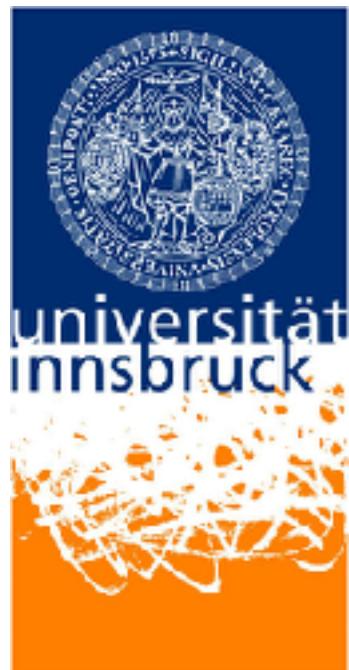


Towards United Reasoning for Automatic Induction in Isabelle/HOL



Yutaka Nagashima
yutaka.nagashima@cvut.cz
YutakangJ for Japanese
YutakangE for English

Yutaka Nagashima
yutaka.nagashima@cvut.cz
YutakangJ for Japanese
YutakangE for English

Yutaka Nagashima
yutaka.nagashima@cvut.cz
YutakangJ for Japanese
YutakangE for English

Yutaka Nagashima
yutaka.nagashima@cvut.cz
YutakangJ for Japanese
YutakangE for English

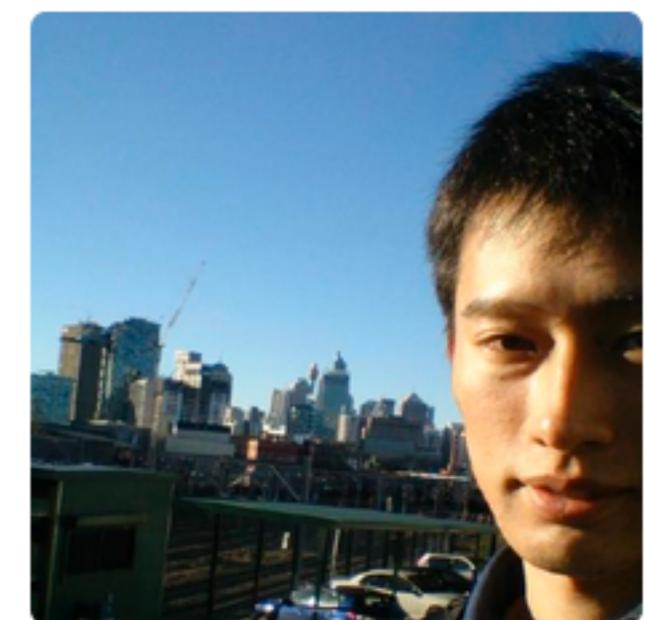
Yutaka Nagashima
yutaka.nagashima@cvut.cz
YutakangJ for Japanese
YutakangE for English

Yutaka Nagashima
yutaka.nagashima@cvut.cz
YutakangJ for Japanese
YutakangE for English

Yutaka Nagashima
yutaka.nagashima@cvut.cz
YutakangJ for Japanese
YutakangE for English

Yutaka Nagashima
yutaka.nagashima@cvut.cz
YutakangJ for Japanese
YutakangE for English

Yutaka Nagashima
yutaka.nagashima@cvut.cz
YutakangJ for Japanese
YutakangE for English



Yutaka Ng

Why proof by induction?

developer

proof assistant / ITP

implementation

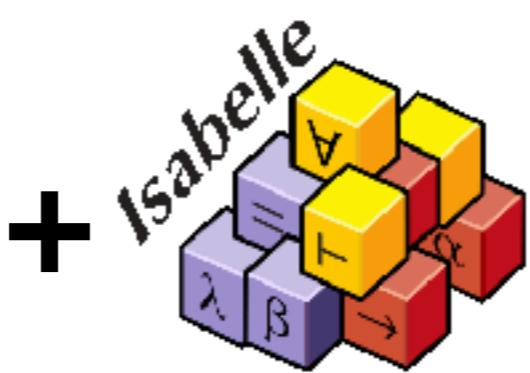
Why proof by induction?

developer



Gewin Klein et. al

proof assistant / ITP



Isabelle/HOL

implementation



verified micro kernel,
seL4

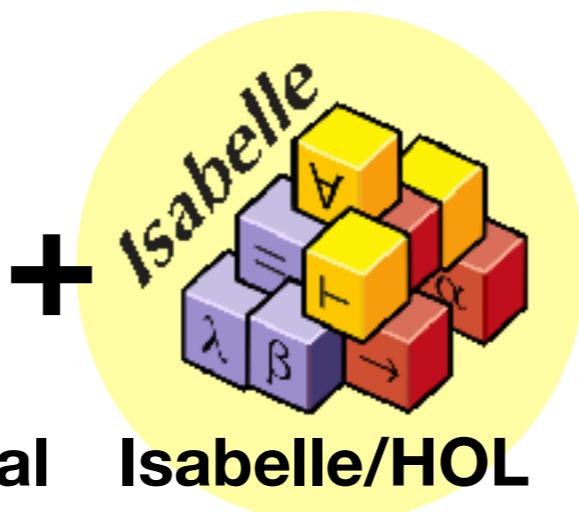
Why proof by induction?

developer



Gewin Klein et. al

proof assistant / ITP



Isabelle/HOL

implementation



verified micro kernel,
seL4

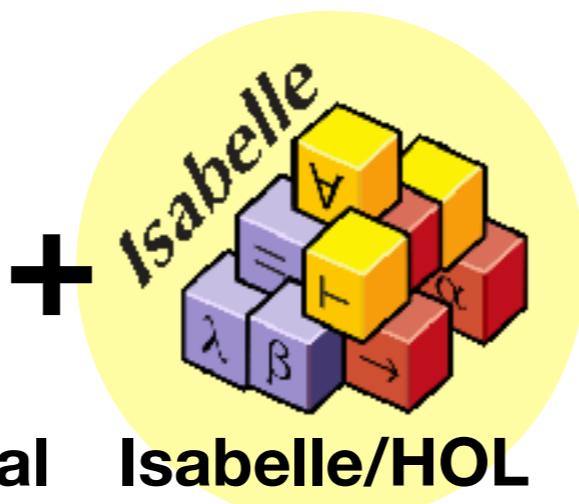
Why proof by induction?

developer



Gewin Klein et. al

proof assistant / ITP



Isabelle/HOL

implementation



verified micro kernel,
seL4



Xavier Leroy



Coq



verified optimizing C compiler,
CompCert

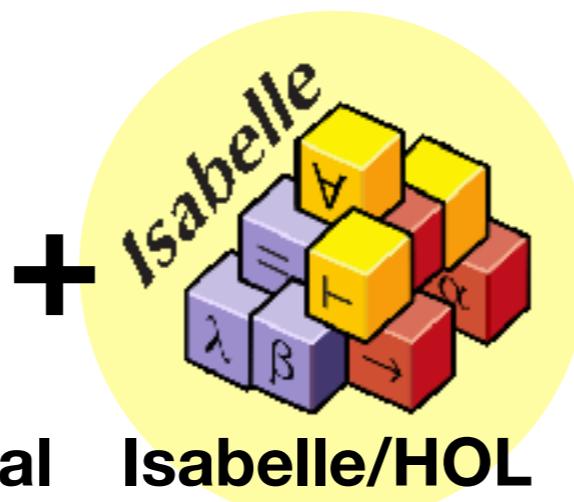
Why proof by induction?

developer



Gewin Klein et. al

proof assistant / ITP



Isabelle/HOL

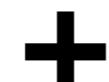
implementation



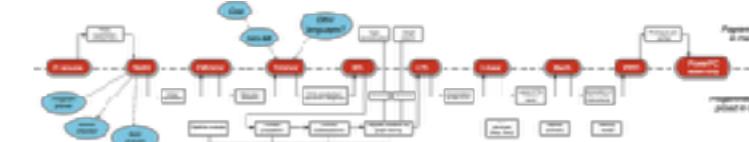
verified micro kernel,
seL4



Xavier Leroy



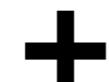
Coq



verified optimizing C compiler,
CompCert



Ramana Kumar et. al



HOL



verified implementation of ML,
CakeML

1



Isabelle's proof methods
(deductive reasoning)

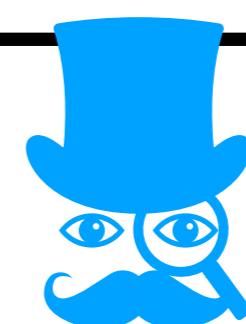
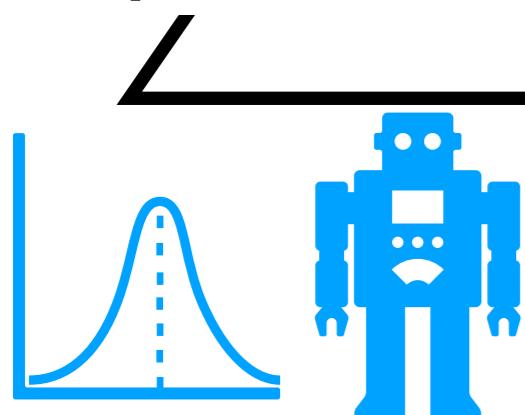
united reasoning
for automatic induction

3

machine learning induction
(inductive reasoning)

2

goal-oriented conjecturing
(abductive reasoning)



1



Isabelle's proof methods (deductive reasoning)

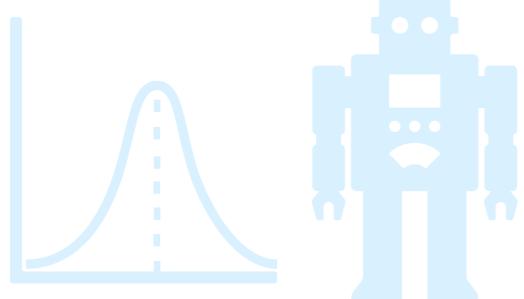
united reasoning
for automatic induction

3

machine learning induction
(inductive reasoning)

2

goal-oriented conjecturing
(abductive reasoning)



DEMO1

The screenshot shows the Isabelle proof assistant interface. The top part 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 is highlighted with a yellow background. The bottom part shows the output of a command:

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

The status bar at the bottom shows the time as 10:42 (221/935) and the file as (isabelle,isabelle,UTF-8-Isabelle). The page number 5 is located in the bottom right corner.

DEMO1

The screenshot shows the Isabelle proof assistant interface. The main window displays a function definition:

```
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)
```

Below the definition, a value is being evaluated:

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

The output of the evaluation is shown in the bottom-left pane:

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

The interface includes a toolbar at the top, a vertical file browser on the left, and a sidebar on the right labeled "Sidekick State Theories". The status bar at the bottom shows the current theory and memory usage.

DEMO1

The screenshot shows the Isabelle proof assistant interface. The main window displays a theory file named `Example.thy`. The code includes a function `sep` and a value declaration, followed by 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 highlighted with a yellow background. The proof state is shown below:

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

The bottom status bar shows the following information:

- Output Query Sledgehammer Symbols
- 17.44 (304/1012) Input/Output complete
- (isabelle.isabelle,UTF-8-Isabelle) nmr@U... 241/535MB 1:17 AM

DEMO1

The screenshot shows the Isabelle proof assistant interface with a blue box highlighting the title "DEMO1". The main window displays a theory file named "Example.thy". The code defines a function "sep" and a value "sep (1::int) [0,0,0]", followed by a strategy definition and a lemma. The "Lemma" section is currently selected.

```
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
11 Lemma
12   "map f (sep x xs) = sep (f x) (map f xs)"
13
14
15
16
17
18
19
20
21
```

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

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)"

Proof state Auto update Update Search: 100% 0

Output Query Sledgehammer Symbols

14.58 (310/1067) (isabelle.isabelle,UTF-8-Isabelle) nmr o U.. 291/535MB 1:20 AM

DEMO1

The screenshot shows a software interface for formal verification, likely PSL, with a toolbar at the top and a sidebar on the right labeled "File Browser Documentation", "Sidekick", "State", and "Theories". The main area displays a code editor for a file named "Example.thy".

```
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)"
find_proof DInd
```

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

DEMO1

The screenshot shows the Isabelle/Isar proof assistant interface. The main window displays a theory file named "Example.thy". The code includes a function definition for "sep", a value declaration for "sep (1::int)", a strategy setting for "DInd", a lemma statement, and a partially completed proof command. The proof command "find_proof DInd" is highlighted with a red rectangle. The interface includes a toolbar at the top, a vertical scroll bar on the right, and various status indicators at the bottom.

```
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)"
  find_proof DInd
```

