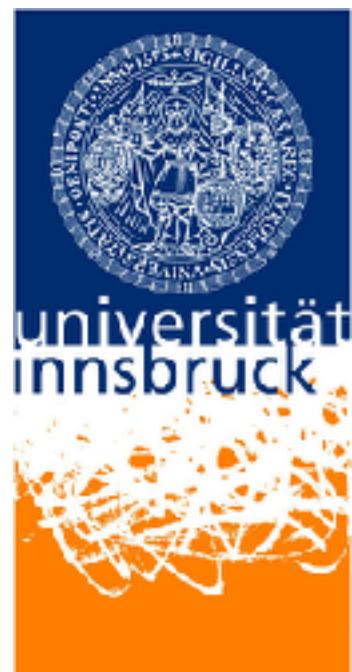
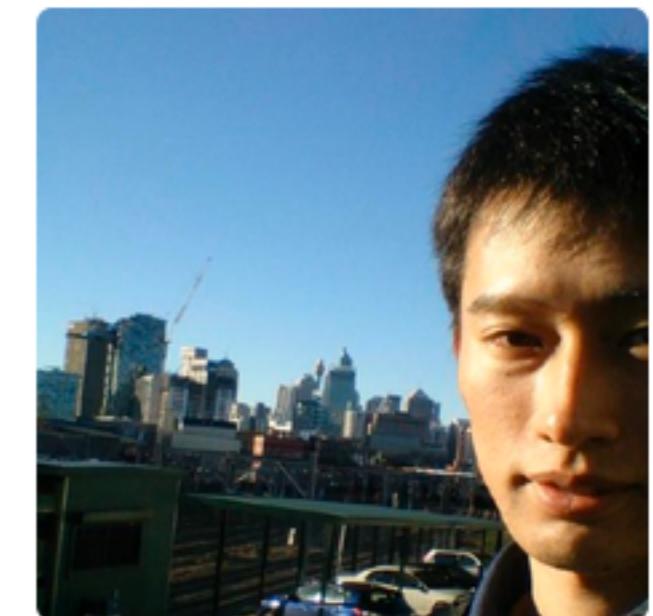


https://github.com/data61/PSL/blob/master/slide/2020_ETH.pdf

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_ETH.pdf

Automating proof by induction in Isabelle/HOL using DSLs



Yutaka Nagashima



Yutaka Ng

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

Background

2013 ~ 2017



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



with Prof. Gerwin Klein

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

Background

2013 ~ 2017



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

with Prof. Gerwin Klein



Cogent

Background

2013 ~ 2017

Intern &
Engineer



with Prof. Gerwin Klein



Cogent

Background

2013 ~ 2017

Intern &
Engineer



PhD in
AI for theorem proving



Cogent

Background

2013 ~ 2017

Intern &
Engineer



Cogent

with Prof. Gerwin Klein

PhD in
AI for theorem proving



2017 ~ 2018



with Prof. Cezary Kaliszyk

Background

2013 ~ 2017

Intern &
Engineer



Cogent

PhD in
AI for theorem proving



PSL

2017 ~ 2018

Background

2013 ~ 2017

Intern &
Engineer



Cogent

PhD in
AI for theorem proving



2017 ~ 2018



PSL

with Prof. Cezary Kaliszyk

PaMpeR

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

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

Background

2013 ~ 2017

Intern & Engineer



Cogent

PhD in AI for theorem proving



PSL

2017 ~ 2018



PaMpeR

**2018 ~
2020/21**

<http://ci-informatik.ulb.ac.at/users/cek/>
with Prof. Cezary Kaliszyk

Background

2013 ~ 2017

Intern &
Engineer



Cogent

PhD in
AI for theorem proving



PSL

2017 ~ 2018

with Prof. Gerwin Klein

with Prof. Cezary Kaliszyk

2018 ~
2020/21



LiFtEr

with Dr. Josef Urban

smart_induct

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

Background

2013 ~ 2017

Intern & Engineer



Cogent

PhD in AI for theorem proving



PSL

2017 ~ 2018



LiFtEr

**2018 ~
2020/21**

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

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

thub.com/data61/PSL/blob/master/slide/2020_ETH.pdf

The screenshot shows the Isabelle proof assistant interface. The top part displays a function definition in ML-like syntax:

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

The third line is highlighted with a yellow background. The bottom part shows the output of a command:

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

The status bar at the bottom indicates the current proof state and termination order.

consts

sep :: "'a :: type list ⇒ 'a :: type list"

Found termination order: "(λp. length (snd p)) <*lex*> {}"

DEMO1

thub.com/data61/PSL/blob/master/slide/2020_ETH.pdf

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

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

Below the definition, a value is being evaluated:

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

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

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

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

DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.pdf

The screenshot shows the Isabelle proof assistant interface. The main window displays ML code for a function `sep` and a lemma. The code is color-coded: blue for identifiers, green for types, and purple for type annotations. The lemma statement is also color-coded. The interface includes a toolbar at the top, a file browser on the left, and a sidebar on the right.

```
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

thub.com/data61/PSL/blob/master/slides/2020_ETH.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 for proofs.

```
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, indicating it is the current selection. The interface includes a toolbar at the top, a vertical scroll bar on the right, and a status bar at the bottom.

DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.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. A proof goal is currently active.

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

1 fun sep :: "'a ⇒ 'a list ⇒ 'a list" where
2   "sep a []      = []" |
3   "sep a [x]     = [x]" |
4   "sep a (x#y#zs) = x # a # sep a (y#zs)"
5
6 value "sep (1::int) [0,0,0]"
7
8 strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved]
9
10 lemma
11   "map f (sep x xs) = sep (f x) (map f xs)" □
12   find_proof DInd
13
14 proof (prove)
15 goal (1 subgoal):
16   1. map f (sep x xs) = sep (f x) (map f xs)
```

The interface includes a toolbar at the top, a vertical scroll bar on the right, and a status bar at the bottom. The status bar shows memory usage (17.44 MB / 361/1084) and a timestamp (1:05 AM).

DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.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 for it, sets a strategy, and states a lemma. The lemma is currently being proved.

```
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
```

Output Query Sledgehammer Symbols

18.18 (379/1084)

(isabelle.isabelle,UTF-8-Isabelle) nmr0 U.. 251/535MB 1:05 AM

DEMO1

thub.com/data61/PSL/blob/master/slide/2020_ETH.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 for induction. A lemma is stated, followed by a proof script involving `apply auto` and `done`. The proof part is highlighted with a yellow background.

```
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!
```

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!

Output Query Sledgehammer Symbols

20.11 (433/1147) (isabelle/isabelle,UTF-8-Isabelle) in mro U.. 255/535MB 1:05 AM

DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.pdf

The screenshot shows the Isabelle proof assistant interface. The main window displays a proof script. The script includes a function definition for 'sep', a value declaration for 'sep', a strategy for induction, a lemma statement, and a proof script for the lemma. The proof script uses 'induct' and 'auto' tactics. A yellow highlight covers the 'apply auto' line. A blue speech bubble in the bottom right corner contains the text 'What happened?'. The interface has a toolbar at the top, a file browser on the left, and a sidebar on the right.

```
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)"

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!
```

What happened?

DEMO1

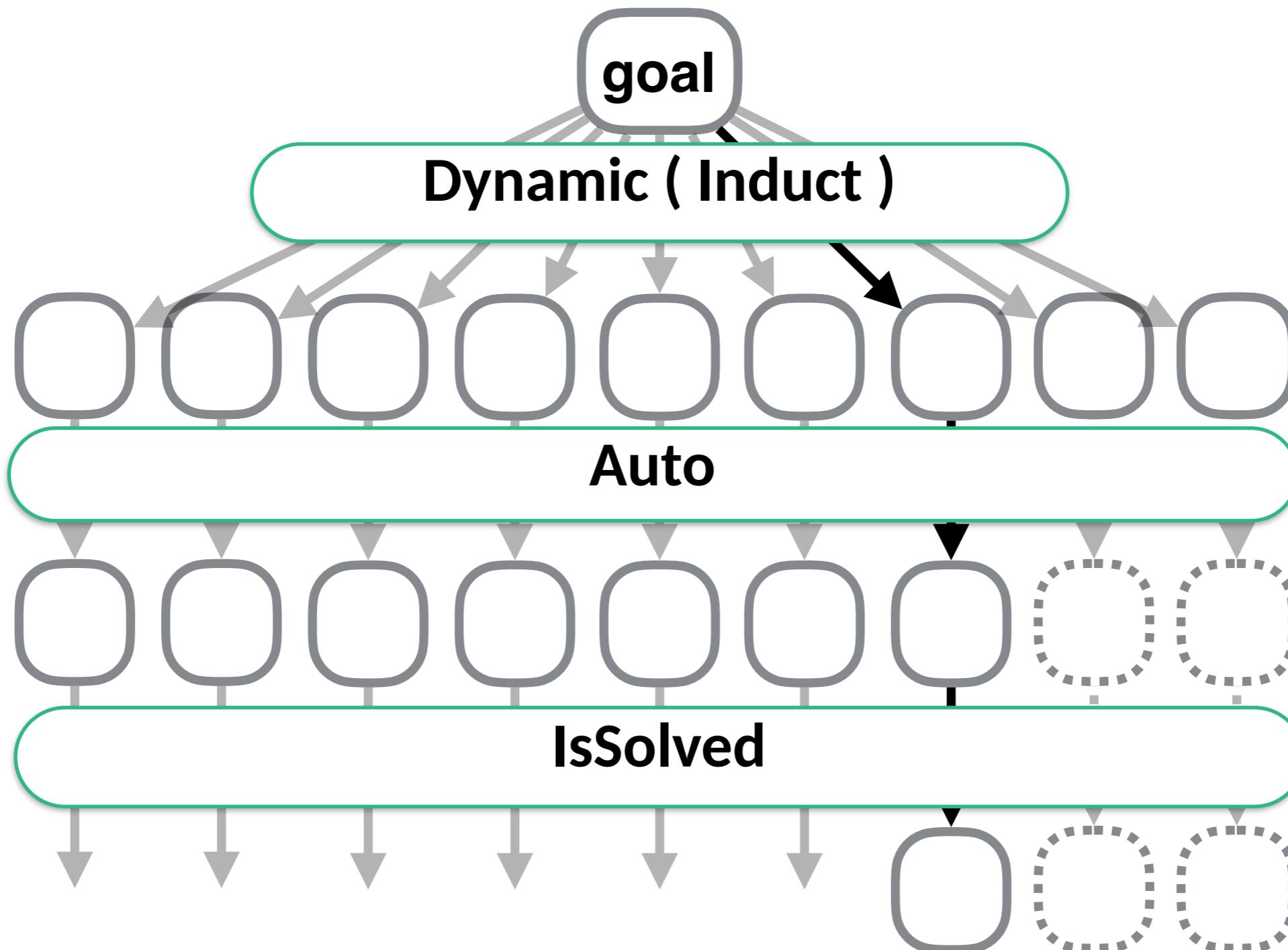
thub.com/data61/PSL/blob/master/slide/2020_ETH.pdf
PSL: Proof Strategy Language

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

DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

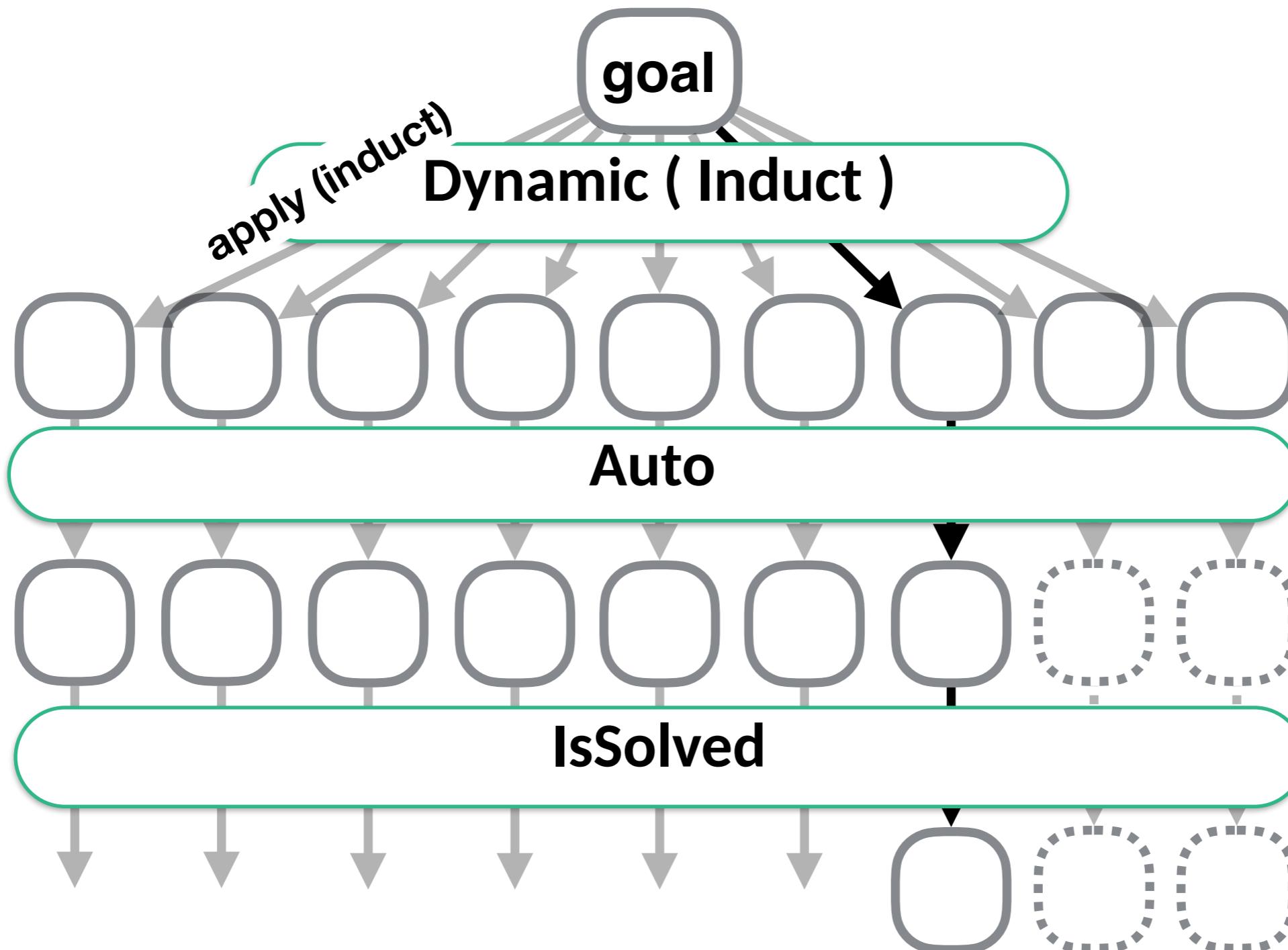
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

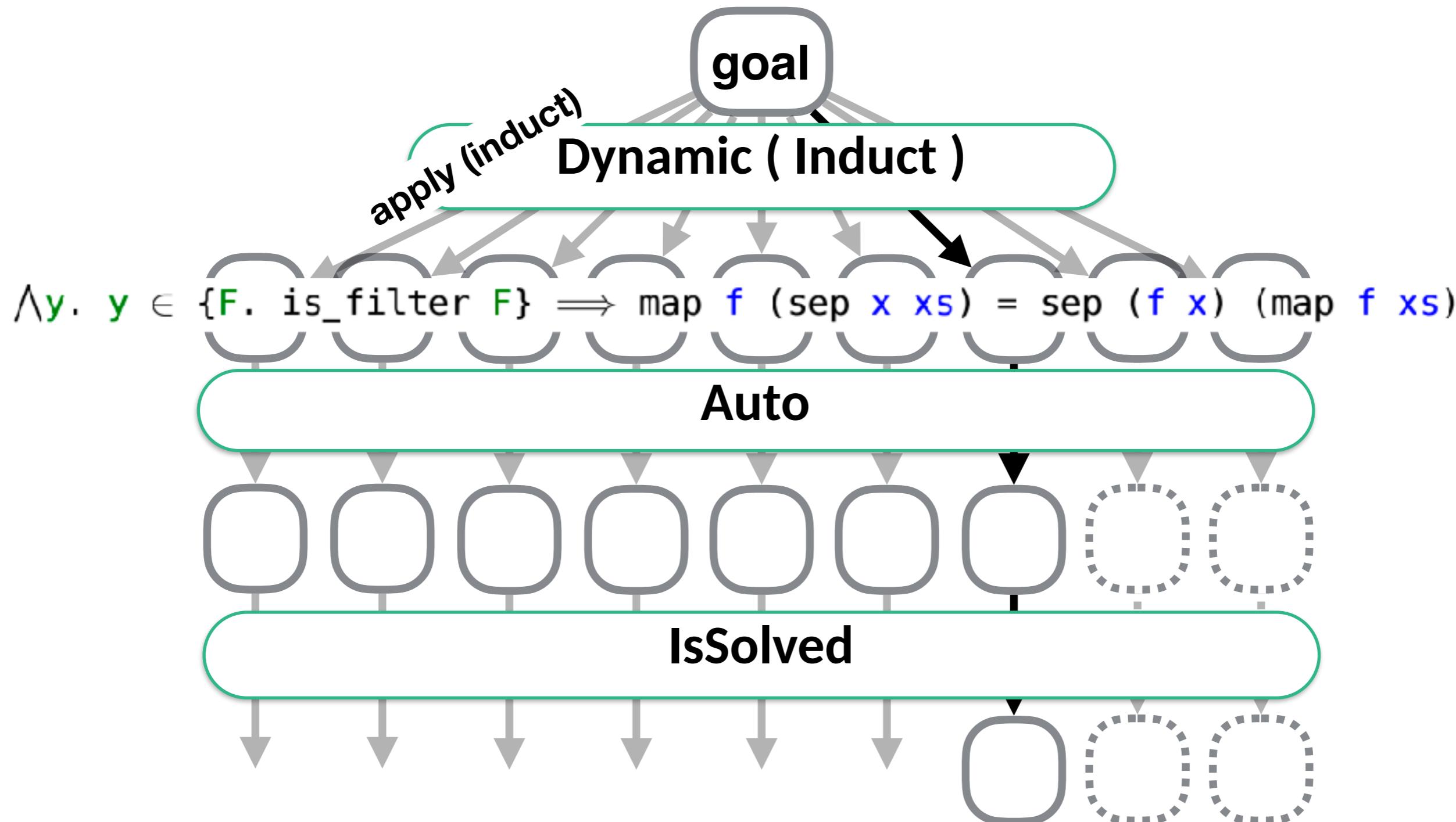
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

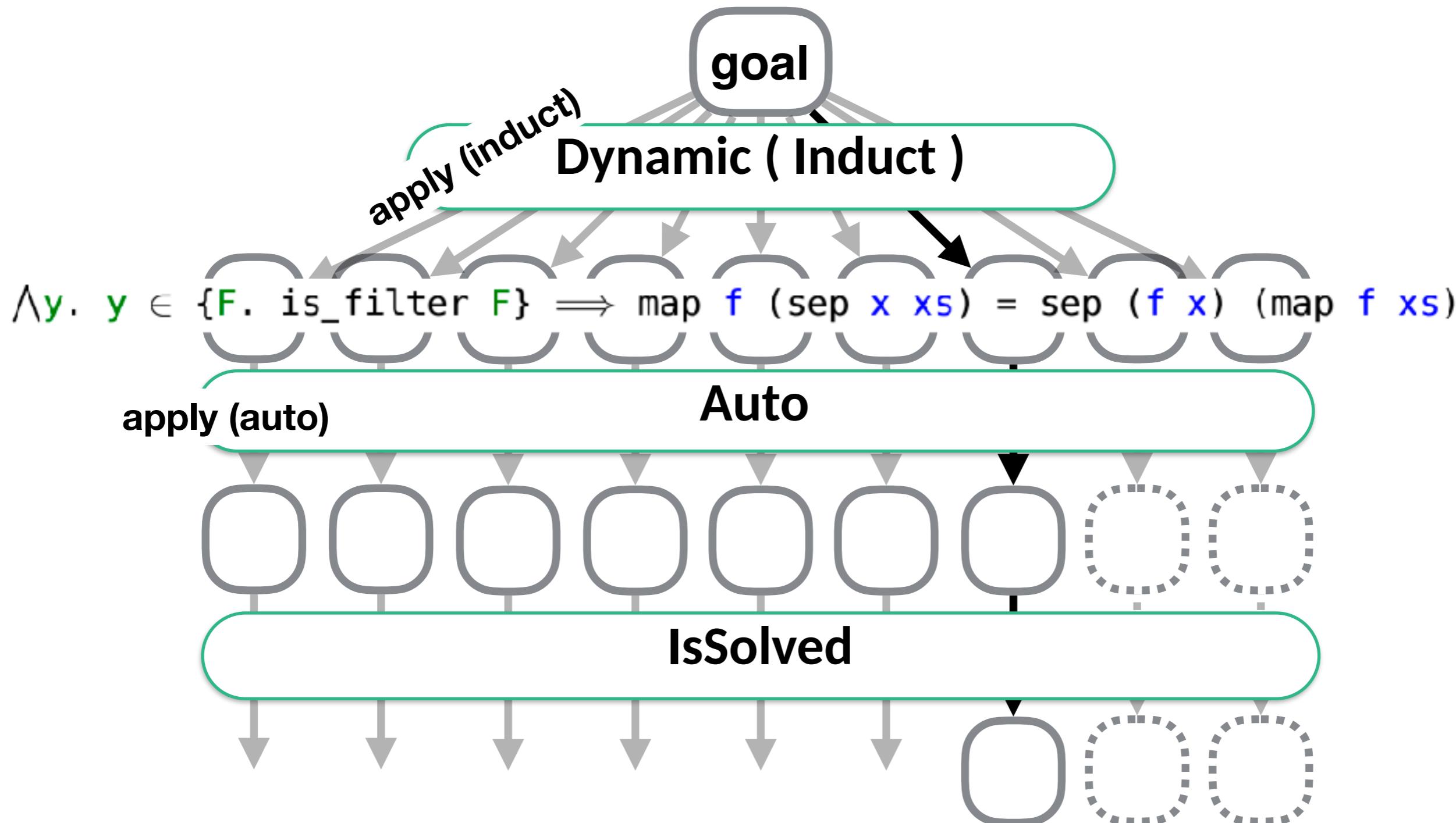
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

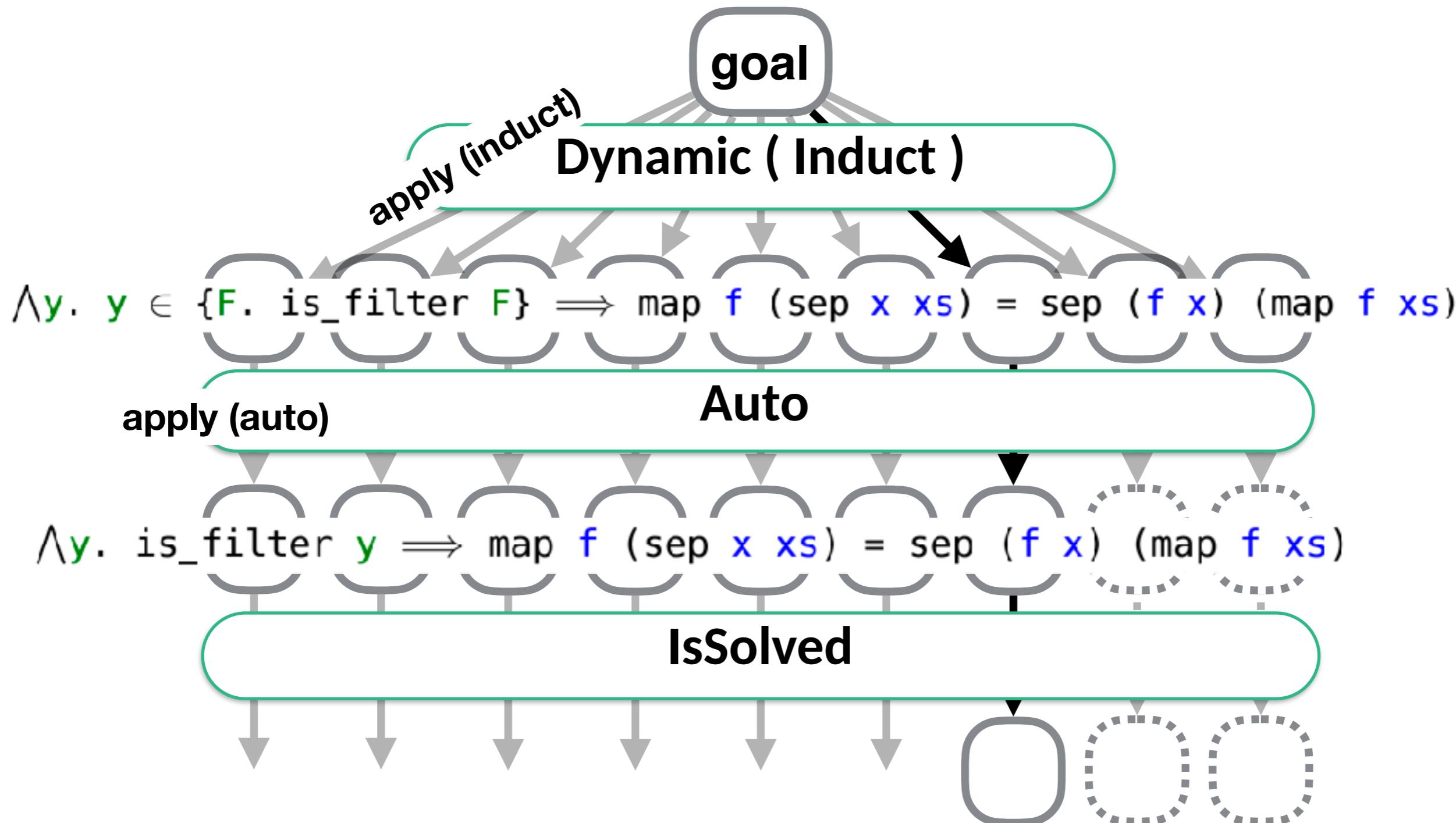
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

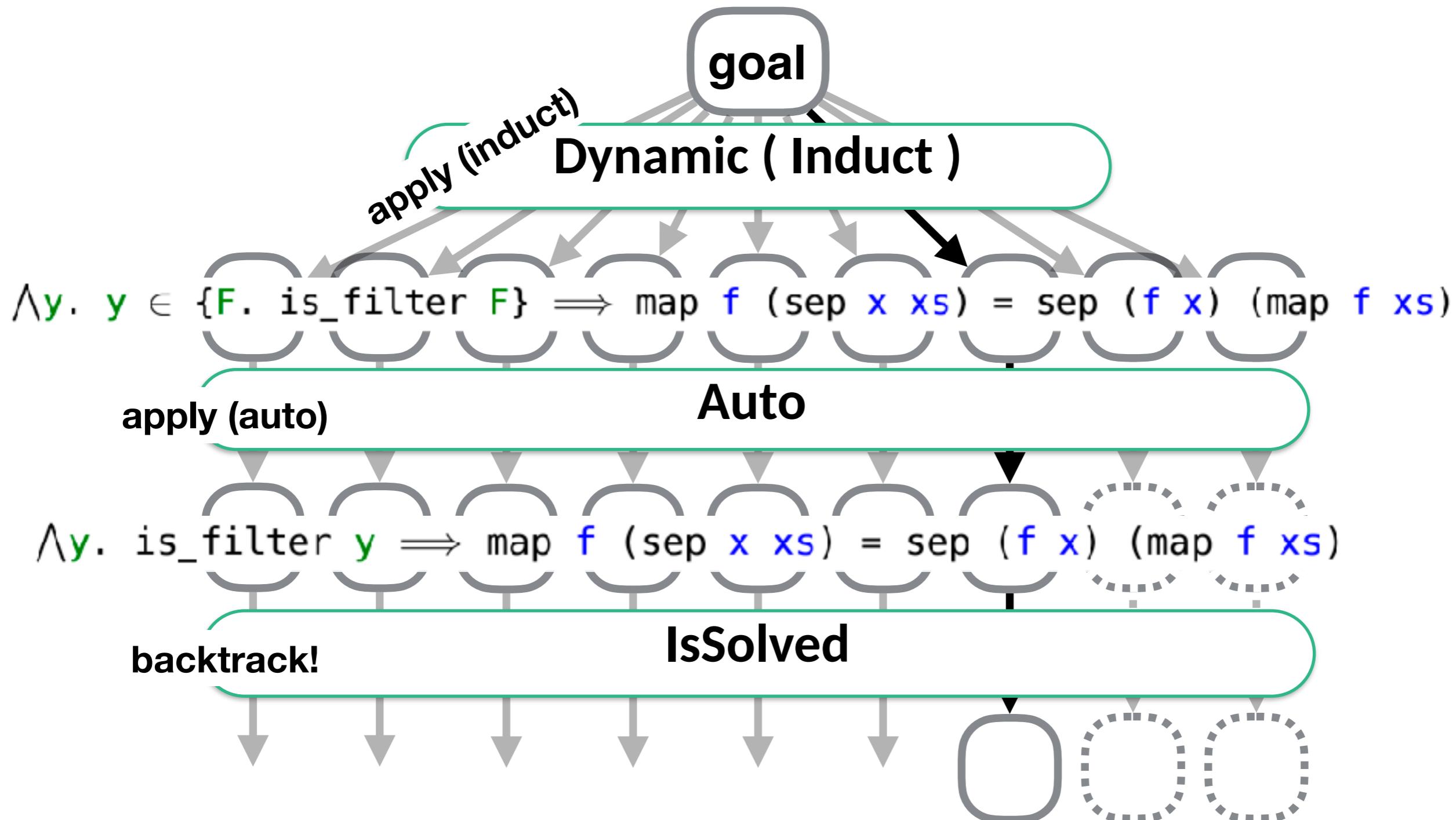
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

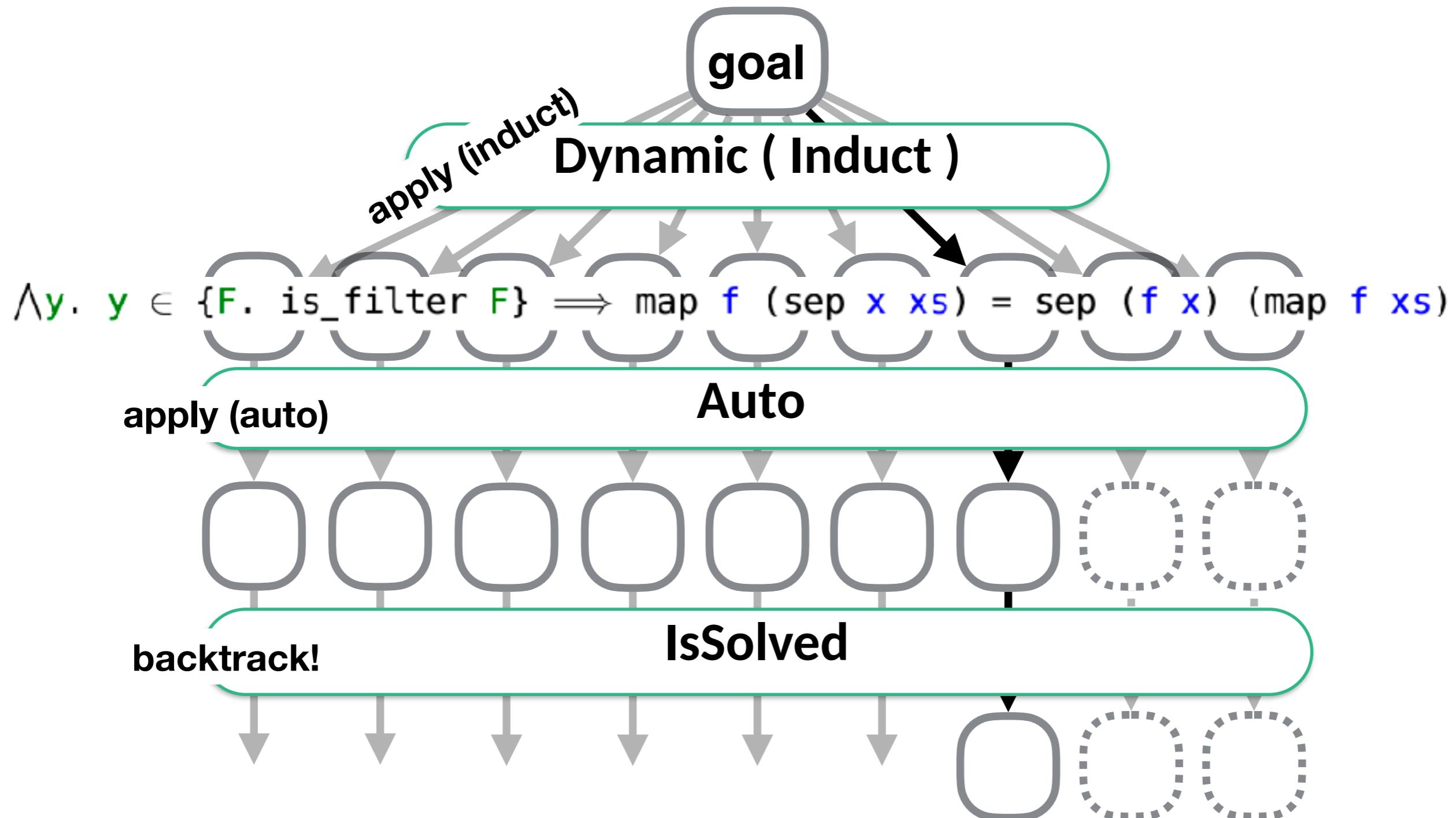
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

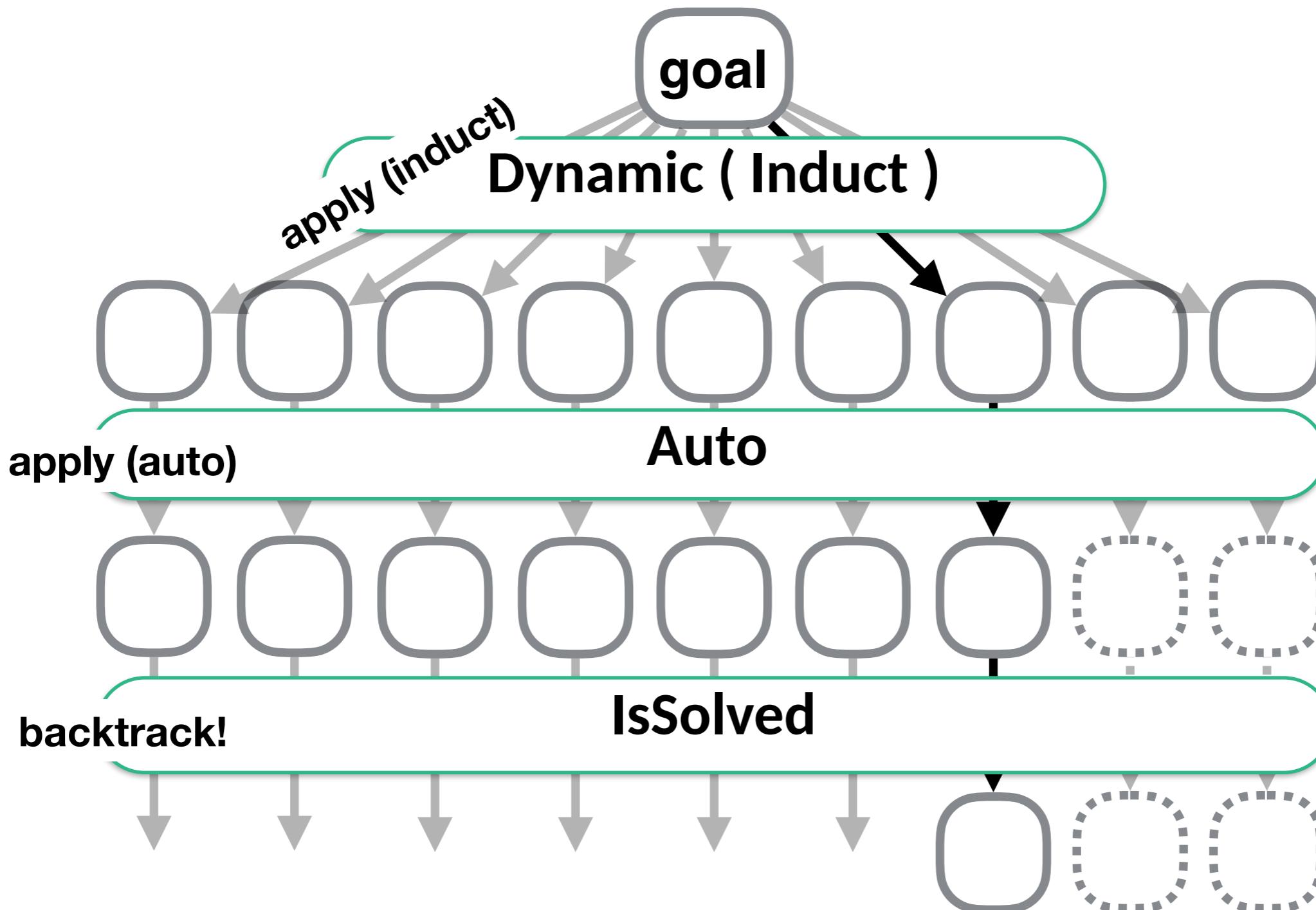
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

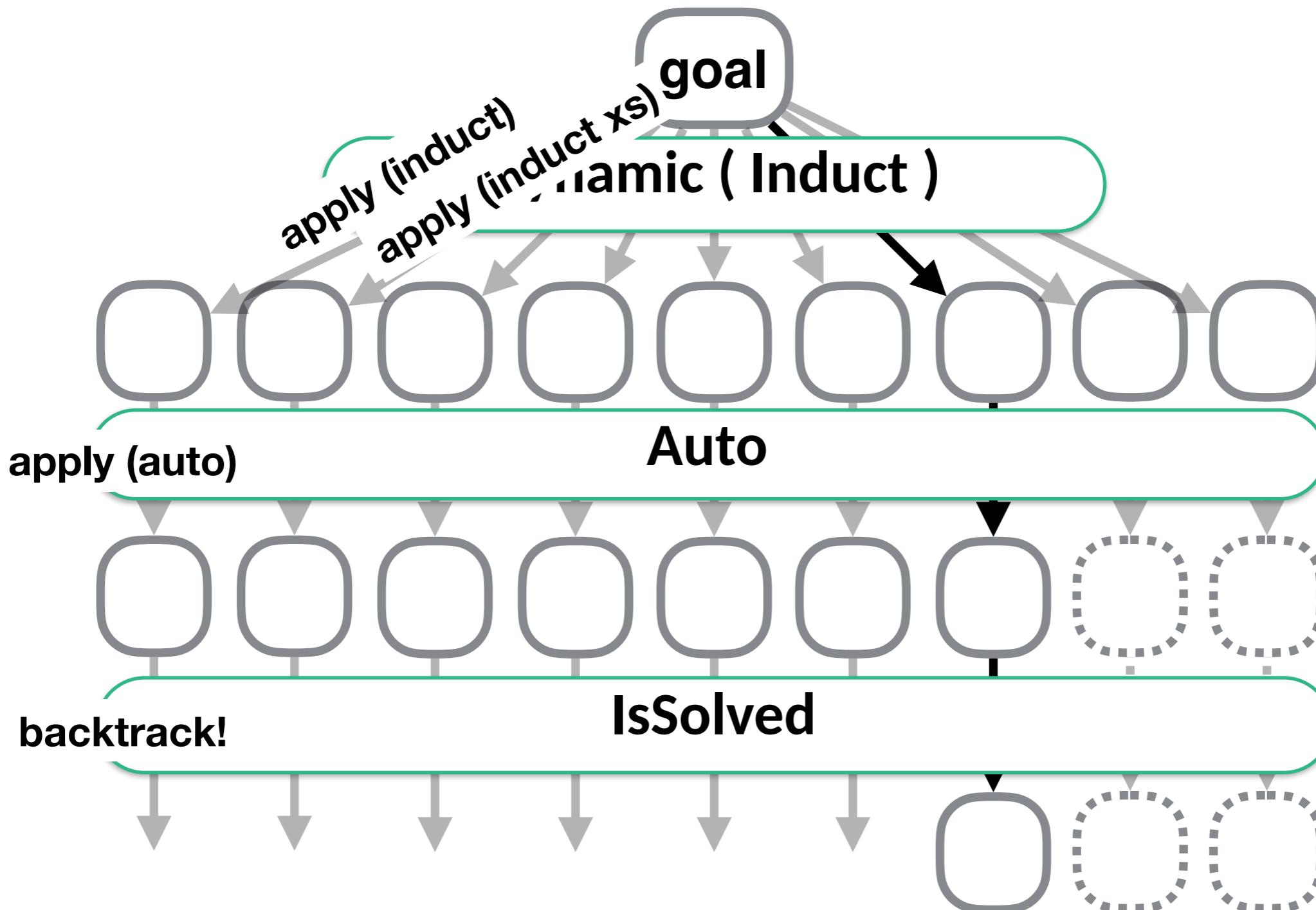
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

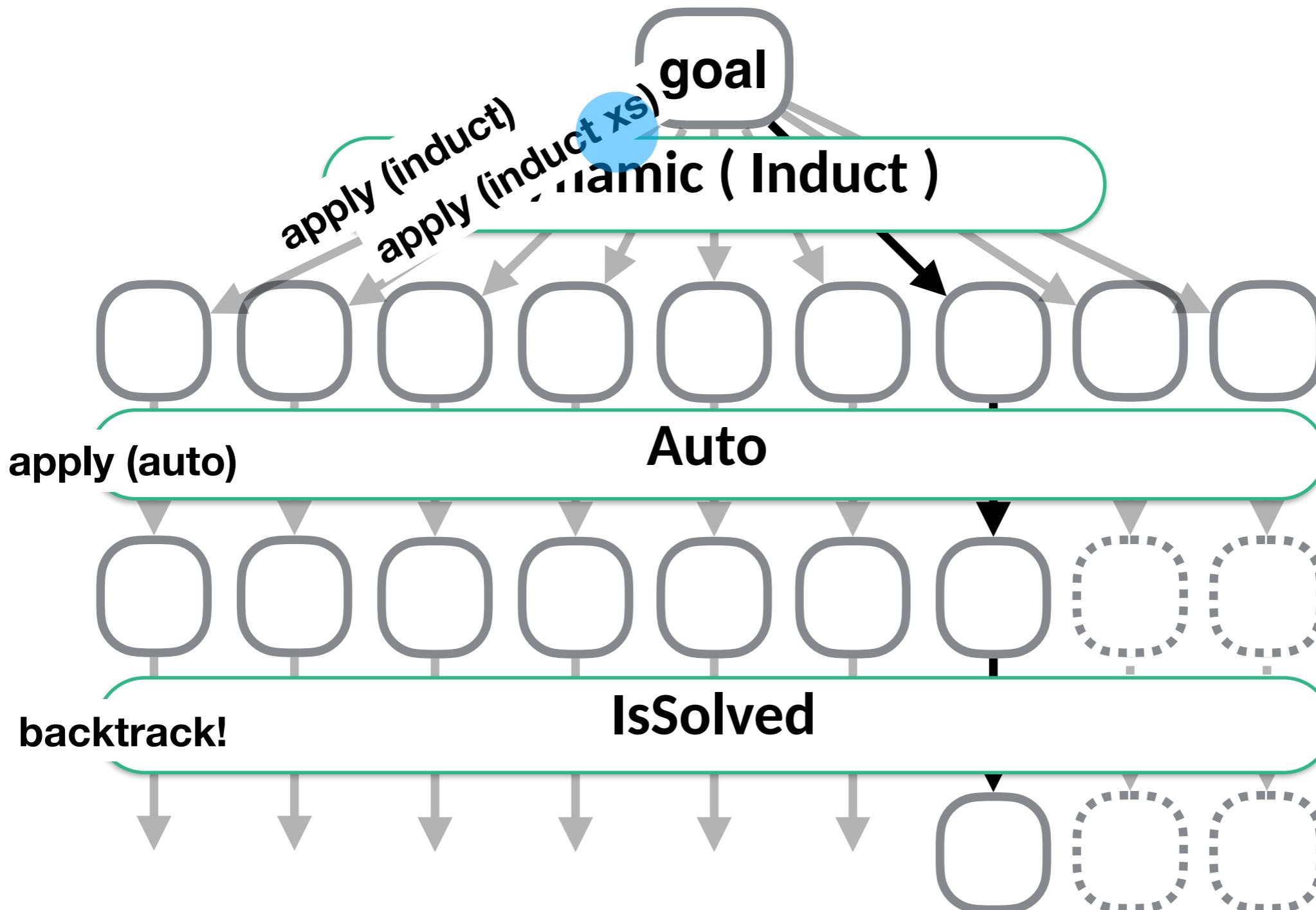
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

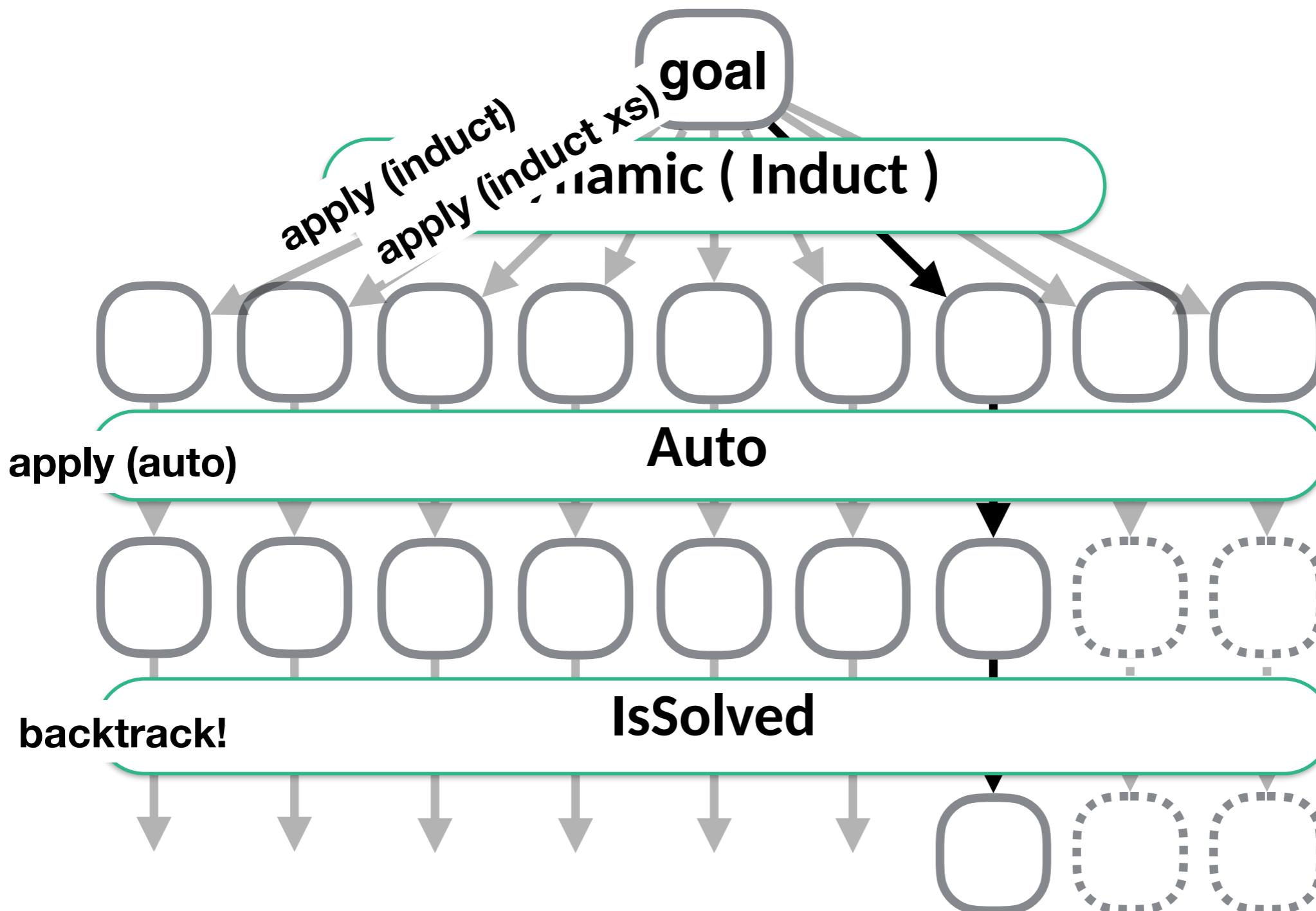
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

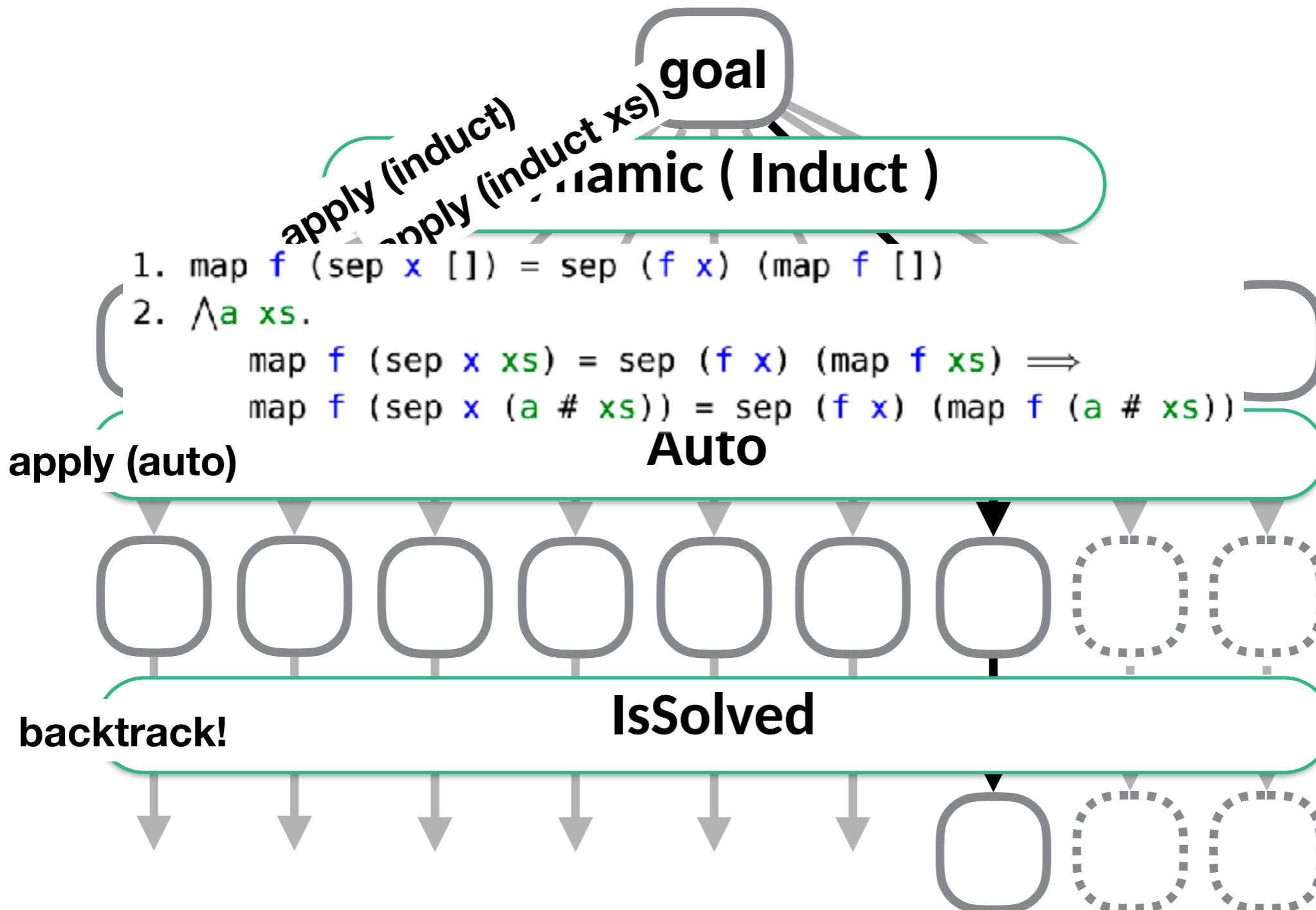
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

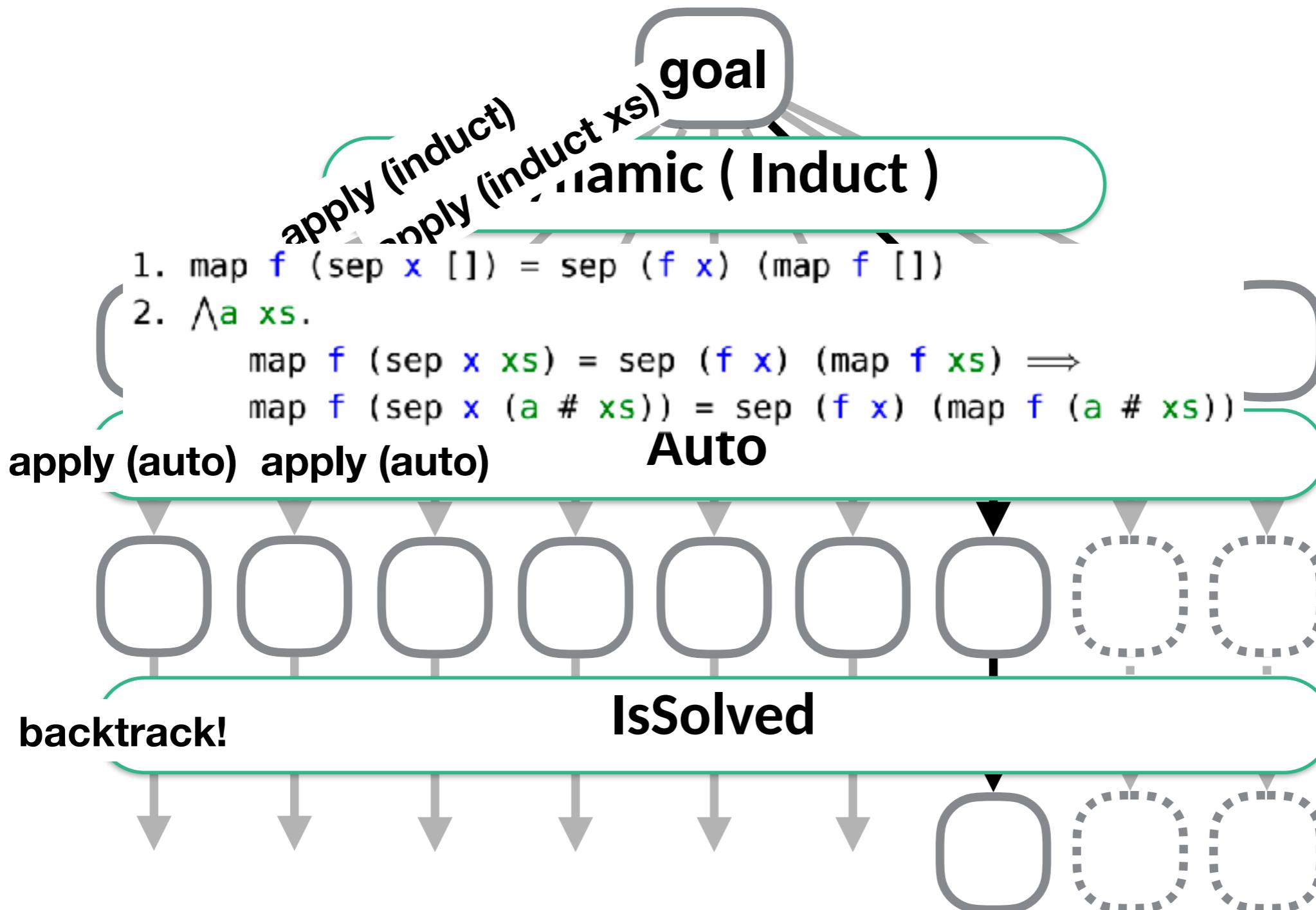
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

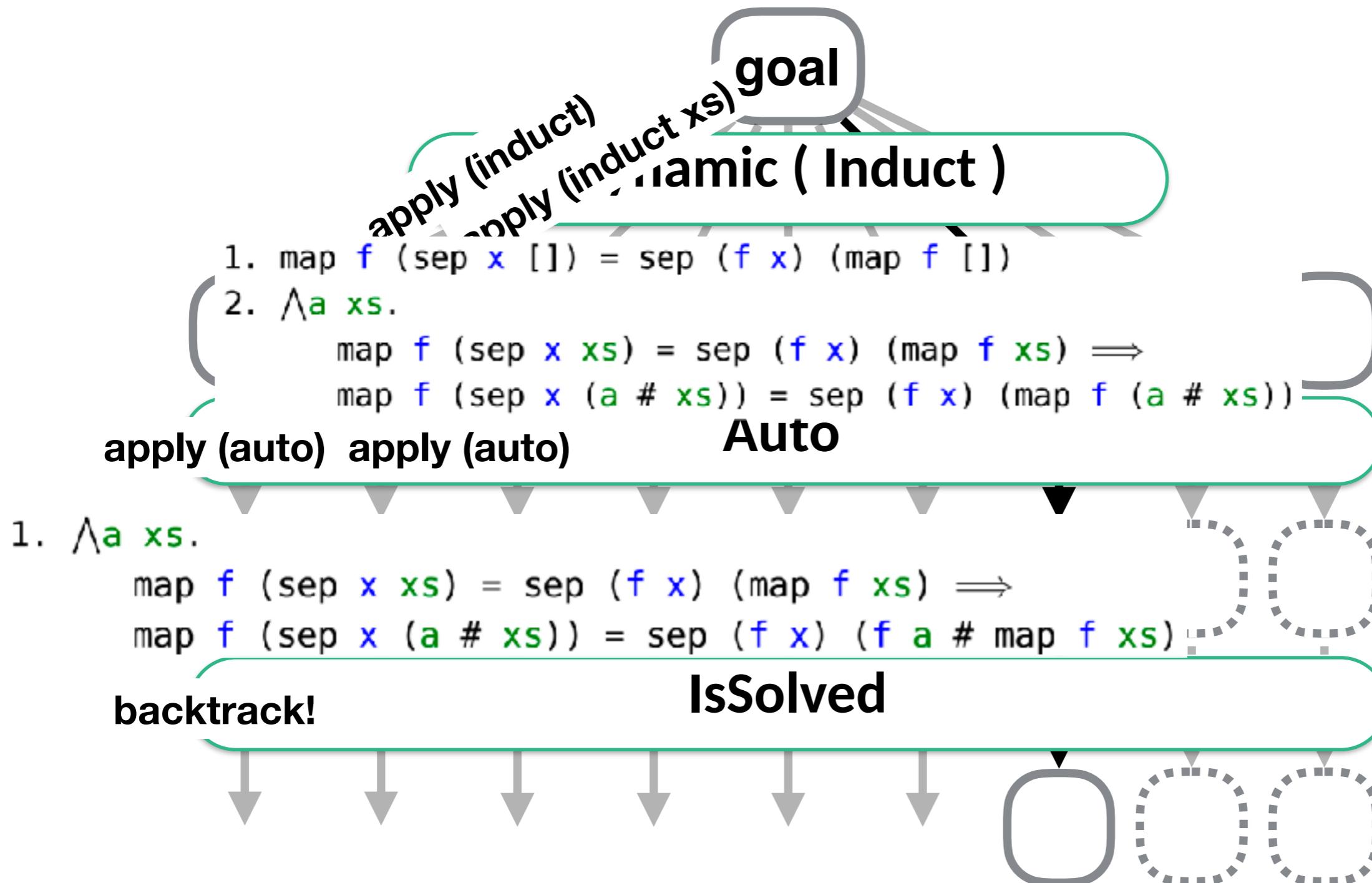
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

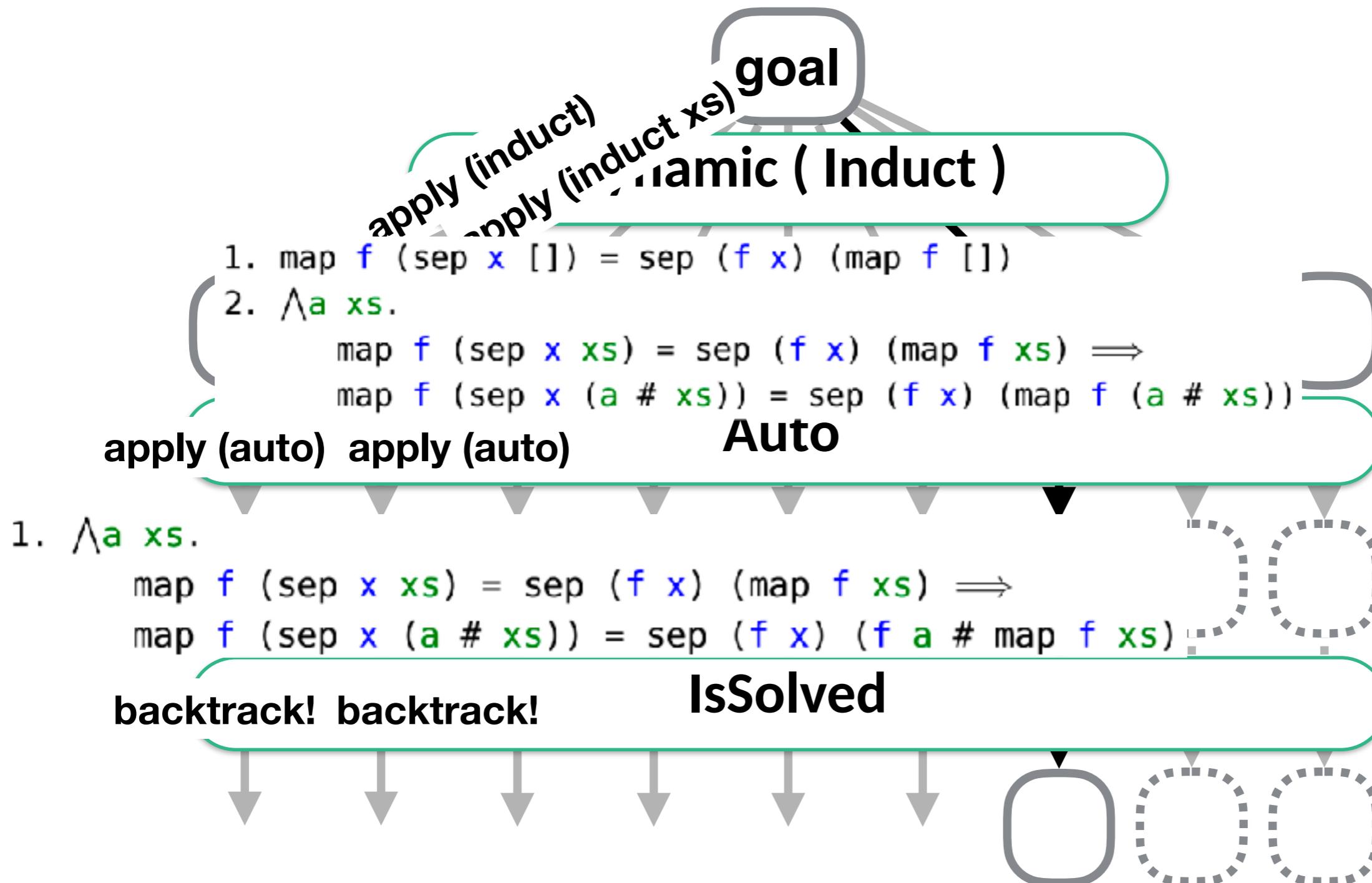
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.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]*)
```

