Artificial Intelligence for Inductive Theorem Proving

Yutaka Nagashima, 25.02.2021

2013 ~ 2017

with Prof. Gerwin Klein

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

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

( )

2013 ~ 2017

with Prof. Gerwin Klein

pre-PhD

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

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

Isabelle HOL

(sel4)

Security. Performance. Proof.

2013 ~ 2017

with Prof. Gerwin Klein

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

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

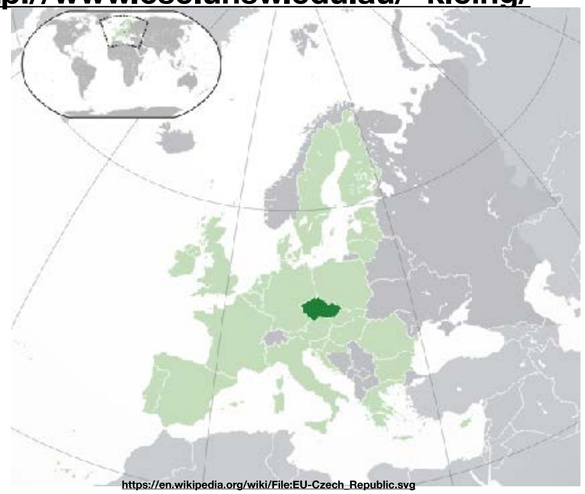
pre-PhD

PhD in
AI for theorem proving

2017 ~ 2020

with Prof. Cezary Kaliszyk
and Dr. Josef Urban

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



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

2013 ~ 2017

with Prof. Gerwin Klein

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

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

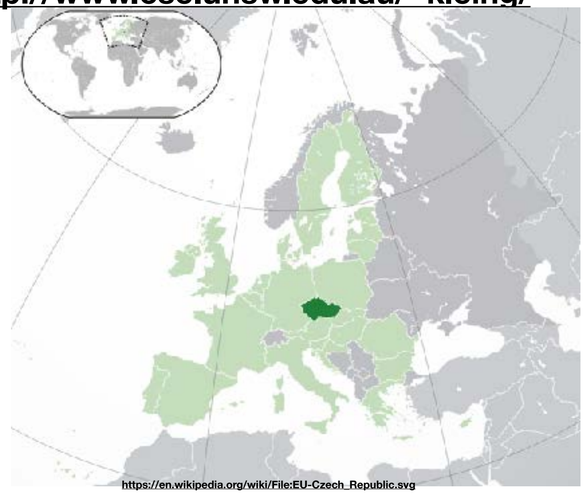
pre-PhD

PhD in AI for theorem proving

2017 ~ 2020

with Prof. Cezary Kaliszyk and Dr. Josef Urban

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



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

2020 ~

with Prof.Ilya Sergey

Seloloving CC BY-SA 3.0 (https://en.wikipedia.org/wiki/Singapore#/media/File:Singapore_on_the_globe_(Southeast_Asia_centered)_zoom.svg)

https://ilyasergey.net

2013 ~ 2017

with Prof. Gerwin Klein

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

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

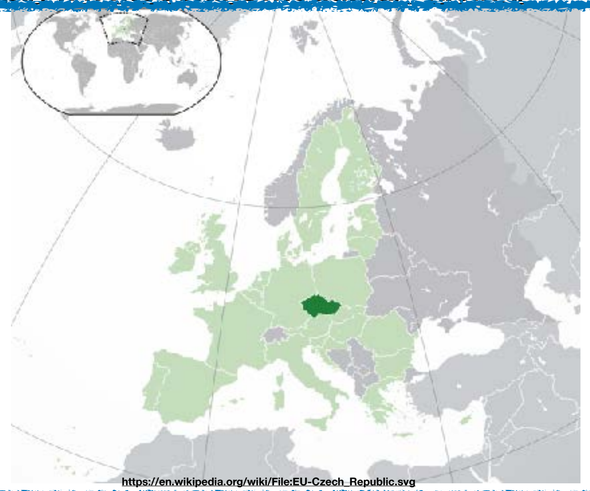
pre-PhD

PhD in
AI for theorem proving

2017 ~ 2020

with Prof. Cezary Kaliszyk
and Dr. Josef Urban

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



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

with focus on
proof by induction

2020 ~

with Prof.Ilya Sergey

Seloloving CC BY-SA 3.0 (https://en.wikipedia.org/wiki/Singapore#/media/File:Singapore_on_the_globe_(Southeast_Asia_centered)_zoom.svg)

https://ilyasergey.net

Why theorem proving?

- Why theorem proving?
- What is Isabelle/HOL?

- Why theorem proving?
- What is Isabelle/HOL?
- What are tactics?

- Why theorem proving?
- What is Isabelle/HOL?
- What are tactics?

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

```
5  primrec rev1::"'a list ⇒ 'a list" where
6    "rev1  []       = []"
7  | "rev1 (x # xs) = rev1 xs @ [x]"
8
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 theorem "rev2 xs ys = rev1 xs @ ys"
14   apply(induct xs ys rule: rev2.induct)
15   apply auto done
```

- Why theorem proving?
- What is Isabelle/HOL?
- What are tactics?

- Why theorem proving?
- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

- Why theorem proving?
- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

we are convinced that substantial progress in ITP will take time.

