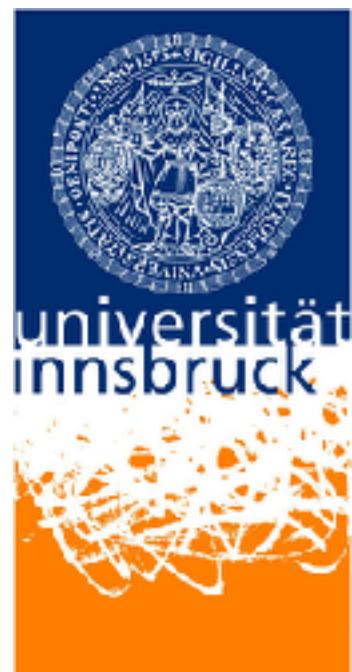


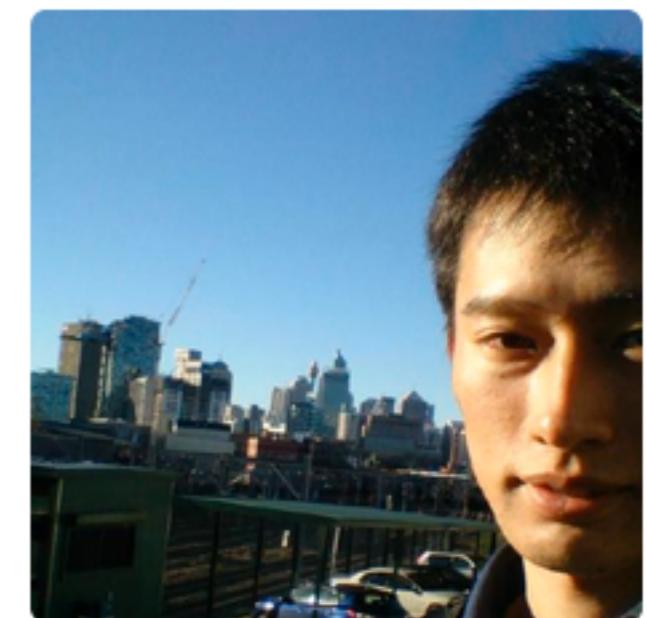
# Automating proof by induction in Isabelle/HOL using DSLs



Yutaka Nagashima



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



Yutaka Ng

# Automating proof by induction in Isabelle/HOL using DSLs



Yutaka Nagashima



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



Yutaka Ng

[https://github.com/data61/PSL/blob/master/slide/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slide/2020_NUS.pdf)

# Background

2013 ~ 2017



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



with Prof. Gerwin Klein

# Background

2013 ~ 2017

Intern &  
Engineer



with Prof. Gerwin Klein

# Background

2013 ~ 2017

Intern &  
Engineer



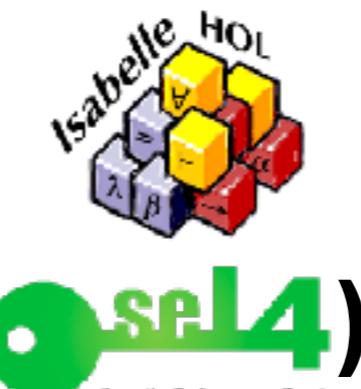
with Prof. Gerwin Klein

PhD in  
AI for theorem proving

# Background

2013 ~ 2017

Intern &  
Engineer



with Prof. Gerwin Klein

PhD in  
AI for theorem proving



2017 ~ 2018



with Prof. Cezary Kaliszyk

# Background

2013 ~ 2017

Intern &  
Engineer



with Prof. Gerwin Klein

PhD in  
AI for theorem proving



with Prof. Cezary Kaliszyk

2018 ~  
2020/21



with Dr. Josef Urban

# Background

2013 ~ 2017

Intern &  
Engineer



**Cogent**

with Prof. Gerwin Klein

PhD in  
AI for theorem proving



with Prof. Cezary Kaliszyk

2018 ~  
2020/21



with Dr. Josef Urban

# Background

2013 ~ 2017

Intern &  
Engineer



Cogent

PhD in  
AI for theorem proving



PSL

2017 ~ 2018

with Prof. Gerwin Klein

2018 ~  
2020/21



with Dr. Josef Urban

# Background

2013 ~ 2017

Intern &  
Engineer



Cogent

PhD in  
AI for theorem proving



PSL

2017 ~ 2018

with Prof. Gerwin Klein

PaMpeR

2018 ~  
2020/21



with Dr. Josef Urban

# Background

2013 ~ 2017

Intern &  
Engineer



Cogent

PhD in  
AI for theorem proving



PSL

2017 ~ 2018

with Prof. Gerwin Klein

PaMpeR

2018 ~  
2020/21



LiFtEr

with Dr. Josef Urban

# Background

2013 ~ 2017

Intern &  
Engineer



**Cogent**

PhD in  
AI for theorem proving



2017 ~ 2018



**PSL**

2018 ~  
2020/21



**LiFtEr**

with Dr. Josef Urban

**smart\_induct**

# Background

2013 ~ 2017

Intern &  
Engineer



**Cogent**

PhD in  
AI for theorem proving



2017 ~ 2018



**PSL**

2018 ~  
2020/21



**LiFtEr**

with Dr. Josef Urban

**smart\_induct**

## PSL: Proof Strategy Language

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

## PSL: Proof Strategy Language

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



DEMO!

# DEMO1

[https://github.com/data61/PSL/blob/master/slide/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slide/2020_NUS.pdf)

The screenshot shows the Isabelle proof assistant interface. 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 is highlighted with a yellow background. The status bar at the bottom shows:

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

The bottom navigation bar includes tabs for Output, Query, Sledgehammer, and Symbols.

# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.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 queries its value.

```
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 value query. Below the code editor, the output window shows the result:

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

The interface includes various toolbars, a vertical scroll bar, and status bars at the bottom showing file statistics and system information.

# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.pdf)

The screenshot shows the Isabelle proof assistant interface with a 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)" □
18
19
20
21
```

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

# DEMO1

[https://github.com/data61/PSL/blob/master/slide/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slide/2020_NUS.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 `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
11 Lemma
12   "map f (sep x xs) = sep (f x) (map f xs)"
13
14
15
16
17
18
19
20
21
```

The `strategy` line is highlighted with a yellow background. The interface includes a toolbar at the top, a vertical scroll bar on the right, and a status bar at the bottom.

# DEMO1

[https://github.com/data61/PSL/blob/master/slide/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slide/2020_NUS.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 `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
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
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
99
```

The lemma `map_f_sep` 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 memory usage: 17.44 (361/1084) and (isabelle.isabelle,UTF-8-Isabelle) nmr o U.. 242/535MB 1:05 AM.

# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.pdf)

The screenshot shows the Isabelle proof assistant interface. The main window displays a theory file named Example.thy. The code includes a function sep, a value declaration, a strategy, a lemma, and a find\_proof command. The find\_proof command is highlighted with a red rectangle.

```
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
15
16
17
18
19
20
21
```

Number of lines of commands: 3

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

# DEMO1

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

The screenshot shows the Isabelle proof assistant interface with a theory file named `Example.thy`. The code defines a function `sep` and a value, and includes a strategy and a lemma with its proof.

```
File Browser Documentation Sidekick State Theories

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

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

The proof script starts with `find_proof DInd`, followed by `apply (induct xs rule: Example.sep.induct)`, `apply auto`, and `done`. The proof is completed with `proof (prove)`, `goal:`, and `No subgoals!`.

At the bottom, the status bar shows: 20.11 (433/1147), (isabelle,isabelle,UTF-8-Isabelle) in miro U., 255/535MB 1:05 AM.

# DEMO1

[https://github.com/data61/PSL/blob/master/slide/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slide/2020_NUS.pdf)

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

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

Below the script, the proof state is shown:

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

At the bottom right, a blue speech bubble contains the text "What happened?".

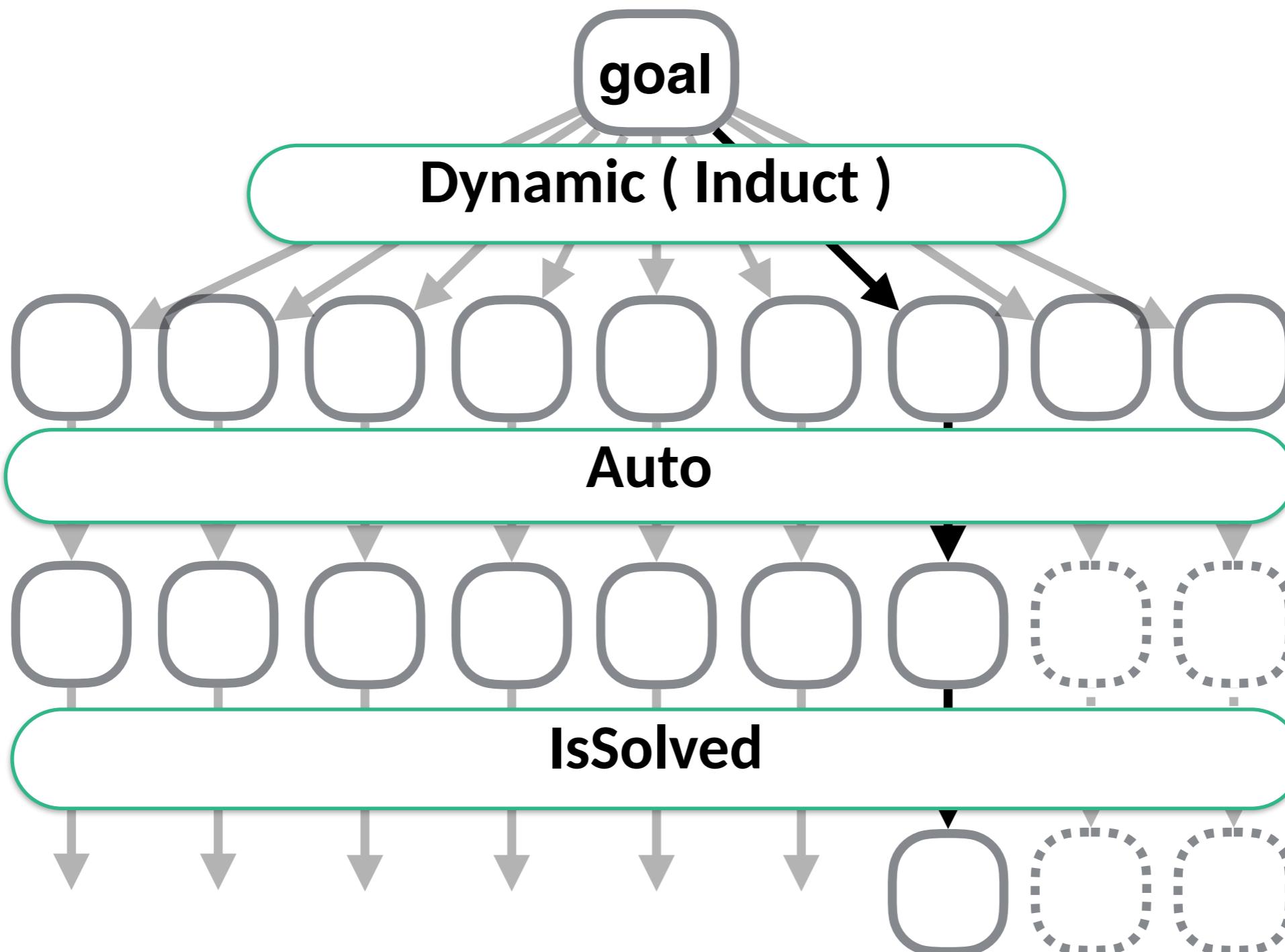
What happened?

# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.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]*)
```

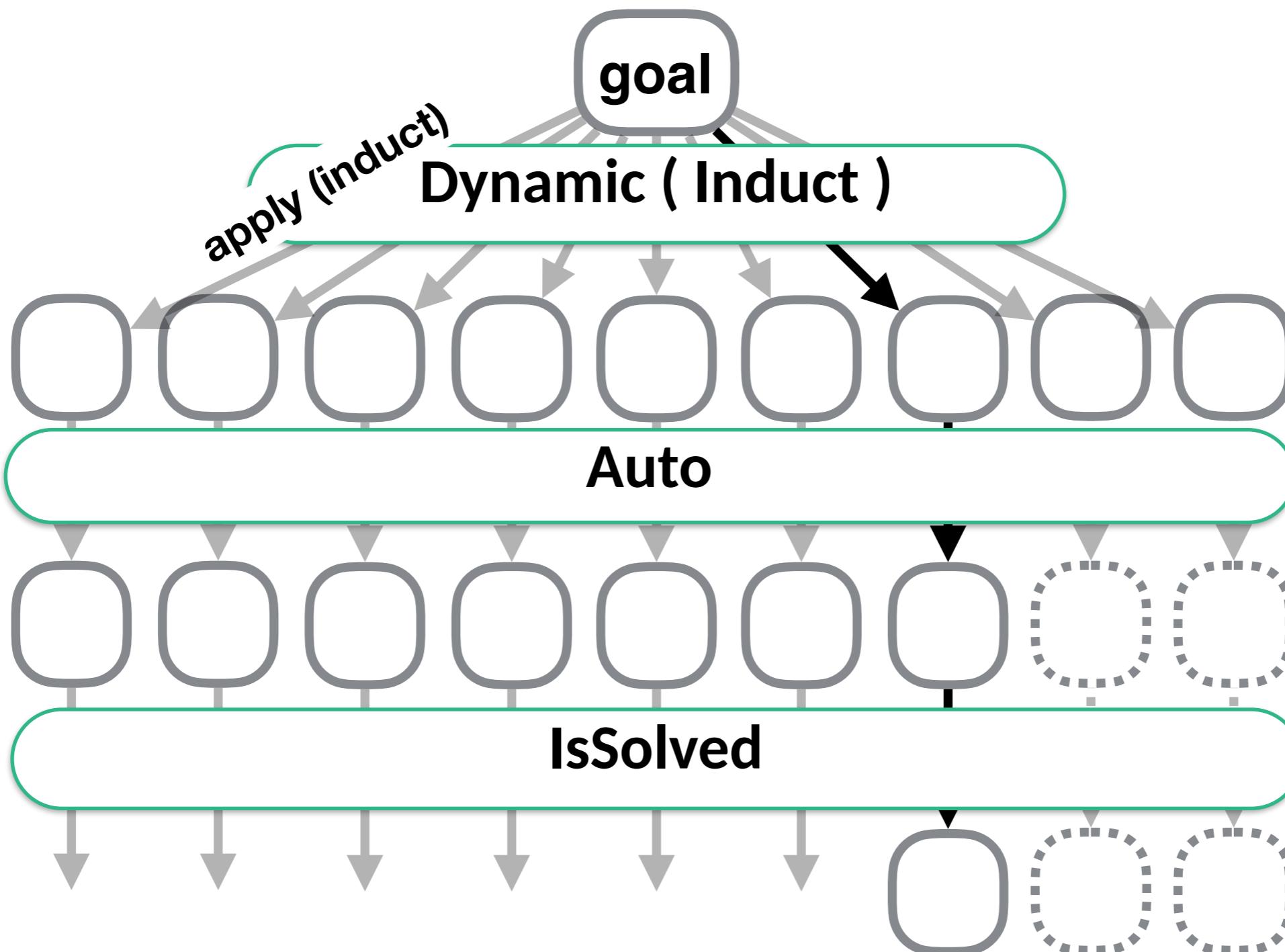


# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.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]*)
```

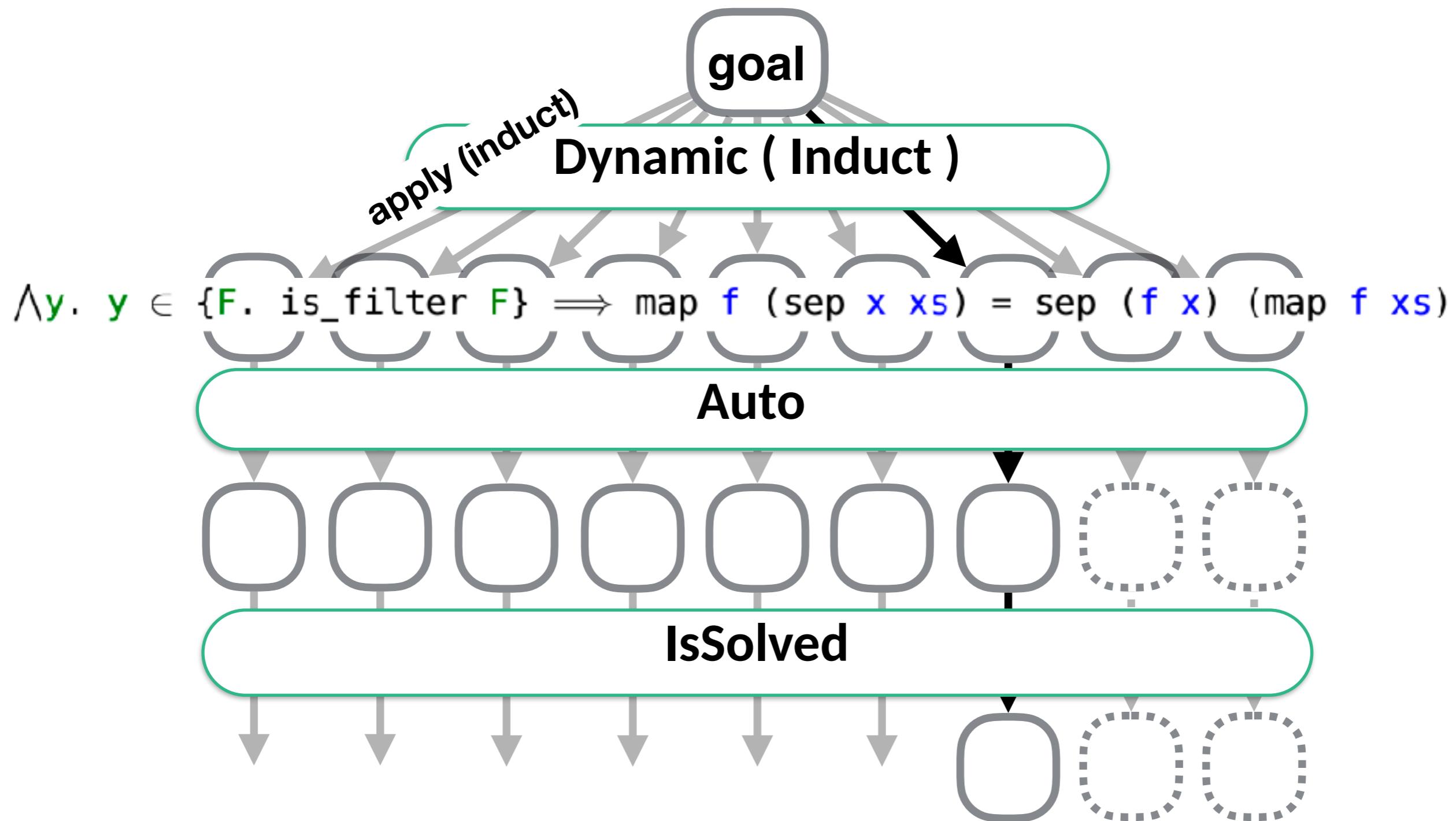


# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.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]*)
```

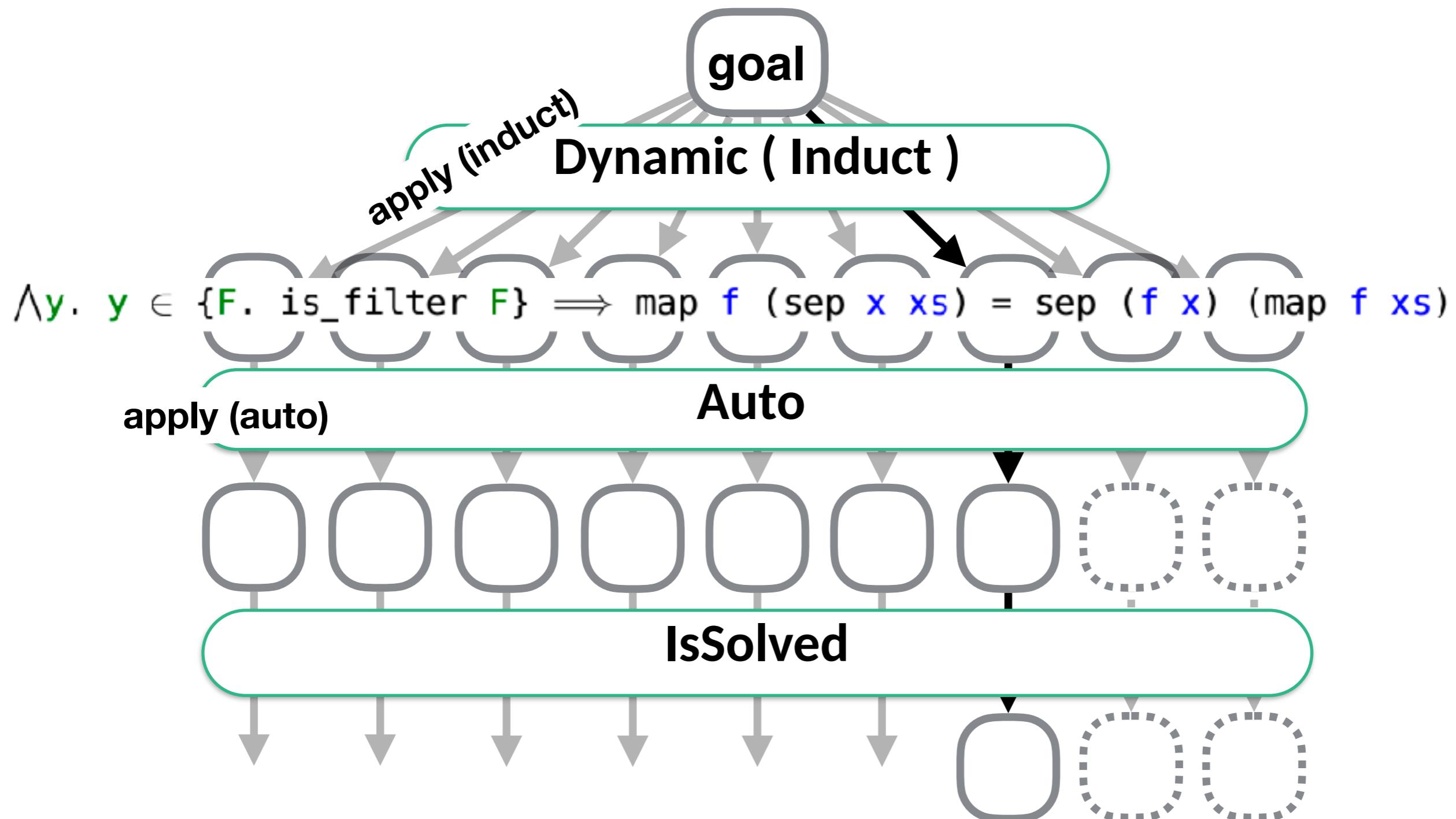


# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.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]*)
```

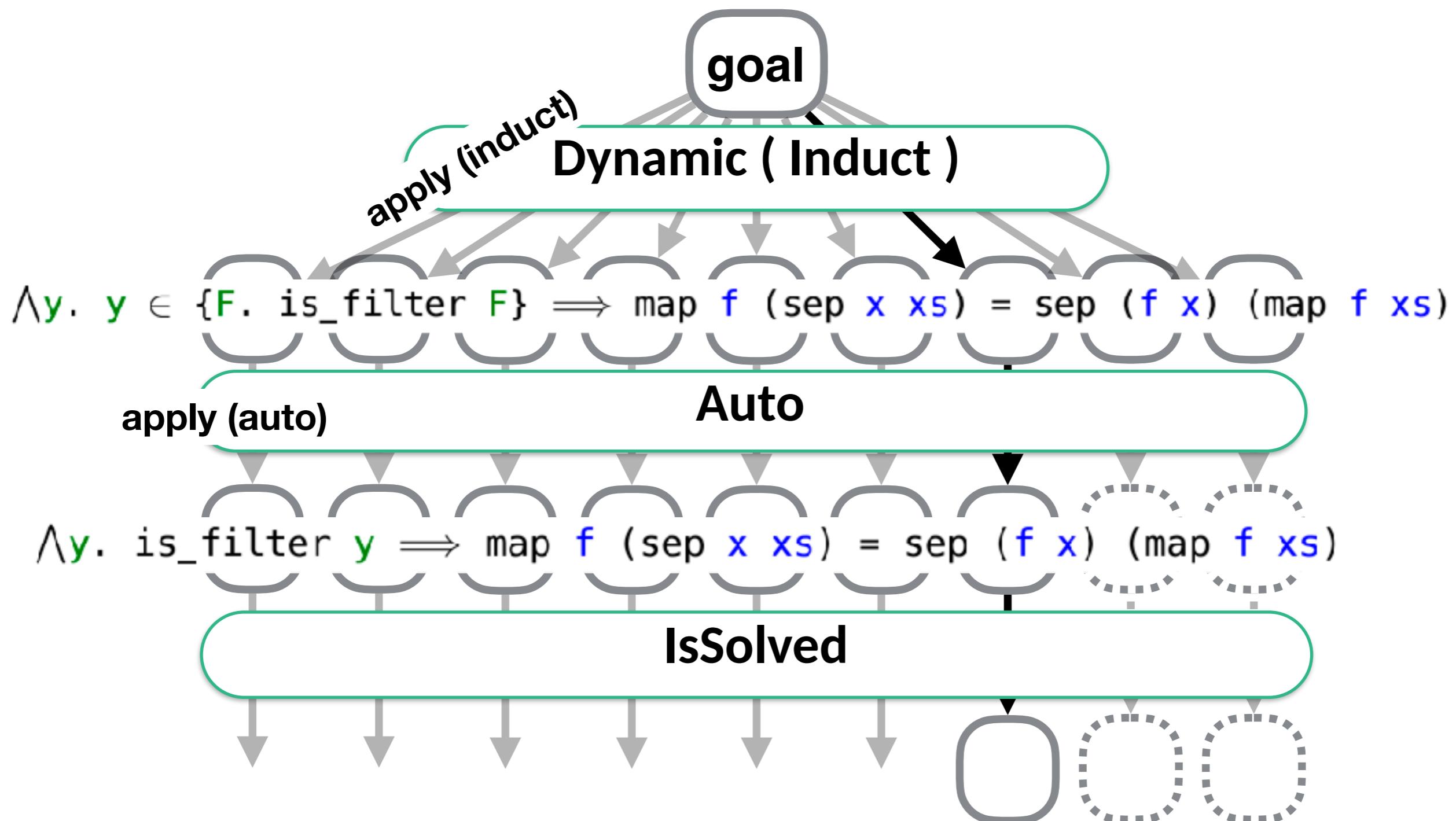


# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.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]*)
```

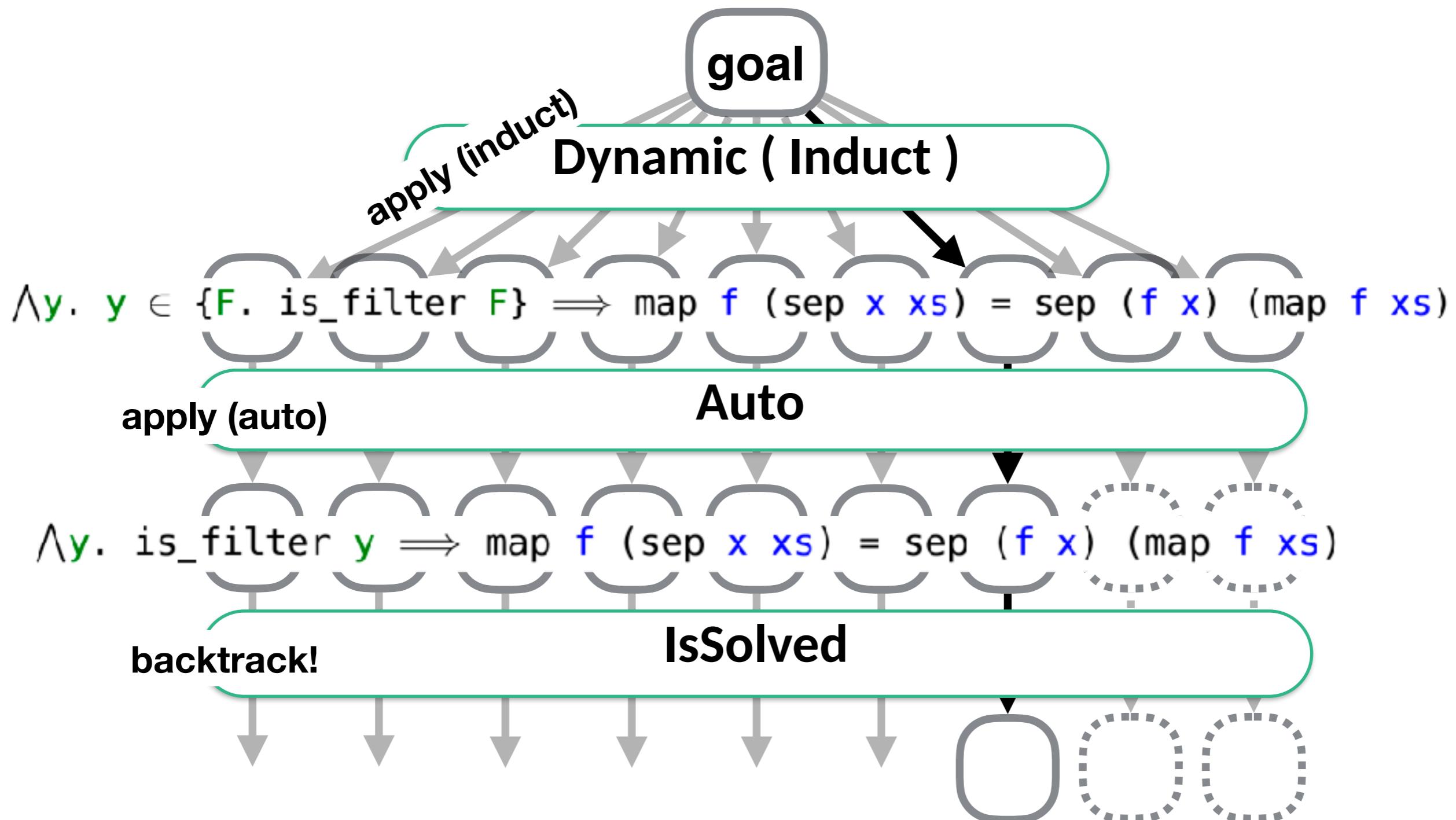


# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.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]*)
```

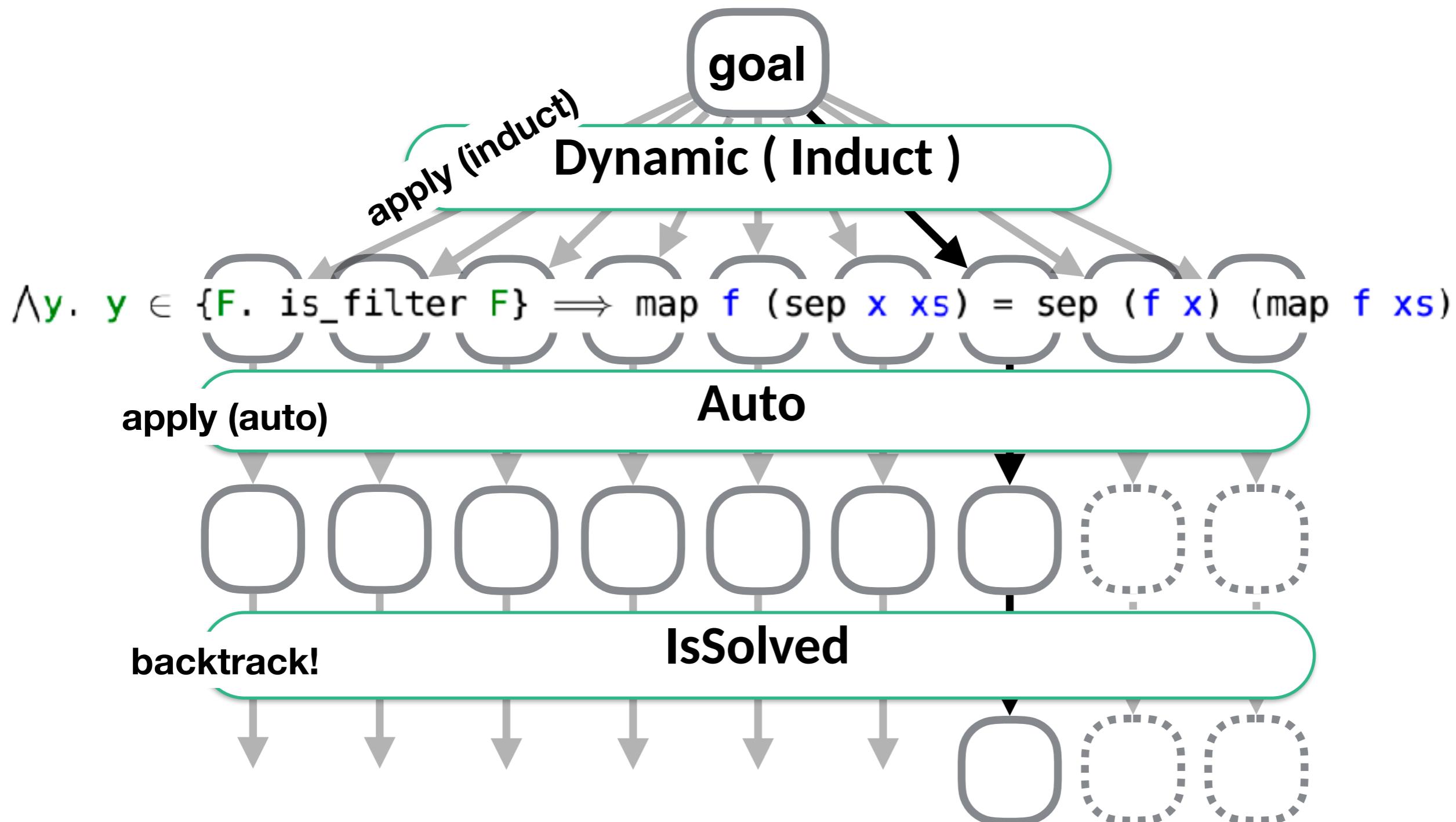


# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.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]*)
```

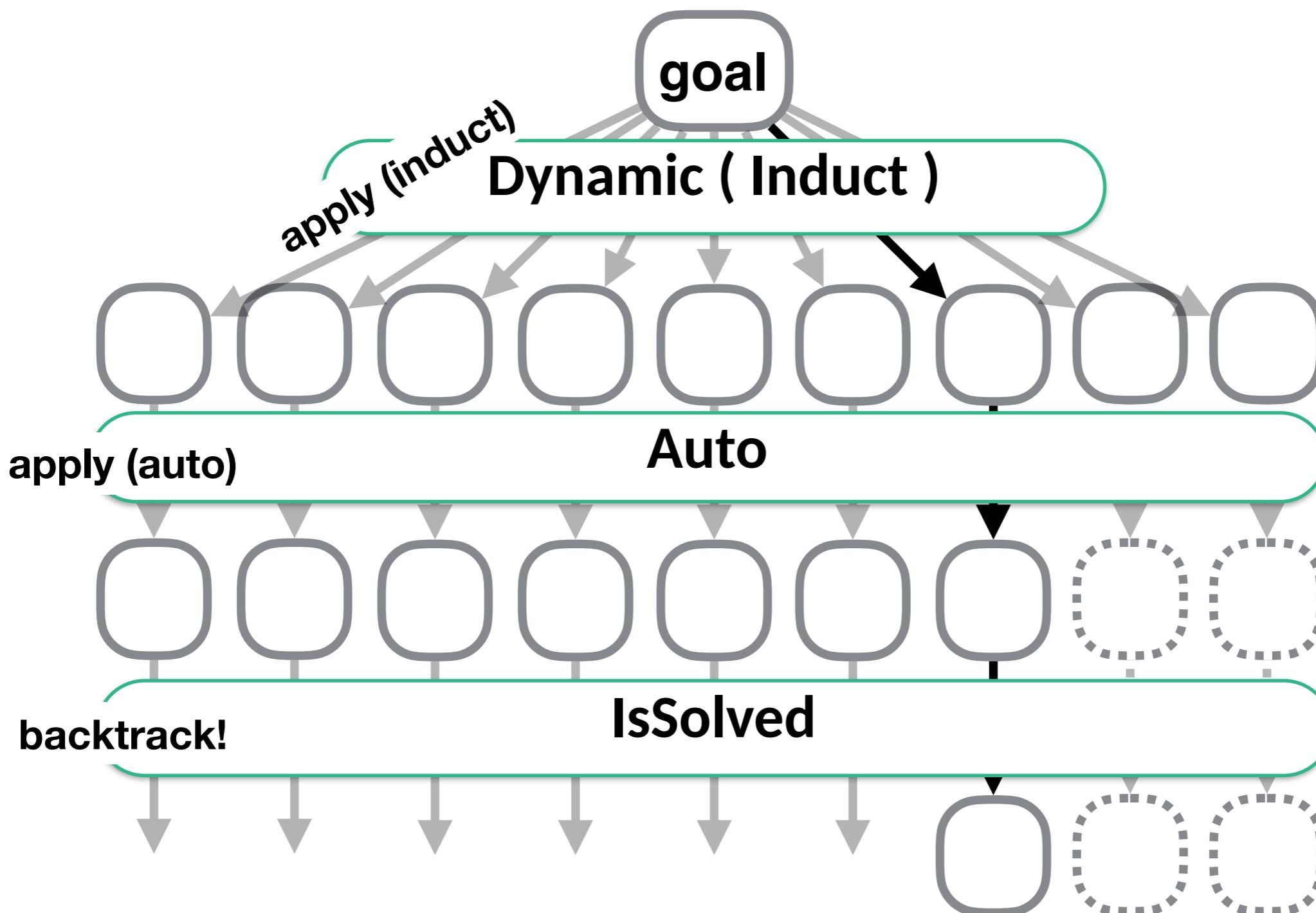


**DEMO1**

github.com/data61/PSL/blob/master/slides/2020\_NUS.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]*)
```

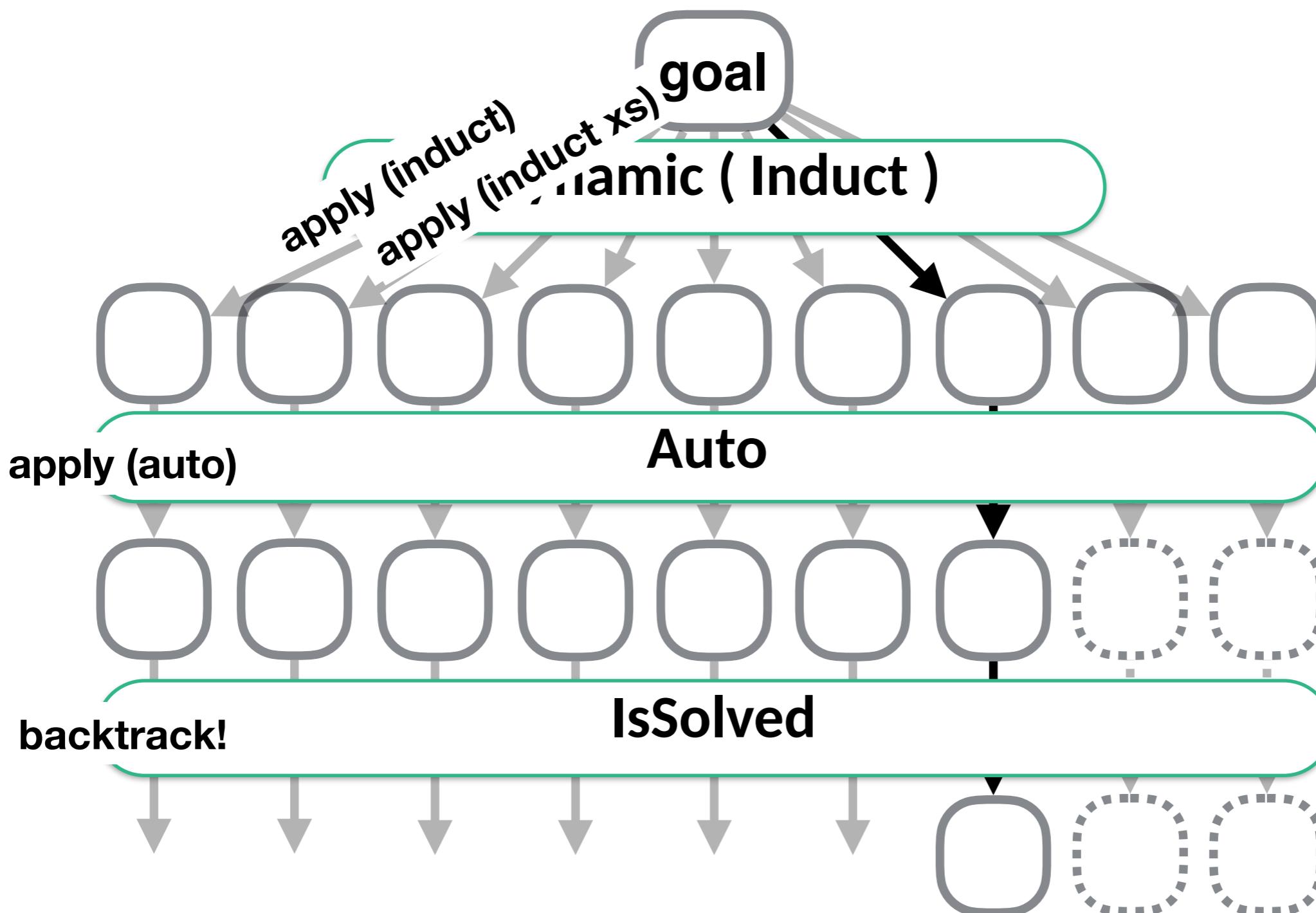


**DEMO1**

github.com/data61/PSL/blob/master/slides/2020\_NUS.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]*)
```

