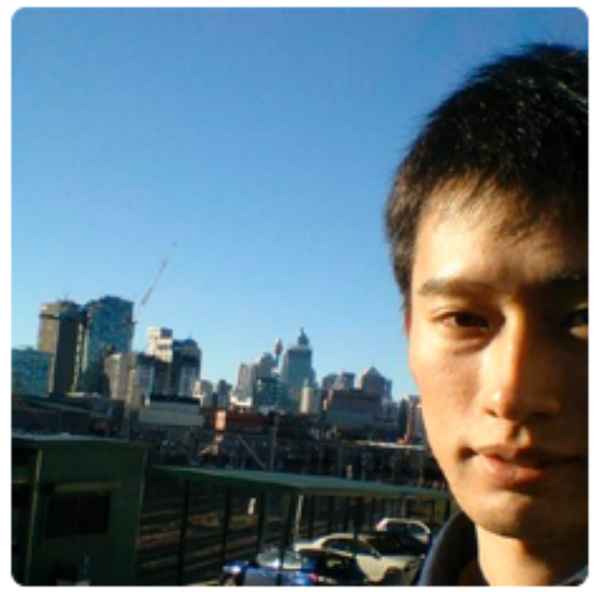
Automating proof by induction in Isabelle/HOL using DSLs

Yutaka Nagashima

universität innsbruck

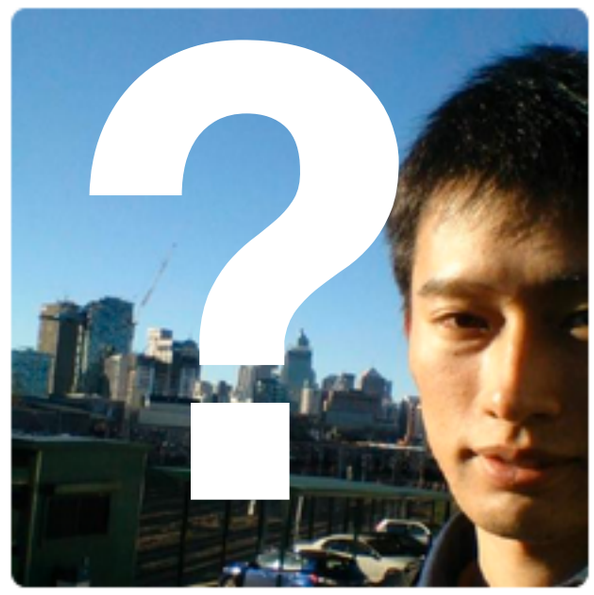
CZECH INSTITUTE OF INFORMATICS ROBOTICS AND CYBERNETICS CTU IN PRAGUE



Yutaka Ng

Automating proof by induction in Isabelle/HOL using DSLs

Yutaka Nagashima



2013 ~ 2017

with Prof. Gerwin Klein

https://en.wikipedia.org/wiki/Australia#/media/File:Australia_with_AAT_(orthographic_projection).svg

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

2013 ~ 2017

with Prof. Gerwin Klein

pre-PhD

https://en.wikipedia.org/wiki/Australia#/media/File:Australia_with_AAT_(orthographic_projection).svg

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

2013 ~ 2017

with Prof. Gerwin Klein

pre-PhD

PhD in
AI for theorem proving

https://en.wikipedia.org/wiki/Australia#/media/File:Australia_with_AAT_(orthographic_projection).svg

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

2013 ~ 2017

with Prof. Gerwin Klein

https://en.wikipedia.org/wiki/Australia#/media/File:Australia_with_AAT_(orthographic_projection).svg

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

pre-PhD

PhD in
AI for theorem proving

2017 ~ 2018

with Prof. Cezary Kaliszyk

https://en.wikipedia.org/wiki/File:EU-Austria.svg

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

2013 ~ 2017

with Prof. Gerwin Klein

https://en.wikipedia.org/wiki/Australia#/media/File:Australia_with_AAT_(orthographic_projection).svg

Isabelle HOL

(seL4)

Security. Performance. Proof.

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

pre-PhD

PhD in
AI for theorem proving

2017 ~ 2018

with Prof. Cezary Kaliszyk

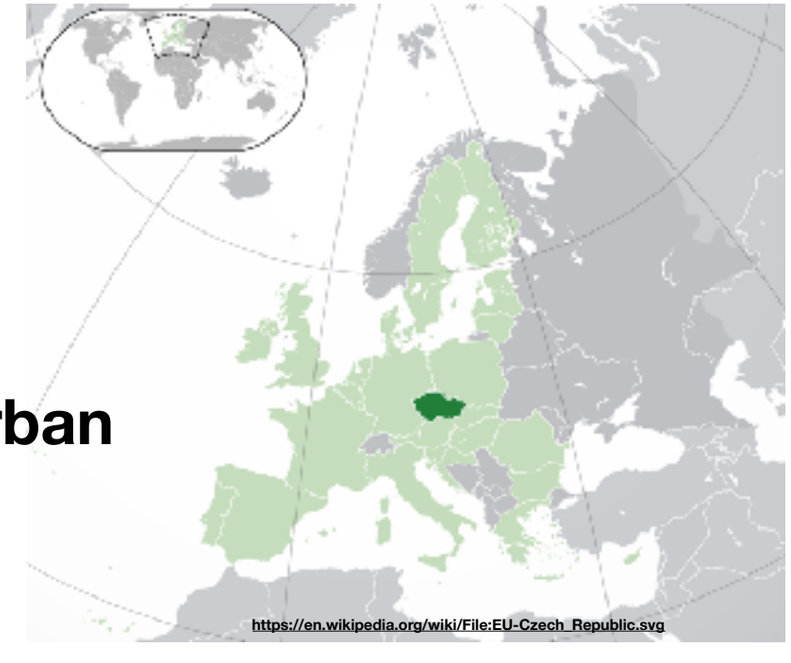
https://en.wikipedia.org/wiki/File:EU-Austria.svg

Isabelle HOL

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

2018 ~ 2020

with Dr. Josef Urban



https://en.wikipedia.org/wiki/File:EU-Czech_Republic.svg

Isabelle HOL

http://ai4reason.org/members.html

PSL: Proof Strategy Language

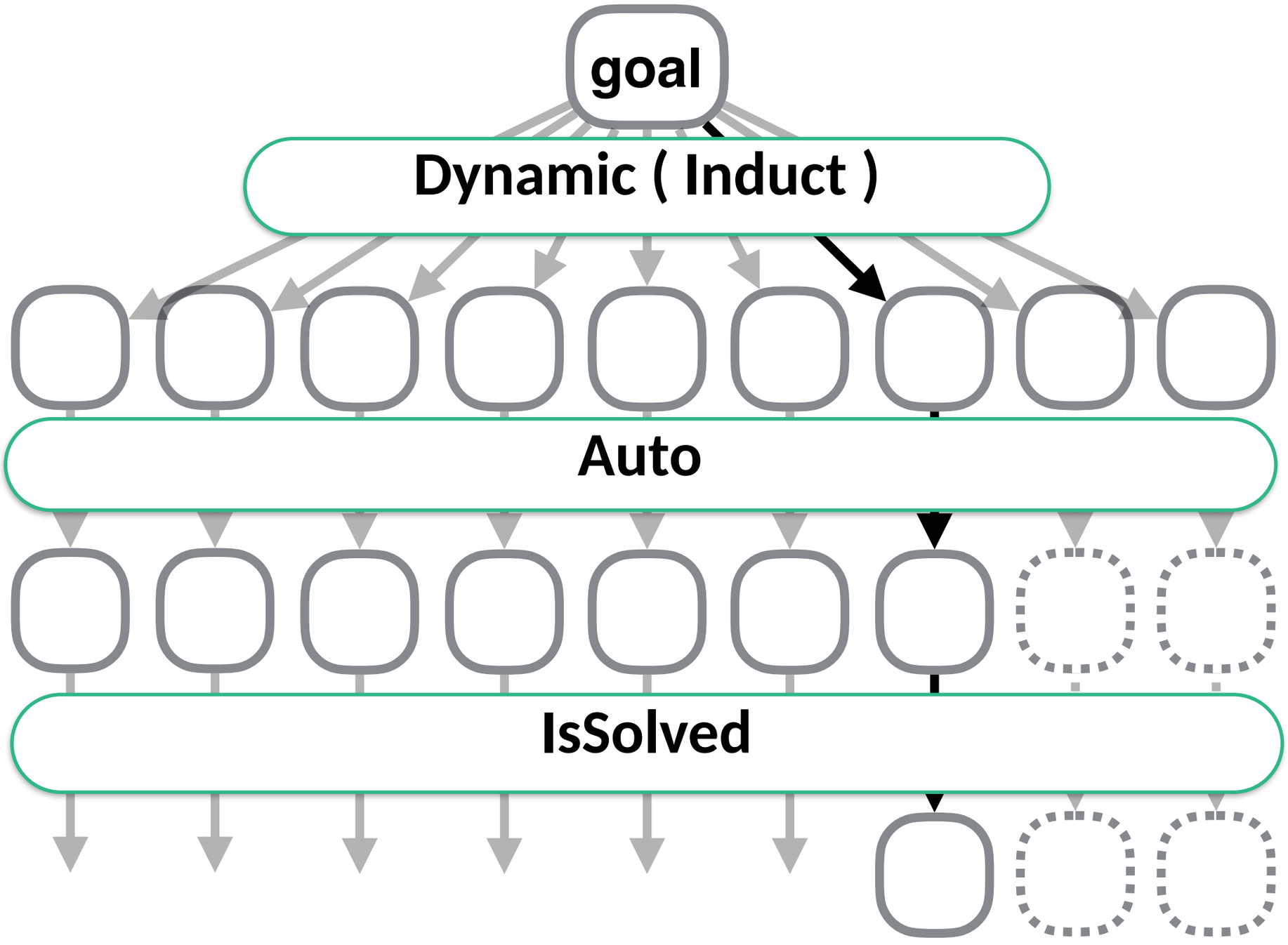
PSL: Proof Strategy Lang

DEMO!

PSL: Proof Strategy Language

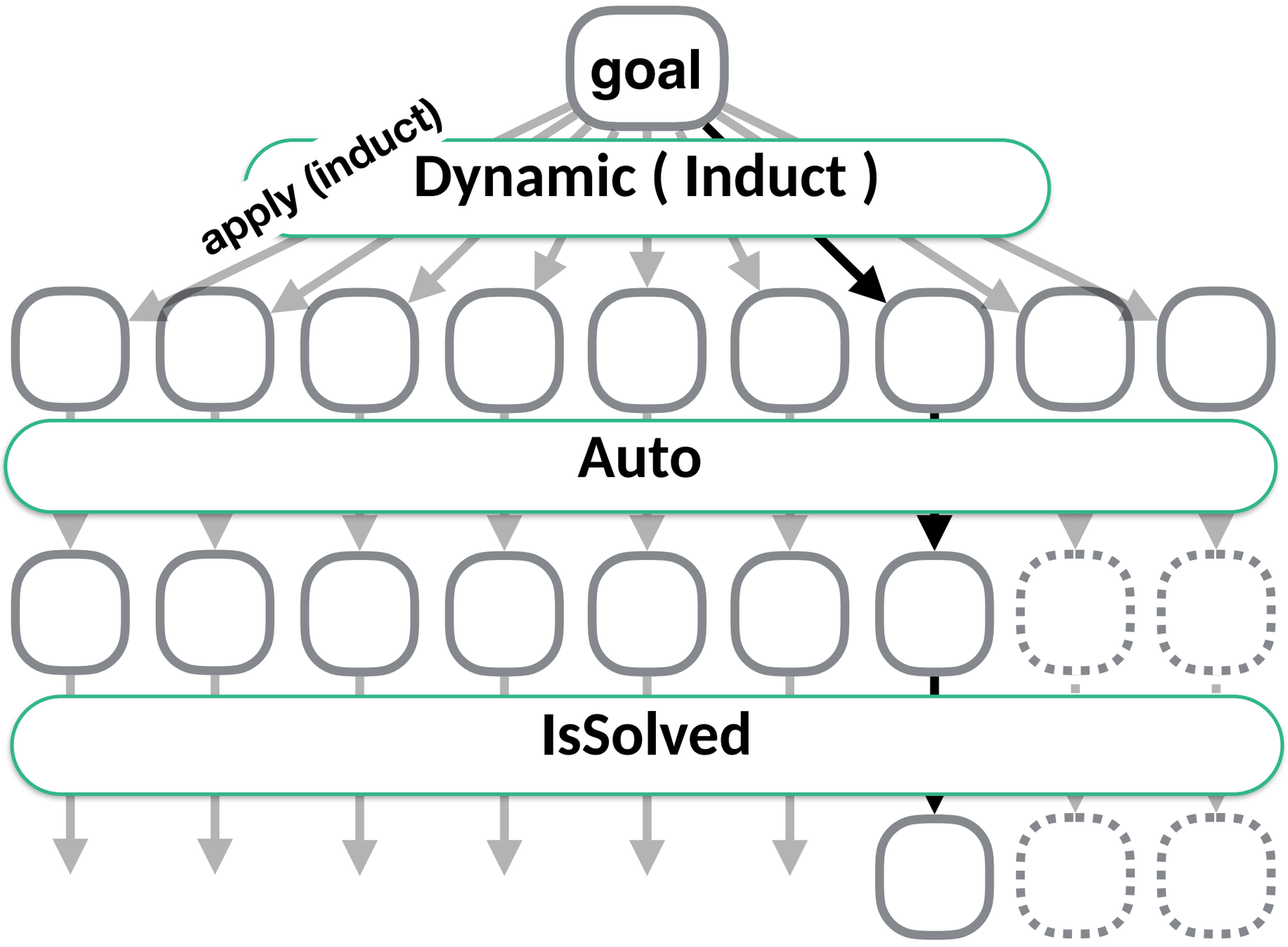
PSL: Proof Strategy Language

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



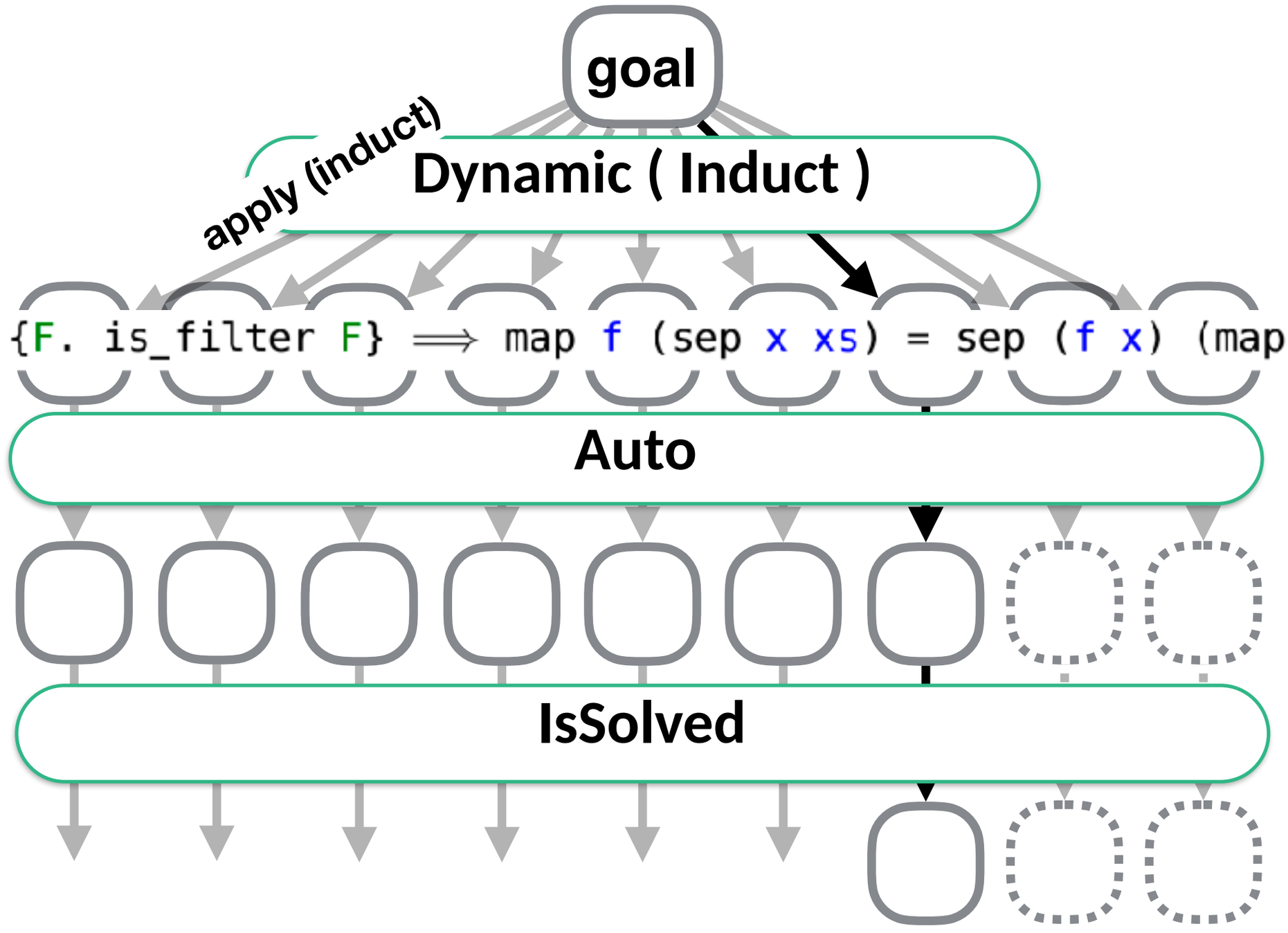
PSL: Proof Strategy Language

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



PSL: Proof Strategy Language

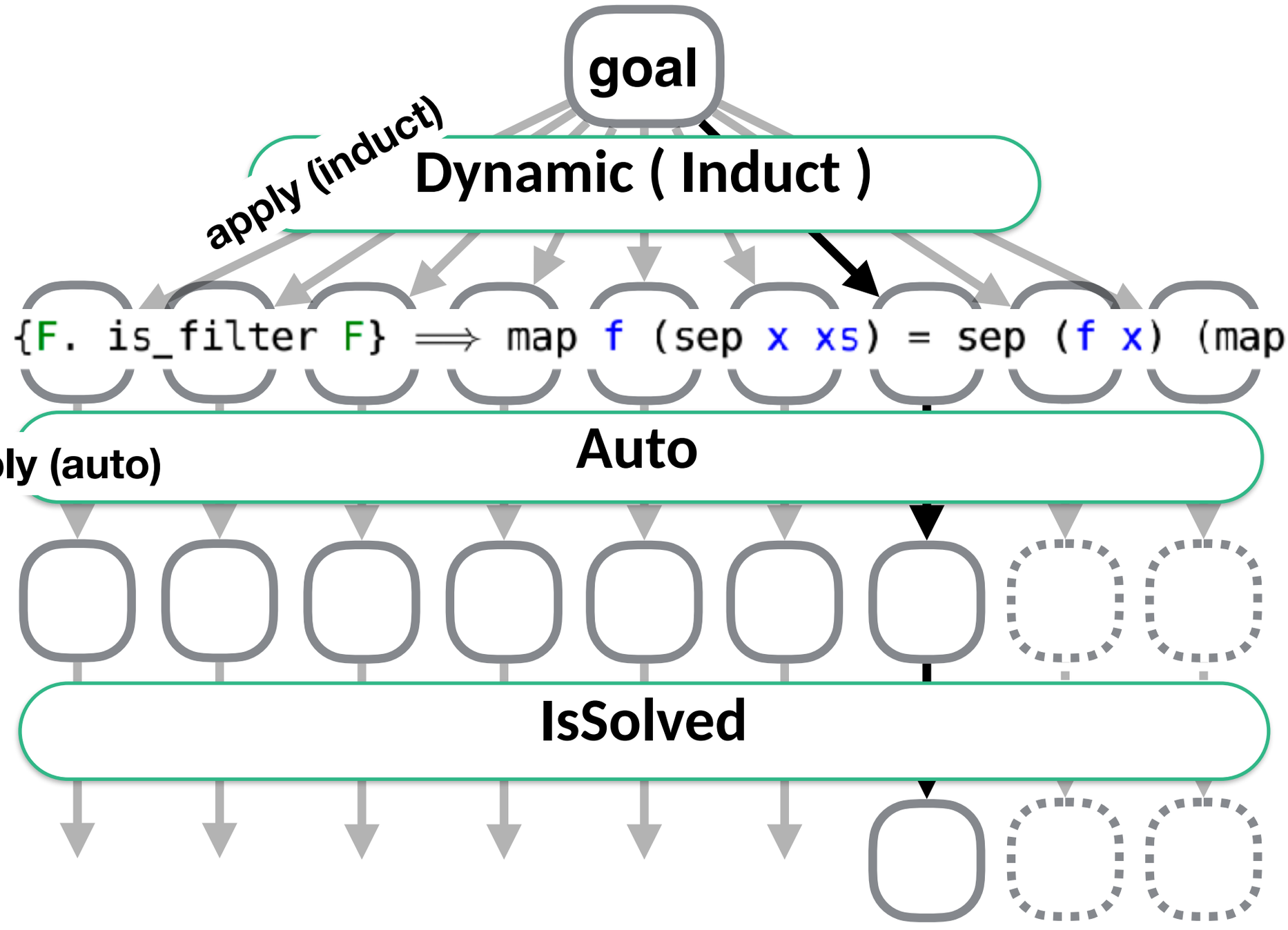
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```




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

PSL: Proof Strategy Language

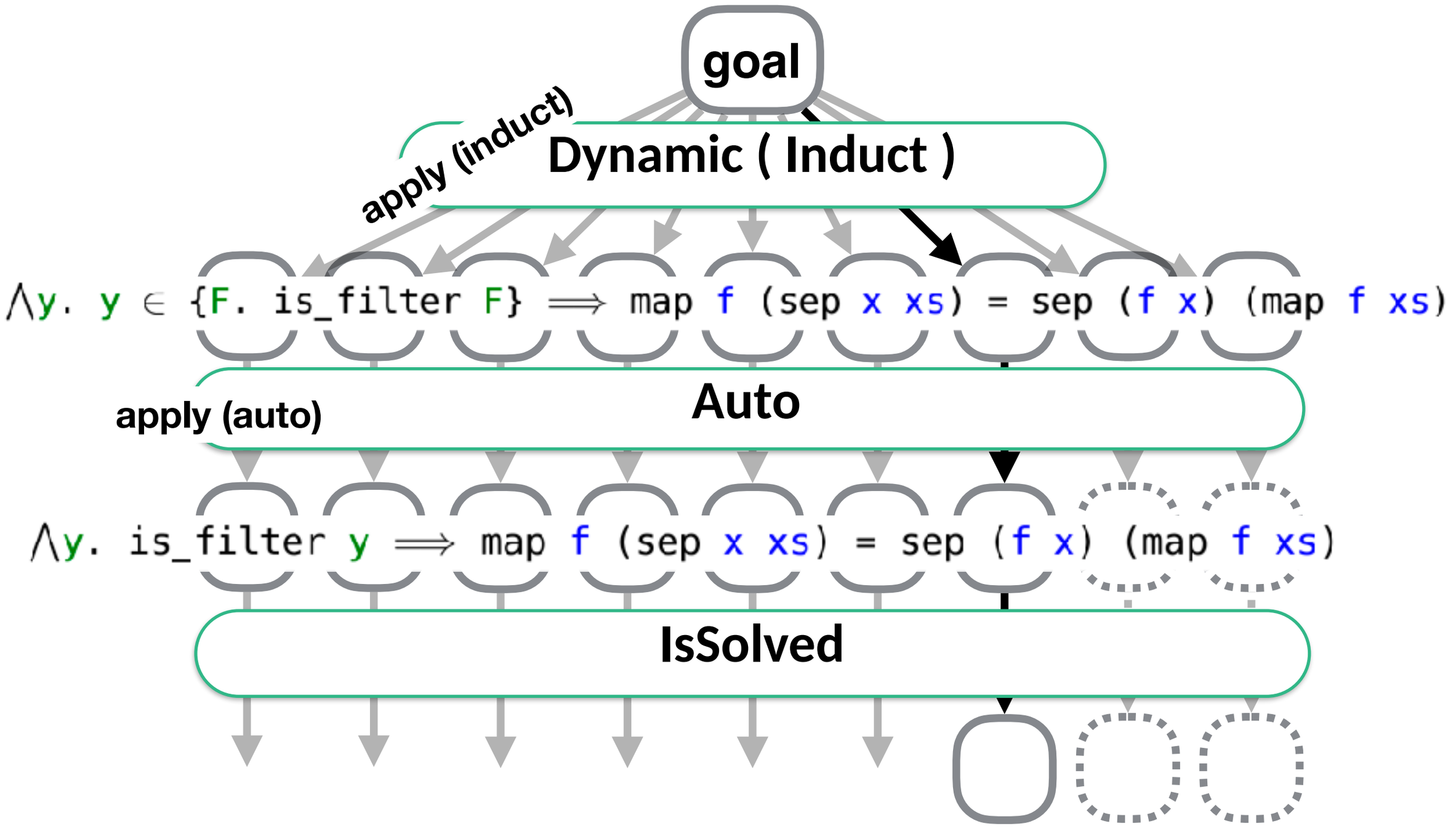
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```