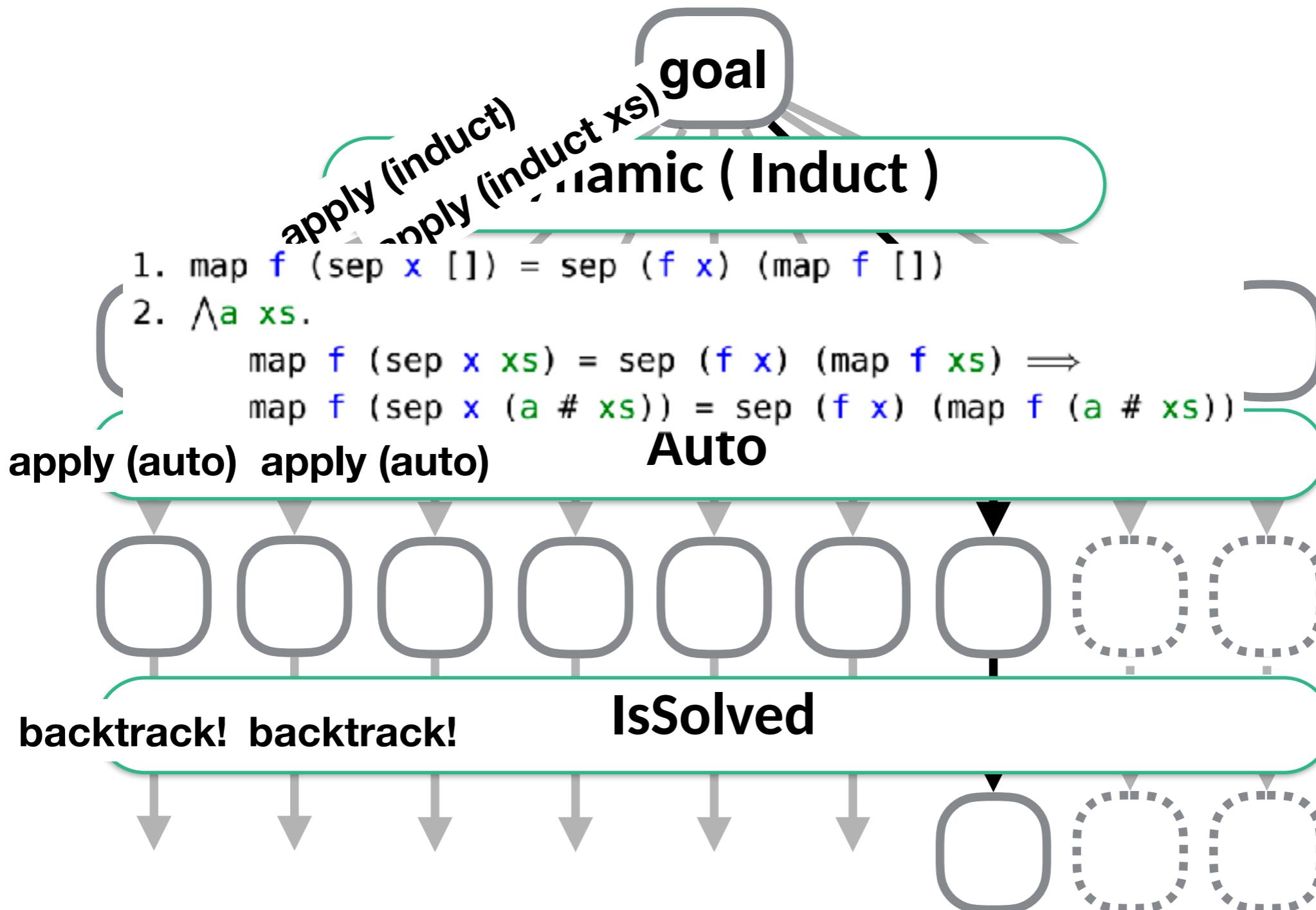


DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.pdf

PSL: Proof Strategy Language

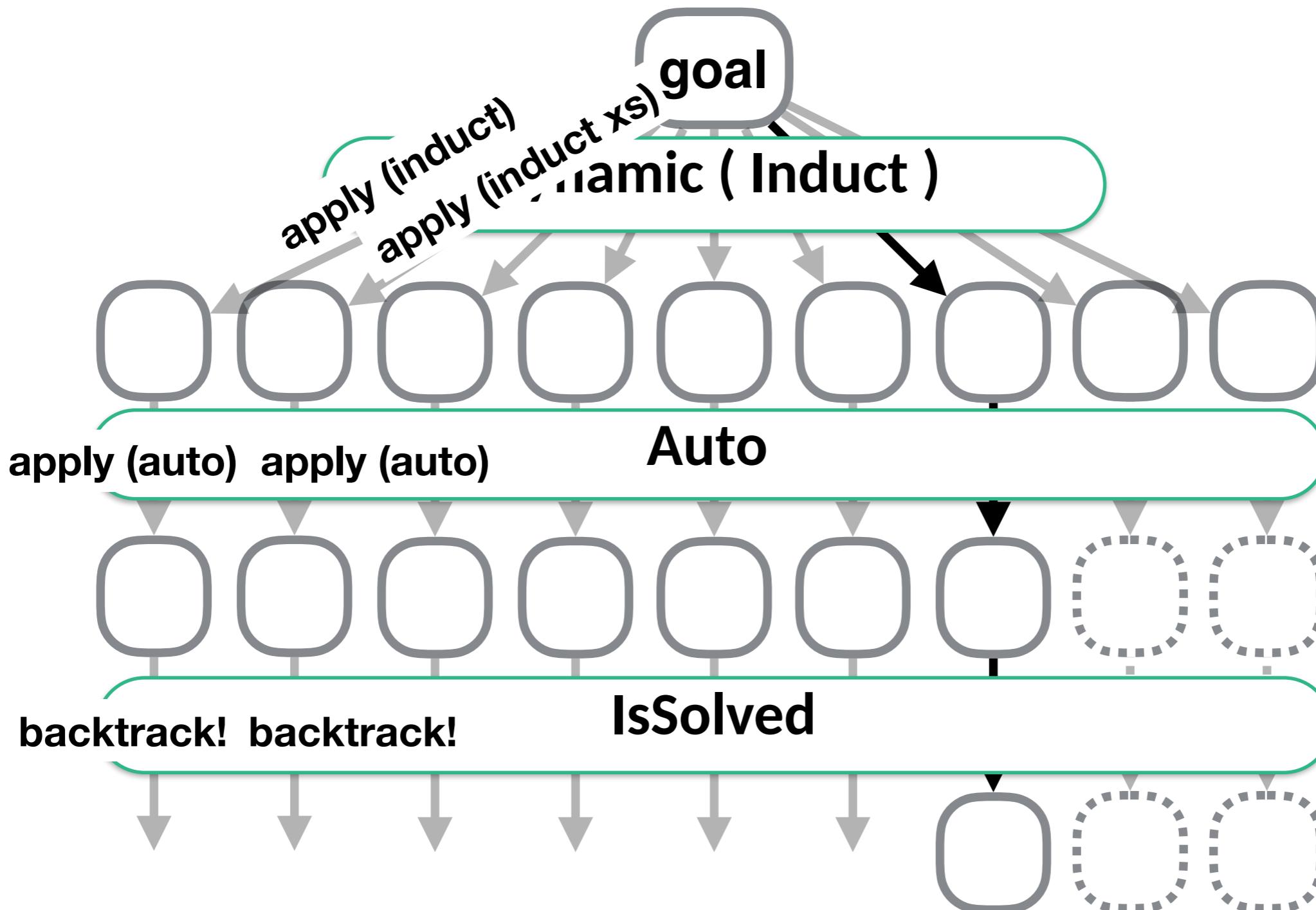
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

