Artificial Intelligence and Domain-Specific Languages for Inductive Theorem Proving

PhD defence of Yutaka Nagashima, 17.06.2021

- Why theorem proving?

- Why theorem proving?
- What is Isabelle/HOL?

- Why theorem proving?
- What is Isabelle/HOL?
- What are tactics?

- Why theorem proving?
- What is Isabelle/HOL?
- What are tactics?

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

File Browser Documentation

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
  apply(induct xs ys rule: rev2.induct)
  apply auto done
```

- Why theorem proving?
- What is Isabelle/HOL?
- What are tactics?

- Why theorem proving?
- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

- Why theorem proving?
- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

we are convinced that substantial progress in ITP will take time.

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

When applying AI for theorem proving, keep in mind…

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

When applying AI for theorem proving, keep in mind…

Algorithmic approaches work well, but not always.

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

When applying AI for theorem proving, keep in mind…

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

we are convinced that substantial progress in ITP will take time.

When applying AI for theorem proving,

Algorithmic approaches work well, but no

There are many good ways to prove one t

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

Documentation

File Browser

```
5  primrec rev1::"'a list ⇒ 'a list" where
6    "rev1  []       = []"
7  | "rev1 (x # xs) = rev1 xs @ [x]"
8
9  fun rev2::"'a list ⇒ 'a list ⇒ 'a list"
10   "rev2  []       ys = ys"
11 | "rev2 (x # xs) ys = rev2 xs (x # ys)"
12
13 theorem "rev2 xs ys = rev1 xs @ ys"
14   apply(induct xs ys rule: rev2.induct)
15   apply auto done
```

Proof state Auto up

```
proof (prove)
goal:
No subgoals!
```

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

When applying AI for theorem proving,

Algorithmic approaches work well, but no

There are many good ways to prove one t

```
11 | "rev2 (x # xs) ys = rev2 xs (x # ys)"
12
13 theorem "rev2 xs ys = rev1 xs @ ys"
14   apply(induct xs arbitrary: ys)
15   apply auto done
```

Proof state Auto up

```
proof (prove)
goal:
No subgoals!
```

Output Query Sledgehammer Symbols

15,13 (344/353)

```
13 theorem "rev2 xs ys = rev1 xs @ ys"
14   apply(induct xs ys rule: rev2.induct)
15   apply auto done
```

Proof state Auto up

```
proof (prove)
goal:
No subgoals!
```

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

When applying AI for theorem proving, keep in mind…

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

When applying AI for theorem proving, keep in mind…

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

When applying AI for theorem proving, keep in mind…

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

Path to automatic induction

Path to automatic induction

Step 1: Tactic recommendation (PaMpeR).

Path to automatic induction

Step 1: Tactic recommendation (PaMpeR).

Step 2: Tactic argument recommendation.

Path to automatic induction

Step 1: Tactic recommendation (PaMpeR).

Step 2: Tactic argument recommendation.

a: Guided backtracking search with tracing (PSL).

Path to automatic induction

Step 1: Tactic recommendation (PaMpeR).

Step 2: Tactic argument recommendation.

- a: Guided backtracking search with tracing (PSL).
- b: Logical Feature Extraction (LiFtEr).
- c: smart_induct using LiFtEr.

Path to automatic induction

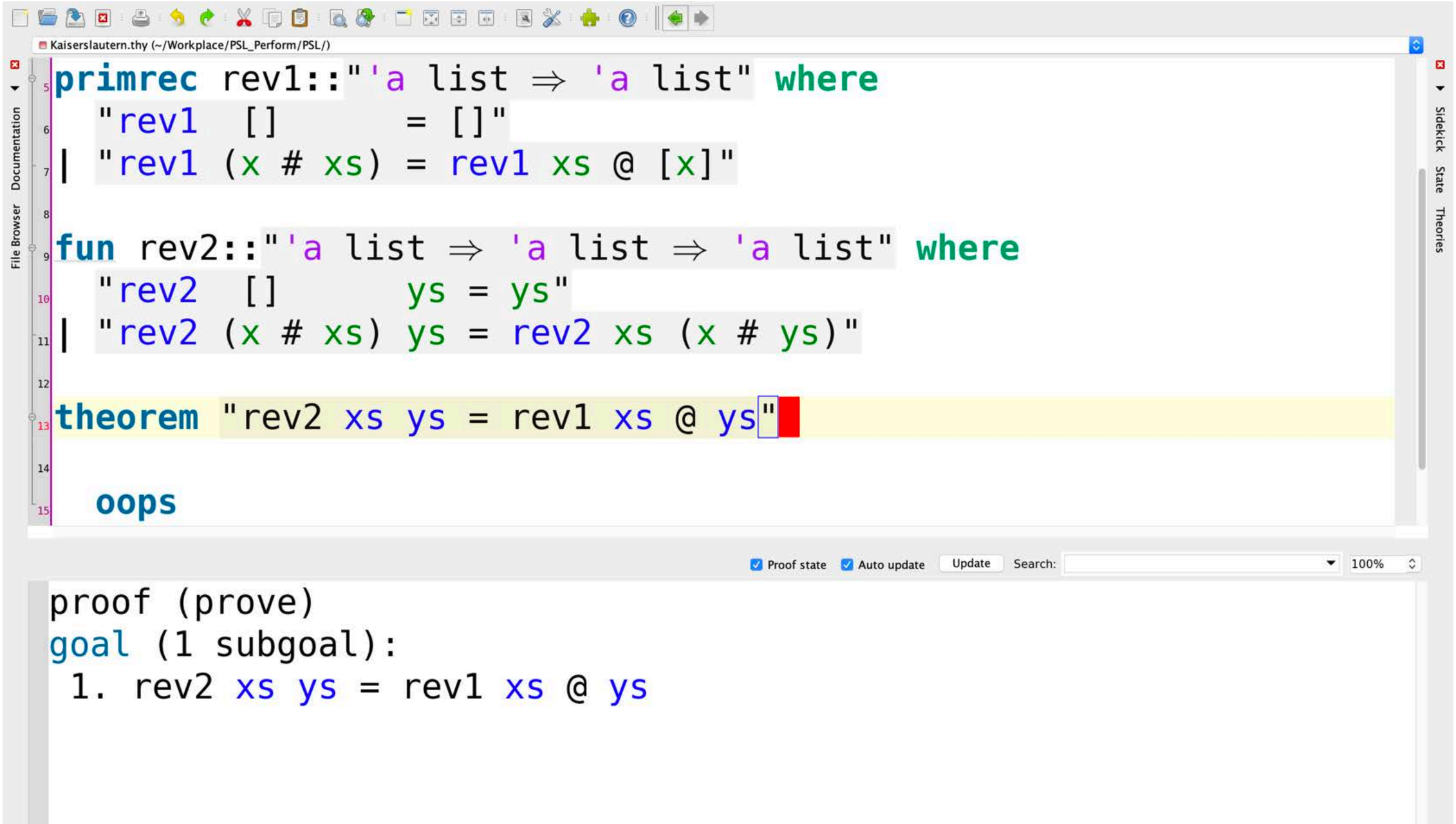
Step 1: Tactic recommendation (PaMpeR).

Step 2: Tactic argument recommendation.

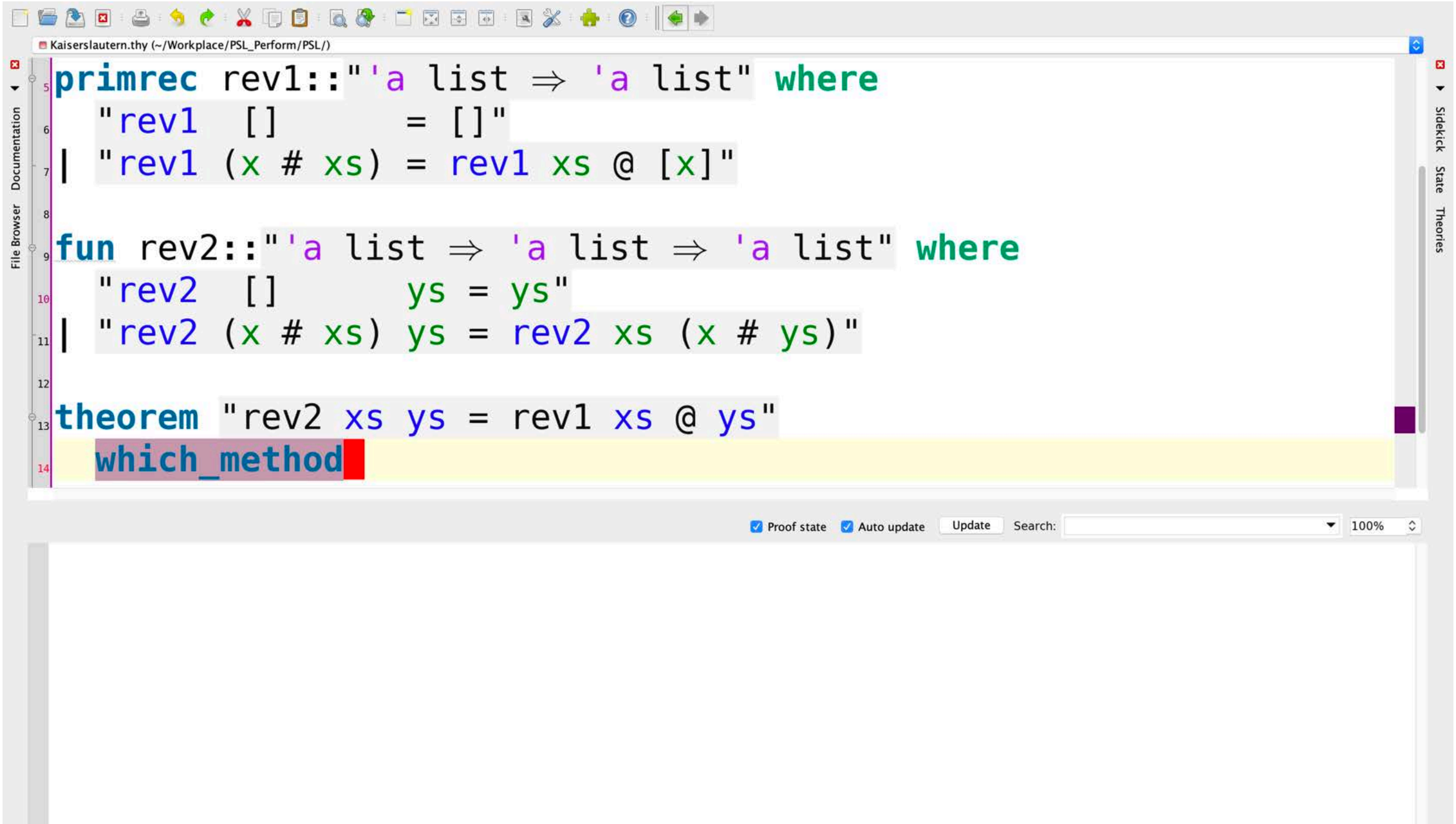
- a: Guided backtracking search with tracing (PSL).
- b: Logical Feature Extraction (LiFtEr).
- c: smart_induct using LiFtEr.

Step 3: Putting things together.

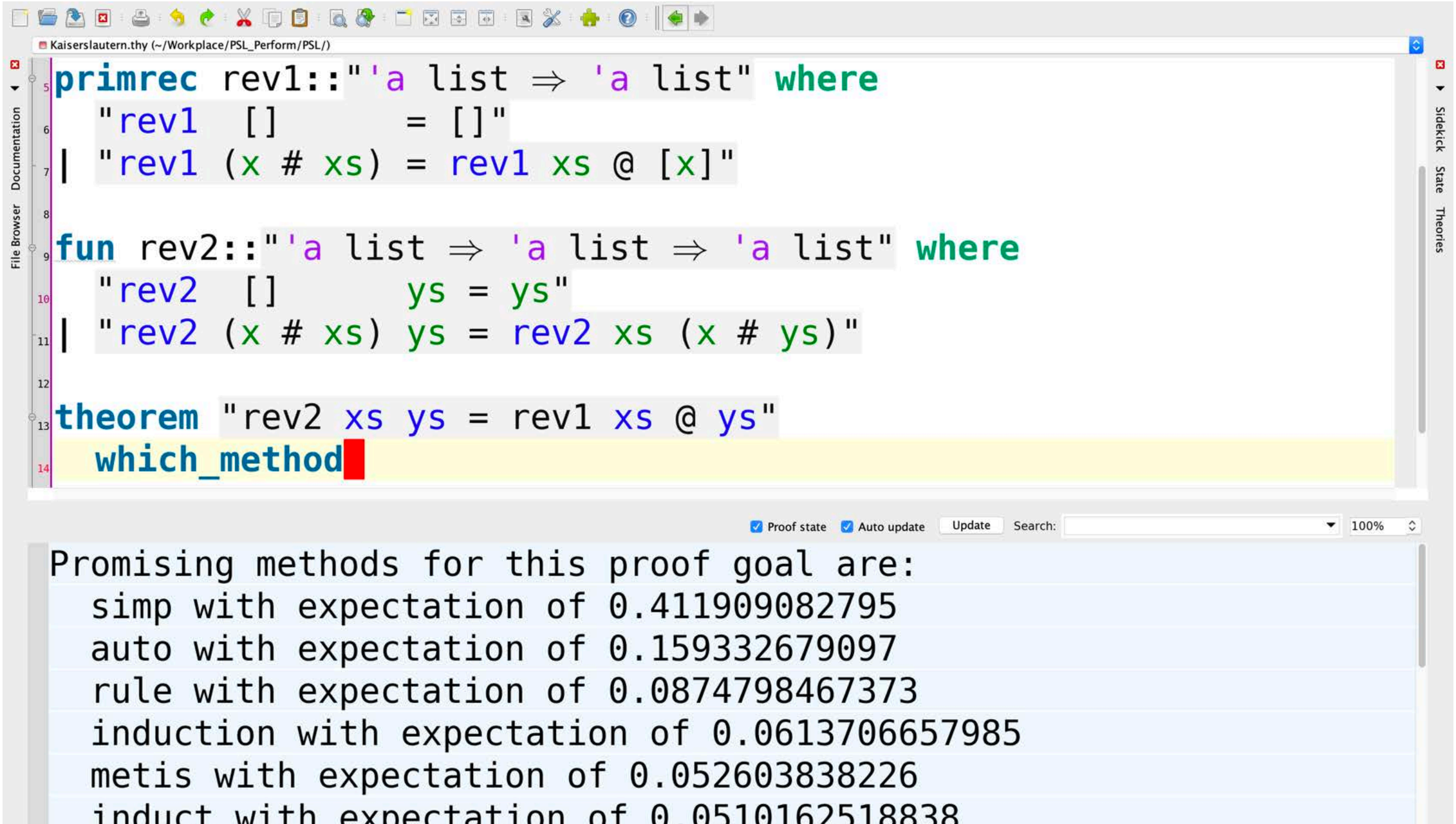
Step1: Proof Method Recommendation (PaMpeR)



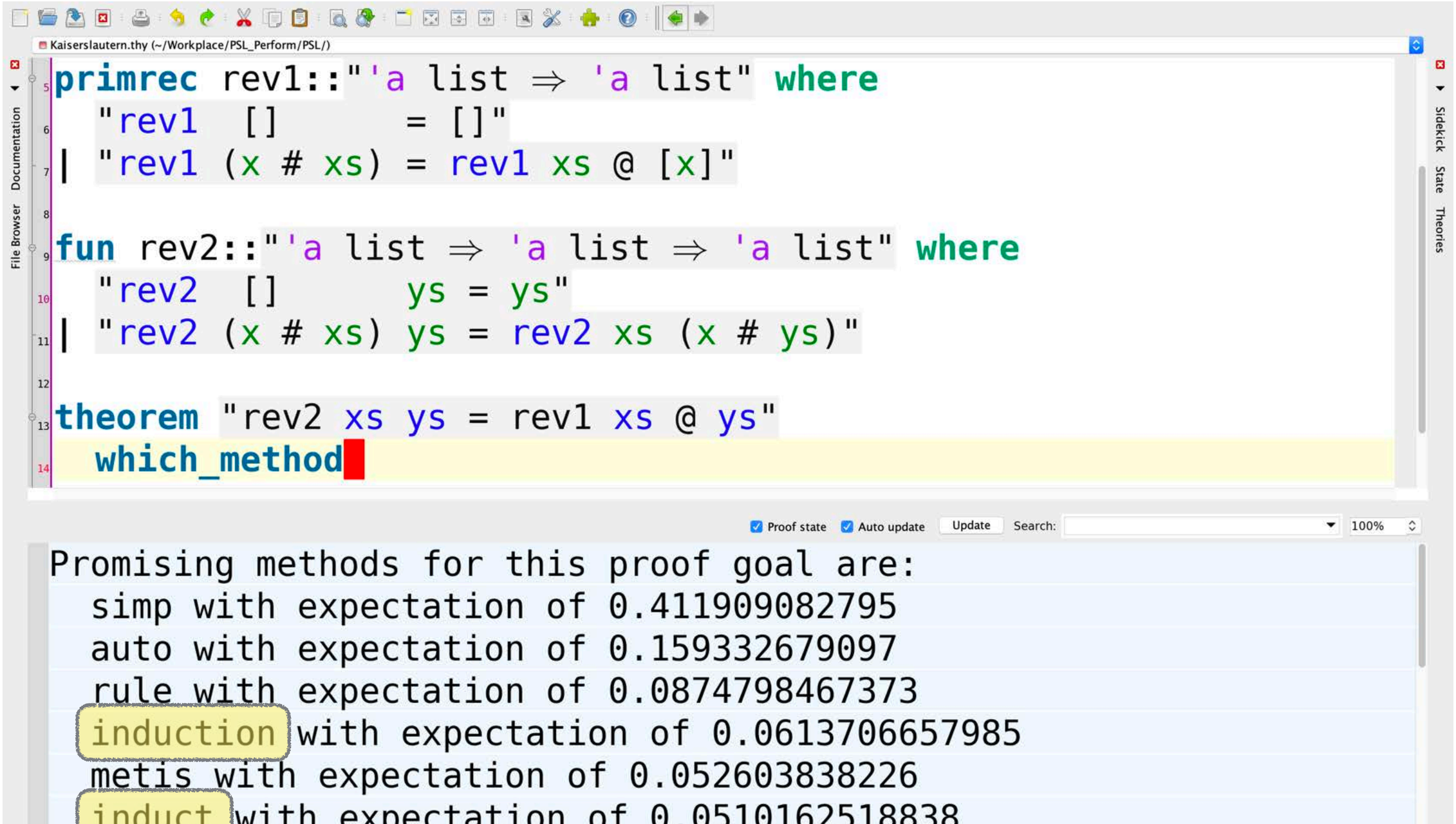
Step1: Proof Method Recommendation (PaMpeR)



Step1: Proof Method Recommendation (PaMpeR)



Step1: Proof Method Recommendation (PaMpeR)



Step1: Proof Method Recommendation (PaMpeR)

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"
```

Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

```
which_method
```

Proof state | Auto update | Update | Search: | 100%

```
Promising methods for this proof goal are:
  simp with expectation of 0.411909082795
  auto with expectation of 0.159332679097
  rule with expectation of 0.0874798467373
  induction with expectation of 0.0613706657985
  metis with expectation of 0.052603838226
  induct with expectation of 0.0510162518838
```

Step1: Proof Method Recommendation (PaMpeR)

preparation phase

How does PaMpeR work?

recommendation phase

Step1: Proof Method Recommendation (PaMpeR)

preparation phase

large proof corpora

AFP and standard library

How does PaMpeR work?

recommendation phase

proof state

proof engineer

Step1: Proof Method Recommendation (PaMpeR)

preparation phase

large proof corpora

AFP and standard library

How does PaMpeR work?

recommendation phase

proof state

proof engineer

Statistics

Archive of Formal Proofs (https://www.isa-afp.org)

- Home
- About
- Submission
- Updating Entries
- Using Entries
- Search
- Statistics

Statistics

Number of Articles: 468
Number of Authors: 313
Number of lemmas: ~128,900
Lines of Code: ~2,170,300

Most used AFP articles:

	Name	Used by ? articles
1.	Collections	15
2.	List-Index	14
3.	Coinductive	12
	Regular-Sets	12

Step1: Proof Method Recommendation (PaMpeR)

preparation phase

large proof corpora

AFP and standard library

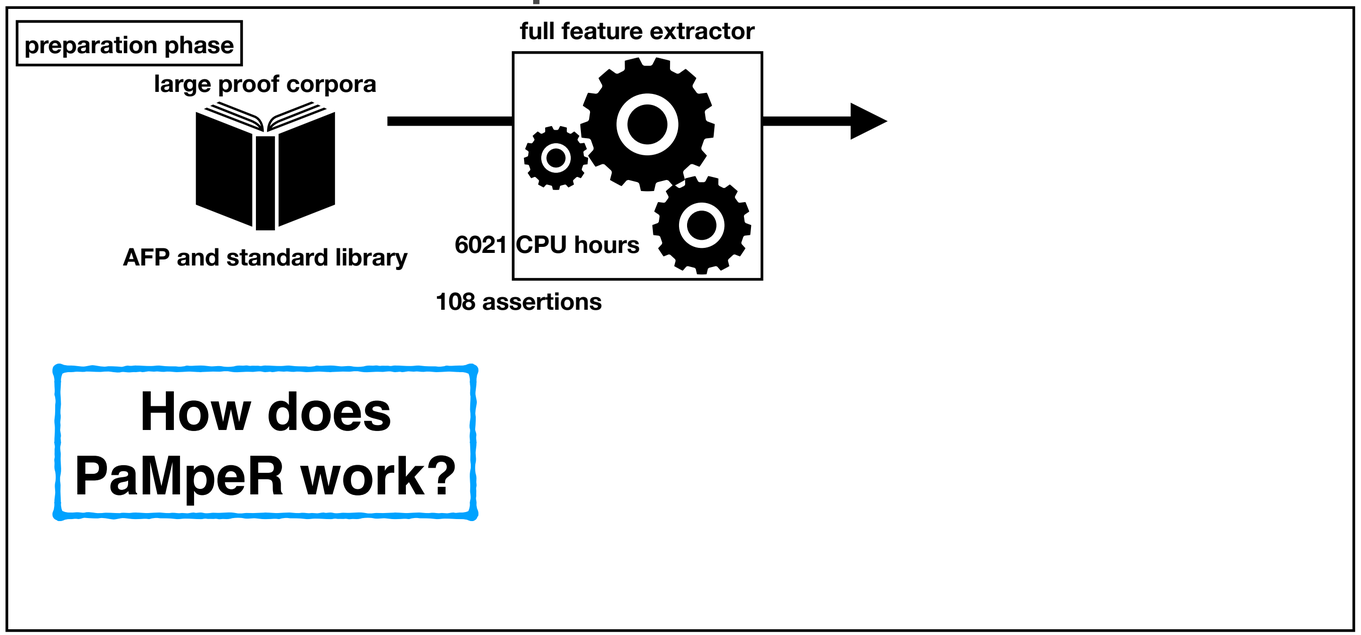
How does PaMpeR work?

recommendation phase

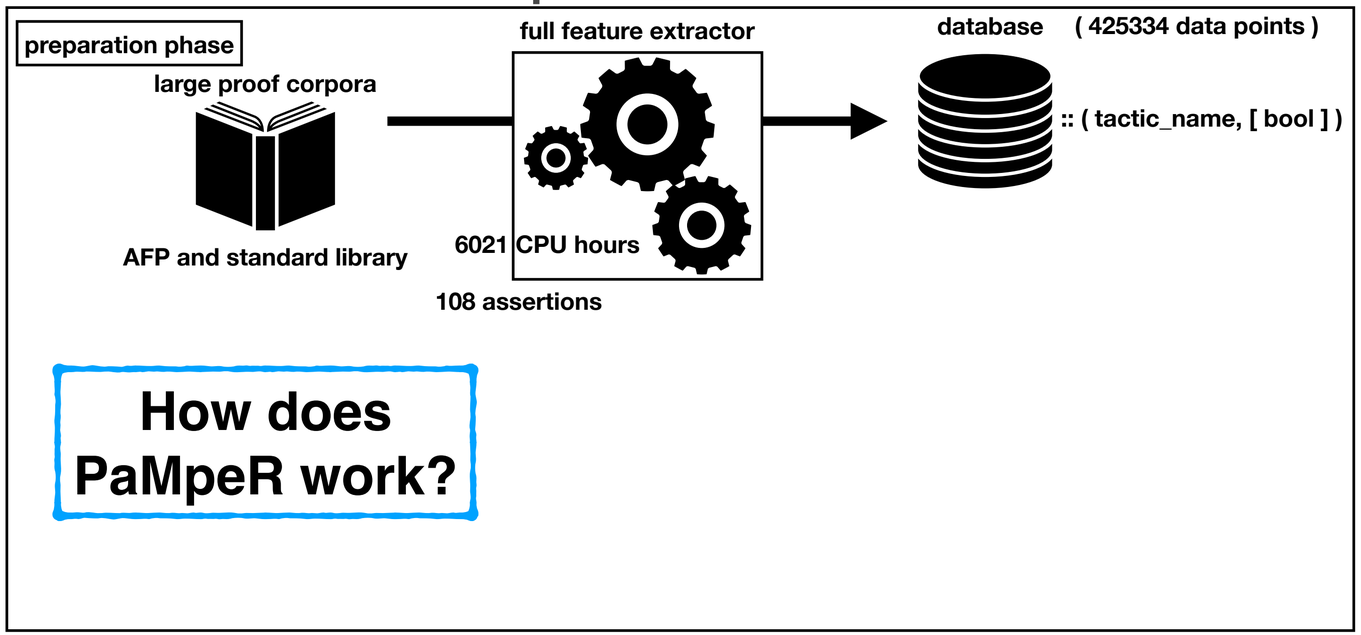
proof state

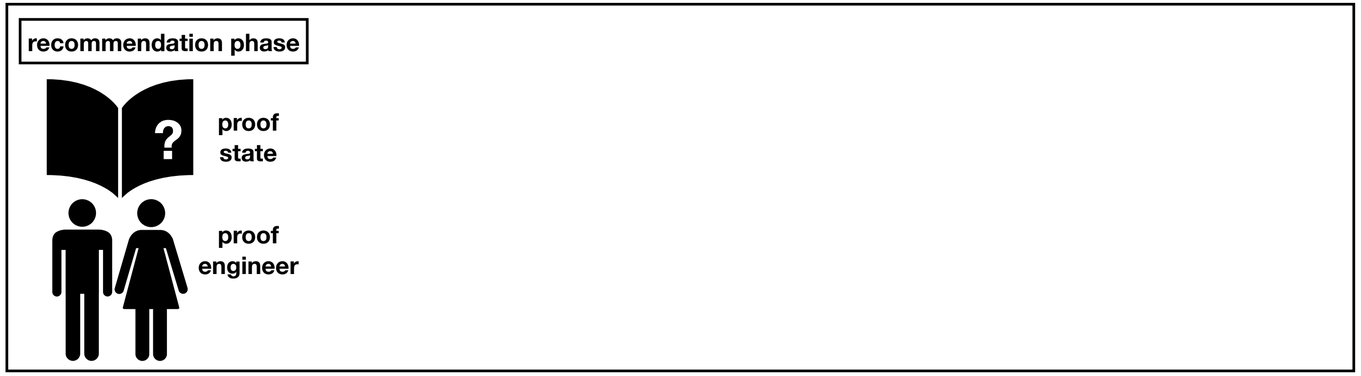
proof engineer

Step1: Proof Method Recommendation (PaMpeR)

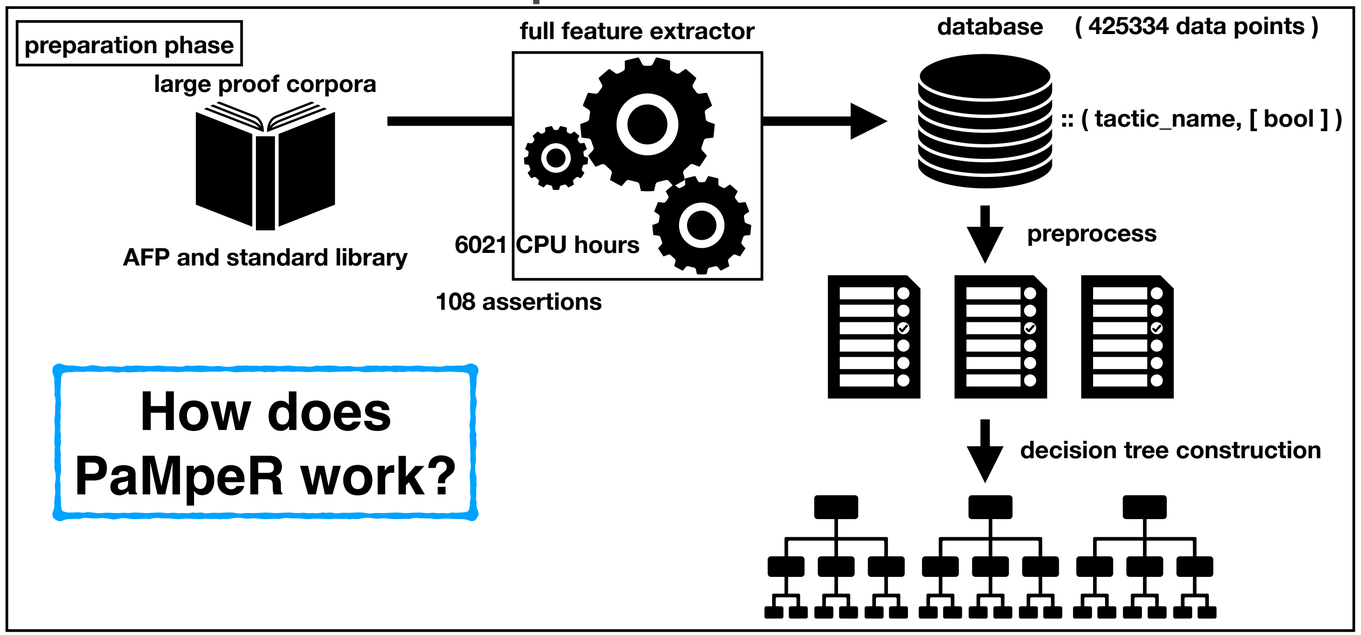


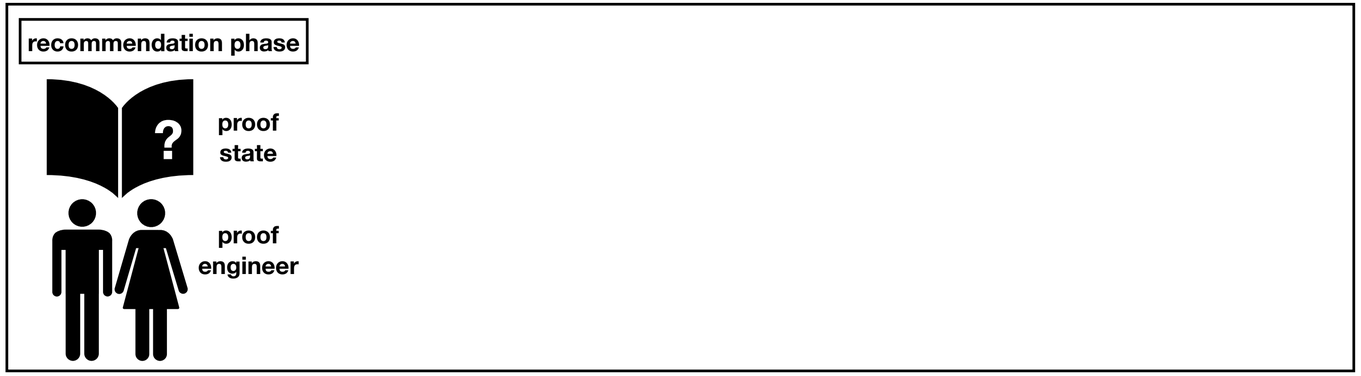
Step1: Proof Method Recommendation (PaMpeR)