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

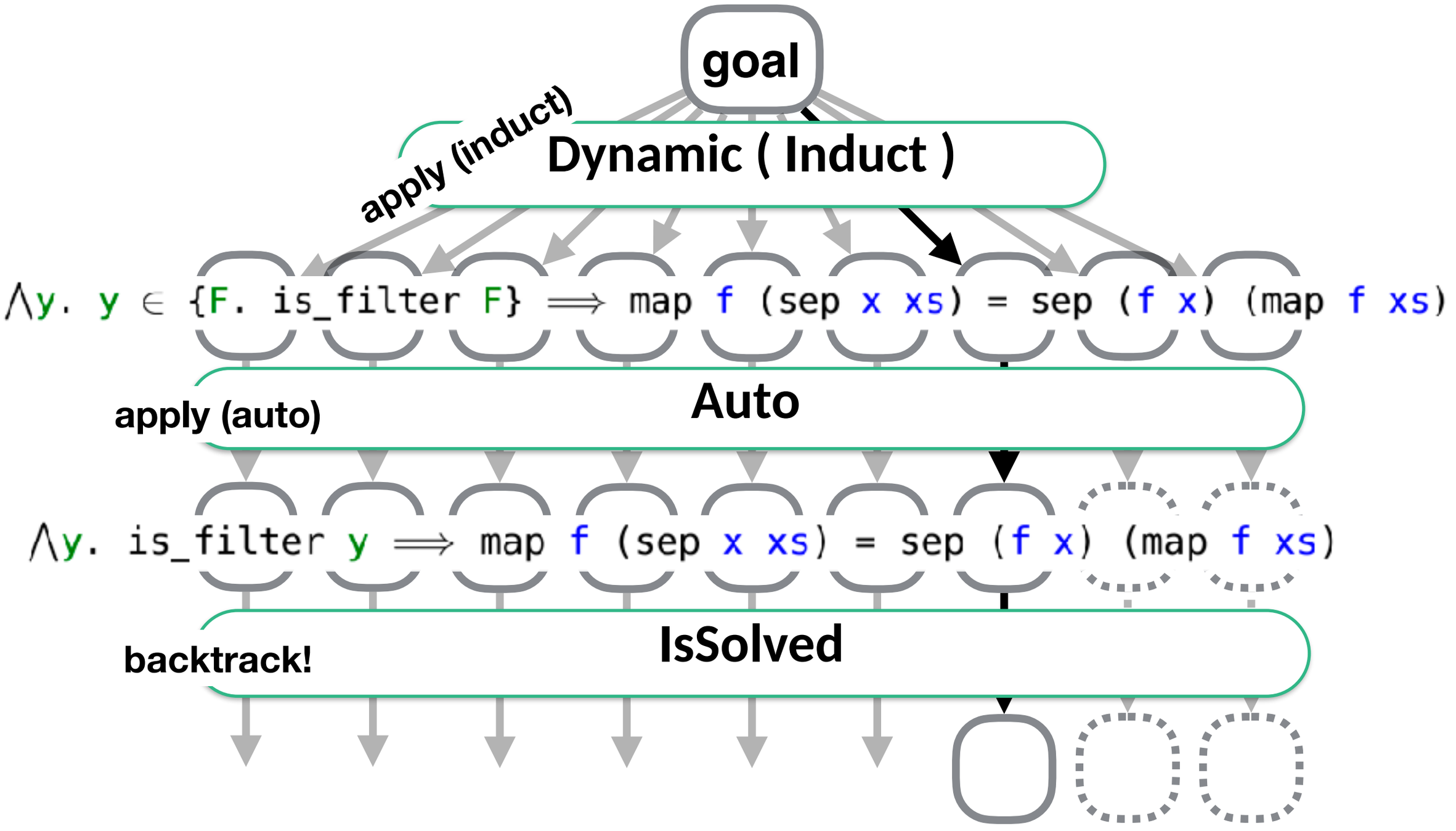
PSL: Proof Strategy Language

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



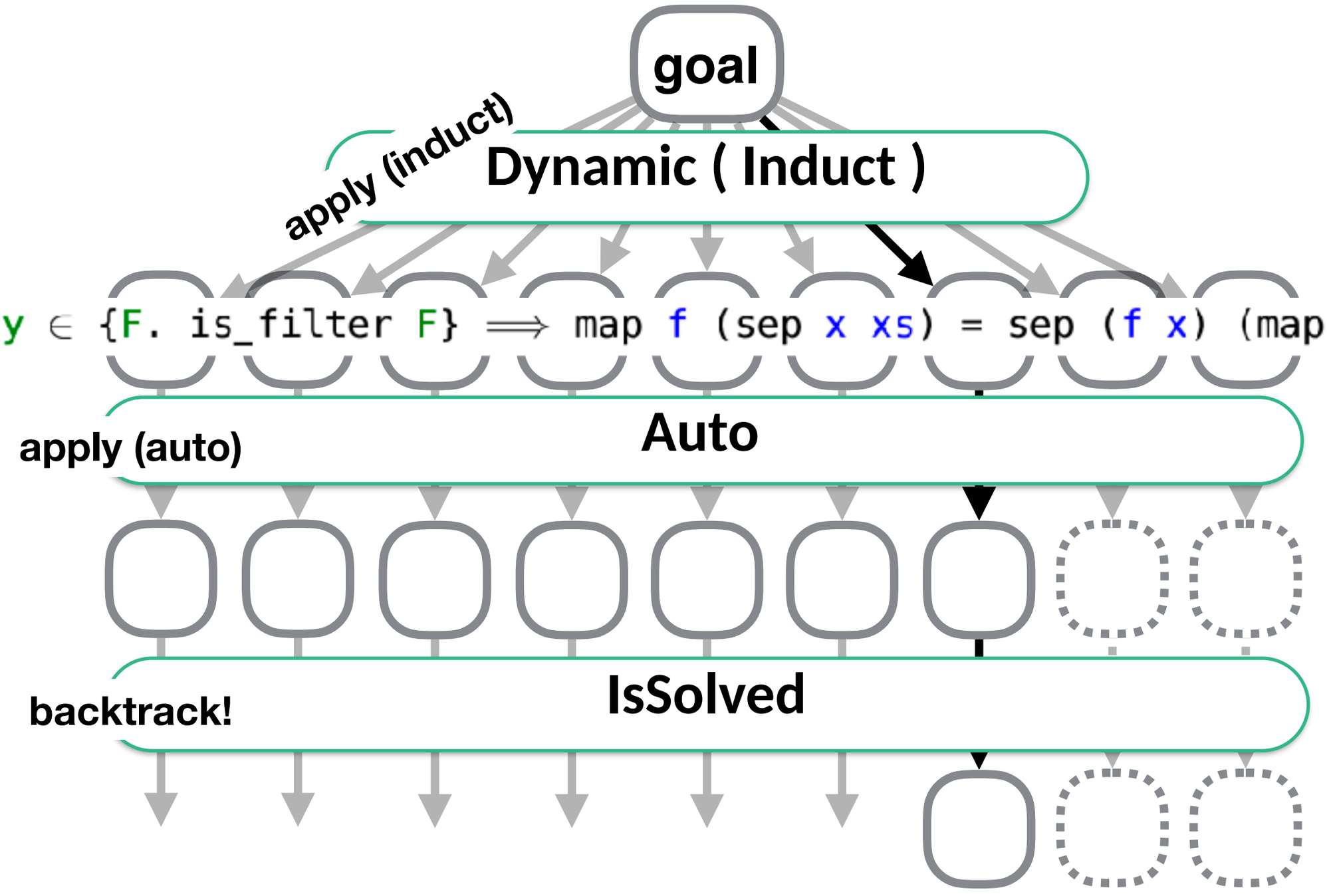
PSL: Proof Strategy Language

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



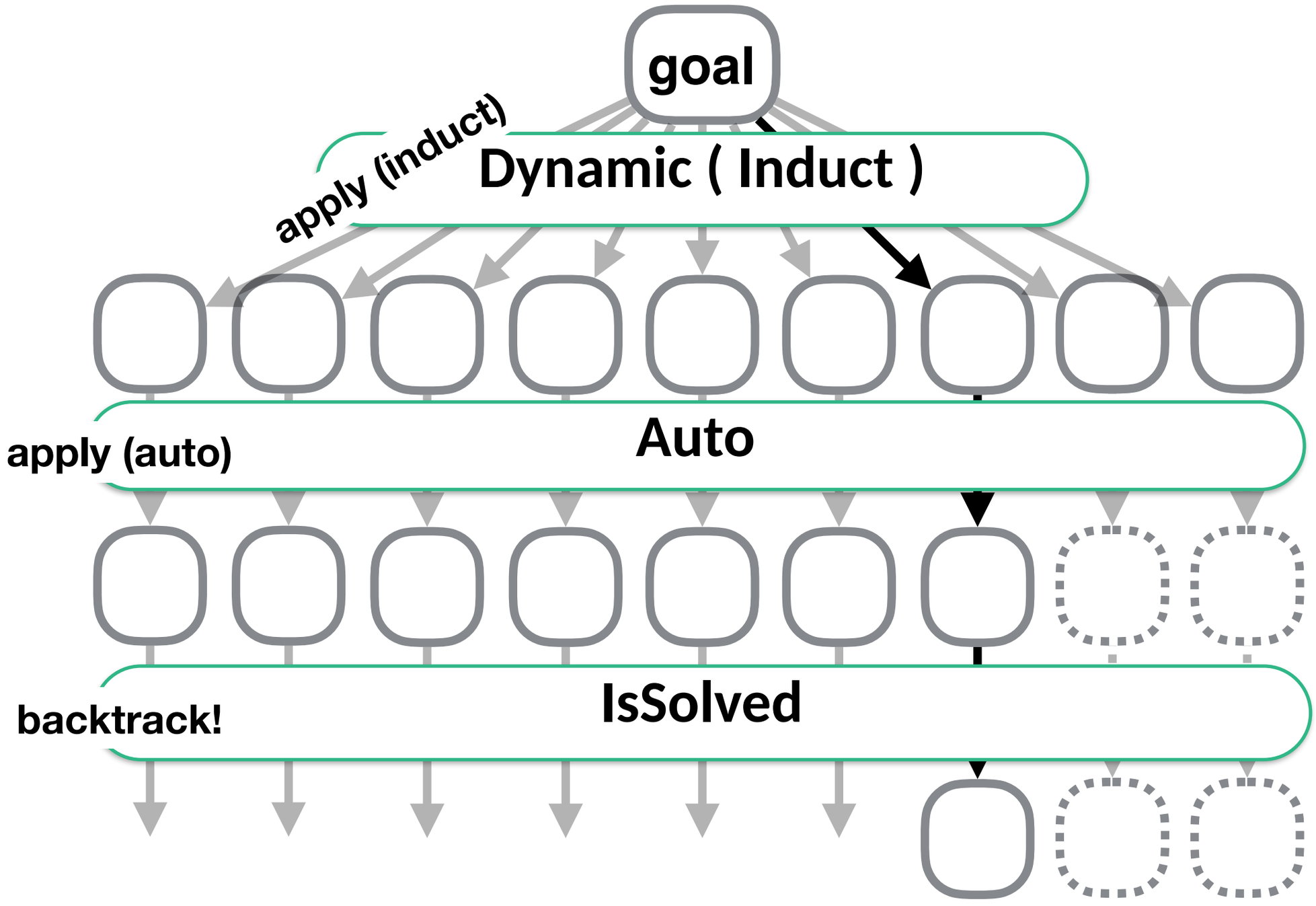
PSL: Proof Strategy Language

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



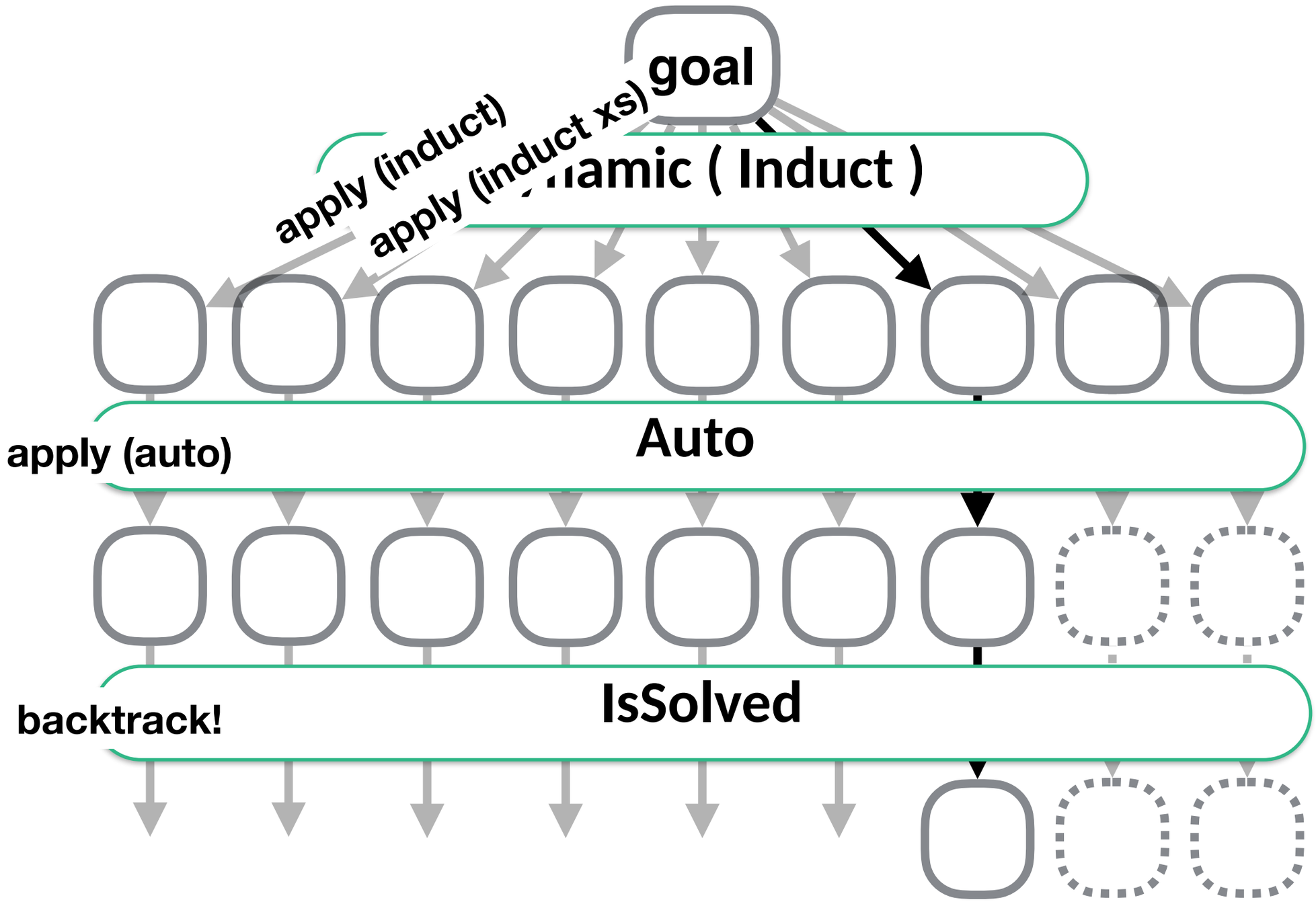
PSL: Proof Strategy Language

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



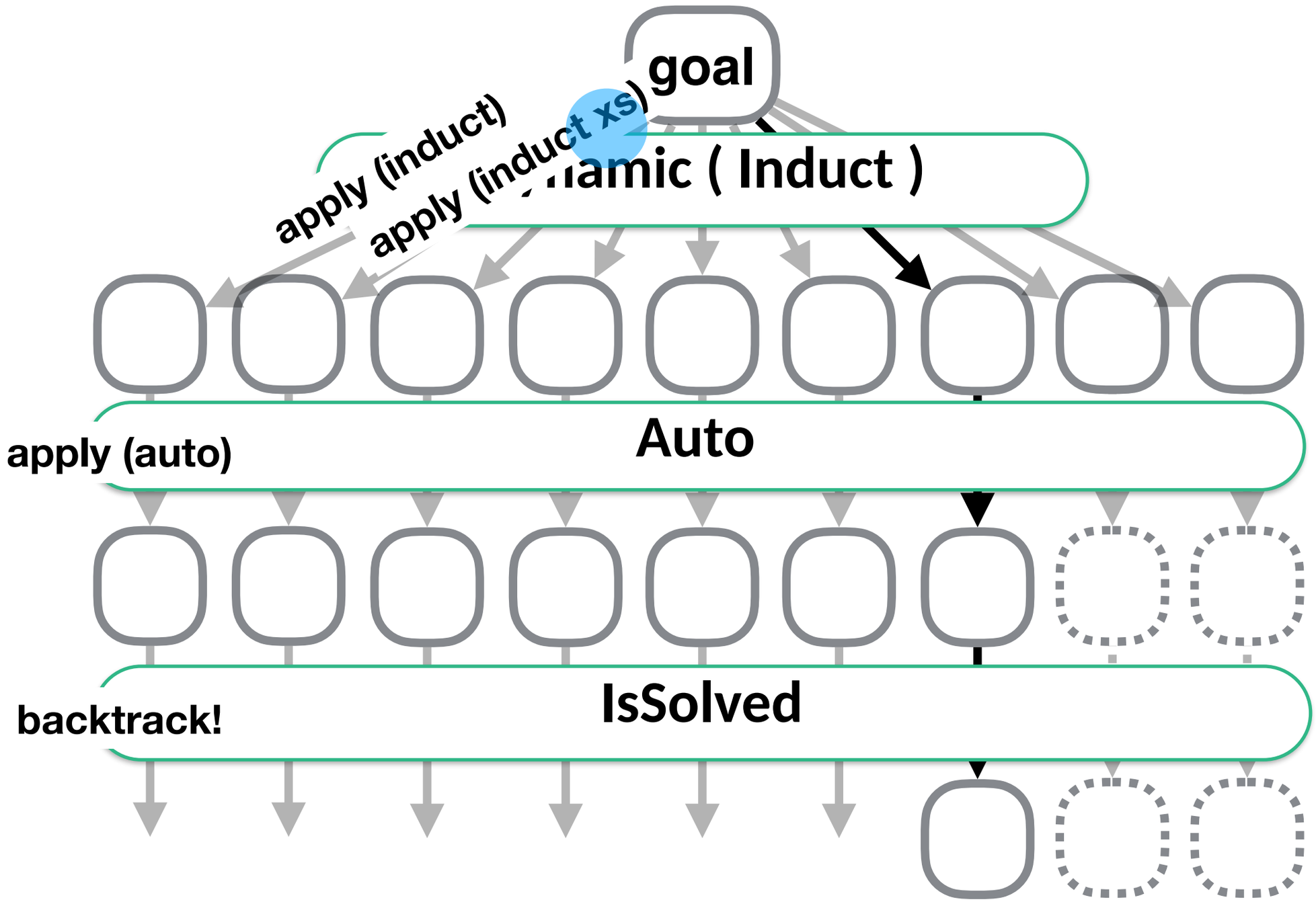
PSL: Proof Strategy Language

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



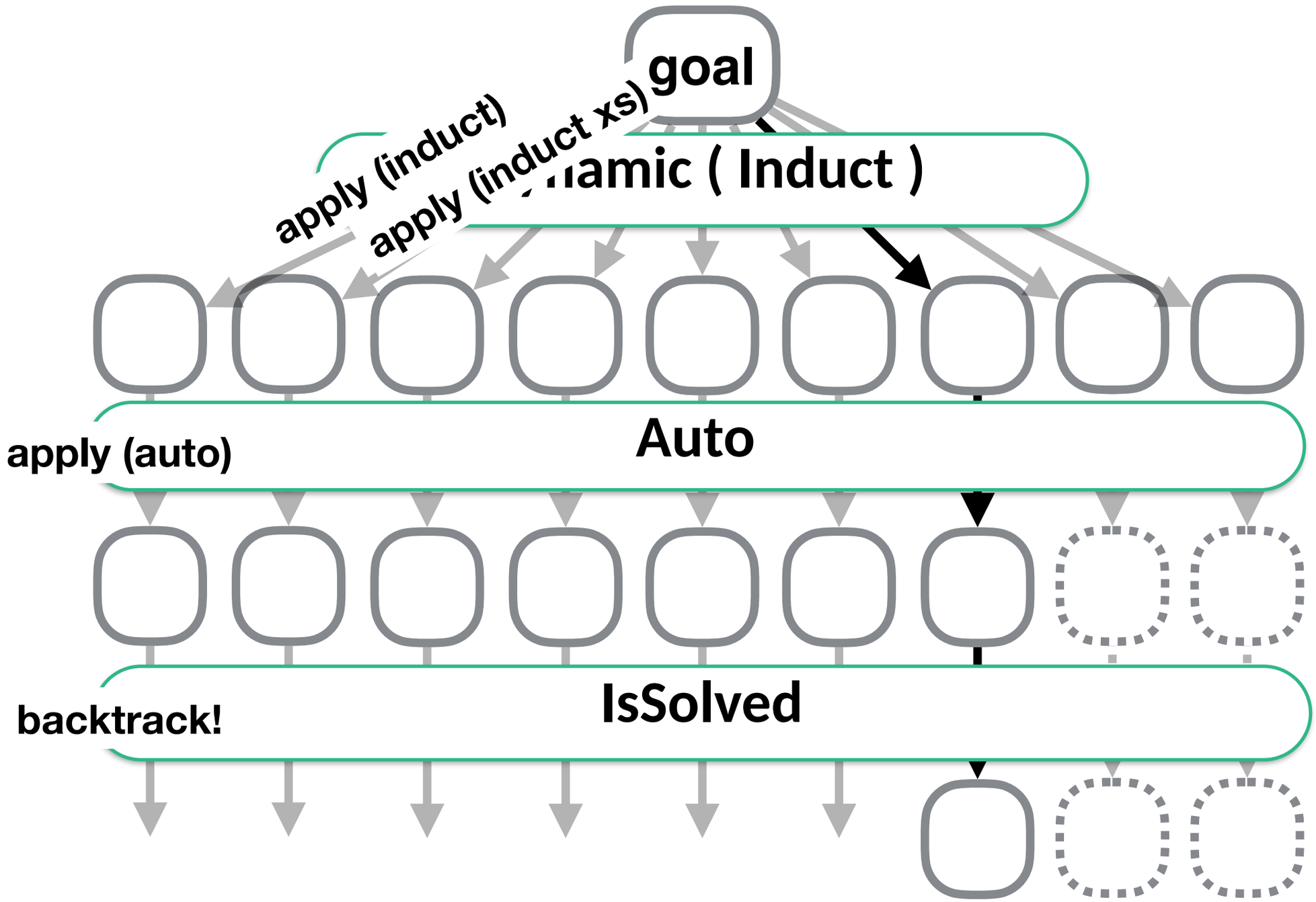
PSL: Proof Strategy Language

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



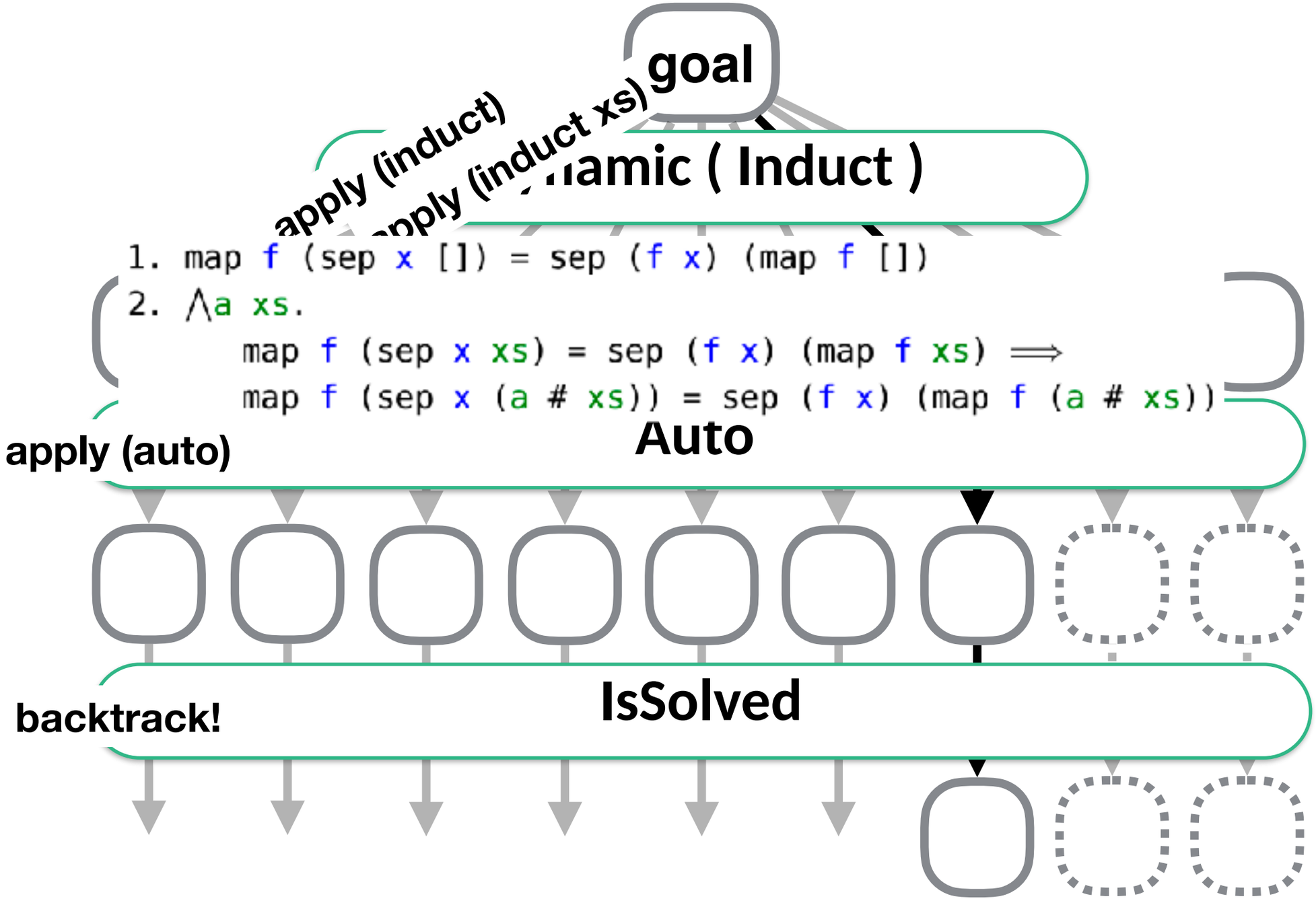
PSL: Proof Strategy Language

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



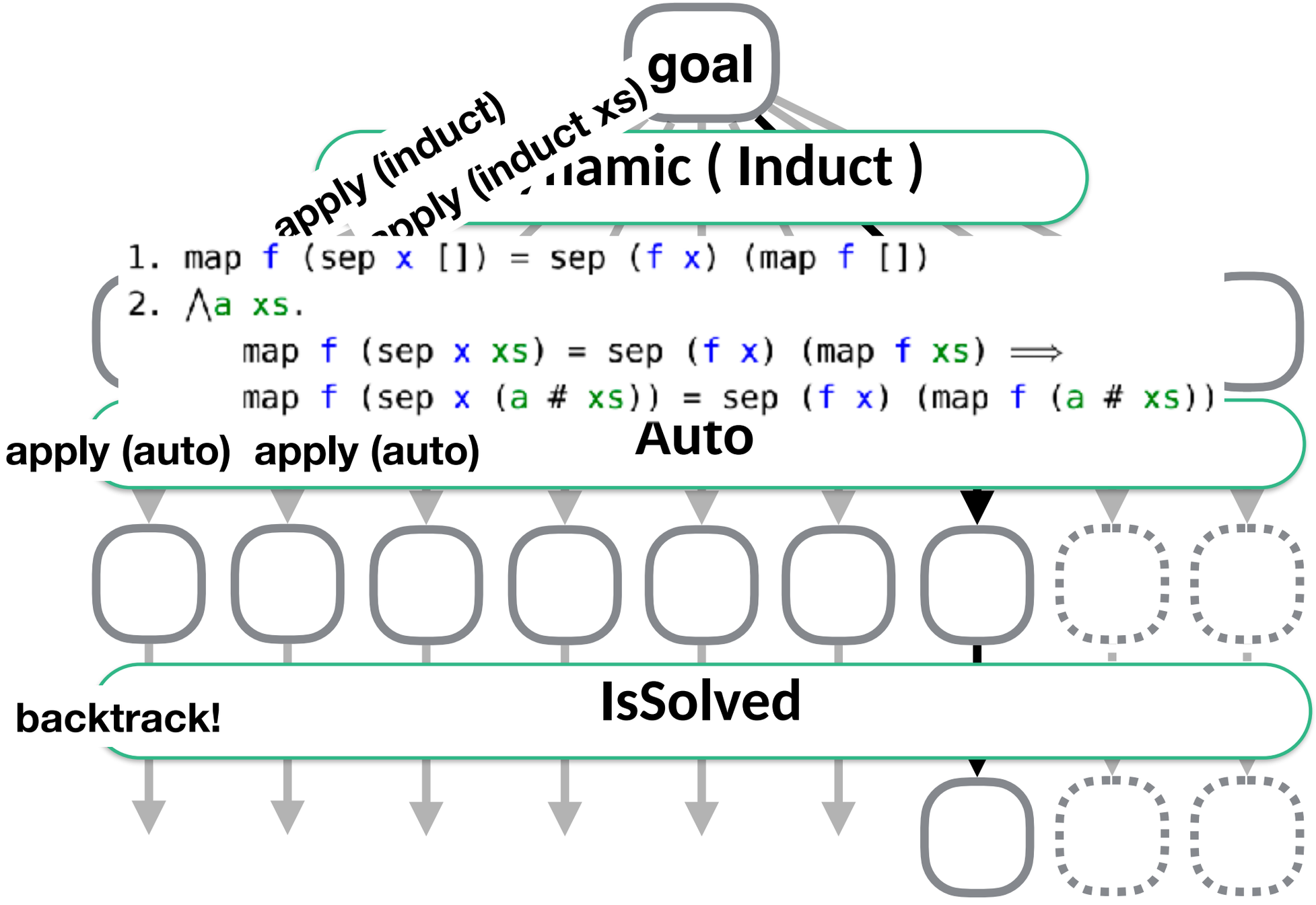
PSL: Proof Strategy Language

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



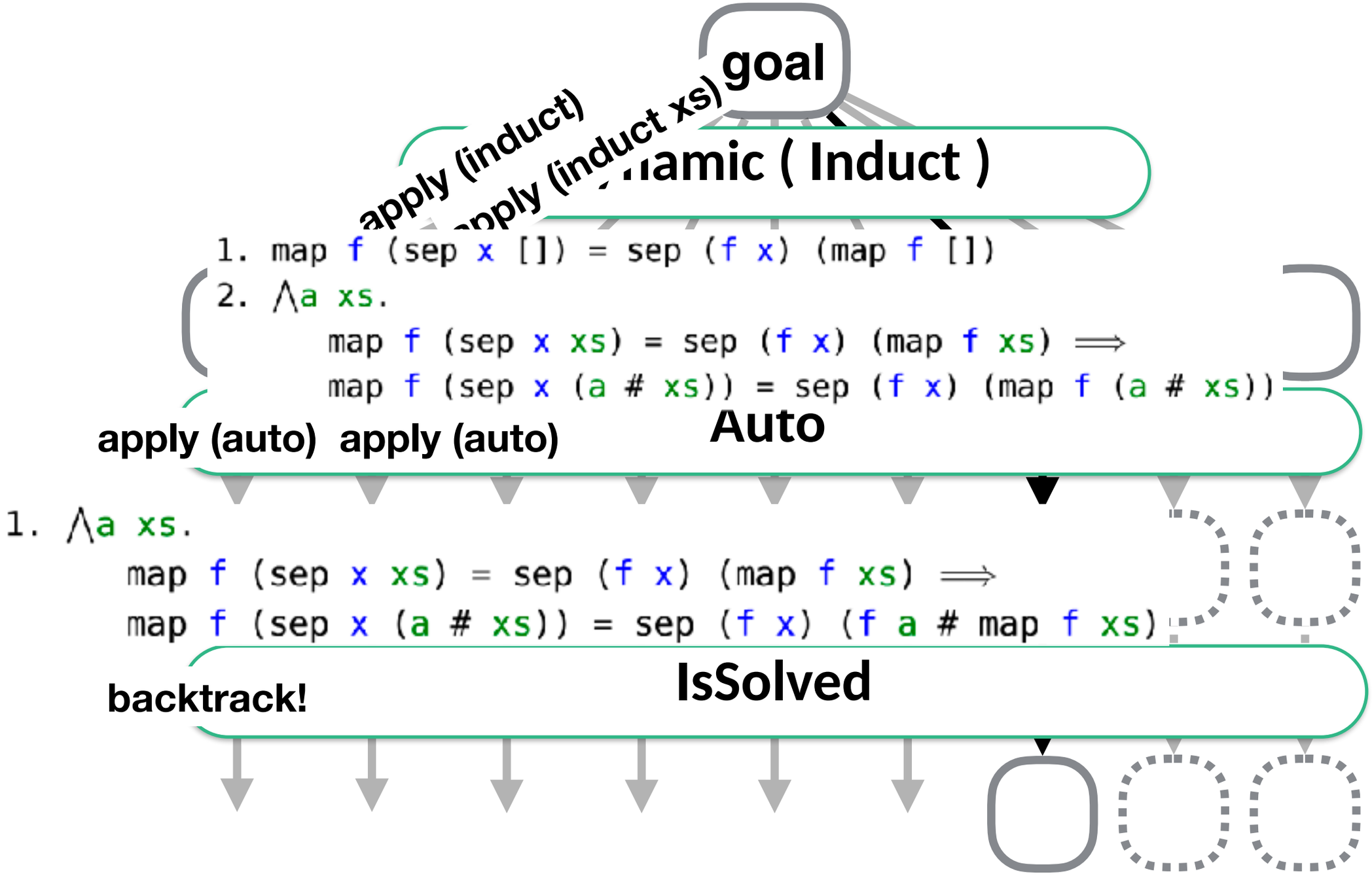
