrite_goal nat_plus_def_1, ⟨goal⟩
1     have ∀n. n + 0 = n ⟶ Suc n + 0 = Suc n with
1.0     fix n :: nat
1.1     assume n + 0 = n
1.2     show Suc n + 0 = Suc n by sorry
2     show x + 0 = x by apply_theorem_for nat_induct, {}, {P: λx::nat. x + 0 = x, x: x
```

Save Reset Cancel

```
OK. 1 gap(s) remaining.
<3/3>  Current state

 nat_plus_def_2 (r): Suc (n + 0) = Suc n
```

Example 2

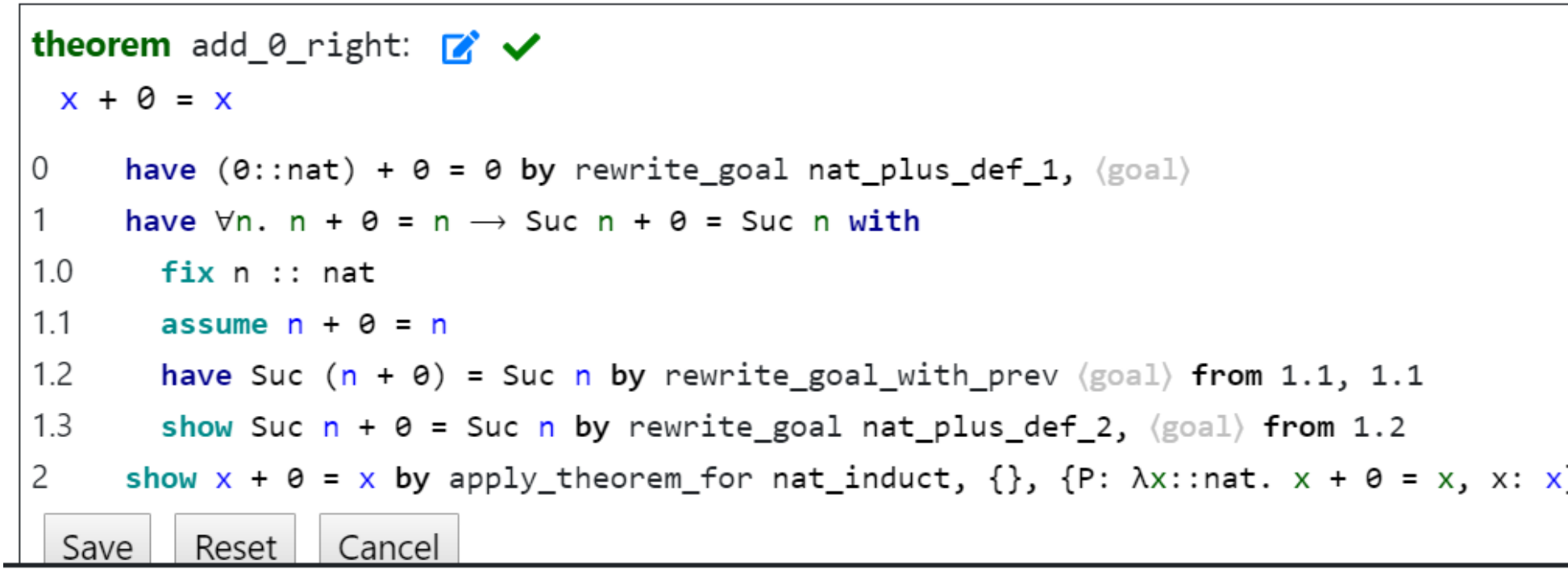


OK. 1 gap(s) remaining.

<4/4> Current state

rewrite with fact: n + 0 = n

Example 2



OK. Proof complete!

<5/5> Current state

Example 3

Third example, from proof of Knaster-Tarski theorem:

```
bnd_mono h → h (lfp h) = lfp h
```

Illustrates:

- Applying existing theorems.
- Inserting intermediate goal.

Example 3

theorem lfp_unfold: ✔

bnd_mono h ⟶ h (lfp h) = lfp h

```
0    assume bnd_mono h
1    show h (lfp h) = lfp h by sorry
```

Save | Reset | Cancel

OK. 1 gap(s) remaining.

<0/0> Current state

lfp_def (r): h (⋂{X. h X ⊆ X}) = ⋂{X. h X ⊆ X}

bnd_mono_def (r): have ∀W. ∀X. W ⊆ X ⟶ h W ⊆ h X

Example 3

theorem lfp_unfold: ✔