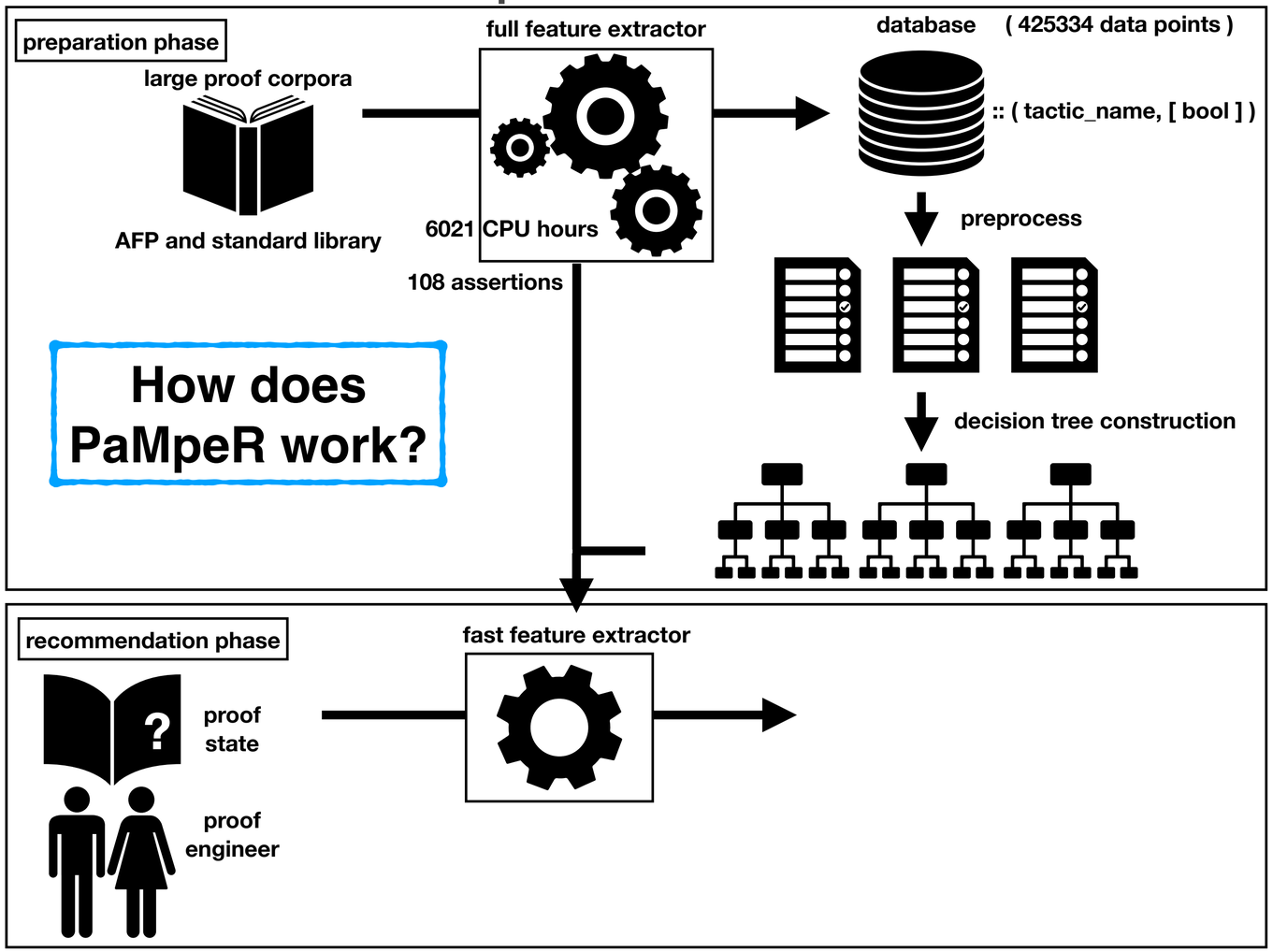


Step1: Proof Method Recommendation (PaMpeR)

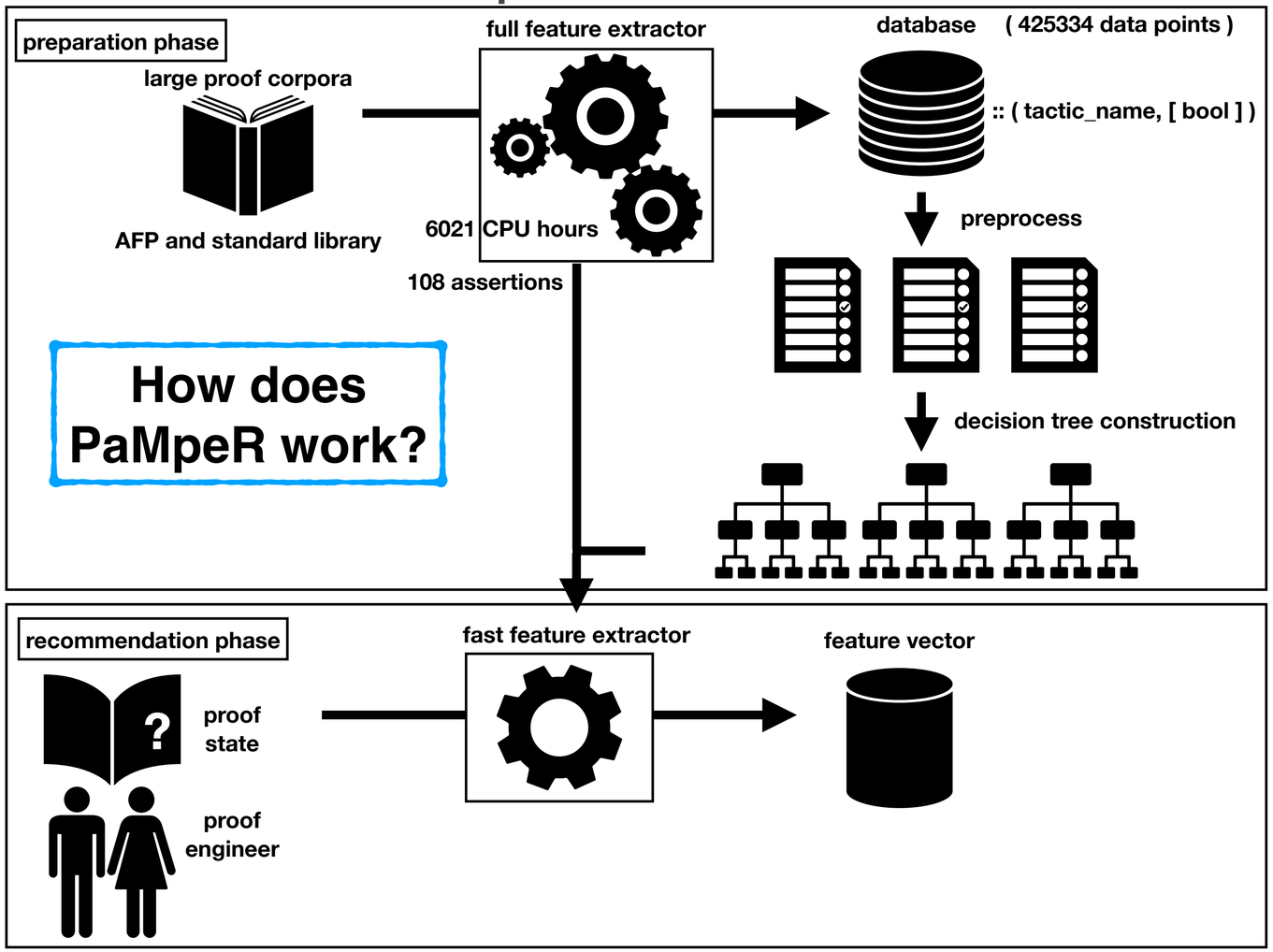




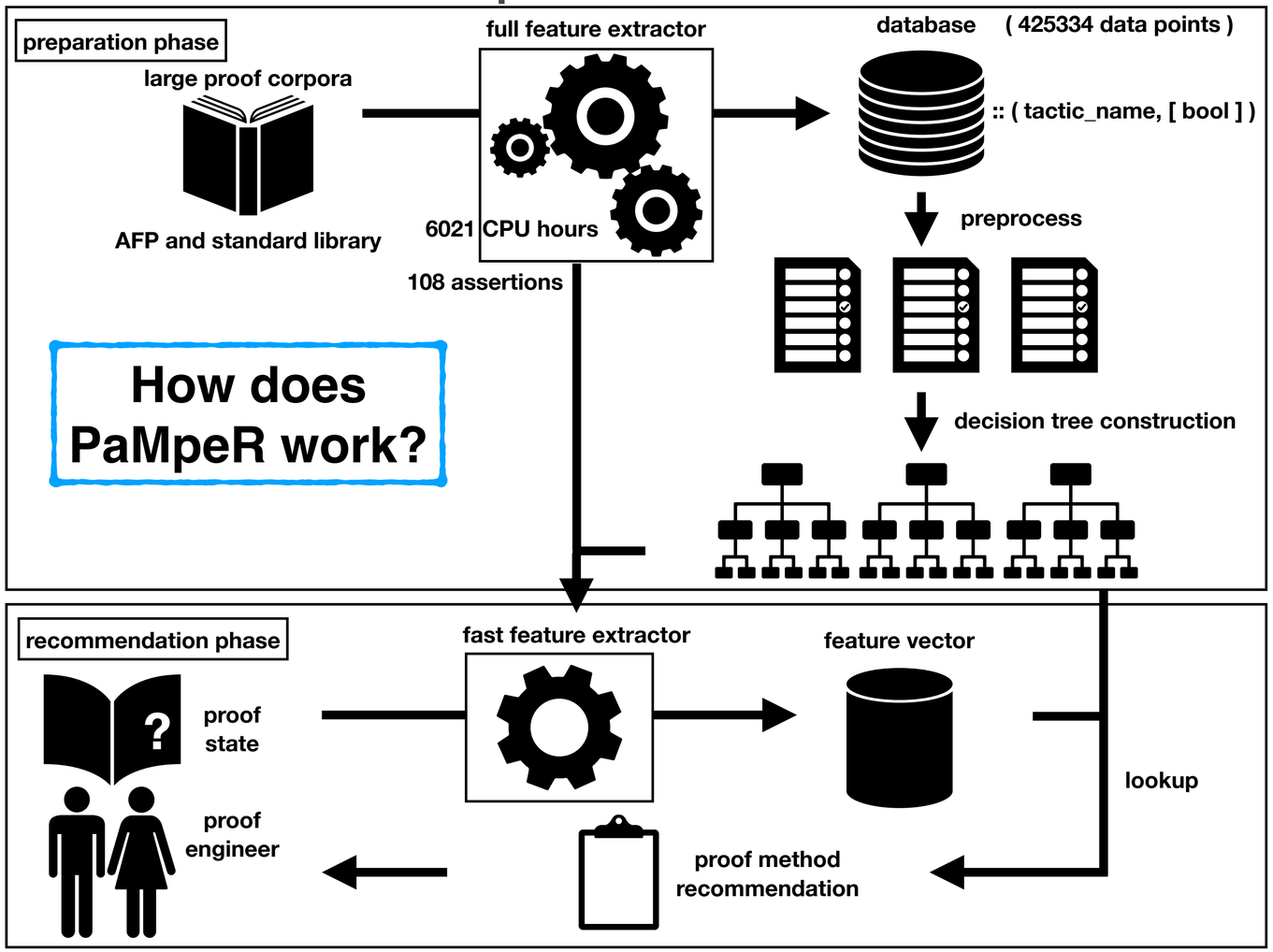
Step1: Proof Method Recommendation (PaMpeR)



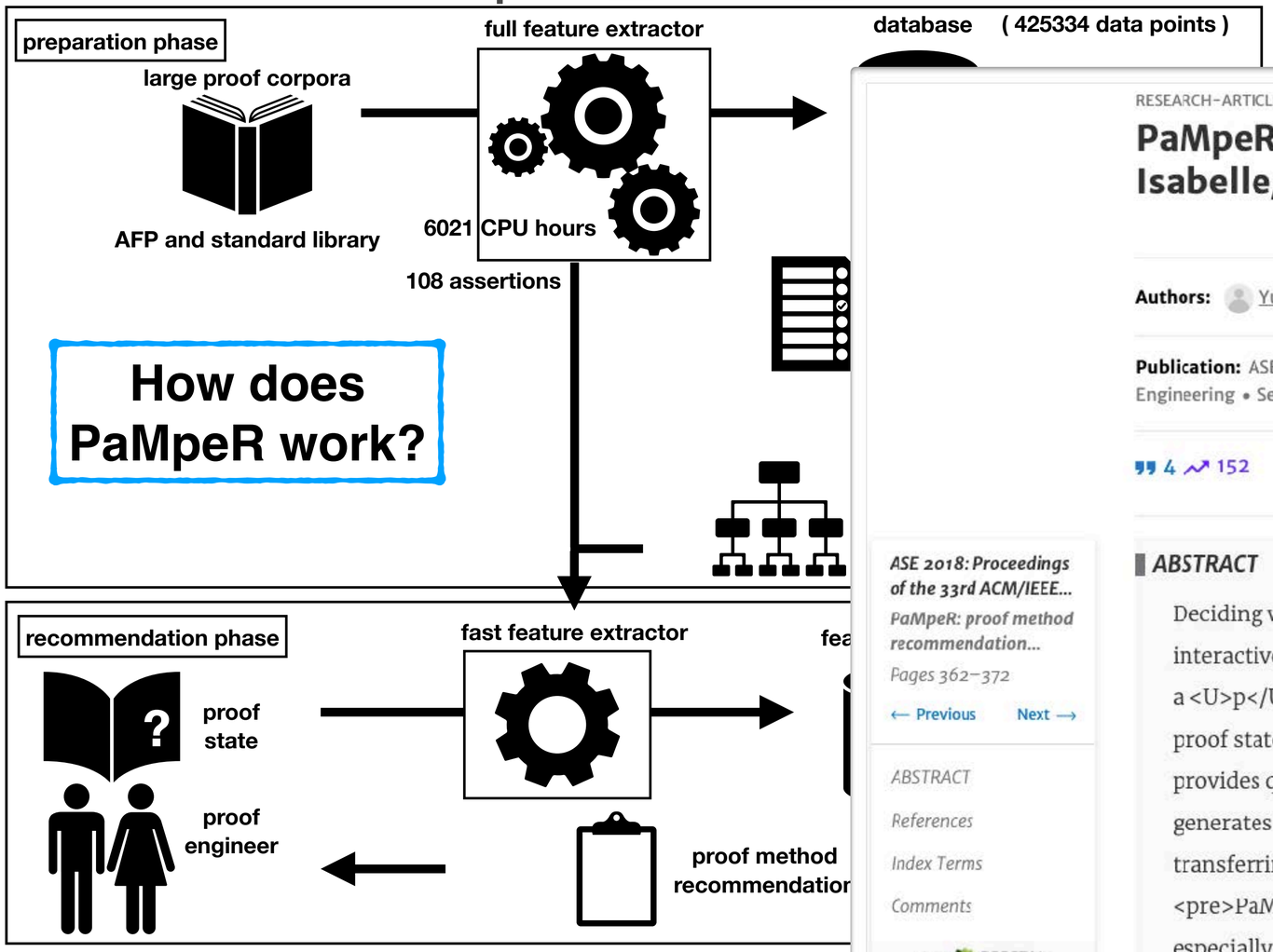
Step1: Proof Method Recommendation (PaMpeR)



Step1: Proof Method Recommendation (PaMpeR)



Step1: Proof Method Recommendation (PaMpeR)



RESEARCH-ARTICLE

PaMpeR: proof method recommendation system for Isabelle/HOL

Authors: Yutaka Nagashima, Yilun He Authors Info & Affiliations

Publication: ASE 2018: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering • September 2018 • Pages 362–372 • https://doi.org/10.1145/3238147.3238210

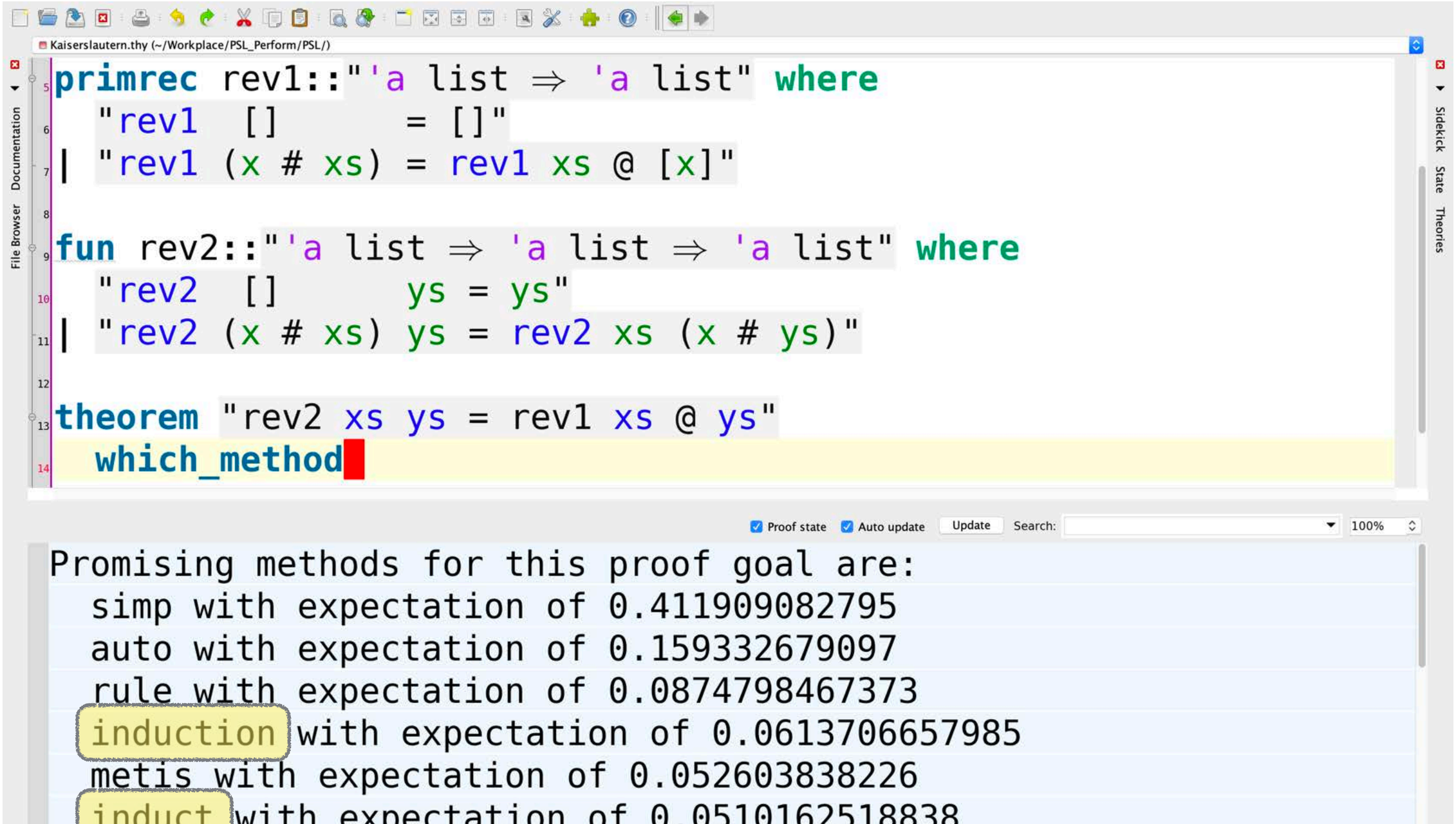
ABSTRACT

Deciding which sub-tool to use for a given proof state requires expertise specific to each interactive theorem prover (ITP). To mitigate this problem, we present <pre>PaMpeR</pre>, a <U>p</U>roof <U>m</U>ethod <U>r</U>ecommendation system for Isabelle/HOL. Given a proof state, <pre>PaMpeR</pre> recommends proof methods to discharge the proof goal and provides qualitative explanations as to why it suggests these methods. <pre>PaMpeR</pre> generates these recommendations based on existing hand-written proof corpora, thus transferring experienced users' expertise to new users. Our evaluation shows that <pre>PaMpeR</pre> correctly predicts experienced users' proof methods invocation especially when it comes to special purpose proof methods.

References

1. 2018. PSL with PGT: CICM2018 for Isabelle2017. (2018). https://github. com/data61/PSL/releases/tag/v0.1.1 To use PaMpeR, one first needs to install Isabelle/HOL, which is distributed at https://isabelle.in.tum.de/. PaMpeR: Proof Method Recommendation System for Isabelle/HOL ASE '18, September 3–7, 2018, Montpellier, France

Step1: Proof Method Recommendation (PaMpeR)



Step1: Proof Method Recommendation (PaMpeR)

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
  which_method
```

Proof state | Auto update | Update | Search: | 100%