PSL: Proof Strategy Language

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



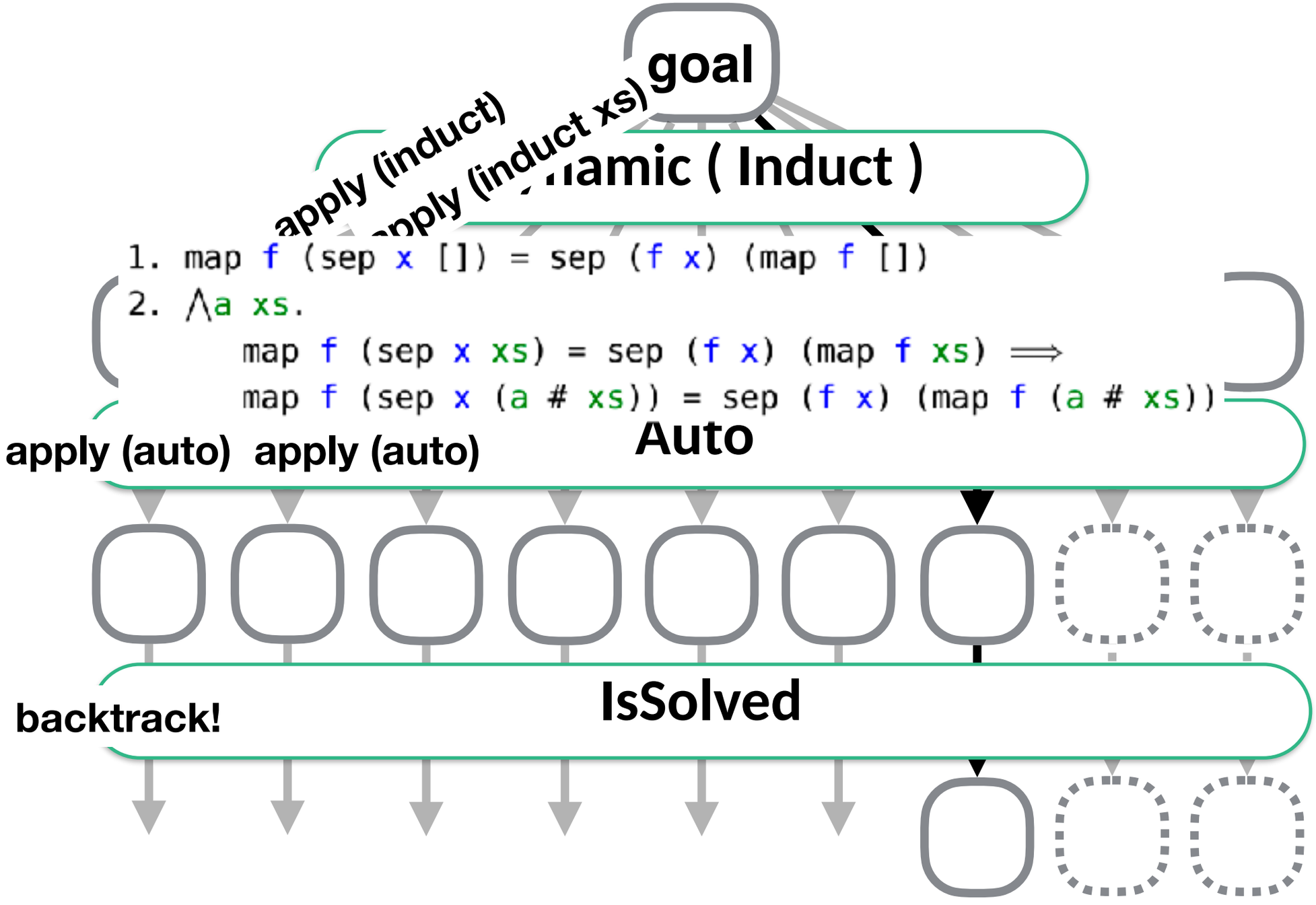
PSL: Proof Strategy Language

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



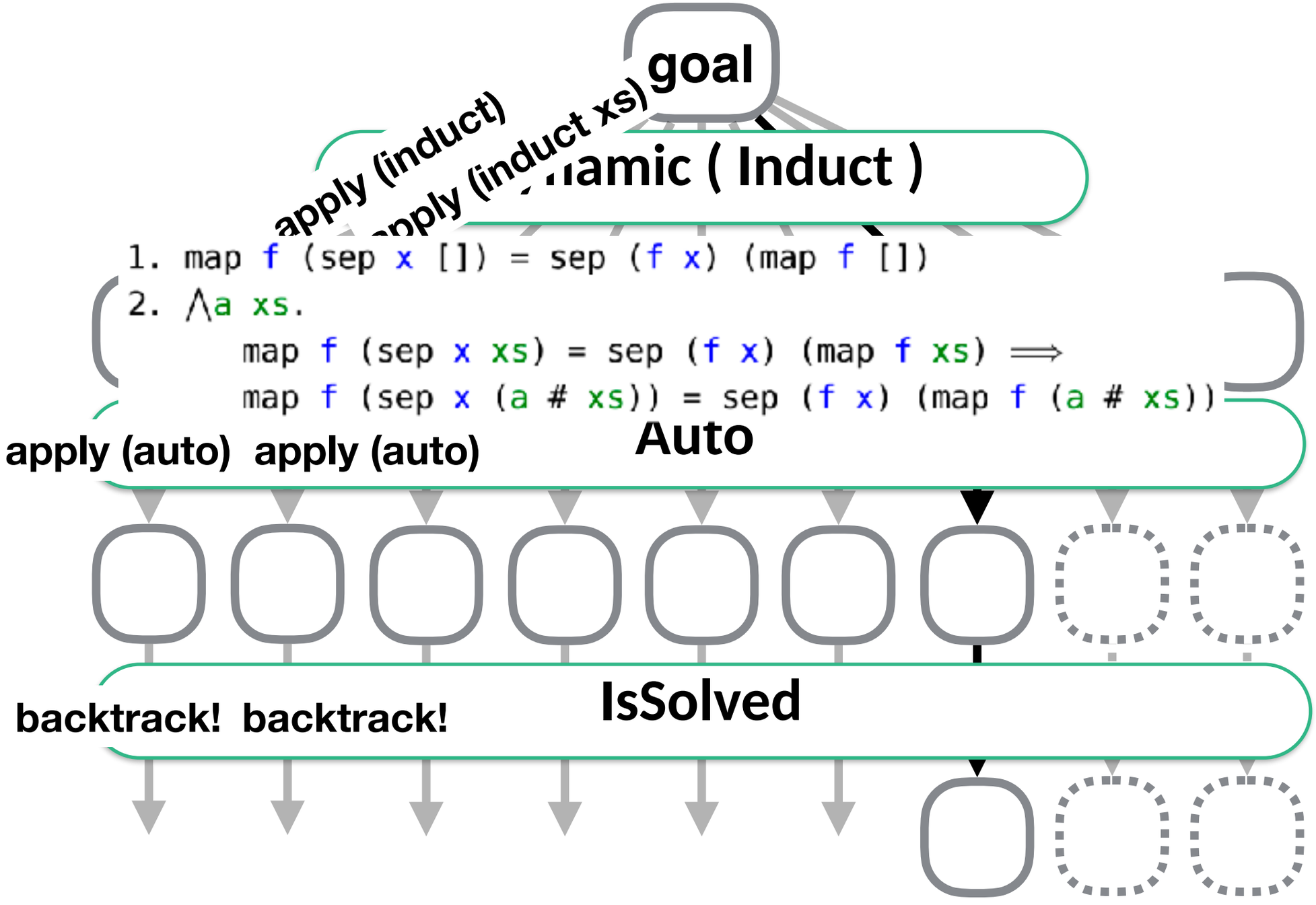
PSL: Proof Strategy Language

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



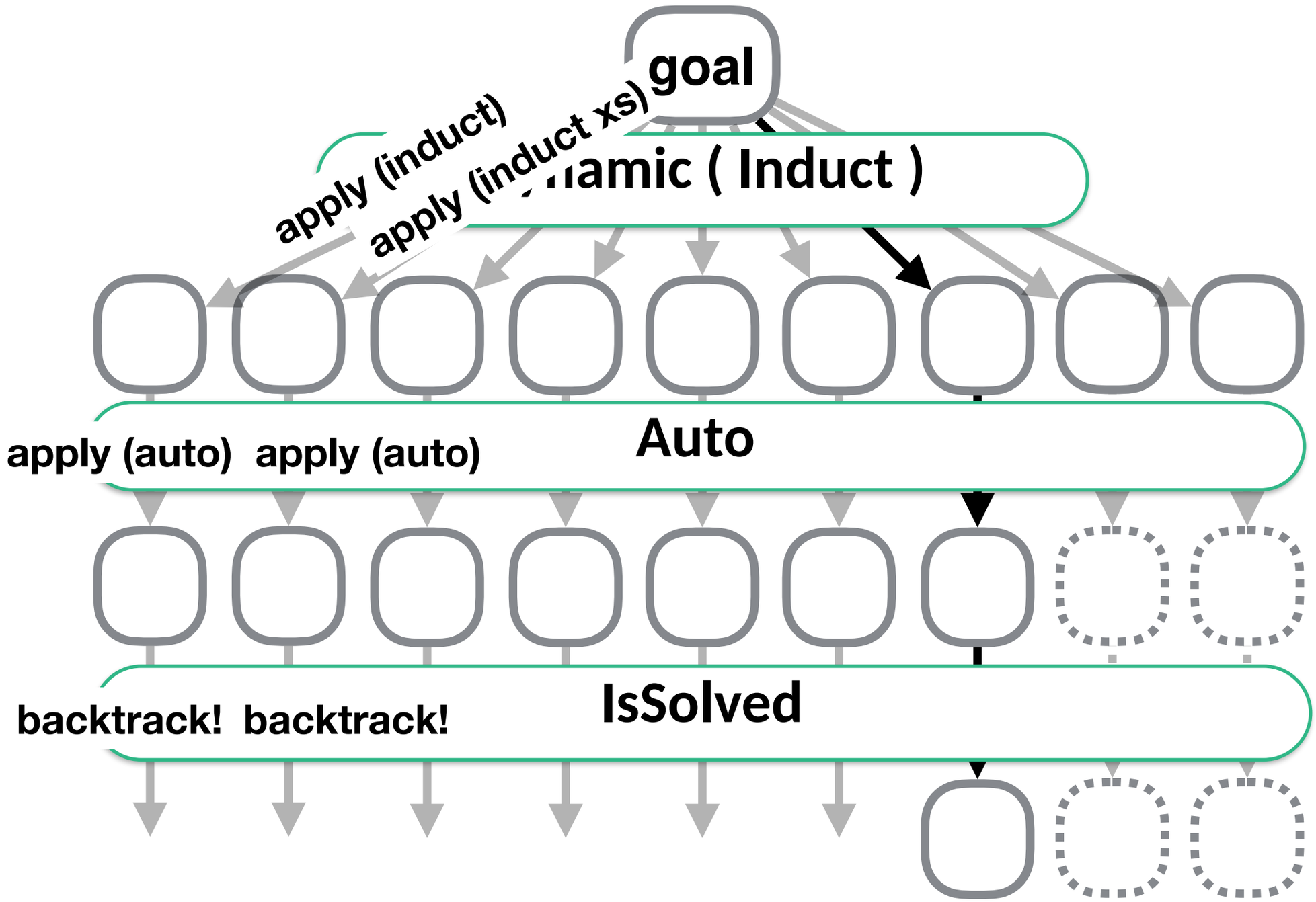
PSL: Proof Strategy Language

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



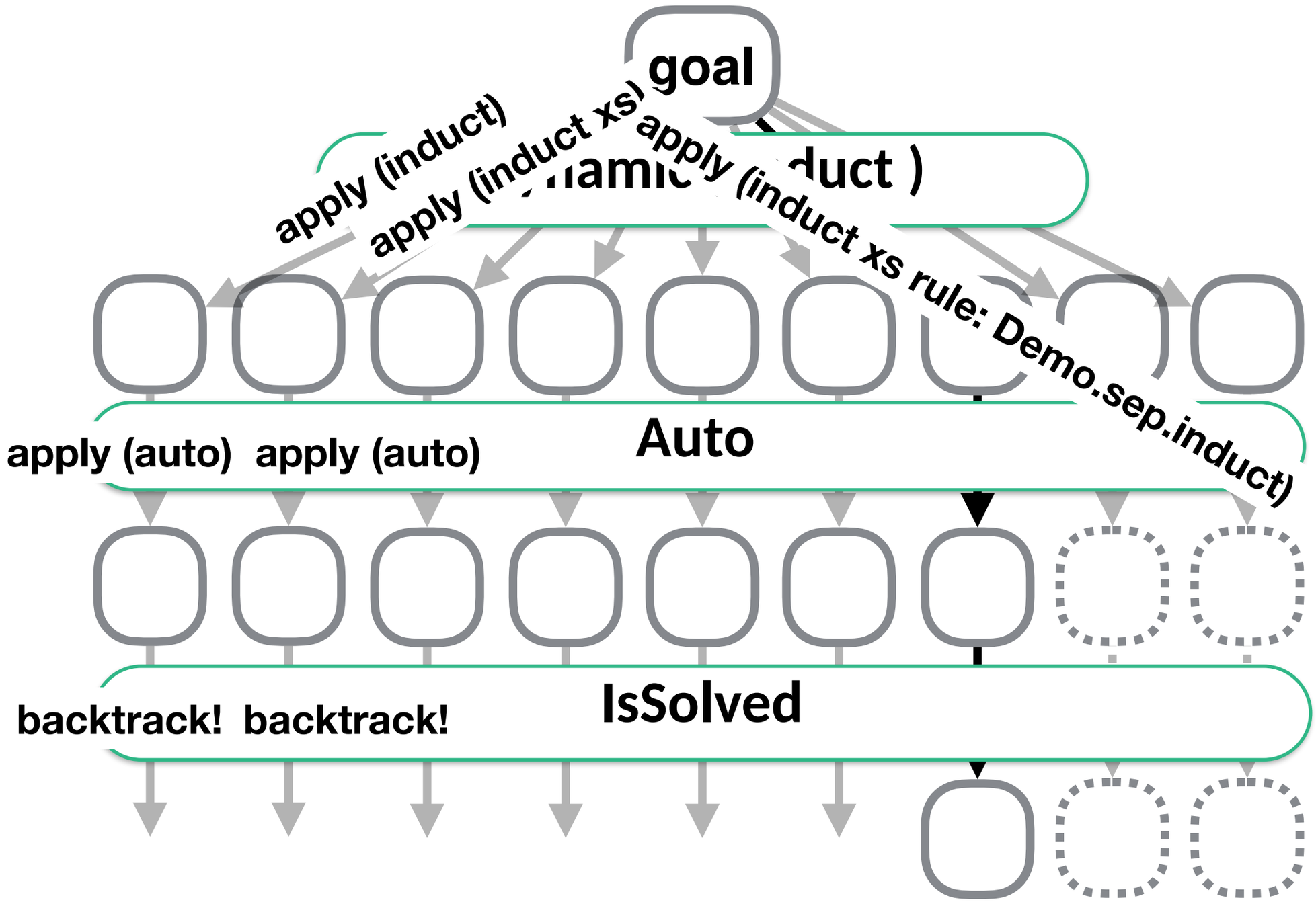
PSL: Proof Strategy Language

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



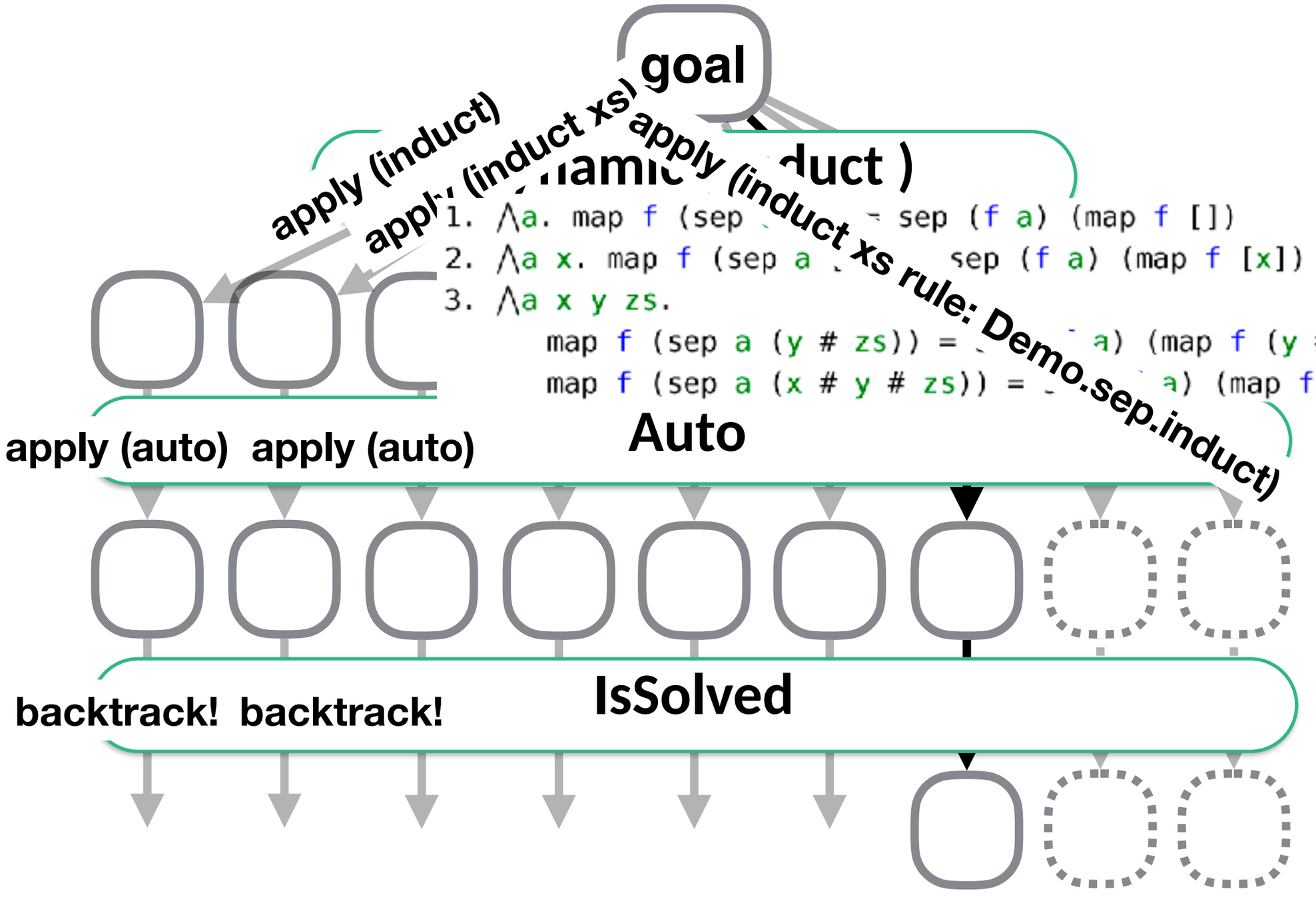
PSL: Proof Strategy Language

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



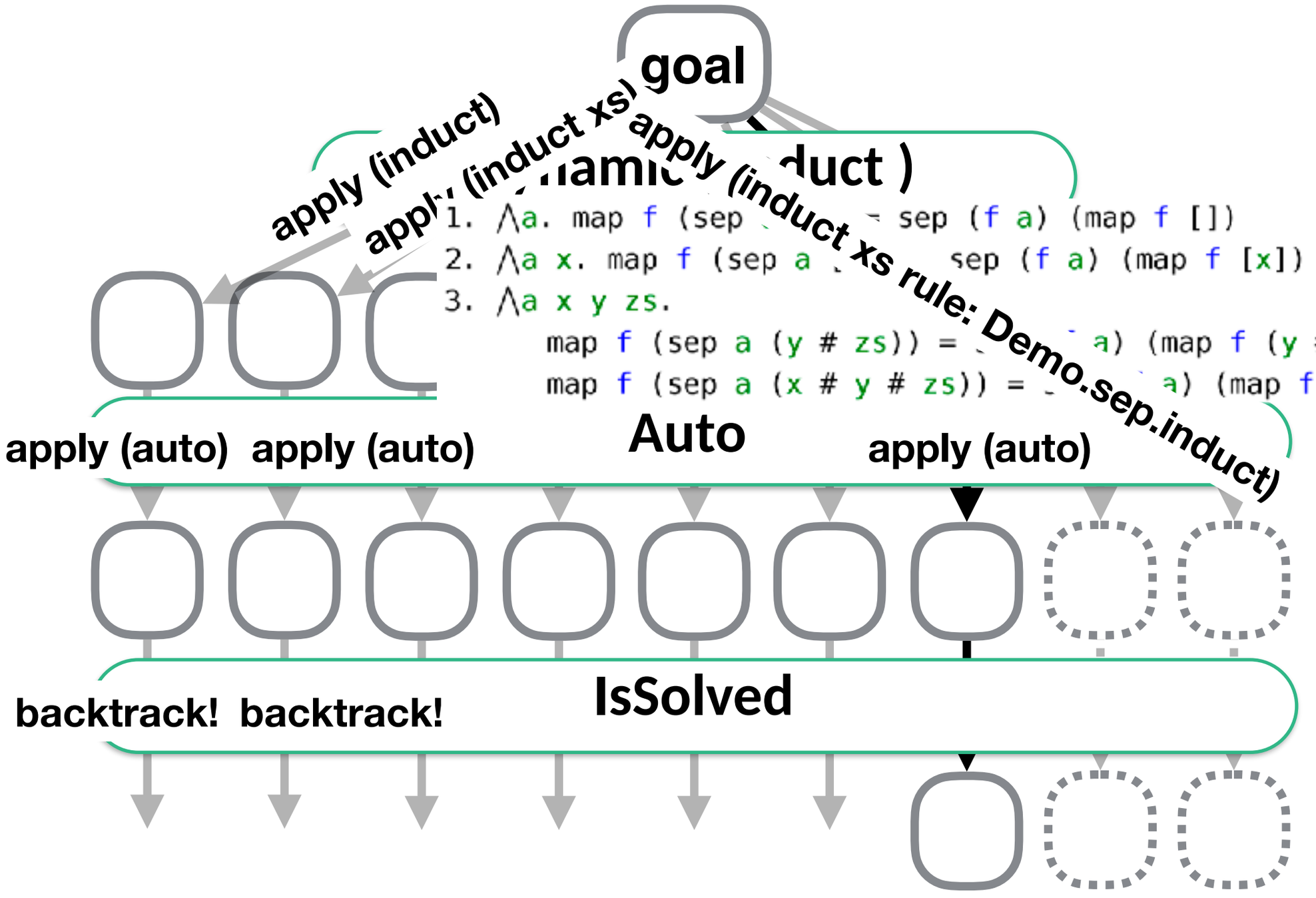
PSL: Proof Strategy Language

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



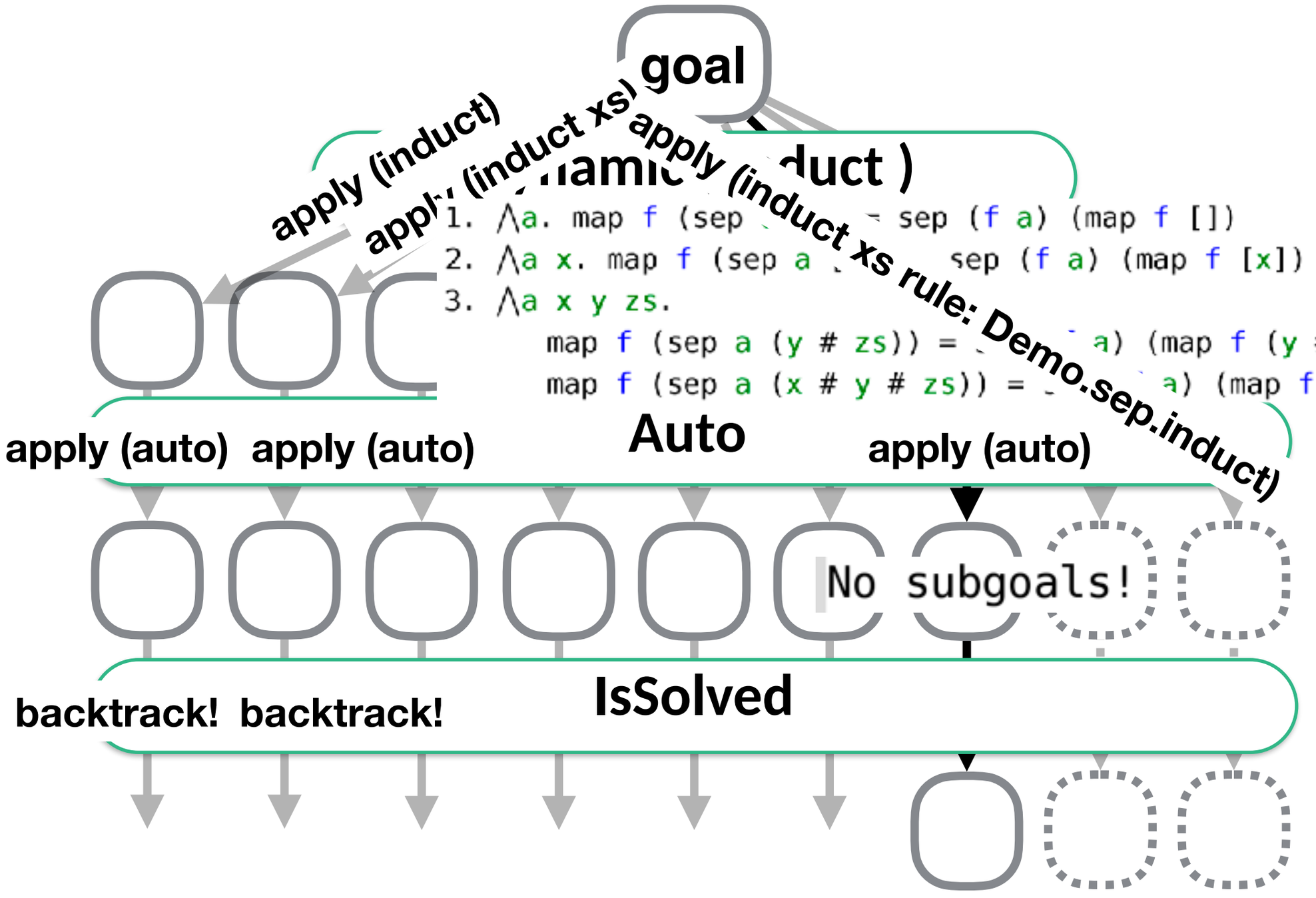
PSL: Proof Strategy Language

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



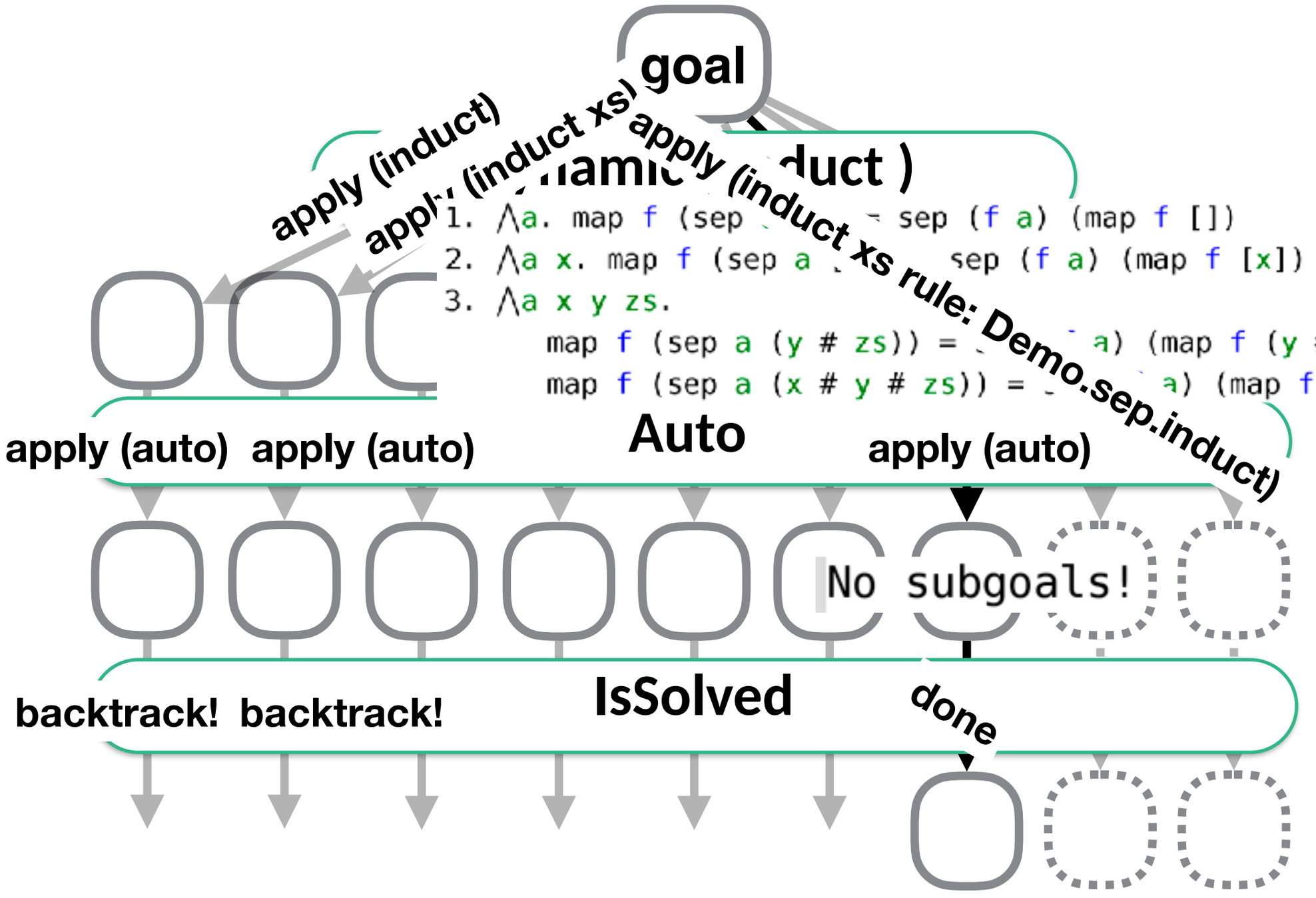
PSL: Proof Strategy Language

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



