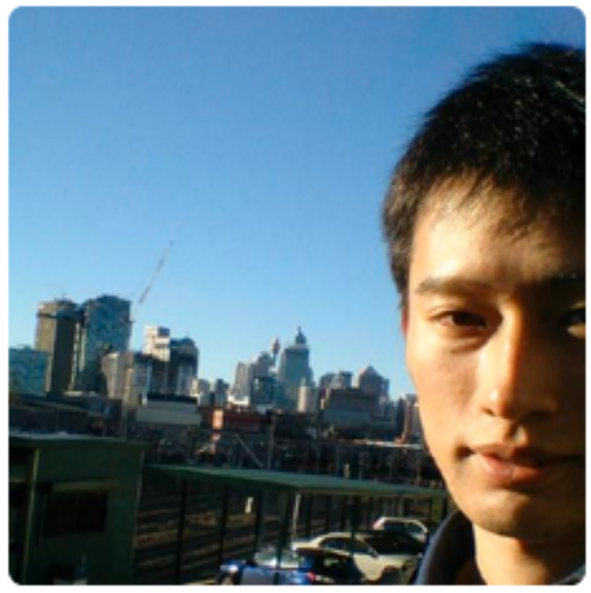
Towards United Reasoning for Automatic Induction in Isabelle/HOL

Yutaka Nagashima
email: yutaka.nagashima(@)cvut.cz
twitter: YutakangJ for Japanese
YutakangE for English



Yutaka Ng

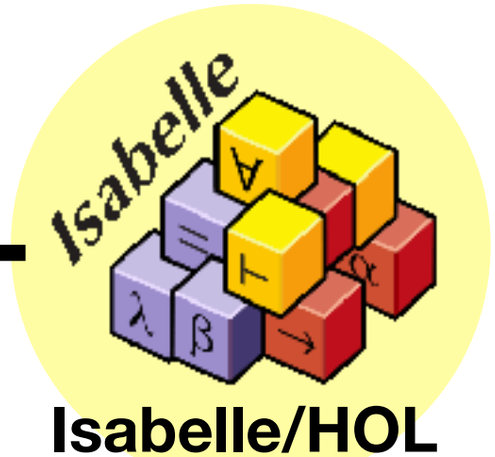
Why proof by induction?

developer **proof assistant / ITP** **implementation**

Why proof by induction?

developer		proof assistant / ITP		implementation
	+		=	
Gewin Klein et. al		Isabelle/HOL		verified micro kernel, seL4

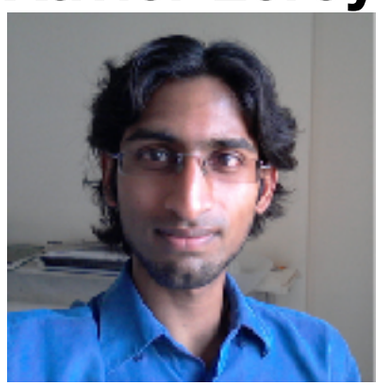
Why proof by induction?

developer		proof assistant / ITP		implementation
	+		=	
Gewin Klein et. al		Isabelle/HOL		verified micro kernel, seL4

Why proof by induction?

developer		proof assistant / ITP		implementation
	+	Isabelle	=	seL4 Security. Performance. Proof.
Gewin Klein et. al		Isabelle/HOL		verified micro kernel, seL4
	+		=	
Xavier Leroy		Coq		verified optimizing C compiler, CompCert

Why proof by induction?

developer		proof assistant / ITP		implementation
	+	Isabelle	=	seL4 Security. Performance. Proof.
Gewin Klein et. al		Isabelle/HOL		verified micro kernel, seL4
	+		=	
Xavier Leroy		Coq		verified optimizing C compiler, CompCert
	+		=	CAKEML A Verified Implementation of ML
Ramana Kumar et. al		HOL		verified implementation of ML, CakeML

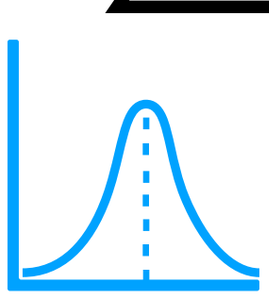
①

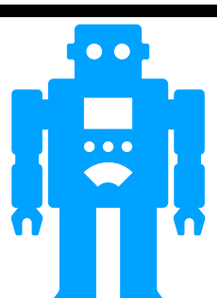
Isabelle's proof methods
(deductive reasoning)

united reasoning
for automatic induction

③

machine learning induction
(inductive reasoning)





②

goal-oriented conjecturing
(abductive reasoning)

①

Isabelle's proof methods
(deductive reasoning)

united reasoning
for automatic induction

③

machine learning induction
(inductive reasoning)

②

goal-oriented conjecturing
(abductive reasoning)

DEMO1

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

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

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

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

Output | Query | Sledgehammer | Symbols

DEMO1

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

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

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

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

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

Output | Query | Sledgehammer | Symbols

DEMO1

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

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

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



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

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

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

Output | Query | Sledgehammer | Symbols

17,44 (304/1012) | Input/output complete | (isabelle,isabelle,UTF-8-Isabelle) | 535MB 1:17 AM

DEMO1

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

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

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

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

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

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

Output | Query | Sledgehammer | Symbols

14,58 (310/1067) (isabelle,isabelle,UTF-8-Isabelle) 1:20 AM

DEMO1

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

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

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

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

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

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

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

Output | Query | Sledgehammer | Symbols

17,44 (361/1084) (isabelle,isabelle,UTF-8-Isabelle) 242/535MB 1:05 AM

DEMO1


Example.thy (~/Workplace/PSL/PGT/)
fun sep::"'a ⇒ 'a list ⇒ 'a list" where
"sep a []   = []" |
"sep a [x]   = [x]" |
"sep a (x#y#zs) = x # a # sep a (y#zs)"
value "sep (1::int) [0,0,0]"
strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved]
lemma
"map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd
Proof state
Auto update
Update
Search:
100%
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)
535MB
1:05 AM

DEMO1

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

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

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

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

lemma
  "map f (sep x xs) = sep (f x) (map f xs)"
  find_proof DInd
apply (induct xs rule: Example.sep.induct)
apply auto
done
```

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

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

Output | Query | Sledgehammer | Symbols

20,11 (433/1147) (isabelle,isabelle,UTF-8-Isabelle)

DEMO1

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

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

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

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

lemma
  "map f (sep x xs) = sep (f x) (map f xs)"
  find_proof DInd
apply (induct xs rule: Example.sep.induct)
apply auto
done
```

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

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

What happened?

Output | Query | Sledgehammer | Symbols

20,11 (433/1147) (isabelle,isabelle,UTF-8-Isabelle) 255/535MB 1:05 AM

DEMO1

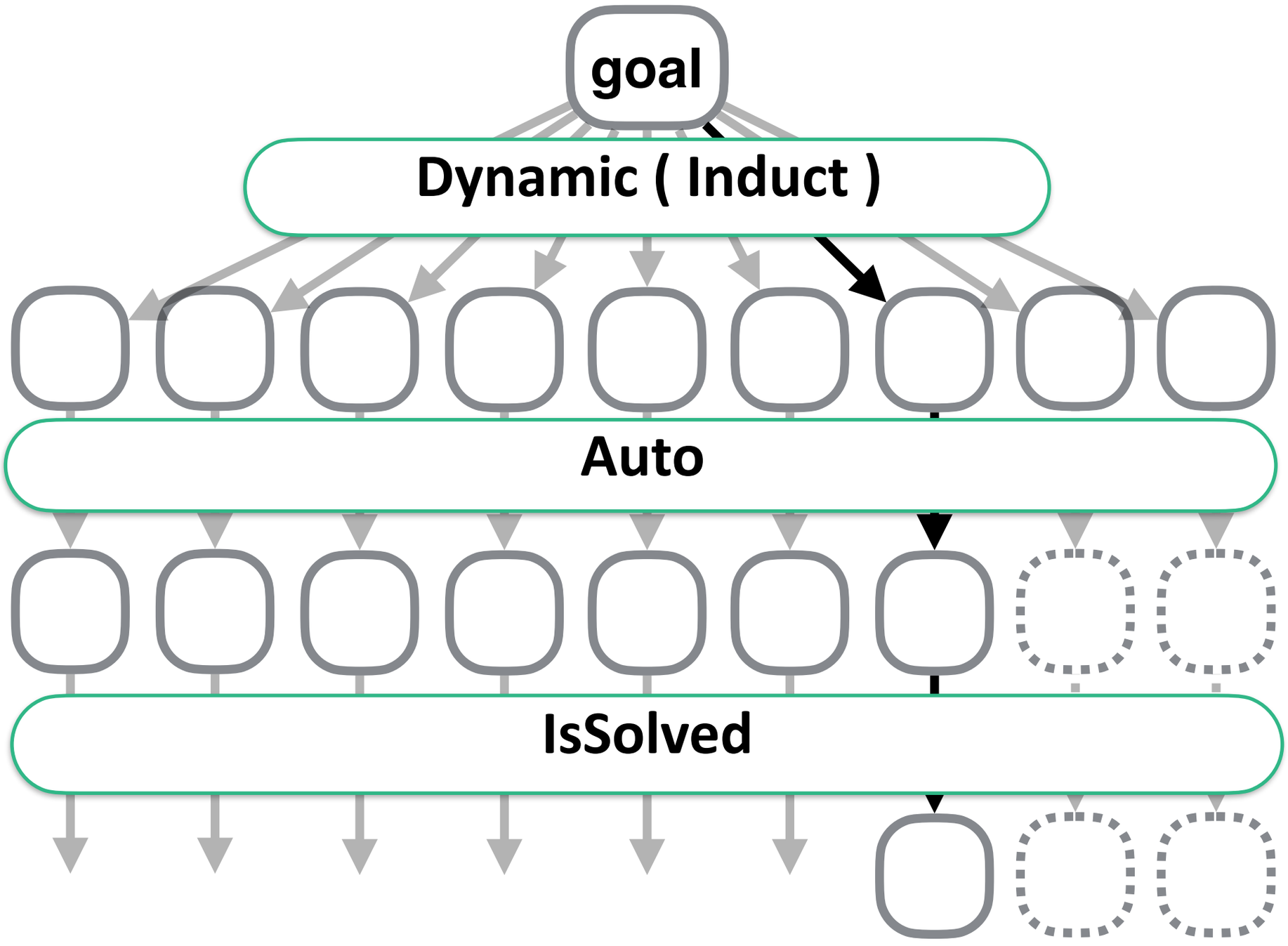
PSL: Proof Strategy Language

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

DEMO1

PSL: Proof Strategy Language

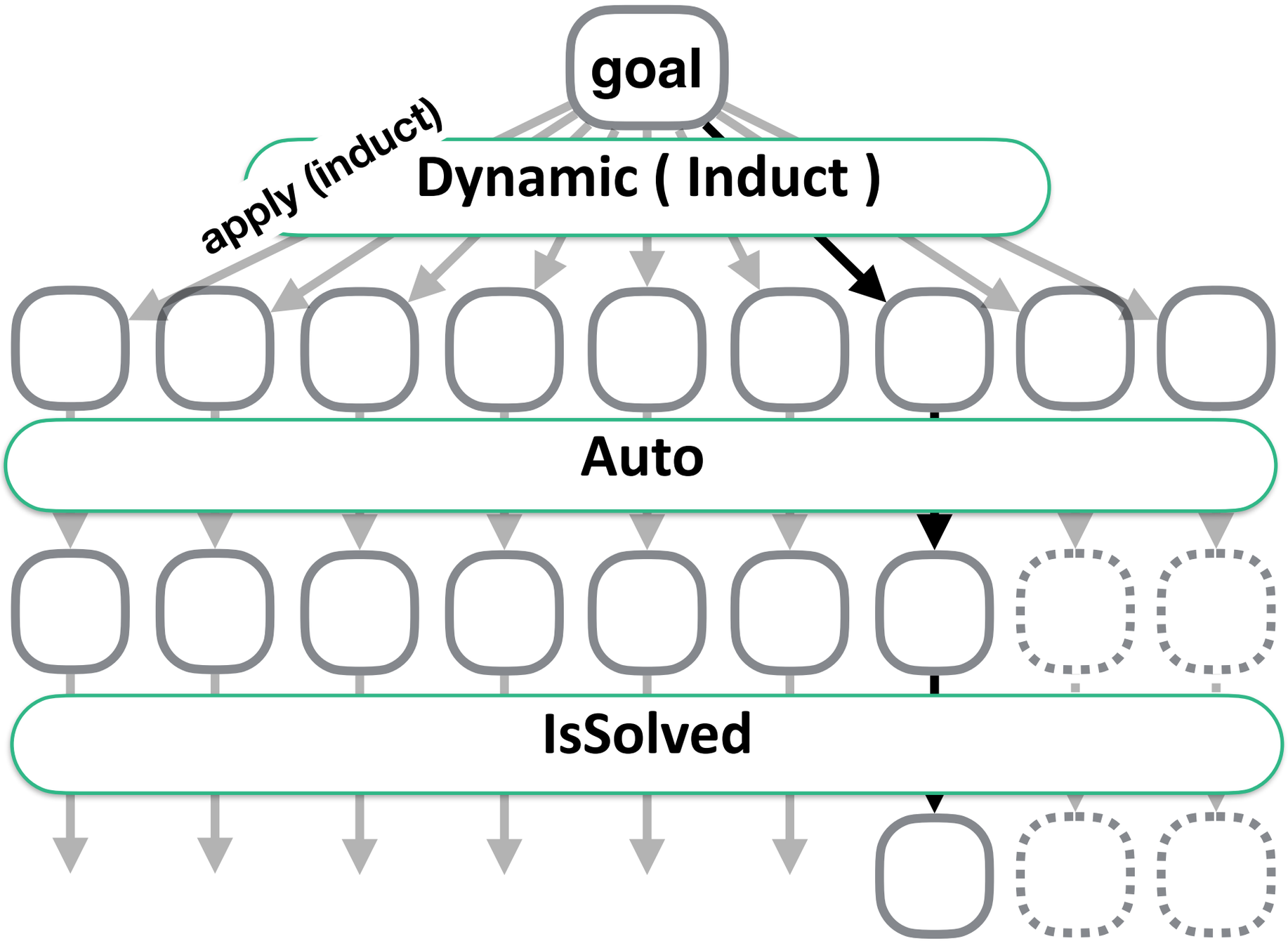
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

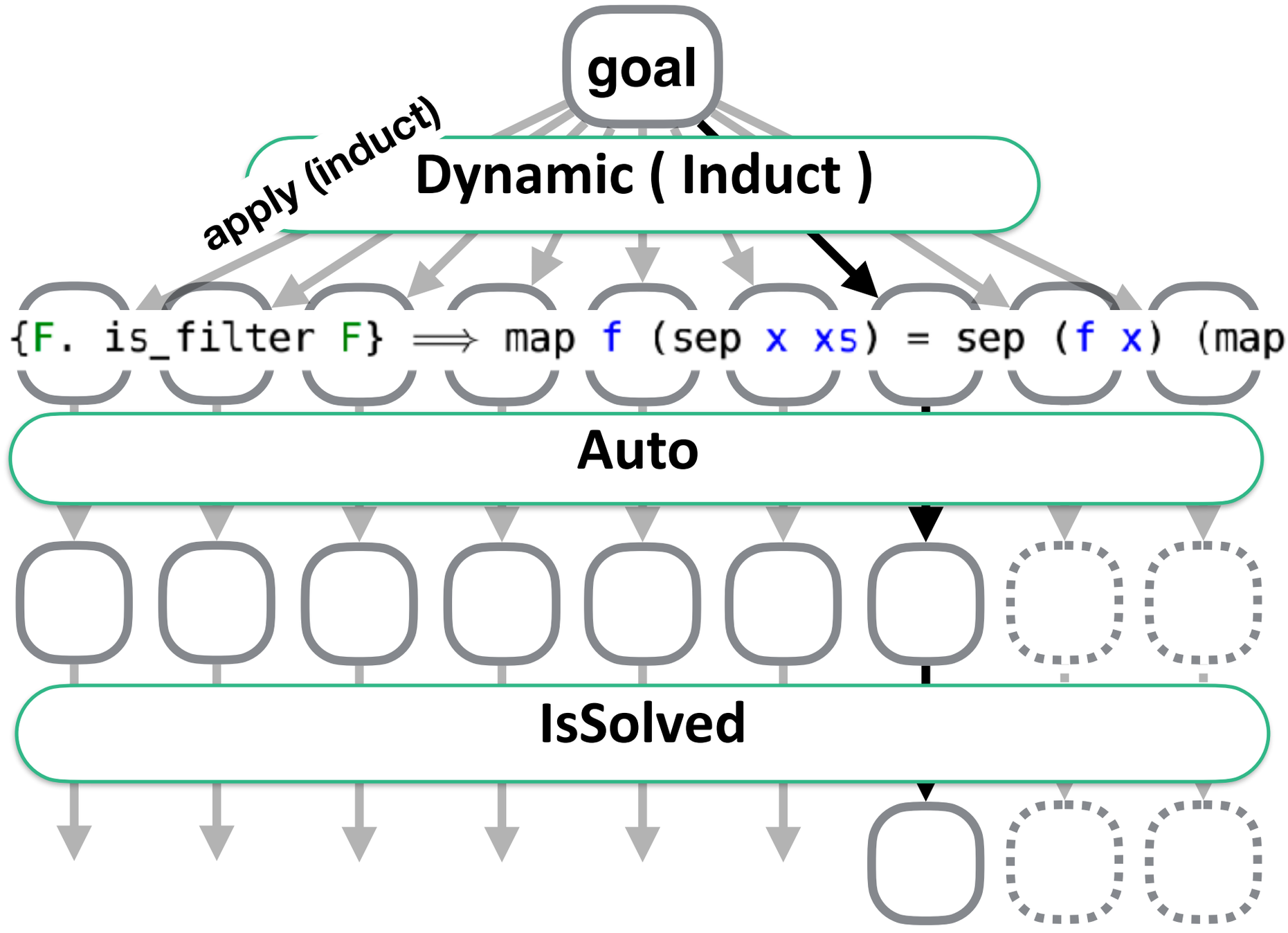
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

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

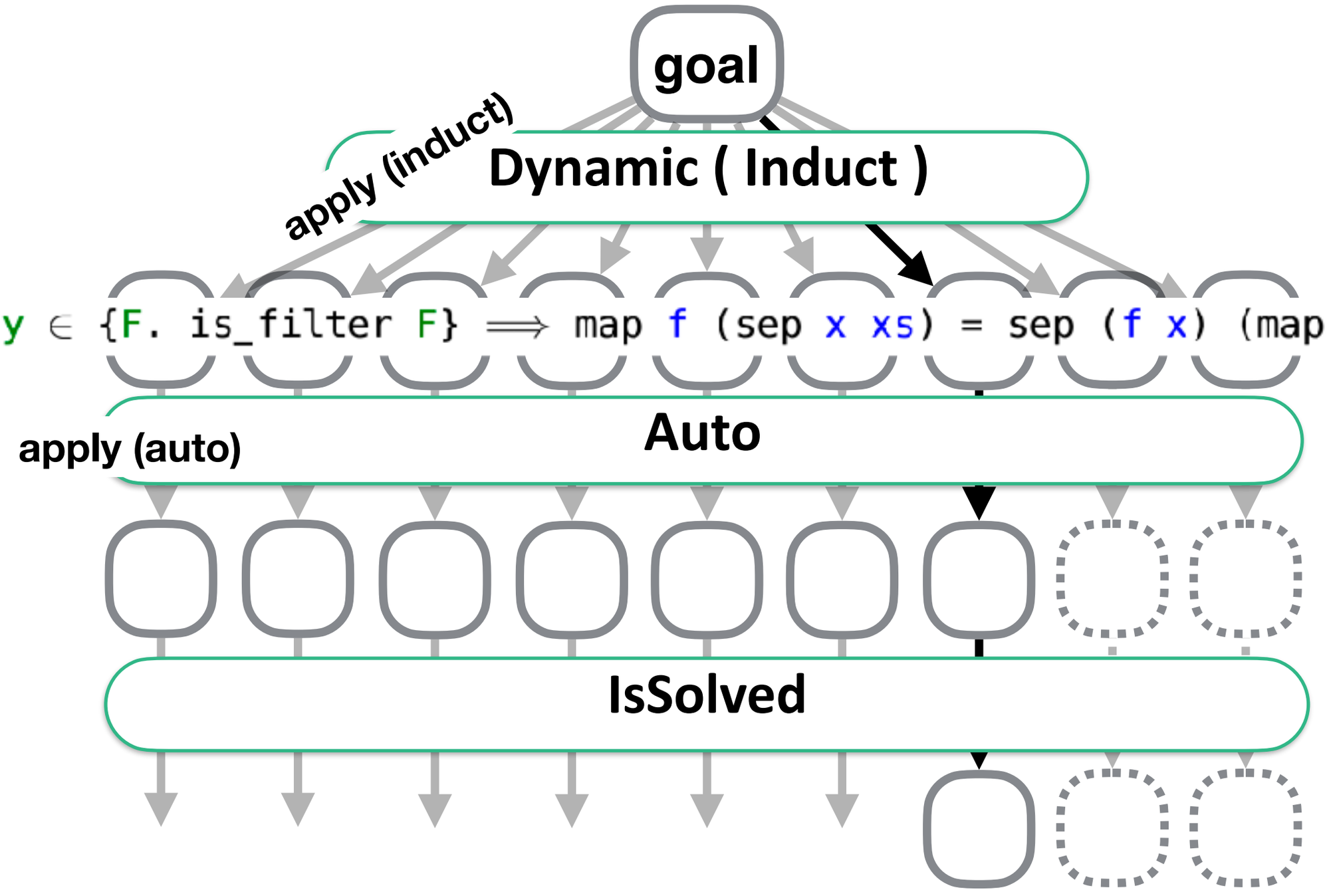



```
⋀y. y ∈ {F. is_filter F} ⟹ map f (sep x xs) = sep (f x) (map f xs)
```

DEMO1

PSL: Proof Strategy Language

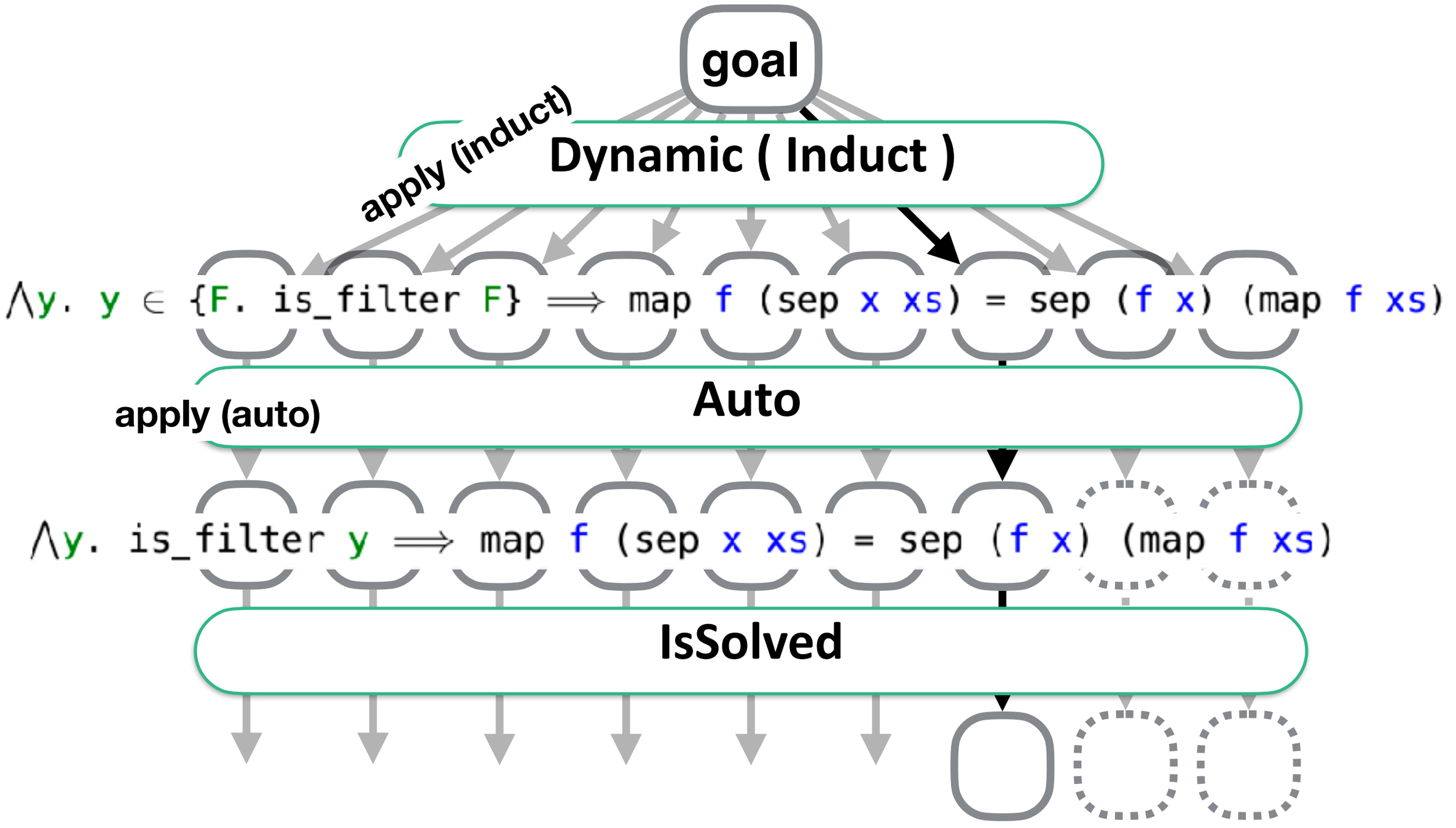
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

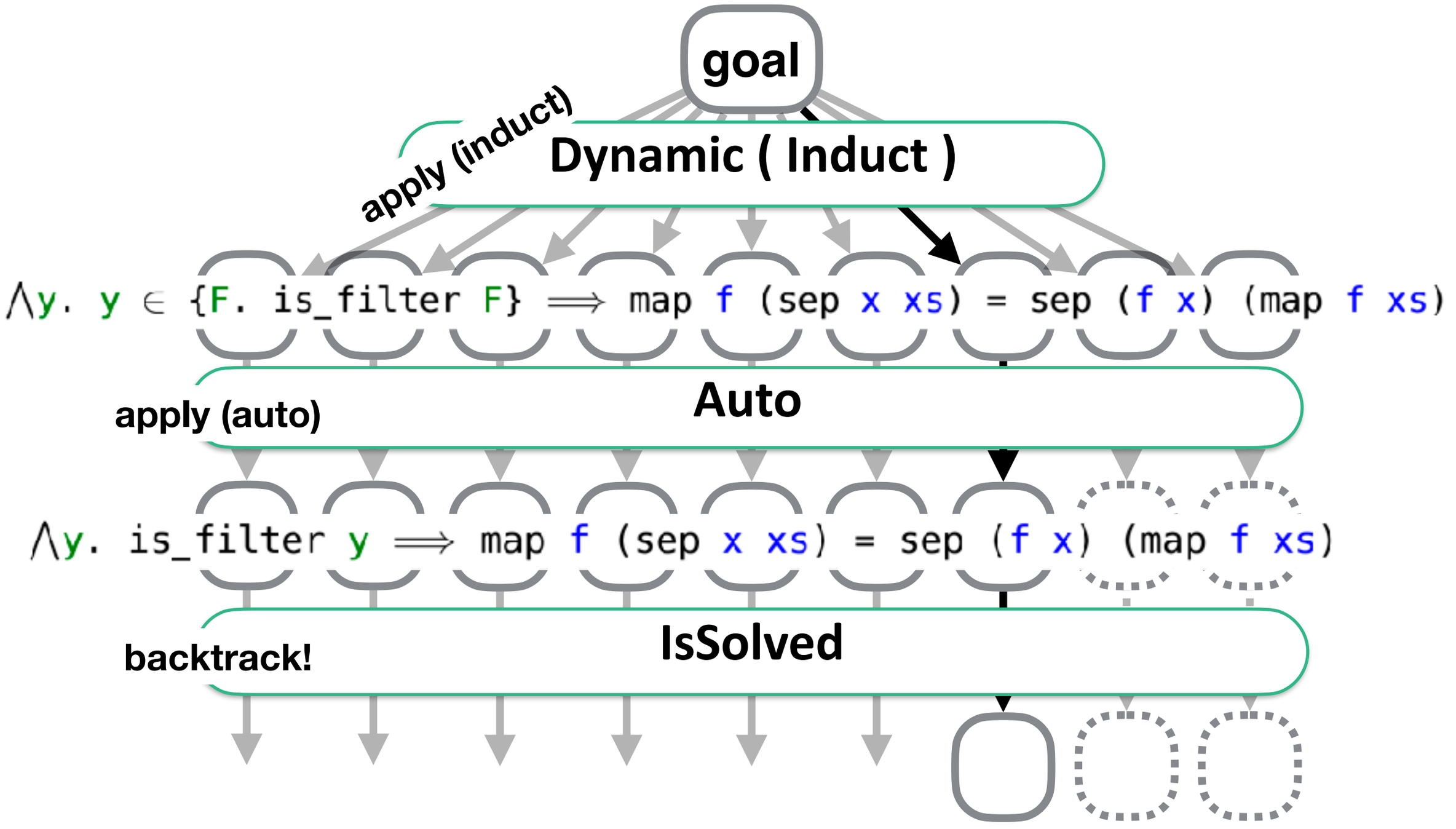
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