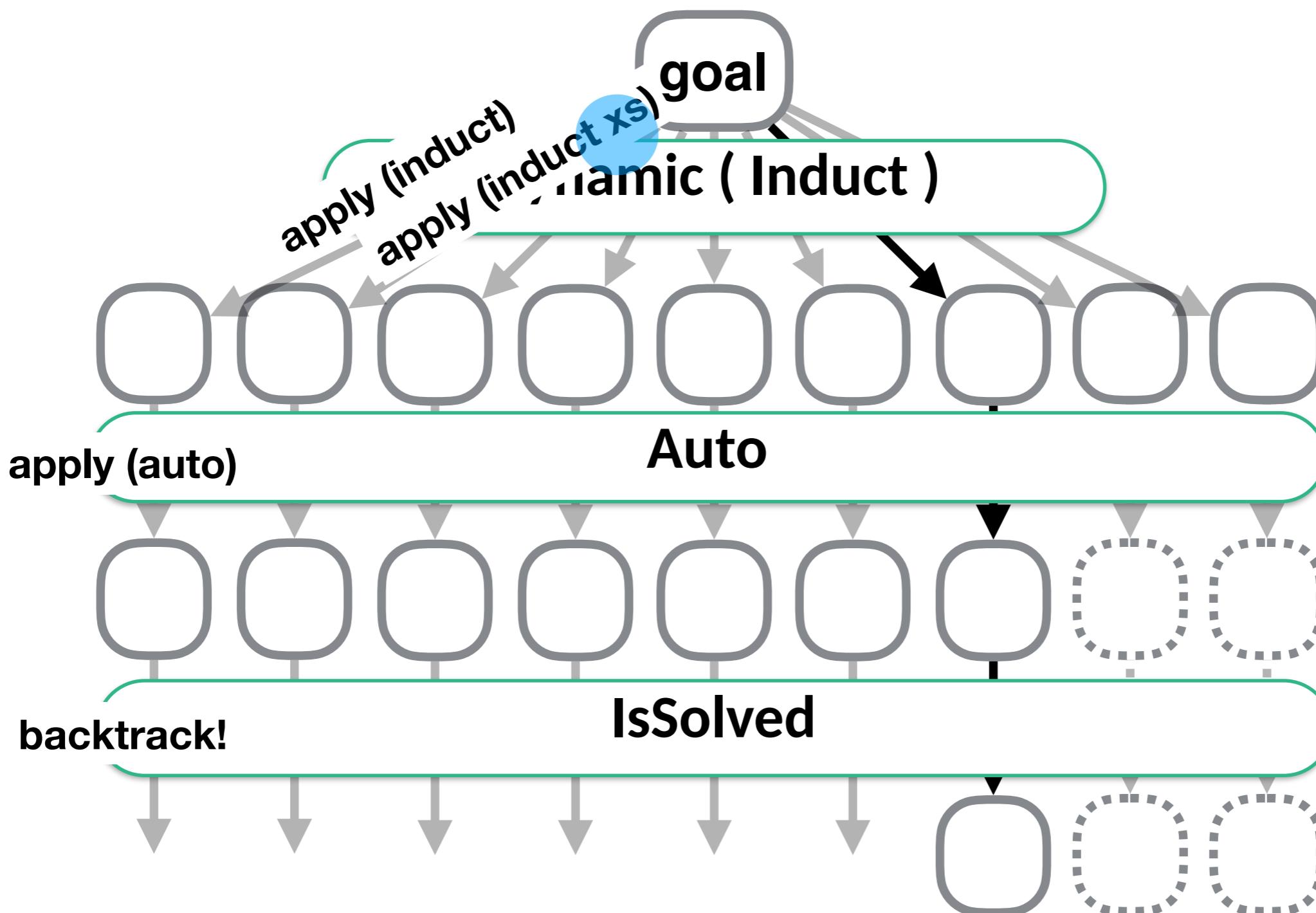


# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.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]*)
```

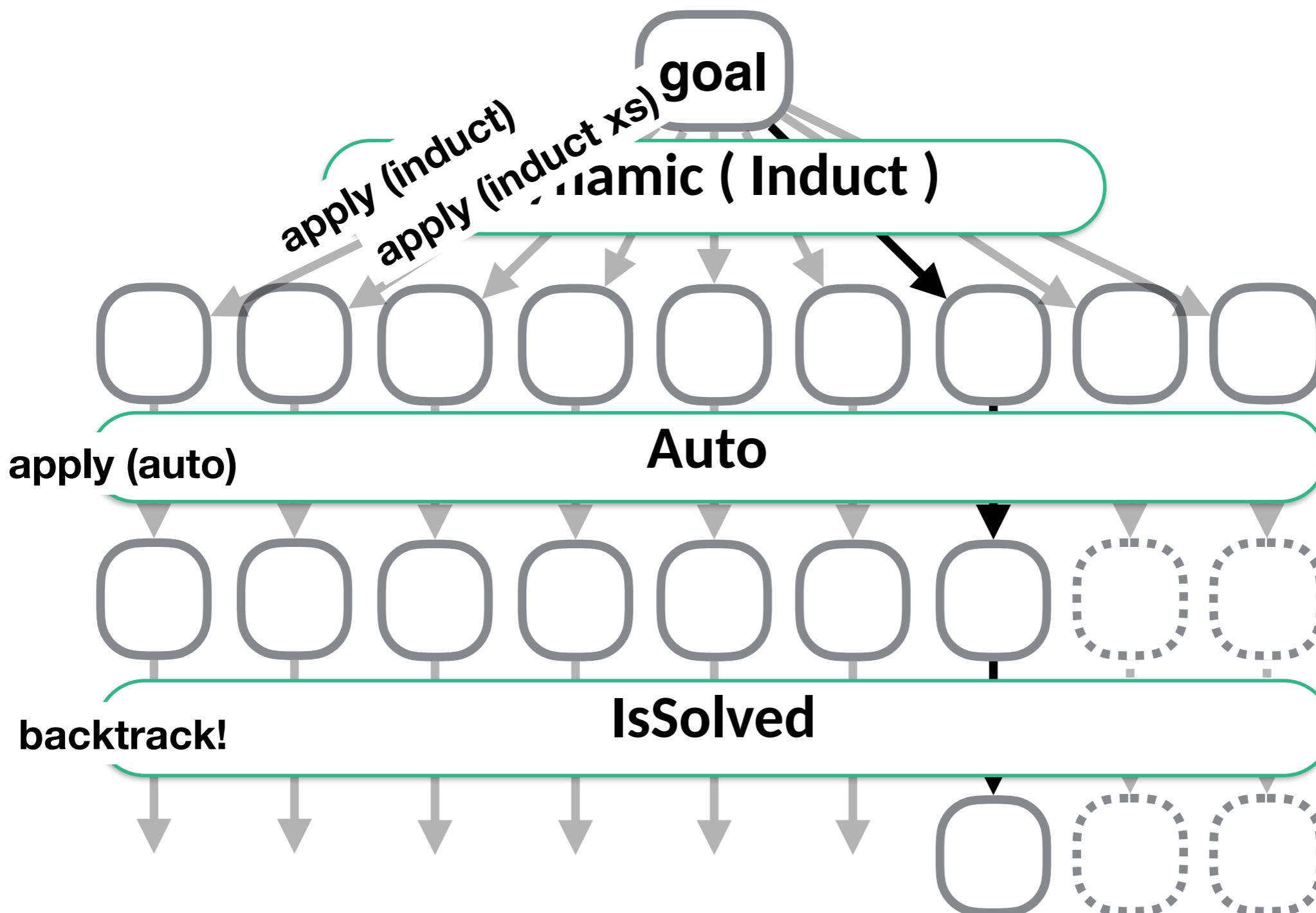


**DEMO1**

github.com/data61/PSL/blob/master/slides/2020\_NUS.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]*)
```

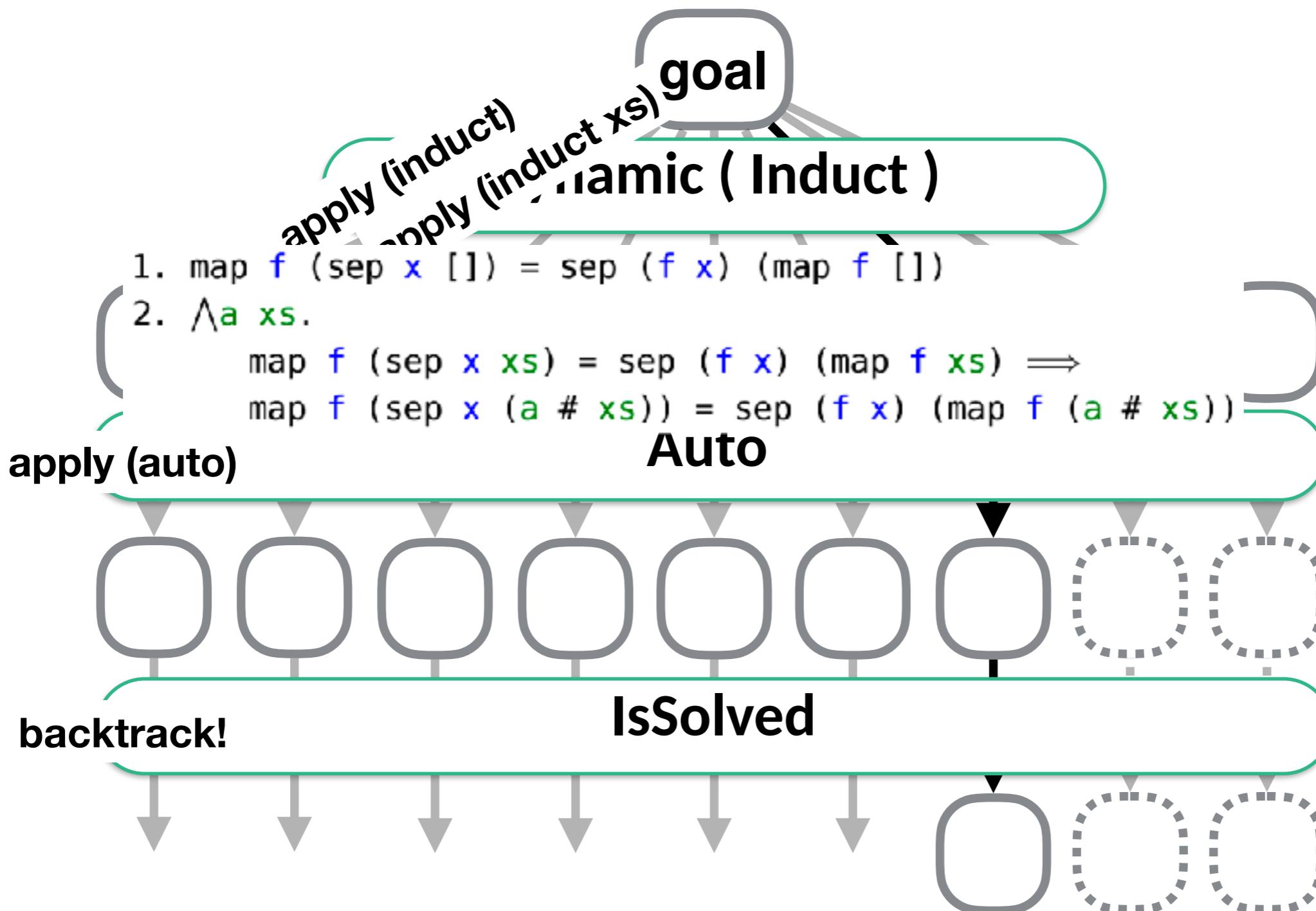


**DEMO1**

github.com/data61/PSL/blob/master/slides/2020\_NUS.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]*)
```

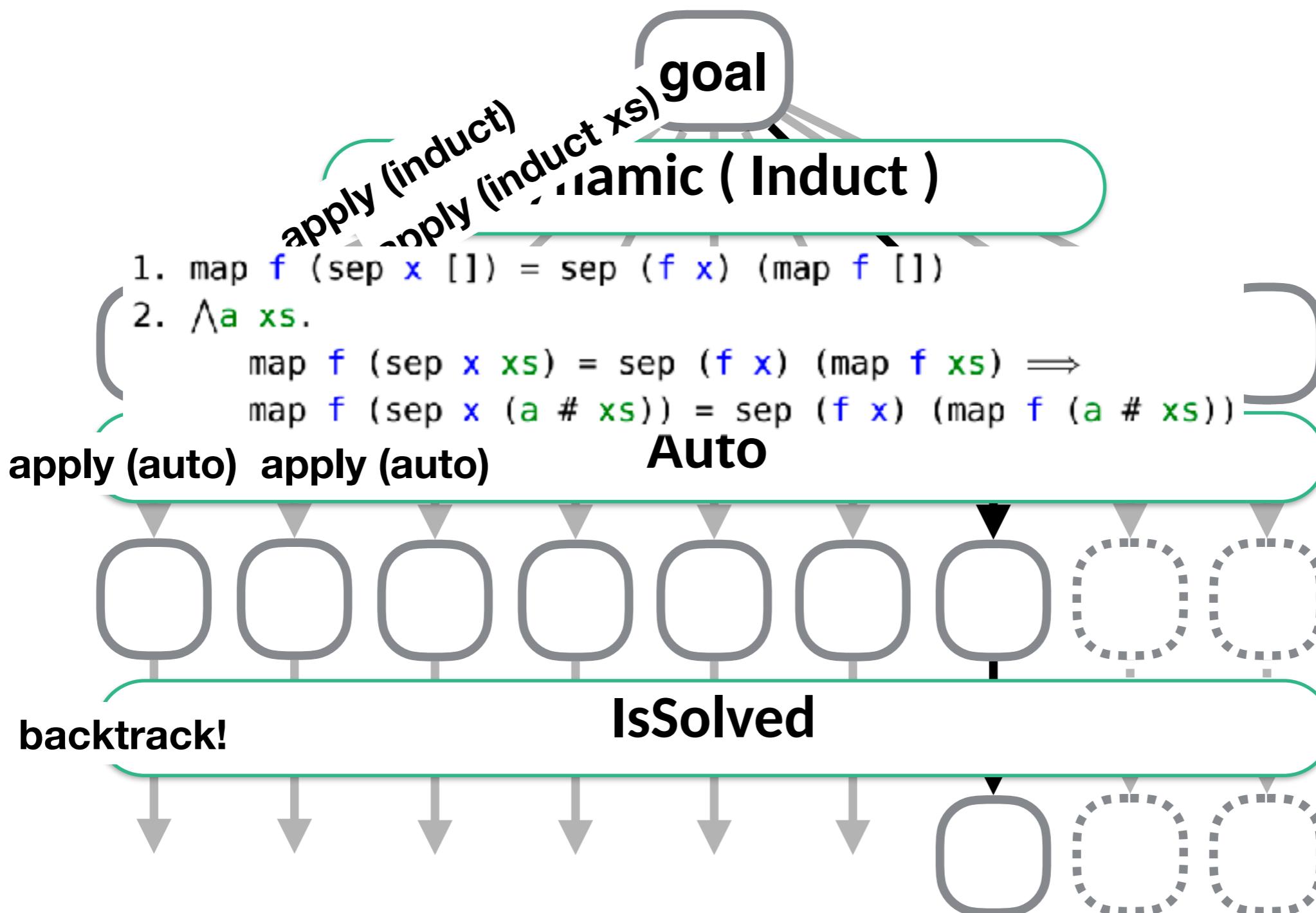


**DEMO1**

github.com/data61/PSL/blob/master/slides/2020\_NUS.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]*)
```

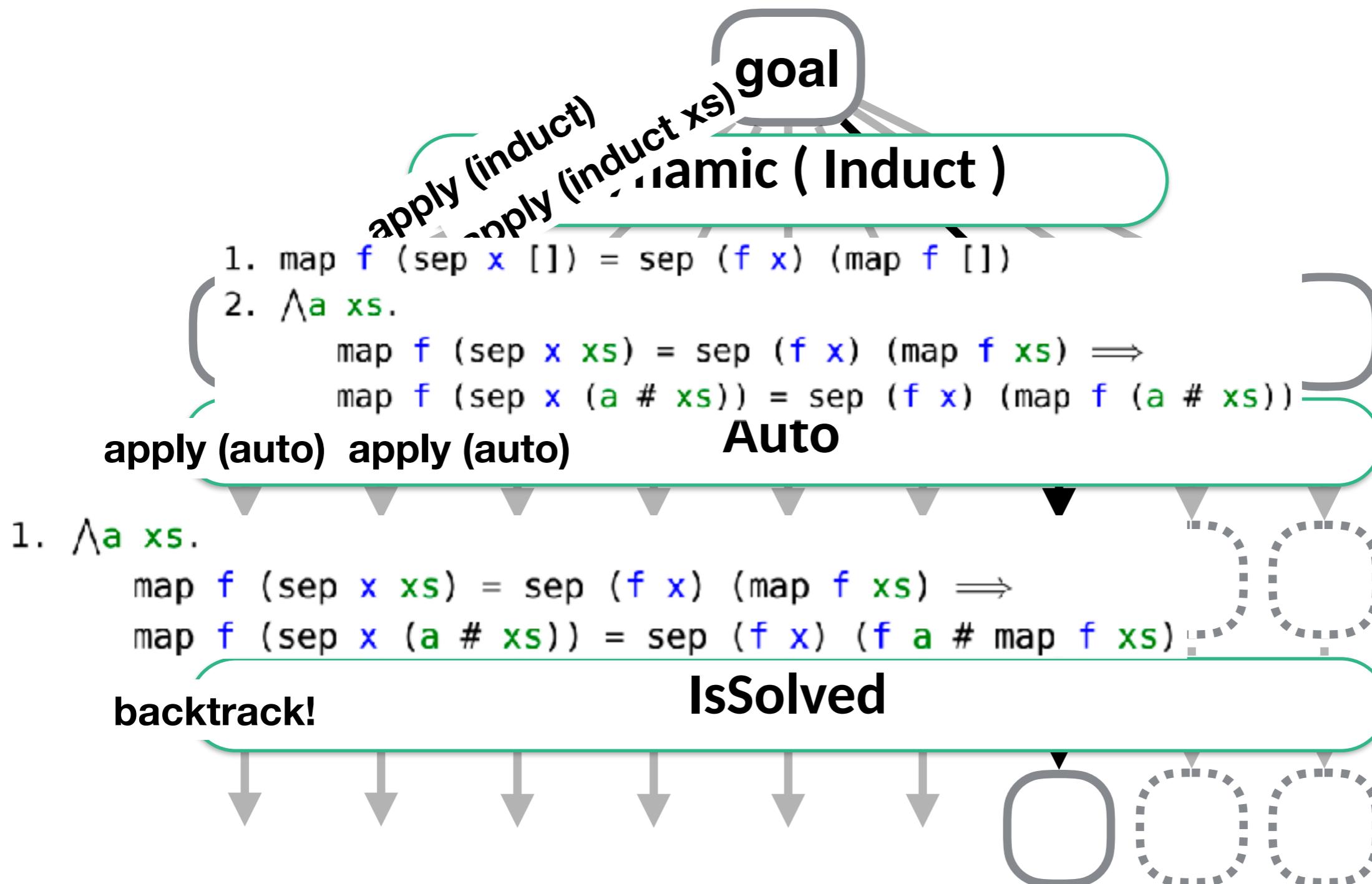


# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.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]*)
```

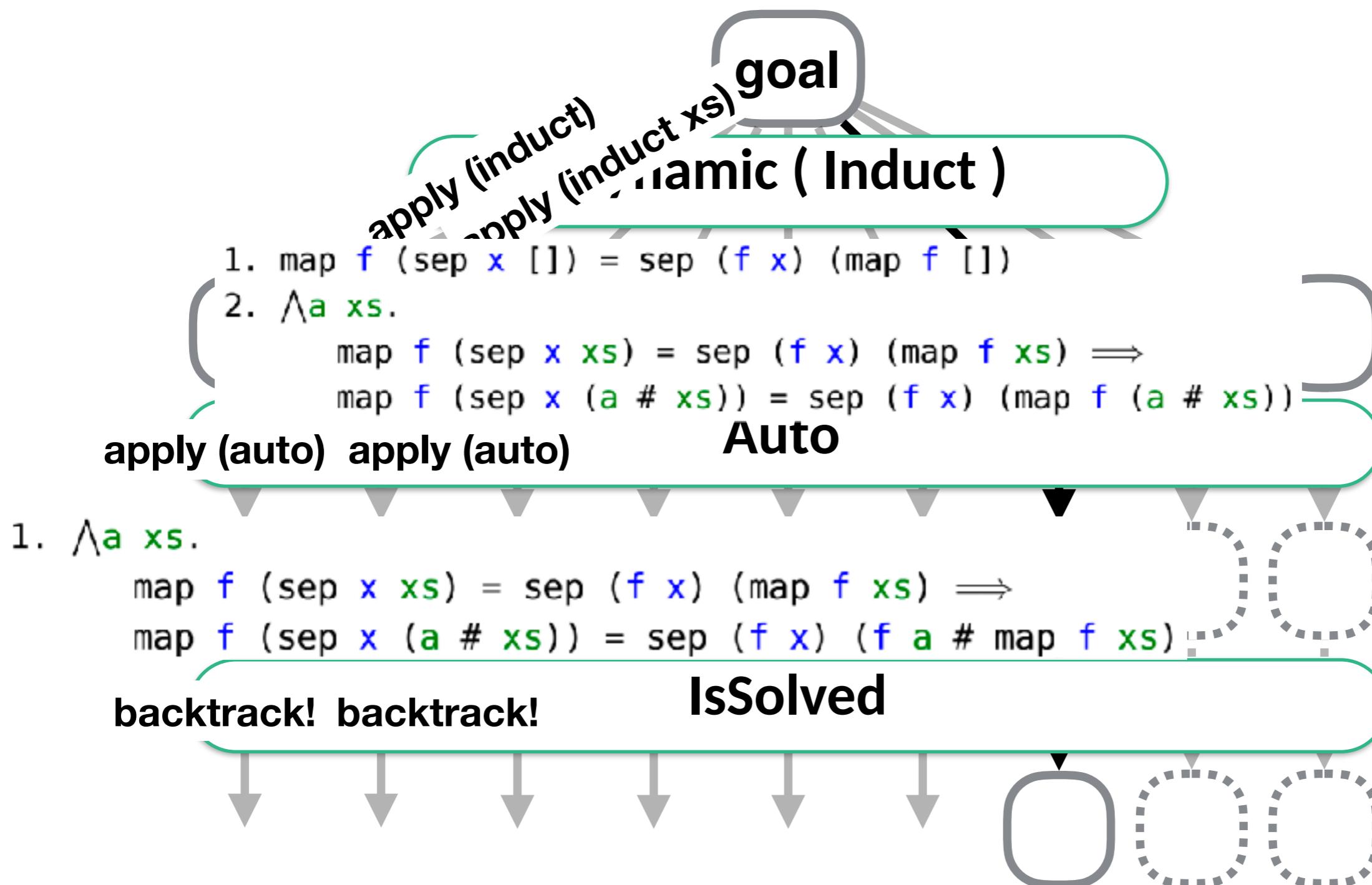


# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.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]*)
```

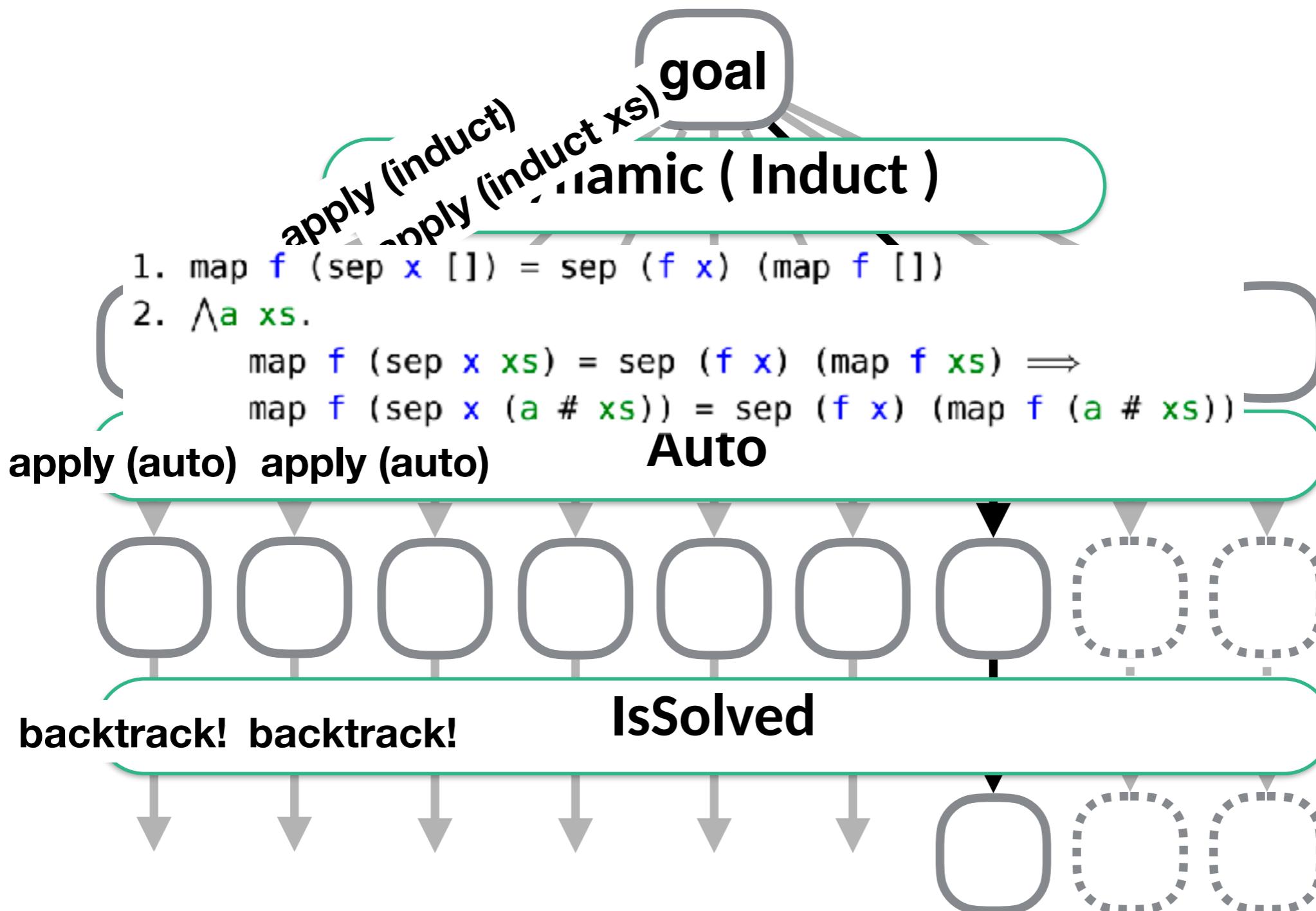


**DEMO1**

github.com/data61/PSL/blob/master/slides/2020\_NUS.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]*)
```

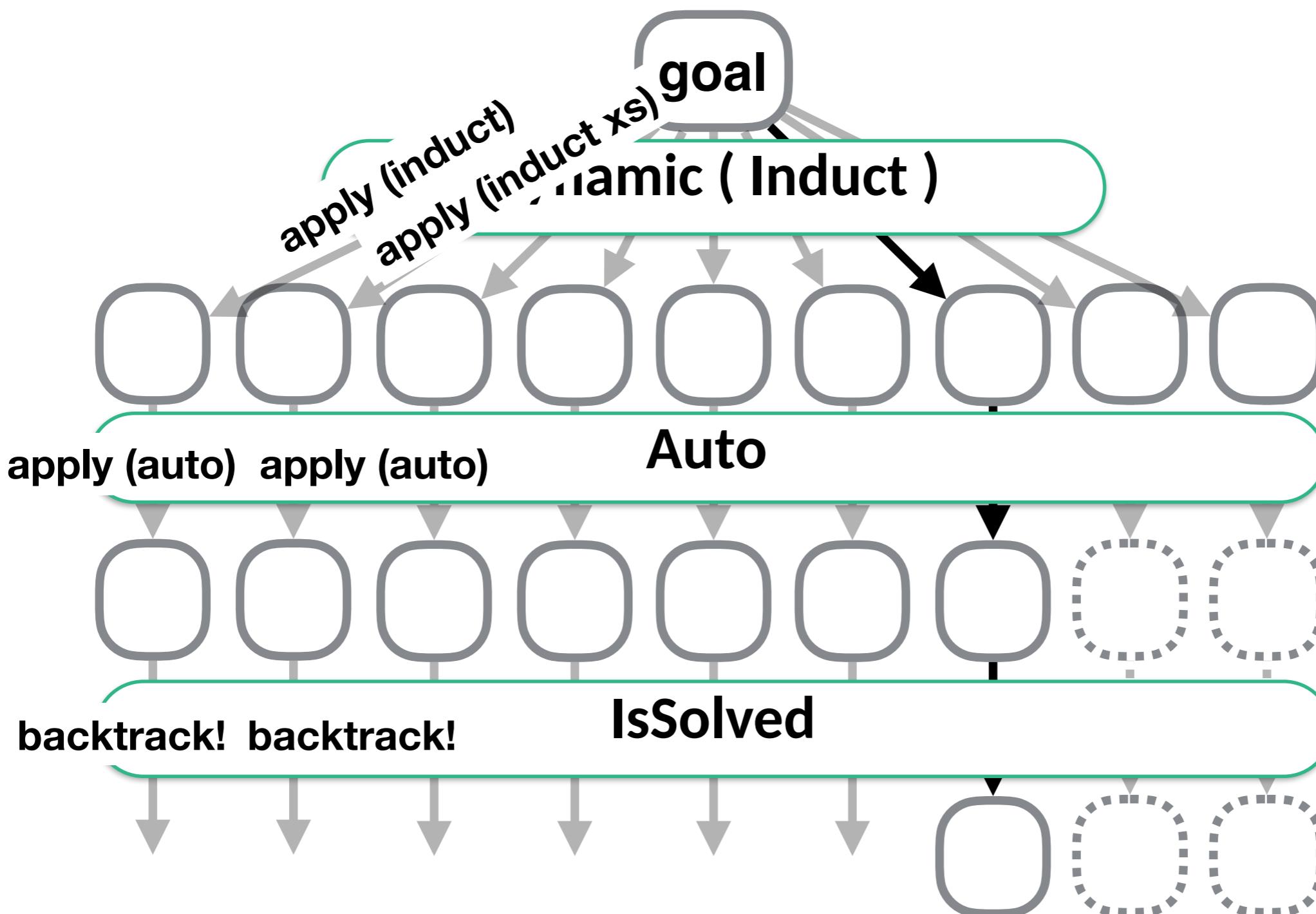


# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.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]*)
```

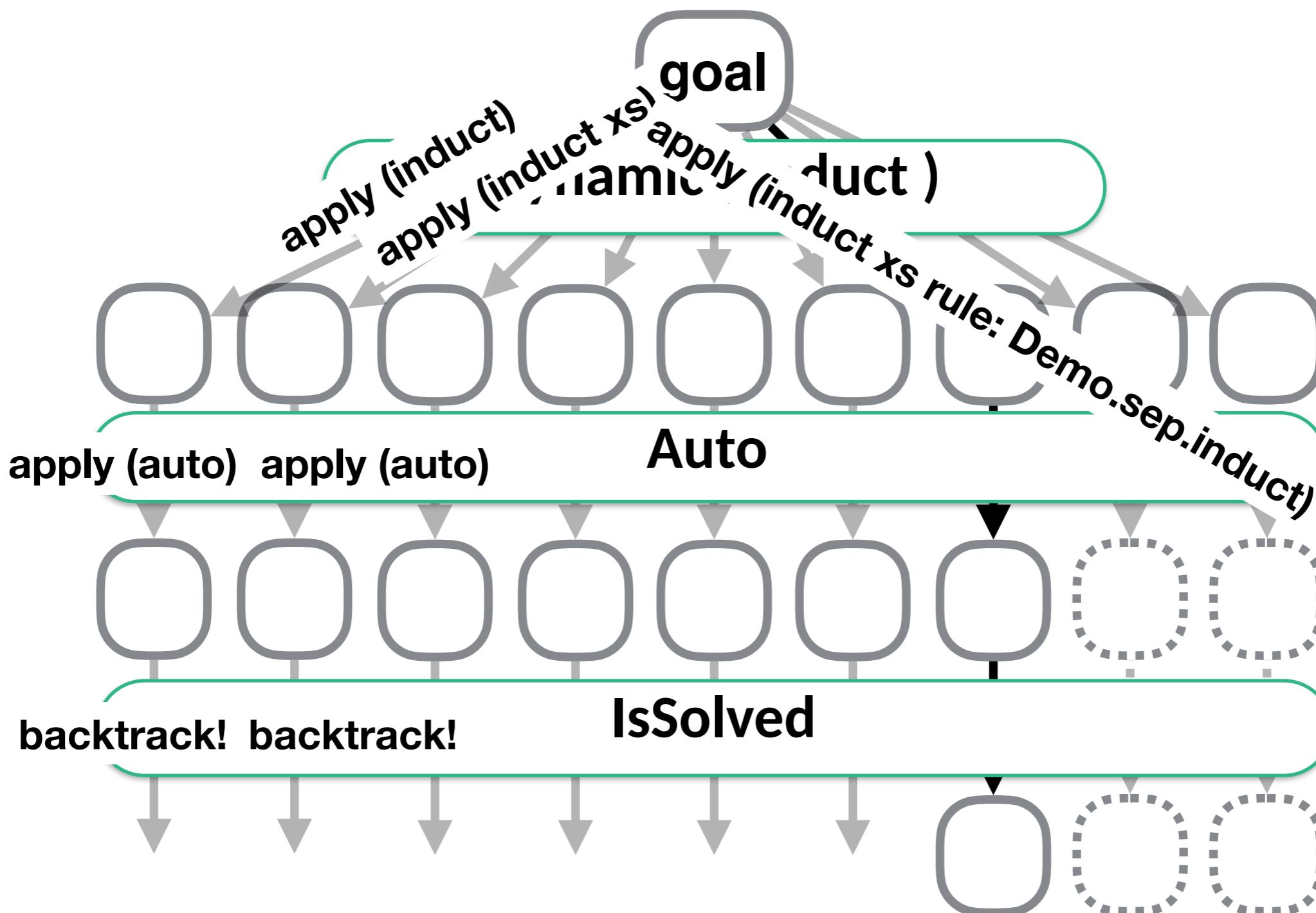


# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.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]*)
```

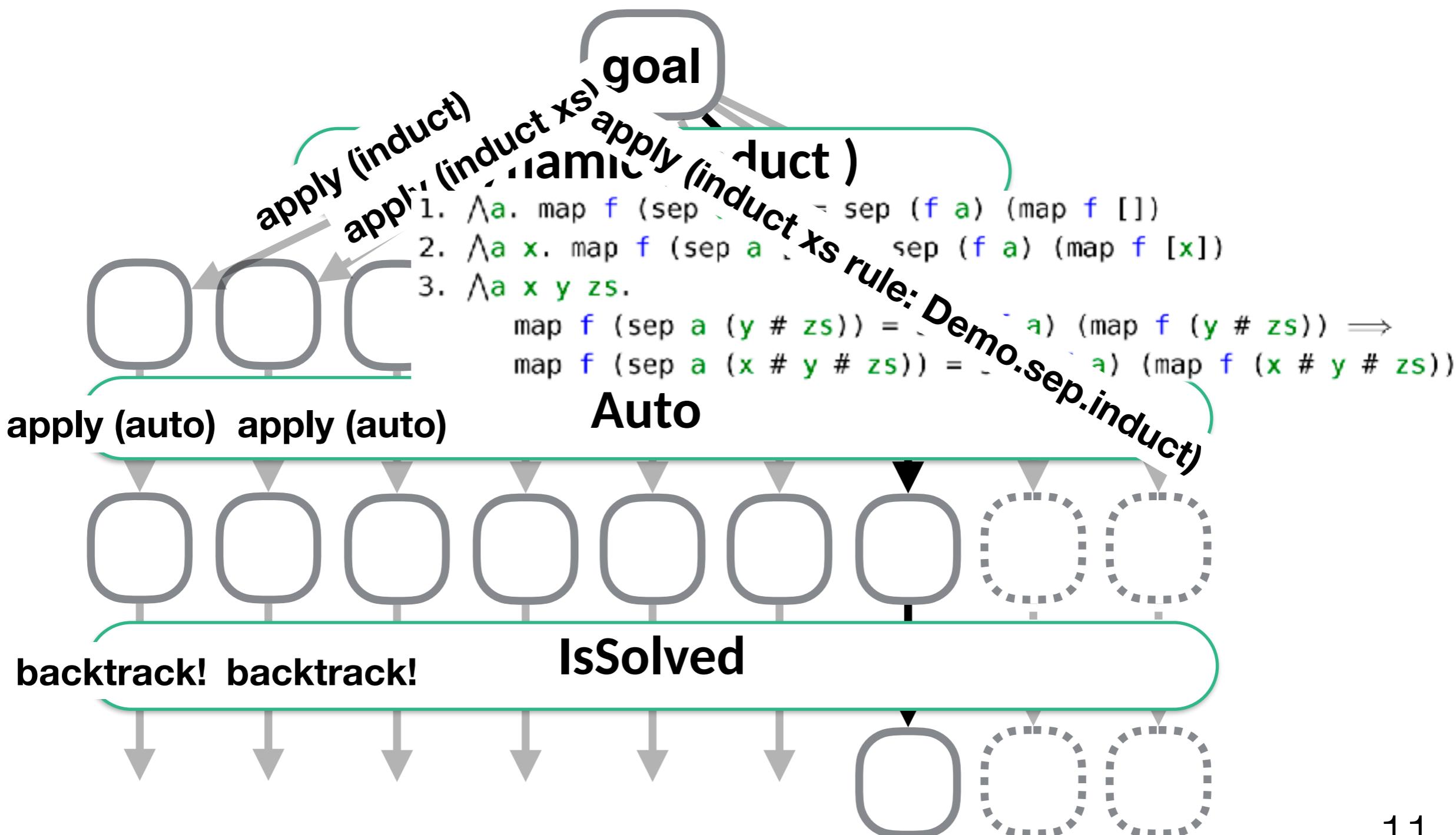


# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.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]*)
```