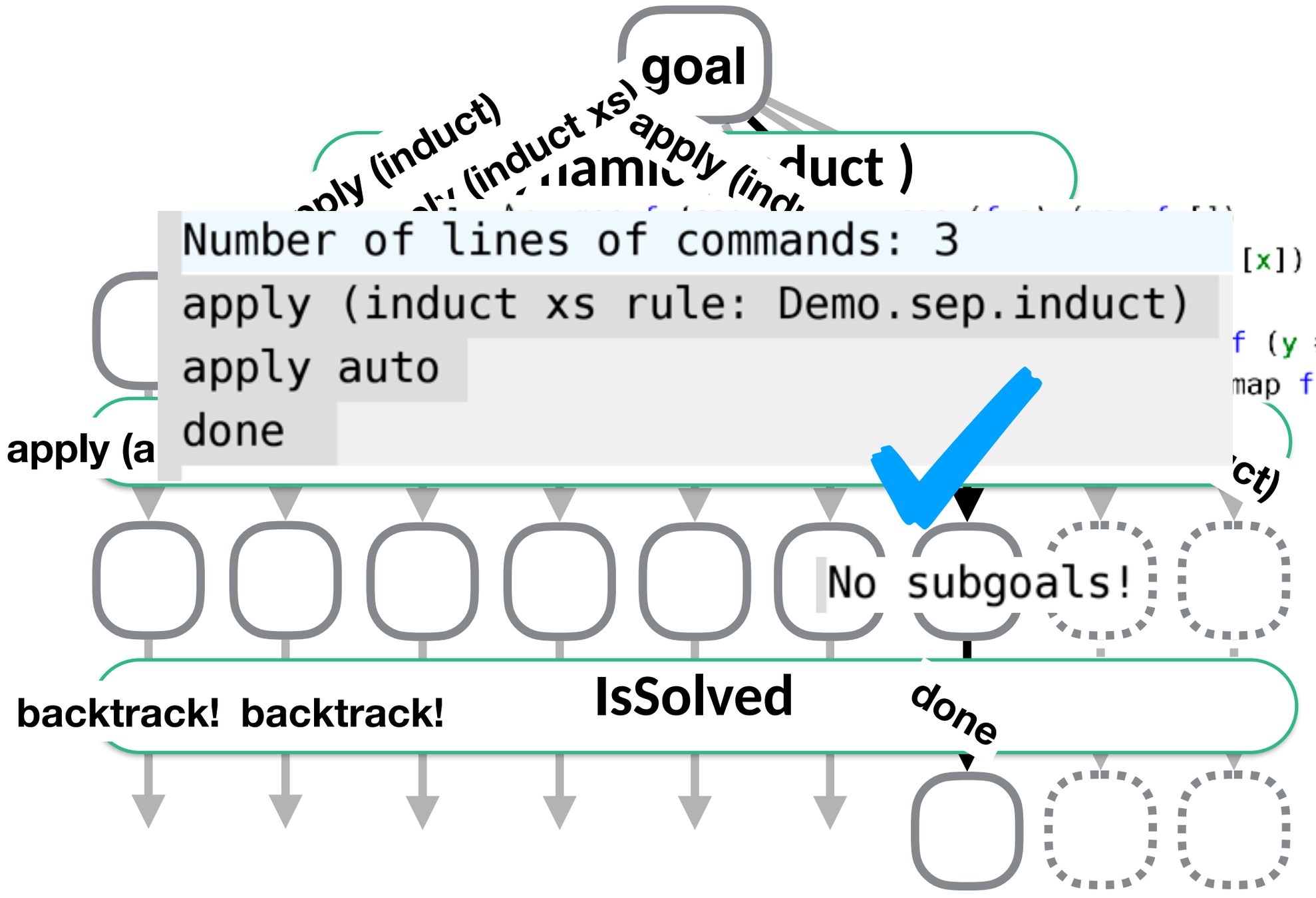
PSL: Proof Strategy Language

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

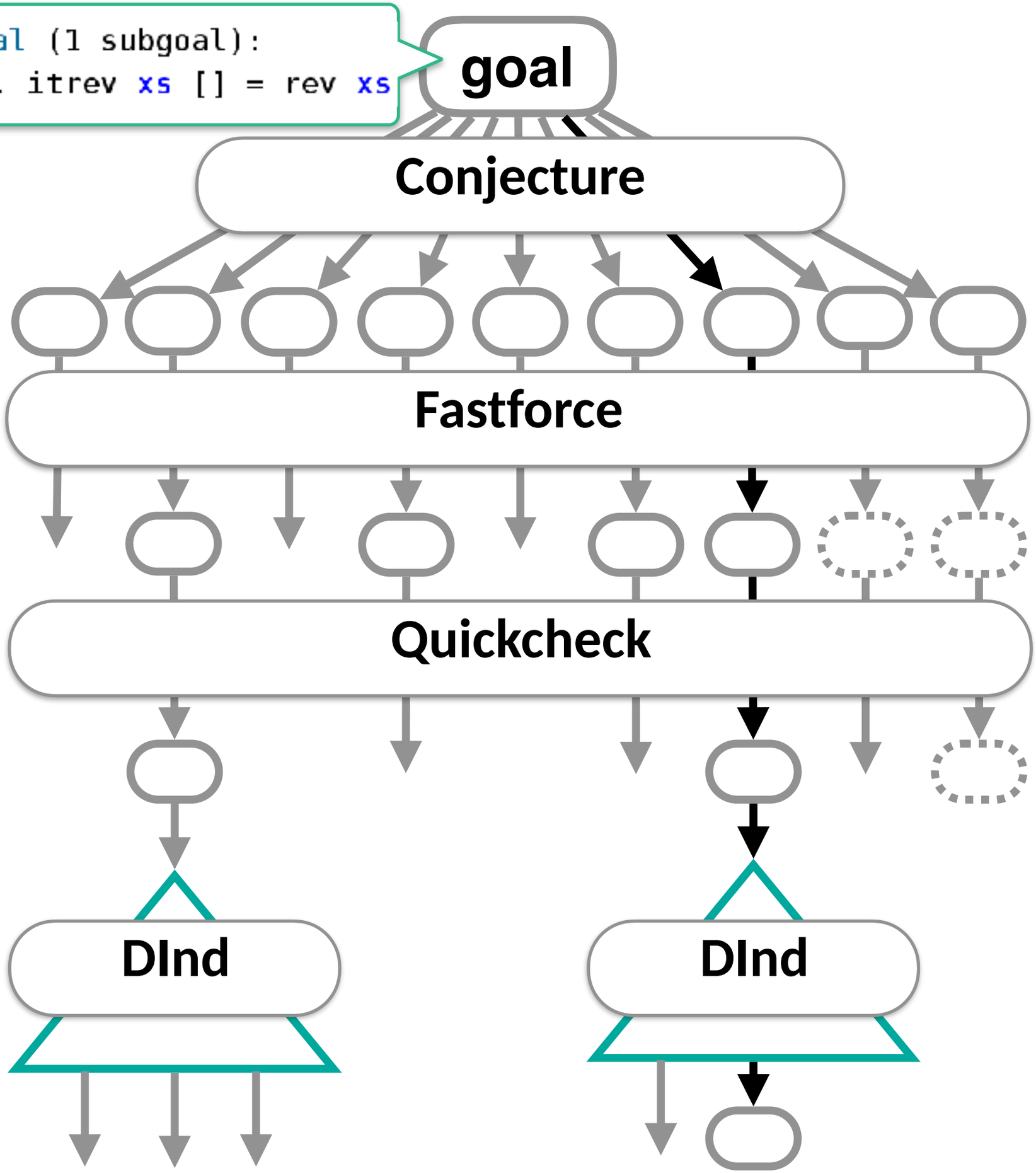


PSL: Proof Strategy Language

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



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



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

goal

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

Conjecture

Fastforce

Quickcheck

DInd

DInd

```
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

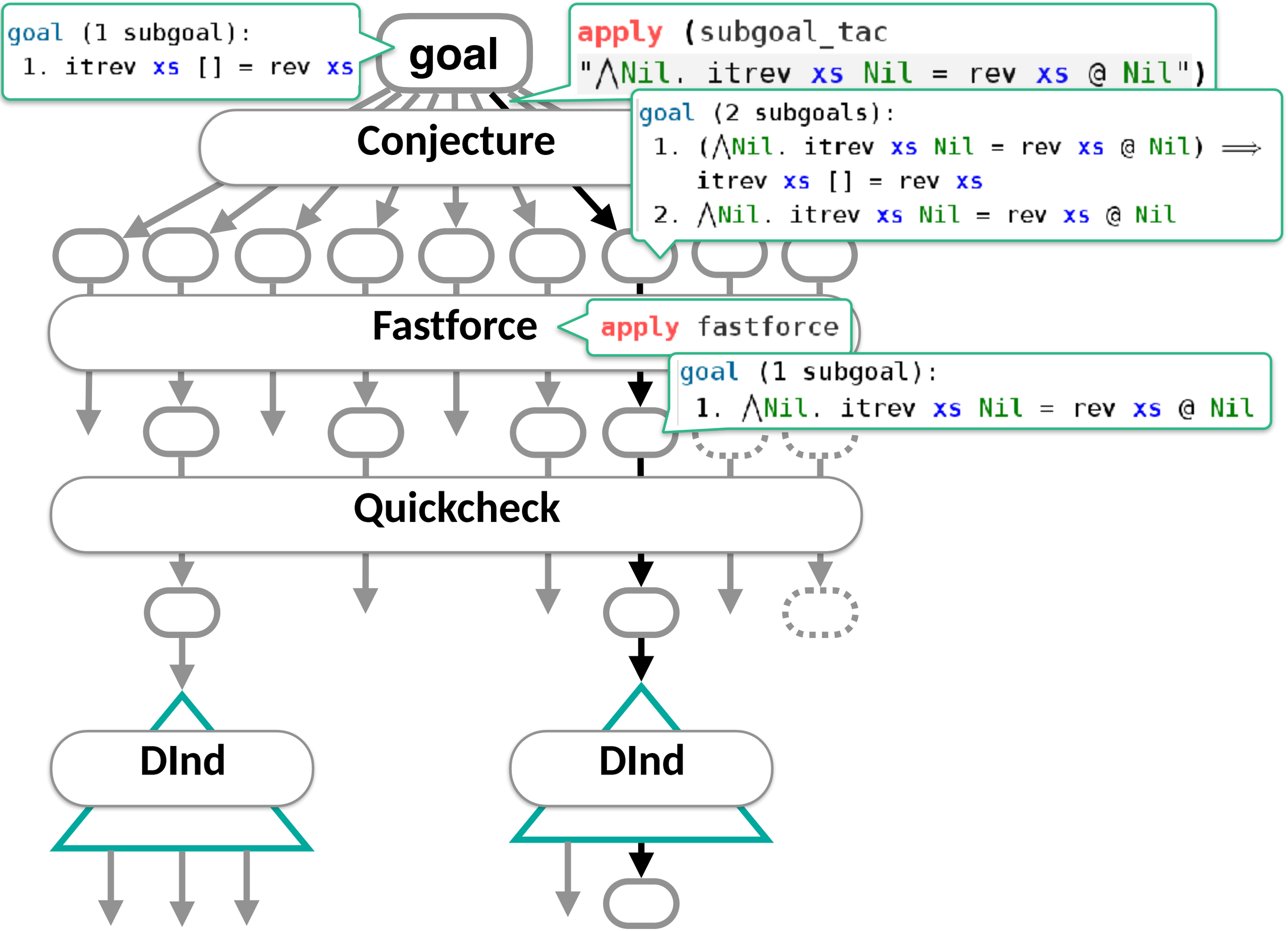
Fastforce

```
apply fastforce
```

Quickcheck

DInd

DInd


goal (1 subgoal):
1. itrev xs [] = rev xs
goal
apply (subgoal_tac
"⋀Nil. itrev xs Nil = rev xs @ Nil")
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
DInd
DInd

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

```
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 (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
```

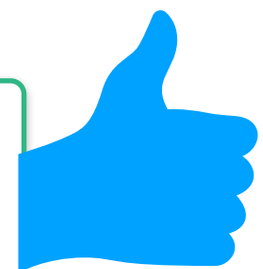
Quickcheck

```
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 (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