```
Promising methods for this proof goal are:
  simp with expectation of 0.411909082795
  auto with expectation of 0.159332679097
  rule with expectation of 0.0874798467377
  induction with expectation of 0.0613706657985
  metis with expectation of 0.052603838226
  induct with expectation of 0.0510162518838
```

Arguments of tactics?

Path to automatic induction

Path to automatic induction

Step 1: Tactic recommendation.

Path to automatic induction

Step 1: Tactic recommendation.

Step 2: Tactic argument recommendation.

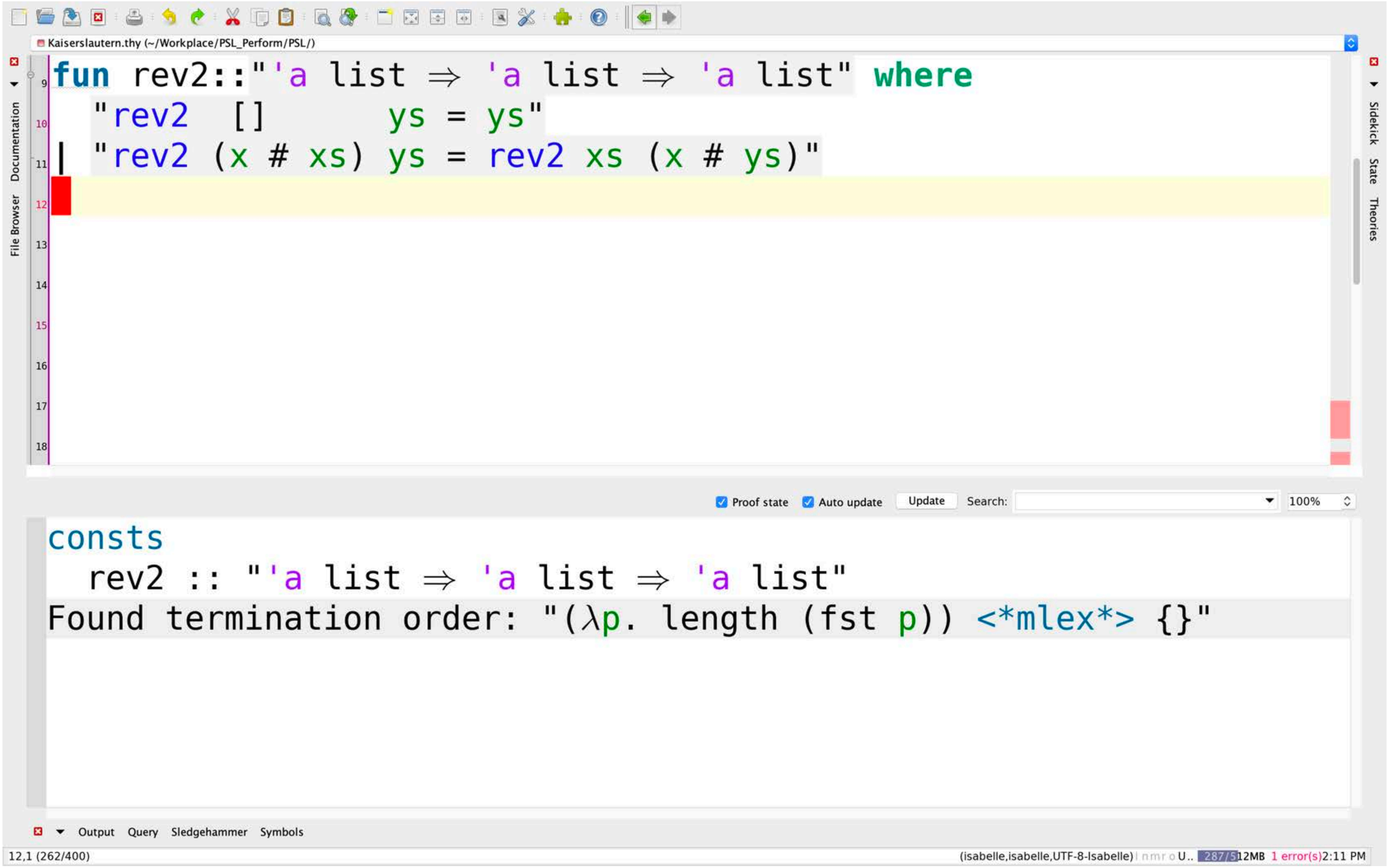
Path to automatic induction

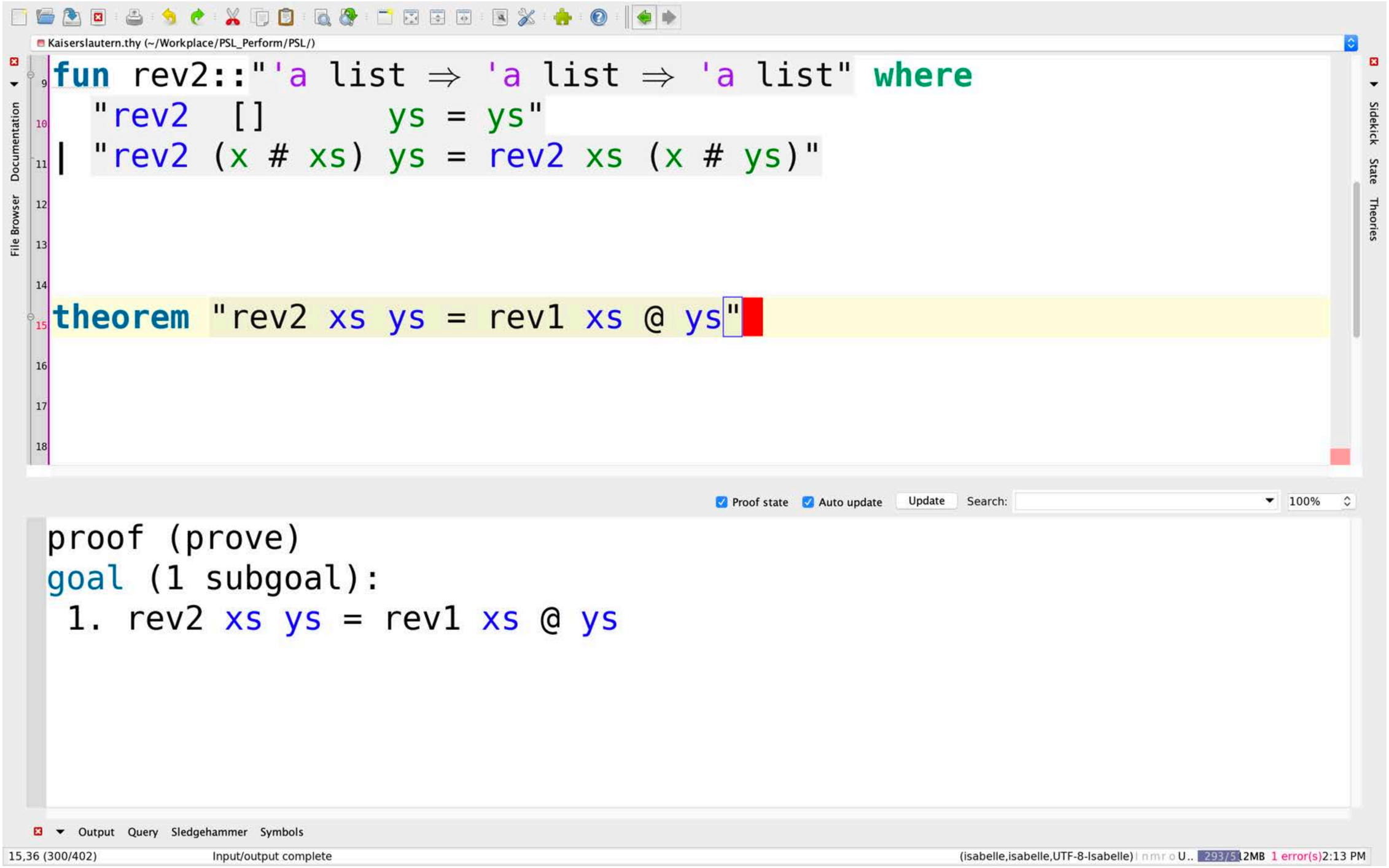
Step 1: Tactic recommendation.

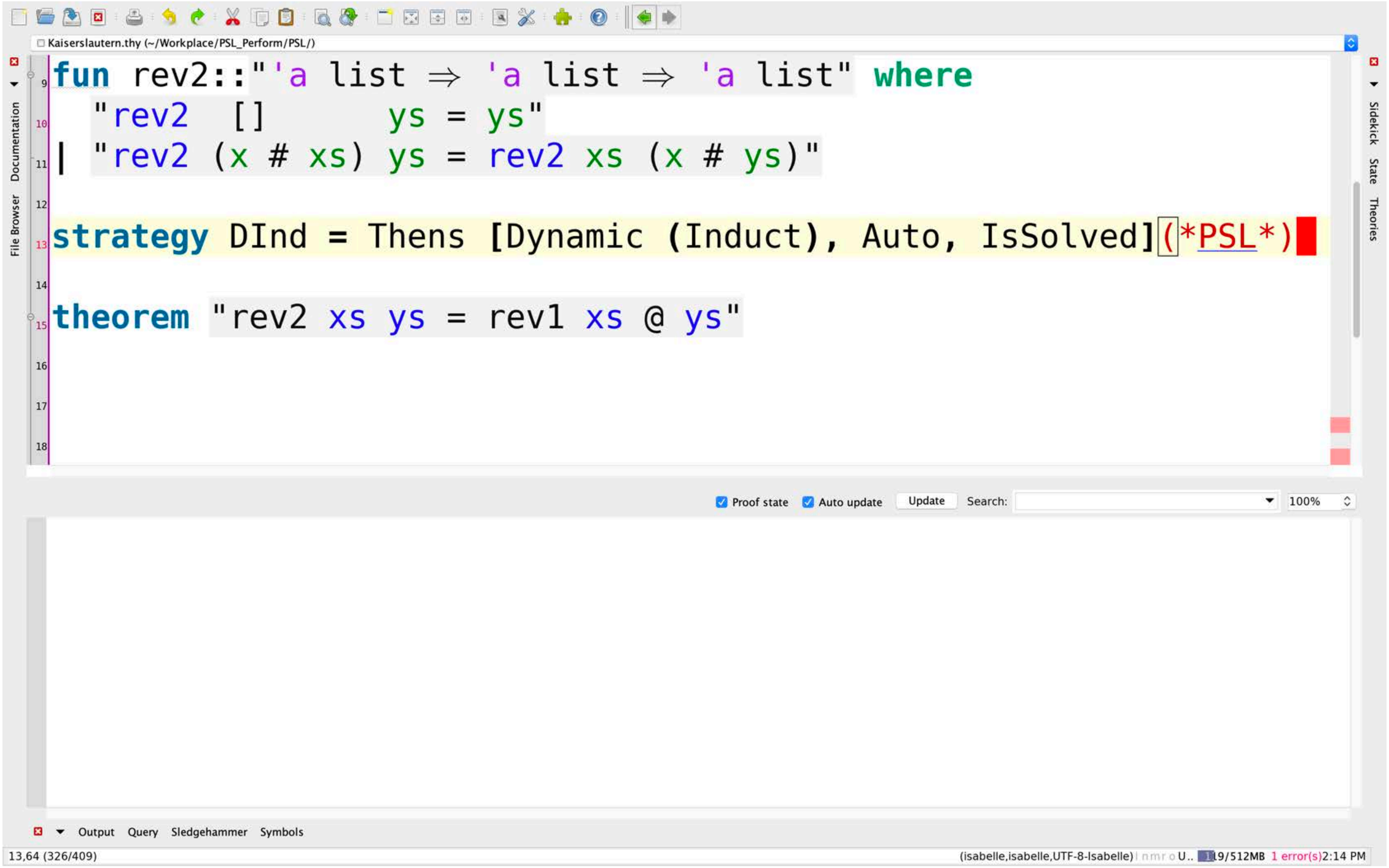
Step 2: Tactic argument recommendation.

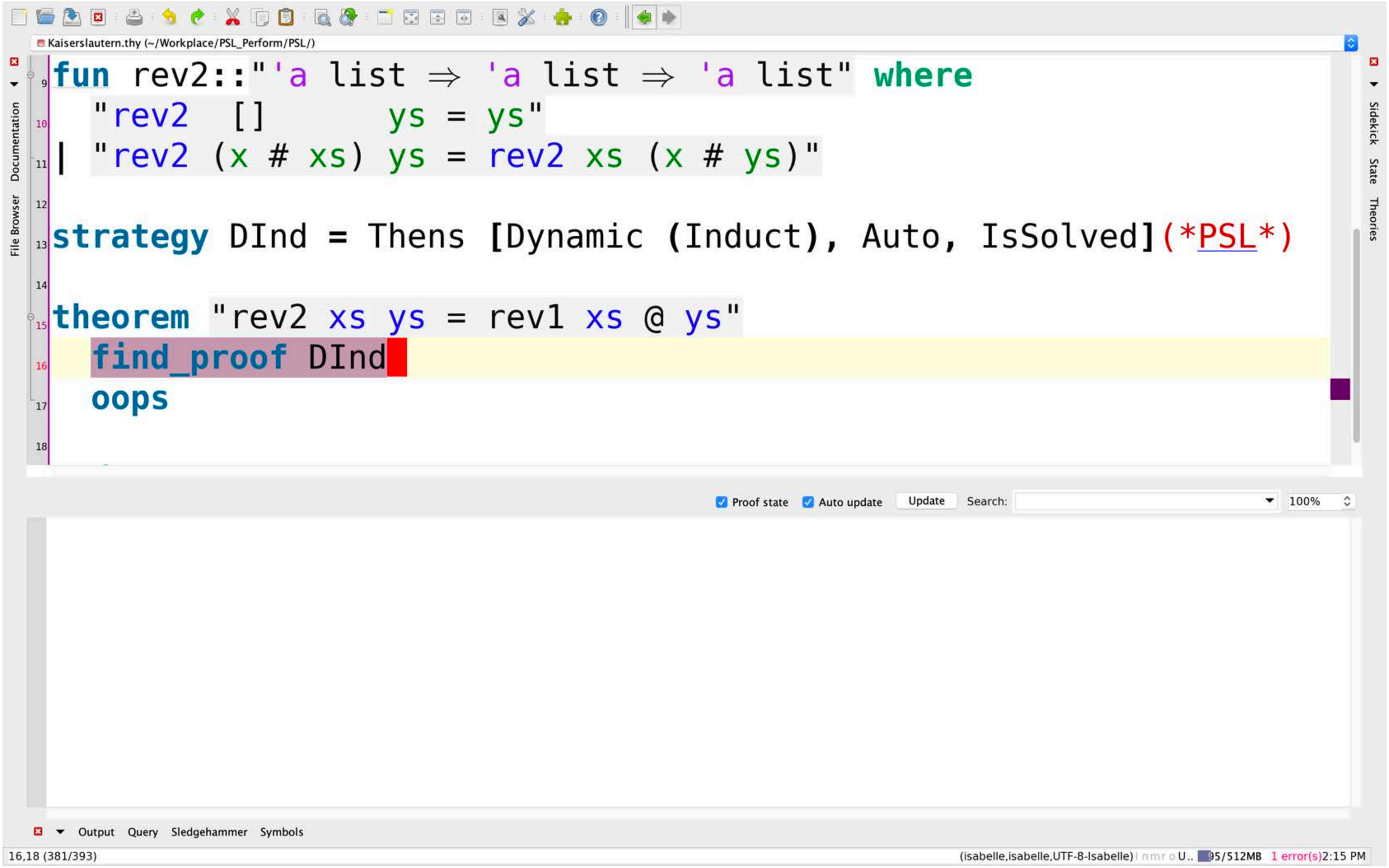
- a: Guided backtracking search with tracing (PSL).
- b: Logical Feature Extraction (LiFtEr).
- c: smart_induct using LiFtEr.

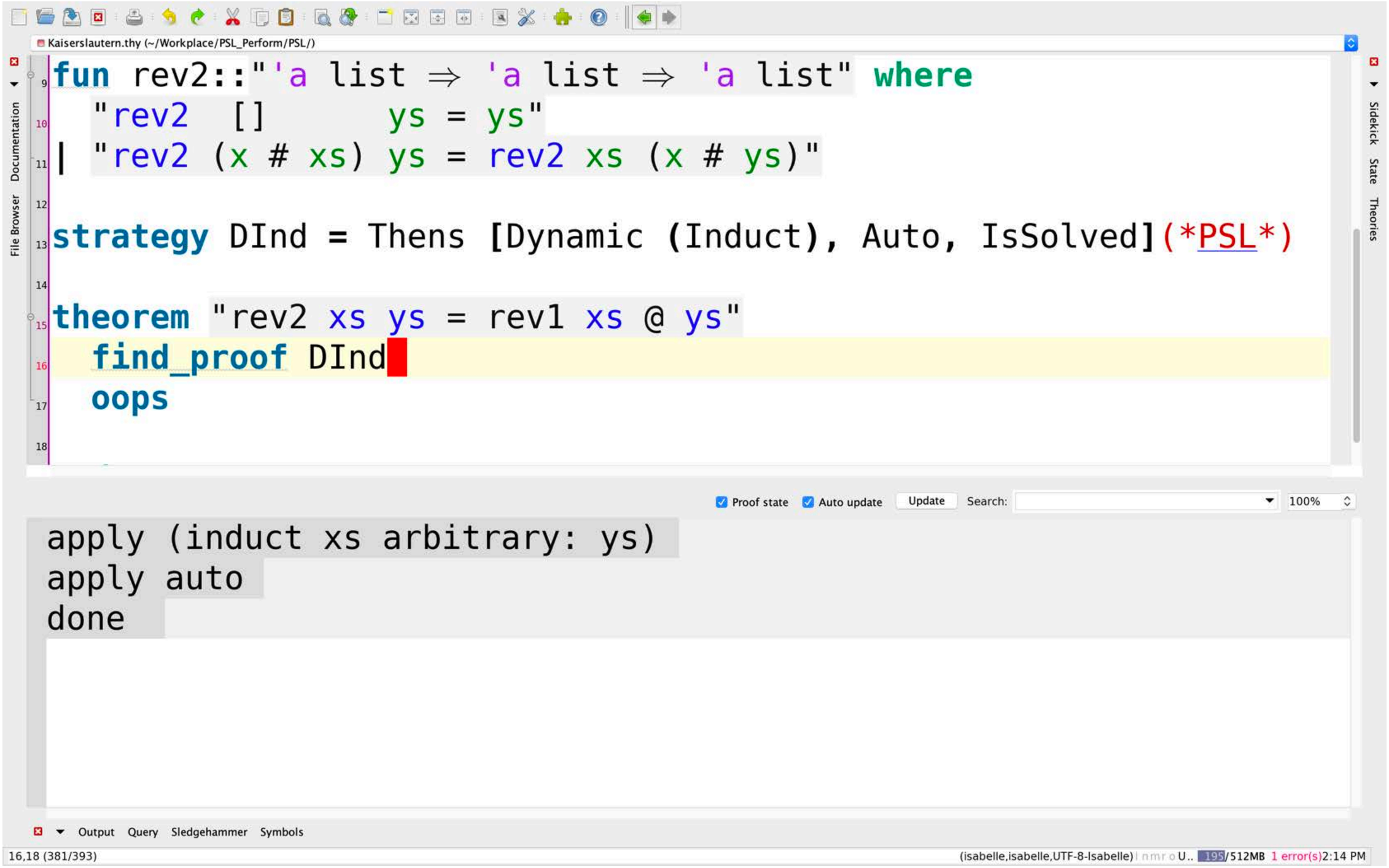
Step 3: Putting things together.


Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)
fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
"rev2 [] ys = ys"
"rev2 (x # xs) ys = rev2 xs (x # ys)"
Documentation
File Browser
Sidekick
State
Theories
Proof state
Auto update
Update
Search:
100%
consts
rev2 :: "'a list ⇒ 'a list ⇒ 'a list"
Found termination order: "(λp. length (fst p)) <*mlex*> {}"
Output
Query
Sledgehammer
Symbols
12,1 (262/400)
(isabelle,isabelle,UTF-8-Isabelle)
1 error(s)
2:11 PM

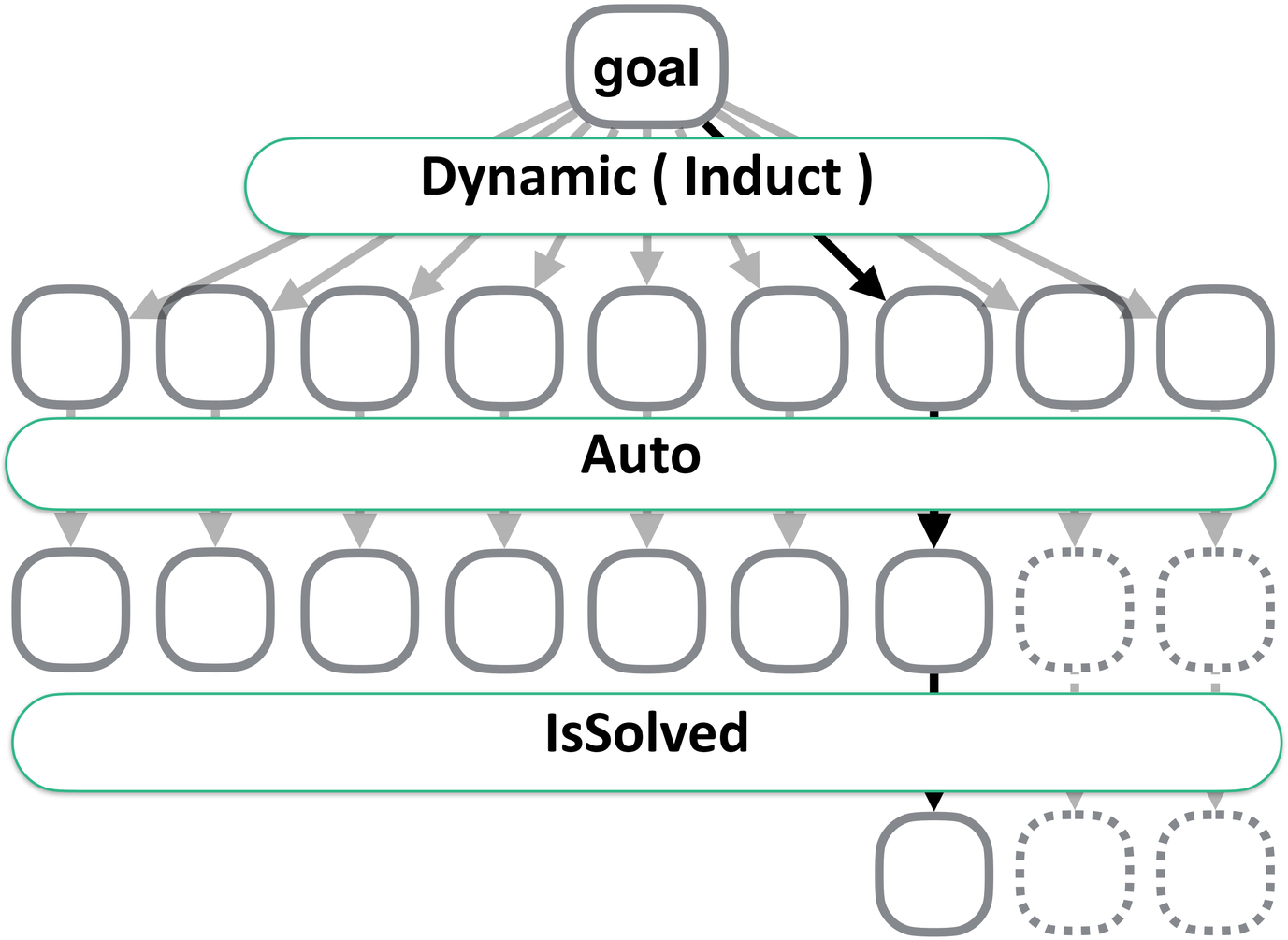

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)
fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
"rev2 [] ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"
theorem "rev2 xs ys = rev1 xs @ ys"
Documentation
File Browser
Sidekick
State
Theories
Proof state
Auto update
Update
Search:
100%
proof (prove)
goal (1 subgoal):
1. rev2 xs ys = rev1 xs @ ys
Output
Query
Sledgehammer
Symbols
15,36 (300/402)
Input/output complete
(isabelle,isabelle,UTF-8-Isabelle)
1 error(s)
2:13 PM


Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)
fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
"rev2 [] ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"
strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved](*PSL*)
theorem "rev2 xs ys = rev1 xs @ ys"
Documentation
File Browser
Sidekick
State
Theories
Proof state
Auto update
Update
Search:
100%
Output
Query
Sledgehammer
Symbols
13,64 (326/409)
(isabelle,isabelle,UTF-8-Isabelle)
1 error(s)
2:14 PM

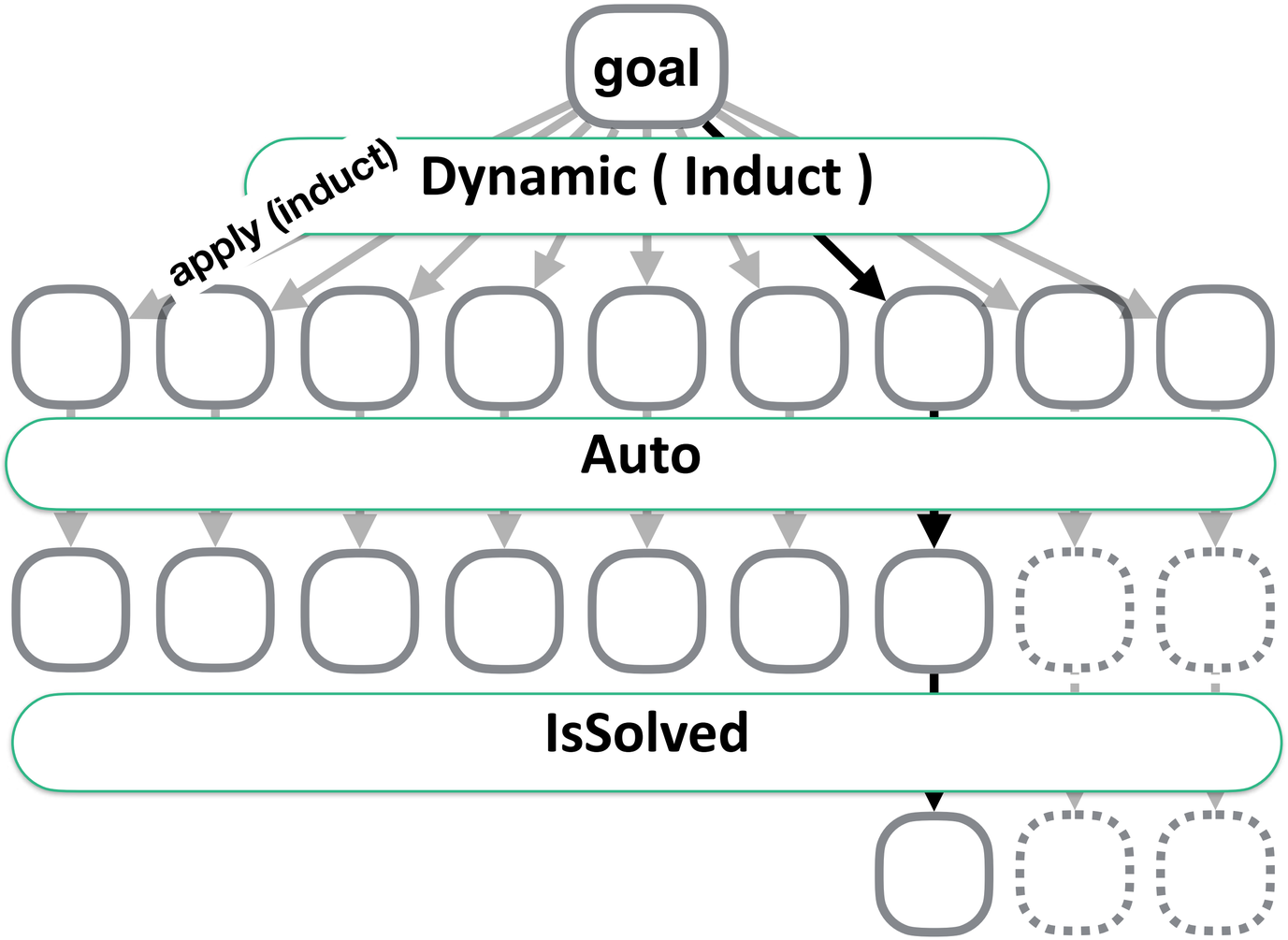

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)
fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
"rev2 [] ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"
strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved](*PSL*)
theorem "rev2 xs ys = rev1 xs @ ys"
find_proof DInd
oops
Documentation
File Browser
Sidekick
State
Theories
Proof state
Auto update
Update
Search:
100%
Output
Query
Sledgehammer
Symbols
16,18 (381/393)
(isabelle,isabelle,UTF-8-Isabelle)
1 error(s)
2:15 PM


Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)
fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
"rev2 [] ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"
strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved](*PSL*)
theorem "rev2 xs ys = rev1 xs @ ys"
find_proof DInd
oops
Proof state
Auto update
Update
Search:
100%
apply (induct xs arbitrary: ys)
apply auto
done
Output
Query
Sledgehammer
Symbols
16,18 (381/393)
(isabelle,isabelle,UTF-8-Isabelle)
195/512MB
1 error(s)
2:14 PM

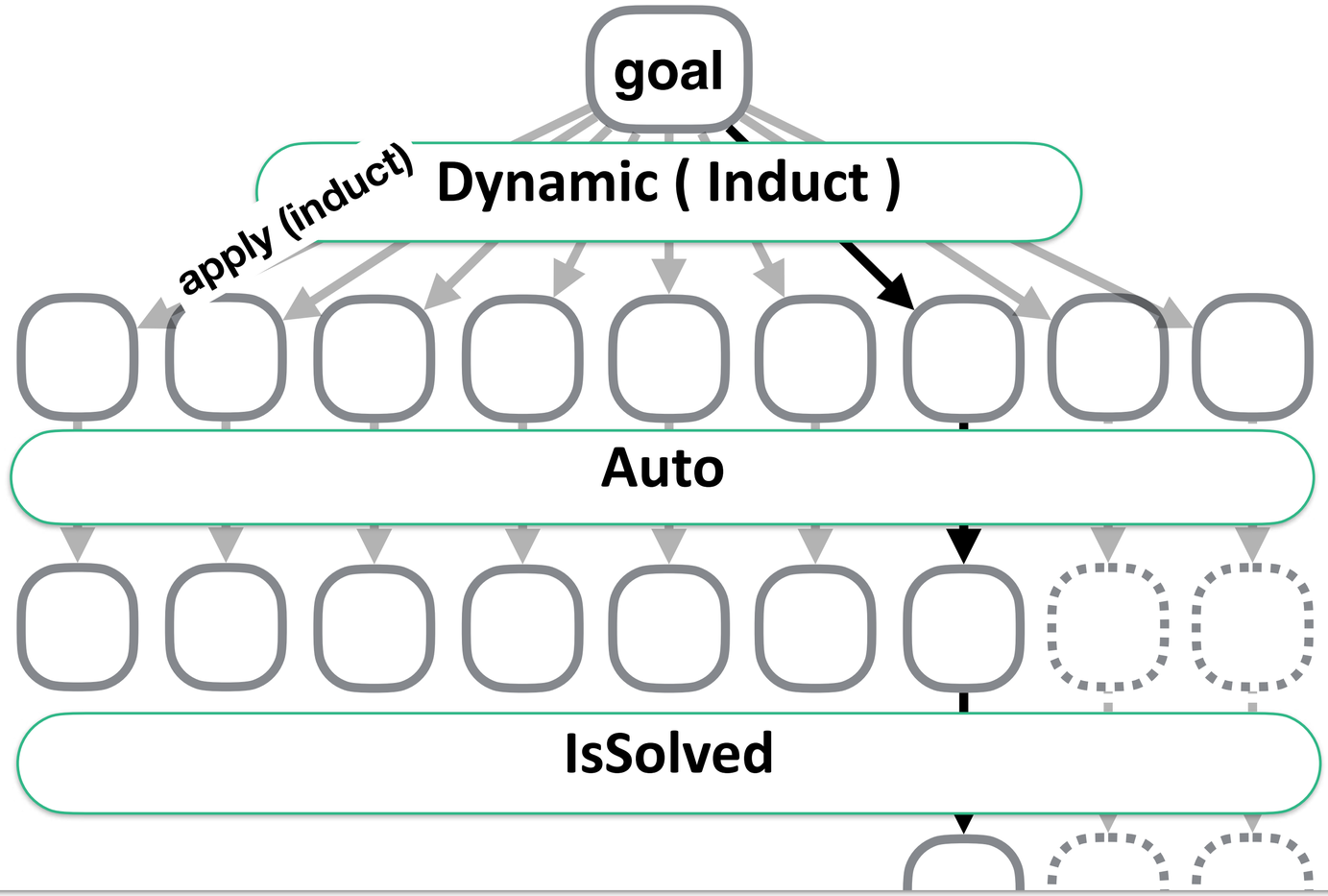
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```

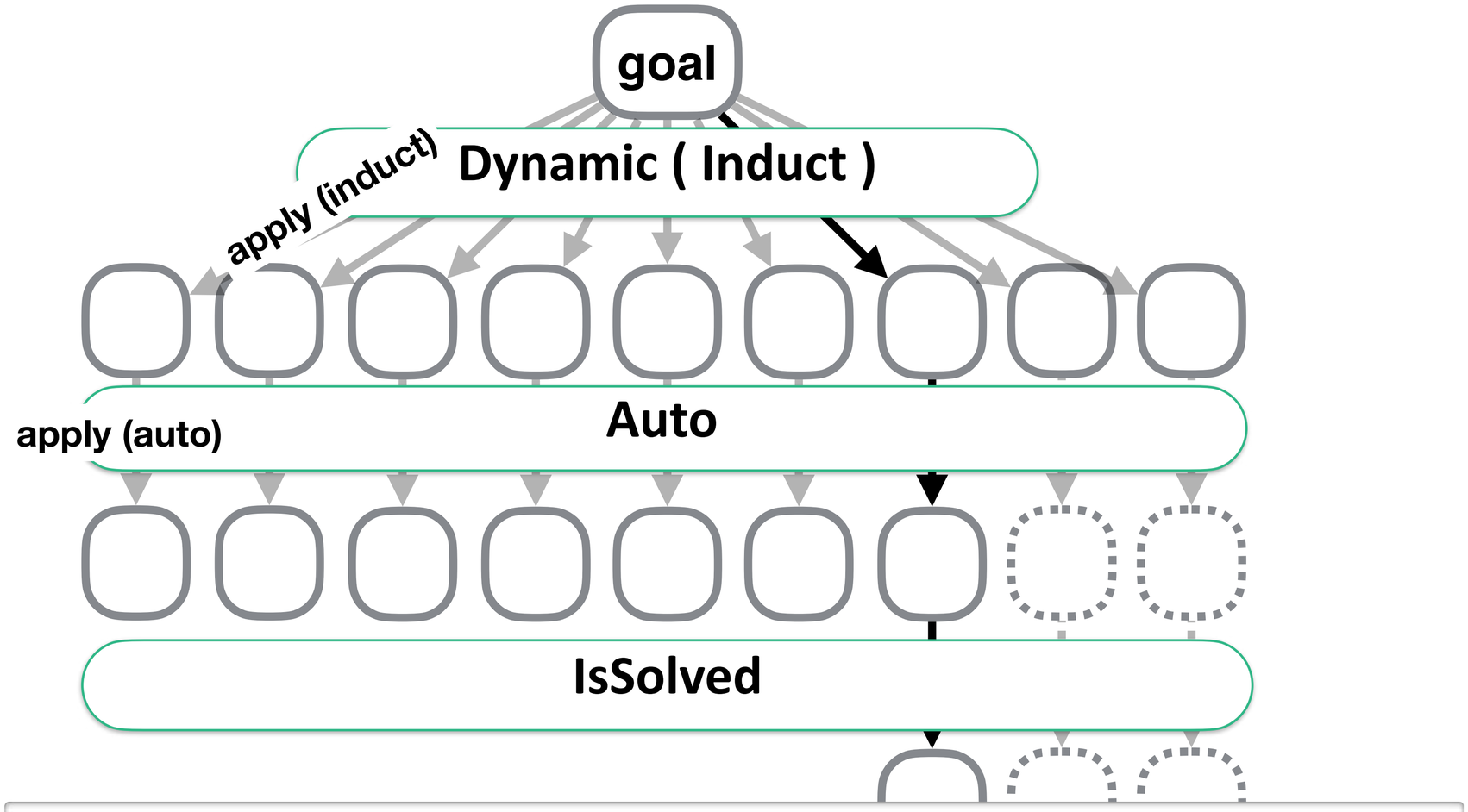