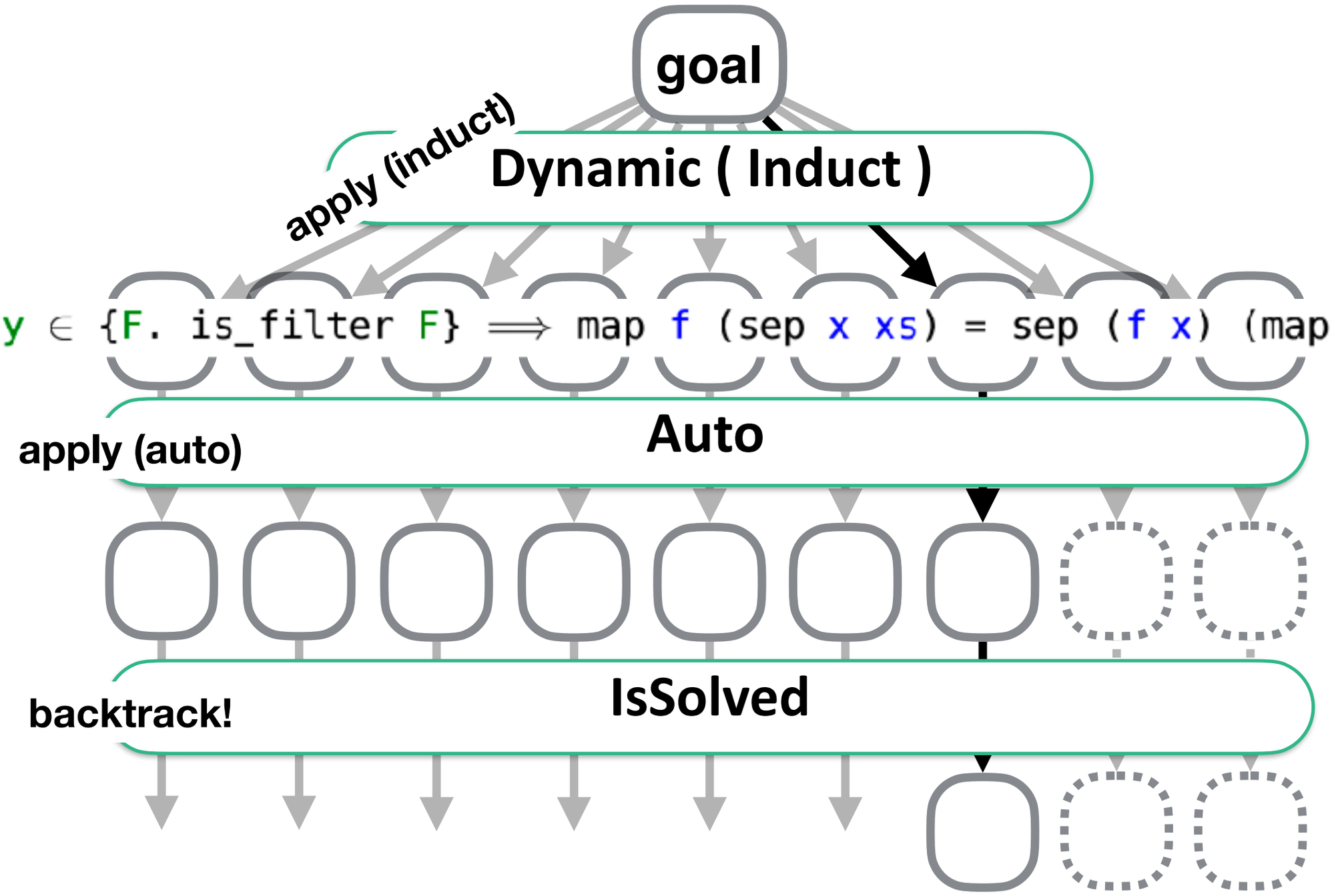
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

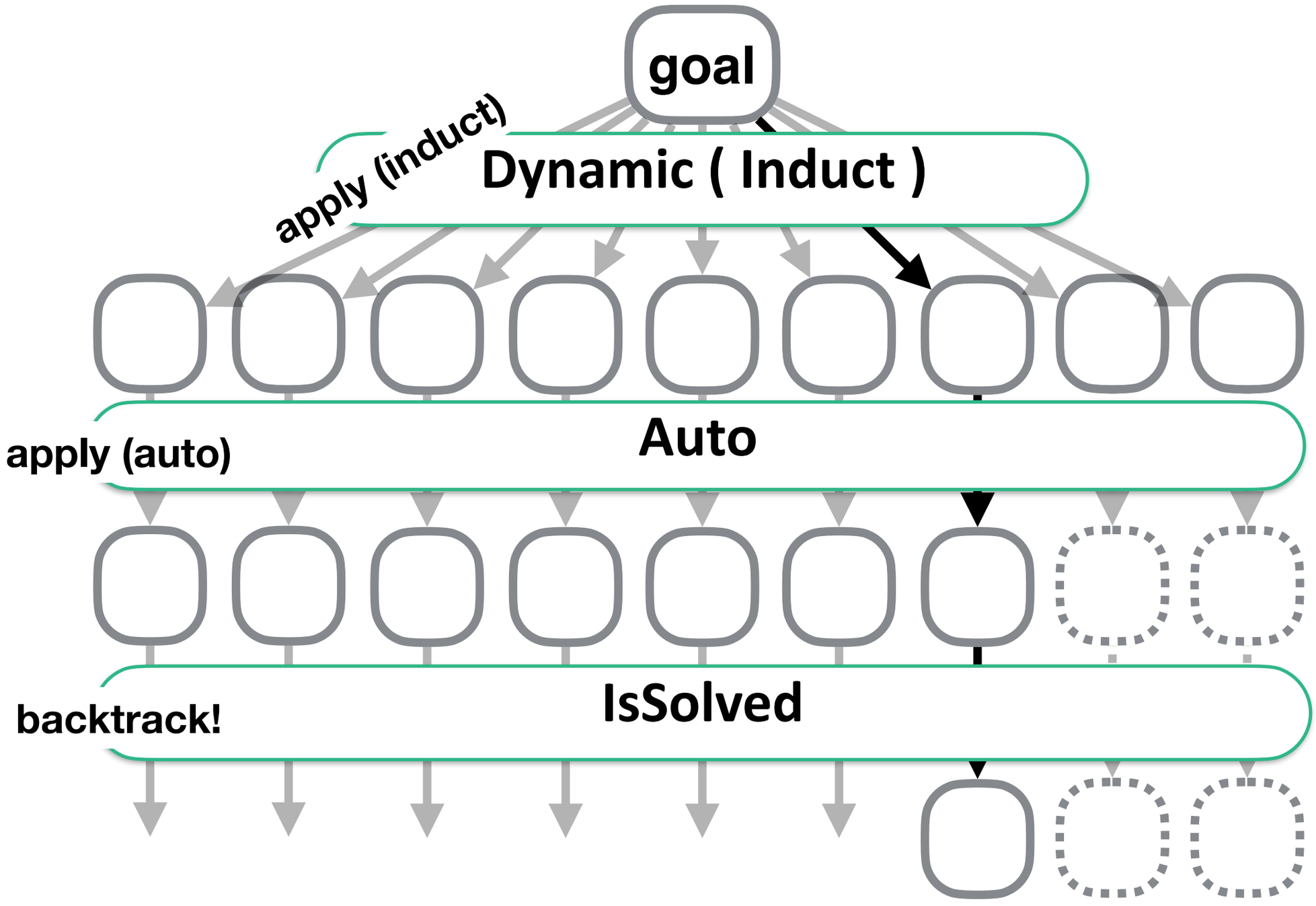
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

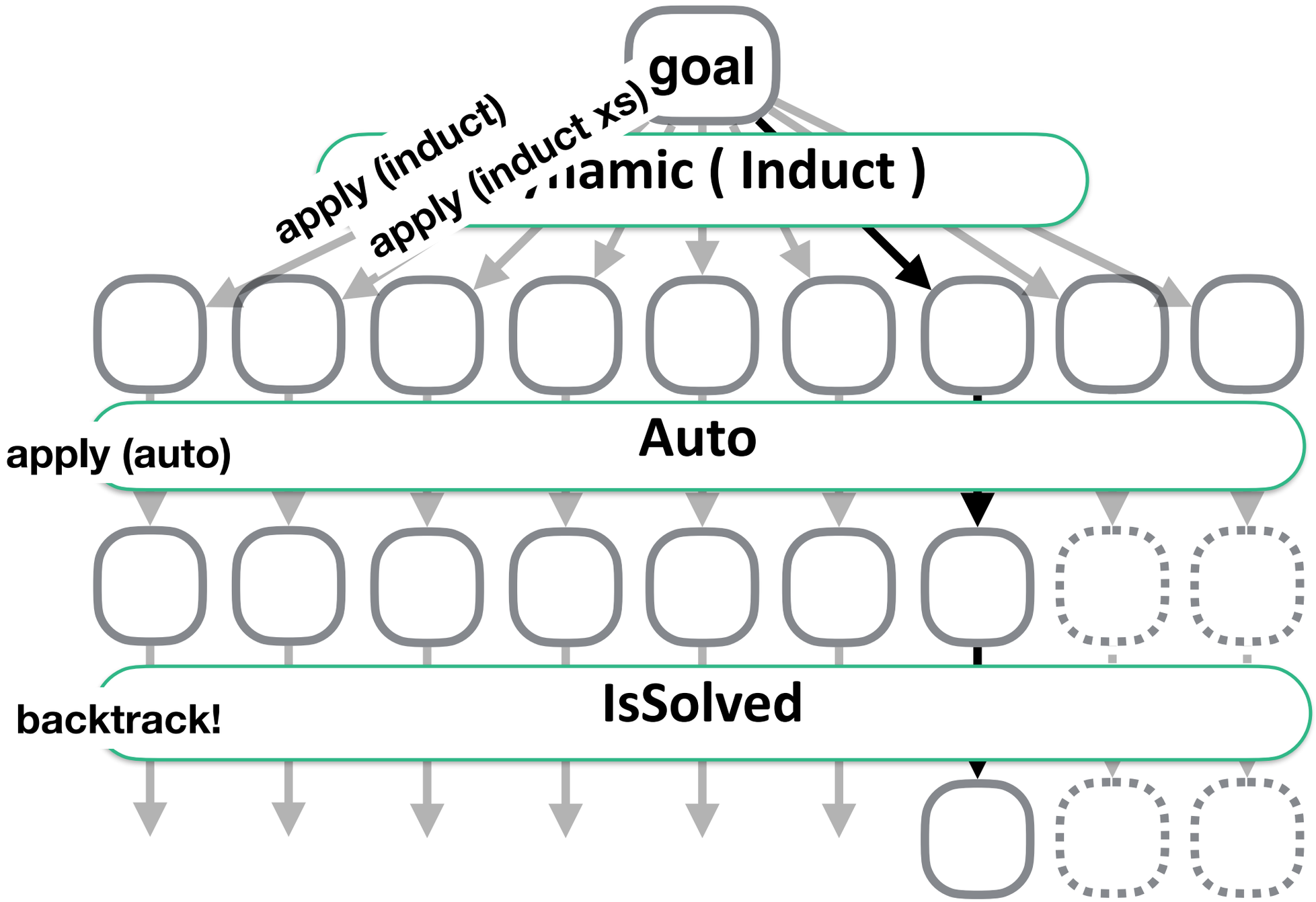
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

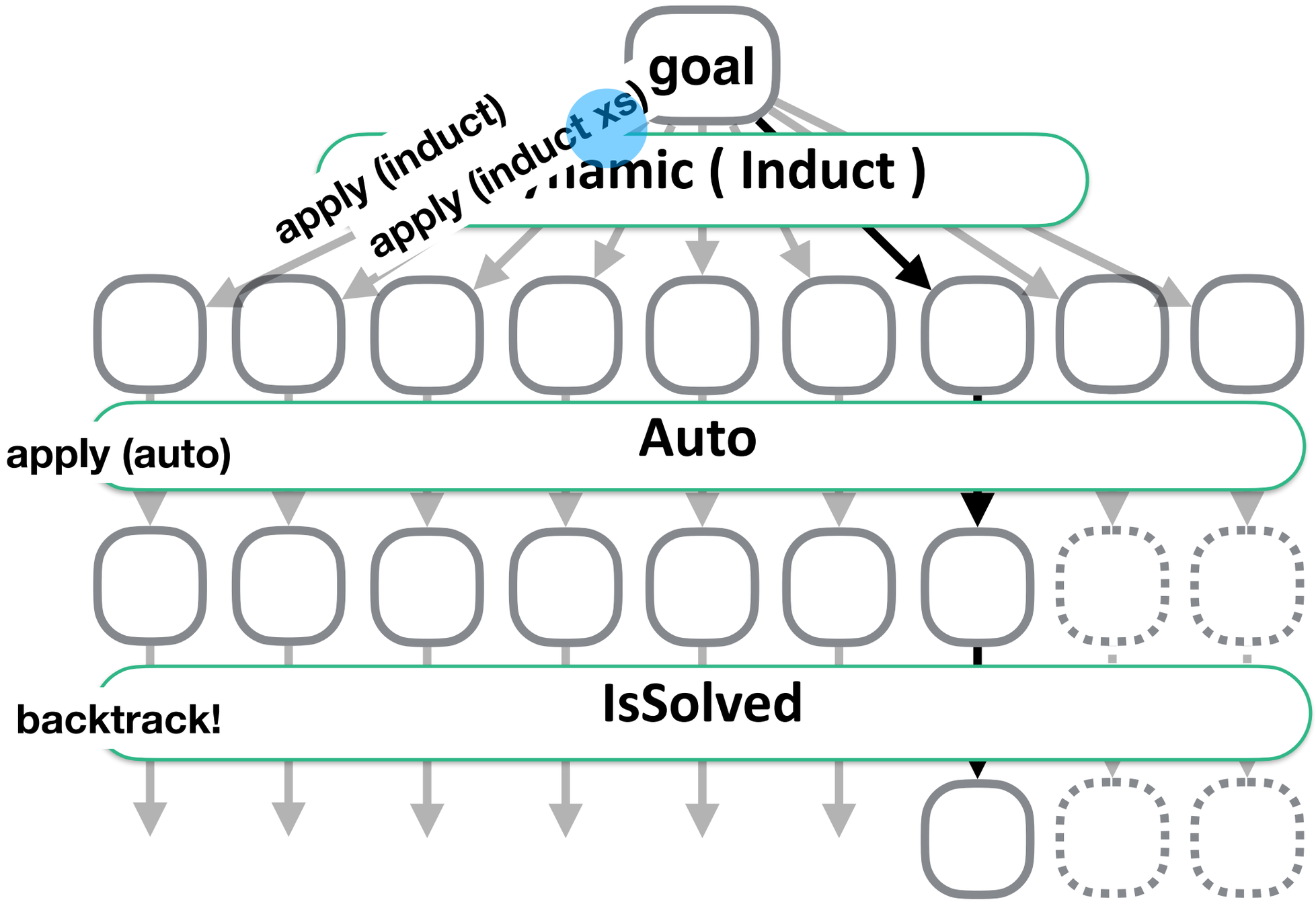
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

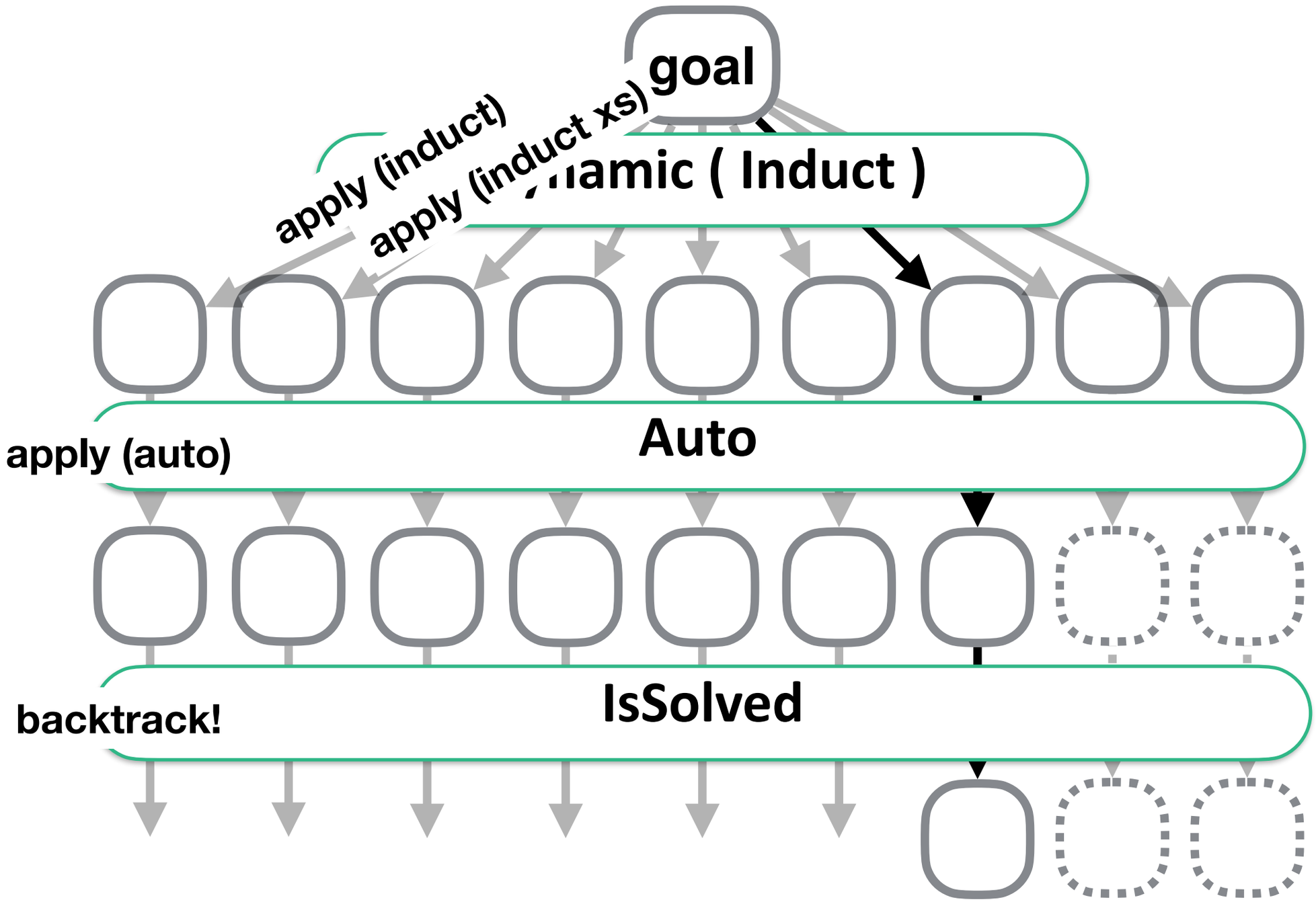
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

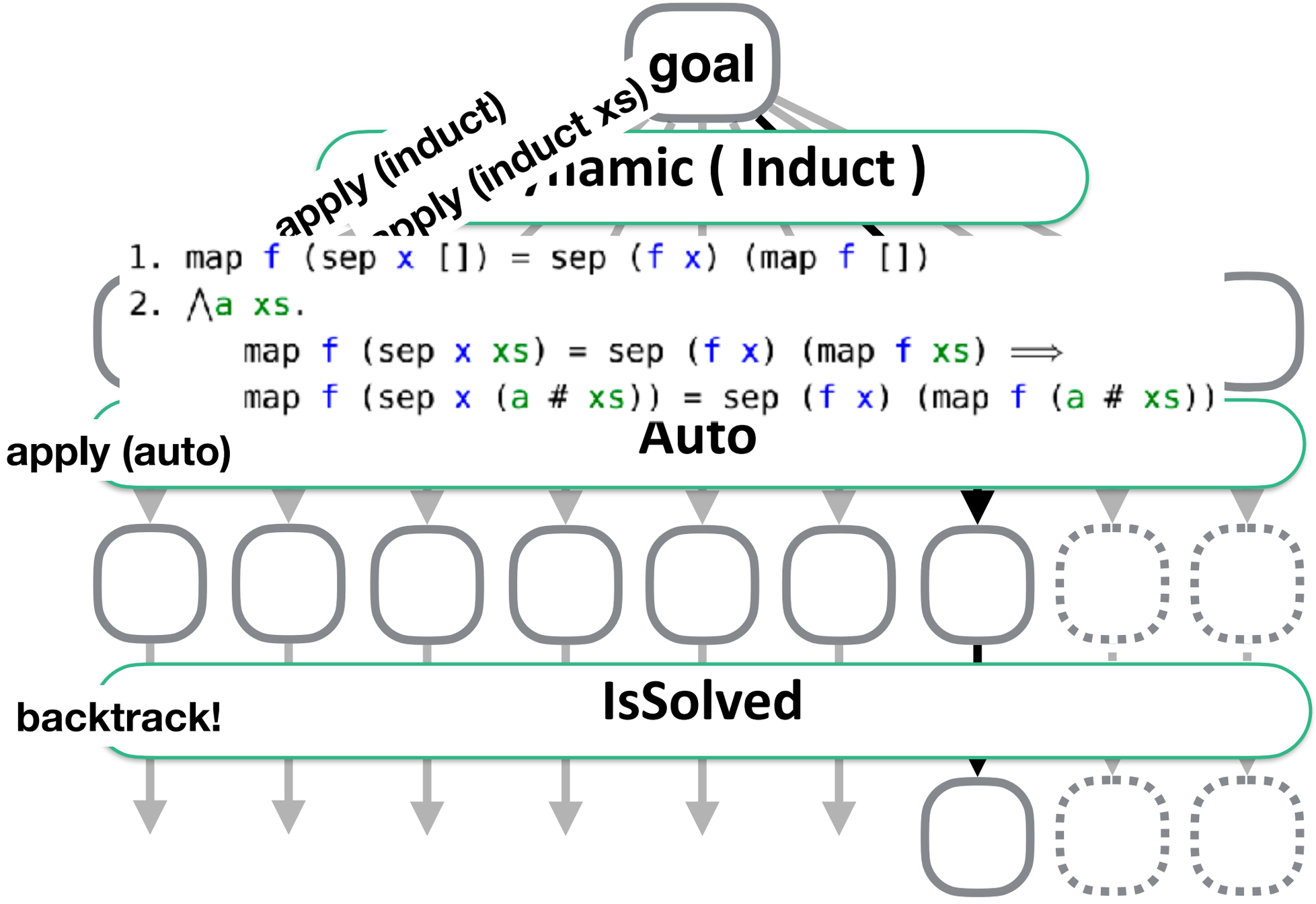
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

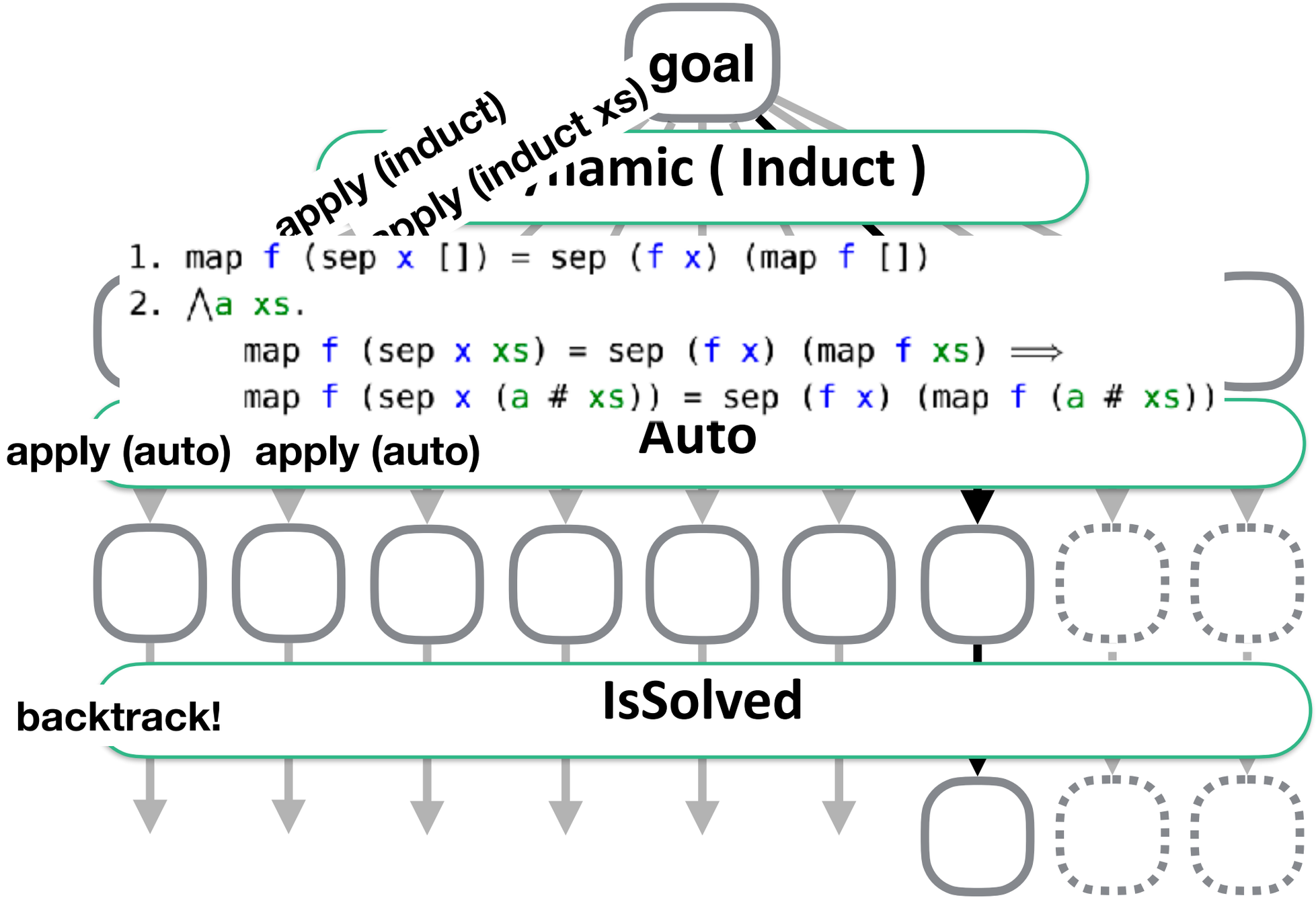
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

