Artificial Intelligence and Domain-Specific Languages for Inductive Theorem Proving

Huawei 01.07.2021

Proof by induction in Isabelle?

Proof by induction in Isabelle?

When applying AI for theorem proving, keep in mind…

Proof by induction in Isabelle?

When applying AI for theorem proving, keep in mind…

Algorithmic approaches work well, but not always.

Proof by induction in Isabelle?

When applying AI for theorem proving, keep in mind…

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Proof by induction in Isabelle?

When applying AI for theo

Algorithmic approaches wor

There are many good ways t

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
  apply(induct xs ys rule: rev2.induct)
  apply auto done
```

Proof by induction in I

```
fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
  apply(induct xs arbitrary: ys)
  apply auto done
```

Proof state Auto update Update Search:

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

Output Query Sledgehammer Symbols

15,13 (344/353) (isabelle,isabelle,U

When applying AI for theo

Algorithmic approaches wor

There are many good ways t

```
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
  apply(induct xs ys rule: rev2.induct)
  apply auto done
```

Proof by induction in Isabelle?

When applying AI for theorem proving, keep in mind…

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Proof by induction in Isabelle?

When applying AI for theorem proving, keep in mind…

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

Proof by induction in Isabelle?

When applying AI for theorem proving, keep in mind…

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

Path to automatic induction

Path to automatic induction

Step 1: Guided backtracking search with tracing (PSL).

Step 2: Logical Feature Extraction (LiFtEr).

Step 3: smart_induct using LiFtEr.

Step 4: Putting things together.

Proof Strategy Language (CADE2017)

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

```
fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"
```

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

```
consts
  rev2 :: "'a list ⇒ 'a list ⇒ 'a list"
Found termination order: "(λp. length (fst p)) <*mlex*> {}"
```

Output | Query | Sledgehammer | Symbols

12,1 (262/400) (isabelle,isabelle,UTF-8-Isabelle) 287/512MB 1 error(s) 2:11 PM

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

```
fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
```

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

```
proof (prove)
goal (1 subgoal):
 1. rev2 xs ys = rev1 xs @ ys
```

Output | Query | Sledgehammer | Symbols

15,36 (300/402) Input/output complete (isabelle,isabelle,UTF-8-Isabelle) 1 error(s) 2:13 PM

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

```
fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

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

theorem "rev2 xs ys = rev1 xs @ ys"
```

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

Output | Query | Sledgehammer | Symbols

13,64 (326/409) (isabelle,isabelle,UTF-8-Isabelle) 119/512MB 1 error(s) 2:14 PM

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

```
fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

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

theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd
  oops
```

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

Output | Query | Sledgehammer | Symbols

16,18 (381/393) (isabelle,isabelle,UTF-8-Isabelle) 95/512MB 1 error(s) 2:15 PM

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

```
9  fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
10   "rev2  []       ys = ys"
11 | "rev2 (x # xs) ys = rev2 xs (x # ys)"
12
13 strategy DInd = Thens [Dynamic (Induct), Auto, IsSolved](*PSL*)
14
15 theorem "rev2 xs ys = rev1 xs @ ys"
16   find_proof DInd
17   oops
18
```

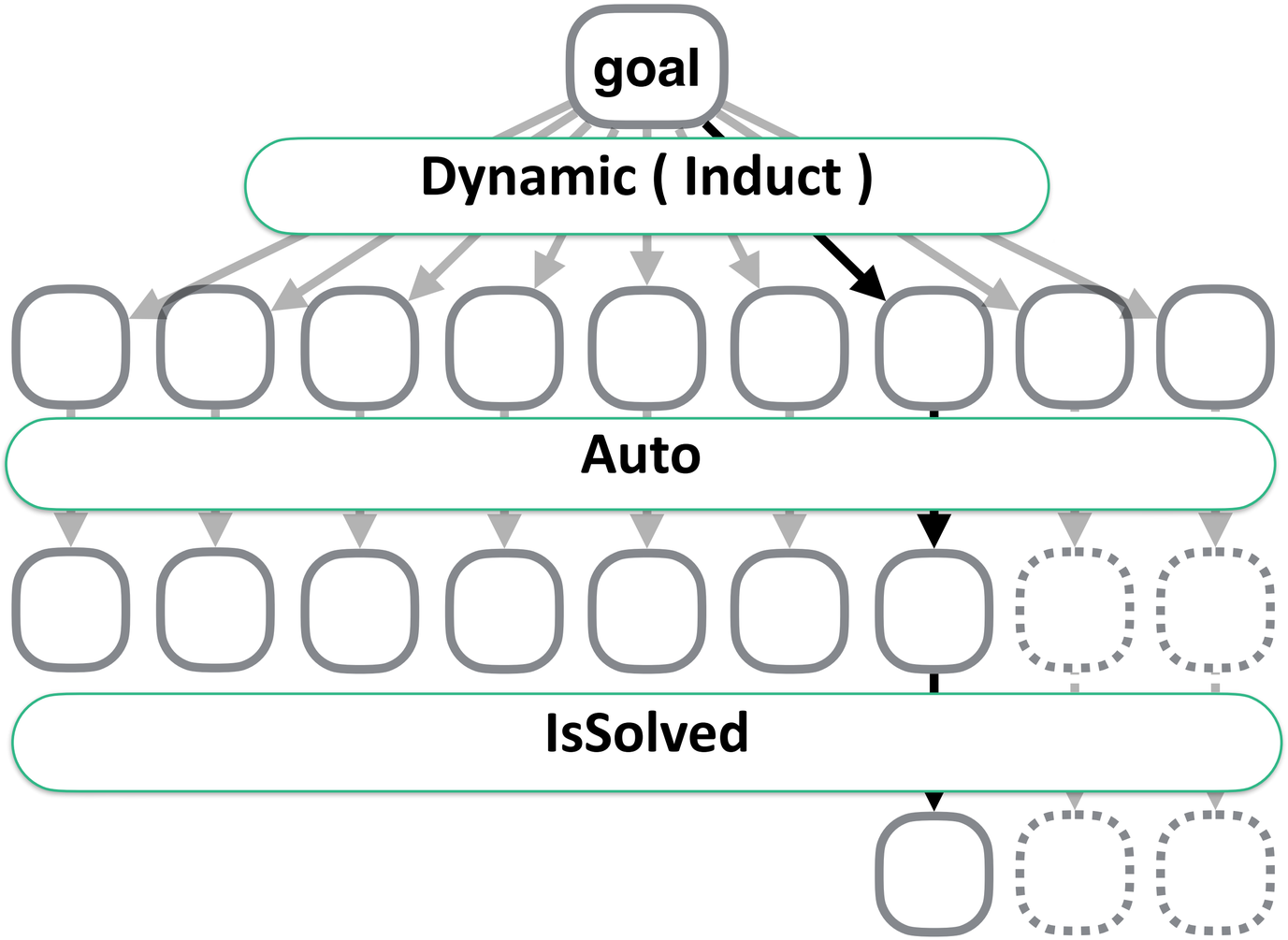
Proof state | Auto update | Update | Search: | 100%

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

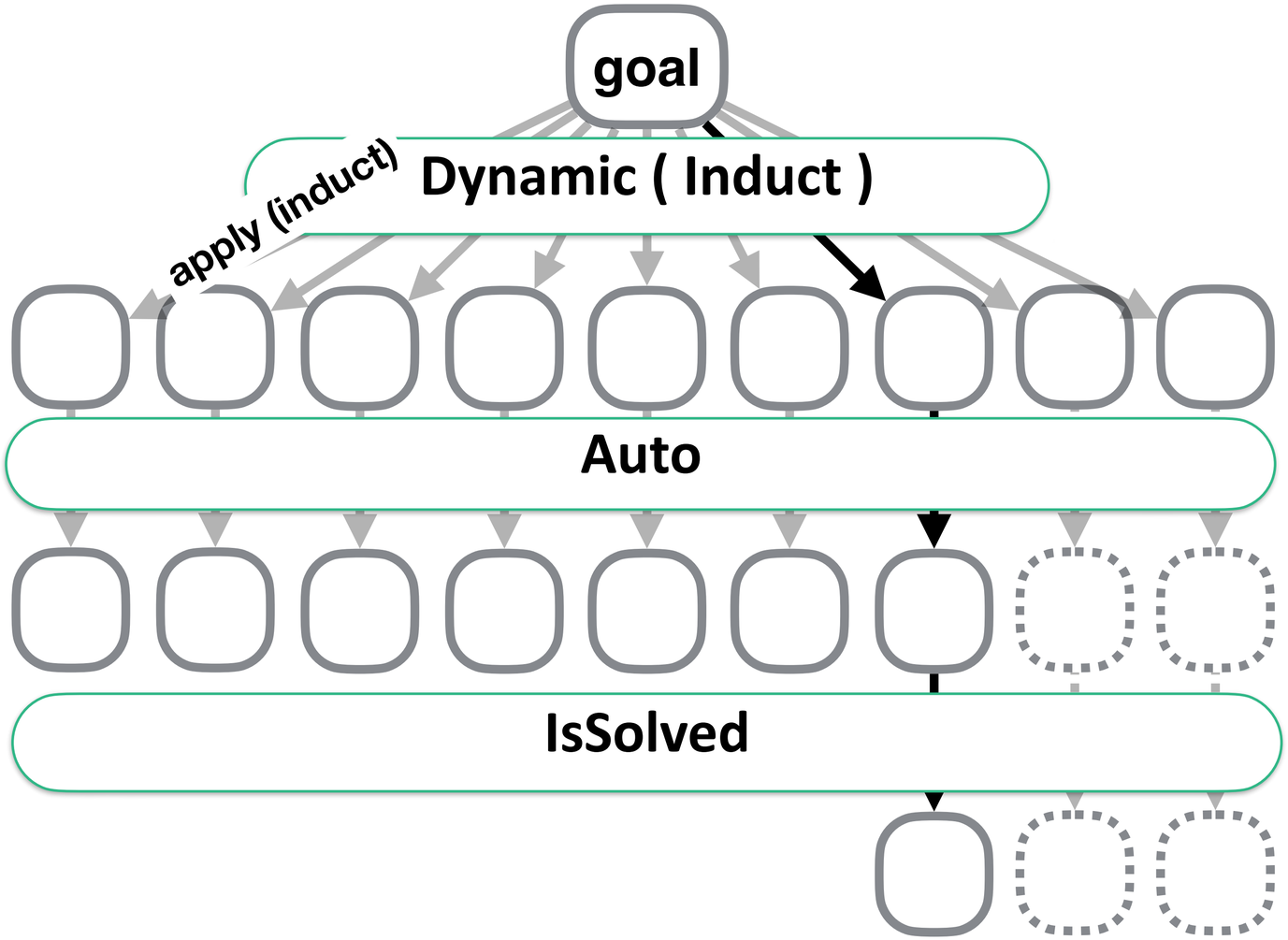
Output | Query | Sledgehammer | Symbols

16,18 (381/393) (isabelle,isabelle,UTF-8-Isabelle) U.. 195/512MB 1 error(s) 2:14 PM

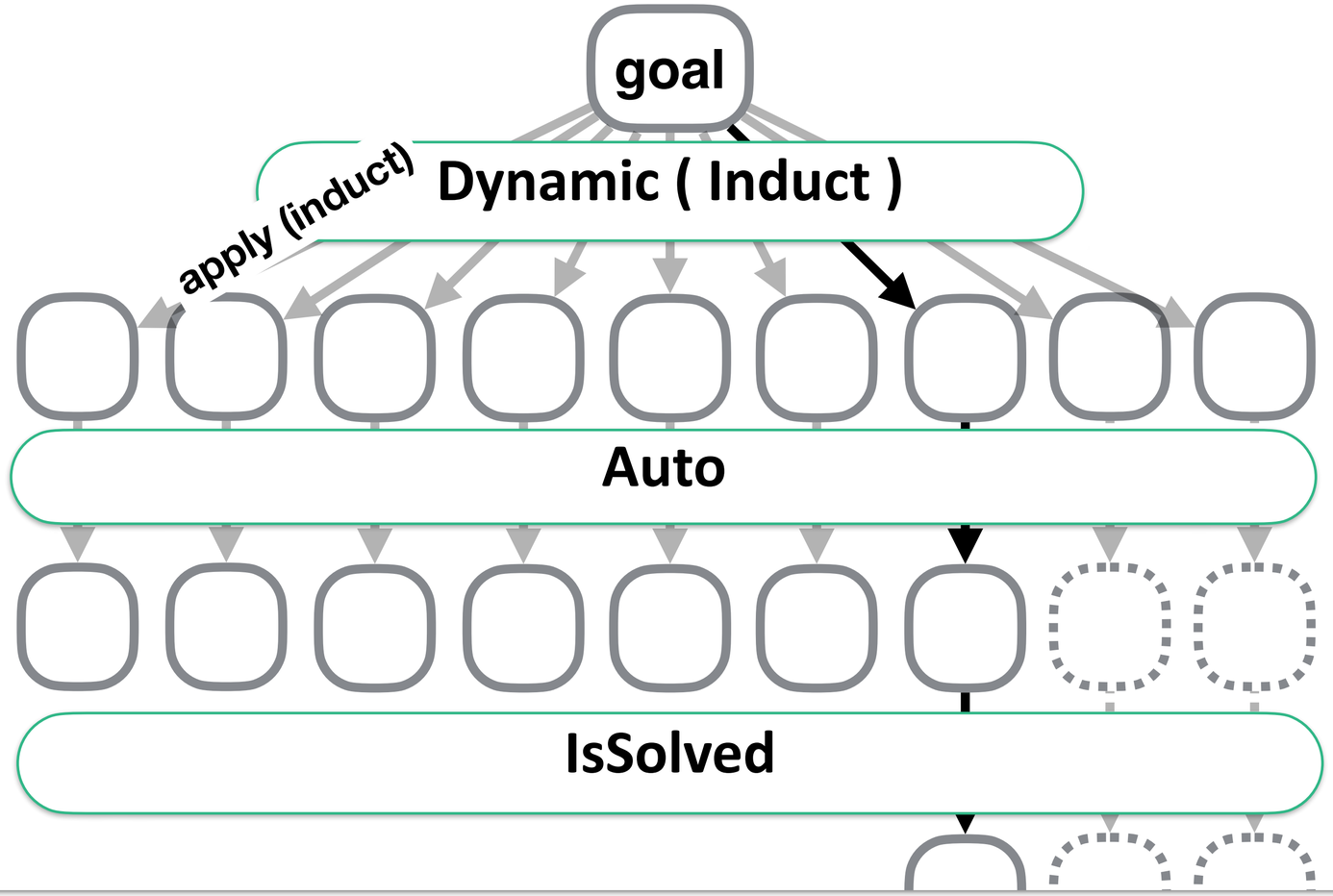
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```

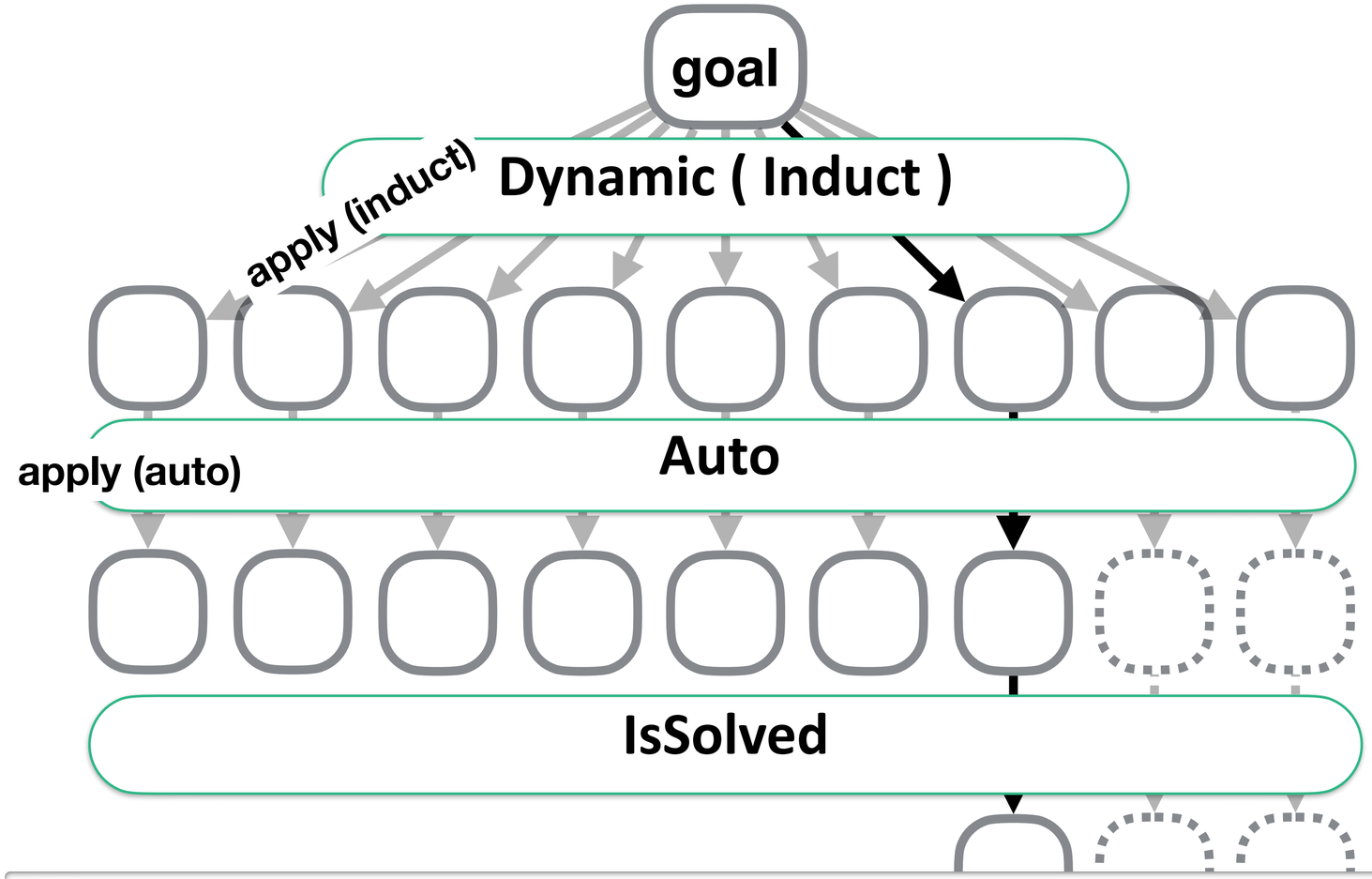


```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



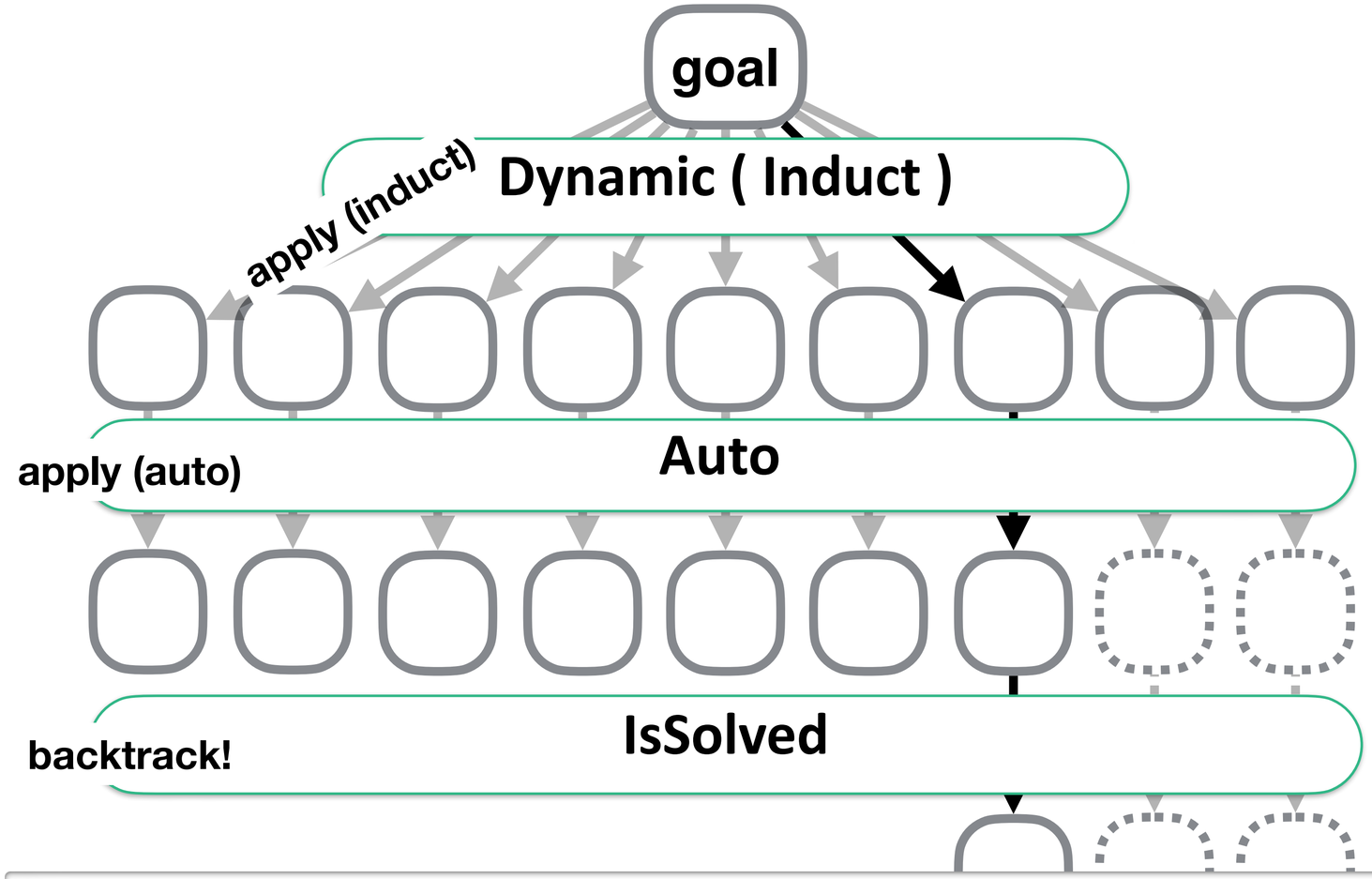

```
goal (1 subgoal):
 1. ⋀y. y ∈ {F. is_filter F} ⟹ rev2 xs ys = rev1 xs @ ys
```

```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



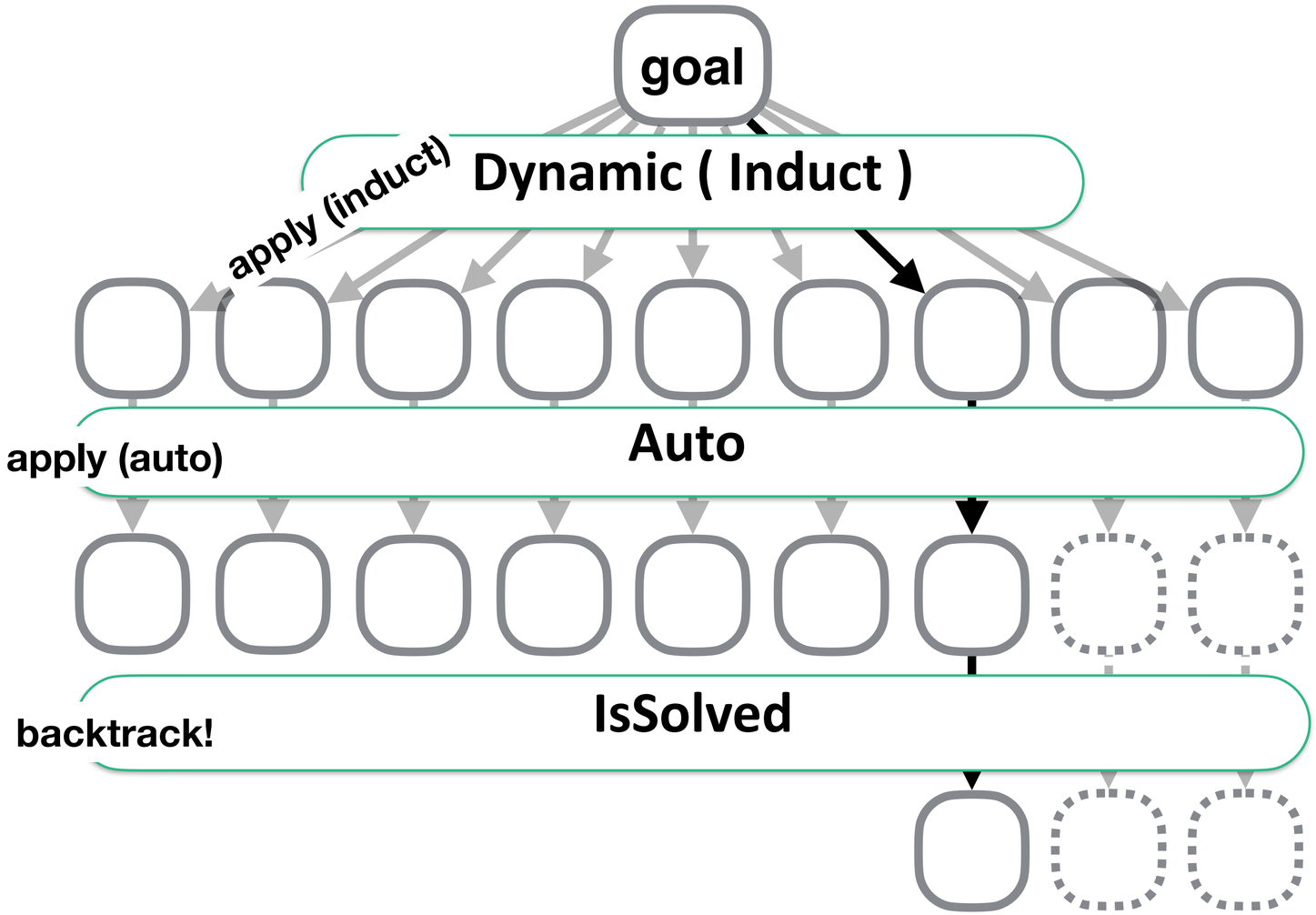

```
goal (1 subgoal):
 1. ⋀y. is_filter y ⟹ rev2 xs ys = rev1 xs @ ys
```

```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```

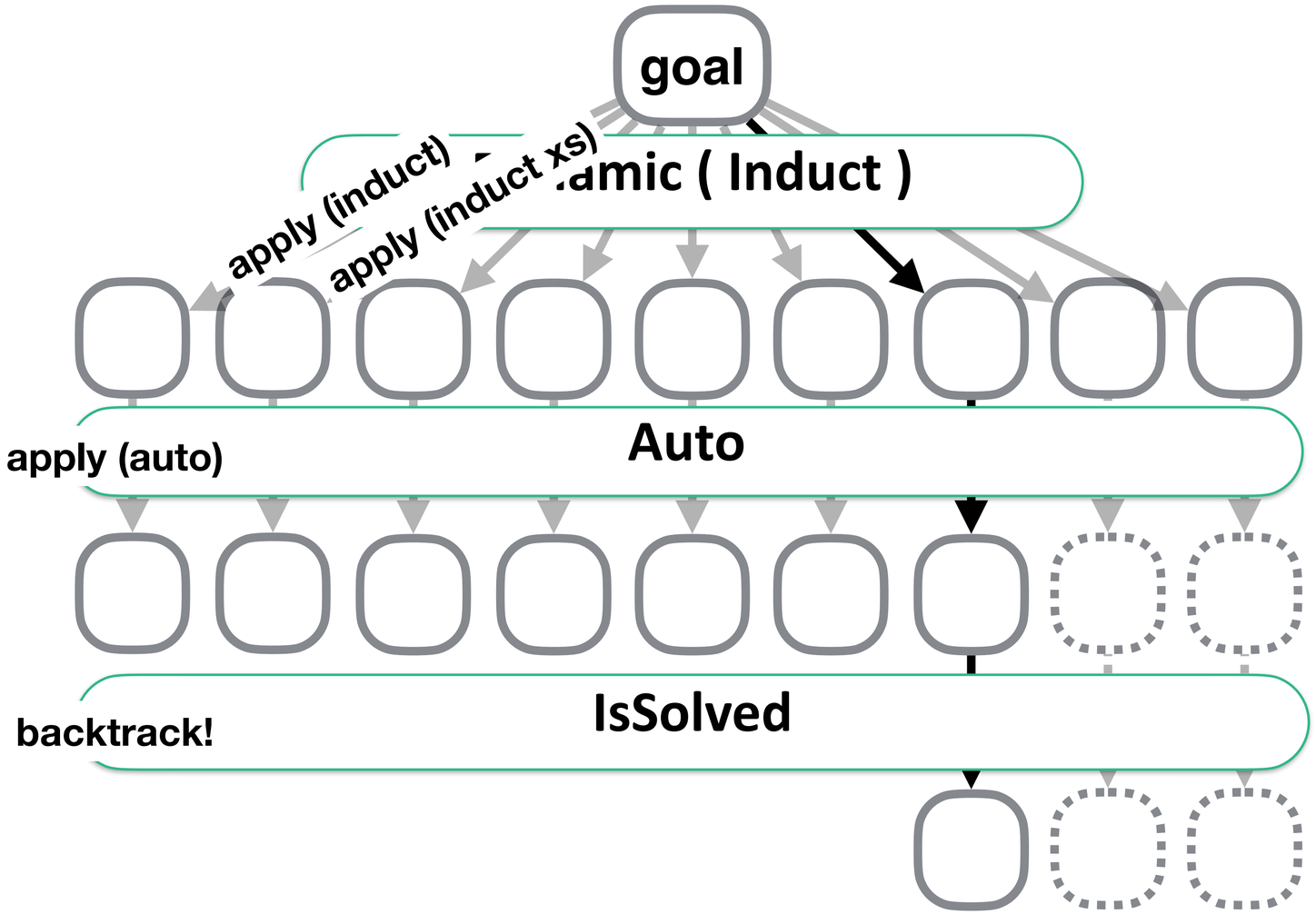



```
goal (1 subgoal):
 1. ⋀y. is_filter y ⟹ rev2 xs ys = rev1 xs @ ys
```