thub.com/data61/PSL/blob/master/slides/2020_ETH.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]*)
```

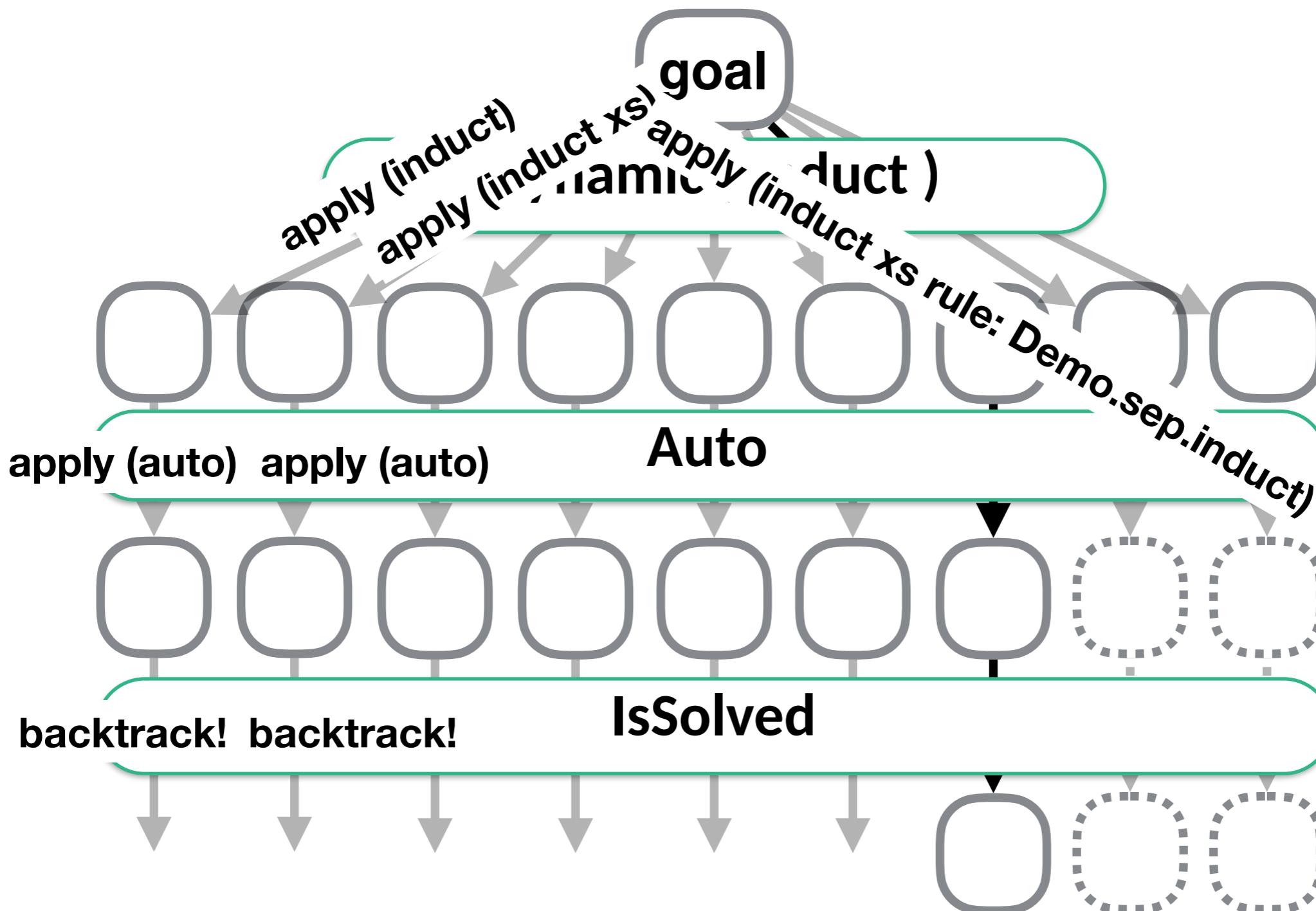


DEMO1

https://github.com/data61/PSL/blob/master/slides/2020_ETH.pdf

PSL: Proof Strategy Language

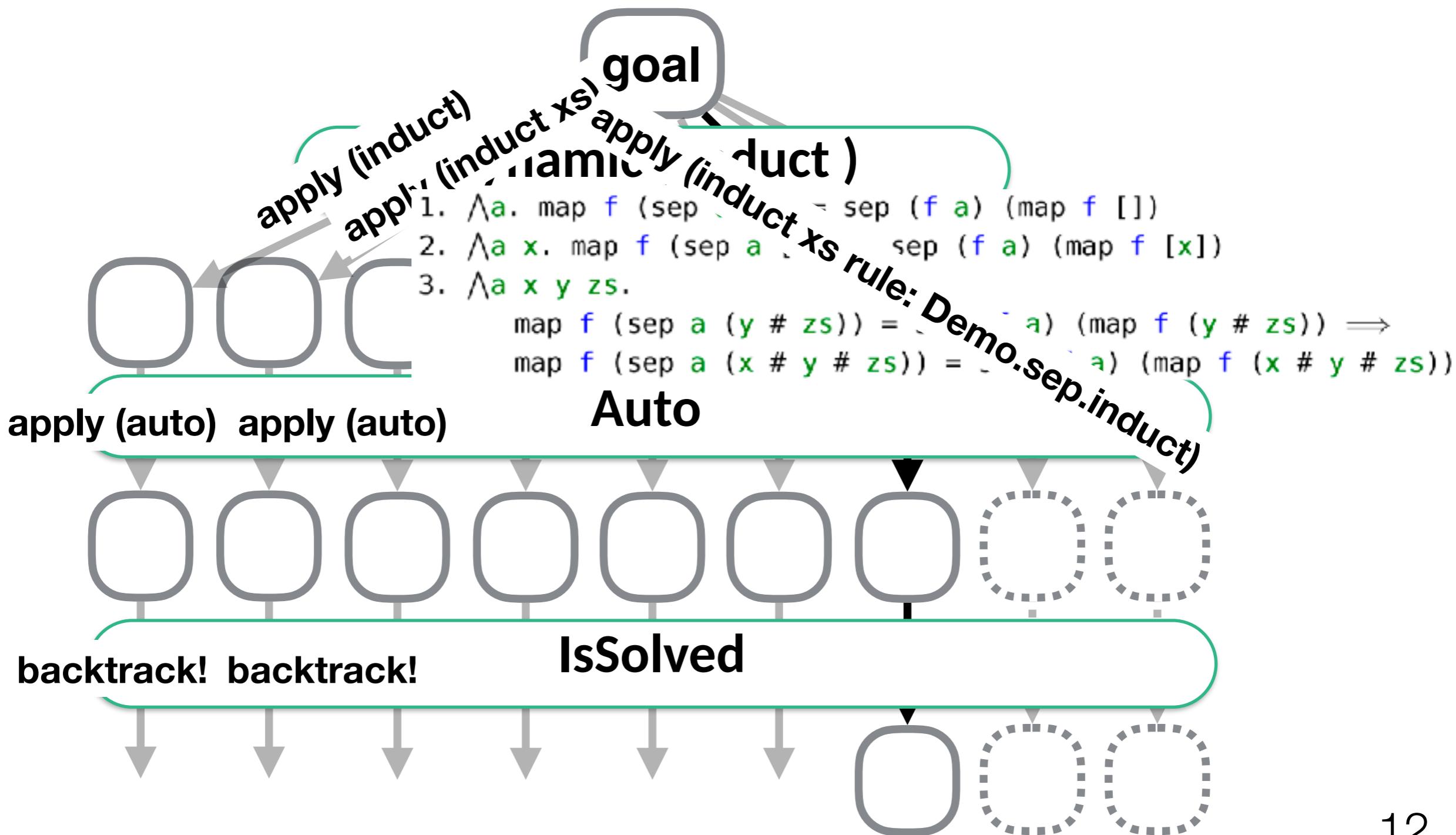
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

hub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

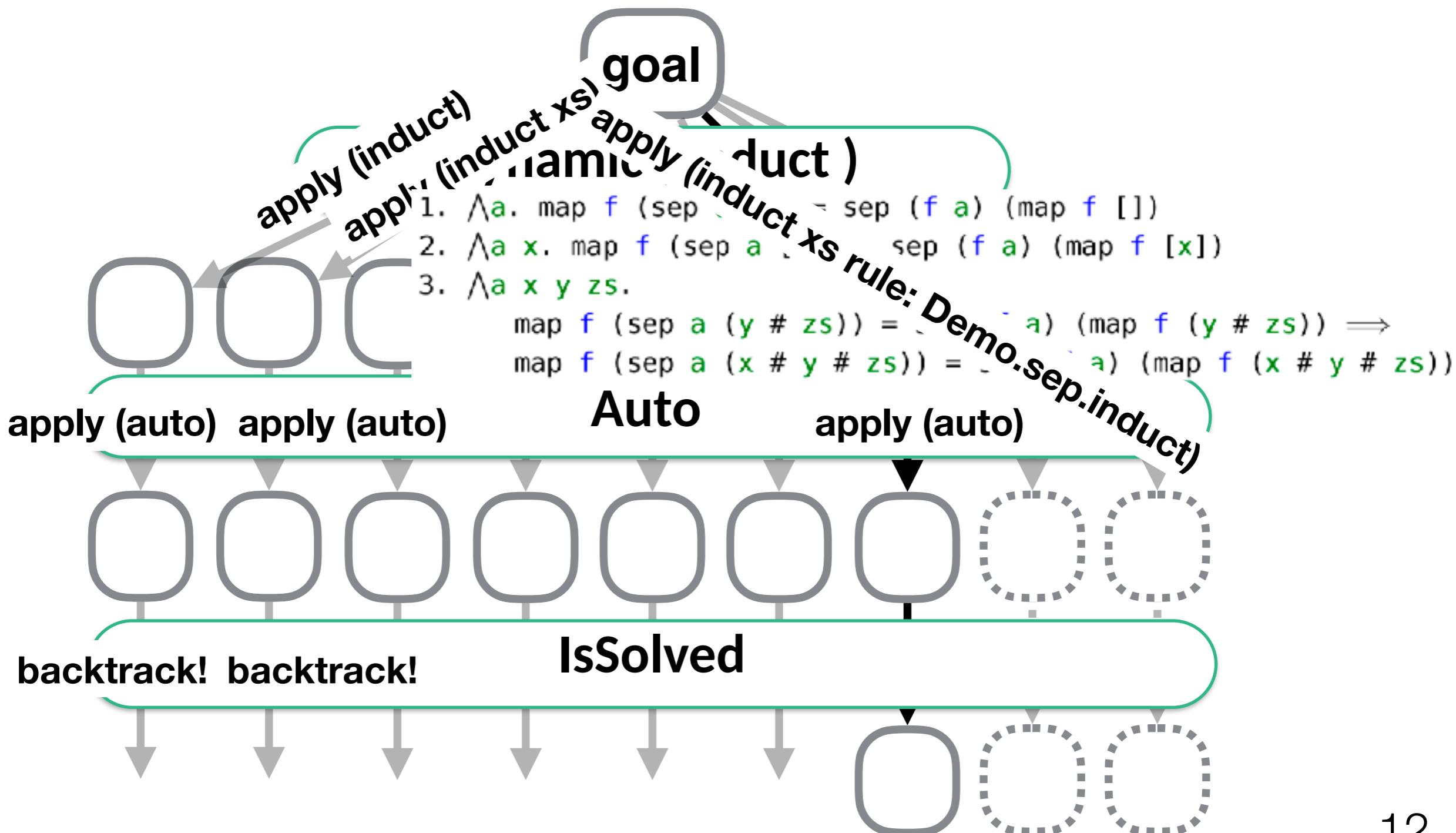
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

hub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

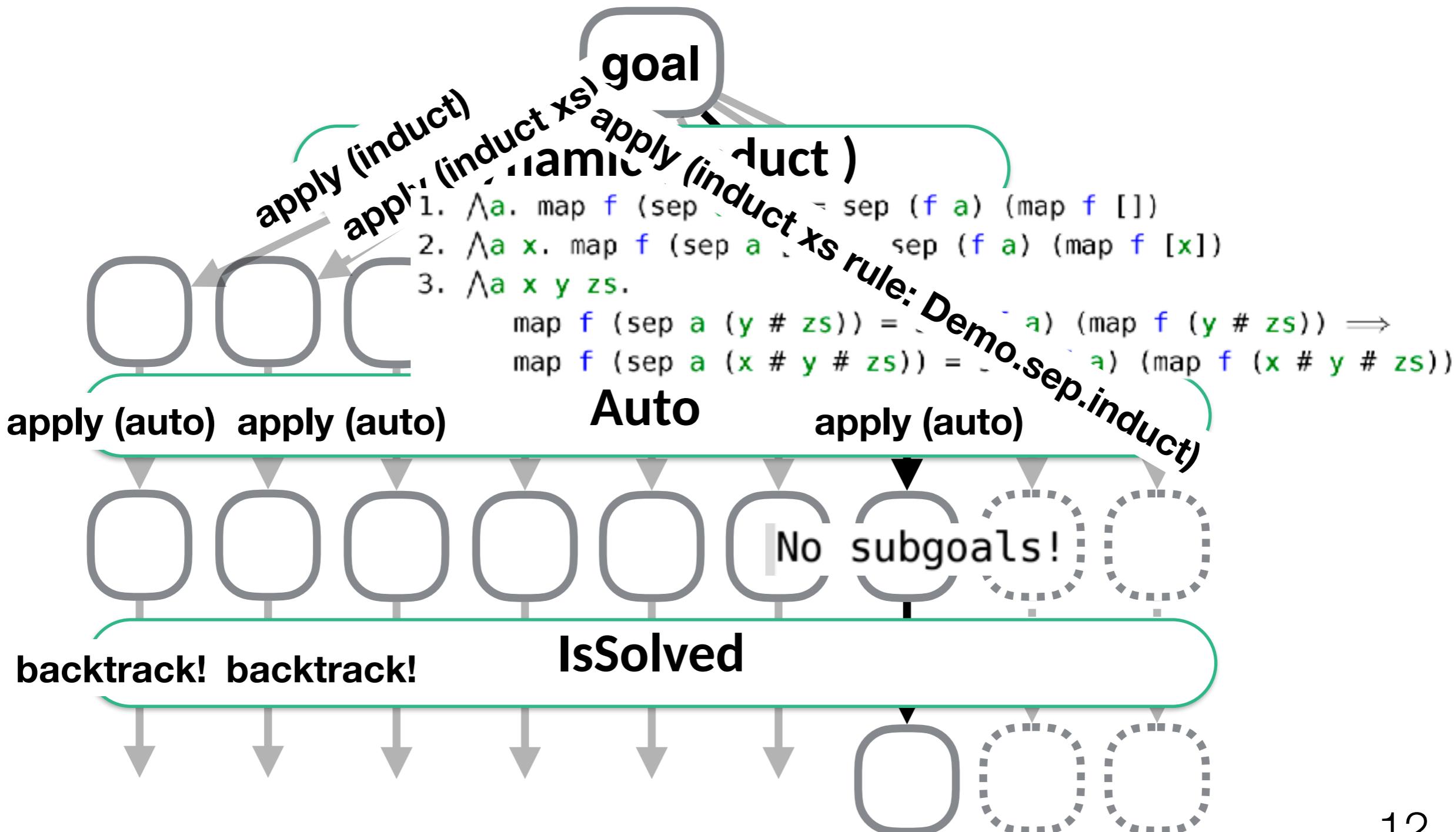
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

hub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

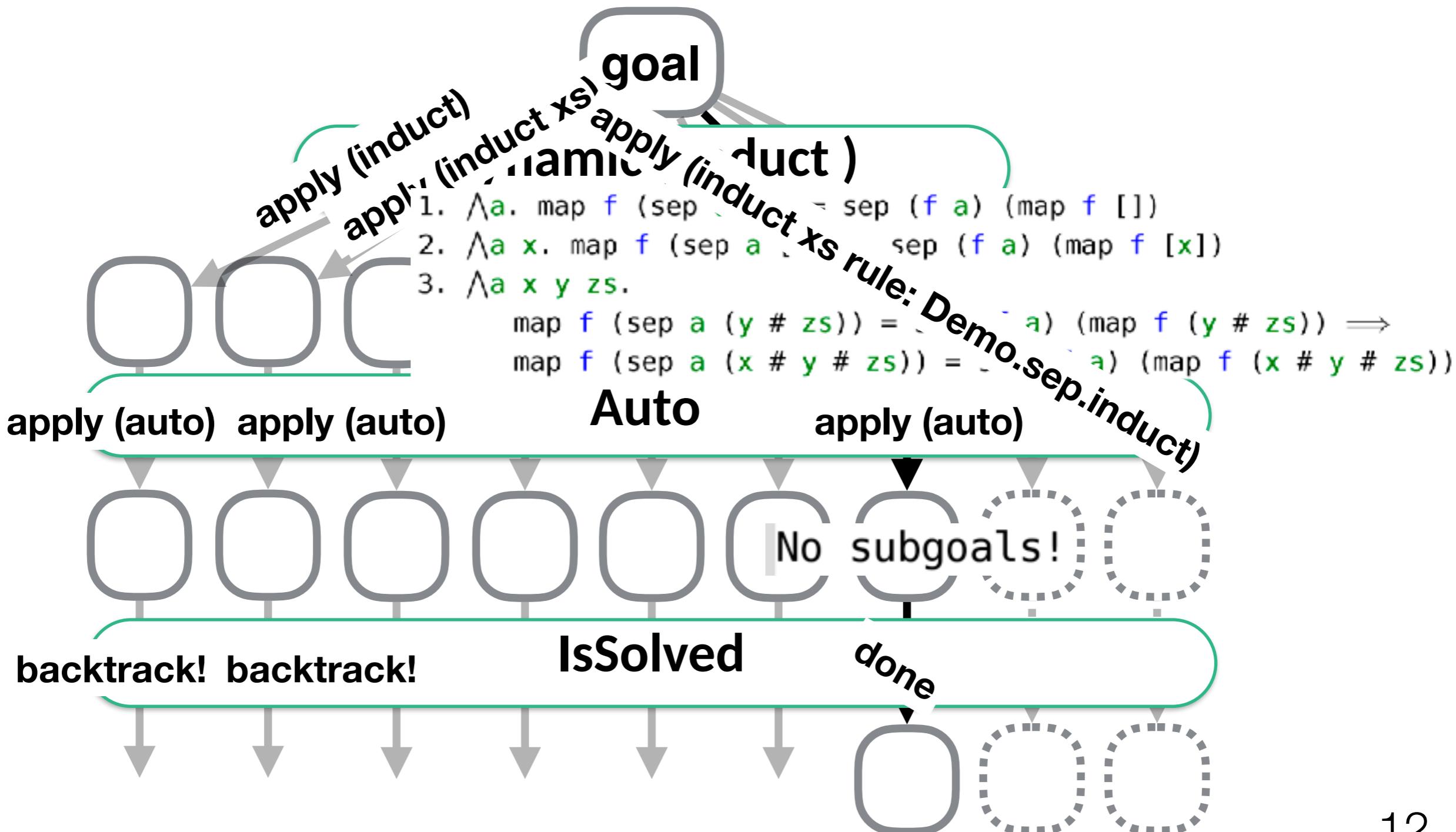
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

hub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

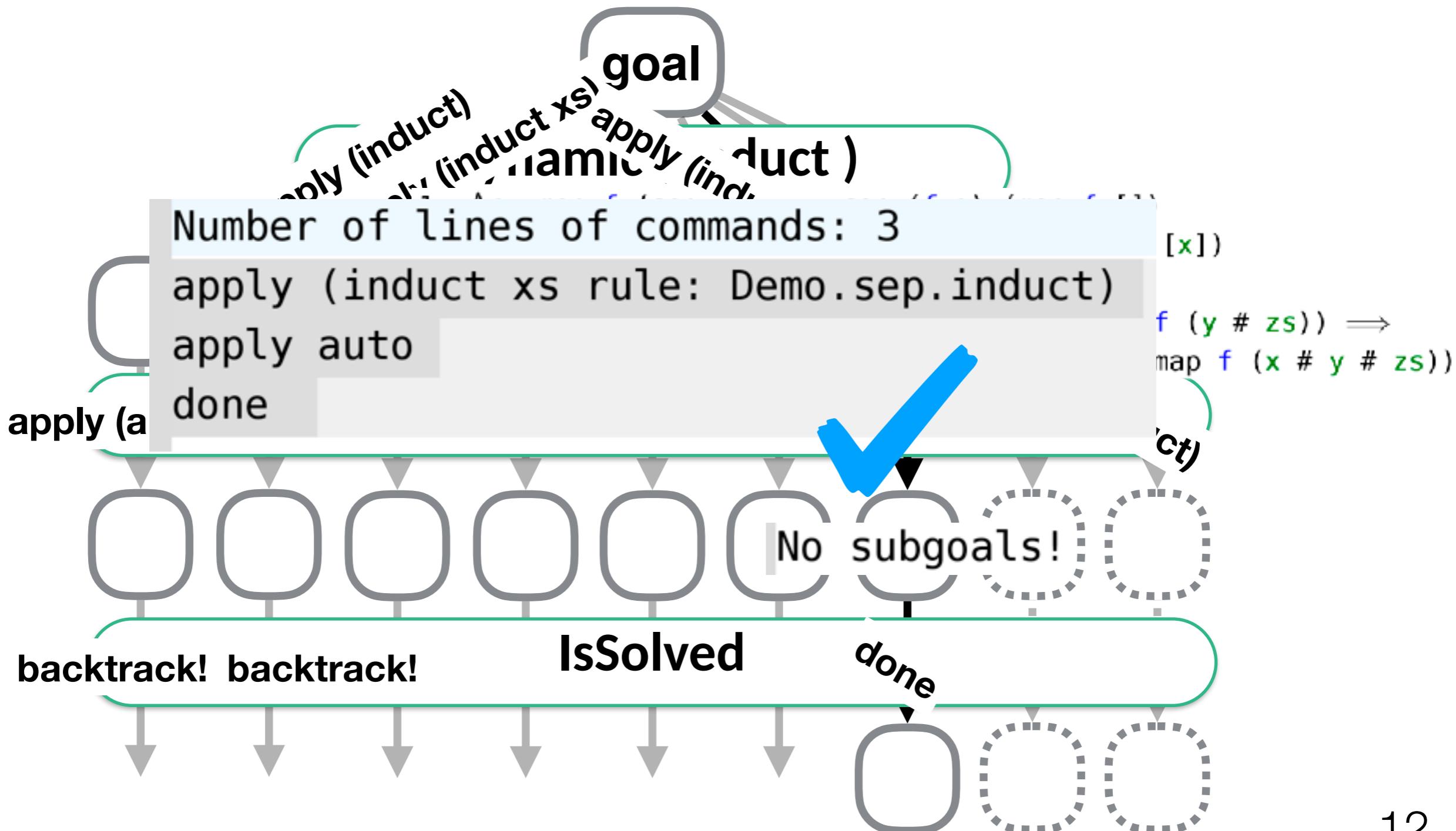
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

hub.com/data61/PSL/blob/master/slides/2020_ETH.pdf
PSL: Proof Strategy Language

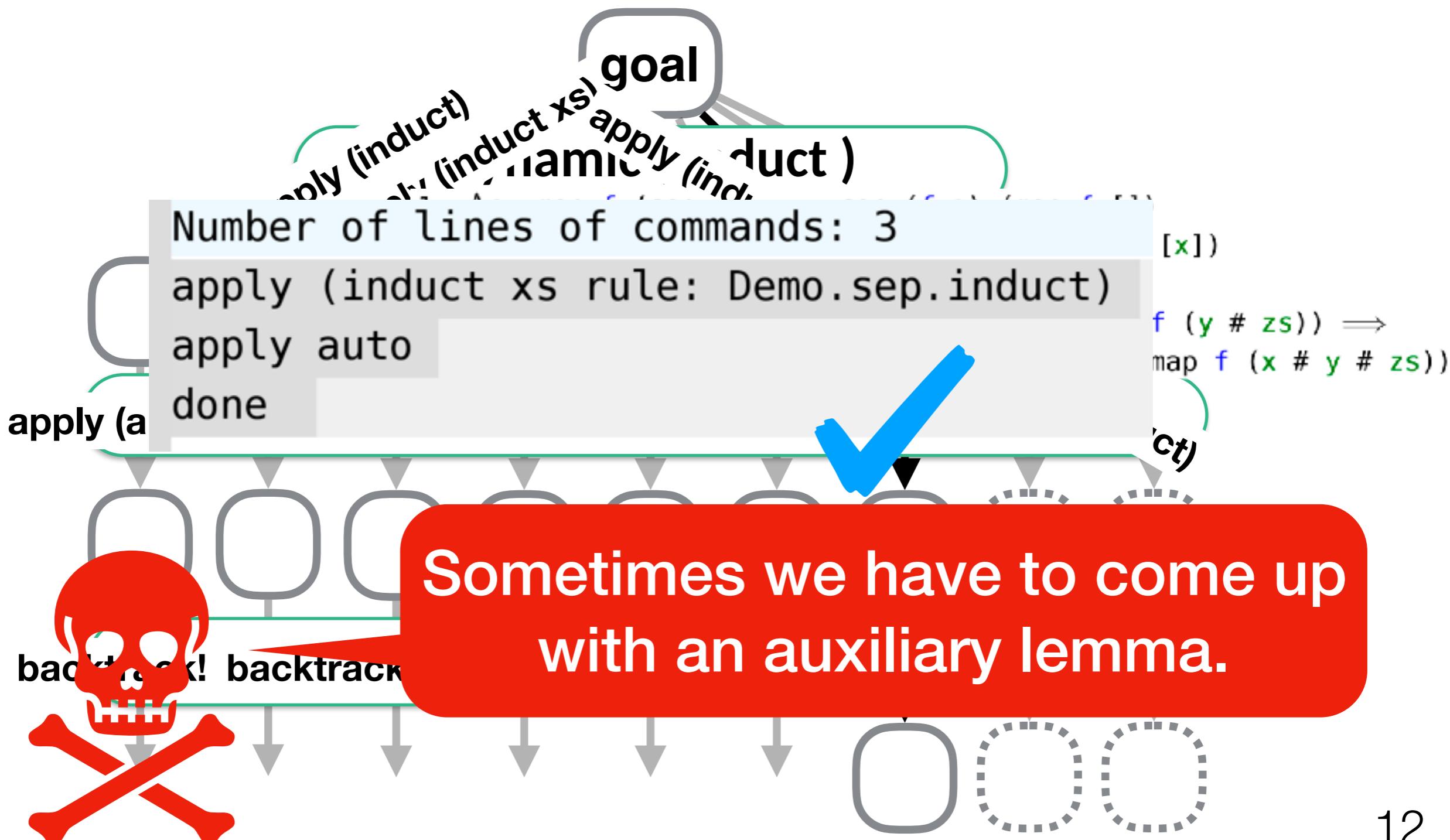
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

thub.com/data61/PSL/blob/master/slide/2020_ETH.pdf

The screenshot shows the Isabelle/HOL proof assistant interface. The top part displays a file named "Example.thy" containing ML code for defining list reversal functions. The code includes primrec declarations for `rev` and `itrev`, and a strategy definition for `DInd`. A lemma is also defined:

```
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
```

The lemma `"itrev xs [] = rev xs"` is highlighted with a yellow background. The bottom part of the interface shows a proof state with a single subgoal:

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

DEMO2

thub.com/data61/PSL/blob/master/slide/2020_ETH.pdf

The screenshot shows the Isabelle proof assistant interface. The main window displays the following ML code:

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

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

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

The code defines two primitive recursive functions: `rev` and `itrev`. The `rev` function takes a list and returns its reverse. The `itrev` function takes a list and a result so far, and returns the reverse by prepending the first element to the result of reversing the rest. A strategy `DInd` is defined using `Thens` with three tactics: `Dynamic (Induct)`, `Auto`, and `IsSolved`. A lemma is stated: `"itrev xs [] = rev xs"`. The command `find_proof DInd` is issued to find a proof for this lemma.

empty sequence. no proof found.

DEMO2

thub.com/data61/PSL/blob/master/slide/2020_ETH.pdf

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

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

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

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

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

The code defines two recursive functions: `rev` and `itrev`. It also defines three proof strategies: `DInd`, `CDInd`, and `DInd_Or_CDInd`. The `Lemma` section contains a conjecture to be proved.

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

thub.com/data61/PSL/blob/master/slide/2020_ETH.pdf

The screenshot shows the PSL/PDT interface with the following code:

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

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

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

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

A yellow circle highlights the word "Conjecture" in the strategy definition for 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

hub.com/data61/PSL/blob/master/slide/2020_ETH.pdf

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

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

38
39 strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved]
40 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
```

Output Query Sledgehammer Symbols

DEMO2

hub.com/data61/PSL/blob/master/slide/2020_ETH.pdf

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

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

38
39 strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved]
40 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

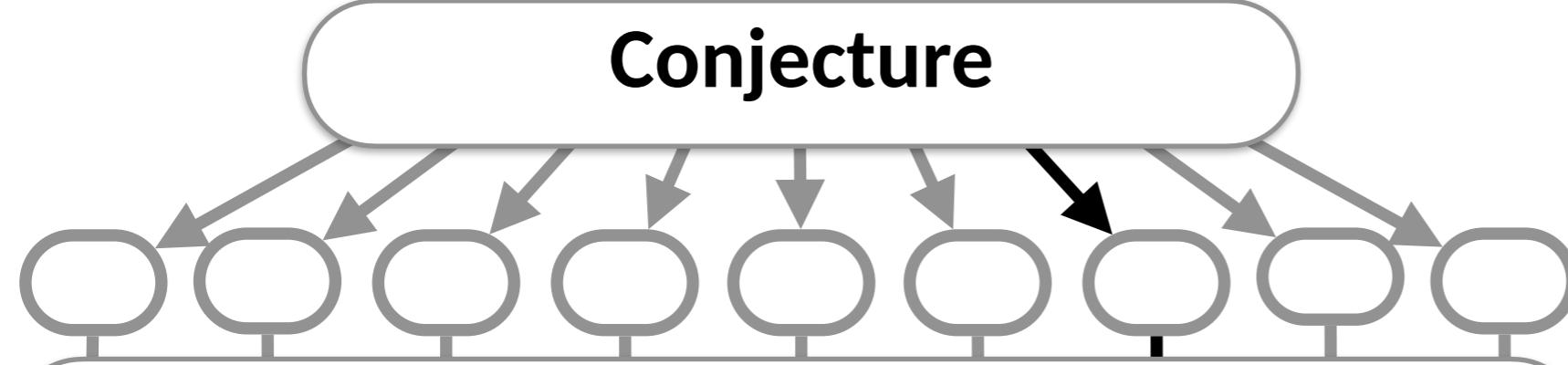
strategy CDInd = Thens [Conjecture, Fastforce, Quickcheck, DInd]

DEMO2

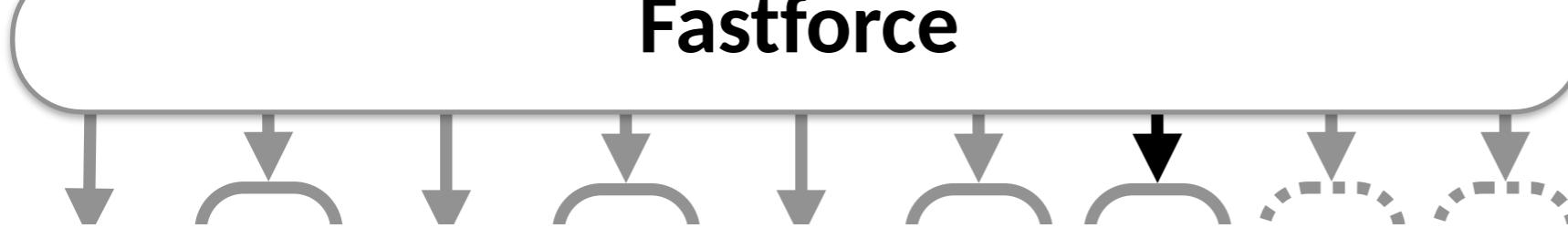
```
goal (1 subgoal):  
1. itrev xs [] = rev xs
```

goal

Conjecture



Fastforce

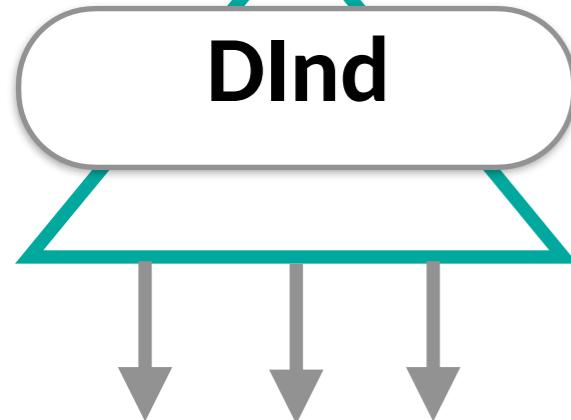


```
strategy CDInd = Thens [Conjecture, Fastforce, Quickcheck, DInd]
```