Number of lines of commands: 3

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

DEMO1

The screenshot shows the Isabelle proof assistant interface with a theory file named `Example.thy`. The code defines a function `sep` and a lemma `map f (sep x xs) = sep (f x) (map f xs)`. The proof for the lemma uses induction on `xs` and automation with `auto`.

```
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
11 lemma
12   "map f (sep x xs) = sep (f x) (map f xs)"
13   find_proof DInd
14 apply (induct xs rule: Example.sep.induct)
15 apply auto
16 done

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

Proof state: Auto update: Update: 100%

Output Query Sledgehammer Symbols
20.11 (433/1147) (isabelle.isabelle,UTF-8-Isabelle) mmr@U... 255/535MB 1:05 AM

DEMO1

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

1 fun sep :: "'a :: type list ⇒ 'a :: type 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
11 lemma
12   "map f (sep x xs) = sep (f x) (map f xs)"
13   find_proof DInd
14   apply (induct xs rule: Example.sep.induct)
15   apply auto
16   done

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

What happened?

DEMO1

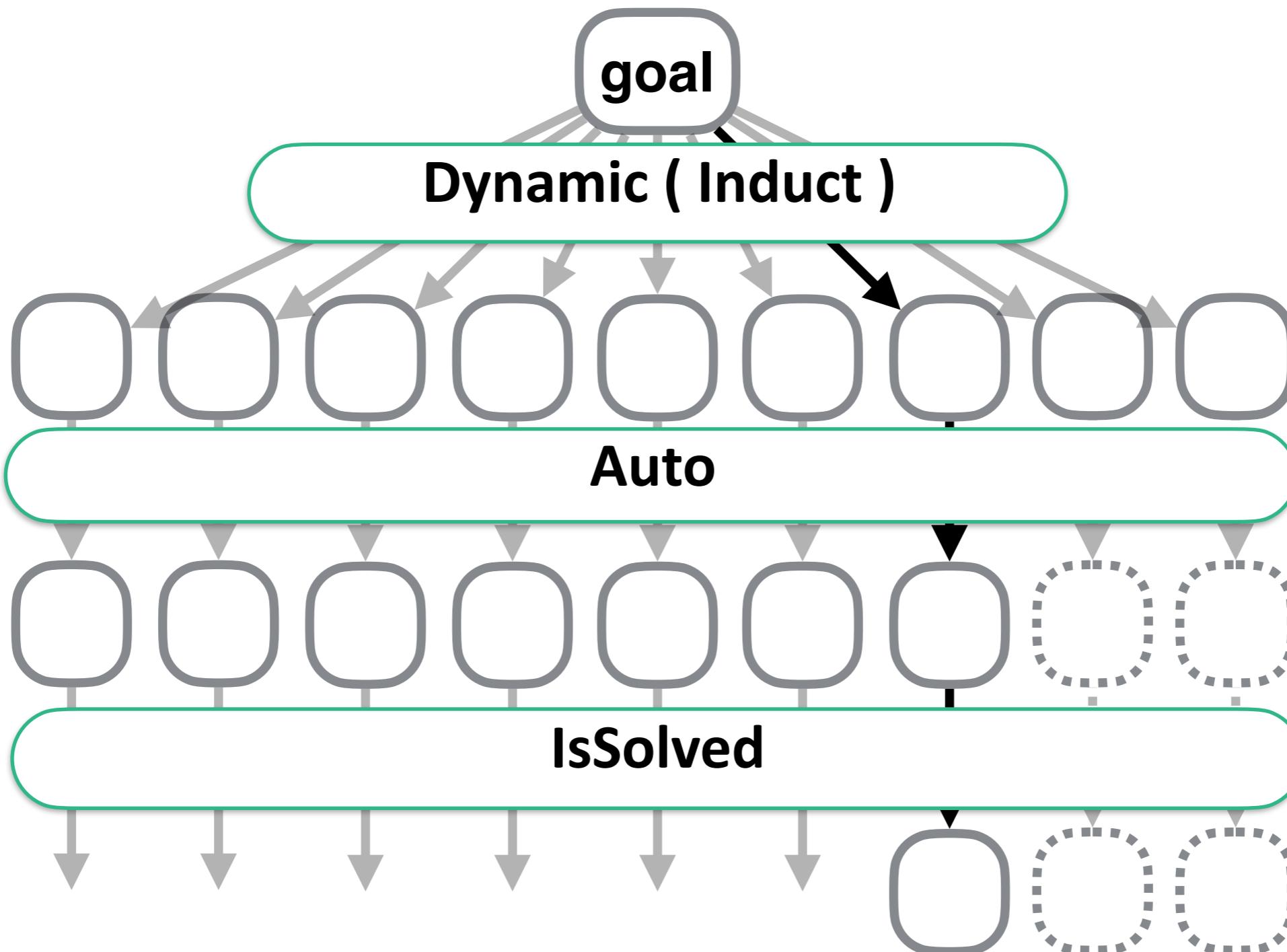
PSL: Proof Strategy Language

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

DEMO1

PSL: Proof Strategy Language

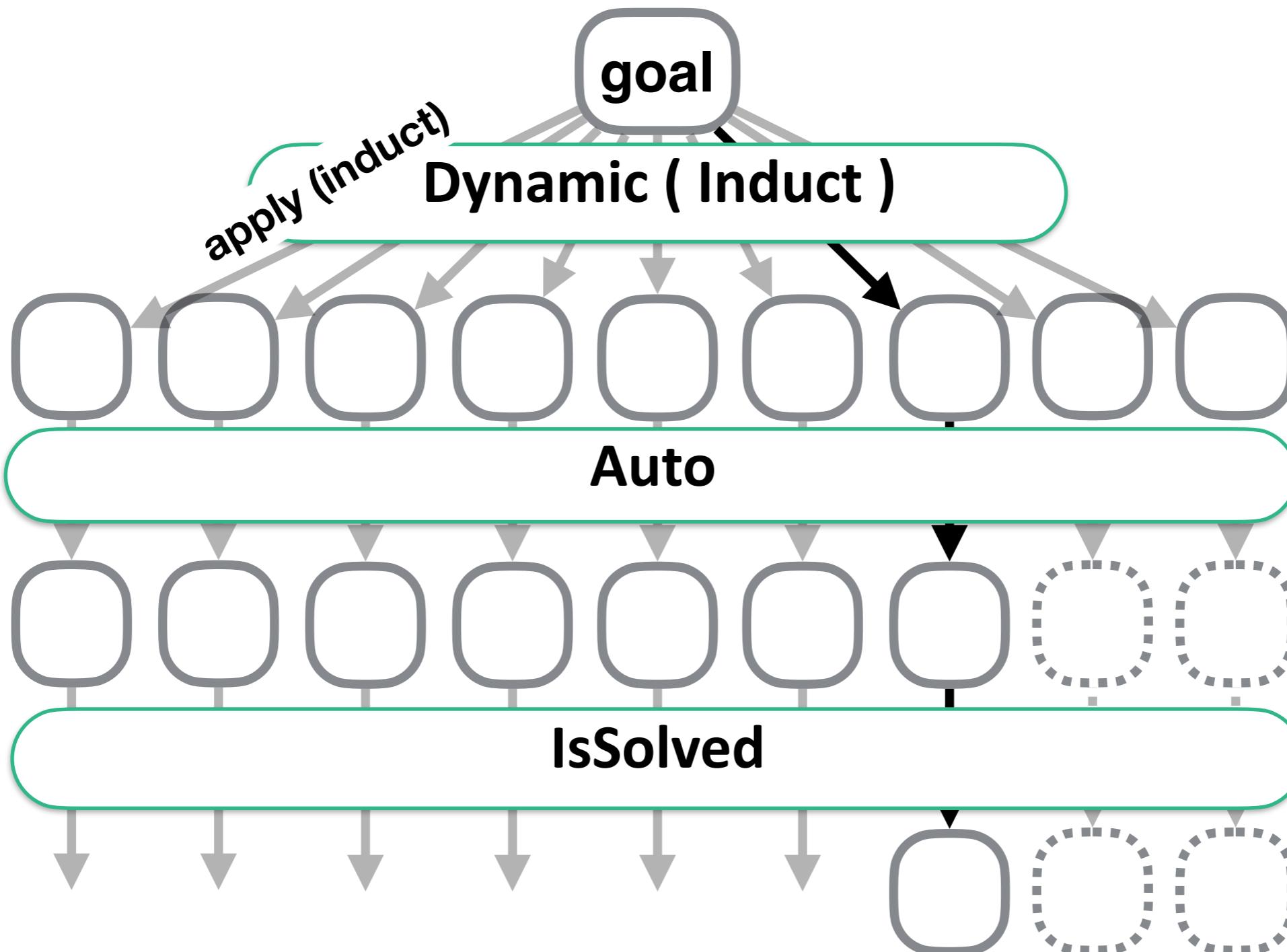
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

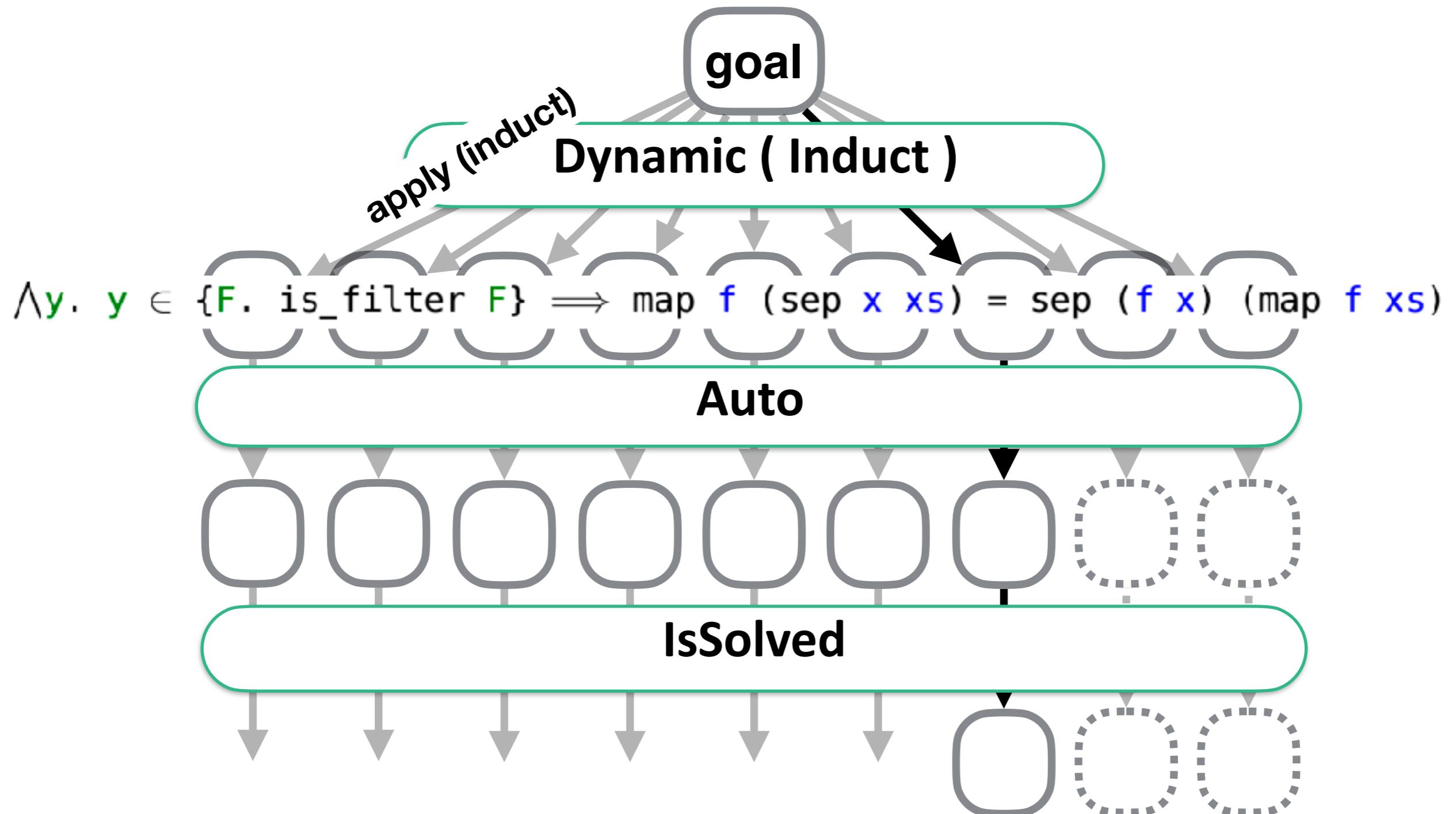
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

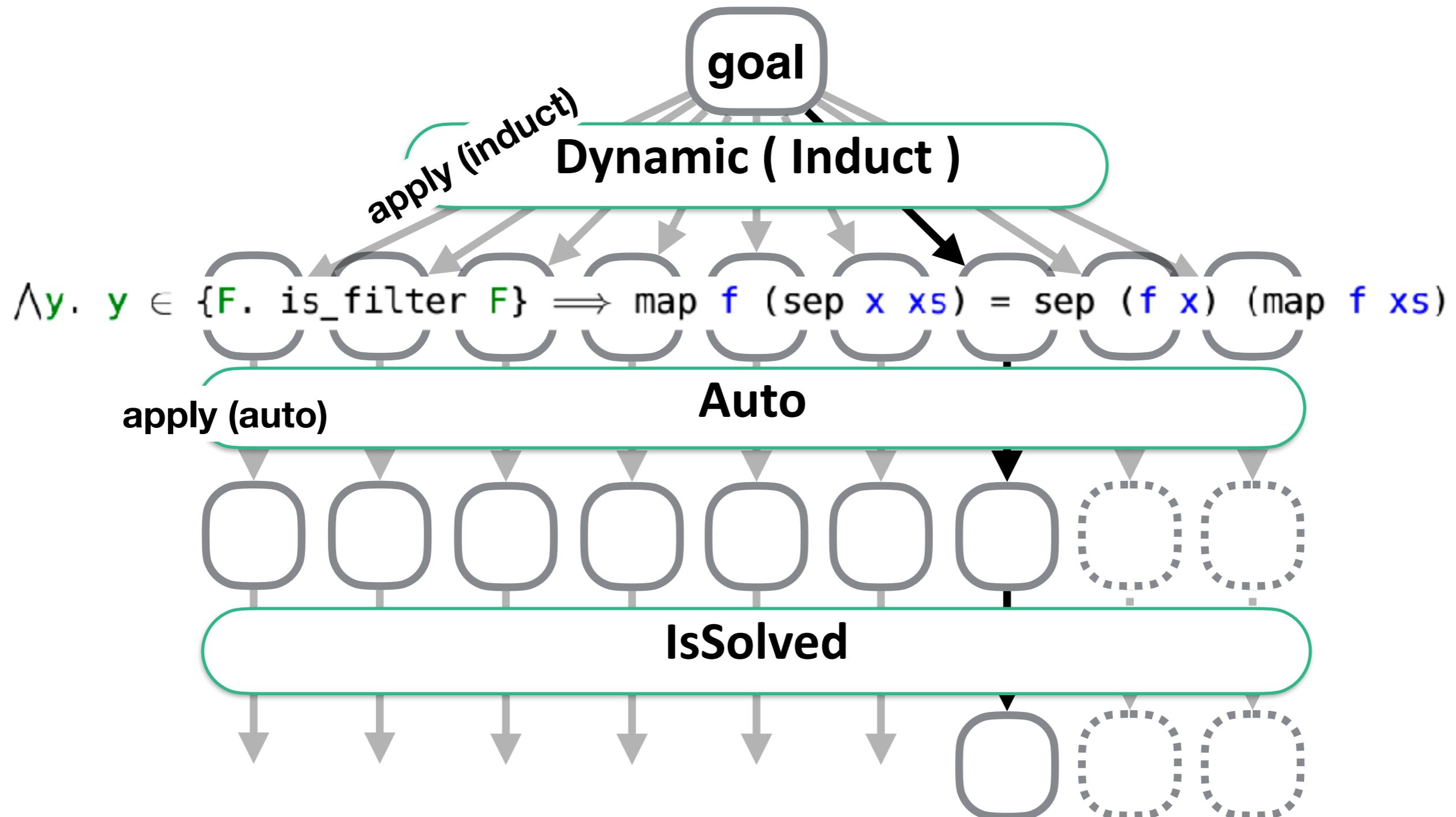
PSL: Proof Strategy Language

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



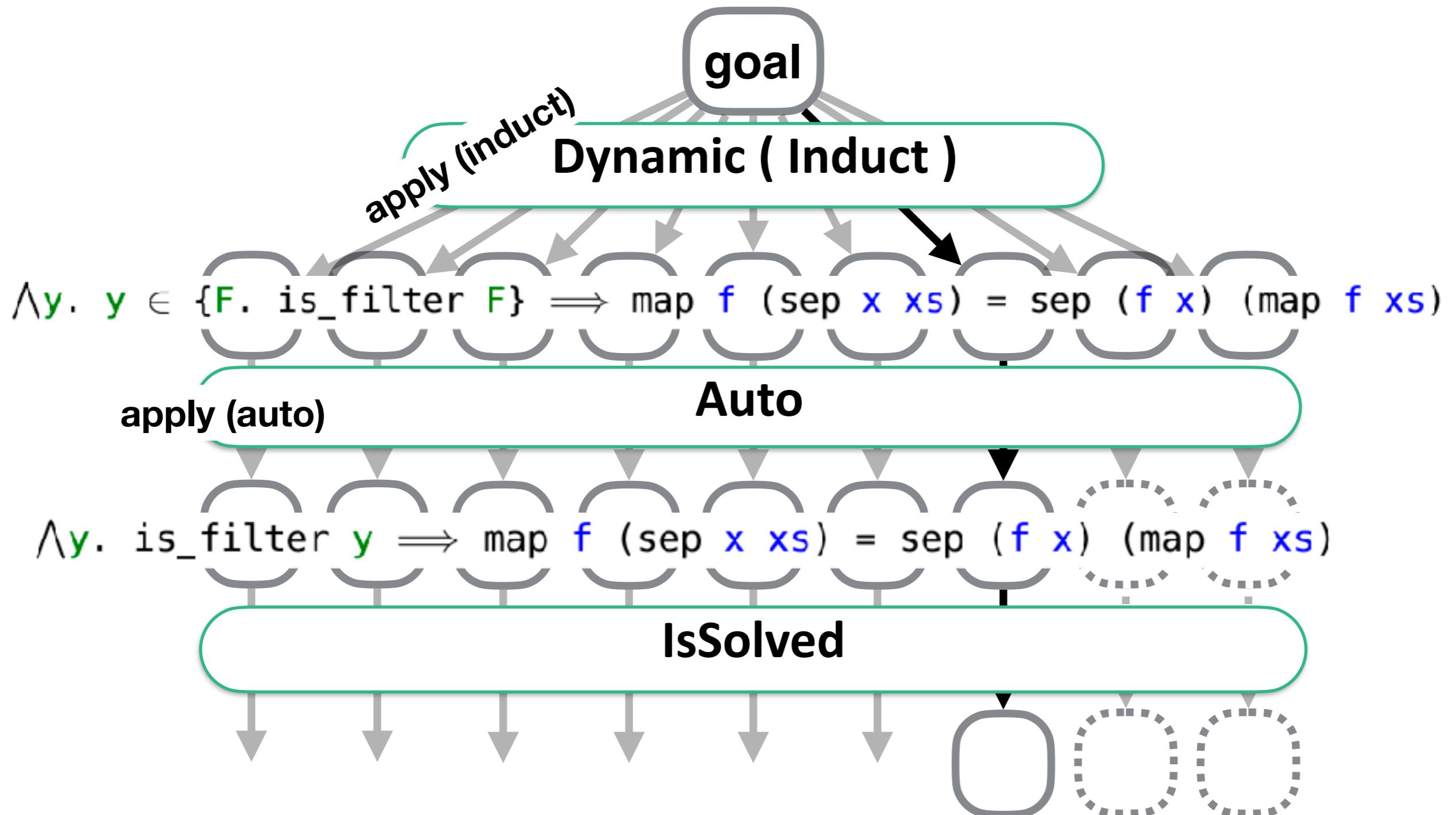
DEMO1 PSL: Proof Strategy Language

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



DEMO1 PSL: Proof Strategy Language

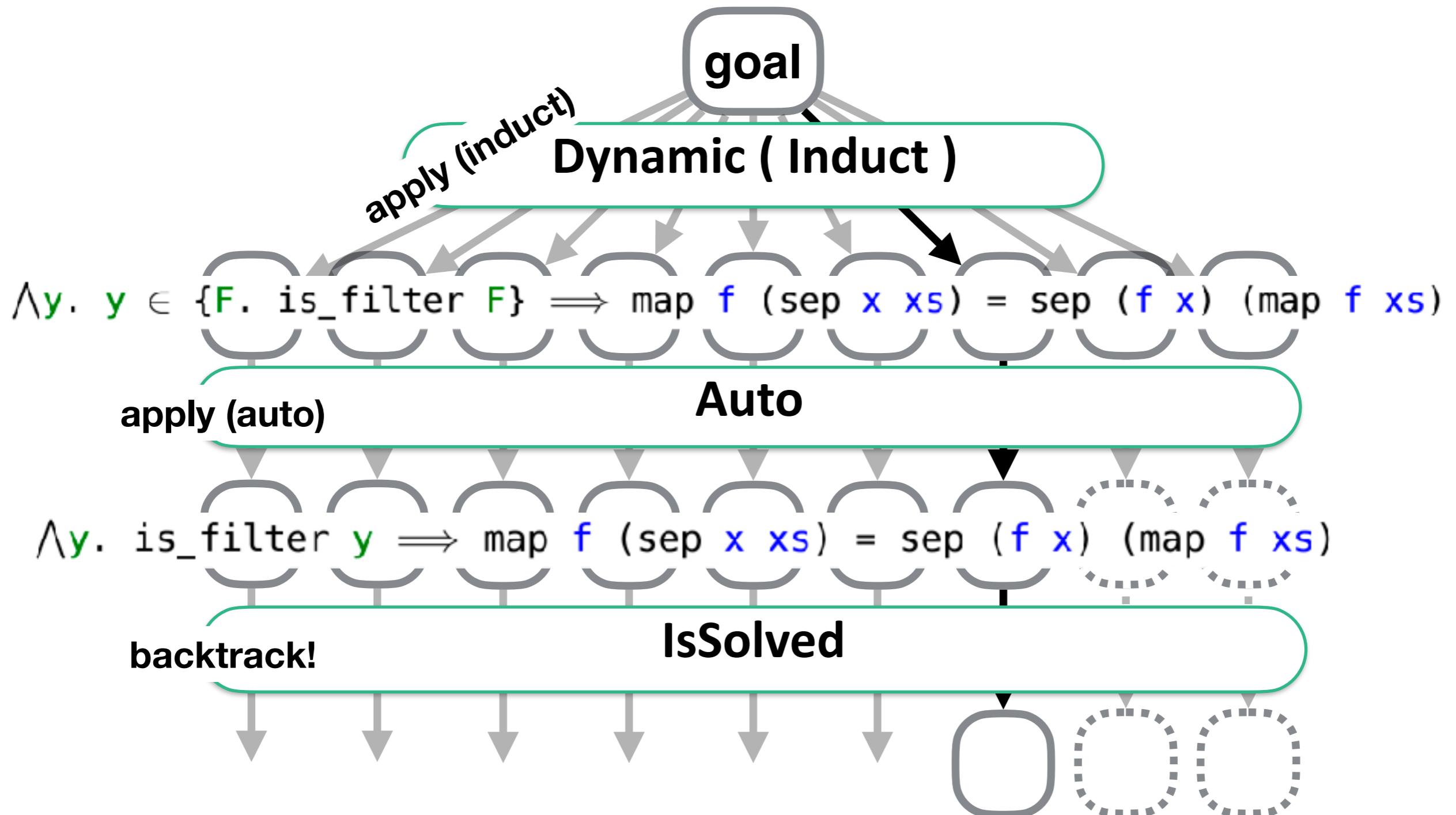
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

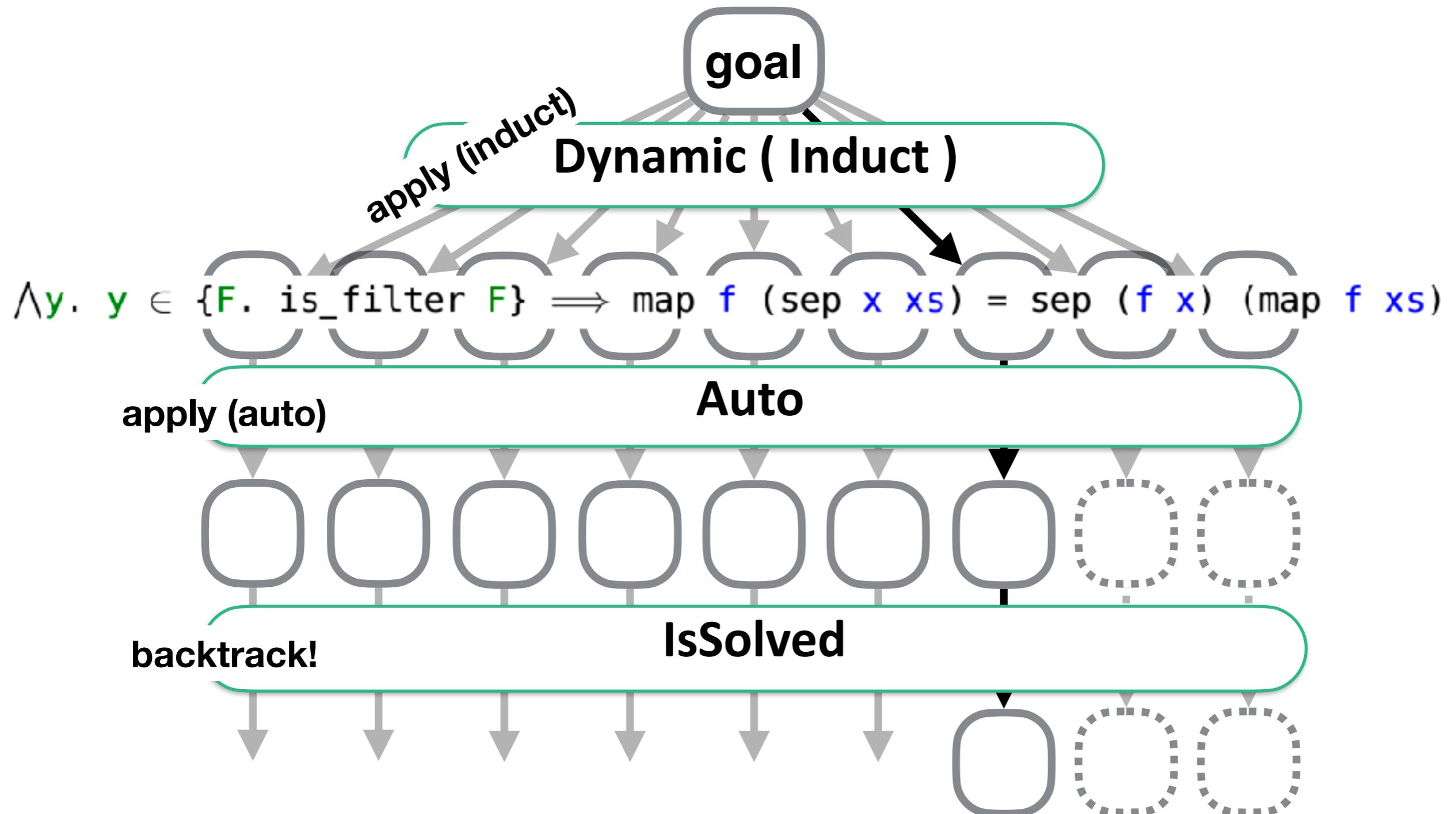
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

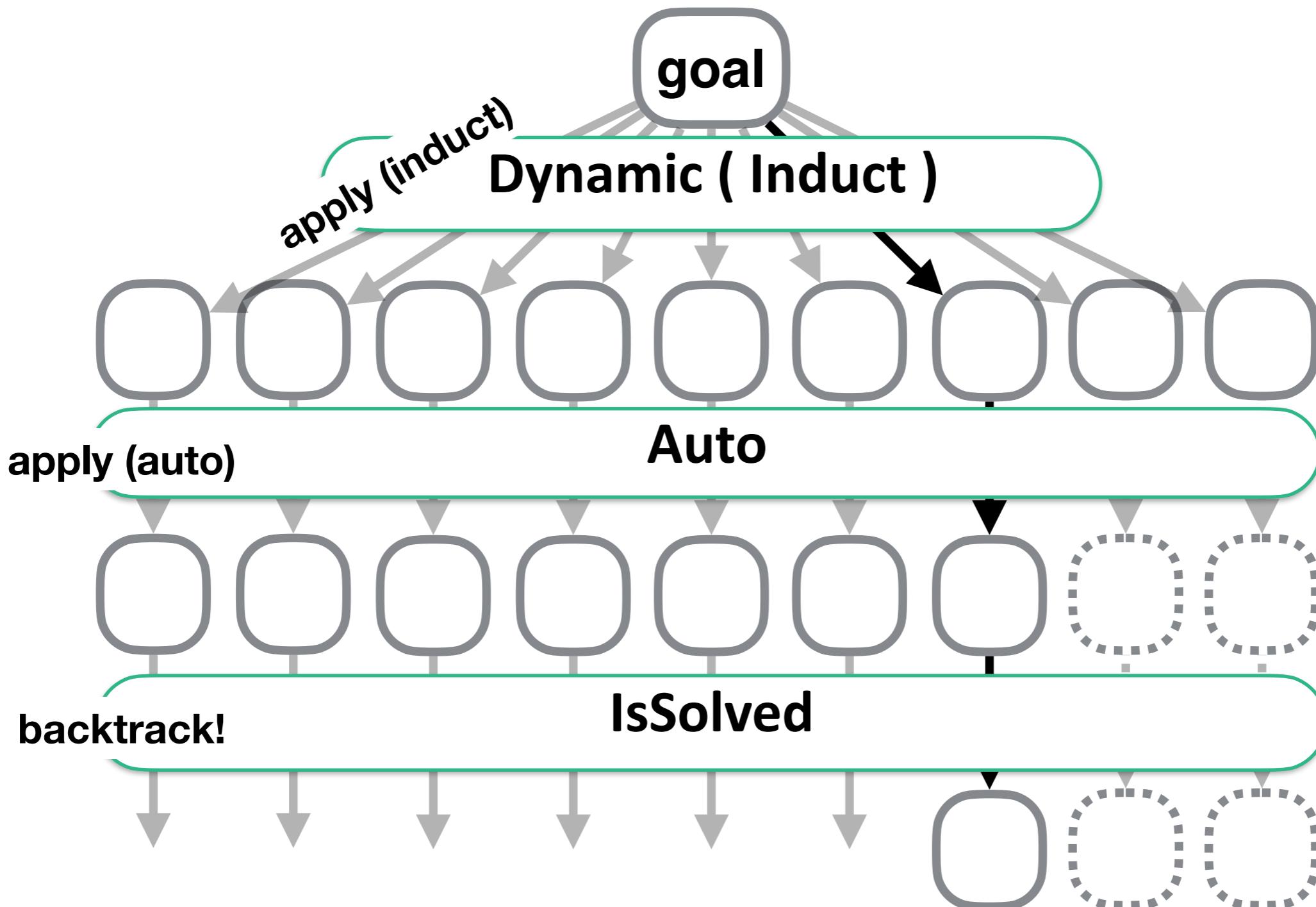
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

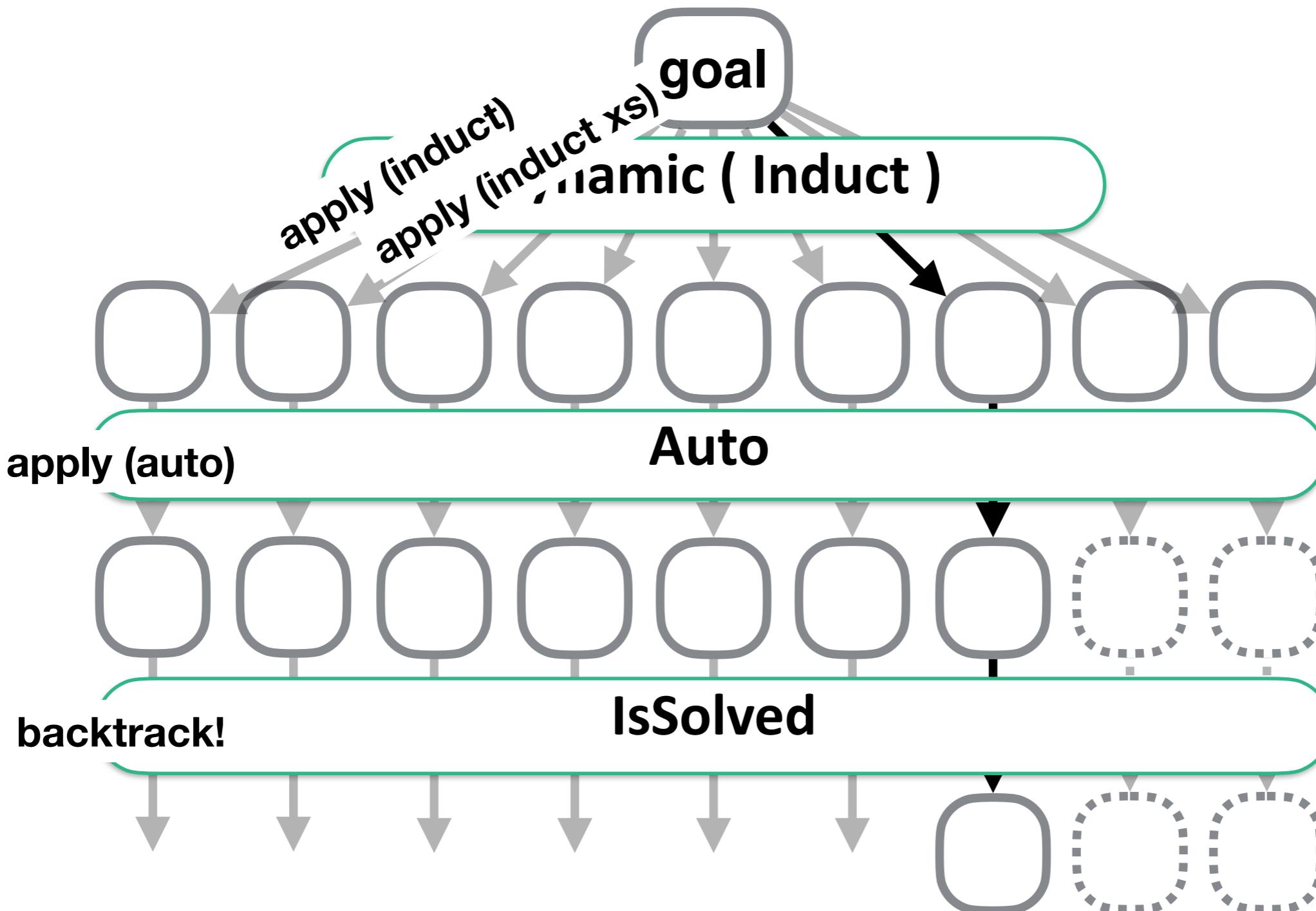
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

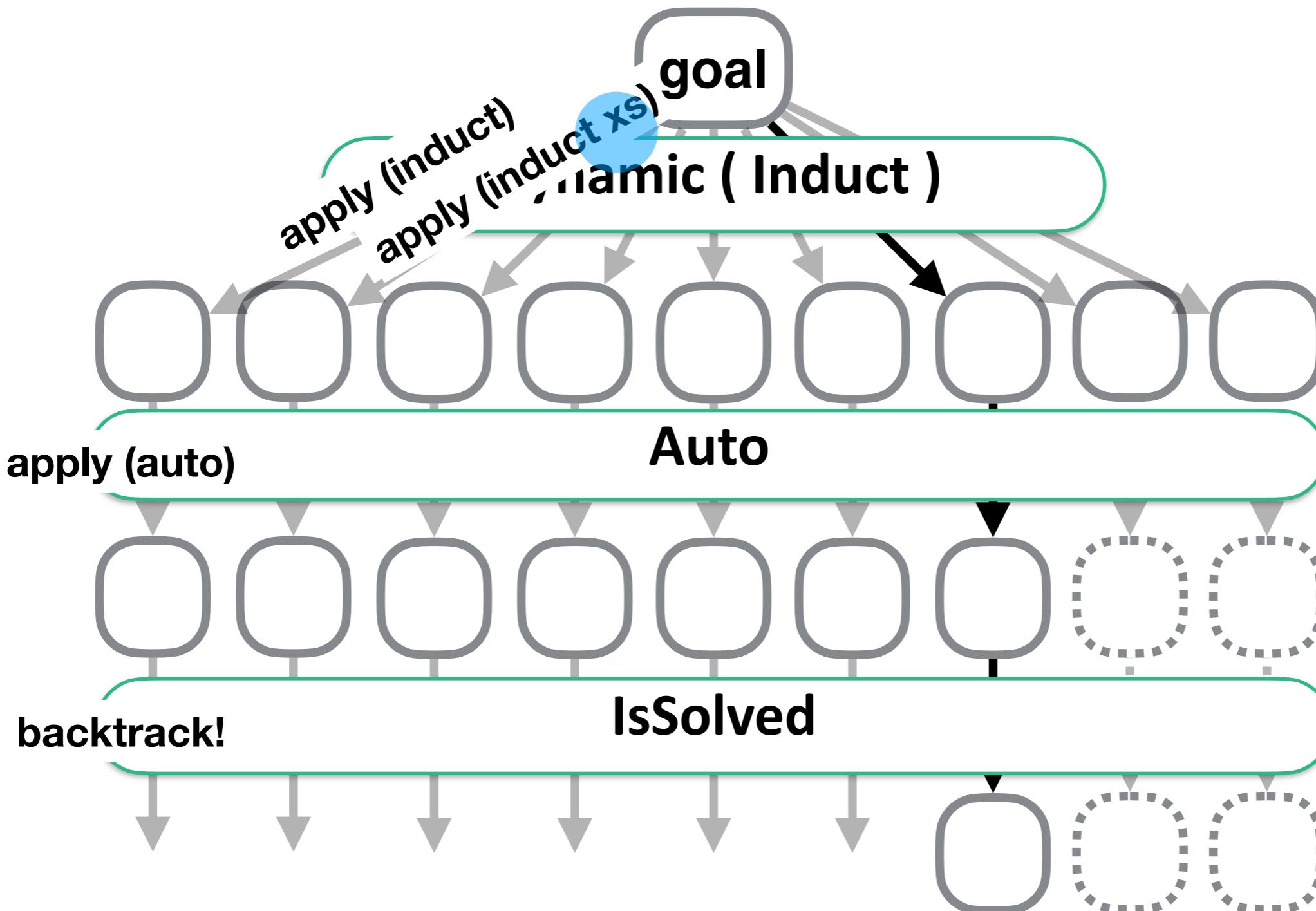
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

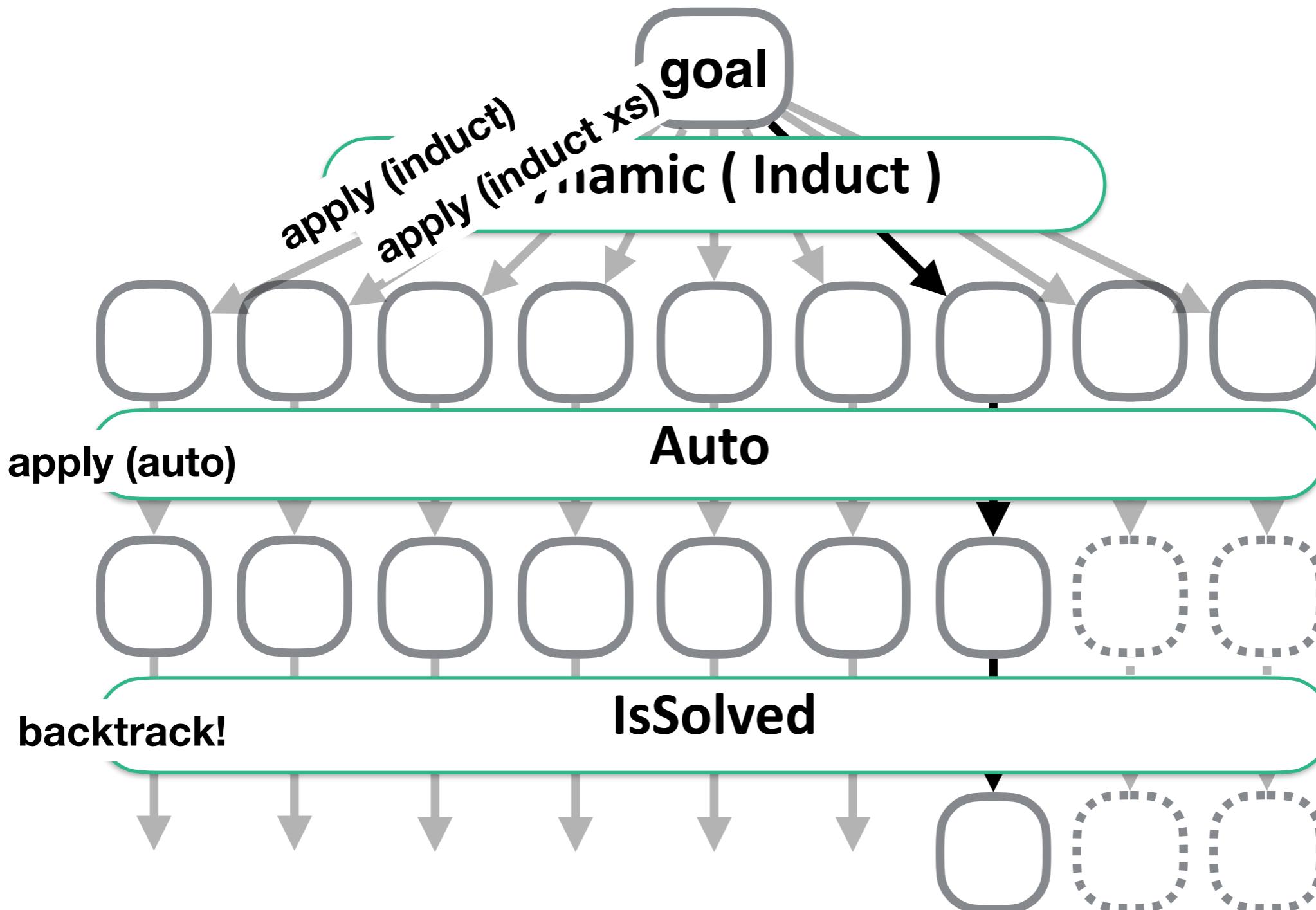
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

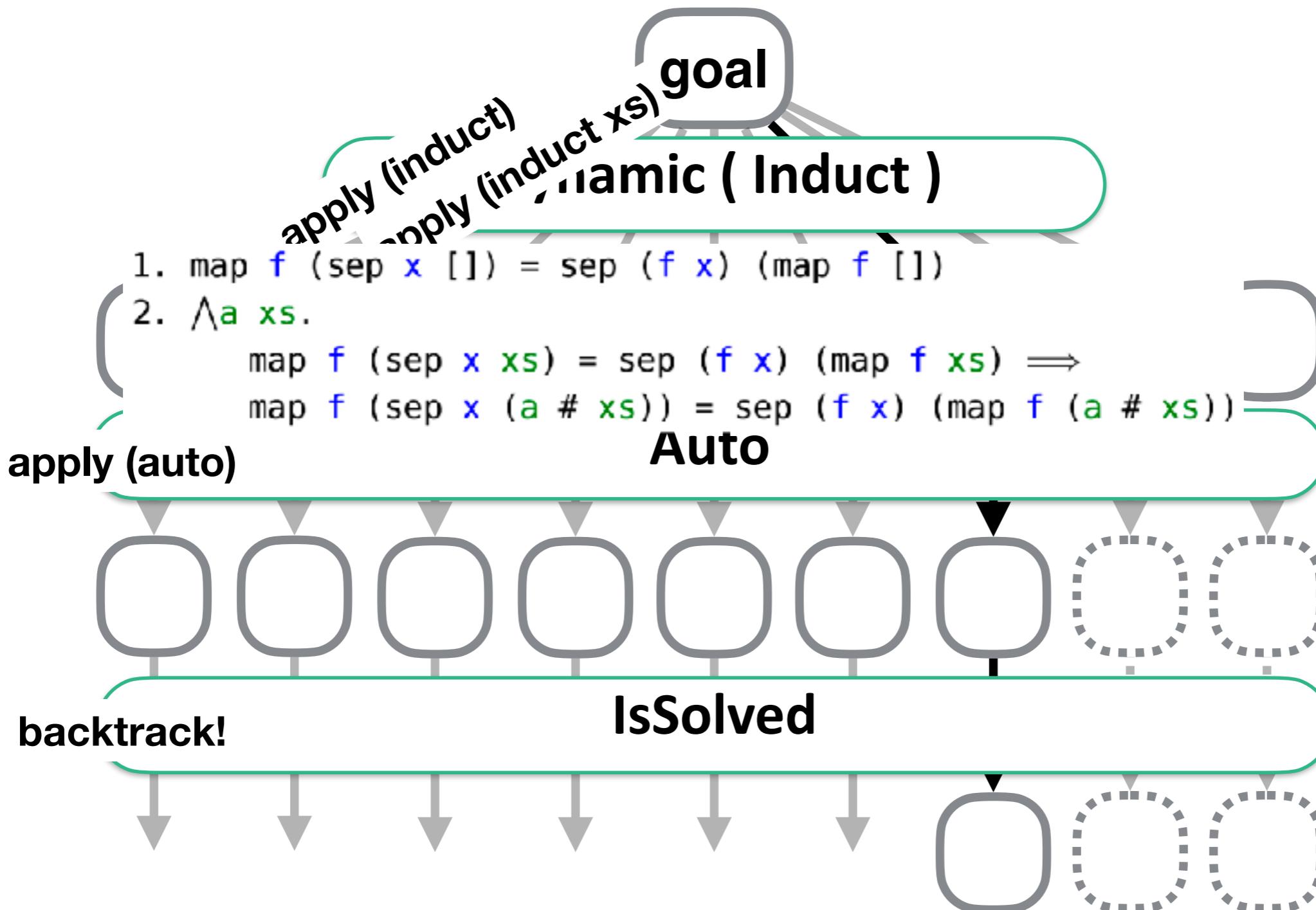
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

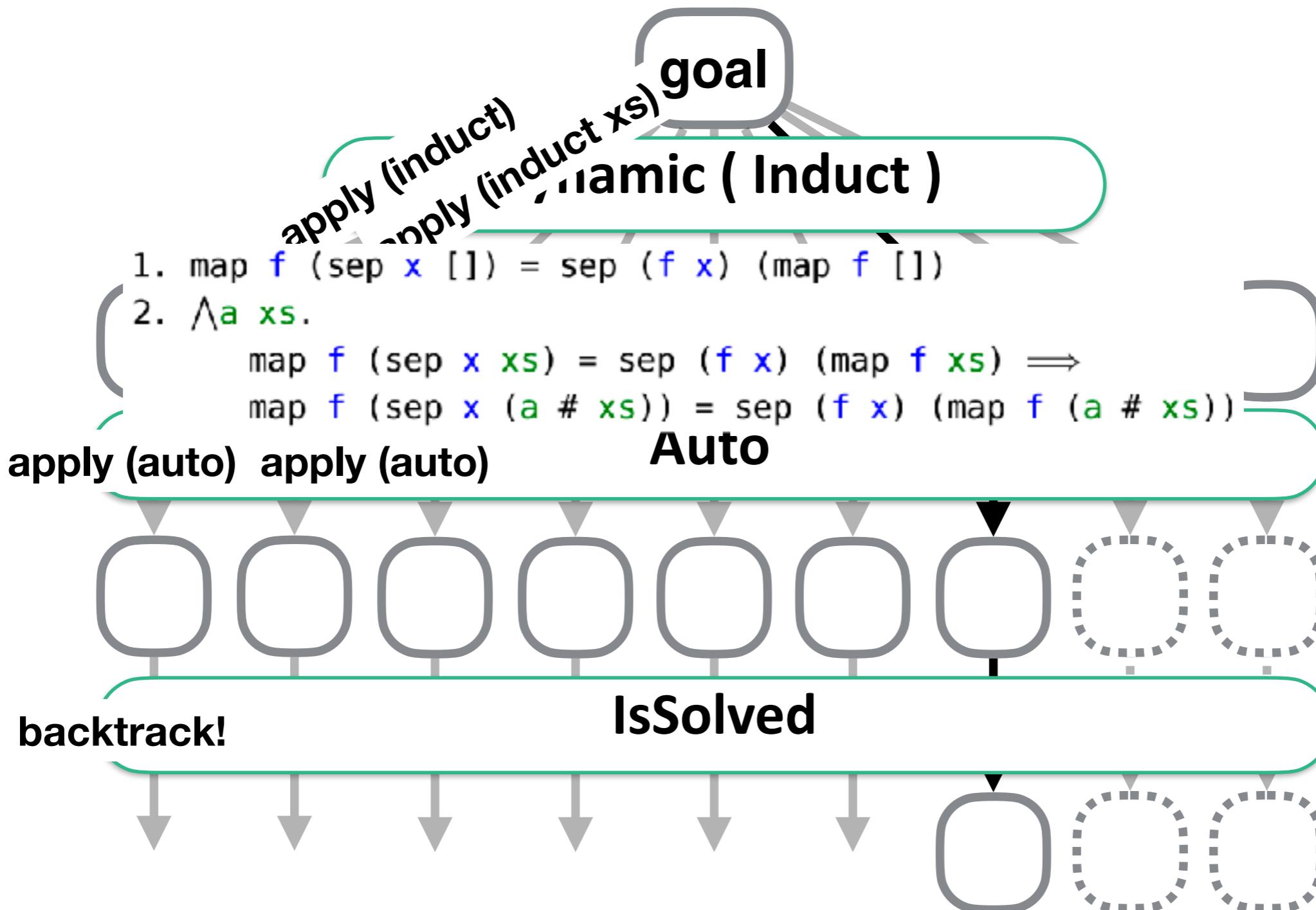
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

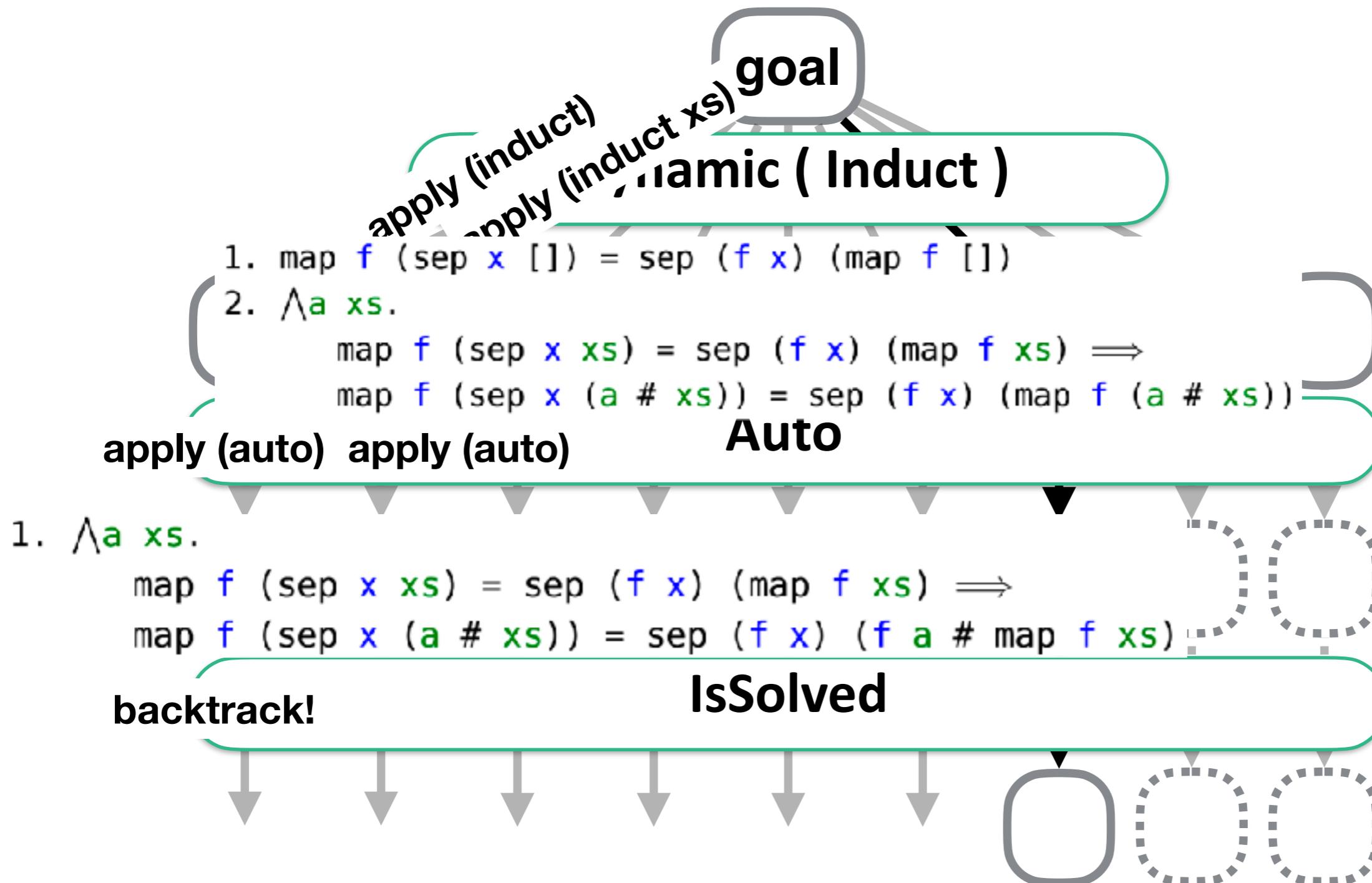
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

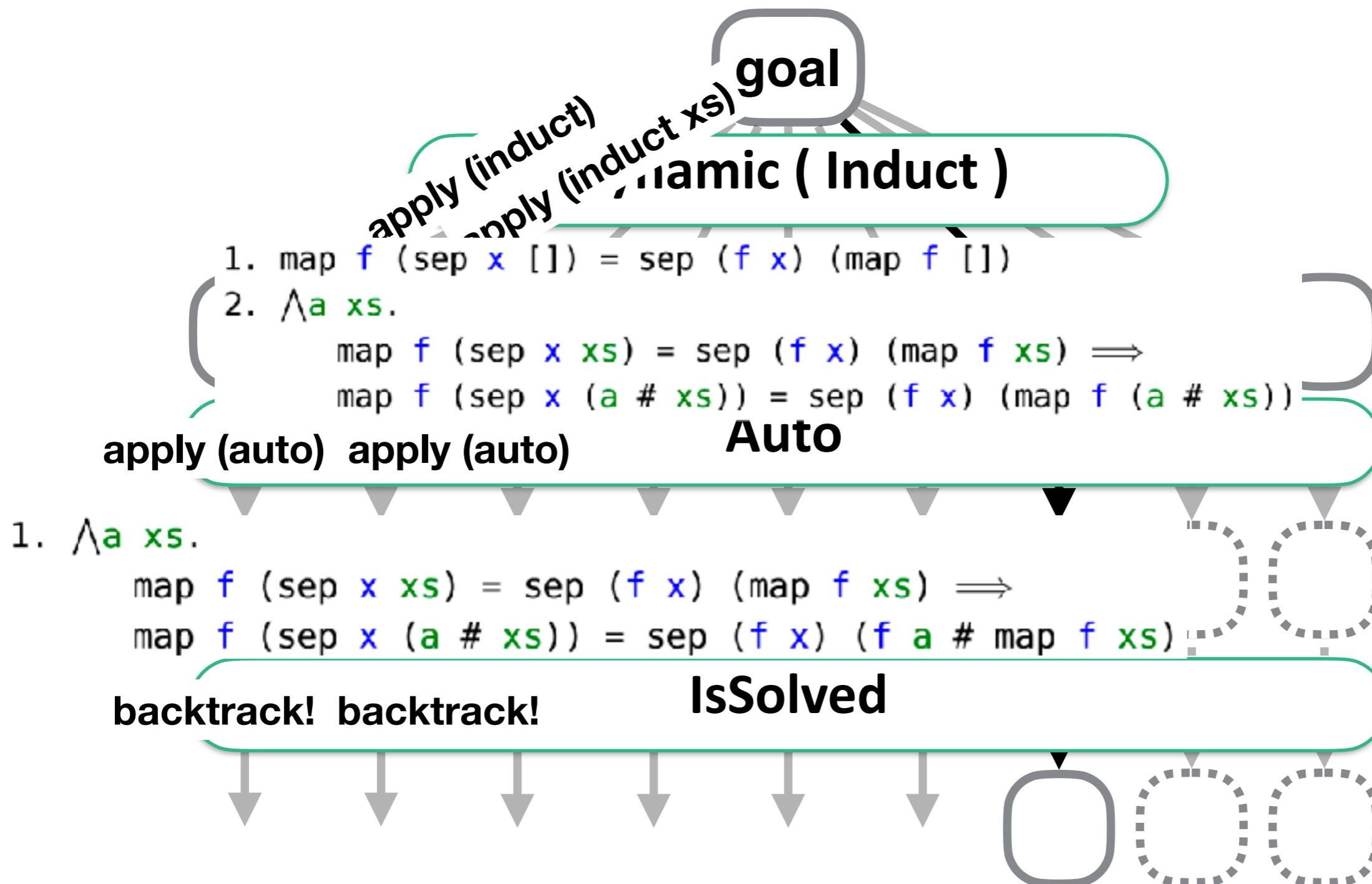
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

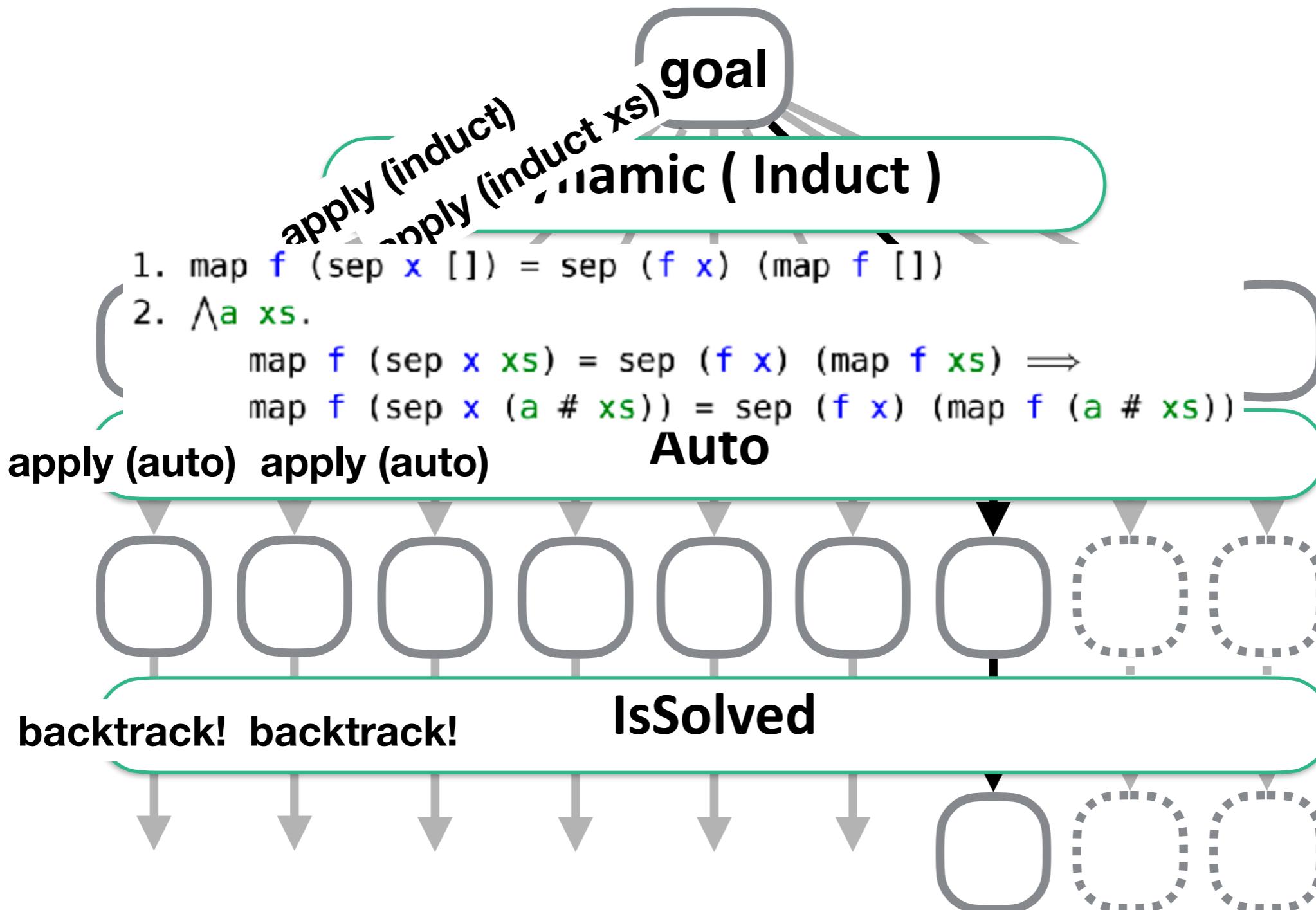
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

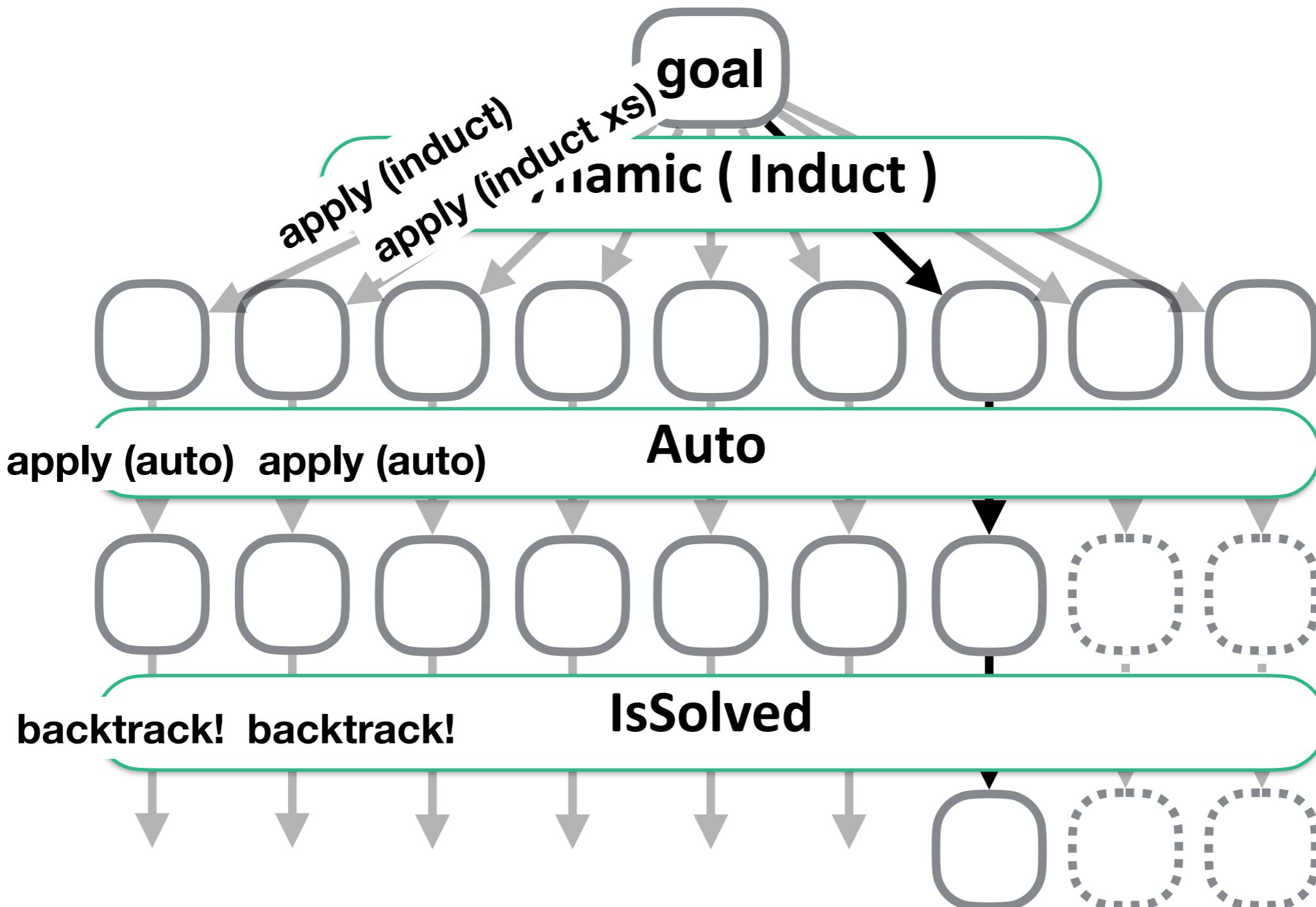
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

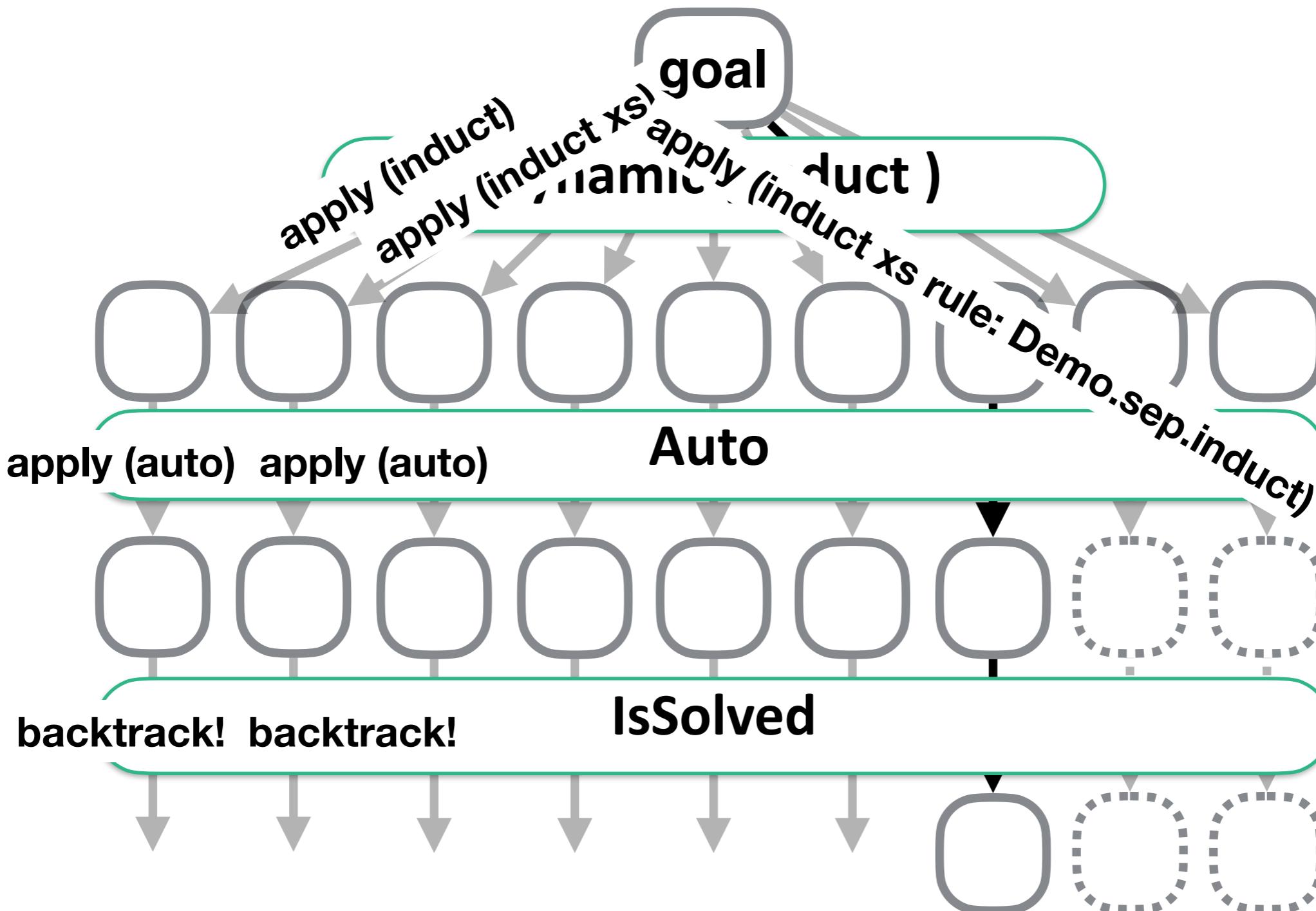
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

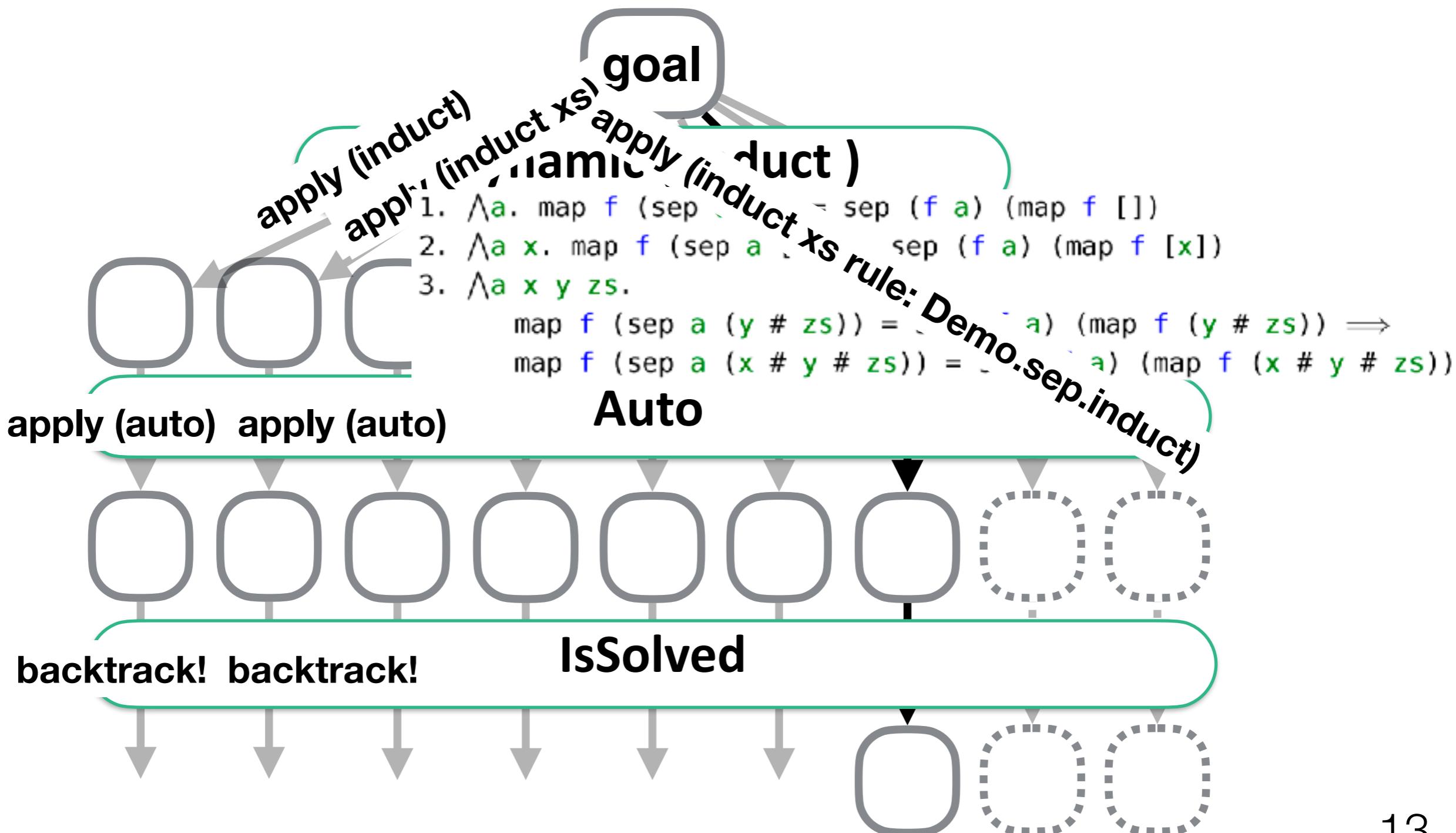
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

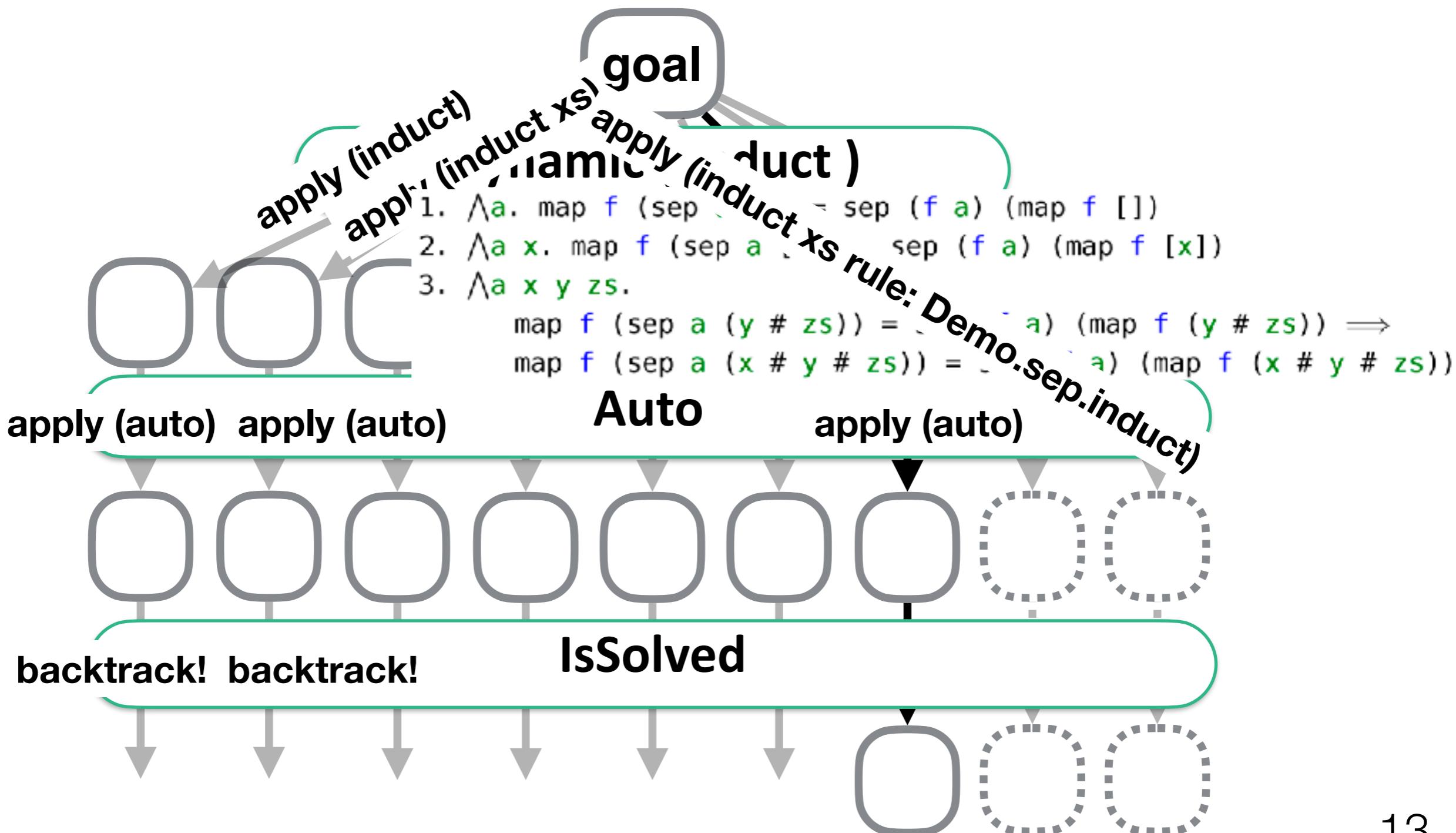
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

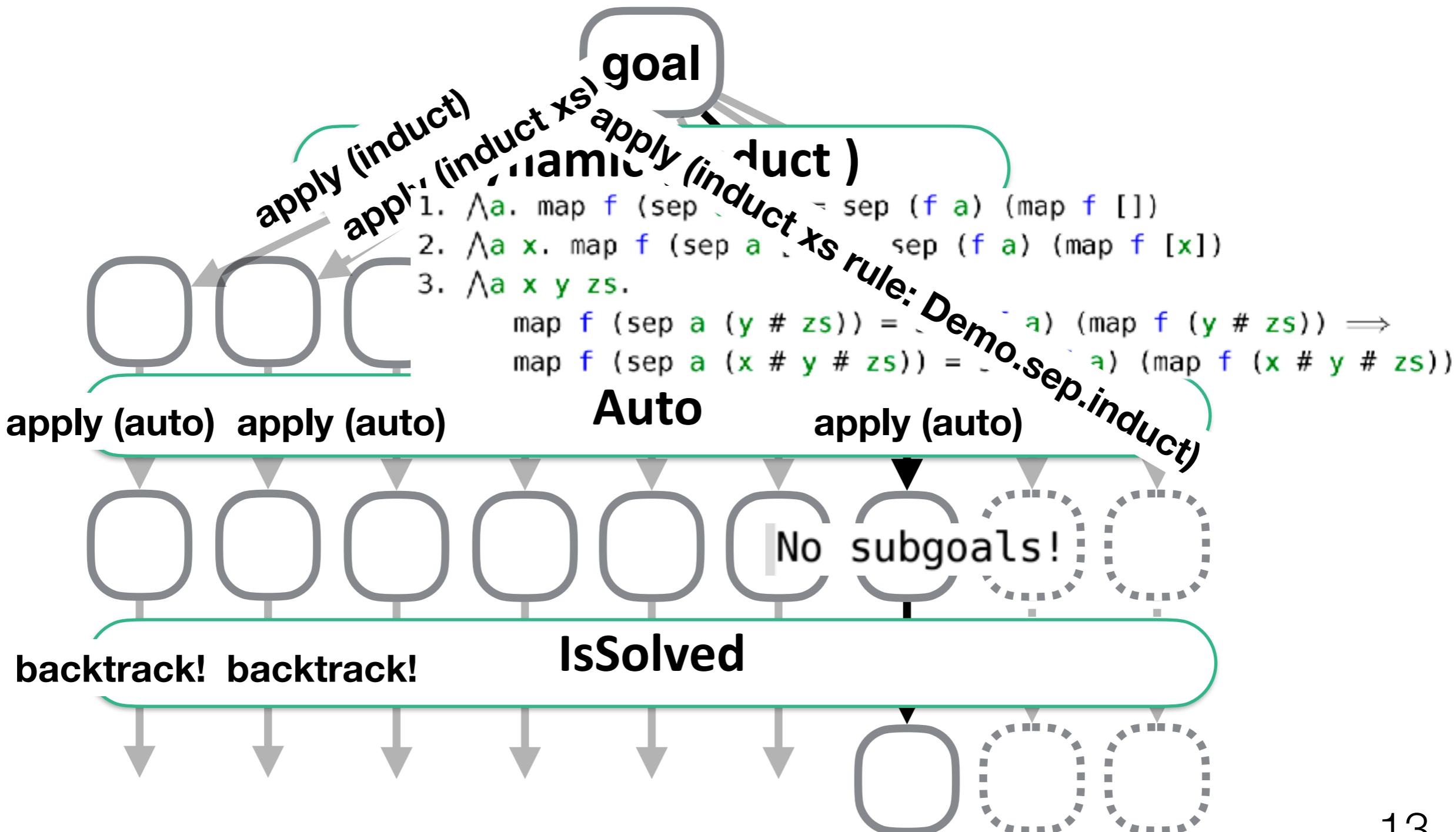
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

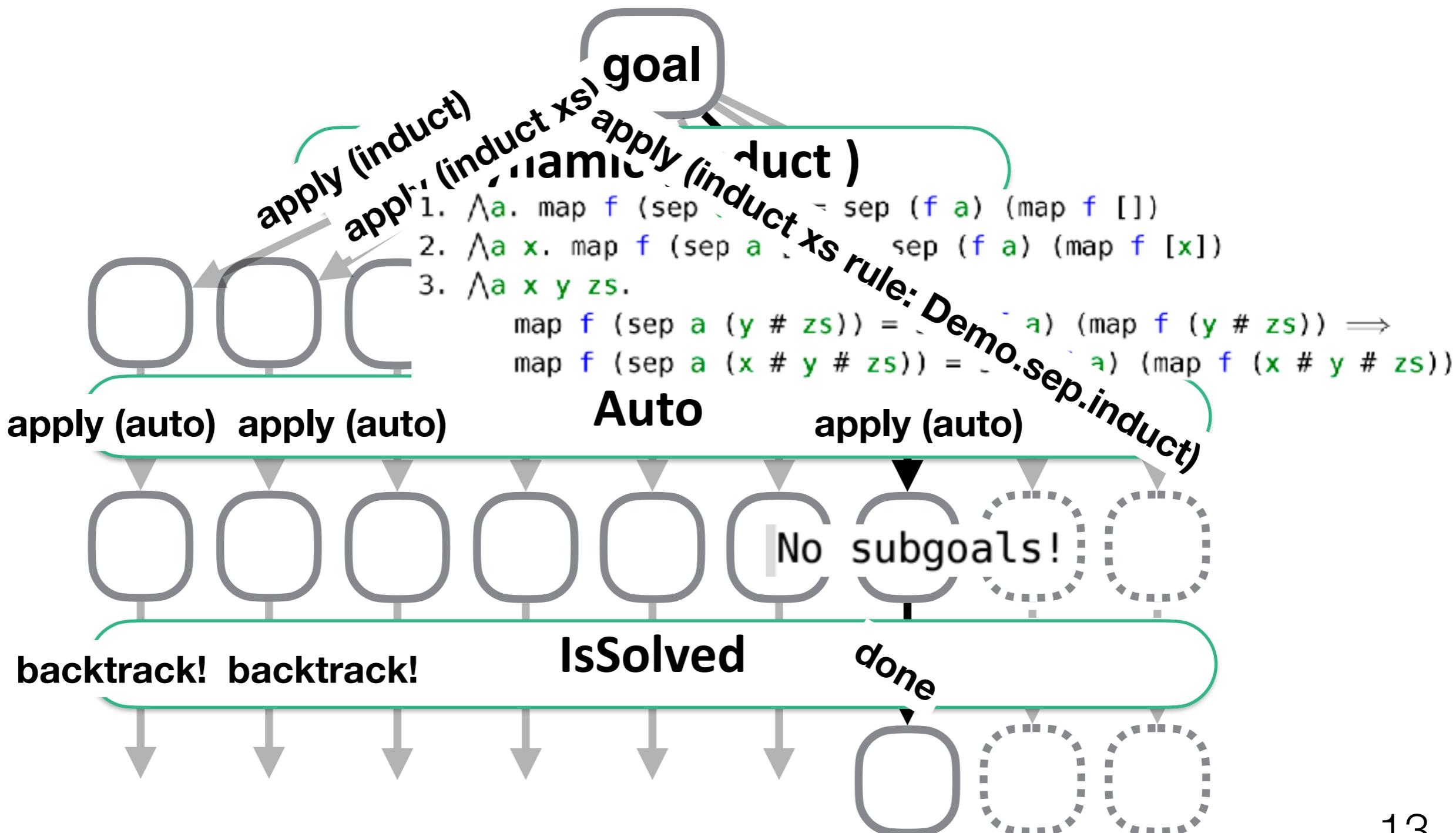
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

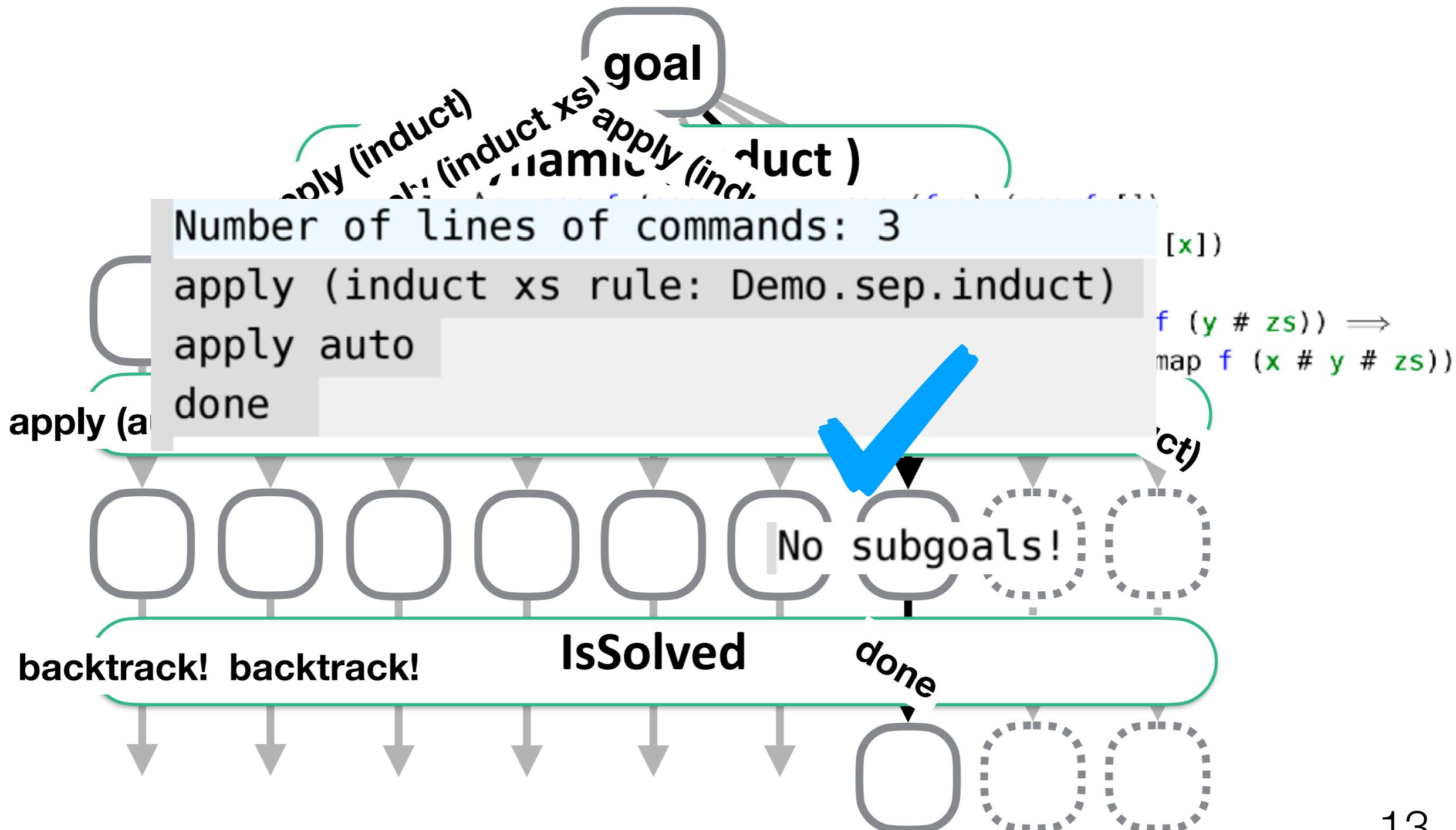
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

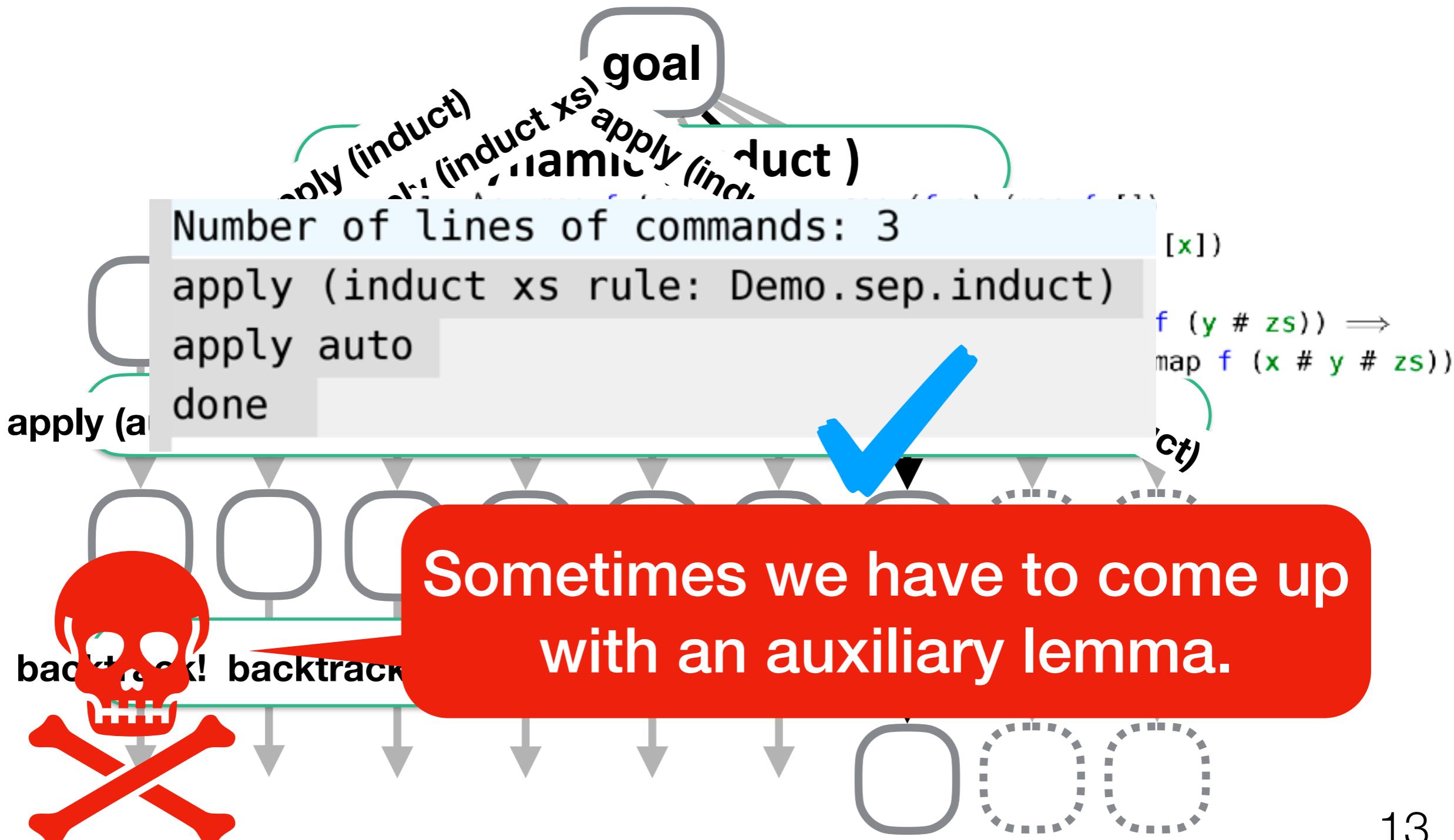
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

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



1



Isabelle's proof methods
(deductive reasoning)

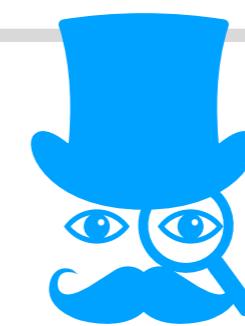
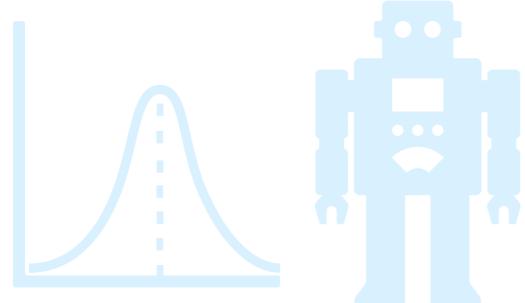
united reasoning
for automatic induction

3

machine learning induction
(inductive reasoning)

2

goal-oriented conjecturing
(abductive reasoning)



DEMO2

The screenshot shows the Isabelle/Isar proof assistant interface. The main window displays a theory file named "Example.thy". The code includes definitions for list reversal (primrec rev) and its inverse (primrec itrev), and a strategy for proofs. A lemma is being defined, and the proof command has been issued.