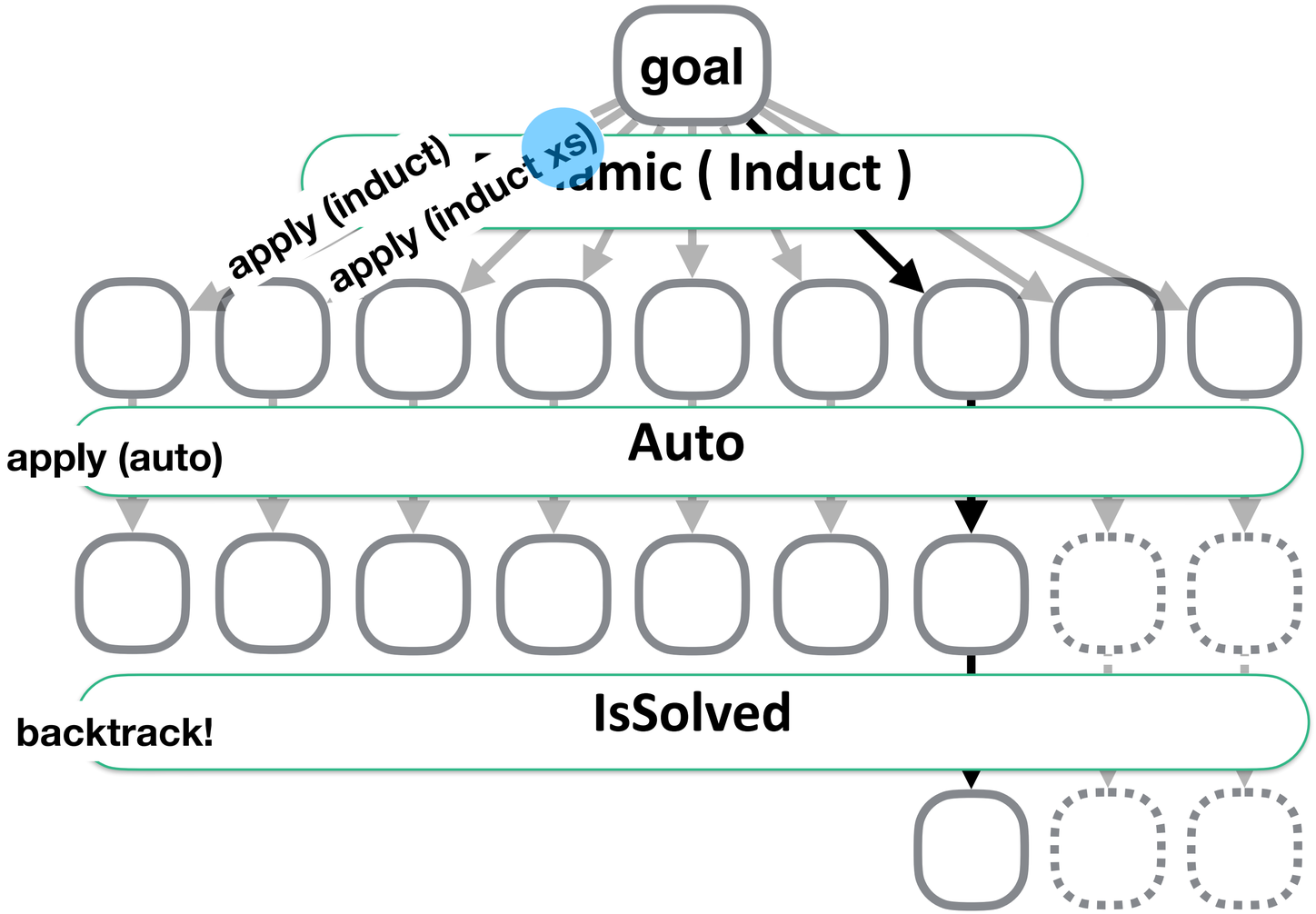
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



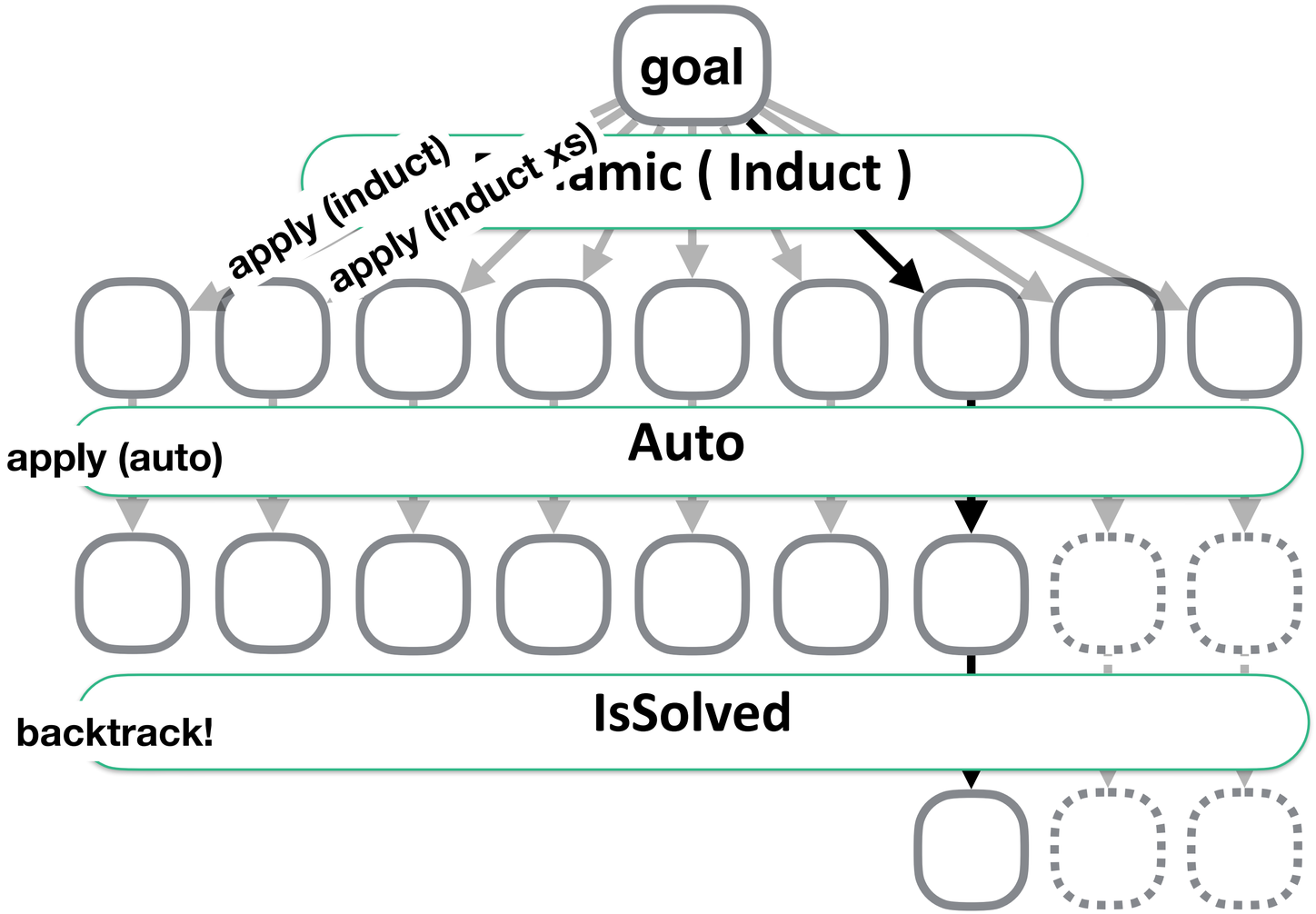
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



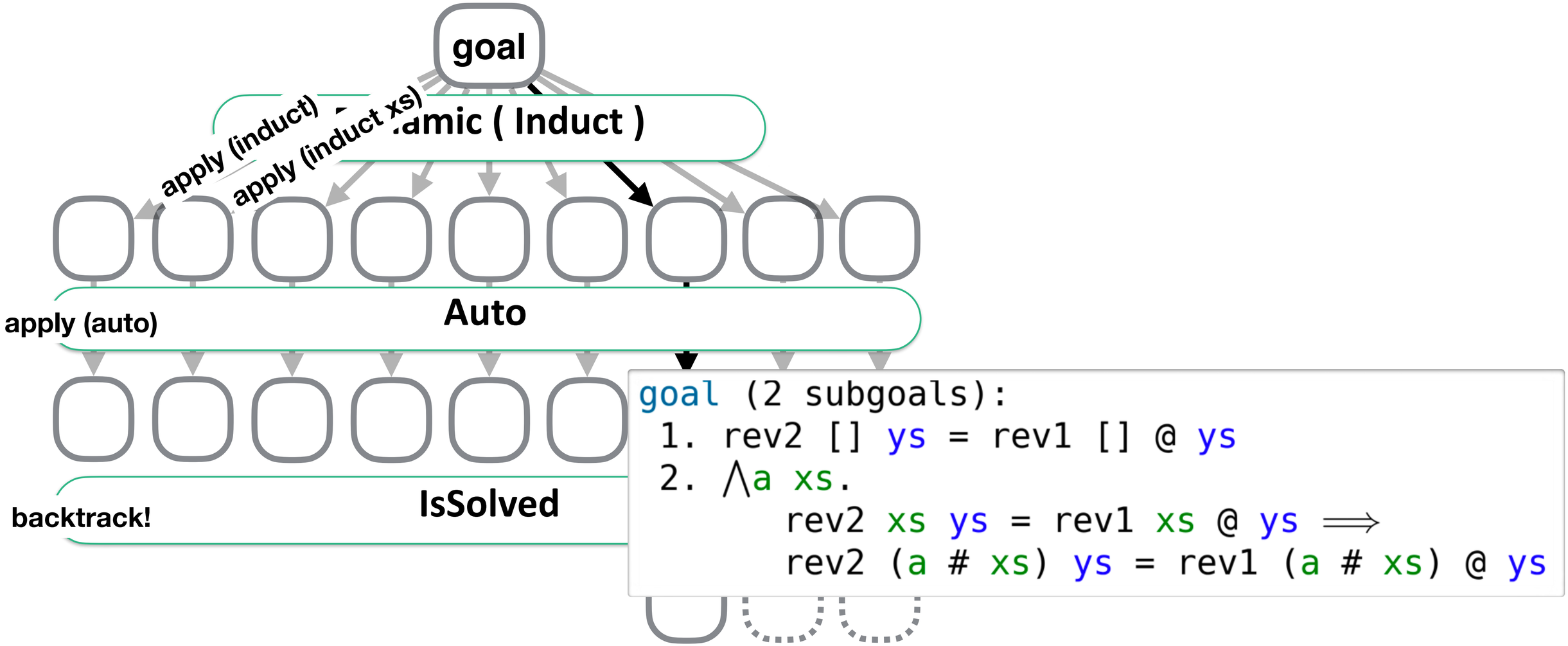
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



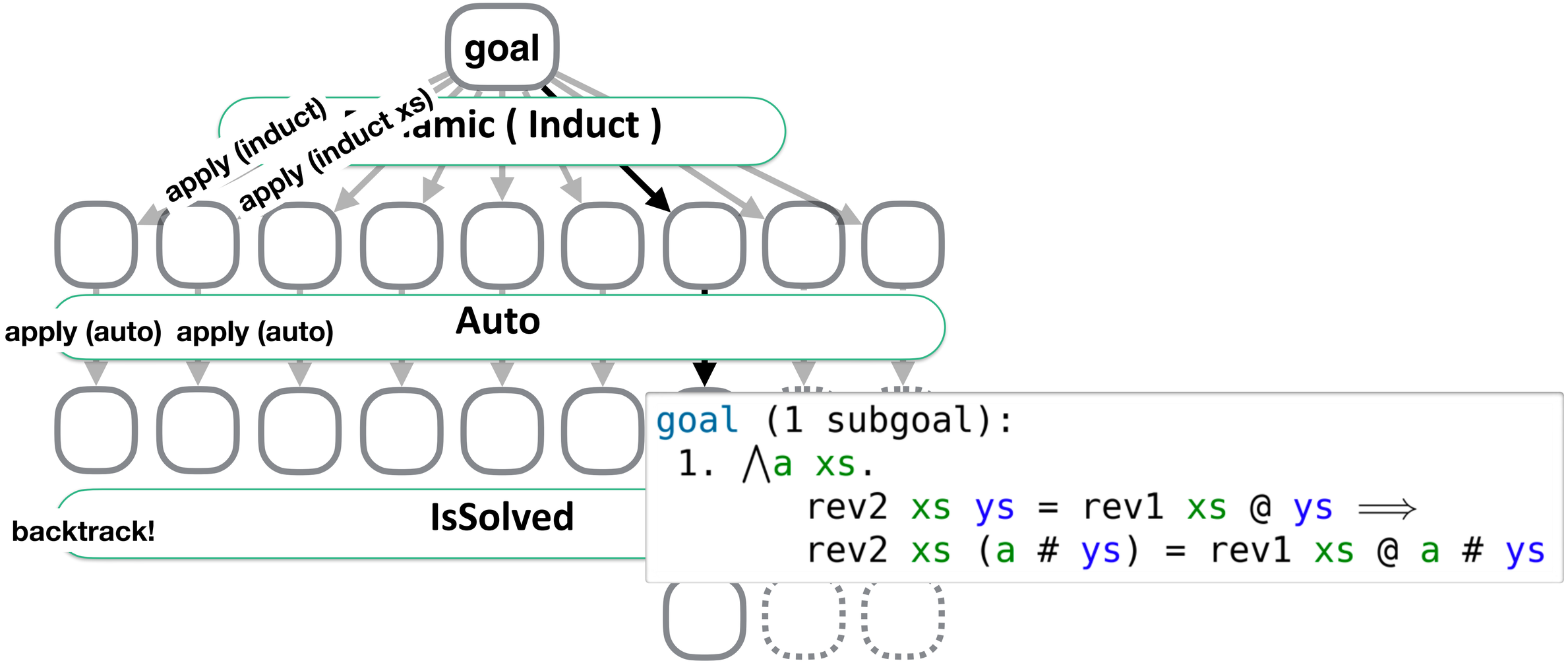
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



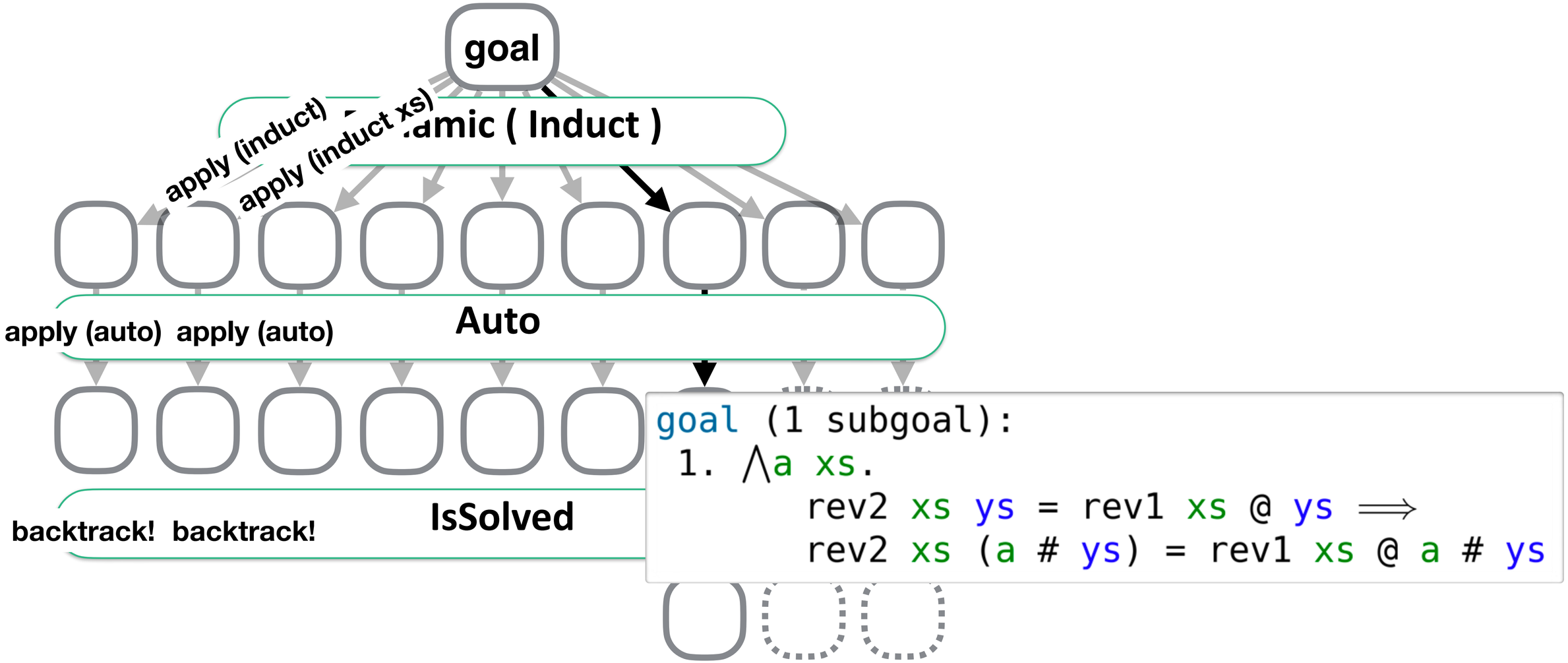
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



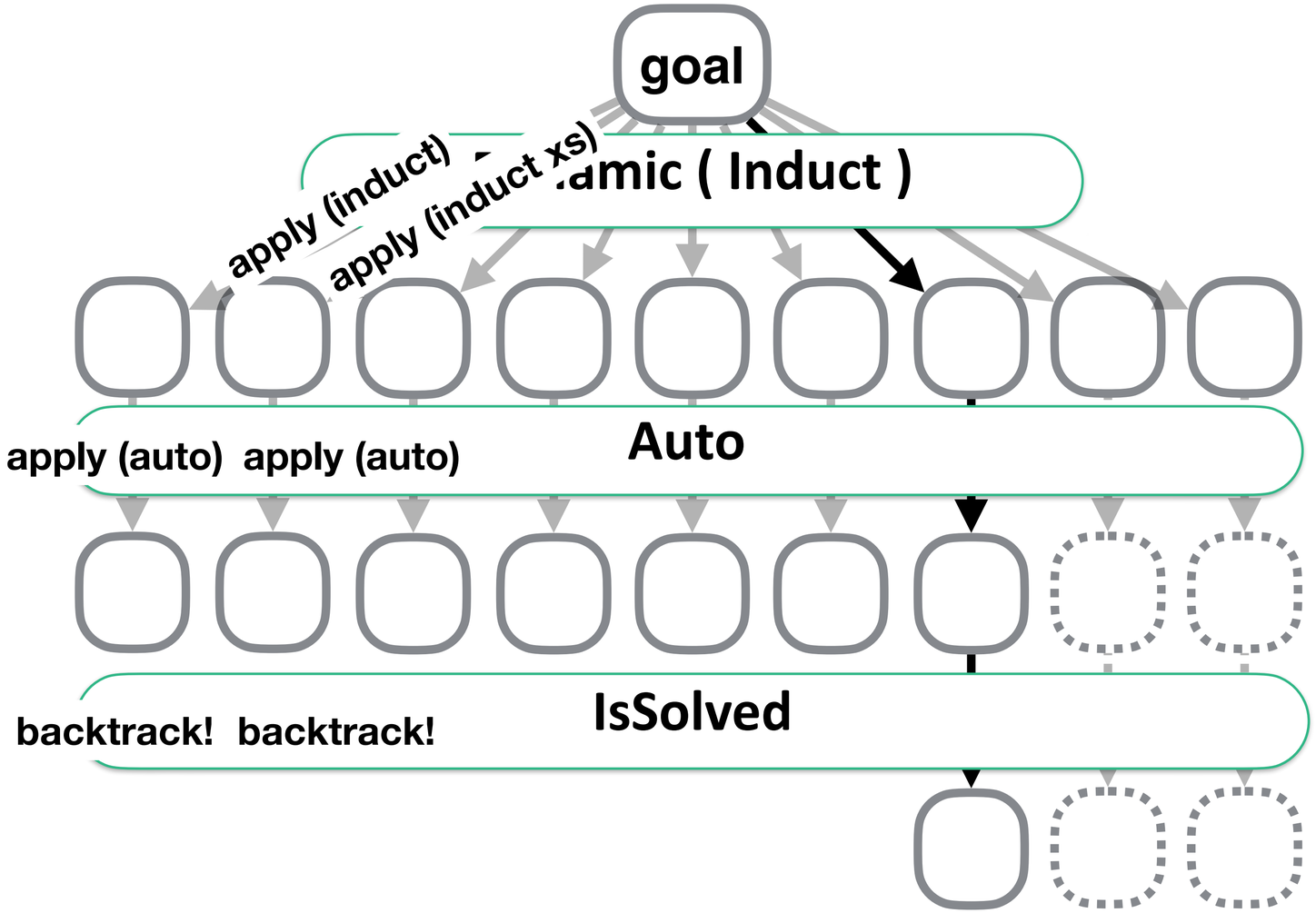
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



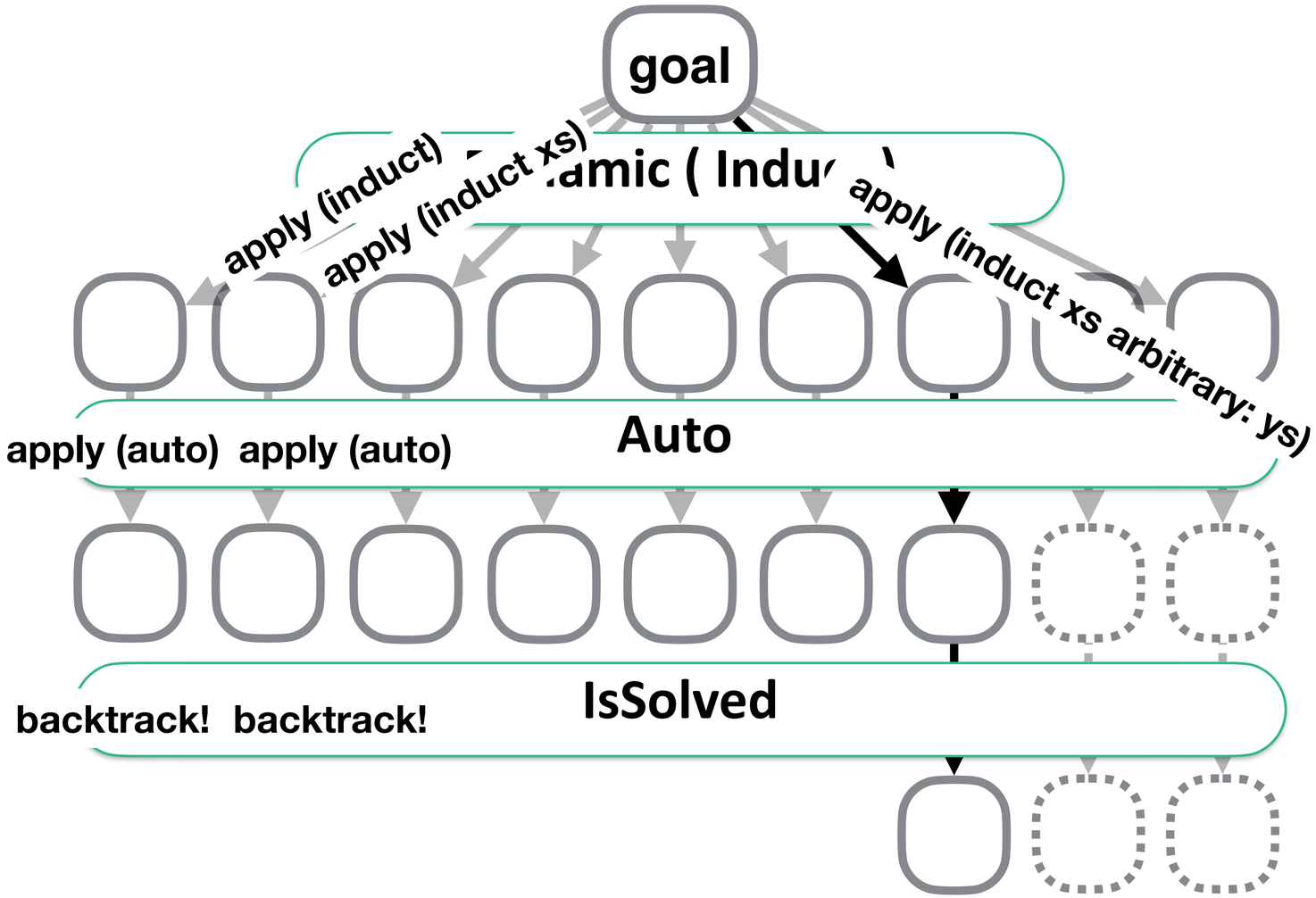
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



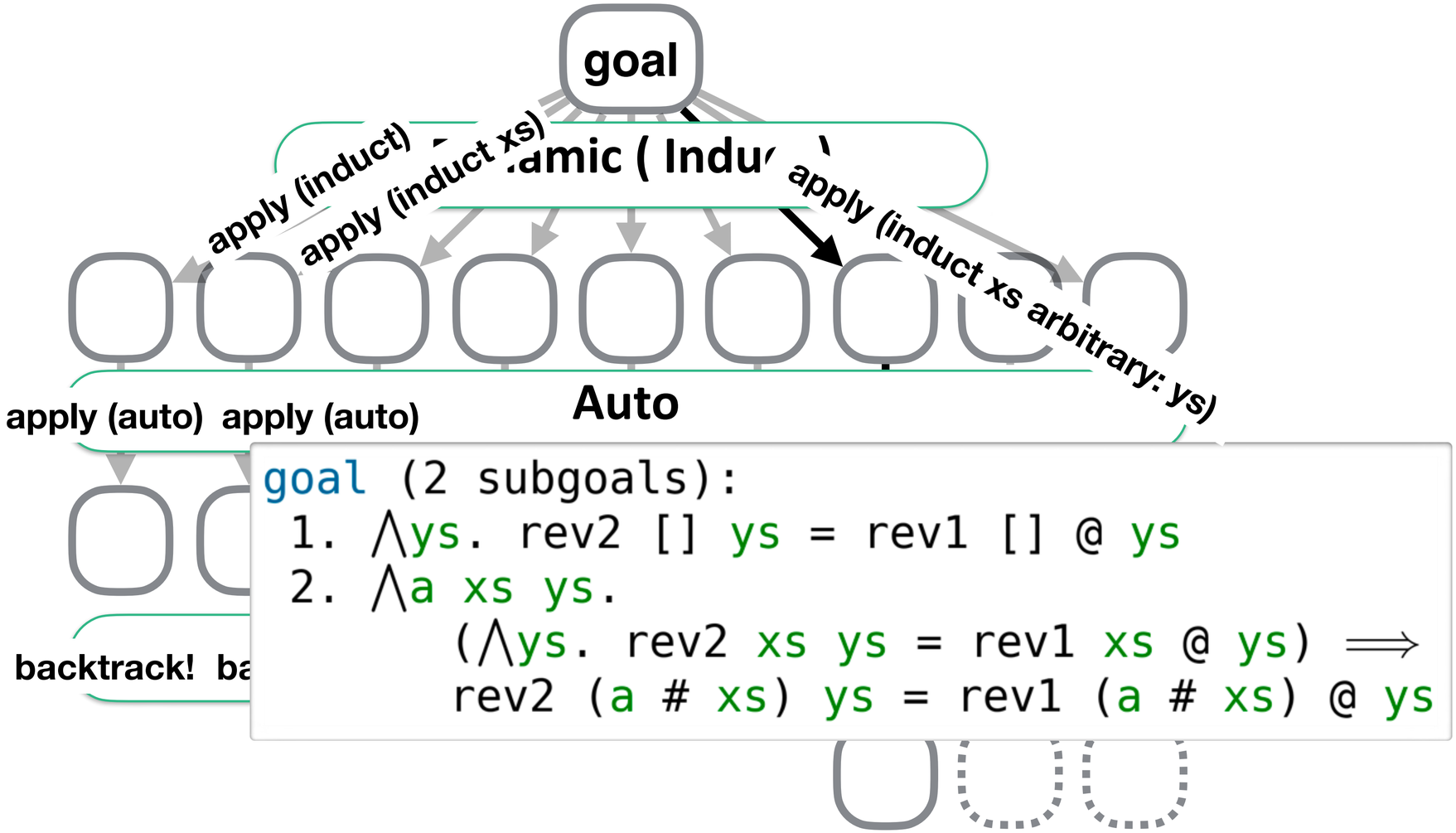
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