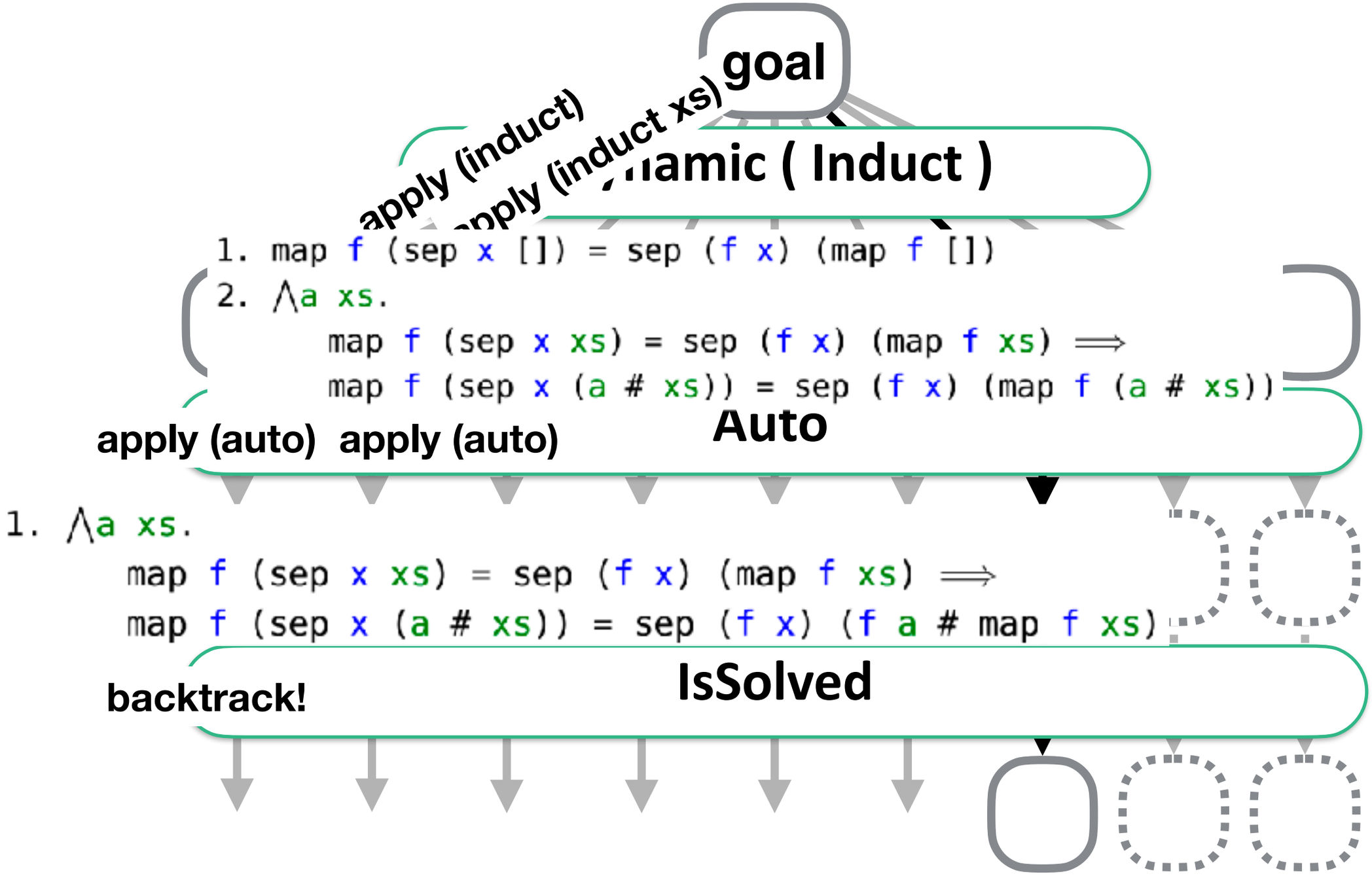
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

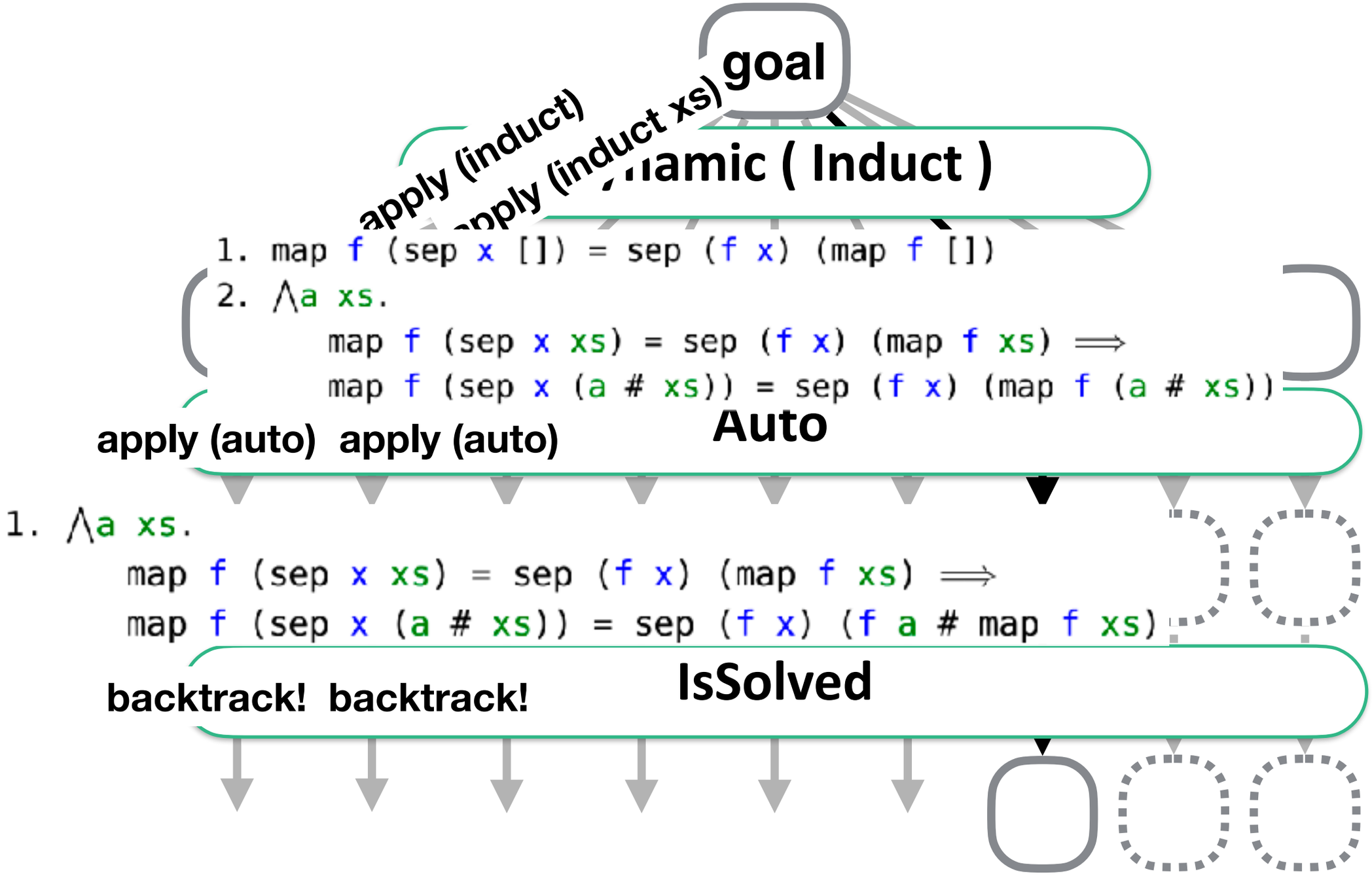
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

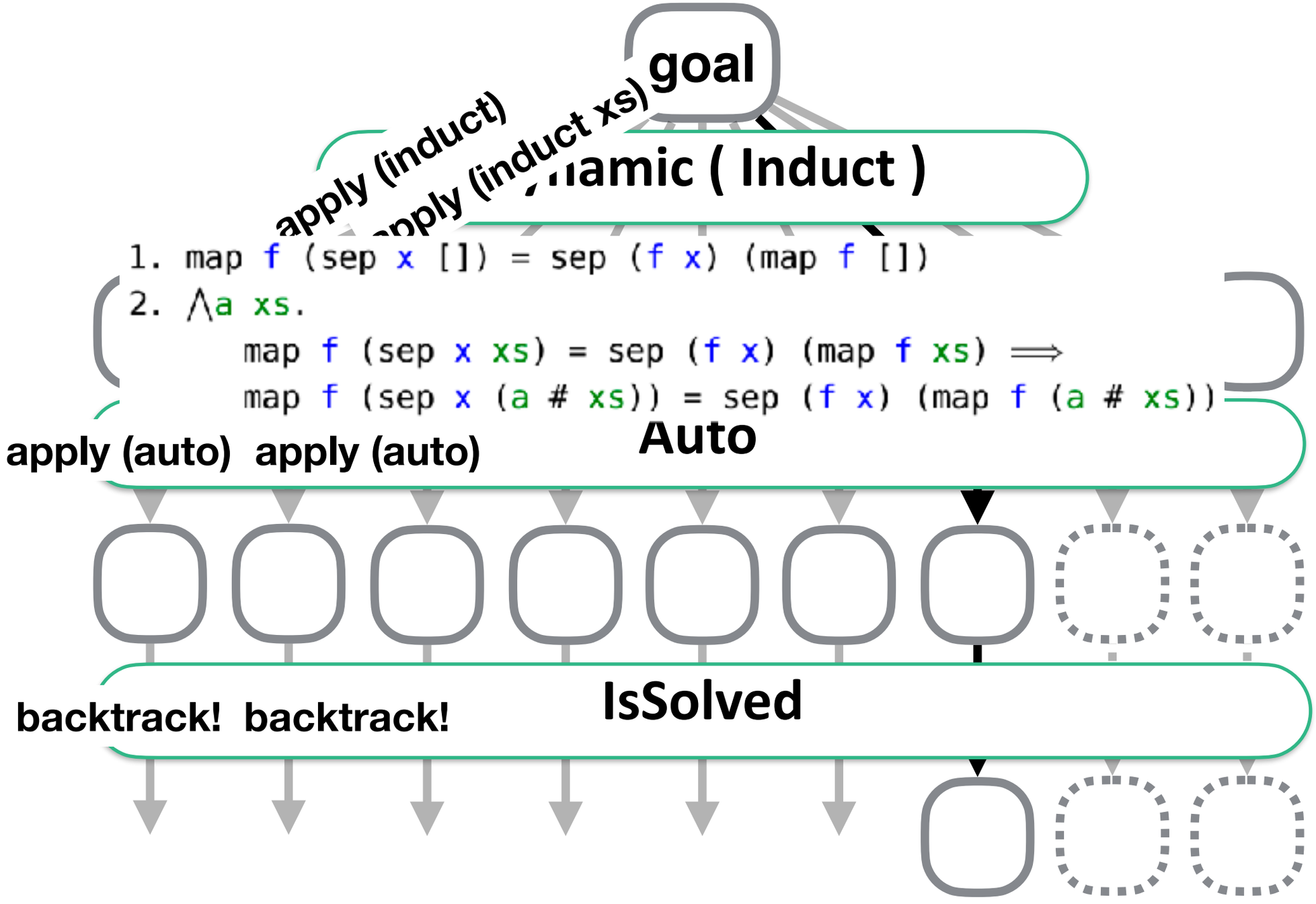
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

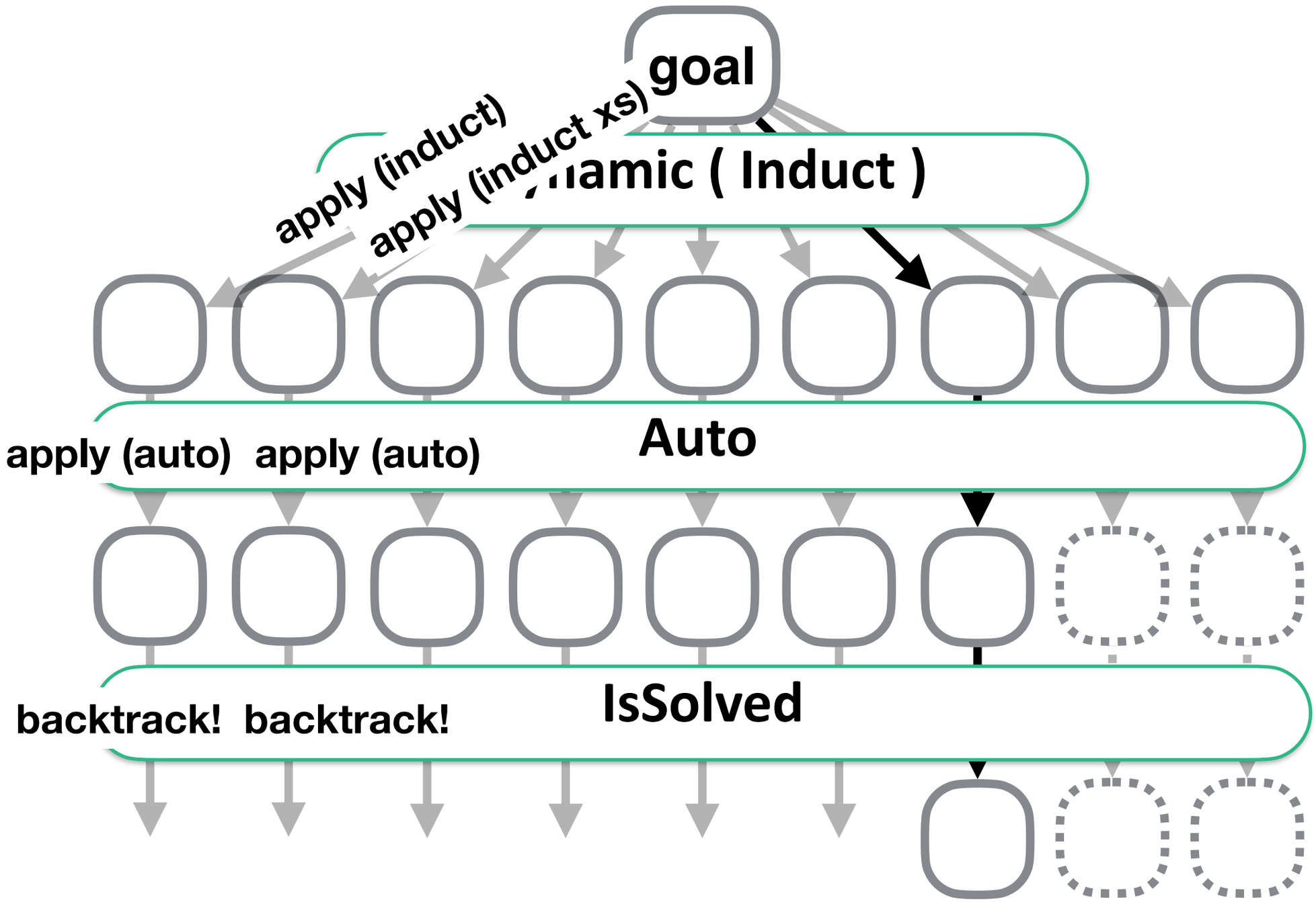
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

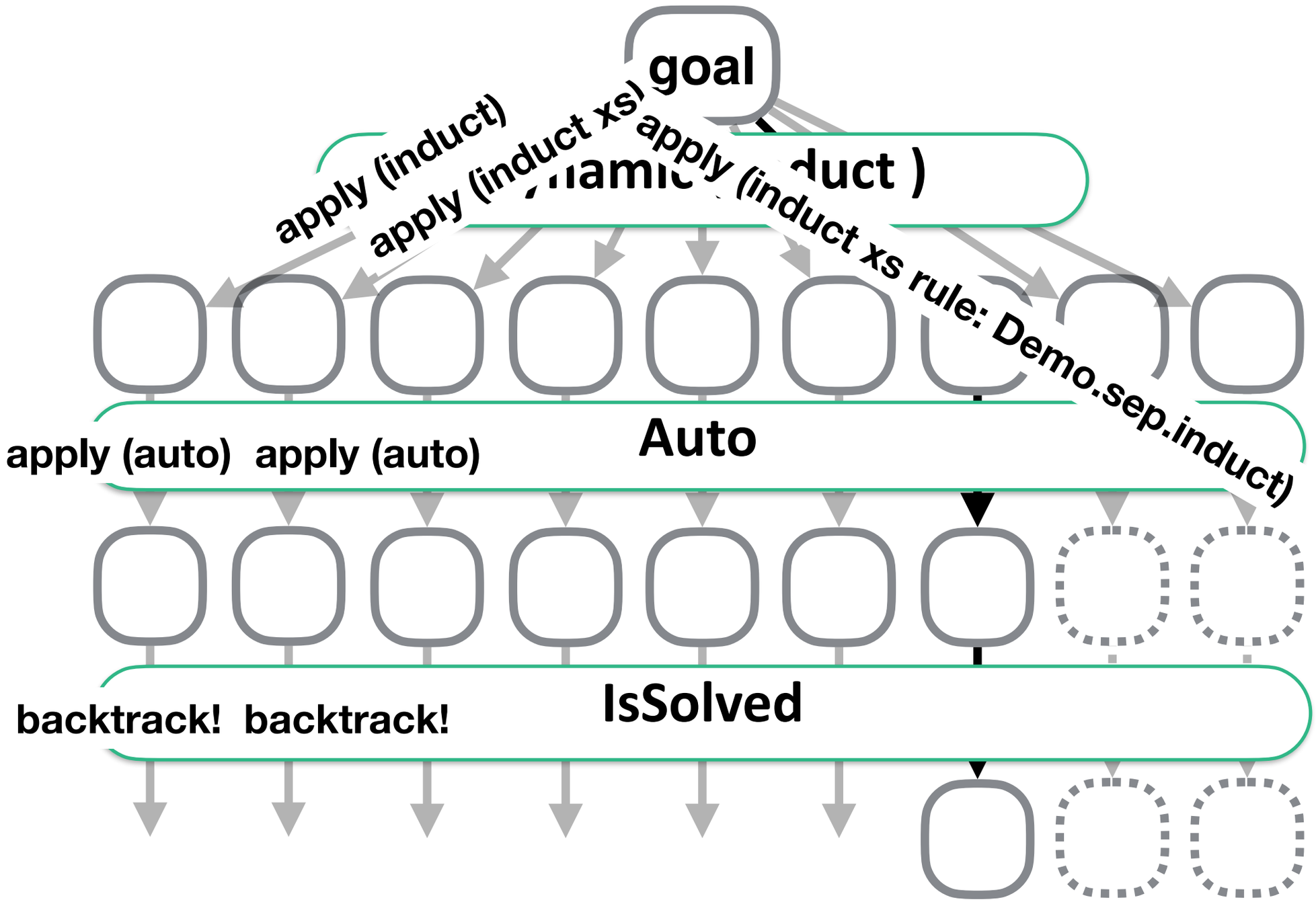
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

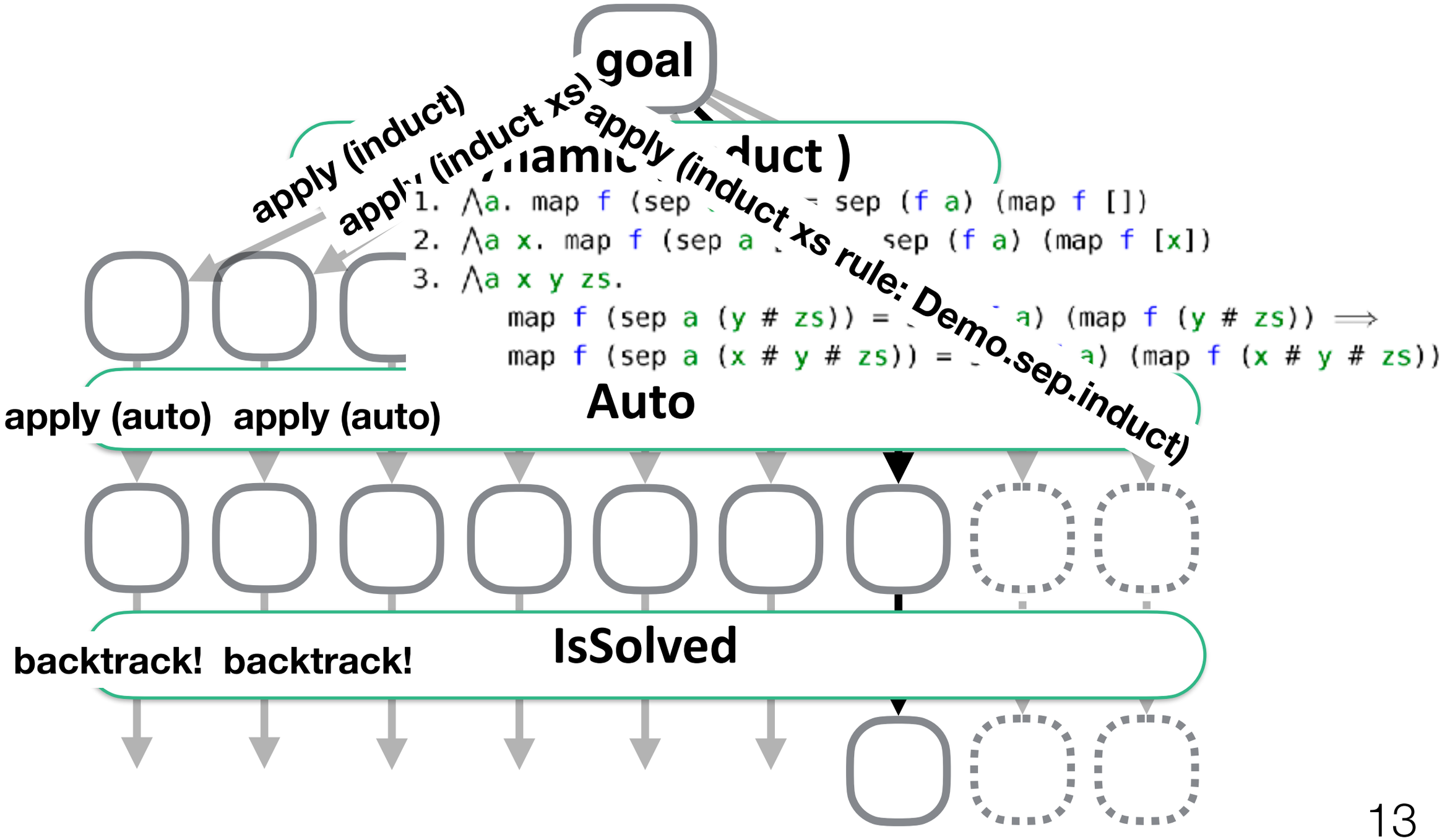
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

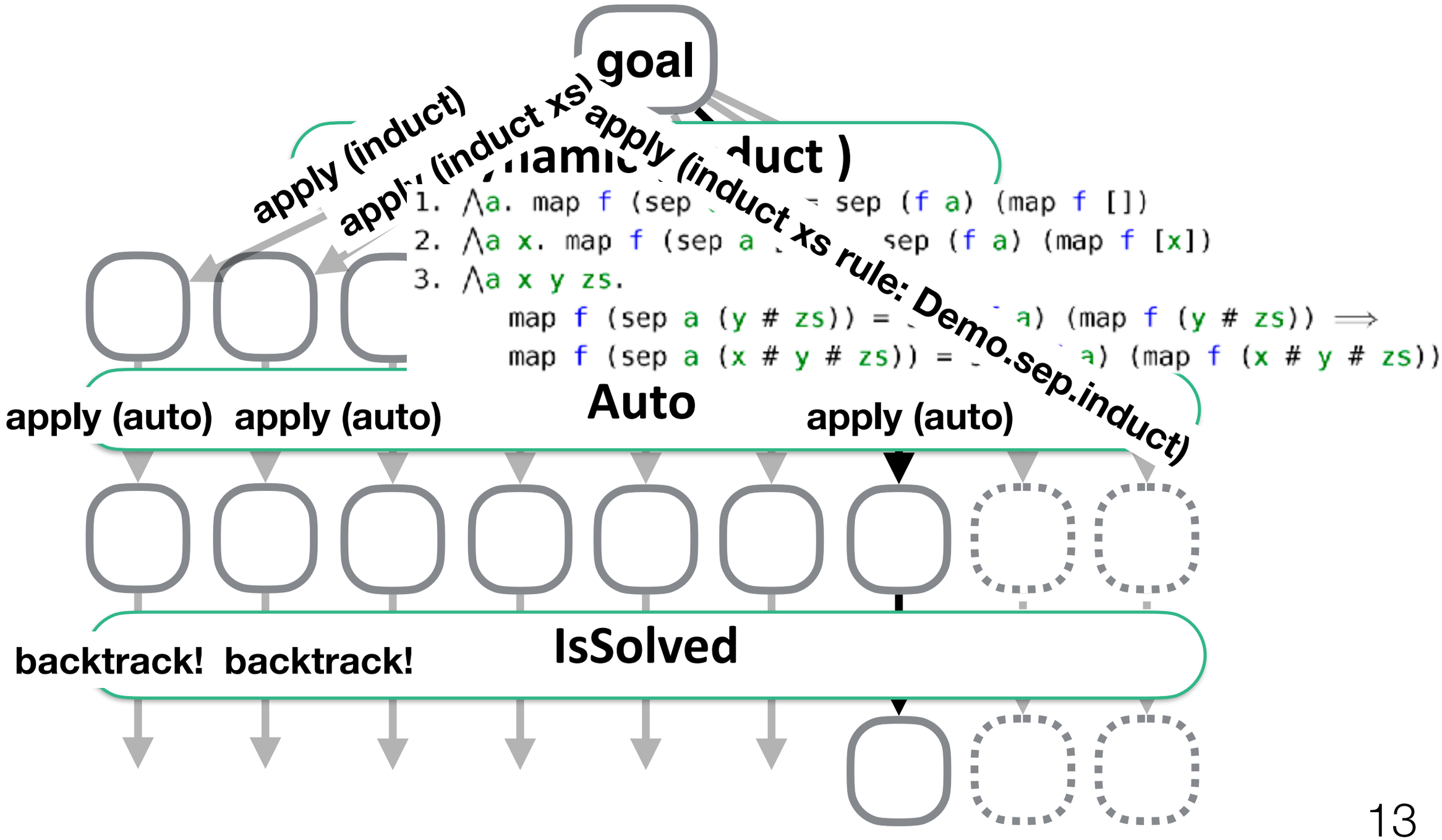
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

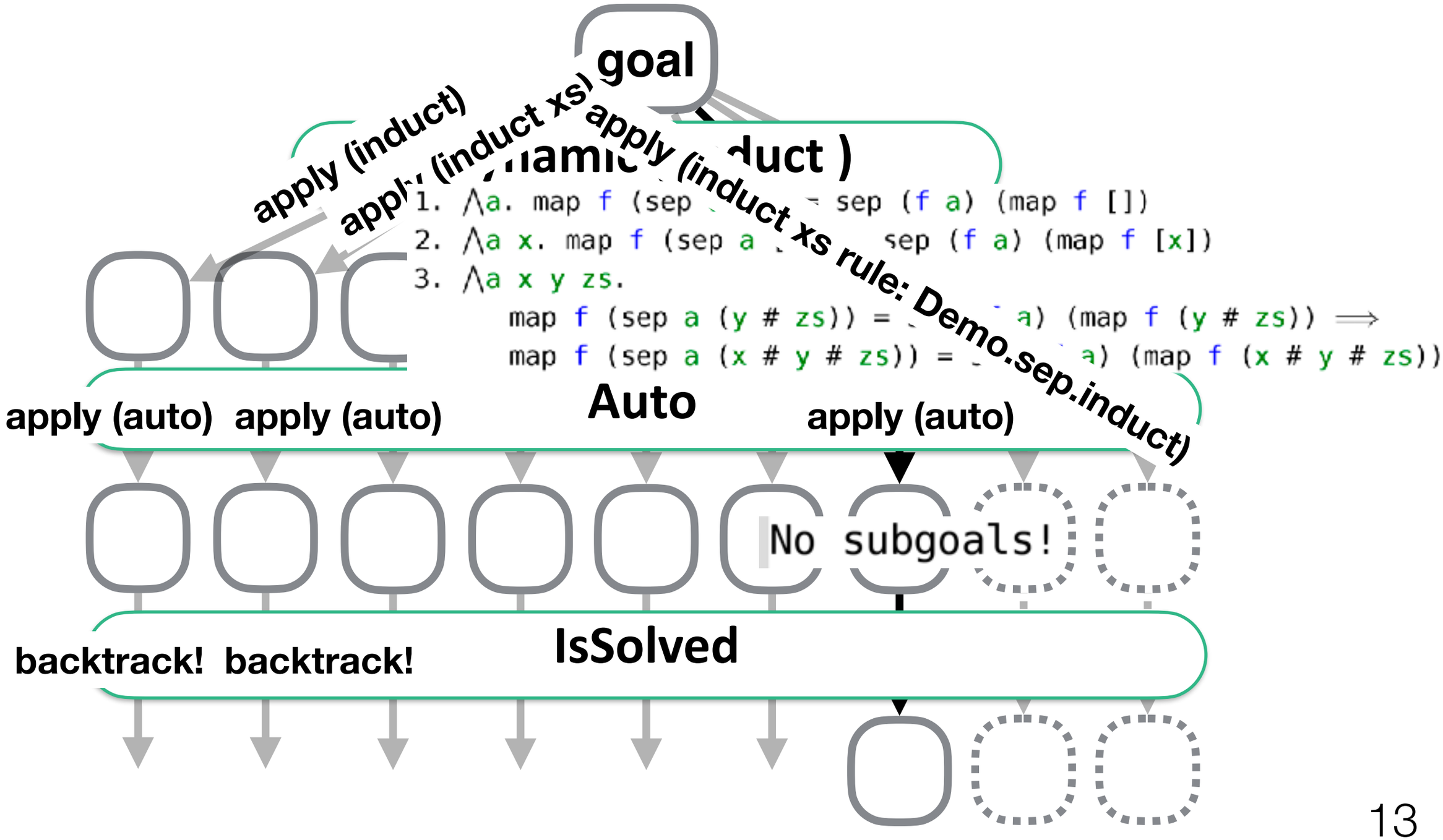
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

