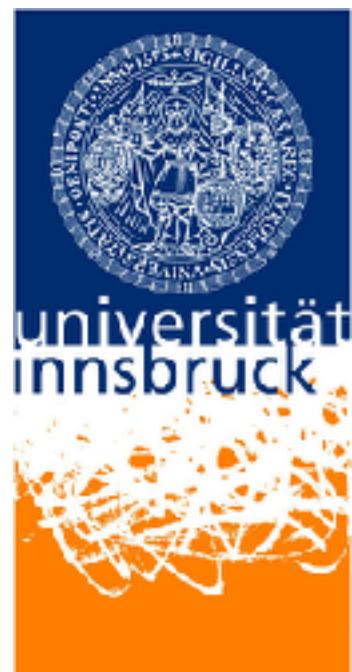


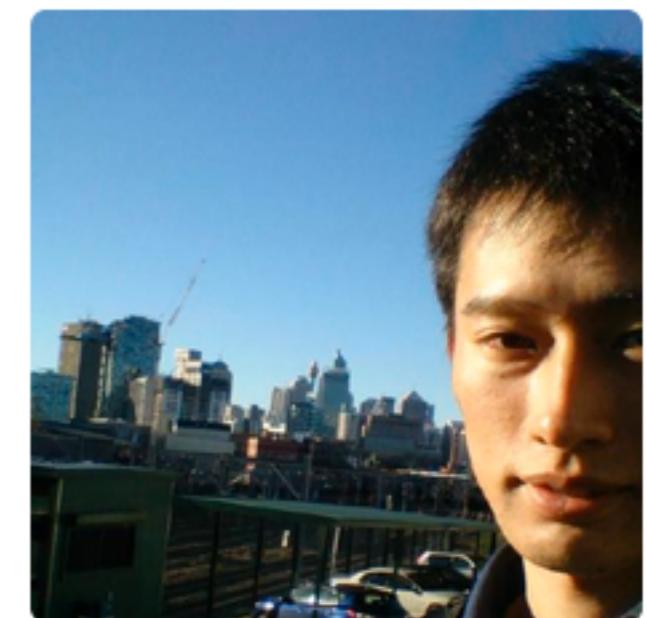
Automating proof by induction in Isabelle/HOL using DSLs



Yutaka Nagashima



**CZECH INSTITUTE
OF INFORMATICS
ROBOTICS AND
CYBERNETICS
CTU IN PRAGUE**



Yutaka Ng

Automating proof by induction in Isabelle/HOL using DSLs



Yutaka Nagashima



**CZECH INSTITUTE
OF INFORMATICS
ROBOTICS AND
CYBERNETICS
CTU IN PRAGUE**



Yutaka Ng

https://github.com/data61/PSL/blob/master/slide/2020_NUS.pdf

Background

2013 ~ 2017



with Prof. Gerwin

Background

2013 ~ 2017

Intern &
Engineer



with Prof. Gerwin

Background

2013 ~ 2017

Intern &
Engineer



with Prof. Gerwin

PhD in
AI for theorem proving

Background

2013 ~ 2017

Intern &
Engineer



with Prof. Gerwin

PhD in
AI for theorem proving



2017 ~ 2018



with Prof. Cezary Kaliszyk

Background

2013 ~ 2017

Intern &
Engineer



with Prof. Gerwin

PhD in
AI for theorem proving



with Prof. Cezary Kaliszyk

2018 ~
2020/21



with Dr. Josef Urban

Background

2013 ~ 2017

Intern &
Engineer



Cogent

with Prof. Gerwin

PhD in
AI for theorem proving



with Prof. Cezary Kaliszyk

2018 ~
2020/21



with Dr. Josef Urban

Background

2013 ~ 2017

Intern &
Engineer



Cogent

PhD in
AI for theorem proving



PSL

2017 ~ 2018

with Prof. Gerwin

2018 ~
2020/21



with Dr. Josef Urban

Background

2013 ~ 2017

Intern & Engineer



PhD in AI for theorem proving



2017 ~ 2018



<http://ci-mifii.mifii.edu.pl/user/cezko>

**2018 ~
2020/21**



<http://ai4reason.org/members.html>



Cogent



https://github.com/data61/PSL/blob/master/slide/2020_NUS.pdf

Background

2013 ~ 2017

Intern & Engineer



PhD in AI for theorem proving



2017 ~ 2018



<http://www.cse.unsw.edu.au/~kleing/>



Cogent

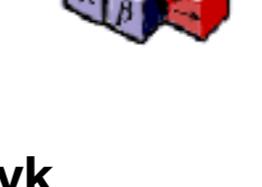
**2018 ~
2020/21**



<http://ai4reason.org/members.html>



PSL



PaMpeR



LiFtEr

https://github.com/data61/PSL/blob/master/slide/2020_NUS.pdf

Background

2013 ~ 2017

Intern & Engineer



Cogent

PhD in AI for theorem proving



PSL

2017 ~ 2018



LiFtEr

**2018 ~
2020/21**

<http://ci-informatik.uibk.ac.at/users/cek/>



smart induct

Background

2013 ~ 2017

Intern &
Engineer



Cogent

PhD in
AI for theorem proving



PSL

2017 ~ 2018

2018 ~
2020/21



LiFtEr

with Dr. Josef Urban



PaMpeR

smart_induct

PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```

PSL: Proof Strategy Language

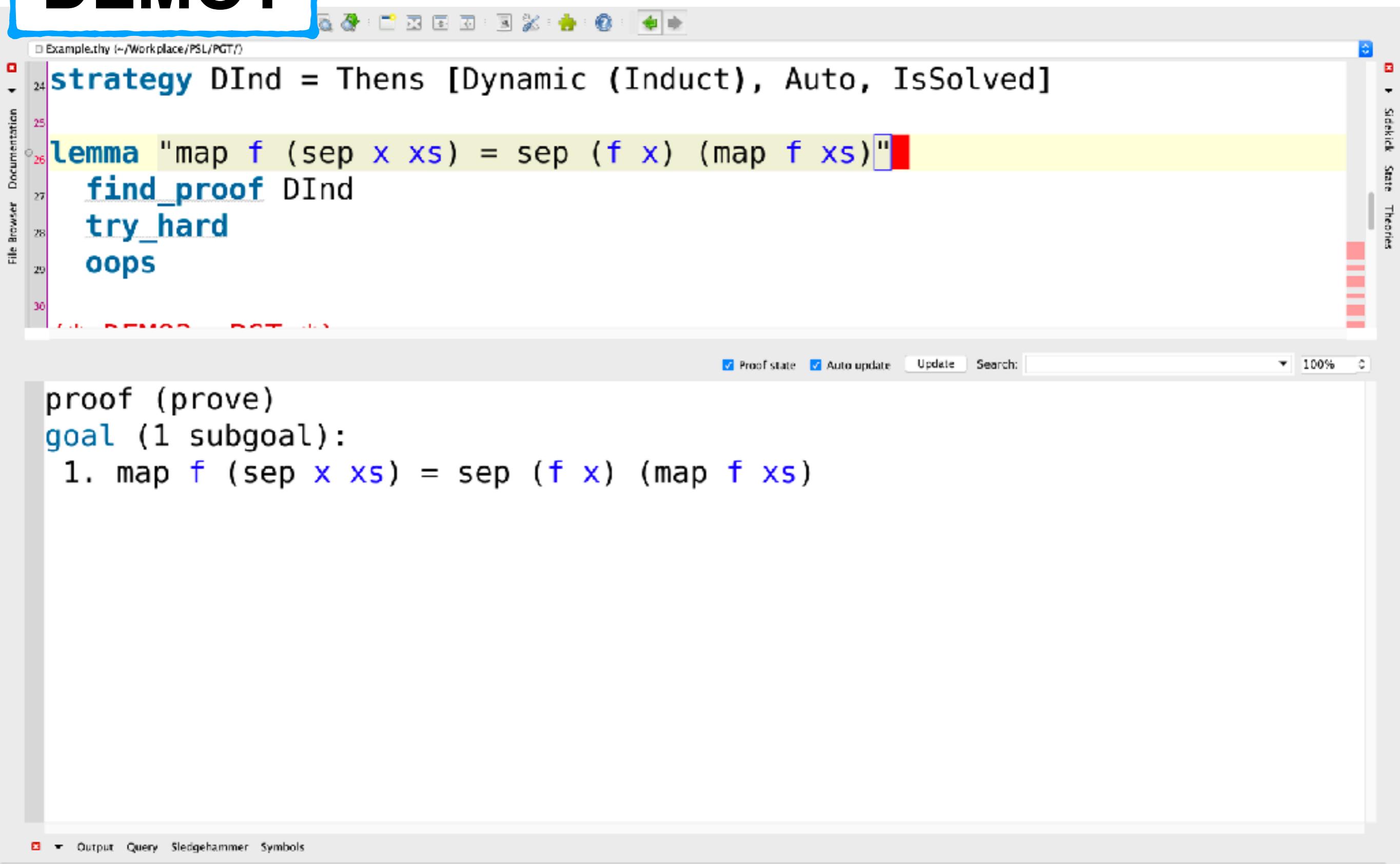
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO!

DEMO1

https://github.com/data61/PSL/blob/master/slide/2020_NUS.pdf



The screenshot shows the Isabelle proof assistant interface. The top part displays a theory file named "Example.thy" with the following content:

```
strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved]

lemma "map f (sep x xs) = sep (f x) (map f xs)"
  find_proof DInd
  try_hard
oops
```

The lemma "map f (sep x xs) = sep (f x) (map f xs)" is highlighted in yellow, indicating it is the current goal. The "find_proof" command has been issued, and the proof attempt has failed, as indicated by the "oops" command.

The bottom part of the interface shows the proof state:

```
proof (prove)
goal (1 subgoal):
  1. map f (sep x xs) = sep (f x) (map f xs)
```

The proof state is empty, and the proof attempt has failed.

At the bottom of the interface, there are tabs for Output, Query, Sledgehammer, and Symbols, and a status bar showing "26.48 (535/952)" and "Isabelle, isabelle, UTF-8-isabelle) in memory U.. 377/535MB 6:11 PM".

DEMO1

github.com/data61/PSL/blob/master/slide/2020_NUS.pdf

The screenshot shows the Isabelle proof assistant interface. The main window displays a theory file named "Example.thy". The code includes a strategy definition and a lemma:

```
strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved]  
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
  find_proof DInd  
  try_hard  
oops
```

The "oops" command at line 30 indicates that the proof attempt failed. The interface has a toolbar at the top, a vertical navigation bar on the left, and a vertical color-coded bar on the right. The bottom of the screen shows the command history and system status.

Number of lines of commands: 3
Number of lines of commands: 3
apply (induct xs rule: Example.sep.induct)
apply auto
done

Output Query Sledgehammer Symbols

27.18 (553/952) (isabelle, isabelle, UTF-8-isabelle) | nmrcd U.. 382/535MB 6:11 PM

DEMO1

github.com/data61/PSL/blob/master/slide/2020_NUS.pdf

The screenshot shows the Isabelle proof assistant interface. The main window displays a theory file named "Example.thy". The code includes a strategy definition and a lemma about the map function over sep lists. The proof attempt has stalled at the "find_proof" step. The status bar at the bottom shows the number of proofs (27.18), the current theory (Isabelle), and the time (6:11 PM).

```
strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved]

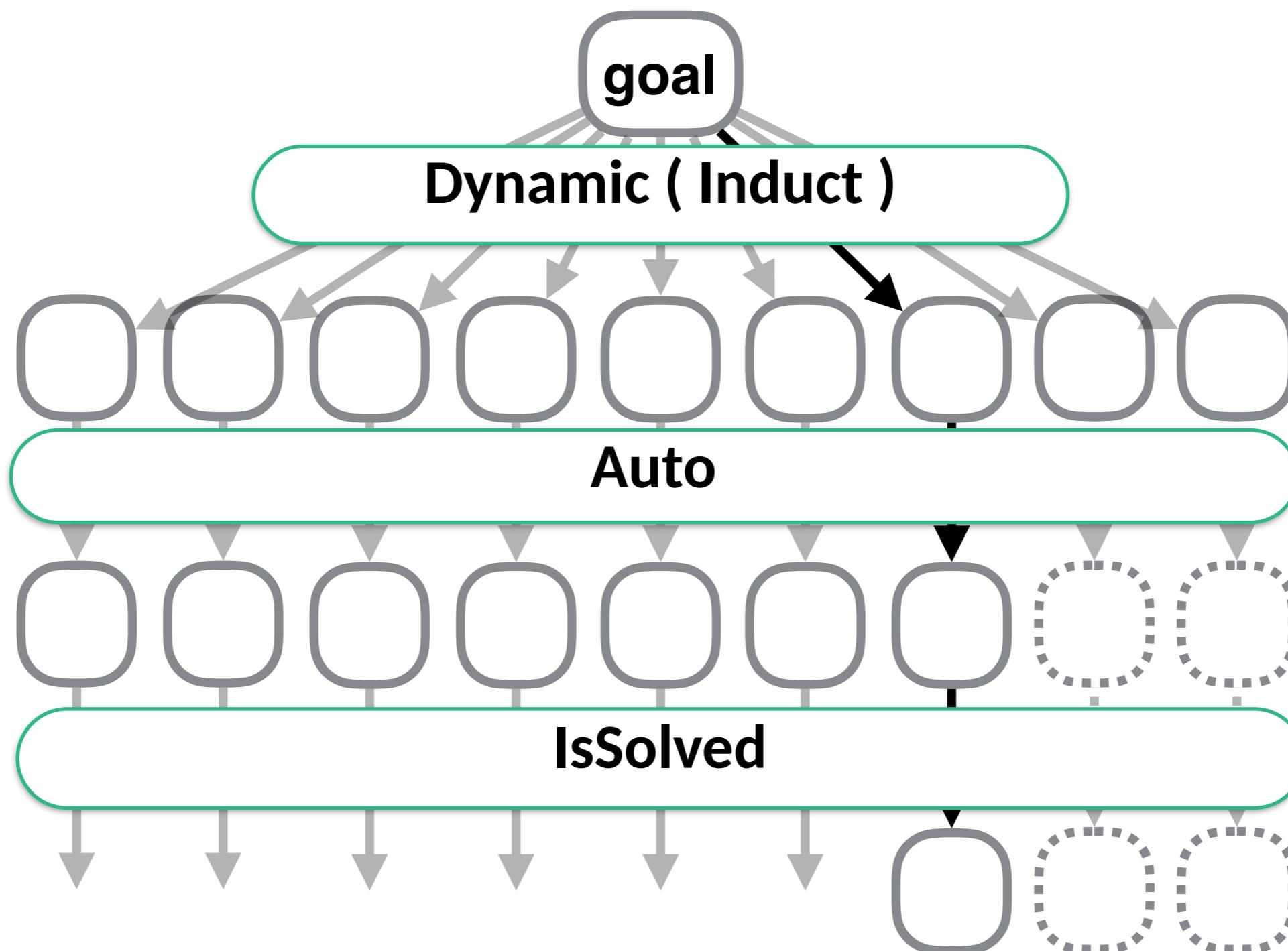
lemma "map f (sep x xs) = sep (f x) (map f xs)"
  find_proof DInd
  try_hard
oops
```

Number of lines of commands: 3
Number of lines of commands: 3
apply (induct xs rule: Example.sep.induct)
apply auto
done

What happened?

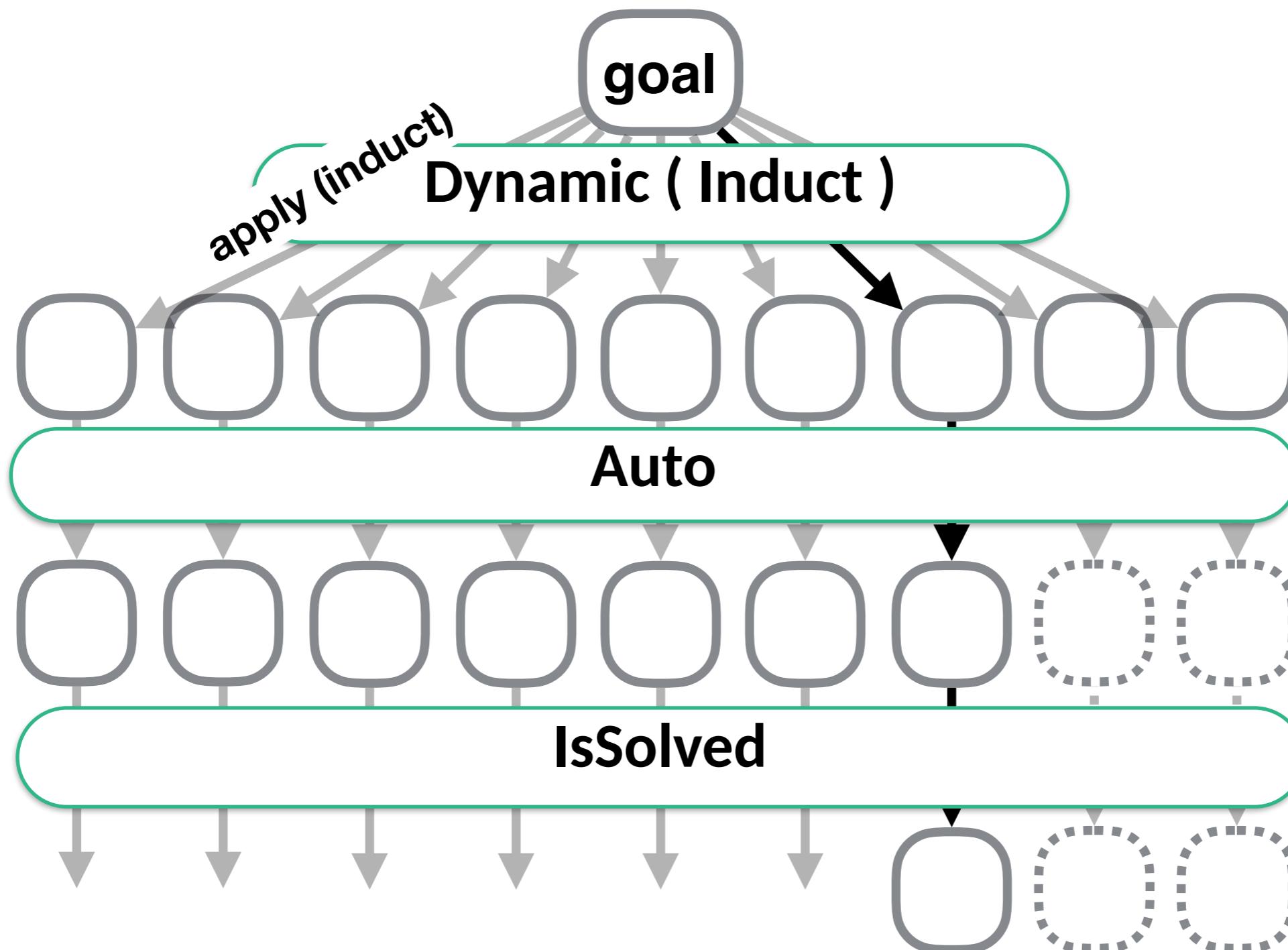
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



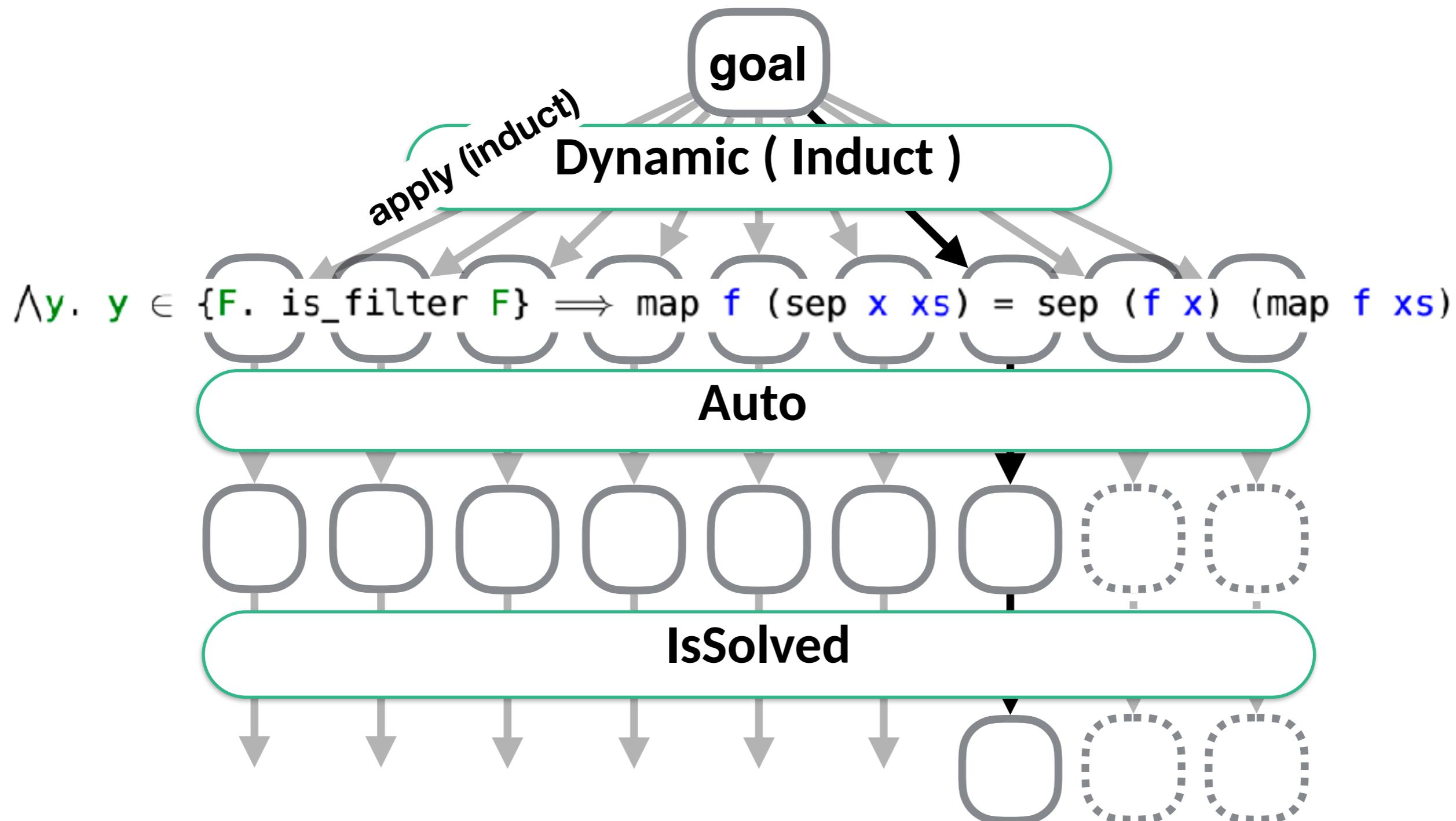
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



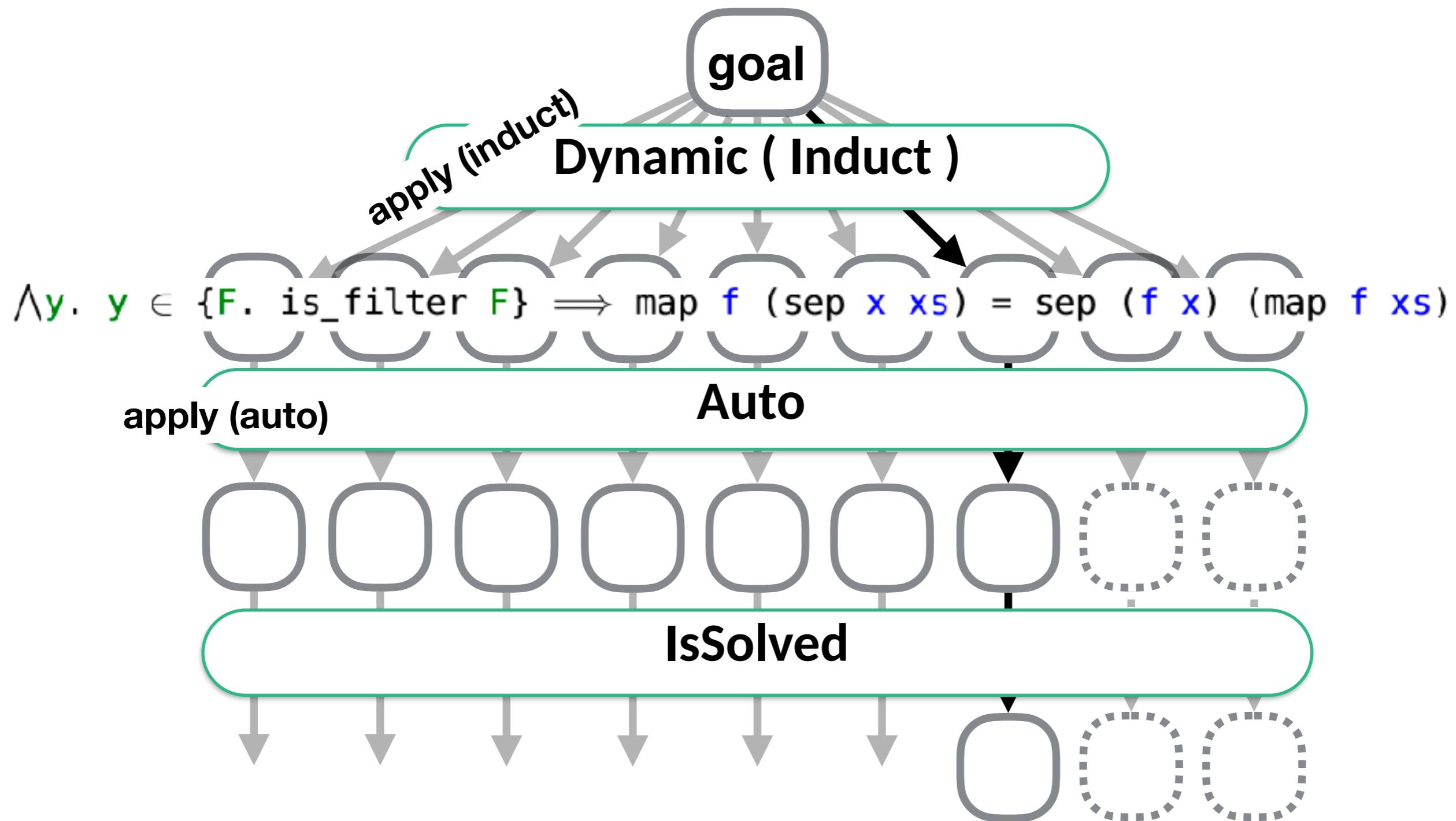
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



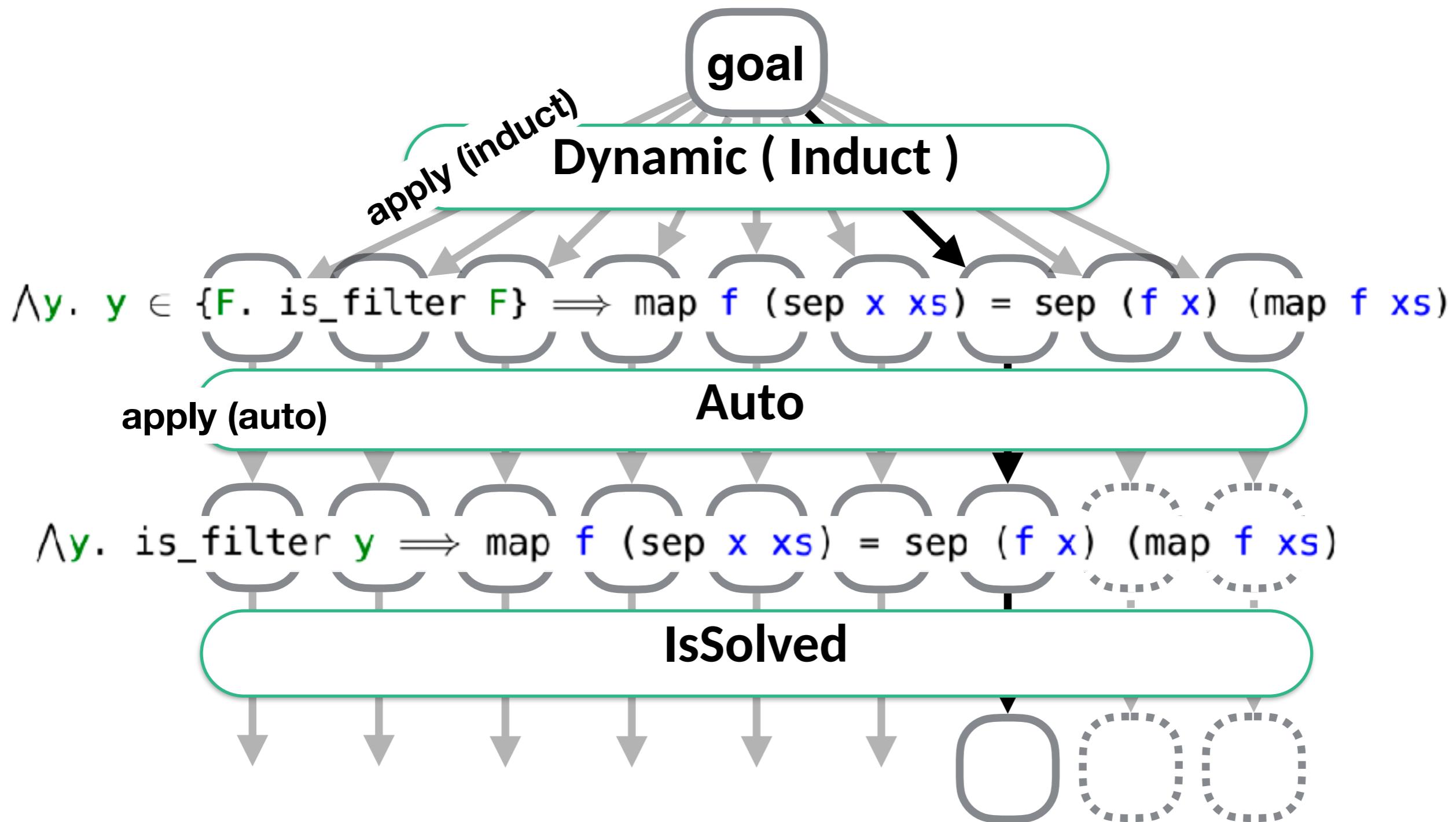
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



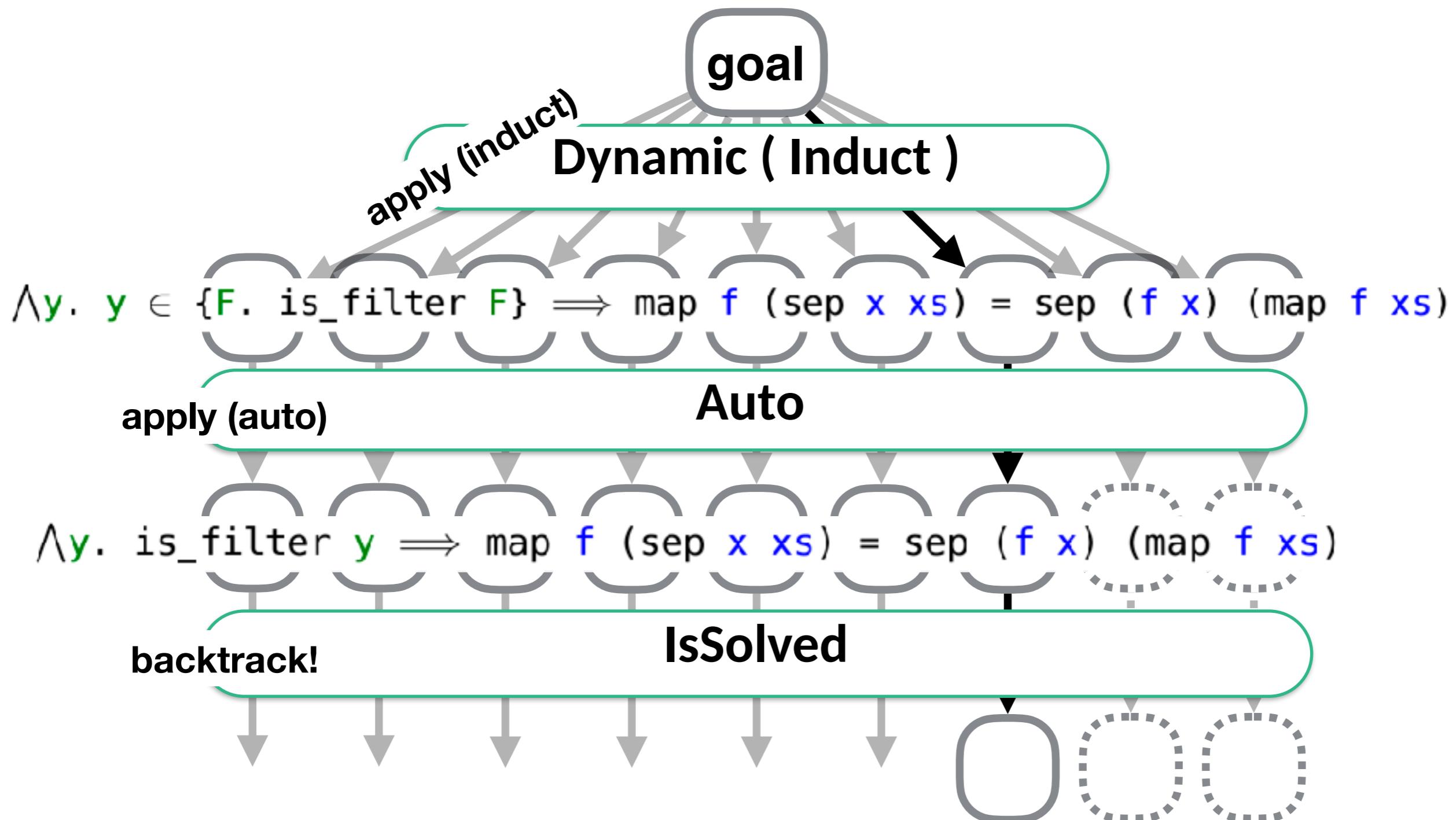
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



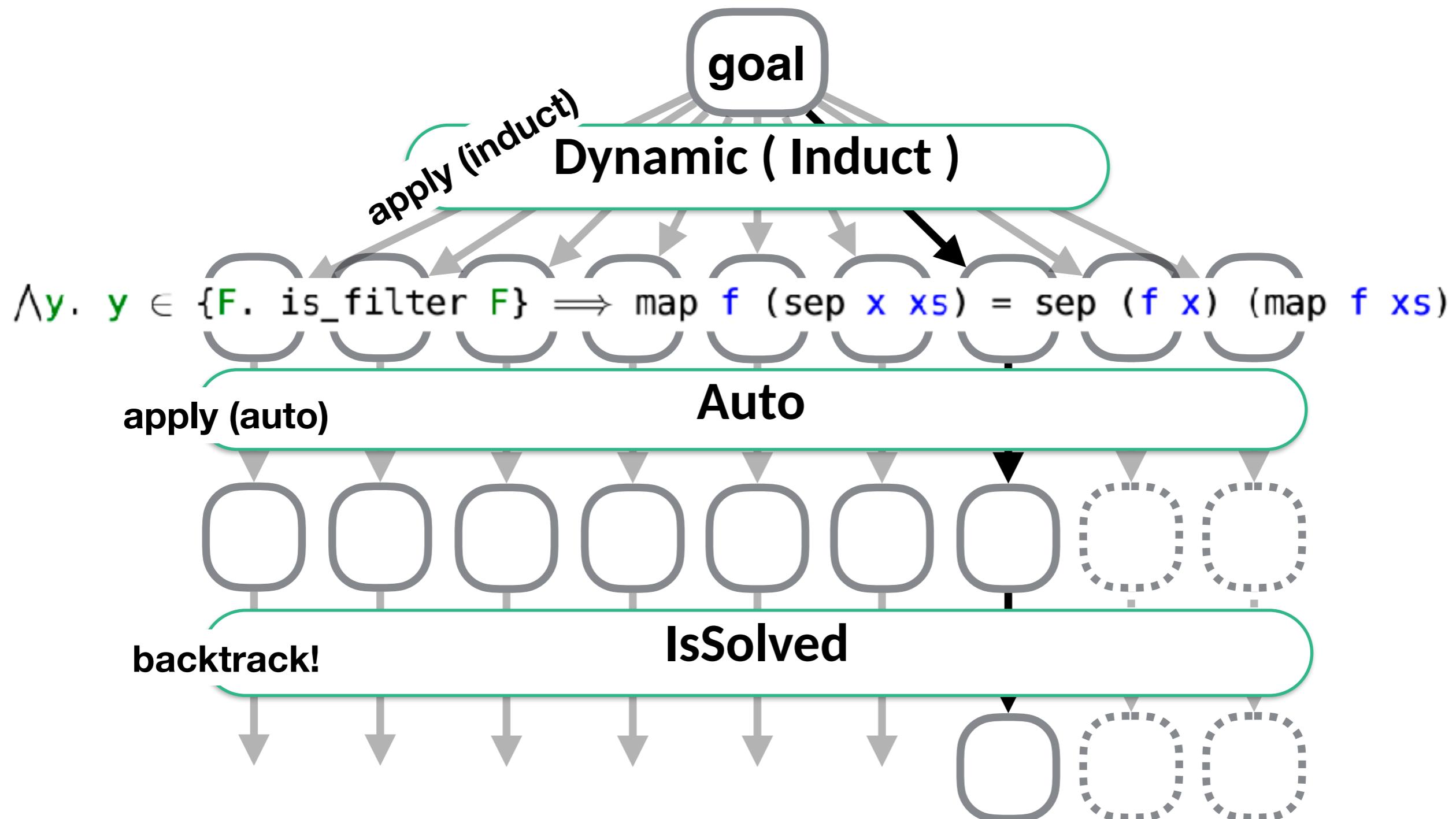
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



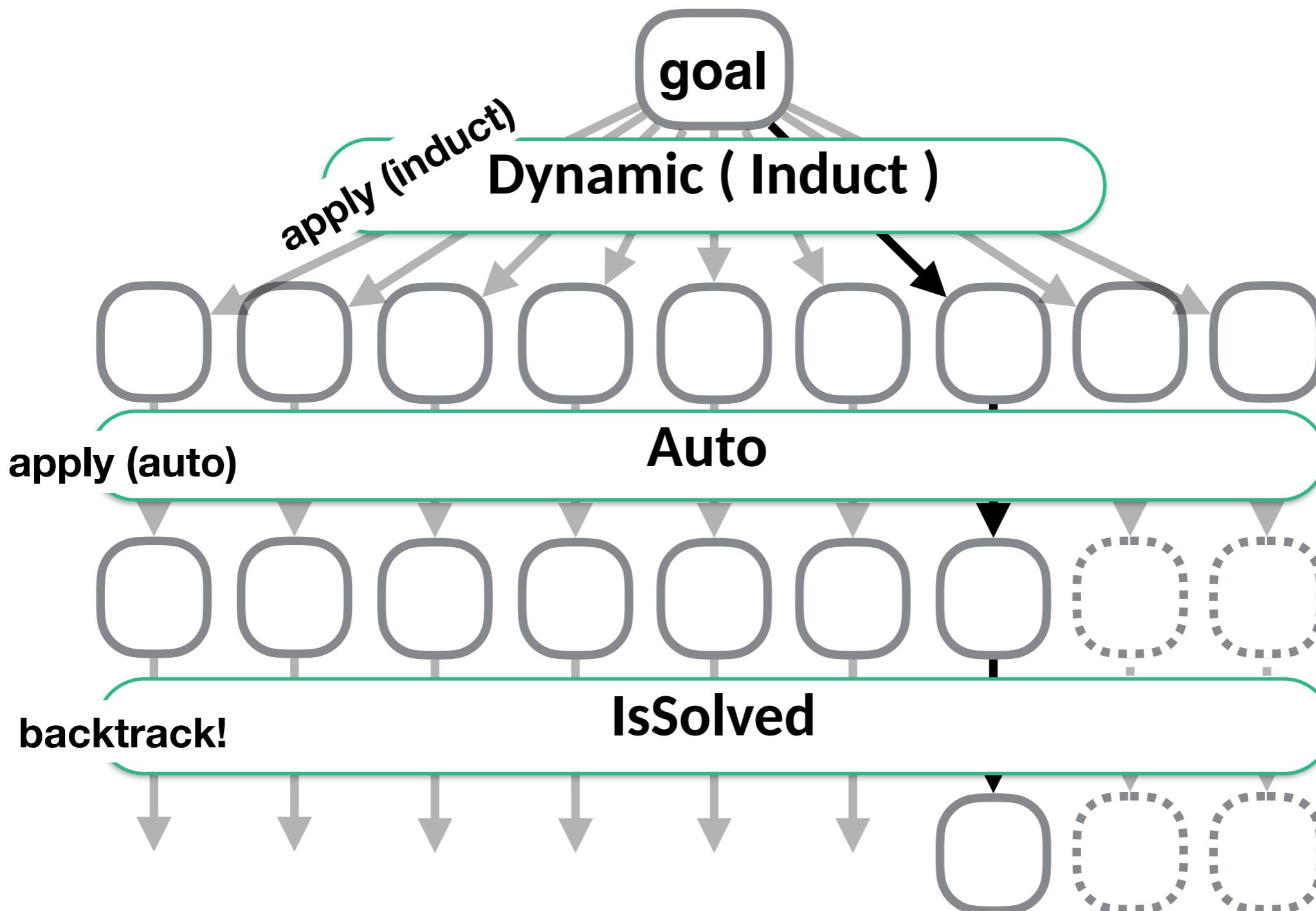
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