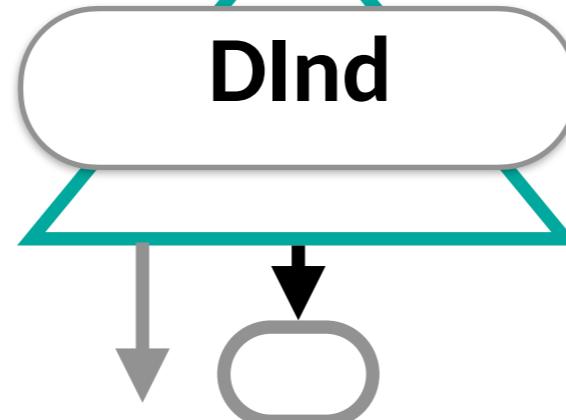
Quickcheck



DInd

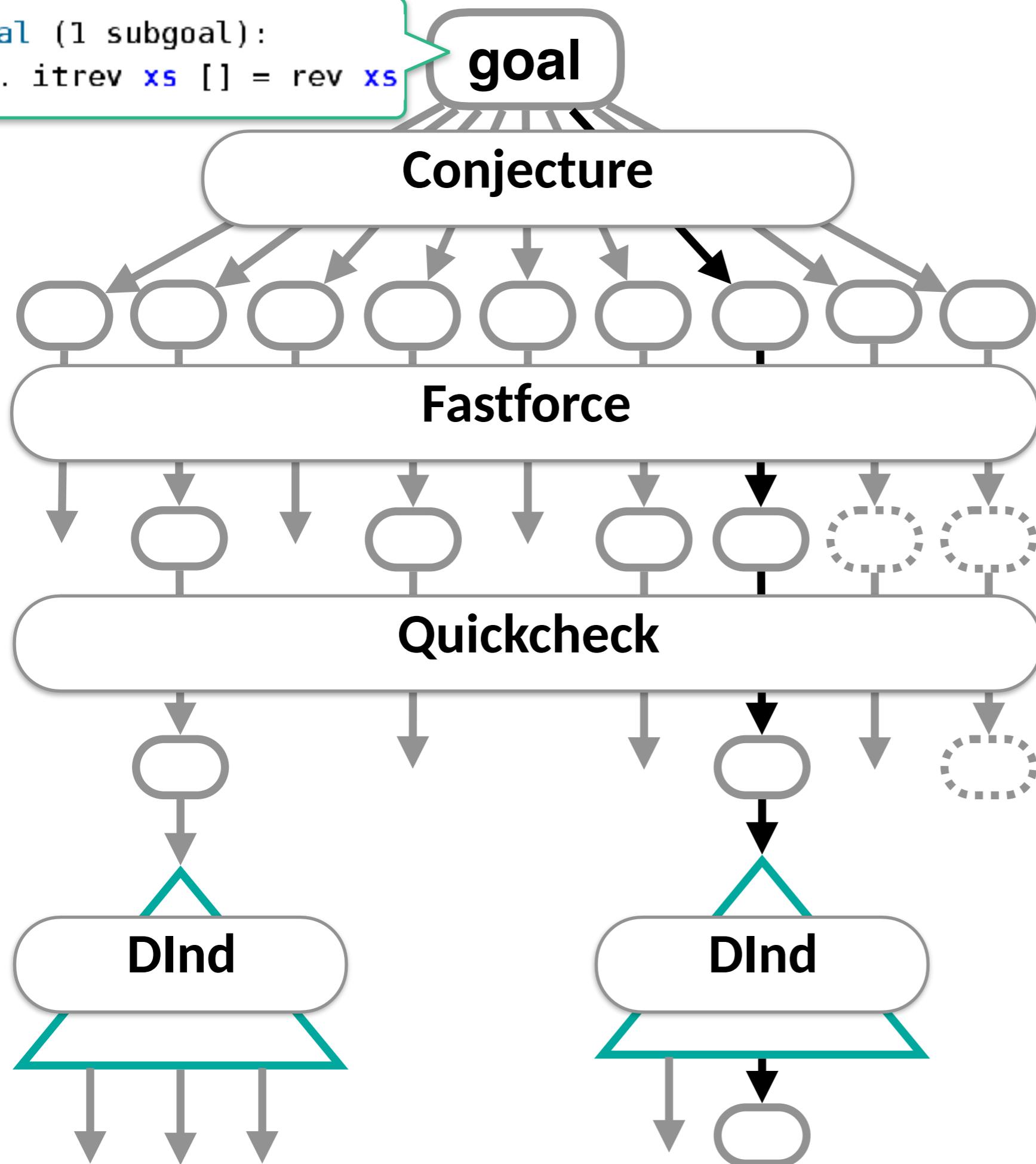


DInd



DEMO2

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



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

goal

apply (subgoal_tac

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

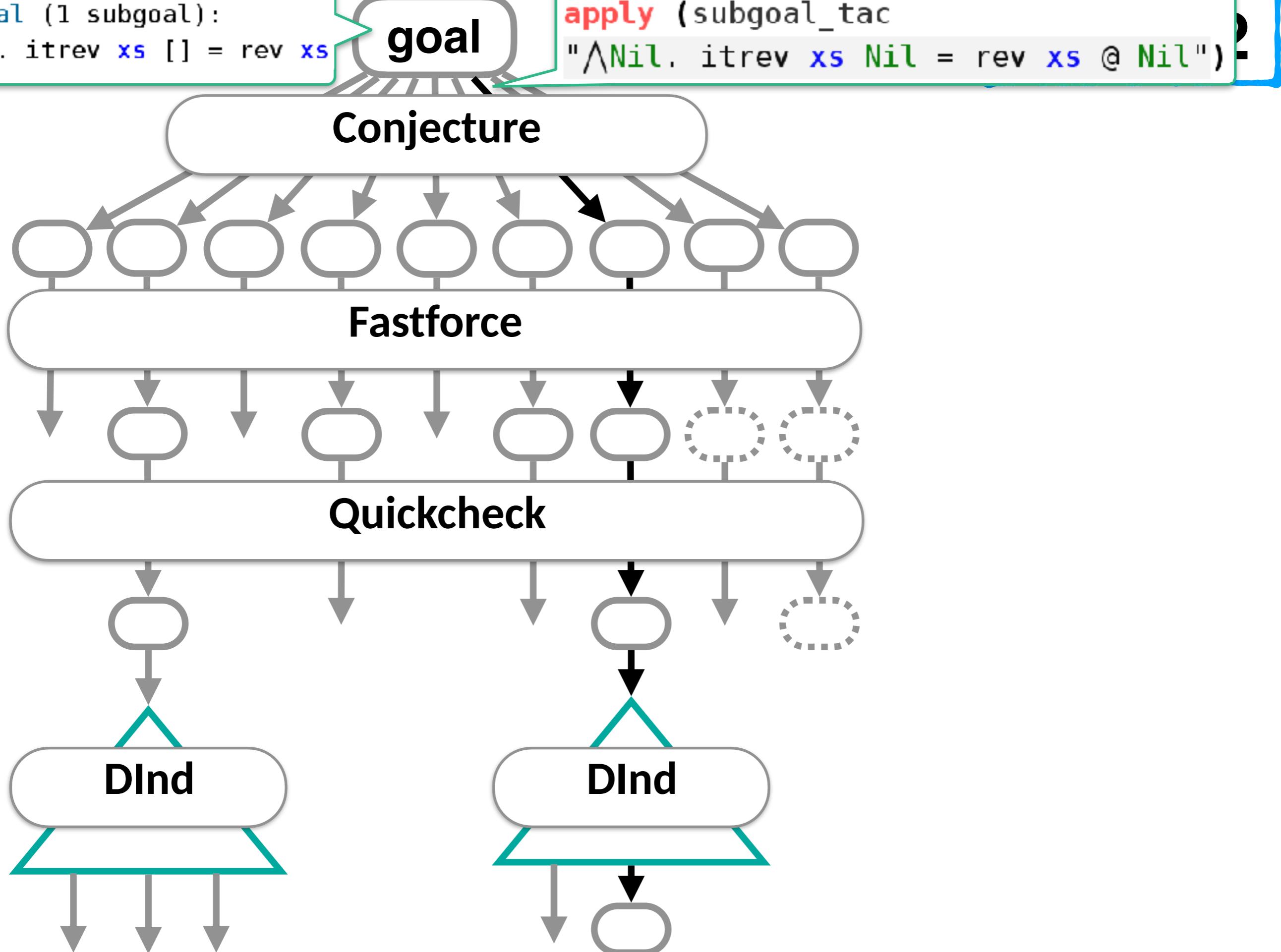
Conjecture

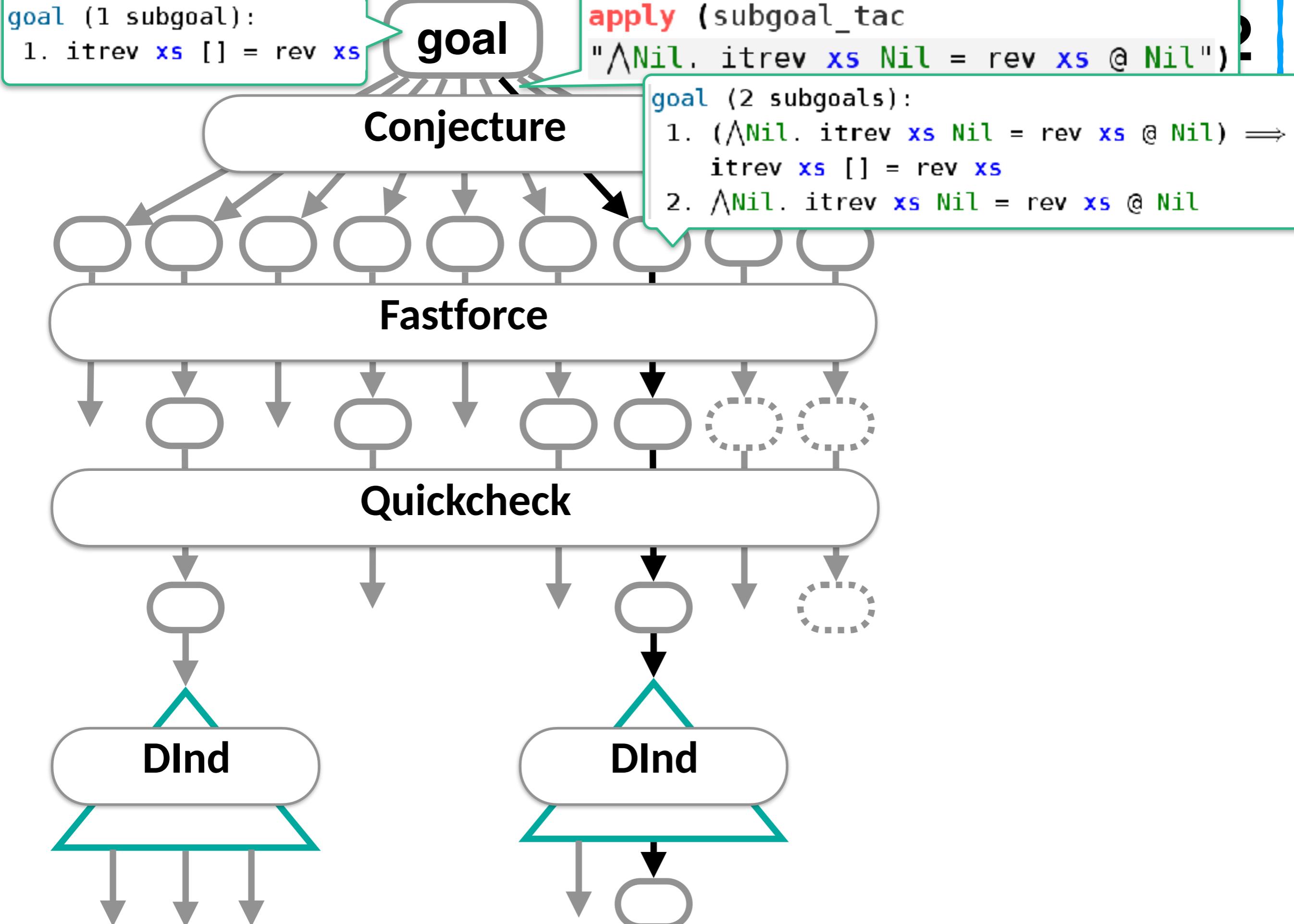
Fastforce

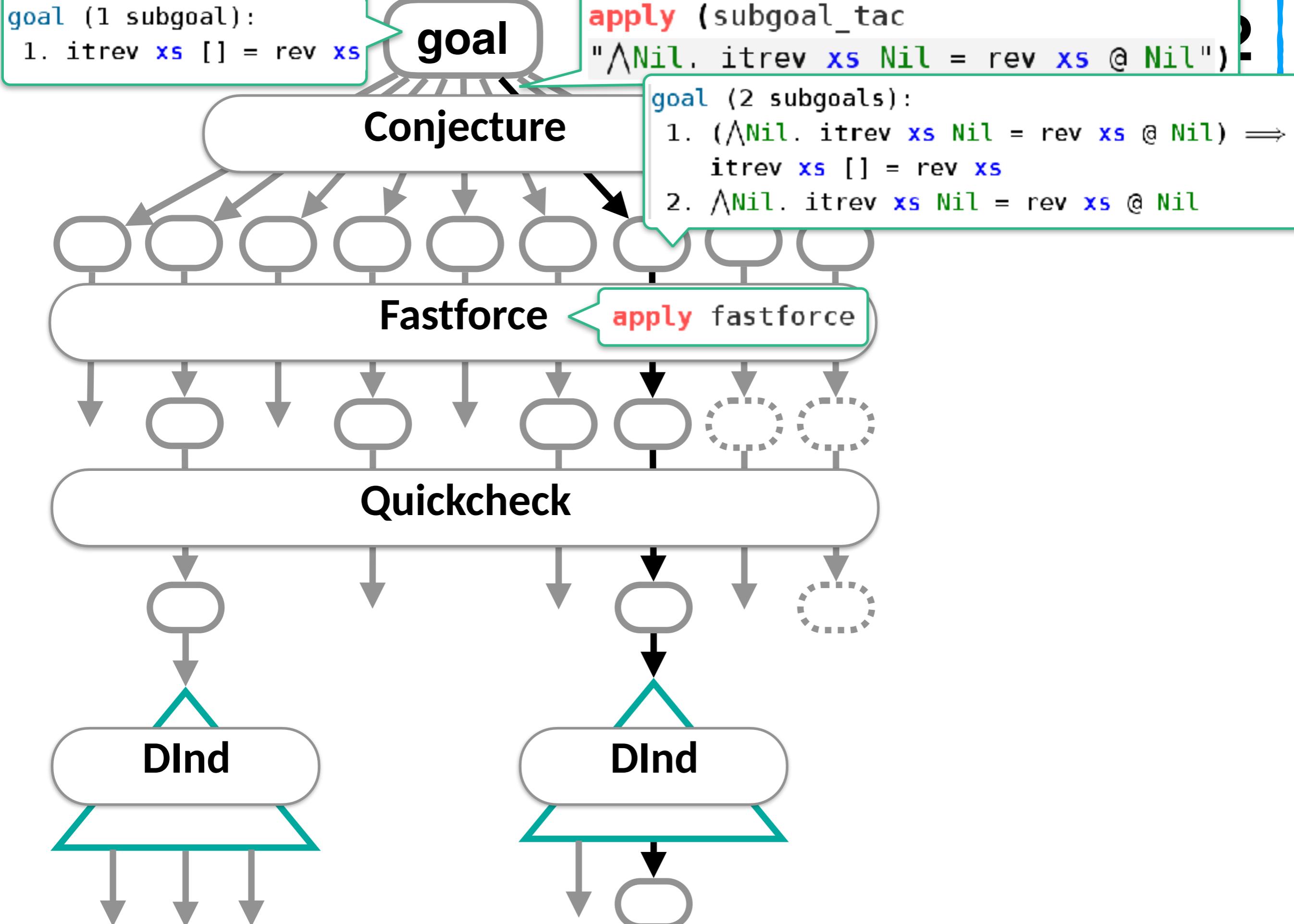
Quickcheck

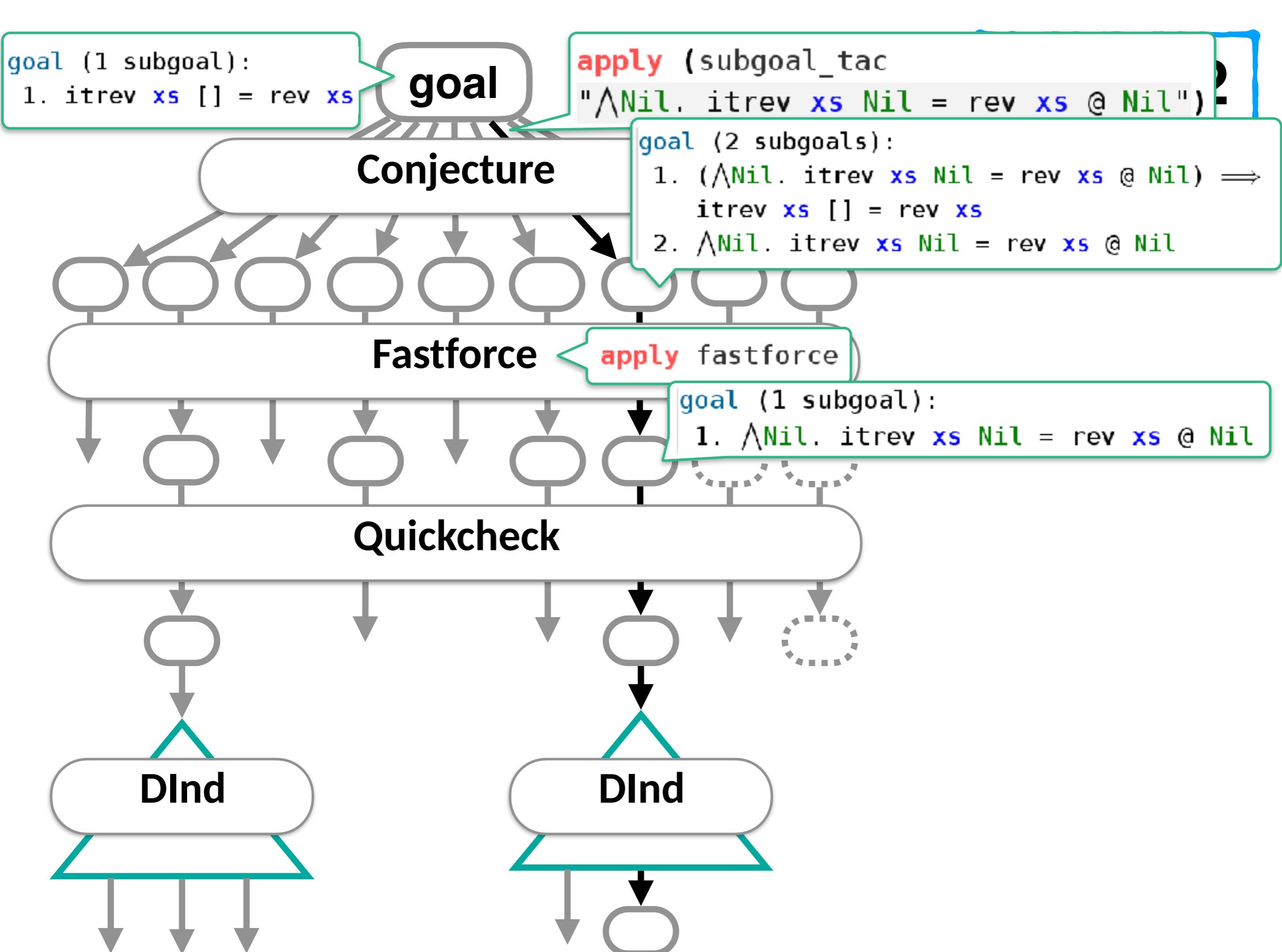
DInd

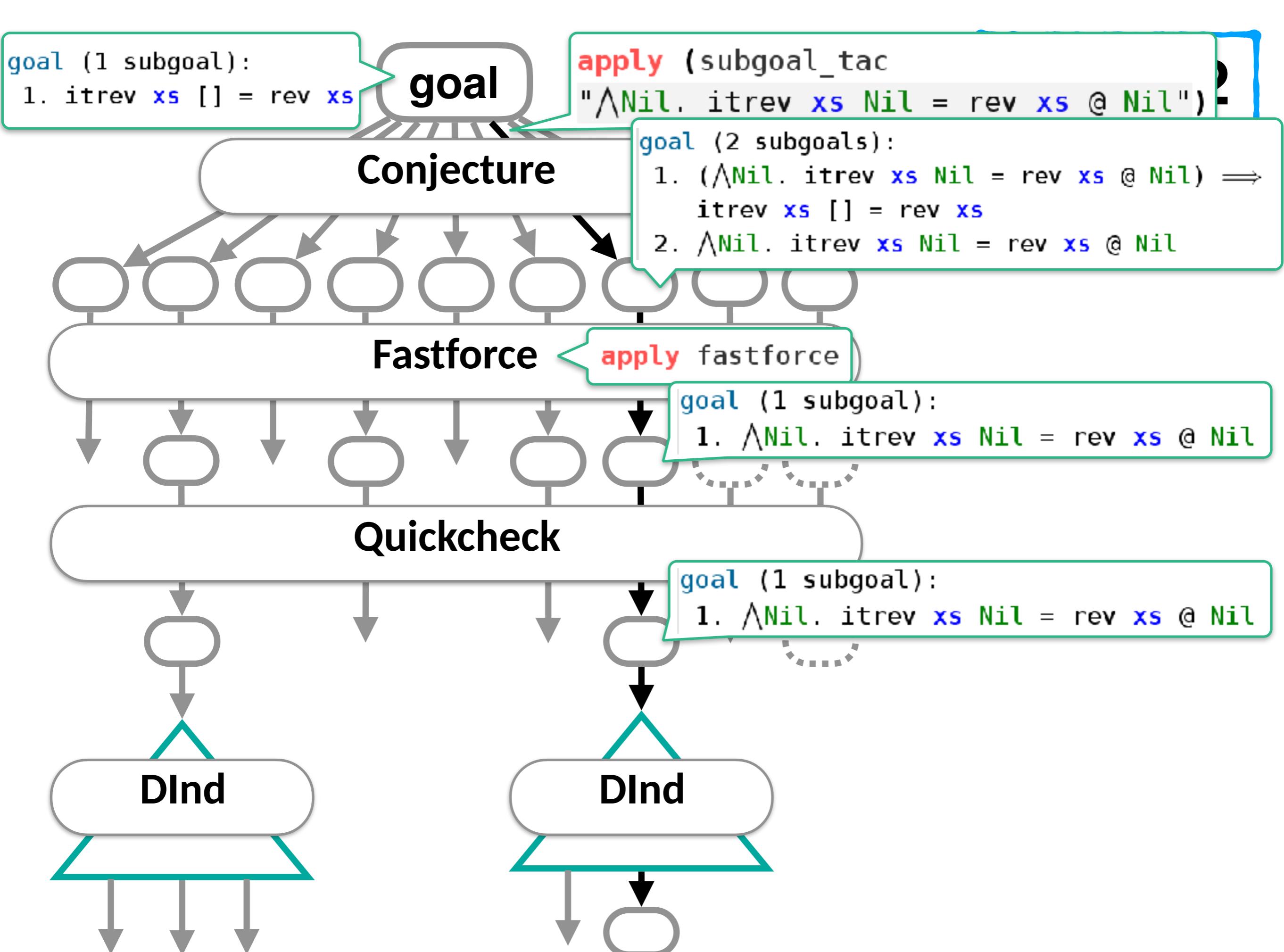
DInd

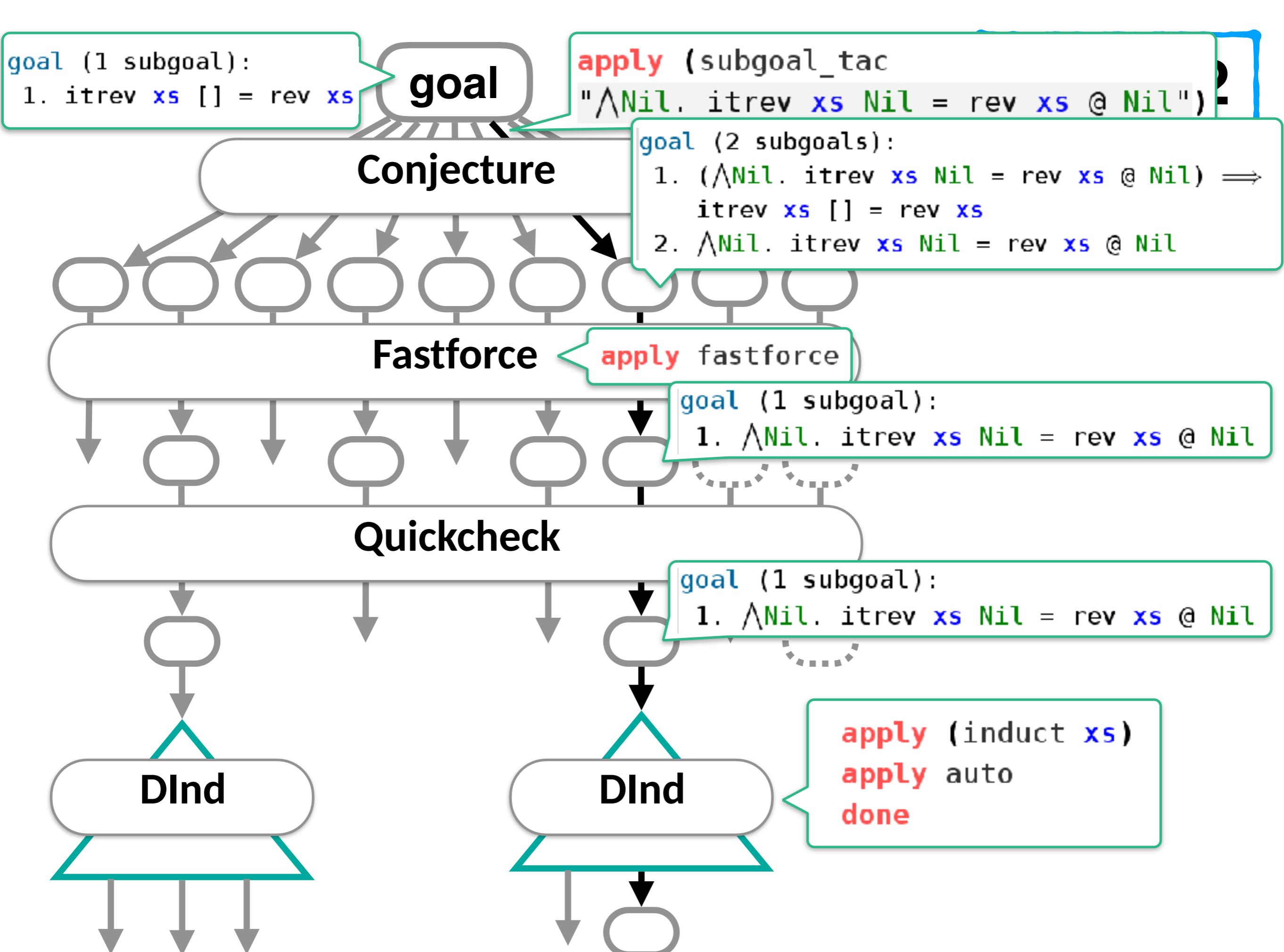


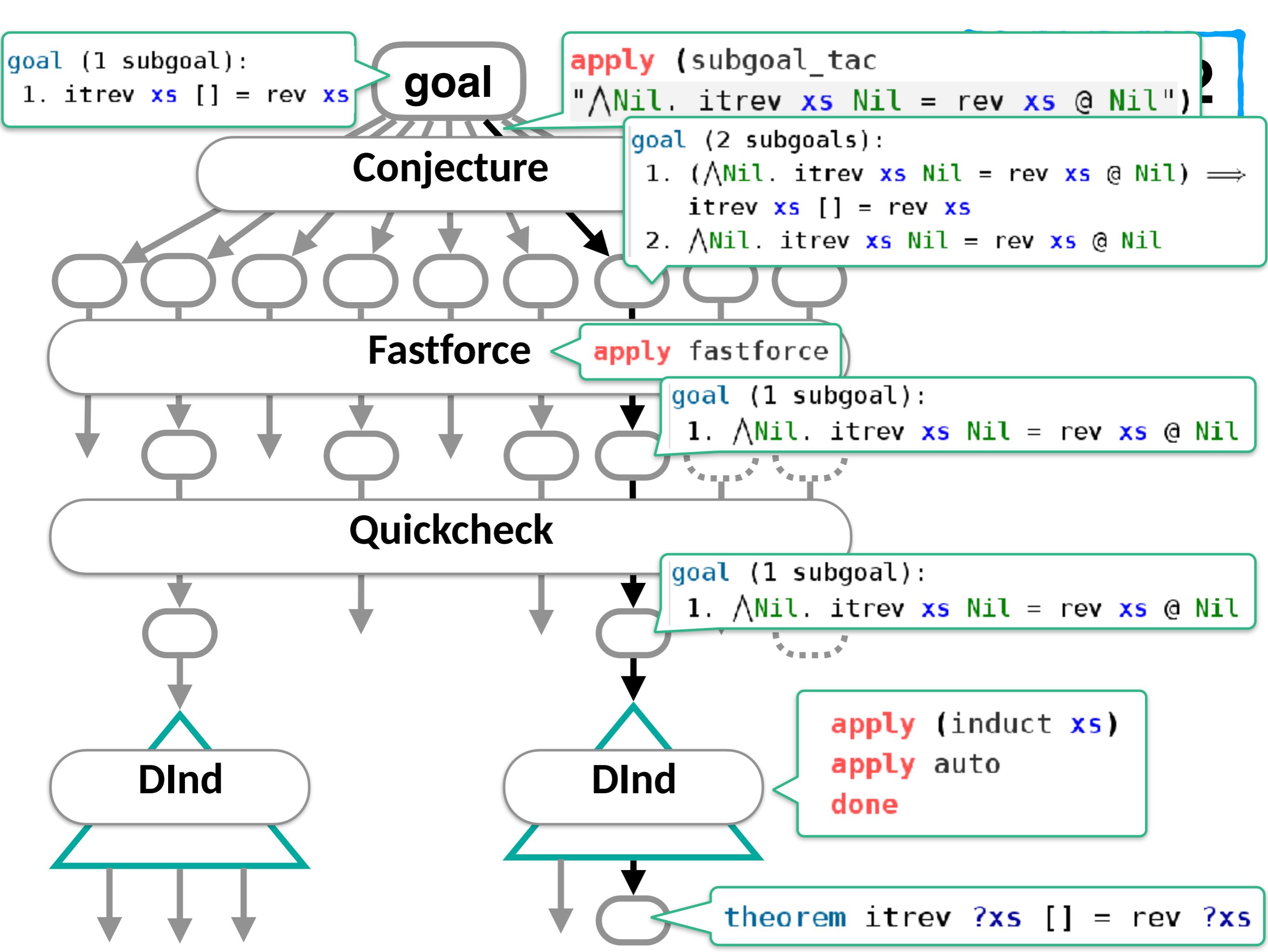


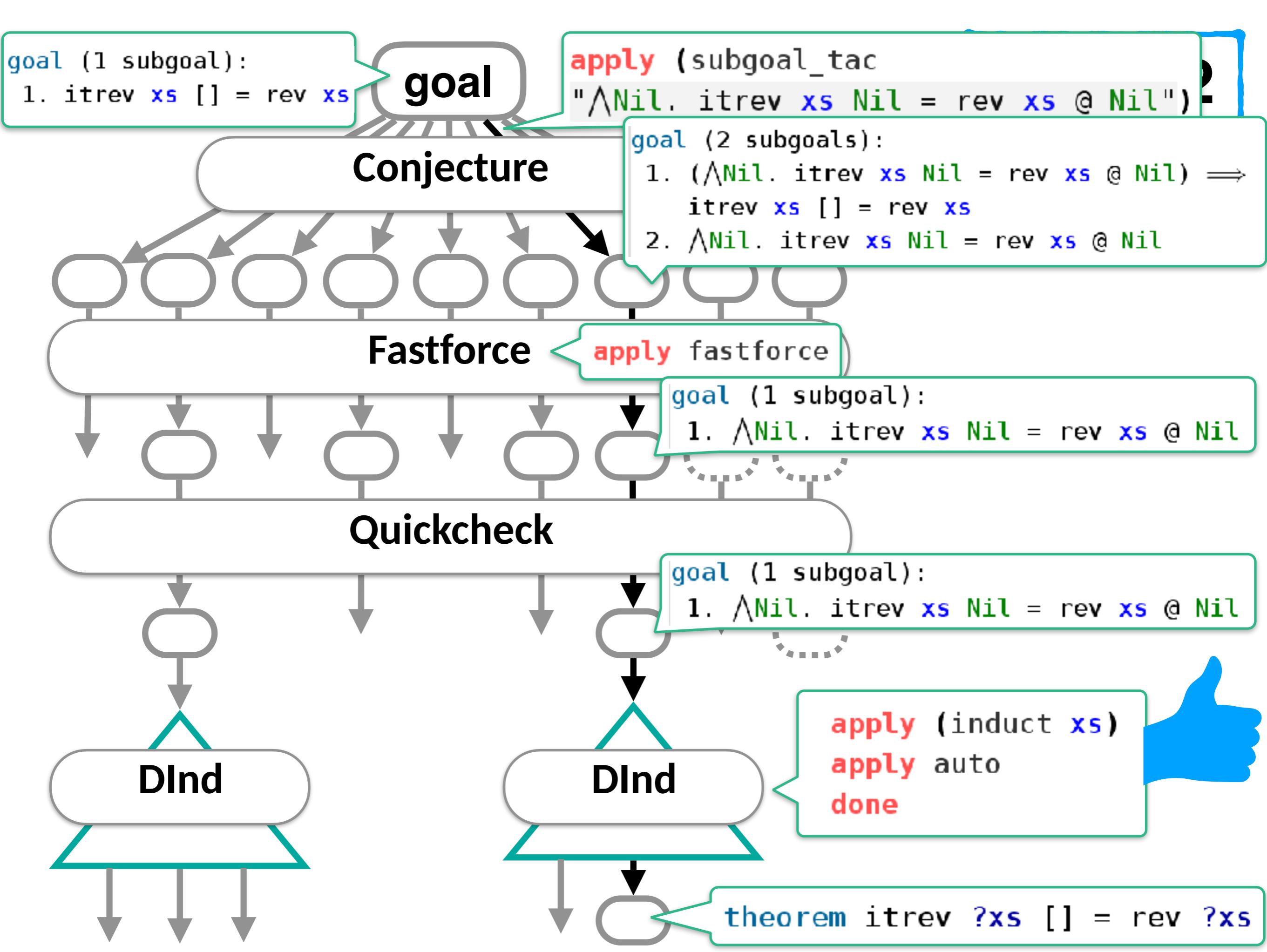






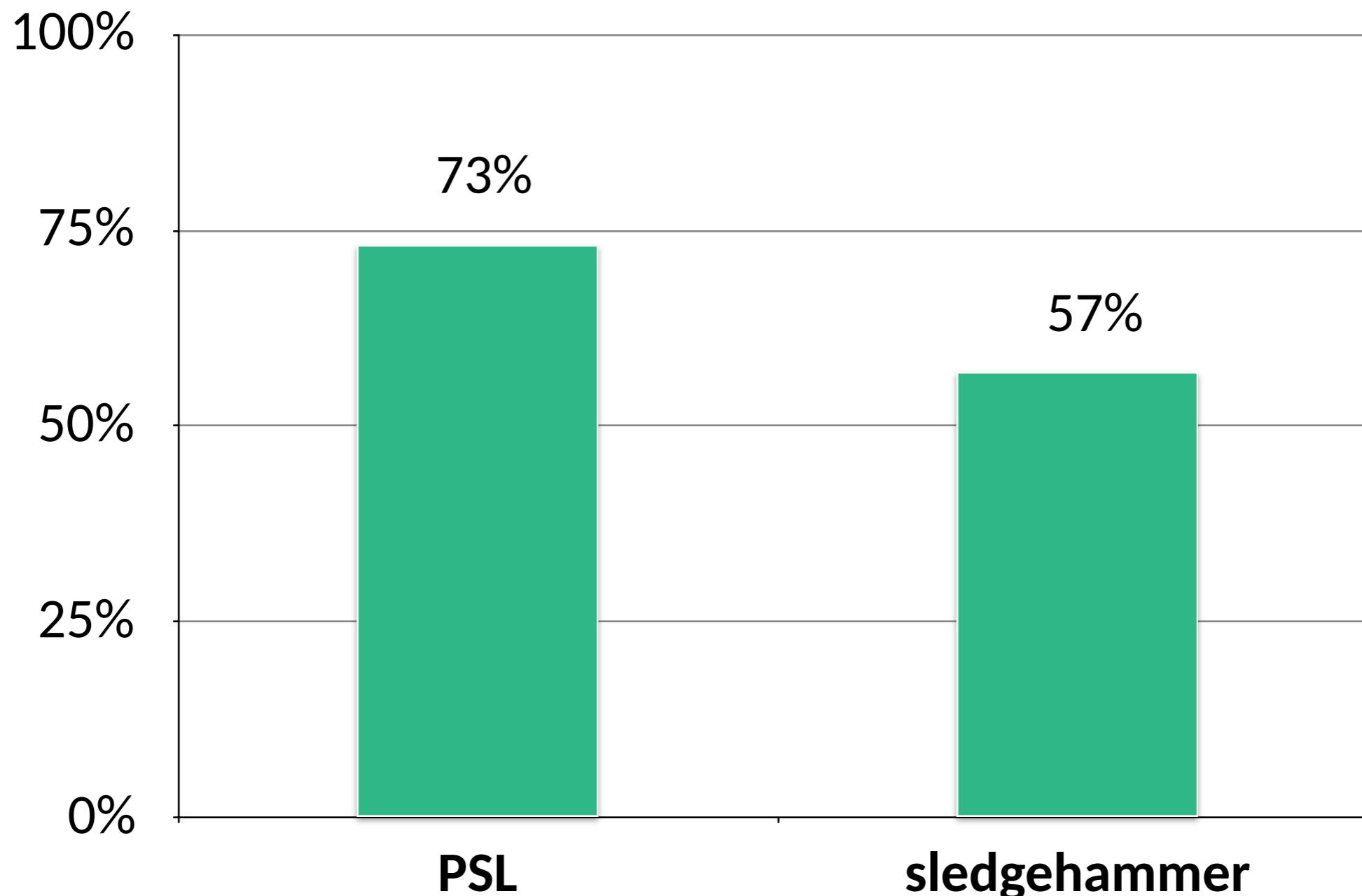






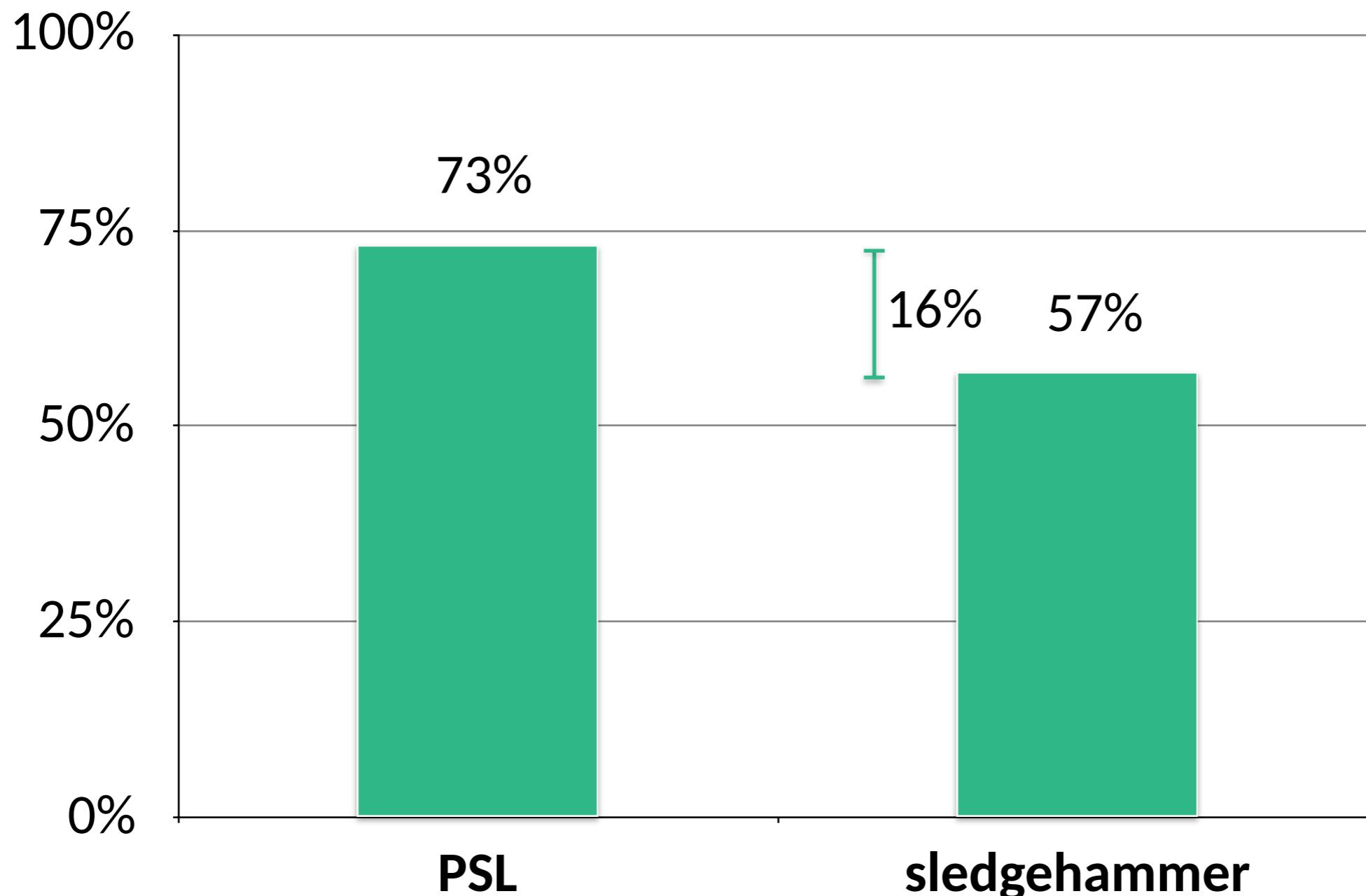
PSL vs sledgehammer

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



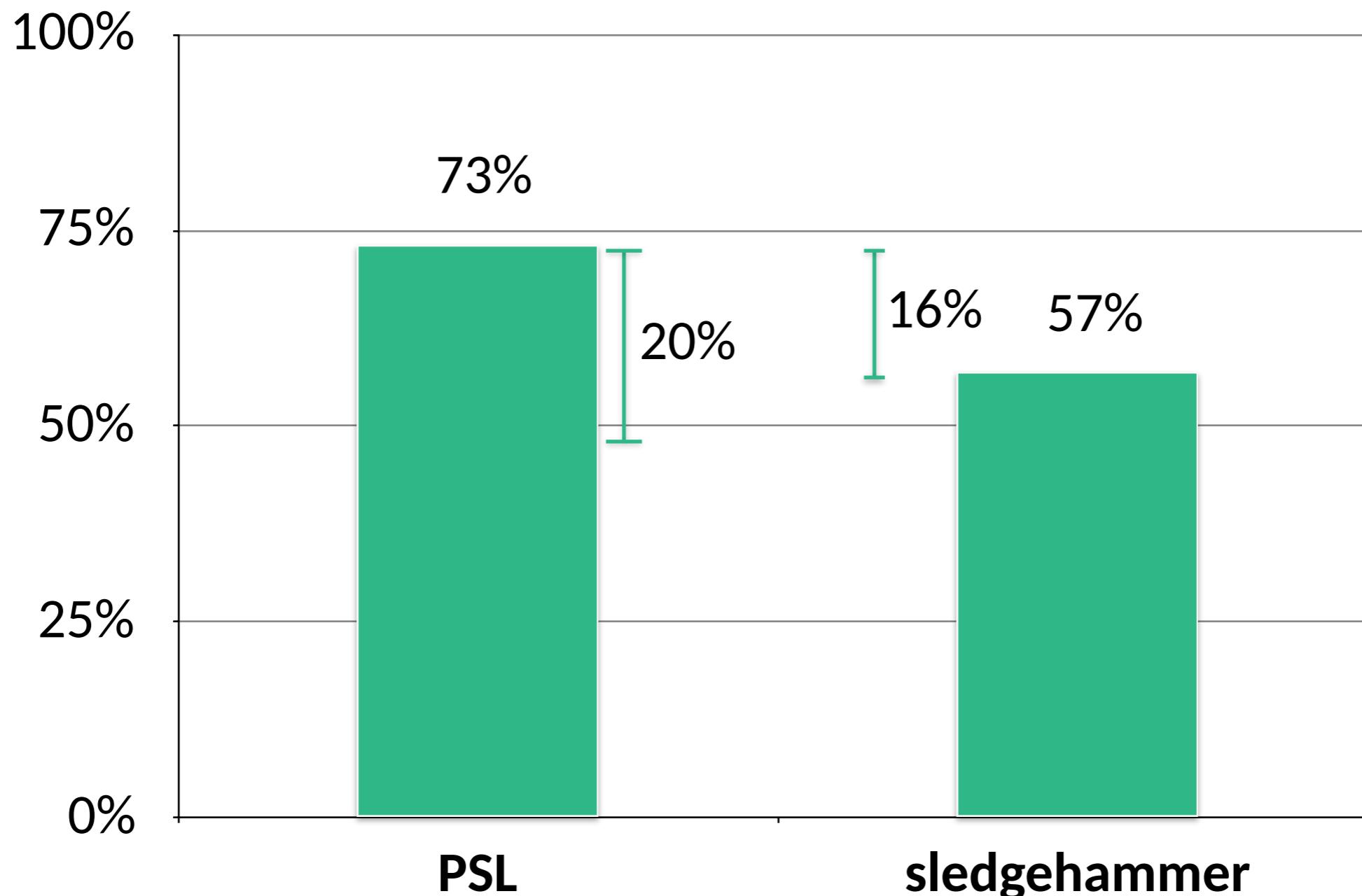
PSL vs sledgehammer

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

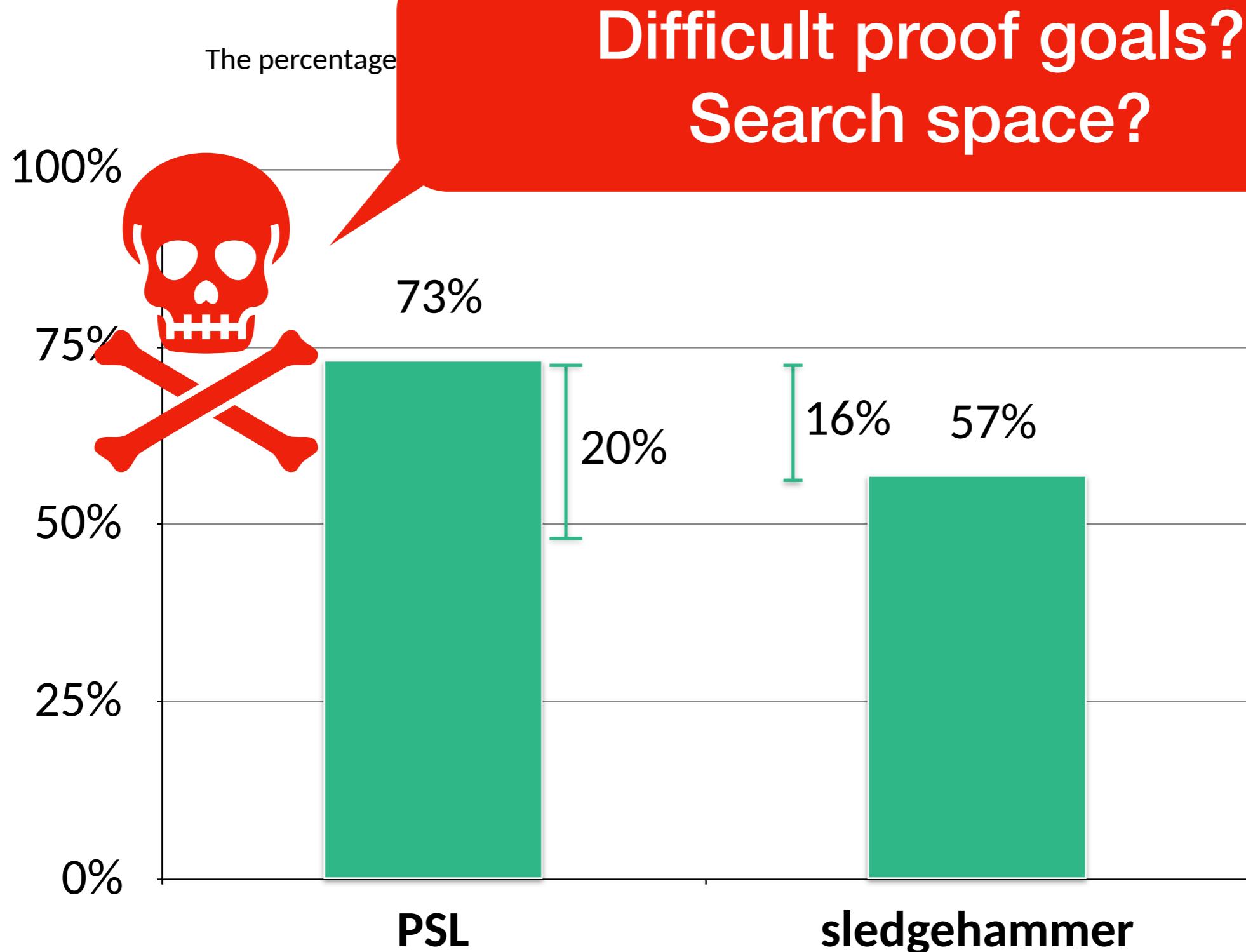


PSL vs sledgehammer

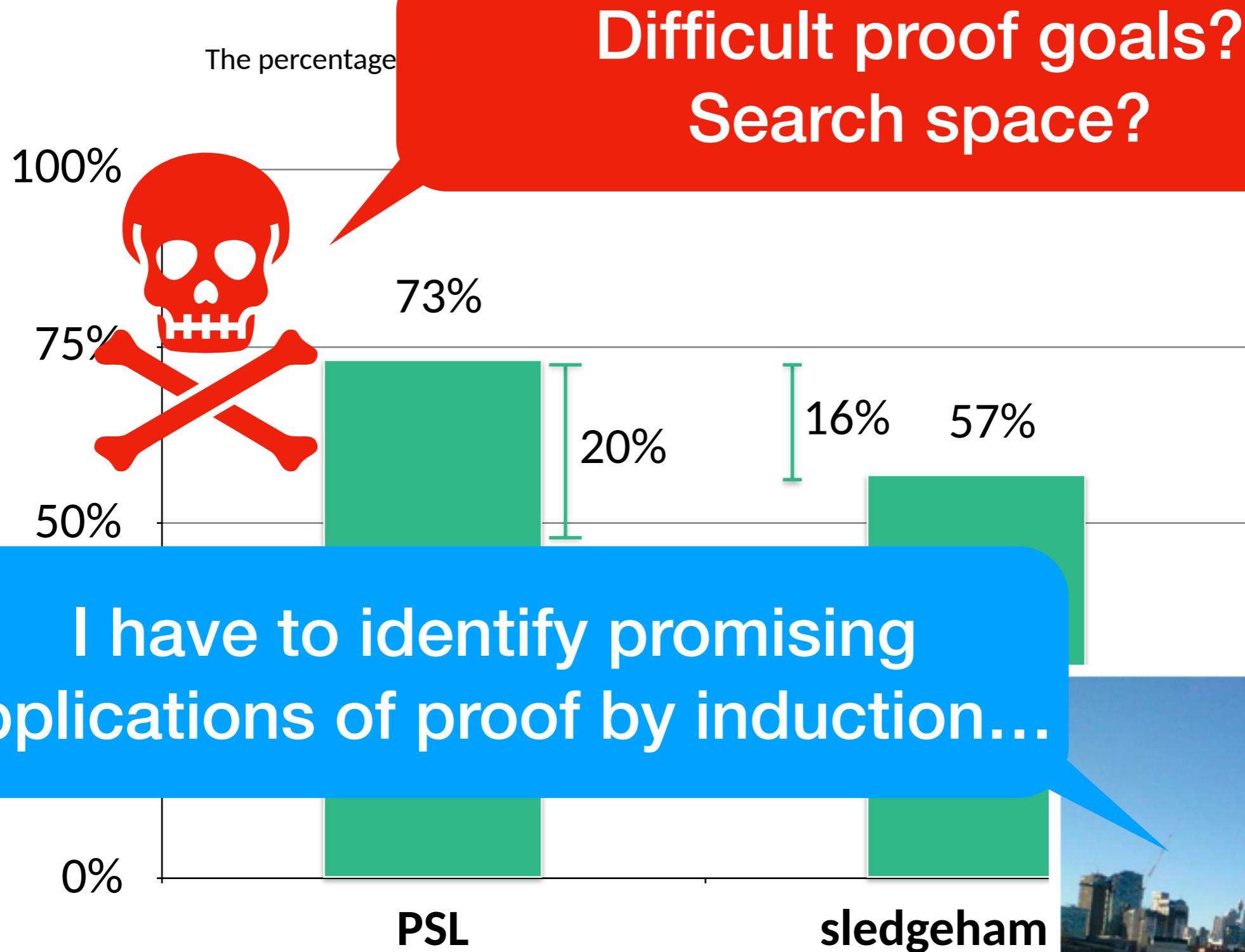
The percentage of automatically proved obligations out of 1526 proof obligations
(timeout = 300s)