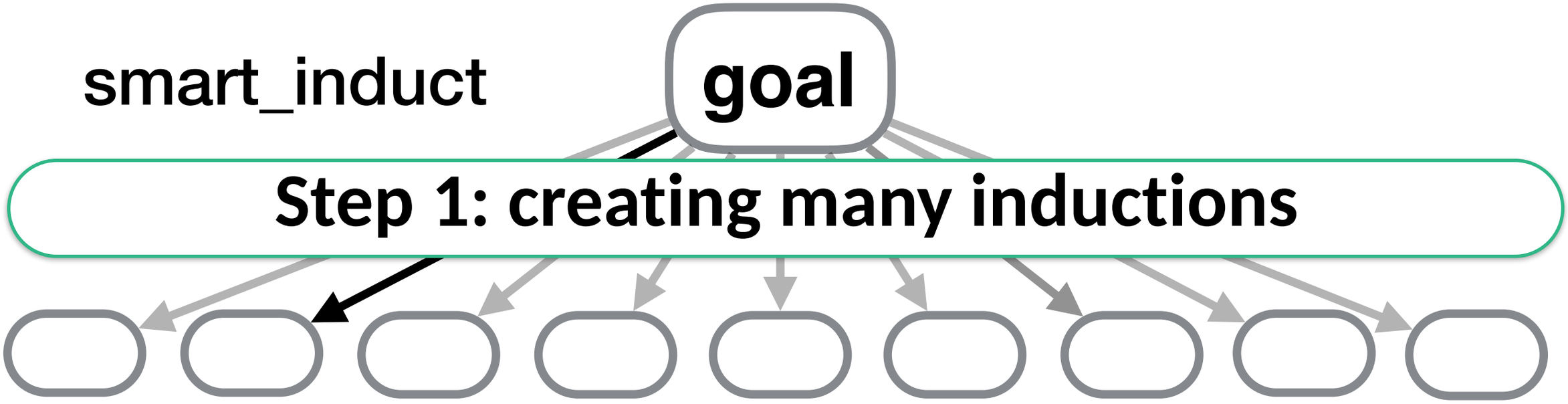
```
goal (1 subgoal):
 1. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

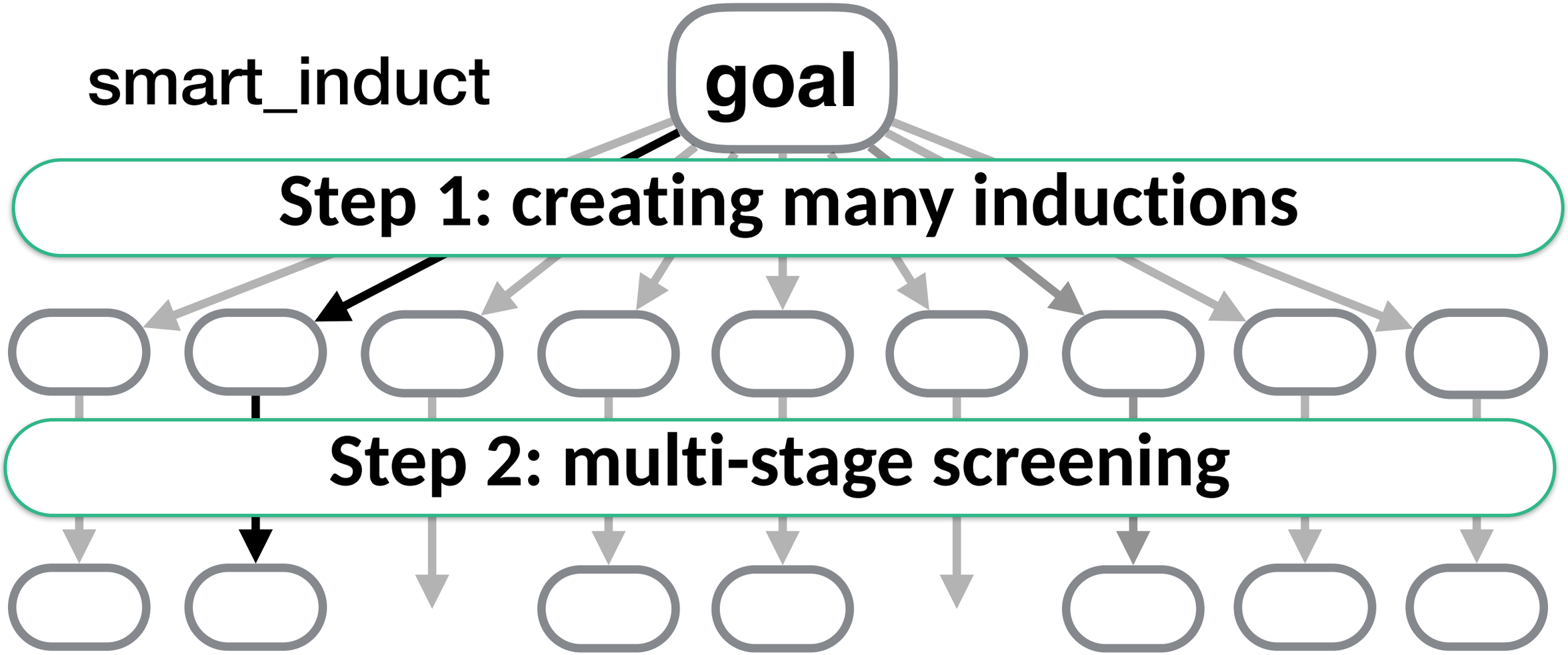
Difficult proof goals?
Search space?

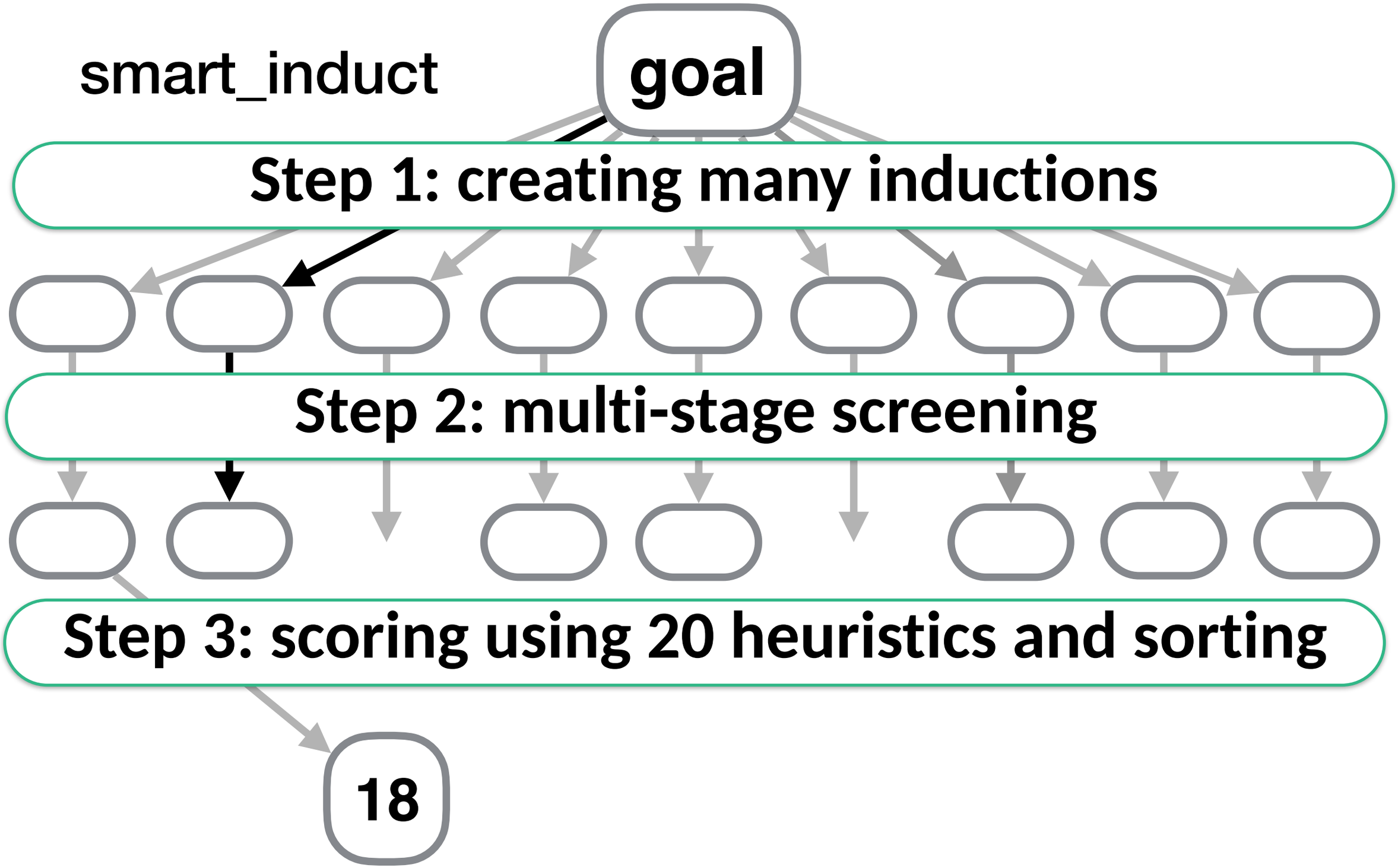
```
theorem itrev ?xs [] = rev ?xs
```

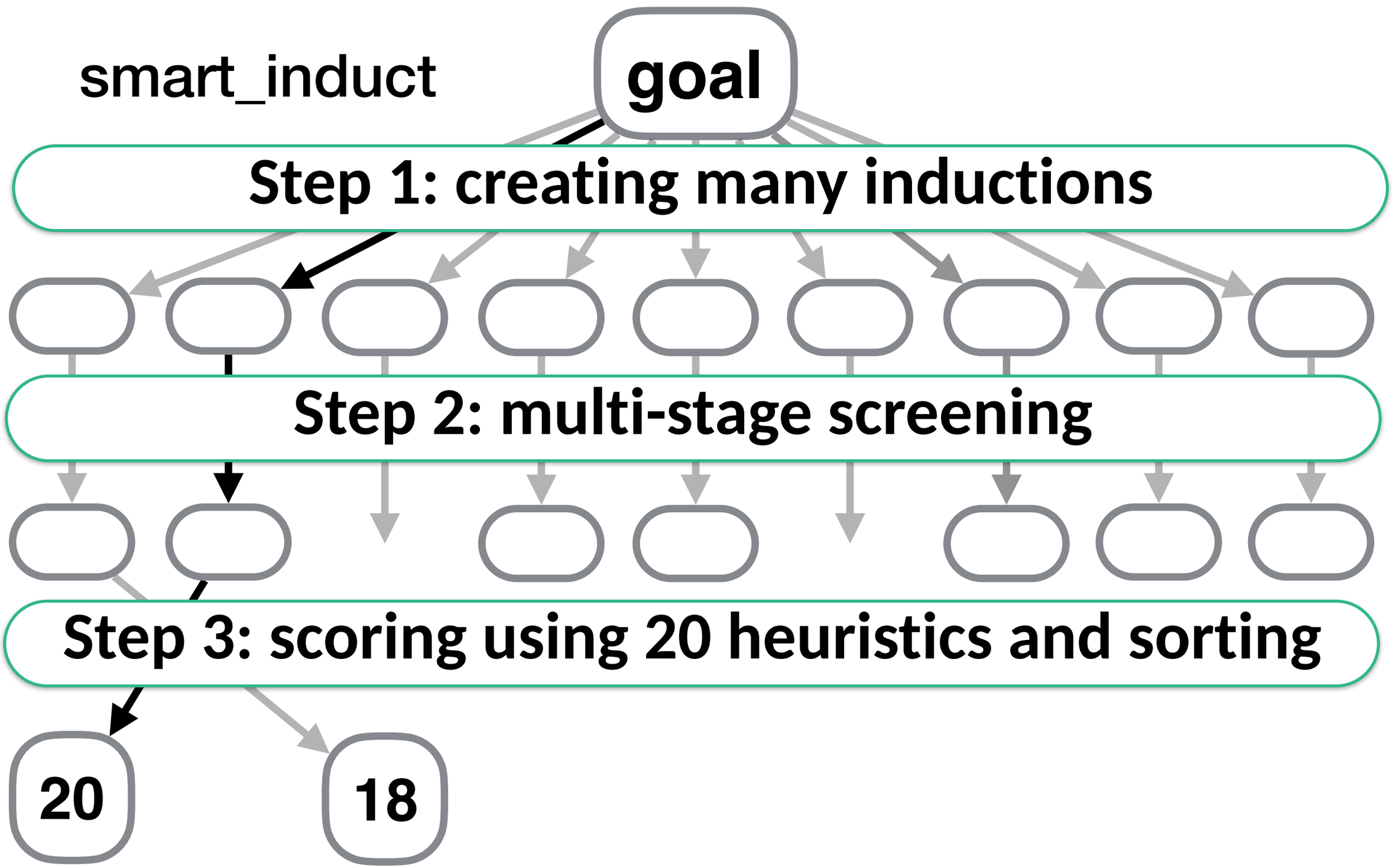
smart_induct

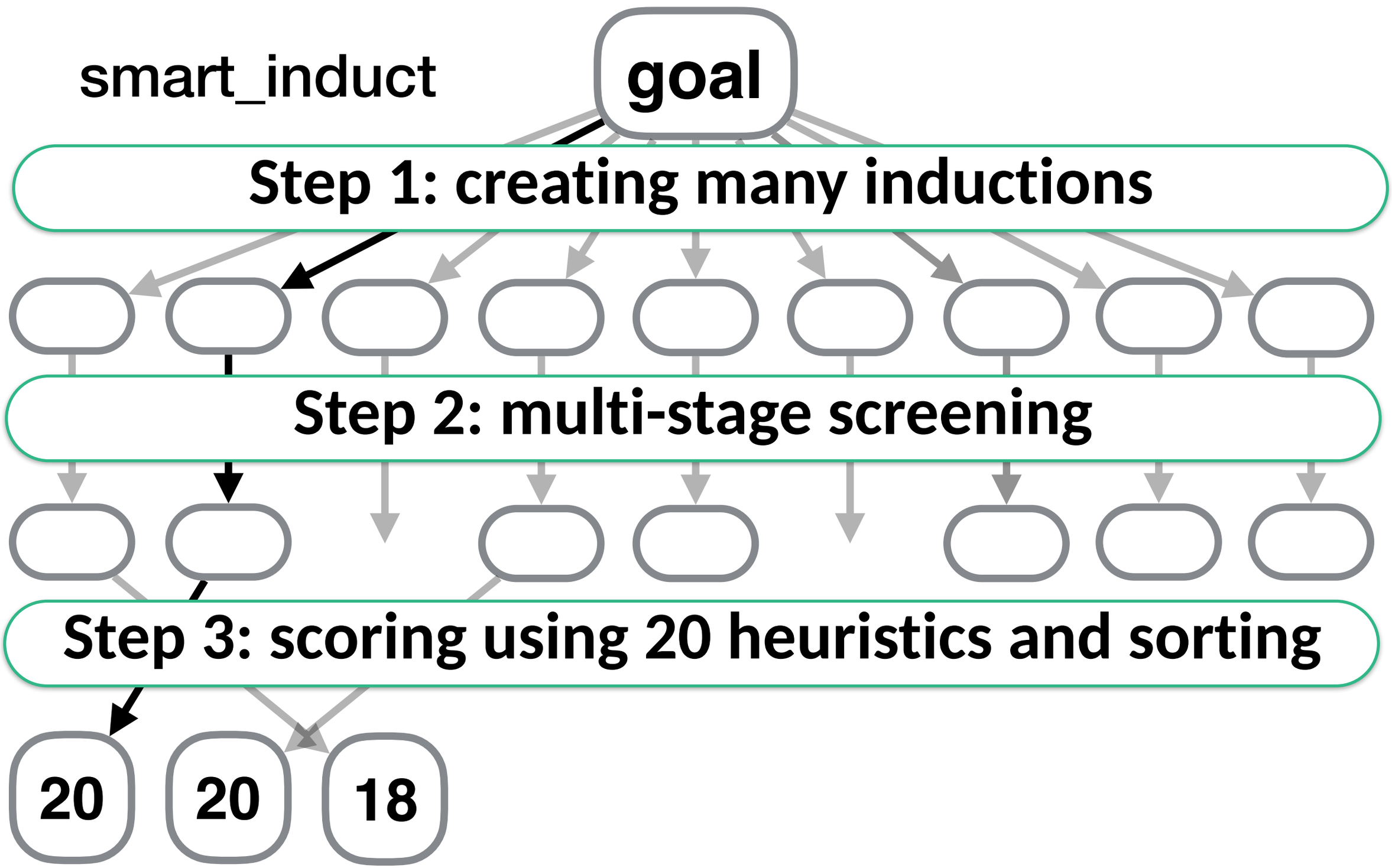
goal

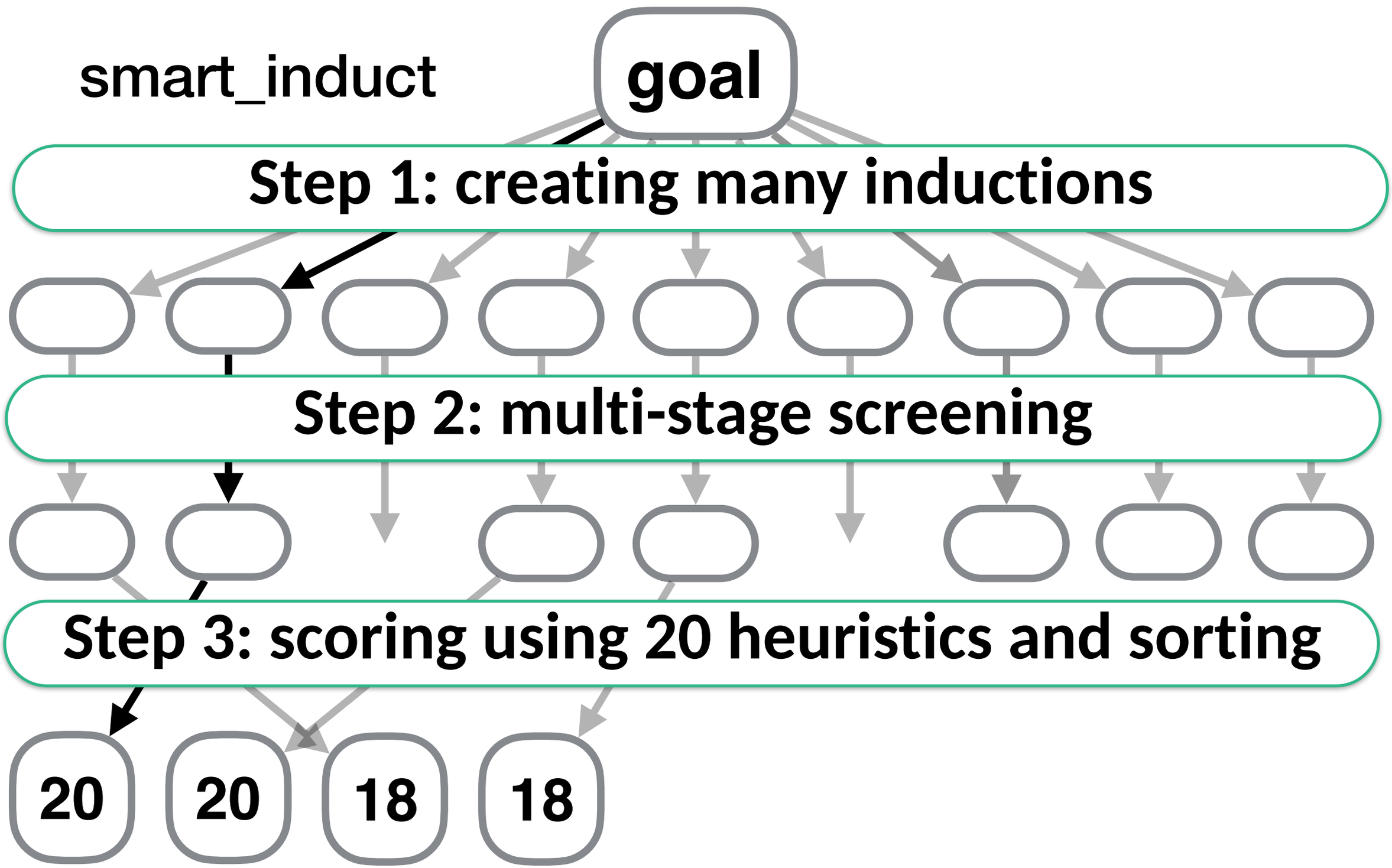

smart_induct
goal
Step 1: creating many inductions

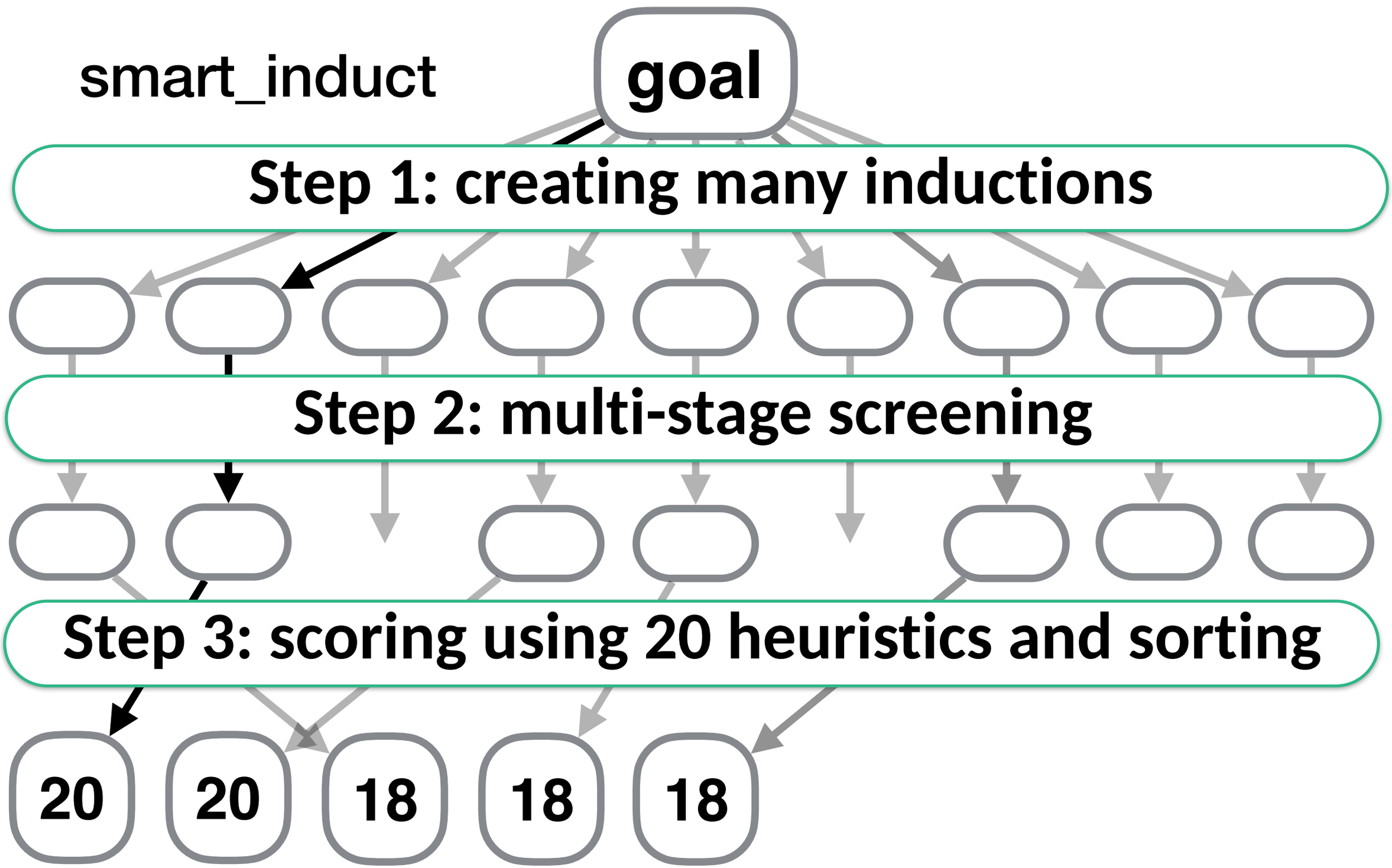

smart_induct
goal
Step 1: creating many inductions
Step 2: multi-stage screening

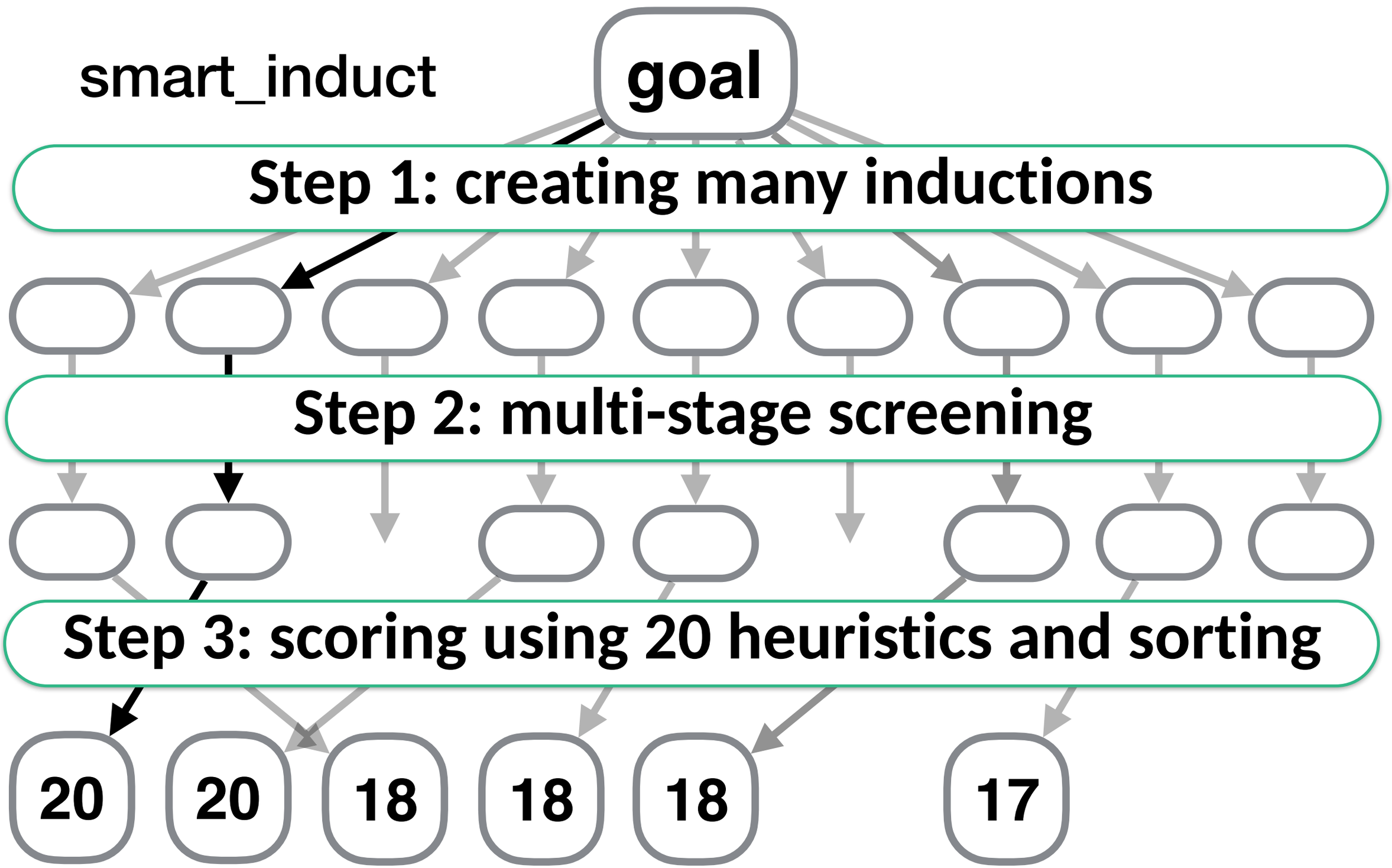

smart_induct
goal
Step 1: creating many inductions
Step 2: multi-stage screening
Step 3: scoring using 20 heuristics and sorting
18


smart_induct
goal
Step 1: creating many inductions
Step 2: multi-stage screening
Step 3: scoring using 20 heuristics and sorting
20
18


smart_induct
goal
Step 1: creating many inductions
Step 2: multi-stage screening
Step 3: scoring using 20 heuristics and sorting
20
20
18


smart_induct
goal
Step 1: creating many inductions
Step 2: multi-stage screening
Step 3: scoring using 20 heuristics and sorting
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
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
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

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

20 20 18 18 18 18 17

Step 4: short-listing

smart_induct

goal

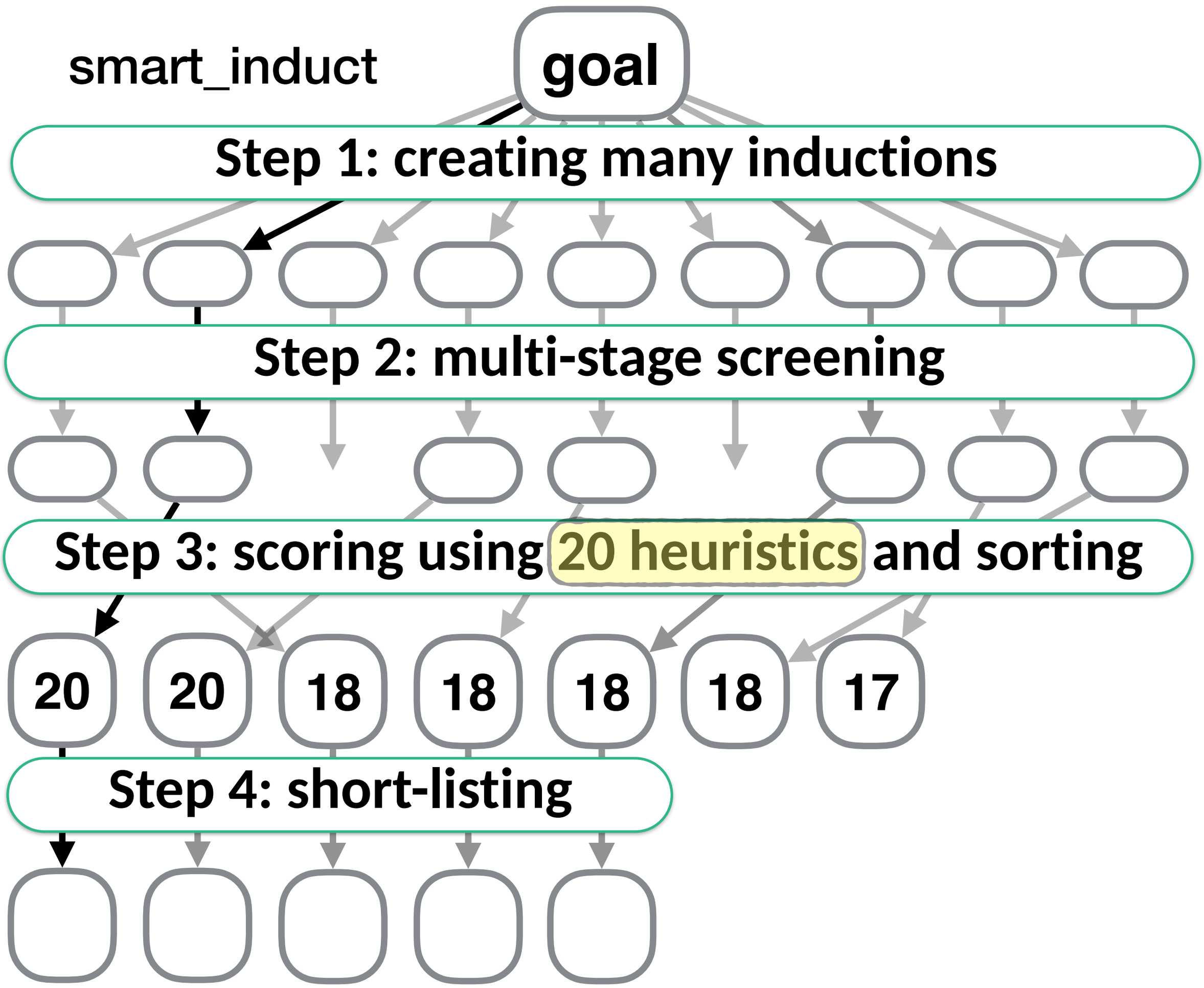
Step 1: creating many inductions

Step 2: multi-stage screening

Step 3: scoring using 20 heuristics and sorting

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
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

?

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

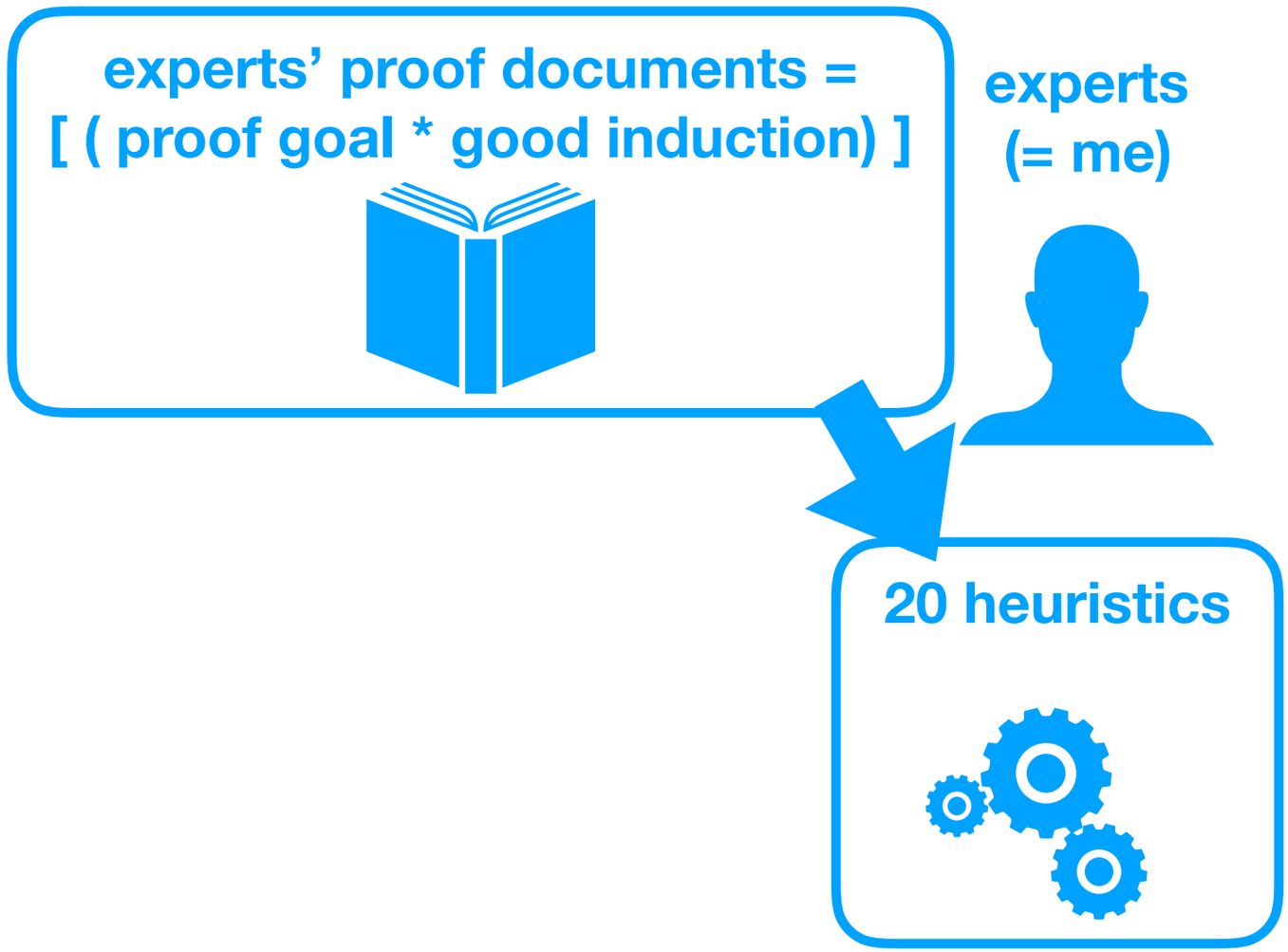
heuristic : (proof goal * induction arguments) -> bool

20 20 18 18 18 18 17

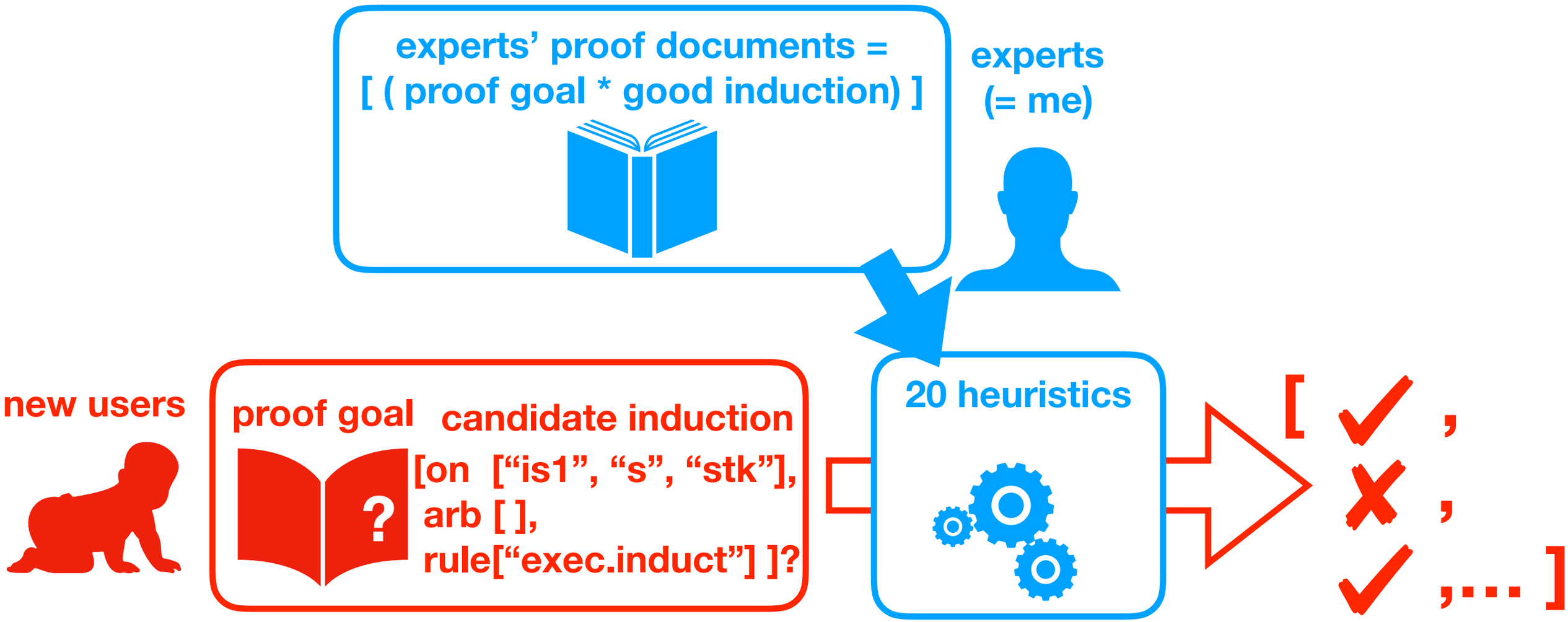
Step 4: short-listing

It is difficult to write induction heuristics.

It is difficult to write induction heuristics.



It is difficult to write induction heuristics.



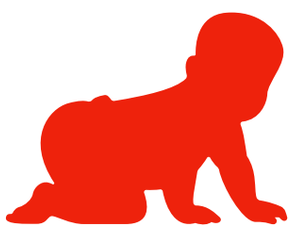
It is difficult to write induction heuristics.

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

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

experts
(= me)

new users

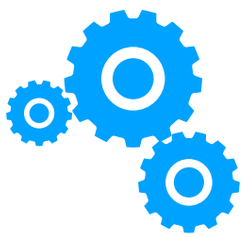


proof goal

candidate induction

[on [“is1”, “s”, “stk”],
arb [ ],
rule[“exec.induct”]]?

20 heuristics



[✓ ,
✗ ,
✓ ,...]

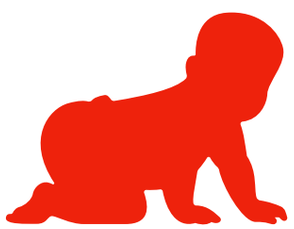
It is difficult to write induction heuristics.

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

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

experts
(= me)

new users

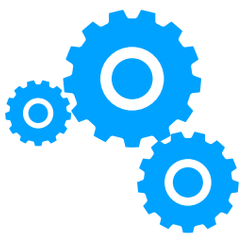


proof goal

candidate induction

[on ["is1", "s", "stk"],
arb [ ],
rule["exec.induct"]]?

20 heuristics



[✓ ,
✗ ,
✓ ,...]

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

It is difficult to write induction heuristics.

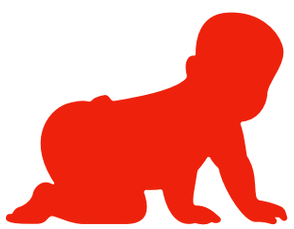
① blue = what I can see before releasing smart_induct

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

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

experts
(= me)

new users

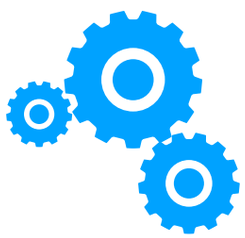


proof goal

candidate induction

[on [“is1”, “s”, “stk”],
arb [ ],
rule[“exec.induct”]]?

20 heuristics



[✓ ,
✗ ,
✓ ,...]

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

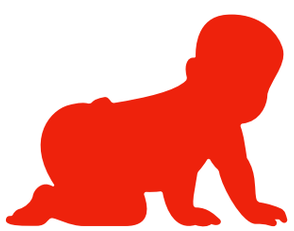
It is difficult to write induction heuristics.

① blue = what I can see before releasing smart_induct

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

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

experts
(= me)



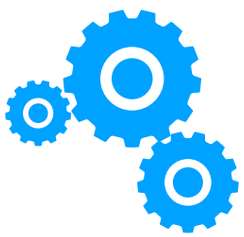
new users

proof goal

candidate induction

[on [“is1”, “s”, “stk”],
arb [ ],
rule[“exec.induct”]]?

20 heuristics



[✓ ,
✗ ,
✓ ,...]

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

② red = what will be developed after releasing smart_induct

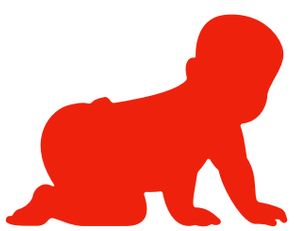
It is difficult to write induction heuristics.

① blue = what I can see before releasing smart_induct

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

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

experts
(= me)



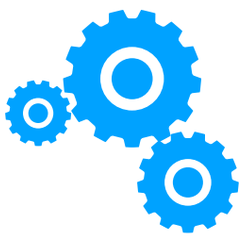
new users

proof goal

candidate induction

[on [“is1”, “s”, “stk”],
arb [ ],
rule[“exec.induct”]]?

20 heuristics



[✓ ,
✗ ,
✓ ,...]

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

② red = what will be developed after releasing smart_induct

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

It is difficult to write induction heuristics.

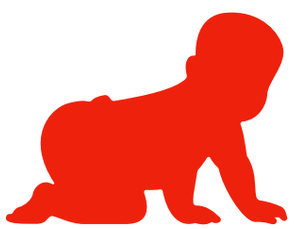
① blue = what I can see before releasing smart_induct

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

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

experts
(= me)

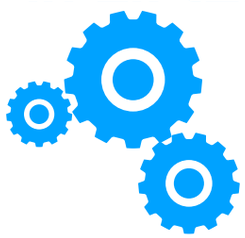
new users



proof goal candidate induction

[on ["is1", "s", "stk"],
arb [ ],
rule["exec.induct"]]?

20 heuristics
in LiFtEr



[✓ ,
✗ ,
✓ ,...]

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

② red = what will be developed after releasing smart_induct

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

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*

Example Assertion in LiFtEr (in Abstract Syntax)

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

variable for auxiliary lemmas

conjunction

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