```
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 Lemma "itrev xs [] = rev xs"
40   find_proof DInd
41
42
43
44
45

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

Output Query Sledgehammer Symbols
43.29 (837/1004) (isabelle,isabelle,UTF-8-isabelle)| nmrc U.. 270/535MB 7:01 PM
```

DEMO2

The screenshot shows the Isabelle/HOL proof assistant interface. The main window displays a theory file named "Example.thy". The code includes two primitive recursive definitions 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 is a strategy declaration:

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

Following this is a lemma:

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

The status bar at the bottom indicates the proof state is empty: "empty sequence. no proof found."

Toolbar icons include: File, Edit, Insert, Find, Replace, Cut, Copy, Paste, Undo, Redo, Save, Load, New, Open, Close, Help, and Proof.

Left sidebar: File Browser, Documentation, Sidekick, State, Theories.

Bottom toolbar: Proof state (checkbox), Auto update (checkbox), Update, Search, zoom (100%), and a progress bar.

Bottom status bar: Output, Query, Sledgehammer, Symbols; 44.18 (855/1004); (isabelle, isabelle, UTF-8-isabelle) in memory U.. 283/535MB 6:17 PM

DEMO2

The screenshot shows the PGT (PVS/GT) interface with the following code:

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

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

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 a list of elements, and returns the list of elements followed by the reverse of the original list. It uses the `rev` function as a helper. Below the definitions are three strategies: `DInd`, `CDInd`, and `DInd_Or_CDInd`. The `DInd` strategy uses `Thens` with `Dynamic (Induct)`, `Auto`, and `IsSolved`. The `CDInd` strategy uses `Thens` with `Conjecture`, `Fastforce`, `Quickcheck`, and `DInd`. The `DInd_Or_CDInd` strategy uses `Ors` with `DInd` and `CDInd`. A lemma is also defined: `"itrev xs [] = rev xs"`, with proof obligations for both `DInd` and `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

The screenshot shows the PGT (PVS/GT) interface with the following code:

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

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

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 a list of elements, and returns the list of elements with the current element added at the front. It uses the `rev` function to build the result. The code also defines three strategies: `DInd`, `CDInd`, and `DInd_Or_CDInd`. The `DInd` strategy uses `Thens` to apply `Dynamic (Induct)`, `Auto`, and `IsSolved`. The `CDInd` strategy uses `Thens` to apply `Conjecture`, `Fastforce`, `Quickcheck`, and `DInd`. The `DInd_Or_CDInd` strategy uses `Ors` to choose between `DInd` and `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

The screenshot shows the Isabelle proof assistant interface. The main window displays a theory file named "Example.thy". The code defines two primitive recursive functions: `rev` and `itrev`, and sets up 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.

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

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

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

Output Query Sledgehammer Symbols
45.27 (988/1000) (isabelle,isabelle,UTF-8-isabelle)| nmrc U.. 423/535 6:17 PM
```

DEMO2

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

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

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

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

A yellow oval highlights the strategy definitions for `DInd`, `CDInd`, and `DInd_Or_CDInd`. A blue arrow points from the `find_proof` command in the lemma section to the `CDInd` strategy. The bottom part of the interface shows the proof state with a green box around the subgoal tac and a yellow box around the equality to prove.

DEMO2

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

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

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



Can we find out how to apply induction without completing a proof?