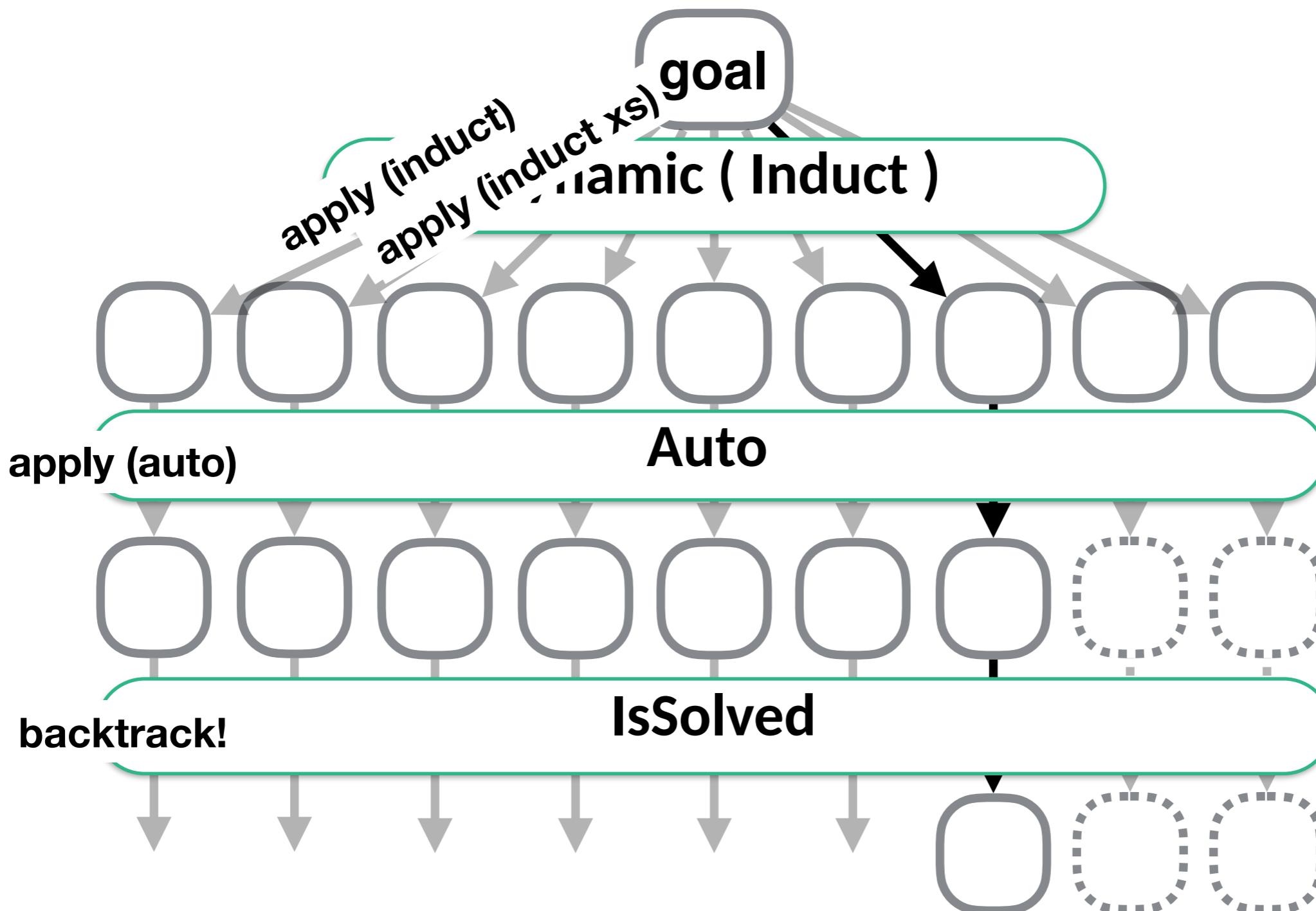
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



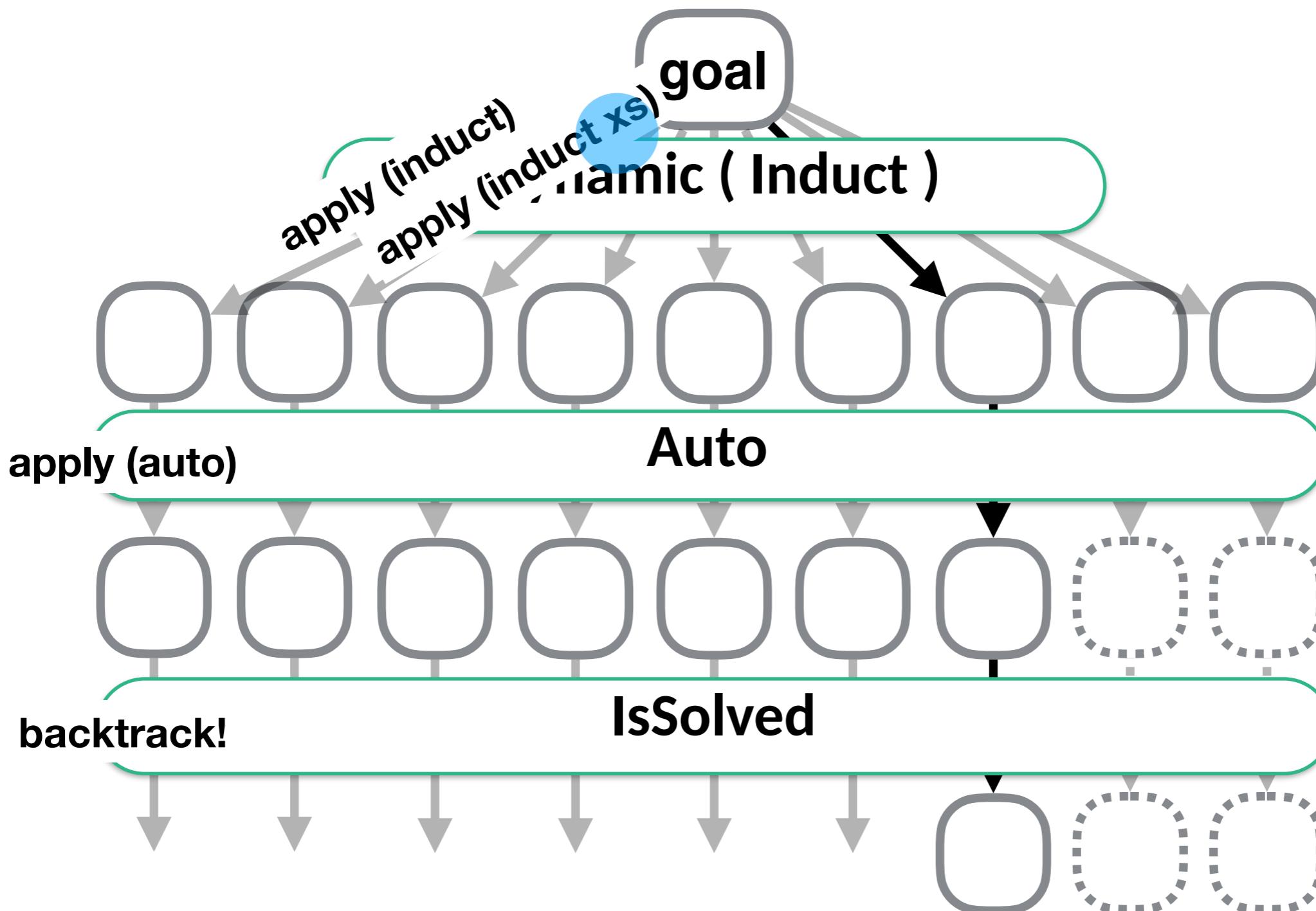
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



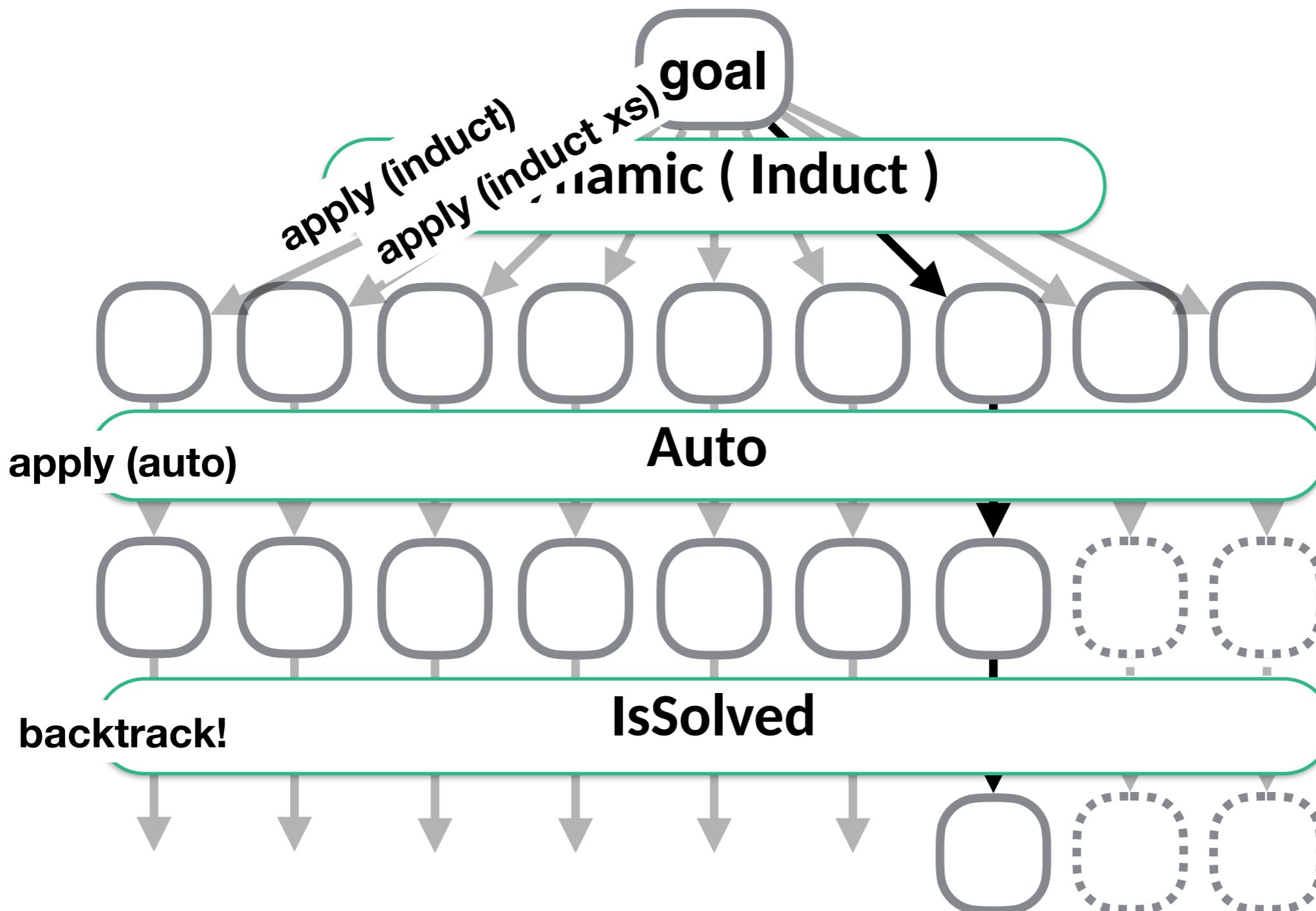
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



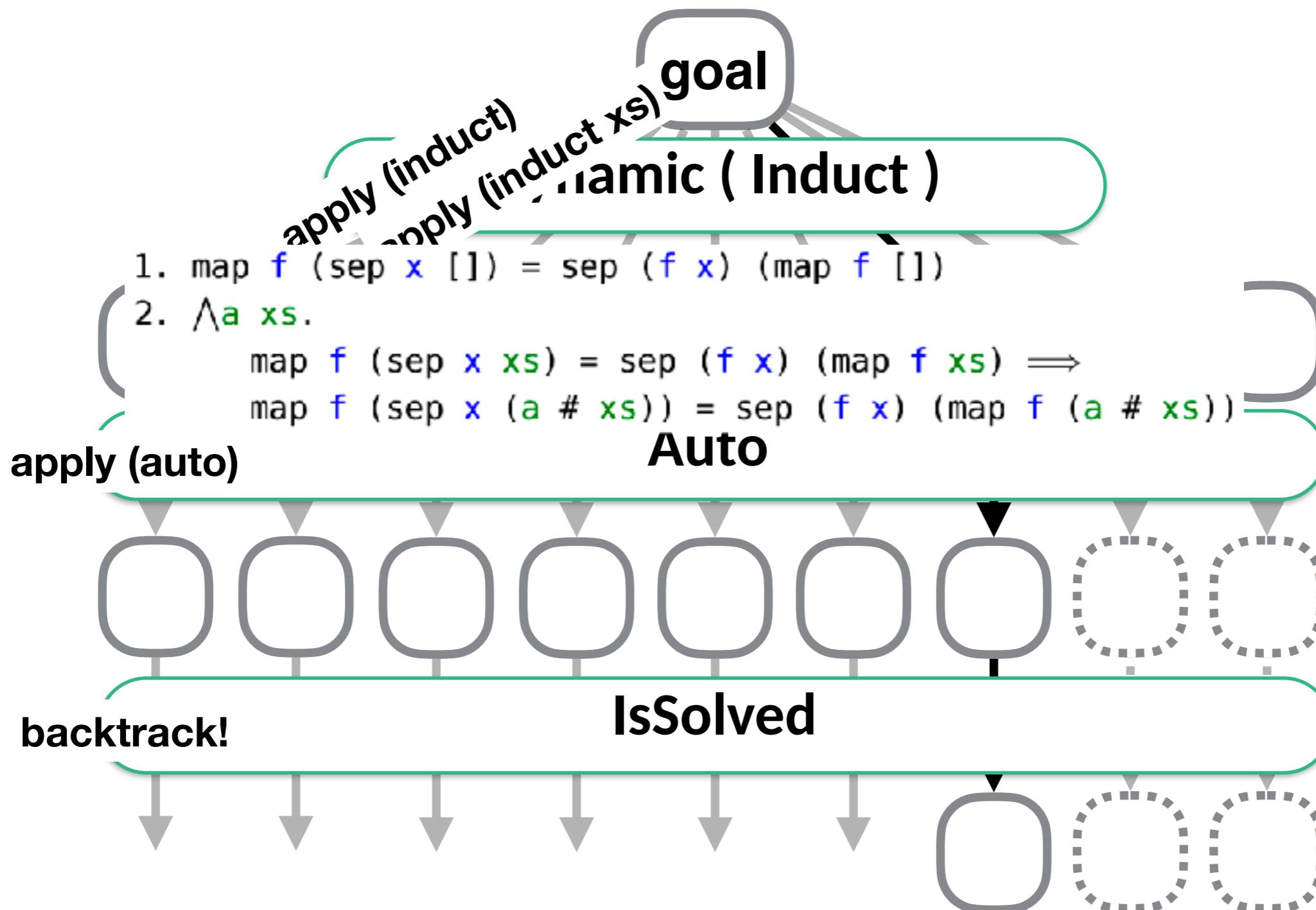
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



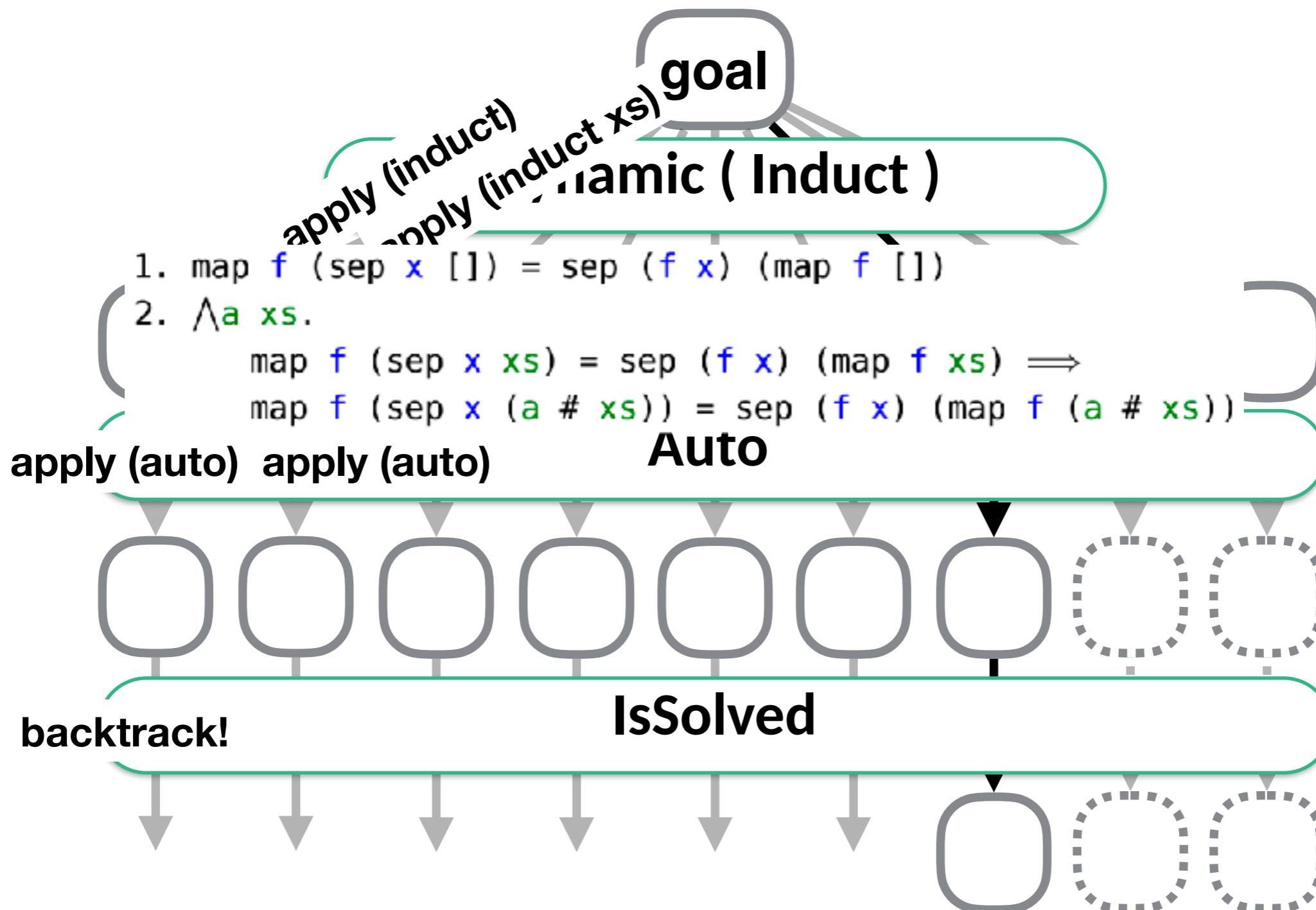
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



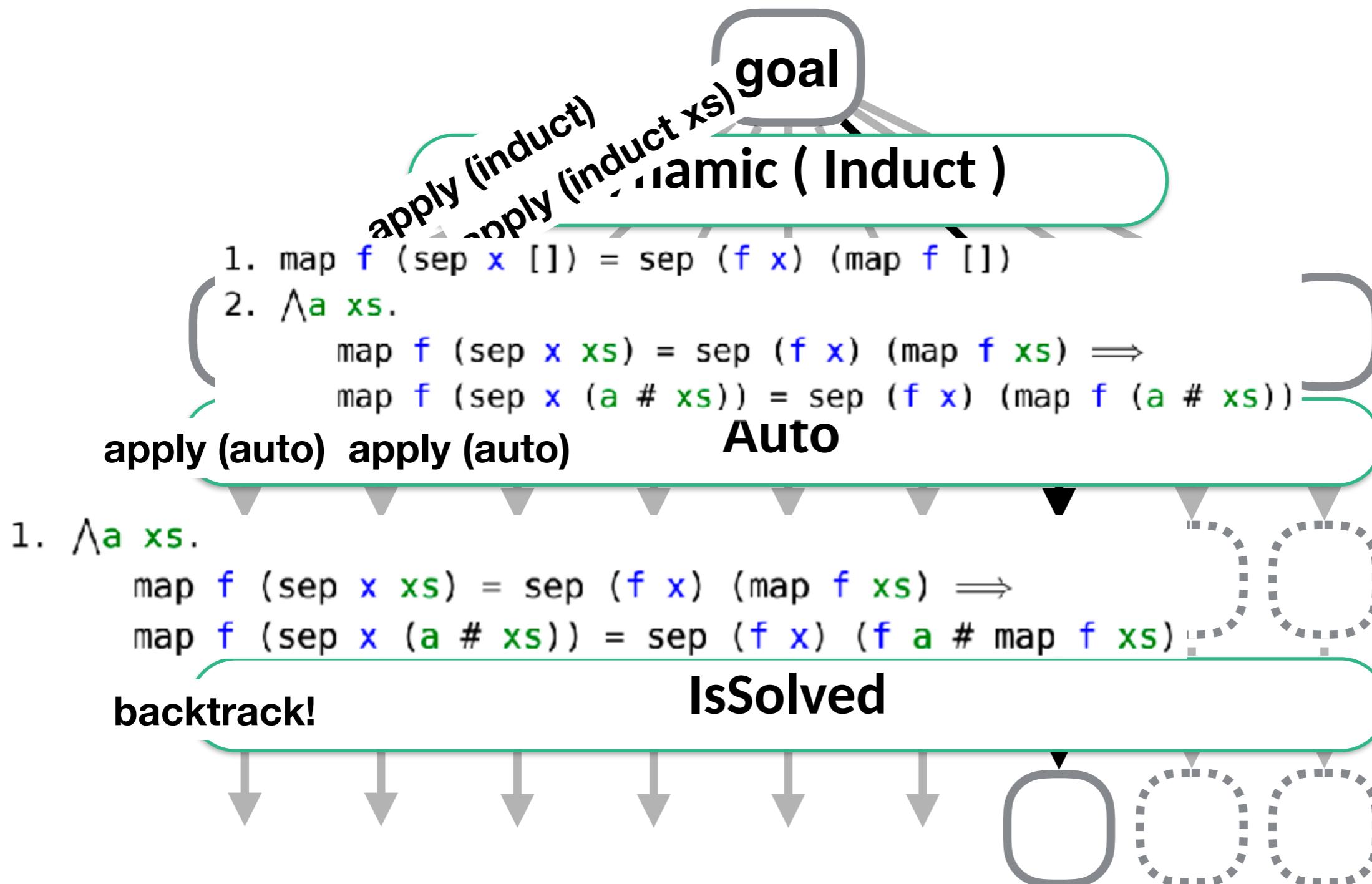
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



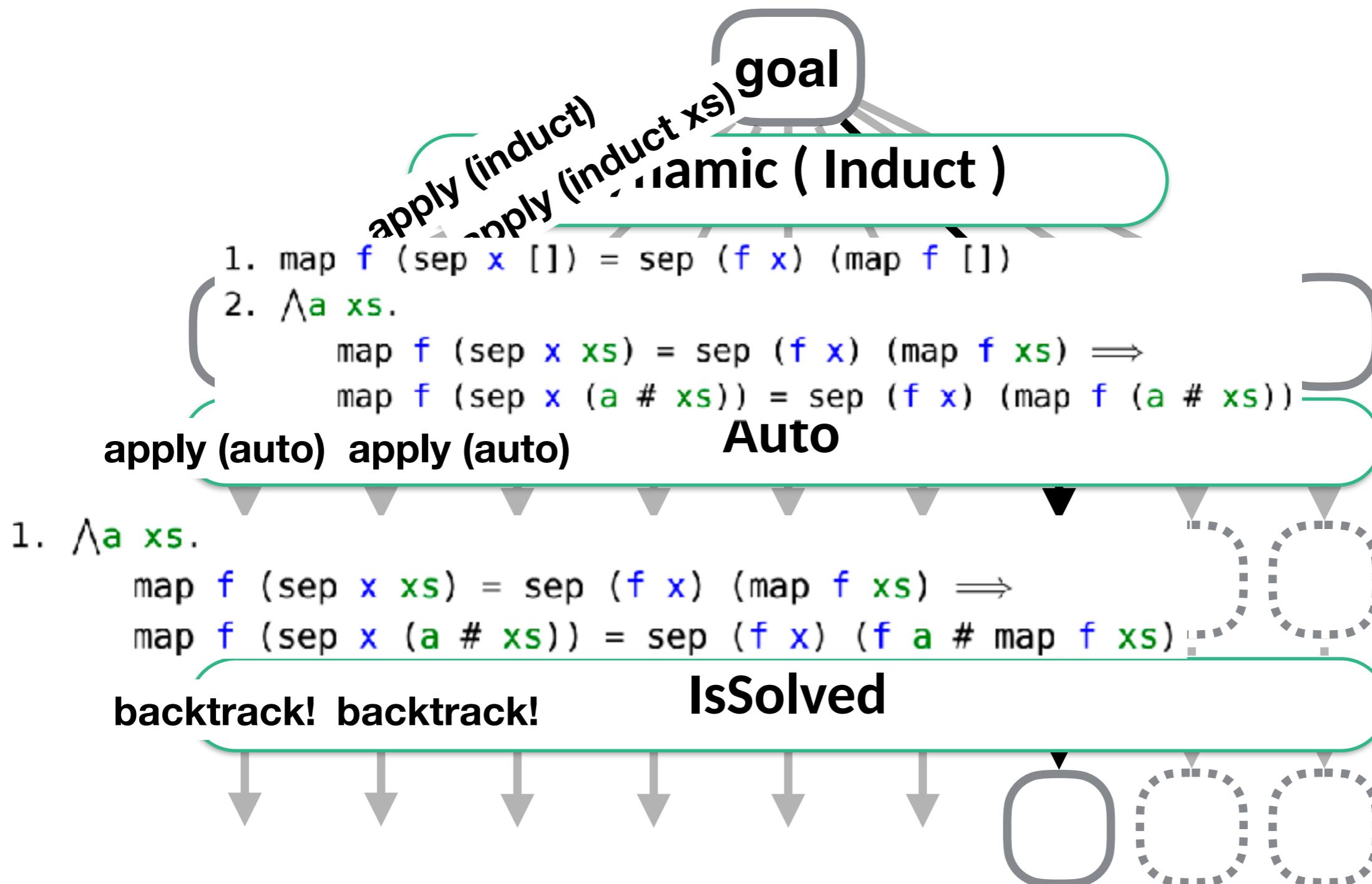
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



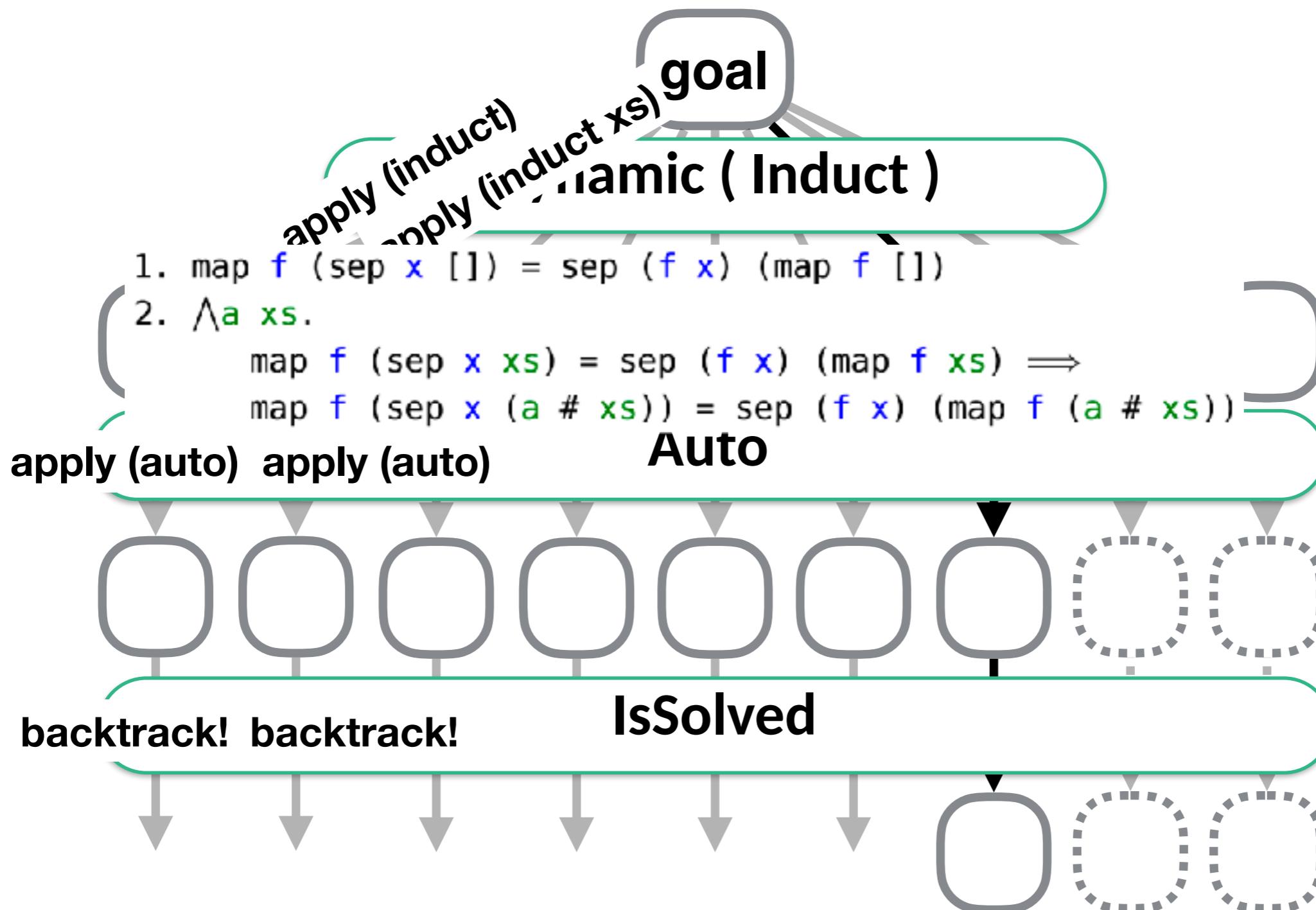
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



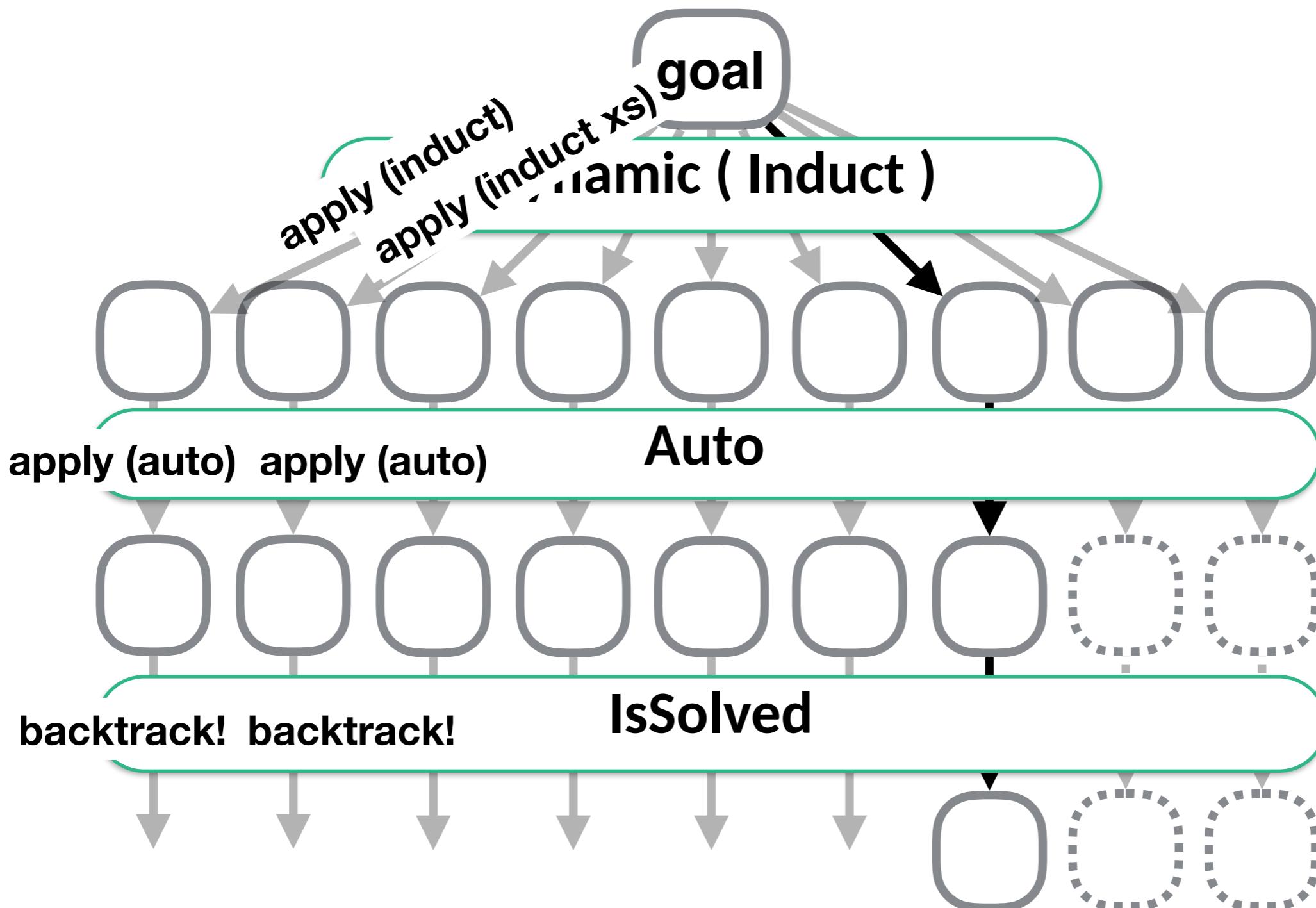
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



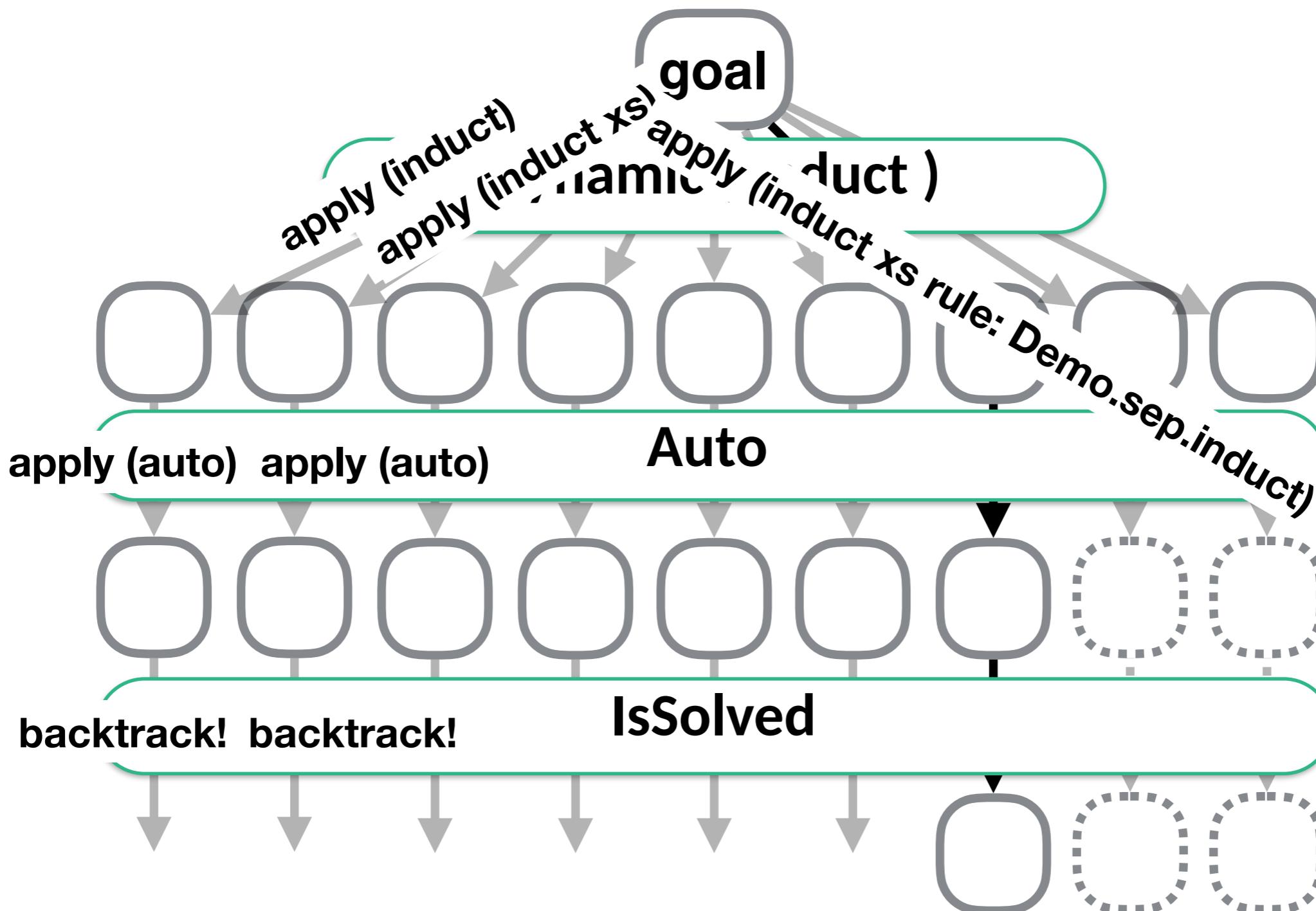
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



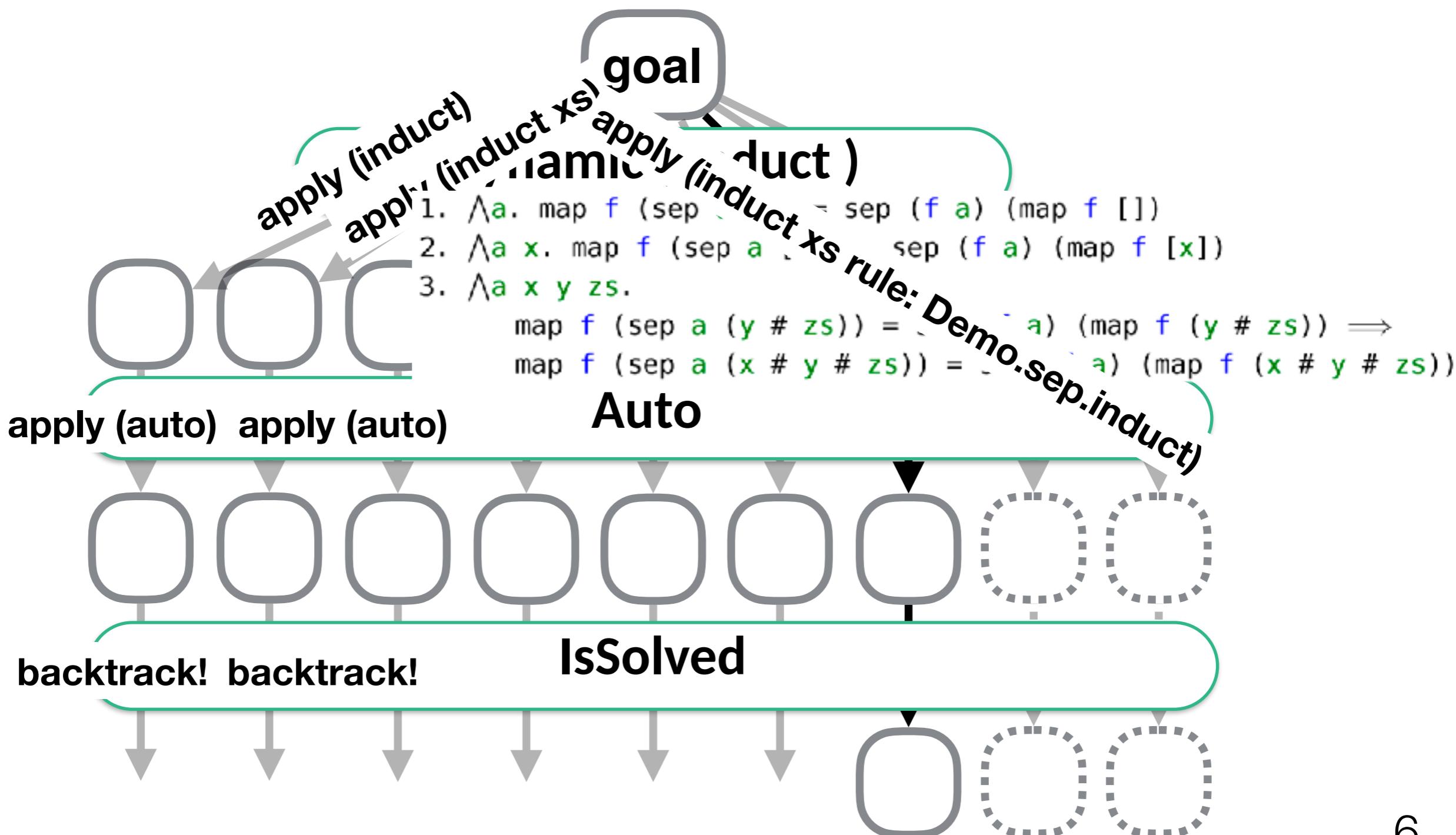
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



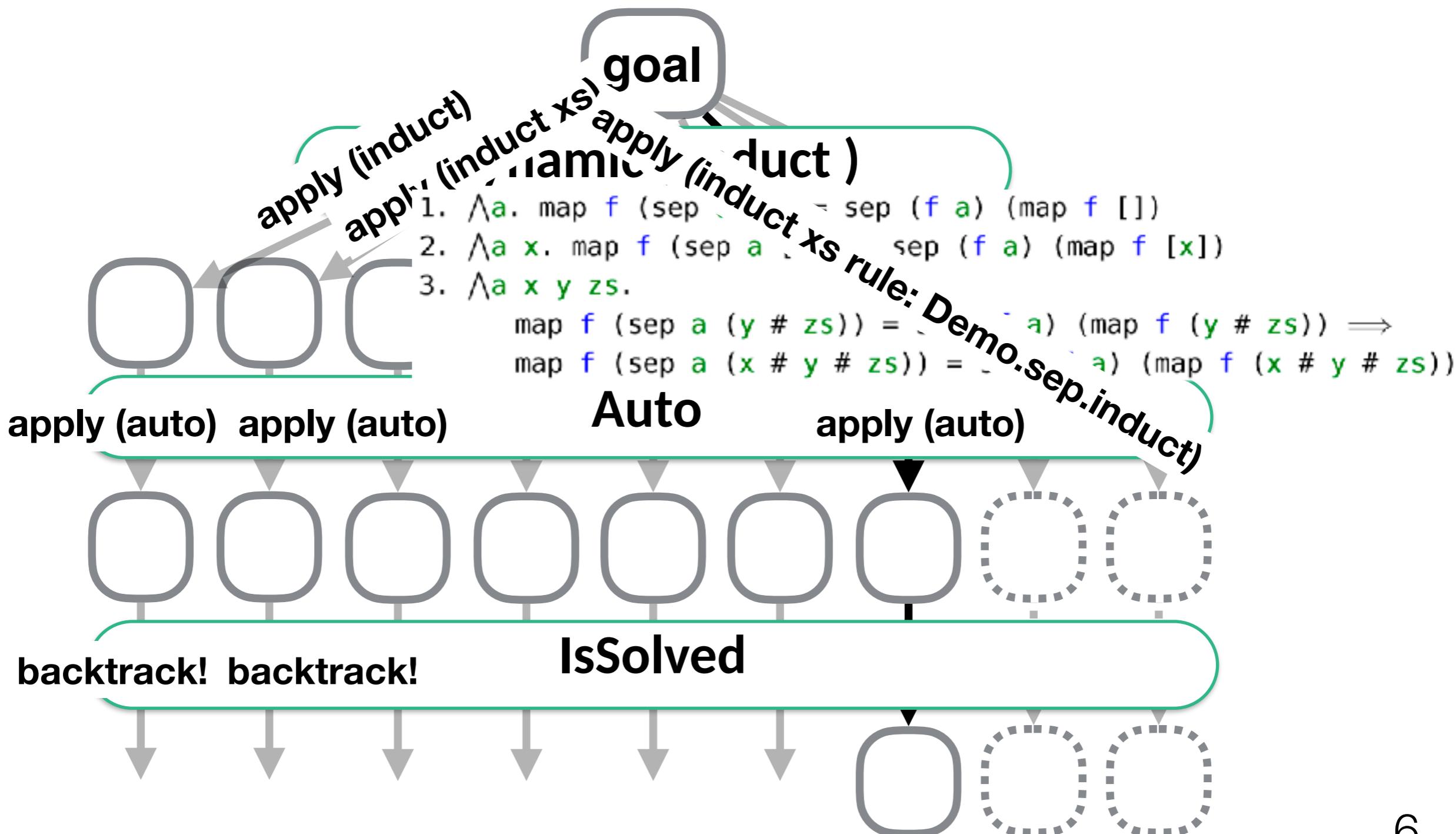
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