```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



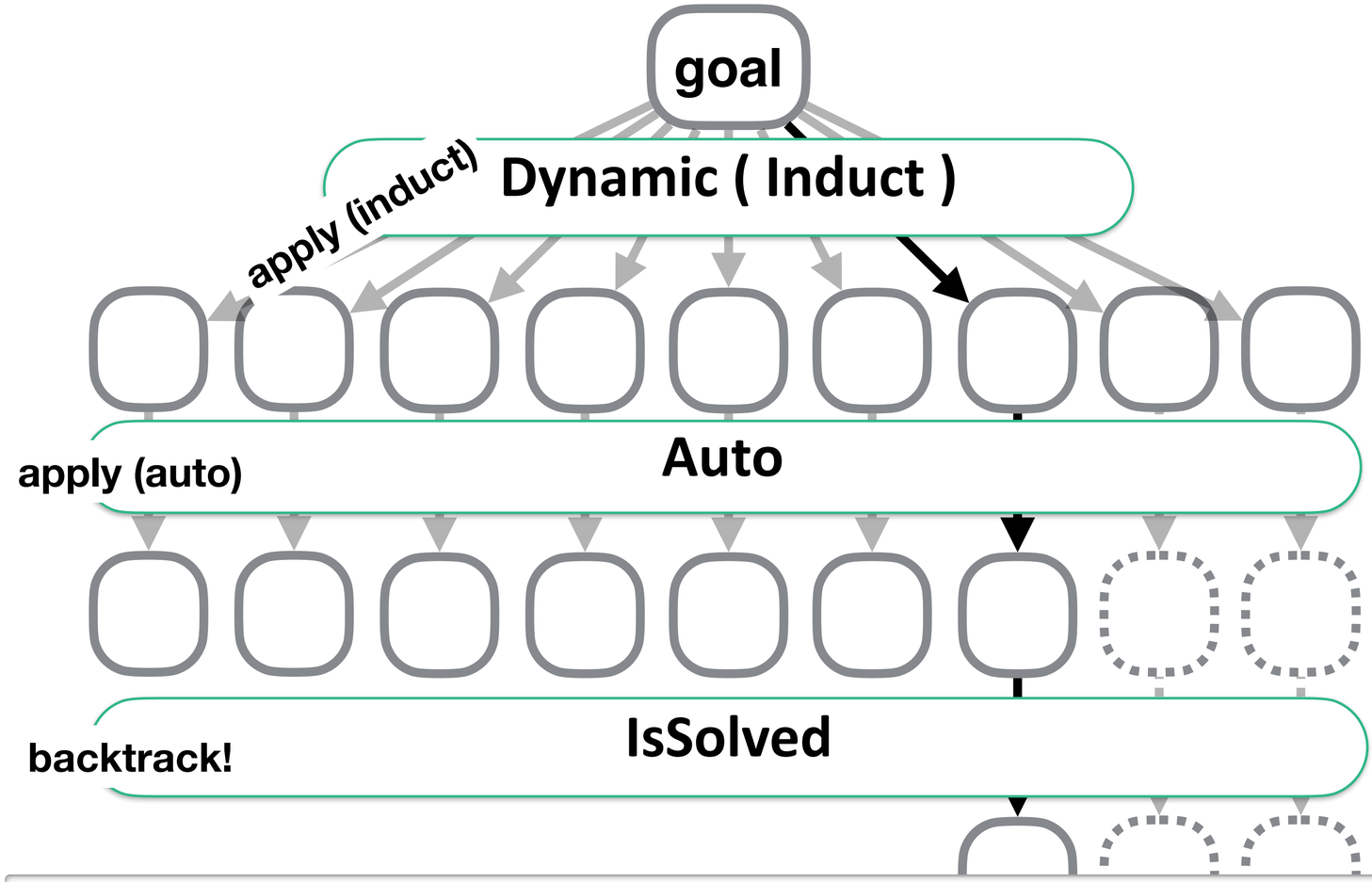

```
goal (1 subgoal):
 1. ⋀y. y ∈ {F. is_filter F} ⟹ rev2 xs ys = rev1 xs @ ys
```

```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



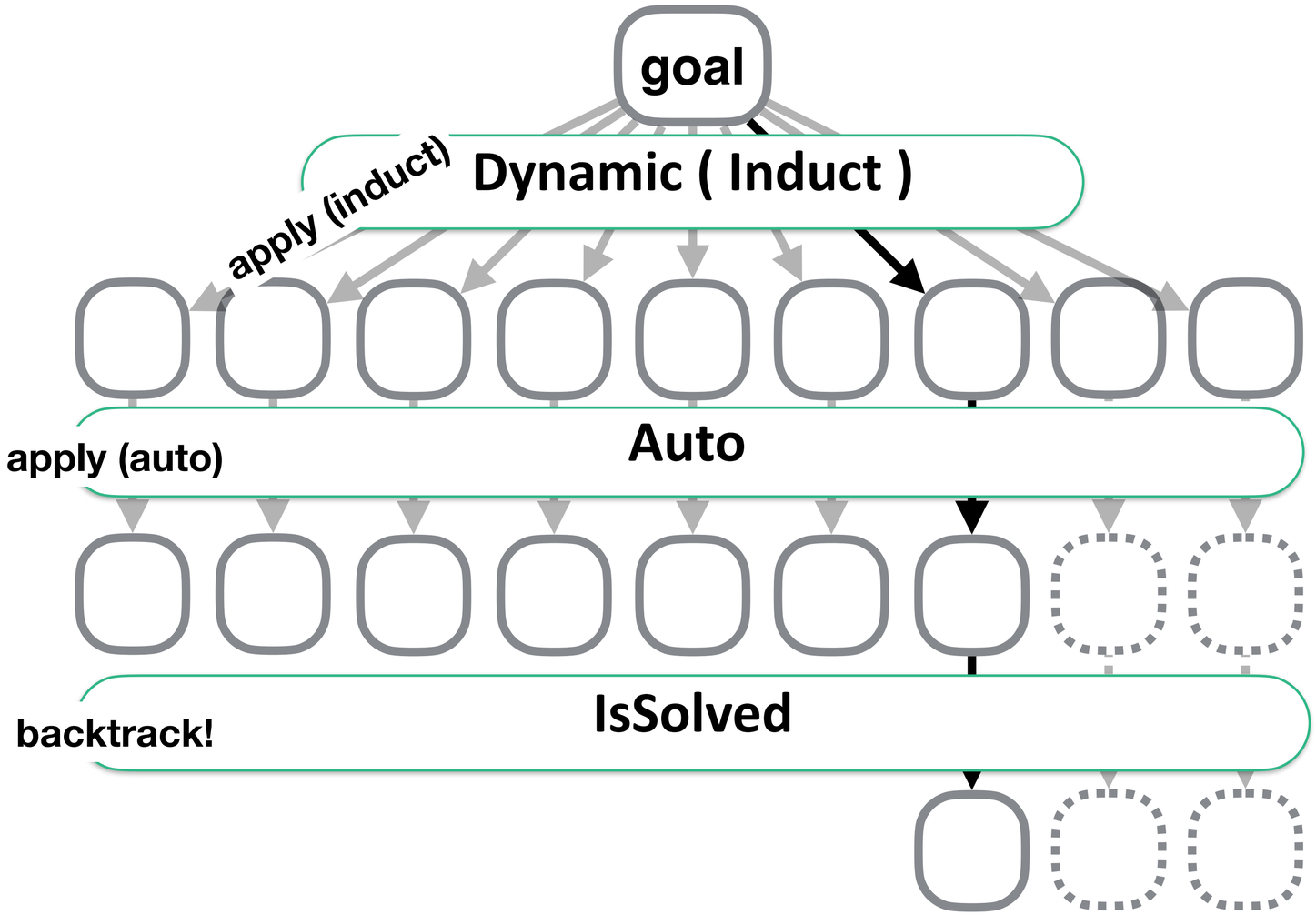

```
goal (1 subgoal):
 1. ⋀y. is_filter y ⟹ rev2 xs ys = rev1 xs @ ys
```

```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```

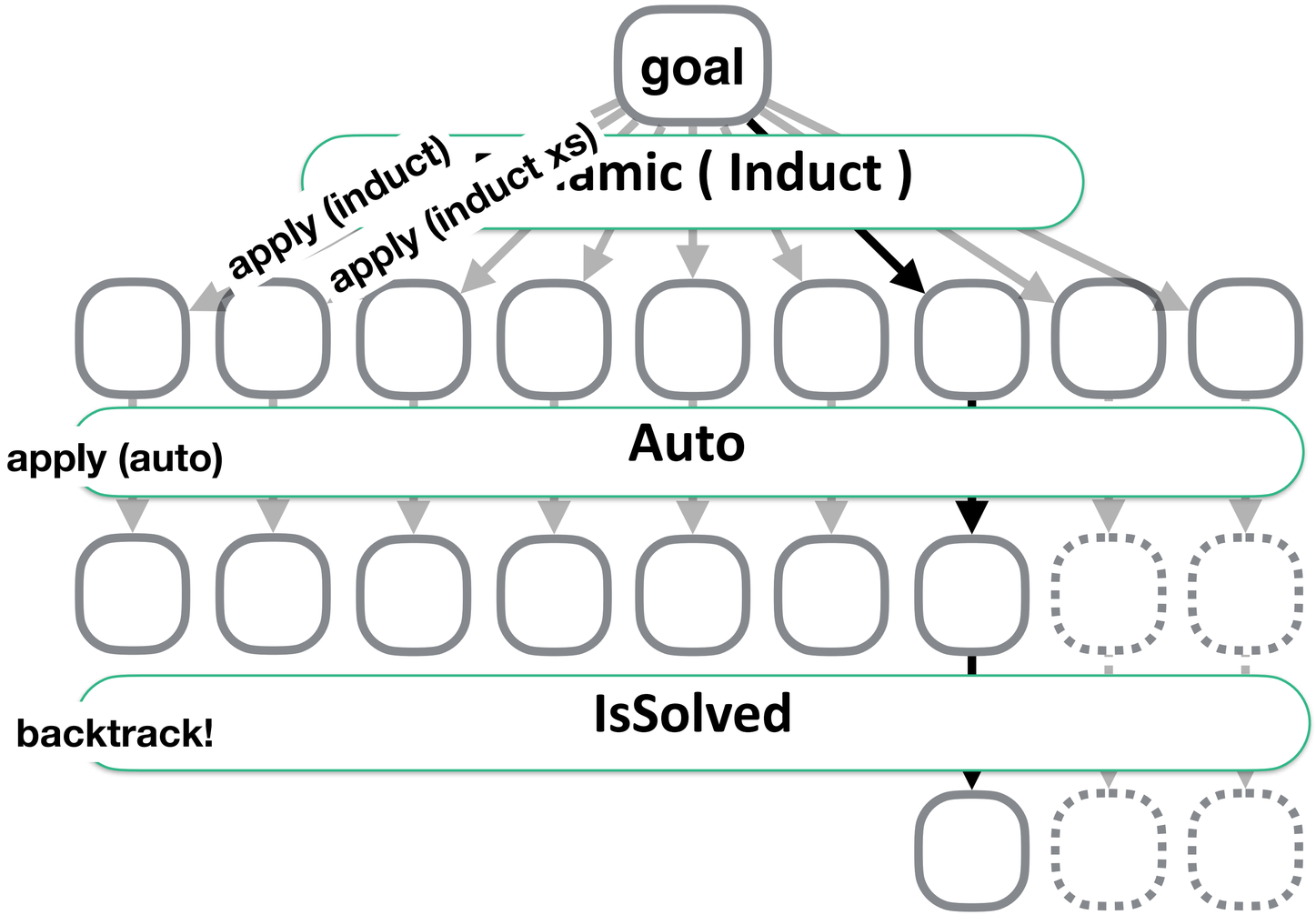



```
goal (1 subgoal):
 1. ⋀y. is_filter y ⟹ rev2 xs ys = rev1 xs @ ys
```

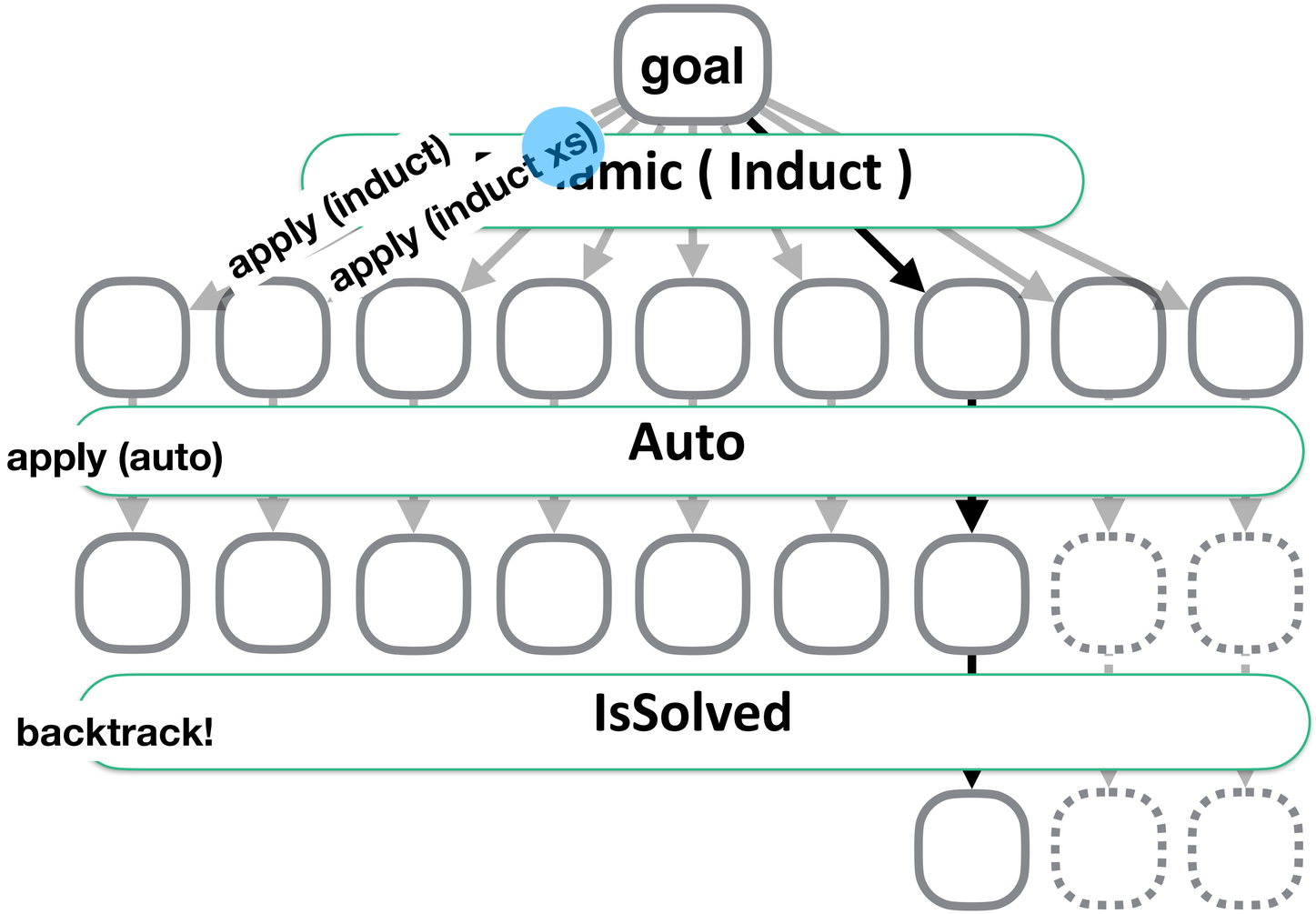
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



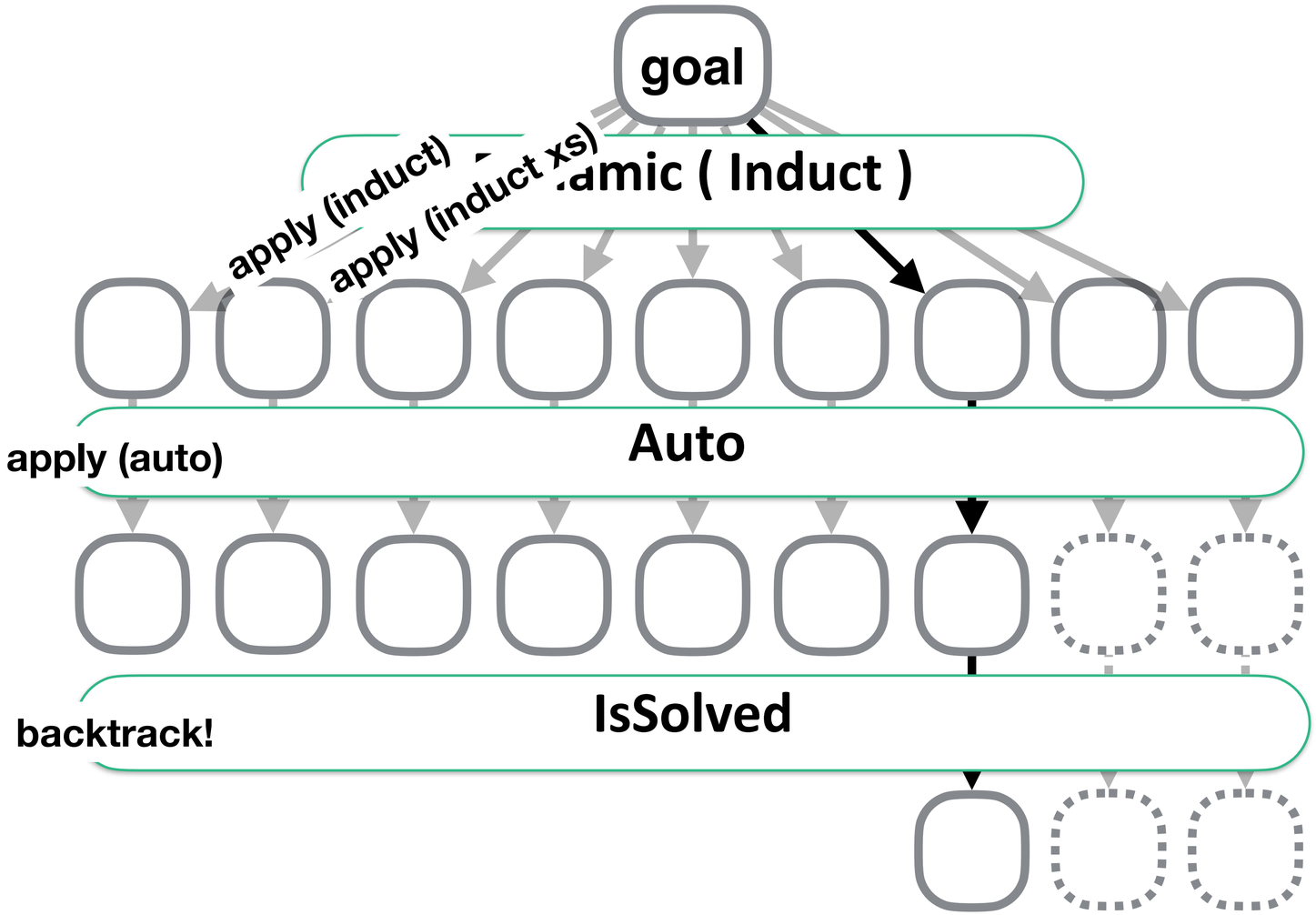
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



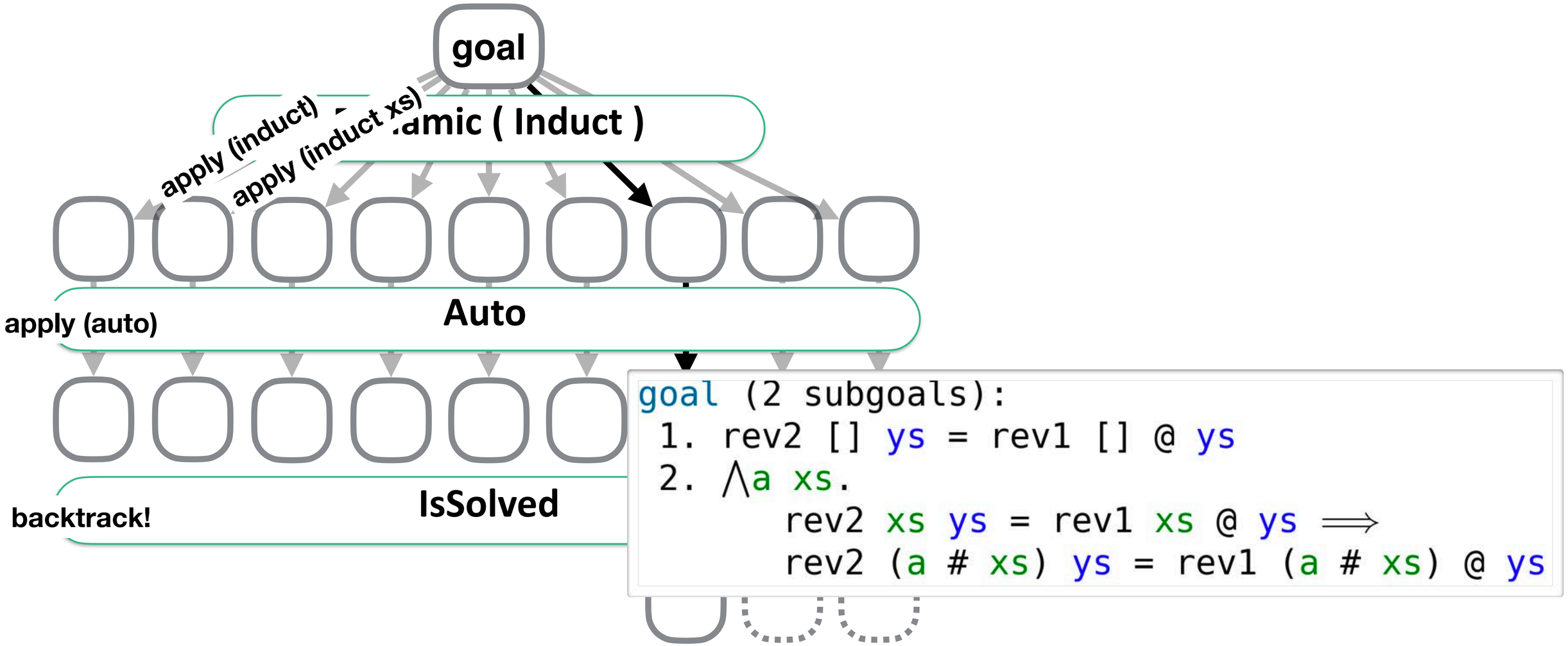
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



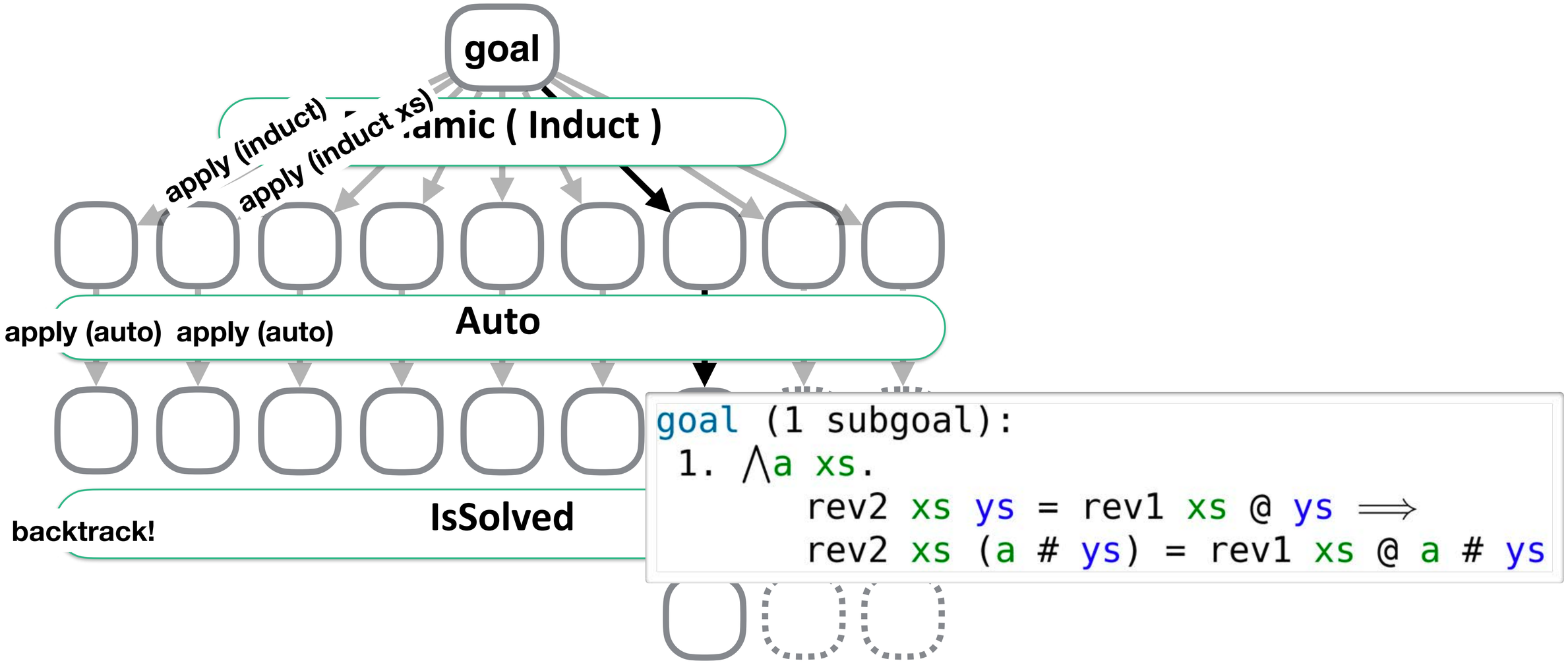
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



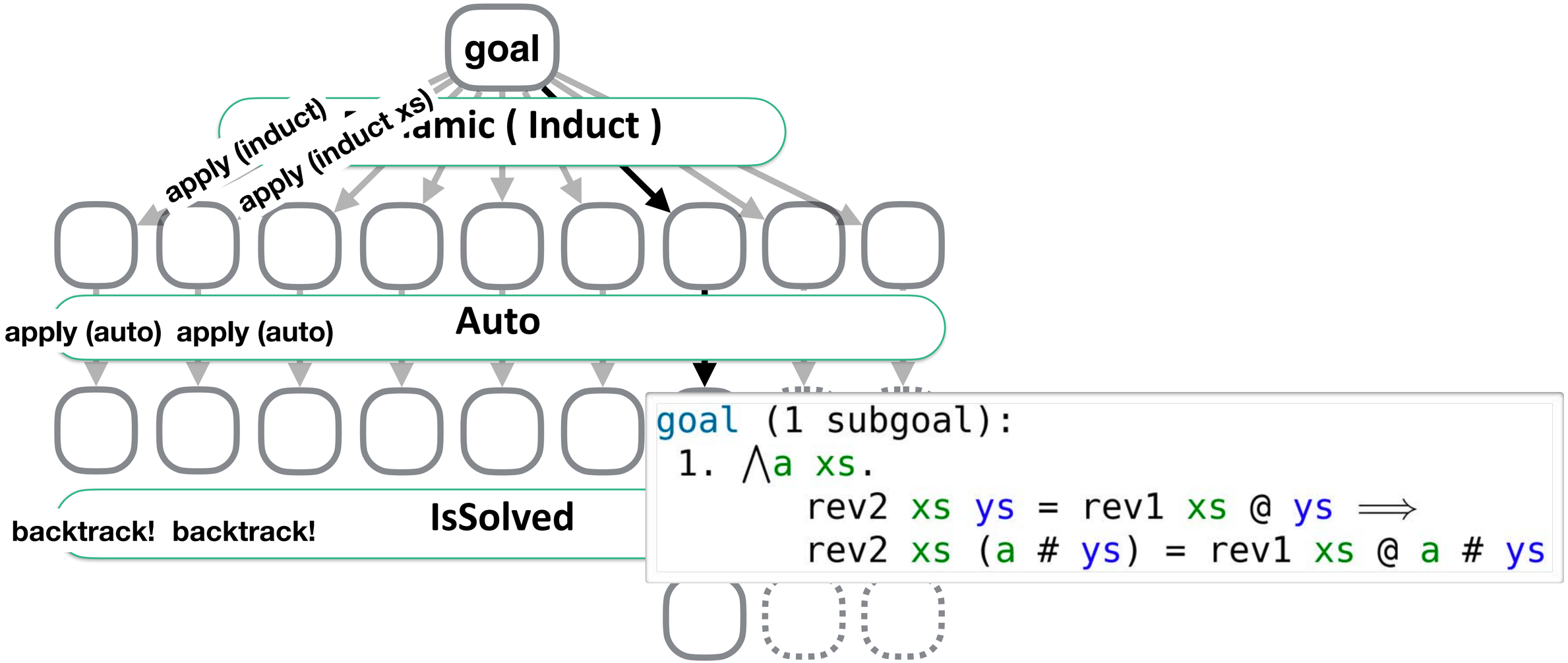
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