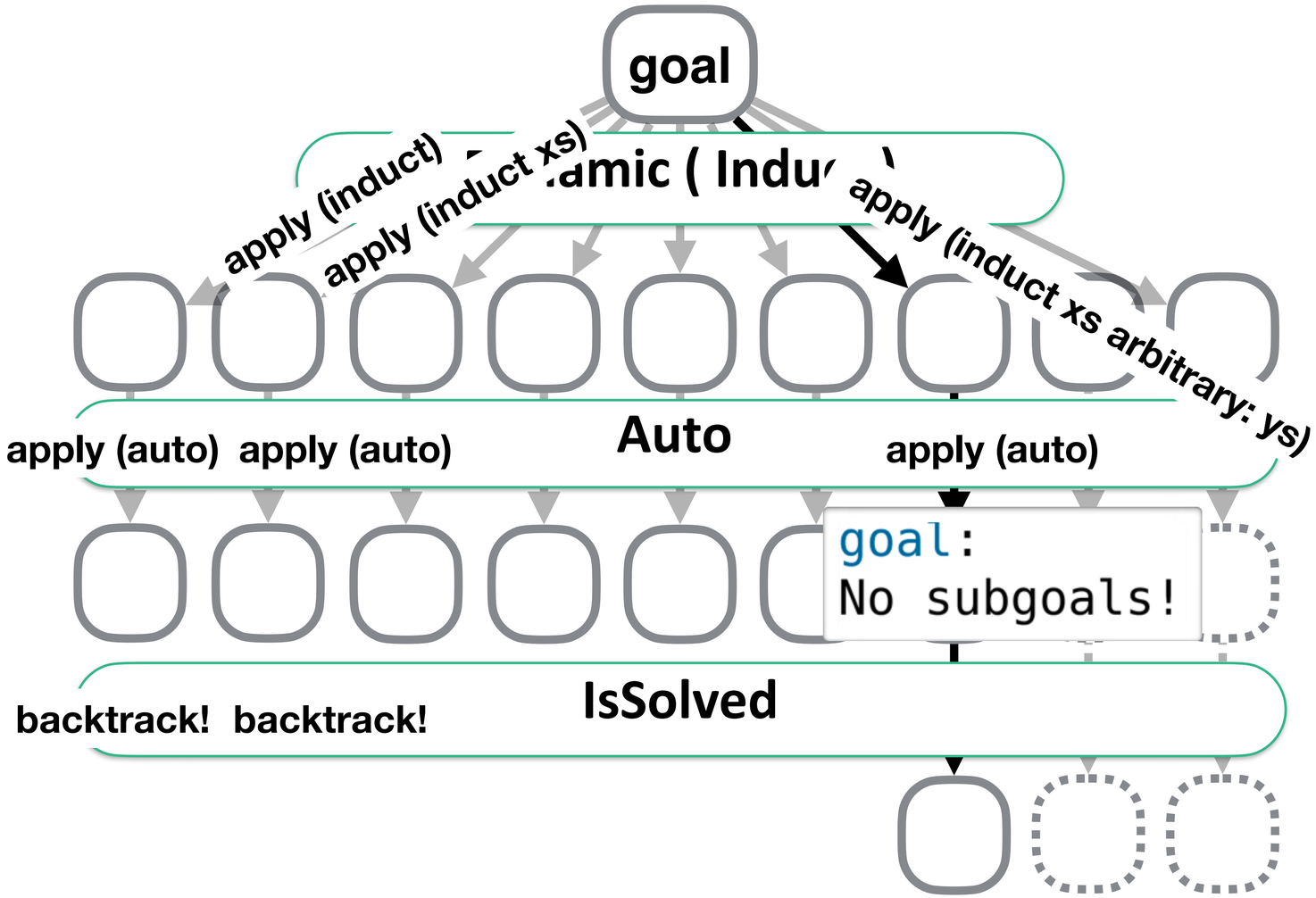
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



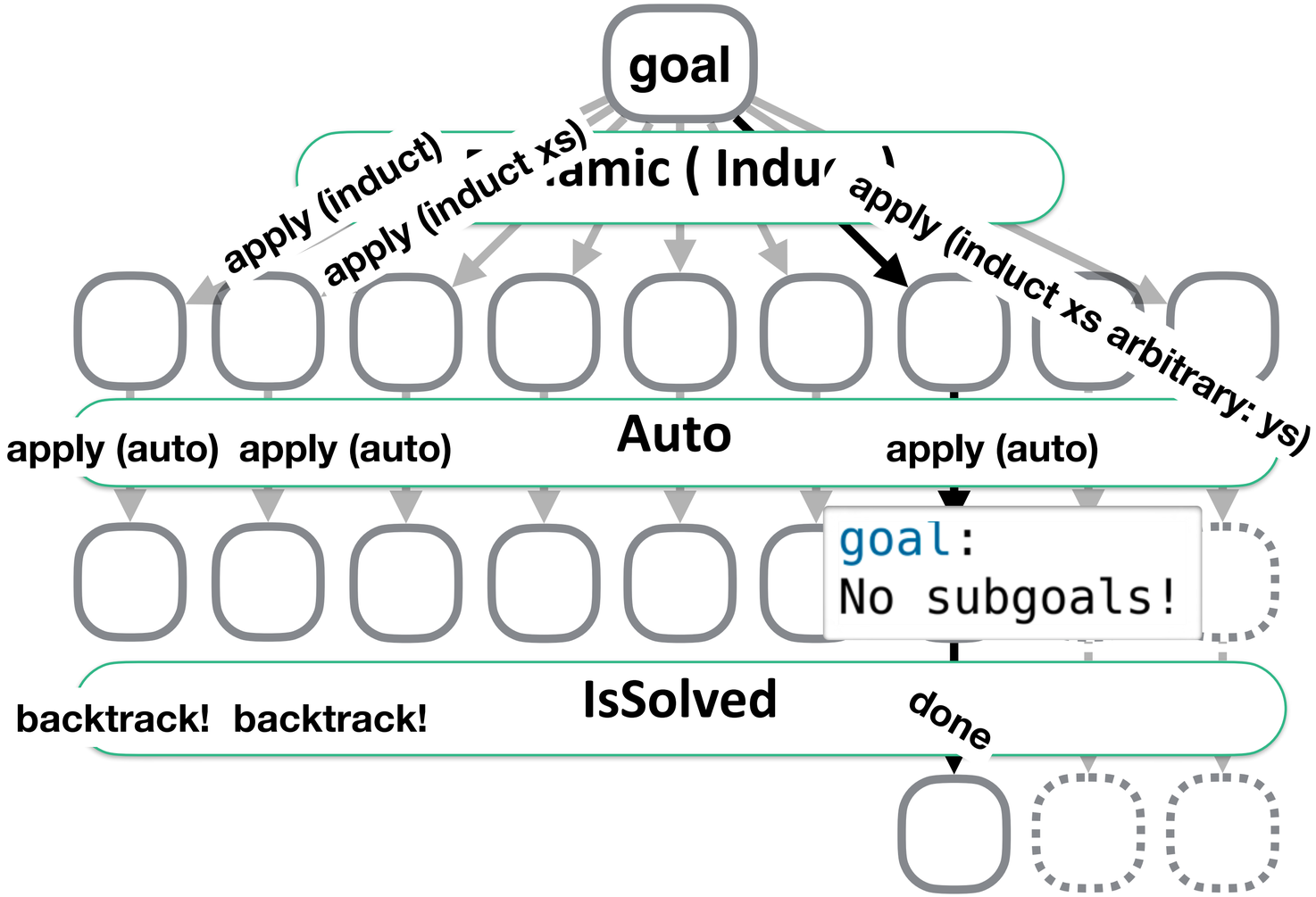
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



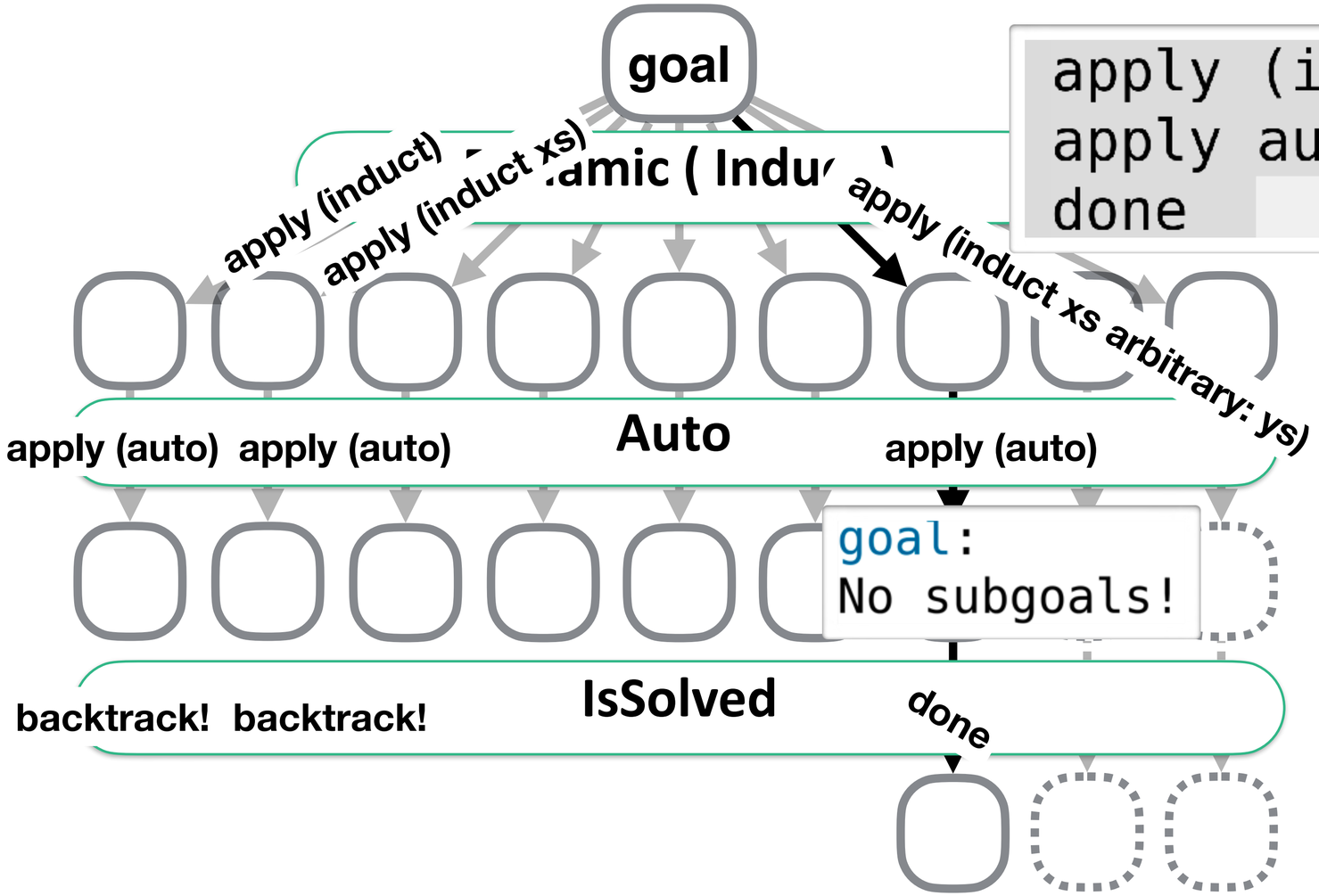
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```




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

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

```
fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

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

theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd
  oops
```

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

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

Output | Query | Sledgehammer | Symbols

16,18 (381/393) (isabelle,isabelle,UTF-8-Isabelle) U.. 195/512MB 1 error(s) 2:14 PM

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

(*PSL*)

```
15 theorem "rev2 xs ys = rev1 xs @ ys"
16   find_proof DInd
17   oops
18
```

Proof state | Auto update | Update | Search:

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

Output | Query | Sledgehammer | Symbols

16,18 (381/393) (isabelle,isabelle,UTF-8-Isabelle) nmr o U.. 195/512MB 1 error(s) 2:14 PM

Sidekick | State | Theories

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

(*PSL*)

```
15 theorem "rev2 xs ys = rev1 xs @ ys"
16   find_proof DInd
17   oops
18
```

Proof state | Auto update | Update | Search:

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

Output | Query | Sledgehammer | Symbols

16,18 (381/393) (isabelle,isabelle,UTF-8-Isabelle) U.. 195/512MB 1 error(s) 2:14 PM

Sidekick | State | Theories

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

(*PSL*)

```
theorem "rev2 xs ys = rev1 xs @ ys"
?find_proof DInd
oops
```

Indeed, auto returned quickly in our case.

Proof state | Auto update | Update | Search:

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

Output | Query | Sledgehammer | Symbols

16,18 (381/393) (isabelle,isabelle,UTF-8-Isabelle) U.. 195/512MB 1 error(s) 2:14 PM

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

(*PSL*)

```
theorem "rev2 xs ys = rev1 xs @ ys"
?find_proof DInd
oops
```

Indeed, auto returned quickly in our case.

What if subsequent steps are more involved?

```
apply (
apply auto
done
```

Sidekick State Theories

Search:

Output Query Sledgehammer Symbols

16,18 (381/393) (isabelle,isabelle,UTF-8-Isabelle) nmr o U.. 195/512MB 1 error(s) 2:14 PM

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

Algorithmic approaches work well, but not always → heuristics.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

(*PSL*)

```
theorem "rev2 xs ys = rev1 xs @ ys"
  ?find_proof DInd
  oops
```

Indeed, auto returned quickly in our case.

What if subsequent steps are more involved?

```
apply (
apply auto
done
```

We have to identify promising arguments for induction tactics

without relying on a search!

Sidekick State Theories

Search:

Output Query Sledgehammer Symbols

16,18 (381/393) (isabelle,isabelle,UTF-8-lsabelle) 195/512MB 1 error(s) 2:14 PM

smart_induct (FMCAD2020)

Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

goal

smart_induct (FMCAD2020)

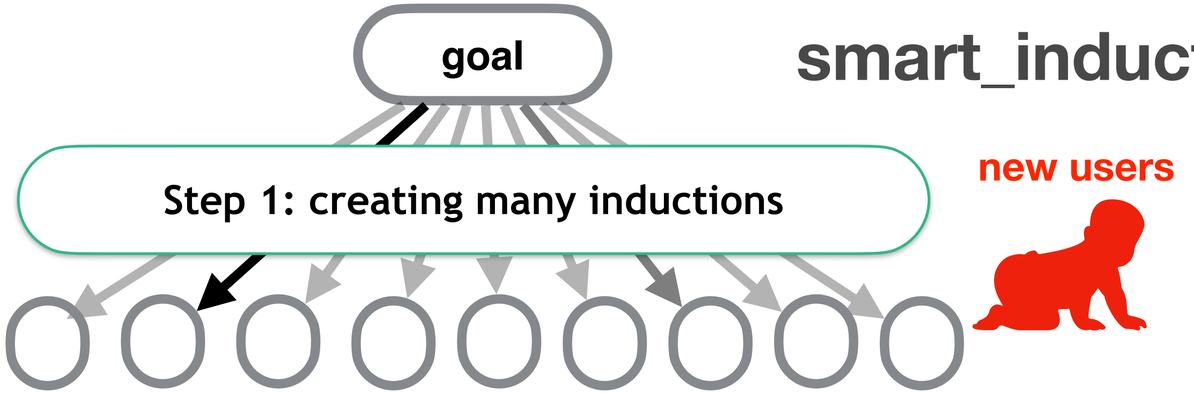
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

smart_induct (FMCAD2020)



new users

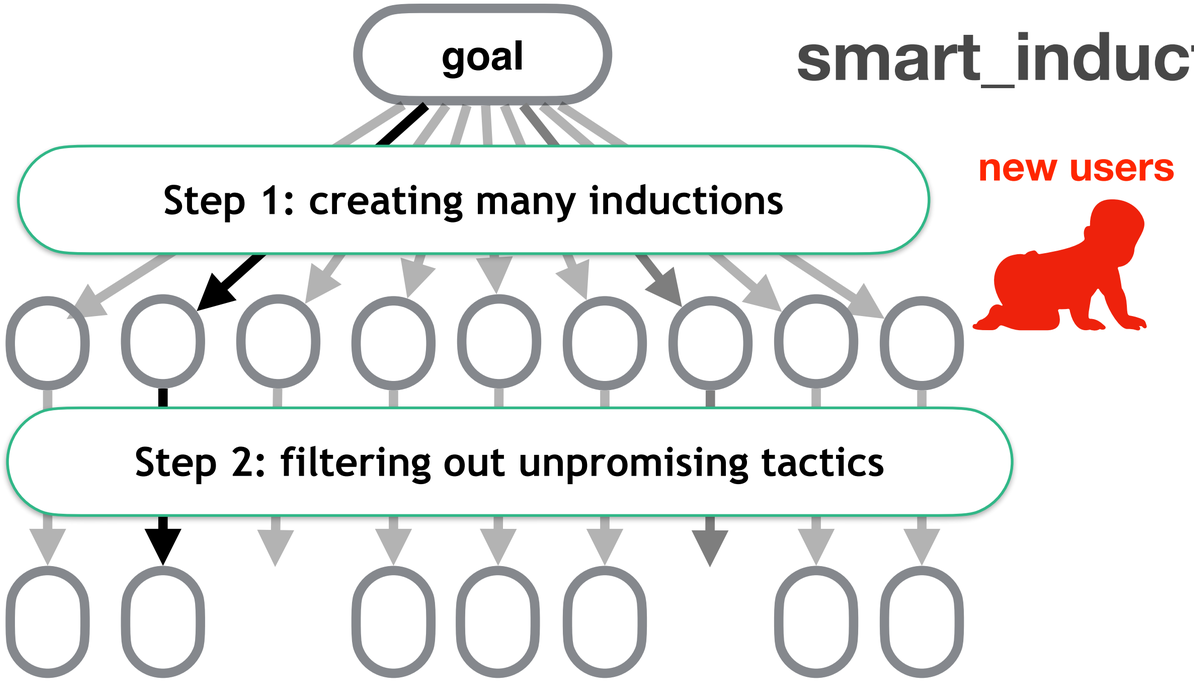
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

smart_induct (FMCAD2020)



new users

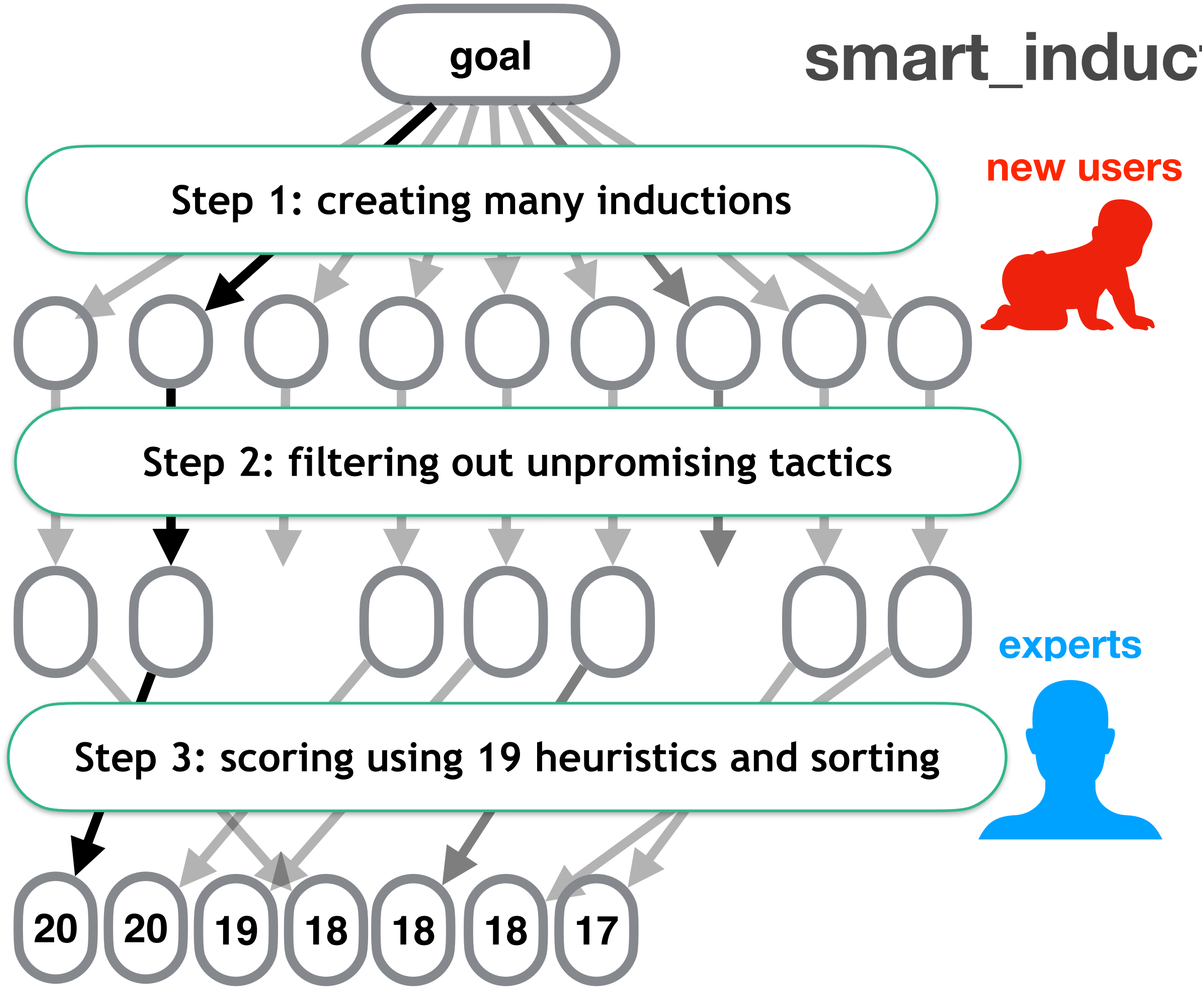
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

smart_induct (FMCAD2020)



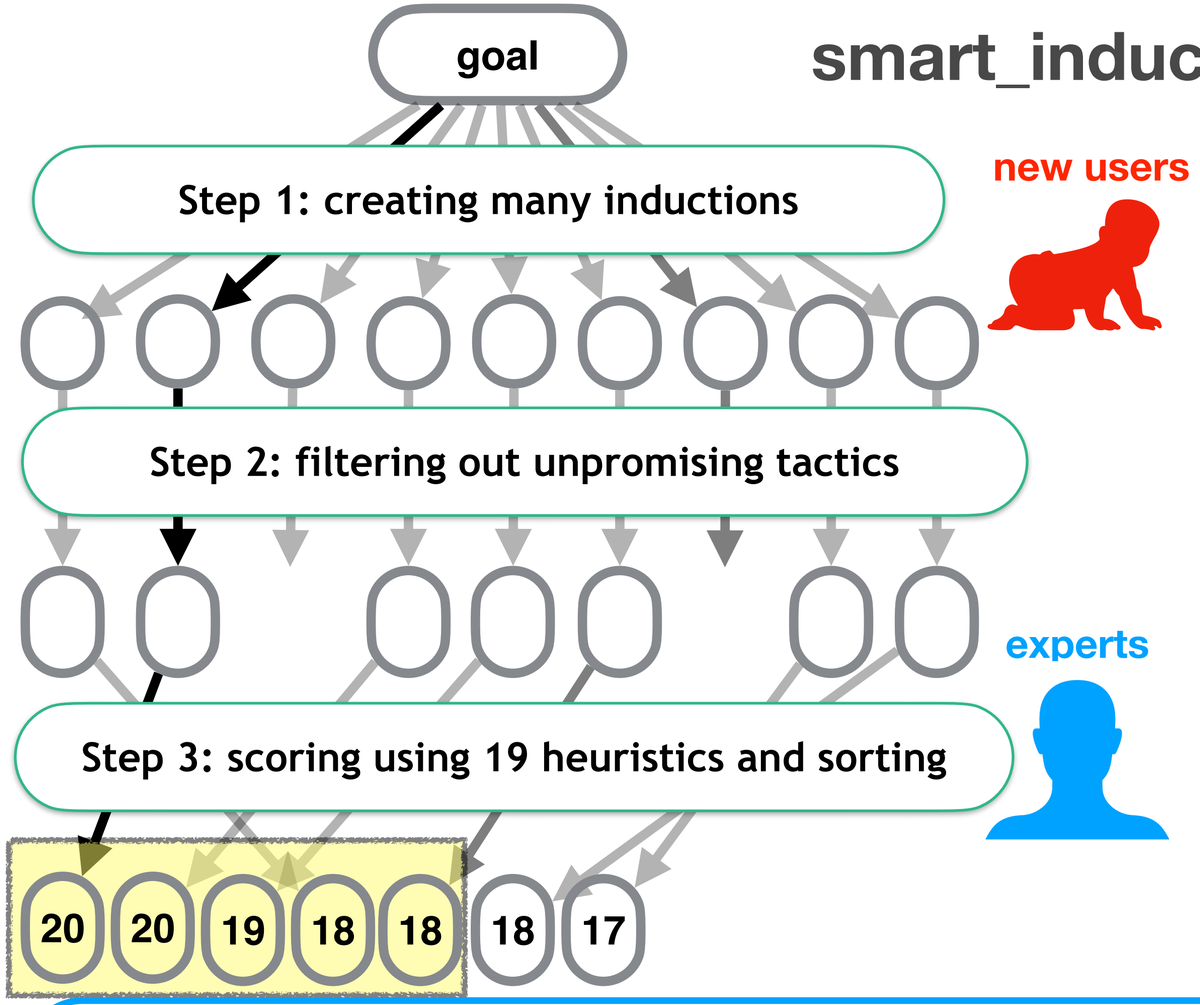
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

smart_induct (FMCAD2020)



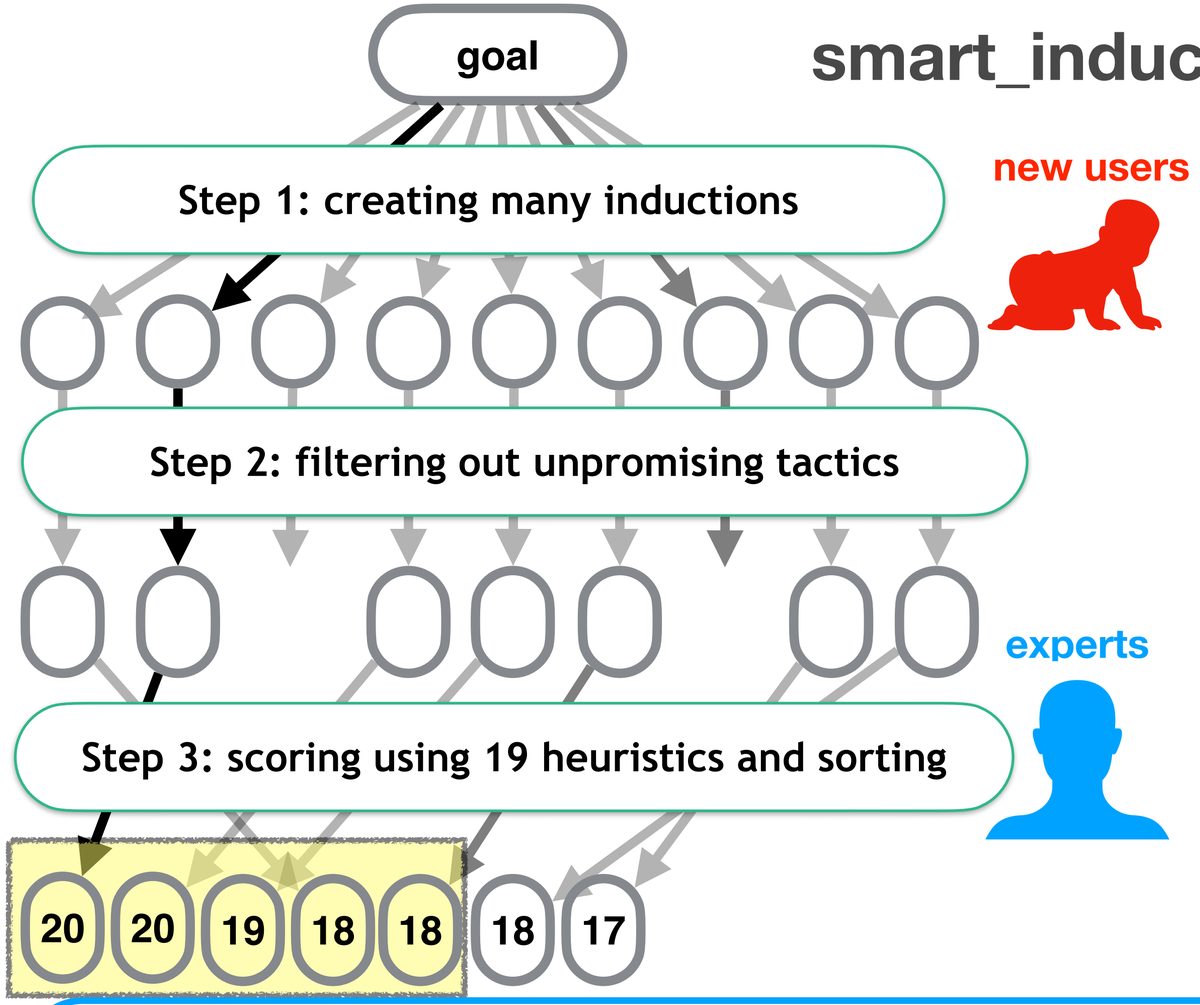
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