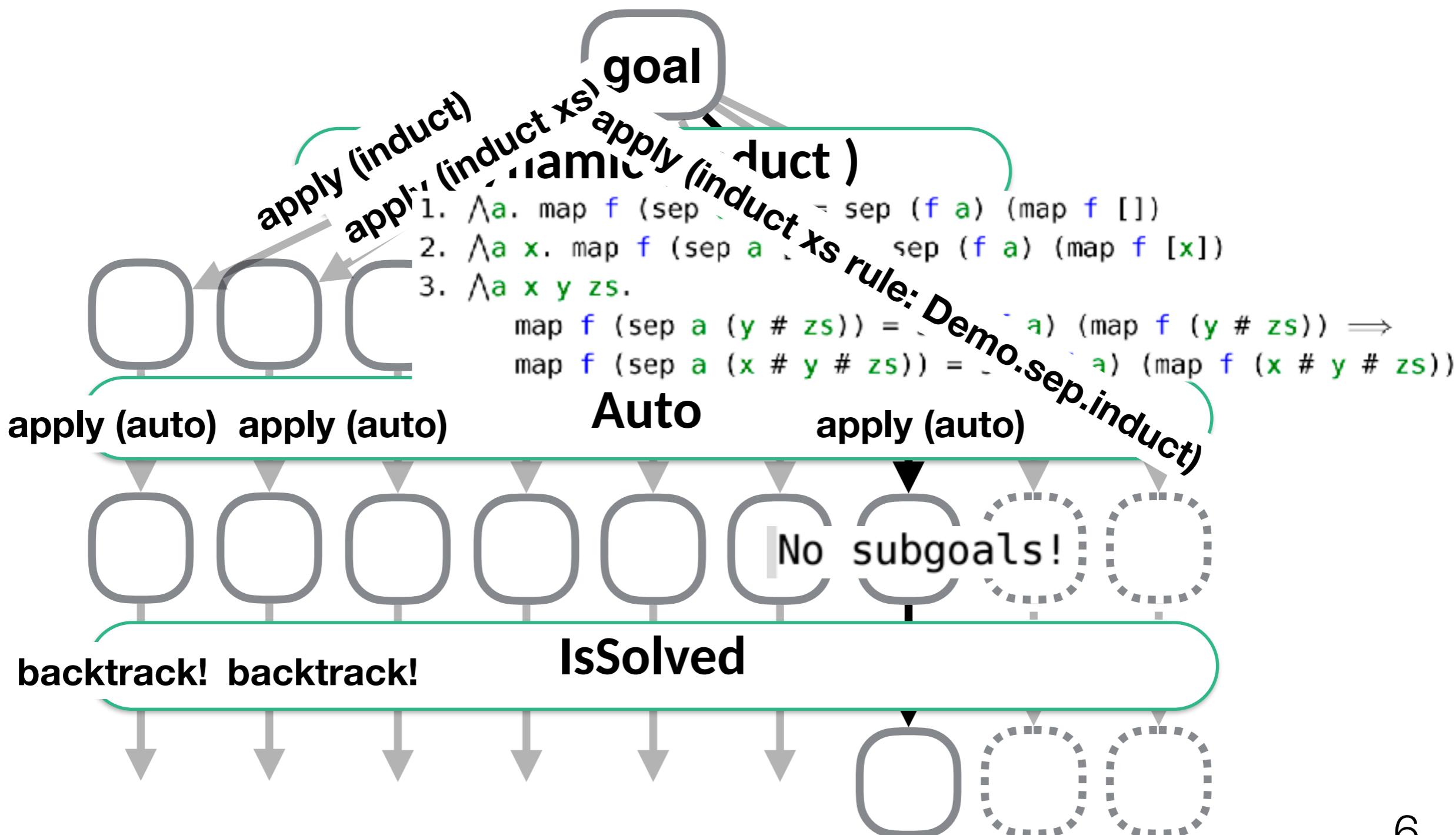
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



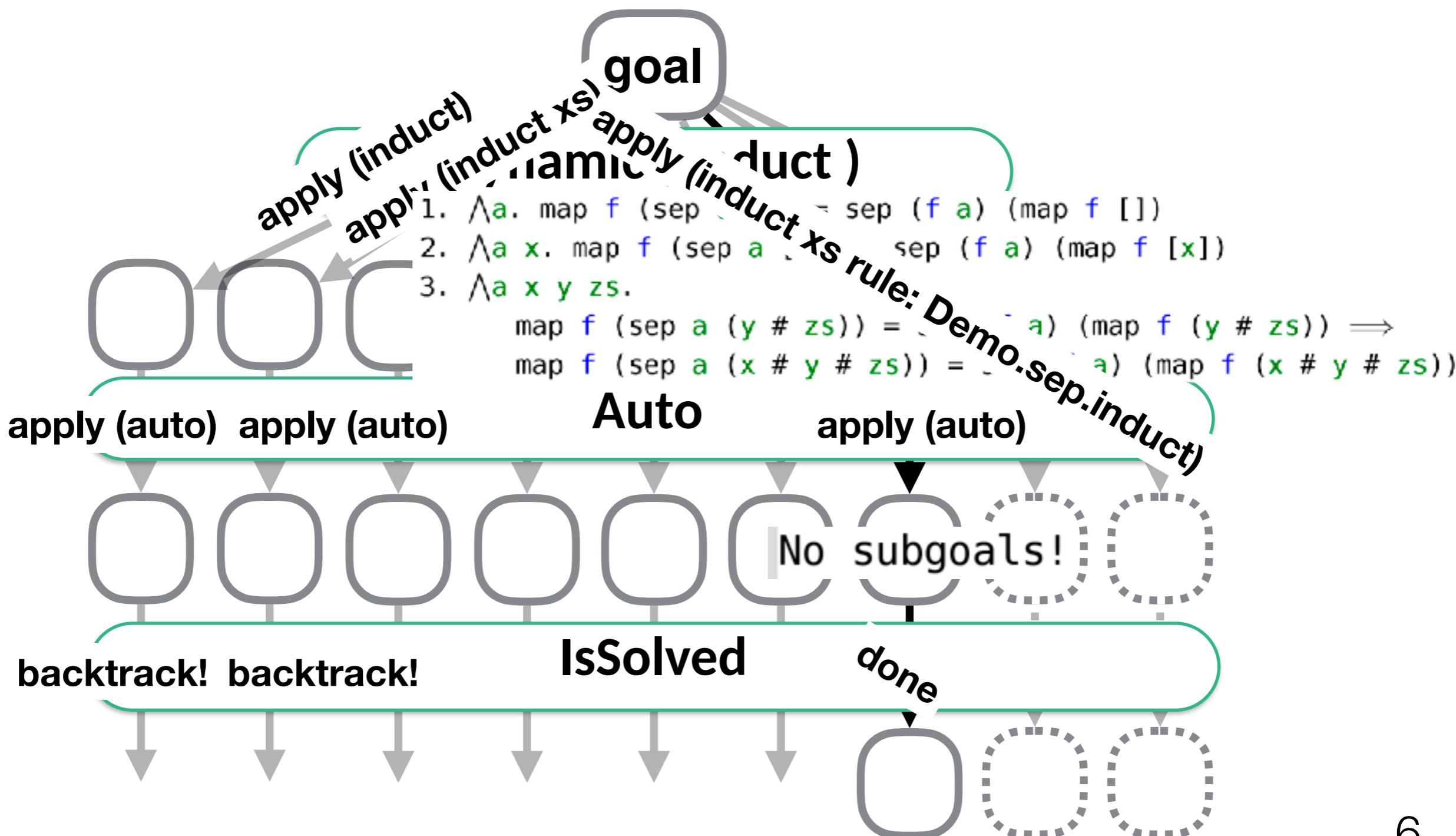
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



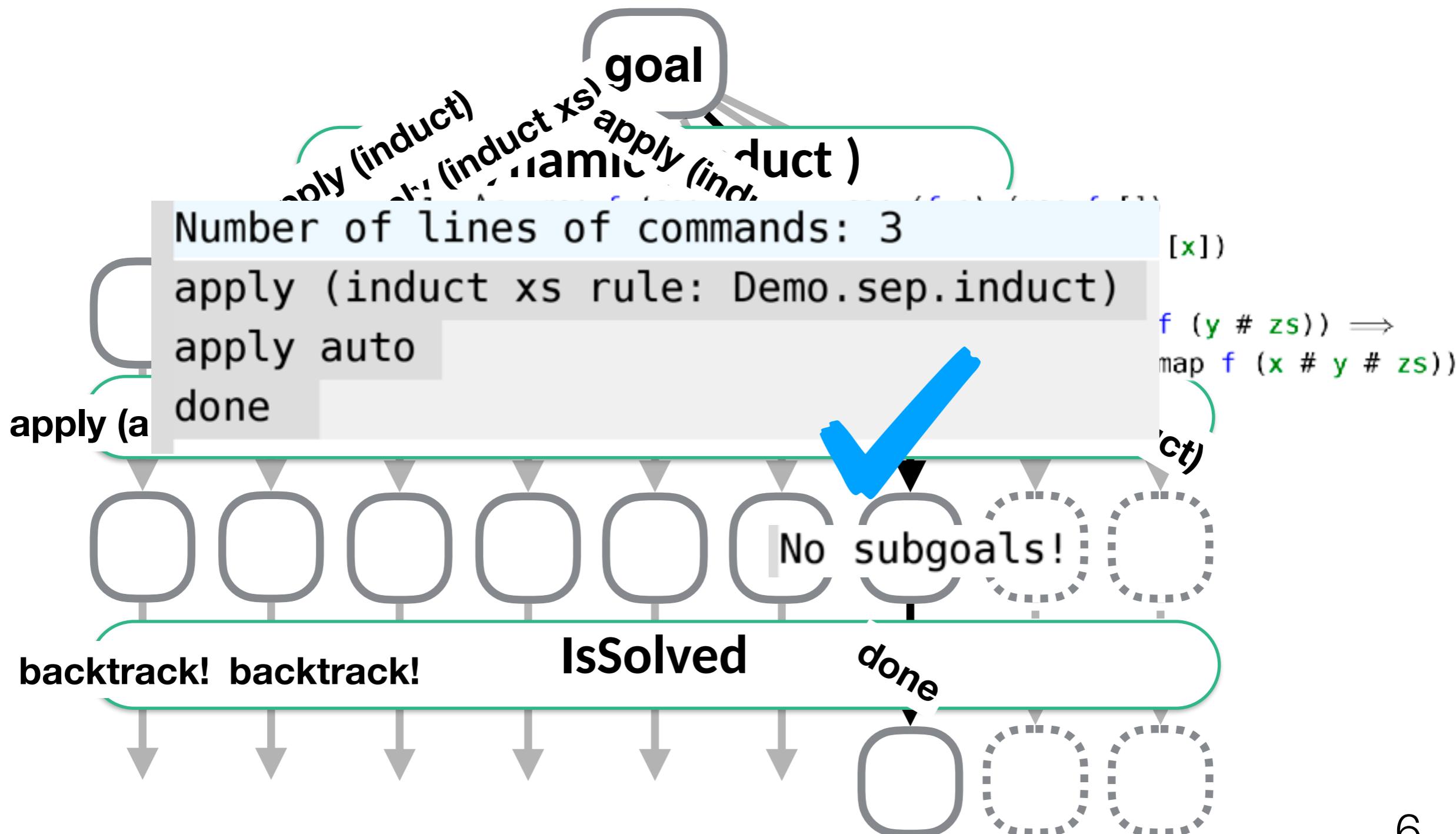
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



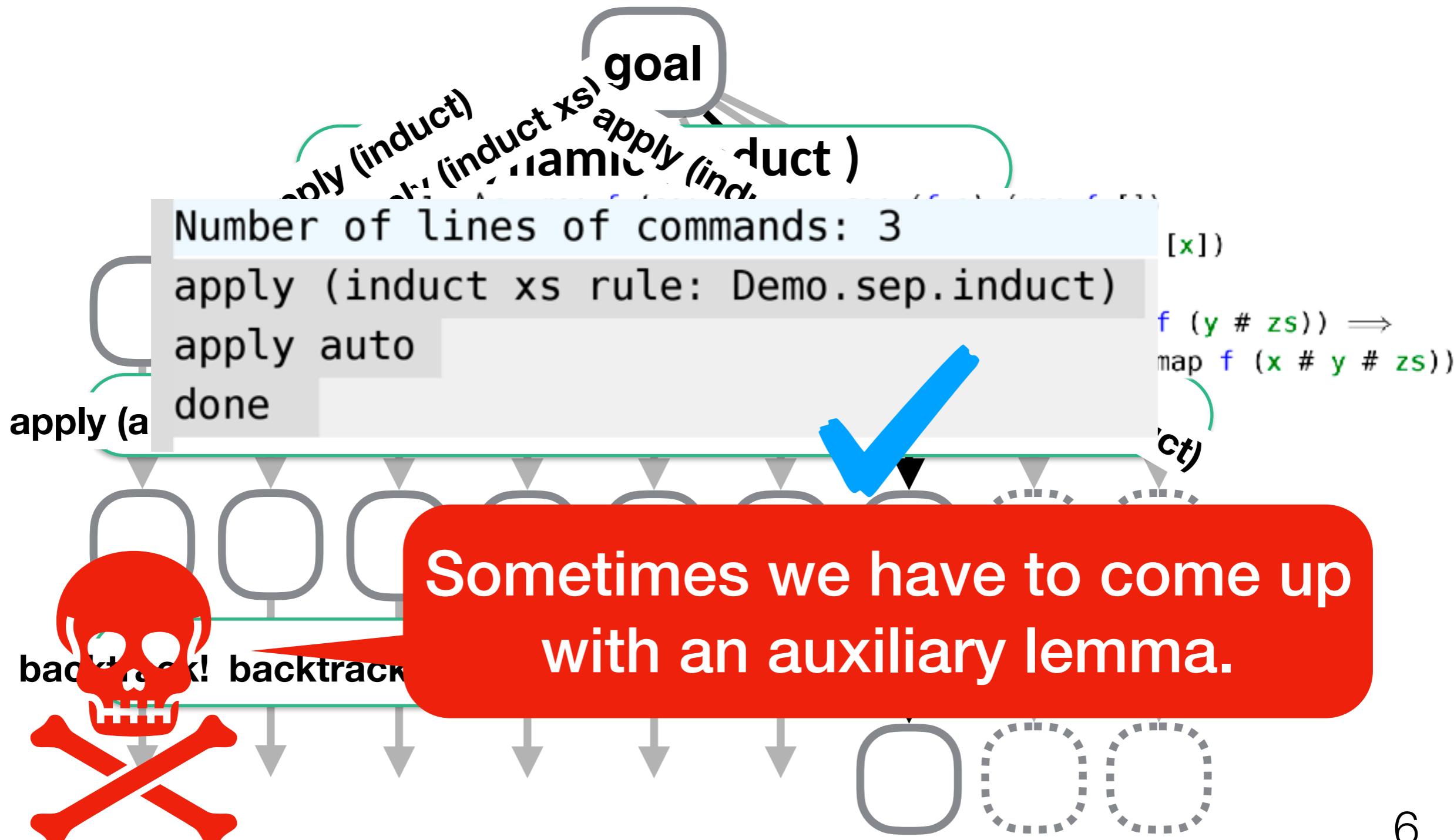
PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



PSL: Proof Strategy Language

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO2

http://github.com/data61/PSL/blob/master/slide/2020_NUS.pdf

The screenshot shows the Isabelle/Isar proof assistant interface. The top part displays a file named Example.thy containing ML code for defining functions rev and itrev, and setting a strategy for itrev. The bottom part shows a proof state with a lemma about itrev.

```
File Browser Documentation Sidekick State Theories
```

```
Example.thy (~-/Workplace/PSL/PGT/)

32 primrec rev:: "'a list ⇒ 'a list" where
33   "rev []      = []"
34 | "rev (x # xs) = rev xs @ [x]"
35 primrec itrev:: "'a list ⇒ 'a list" where
36   "itrev [] ys      = ys"
37 | "itrev (x#xs) ys = itrev xs (x#ys)"

38
39 strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved]
40
41
42
43 Lemma "itrev xs [] = rev xs"
44 find_proof DInd
45
```

Proof state: Proof state Auto update Update Search: 100% 0

```
proof (prove)
goal (1 subgoal):
  1. itrev xs [] = Example.rev xs
```

DEMO2

https://github.com/data61/PSL/blob/master/slide/2020_NUS.pdf

The screenshot shows the Isabelle/Isar proof assistant interface. The main window displays a file named "Example.thy". The code defines two primitive recursive functions for reversing lists:

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

Below these definitions, a strategy is specified:

```
strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved]
```

Then, a lemma is declared:

```
Lemma "itrev xs [] = rev xs"
```

The command `find_proof DInd` is being typed at the bottom of the editor window.

empty sequence. no proof found.

DEMO2

https://github.com/data61/PSL/blob/master/slide/2020_NUS.pdf

The screenshot shows the Isabelle/PGT interface with a blue box highlighting the title "DEMO2". The main window displays a file named "Example.thy" containing ML code. The code defines two primitive recursive functions for reversing lists: "rev" and "itrev". It also defines three strategies: "DInd", "CDInd", and "DInd_Or_CDInd". A lemma is stated: "itrev xs [] = rev xs". The "find_proof" command is used to find proofs for this lemma under the strategies DInd and DInd_Or_CDInd. The interface includes a toolbar at the top, a file browser on the left, and a sidebar on the right.

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

strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved]
strategy CDInd = Thens [Conjecture, Fastforce, Quickcheck, DInd]
strategy DInd_Or_CDInd = Ors [DInd, CDInd]

Lemma "itrev xs [] = rev xs"
find_proof DInd
find_proof DInd_Or_CDInd
```

PGT creates 131 conjectures.

Testing conjecture with Quickcheck-exhaustive...

Testing conjecture with Quickcheck-exhaustive...

Testing conjecture with Quickcheck-exhaustive...

Testing conjecture with Quickcheck-exhaustive...

DEMO2

http://github.com/data61/PSL/blob/master/slide/2020_NUS.pdf

The screenshot shows the Isabelle/PGT interface with the following code:

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

strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved]
strategy CDInd = Thens [Conjecture, Fastforce, Quickcheck, DInd]
strategy DInd_Or_CDInd = Ors [DInd, CDInd]

Lemma "itrev xs [] = rev xs"
  find_proof DInd
  find_proof DInd_Or_CDInd
```

The code defines two recursive functions for list reversal: `rev` and `itrev`. It also defines three proof strategies: `DInd`, `CDInd`, and `DInd_Or_CDInd`. A lemma is stated: `"itrev xs [] = rev xs"`, followed by two proof attempts: `find_proof DInd` and `find_proof DInd_Or_CDInd`.

PGT creates 131 conjectures.

Testing conjecture with Quickcheck-exhaustive...

Testing conjecture with Quickcheck-exhaustive...

Testing conjecture with Quickcheck-exhaustive...

Testing conjecture with Quickcheck-exhaustive...

DEMO2

https://github.com/data61/PSL/blob/master/slide/2020_NUS.pdf

The screenshot shows the Isabelle proof assistant interface. The top part displays a file named "Example.thy" containing OCaml-style code for defining list reversal functions and specifying proof strategies. The bottom part shows the proof state with tactics being applied to prove a lemma.

```
File Browser Documentation Example.thy (~/Workplace/PSL/PGT/)

32 primrec rev:: "'a list ⇒ 'a list" where
33   "rev []      = []"
34 | "rev (x # xs) = rev xs @ [x]"
35 primrec itrev:: "'a list ⇒ 'a list" where
36   "itrev [] ys      = ys"
37 | "itrev (x#xs) ys = itrev xs (x#ys)"

38 strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved]
39 strategy CDInd = Thens [Conjecture, Fastforce, Quickcheck, DInd]
40 strategy DInd_Or_CDInd = Ors [DInd, CDInd]

42 Lemma "itrev xs [] = rev xs"
43   find_proof DInd
44   find_proof DInd_Or_CDInd
```

```
apply (subgoal_tac "Nil. itrev xs Nil = Example.rev xs @ Nil")
apply fastforce
apply (induct xs)
apply auto
done
```

DEMO2

http://github.com/data61/PSL/blob/master/slide/2020_NUS.pdf

The screenshot shows the Isabelle proof assistant interface. The main window displays a file named "Example.thy" containing ML code. The code defines two primitive recursive functions: `rev` and `itrev`. The `rev` function takes a list and returns its reverse. The `itrev` function takes a list and returns its reverse. The code also defines three strategies: `DInd`, `CDInd`, and `DInd_Or_CDInd`. A lemma is stated: `"itrev xs [] = rev xs"`. The proof search command `find_proof DInd` is shown, followed by `find_proof DInd_Or_CDInd`, which is highlighted with a red rectangle.

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

strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved]
strategy CDInd = Thens [Conjecture, Fastforce, Quickcheck, DInd]
strategy DInd_Or_CDInd = Ors [DInd, CDInd]

Lemma "itrev xs [] = rev xs"
  find_proof DInd
  find_proof DInd_Or_CDInd
```

The screenshot shows the Isabelle proof assistant interface with a proof trace. The trace consists of several `apply` commands:

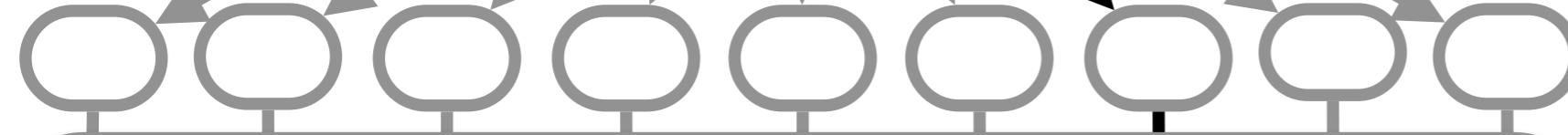
```
apply (subgoal_tac "Nil. itrev xs Nil = Example.rev xs @ Nil")
apply fastforce
apply (induct xs)
apply auto
done
```

What happened?

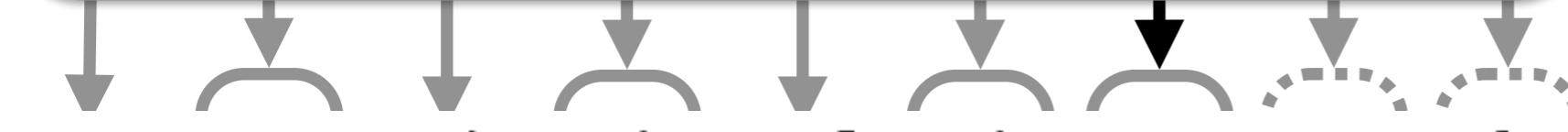
```
goal (1 subgoal):  
1. itrev xs [] = rev xs
```

goal

Conjecture



Fastforce

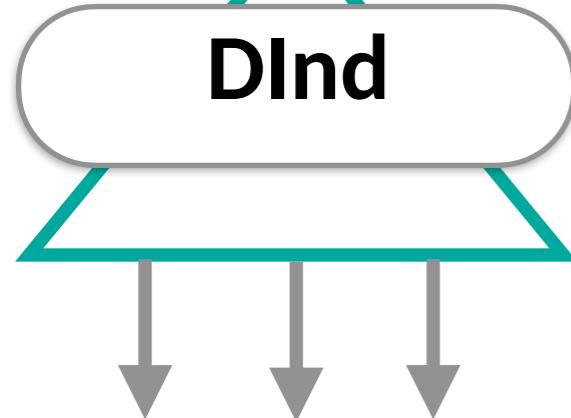


```
strategy CDInd = Thens [Conjecture, Fastforce, Quickcheck, DInd]
```

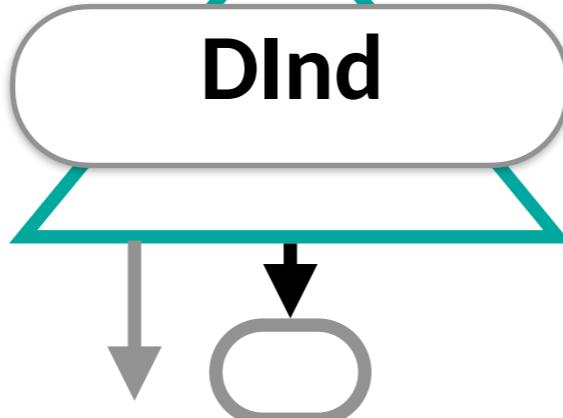
Quickcheck



DInd



DInd



```
goal (1 subgoal):  
1. itrev xs [] = rev xs
```

goal

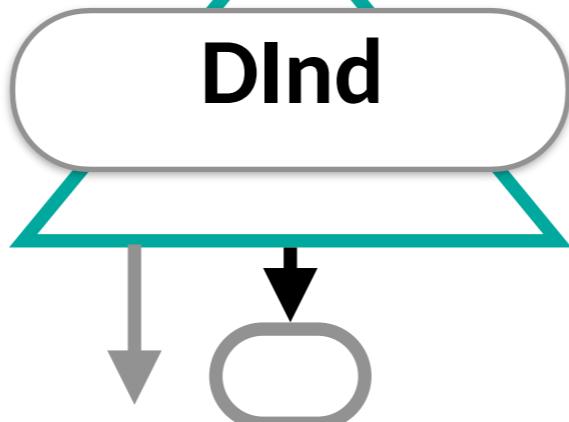
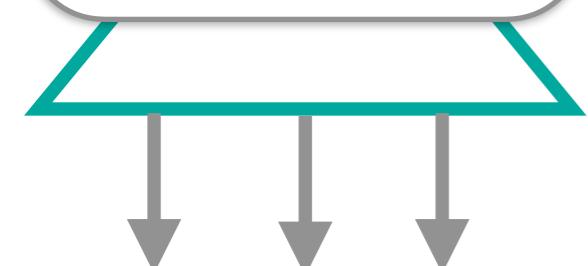
Conjecture

Fastforce

Quickcheck

DInd

DInd



