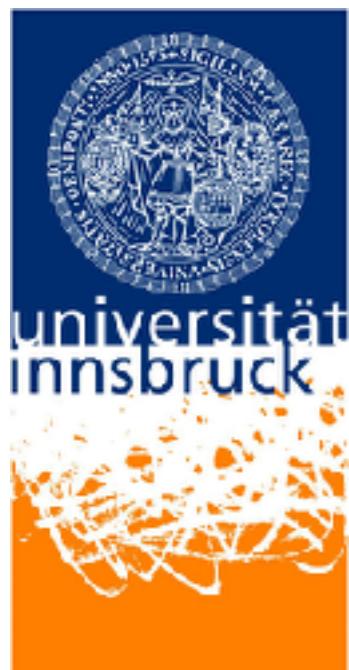




paper draft

(https://github.com/data61/PSL/blob/master/Smart_Induct/document/smart_induct.pdf)

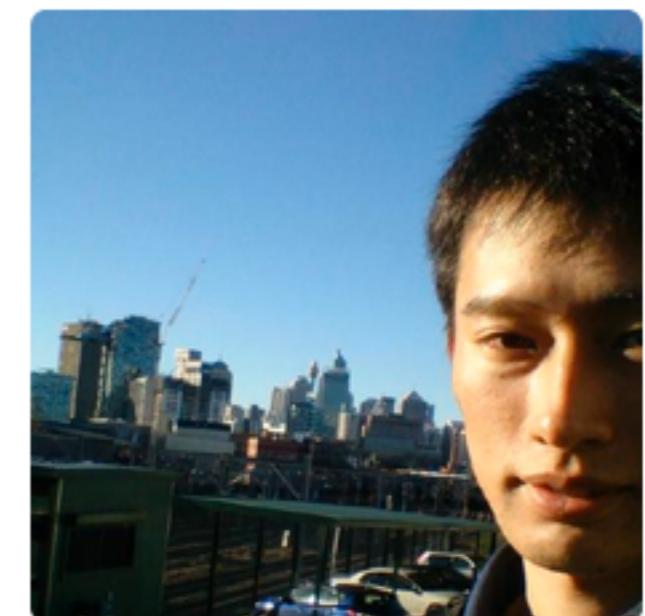
smart_induct



Yutaka Nagashima

email: yutaka.nagashima@cvut.cz

twitter: YutakangE



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

DEMO1: PSL

ta61/PSL/blob/master/slides/2020_Isabelle.pdf

The screenshot shows the Isabelle/Isar interface with a blue box highlighting the title "DEMO1: PSL". The main window displays a function definition:

```
fun sep :: "'a :: type list ⇒ 'a :: type list" where
  "sep a []" = []
  "sep a [x]" = [x]
  "sep a (x#y#zs)" = x # a # sep a (y#zs)
```

The third line of the definition is highlighted with a yellow background. Below the code editor, the output pane shows:

```
consts
  sep :: "'a :: type list ⇒ 'a :: type list"
Found termination order: "(λp. length (snd p)) <*lex*> {}"
```

The bottom status bar indicates the current session state and time.

DEMO1: PSL

data61/PSL/blob/master/slides/2020_Isabelle.pdf

The screenshot shows the Isabelle proof assistant interface with a PSL theory file open. The theory is named Example.thy and contains the following code:

```
fun sep:: "'a ⇒ 'a list ⇒ 'a list" where
  "sep a []"      = []
  "sep a [x]"     = [x]
  "sep a (x#y#zs)" = x # a # sep a (y#zs)

value "sep (1::int) [0,0,0]"
```

The cursor is positioned at the end of the third line of the theory definition. Below the theory editor, the output window displays the result of the value command:

```
"[0, 1, 0, 1, 0]"
:: "int list"
```

The interface includes standard Isabelle toolbars and status bars at the bottom.

DEMO1: PSL

ta61/PSL/blob/master/slides/2020_Isabelle.pdf

The screenshot shows the Isabelle proof assistant interface. The main window displays a theory file named `Example.thy`. The code defines a function `sep` and a value, and states a lemma. The lemma is currently being edited.

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

1 fun sep :: "'a ⇒ 'a list ⇒ 'a list" where
2   "sep a []      = []" |
3   "sep a [x]     = [x]" |
4   "sep a (x#y#zs) = x # a # sep a (y#zs)"
5
6 value "sep (1::int) [0,0,0]"
7
8
9
10
11
12
13
14
15
16 Lemma
17   "map f (sep x xs) = sep (f x) (map f xs)"
```

The lemma is:

$$\text{map } f \text{ (sep } x \text{ xs)} = \text{sep } (f \text{ x}) \text{ (map } f \text{ xs)}$$

The proof state is shown below:

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

At the bottom, there are tabs for Output, Query, Sledgehammer, and Symbols. The status bar at the bottom shows: 17.44 (304/1012), Input/Output complete, (isabelle.isabelle,UTF-8-Isabelle), 241/535MB, 1:17 AM.

DEMO1: PSL

ta61/PSL/blob/master/slides/2020_Isabelle.pdf

The screenshot shows the Isabelle proof assistant interface with a theory file named `Example.thy`. The code defines a function `sep` and a lemma. The `sep` function is defined with three cases: an empty list, a single element, and a list of elements. The lemma states that applying a function `f` to each element of a separated list is equivalent to separating the results of applying `f` to each element. The `strategy` is set to `DInd`, which is highlighted in yellow.

```
fun sep:: "'a ⇒ 'a list ⇒ 'a list" where
  "sep a []      = []" |
  "sep a [x]     = [x]" |
  "sep a (x#y#zs) = x # a # sep a (y#zs)"

value "sep (1::int) [0,0,0]"

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

lemma
  "map f (sep x xs) = sep (f x) (map f xs)"
```

DEMO1: PSL

data61/PSL/blob/master/slides/2020_Isabelle.pdf

The screenshot shows the Isabelle/Isar interface with a theory file named `Example.thy`. The code defines a function `sep` and a lemma `map_sep`.

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

1 fun sep :: "'a ⇒ 'a list ⇒ 'a list" where
2   "sep a []      = []" |
3   "sep a [x]     = [x]" |
4   "sep a (x#y#zs) = x # a # sep a (y#zs)"
5
6 value "sep (1::int) [0,0,0]"
7
8 strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved]
9
10 lemma
11   "map f (sep x xs) = sep (f x) (map f xs)" □
12   find_proof DInd
13
14 proof (prove)
15 goal (1 subgoal):
16   1. map f (sep x xs) = sep (f x) (map f xs)
17
18 Output Query Sledgehammer Symbols
19
20 17.44 (361/1084) (isabelle/isabelle,UTF-8-Isabelle) nmr@U... 242/535MB 1:05 AM
```

DEMO1: PSL

ta61/PSL/blob/master/slides/2020_Isabelle.pdf

The screenshot shows the Isabelle/Isar interface with a theory file named Example.thy. The code defines a function sep and a lemma map_f_sep.

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

1 fun sep :: "'a ⇒ 'a list ⇒ 'a list" where
2   "sep a []      = []" |
3   "sep a [x]     = [x]" |
4   "sep a (x#y#zs) = x # a # sep a (y#zs)"

5 value "sep (1::int) [0,0,0]"

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

7 Lemma
8   "map f (sep x xs) = sep (f x) (map f xs)"
9   find_proof DInd

10
11
12
13
14
15
16
17
18
19
20
21
```

The interface includes a toolbar at the top, a vertical navigation bar on the left, and a status bar at the bottom. The status bar shows the file name, encoding, memory usage, and current time.

Number of lines of commands: 3

```
apply (induct xs rule: Example.sep.induct)
apply auto
done
```

DEMO1: PSL

ta61/PSL/blob/master/slides/2020_Isabelle.pdf

The screenshot shows the Isabelle proof assistant interface with a theory file named `Example.thy`. The code defines a function `sep` and proves a lemma about its behavior when composed with a map operation.

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

1 fun sep :: "'a ⇒ 'a list ⇒ 'a list" where
2   "sep a []      = []" |
3   "sep a [x]     = [x]" |
4   "sep a (x#y#zs) = x # a # sep a (y#zs)"

5 value "sep (1::int) [0,0,0]"

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

7 Lemma
8   "map f (sep x xs) = sep (f x) (map f xs)"
9   find_proof DInd
10  apply (induct xs rule: Example.sep.induct)
11  apply auto
12  done

13 proof (prove)
14 goal:
15 No subgoals!
```

The proof state at the bottom shows the goal `proof (prove)` and `goal:`, with the message `No subgoals!`.

At the bottom of the interface, there are tabs for `Output`, `Query`, `Sledgehammer`, and `Symbols`. The status bar at the bottom right indicates the session name `(isabelle.isabelle,UTF-8-Isabelle)`, memory usage `255/535MB`, and the current time `1:05 AM`.

DEMO1: PSL

ta61/PSL/blob/master/slides/2020_Isabelle.pdf

The screenshot shows the Isabelle/Isar proof assistant interface. The main window displays a proof script in ML-like syntax. A blue box highlights the section from 'fun' to 'where'. A yellow highlight covers the 'apply auto' command. A red box highlights the 'done' command. A blue speech bubble in the bottom right corner contains the text 'What happened?'. The status bar at the bottom shows '20.11 (433/1147)' and '(isabelle.isabelle,UTF-8-Isabelle) n m r o U.. 255/535MB 1:05 AM'.

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

1 fun sep :: "'a ⇒ 'a list ⇒ 'a list" where
2   "sep a []"      = []
3   "sep a [x]"     = [x]
4   "sep a (x#y#zs)" = x # a # sep a (y#zs)
5
6 value "sep (1::int) [0,0,0]"
7
8 strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved]
9
10 lemma
11   "map f (sep x xs) = sep (f x) (map f xs)"
12   find_proof DInd
13 apply (induct xs rule: Example.sep.induct)
14 apply auto
15 done

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

What happened?

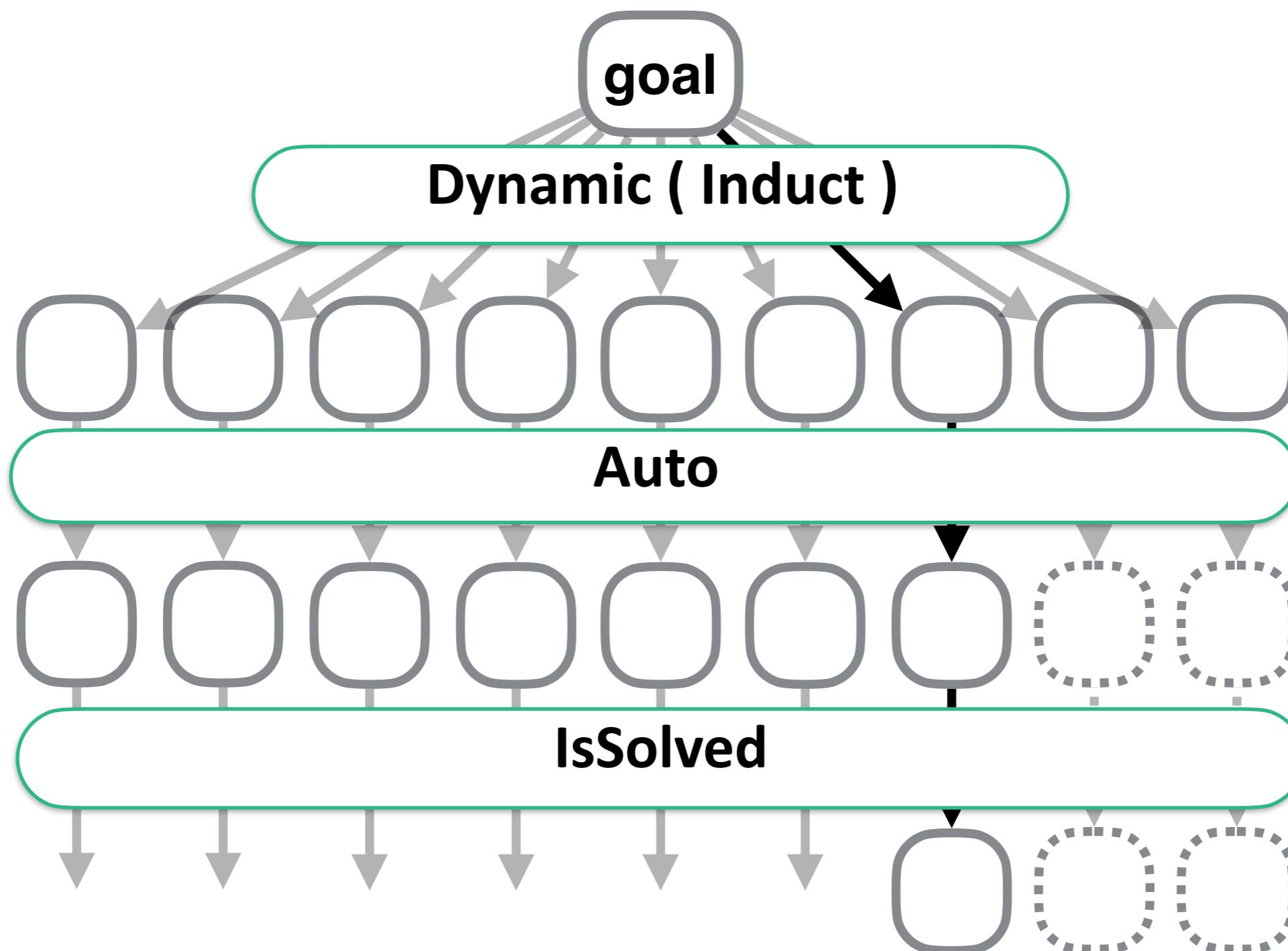
https://github.com/data61/PSL/blob/master/slide/2020_Isabelle.pdf