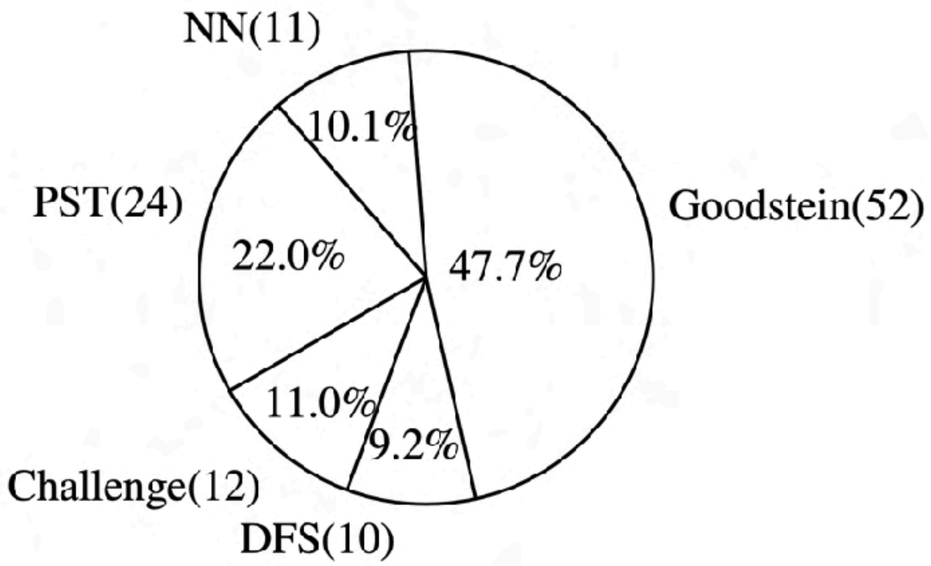
Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

smart_induct (FMCAD2020)



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



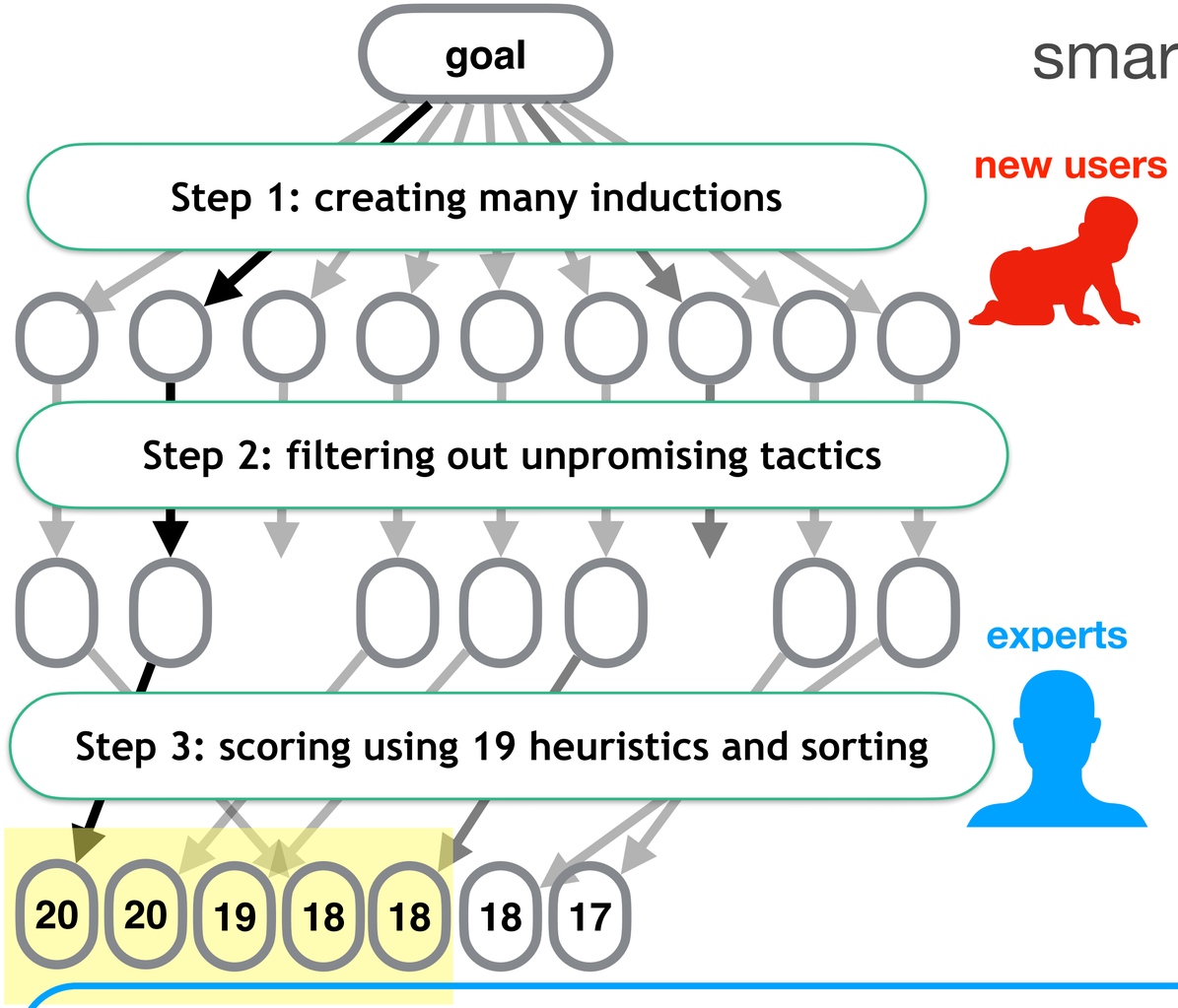
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

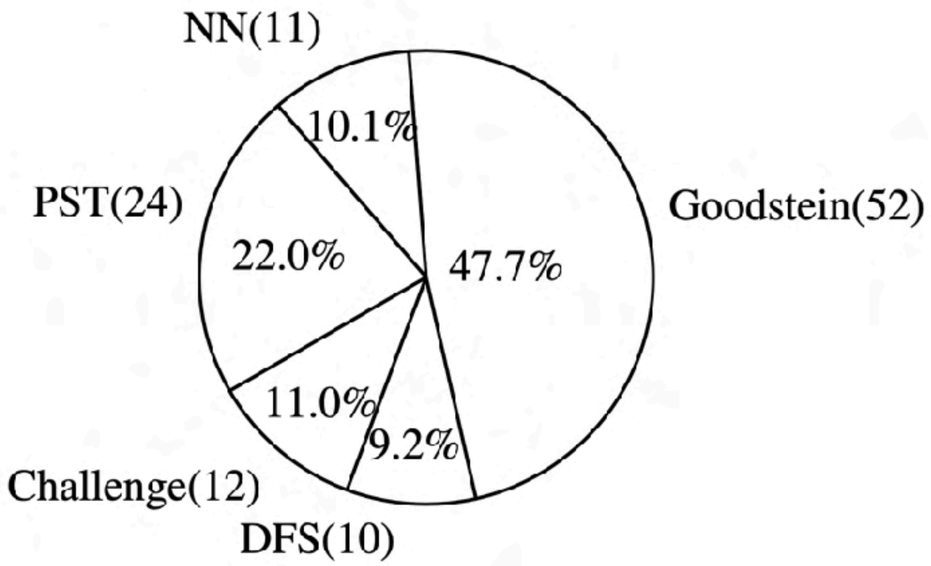
Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

smart_induct



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



Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

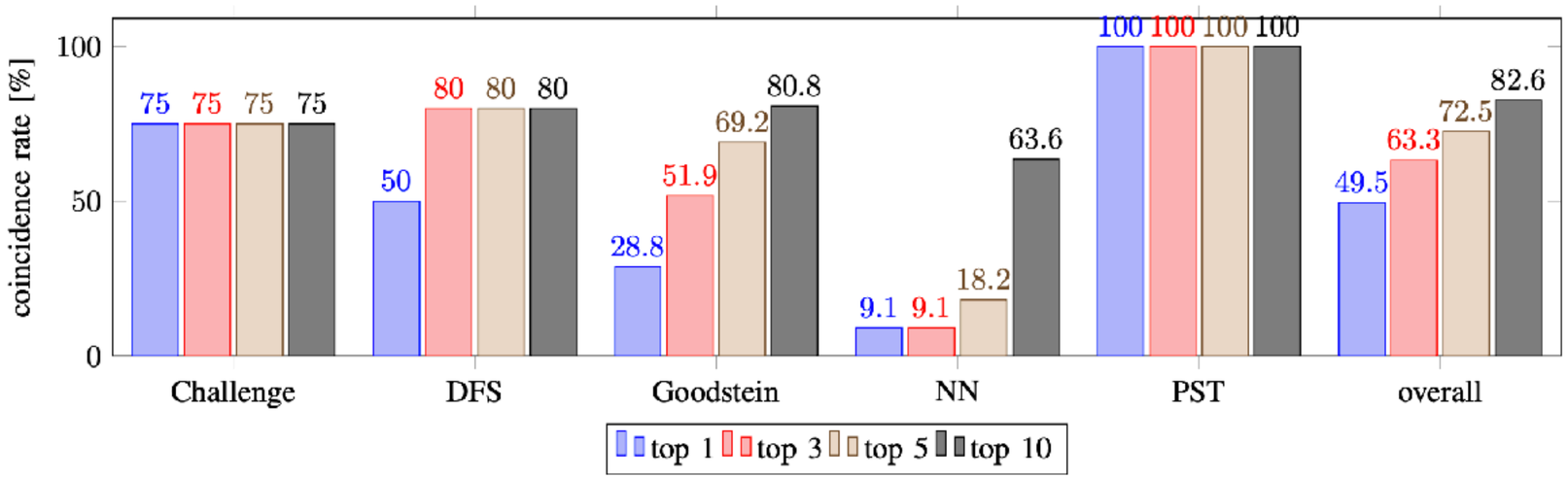
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

Coincidence rates



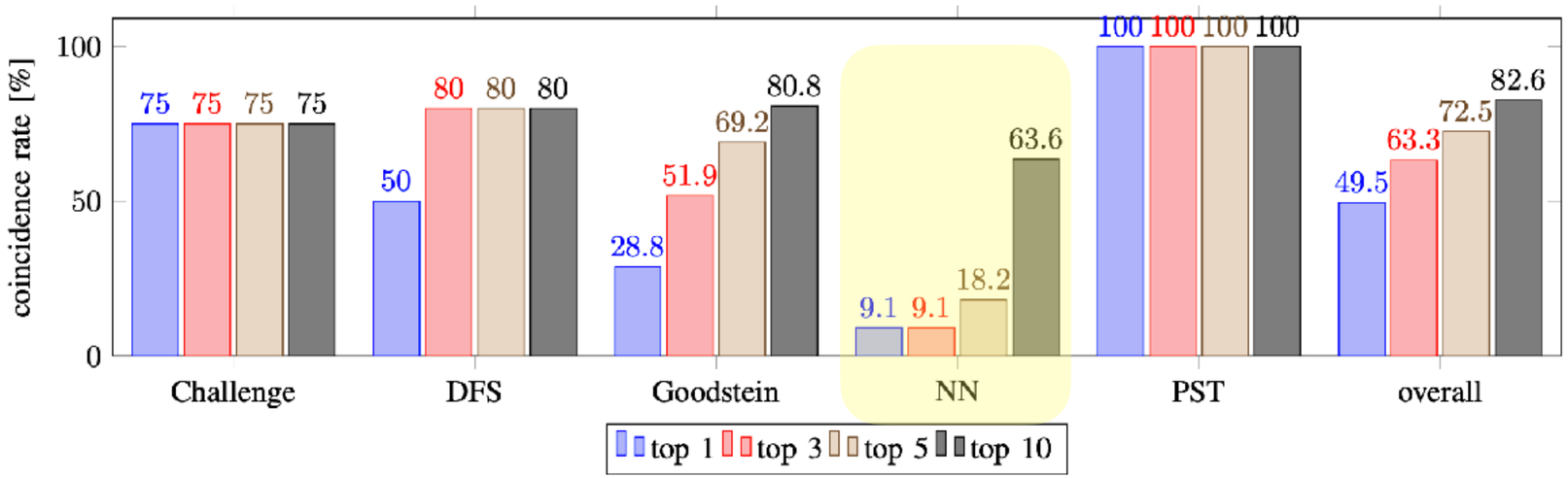
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

Coincidence rates



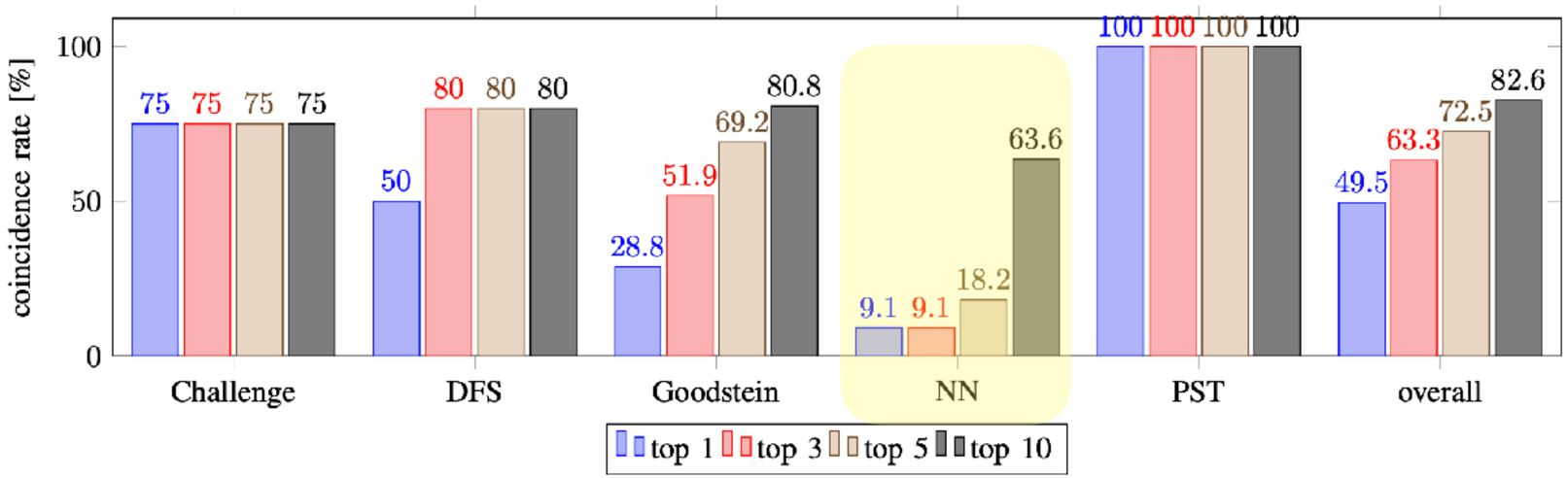
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

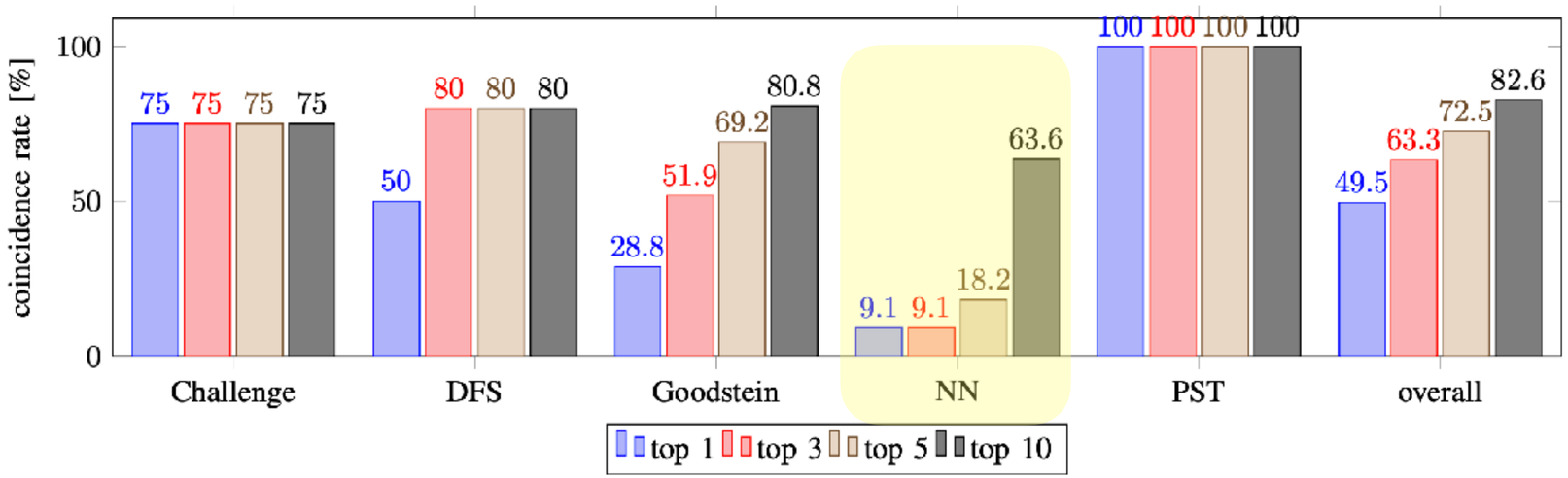
Abstract reasoning is often necessary.

It is okay to make mistakes if we can identify them quickly.

Coincidence rates

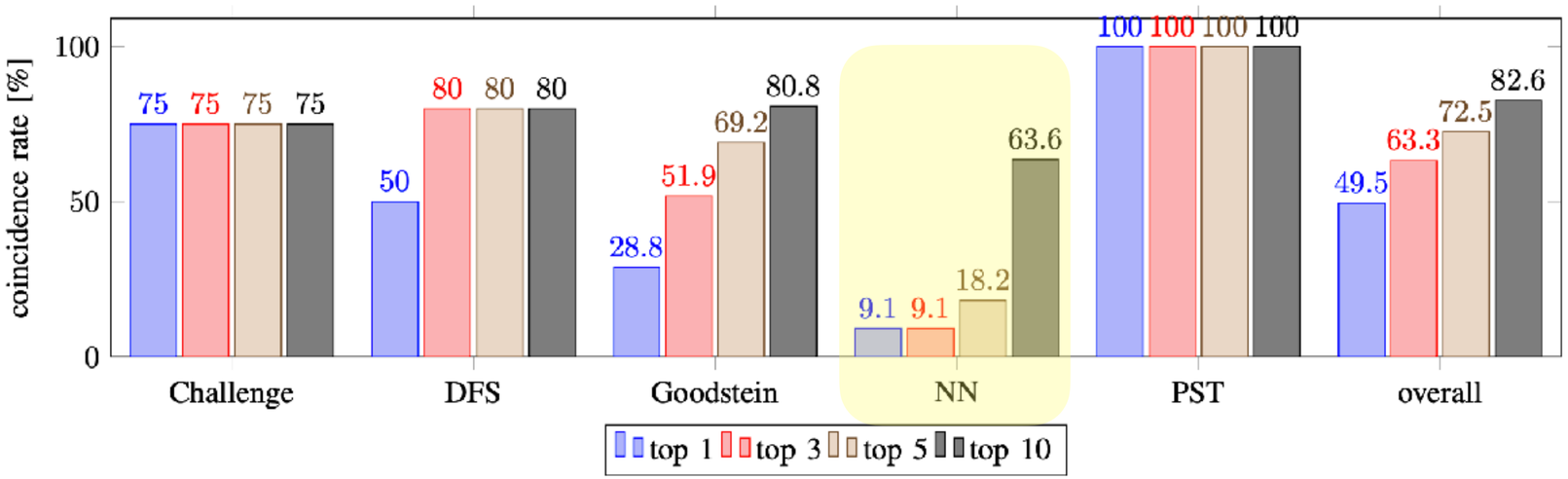


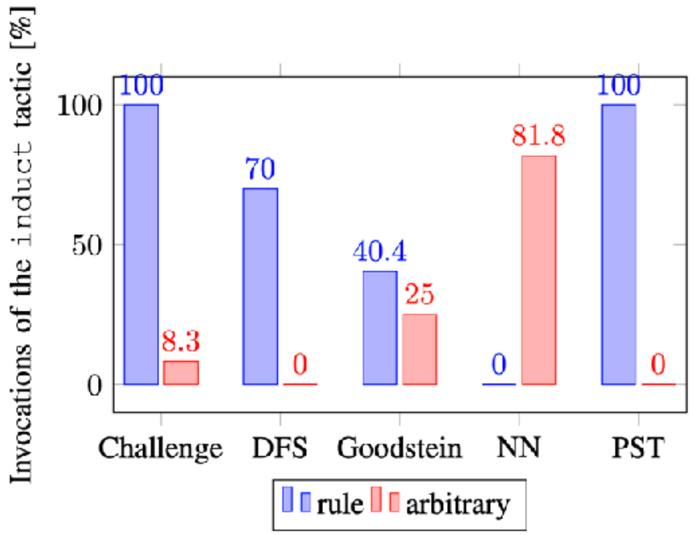
Coincidence rates



Is smart_induct Domain dependent?

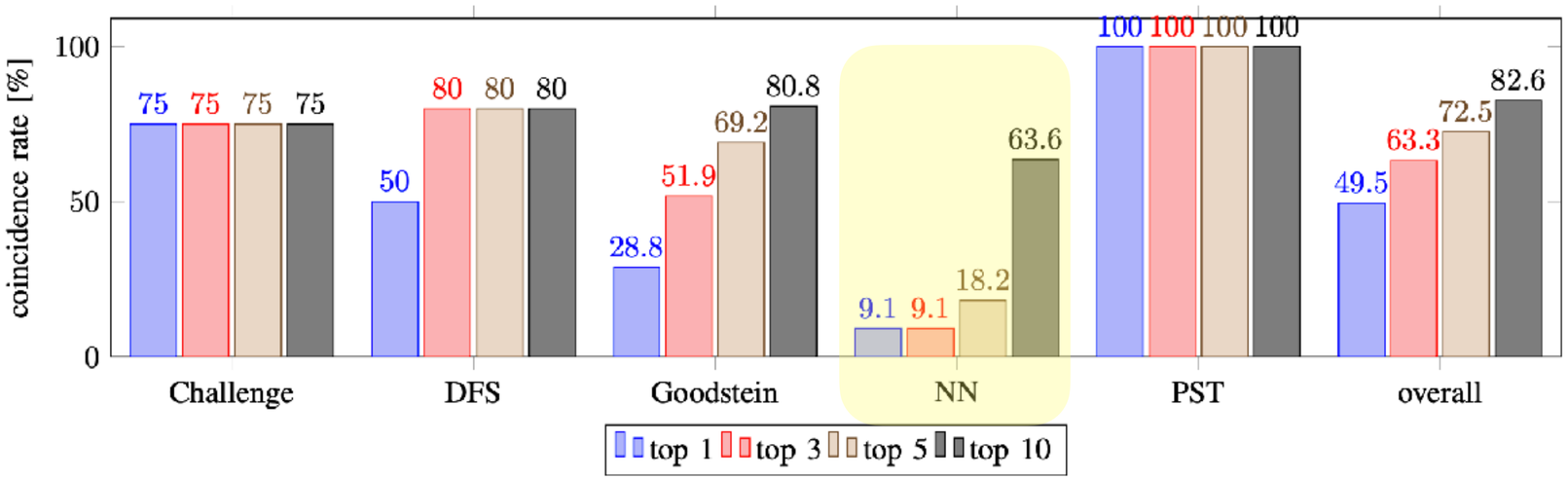
Coincidence rates

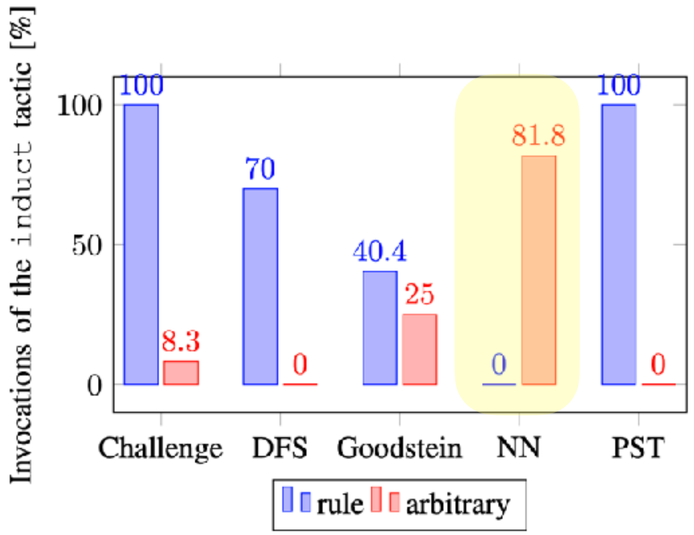




Is smart_induct Domain dependent?

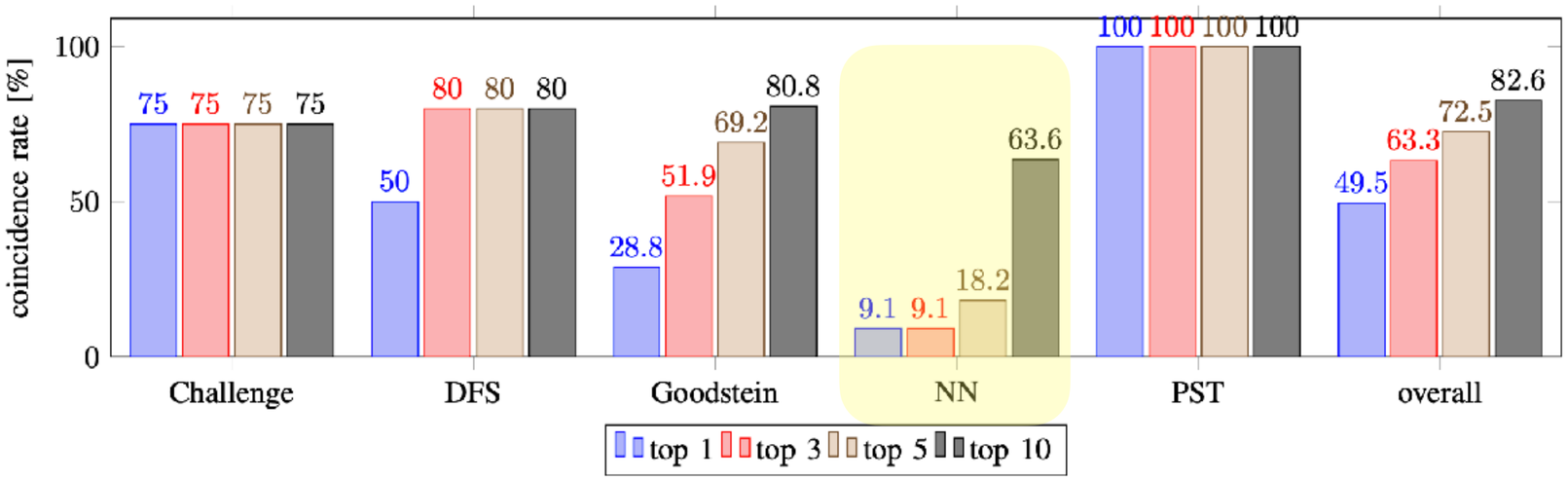
Coincidence rates

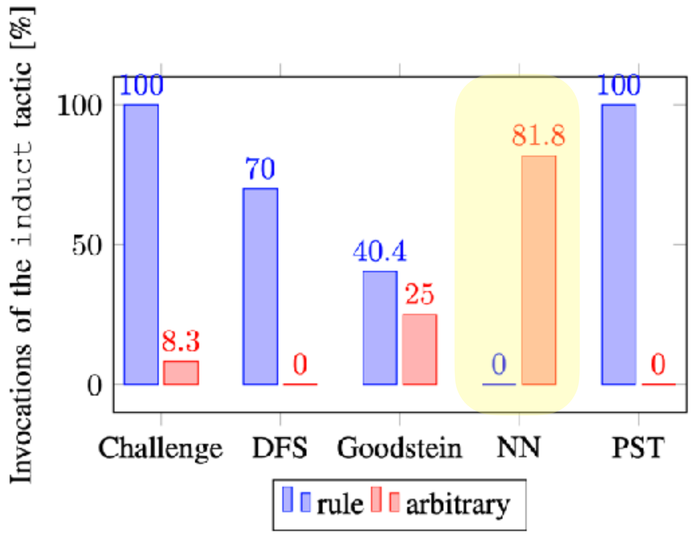


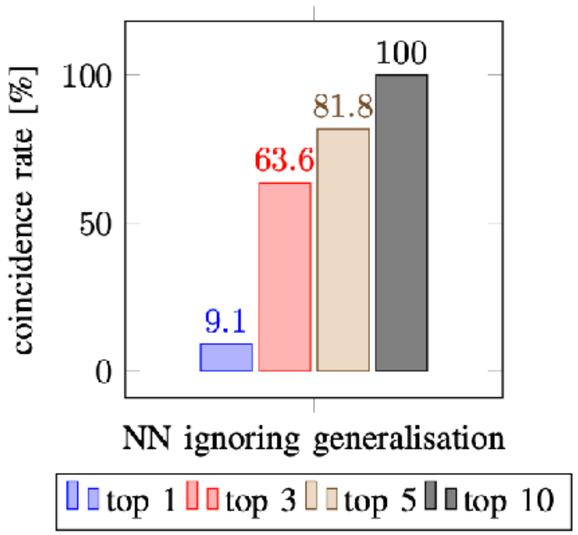


Is smart_induct Domain dependent?