Proof state Auto update Update Search:

1



Isabelle's proof methods
(deductive reasoning)

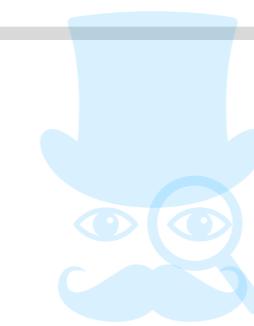
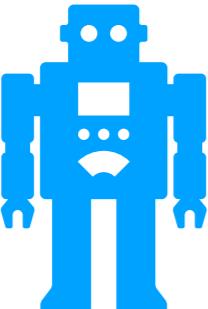
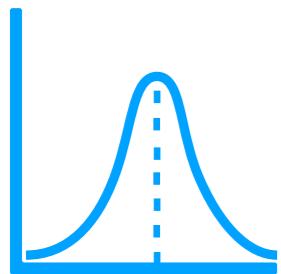
united reasoning
for automatic induction

3

machine learning induction
(inductive reasoning)

2

goal-oriented conjecturing
(abductive reasoning)



DEMO3

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 states a lemma. The lemma is currently being edited, indicated by a yellow background and a red cursor. The text area contains:

```
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"
oops
```

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

DEMO3

```
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   oops
20
21
```

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. *)

DEMO3

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 as follows:

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

The lemma states that `itrev xs ys = rev xs @ ys`. The proof script for this lemma uses the `smart_induct` method and applies induction on `xs`, followed by auto and done.

Below the code editor, the proof state is shown:

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

The interface includes various toolbars and panels, such as the File Browser, Documentation, Sidekick, State, and Theories. The bottom status bar shows the current proof state, memory usage (512MB), and time (3:23 PM).

1

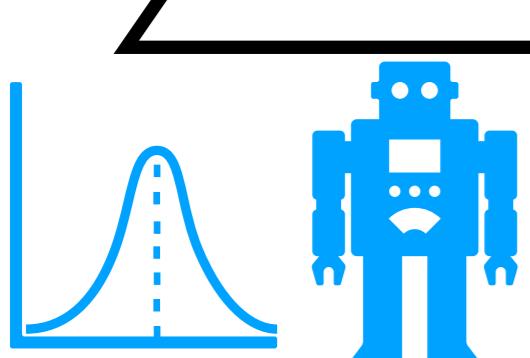


Isabelle's proof methods
(deductive reasoning)

fin!

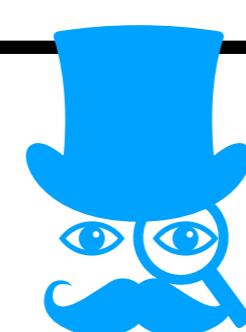
3

machine learning induction
(inductive reasoning)

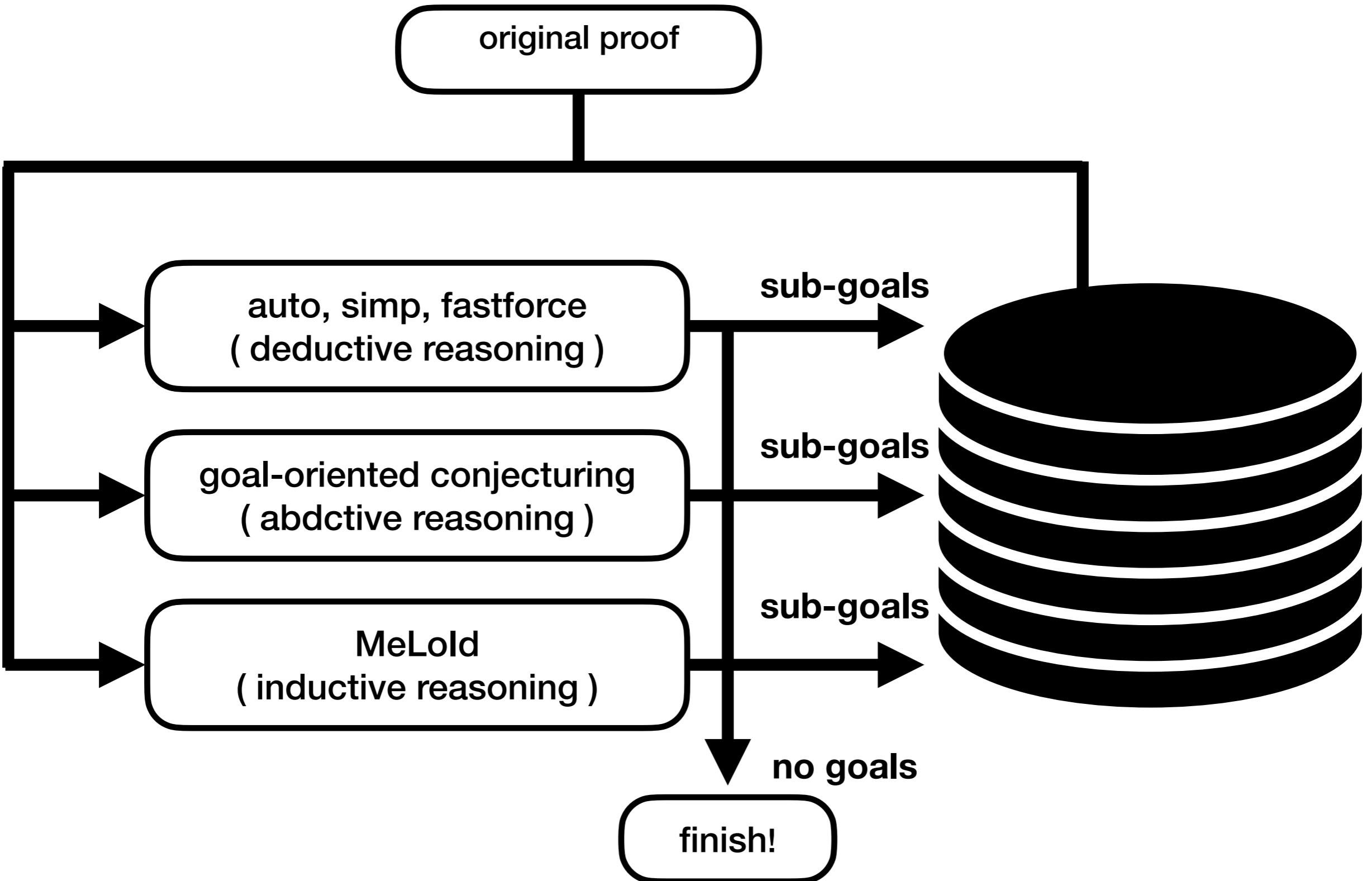


2

goal-oriented conjecturing
(abductive reasoning)



backup slides for Q&A



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

goal

DEMO2

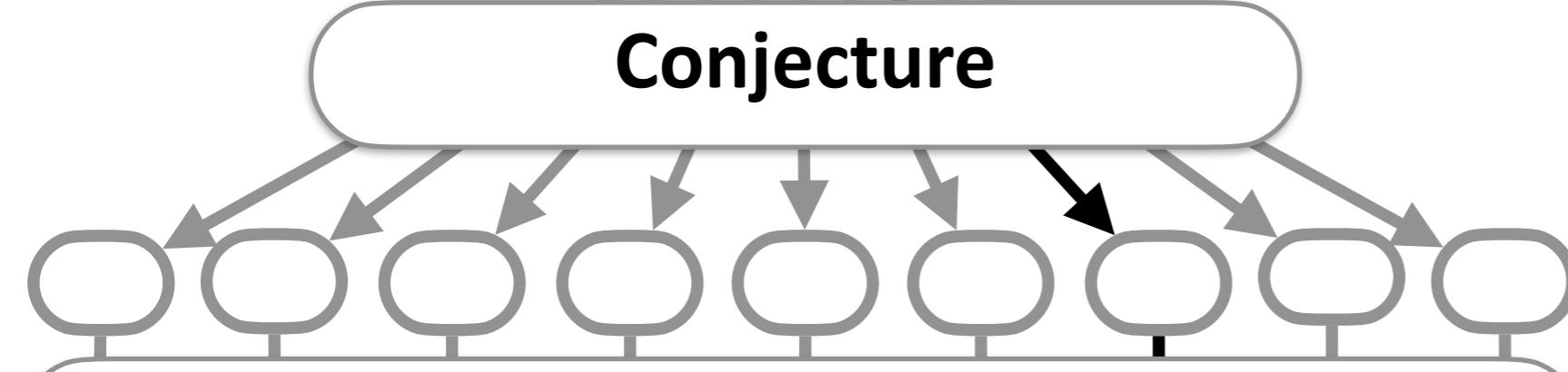
strategy CDInd = Thens [Conjecture, Fastforce, Quickcheck, DInd]

DEMO2

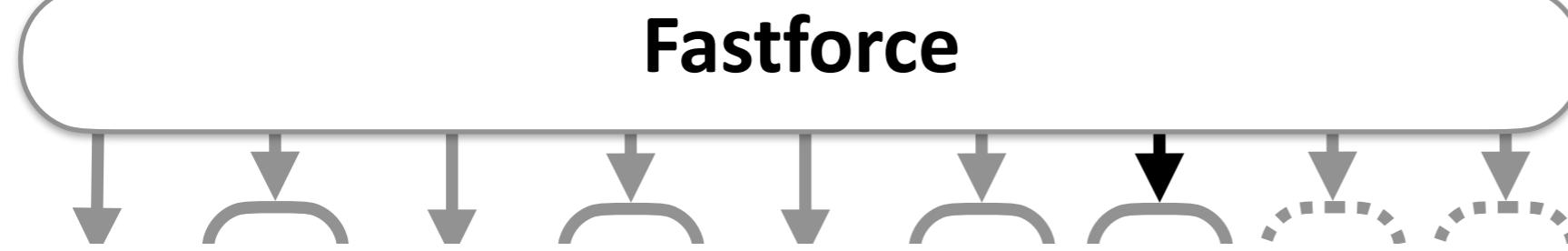
```
goal (1 subgoal):  
1. itrev xs [] = rev xs
```

goal

Conjecture



Fastforce

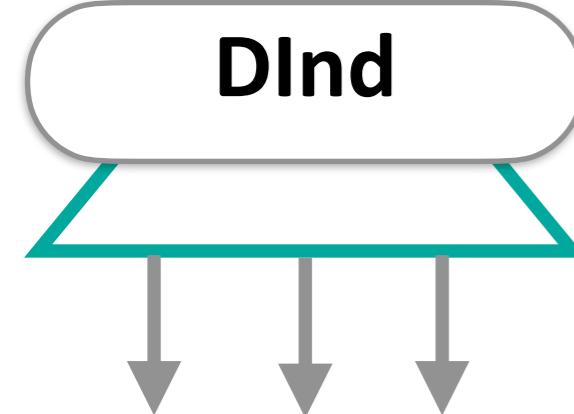


```
strategy CDInd = Thens [Conjecture, Fastforce, Quickcheck, DInd]
```

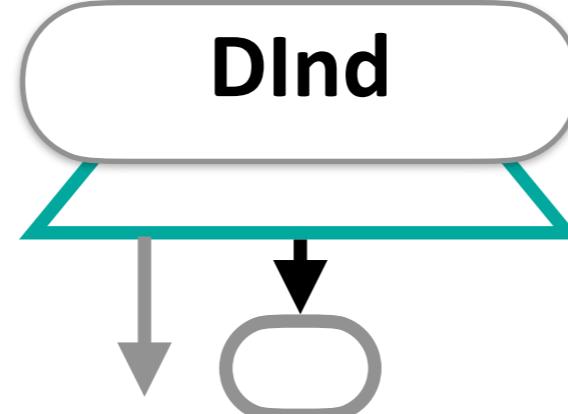
Quickcheck



DInd

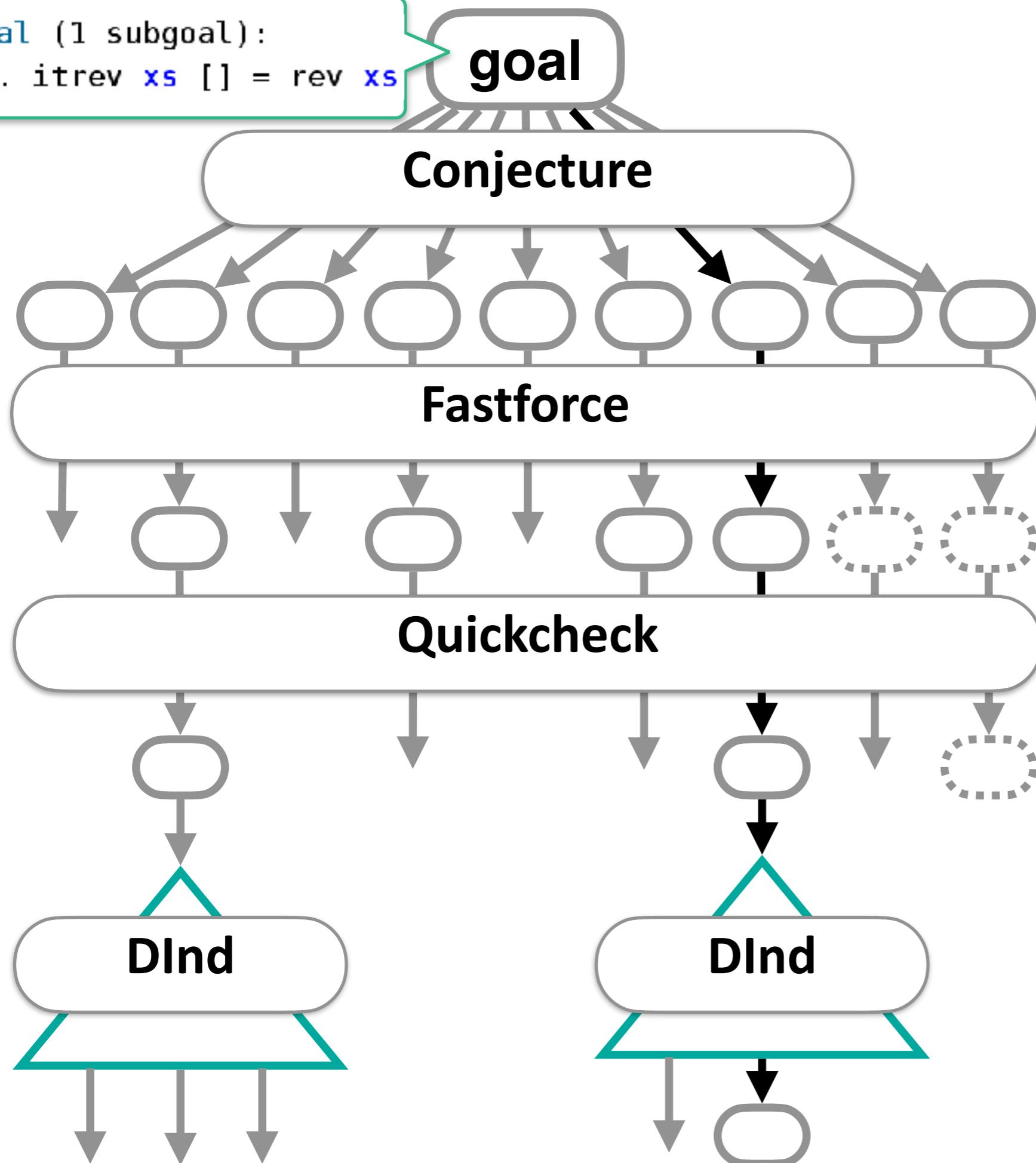


DInd



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")