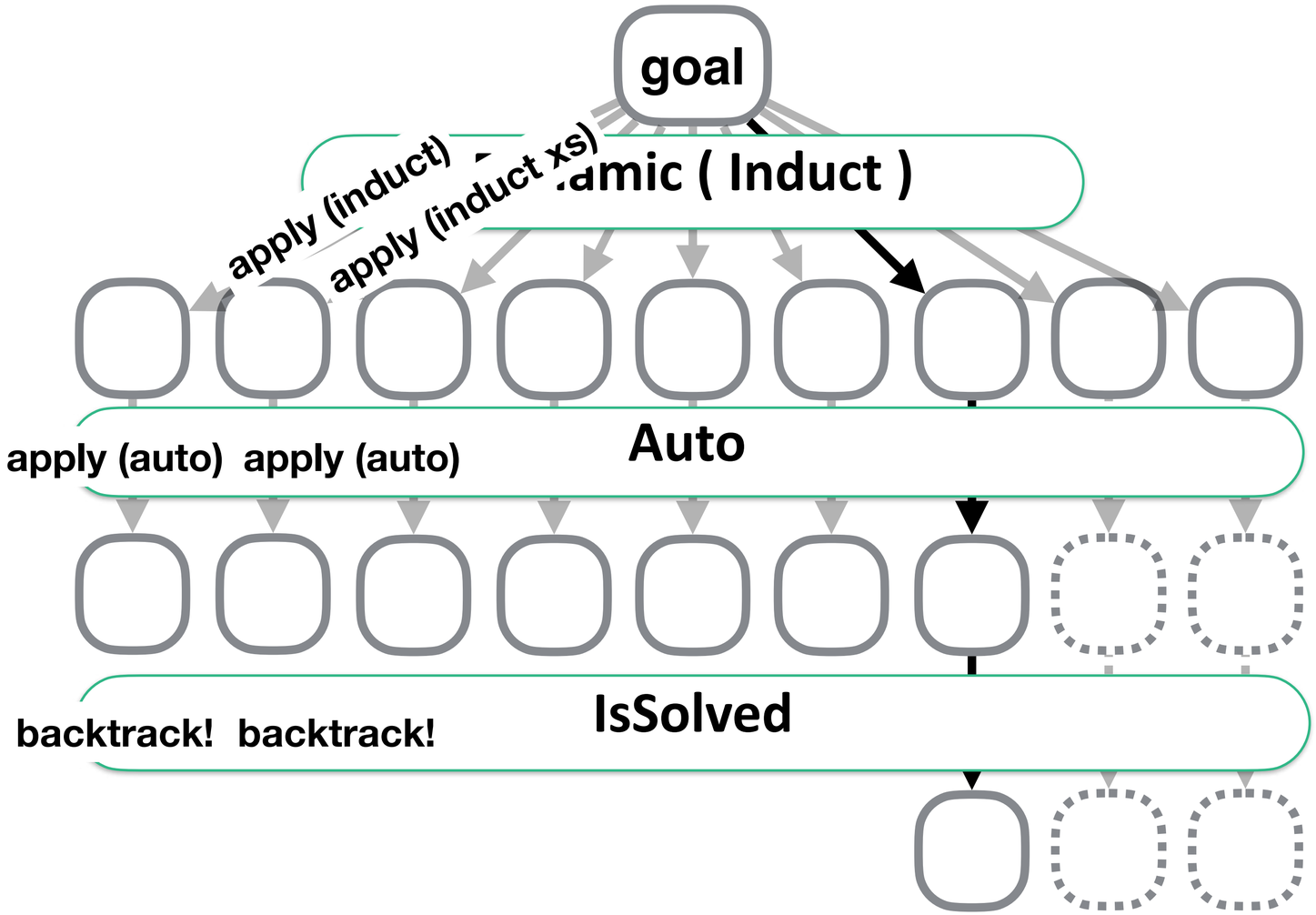
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



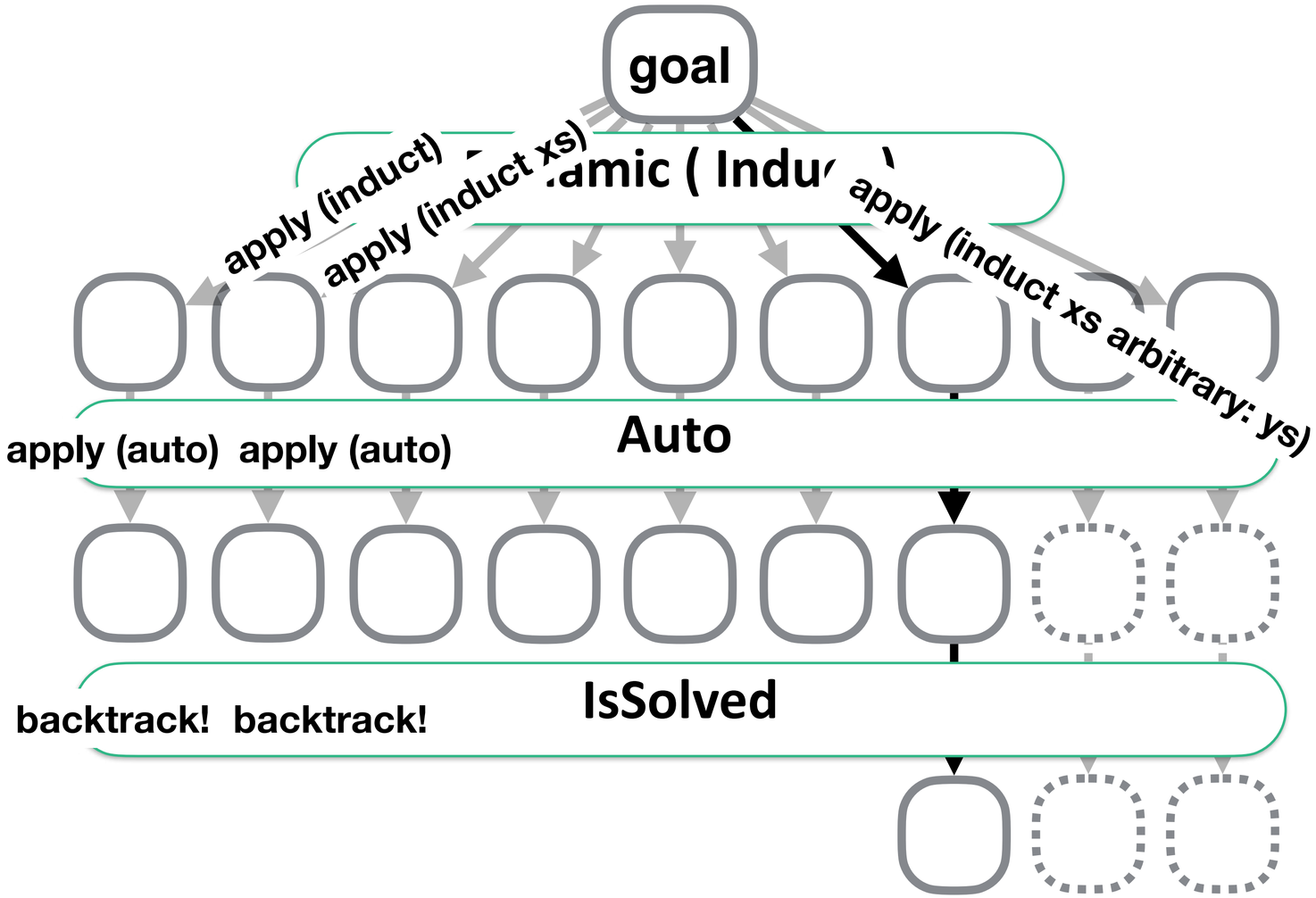
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



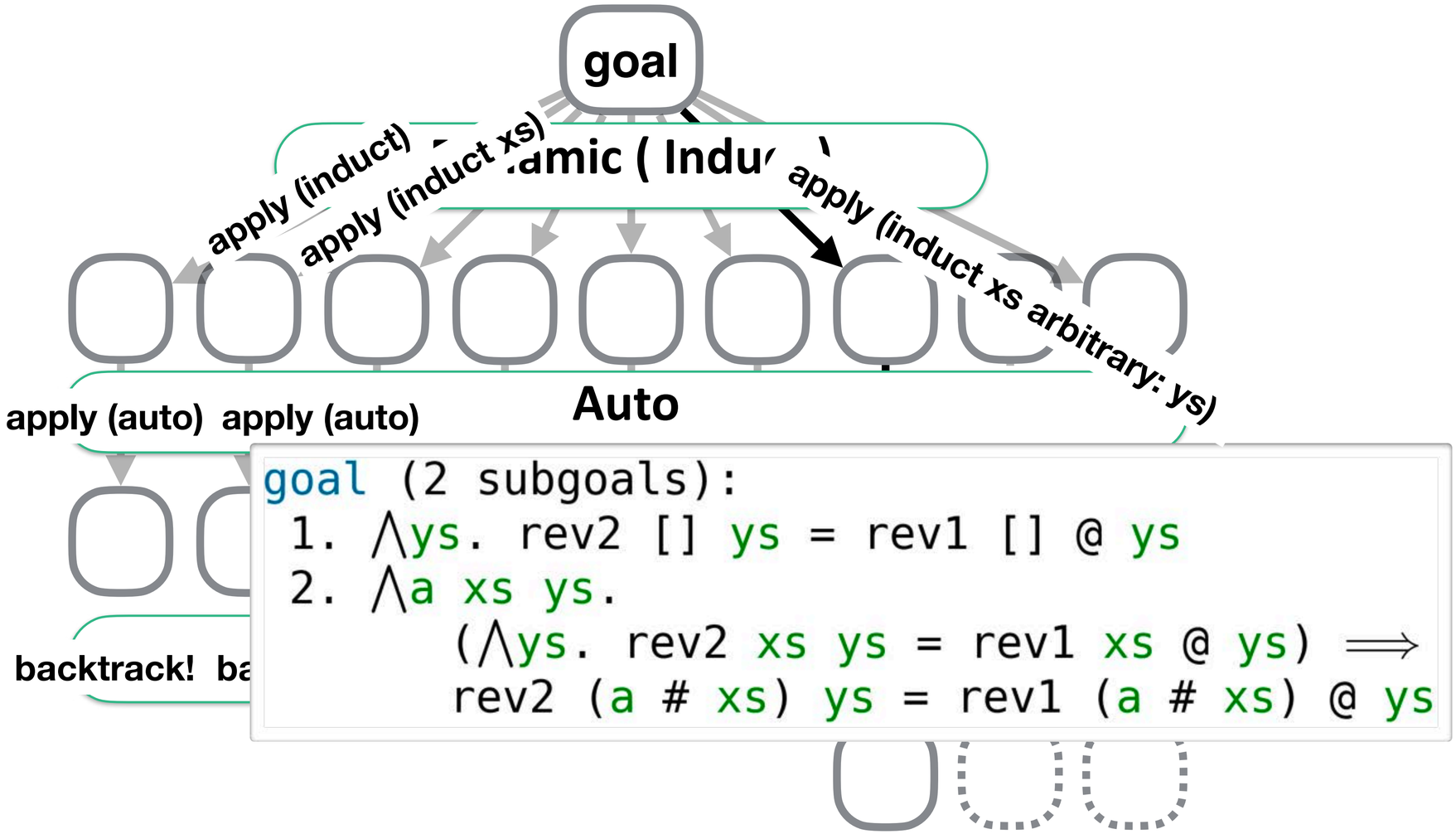
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



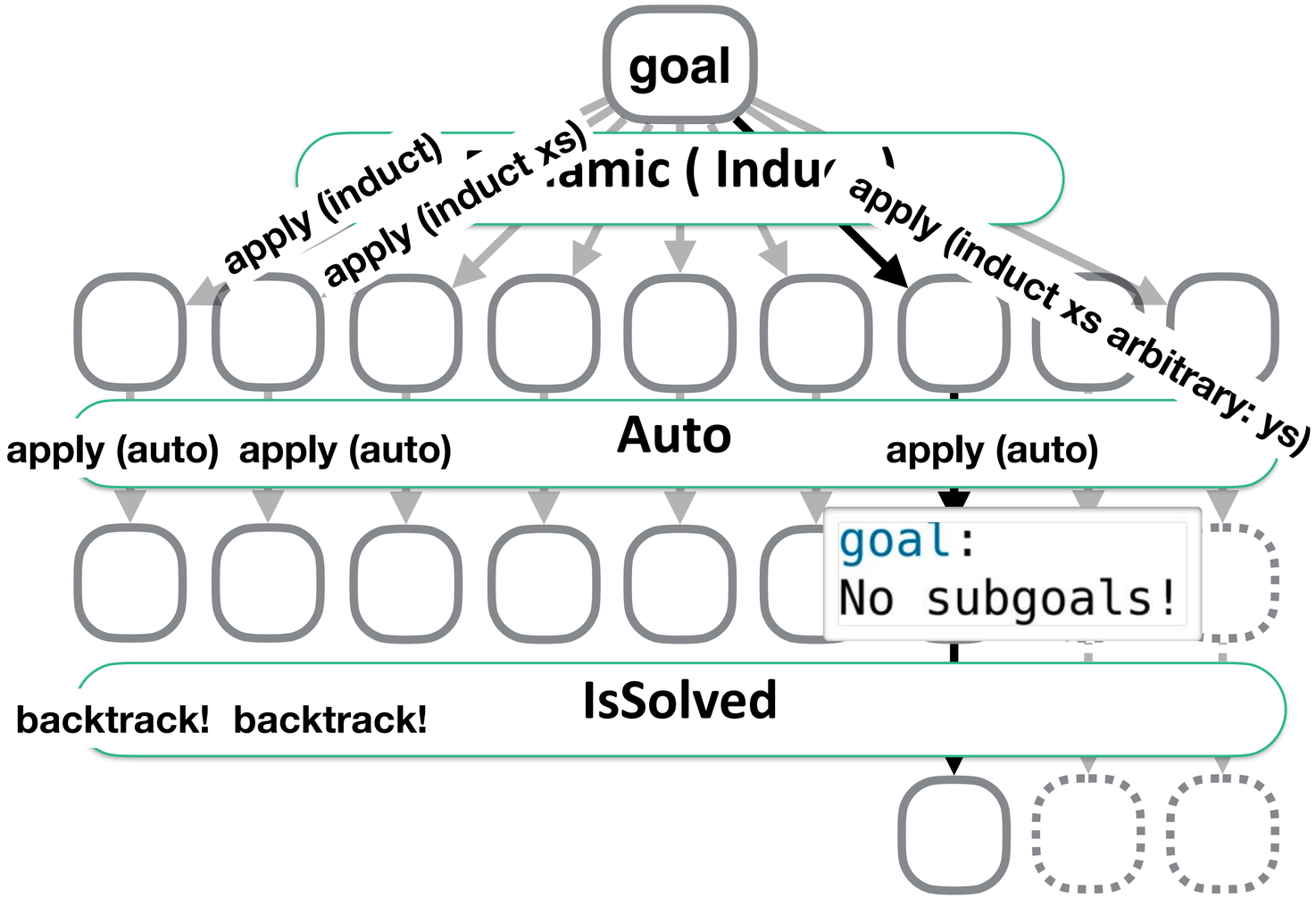
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



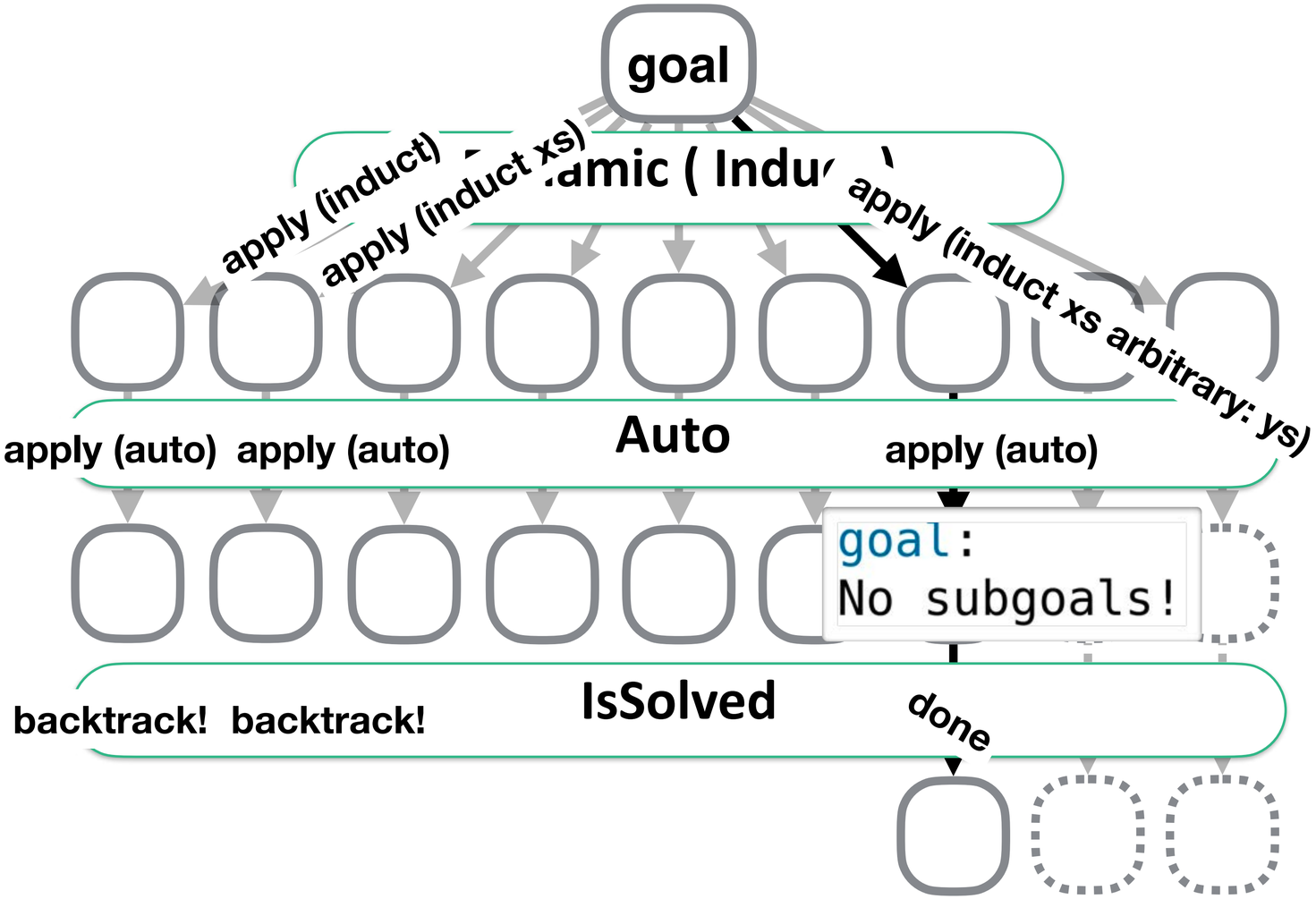
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



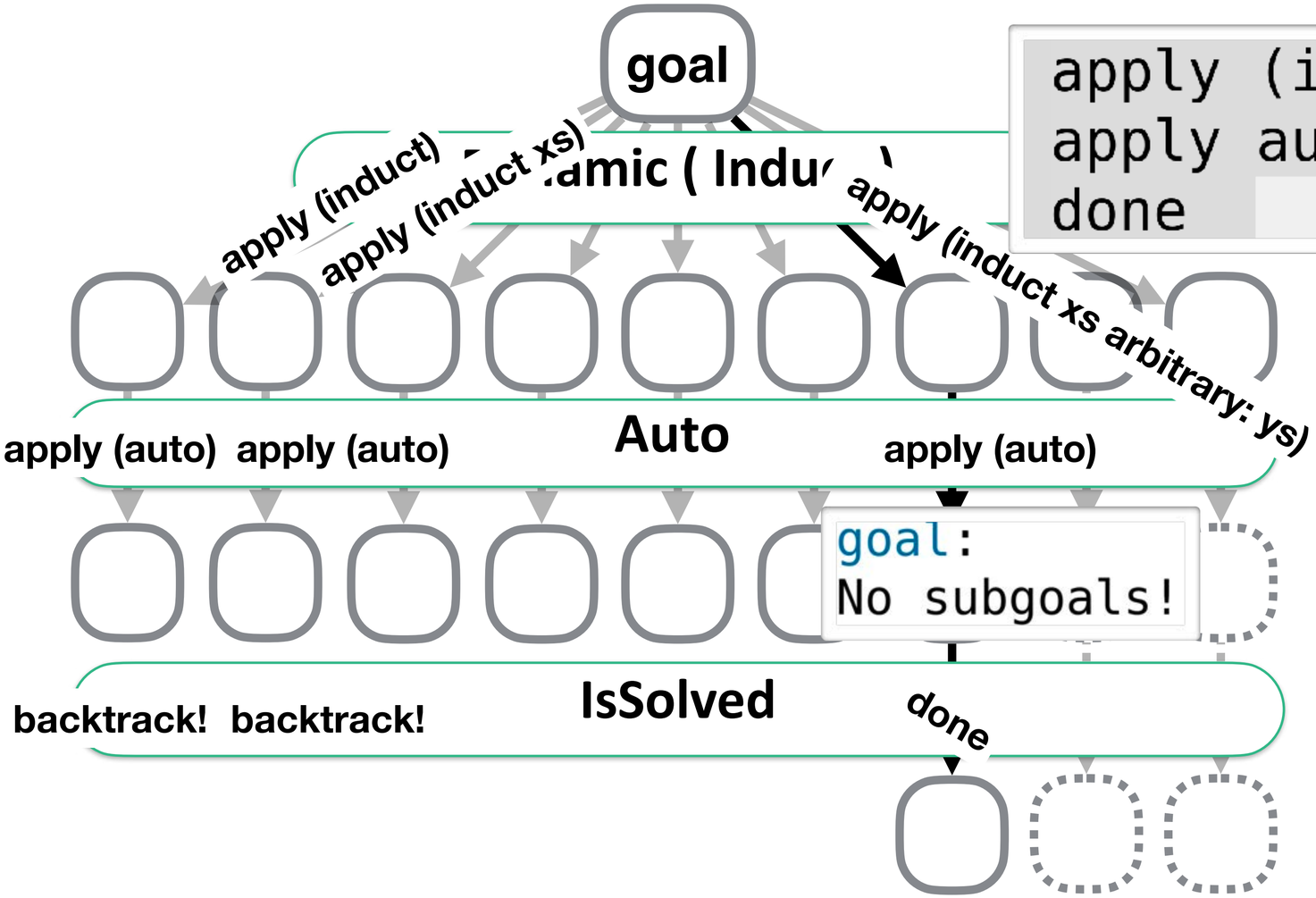
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```




```
apply (induct xs arbitrary: ys)
apply auto
done
```


Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)
fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
"rev2 [] ys = ys"
"rev2 (x # xs) ys = rev2 xs (x # ys)"
strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved](*PSL*)
theorem "rev2 xs ys = rev1 xs @ ys"
find_proof DInd
oops
Proof state
Auto update
Update
Search:
100%
apply (induct xs arbitrary: ys)
apply auto
done
Output
Query
Sledgehammer
Symbols
16,18 (381/393)
(isabelle,isabelle,UTF-8-Isabelle)
195/512MB
1 error(s)
2:14 PM

Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

(*PSL*)

```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd
  oops
```

Proof state | Auto update | Update | Search:

```
apply (induct xs arbitrary: ys)
apply auto
done
```

Output | Query | Sledgehammer | Symbols

16,18 (381/393)

(isabelle,isabelle,UTF-8-Isabelle) 195/512MB 1 error(s) 2:14 PM

Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

(*PSL*)

```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd
  oops
```

Proof state | Auto update | Update | Search:

```
apply (induct xs arbitrary: ys)
apply auto
done
```

Output | Query | Sledgehammer | Symbols

16,18 (381/393)

Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

(*PSL*)

Indeed, auto returned quickly in our case.

```
theorem "rev2 xs ys = rev1 xs @ ys"
?find_proof DInd
 oops
```

Proof state | Auto update | Update | Search:

```
apply (induct xs arbitrary: ys)
apply auto
done
```

Output | Query | Sledgehammer | Symbols

16,18 (381/393)

(isabelle,isabelle,UTF-8-Isabelle) 195/512MB 1 error(s) 2:14 PM

Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

(*PSL*)

Indeed, auto returned quickly in our case.

What if subsequent steps are more involved?

```
theorem "rev2 xs ys = rev1 xs @ ys"
?find_proof DInd
 oops

apply (
apply auto
done
```

Search:

Output Query Sledgehammer Symbols

16,18 (381/393)

(isabelle,isabelle,UTF-8-Isabelle) 195/512MB 1 error(s) 2:14 PM

Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

(*PSL*)

```
theorem "rev2 xs ys = rev1 xs @ ys"
?find_proof DInd
 oops
```

Indeed, auto returned quickly in our case.

What if subsequent steps are more involved?

```
pply (
apply auto
done
```

We have to identify promising arguments for induction tactics

without relying on a search!

Search:

Output Query Sledgehammer Symbols

16,18 (381/393)

(isabelle,isabelle,UTF-8-Isabelle) 195/512MB 1 error(s) 2:14 PM

Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

(*PSL*)

Indeed, auto returned quickly in our case.

What if subsequent steps are more involved?

We have to identify promising arguments for induction tactics

without relying on a search!

Deep learning?

Many key challenges remain

Unsupervised Learning

Memory and one-shot learning

Imagination-based Planning with Generative Models

Learning Abstract Concepts

Transfer Learning

Language understanding

CENTER FOR Brains Minds+ Machines

March 20, 2019

The Power of Self-Learning Systems

Demis Hassabis

DeepMind

https://cbmm.mit.edu/video/power-self-learning-systems

Many key challenges remain

Unsupervised Learning

Memory and one-shot learning

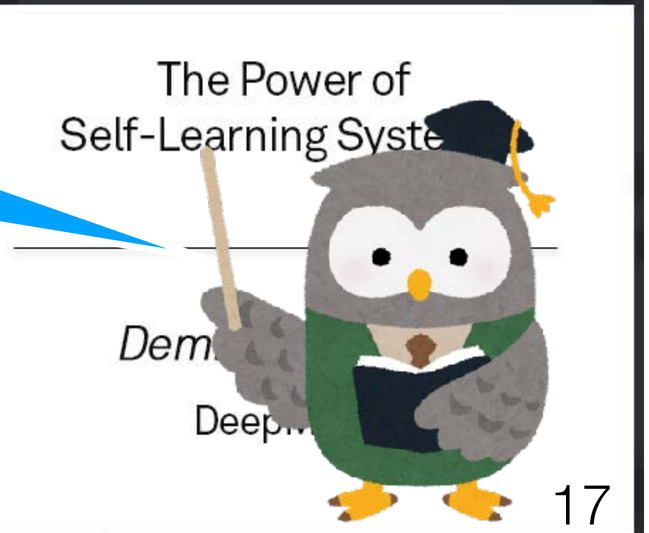
Imagination-based Planning with Generative Models

Learning Abstract Concepts

Abstract concepts?

CENTER FOR Brains Minds+ Machines

March 20, 2019



https://cbmm.mit.edu/video/power-self-learning-systems

Many key challenges remain

Unsupervised Learning

Memory and one-shot learning

Imagination-based Planning with Generative Models

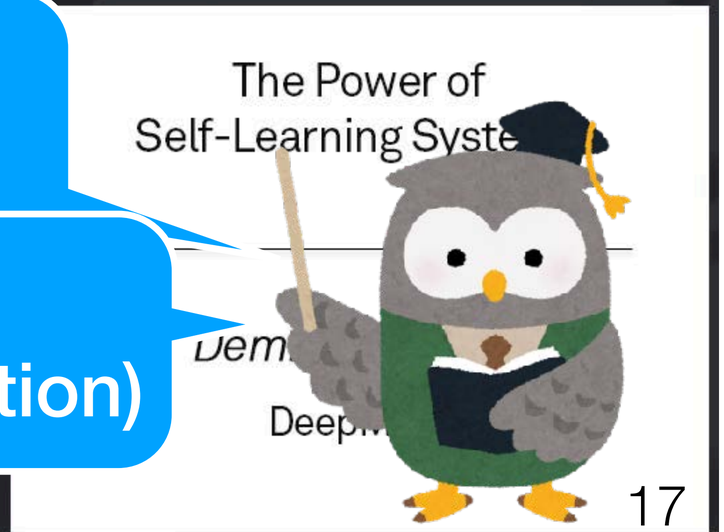
Learning Abstract Concepts

CENTER FOR Brains Minds+ Machines

March 20, 2019

Abstract concepts?