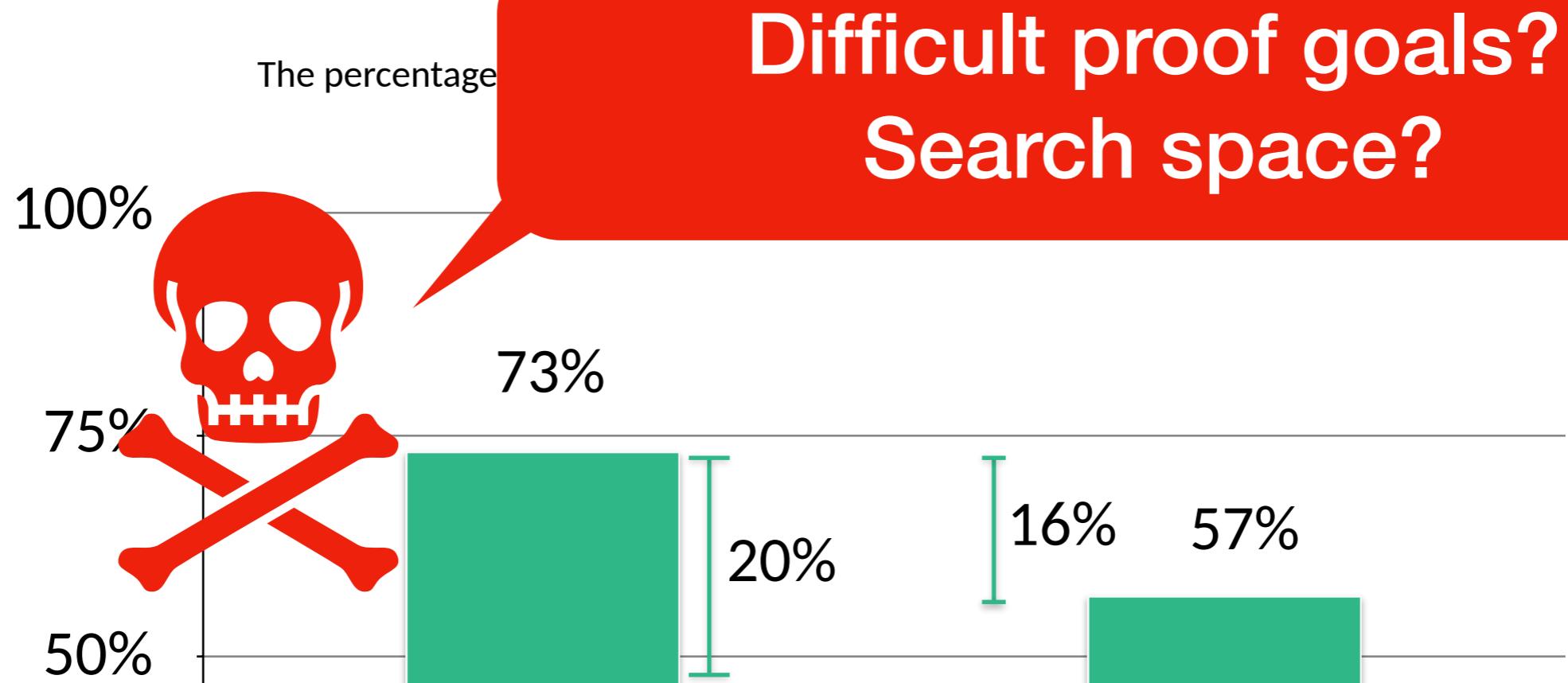
PSL vs sledgehammer



PSL vs sledgehammer



PSL vs sleddaehammer



I have to identify promising applications of proof by induction...

without completing a proof search



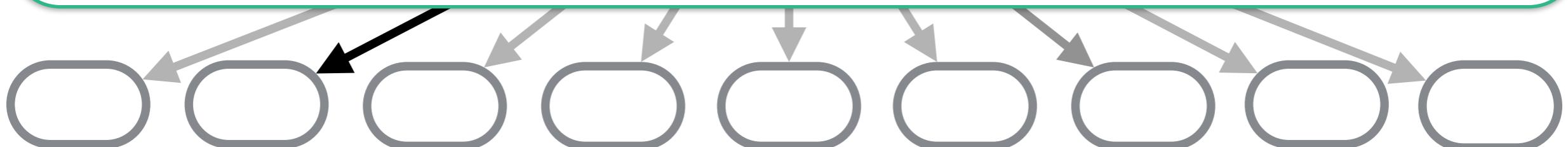
`smart_induct`

`goal`

smart_induct

goal

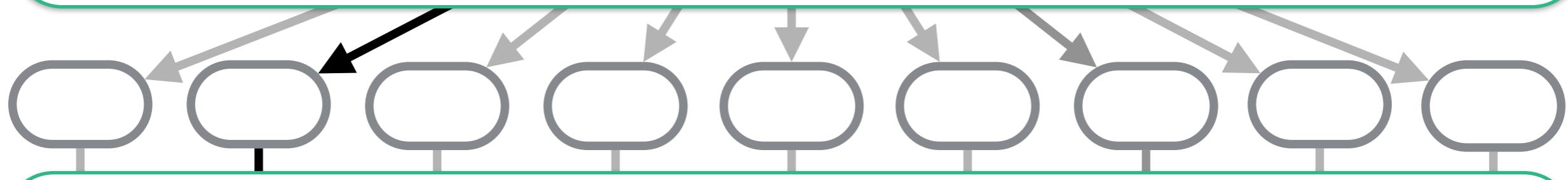
Step 1: creating many inductions



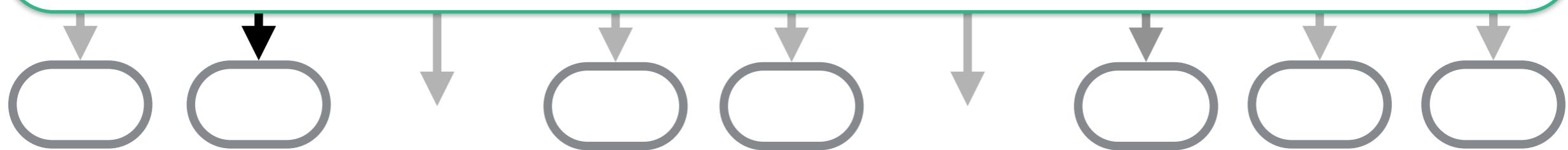
smart_induct

goal

Step 1: creating many inductions



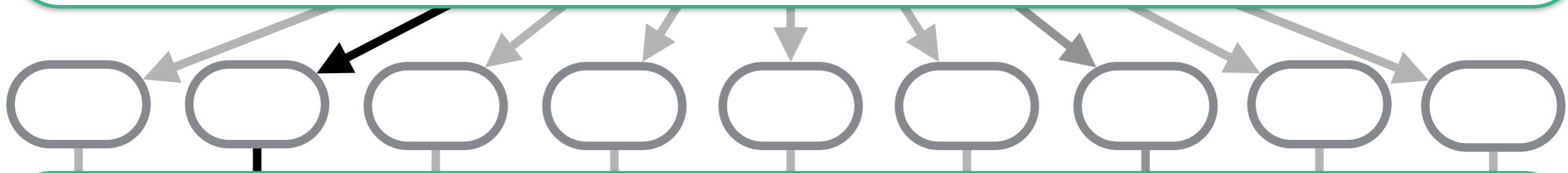
Step 2: multi-stage screening



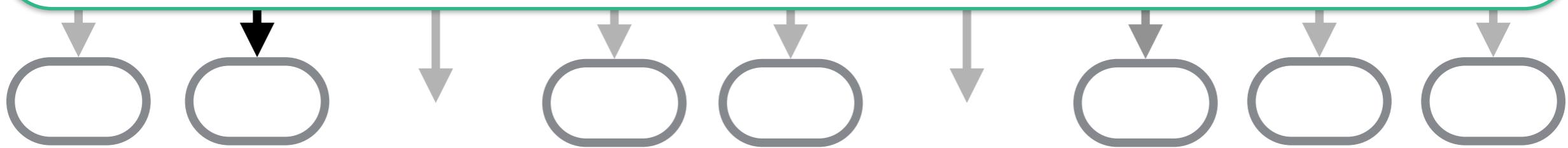
smart_induct

goal

Step 1: creating many inductions



Step 2: multi-stage screening

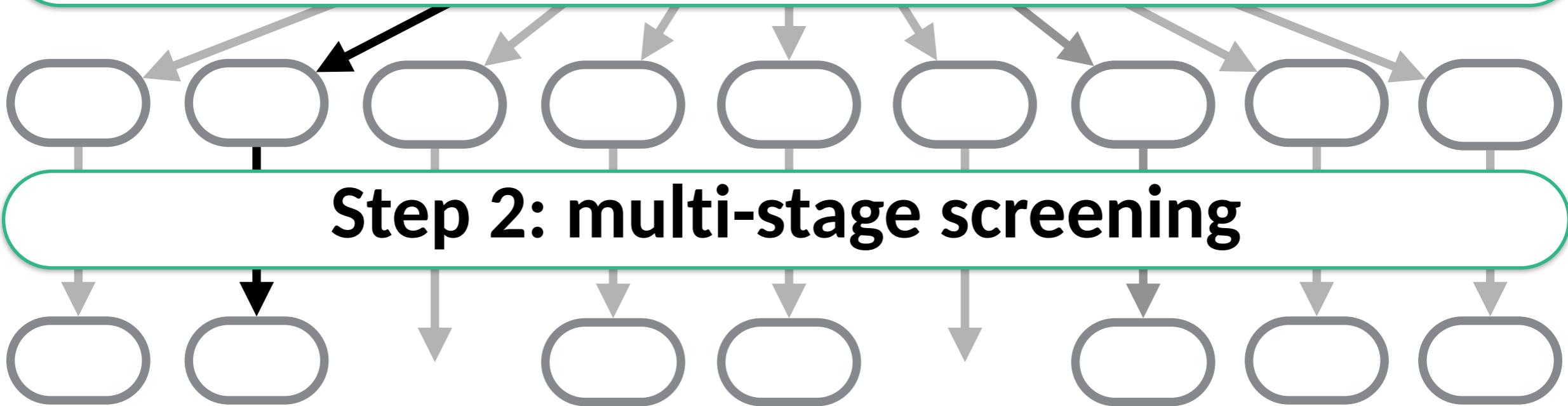


Step 3: scoring using 20 heuristics and sorting

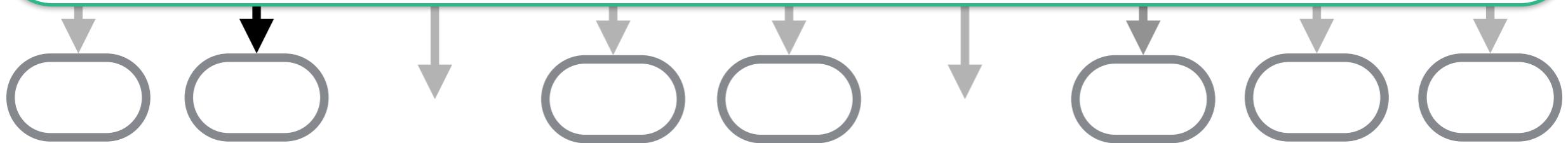
smart_induct

goal

Step 1: creating many inductions



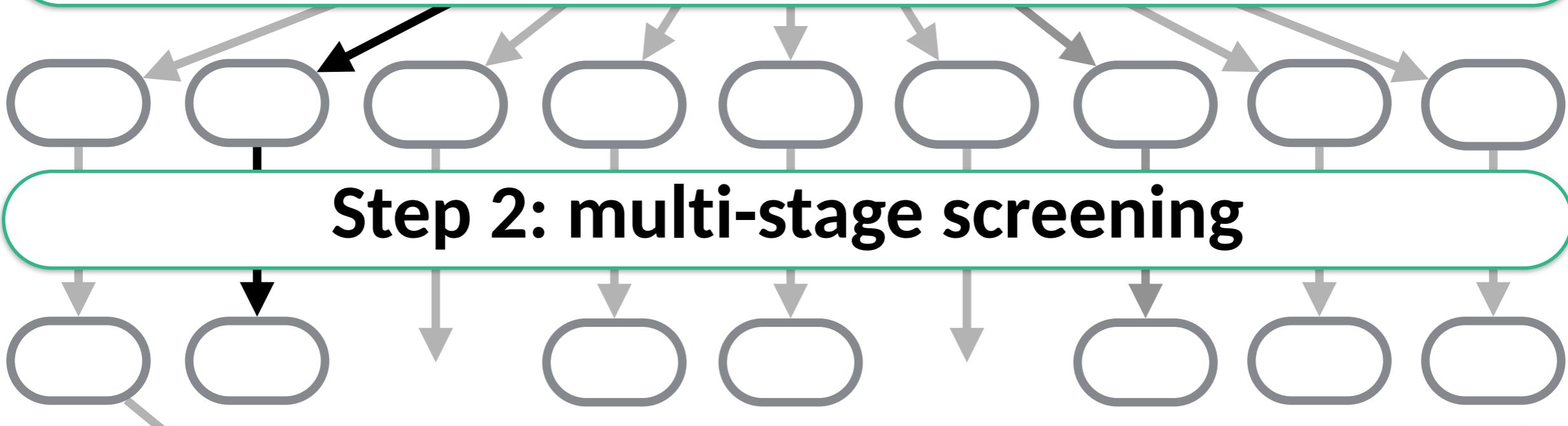
Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting

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

Step 1: creating many inductions



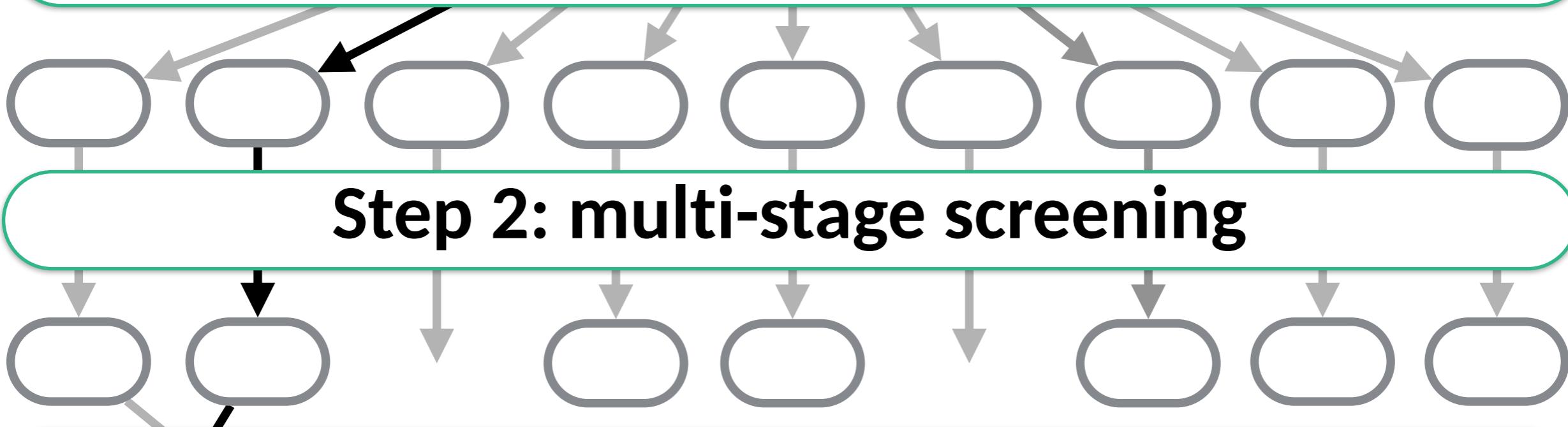
Step 3: scoring using 20 heuristics and sorting

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

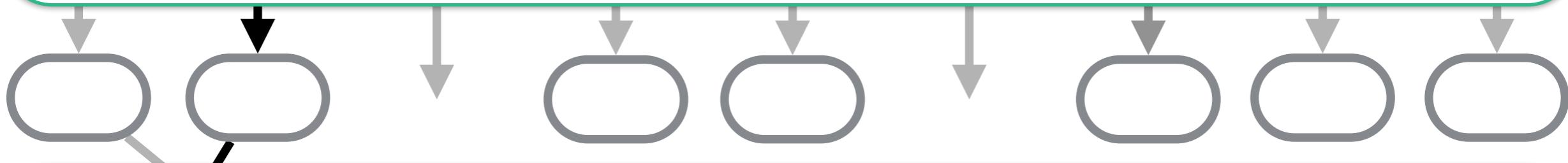
smart_induct

goal

Step 1: creating many inductions



Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting

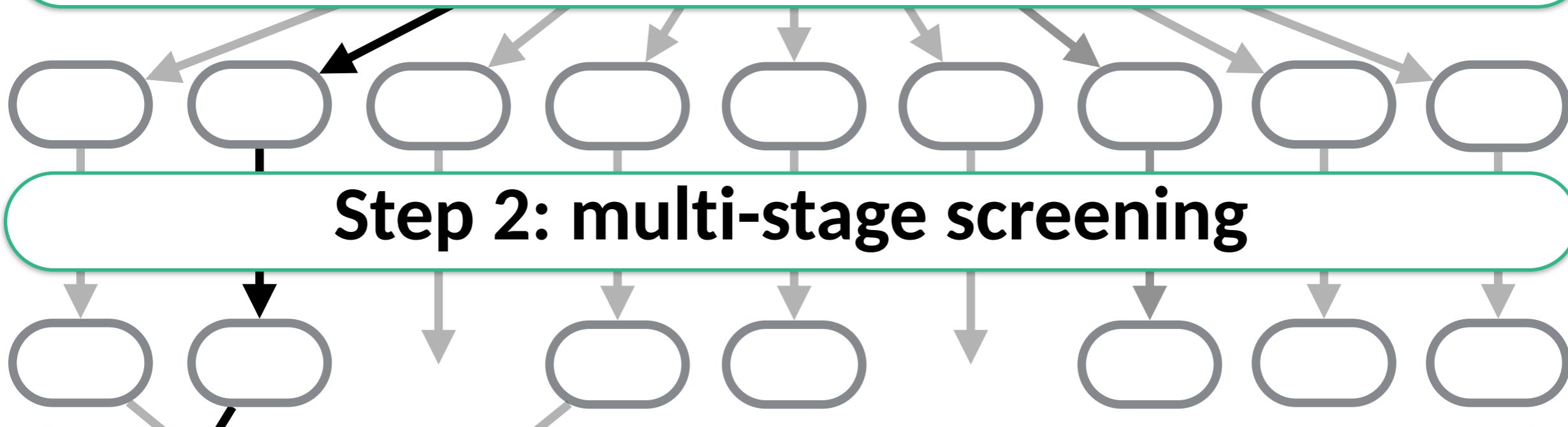


heuristic : (proof goal * induction arguments) -> bool

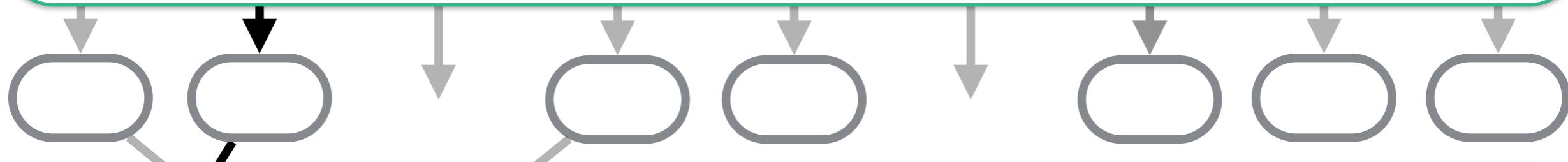
smart_induct

goal

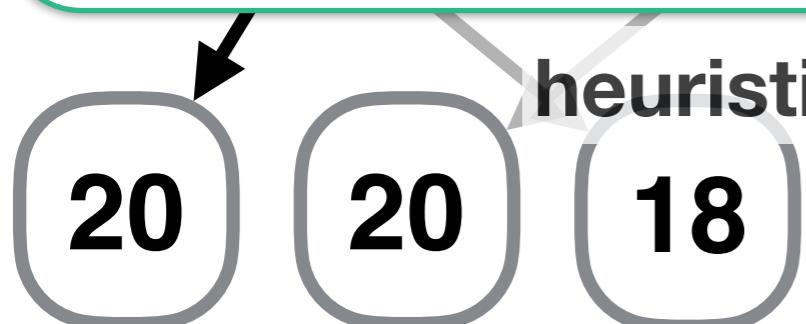
Step 1: creating many inductions



Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting



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

20

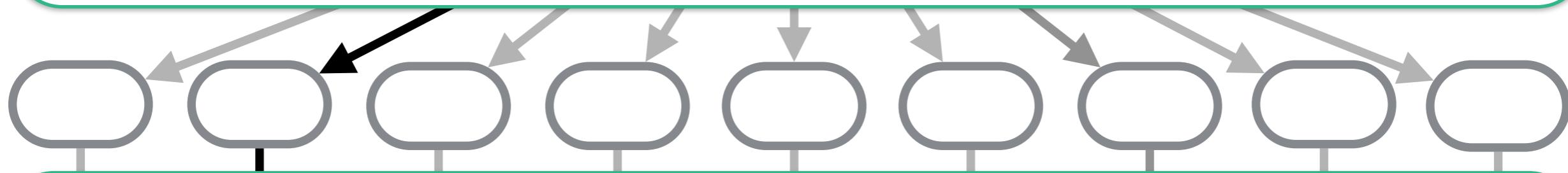
20

18

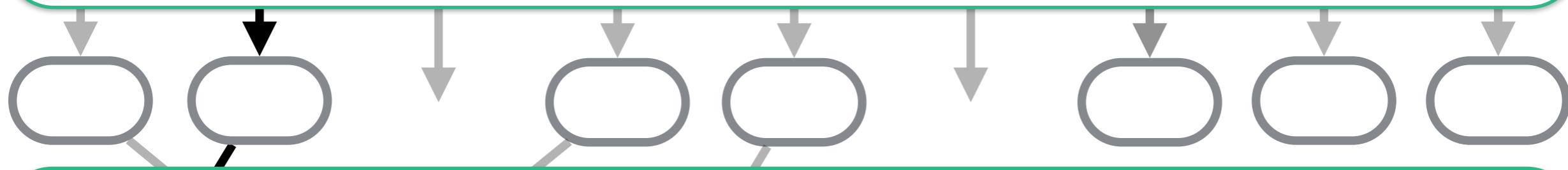
smart_induct

goal

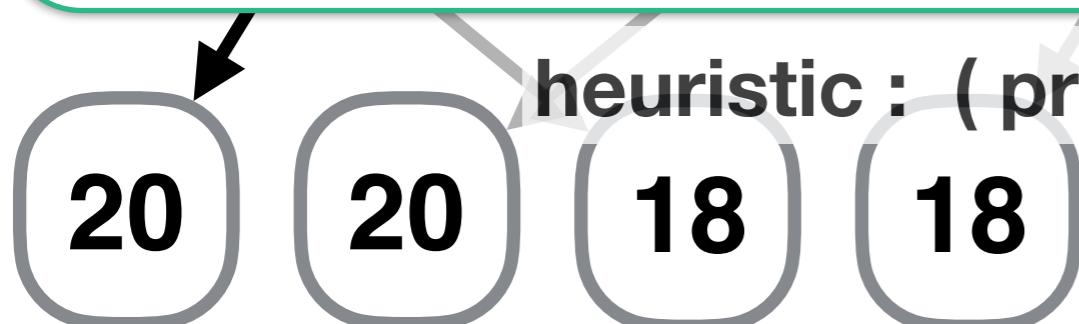
Step 1: creating many inductions



Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting



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

20

20

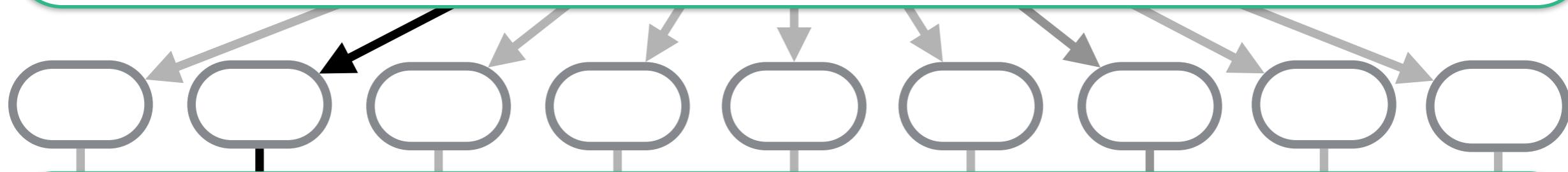
18

18

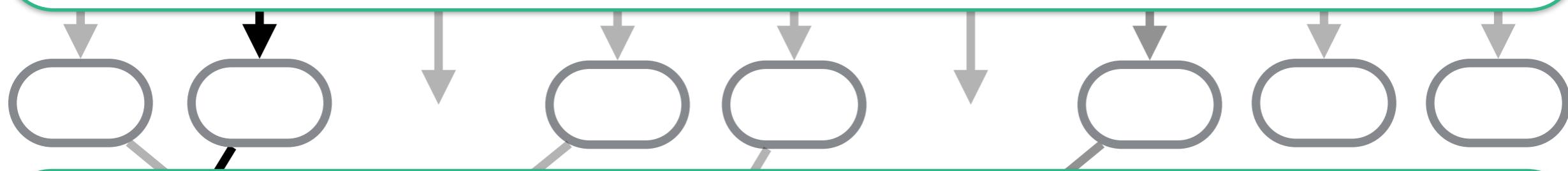
smart_induct

goal

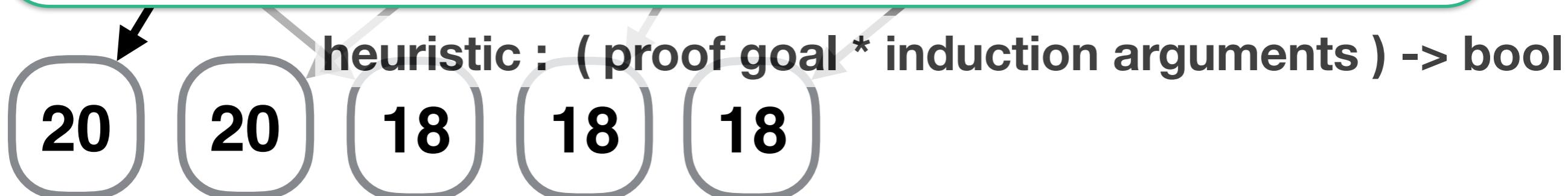
Step 1: creating many inductions



Step 2: multi-stage screening



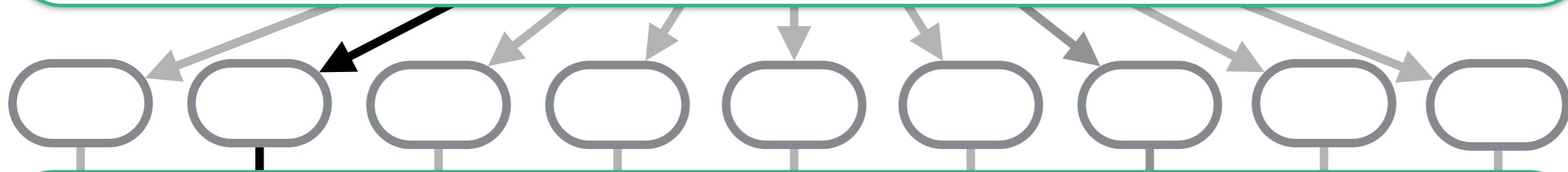
Step 3: scoring using 20 heuristics and sorting



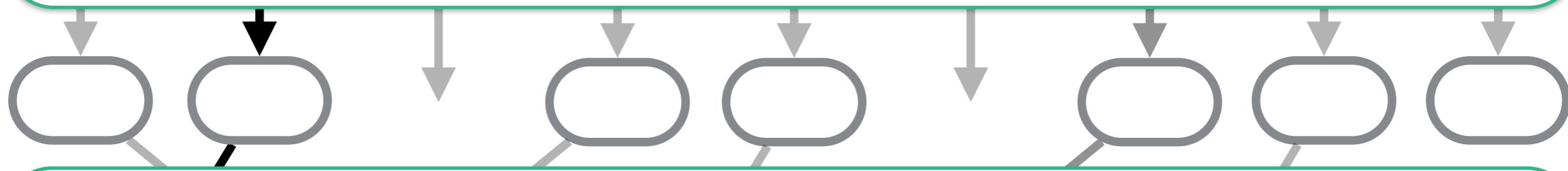
smart_induct

goal

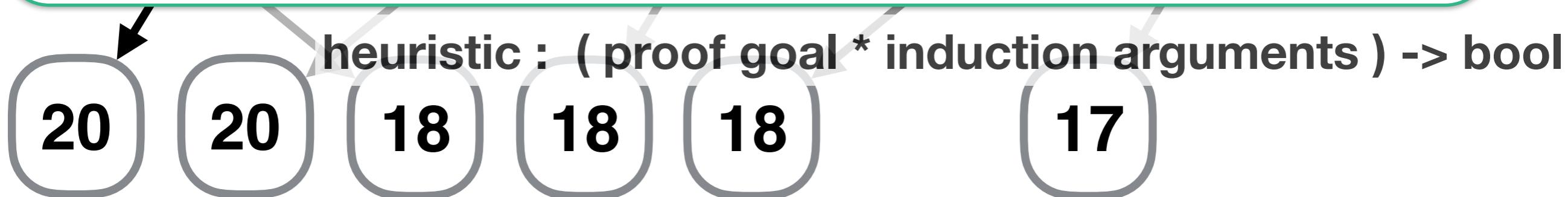
Step 1: creating many inductions



Step 2: multi-stage screening



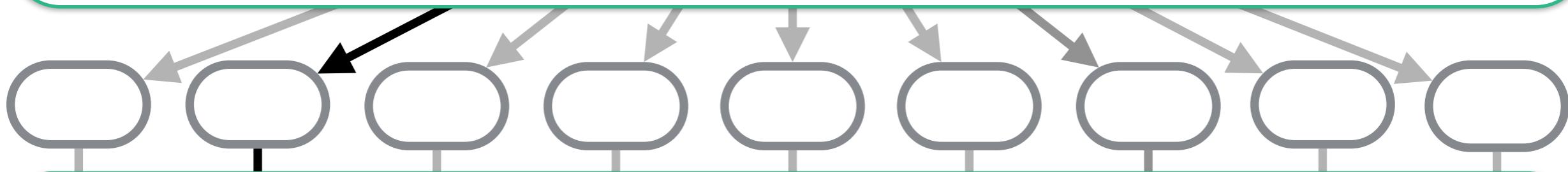
Step 3: scoring using 20 heuristics and sorting



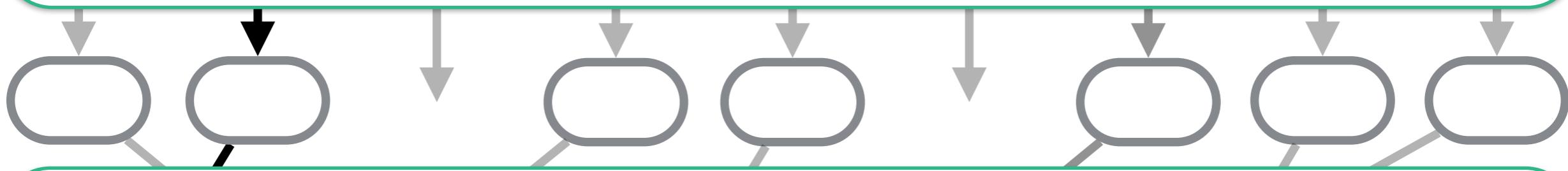
smart_induct

goal

Step 1: creating many inductions



Step 2: multi-stage screening



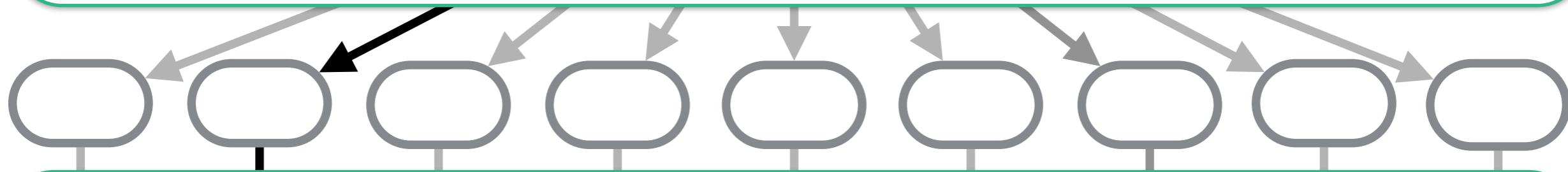
Step 3: scoring using 20 heuristics and sorting



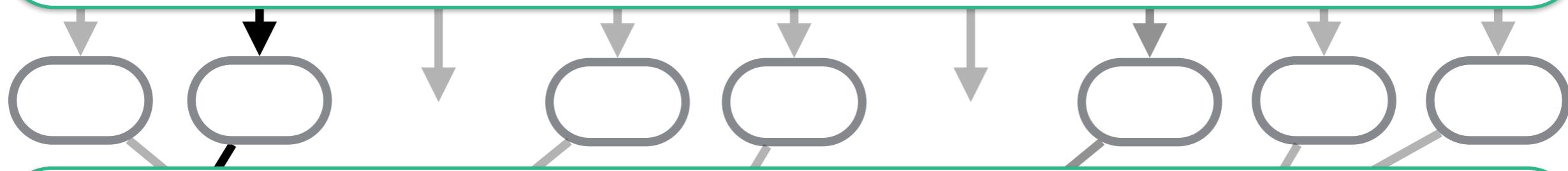
smart_induct

goal

Step 1: creating many inductions



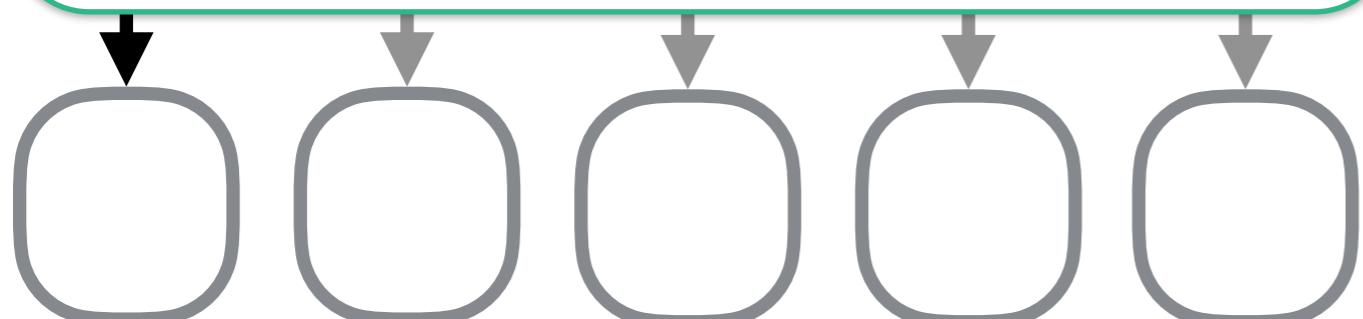
Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting



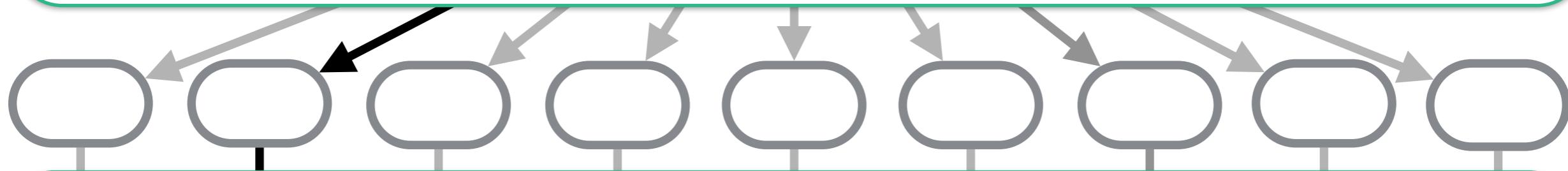
Step 4: short-listing



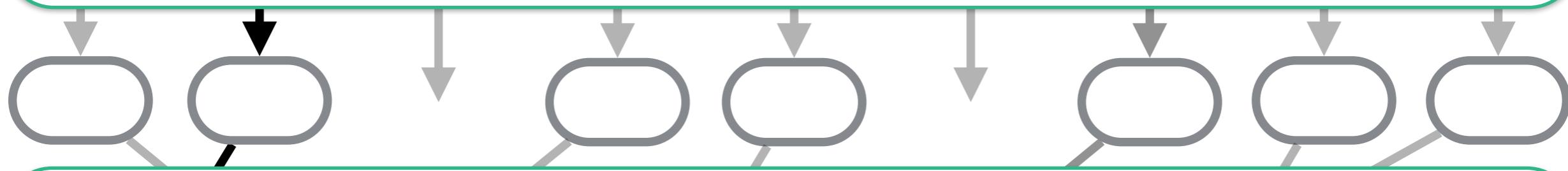
smart_induct

goal

Step 1: creating many inductions



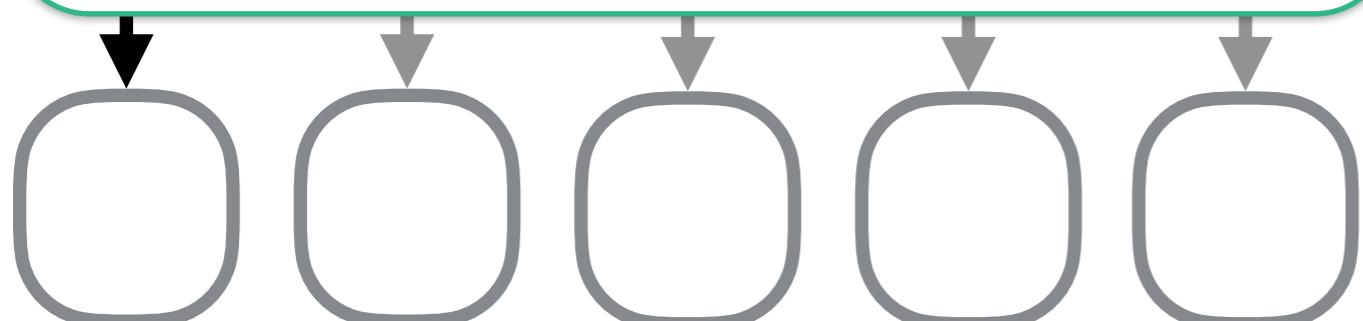
Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting



Step 4: short-listing



DEMO!

DEMO3

The screenshot shows the Isabelle proof assistant 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 with a red rectangle. The bottom part 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.

DEMO3

```
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. *)

DEMO3

```
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. *)

DEMO3

```
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. *)

DEMO3

The screenshot shows the Isabelle proof assistant interface. The top part displays a theory file named "Induction_Demo.thy". The code defines a function "itrev" and a lemma. The lemma is annotated with "smart_induct". The bottom part of the interface shows the output window where the "smart_induct" command has been executed, providing statistics about the generated induction arguments.

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

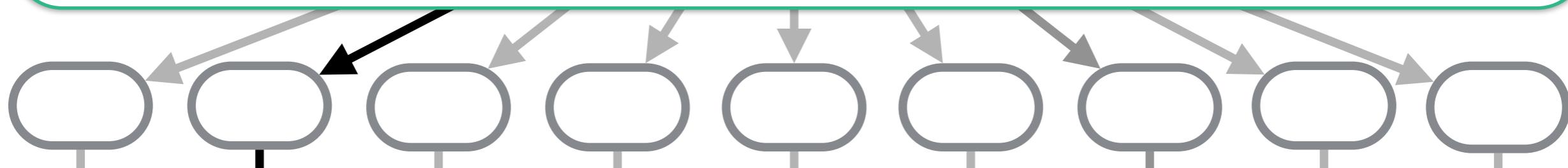
lemma "itrev xs ys = rev xs @ ys" smart_induct
apply(induction xs arbitrary: ys)
apply(auto)
```

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!