PSL: Proof Strategy Language

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

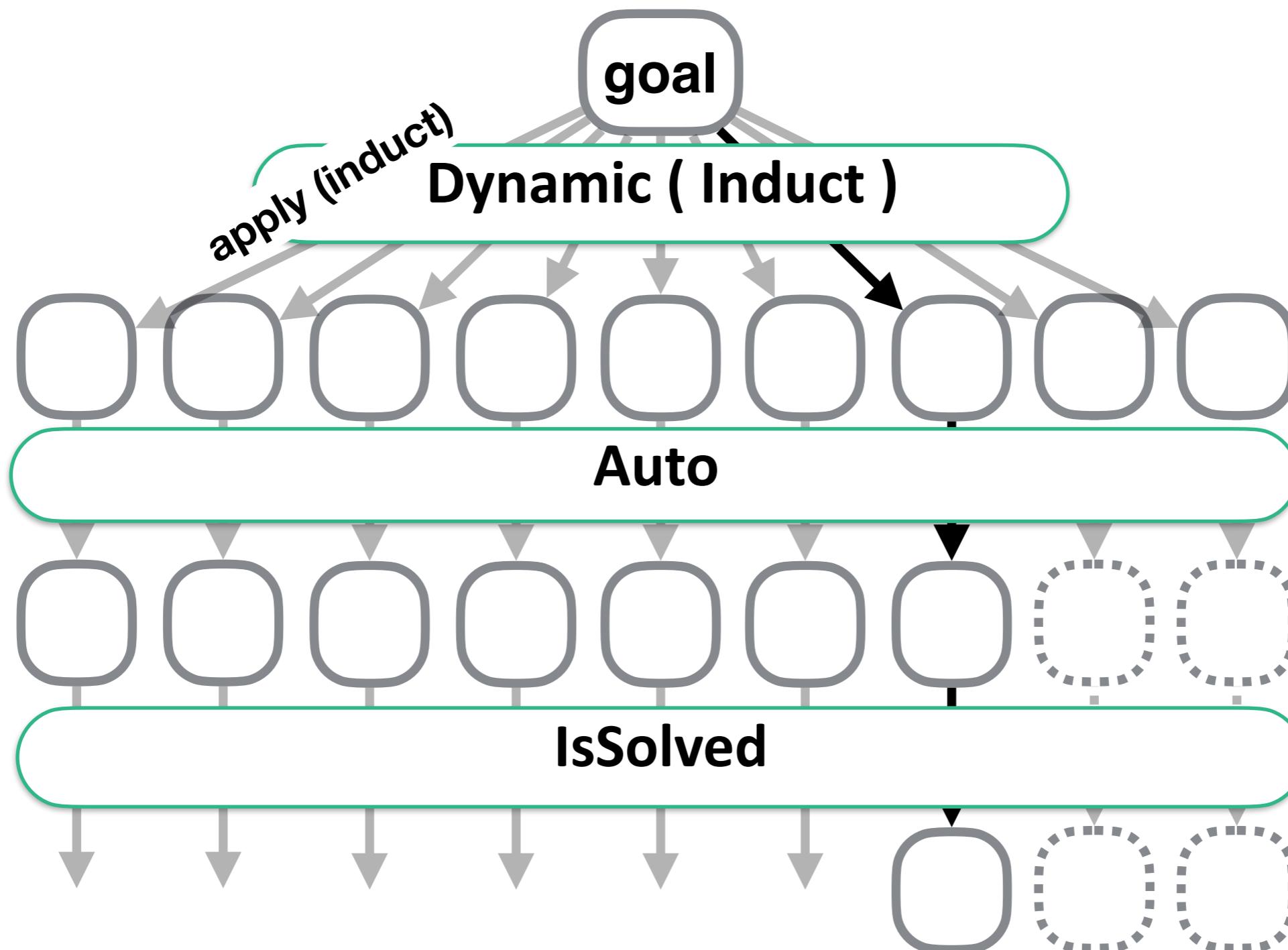
PSL: Proof Strategy Language

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



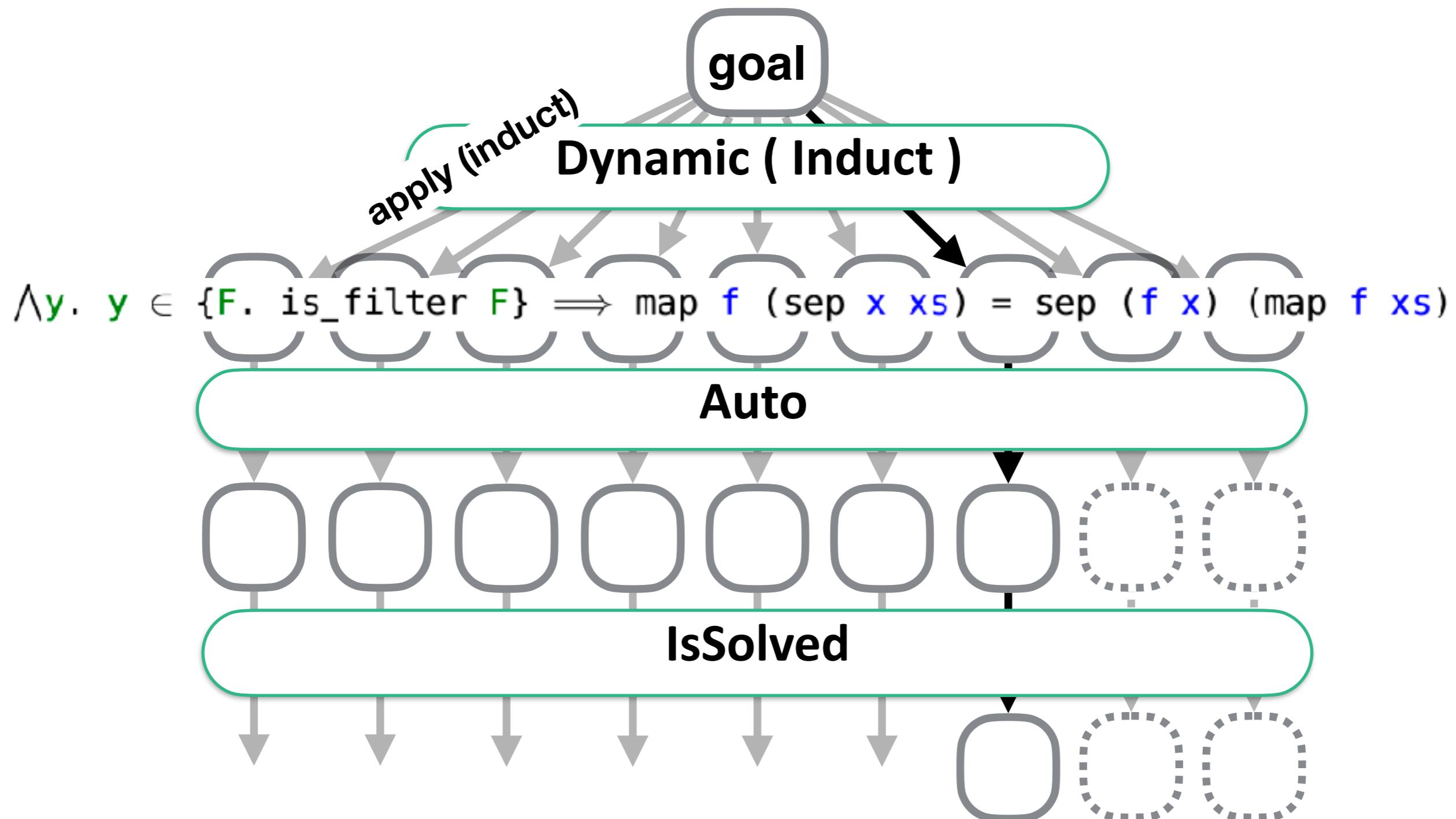
PSL: Proof Strategy Language

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



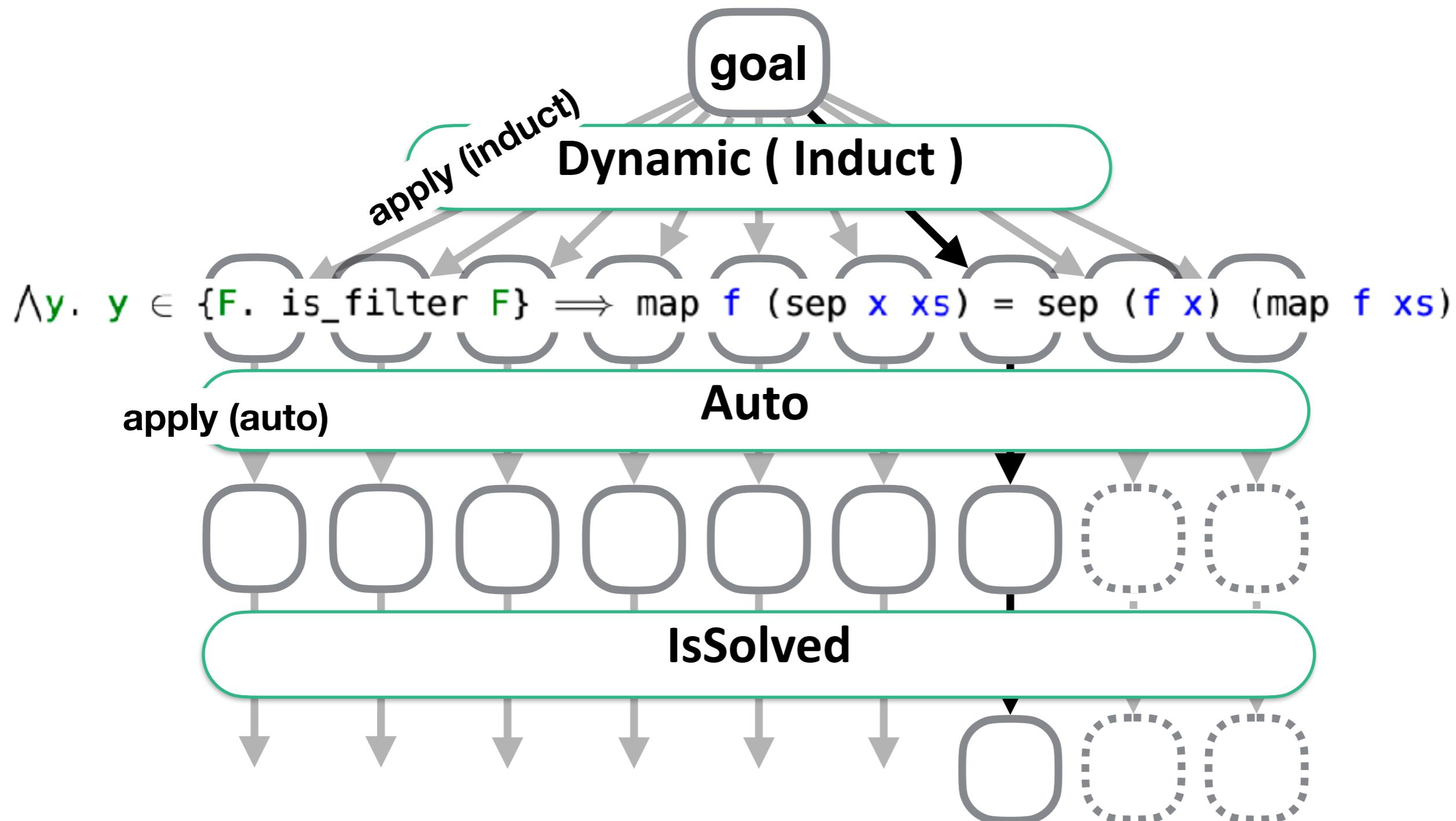
PSL: Proof Strategy Language

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



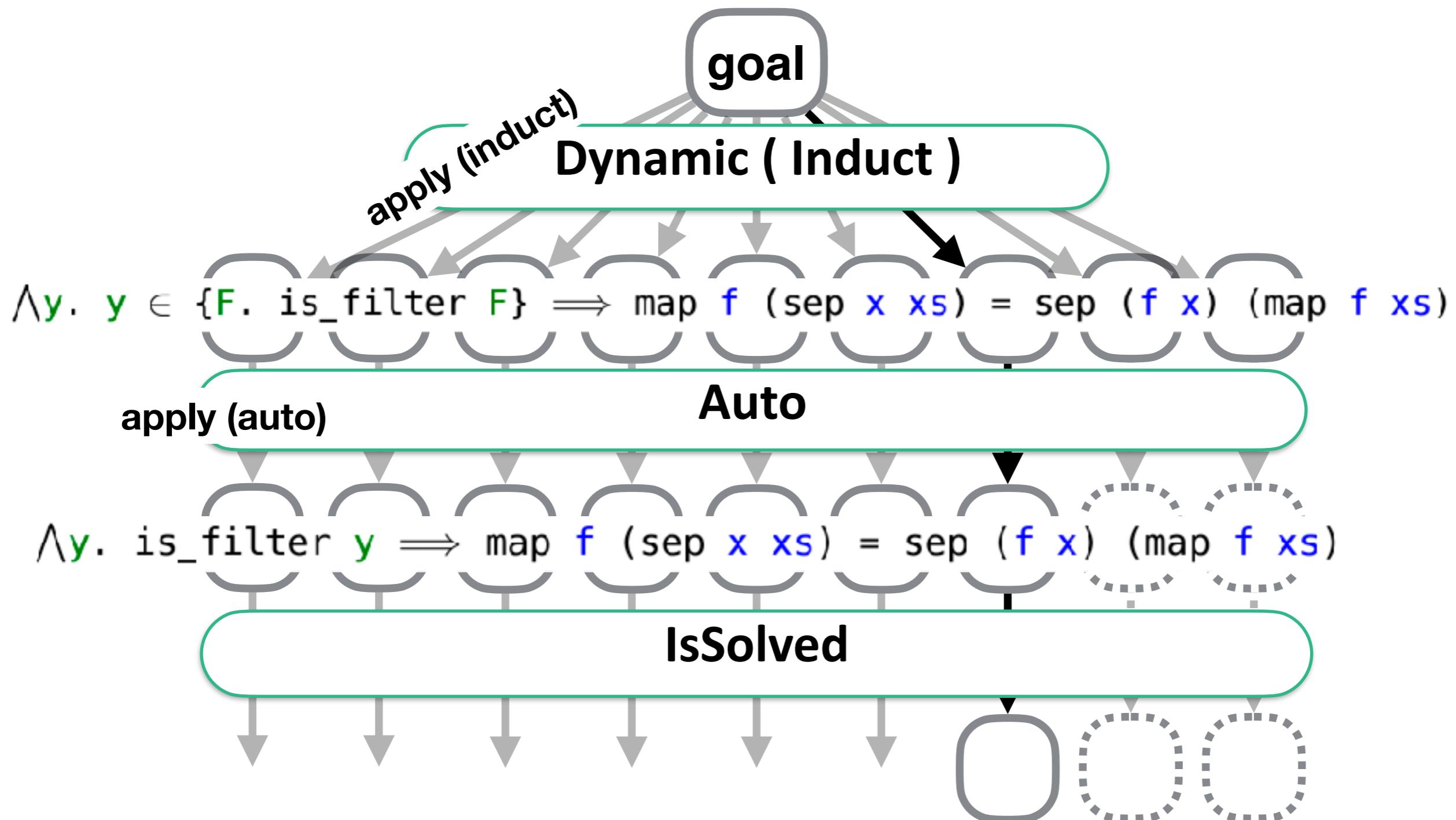
PSL: Proof Strategy Language

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



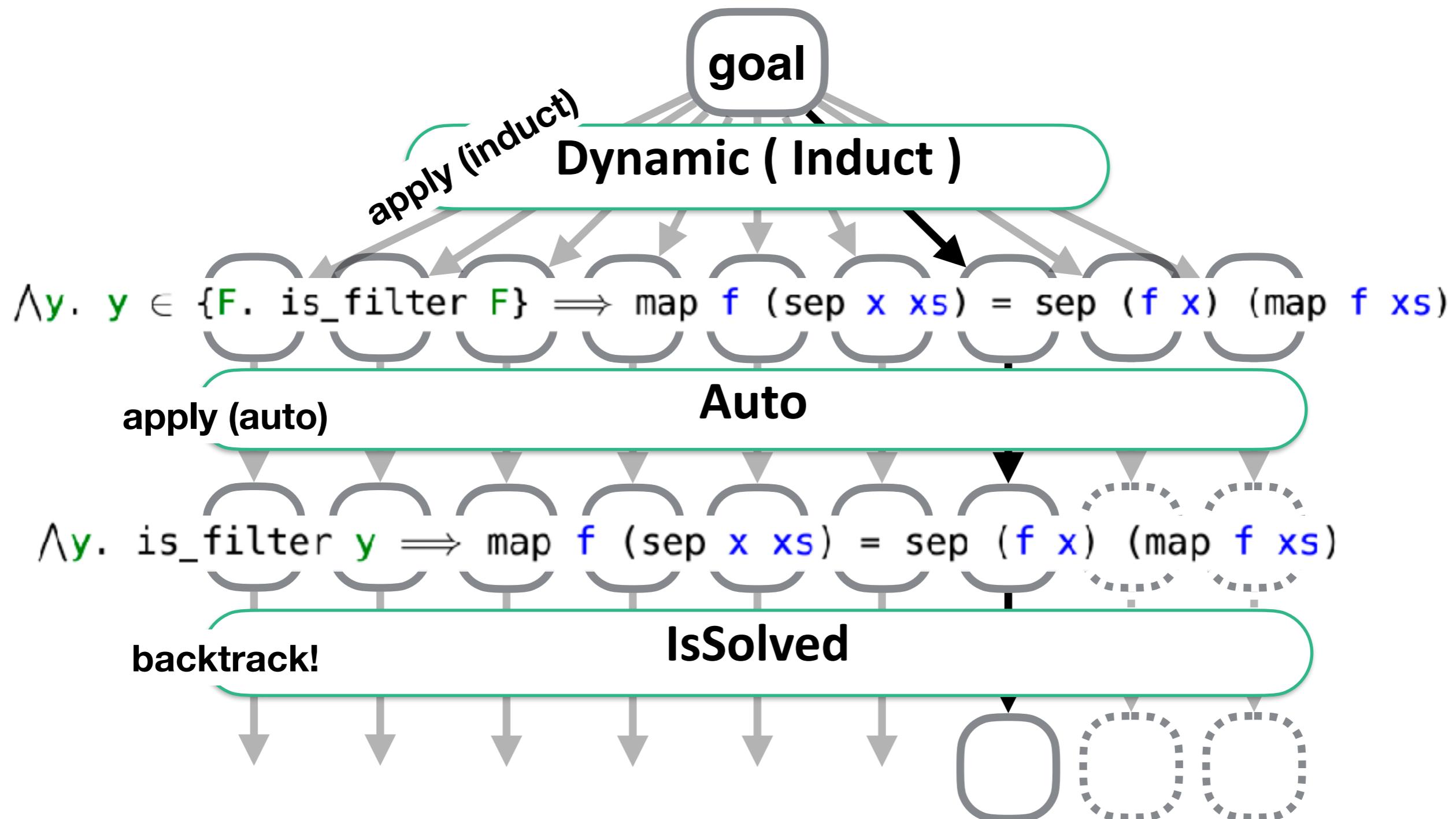
PSL: Proof Strategy Language

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



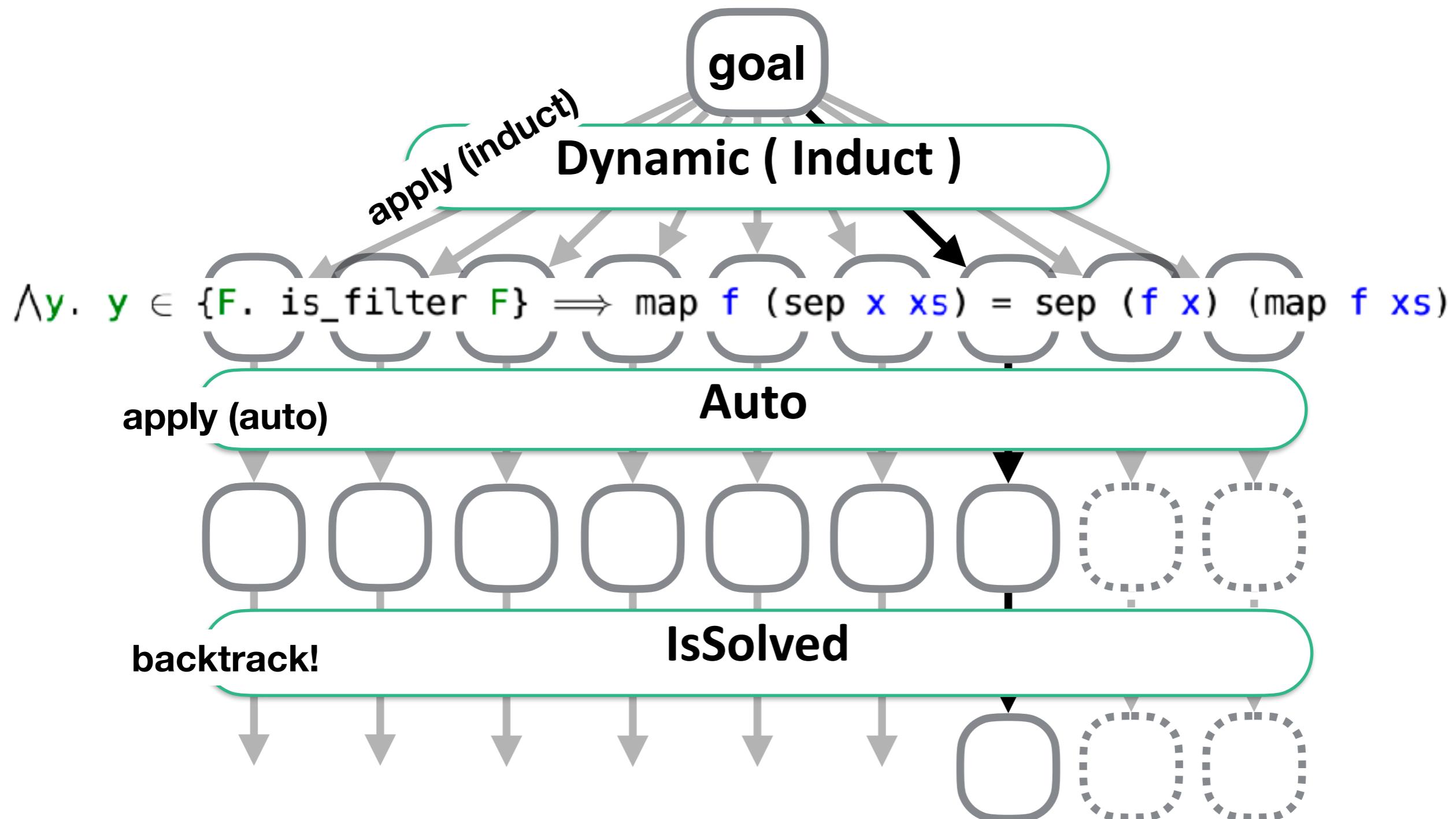
PSL: Proof Strategy Language

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



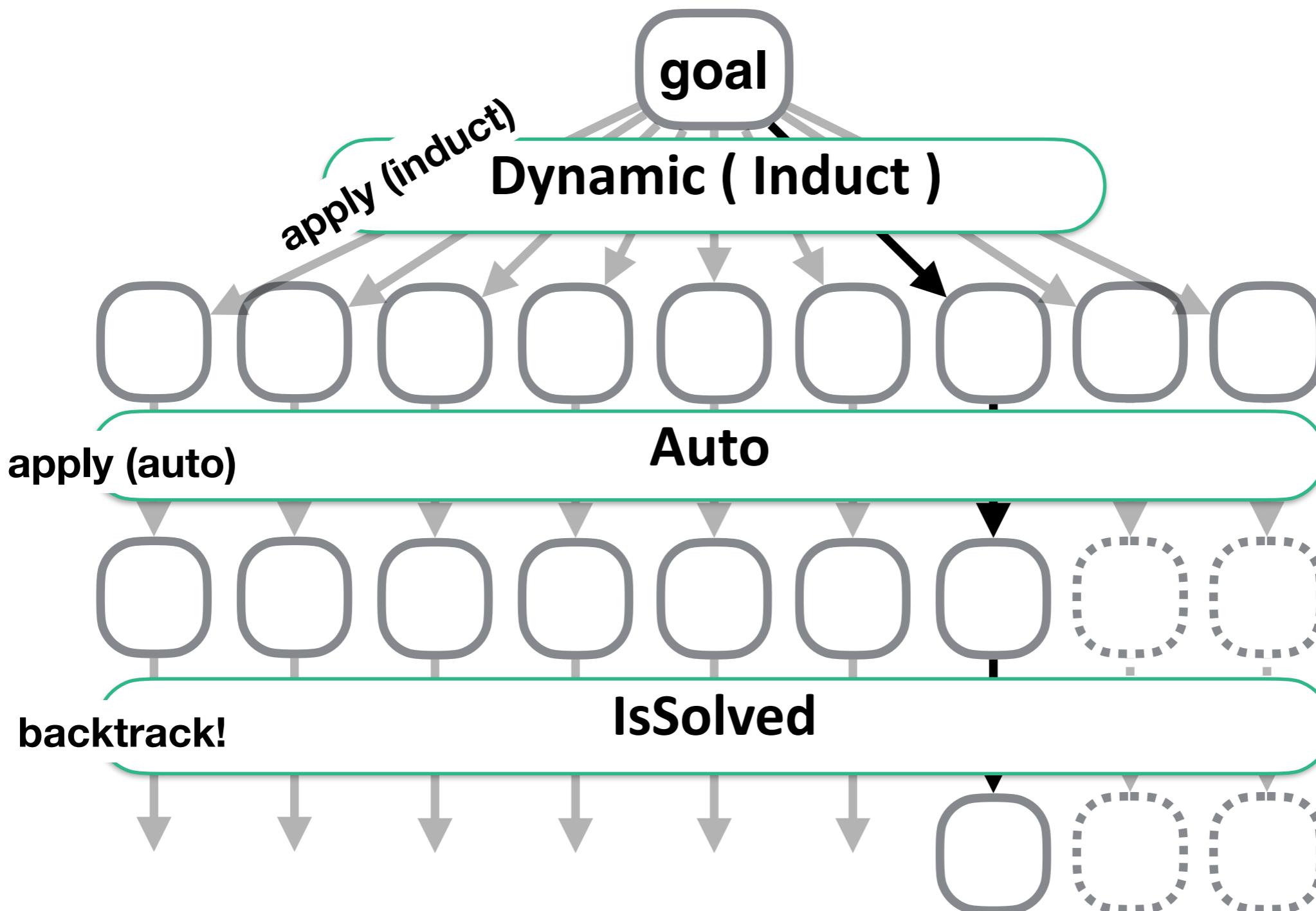
PSL: Proof Strategy Language

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



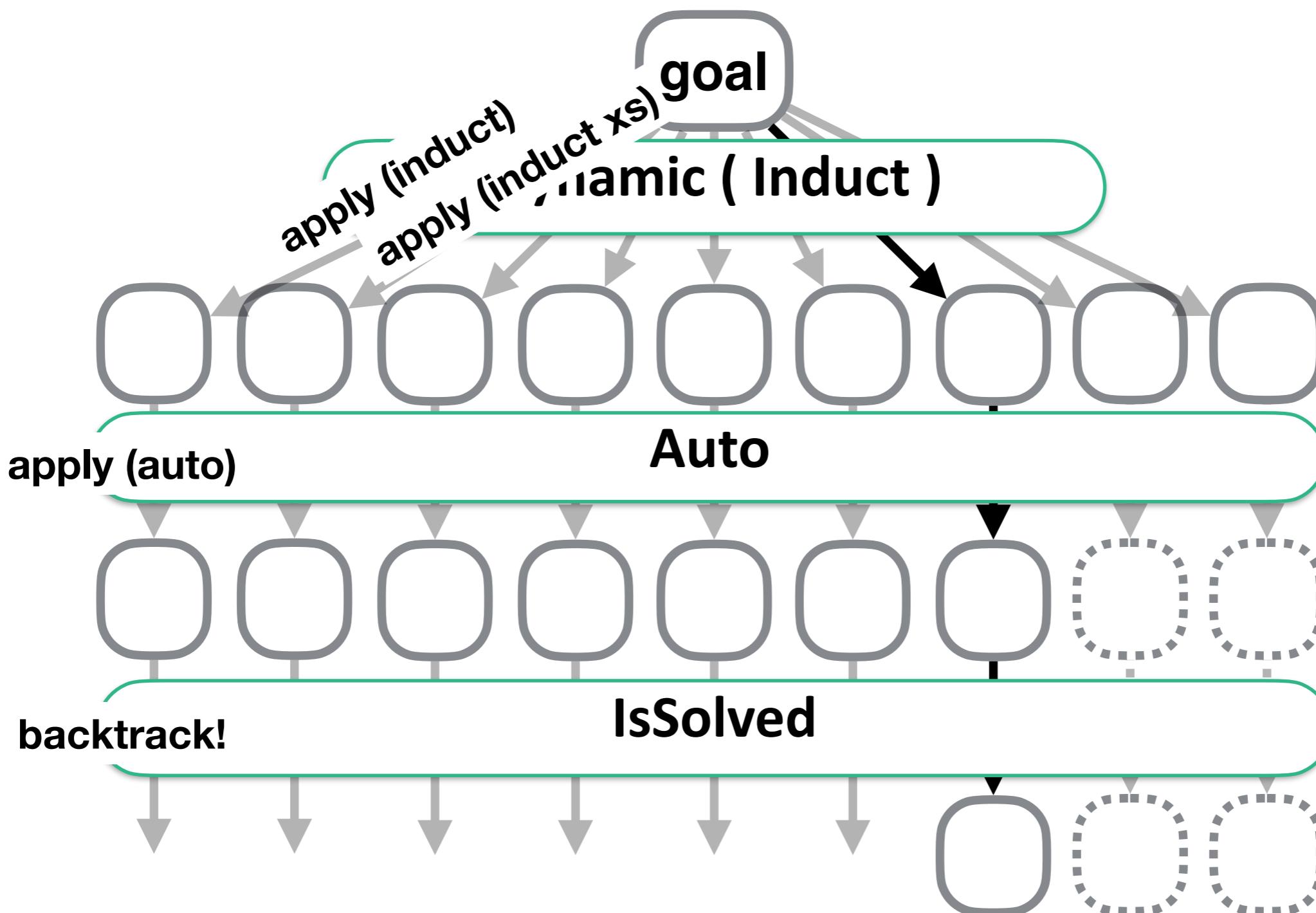
PSL: Proof Strategy Language

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



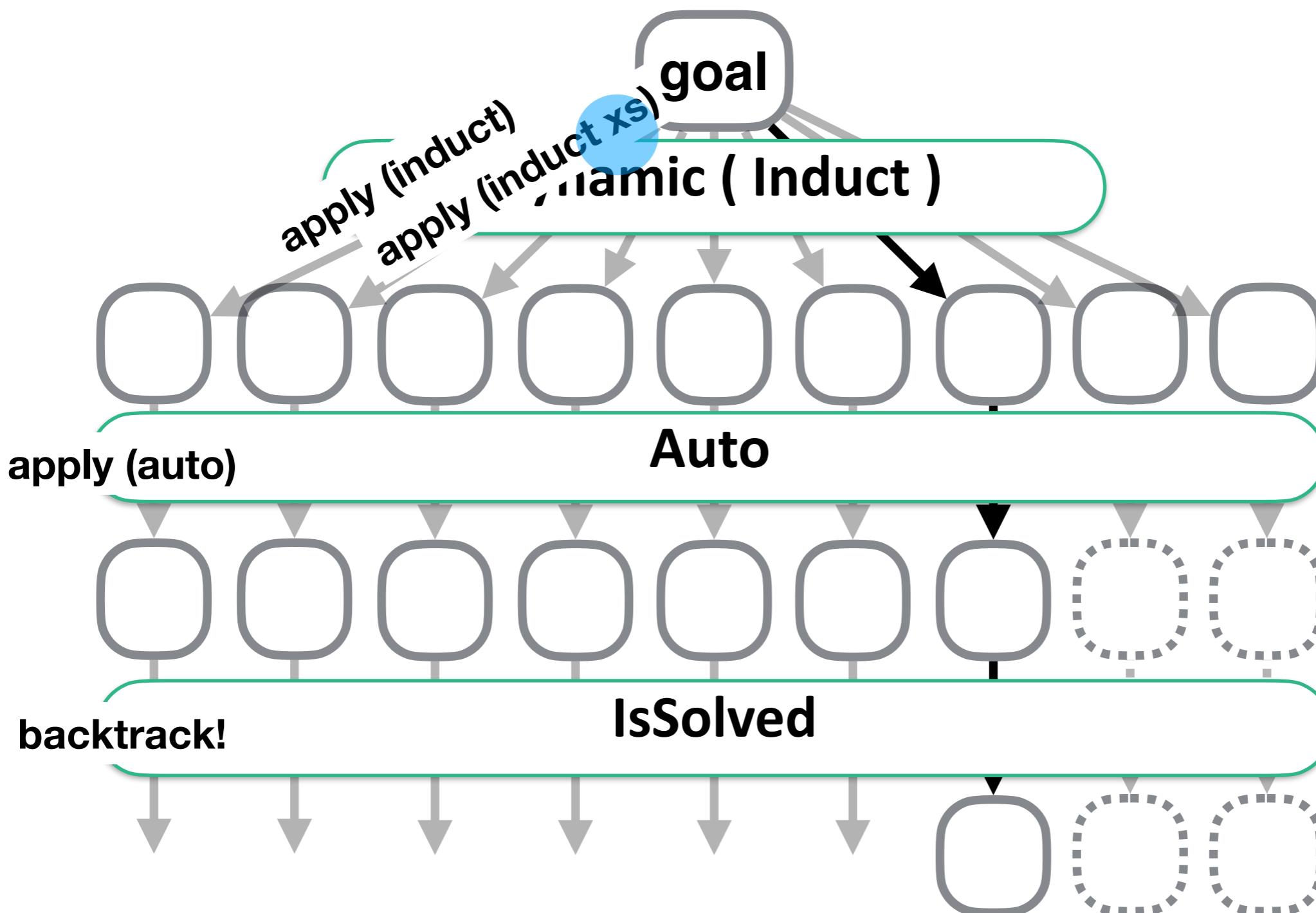
PSL: Proof Strategy Language

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



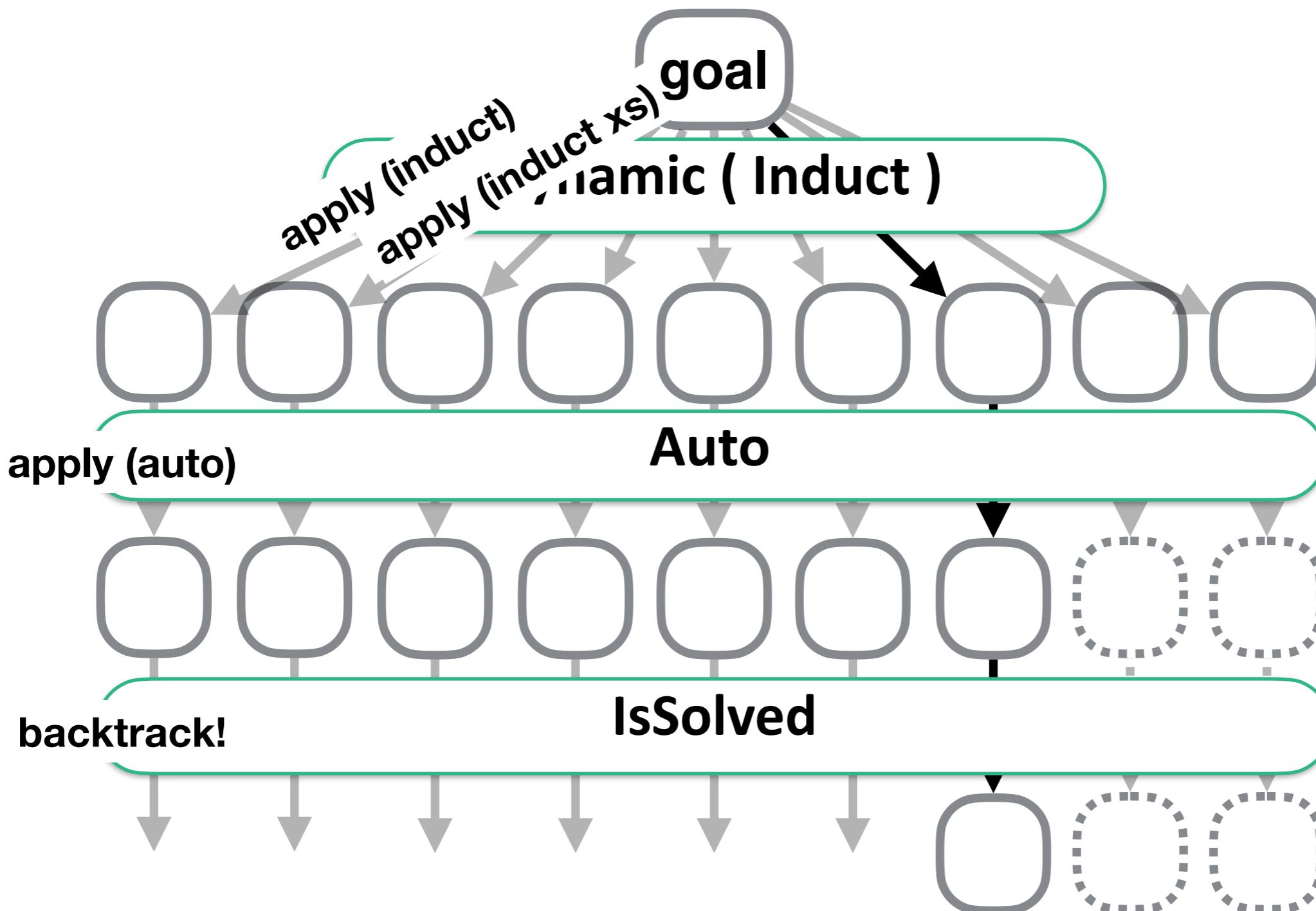
PSL: Proof Strategy Language

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



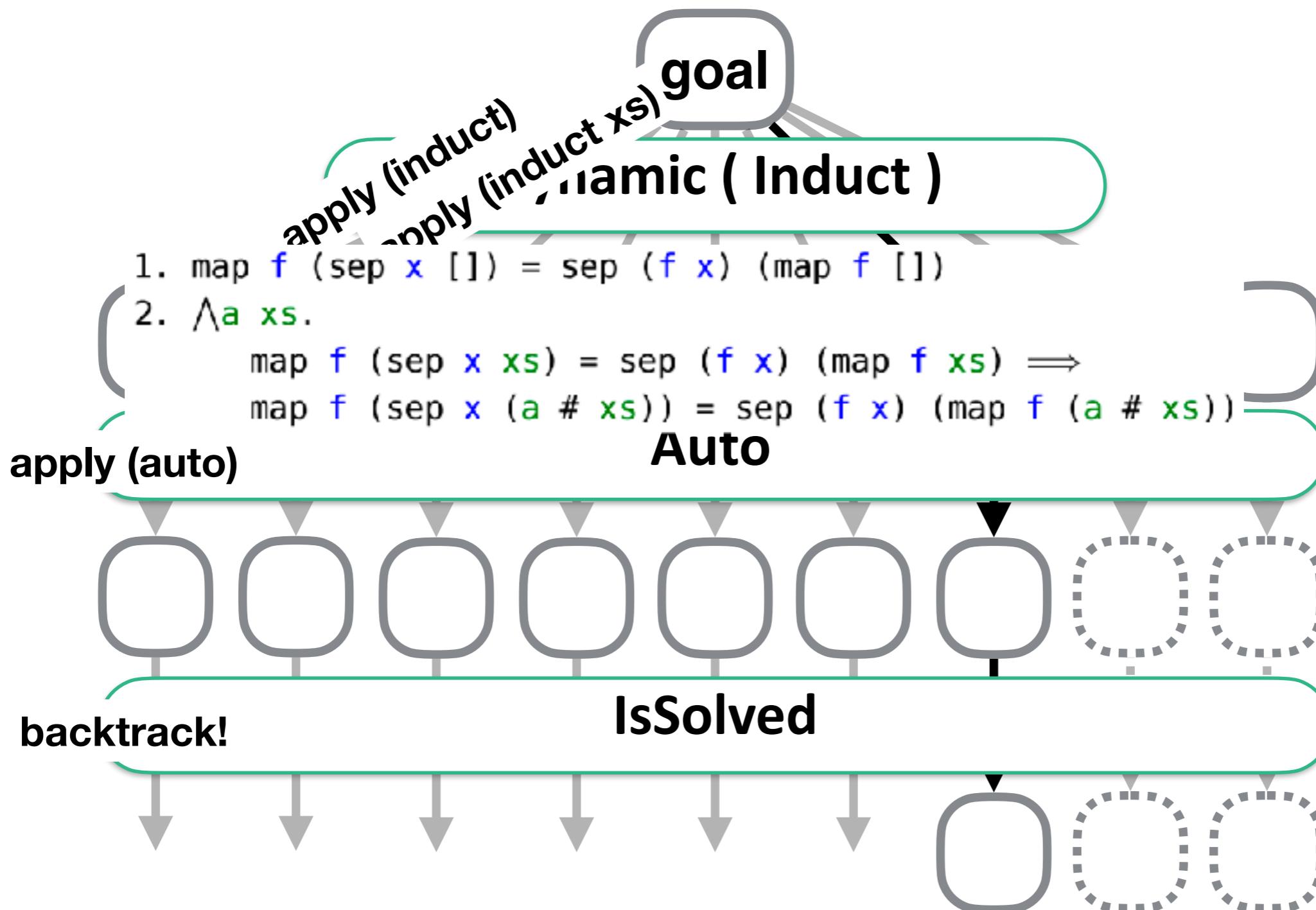
PSL: Proof Strategy Language

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



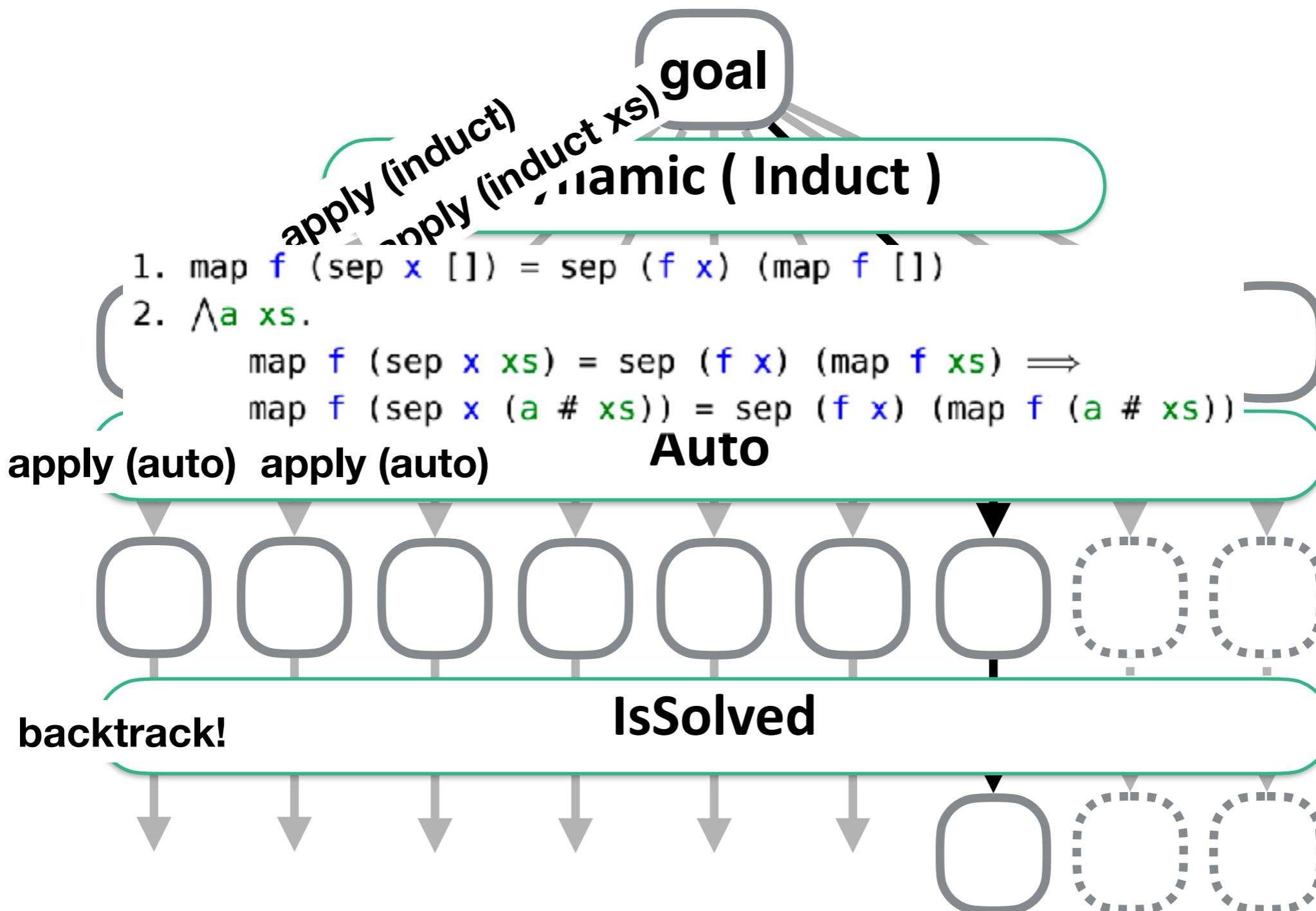
PSL: Proof Strategy Language

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



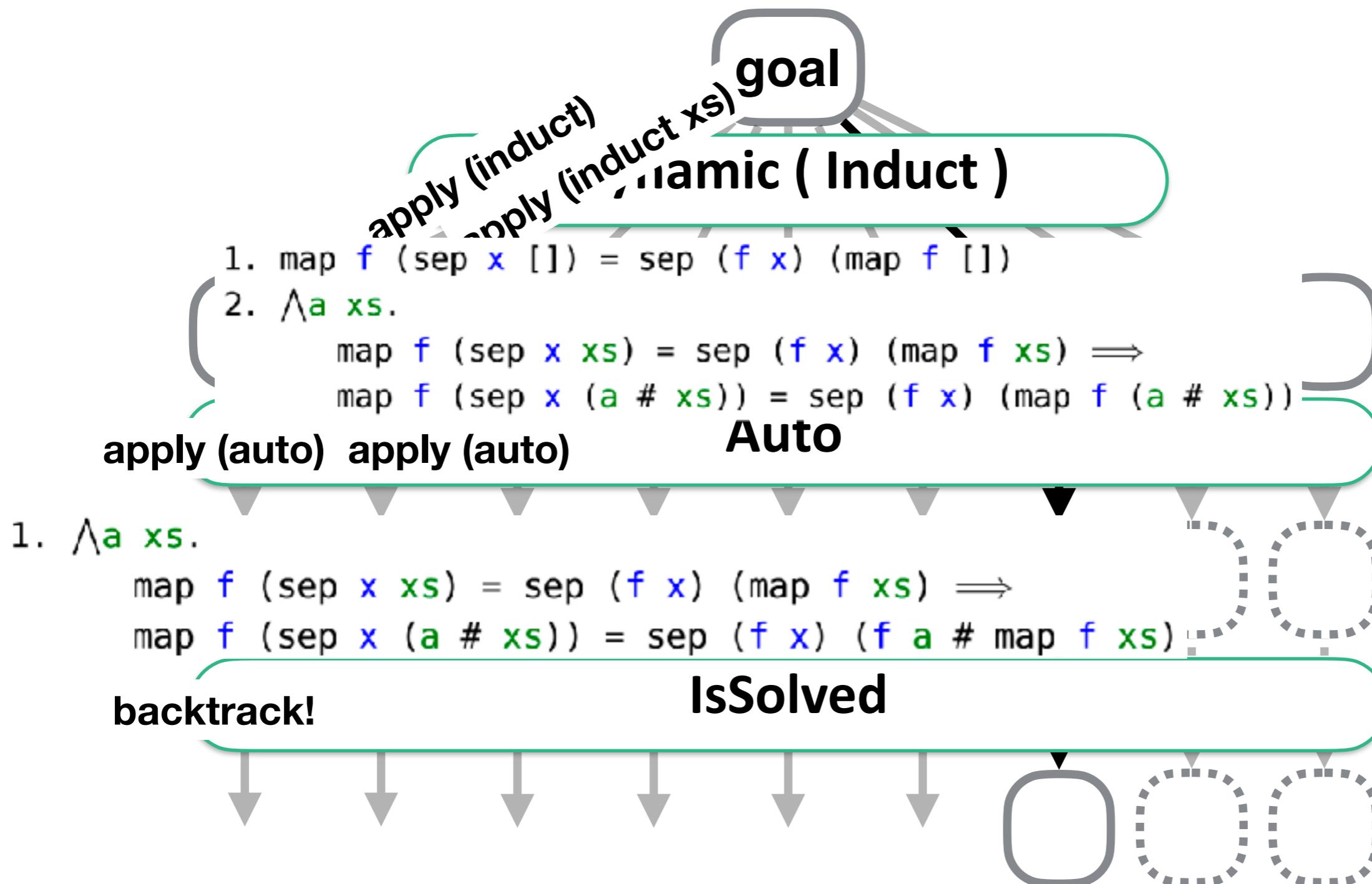
PSL: Proof Strategy Language

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



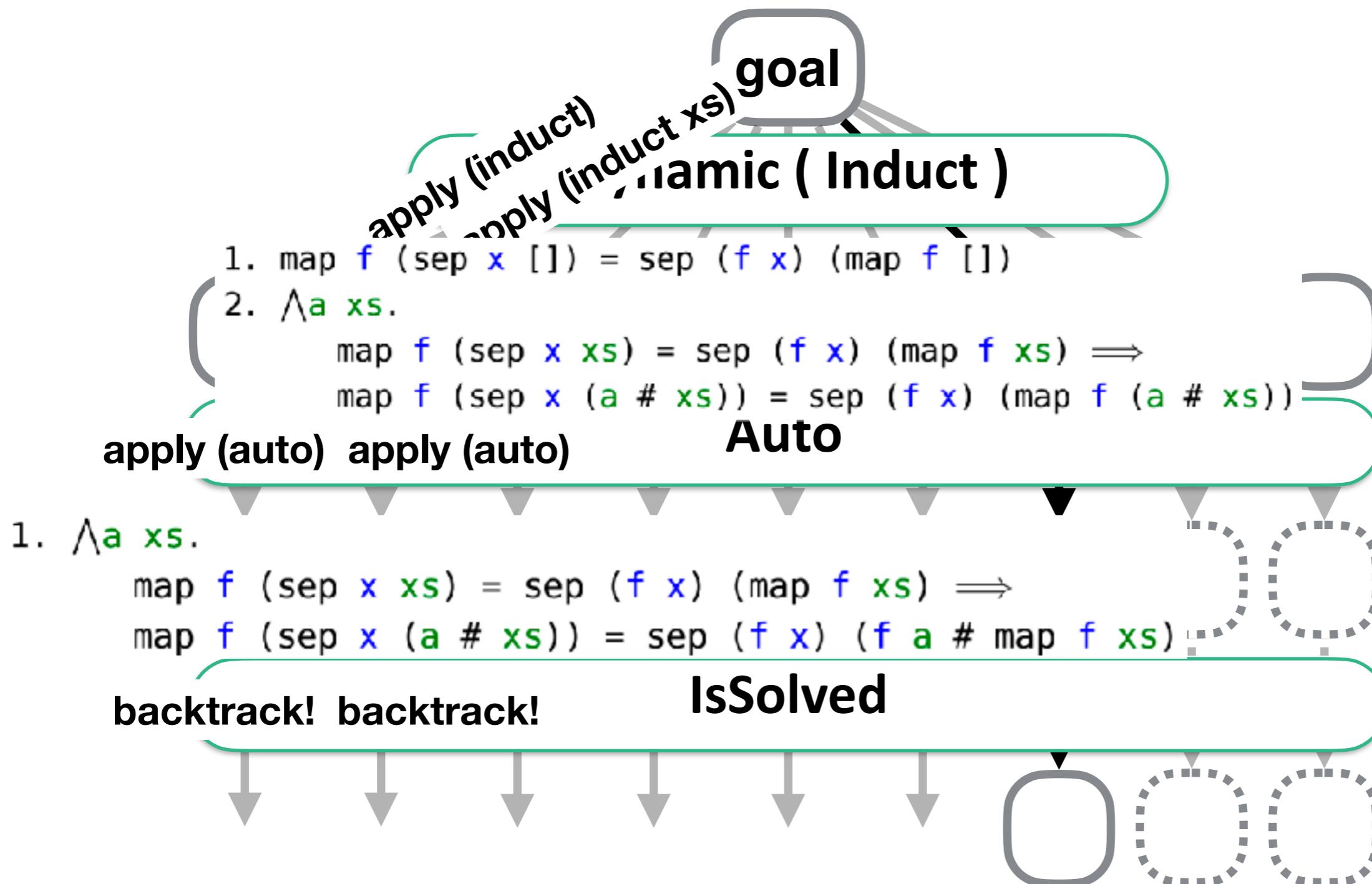
PSL: Proof Strategy Language

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



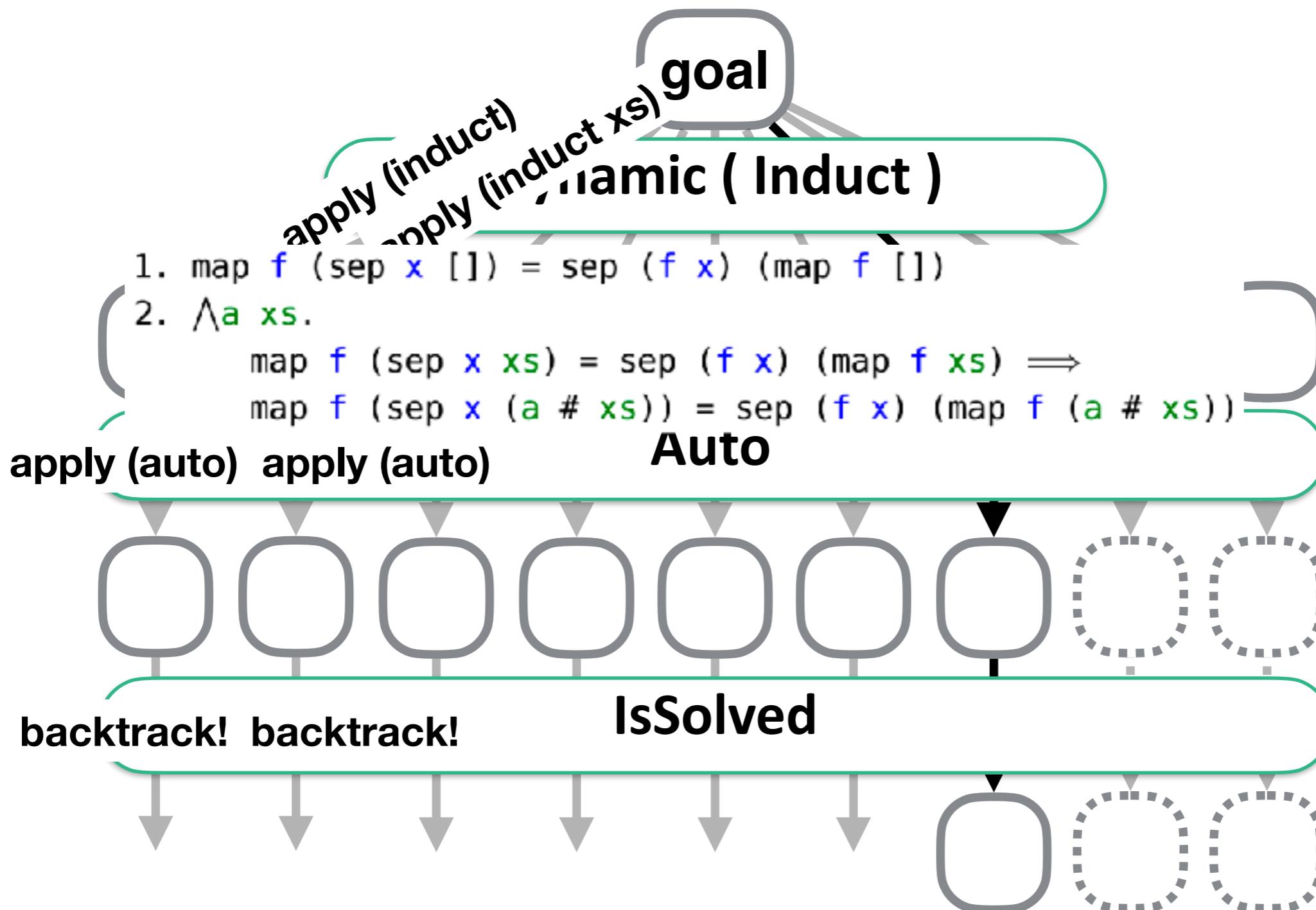
PSL: Proof Strategy Language

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



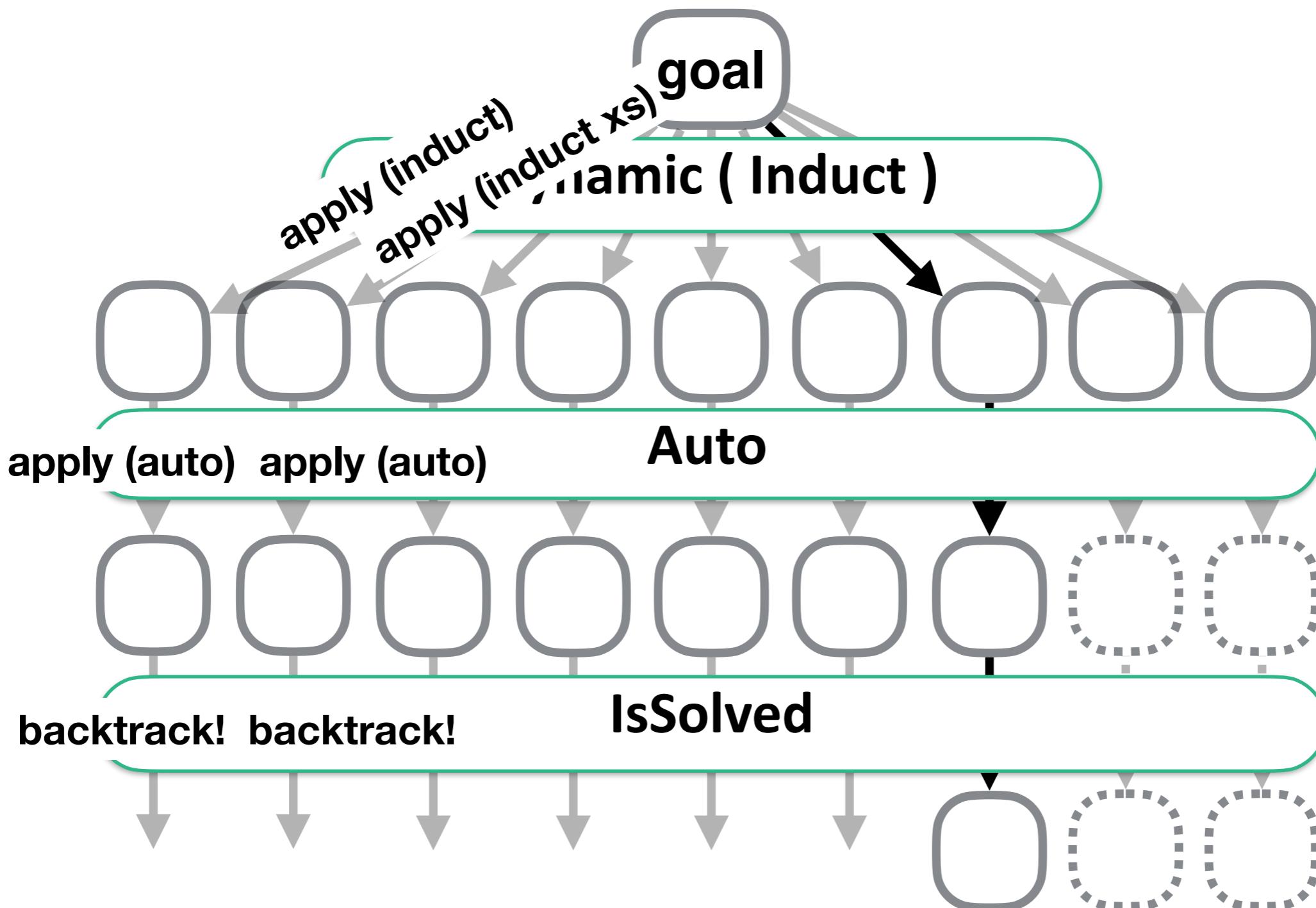
PSL: Proof Strategy Language

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



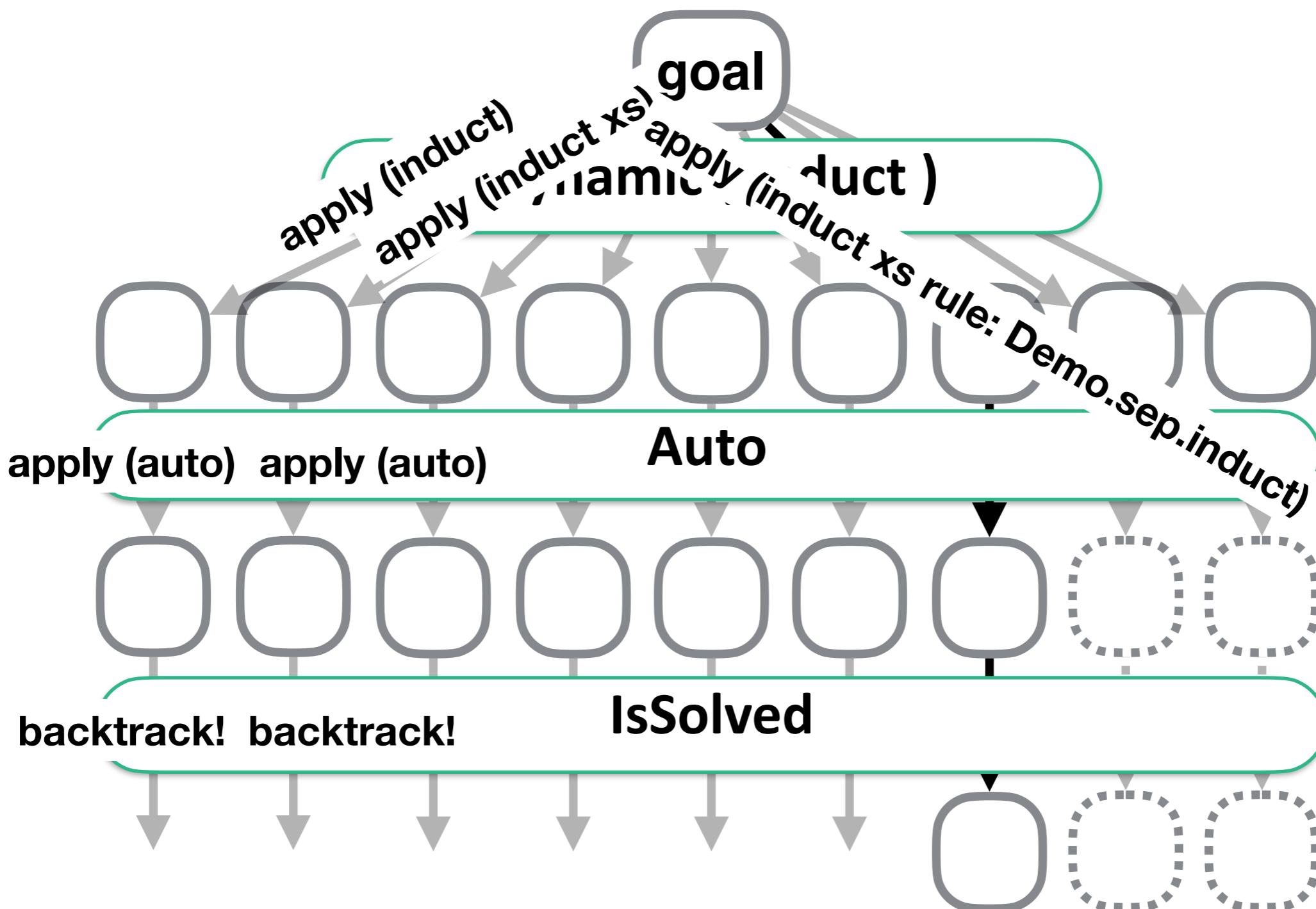
PSL: Proof Strategy Language

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



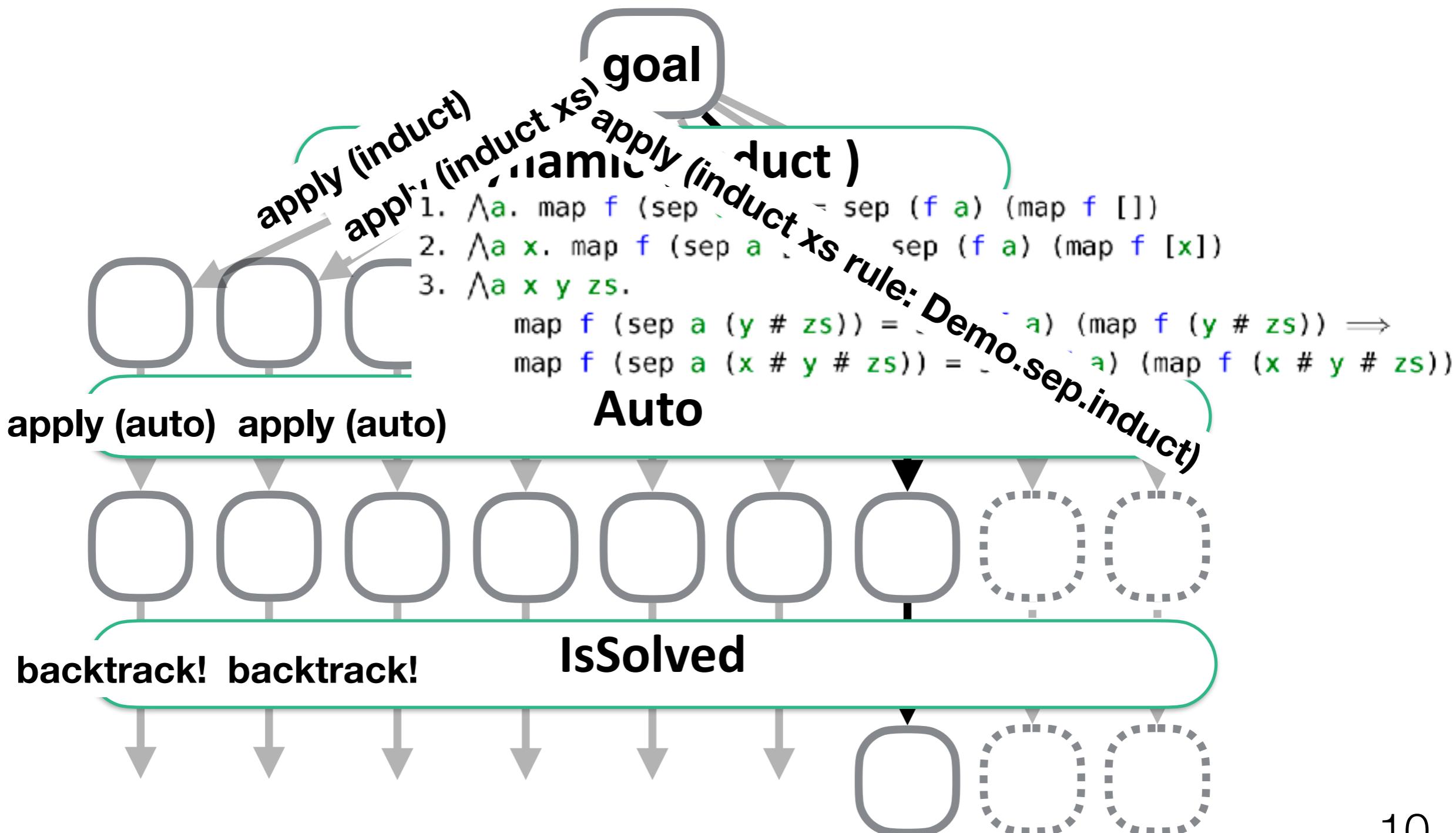
PSL: Proof Strategy Language

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



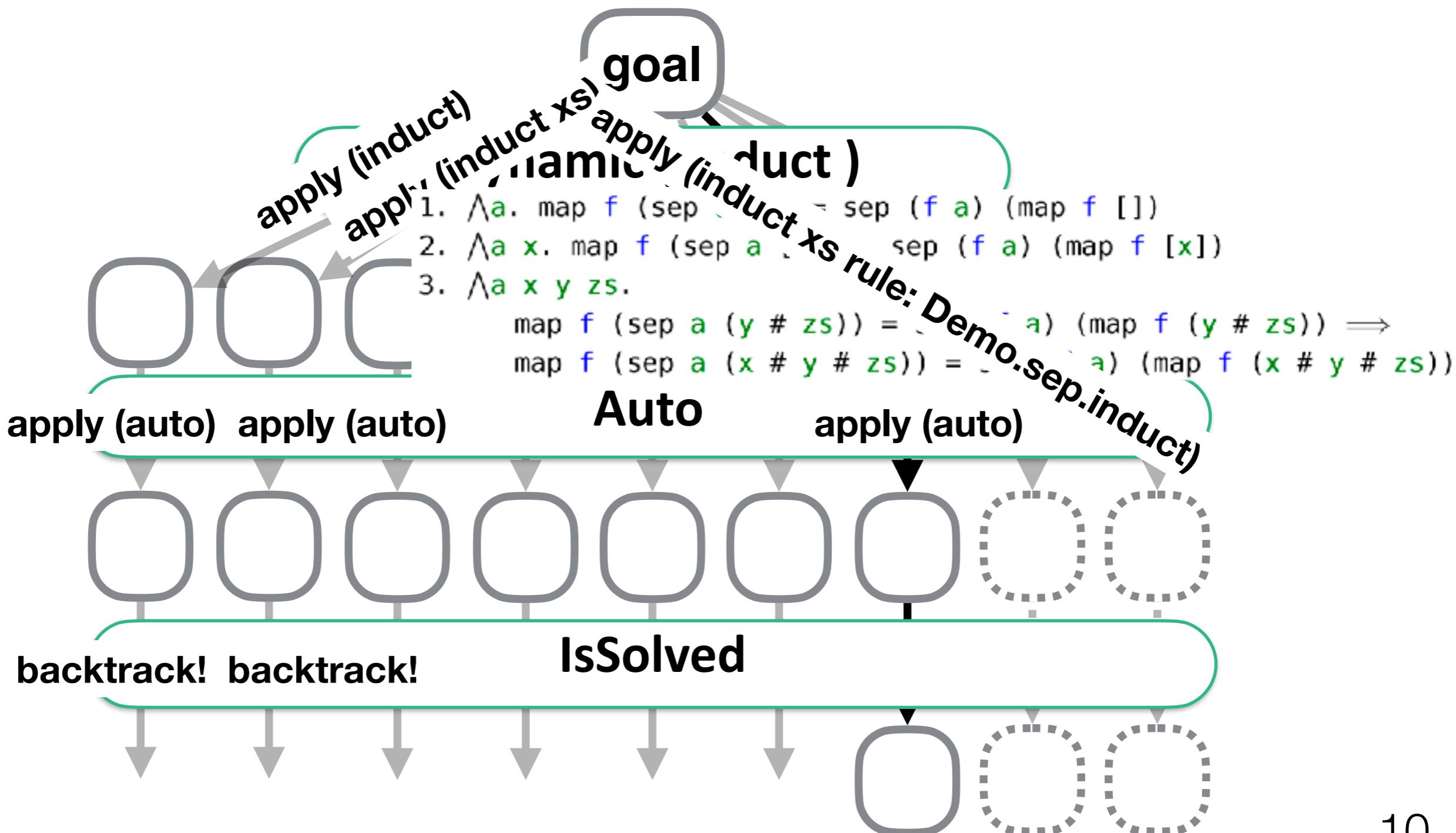
PSL: Proof Strategy Language

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



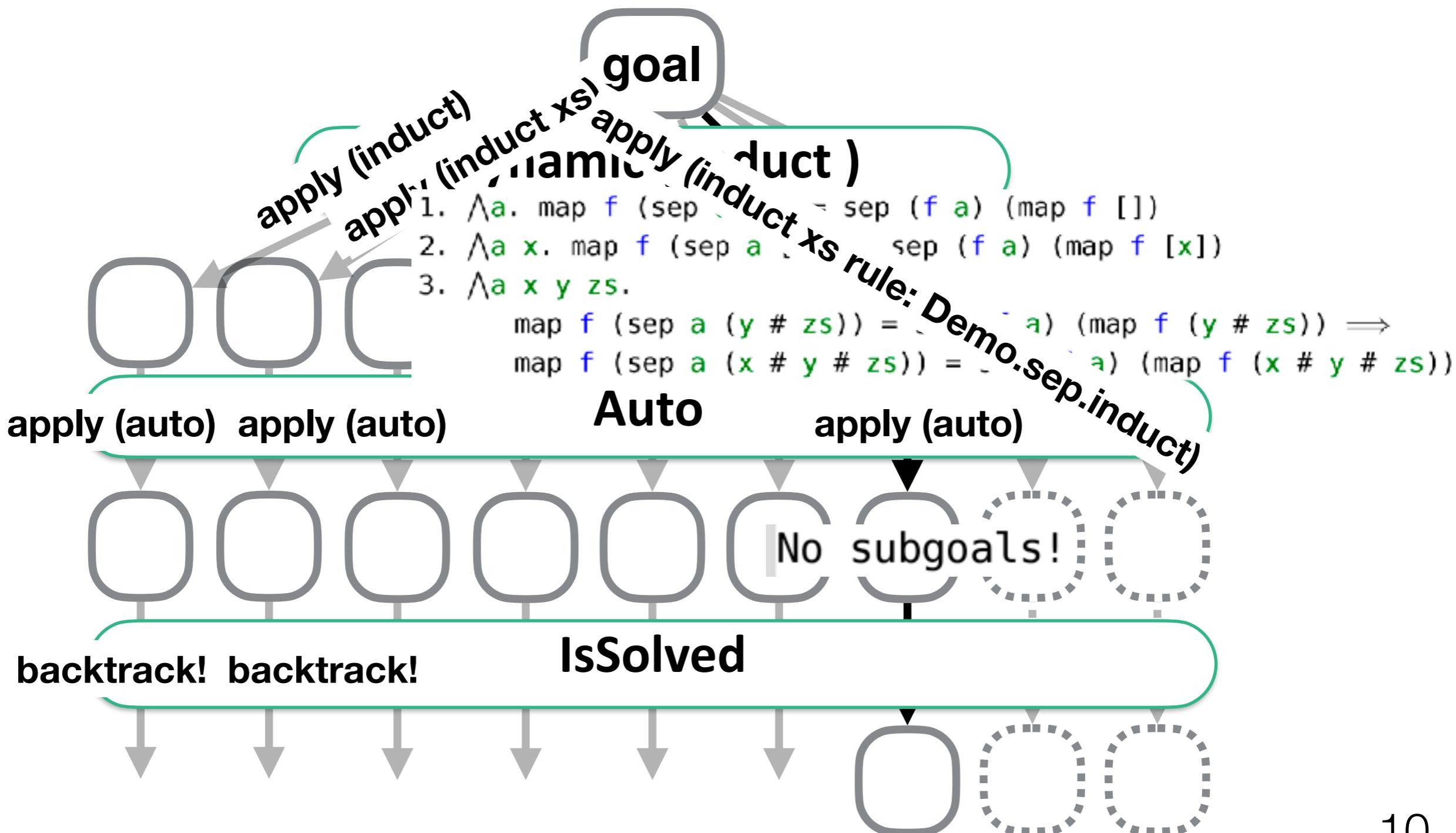
PSL: Proof Strategy Language

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



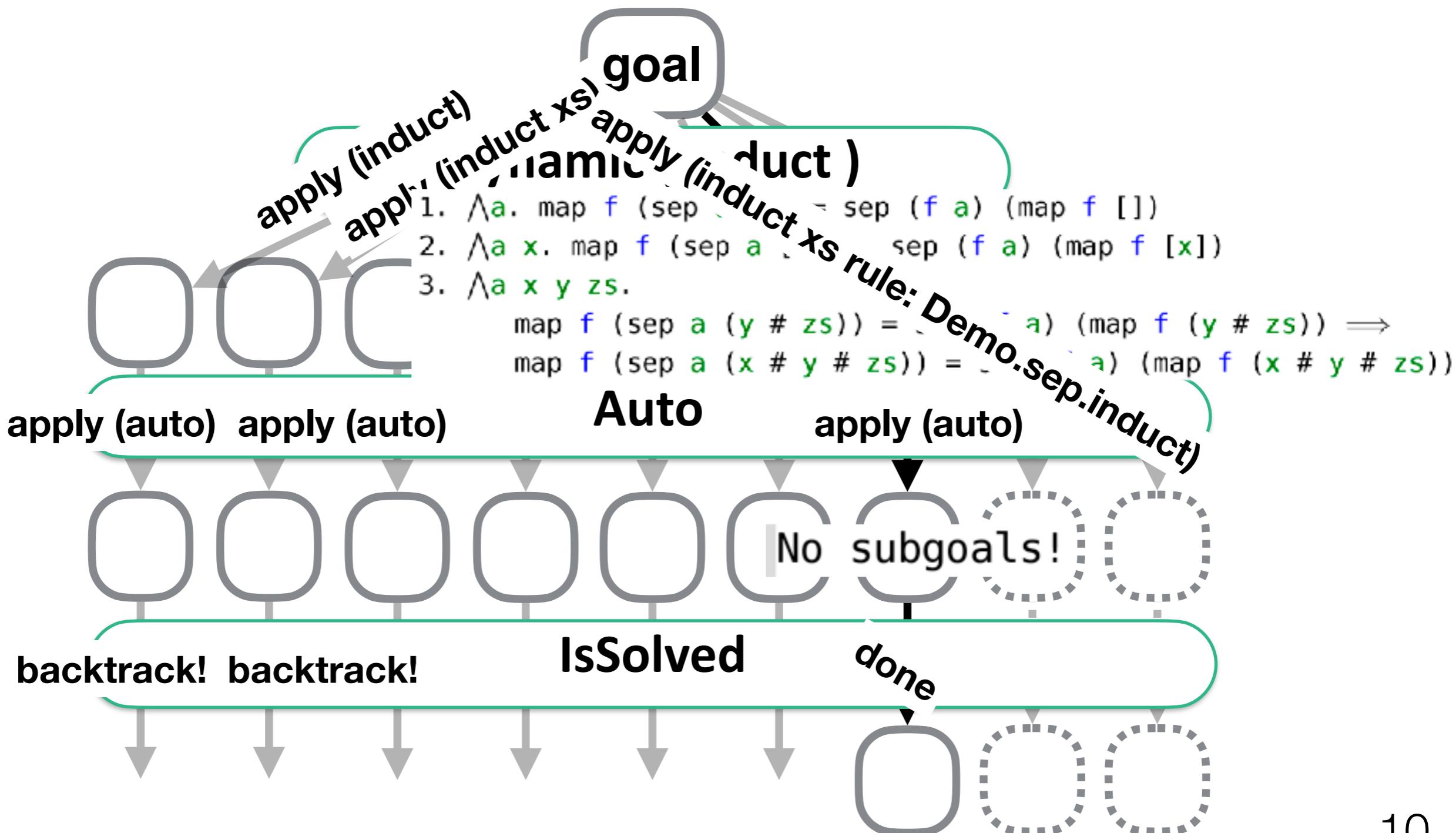
PSL: Proof Strategy Language

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



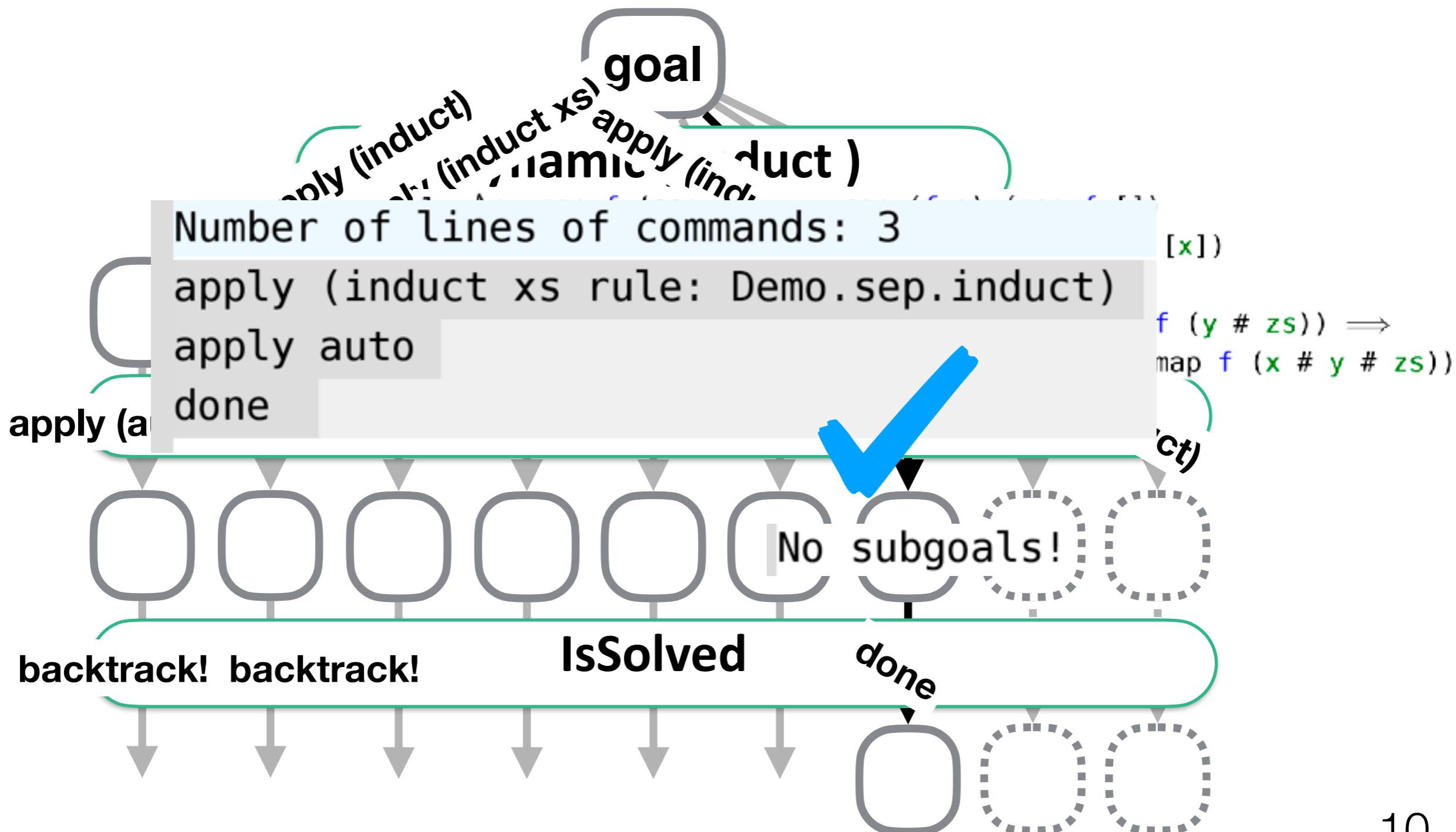
PSL: Proof Strategy Language

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



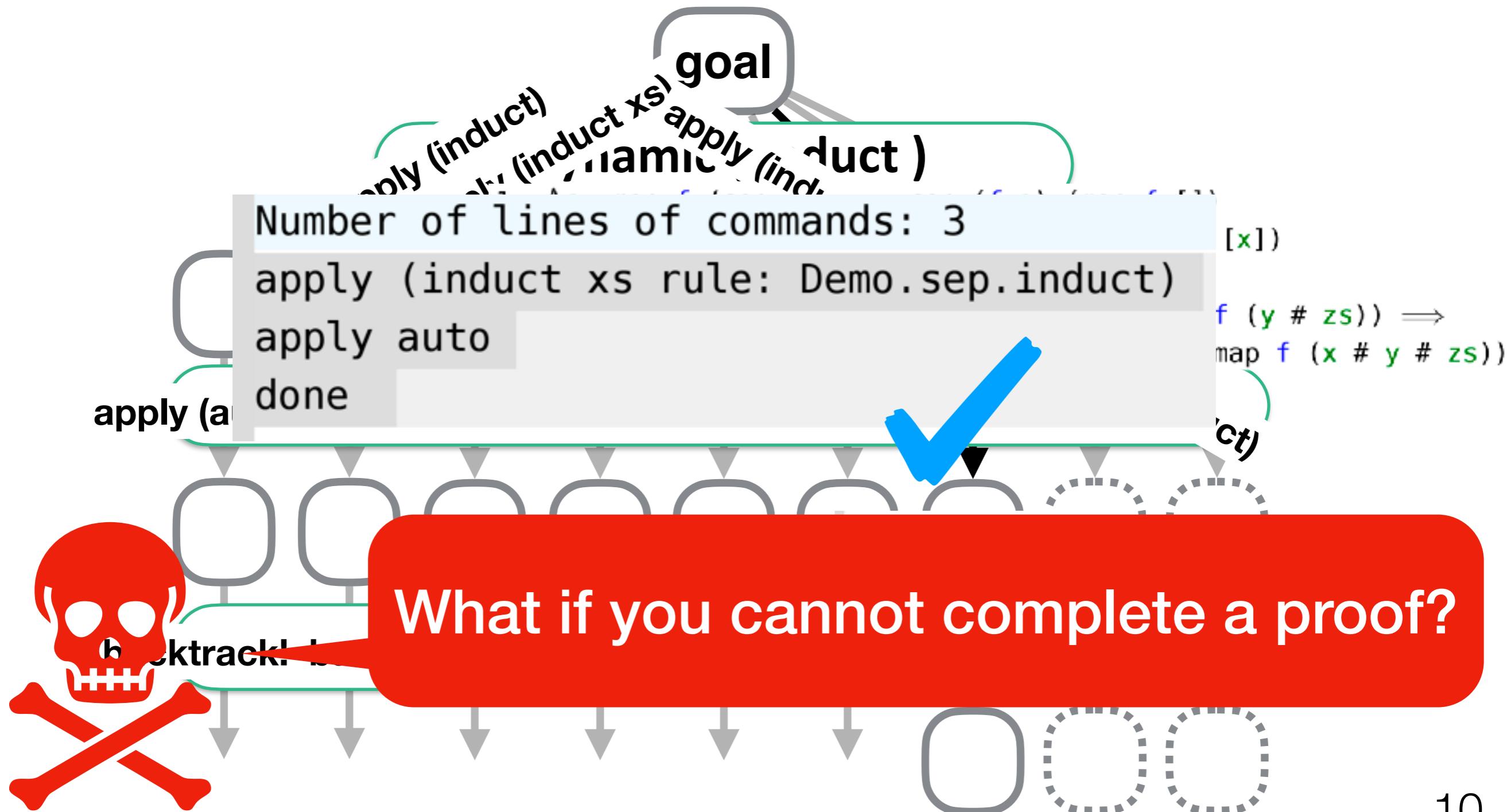
PSL: Proof Strategy Language

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



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: smart_induct

/master/slides/2020_Isabelle.pdf

The screenshot shows the Isabelle IDE interface. The main window displays a file named "Induction_Demo.thy". The code defines a function "itrev" and states a lemma. The "itrev" function is defined as follows:

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

Below the function definition, a lemma is stated:

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

The word "oops" is typed below the lemma statement.

The interface includes a toolbar at the top with icons for File, Edit, Insert, Tools, and Help. Below the toolbar is a menu bar with File, Edit, Insert, Tools, and Help. The main workspace has tabs for File Browser, Documentation, and Sidekick. On the right side, there are tabs for State and Theories. The status bar at the bottom shows the current file path, the number of lines (18), and the current time (3:24 PM).