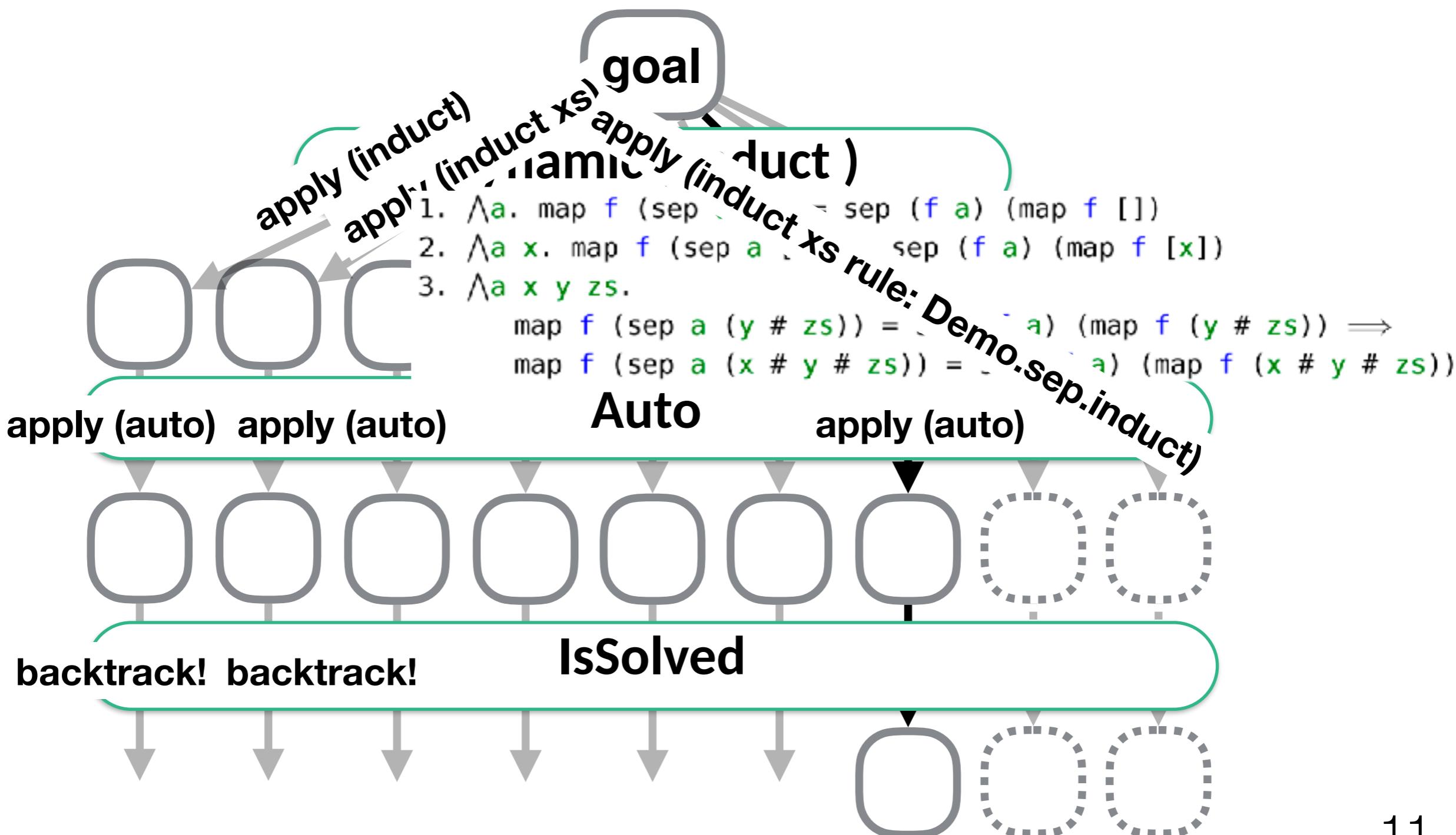


# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.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]*)
```

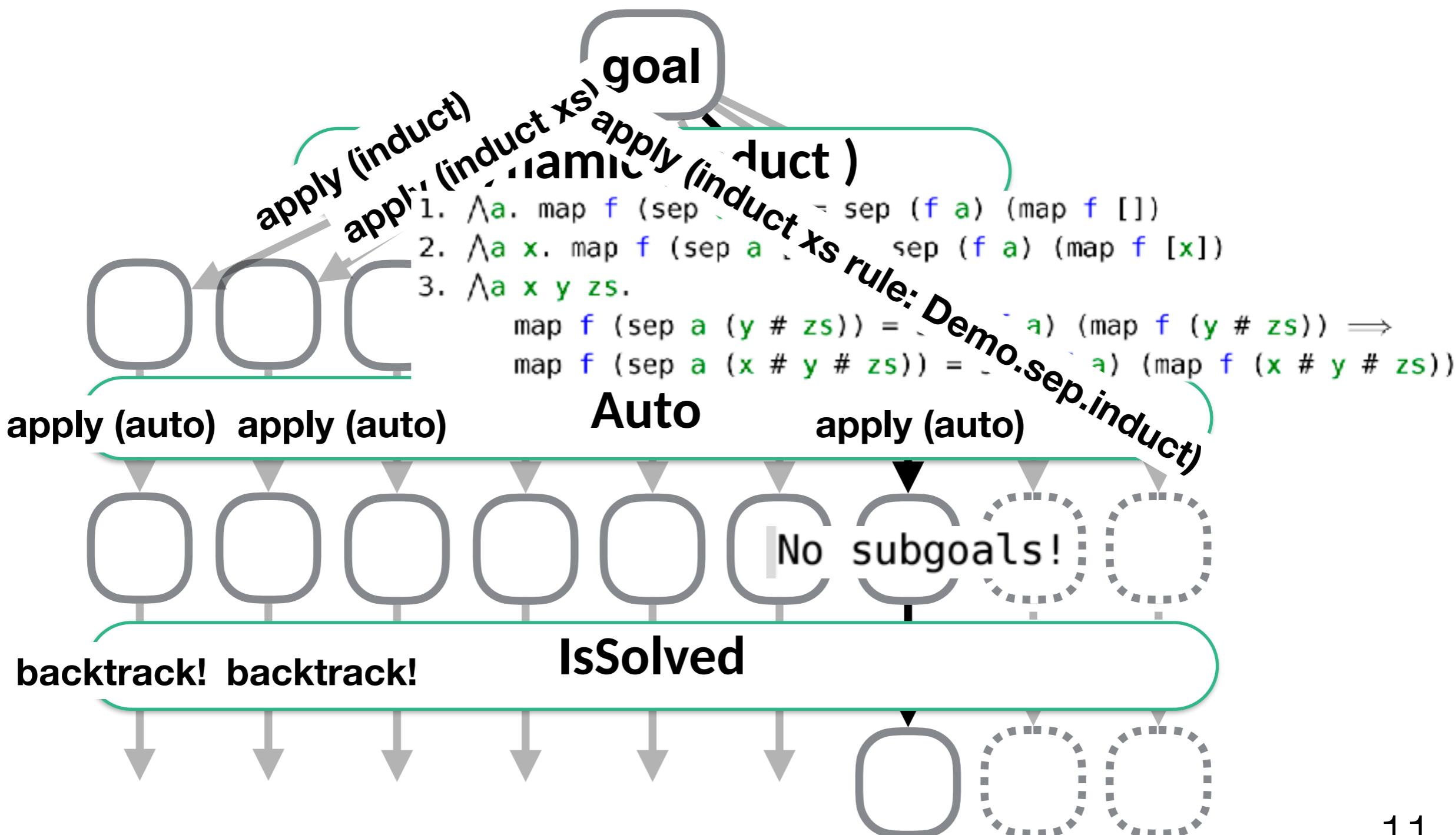


# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.pdf)

## PSL: Proof Strategy Language

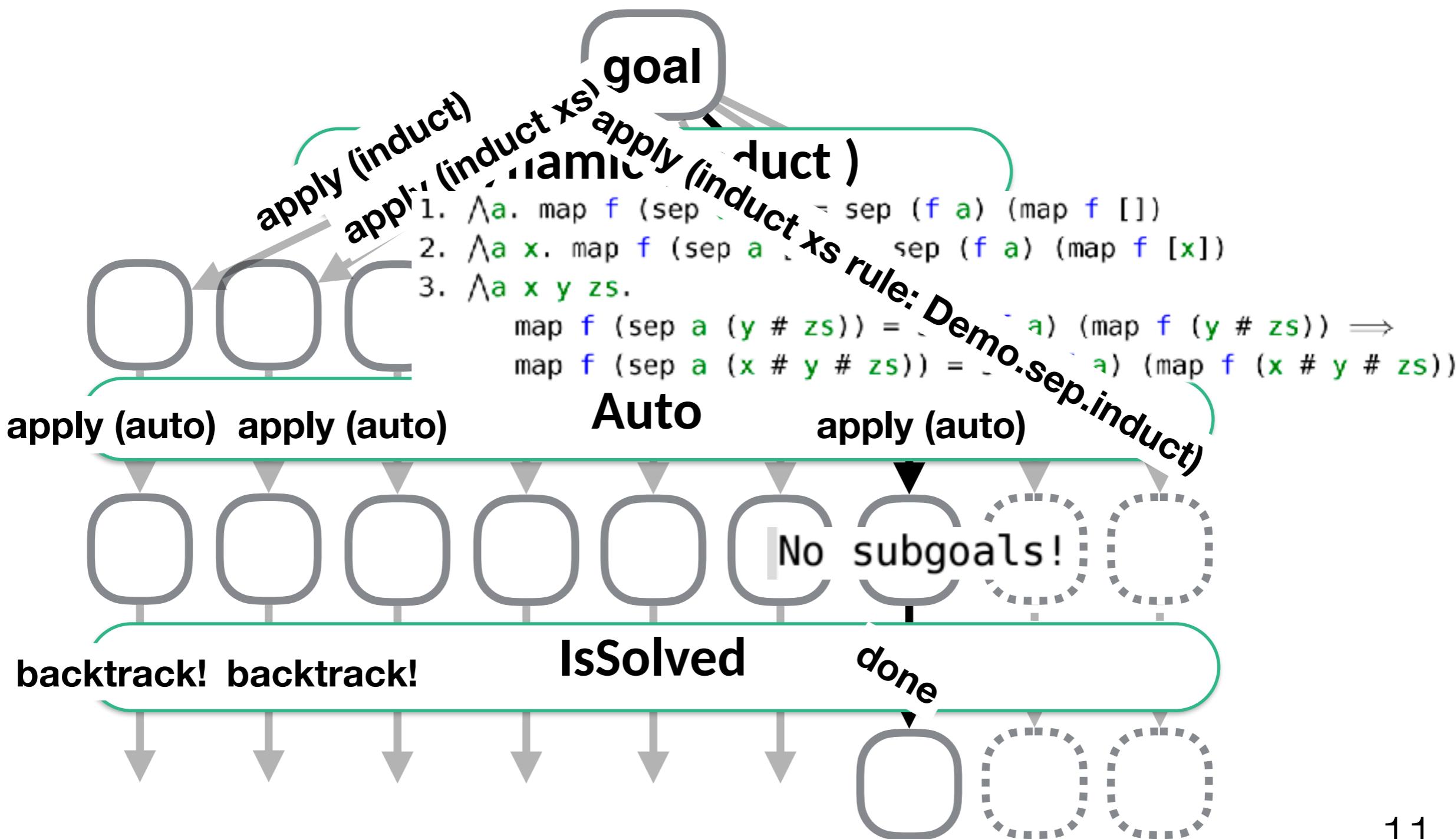
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



# 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]*)
```

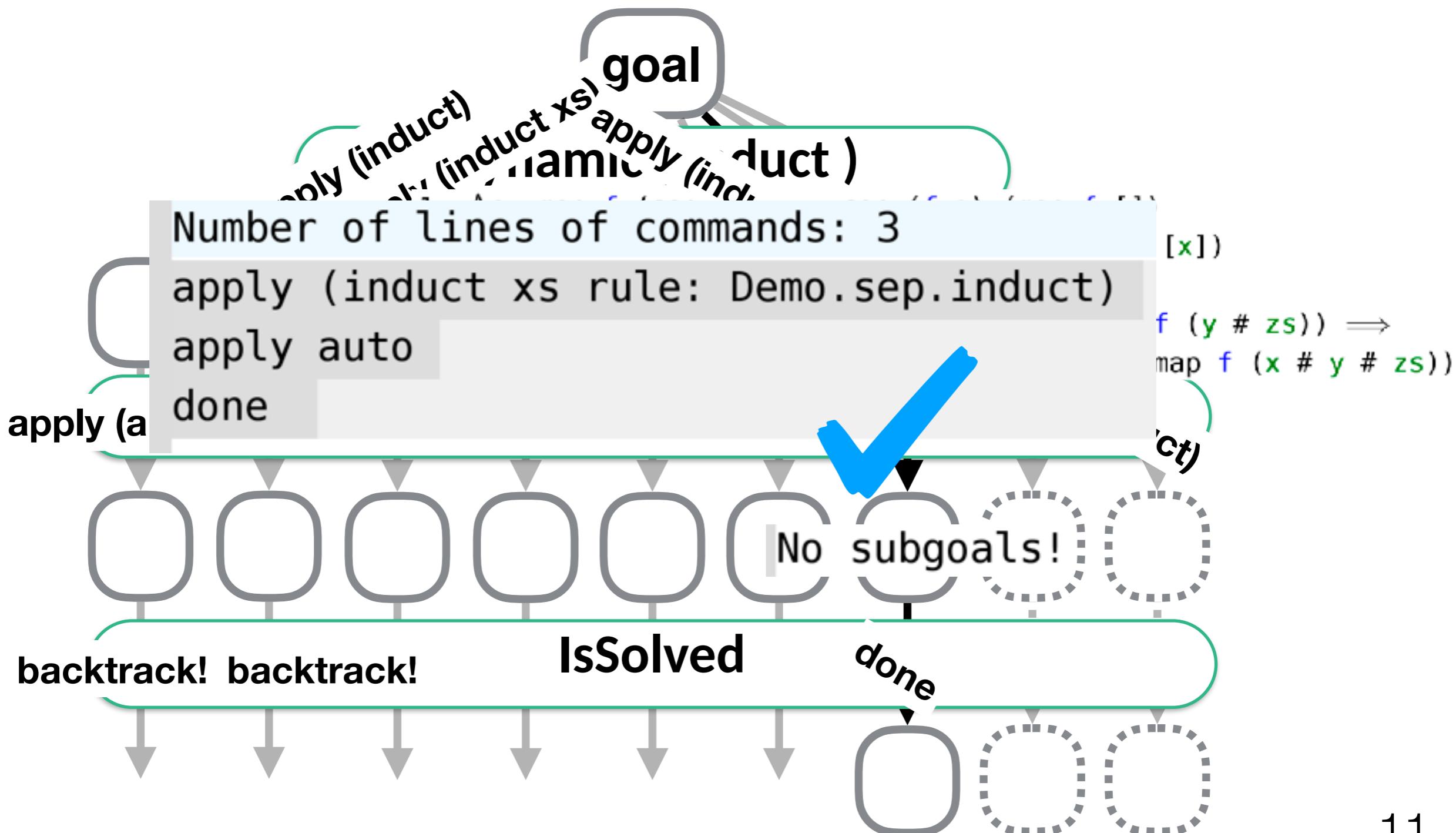


# DEMO1

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.pdf)

## PSL: Proof Strategy Language

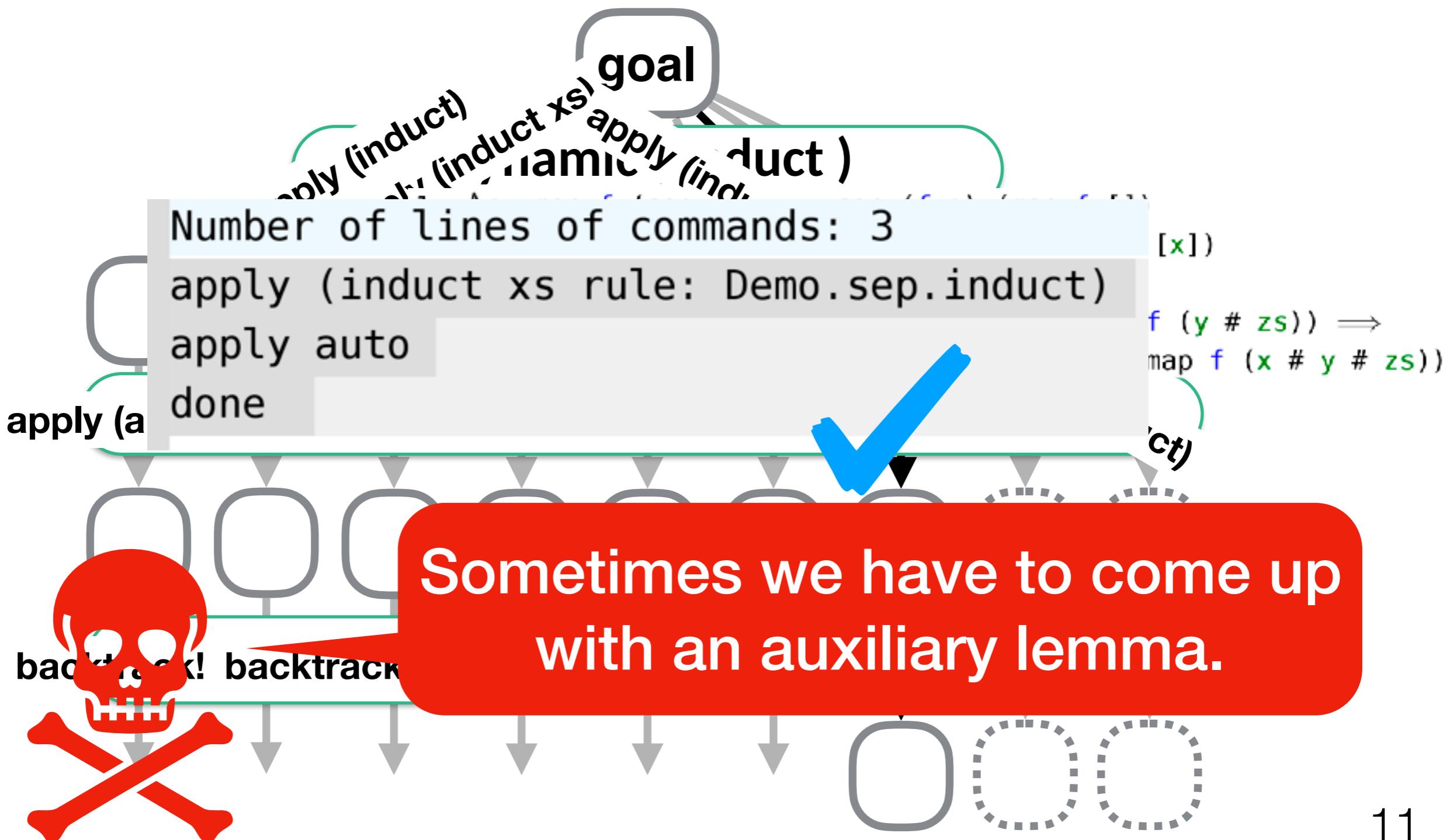
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



# 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]*)
```



# DEMO2

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

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

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

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

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

Proof controls at the bottom:

- Proof state (checkbox)
- Auto update (checkbox)
- Update button
- Search input field
- Zoom: 100%

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

# DEMO2

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

The screenshot shows the Isabelle proof assistant interface. The main window displays a file named "Example.thy". The code defines two primitive recursive functions for reversing lists: "rev" and "itrev". The "rev" function takes a list and returns its reverse. The "itrev" function takes a list and returns its reverse, with a specific rule for the empty list. A strategy "DInd" is defined for theorems involving these functions. A lemma is also present, stating that the empty list's reverse is the empty list. The cursor is currently over the "find\_proof" command.

```
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]

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

empty sequence. no proof found.

# DEMO2

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

The screenshot shows the PSL IDE interface with a theory file named `Example.thy`. The code defines two primitive recursive functions: `rev` and `itrev`, their corresponding inverse strategies, and a lemma. The code is as follows:

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

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

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

The line `find_proof DInd_Or_CDInd` is highlighted in red, indicating an error or a warning.

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

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

The screenshot shows the PSL IDE interface with a theory file named `Example.thy`. The code defines two primitive recursive functions for reversing lists: `rev` and `itrev`, and three strategies: `DInd`, `CDInd`, and `DInd_Or_CDInd`. The `itrev` lemma is currently being worked on, with its proof steps highlighted.

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

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

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

PGT creates 131 conjectures.

Testing conjecture with Quickcheck-exhaustive...

Testing conjecture with Quickcheck-exhaustive...

Testing conjecture with Quickcheck-exhaustive...

Testing conjecture with Quickcheck-exhaustive...

# DEMO2

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.pdf)

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

# DEMO2

[https://github.com/data61/PSL/blob/master/slides/2020\\_NUS.pdf](https://github.com/data61/PSL/blob/master/slides/2020_NUS.pdf)

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

# What happened?

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

goal

DEMO2

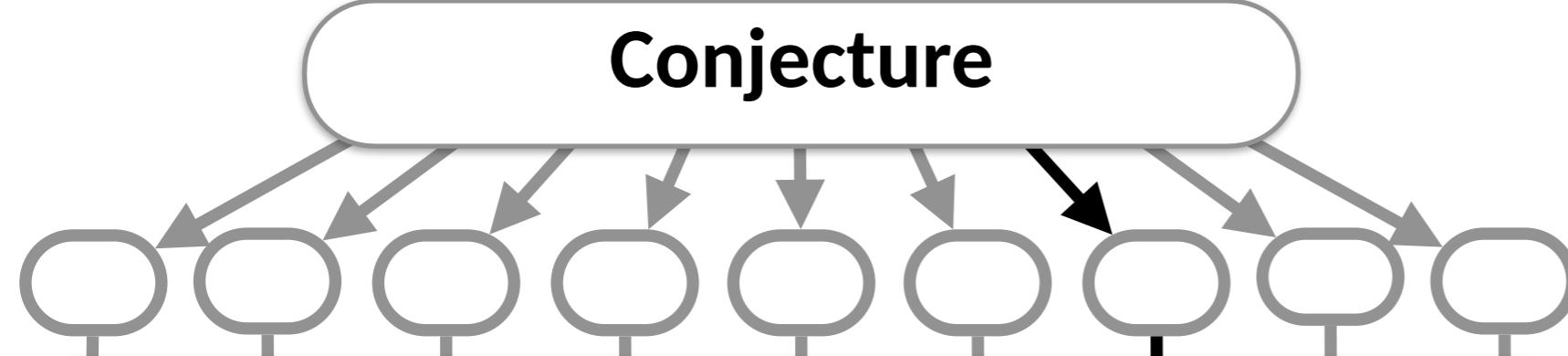
**strategy** CDInd = Thens [Conjecture, Fastforce, Quickcheck, DInd]

# DEMO2

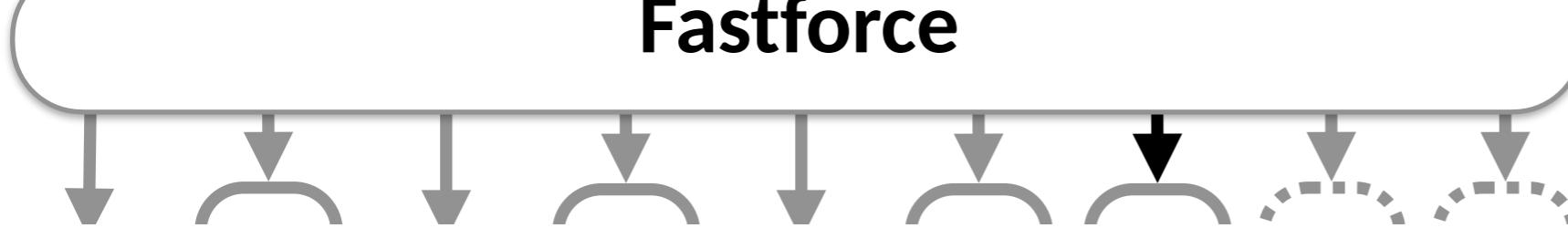
```
goal (1 subgoal):  
1. itrev xs [] = rev xs
```

goal

Conjecture



Fastforce

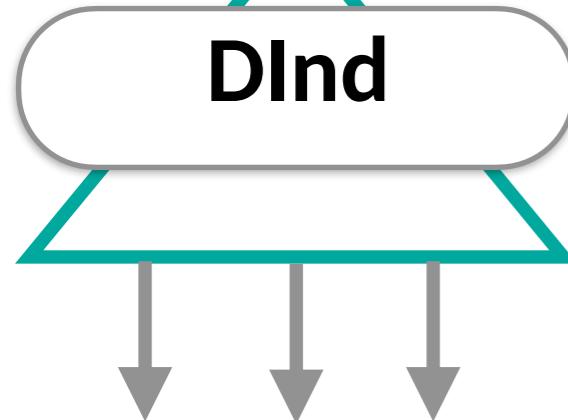


```
strategy CDInd = Thens [Conjecture, Fastforce, Quickcheck, DInd]
```