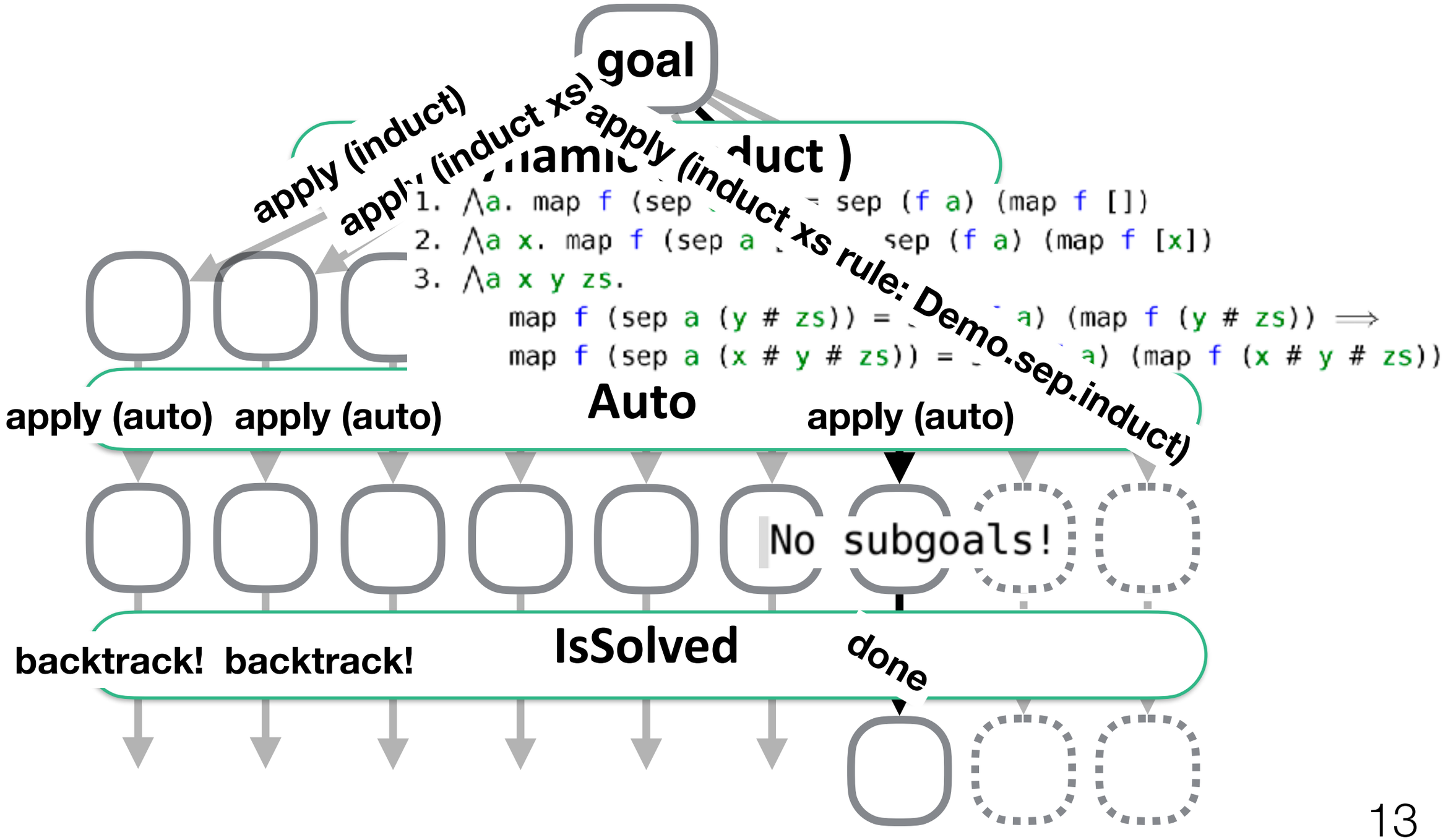
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

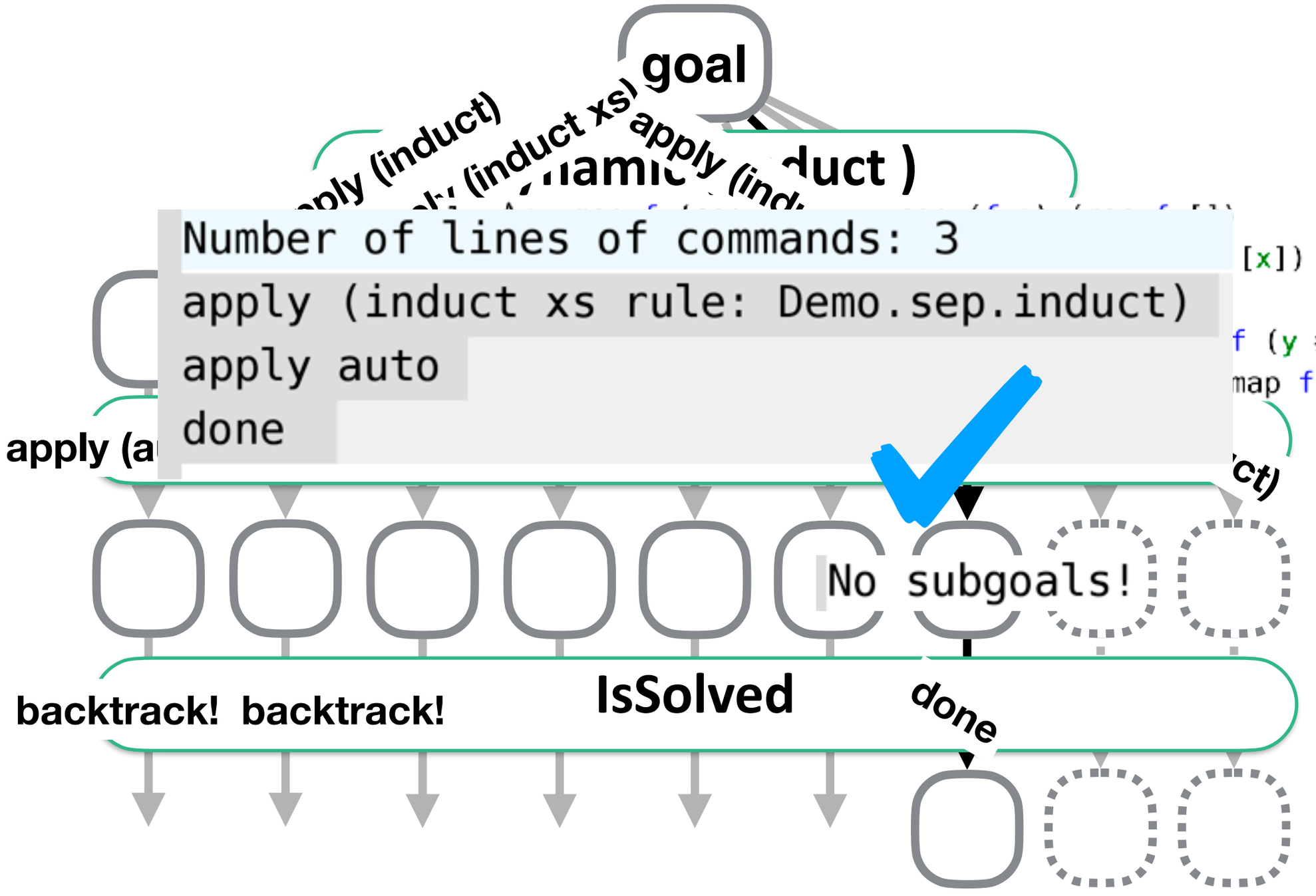
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

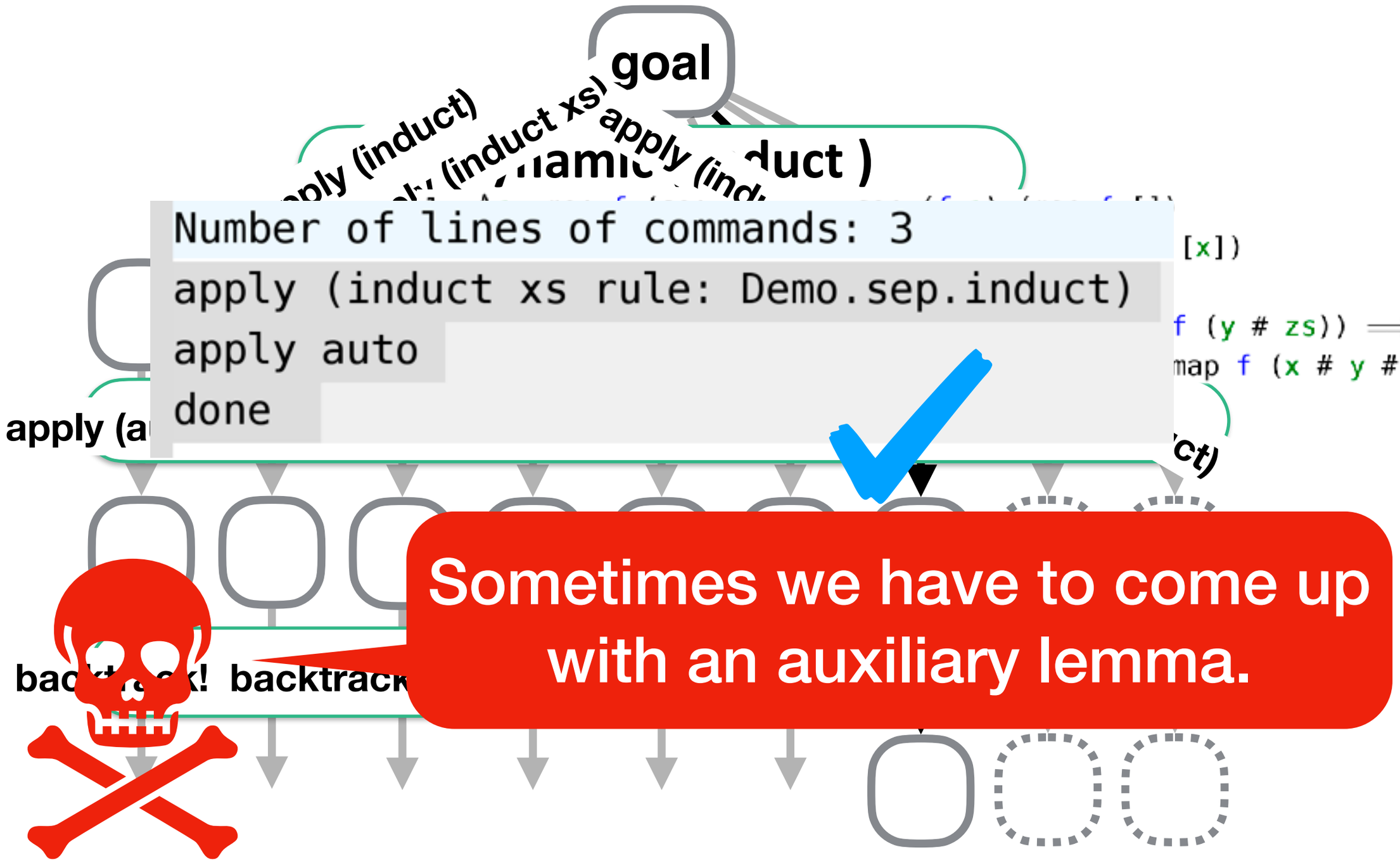
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



DEMO1

PSL: Proof Strategy Language

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



①

Isabelle's proof methods
(deductive reasoning)

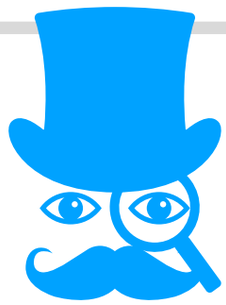
united reasoning
for automatic induction

③

machine learning induction
(inductive reasoning)

②

goal-oriented conjecturing
(abductive reasoning)



DEMO2

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

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

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

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

Output | Query | Sledgehammer | Symbols

43,29 (837/1004) (isabelle,isabelle,UTF-8-Isabelle) 35MB 7:01 PM

DEMO2

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

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

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

```
empty sequence. no proof found.
```

Output | Query | Sledgehammer | Symbols

44,18 (855/1004)

DEMO2

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

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

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

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

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

```
PGT creates 131 conjectures.
Testing conjecture with Quickcheck-exhaustive...
Testing conjecture with Quickcheck-exhaustive...
Testing conjecture with Quickcheck-exhaustive...
Testing conjecture with Quickcheck-exhaustive...
```

Output | Query | Sledgehammer | Symbols

45,27 (988/1000) (isabelle,isabelle,UTF-8-Isabelle) 371/535MB 6:17 PM

DEMO2

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

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

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

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

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

```
PGT creates 131 conjectures.
Testing conjecture with Quickcheck-exhaustive...
Testing conjecture with Quickcheck-exhaustive...
Testing conjecture with Quickcheck-exhaustive...
Testing conjecture with Quickcheck-exhaustive...
```

Output | Query | Sledgehammer | Symbols

45,27 (988/1000)

DEMO2

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

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

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

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

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

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

Output | Query | Sledgehammer | Symbols

45,27 (988/1000)

DEMO2

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

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

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

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

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

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

Output | Query | Sledgehammer | Symbols

45,27 (988/1000)

DEMO2

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

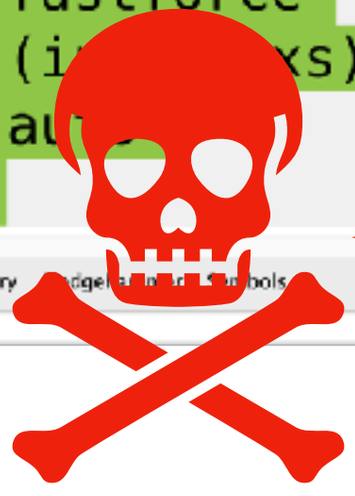
```
primrec rev:: "'a list ⇒ 'a list" where
  "rev []       = []"
| "rev (x # xs) = rev xs @ [x]"
primrec itrev:: "'a list ⇒ 'a list ⇒ 'a list" where
  "itrev [] ys     = ys"
| "itrev (x#xs) ys = itrev xs (x#ys)"

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

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

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

```
apply (subgoal_tac "∧Nil. itrev xs Nil = Example.rev xs @ Nil")
apply fastforce
apply (i      xs)
apply au
done
```

Output | Query | Sledgehammer | Symbols

45,27 (988/1000)



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

①

Isabelle's proof methods
(deductive reasoning)

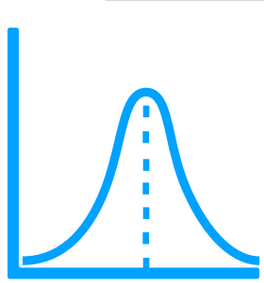
united reasoning
for automatic induction

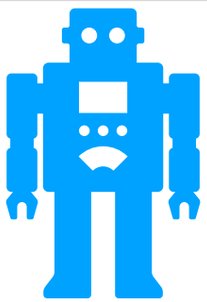
②

goal-oriented conjecturing
(abductive reasoning)

③

machine learning induction
(inductive reasoning)





DEMO3

Induction_Demo.thy (~/Workplace/PSL/Smart_Induct/Example/)

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

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

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

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

Output | Query | Sledgehammer | Symbols

18,34 (548/835) | Input/output complete | (isabelle,isabelle,UTF-8-Isabelle) | 3:24 PM

DEMO3

Induction_Demo.thy (~/Workplace/PSL/Smart_Induct/Example/)

```
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
  oops
```

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

```
smart_induct produced 40 combinations of arguments for the induct me
... out of which 32 of them return some results.
... out of which only 32 of them passes the second screening stage.
LiFtEr assertions are evaluating the first 32 of them.
Try these 10 most promising inductions!
1st candidate is apply (induct xs arbitrary: ys)
  (* The score is 21 out of 21. *)
2nd candidate is apply (induct xs)
  (* The score is 21 out of 21. *)
3th candidate is apply (induct xs ys rule: Induction_Demo.itrev.indu
  (* The score is 19 out of 21. *)
```

Output | Query | Sledgehammer | Symbols

18,47 (561/848) | Input/output complete

DEMO3

Induction_Demo.thy (~/Workplace/PSL/Smart_Induct/Example/)

```
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 (induct xs arbitrary: ys) apply auto done
```

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

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

Output | Query | Sledgehammer | Symbols

19.45 (606/891) (isabelle,isabelle,UTF-8-Isabelle) 227/512MB 3:23 PM

①

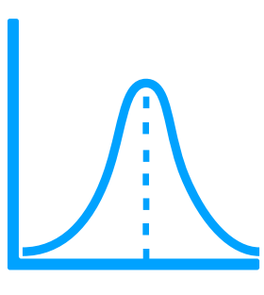
Isabelle's proof methods
(deductive reasoning)

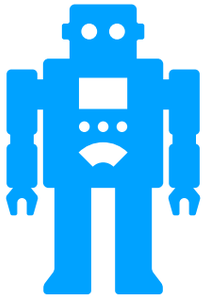
united reasoning
for automatic induction

fin!

③

machine learning induction
(inductive reasoning)

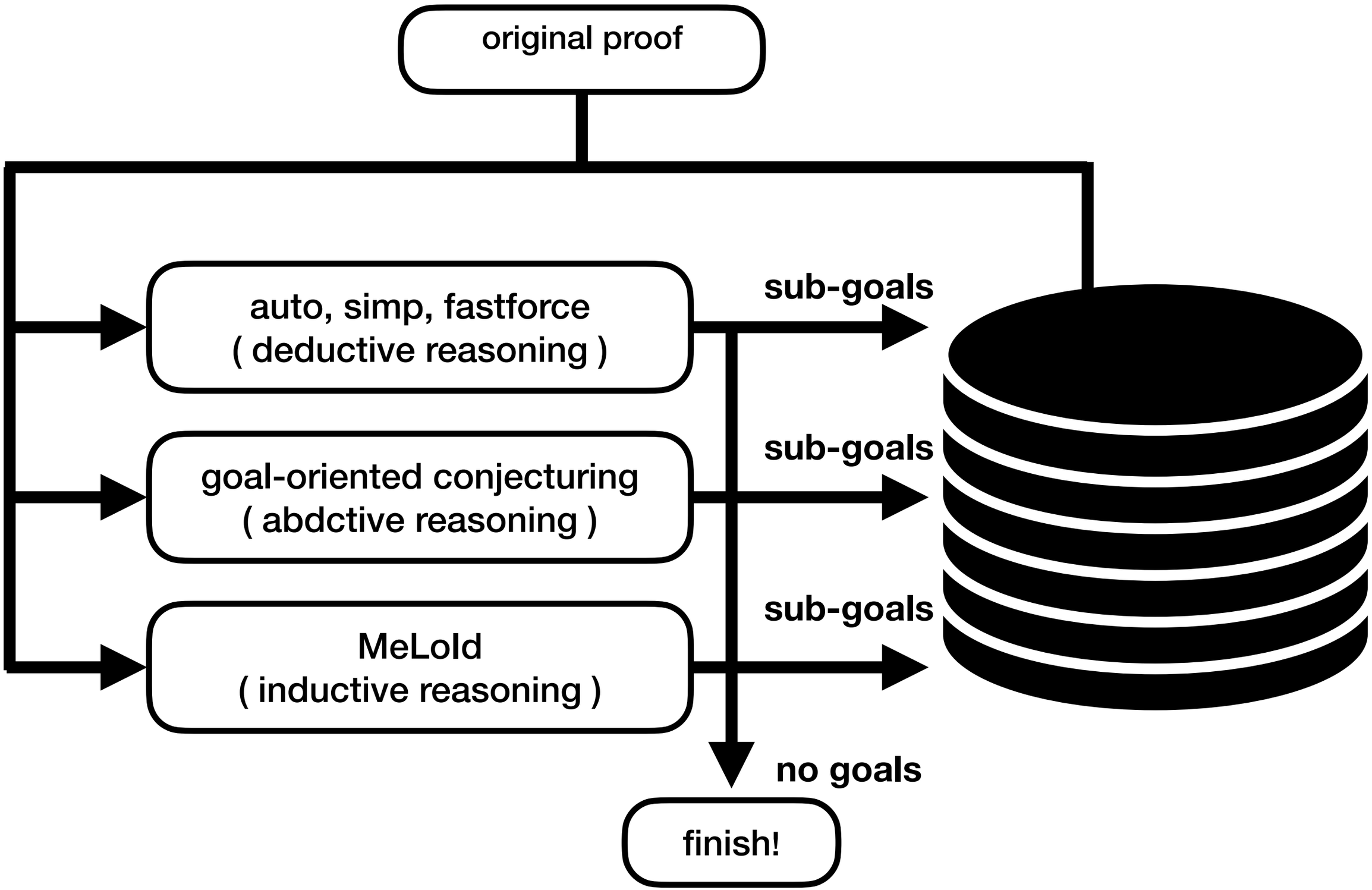




②

goal-oriented conjecturing
(abductive reasoning)

backup slides for Q&A


original proof
auto, simp, fastforce
(deductive reasoning)
goal-oriented conjecturing
(abdctive reasoning)
MeLoId
(inductive reasoning)
sub-goals
sub-goals
sub-goals
no goals
finish!

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

goal

DEMO2

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

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

goal

Conjecture

Fastforce

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

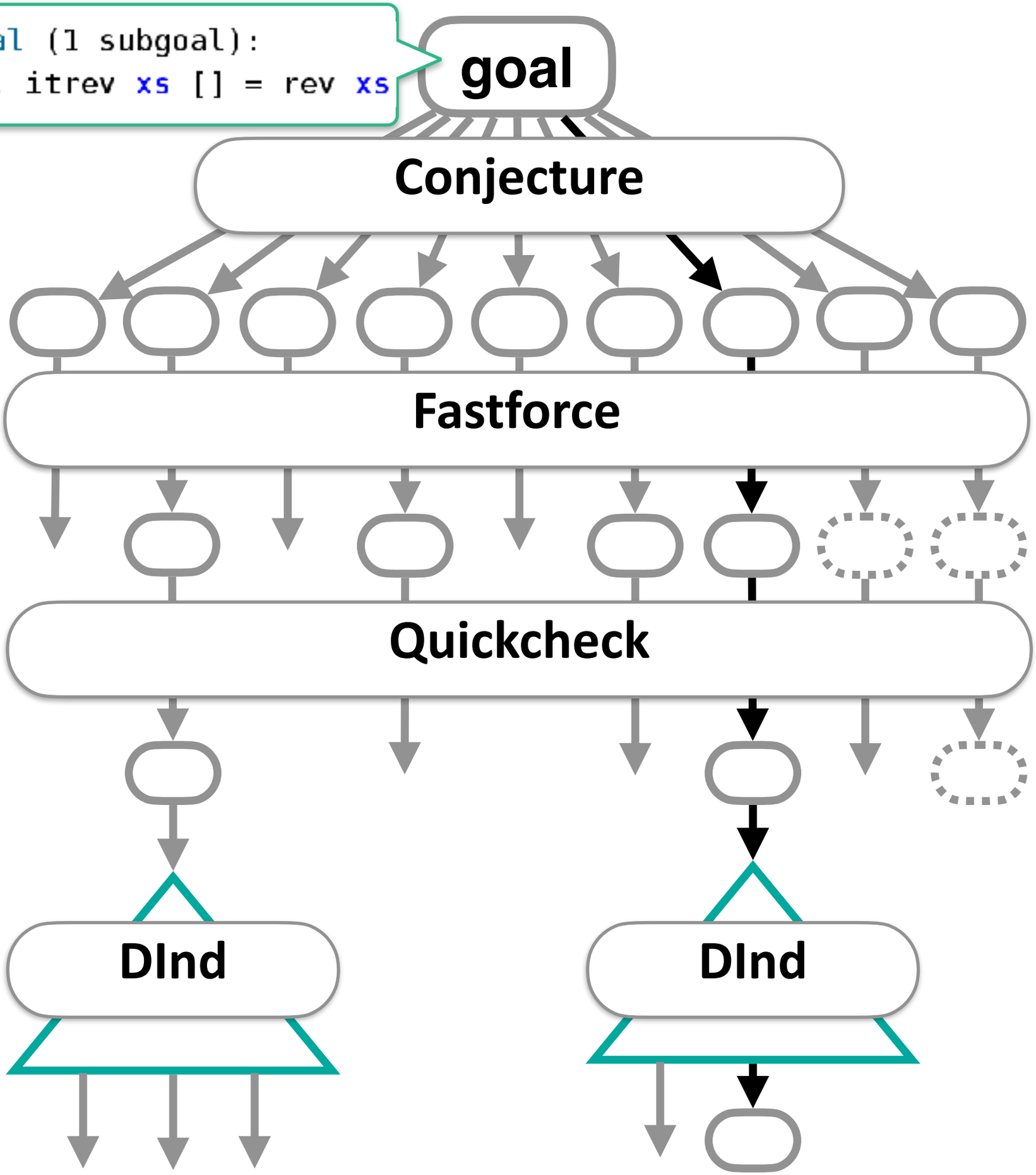
Quickcheck

DInd

DInd

DEMO2

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



DEMO2

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

goal

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

Conjecture

Fastforce

Quickcheck

DInd

DInd

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

goal

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

```
goal (2 subgoals):
 1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
    itrev xs [] = rev xs
 2. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Conjecture

Fastforce

Quickcheck

DInd

DInd

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

goal

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

```
goal (2 subgoals):
 1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
    itrev xs [] = rev xs
 2. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Conjecture

Fastforce

```
apply fastforce
```

Quickcheck

DInd

DInd

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

goal

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

Conjecture

```
goal (2 subgoals):
 1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
    itrev xs [] = rev xs
 2. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Fastforce

```
apply fastforce
```

```
goal (1 subgoal):
 1. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Quickcheck

DInd

DInd

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

goal

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

goal (2 subgoals):
1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
   itrev xs [] = rev xs
2. ⋀Nil. itrev xs Nil = rev xs @ Nil

Conjecture

Fastforce

apply fastforce

goal (1 subgoal):
1. ⋀Nil. itrev xs Nil = rev xs @ Nil

Quickcheck

goal (1 subgoal):
1. ⋀Nil. itrev xs Nil = rev xs @ Nil

DInd

DInd

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

goal

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

Conjecture

```
goal (2 subgoals):
 1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
    itrev xs [] = rev xs
 2. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Fastforce

```
apply fastforce
```

```
goal (1 subgoal):
 1. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Quickcheck

```
goal (1 subgoal):
 1. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

DInd

DInd

```
apply (induct xs)
apply auto
done
```

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

goal

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

Conjecture

```
goal (2 subgoals):
 1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
    itrev xs [] = rev xs
 2. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Fastforce

```
apply fastforce
```

```
goal (1 subgoal):
 1. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Quickcheck

```
goal (1 subgoal):
 1. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

DInd

DInd

```
apply (induct xs)
apply auto
done
```

```
theorem itrev ?xs [] = rev ?xs
```

goal

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

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

Conjecture

```
goal (2 subgoals):
 1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
    itrev xs [] = rev xs
 2. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Fastforce

```
apply fastforce
```

```
goal (1 subgoal):
 1. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Quickcheck