Coincidence rates

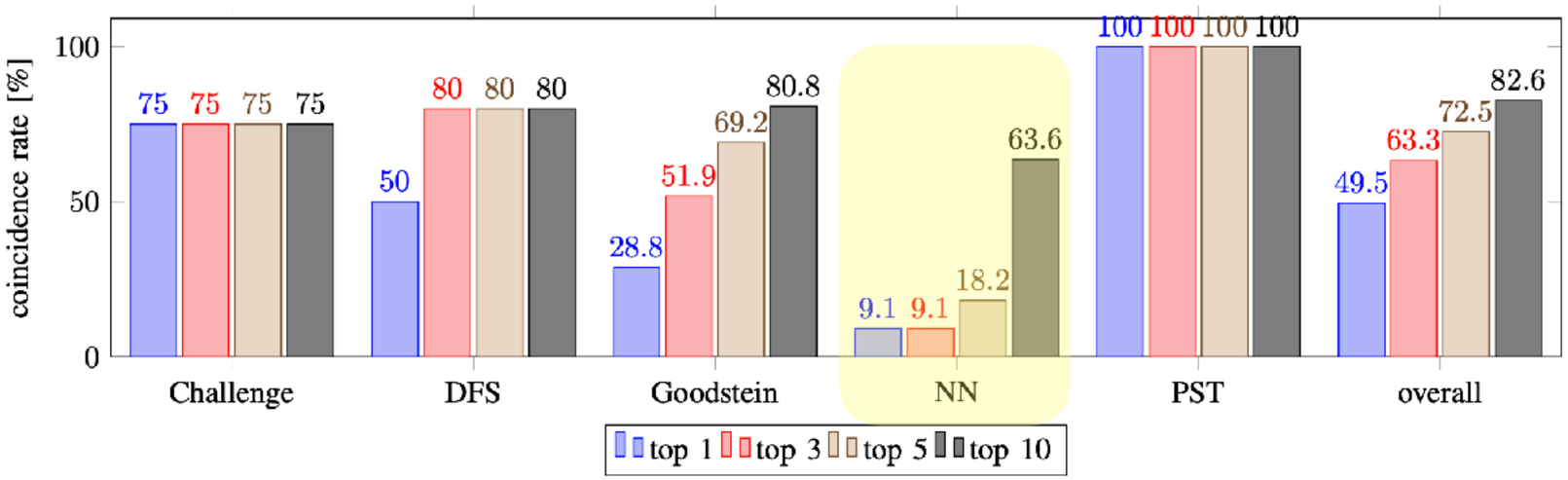


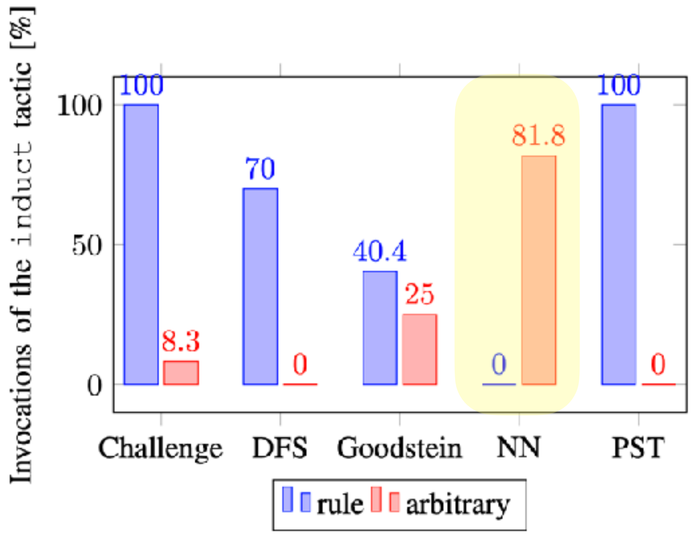


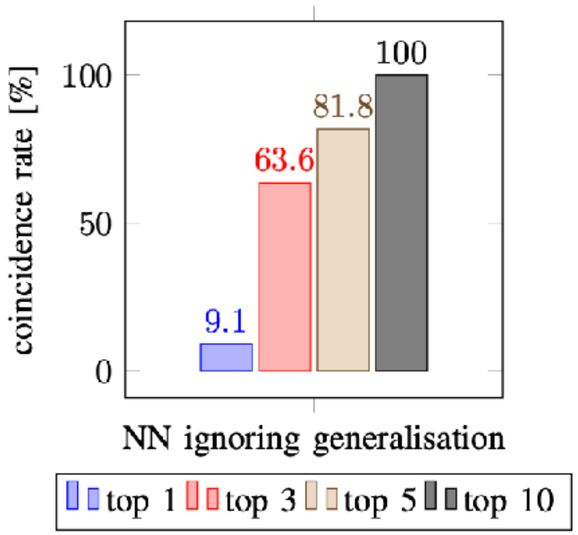


Is smart_induct Domain dependent?

Coincidence rates







Is smart_induct Domain dependent?

smart_induct is bad at generalisation heuristics.

Program 4 Automatic inductive prover without SeLFiE

```
strategy Auto_Solve = Thens [Auto, IsSolved]
strategy PSL_WO_SeLFiE =
  Ors [
    Auto_Solve,
    PThenOne [Dynamic (Induct), Auto_Solve]
    PThenOne [
      Dynamic (Induct),
      Thens [Auto, RepeatN(Hammer), IsSolved]
    ]
  ]
```

Program 5 Automatic inductive prover with SeLFiE

```
strategy PSL_W_SeLFiE =
  Ors [
    Auto_Solve,
    PThenOne [Semantic_Induct, Auto_Solve]
    PThenOne [
      Semantic_Induct,
      Thens [Auto, RepeatN(Hammer), IsSolved]
    ]
  ]
```

Program 4 Automatic inductive prover without SeLFiE

```
strategy Auto_Solve = Thens [Auto, IsSolved]
strategy PSL_WO_SeLFiE =
  Ors [
    Auto_Solve,
    PThenOne [Dynamic (Induct), Auto_Solve]
    PThenOne [
      Dynamic (Induct),
      Thens [Auto, RepeatN(Hammer), IsSolved]
    ]
  ]
```

Program 5 Automatic inductive prover with SeLFiE

```
strategy PSL_W_SeLFiE =
  Ors [
    Auto_Solve,
    PThenOne [Semantic_Induct, Auto_Solve]
    PThenOne [
      Semantic_Induct,
      Thens [Auto, RepeatN(Hammer), IsSolved]
    ]
  ]
```

Success rate

timeouts	Program 5	Program 4
0.3[s]	11.0%	1.2%
1.0[s]	25.6%	1.7%
3.0[s]	28.2%	21.9%
10.0[s]	34.9%	28.0%
30.0[s]	45.8%	38.3%

Bibliography

Step 1: Tactic recommendation.
https://doi.org/10.1145/3238147.3238210

Step 2: Tactic argument recommendation.

a: Guided backtracking search with tracing (PSL).
https://doi.org/10.1007/978-3-319-63046-5_32

b: Logical Feature Extraction (LiFtEr).
https://doi.org/10.1007/978-3-030-34175-6_14

c: smart_induct using LiFtEr.
https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_32

d: Semantic-aware Logical Feature Extraction (SeLFiE).
https://arxiv.org/pdf/2010.10296.pdf

e: sem_ind using SeLFiE.
https://arxiv.org/pdf/2009.09215.pdf

f: conjecturing using abductive-reasoning.
https://doi.org/10.1007/978-3-319-96812-4_19

Bibliography

Step 1: Tactic recommendation.
https://doi.org/10.1145/3238147.3238210

Step 2: Tactic argument recommendation.

a: Guided backtracking search with tracing (PSL).
https://doi.org/10.1007/978-3-319-63046-5_32

b: Logical Feature Extraction (LiFtEr).
https://doi.org/10.1007/978-3-030-34175-6_14

c: smart_induct using LiFtEr.
https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_32

d: Semantic-aware Logical Feature Extraction (SeLFiE).
https://arxiv.org/pdf/2010.10296.pdf

e: sem_ind using SeLFiE.
https://arxiv.org/pdf/2009.09215.pdf

f: conjecturing using abductive-reasoning.
https://doi.org/10.1007/978-3-319-96812-4_19

fin!

smart_induct:

smart_induct:

1: does not rely on a proof search (unlike PSL). 😄

smart_induct:

1: does not rely on a proof search (unlike PSL). 😄

2: is domain-agnostic because of LiFtEr. 😄

smart_induct:

1: does not rely on a proof search (unlike PSL). 😄

2: is domain-agnostic because of LiFtEr. 😄

3: can predict induction terms. 😄

smart_induct:

1: does not rely on a proof search (unlike PSL). 😄

2: is domain-agnostic because of LiFtEr. 😄

3: can predict induction terms. 😄

4: cannot predict variable generalisation. 😱

backup slides

It is difficult to write induction heuristics.

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

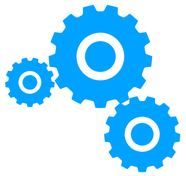
It is difficult to write induction heuristics.

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

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

experts
(= me)

20 heuristics



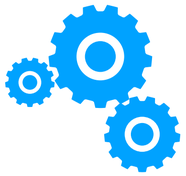
It is difficult to write induction heuristics.

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

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

experts
(= me)

20 heuristics



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

It is difficult to write induction heuristics.

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

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

experts
(= me)

new users

proof goal

candidate induction

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

20 heuristics

[✓ ,
✗ ,
✓ ,...]

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

It is difficult to write induction heuristics.

① blue = what I can see before releasing smart_induct

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

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

experts
(= me)

new users

proof goal

candidate induction

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

20 heuristics

[✓ ,
✗ ,
✓ ,...]

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

It is difficult to write induction heuristics.

① blue = what I can see before releasing smart_induct

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

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

experts
(= me)

new users

proof goal

candidate induction

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

20 heuristics

[✓ ,
✗ ,
✓ ,...]

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

② red = what will be developed after releasing smart_induct

It is difficult to write induction heuristics.

① blue = what I can see before releasing smart_induct

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

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

experts
(= me)

new users

proof goal

candidate induction

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

20 heuristics

[✓ ,
✗ ,
✓ ,...]

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

② red = what will be developed after releasing smart_induct

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

It is difficult to write induction heuristics.

① blue = what I can see before releasing smart_induct

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

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

experts
(= me)

I can't refer to terms w/o knowing proof goals.

new users

proof goal candidate induction

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

20 heuristics

[✓ ,
✗ ,
✓ ,...]

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

② red = what will be developed after releasing smart_induct

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

ML for Inductive Theorem Proving
the BAD

```
lemma "rev2 xs ys = rev1 xs @ ys"
```

<- one abstract representation

```
lemma "rev2 [1,2]          [] = rev1 [1,2]          @ []" by auto
lemma "rev2 [1,2,3]        [] = rev1 [1,2,3]        @ []" by auto
lemma "rev2 [''a'',''b''] [] = rev1 [''a'',''b''] @ []" by auto
lemma "rev2 [x,y,z]        [] = rev1 [x,y,z]        @ []" by auto
```

<- many concrete cases

ML for Inductive Theorem Proving
the BAD

```
lemma "rev2 xs ys = rev1 xs @ ys"
```

<- one abstract representation

ML

```
lemma "rev2 [1,2]          [] = rev1 [1,2]          @ []" by auto
lemma "rev2 [1,2,3]        [] = rev1 [1,2,3]        @ []" by auto
lemma "rev2 [''a'',''b''] [] = rev1 [''a'',''b''] @ []" by auto
lemma "rev2 [x,y,z]        [] = rev1 [x,y,z]        @ []" by auto
```

<- many concrete cases

ML for Inductive Theorem Proving
the BAD

```
lemma "rev2 xs ys = rev1 xs @ ys"
```

by auto <- one abstract representation

ML

```
lemma "rev2 [1,2]          [] = rev1 [1,2]          @ []" by auto
lemma "rev2 [1,2,3]        [] = rev1 [1,2,3]        @ []" by auto
lemma "rev2 [''a'',''b''] [] = rev1 [''a'',''b''] @ []" by auto
lemma "rev2 [x,y,z]        [] = rev1 [x,y,z]        @ []" by auto
```

<- many concrete cases

ML for Inductive Theorem Proving the BAD

```
lemma "rev2 xs ys = rev1 xs @ ys"
```

```
by auto
```

<- one abstract representation

```
Failed to apply initial proof method⌂:
goal (1 subgoal):
 1. rev2 xs ys = rev1 xs @ ys
```

ML

```
lemma "rev2 [1,2]          [] = rev1 [1,2]          @ []" by auto
lemma "rev2 [1,2,3]        [] = rev1 [1,2,3]        @ []" by auto
lemma "rev2 [''a'',''b''] [] = rev1 [''a'',''b''] @ []" by auto
lemma "rev2 [x,y,z]        [] = rev1 [x,y,z]        @ []" by auto
```

<- many concrete cases

ML for Inductive Theorem Proving the BAD

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

~~by auto~~ <- one abstract representation

```
Failed to apply initial proof method⌂:
goal (1 subgoal):
 1. rev2 xs ys = rev1 xs @ ys
```

```
lemma "rev2 [1,2]          [] = rev1 [1,2]          @ []" by auto
lemma "rev2 [1,2,3]        [] = rev1 [1,2,3]        @ []" by auto
lemma "rev2 [''a'',''b''] [] = rev1 [''a'',''b''] @ []" by auto
lemma "rev2 [x,y,z]        [] = rev1 [x,y,z]        @ []" by auto
```

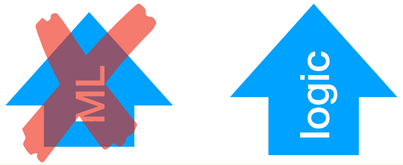
<- many concrete cases

ML for Inductive Theorem Proving the BAD

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

~~by auto~~ <- one abstract representation

```
Failed to apply initial proof method⌂:
goal (1 subgoal):
 1. rev2 xs ys = rev1 xs @ ys
```



<- abstraction using expressive logic

```
lemma "rev2 [1,2]          [] = rev1 [1,2]          @ []" by auto
lemma "rev2 [1,2,3]        [] = rev1 [1,2,3]        @ []" by auto
lemma "rev2 [''a'',''b''] [] = rev1 [''a'',''b''] @ []" by auto
lemma "rev2 [x,y,z]        [] = rev1 [x,y,z]        @ []" by auto
```

<- many concrete cases

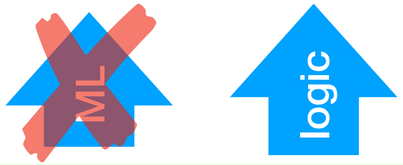
ML for Inductive Theorem Proving the BAD

polymorphism

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

~~by auto~~ <- one abstract representation

```
Failed to apply initial proof method⌂:
goal (1 subgoal):
 1. rev2 xs ys = rev1 xs @ ys
```



<- abstraction using expressive logic

```
lemma "rev2 [1,2]          [] = rev1 [1,2]          @ []" by auto
lemma "rev2 [1,2,3]        [] = rev1 [1,2,3]        @ []" by auto
lemma "rev2 [''a'',''b''] [] = rev1 [''a'',''b''] @ []" by auto
lemma "rev2 [x,y,z]        [] = rev1 [x,y,z]        @ []" by auto
```

<- many concrete cases

ML for Inductive Theorem Proving the BAD

polymorphism **type class**

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

~~by auto~~ <- one abstract representation

```
Failed to apply initial proof method⌂:
goal (1 subgoal):
 1. rev2 xs ys = rev1 xs @ ys
```

~~ML~~ logic <- abstraction using expressive logic

```
lemma "rev2 [1,2]          [] = rev1 [1,2]          @ []" by auto
lemma "rev2 [1,2,3]        [] = rev1 [1,2,3]        @ []" by auto
lemma "rev2 [''a'',''b''] [] = rev1 [''a'',''b''] @ []" by auto
lemma "rev2 [x,y,z]        [] = rev1 [x,y,z]        @ []" by auto
```

<- many concrete cases

ML for Inductive Theorem Proving the BAD

polymorphism

type class

universal quantifier

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

~~by auto~~ <- one abstract representation

```
Failed to apply initial proof method⌂:
goal (1 subgoal):
 1. rev2 xs ys = rev1 xs @ ys
```

~~ML~~ logic <- abstraction using expressive logic

```
lemma "rev2 [1,2]          [] = rev1 [1,2]          @ []" by auto
lemma "rev2 [1,2,3]        [] = rev1 [1,2,3]        @ []" by auto
lemma "rev2 [''a'',''b''] [] = rev1 [''a'',''b''] @ []" by auto
lemma "rev2 [x,y,z]        [] = rev1 [x,y,z]        @ []" by auto
```

<- many concrete cases

ML for Inductive Theorem Proving the BAD

Higher-Order functions

polymorphism **type class**

universal quantifier

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

~~by auto~~ <- one abstract representation

```
Failed to apply initial proof method⌂:
goal (1 subgoal):
 1. rev2 xs ys = rev1 xs @ ys
```

~~ML~~ logic <- abstraction using expressive logic

```
lemma "rev2 [1,2]          [] = rev1 [1,2]          @ []" by auto
lemma "rev2 [1,2,3]        [] = rev1 [1,2,3]        @ []" by auto
lemma "rev2 [''a'',''b''] [] = rev1 [''a'',''b''] @ []" by auto
lemma "rev2 [x,y,z]        [] = rev1 [x,y,z]        @ []" by auto
```

<- many concrete cases

ML for Inductive Theorem Proving the BAD

Higher-Order functions

polymorphism

type class

universal quantifier

lambda abstraction

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

~~by auto~~ <- one abstract representation

```
Failed to apply initial proof method⌂:
goal (1 subgoal):
 1. rev2 xs ys = rev1 xs @ ys
```

~~ML~~ logic <- abstraction using expressive logic

```
lemma "rev2 [1,2]          [] = rev1 [1,2]          @ []" by auto
lemma "rev2 [1,2,3]        [] = rev1 [1,2,3]        @ []" by auto
lemma "rev2 [''a'',''b''] [] = rev1 [''a'',''b''] @ []" by auto
lemma "rev2 [x,y,z]        [] = rev1 [x,y,z]        @ []" by auto
```

<- many concrete cases

ML for Inductive Theorem Proving the BAD

Higher-Order functions

polymorphism

type class

universal quantifier

lambda abstraction

concise formula that can cover many concrete cases

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

~~by auto~~ <- one abstract representation

```
Failed to apply initial proof method⌂:
goal (1 subgoal):
 1. rev2 xs ys = rev1 xs @ ys
```

~~ML~~

logic

<- abstraction using expressive logic

```
lemma "rev2 [1,2]          [] = rev1 [1,2]          @ []" by auto
lemma "rev2 [1,2,3]        [] = rev1 [1,2,3]        @ []" by auto
lemma "rev2 [''a'',''b''] [] = rev1 [''a'',''b''] @ []" by auto
lemma "rev2 [x,y,z]        [] = rev1 [x,y,z]        @ []" by auto
```

<- many concrete cases

ML for Inductive Theorem Proving the BAD

Higher-Order functions

polymorphism

type class

universal quantifier

lambda abstraction

concise formula that can cover many concrete cases

different proof for general case

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

~~by auto~~ <- one abstract representation

```
Failed to apply initial proof method⌂:
goal (1 subgoal):
 1. rev2 xs ys = rev1 xs @ ys
```

~~ML~~

logic

<- abstraction using expressive logic

```
lemma "rev2 [1,2]          [] = rev1 [1,2]          @ []" by auto
lemma "rev2 [1,2,3]        [] = rev1 [1,2,3]        @ []" by auto
lemma "rev2 [''a'',''b''] [] = rev1 [''a'',''b''] @ []" by auto
lemma "rev2 [x,y,z]        [] = rev1 [x,y,z]        @ []" by auto
```

<- many concrete cases

ML for Inductive Theorem Proving the BAD

Higher-Order functions

polymorphism

type class

universal quantifier

lambda abstraction

concise formula that can cover many concrete cases

different proof for general case

A small data set is not a failure but an achievement!

```
lemma "rev2 xs ys = rev1 xs @ ys"
 by(induct xs ys rule: rev2.induct) auto
```

~~by auto~~ <- one abstract representation

```
Failed to apply initial proof method⌂:
goal (1 subgoal):
 1. rev2 xs ys = rev1 xs @ ys
```

~~ML~~ logic <- abstraction using expressive logic

```
lemma "rev2 [1,2]          [] = rev1 [1,2]          @ []" by auto
lemma "rev2 [1,2,3]        [] = rev1 [1,2,3]        @ []" by auto
lemma "rev2 [''a'',''b''] [] = rev1 [''a'',''b''] @ []" by auto
lemma "rev2 [x,y,z]        [] = rev1 [x,y,z]        @ []" by auto
```

<- many concrete cases

Example Assertion in LiFtEr (in Abstract Syntax)

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

variable for auxiliary lemmas

conjunction

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

variable for auxiliary lemmas

variable for terms

conjunction

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

variable for auxiliary lemmas

variable for terms

variable for term occurrences

conjunction

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

variable for auxiliary lemmas

variable for terms

variable for term occurrences

conjunction

variable for natural numbers

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

variable for auxiliary lemmas

variable for terms

variable for term occurrences

conjunction

universal quantifier

variable for natural numbers

Example Assertion in LiFtEr (in Abstract Syntax)

implication

existential quantifier

```
∃ 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

universal quantifier

variable for natural numbers

Example Assertion in LiFtEr (in Abstract Syntax)

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

implication

existential quantifier

```
∃ 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

universal quantifier

variable for natural numbers

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

lemma "rev2 xs ys = rev1 xs @ ys"
  apply(induct xs ys rule: rev2.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
```

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

lemma "rev2 xs ys = rev1 xs @ ys"
  apply(induct xs ys rule: rev2.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
```

True

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2:: "'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

lemma "rev2 xs ys = rev xs @ ys"
  apply(induct ys xs rule: rev2.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
```

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2:: "'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

lemma "rev2 xs ys = rev xs @ ys"
  apply(induct ys xs rule: rev2.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
```

False

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
  apply(induct xs arbitrary: ys)
  apply auto done
```

generalisation!

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

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

Output | Query | Sledgehammer | Symbols

15,13 (344/353) (isabelle,isabelle,UTF-8-Isabelle) 140/512MB 1 error(s) 2:57 PM

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
  apply(induct xs arbitrary: ys)
  apply auto done
```

generalisation!

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

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

Program 1

Program 2

Output | Query | Sledgehammer | Symbols

15,13 (344/353) (isabelle,isabelle,UTF-8-Isabelle) 140/512MB 1 error(s) 2:57 PM

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
  apply(induct xs arbitrary: ys)
  apply auto done
```

} Program 1

generalisation!

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

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

May I generalise *ys*, which appears as the second argument of rev2?

Program 1

Program 2

Output | Query | Sledgehammer | Symbols

15,13 (344/353) (isabelle,isabelle,UTF-8-Isabelle) 140/512MB 1 error(s) 2:57 PM

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
  apply(induct xs arbitrary: ys)
  apply auto done
```

} Program 2

} Program 1

generalisation!

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

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

Program 1

May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

Program 2

Output | Query

15,13 (344/353)

Kaiserslautern.thy (~/Workplace/PSL_Perform/PSL/)

```
primrec rev1::"'a list ⇒ 'a list" where
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

fun rev2::"'a list ⇒ 'a list ⇒ 'a list" where
  "rev2  []       ys = ys"
| "rev2 (x # xs) ys = rev2 xs (x # ys)"

theorem "rev2 xs ys = rev1 xs @ ys"
  apply(induct xs arbitrary: ys)
  apply auto done
```

} Program 2

} Program 1

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

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

Program 1

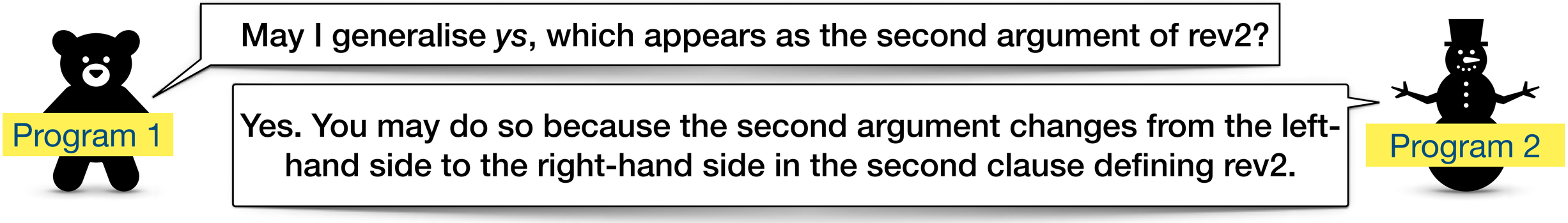
May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

Program 2

Output | Query

15,13 (344/353)


May I generalise *ys*, which appears as the second argument of rev2?
Program 1
Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.
Program 2

Abstract the communication between two programs using <u>quantifiers over definitions</u>.

Program 1

May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

Program 2

Abstract the communication between two programs using <u>quantifiers over definitions</u>.

Program 1

```
∀ arb_term : term ∈ arbitrary_term.
 ∃ f_term : term.
  ∃ f_occ : term_occ ∈ f_term.
   ∃ arb_occ ∈ arb_term.
    ∃ generalise_nth : number.
      is_or_below_nth_argument_of
         (arb_occ, generalise_nth, f_occ)
     ∧
      in_some_def
         (f_term,
          generalise_nth_argument_of,
          [generalise_nth, f_term])
```

Program 1

May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

Program 2

Abstract the communication between two programs using <u>quantifiers over definitions</u>.

Program 1

```
∀ arb_term : term ∈ arbitrary_term.
 ∃ f_term : term.
  ∃ f_occ : term_occ ∈ f_term.
   ∃ arb_occ ∈ arb_term.
    ∃ generalise_nth : number.
      is_or_below_nth_argument_of
        (arb_occ, generalise_nth, f_occ)
     ∧
      in_some_def
        (f_term,
         generalise_nth_argument_of,
         [generalise_nth, f_term])
```

Program 2 (generalise_nth_argument_of)

```
λ [generalise_nth, f_term].
 ∃ lhs_occ : term_occ.
  is_left_hand_side (lhs_occ)
 ∧
  ∃ nth_param_on_lhs : term_occ.
    is_nth_argument_of
     (nth_param_on_lhs, generalise_nth,
      lhs_occ)
   ∧
    ∃ nth_param_on_rhs : term_occ.
      ¬ are_of_same_term
       (nth_param_on_rhs, nth_param_on_lhs)
     ∧
      ∃ f_occ_on_rhs : term_occ ∈ f_term.
       is_nth_argument_of
        (nth_param_on_rhs,
         generalise_nth,
         f_occ_on_rhs)
```

Program 1

May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

Program 2

Abstract the communication between two programs using <u>quantifiers over definitions</u>.