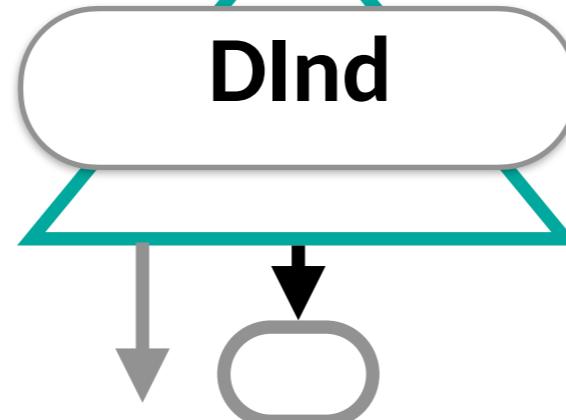
Quickcheck



DInd

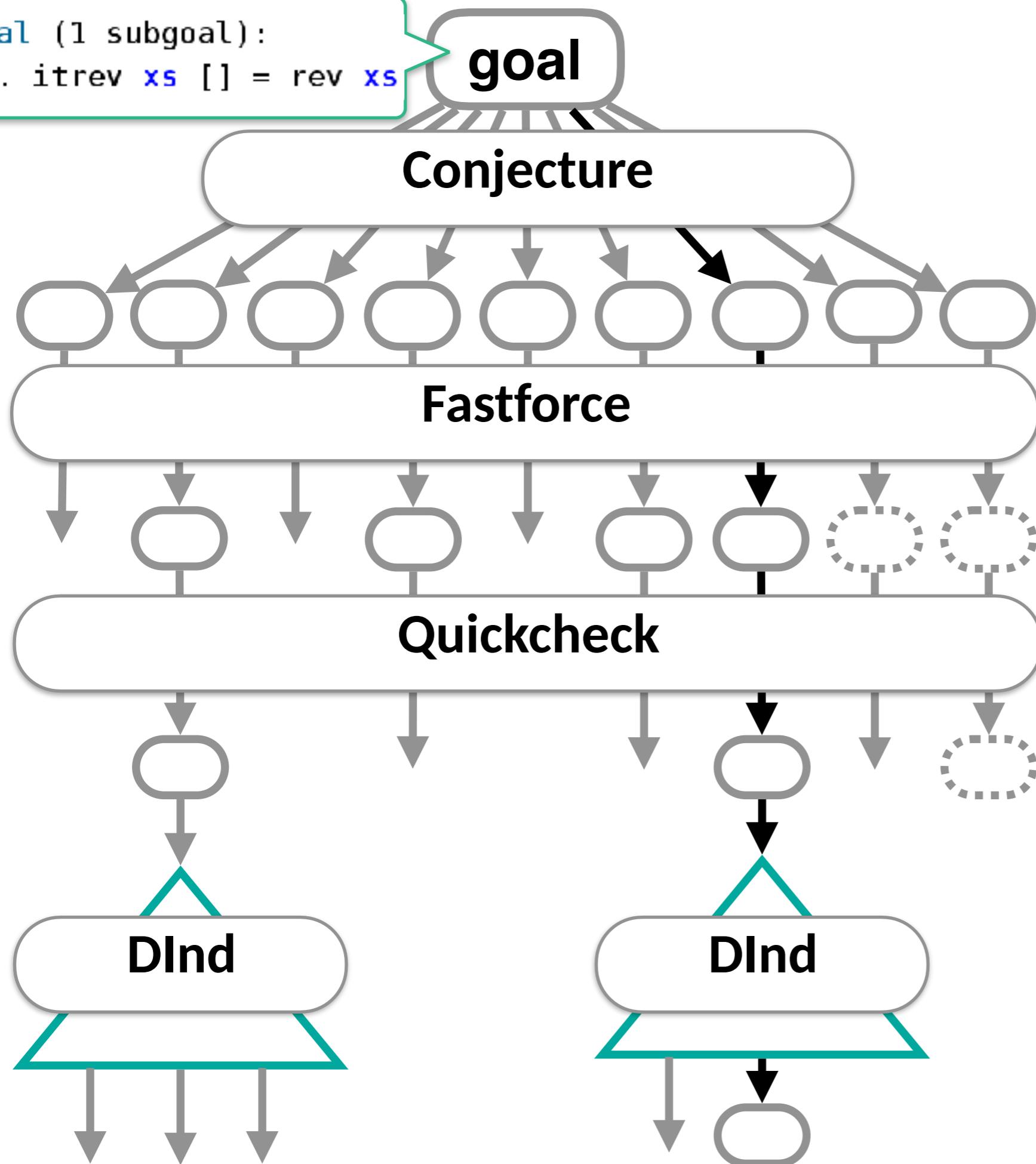


DInd



# DEMO2

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



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

goal

```
apply (subgoal_tac  
"Nil. itrev xs Nil = rev xs @ Nil")
```

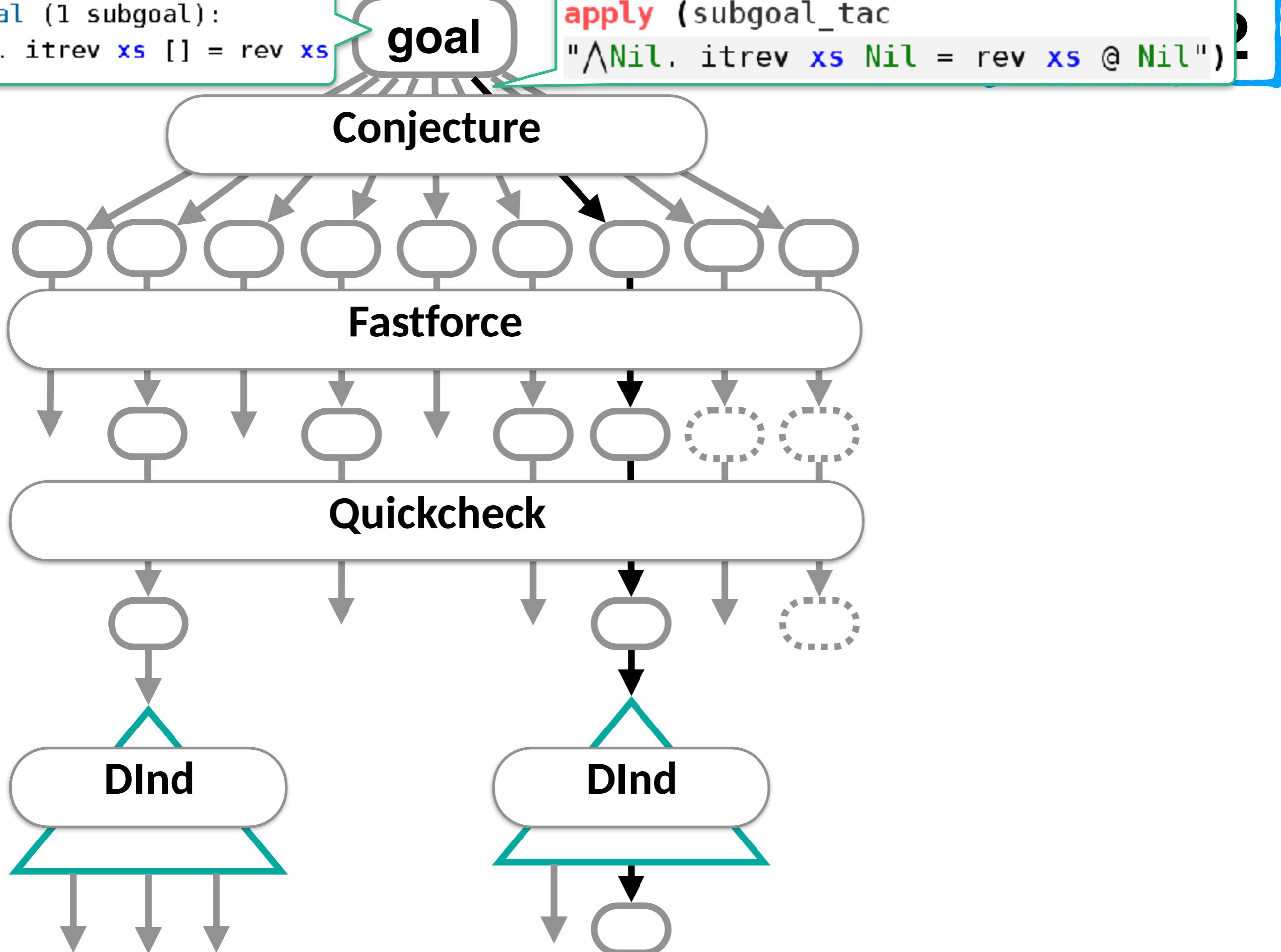
Conjecture

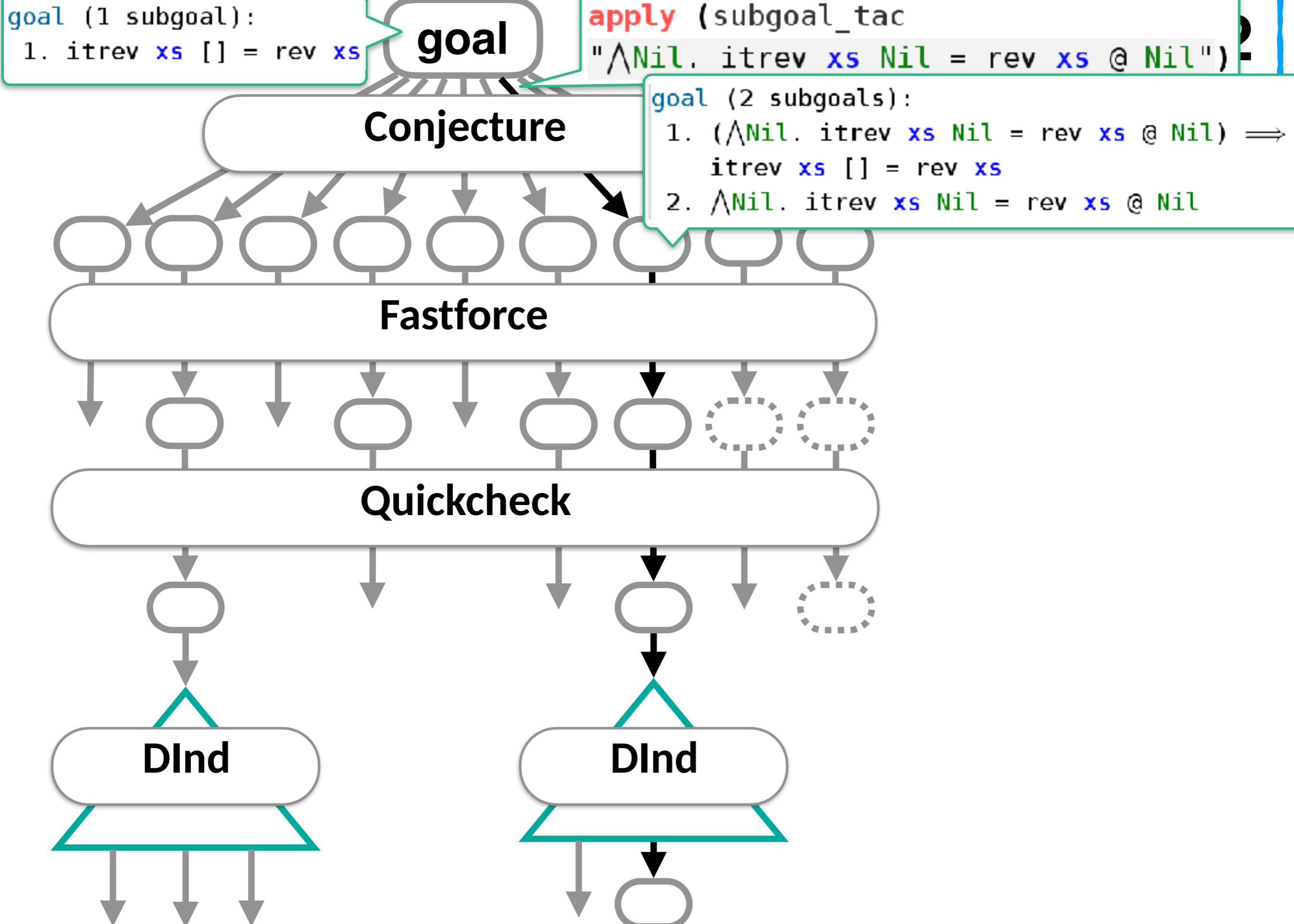
Fastforce

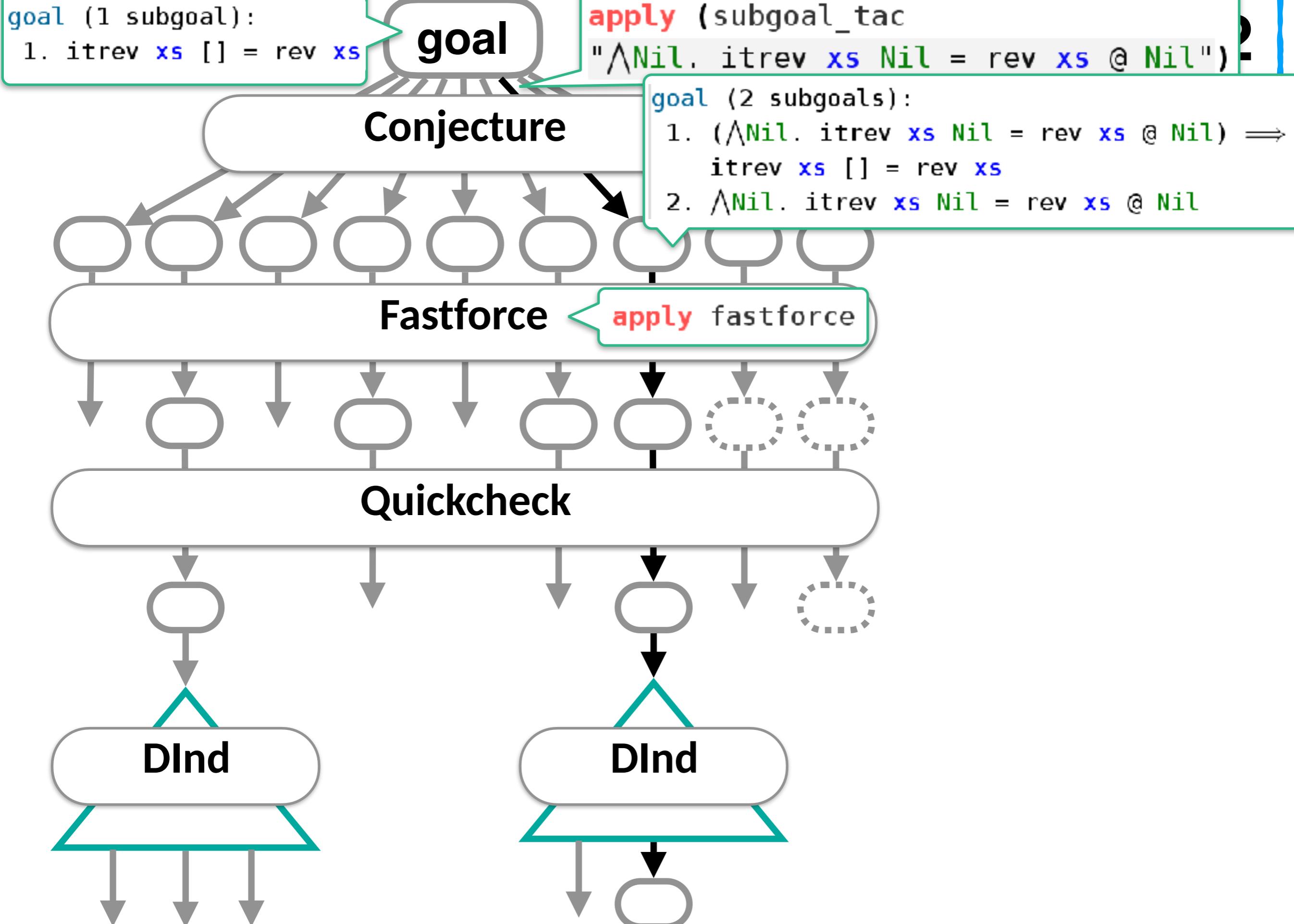
Quickcheck

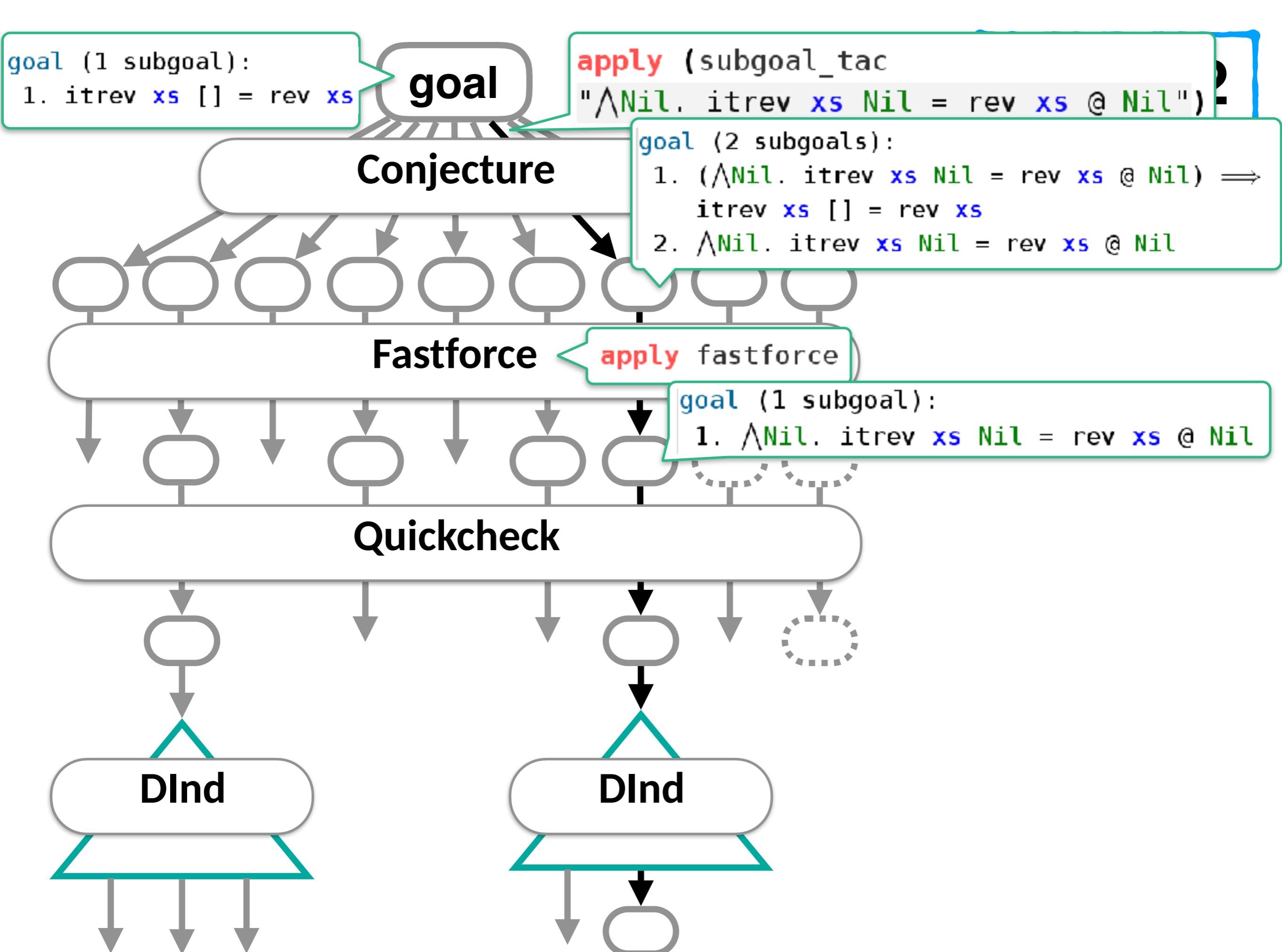
DInd

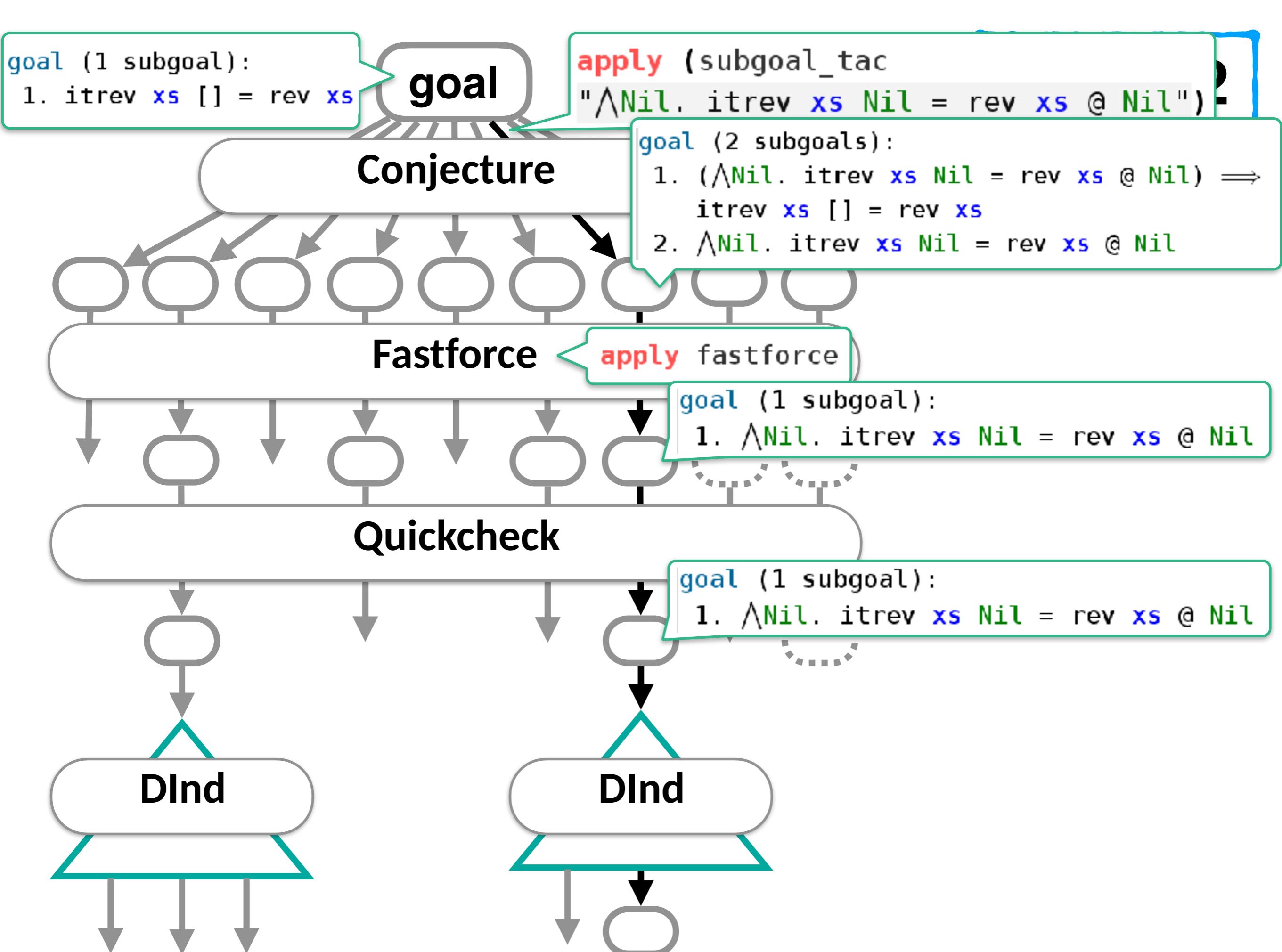
DInd

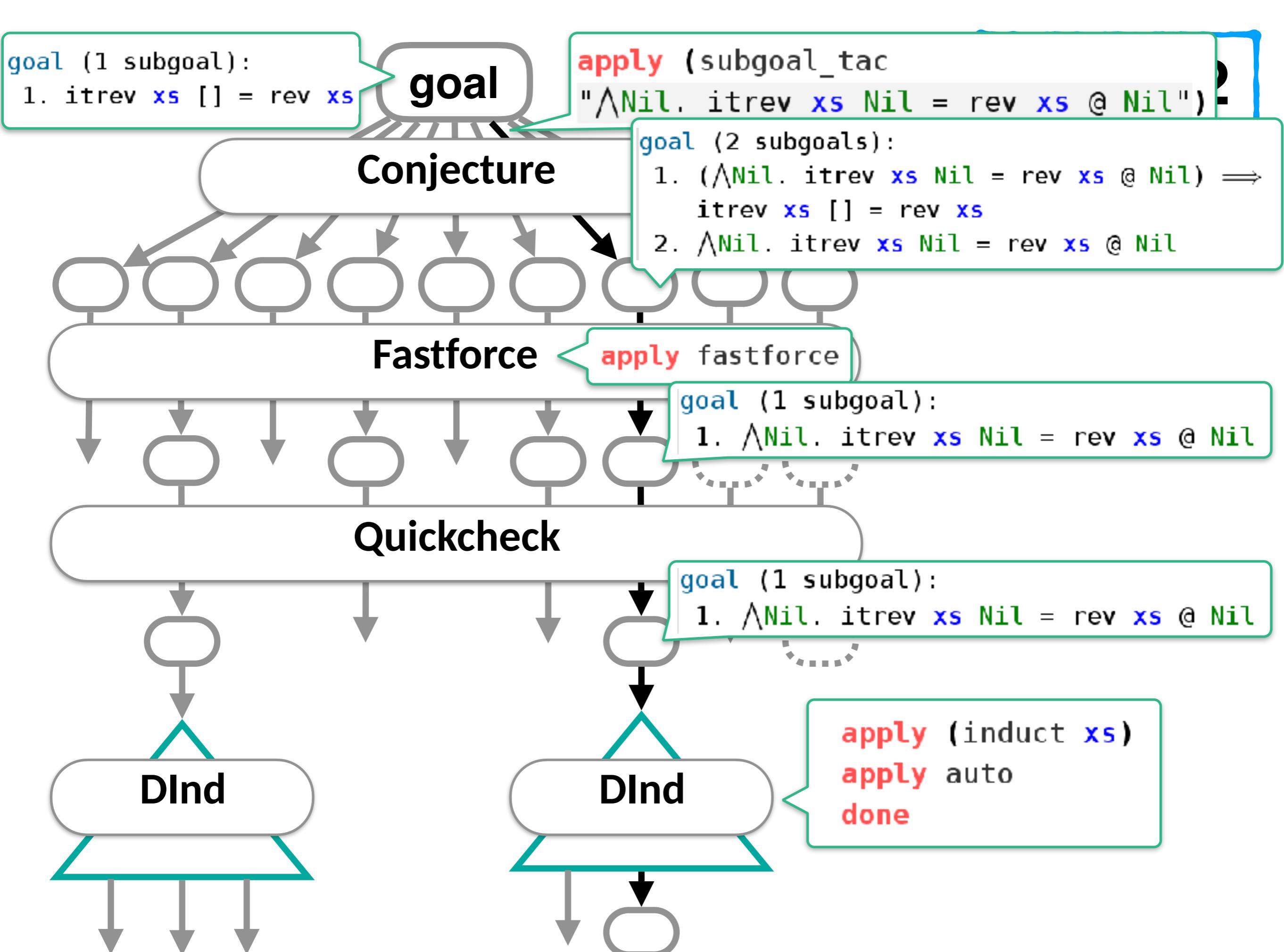


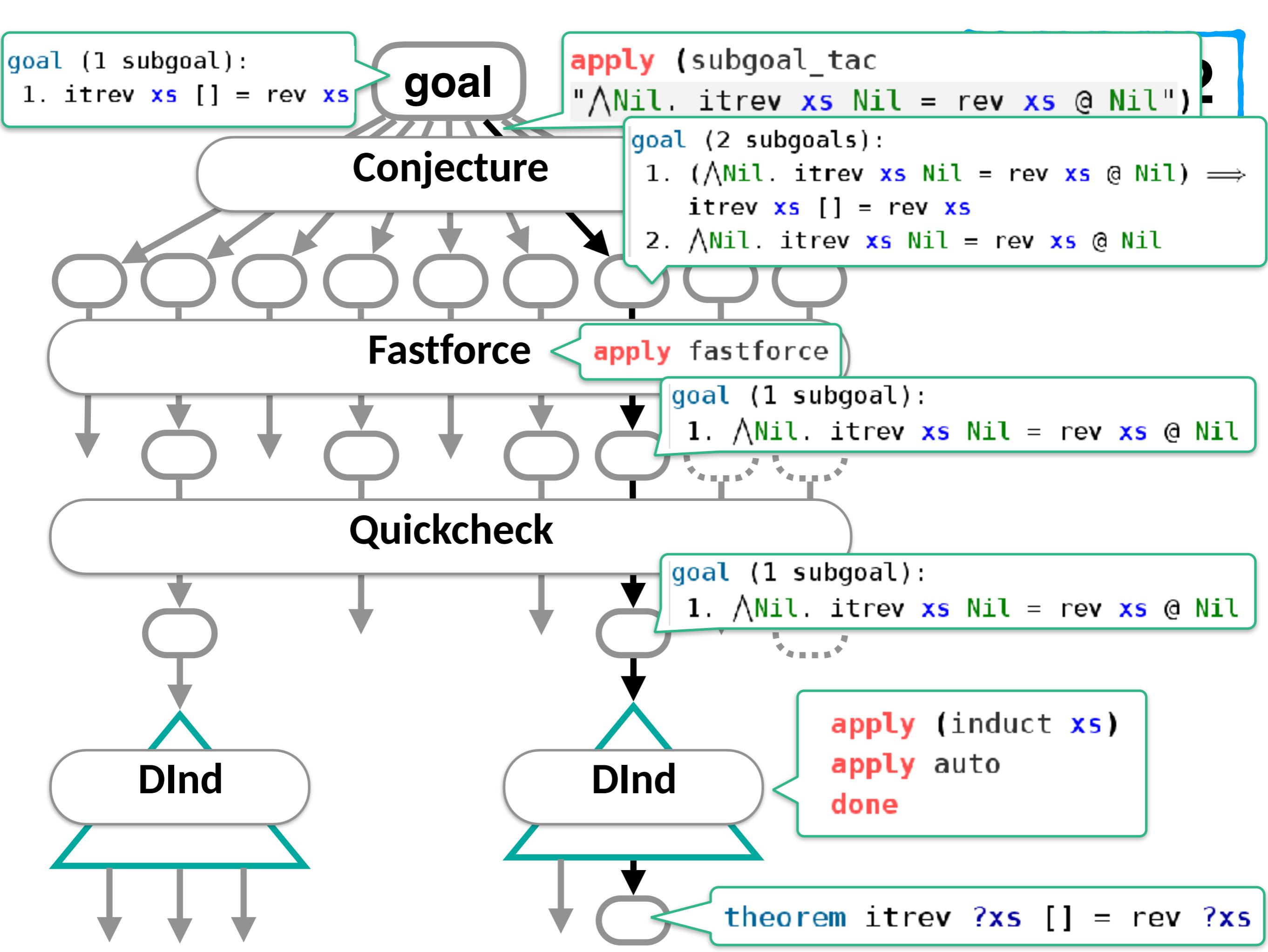


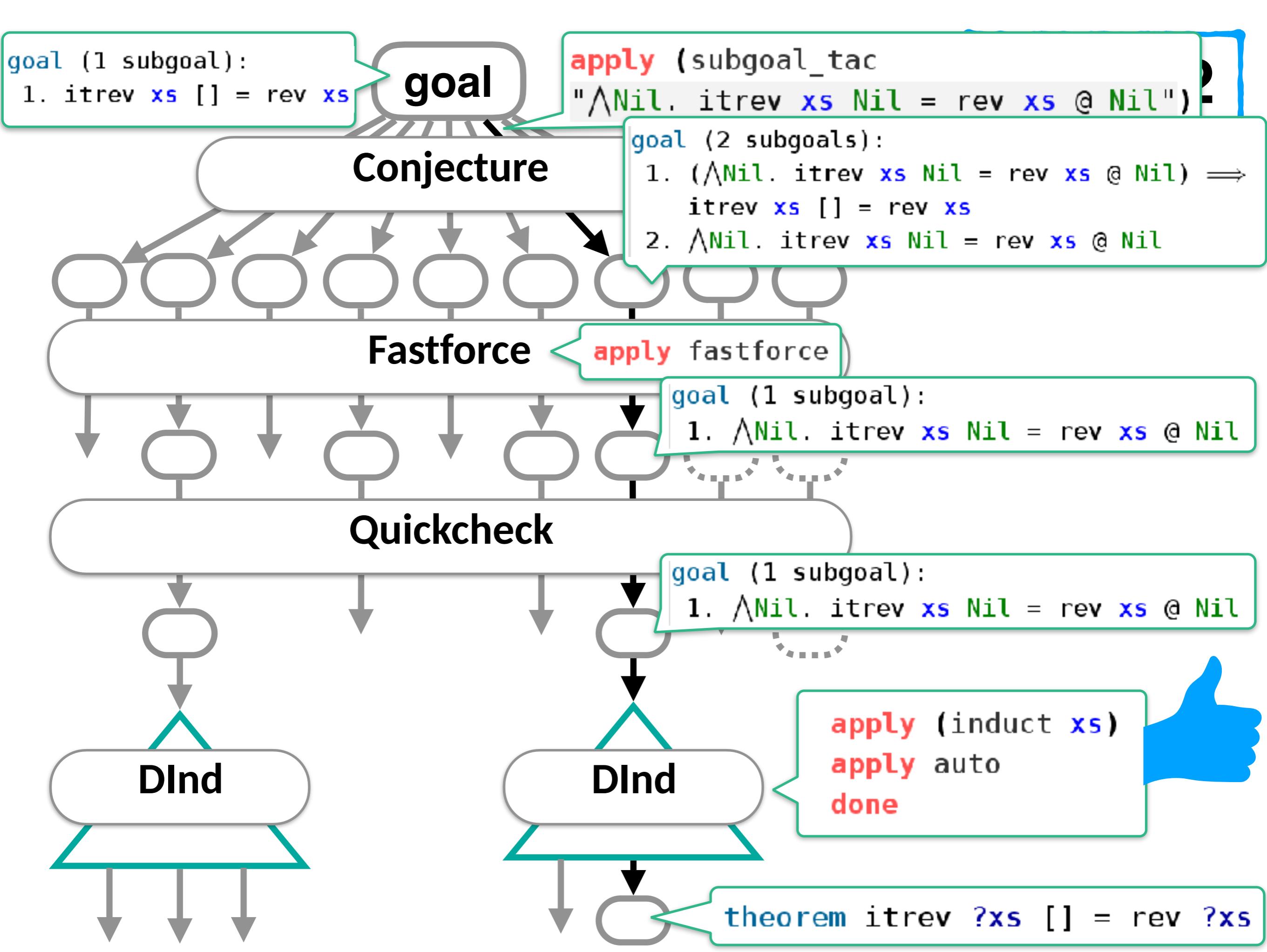






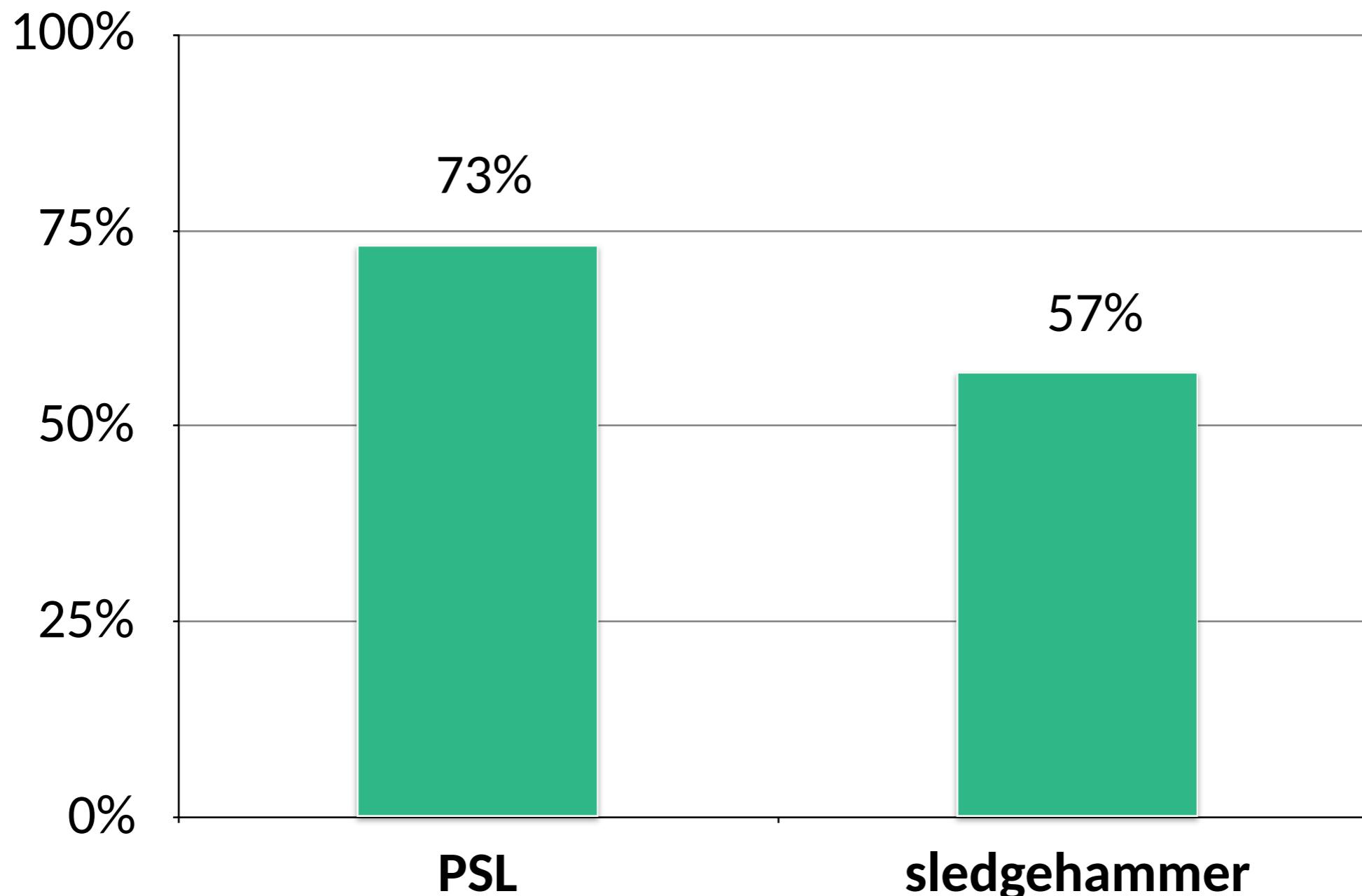






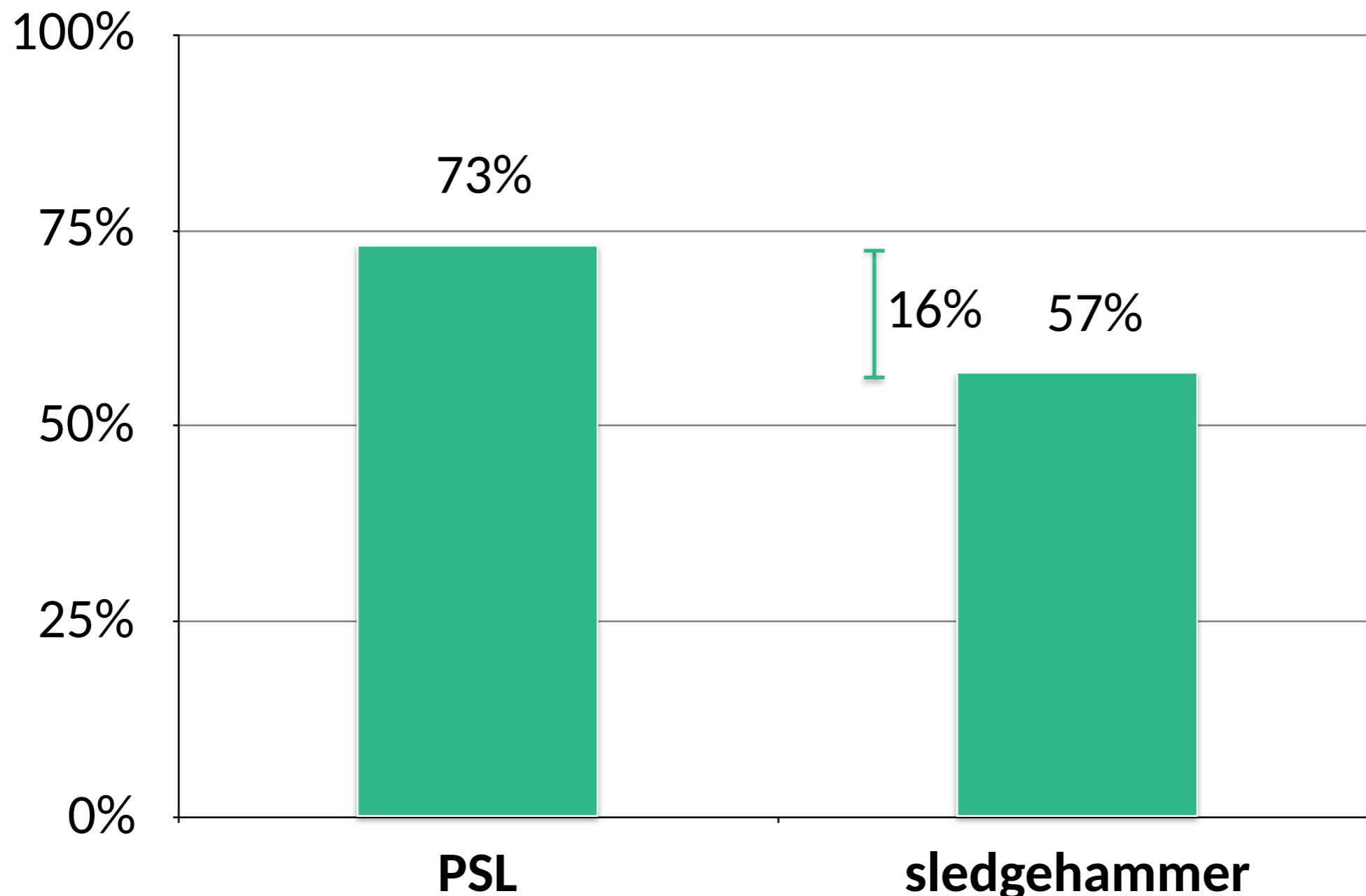
# PSL vs sledgehammer

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



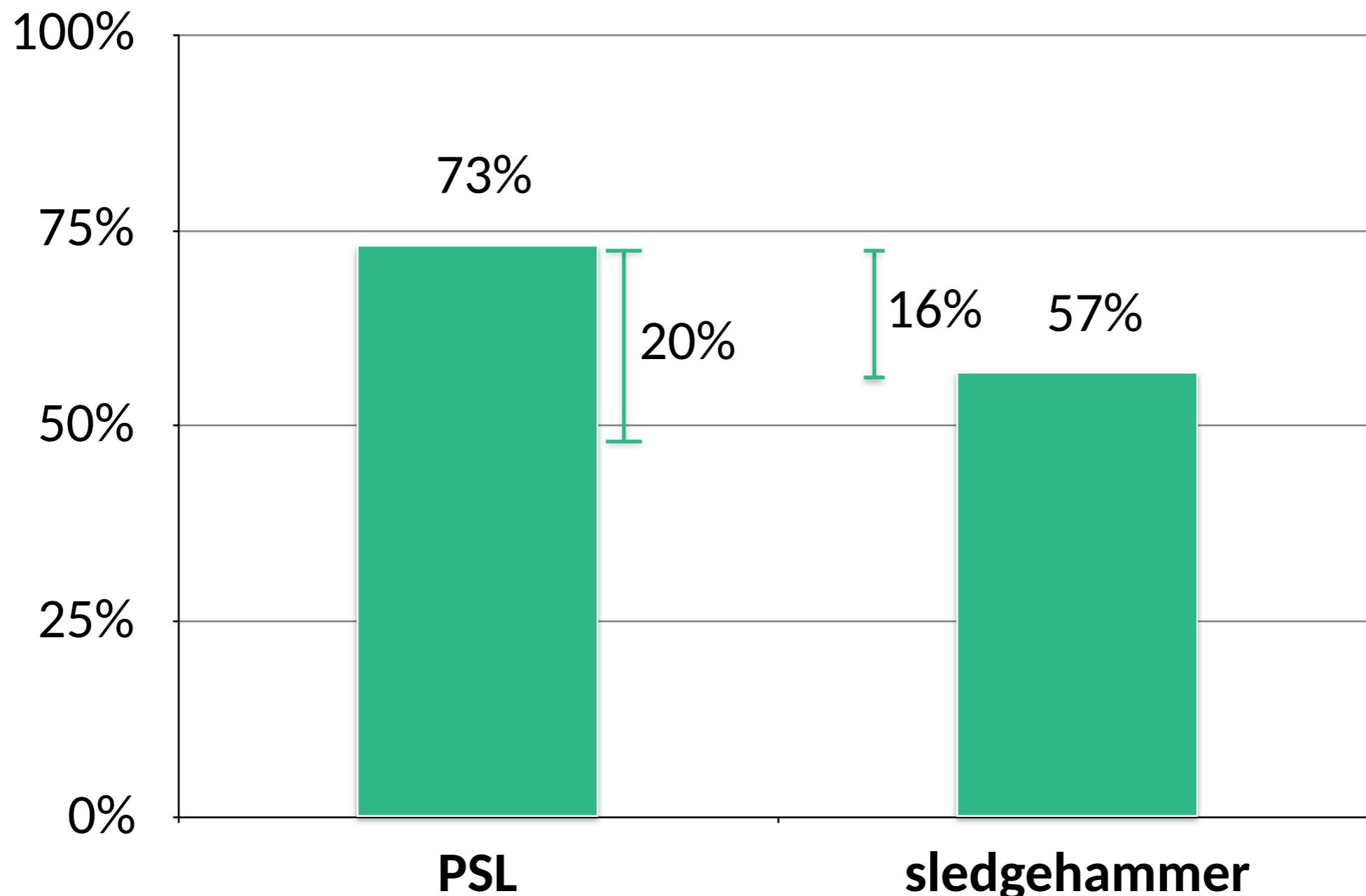
# PSL vs sledgehammer

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

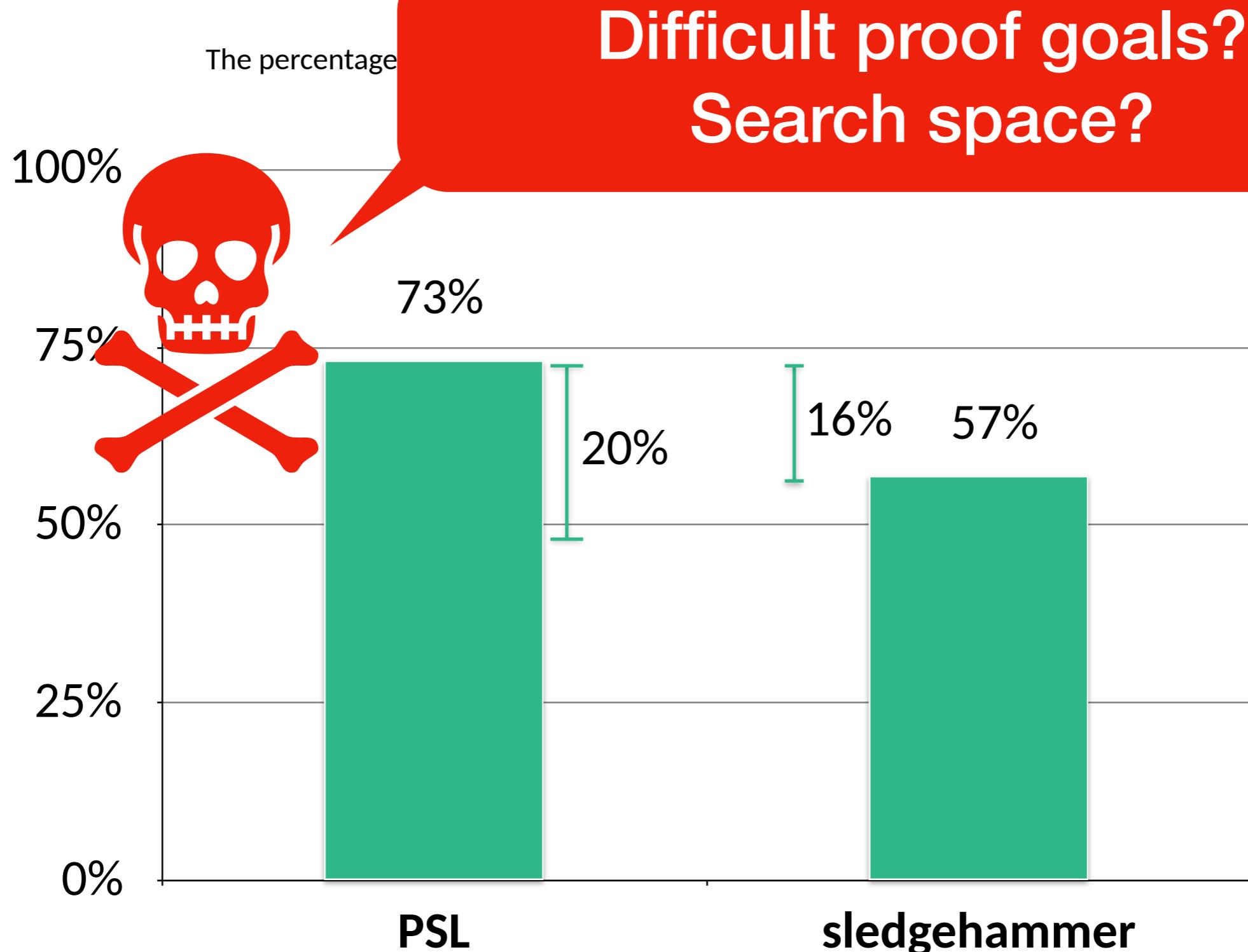


# PSL vs sledgehammer

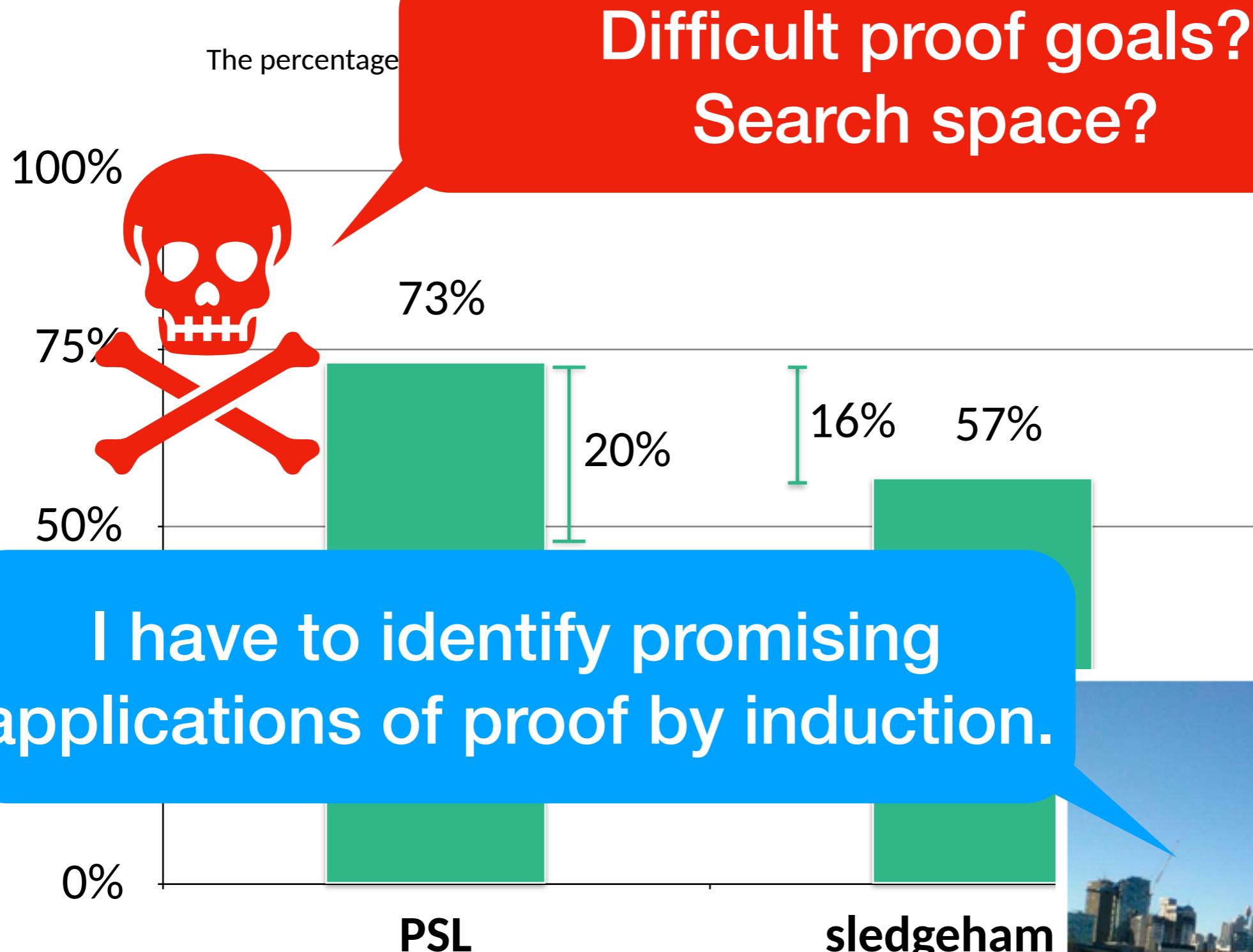
The percentage of automatically proved obligations out of 1526 proof obligations  
(timeout = 300s)



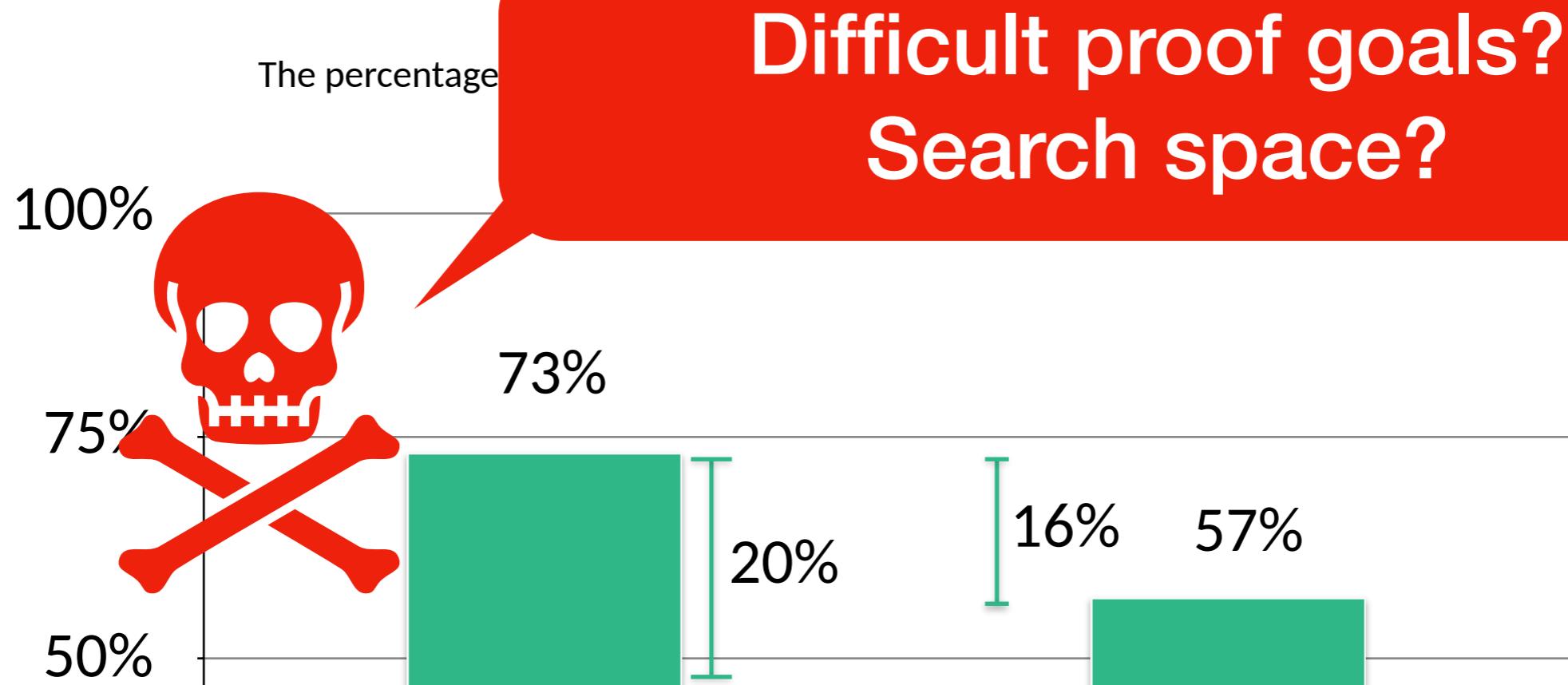
# PSL vs sledgehammer



# PSL vs sledgehammer



# PSL vs sleddaehammer



I have to identify promising applications of proof by induction.