```
goal (1 subgoal):  
1. itrev xs [] = rev xs
```

goal

apply (subgoal_tac

" $\wedge \text{Nil}.$ itrev xs Nil = rev xs @ Nil")

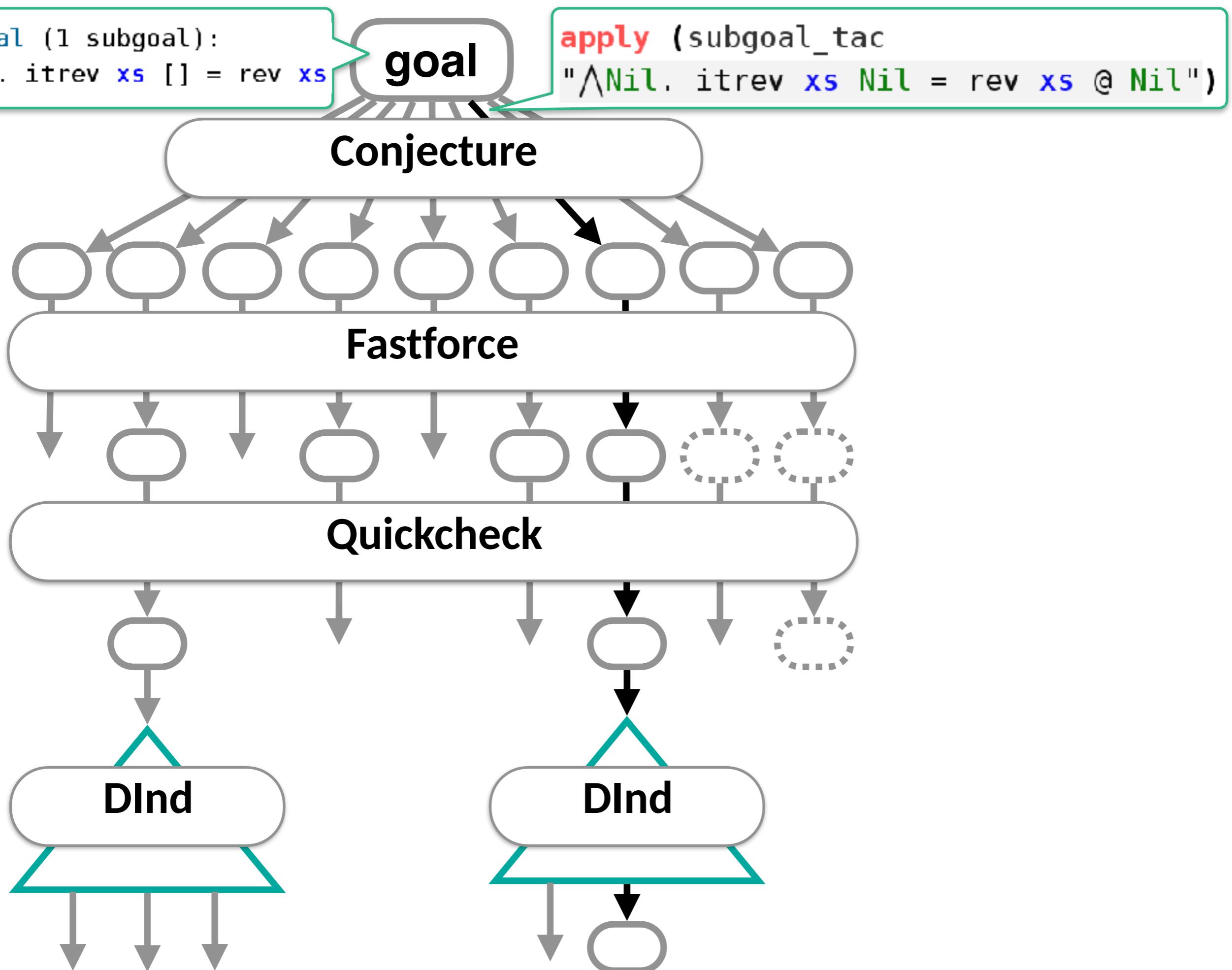
Conjecture

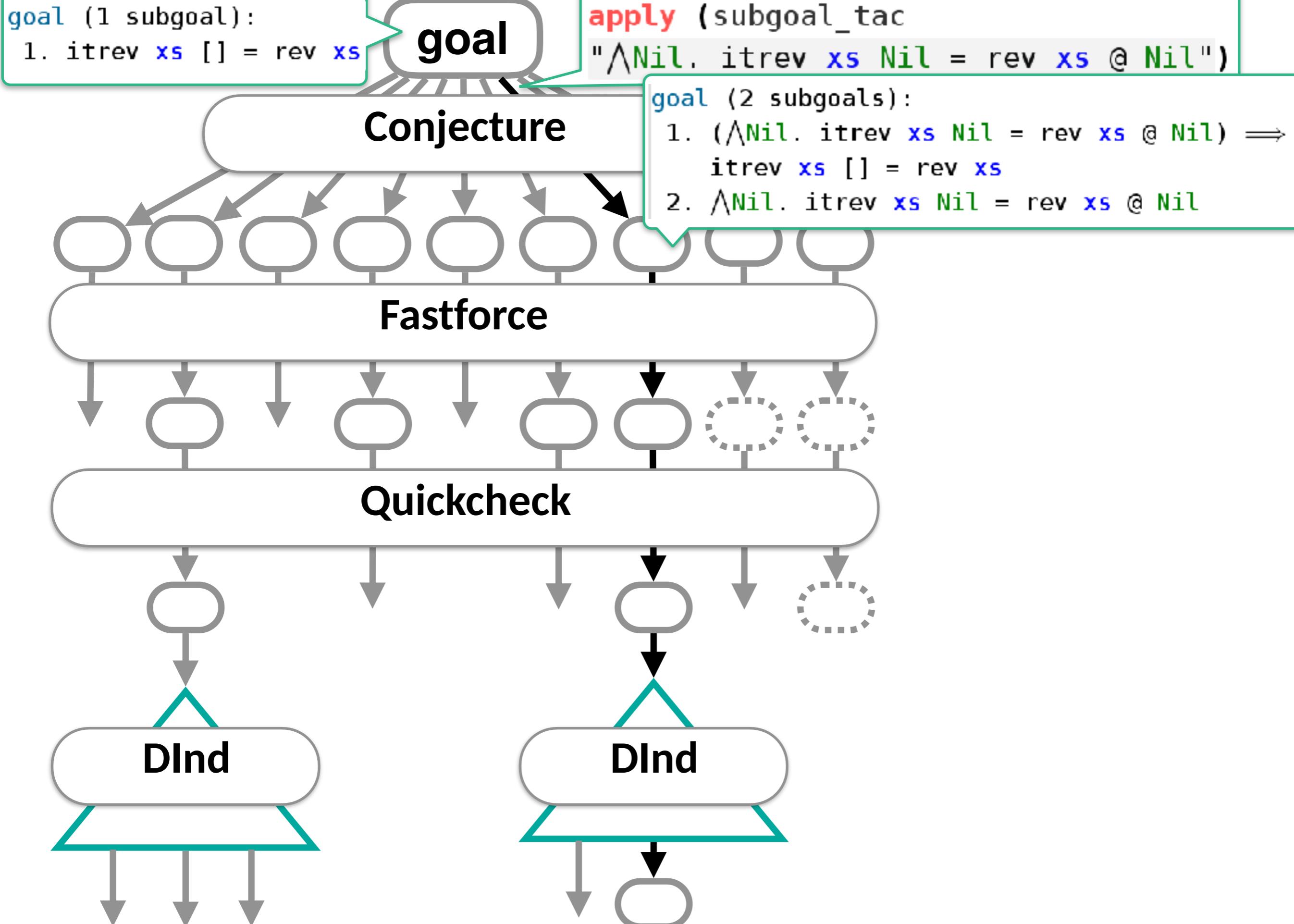
Fastforce

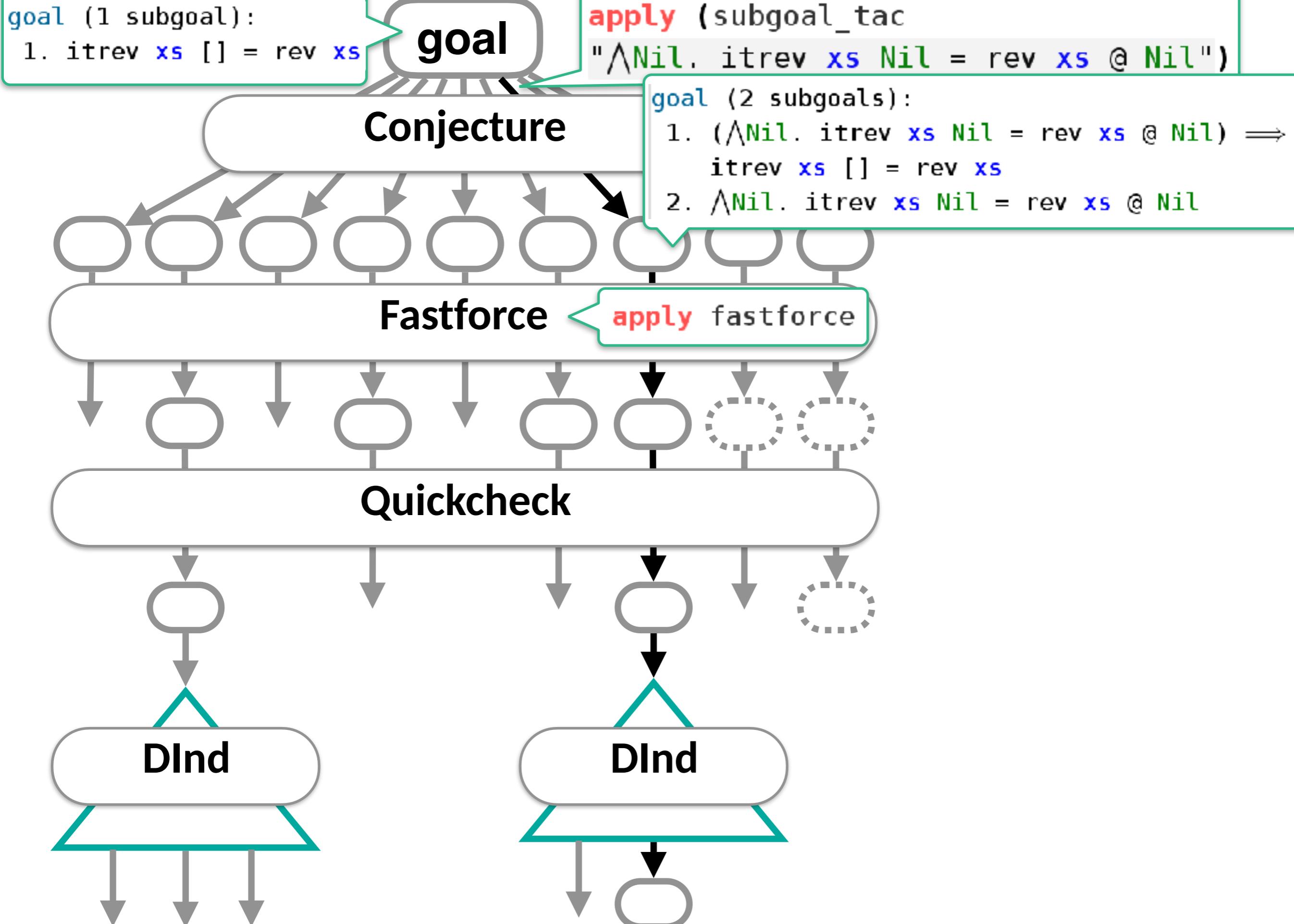
Quickcheck

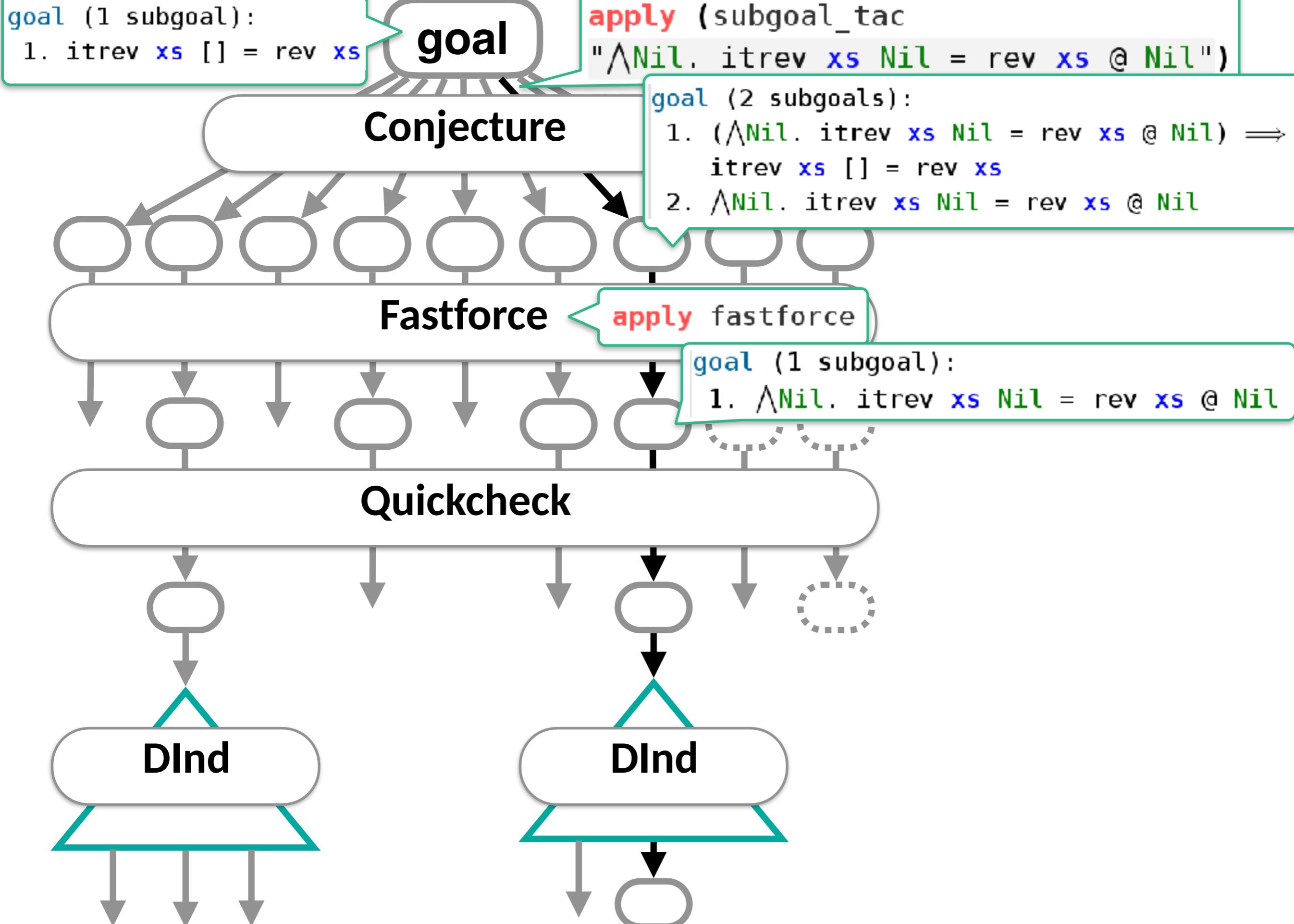
DInd

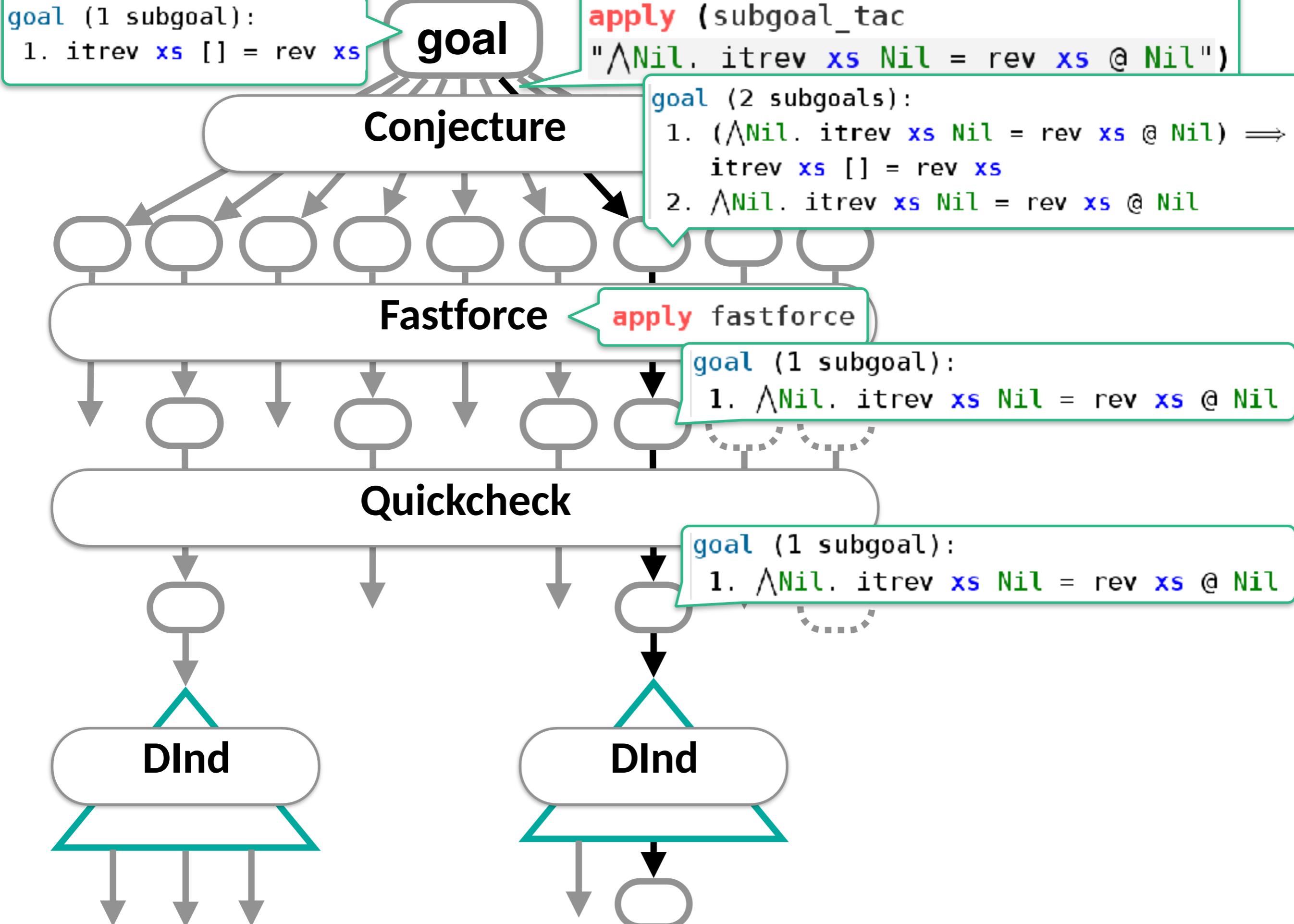
DInd

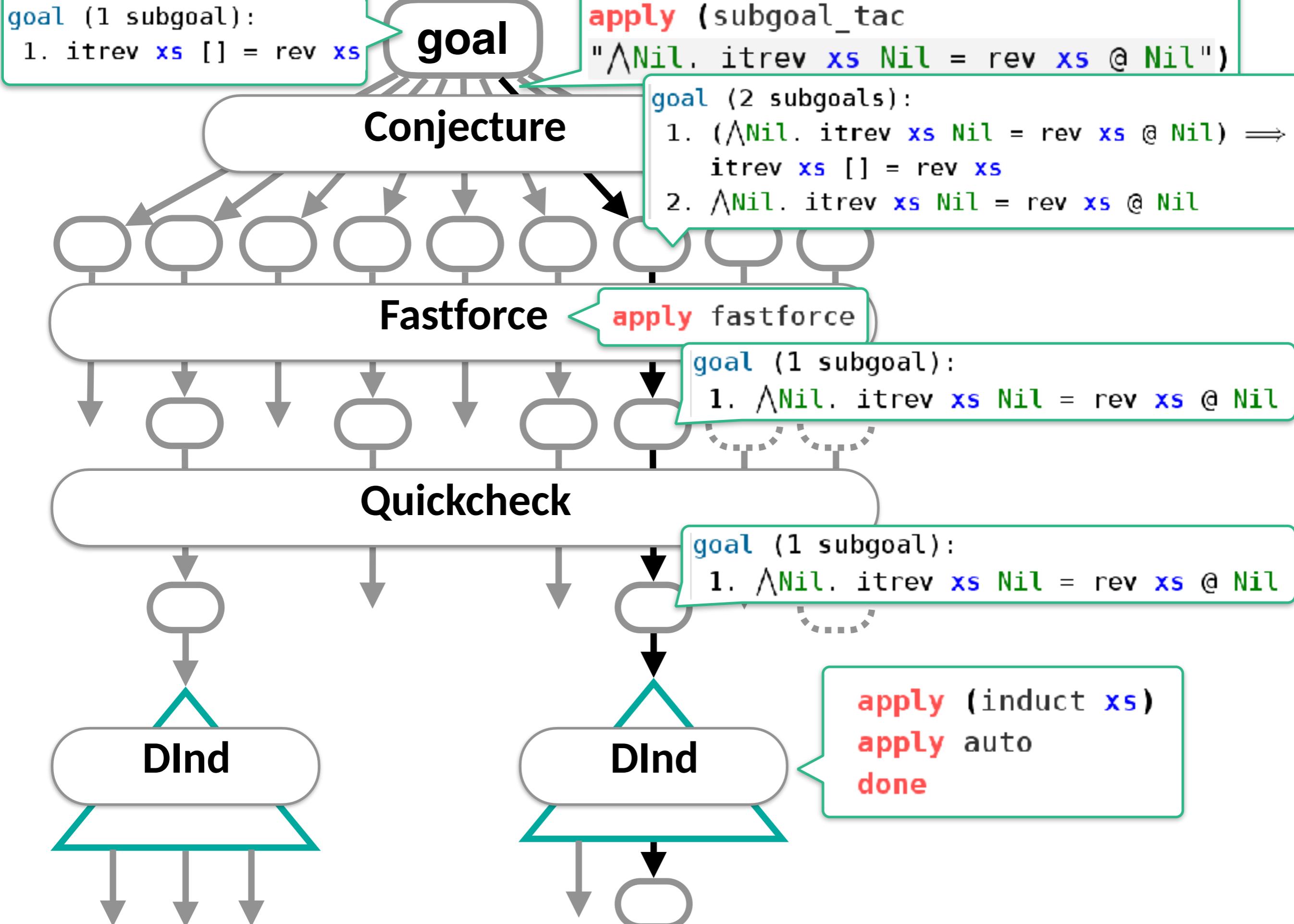


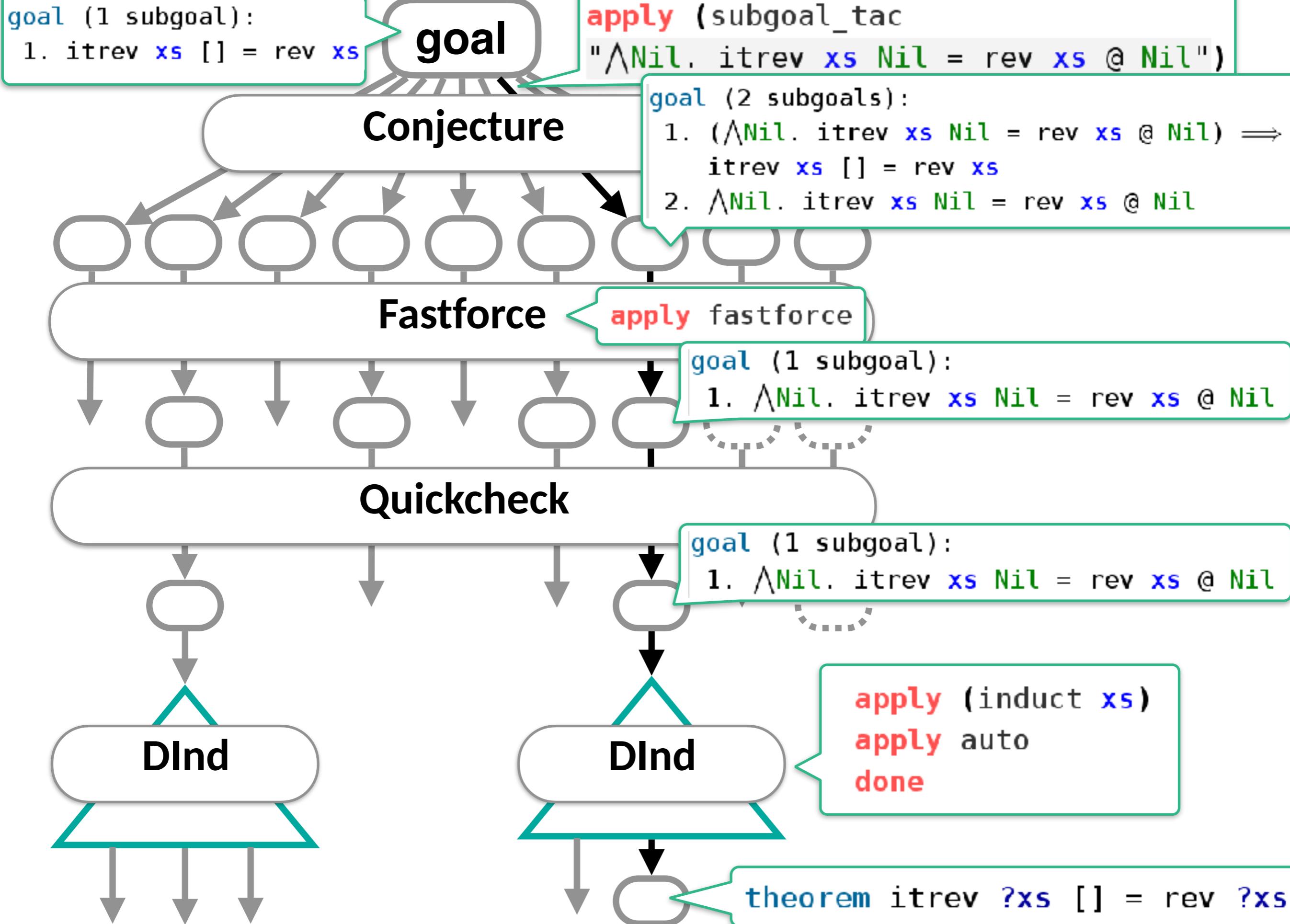


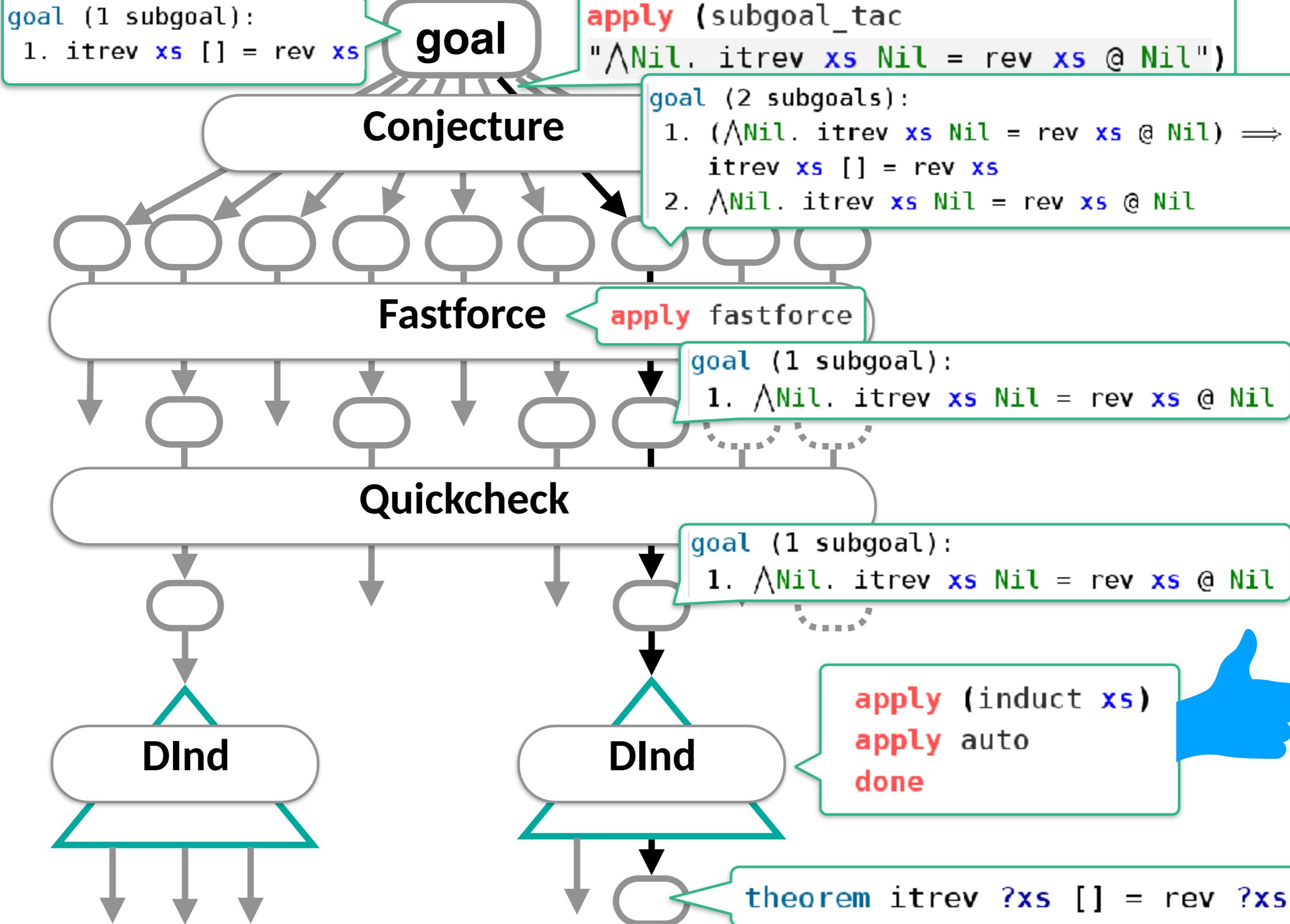












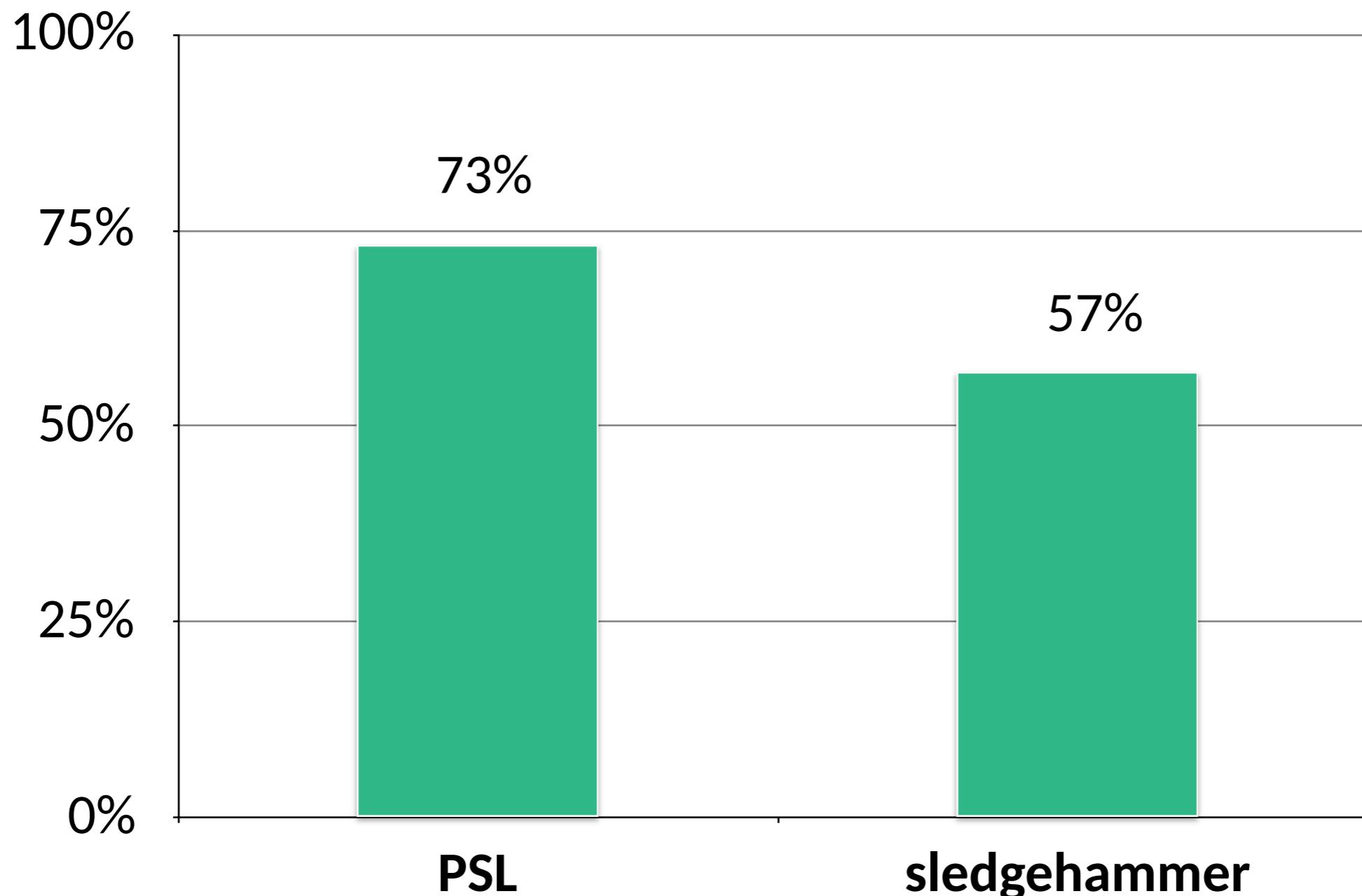
```
goal (1 subgoal):  
1. itrev xs [] = rev xs
```

goal

strategy CDInd = Thens [Conjecture, Fastforce, Quickcheck, DInd]

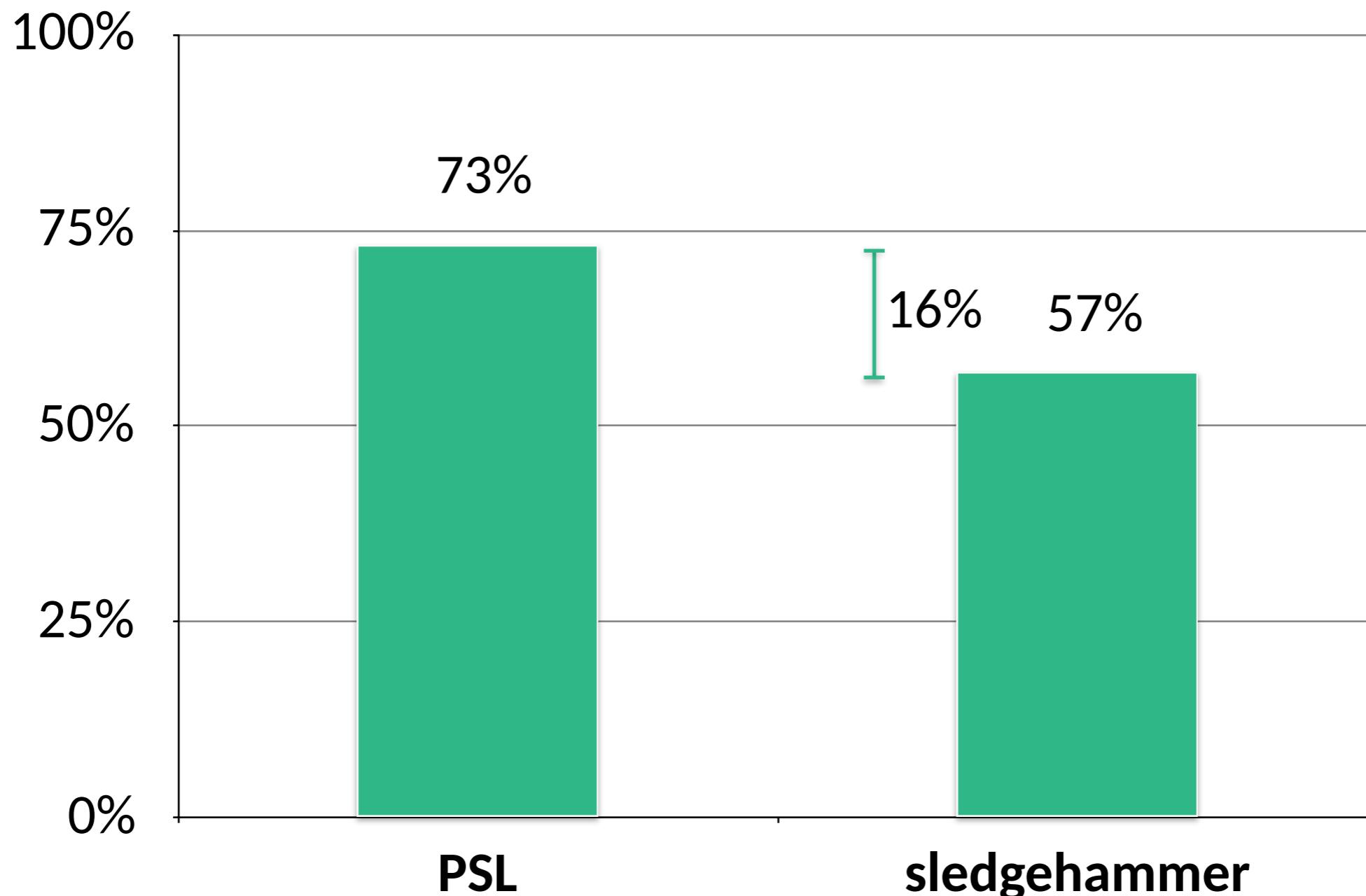
PSL vs sledgehammer

The percentage of automatically proved obligations out of 1526 proof obligations
(timeout = 300s)



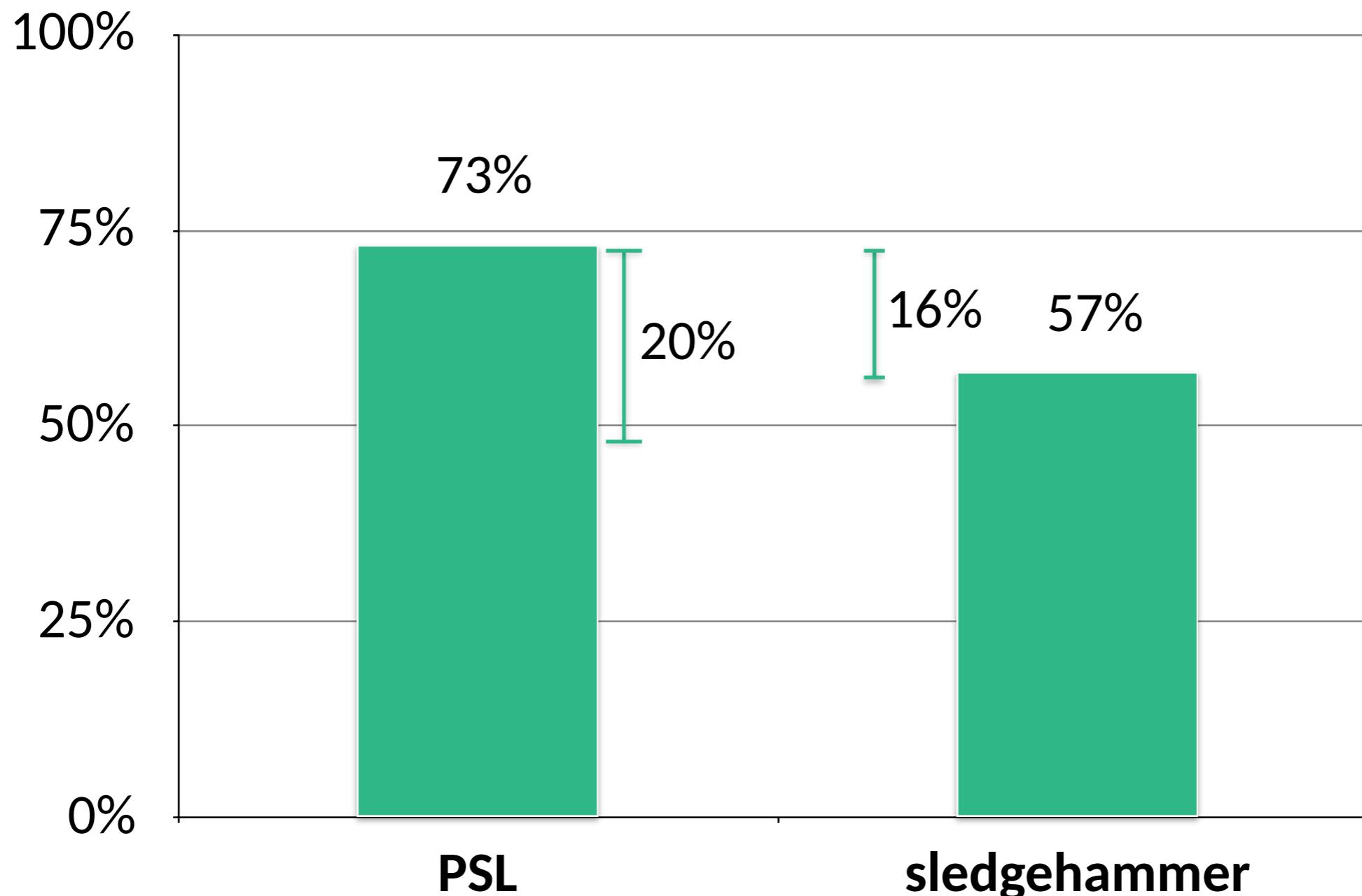
PSL vs sledgehammer

The percentage of automatically proved obligations out of 1526 proof obligations
(timeout = 300s)

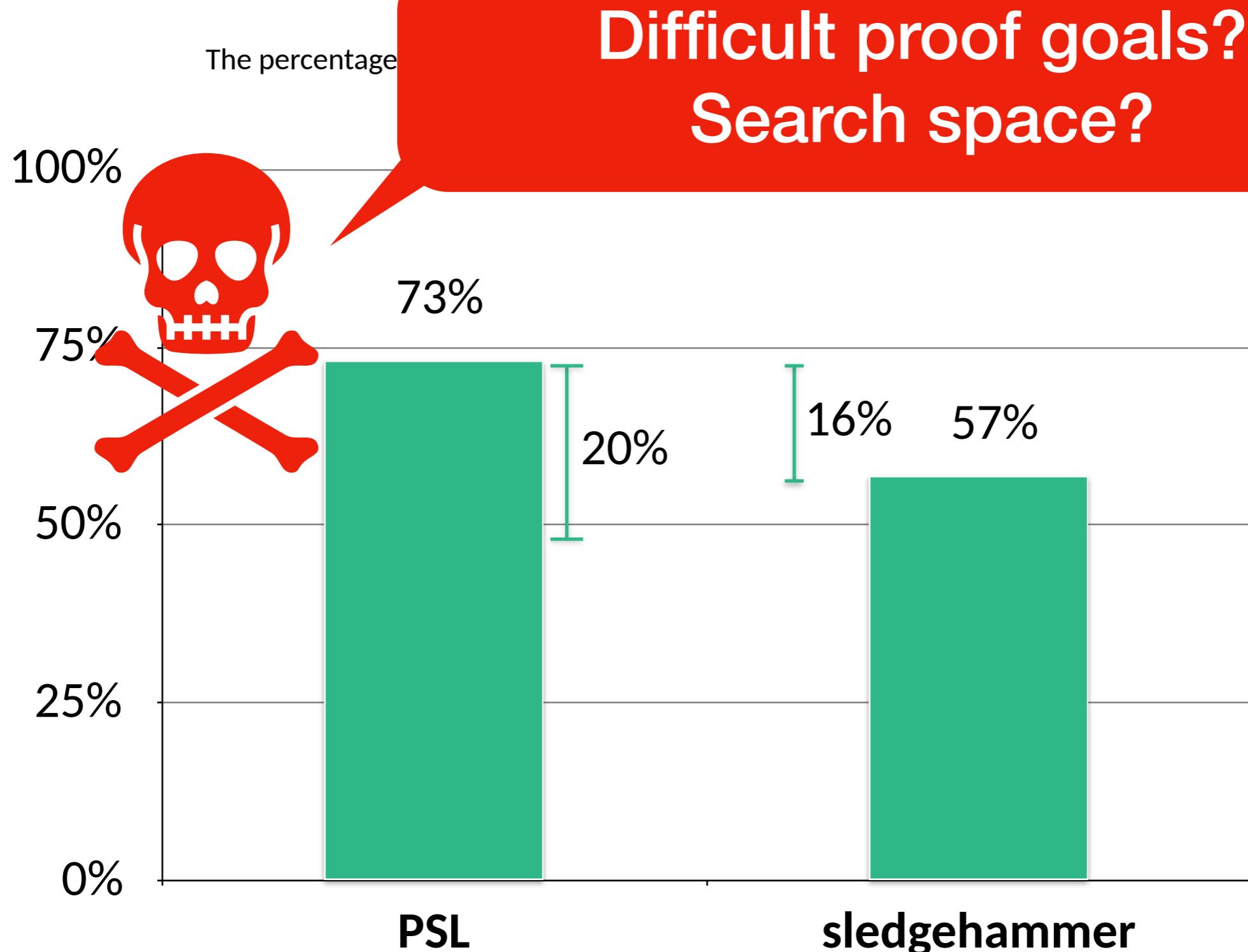


PSL vs sledgehammer

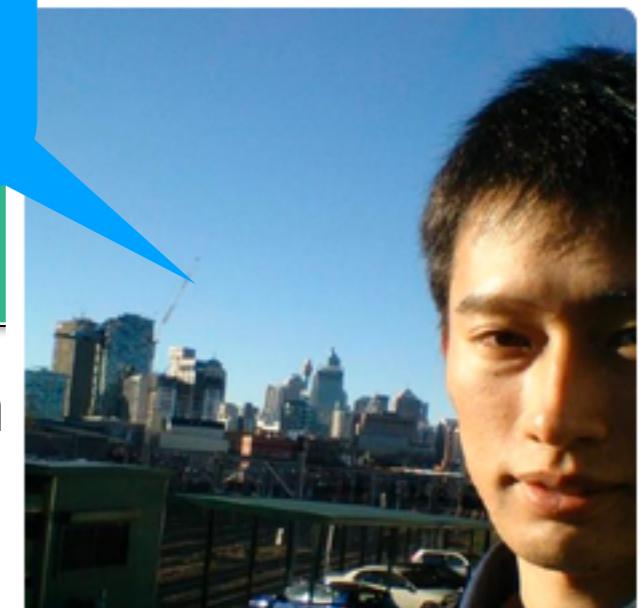
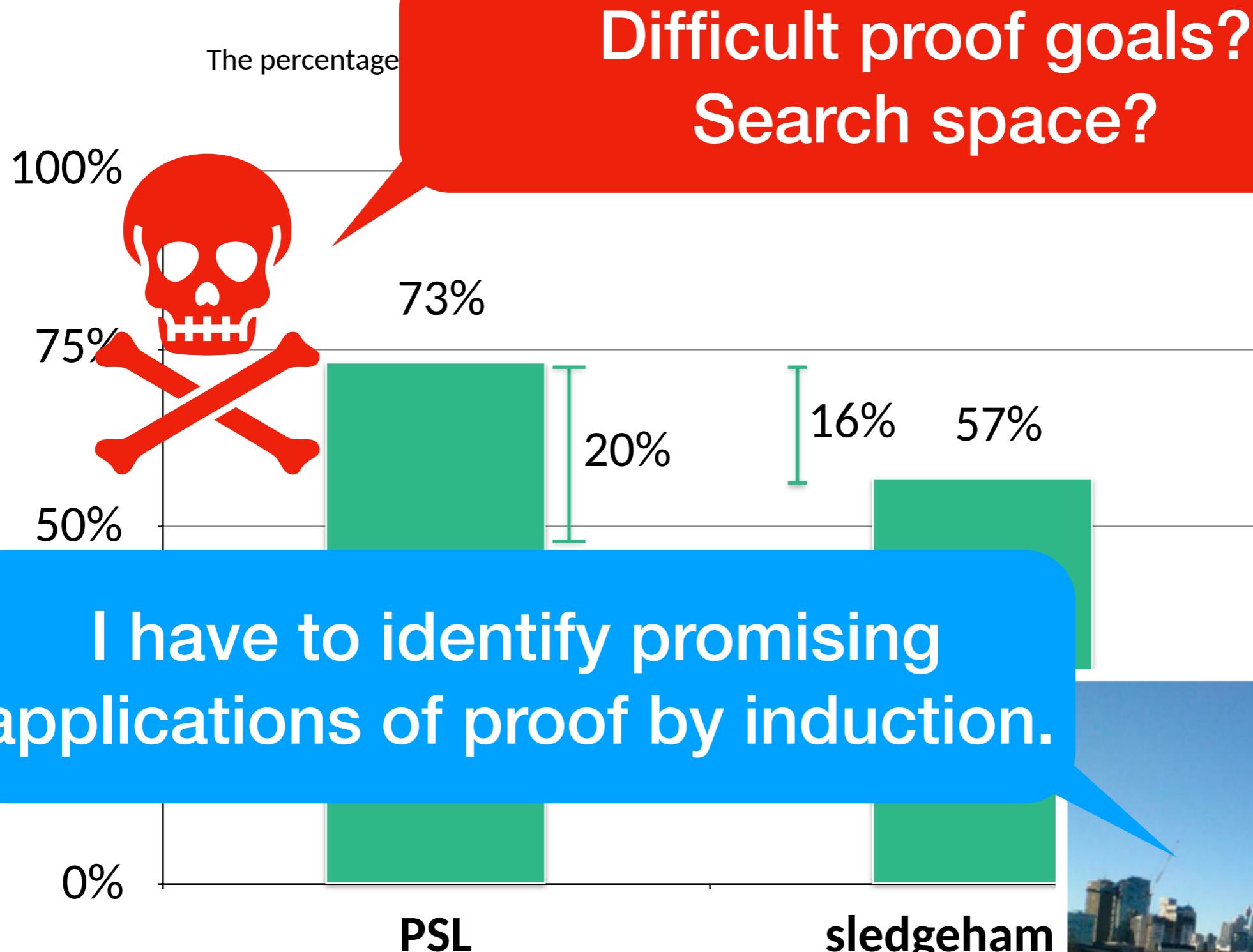
The percentage of automatically proved obligations out of 1526 proof obligations
(timeout = 300s)



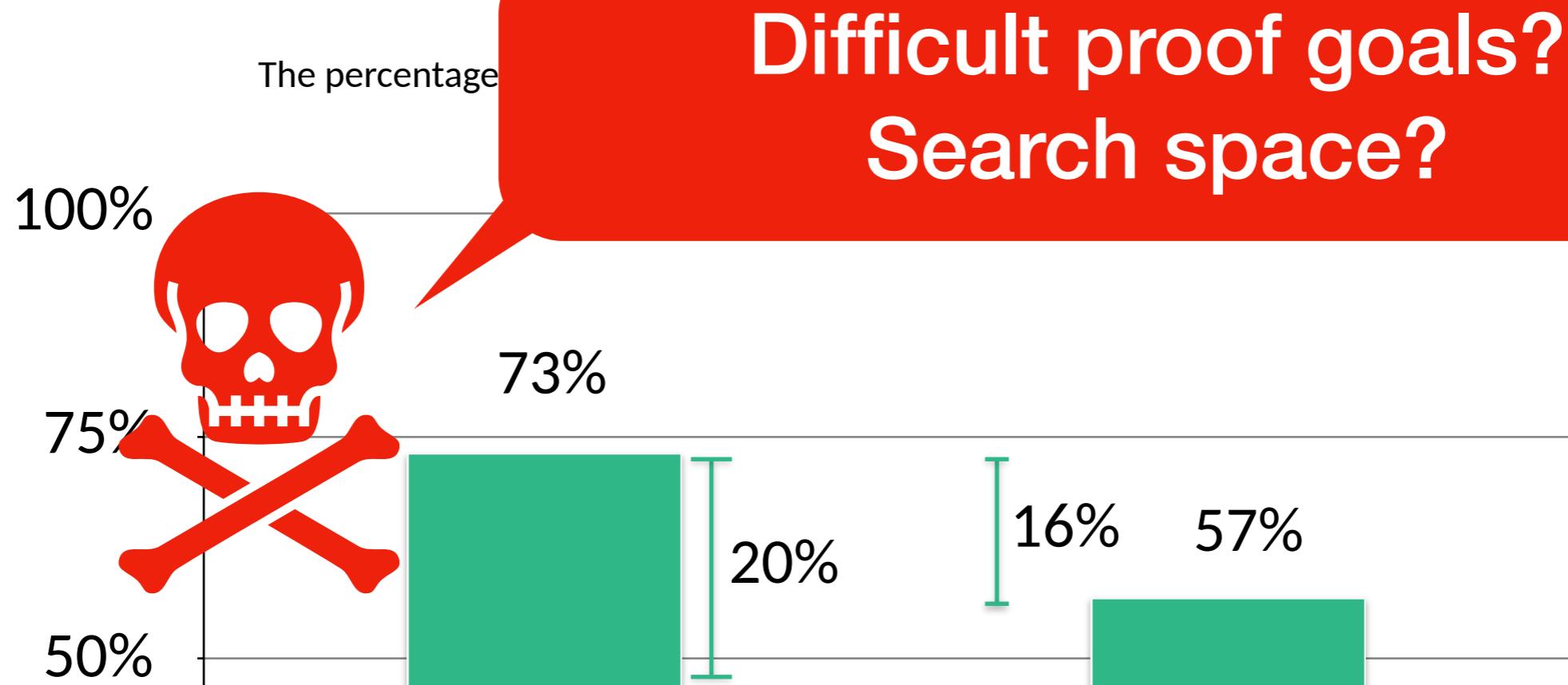
PSL vs sledgehammer



PSL vs sledgehammer



PSL vs sleddaehammer



I have to identify promising applications of proof by induction.

without completing a proof search



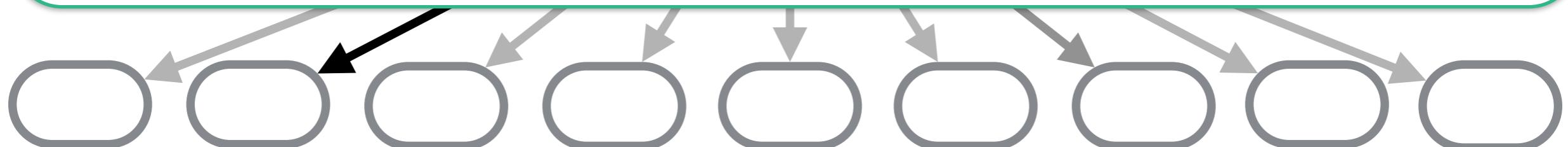
`smart_induct`

`goal`

smart_induct

goal

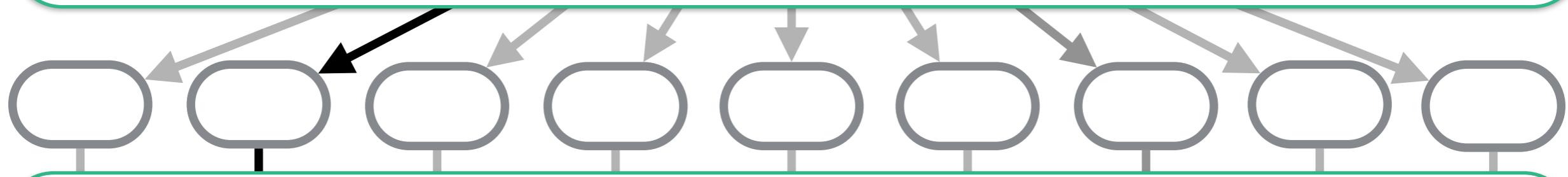
Step 1: creating many inductions



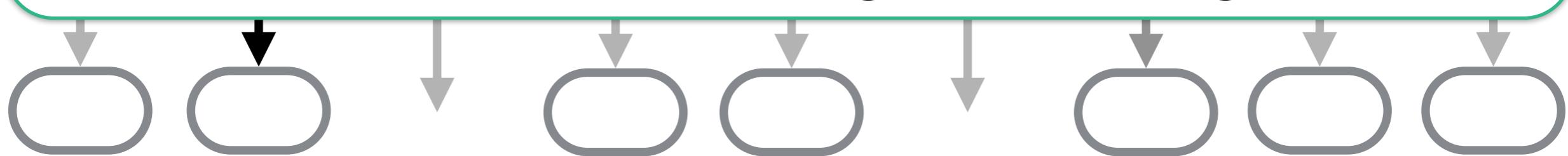
smart_induct

goal

Step 1: creating many inductions



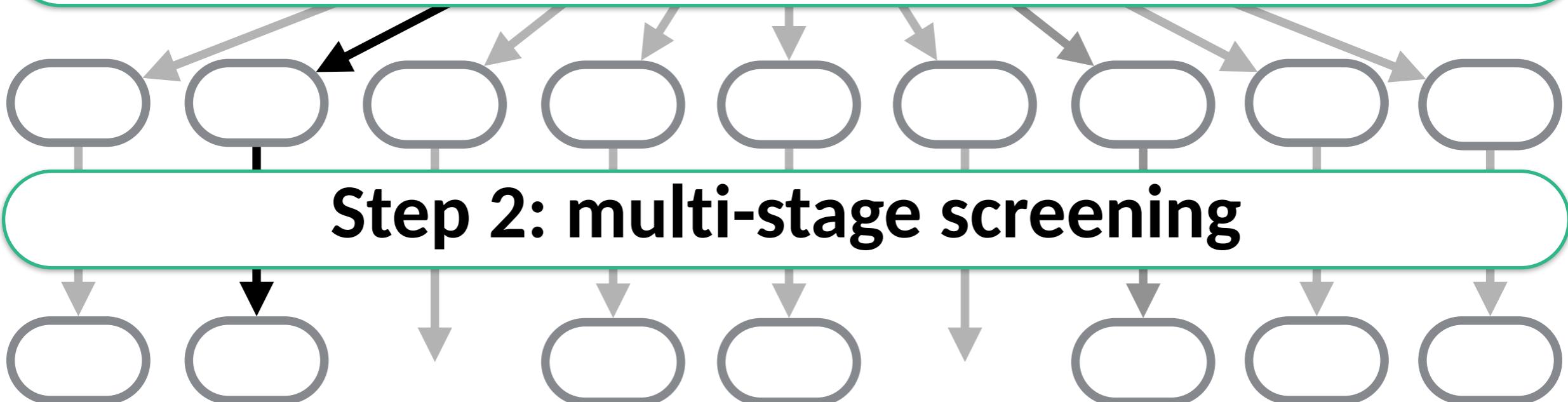
Step 2: multi-stage screening



smart_induct

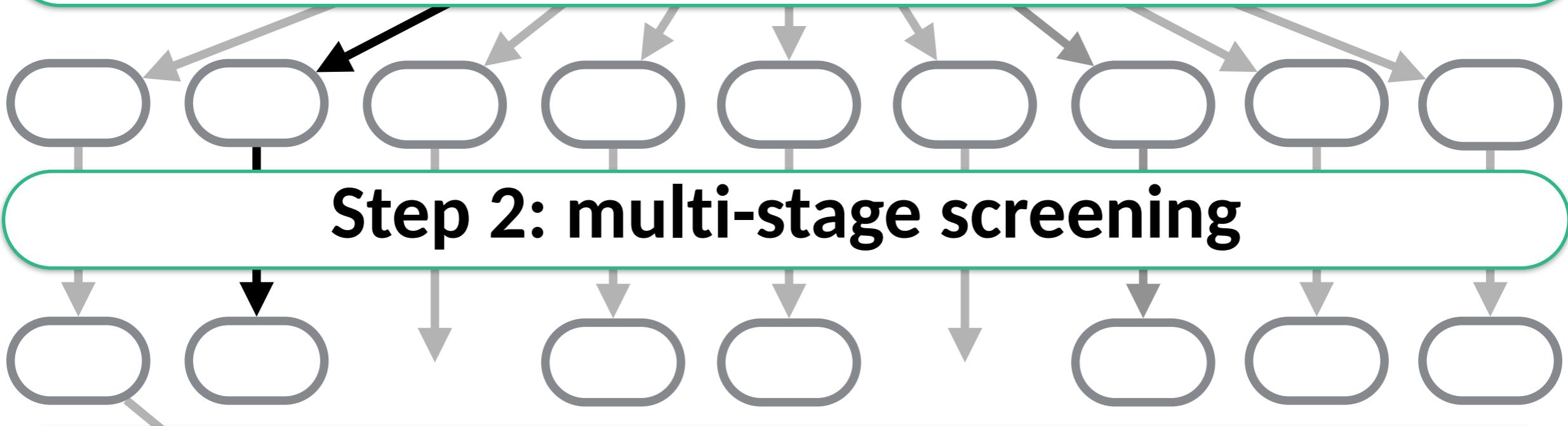
goal

Step 1: creating many inductions



heuristic : (proof goal * induction arguments) -> bool

Step 1: creating many inductions



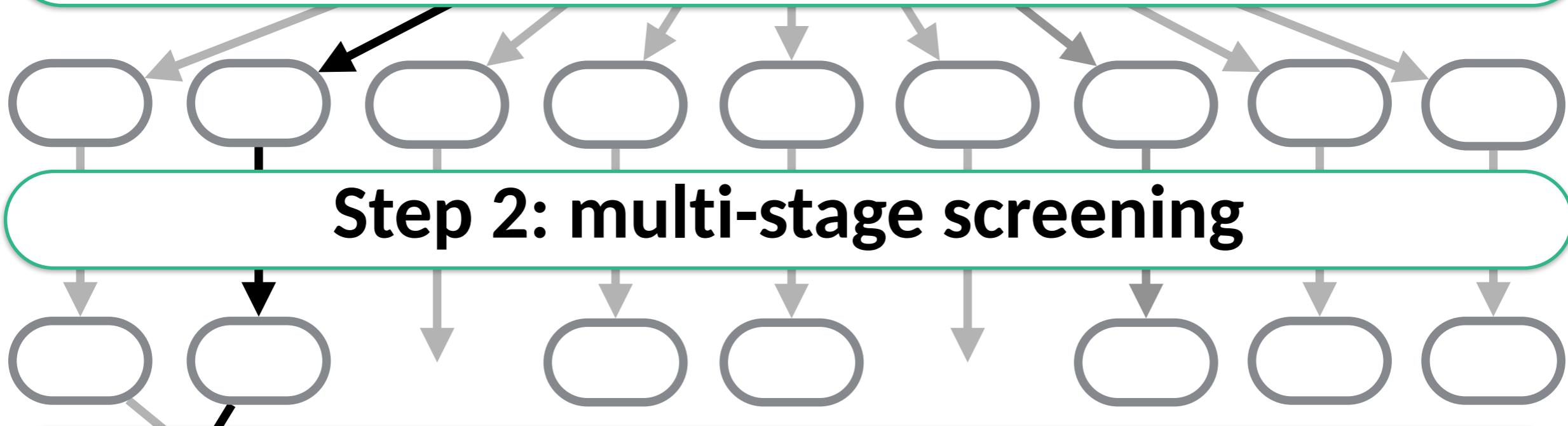
Step 3: scoring using 20 heuristics and sorting

heuristic : (proof goal * induction arguments) -> bool

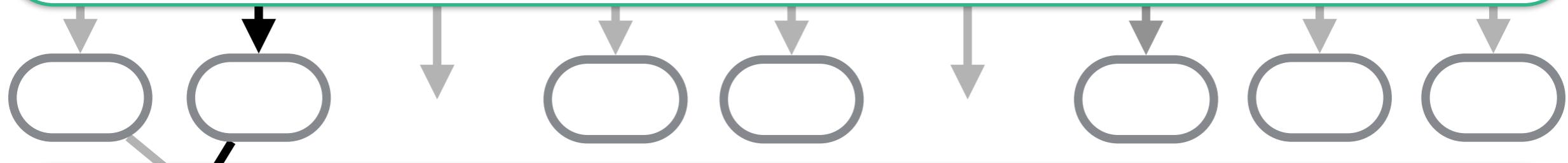
smart_induct

goal

Step 1: creating many inductions



Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting

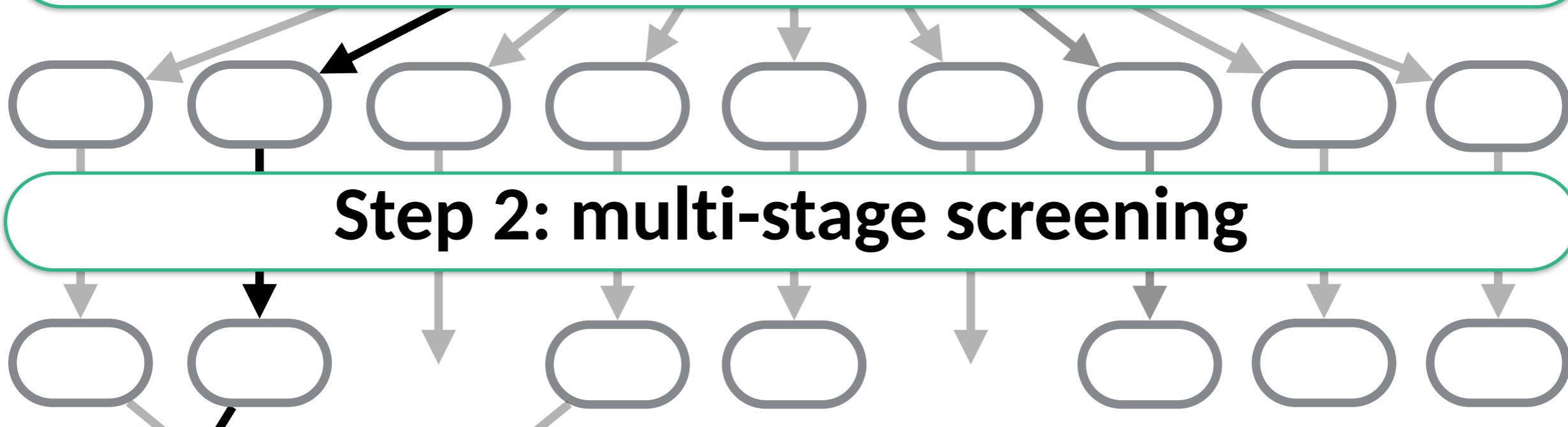


heuristic : (proof goal * induction arguments) -> bool

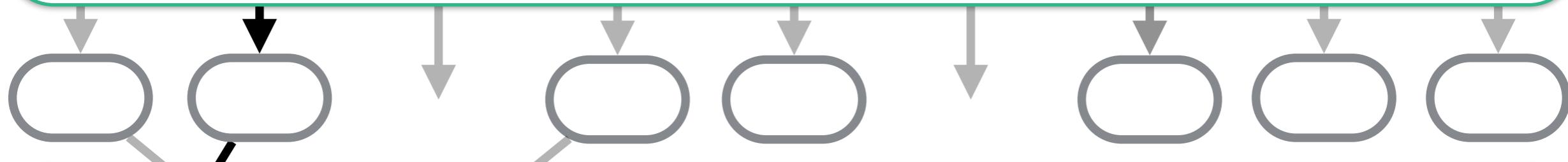
smart_induct

goal

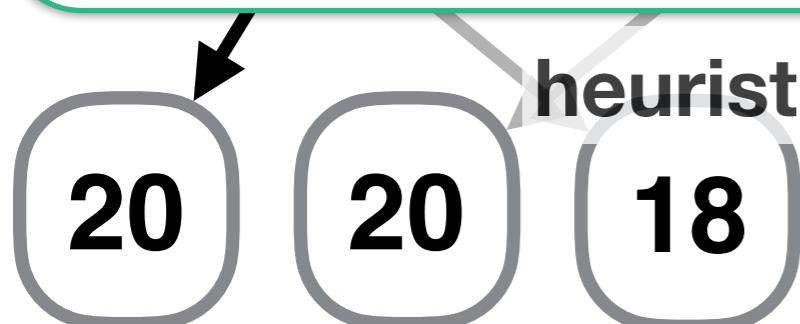
Step 1: creating many inductions



Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting



heuristic : (proof goal * induction arguments) -> bool

20

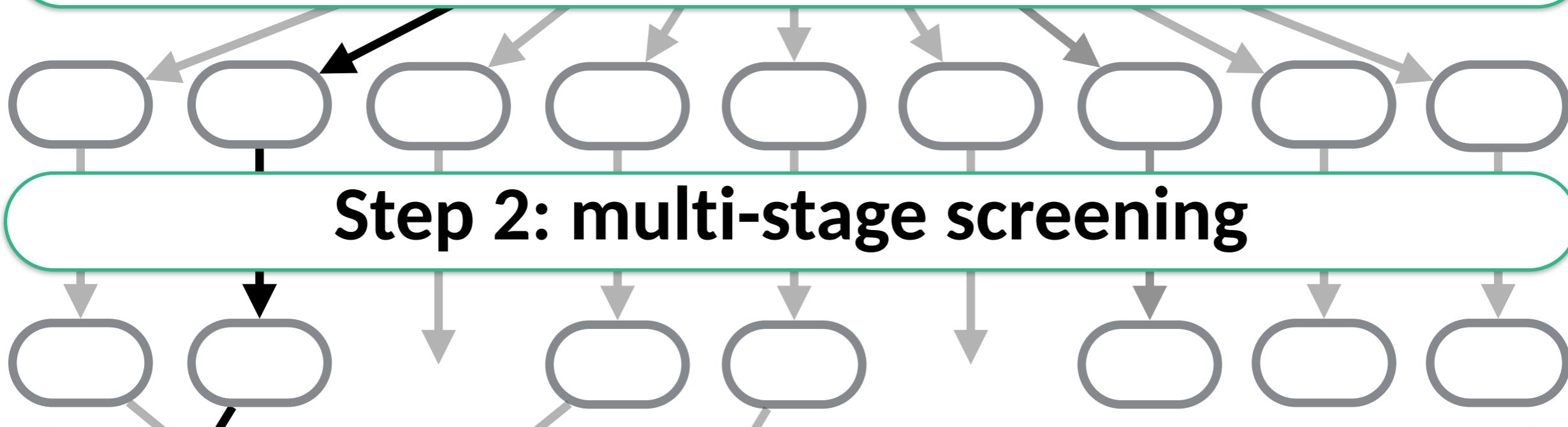
20

18

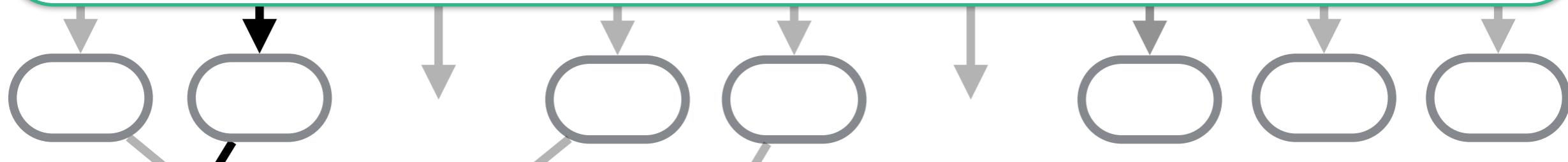
smart_induct

goal

Step 1: creating many inductions



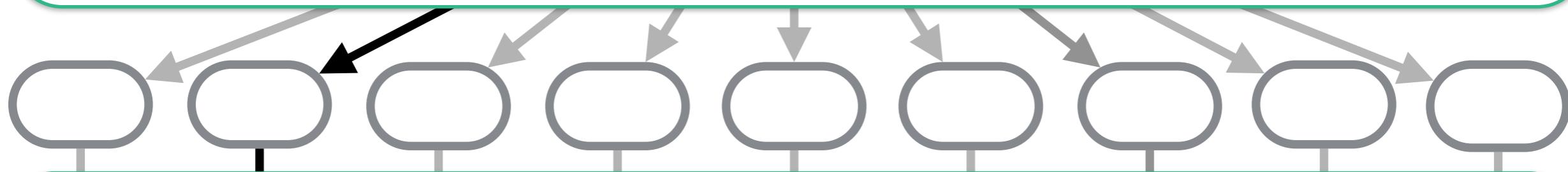
Step 2: multi-stage screening



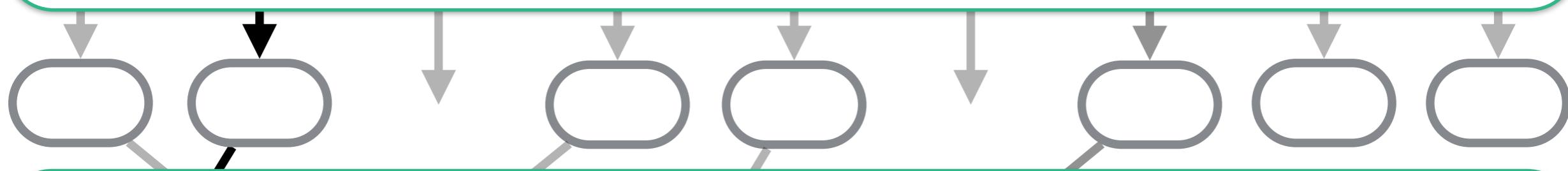
smart_induct

goal

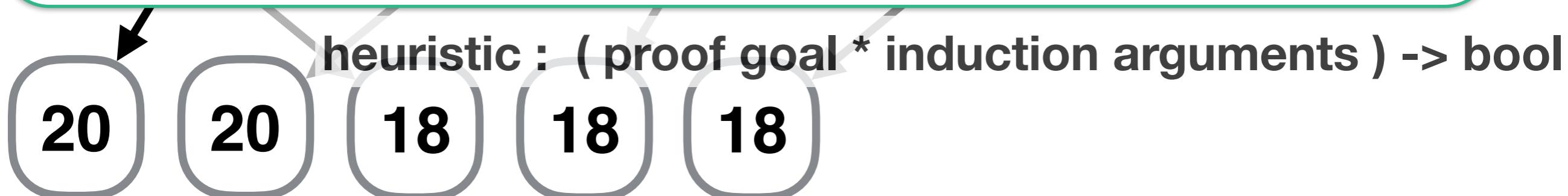
Step 1: creating many inductions



Step 2: multi-stage screening



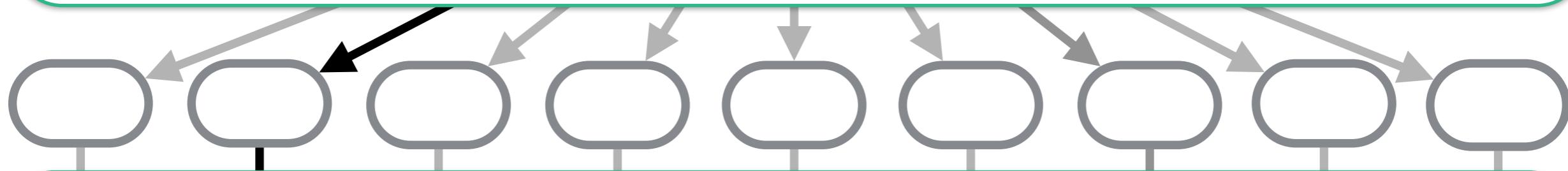
Step 3: scoring using 20 heuristics and sorting



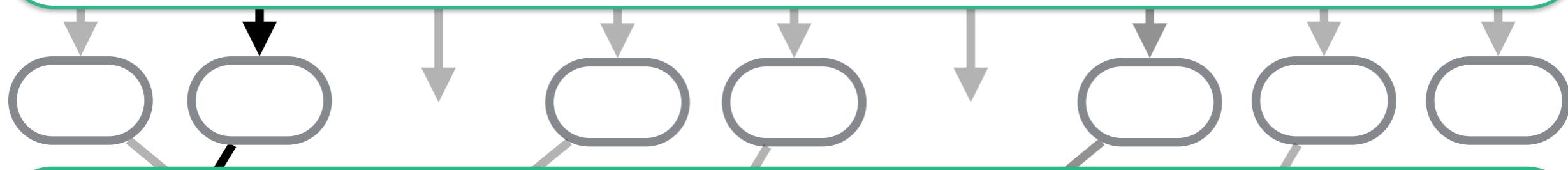
smart_induct

goal

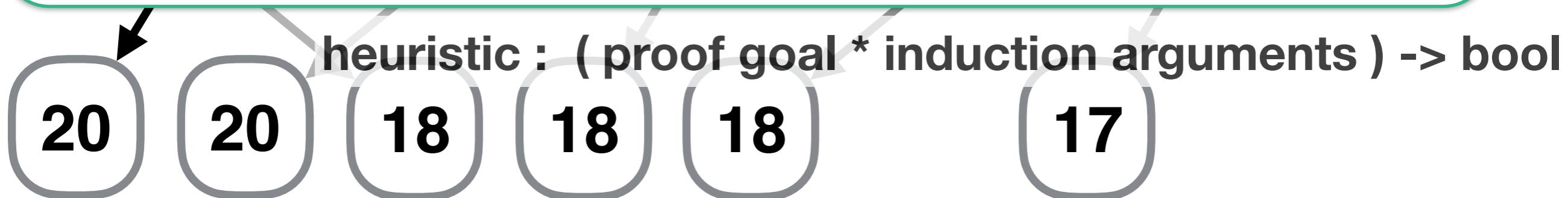
Step 1: creating many inductions



Step 2: multi-stage screening



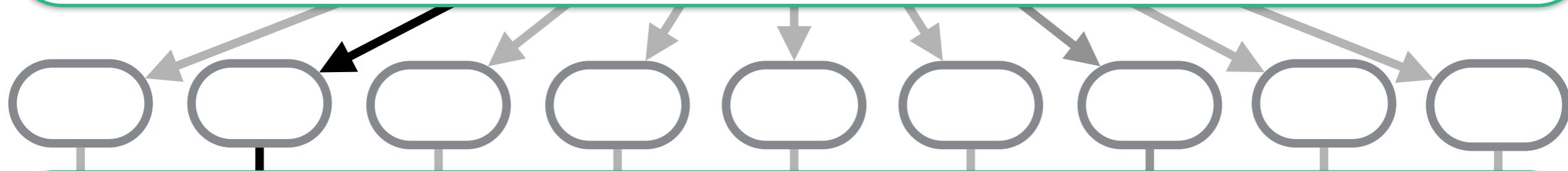
Step 3: scoring using 20 heuristics and sorting



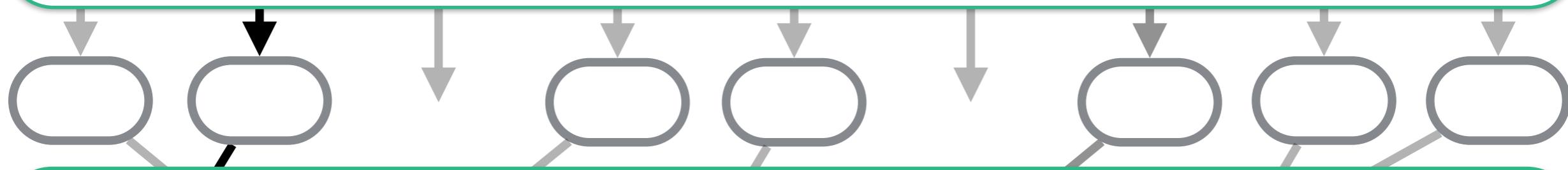
smart_induct

goal

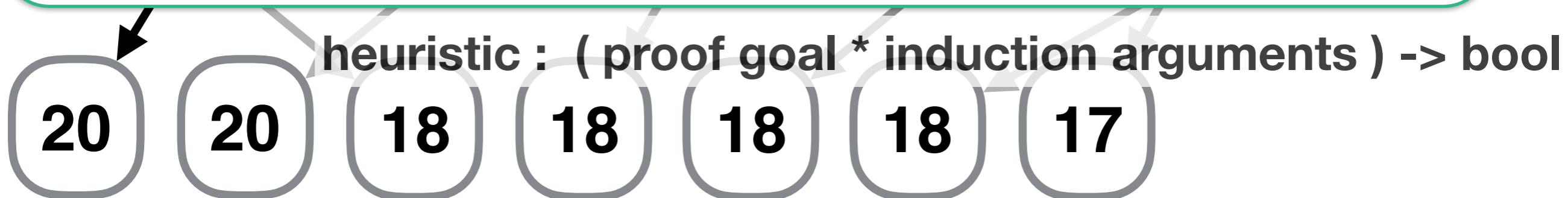
Step 1: creating many inductions



Step 2: multi-stage screening



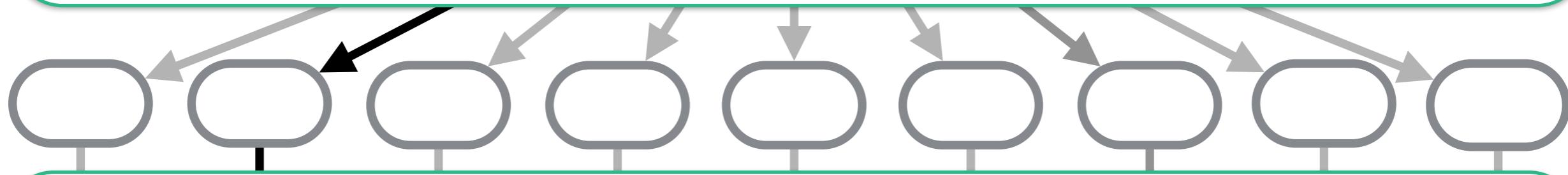
Step 3: scoring using 20 heuristics and sorting



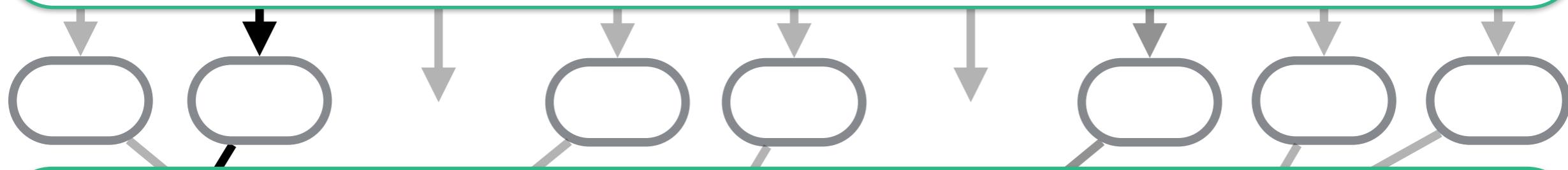
smart_induct

goal

Step 1: creating many inductions



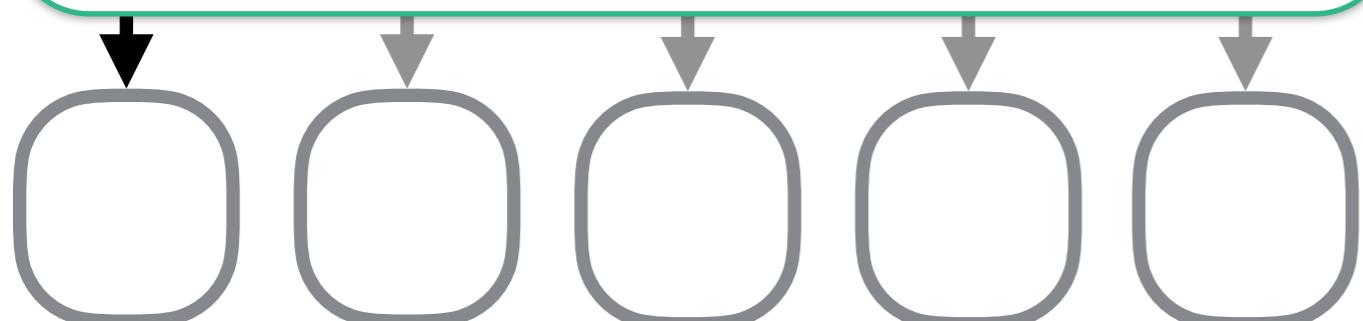
Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting



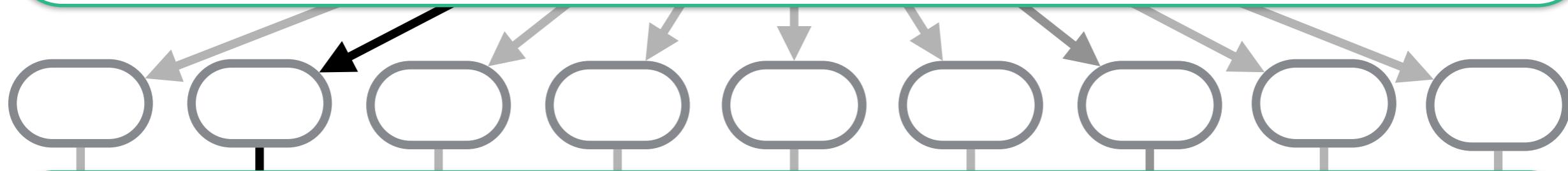
Step 4: short-listing



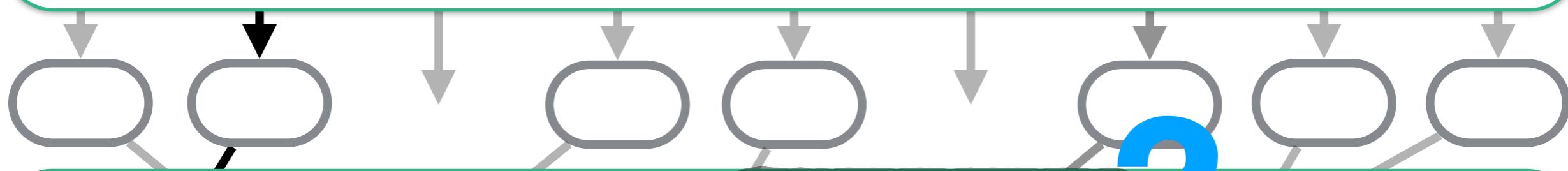
smart_induct

goal

Step 1: creating many inductions



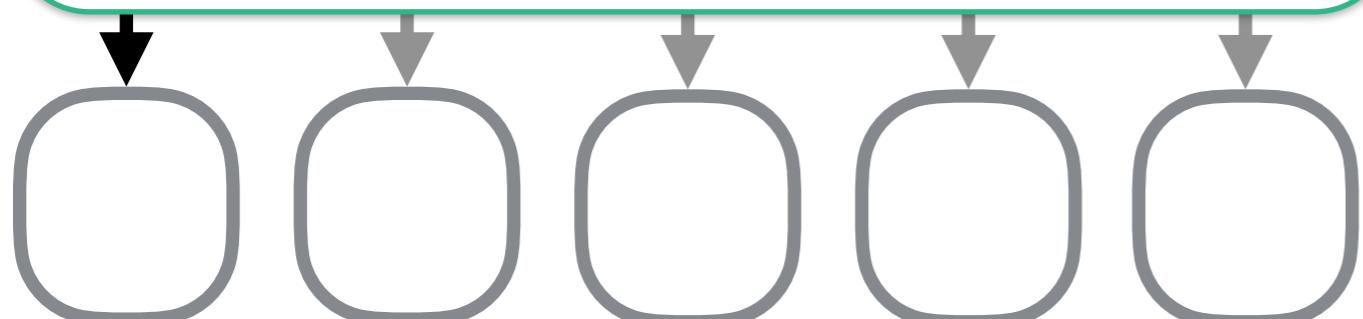
Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting

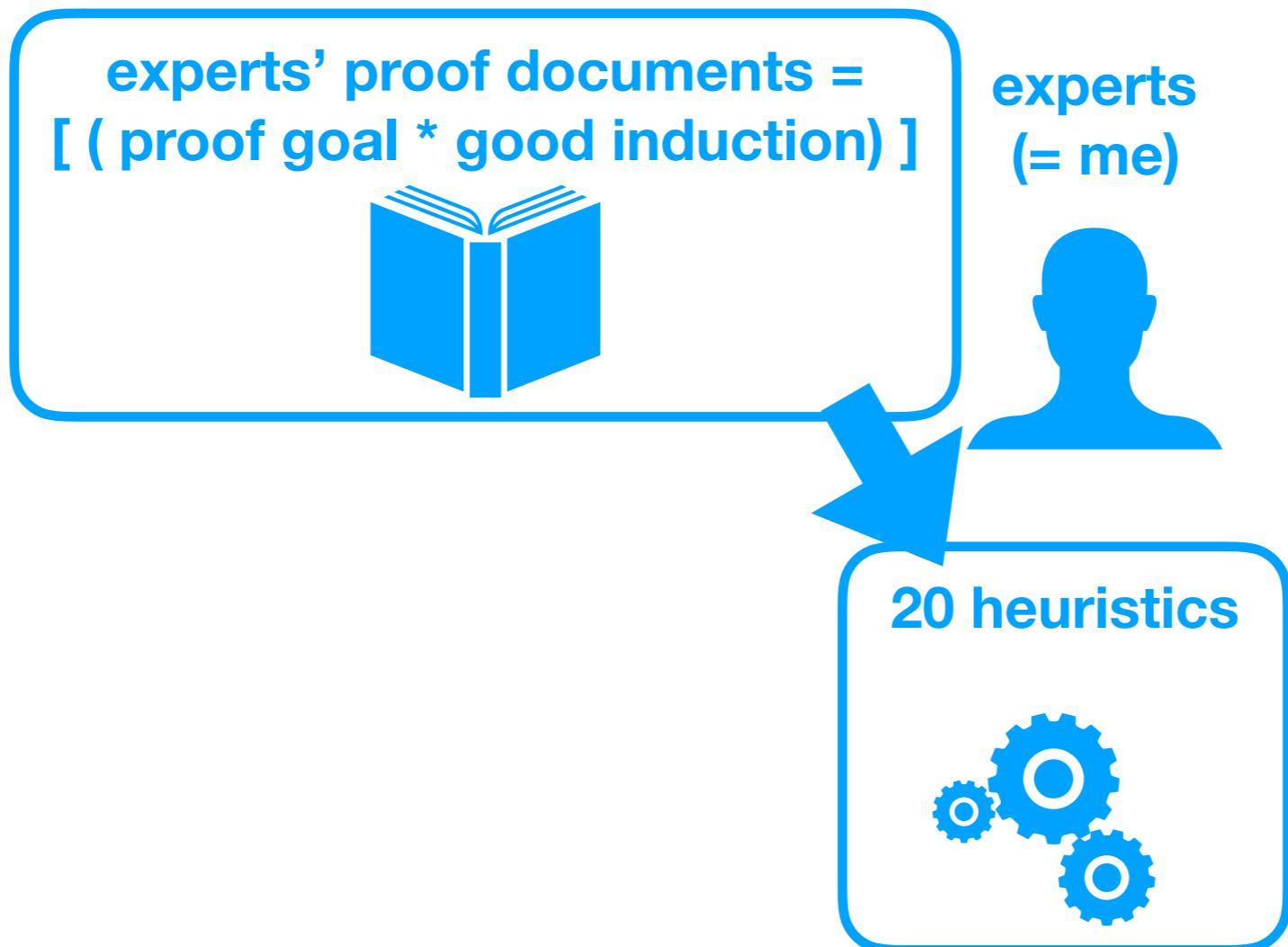


Step 4: short-listing



It is difficult to write induction heuristics.

It is difficult to write induction heuristics.



It is difficult to write induction heuristics.

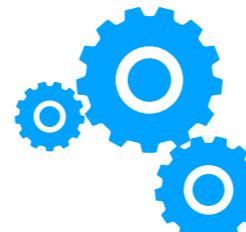
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

experts' proof documents =
[(proof goal * good induction)]



experts
(= me)

20 heuristics



It is difficult to write induction heuristics.