Program 1 ← for syntactic analysis

```
∀ arb_term : term ∈ arbitrary_term.
 ∃ f_term : term.
  ∃ f_occ : term_occ ∈ f_term.
   ∃ arb_occ ∈ arb_term.
    ∃ generalise_nth : number.
      is_or_below_nth_argument_of
        (arb_occ, generalise_nth, f_occ)
     ∧
      in_some_def
        (f_term,
         generalise_nth_argument_of,
         [generalise_nth, f_term])
```

Program 2 (generalise_nth_argument_of)

```
λ [generalise_nth, f_term].
 ∃ lhs_occ : term_occ.
  is_left_hand_side (lhs_occ)
 ∧
  ∃ nth_param_on_lhs : term_occ.
    is_nth_argument_of
     (nth_param_on_lhs, generalise_nth,
      lhs_occ)
   ∧
    ∃ nth_param_on_rhs : term_occ.
      ¬ are_of_same_term
       (nth_param_on_rhs, nth_param_on_lhs)
     ∧
      ∃ f_occ_on_rhs : term_occ ∈ f_term.
       is_nth_argument_of
        (nth_param_on_rhs,
         generalise_nth,
         f_occ_on_rhs)
```

Program 1

May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

Program 2

Abstract the communication between two programs using <u>quantifiers over definitions</u>.

Program 1

← for syntactic analysis

```
∀ arb_term : term ∈ arbitrary_term.
 ∃ f_term : term.
  ∃ f_occ : term_occ ∈ f_term.
   ∃ arb_occ ∈ arb_term.
    ∃ generalise_nth : number.
      is_or_below_nth_argument_of
        (arb_occ, generalise_nth, f_occ)
     ∧
      in_some_def
        (f_term,
         generalise_nth_argument_of,
         [generalise_nth, f_term])
```

↓ for semantic analysis

Program 2 (generalise_nth_argument_of)

```
λ [generalise_nth, f_term].
 ∃ lhs_occ : term_occ.
  is_left_hand_side (lhs_occ)
 ∧
  ∃ nth_param_on_lhs : term_occ.
    is_nth_argument_of
     (nth_param_on_lhs, generalise_nth,
      lhs_occ)
   ∧
    ∃ nth_param_on_rhs : term_occ.
      ¬ are_of_same_term
       (nth_param_on_rhs, nth_param_on_lhs)
     ∧
      ∃ f_occ_on_rhs : term_occ ∈ f_term.
       is_nth_argument_of
        (nth_param_on_rhs,
         generalise_nth,
         f_occ_on_rhs)
```

Program 1

May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

Program 2

Abstract the communication between two programs using <u>quantifiers over definitions</u>.

Program 1

← for syntactic analysis

```
∀ arb_term : term ∈ arbitrary_term.
 ∃ f_term : term.
  ∃ f_occ : term_occ ∈ f_term.
   ∃ arb_occ ∈ arb_term.
    ∃ generalise_nth : number.
      is_or_below_nth_argument_of
        (arb_occ, generalise_nth, f_occ)
     ∧
      in_some_def
        (f_term,
         generalise_nth_argument_of,
         [generalise_nth, f_term])
```

↓ for semantic analysis

Program 2 (generalise_nth_argument_of)

```
λ [generalise_nth, f_term].
 ∃ lhs_occ : term_occ.
  is_left_hand_side (lhs_occ)
 ∧
  ∃ nth_param_on_lhs : term_occ.
    is_nth_argument_of
     (nth_param_on_lhs, generalise_nth,
      lhs_occ)
   ∧
    ∃ nth_param_on_rhs : term_occ.
      ¬ are_of_same_term
       (nth_param_on_rhs, nth_param_on_lhs)
     ∧
      ∃ f_occ_on_rhs : term_occ ∈ f_term.
       is_nth_argument_of
        (nth_param_on_rhs,
         generalise_nth,
         f_occ_on_rhs)
```

Program 1

May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

Program 2

Abstract the communication between two programs using <u>quantifiers over definitions</u>.

Program 1

← for syntactic analysis

```
∀ arb_term : term ∈ arbitrary_term.
 ∃ f_term : term.
  ∃ f_occ : term_occ ∈ f_term.
   ∃ arb_occ ∈ arb_term.
    ∃ generalise_nth : number.
      is_or_below_nth_argument_of
        (arb_occ, generalise_nth, f_occ)
     ∧
      in_some_def
        (f_term,
         generalise_nth_argument_of,
         [generalise_nth, f_term])
```

∃ over definitions

↓ for semantic analysis

Program 2 (generalise_nth_argument_of)

```
λ [generalise_nth, f_term].
 ∃ lhs_occ : term_occ.
  is_left_hand_side (lhs_occ)
 ∧
  ∃ nth_param_on_lhs : term_occ.
    is_nth_argument_of
      (nth_param_on_lhs, generalise_nth,
       lhs_occ)
   ∧
    ∃ nth_param_on_rhs : term_occ.
      ¬ are_of_same_term
        (nth_param_on_rhs, nth_param_on_lhs)
     ∧
      ∃ f_occ_on_rhs : term_occ ∈ f_term.
        is_nth_argument_of
          (nth_param_on_rhs,
           generalise_nth,
           f_occ_on_rhs)
```

Program 1

May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

Program 2

Abstract the communication between two programs using <u>quantifiers over definitions</u>.

Program 1

← for syntactic analysis

```
∀ arb_term : term ∈ arbitrary_term.
 ∃ f_term : term.
  ∃ f_occ : term_occ ∈ f_term.
   ∃ arb_occ ∈ arb_term.
    ∃ generalise_nth : number.
      is_or_below_nth_argument_of
        (arb_occ, generalise_nth, f_occ)
     ∧
      in_some_def
        (f_term,
         generalise_nth_argument_of,
         [generalise_nth, f_term])
```

∃ over definitions

← in some defining clause of

↓ for semantic analysis

Program 2 (generalise_nth_argument_of)

```
λ [generalise_nth, f_term].
 ∃ lhs_occ : term_occ.
  is_left_hand_side (lhs_occ)
 ∧
  ∃ nth_param_on_lhs : term_occ.
    is_nth_argument_of
      (nth_param_on_lhs, generalise_nth,
       lhs_occ)
   ∧
    ∃ nth_param_on_rhs : term_occ.
      ¬ are_of_same_term
        (nth_param_on_rhs, nth_param_on_lhs)
     ∧
      ∃ f_occ_on_rhs : term_occ ∈ f_term.
        is_nth_argument_of
          (nth_param_on_rhs,
           generalise_nth,
           f_occ_on_rhs)
```

Program 1

May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

Program 2

Abstract the communication between two programs using <u>quantifiers over definitions</u>.

Program 1

← for syntactic analysis

```
∀ arb_term : term ∈ arbitrary_term.
 ∃ f_term : term.
  ∃ f_occ : term_occ ∈ f_term.
   ∃ arb_occ ∈ arb_term.
    ∃ generalise_nth : number.
      is_or_below_nth_argument_of
        (arb_occ, generalise_nth, f_occ)
     ∧
      in_some_def
        (f_term,
         generalise_nth_argument_of,
         [generalise_nth, f_term])
```

∃ over definitions

← in some defining clause of

← rev2, check if

↓ for semantic analysis

Program 2 (generalise_nth_argument_of)

```
λ [generalise_nth, f_term].
 ∃ lhs_occ : term_occ.
  is_left_hand_side (lhs_occ)
 ∧
  ∃ nth_param_on_lhs : term_occ.
    is_nth_argument_of
      (nth_param_on_lhs, generalise_nth,
       lhs_occ)
   ∧
    ∃ nth_param_on_rhs : term_occ.
      ¬ are_of_same_term
        (nth_param_on_rhs, nth_param_on_lhs)
     ∧
      ∃ f_occ_on_rhs : term_occ ∈ f_term.
        is_nth_argument_of
          (nth_param_on_rhs,
           generalise_nth,
           f_occ_on_rhs)
```

Program 1

May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

Program 2

Abstract the communication between two programs using <u>quantifiers over definitions</u>.

Program 1 ← for syntactic analysis

```
∀ arb_term : term ∈ arbitrary_term.
 ∃ f_term : term.
  ∃ f_occ : term_occ ∈ f_term.
   ∃ arb_occ ∈ arb_term.
    ∃ generalise_nth : number.
      is_or_below_nth_argument_of
        (arb_occ, generalise_nth, f_occ)
     ∧
      in_some_def
        (f_term,
         generalise_nth_argument_of,
         [generalise_nth, f_term])
```

∃ over definitions
← in some defining clause of
← rev2, check if
← Program 2 returns True when

↓ for semantic analysis

Program 2 (generalise_nth_argument_of)

```
λ [generalise_nth, f_term].
 ∃ lhs_occ : term_occ.
  is_left_hand_side (lhs_occ)
 ∧
  ∃ nth_param_on_lhs : term_occ.
    is_nth_argument_of
      (nth_param_on_lhs, generalise_nth,
       lhs_occ)
   ∧
    ∃ nth_param_on_rhs : term_occ.
      ¬ are_of_same_term
        (nth_param_on_rhs, nth_param_on_lhs)
     ∧
      ∃ f_occ_on_rhs : term_occ ∈ f_term.
        is_nth_argument_of
          (nth_param_on_rhs,
           generalise_nth,
           f_occ_on_rhs)
```

Program 1

May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

Program 2

Abstract the communication between two programs using <u>quantifiers over definitions</u>.

Program 1 ← for syntactic analysis

```
∀ arb_term : term ∈ arbitrary_term.
 ∃ f_term : term.
  ∃ f_occ : term_occ ∈ f_term.
   ∃ arb_occ ∈ arb_term.
    ∃ generalise_nth : number.
      is_or_below_nth_argument_of
        (arb_occ, generalise_nth, f_occ)
     ∧
      in_some_def
        (f_term,
         generalise_nth_argument_of,
         [generalise_nth, f_term])
```

∃ over definitions
← in some defining clause of
← rev2, check if
← Program 2 returns True when
← are passed as arguments.

↓ for semantic analysis

Program 2 (generalise_nth_argument_of)

```
λ [generalise_nth, f_term].
 ∃ lhs_occ : term_occ.
  is_left_hand_side (lhs_occ)
 ∧
  ∃ nth_param_on_lhs : term_occ.
    is_nth_argument_of
      (nth_param_on_lhs, generalise_nth,
       lhs_occ)
   ∧
    ∃ nth_param_on_rhs : term_occ.
      ¬ are_of_same_term
        (nth_param_on_rhs, nth_param_on_lhs)
     ∧
      ∃ f_occ_on_rhs : term_occ ∈ f_term.
        is_nth_argument_of
          (nth_param_on_rhs,
           generalise_nth,
           f_occ_on_rhs)
```

Program 1

May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

Program 2

Abstract the communication between two programs using <u>quantifiers over definitions</u>.

Program 1 ← for syntactic analysis

```
∀ arb_term : term ∈ arbitrary_term.
 ∃ f_term : term.
  ∃ f_occ : term_occ ∈ f_term.
   ∃ arb_occ ∈ arb_term.
    ∃ generalise_nth : number.
      is_or_below_nth_argument_of
        (arb_occ, generalise_nth, f_occ)
     ∧
      in_some_def
        (f_term,
         generalise_nth_argument_of,
         [generalise_nth, f_term])
```

2

∃ over definitions
← in some defining clause of
← rev2, check if
← Program 2 returns True when
← are passed as arguments.

↓ for semantic analysis

Program 2 (generalise_nth_argument_of)

```
λ [generalise_nth, f_term].
 ∃ lhs_occ : term_occ.
  is_left_hand_side (lhs_occ)
 ∧
  ∃ nth_param_on_lhs : term_occ.
    is_nth_argument_of
      (nth_param_on_lhs, generalise_nth,
       lhs_occ)
   ∧
    ∃ nth_param_on_rhs : term_occ.
      ¬ are_of_same_term
        (nth_param_on_rhs, nth_param_on_lhs)
     ∧
      ∃ f_occ_on_rhs : term_occ ∈ f_term.
        is_nth_argument_of
          (nth_param_on_rhs,
           generalise_nth,
           f_occ_on_rhs)
```

Program 1

May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

Program 2

Abstract the communication between two programs using <u>quantifiers over definitions</u>.

Program 1 ← for syntactic analysis

```
∀ arb_term : term ∈ arbitrary_term.
 ∃ f_term : term.
  ∃ f_occ : term_occ ∈ f_term.
   ∃ arb_occ ∈ arb_term.
    ∃ generalise_nth : number.
      is_or_below_nth_argument_of
        (arb_occ, generalise_nth, f_occ)
     ∧
      in_some_def
        (f_term,
         generalise_nth_argument_of,
         [generalise_nth, f_term])
```

2 rev2

∃ over definitions
← in some defining clause of
← rev2, check if
← Program 2 returns True when
← are passed as arguments.

↓ for semantic analysis

Program 2 (generalise_nth_argument_of)

```
λ [generalise_nth, f_term].
 ∃ lhs_occ : term_occ.
  is_left_hand_side (lhs_occ)
 ∧
  ∃ nth_param_on_lhs : term_occ.
    is_nth_argument_of
      (nth_param_on_lhs, generalise_nth,
       lhs_occ)
   ∧
    ∃ nth_param_on_rhs : term_occ.
      ¬ are_of_same_term
        (nth_param_on_rhs, nth_param_on_lhs)
     ∧
      ∃ f_occ_on_rhs : term_occ ∈ f_term.
        is_nth_argument_of
          (nth_param_on_rhs,
           generalise_nth,
           f_occ_on_rhs)
```

Program 1

May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

Program 2

↓ for semantic analysis

Program 2 (generalise_nth_argument_of)

```
λ [generalise_nth, f_term].
 ∃ lhs_occ : term_occ.
  is_left_hand_side (lhs_occ)
 ∧
  ∃ nth_param_on_lhs : term_occ.
    is_nth_argument_of
     (nth_param_on_lhs, generalise_nth,
      lhs_occ)
   ∧
    ∃ nth_param_on_rhs : term_occ.
      ¬ are_of_same_term
       (nth_param_on_rhs, nth_param_on_lhs)
     ∧
      ∃ f_occ_on_rhs : term_occ ∈ f_term.
       is_nth_argument_of
        (nth_param_on_rhs,
         generalise_nth,
         f_occ_on_rhs)
```

Program 1 ← for syntactic analysis

```
∀ arb_term : term ∈ arbitrary_term.
 ∃ f_term : term.
  ∃ f_occ : term_occ ∈ f_term.
   ∃ arb_occ ∈ arb_term.
    ∃ generalise_nth : number.
      is_or_below_nth_argument_of
        (arb_occ, generalise_nth, f_occ)
     ∧
      in_some_def
        (f_term,
         generalise_nth_argument_of,
         [generalise_nth, f_term])
```

2 rev2

∃ over definitions
← in some defining clause of
← rev2, check if
← Program 2 returns True when
← are passed as arguments.

Program 1

May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

Program 2

```
theorem "rev2 xs ys = rev1 xs @ ys"
 apply(induct xs arbitrary: ys)
 apply auto done
```

↓ for semantic analysis

Program 2 (generalise_nth_argument_of)

```
λ [generalise_nth, f_term].
 ∃ lhs_occ : term_occ.
  is_left_hand_side (lhs_occ)
 ∧
  ∃ nth_param_on_lhs : term_occ.
    is_nth_argument_of
     (nth_param_on_lhs, generalise_nth,
      lhs_occ)
   ∧
    ∃ nth_param_on_rhs : term_occ.
      ¬ are_of_same_term
       (nth_param_on_rhs, nth_param_on_lhs)
     ∧
      ∃ f_occ_on_rhs : term_occ ∈ f_term.
       is_nth_argument_of
        (nth_param_on_rhs,
         generalise_nth,
         f_occ_on_rhs)
```

Program 1 ← for syntactic analysis

```
∀ arb_term : term ∈ arbitrary_term.
 ∃ f_term : term.
  ∃ f_occ : term_occ ∈ f_term.
   ∃ arb_occ ∈ arb_term.
    ∃ generalise_nth : number.
      is_or_below_nth_argument_of
        (arb_occ, generalise_nth, f_occ)
     ∧
      in_some_def
        (f_term,
         generalise_nth_argument_of,
         [generalise_nth, f_term])
```

2 rev2

∃ over definitions
← in some defining clause of
← rev2, check if
← Program 2 returns True when
← are passed as arguments.

Program 1

May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

Program 2

```
theorem "rev2 xs ys = rev1 xs @ ys"
 apply(induct xs arbitrary: ys)
 apply auto done
```

↓ for semantic analysis

Program 2 (generalise_nth_argument_of)

```
λ [generalise_nth, f_term].
 ∃ lhs_occ : term_occ.
  is_left_hand_side (lhs_occ)
 ∧
  ∃ nth_param_on_lhs : term_occ.
    is_nth_argument_of
     (nth_param_on_lhs, generalise_nth,
      lhs_occ)
   ∧
    ∃ nth_param_on_rhs : term_occ.
      ¬ are_of_same_term
       (nth_param_on_rhs, nth_param_on_lhs)
     ∧
      ∃ f_occ_on_rhs : term_occ ∈ f_term.
       is_nth_argument_of
        (nth_param_on_rhs,
         generalise_nth,
         f_occ_on_rhs)
```

Program 1 ← for syntactic analysis

```
∀ arb_term : term ∈ arbitrary_term.
 ∃ f_term : term.
  ∃ f_occ : term_occ ∈ f_term.
   ∃ arb_occ ∈ arb_term.
    ∃ generalise_nth : number.
      is_or_below_nth_argument_of
        (arb_occ, generalise_nth, f_occ)
     ∧
      in_some_def
        (f_term,
         generalise_nth_argument_of,
         [generalise_nth, f_term])
```

2 rev2

∃ over definitions
← in some defining clause of
← rev2, check if
← Program 2 returns True when
← are passed as arguments.

Program 1

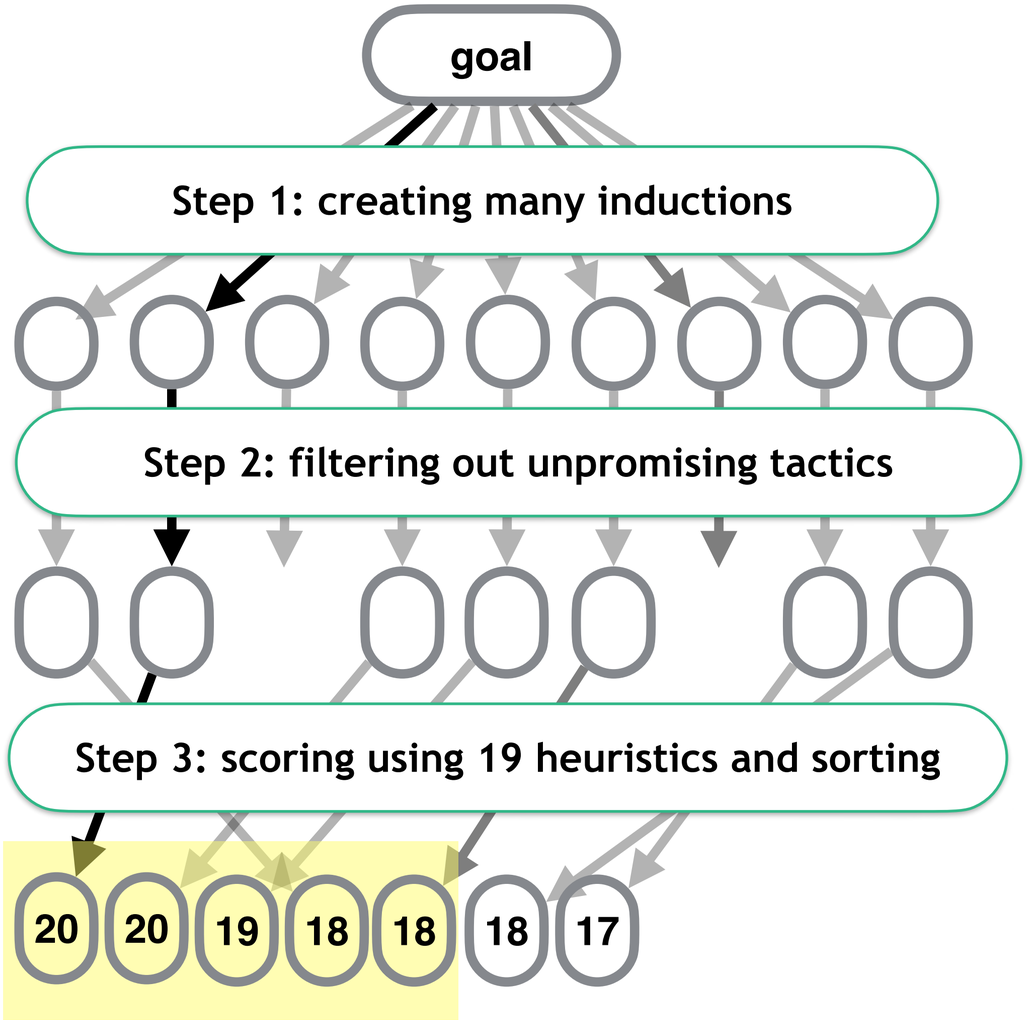
May I generalise *ys*, which appears as the second argument of rev2?

Yes. You may do so because the second argument changes from the left-hand side to the right-hand side in the second clause defining rev2.

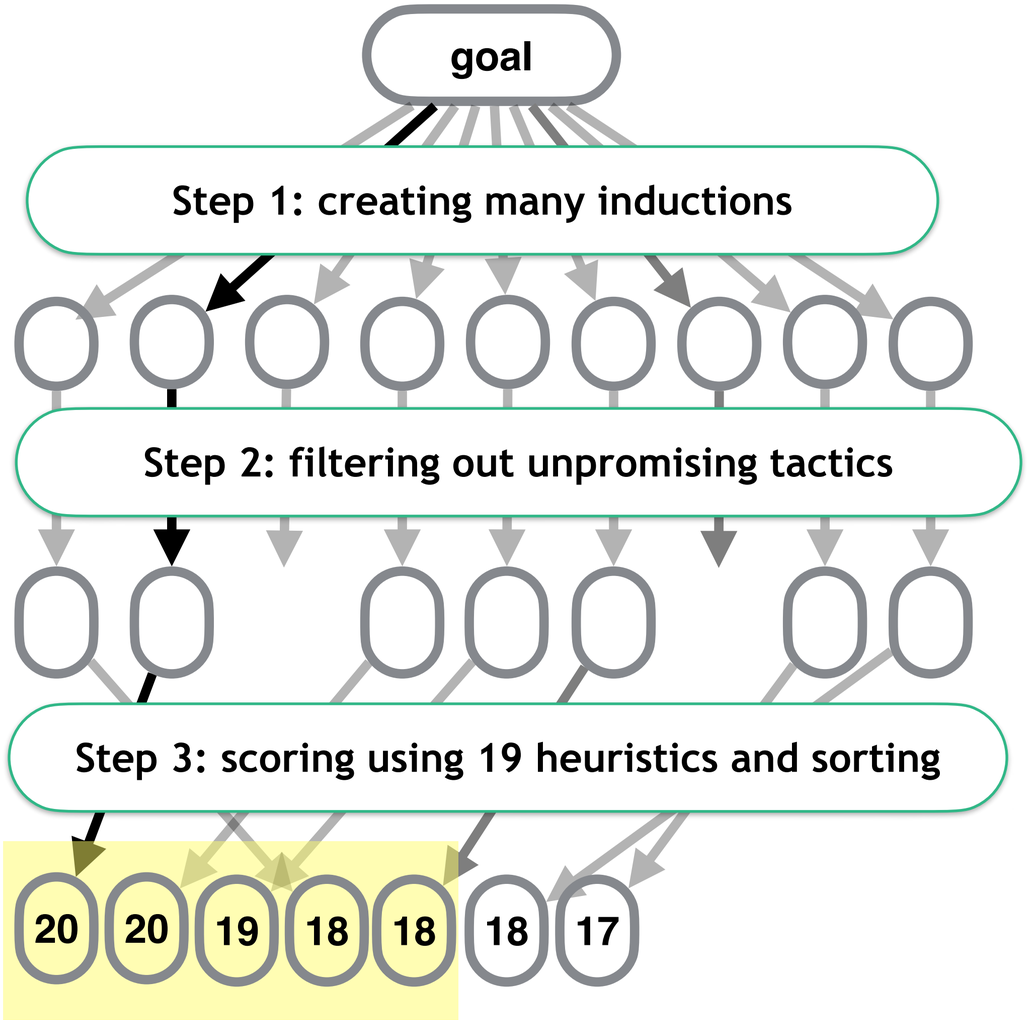
Program 2

smart_induct (based on LiFtEr)

sem_ind (based on SeLFiE)

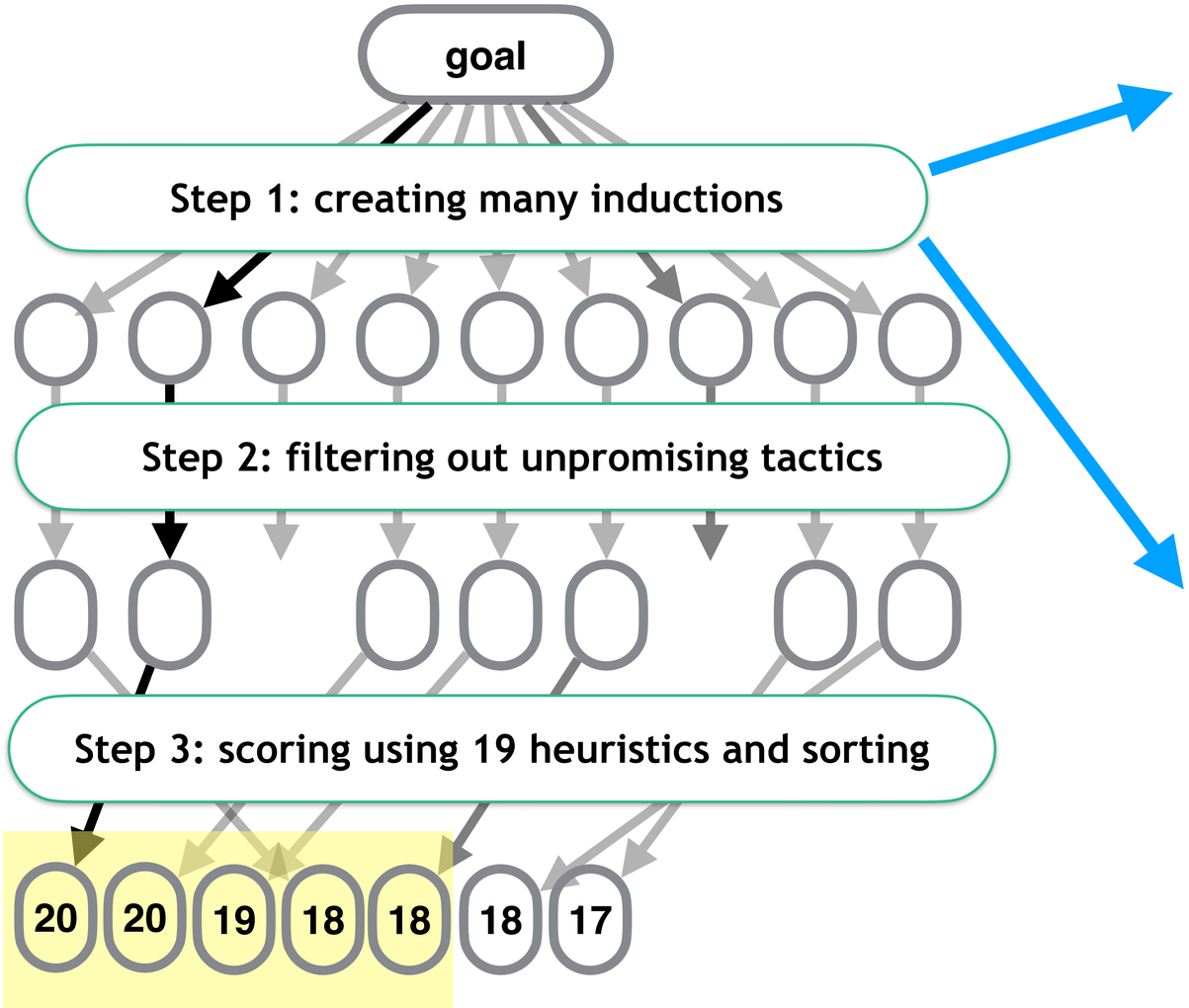


smart_induct (based on LiFtEr)