*without completing a proof search*



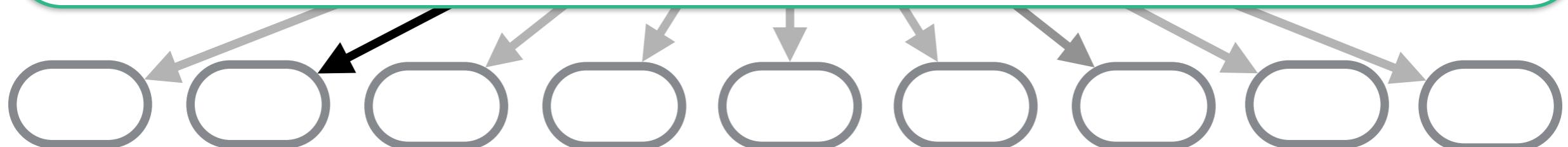
`smart_induct`

`goal`

# smart\_induct

goal

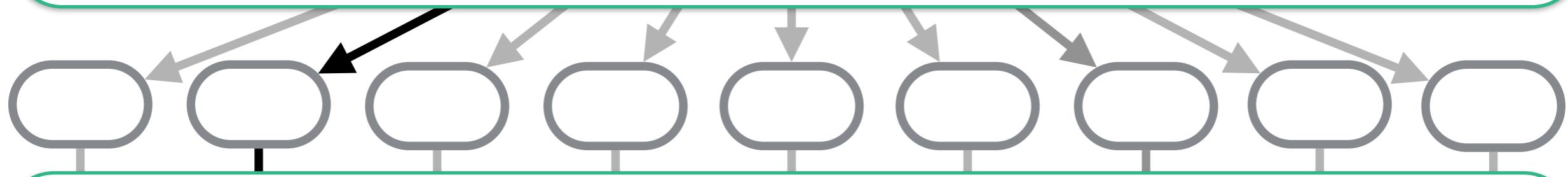
## Step 1: creating many inductions



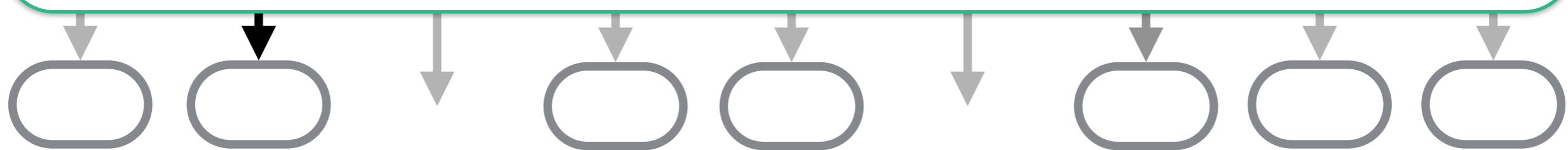
# smart\_induct

goal

**Step 1: creating many inductions**



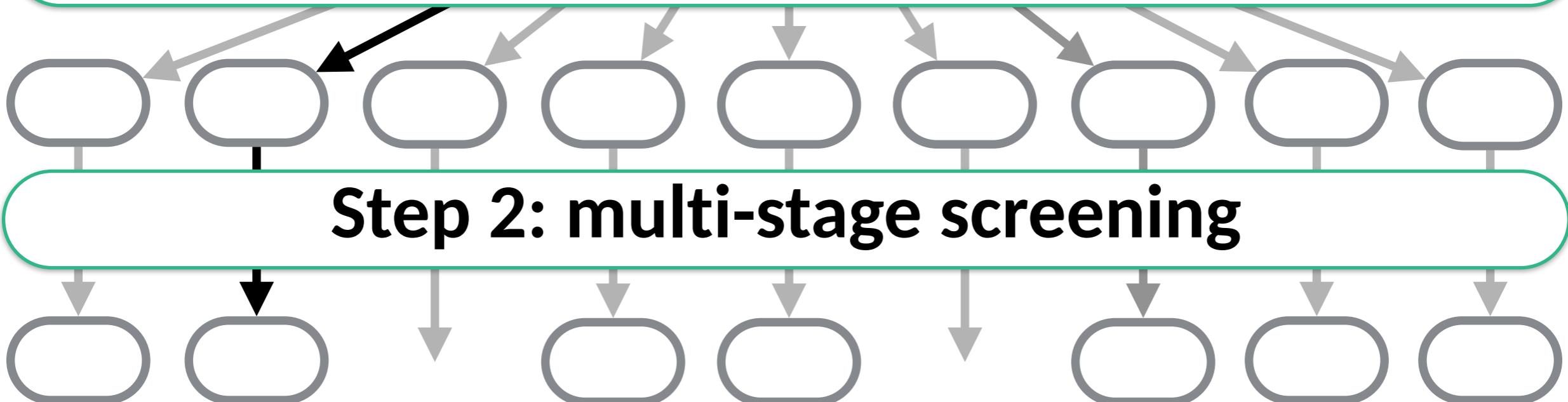
**Step 2: multi-stage screening**



# smart\_induct

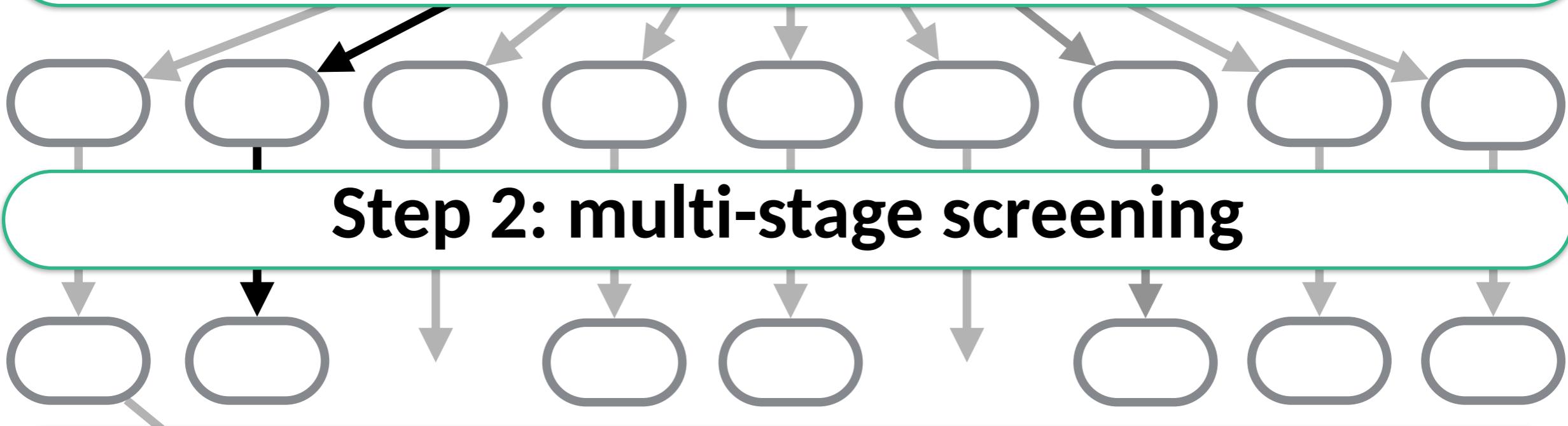
goal

**Step 1: creating many inductions**



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

### Step 1: creating many inductions



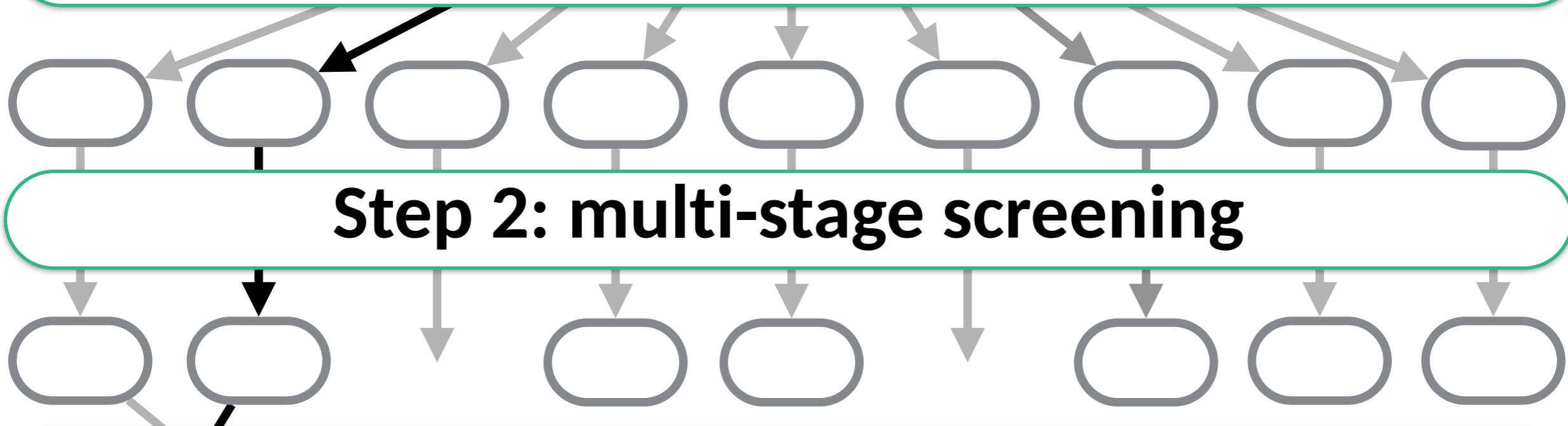
### Step 3: scoring using 20 heuristics and sorting

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

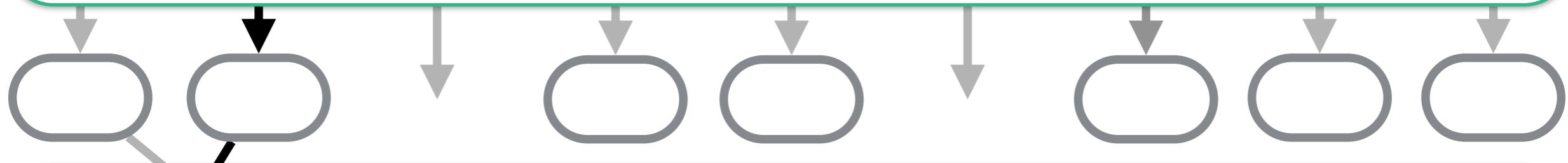
# smart\_induct

goal

## Step 1: creating many inductions



## Step 2: multi-stage screening



## Step 3: scoring using 20 heuristics and sorting

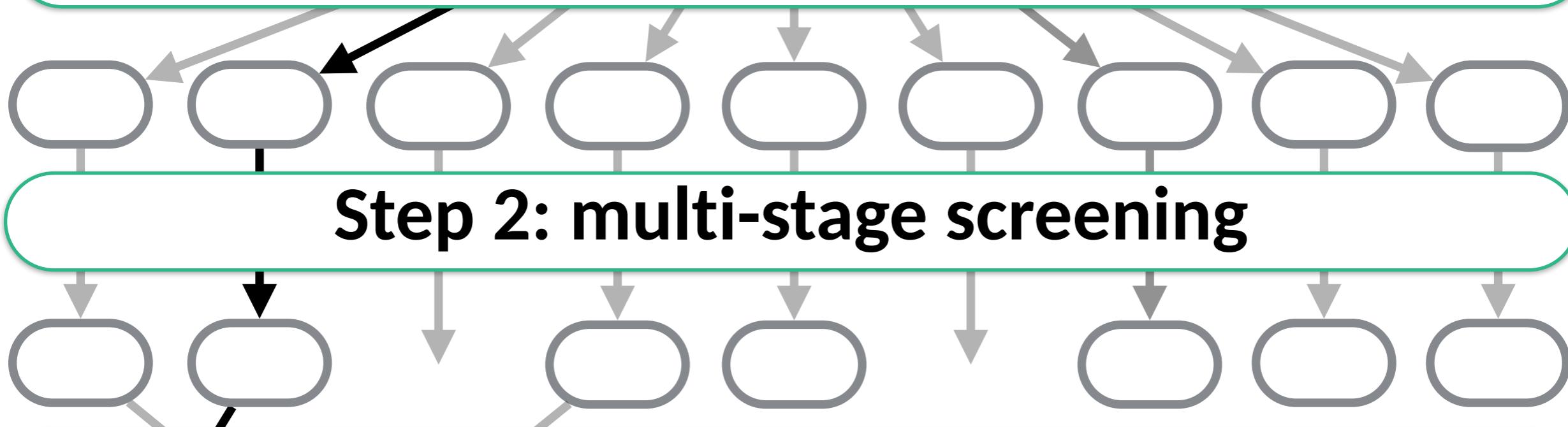


heuristic : ( proof goal \* induction arguments ) -> bool

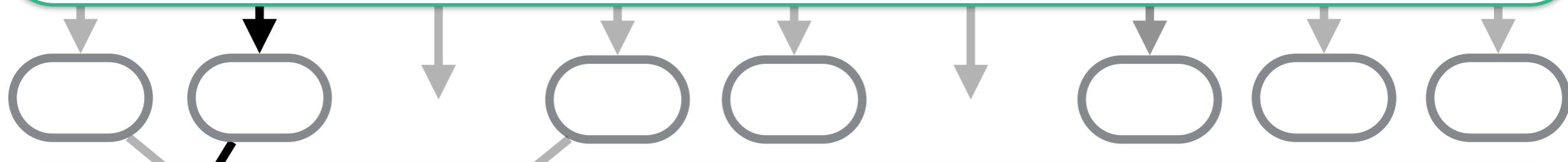
**smart\_induct**

**goal**

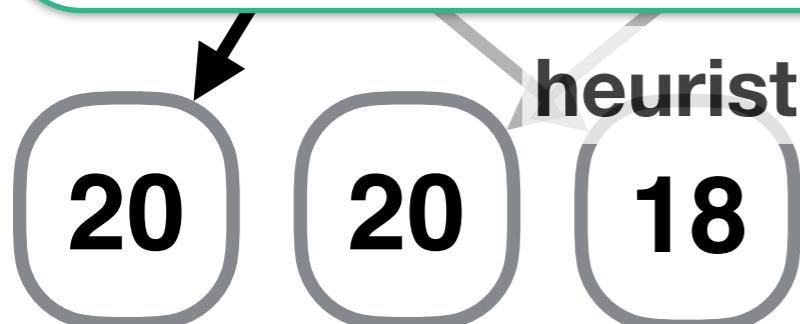
## Step 1: creating many inductions



## Step 2: multi-stage screening



## Step 3: scoring using 20 heuristics and sorting

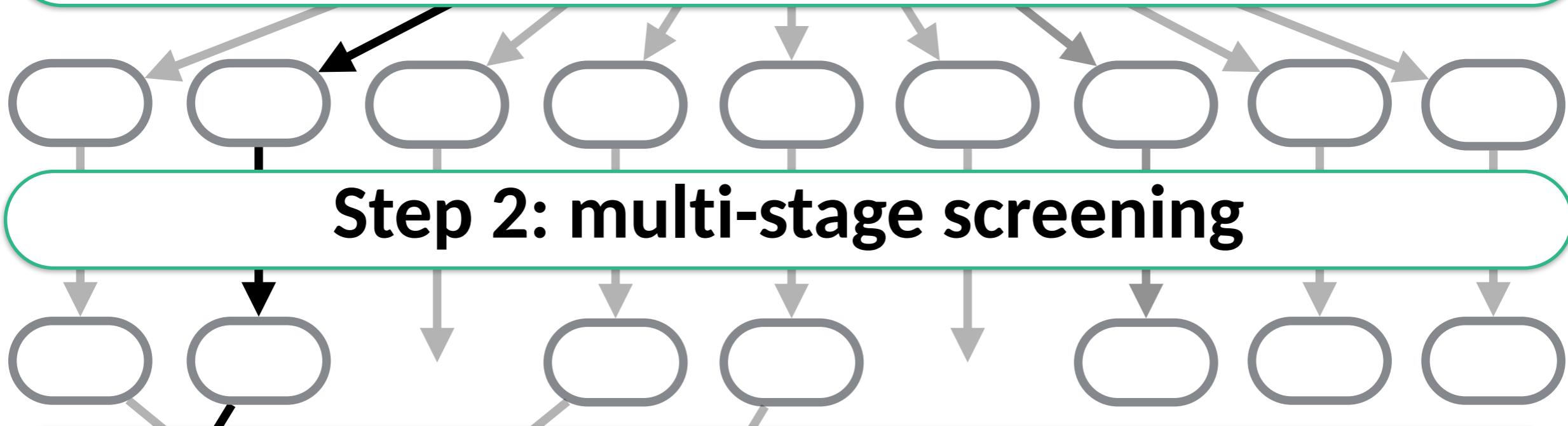


heuristic : ( proof goal \* induction arguments ) -> bool

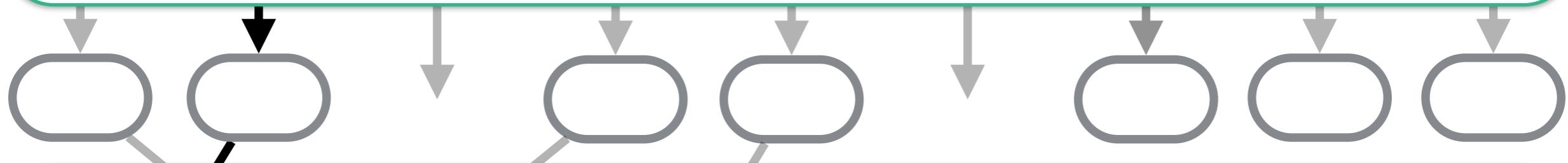
# smart\_induct

goal

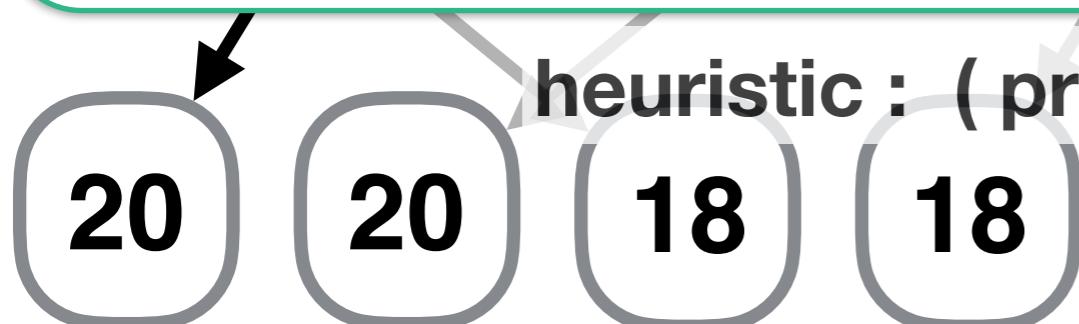
## Step 1: creating many inductions



## Step 2: multi-stage screening



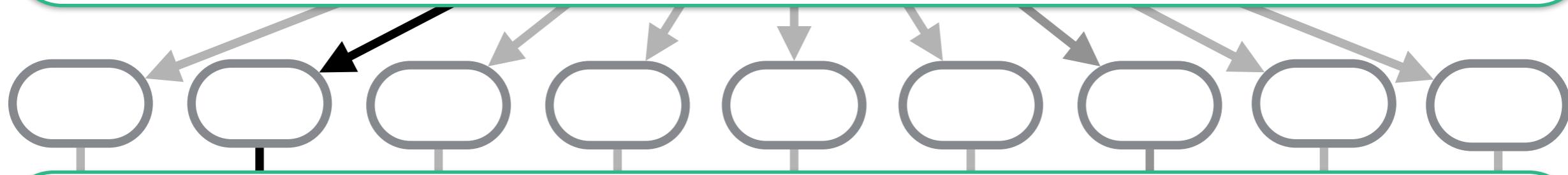
## Step 3: scoring using 20 heuristics and sorting



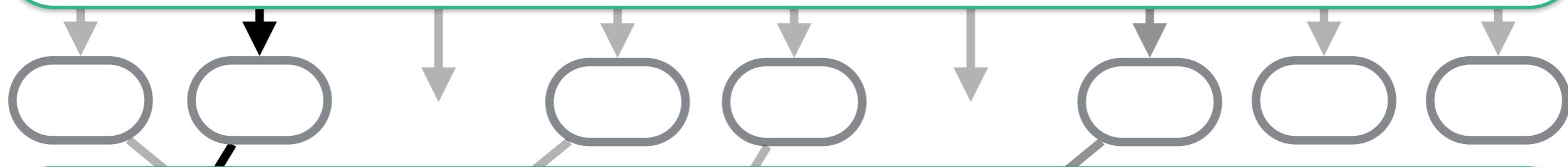
**smart\_induct**

**goal**

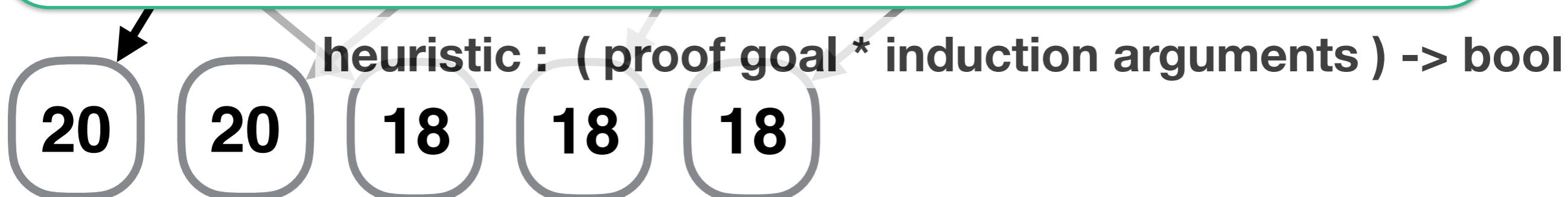
## Step 1: creating many inductions



## Step 2: multi-stage screening



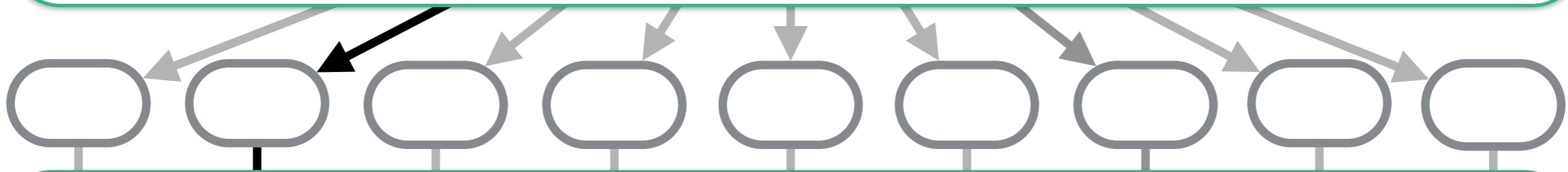
## Step 3: scoring using 20 heuristics and sorting



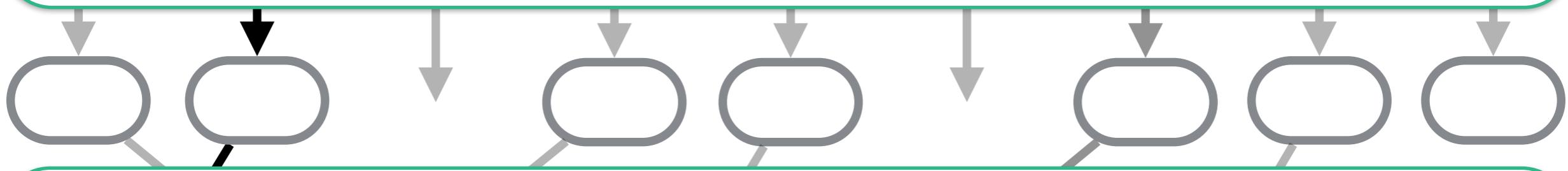
# smart\_induct

goal

## Step 1: creating many inductions



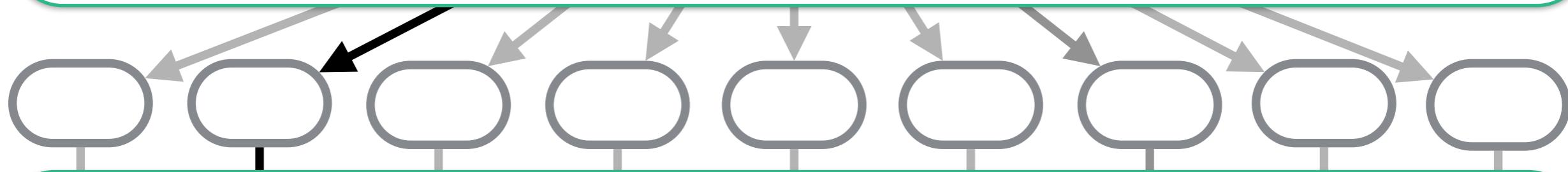
## Step 2: multi-stage screening



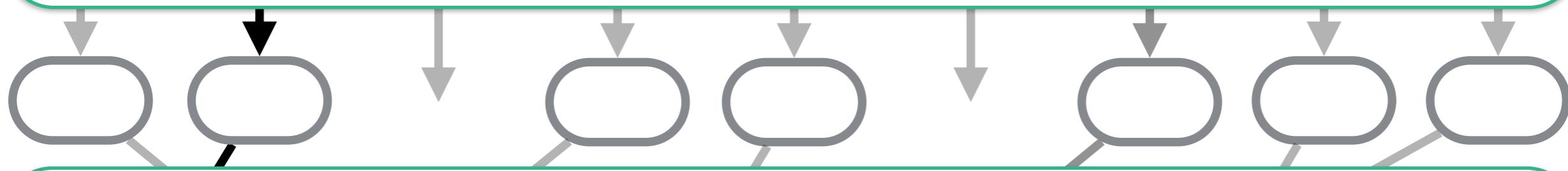
# smart\_induct

goal

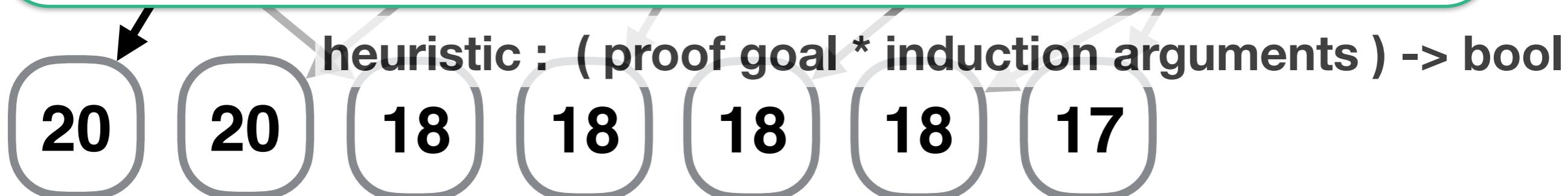
## Step 1: creating many inductions



## Step 2: multi-stage screening



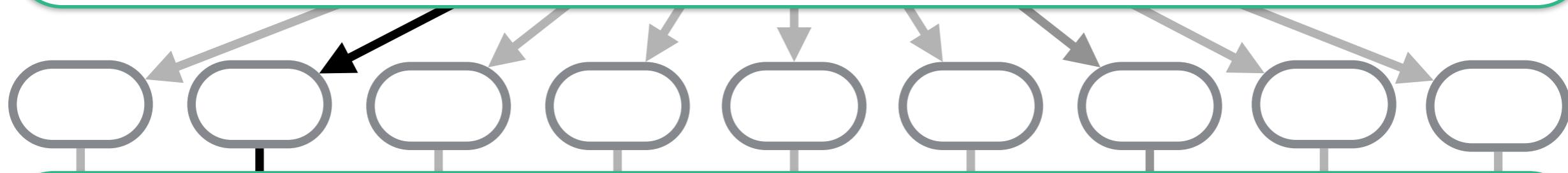
## Step 3: scoring using 20 heuristics and sorting



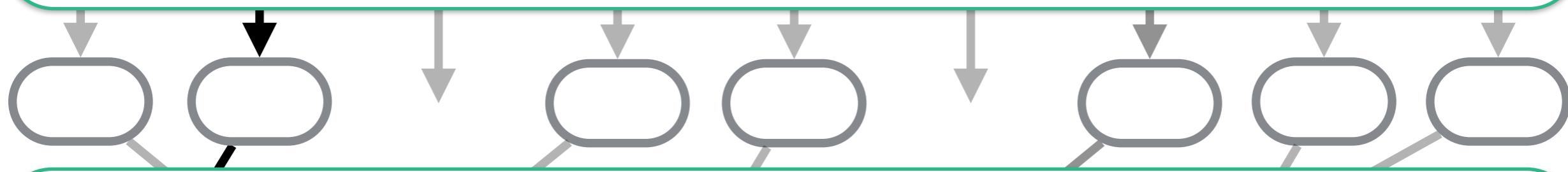
**smart\_induct**

**goal**

## Step 1: creating many inductions



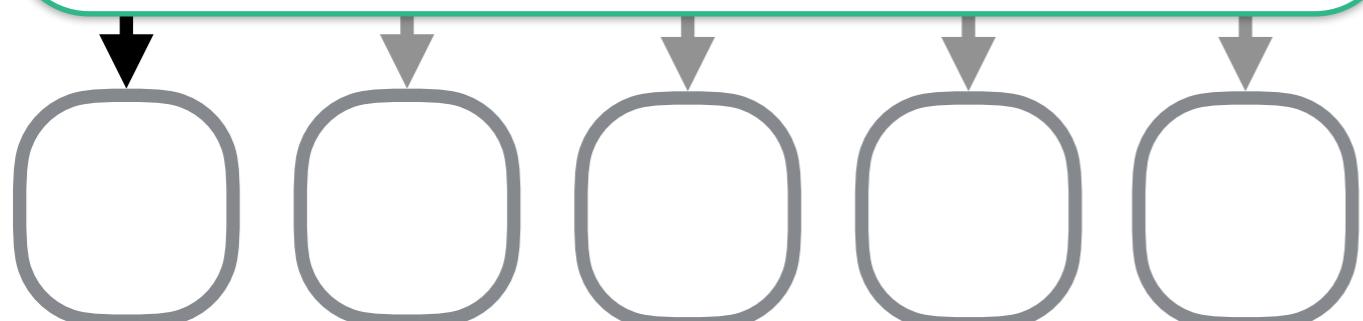
## Step 2: multi-stage screening



## Step 3: scoring using 20 heuristics and sorting



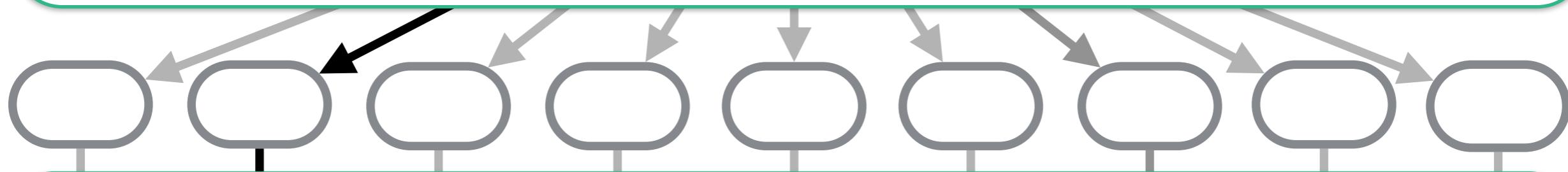
## Step 4: short-listing



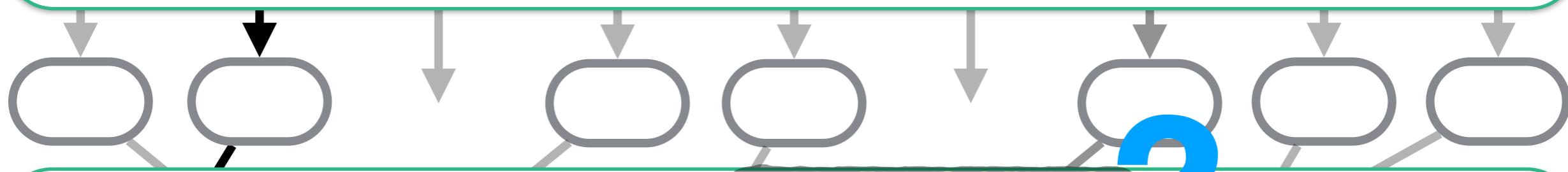
# smart\_induct

goal

## Step 1: creating many inductions



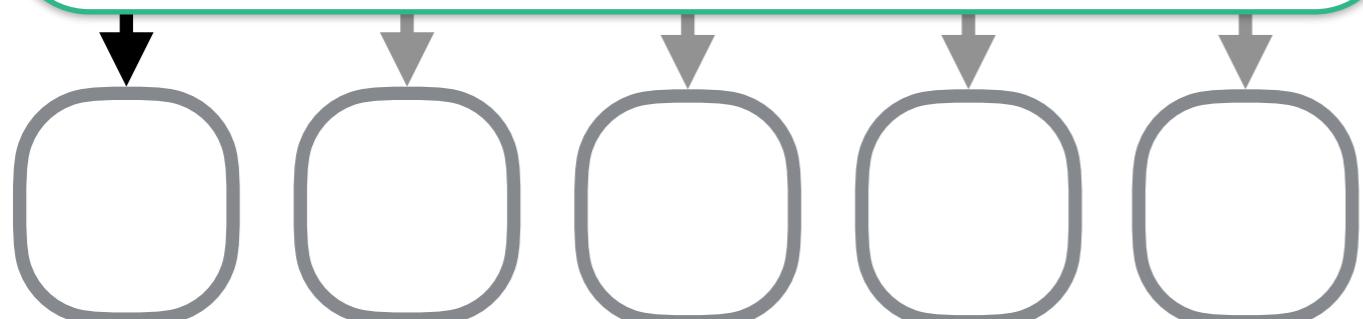
## Step 2: multi-stage screening



## Step 3: scoring using 20 heuristics and sorting

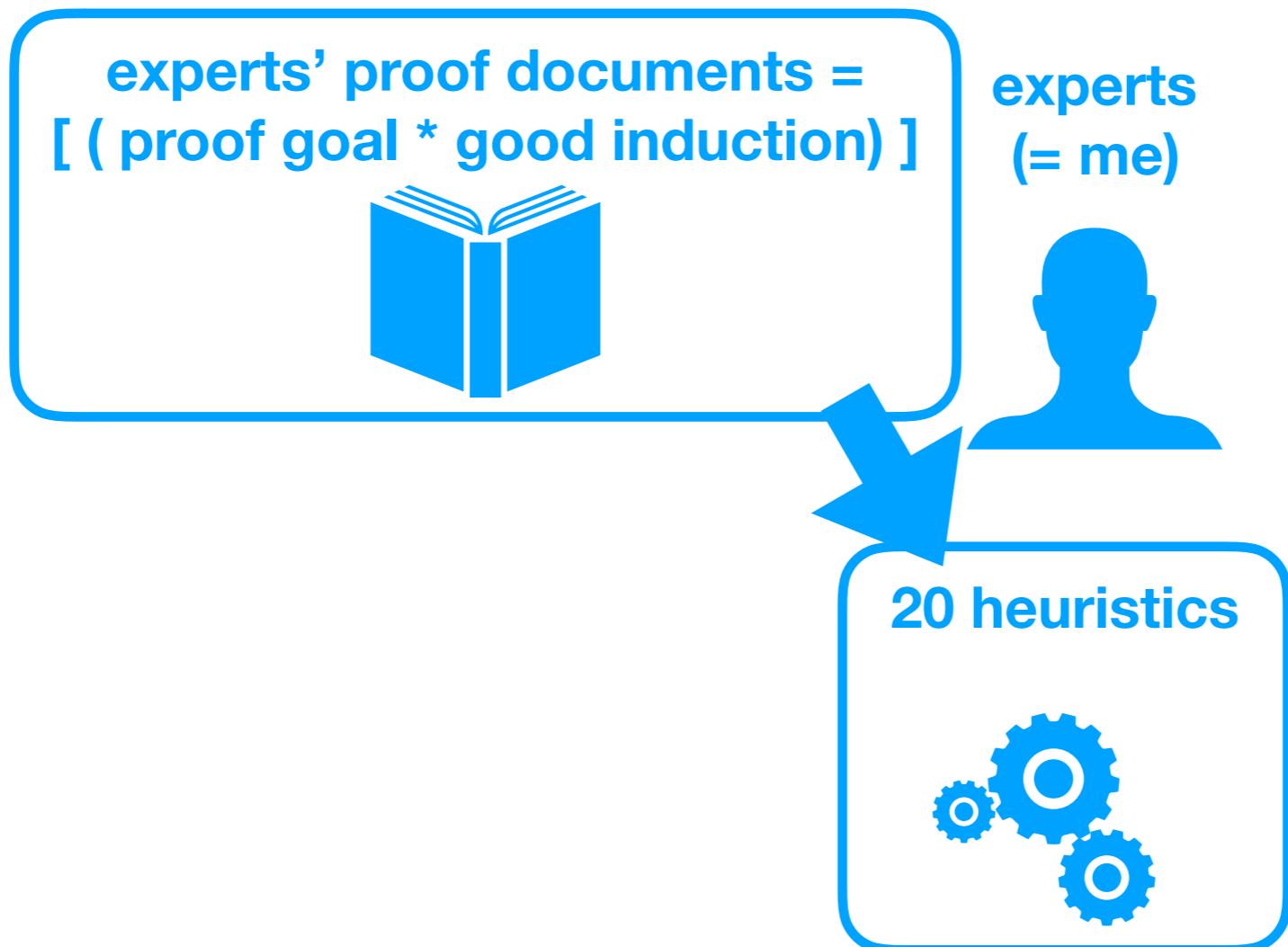


## Step 4: short-listing



**It is difficult to write induction heuristics.**

# It is difficult to write induction heuristics.



# It is difficult to write induction heuristics.

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

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



experts  
(= me)

20 heuristics



# It is difficult to write induction heuristics.

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

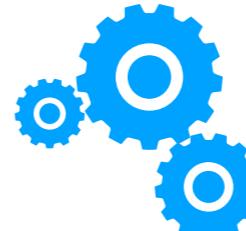
experts' proof documents =  
[ ( proof goal \* good induction) ]



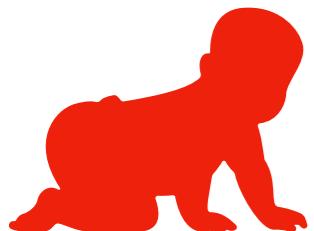
experts  
(= me)



20 heuristics



new users



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

# It is difficult to write induction heuristics.

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

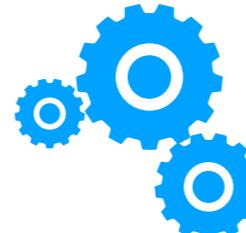
experts' proof documents =  
[ ( proof goal \* good induction) ]



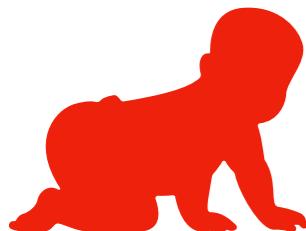
experts  
(= me)



20 heuristics



new users



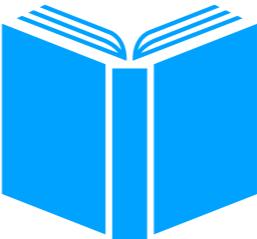
```
lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
```

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

# It is difficult to write induction heuristics.

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

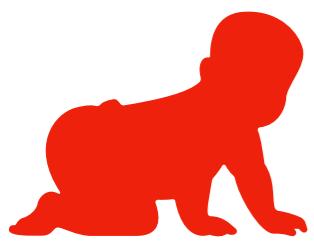
experts' proof documents =  
[ ( proof goal \* good induction) ]



experts  
(= me)

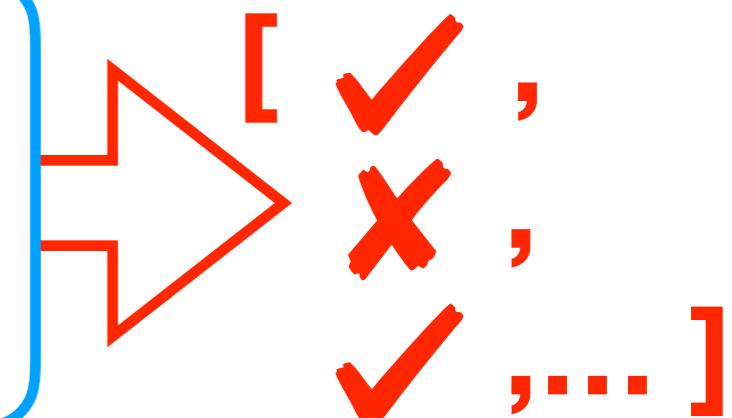
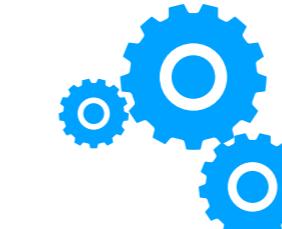


new users



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

20 heuristics



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

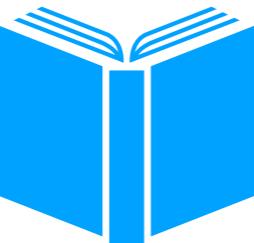
new proof goal consisting of new constants and variables of new types!