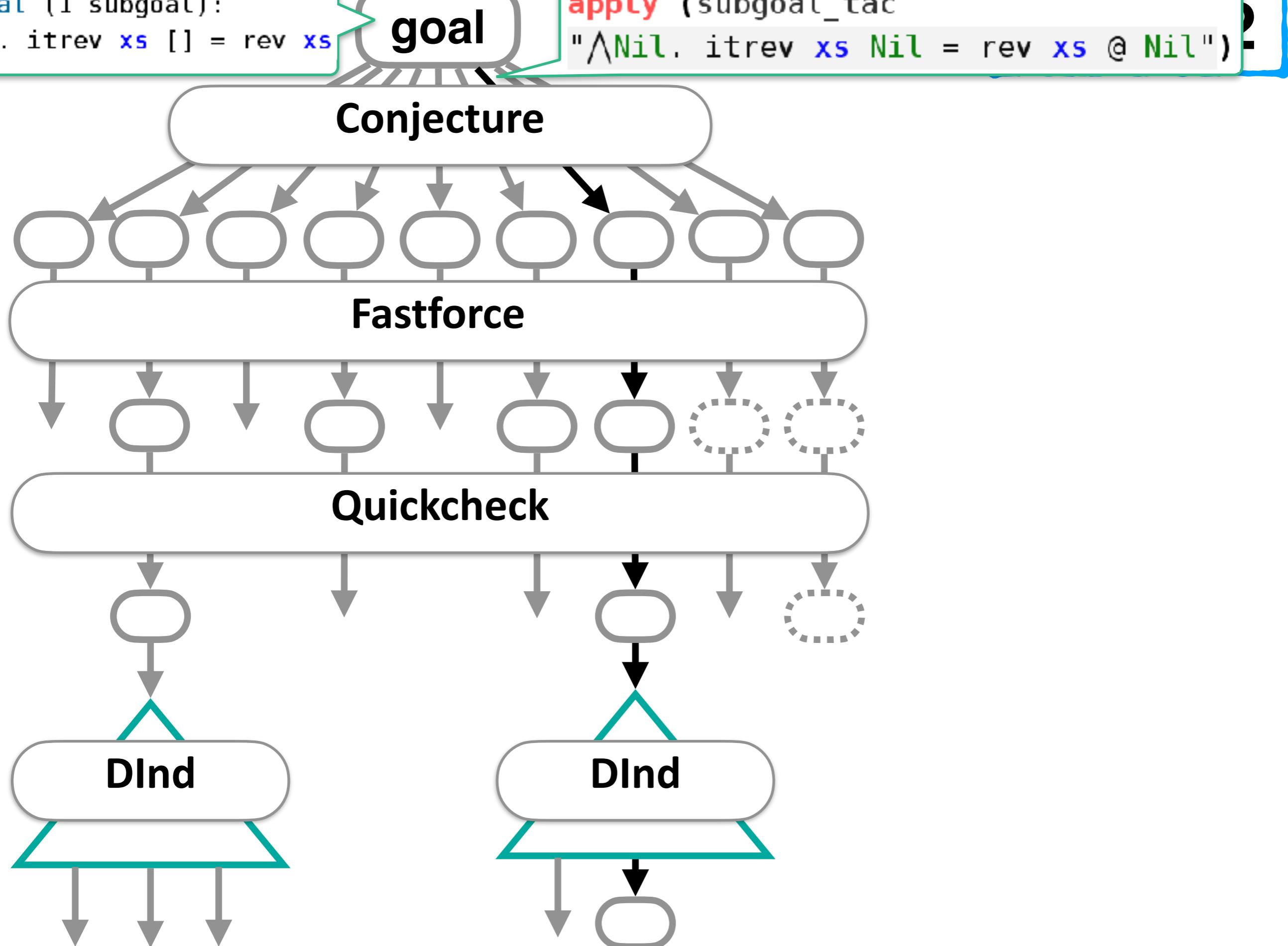
Conjecture

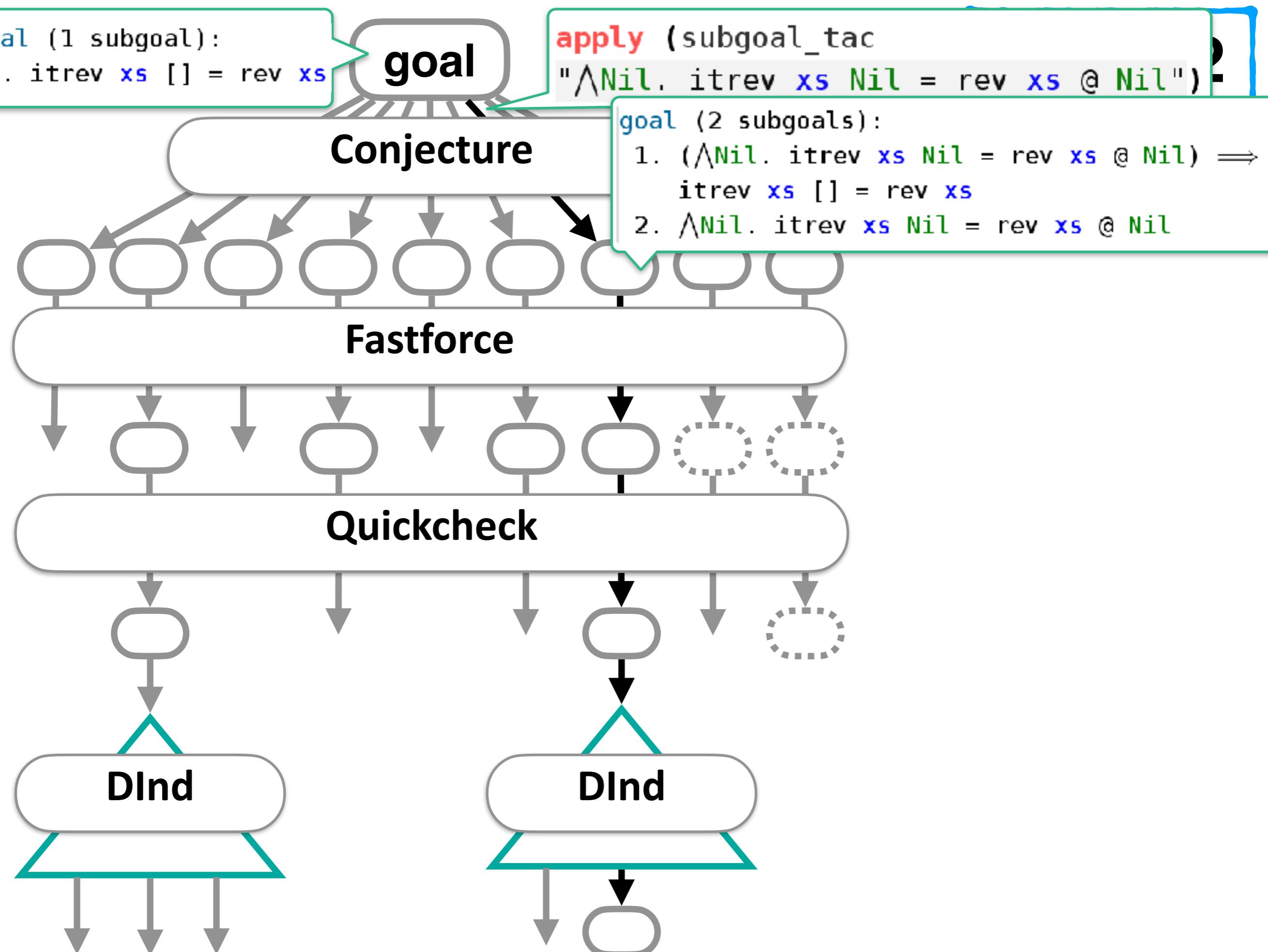
Fastforce

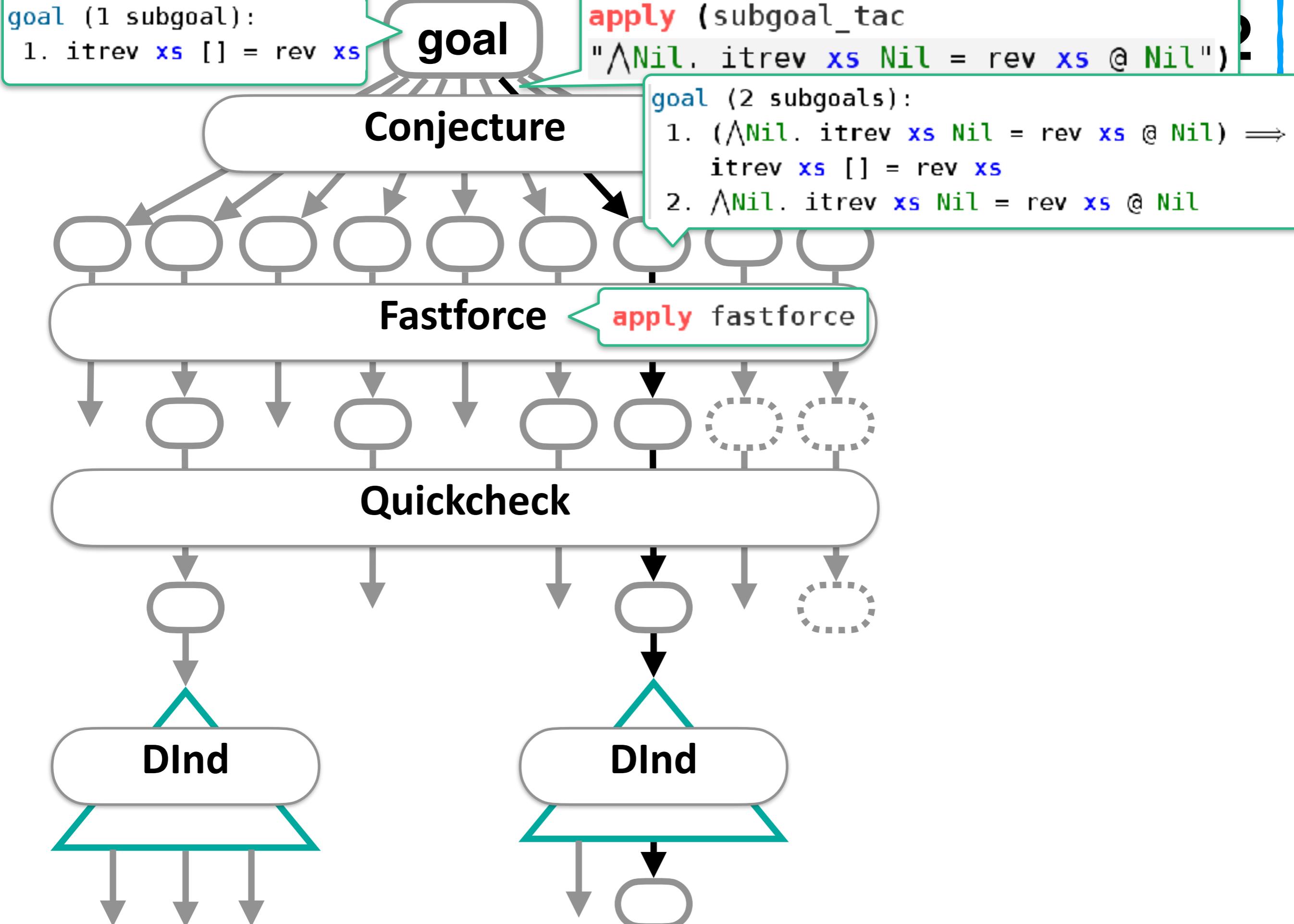
Quickcheck

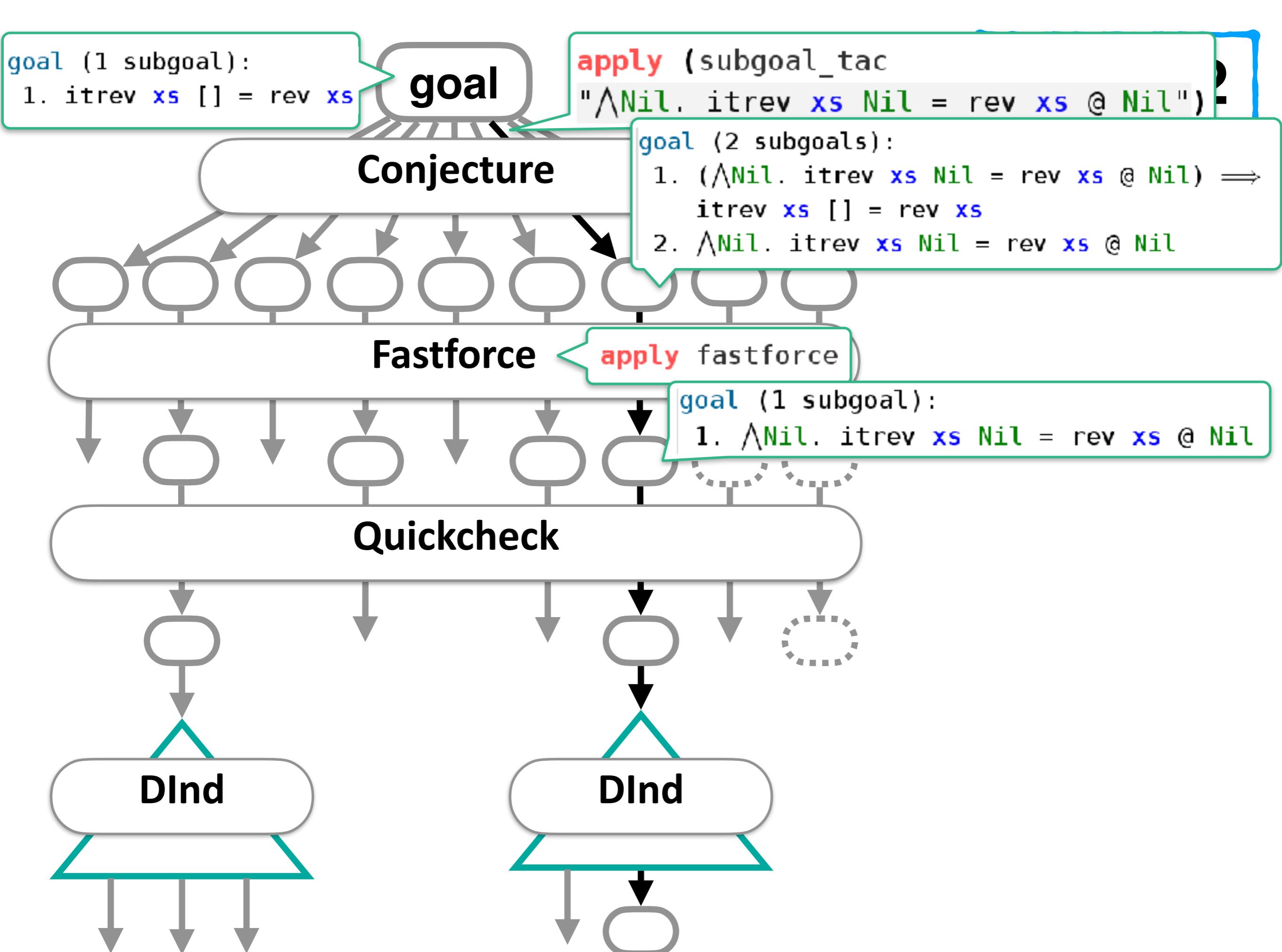
DInd

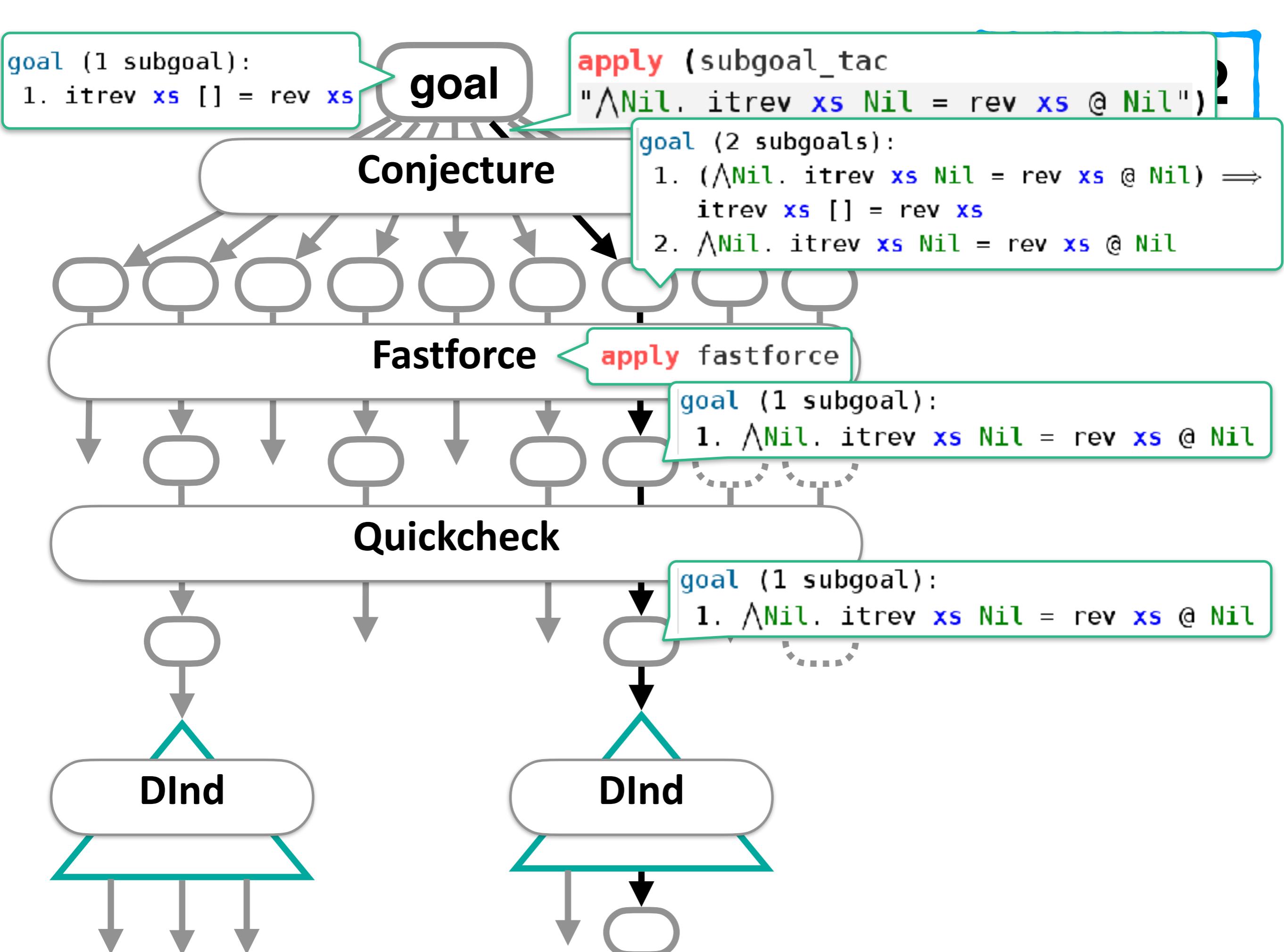
DInd

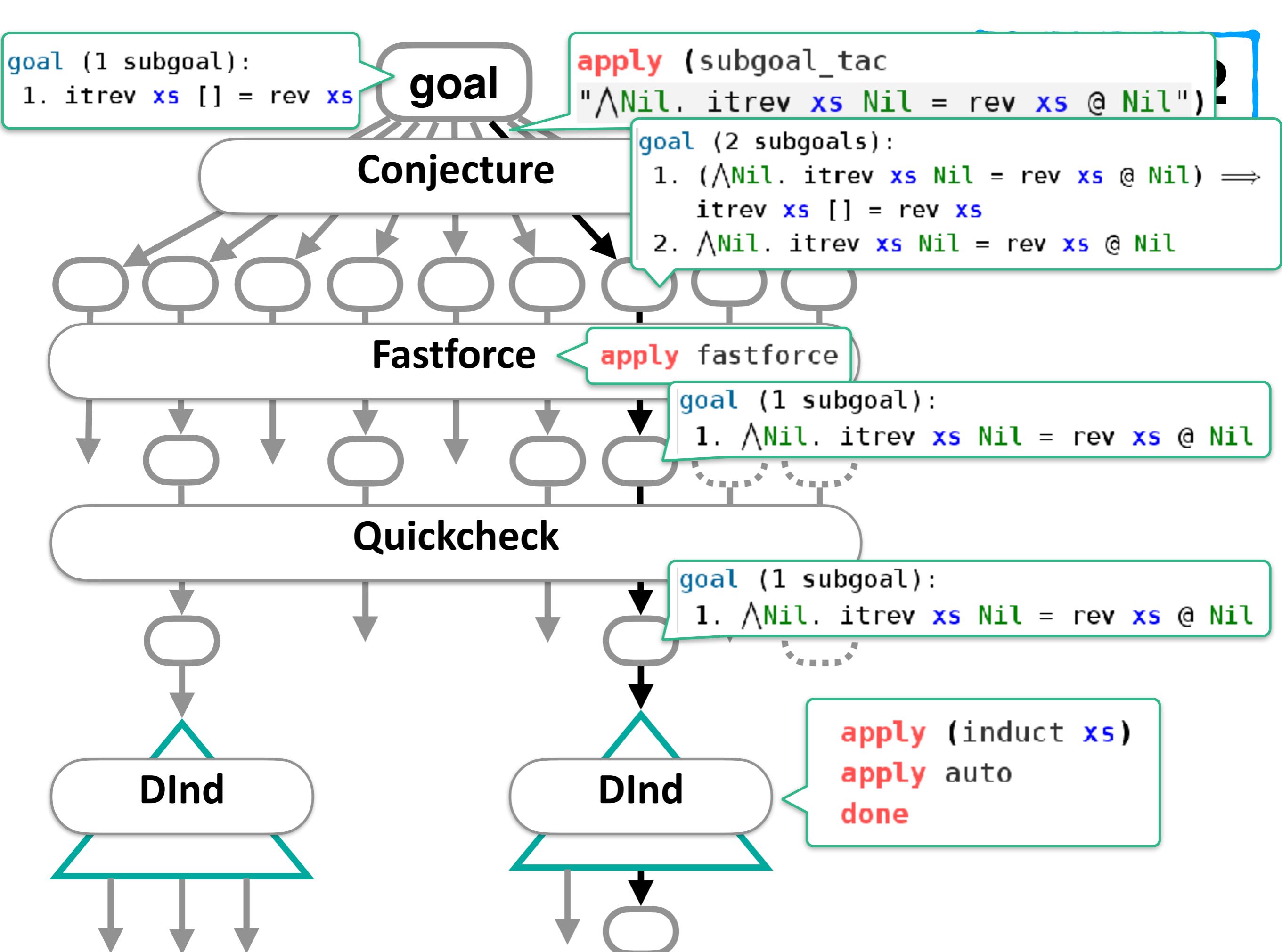


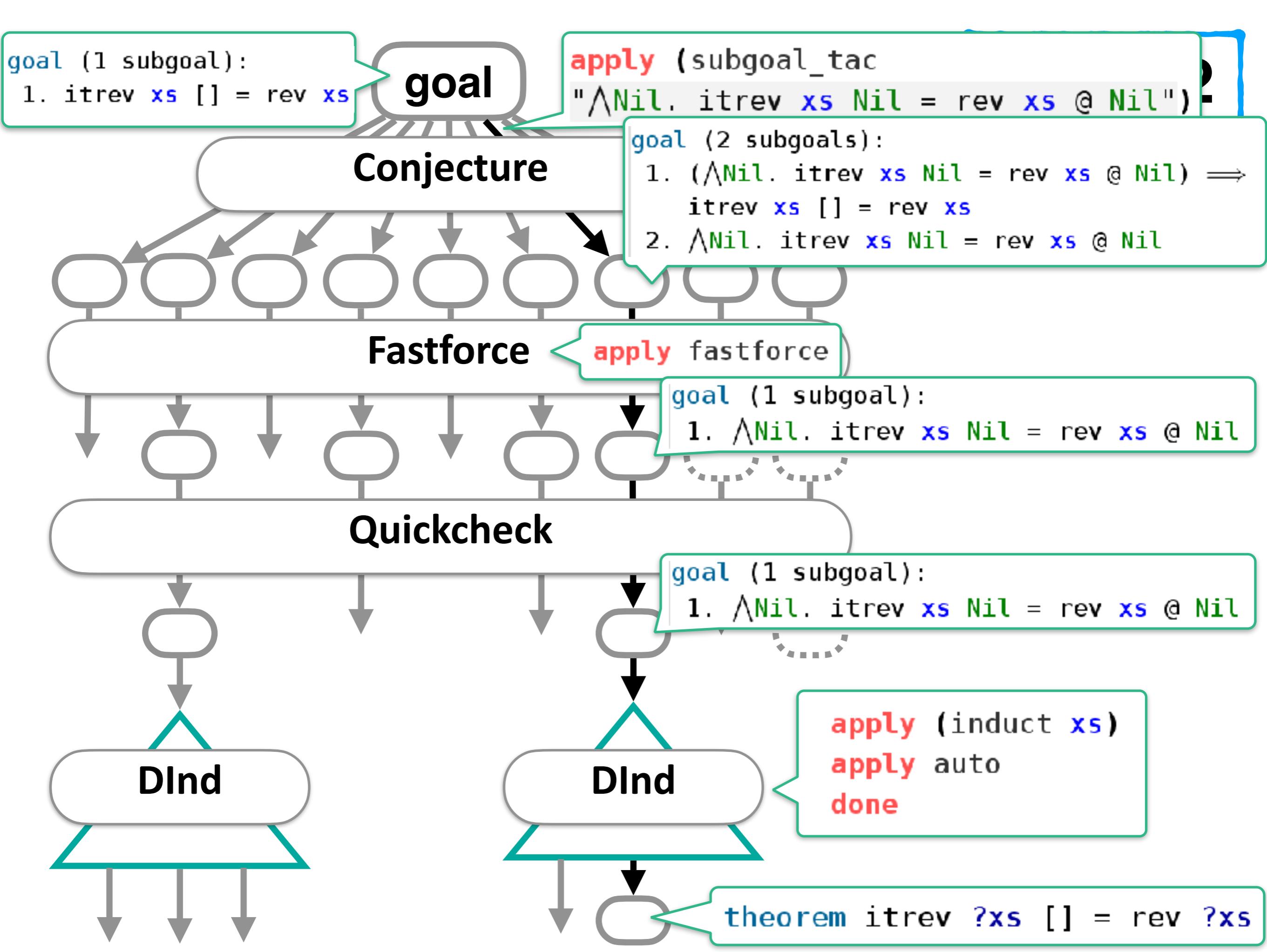


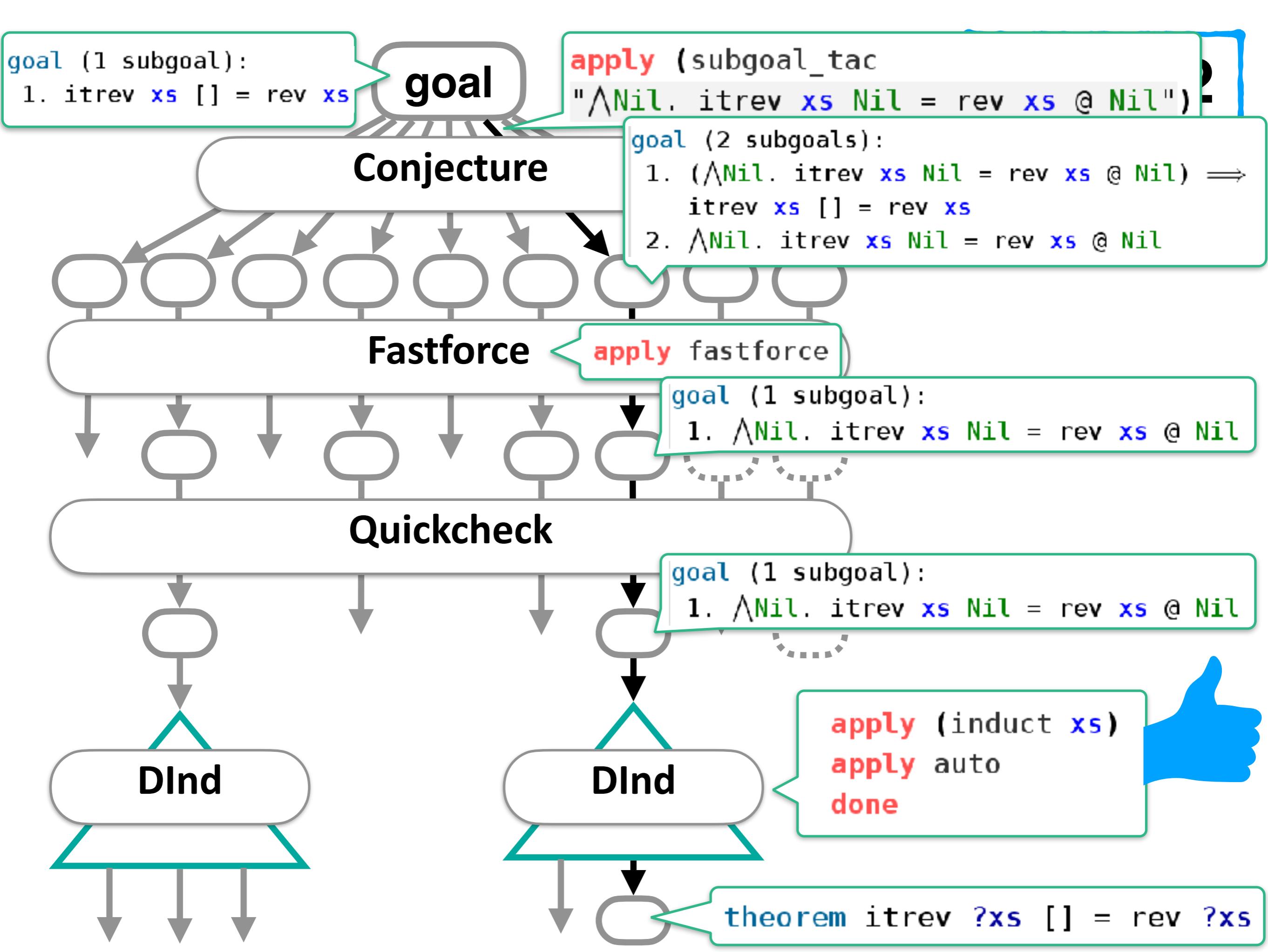












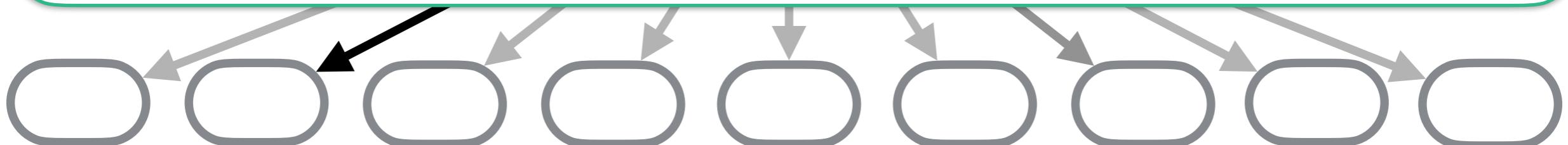
smart_induct

goal

smart_induct

goal

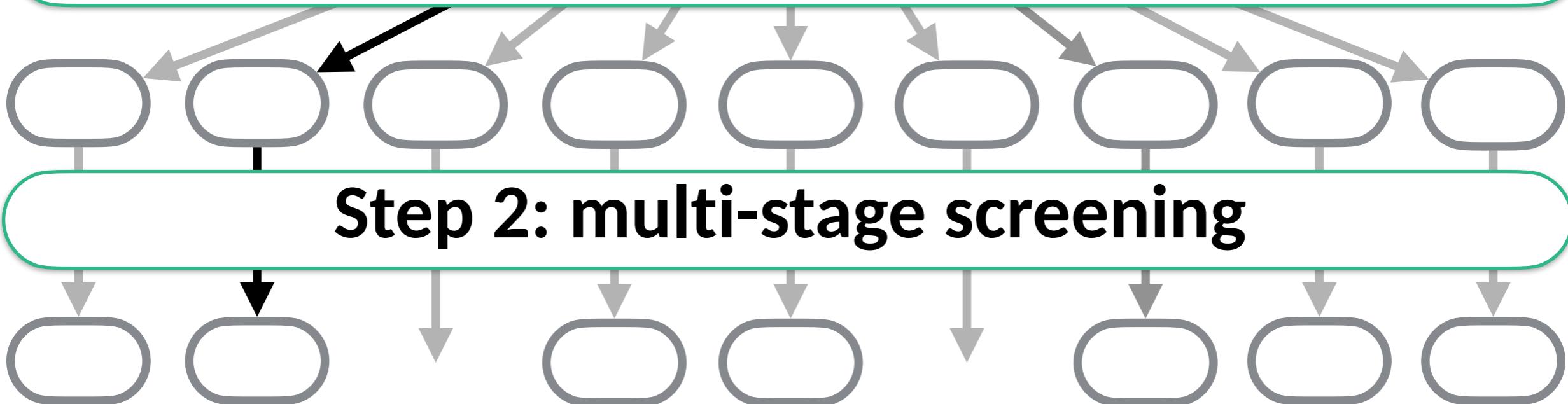
Step 1: creating many inductions



smart_induct

goal

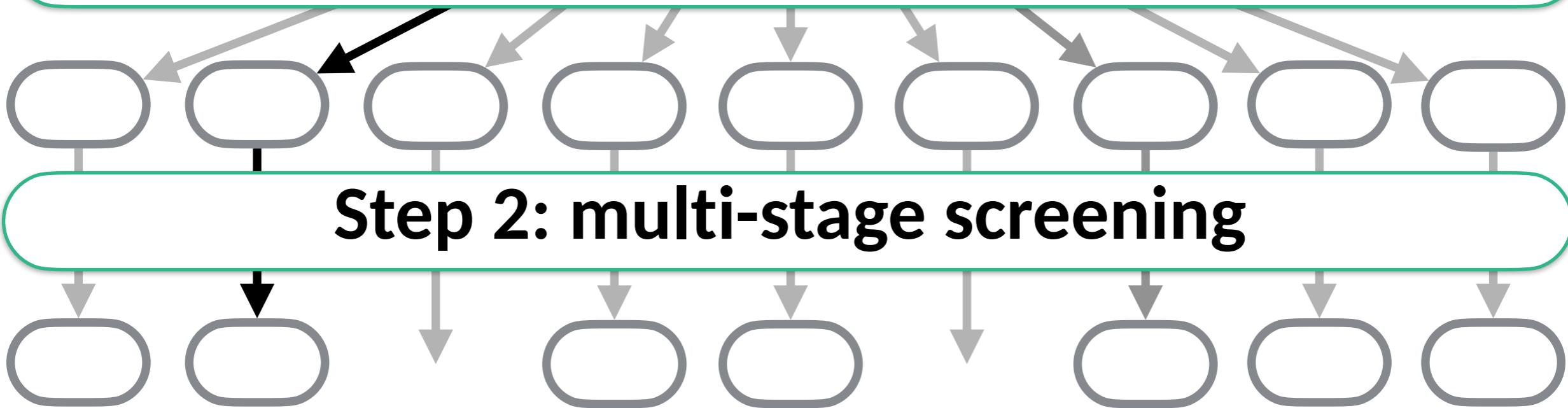
Step 1: creating many inductions



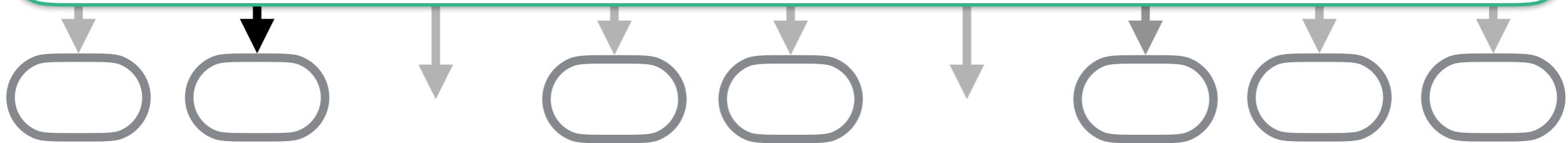
smart_induct

goal

Step 1: creating many inductions



Step 2: multi-stage screening

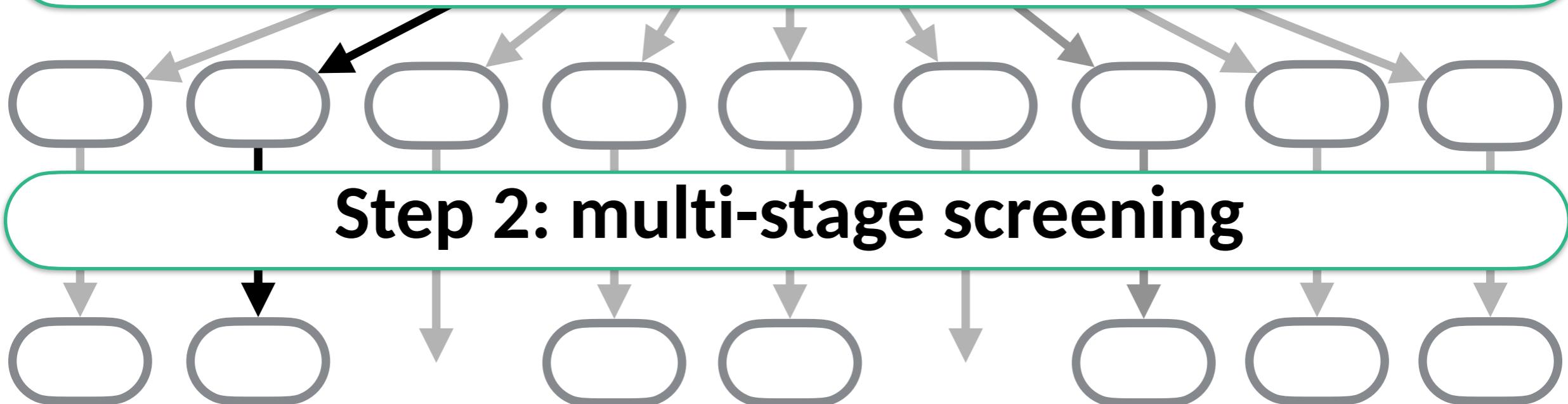


Step 3: scoring using 20 heuristics and sorting

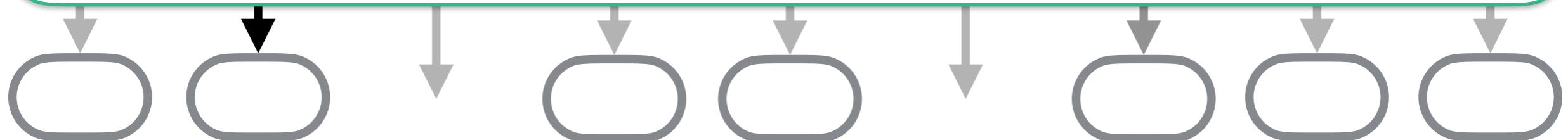
smart_induct

goal

Step 1: creating many inductions



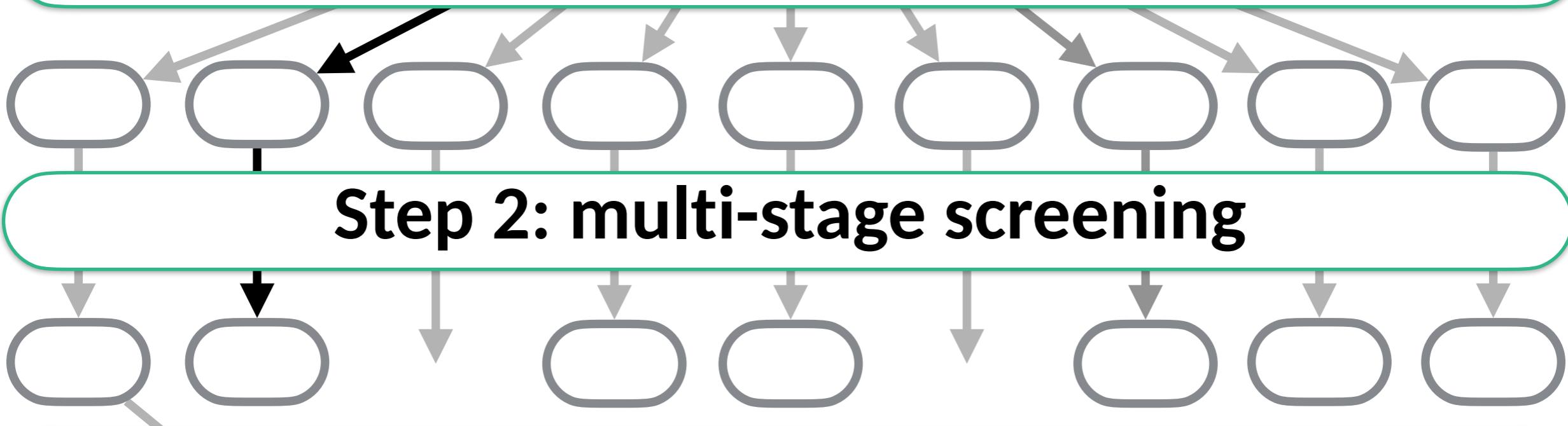
Step 2: multi-stage screening



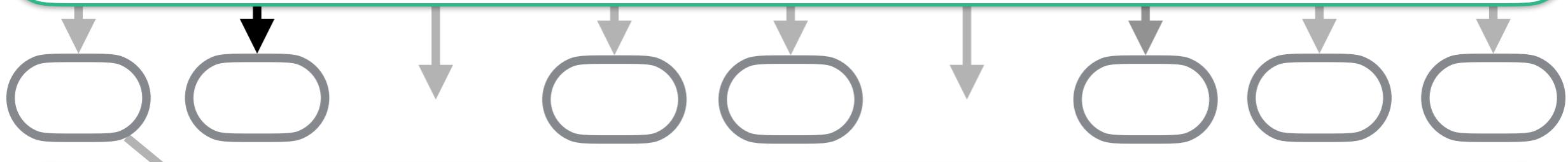
Step 3: scoring using 20 heuristics and sorting

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

Step 1: creating many inductions



Step 2: multi-stage screening



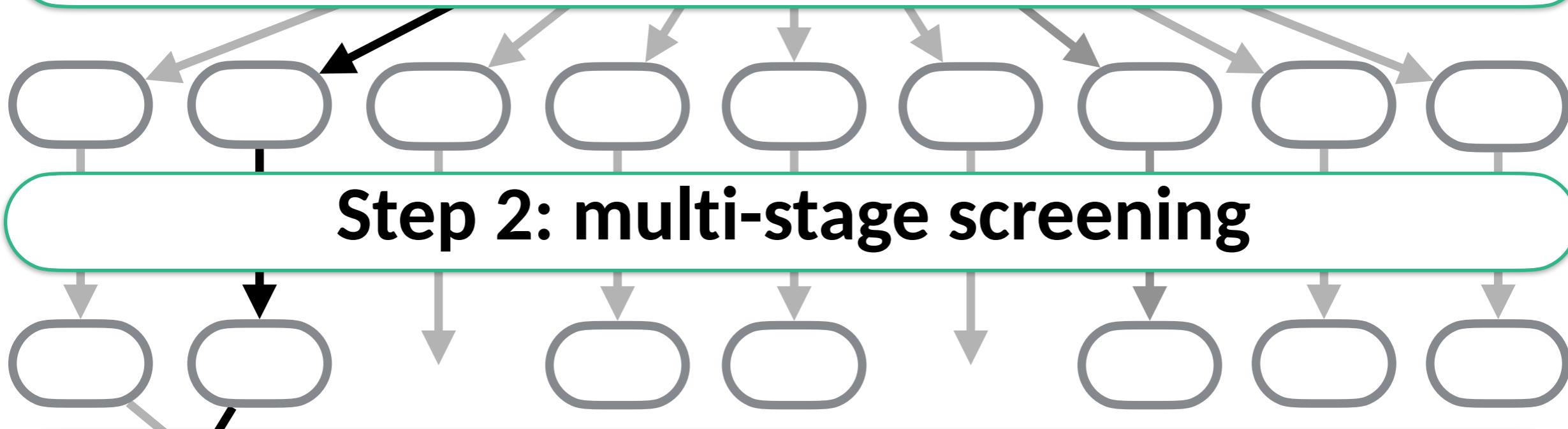
Step 3: scoring using 20 heuristics and sorting

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

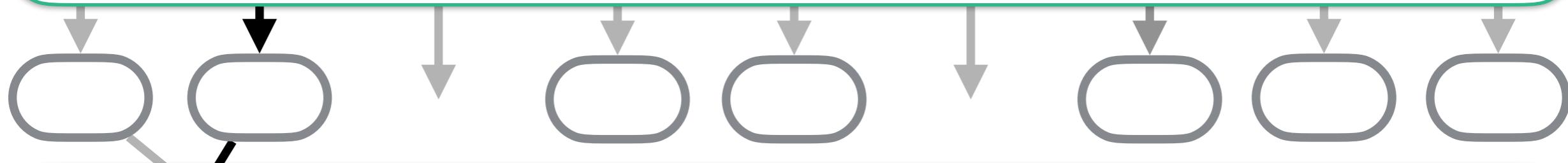
smart_induct

goal

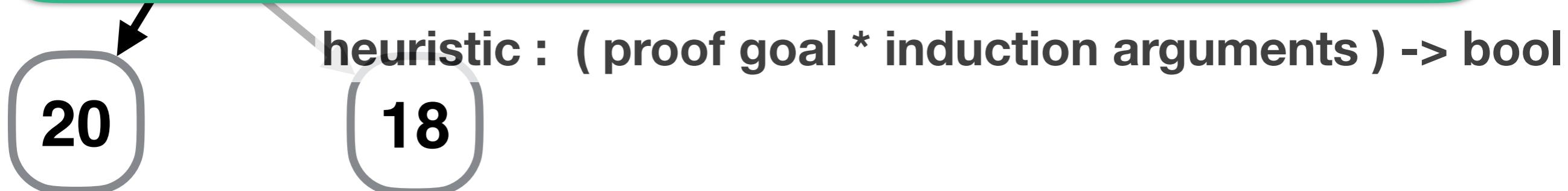
Step 1: creating many inductions



Step 2: multi-stage screening



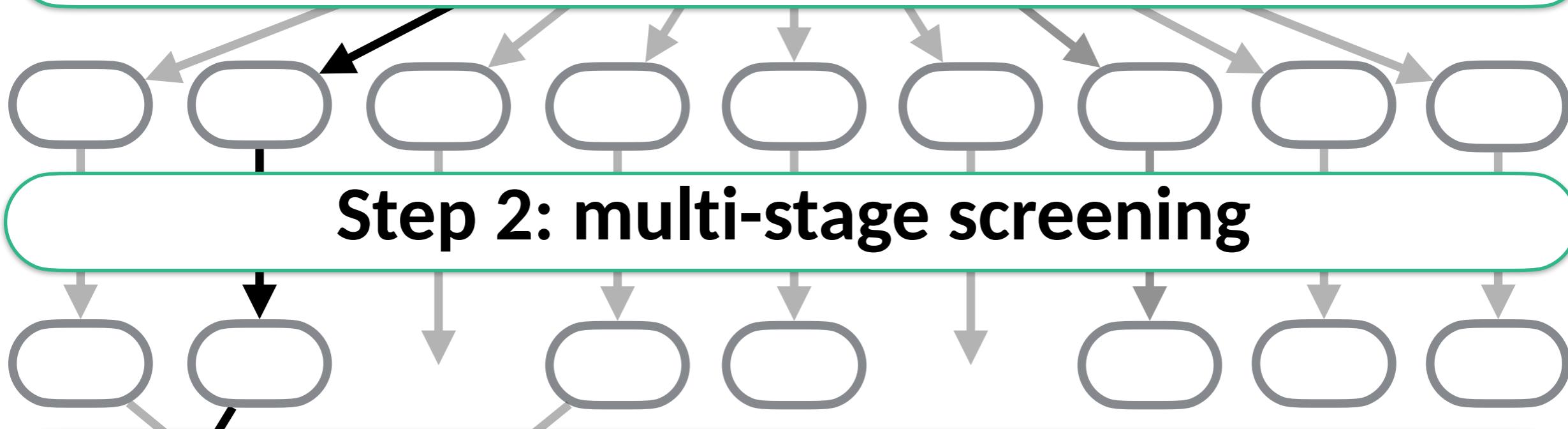
Step 3: scoring using 20 heuristics and sorting



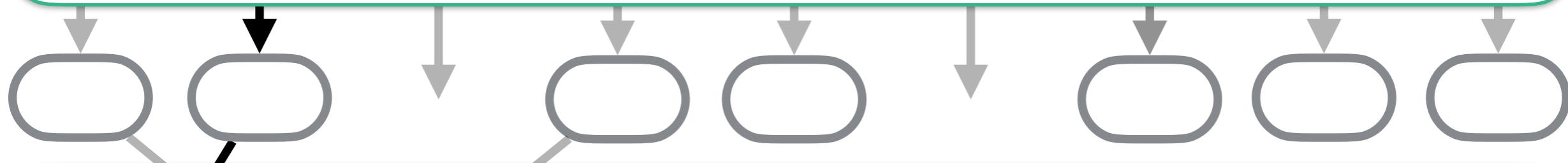
smart_induct

goal

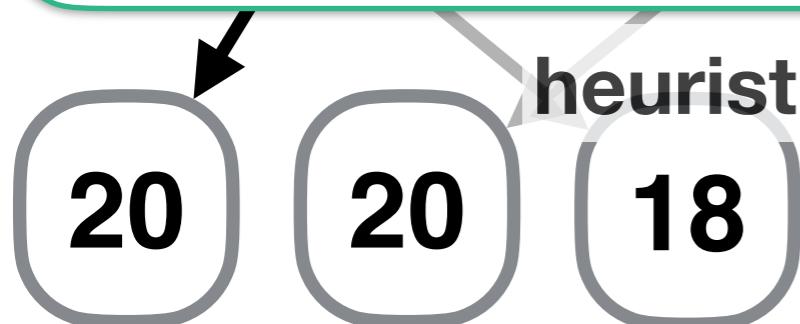
Step 1: creating many inductions



Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting



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

20

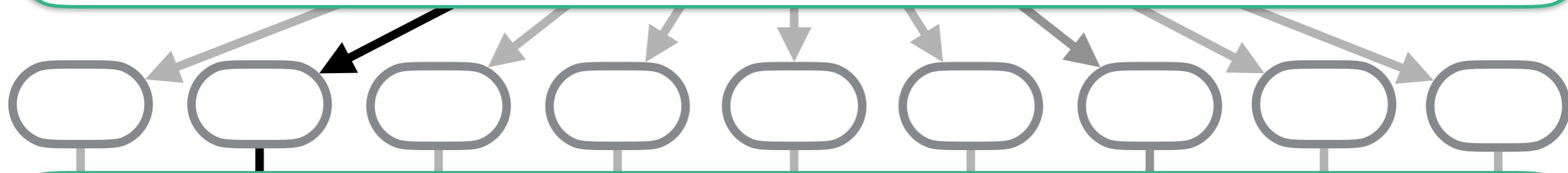
20

18

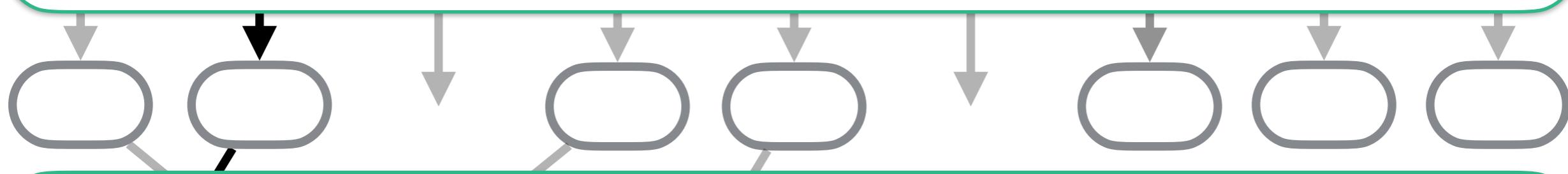
smart_induct

goal

Step 1: creating many inductions



Step 2: multi-stage screening



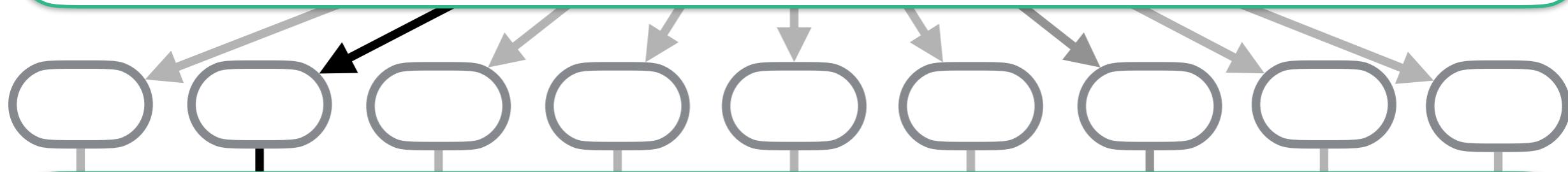
Step 3: scoring using 20 heuristics and sorting



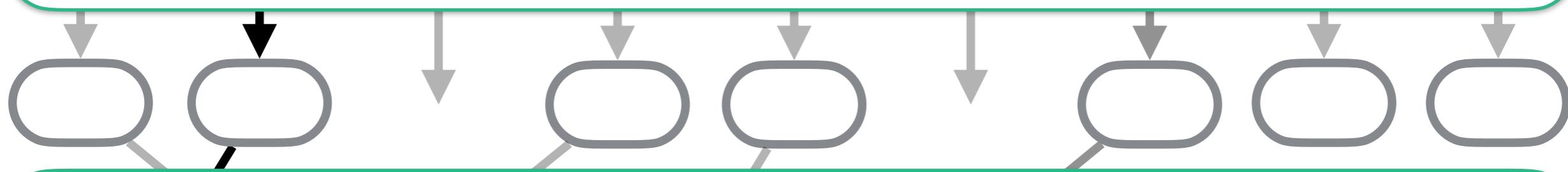
smart_induct

goal

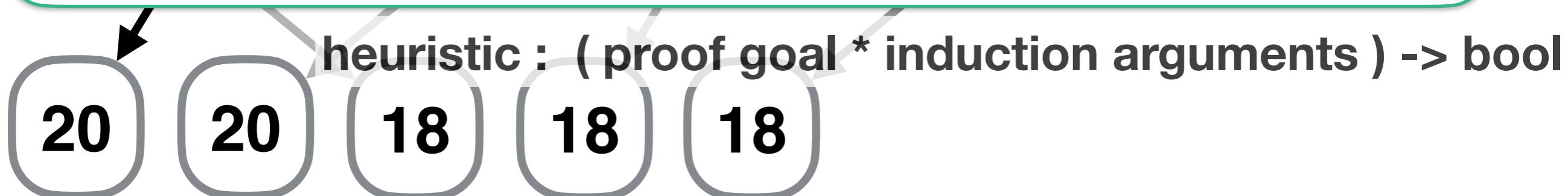
Step 1: creating many inductions



Step 2: multi-stage screening



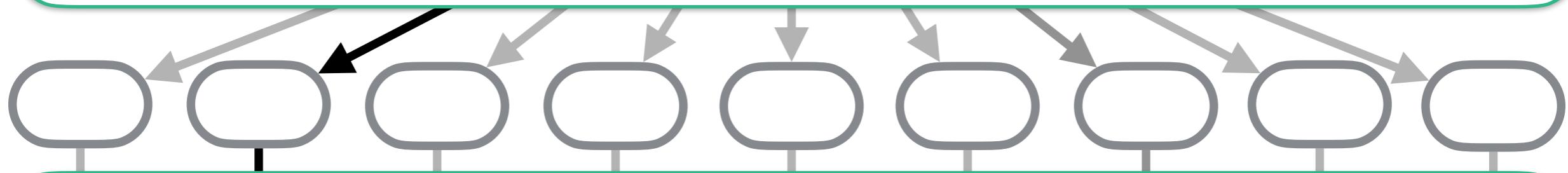
Step 3: scoring using 20 heuristics and sorting



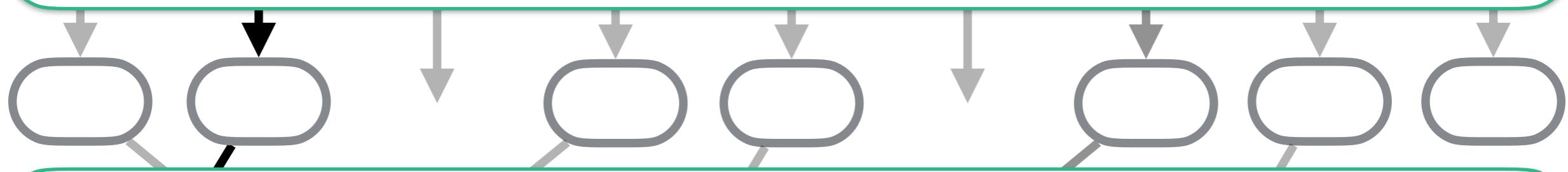
smart_induct

goal

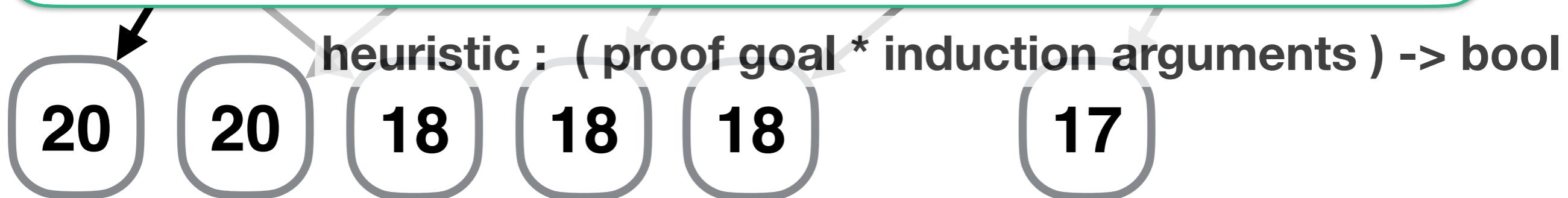
Step 1: creating many inductions



Step 2: multi-stage screening



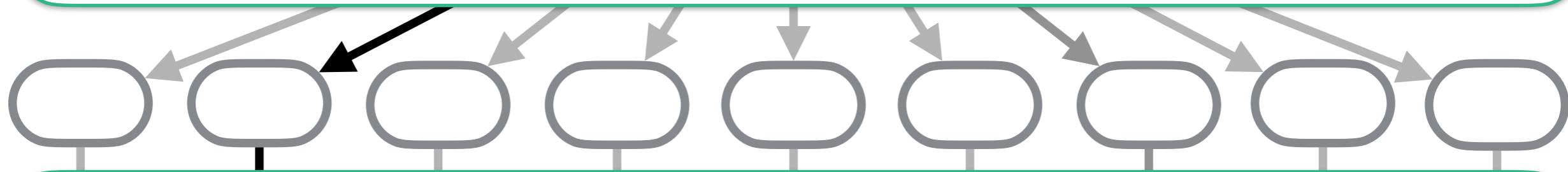
Step 3: scoring using 20 heuristics and sorting