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

DEMO2: smart_induct

/master/slides/2020_Isabelle.pdf

The screenshot shows the Isabelle proof assistant interface. The left pane displays a file named "Induction_Demo.thy" containing the following code:

```
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 oops
```

The right pane shows the proof state with the lemma being proved. The status bar at the bottom indicates "Input/output complete".

smart_induct produced 40 combinations of arguments for the induct method.
... out of which 32 of them return some results.

... out of which only 32 of them passes the second screening stage.

LiFtEr assertions are evaluating the first 32 of them.

Try these 10 most promising inductions!

1st candidate is apply (induct xs arbitrary: ys)

(* The score is 21 out of 21. *)

2nd candidate is apply (induct xs)

(* The score is 21 out of 21. *)

3rd candidate is apply (induct xs ys rule: Induction_Demo.itrev.induct)

(* The score is 19 out of 21. *)

DEMO2: smart_induct

/master/slides/2020_Isabelle.pdf

The screenshot shows the Isabelle proof assistant interface. The main window displays a theory file named "Induction_Demo.thy". The code defines a function `itrev` and a lemma. The `itrev` function is defined by induction on its first argument. The lemma states that `itrev xs ys = rev xs @ ys`. The proof script uses `smart_induct` for the function definition and `apply auto` for the lemma, with the final step being `done`.

```
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 (induct xs arbitrary: ys) apply auto done
20
21
22
23
24
25
26
27
28
```

Proof state Auto update Update Search: 100% C

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

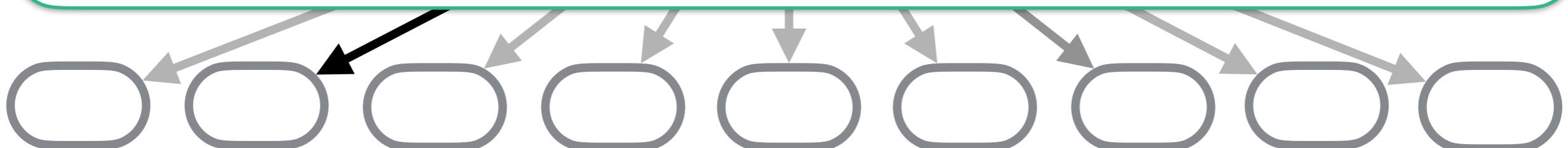
Output Query Sledgehammer Symbols

19.45 (606/891) (isabelle, isabelle, UTF-8-isabelle) | nmrc U.. 227/512MB 3:23 PM

goal

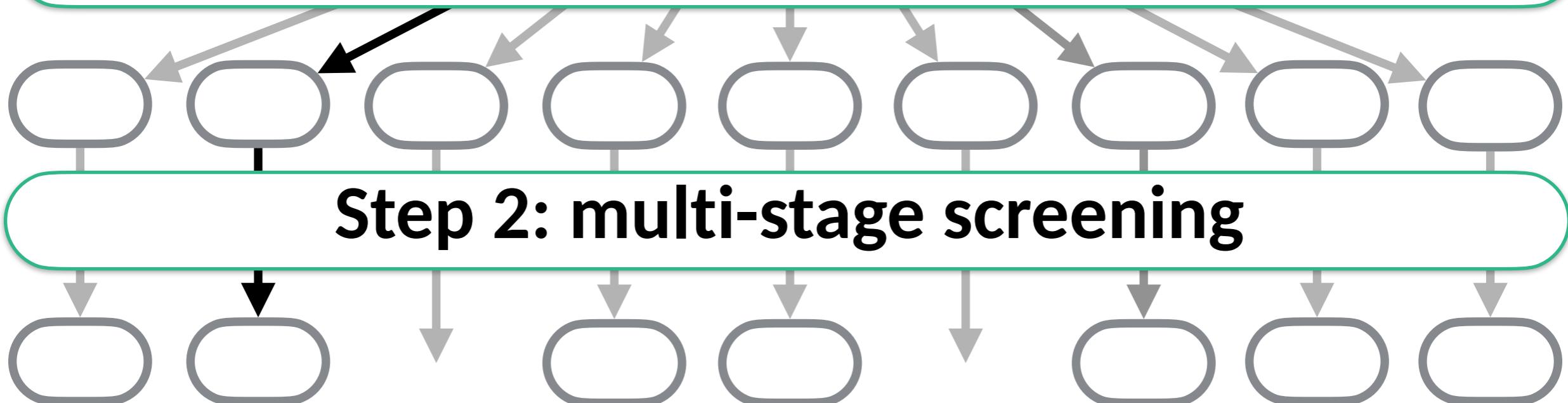
goal

Step 1: creating many inductions



goal

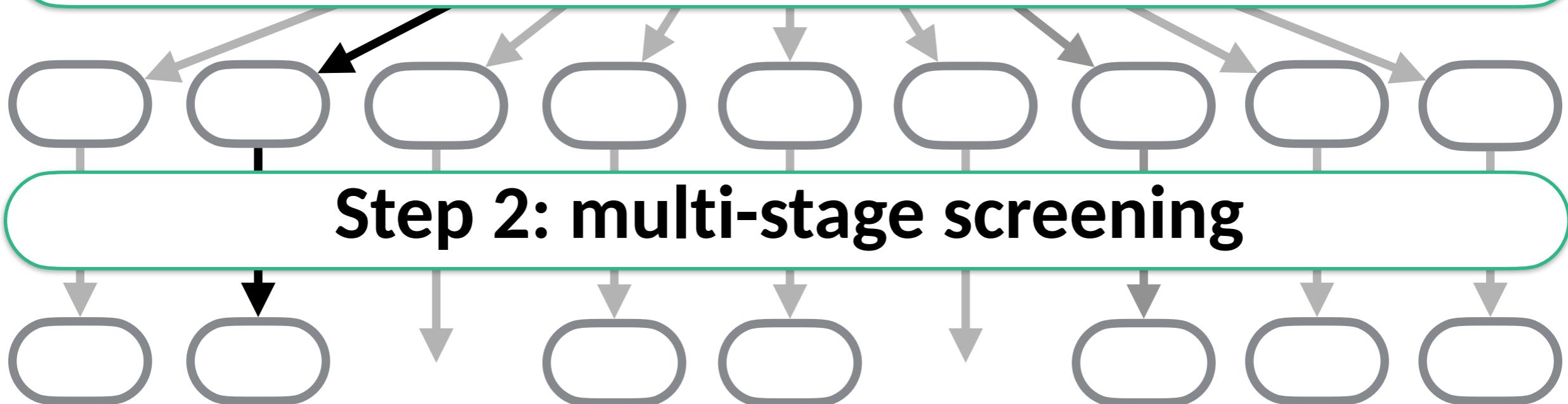
Step 1: creating many inductions



Step 2: multi-stage screening

goal

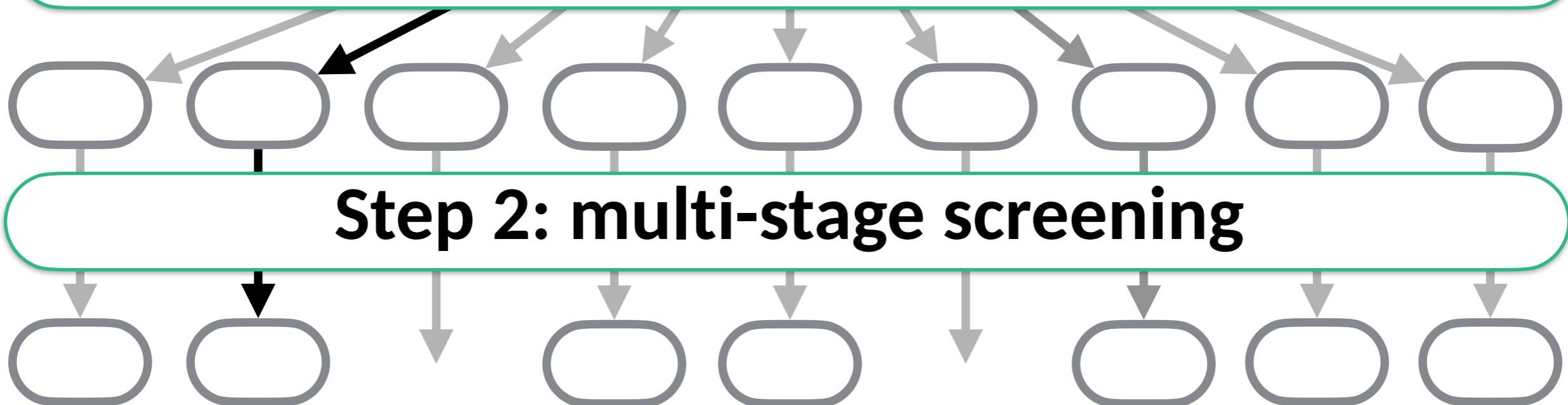
Step 1: creating many inductions



Step 3: scoring using 20 heuristics and sorting

goal

Step 1: creating many inductions



Step 2: multi-stage screening

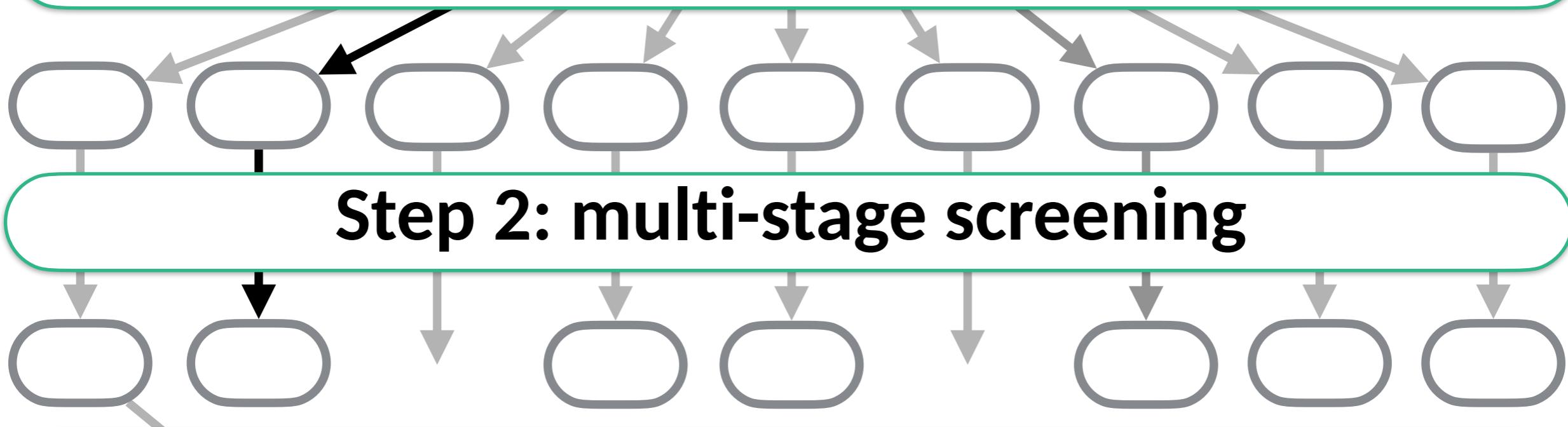


Step 3: scoring using 20 heuristics and sorting

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

goal

Step 1: creating many inductions



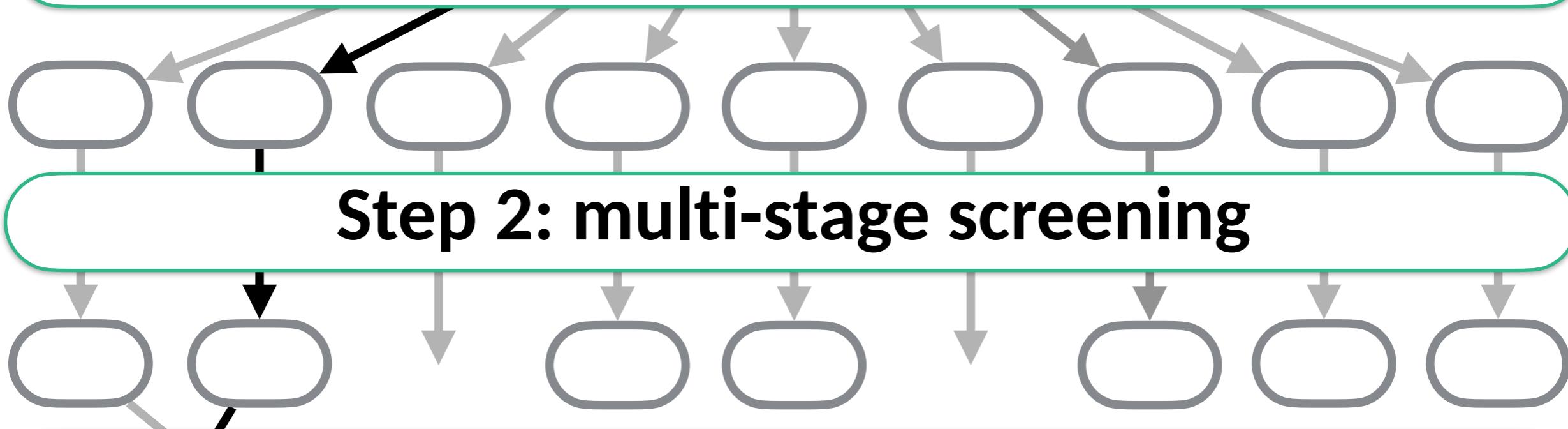
Step 2: multi-stage screening

Step 3: scoring using 20 heuristics and sorting

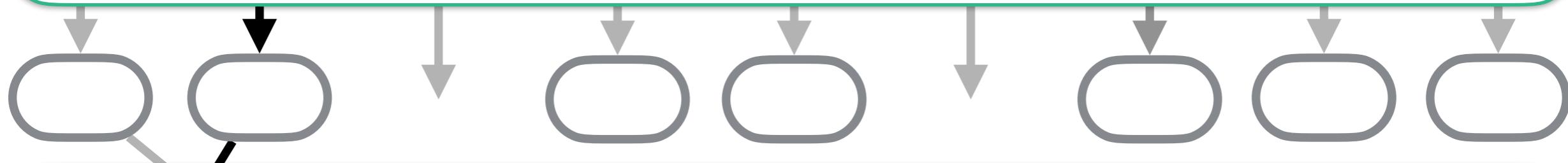
heuristic : (proof goal * induction arguments) -> bool

goal

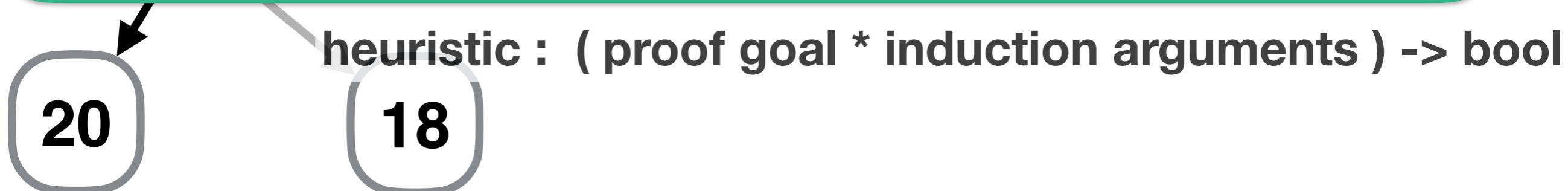
Step 1: creating many inductions



Step 2: multi-stage screening

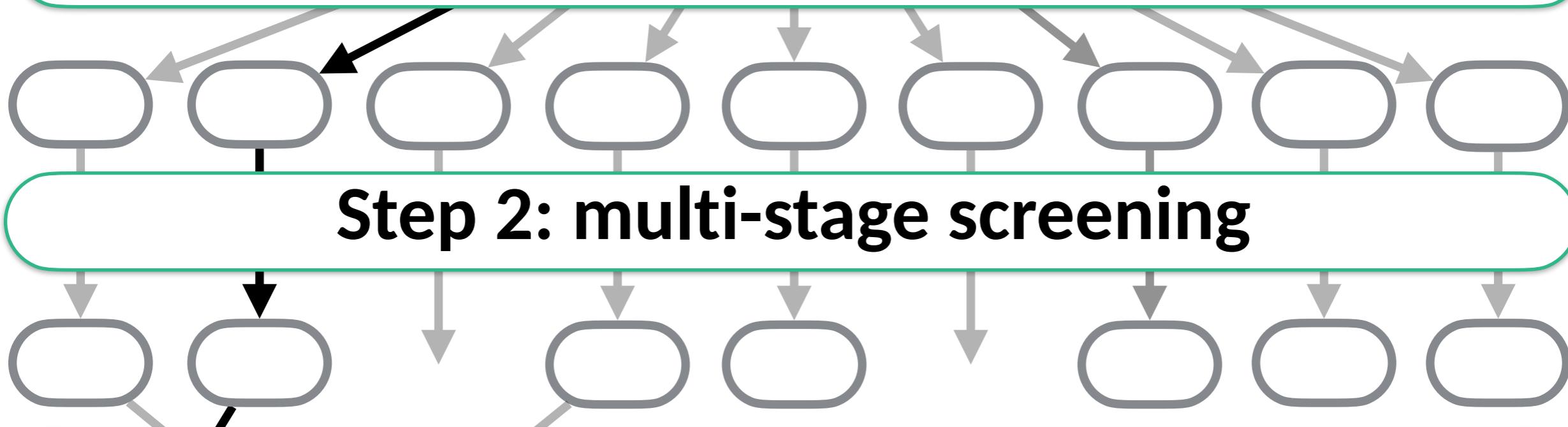


Step 3: scoring using 20 heuristics and sorting

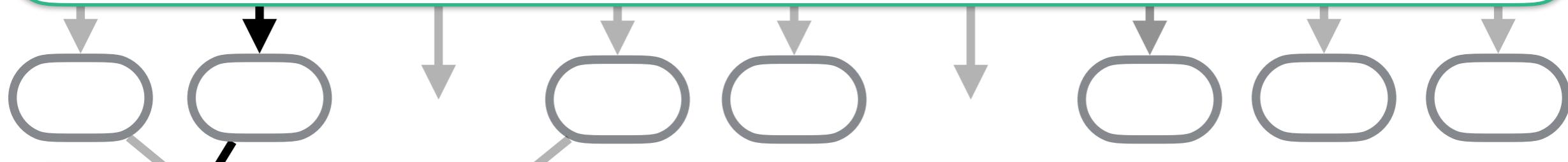


goal

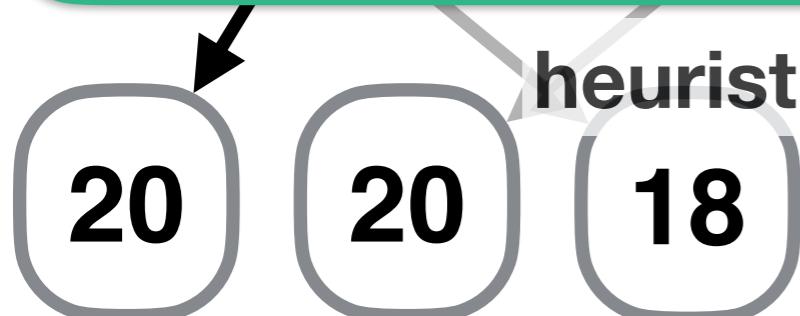
Step 1: creating many inductions



Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting



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

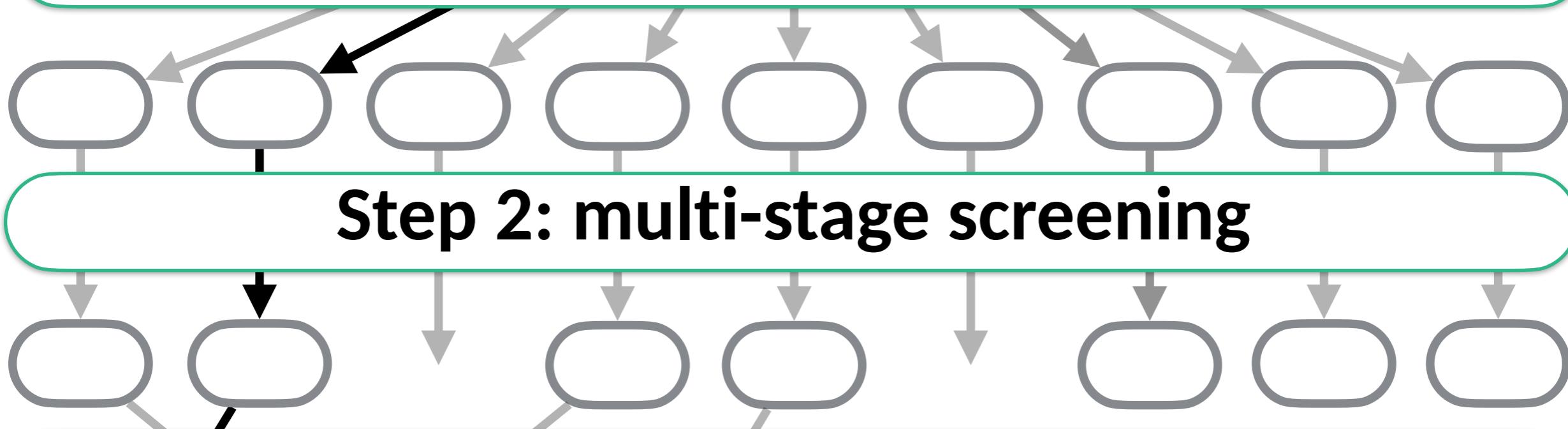
20

20

18

goal

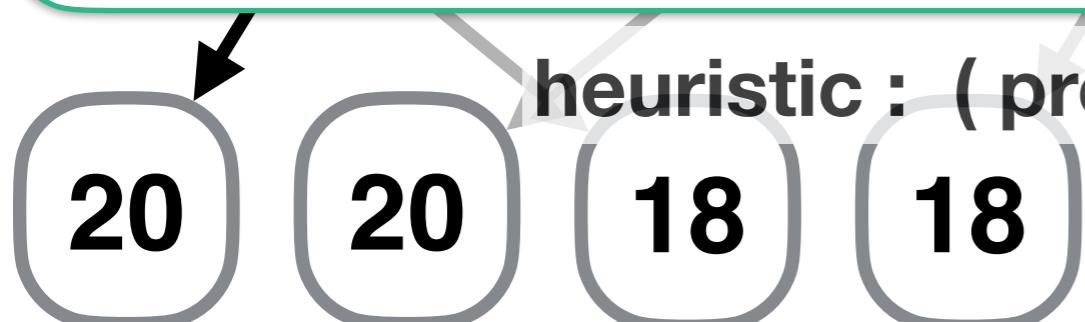
Step 1: creating many inductions



Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting



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

20

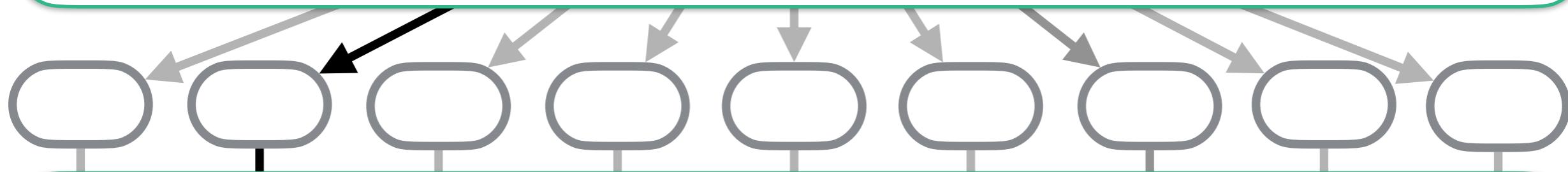
20

18

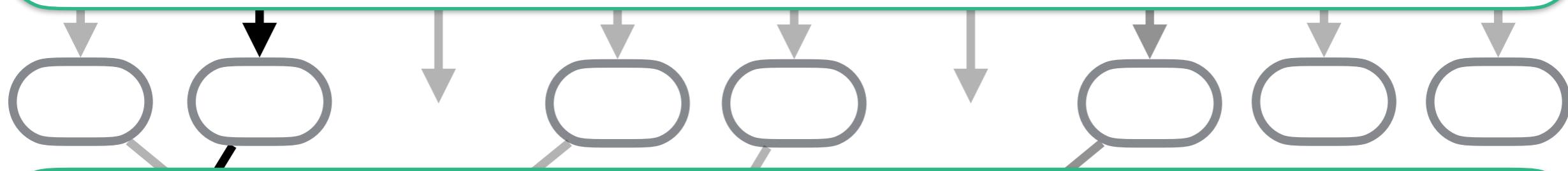
18

goal

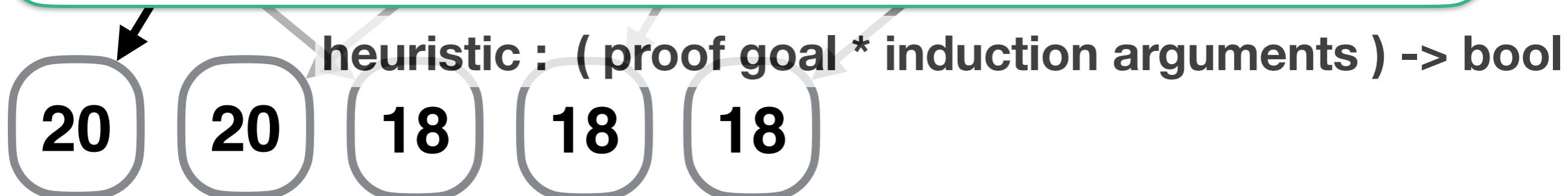
Step 1: creating many inductions



Step 2: multi-stage screening

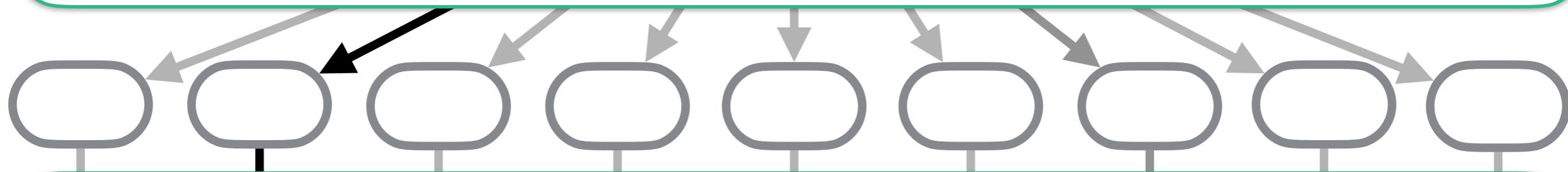


Step 3: scoring using 20 heuristics and sorting

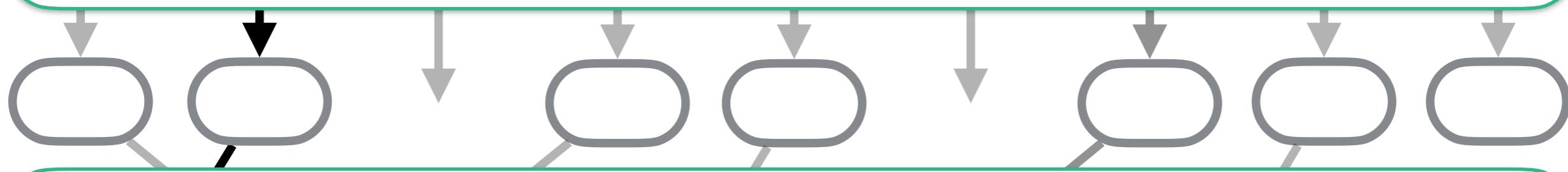


goal

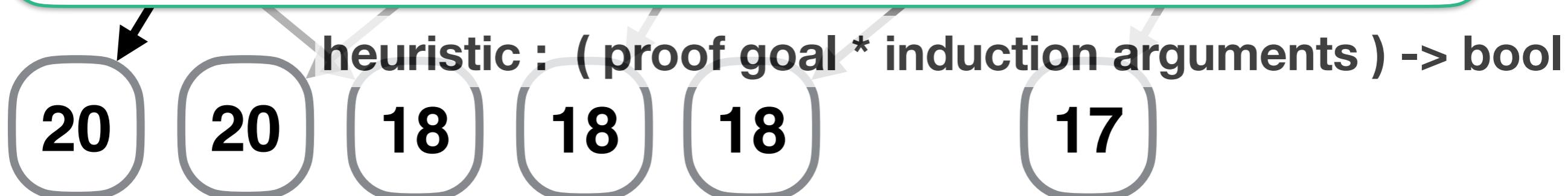
Step 1: creating many inductions

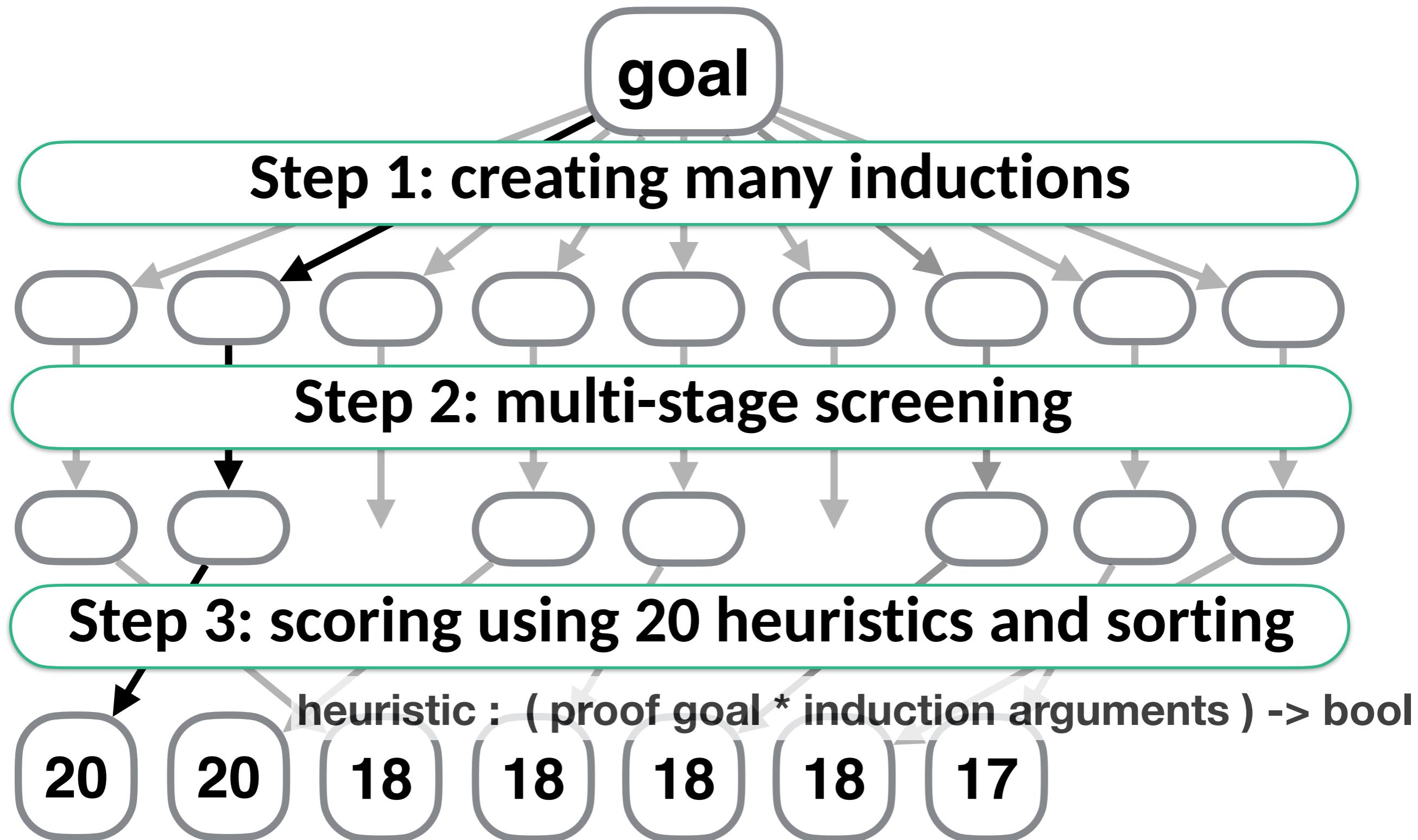


Step 2: multi-stage screening



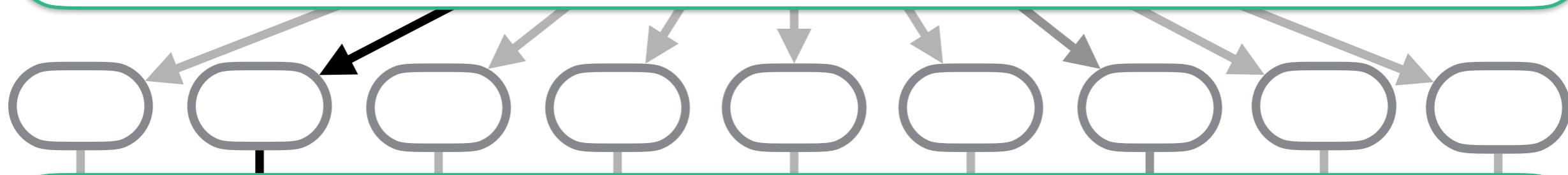
Step 3: scoring using 20 heuristics and sorting