```
∃ 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
```

variable for auxiliary lemmas

variable for terms

conjunction

Example Assertion in LiFtEr (in Abstract Syntax)

implication

```
∃ 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
```

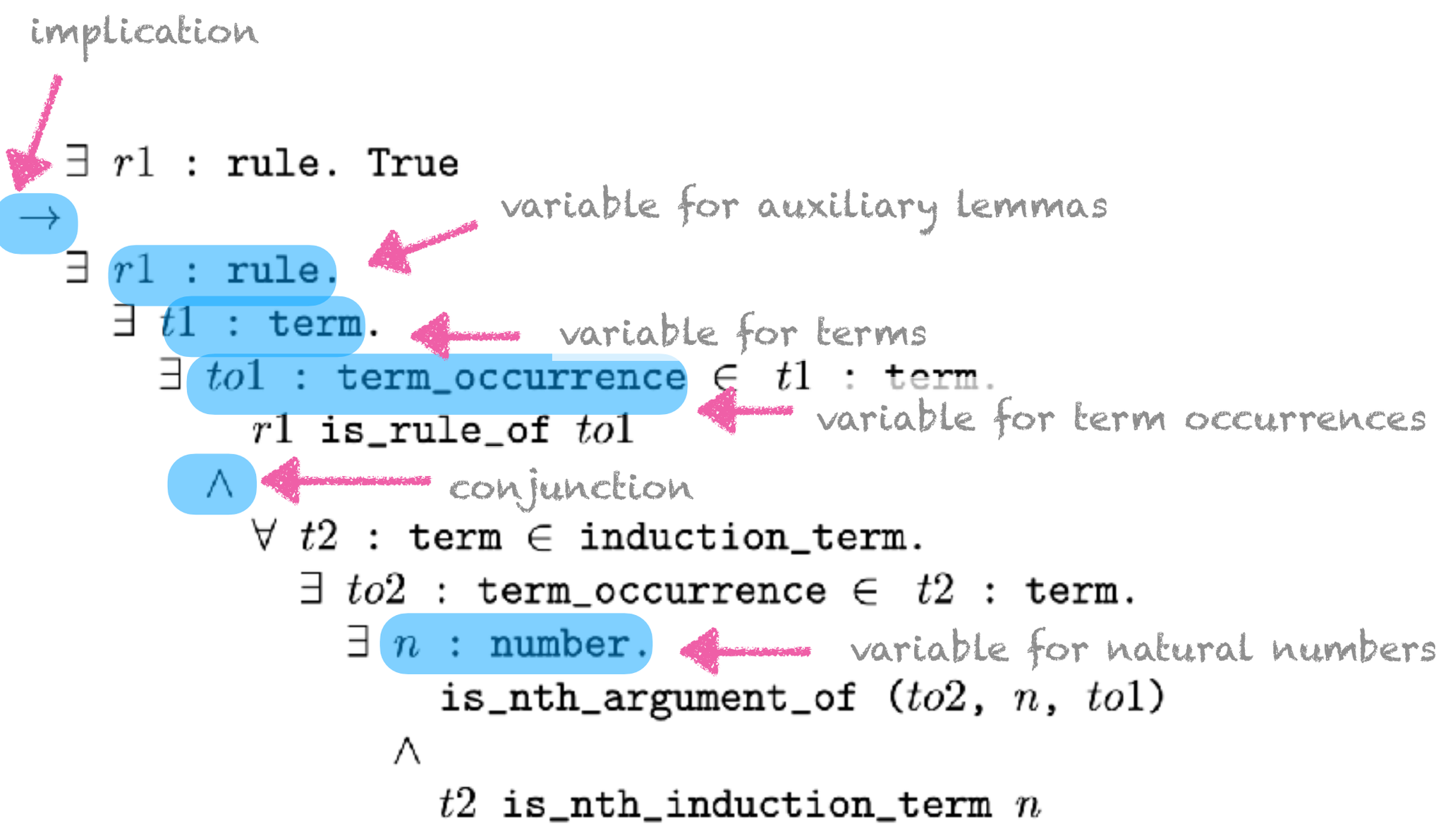
variable for auxiliary lemmas

variable for terms

variable for term occurrences

conjunction

Example Assertion in LiFtEr (in Abstract Syntax)



Example Assertion in LiFtEr (in Abstract Syntax)

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

implication

variable for auxiliary lemmas

variable for terms

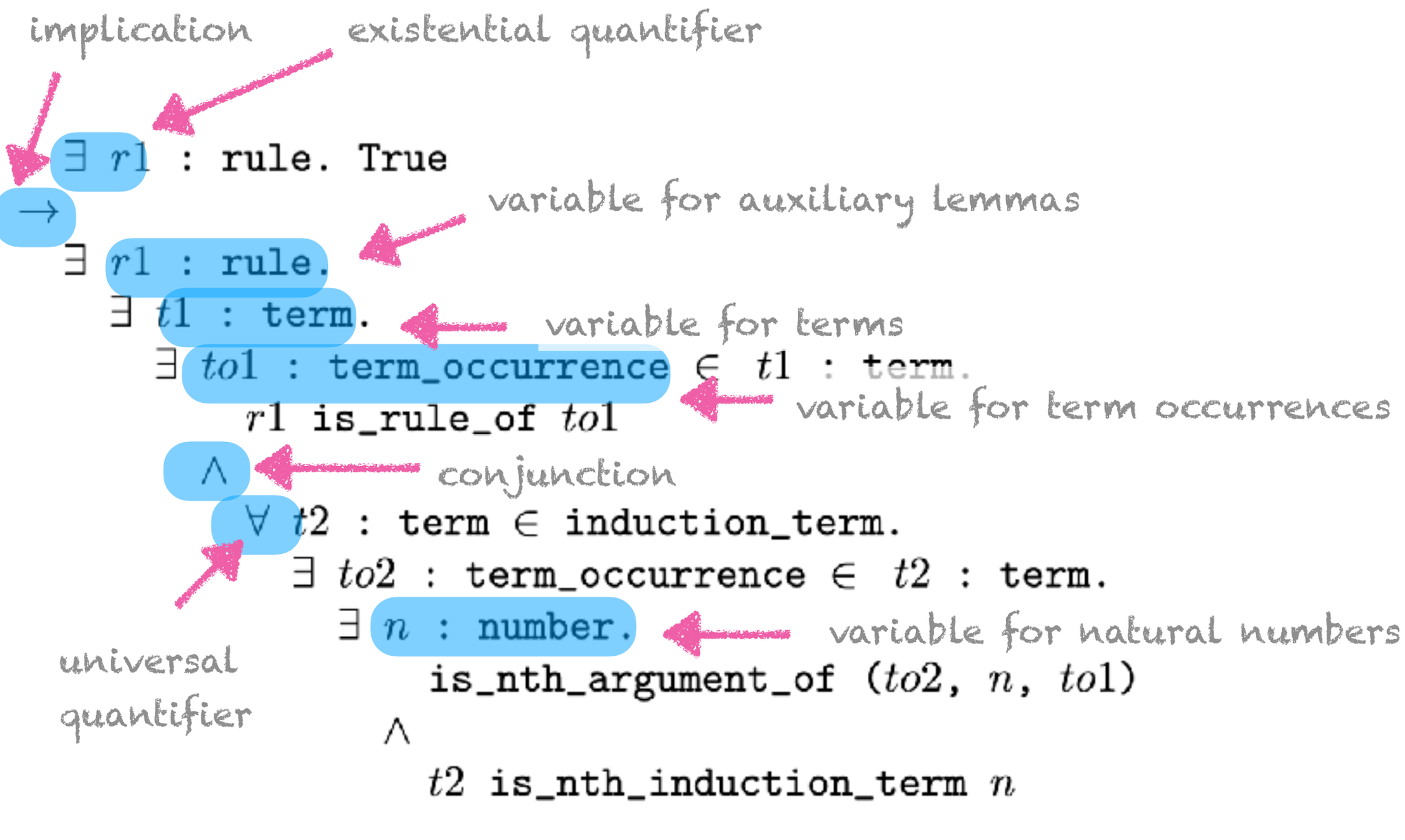
variable for term occurrences

conjunction

universal quantifier

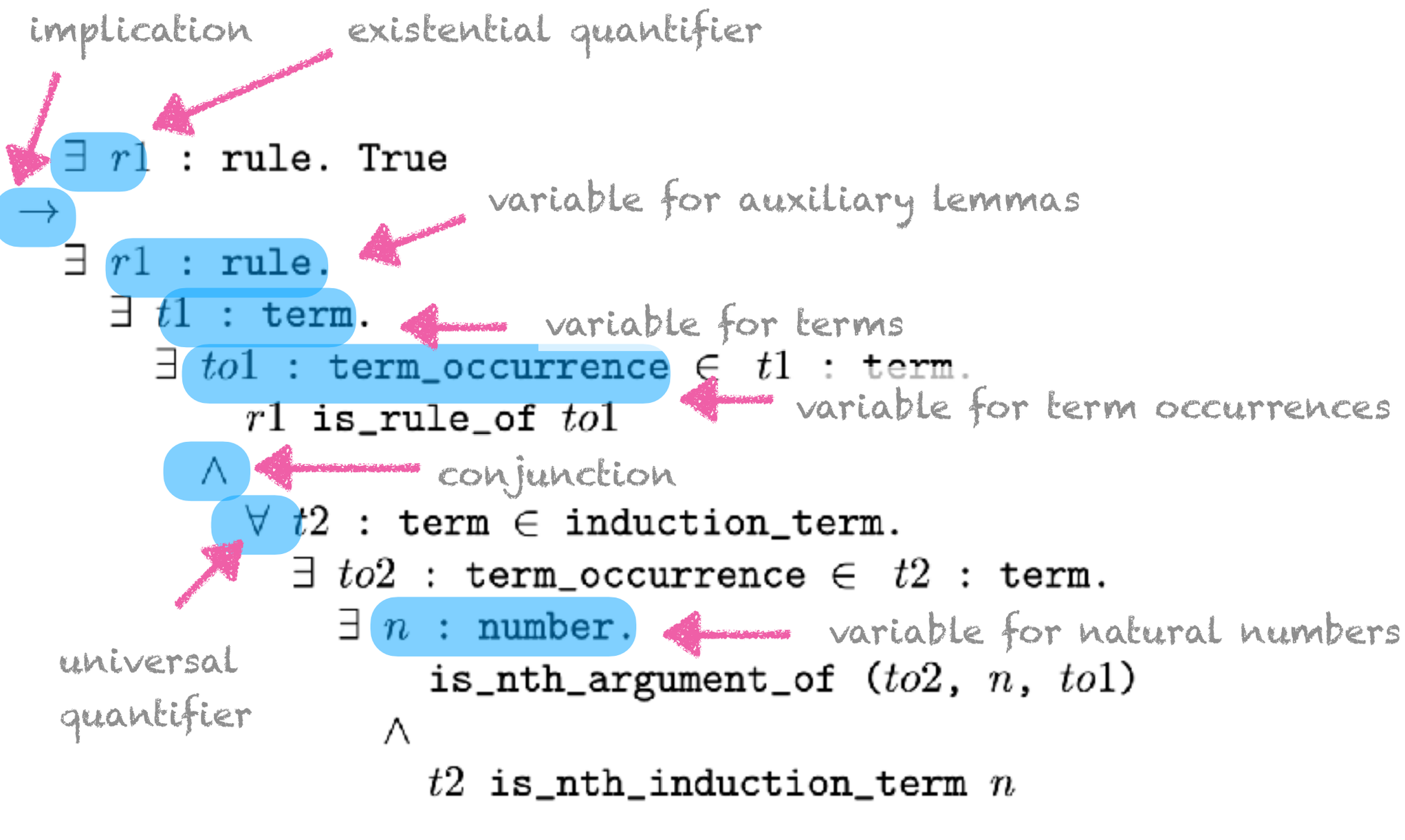
variable for natural numbers

Example Assertion in LiFtEr (in Abstract Syntax)



Example Assertion in LiFtEr (in Abstract Syntax)

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



https://twitter.com/YutakangE

```
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
```

```
∃ 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
```

https://twitter.com/YutakangE

```
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
```

r1

(r1 = itrev.induct)

```
∃ 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
```

https://twitter.com/YutakangE

```
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
```

```
∃ r1 : rule. True
```

→ r1

(r1 = itrev.induct)

```
∃ 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
```

https://twitter.com/YutakangE

```
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
```

```
∃ r1 : rule. True
```

→ r1

```
∃ 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
```

(r1 = itrev.induct)
(t1 = itrev)

https://twitter.com/YutakangE

```
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
```

r1

```
∃ 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
```

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

https://twitter.com/YutakangE

```
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
```

r1

```
∃ r1 : rule. True
→
  ∃ r1 : rule.                                    ( r1 = itrev.induct )
    ∃ t1 : term.                                  ( t1  = itrev )
      ∃ to1 : term_occurrence ∈ t1 : term.        ( to1 = itrev )
          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
```

https://twitter.com/YutakangE

```
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
```

r1

```
∃ r1 : rule. True
→
  ∃ r1 : rule.                                    ( r1 = itrev.induct )
    ∃ t1 : term.                                  ( t1  = itrev )
      ∃ to1 : term_occurrence ∈ t1 : term.        ( to1 = itrev )
          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
```

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

https://twitter.com/YutakangE

```
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
```

r1

```
∃ r1 : rule. True
→
  ∃ r1 : rule.                                    ( r1 = itrev.induct )
    ∃ t1 : term.                                  ( t1  = itrev )
      ∃ to1 : term_occurrence ∈ t1 : term.        ( to1 = itrev )
          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
```

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

https://twitter.com/YutakangE

```
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
```

r1

```
∃ 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
```

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

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

https://twitter.com/YutakangE

```
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
```

r1

$\exists$ *r1* : rule. True

$\rightarrow$

$\exists$ *r1* : rule. (r1 = itrev.induct)
 $\exists$ *t1* : term. (t1 = itrev)
  $\exists$ *to1* : term_occurrence $\in$ *t1* : term. (to1 = itrev)
   r1 is_rule_of *to1* True! r1 (= itrev.induct) is a lemma about to1 (= itrev).
  $\wedge$
   $\forall$ *t2* : term $\in$ induction_term.
    $\exists$ *to2* : term_occurrence $\in$ *t2* : term.
     $\exists$ *n* : number.
      is_nth_argument_of (*to2*, *n*, *to1*)
     $\wedge$
      t2 is_nth_induction_term *n*

https://twitter.com/YutakangE

```
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
```

to1

t2

r1

∃ r1 : rule. True

→

∃ r1 : rule. (r1 = itrev.induct)
  ∃ t1 : term. (t1 = itrev)
    ∃ to1 : term_occurrence ∈ t1 : term. (to1 = itrev)
      r1 is_rule_of to1 True! r1 (= itrev.induct) is a lemma about to1 (= itrev).
    ∧
      ∀ t2 : term ∈ induction_term. (t2 = xs and ys)
        ∃ to2 : term_occurrence ∈ t2 : term.
          ∃ n : number.
            is_nth_argument_of (to2, n, to1)
          ∧
            t2 is_nth_induction_term n

https://twitter.com/YutakangE

```
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 to2

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

t2 r1

∃ r1 : rule. True

→

∃ r1 : rule. (r1 = itrev.induct)
∃ t1 : term. (t1 = itrev)
∃ to1 : term_occurrence ∈ t1 : term. (to1 = itrev)
r1 is_rule_of to1 True! r1 (= itrev.induct) is a lemma about to1 (= itrev).
∧
∀ t2 : term ∈ induction_term. (t2 = xs and ys)
∃ to2 : term_occurrence ∈ t2 : term. (to2 = xs and ys)
∃ n : number.
is_nth_argument_of (to2, n, to1)
∧
t2 is_nth_induction_term n

https://twitter.com/YutakangE

```
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
```

to1 to2 t2 r1

∃ r1 : rule. True

→

∃ r1 : rule. (r1 = itrev.induct)
  ∃ t1 : term. (t1 = itrev)
    ∃ to1 : term_occurrence ∈ t1 : term. (to1 = itrev)
      r1 is_rule_of to1 True! r1 (= itrev.induct) is a lemma about to1 (= itrev).
    ∧
      ∀ t2 : term ∈ induction_term. (t2 = xs and ys)
        ∃ to2 : term_occurrence ∈ t2 : term. (to2 = xs and ys)
          ∃ n : number.
            is_nth_argument_of (to2, n, to1)
          ∧
            t2 is_nth_induction_term n

https://twitter.com/YutakangE

```
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
```

to1 to2 t2 r1

∃ r1 : rule. True

→

∃ r1 : rule. (r1 = itrev.induct)
  ∃ t1 : term. (t1 = itrev)
    ∃ to1 : term_occurrence ∈ t1 : term. (to1 = itrev)
      r1 is_rule_of to1 True! r1 (= itrev.induct) is a lemma about to1 (= itrev).
    ∧
      ∀ t2 : term ∈ induction_term. (t2 = xs and ys)
        ∃ to2 : term_occurrence ∈ t2 : term. (to2 = xs and ys)
          ∃ n : number.
            is_nth_argument_of (to2, n, to1)
          ∧
            t2 is_nth_induction_term n

https://twitter.com/YutakangE

```
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
```

to1 first to2 t2 first r1

∃ r1 : rule. True

→

∃ r1 : rule. (r1 = itrev.induct)
∃ t1 : term. (t1 = itrev)
∃ to1 : term_occurrence ∈ t1 : term. (to1 = itrev)
r1 is_rule_of to1 True! r1 (= itrev.induct) is a lemma about to1 (= itrev).
∧
∀ t2 : term ∈ induction_term. (t2 = xs and ys)
∃ to2 : term_occurrence ∈ t2 : term. (to2 = xs and ys)
∃ n : number.
is_nth_argument_of (to2, n, to1) when t2 is xs (n = 1) ?
∧
t2 is_nth_induction_term n

https://twitter.com/YutakangE

```
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
```

to1 first to2 t2 first r1

∃ r1 : rule. True

→

∃ r1 : rule. (r1 = itrev.induct)
∃ t1 : term. (t1 = itrev)
∃ to1 : term_occurrence ∈ t1 : term. (to1 = itrev)
r1 is_rule_of to1 True! r1 (= itrev.induct) is a lemma about to1 (= itrev).
∧
∀ t2 : term ∈ induction_term. (t2 = xs and ys)
∃ to2 : term_occurrence ∈ t2 : term. (to2 = xs and ys)
∃ n : number.
is_nth_argument_of (to2, n, to1) when t2 is xs (n = 1)
∧
t2 is_nth_induction_term n

https://twitter.com/YutakangE

```
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
```

to1 first second to2 t2 second first r1

∃ r1 : rule. True

→

∃ r1 : rule. (r1 = itrev.induct)
∃ t1 : term. (t1 = itrev)
∃ to1 : term_occurrence ∈ t1 : term. (to1 = itrev)
r1 is_rule_of to1 True! r1 (= itrev.induct) is a lemma about to1 (= itrev).
∧
∀ t2 : term ∈ induction_term. (t2 = xs and ys)
∃ to2 : term_occurrence ∈ t2 : term. (to2 = xs and ys)
∃ n : number.
is_nth_argument_of (to2, n, to1) when t2 is xs (n = 1) ?
∧
t2 is_nth_induction_term n when t2 is ys (n = 2) ?

https://twitter.com/YutakangE

```
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
```

to1 first second to2 t2 second first r1

∃ 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

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

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

(t2 = xs and ys)
(to2 = xs and ys)

when t2 is xs (n = 1)

when t2 is ys (n = 2)

https://twitter.com/YutakangE

```
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
```

to1 first second to2 t2 second first r1

∃ r1 : rule. True

→

∃ r1 : rule. (r1 = itrev.induct)
∃ t1 : term. (t1 = itrev)
∃ to1 : term_occurrence ∈ t1 : term. (to1 = itrev)
r1 is_rule_of to1 True! r1 (= itrev.induct) is a lemma about to1 (= itrev).
∧
∀ t2 : term ∈ induction_term. (t2 = xs and ys)
∃ to2 : term_occurrence ∈ t2 : term. (to2 = xs and ys)
∃ n : number.
is_nth_argument_of (to2, n, to1) when t2 is xs (n = 1)
∧
t2 is_nth_induction_term n when t2 is ys (n = 2)

$\exists$ $r1$: rule. True

$\rightarrow$

$\exists$ $r1$: rule.
$\exists$ $t1$: term.
$\exists$ $to1$: term_occurrence $\in$ $t1$: term.
$r1$ is_rule_of $to1$
$\wedge$
$\forall$ $t2$: term $\in$ induction_term.
$\exists$ $to2$: term_occurrence $\in$ $t2$: term.
$\exists$ n : number.
is_nth_argument_of ($to2$, n, $to1$)
$\wedge$
$t2$ is_nth_induction_term n

the same LiFtEr assertion

$\exists\ r1$: rule. True

$\rightarrow$

$\exists\ r1$: rule.
$\exists\ t1$: term.
$\exists\ to1$: term_occurrence $\in\ t1$: term.
$r1$ is_rule_of $to1$
$\wedge$
$\forall\ t2$: term $\in$ induction_term.
$\exists\ to2$: term_occurrence $\in\ t2$: term.
$\exists\ n$: number.
is_nth_argument_of ($to2$, n, $to1$)
$\wedge$
$t2$ is_nth_induction_term n

new types ->

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

the same LiFtEr assertion

```
∃ 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
```

new types ->

new constants ->

```
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)"
```

the same LiFtEr assertion

```
∃ 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
```

new types ->

new constants ->

```
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)"
```

the same LiFtEr assertion

new lemma ->

a model proof ->

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

```
∃ 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
```

new types ->

new constants ->

new lemma ->

a model proof ->

```
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 is1 s stk rule:exec.induct)
 apply auto done
```

∃ 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
```

new types ->

new constants ->

```
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)"
```

new lemma ->

a model proof ->

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

∃ r1 : rule. True

→

r1

(r1 = exec.induct)

∃ 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

new types ->

new constants ->

```
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)"
```

new lemma ->

a model proof ->

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

∃ 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
```

r1

(r1 = exec.induct)

new types ->

new constants ->

```
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)"
```

new lemma ->

a model proof ->

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

∃ 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
```

r1

(r1 = exec.induct)

(t1 = exec)

new types ->

new constants ->

new lemma ->

a model proof ->

```
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 is1 s stk rule:exec.induct)
  apply auto done
```

to1

r1

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

∃ 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
```

new types ->

new constants ->

```
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)"
```

new lemma ->
a model proof ->

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

to1

r1

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

∃ 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
```

new types ->

new constants ->

```
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)"
```

new lemma ->

a model proof ->

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

to1

r1

∃ 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
```

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

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

new types ->

new constants ->

```
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)"
```

new lemma ->

a model proof ->

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

to1

r1

∃ 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
```

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

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

new types ->

new constants ->

```
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)"
```

new lemma ->
a model proof ->

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

to1

r1

∃ 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
```

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

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

new types ->

new constants ->

```
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)"
```

new lemma ->

a model proof ->

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

∃ r1 : rule. True

to1

r1

→

∃ r1 : rule.
∃ t1 : term.
∃ to1 : term_occurrence ∈ t1 : term.