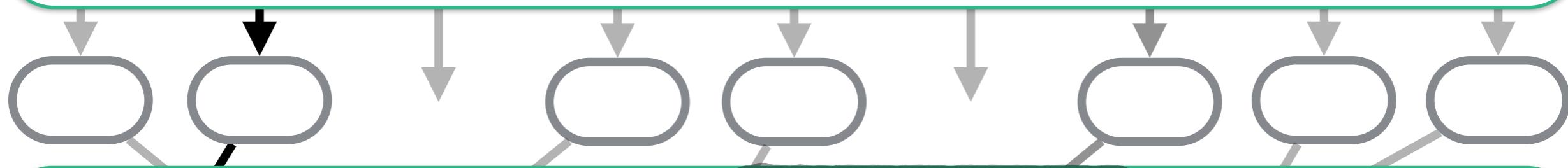
1st candidate is apply (induct xs arbitrary: ys)
(* The score is 20 out of 20. *)
2nd candidate is apply (induct xs)
(* The score is 20 out of 20. *)
3th candidate is apply (induct xs ys rule: Induction_Demo.itrev.induct)
(* The score is 18 out of 20. *)

goal

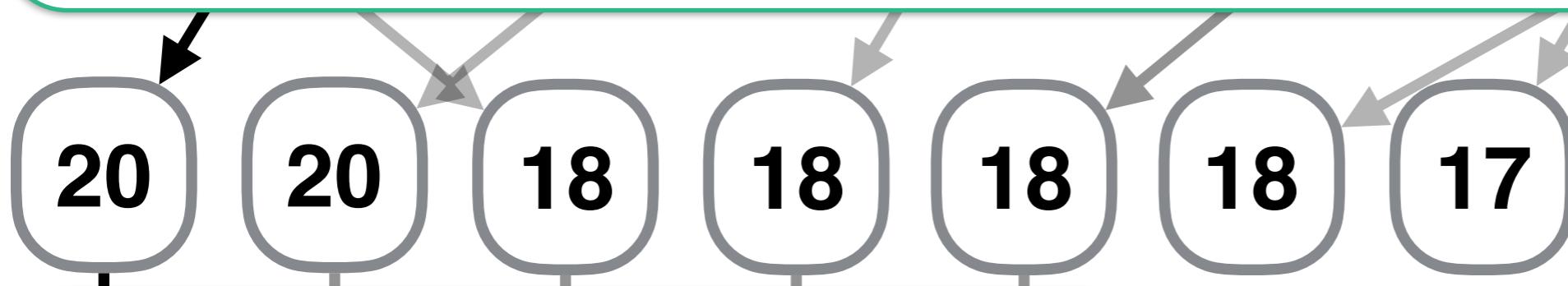
Step 1: creating many inductions



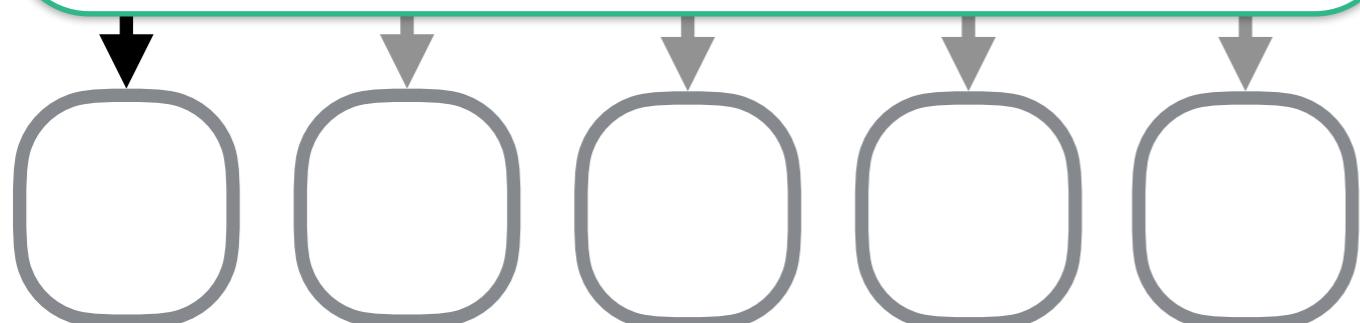
Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting

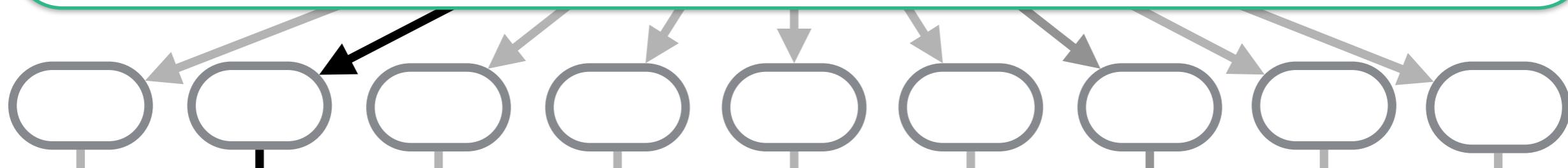


Step 4: short-listing

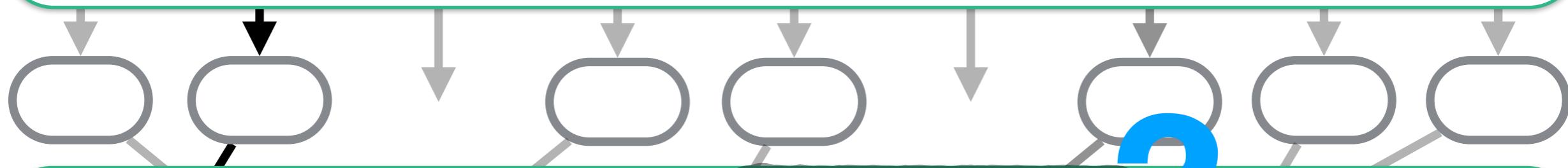


goal

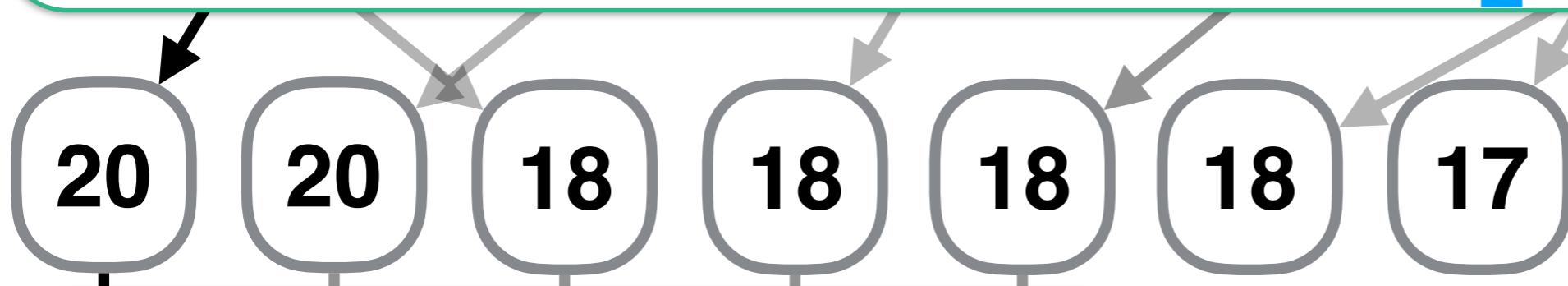
Step 1: creating many inductions



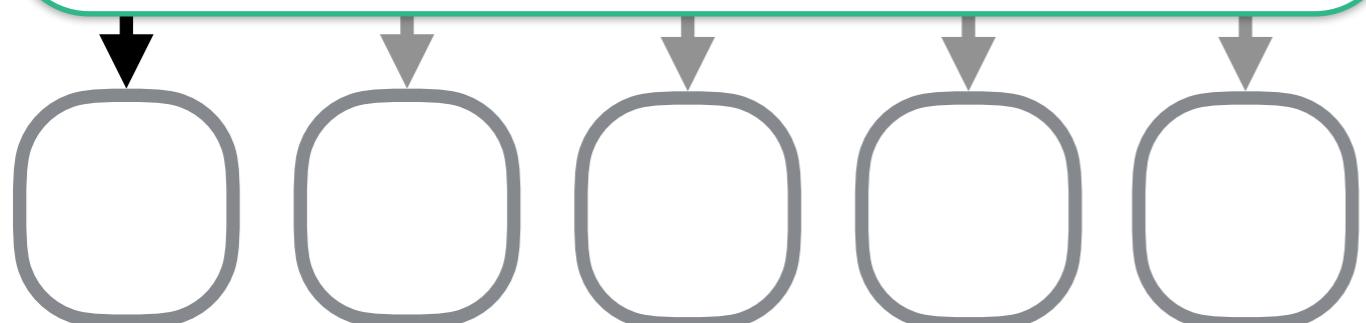
Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting

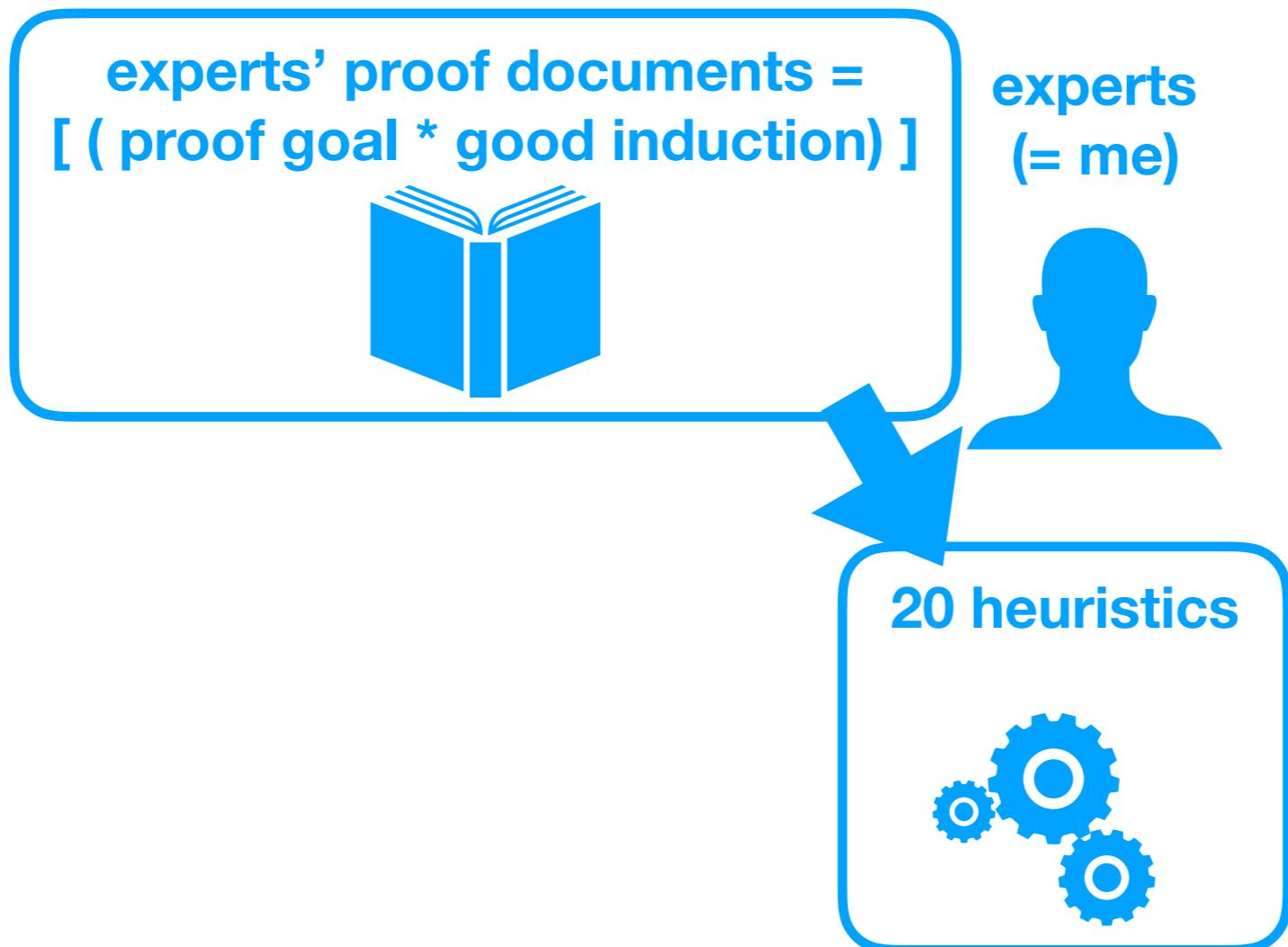


Step 4: short-listing



It is difficult to write induction heuristics.

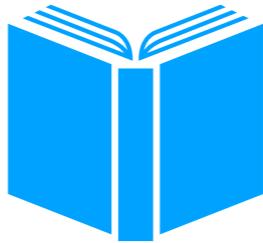
It is difficult to write induction heuristics.



It is difficult to write induction heuristics.

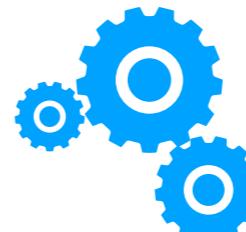
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

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



experts
(= me)

20 heuristics



It is difficult to write induction heuristics.

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

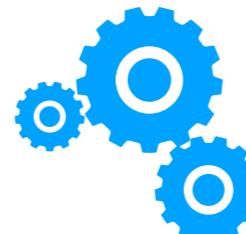
experts' proof documents =
[(proof goal * good induction)]



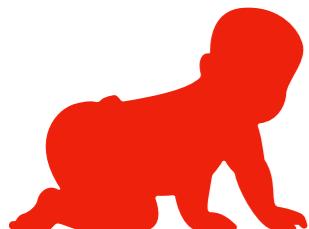
experts
(= me)



20 heuristics



new users



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

It is difficult to write induction heuristics.

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

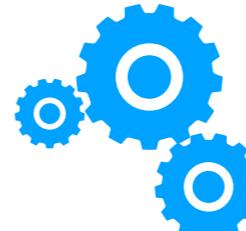
experts' proof documents =
[(proof goal * good induction)]



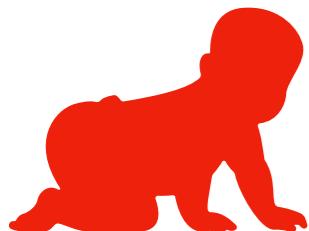
experts
(= me)



20 heuristics



new users



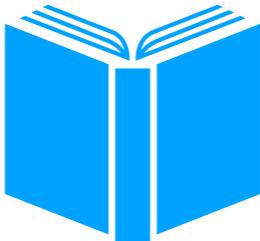
```
lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
```

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

It is difficult to write induction heuristics.

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

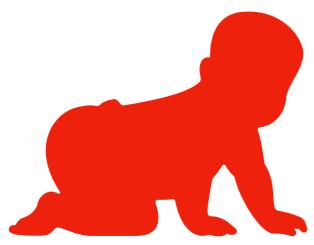
experts' proof documents =
[(proof goal * good induction)]



experts
(= me)

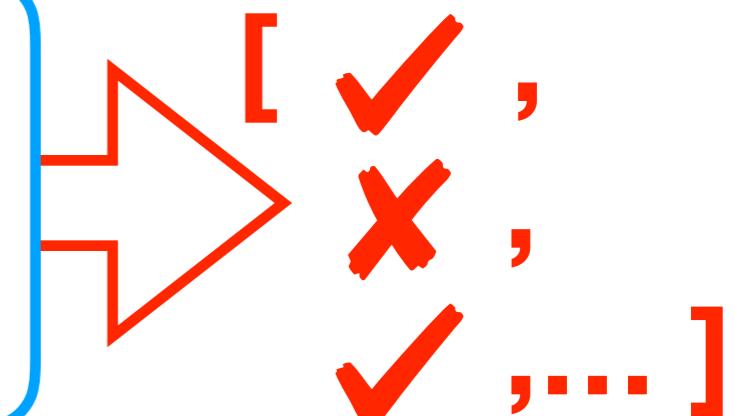
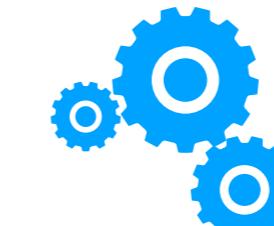


new users



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

20 heuristics



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

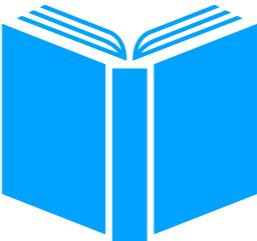
new proof goal consisting of new constants and variables of new types!

It is difficult to write induction heuristics.

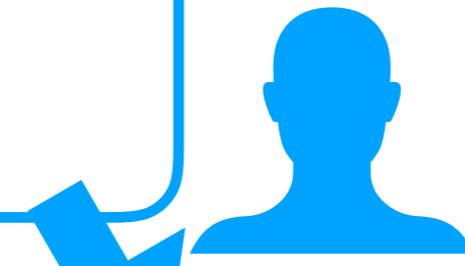
① blue = what I can see before releasing smart_induct

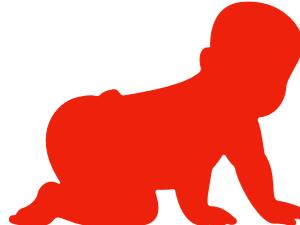
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

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

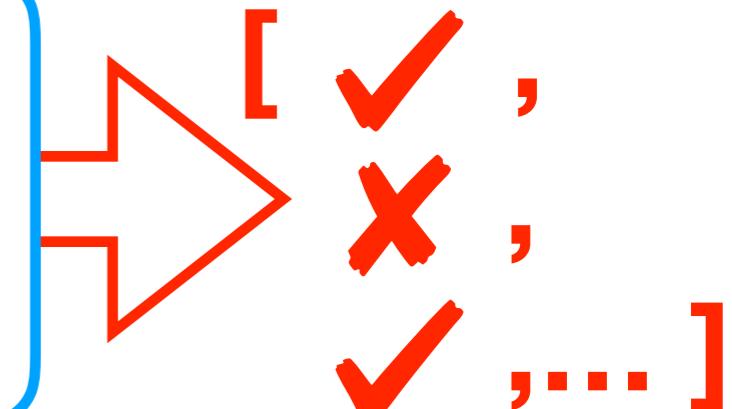
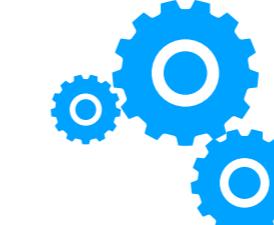


experts
(= me)



new users

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

20 heuristics



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

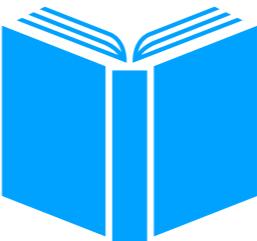
new proof goal consisting of new constants and variables of new types!

It is difficult to write induction heuristics.

① blue = what I can see before releasing smart_induct

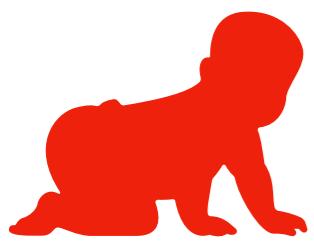
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

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

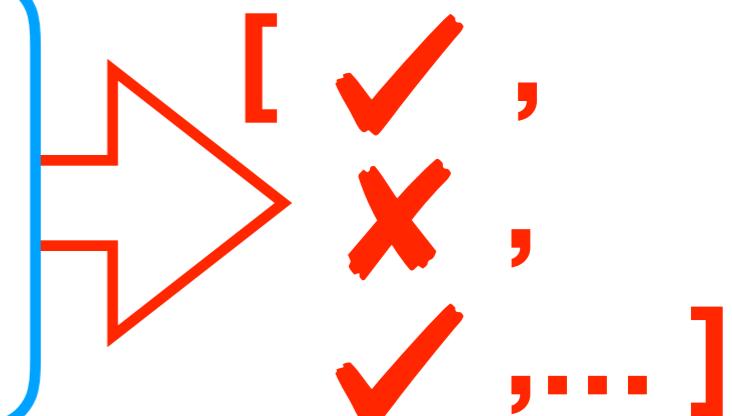
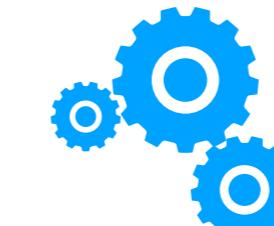


experts
(= me)



new users

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

20 heuristics



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

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

② red = what will be developed after releasing smart_induct

It is difficult to write induction heuristics.

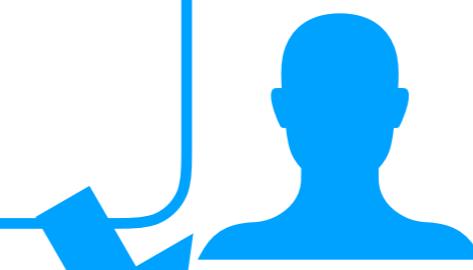
① blue = what I can see before releasing smart_induct

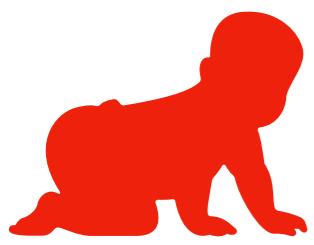
```
lemma "itrev xs ys = rev xs @ ys"  
by(induct xs ys rule:"itrev.induct") auto
```

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



experts
(= me)



new users

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

20 heuristics
in LiFtEr



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

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

② red = what will be developed after releasing smart_induct

It is difficult to write induction heuristics.

① blue = what I can see before releasing smart_induct

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

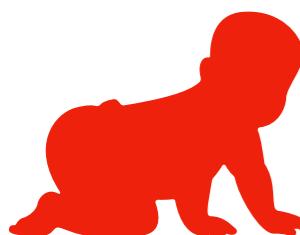
experts' proof documents =
[(proof goal * good induction)]



experts
(= me)

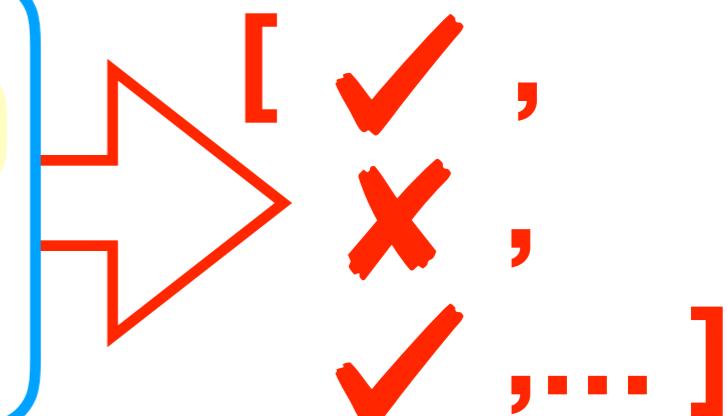
LiFtEr: Logical
Feature Extractor

new users



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

20 heuristics
in LiFtEr



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

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

② red = what will be developed after releasing smart_induct

Abstract Syntax of LiFtEr

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

Abstract Syntax of LiFtEr

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

Example Heuristic 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 Heuristic in LiFtEr (in Abstract Syntax)

implication



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

\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

$t2 \text{ is_nth_induction_term } n$

Example Heuristic 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 Heuristic in LiFtEr (in Abstract Syntax)

implication

↓

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

→

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

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

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

$r1 \text{ is_rule_of } to1$

 ∧ 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 Heuristic 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 Heuristic 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

\wedge conjunction

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

Example Heuristic 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

\wedge conjunction

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

Example Heuristic 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 Heuristic in LiFtEr (in Abstract Syntax)

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

$t2 \text{ is_nth_induction_term } n$

Example Heuristic in LiFtEr (in Abstract Syntax)

LiFtEr heuristic: (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

$t2 \text{ is_nth_induction_term } n$

$r1$

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

$(t1 = \text{itrev})$

```
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

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

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

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

$r1 \text{ is_rule_of } to1$

\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

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

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

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

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

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

$r1 \text{ is_rule_of } to1$

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

$to1$

$r1$

```
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

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

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

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

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

r1

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

($to1 = \text{itrev}$)

r1 is_rule_of to1

True! r1 (= itrev.induct) is a lemma about to1 (= 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)"

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 is_rule_of to1

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

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

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

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

to1

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

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

\rightarrow

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

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

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

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

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

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

\wedge

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

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

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

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

\wedge

$t2 \text{is_nth_induction_term } n$

r1

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

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

to1

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

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

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

\rightarrow

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

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

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

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

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

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

\wedge

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

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

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

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

\wedge

$t2 \text{is_nth_induction_term } n$

r1

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

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

to1

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

t2

r1

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

\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 $to1 (= \text{itrev}).$

^

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

($t2 = xs \text{ and } ys$)

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

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

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

^

$t2 \text{ is_nth_induction_term } n$

```

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

```

```

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

```

t_{01}

t_{02}

lemma "itrev xs ys = rev xs @ ys"

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

t_2

r_1

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

\rightarrow

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

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

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

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

\wedge

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

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

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

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

\wedge

$t_2 \text{ is_nth_induction_term } n$

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

($t_1 = \text{itrev}$)

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

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

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

```

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

```

```

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

```

t_{01}

t_{02}

lemma "itrev xs ys = rev xs @ ys"

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

t_2

r_1

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

\rightarrow

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

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

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

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

\wedge

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

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

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

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

\wedge

$t_2 \text{ is_nth_induction_term } n$

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

($t_1 = \text{itrev}$)

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

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

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

```

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

```

```

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

```

t_{01}

t_{02}

lemma "itrev xs ys = rev xs @ ys"

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

t_2

r_1

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

\rightarrow

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

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

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

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

\wedge

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

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

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

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

\wedge

$t_2 \text{ is_nth_induction_term } n$

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

($t_1 = \text{itrev}$)

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

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

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

```

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

```

```

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

```

t_{01}

first

t_{02}

lemma "itrev xs ys = rev xs @ ys"

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

t_2

first

r_1

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

\rightarrow

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

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

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

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

\wedge

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

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

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

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

\wedge

$t_2 \text{ is_nth_induction_term } n$

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

($t_1 = \text{itrev}$)

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

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

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

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

```

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

```

```

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

```

t_{01}

first

t_{02}

lemma "itrev xs ys = rev xs @ ys"

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

t_2

first

r_1

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

\rightarrow

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

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

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

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

\wedge

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

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

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

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

\wedge

$t_2 \text{ is_nth_induction_term } n$

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

($t_1 = \text{itrev}$)

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

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

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

when t_2 is xs ($n = 1$)



```

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

```

```

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

```

t_{01}

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

lemma "itrev xs ys = rev xs @ ys"

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

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

r_1

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

\rightarrow

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

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

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

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

\wedge

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

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

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

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

\wedge

$t_2 \text{ is_nth_induction_term } n$

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

($t_1 = \text{itrev}$)

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

($t_2 = xs$ and ys)

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

when t_2 is xs ($n = 1$)

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

```

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

```

```

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

```

t_{01}

lemma

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

apply(induct $xs\ ys$ **rule:"itrev.induct")**

apply auto done

t_2

t_{02}

first

second

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

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

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



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

\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 ys xs rule: itrev.induct)
  apply auto oops

 $\exists r1 : \text{rule}. \text{True}$ 
 $\rightarrow$ 
 $\exists r1 : \text{rule}.$ 
 $\exists t1 : \text{term}.$ 
 $\exists t01 : \text{term\_occurrence} \in t1 : \text{term}.$ 
 $\quad r1 \text{ is\_rule\_of } t01$ 
 $\wedge$ 
 $\forall t2 : \text{term} \in \text{induction\_term}.$ 
 $\quad \exists t02 : \text{term\_occurrence} \in t2 : \text{term}.$ 
 $\quad \exists n : \text{number}.$ 
 $\quad \quad \text{is\_nth\_argument\_of } (t02, n, t01)$ 
 $\wedge$ 
 $\quad t2 \text{ is\_nth\_induction\_term } n$ 

```

```

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)"

```

same lemma → **lemma** "itrev xs ys = rev xs @ ys"
apply(induct ys xs rule: itrev.induct)
apply auto **oops**

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

→

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

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

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

$r1 \text{ is_rule_of } to1$

\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

$t2 \text{ is_nth_induction_term } n$

```

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)"

```

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

different order → **apply**(induct ys xs rule: itrev.induct)
 $\exists r1 : \text{rule}. \text{True}$ **apply** auto **oops**

→

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

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

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

$r1 \text{ is_rule_of } tol$

\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

$t2 \text{ is_nth_induction_term } n$

```
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)
```

same lemma → lemma "itrev xs ys = rev xs @ ys"

different order → apply(induct ys xs rule: itrev.induct)
apply auto oops

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

→

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

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

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

$r1 \text{ is_rule_of } to1$

\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

$t2 \text{ is_nth_induction_term } n$

```
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)
```

same lemma → lemma "itrev xs ys = rev xs @ ys"

different order → apply(induct ys xs rule: itrev.induct)
apply auto oops

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

→

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

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

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

$r1 \text{ is_rule_of } to1$

\wedge

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

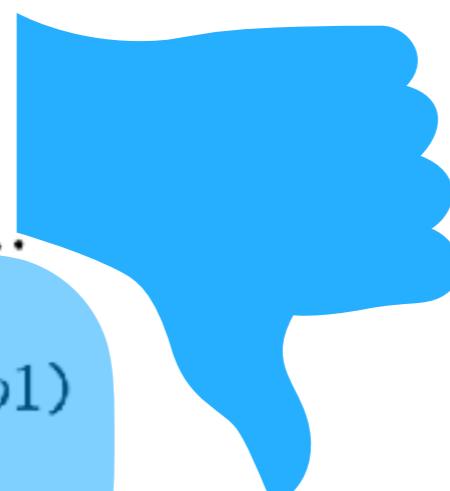
$\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$



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

\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



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

\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"

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

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

(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

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

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

($t1 = \text{exec}$)

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

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

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

new types ->

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

new constants ->

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

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

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

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

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

to1

r1

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

(r1 = 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 $t1 (= \text{exec})$.



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

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

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

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



$t2 \text{ is_nth_induction_term } n$

new types ->

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

new constants ->

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

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

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

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

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

to1

r1

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

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

($t1 = \text{exec}$)

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

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

\wedge

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

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

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

is_nth_argument_of (t2, n, to1)

\wedge

t2 is_nth_induction_term n

new types ->

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

new constants ->

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

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

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

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

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

b2

to1

r1

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

(r1 = exec.induct)

(t1 = exec)

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

r1 is_rule_of to1 True! r1 (= exec.induct) is a lemma about to1 (= exec).

^

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

(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)

^

t2 is_nth_induction_term n

new types ->

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

new constants ->

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

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

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

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

→

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

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

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

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

^

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

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

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

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

^

$t2 \text{ is_nth_induction_term } n$

new types ->

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

new constants ->

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

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

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

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

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



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

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

($t1 = \text{exec}$)

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

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



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

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

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

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



$t2 \text{ is_nth_induction_term } n$

new types ->

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

new constants ->

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

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

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

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

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



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

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

($t1 = \text{exec}$)

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

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



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

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

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

($to2 = \text{is1, s, and stk}$)

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

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



$t2 \text{ is_nth_induction_term } n$

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**

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

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

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

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



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

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

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

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



$t2 \text{ is_nth_induction_term } n$

new types ->

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

new constants ->

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

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

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

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

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

\rightarrow

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

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

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

$r1 \text{ is_rule_of } to1$ 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

$t2 \text{ is_nth_induction_term } n$

t_2

t_{01}

t_{02}

first

r_1

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

$(t_1 = \text{exec})$

$(to1 = \text{exec})$

$(t_2 = \text{is1, s, and stk})$

$(to2 = \text{is1, s, and stk})$

when t_2 is is1 ($n \rightarrow 1$)



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)

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

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

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

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



$t2 \text{ is_nth_induction_term } n$

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

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

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

→

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

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

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

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

^ 

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

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

(t2 = is1, s, and stk)

(t02 = is1, s, and stk)

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

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

^

$t2 \text{ is_nth_induction_term } n$

when t2 is is1 (n → 1) 

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"
```

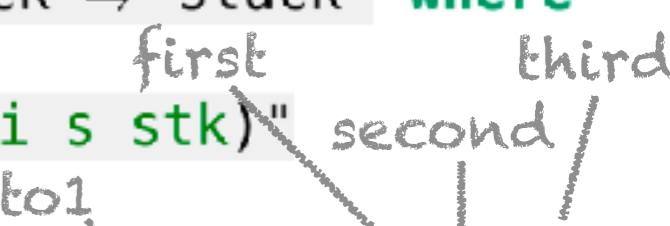
```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

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



→

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

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

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

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



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

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

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

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



$t2 \text{ is_nth_induction_term } n$

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

($to2 = is1, s, \text{and } stk$)

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

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

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



new types →

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

new constants \rightarrow

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

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

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

apply auto done

→

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

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

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

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

8

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

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

\exists n : number.

is_nth_argument_of (to2, n, to1)

A

t2 is_nth_induction_term *n*

when t_2 is $i s 1$ ($n \rightarrow 1$)

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 third
  "exec (i#is) s stk = exec is s (exec1 i s stk)" second
t2 t01
lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

```

new lemma \rightarrow **lemma** "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
 a model proof \rightarrow **apply(induct is1 s stk rule:exec.induct)**
 : rule. True **apply auto done**


 : rule.
 t_1 : term.
 $\exists t_{01} : \text{term_occurrence} \in t_1 : \text{term. } (t_{01} = \text{exec})$

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

1

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

($t_2 = is1, s, \text{ and } stk$)

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

(to2 = is1, s, and stk)

$\exists n$: number.

is_nth_argument_of (to2, n, to1)

when t_2 is $i s 1$ ($n \rightarrow 1$)

t2 : s_nth_induction_term n

what is t_3 if $t_1 = 3^2$?

```

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

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)"

lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
apply(induct stk s is1 rule:exec.induct)
  ∃ r1 : rule. True  oops
→
  ∃ 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

```

```

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

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)"

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

```

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

→

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

\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
 $t2 \text{ is_nth_induction_term } n$

the same Lifter assertion



```

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

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)"

lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
apply(induct stk s is1 rule:exec.induct)
  ∃ r1 : rule. True  oops
→
  ∃ 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

```

```

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

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)"

```

same lemma → lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
 apply(induct stk s is1 rule:exec.induct)

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

→

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

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

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

$r1 \text{ is_rule_of } tol$

\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

$t2 \text{ is_nth_induction_term } n$

```

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

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)"

```

same lemma → lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

different order → apply(induct stk s is1 rule:exec.induct)

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

→

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

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

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

$r1 \text{ is_rule_of } to1$

\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

$t2 \text{ is_nth_induction_term } n$

```

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

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)"

```

same lemma → lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

different order → apply(induct stk s is1 rule:exec.induct)
 $\exists r1 : \text{rule}. \text{True}$ oops

→

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

$r1 \text{ is_rule_of } to1$

\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

$t2 \text{ is_nth_induction_term } n$

```

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

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)"

```

same lemma → lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

different order → apply(induct stk s is1 rule:exec.induct)
 $\exists r1 : \text{rule. True}$ oops

→

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

$r1 \text{ is_rule_of } to1$
 \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
 $t2 \text{ is_nth_induction_term } n$



smart_induct... does it work?

Table 1: Coincidence Rates of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

smart_induct... does it work?

Table 1: **Coincidence Rates** of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

smart_induct... does it work?

Table 1: **Coincidence Rates** of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

smart_induct... does it work?

Table 1: Coincidence Rates of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

smart_induct... does it work?

Table 1: Coincidence Rates of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

smart_induct... does it work?

Table 1: Coincidence Rates of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

smart_induct... does it work?

Table 1: Coincidence Rates of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

smart_induct... does it work?

Table 1: Coincidence Rates of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

Table 2: Coincidence Rates of `smart_induct` Based Only on Induction Terms.

theory	total	top_1	top_3	top_5	top_10
Nearest_Neighbors	16	5 (31%)	12 (75%)	15 (94%)	15 (94%)

smart_induct... does it work?

Table 1: Coincidence Rates of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

Table 2: Coincidence Rates of `smart_induct` Based Only on Induction Terms.

theory	total	top_1	top_3	top_5	top_10
Nearest_Neighbors	16	5 (31%)	12 (75%)	15 (94%)	15 (94%)

smart_induct... does it work?

Table 1: Coincidence Rates of smart_induct.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

Table 2: Coincidence Rates of smart_induct Based Only on Induction Terms.

theory	total	top_1	top_3	top_5	top_10
Nearest_Neighbors	16	5 (31%)	12 (75%)	15 (94%)	15 (94%)

Overall, the results are good!

But finding out what variables to generalise remains as a challenge!

smart_induct... does it work?

Table 1: Coincidence Rates of smart_induct.

theory	total	top_1	top_3	top_5	top_10
DFS	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
Nearest_Neighbors	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
RST_RBT	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

Table 2: Coincidence Rates of smart_induct Based Only on Induction Terms.

theory	total	top_1	top_3	top_5	top_10
Nearest_Neighbors	16	5 (31%)	12 (75%)	15 (94%)	15 (94%)

Overall, the results are good!

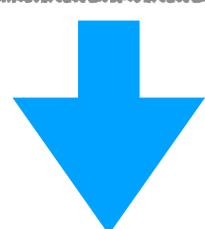
But finding out what variables to generalise remains as a challenge!

WIP!

Today I talked about...

2013 ~ 2017

Intern &
Engineer



PhD in
AI for theorem proving

2017 ~ 2018



2018 ~
2020/21



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

with Prof. Gerwin Klein



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

with Prof. Cezary Kaliszyk



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

with Dr. Josef Urban

Cogent

PSL

PaMpeR

LiFtEr

smart_induct

Today I talked about...