```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

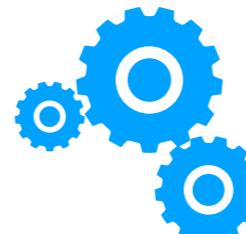
experts' proof documents =
[(proof goal * good induction)]



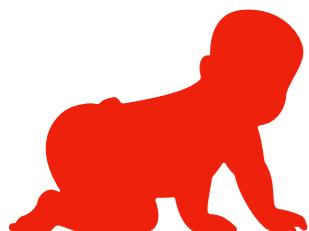
experts
(= me)



20 heuristics



new users



```
lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
```

It is difficult to write induction heuristics.

```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

experts' proof documents =
[(proof goal * good induction)]



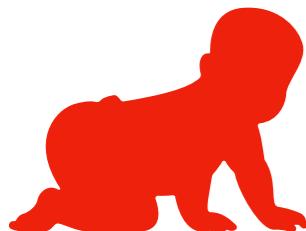
experts
(= me)



20 heuristics



new users



```
lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
```

new proof goal consisting of new constants and variables of new types!

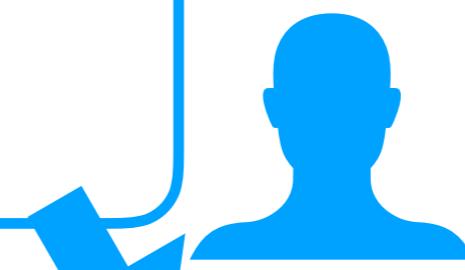
It is difficult to write induction heuristics.

```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

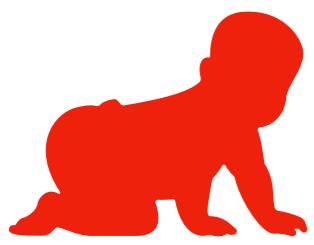
experts' proof documents =
[(proof goal * good induction)]



experts
(= me)

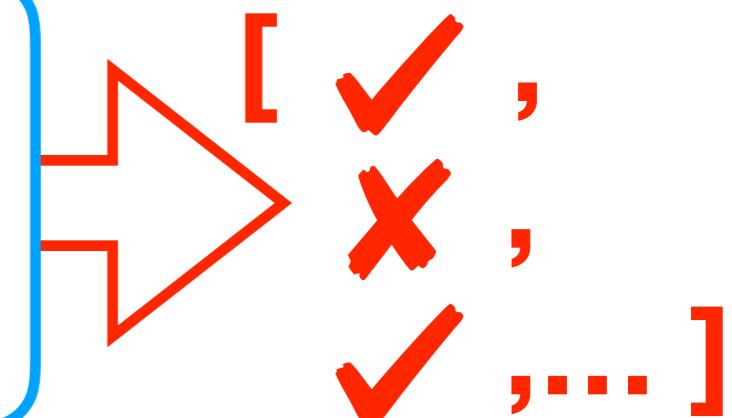
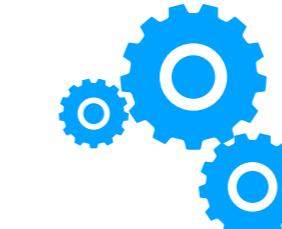


new users



```
proof goal candidate induction  
[on ["is1", "s", "stk"],  
 arb [],  
 rule["exec.induct"]]?]
```

20 heuristics



```
lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
```

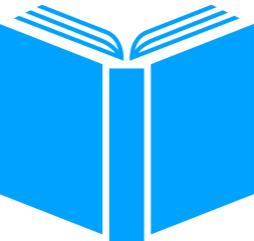
new proof goal consisting of new constants and variables of new types!

It is difficult to write induction heuristics.

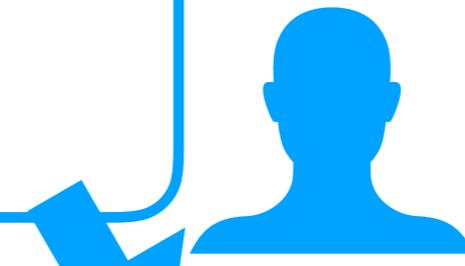
① blue = what I can see before releasing smart_induct

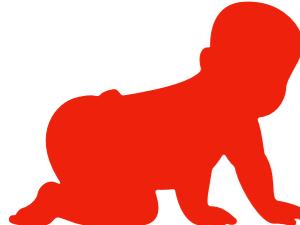
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

experts' proof documents =
[(proof goal * good induction)]

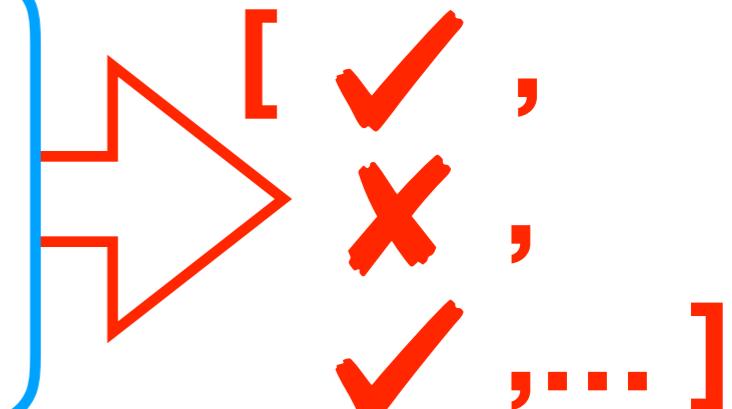
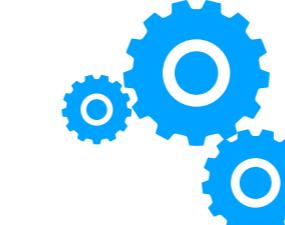


experts
(= me)



new users

proof goal candidate induction
[on ["is1", "s", "stk"],
arb [],
rule["exec.induct"]]?]

20 heuristics



```
lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
```

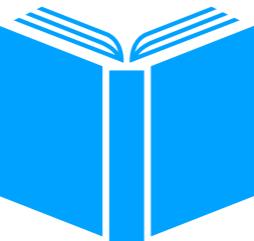
new proof goal consisting of new constants and variables of new types!

It is difficult to write induction heuristics.

① blue = what I can see before releasing smart_induct

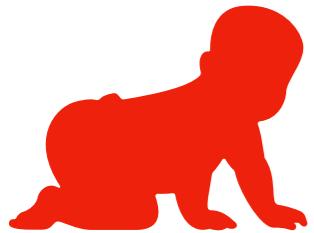
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

experts' proof documents =
[(proof goal * good induction)]

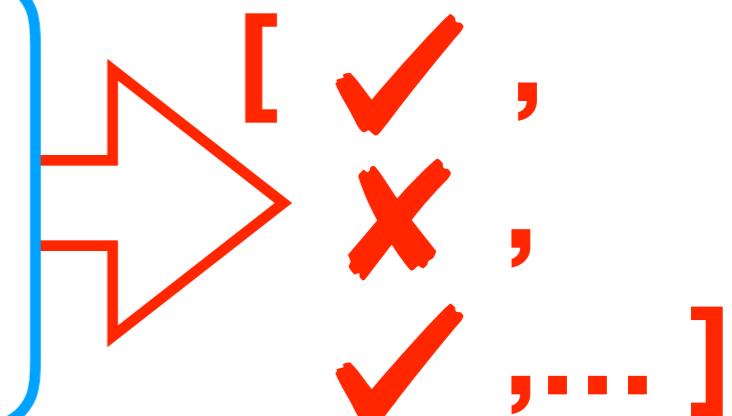
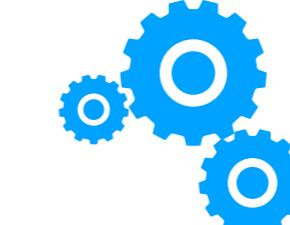


experts
(= me)



new users

proof goal candidate induction
[on ["is1", "s", "stk"],
arb [],
rule["exec.induct"]]?]

20 heuristics



```
lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
```

new proof goal consisting of new constants and variables of new types!

② red = what will be developed after releasing smart_induct

It is difficult to write induction heuristics.

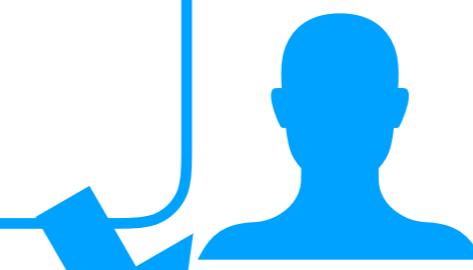
① blue = what I can see before releasing smart_induct

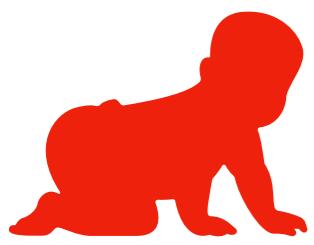
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

experts' proof documents =
[(proof goal * good induction)]

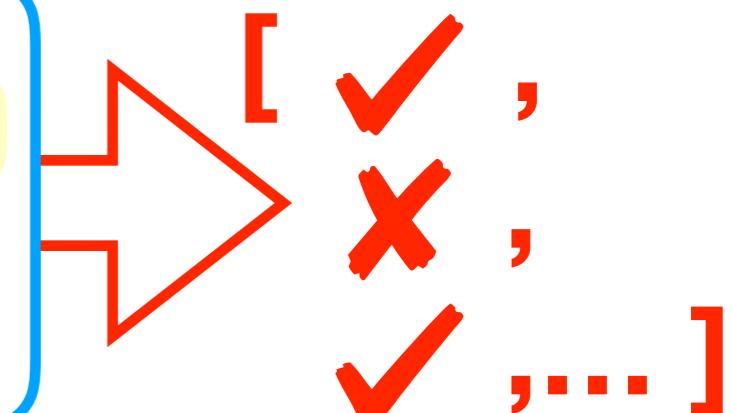


experts
(= me)



new users

proof goal candidate induction
[on ["is1", "s", "stk"],
arb [],
rule["exec.induct"]]?]

20 heuristics
in LiFtEr



```
lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
```

new proof goal consisting of new constants and variables of new types!

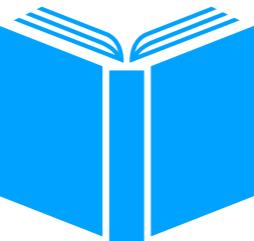
② red = what will be developed after releasing smart_induct

It is difficult to write induction heuristics.