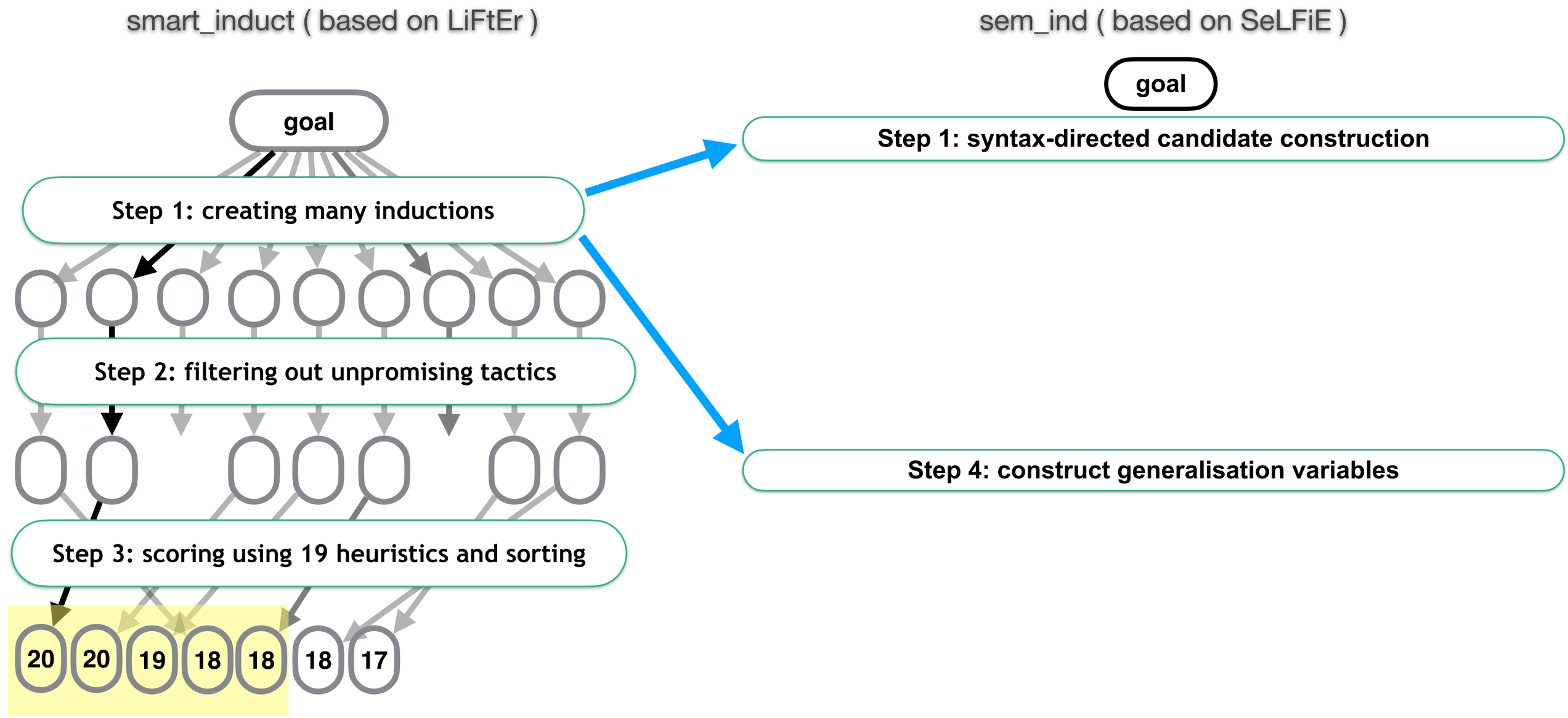


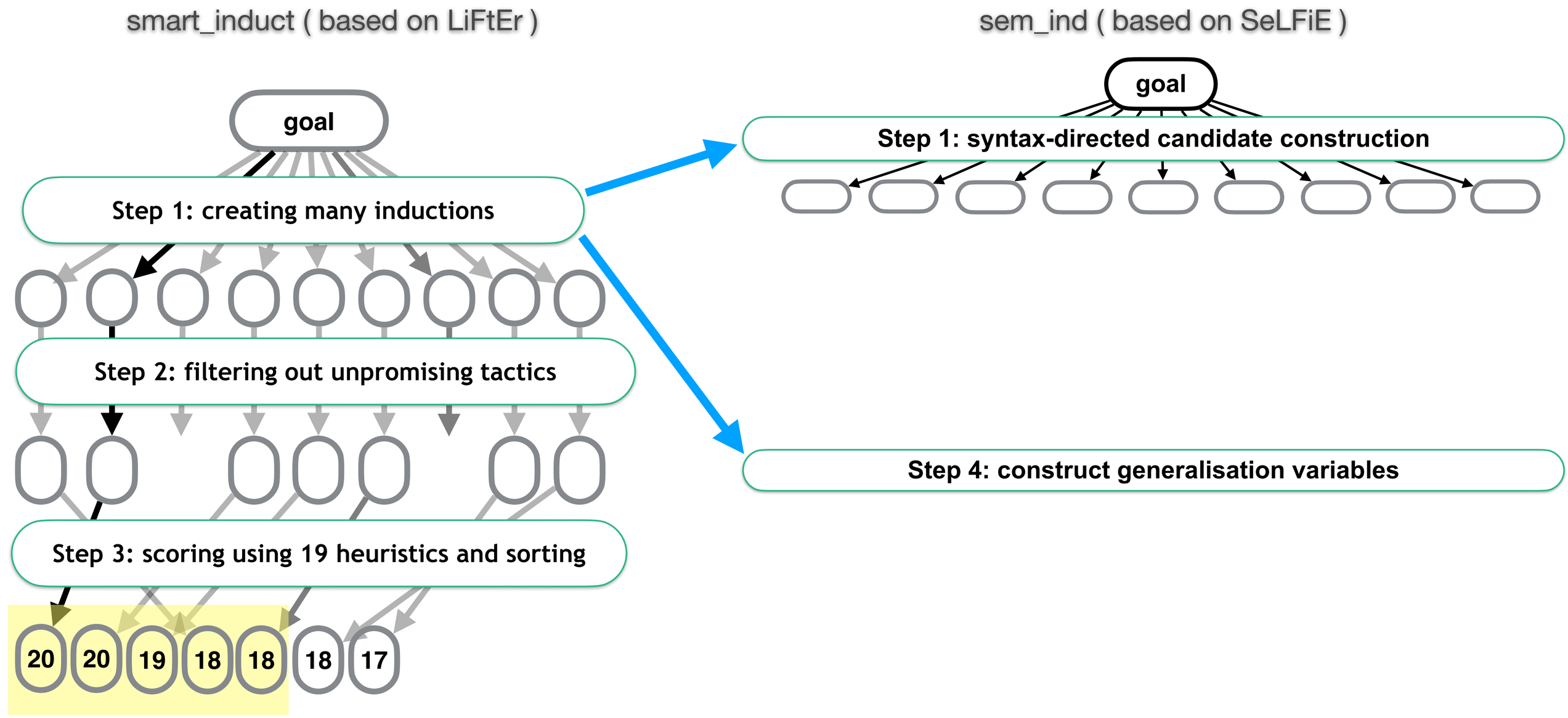
sem_ind (based on SeLFiE)

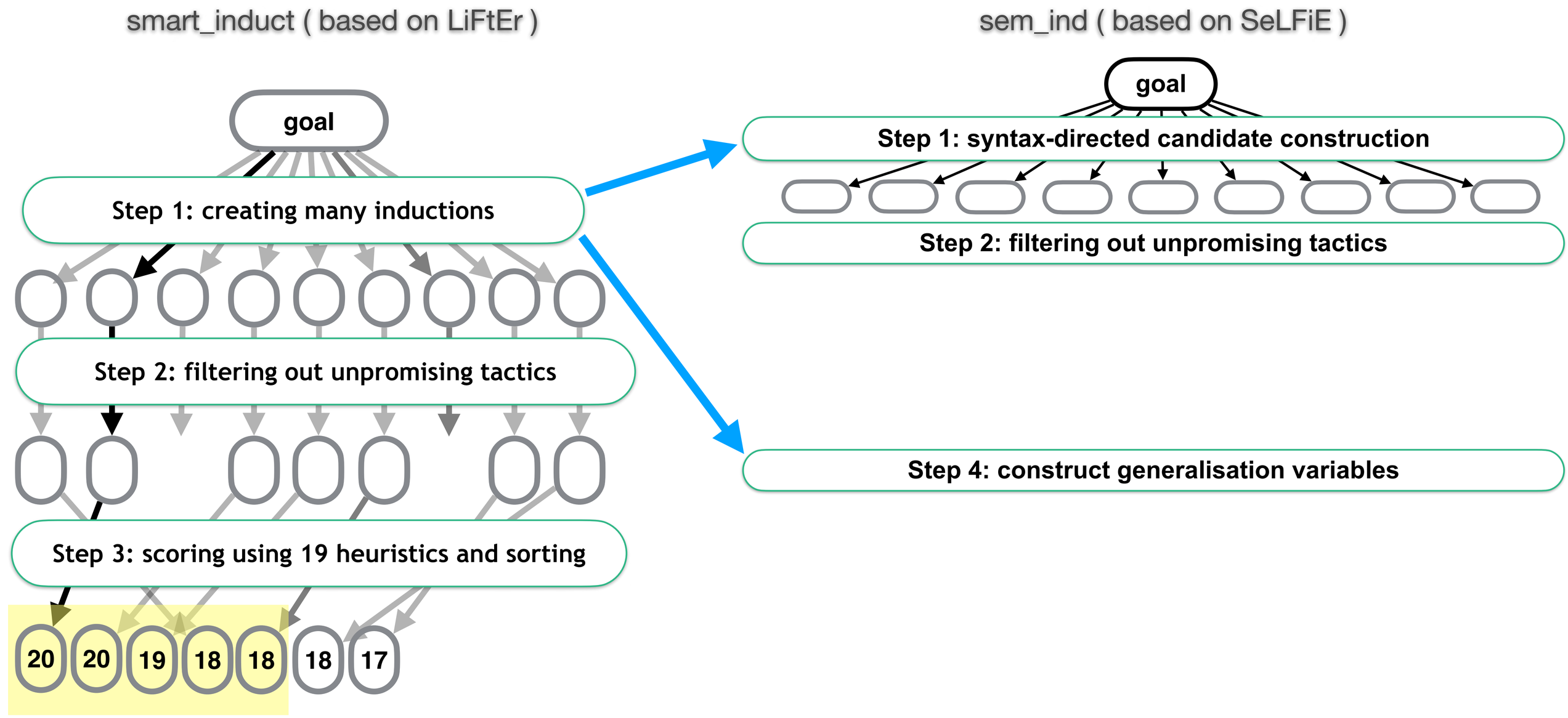
smart_induct (based on LiFtEr)

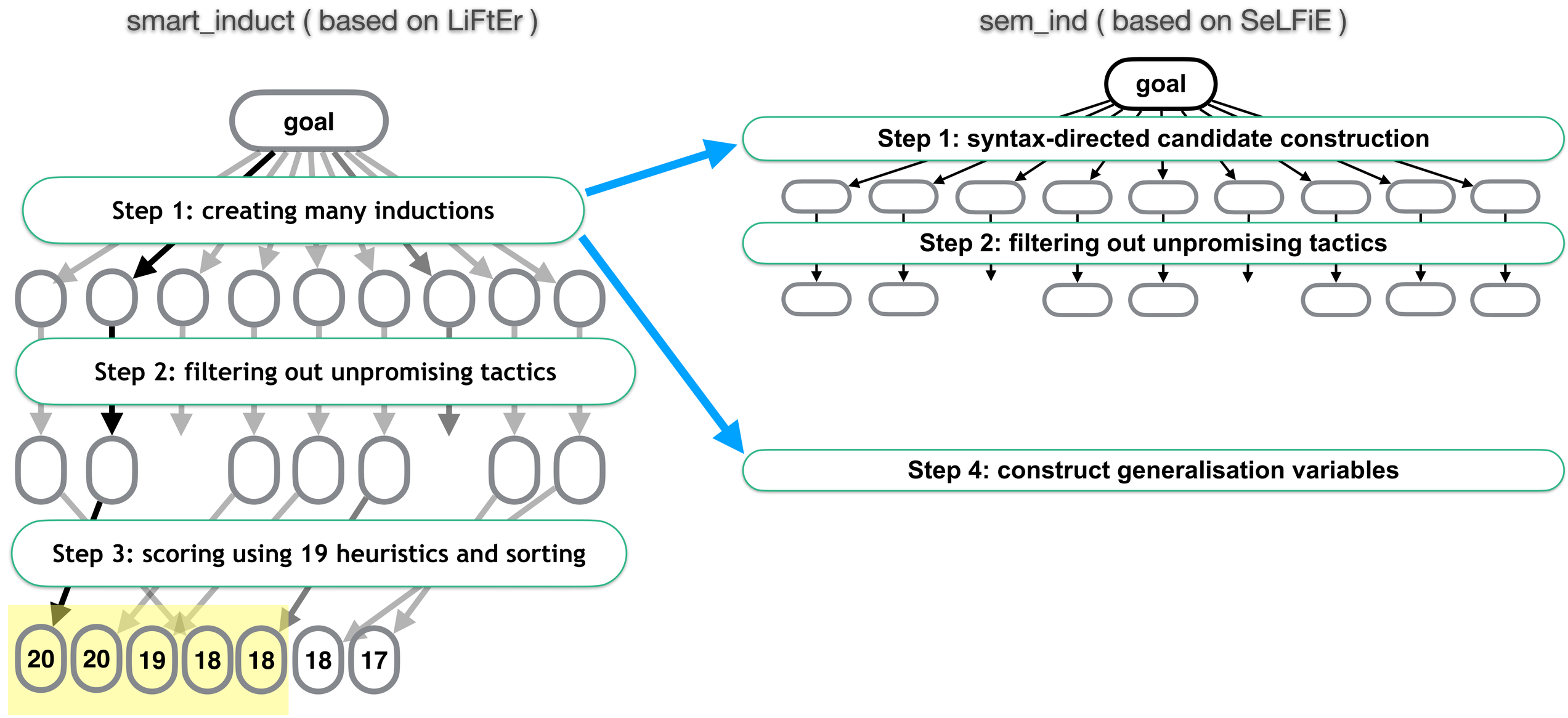


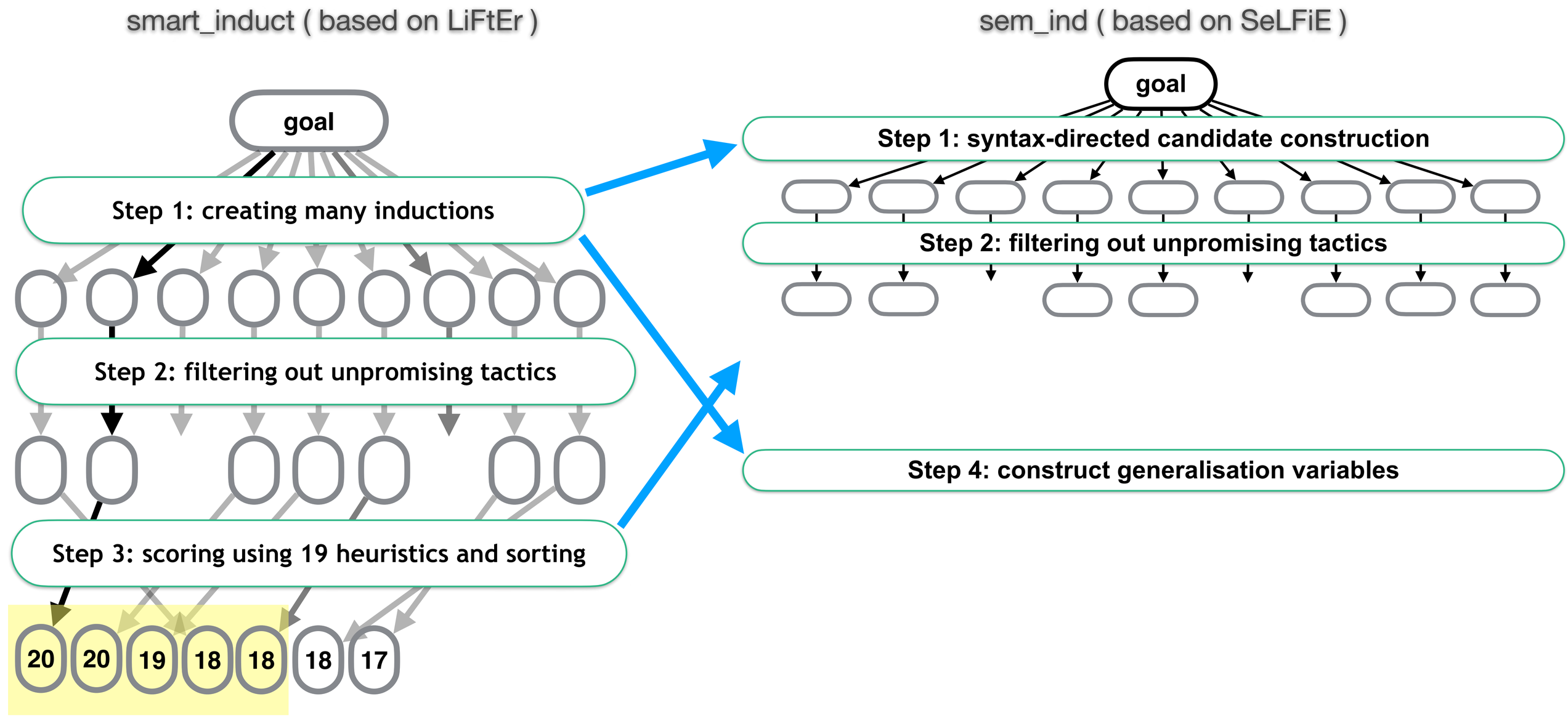
sem_ind (based on SeLFiE)

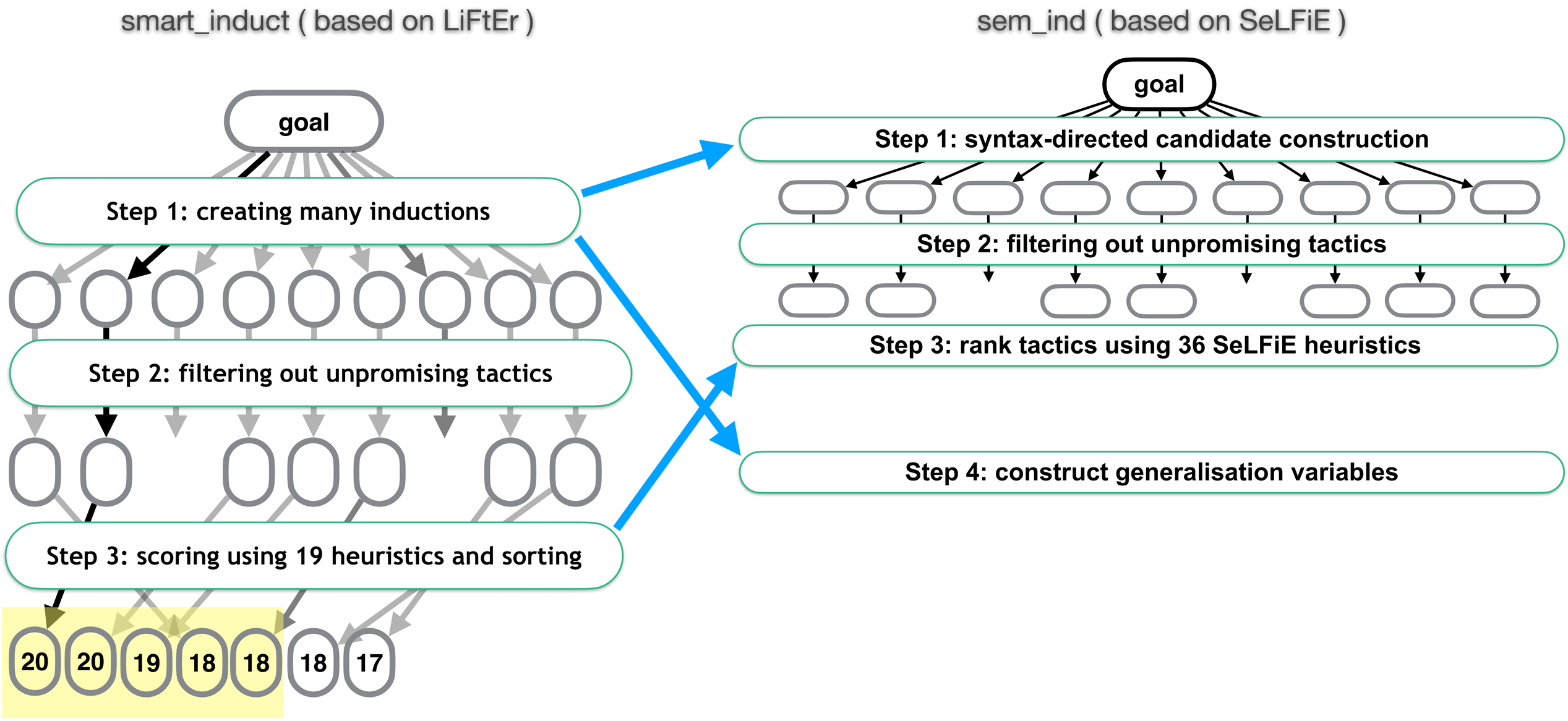

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 4: construct generalisation variables

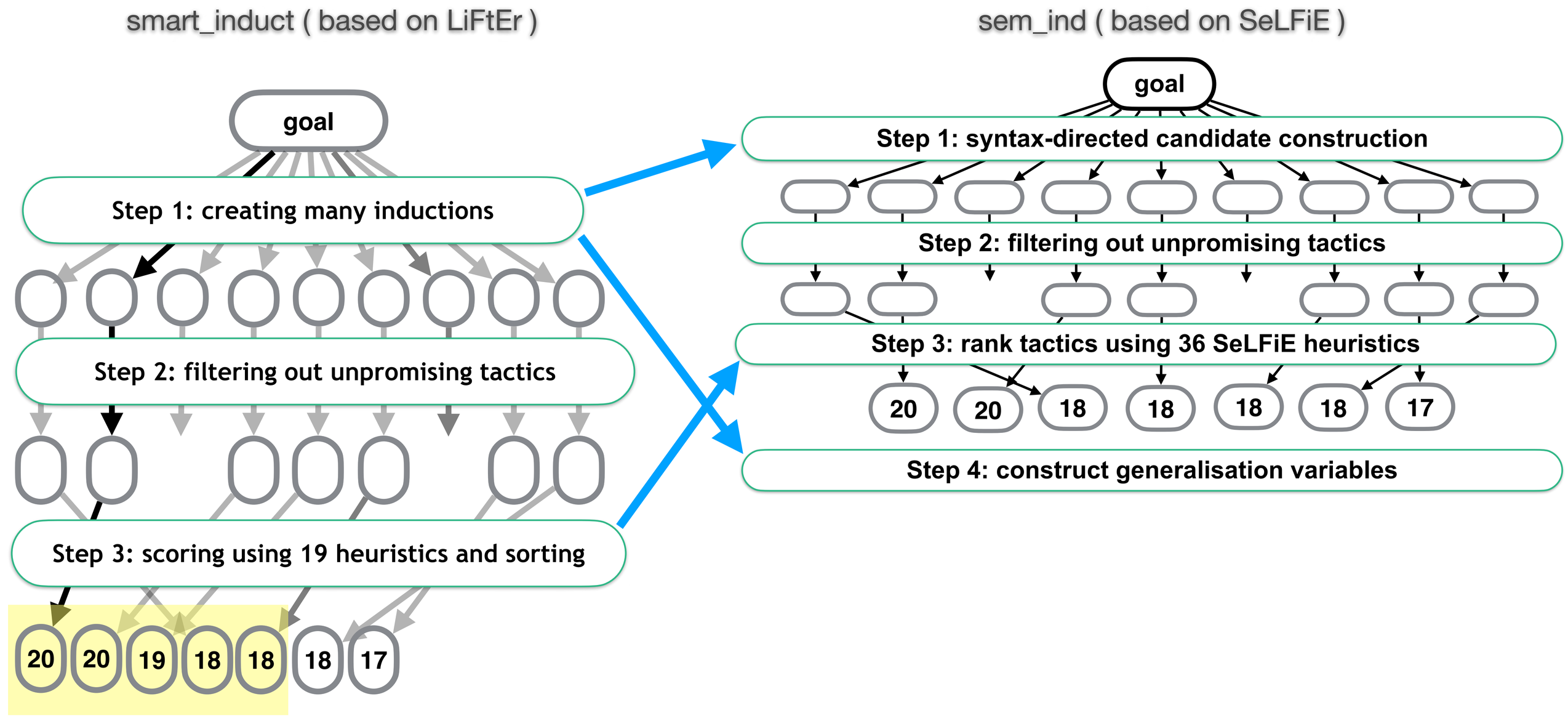

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 4: construct generalisation variables

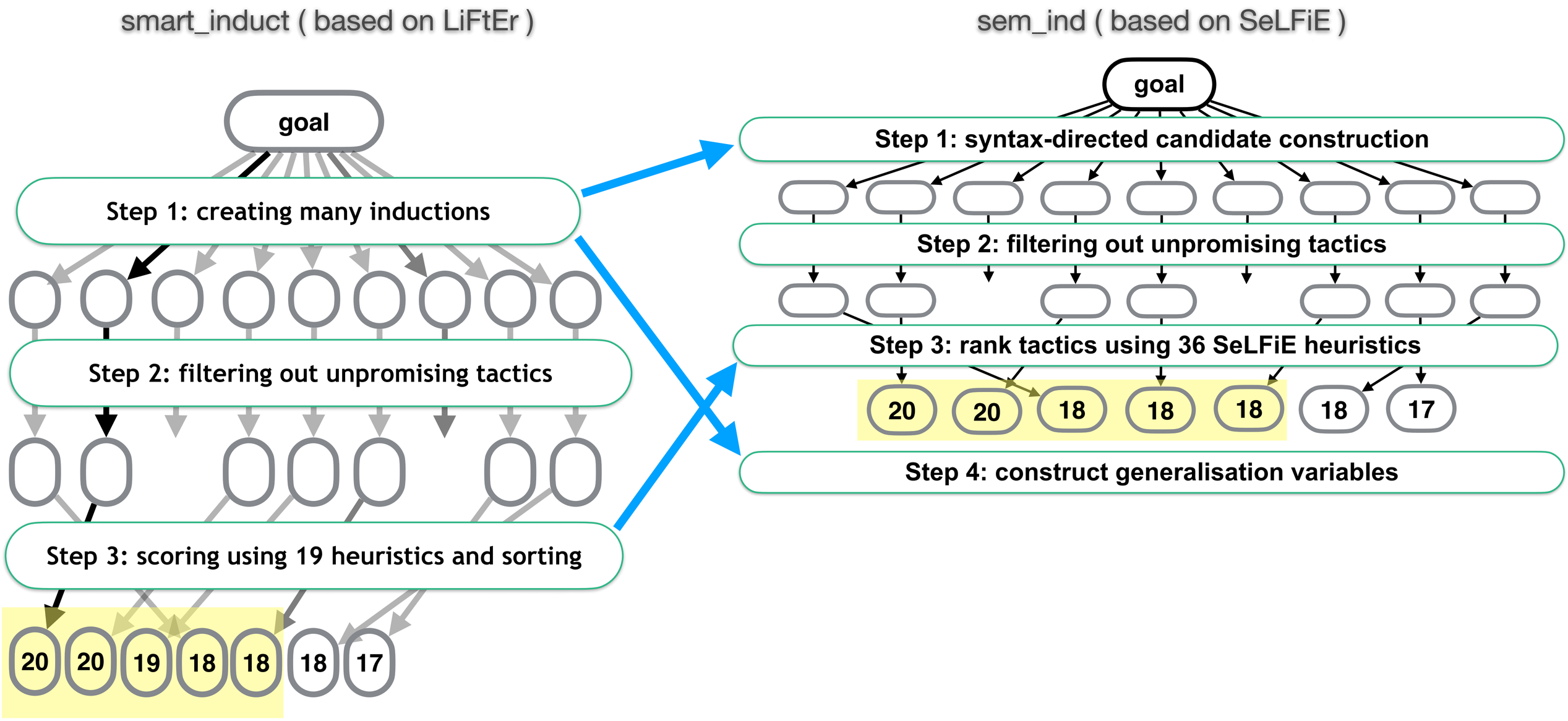

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step 4: construct generalisation variables