2013 ~ 2017

Intern &
Engineer

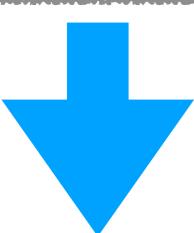


Cogent

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

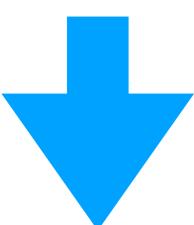


Cogent

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

<- 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)

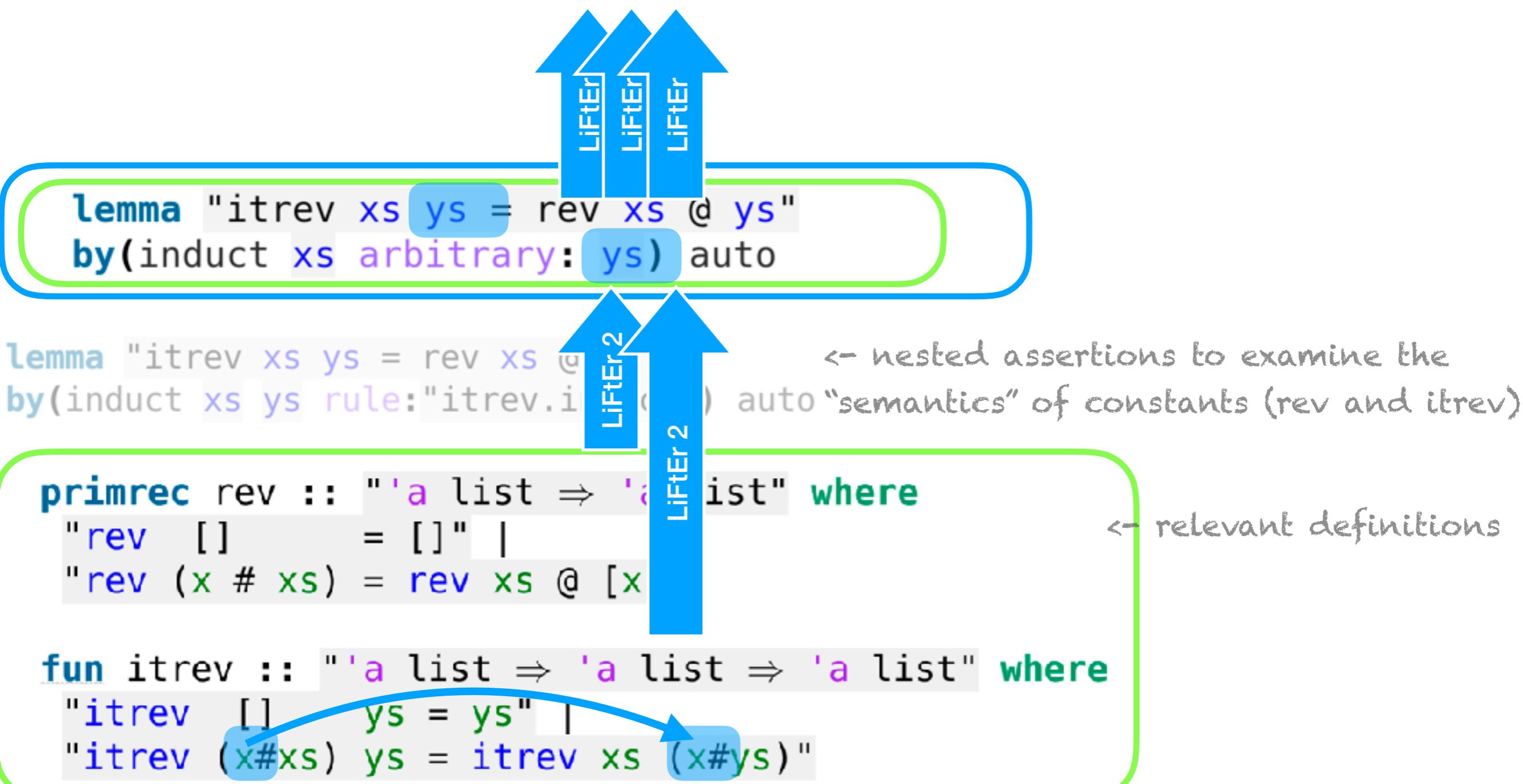
```
primrec rev :: "'a list ⇒ 'a list" where  
"rev [] = []" |  
"rev (x # xs) = rev xs @ [x]"
```

\leftarrow relevant definitions

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

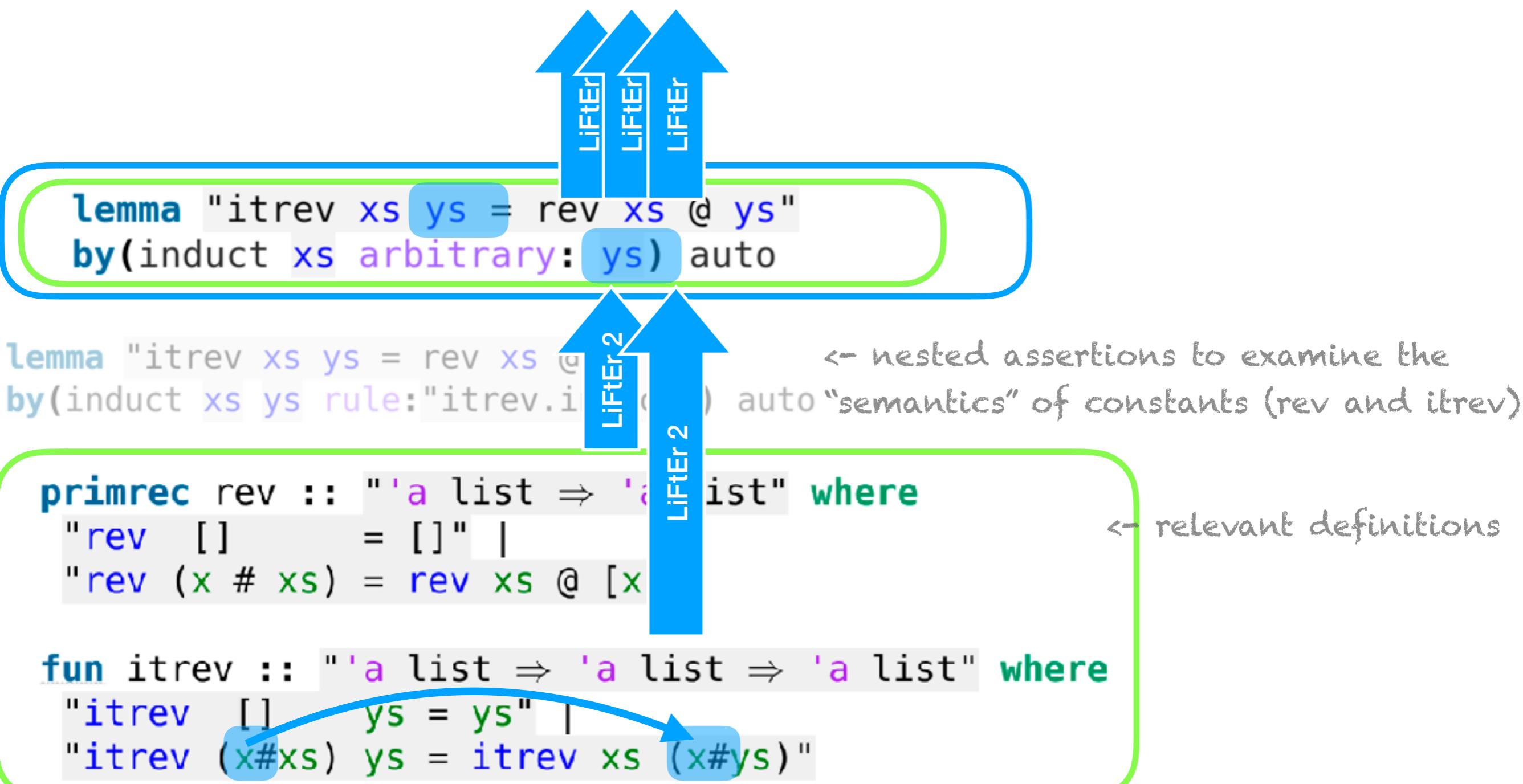
LiFtEr: (proof goal * induction arguments) -> bool

<- simple representation



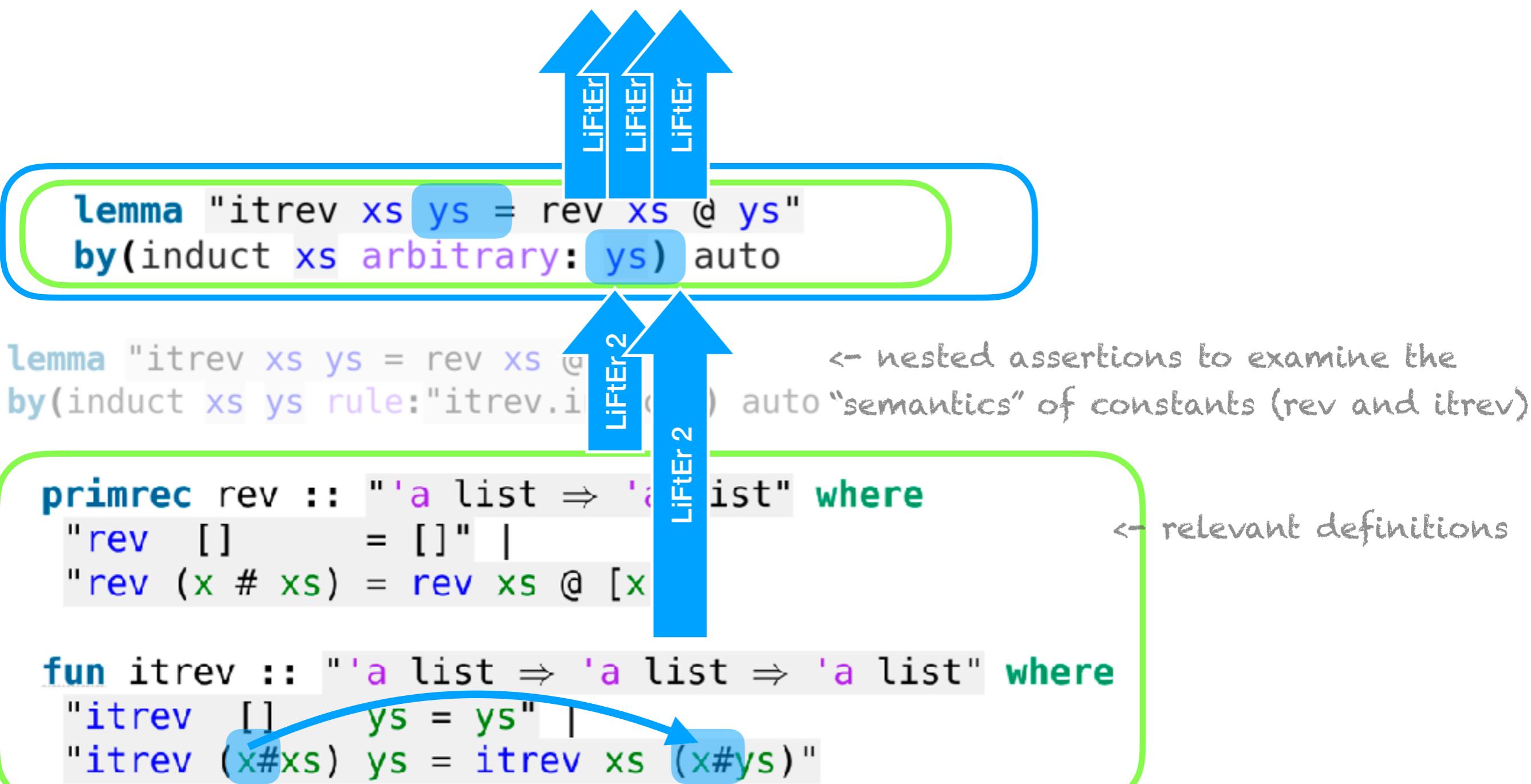
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

<- simple representation



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

coming soon

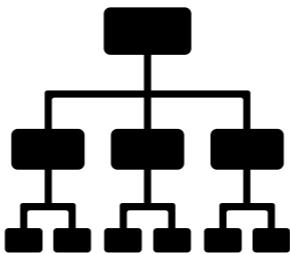


LiFtEr: (proof goal * induction arguments) -> bool

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

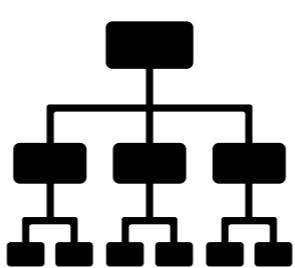
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

proof goal

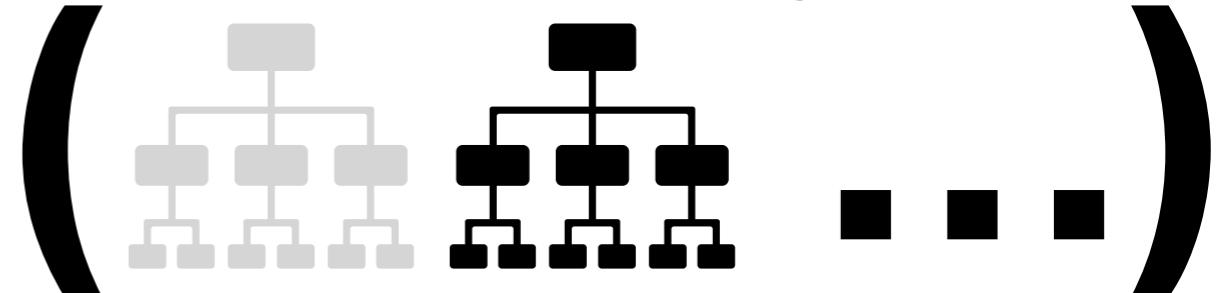


LiFtEr₂ (proof goal * induction arguments)-> bool
* relevant definitions

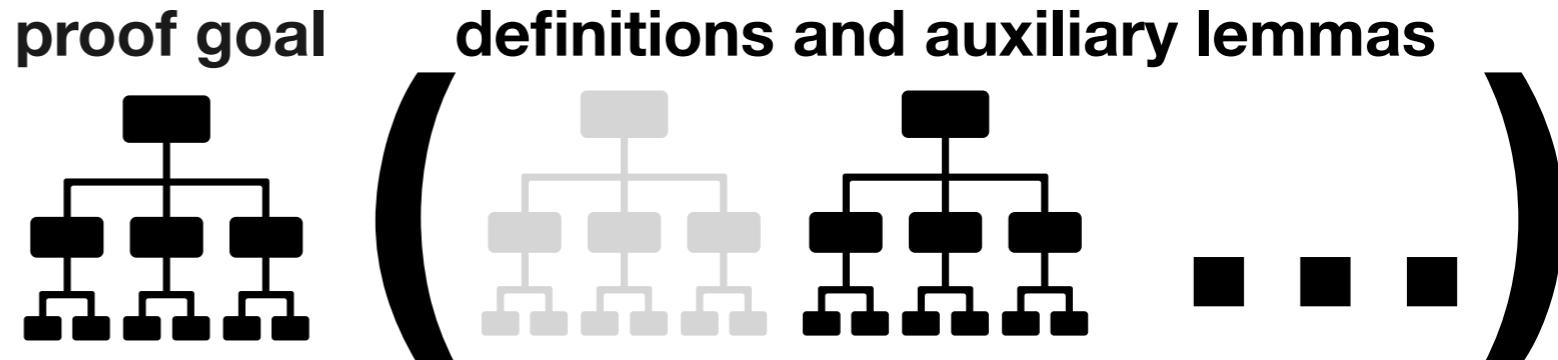
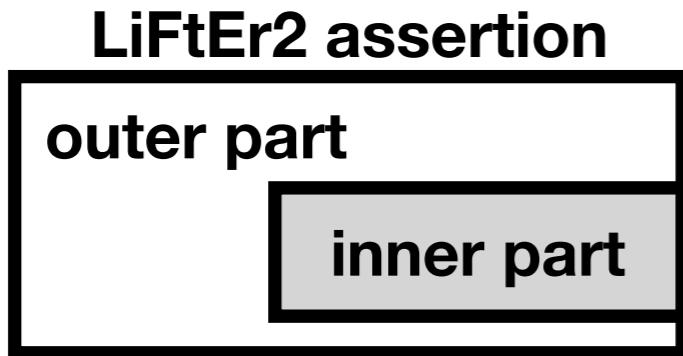
proof goal



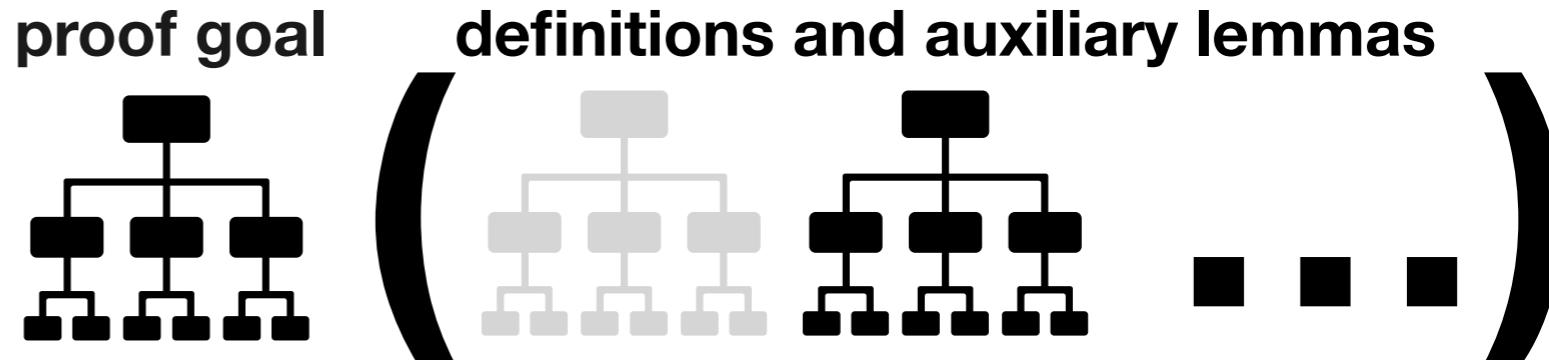
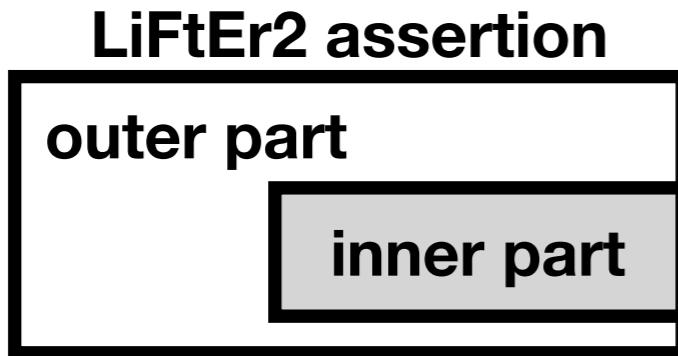
definitions and auxiliary lemmas



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

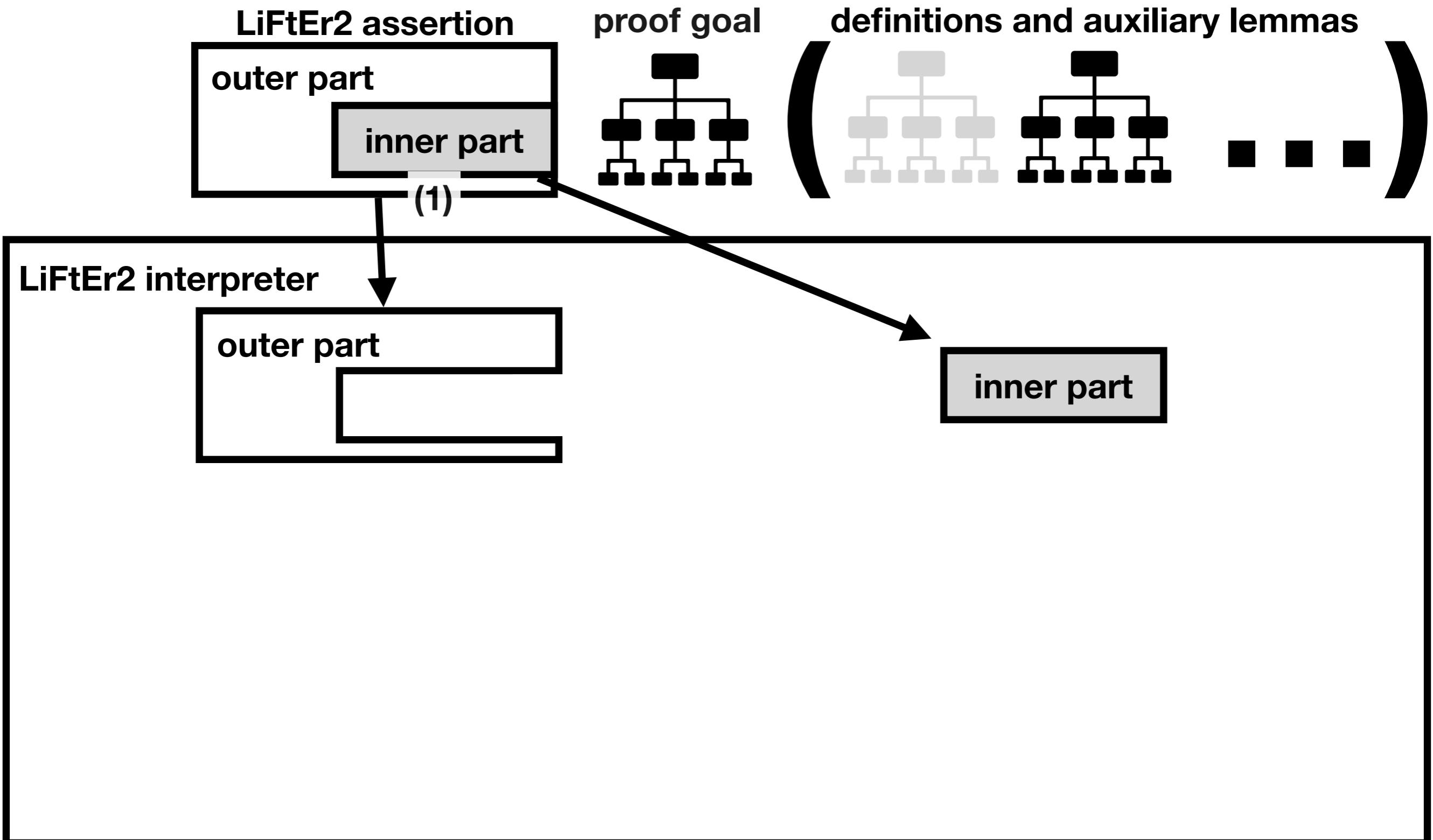


LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

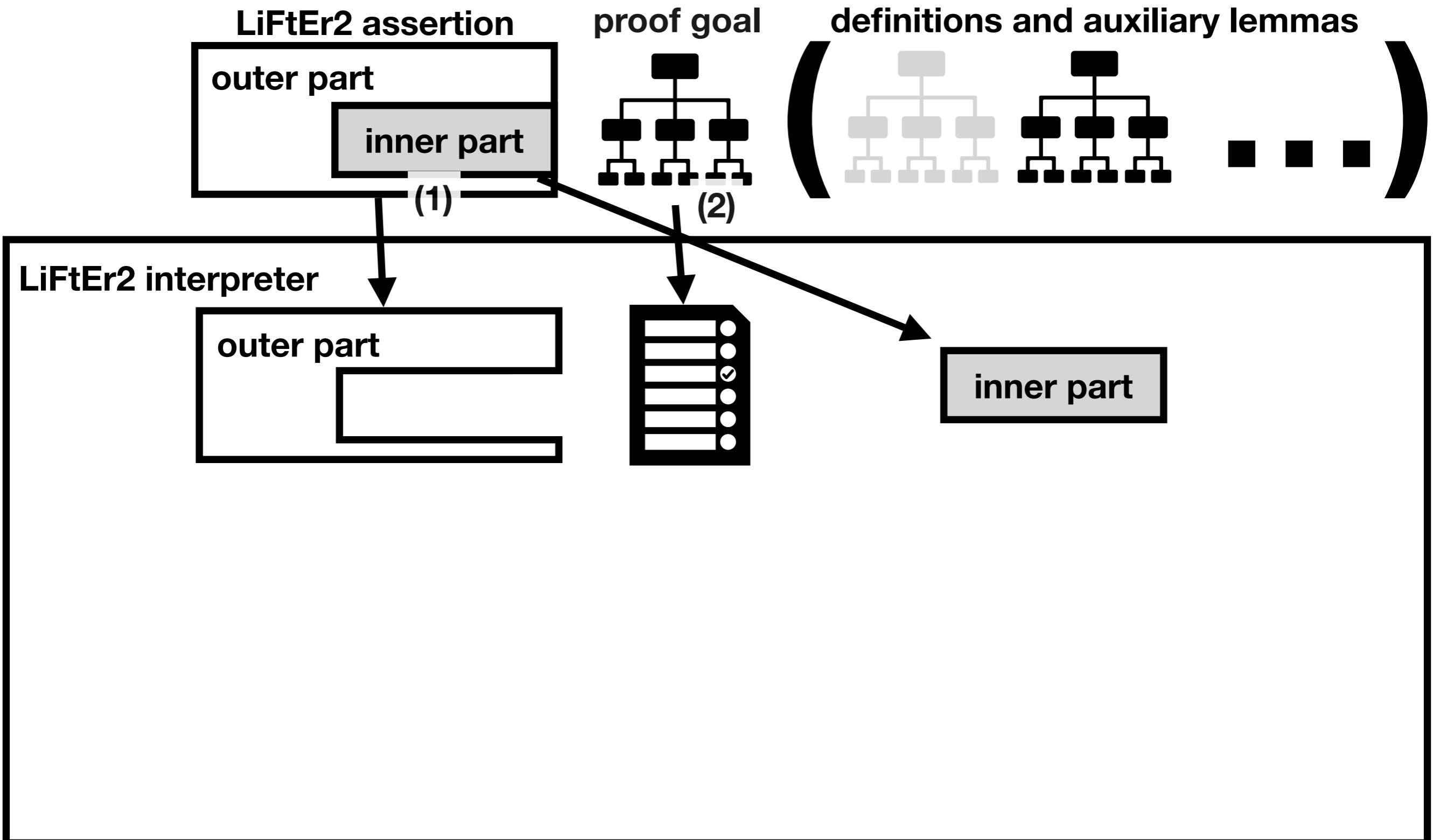


LiFtEr2 interpreter

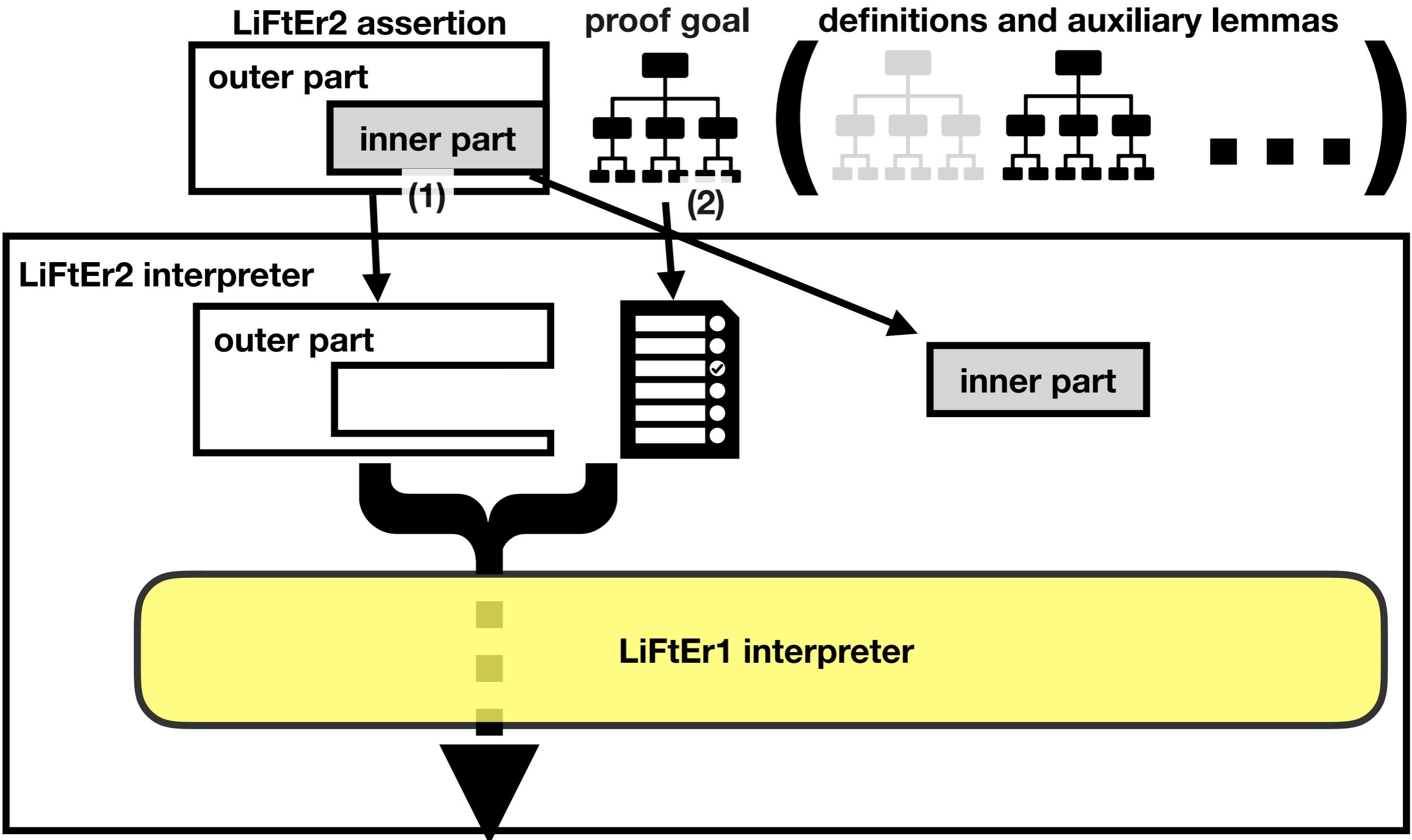
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



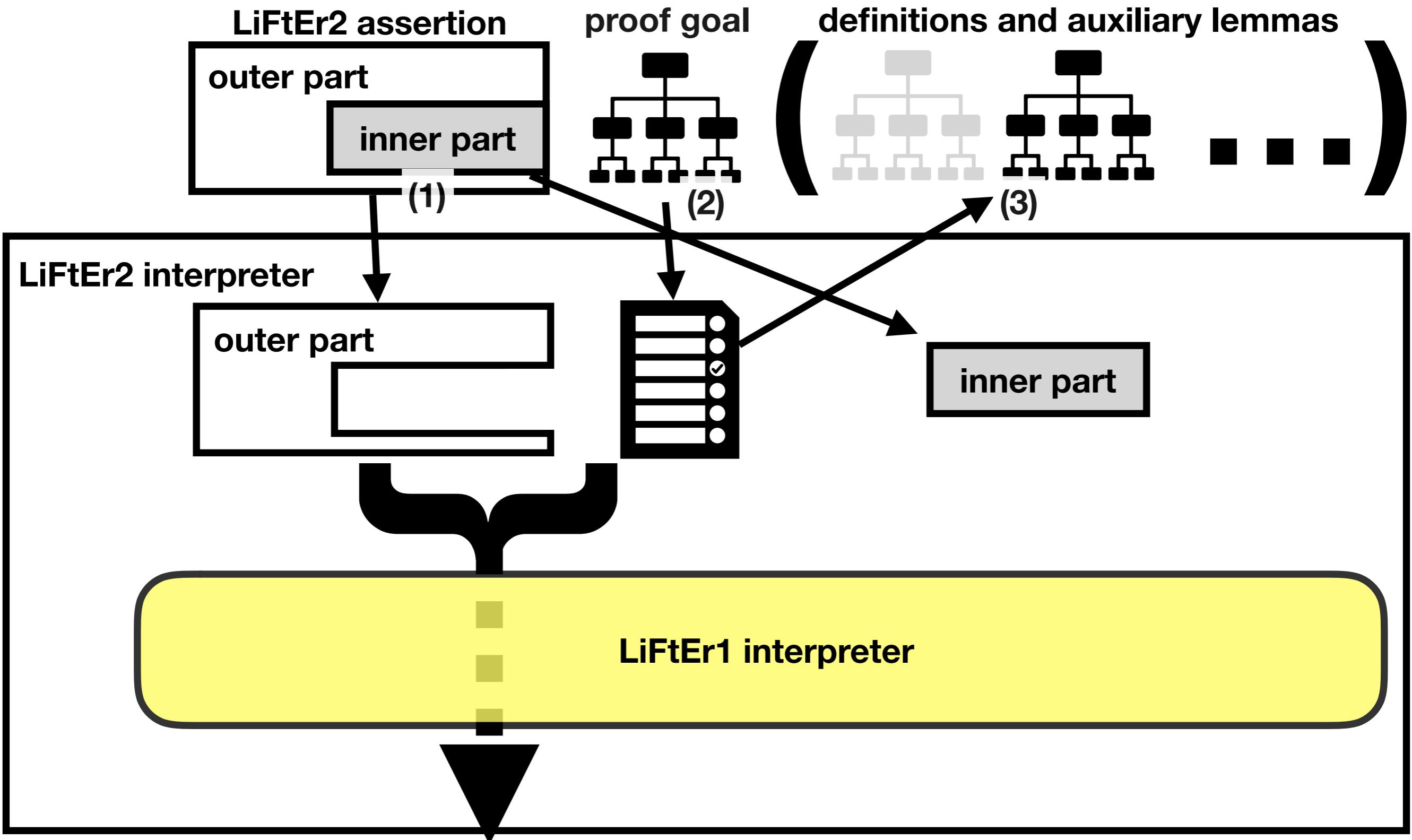
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



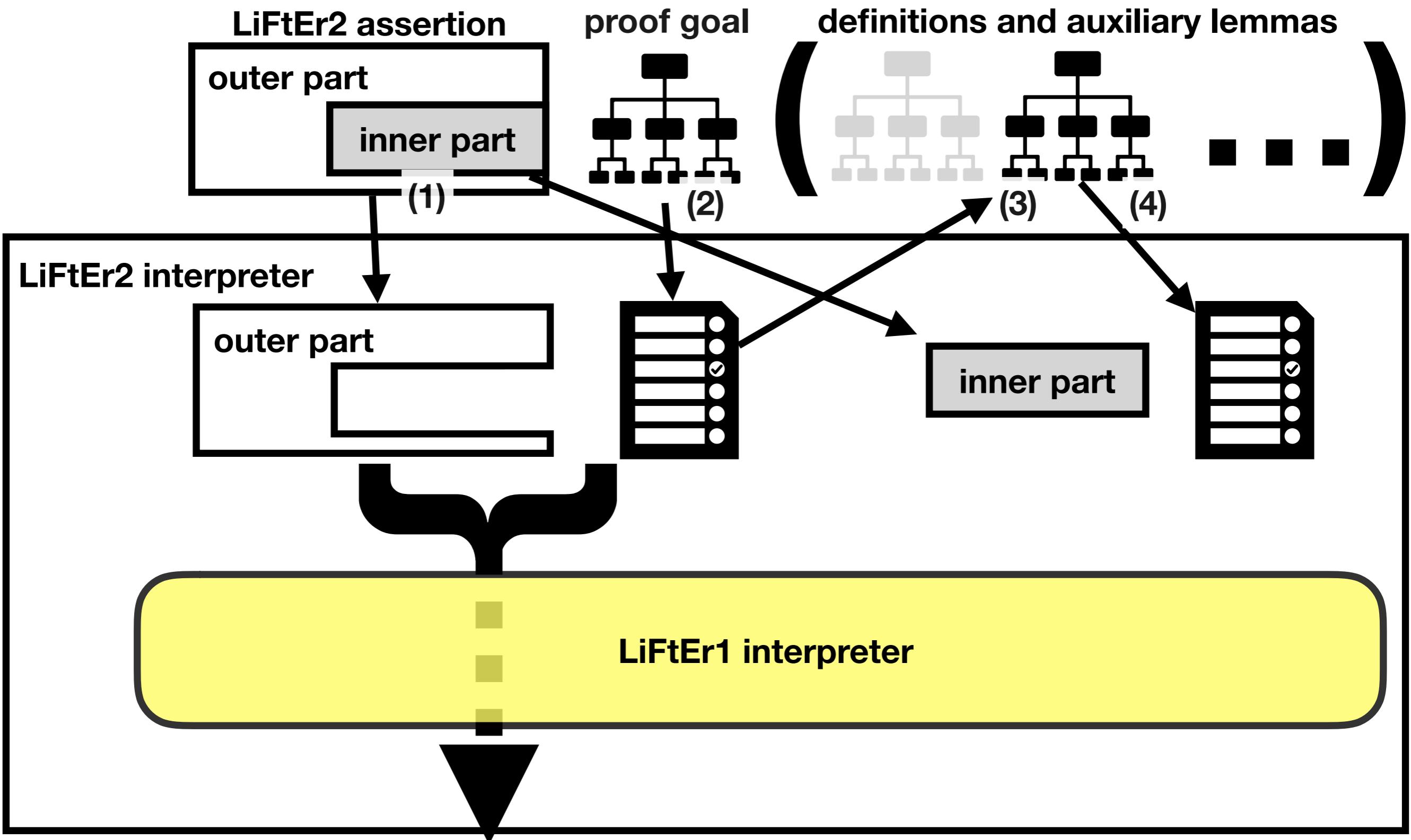
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



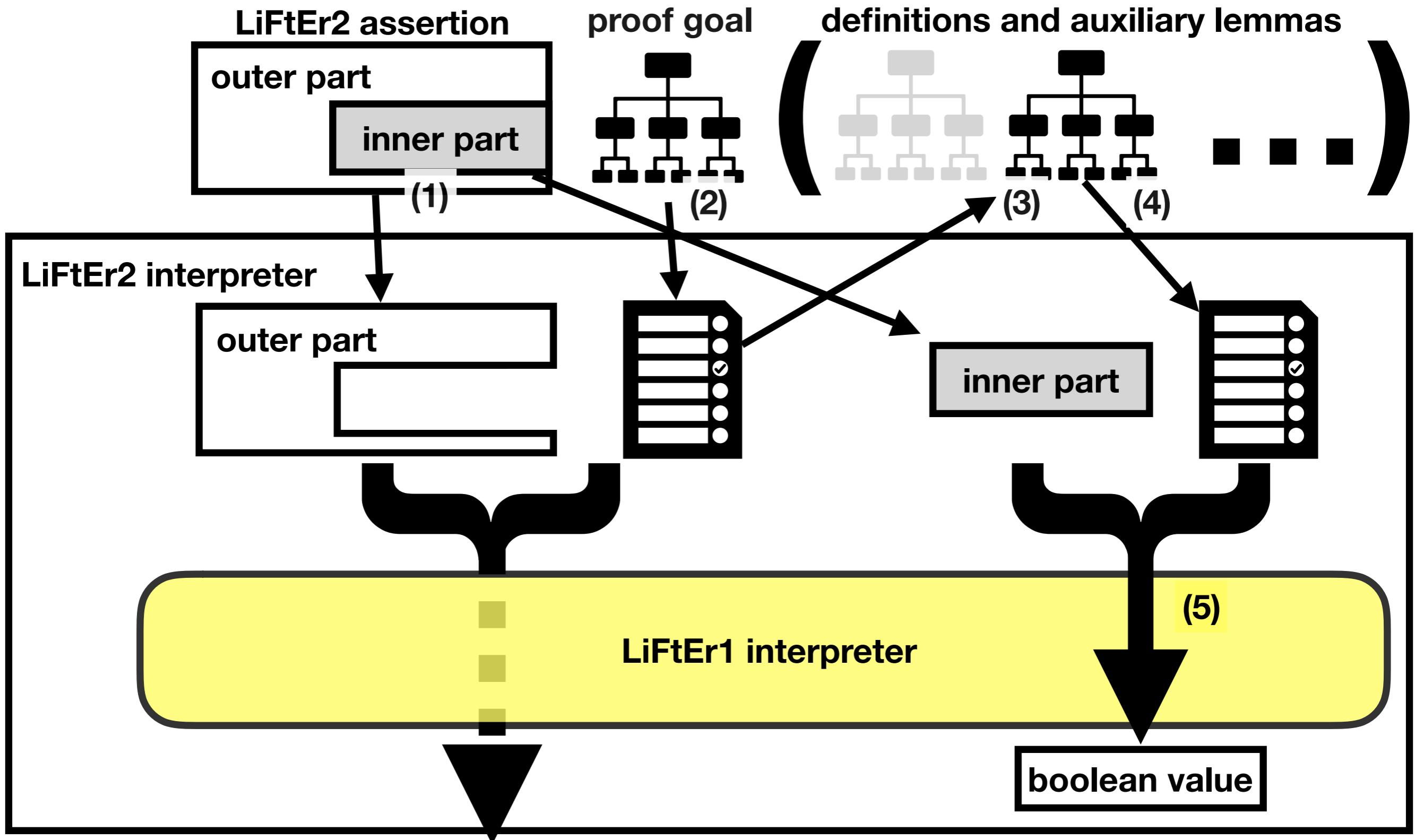
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