```
goal (1 subgoal):
 1. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

DInd

DInd

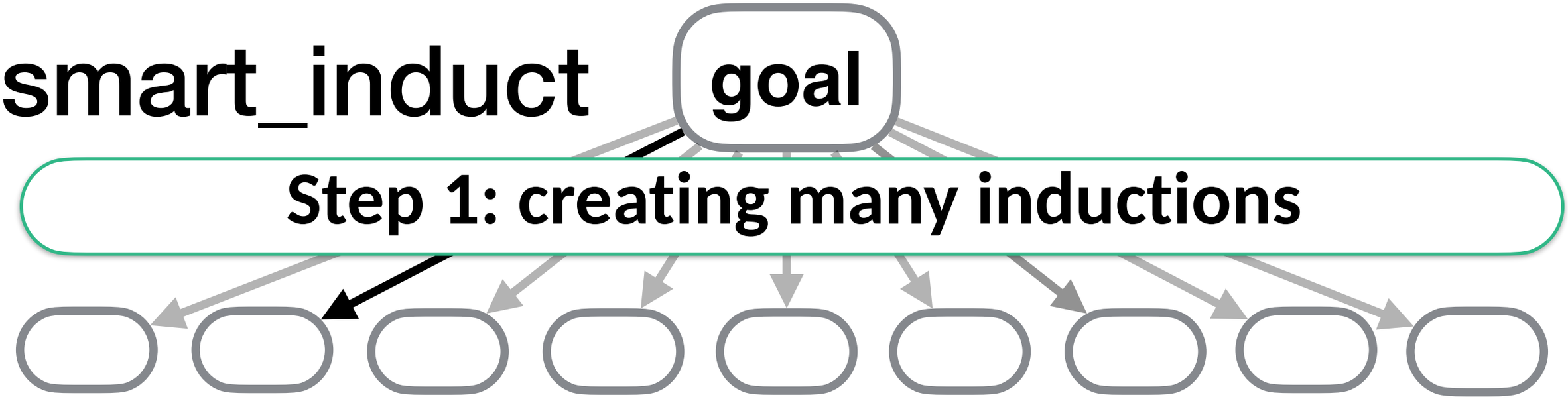
```
apply (induct xs)
apply auto
done
```

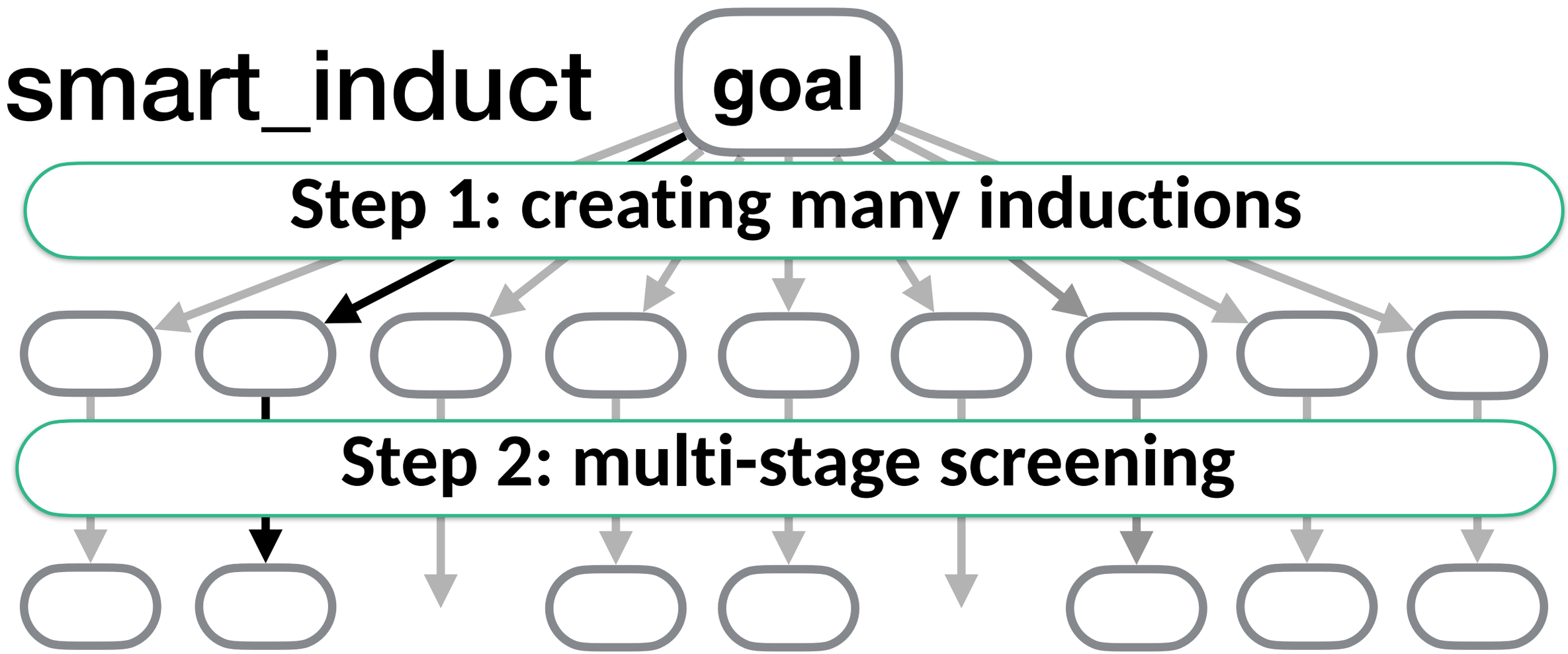
```
theorem itrev ?xs [] = rev ?xs
```

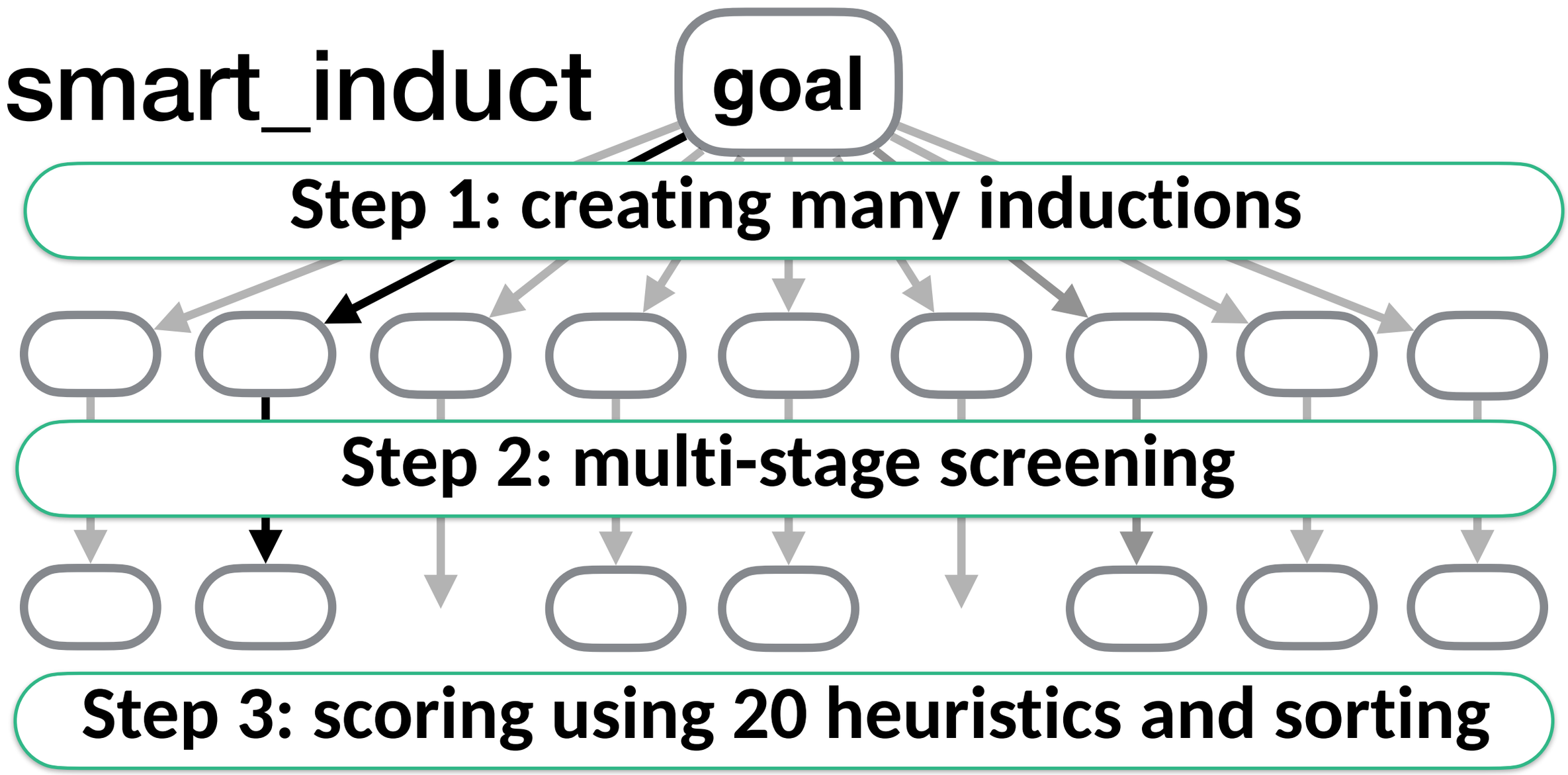
smart_induct

goal

smart_induct

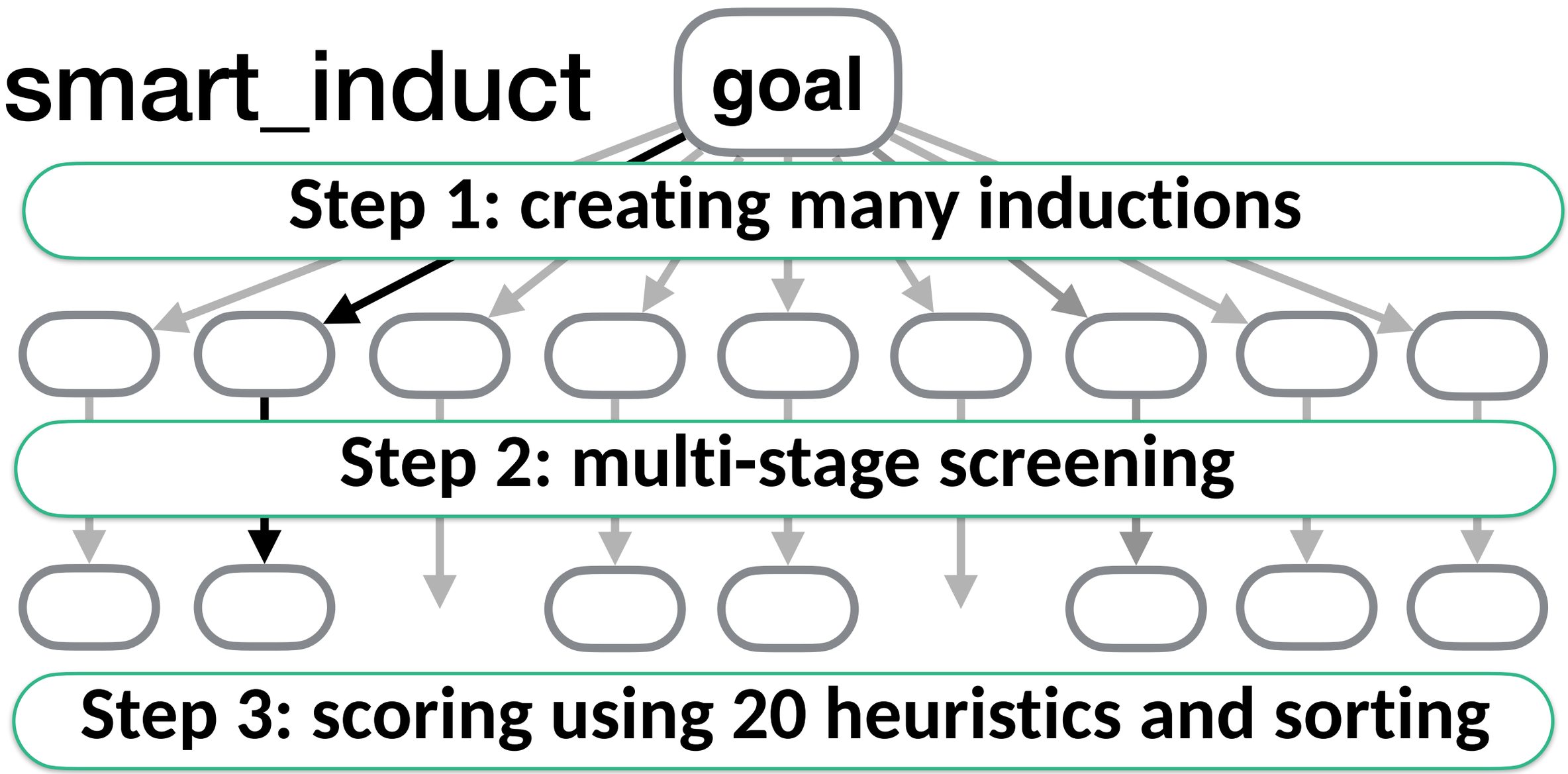



smart_induct
goal
Step 1: creating many inductions
Step 2: multi-stage screening



Step 3: scoring using 20 heuristics and sorting

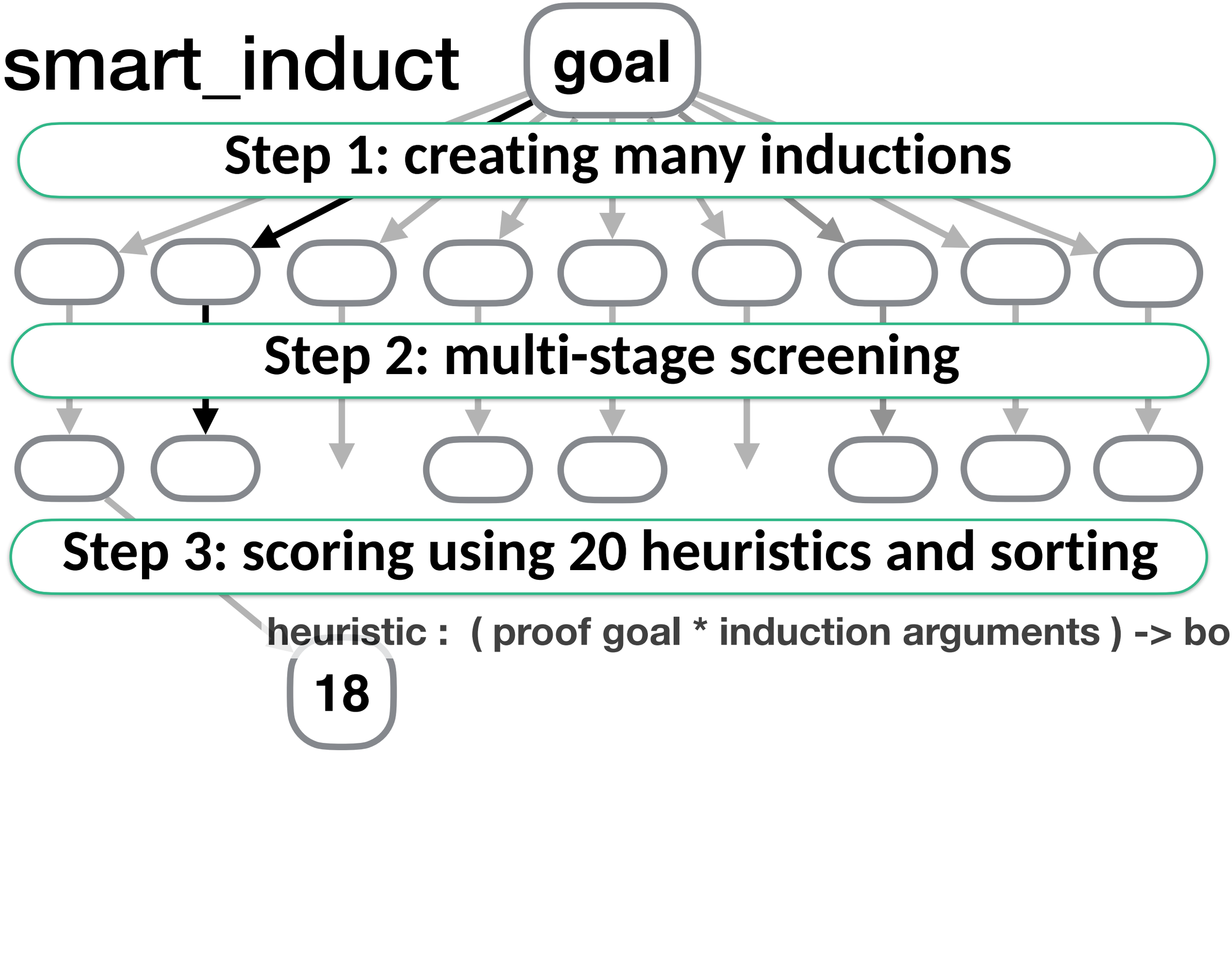
smart_induct



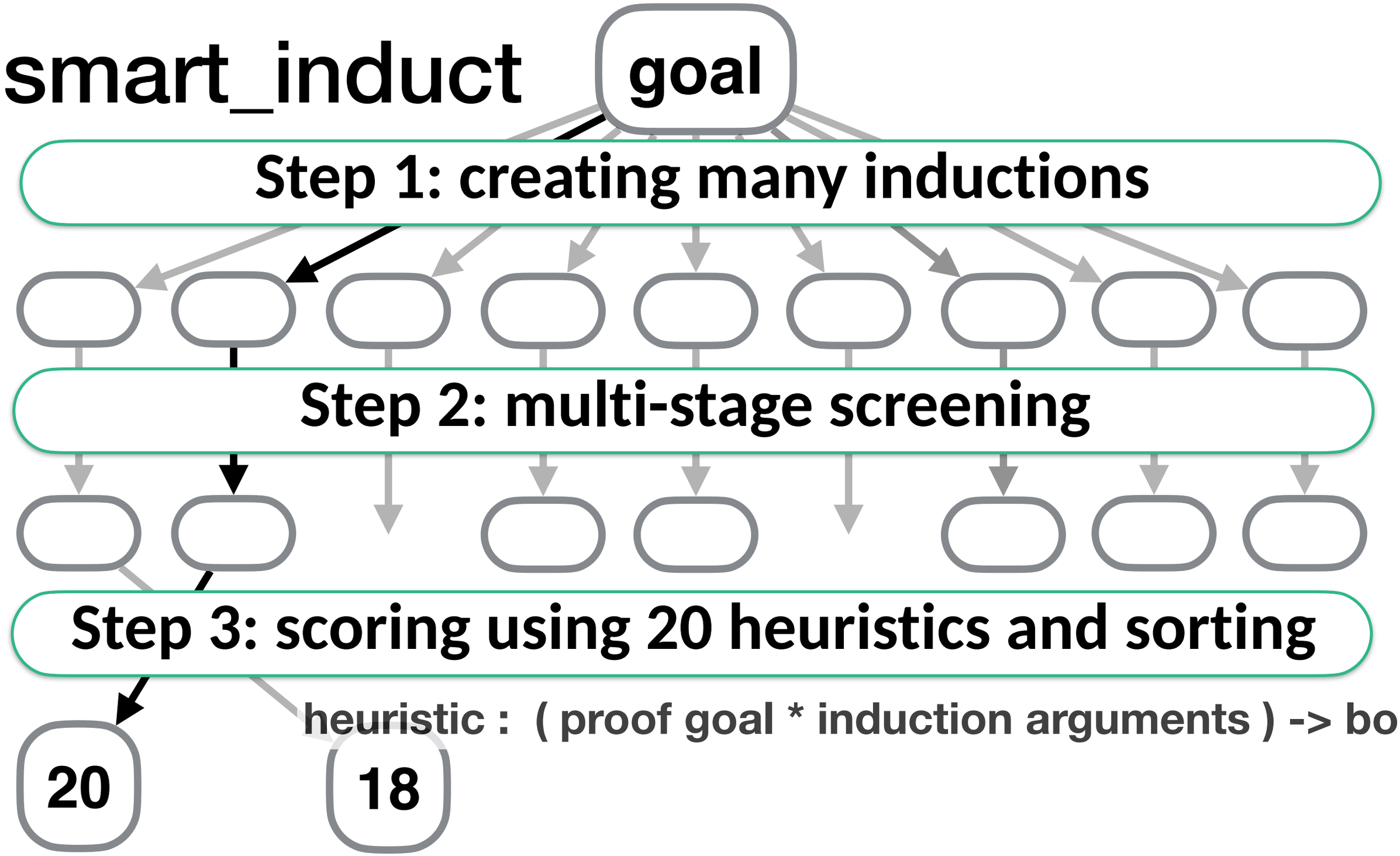
Step 3: scoring using 20 heuristics and sorting

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

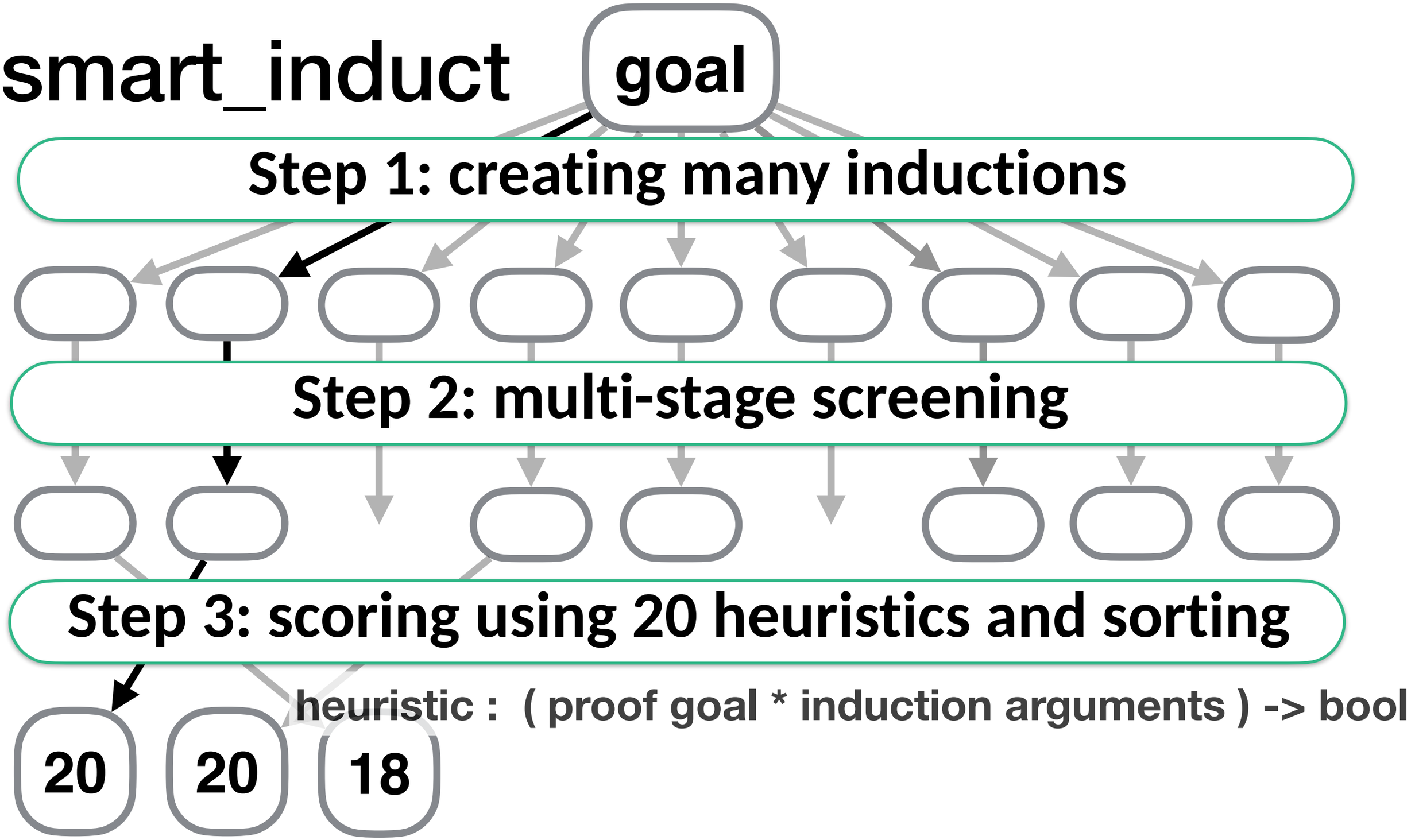
smart_induct



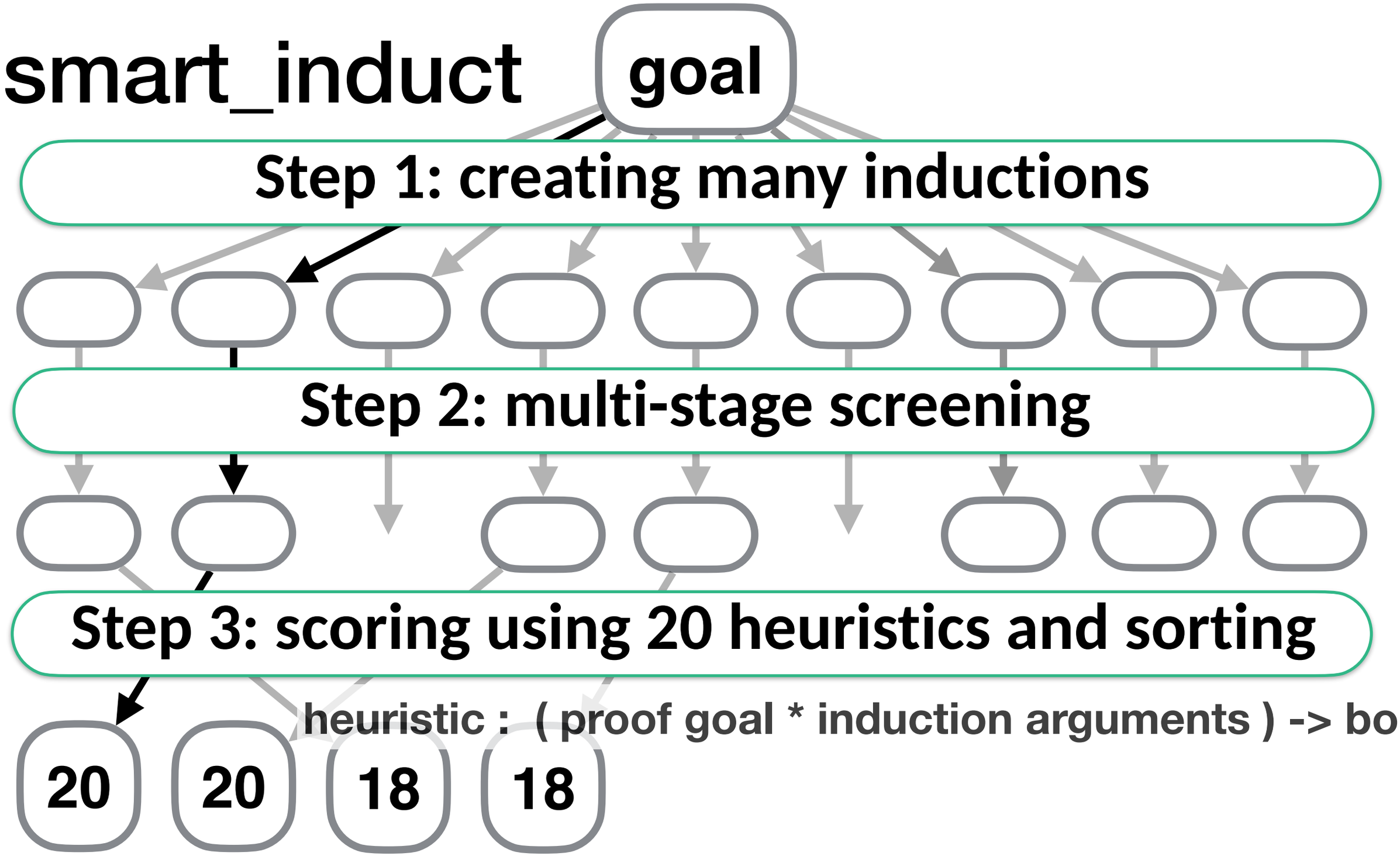
smart_induct



smart_induct



smart_induct



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 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 18 17