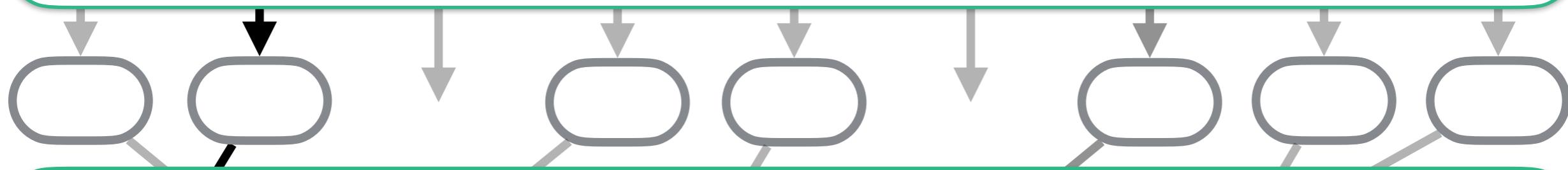


goal

Step 1: creating many inductions



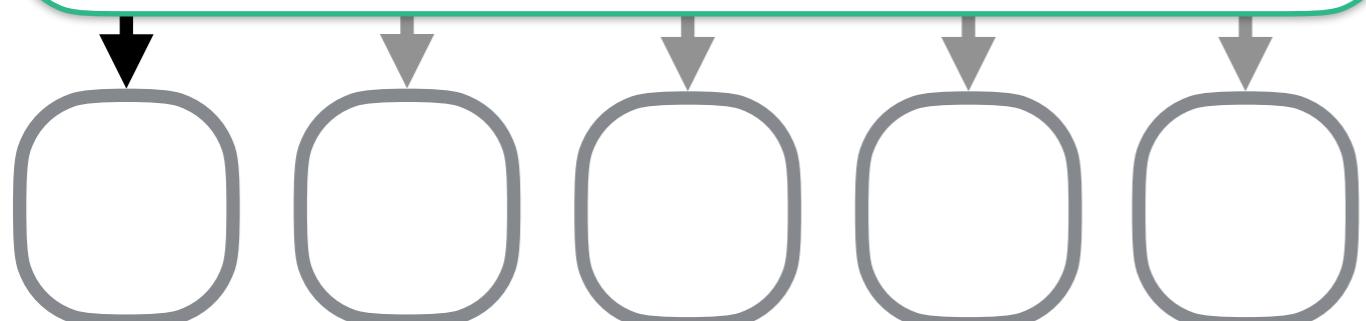
Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting

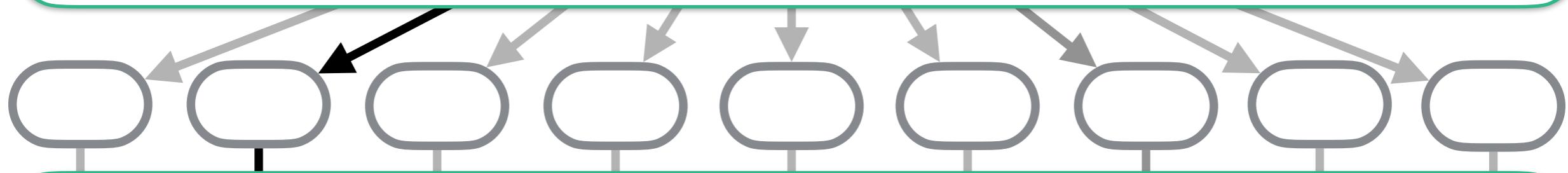


Step 4: short-listing

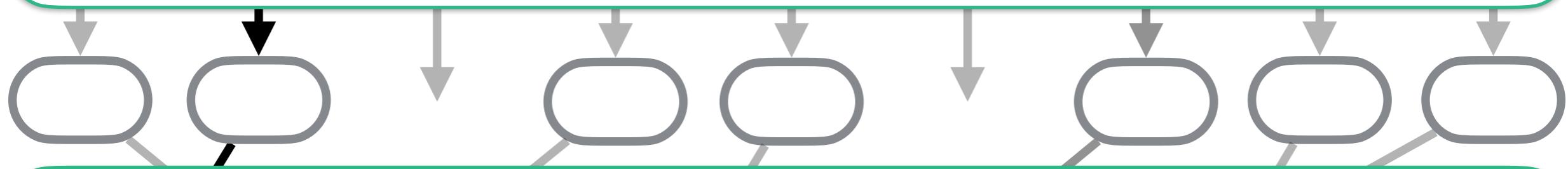


goal

Step 1: creating many inductions



Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting



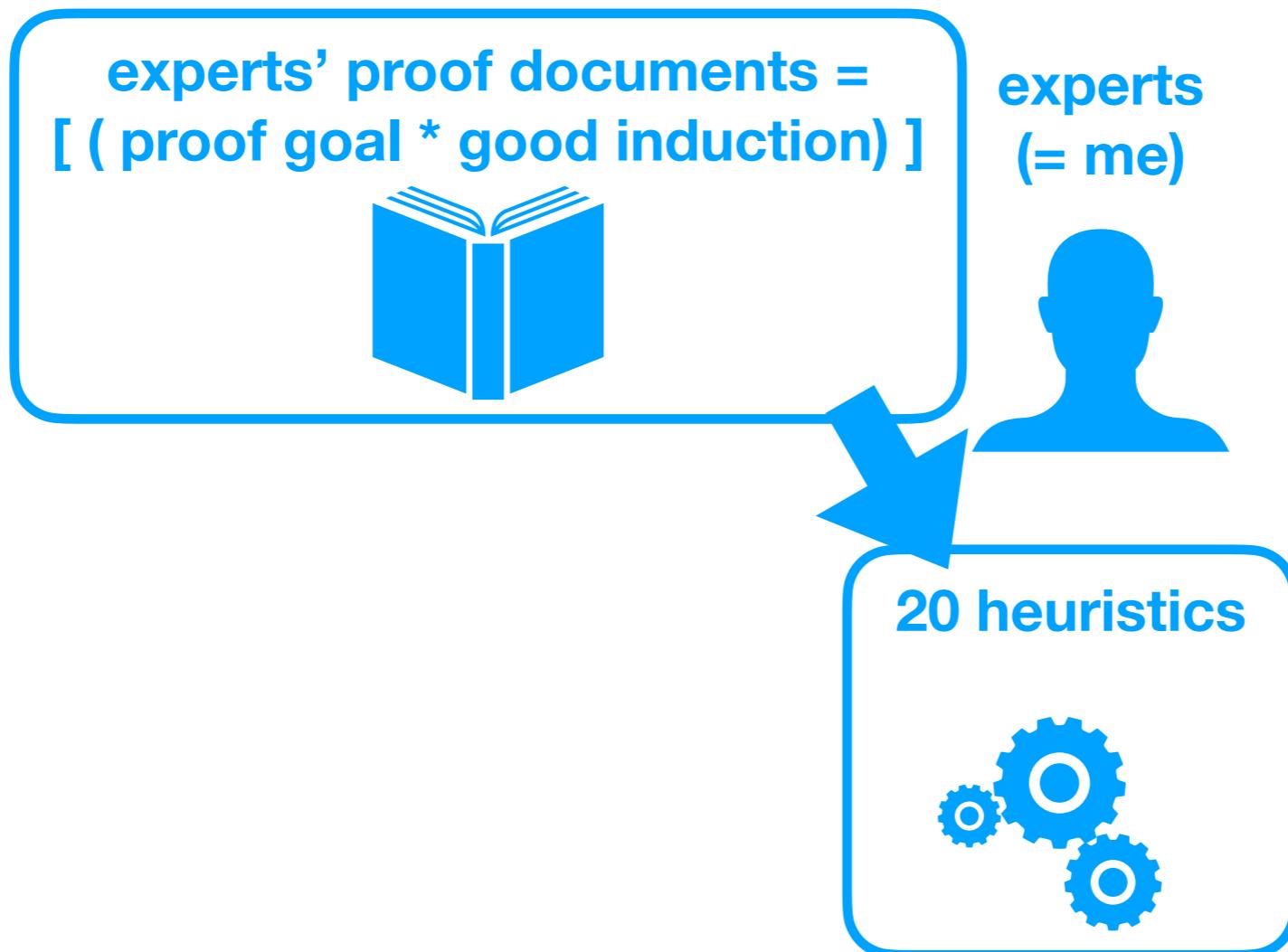
Step 4: short-listing

So, what heuristics do you use?



It is difficult to write induction heuristics.

It is difficult to write induction heuristics.



It is difficult to write induction heuristics.

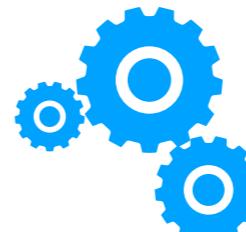
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

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



experts
(= me)

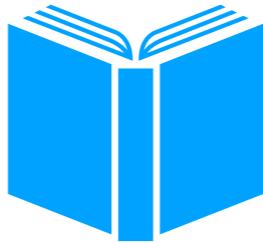
20 heuristics



It is difficult to write induction heuristics.

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

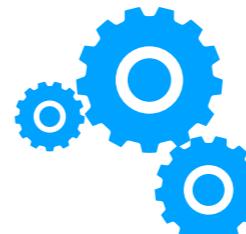
experts' proof documents =
[(proof goal * good induction)]



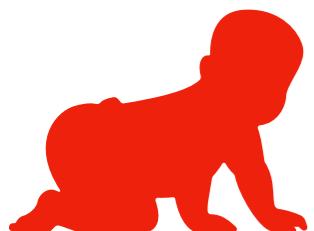
experts
(= me)



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)]



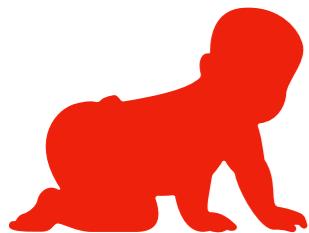
experts
(= me)



20 heuristics



new users



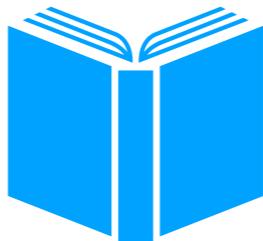
```
lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
```

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
```

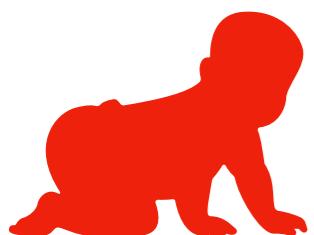
experts' proof documents =
[(proof goal * good induction)]



experts
(= me)

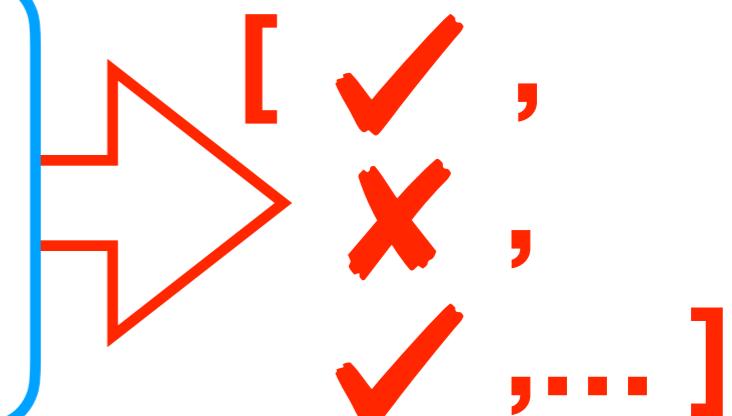
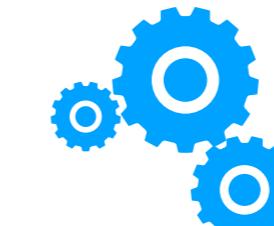


new users



```
proof goal candidate induction  
apply (  
induct is1 s stk  
rule: exec.induct)
```

20 heuristics



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

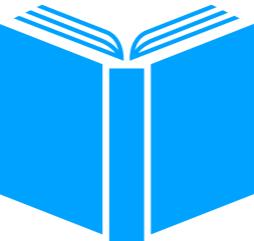
new proof goal consisting of new constants and variables of new types!

It is difficult to write induction heuristics.

① blue = what I can see before releasing smart_induct

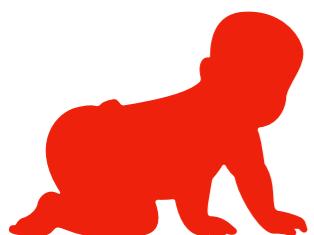
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

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

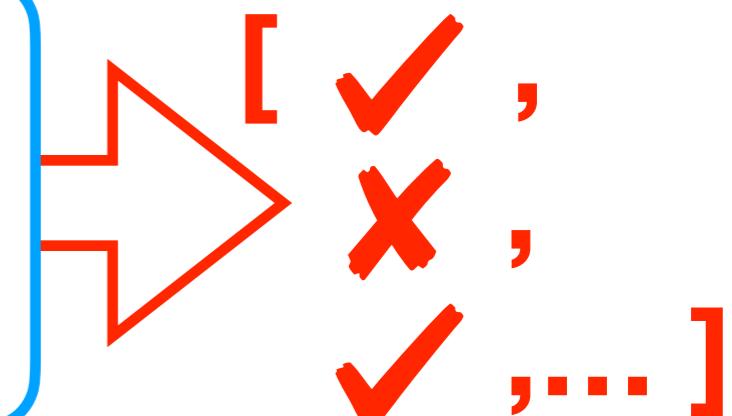
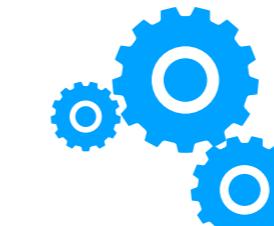


experts
(= me)



new users

proof goal candidate induction
apply (
induct is1 s stk
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

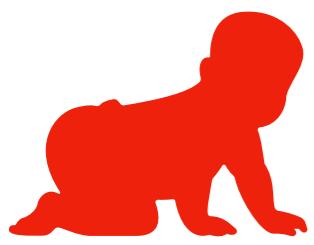
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

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

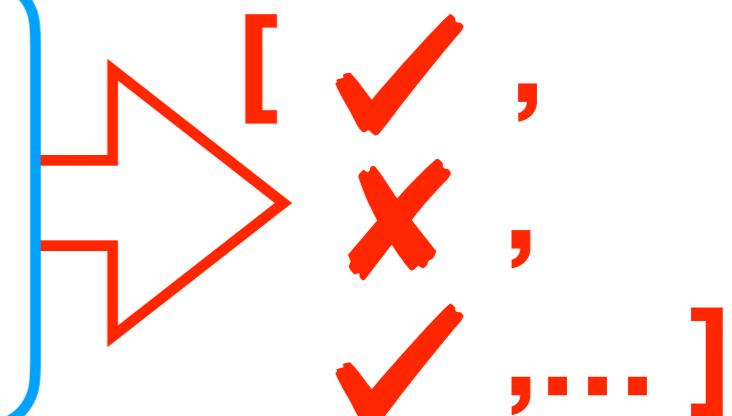
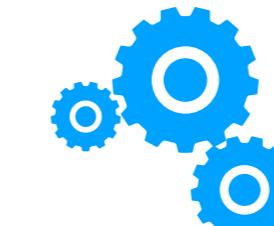


experts
(= me)



new users

proof goal candidate induction
apply (
induct is1 s stk
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!

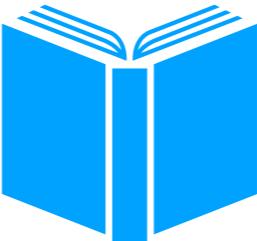
② red = what will be developed after releasing smart_induct

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
```

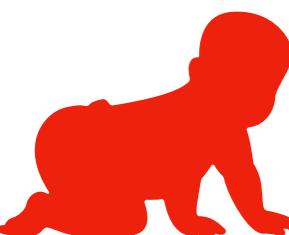
experts' proof documents =
[(proof goal * good induction)]



experts
(= me)

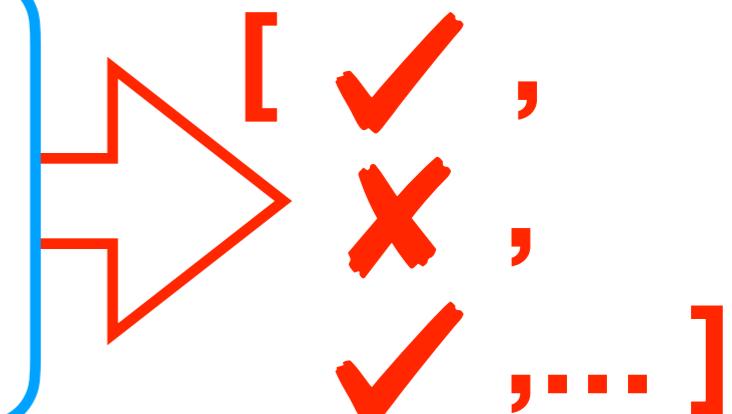
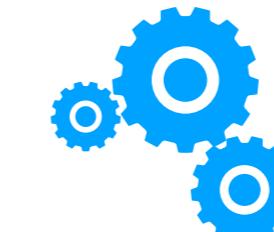


new users



proof goal candidate induction
apply (
induct is1 s stk
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!

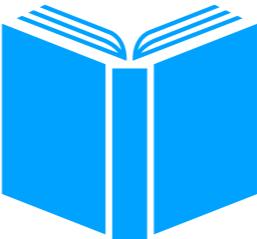
② red = what will be developed after releasing smart_induct

It is difficult to write induction heuristics.

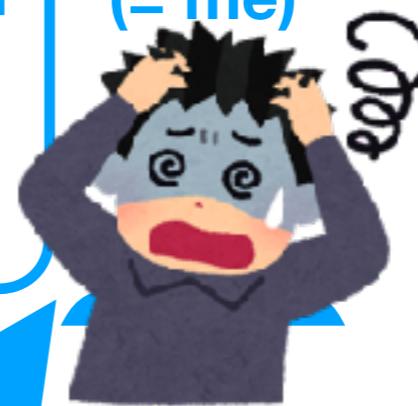
① blue = what I can see before releasing smart_induct

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

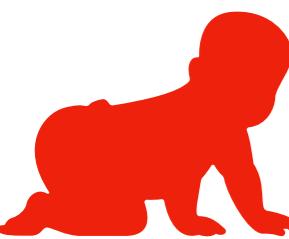
experts' proof documents =
[(proof goal * good induction)]



experts
(= me)

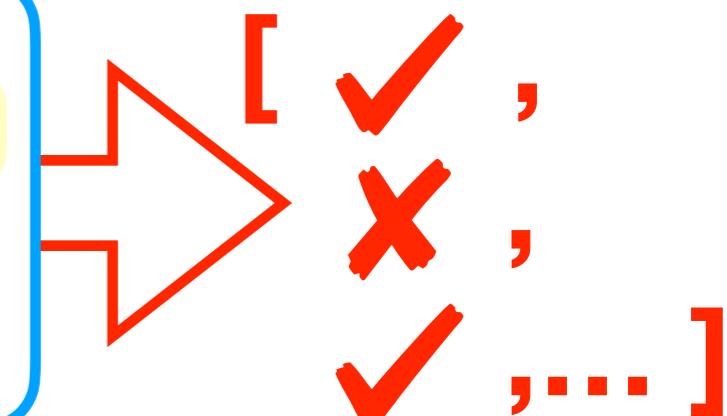


new users



proof goal candidate induction
apply (
induct is1 s stk
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!

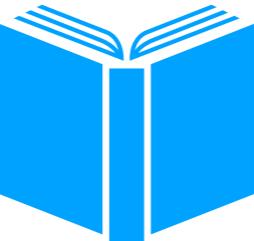
② red = what will be developed after releasing smart_induct

It is difficult to write induction heuristics.

① blue = what I can see before releasing smart_induct

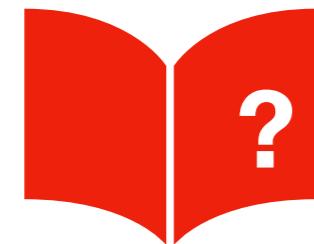
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

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

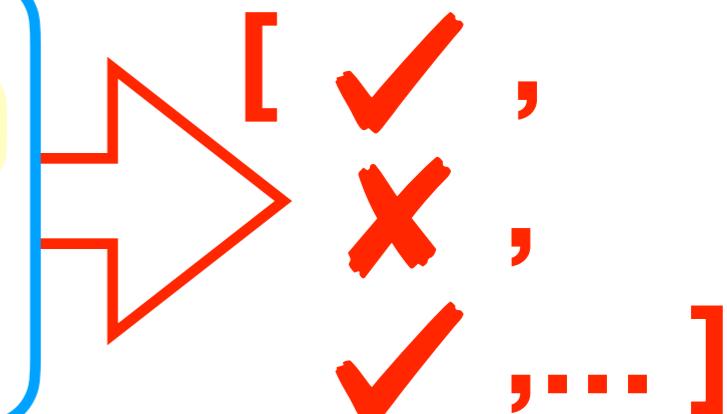


new users

proof goal candidate induction
apply (
induct is1 s stk
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!

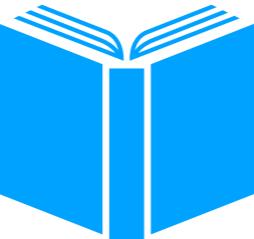
② red = what will be developed after releasing smart_induct