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

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

∧

∀ t2 : term ∈ induction_term.
∃ to2 : term_occurrence ∈ t2 : term.
∃ n : number.
is_nth_argument_of (to2, n, to1)

∧

t2 is_nth_induction_term n

new types ->

new constants ->

```
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)"
```

new lemma ->
a model proof ->
∃ r1 : rule. True

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

t2 — is1, s, stk

to1 — exec is1 s stk

r1 — exec.induct

→

∃ 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

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

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

(t2 = is1, s, and stk)

new types ->

new constants ->

```
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)"
```

new lemma ->
a model proof ->

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

t2 t01 to2

∃ 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

r1

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

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

(t2 = is1, s, and stk)
(to2 = is1, s, and stk)

new types ->

new constants ->

```
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)"
```

new lemma ->
a model proof ->

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

t2 t01 t02 r1

∃ r1 : rule. True

→

∃ r1 : rule.
(r1 = exec.induct)
∃ t1 : term.
(t1 = exec)
∃ to1 : term_occurrence ∈ t1 : term.
(to1 = exec)
r1 is_rule_of to1
True! r1 (= exec.induct) is a lemma about to1 (= exec).
∧
∀ t2 : term ∈ induction_term.
(t2 = is1, s, and stk)
∃ to2 : term_occurrence ∈ t2 : term.
(to2 = is1, s, and stk)
∃ n : number.
is_nth_argument_of (to2, n, to1)
∧
t2 is_nth_induction_term n

new types ->

new constants ->

```
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)"
```

new lemma ->
a model proof ->

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

t2 t01 t02 r1

∃ r1 : rule. True

→

∃ r1 : rule.
(r1 = exec.induct)
∃ t1 : term.
(t1 = exec)
∃ to1 : term_occurrence ∈ t1 : term.
(to1 = exec)
r1 is_rule_of to1
True! r1 (= exec.induct) is a lemma about to1 (= exec).
∧
∀ t2 : term ∈ induction_term.
(t2 = is1, s, and stk)
∃ to2 : term_occurrence ∈ t2 : term.
(to2 = is1, s, and stk)
∃ n : number.
is_nth_argument_of (to2, n, to1)
∧
t2 is_nth_induction_term n

new types ->

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

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)
  apply auto done
```

first

to1

to2

t2

r1

first

∃ 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

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

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

(t2 = is1, s, and stk)
(to2 = is1, s, and stk)

when t2 is is1 (n -> 1) ?

new types ->

new constants ->

```
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)"
```

first

to1

t2

to2

new lemma ->

a model proof ->

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

r1

first

$\exists$ r1 : rule. True

$\rightarrow$

$\exists$ $r1$: rule.
$\exists$ $t1$: term.
$\exists$ $to1$: term_occurrence $\in$ $t1$: term.
r1 is_rule_of $to1$
$\wedge$
$\forall$ $t2$: term $\in$ induction_term.
$\exists$ $to2$: term_occurrence $\in$ $t2$: term.
$\exists$ n : number.
is_nth_argument_of ($to2$, n, $to1$)
$\wedge$
$t2$ is_nth_induction_term n

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

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

(t2 = is1, s, and stk)
(to2 = is1, s, and stk)

when t2 is is1 (n -> 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)
  apply auto done
```

first second t2 to1 to2 r1

∃ r1 : rule. True

→

∃ $r1$: rule.
∃ $t1$: term.
∃ $to1$: term_occurrence ∈ $t1$: term.
$r1$ is_rule_of $to1$ True! r1 (= exec.induct) is a lemma about to1 (= exec).
∧
∀ $t2$: term ∈ induction_term. (t2 = is1, s, and stk) ✓
∃ $to2$: term_occurrence ∈ $t2$: term. (to2 = is1, s, and stk)
∃ n : number.
is_nth_argument_of ($to2$, n, $to1$) when t2 is is1 (n -> 1)
∧ when t2 is s (n -> 2) ?
$t2$ is_nth_induction_term n

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

new types ->

new constants ->

```
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)"
```

new lemma ->

a model proof ->

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

first second t2 to1 to2 r1

∃ 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

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

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

(t2 = is1, s, and stk)
(to2 = is1, s, and stk)

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)
 apply auto done
```

∃ 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

first second third t2 s stk

first second third to1 s (exec is1 s stk)

to2

r1

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

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

(t2 = is1, s, and stk)
(to2 = is1, s, and stk)

when t2 is is1 (n -> 1) ?
when t2 is s (n -> 2) ?
when t2 is stk (n -> 3) ?

new types ->

new constants ->

```
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)"
```

first second third

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)
 apply auto done
```

t2 first second third

to1 r1

to2

∃ r1 : rule. True

→

∃ r1 : rule. (r1 = exec.induct)
∃ t1 : term. (t1 = exec)
∃ to1 : term_occurrence ∈ t1 : term. (to1 = exec)
r1 is_rule_of to1 True! r1 (= exec.induct) is a lemma about to1 (= exec).
∧
∀ t2 : term ∈ induction_term. (t2 = is1, s, and stk)
∃ to2 : term_occurrence ∈ t2 : term. (to2 = is1, s, and stk)
∃ n : number.
is_nth_argument_of (to2, n, to1) when t2 is is1 (n -> 1)
∧ when t2 is s (n -> 2)
t2 is_nth_induction_term n when t2 is stk (n -> 3)

new types ->

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

new constants ->

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

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

new lemma ->

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

a model proof ->

```
apply(induct is1 s stk rule:exec.induct)
 apply auto done
```

∃ 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

first second third t2

first second third to1

r1

to2

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

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

(t2 = is1, s, and stk)
(to2 = is1, s, and stk)

when t2 is is1 (n -> 1)
when t2 is s (n -> 2)
when t2 is stk (n -> 3)

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

DEMO!

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%)

Overall, the resits 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 resits are good.
But finding out what variables to generalise remains as a challenge!

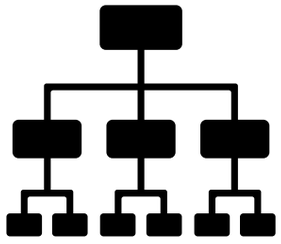
fin.

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

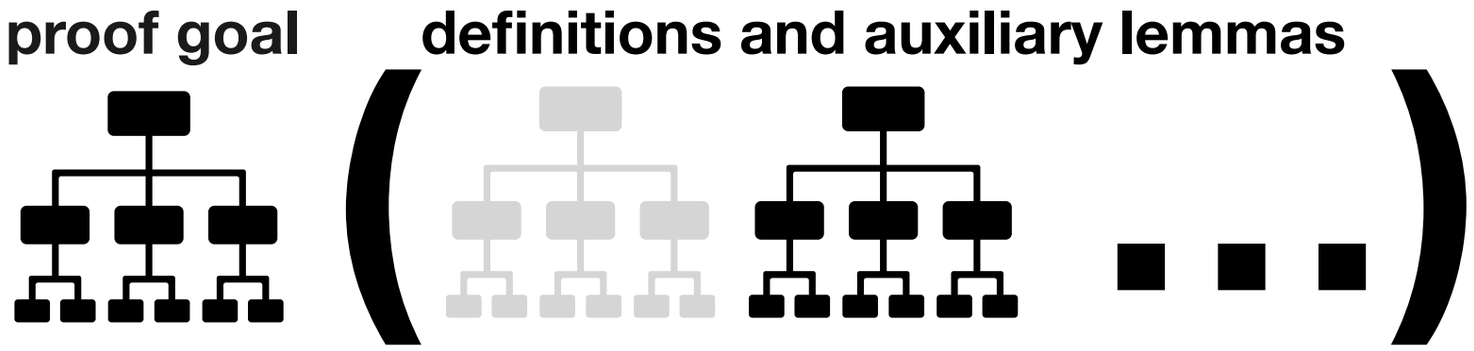
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions

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

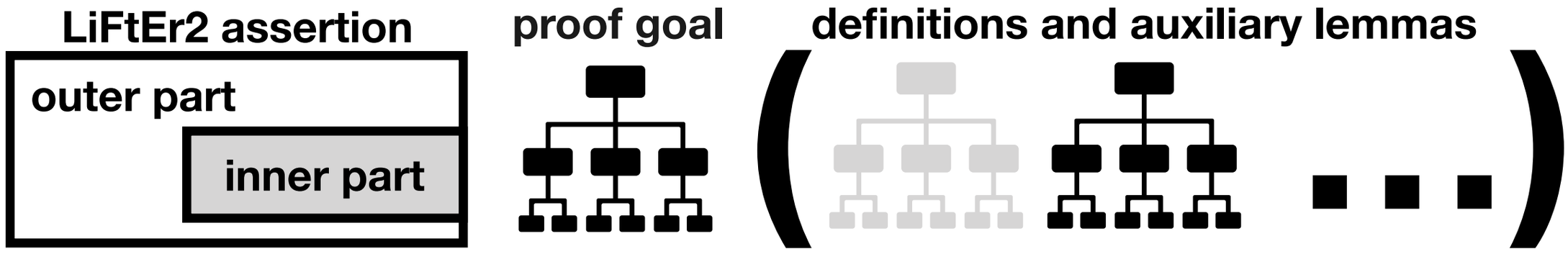
proof goal



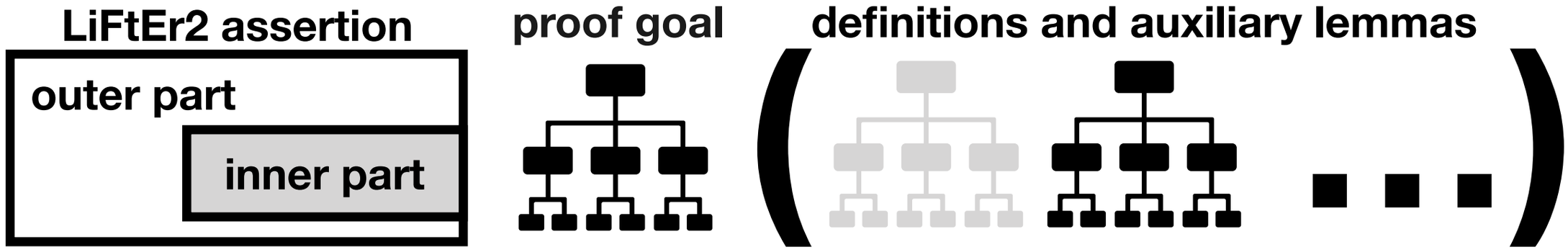
LiFtEr$_2$ (proof goal * induction arguments) -> bool
*** relevant definitions**



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

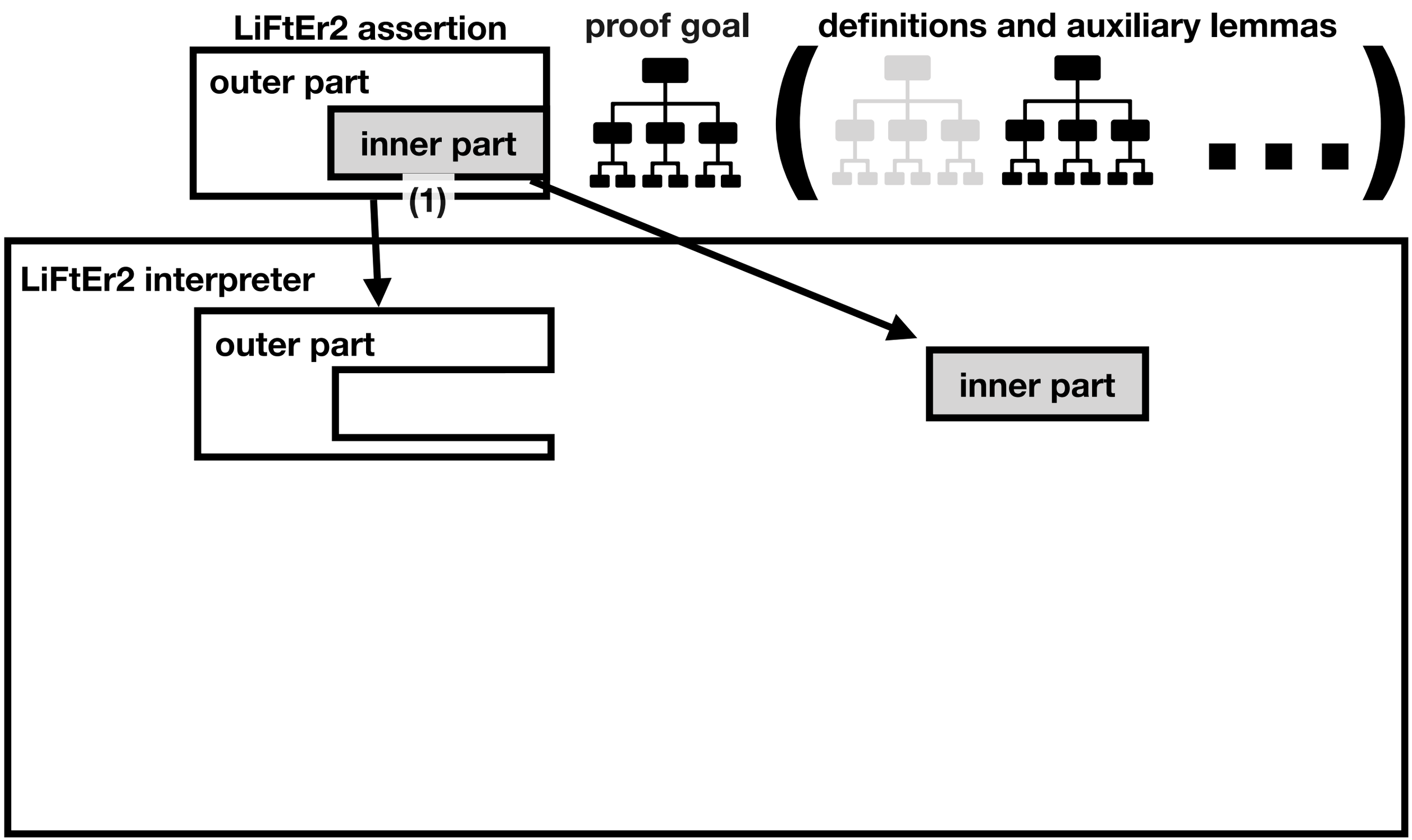


LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions

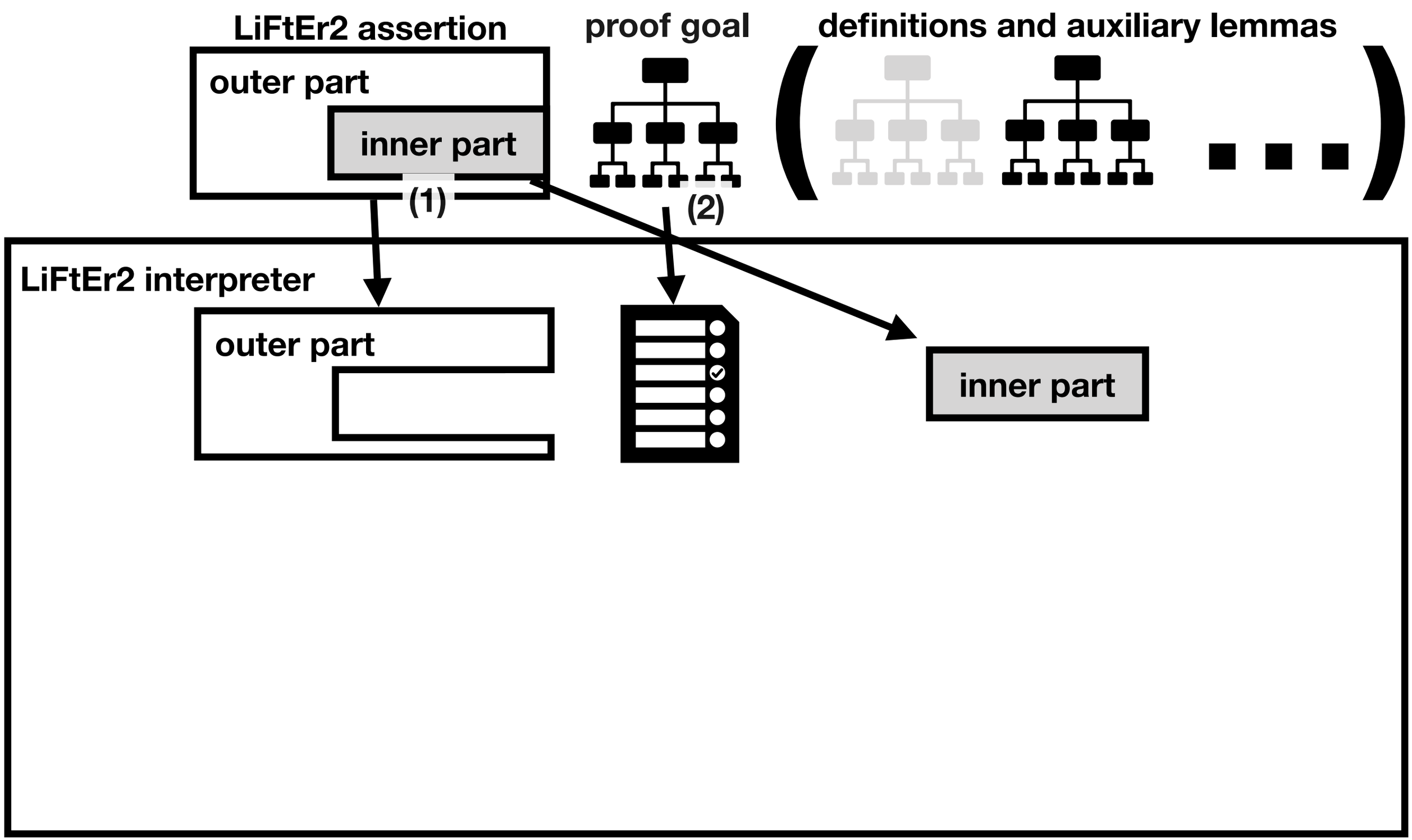


LiFtEr2 interpreter

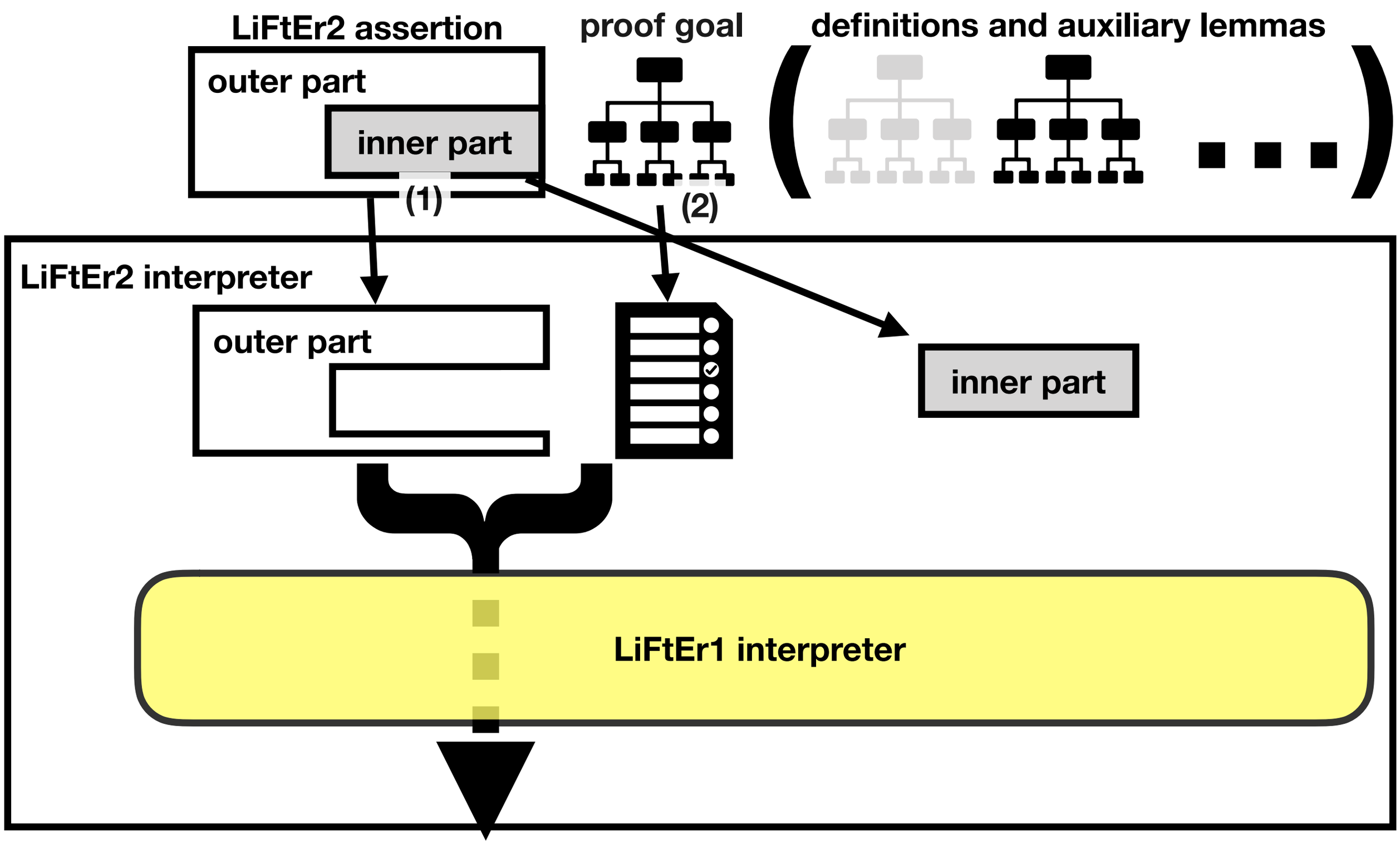
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions



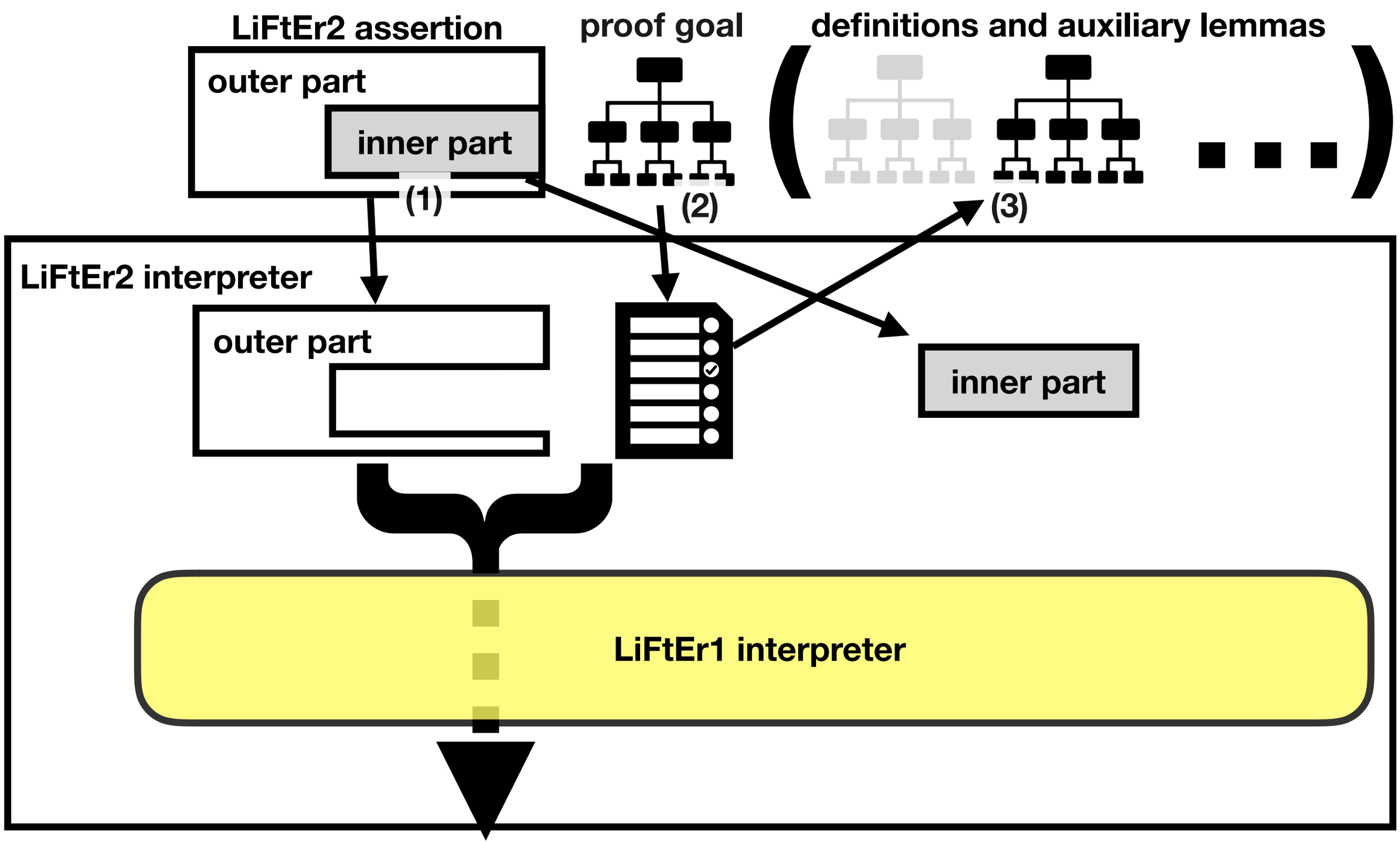
LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