# It is difficult to write induction heuristics.

① blue = what I can see before releasing smart\_induct

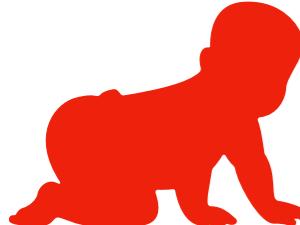
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

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

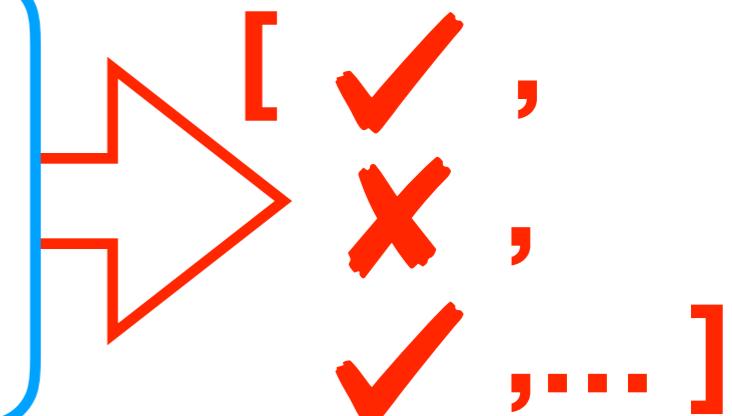
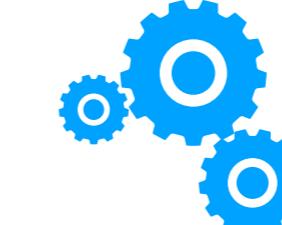


experts  
(= me)



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

20 heuristics



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

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

# It is difficult to write induction heuristics.

① blue = what I can see before releasing smart\_induct

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

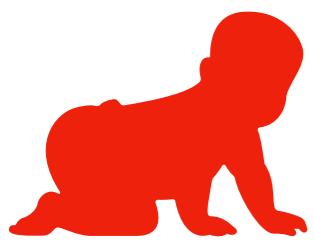
experts' proof documents =  
[ ( proof goal \* good induction) ]



experts  
(= me)

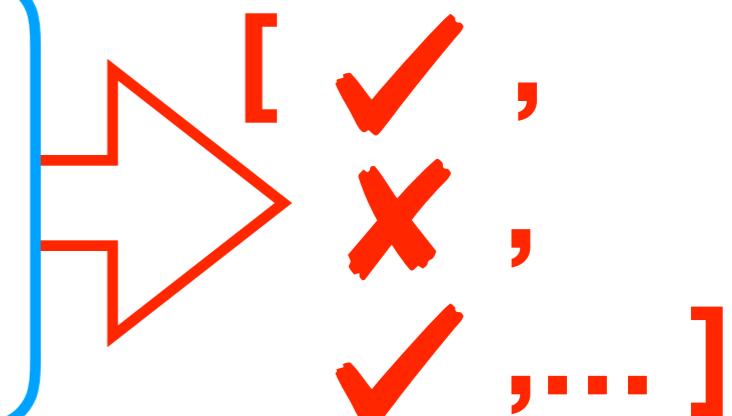
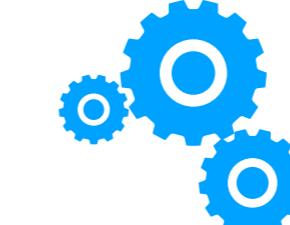


new users



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

20 heuristics



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

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

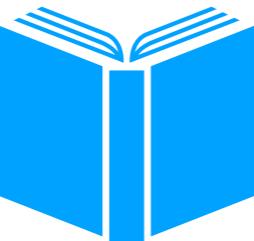
② red = what will be developed after releasing smart\_induct

# It is difficult to write induction heuristics.

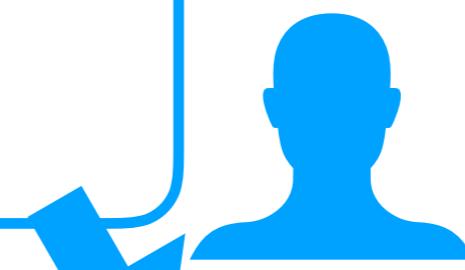
① blue = what I can see before releasing smart\_induct

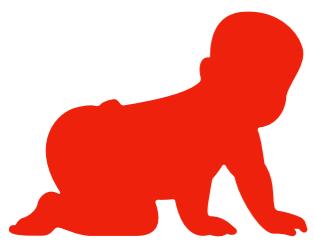
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

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

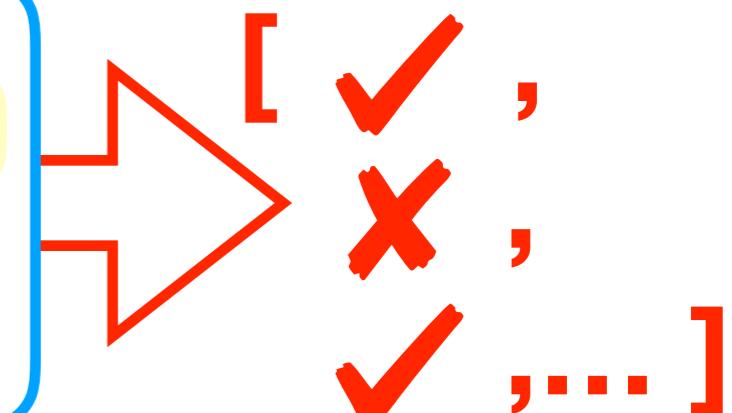


experts  
(= me)



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

20 heuristics  
in LiFtEr



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

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

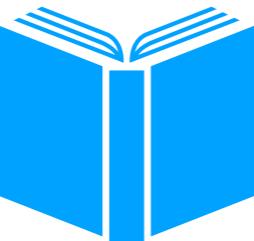
② red = what will be developed after releasing smart\_induct

# It is difficult to write induction heuristics.

① blue = what I can see before releasing smart\_induct

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

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

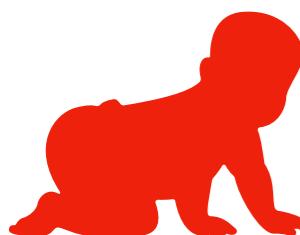


experts  
(= me)

LiFtEr: Logical  
Feature Extractor

new users

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



20 heuristics  
in LiFtEr



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

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

② red = what will be developed after releasing smart\_induct

# Abstract Syntax of LiFtEr

```
assertion := atomic | connective | quantifier | ( assertion )
type := term | term_occurrence | rule | number
modifier_term := induction_term | arbitrary_term
quantifier :=  $\exists x : type . assertion$ 
            |  $\forall x : type . assertion$ 
            |  $\exists x : term \in modifier\_term . assertion$ 
            |  $\forall x : term \in modifier\_term . assertion$ 
            |  $\exists x : rule . assertion$ 
            |  $\exists x : term\_occurrence \in y : term . assertion$ 
            |  $\forall x : term\_occurrence \in y : term . assertion$ 
connective := True | False | assertion  $\vee$  assertion | assertion  $\wedge$  assertion
            | assertion  $\rightarrow$  assertion |  $\neg$  assertion
pattern := all_only_var | all_constructor | mixed
```

atomic :=

```
rule is_rule_of term_occurrence
| term_occurrence term_occurrence_is_of_term term
| are_same_term ( term_occurrence , term_occurrence )
| term_occurrence is_in_term_occurrence term_occurrence
| is_atomic term_occurrence
| is_constant term_occurrence
| is_recursive_constant term_occurrence
| is_variable term_occurrence
| is_free_variable term_occurrence
| is_bound_variable term_occurrence
| is_lambda term_occurrence
| is_application term_occurrence
| term_occurrence is_an_argument_of term_occurrence
| term_occurrence is_nth_argument_of term_occurrence
```

# Abstract Syntax of LiFtEr

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

# Example Assertion in LiFtEr (in Abstract Syntax)

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

# Example Assertion in LiFtEr (in Abstract Syntax)

implication



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

# Example Assertion in LiFtEr (in Abstract Syntax)

implication

↓

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

→

$\exists r1 : \text{rule}.$   
 $\exists t1 : \text{term}.$   
 $\exists to1 : \text{term\_occurrence} \in t1 : \text{term}.$   
     $r1 \text{ is\_rule\_of } to1$

$\wedge$  ↪ conjunction

$\forall t2 : \text{term} \in \text{induction\_term}.$   
 $\exists to2 : \text{term\_occurrence} \in t2 : \text{term}.$   
     $\exists n : \text{number}.$   
         $\text{is\_nth\_argument\_of} (to2, n, to1)$

$\wedge$

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

# Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

$\exists r1 : \text{rule}.$  ←

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

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

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

$\wedge$  ← conjunction

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

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

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

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

$\wedge$

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

# Example Assertion in LiFtEr (in Abstract Syntax)

implication

↓  
 $\exists r1 : \text{rule}. \text{True}$  → variable for auxiliary lemmas  
 $\exists r1 : \text{rule}.$  ←  
 $\exists t1 : \text{term}.$  ← variable for terms  
 $\exists to1 : \text{term\_occurrence} \in t1 : \text{term}.$   
     $r1 \text{ is\_rule\_of } to1$   
    ∧ conjunction  
     $\forall t2 : \text{term} \in \text{induction\_term}.$   
         $\exists to2 : \text{term\_occurrence} \in t2 : \text{term}.$   
             $\exists n : \text{number}.$   
                 $\text{is\_nth\_argument\_of } (to2, n, to1)$   
             $\wedge$   
                 $t2 \text{ is\_nth\_induction\_term } n$

# Example Assertion in LiFtEr (in Abstract Syntax)

implication

↓  
 $\exists r1 : \text{rule}. \text{True}$  → variable for auxiliary lemmas  
 $\exists r1 : \text{rule}.$  ←  
 $\exists t1 : \text{term}.$  ← variable for terms  
 $\exists to1 : \text{term\_occurrence} \in t1 : \text{term}.$   
     $r1 \text{ is\_rule\_of } to1$  ← variable for term occurrences  
    ∧ ← conjunction  
     $\forall t2 : \text{term} \in \text{induction\_term}.$   
         $\exists to2 : \text{term\_occurrence} \in t2 : \text{term}.$   
             $\exists n : \text{number}.$   
                 $\text{is\_nth\_argument\_of } (to2, n, to1)$   
            ∧  
                 $t2 \text{ is\_nth\_induction\_term } n$

# Example Assertion in LiFtEr (in Abstract Syntax)

implication

↓  
 $\exists r1 : \text{rule}. \text{True}$  → variable for auxiliary lemmas  
 $\exists r1 : \text{rule}.$  ←  
 $\exists t1 : \text{term}.$  ← variable for terms  
 $\exists to1 : \text{term\_occurrence} \in t1 : \text{term}.$   
     $r1 \text{ is\_rule\_of } to1$  ← variable for term occurrences  
    ∧ ← conjunction  
     $\forall t2 : \text{term} \in \text{induction\_term}.$   
         $\exists to2 : \text{term\_occurrence} \in t2 : \text{term}.$   
             $\exists n : \text{number}.$  ← variable for natural numbers  
                 $\text{is\_nth\_argument\_of } (to2, n, to1)$   
             $\wedge$   
                 $t2 \text{ is\_nth\_induction\_term } n$

# Example Assertion in LiFtEr (in Abstract Syntax)

implication

↓  
 $\exists r1 : \text{rule}. \text{True}$  → variable for auxiliary lemmas  
 $\exists r1 : \text{rule}.$  ←  
 $\exists t1 : \text{term}.$  ← variable for terms  
 $\exists to1 : \text{term\_occurrence} \in t1 : \text{term}.$  ← variable for term occurrences  
     $r1 \text{ is\_rule\_of } to1$  ←  
 $\wedge$  conjunction  
     $\forall t2 : \text{term} \in \text{induction\_term}.$   
         $\exists to2 : \text{term\_occurrence} \in t2 : \text{term}.$   
             $\exists n : \text{number}.$  ← variable for natural numbers  
                 $\text{is\_nth\_argument\_of } (to2, n, to1)$   
             $\wedge$   
                 $t2 \text{ is\_nth\_induction\_term } n$

universal quantifier

# Example Assertion in LiFtEr (in Abstract Syntax)

implication existential quantifier

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

$\rightarrow$  variable for auxiliary lemmas

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

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

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

$r1 \text{ is\_rule\_of } to1$  variable for term occurrences

$\wedge$  conjunction

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

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

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

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

$\wedge$

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

universal quantifier

# Example Assertion in LiFtEr (in Abstract Syntax)

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

implication existential quantifier

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

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

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

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

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

$\wedge$  conjunction

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

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

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

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

$\wedge$

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

```

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

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

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

```

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

$\rightarrow$

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

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

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

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

$\wedge$

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

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

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

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

$\wedge$

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

```

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

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

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

```

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

→

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

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

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

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

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

$\wedge$

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

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

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

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

$\wedge$

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

r1

```

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

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

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

```

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

$\rightarrow$

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

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

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

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

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

$\wedge$

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

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

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

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

$\wedge$

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

$r1$

```

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

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

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

```

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

$\rightarrow$

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

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

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

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

$\wedge$

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

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

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

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

$\wedge$

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

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

$r1$

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

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

to1

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

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

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

$\rightarrow$

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

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

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

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

$\wedge$

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

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

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

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

$\wedge$

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

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

r1

```

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

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

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

```

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

$\rightarrow$

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

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

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

$r1 \text{ is\_rule\_of } tol$

$\wedge$

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

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

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

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

$\wedge$

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

$tol$

$r1$

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

( $t1 = \text{itrev}$ )

( $tol = \text{itrev}$ )

```
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)"
```

to1

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

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

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

$\rightarrow$

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

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

( $t1 = \text{itrev}$ )

( $to1 = \text{itrev}$ )

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

r1

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

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

$\wedge$

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

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

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

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

$\wedge$

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

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

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

to1

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

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

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

$\rightarrow$

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

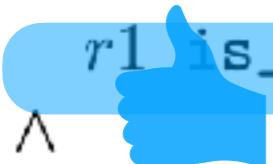
( $r1 = \text{itrev.induct}$ )

( $t1 = \text{itrev}$ )

( $to1 = \text{itrev}$ )

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

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

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

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

$\wedge$

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

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

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

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

$\wedge$

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

$r1$

```

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

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

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

```

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

$\rightarrow$

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

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

( $t1 = \text{itrev}$ )

( $to1 = \text{itrev}$ )

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

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

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

$\wedge$

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

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

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

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

$\wedge$

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

$to1$

$r1$



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

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

to1

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

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

$\rightarrow$

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

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

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

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

$\wedge$

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

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

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

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

$\wedge$

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

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

( $t1 = \text{itrev}$ )

( $to1 = \text{itrev}$ )

$r1$

```

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

```

```

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

```

$t_{01}$

```

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

```

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

$t_2$

$r_1$

→

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

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

( $t_1 = \text{itrev}$ )

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

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

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

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

^

$\forall t_2 : \text{term} \in \text{induction\_term}.$  ( $t_2 = \text{xs and ys}$ )

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

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

$\text{is\_nth\_argument\_of } (t_{02}, n, t_{01})$

^

$t_2 \text{is\_nth\_induction\_term } n$

```

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

```

```

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

```

$t_{01}$

$t_{02}$

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

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

$t_2$

$r_1$

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

$\rightarrow$

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

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

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

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

$\wedge$

$\forall t_2 : \text{term} \in \text{induction\_term}.$

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

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

$\text{is\_nth\_argument\_of} (t_{02}, n, t_{01})$

$\wedge$

$t_2 \text{ is\_nth\_induction\_term } n$

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

( $t_1 = \text{itrev}$ )

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

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

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

```

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

```

```

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

```

$t_{01}$

$t_{02}$

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

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

$t_2$

$r_1$

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

$\rightarrow$

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

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

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

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



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

( $t_1 = \text{itrev}$ )

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

$\forall t_2 : \text{term} \in \text{induction\_term}.$

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

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

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

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

$\text{is\_nth\_argument\_of } (t_{02}, n, t_{01})$

$\wedge$

$t_2 \text{ is\_nth\_induction\_term } n$

```

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

```

```

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

```

$t_{01}$

$t_{02}$

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

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

$t_2$

$r_1$

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

$\rightarrow$

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

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

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

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

$\wedge$

$\forall t_2 : \text{term} \in \text{induction\_term}.$

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

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

$\text{is\_nth\_argument\_of } (t_{02}, n, t_{01})$

$\wedge$

$t_2 \text{ is\_nth\_induction\_term } n$

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

( $t_1 = \text{itrev}$ )

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

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

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

```

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

```

```

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

```

$t_{01}$

$\text{first}$

$t_{02}$

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

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

$t_2$

$\text{first}$

$r_1$

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

$\rightarrow$

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

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

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

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

$\wedge$

$\forall t_2 : \text{term} \in \text{induction\_term}.$

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

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

$\text{is\_nth\_argument\_of } (t_{02}, n, t_{01})$

$\wedge$

$t_2 \text{ is\_nth\_induction\_term } n$

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

( $t_1 = \text{itrev}$ )

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

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

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

when  $t_2$  is  $\text{xs}$  ( $n = 1$ ) ?

```

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

```

```

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

```

$t_{01}$

$\text{first}$

$t_{02}$

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

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

$t_2$

$\text{first}$

$r_1$

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

$\rightarrow$

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

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

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

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

$\wedge$

$\forall t_2 : \text{term} \in \text{induction\_term}.$

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

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

$\text{is\_nth\_argument\_of } (t_{02}, n, t_{01})$

$\wedge$

$t_2 \text{ is\_nth\_induction\_term } n$

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

( $t_1 = \text{itrev}$ )

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

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

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

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

```

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

```

```

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

```

$t_{01}$

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

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

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

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

$\rightarrow$

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

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

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

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

$\wedge$

$\forall t_2 : \text{term} \in \text{induction\_term}.$

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

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

$\text{is\_nth\_argument\_of } (t_{02}, n, t_{01})$

$\wedge$

$t_2 \text{ is\_nth\_induction\_term } n$

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

( $t_1 = \text{itrev}$ )

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

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

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

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

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

```

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

```

```

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

```

$t_{01}$

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

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

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

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

$r_1$

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

$\rightarrow$

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

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

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

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

$\wedge$

$\forall t_2 : \text{term} \in \text{induction\_term}.$

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

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

$\text{is\_nth\_argument\_of } (t_{02}, n, t_{01})$

$\wedge$

$t_2 \text{ is\_nth\_induction\_term } n$

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

( $t_1 = \text{itrev}$ )

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

( $t_2 = xs$  and  $ys$ )

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

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

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

```

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

```

```

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

```

$t_{01}$

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

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

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

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

$r_1$

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

$\rightarrow$

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

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

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

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

$\wedge$

$\forall t_2 : \text{term} \in \text{induction\_term}.$

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

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

$\text{is\_n}^{\text{th}}\text{\_argument\_of } (t_{02}, n, t_{01})$

$\wedge$

$t_2 \text{ is\_n}^{\text{th}}\text{ induction term } n$

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

( $t_1 = \text{itrev}$ )

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

( $t_2 = xs$  and  $ys$ )

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

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

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

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

$\rightarrow$

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

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

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

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

$\wedge$

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

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

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

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

$\wedge$

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

the same LIFTER assertion



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

new types ->

**datatype** instr = LOADI val | LOAD vname | ADD  
**type\_synonym** stack = "val list"

the same LIFTER assertion



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

$\rightarrow$

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

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

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

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

$\wedge$

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

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

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

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

$\wedge$

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

new types ->

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

new constants ->

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

the same LIFTER assertion

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



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

$\rightarrow$

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

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

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

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

$\wedge$

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

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

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

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

$\wedge$

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

new types ->

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

new constants ->

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

the same LIFTER assertion

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

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

→

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

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

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

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

$\wedge$

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

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

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

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

$\wedge$

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

new types ->

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

new constants ->

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

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

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

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

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

→

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

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

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

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

$\wedge$

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

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

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

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

$\wedge$

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

new types ->

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

new constants ->

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

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

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

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

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

r1

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

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

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

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

$\wedge$

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

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

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

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

$\wedge$

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

new types ->

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

new constants ->

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

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

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

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

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

r1

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

( r1 = exec.induct )

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

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

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

$\wedge$

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

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

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

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

$\wedge$

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

new types ->

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

new constants ->

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

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

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

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

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

r1

→

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

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

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

( r1 = exec.induct )  
( t1 = exec )

r1 is\_rule\_of tol

^

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

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

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

is\_nth\_argument\_of (to2, n, tol)

^

t2 is\_nth\_induction\_term n

new types ->

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

new constants ->

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

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

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

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

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

to1

r1

→

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

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

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

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

( r1 = exec.induct )  
( t1 = exec )  
( to1 = exec )

$\wedge$

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

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

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

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

$\wedge$

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

new types ->

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

new constants ->

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

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

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

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

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

to1

r1

→

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

( r1 = exec.induct )

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

( t1 = exec )

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

( to1 = exec )

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

$\wedge$

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

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

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

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

$\wedge$

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

new types ->

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

new constants ->

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

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

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

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

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

to1

r1

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

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

(  $t1 = \text{exec}$  )

(  $to1 = \text{exec}$  )

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

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

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

$\wedge$

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

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

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

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

$\wedge$

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

new types ->

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

new constants ->

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

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

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

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

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

to1

r1

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

( r1 = exec.induct )

( t1 = exec )

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

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

$\wedge$

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

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

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

$\text{is\_nth\_argument\_of } (t2, n, t1)$

$\wedge$

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

new types ->

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

new constants ->

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

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

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

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

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

to1

r1

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

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

(  $t1 = \text{exec}$  )

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

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

$\wedge$

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

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

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

$\text{is\_nth\_argument\_of} (t2, n, t1)$

$\wedge$

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

new types ->

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

new constants ->

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

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

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

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

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

to1

r1

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

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

(  $t1 = \text{exec}$  )

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

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

$\wedge$

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

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

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

is\_nth\_argument\_of (t2, n, to1)

$\wedge$

t2 is\_nth\_induction\_term n

new types ->

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

new constants ->

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

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

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

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

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

b2

to1

r1

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

( r1 = exec.induct )

( t1 = exec )

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

( to1 = exec )

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

r1 is\_rule\_of to1    True! r1 (= exec.induct) is a lemma about to1 (= exec).

$\wedge$

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

( t2 = is1, s, and stk )

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

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

is\_nth\_argument\_of (to2, n, to1)

$\wedge$

t2 is\_nth\_induction\_term n

new types ->

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

new constants ->

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

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

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

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

$\rightarrow$

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

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

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

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

$\wedge$

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

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

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

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

$\wedge$

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

new types ->

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

new constants ->

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

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

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

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

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



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

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

(  $t1 = \text{exec}$  )

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

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



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

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

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

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



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

new types ->

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

new constants ->

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

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

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

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

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



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

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

(  $t1 = \text{exec}$  )

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

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



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

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

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

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

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



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

new types ->

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

new constants ->

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

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

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

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

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

**apply auto done**



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

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

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

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



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

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

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

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

$\text{is\_nth\_argument\_of } (t02, n, t01) \text{ when } t2 \text{ is is1 (n -> 1) ?}$



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

new types ->

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

new constants ->

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

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

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

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

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

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

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

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

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



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

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

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

$\text{is\_nth\_argument\_of } (to2, n, to1)$  when  $t2$  is  $\text{is1}$  ( $n \rightarrow 1$ )



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

new types ->

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

new constants ->

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

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

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

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

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

**apply auto done**



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

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

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

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



first  
second (r1 = exec.induct)

(t1 = exec)

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

(t2 = is1, s, and stk)

(to2 = is1, s, and stk)

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

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

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

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

when t2 is is1 (n -> 1)



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

when t2 is s (n -> 2)?

new types ->

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

new constants ->

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

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

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

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

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

**apply auto done**

r1

→

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

first  
second (r1 = exec.induct)

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

(t1 = exec)

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

r1 is\_rule\_of t01 True! r1 (= exec.induct) is a lemma about t01 (= exec).

^ 

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

(t2 = is1, s, and stk)

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

(t02 = is1, s, and stk)

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

is\_nth\_argument\_of (t02, n, t01)

when t2 is is1 (n → 1) 

^

t2 is\_nth\_induction\_term n

when t2 is s (n → 2) 

new types →

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

new constants  $\rightarrow$

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

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

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

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

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

→

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

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

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

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

8

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

( Fig. 31 is in the book and not in the box )

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

( to2 = is1, s, and stk )

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

is\_nth\_argument\_of (*to2*, *n*, *to1*)

△

*t2* is\_nth\_induction\_term *n*

when  $t_2$  is  $i \neq 1$  ( $n \rightarrow 1$ )

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

when t2 is std (n → 3) ?

new types →

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

new constants  $\rightarrow$

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

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

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

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

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

→

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

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

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

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

8

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

(to == first, and so on)

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

( to2 = is1, s, and stk )

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

is\_nth\_argument\_of (to2, n, to1)

when  $t_2$  is  $i \neq 1$  ( $n \rightarrow 1$ )

1

*t2* is\_nth\_induction\_term *n*

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

new types ->

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

new constants ->

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

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

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

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

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

**apply auto done**



→

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

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

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

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

^

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

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

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

$\text{is\_nth\_argument\_of} (t02, n, t01)$

^

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

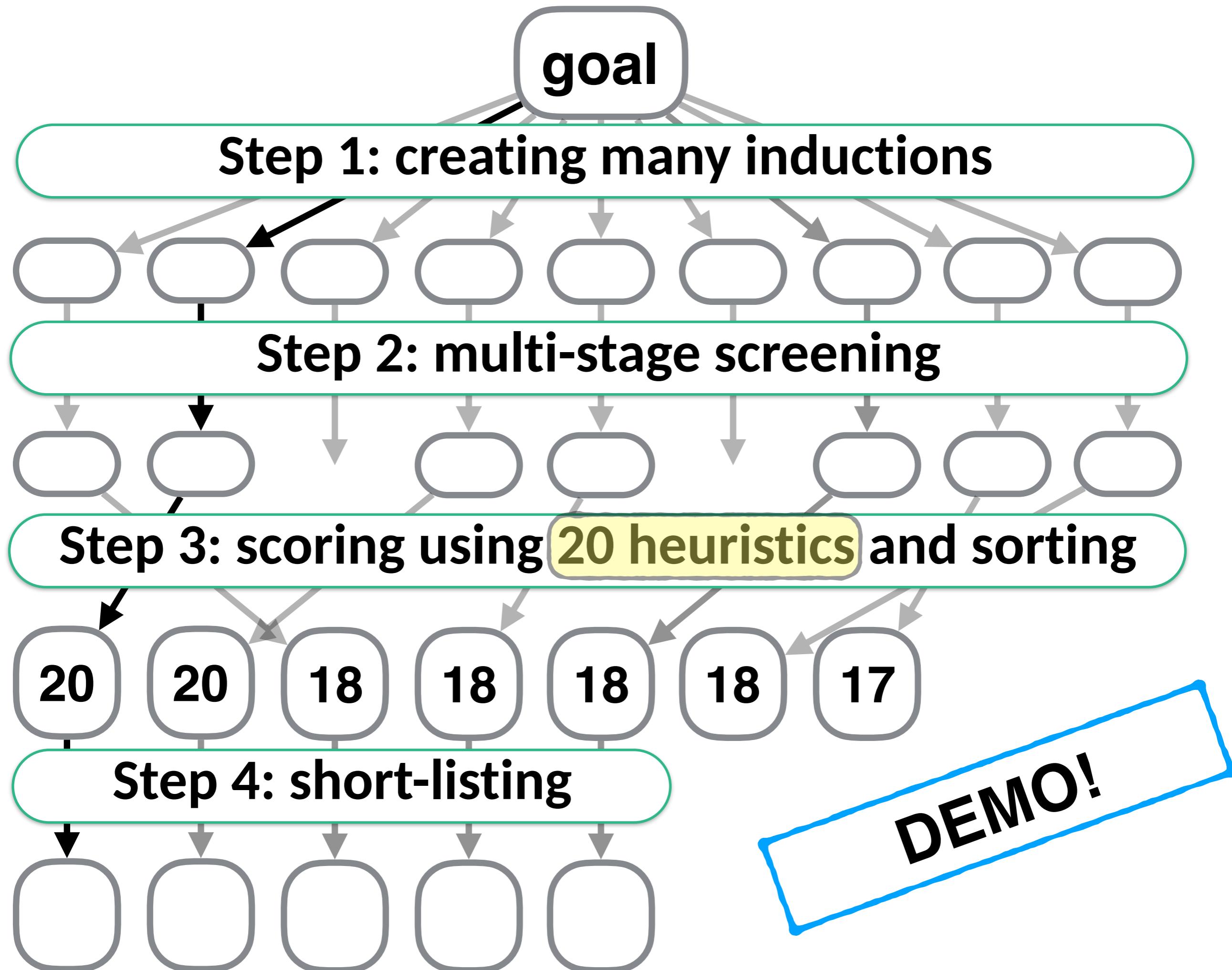
( $t2 = \text{is1, s, and stk}$ )

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

when  $t2$  is  $\text{is1}$  ( $n \rightarrow 1$ )

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

when  $t2$  is  $\text{stk}$  ( $n \rightarrow 3$ )



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

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

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

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

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

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

The screenshot shows the Isabelle/Isar interface. The top part displays a theory file named `Induction_Demo.thy` with the following content:

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

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

smart\_induct started producing combinations of induction arguments.  
smart\_induct produced 40 combinations of arguments for the induct method.  
... out of which 32 of them return some results.  
... out of which only 28 of them passes the second screening stage.  
LiFtEr assertions are evaluating the first 28 of them.  
Try these 10 most promising inductions!

1st candidate is apply (induct xs arbitrary: ys)  
(\* The score is 20 out of 20. \*)  
2nd candidate is apply (induct xs)  
(\* The score is 20 out of 20. \*)  
3th candidate is apply (induct xs ys rule: Induction\_Demo.itrev.induct)  
(\* The score is 18 out of 20. \*)

```
File Browser Documentation Sidekick State Theories  
Induction_Demo.thy (~/Workplace/PSL/Smart_Induct/Example/)  
14 fun itrev :: "'a list ⇒ 'a list ⇒ 'a list" where  
15 "itrev [] ys = ys" |  
16 "itrev (x#xs) ys = itrev xs (x#ys)"  
17  
18 lemma "itrev xs ys = rev xs @ ys" smart_induct  
19 apply(induction xs arbitrary: ys)  
20 apply(auto)
```

smart\_induct started producing combinations of induction arguments.  
smart\_induct produced 40 combinations of arguments for the induct method.  
... out of which 32 of them return some results.  
... out of which only 28 of them passes the second screening stage.  
LiFtEr assertions are evaluating the first 28 of them.  
Try these 10 most promising inductions!  
1st candidate is apply (induct xs arbitrary: ys)  
(\* The score is 20 out of 20. \*)  
2nd candidate is apply (induct xs)  
(\* The score is 20 out of 20. \*)  
3th candidate is apply (induct xs ys rule: Induction\_Demo.itrev.induct)  
(\* The score is 18 out of 20. \*)

The screenshot shows the Isabelle/Isar interface. The top part displays a theory file named `Induction_Demo.thy` with the following content:

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

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

smart\_induct started producing combinations of induction arguments.  
smart\_induct produced 40 combinations of arguments for the induct method.  
... out of which 32 of them return some results.  
... out of which only 28 of them passes the second screening stage.  
LiFtEr assertions are evaluating the first 28 of them.  
Try these 10 most promising inductions!

1st candidate is apply (induct xs arbitrary: ys)  
(\* The score is 20 out of 20. \*)  
2nd candidate is apply (induct xs)  
(\* The score is 20 out of 20. \*)  
3th candidate is apply (induct xs ys rule: Induction\_Demo.itrev.induct)  
(\* The score is 18 out of 20. \*)

The screenshot shows the Isabelle/Isar interface. The top part displays a theory file named `Induction_Demo.thy` with the following content:

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

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

The word `smart_induct` is highlighted in red, indicating an error or a placeholder. The line `apply(induction xs arbitrary: ys)` is highlighted in yellow.

smart\_induct started producing combinations of induction arguments.  
smart\_induct produced 40 combinations of arguments for the induct method.  
... out of which 32 of them return some results.  
... out of which only 28 of them passes the second screening stage.  
LiFtEr assertions are evaluating the first 28 of them.  
Try these 10 most promising inductions!

1st candidate is apply (induct xs arbitrary: ys)  
(\* The score is 20 out of 20. \*)  
2nd candidate is apply (induct xs)  
(\* The score is 20 out of 20. \*)  
3th candidate is apply (induct xs ys rule: Induction\_Demo.itrev.induct)  
(\* The score is 18 out of 20. \*)

# **smart\_induct... does it work?**

Table 1: Coincidence Rates of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

# **smart\_induct... does it work?**

Table 1: **Coincidence Rates** of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

# **smart\_induct... does it work?**

Table 1: **Coincidence Rates** of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

# **smart\_induct... does it work?**

Table 1: **Coincidence Rates** of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

Table 2: Coincidence Rates of `smart_induct` Based Only on Induction Terms.

theory	total	top_1	top_3	top_5	top_10
Nearest_Neighbors	16	5 (31%)	12 (75%)	15 (94%)	15 (94%)

# smart\_induct... does it work?

Table 1: Coincidence Rates of smart\_induct.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

Table 2: Coincidence Rates of smart\_induct Based Only on Induction Terms.

theory	total	top_1	top_3	top_5	top_10
Nearest_Neighbors	16	5 (31%)	12 (75%)	15 (94%)	15 (94%)

# smart\_induct... does it work?

Table 1: Coincidence Rates of smart\_induct.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

Table 2: Coincidence Rates of smart\_induct Based Only on Induction Terms.

theory	total	top_1	top_3	top_5	top_10
Nearest_Neighbors	16	5 (31%)	12 (75%)	15 (94%)	15 (94%)

Overall, the results are good!

But finding out what variables to generalise remains as a challenge!

# smart\_induct... does it work?

Table 1: Coincidence Rates of smart\_induct.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

Table 2: Coincidence Rates of smart\_induct Based Only on Induction Terms.

theory	total	top_1	top_3	top_5	top_10
Nearest_Neighbors	16	5 (31%)	12 (75%)	15 (94%)	15 (94%)

Overall, the results are good!

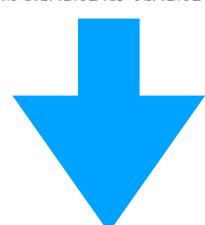
But finding out what variables to generalise remains as a challenge!

WIP!