Why not logic?
(LiFtEr = Logical Feature Extraction)



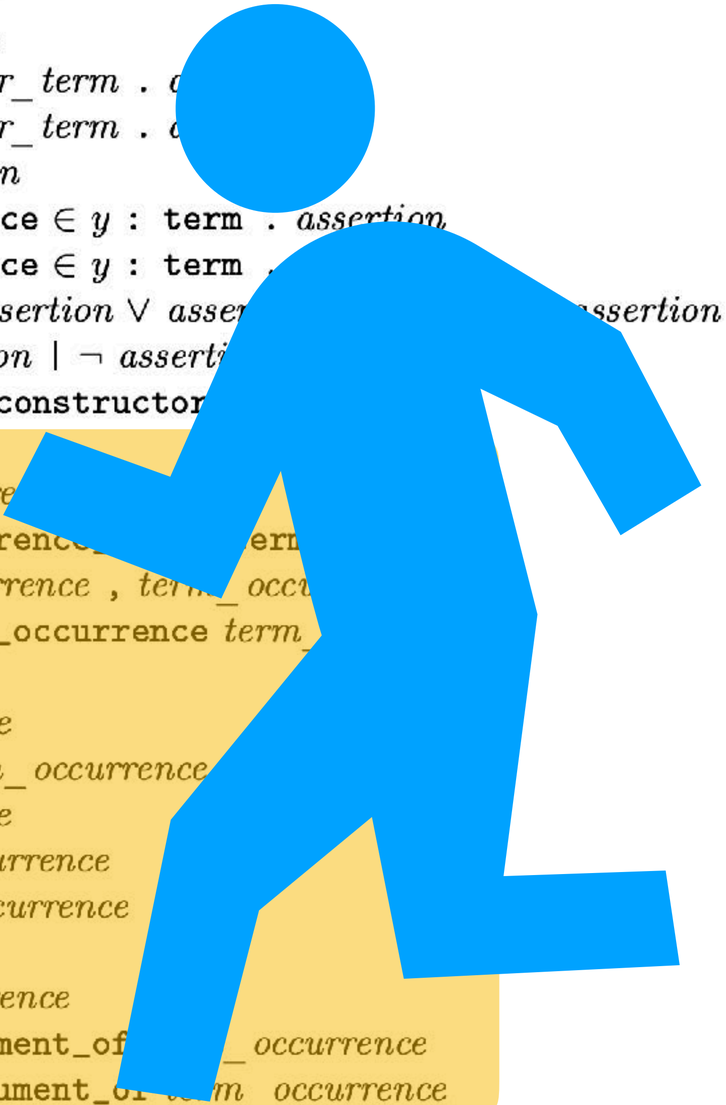
https://cbmm.mit.edu/video/power-self-learning-systems

Abstract Syntax of LiFtEr

assertion := *atomic* | *connective* | *quantifier* | (*assertion*)
type := `term` | `term_occurrence` | `rule` | `number`
modifier_term := `induction_term` | `arbitrary_term`
quantifier := $\exists x$: *type*. *assertion*
| $\forall x$: *type*. *assertion*
| $\exists x$: `term` $\in$ *modifier_term* . *assertion*
| $\forall x$: `term` $\in$ *modifier_term* . *assertion*
| $\exists x$: `rule` . *assertion*
| $\exists x$: `term_occurrence` $\in y$: `term` . *assertion*
| $\forall x$: `term_occurrence` $\in y$: `term` . *assertion*
connective := `True` | `False` | *assertion* $\vee$ *assertion* | *assertion* $\wedge$ *assertion*
assertion $\rightarrow$ *assertion* | $\neg$ *assertion*
pattern := `all_only_var` | `all_constructor` | `mixed`
atomic :=
rule `is_rule_of` *term_occurrence*
| *term_occurrence* `term_occurrence_is_of_term` *term*
| `are_same_term` (*term_occurrence* , *term_occurrence*)
| *term_occurrence* `is_in_term_occurrence` *term_occurrence*
| `is_atomic` *term_occurrence*
| `is_constant` *term_occurrence*
| `is_recursive_constant` *term_occurrence*
| `is_variable` *term_occurrence*
| `is_free_variable` *term_occurrence*
| `is_bound_variable` *term_occurrence*
| `is_lambda` *term_occurrence*
| `is_application` *term_occurrence*
| *term_occurrence* `is_an_argument_of` *term_occurrence*
| *term_occurrence* `is_nth_argument_of` *term_occurrence*

Abstract Syntax of LiFtEr

assertion := *atomic* | *connective* | *quantifier* | (*assertion*)
type := **term** | **term_occurrence** | **rule** | **number**
modifier_term := **induction_term** | **arbitrary_term**
quantifier := $\exists x$: *type*. *assertion*
| $\forall x$: *type*. *assertion*
| $\exists x$: **term** $\in$ *modifier_term* . *a*
| $\forall x$: **term** $\in$ *modifier_term* . *a*
| $\exists x$: **rule** . *assertion*
| $\exists x$: **term_occurrence** $\in y$: **term** . *assertion*
| $\forall x$: **term_occurrence** $\in y$: **term** .
connective := **True** | **False** | *assertion* $\vee$ *asser* *ssertion*
assertion $\rightarrow$ *assertion* | $\neg$ *asserti*
pattern := **all_only_var** | **all_constructor**
atomic :=
rule **is_rule_of** *term_occurre*
| *term_occurrence* **term_occurrence** *erm*
| **are_same_term** (*term_occurrence* , *term_occ*
| *term_occurrence* **is_in_term_occurrence** *term*
| **is_atomic** *term_occurrence*
| **is_constant** *term_occurrence*
| **is_recursive_constant** *term_occurrence*
| **is_variable** *term_occurrence*
| **is_free_variable** *term_occurrence*
| **is_bound_variable** *term_occurrence*
| **is_lambda** *term_occurrence*
| **is_application** *term_occurrence*
| *term_occurrence* **is_an_argument_of** *_occurrence*
| *term_occurrence* **is_nth_argument_of** *m_occurrence*



Example Assertion in LiFtEr (in Abstract Syntax)

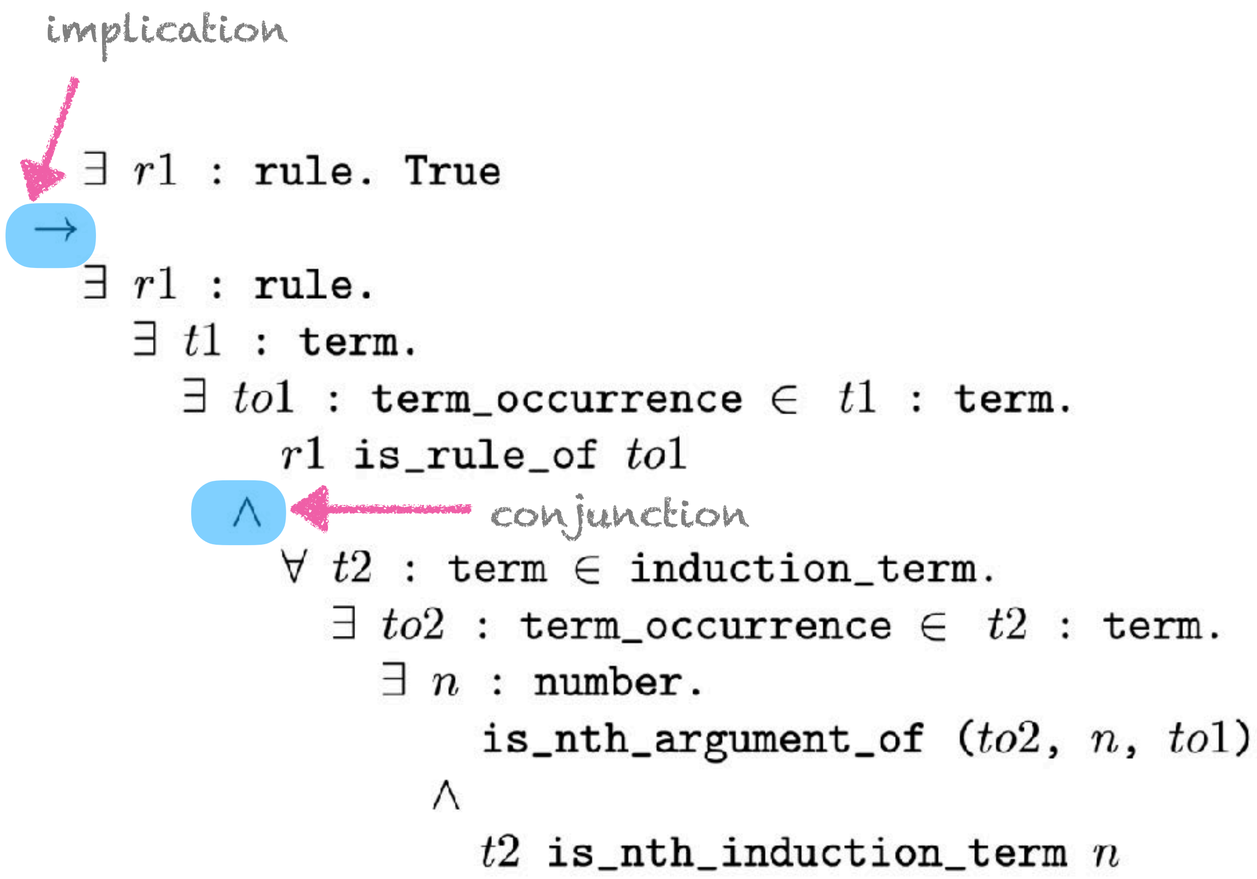
```
  ∃ r1 : rule. True
→
  ∃ r1 : rule.
    ∃ t1 : term.
      ∃ to1 : term_occurrence ∈ t1 : term.
          r1 is_rule_of to1
        ∧
          ∀ t2 : term ∈ induction_term.
            ∃ to2 : term_occurrence ∈ t2 : term.
              ∃ n : number.
                  is_nth_argument_of (to2, n, to1)
                ∧
                  t2 is_nth_induction_term n
```

Example Assertion in LiFtEr (in Abstract Syntax)

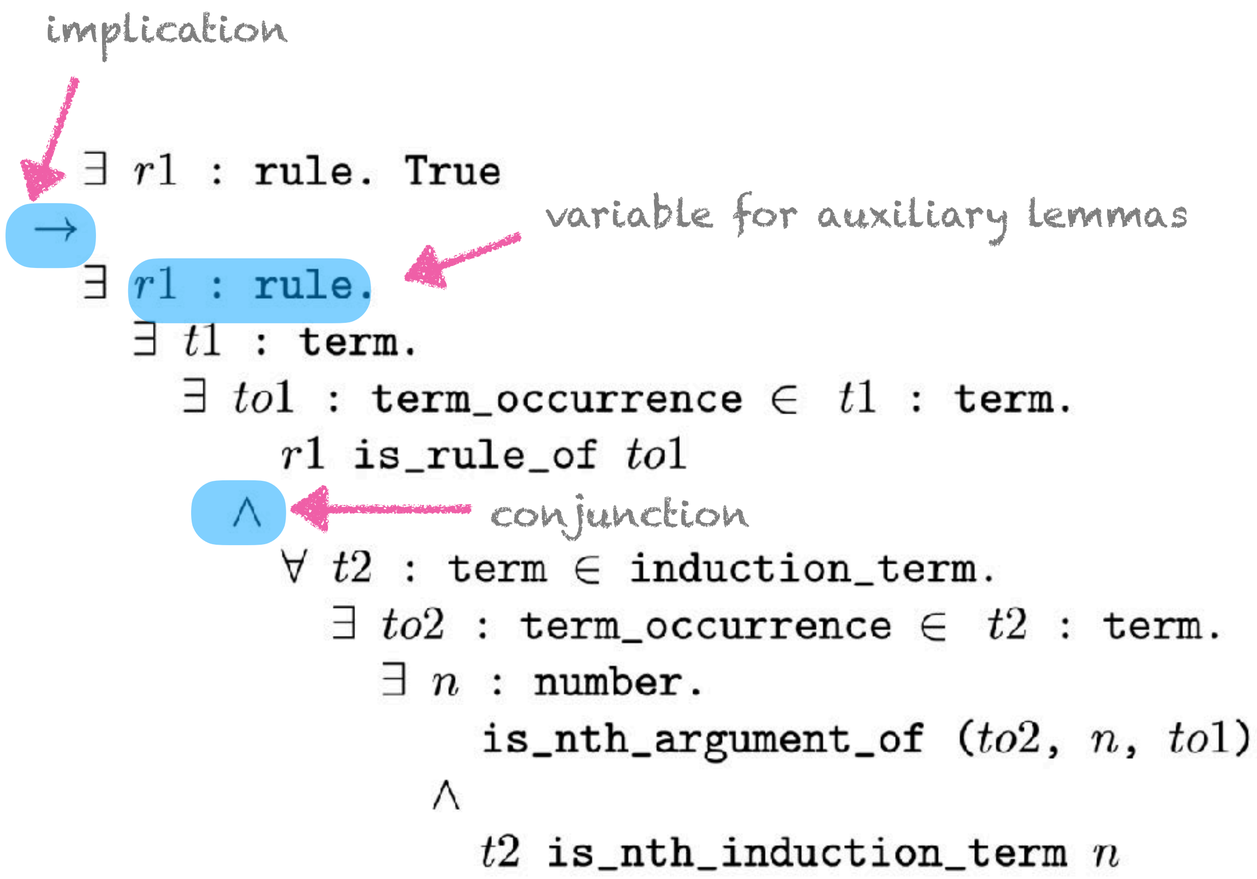
implication

```
∃ r1 : rule. True
→
∃ r1 : rule.
  ∃ t1 : term.
    ∃ to1 : term_occurrence ∈ t1 : term.
        r1 is_rule_of to1
      ∧
        ∀ t2 : term ∈ induction_term.
          ∃ to2 : term_occurrence ∈ t2 : term.
            ∃ n : number.
                is_nth_argument_of (to2, n, to1)
              ∧
                t2 is_nth_induction_term n
```

Example Assertion in LiFtEr (in Abstract Syntax)

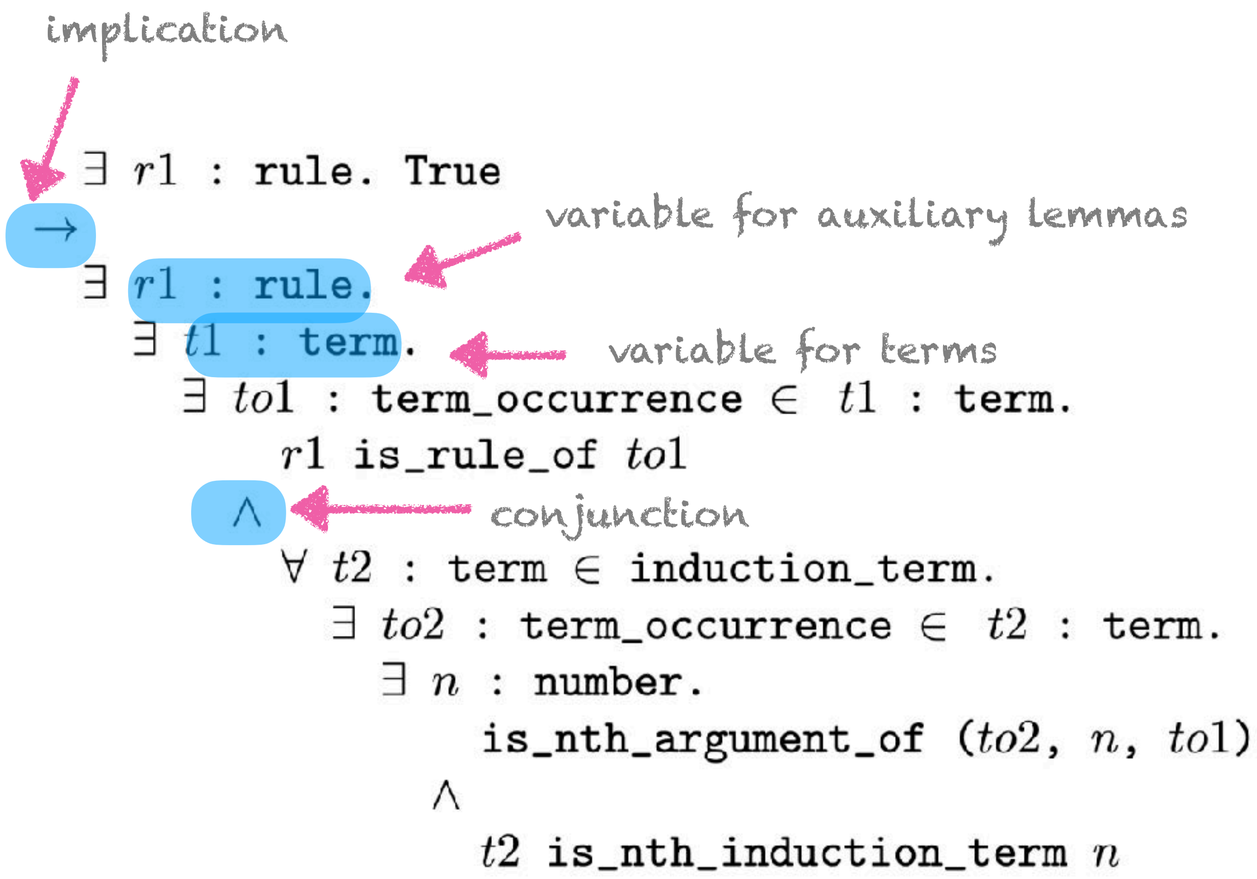


Example Assertion in LiFtEr (in Abstract Syntax)

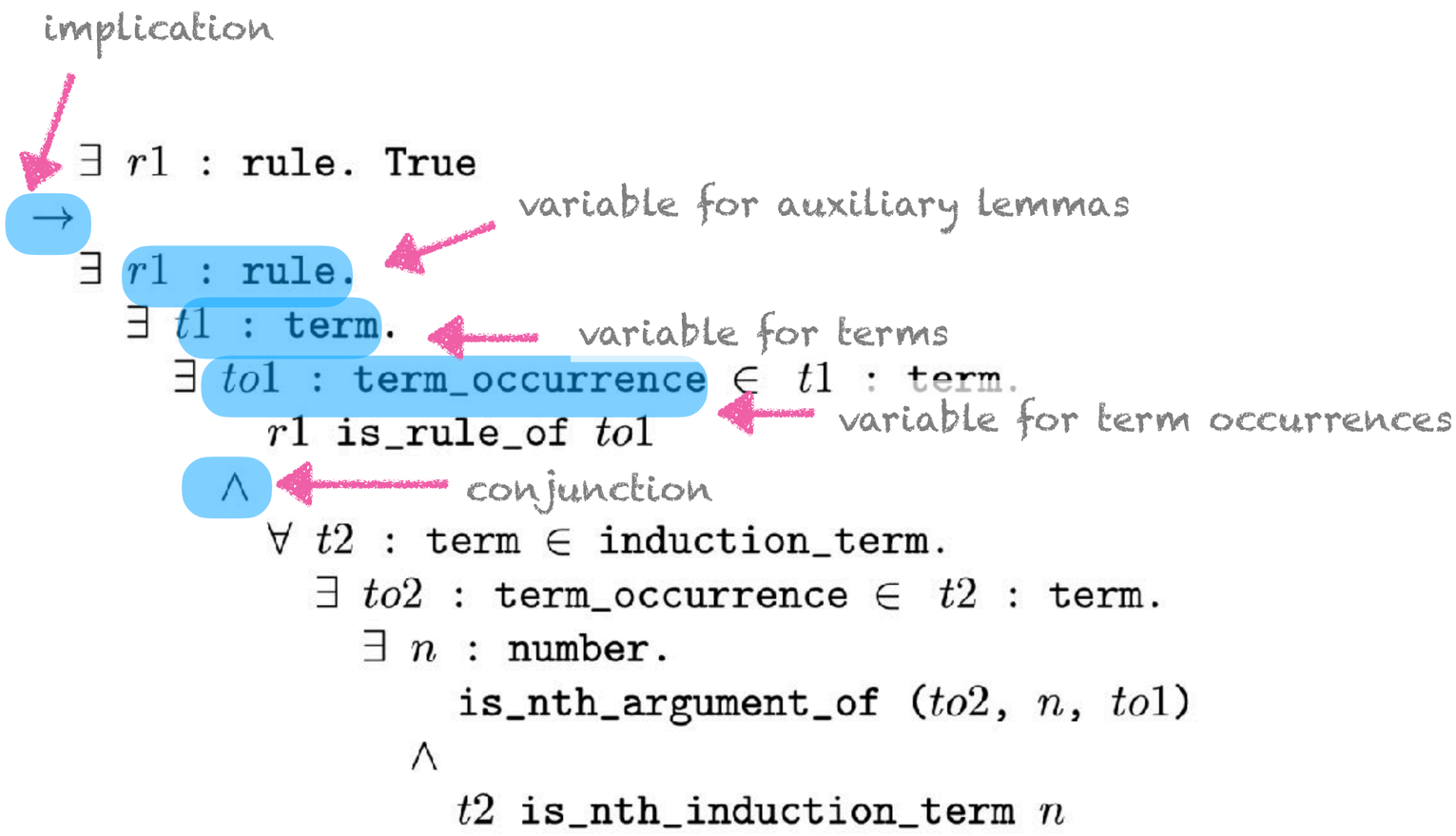



```
∃ r1 : rule. True
→
  ∃ r1 : rule.
    ∃ t1 : term.
      ∃ to1 : term_occurrence ∈ t1 : term.
          r1 is_rule_of to1
        ∧
          ∀ t2 : term ∈ induction_term.
            ∃ to2 : term_occurrence ∈ t2 : term.
              ∃ n : number.
                  is_nth_argument_of (to2, n, to1)
                ∧
                  t2 is_nth_induction_term n
```

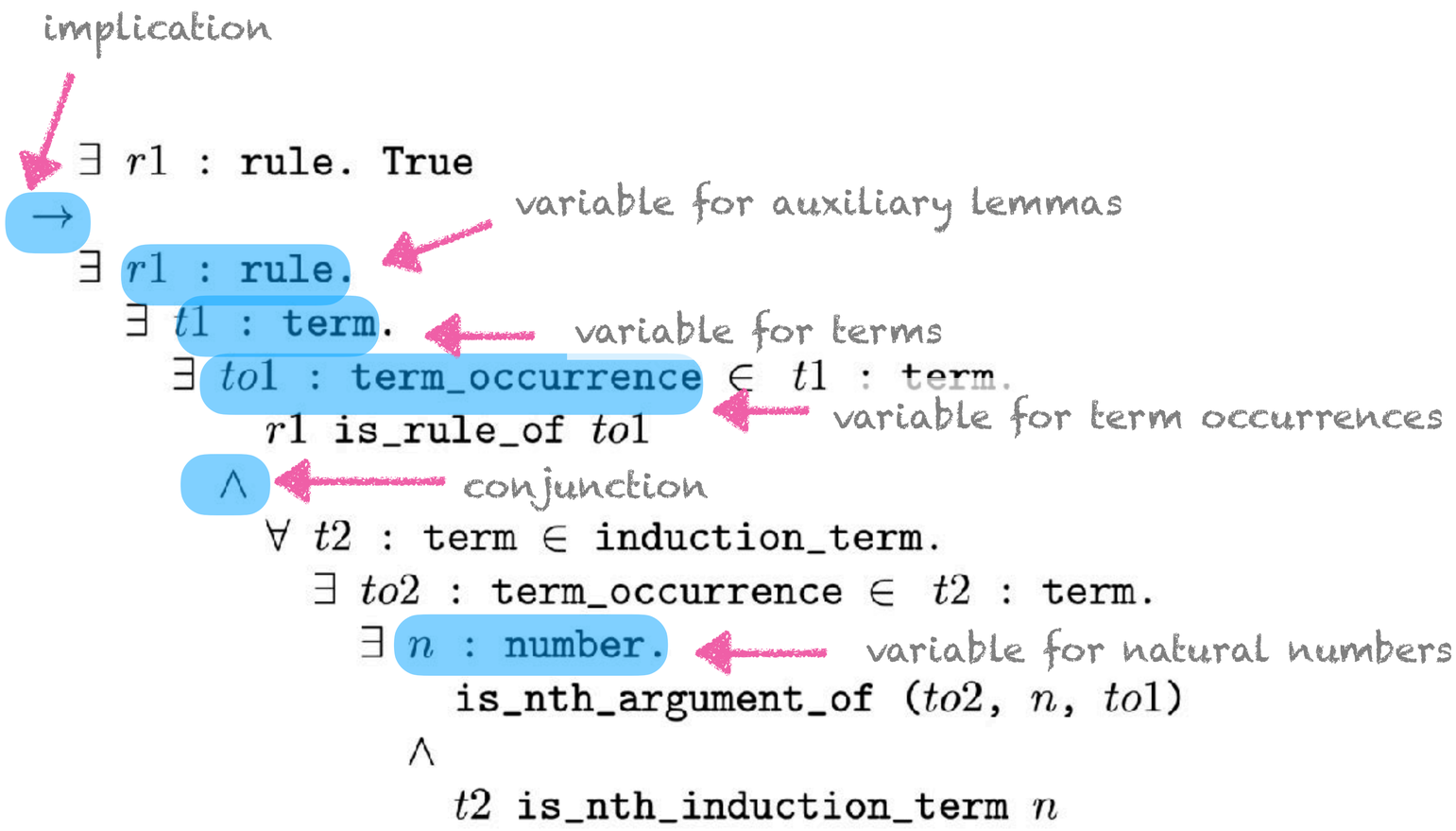
Example Assertion in LiFtEr (in Abstract Syntax)



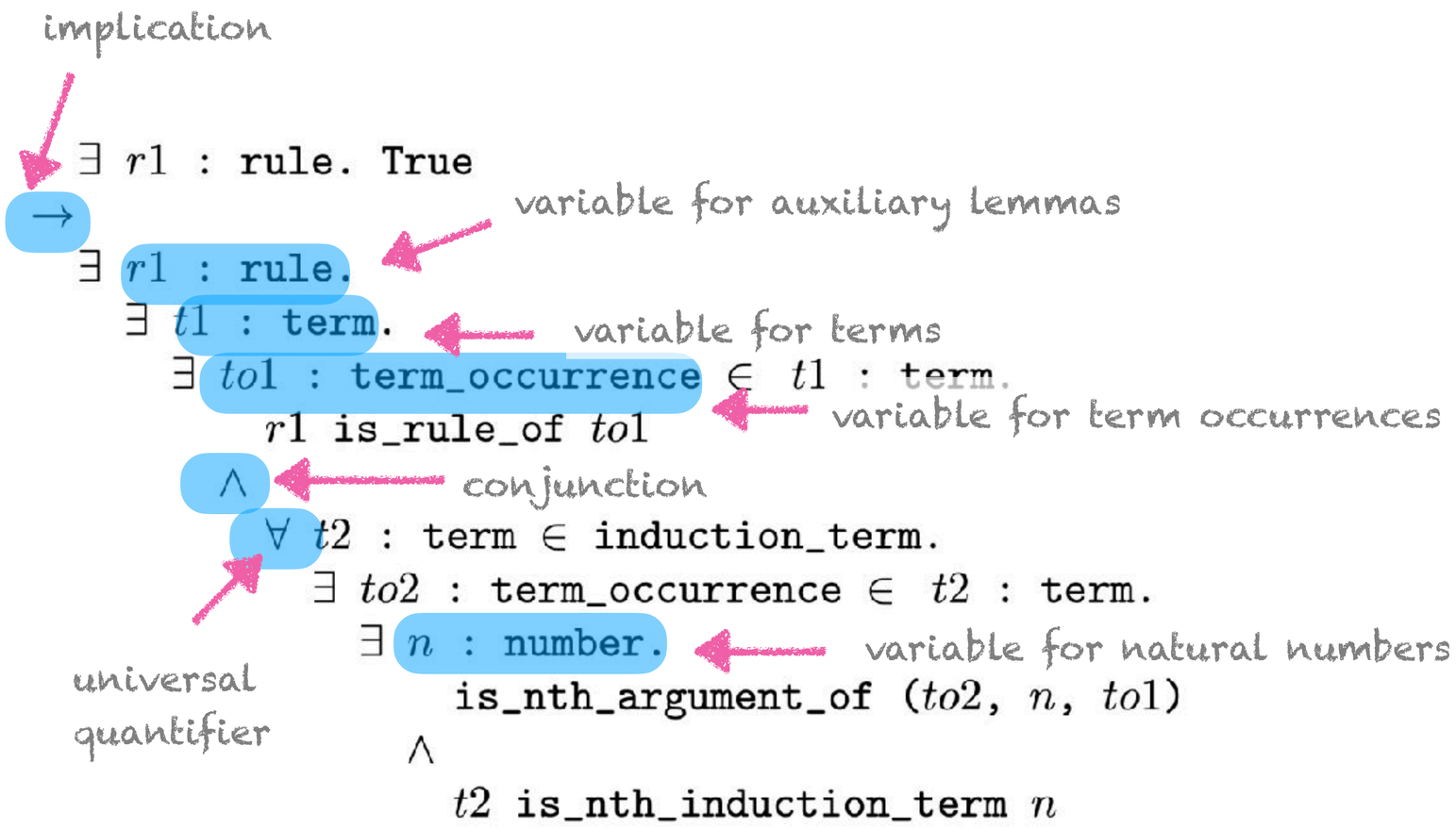
Example Assertion in LiFtEr (in Abstract Syntax)



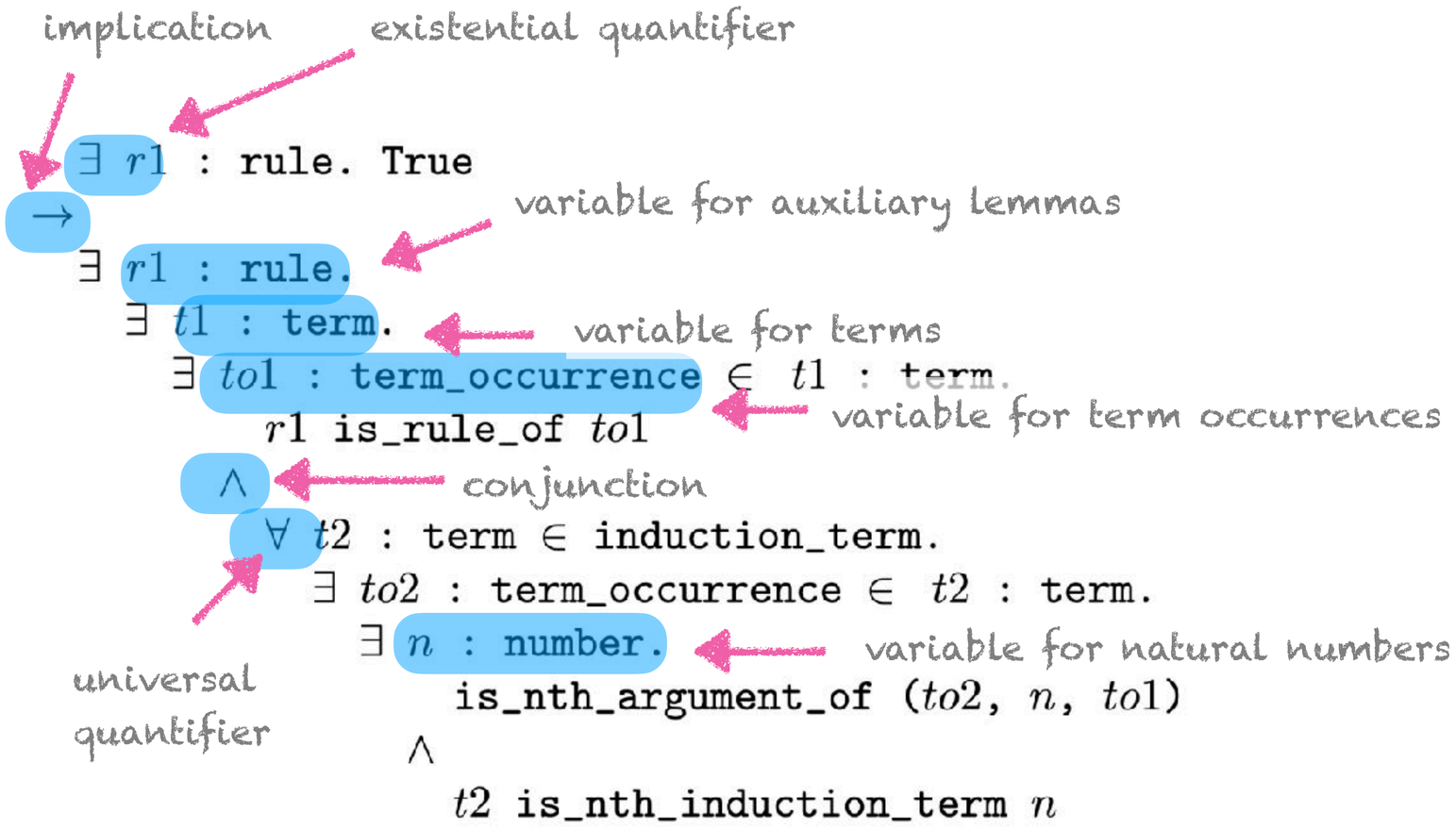
Example Assertion in LiFtEr (in Abstract Syntax)