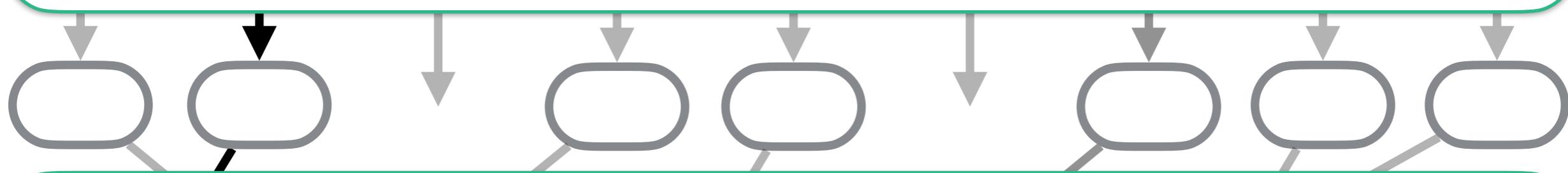
smart_induct

goal

Step 1: creating many inductions



Step 2: multi-stage screening



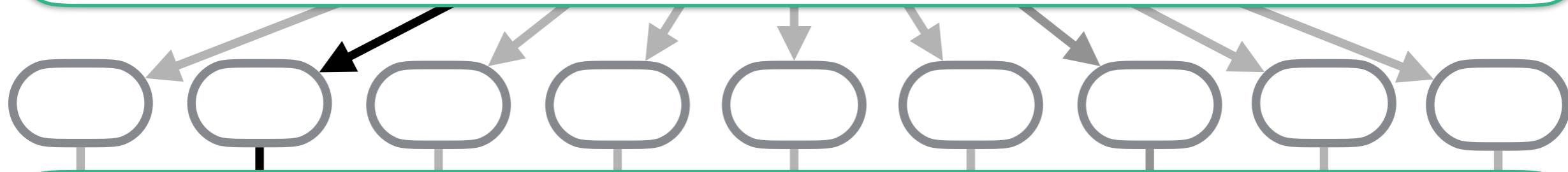
Step 3: scoring using 20 heuristics and sorting



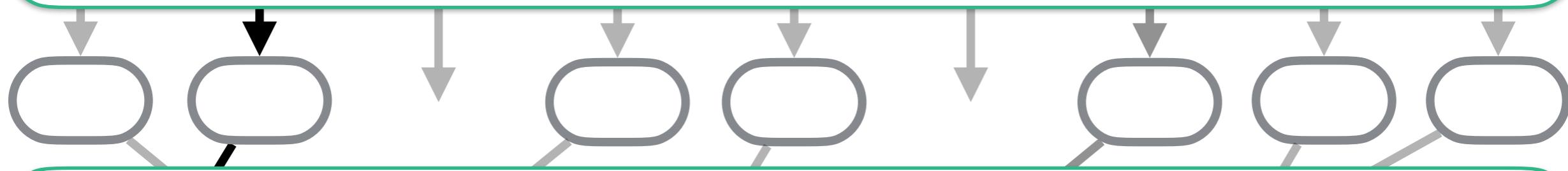
smart_induct

goal

Step 1: creating many inductions



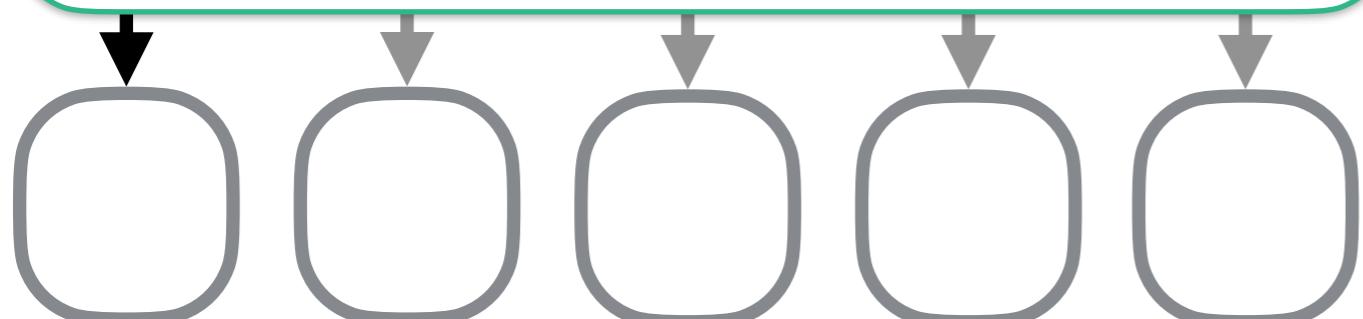
Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting



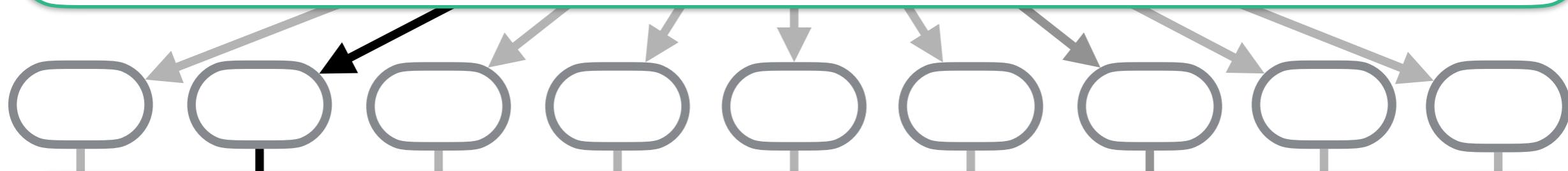
Step 4: short-listing



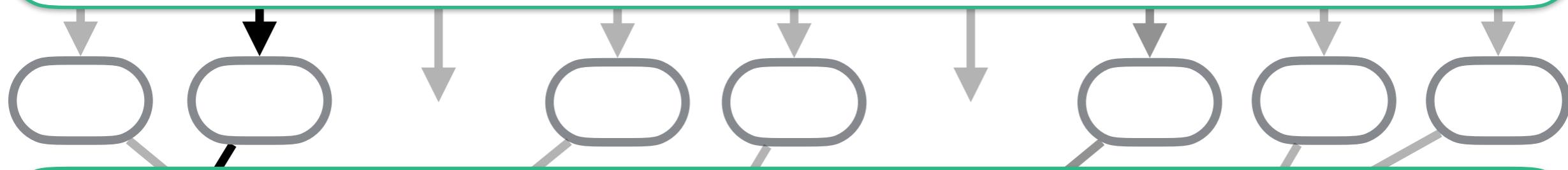
smart_induct

goal

Step 1: creating many inductions



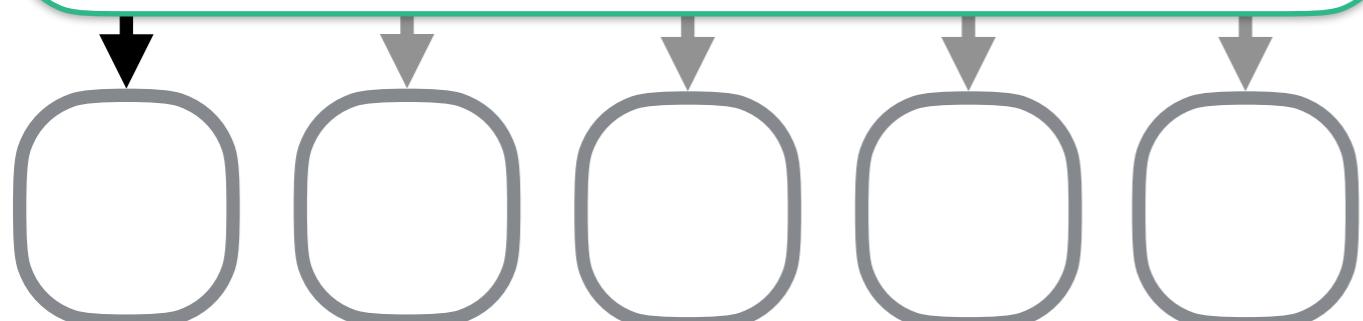
Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting



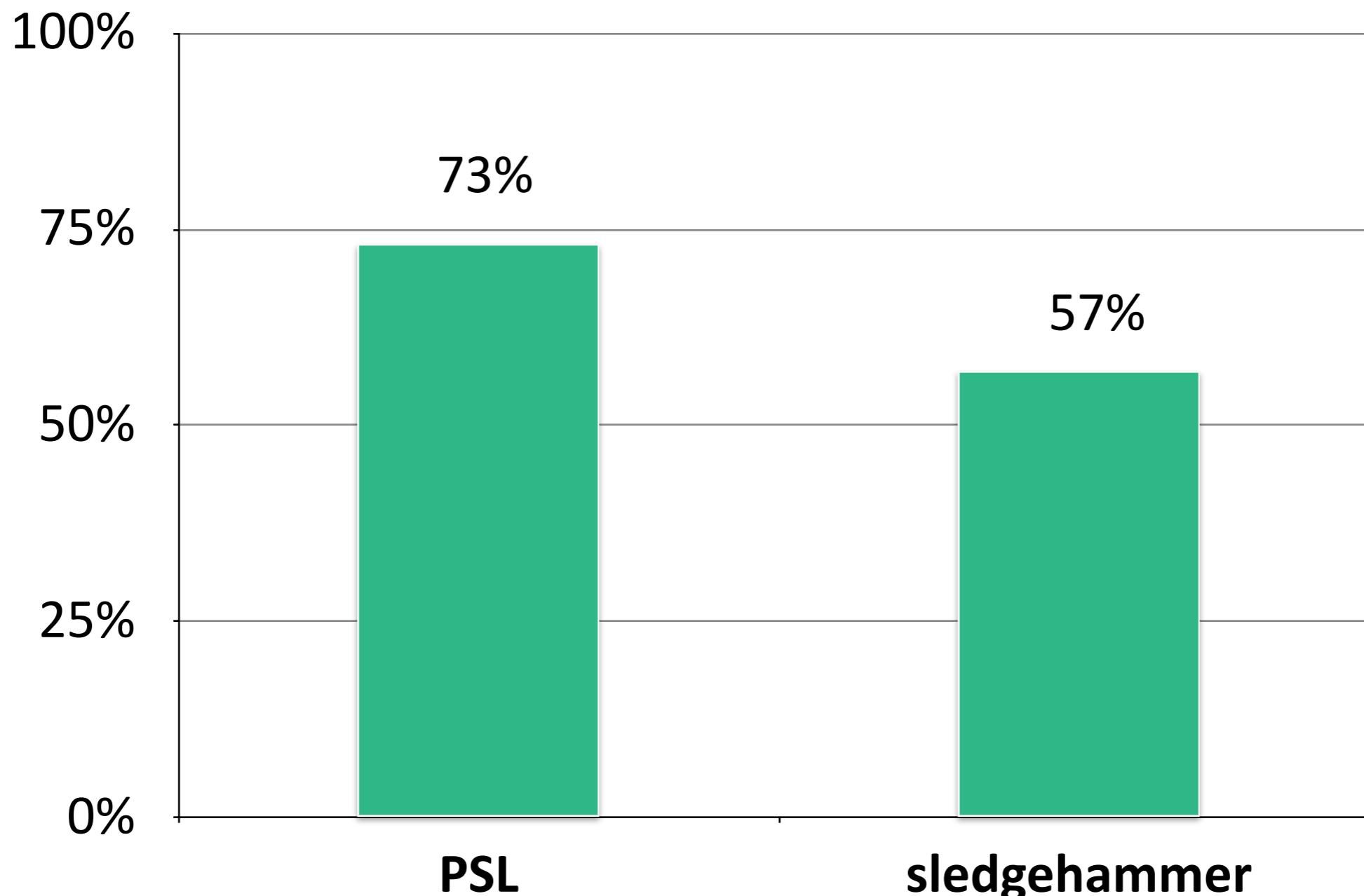
Step 4: short-listing



DEMO!

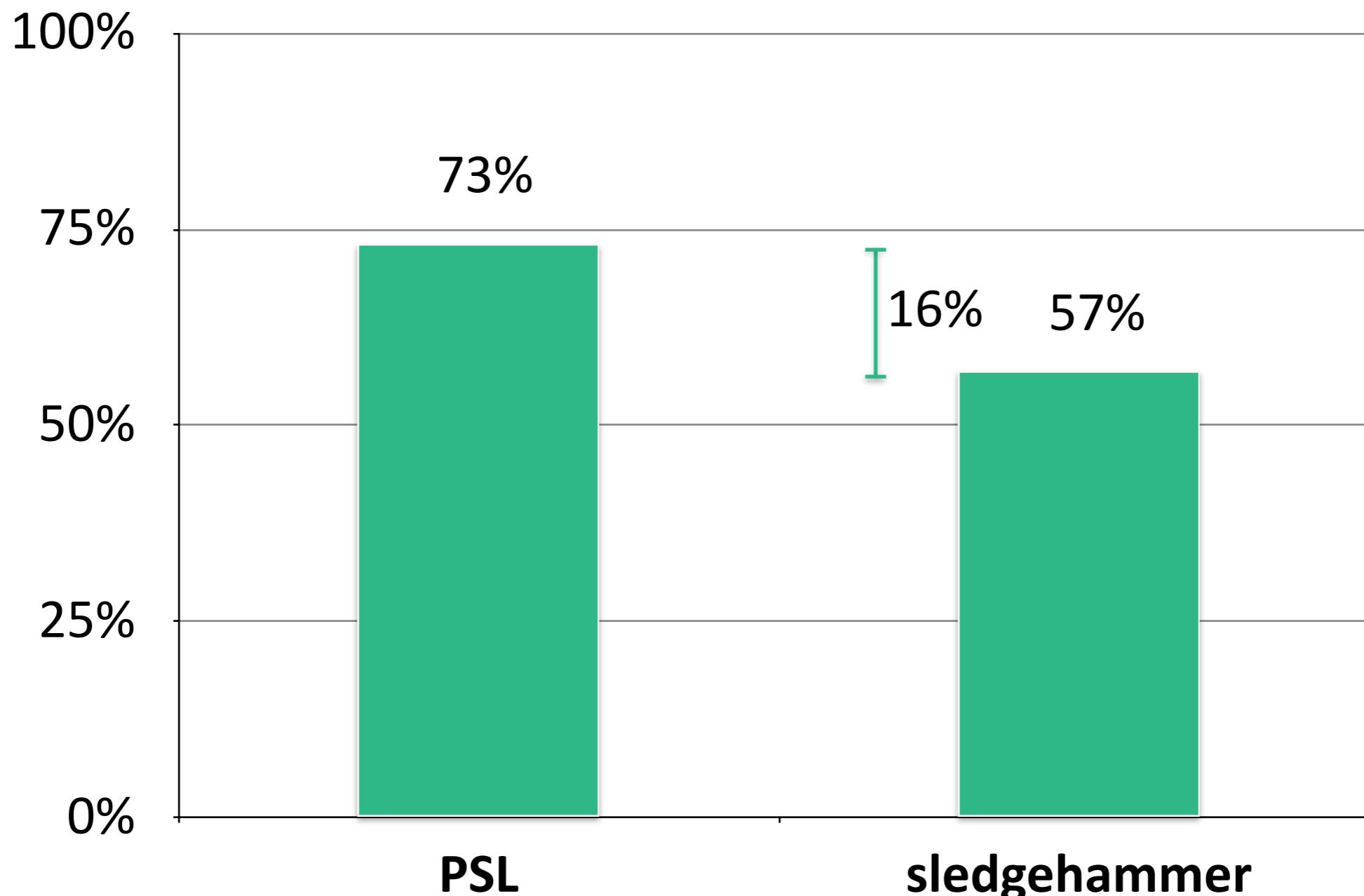
PSL vs sledgehammer

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



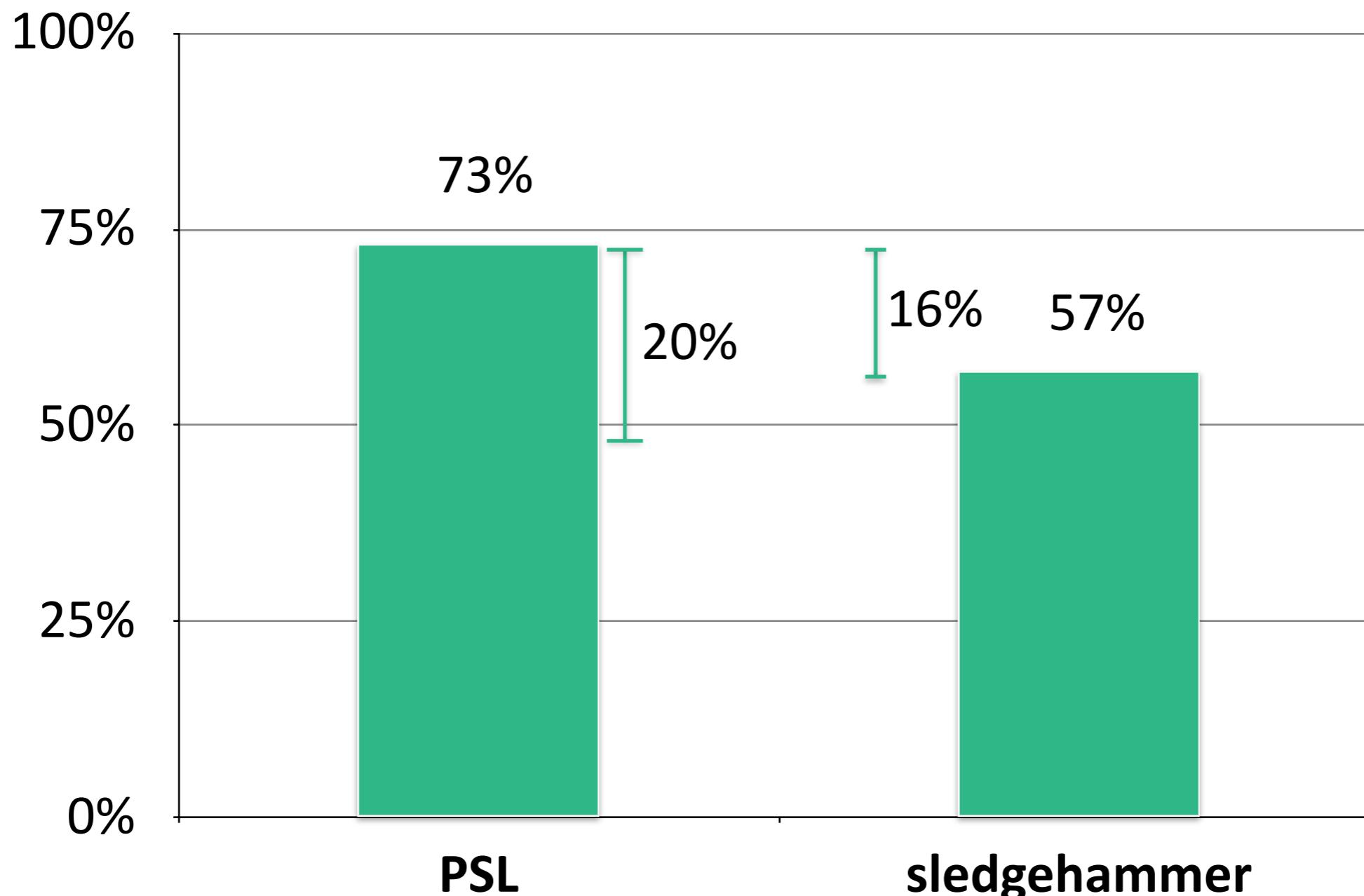
PSL vs sledgehammer

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

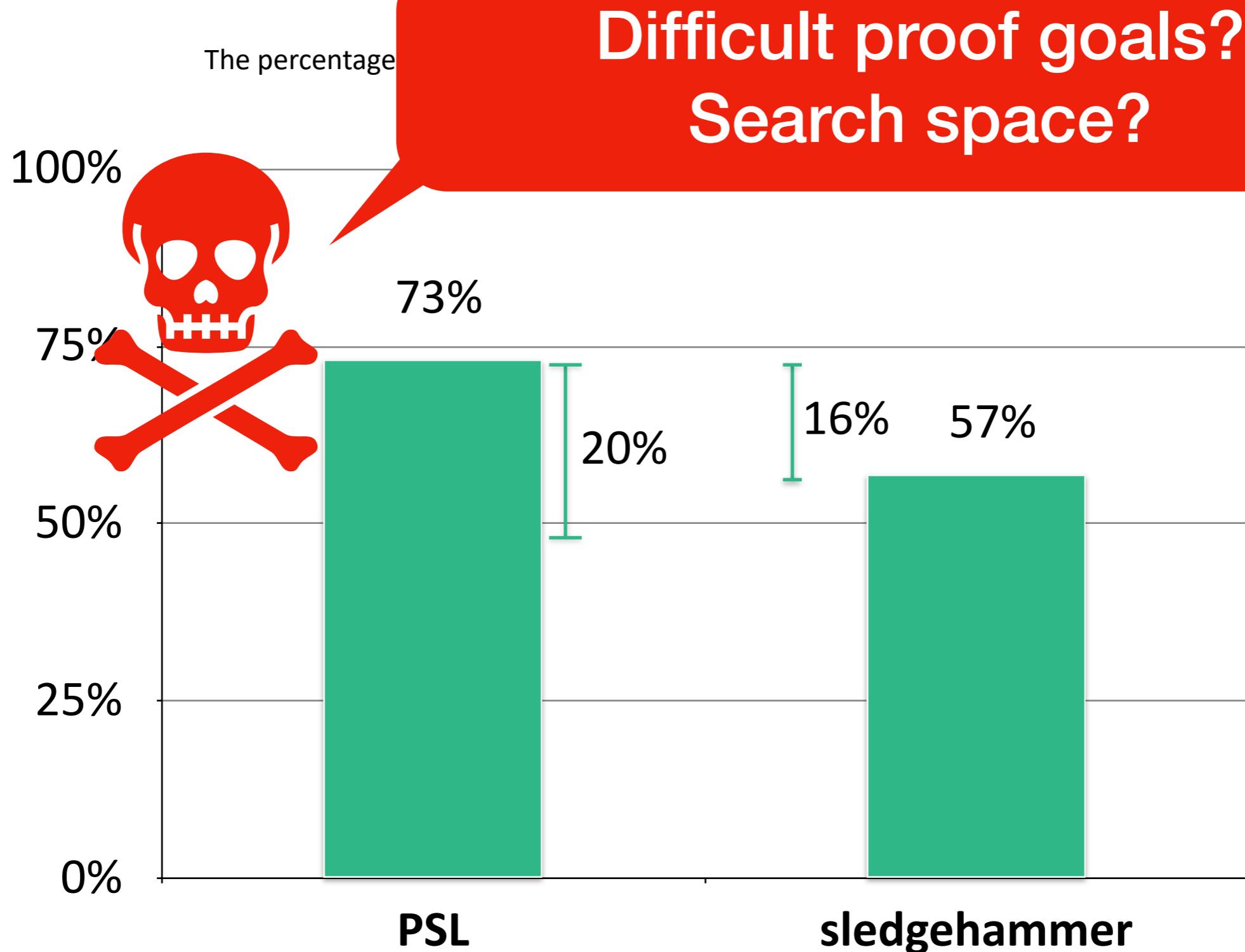


PSL vs sledgehammer

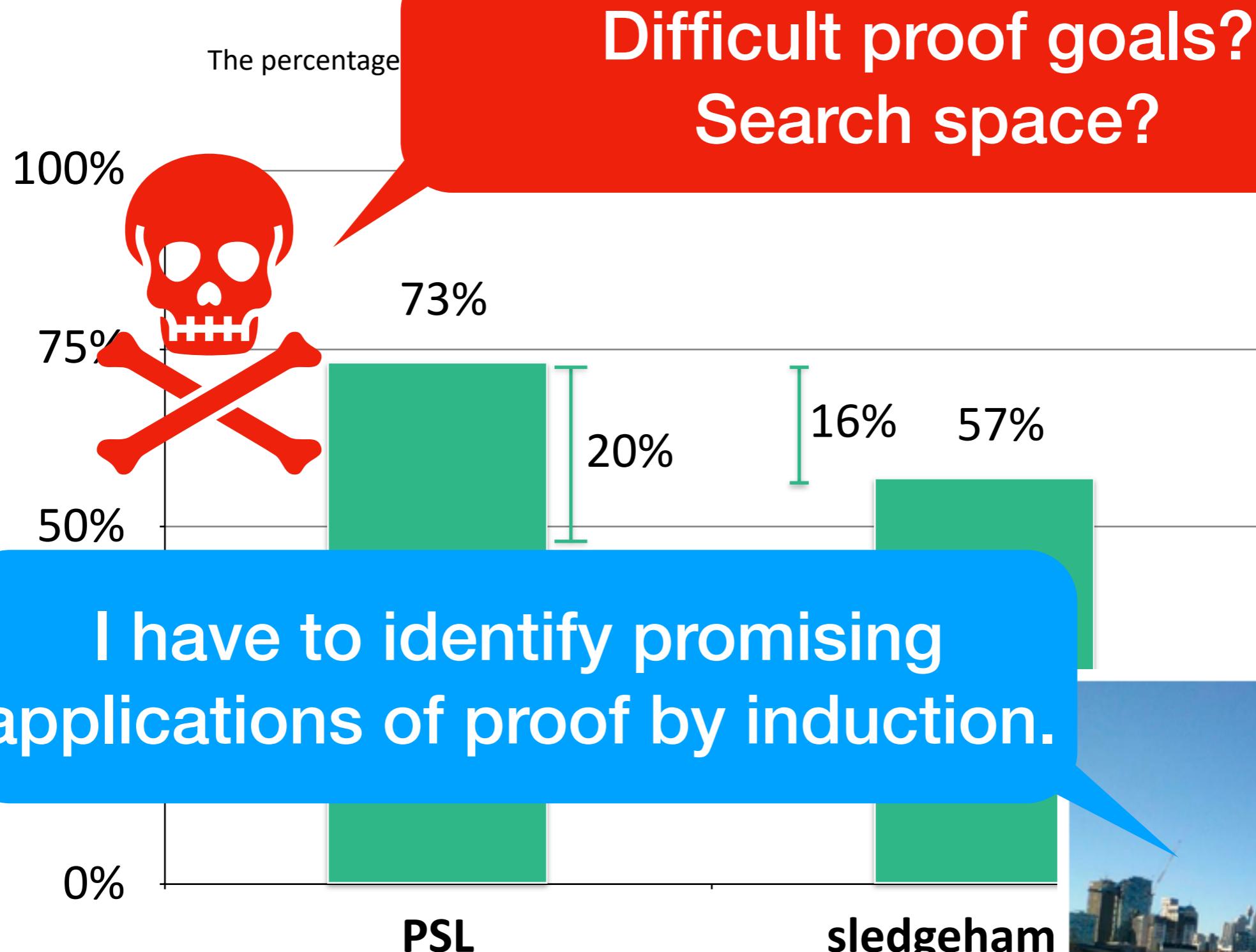
The percentage of automatically proved obligations out of 1526 proof obligations
(timeout = 300s)



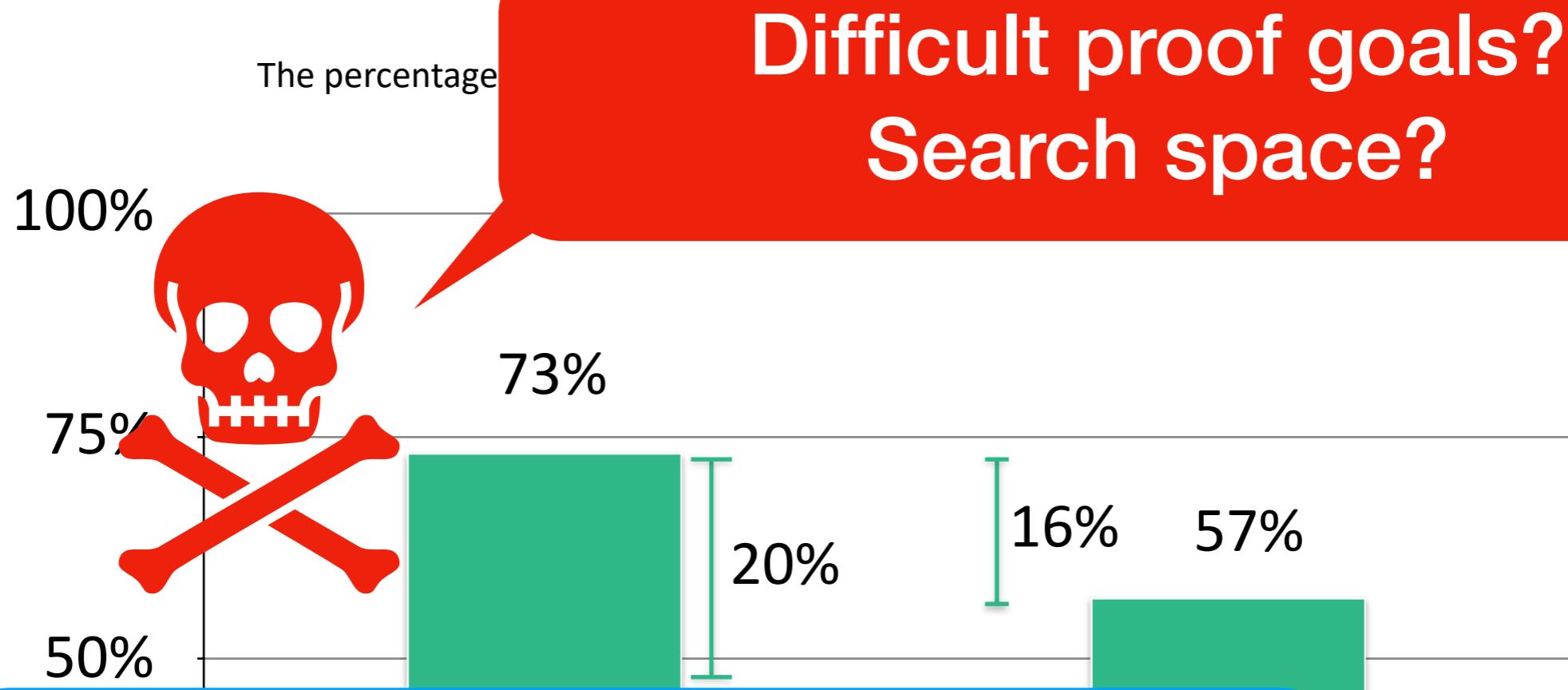
PSL vs sledgehammer



PSL vs sledgehammer



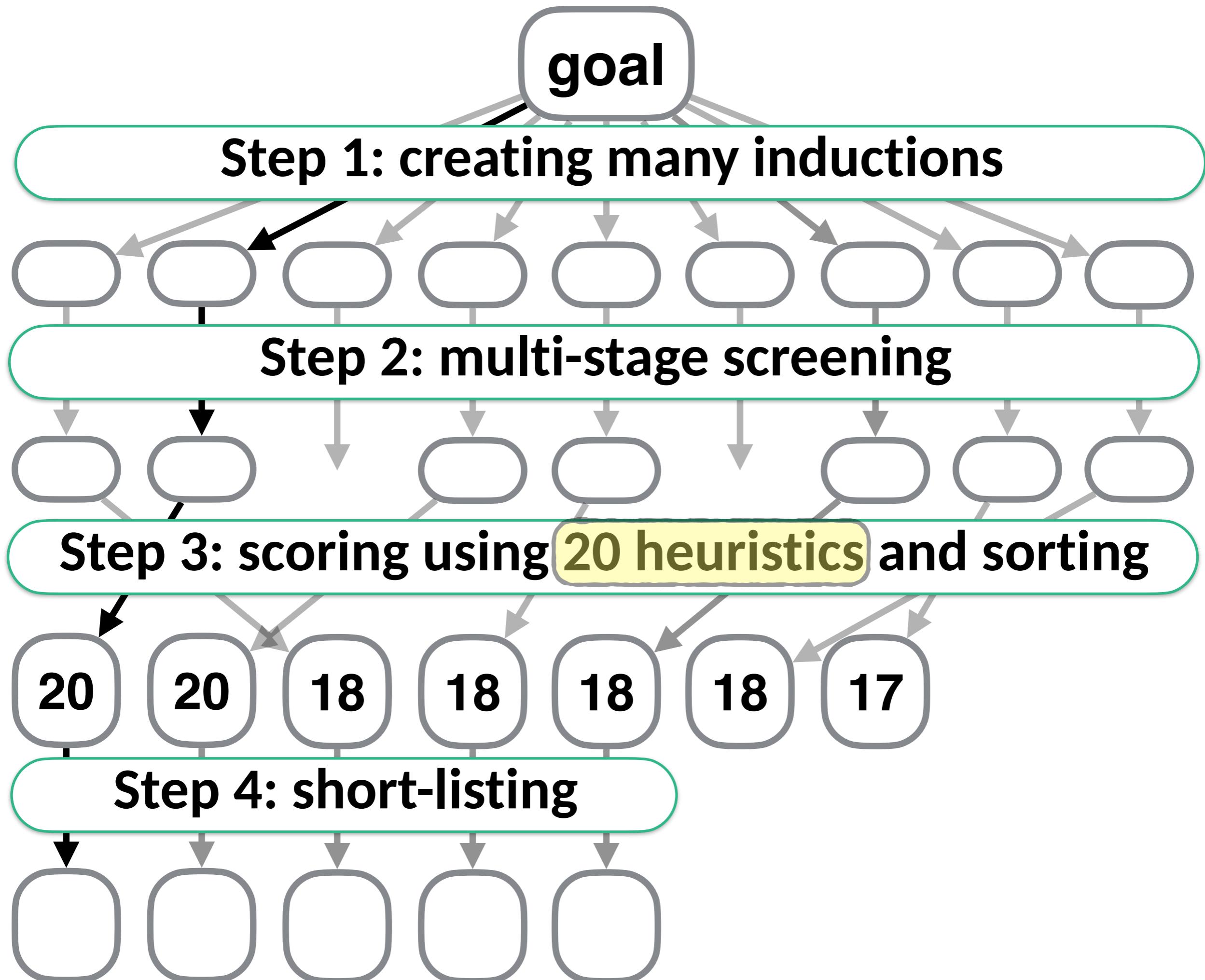
PSL vs sleddaehammer



I have to identify promising applications of proof by induction.