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
```

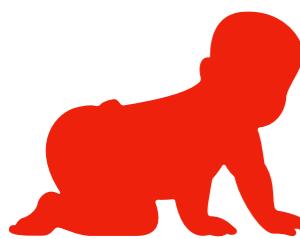
experts' proof documents =
[(proof goal * good induction)]



[https://doi.org/
10.1007/978-3-030-34175-6_14](https://doi.org/10.1007/978-3-030-34175-6_14)

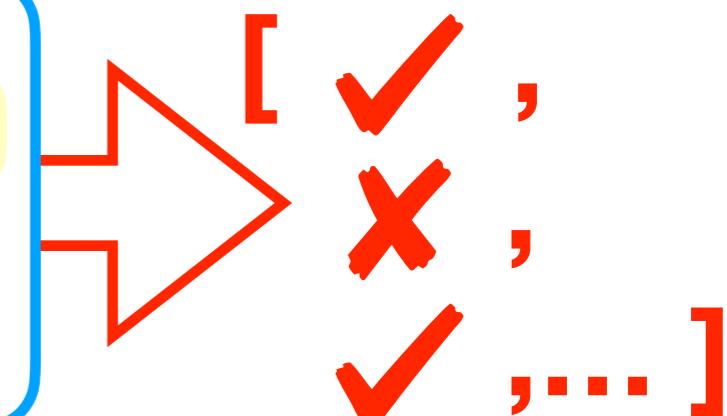


new users



proof goal candidate induction
apply (
induct is1 s stk
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!

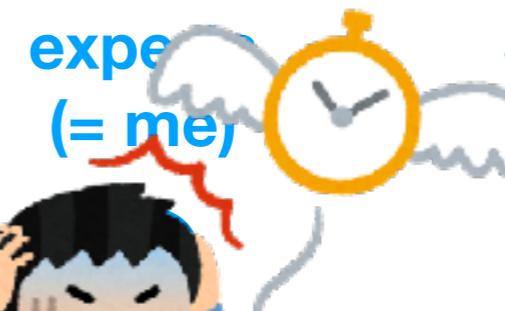
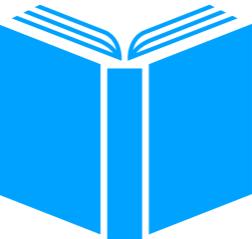
② red = what will be developed after releasing smart_induct

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)]

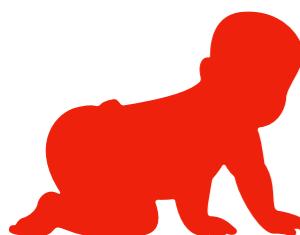


[https://doi.org/
10.1007/978-3-030-34175-6_14](https://doi.org/10.1007/978-3-030-34175-6_14)

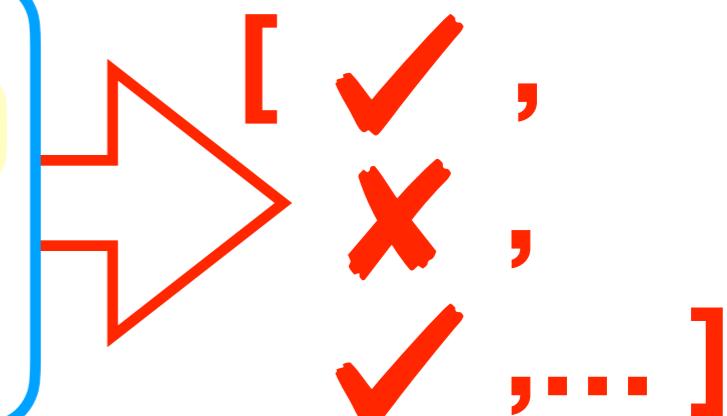
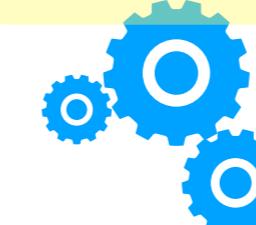


new users

proof goal candidate induction
apply (
induct is1 s stk
rule: exec.induct)



20 heuristics
in LiFtEr



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

②

Does it (smart_induct) really work?

types!



It is difficult to write induction heuristics.

① blue = what I can see before releasing smart_induct

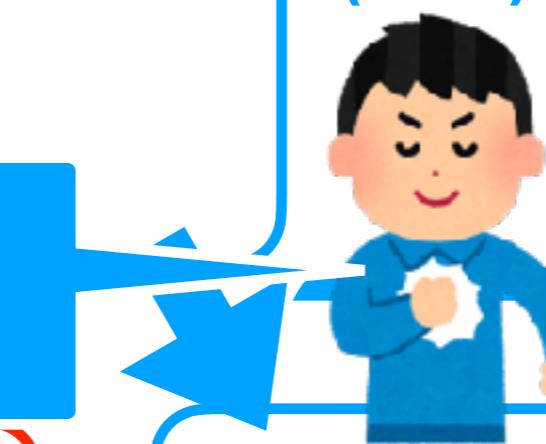
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

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



YES!

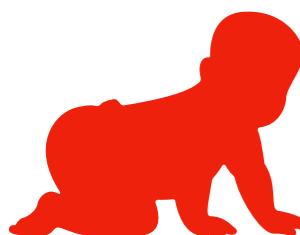
experts
(= me)



[https://doi.org/
10.1007/978-3-030-34175-6_14](https://doi.org/10.1007/978-3-030-34175-6_14)



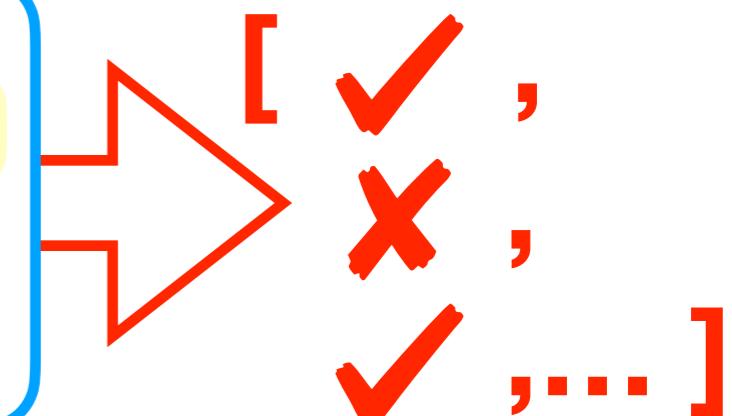
new users



proof goal candidate induction
apply (
induct is1 s stk
rule: exec.induct)



20 heuristics
in LiFtEr



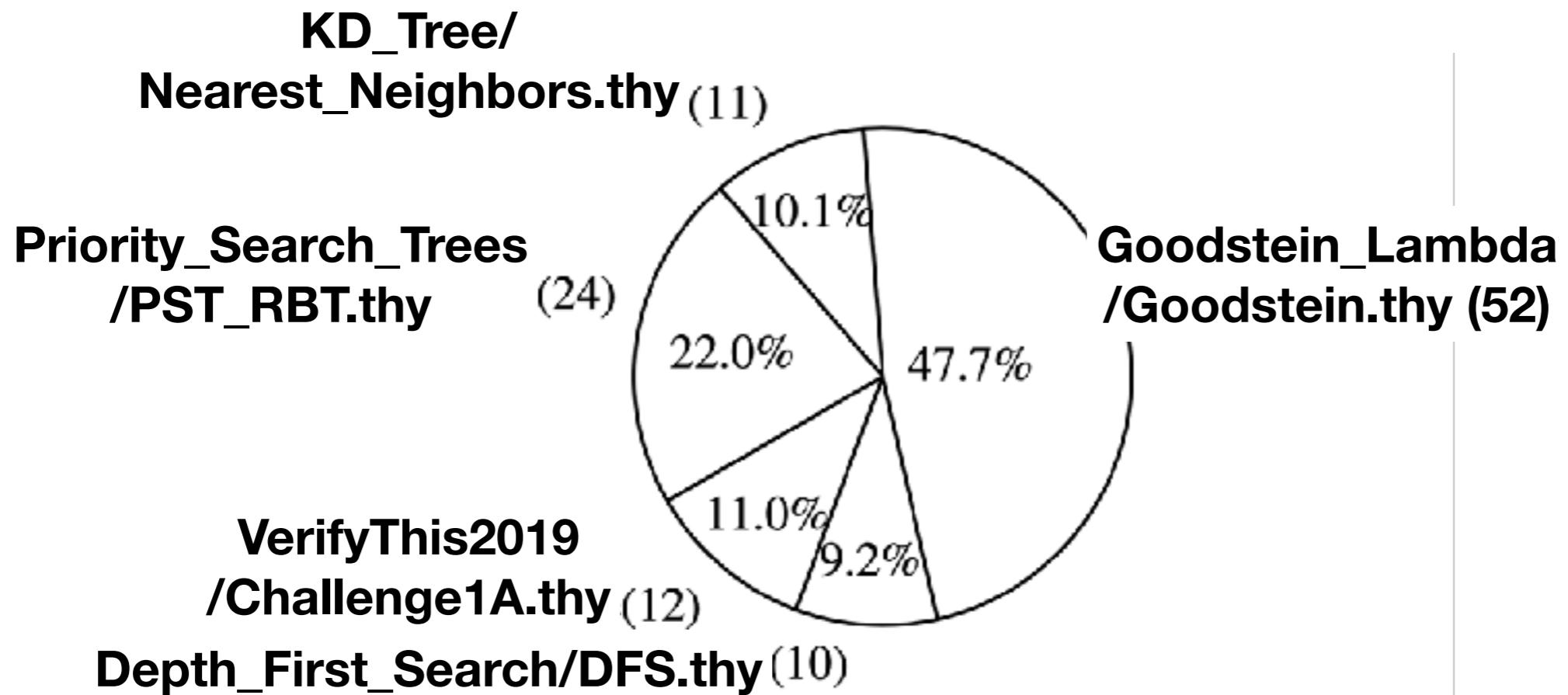
```
lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
```

Does it (smart_induct) really work?

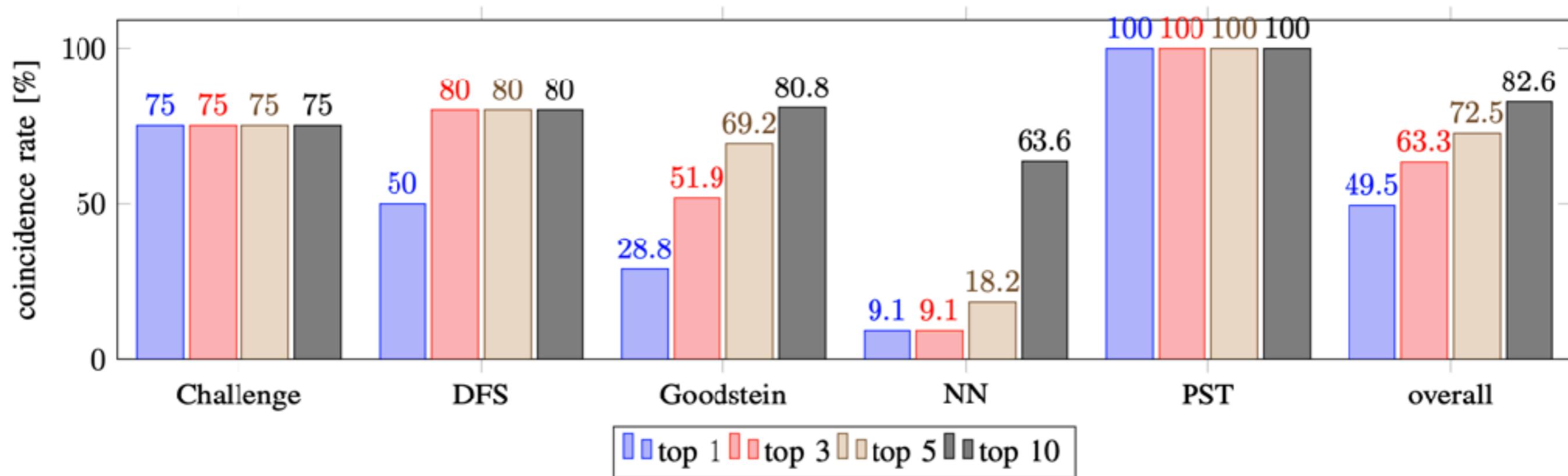
②



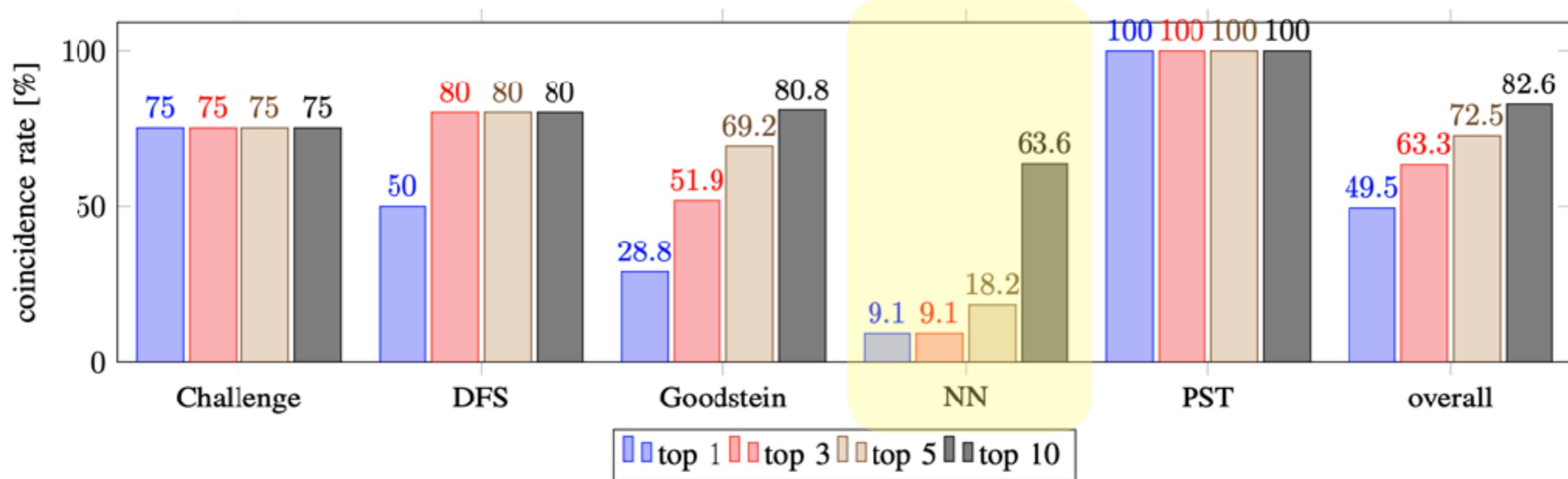
Dataset for evaluation (109 proofs by induction from the AFP entries)



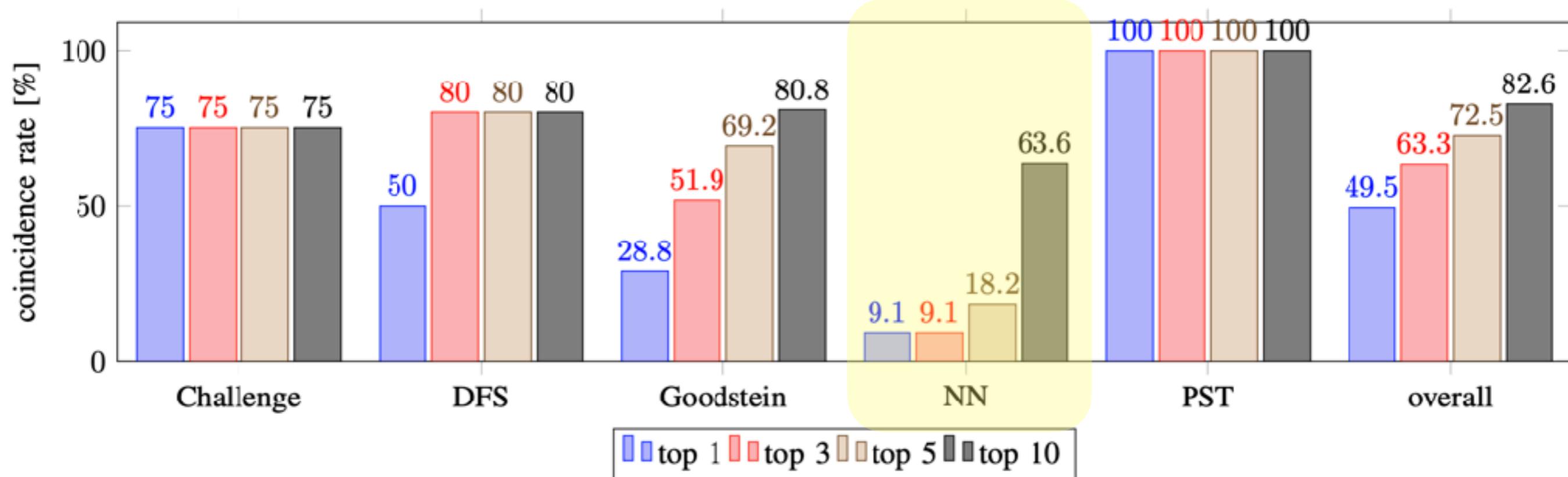
Evaluation results in terms of theory files.



Evaluation results in terms of theory files.



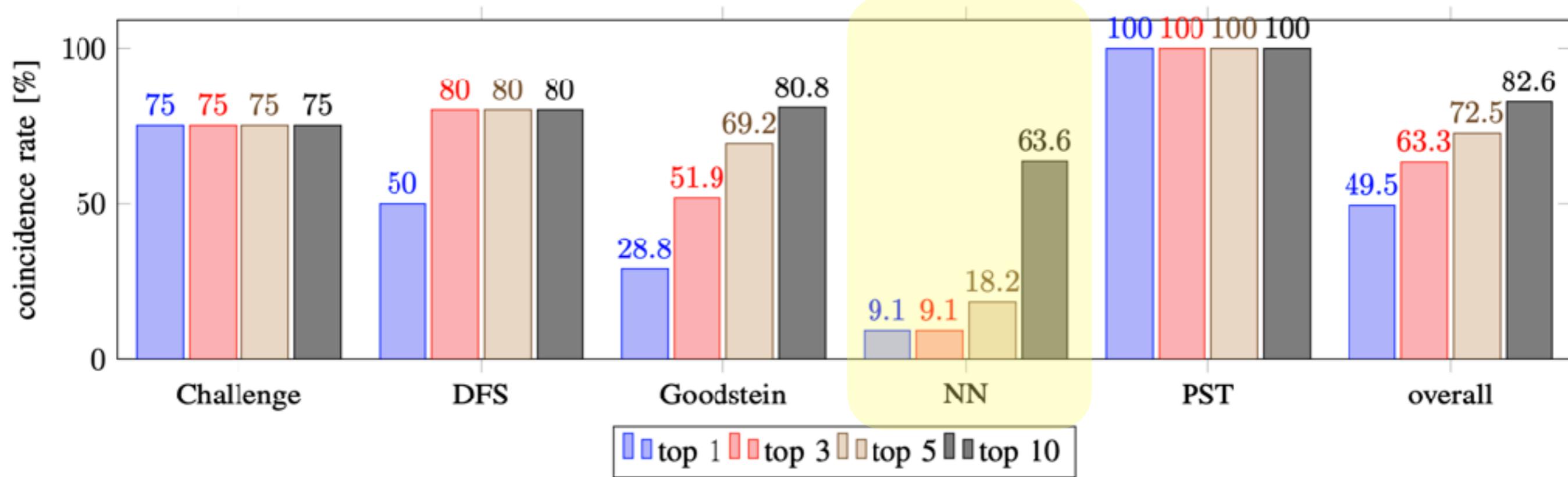
Evaluation results in terms of theory files.



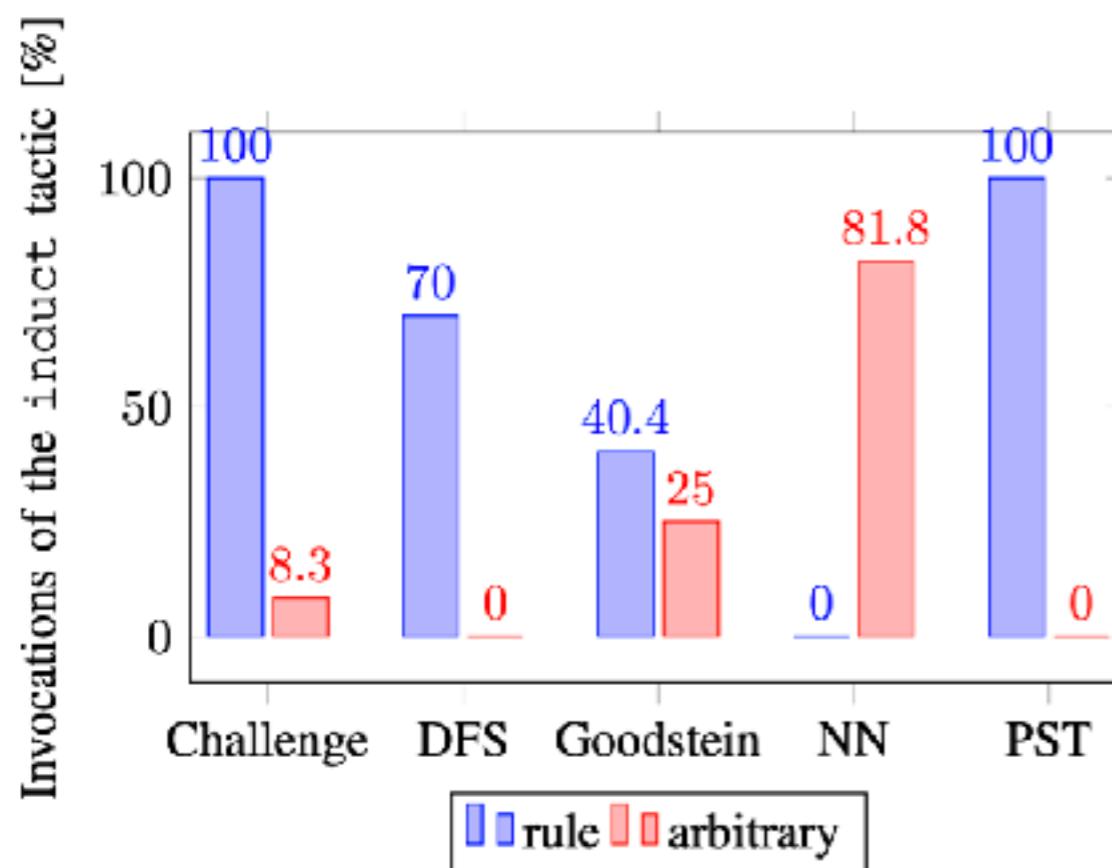
Doesn't the performance depend heavily
on problem domains?



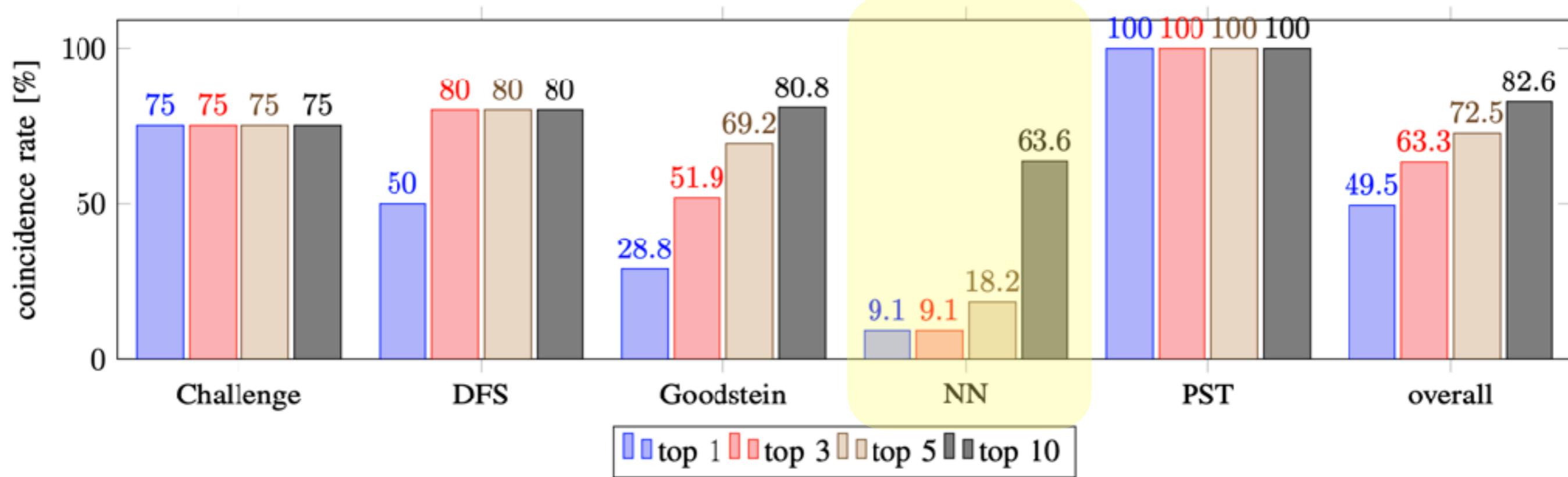
Evaluation results in terms of theory files.



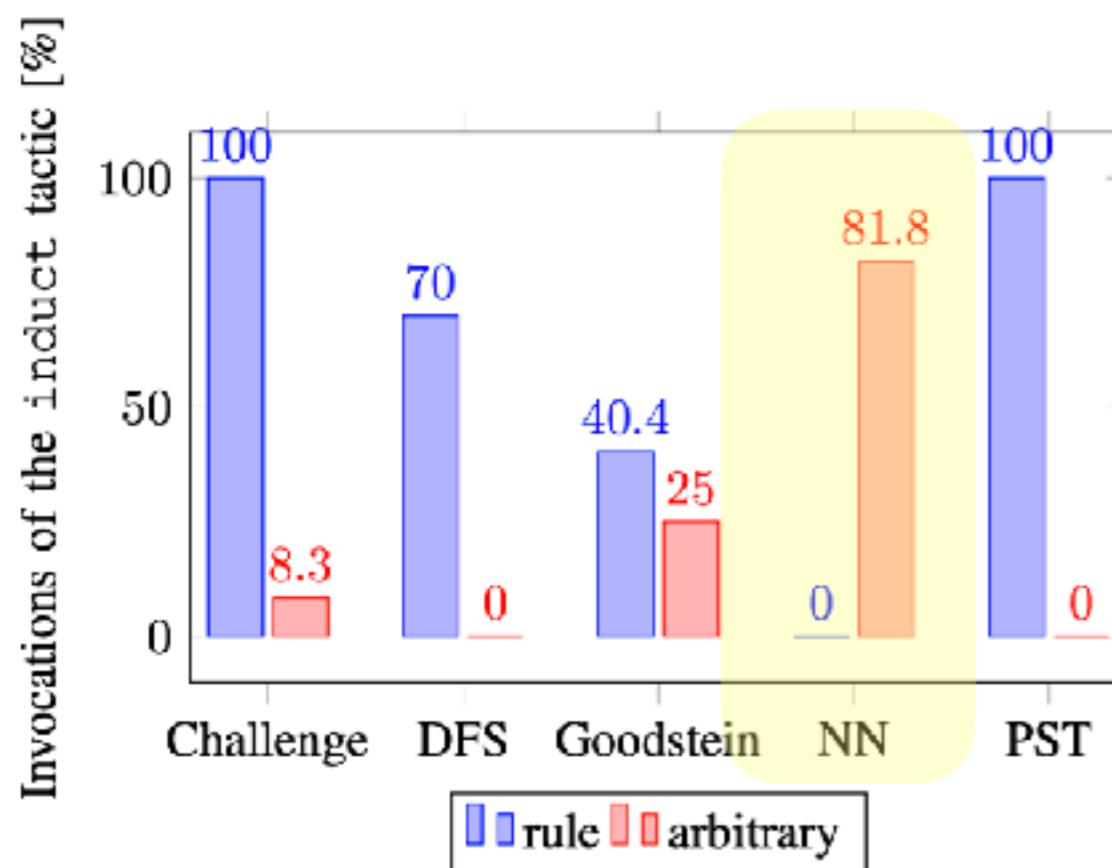
Doesn't the performance depend heavily on problem domains?



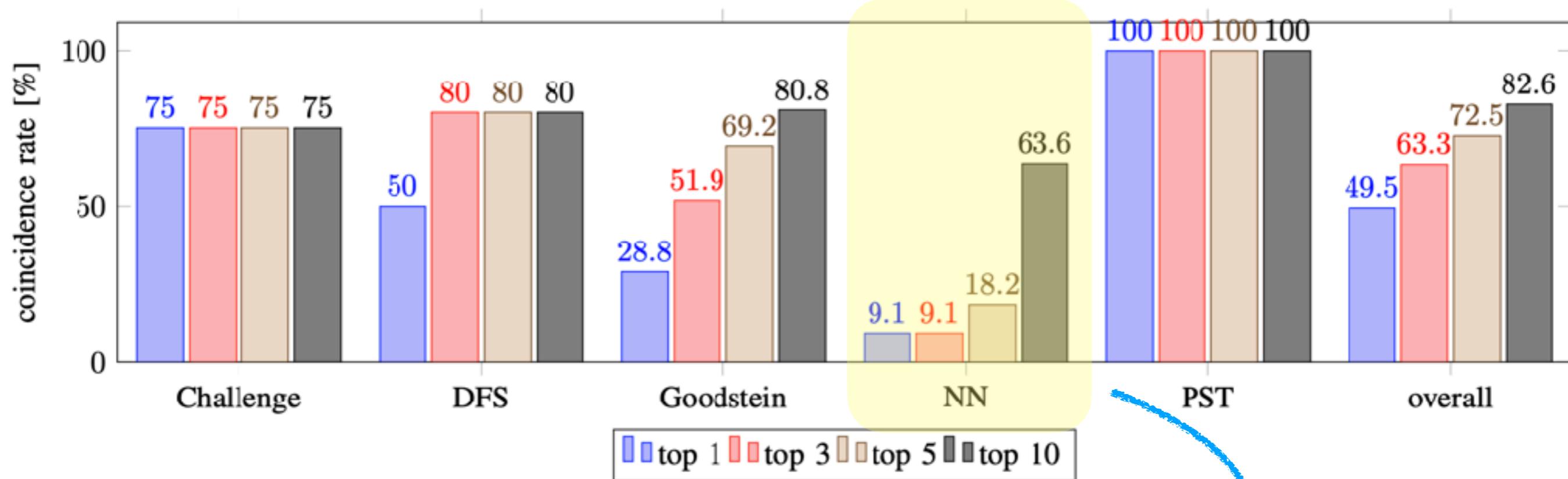
Evaluation results in terms of theory files.



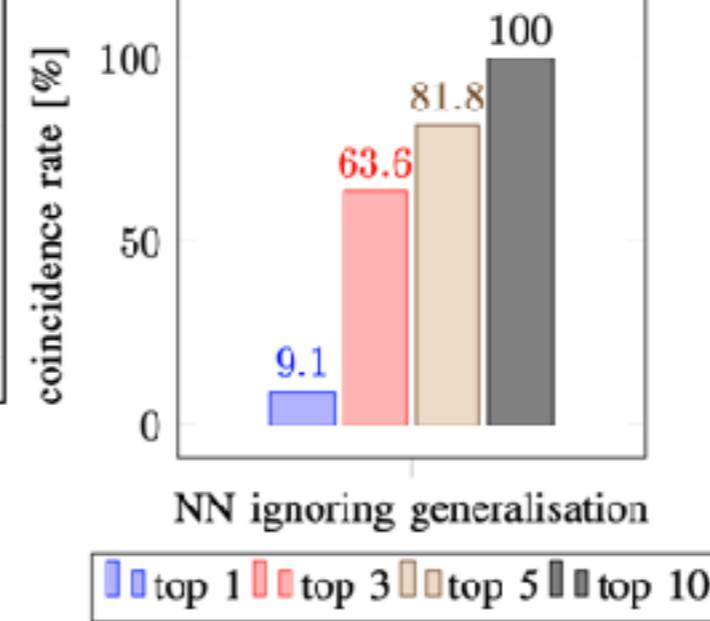
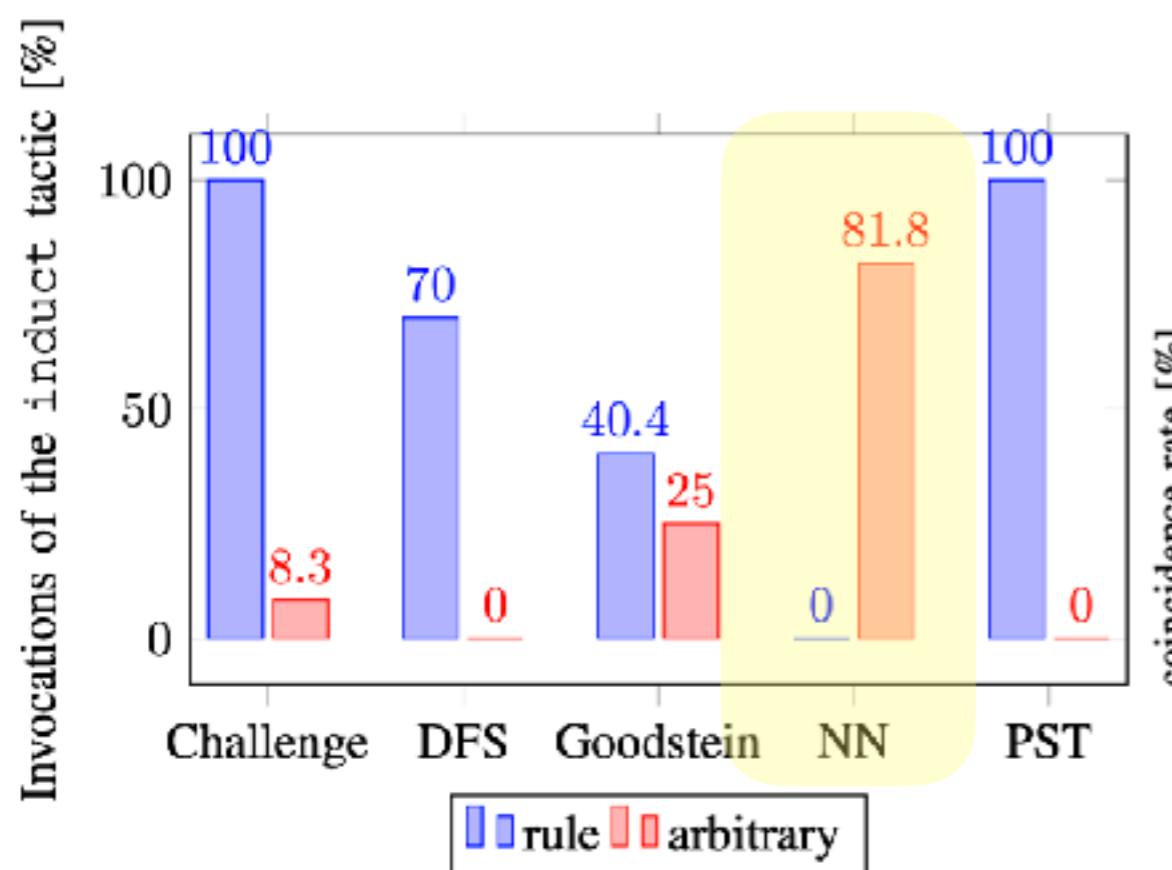
Doesn't the performance depend heavily on problem domains?



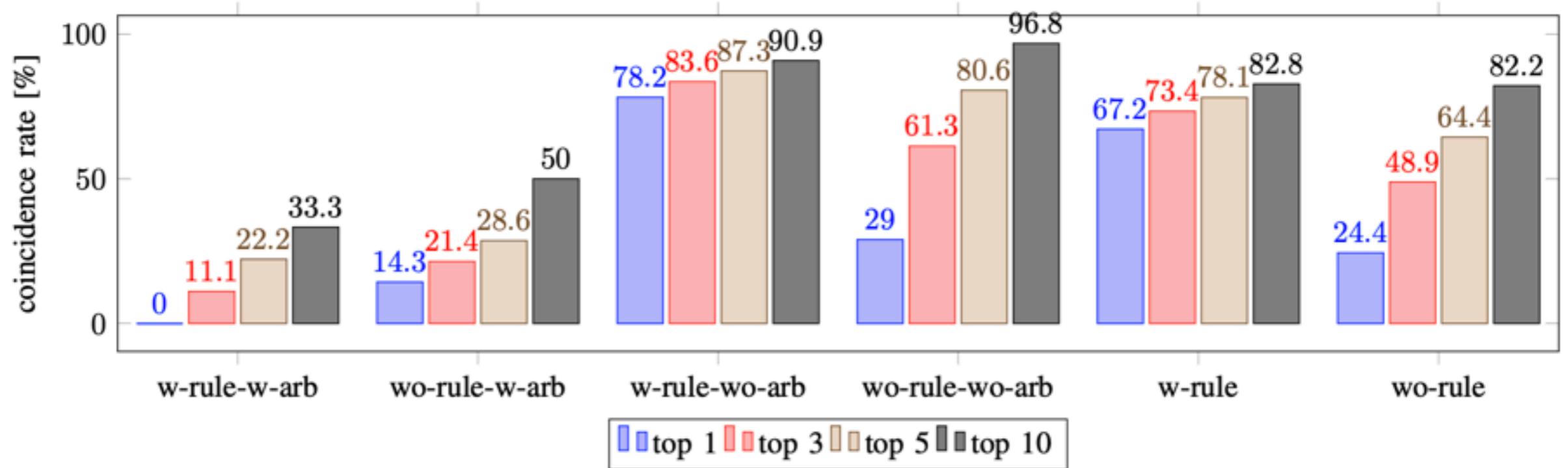
Evaluation results in terms of theory files.



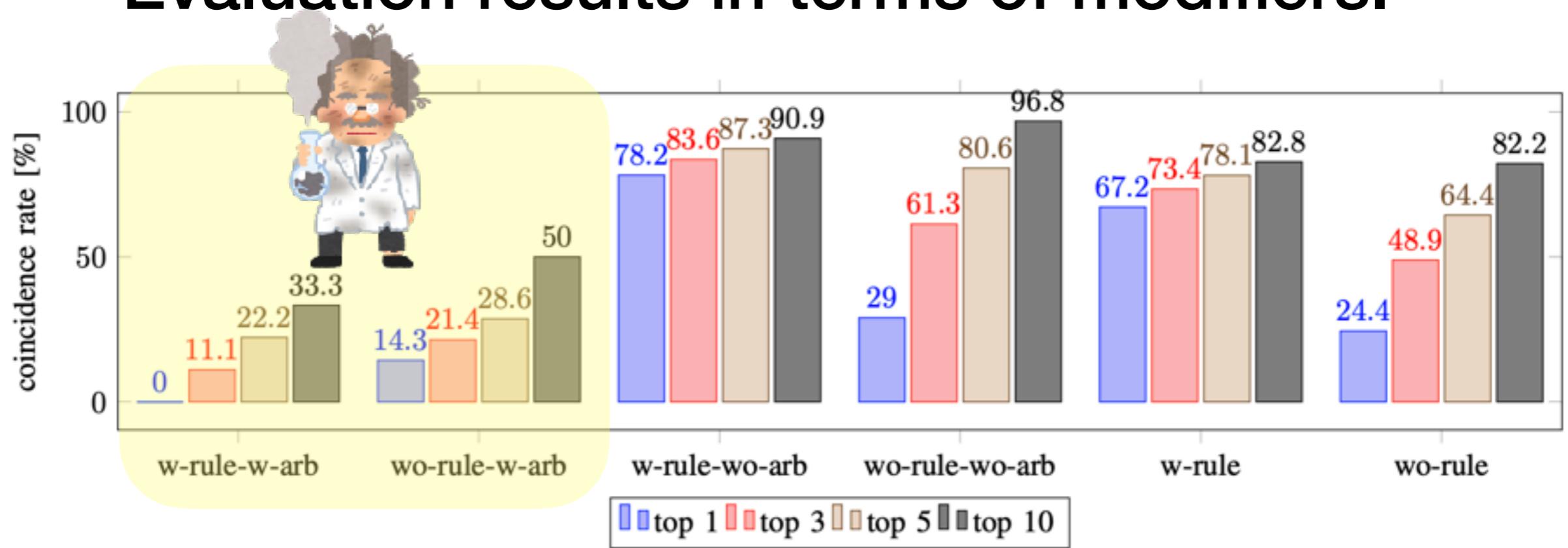
Doesn't the performance depend heavily on problem domains?



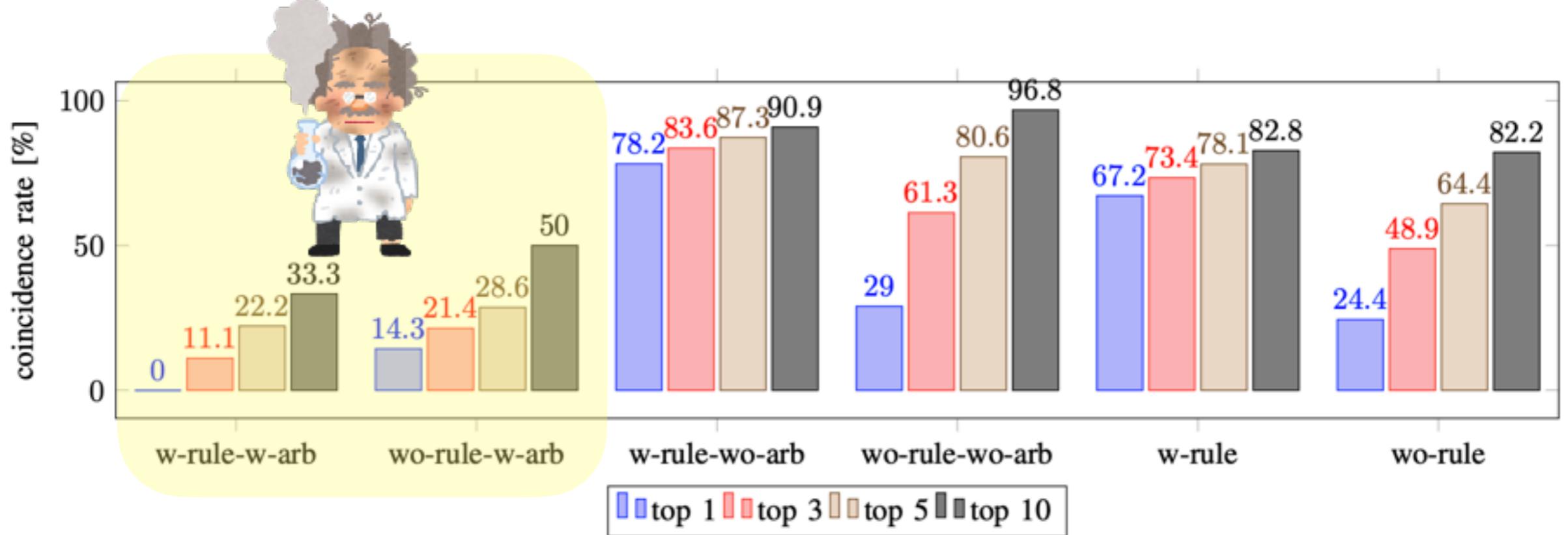
Evaluation results in terms of modifiers.



Evaluation results in terms of modifiers.



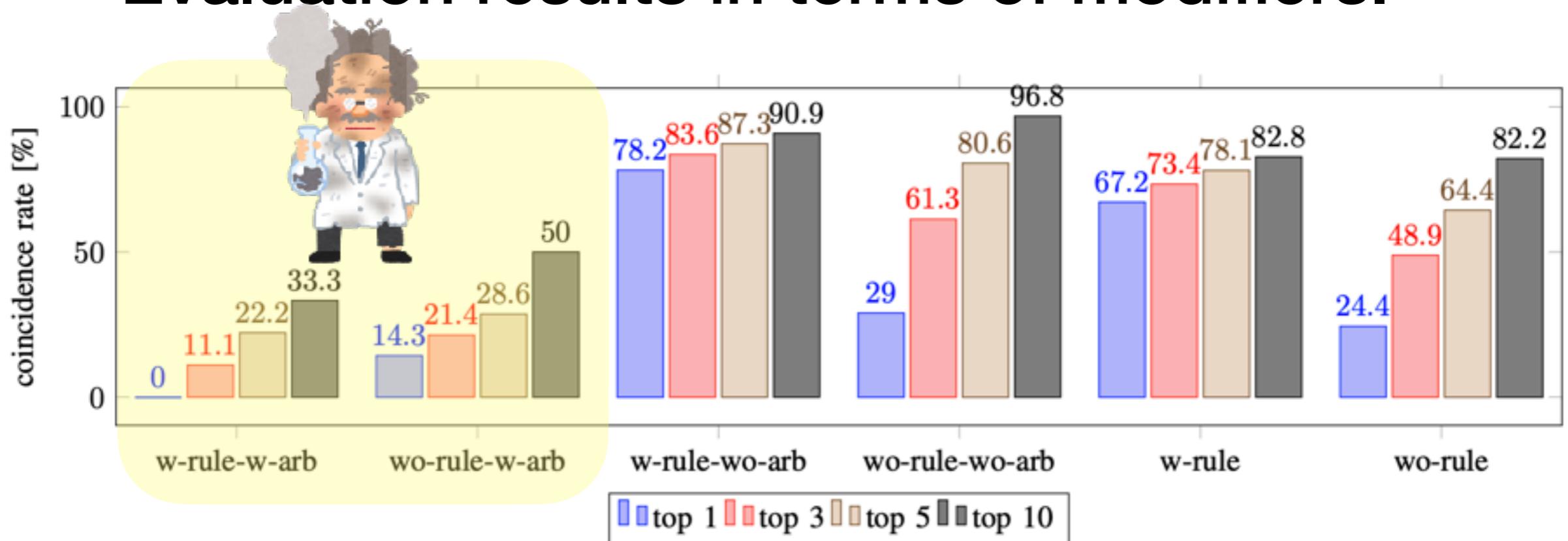
Evaluation results in terms of modifiers.



Are you going to address the problem of generalization?



Evaluation results in terms of modifiers.



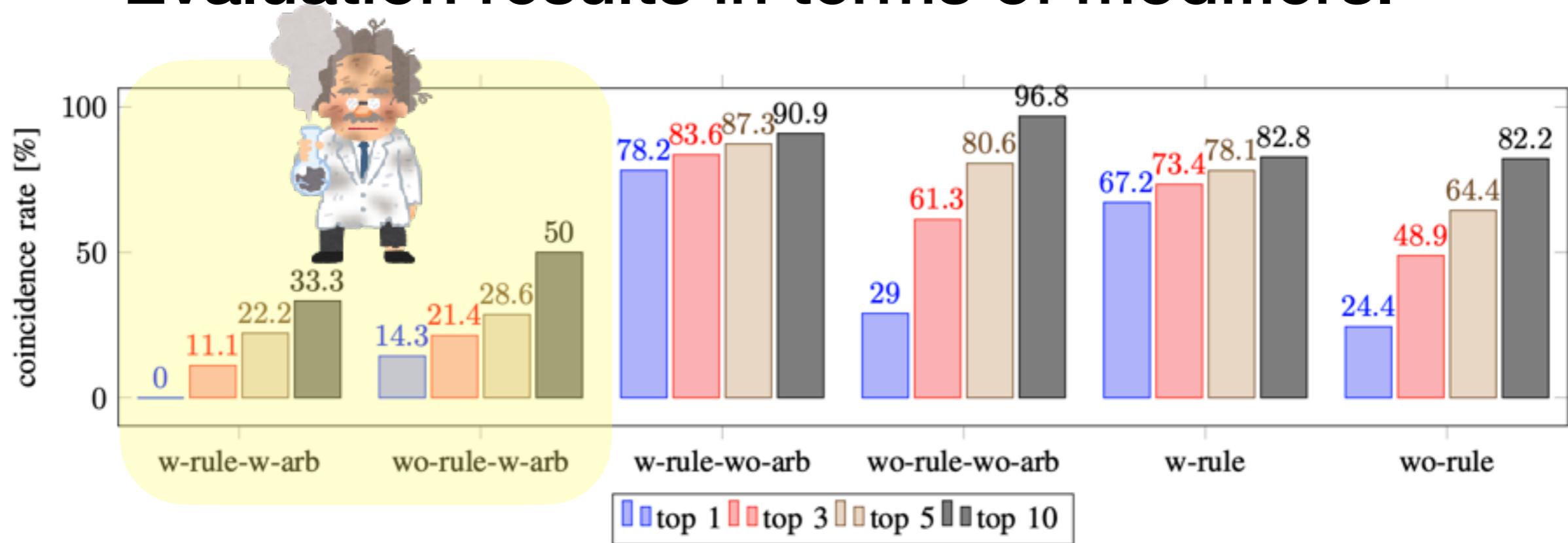
Are you going to address the problem of generalization?



Yes. smart_induct2 is coming soon with LiFtEr2.



Evaluation results in terms of modifiers.



Are you going to address the problem of generalization?



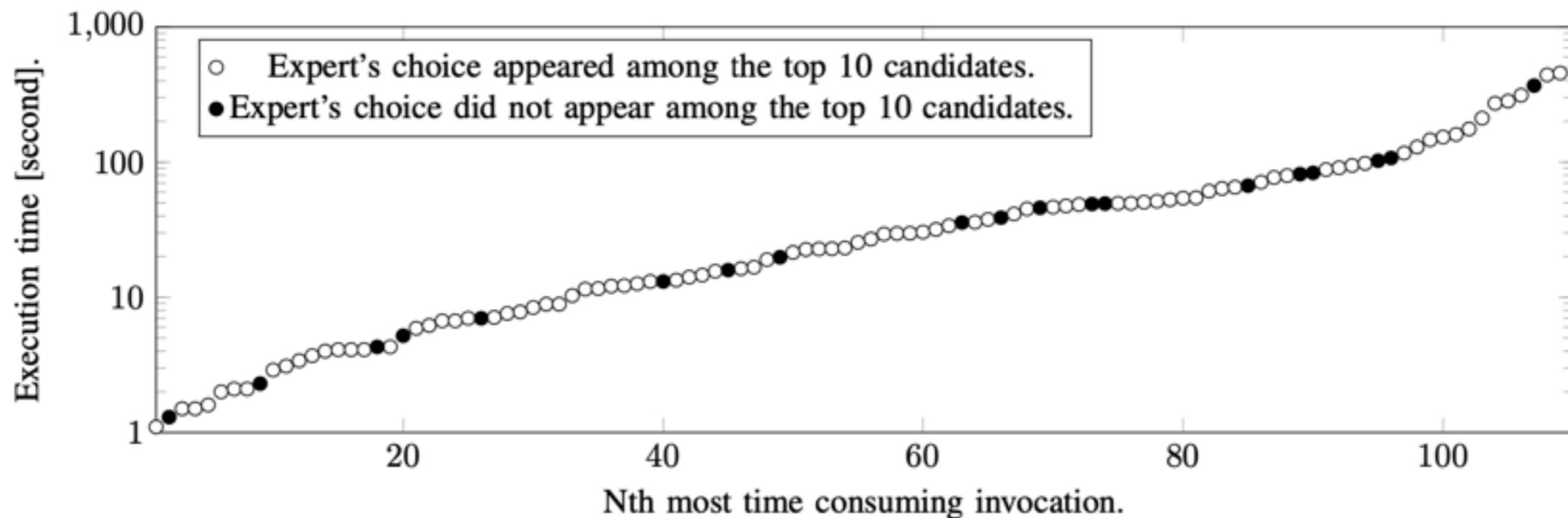
Yes. smart_induct2 is coming soon with LiFtEr2.



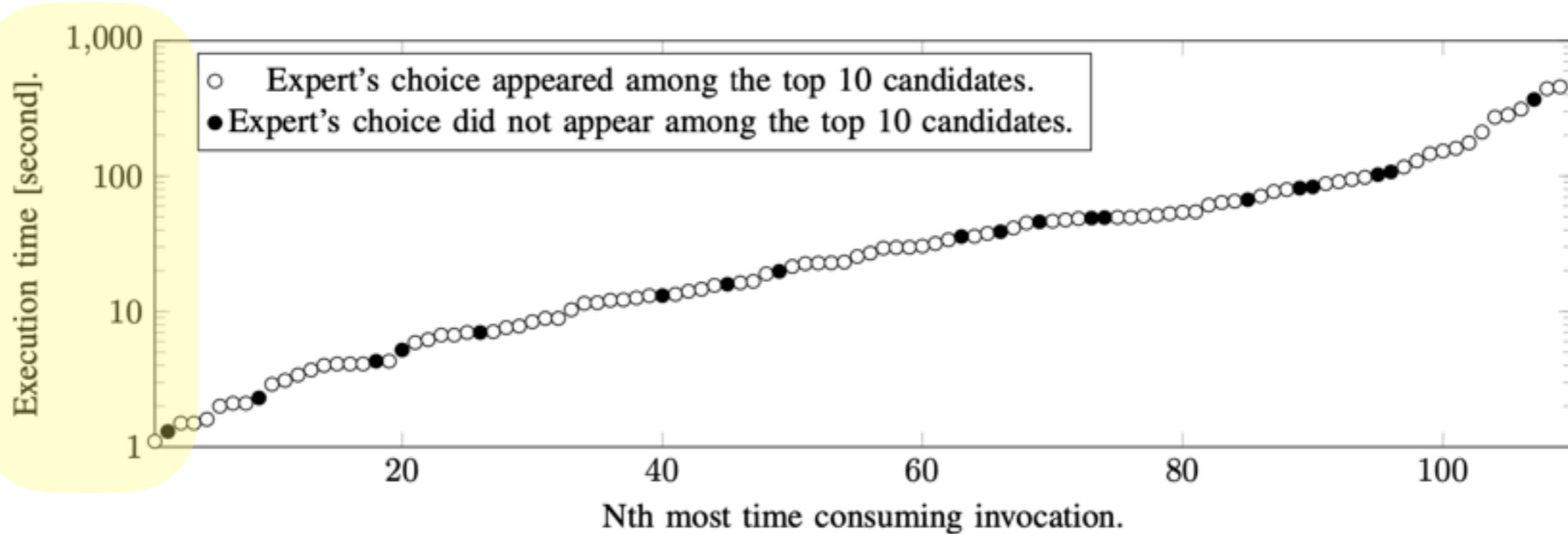
How fast does smart_induct work?

Execution time of smart_induct

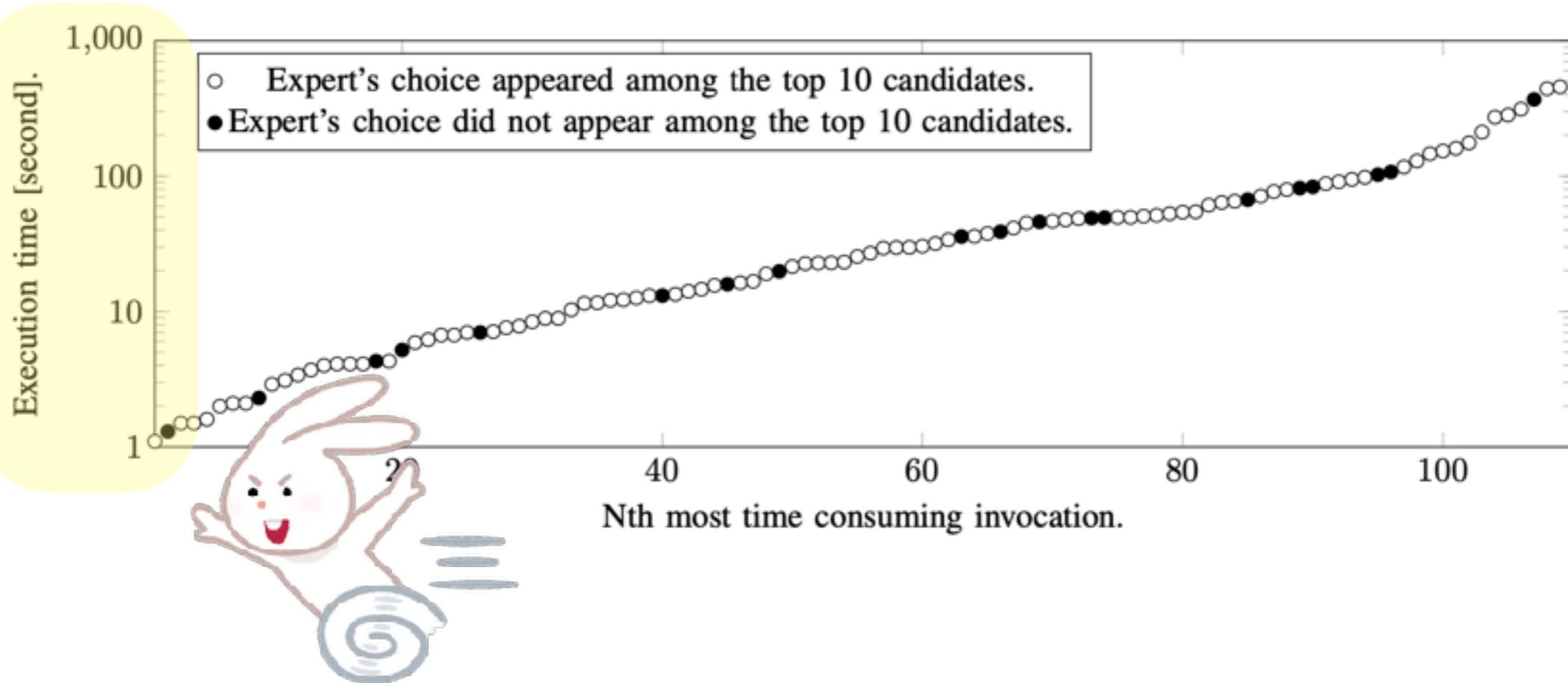
Execution time of smart_induct



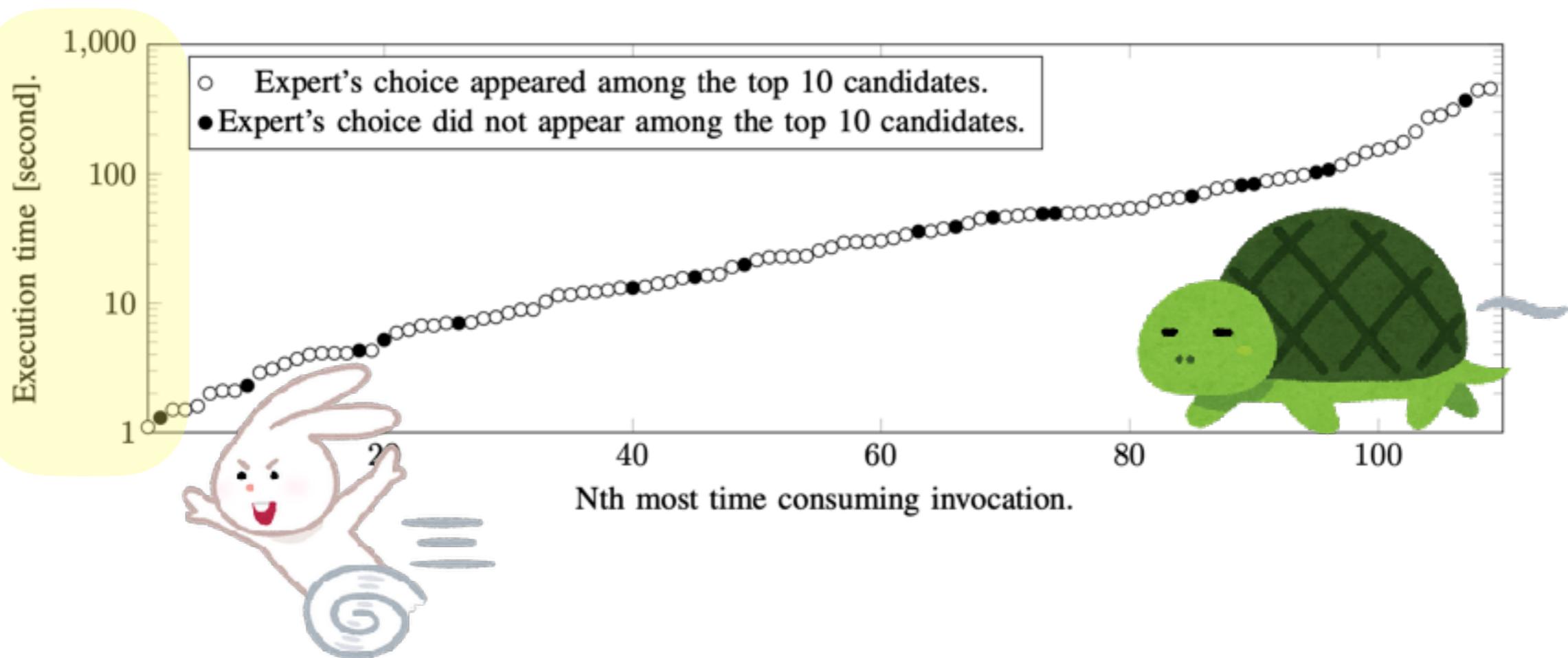
Execution time of smart_induct



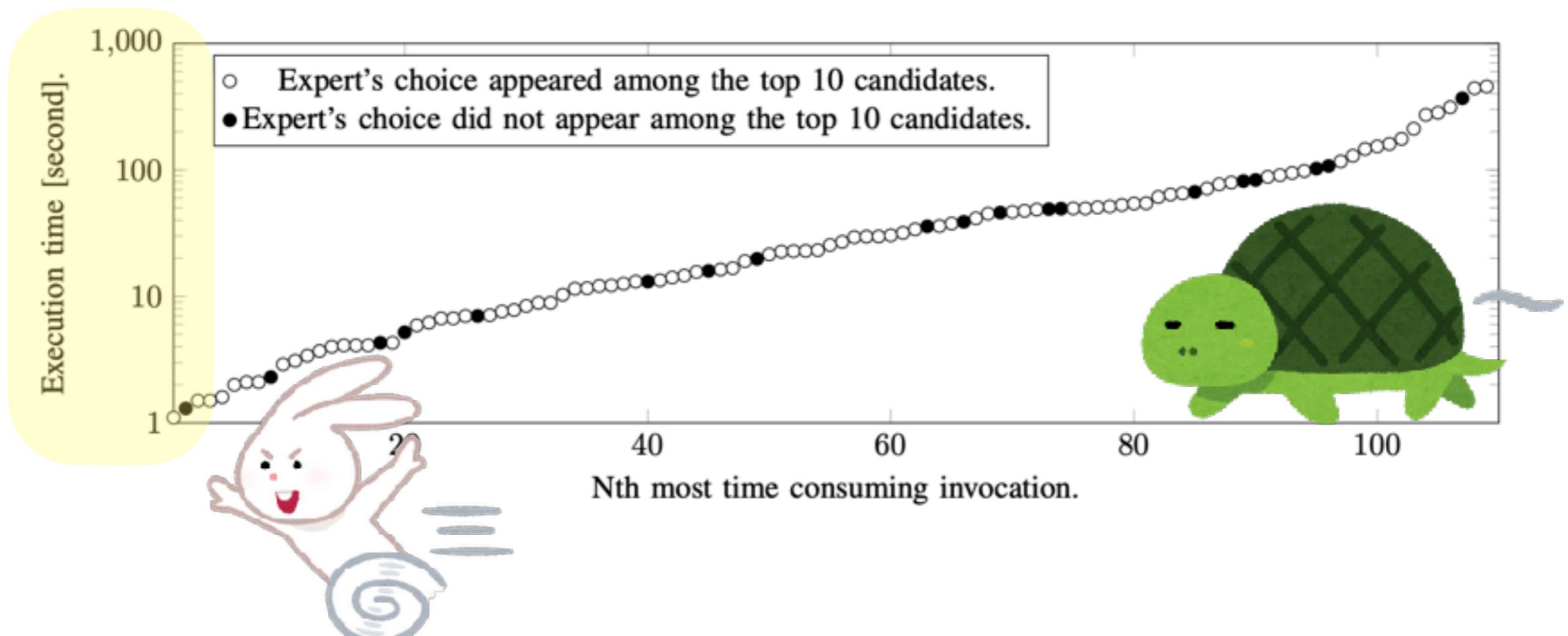
Execution time of smart_induct



Execution time of smart_induct



Execution time of smart_induct

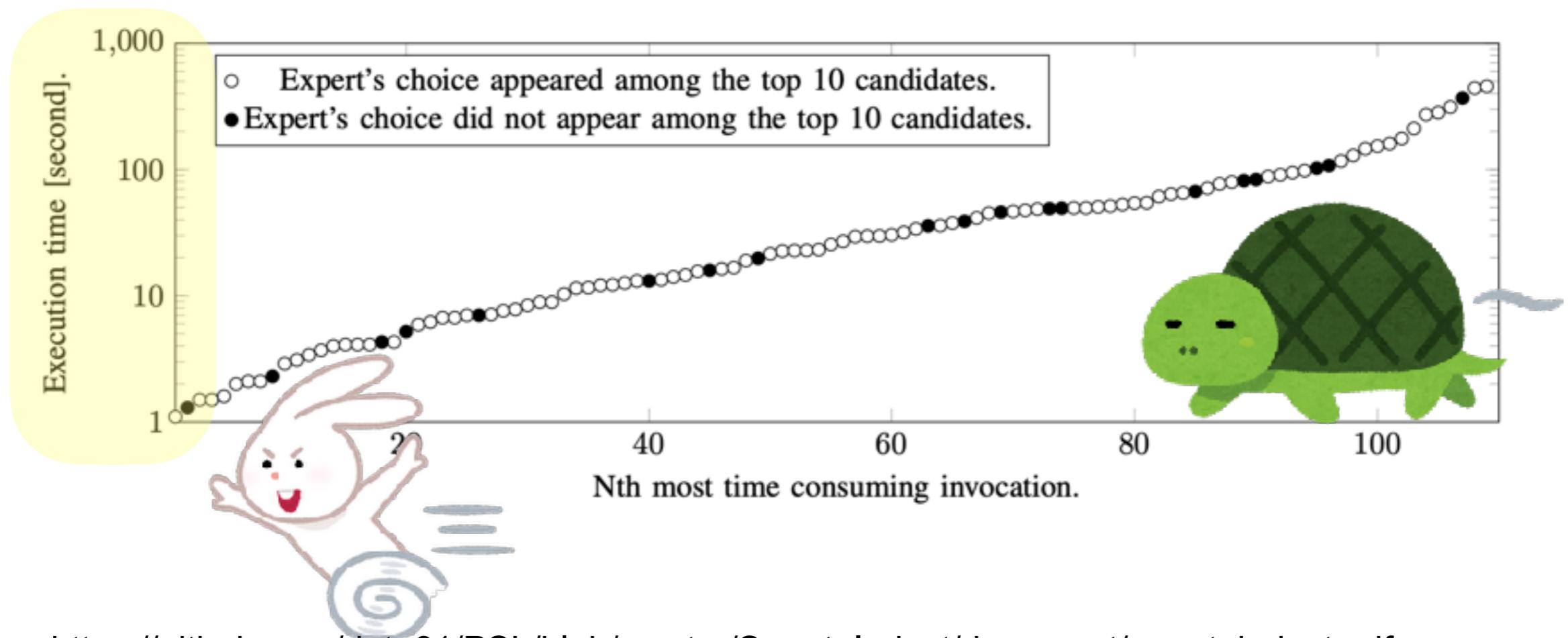


paper: https://github.com/data61/PSL/blob/master/Smart_Induct/document/smart_induct.pdf

smart_induct: <https://github.com/data61/PSL/releases/tag/0.1.7-alpha>

Q&A!

Execution time of smart_induct



paper: https://github.com/data61/PSL/blob/master/Smart_Induct/document/smart_induct.pdf

smart_induct: <https://github.com/data61/PSL/releases/tag/0.1.7-alpha>

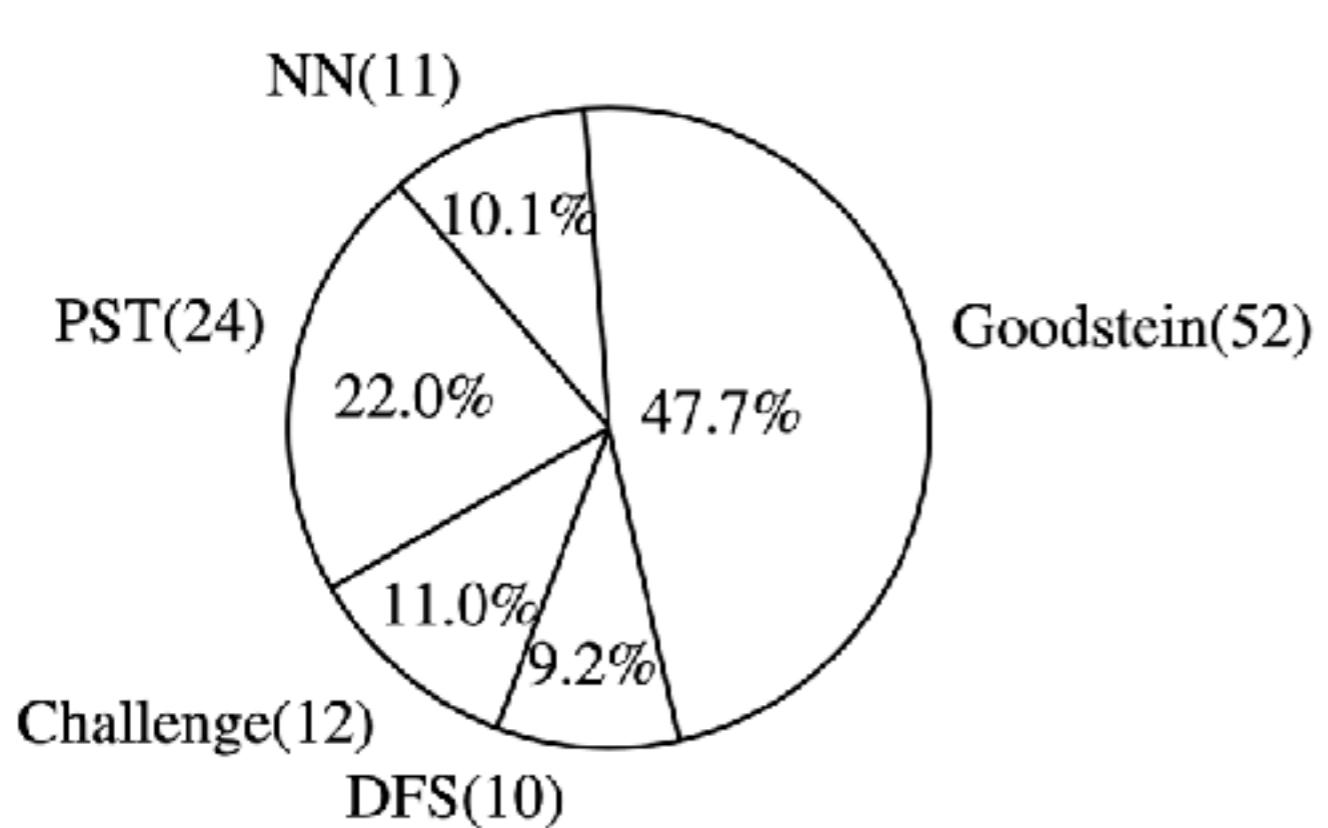


Do you want to have PSL and smart_induct in
the official release of Isabelle?

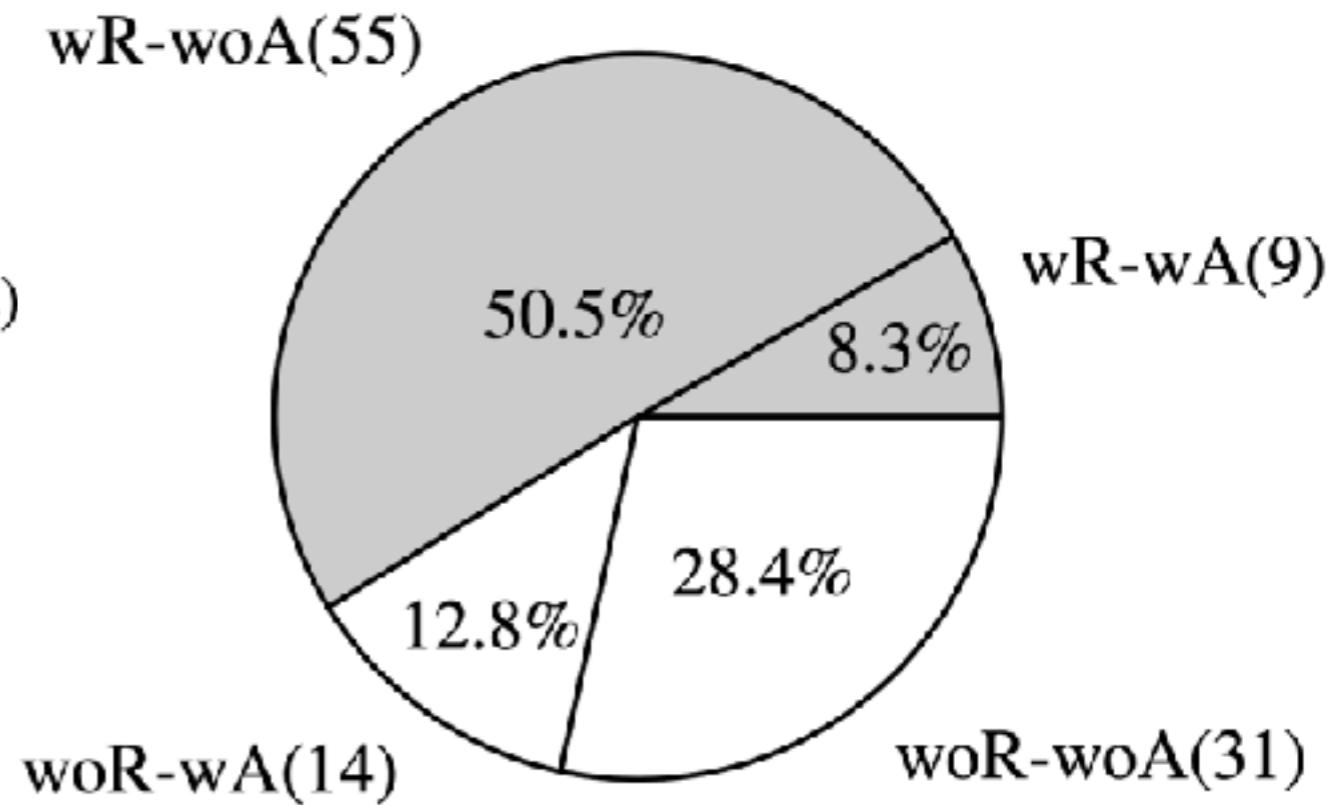
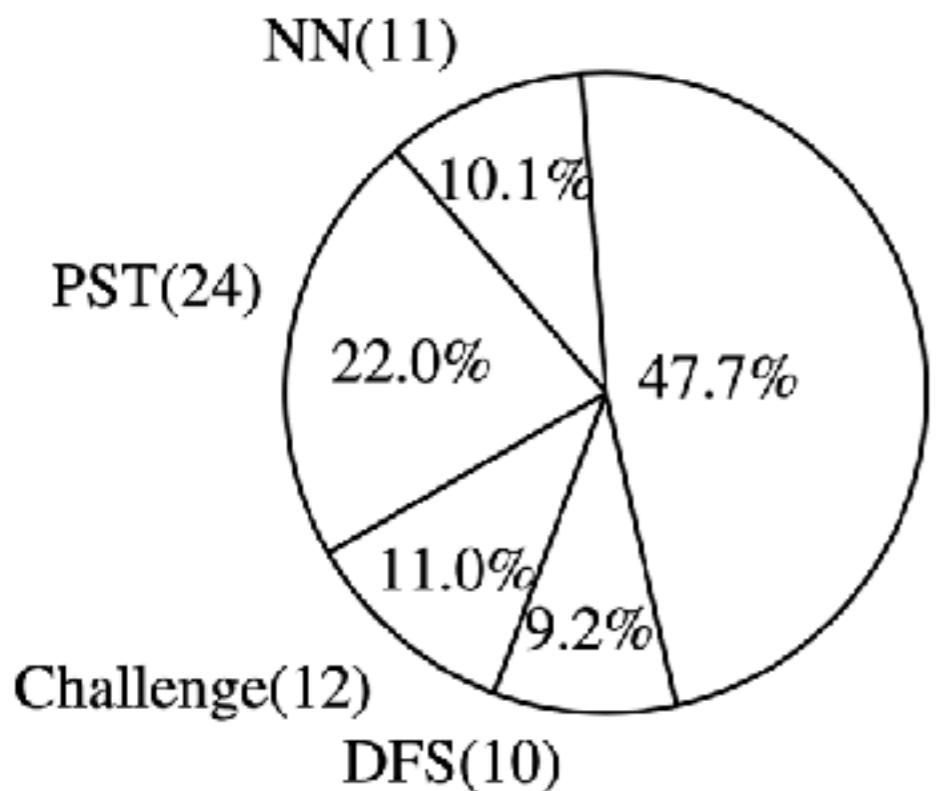
Q&A!

backup slides for Q&A

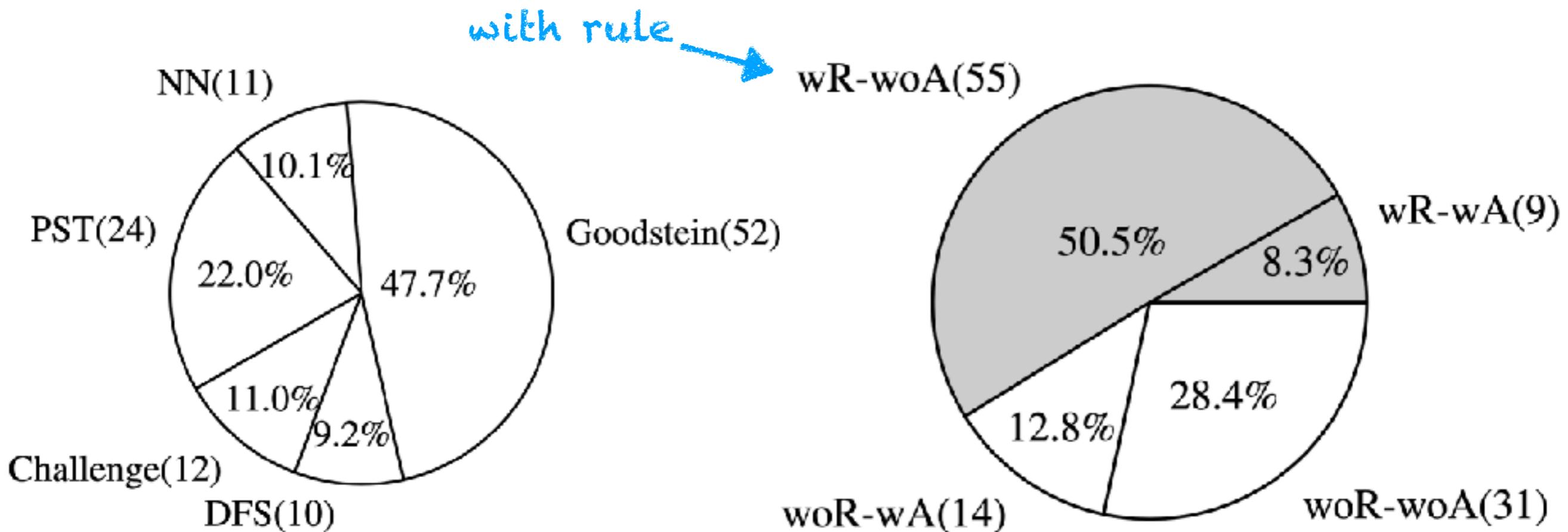
Dataset for evaluation (109 proofs by induction from the AFP entries)



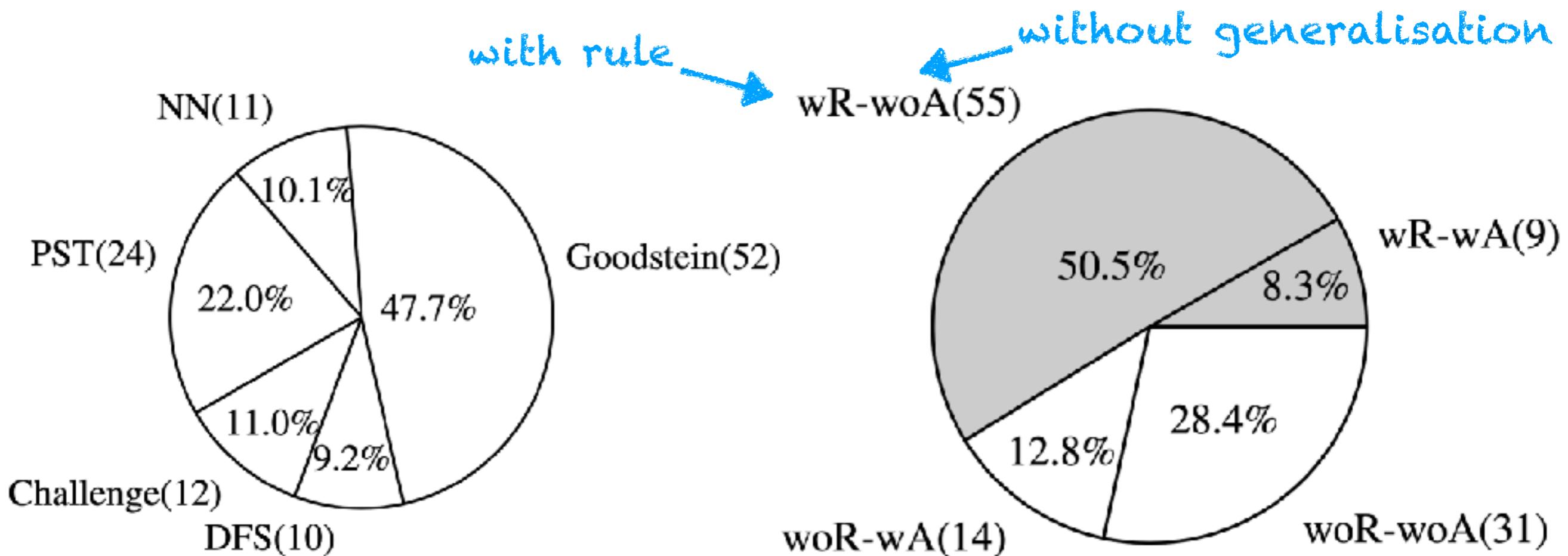
Dataset for evaluation (109 proofs by induction from the AFP entries)



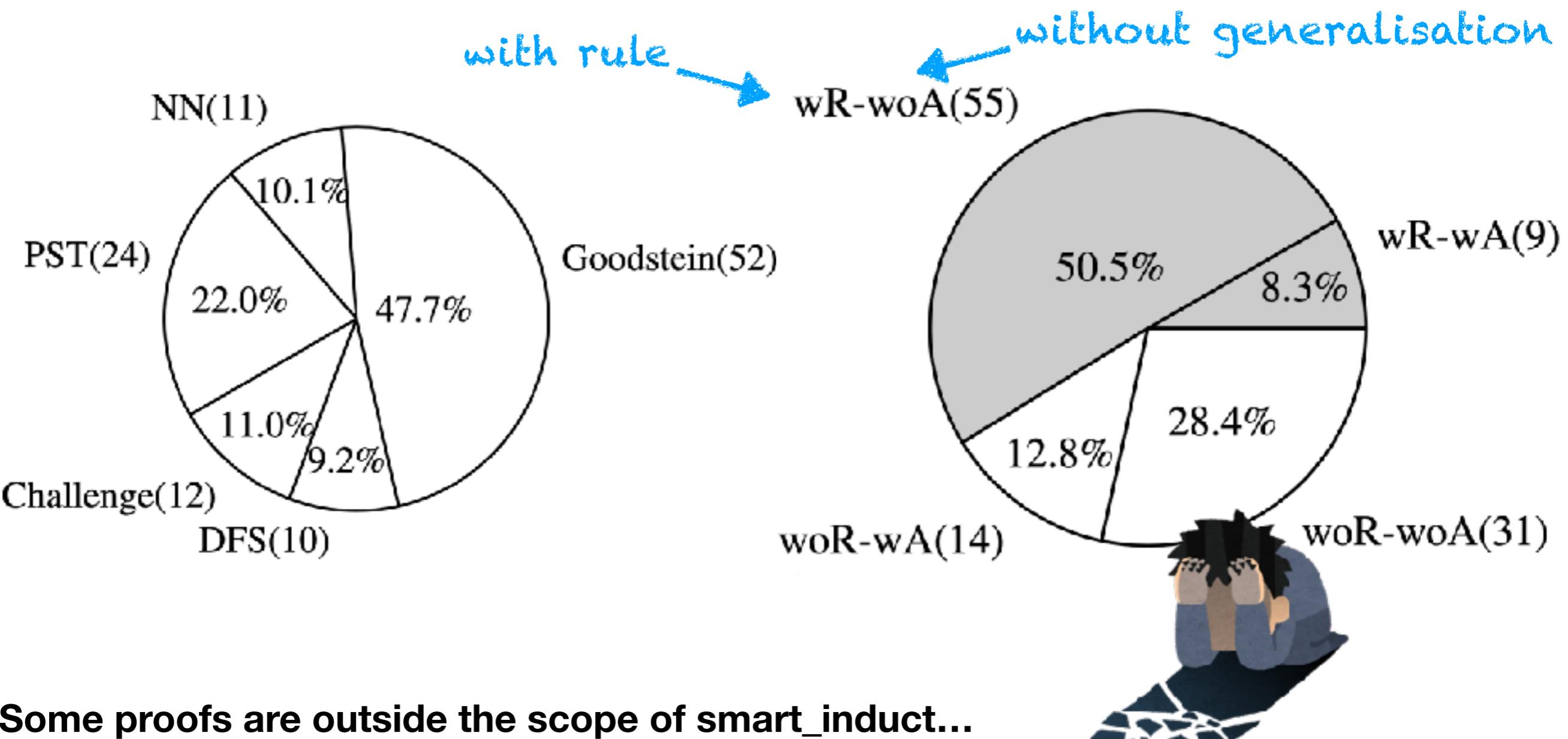
Dataset for evaluation (109 proofs by induction from the AFP entries)



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



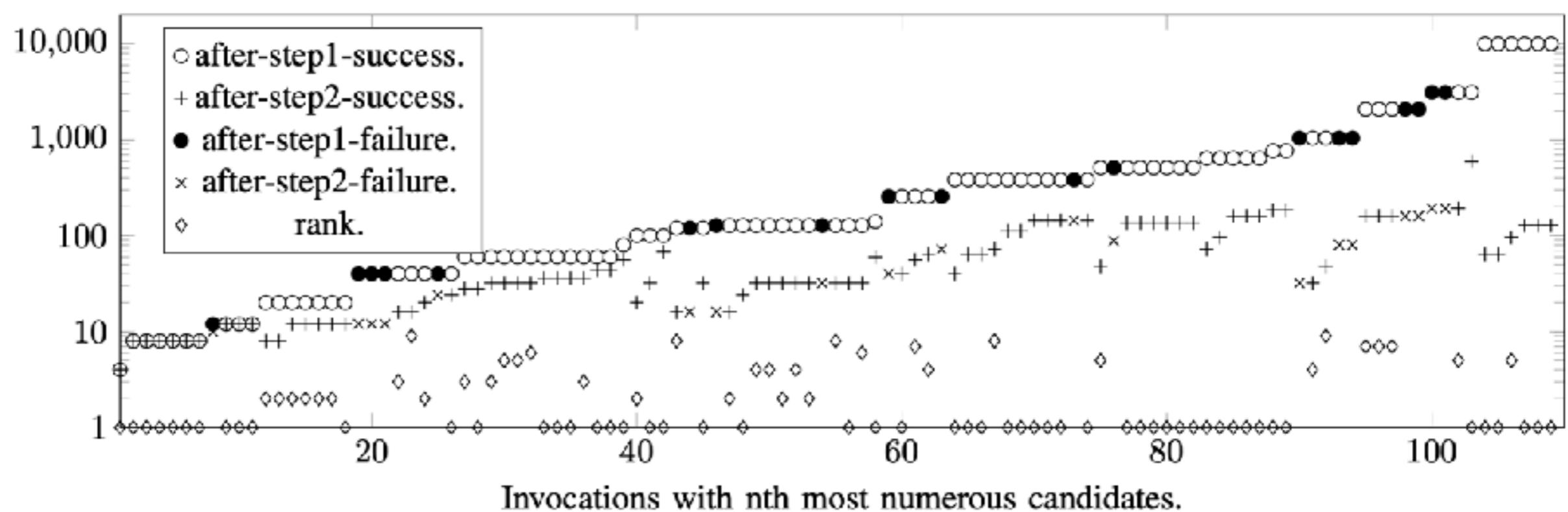
Dataset for evaluation (109 proofs by induction from the AFP entries)



Some proofs are outside the scope of smart_induct...

-	w/ handwritten rule	w/o handwritten rule
w/ compound term	1 (0.9%)	1 (0.9%)
w/o compound term	5 (4.6%)	102 (93.6%)

Number of candidates after each stage.



$[[T,F,T], [T,T,T], [F,T,T]]$: bool list

 **lemma** "star r x y \Rightarrow star r x z"
by(induction rule: star.induct) auto simp: step)

 **lemma** "exec (is1 @ is2) s = exec is2 s (exec is1 s)"
by(induct is1 s stk rule:exec.induct) auto

 **lemma** "itrev xs ys = rev xs @ ys"
by(induct xs arbitrary: ys) auto

\leftarrow simple representation

\leftarrow small dataset about different domains

\leftarrow one abstract representation

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

$[[T,F,T], [T,T,T], [F,T,T]]$: bool list

\leftarrow simple representation

