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

paper draft
(https://github.com/data61/PSL/blob/master/Smart_Induct/document/smart_induct.pdf)

smart_induct

Yutaka Nagashima
email: yutaka.nagashima(@)cvut.cz
twitter: YutakangE

DEMO1: PSL

ta61/PSL/blob/master/slide/2020_Isabelle.pdf

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

10.42 (221/935)

DEMO1: PSL

ta61/PSL/blob/master/slide/2020_Isabelle.pdf

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: PSL

ta61/PSL/blob/master/slide/2020_Isabelle.pdf

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) | 241/535MB 1:17 AM

DEMO1: PSL

ta61/PSL/blob/master/slide/2020_Isabelle.pdf

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: PSL

ta61/PSL/blob/master/slide/2020_Isabelle.pdf

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) 1:05 AM

DEMO1: PSL

ta61/PSL/blob/master/slide/2020_Isabelle.pdf

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) 251/535MB 1:05 AM

DEMO1: PSL

ta61/PSL/blob/master/slide/2020_Isabelle.pdf

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) 1:05 AM

DEMO1: PSL

ta61/PSL/blob/master/slide/2020_Isabelle.pdf

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)

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

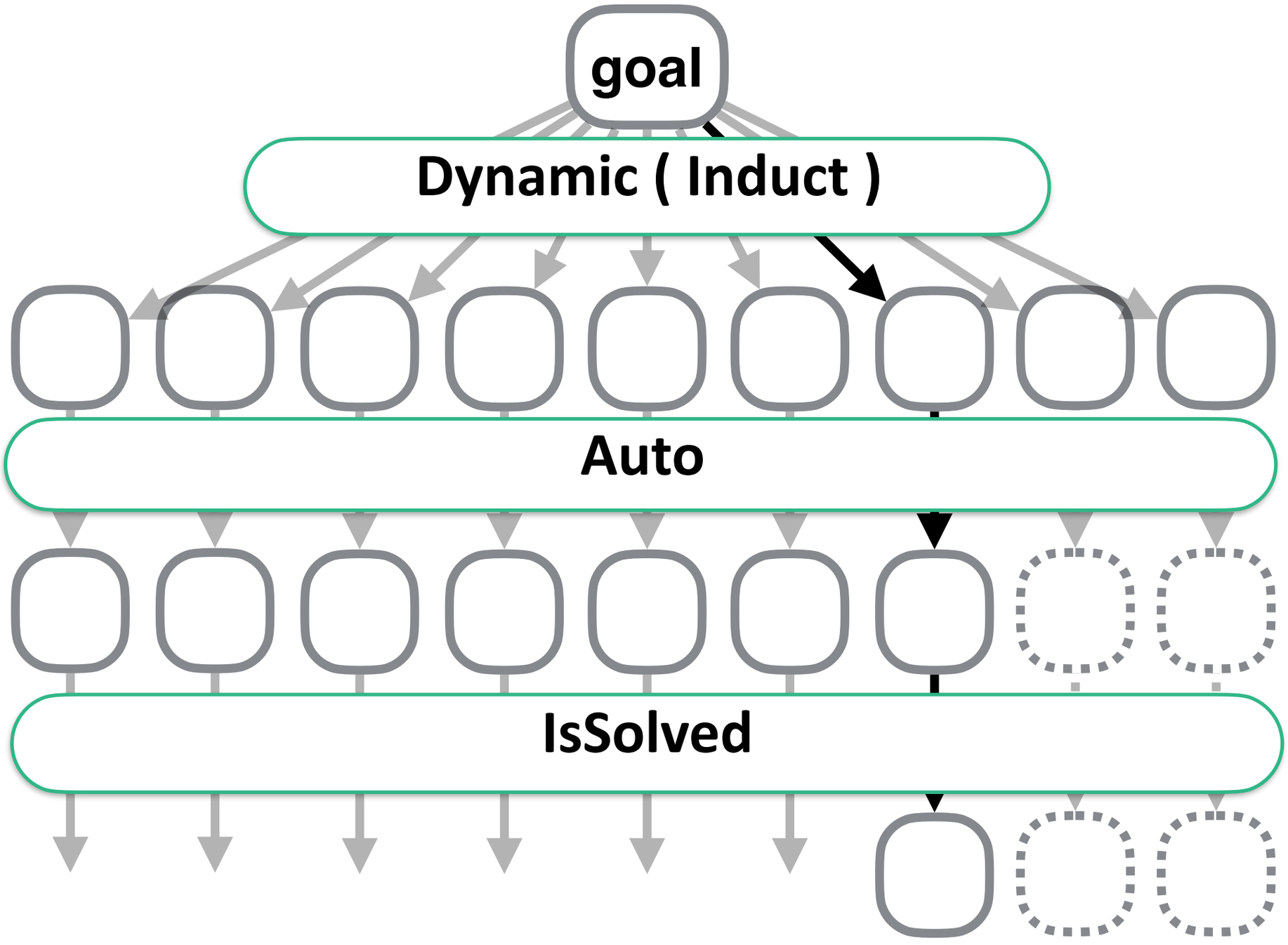
PSL: Proof Strategy Language

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

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

PSL: Proof Strategy Language

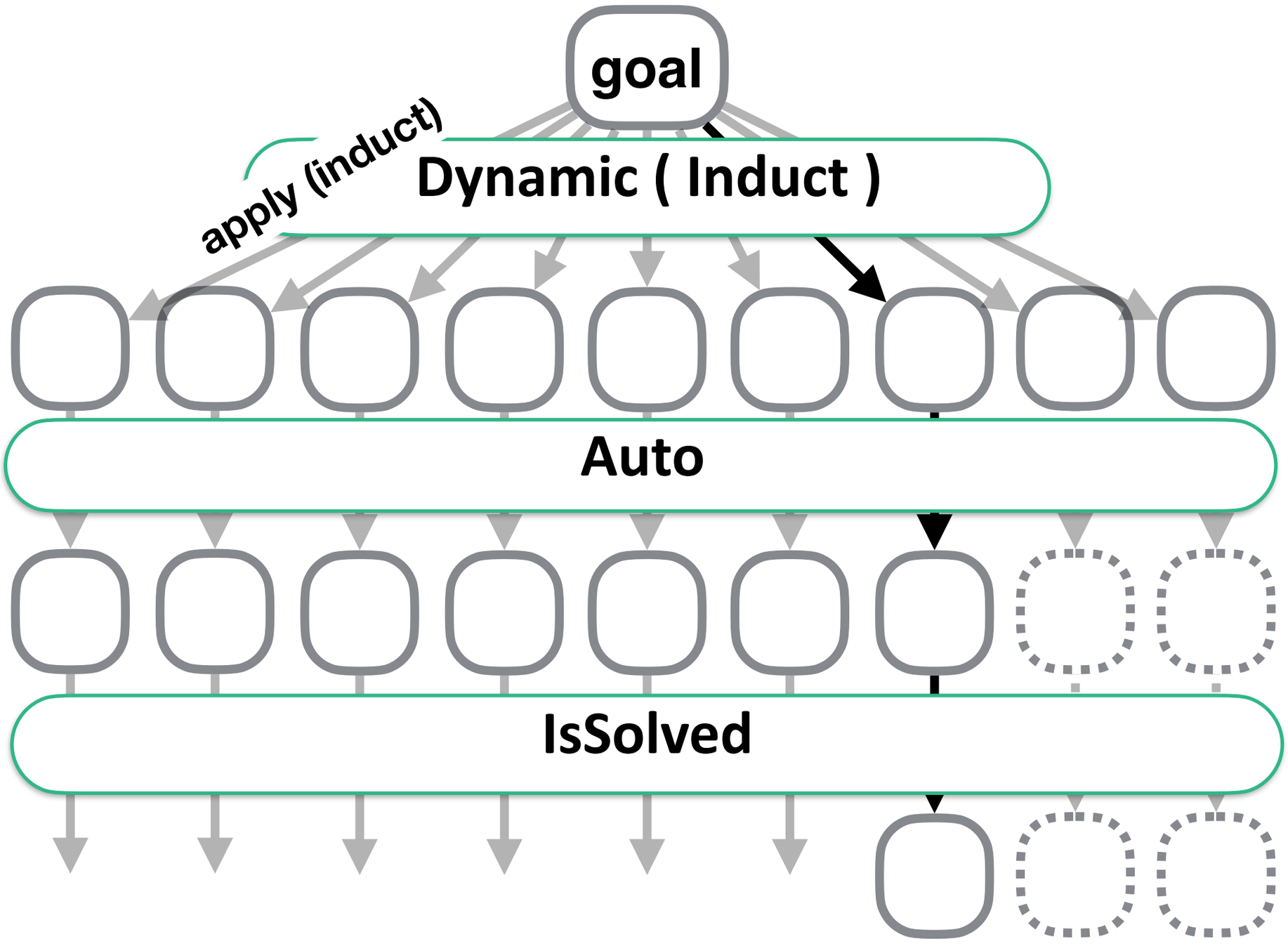
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

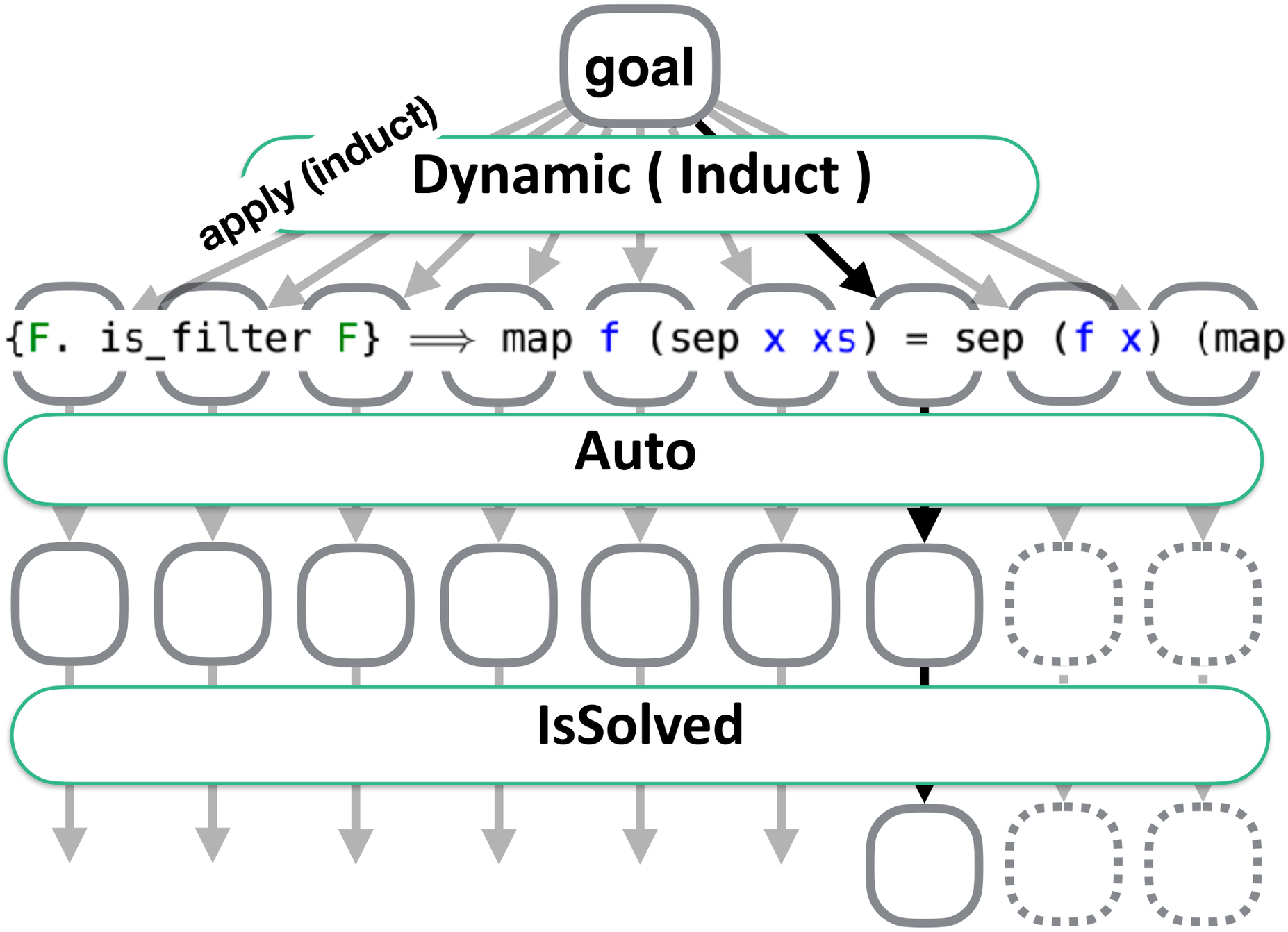
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

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

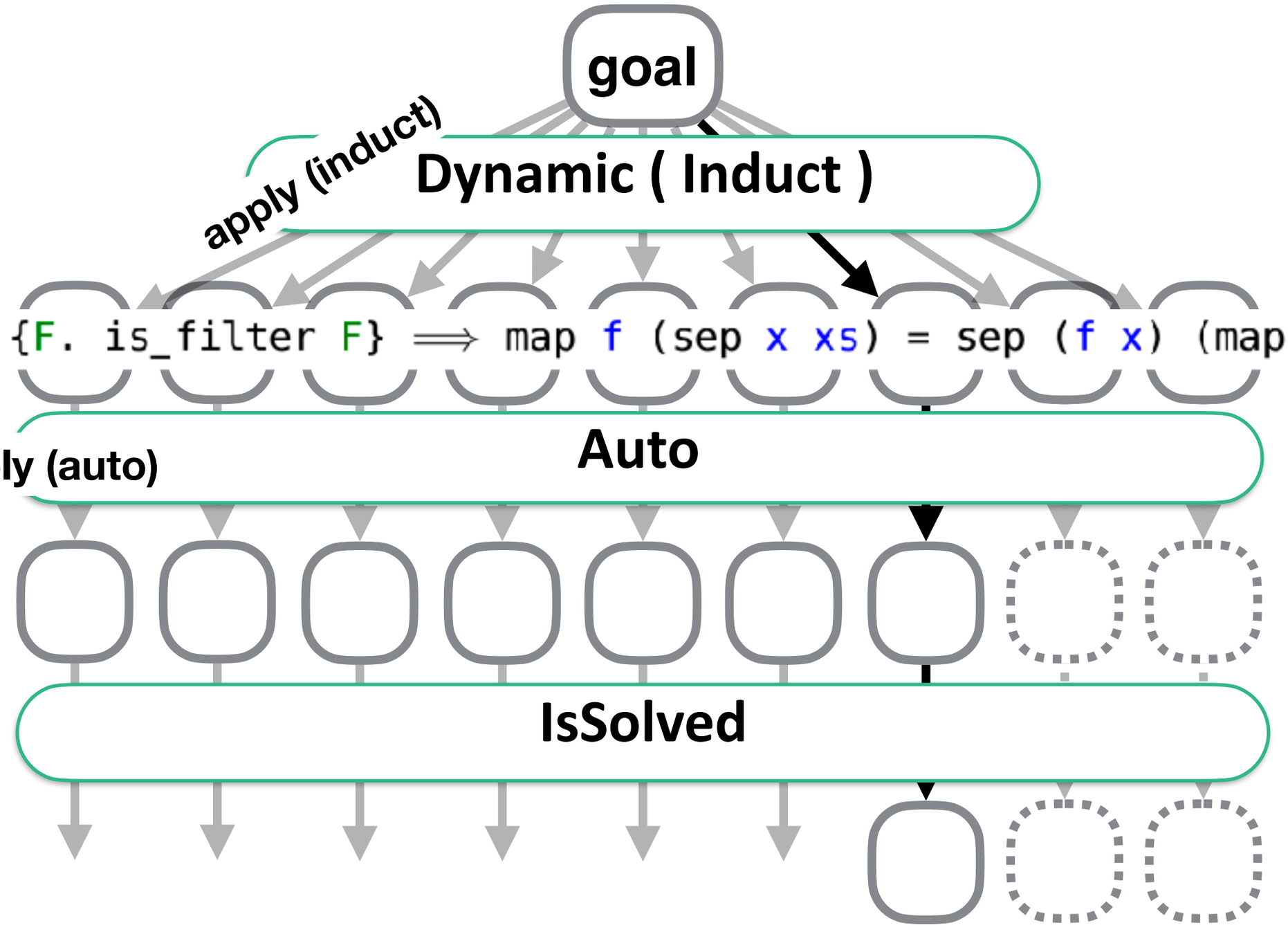



```
⋀y. y ∈ {F. is_filter F} ⟹ map f (sep x xs) = sep (f x) (map f xs)
```

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

PSL: Proof Strategy Language

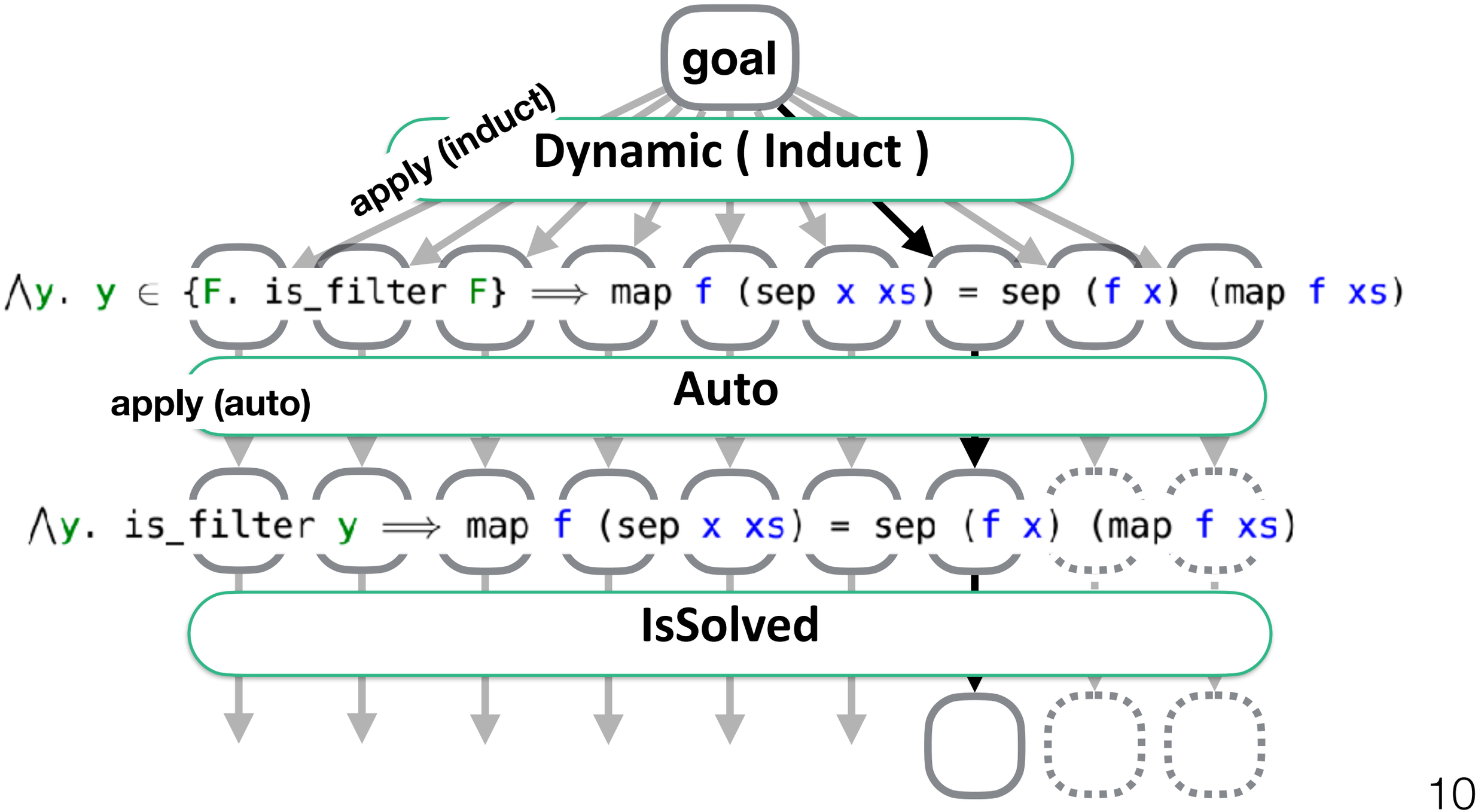
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

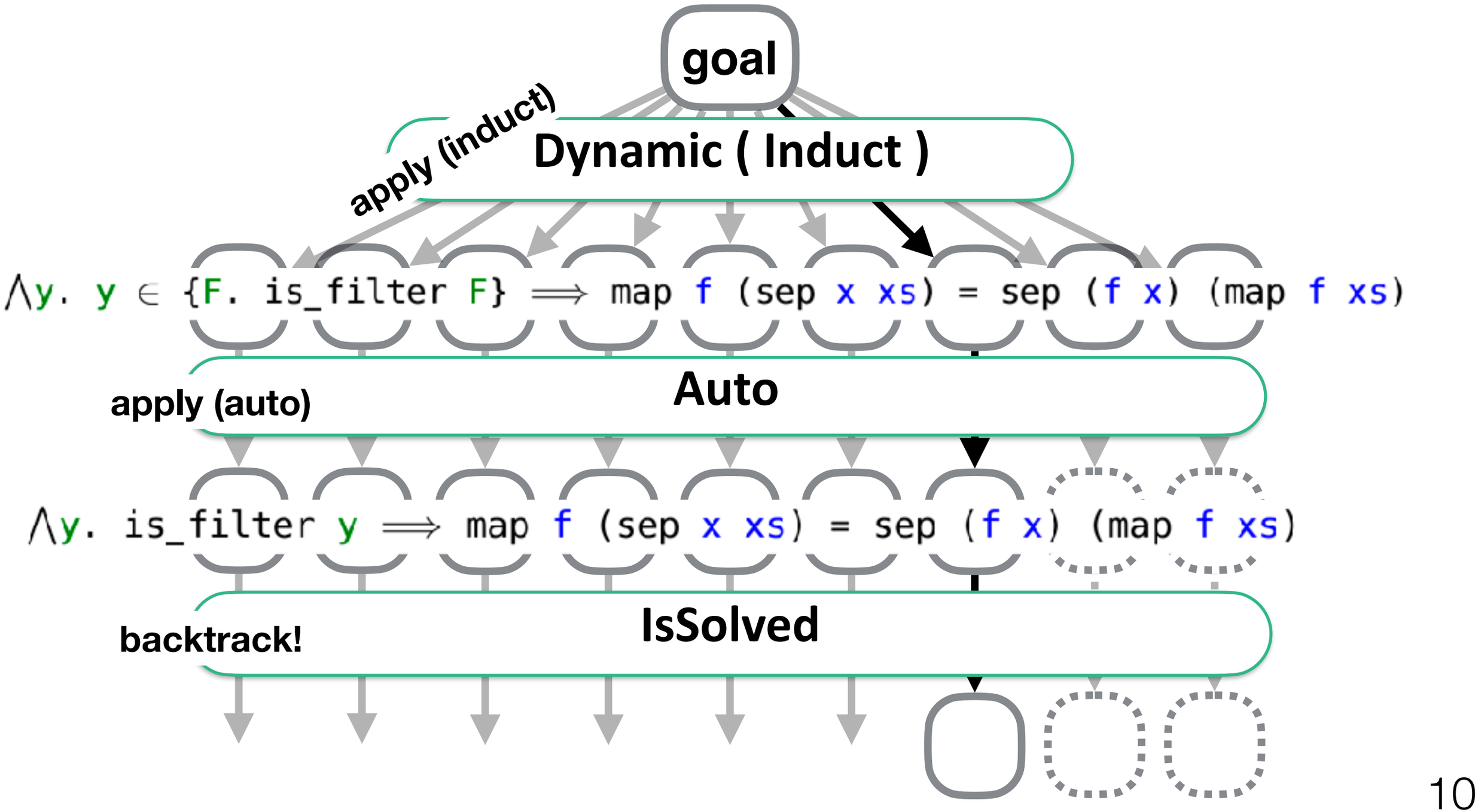
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

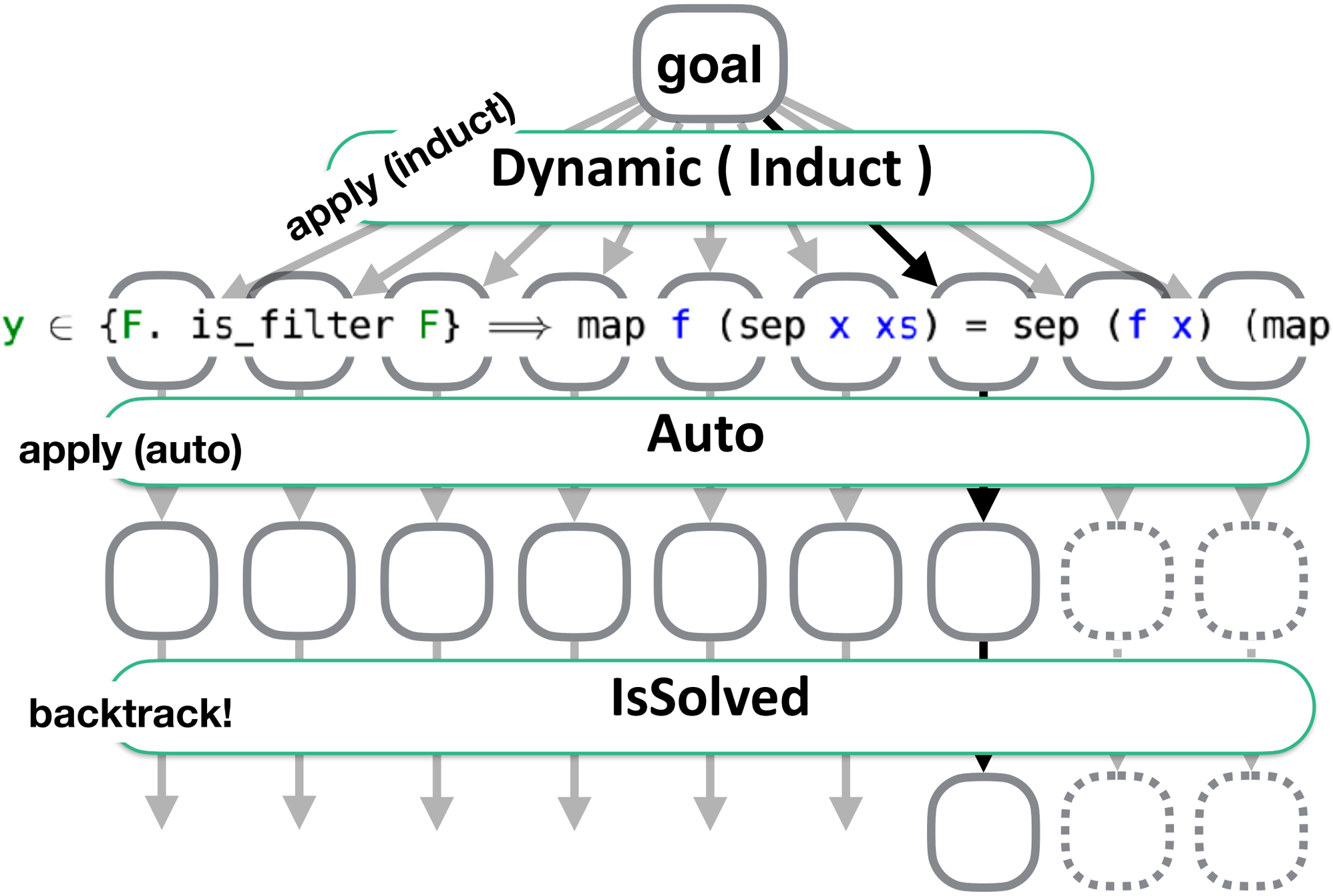
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

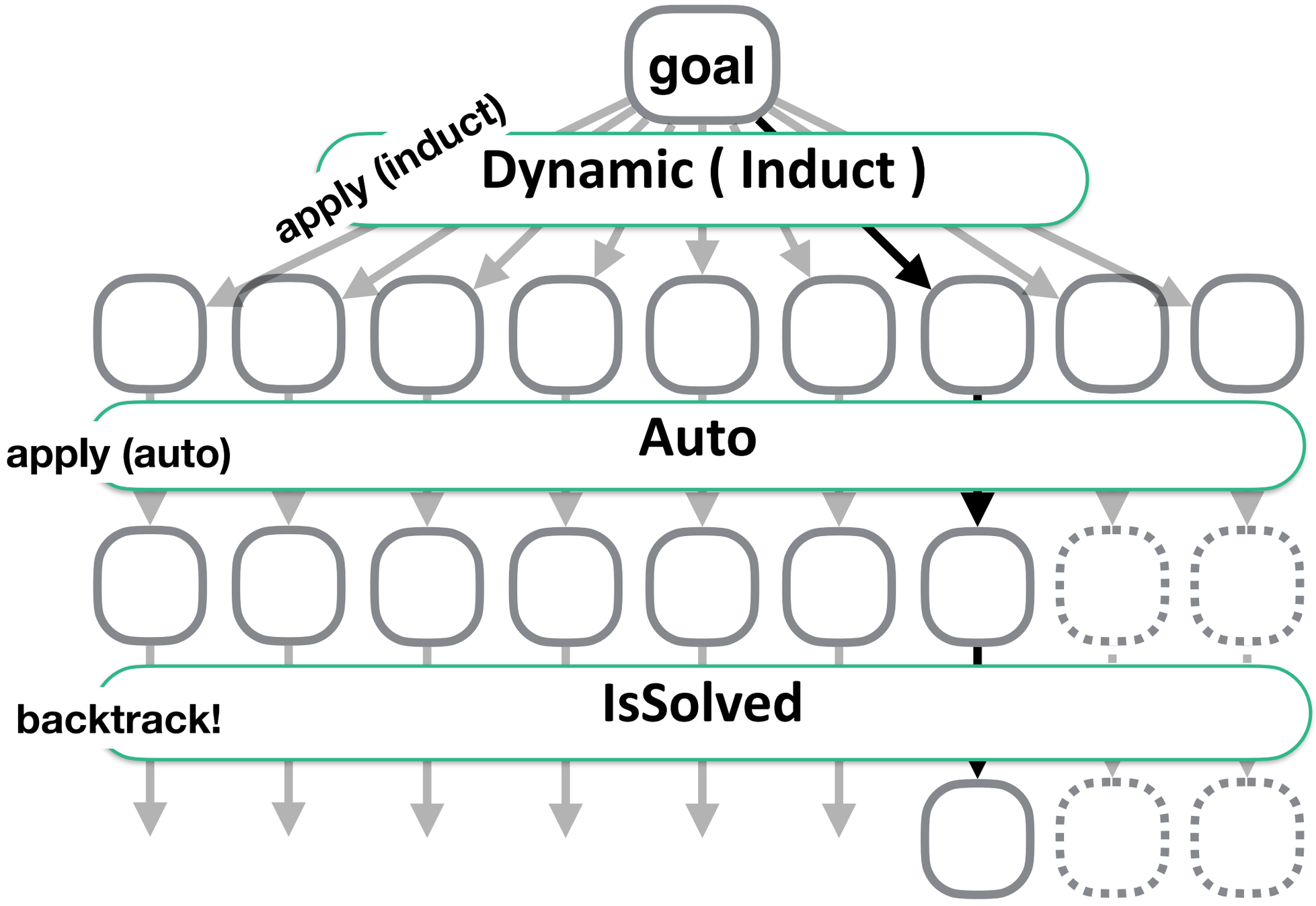
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

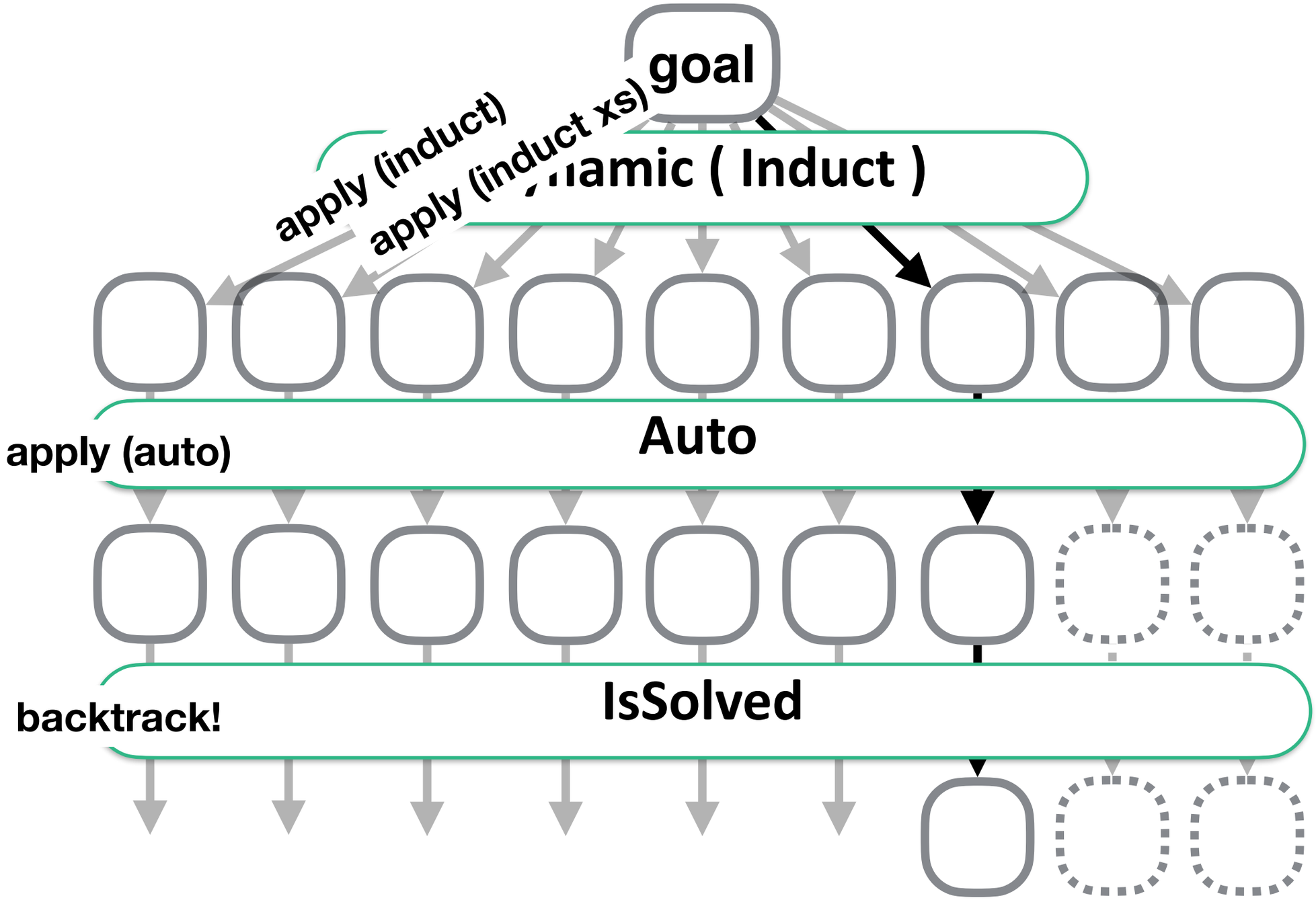
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

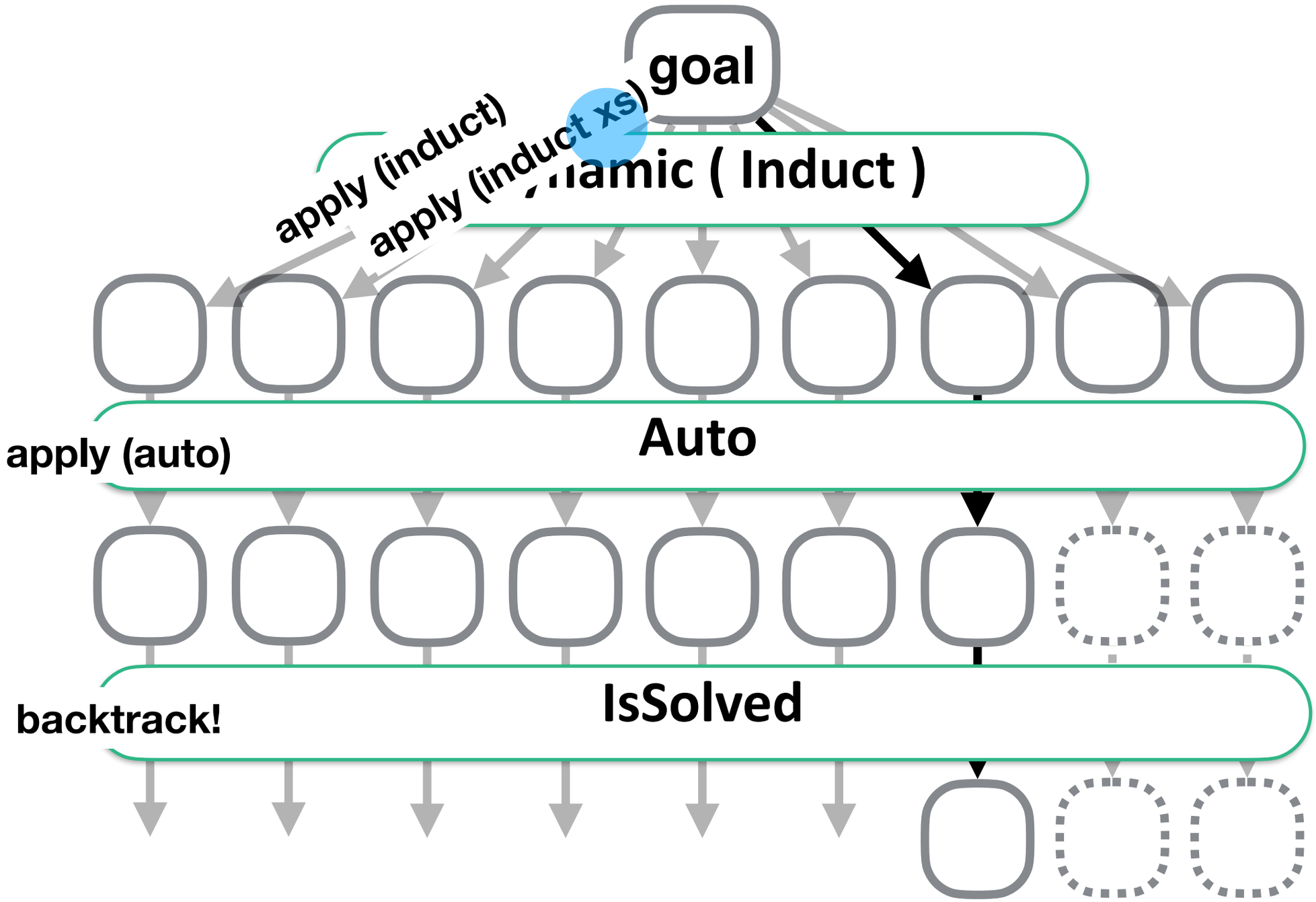
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

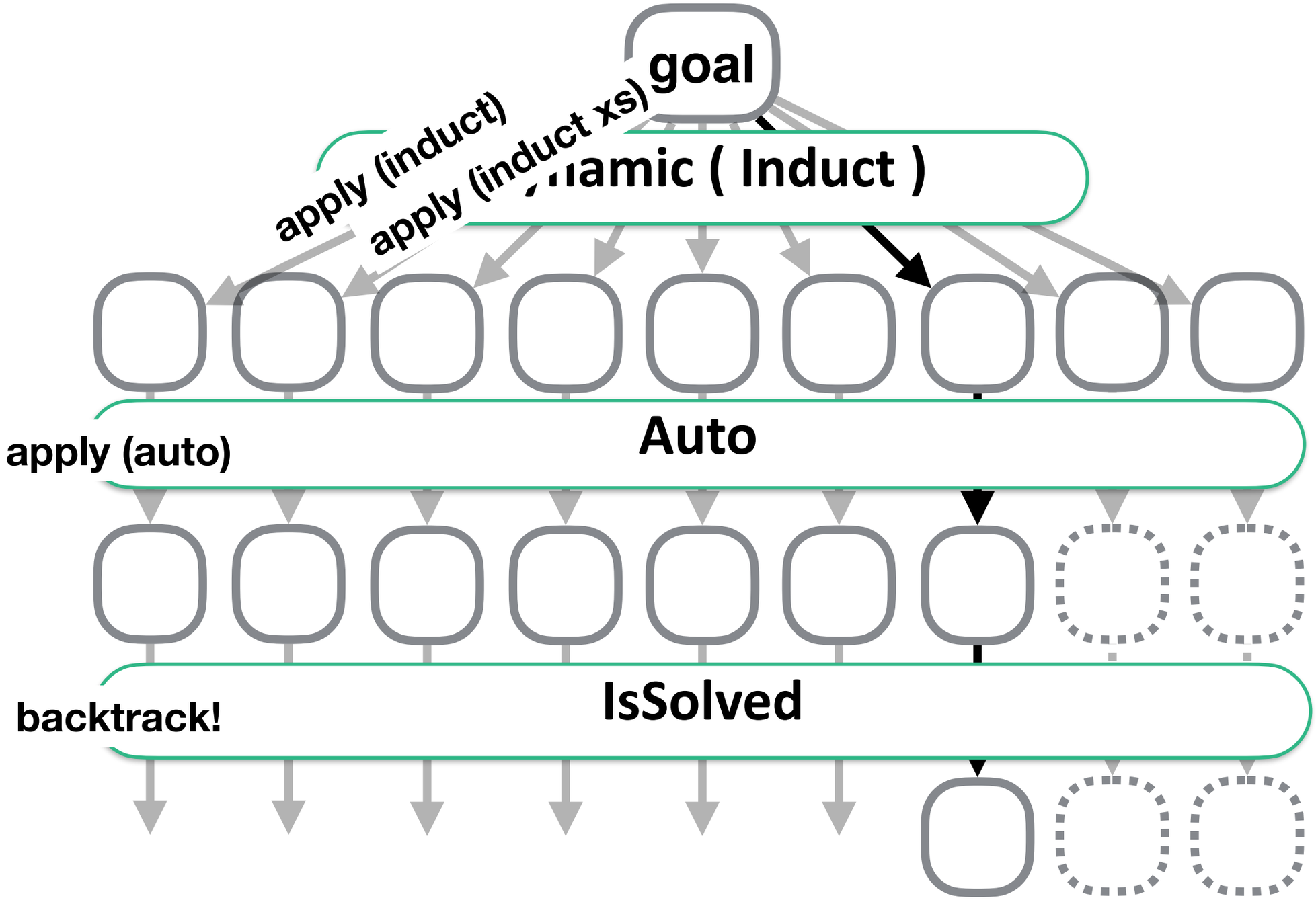
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

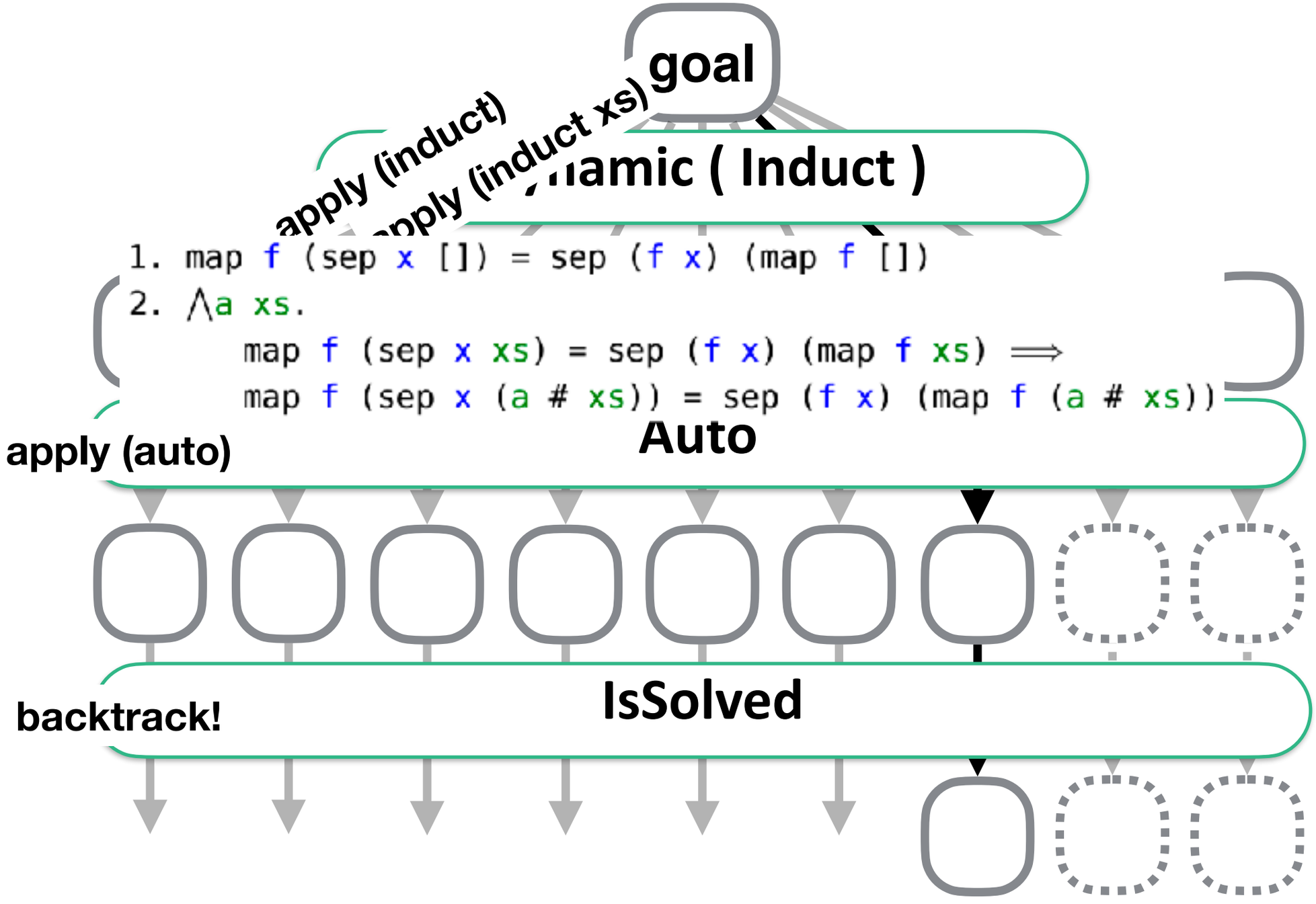
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

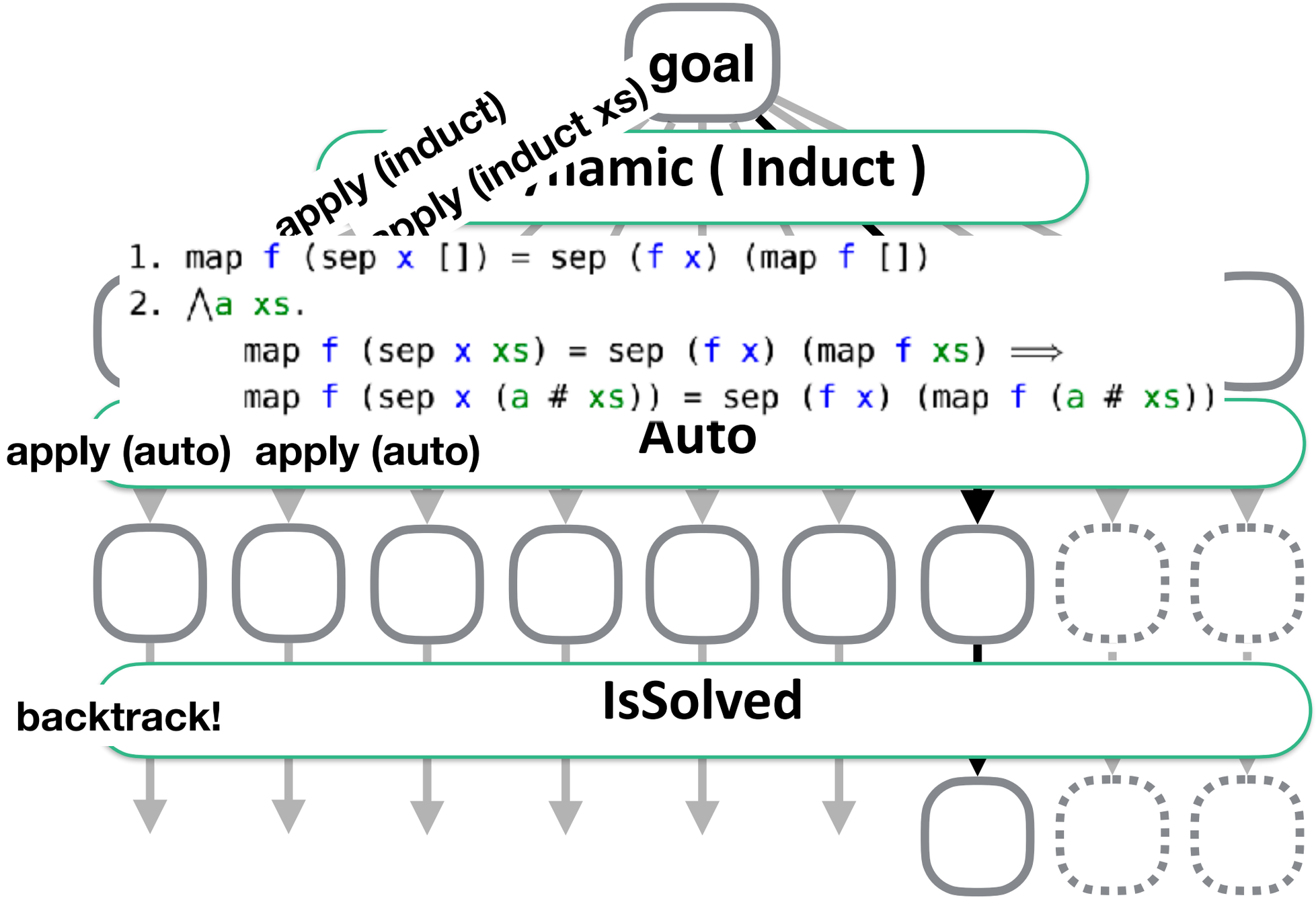
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

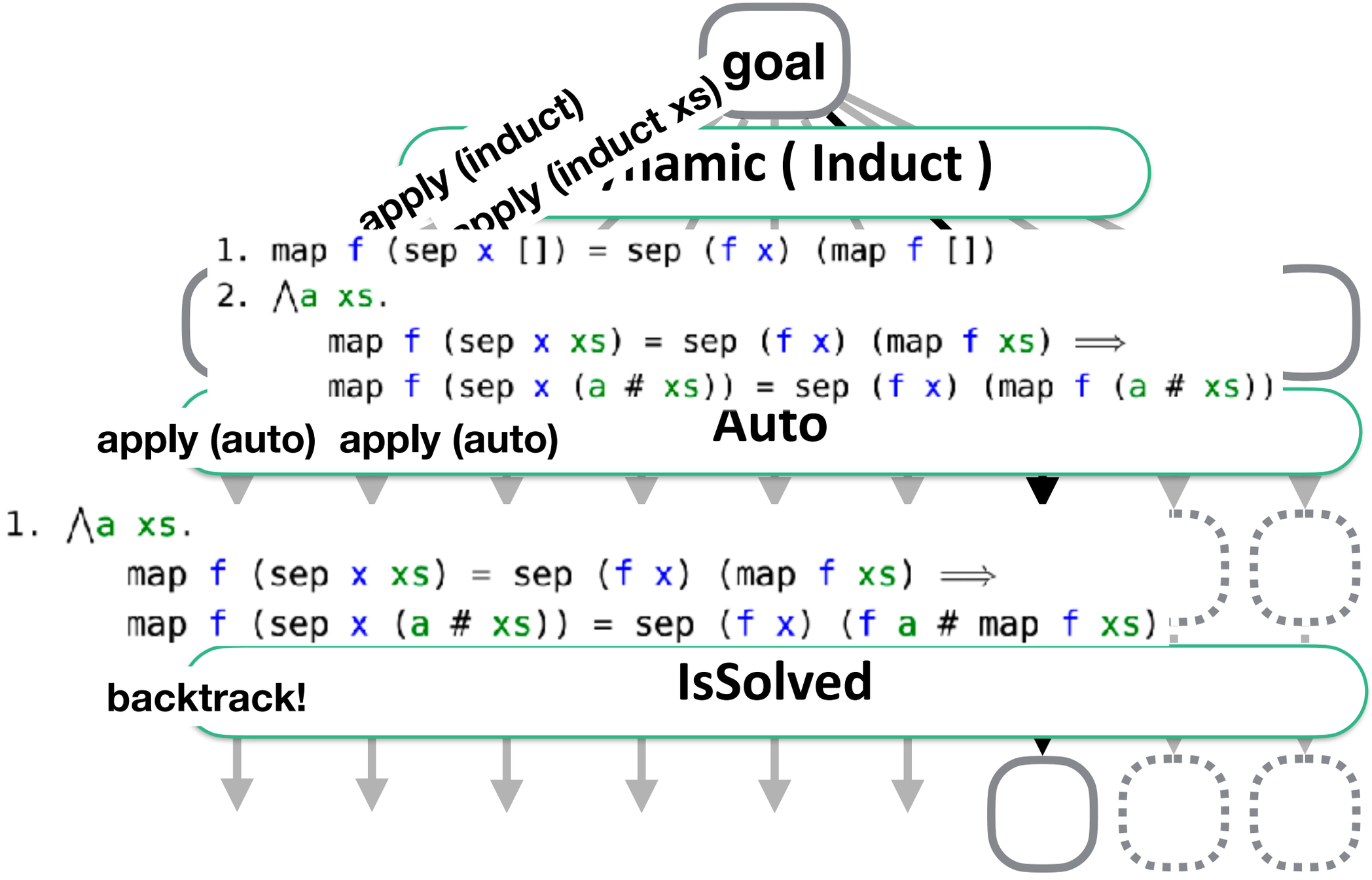
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

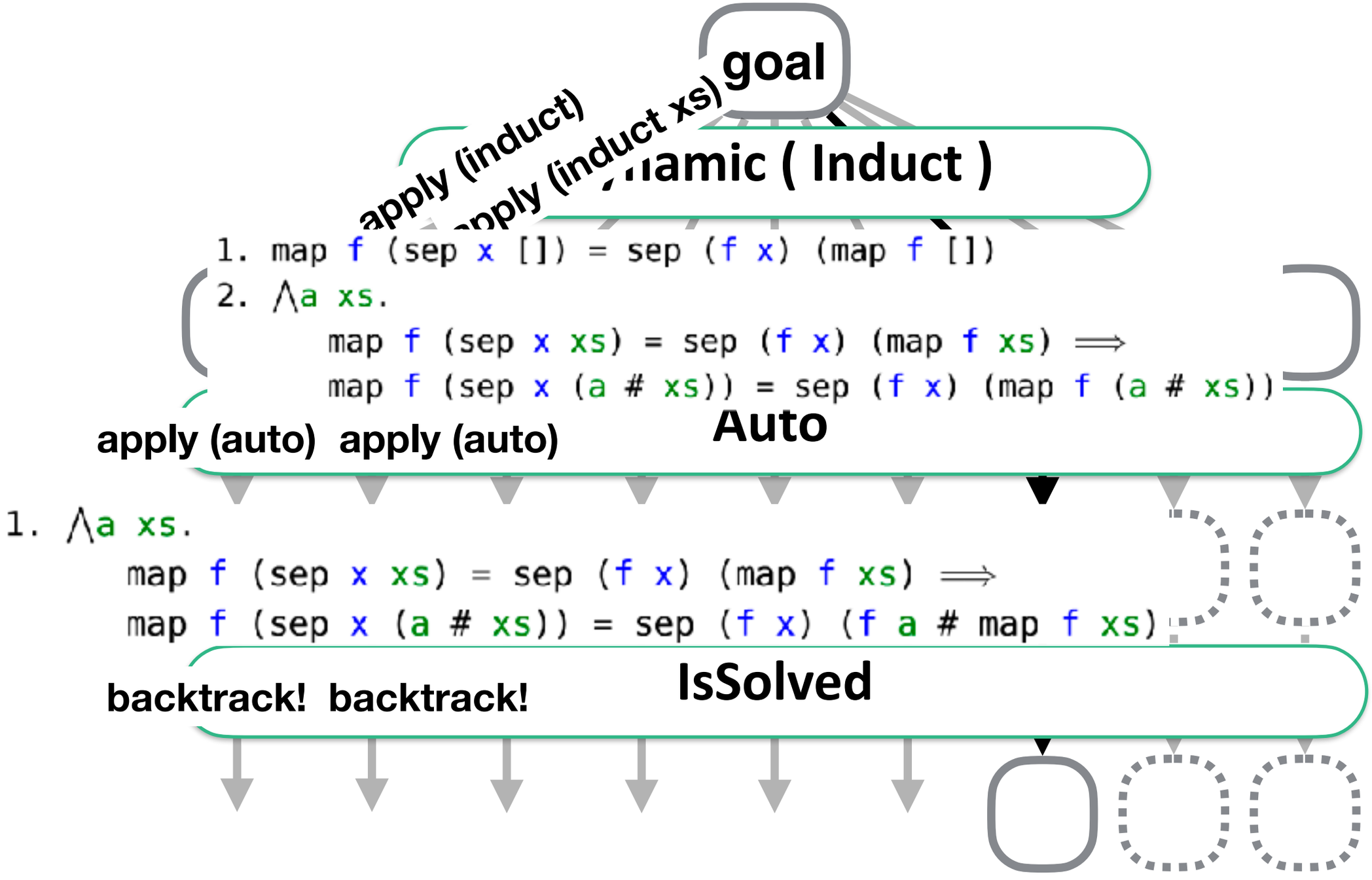
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

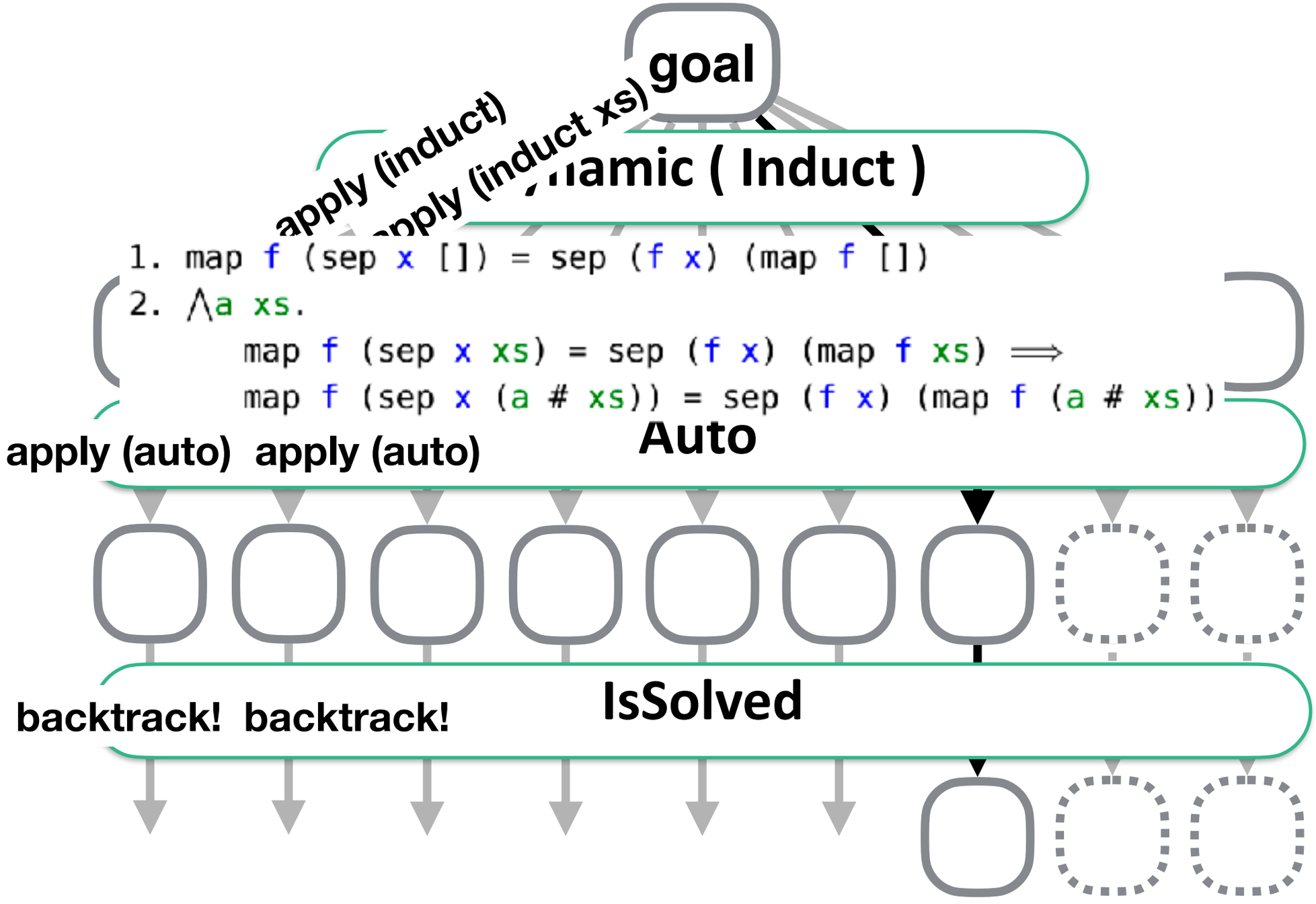
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

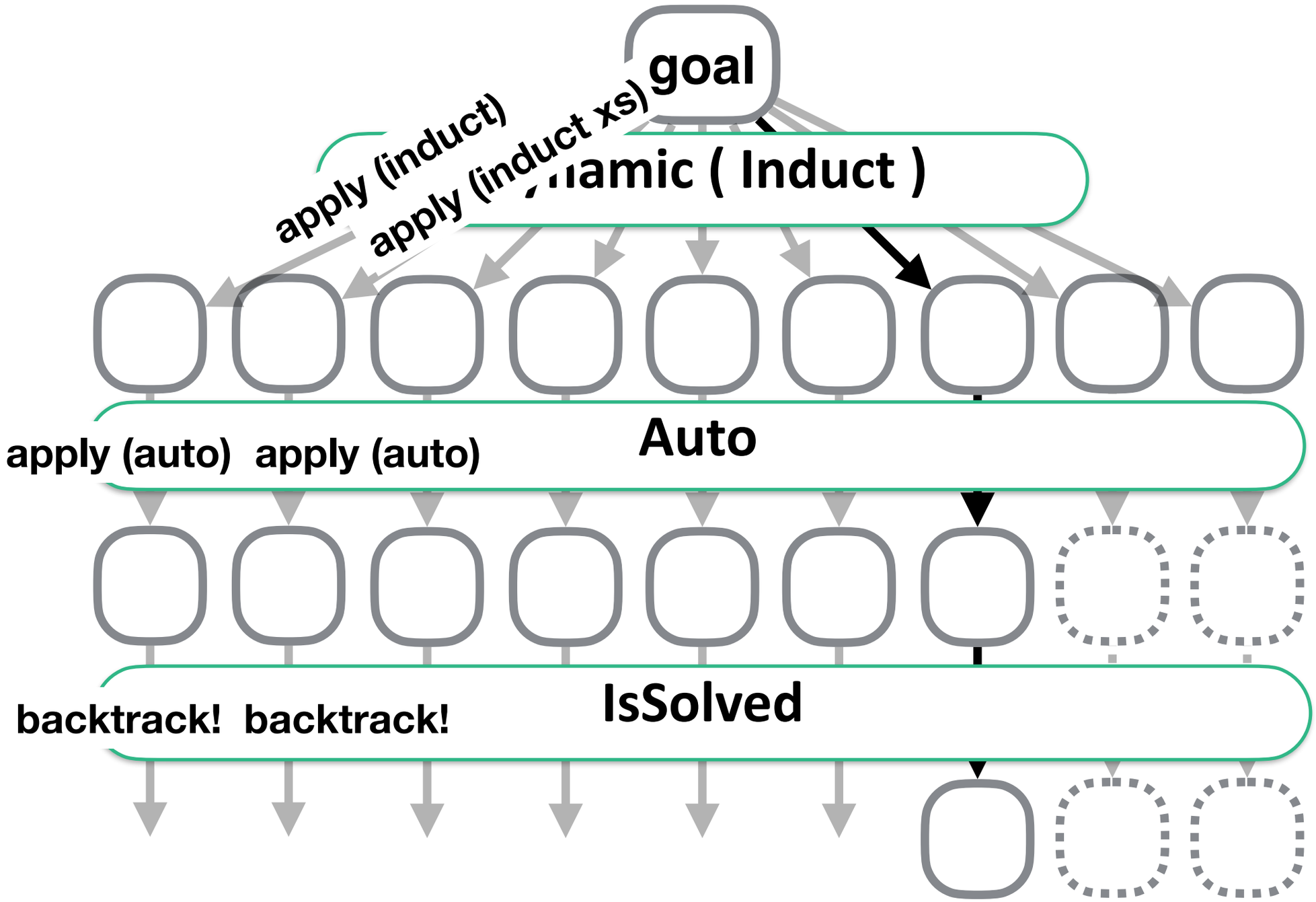
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

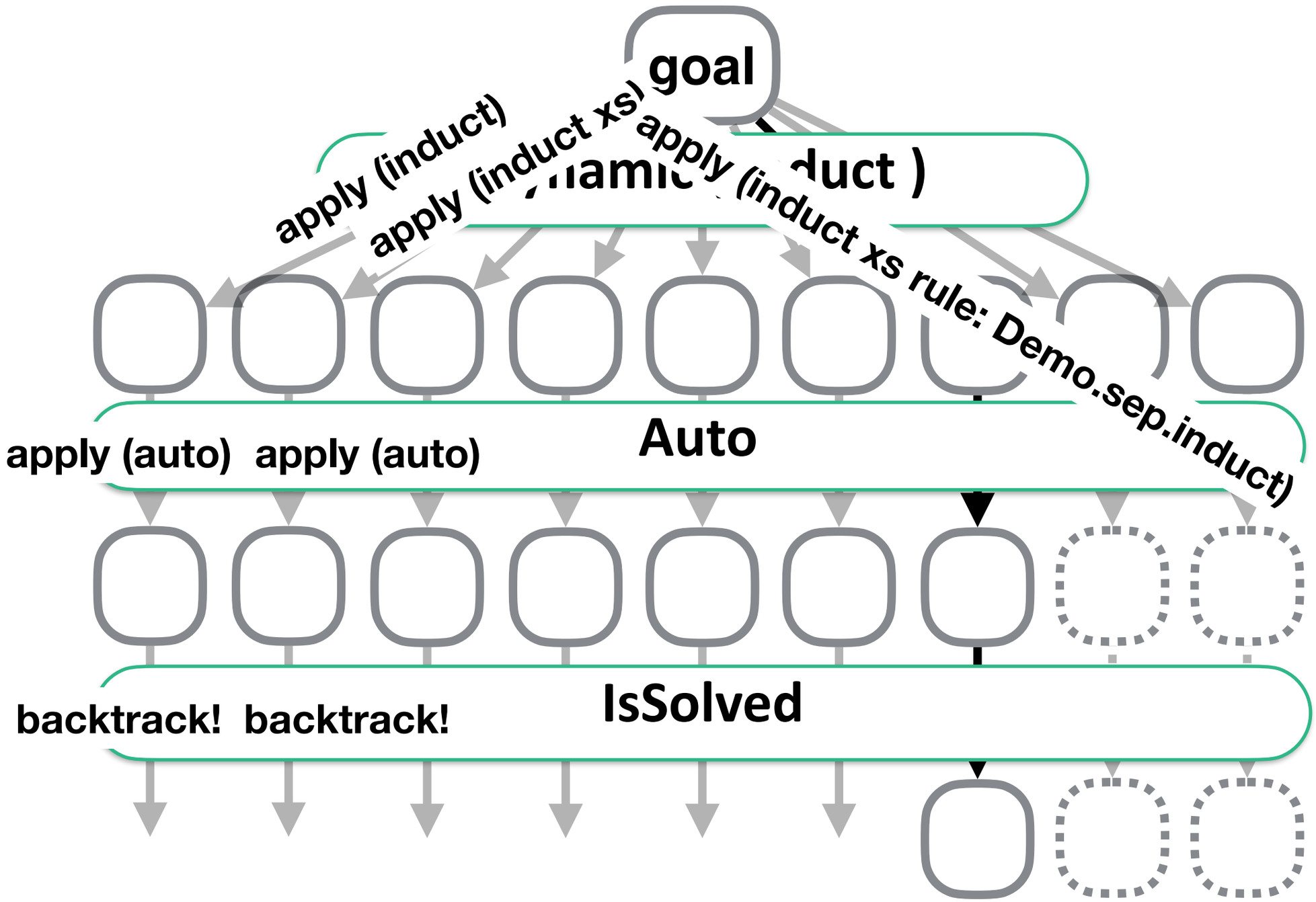
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

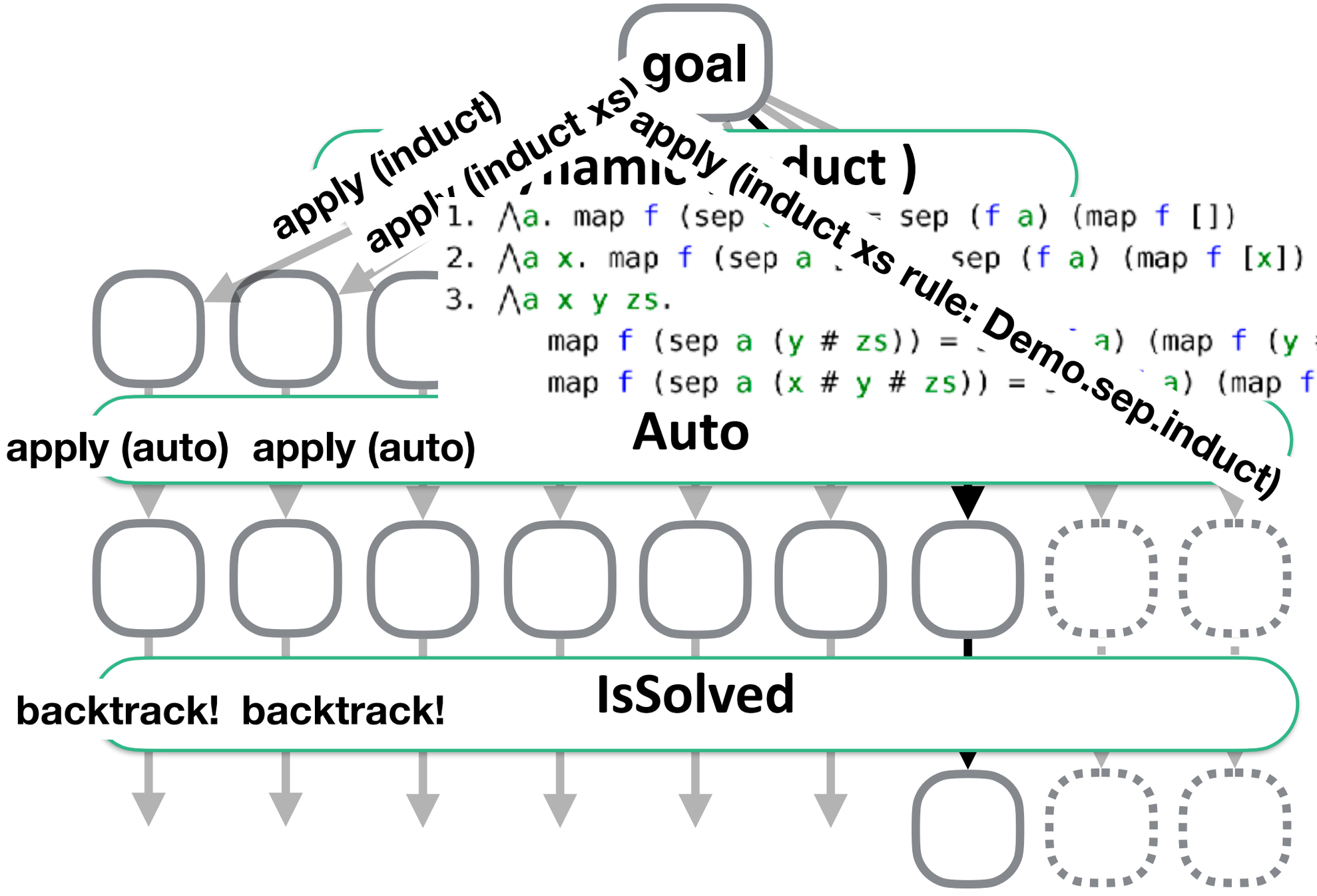
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

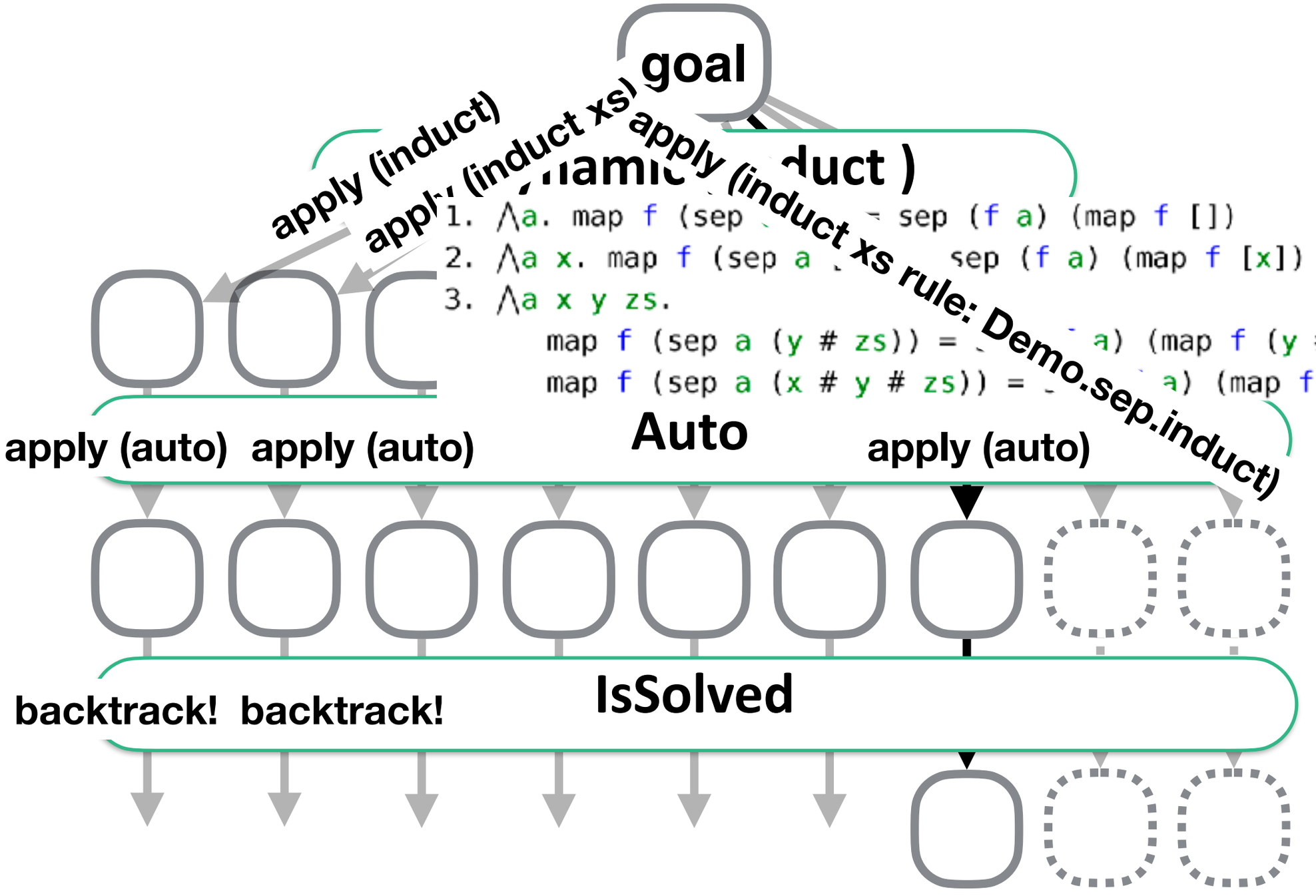
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

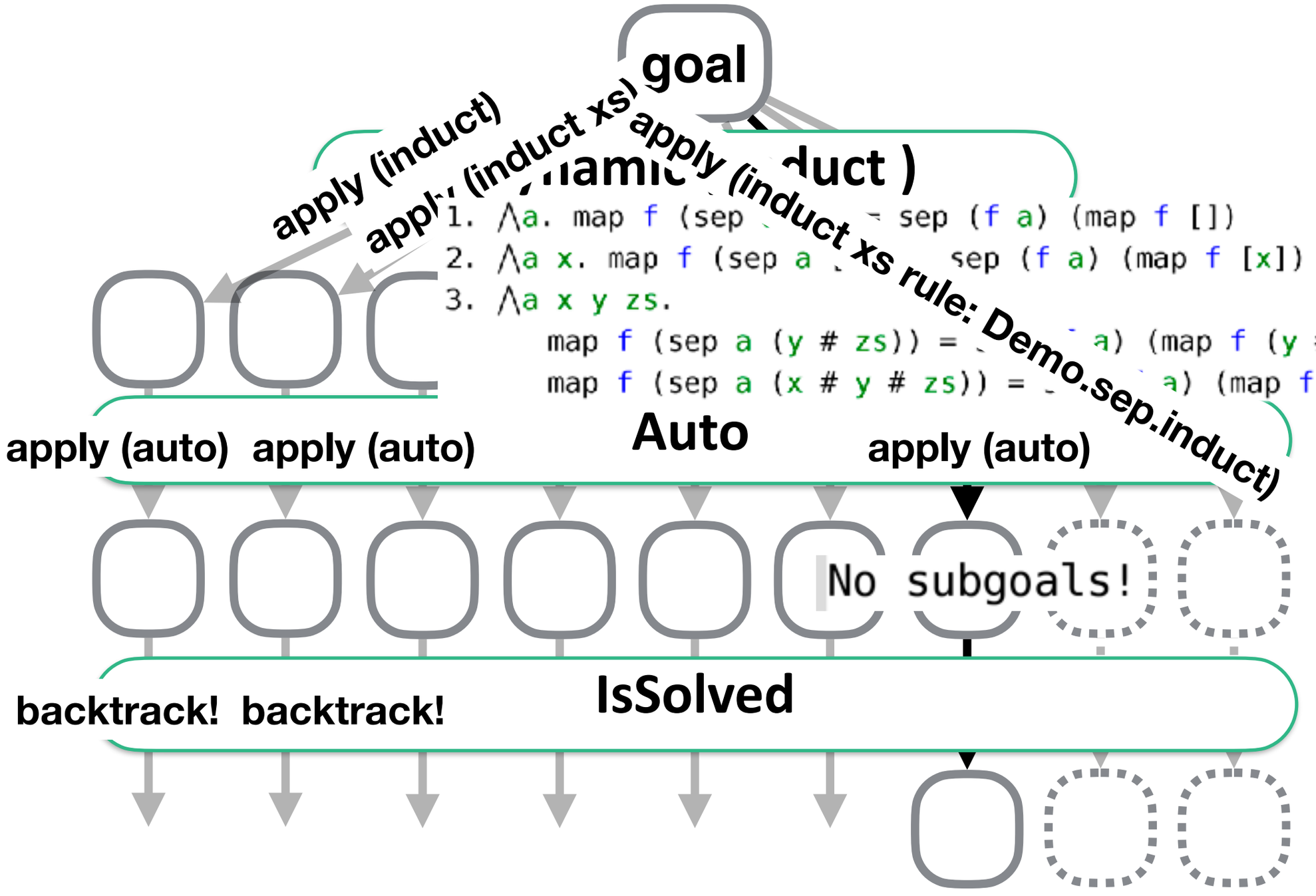
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

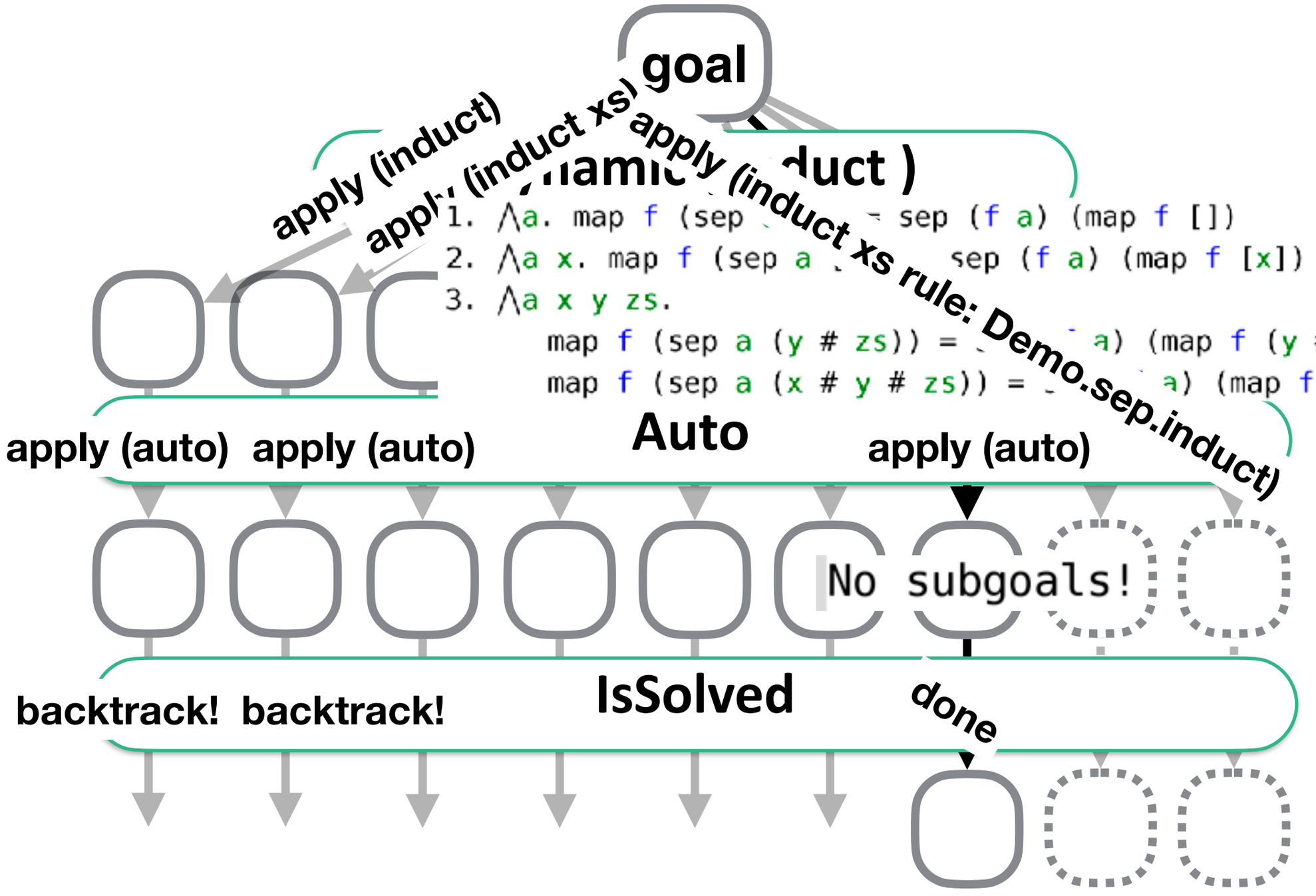
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

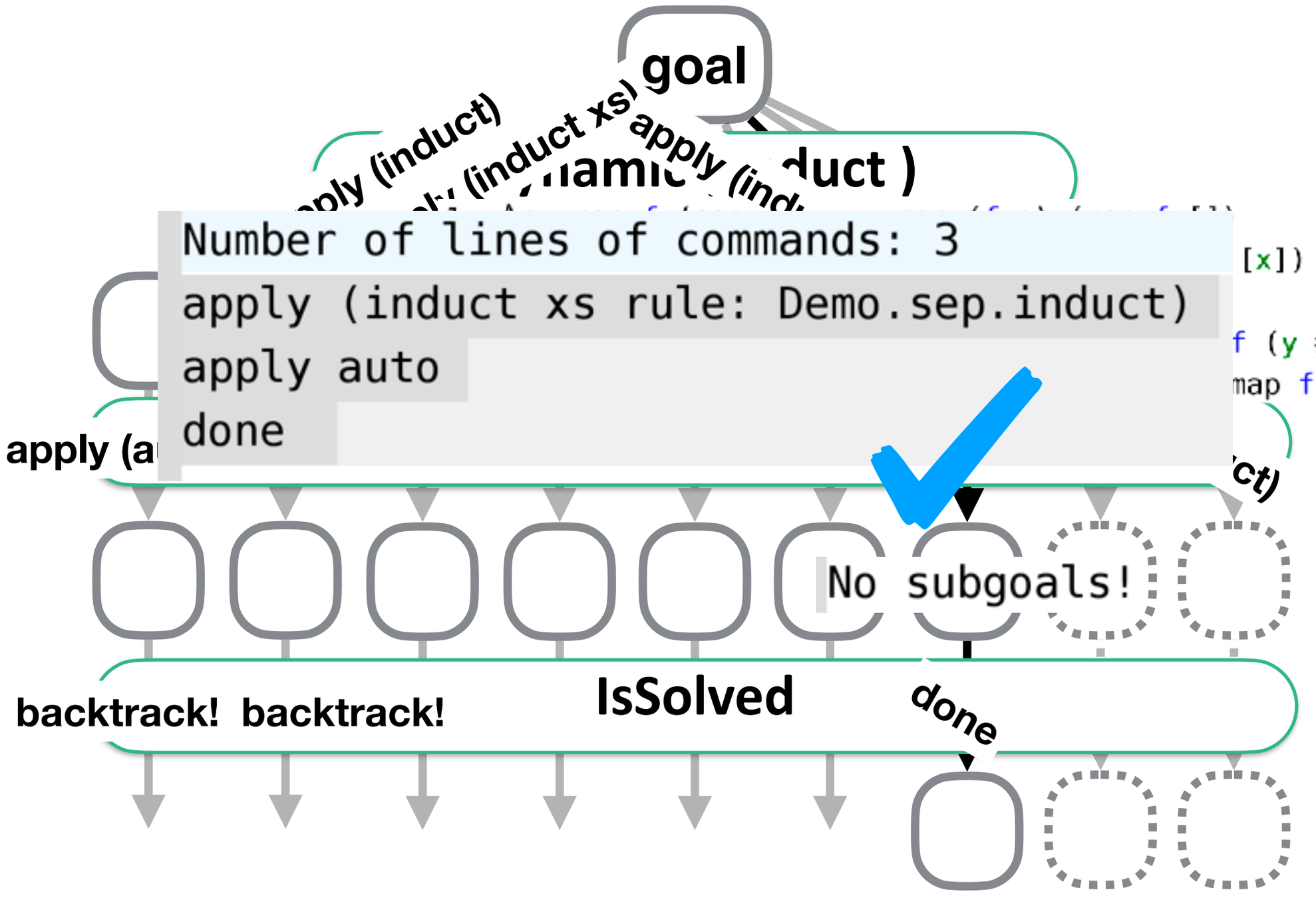
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

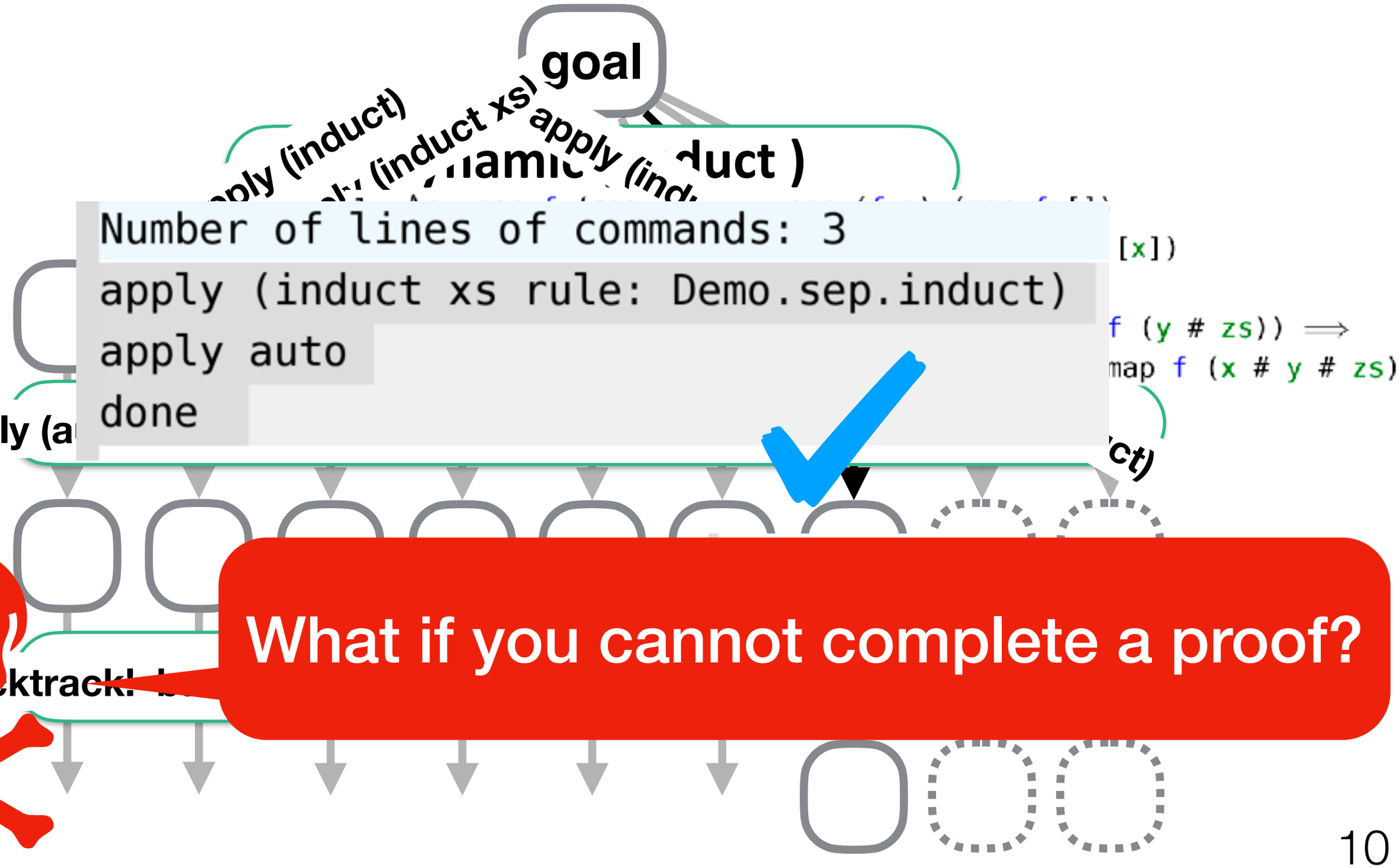
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

PSL: Proof Strategy Language

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



DEMO2: smart_induct

/master/slide/2020_Isabelle.pdf

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

DEMO2: smart_induct

/master/slide/2020_Isabelle.pdf

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

DEMO2: smart_induct

/master/slide/2020_Isabelle.pdf

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
```

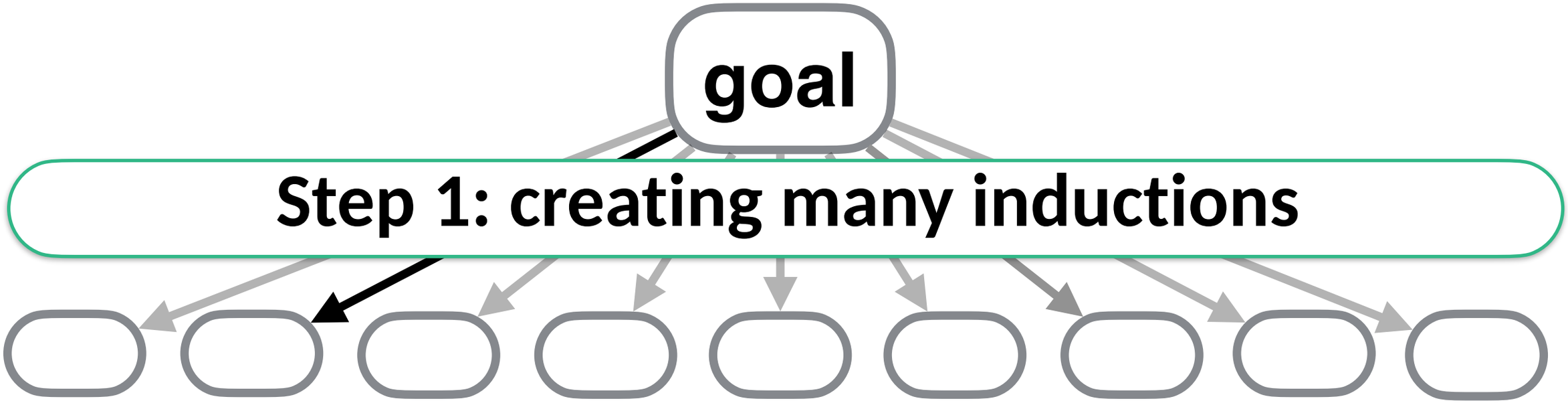
Proof state | Auto update | Update | Search: | 100%

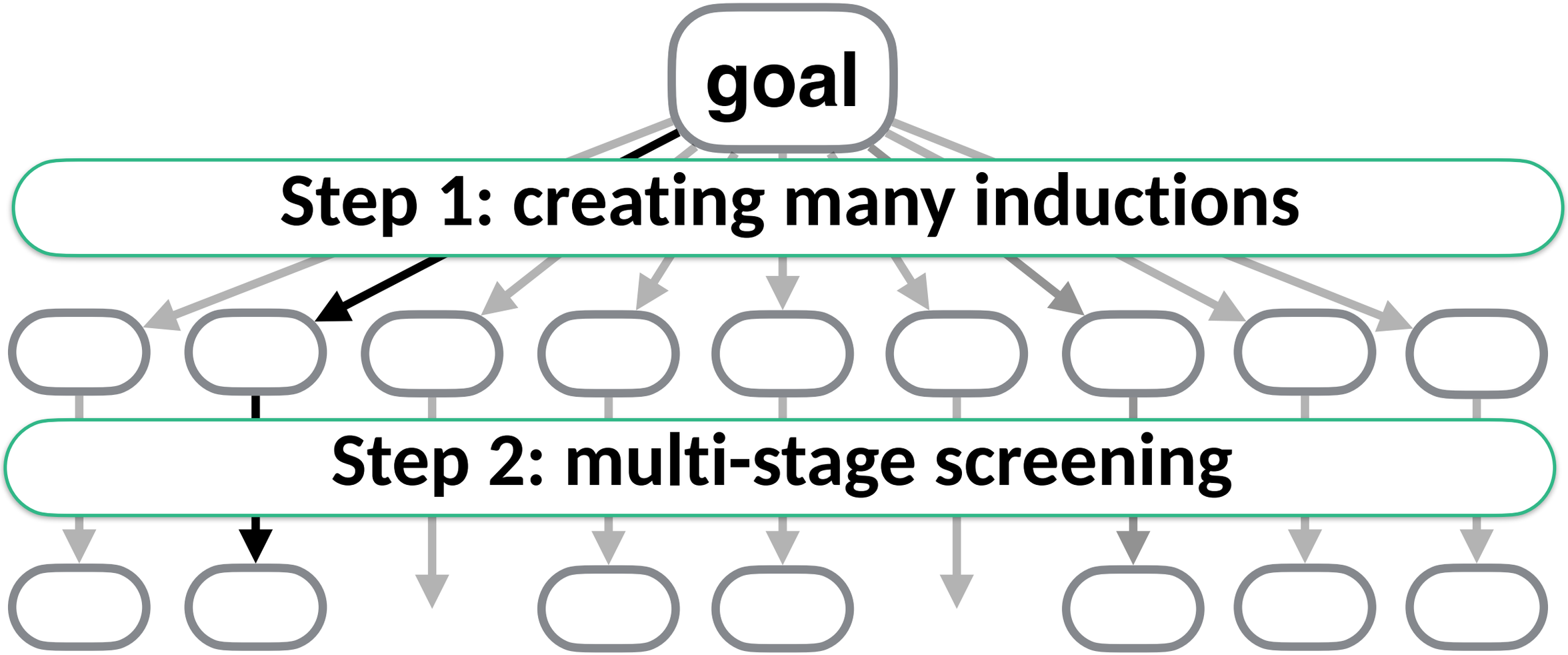
```
proof (prove)
goal:
No subgoals!
```

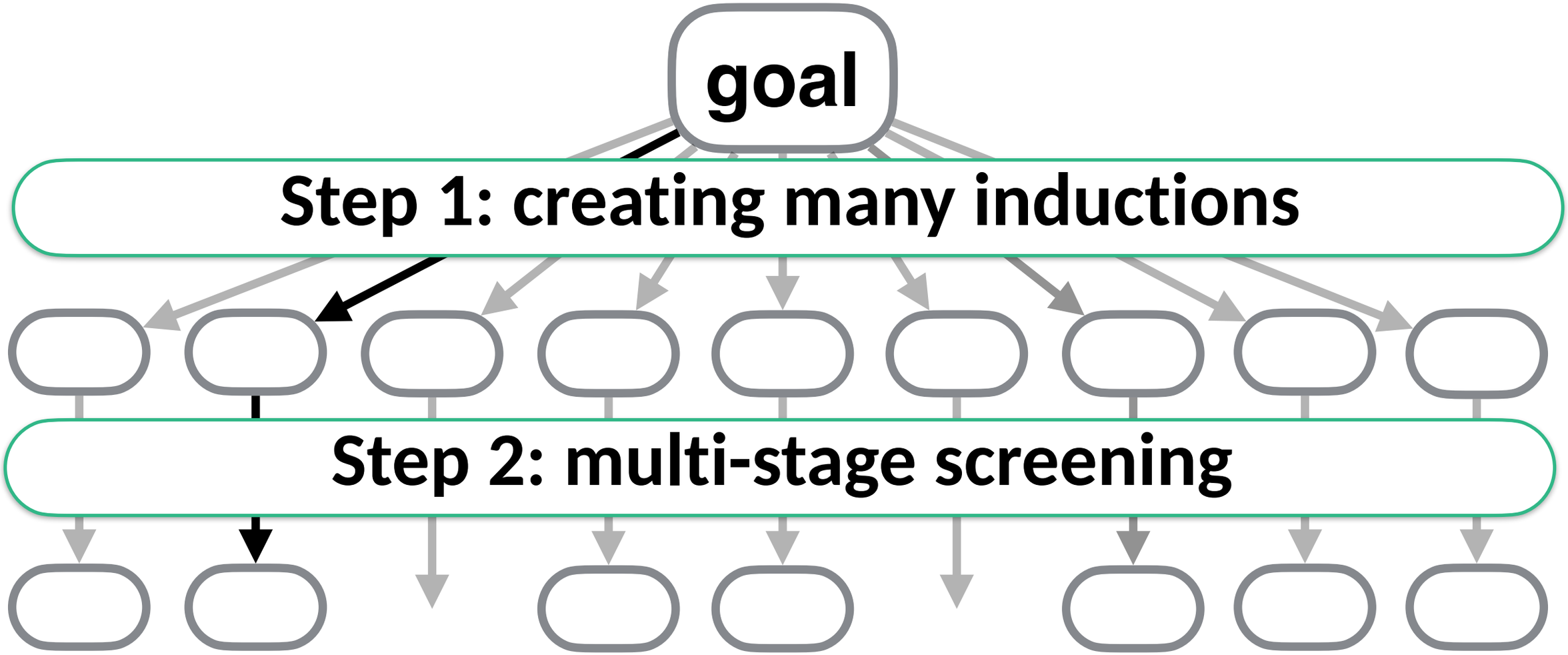
Output | Query | Sledgehammer | Symbols

19.45 (606/891)

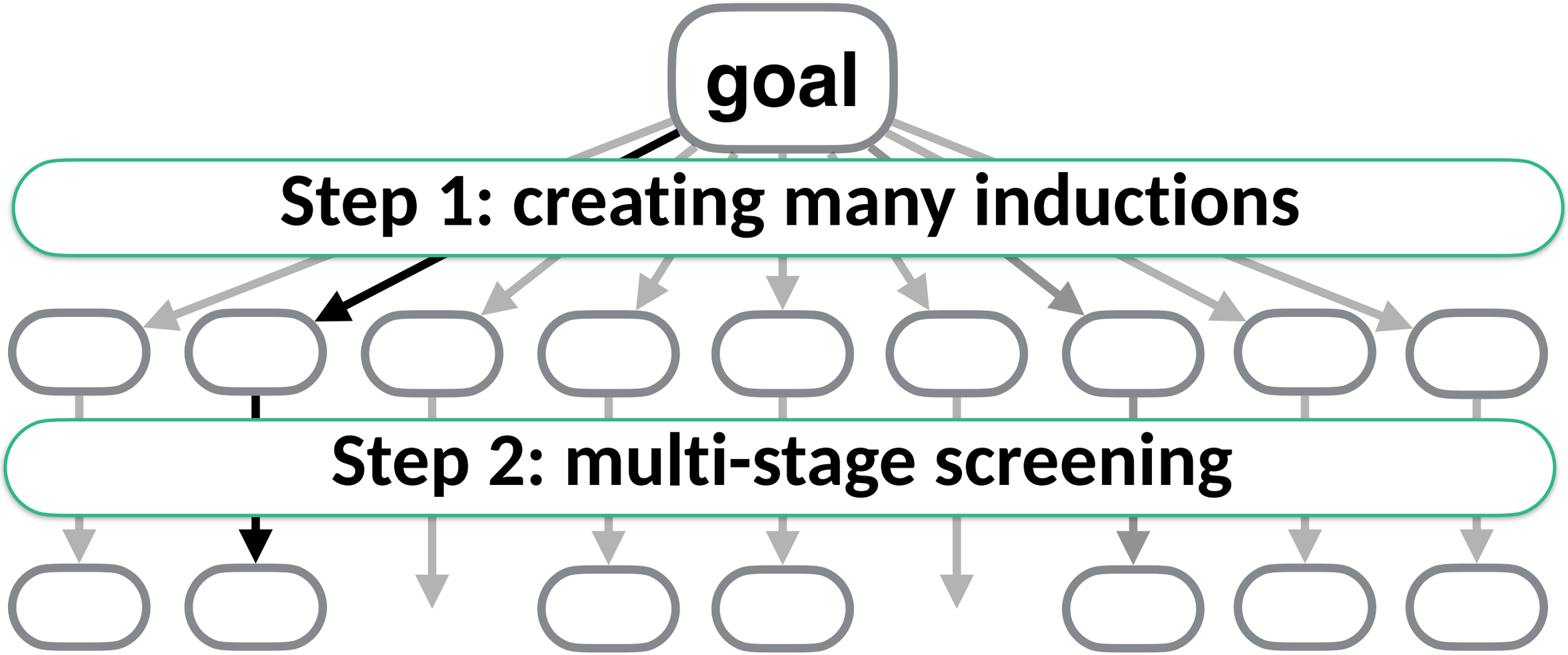
goal


goal
Step 1: creating many inductions


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



Step 3: scoring using 20 heuristics and sorting



Step 3: scoring using 20 heuristics and sorting

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

goal

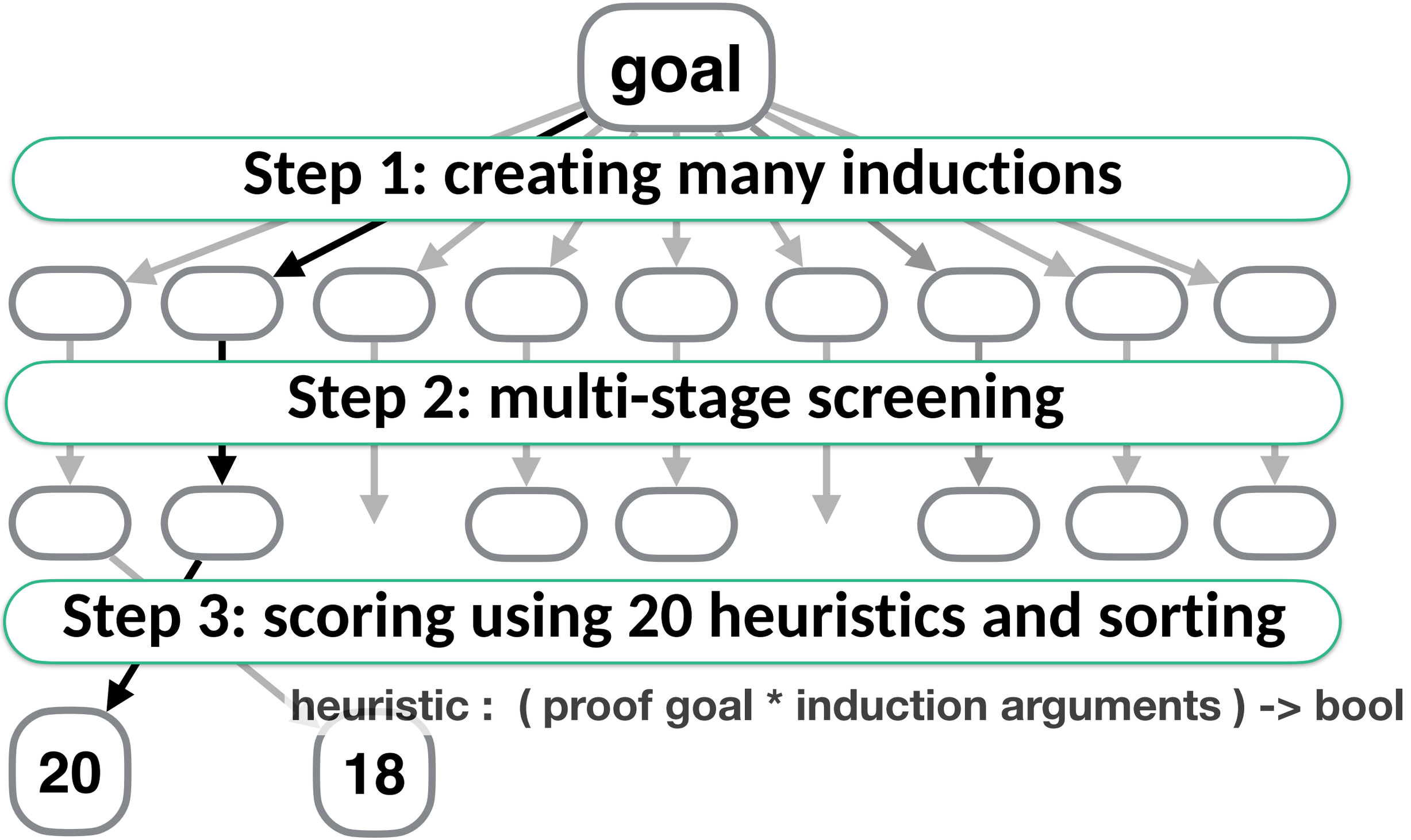
Step 1: creating many inductions

Step 2: multi-stage screening

Step 3: scoring using 20 heuristics and sorting

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

18


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
18

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

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

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

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

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

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

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

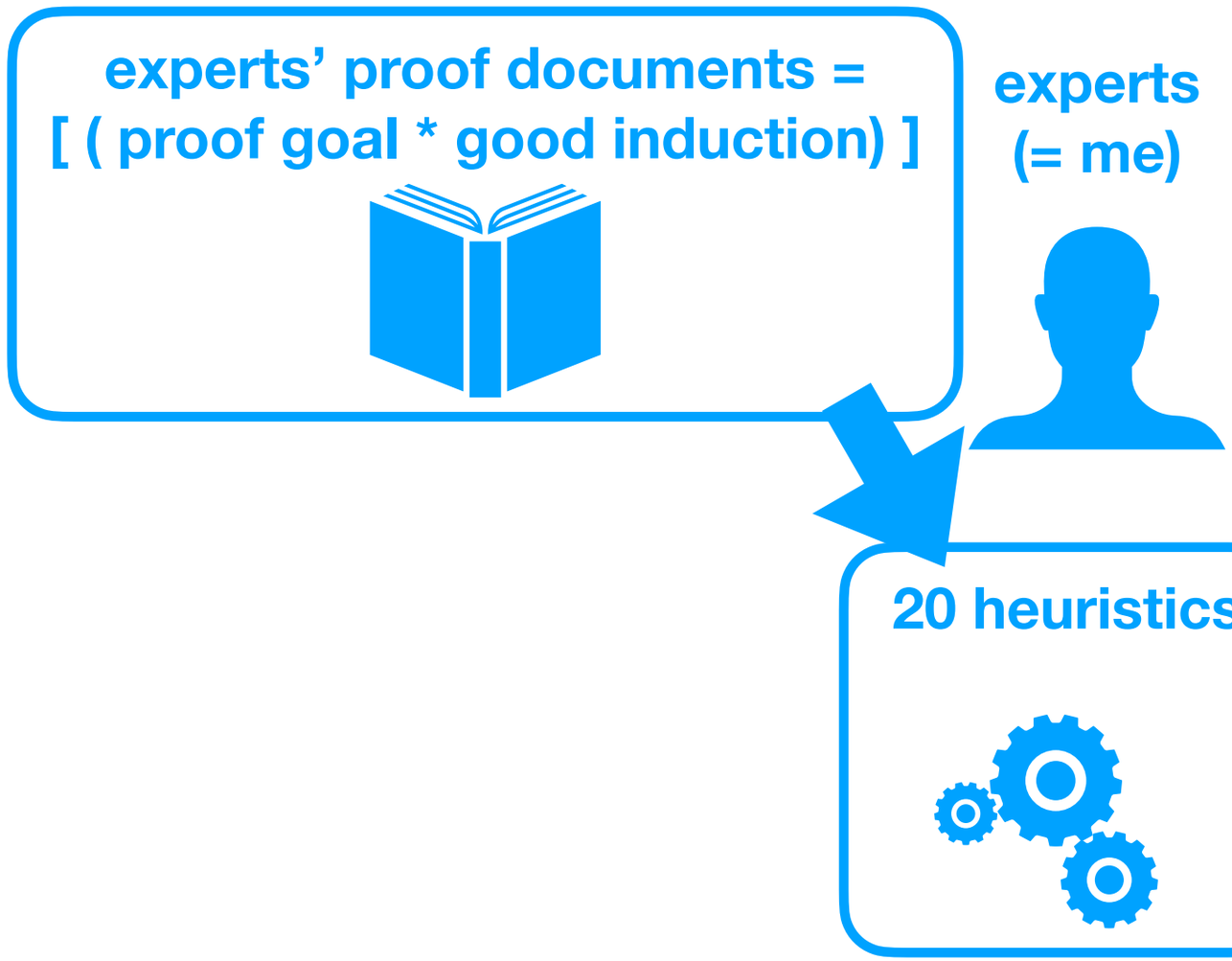
20 20 18 18 18 18 17

Step 4: short-listing

So, what heuristics do you use?

It is difficult to write induction heuristics.

It is difficult to write induction heuristics.



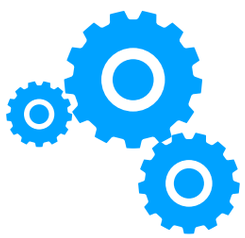
It is difficult to write induction heuristics.

```
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



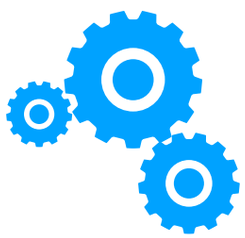
It is difficult to write induction heuristics.

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

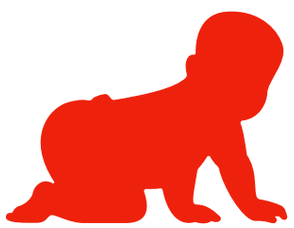
experts' proof documents =
[(proof goal * good induction)]

experts
(= me)

20 heuristics



new users



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

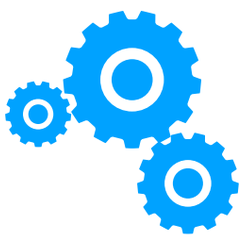
It is difficult to write induction heuristics.

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

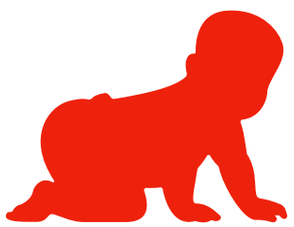
experts' proof documents =
[(proof goal * good induction)]

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!

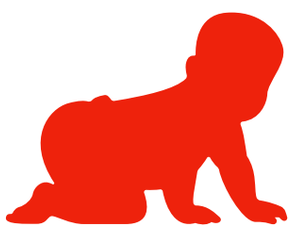
It is difficult to write induction heuristics.

```
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

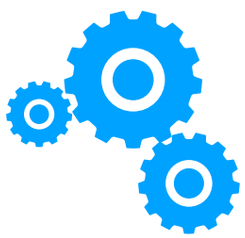


proof goal

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.

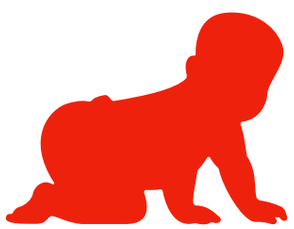
① blue = what I can see before releasing smart_induct

```
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

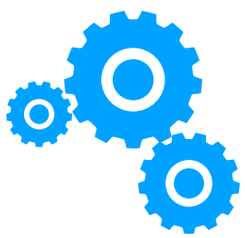


proof goal

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.

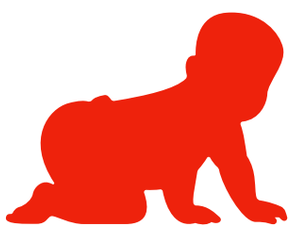
① blue = what I can see before releasing smart_induct

```
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

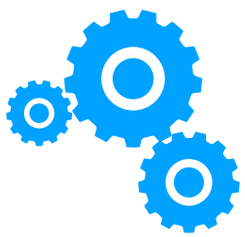


proof goal

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.

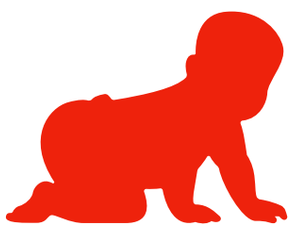
① blue = what I can see before releasing smart_induct

```
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

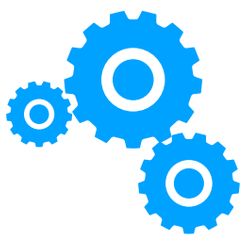


proof goal

candidate induction

apply (
induct is1 s stk
rule: exec.induct)

20 heuristics



[✓ ,
✗ ,
✓ ,...]

```
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.

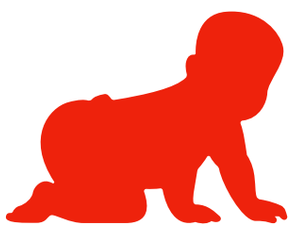
① blue = what I can see before releasing smart_induct

```
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

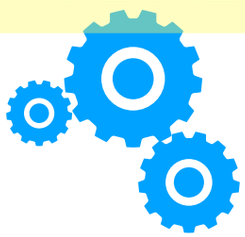


proof goal

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!

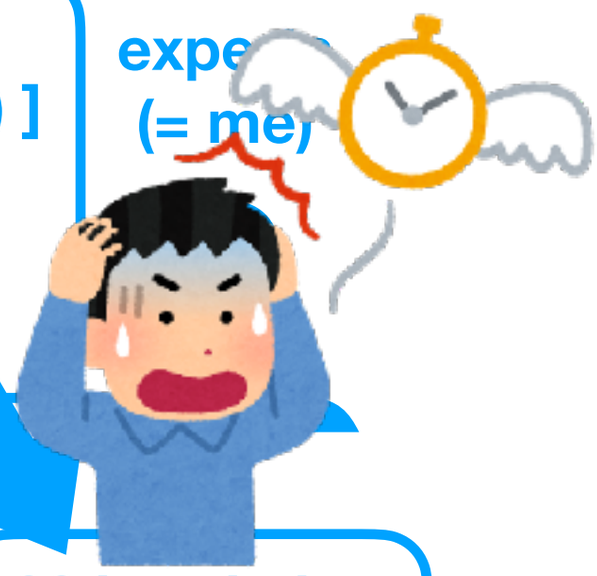
② red = what will be developed after releasing smart_induct

It is difficult to write induction heuristics.

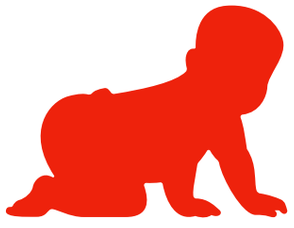
① blue = what I can see before releasing smart_induct

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

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



new users

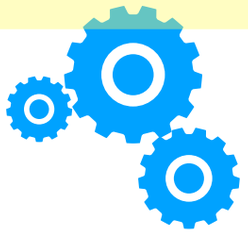


proof goal

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

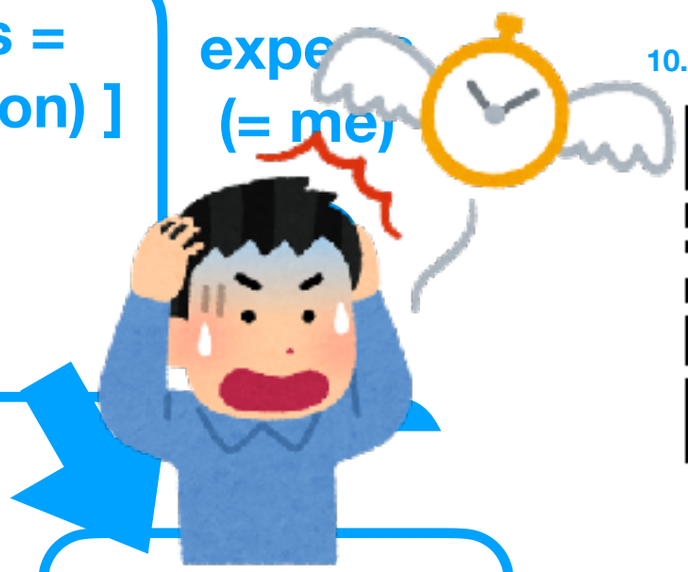
It is difficult to write induction heuristics.

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
```

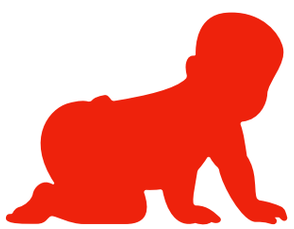
experts' proof documents =
[(proof goal * good induction)]

expe
(= me)



https://doi.org/
10.1007/978-3-030-34175-6_14

new users

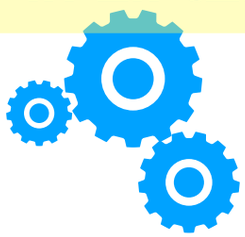


proof goal

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

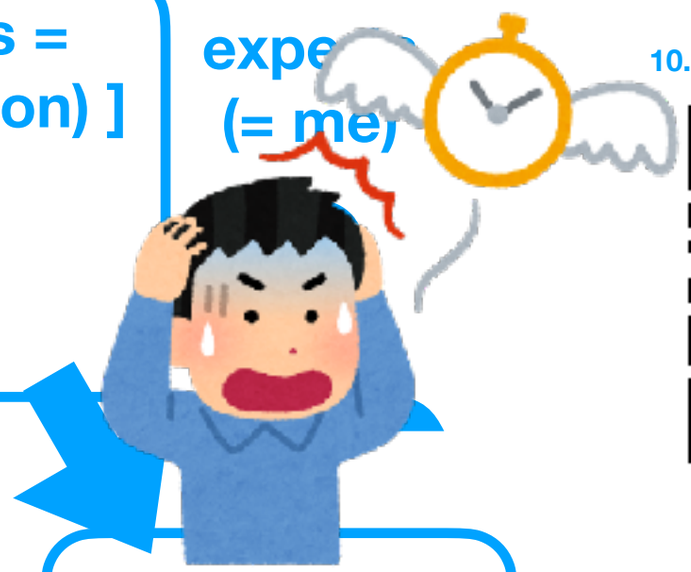
It is difficult to write induction heuristics.

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
```

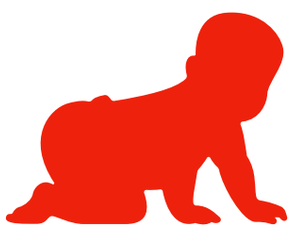
experts' proof documents =
[(proof goal * good induction)]

expe
(= me)



https://doi.org/
10.1007/978-3-030-34175-6_14

new users

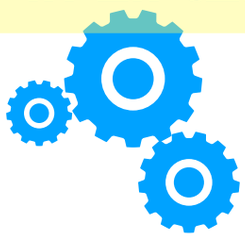


proof goal

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)"
```

② n ...types!

Does it (smart_induct) really work?

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
```

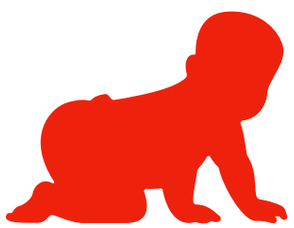
experts' proof documents =
[(proof goal * good induction)]

YES!

experts
(= me)

https://doi.org/
10.1007/978-3-030-34175-6_14

new users

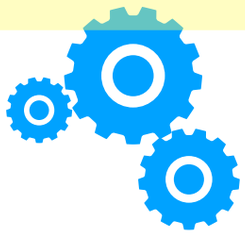


proof goal

candidate induction

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

20 heuristics
in LiFtEr

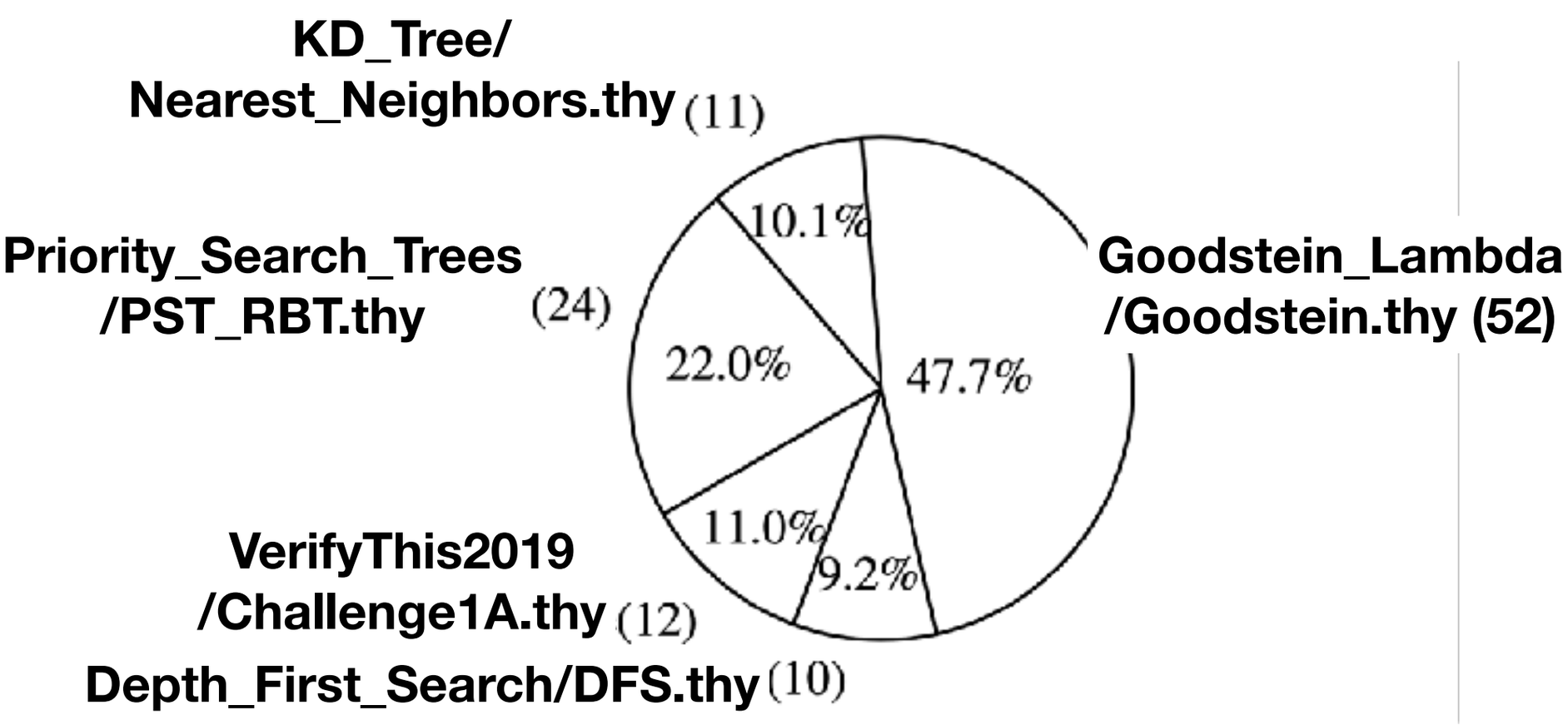


[✓ ,
✗ ,
✓ ,...]

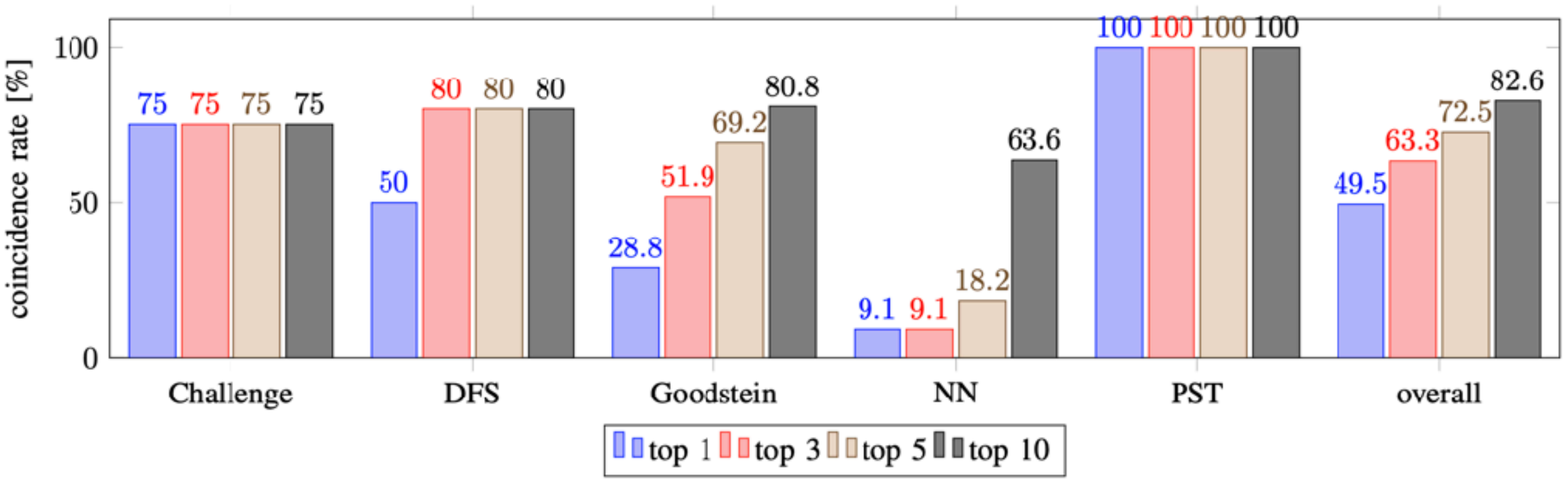
```
lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
```

② Does it (smart_induct) really work?

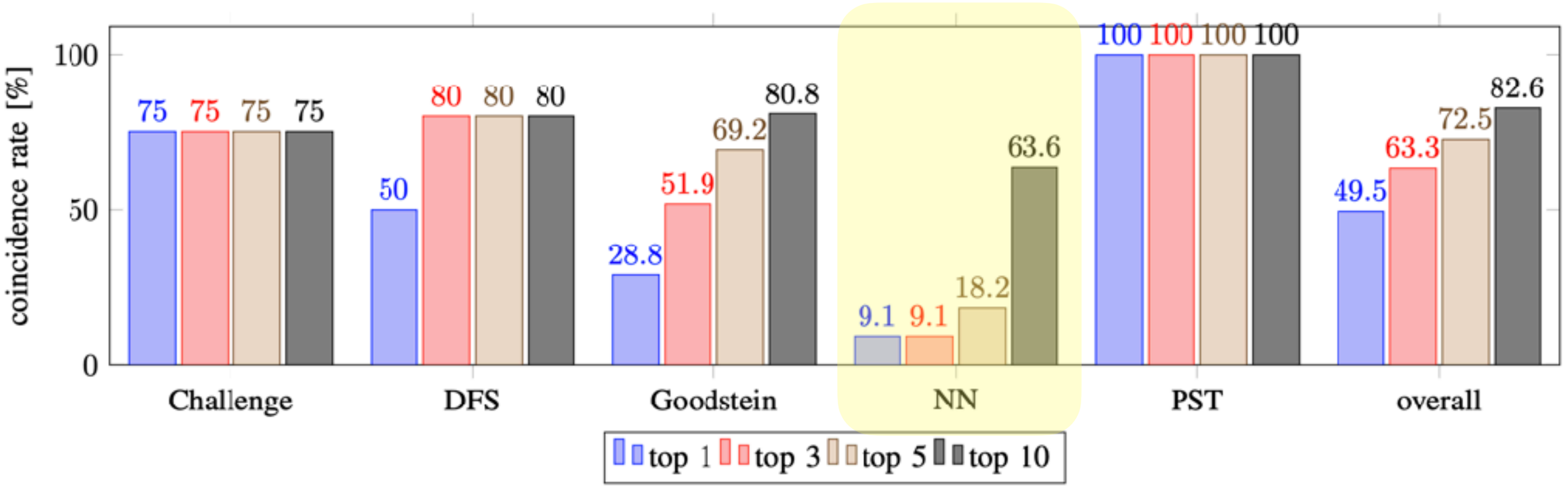
Dataset for evaluation
(109 proofs by induction from the AFP entries)



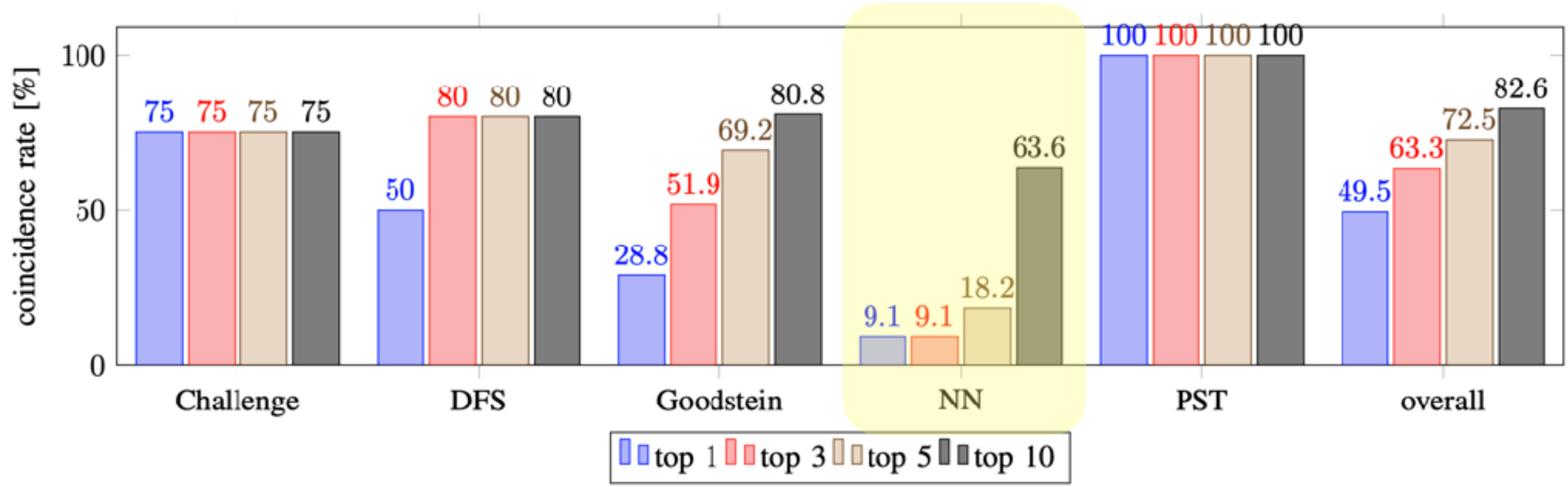
Evaluation results in terms of theory files.



Evaluation results in terms of theory files.

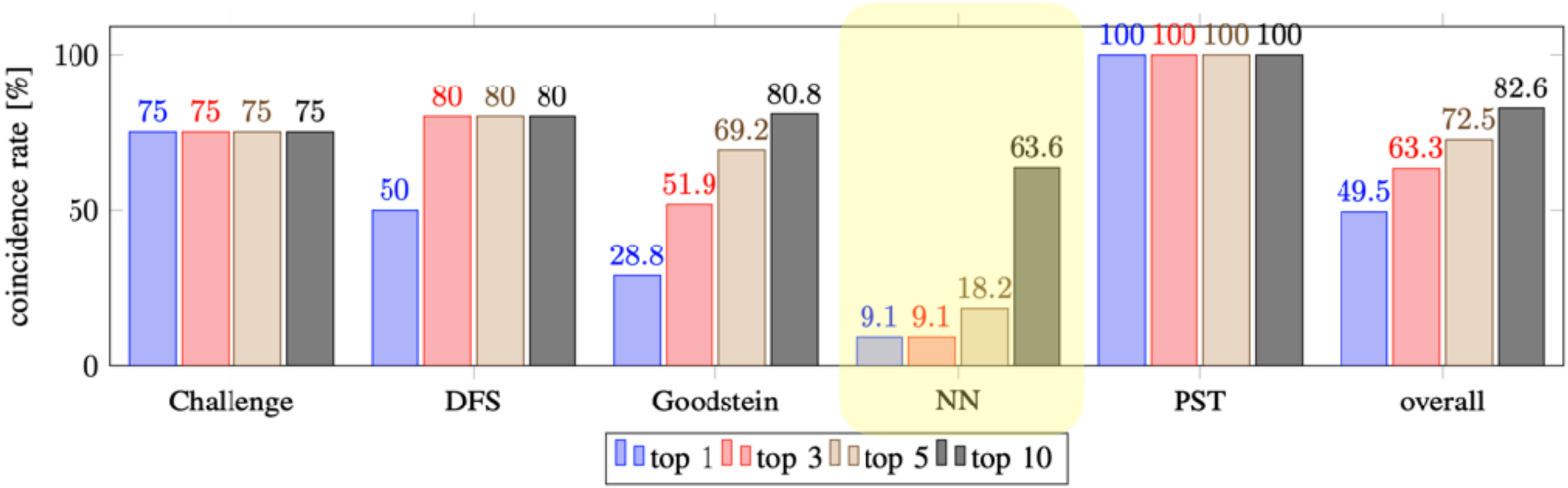


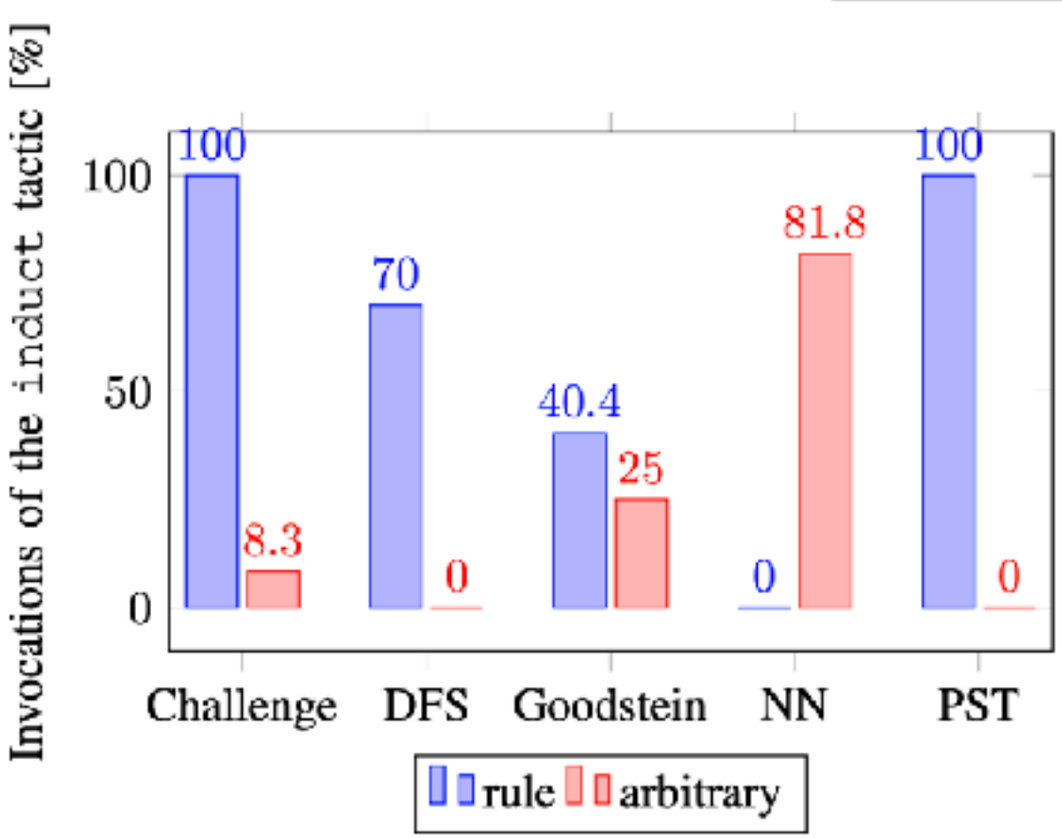
Evaluation results in terms of theory files.



Doesn’t the performance depend heavily on problem domains?

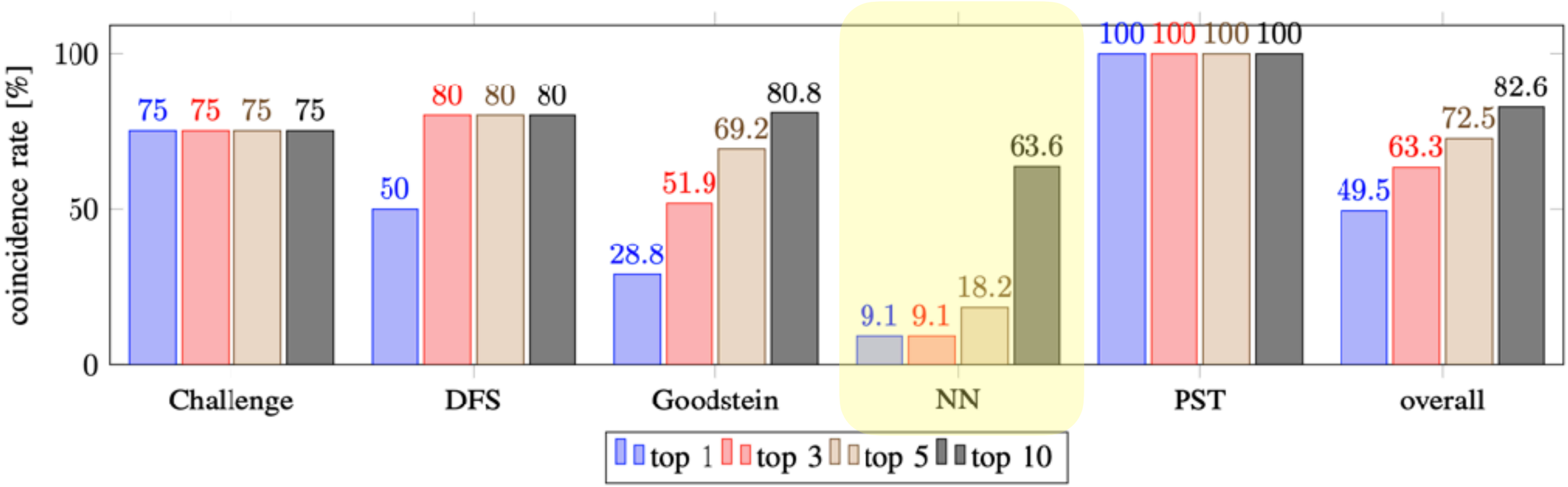
Evaluation results in terms of theory files.

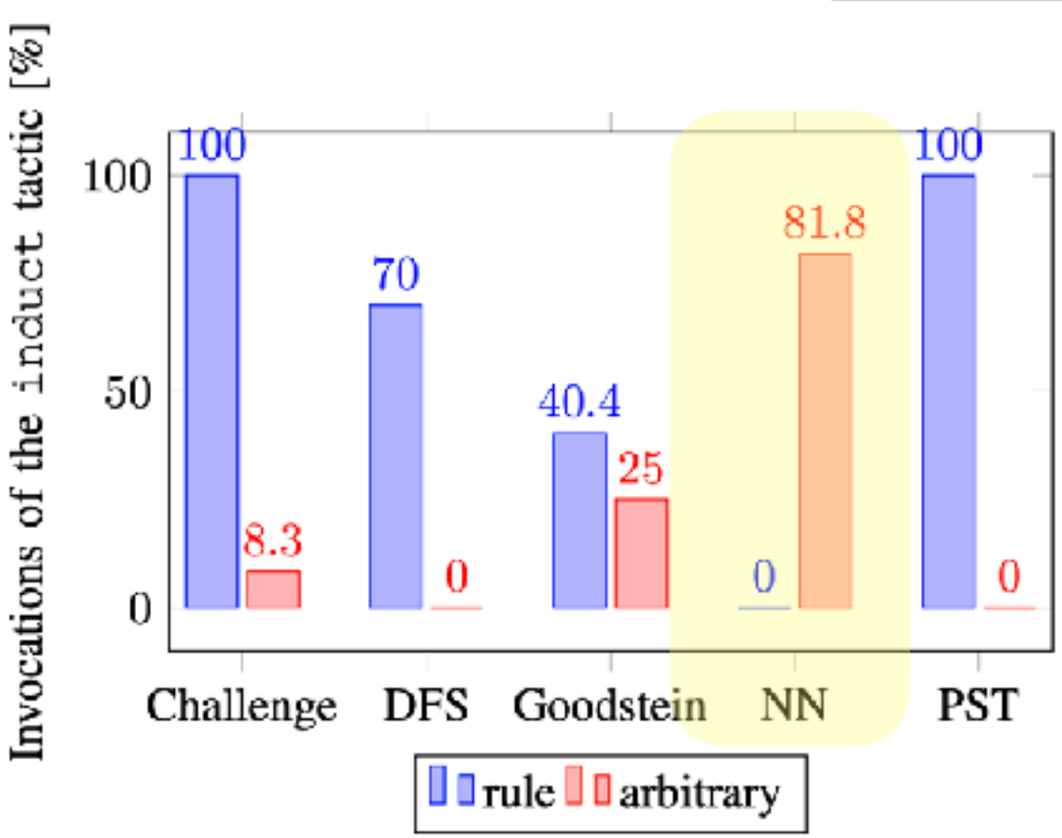




Doesn’t the performance depend heavily on problem domains?

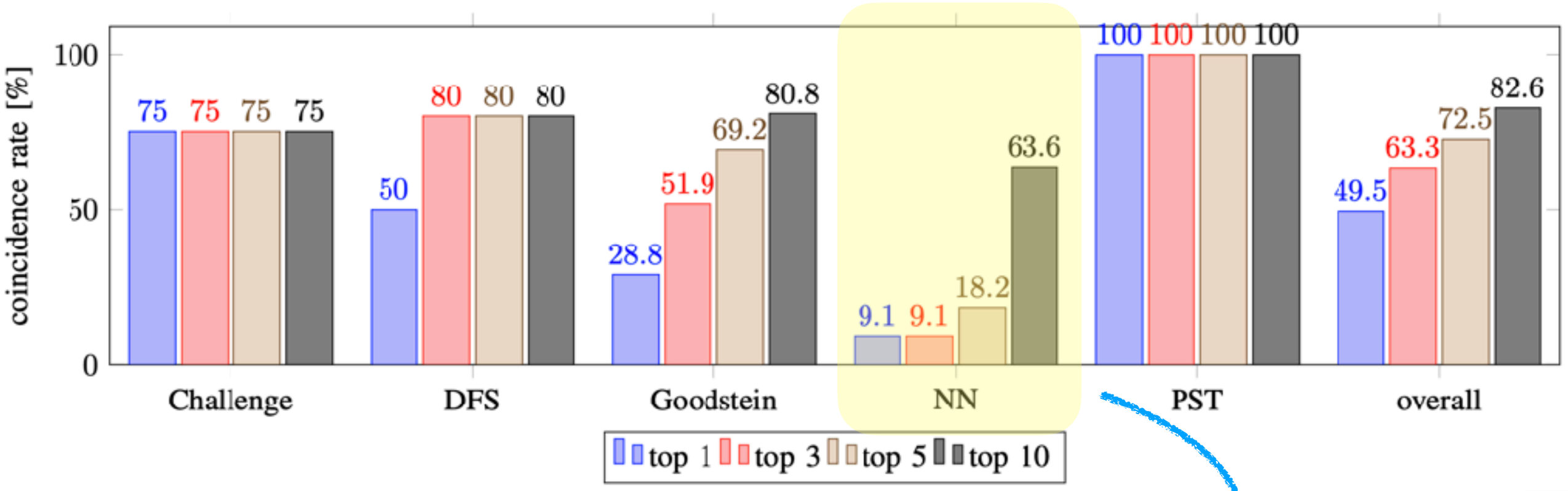
Evaluation results in terms of theory files.



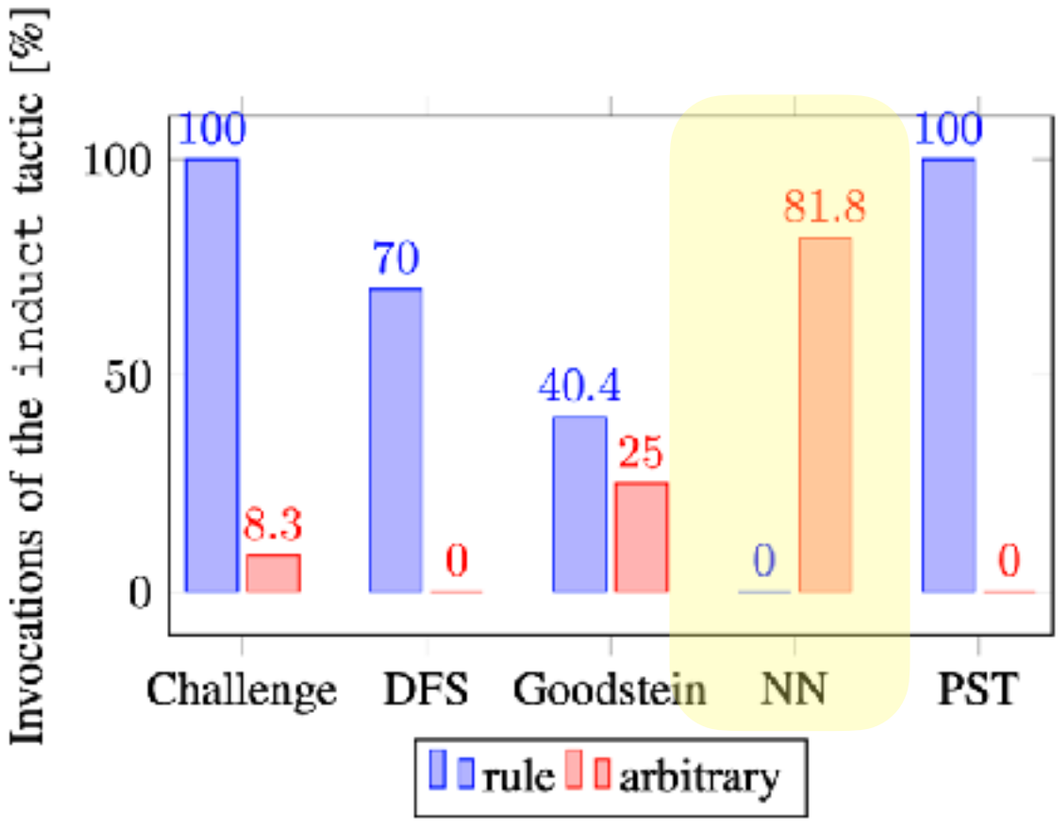


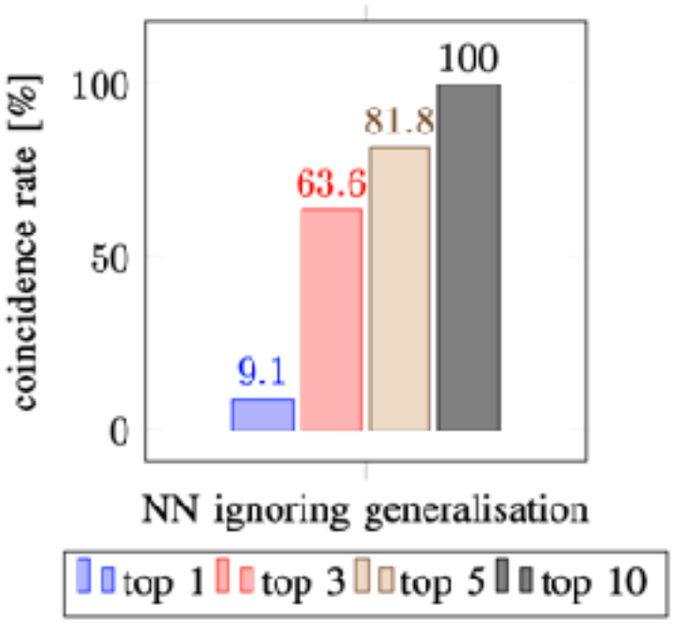
Doesn’t the performance depend heavily on problem domains?

Evaluation results in terms of theory files.

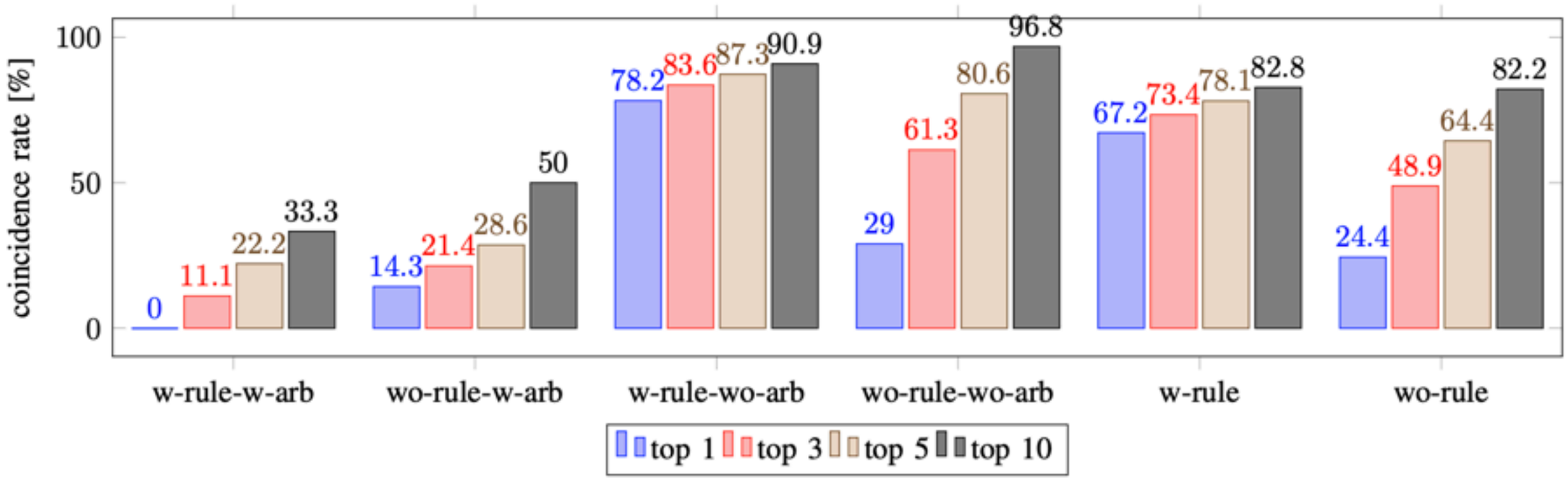


Doesn't the performance depend heavily on problem domains?

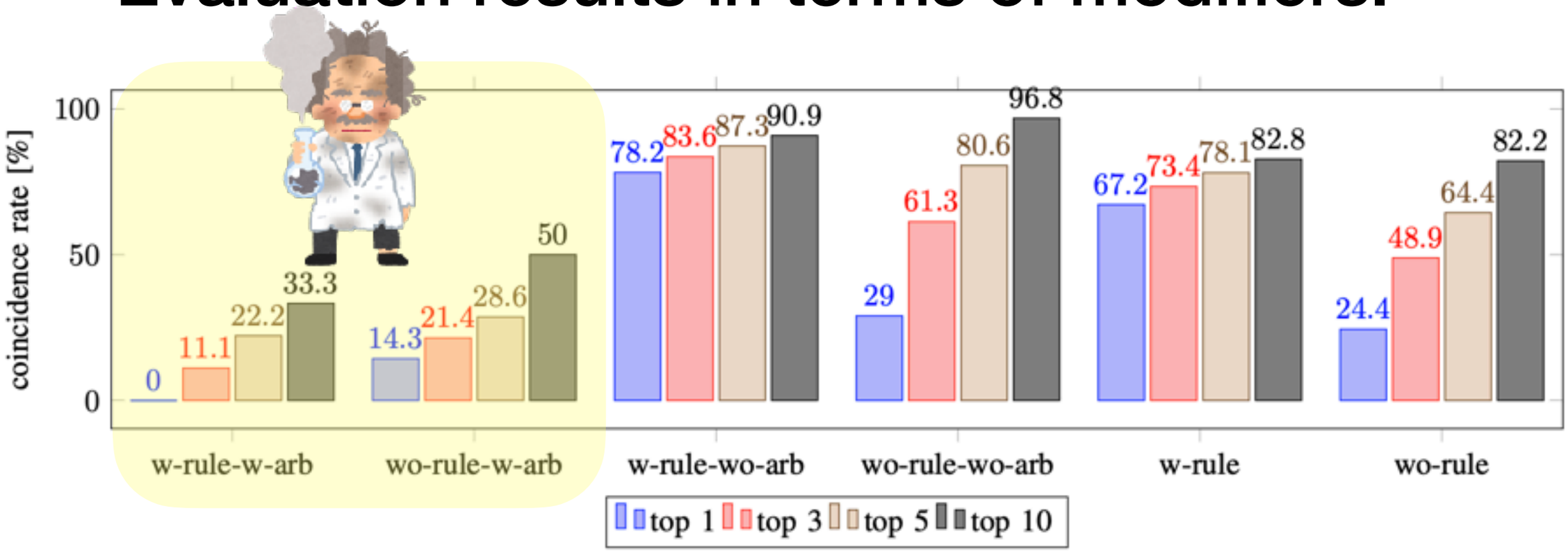




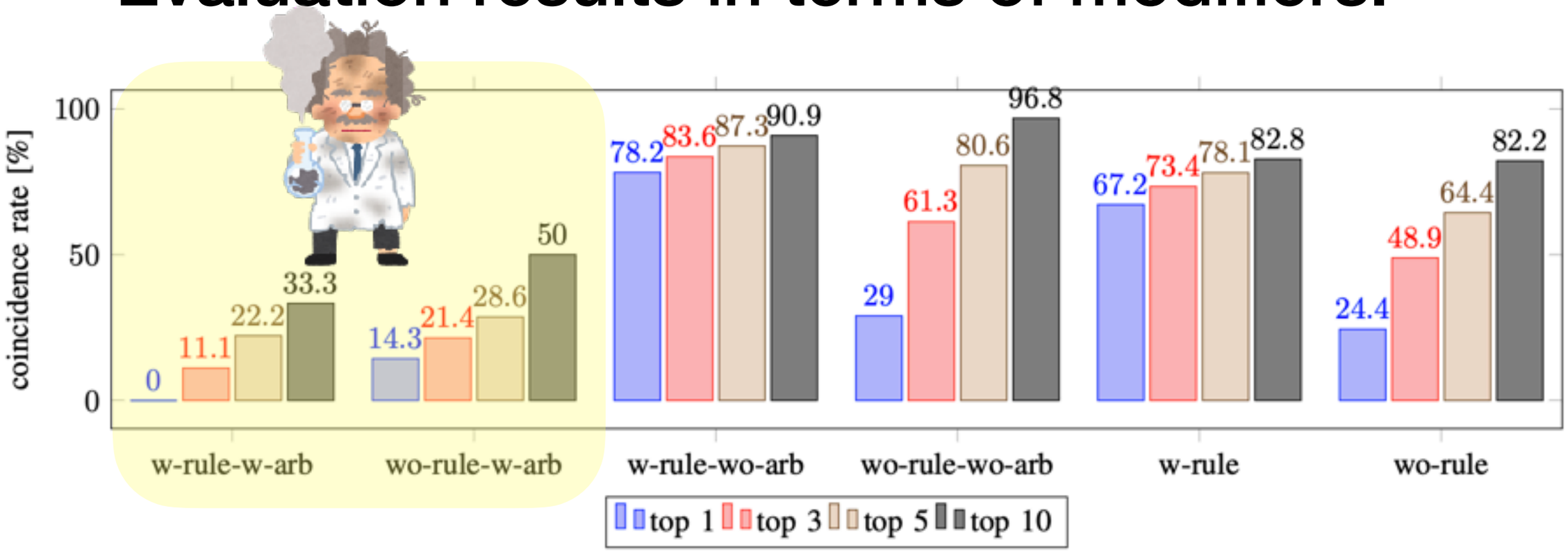
Evaluation results in terms of modifiers.



Evaluation results in terms of modifiers.

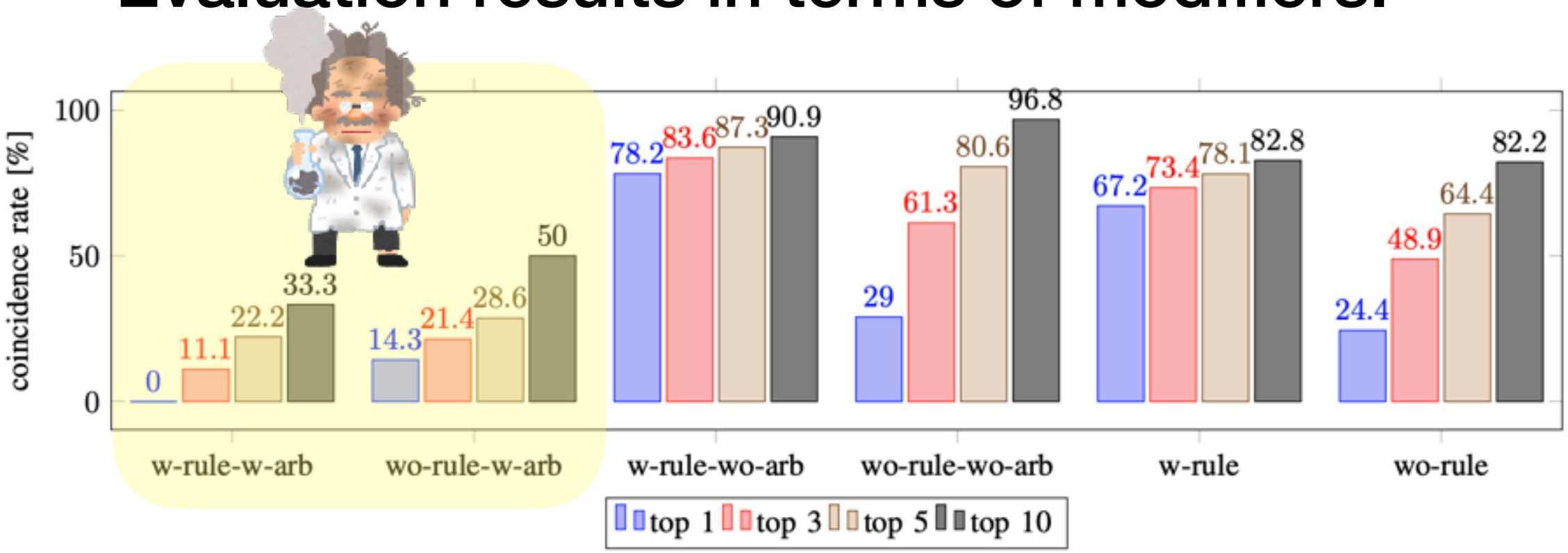


Evaluation results in terms of modifiers.



Are you going to address the problem of generalization?

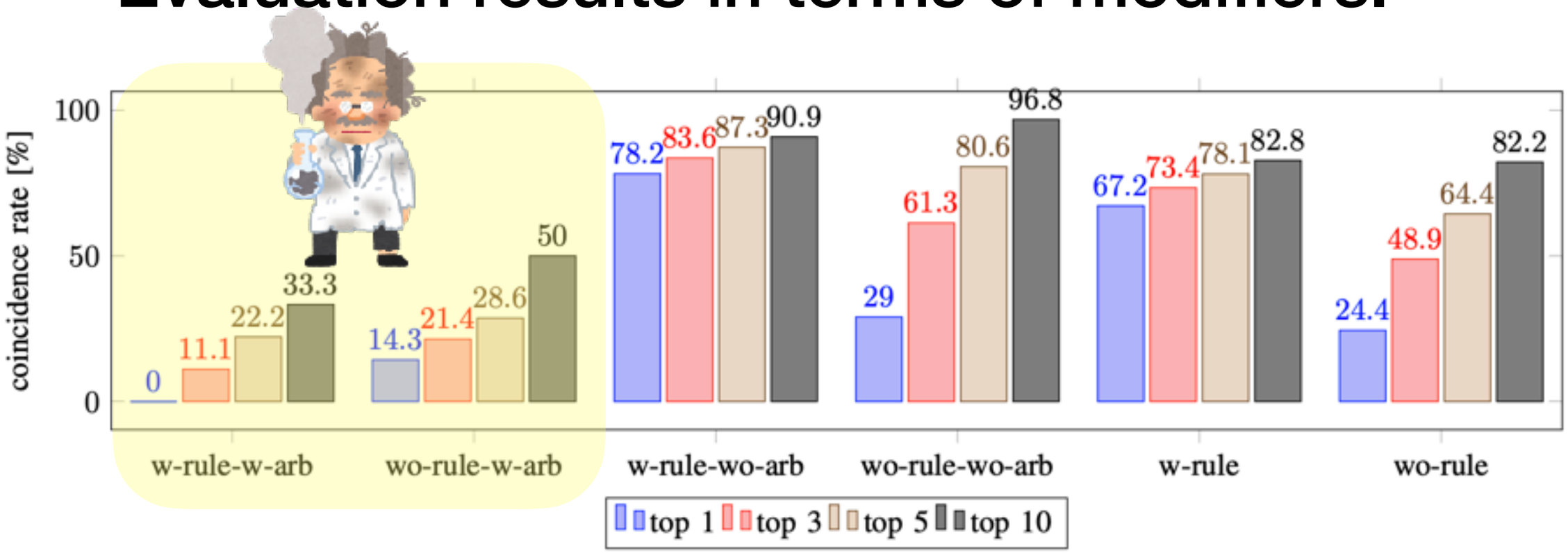
Evaluation results in terms of modifiers.



Are you going to address the problem of generalization?

Yes. smart_induct2 is coming soon with LiFtEr2.

Evaluation results in terms of modifiers.



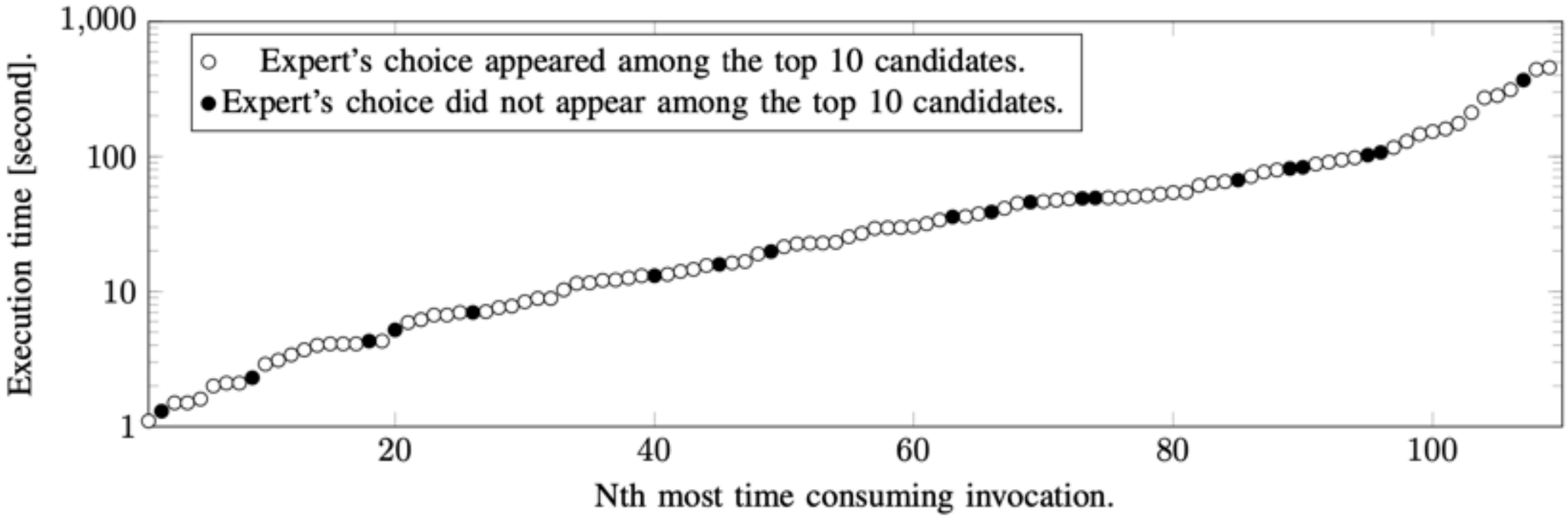
Are you going to address the problem of generalization?

Yes. smart_induct2 is coming soon with LiFtEr2.

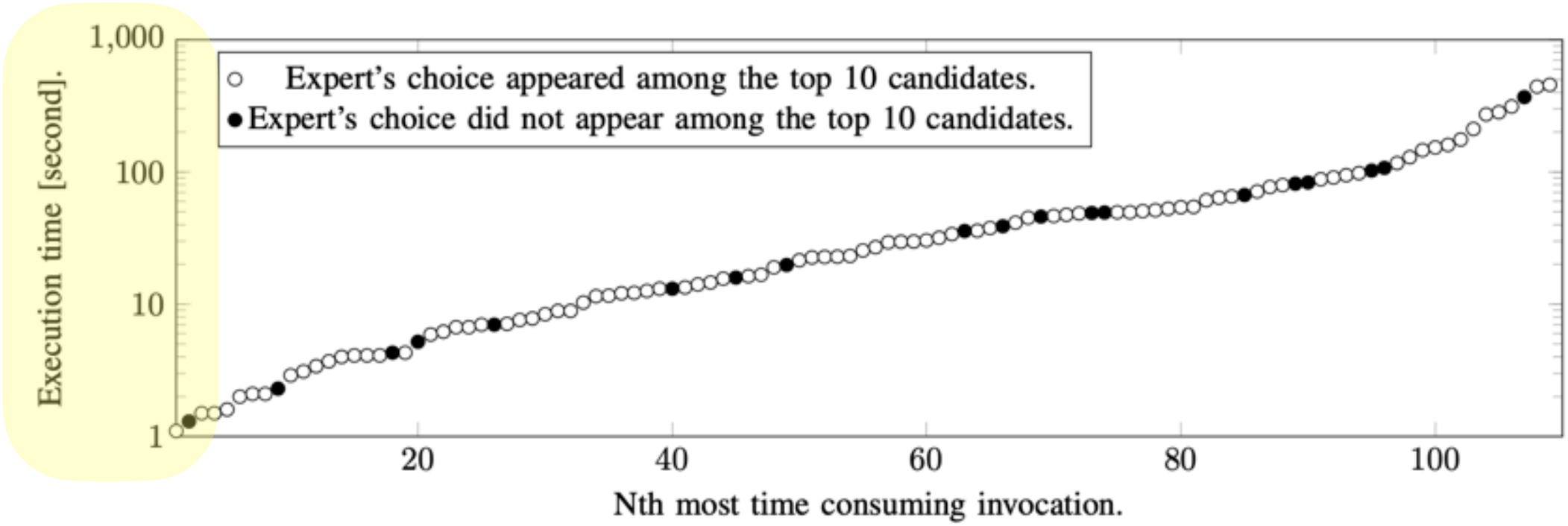
How fast does smart_induct work?

Execution time of smart_induct

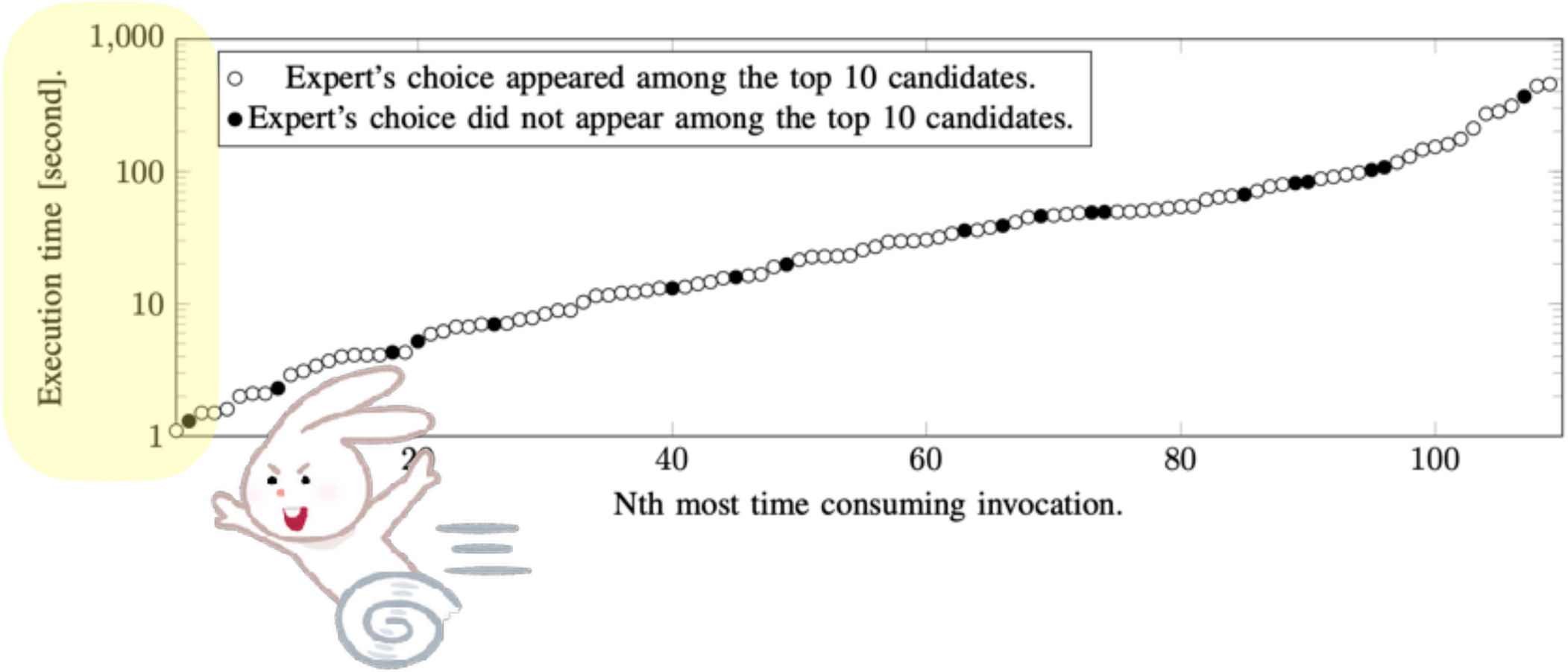
Execution time of smart_induct



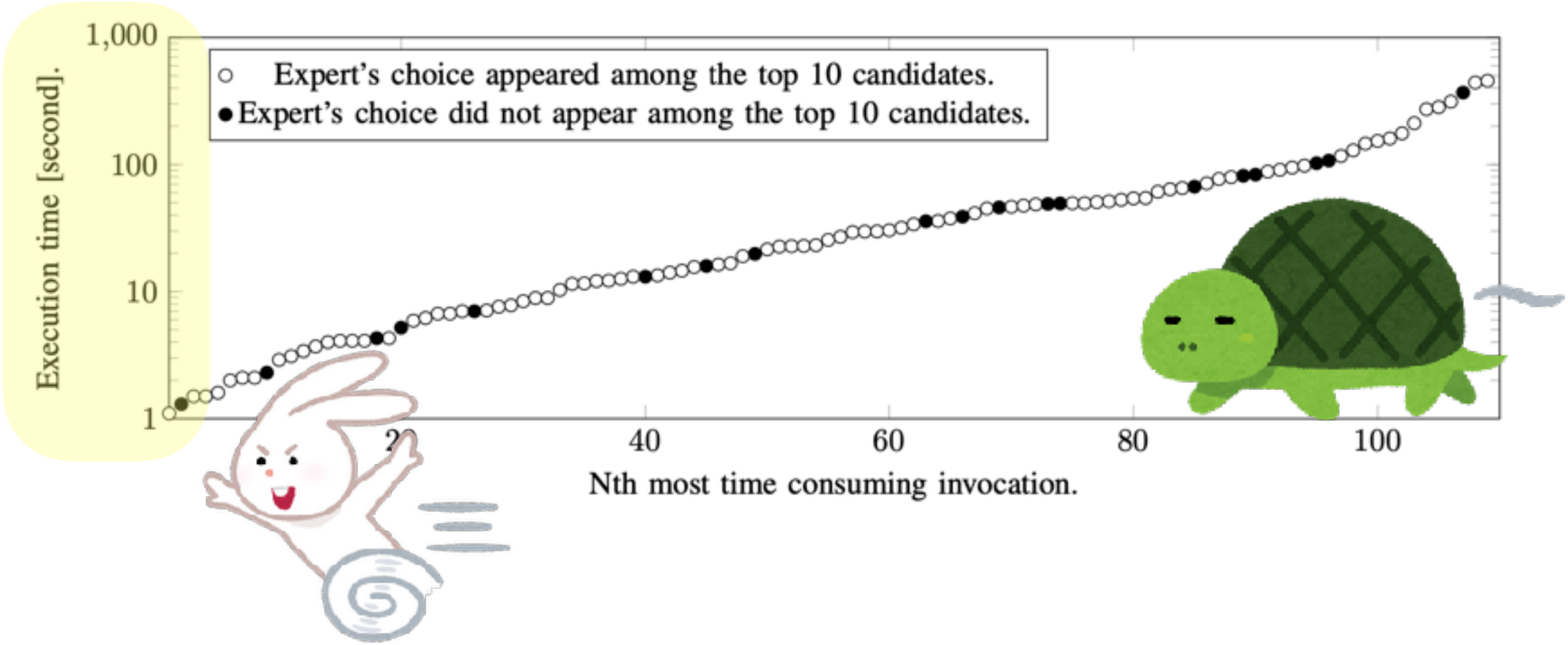
Execution time of smart_induct



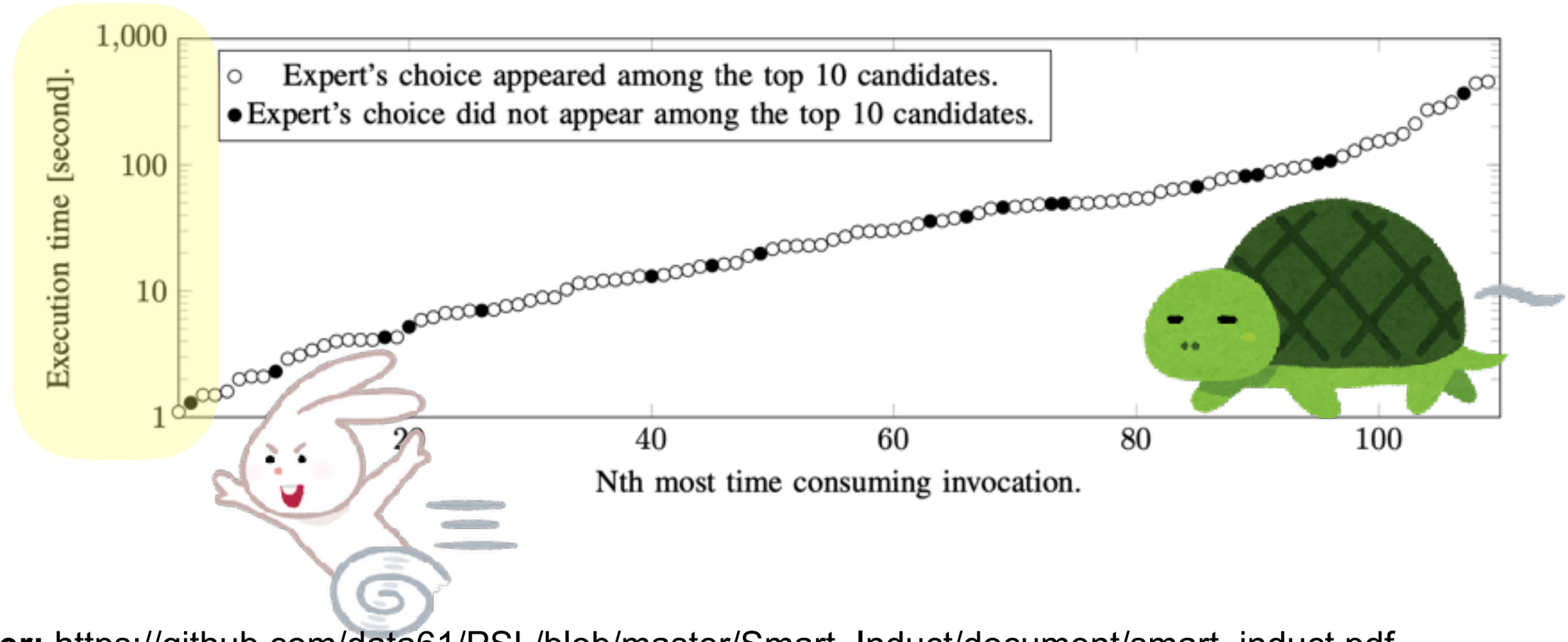
Execution time of smart_induct



Execution time of smart_induct



Execution time of smart_induct

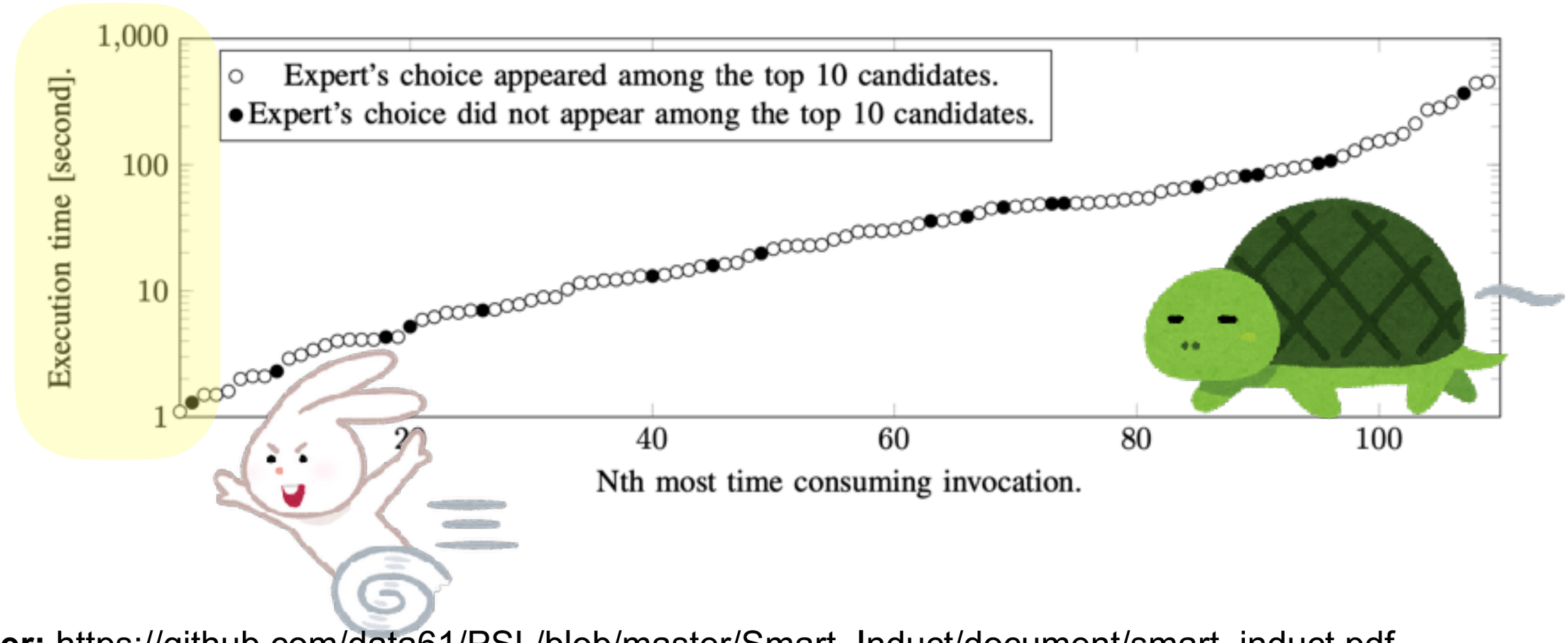


paper: https://github.com/data61/PSL/blob/master/Smart_Induct/document/smart_induct.pdf

smart_induct: https://github.com/data61/PSL/releases/tag/0.1.7-alpha

Q&A!

Execution time of smart_induct



paper: https://github.com/data61/PSL/blob/master/Smart_Induct/document/smart_induct.pdf

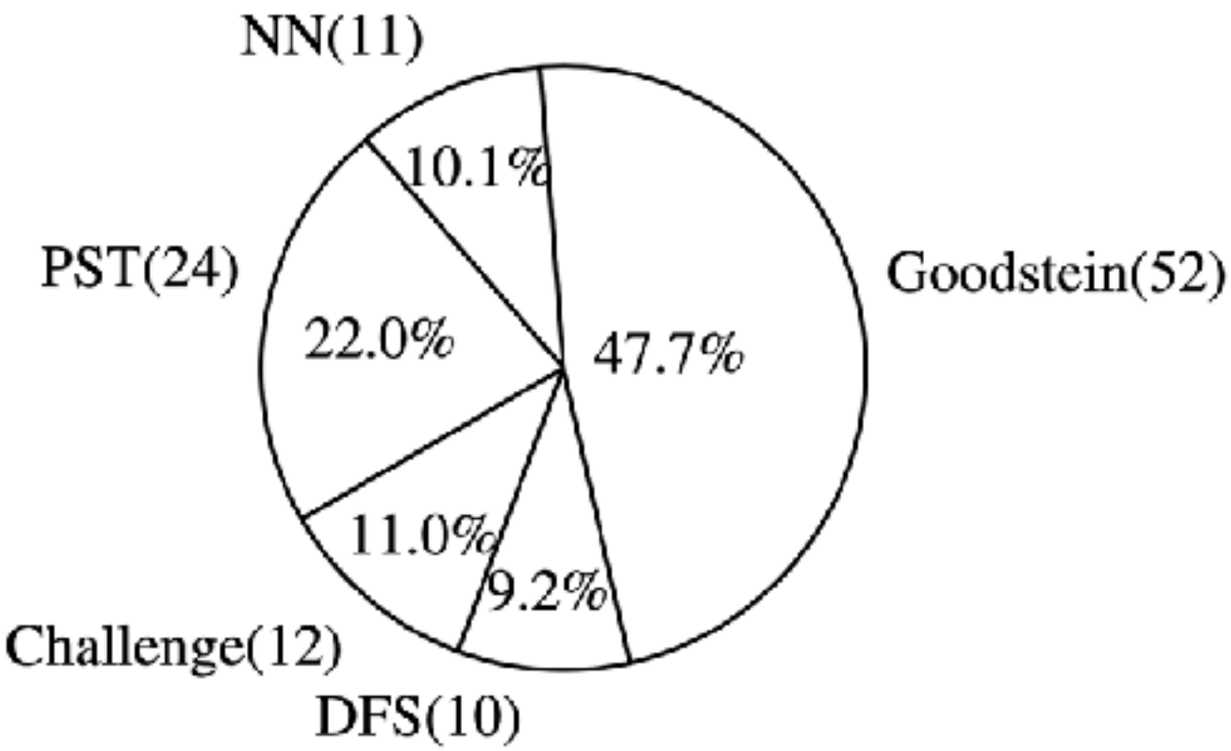
smart_induct: https://github.com/data61/PSL/releases/tag/0.1.7-alpha

Q&A!

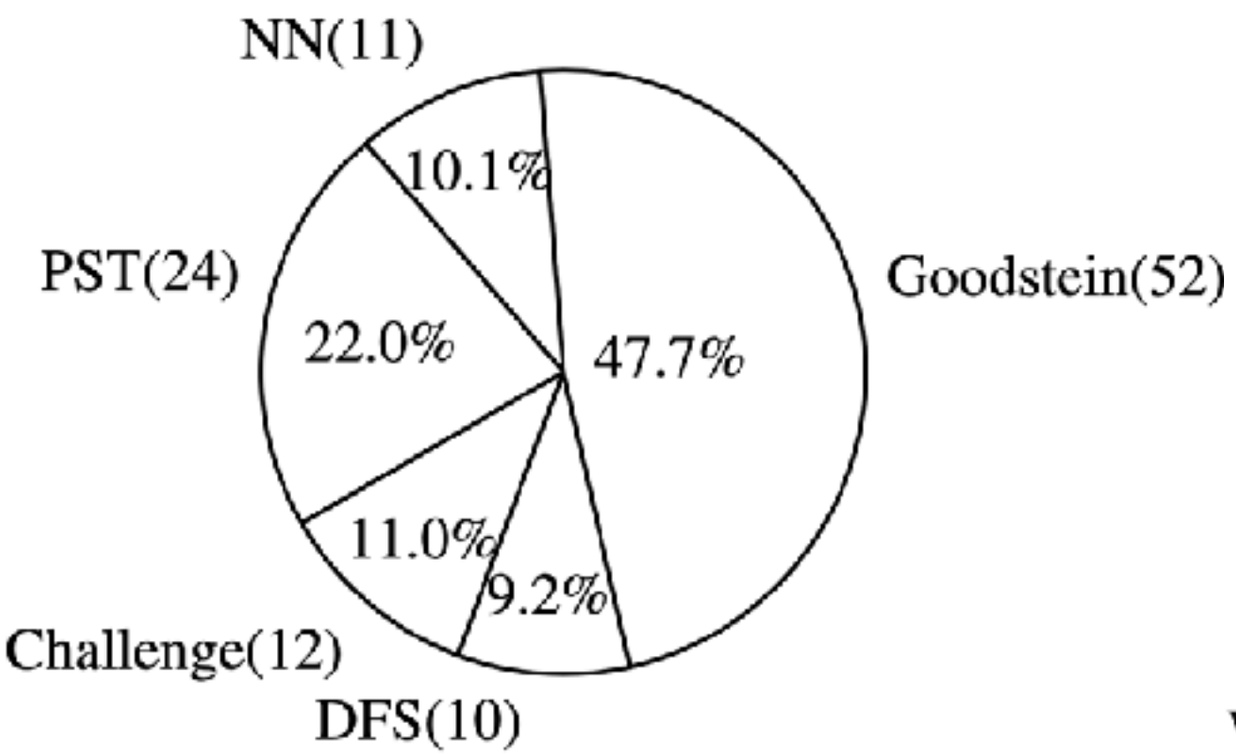
Do you want to have PSL and smart_inducdt in the official release of Isabelle?

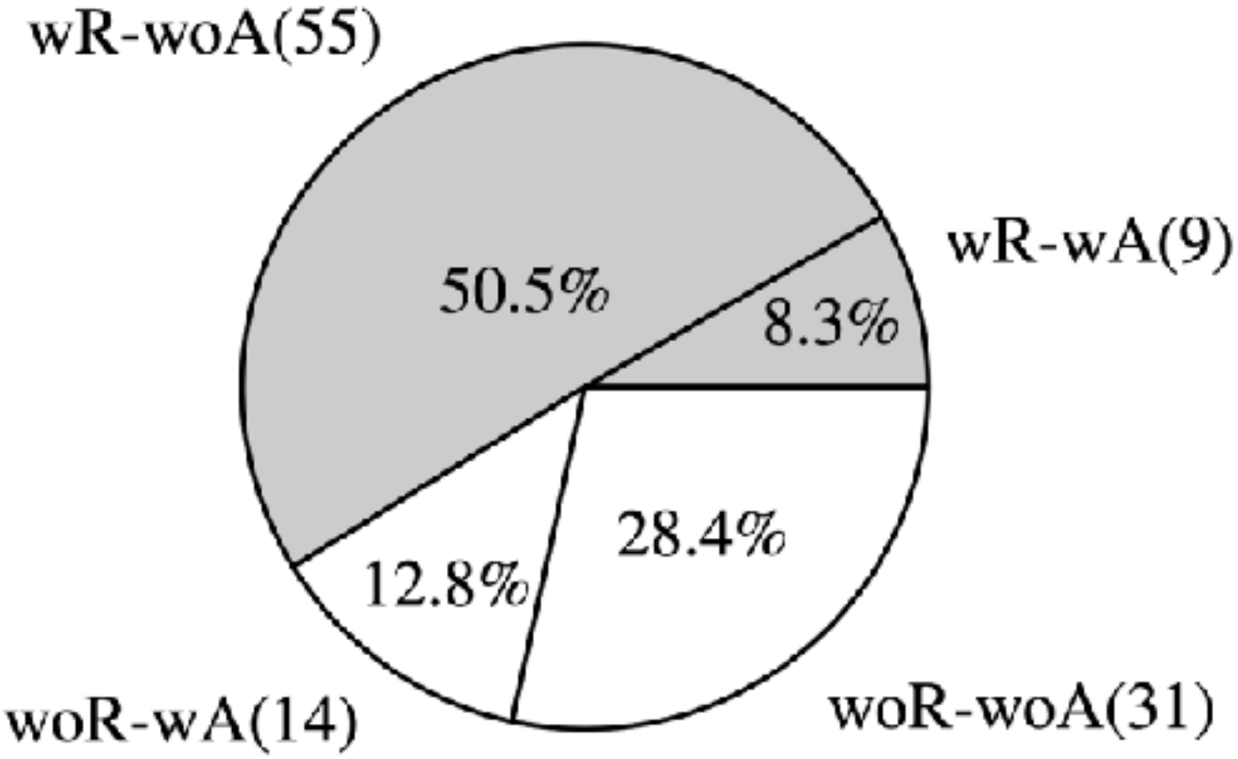
backup slides for Q&A

Dataset for evaluation
(109 proofs by induction from the AFP entries)

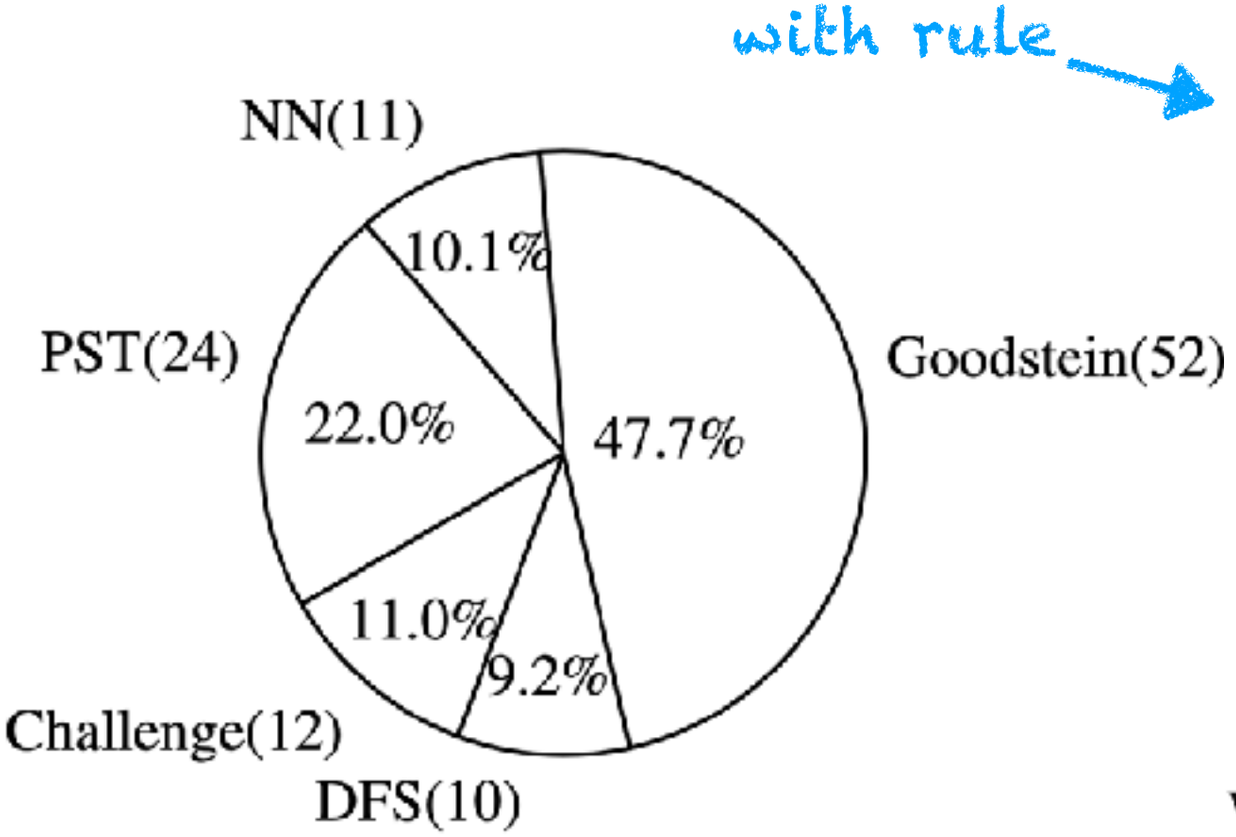


Dataset for evaluation
(109 proofs by induction from the AFP entries)

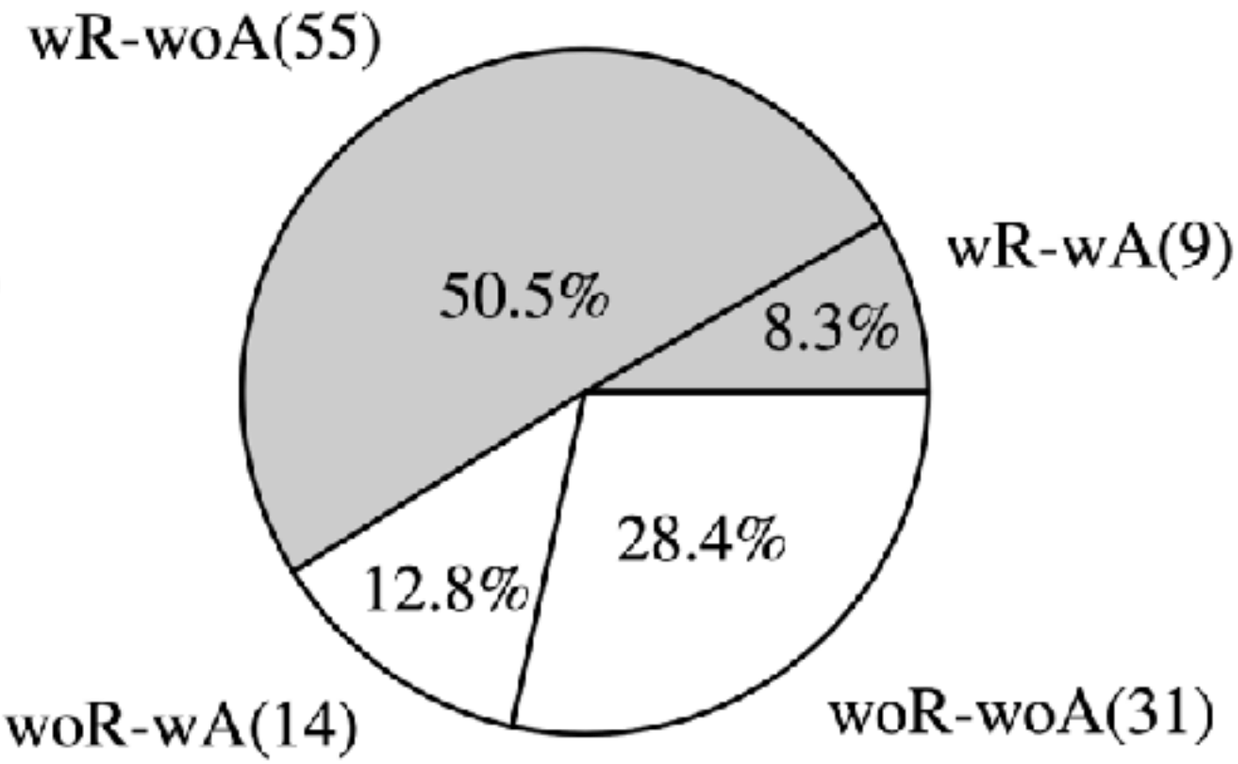




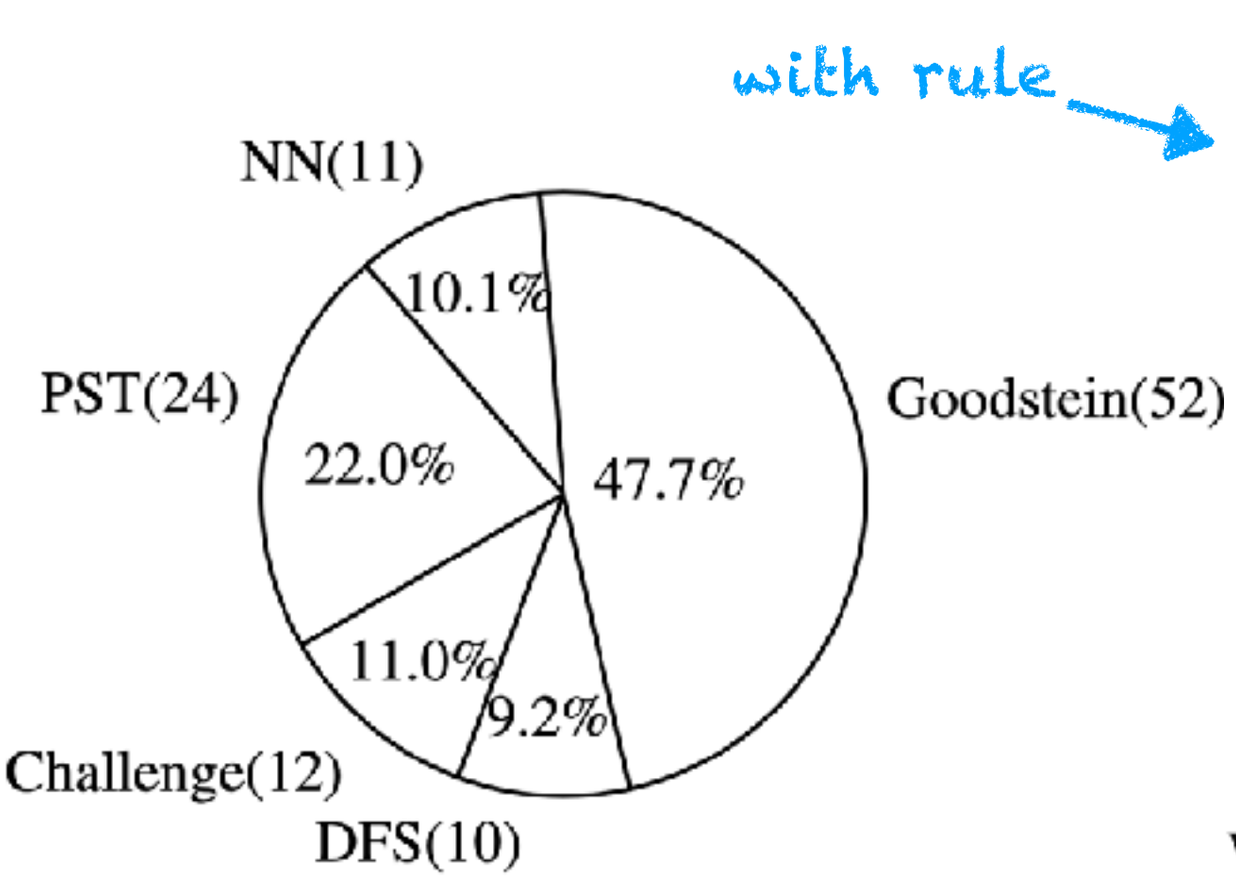
Dataset for evaluation
(109 proofs by induction from the AFP entries)

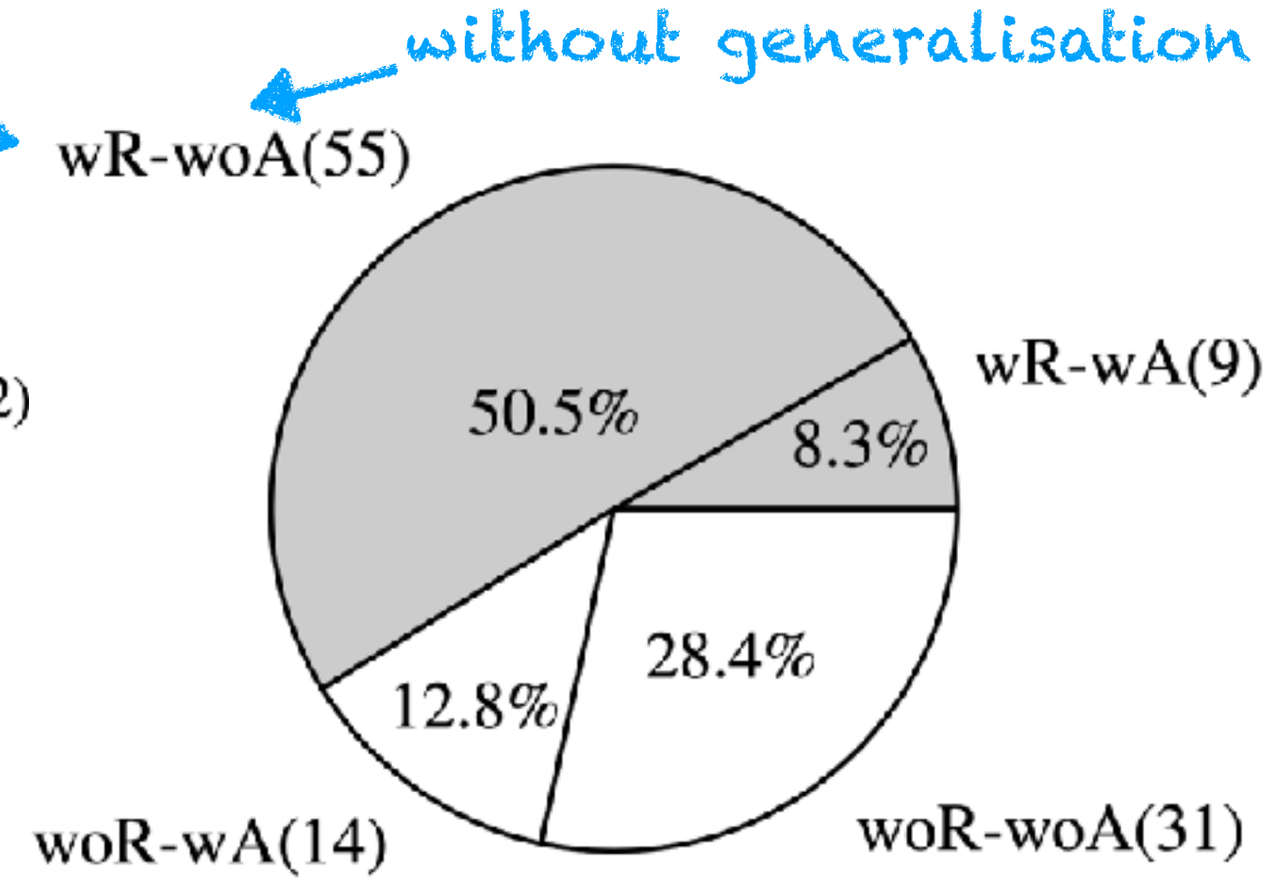


with rule

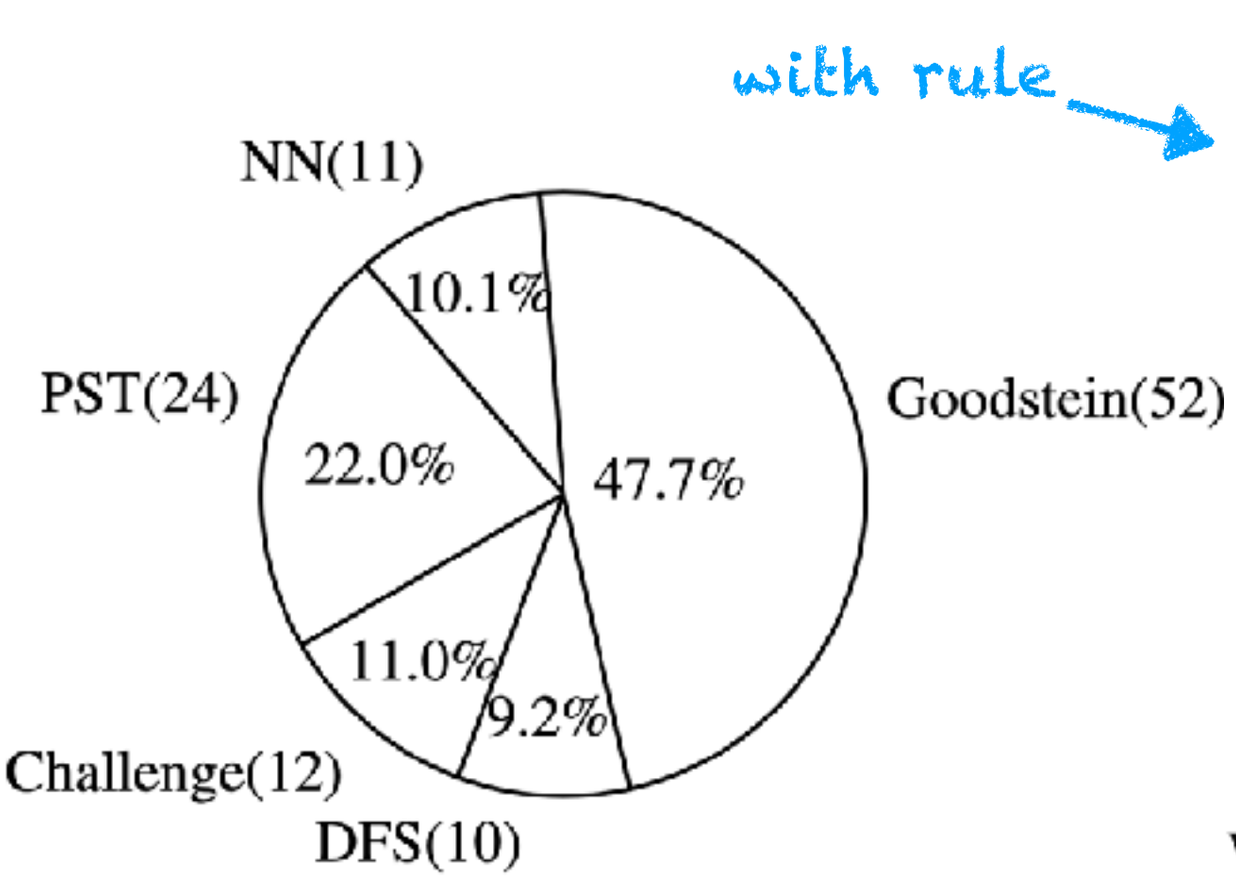


Dataset for evaluation
(109 proofs by induction from the AFP entries)



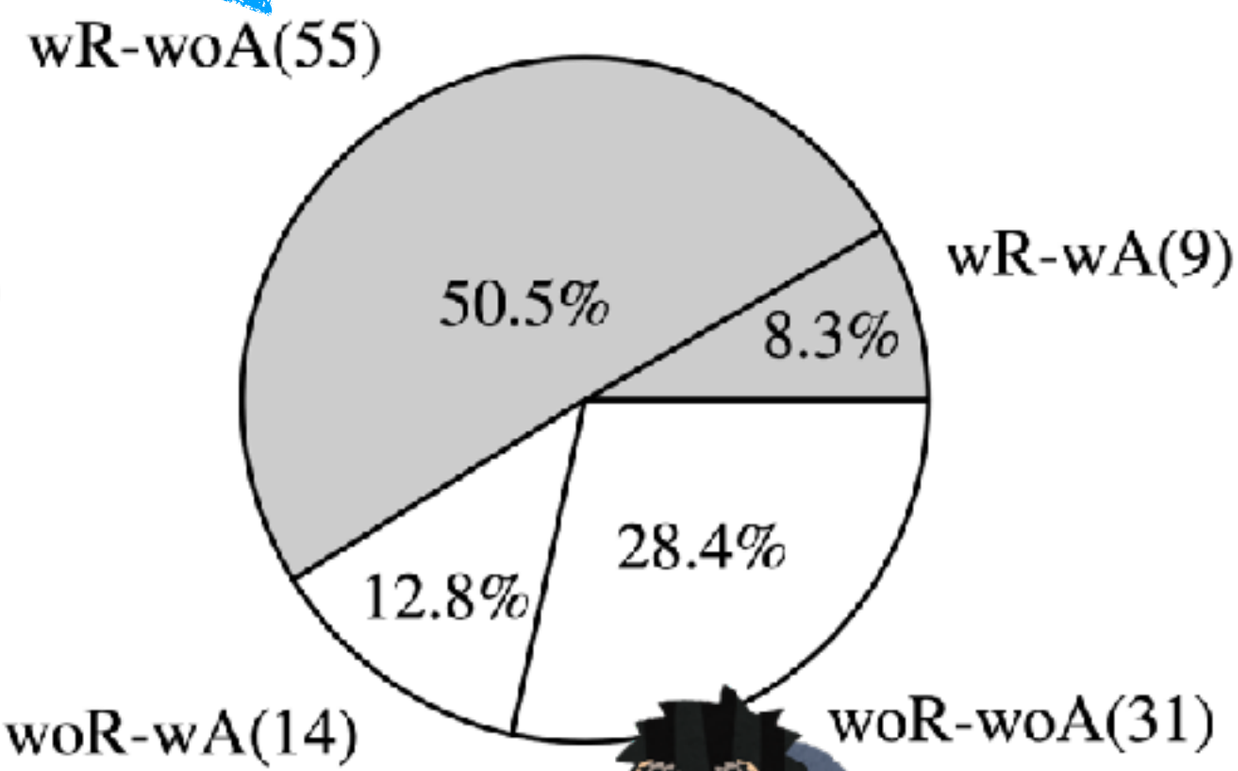


Dataset for evaluation
(109 proofs by induction from the AFP entries)



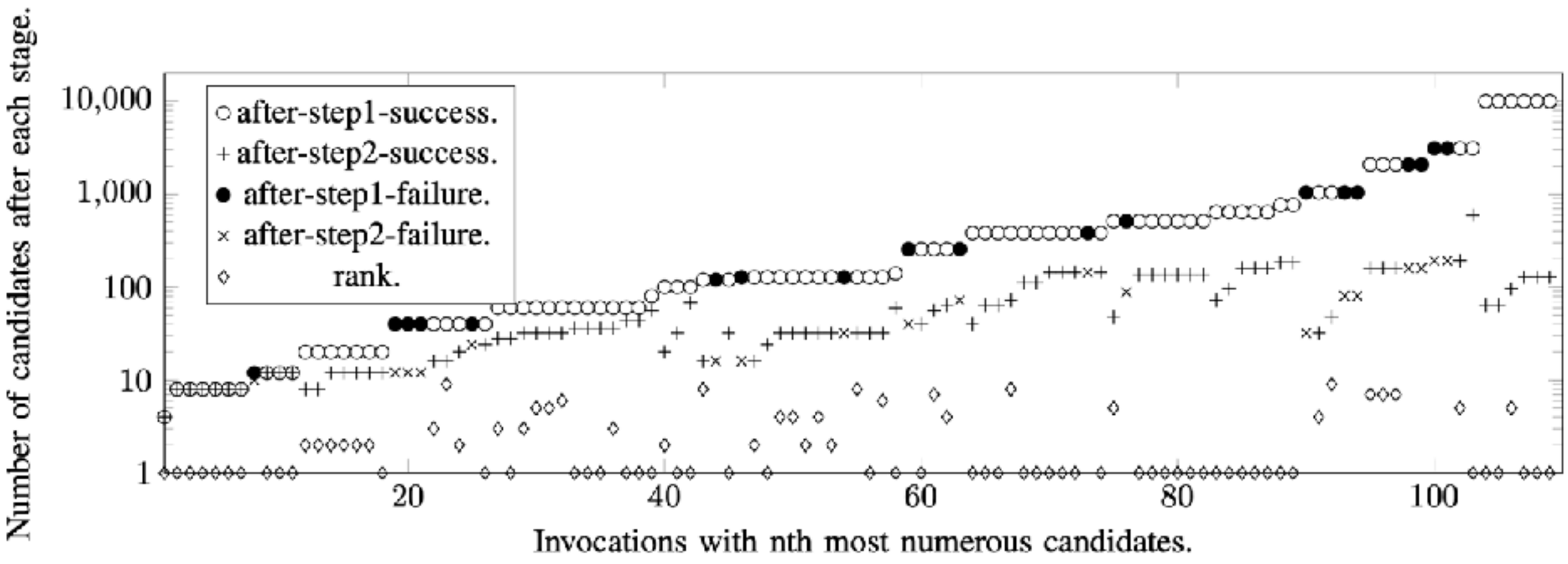
with rule

without generalisation



Some proofs are outside the scope of smart_induct…

-	w/ handwritten rule	w/o handwritten rule
w/ compound term	1 (0.9%)	1 (0.9%)
w/o compound term	5 (4.6%)	102 (93.6%)


Number of candidates after each stage.
10,000
1,000
100
10
1
20
40
60
80
100
Invocations with nth most numerous candidates.
after-step1-success.
after-step2-success.
after-step1-failure.
after-step2-failure.
rank.

[T,F,T], [T,T,T], [F,T,T]: bool list <- simple representation

LiFtEr LiFtEr LiFtEr

```
lemma "star r x y ⟹ sta                ⟹ star r x z"
by(induction rule: star.in              uto simp: step)

lemma "exec (is1 @ is2) s
       exec is2 s (exec is                )"
by(induct is1 s stk rule:e                uct) auto

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

<- small dataset about different domains

<- one abstract representation

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

[[T,F,T], [T,T,T], [F,T,T]]: bool list <- simple representation

```
lemma "star r x y ⟹ sta                ⟹ star r x z"
by(induction rule: star.in              uto simp: step)

lemma "exec (is1 @ is2) s
       exec is2 s (exec is                )"
by(induct is1 s stk rule:e                uct) auto
```

LiFtEr LiFtEr LiFtEr

<- small dataset about different domains

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

<- one abstract representation

```
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

[[T,F,T], [T,T,T], [F,T,T]]: bool list <- simple representation

LiFtEr LiFtEr LiFtEr

```
lemma "star r x y ⟹ sta                ⟹ star r x z"
by(induction rule: star.in              uto simp: step)

lemma "exec (is1 @ is2) s
       exec is2 s (exec is                )"
by(induct is1 s stk rule:e                 uct) auto
```

<- small dataset about different domains

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

<- one abstract representation

```
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)"
```

[T,F,T], [T,T,T], [F,T,T]: bool list <- simple representation

```
lemma "star r x y ⟹ sta                ⇒ star r x z"
by(induction rule: star.in              uto simp: step)

lemma "exec (is1 @ is2) s
       exec is2 s (exec is              )"
by(induct is1 s stk rule:e              uct) auto

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

LiFtEr LiFtEr LiFtEr

<- small dataset about different domains

<- one abstract representation

LiFtEr 2 LiFtEr 2

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

[[T,F,T], [T,T,T], [F,T,T]]: bool list <- simple representation

```
lemma "star r x y ⟹ sta                ⟹ star r x z"
by(induction rule: star.in             uto simp: step)

lemma "exec (is1 @ is2) s
       exec is2 s (exec is                )"
by(induct is1 s stk rule:e                uct) auto
```

LiFtEr LiFtEr LiFtEr

<- small dataset about different domains

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

<- one abstract representation

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

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

[[T,F,T], [T,T,T], [F,T,T]]: bool list <- simple representation

```
lemma "star r x y ⟹ sta                ⟹ star r x z"
by(induction rule: star.in             uto simp: step)

lemma "exec (is1 @ is2) s
       exec is2 s (exec is                )"
by(induct is1 s stk rule:e                uct) auto

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

LiFtEr LiFtEr LiFtEr

<- small dataset about different domains

<- one abstract representation

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

LiFtEr 2 LiFtEr 2

<- 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)"
```

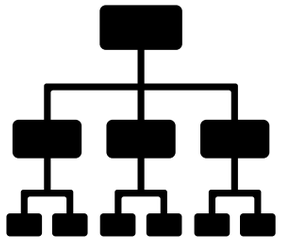
<- relevant definitions

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

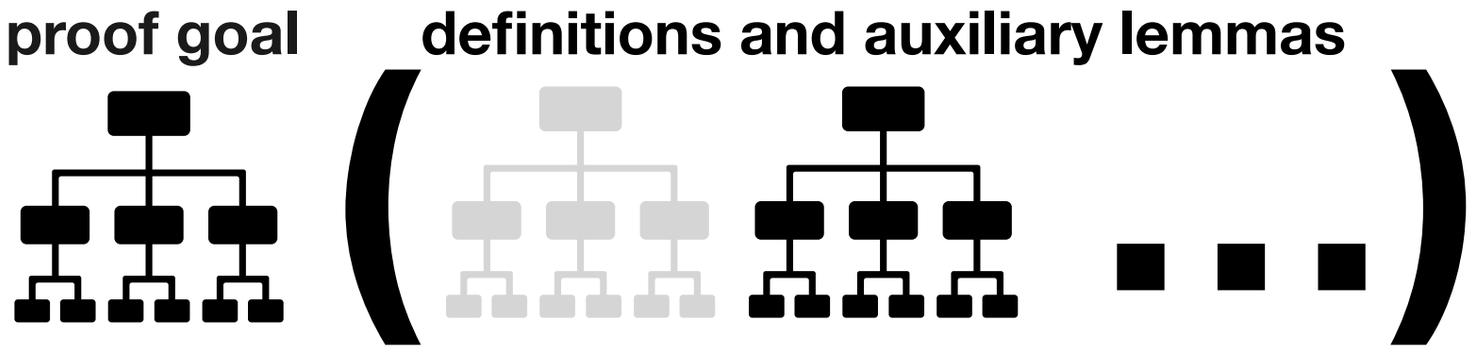
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions

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

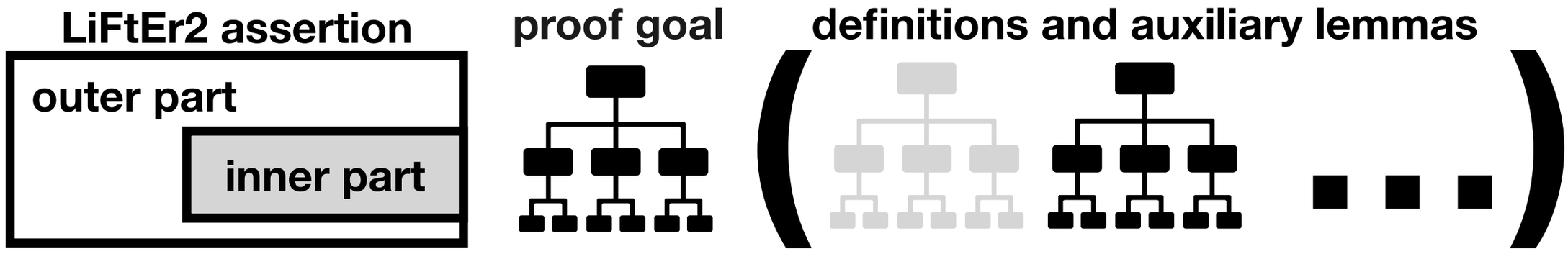
proof goal



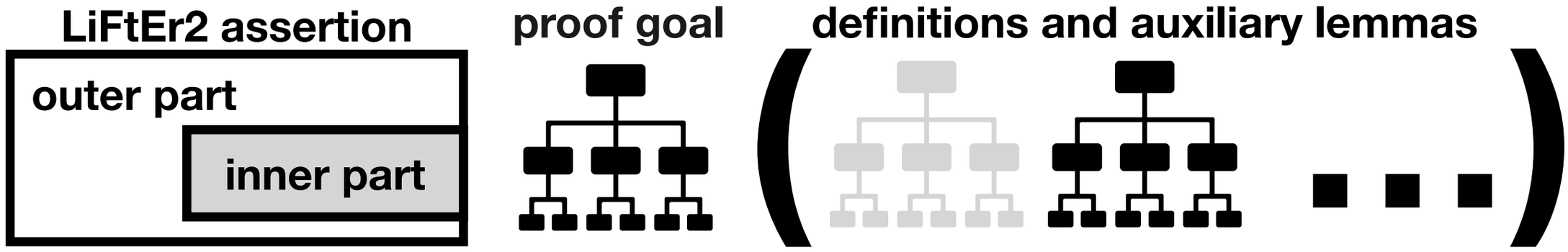
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions



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

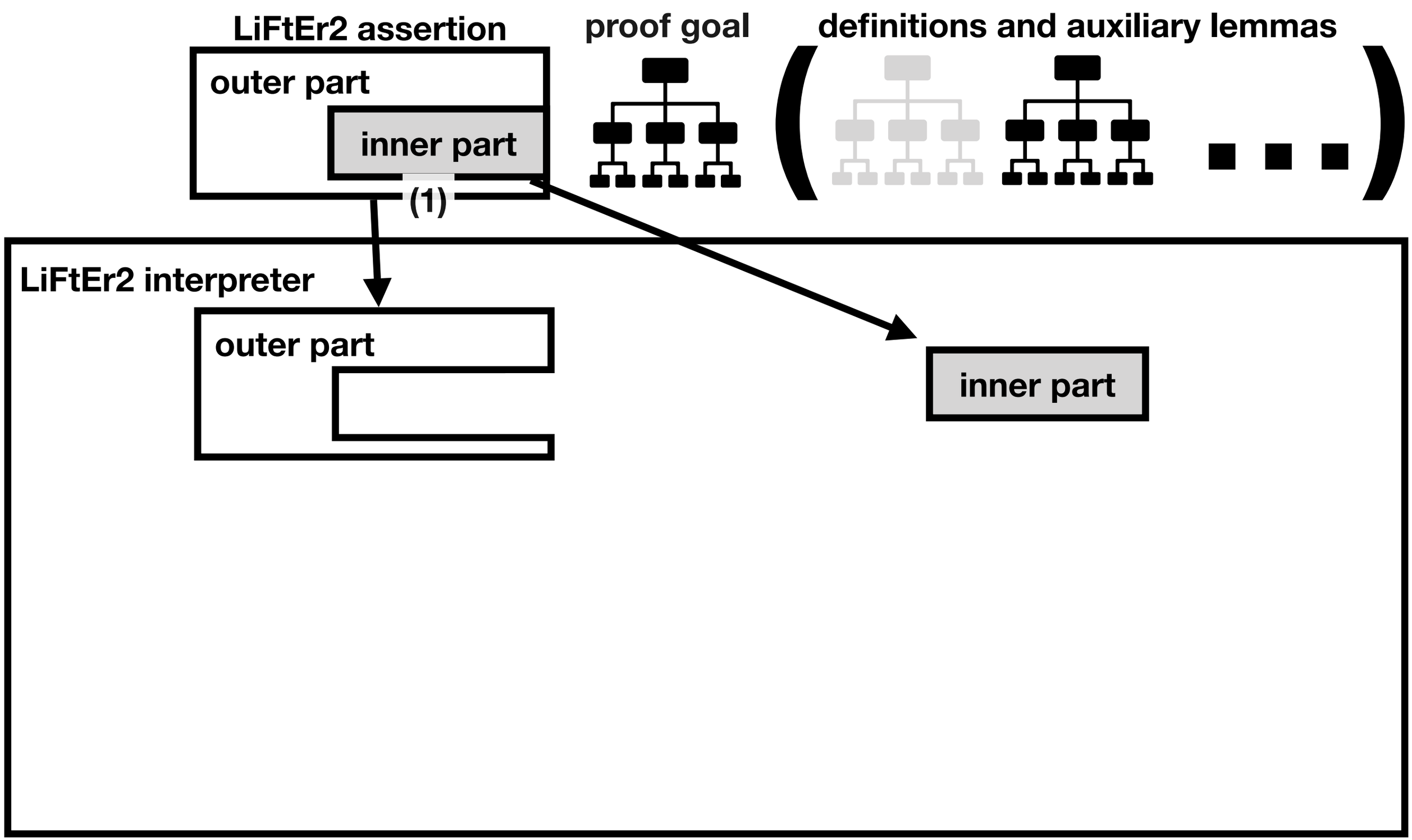


LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions

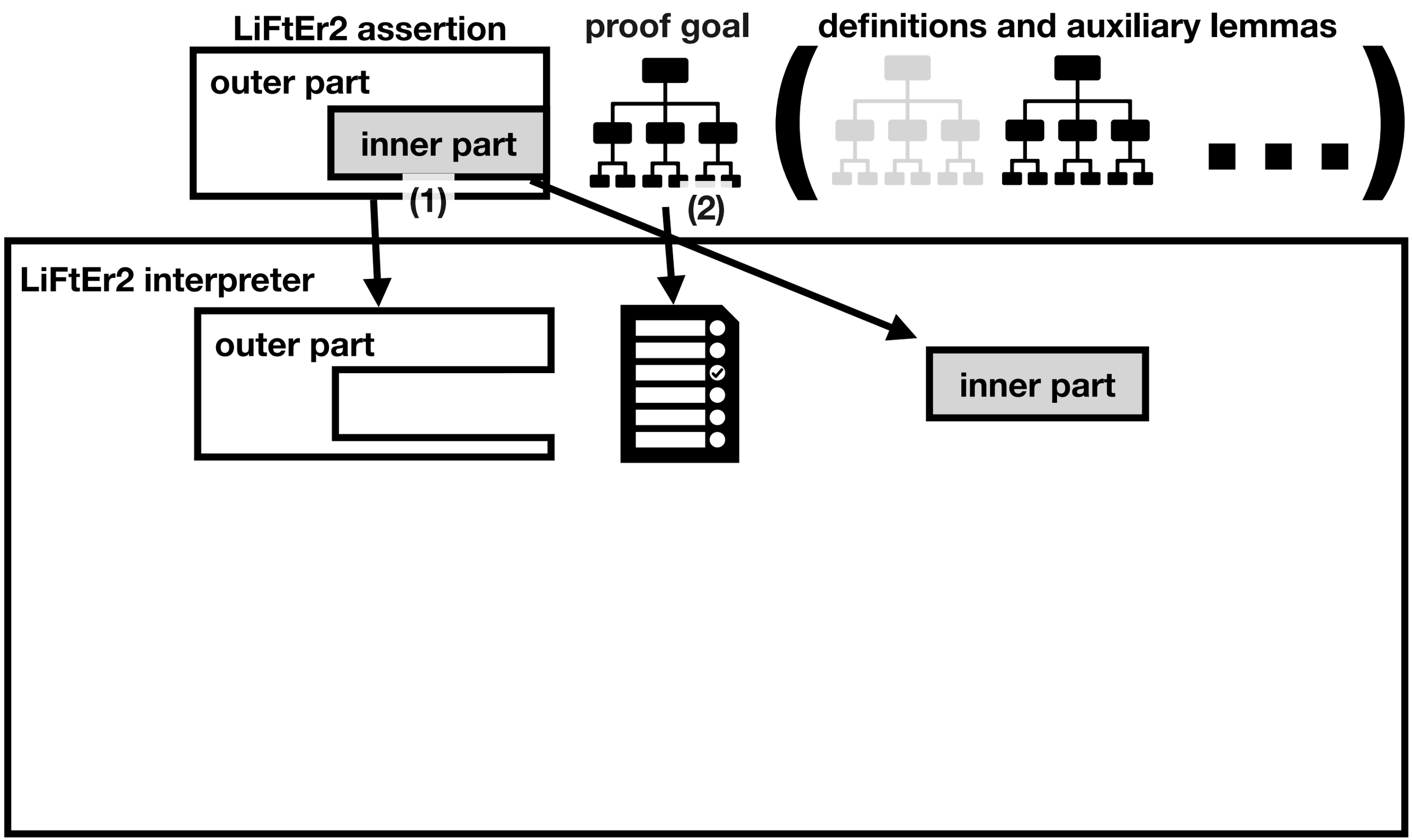


LiFtEr2 interpreter

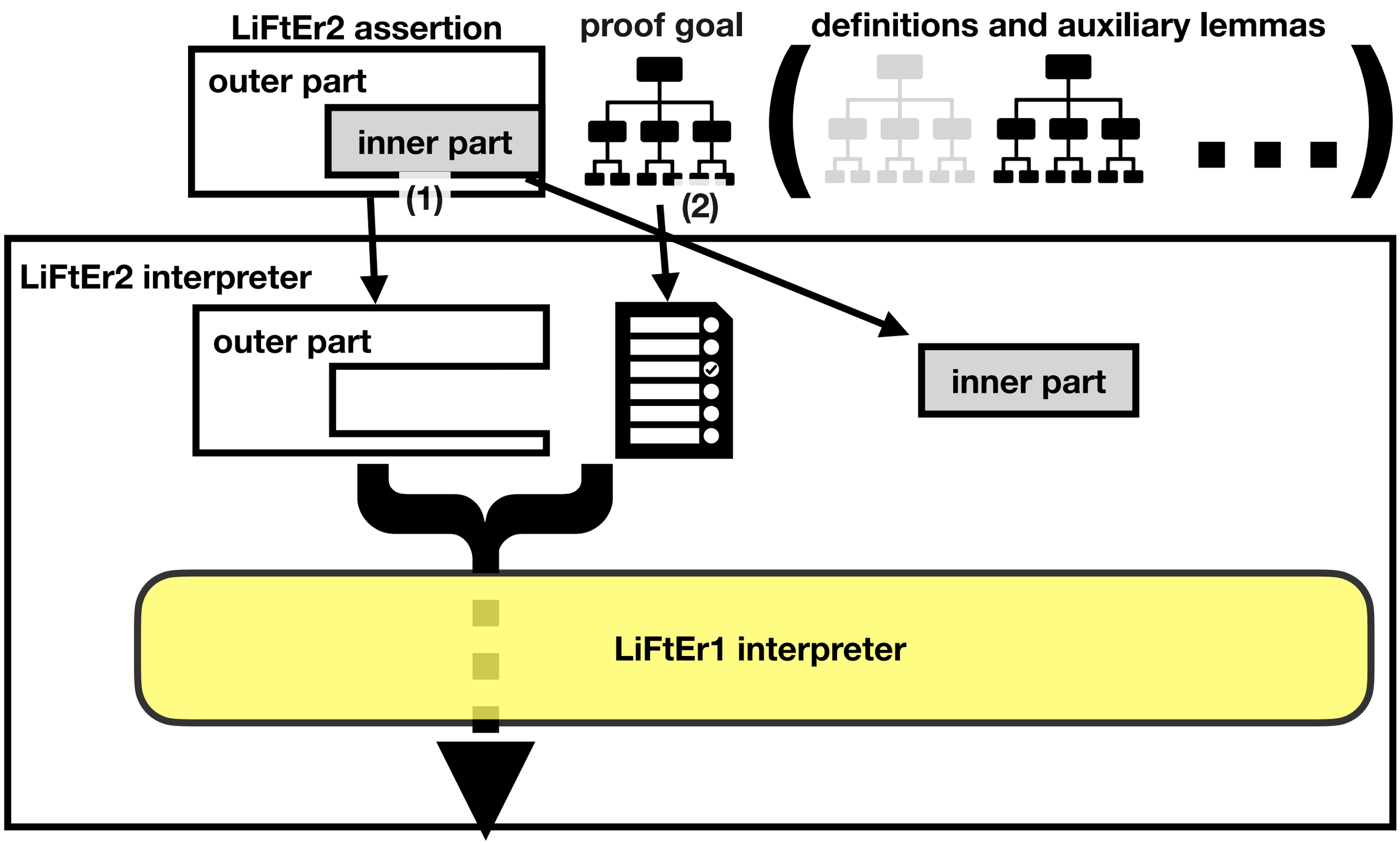
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions



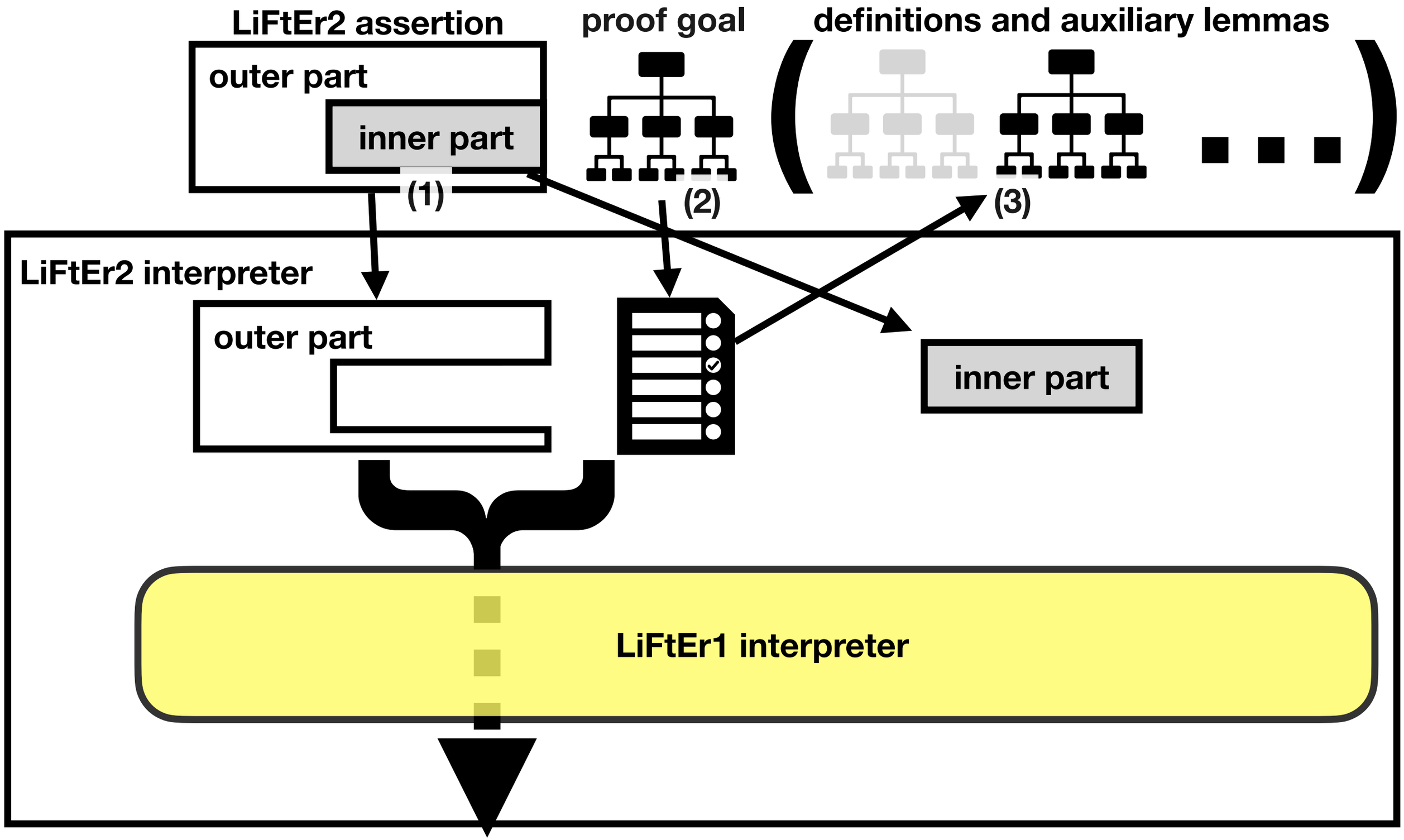
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions



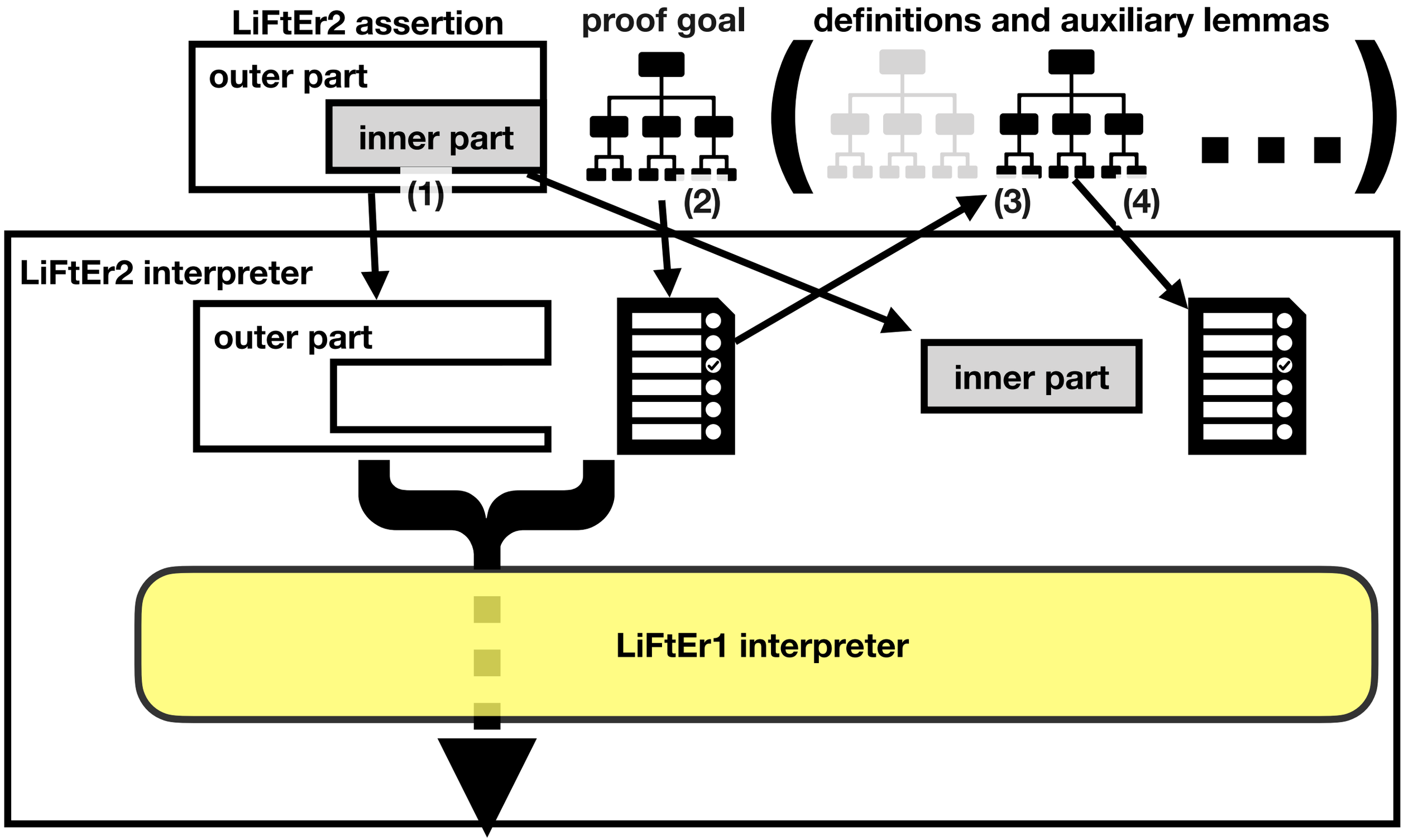
LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



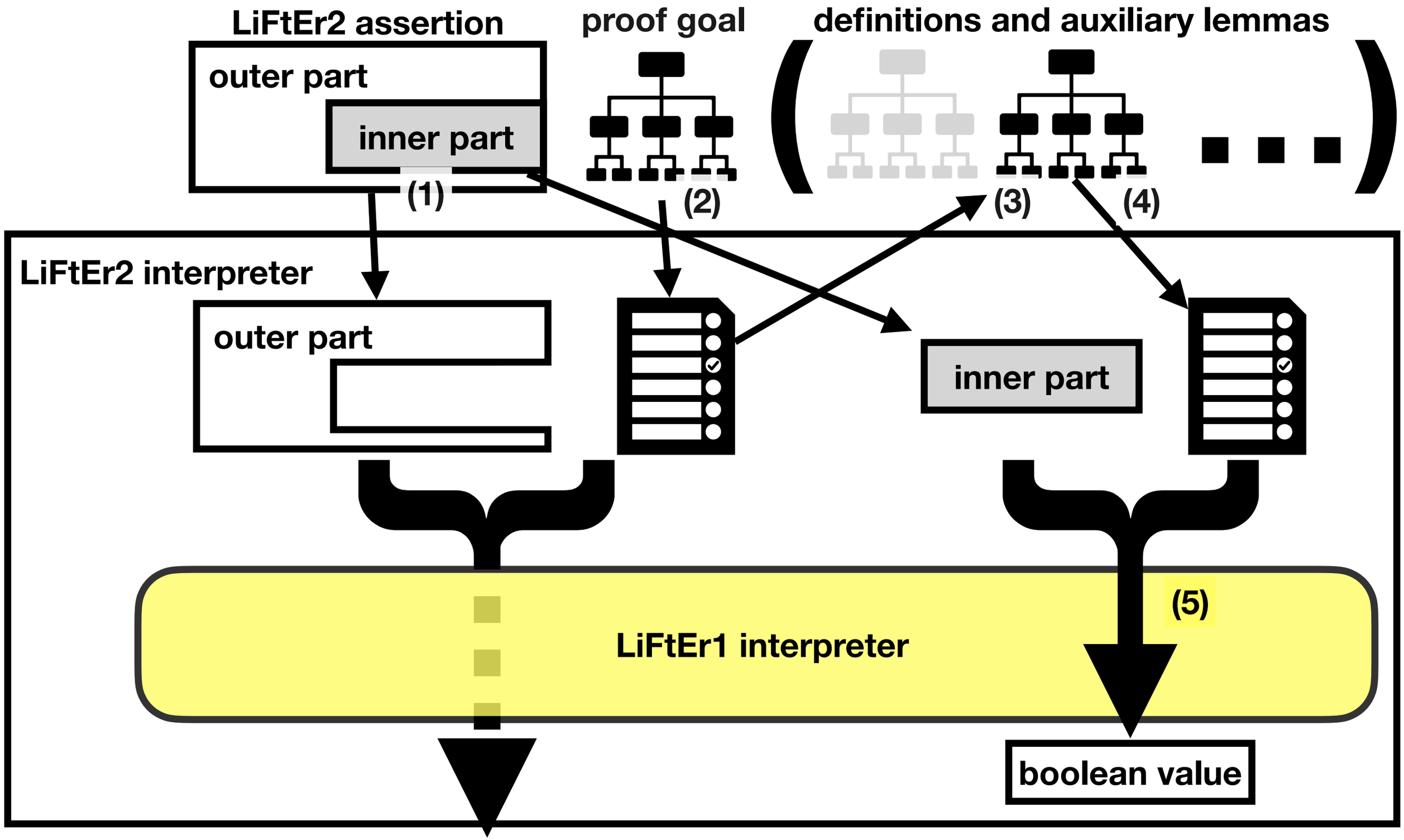
LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



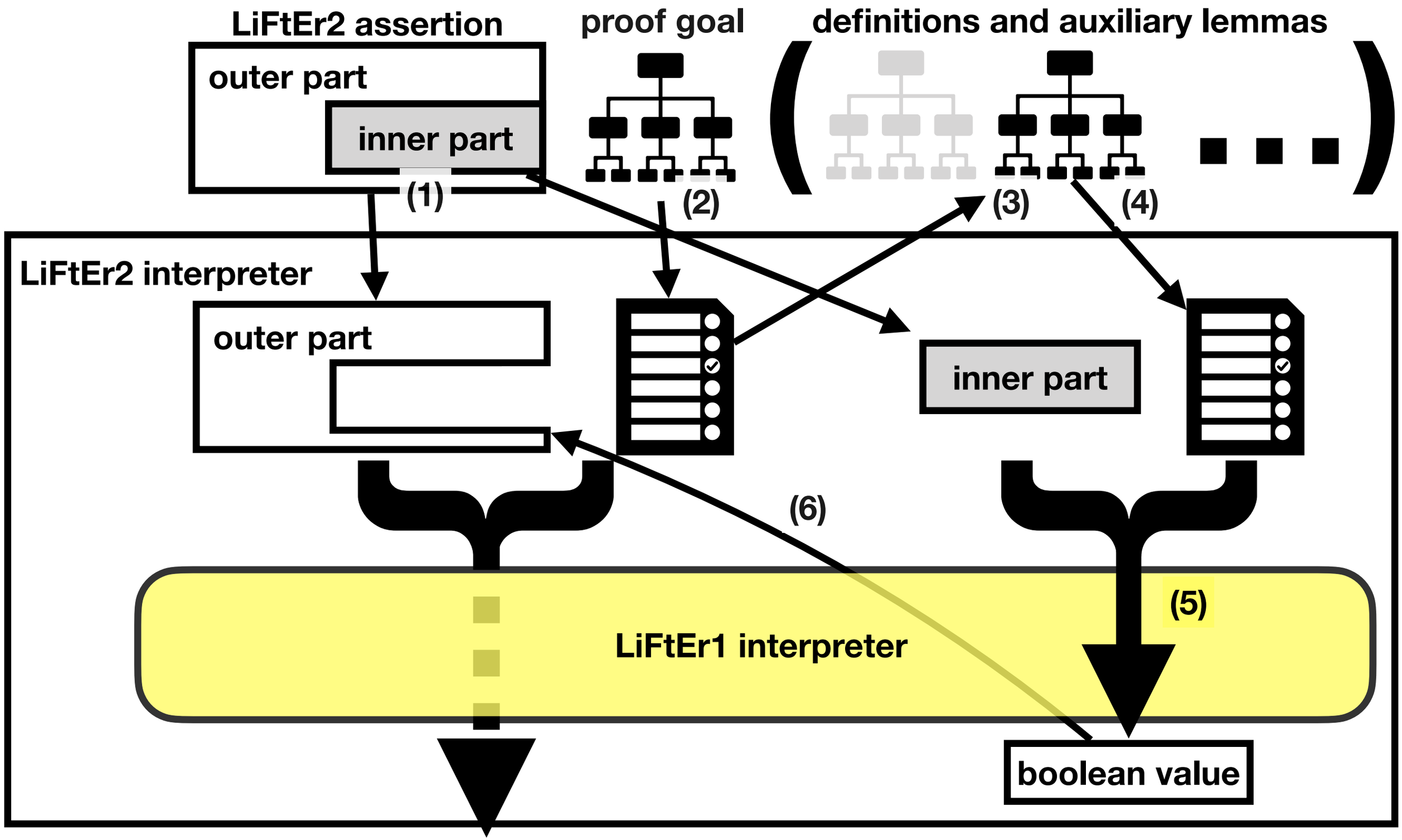
LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



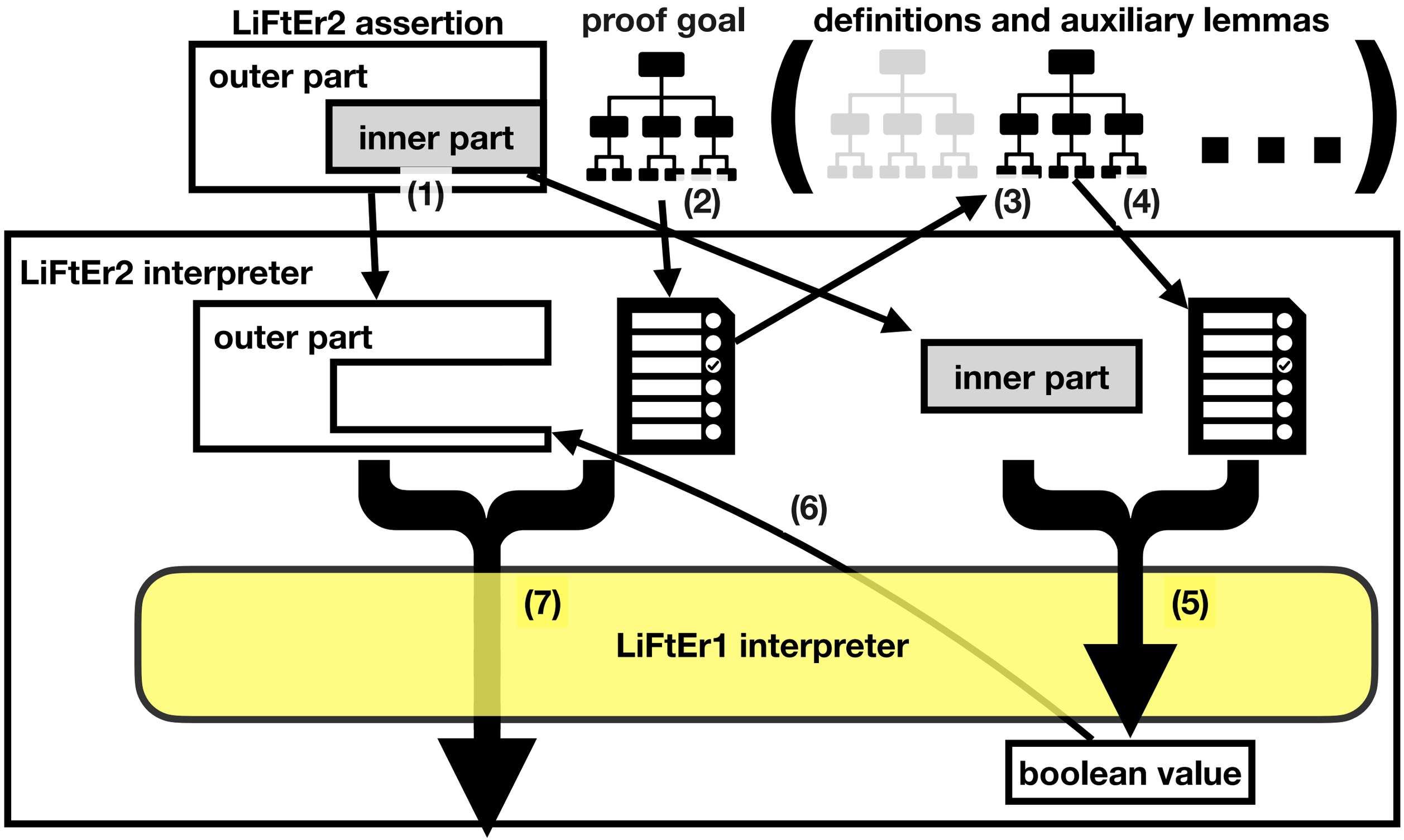
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions



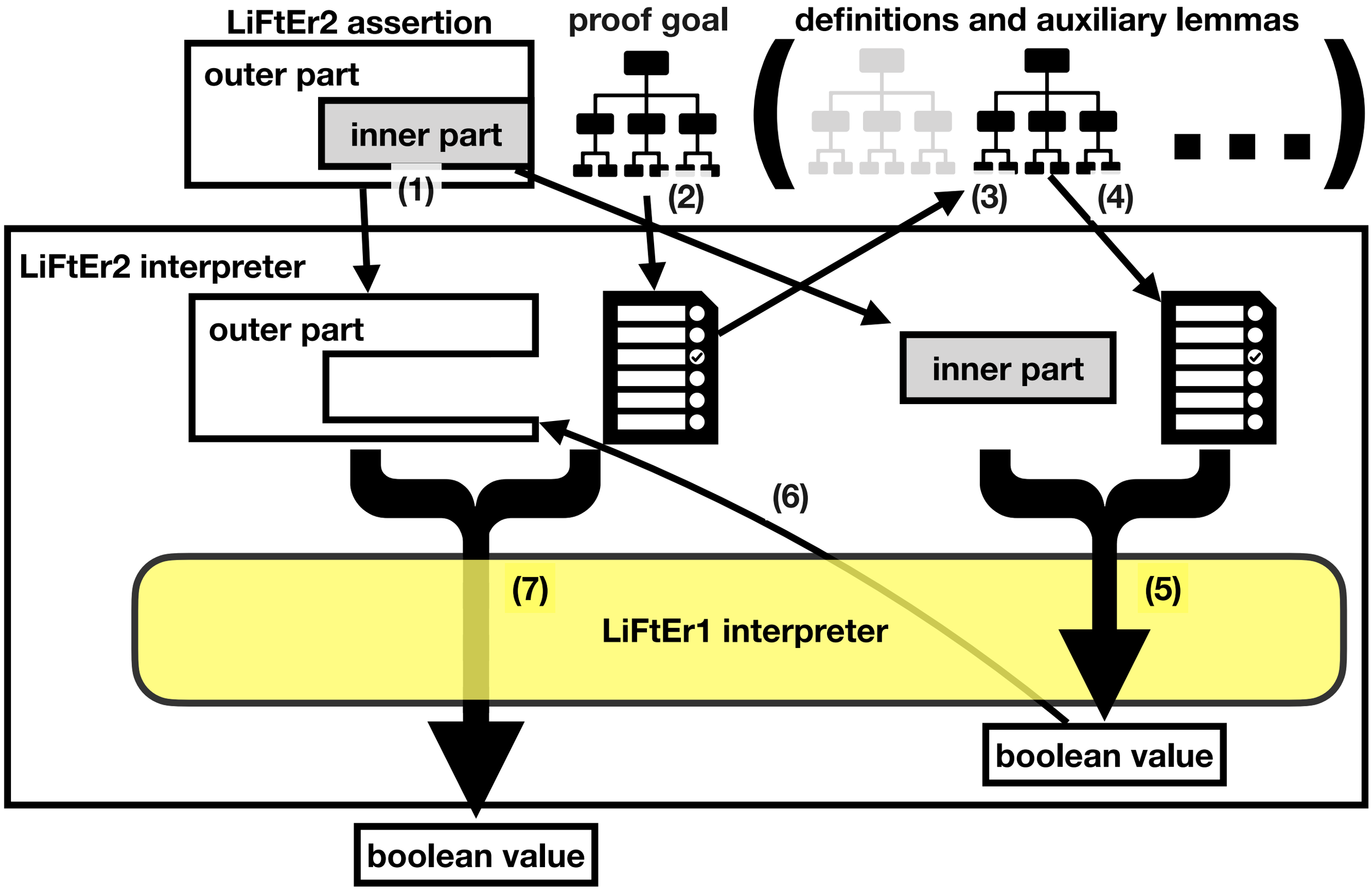
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions



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

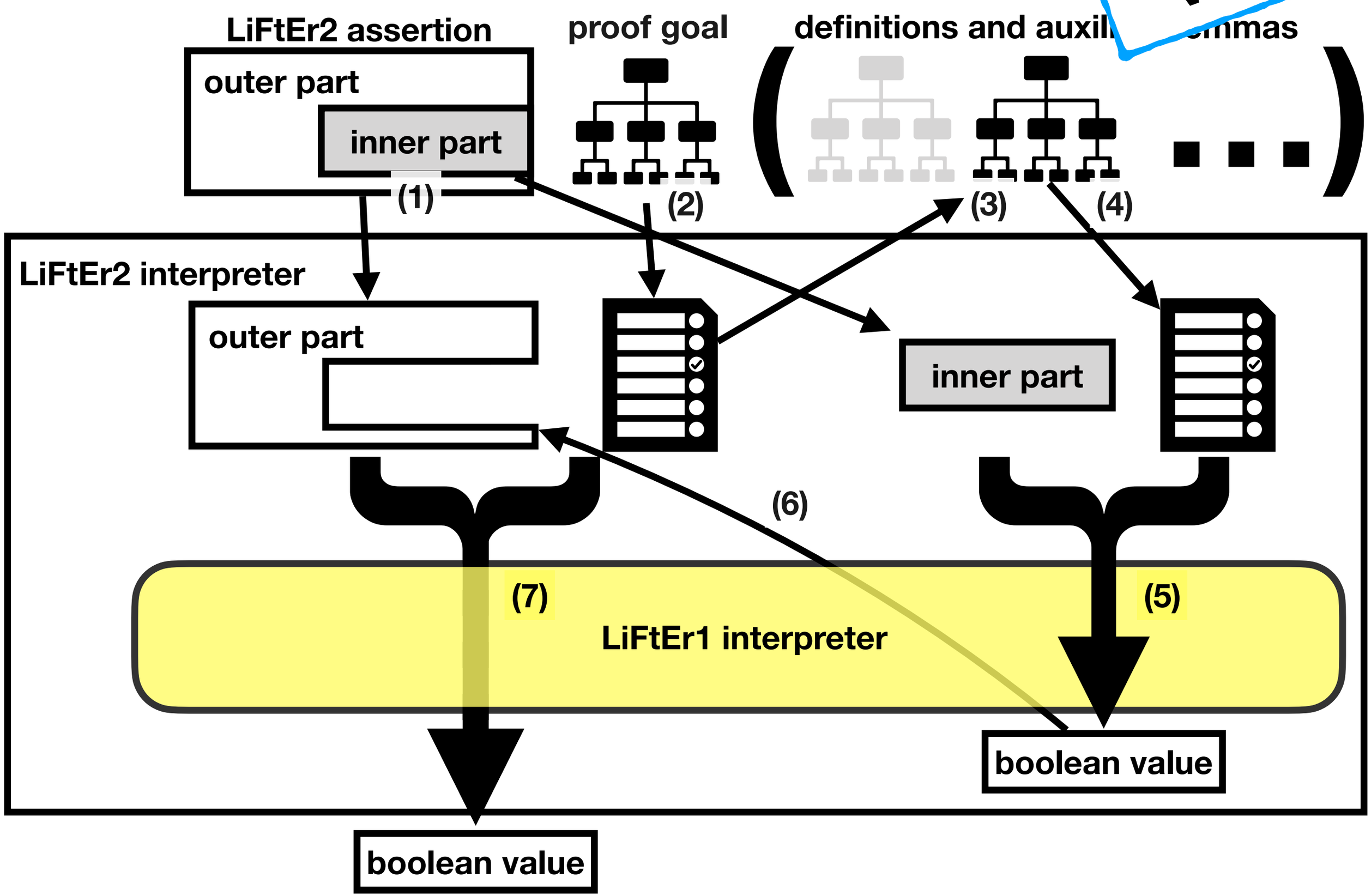


LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool

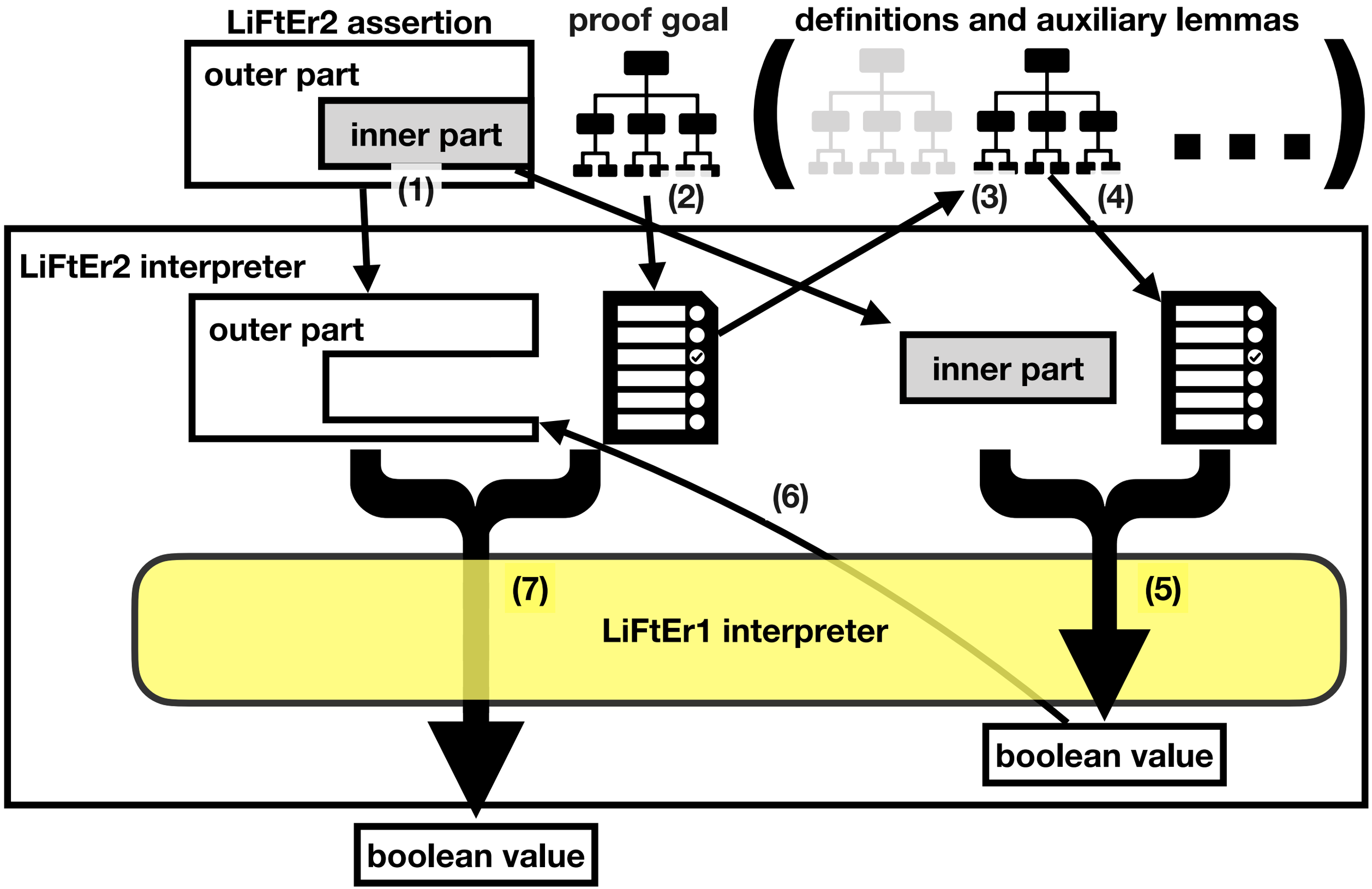


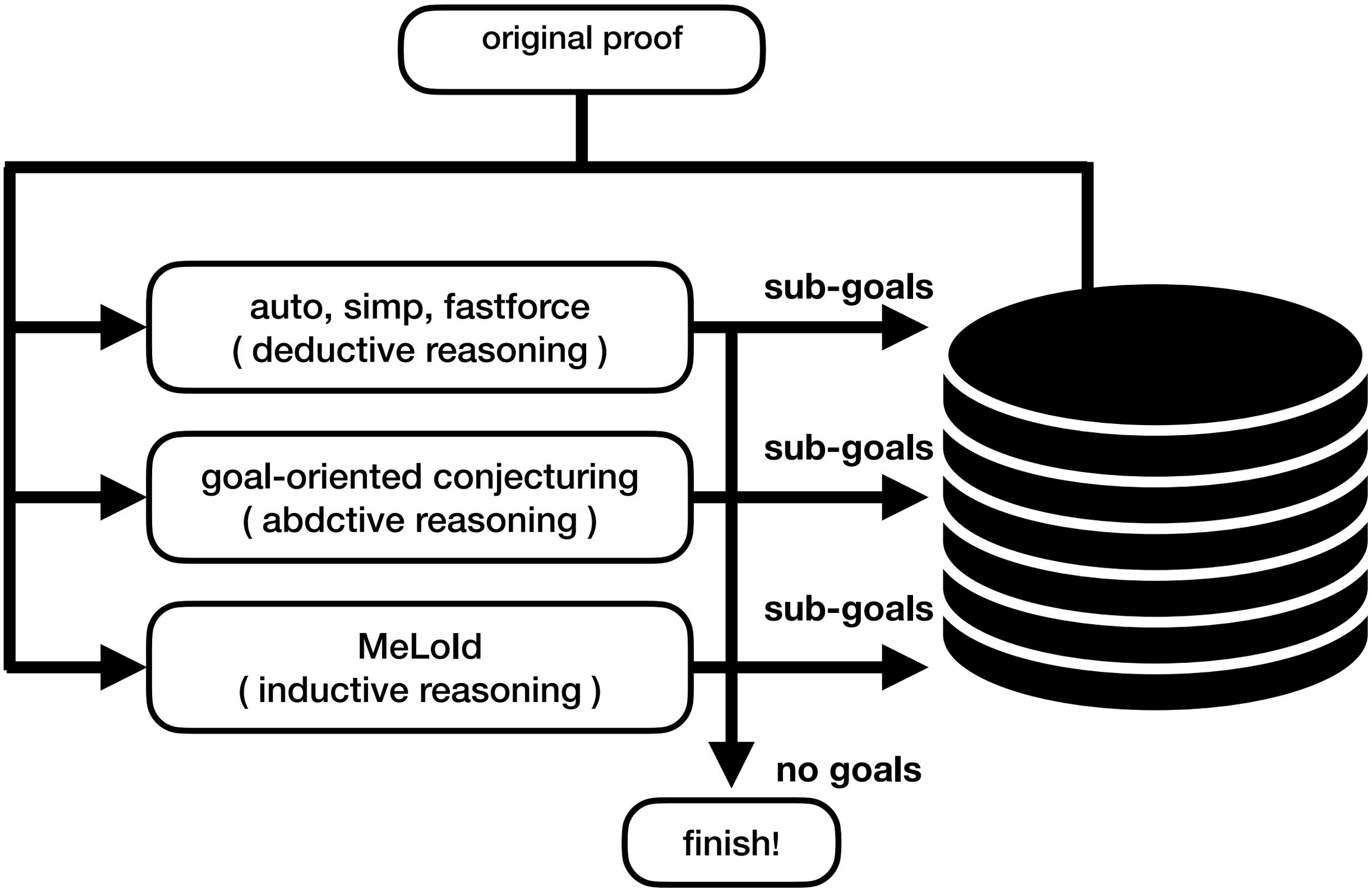
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions

WIP!



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




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]
```

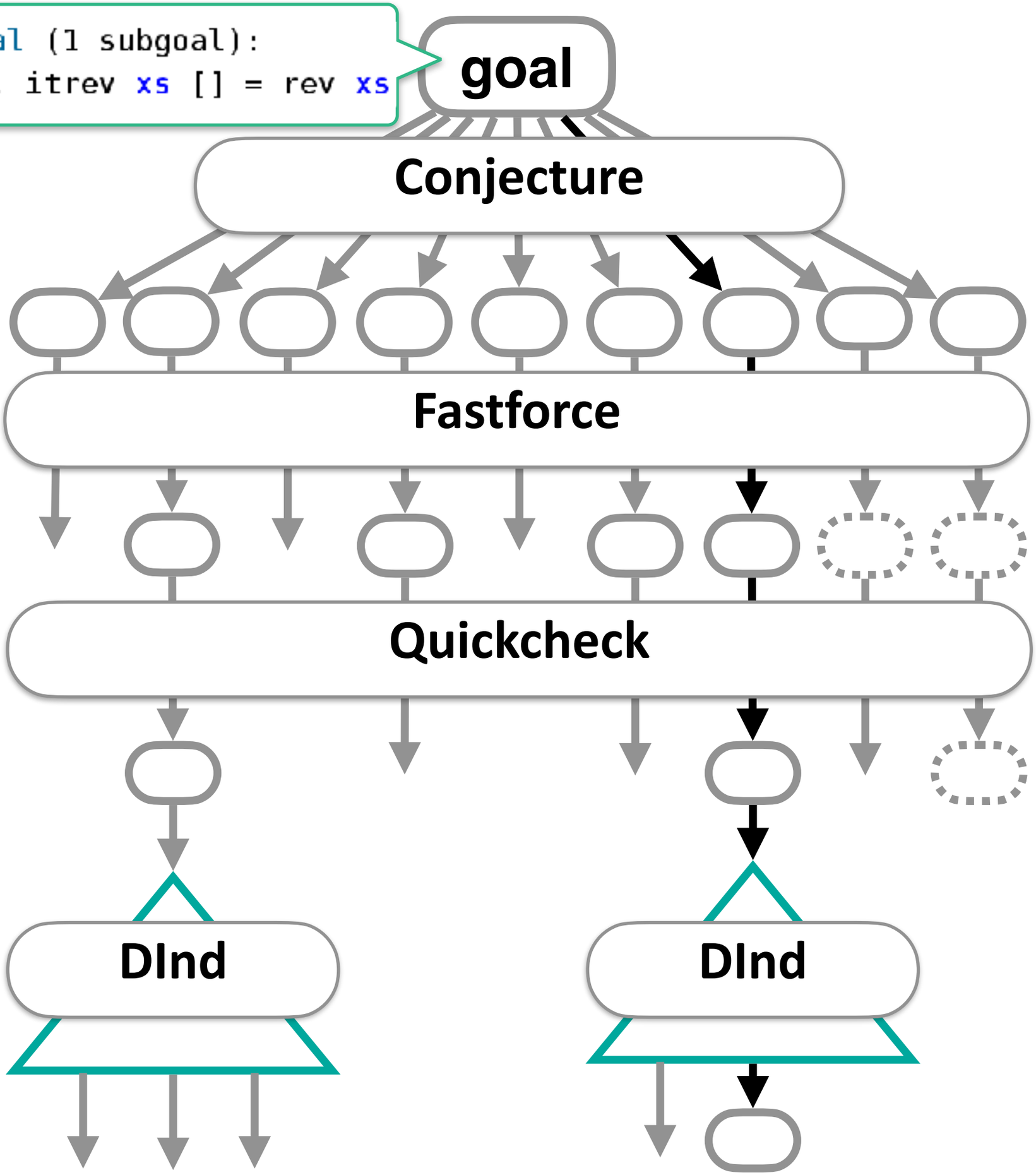
Quickcheck

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
```

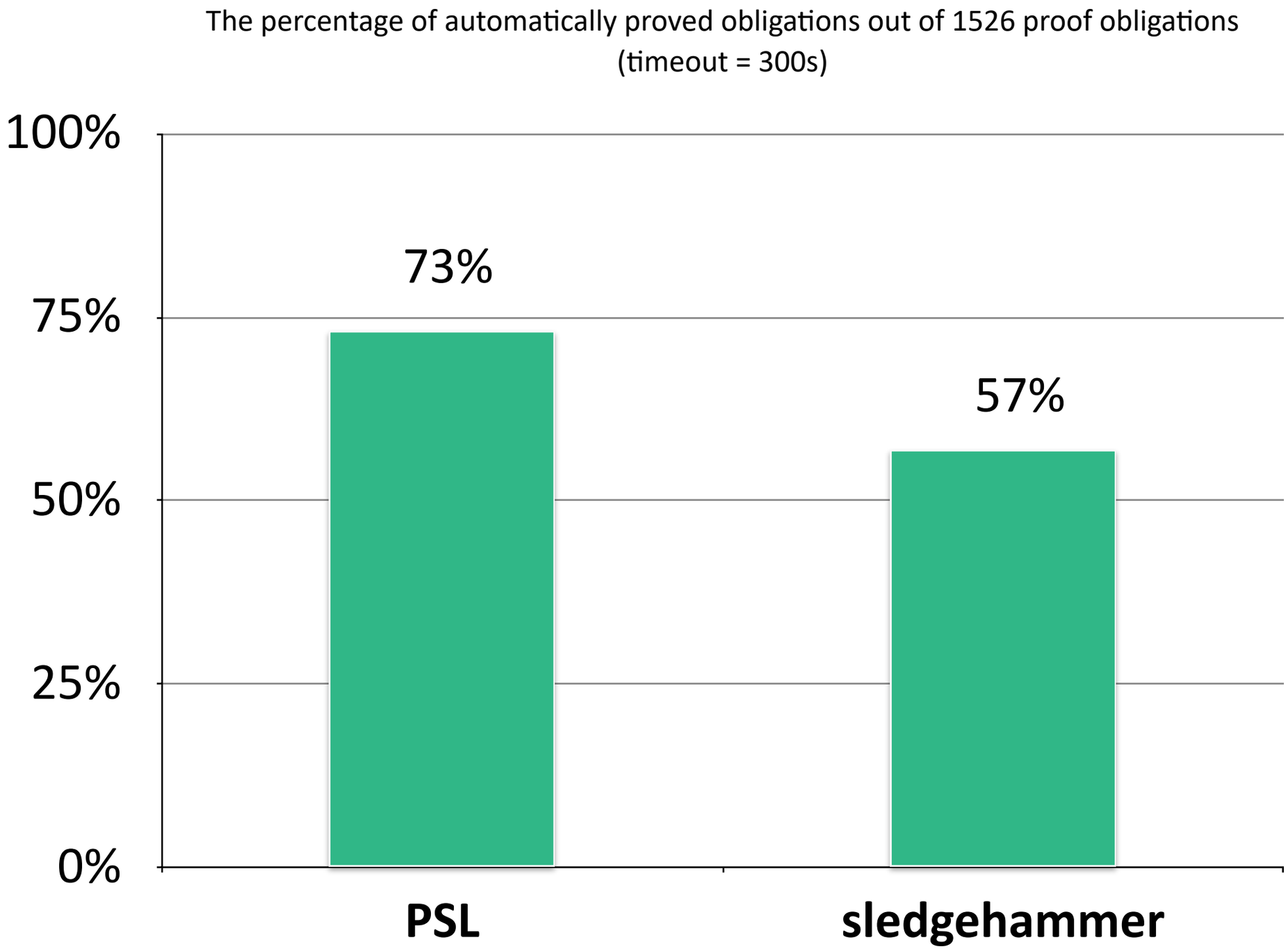
DInd

DInd

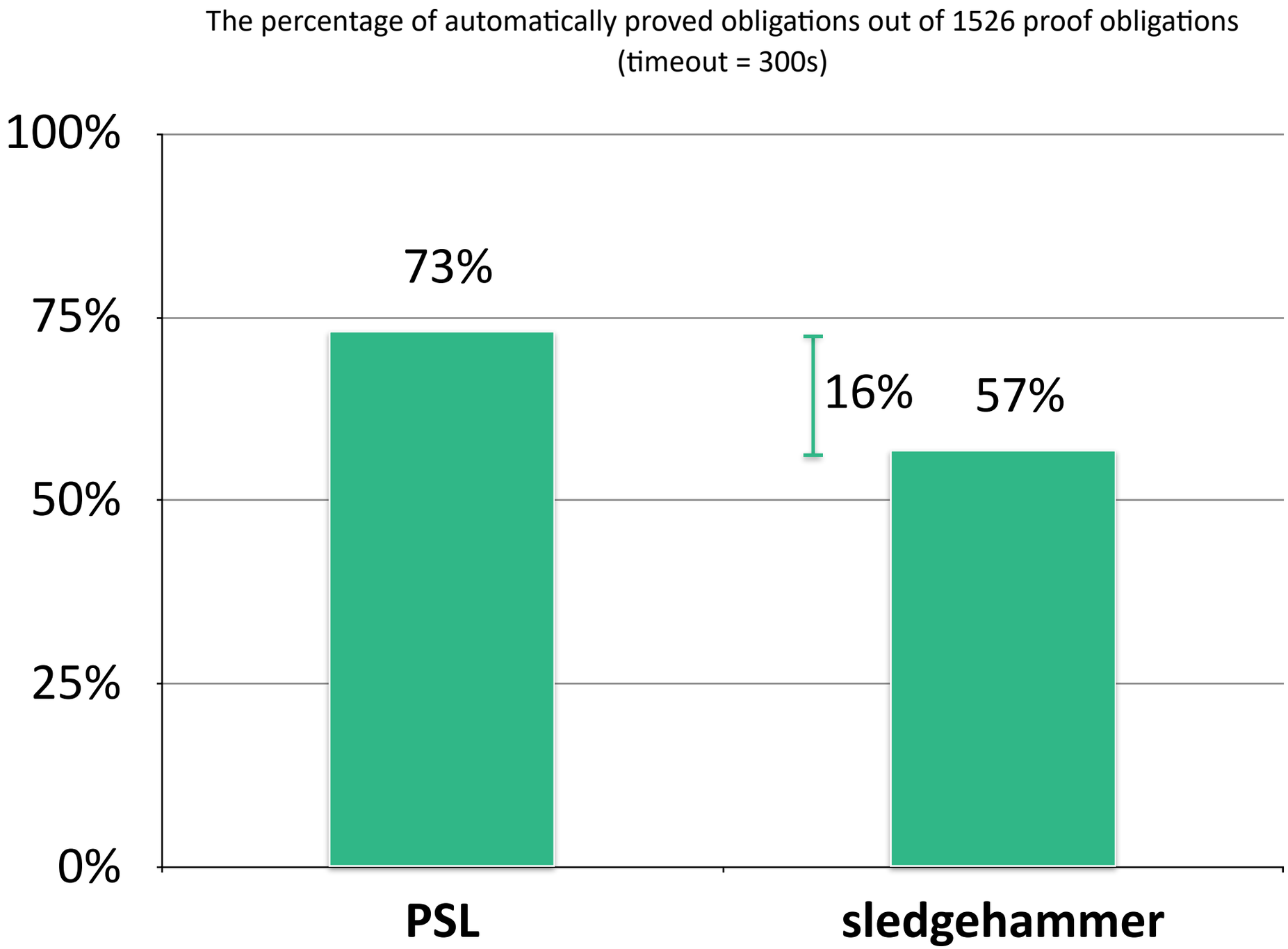
```
apply (induct xs)
apply auto
done
```

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

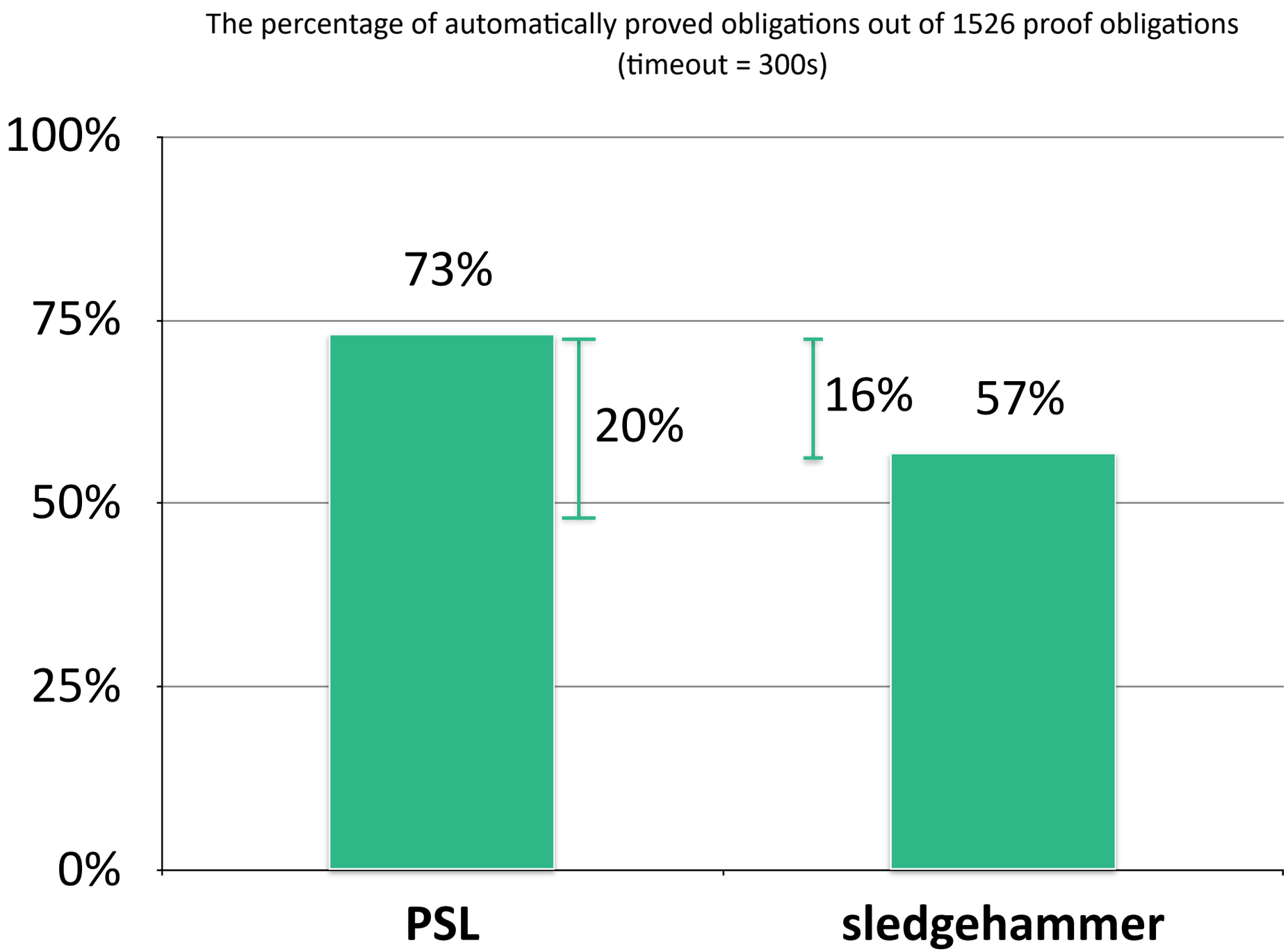
PSL vs sledgehammer



PSL vs sledgehammer

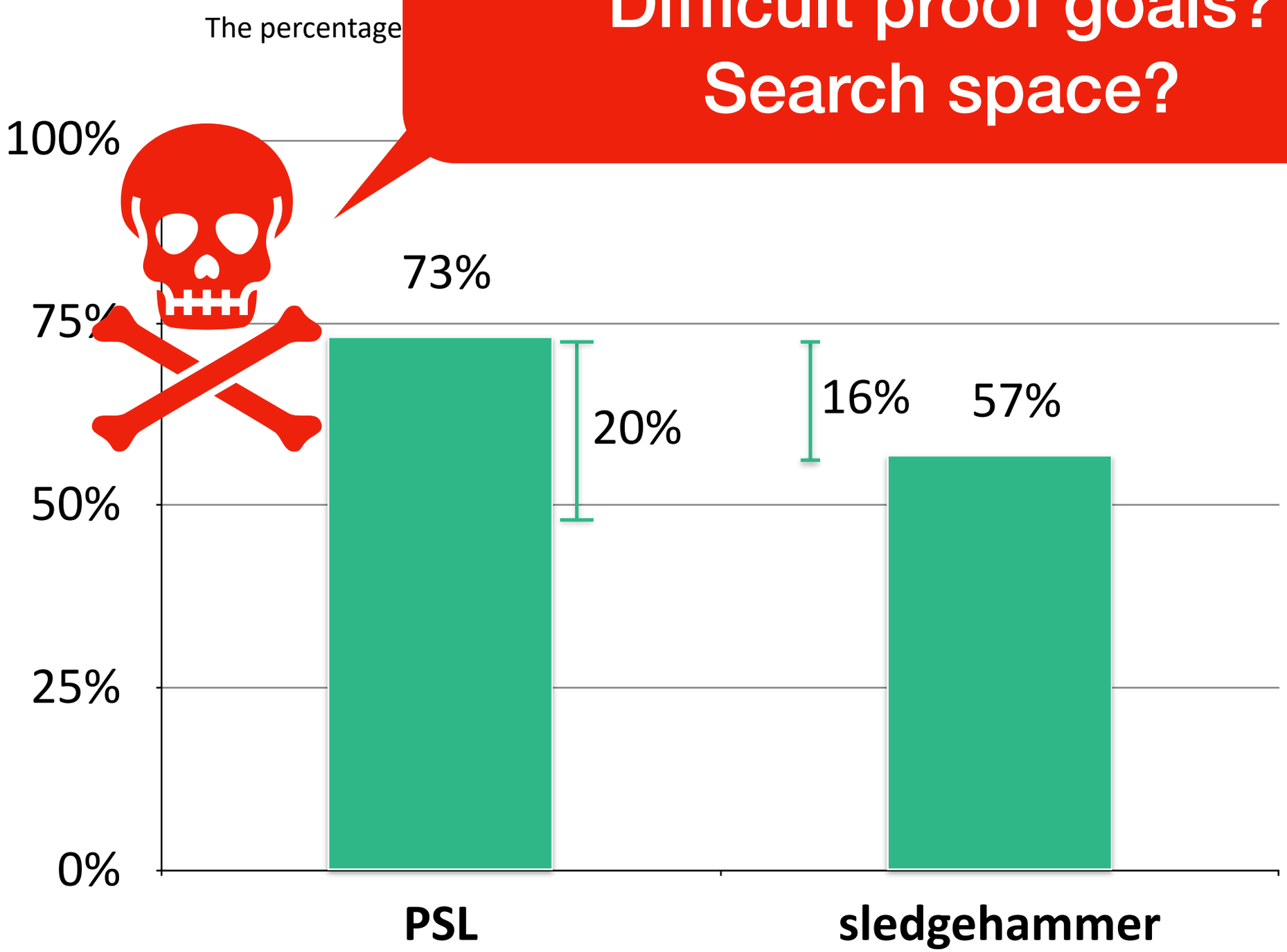


PSL vs sledgehammer



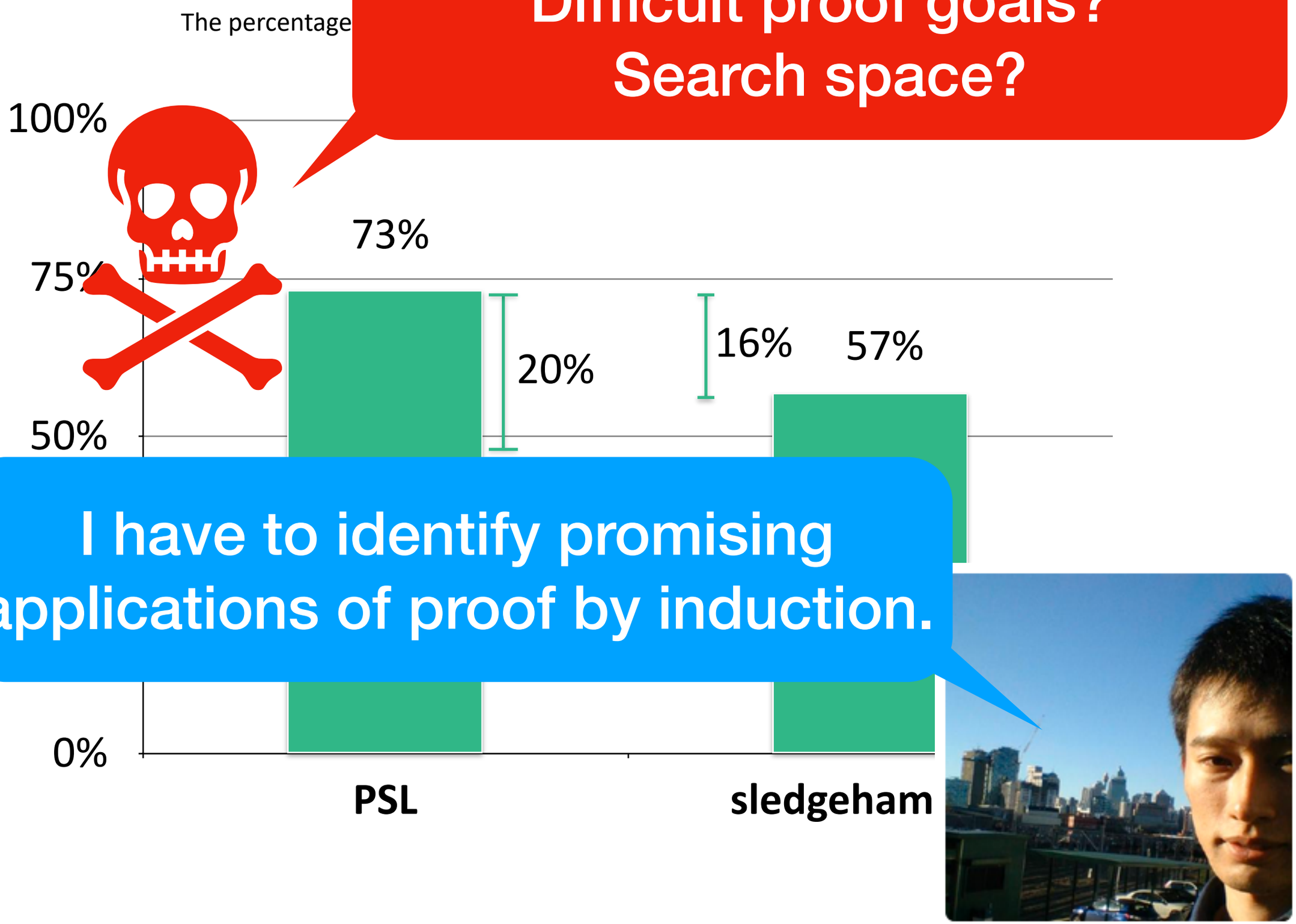
PSL vs sledgehammer

Difficult proof goals?
Search space?



PSL vs sledgehammer

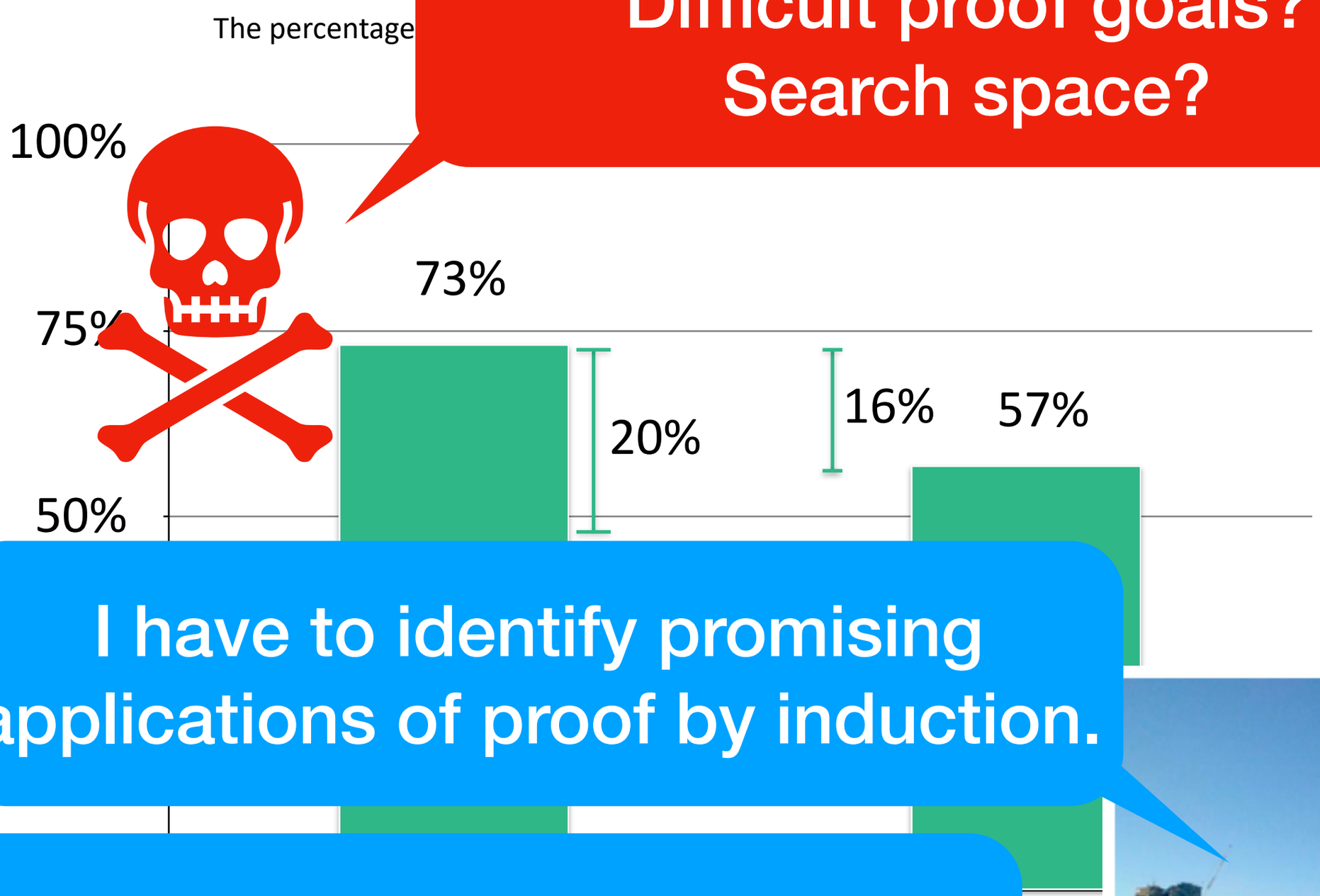
Difficult proof goals?
Search space?



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

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*