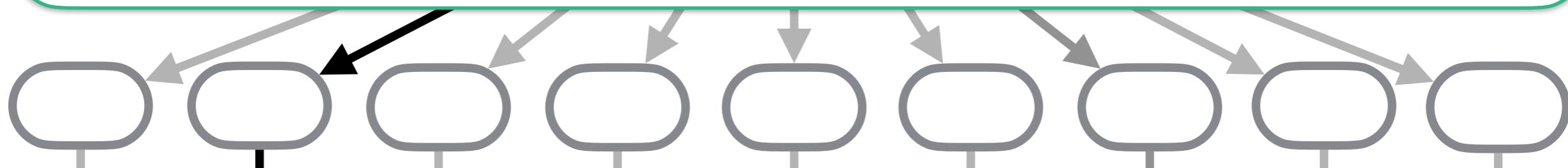
without completing a proof search



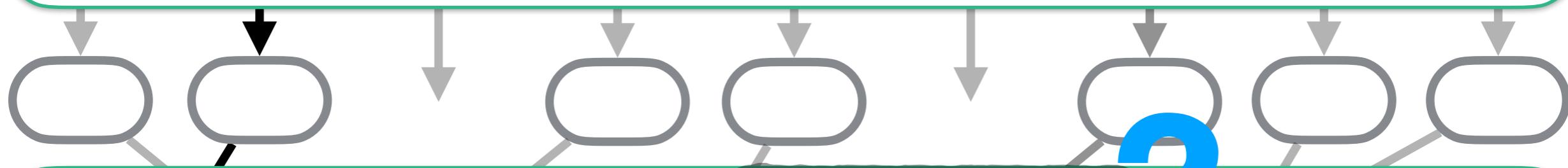


goal

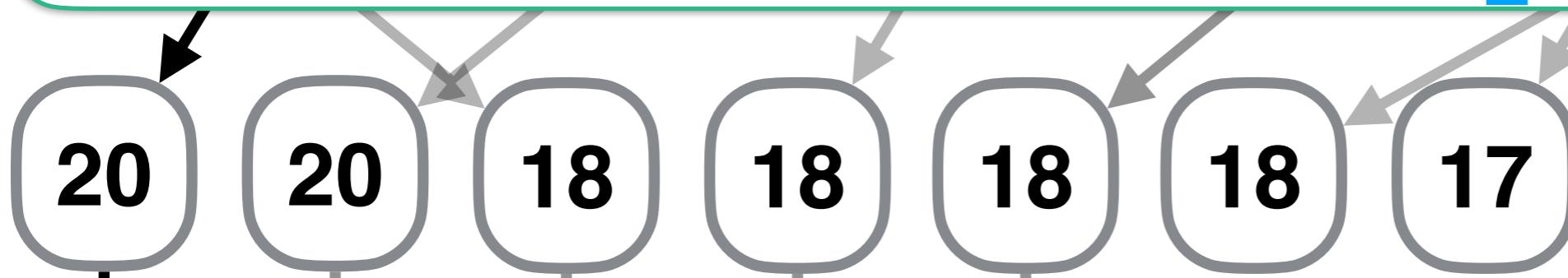
Step 1: creating many inductions



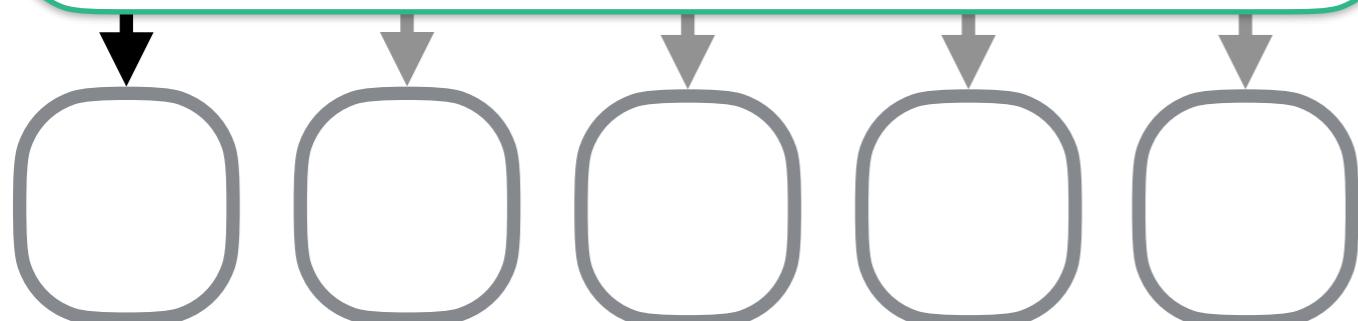
Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting

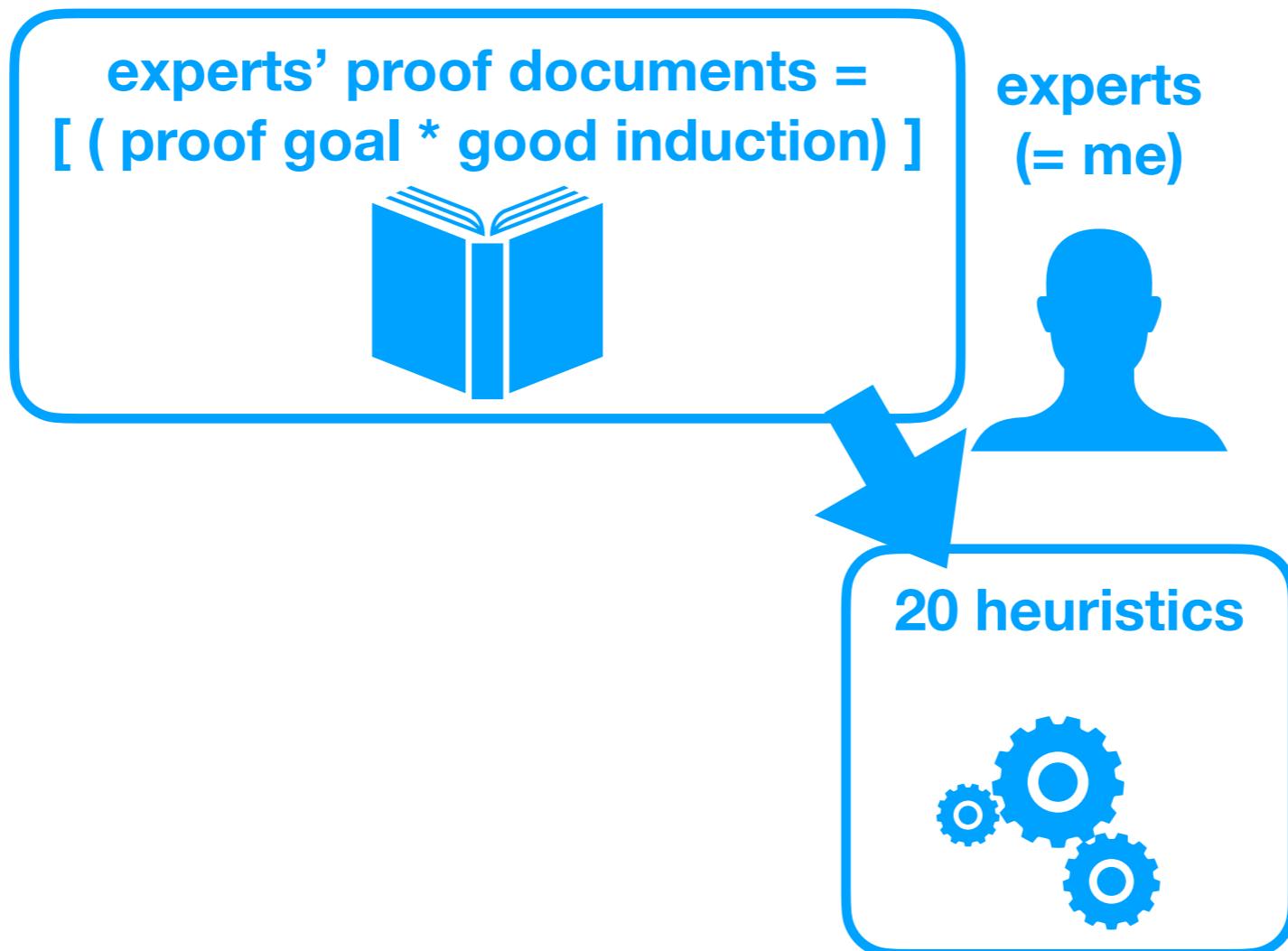


Step 4: short-listing



It is difficult to write induction heuristics.

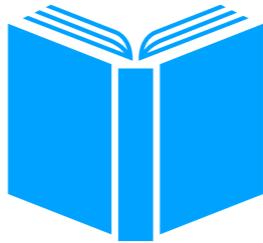
It is difficult to write induction heuristics.



It is difficult to write induction heuristics.

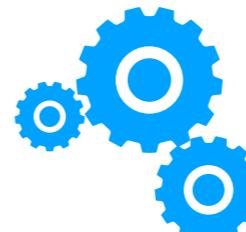
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

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
```

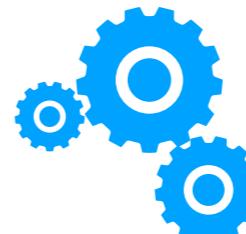
experts' proof documents =
[(proof goal * good induction)]



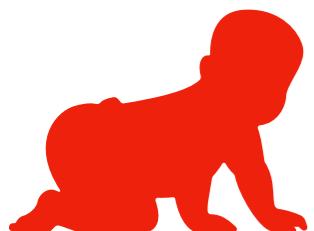
experts
(= me)



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
```

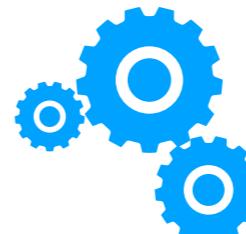
experts' proof documents =
[(proof goal * good induction)]



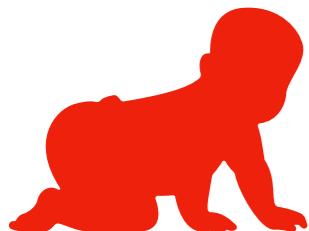
experts
(= me)



20 heuristics



new users



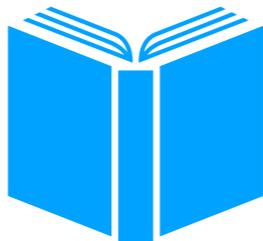
```
lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
```

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
```

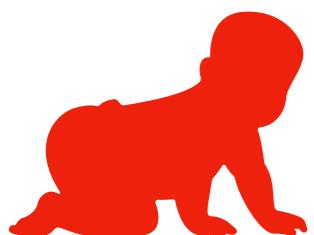
experts' proof documents =
[(proof goal * good induction)]



experts
(= me)

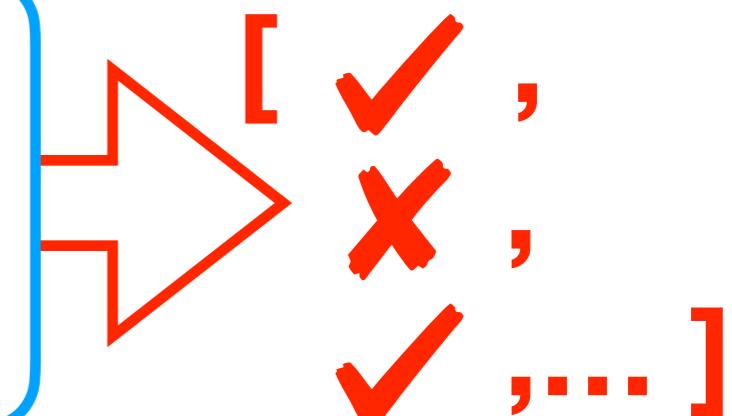
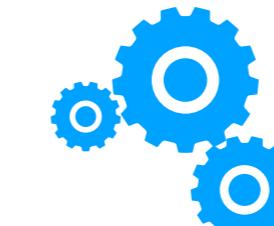


new users



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

20 heuristics



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

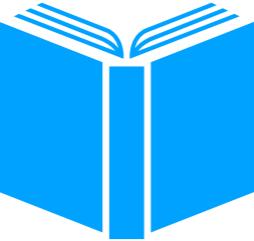
new proof goal consisting of new constants and variables of new types!

It is difficult to write induction heuristics.

① blue = what I can see before releasing smart_induct

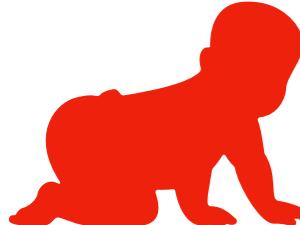
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

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

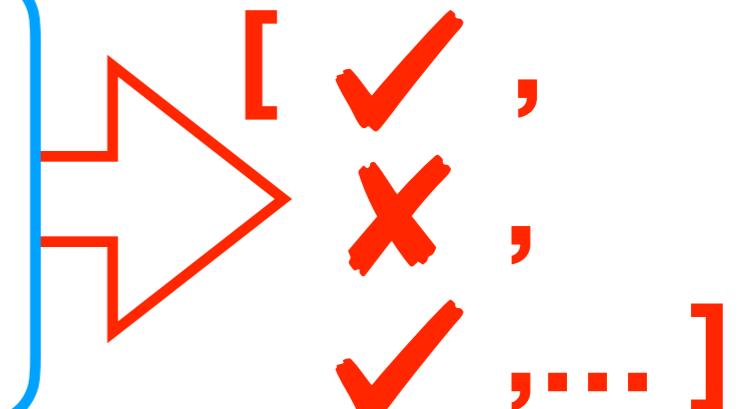
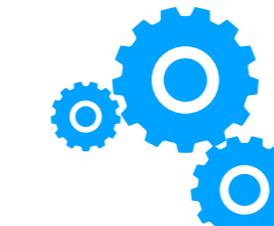


experts
(= me)



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

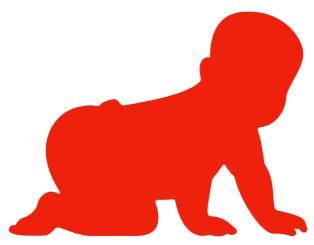
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

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

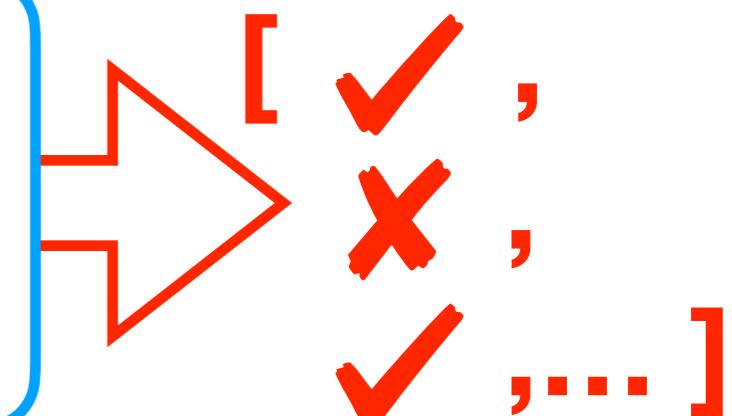
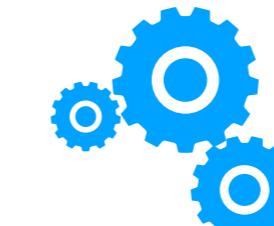


experts
(= me)



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!

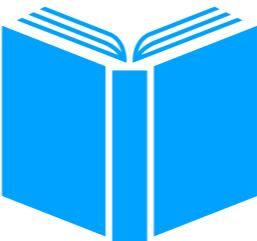
② red = what will be developed after releasing smart_induct

It is difficult to write induction heuristics.

① blue = what I can see before releasing smart_induct

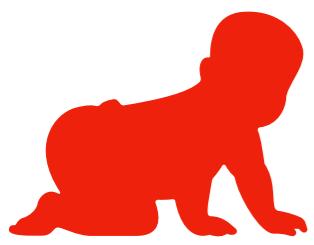
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

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

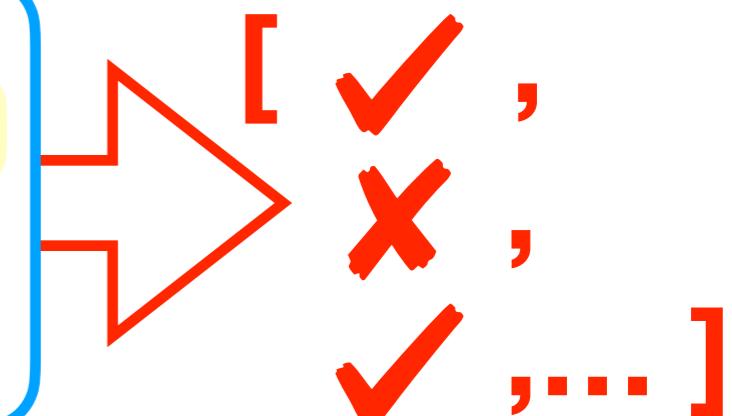
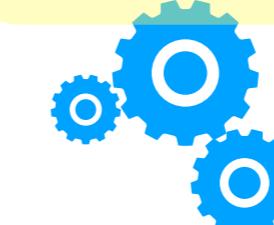


experts
(= me)



new users

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

20 heuristics
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!

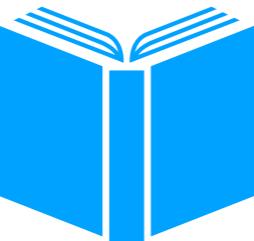
② red = what will be developed after releasing smart_induct

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
```

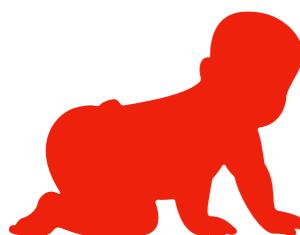
experts' proof documents =
[(proof goal * good induction)]



experts
(= me)

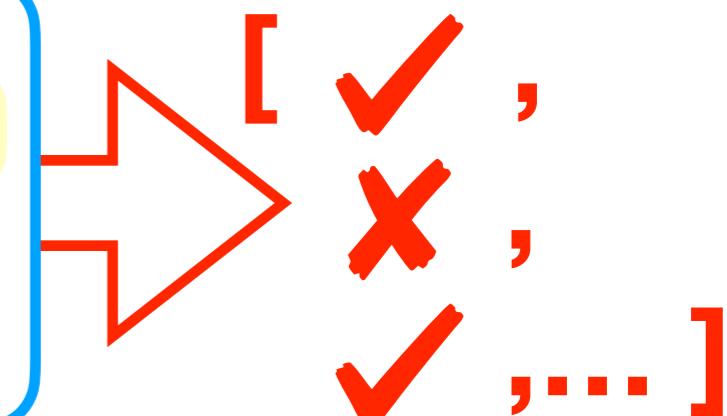
LiFtEr: Logical
Feature Extractor

new users



proof goal candidate induction
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

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
```

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%)

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!

<- 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
```

\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.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]"
```

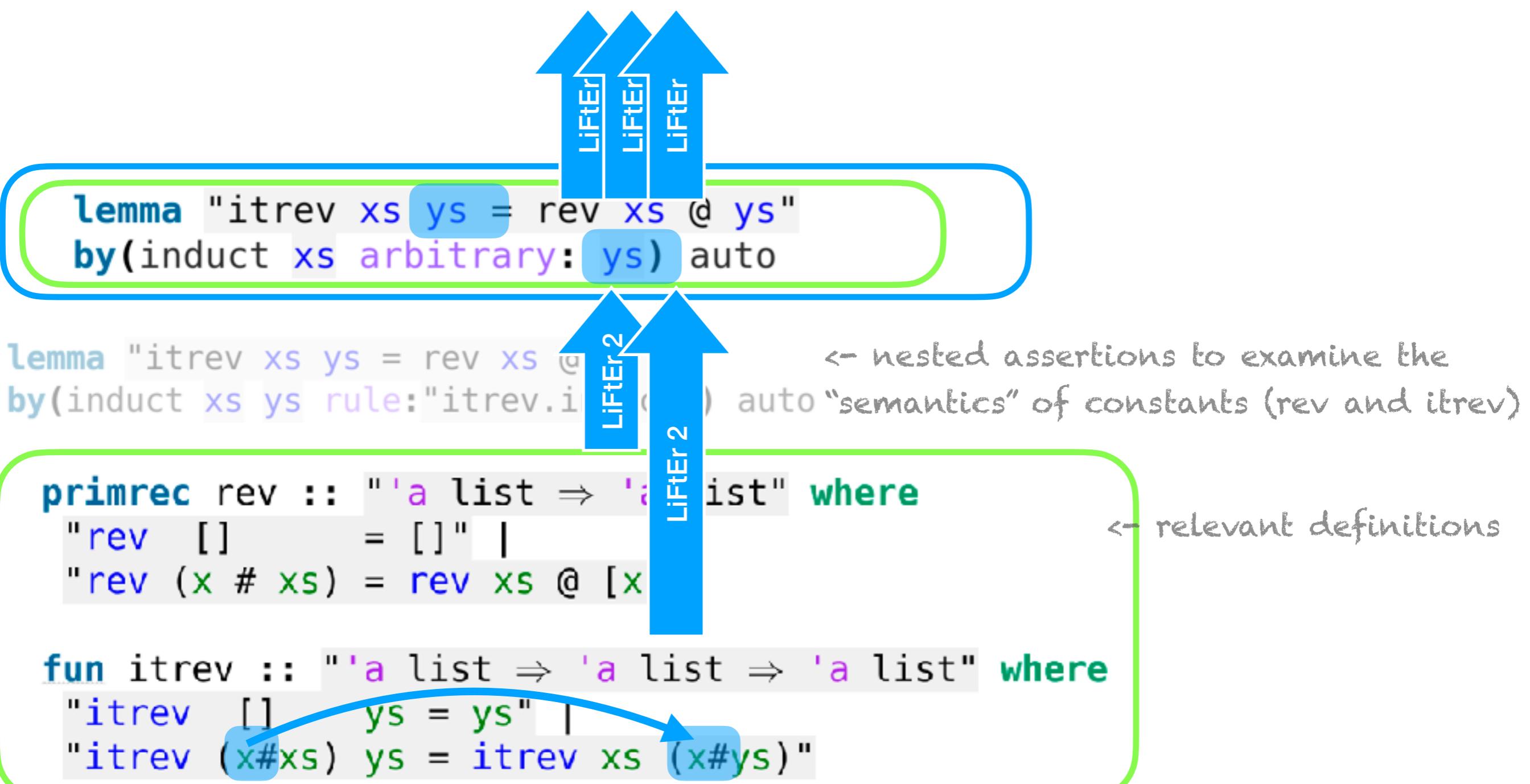