```
lemma "star r x y ==> star r x z"  
by(induction rule: star.induct) auto simp: step)
```

\leftarrow small dataset about different domains

```
lemma "exec (is1 @ is2) s ==>  
      exec is2 s (exec is1 s)"  
by(induct is1 s stk rule: exec.induct) auto
```

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

\leftarrow one abstract representation

```
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)"
```

$[[T,F,T], [T,T,T], [F,T,T]]$: bool list

← simple representation

```
lemma "star r x y ==> star r x z"  
by(induction rule: star.induct) auto simp: step)
```

← small dataset about different domains

```
lemma "exec (is1 @ is2) s ==>  
      exec is2 s (exec is1 s)"  
by(induct is1 s stk rule:exec.induct) auto
```

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

← one abstract representation

```
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)"
```

$[[T,F,T], [T,T,T], [F,T,T]]$: bool list

lemma "star r x y \Rightarrow star r x z"
by(induction rule: star.induct) auto simp: step)

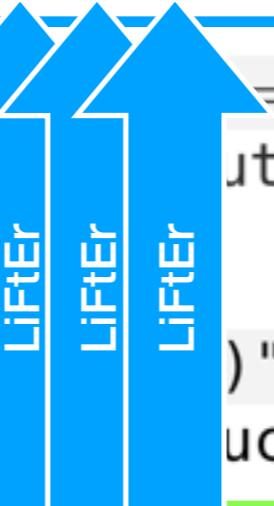
lemma "exec (is1 @ is2) s = exec is2 s (exec is1 s)"
by(induct is1 s stk rule: exec.induct) auto

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 "semantics" of constants (rev and itrev)

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

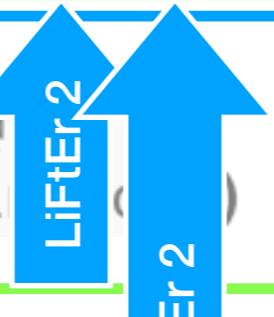
fun itrev :: "'a list \Rightarrow 'a list \Rightarrow 'a list" where
"itrev [] ys = ys" |
"itrev (x#xs) ys = itrev xs (x#ys)"



\leftarrow simple representation

\leftarrow small dataset about different domains

\leftarrow one abstract representation



\leftarrow nested assertions to examine the "semantics" of constants (rev and itrev)

\leftarrow relevant definitions

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

`[[T,F,T], [T,T,T], [F,T,T]]`: bool list

← simple representation

```
lemma "star r x y ==> star r x z"  
by(induction rule: star.in  
    auto simp: step)
```

← small dataset about different domains

```
lemma "exec (is1 @ is2) s ==>  
      exec is2 s (exec is1 s)"  
by(induct is1 s stk rule:e  
    auto)
```

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

← one abstract representation

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

← 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]"
```

← relevant definitions

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

LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

[[T,F,T], [T,T,T], [F,T,T]]: bool list ← simple representation

lemma "star r x y ==> star r x z"
by(induction rule: star.induct) auto simp: step)

lemma "exec (is1 @ is2) s ==>
exec is2 s (exec is1 s)"
by(induct is1 s stk rule: exec.induct) auto

lemma "itrev xs ys = rev xs @ ys"
by(induct xs arbitrary: ys) auto

← small dataset about different domains

← one abstract representation

lemma "itrev xs ys = rev xs @ ys"
by(induct xs ys rule:"itrev.induct") auto "semantics" of constants (rev and itrev)

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

← relevant definitions

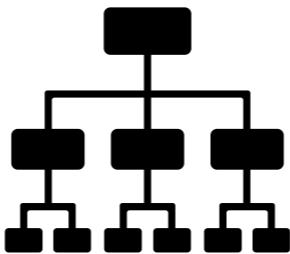
fun itrev :: "'a list => 'a list => 'a list" where
"itrev [] ys = ys" |
"itrev (x#xs) ys = itrev xs (x#ys)"

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

LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

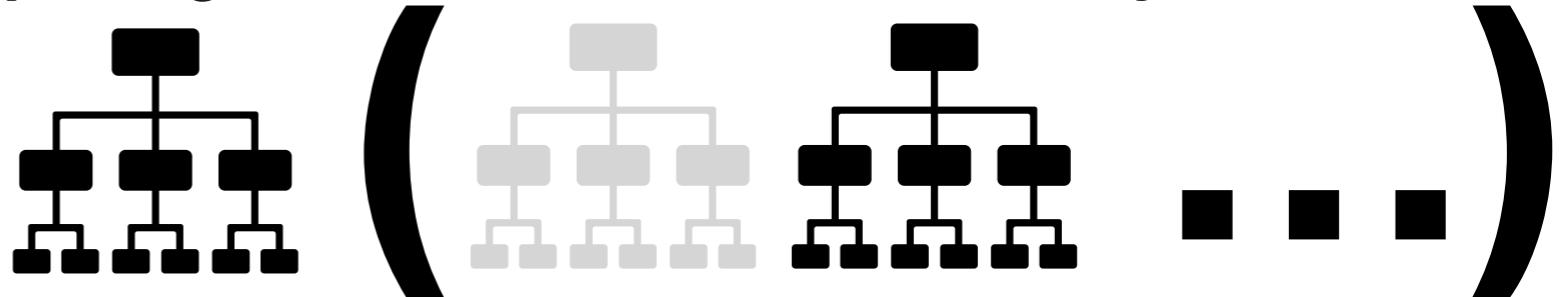
LiFtEr₂(proof goal * induction arguments)-> bool
* relevant definitions

proof goal

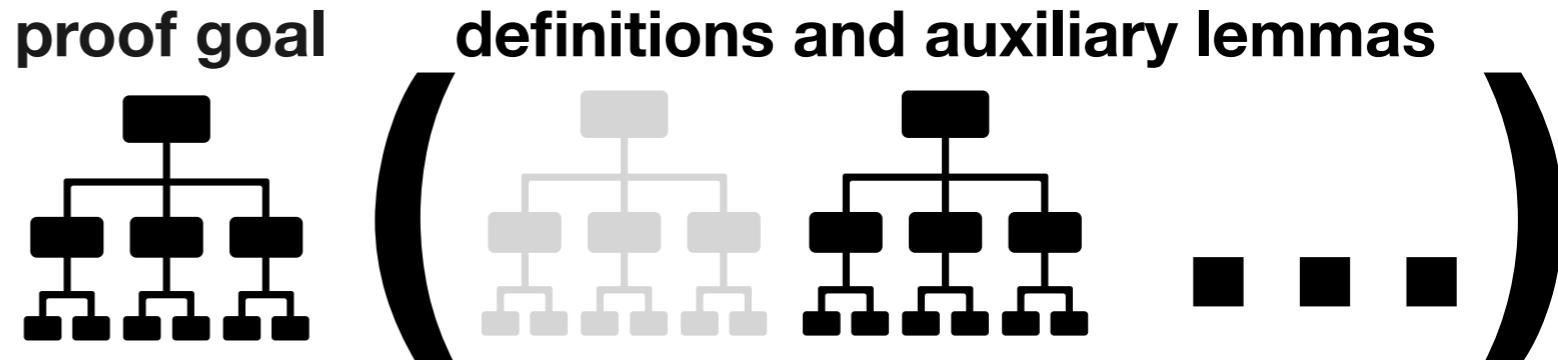
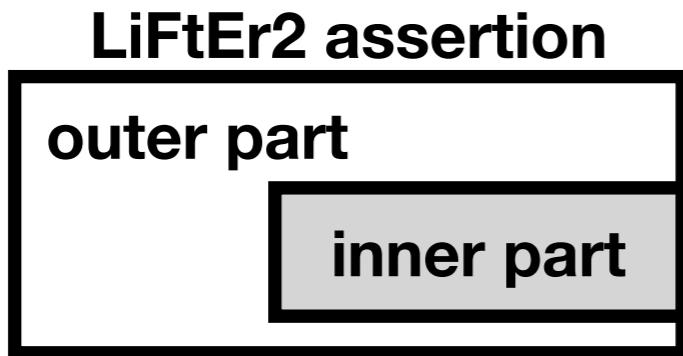


LiFtEr₂ (proof goal * induction arguments)-> bool
* relevant definitions

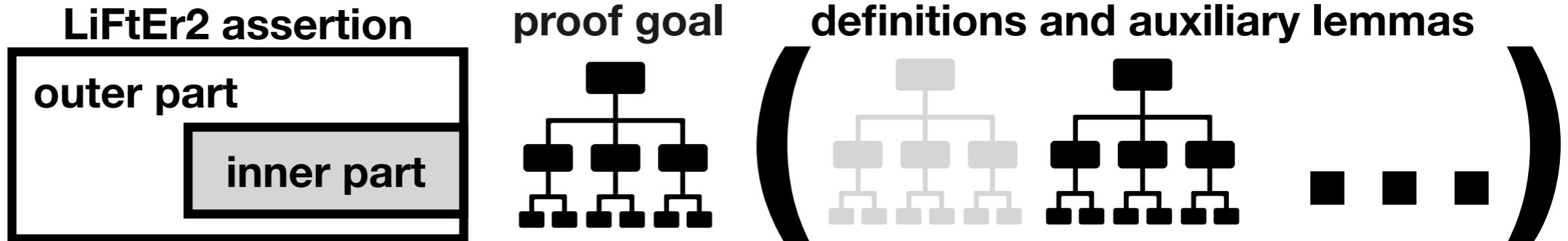
proof goal definitions and auxiliary lemmas



LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

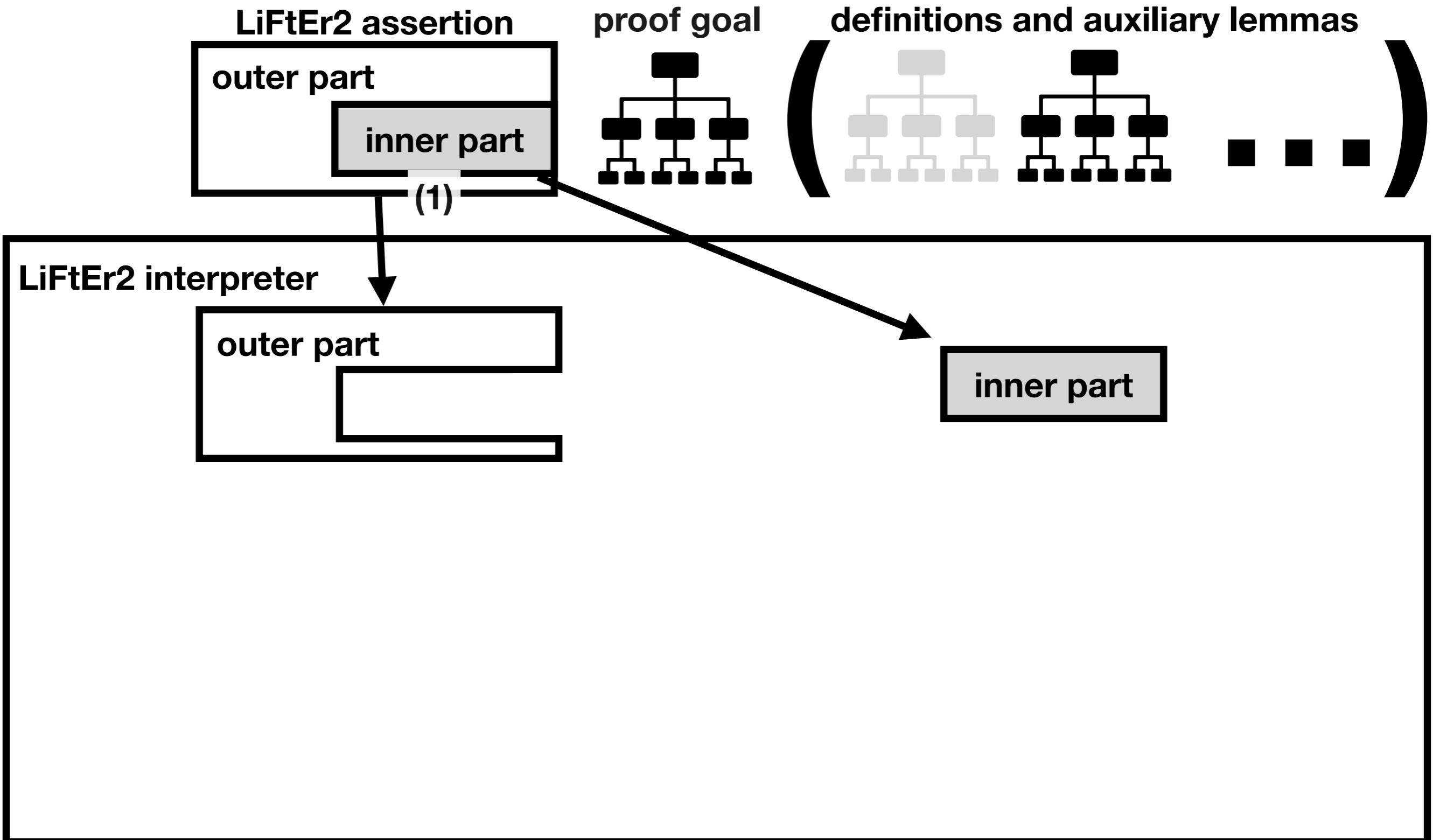


LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

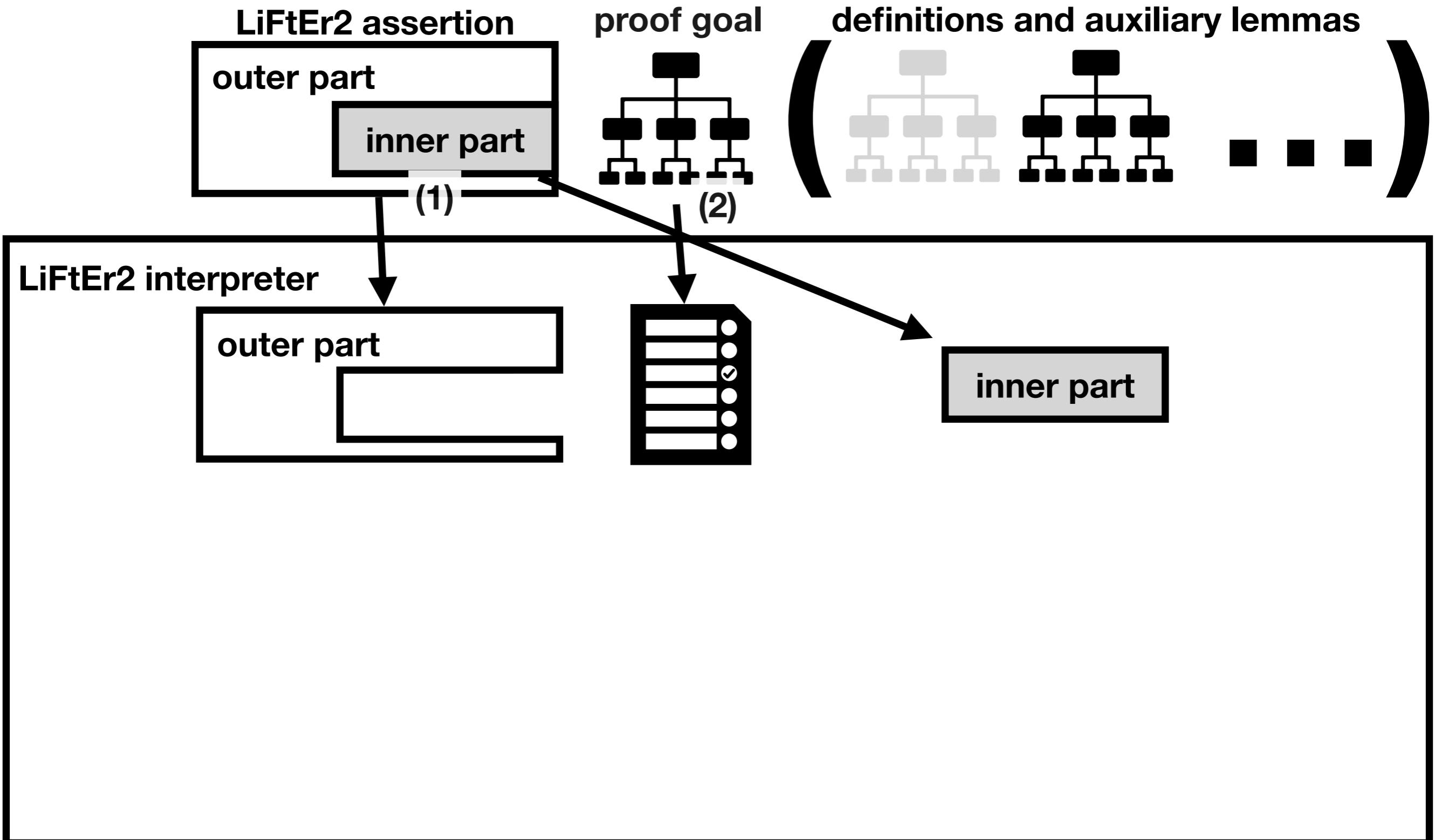


LiFtEr2 interpreter

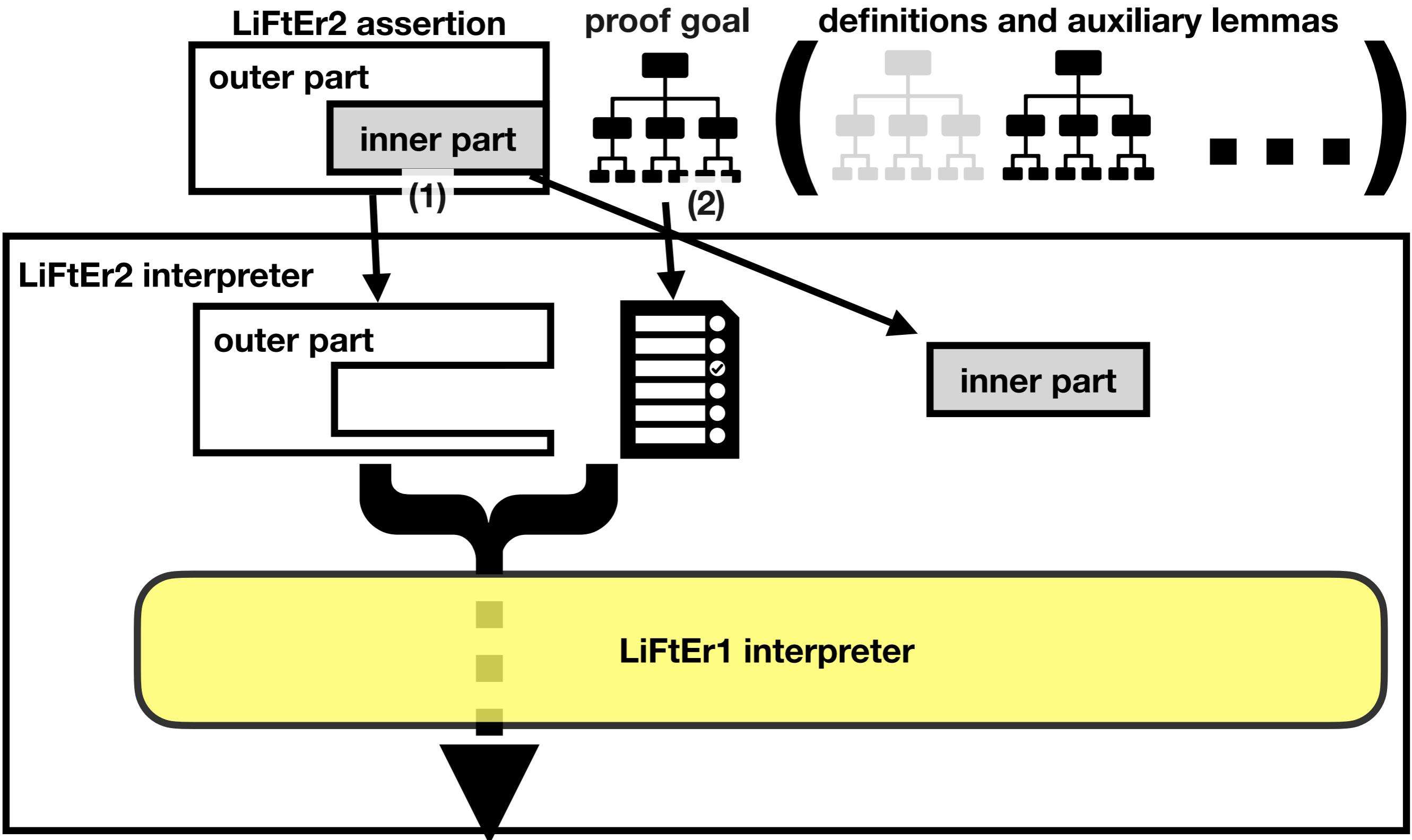
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



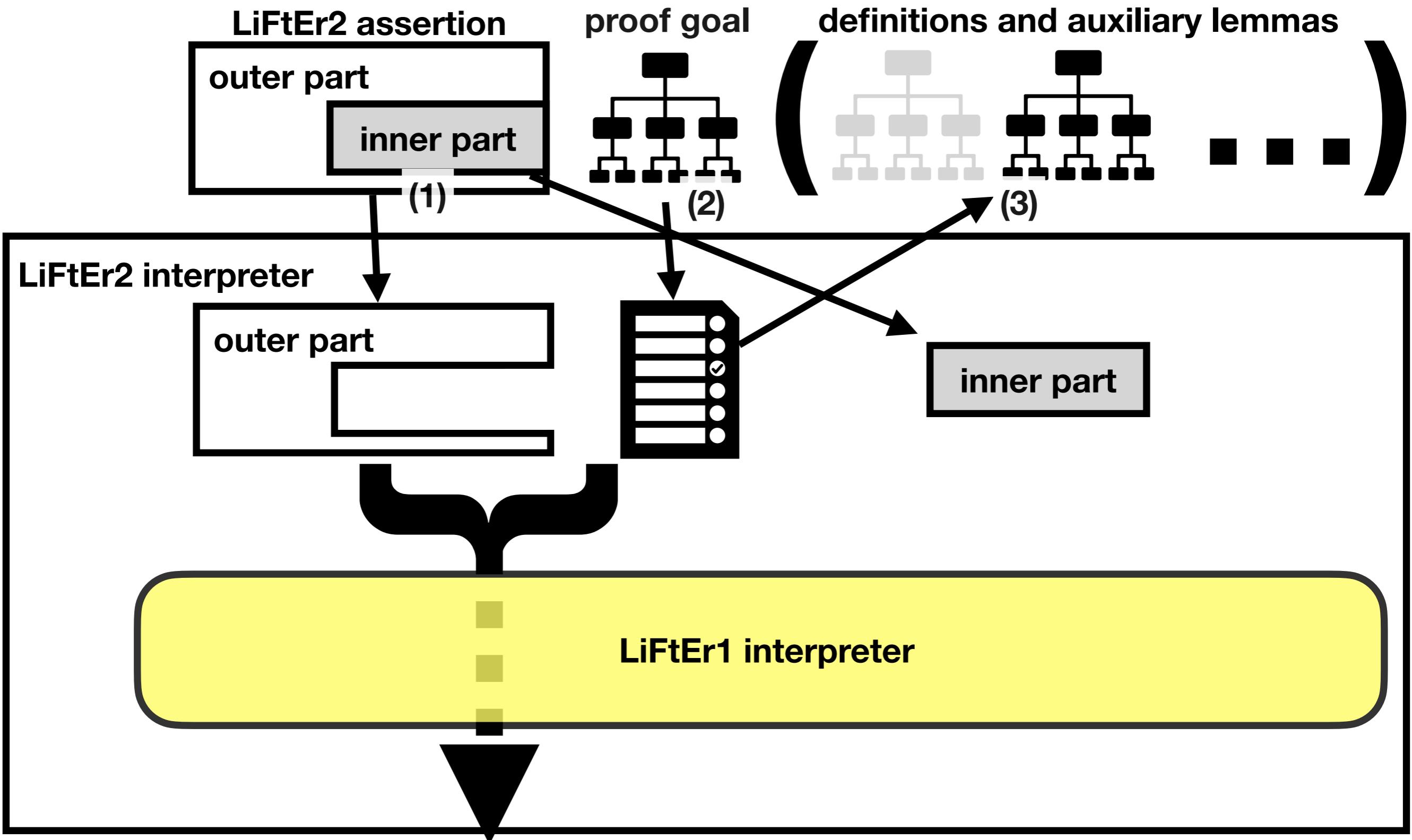
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