① blue = what I can see before releasing smart_induct

```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

experts' proof documents =
[(proof goal * good induction)]

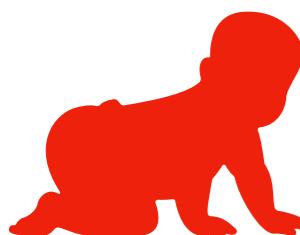


experts
(= me)

LiFtEr: Logical
Feature Extractor

new users

proof goal candidate induction
[on ["is1", "s", "stk"],
arb [],
rule["exec.induct"]]?]



20 heuristics
in LiFtEr



```
lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
```

new proof goal consisting of new constants and variables of new types!

② red = what will be developed after releasing smart_induct

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 :=  $\exists x : type . assertion$ 
            |  $\forall x : type . assertion$ 
            |  $\exists x : term \in modifier\_term . assertion$ 
            |  $\forall x : term \in modifier\_term . assertion$ 
            |  $\exists x : rule . assertion$ 
            |  $\exists x : term\_occurrence \in y : term . assertion$ 
            |  $\forall x : term\_occurrence \in y : term . assertion$ 
connective := True | False | assertion  $\vee$  assertion
              | assertion  $\rightarrow$  assertion |  $\neg$  assertion
pattern := all_only_var | all_constructor
atomic :=
    rule_is_rule_of term_occurrence
    | term_occurrence term_occurrence
    | 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
```

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



```
→  $\exists r1 : \text{rule}. \text{True}$ 
    $\exists r1 : \text{rule}.$ 
    $\exists t1 : \text{term}.$ 
    $\exists to1 : \text{term\_occurrence} \in t1 : \text{term}.$ 
       $r1 \text{ is\_rule\_of } to1$ 
       $\wedge$ 
       $\forall t2 : \text{term} \in \text{induction\_term}.$ 
          $\exists to2 : \text{term\_occurrence} \in t2 : \text{term}.$ 
          $\exists n : \text{number}.$ 
             $\text{is\_nth\_argument\_of } (to2, n, to1)$ 
             $\wedge$ 
             $t2 \text{ is\_nth\_induction\_term } n$ 
```

Example Assertion in LiFtEr (in Abstract Syntax)

implication

↓

$\exists r1 : \text{rule}. \text{True}$

→

$\exists r1 : \text{rule}.$
 $\exists t1 : \text{term}.$
 $\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$
 $r1 \text{ is_rule_of } to1$

\wedge ↪ conjunction

$\forall t2 : \text{term} \in \text{induction_term}.$
 $\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$
 $\exists n : \text{number}.$
 $\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

↓
→ $\exists r1 : \text{rule}. \text{True}$ variable for auxiliary lemmas

$\exists r1 : \text{rule}.$ ←

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge ← conjunction

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of } (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

↓
 $\exists r1 : \text{rule}. \text{True}$ → variable for auxiliary lemmas
 $\exists r1 : \text{rule}.$ ↙
 $\exists t1 : \text{term}.$ ↙ variable for terms
 $\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$
 $r1 \text{ is_rule_of } to1$
 ∧ ↙ conjunction
 $\forall t2 : \text{term} \in \text{induction_term}.$
 $\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$
 $\exists n : \text{number}.$
 $\text{is_nth_argument_of } (to2, n, to1)$
 \wedge
 $t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

↓
 $\exists r1 : \text{rule}. \text{True}$ → variable for auxiliary lemmas
 $\exists r1 : \text{rule}.$ ←
 $\exists t1 : \text{term}.$ ← variable for terms
 $\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$
 $r1 \text{ is_rule_of } to1$ ← variable for term occurrences
 ∧ ← conjunction
 $\forall t2 : \text{term} \in \text{induction_term}.$
 $\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$
 $\exists n : \text{number}.$
 $\text{is_nth_argument_of } (to2, n, to1)$
 ∧
 $t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

↓
 $\exists r1 : \text{rule}. \text{True}$ → variable for auxiliary lemmas
 $\exists r1 : \text{rule}.$ ←
 $\exists t1 : \text{term}.$ ← variable for terms
 $\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$ ← variable for term occurrences
 $r1 \text{ is_rule_of } to1$ ←
 \wedge conjunction
 $\forall t2 : \text{term} \in \text{induction_term}.$
 $\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$
 $\exists n : \text{number}.$ ← variable for natural numbers
 $\text{is_nth_argument_of } (to2, n, to1)$
 \wedge
 $t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

$\exists r1 : \text{rule}. \text{True}$

\rightarrow variable for auxiliary lemmas

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$ variable for terms

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$ variable for term occurrences

\wedge conjunction

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$ variable for natural numbers

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

universal
quantifier

Example Assertion in LiFtEr (in Abstract Syntax)

implication existential quantifier

$\exists r1 : \text{rule}. \text{True}$

\rightarrow variable for auxiliary lemmas

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$ variable for terms

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$ variable for term occurrences

\wedge conjunction

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$ variable for natural numbers

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

universal quantifier

Example Assertion in LiFtEr (in Abstract Syntax)

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

implication existential quantifier

$\exists r1 : \text{rule}. \text{True}$ variable for auxiliary lemmas

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$ variable for terms

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$ variable for term occurrences

$r1 \text{ is_rule_of } to1$

\wedge conjunction

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$ variable for natural numbers

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

universal
quantifier

```

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

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

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

```

$\exists r1 : \text{rule}. \text{True}$

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

```

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

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

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

```

$\exists r1 : \text{rule}. \text{True}$

→

$\exists r1 : \text{rule}.$

$(r1 = \text{itrev.induct})$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

r1

```

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

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

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

```

$\exists r1 : \text{rule}. \text{True}$

\rightarrow

$\exists r1 : \text{rule}.$

$(r1 = \text{itrev.induct})$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

$r1$

```

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

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

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

```

$\exists r1 : \text{rule}. \text{True}$

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

($r1 = \text{itrev.induct}$)
 ($t1 = \text{itrev}$)

$r1$

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

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

to1

```
lemma "itrev xs ys = rev xs @ ys"
```

```
apply(induct xs ys rule:"itrev.induct")
apply auto done
```

$\exists r1 : \text{rule}. \text{True}$

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

($r1 = \text{itrev.induct}$)
($t1 = \text{itrev}$)
($to1 = \text{itrev}$)

r1

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

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

to1

```
lemma "itrev xs ys = rev xs @ ys"
```

```
apply(induct xs ys rule:"itrev.induct")
apply auto done
```

$\exists r1 : \text{rule}. \text{True}$

\rightarrow

$\exists r1 : \text{rule}.$

($r1 = \text{itrev.induct}$)

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

r1

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

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

to1

```
lemma "itrev xs ys = rev xs @ ys"
```

```
apply(induct xs ys rule:"itrev.induct")
apply auto done
```

$\exists r1 : \text{rule}. \text{True}$

\rightarrow

$\exists r1 : \text{rule}.$

($r1 = \text{itrev.induct}$)

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

$\exists t1 : \text{term}.$

r1

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$ True! $r1 (= \text{itrev.induct})$ is a lemma about $to1 (= \text{itrev})$.

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

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

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

to1

```
lemma "itrev xs ys = rev xs @ ys"
```

```
apply(induct xs ys rule:"itrev.induct")
apply auto done
```

$\exists r1 : \text{rule}. \text{True}$

\rightarrow

$\exists r1 : \text{rule}.$

($r1 = \text{itrev.induct}$)

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

 $r1 \text{ is_rule_of } to1$

True! $r1 (= \text{itrev.induct})$ is a lemma about $to1 (= \text{itrev})$.

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

$r1$

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

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

to1

```
lemma "itrev xs ys = rev xs @ ys"
```

```
apply(induct xs ys rule:"itrev.induct")
apply auto done
```

$\exists r1 : \text{rule}. \text{True}$

\rightarrow

$\exists r1 : \text{rule}.$

($r1 = \text{itrev.induct}$)

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{is_rule_of } to1 \quad \text{True! } r1 (= \text{itrev.induct}) \text{ is a lemma about } to1 (= \text{itrev}).$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of } (to2, n, to1)$

\wedge

$t2 \text{is_nth_induction_term } n$

r1

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

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

to1

```
lemma "itrev xs ys = rev xs @ ys"
```

```
apply(induct xs ys rule:"itrev.induct")
apply auto done
```

$\exists r1 : \text{rule}. \text{True}$

\rightarrow

$\exists r1 : \text{rule}.$

($r1 = \text{itrev.induct}$)

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{is_rule_of } to1$ True! $r1 (= \text{itrev.induct})$ is a lemma about $to1 (= \text{itrev})$.

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{is_nth_induction_term } n$

r1

```

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

```

```

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

```

t_{01}

```

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

```

t_2

r_1

$\exists r_1 : \text{rule}. \text{True}$

\rightarrow

$\exists r_1 : \text{rule}.$

$\exists t_1 : \text{term}.$

$\exists t_{01} : \text{term_occurrence} \in t_1 : \text{term}.$

$r_1 \text{ is_rule_of } t_{01}$ True! $r_1 (= \text{itrev.induct})$ is a lemma about $t_{01} (= \text{itrev})$.

\wedge

$\forall t_2 : \text{term} \in \text{induction_term}.$

$(t_2 = xs \text{ and } ys)$

$\exists t_{02} : \text{term_occurrence} \in t_2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (t_{02}, n, t_{01})$

\wedge

$t_2 \text{ is_nth_induction_term } n$

```

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

```

```

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

```

t_{01}

t_{02}

lemma "itrev xs ys = rev xs @ ys"

apply(induct xs ys rule:"itrev.induct")
apply auto done

t_2

r_1

$\exists r_1 : \text{rule}. \text{True}$

\rightarrow

$\exists r_1 : \text{rule}.$

$\exists t_1 : \text{term}.$

$\exists t_{01} : \text{term_occurrence} \in t_1 : \text{term}.$

$r_1 \text{ is_rule_of } t_{01}$ True! $r_1 (= \text{itrev.induct})$ is a lemma about $t_{01} (= \text{itrev})$.

\wedge

$\forall t_2 : \text{term} \in \text{induction_term}.$

$\exists t_{02} : \text{term_occurrence} \in t_2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of } (t_{02}, n, t_{01})$

\wedge

$t_2 \text{ is_nth_induction_term } n$

($r_1 = \text{itrev.induct}$)

($t_1 = \text{itrev}$)

($t_{01} = \text{itrev}$)

($t_2 = \text{xs and ys}$)

($t_{02} = \text{xs and ys}$)

```

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

```

```

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

```

t_{01}

t_{02}

lemma "itrev xs ys = rev xs @ ys"

apply(induct xs ys rule:"itrev.induct")
apply auto done

t_2

r_1

$\exists r_1 : \text{rule}. \text{True}$

\rightarrow

$\exists r_1 : \text{rule}.$

$\exists t_1 : \text{term}.$

$\exists t_{01} : \text{term_occurrence} \in t_1 : \text{term}.$

$r_1 \text{ is_rule_of } t_{01}$ True! $r_1 (= \text{itrev.induct})$ is a lemma about $t_{01} (= \text{itrev})$.

\wedge

$\forall t_2 : \text{term} \in \text{induction_term}.$

$\exists t_{02} : \text{term_occurrence} \in t_2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of } (t_{02}, n, t_{01})$

\wedge

$t_2 \text{ is_nth_induction_term } n$

($r_1 = \text{itrev.induct}$)

($t_1 = \text{itrev}$)

($t_{01} = \text{itrev}$)

($t_2 = \text{xs and ys}$)

($t_{02} = \text{xs and ys}$)

```

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

```

```

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

```

t_{01}

t_{02}

lemma "itrev xs ys = rev xs @ ys"

apply(induct xs ys rule:"itrev.induct")
apply auto done

t_2

r_1

$\exists r_1 : \text{rule}. \text{True}$

\rightarrow

$\exists r_1 : \text{rule}.$

$\exists t_1 : \text{term}.$

$\exists t_{01} : \text{term_occurrence} \in t_1 : \text{term}.$

$r_1 \text{ is_rule_of } t_{01}$ True! $r_1 (= \text{itrev.induct})$ is a lemma about $t_{01} (= \text{itrev})$.

\wedge

$\forall t_2 : \text{term} \in \text{induction_term}.$

$\exists t_{02} : \text{term_occurrence} \in t_2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of } (t_{02}, n, t_{01})$

\wedge

$t_2 \text{ is_nth_induction_term } n$

($r_1 = \text{itrev.induct}$)

($t_1 = \text{itrev}$)

($t_{01} = \text{itrev}$)

($t_2 = \text{xs and ys}$)

($t_{02} = \text{xs and ys}$)

```

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

```

```

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

```

t_{01}

first

t_{02}

lemma "itrev xs ys = rev xs @ ys"

apply(induct xs ys rule:"itrev.induct")
apply auto done

t_2

first

r_1

$\exists r_1 : \text{rule}. \text{True}$

\rightarrow

$\exists r_1 : \text{rule}.$

$\exists t_1 : \text{term}.$

$\exists t_{01} : \text{term_occurrence} \in t_1 : \text{term}.$

$r_1 \text{ is_rule_of } t_{01}$ True! $r_1 (= \text{itrev.induct})$ is a lemma about $t_{01} (= \text{itrev})$.

\wedge

$\forall t_2 : \text{term} \in \text{induction_term}.$

$\exists t_{02} : \text{term_occurrence} \in t_2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of } (t_{02}, n, t_{01})$

\wedge

$t_2 \text{ is_nth_induction_term } n$

($r_1 = \text{itrev.induct}$)

($t_1 = \text{itrev}$)

($t_{01} = \text{itrev}$)

($t_2 = \text{xs and ys}$)

($t_{02} = \text{xs and ys}$)

when t_2 is xs ($n = 1$)?

```

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

```

```

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

```

t_{01}

first

t_{02}

lemma "itrev xs ys = rev xs @ ys"

apply(induct xs ys rule:"itrev.induct")
apply auto done

t_2

first

r_1

$\exists r_1 : \text{rule}. \text{True}$

\rightarrow

$\exists r_1 : \text{rule}.$

$\exists t_1 : \text{term}.$

$\exists t_{01} : \text{term_occurrence} \in t_1 : \text{term}.$

$r_1 \text{ is_rule_of } t_{01}$ True! $r_1 (= \text{itrev.induct})$ is a lemma about $t_{01} (= \text{itrev})$.

\wedge

$\forall t_2 : \text{term} \in \text{induction_term}.$

$\exists t_{02} : \text{term_occurrence} \in t_2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of } (t_{02}, n, t_{01})$

\wedge

$t_2 \text{ is_nth_induction_term } n$

($r_1 = \text{itrev.induct}$)

($t_1 = \text{itrev}$)

($t_{01} = \text{itrev}$)

($t_2 = xs \text{ and } ys$)

($t_{02} = xs \text{ and } ys$)

when t_2 is xs ($n = 1$)

```

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

```

```

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

```

t_{01}

$\begin{matrix} \text{first} & \text{second} \\ t_{01} & t_{02} \end{matrix}$

lemma "itrev xs ys = rev xs @ ys"

apply(induct xs ys rule:"itrev.induct")
apply auto done

$\begin{matrix} t_2 & & \text{second} \\ & \downarrow & \\ & \text{first} & \end{matrix}$

r_1

$\exists r_1 : \text{rule}. \text{True}$

\rightarrow

$\exists r_1 : \text{rule}.$

$\exists t_1 : \text{term}.$

$\exists t_{01} : \text{term_occurrence} \in t_1 : \text{term}.$

$r_1 \text{ is_rule_of } t_{01}$ True! $r_1 (= \text{itrev.induct})$ is a lemma about $t_{01} (= \text{itrev})$.

\wedge

$\forall t_2 : \text{term} \in \text{induction_term}.$

$\exists t_{02} : \text{term_occurrence} \in t_2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of } (t_{02}, n, t_{01})$

\wedge

$t_2 \text{ is_nth_induction_term } n$

($r_1 = \text{itrev.induct}$)

($t_1 = \text{itrev}$)

($t_{01} = \text{itrev}$)

($t_2 = xs$ and ys)

($t_{02} = xs$ and ys)

when t_2 is xs ($n = 1$)

when t_2 is ys ($n = 2$) ?

```

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

```

```

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

```

t_{01}

$\begin{matrix} \text{first} & \text{second} \\ t_{01} & t_{02} \end{matrix}$

lemma "itrev xs ys = rev xs @ ys"

apply(induct xs ys rule:"itrev.induct")
apply auto done

$\begin{matrix} t_2 & & \text{second} \\ & \downarrow & \\ & \text{first} & \end{matrix}$

r_1

$\exists r_1 : \text{rule}. \text{True}$

\rightarrow

$\exists r_1 : \text{rule}.$

$\exists t_1 : \text{term}.$

$\exists t_{01} : \text{term_occurrence} \in t_1 : \text{term}.$

$r_1 \text{ is_rule_of } t_{01}$ True! $r_1 (= \text{itrev.induct})$ is a lemma about $t_{01} (= \text{itrev})$.



$\forall t_2 : \text{term} \in \text{induction_term}.$

$\exists t_{02} : \text{term_occurrence} \in t_2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of } (t_{02}, n, t_{01})$

\wedge

$t_2 \text{ is_nth_induction_term } n$

($r_1 = \text{itrev.induct}$)

($t_1 = \text{itrev}$)

($t_{01} = \text{itrev}$)

($t_2 = xs$ and ys)

($t_{02} = xs$ and ys)

when t_2 is xs ($n = 1$)

when t_2 is ys ($n = 2$)

```

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

```

```

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

```

t_{01}

lemma

" $itrev \underline{xs} \underline{ys} = rev \underline{xs} @ \underline{ys}$ "

apply(induct $\underline{xs} \underline{ys}$ rule:"itrev.induct")

apply auto done

t_2

second

first

r_1

$\exists r_1 : \text{rule}. \text{True}$

\rightarrow

$\exists r_1 : \text{rule}.$

$\exists t_1 : \text{term}.$

$\exists t_{01} : \text{term_occurrence} \in t_1 : \text{term}.$

$r_1 \text{is_rule_of } t_{01}$ True! $r_1 (= \text{itrev.induct})$ is a lemma about $t_{01} (= \text{itrev})$.

\wedge

$\forall t_2 : \text{term} \in \text{induction_term}.$

$\exists t_{02} : \text{term_occurrence} \in t_2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_n}^{\text{th}}\text{argument_of } (t_{02}, n, t_{01})$

\wedge

$t_2 \text{is_n}^{\text{th}}\text{induction_term } n$

($r_1 = \text{itrev.induct}$)

($t_1 = \text{itrev}$)

($t_{01} = \text{itrev}$)

($t_2 = \underline{xs}$ and \underline{ys})

($t_{02} = \underline{xs}$ and \underline{ys})

when t_2 is xs ($n = 1$)

when t_2 is ys ($n = 2$)

```
 $\exists r1 : \text{rule}. \text{True}$ 
```

\rightarrow

```
 $\exists r1 : \text{rule}.$ 
```

```
 $\exists t1 : \text{term}.$ 
```

```
 $\exists to1 : \text{term\_occurrence} \in t1 : \text{term}.$ 
```

```
 $r1 \text{ is\_rule\_of } to1$ 
```

\wedge

```
 $\forall t2 : \text{term} \in \text{induction\_term}.$ 
```

```
 $\exists to2 : \text{term\_occurrence} \in t2 : \text{term}.$ 
```

```
 $\exists n : \text{number}.$ 
```

```
 $\text{is\_nth\_argument\_of} (to2, n, to1)$ 
```

\wedge

```
 $t2 \text{ is\_nth\_induction\_term } n$ 
```

the same LIFTER assertion



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

new types ->

datatype instr = LOADI val | LOAD vname | ADD
type_synonym stack = "val list"

the same LIFTER assertion



$\exists r1 : \text{rule}. \text{True}$

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

new types ->

```
datatype      instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
  "exec1 (LOADI n) _ stk = n # stk" |  
  "exec1 (LOAD x)  s stk = s(x) # stk" |  
  "exec1 ADD      _ (j#i#stk) = (i + j) # stk"
```

the same LIFTER assertion

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
  "exec []      stk = stk" |  
  "exec (i#is)  s stk = exec is s (exec1 i s stk)"
```



$\exists r1 : \text{rule}. \text{True}$

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
  "exec1 (LOADI n) s stk = n # stk" |  
  "exec1 (LOAD x) s stk = s(x) # stk" |  
  "exec1 ADD s (j#i#stk) = (i + j) # stk"
```

the same LIFTER assertion

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
  "exec [] s stk = stk" |  
  "exec (i#is) s stk = exec is s (exec1 i s stk)"
```

↓
new lemma -> lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
a model proof -> apply(induct is1 s stk rule:exec.induct)
 $\exists r1 : \text{rule}. \text{True}$ apply auto done

→

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> apply(induct is1 s stk rule:exec.induct)

$\exists r1 : \text{rule}. \text{True}$ apply auto done

→

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> apply(induct is1 s stk rule:exec.induct)

$\exists r1 : \text{rule. } \text{True }$ apply auto done

r1

$\exists r1 : \text{rule. }$ (r1 = exec.induct)

$\exists t1 : \text{term. }$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term. }$

$r1 \text{ is_rule_of } to1$

\wedge

$\forall t2 : \text{term} \in \text{induction_term. }$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term. }$

$\exists n : \text{number. }$

$\text{is_nth_argument_of } (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> apply(induct is1 s stk rule:exec.induct)

$\exists r1 : \text{rule}. \text{True}$ apply auto done

r1

$\exists r1 : \text{rule}.$

(r1 = exec.induct)

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> apply(induct is1 s stk rule:exec.induct)

$\exists r1 : \text{rule}. \text{True}$ apply auto done

r1

→

$\exists r1 : \text{rule}.$

(r1 = exec.induct)

(t1 = exec)

$\exists t1 : \text{term}.$

$\exists tol : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } tol$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, tol)$

\wedge

$t2 \text{ is_nth_induction_term } n$

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> apply(induct is1 s stk rule:exec.induct)

$\exists r1 : \text{rule}. \text{True}$ apply auto done

to1

r1

→

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

(r1 = exec.induct)

(t1 = exec)

(to1 = exec)

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> apply(induct is1 s stk rule:exec.induct)

$\exists r1 : \text{rule}. \text{True}$ apply auto done

to1

r1

$\exists r1 : \text{rule}.$

(r1 = exec.induct)

(t1 = exec)

(to1 = exec)

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> apply(induct is1 s stk rule:exec.induct)

$\exists r1 : \text{rule}. \text{True}$ apply auto done

to1

r1

$\rightarrow \exists r1 : \text{rule}.$

($r1 = \text{exec.induct}$)

($t1 = \text{exec}$)

$\exists t1 : \text{term}.$ ($t1 = \text{exec}$)

$r1 \text{ is_rule_of } t1$ True! $r1 (= \text{exec.induct})$ is a lemma about $t1 (= \text{exec})$.

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists t2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (t2, n, t1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> apply(induct is1 s stk rule:exec.induct)

$\exists r1 : \text{rule}. \text{True}$ apply auto done

to1

r1

$\rightarrow \exists r1 : \text{rule}.$

($r1 = \text{exec.induct}$)

($t1 = \text{exec}$)

$\exists t1 : \text{term}.$ ($t1 = \text{exec}$)

$r1 \text{ is_rule_of } t1 \quad \text{True! } r1 (= \text{exec.induct}) \text{ is a lemma about } t1 (= \text{exec}).$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists t2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (t2, n, t1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> apply(induct is1 s stk rule:exec.induct)

$\exists r1 : \text{rule}. \text{True}$ apply auto done

to1

r1

$\exists r1 : \text{rule}.$

($r1 = \text{exec.induct}$)

($t1 = \text{exec}$)

$\exists t1 : \text{term}.$ ($t1 = \text{exec}$)

$r1 \text{ is_rule_of } t1$ True! $r1 (= \text{exec.induct})$ is a lemma about $t1 (= \text{exec})$.



$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists t2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (t2, n, t1)$



$t2 \text{ is_nth_induction_term } n$

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> apply(induct is1 s stk rule:exec.induct)

$\exists r1 : \text{rule}. \text{True}$ apply auto done

to1

r1

$\exists r1 : \text{rule}.$

($r1 = \text{exec.induct}$)

($t1 = \text{exec}$)

$\exists t1 : \text{term}.$ ($t1 = \text{exec}$)

$r1 \text{ is_rule_of } t1$ True! $r1 (= \text{exec.induct})$ is a lemma about $t1 (= \text{exec})$.

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists t2 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

is_nth_argument_of (t2, n, to1)

\wedge

t2 is_nth_induction_term n

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> apply(induct is1 s stk rule:exec.induct)

$\exists r1 : \text{rule}. \text{True}$ apply auto done

t₂

t₀₁

r₁

$\exists r1 : \text{rule}.$

(r₁ = exec.induct)

(t₁ = exec)

$\exists t01 : \text{term_occurrence} \in t1 : \text{term}.$ (t₀₁ = exec)

r₁ is_rule_of t₀₁ True! r₁ (= exec.induct) is a lemma about t₀₁ (= exec).

^

$\forall t2 : \text{term} \in \text{induction_term}.$

(t₂ = is1, s, and stk)

$\exists t02 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

is_nth_argument_of (t₀₂, n, t₀₁)

^

t₂ is_nth_induction_term n

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> **lemma** "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
a model proof -> **apply(induct is1 s stk rule:exec.induct)**

$\exists r1 : \text{rule}. \text{True}$ **apply auto done**

→

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}. \quad (to1 = \text{exec})$

$r1 \text{ is_rule_of } to1 \quad \text{True! } r1 (= \text{exec.induct}) \text{ is a lemma about } to1 (= \text{exec}).$

^

$\forall t2 : \text{term} \in \text{induction_term}. \quad (t2 = \text{is1, s, and stk})$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}. \quad (to2 = \text{is1, s, and stk})$

$\exists n : \text{number}.$

$\text{is_nth_argument_of } (to2, n, to1)$

^

$t2 \text{ is_nth_induction_term } n$

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> **lemma** "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> **apply(induct is1 s stk rule:exec.induct)**

$\exists r1 : \text{rule}. \text{True}$ **apply auto done**



$\rightarrow \exists r1 : \text{rule}.$

($r1 = \text{exec.induct}$)

($t1 = \text{exec}$)

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}. (to1 = \text{exec})$

$r1 \text{ is_rule_of } to1$ True! $r1 (= \text{exec.induct})$ is a lemma about $to1 (= \text{exec})$.



$\forall t2 : \text{term} \in \text{induction_term}. (t2 = \text{is1, s, and stk})$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}. (to2 = \text{is1, s, and stk})$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$



$t2 \text{ is_nth_induction_term } n$

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> **lemma** "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> **apply(induct is1 s stk rule:exec.induct)**

$\exists r1 : \text{rule}. \text{True}$ **apply auto done**



$\rightarrow \exists r1 : \text{rule}.$

($r1 = \text{exec.induct}$)

($t1 = \text{exec}$)

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}. (to1 = \text{exec})$

$r1 \text{ is_rule_of } to1$ True! $r1 (= \text{exec.induct})$ is a lemma about $to1 (= \text{exec})$.



$\wedge \forall t2 : \text{term} \in \text{induction_term}.$

($t2 = \text{is1, s, and stk}$)

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}. (to2 = \text{is1, s, and stk})$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (to2, n, to1)$



$t2 \text{ is_nth_induction_term } n$

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> **lemma** "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> **apply(induct is1 s stk rule:exec.induct)**

$\exists r1 : \text{rule}. \text{True}$ **apply auto done**

$\rightarrow \exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$ $(t1 = \text{exec.induct})$

$r1 \text{ is_rule_of } to1$ True! $r1 (= \text{exec.induct})$ is a lemma about $to1 (= \text{exec})$.



$\wedge \forall t2 : \text{term} \in \text{induction_term}.$ $(t2 = \text{is1, s, and stk})$

$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$ $(to2 = \text{is1, s, and stk})$

$\exists n : \text{number}.$

$\text{is_nth_argument_of } (to2, n, to1)$ when $t2$ is is1 ($n \rightarrow 1$) ?



$t2 \text{ is_nth_induction_term } n$

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> **lemma** "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> **apply(induct is1 s stk rule:exec.induct)**

$\exists r1 : \text{rule}. \text{True}$

apply auto done



$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists t01 : \text{term_occurrence} \in t1 : \text{term}. \quad (t01 = \text{exec})$

$r1 \text{ is_rule_of } t01 \quad \text{True! } r1 (= \text{exec.induct}) \text{ is a lemma about } t01 (= \text{exec}).$



$\forall t2 : \text{term} \in \text{induction_term}.$

$\exists t02 : \text{term_occurrence} \in t2 : \text{term}. \quad (t02 = \text{is1, s, and stk})$

$\exists n : \text{number}.$

$\text{is_nth_argument_of } (t02, n, t01) \text{ when } t2 \text{ is } \text{is1} (n \rightarrow 1)$



$t2 \text{ is_nth_induction_term } n$

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> **lemma** "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> **apply(induct is1 s stk rule:exec.induct)**

$\exists r1 : \text{rule}. \text{True}$

apply auto done

r1

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

first
second (r1 = exec.induct)

$\exists t01 : \text{term_occurrence} \in t1 : \text{term}. (t01 = \text{exec})$

(t1 = exec)

$r1 \text{ is_rule_of } t01$ True! r1 (= exec.induct) is a lemma about t01 (= exec).



$\forall t2 : \text{term} \in \text{induction_term}.$

(t2 = is1, s, and stk)

$\exists t02 : \text{term_occurrence} \in t2 : \text{term}. (t02 = \text{is1, s, and stk})$

(t02 = is1, s, and stk)

$\exists n : \text{number}.$

is_nth_argument_of (t02, n, t01)

when t2 is is1 (n → 1)



t2 is_nth_induction_term n

when t2 is s (n → 2)?

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> **lemma** "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> **apply(induct is1 s stk rule:exec.induct)**

$\exists r1 : \text{rule}. \text{True}$

apply auto done

r1

$\exists r1 : \text{rule}.$

first
second (r1 = exec.induct)

$\exists t1 : \text{term}.$

(t1 = exec)

$\exists t01 : \text{term_occurrence} \in t1 : \text{term}. (t01 = exec)$

r1 is_rule_of t01 True! r1 (= exec.induct) is a lemma about t01 (= exec).

^ 

$\forall t2 : \text{term} \in \text{induction_term}.$

(t2 = is1, s, and stk)

$\exists t02 : \text{term_occurrence} \in t2 : \text{term}. (t02 = is1, s, and stk)$

(t02 = is1, s, and stk)

$\exists n : \text{number}.$

is_nth_argument_of (t02, n, t01)

when t2 is is1 (n -> 1) 

^

t2 is_nth_induction_term n

when t2 is s (n -> 2) 

new types \rightarrow

```
datatype      instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants \rightarrow

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where
  "exec1 (LOADI n) _ stk = n # stk" |
  "exec1 (LOAD x)  s stk = s(x) # stk" |
  "exec1 ADD      (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where
  "exec []      _ stk = stk" | first
  "exec (i#is) s stk = exec is s (exec i s stk)" second
                                         | third
                                         | t2
                                         | t01
lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
```

new lemma \rightarrow **lemma** "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof \rightarrow **apply(induct is1 s stk rule:exec.induct)**

$\exists r1 : \text{rule}. \text{ True}$

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists \text{tol} : \text{term_occurrence} \in t1 : \text{term}. \quad (\text{tol} = \text{exec})$

`r1 is_rule_of tol` True! r1 (= exec.induct) is a lemma about tol (= exec).

8

$\forall t2 : \text{term} \in \text{induction_term}.$

(t2 = is1, s, and stk)

$\exists \text{ } to2 : \text{term_occurrence} \in t2 : \text{term}.$

(to2 = is1, s, and stk)

$\exists n : \text{number}.$

is_nth_argument_of (to2, n, to1)

3

t2 is_nth_induction_term *n*

when t_2 is is_1 ($n \rightarrow 1$)

when t_2 is s ($n \rightarrow 2$)

when t_2 is stk ($n \rightarrow 3$) ?

new types →

```
datatype      instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants \rightarrow

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where
  "exec1 (LOADI n) _ stk = n # stk" |
  "exec1 (LOAD x)  s stk = s(x) # stk" |
  "exec1 ADD      (j#i#stk) = (i + j) # stk"
```

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where
  "exec []      _ stk = stk" | first
  "exec (i#is) s stk = exec is s (exec i s stk)" second
                                         | third
                                         | t2
                                         | t01
lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
```

~~new lemma \rightarrow lemma "exec (is1 @ is2) s\ stk = exec is2 s\ (exec is1 s\ stk)"~~

a model proof \rightarrow **apply(induct is1 s stk rule:exec.induct)**

$\exists r1 : \text{rule}. \text{ True}$

→

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists \text{tol} : \text{term_occurrence} \in t1 : \text{term}. \quad (\text{tol} = \text{exec})$

`r1 is_rule_of t01` True! `r1` (`= exec.induct`) is a lemma about `t01` (`= exec`).

8

$\forall t2 : \text{term} \in \text{induction_term}.$

(12 11 10 9 8 7 6 5 4 3 2 1)

$\exists \text{ } to2 : \text{term occurrence} \in t2 : \text{term}.$

(to2 = is1, s, and stk)

$\exists n : \text{number}.$

is_nth_argument_of (to2, n, to1)

when t_2 is i_1 ($n \rightarrow 1$)

1

t2 is_nth_induction_term *n*

when k^2 is still ($n = 3$)

new types ->

```
datatype instr = LOADI val | LOAD vname | ADD  
type_synonym stack = "val list"
```

new constants ->

```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) _ s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

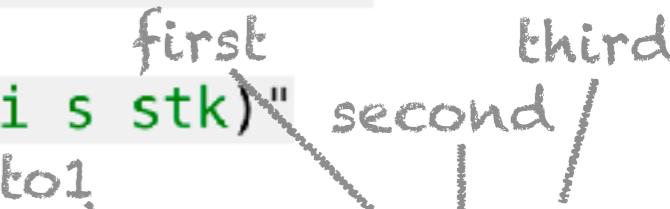
```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

new lemma -> **lemma** "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> **apply(induct is1 s stk rule:exec.induct)**

$\exists r1 : \text{rule}. \text{True}$

apply auto done



→

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists t01 : \text{term_occurrence} \in t1 : \text{term}. (t01 = \text{exec})$

$r1 \text{ is_rule_of } t01$ True! $r1 (= \text{exec.induct})$ is a lemma about $t01 (= \text{exec})$.

^

$\forall t2 : \text{term} \in \text{induction_term}.$

($t2 = \text{is1, s, and stk}$)
($t02 = \text{is1, s, and stk}$)

$\exists t02 : \text{term_occurrence} \in t2 : \text{term}.$

$\exists n : \text{number}.$

$\text{is_nth_argument_of} (t02, n, t01)$

when $t2$ is is1 ($n \rightarrow 1$)

^

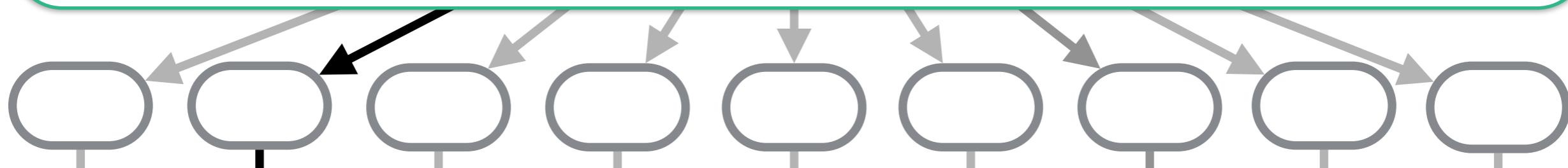
$t2 \text{ is_nth_induction_term } n$

when $t2$ is s ($n \rightarrow 2$)

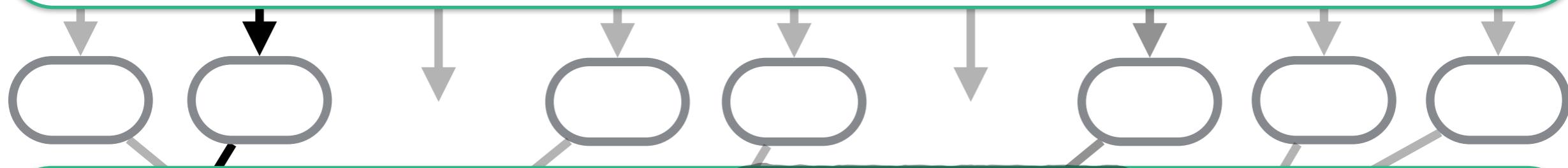
when $t2$ is stk ($n \rightarrow 3$)

goal

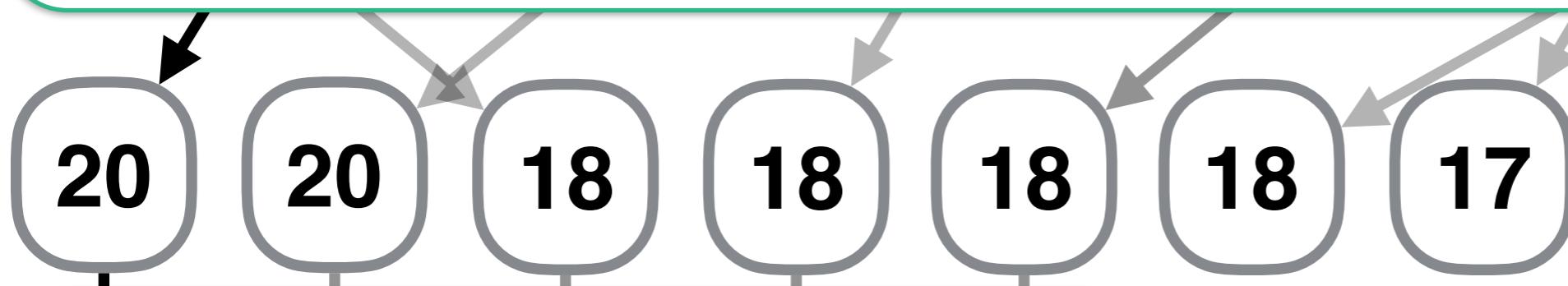
Step 1: creating many inductions



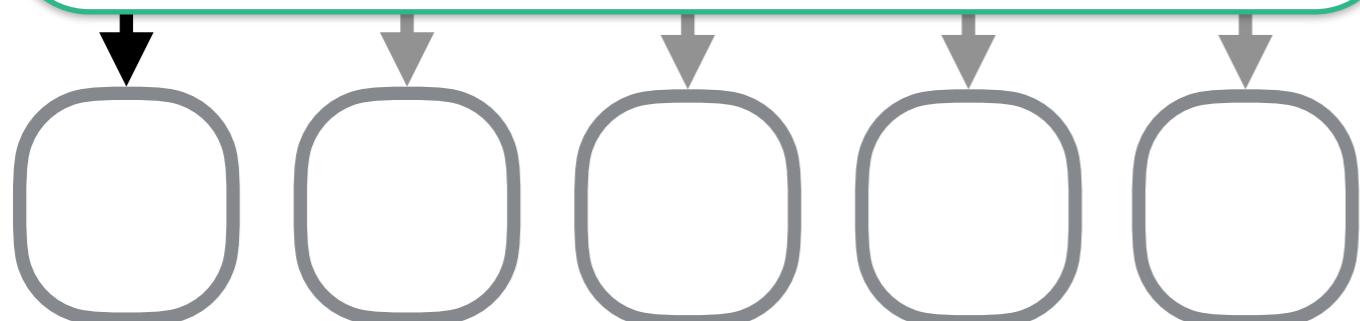
Step 2: multi-stage screening



Step 3: scoring using **20 heuristics** and sorting



Step 4: short-listing



DEMO!

The screenshot shows the Isabelle/HOL proof assistant interface. The top half displays a theory file named `Induction_Demo.thy` with the following content:

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

lemma "itrev xs ys = rev xs @ ys" smart_induct
apply(induction xs arbitrary: ys)
apply(auto)
```

The line `apply(induction xs arbitrary: ys)` is highlighted with a yellow background. The bottom half shows the proof state:

```
proof (prove)
goal (1 subgoal):
  1. itrev xs ys = rev xs @ ys
```

The interface includes a toolbar at the top, a vertical navigation bar on the left, and a status bar at the bottom.

The screenshot shows the Isabelle/HOL proof assistant interface. The top part displays a theory file named `Induction_Demo.thy` with the following code:

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

lemma "itrev xs ys = rev xs @ ys" smart_induct
apply(induction xs arbitrary: ys)
apply(auto)
```

smart_induct started producing combinations of induction arguments.
smart_induct produced 40 combinations of arguments for the induct method.
... out of which 32 of them return some results.
... out of which only 28 of them passes the second screening stage.
LiFtEr assertions are evaluating the first 28 of them.
Try these 10 most promising inductions!

1st candidate is apply (induct xs arbitrary: ys)
(* The score is 20 out of 20. *)
2nd candidate is apply (induct xs)
(* The score is 20 out of 20. *)
3th candidate is apply (induct xs ys rule: Induction_Demo.itrev.induct)
(* The score is 18 out of 20. *)

```
File Browser Documentation Sidekick State Theories  
Induction_Demo.thy (~/Workplace/PSL/Smart_Induct/Example/)  
14 fun itrev :: "'a list ⇒ 'a list ⇒ 'a list" where  
15 "itrev [] ys = ys" |  
16 "itrev (x#xs) ys = itrev xs (x#ys)"  
17  
18 lemma "itrev xs ys = rev xs @ ys" smart_induct  
19 apply(induction xs arbitrary: ys)  
20 apply(auto)
```

smart_induct started producing combinations of induction arguments.
smart_induct produced 40 combinations of arguments for the induct method.
... out of which 32 of them return some results.
... out of which only 28 of them passes the second screening stage.
LiFtEr assertions are evaluating the first 28 of them.
Try these 10 most promising inductions!
1st candidate is apply (induct xs arbitrary: ys)
(* The score is 20 out of 20. *)
2nd candidate is apply (induct xs)
(* The score is 20 out of 20. *)
3th candidate is apply (induct xs ys rule: Induction_Demo.itrev.induct)
(* The score is 18 out of 20. *)

The screenshot shows the Isabelle/Isar interface. The top part displays a theory file named `Induction_Demo.thy` with the following content:

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

lemma "itrev xs ys = rev xs @ ys" smart_induct
apply(induction xs arbitrary: ys)
apply(auto)
```

smart_induct started producing combinations of induction arguments.
smart_induct produced 40 combinations of arguments for the induct method.
... out of which 32 of them return some results.
... out of which only 28 of them passes the second screening stage.
LiFtEr assertions are evaluating the first 28 of them.
Try these 10 most promising inductions!

1st candidate is apply (induct xs arbitrary: ys)
(* The score is 20 out of 20. *)
2nd candidate is apply (induct xs)
(* The score is 20 out of 20. *)
3th candidate is apply (induct xs ys rule: Induction_Demo.itrev.induct)
(* The score is 18 out of 20. *)

The screenshot shows the Isabelle/Isar interface. The top part displays a theory file named `Induction_Demo.thy` with the following content:

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

lemma "itrev xs ys = rev xs @ ys" smart_induct
apply(induction xs arbitrary: ys)
apply(auto)
```

The word `smart_induct` is highlighted in red, indicating an error or a placeholder. The line `apply(induction xs arbitrary: ys)` is highlighted in yellow.

smart_induct started producing combinations of induction arguments.
smart_induct produced 40 combinations of arguments for the induct method.
... out of which 32 of them return some results.
... out of which only 28 of them passes the second screening stage.
LiFtEr assertions are evaluating the first 28 of them.
Try these 10 most promising inductions!

1st candidate is apply (induct xs arbitrary: ys)
(* The score is 20 out of 20. *)
2nd candidate is apply (induct xs)
(* The score is 20 out of 20. *)
3th candidate is apply (induct xs ys rule: Induction_Demo.itrev.induct)
(* The score is 18 out of 20. *)

smart_induct... does it work?

Table 1: Coincidence Rates of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

smart_induct... does it work?

Table 1: **Coincidence Rates** of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

smart_induct... does it work?

Table 1: **Coincidence Rates** of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

smart_induct... does it work?