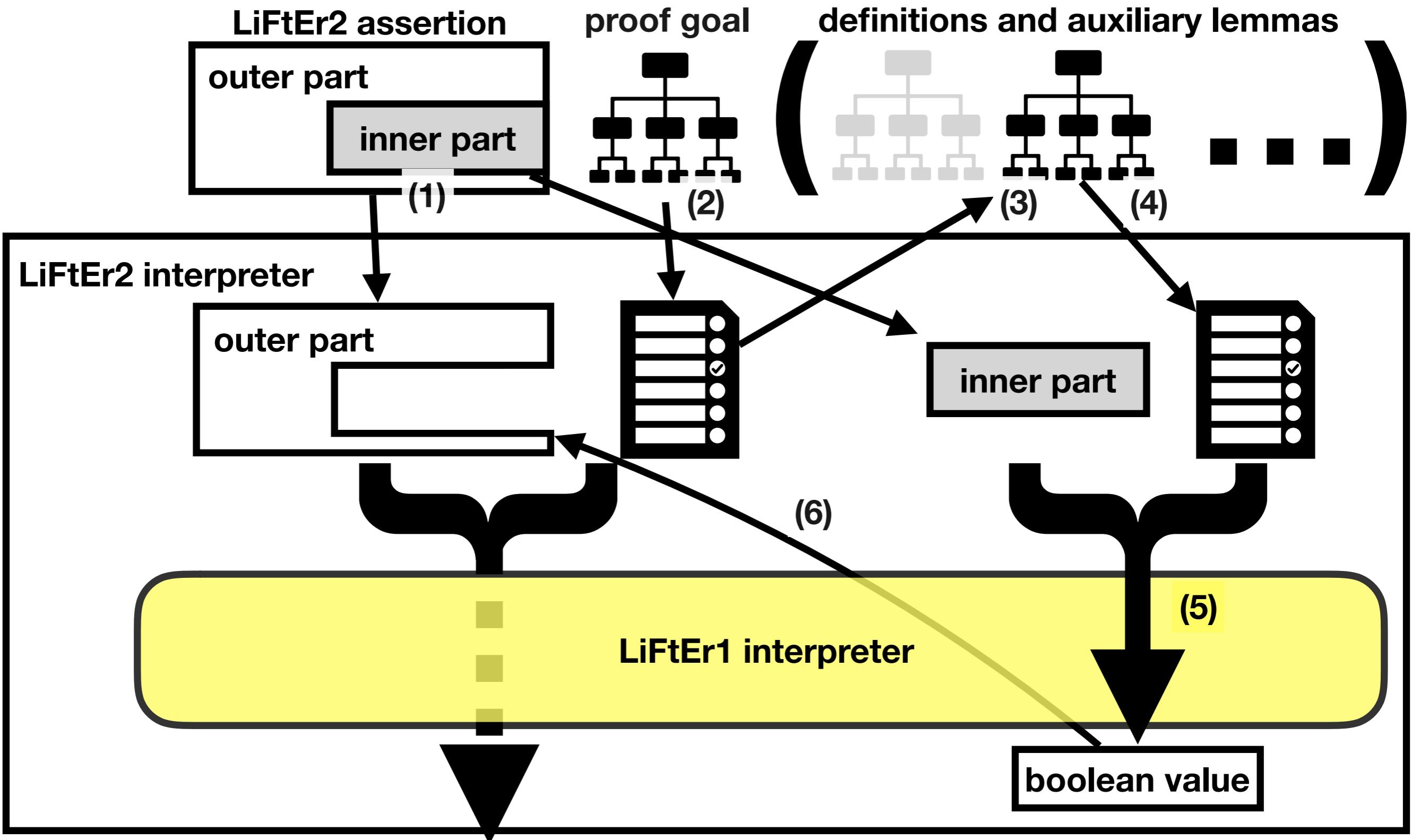
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



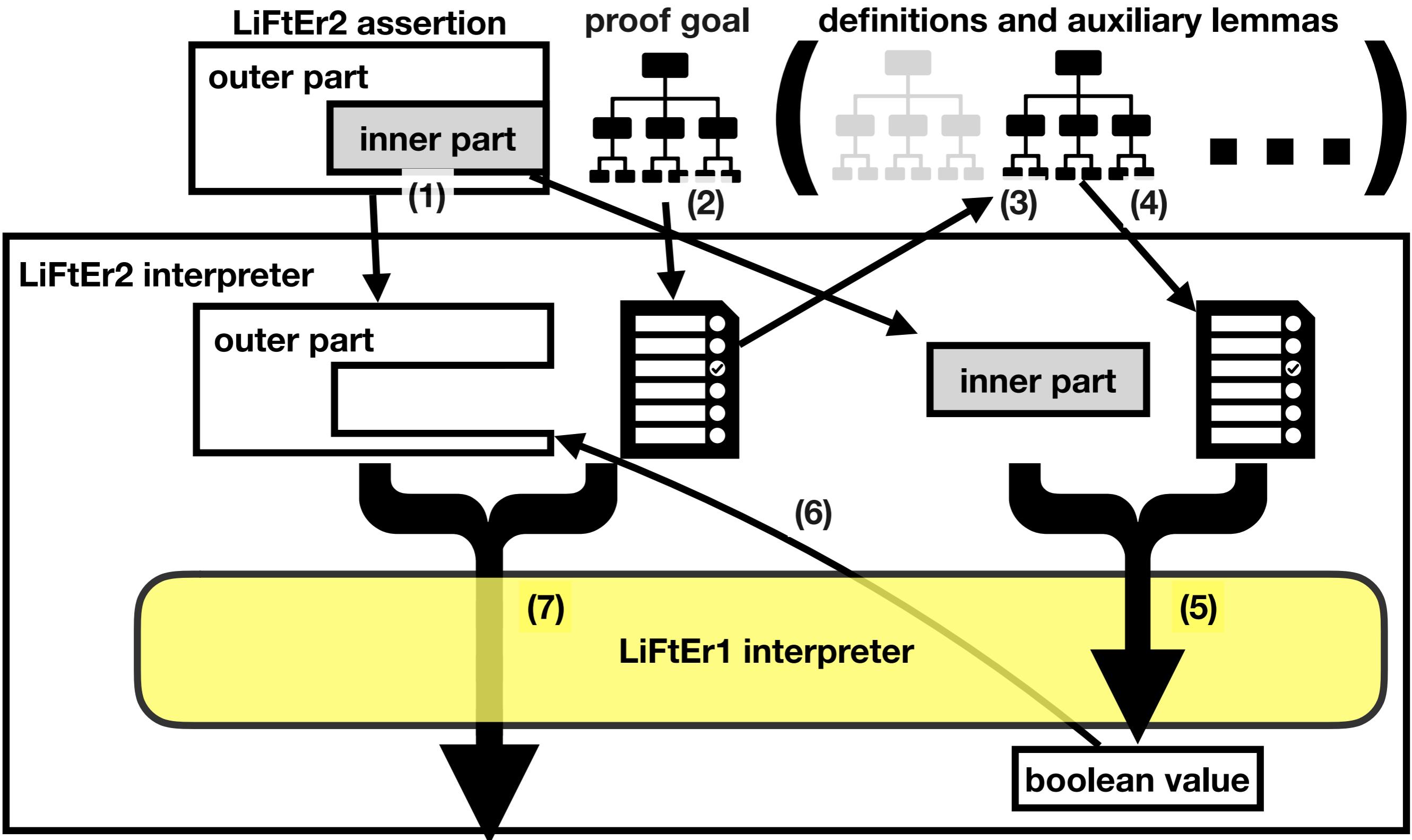
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



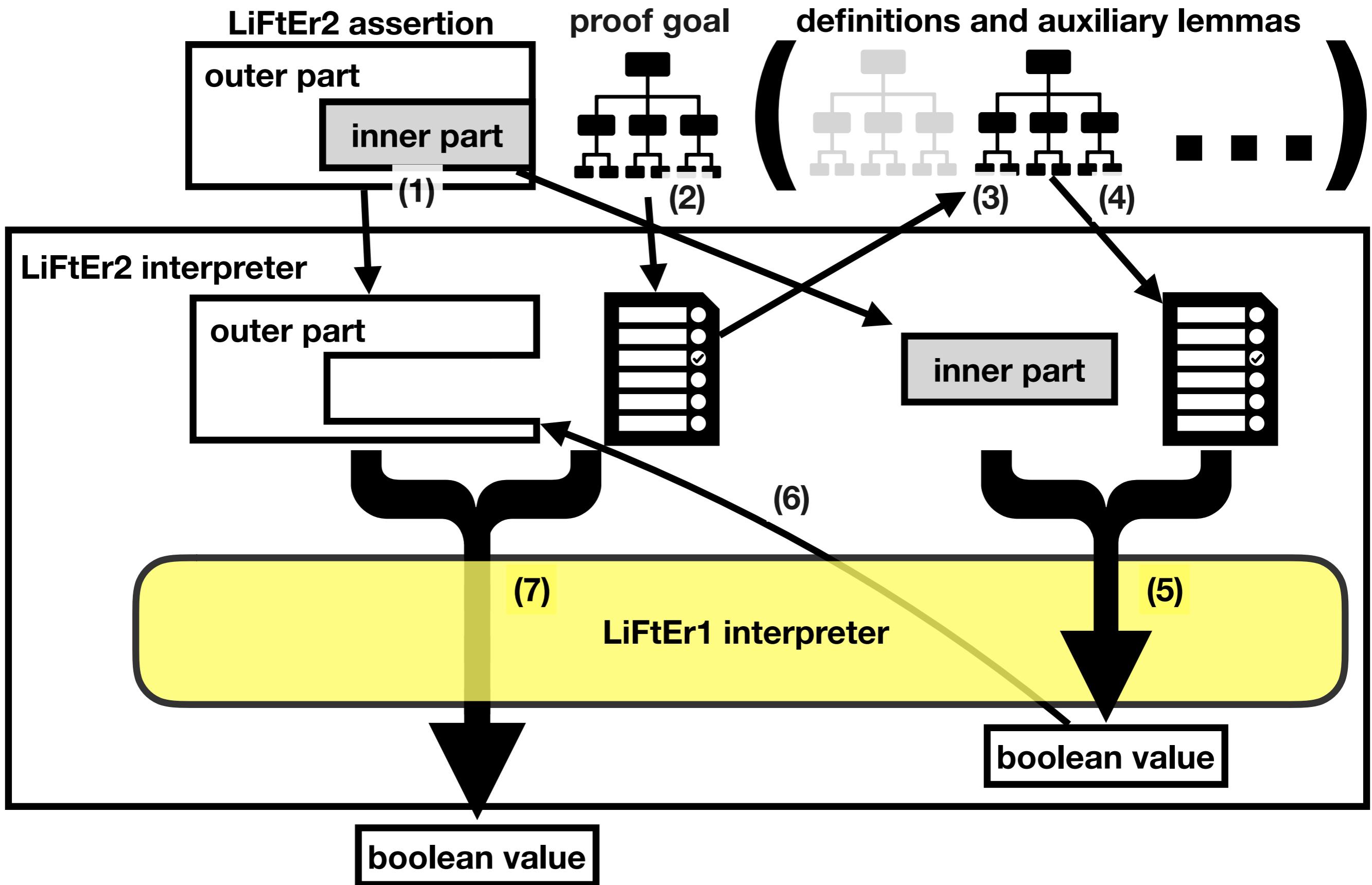
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions



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

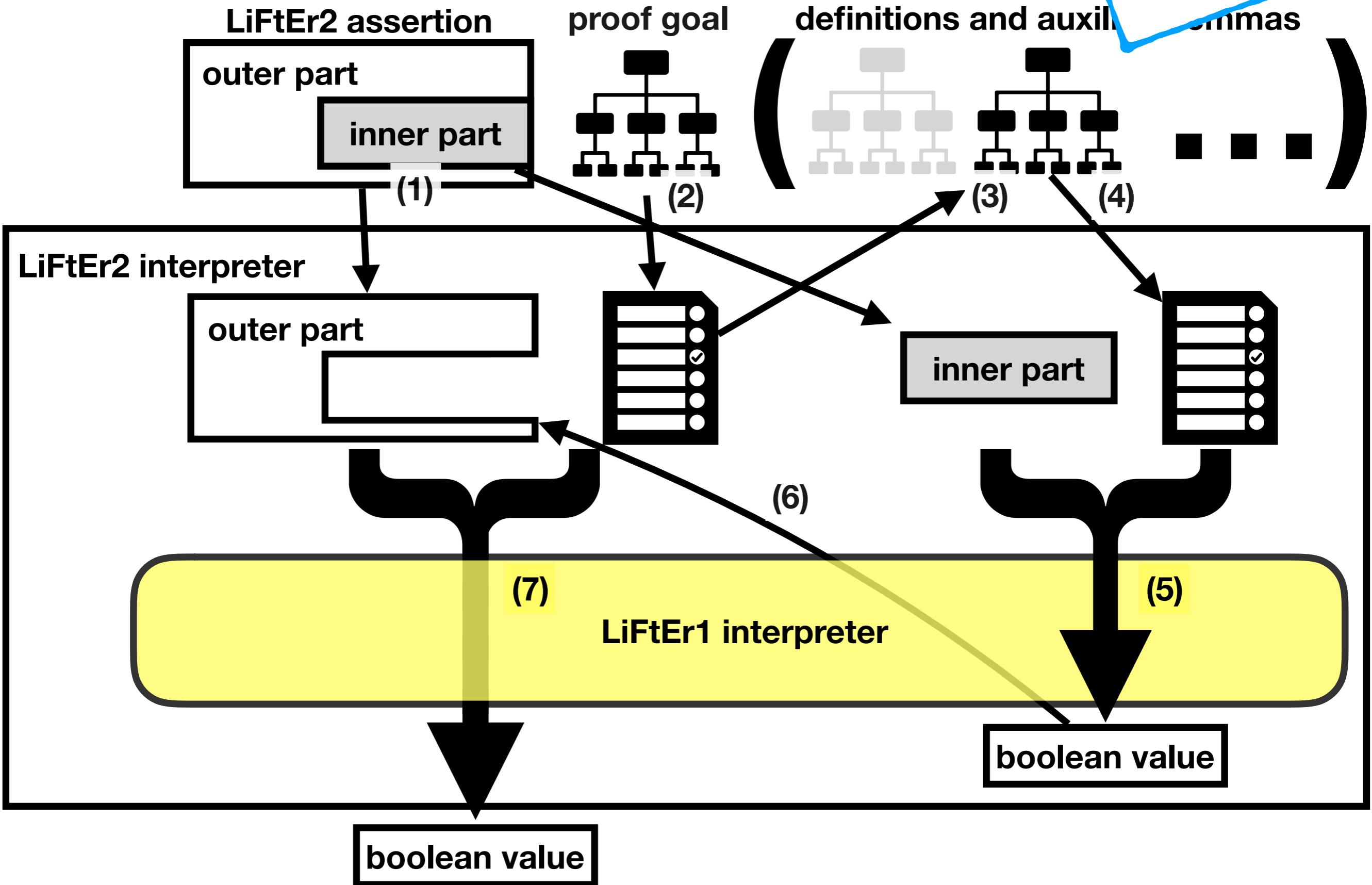


LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

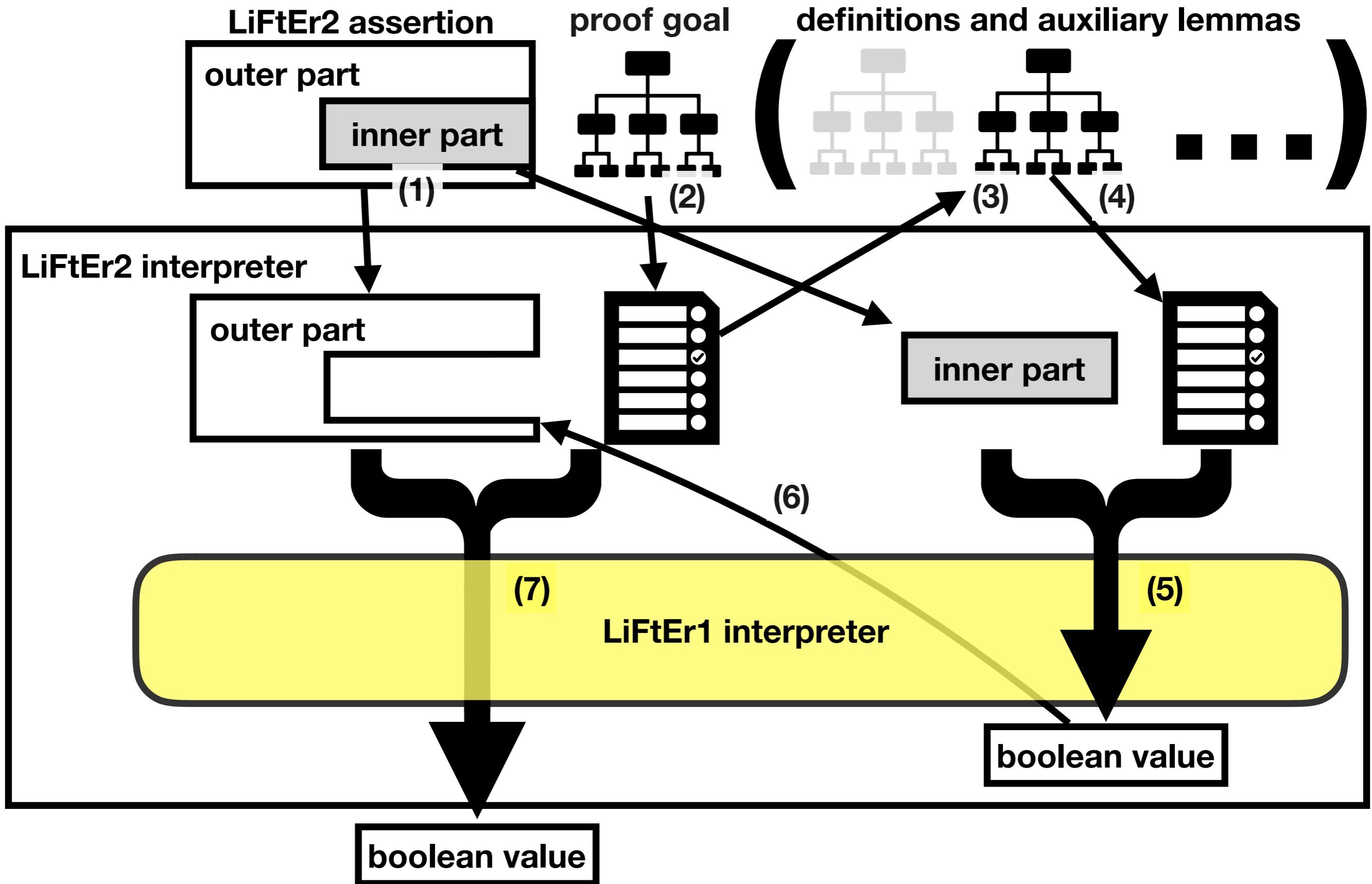


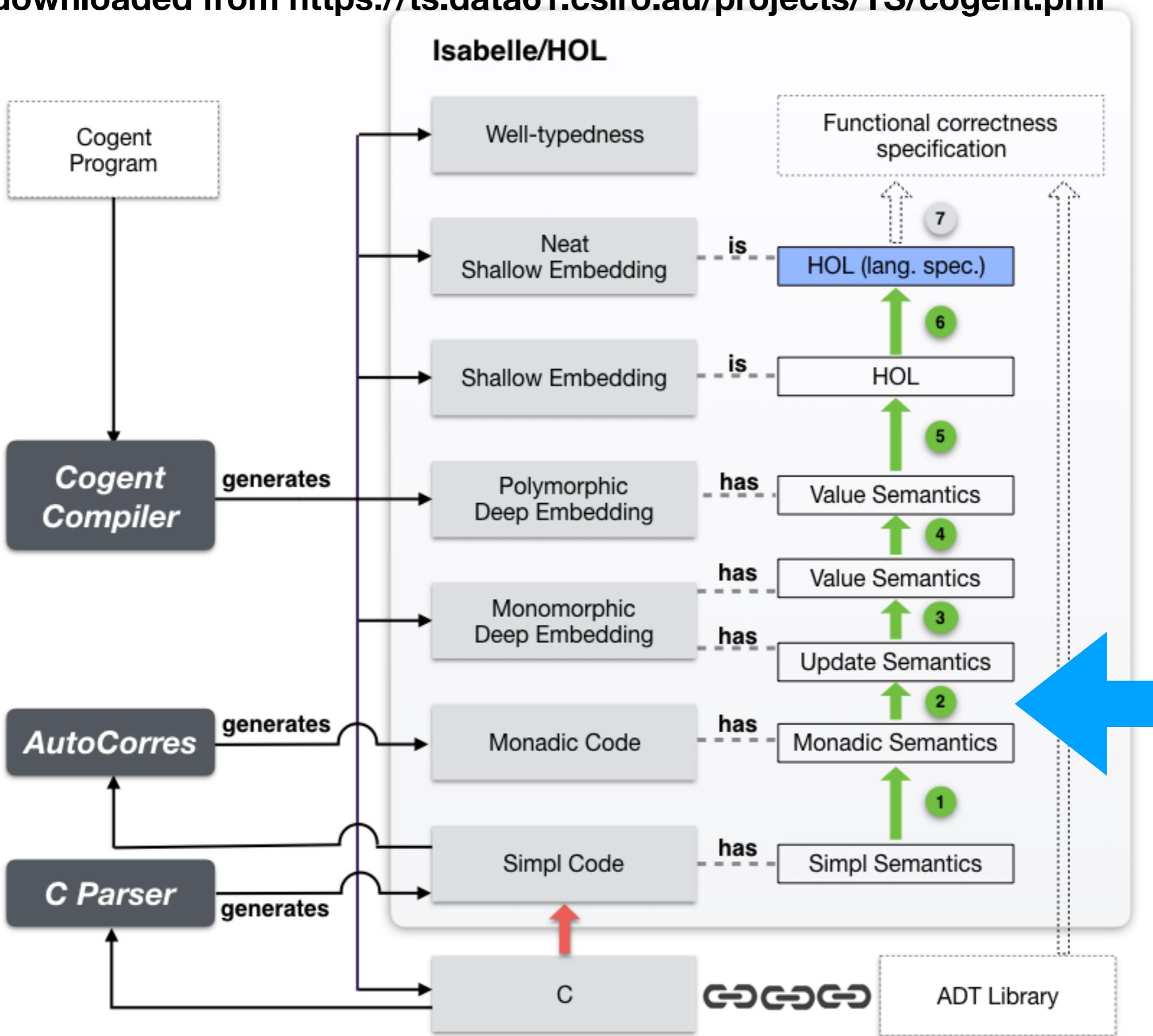
LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

WIP!

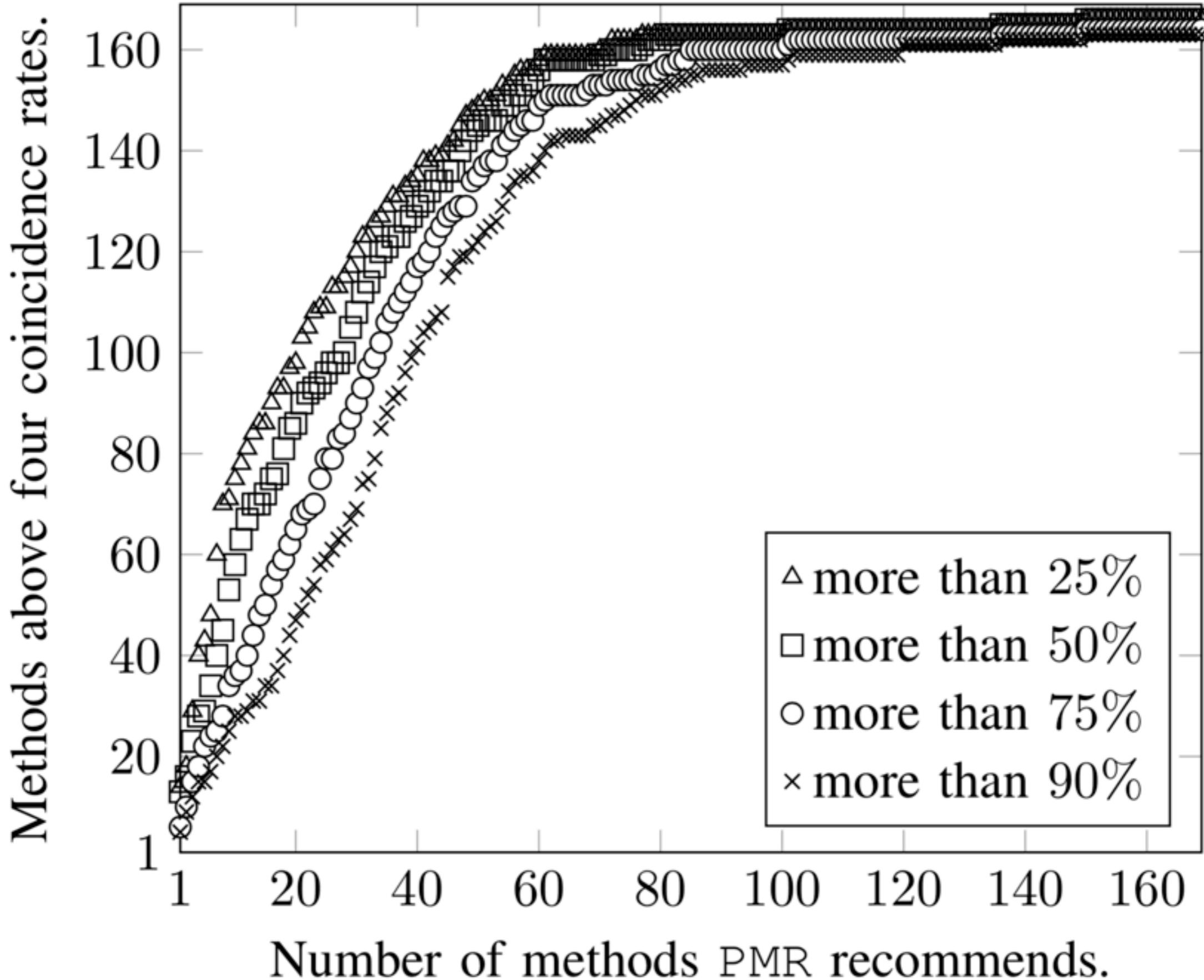


LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

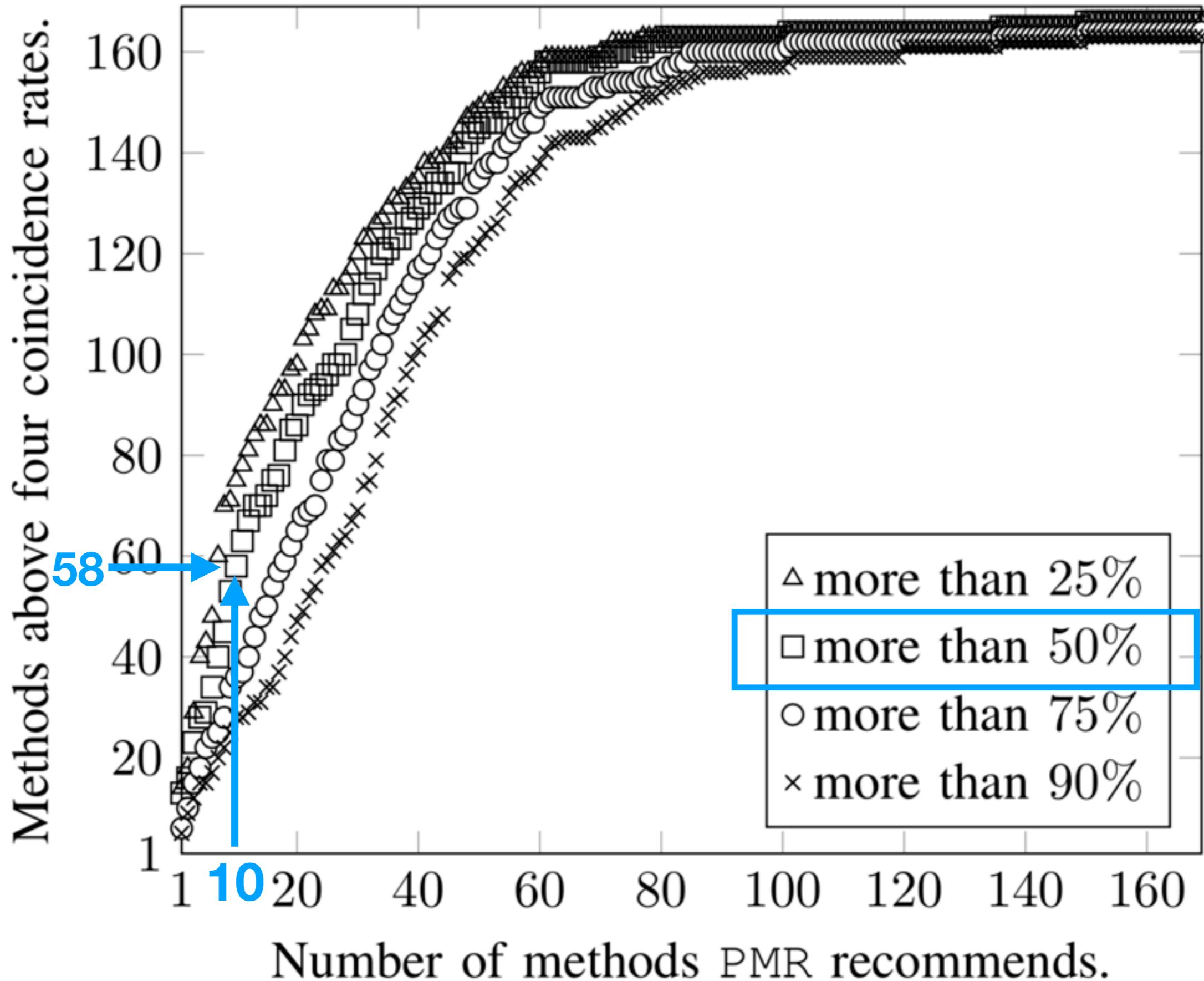




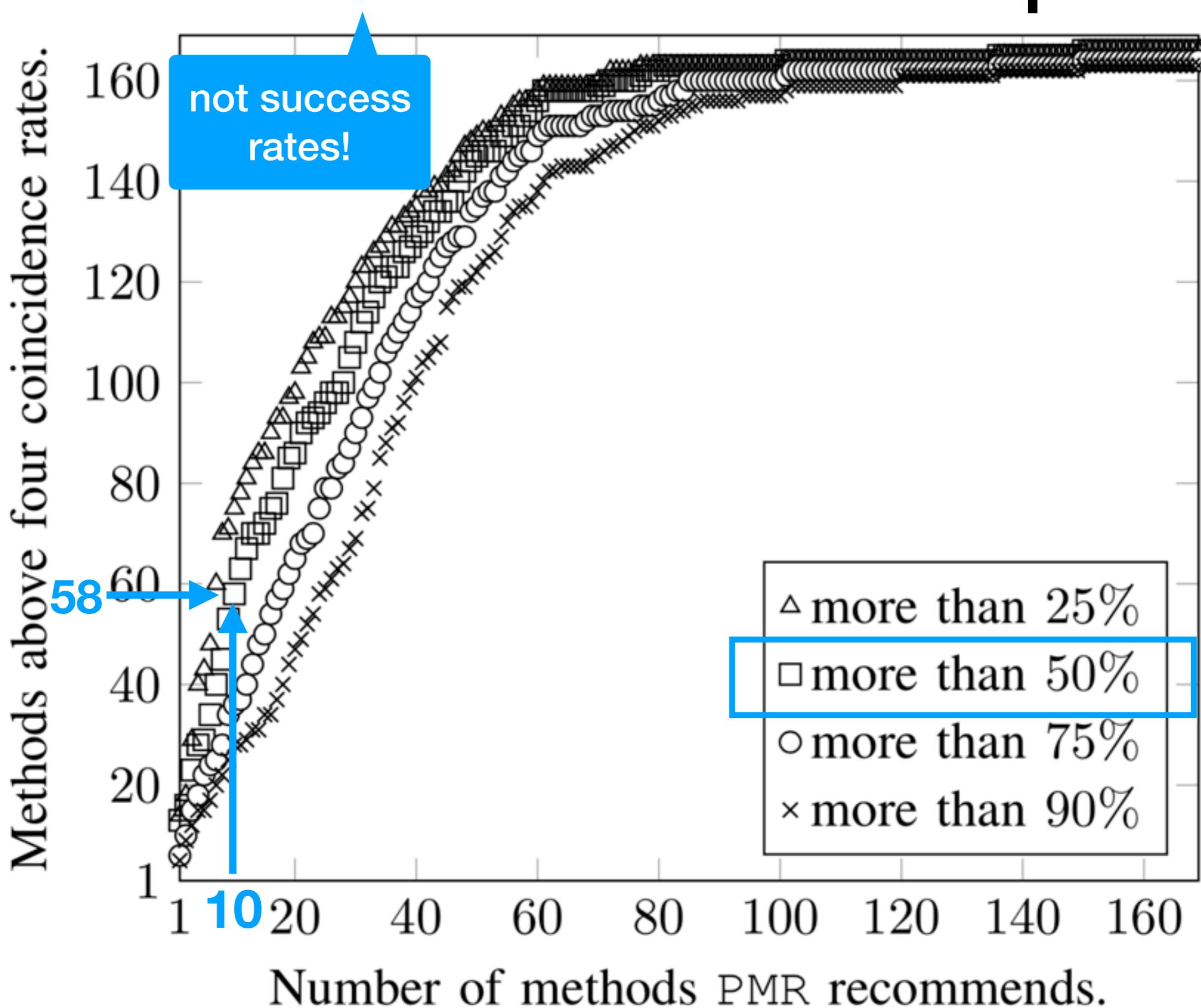
Coincidence rates of PaMpeR



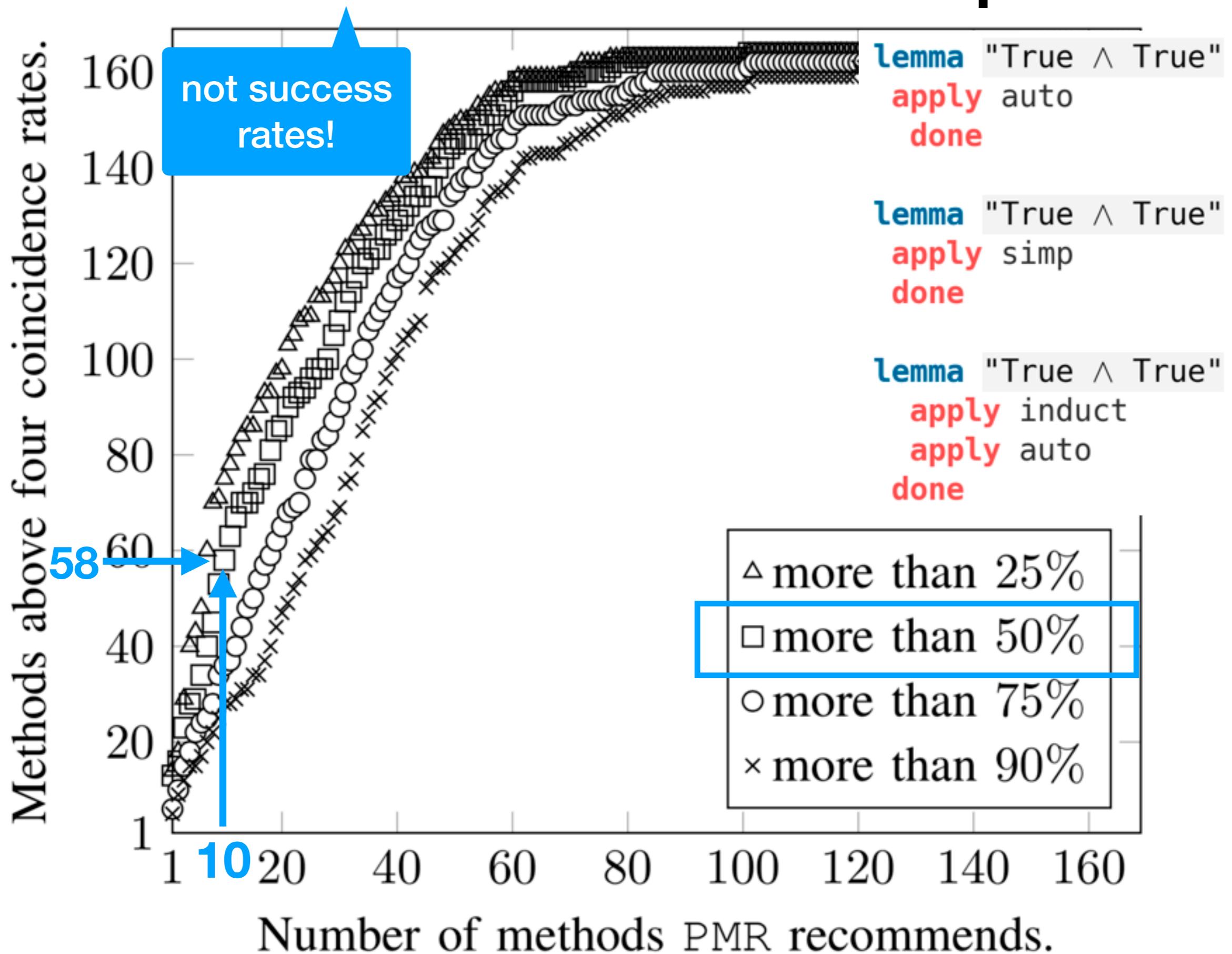
Coincidence rates of PaMpeR



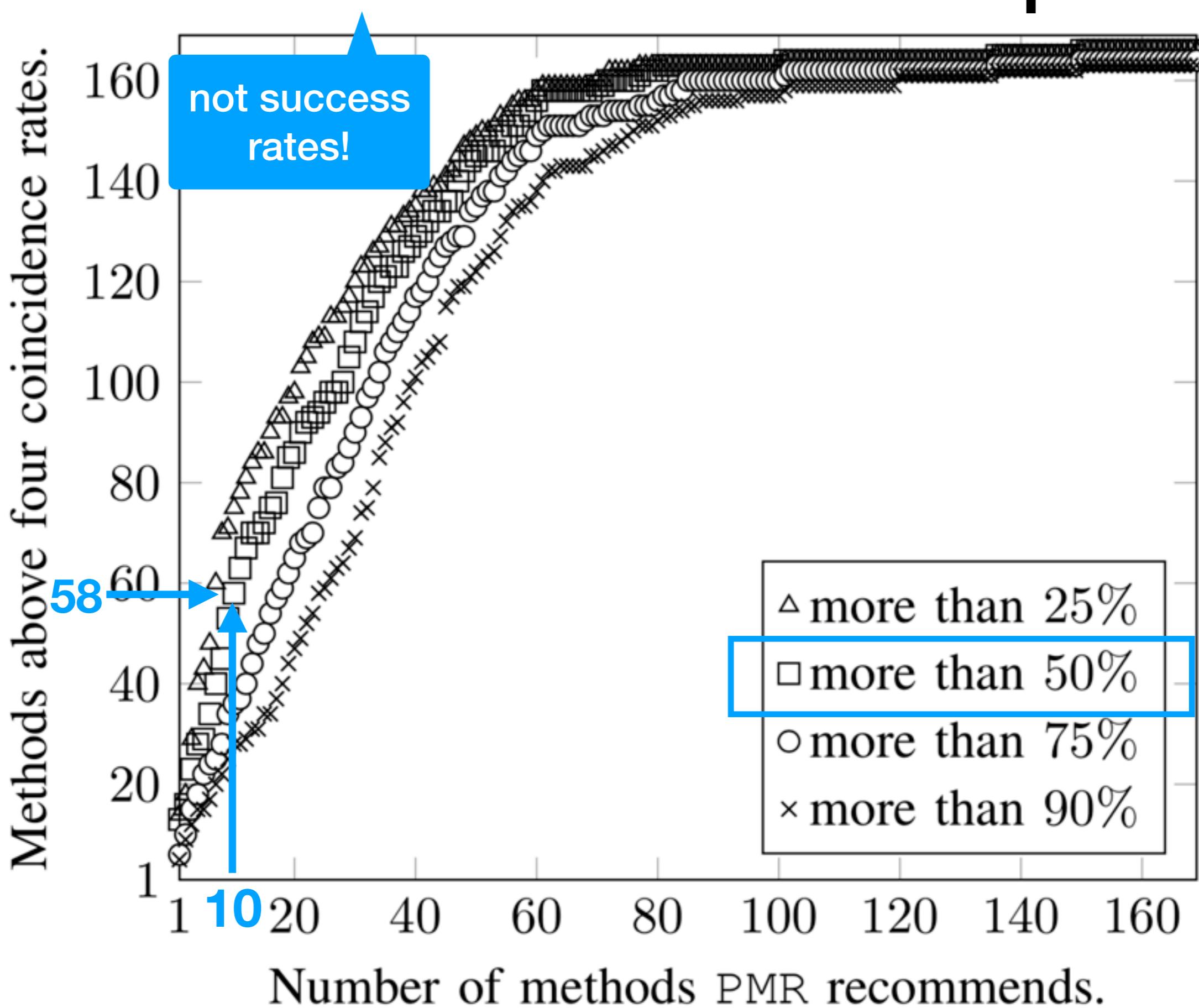
Coincidence rates of PaMpeR



Coincidence rates of PaMpeR



Coincidence rates of PaMpeR



Coincidence rates of PaMpeR