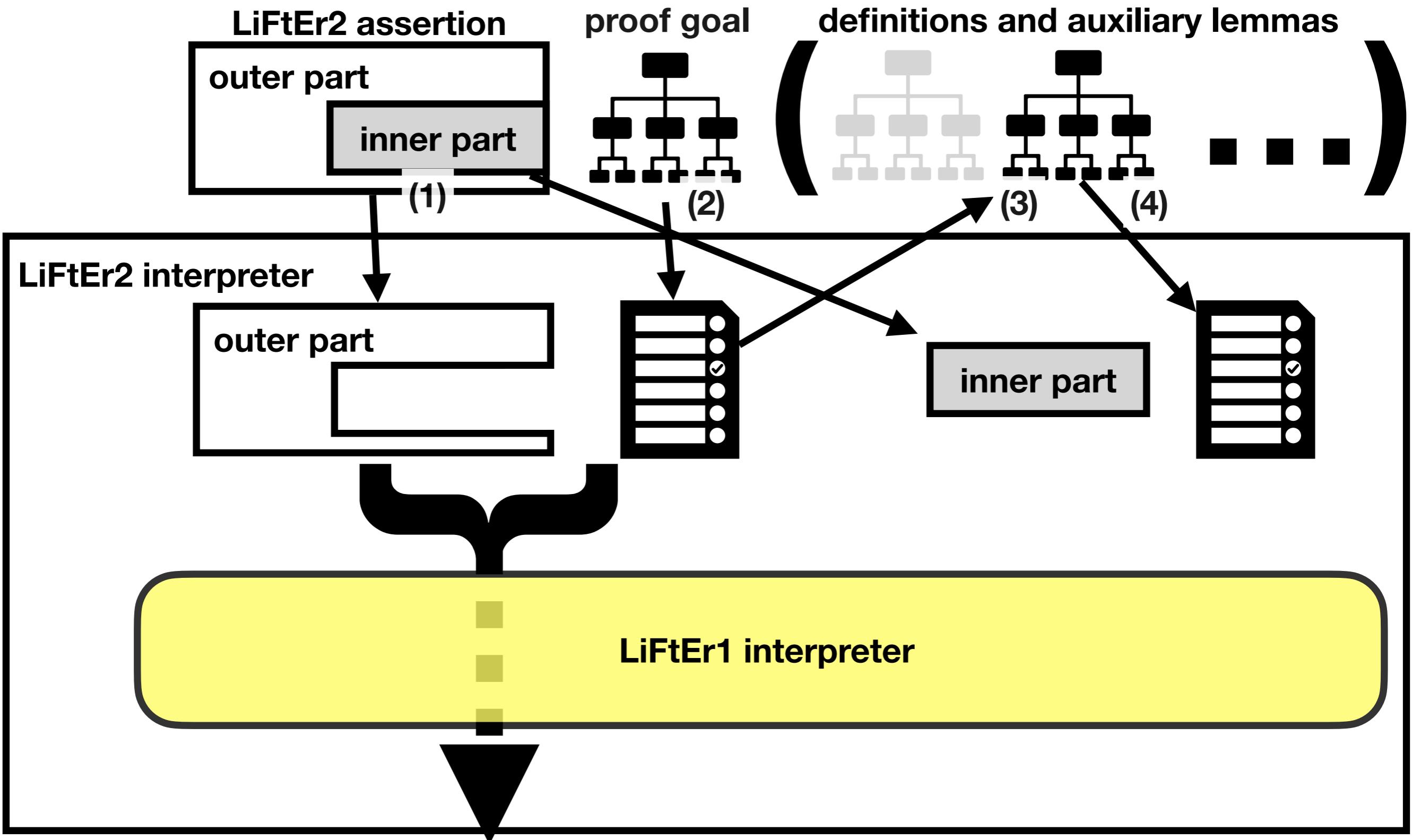
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



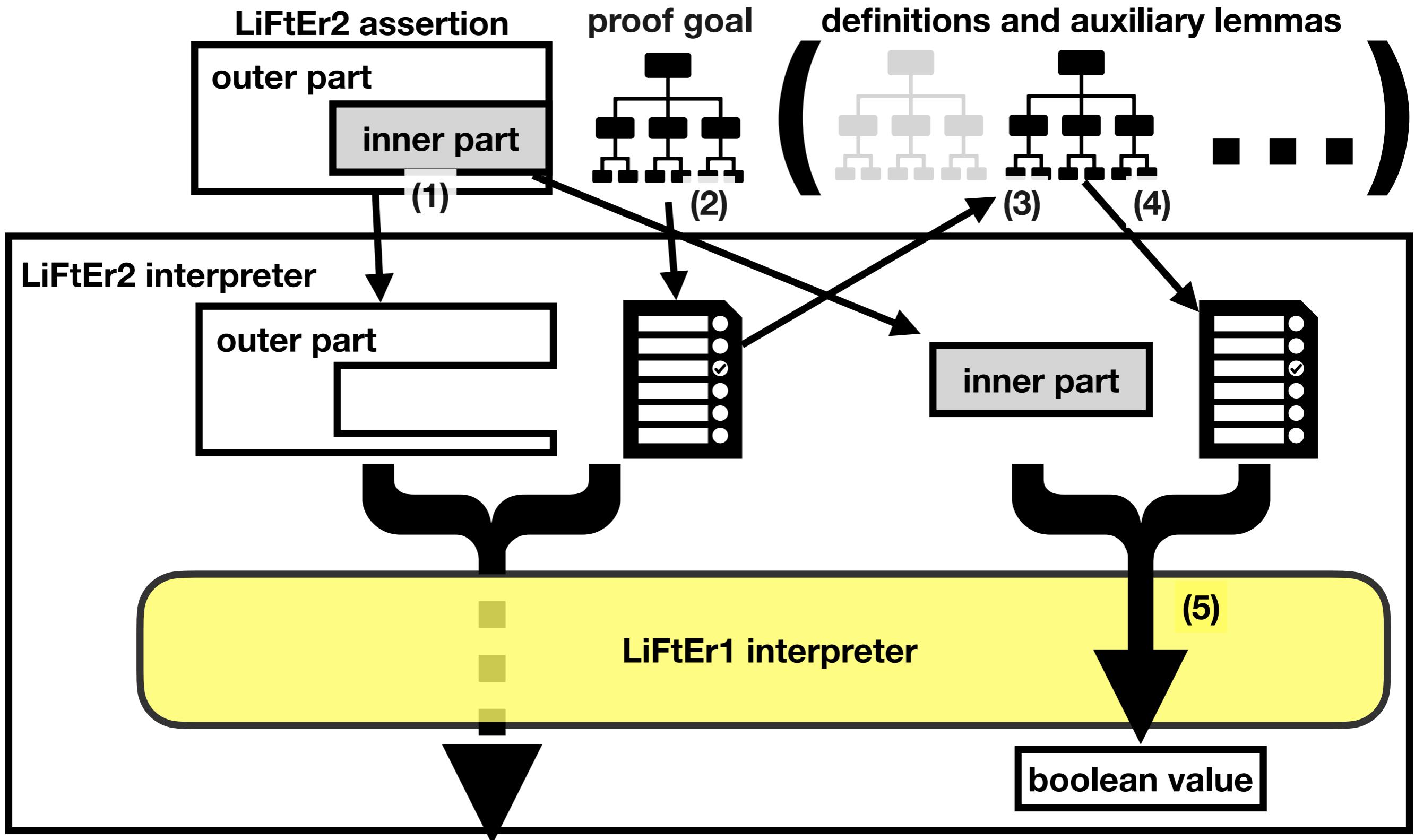
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



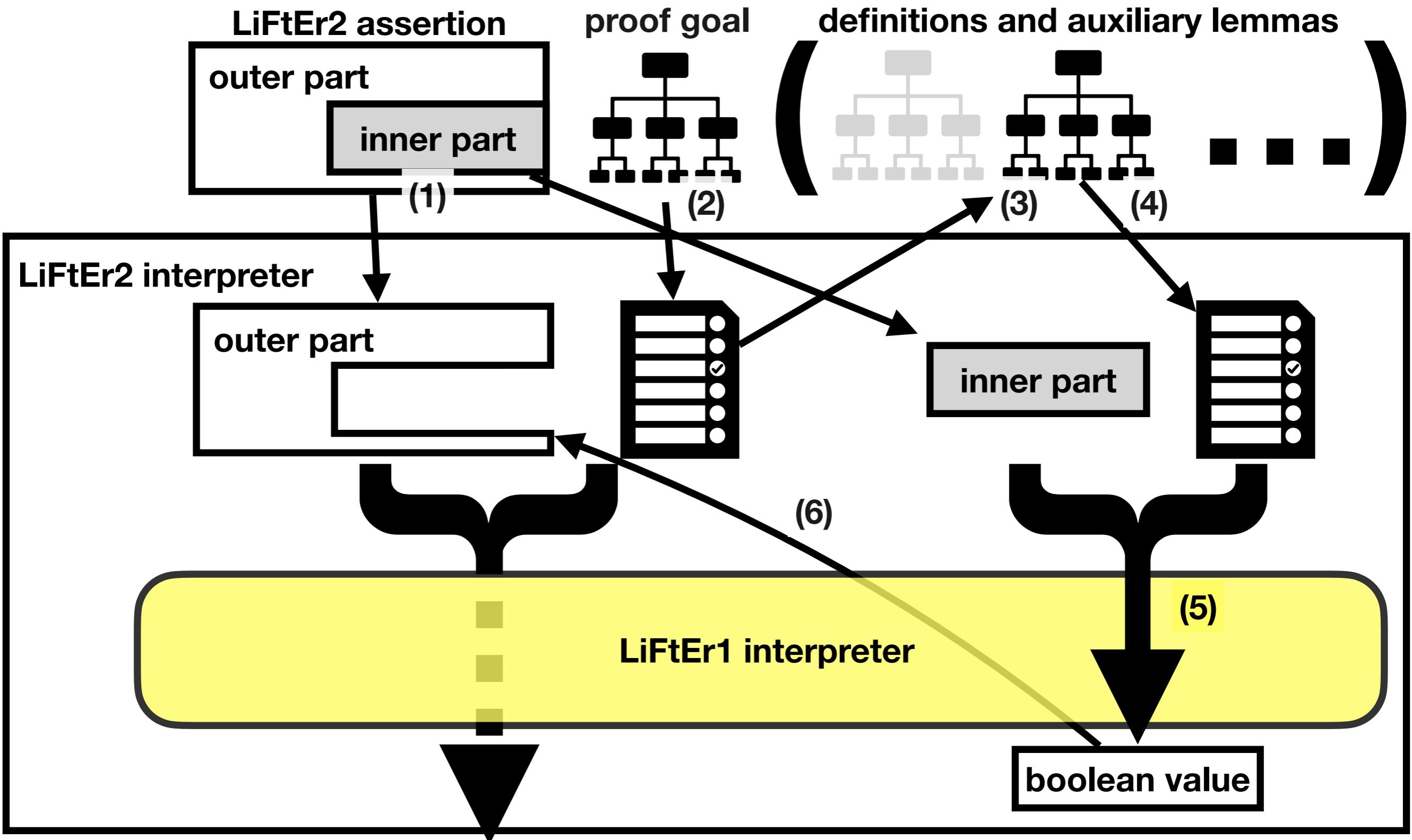
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



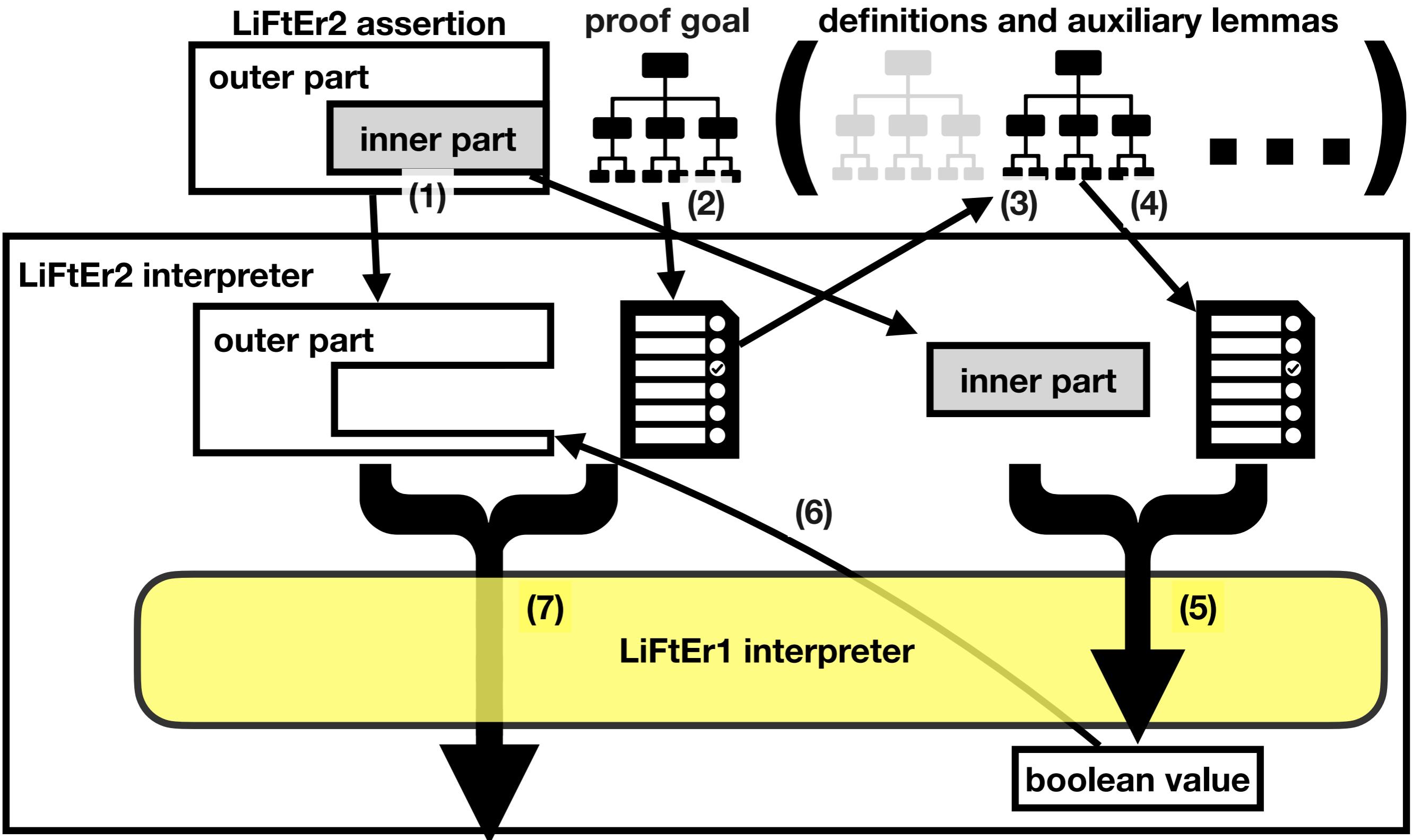
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



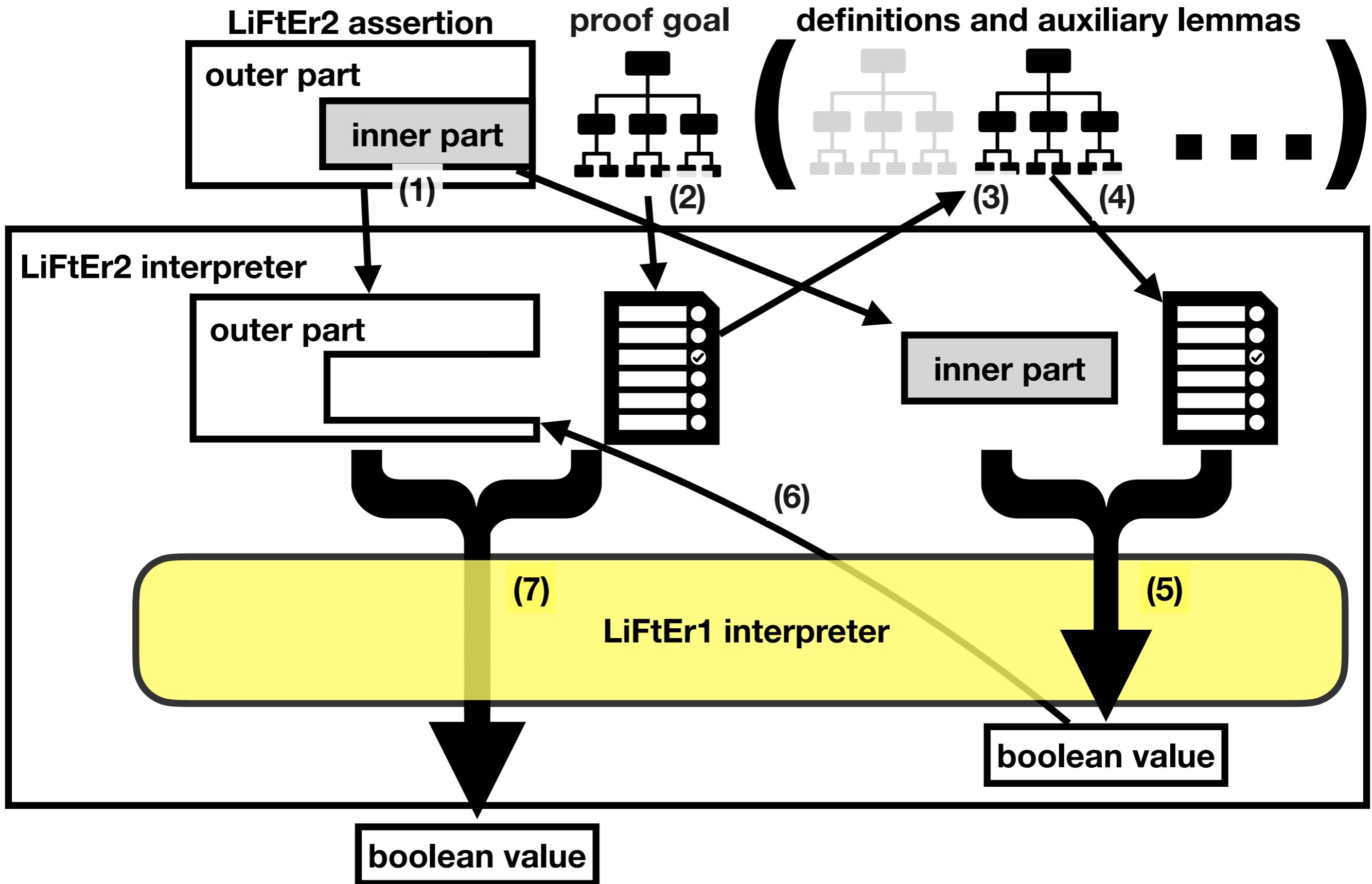
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



LiFtEr₂ (proof goal * induction arguments) -> bool
 * relevant definitions

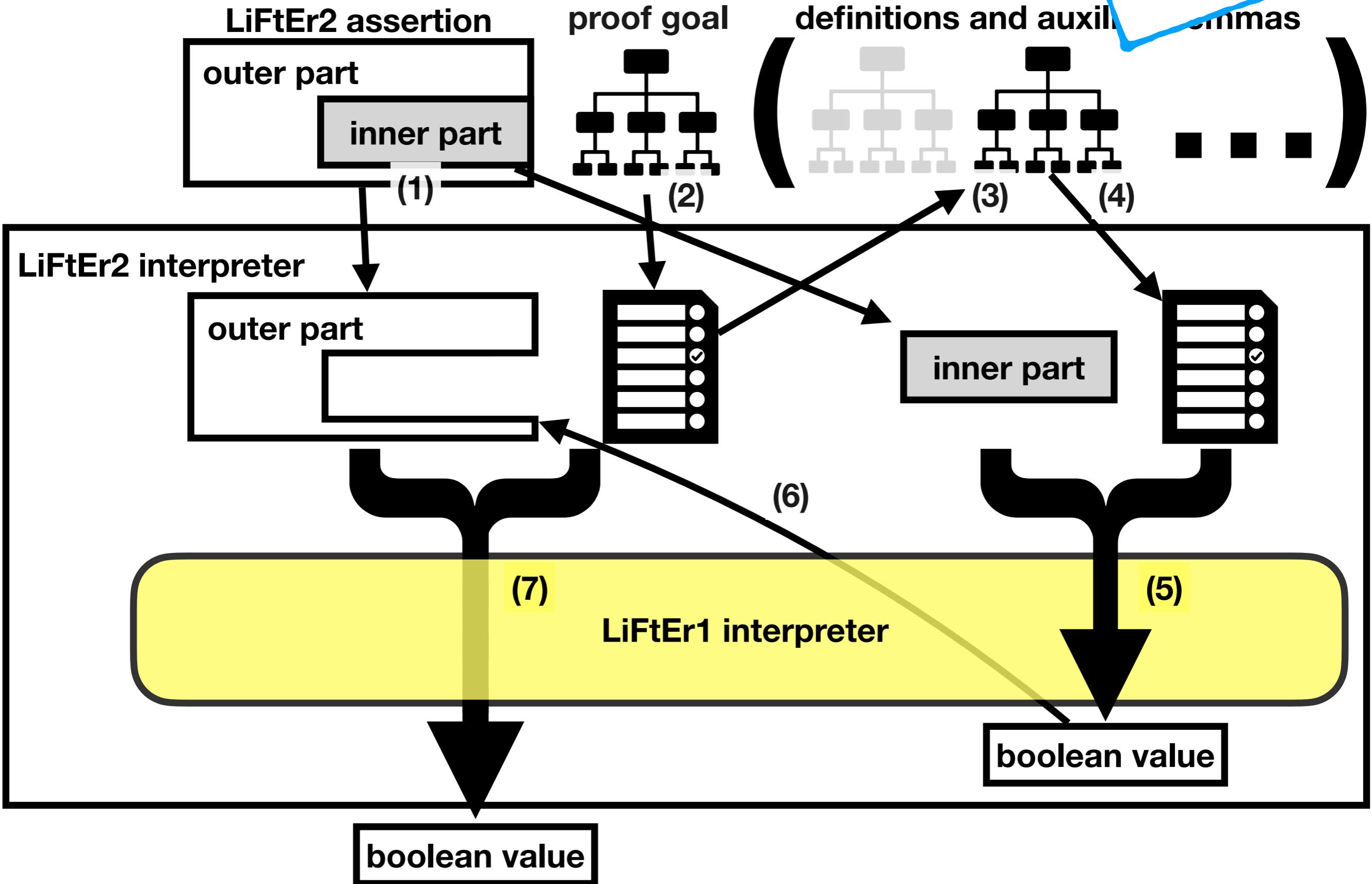


LiFtEr₂ (proof goal * induction arguments) -> bool
 * relevant definitions

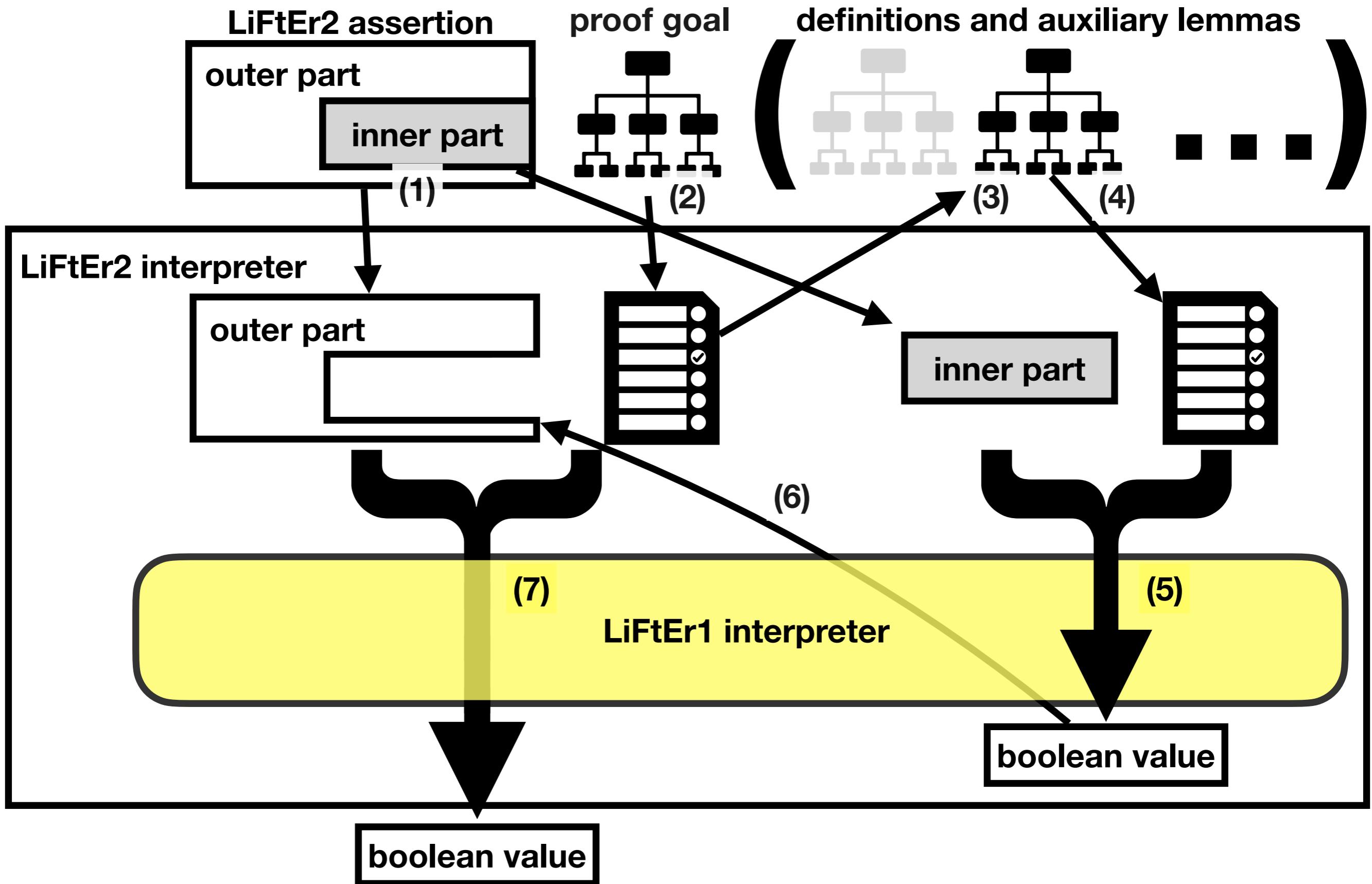


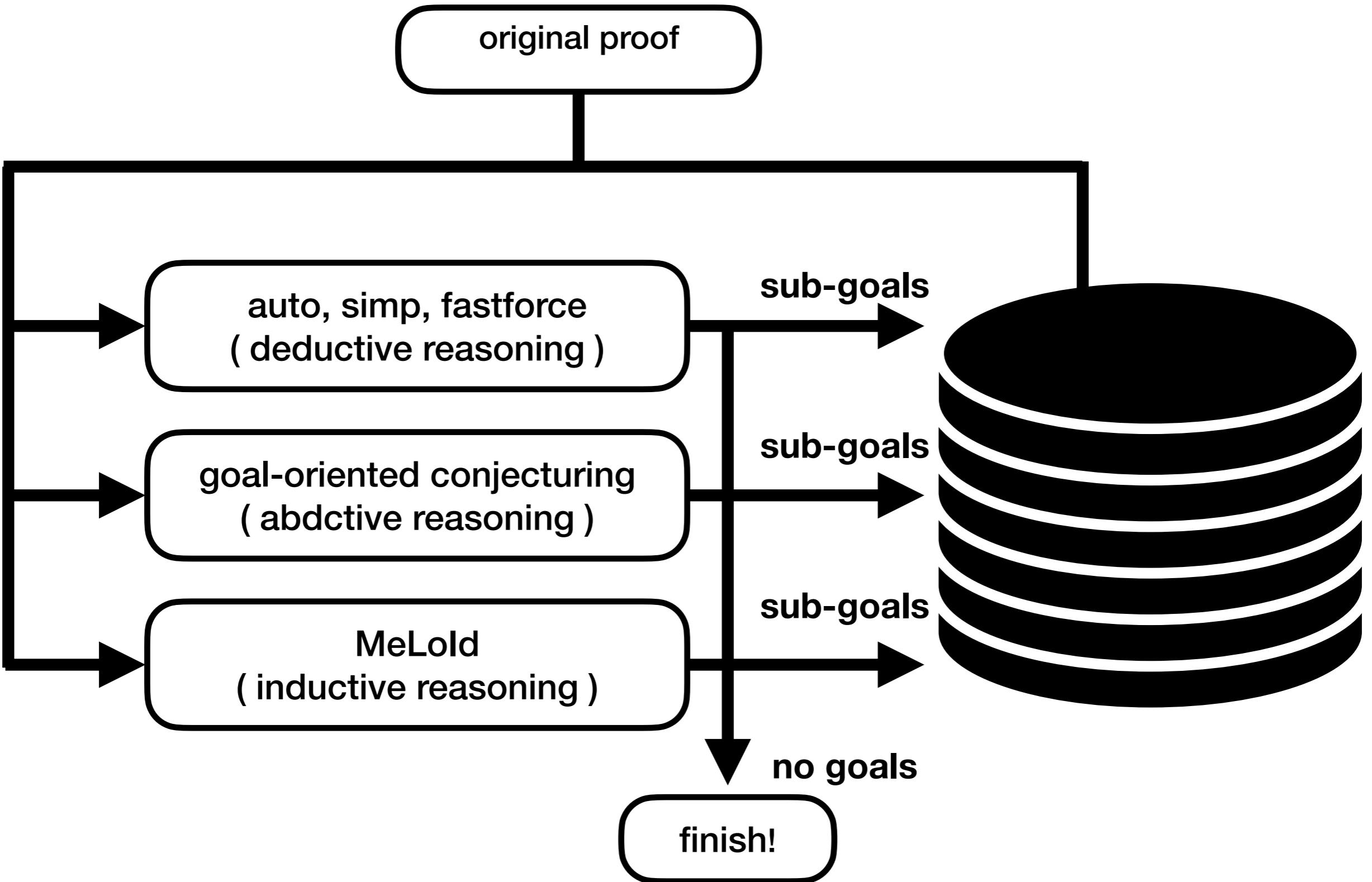
LiFtEr₂ (proof goal * induction arguments) -> bool
 * relevant definitions

WIP!