Table 1: **Coincidence Rates** of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

Table 2: Coincidence Rates of `smart_induct` Based Only on Induction Terms.

theory	total	top_1	top_3	top_5	top_10
Nearest_Neighbors	16	5 (31%)	12 (75%)	15 (94%)	15 (94%)

smart_induct... does it work?

Table 1: Coincidence Rates of smart_induct.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

Table 2: Coincidence Rates of smart_induct Based Only on Induction Terms.

theory	total	top_1	top_3	top_5	top_10
Nearest_Neighbors	16	5 (31%)	12 (75%)	15 (94%)	15 (94%)

smart_induct... does it work?

Table 1: Coincidence Rates of smart_induct.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

Table 2: Coincidence Rates of smart_induct Based Only on Induction Terms.

theory	total	top_1	top_3	top_5	top_10
Nearest_Neighbors	16	5 (31%)	12 (75%)	15 (94%)	15 (94%)

Overall, the results are good!

But finding out what variables to generalise remains as a challenge!

smart_induct... does it work?

Table 1: Coincidence Rates of smart_induct.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

Table 2: Coincidence Rates of smart_induct Based Only on Induction Terms.

theory	total	top_1	top_3	top_5	top_10
Nearest_Neighbors	16	5 (31%)	12 (75%)	15 (94%)	15 (94%)

Overall, the results are good!

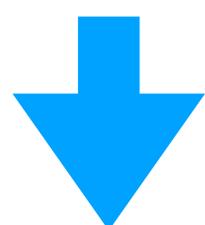
But finding out what variables to generalise remains as a challenge!

WIP!

Today I talked about...

2013 ~ 2017

Intern &
Engineer



PhD in
AI for theorem proving

2017 ~ 2018



2018 ~
2020/21



with Prof. Gerwin

Cogent



with Prof. Cezary Kaliszyk

PSL



with Dr. Josef Urban

PaMpeR

LiFtEr

smart_induct

Today I talked about...

2013 ~ 2017

Intern &
Engineer



PhD in
AI for theorem proving



2017 ~ 2018



with Prof. Gerwin



with Prof. Cezary Kaliszyk

2018 ~
2020/21



with Dr. Josef Urban

Cogent

ASPLOS2016

ITP2016

ICFP2016

PSL

CADE2017

CICM2018

PaMpeR

ASE2018

LiFtEr

APLAS2019

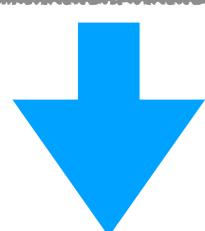
smart_indu

under review at
IJCAR2020

Today I talked about... *fin.*

2013 ~ 2017

Intern &
Engineer



PhD in
AI for theorem proving



2017 ~ 2018



with Prof. Gerwin

Cogent

ASPLOS2016

ITP2016

ICFP2016

2018 ~
2020/21



with Prof. Cezary Kaliszyk



with Dr. Josef Urban

PSL

CADE2017

CICM2018

PaMpeR

ASE2018

LiFtEr

APLAS2019

smart_indu

under review at
IJCAR2020

backup slides for Q&A

\leftarrow simple representation

```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs arbitrary: ys) auto
```



```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

<- simple representation



```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs arbitrary: ys) auto
```

```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

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

<- relevant definitions

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

\leftarrow simple representation

```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs arbitrary: ys) auto
```

```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

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

\leftarrow relevant definitions

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

\leftarrow simple representation

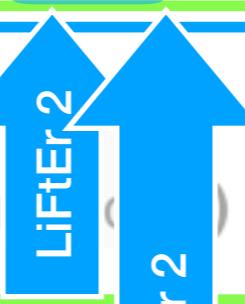
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs arbitrary: ys) auto
```



```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.ind") auto
```

\leftarrow nested assertions to examine the "semantics" of constants (rev and itrev)

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

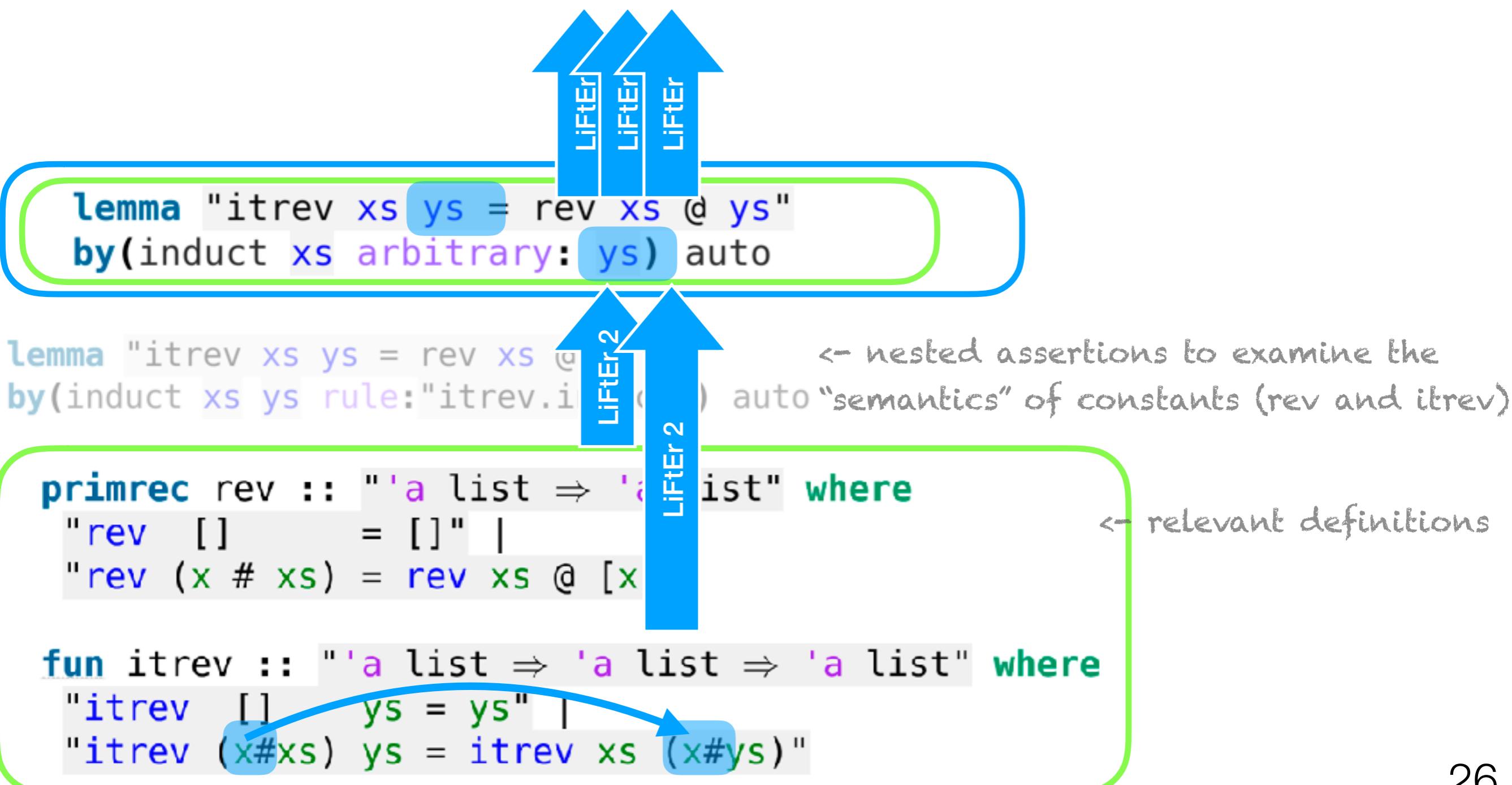


```
fun itrev :: "'a list ⇒ 'a list ⇒ 'a list" where  
"itrev [] ys = ys" |  
"itrev (x#xs) ys = itrev xs (x#ys)"
```

\leftarrow relevant definitions

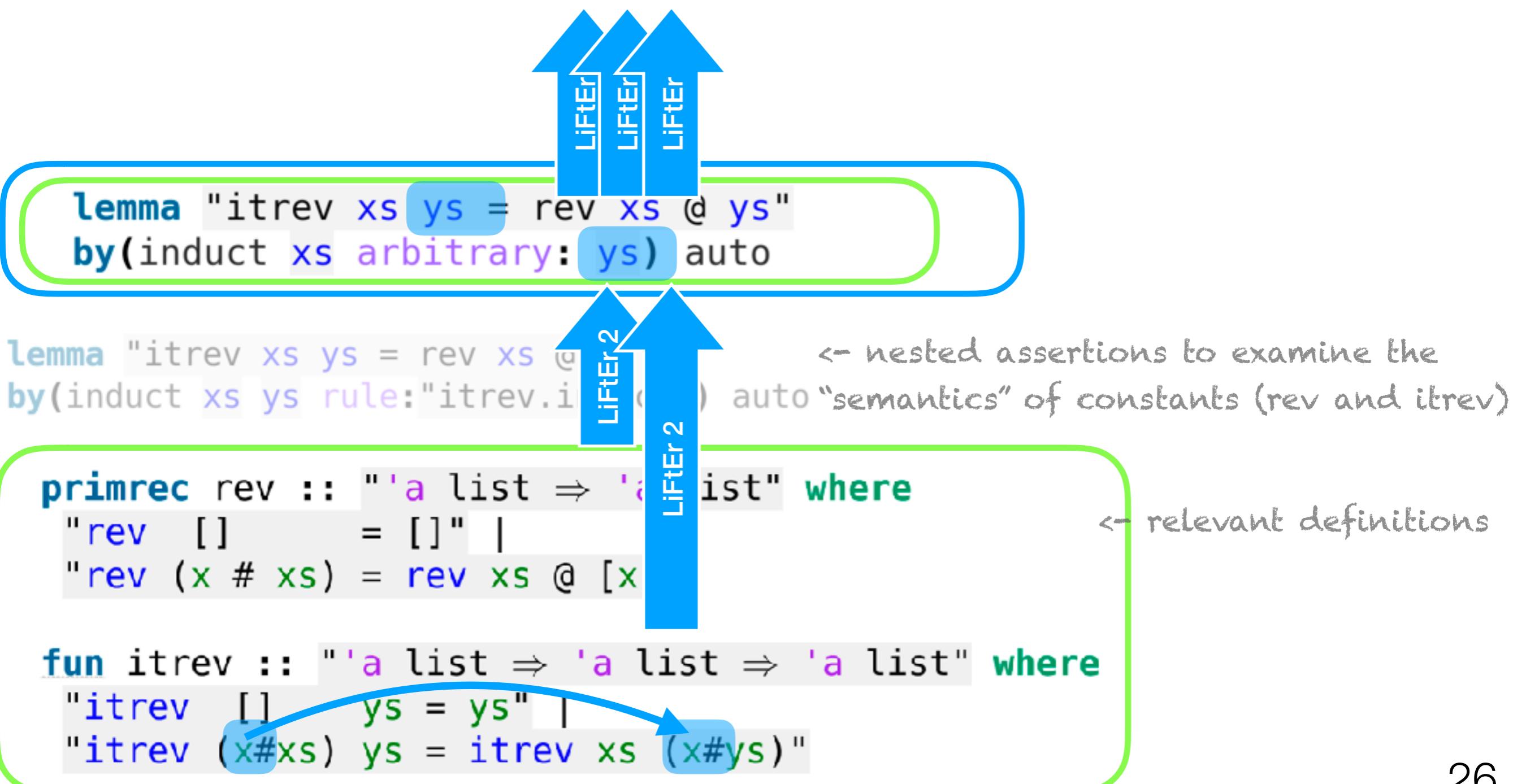
LiFtEr: (proof goal * induction arguments) -> bool

<- simple representation



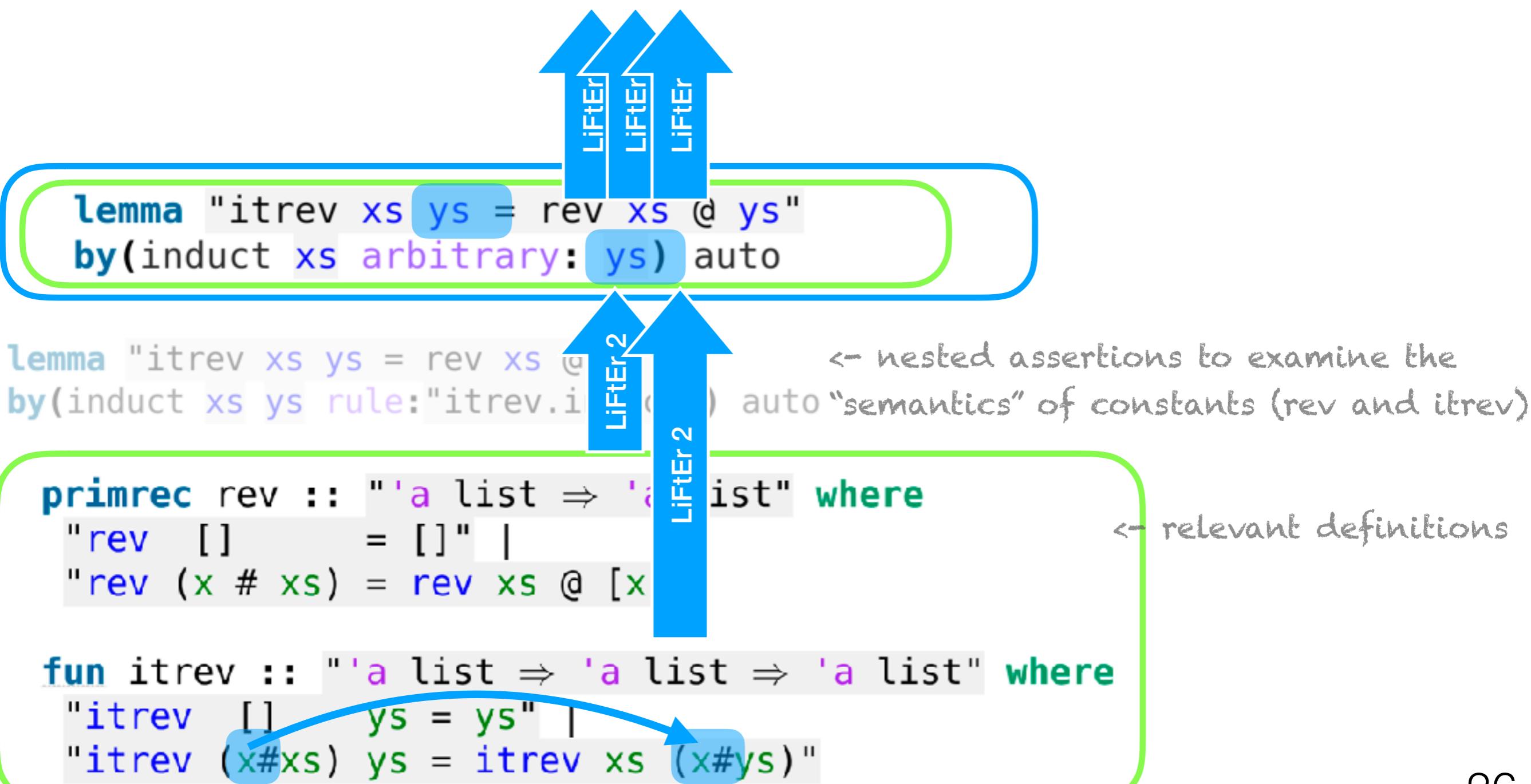
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

<- simple representation



LiFtEr₂ (proof goal * induction arguments) -> bool * relevant definitions

coming soon

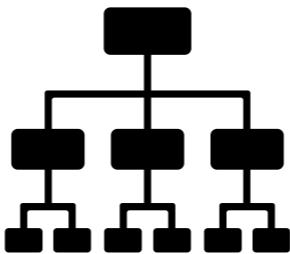


LiFtEr: (proof goal * induction arguments) -> bool

LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

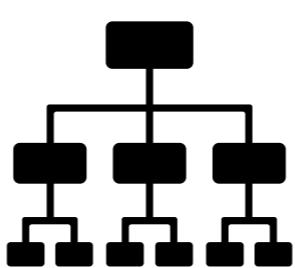
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

proof goal

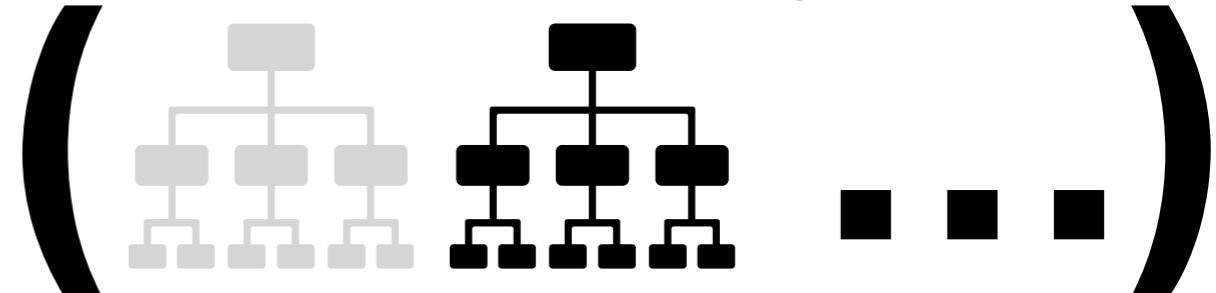


LiFtEr₂ (proof goal * induction arguments)-> bool
* relevant definitions

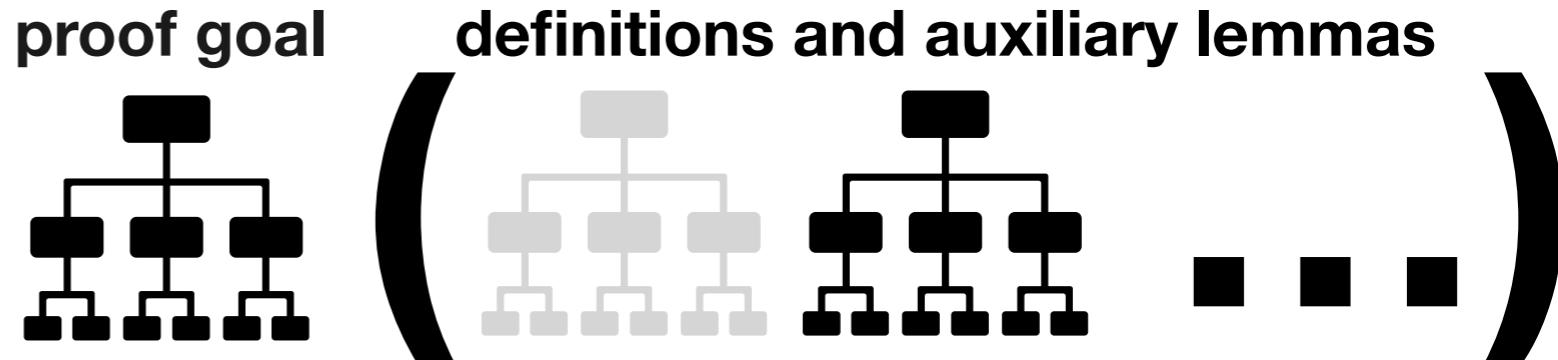
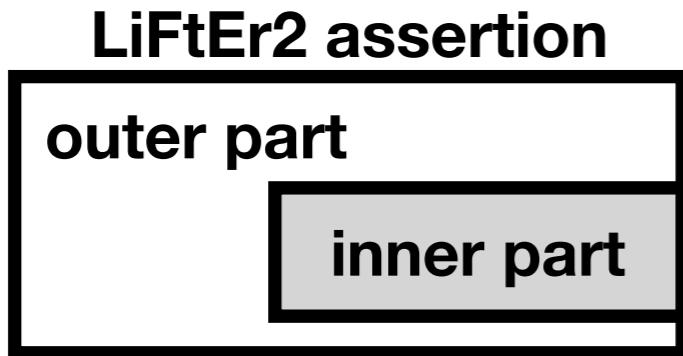
proof goal



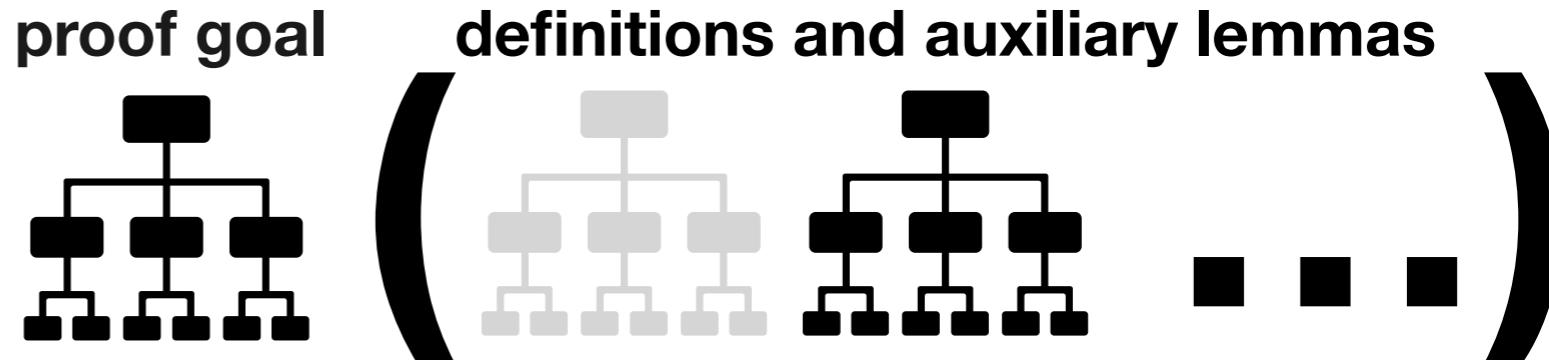
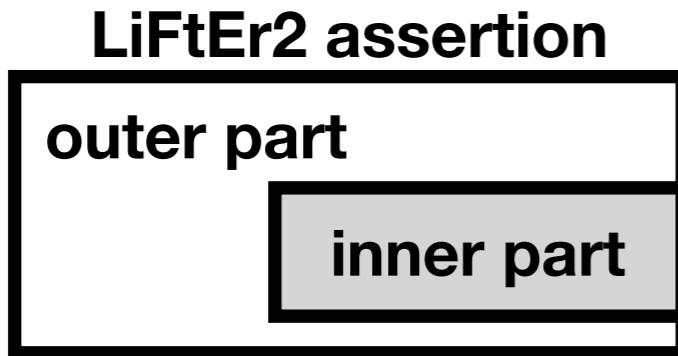
definitions and auxiliary lemmas



LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

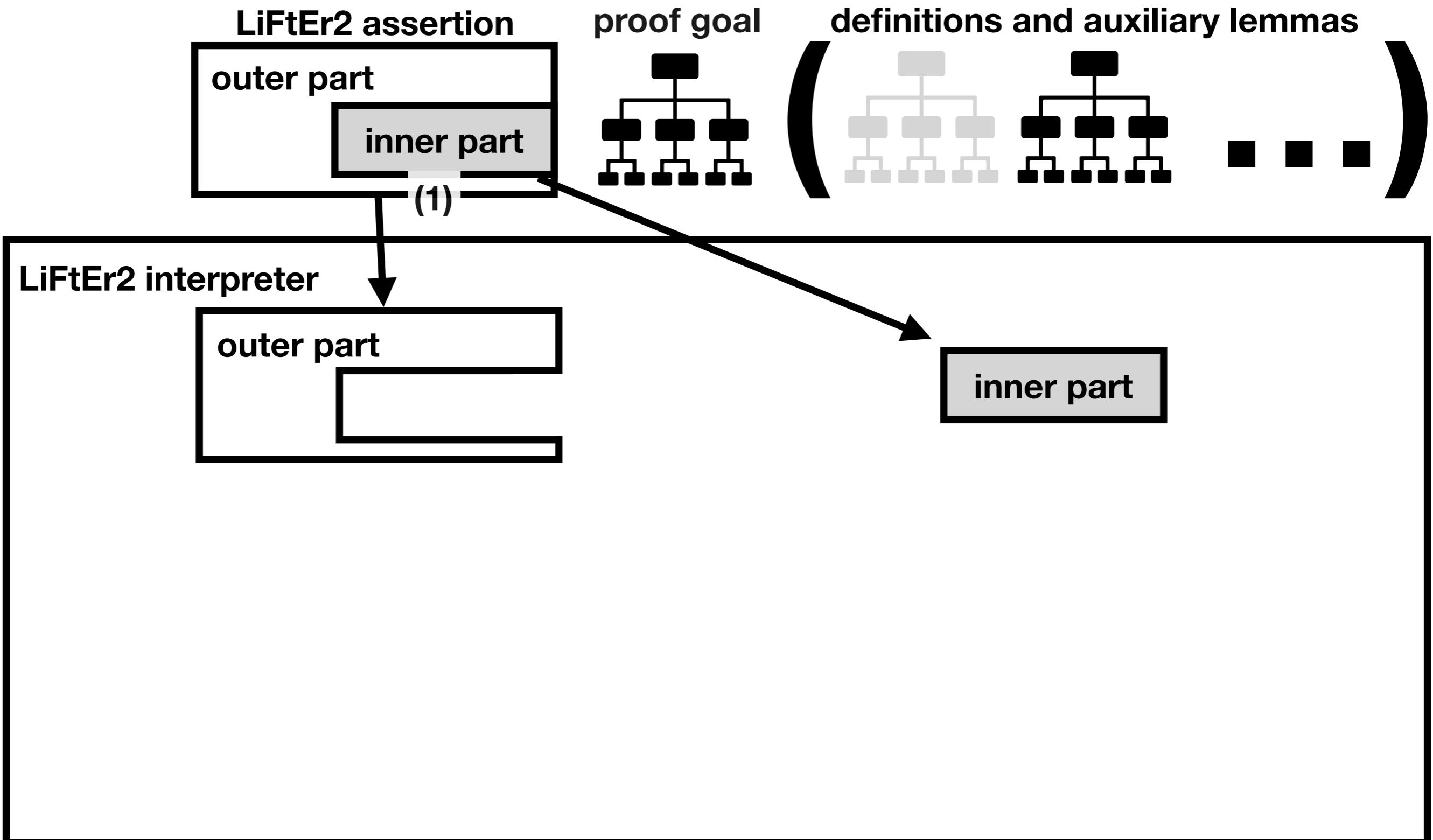


LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

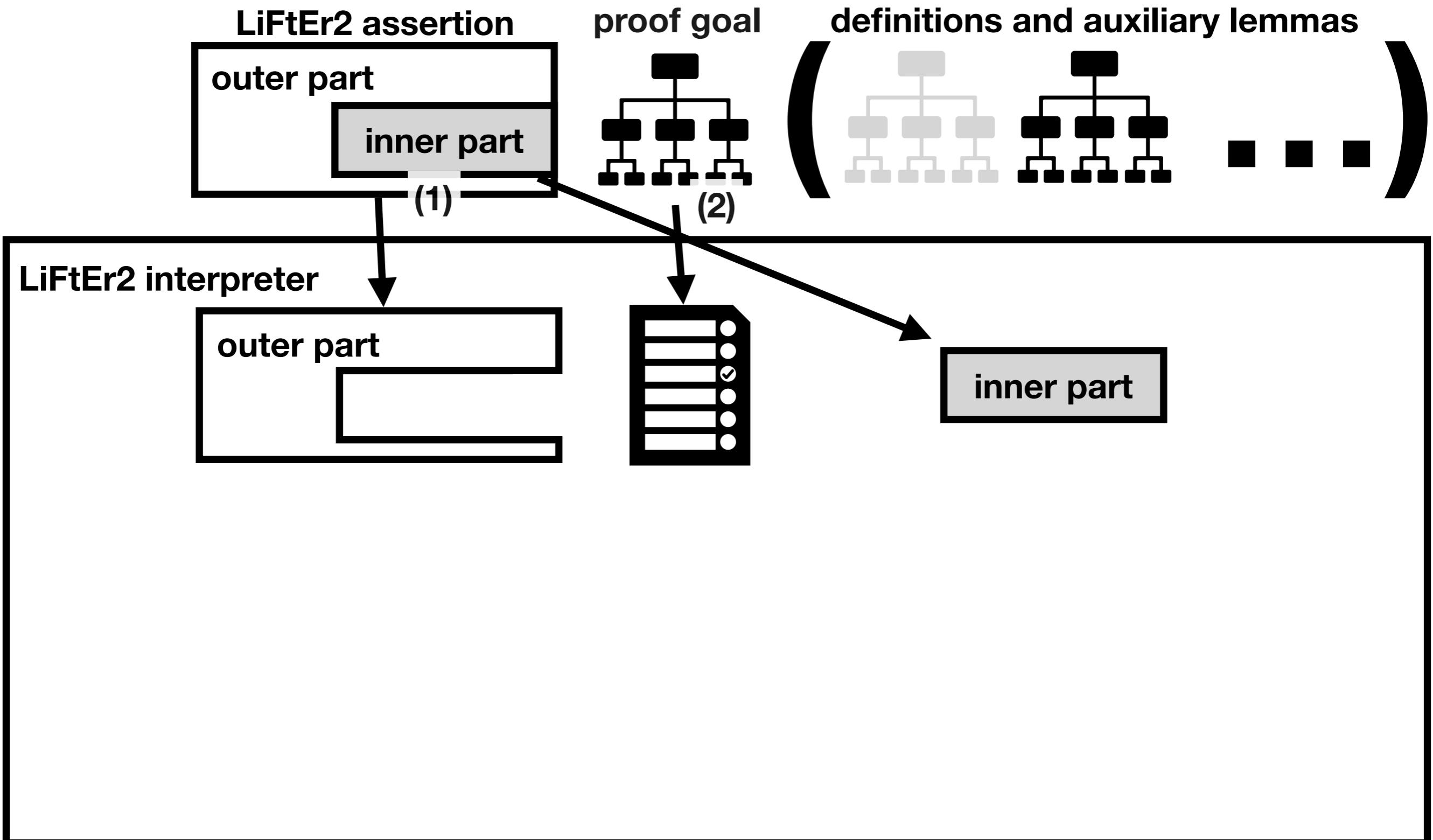


LiFtEr2 interpreter

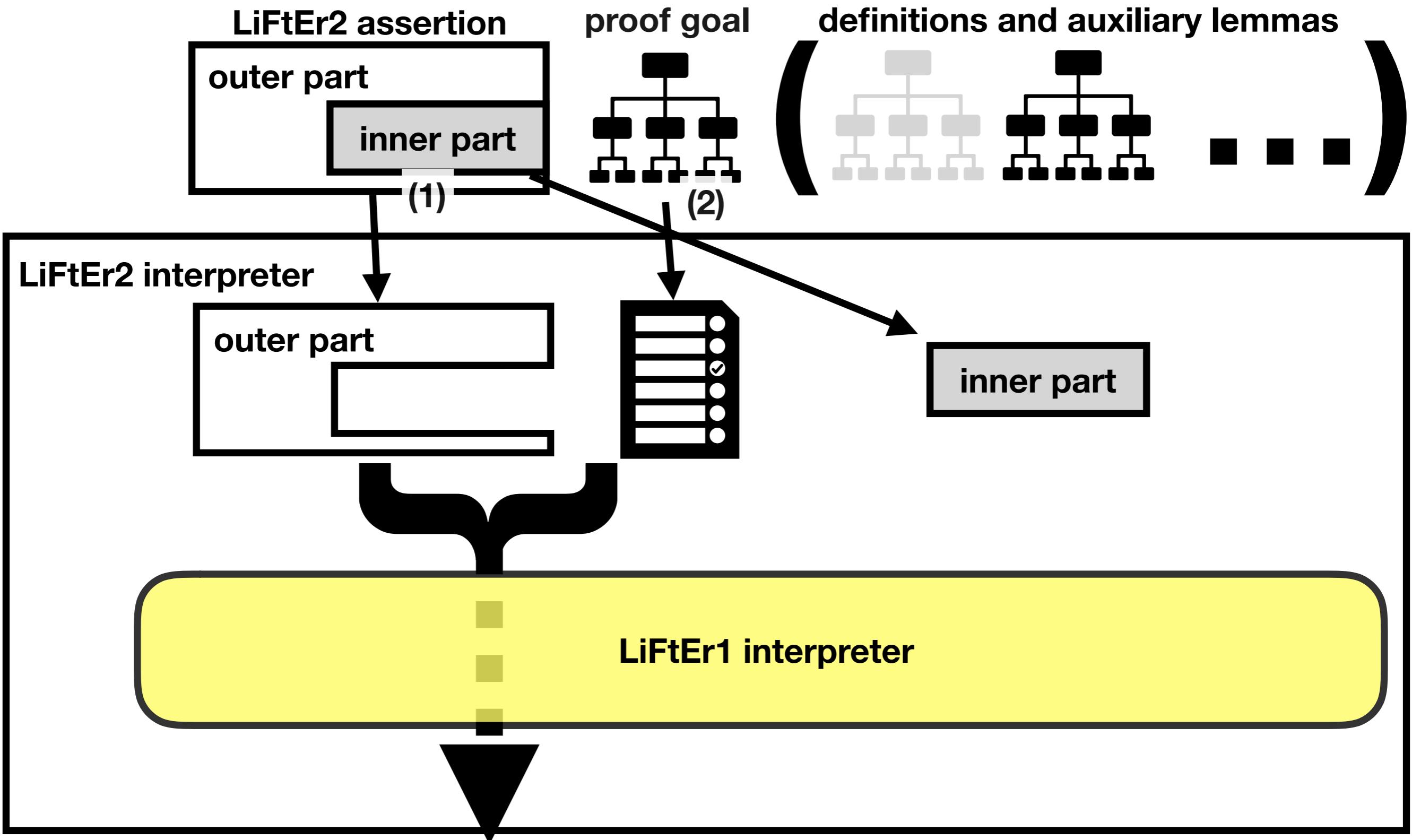
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



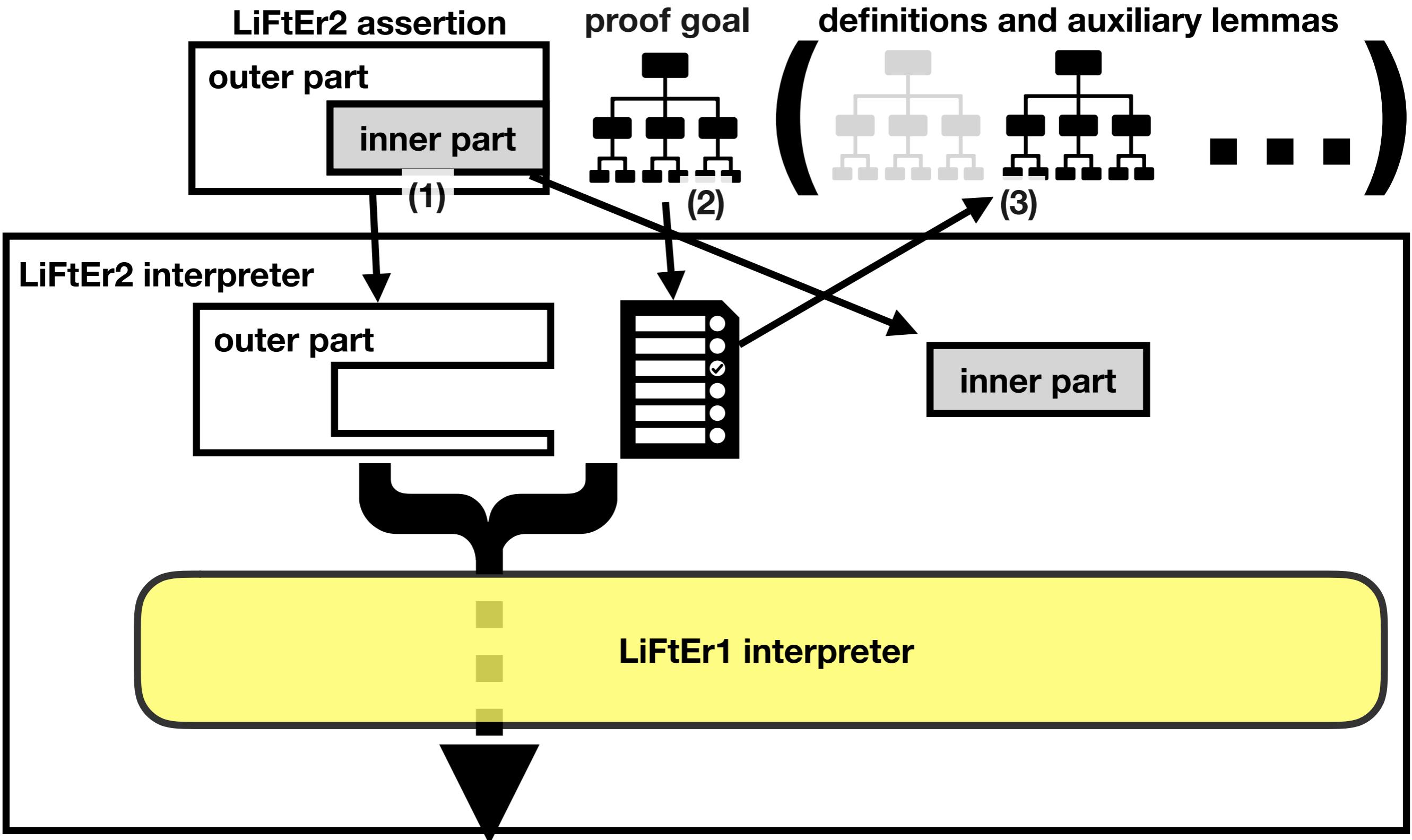
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



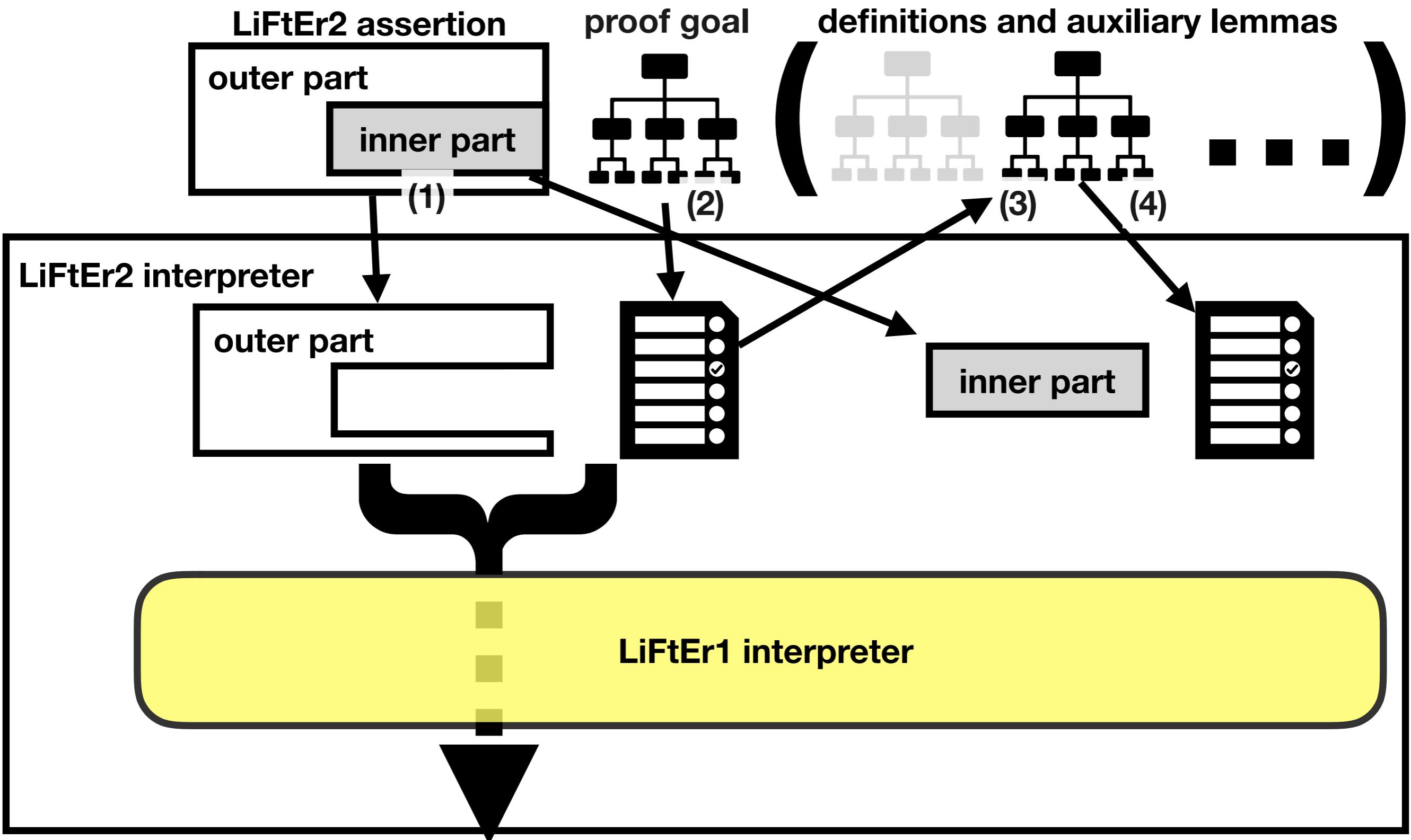
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



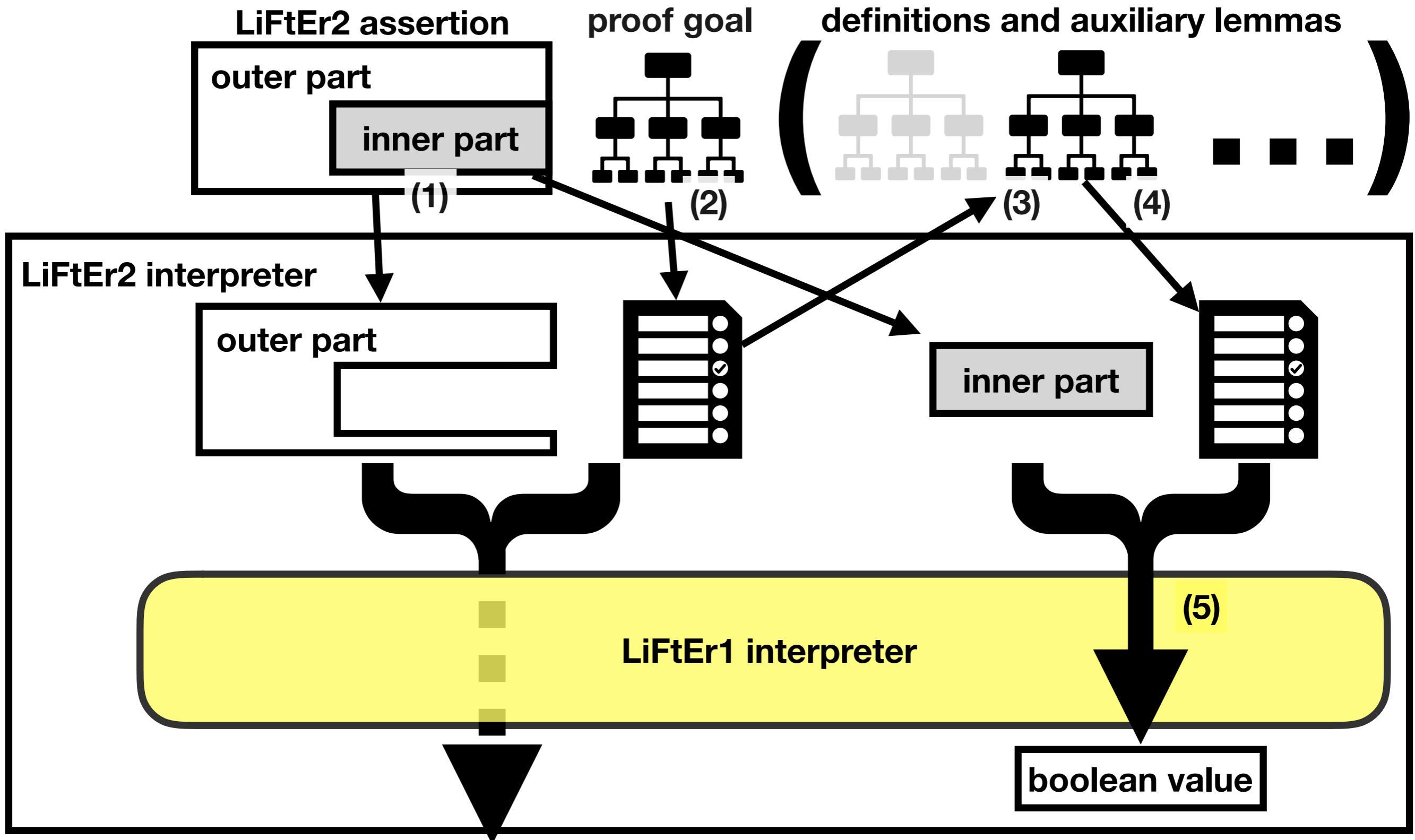
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