```
fun itrev :: "'a list ⇒ 'a list ⇒ 'a list" where  
"itrev [] ys = ys" |  
"itrev (x#xs) ys = itrev xs (x#ys)"
```

\leftarrow relevant definitions

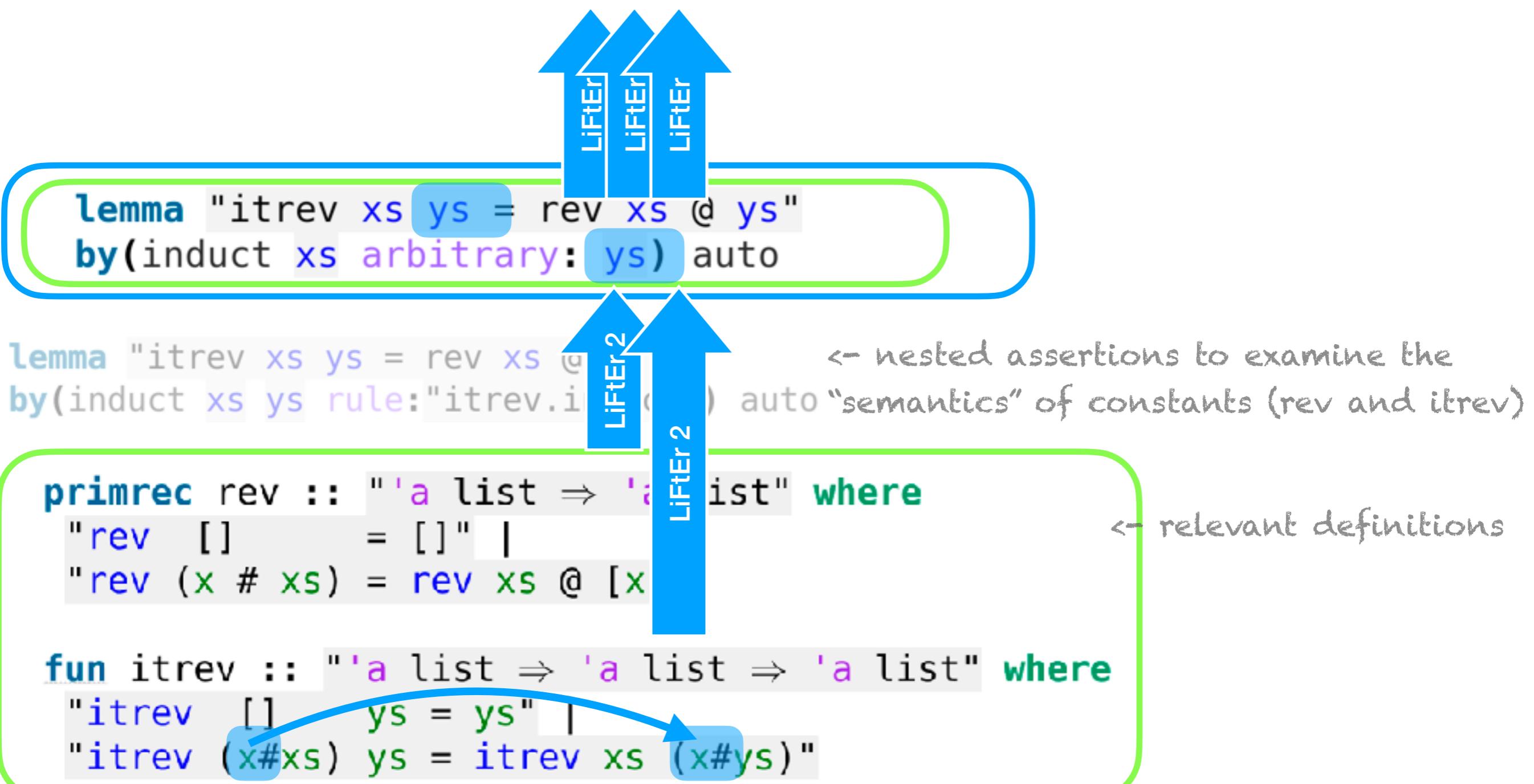
LiFtEr: (proof goal * induction arguments) -> bool

<- simple representation



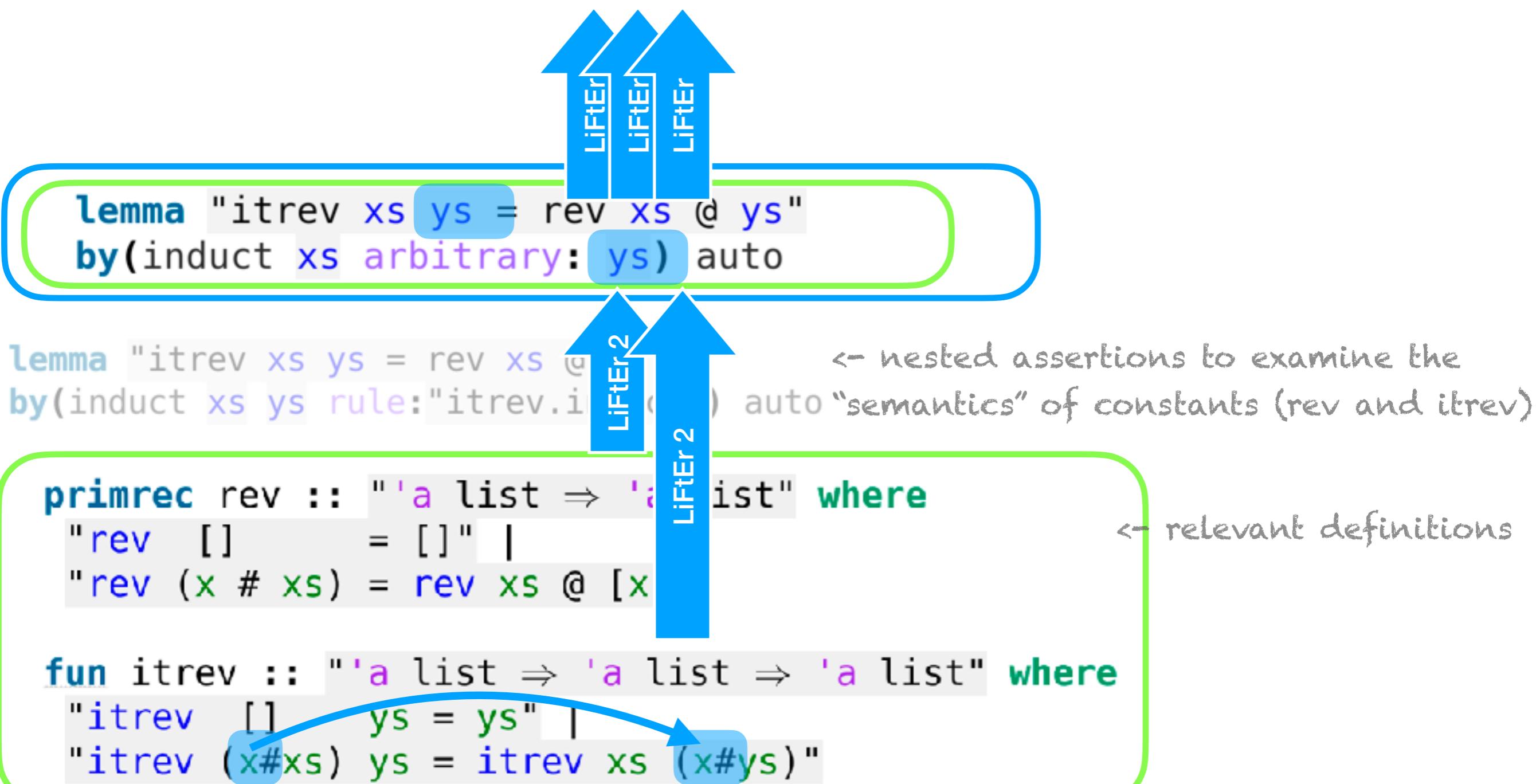
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

<- simple representation



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

coming soon

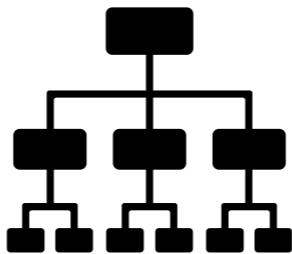


LiFtEr: (proof goal * induction arguments) -> bool

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

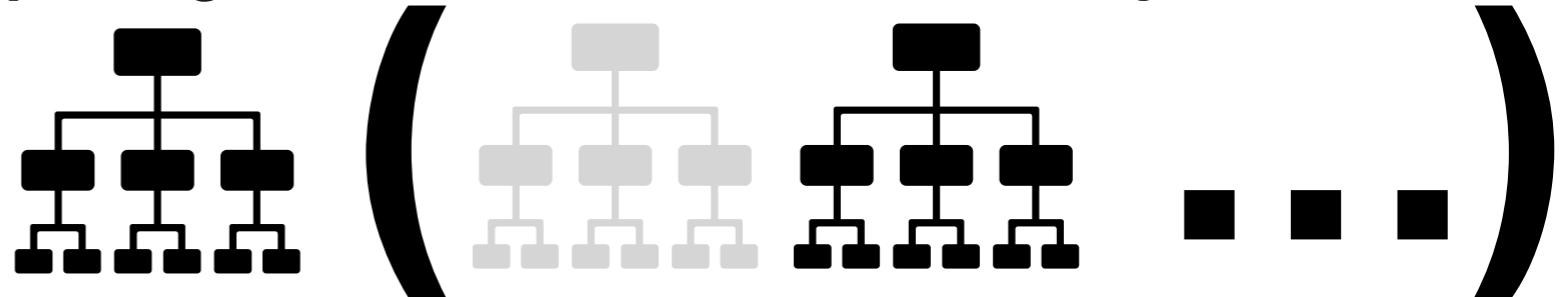
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

proof goal

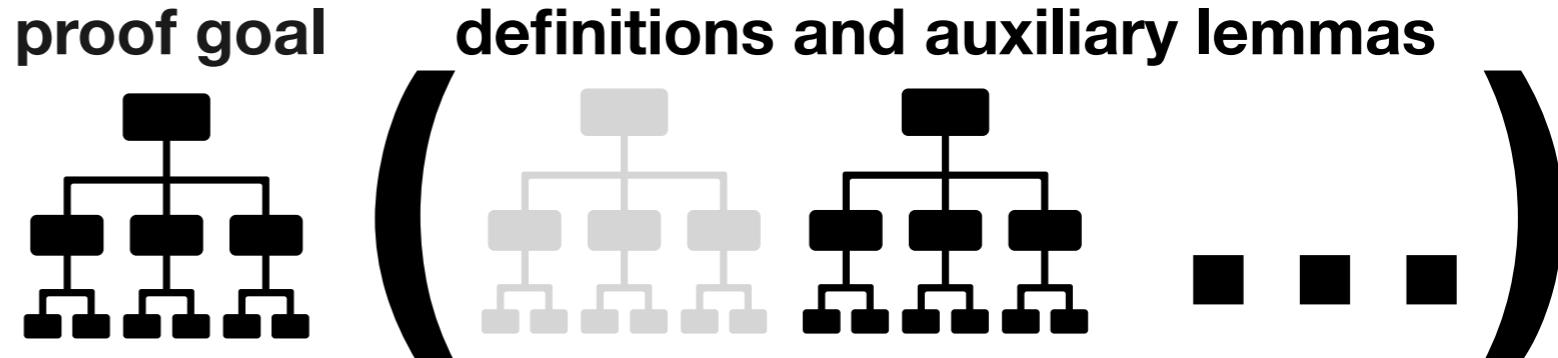
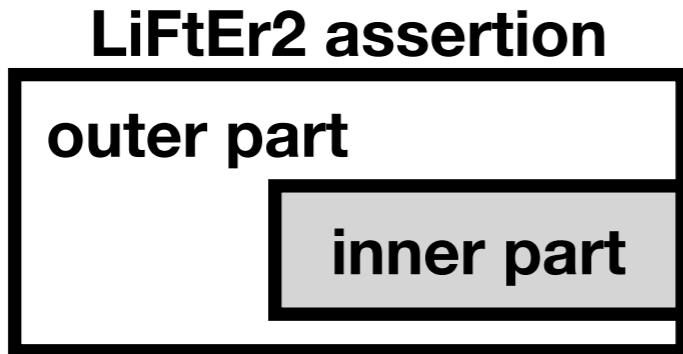


LiFtEr₂ (proof goal * induction arguments)-> bool
* relevant definitions

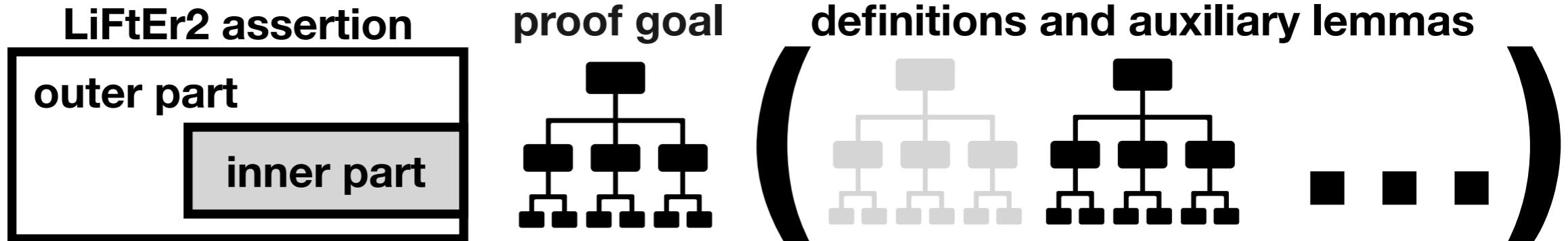
proof goal definitions and auxiliary lemmas



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

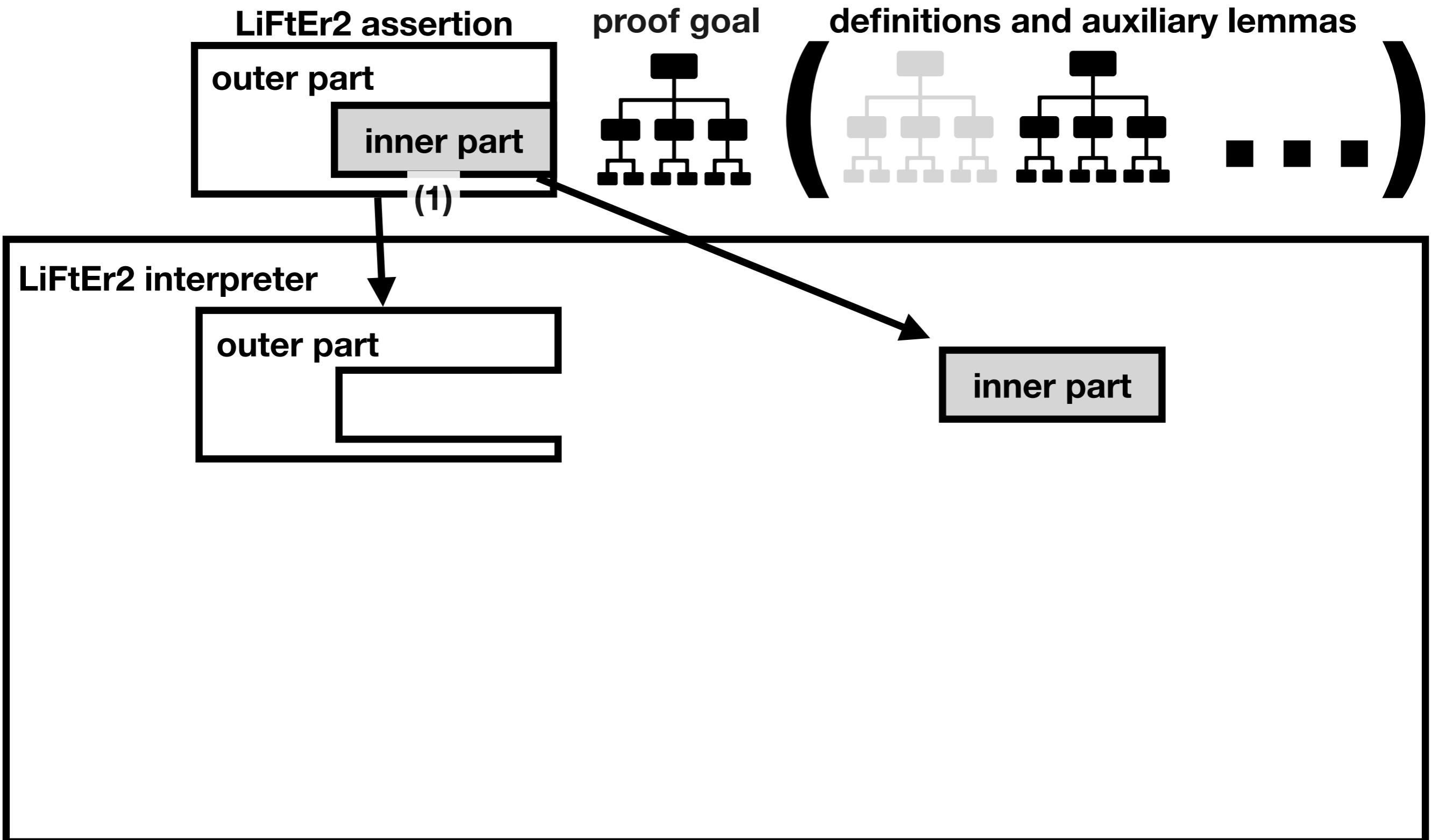


LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

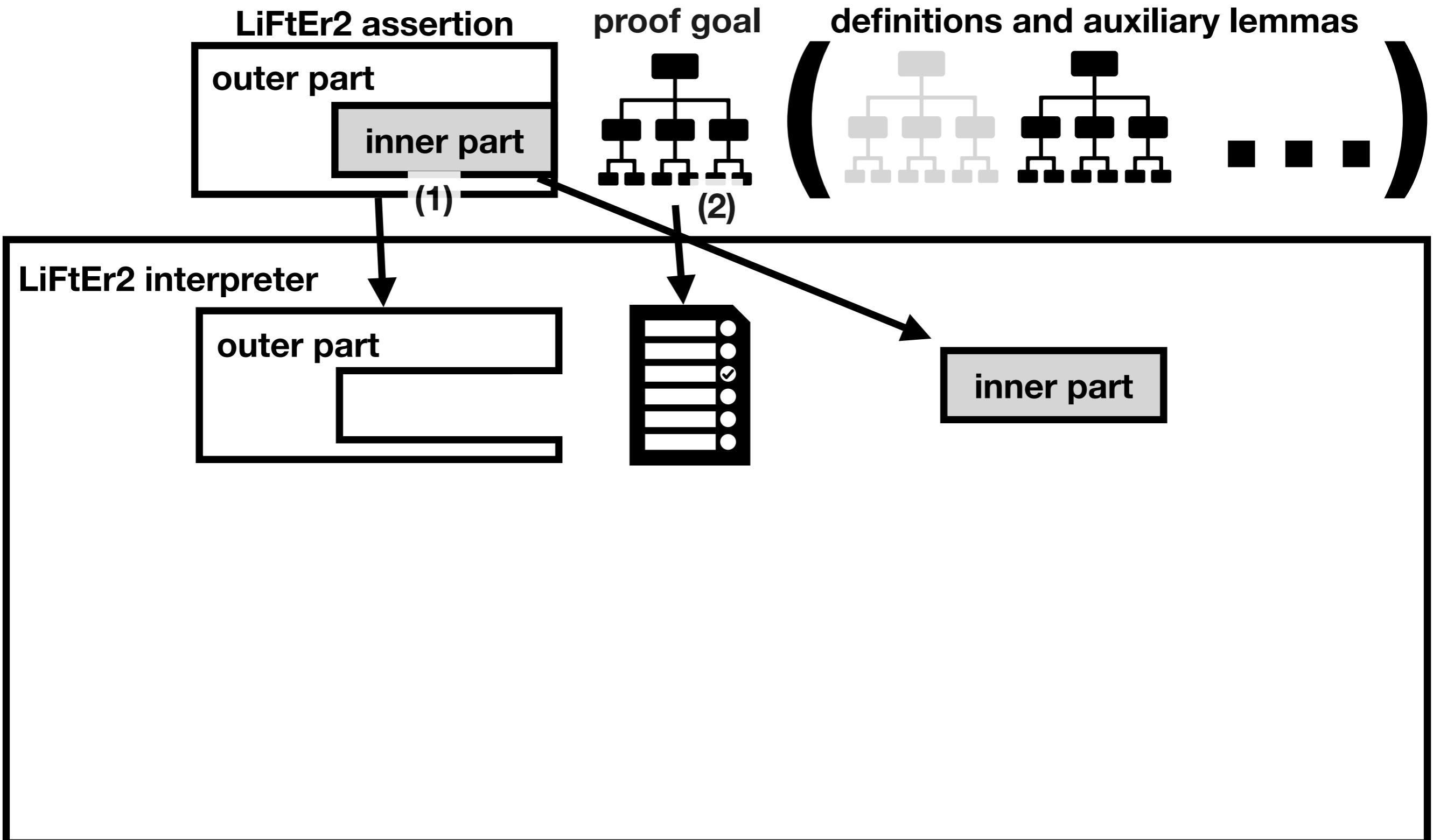


LiFtEr2 interpreter

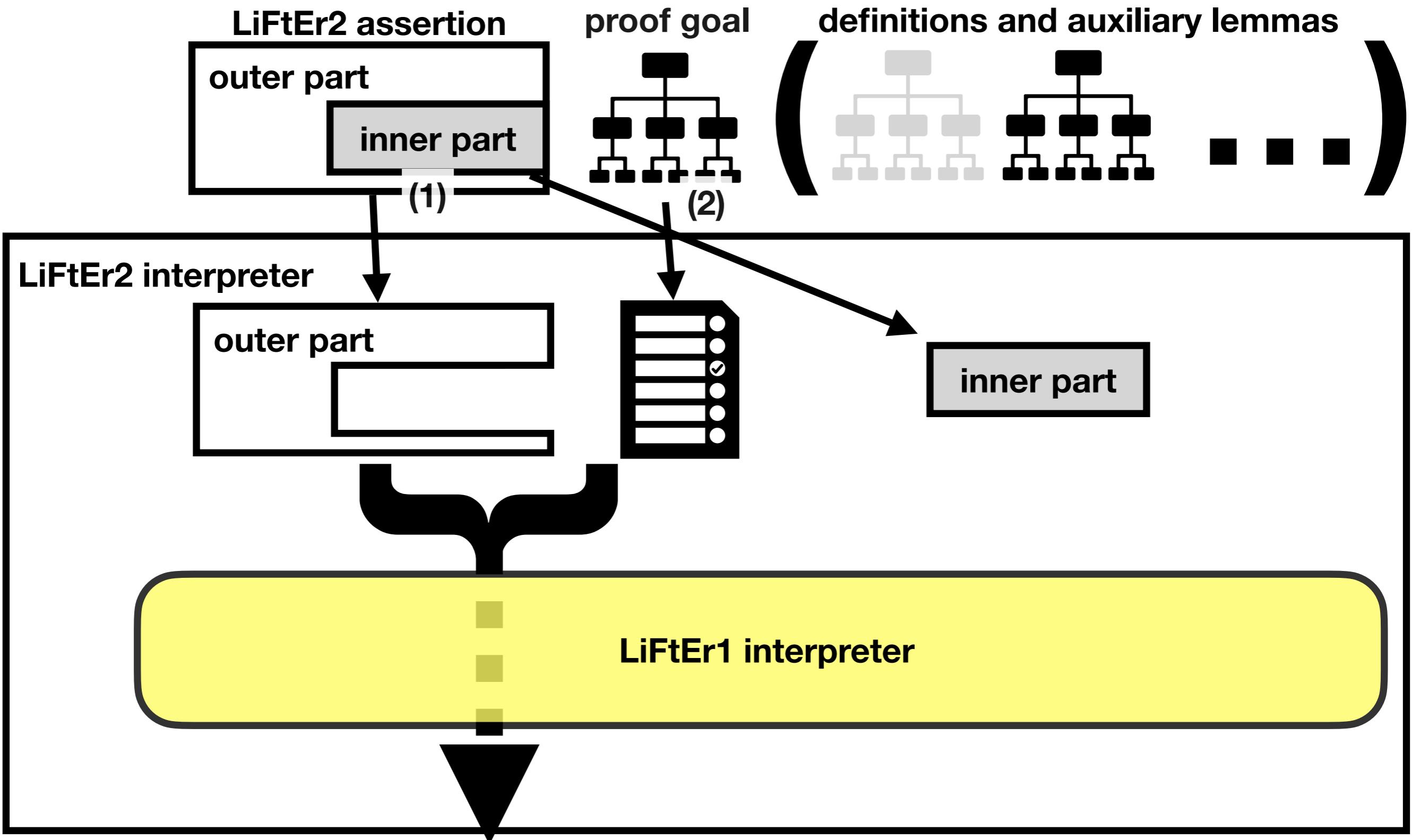
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



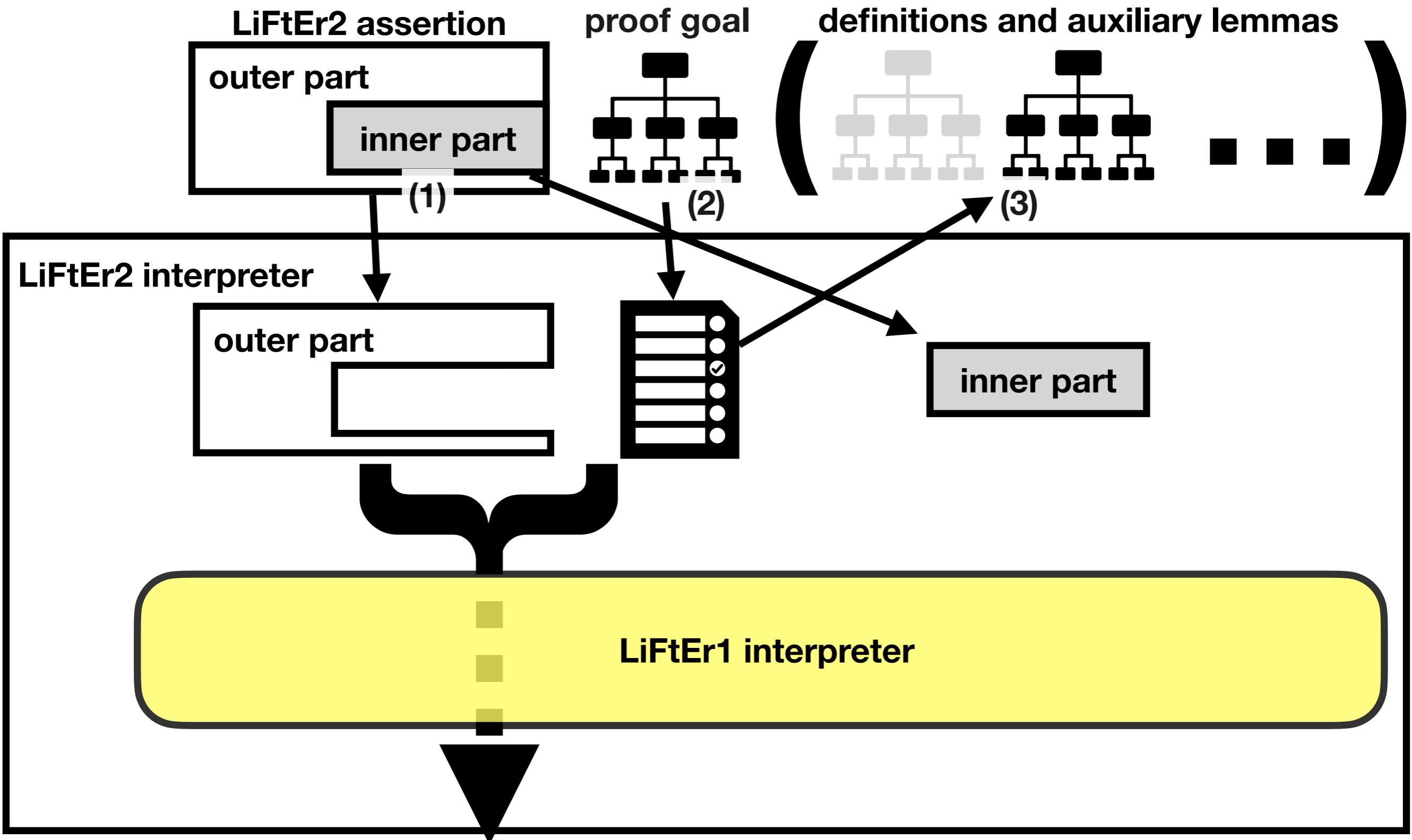
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



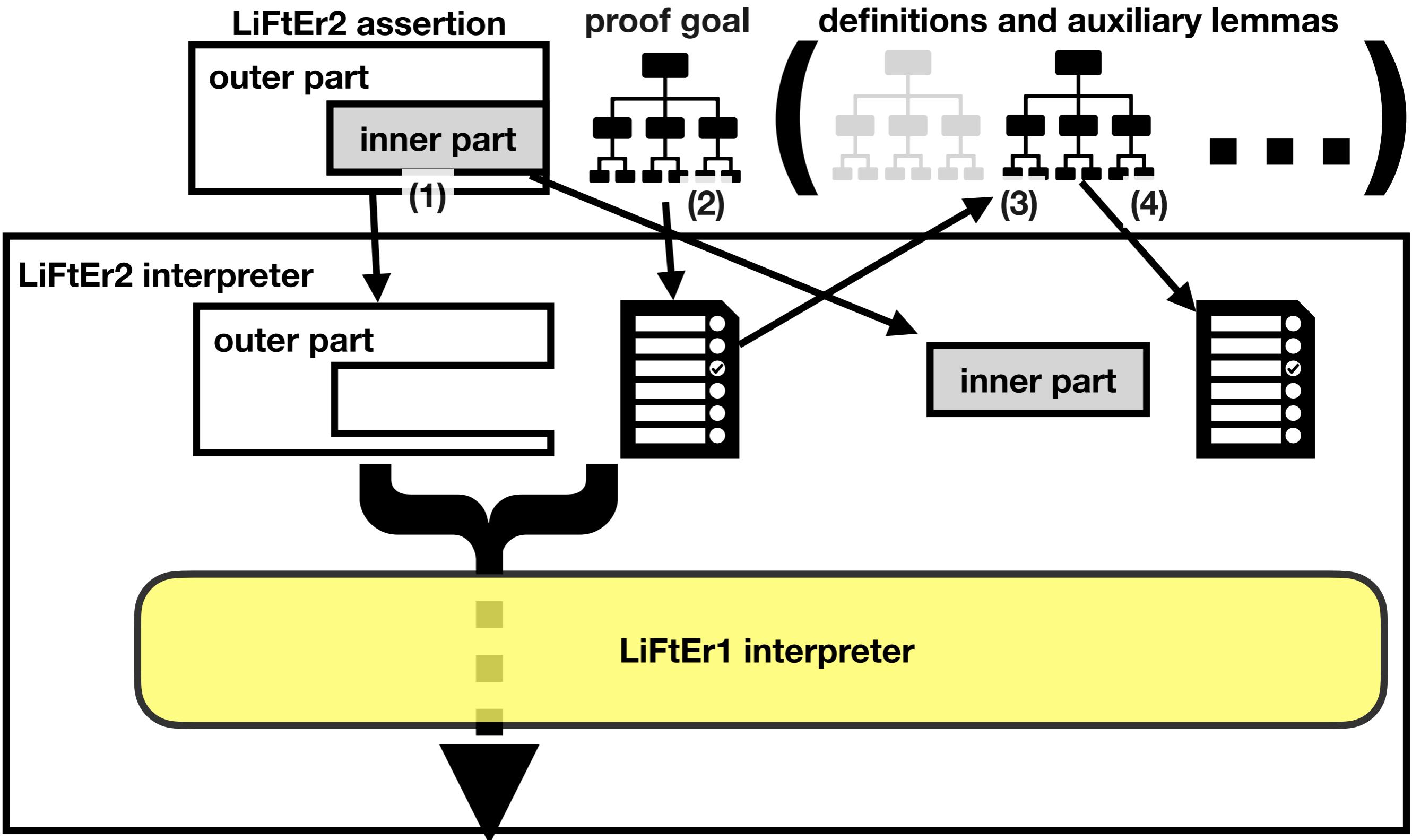
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



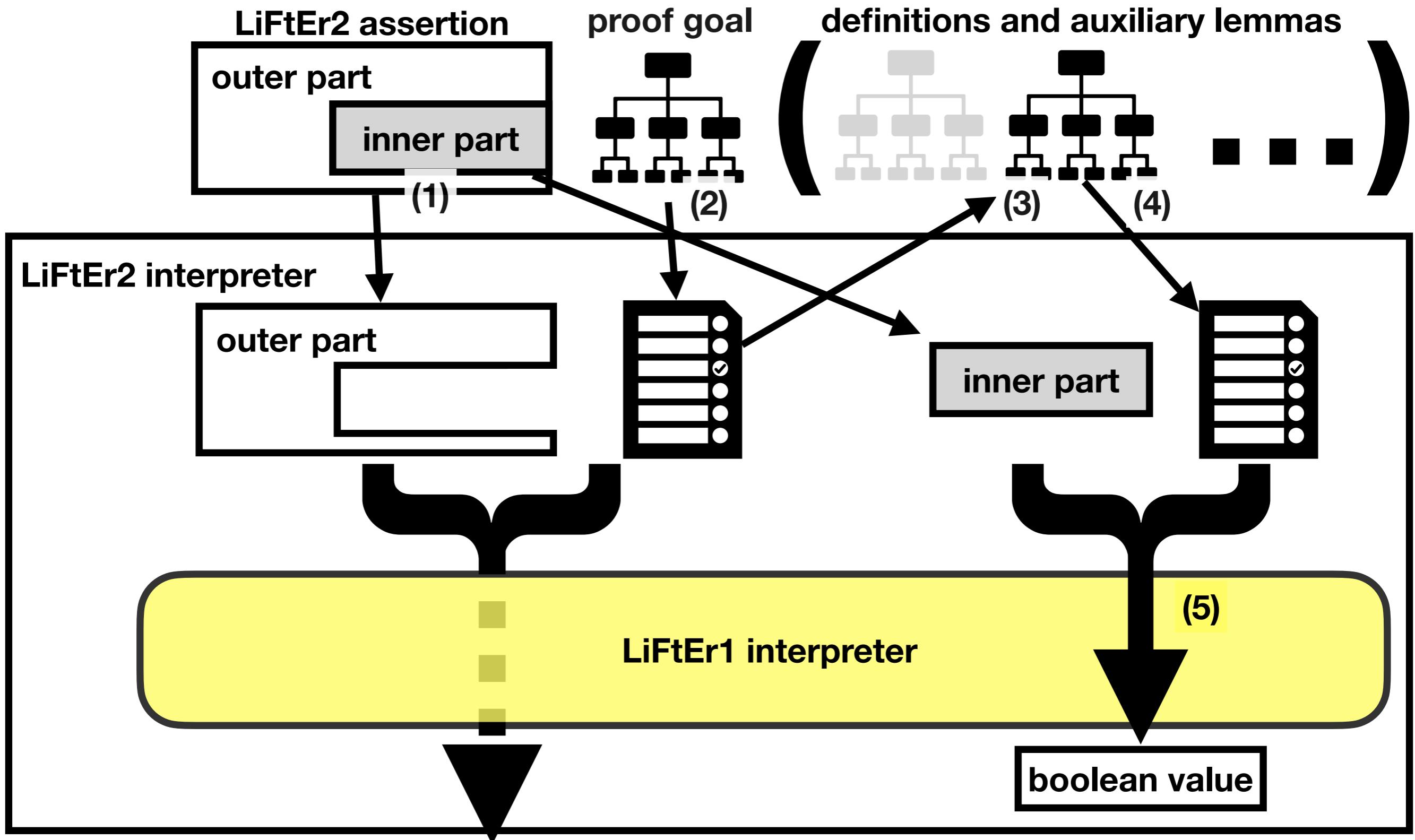
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



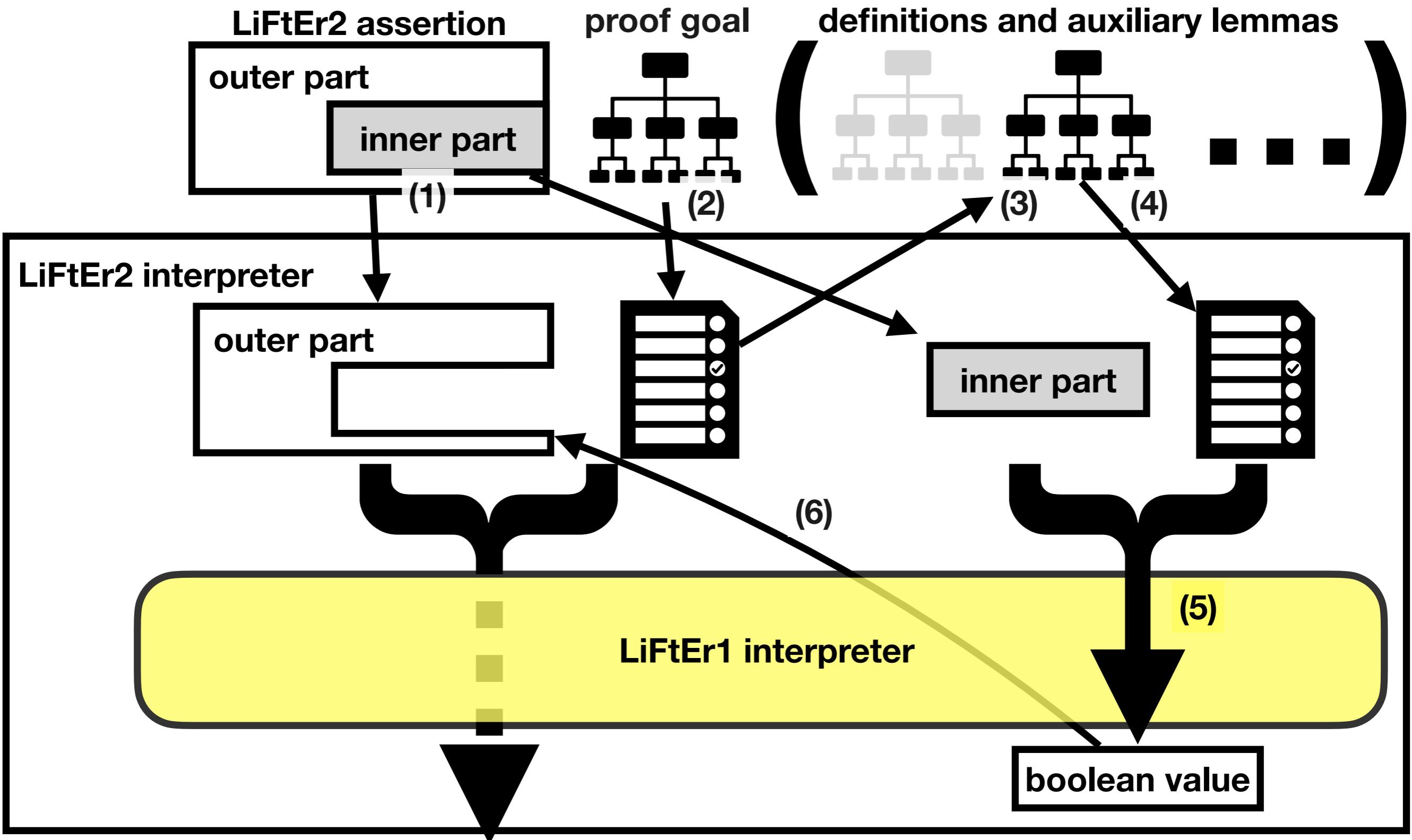
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



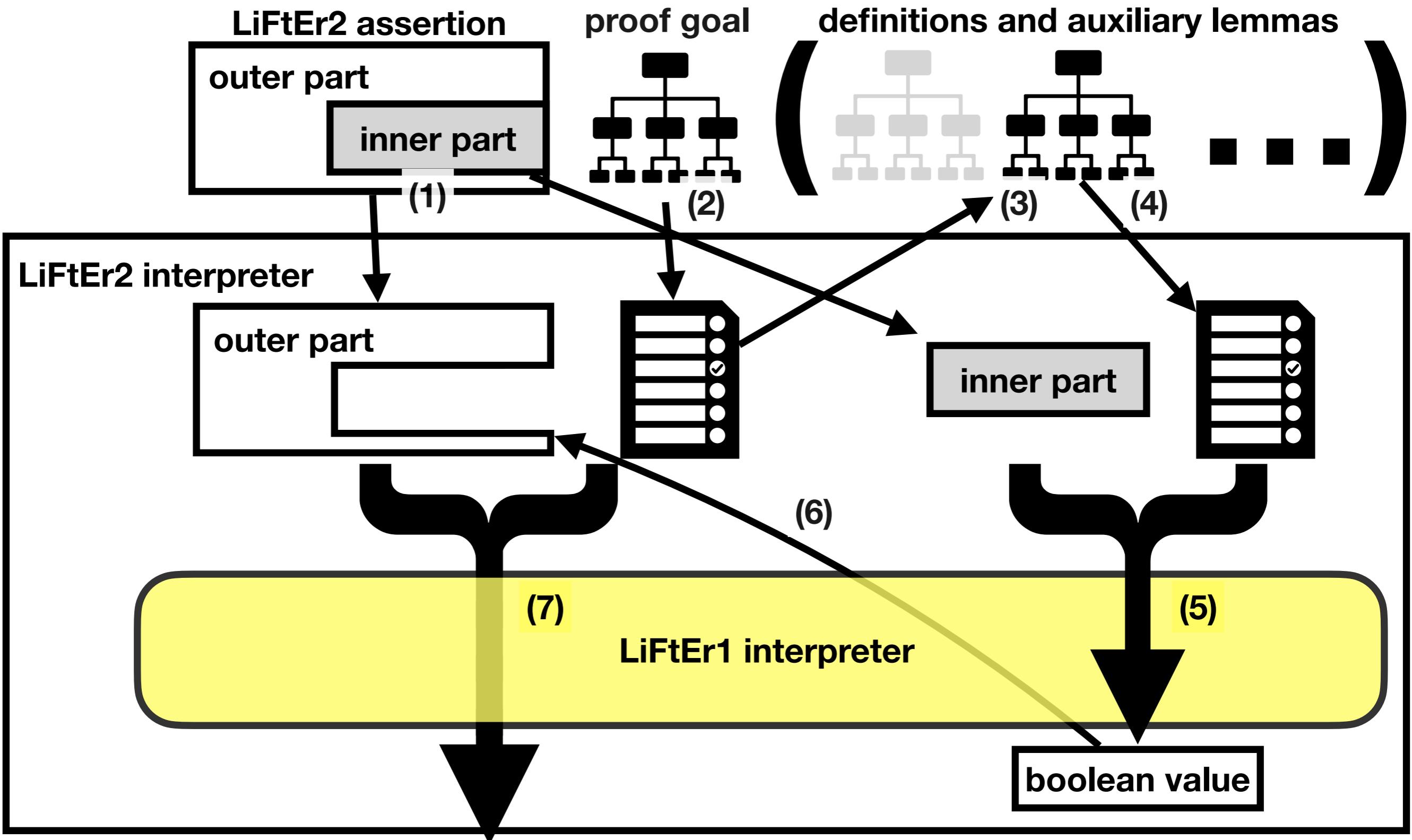
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



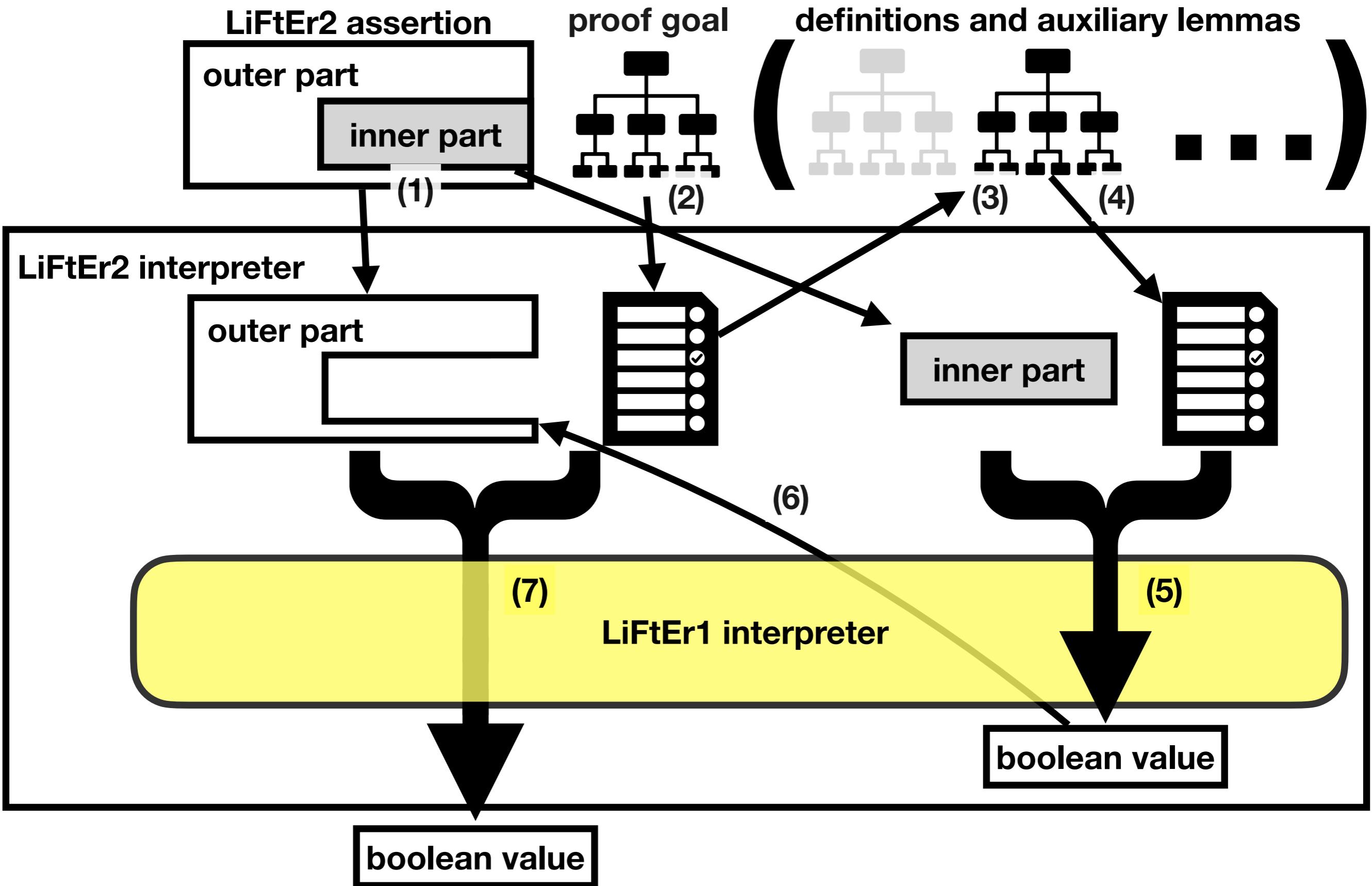
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



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

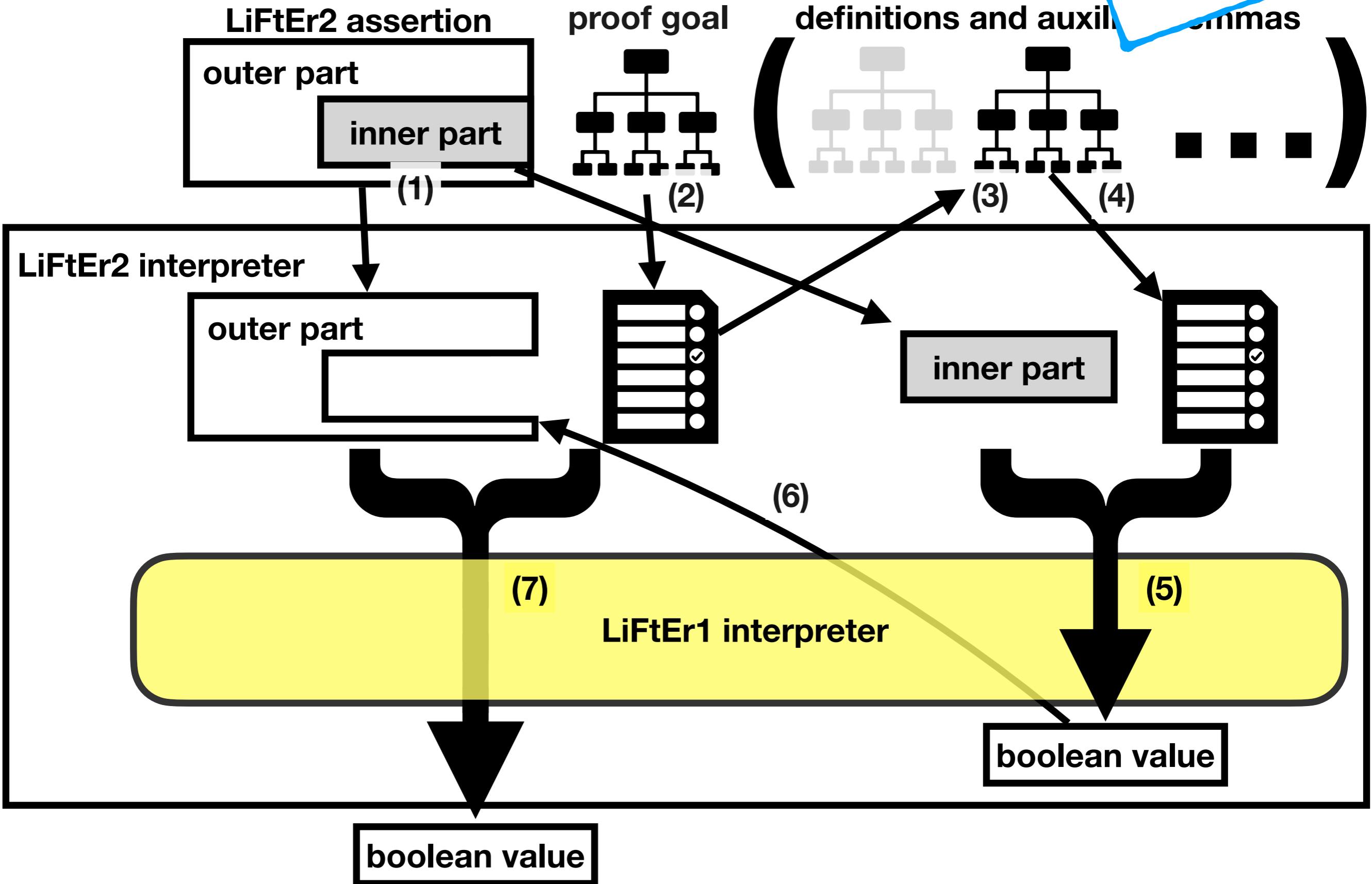


LiFtEr₂ (proof goal * induction arguments) -> bool
 * relevant definitions



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

WIP!



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