Example Assertion in LiFtEr (in Abstract Syntax)

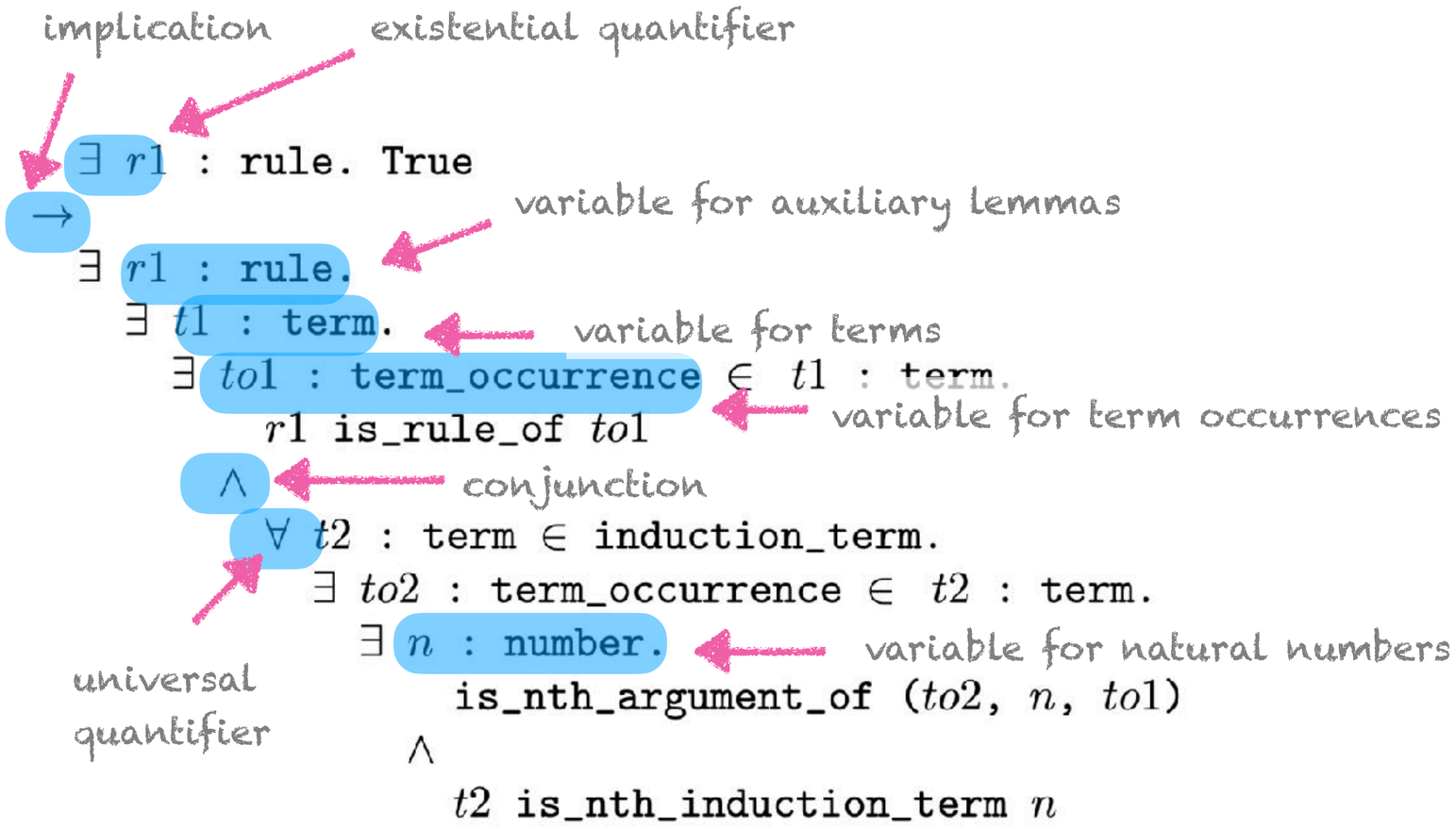


Example Assertion in LiFtEr (in Abstract Syntax)



Example Assertion in LiFtEr (in Abstract Syntax)

LiFtEr assertion: (proof goal * induction arguments) -> bool



```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

lemma "rev2 xs ys = rev1 xs @ ys"
  apply(induct xs ys rule: rev2.induct)
apply auto done
```

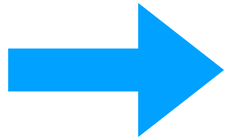
```
  ∃ r1 : rule. True
→
  ∃ r1 : rule.
    ∃ t1 : term.
      ∃ to1 : term_occurrence ∈ t1 : term.
          r1 is_rule_of to1
        ∧
          ∀ t2 : term ∈ induction_term.
            ∃ to2 : term_occurrence ∈ t2 : term.
              ∃ n : number.
                  is_nth_argument_of (to2, n, to1)
                ∧
                  t2 is_nth_induction_term n
```

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

lemma "rev2 xs ys = rev1 xs @ ys"
  apply(induct xs ys rule: rev2.induct)
apply auto done
```

```
∃ r1 : rule. True
→
∃ r1 : rule.
  ∃ t1 : term.
    ∃ to1 : term_occurrence ∈ t1 : term.
        r1 is_rule_of to1
      ∧
        ∀ t2 : term ∈ induction_term.
          ∃ to2 : term_occurrence ∈ t2 : term.
            ∃ n : number.
                is_nth_argument_of (to2, n, to1)
              ∧
                t2 is_nth_induction_term n
```



True

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2:: "'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

lemma "rev2 xs ys = rev xs @ ys"
  apply(induct ys xs rule: rev2.induct)
apply auto done
```

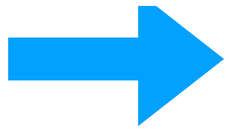
```
∃ r1 : rule. True
→
∃ r1 : rule.
  ∃ t1 : term.
    ∃ to1 : term_occurrence ∈ t1 : term.
        r1 is_rule_of to1
      ∧
        ∀ t2 : term ∈ induction_term.
          ∃ to2 : term_occurrence ∈ t2 : term.
            ∃ n : number.
                is_nth_argument_of (to2, n, to1)
              ∧
                t2 is_nth_induction_term n
```

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2:: "'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

lemma "rev2 xs ys = rev xs @ ys"
  apply(induct ys xs rule: rev2.induct)
apply auto done
```