LiFtEr₂ (proof goal * induction arguments) -> bool
 * relevant definitions





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

goal

DEMO2

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

DEMO2

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

goal

Conjecture

Fastforce

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

Quickcheck



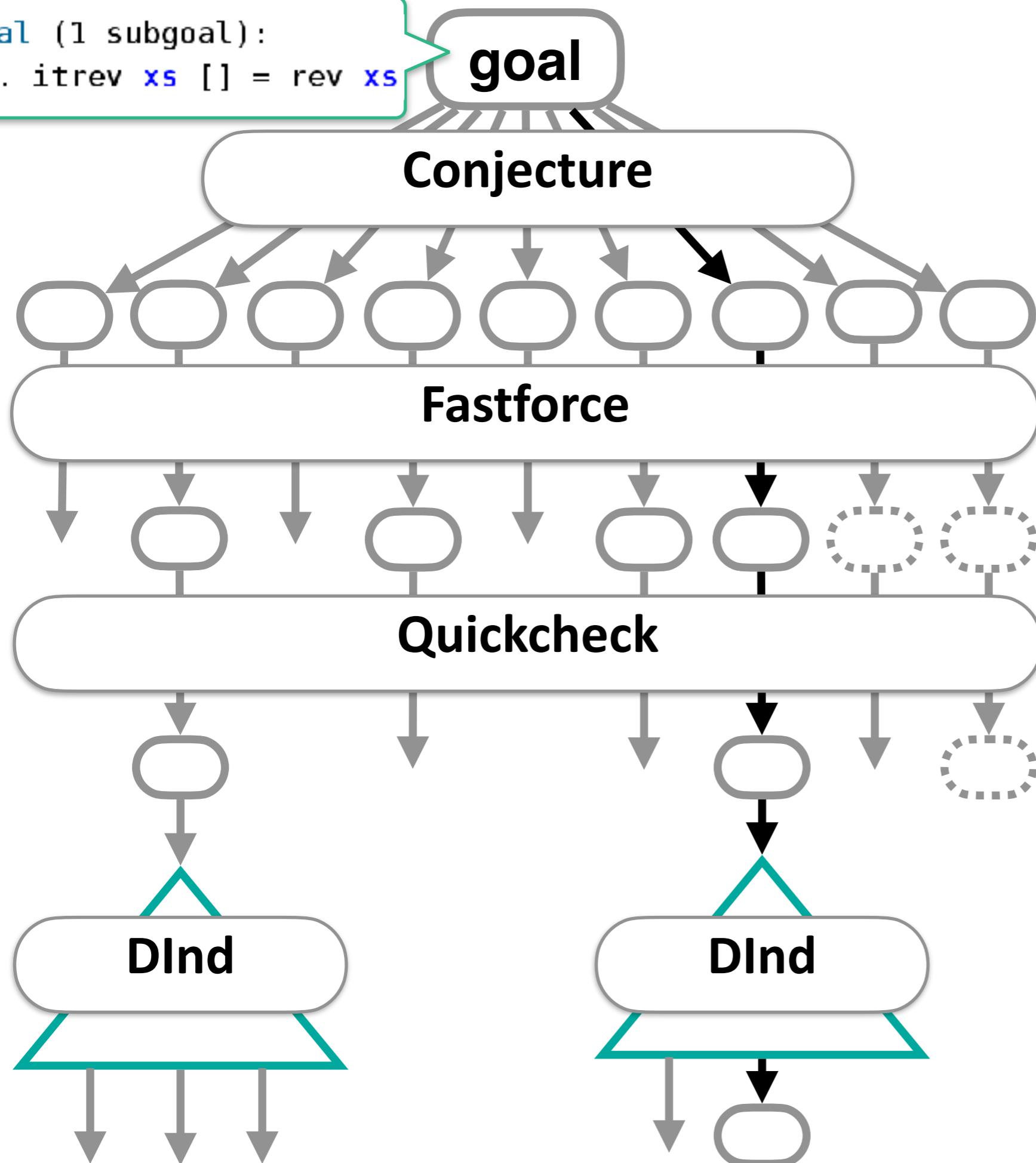
DInd

DInd



DEMO2

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



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

goal

apply (subgoal_tac

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

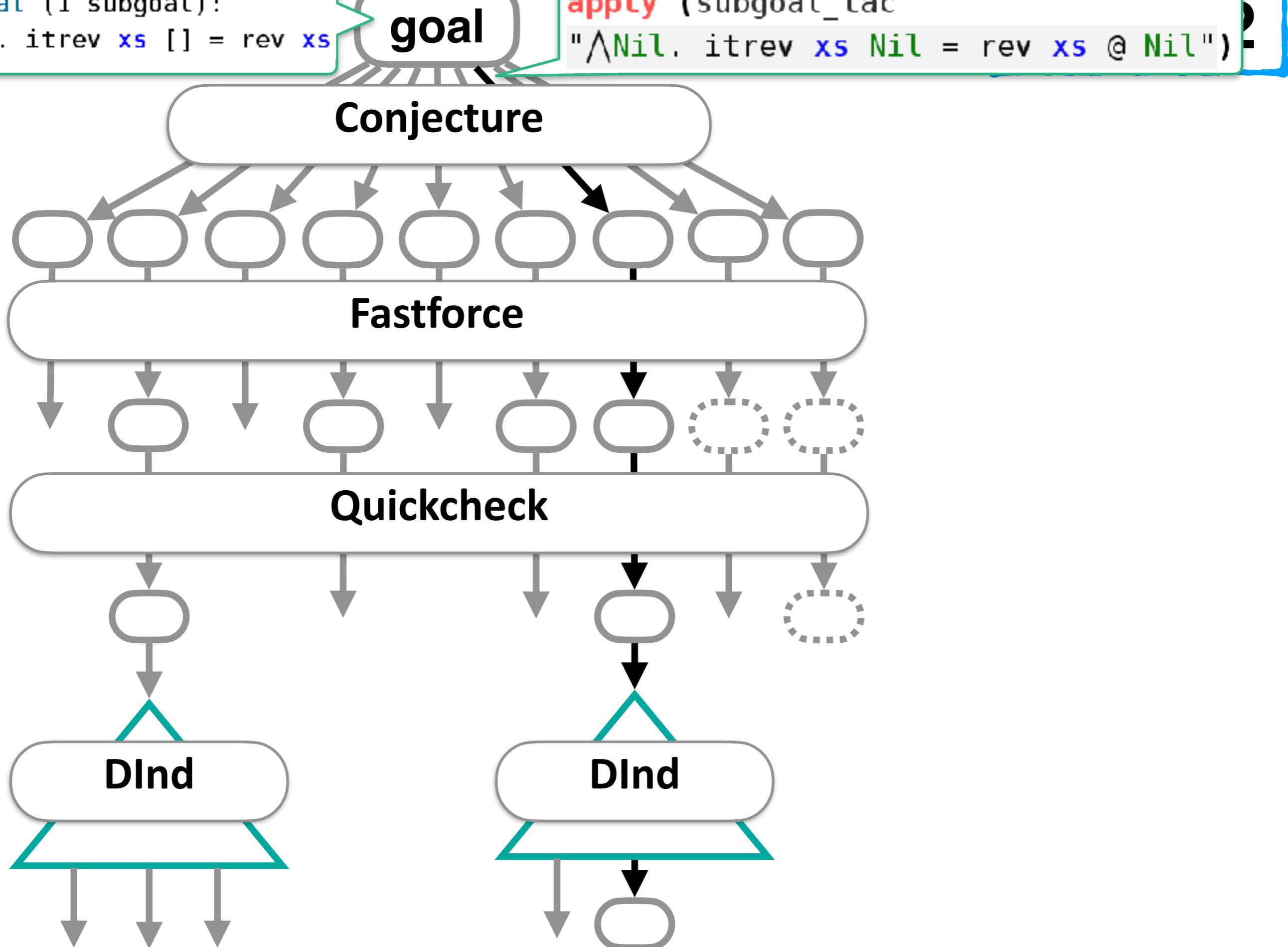
Conjecture

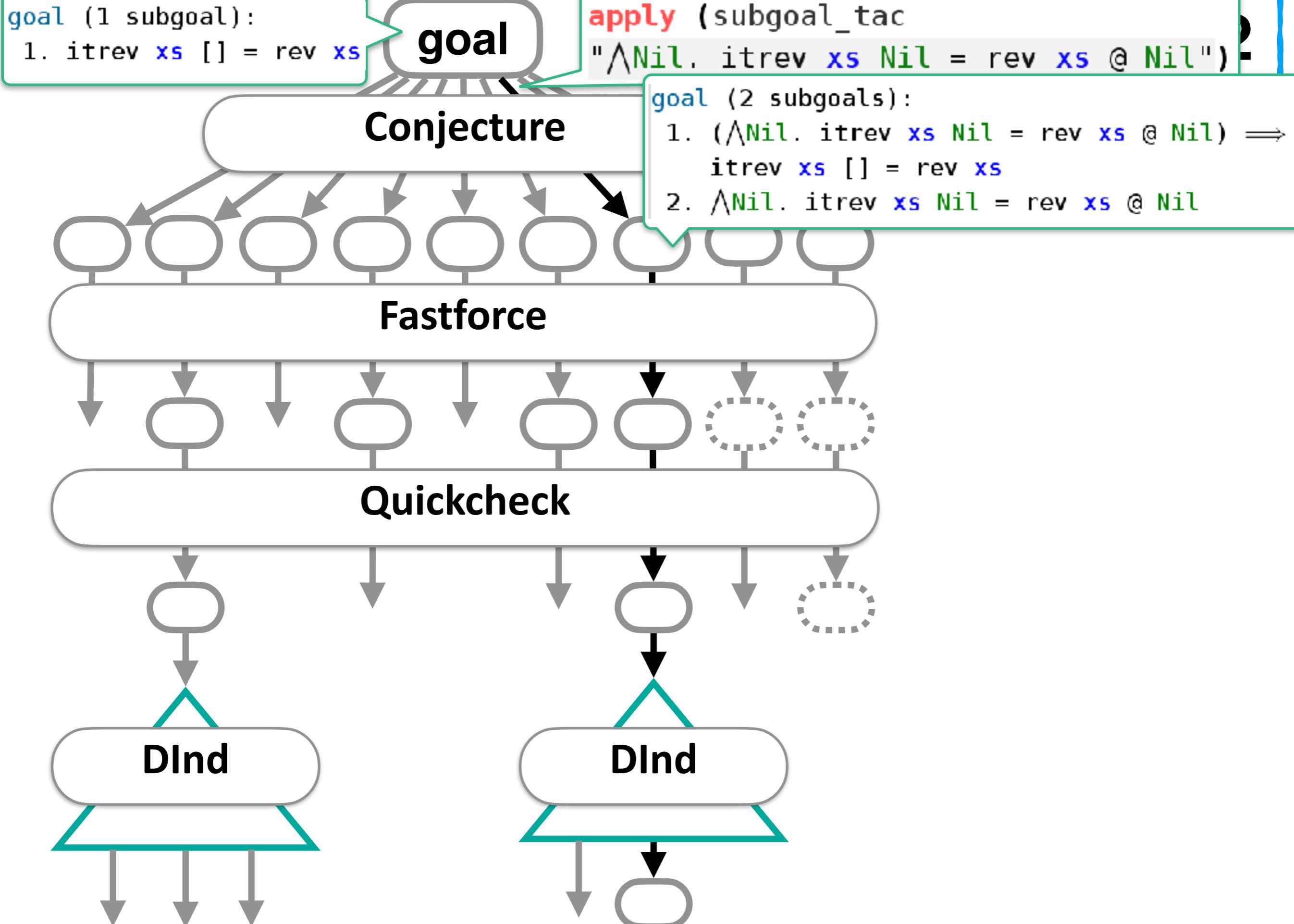
Fastforce

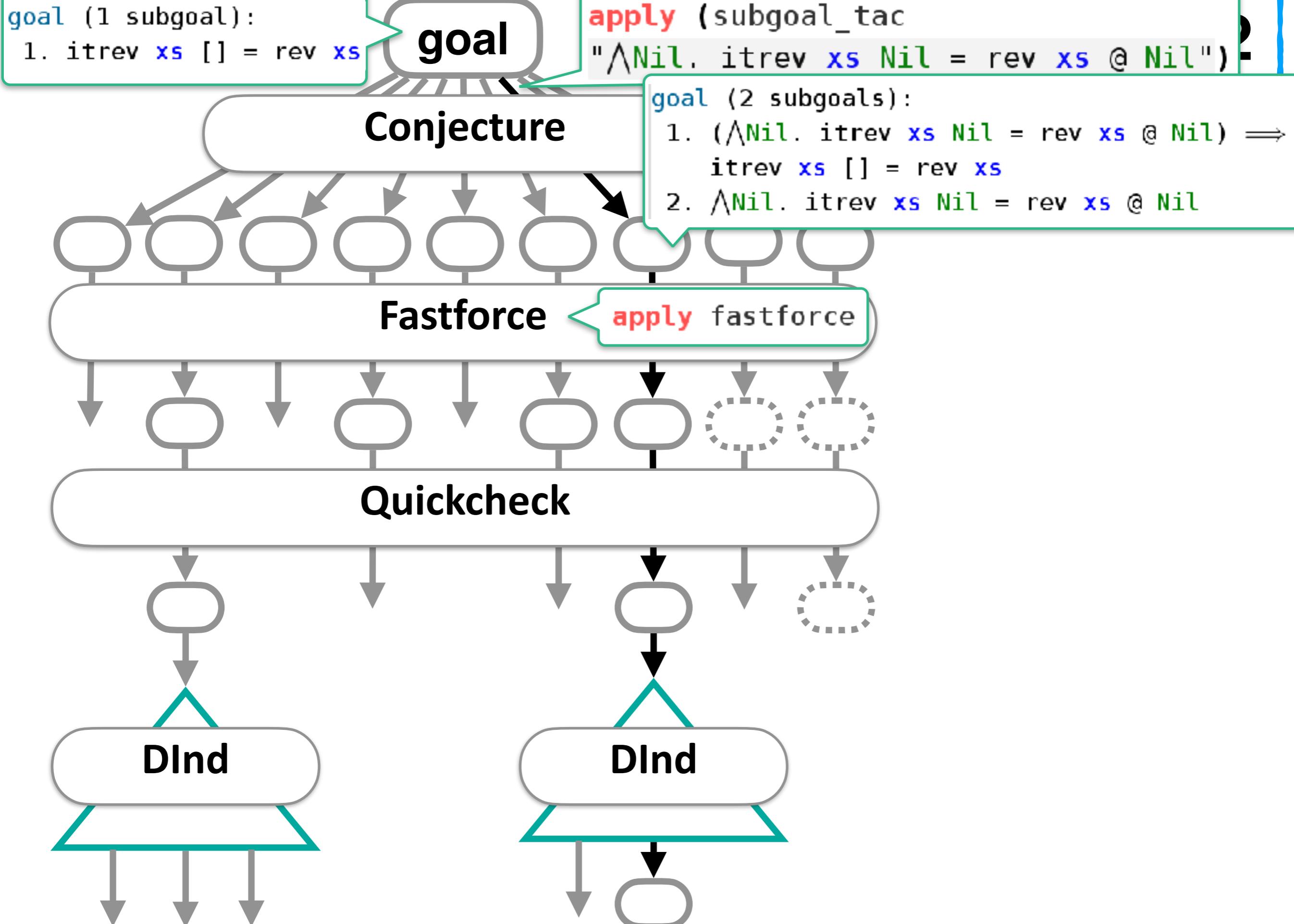
Quickcheck

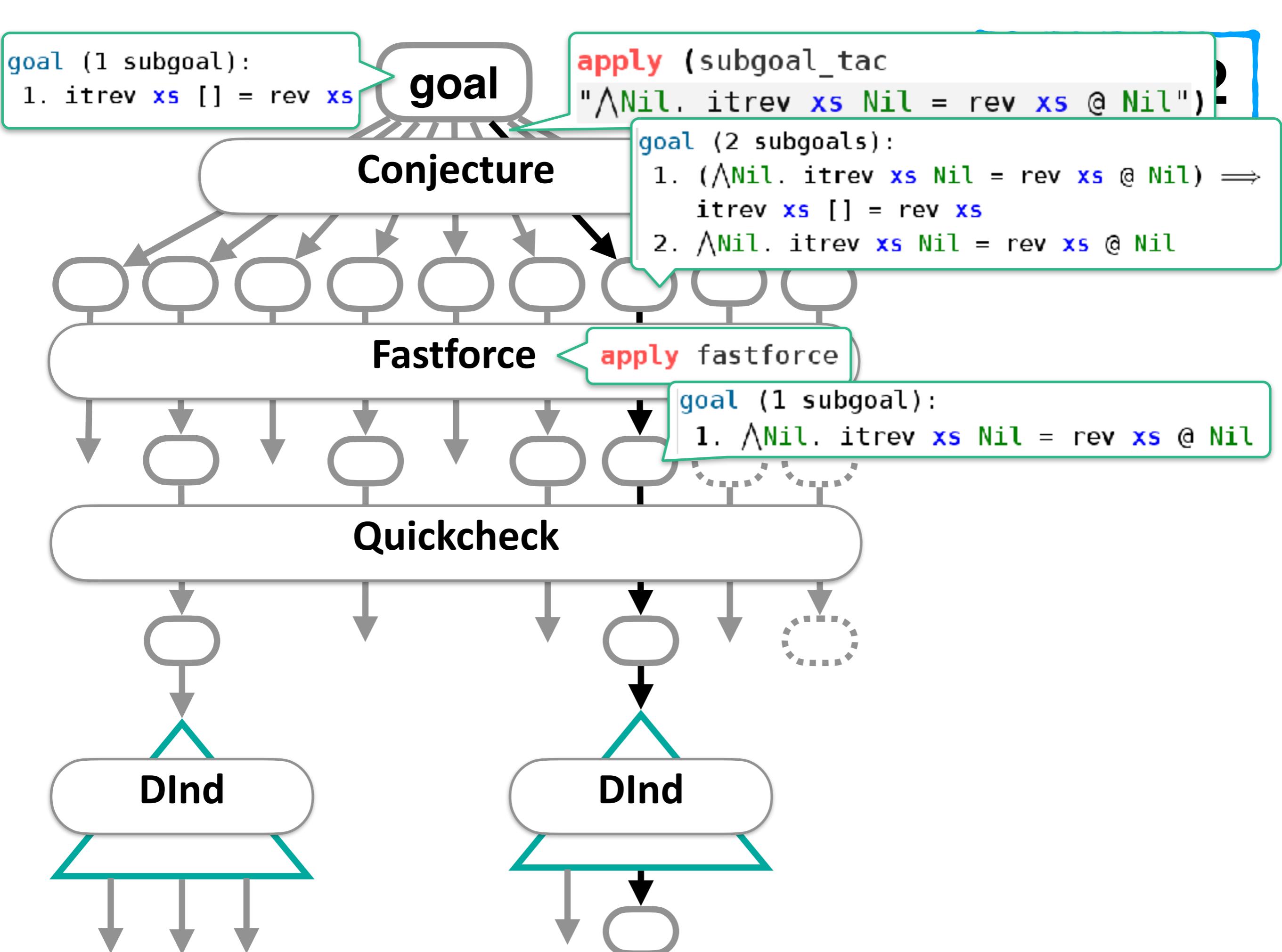
DInd

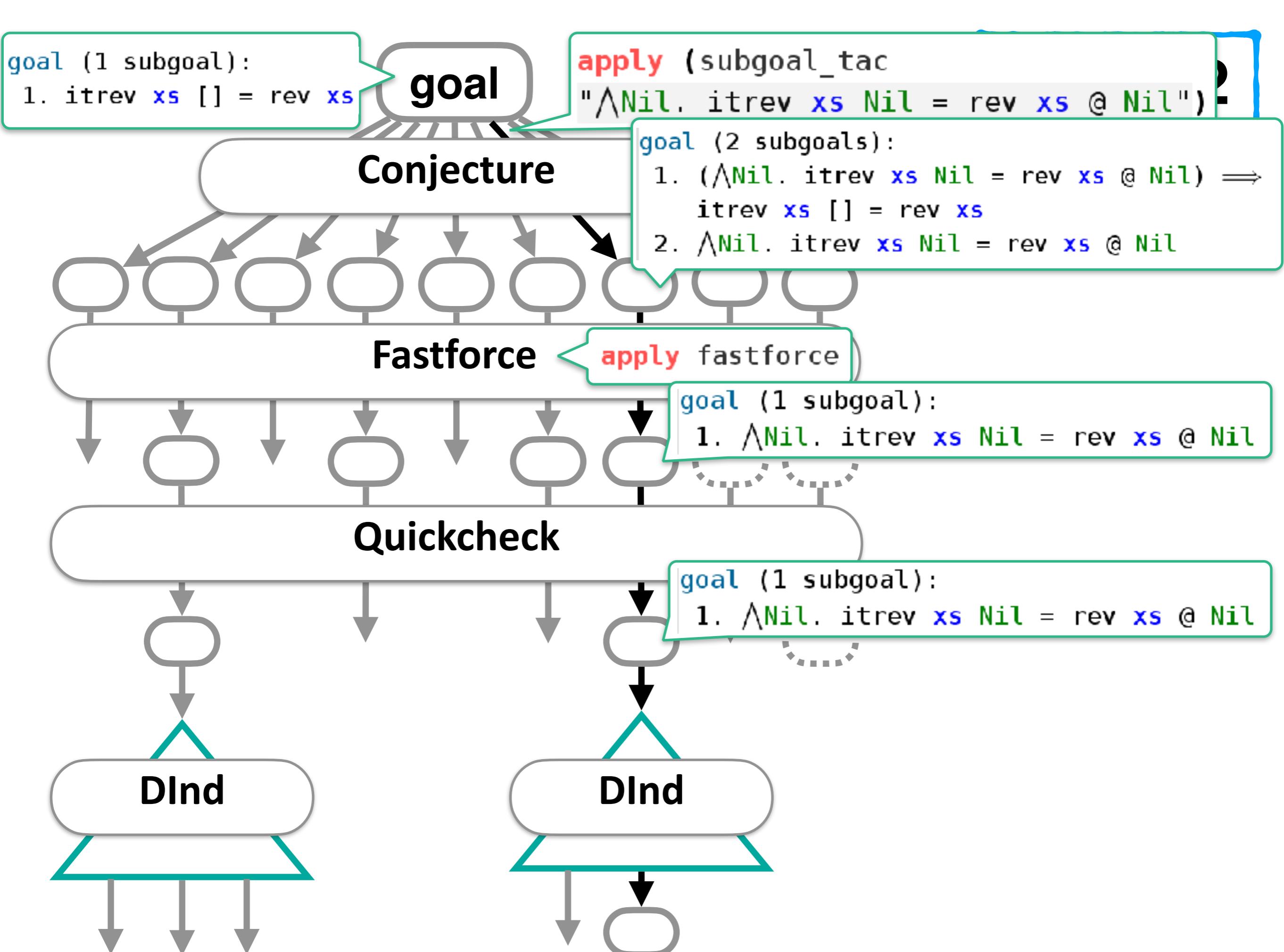
DInd

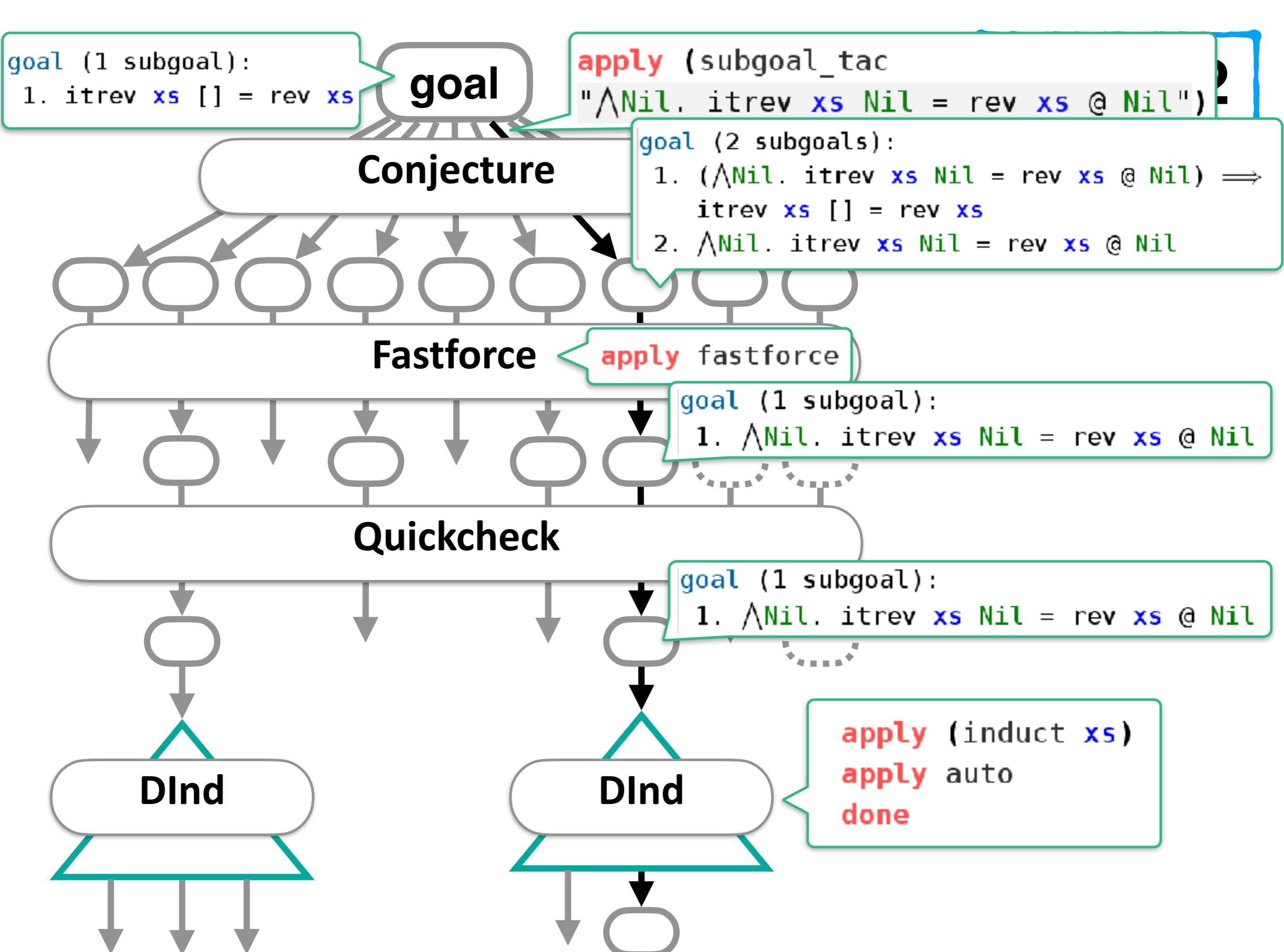


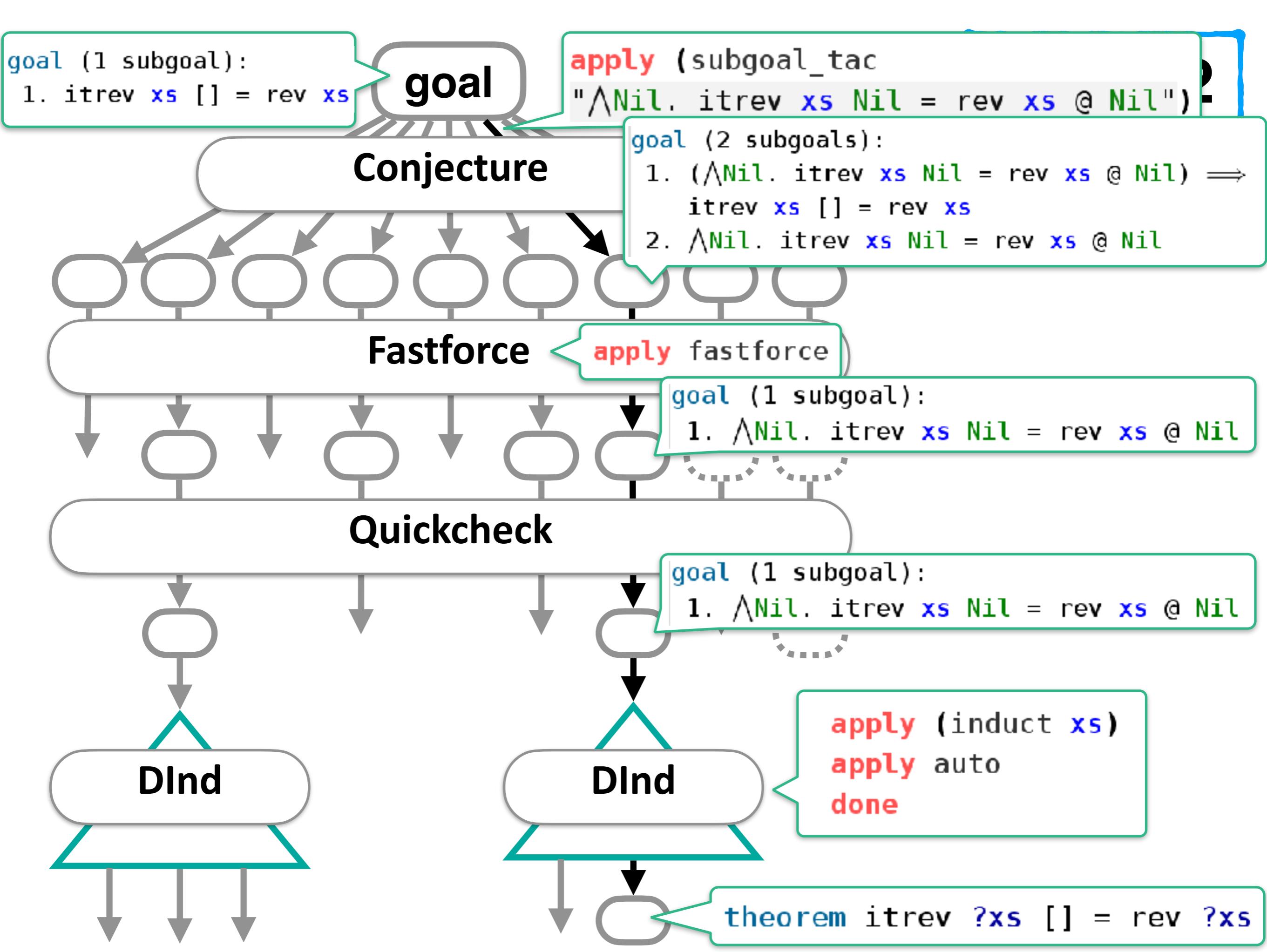


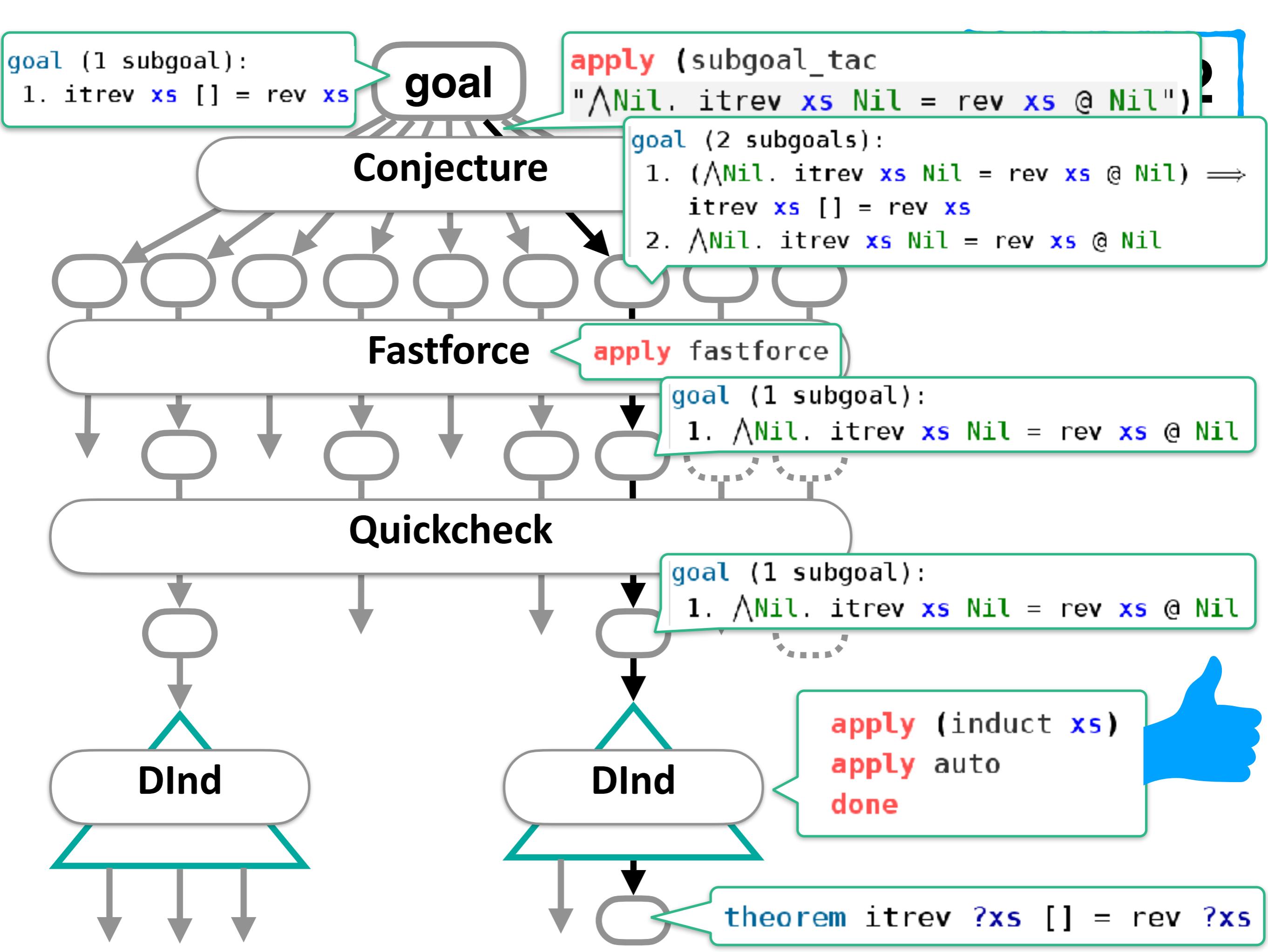






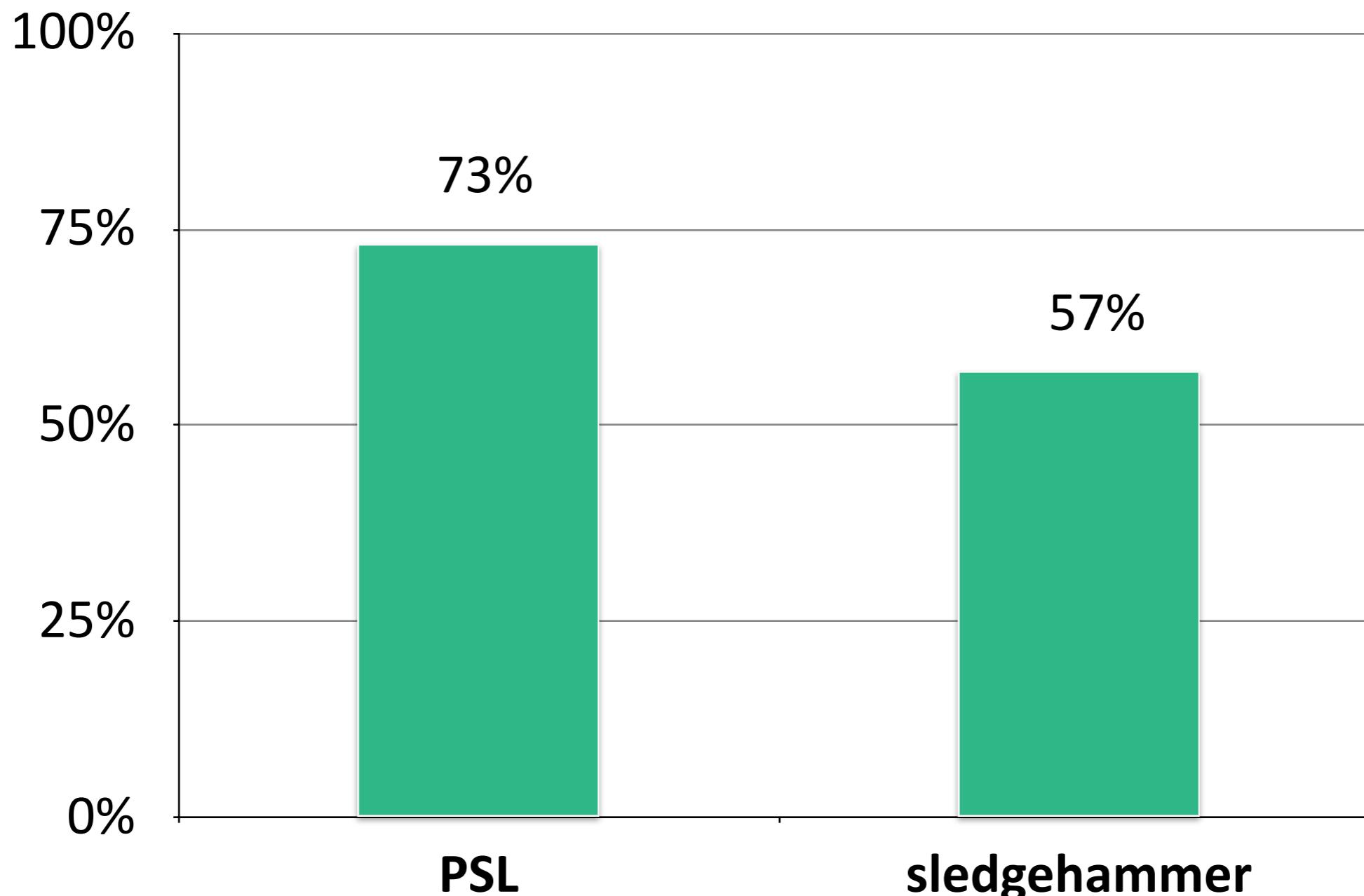






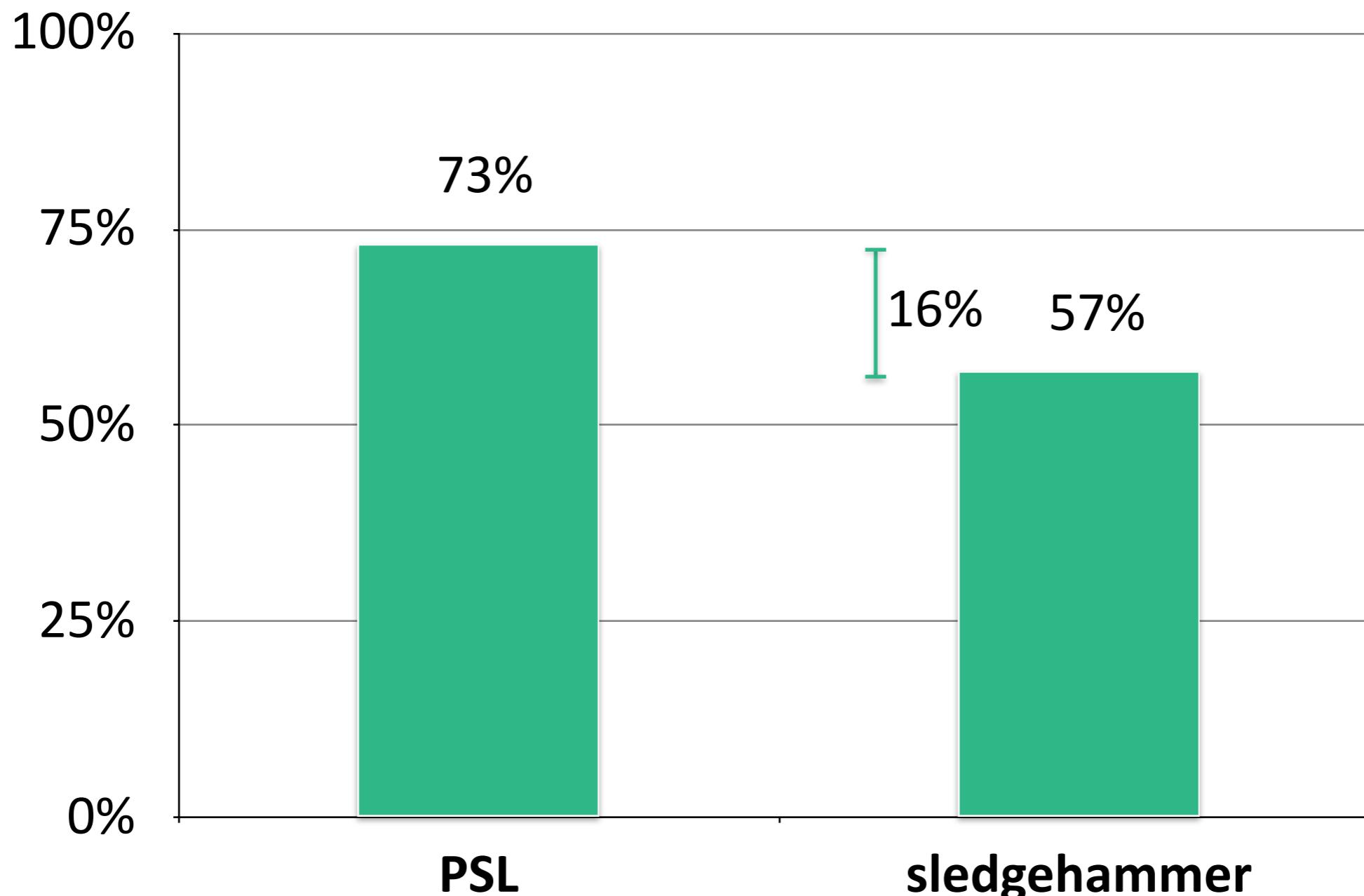
PSL vs sledgehammer

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



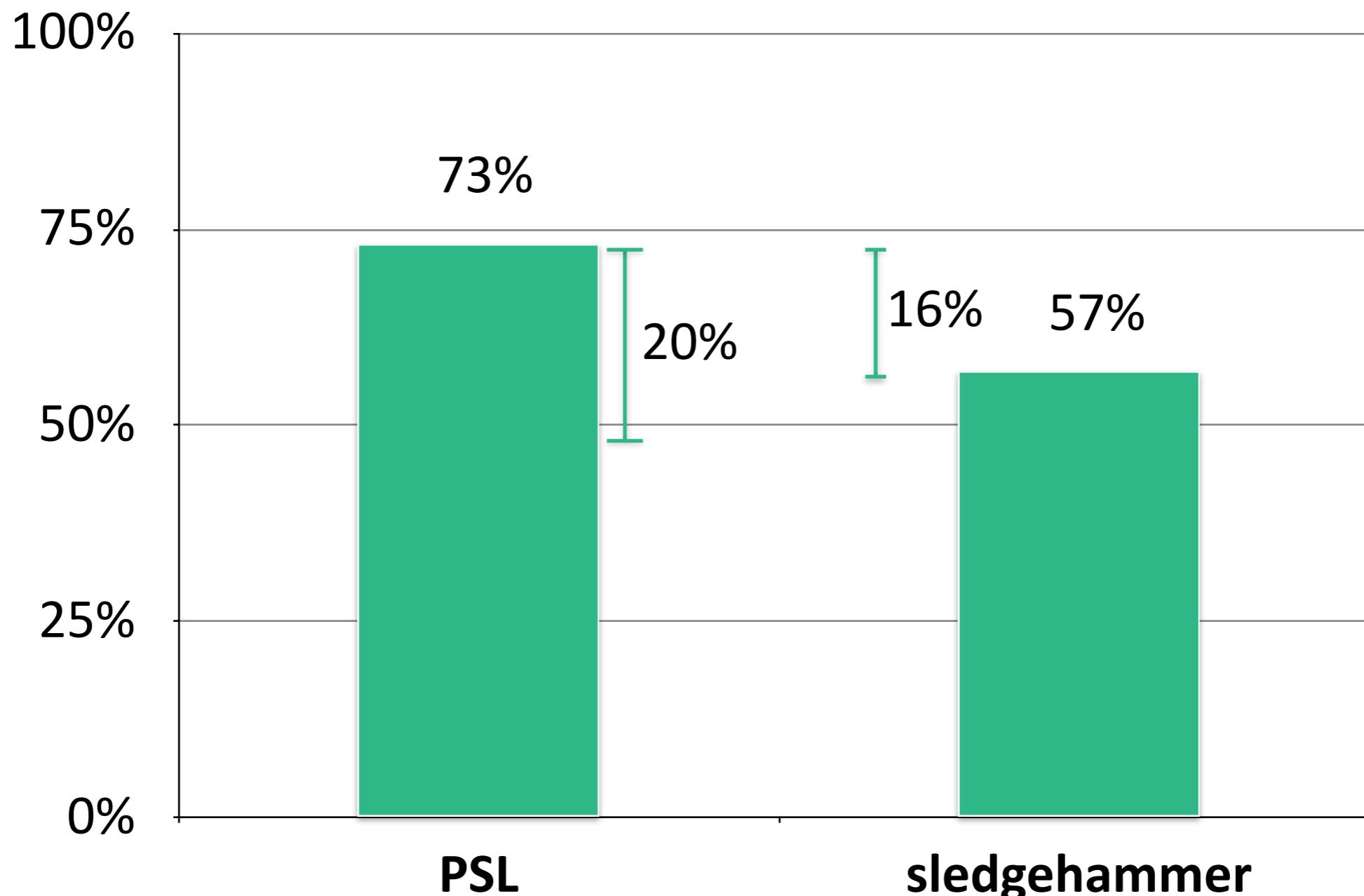
PSL vs sledgehammer

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

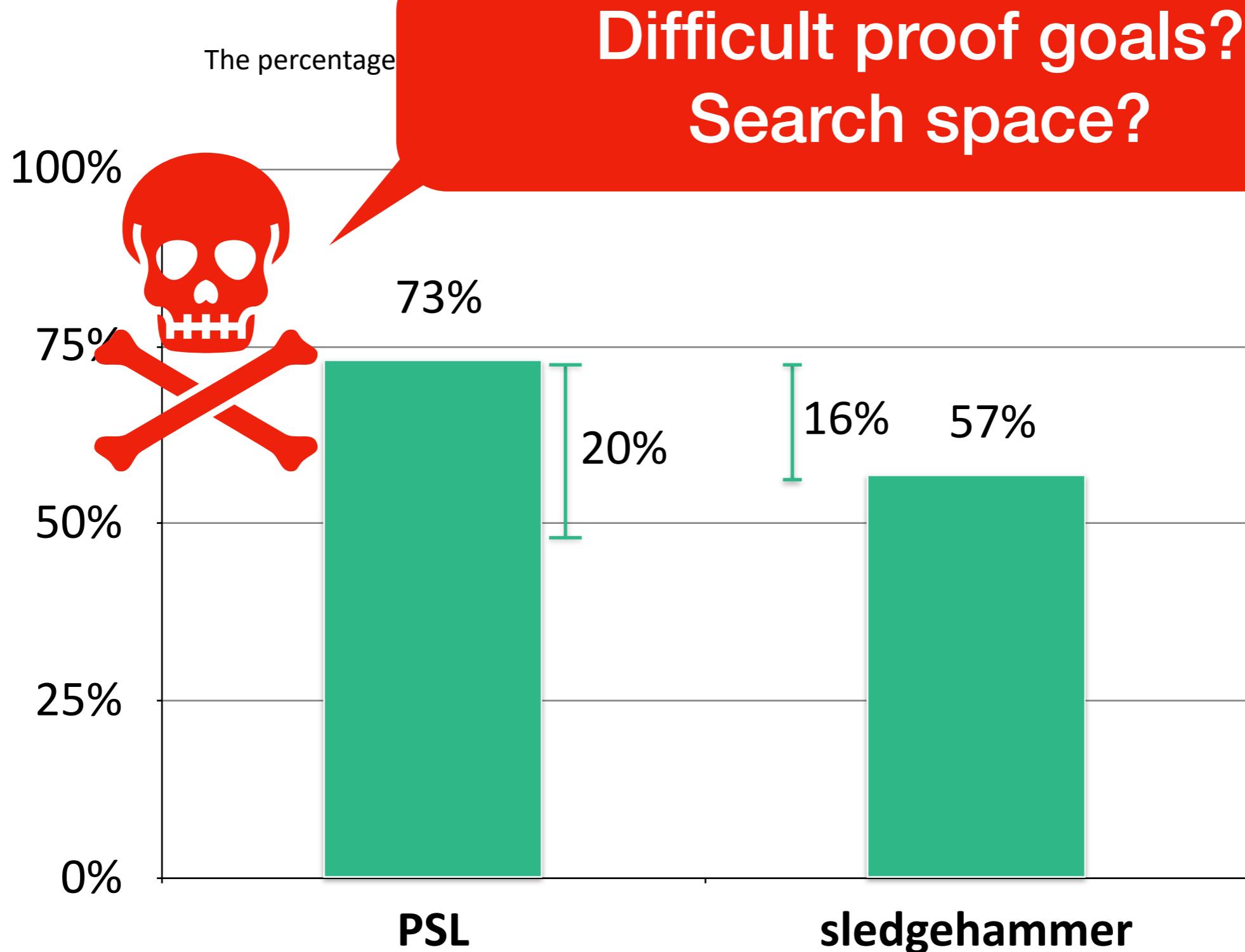


PSL vs sledgehammer

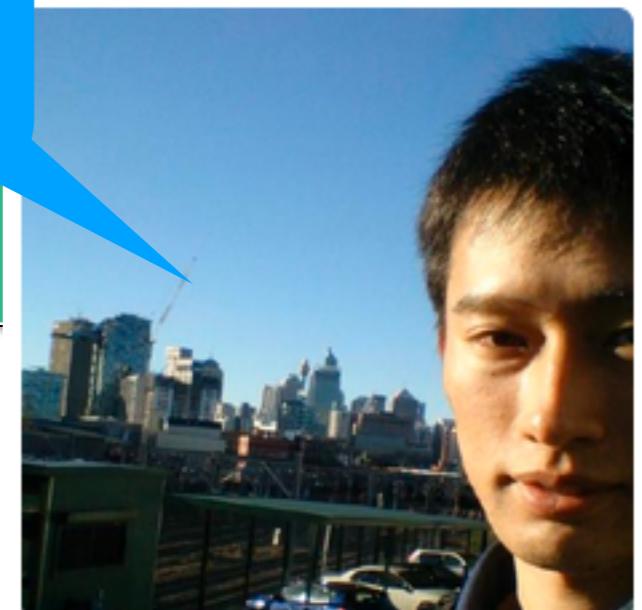
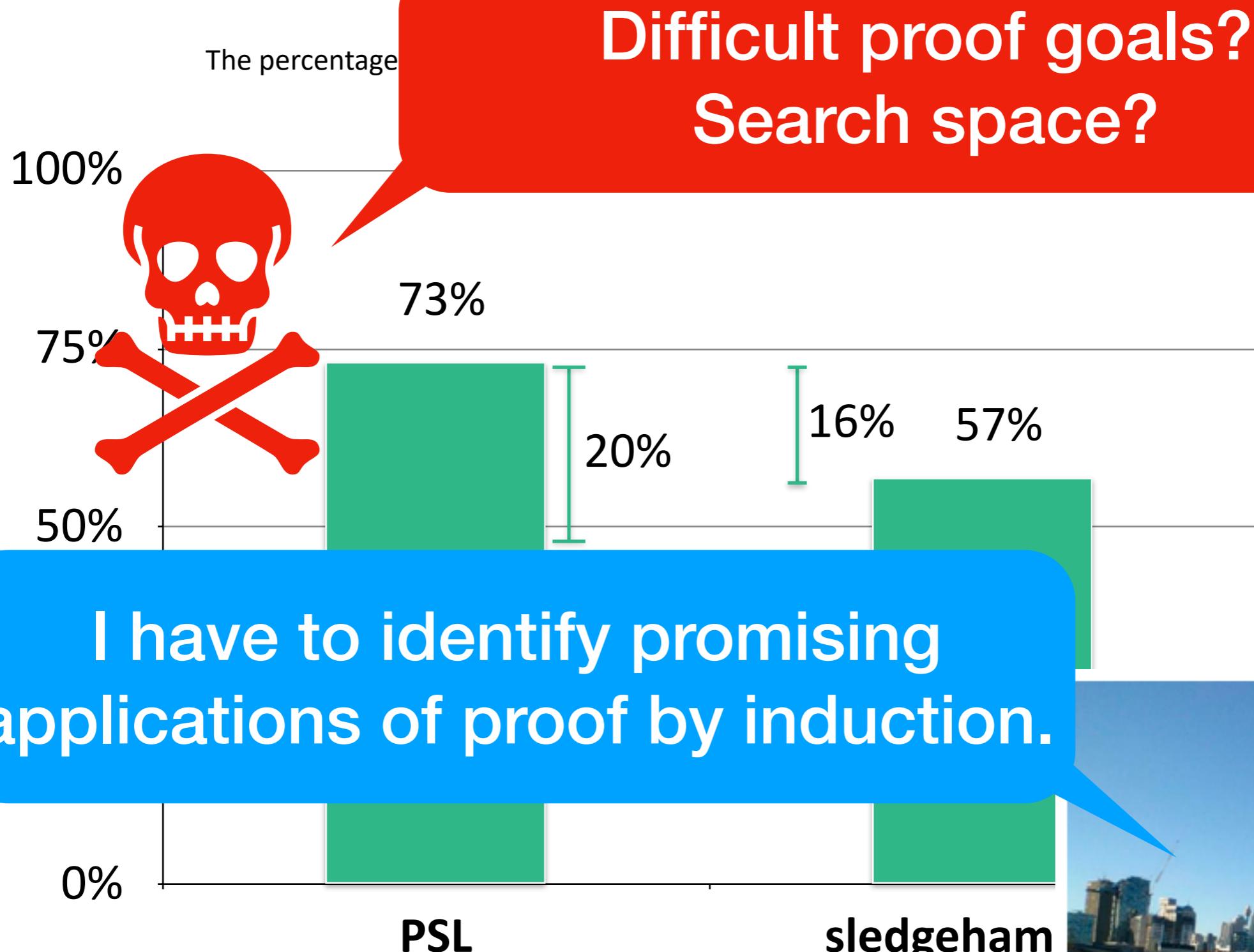
The percentage of automatically proved obligations out of 1526 proof obligations
(timeout = 300s)



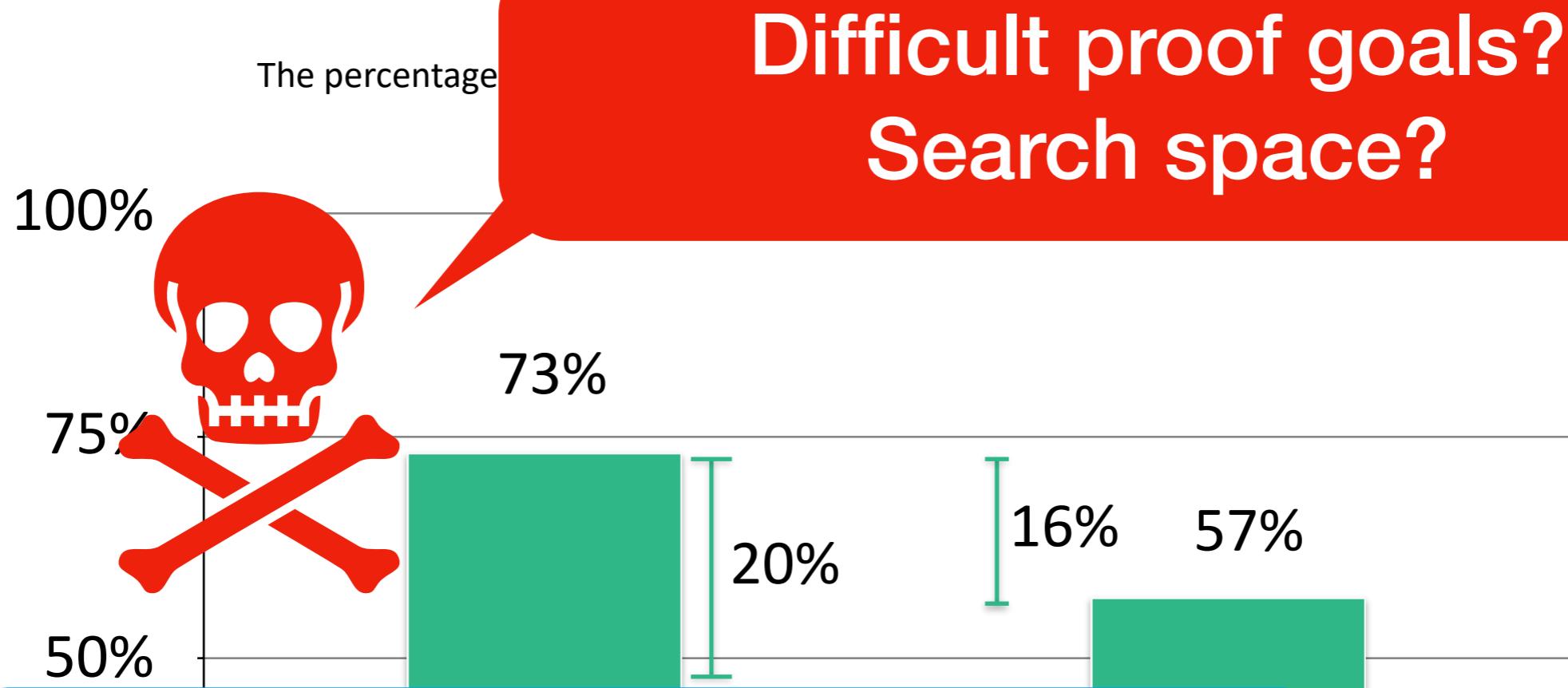
PSL vs sledgehammer



PSL vs sledgehammer



PSL vs sleddaehammer



I have to identify promising applications of proof by induction.

without completing a proof search



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
```