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 18 18 17

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 18 18 17

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

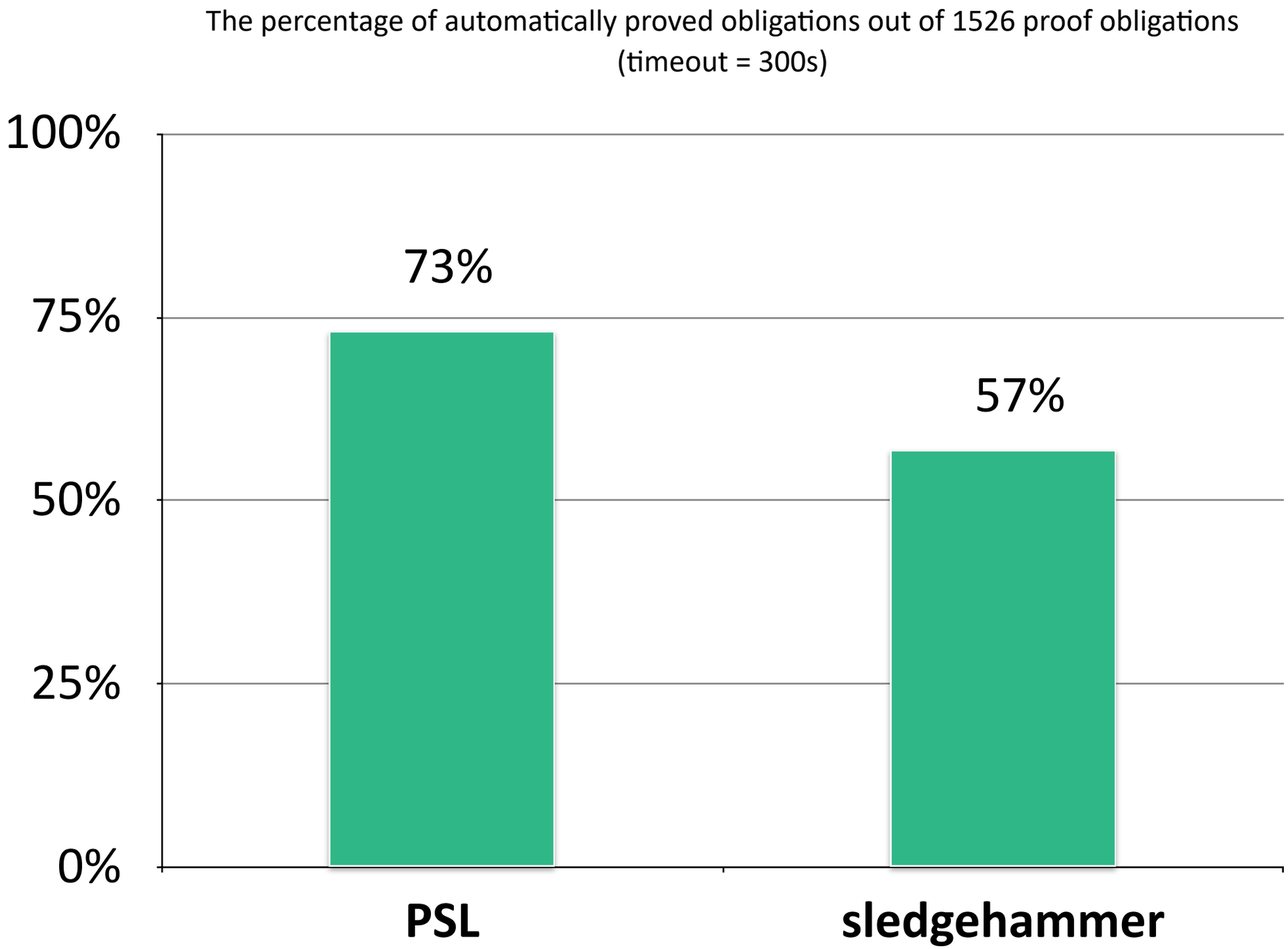
heuristic : (proof goal * induction arguments) -> bool

20 20 18 18 18 18 17

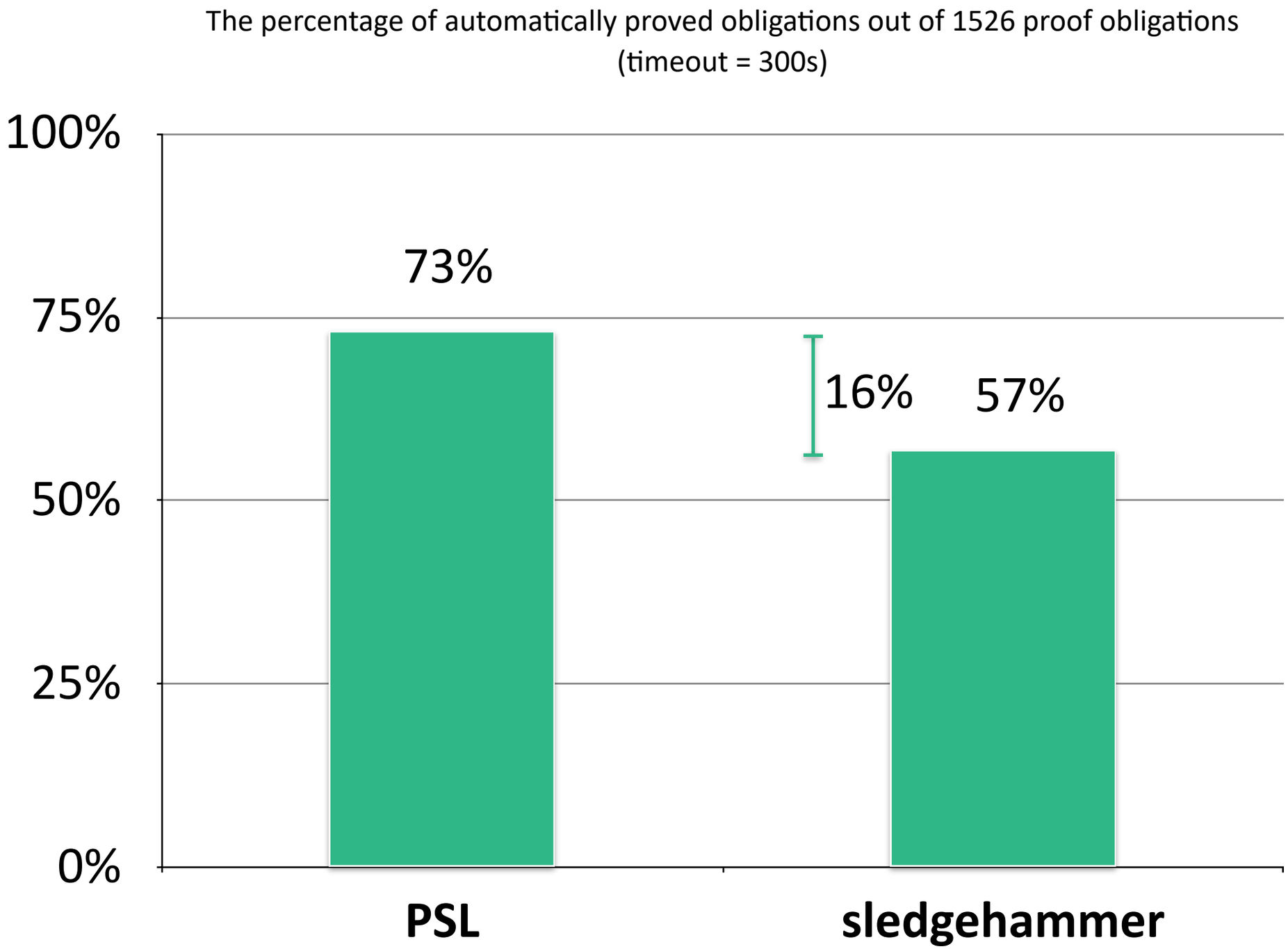
Step 4: short-listing

DEMO!

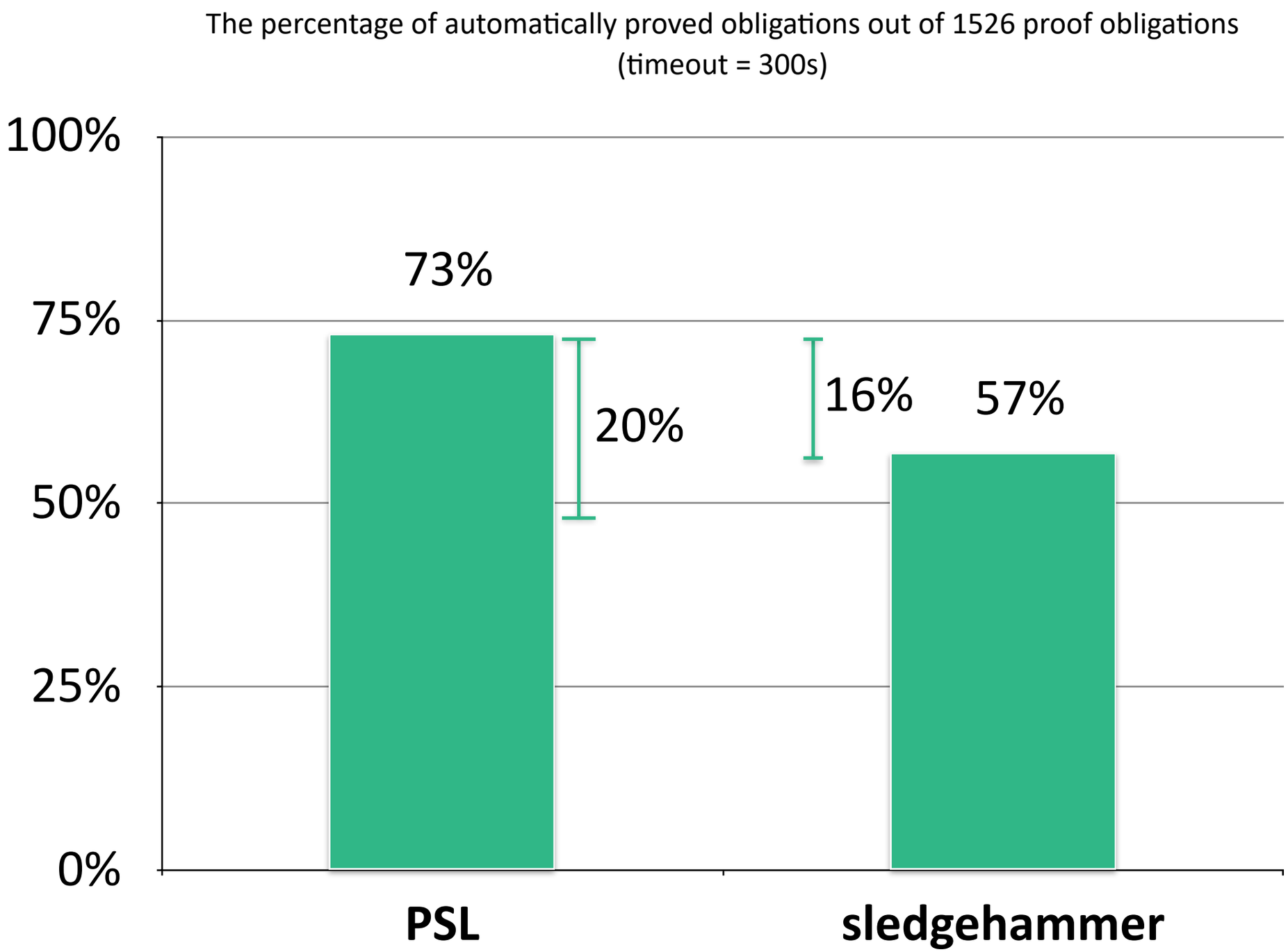
PSL vs sledgehammer



PSL vs sledgehammer

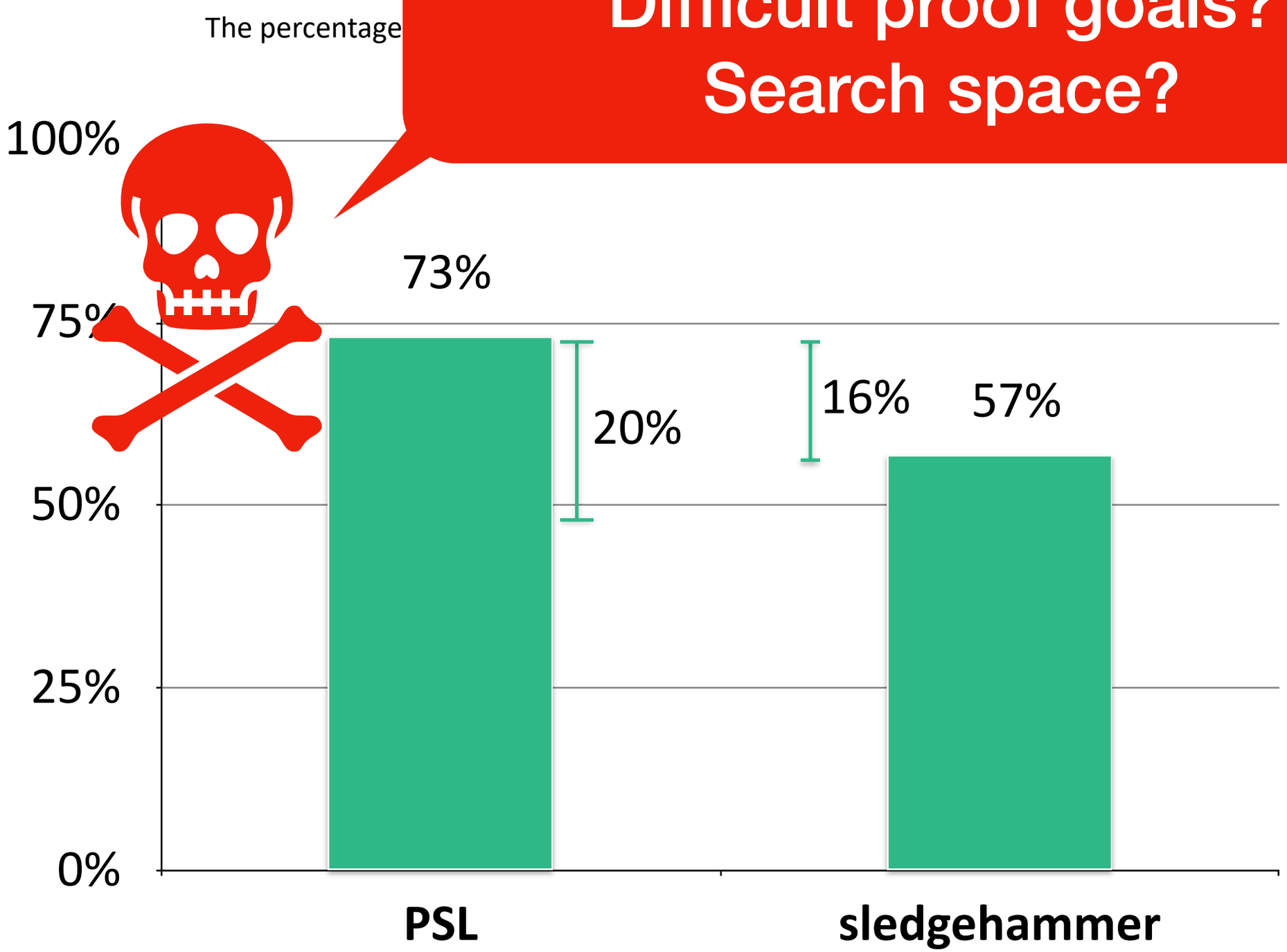


PSL vs sledgehammer



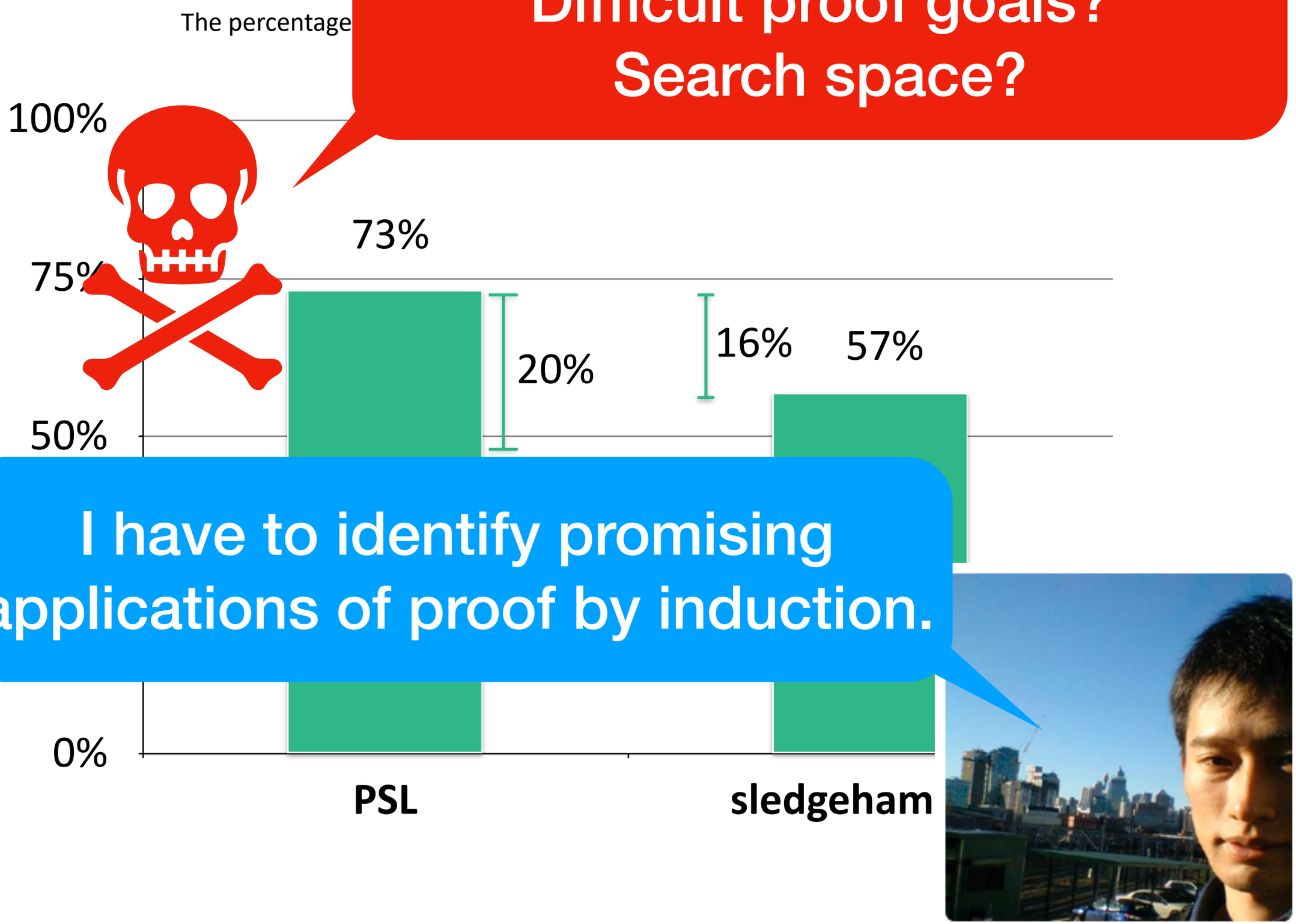
PSL vs sledgehammer

Difficult proof goals?
Search space?



PSL vs sledgehammer

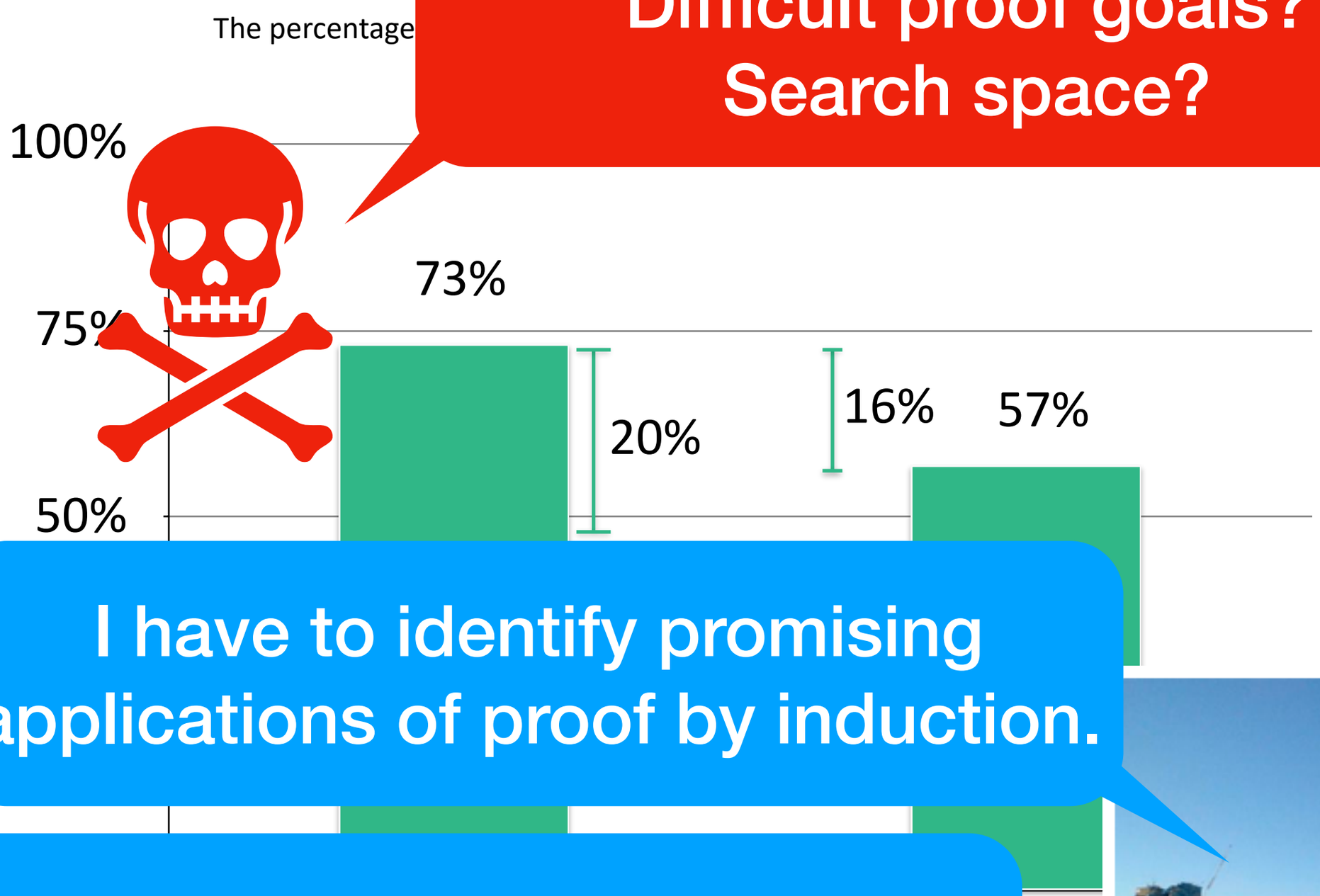
Difficult proof goals?
Search space?



I have to identify promising applications of proof by induction.

PSL vs sledgehammer

Difficult proof goals?
Search space?



I have to identify promising applications of proof by induction.

without completing a proof search

goal

Step 1: creating many inductions

Step 2: multi-stage screening

Step 3: scoring using 20 heuristics and sorting

20 20 18 18 18 18 17

Step 4: short-listing

goal

Step 1: creating many inductions

Step 2: multi-stage screening

Step 3: scoring using 20 heuristics and sorting

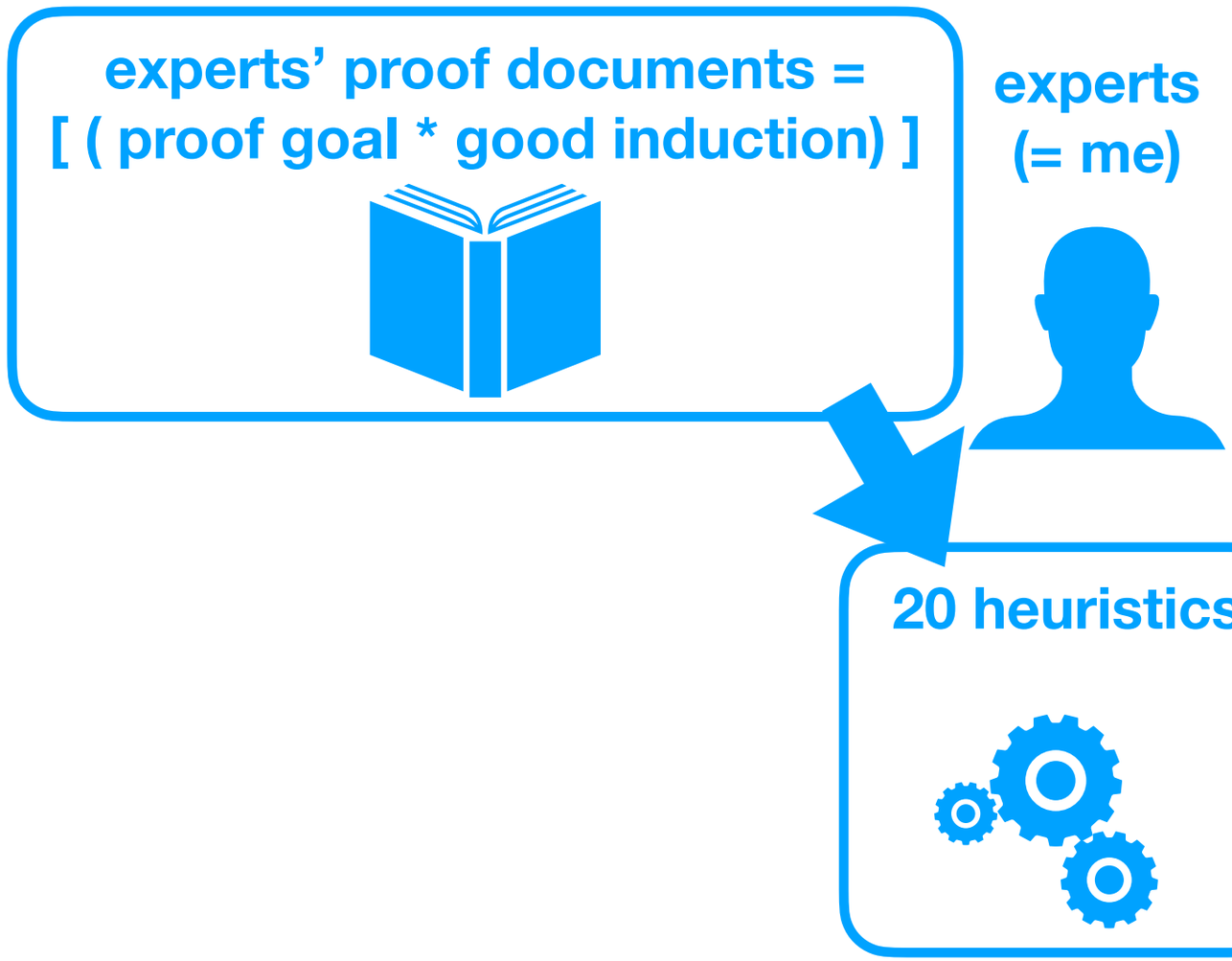
?

20 20 18 18 18 18 17

Step 4: short-listing

It is difficult to write induction heuristics.

It is difficult to write induction heuristics.



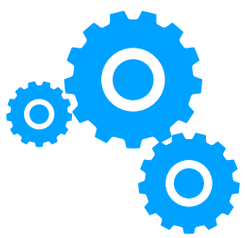
It is difficult to write induction heuristics.

```
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



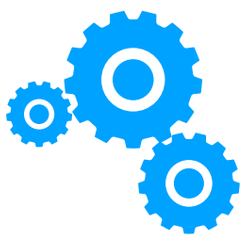
It is difficult to write induction heuristics.

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

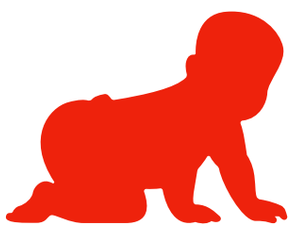
experts' proof documents =
[(proof goal * good induction)]

experts
(= me)

20 heuristics



new users



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

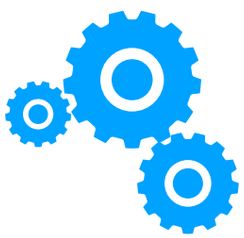
It is difficult to write induction heuristics.

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

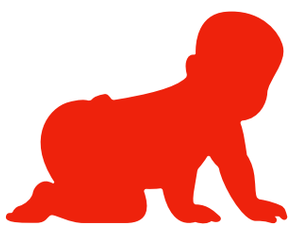
experts' proof documents =
[(proof goal * good induction)]

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!

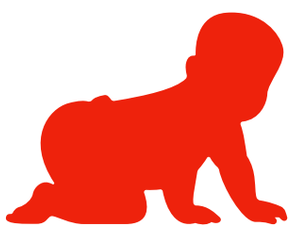
It is difficult to write induction heuristics.