# Today I talked about...

2013 ~ 2017

Intern &  
Engineer



PhD in  
AI for theorem proving

2017 ~ 2018



2018 ~  
2020/21



with Prof. Gerwin Klein



with Prof. Cezary Kaliszyk



with Dr. Josef Urban

Cogent

PSL

PaMpeR

LiFtEr

smart\_induct

# Today I talked about...

2013 ~ 2017

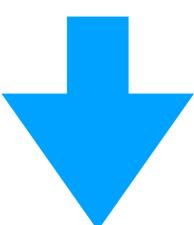
Intern &  
Engineer



**Cogent** ASPLOS2016

ITP2016

ICFP2016



PhD in  
AI for theorem proving



2017 ~ 2018



**PSL** CADE2017

CICM2018

**PaMpeR** ASE2018

2018 ~  
2020/21



with Dr. Josef Urban

**LiFtEr** APLAS2019

**smart\_indu** under review at  
IJCAR2020

# Today I talked about... *fin.*

2013 ~ 2017

Intern &  
Engineer

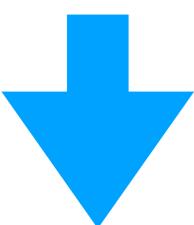


**Cogent**

ASPLOS2016

ITP2016

ICFP2016



PhD in  
AI for theorem proving



2017 ~ 2018



**PSL**

CADE2017

CICM2018

**PaMpeR**

ASE2018

2018 ~  
2020/21



with Dr. Josef Urban

**LiFtEr**

APLAS2019

**smart\_indu**

under review at  
IJCAR2020

**backup slides for Q&A**

$\leftarrow$  simple representation



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

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

*<- simple representation*



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

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

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

*<- relevant definitions*

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

*<- simple representation*

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

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

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

*<- relevant definitions*

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

$\leftarrow$  simple representation

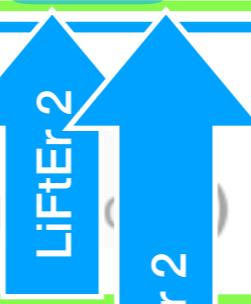
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs arbitrary: ys) auto
```



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

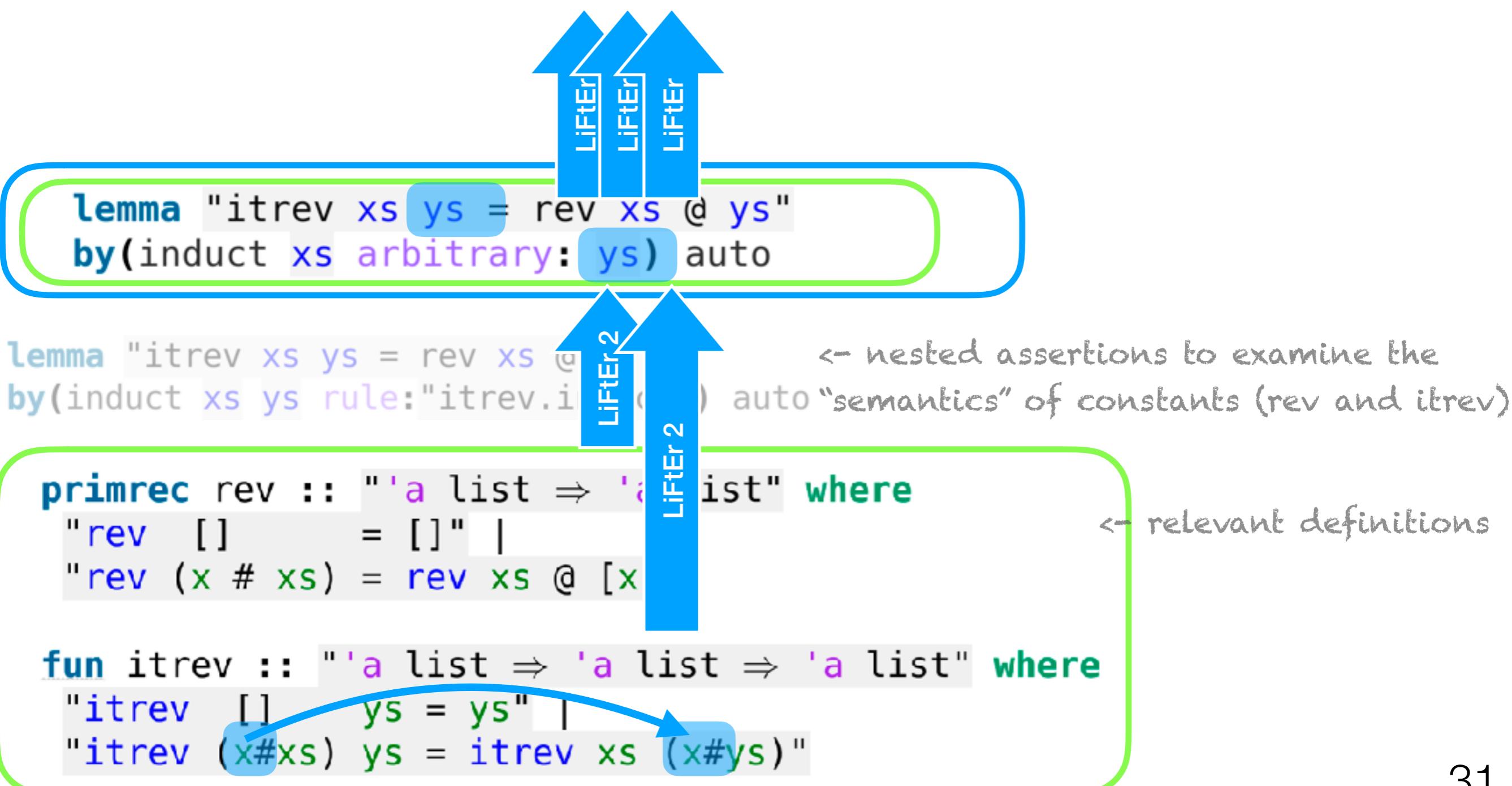


```
fun itrev :: "'a list ⇒ 'a list ⇒ 'a list" where  
"itrev [] ys = ys" |  
"itrev (x#xs) ys = itrev xs (x#ys)"
```

$\leftarrow$  relevant definitions

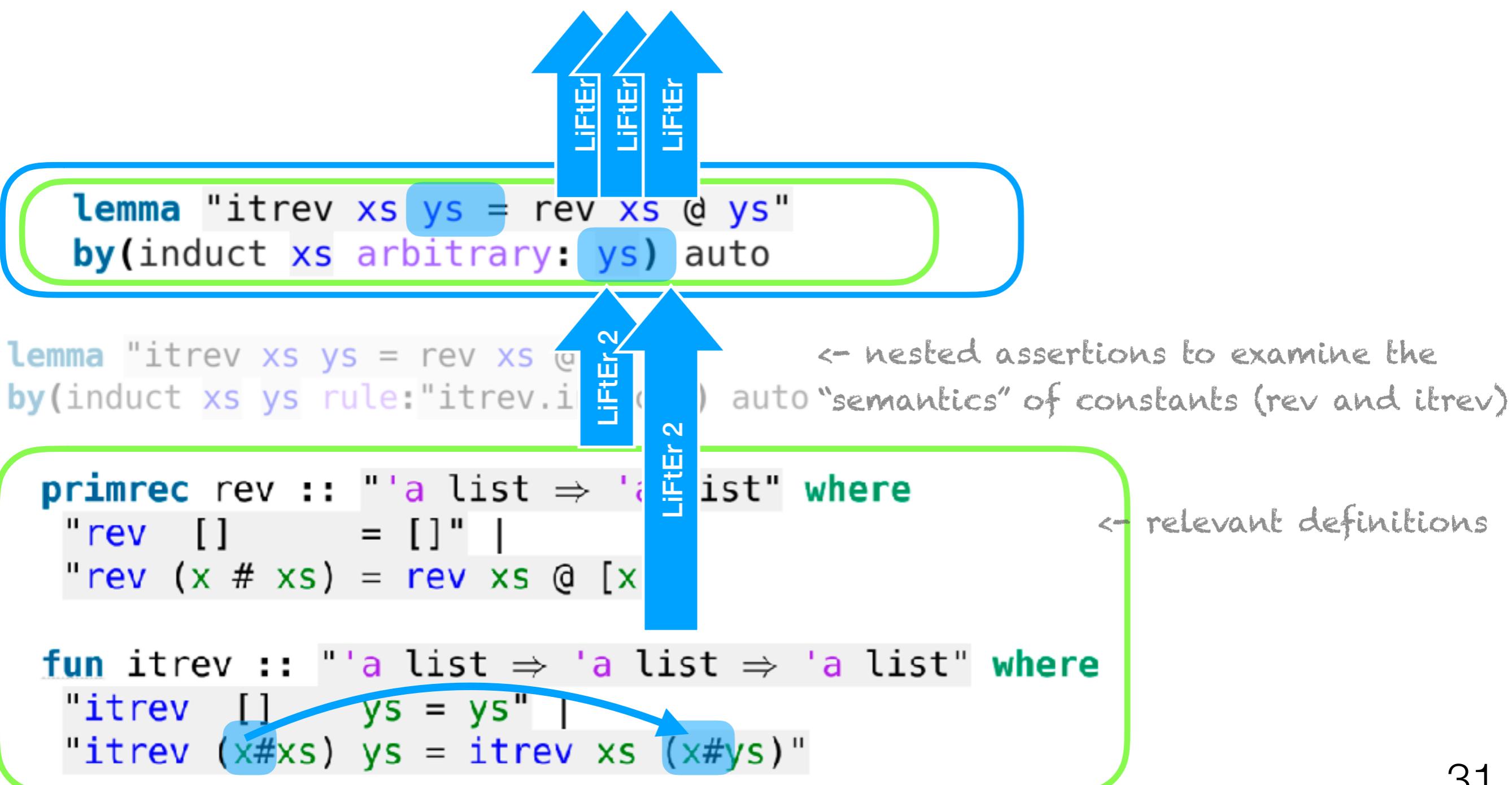
# LiFtEr: ( proof goal \* induction arguments ) -> bool

<- simple representation



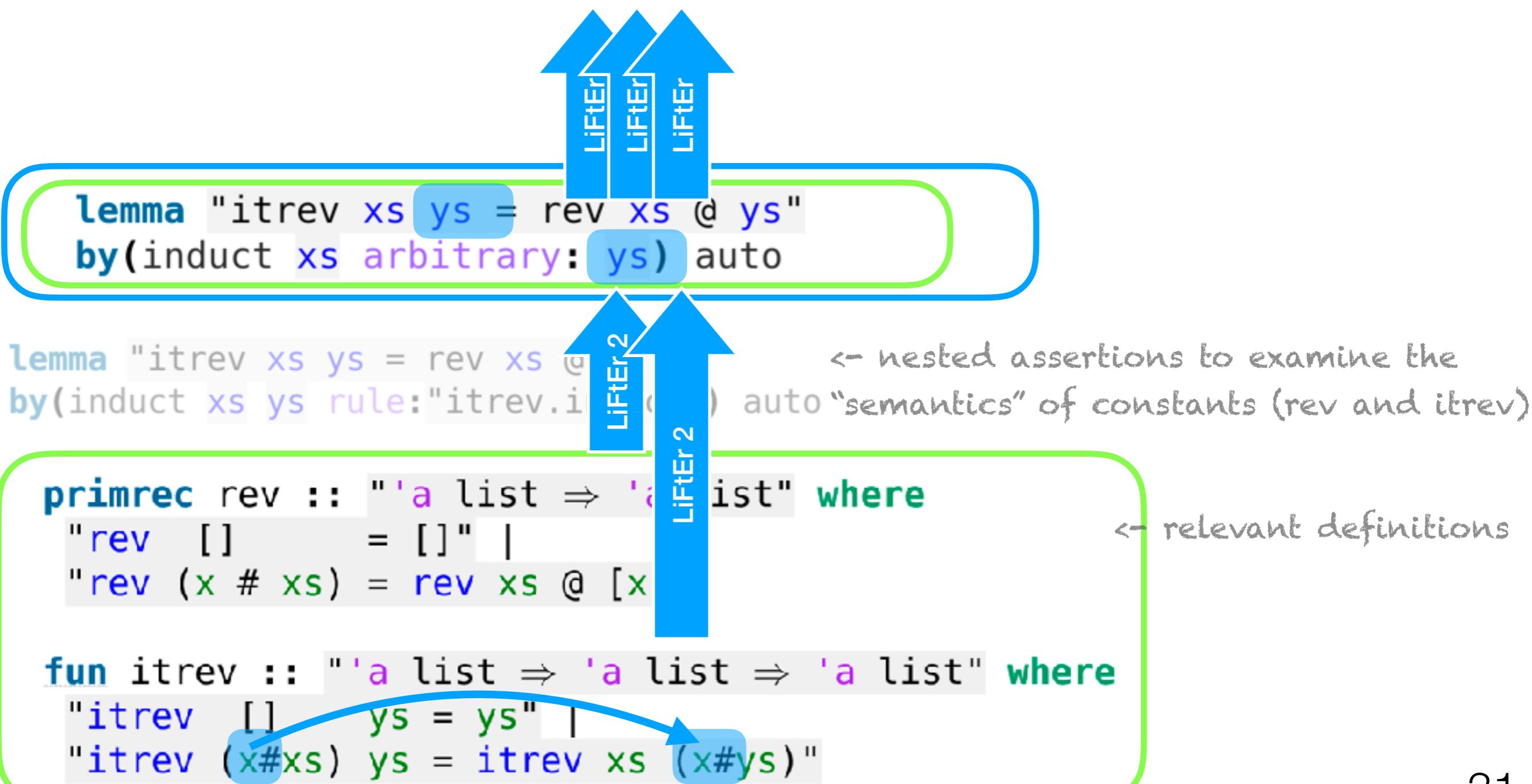
# LiFtEr<sub>2</sub> ( proof goal \* induction arguments ) -> bool \* relevant definitions

<- simple representation



# LiFtEr<sub>2</sub> ( proof goal \* induction arguments ) -> bool \* relevant definitions

coming soon



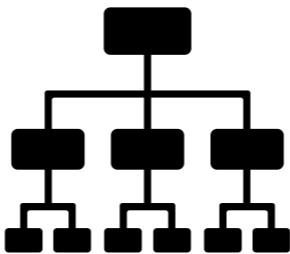


**LiFtEr**: ( proof goal \* induction arguments ) -> bool

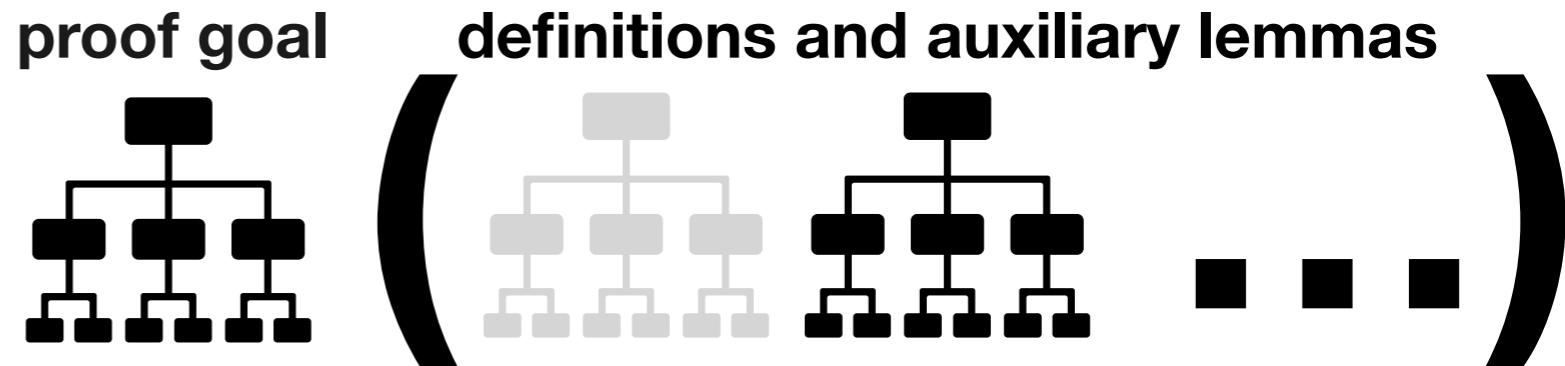
**LiFtEr<sub>2</sub>** ( proof goal \* induction arguments ) -> bool  
\* relevant definitions

**LiFtEr<sub>2</sub>** ( proof goal \* induction arguments ) -> bool  
\* relevant definitions

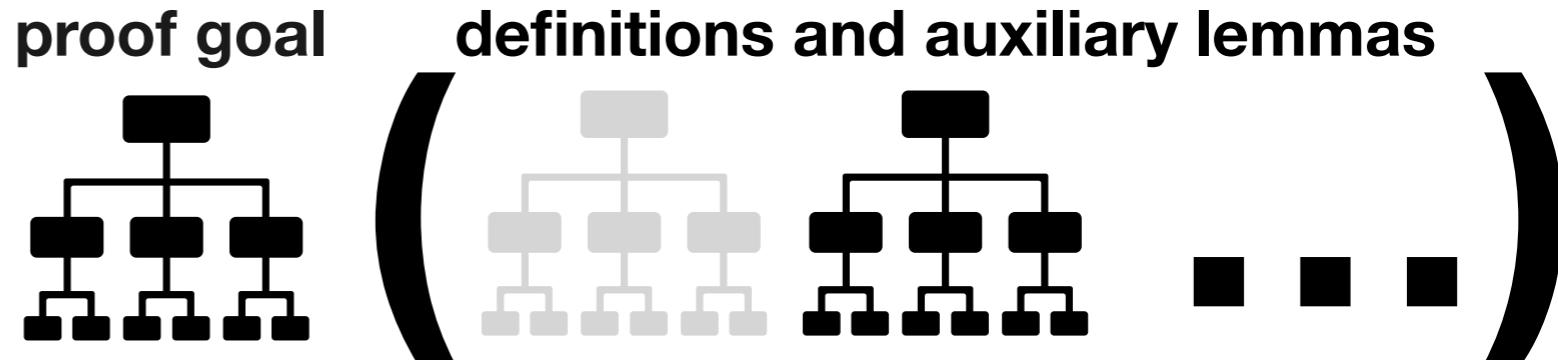
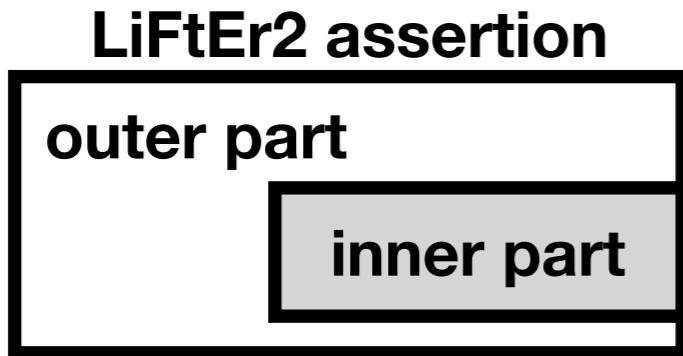
proof goal



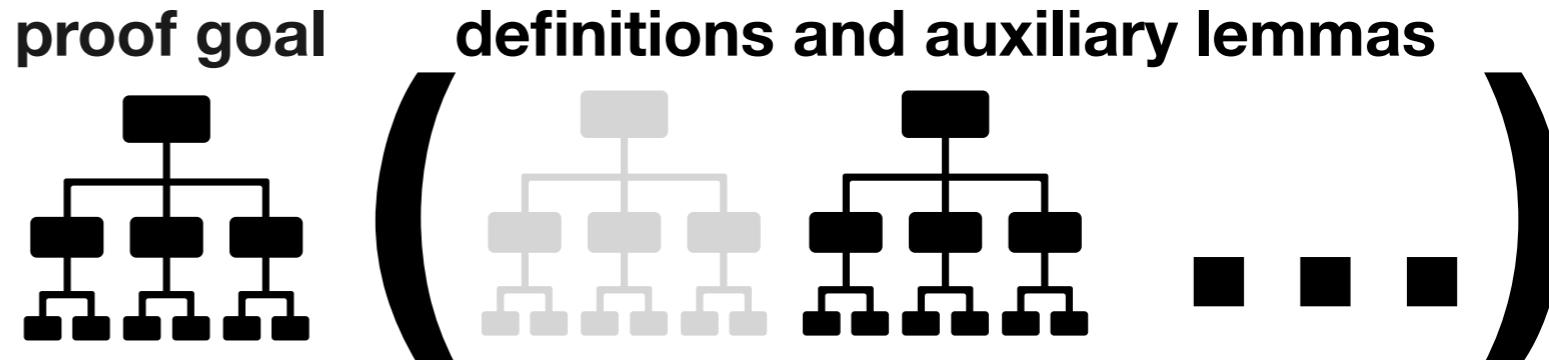
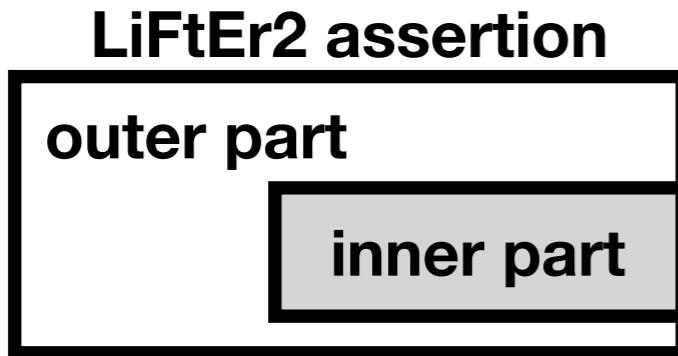
**LiFtEr<sub>2</sub>** ( proof goal \* induction arguments ) -> bool  
\* relevant definitions



**LiFtEr<sub>2</sub>** ( proof goal \* induction arguments ) -> bool  
\* relevant definitions

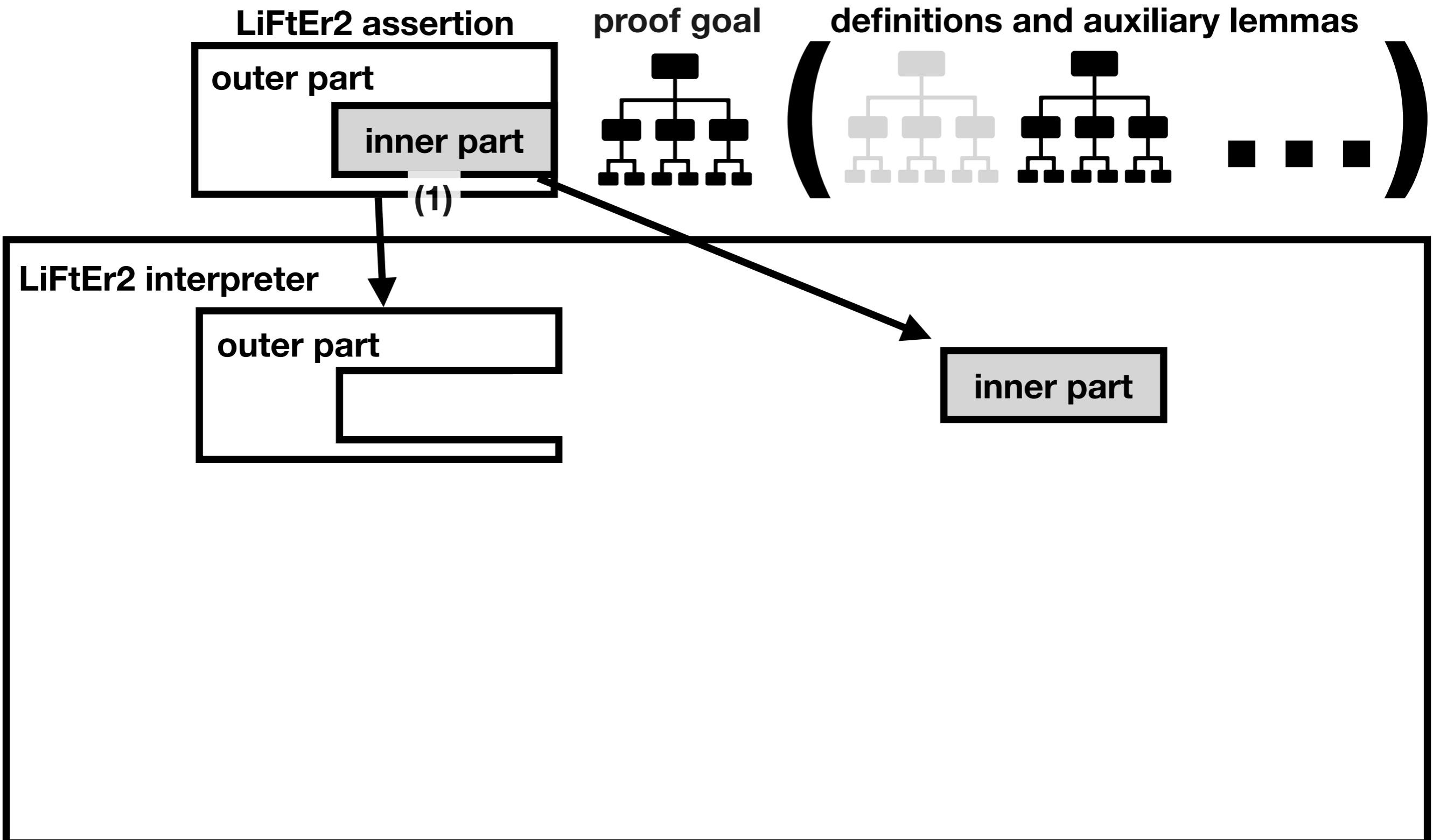


**LiFtEr<sub>2</sub>** ( proof goal \* induction arguments ) -> bool  
\* relevant definitions

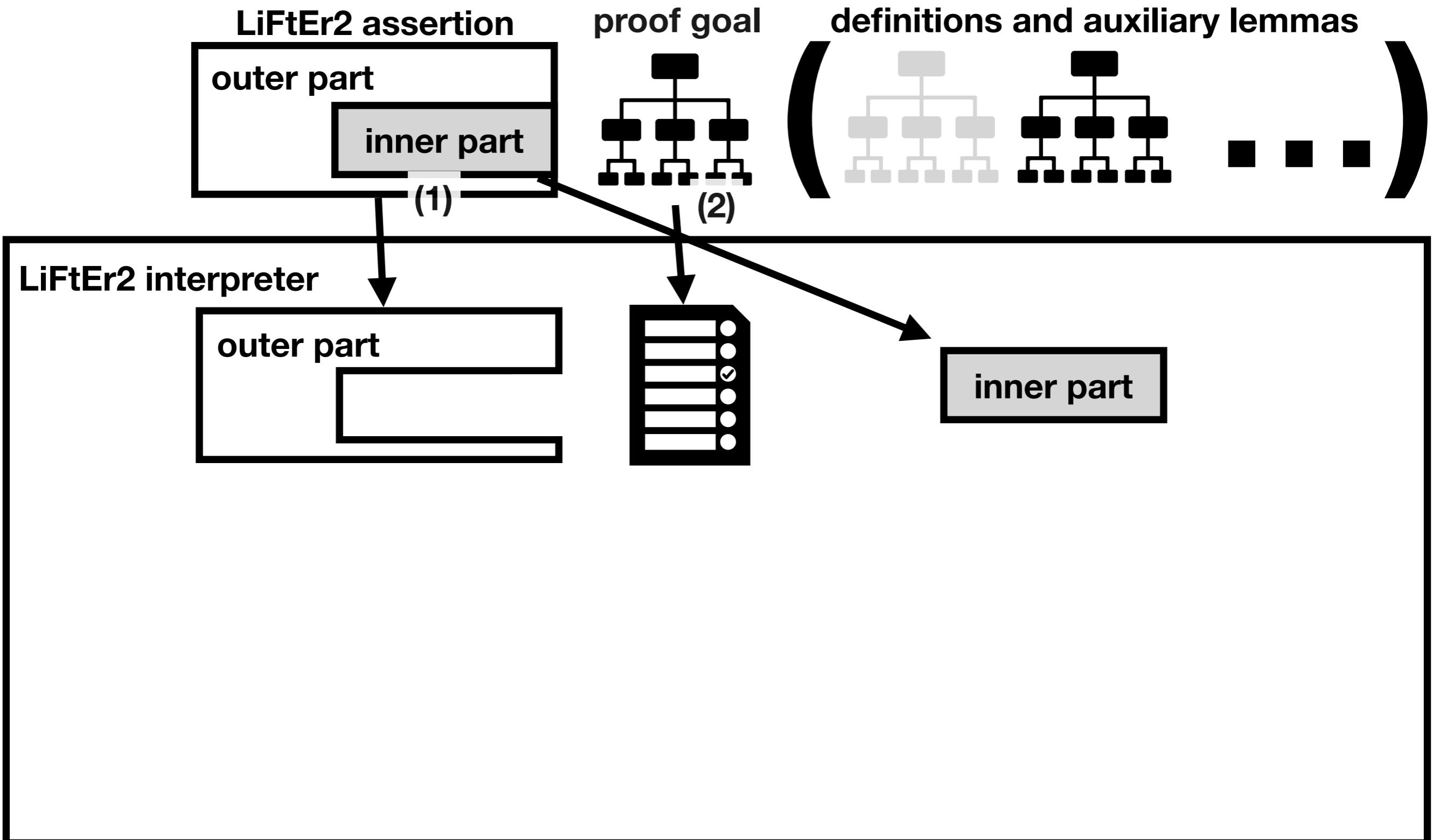


LiFtEr2 interpreter

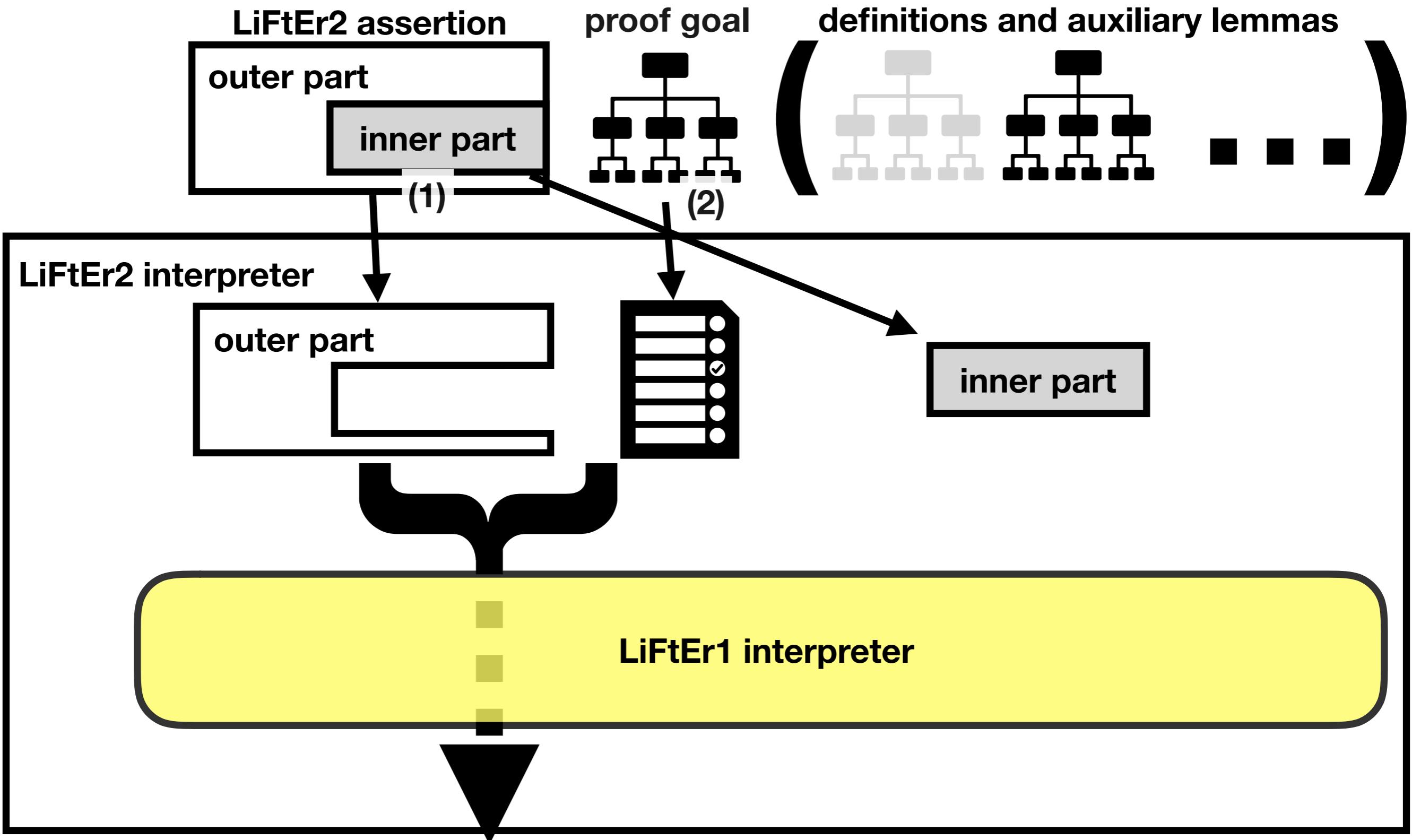
**LiFtEr<sub>2</sub>** ( proof goal \* induction arguments ) -> bool  
\* relevant definitions



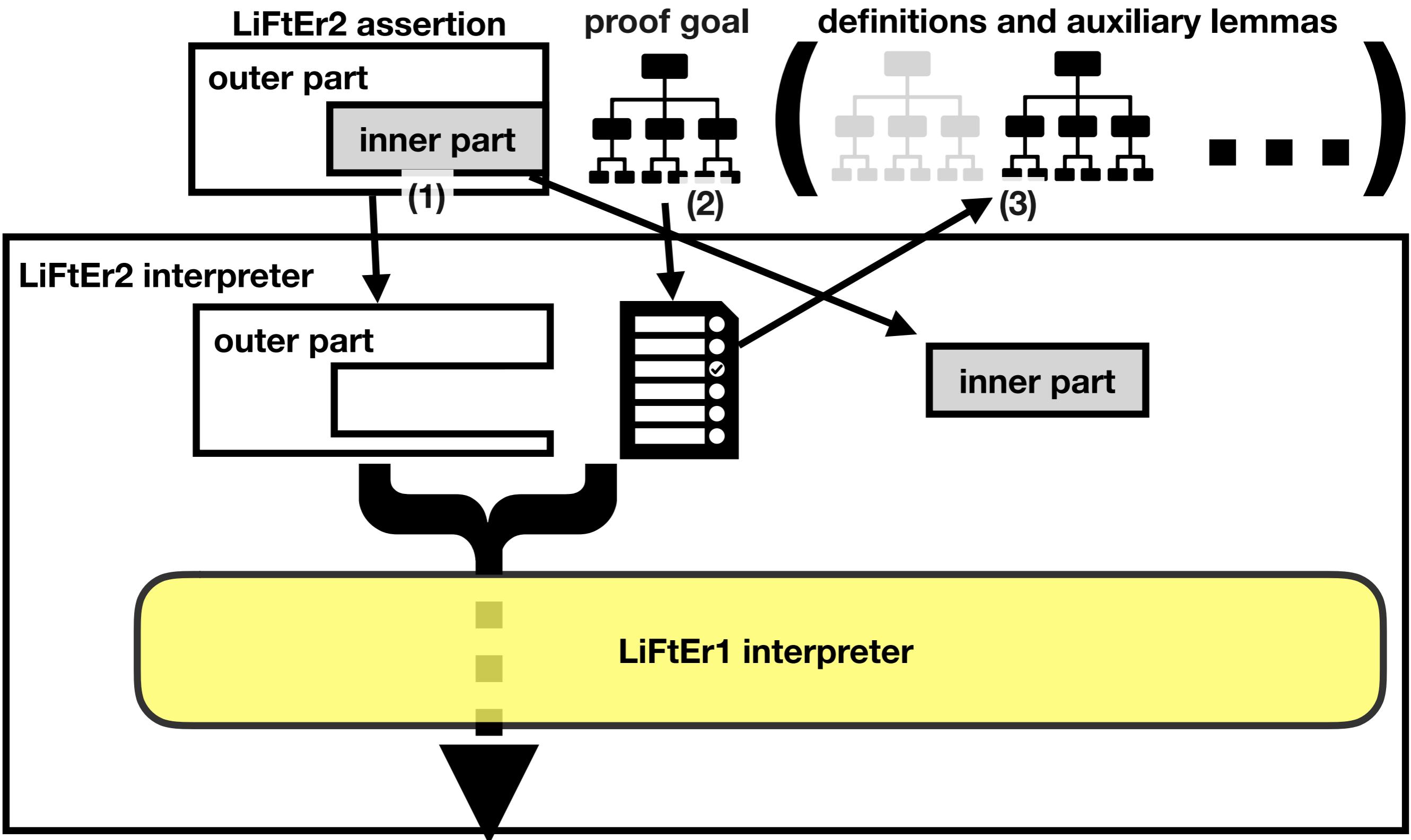
**LiFtEr<sub>2</sub>** ( proof goal \* induction arguments ) -> bool  
\* relevant definitions



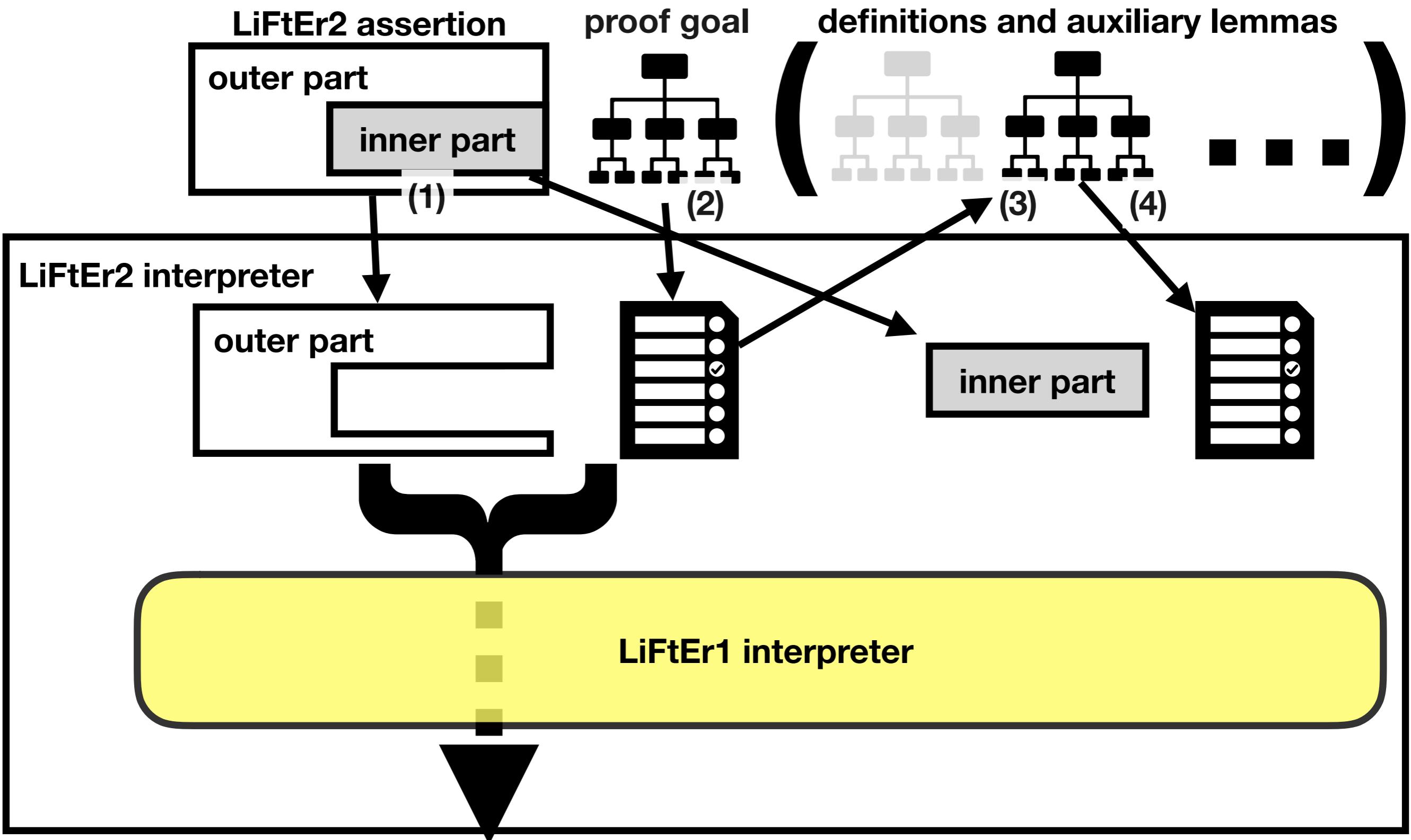
**LiFtEr<sub>2</sub>** ( proof goal \* induction arguments ) -> bool  
\* relevant definitions



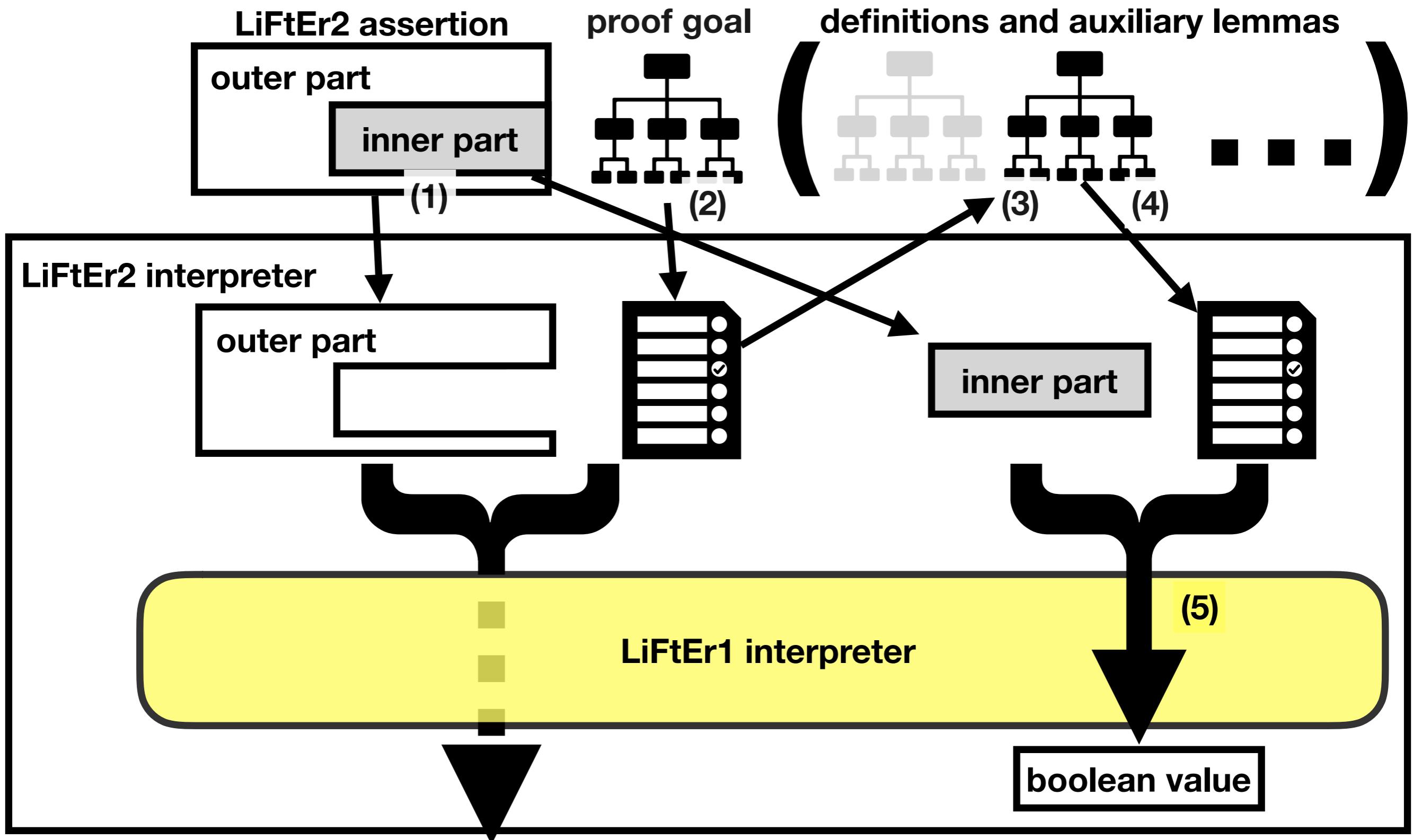
**LiFtEr<sub>2</sub>** ( proof goal \* induction arguments ) -> bool  
\* relevant definitions



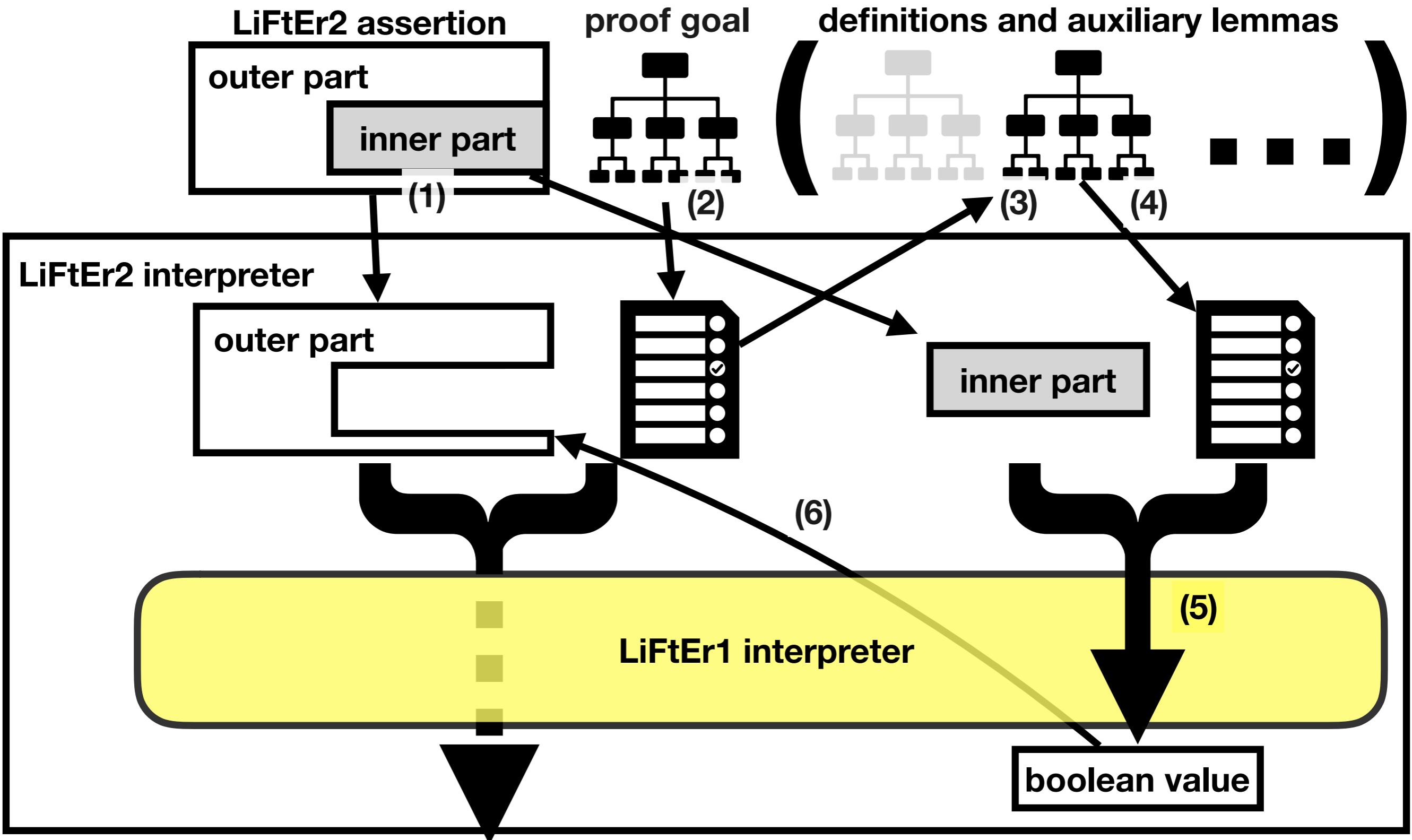
**LiFtEr<sub>2</sub>** ( proof goal \* induction arguments ) -> bool  
\* relevant definitions



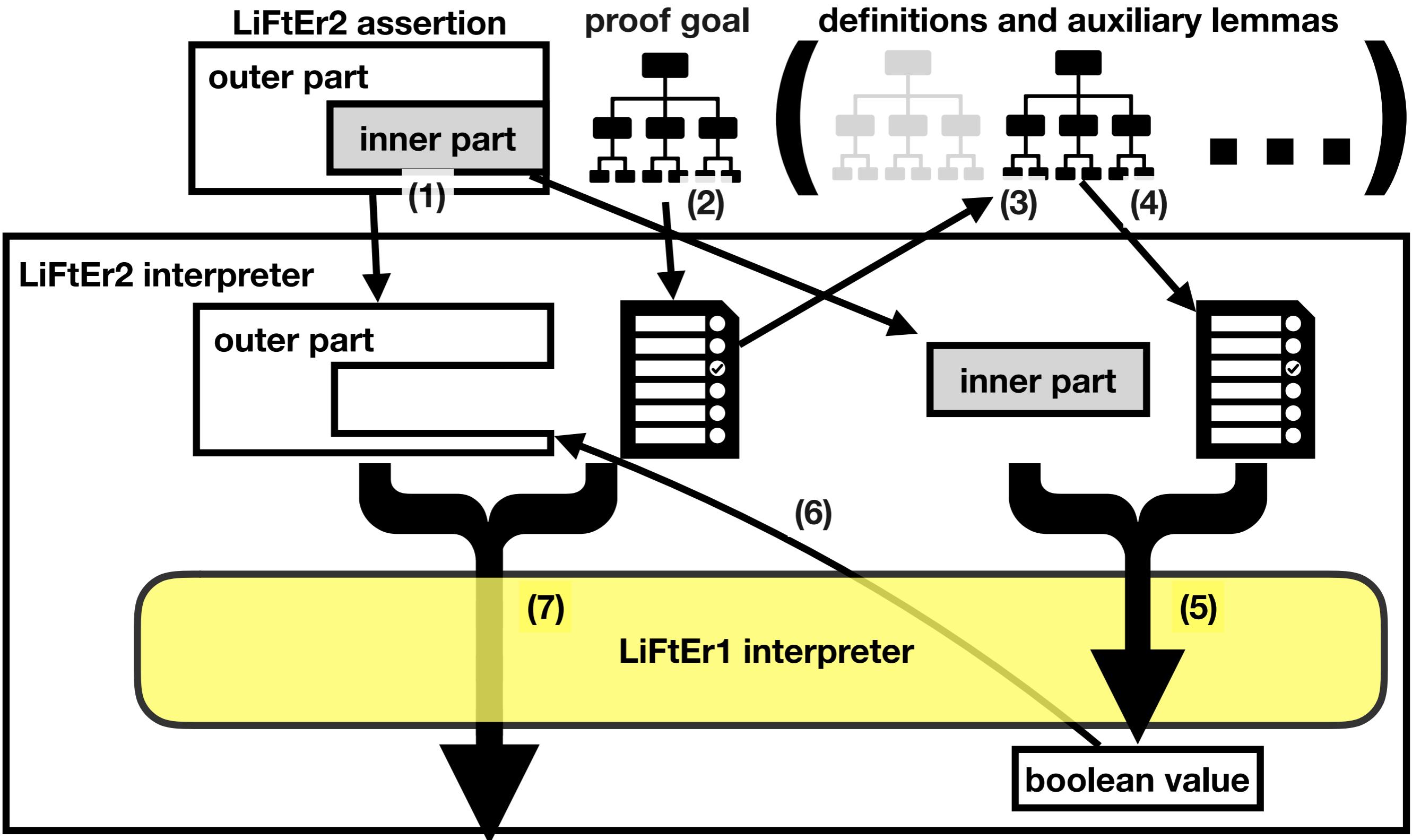
**LiFtEr<sub>2</sub>** ( proof goal \* induction arguments ) -> bool  
\* relevant definitions



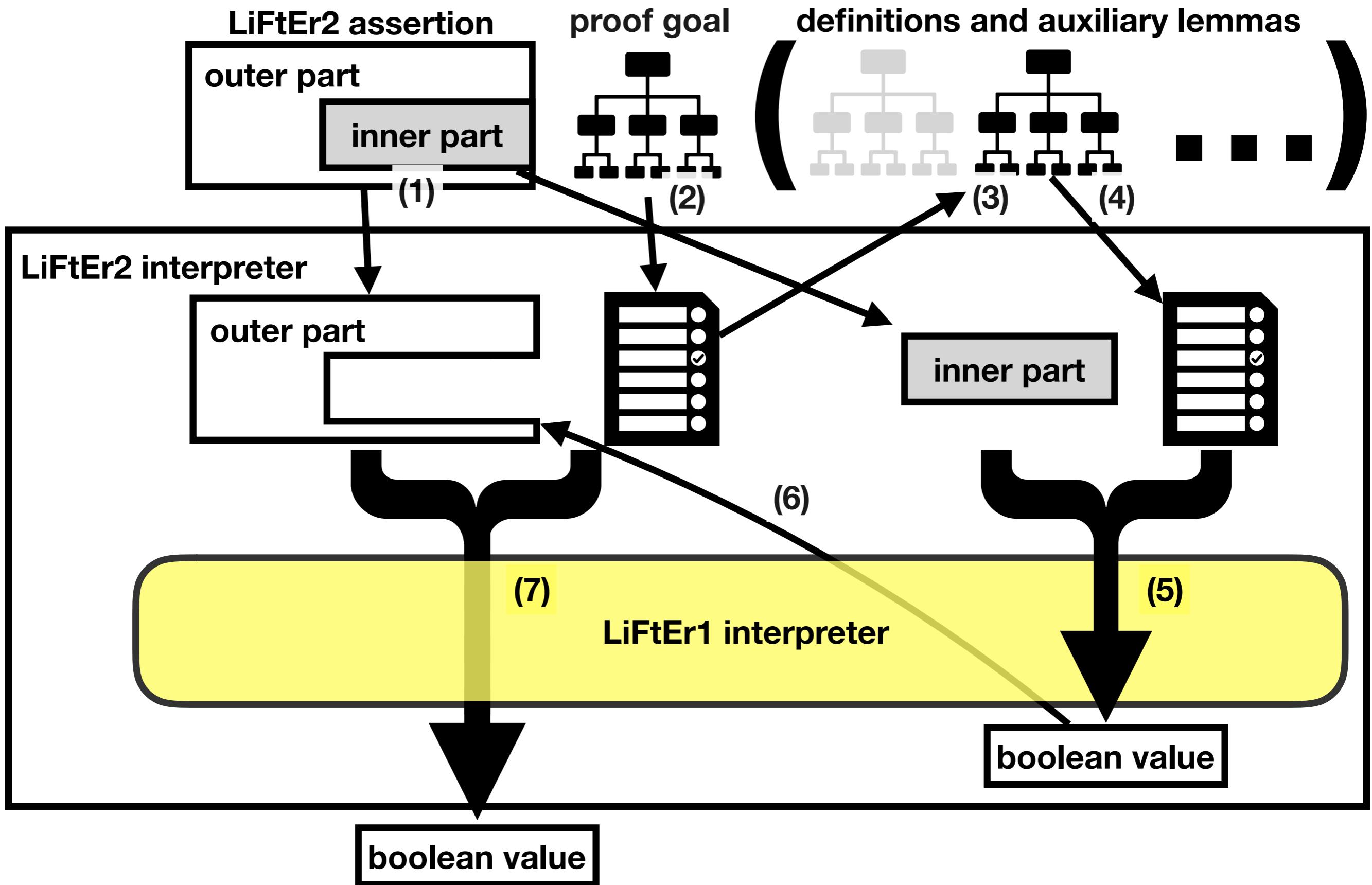
**LiFtEr<sub>2</sub>** ( proof goal \* induction arguments ) -> bool  
\* relevant definitions



**LiFtEr<sub>2</sub>** ( proof goal \* induction arguments ) -> bool  
\* relevant definitions

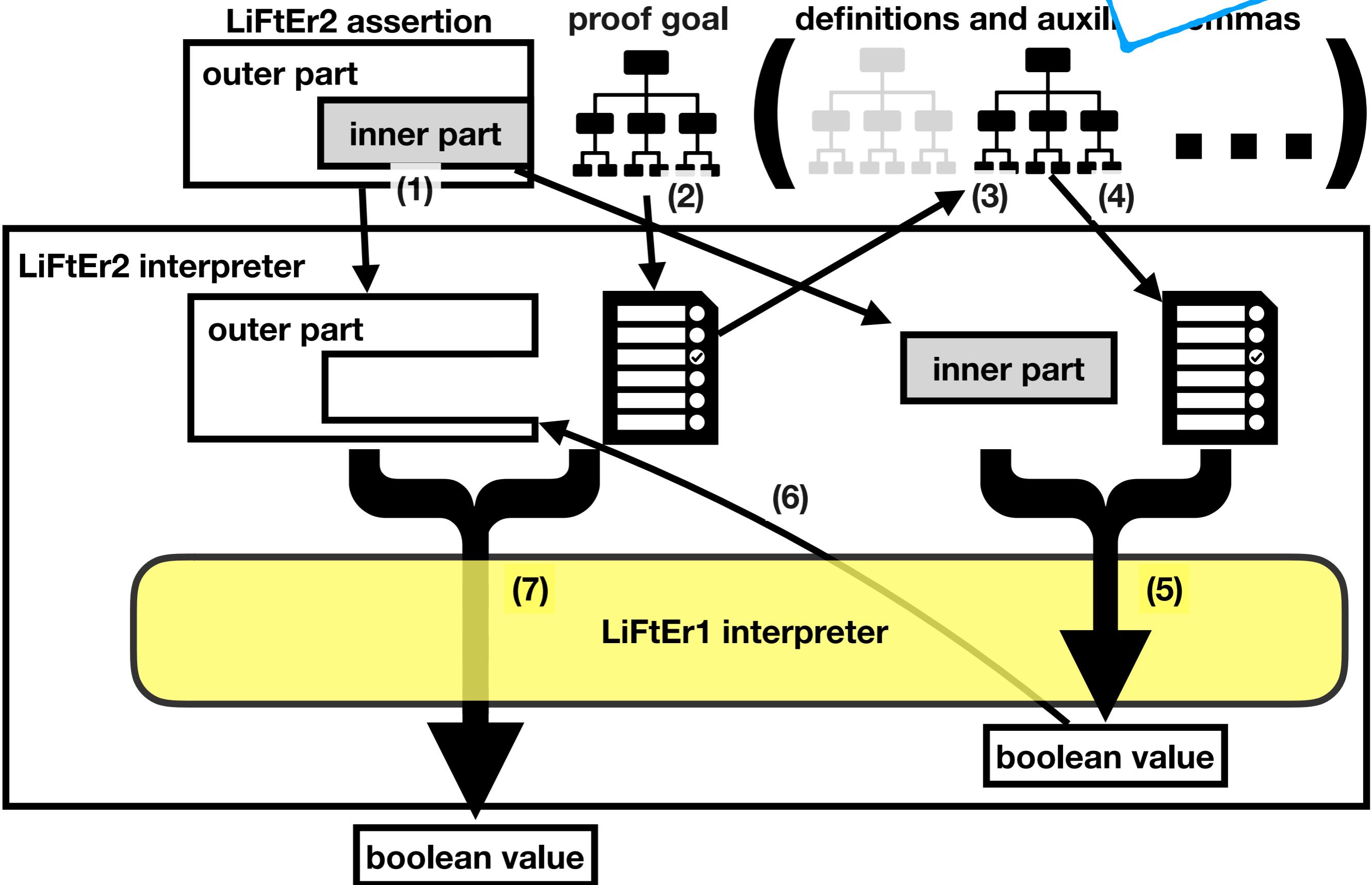


**LiFtEr<sub>2</sub>** ( proof goal \* induction arguments ) -> bool  
\* relevant definitions

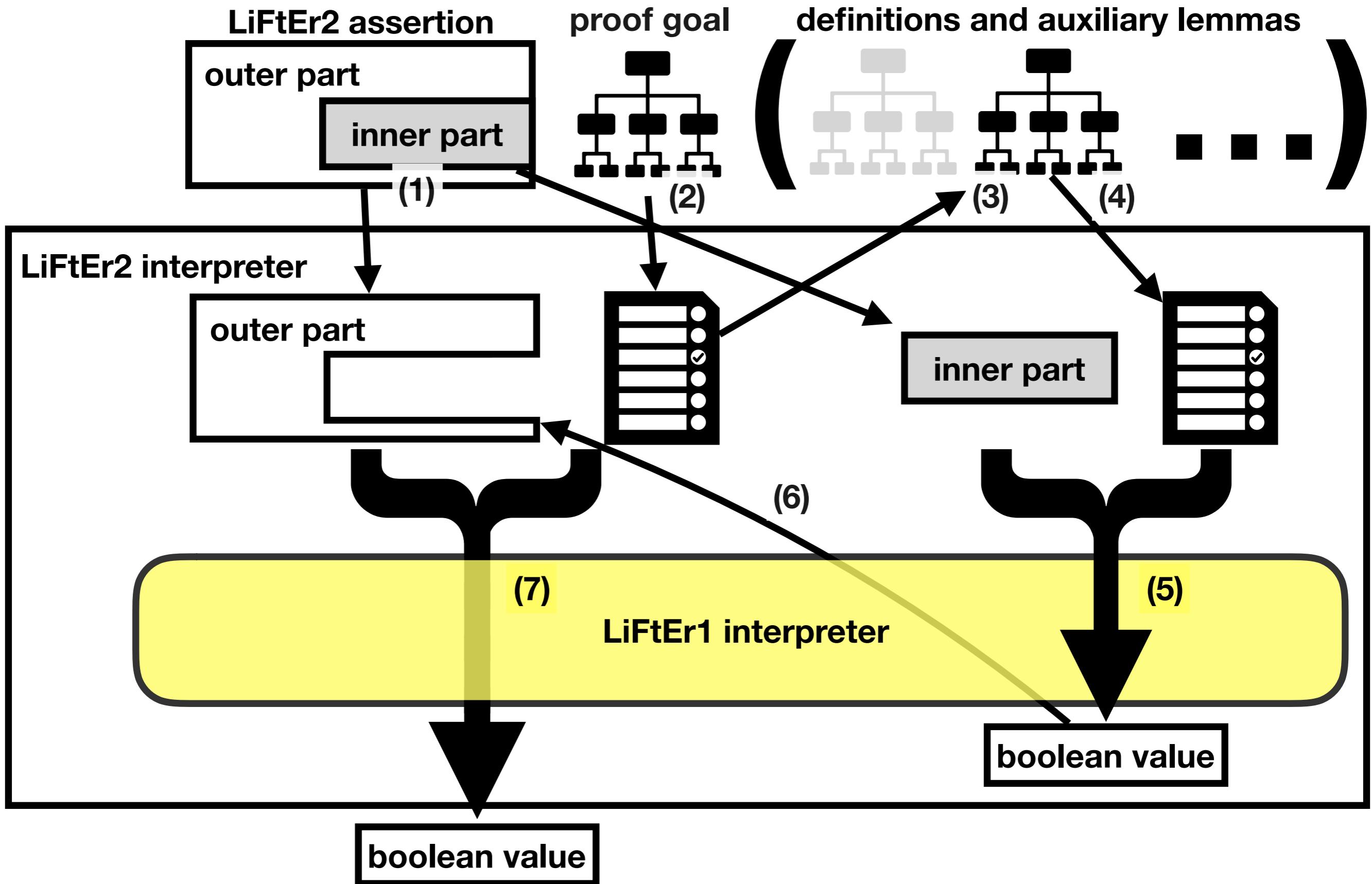


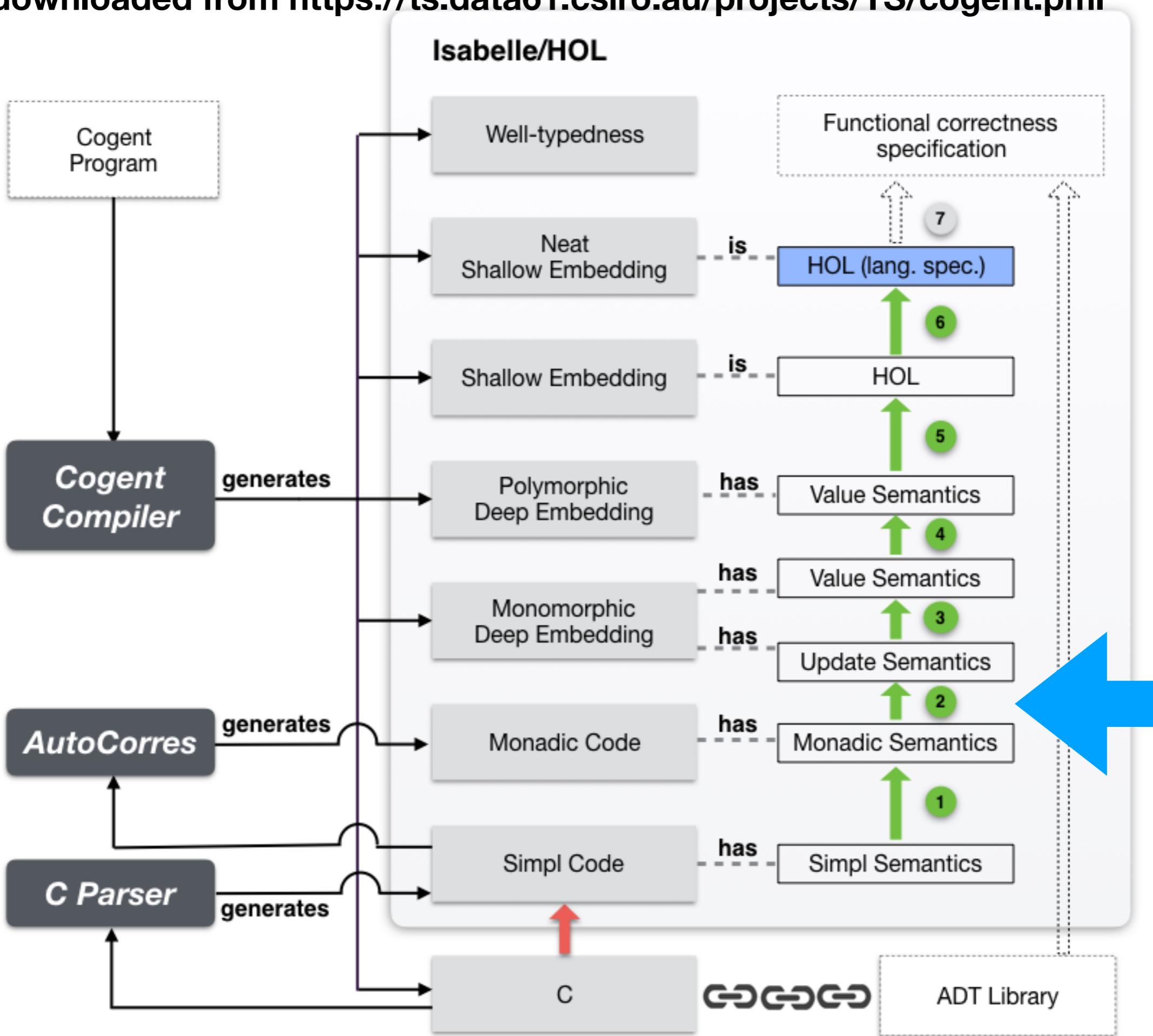
**LiFtEr<sub>2</sub>** ( proof goal \* induction arguments ) -> bool  
\* relevant definitions

**WIP!**

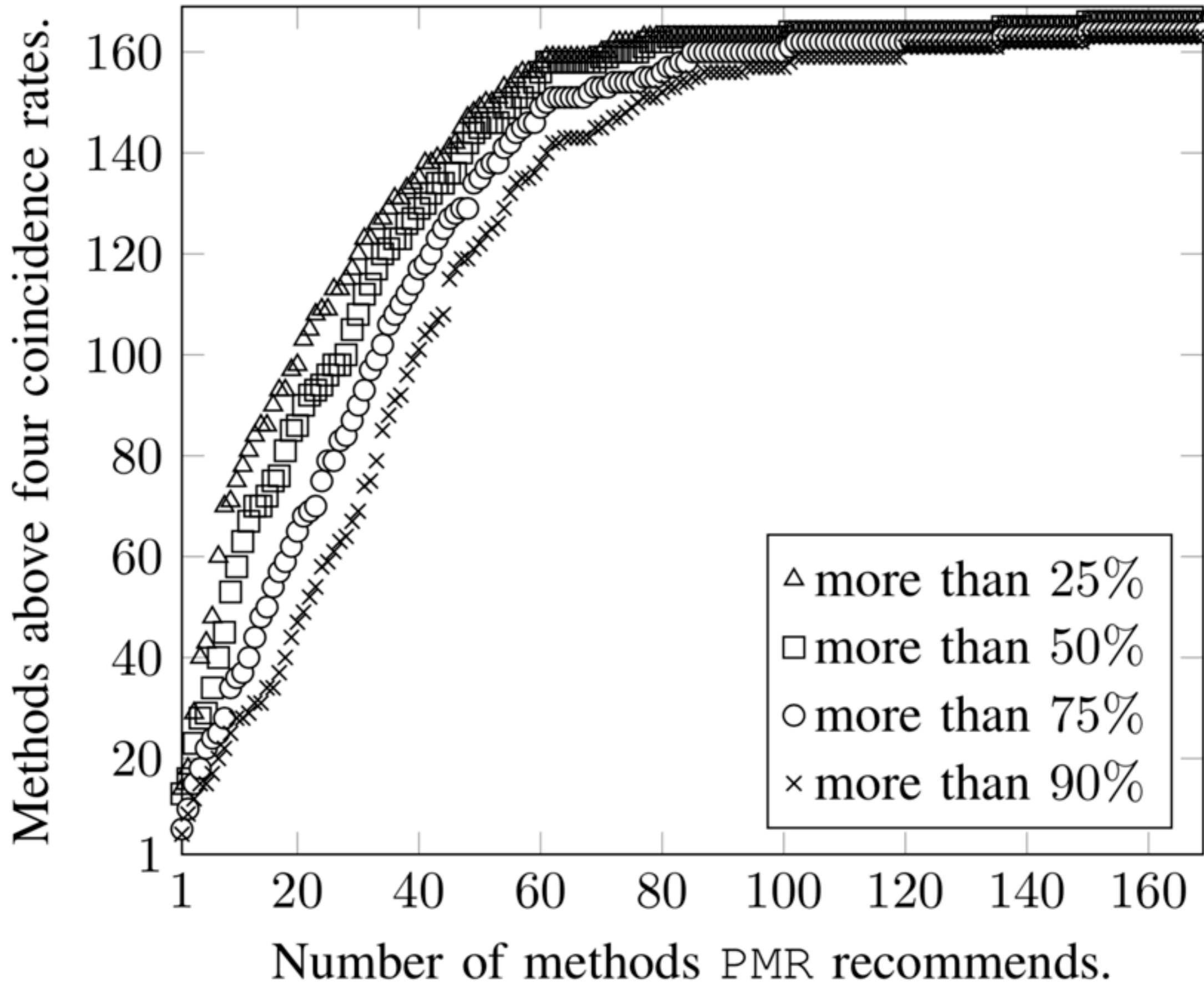


**LiFtEr<sub>2</sub>** ( proof goal \* induction arguments ) -> bool  
\* relevant definitions

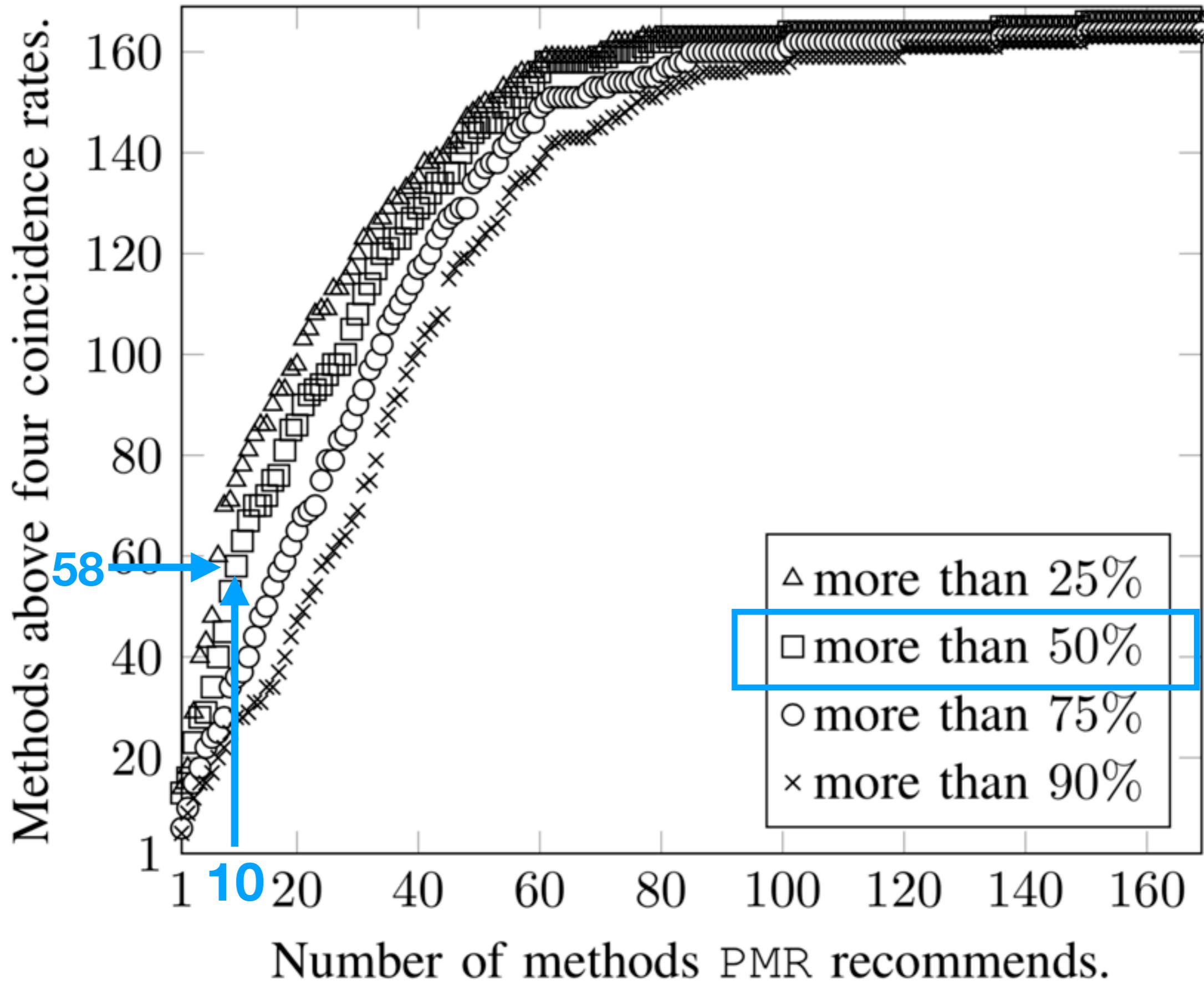




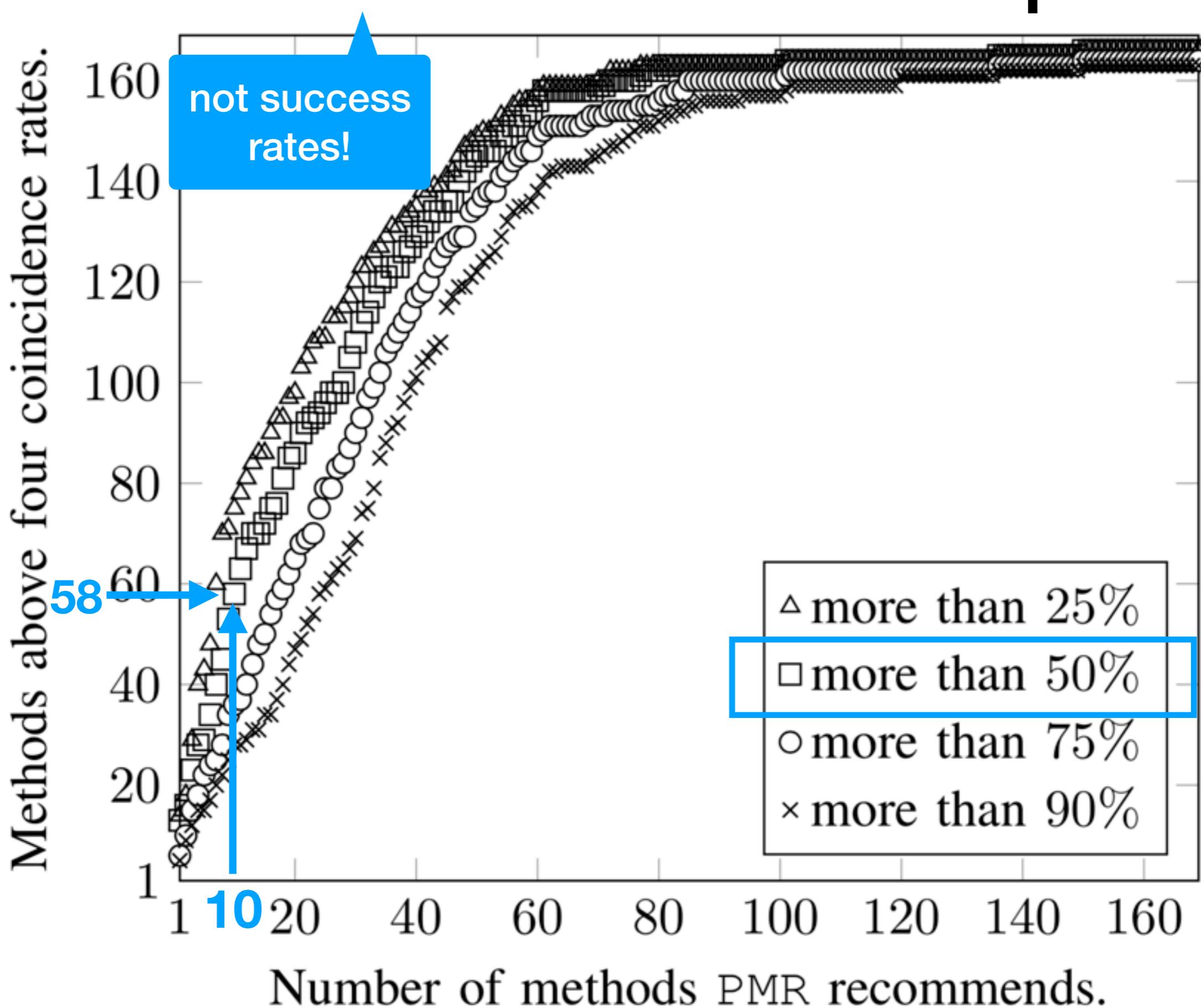
# Coincidence rates of PaMpeR



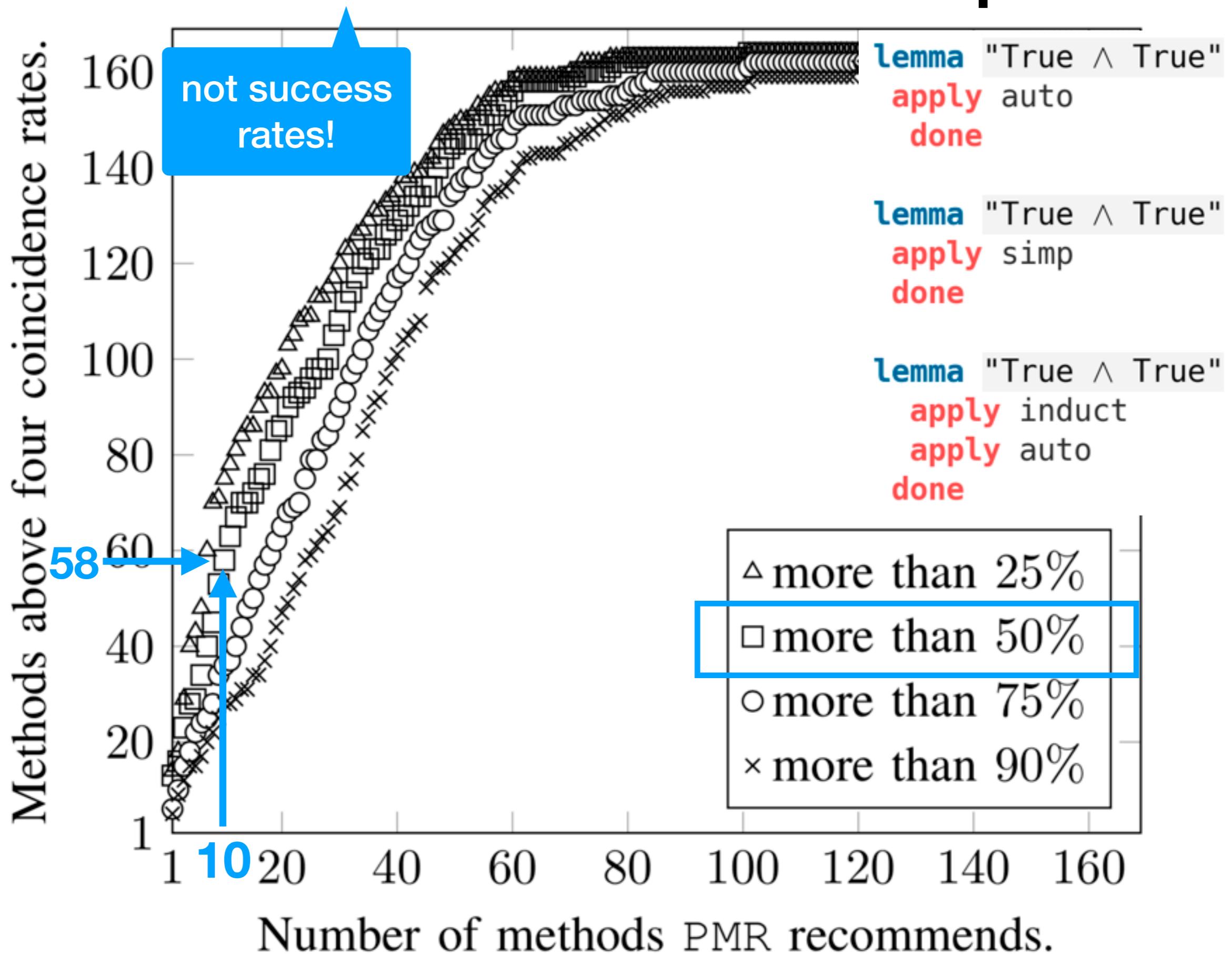
# Coincidence rates of PaMpeR



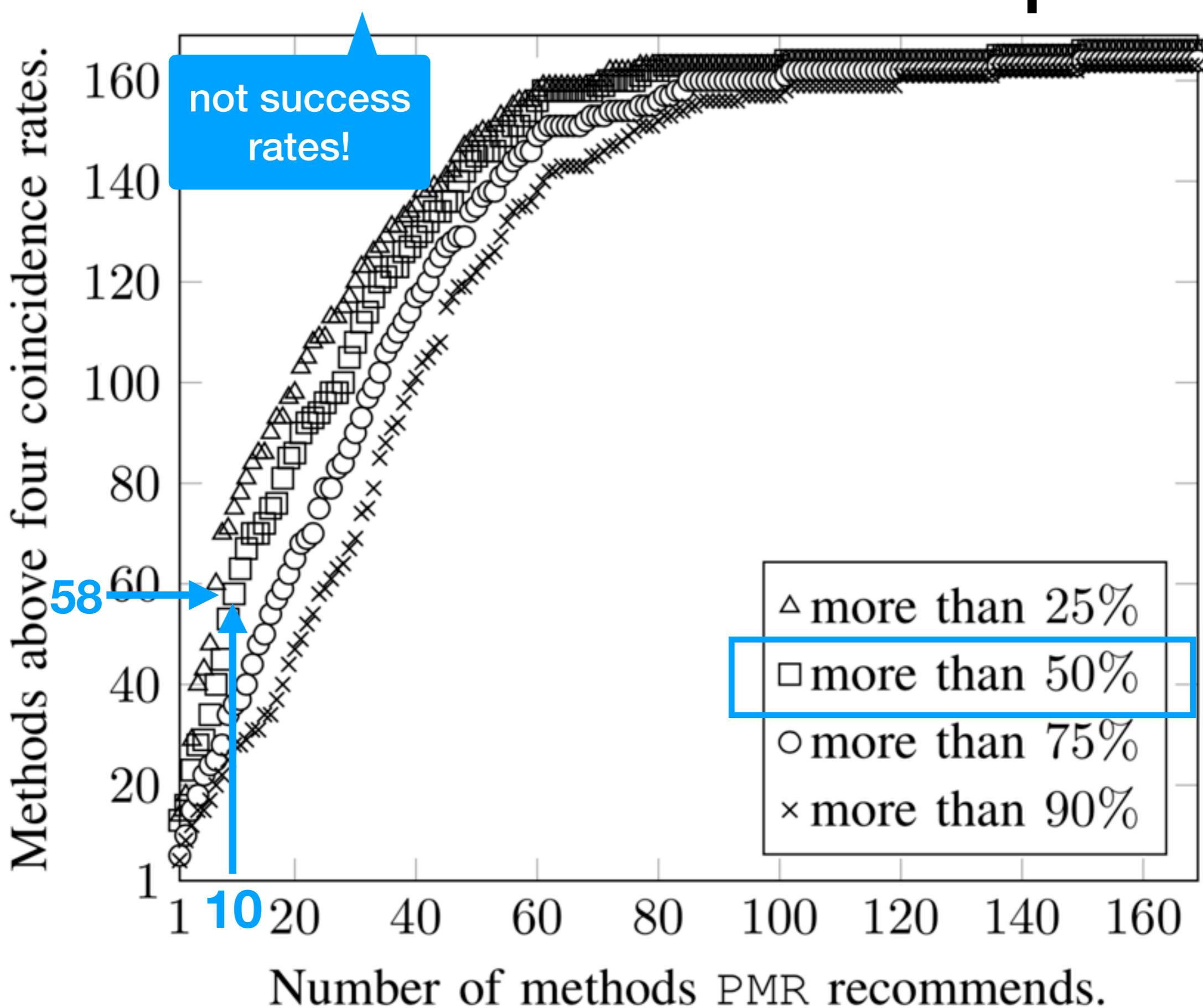
# Coincidence rates of PaMpeR



# Coincidence rates of PaMpeR



# Coincidence rates of PaMpeR



# Coincidence rates of PaMpeR