```
bnd_mono h ⟶ h (lfp h) = lfp h
```

```
0    assume bnd_mono h
1    have ∀W. ∀X. W ⊆ X ⟶ h W ⊆ h X by rewrite_fact bnd_mono_def from 0
2    show h (lfp h) = lfp h by sorry
```

Save Reset Cancel

```
OK. 1 gap(s) remaining.
<1/1>  Current state

lfp_def (r): h (⋂{X. h X ⊆ X}) = ⋂{X. h X ⊆ X}

member_ext (b): ∀x. x ∈ h (lfp h) ⟷ x ∈ lfp h

subset_antisym (b): h (lfp h) ⊆ lfp h, lfp h ⊆ h (lfp h)
```

Example 3

theorem lfp_unfold:

```
bnd_mono h ⟶ h (lfp h) = lfp h
```

```
0    assume bnd_mono h
1    have ∀W. ∀X. W ⊆ X ⟶ h W ⊆ h X by rewrite_fact bnd_mono_def from 0
2    have h (lfp h) ⊆ lfp h by sorry
3    have lfp h ⊆ h (lfp h) by sorry
4    show h (lfp h) = lfp h by apply_theorem subset_antisym from 2, 3
```

Save | Reset | Cancel

OK. 2 gap(s) remaining.

<2/2> Current state

```
lfp_def (r): h (⋂{X. h X ⊆ X}) ⊆ ⋂{X. h X ⊆ X}
lfp_greatest (b): ∀X. h X ⊆ X ⟶ h (lfp h) ⊆ X
subsetI (b): ∀x. x ∈ h (lfp h) ⟶ x ∈ lfp h
subset_trans (b)
```

Example 3

theorem lfp_unfold:

```
bnd_mono h ⟶ h (lfp h) = lfp h
```

```
0    assume bnd_mono h
1    have ∀W. ∀X. W ⊆ X ⟶ h W ⊆ h X by rewrite_fact bnd_mono_def from 0
2    have ∀X. h X ⊆ X ⟶ h (lfp h) ⊆ X by sorry
3    have h (lfp h) ⊆ lfp h by apply_theorem lfp_greatest from 2
4    have lfp h ⊆ h (lfp h) by sorry
5    show h (lfp h) = lfp h by apply_theorem subset_antisym from 3, 4
```

Save | Reset | Cancel

```
OK. 2 gap(s) remaining.
<3/3>  Current state

 introduction on 2
```

Example 3

theorem lfp_unfold:

```
bnd_mono h ⟶ h (lfp h) = lfp h
```

```
0     assume bnd_mono h
1     have ∀W. ∀X. W ⊆ X ⟶ h W ⊆ h X by rewrite_fact bnd_mono_def from 0
2     have ∀X. h X ⊆ X ⟶ h (lfp h) ⊆ X with
2.0     fix X :: 'a set
2.1     assume h X ⊆ X
2.2     show h (lfp h) ⊆ X by sorry
3     have h (lfp h) ⊆ lfp h by apply_theorem lfp_greatest from 2
4     have lfp h ⊆ h (lfp h) by sorry
5     show h (lfp h) = lfp h by apply_theorem subset_antisym from 3, 4
```

Save | Reset | Cancel

```
OK. 2 gap(s) remaining.
<4/4>  Current state

lfp_def (r): h (⋂{X. h X ⊆ X}) ⊆ X

subsetI (b): ∀x. x ∈ h (lfp h) ⟶ x ∈ X

subset_trans (b)
```

Example 3

theorem lfp_unfold: ✓

bnd_mono h ⟶ h (lfp h) = lfp h

```
0     assume bnd_mono h
1     have ∀W. ∀X. W ⊆ X ⟶ h W ⊆ h X by rewrite_fact bnd_mono_def from 0
2     have ∀X. h X ⊆ X ⟶ h (lfp h) ⊆ X with
2.0     fix X :: 'a set
2.1     assume h X ⊆ X
2.2     have h (lfp h) ⊆ h X by sorry
2.3     show h (lfp h) ⊆ X by sorry
3     have h (lfp h) ⊆ lfp h by apply_theorem lfp_greatest from 2
4     have lfp h ⊆ h (lfp h) by sorry
5     show h (lfp h) = lfp h by apply_theorem subset_antisym from 3, 4
```

Save Reset Cancel

OK. 3 gap(s) remaining.

<5/5> Current state

Apply fact (b): lfp h ⊆ X

lfp_def (r): h (⋂{X. h X ⊆ X}) ⊆ h X

forall elimination

Example 3

theorem lfp_unfold: ✓

bnd_mono h ⟶ h (lfp h) = lfp h