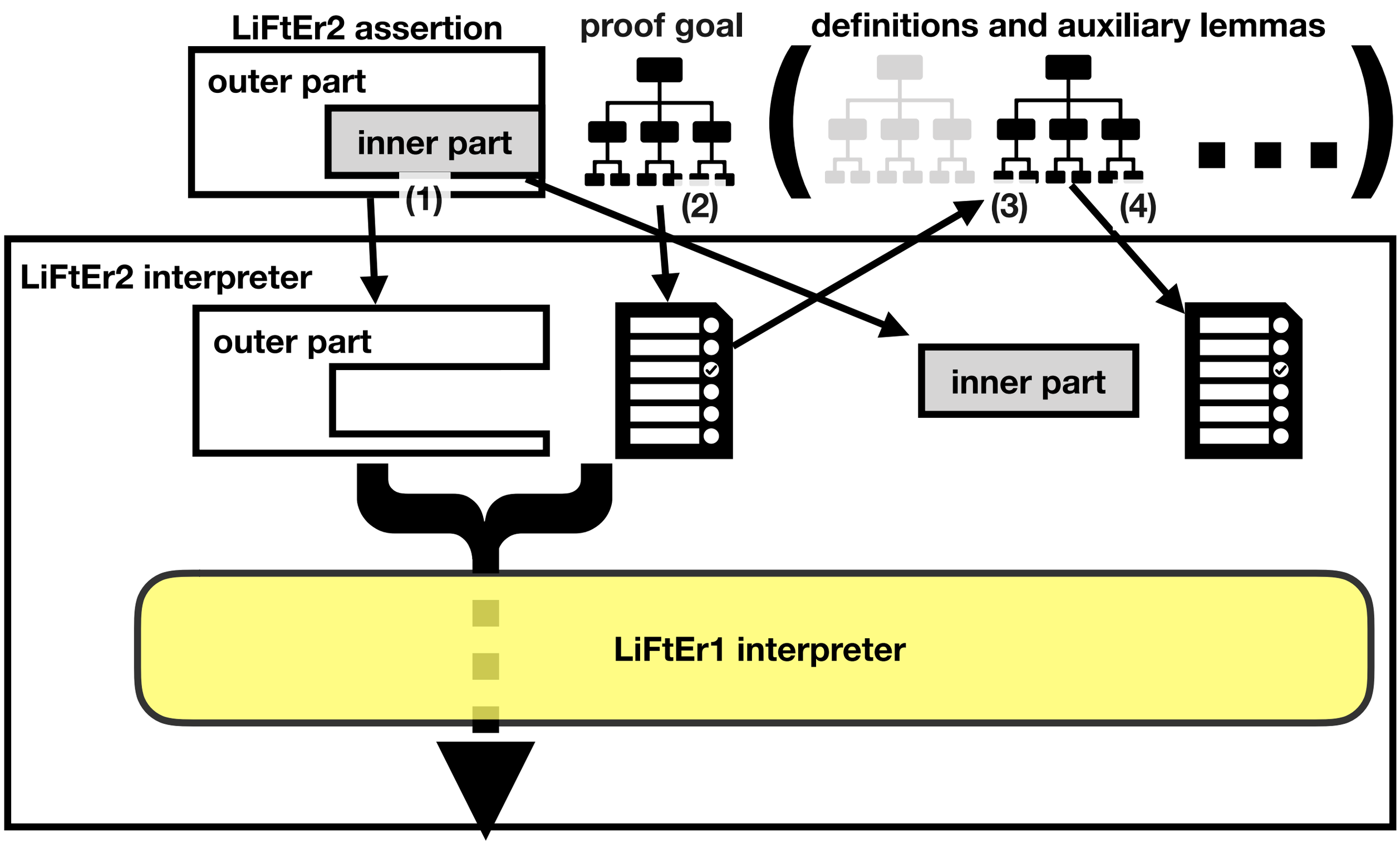
LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



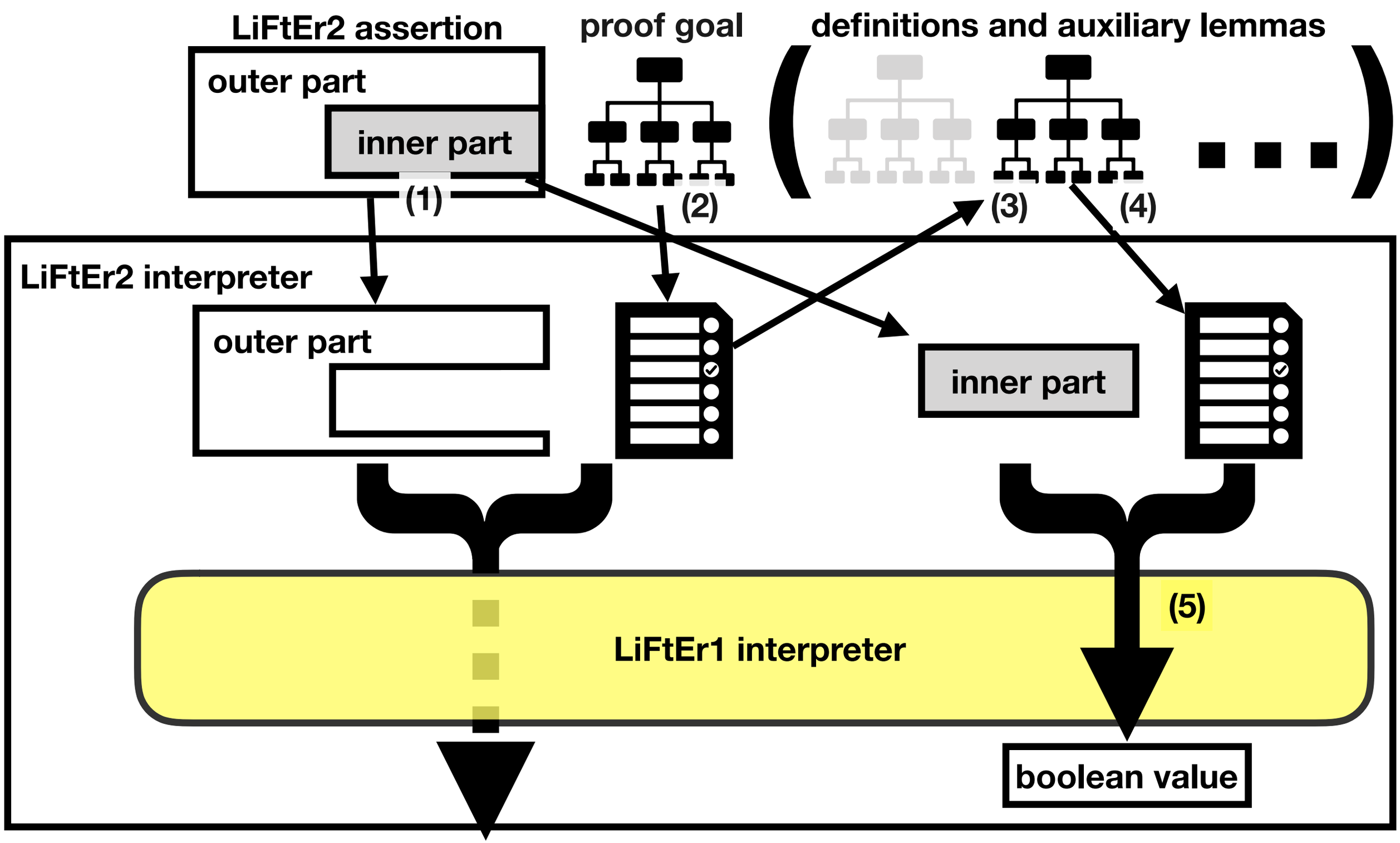
LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



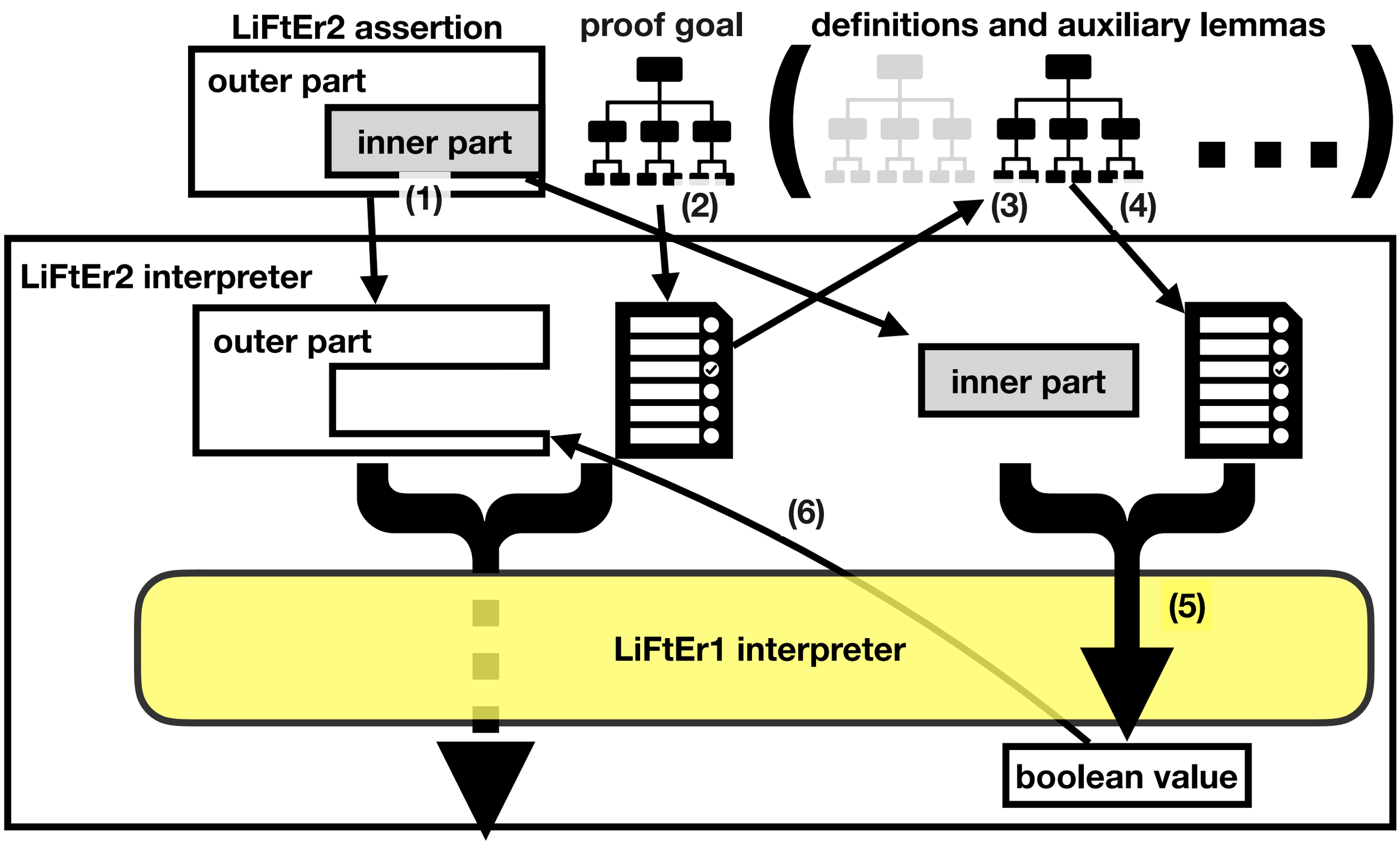
LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



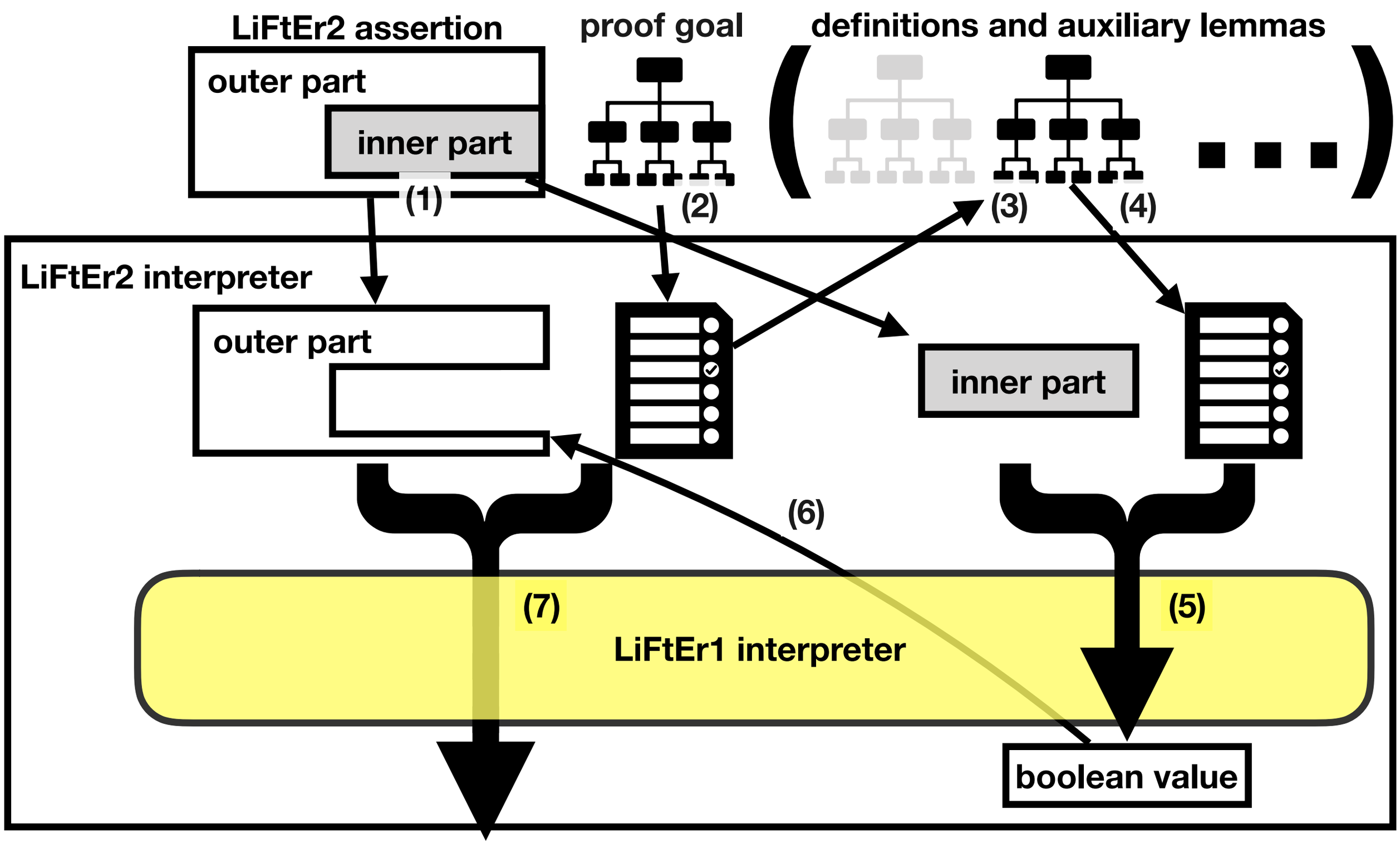
LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



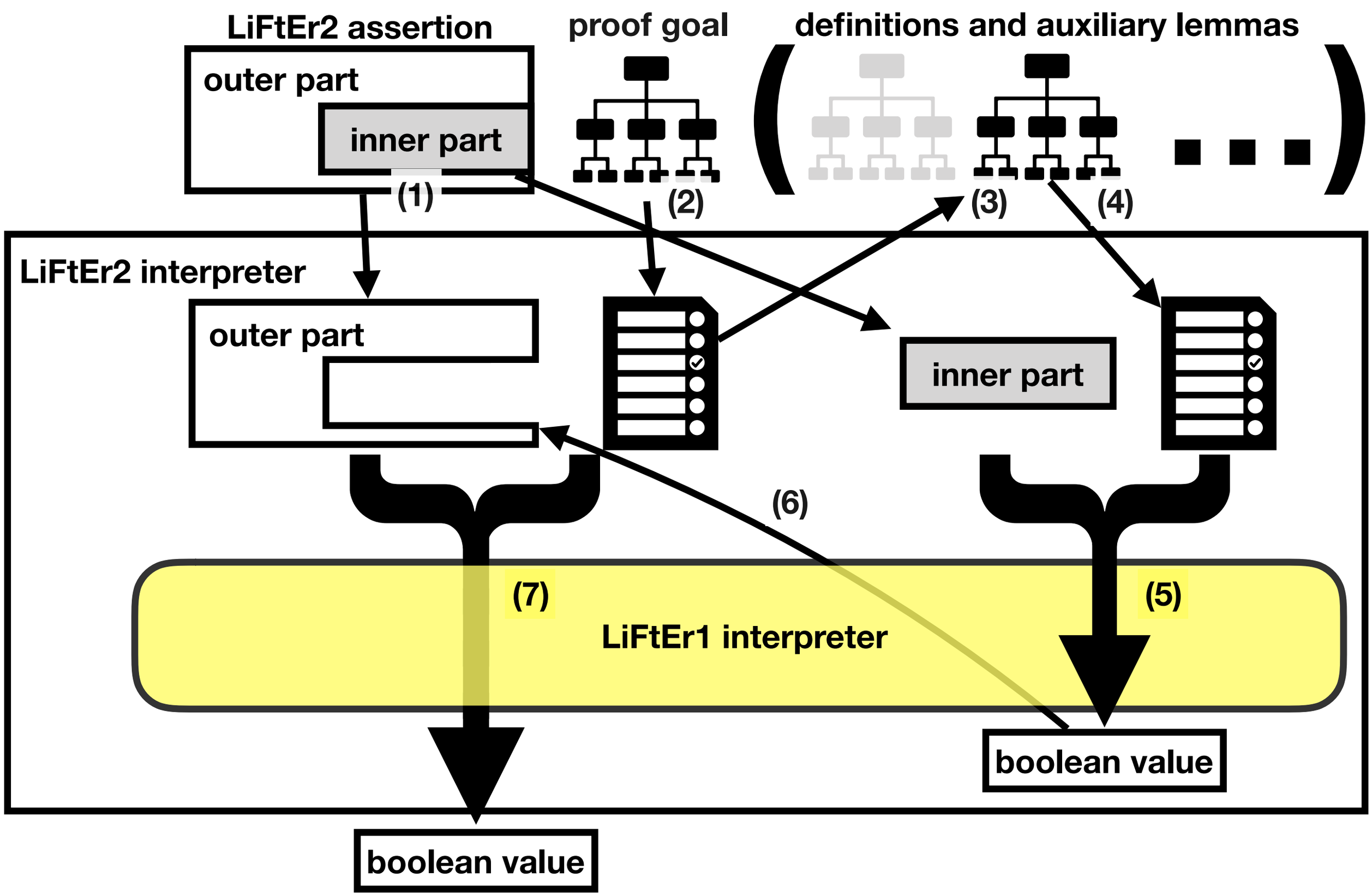
LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



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

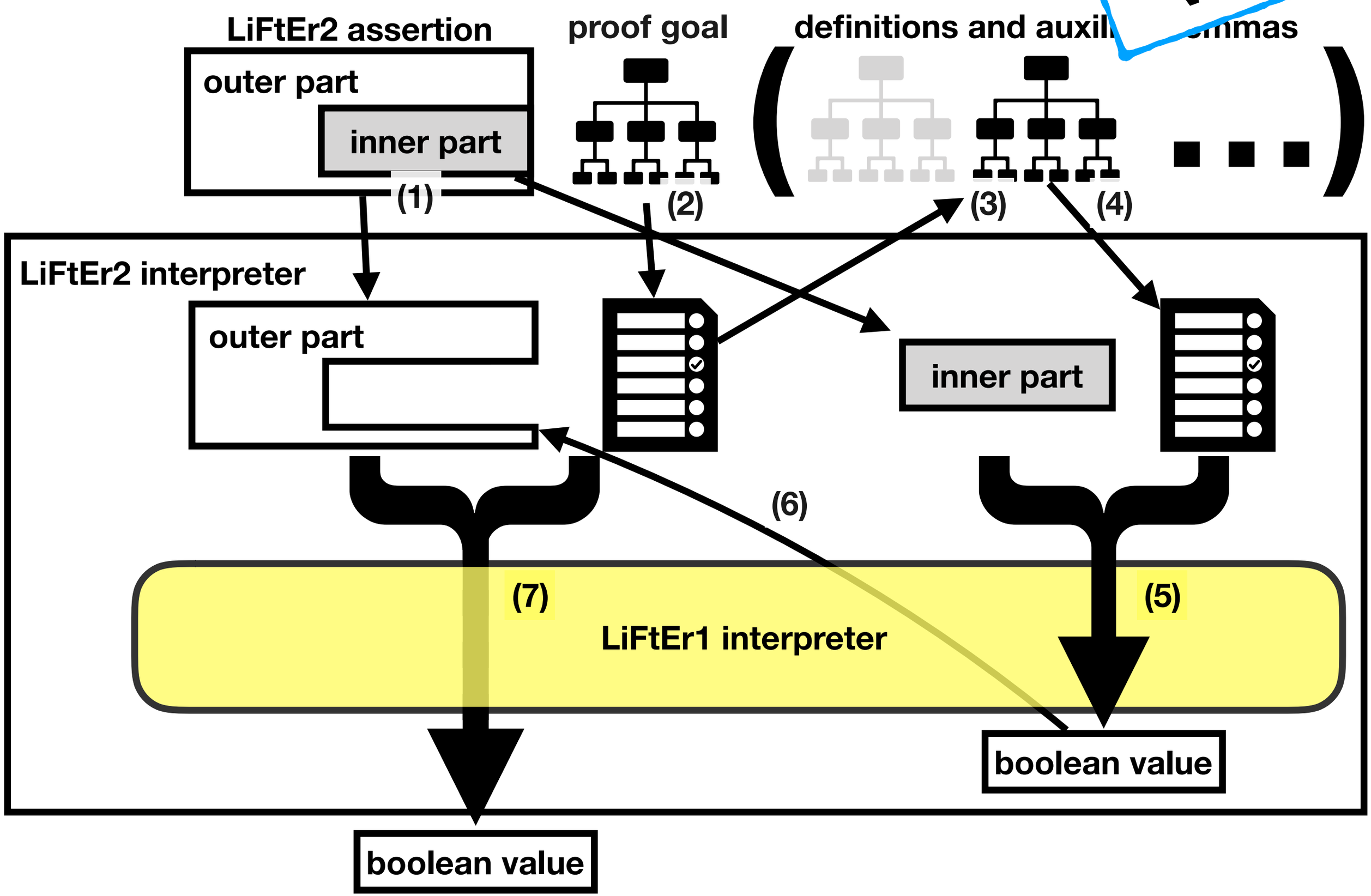


LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool

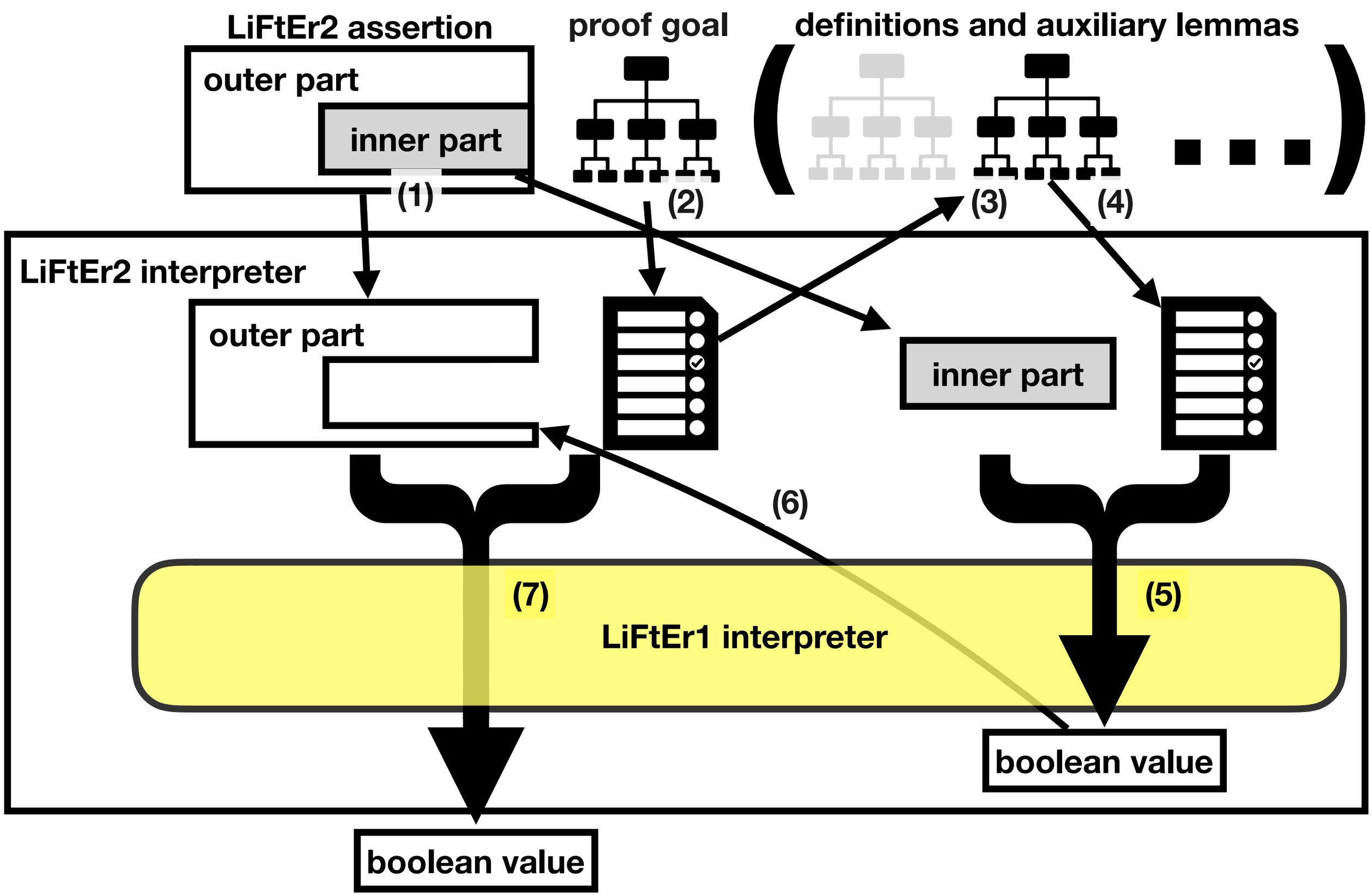


LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions

WIP!



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



<- 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
```

```
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

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

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
```

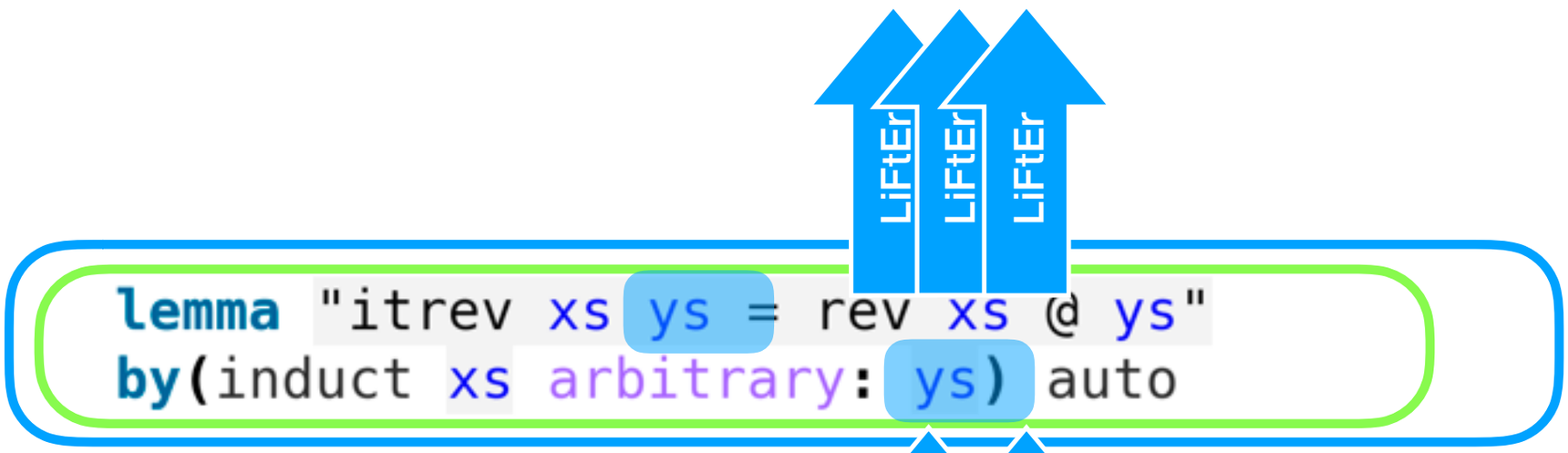
<- relevant definitions

```
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$_2$ (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)

```
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 LiFtEr LiFtEr

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

LiFtEr 2 LiFtEr 2

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