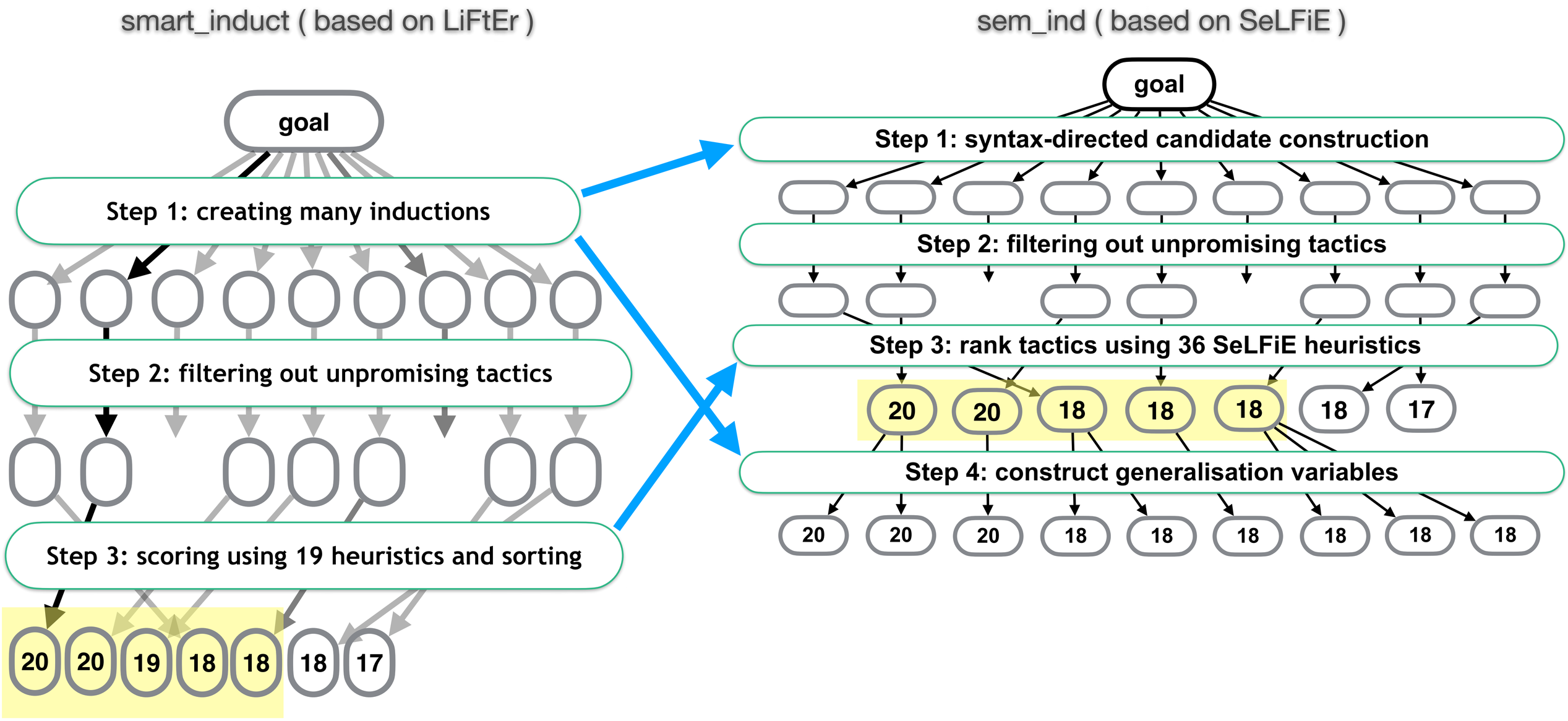

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step 4: construct generalisation variables

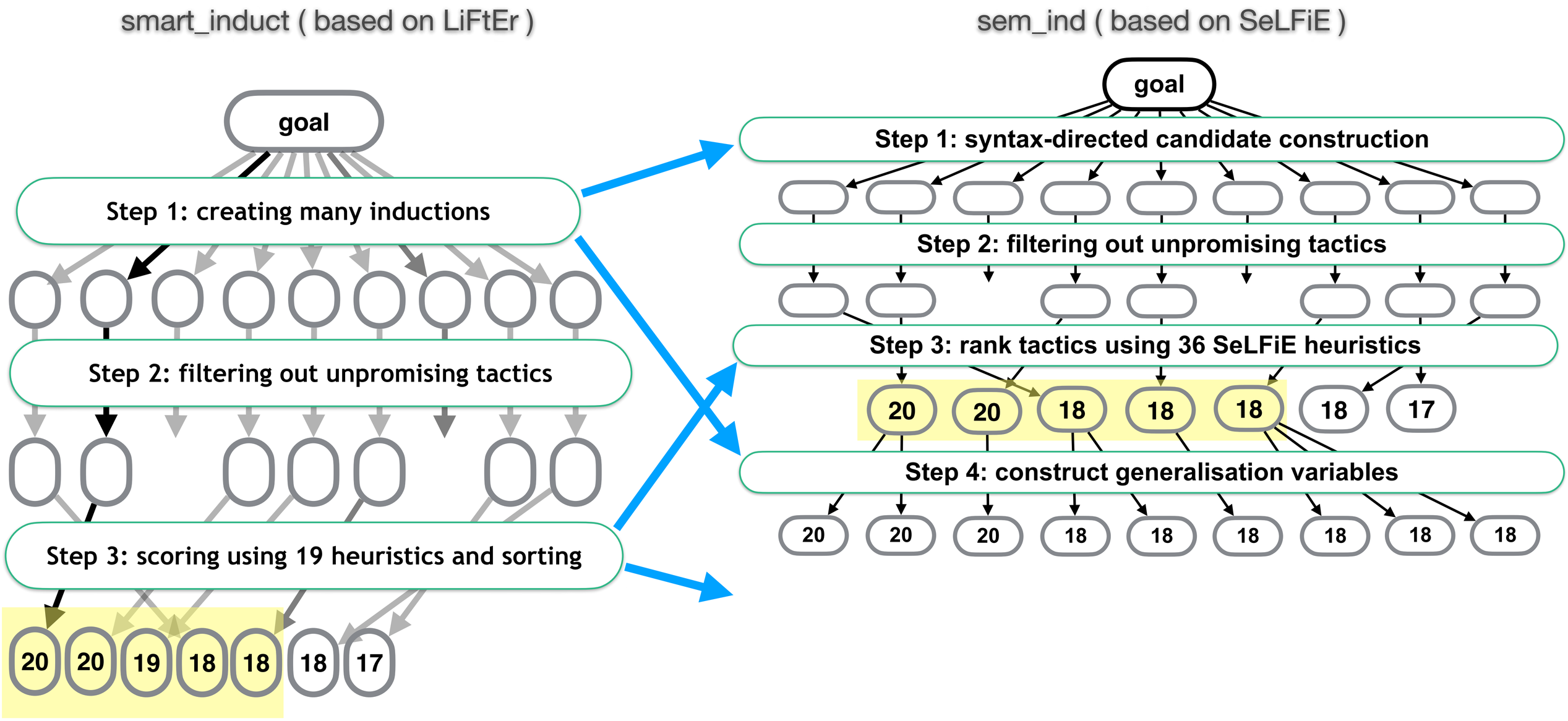

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step 4: construct generalisation variables

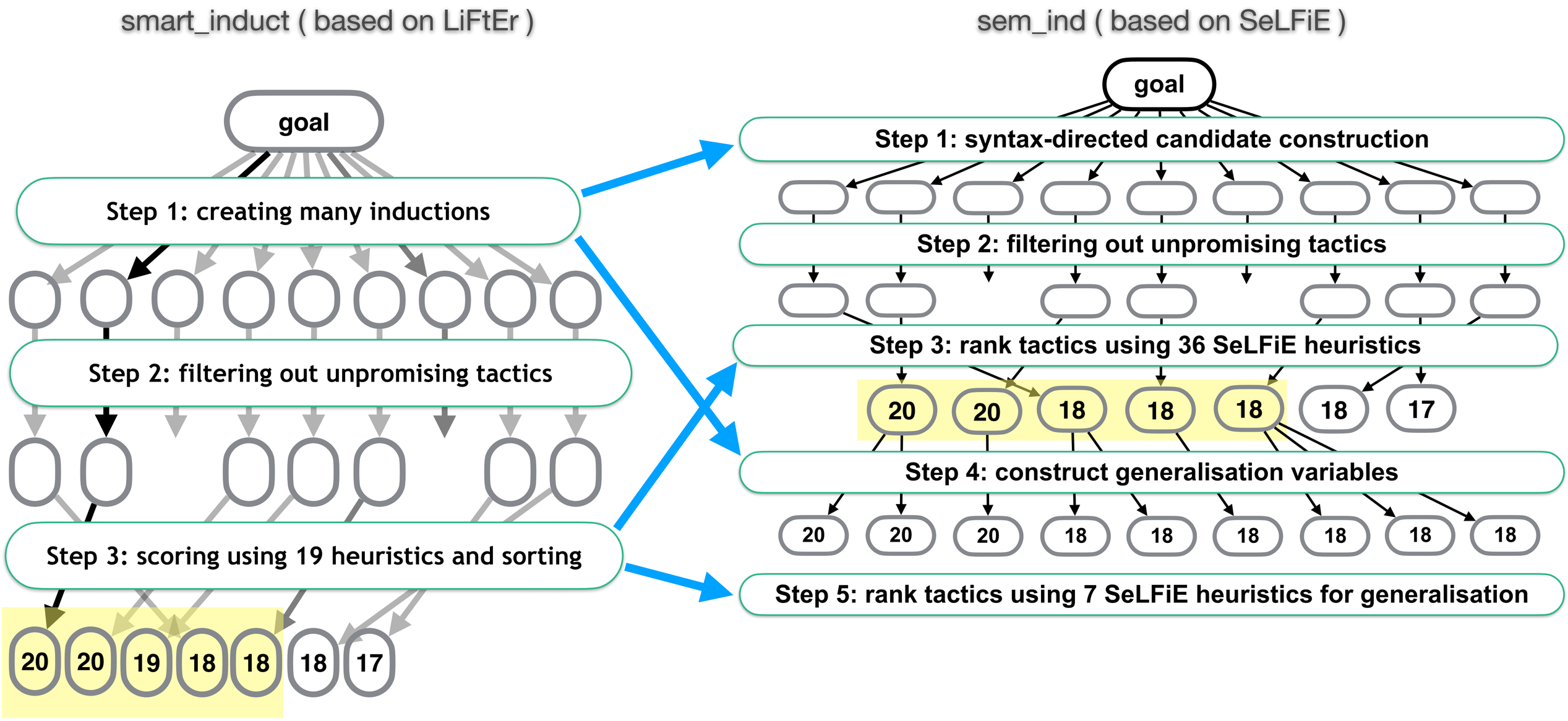

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step 3: rank tactics using 36 SeLFiE heuristics
Step 4: construct generalisation variables

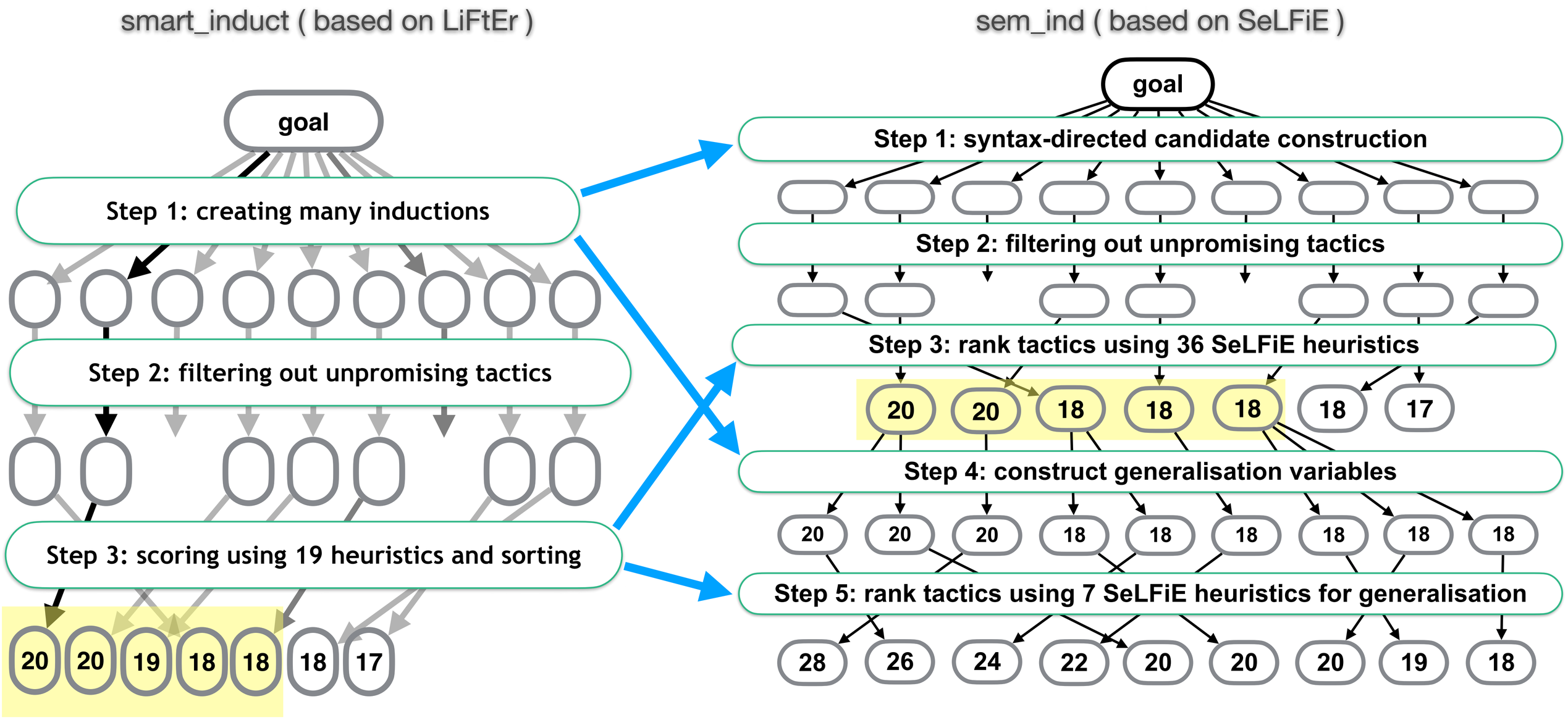

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step 3: rank tactics using 36 SeLFiE heuristics
20
20
18
18
18
18
17
Step 4: construct generalisation variables

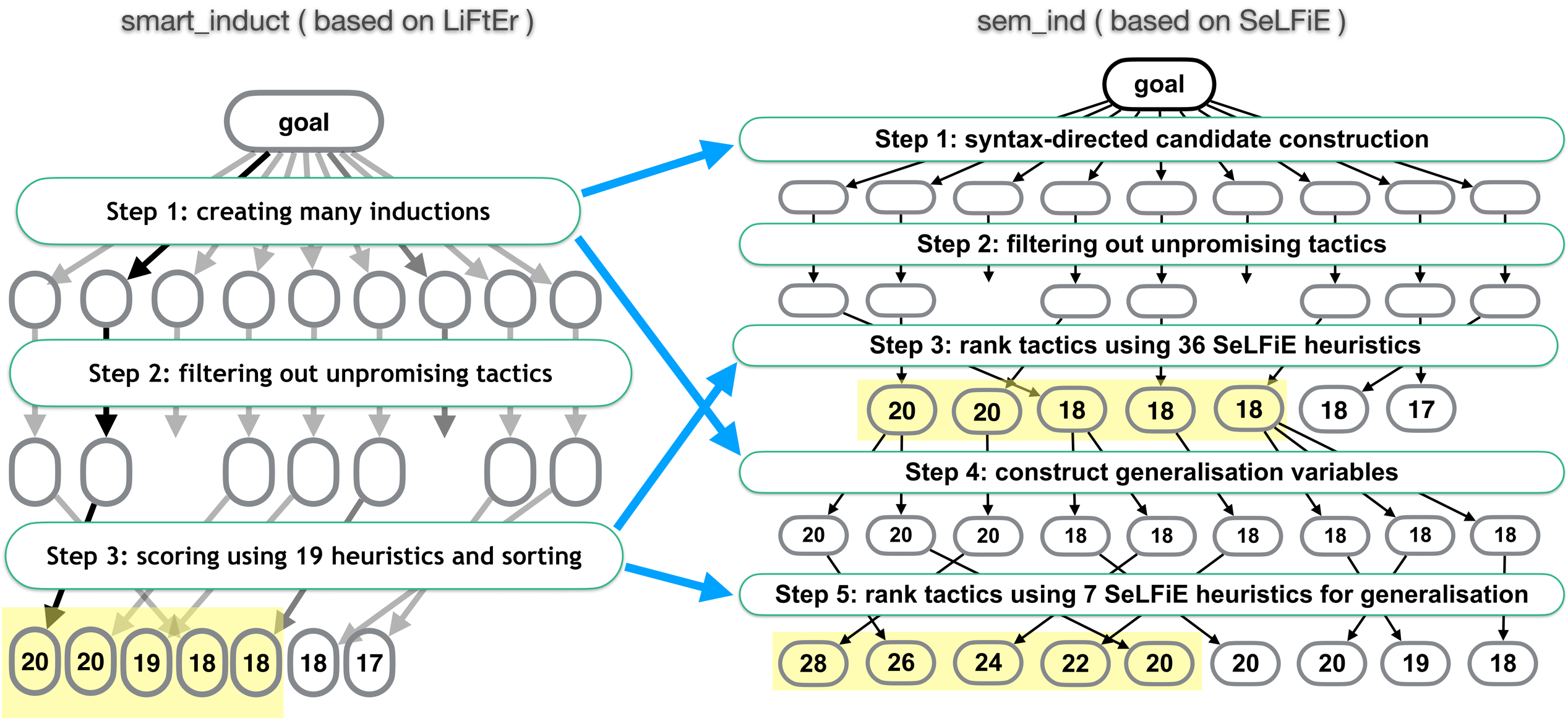

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step 3: rank tactics using 36 SeLFiE heuristics
20
20
18
18
18
18
17
Step 4: construct generalisation variables


smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step 3: rank tactics using 36 SeLFiE heuristics
20
20
18
18
18
18
17
Step 4: construct generalisation variables
20
20
20
18
18
18
18
18
18

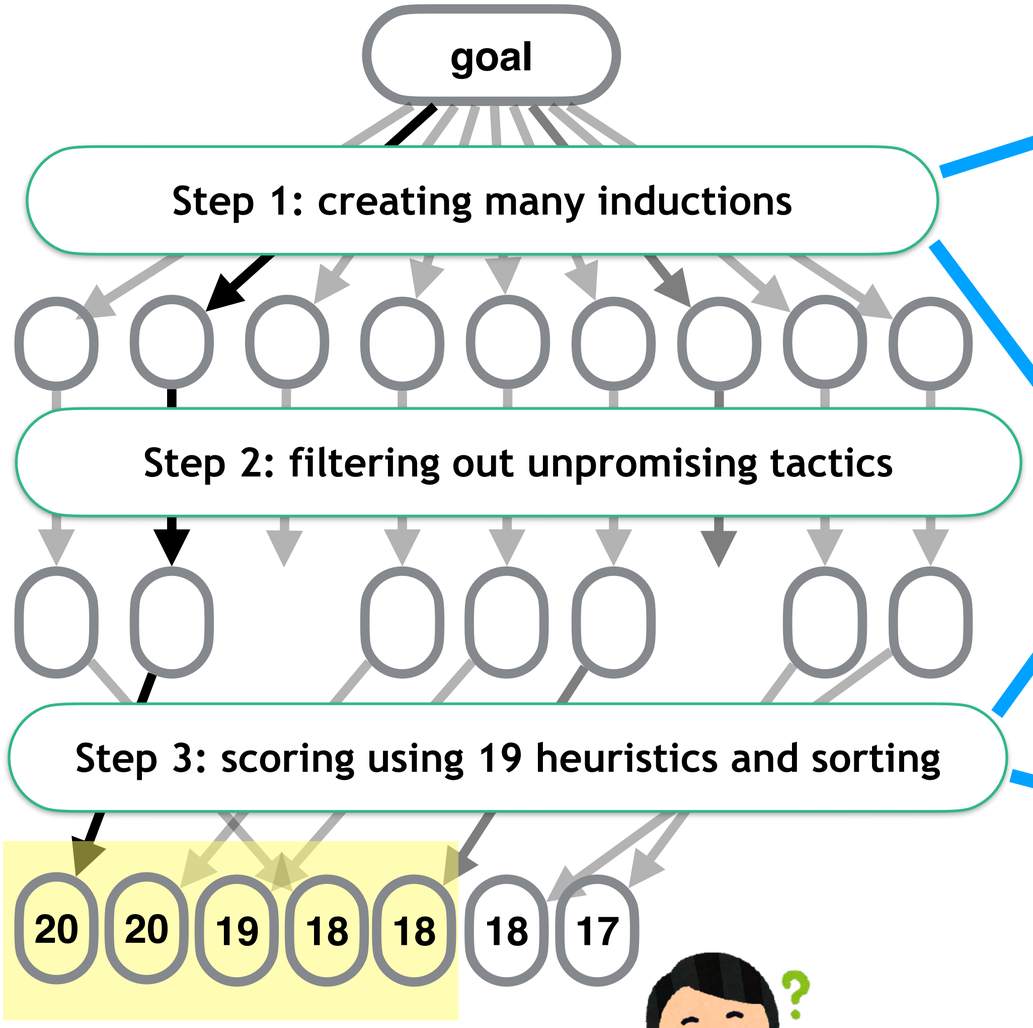

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step 3: rank tactics using 36 SeLFiE heuristics
20
20
18
18
18
18
17
Step 4: construct generalisation variables
20
20
20
18
18
18
18
18
18


smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step 3: rank tactics using 36 SeLFiE heuristics
20
20
18
18
18
18
17
Step 4: construct generalisation variables
20
20
20
18
18
18
18
18
18
Step 5: rank tactics using 7 SeLFiE heuristics for generalisation

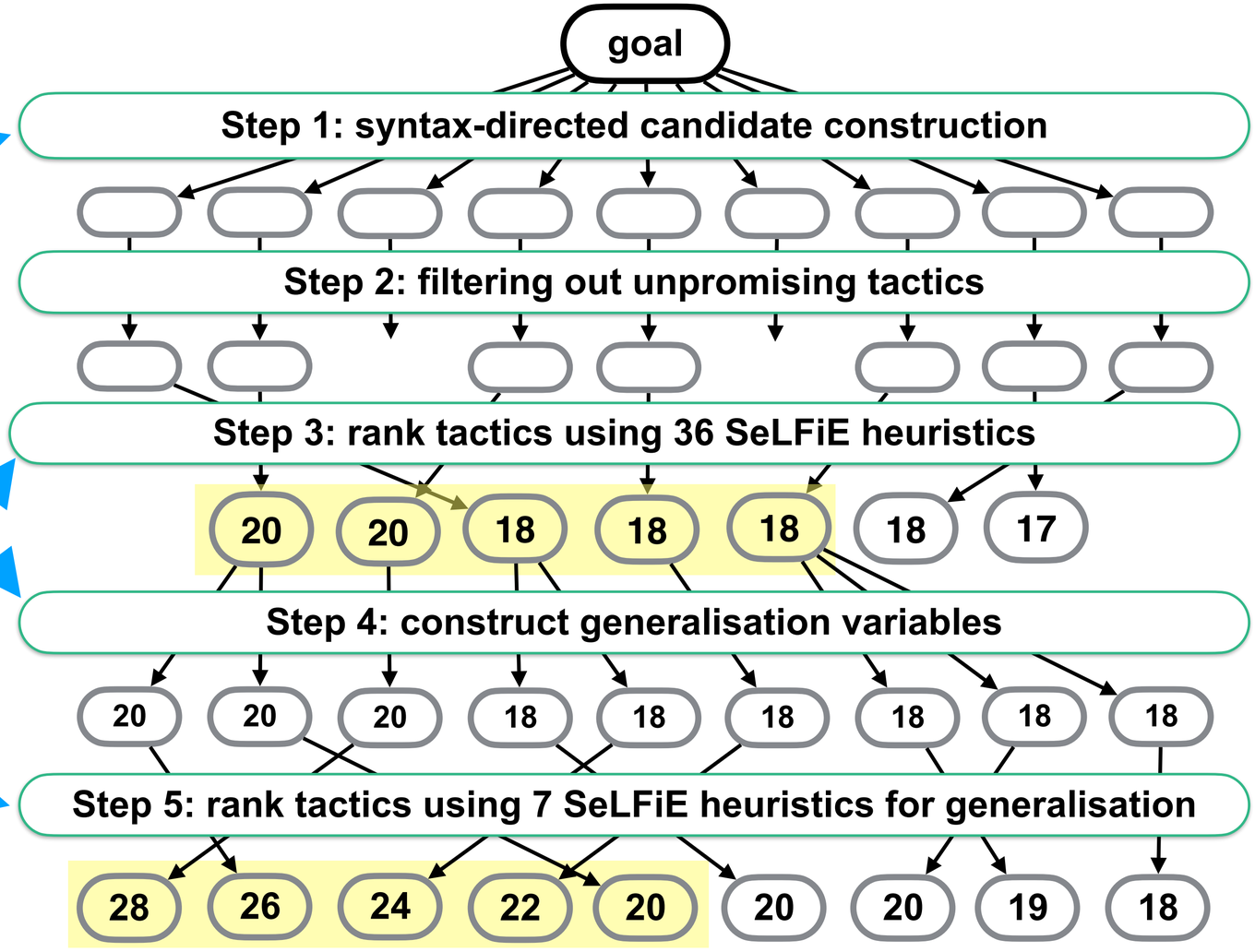

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20 20 19 18 18 18 17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step 3: rank tactics using 36 SeLFiE heuristics
20 20 18 18 18 18 17
Step 4: construct generalisation variables
20 20 20 18 18 18 18 18 18
Step 5: rank tactics using 7 SeLFiE heuristics for generalisation
28 26 24 22 20 20 20 19 18


smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20 20 19 18 18 18 17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step 3: rank tactics using 36 SeLFiE heuristics
20 20 18 18 18 18 17
Step 4: construct generalisation variables
20 20 20 18 18 18 18 18 18
Step 5: rank tactics using 7 SeLFiE heuristics for generalisation
28 26 24 22 20 20 20 19 18

smart_induct (based on LiFtEr)



sem_ind (based on SeLFiE)



Did you achieve better performance (accuracy & speed)?

Coincidence rates (accuracy)
based on 1095 inductive problems from the Archive of Formal Proofs

tool	top 1	top 3	top 5	top 10
`smart_induct`	20.1%	42.8%	48.5%	55.3%

Table 1: Overall coincidence rates within 5.0 seconds of timeout.

Coincidence rates (accuracy)
based on 1095 inductive problems from the Archive of Formal Proofs

tool	top 1	top 3	top 5	top 10
`smart_induct`	20.1%	42.8%	48.5%	55.3% 😱

Table 1: Overall coincidence rates within 5.0 seconds of timeout.

Coincidence rates (accuracy)
based on 1095 inductive problems from the Archive of Formal Proofs

tool	top 1	top 3	top 5	top 10
`sem_ind`	38.2%	59.3%	64.5%	72.7%
`smart_induct`	20.1%	42.8%	48.5%	55.3% 😱

Table 1: Overall coincidence rates within 5.0 seconds of timeout.

Coincidence rates (accuracy)
based on 1095 inductive problems from the Archive of Formal Proofs

tool	top 1	top 3	top 5	top 10
`sem_ind`	38.2%	59.3%	64.5%	72.7% 😄
`smart_induct`	20.1%	42.8%	48.5%	55.3% 😱

Table 1: Overall coincidence rates within 5.0 seconds of timeout.

Coincidence rates (accuracy)
based on 1095 inductive problems from the Archive of Formal Proofs

tool	top 1	top 3	top 5	top 10
sem_ind	38.2%	59.3%	64.5%	72.7% 😄
smart_induct	20.1%	42.8%	48.5%	55.3% 😱

Table 1: Overall coincidence rates within 5.0 seconds of timeout.

tool	top 1	top 3	top 5	top 10
smart_induct	0.0%	0.0%	0.0%	9.1%

Table 2: Coincidence rates for `Nearest_Neighbors.thy`.

Coincidence rates (accuracy)
based on 1095 inductive problems from the Archive of Formal Proofs

tool	top 1	top 3	top 5	top 10
`sem_ind`	38.2%	59.3%	64.5%	72.7% 😄
`smart_induct`	20.1%	42.8%	48.5%	55.3% 😱

Table 1: Overall coincidence rates within 5.0 seconds of timeout.

tool	top 1	top 3	top 5	top 10
`smart_induct`	0.0%	0.0%	0.0%	9.1% 😱

Table 2: Coincidence rates for `Nearest_Neighbors.thy`.

Coincidence rates (accuracy)
based on 1095 inductive problems from the Archive of Formal Proofs

tool	top 1	top 3	top 5	top 10
sem_ind	38.2%	59.3%	64.5%	72.7% 😄
smart_induct	20.1%	42.8%	48.5%	55.3% 😱

Table 1: Overall coincidence rates within 5.0 seconds of timeout.

tool	top 1	top 3	top 5	top 10
sem_ind	54.5%	63.6%	72.7%	72.7%
smart_induct	0.0%	0.0%	0.0%	9.1% 😱

Table 2: Coincidence rates for `Nearest_Neighbors.thy`.

Coincidence rates (accuracy)
based on 1095 inductive problems from the Archive of Formal Proofs

tool	top 1	top 3	top 5	top 10	
sem_ind	38.2%	59.3%	64.5%	72.7%	😄
smart_induct	20.1%	42.8%	48.5%	55.3%	😱

Table 1: Overall coincidence rates within 5.0 seconds of timeout.

tool	top 1	top 3	top 5	top 10	
sem_ind	54.5%	63.6%	72.7%	72.7%	😄
smart_induct	0.0%	0.0%	0.0%	9.1%	😱

Table 2: Coincidence rates for `Nearest_Neighbors.thy`.