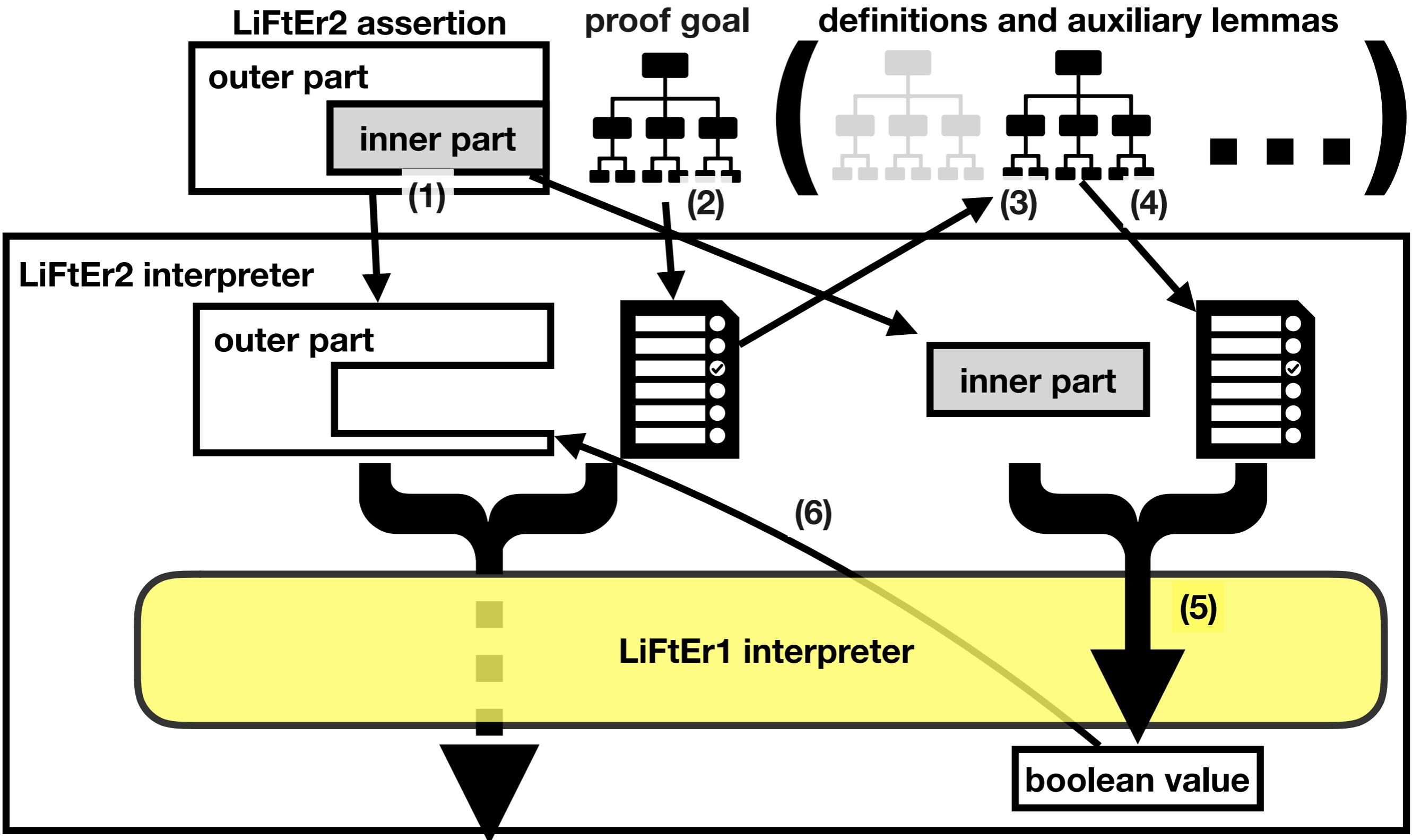
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



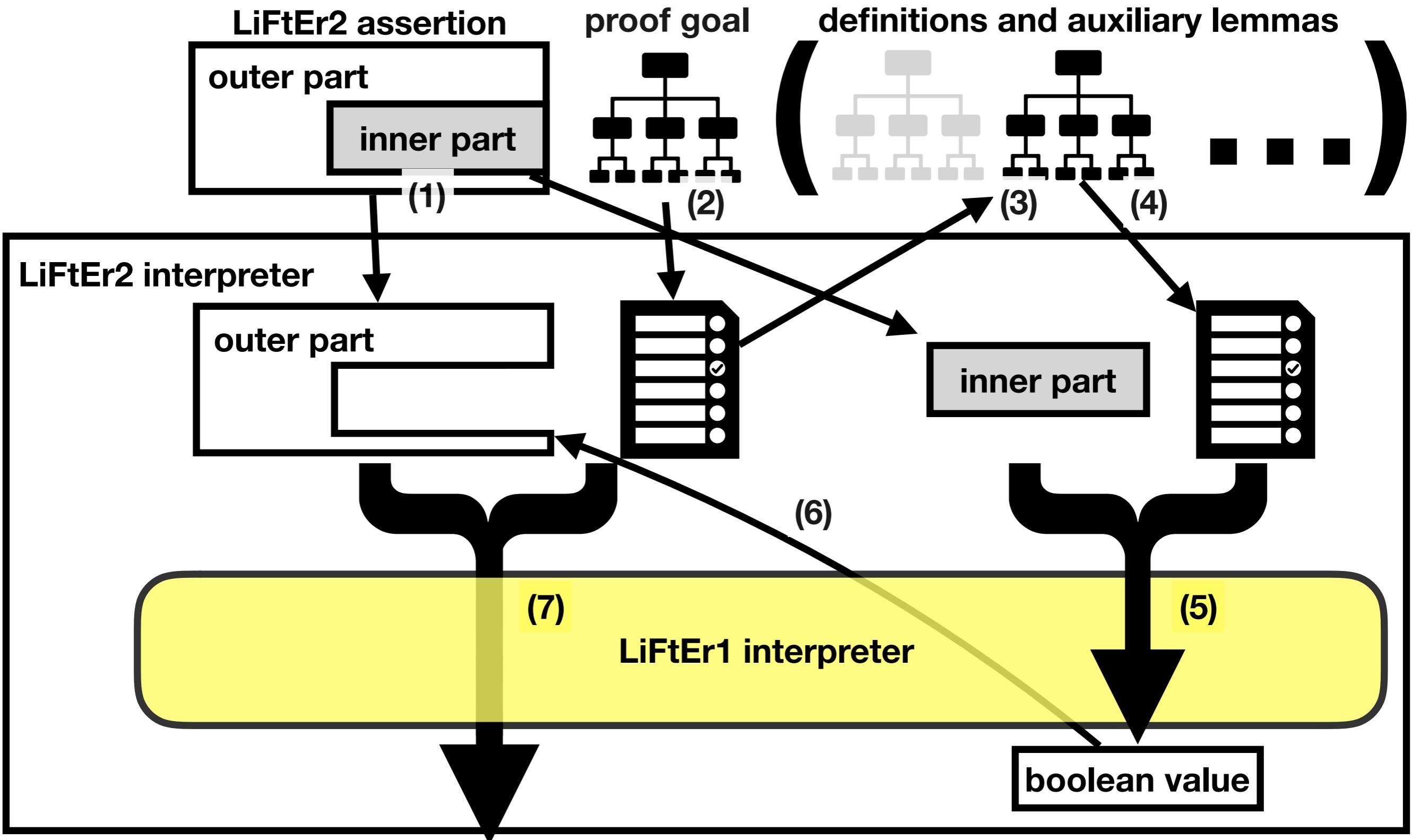
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



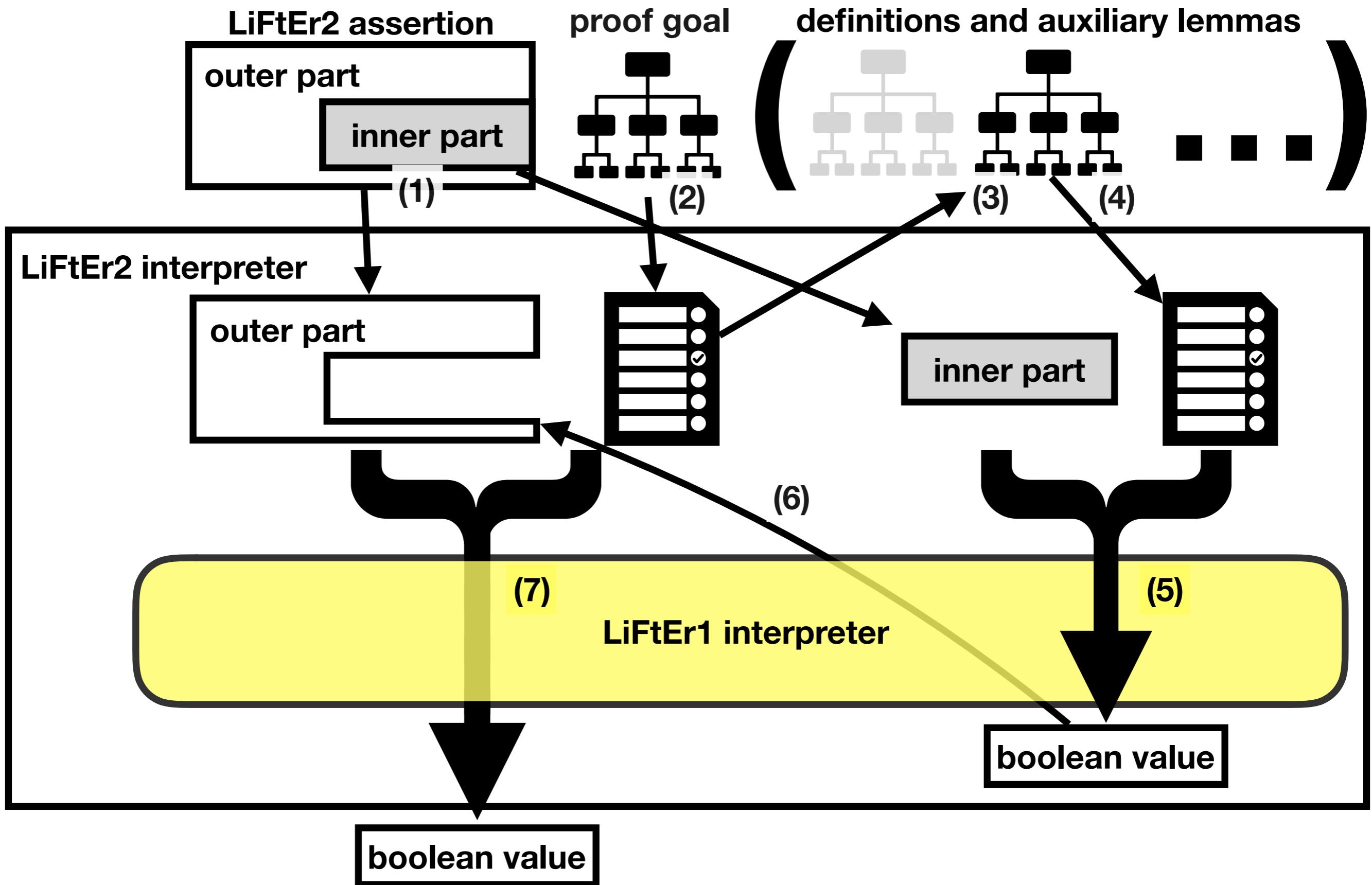
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

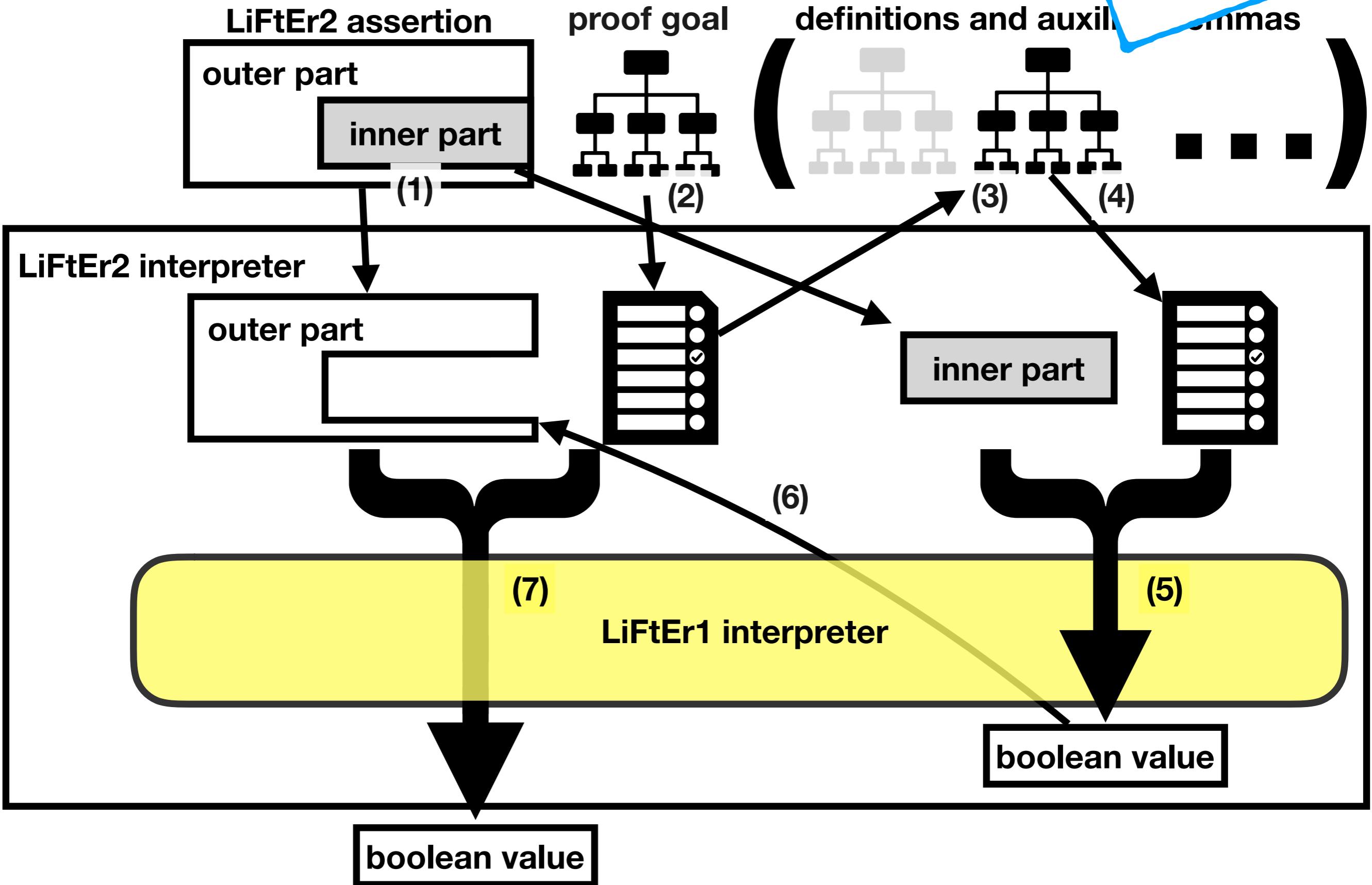


LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

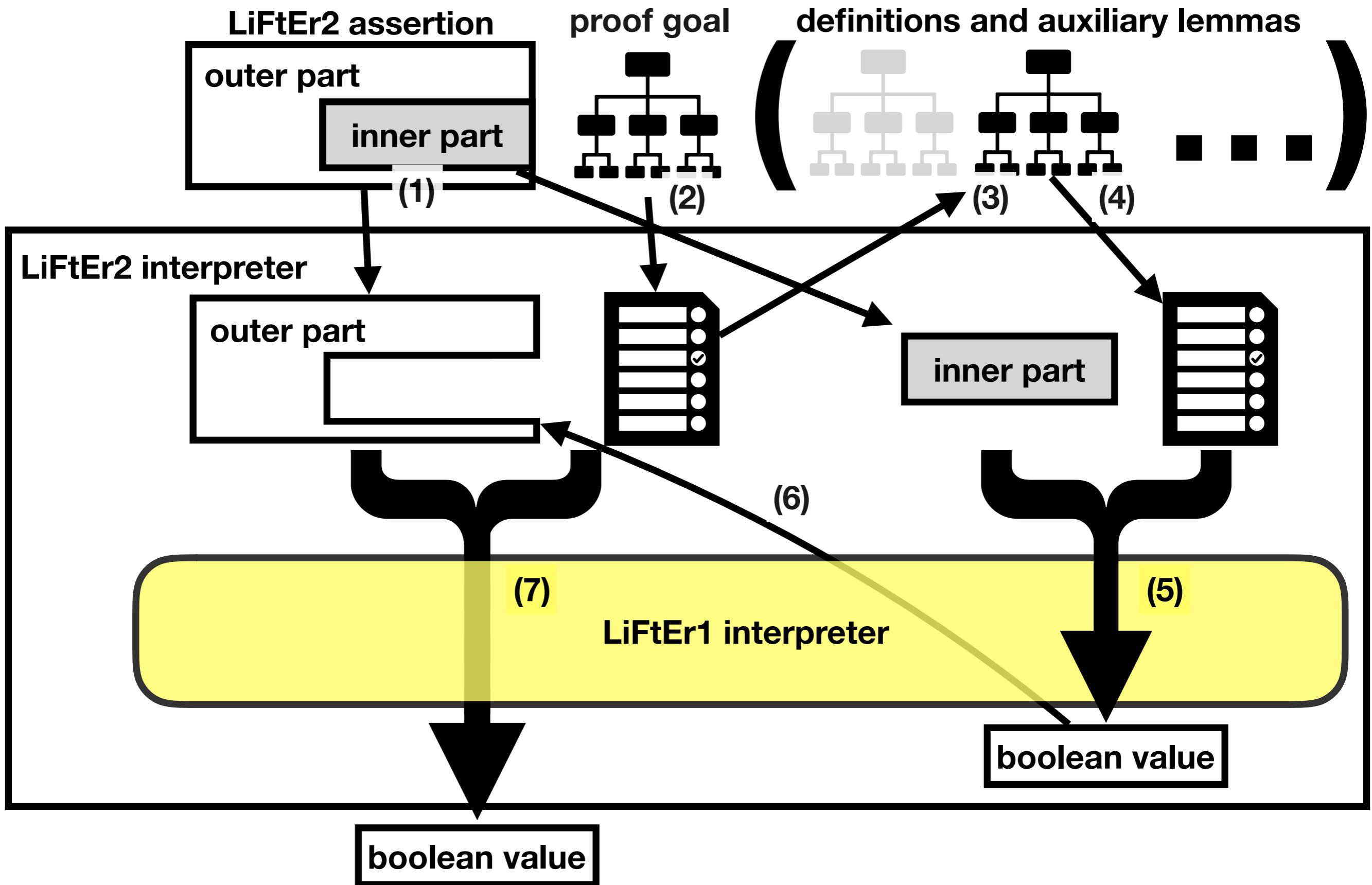


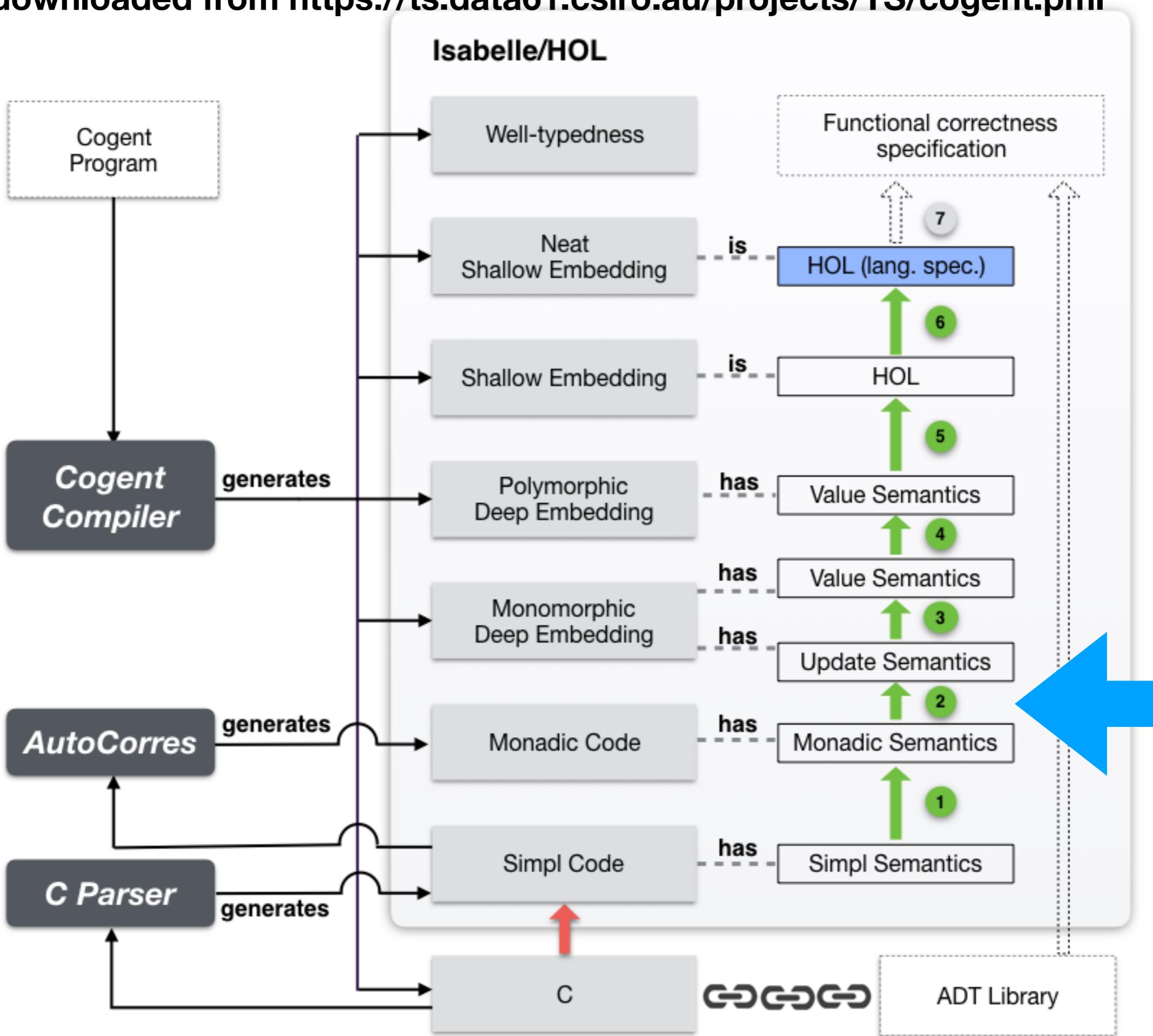
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

WIP!

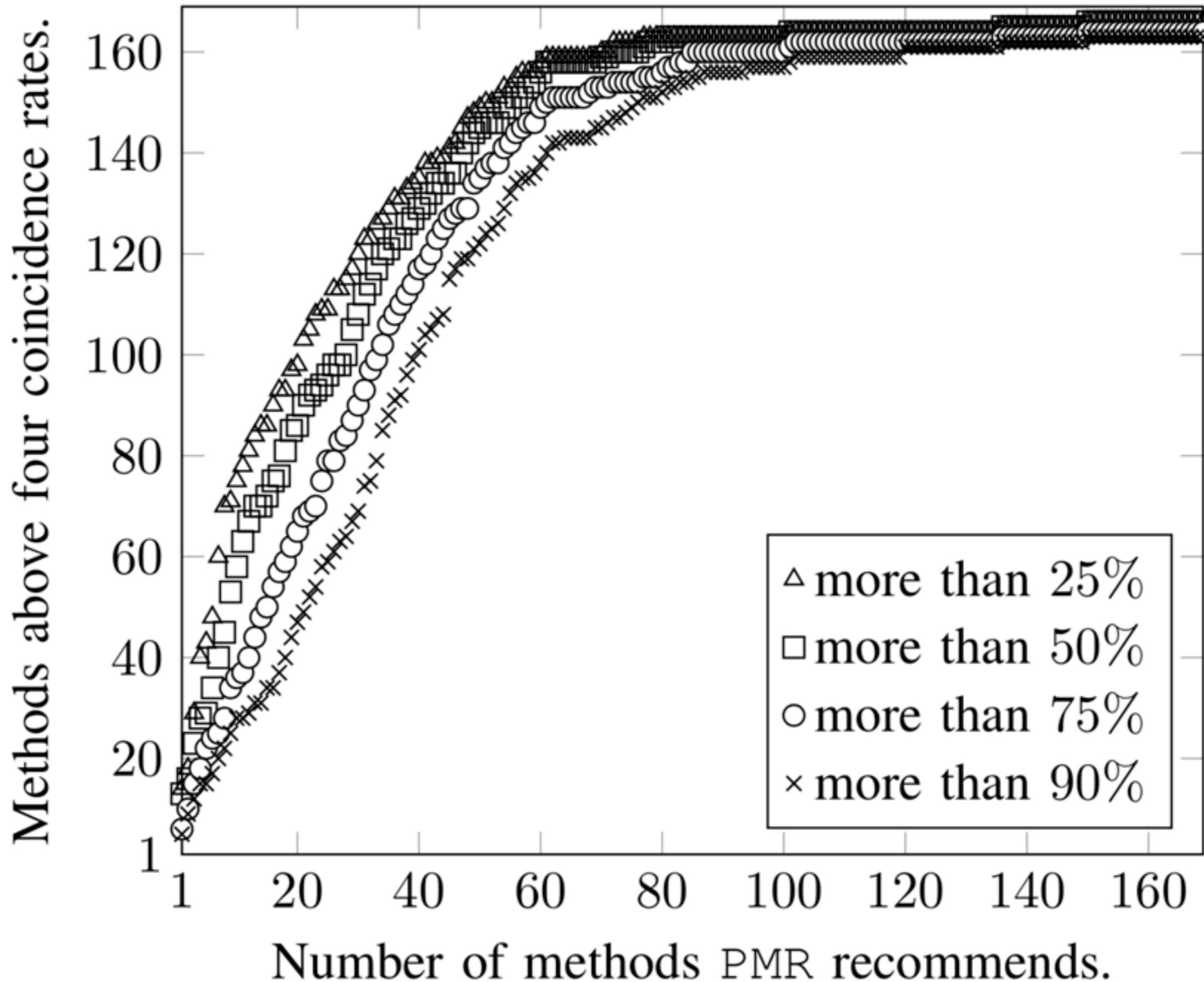


LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

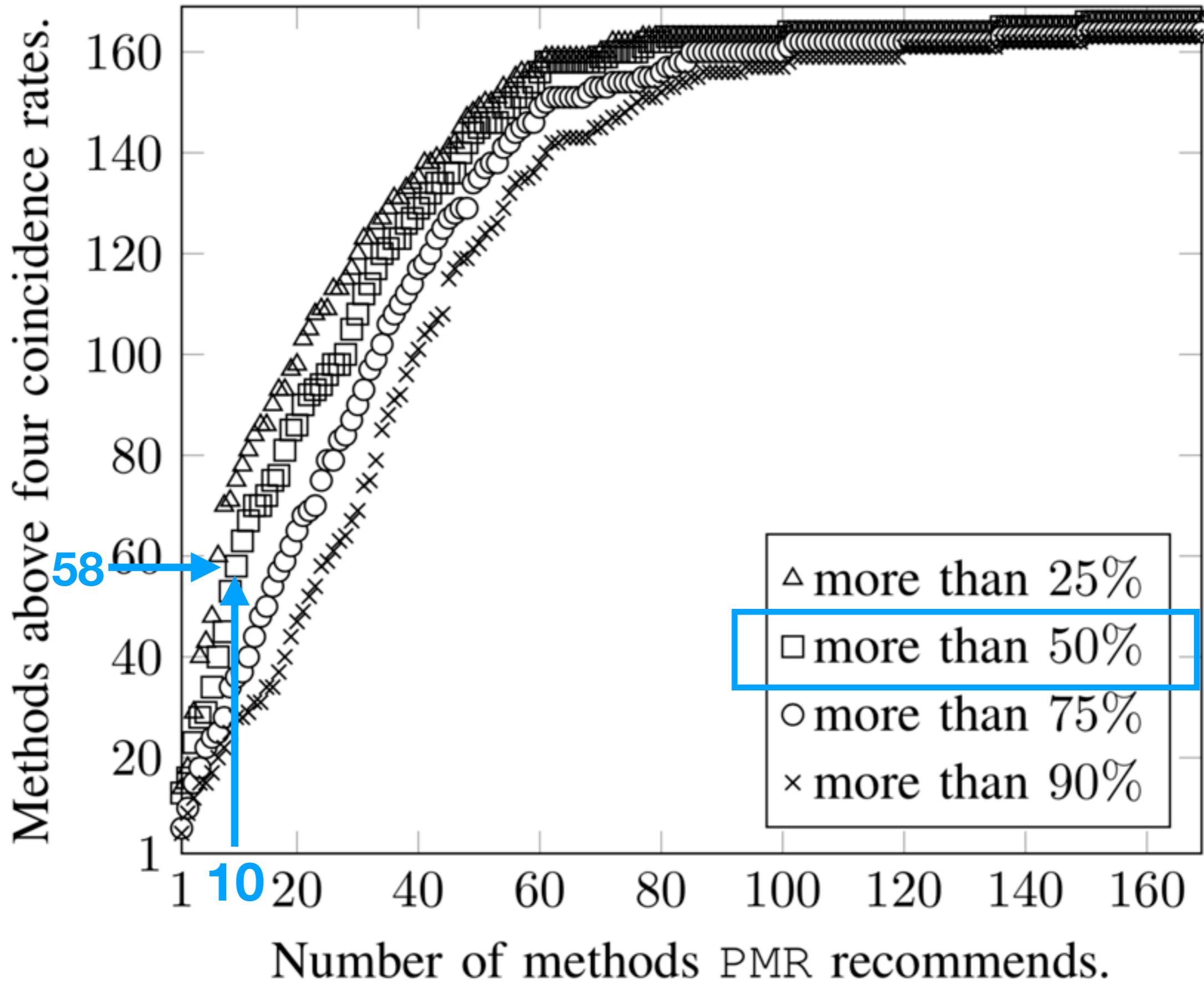




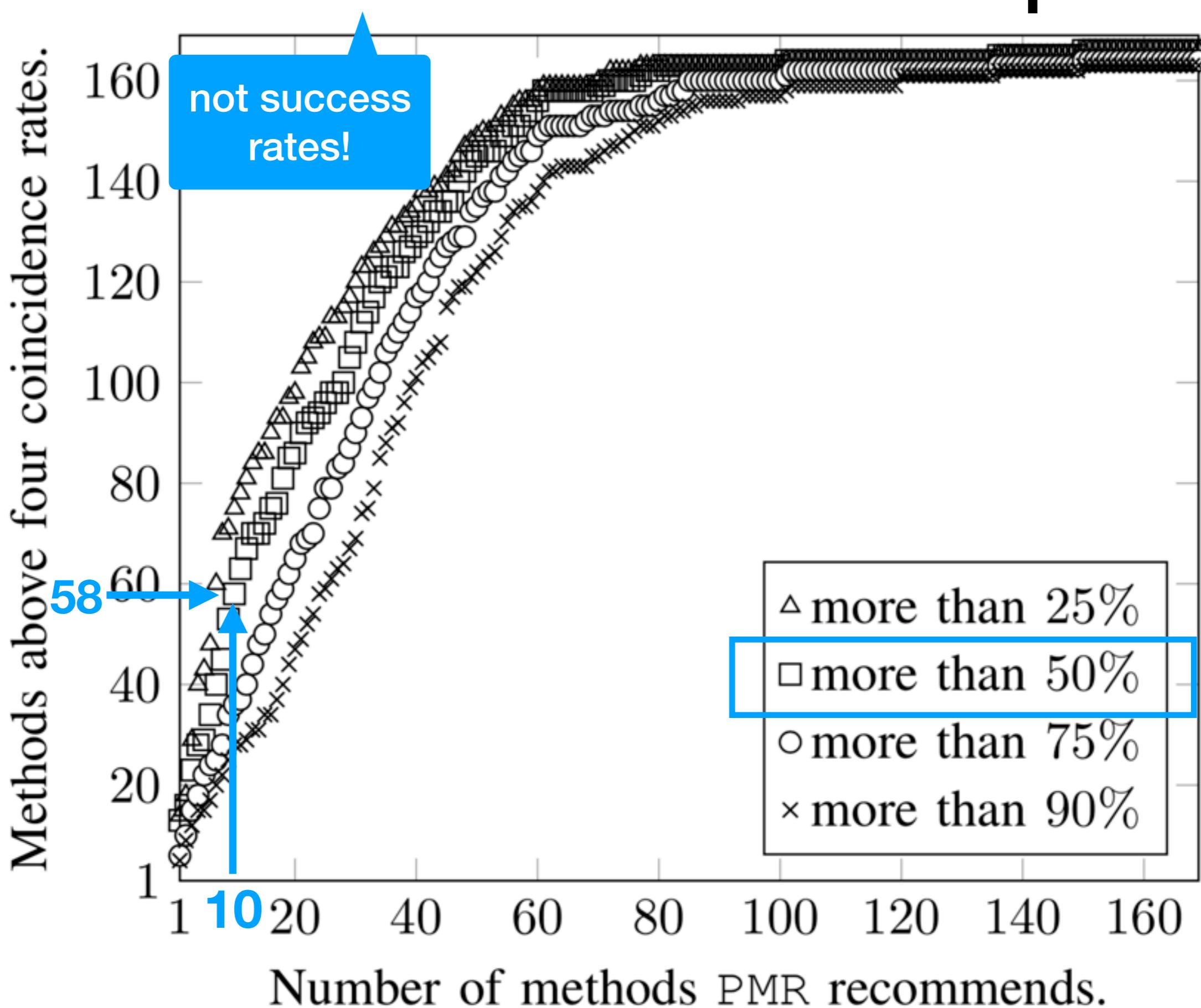
Coincidence rates of PaMpeR



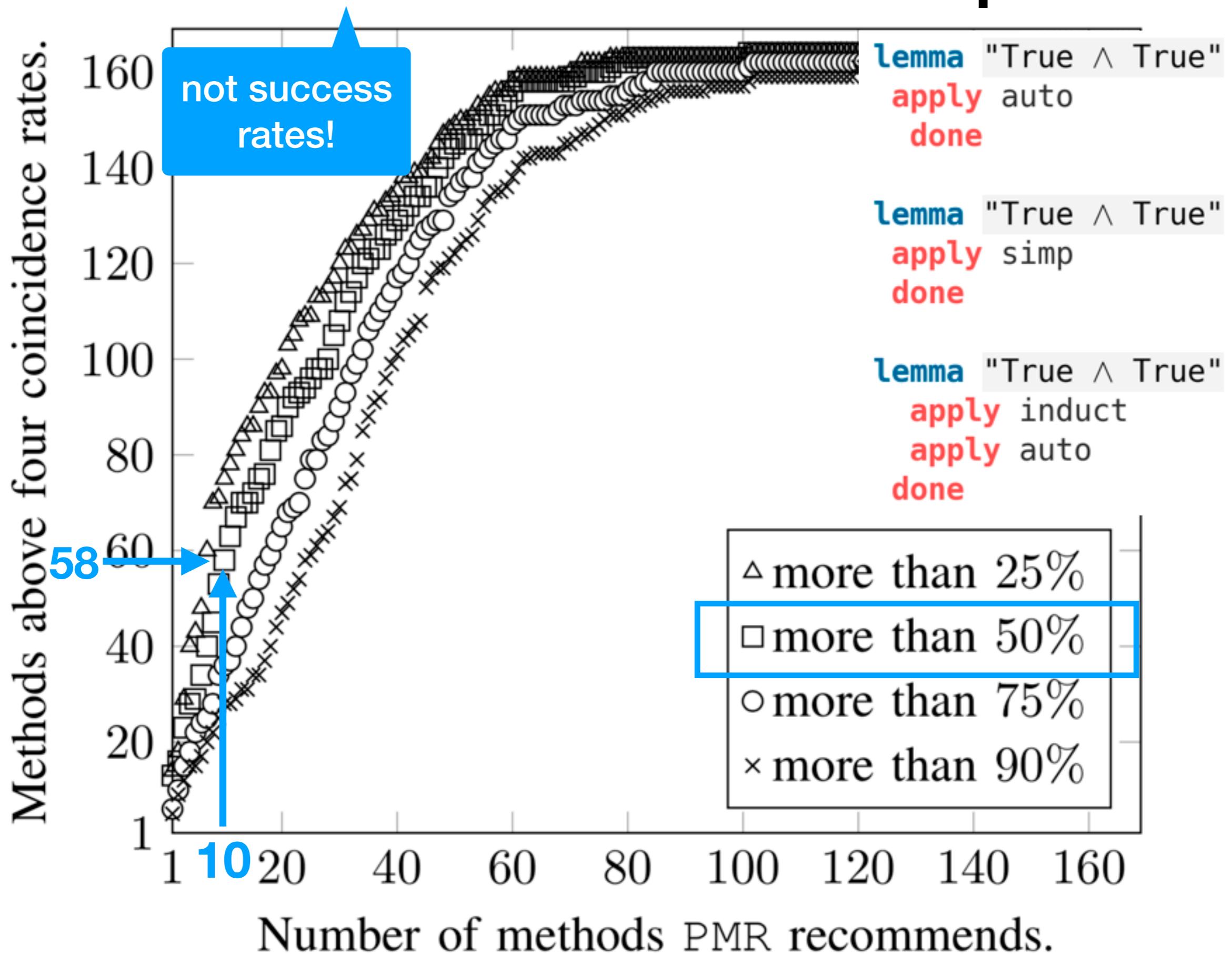
Coincidence rates of PaMpeR



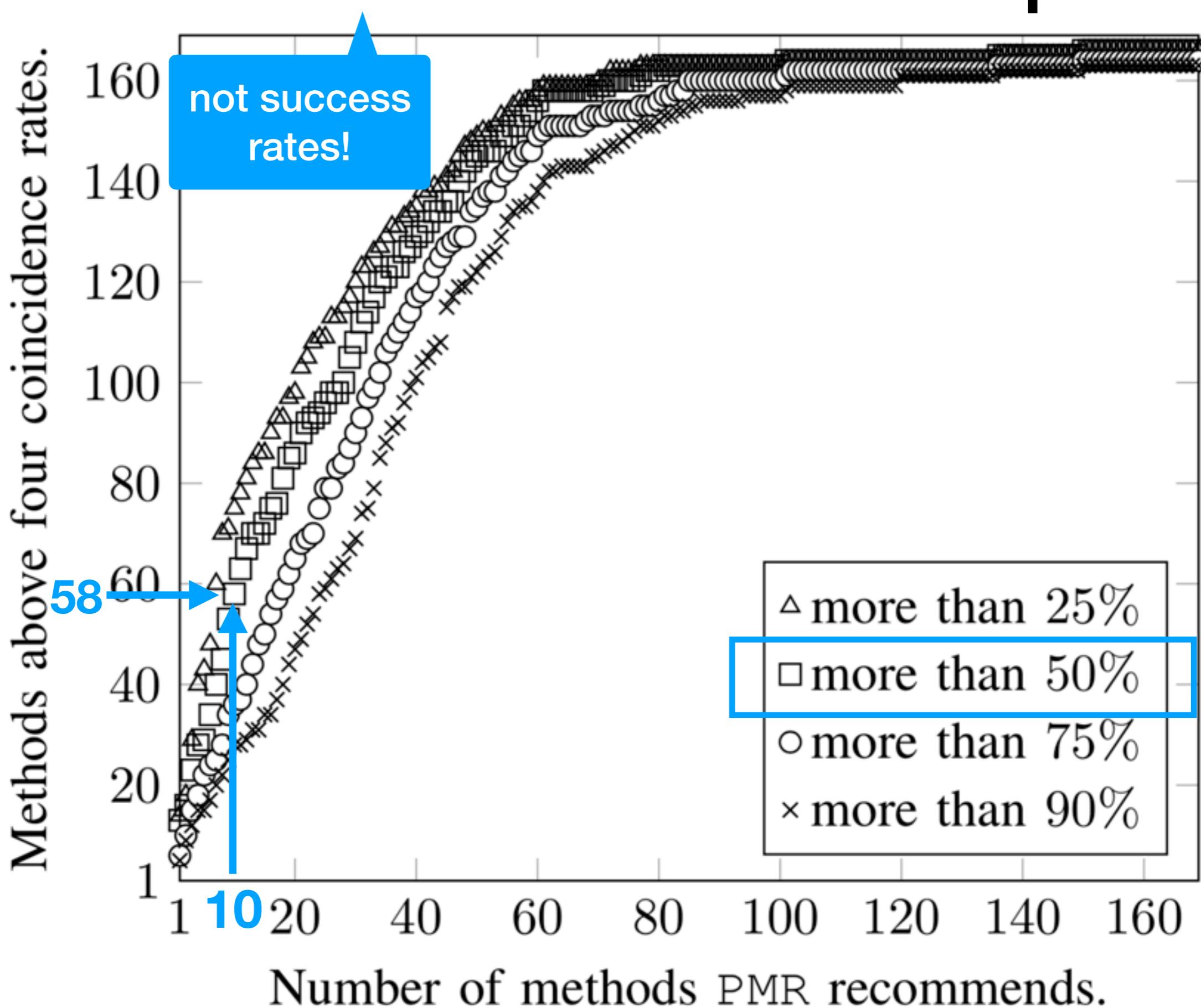
Coincidence rates of PaMpeR



Coincidence rates of PaMpeR



Coincidence rates of PaMpeR



Coincidence rates of PaMpeR