```
∃ r1 : rule. True
→
∃ r1 : rule.
  ∃ t1 : term.
    ∃ to1 : term_occurrence ∈ t1 : term.
        r1 is_rule_of to1
      ∧
        ∀ t2 : term ∈ induction_term.
          ∃ to2 : term_occurrence ∈ t2 : term.
            ∃ n : number.
                is_nth_argument_of (to2, n, to1)
              ∧
                t2 is_nth_induction_term n
```



False

Candidate induction?

Candidate induction?

Can't you tell me what arguments to pass to the induct tactic?

Candidate induction?

Can't you tell me what arguments to pass to the induct tactic?

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

smart_induct

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

goal

smart_induct

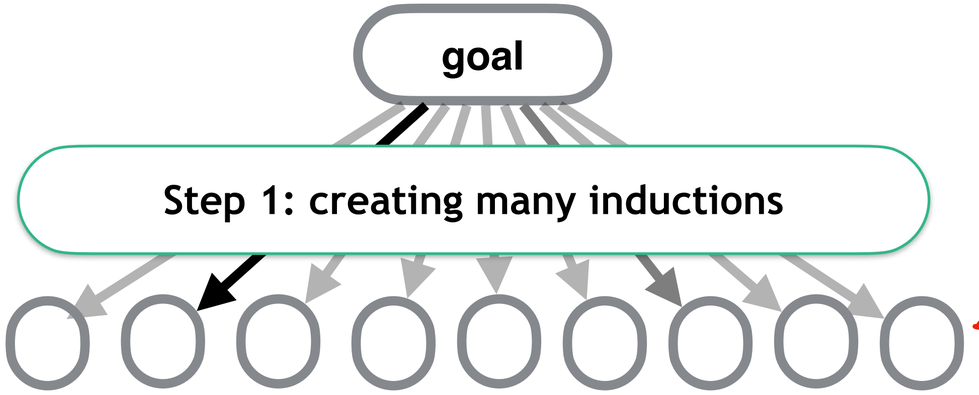
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

smart_induct



new users

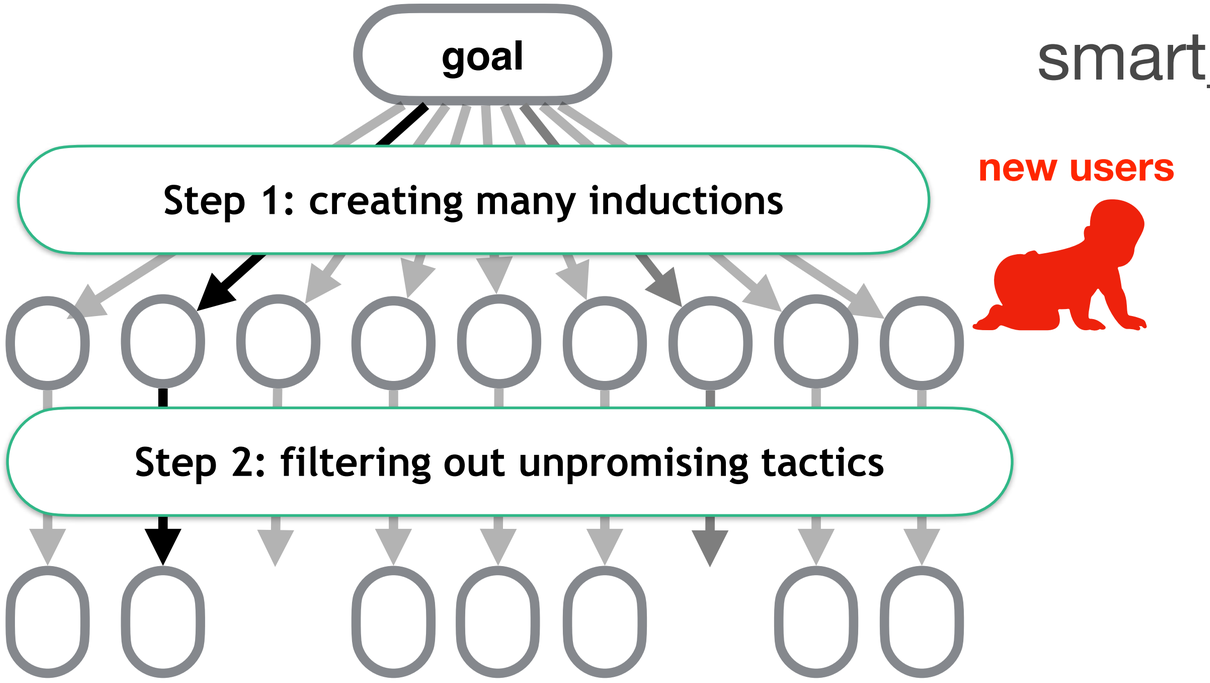
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

smart_induct



new users

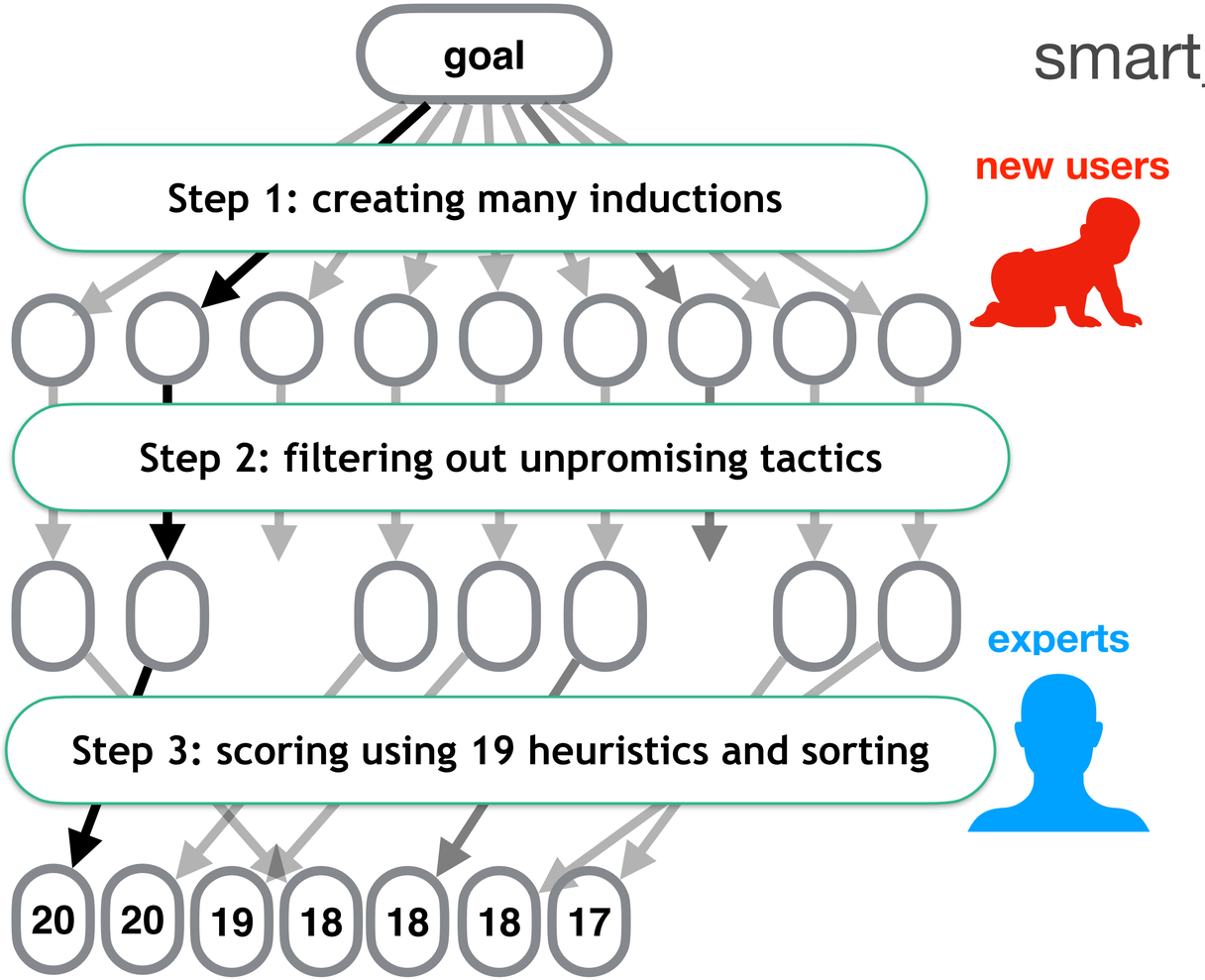
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

smart_induct



new users

experts

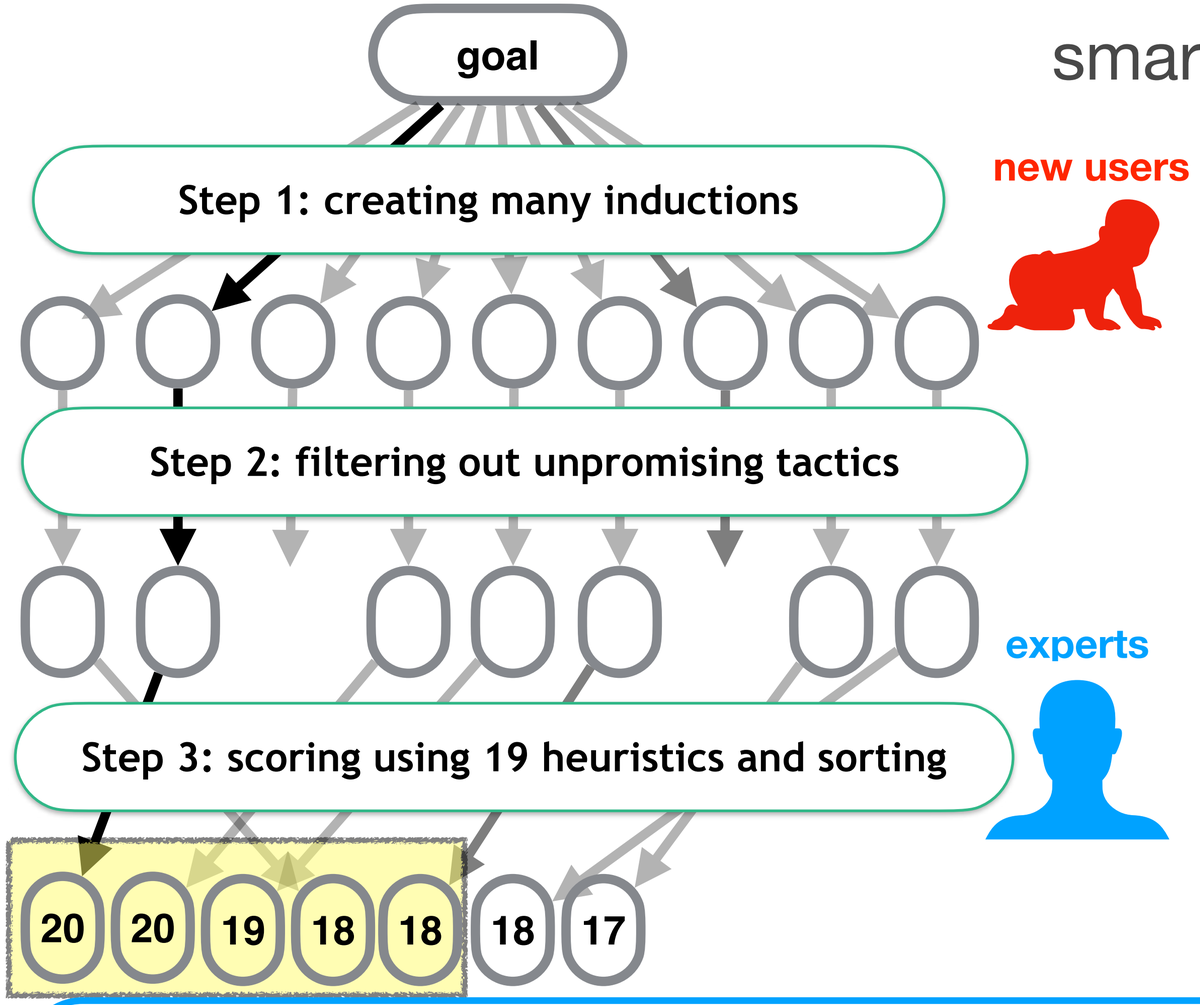
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

smart_induct



new users

experts

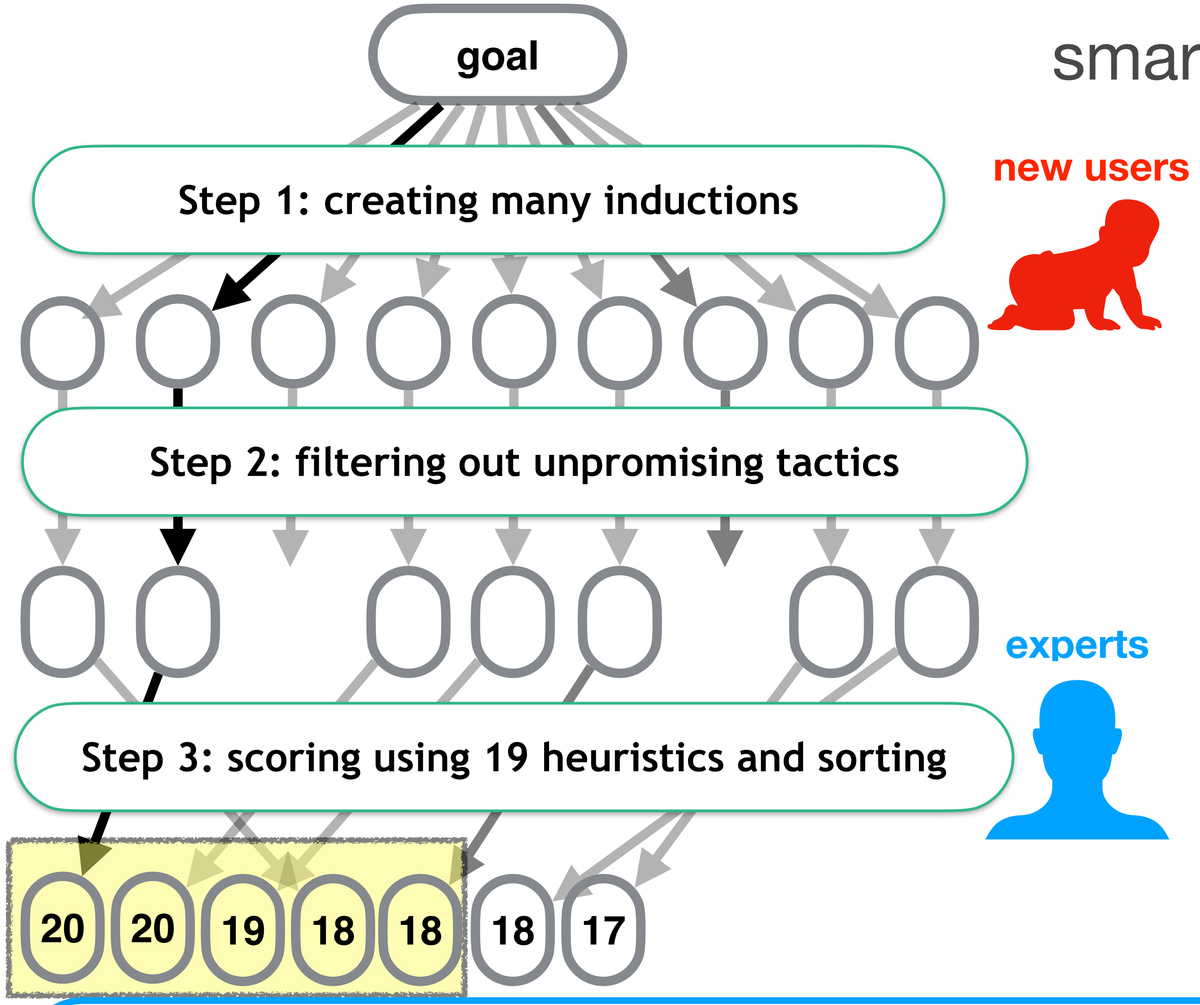
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

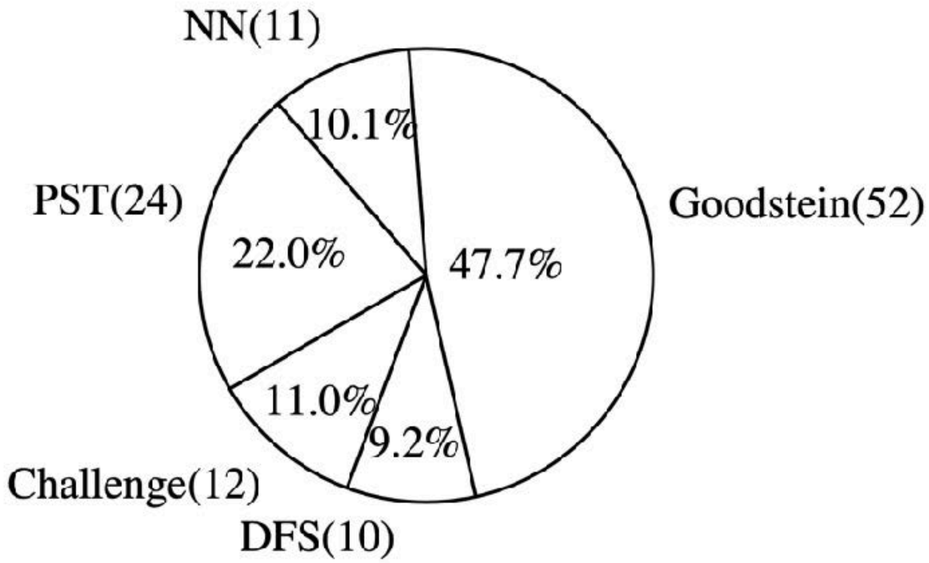
smart_induct



new users

experts

Dataset for evaluation
(109 proofs by induction from the AFP entries)



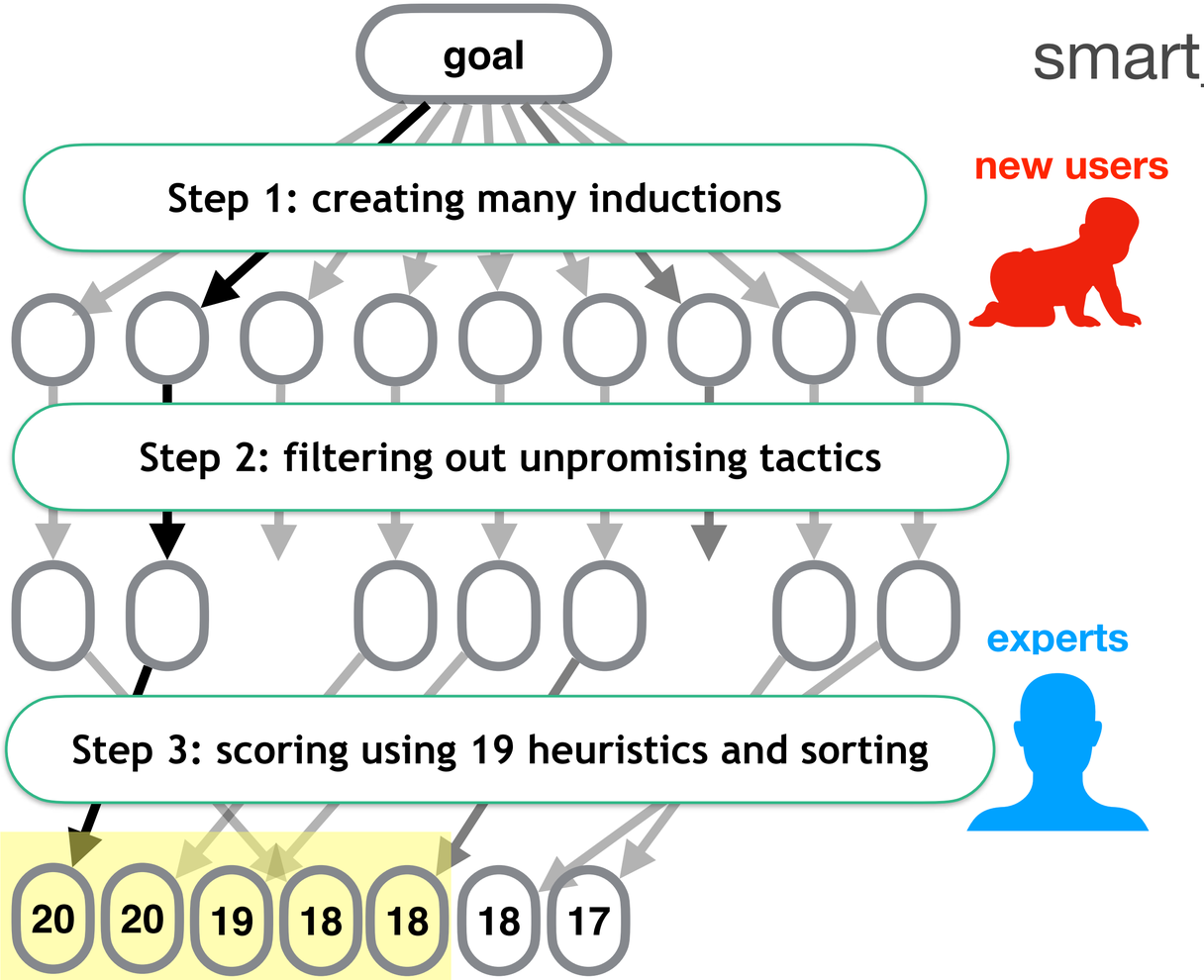
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

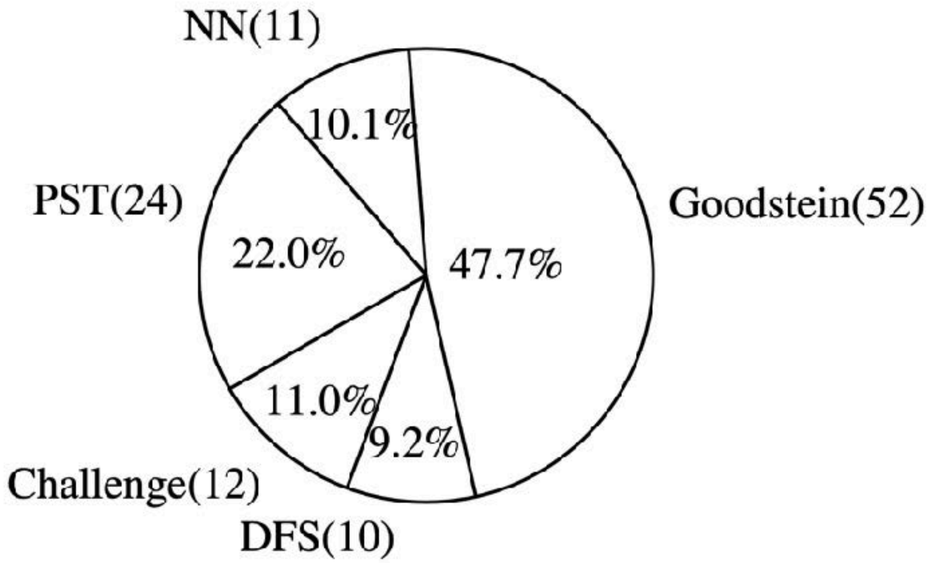
smart_induct



new users

experts

Dataset for evaluation
(109 proofs by induction from the AFP entries)



Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

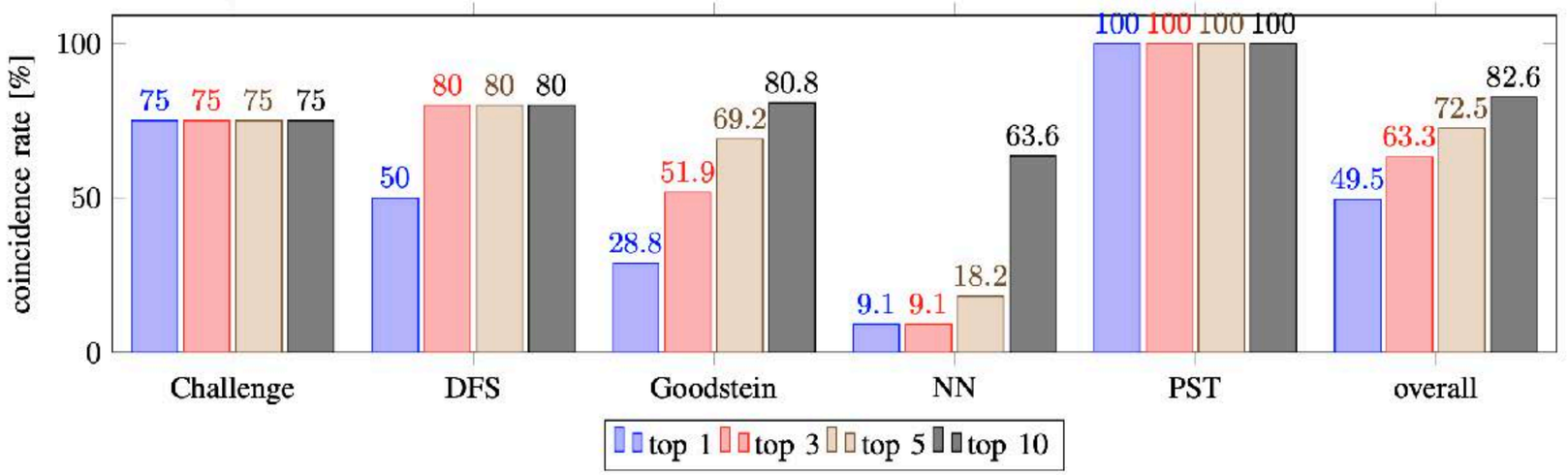
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

Coincidence rates



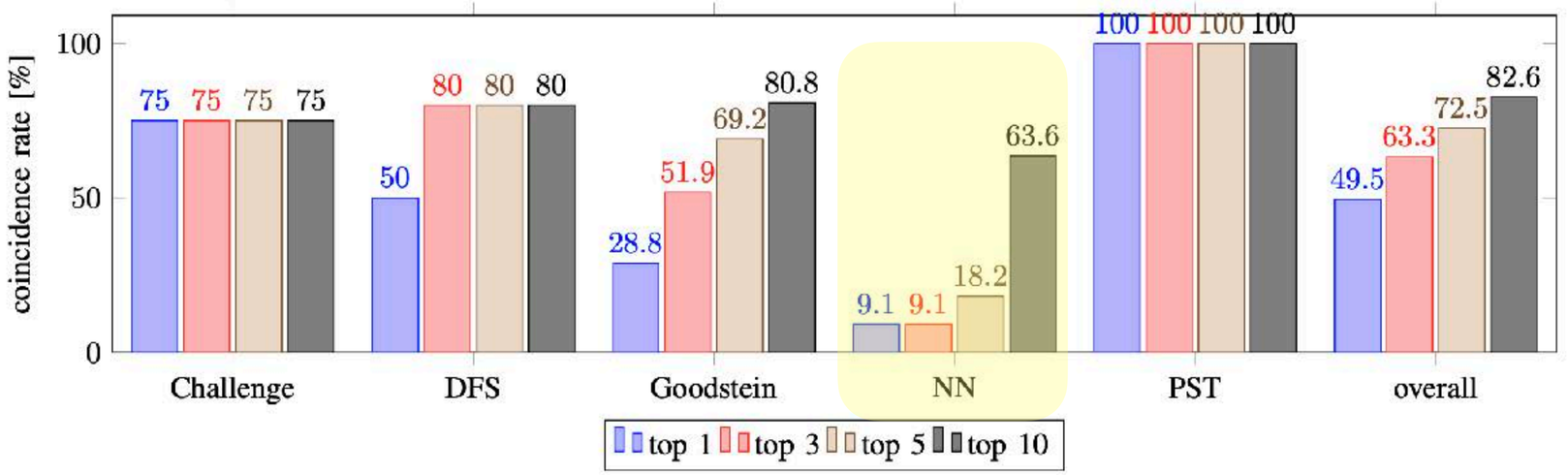
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

Coincidence rates



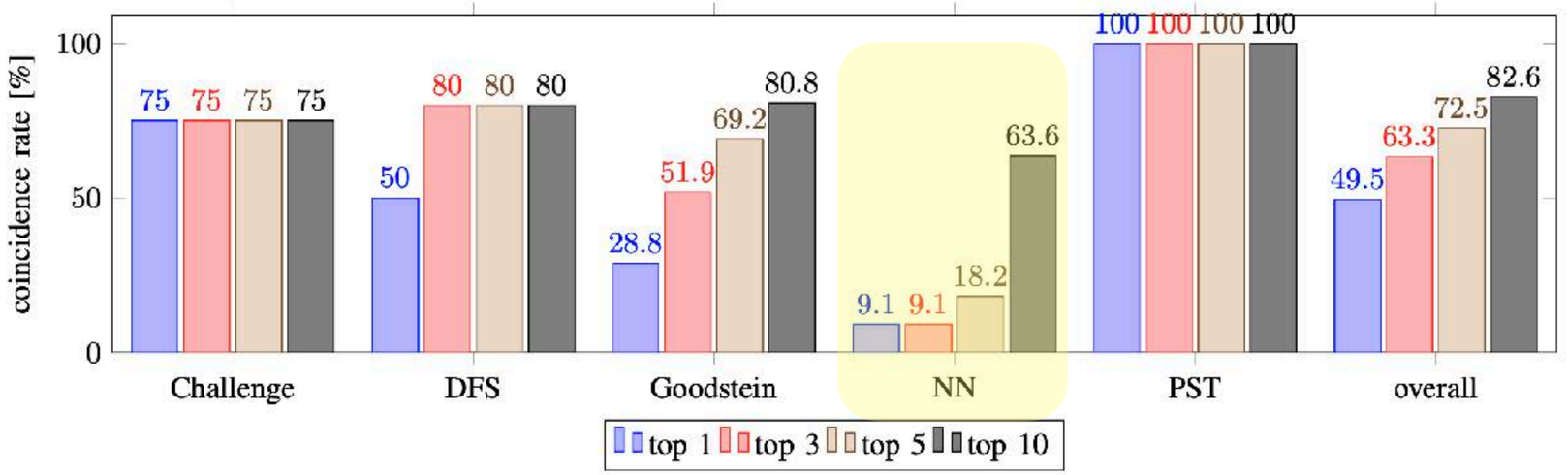
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

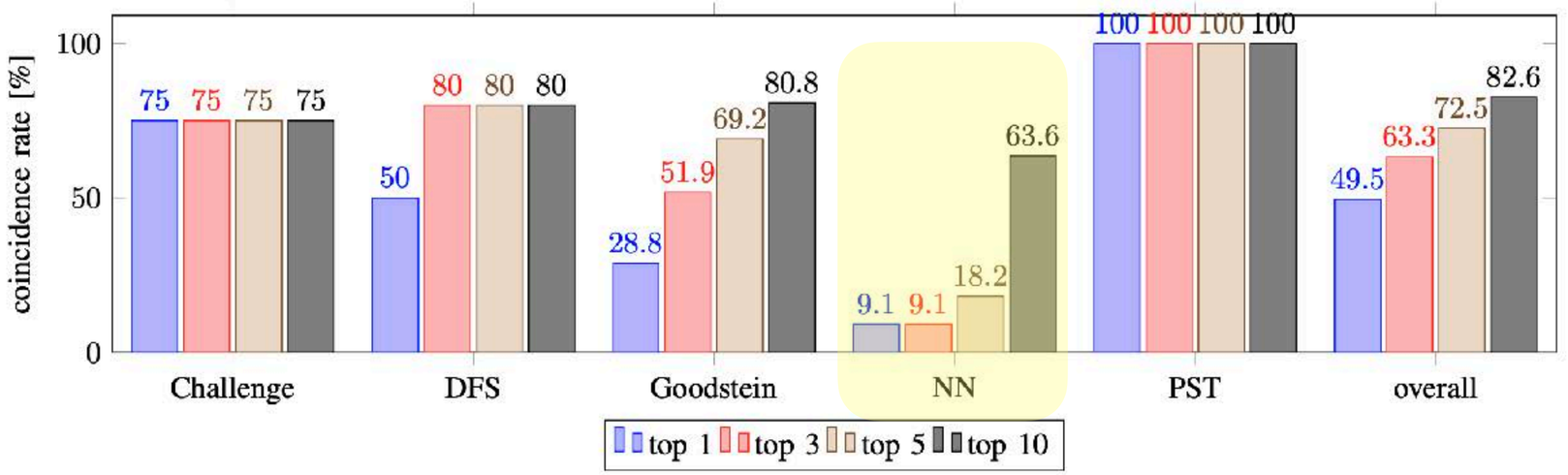
Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

Coincidence rates

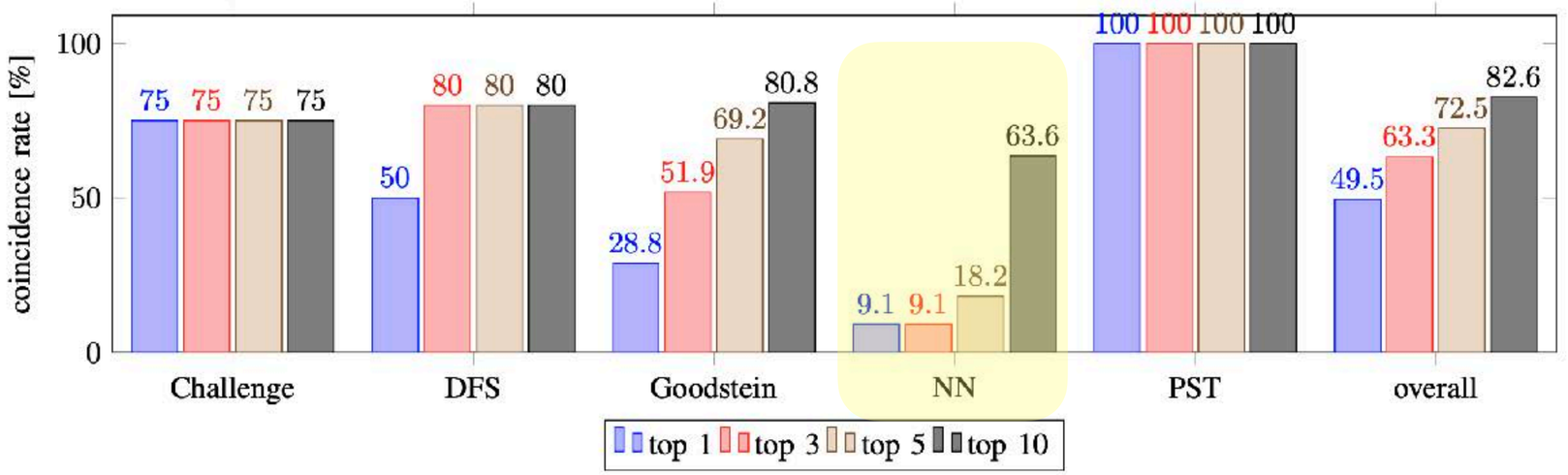


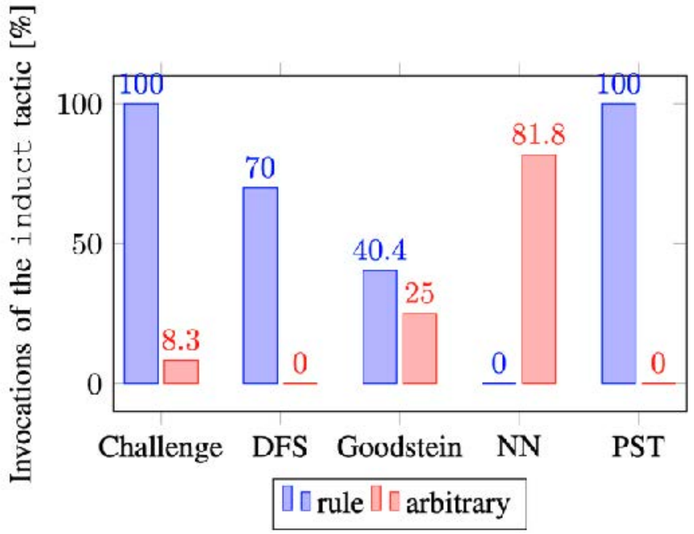
Coincidence rates



Is smart_induct Domain dependent?

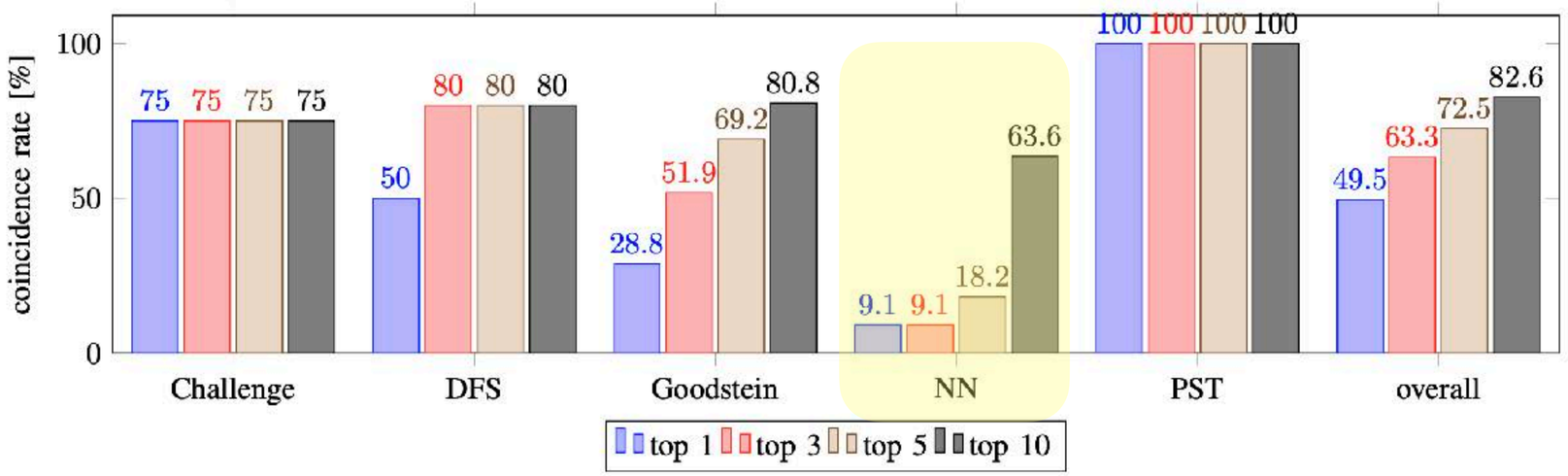
Coincidence rates

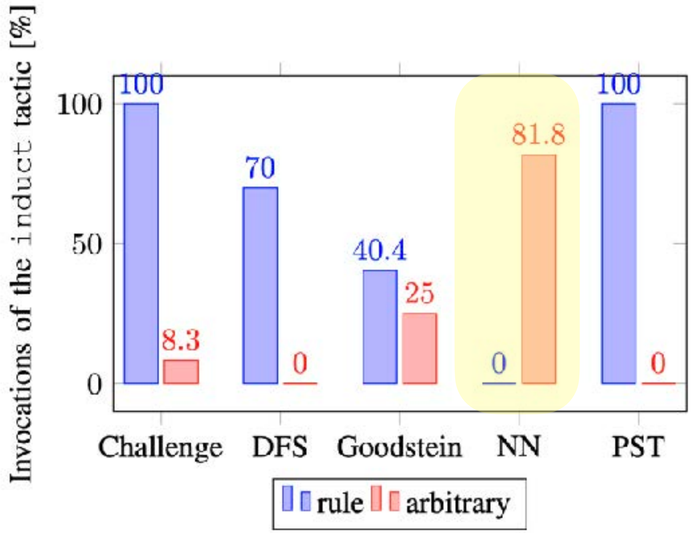




Is smart_induct Domain dependent?

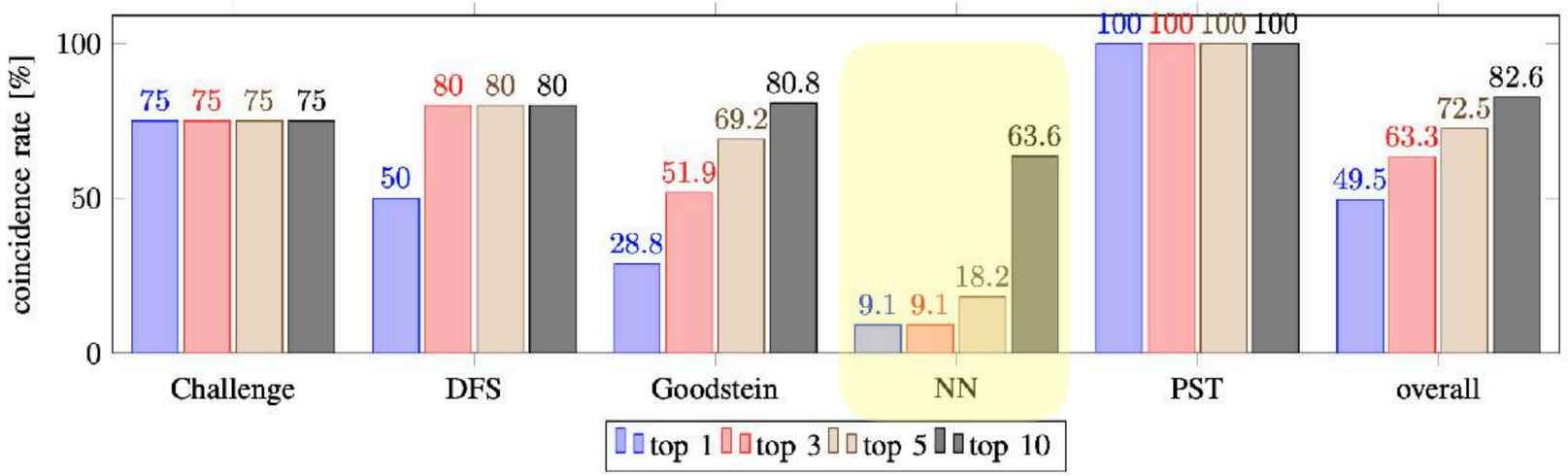
Coincidence rates

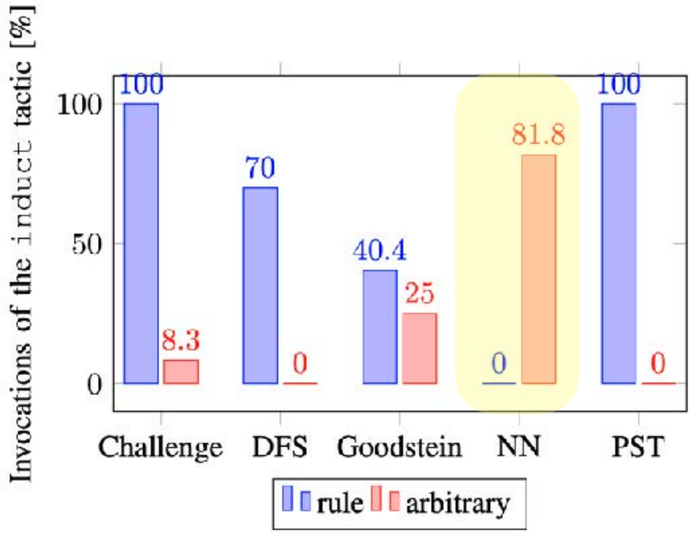


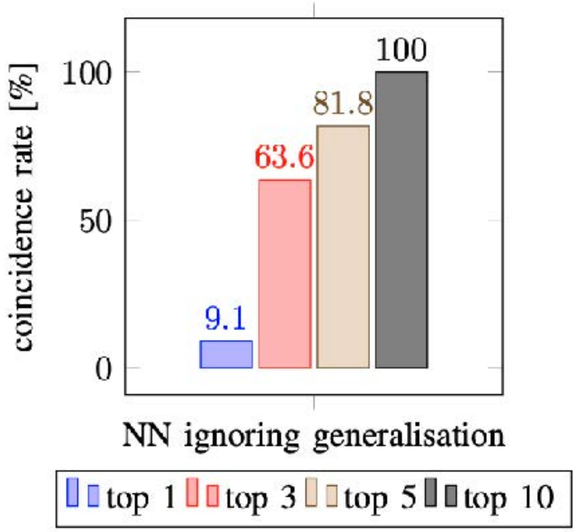


Is smart_induct Domain dependent?

Coincidence rates

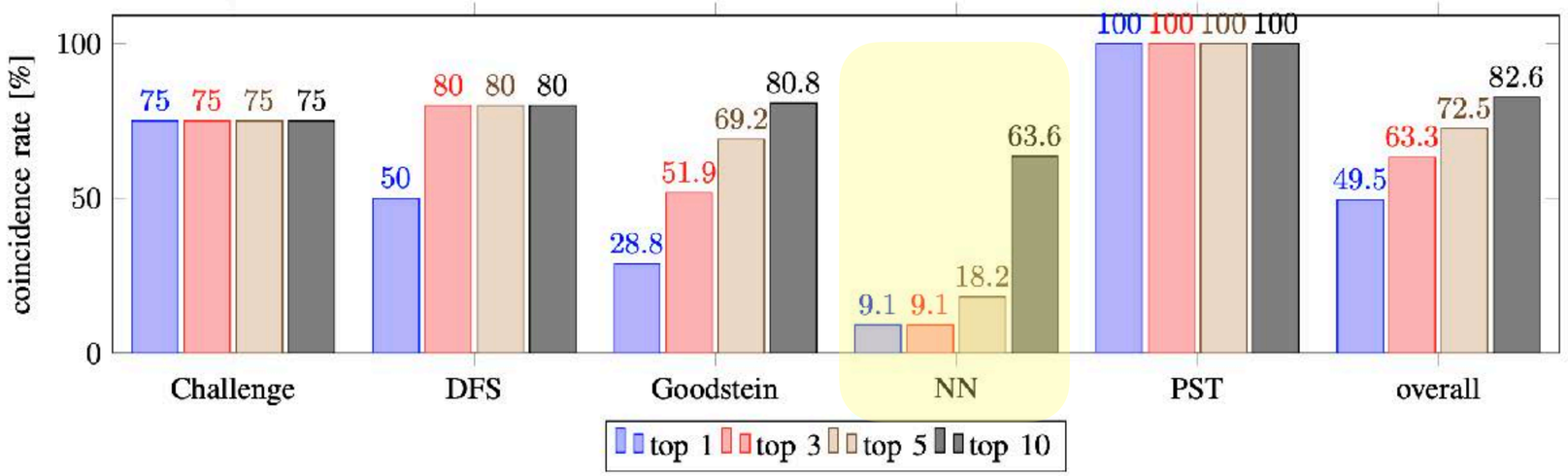


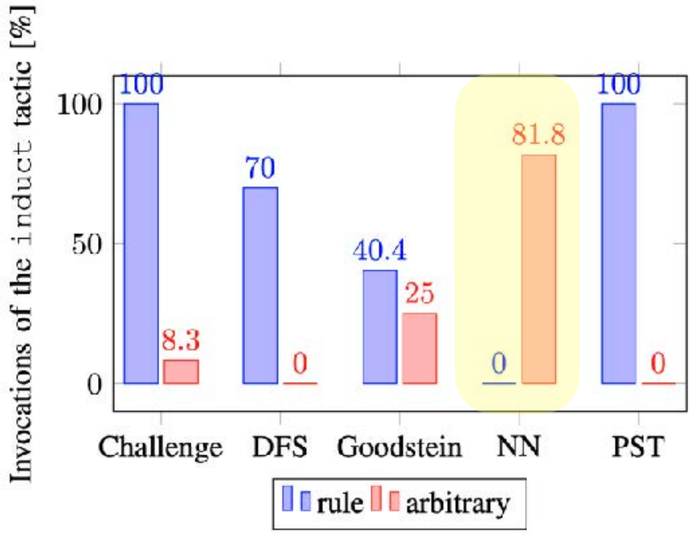


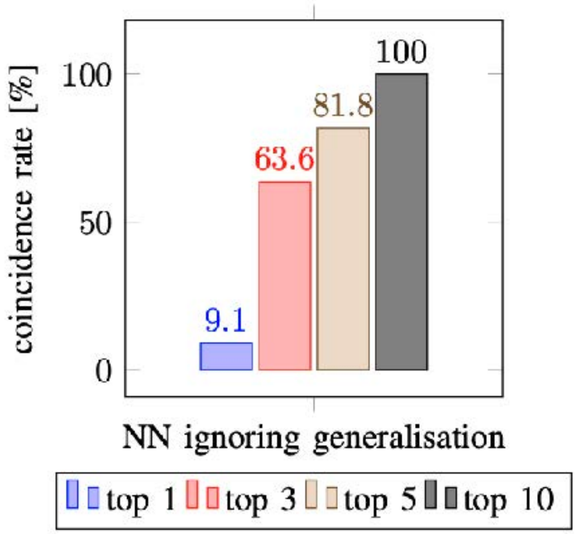


Is smart_induct Domain dependent?

Coincidence rates







Is smart_induct Domain dependent?

smart_induct is bad at generalisation heuristics.

Path to automatic induction

Step 1: Tactic recommendation.

Step 2: Tactic argument recommendation.

- a: Guided backtracking search with tracing (PSL).
- b: Logical Feature Extraction (LiFtEr).
- c: smart_induct using LiFtEr.

Step 3: Putting things together.

Program 4 Automatic inductive prover without SeLFiE

```
strategy Auto_Solve = Thens [Auto, IsSolved]
strategy PSL_WO_SeLFiE =
  Ors [
    Auto_Solve,
    PThenOne [Dynamic (Induct), Auto_Solve]
    PThenOne [
      Dynamic (Induct),
      Thens [Auto, RepeatN(Hammer), IsSolved]
    ]
  ]
```

Program 5 Automatic inductive prover with SeLFiE

```
strategy PSL_W_SeLFiE =
  Ors [
    Auto_Solve,
    PThenOne [Semantic_Induct, Auto_Solve]
    PThenOne [
      Semantic_Induct,
      Thens [Auto, RepeatN(Hammer), IsSolved]
    ]
  ]
```

Program 4 Automatic inductive prover without SeLFiE

```
strategy Auto_Solve = Thens [Auto, IsSolved]
strategy PSL_WO_SeLFiE =
  Ors [
    Auto_Solve,
    PThenOne [Dynamic (Induct), Auto_Solve]
    PThenOne [
      Dynamic (Induct),
      Thens [Auto, RepeatN(Hammer), IsSolved]
    ]
  ]
```

Program 5 Automatic inductive prover with SeLFiE

```
strategy PSL_W_SeLFiE =
  Ors [
    Auto_Solve,
    PThenOne [Semantic_Induct, Auto_Solve]
    PThenOne [
      Semantic_Induct,
      Thens [Auto, RepeatN(Hammer), IsSolved]
    ]
  ]
```

Success rate

timeouts	Program 5	Program 4
0.3[s]	11.0%	1.2%
1.0[s]	25.6%	1.7%
3.0[s]	28.2%	21.9%
10.0[s]	34.9%	28.0%
30.0[s]	45.8%	38.3%

Bibliography

Step 1: Tactic recommendation.
https://doi.org/10.1145/3238147.3238210

Step 2: Tactic argument recommendation.

a: Guided backtracking search with tracing (PSL).
https://doi.org/10.1007/978-3-319-63046-5_32

b: Logical Feature Extraction (LiFtEr).
https://doi.org/10.1007/978-3-030-34175-6_14

c: smart_induct using LiFtEr.
https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_32

d: Semantic-aware Logical Feature Extraction (SeLFiE).
https://arxiv.org/pdf/2010.10296.pdf

e: sem_ind using SeLFiE.
https://arxiv.org/pdf/2009.09215.pdf

f: conjecturing using abductive-reasoning.
https://doi.org/10.1007/978-3-319-96812-4_19

fin!