```
0    assume bnd_mono h
1    have ∀W. ∀X. W ⊆ X ⟶ h W ⊆ h X by rewrite_fact bnd_mono_def from 0
2    have ∀X. h X ⊆ X ⟶ h (lfp h) ⊆ X with
2.0    fix X :: 'a set
2.1    assume h X ⊆ X
2.2    have lfp h ⊆ X by sorry
2.3    have h (lfp h) ⊆ h X by apply_fact from 1, 2.2
2.4    show h (lfp h) ⊆ X by sorry
3    have h (lfp h) ⊆ lfp h by apply_theorem lfp_greatest from 2
4    have lfp h ⊆ h (lfp h) by sorry
5    show h (lfp h) = lfp h by apply_theorem subset_antisym from 3, 4
```

OK. 3 gap(s) remaining.

<6/6> Current state

lfp_lowerbound (b): (solves)

Example 3

theorem lfp_unfold:

```
bnd_mono h ⟶ h (lfp h) = lfp h
```

```
0    assume bnd_mono h
1    have ∀W. ∀X. W ⊆ X ⟶ h W ⊆ h X by rewrite_fact bnd_mono_def from 0
2    have ∀X. h X ⊆ X ⟶ h (lfp h) ⊆ X with
2.0    fix X :: 'a set
2.1    assume h X ⊆ X
2.2    have lfp h ⊆ X by apply_theorem_for lfp_lowerbound, {a: 'a}, {A: X, h: h} from 2.1
2.3    have h (lfp h) ⊆ h X by apply_fact from 1, 2.2
2.4    show h (lfp h) ⊆ X by sorry
3    have h (lfp h) ⊆ lfp h by apply_theorem lfp_greatest from 2
4    have lfp h ⊆ h (lfp h) by sorry
5    show h (lfp h) = lfp h by apply_theorem subset_antisym from 3, 4
```

Save Reset Cancel

```
OK. 2 gap(s) remaining.
<7/7>  Current state

 subset_trans (b): (solves)
```

Example 3

theorem lfp_unfold: ✎ ✔

bnd_mono h ⟶ h (lfp h) = lfp h

```
0     assume bnd_mono h
1     have ∀W. ∀X. W ⊆ X ⟶ h W ⊆ h X by rewrite_fact bnd_mono_def from 0
2     have ∀X. h X ⊆ X ⟶ h (lfp h) ⊆ X with
2.0     fix X :: 'a set
2.1     assume h X ⊆ X
2.2     have lfp h ⊆ X by apply_theorem_for lfp_lowerbound, {a: 'a}, {A: X, h: h} from 2.1
2.3     have h (lfp h) ⊆ h X by apply_fact from 1, 2.2
2.4     show h (lfp h) ⊆ X by apply_theorem subset_trans from 2.3, 2.1
3     have h (lfp h) ⊆ lfp h by apply_theorem lfp_greatest from 2
4     have lfp h ⊆ h (lfp h) by sorry
5     show h (lfp h) = lfp h by apply_theorem subset_antisym from 3, 4
```

Save | Reset | Cancel

OK. 1 gap(s) remaining.

<8/8> Current state

lfp_def (r): ⋂{X. h X ⊆ X} ⊆ h (⋂{X. h X ⊆ X})

lfp_lowerbound (b): h (h (lfp h)) ⊆ h (lfp h)

Thesis (still to be fully demonstrated)

Improvements to user interface can significantly increase efficiency and make proof assistants easier to learn.

Thesis (still to be fully demonstrated)

Improvements to user interface can significantly increase efficiency and make proof assistants easier to learn.

Corollary

More effort should be devoted to user interface design in interactive theorem proving.

Future work

- Test the user interface on a larger body of theorems.

Future work

- Test the user interface on a larger body of theorems.
  - Currently underway for real analysis.

Future work

- Test the user interface on a larger body of theorems.
  - Currently underway for real analysis.
- Incorporate various proof automation.

Future work

- Test the user interface on a larger body of theorems.
  - Currently underway for real analysis.
- Incorporate various proof automation.
  - Can be easily represented as methods.

Future work

- Test the user interface on a larger body of theorems.
  - Currently underway for real analysis.
- Incorporate various proof automation.
  - Can be easily represented as methods.
- Machine learning for ordering suggestions.

Future work

- Test the user interface on a larger body of theorems.
  - Currently underway for real analysis.
- Incorporate various proof automation.
  - Can be easily represented as methods.
- Machine learning for ordering suggestions.
  - Main obstacle is the need to build a large library for learning.