```
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

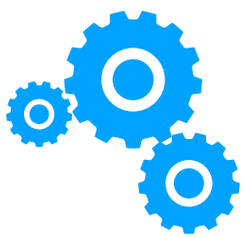


proof goal

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.

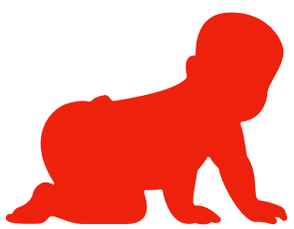
① blue = what I can see before releasing smart_induct

```
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

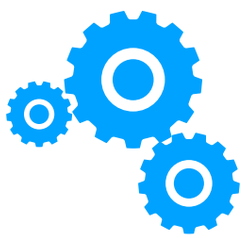


proof goal

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.

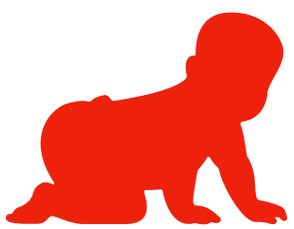
① blue = what I can see before releasing smart_induct

```
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

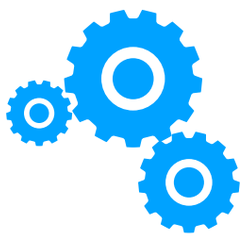


proof goal

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

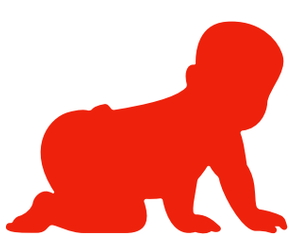
It is difficult to write induction heuristics.

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)



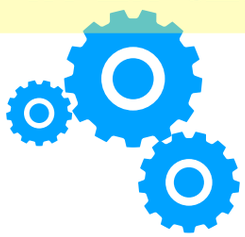
new users

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

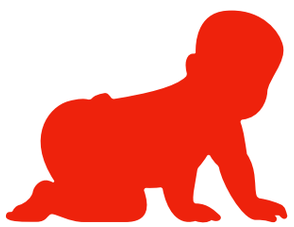
```
lemma "itrev xs ys = rev xs @ ys"
by(induct xs ys rule:"itrev.induct") auto
```

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

experts
(= me)

LiFtEr: Logical
Feature Extractor

new users

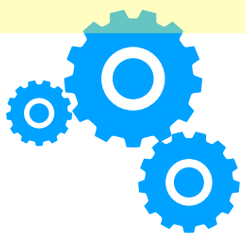


proof goal

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* . *a*
| $\forall x$: `term` $\in$ *modifier_term* . *a*
| $\exists x$: `rule` . *assertion*
| $\exists x$: `term_occurrence` $\in y$: `term` . *assertion*
| $\forall x$: `term_occurrence` $\in y$: `term` .
connective := `True` | `False` | *assertion* $\vee$ *asse* *assertion*
assertion $\rightarrow$ *assertion* | $\neg$ *asserti*
pattern := `all_only_var` | `all_constructor`

atomic :=
rule `is_rule_of` *term_occurre*
| *term_occurrence* `term_occurrence` *erm*
| `are_same_term` (*term_occurrence* , *term_occ*
| *term_occurrence* `is_in_term_occurrence` *term*
| `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` *occurrence*
| *term_occurrence* `is_nth_argument_of` *occurrence*

smart_induct... does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

smart_induct... does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

smart_induct... does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

smart_induct... does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

smart_induct… does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

smart_induct... does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

smart_induct... does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

smart_induct… does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

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

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

smart_induct… does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

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

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

smart_induct… does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

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

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

Overall, the results are good!

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

smart_induct… does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

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

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

Overall, the results are good!

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

WIP!

<- simple representation


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

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

<- 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]"

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

<- relevant definitions

<- 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
```

<- relevant definitions

```
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)"
```

<- simple representation

LiFtEr
LiFtEr
LiFtEr

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

LiFtEr 2
LiFtEr 2

<- nested assertions to examine the "semantics" of constants (rev and itrev)

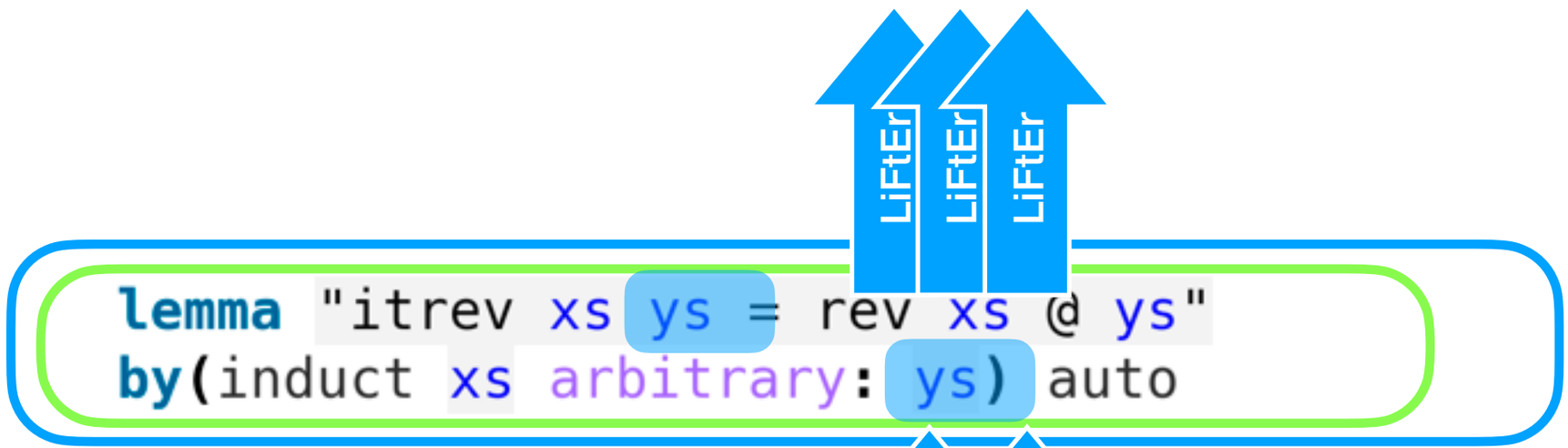
```
lemma "itrev xs ys = rev xs @
by(induct xs ys rule:"itrev.i                ) auto
```

<- relevant definitions

```
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)"
```

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

<- simple representation




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

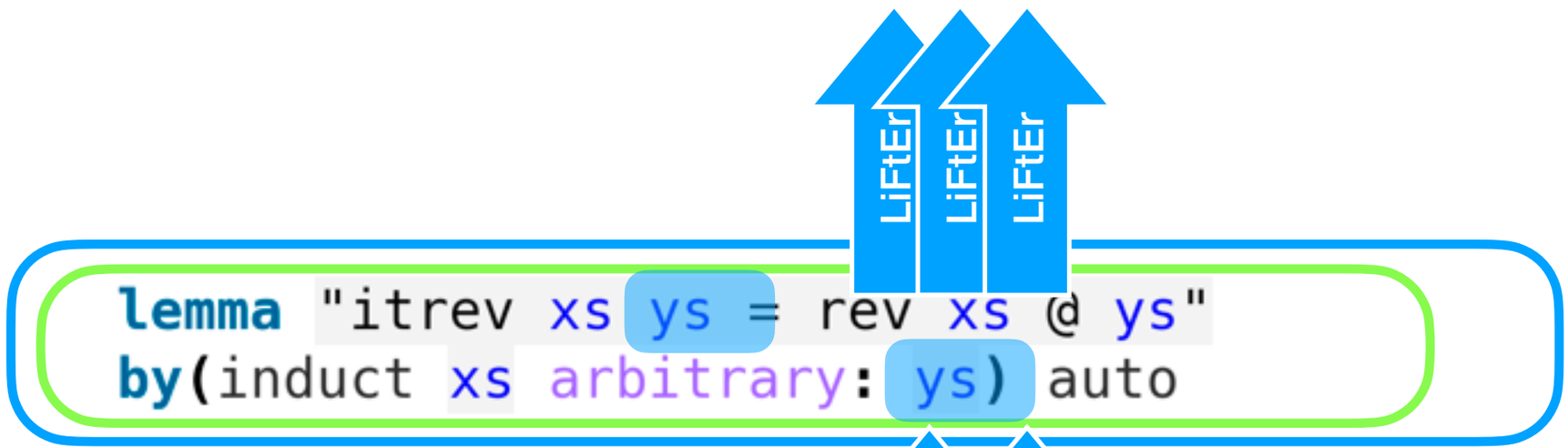
<- nested assertions to examine the "semantics" of constants (rev and itrev)

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

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

<- relevant definitions

LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions

<- simple representation




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

<- nested assertions to examine the "semantics" of constants (rev and itrev)

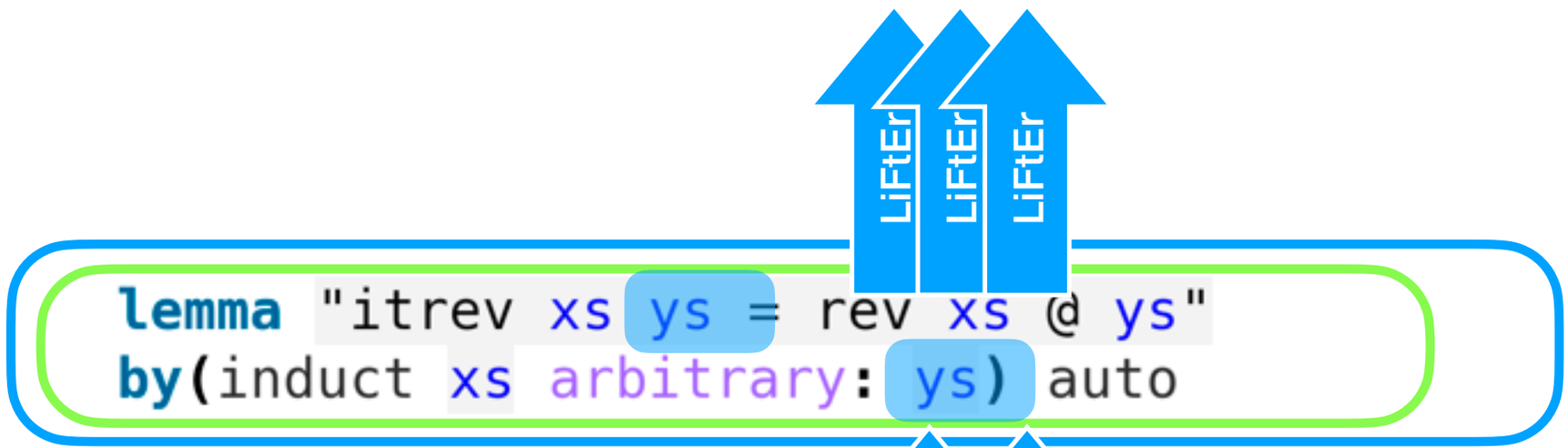
```
lemma "itrev xs ys = rev xs @
by(induct xs ys rule:"itrev.i                ) auto
```

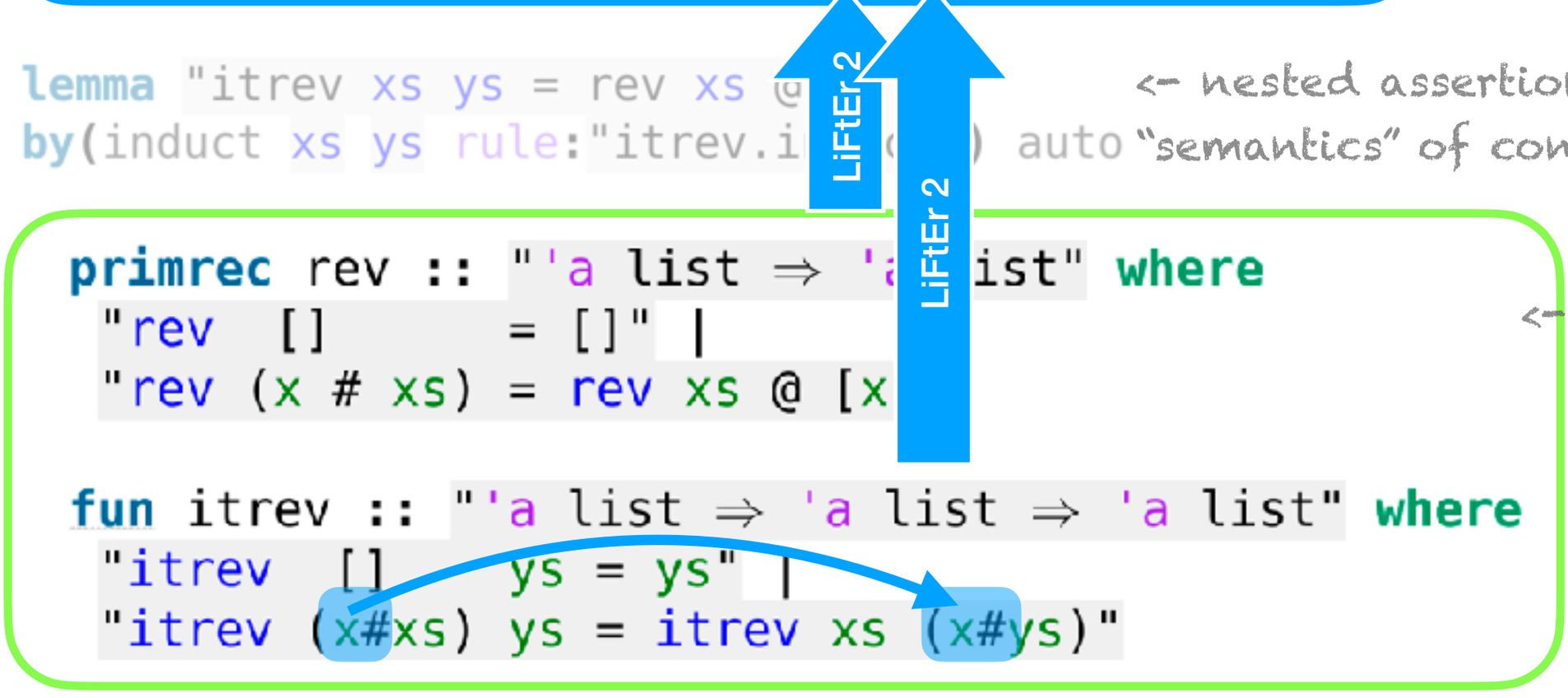
```
primrec rev :: "'a list ⇒ 'a list" where
 "rev  []       = []" |
 "rev (x # xs) = rev xs @ [x

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

<- relevant definitions

LiFtEr$_2$ (proof goal * induction arguments) -> bool
*** relevant definitions**

coming soon



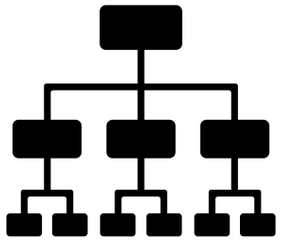


LiFtEr: (proof goal * induction arguments) -> bool

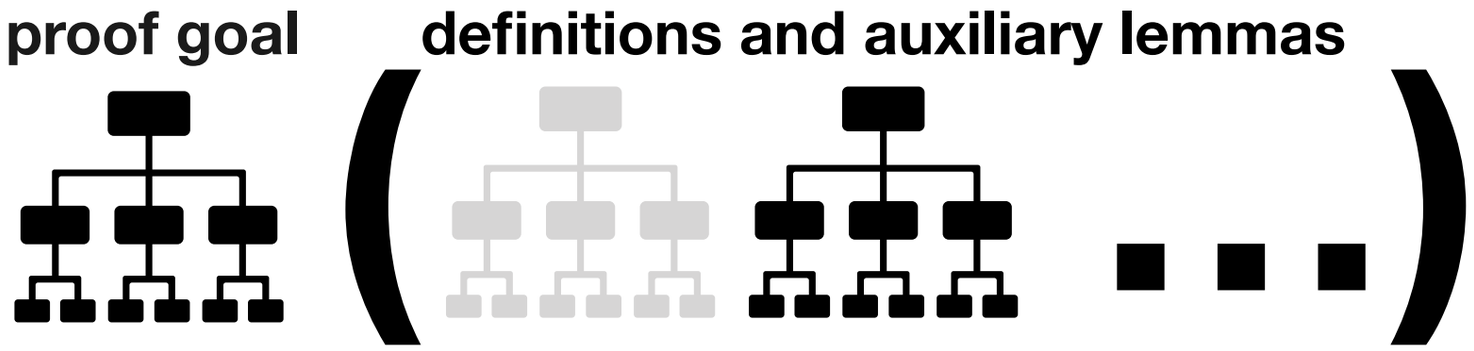
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions

LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions

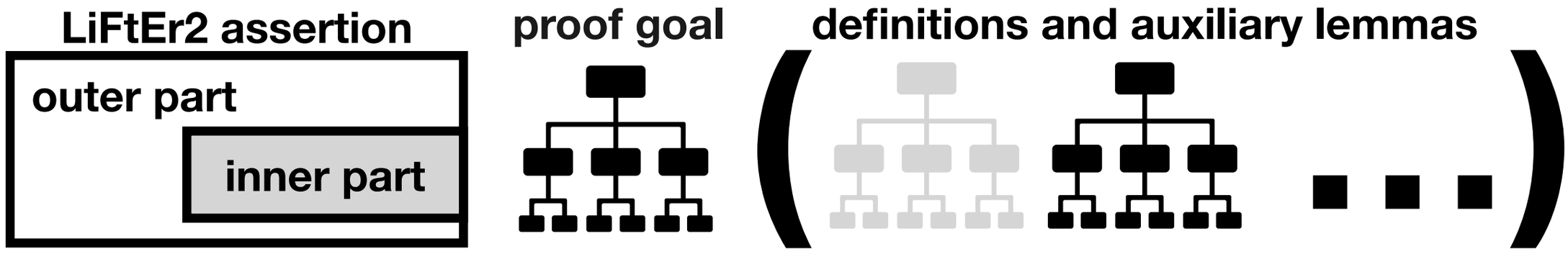
proof goal



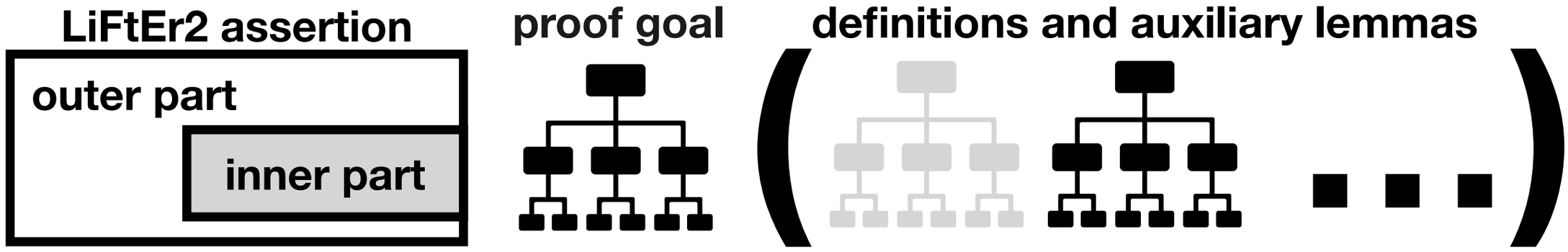
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions



LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions

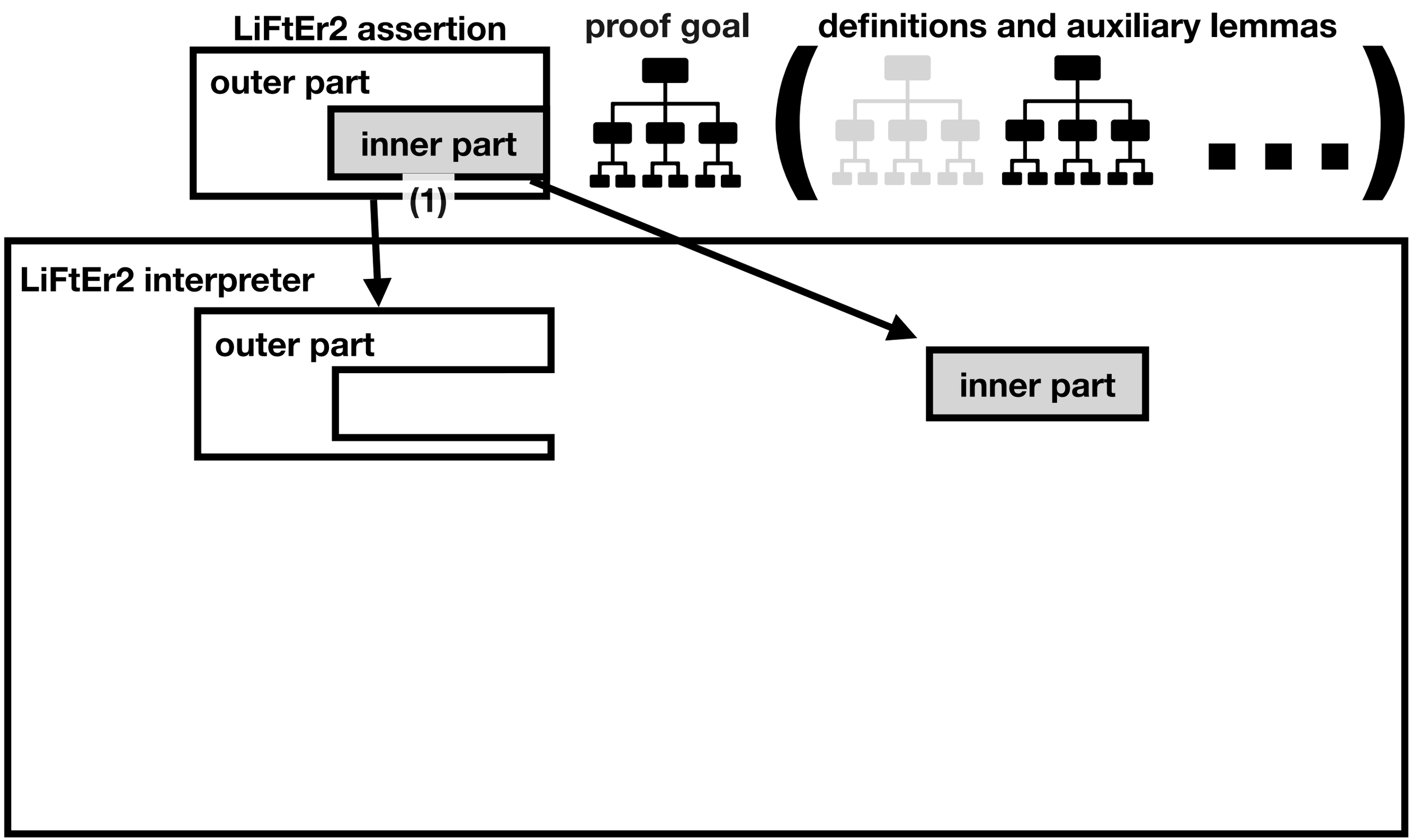


LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions

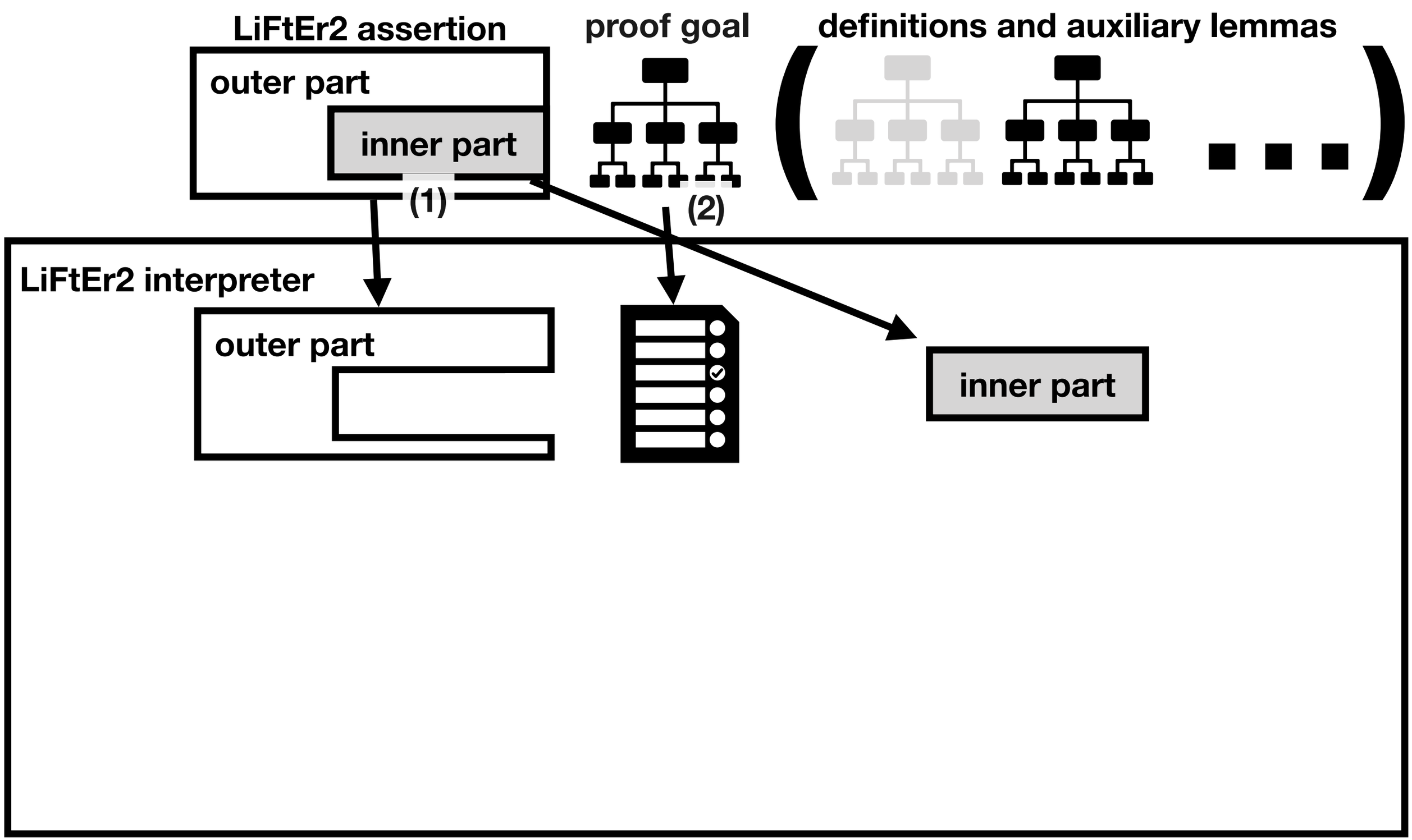


LiFtEr2 interpreter

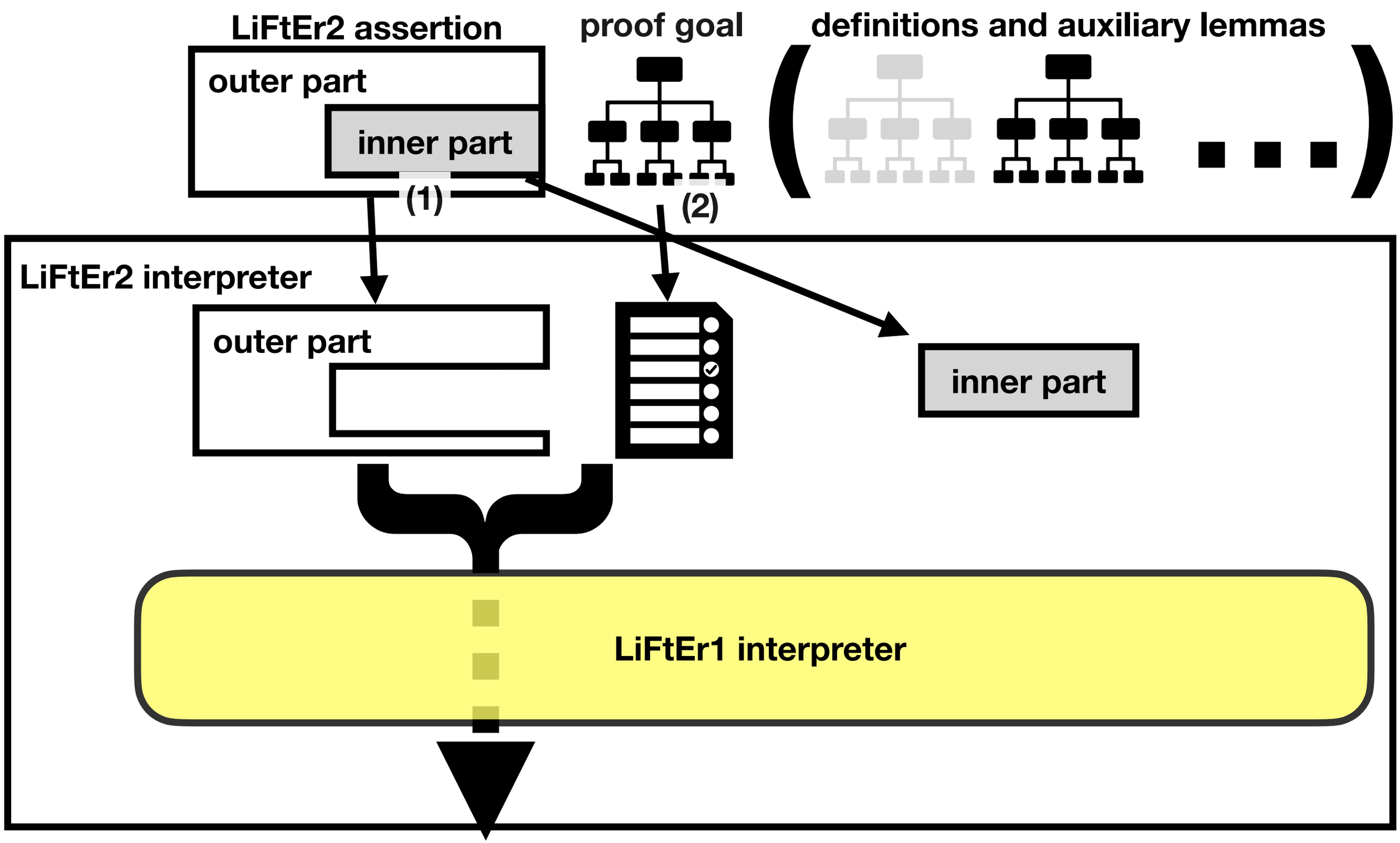
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions



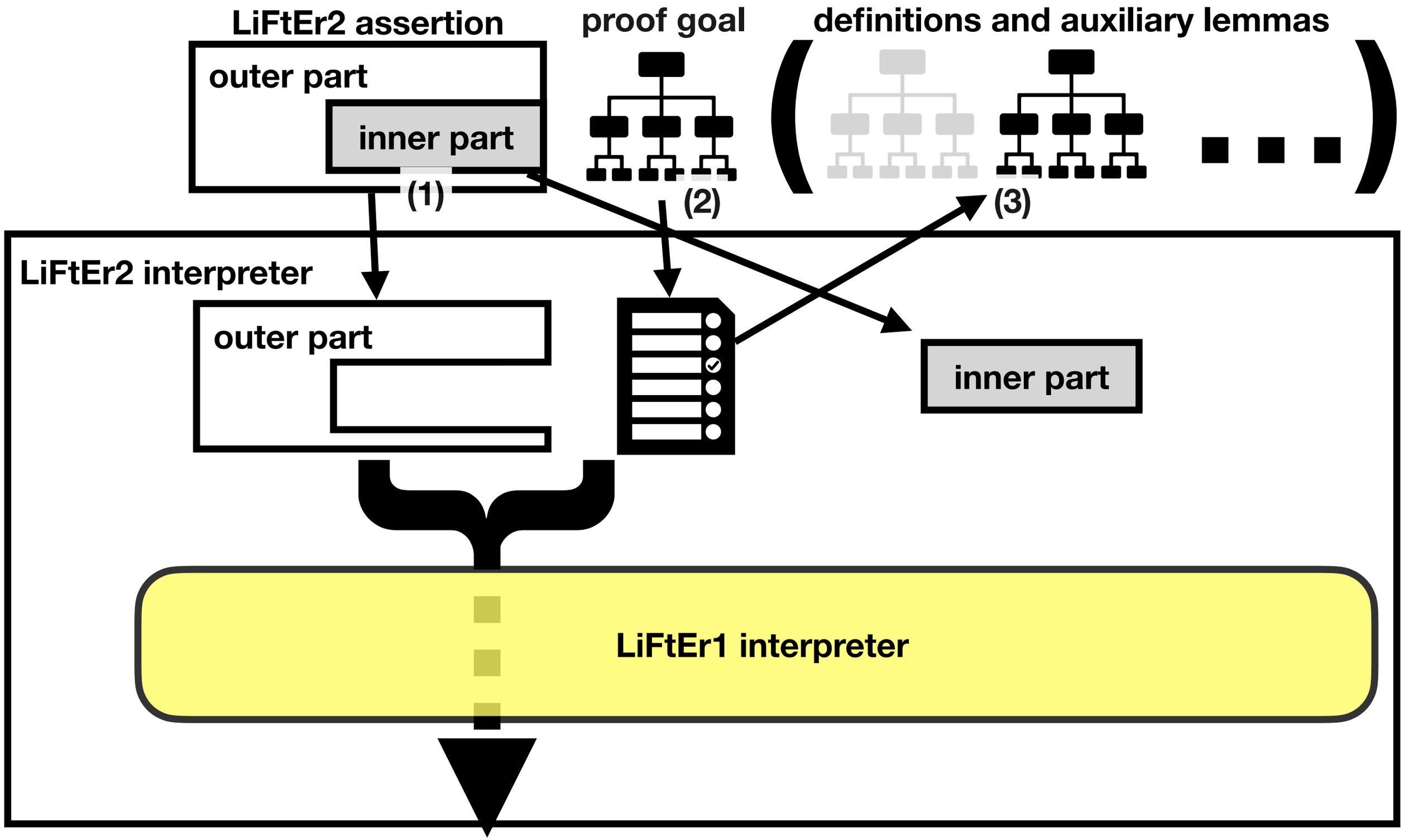
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions



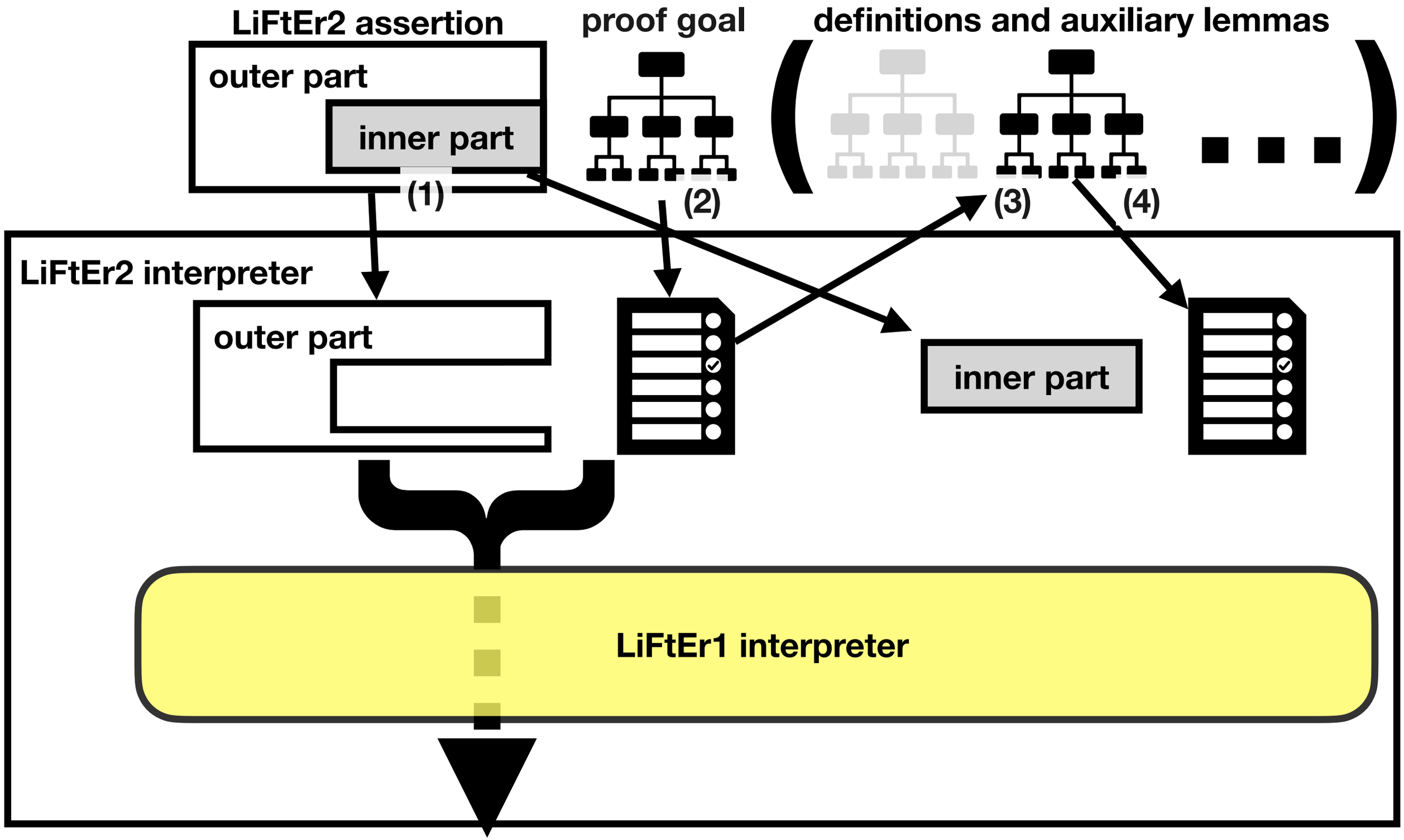
LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



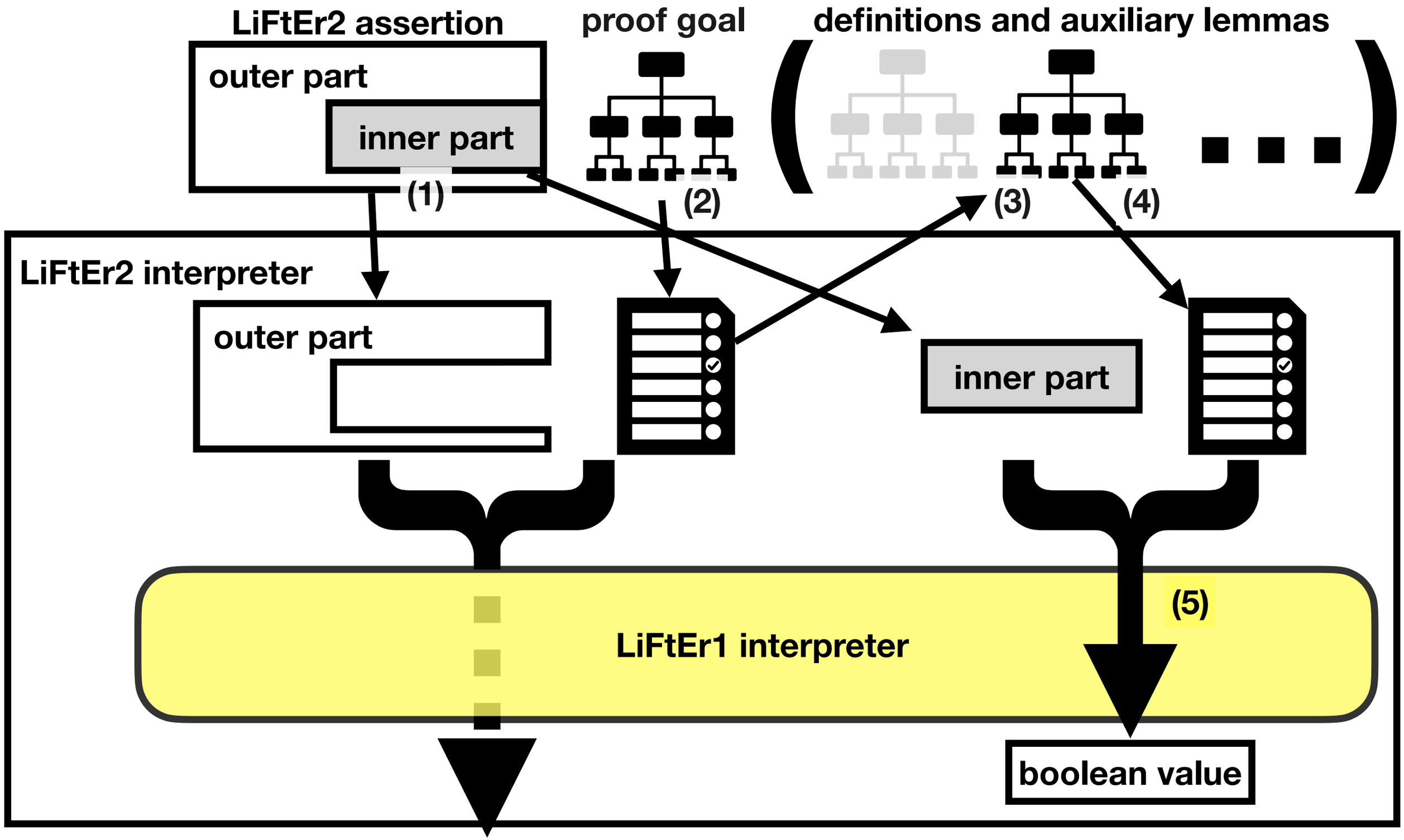
LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



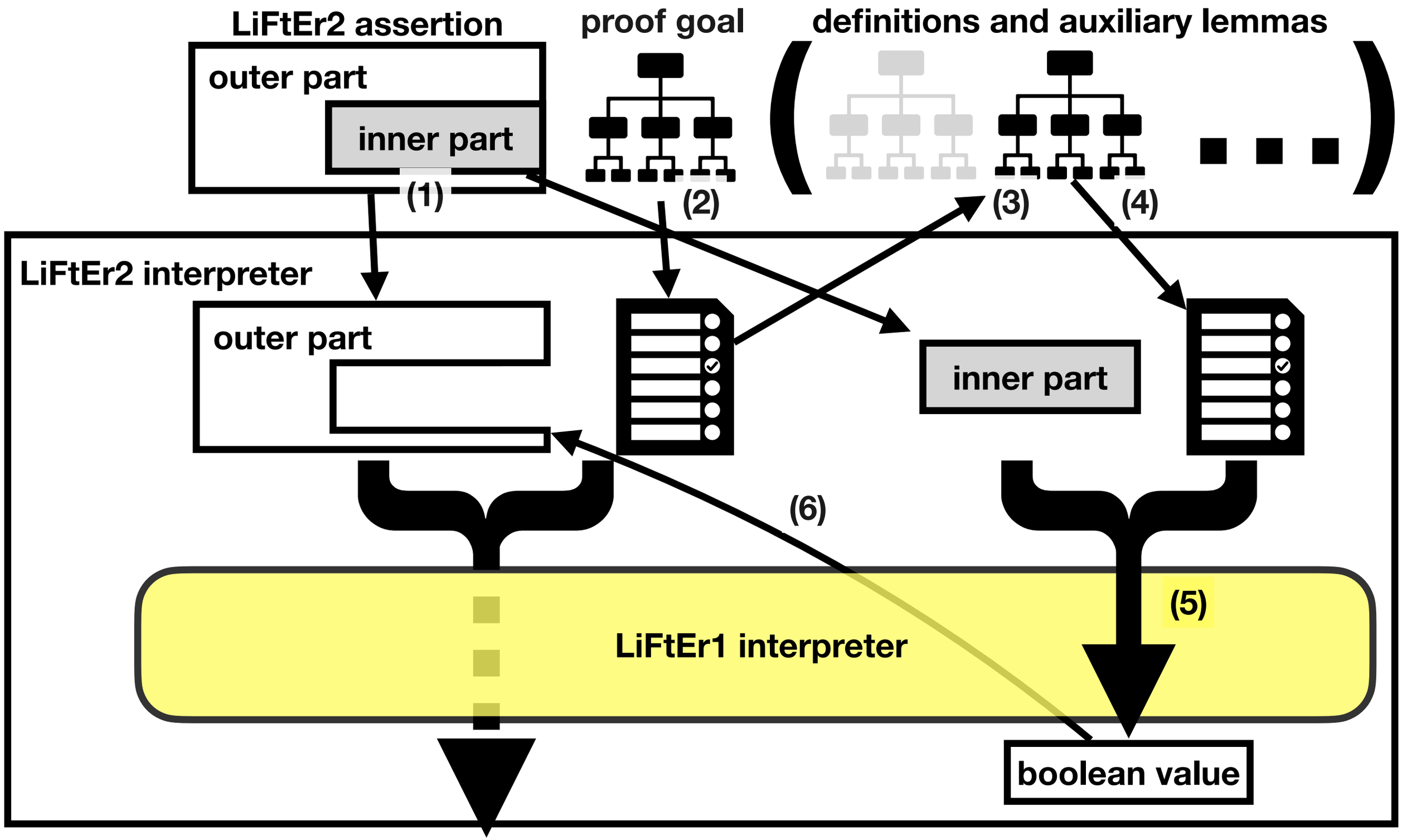
LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



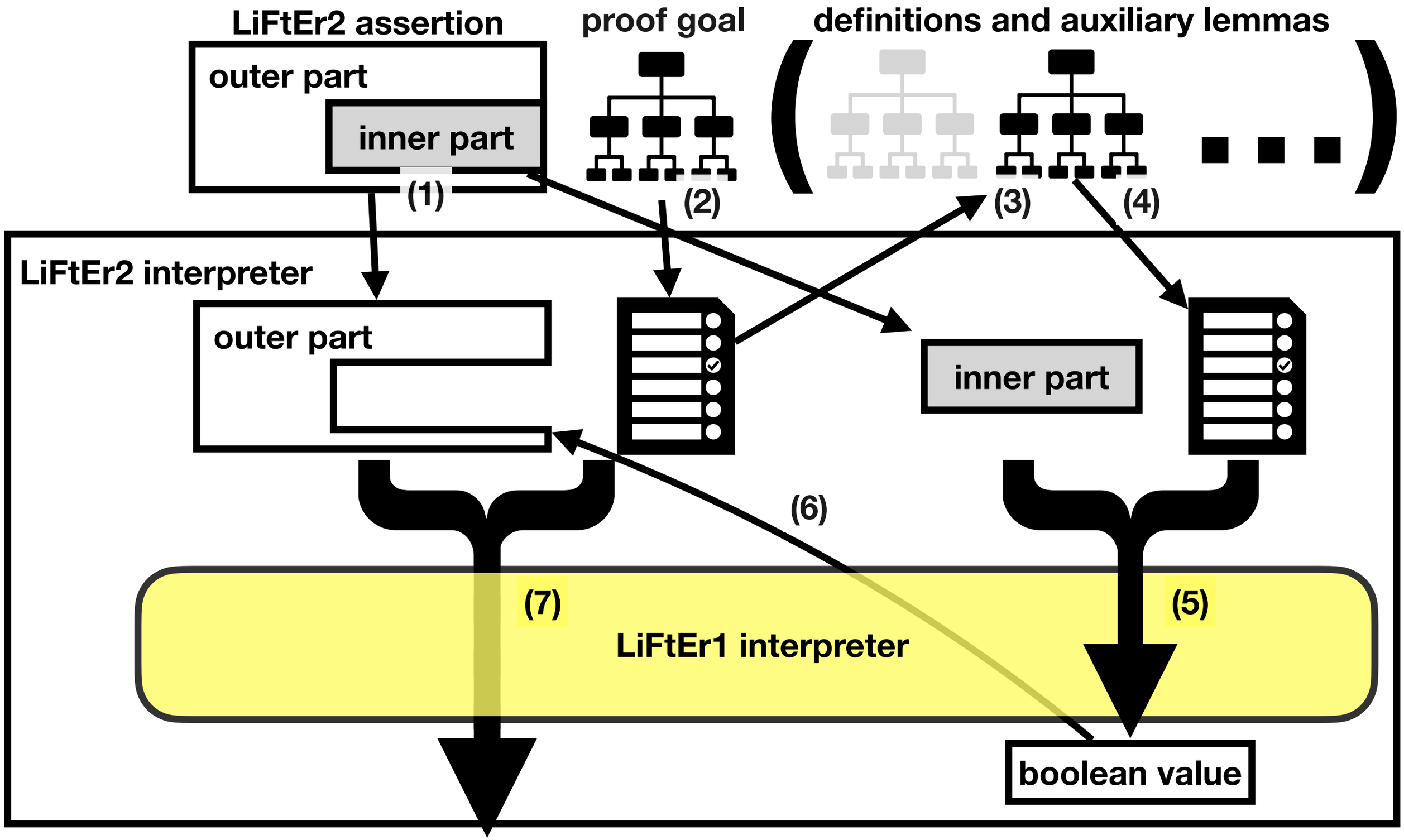
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions



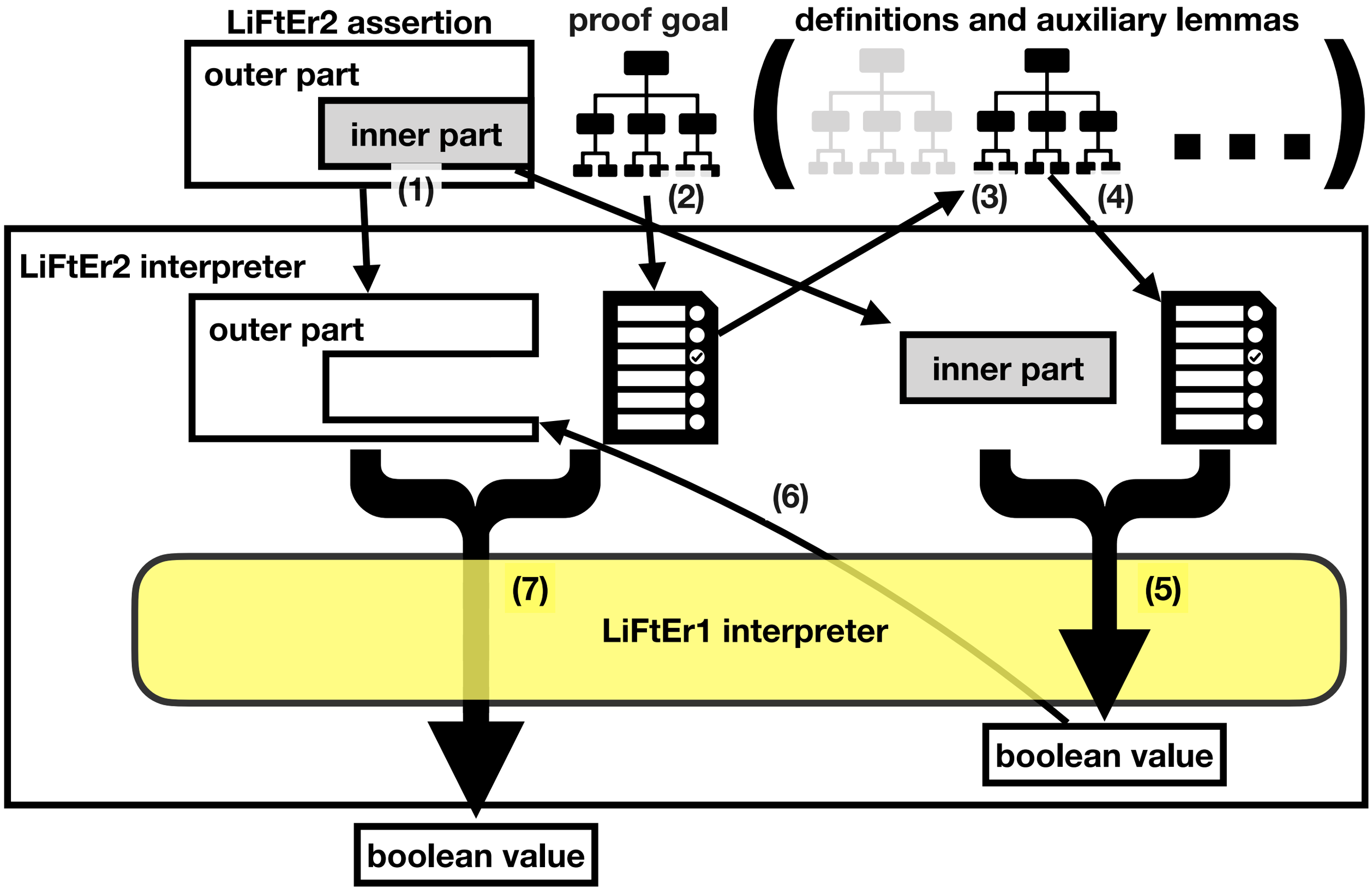
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions



LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool

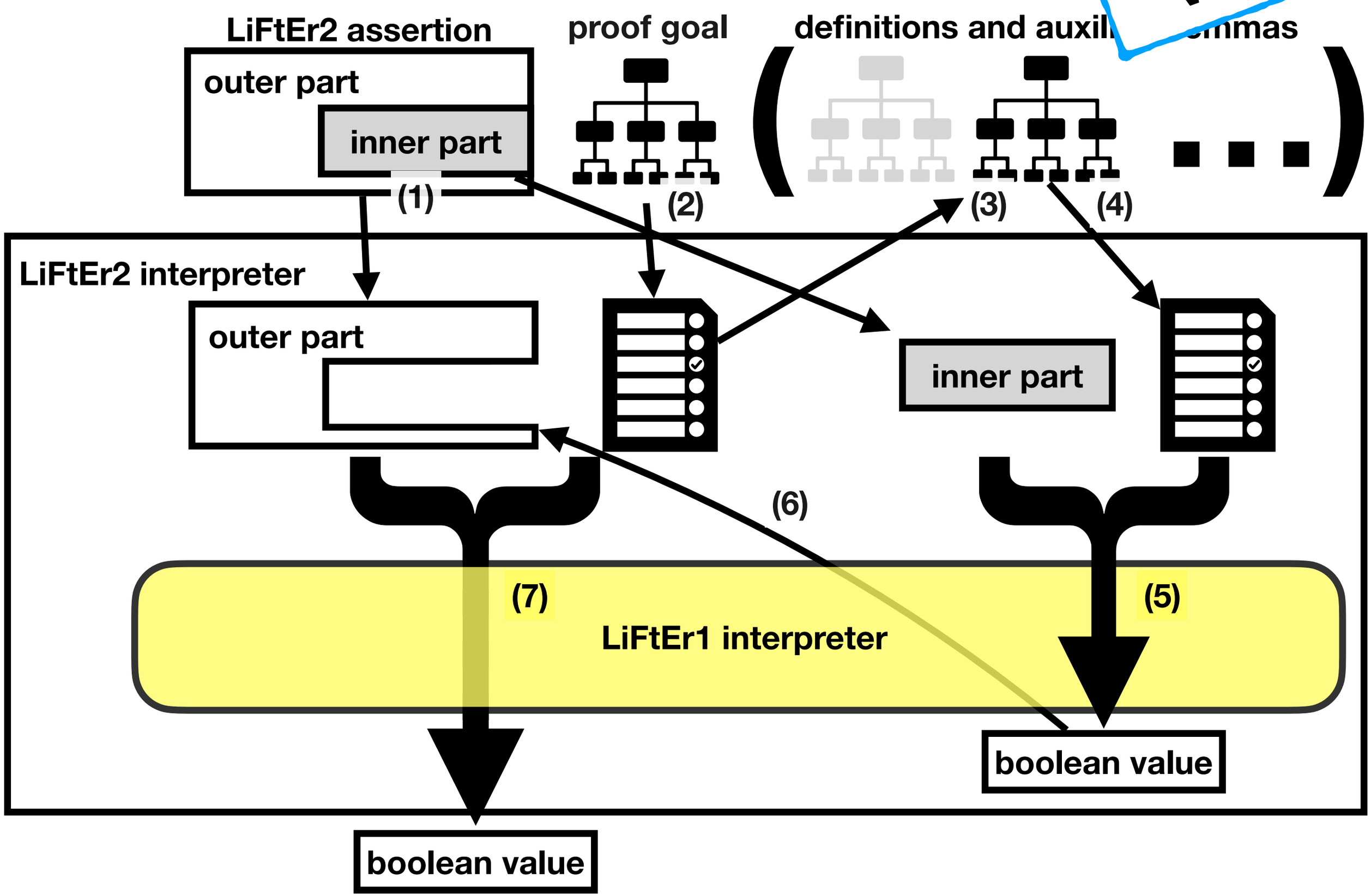


LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions

WIP!



LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool