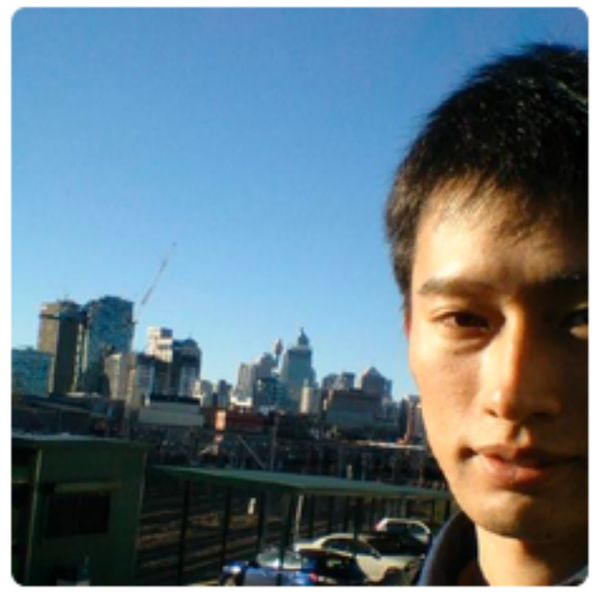
https://github.com/data61/PSL/blob/master/slide/2020_ETH.pdf

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

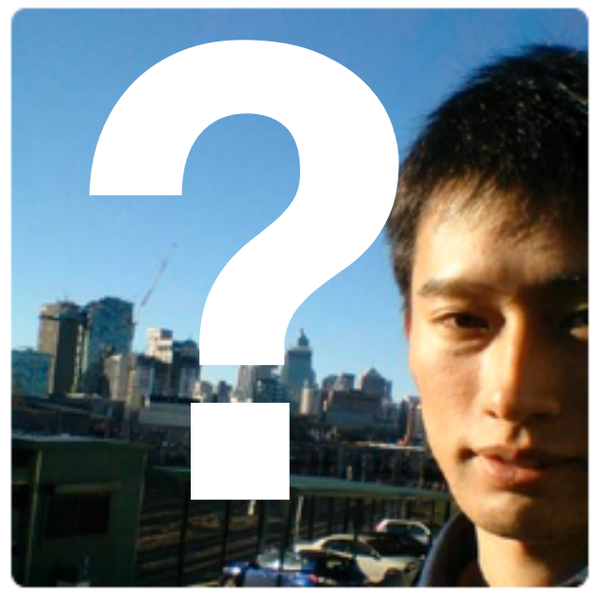
https://github.com/data61/PSL/blob/master/slide/2020_ETH.pdf

Automating proof by induction in Isabelle/HOL using DSLs

Yutaka Nagashima

universität innsbruck

CZECH INSTITUTE OF INFORMATICS ROBOTICS AND CYBERNETICS CTU IN PRAGUE



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

Background

2013 ~ 2017

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

with Prof. Gerwin Klein

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

Background

2013 ~ 2017

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

Isabelle HOL

(seL4)

Security. Performance. Proof.

Cogent

with Prof. Gerwin Klein

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

Background

2013 ~ 2017

Intern & Engineer

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

with Prof. Gerwin Klein

Cogent

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

Background

2013 ~ 2017

Intern & Engineer

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

with Prof. Gerwin Klein

Cogent

PhD in AI for theorem proving

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

Background

2013 ~ 2017

Intern & Engineer

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

Cogent

with Prof. Gerwin Klein

PhD in AI for theorem proving

2017 ~ 2018

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

with Prof. Cezary Kaliszyk

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

Background

2013 ~ 2017

Intern & Engineer

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

with Prof. Gerwin Klein

Cogent

PhD in AI for theorem proving

2017 ~ 2018

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

with Prof. Cezary Kaliszyk

PSL

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

Background

2013 ~ 2017

Intern & Engineer

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

with Prof. Gerwin Klein

Cogent

PhD in AI for theorem proving

2017 ~ 2018

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

with Prof. Cezary Kaliszyk

PSL

PaMpeR

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

Background

2013 ~ 2017

Intern & Engineer

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

with Prof. Gerwin Klein

Cogent

PhD in AI for theorem proving

2017 ~ 2018

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

with Prof. Cezary Kaliszyk

PSL

PaMpeR

2018 ~ 2020/21

http://ai4reason.org/members.html

with Dr. Josef Urban

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

Background

2013 ~ 2017

Intern & Engineer

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

with Prof. Gerwin Klein

Cogent

PhD in AI for theorem proving

2017 ~ 2018

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

with Prof. Cezary Kaliszyk

PSL

PaMpeR

2018 ~ 2020/21

http://ai4reason.org/members.html

with Dr. Josef Urban

LiFtEr

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

Background

2013 ~ 2017

Intern & Engineer

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

with Prof. Gerwin Klein

Cogent

PhD in AI for theorem proving

2017 ~ 2018

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

with Prof. Cezary Kaliszyk

PSL

PaMpeR

2018 ~ 2020/21

http://ai4reason.org/members.html

with Dr. Josef Urban

LiFtEr

smart_induct

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

Background

2013 ~ 2017

Intern & Engineer

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

with Prof. Gerwin Klein

Cogent

PhD in AI for theorem proving

2017 ~ 2018

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

with Prof. Cezary Kaliszyk

PSL

PaMpeR

2018 ~ 2020/21

http://ai4reason.org/members.html

with Dr. Josef Urban

LiFtEr

smart_induct

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

PSL: Proof Strategy Language

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

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

PSL: Proof Strategy Language

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

DEMO1

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

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

12,29 (251/963) (isabelle,isabelle,UTF-8-Isabelle) 1:17 AM

DEMO1

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

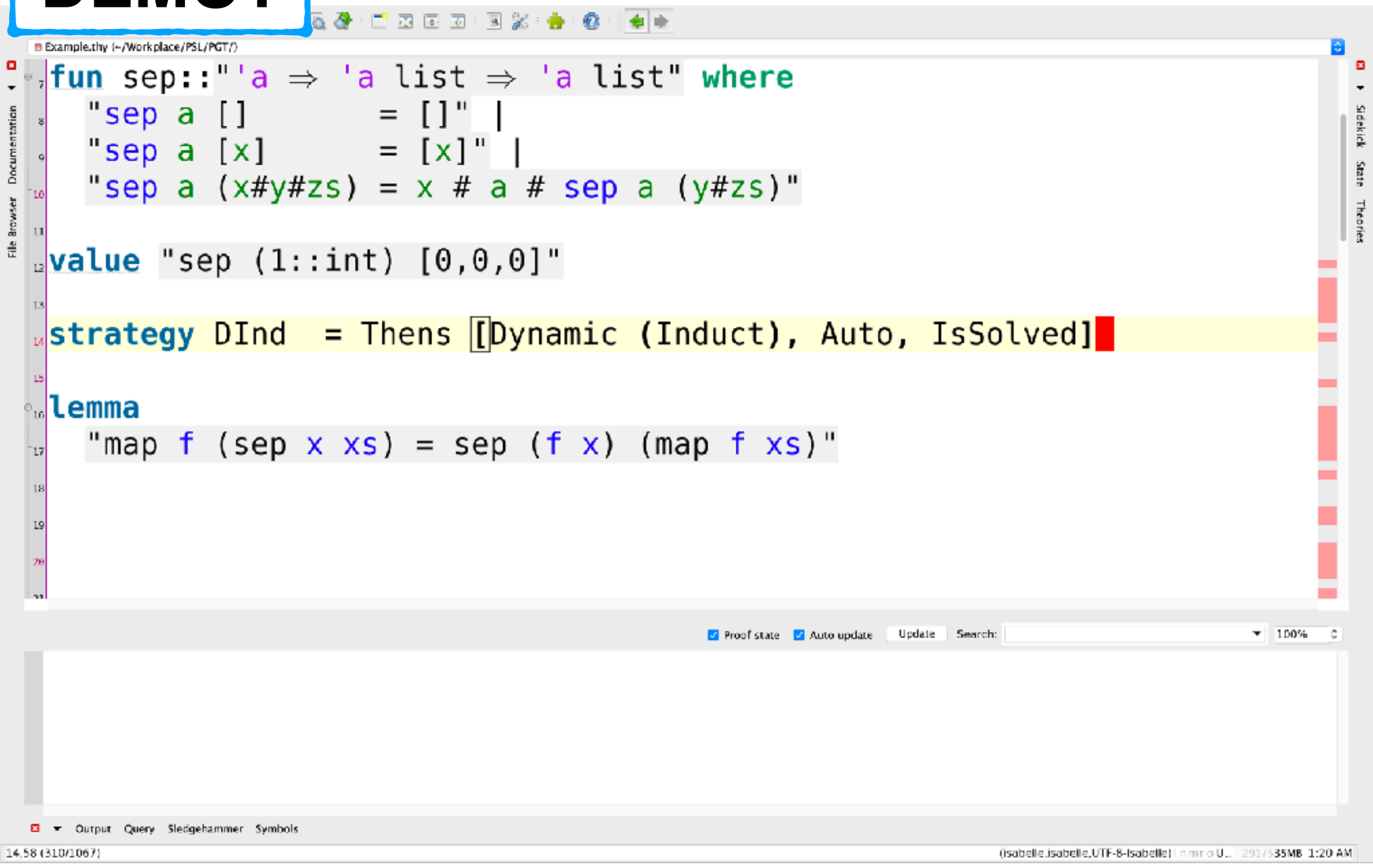
```
proof (prove)
goal (1 subgoal):
 1. map f (sep x xs) = sep (f x) (map f xs)
```

Output | Query | Sledgehammer | Symbols

17,44 (304/1012) | Input/output complete | (isabelle,isabelle,UTF-8-Isabelle) | 241/535MB 1:17 AM

DEMO1

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



DEMO1

thub.com/data61/PSL/blob/master/slide/2020_ETH.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) 242/535MB 1:05 AM

DEMO1

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

thub.com/data61/PSL/blob/master/slide/2020_ETH.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) 255/535MB 1:05 AM

DEMO1

thub.com/data61/PSL/blob/master/slide/2020_ETH.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) (isabelle,isabelle,UTF-8-Isabelle) 255/535MB 1:05 AM

DEMO1

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

PSL: Proof Strategy Language

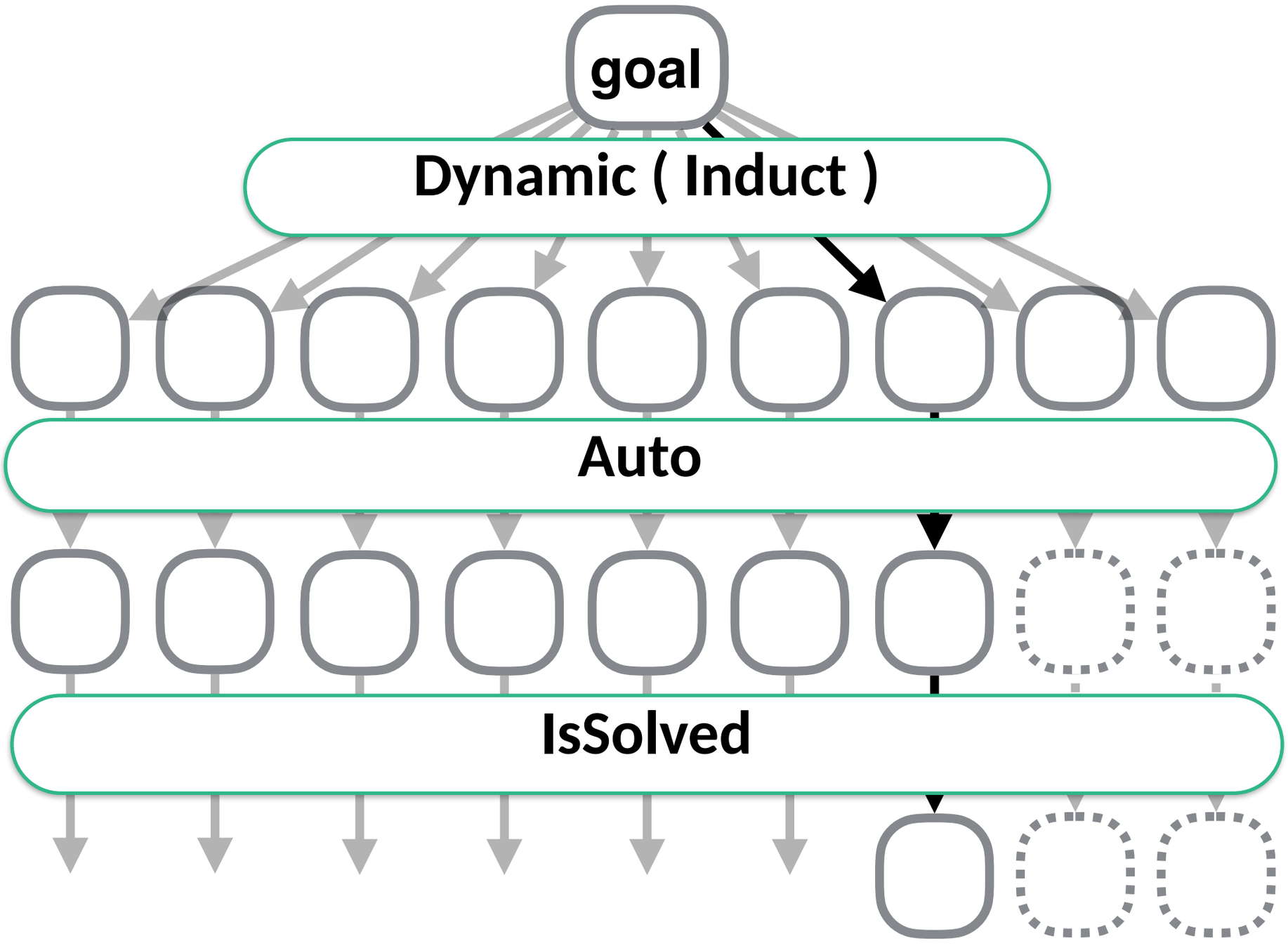
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```

DEMO1

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

PSL: Proof Strategy Language

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

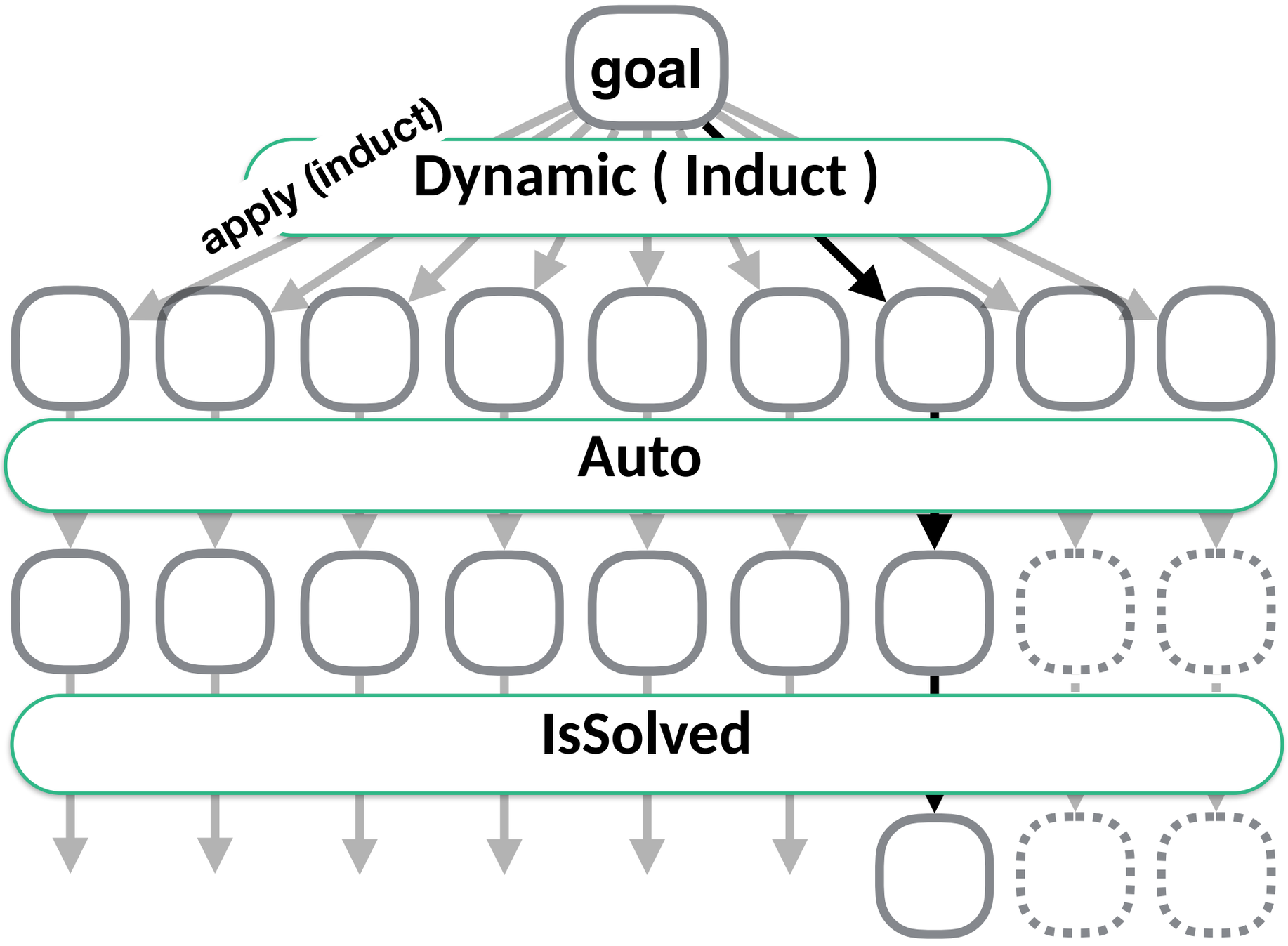


DEMO1

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

PSL: Proof Strategy Language

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

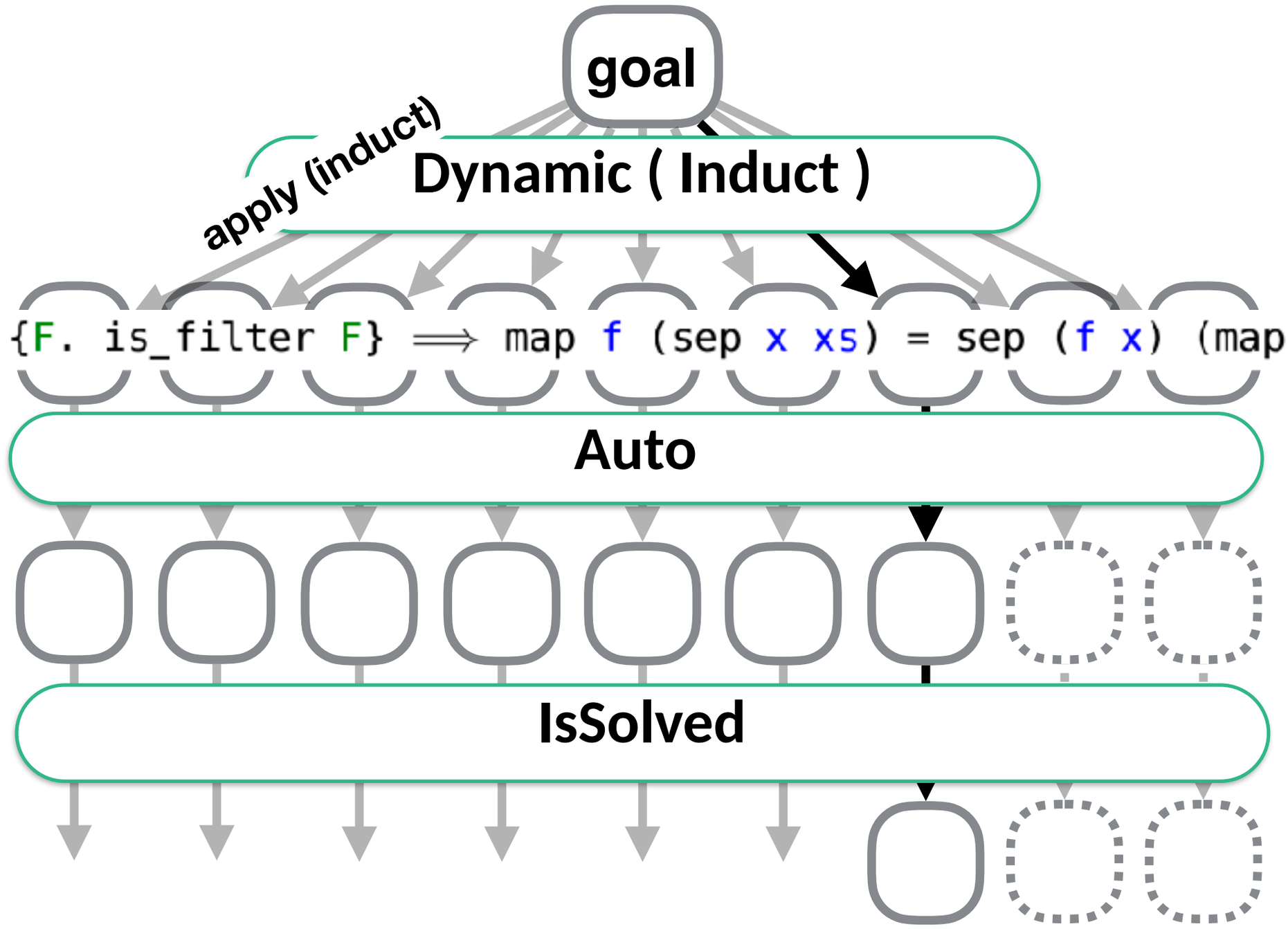


DEMO1

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

PSL: Proof Strategy Language

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



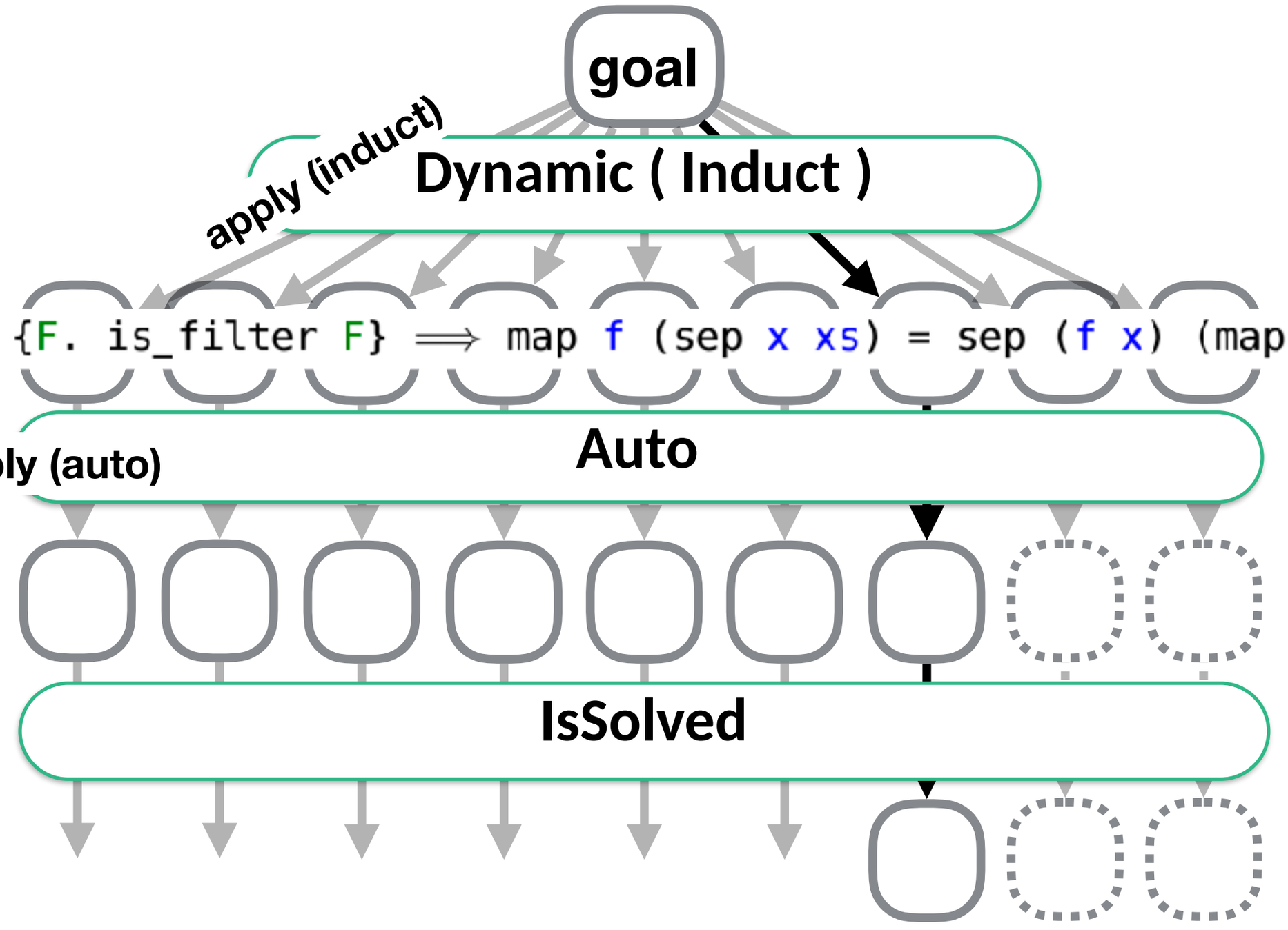

```
⋀y. y ∈ {F. is_filter F} ⟹ map f (sep x xs) = sep (f x) (map f xs)
```

DEMO1

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

PSL: Proof Strategy Language

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

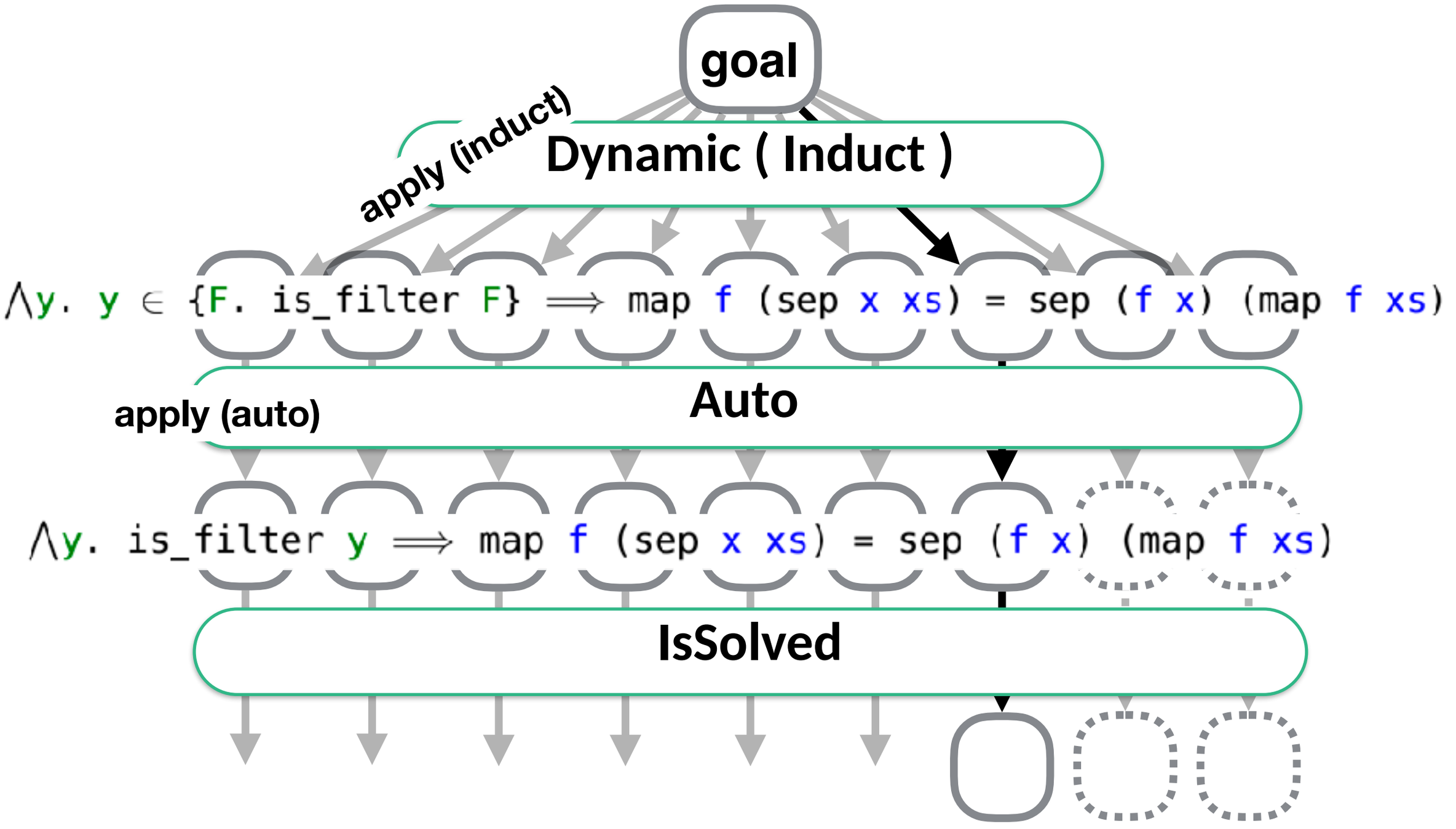


DEMO1

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

PSL: Proof Strategy Language

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

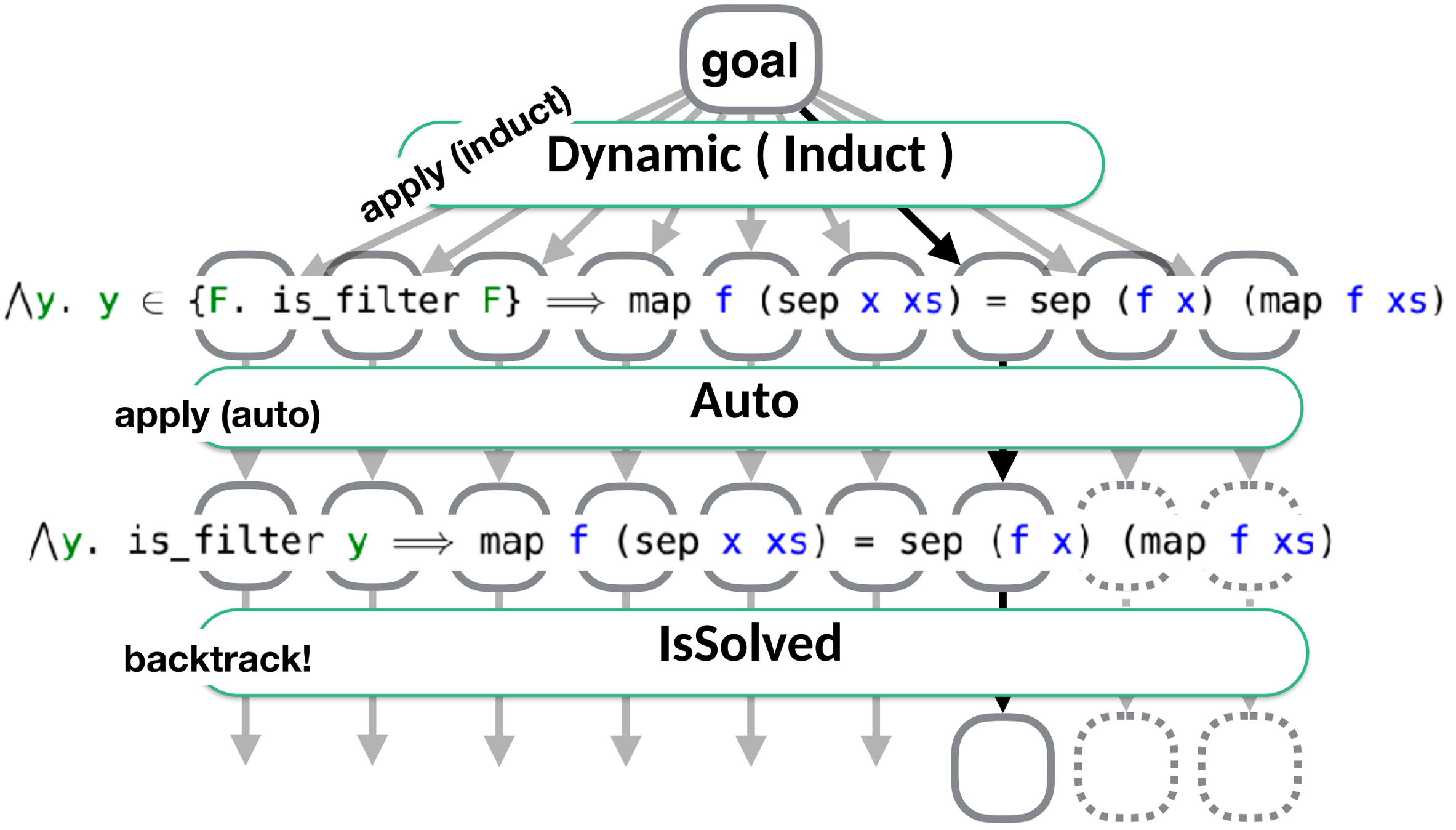


DEMO1

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

PSL: Proof Strategy Language

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

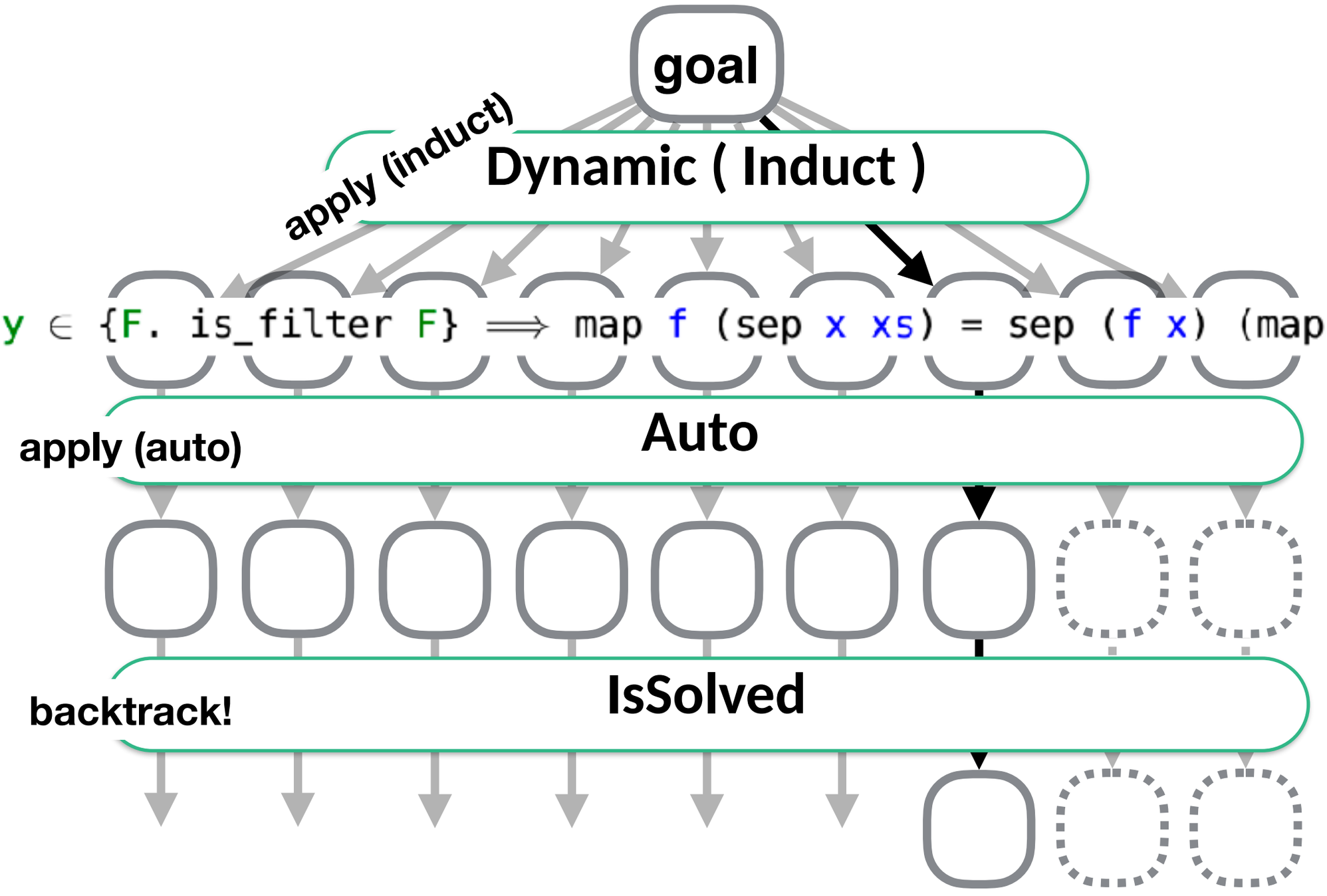


DEMO1

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

PSL: Proof Strategy Language

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

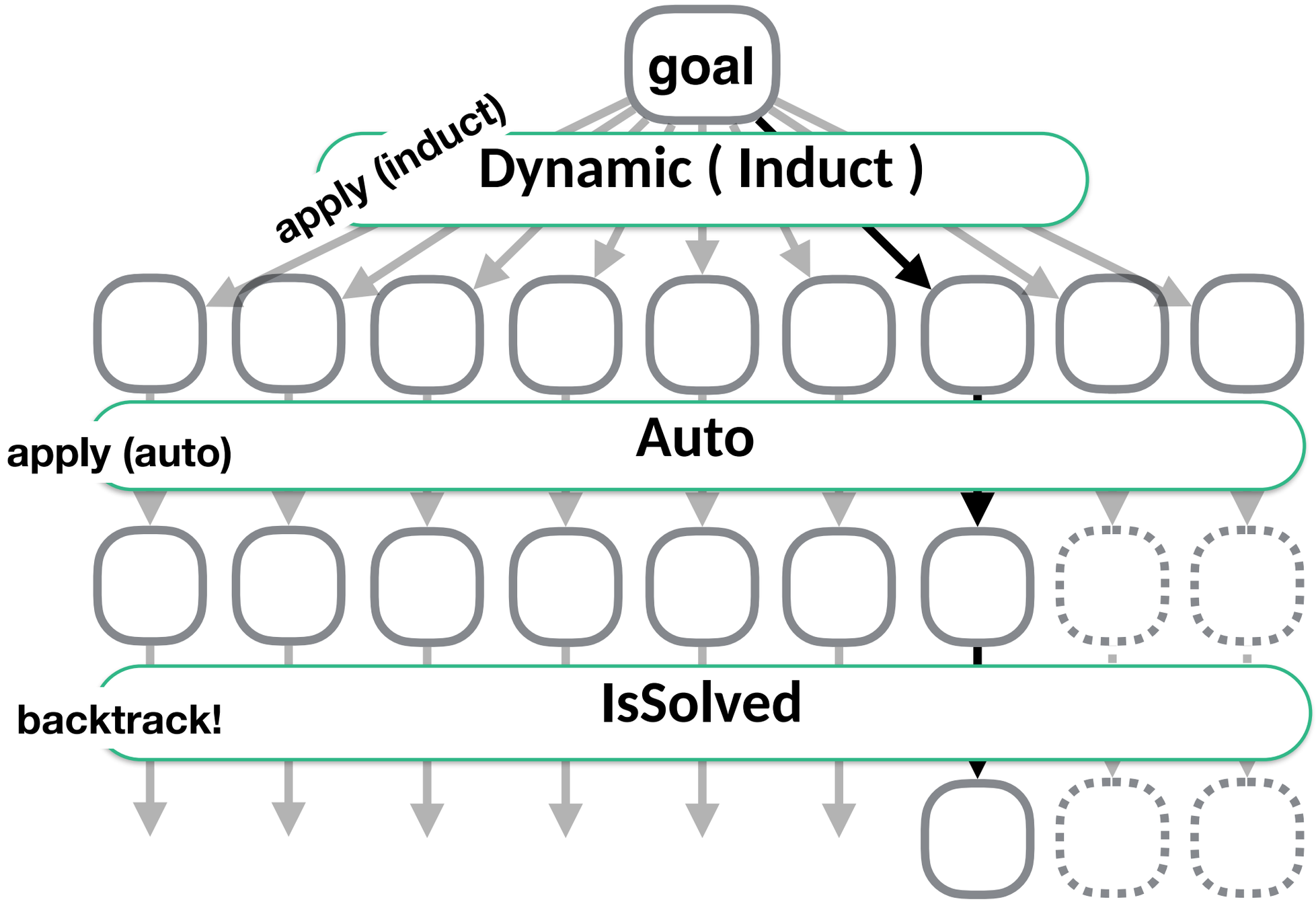


DEMO1

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

PSL: Proof Strategy Language

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

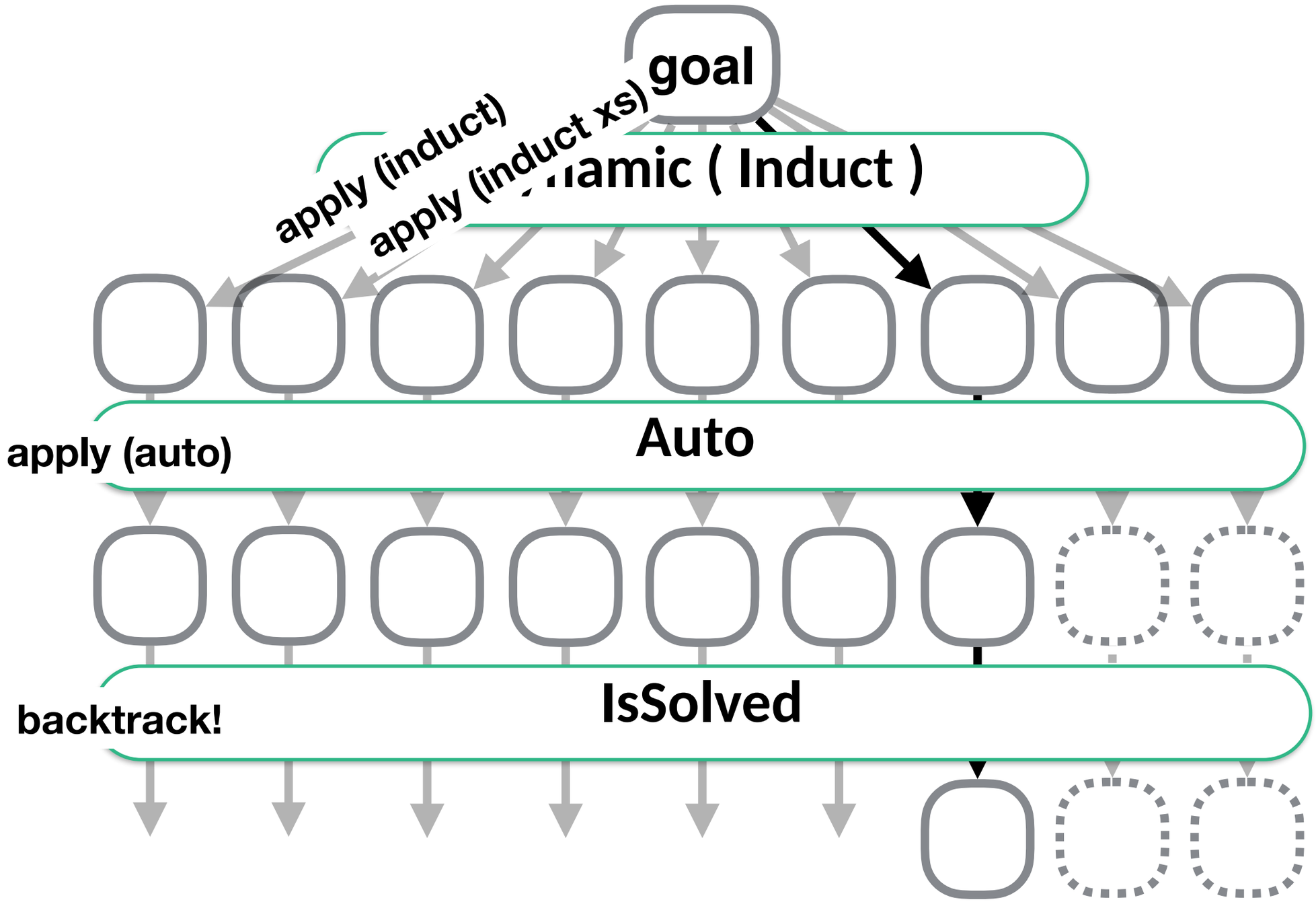


DEMO1

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

PSL: Proof Strategy Language

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

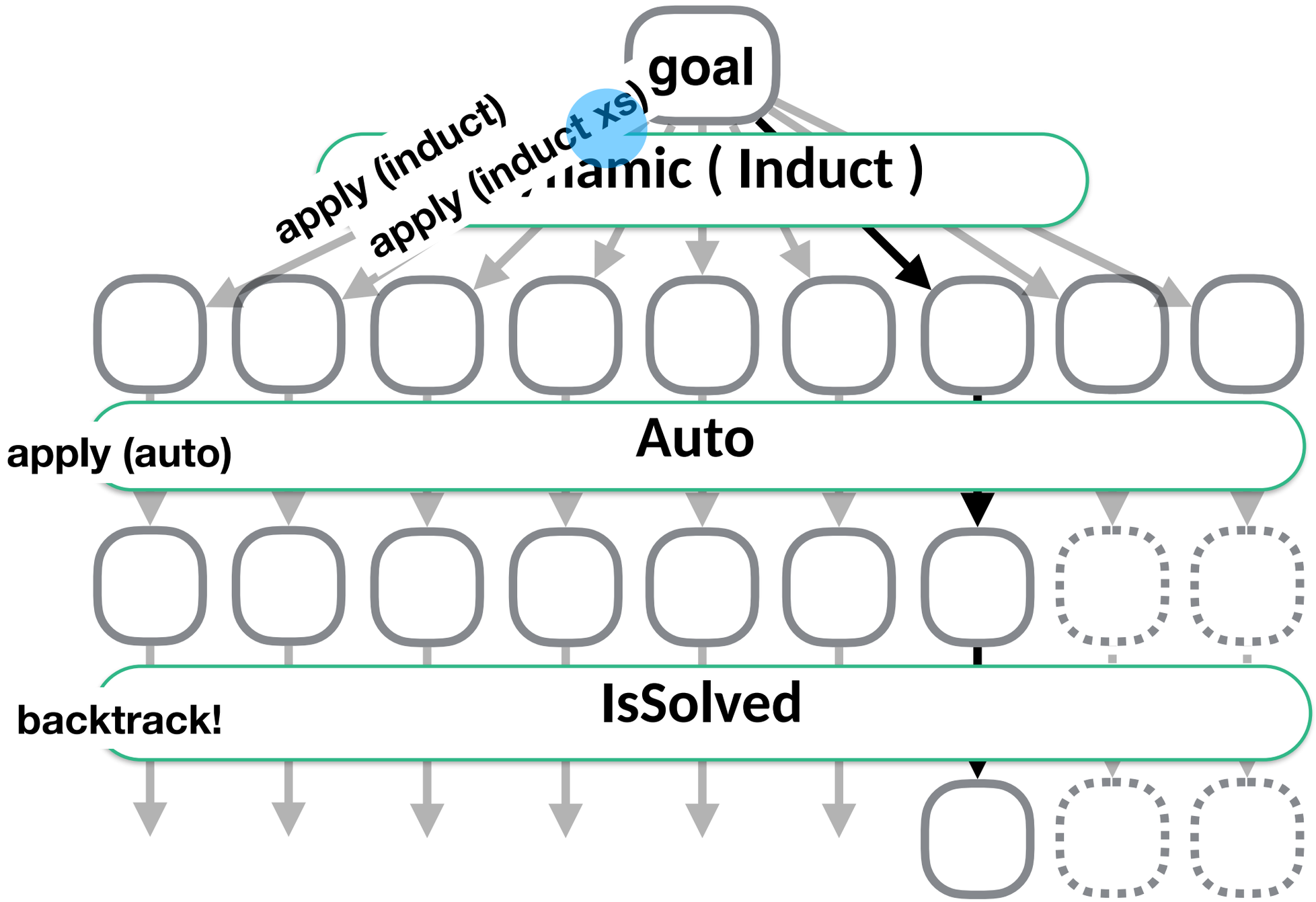


DEMO1

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

PSL: Proof Strategy Language

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

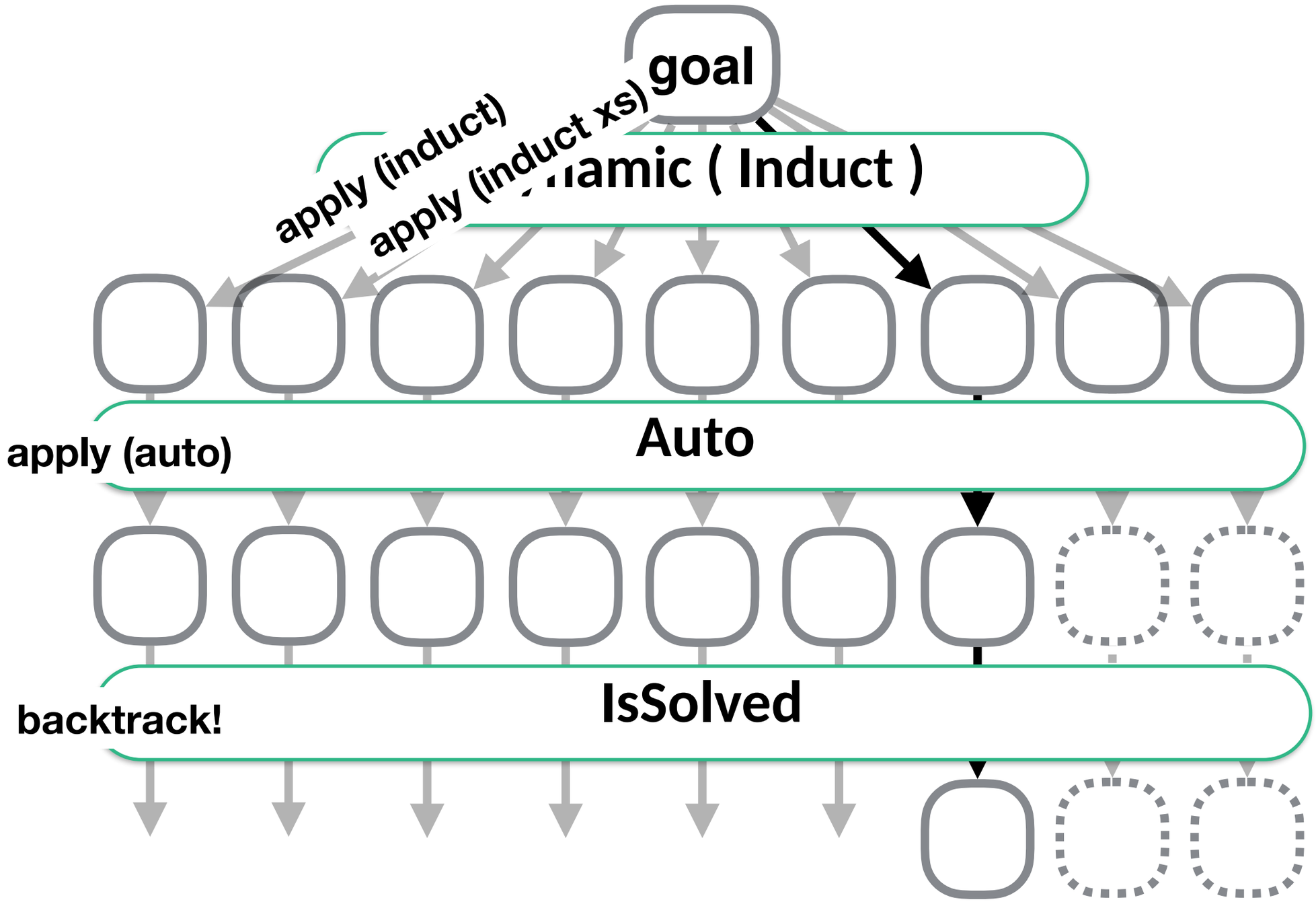


DEMO1

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

PSL: Proof Strategy Language

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

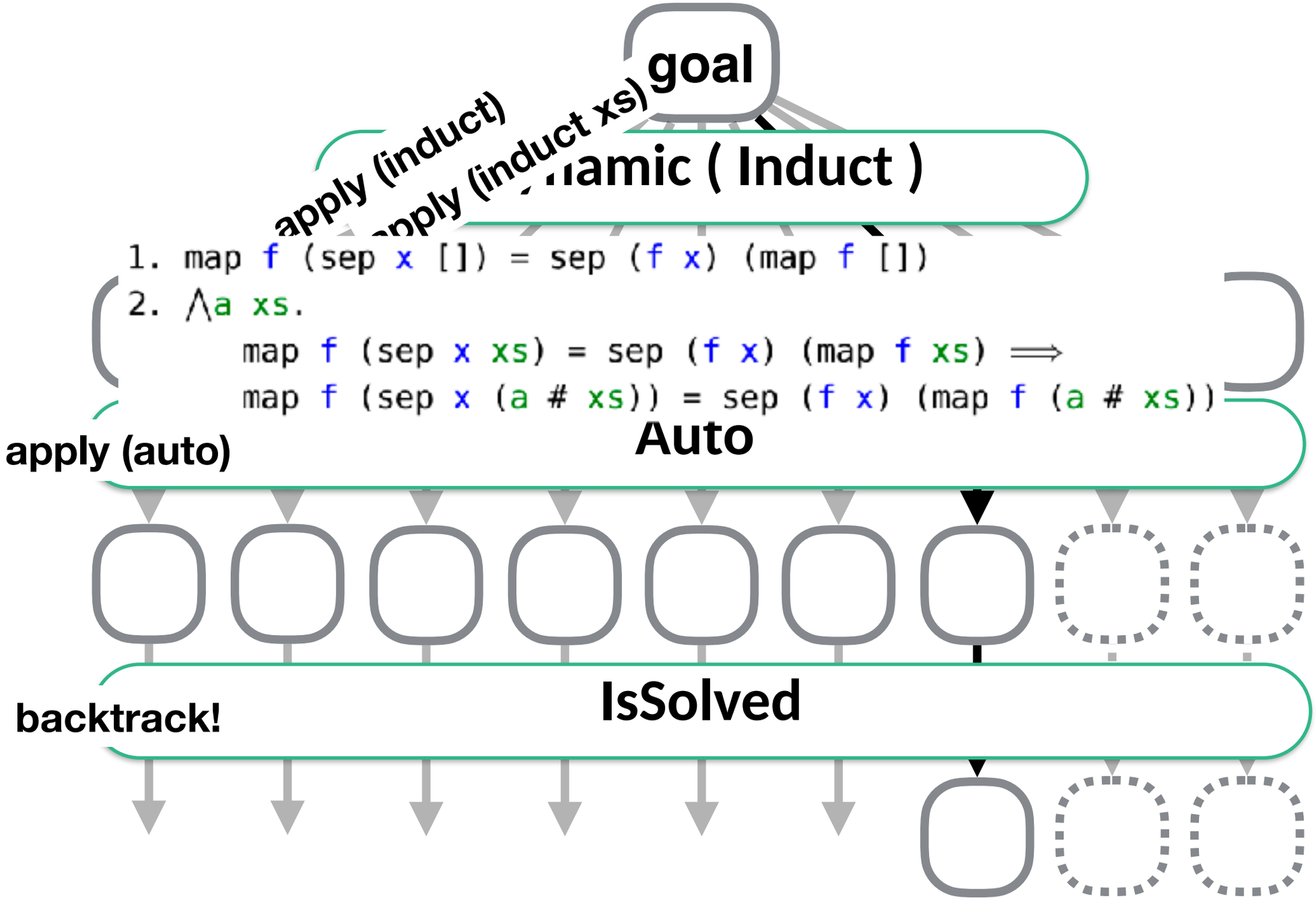


DEMO1

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

PSL: Proof Strategy Language

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

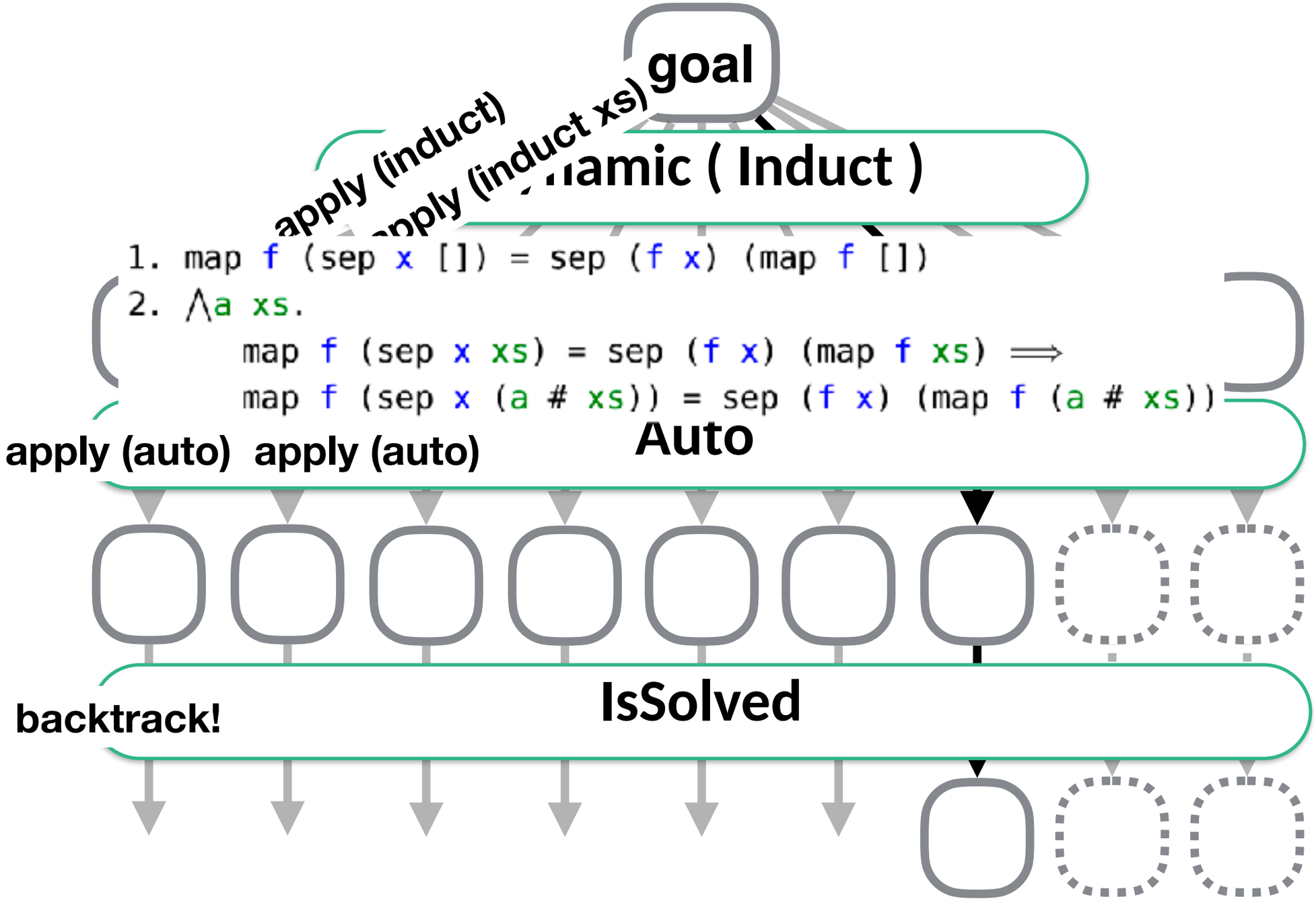


DEMO1

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

PSL: Proof Strategy Language

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

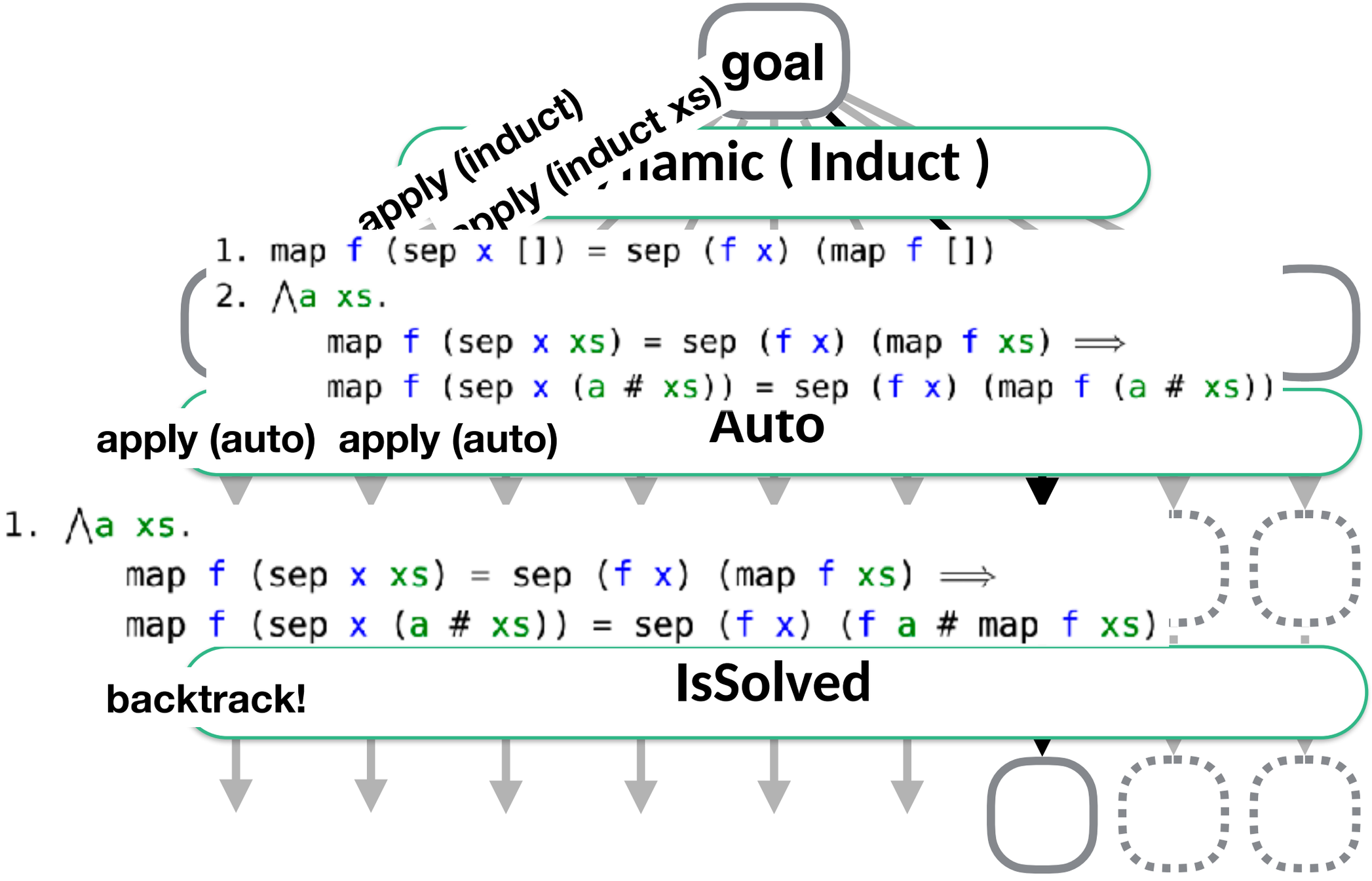


DEMO1

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

PSL: Proof Strategy Language

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

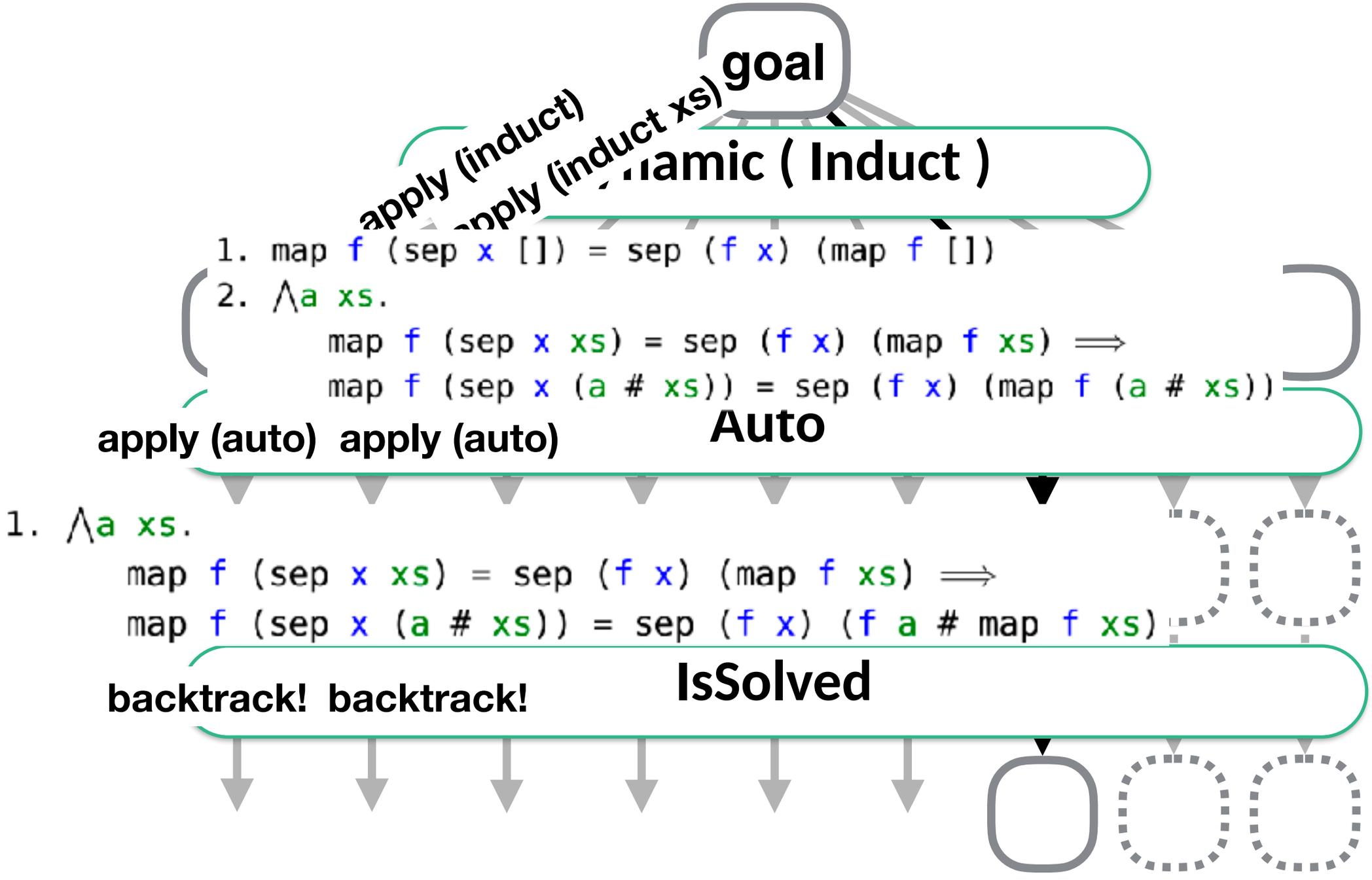


DEMO1

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

PSL: Proof Strategy Language

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

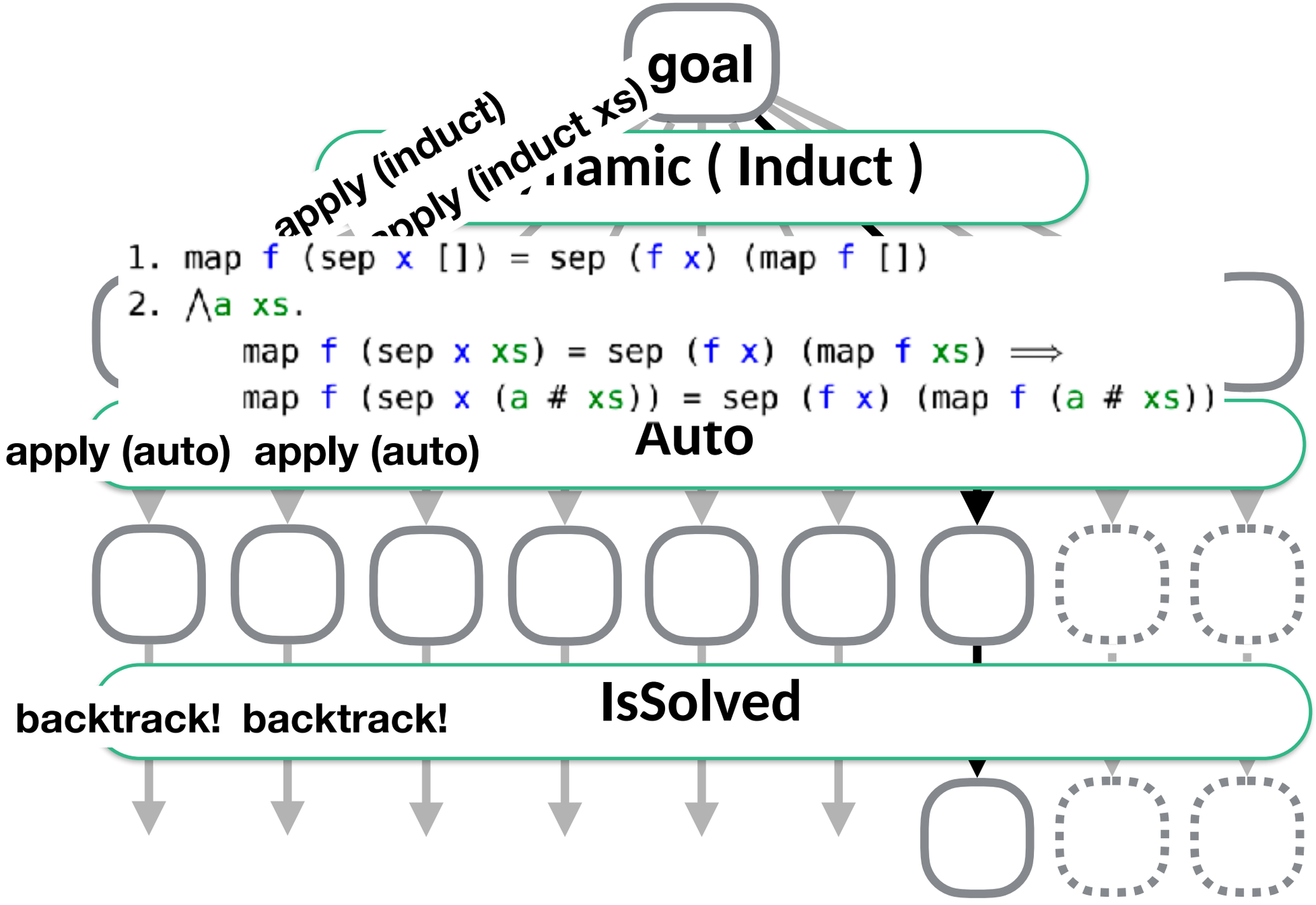


DEMO1

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

PSL: Proof Strategy Language

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

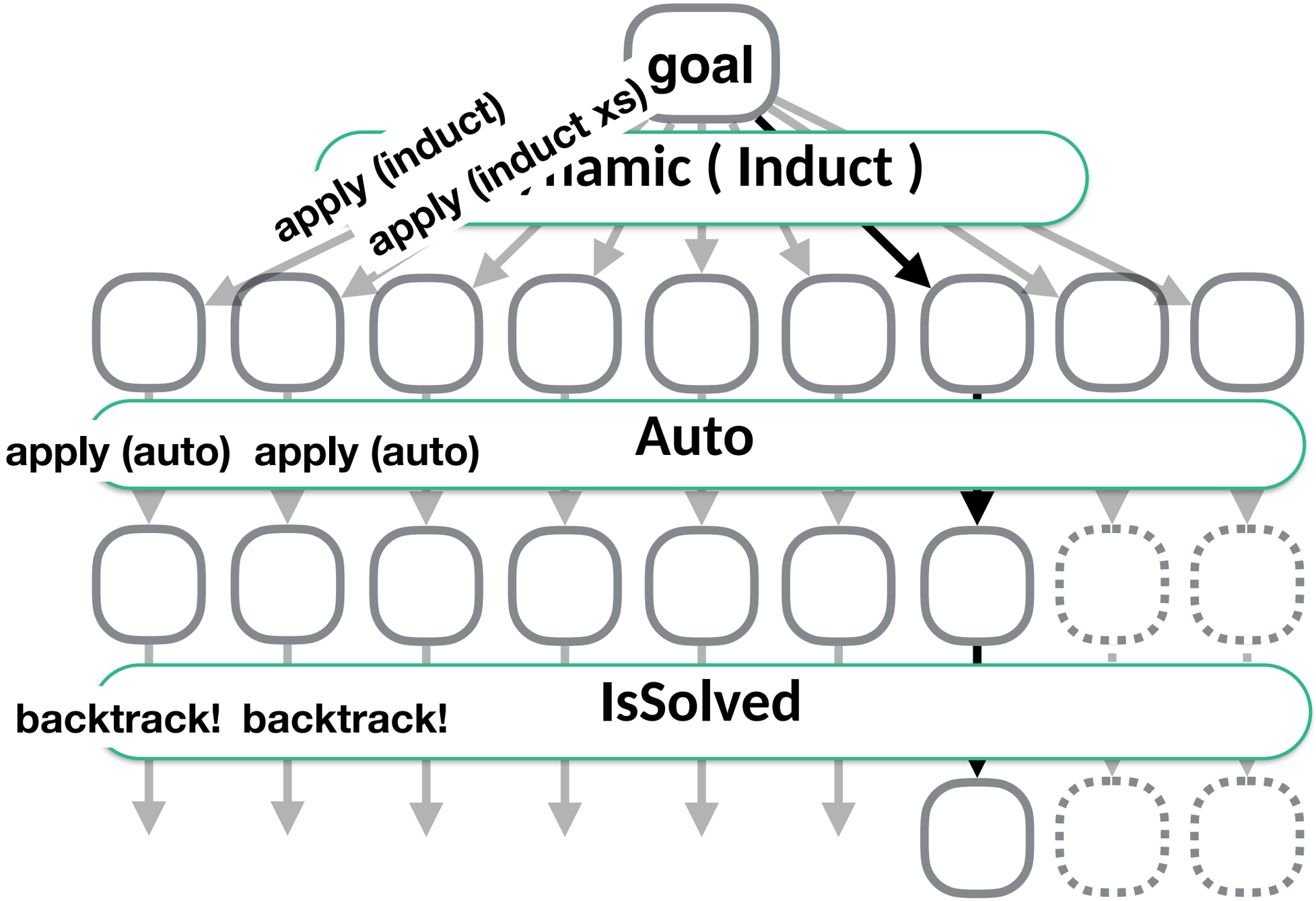


DEMO1

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

PSL: Proof Strategy Language

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

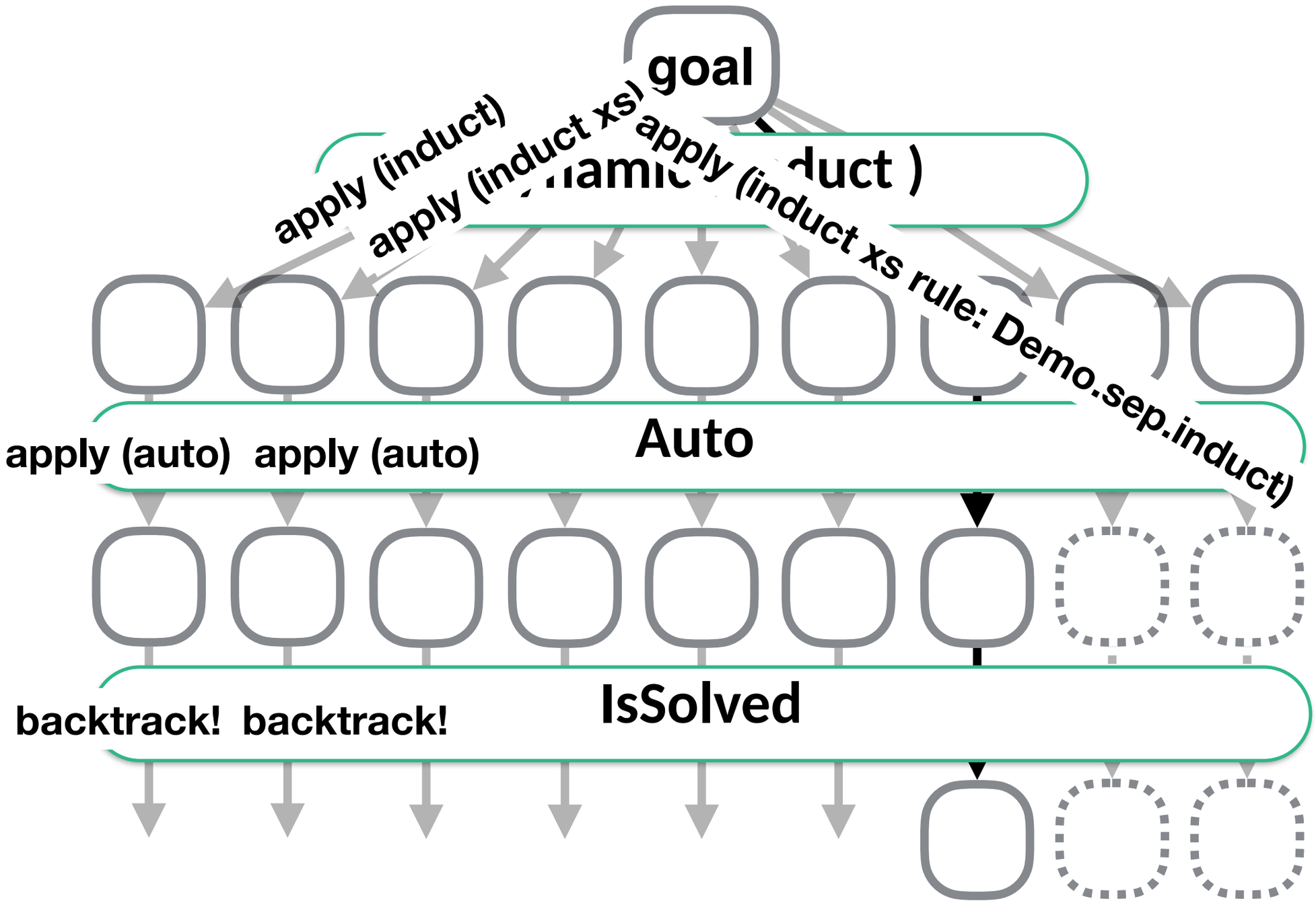


DEMO1

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

PSL: Proof Strategy Language

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

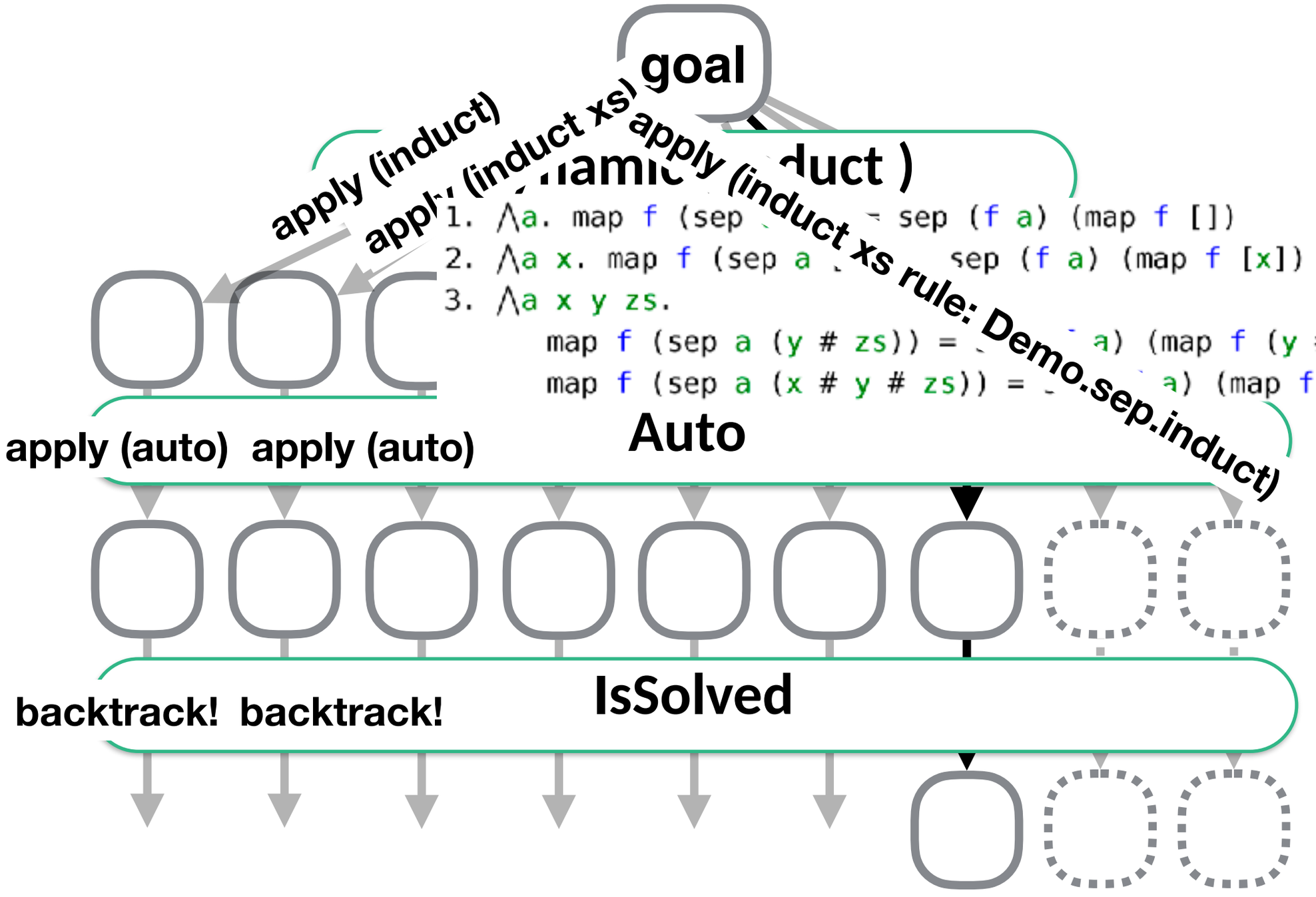


DEMO1

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

PSL: Proof Strategy Language

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

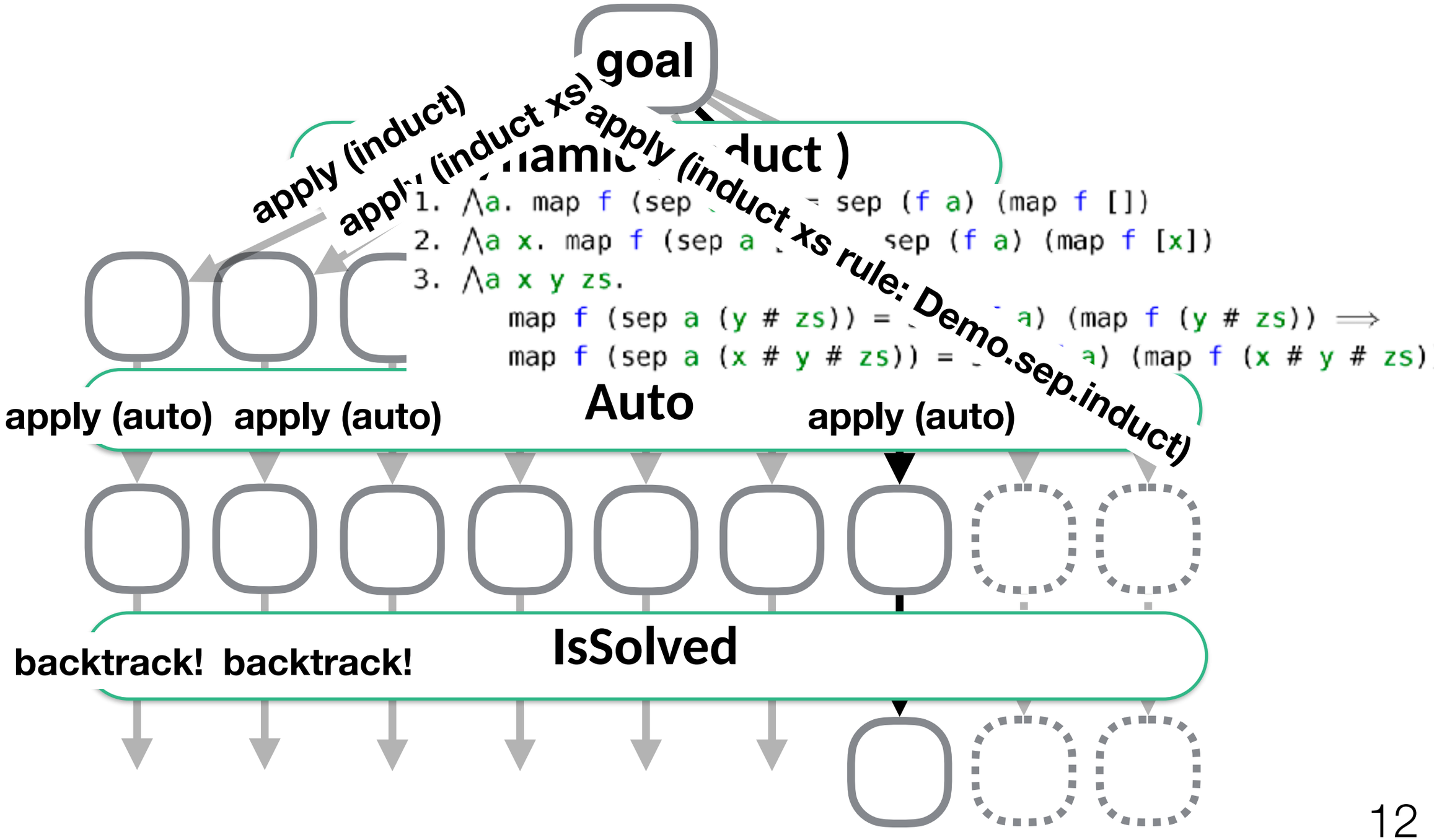


DEMO1

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

PSL: Proof Strategy Language

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

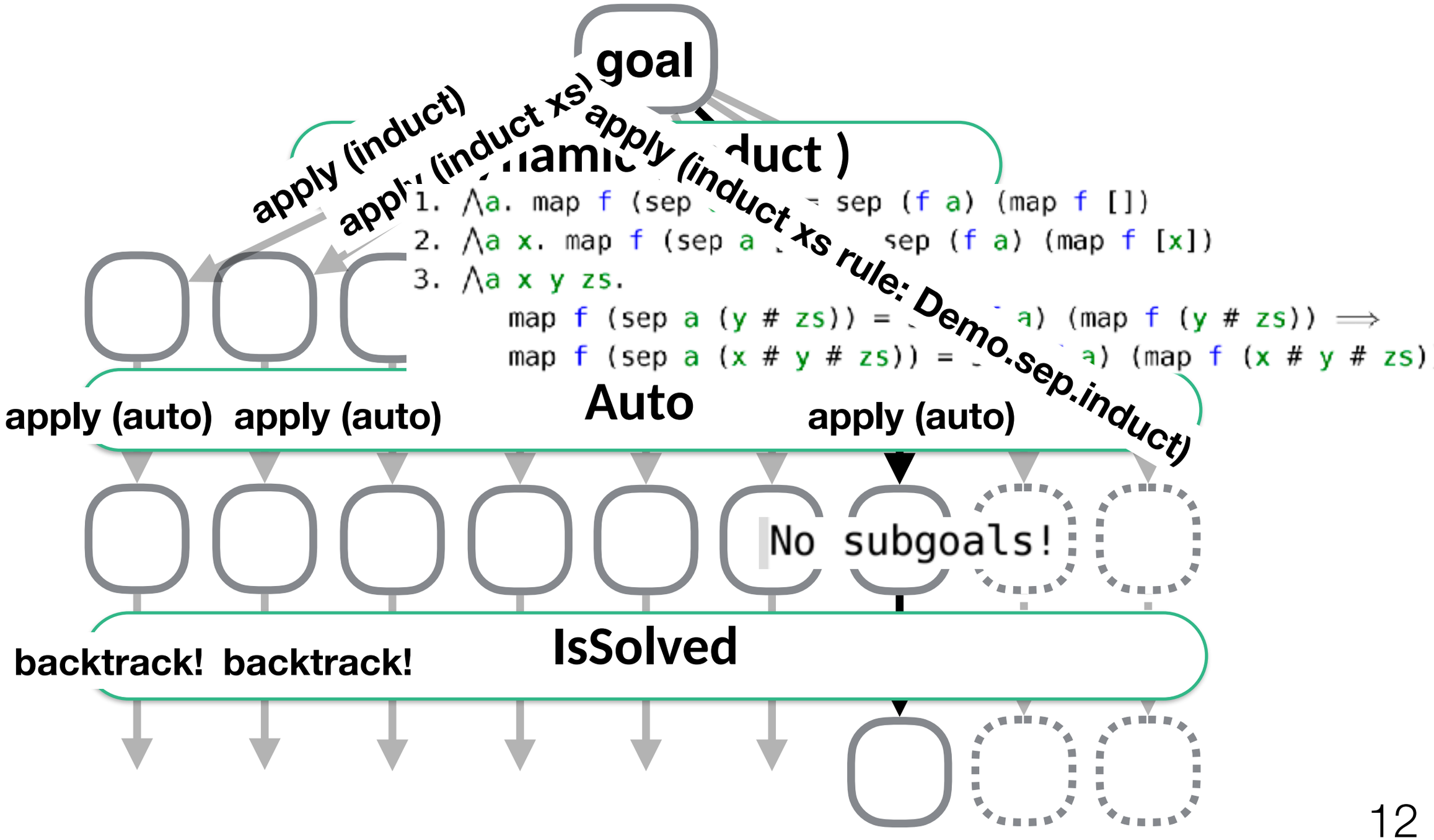


DEMO1

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

PSL: Proof Strategy Language

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

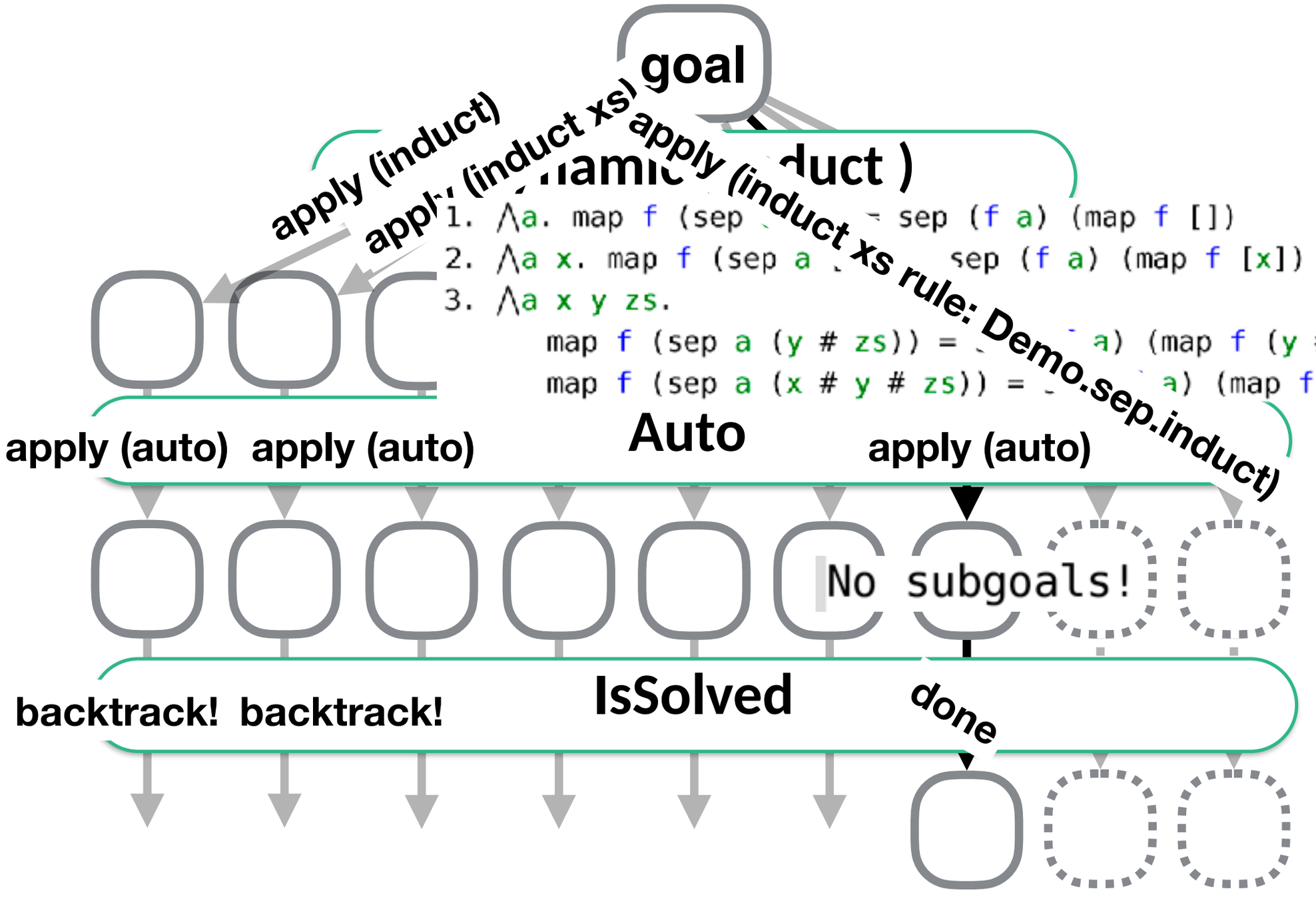


DEMO1

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

PSL: Proof Strategy Language

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

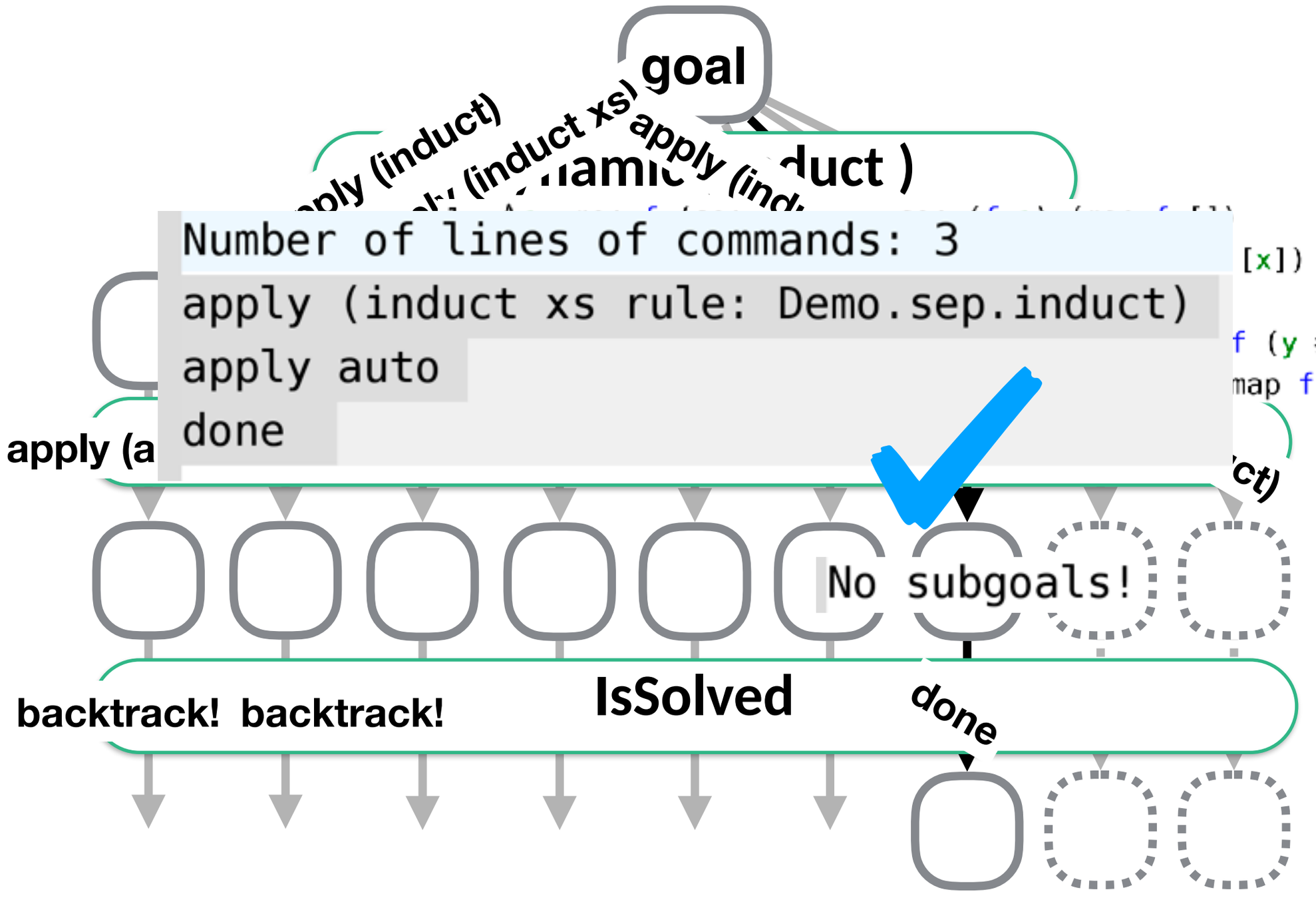


DEMO1

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

PSL: Proof Strategy Language

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

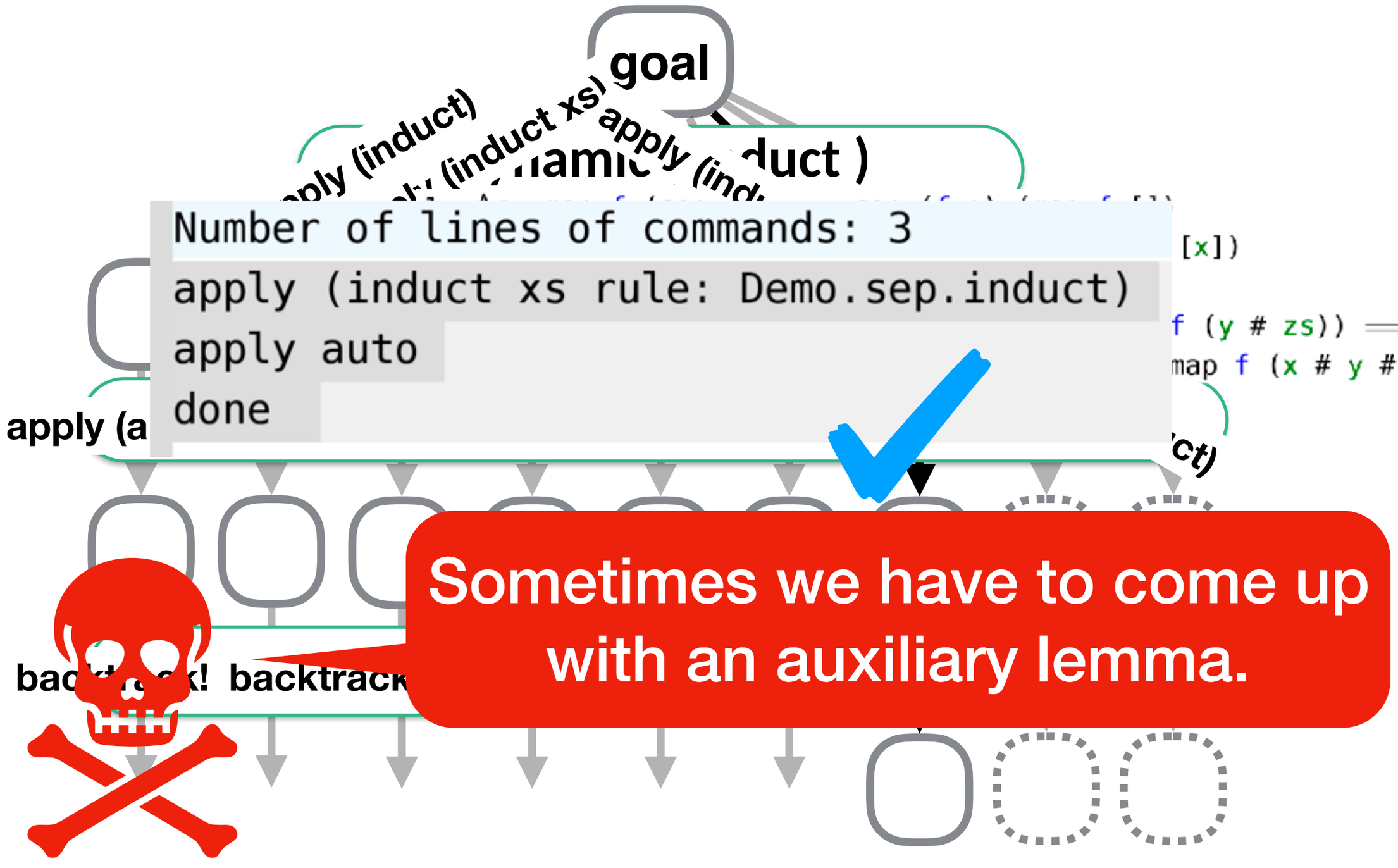


DEMO1

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

PSL: Proof Strategy Language

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



DEMO2

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

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

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

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



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

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

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

Output | Query | Sledgehammer | Symbols

43,29 (837/1004)

DEMO2

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

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

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

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



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


```

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

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

Output | Query | Sledgehammer | Symbols

44,18 (855/1004)

DEMO2

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

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

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

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


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

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

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

Output | Query | Sledgehammer | Symbols

45,27 (988/1000)

DEMO2

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

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

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

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


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

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

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

Output | Query | Sledgehammer | Symbols

45,27 (988/1000)

DEMO2

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

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

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

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

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

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

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

Output | Query | Sledgehammer | Symbols

45,27 (988/1000)

DEMO2

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

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

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

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

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

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

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

What happened?

Output | Query | Sledgehammer | Symbols

45,27 (988/1000)

```
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]
```

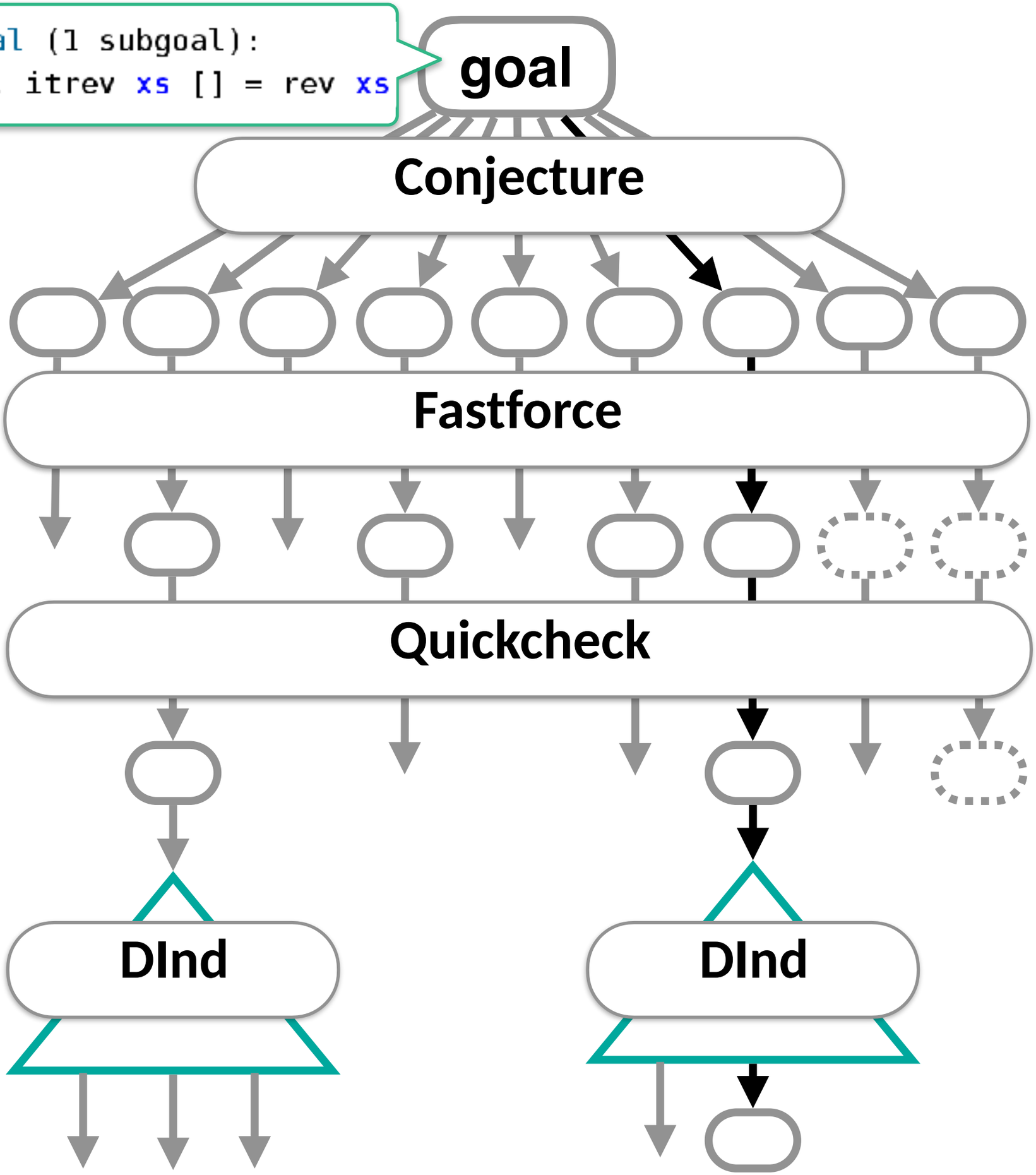
Quickcheck

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

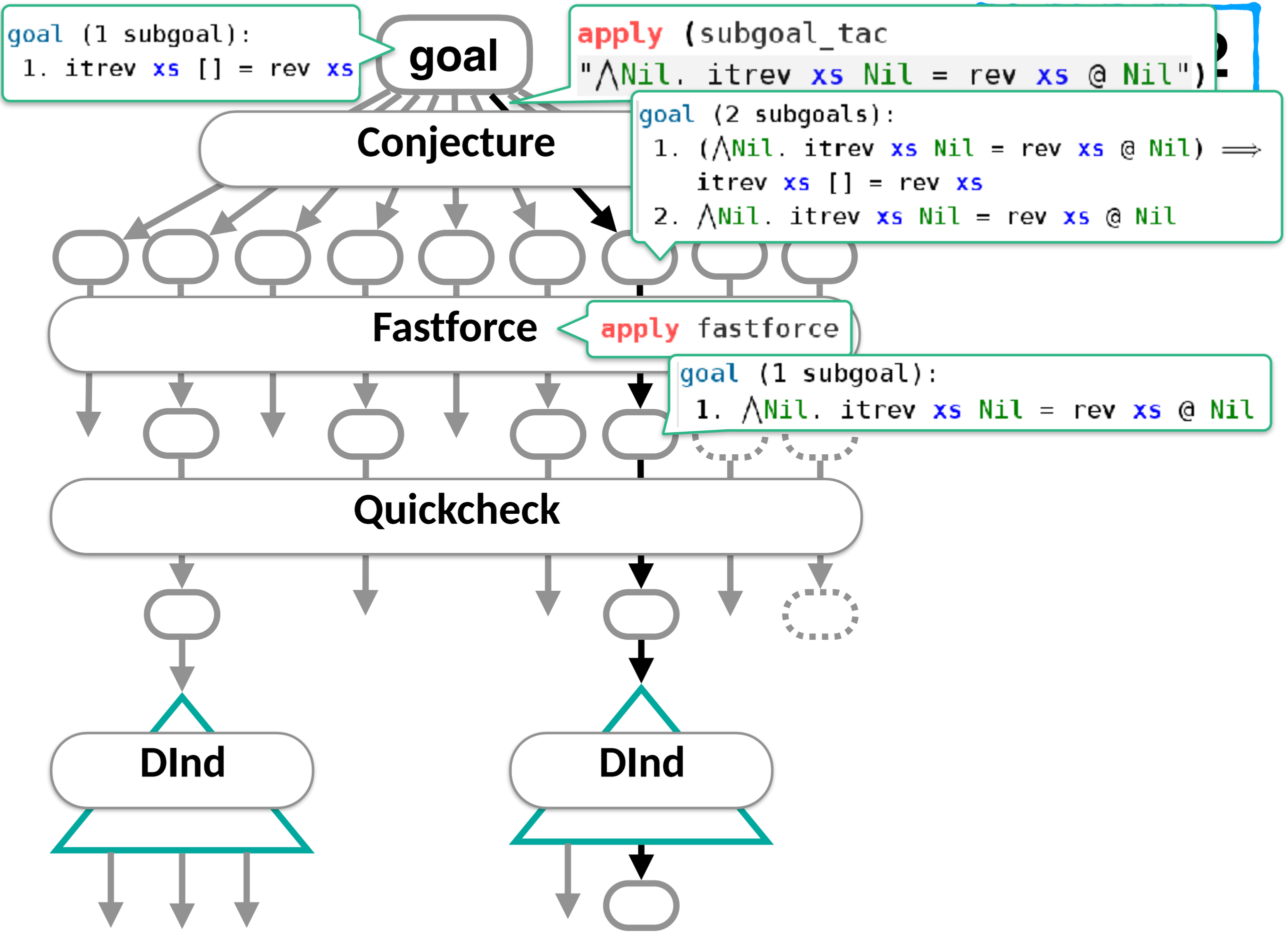
Fastforce

```
apply fastforce
```

Quickcheck

DInd

DInd


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

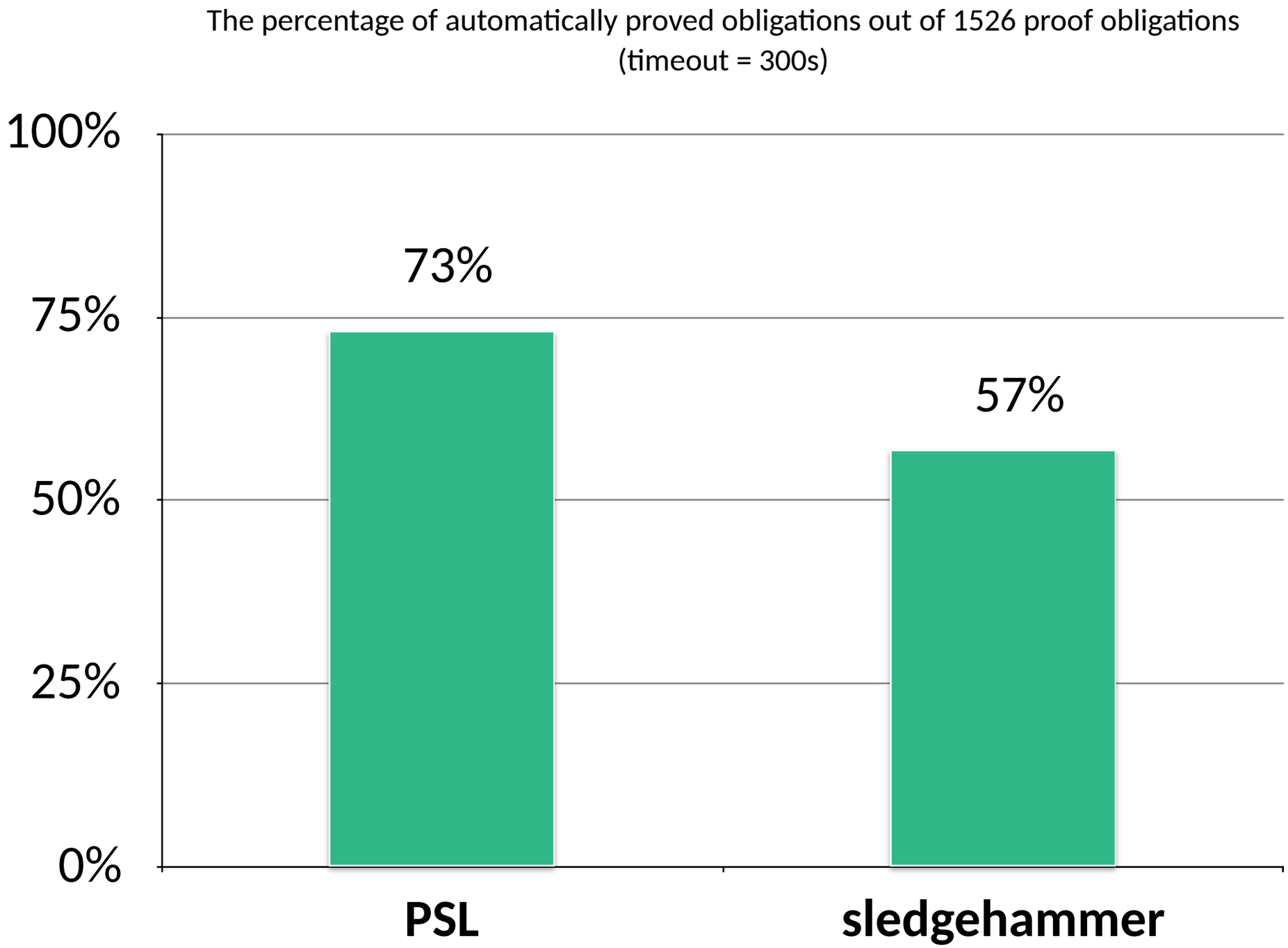
DInd

DInd

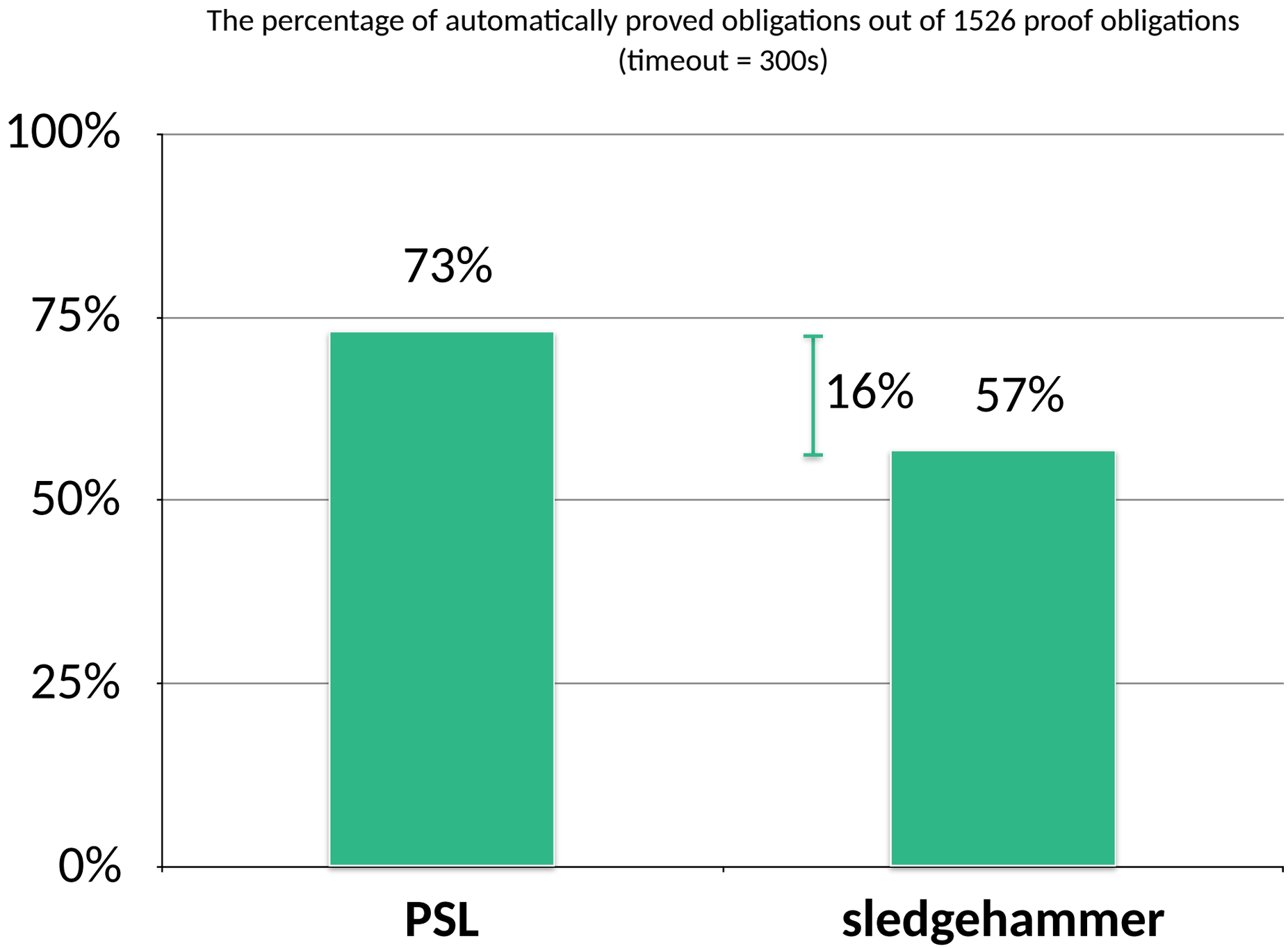
```
apply (induct xs)
apply auto
done
```

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

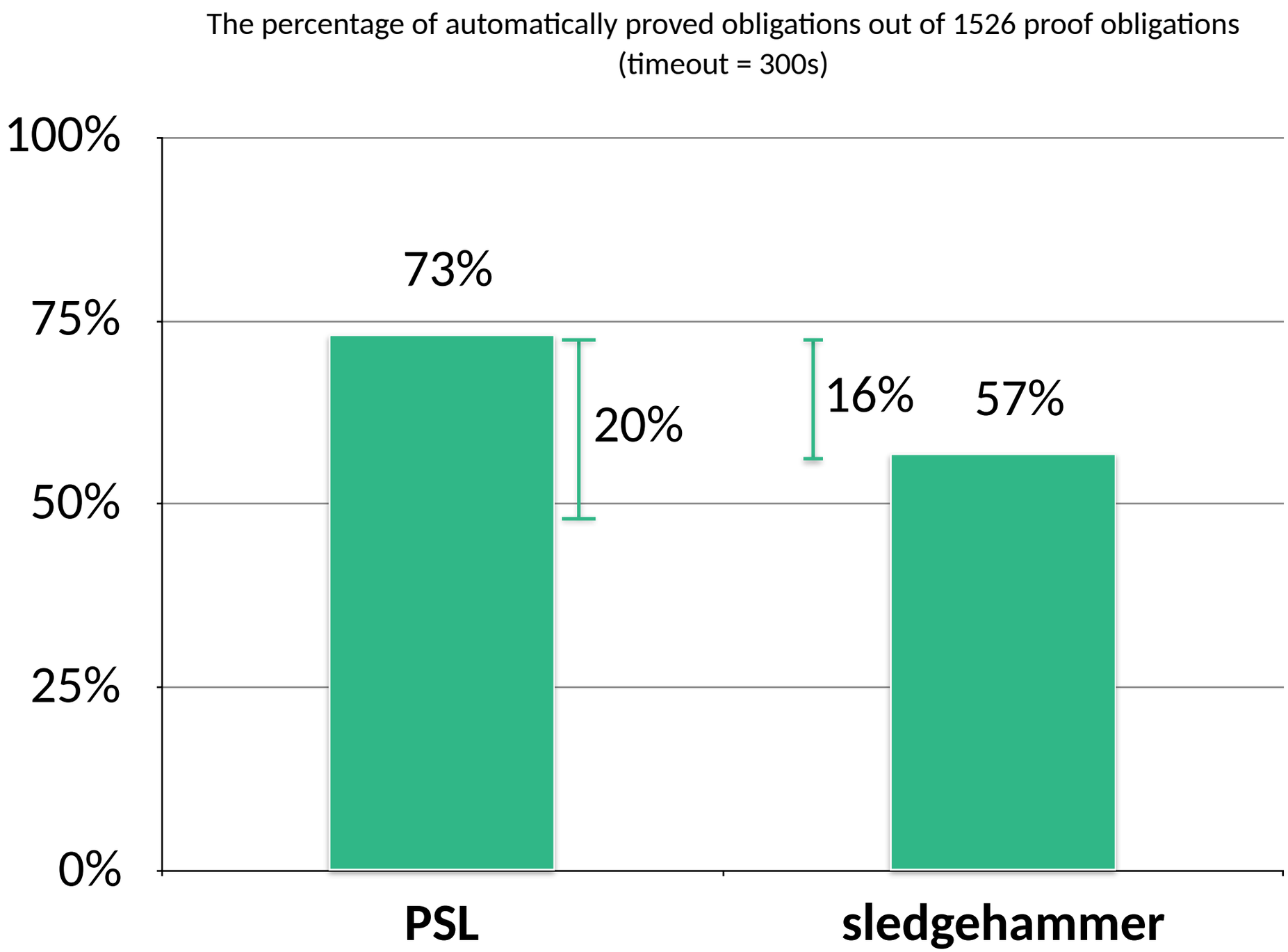
PSL vs sledgehammer



PSL vs sledgehammer

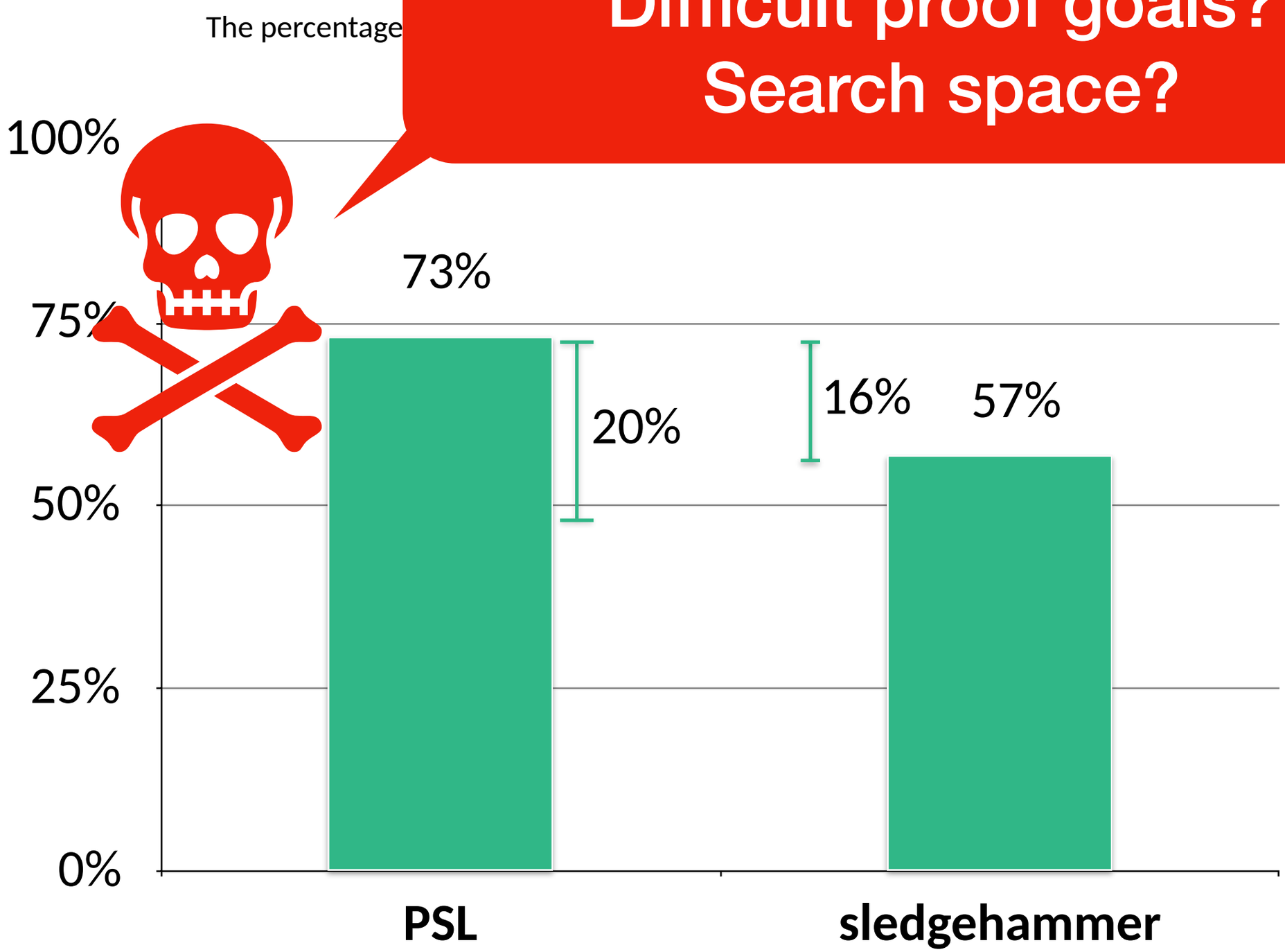


PSL vs sledgehammer



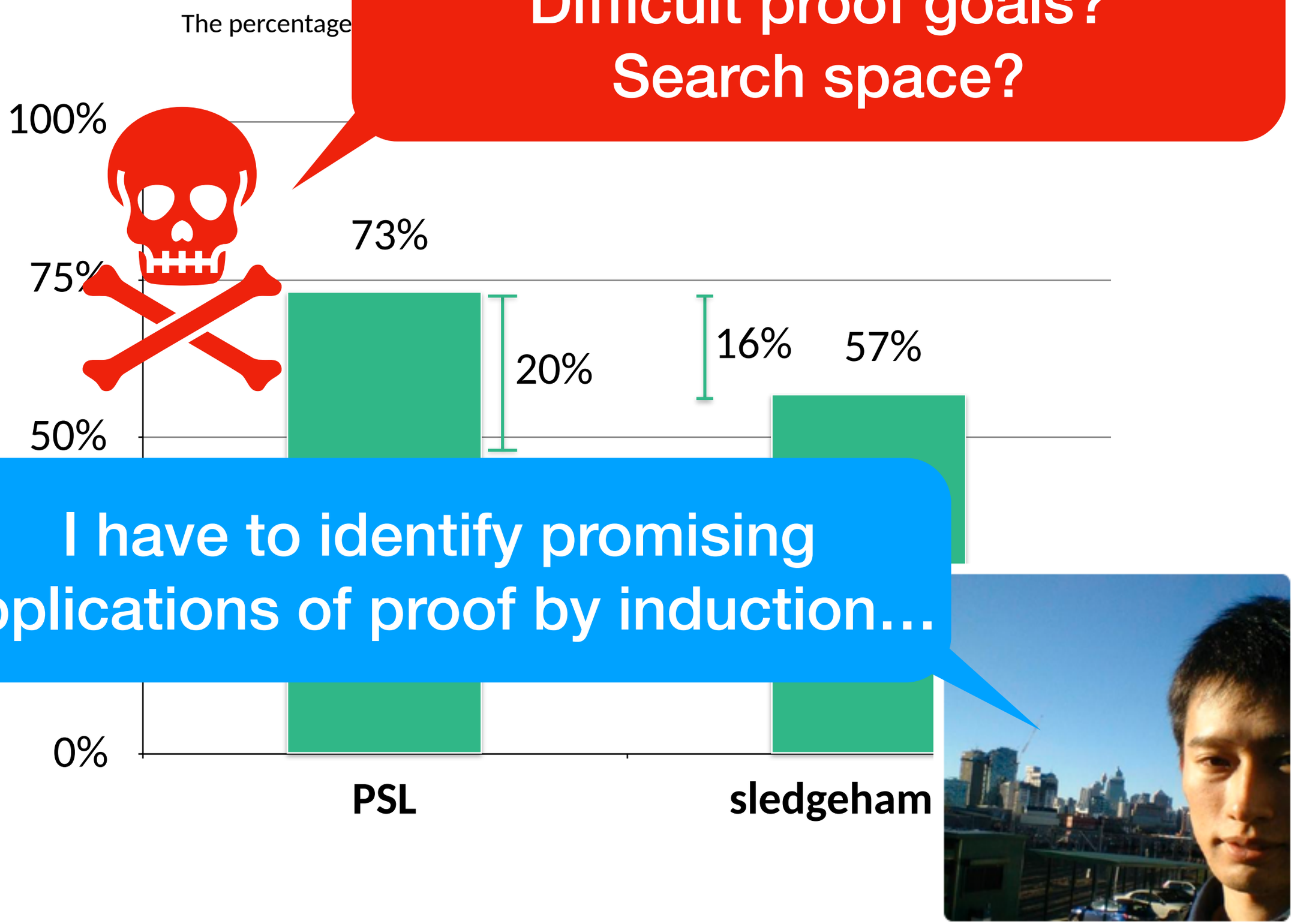
PSL vs sledgehammer

Difficult proof goals?
Search space?



PSL vs sledgehammer

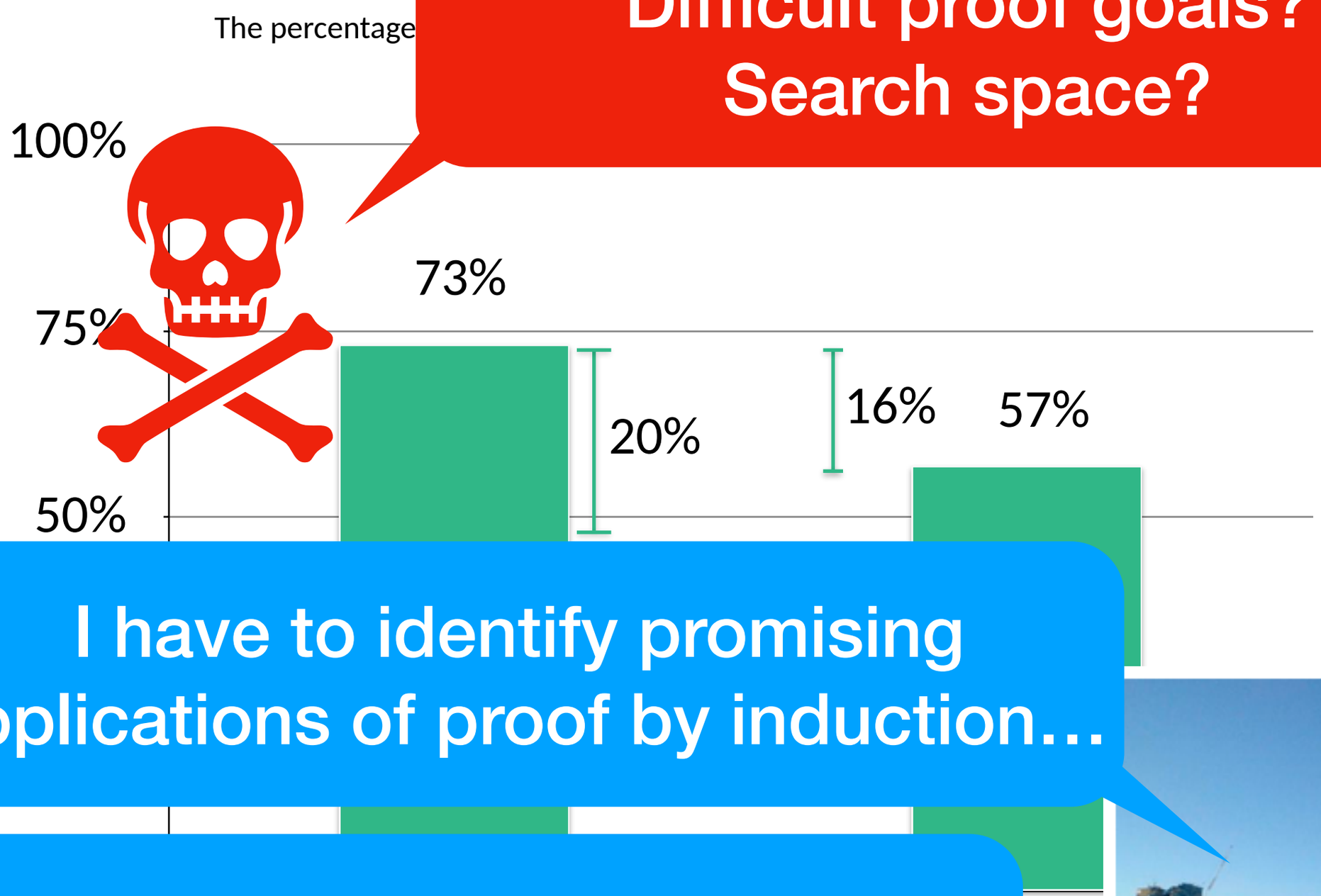
Difficult proof goals?
Search space?



I have to identify promising applications of proof by induction…

PSL vs sledgehammer

Difficult proof goals?
Search space?



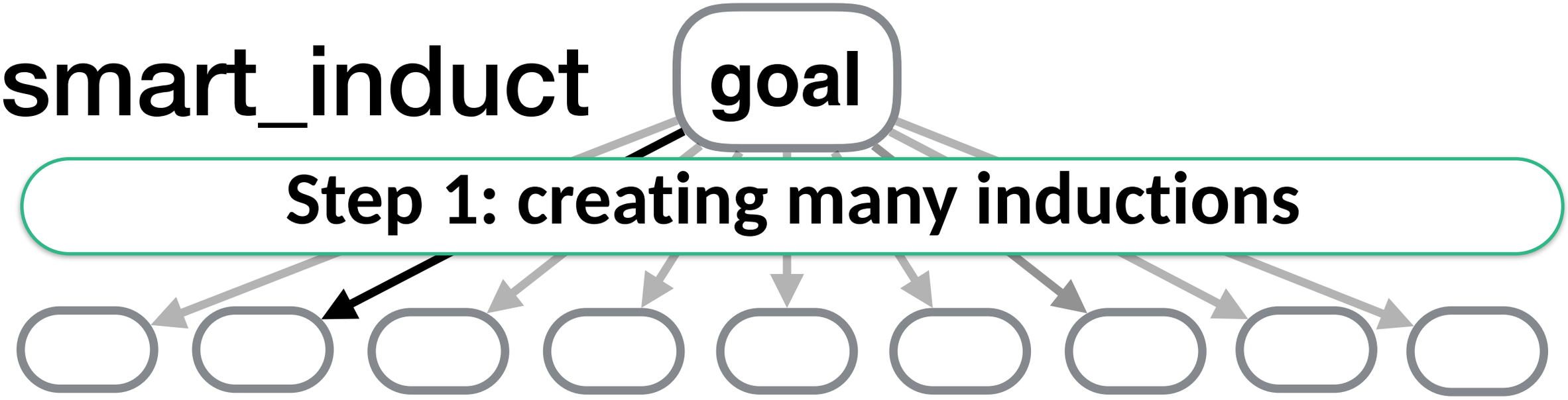
I have to identify promising applications of proof by induction…

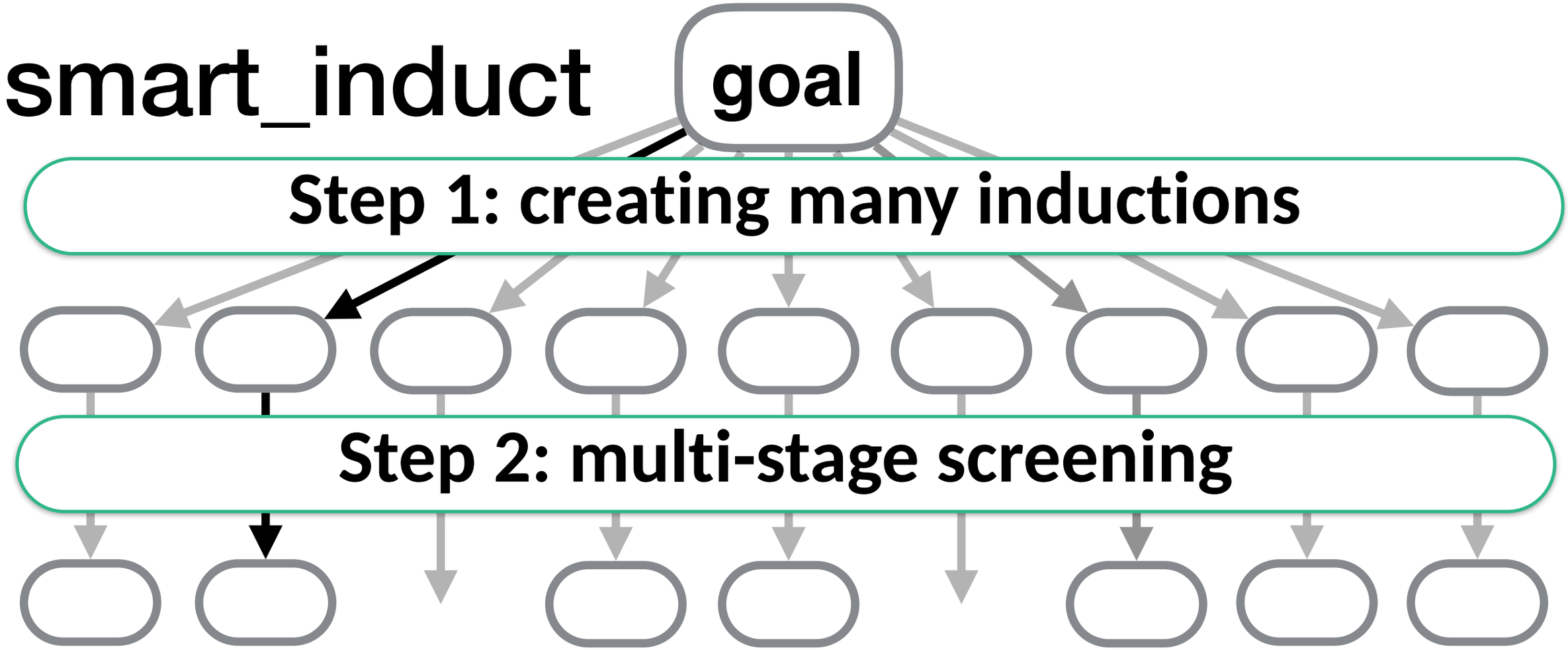
without completing a proof search

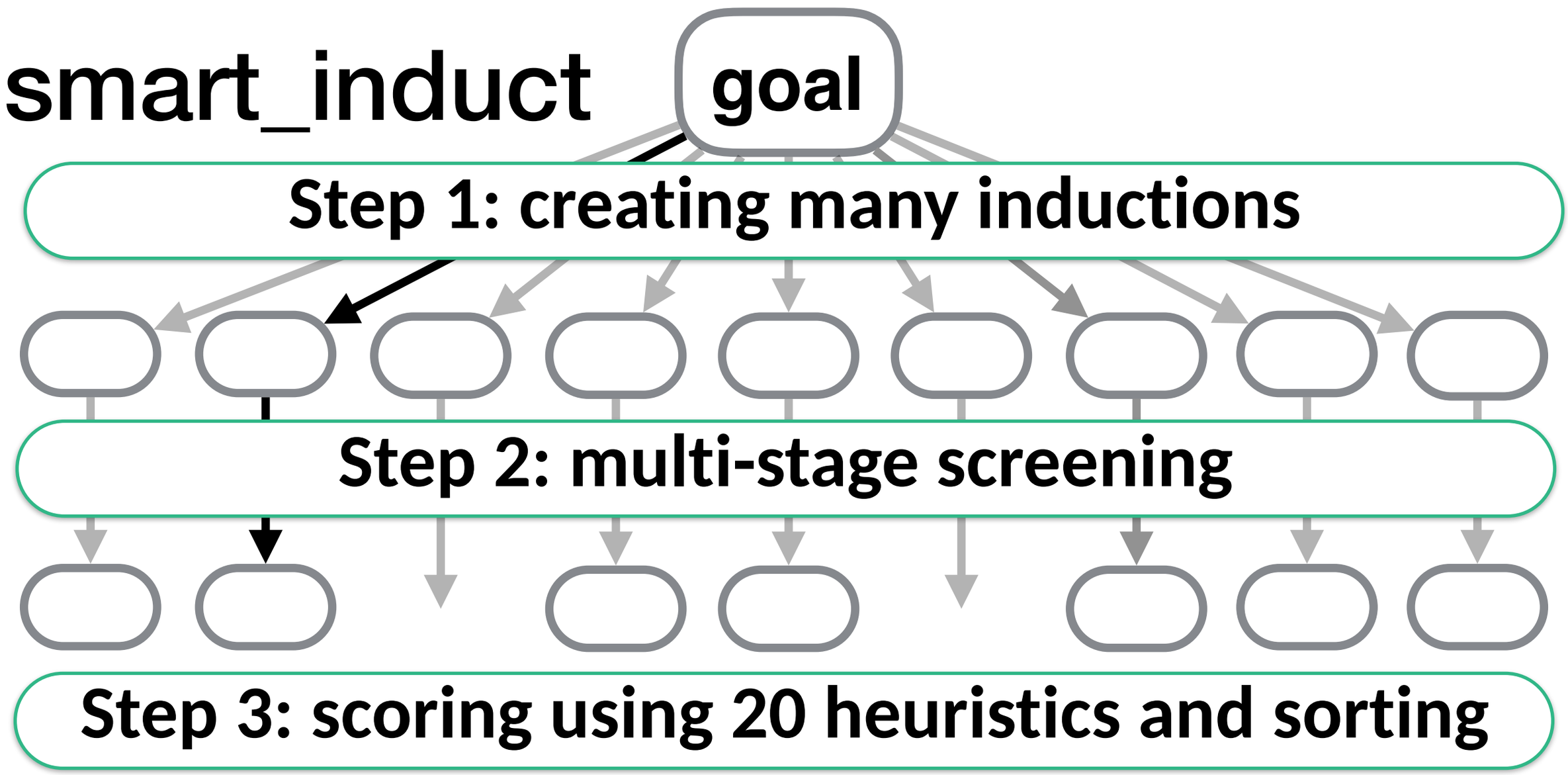
smart_induct

goal

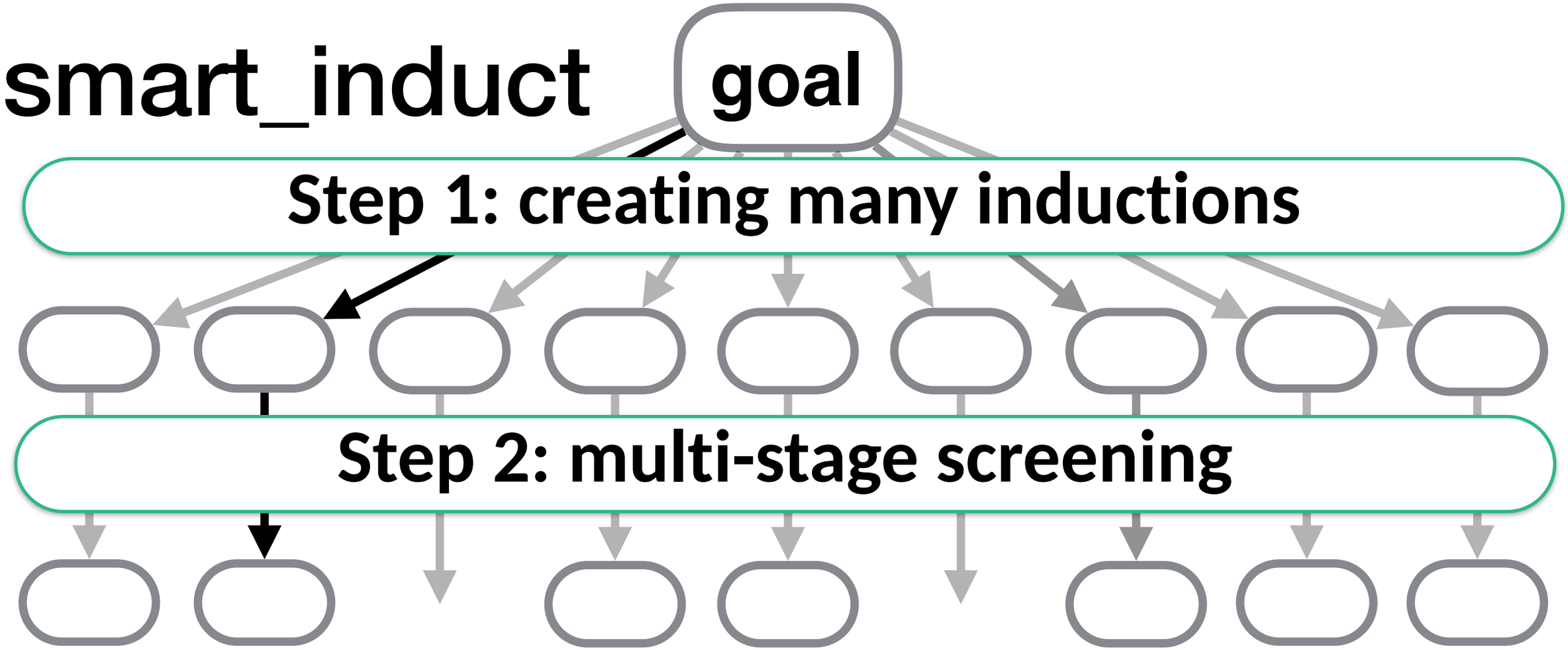
smart_induct




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


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

smart_induct



Step 3: scoring using 20 heuristics and sorting

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

smart_induct

goal

Step 1: creating many inductions

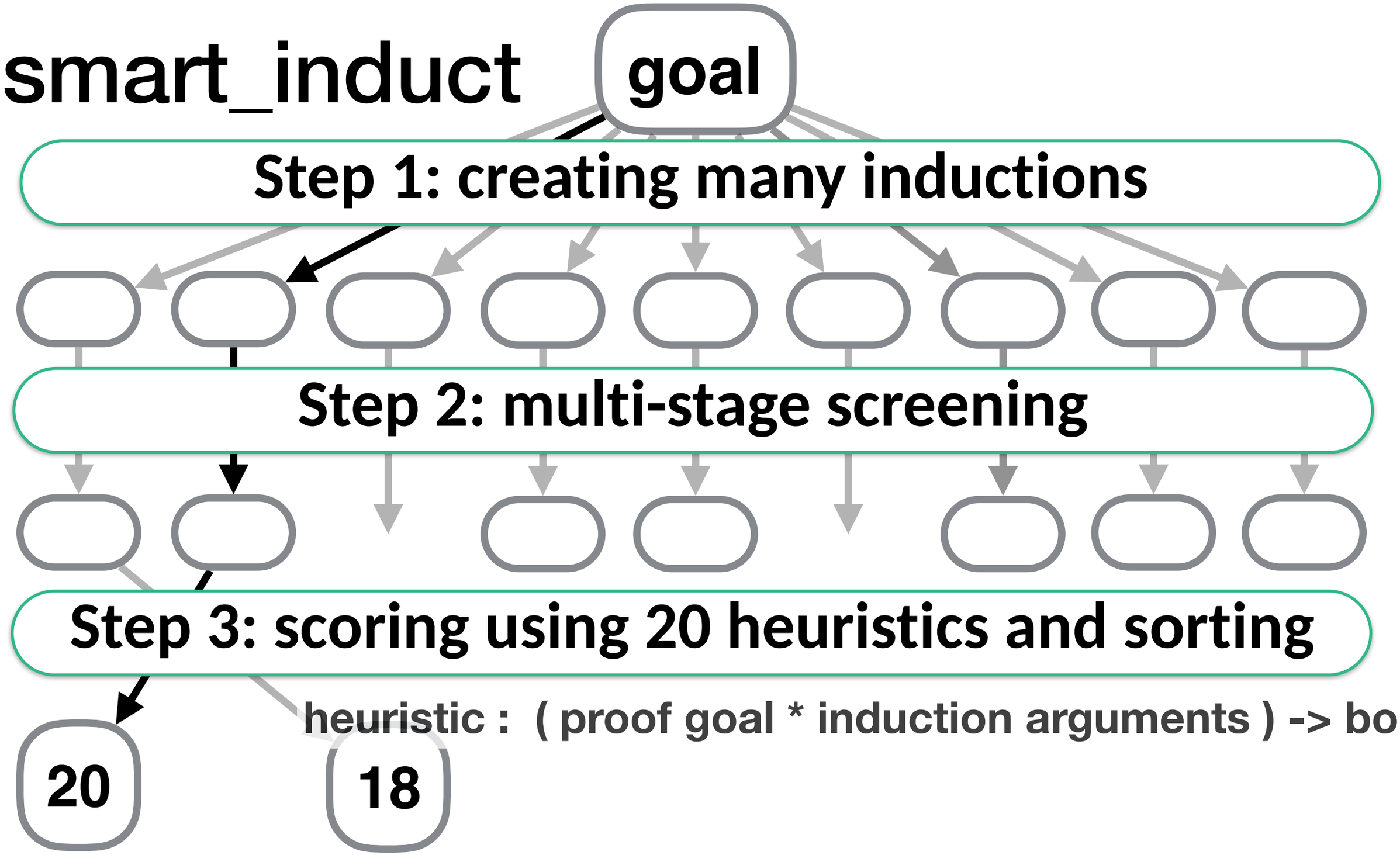
Step 2: multi-stage screening

Step 3: scoring using 20 heuristics and sorting

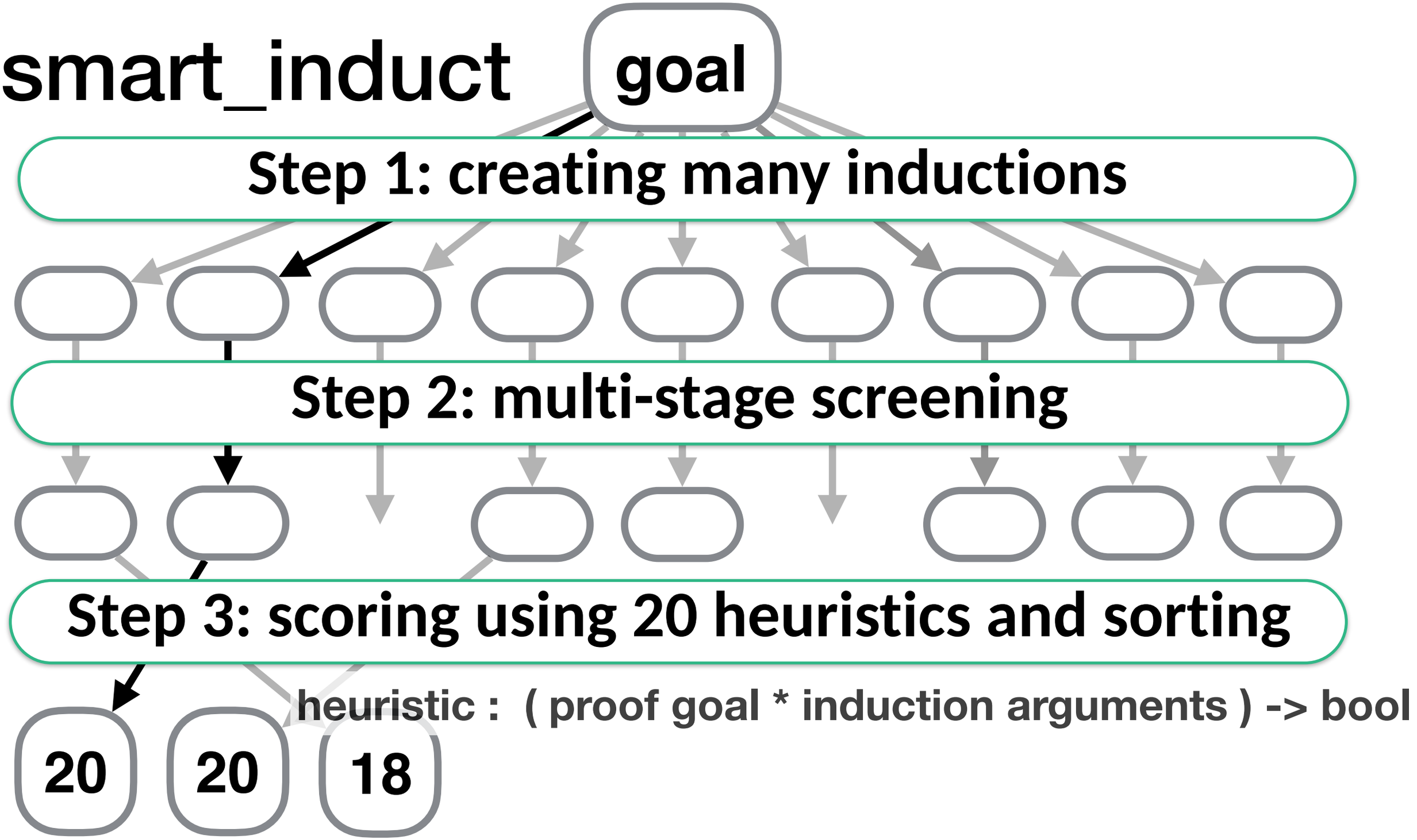
heuristic : (proof goal * induction arguments) -> bool

18

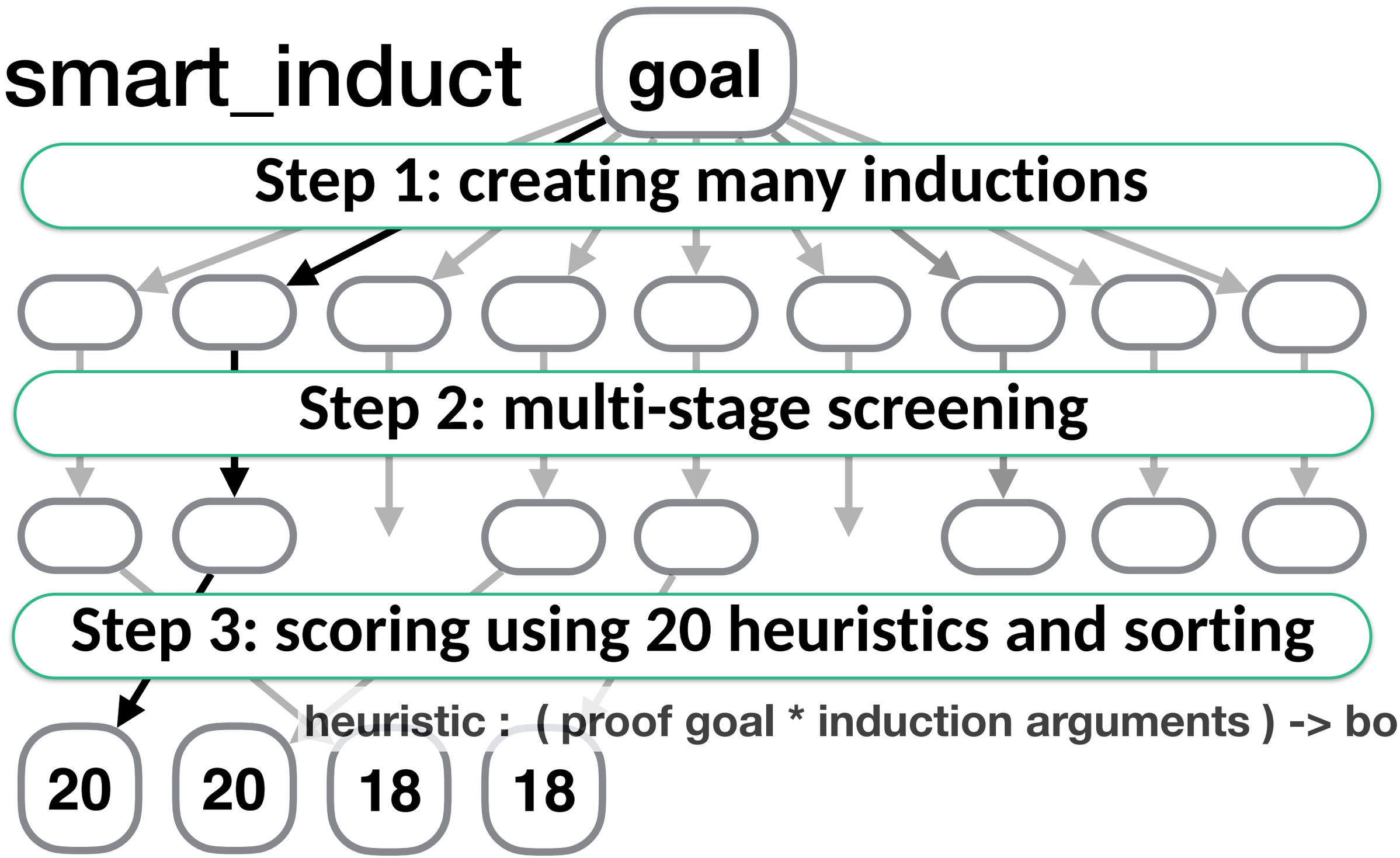
smart_induct



smart_induct



smart_induct



smart_induct

goal

Step 1: creating many inductions

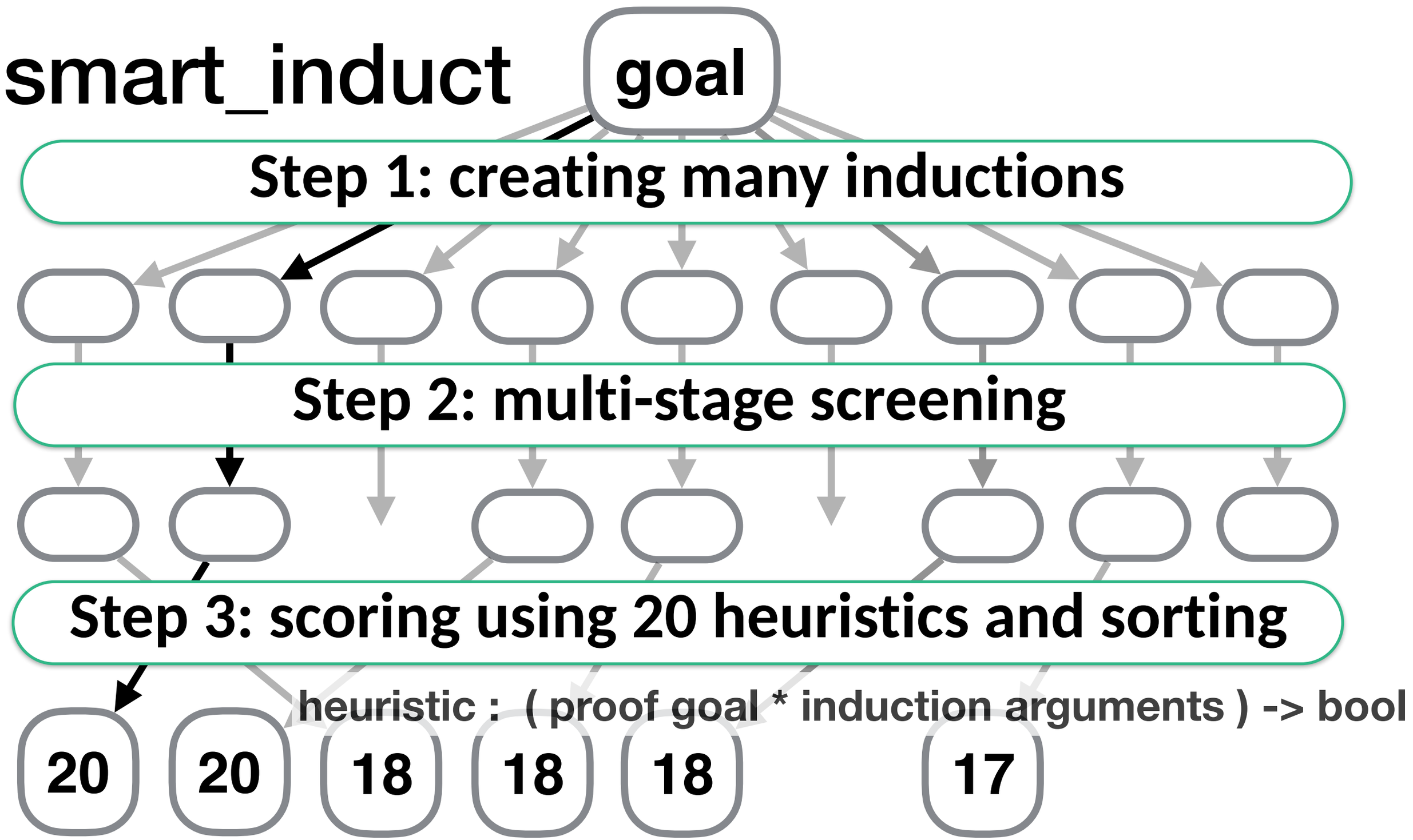
Step 2: multi-stage screening

Step 3: scoring using 20 heuristics and sorting

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

20 20 18 18 18

smart_induct



smart_induct

goal

Step 1: creating many inductions

Step 2: multi-stage screening

Step 3: scoring using 20 heuristics and sorting

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

20 20 18 18 18 18 17

smart_induct

goal

Step 1: creating many inductions

Step 2: multi-stage screening

Step 3: scoring using 20 heuristics and sorting

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

20 20 18 18 18 18 17

Step 4: short-listing

smart_induct

goal

Step 1: creating many inductions

Step 2: multi-stage screening

Step 3: scoring using 20 heuristics and sorting

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

20 20 18 18 18 18 17

Step 4: short-listing

DEMO!

DEMO3

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

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

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

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 (538/882)

DEMO3

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

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

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

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

```
smart_induct started producing combinations of induction arguments.
smart_induct produced 40 combinations of arguments for the induct method.
... out of which 32 of them return some results.
... out of which only 28 of them passes the second screening stage.
LiFtEr assertions are evaluating the first 28 of them.
Try these 10 most promising inductions!
1st candidate is apply (induct xs arbitrary: ys)
  (* The score is 20 out of 20. *)
2nd candidate is apply (induct xs)
  (* The score is 20 out of 20. *)
3th candidate is apply (induct xs ys rule: Induction_Demo.itrev.induct)
  (* The score is 18 out of 20. *)
```

Output | Query | Sledgehammer | Symbols

18,47 (551/882)

DEMO3

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

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

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

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

```
smart_induct started producing combinations of induction arguments.
smart_induct produced 40 combinations of arguments for the induct method.
... out of which 32 of them return some results.
... out of which only 28 of them passes the second screening stage.
LiFtEr assertions are evaluating the first 28 of them.
Try these 10 most promising inductions!
1st candidate is apply (induct xs arbitrary: ys)
  (* The score is 20 out of 20. *)
2nd candidate is apply (induct xs)
  (* The score is 20 out of 20. *)
3th candidate is apply (induct xs ys rule: Induction_Demo.itrev.induct)
  (* The score is 18 out of 20. *)
```

Output | Query | Sledgehammer | Symbols

18,47 (551/882)

DEMO3

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

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

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

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

```
smart_induct started producing combinations of induction arguments.
smart_induct produced 40 combinations of arguments for the induct method.
... out of which 32 of them return some results.
... out of which only 28 of them passes the second screening stage.
LiFtEr assertions are evaluating the first 28 of them.
Try these 10 most promising inductions!
1st candidate is apply (induct xs arbitrary: ys)
  (* The score is 20 out of 20. *)
2nd candidate is apply (induct xs)
  (* The score is 20 out of 20. *)
3th candidate is apply (induct xs ys rule: Induction_Demo.itrev.induct)
  (* The score is 18 out of 20. *)
```

Output | Query | Sledgehammer | Symbols

18,47 (551/882)

DEMO3

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

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

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

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

```
smart_induct started producing combinations of induction arguments.
smart_induct produced 40 combinations of arguments for the induct method.
... out of which 32 of them return some results.
... out of which only 28 of them passes the second screening stage.
LiFtEr assertions are evaluating the first 28 of them.
Try these 10 most promising inductions!
1st candidate is apply (induct xs arbitrary: ys)
  (* The score is 20 out of 20. *)
2nd candidate is apply (induct xs)
  (* The score is 20 out of 20. *)
3th candidate is apply (induct xs ys rule: Induction_Demo.itrev.induct)
  (* The score is 18 out of 20. *)
```

Output | Query | Sledgehammer | Symbols

18,47 (551/882)

goal

Step 1: creating many inductions

Step 2: multi-stage screening

Step 3: scoring using 20 heuristics and sorting

20 20 18 18 18 18 17

Step 4: short-listing

goal

Step 1: creating many inductions

Step 2: multi-stage screening

Step 3: scoring using 20 heuristics and sorting

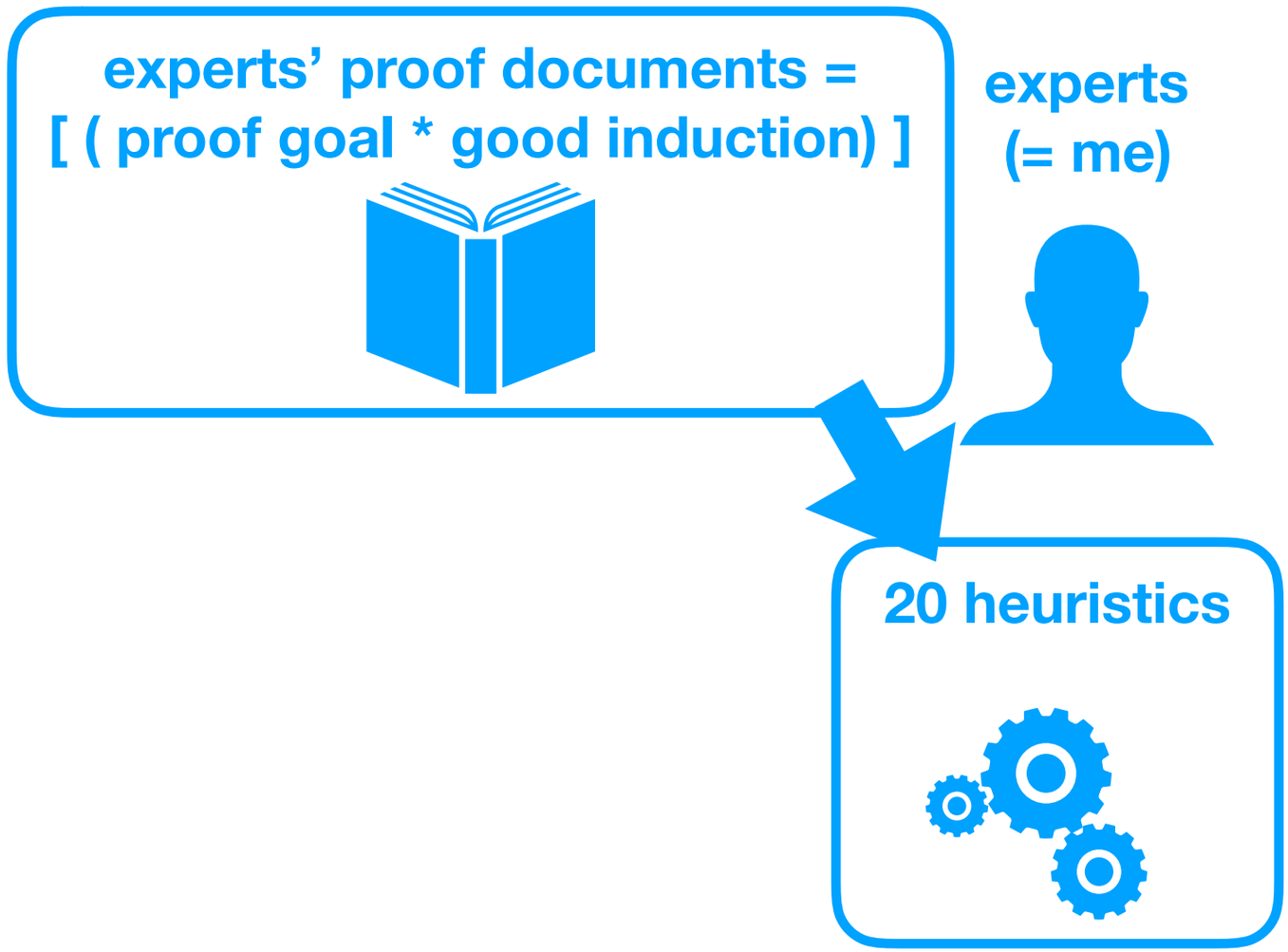
?

20 20 18 18 18 18 17

Step 4: short-listing

It is difficult to write induction heuristics.

It is difficult to write induction heuristics.



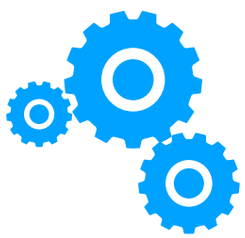
It is difficult to write induction heuristics.

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

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

experts
(= me)

20 heuristics



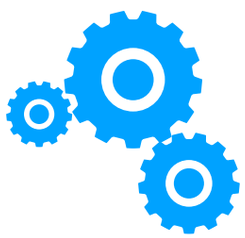
It is difficult to write induction heuristics.

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

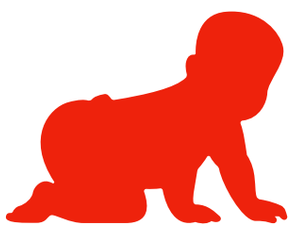
experts' proof documents =
[(proof goal * good induction)]

experts
(= me)

20 heuristics



new users



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

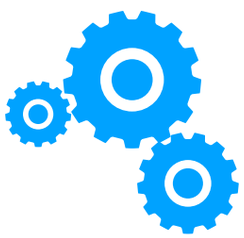
It is difficult to write induction heuristics.

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

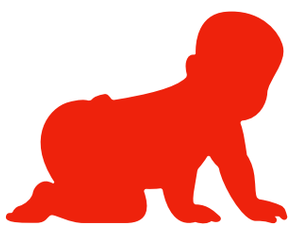
experts' proof documents =
[(proof goal * good induction)]

experts
(= me)

20 heuristics



new users



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

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

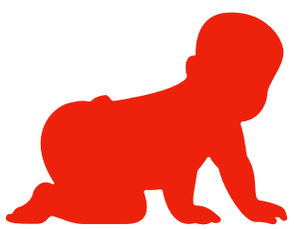
It is difficult to write induction heuristics.

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

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

experts
(= me)

new users

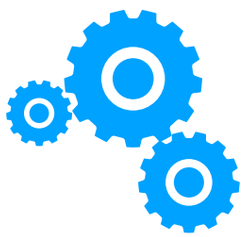


proof goal

candidate induction

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

20 heuristics



[✓ ,
✗ ,
✓ ,...]

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

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

It is difficult to write induction heuristics.

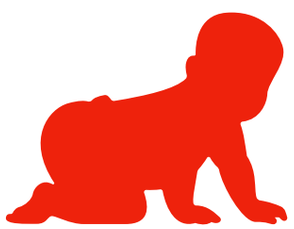
① blue = what I can see before releasing smart_induct

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

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

experts
(= me)

new users

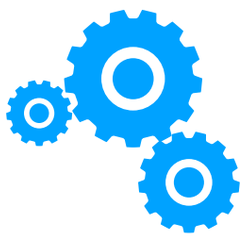


proof goal

candidate induction

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

20 heuristics



[✓ ,
✗ ,
✓ ,...]

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

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

It is difficult to write induction heuristics.

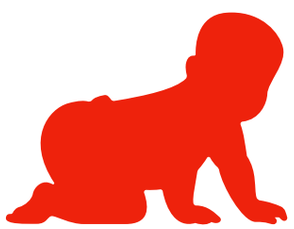
① blue = what I can see before releasing smart_induct

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

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

experts
(= me)

new users

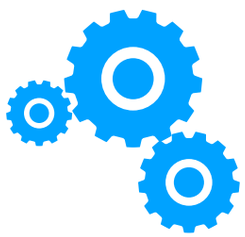


proof goal

candidate induction

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

20 heuristics



[✓ ,
✗ ,
✓ ,...]

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

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

② red = what will be developed after releasing smart_induct

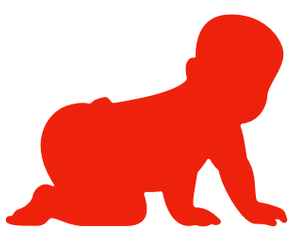
It is difficult to write induction heuristics.

① blue = what I can see before releasing smart_induct

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

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

experts
(= me)



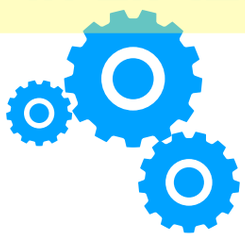
new users

proof goal

candidate induction

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

20 heuristics
in LiFtEr



[✓ ,
✗ ,
✓ ,...]

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

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

② red = what will be developed after releasing smart_induct

It is difficult to write induction heuristics.

① blue = what I can see before releasing smart_induct

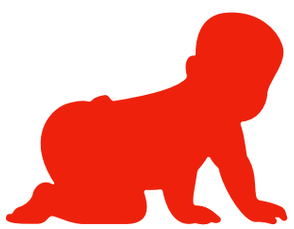
```
lemma "itrev xs ys = rev xs @ ys"
by(induct xs ys rule:"itrev.induct") auto
```

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

experts
(= me)

LiFtEr: Logical
Feature Extractor

new users

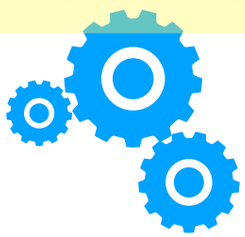


proof goal

candidate induction

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

20 heuristics
in LiFtEr



[✓ ,
✗ ,
✓ ,...]

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

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

② red = what will be developed after releasing smart_induct

Abstract Syntax of LiFtEr

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

Abstract Syntax of LiFtEr

assertion := *atomic* | *connective* | *quantifier* | (*assertion*)
type := `term` | `term_occurrence` | `rule` | `number`
modifier_term := `induction_term` | `arbitrary_term`
quantifier := $\exists x$: *type* . *assertion*
| $\forall x$: *type* . *assertion*
| $\exists x$: `term` $\in$ *modifier_term* . *a*
| $\forall x$: `term` $\in$ *modifier_term* . *a*
| $\exists x$: `rule` . *assertion*
| $\exists x$: `term_occurrence` $\in y$: `term` . *assertion*
| $\forall x$: `term_occurrence` $\in y$: `term` .
connective := `True` | `False` | *assertion* $\vee$ *asse* *assertion*
assertion $\rightarrow$ *assertion* | $\neg$ *asserti*
pattern := `all_only_var` | `all_constructor`

atomic :=
rule `is_rule_of` *term_occurre*
| *term_occurrence* `term_occurrence` *erm*
| `are_same_term` (*term_occurrence* , *term_occ*
| *term_occurrence* `is_in_term_occurrence` *term*
| `is_atomic` *term_occurrence*
| `is_constant` *term_occurrence*
| `is_recursive_constant` *term_occurrence*
| `is_variable` *term_occurrence*
| `is_free_variable` *term_occurrence*
| `is_bound_variable` *term_occurrence*
| `is_lambda` *term_occurrence*
| `is_application` *term_occurrence*
| *term_occurrence* `is_an_argument_of` *occurrence*
| *term_occurrence* `is_nth_argument_of` *occurrence*

Example Heuristic in LiFtEr (in Abstract Syntax)

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

Example Heuristic in LiFtEr (in Abstract Syntax)

implication

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

Example Heuristic in LiFtEr (in Abstract Syntax)

implication

```
∃ 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 Heuristic 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 Heuristic in LiFtEr (in Abstract Syntax)

implication

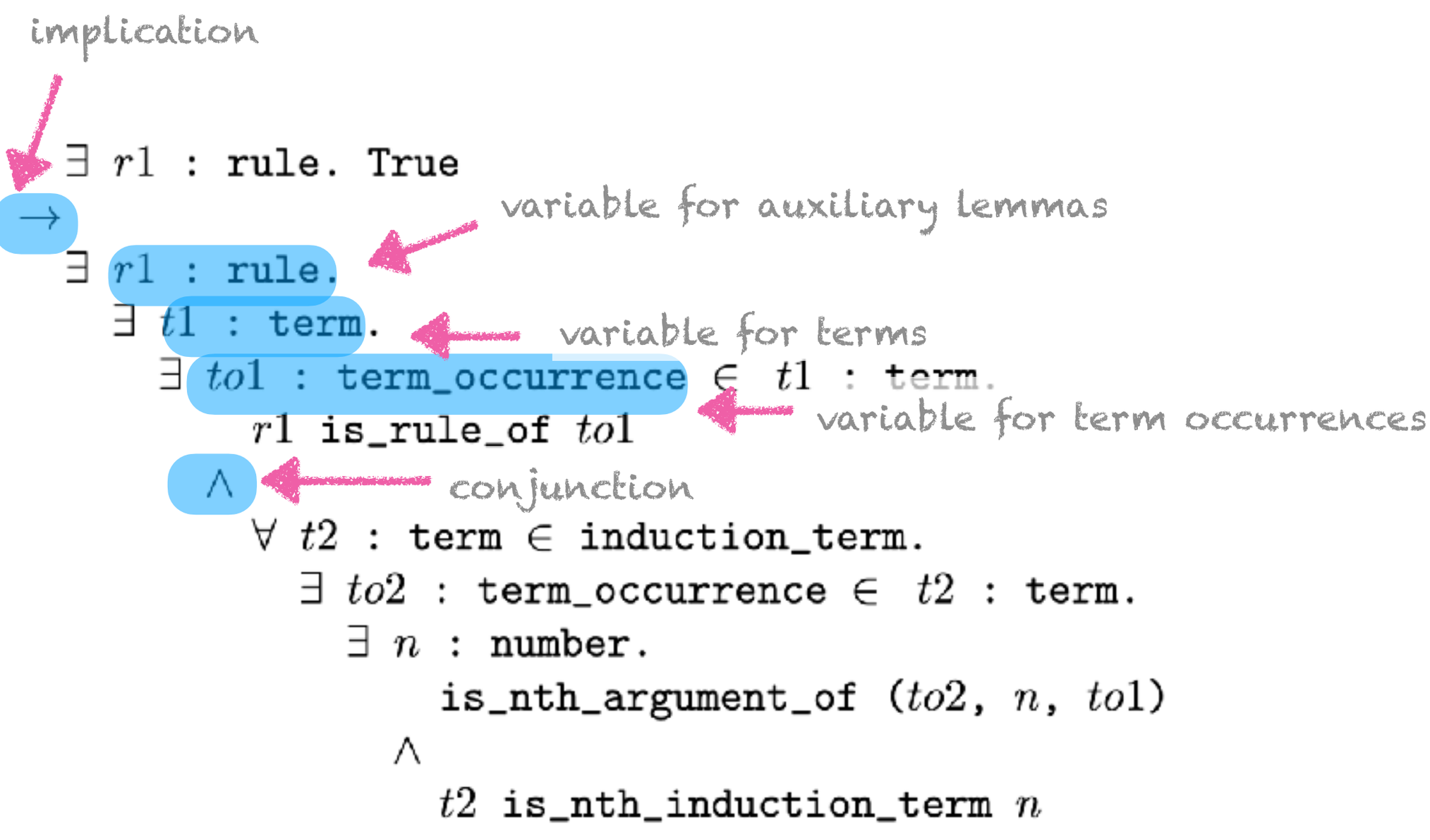
```
∃ r1 : rule. True
→
  ∃ r1 : rule.
    ∃ t1 : term.
      ∃ to1 : term_occurrence ∈ t1 : term.
          r1 is_rule_of to1
        ∧
          ∀ t2 : term ∈ induction_term.
            ∃ to2 : term_occurrence ∈ t2 : term.
              ∃ n : number.
                  is_nth_argument_of (to2, n, to1)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

variable for auxiliary lemmas

variable for terms

conjunction

Example Heuristic in LiFtEr (in Abstract Syntax)



Example Heuristic 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

variable for natural numbers

Example Heuristic in LiFtEr (in Abstract Syntax)

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

implication

variable for auxiliary lemmas

variable for terms

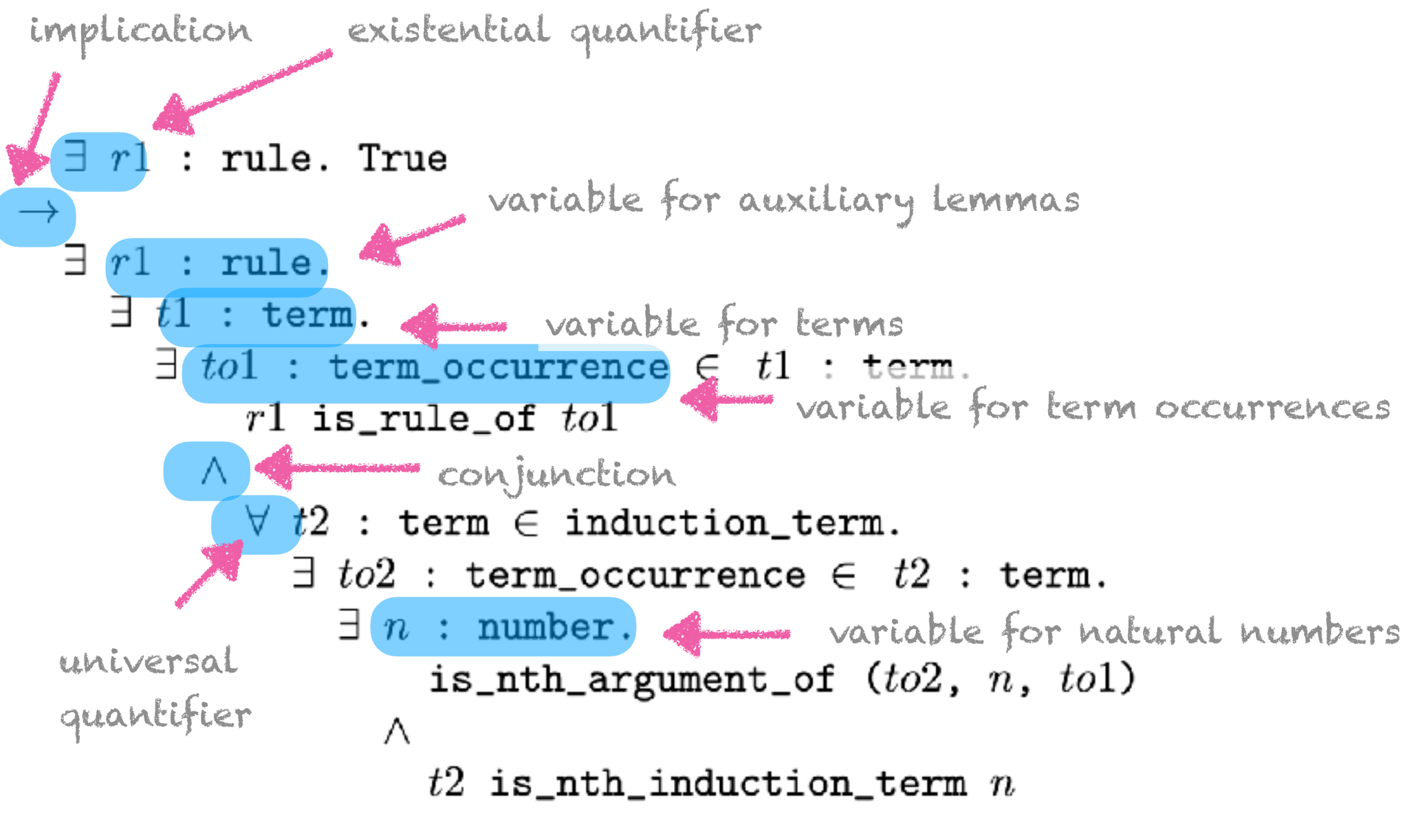
variable for term occurrences

conjunction

universal quantifier

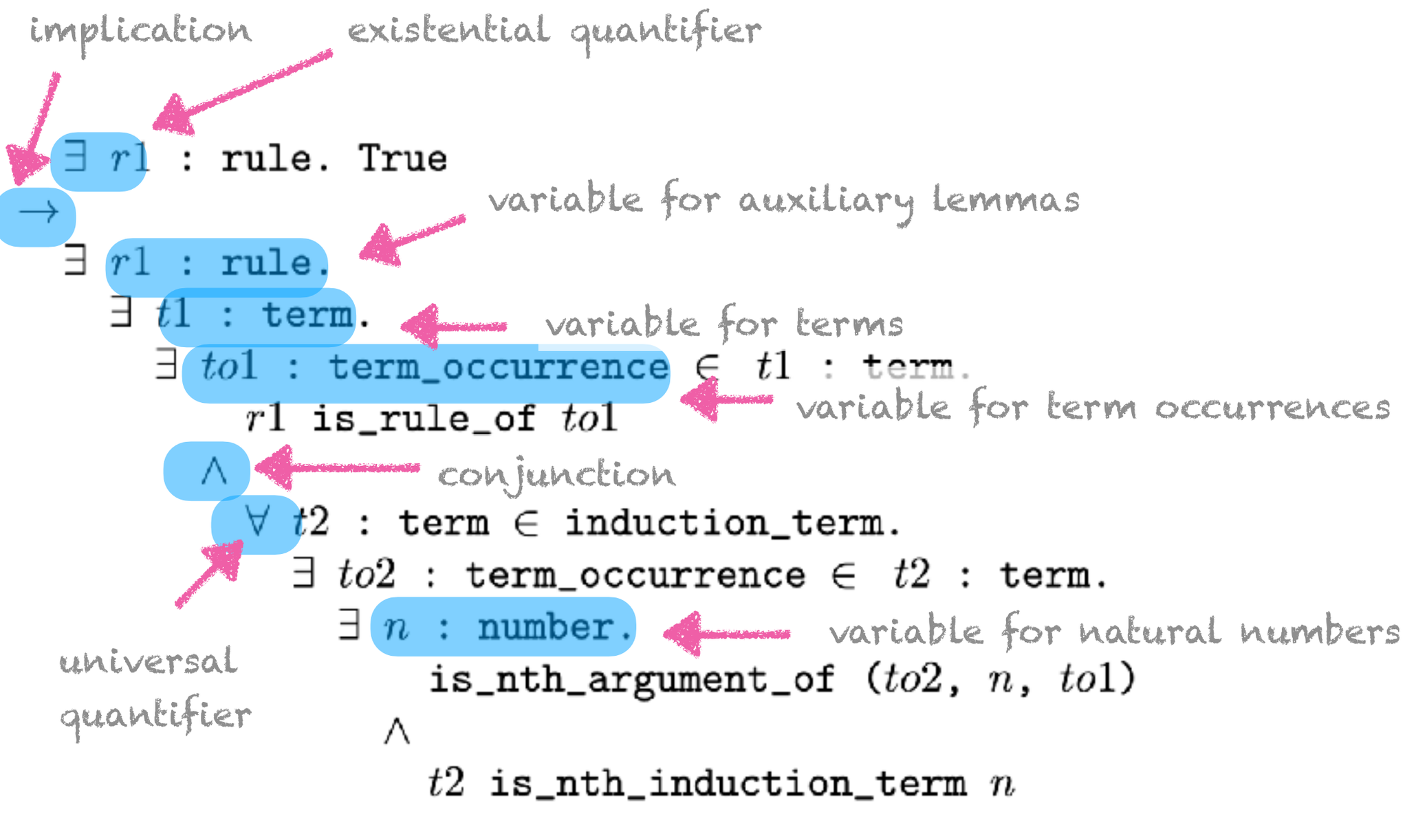
variable for natural numbers

Example Heuristic in LiFtEr (in Abstract Syntax)



Example Heuristic in LiFtEr (in Abstract Syntax)

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

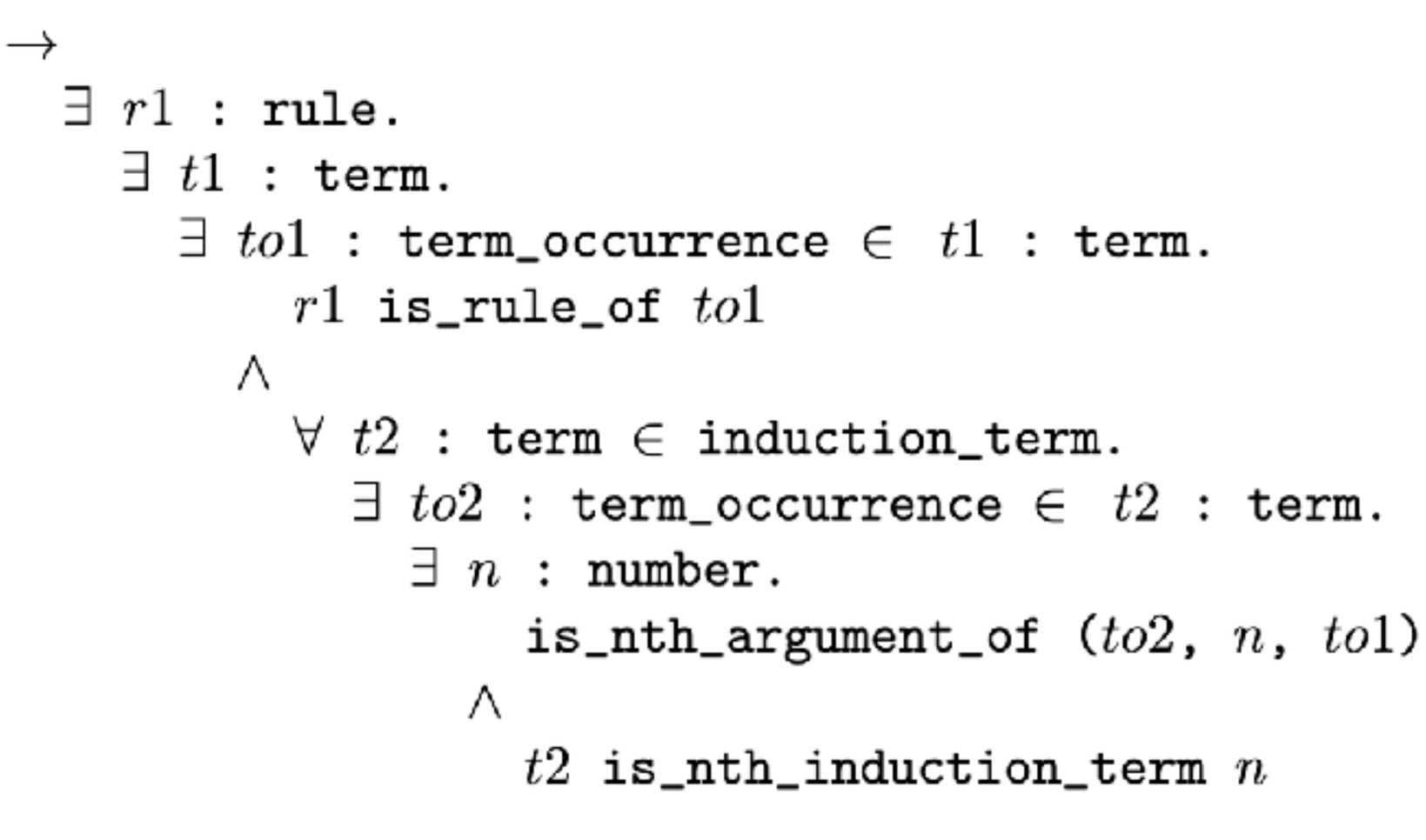


```
primrec rev :: "'a list ⇒ 'a list" where
 "rev  []       = []" |
 "rev (x # xs) = rev xs @ [x]"

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
```



```
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.

(r1 = itrev.induct)

∃ 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

```
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

```
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)

```
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)

```
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
```

```
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 $to1$ True! r1 (= itrev.induct) is a lemma about to1 (= itrev).
      ∧
        ∀ $t2$: term ∈ induction_term.
          ∃ $to2$: term_occurrence ∈ $t2$: term.
            ∃ n : number.
              is_nth_argument_of ($to2$, n, $to1$)
              ∧
              $t2$ is_nth_induction_term n

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

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

to1

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

∃ *r1* : rule. True

→ r1

∃ *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.
          ∃ *to2* : term_occurrence ∈ *t2* : term.
            ∃ *n* : number.
              is_nth_argument_of (*to2*, *n*, *to1*)
              ∧
              t2 is_nth_induction_term *n*

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

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

to1

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

r1

$\exists$ *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 = itrev.induct)
(t1 = itrev)
(to1 = itrev)

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

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

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

to1

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

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.
∃ *to2* : term_occurrence ∈ *t2* : term.
∃ *n* : number.
is_nth_argument_of (*to2*, *n*, *to1*)
∧
t2 is_nth_induction_term *n*

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

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

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

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

```
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

```
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

```
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

```
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

```
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

```
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) ?

```
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)

```
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)

```
∃ 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 : 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
```

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
```

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

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

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

∃ $r1$: 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

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

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

same lemma ->

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

∃ 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
```

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

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

same lemma ->

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

different order ->

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

∃ 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

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

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

same lemma ->
different order ->

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

∃ 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

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

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

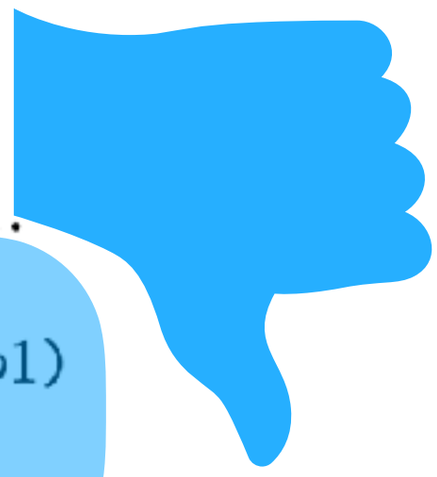
same lemma ->

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

different order ->

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

```
∃ 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 : 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
```

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 ->

```
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
```

$\exists$ *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 ->

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 ->

```
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
```

to1

r1

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

→

$$
\begin{array}{l}
\exists\, r1 : \mathtt{rule}.\\
\quad \exists\, t1 : \mathtt{term}.\\
\quad\quad \exists\, to1 : \mathtt{term\_occurrence} \in t1 : \mathtt{term}.\\
\qquad r1\ \mathtt{is\_rule\_of}\ to1\\
\qquad\quad \wedge\\
\qquad\quad \forall\, t2 : \mathtt{term} \in \mathtt{induction\_term}.\\
\qquad\qquad \exists\, to2 : \mathtt{term\_occurrence} \in t2 : \mathtt{term}.\\
\qquad\qquad\quad \exists\, n : \mathtt{number}.\\
\qquad\qquad\qquad \mathtt{is\_nth\_argument\_of}\ (to2,\ n,\ to1)\\
\qquad\qquad\qquad\quad \wedge\\
\qquad\qquad\qquad t2\ \mathtt{is\_nth\_induction\_term}\ n
\end{array}
$$

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)

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 ->
∃ 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
```

to1

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 = 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 ->
∃ 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
```

to1

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 = 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
```

t2 t01

∃ 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)

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 to1 to2

∃ r1 : rule. True

→

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

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

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

∧

∀ t2 : term ∈ induction_term.
∃ to2 : term_occurrence ∈ t2 : term.

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

∃ 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

t2 = is1, s, and stk
to1 = exec
to2 = is1, s, and stk
r1 = exec.induct

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
```

t2 t01 t02 r1

∃ r1 : rule. True

→

∃ 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.

(t2 = is1, s, and stk)
(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 ->

```
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
```

t2 first

to1 first

r1

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 = 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 = 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.

(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) ?

∧

t2 is_nth_induction_term n — when t2 is s (n -> 2) ?

new types ->

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

new constants ->

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

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)

∧

t2 is_nth_induction_term n when t2 is s (n -> 2)

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)"
```

t2 to1 to2

a model proof ->

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

first second third r1

∃ r1 : rule. True

→

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

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

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

(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 ->

```
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)

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)

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

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

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

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

$\exists\ r1$: 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

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

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

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

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

$\exists\ r1$: 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

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

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

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

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

$\exists\ r1$: 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

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

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

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

same lemma ->

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

$\exists$ 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

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

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

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

same lemma ->

different order ->

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

∃ 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

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

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

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

same lemma ->

different order ->

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

∃ 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

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

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

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

same lemma ->

different order ->

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

$\exists$ 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

smart_induct… does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

smart_induct... does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

smart_induct... does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

smart_induct… does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

smart_induct… does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

smart_induct… does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

smart_induct… does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

smart_induct… does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

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

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

smart_induct… does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

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

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

smart_induct… does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

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

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

Overall, the results are good!

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

smart_induct... does it work?

Table 1: Coincidence Rates of `smart_induct`.

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

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

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

Overall, the results are good!

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

WIP!

Today I talked about…

2013 ~ 2017

Intern & Engineer

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

with Prof. Gerwin Klein

Cogent

PhD in AI for theorem proving

2017 ~ 2018

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

with Prof. Cezary Kaliszyk

PSL

PaMpeR

2018 ~ 2020/21

http://ai4reason.org/members.html

with Dr. Josef Urban

LiFtEr

smart_induct

Today I talked about…

2013 ~ 2017

Intern & Engineer

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

with Prof. Gerwin Klein

Cogent — **ASPLOS2016**, **ITP2016**, **ICFP2016**

PhD in AI for theorem proving

2017 ~ 2018

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

with Prof. Cezary Kaliszyk

PSL — **CADE2017**, **CICM2018**

PaMpeR — **ASE2018**

2018 ~ 2020/21

http://ai4reason.org/members.html

with Dr. Josef Urban

LiFtEr — **APLAS2019**

smart_indu — **under review at IJCAR2020**

Today I talked about…　*fin.*

2013 ~ 2017

Intern & Engineer

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

with Prof. Gerwin Klein

Cogent

ASPLOS2016
ITP2016
ICFP2016

PhD in AI for theorem proving

2017 ~ 2018

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

with Prof. Cezary Kaliszyk

PSL

CADE2017
CICM2018

PaMpeR

ASE2018

2018 ~ 2020/21

http://ai4reason.org/members.html

with Dr. Josef Urban

LiFtEr

APLAS2019

smart_indu

under review at IJCAR2020

backup slides for Q&A

<- simple representation


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

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

<- simple representation


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

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

<- relevant definitions

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

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

<- simple representation


```
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

```
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)

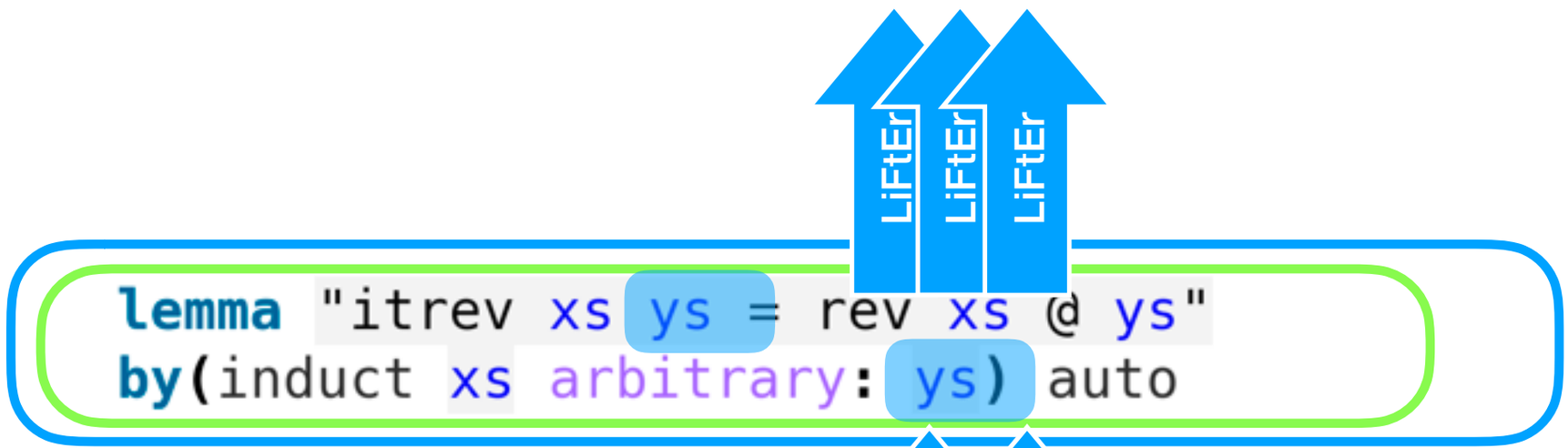
```
primrec rev :: "'a list ⇒ 'a list" where
 "rev   []       = []" |
 "rev (x # xs) = rev xs @ [x

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

<- relevant definitions

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

<- simple representation




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

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

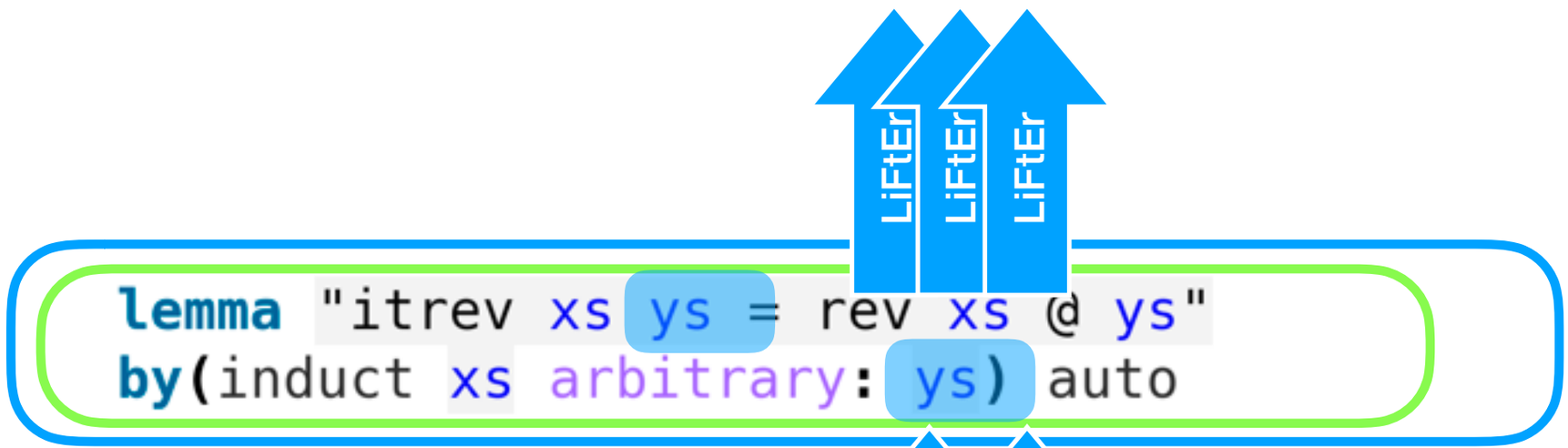
```
primrec rev :: "'a list ⇒ 'a list" where
 "rev   []       = []" |
 "rev (x # xs) = rev xs @ [x

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

<- relevant definitions

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

<- simple representation




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

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

```
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)"
```

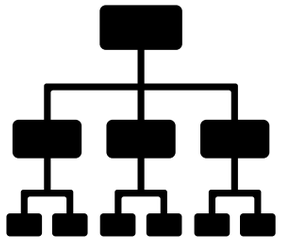
<- relevant definitions

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

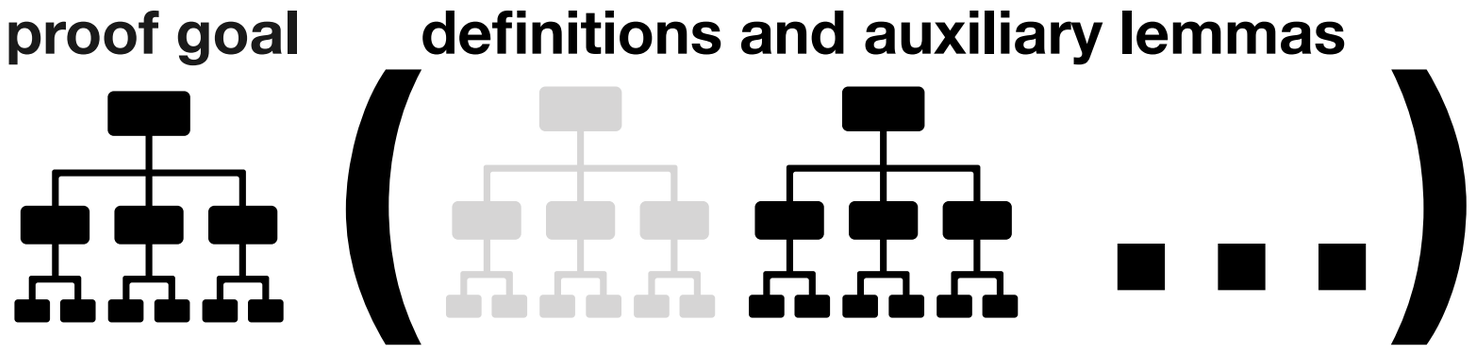
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions

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

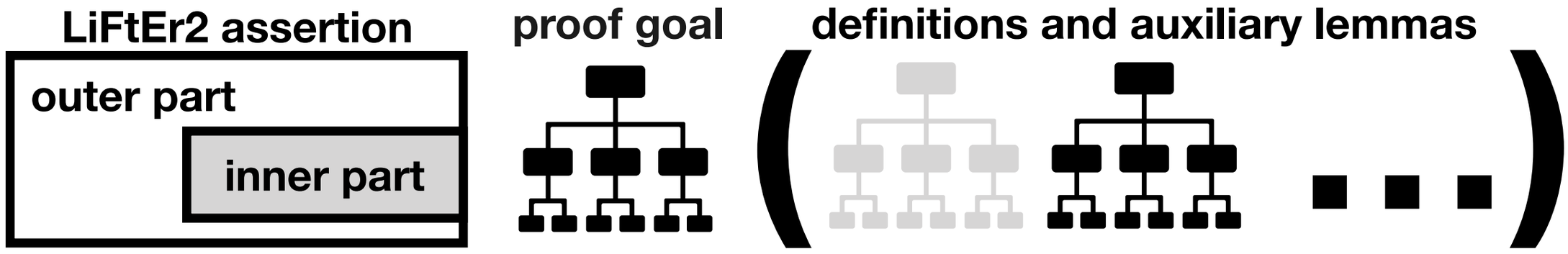
proof goal



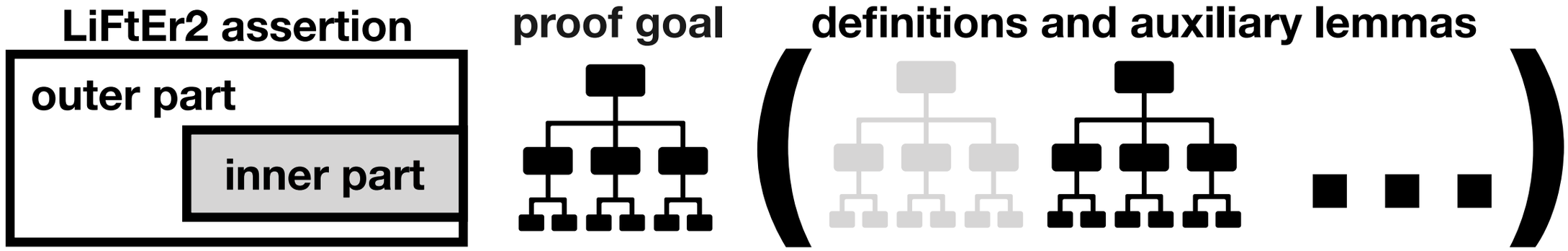
LiFtEr$_2$ (proof goal * induction arguments) -> bool
*** relevant definitions**



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

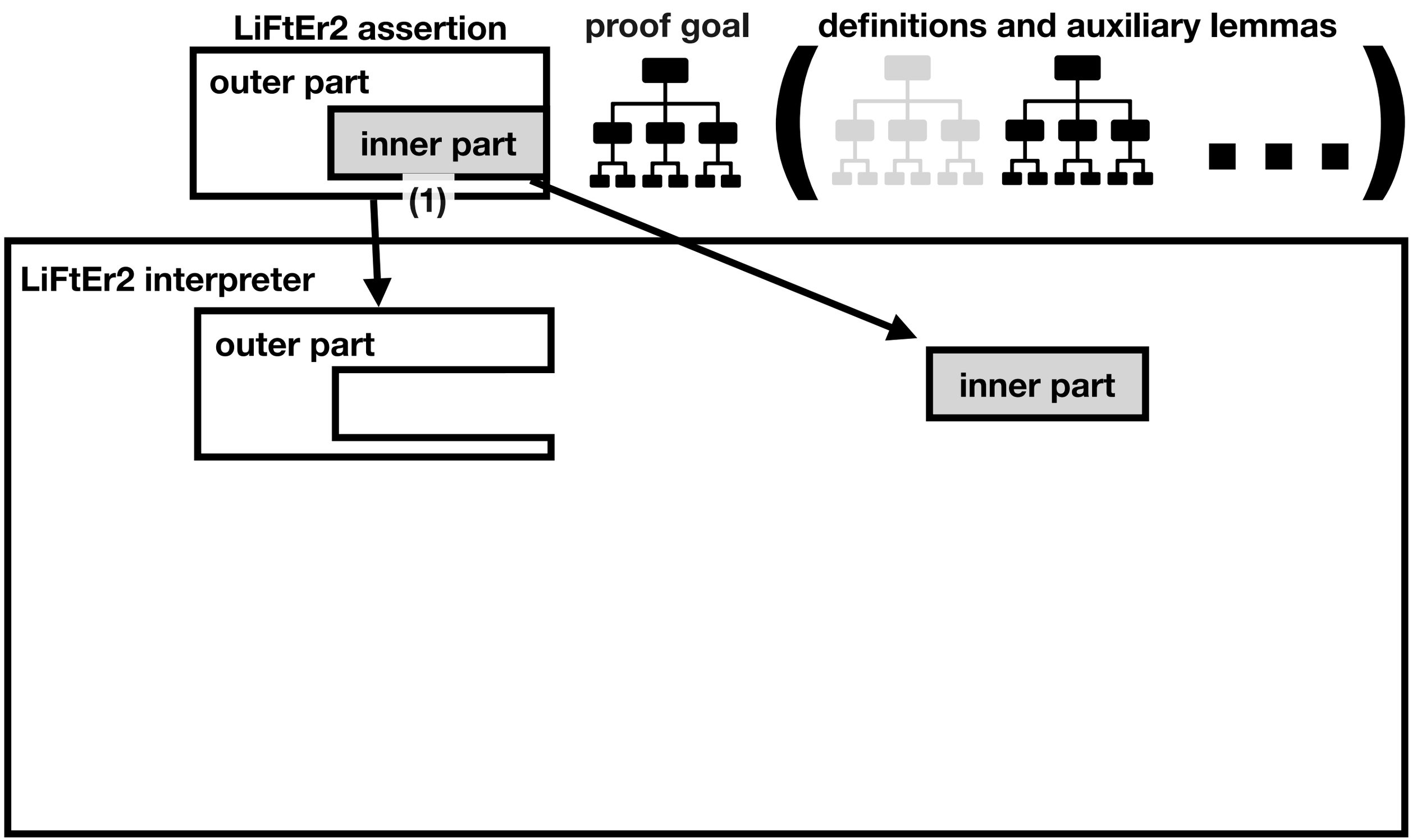


LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions

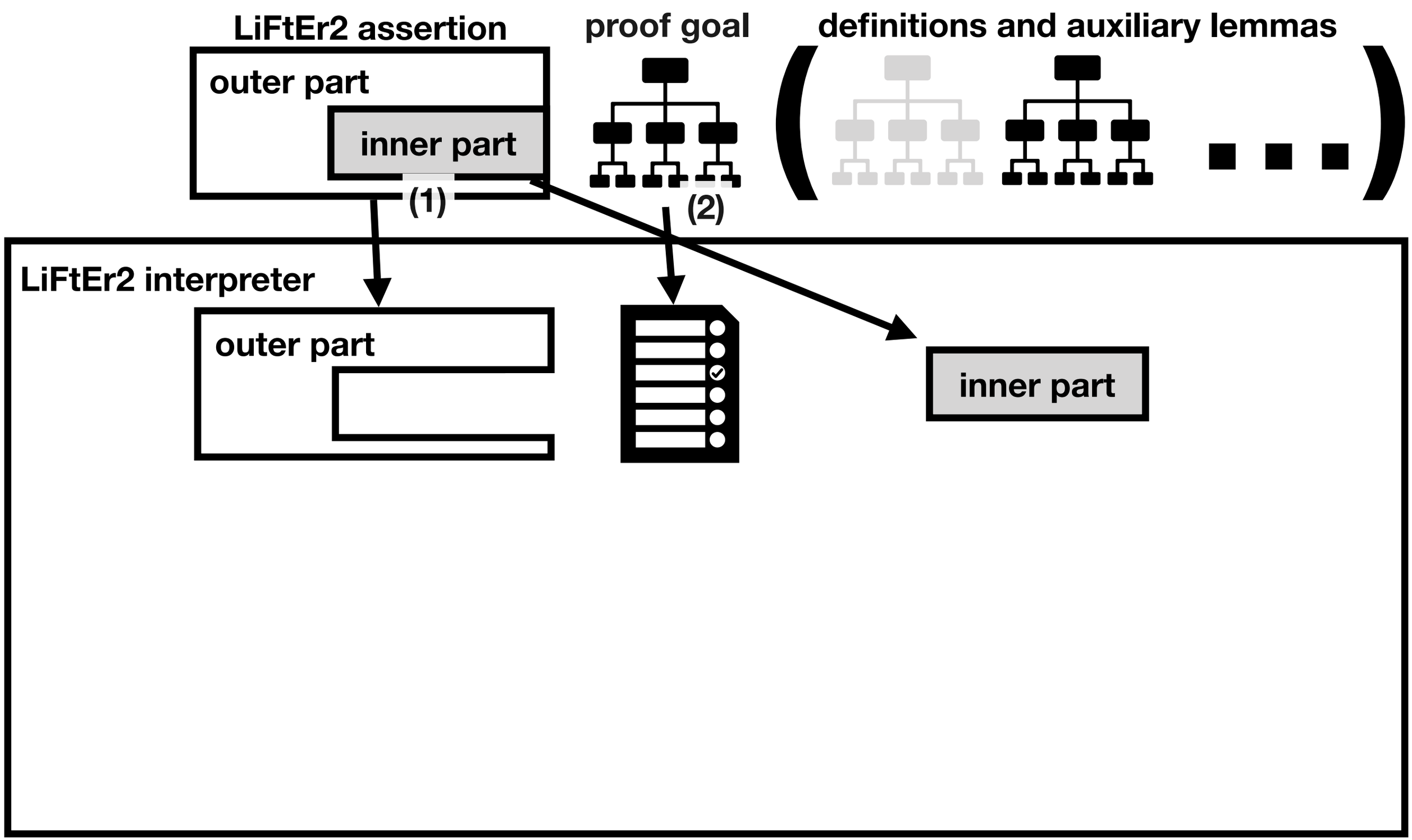


LiFtEr2 interpreter

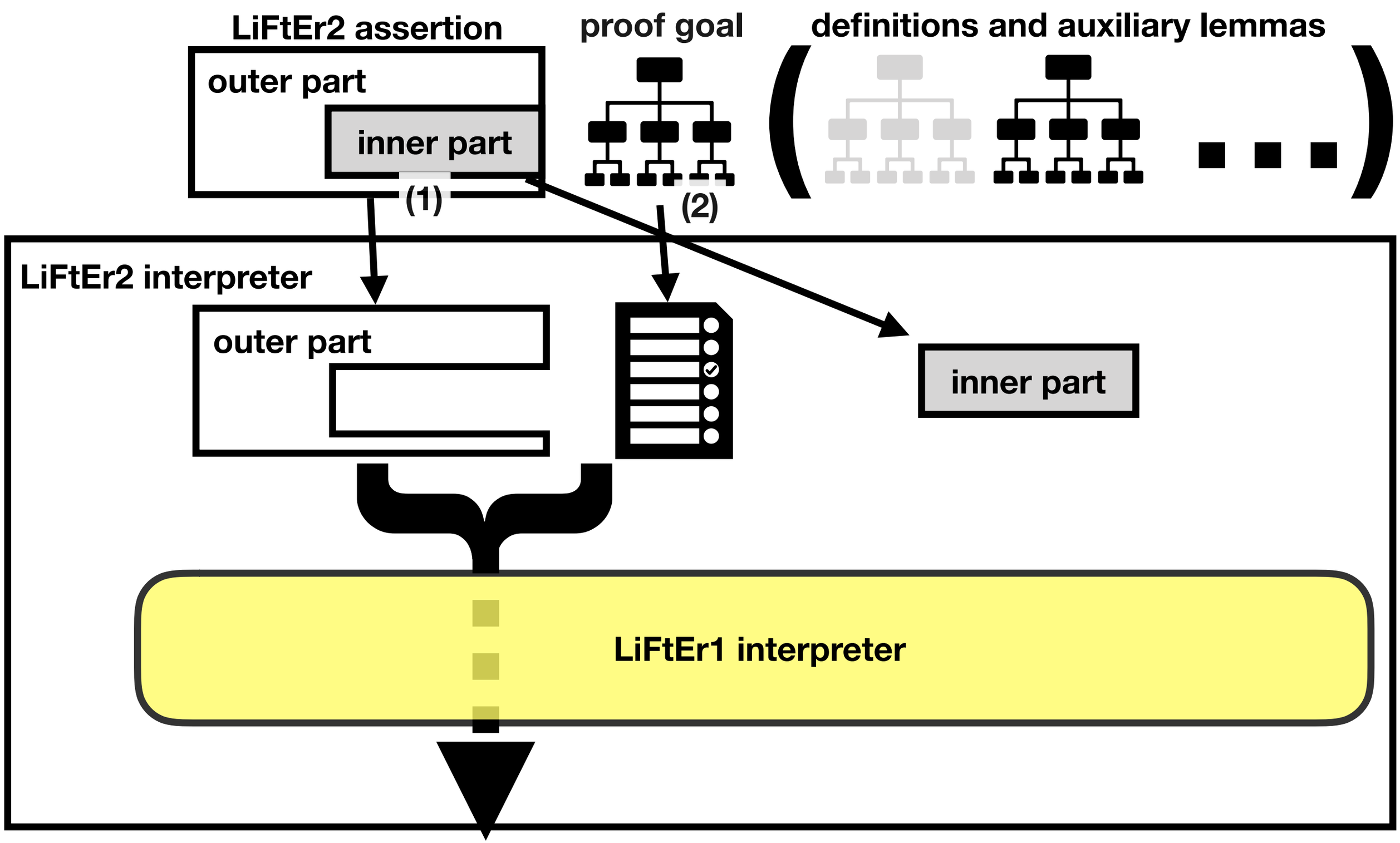
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions



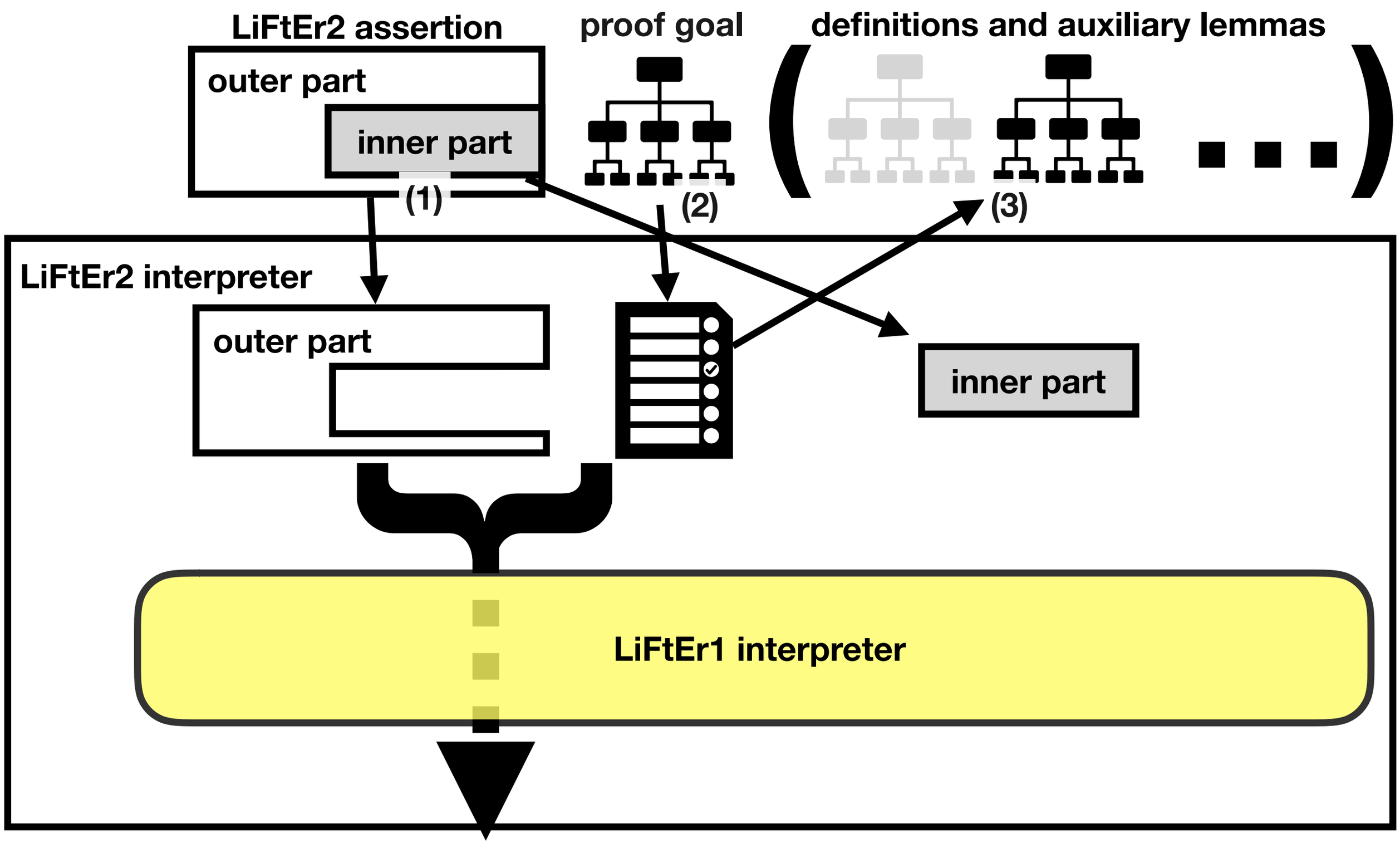
LiFtEr$_2$ (proof goal * induction arguments) -> bool
* relevant definitions



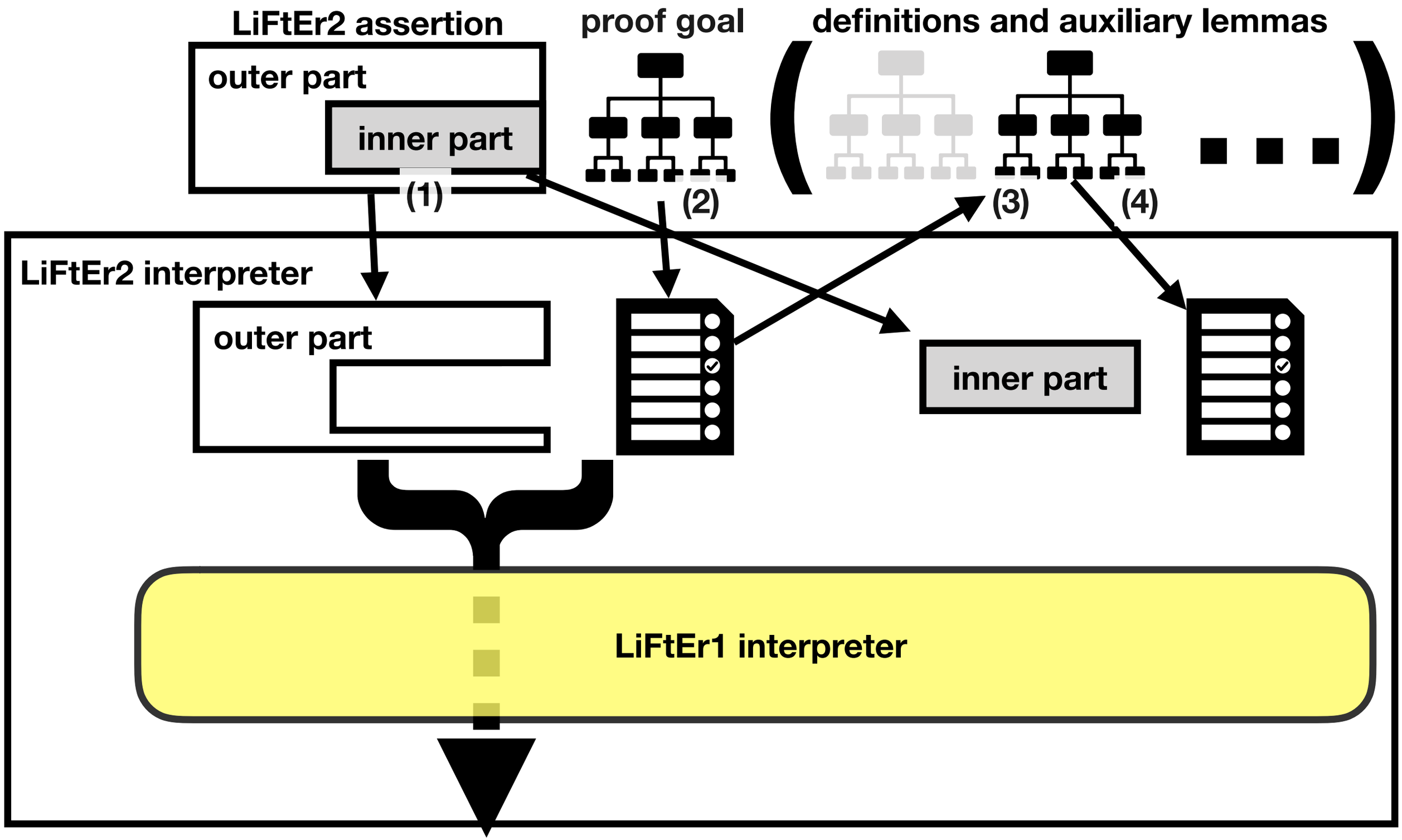
LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



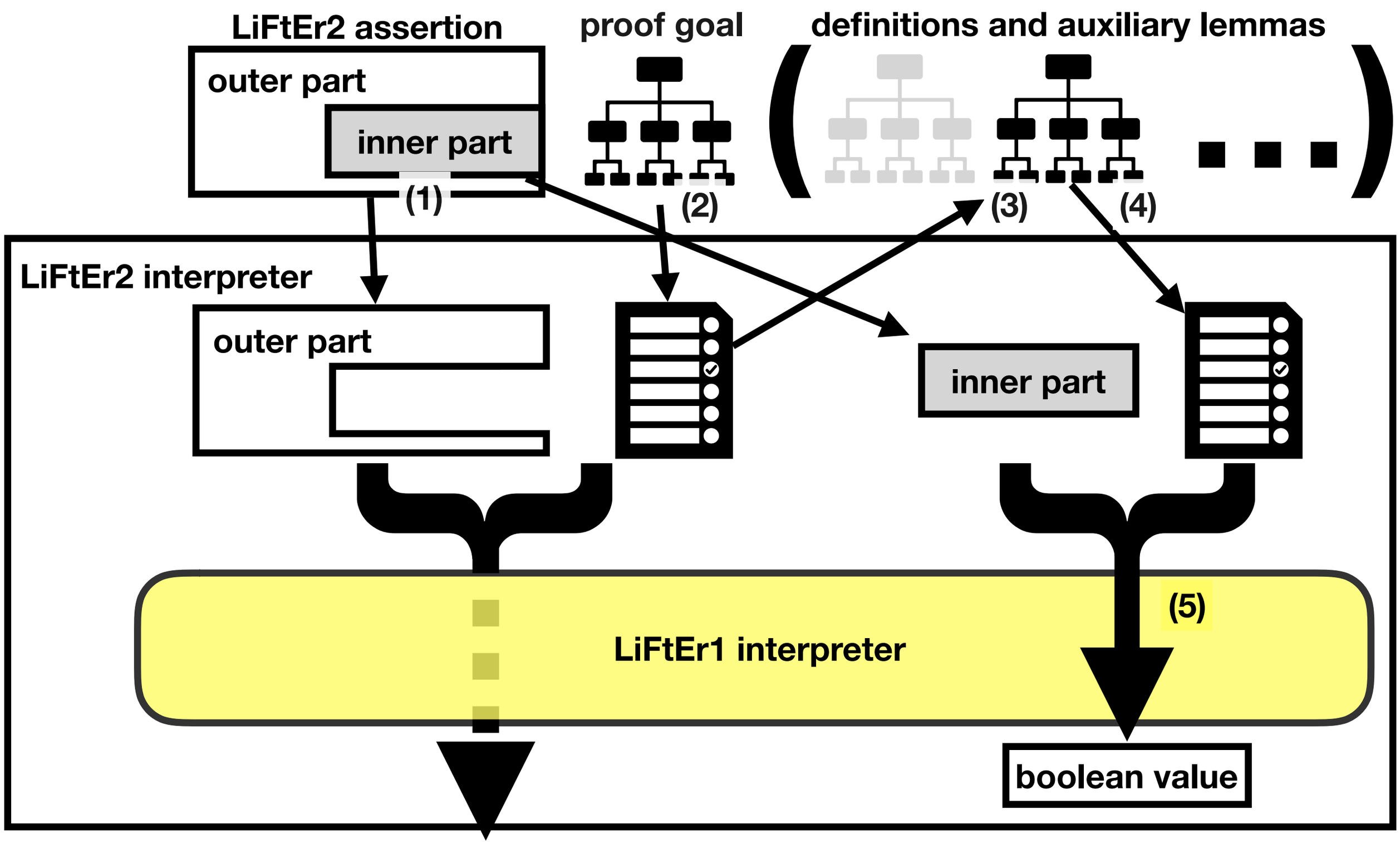
LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



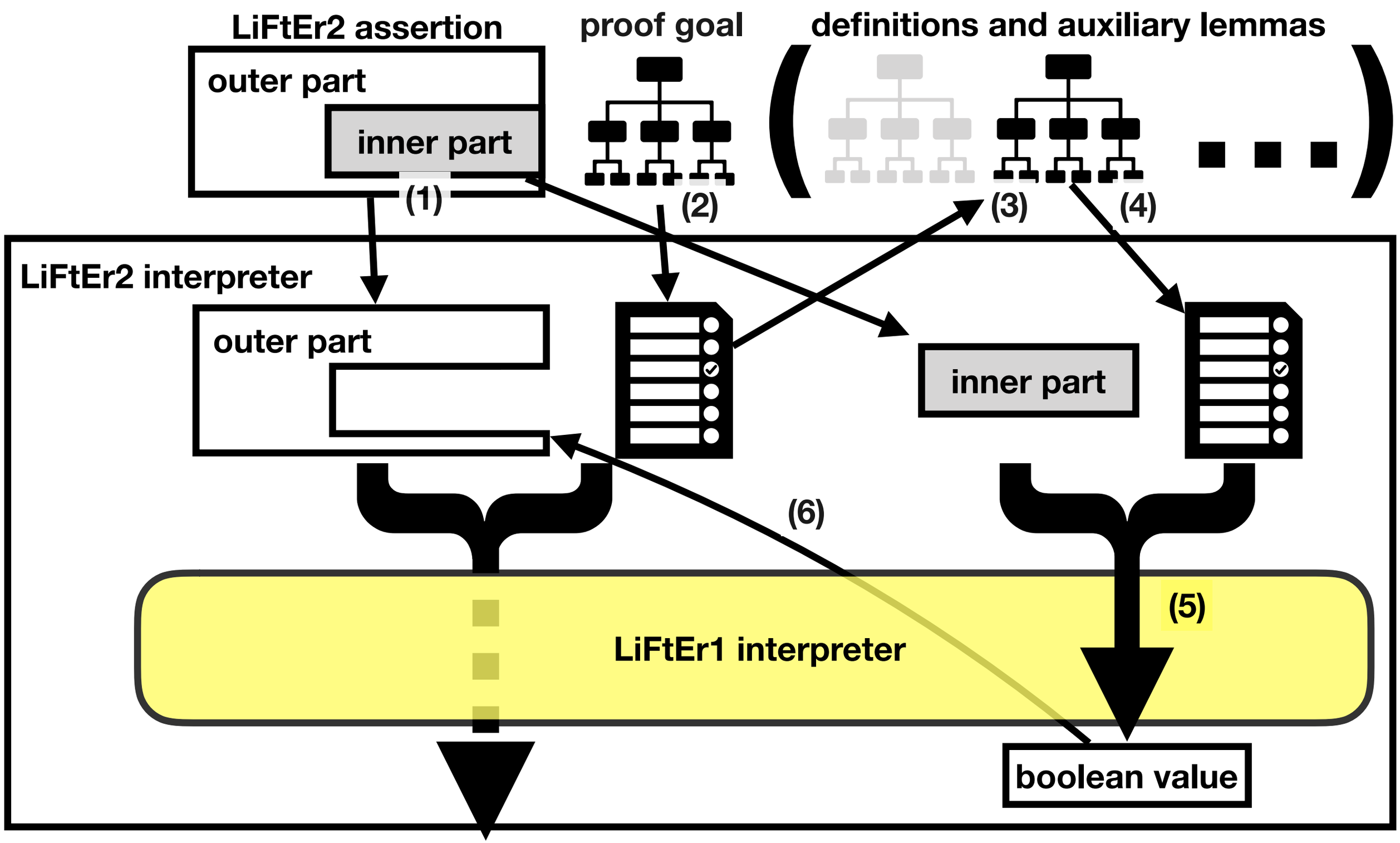
LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



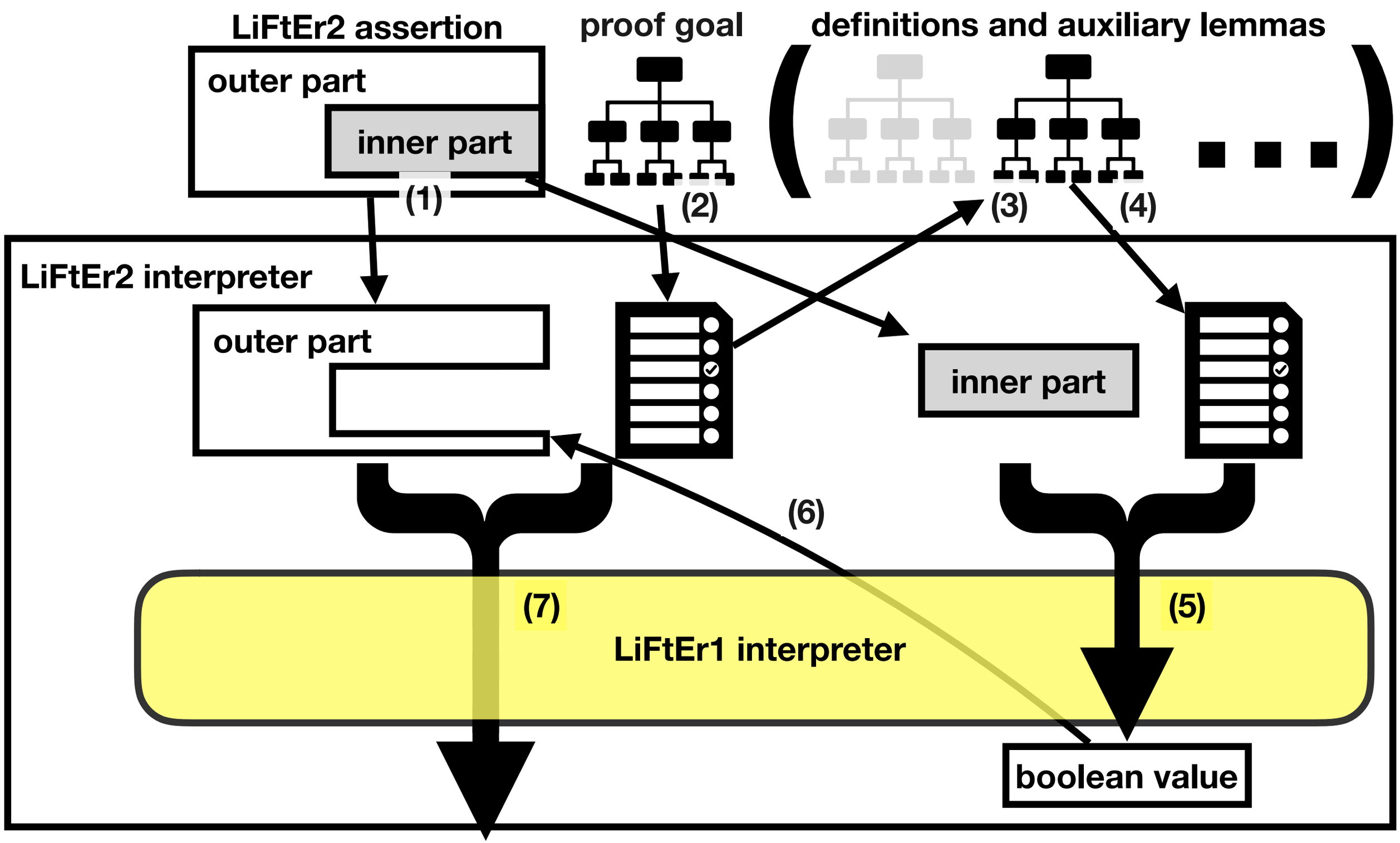
LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



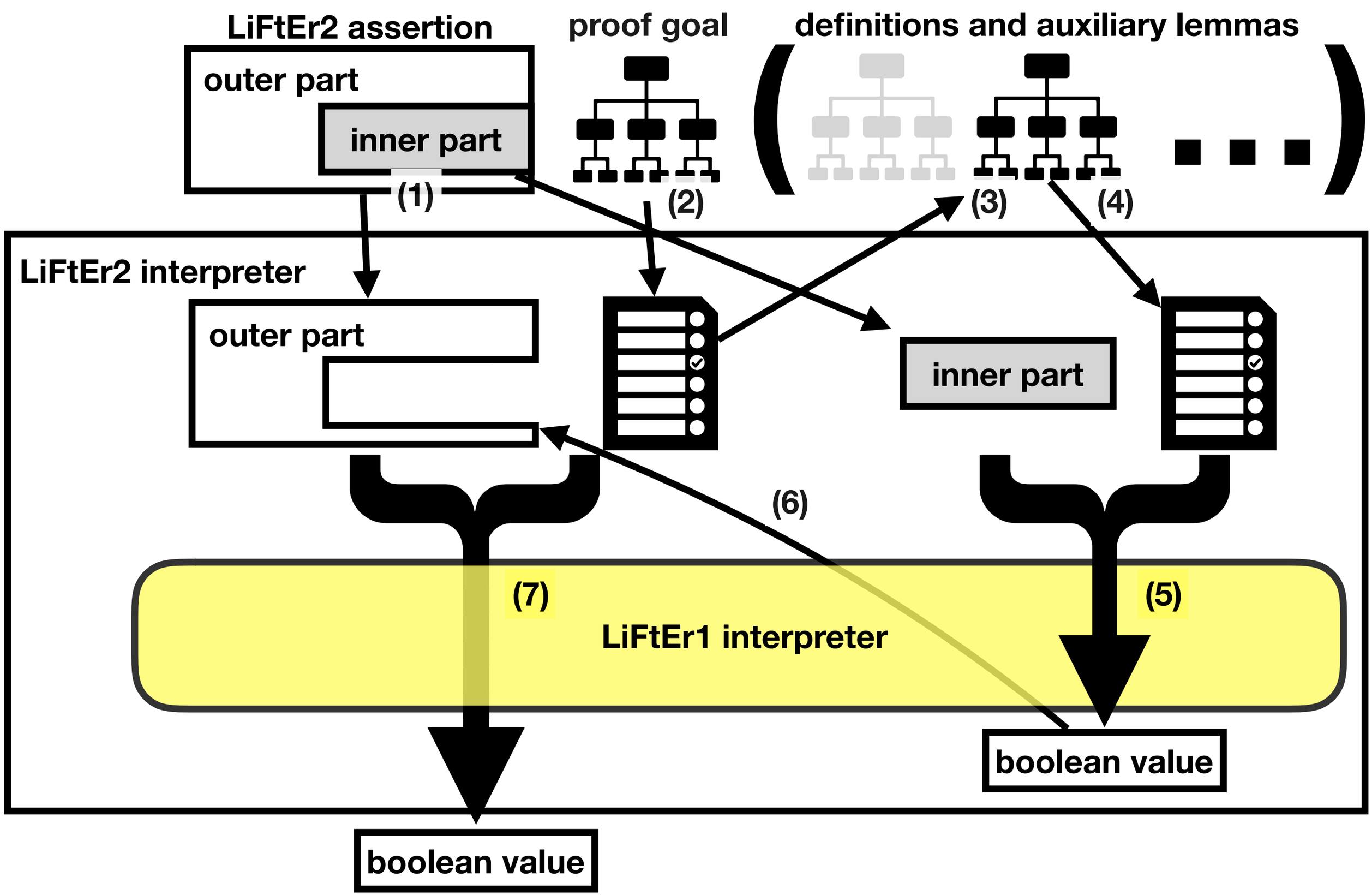
LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



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

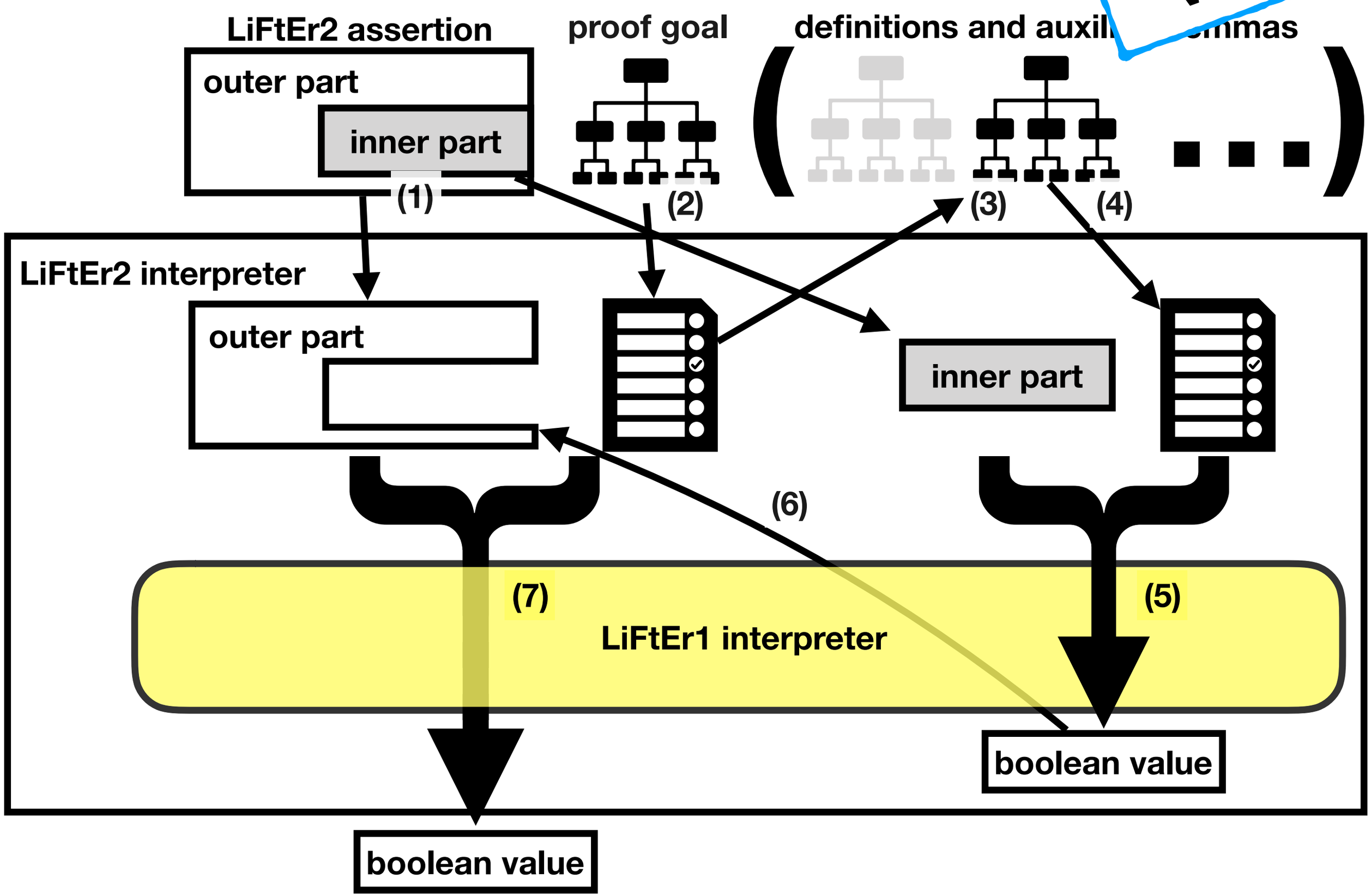


LiFtEr$_2$ (proof goal * induction arguments * relevant definitions) -> bool



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

WIP!



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

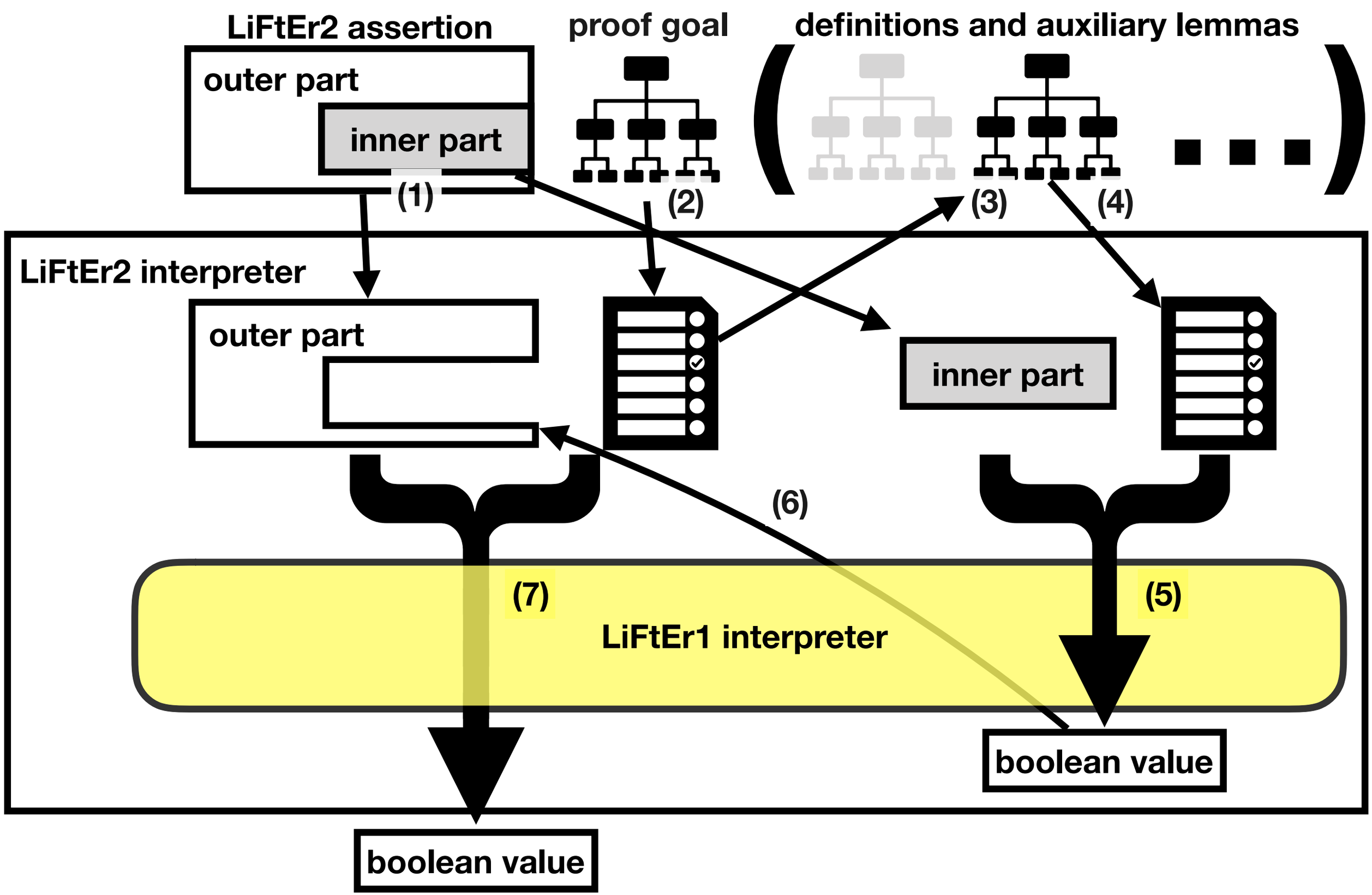
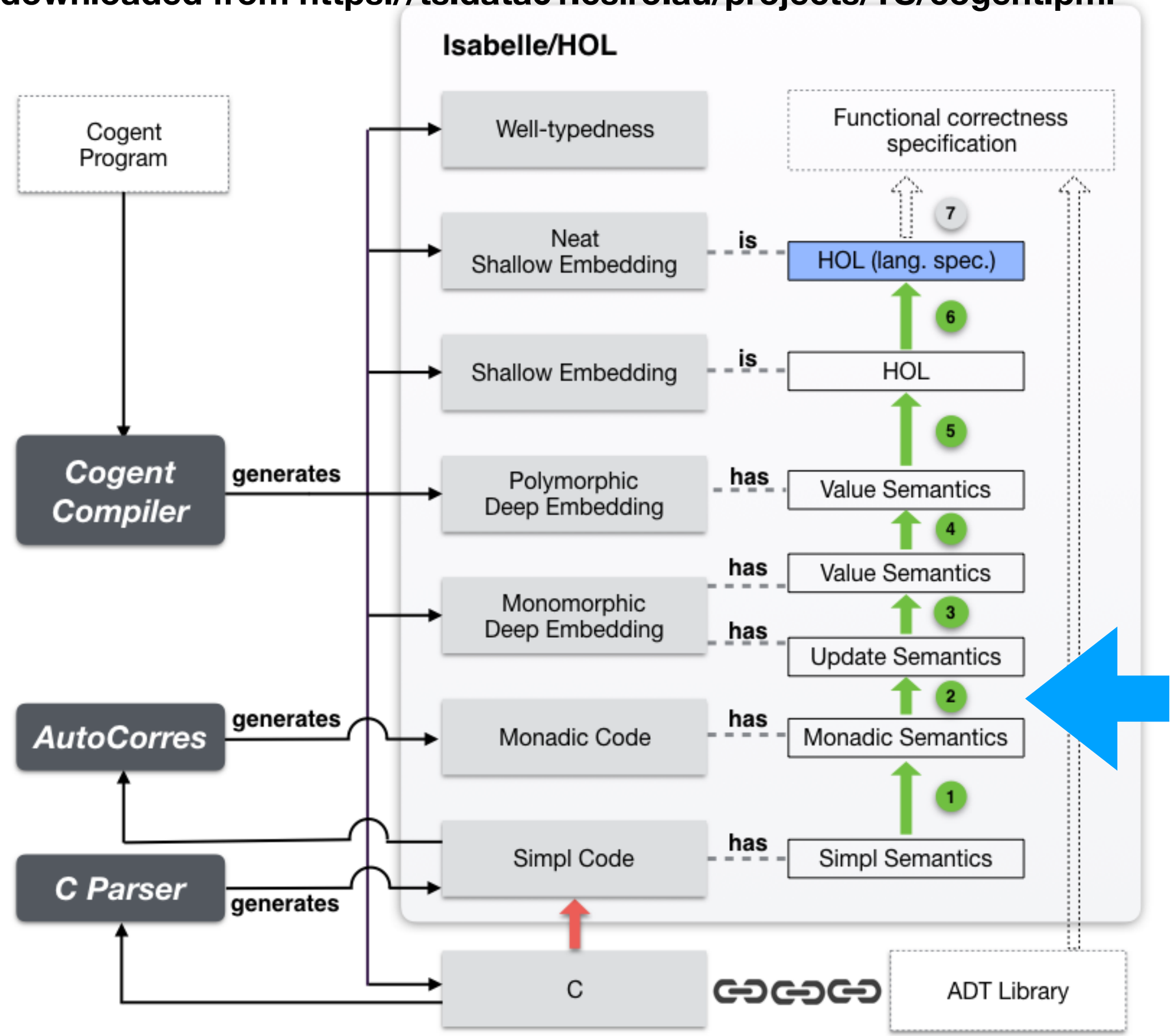
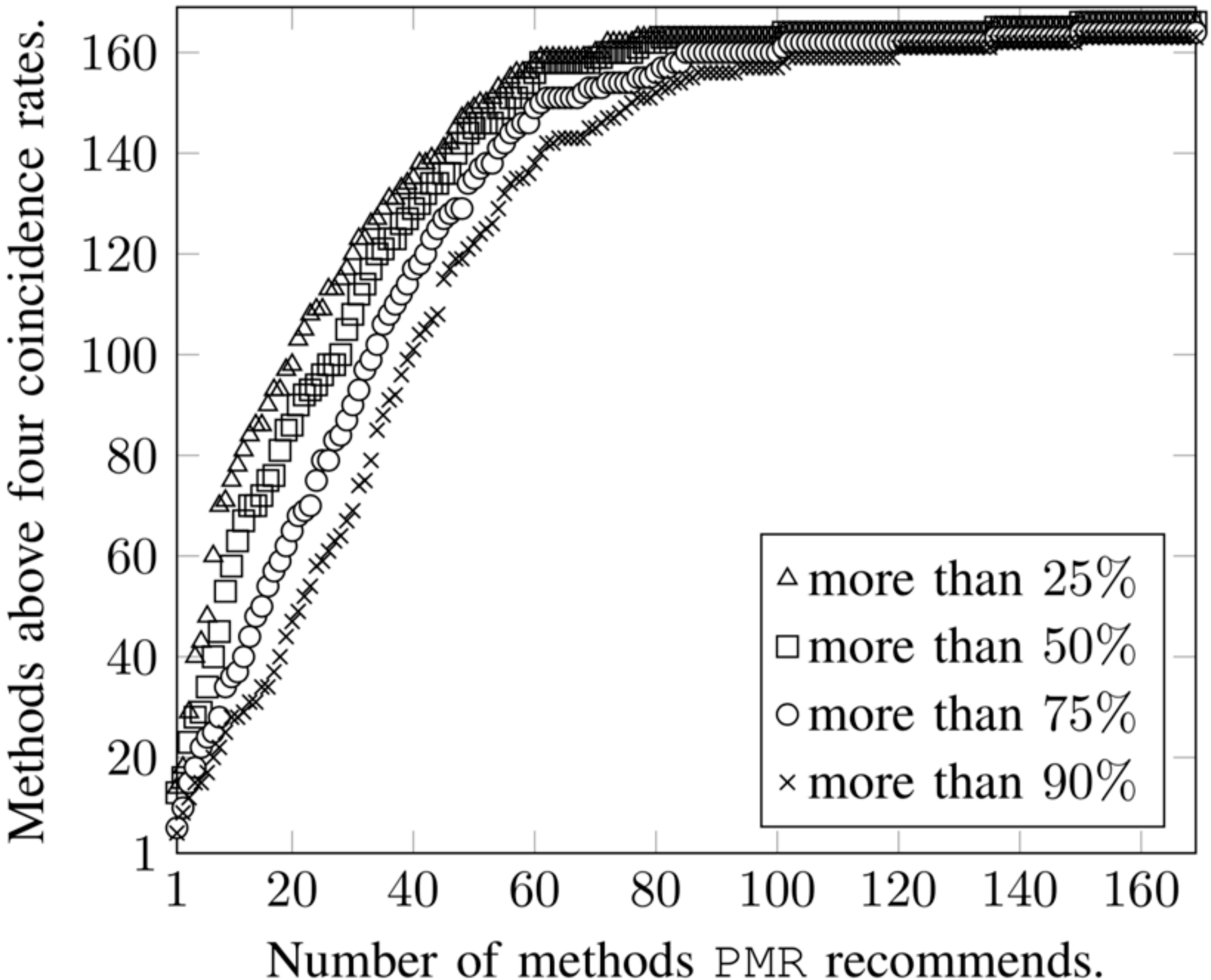


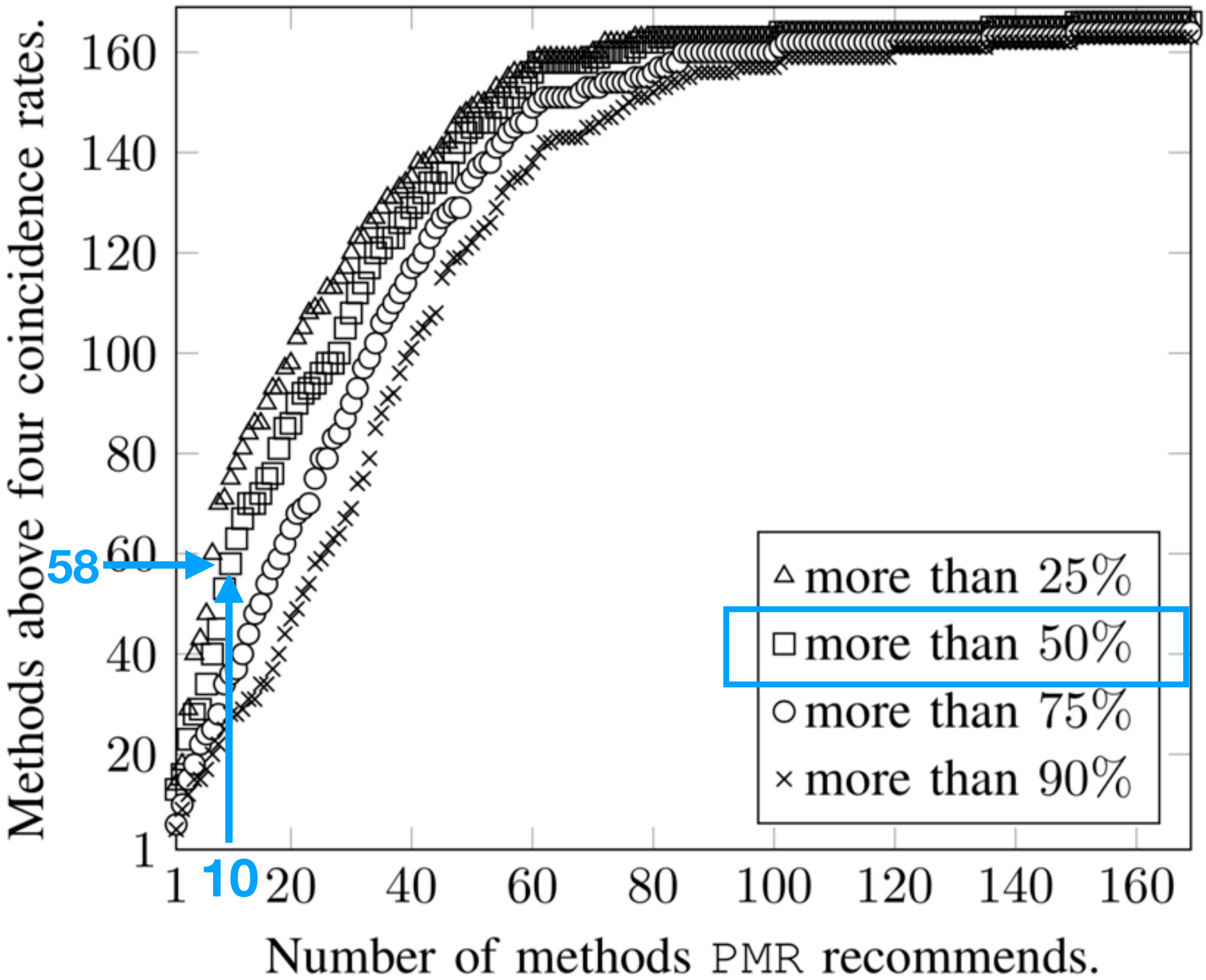
image downloaded from https://ts.data61.csiro.au/projects/TS/cogent.pml



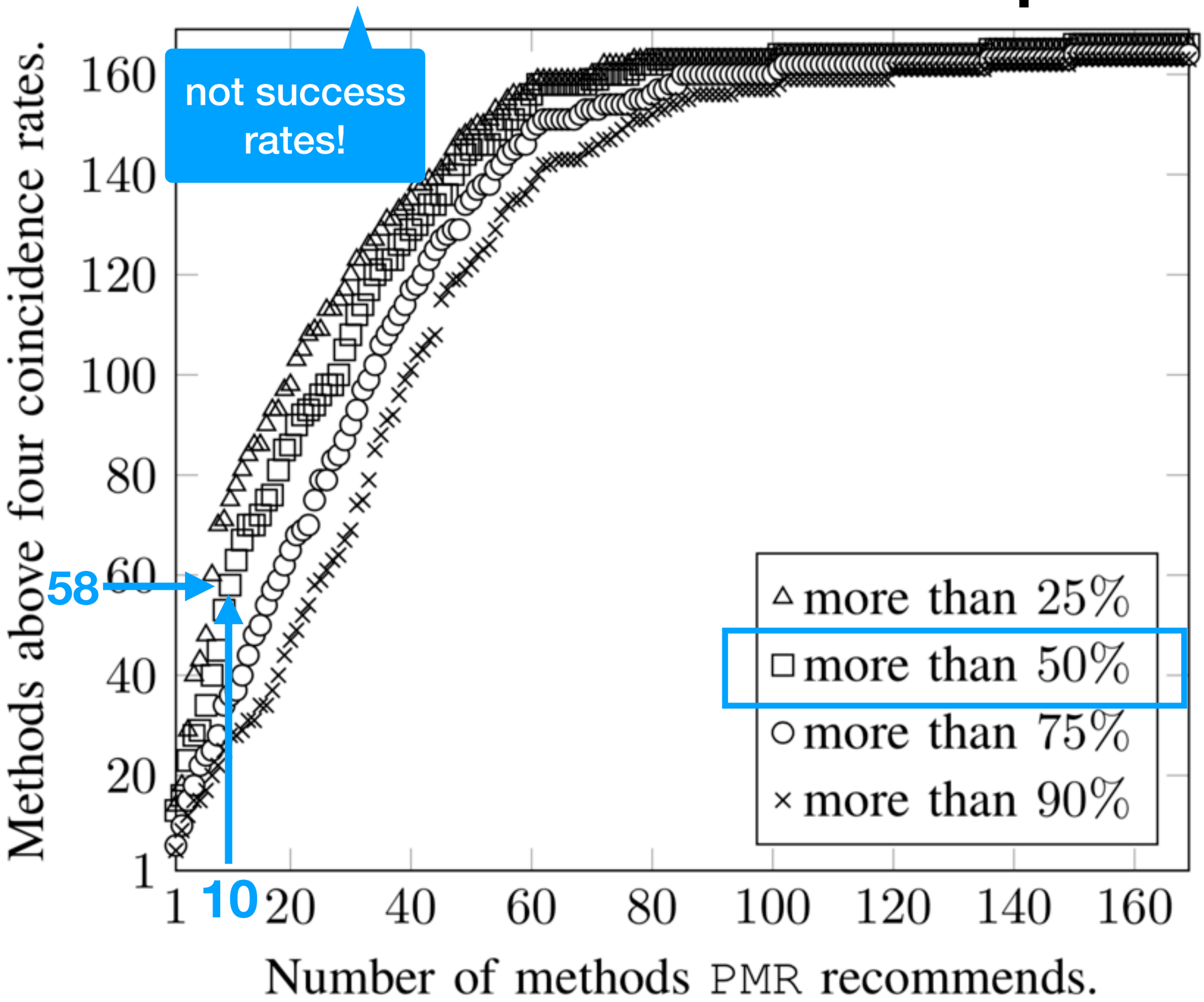
Coincidence rates of PaMpeR



Coincidence rates of PaMpeR



Coincidence rates of PaMpeR



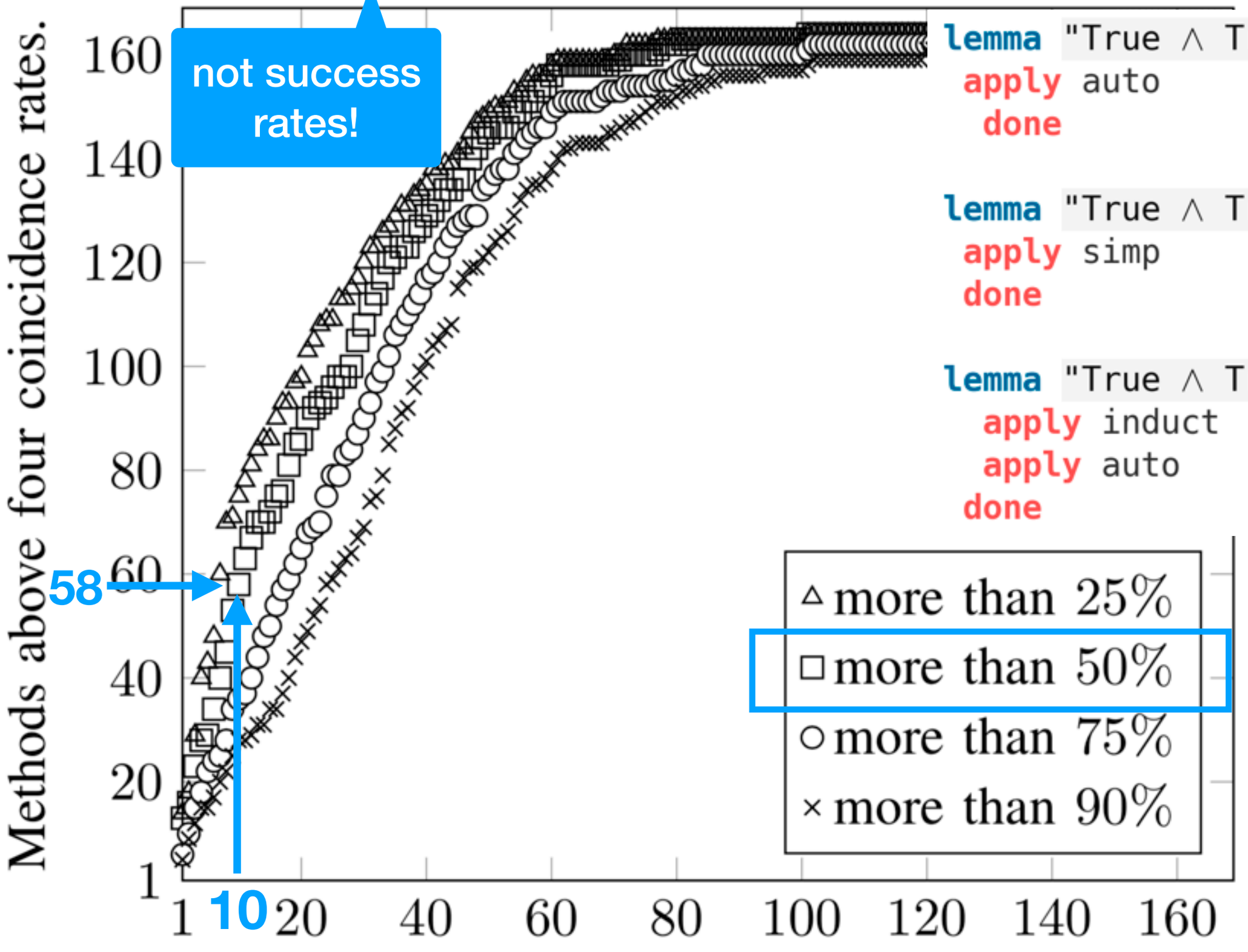
Coincidence rates of PaMpeR

not success rates!

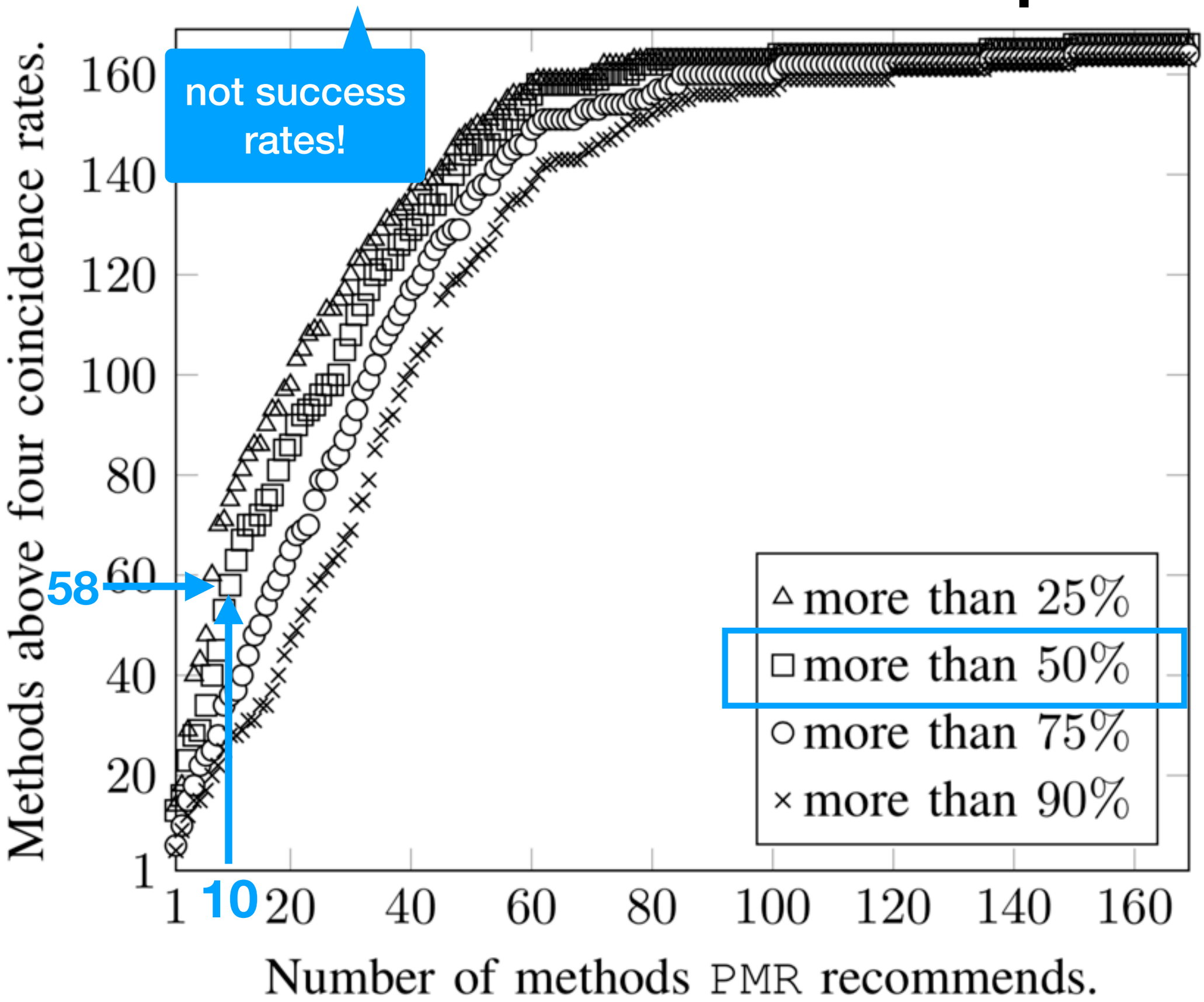
```
lemma "True ∧ True"
 apply auto
 done

lemma "True ∧ True"
 apply simp
 done

lemma "True ∧ True"
  apply induct
  apply auto
done
```



Coincidence rates of PaMpeR



Coincidence rates of PaMpeR