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

When applying AI for theorem proving, keep in mind…

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

When applying AI for theorem proving, keep in mind…

Algorithmic approaches work well, but not always.

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

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.

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

we are convinced that substantial progress in ITP will take time.

When applying AI for theorem proving,

Algorithmic approaches work well, but no

There are many good ways to prove one t

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

```
5  primrec rev1::"'a list ⇒ 'a list" where
6    "rev1  []       = []"
7  | "rev1 (x # xs) = rev1 xs @ [x]"
8
9  fun rev2::"'a list ⇒ 'a list ⇒ 'a list"
10   "rev2  []       ys = ys"
11 | "rev2 (x # xs) ys = rev2 xs (x # ys)"
12
13 theorem "rev2 xs ys = rev1 xs @ ys"
14   apply(induct xs ys rule: rev2.induct)
15   apply auto done
```

Proof state Auto up

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

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

When applying AI for theorem proving,

Algorithmic approaches work well, but no

There are many good ways to prove one t

```
| "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 up

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

Output Query Sledgehammer Symbols

15,13 (344/353)

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

Proof state Auto up

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

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

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.

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

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.

ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

- What is Isabelle/HOL?
- What are tactics?
- Why proof by induction?

Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

we are convinced that substantial progress in ITP will take time.

spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving.

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

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: Tactic recommendation (PaMpeR).

Path to automatic induction

Step 1: Tactic recommendation (PaMpeR).

Step 2: Tactic argument recommendation.

Path to automatic induction

Step 1: Tactic recommendation (PaMpeR).

Step 2: Tactic argument recommendation.

a: Guided backtracking search with tracing (PSL).

Path to automatic induction

Step 1: Tactic recommendation (PaMpeR).

Step 2: Tactic argument recommendation.

- a: Guided backtracking search with tracing (PSL).
- b: Logical Feature Extraction (LiFtEr).
- c: smart_induct using LiFtEr.
- d: Semantic-aware Logical Feature Extraction (SeLFiE).
- e: sem_ind using SeLFiE.
- f: conjecturing using abductive-reasoning.

Path to automatic induction

Step 1: Tactic recommendation (PaMpeR).

Step 2: Tactic argument recommendation.

- a: Guided backtracking search with tracing (PSL).
- b: Logical Feature Extraction (LiFtEr).
- c: smart_induct using LiFtEr.
- d: Semantic-aware Logical Feature Extraction (SeLFiE).
- e: sem_ind using SeLFiE.
- f: conjecturing using abductive-reasoning.

Step 3: Putting things together.

Proof Method Recommendation (PaMpeR)

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

```
5  primrec rev1::"'a list ⇒ 'a list" where
6    "rev1  []       = []"
7  | "rev1 (x # xs) = rev1 xs @ [x]"
8
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 theorem "rev2 xs ys = rev1 xs @ ys"
14
15   oops
```

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

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

Proof Method Recommendation (PaMpeR)

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

```
5  primrec rev1::"'a list ⇒ 'a list" where
6    "rev1  []       = []"
7  | "rev1 (x # xs) = rev1 xs @ [x]"
8
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 theorem "rev2 xs ys = rev1 xs @ ys"
14   which_method
```

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

Proof Method Recommendation (PaMpeR)

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

```
5  primrec rev1::"'a list ⇒ 'a list" where
6    "rev1  []       = []"
7  | "rev1 (x # xs) = rev1 xs @ [x]"
8
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 theorem "rev2 xs ys = rev1 xs @ ys"
14   which_method
```

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

```
Promising methods for this proof goal are:
  simp with expectation of 0.411909082795
  auto with expectation of 0.159332679097
  rule with expectation of 0.0874798467373
  induction with expectation of 0.0613706657985
  metis with expectation of 0.052603838226
  induct with expectation of 0.0510162518838
```

Proof Method Recommendation (PaMpeR)

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

```
5  primrec rev1::"'a list ⇒ 'a list" where
6    "rev1  []       = []"
7  | "rev1 (x # xs) = rev1 xs @ [x]"
8
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 theorem "rev2 xs ys = rev1 xs @ ys"
14   which_method
```

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

```
Promising methods for this proof goal are:
  simp with expectation of 0.411909082795
  auto with expectation of 0.159332679097
  rule with expectation of 0.0874798467373
  induction with expectation of 0.0613706657985
  metis with expectation of 0.052603838226
  induct with expectation of 0.0510162518838
```

Proof Method Recommendation (PaMpeR)

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

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

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.

```
which_method
```

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

```
Promising methods for this proof goal are:
  simp with expectation of 0.411909082795
  auto with expectation of 0.159332679097
  rule with expectation of 0.0874798467373
  induction with expectation of 0.0613706657985
  metis with expectation of 0.052603838226
  induct with expectation of 0.0510162518838
```

preparation phase

How does PaMpeR work?

recommendation phase

preparation phase

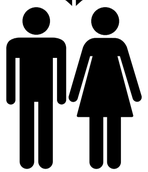
large proof corpora

AFP and standard library

How does PaMpeR work?

recommendation phase

proof state



proof engineer

preparation phase

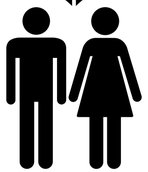
large proof corpora

AFP and standard library

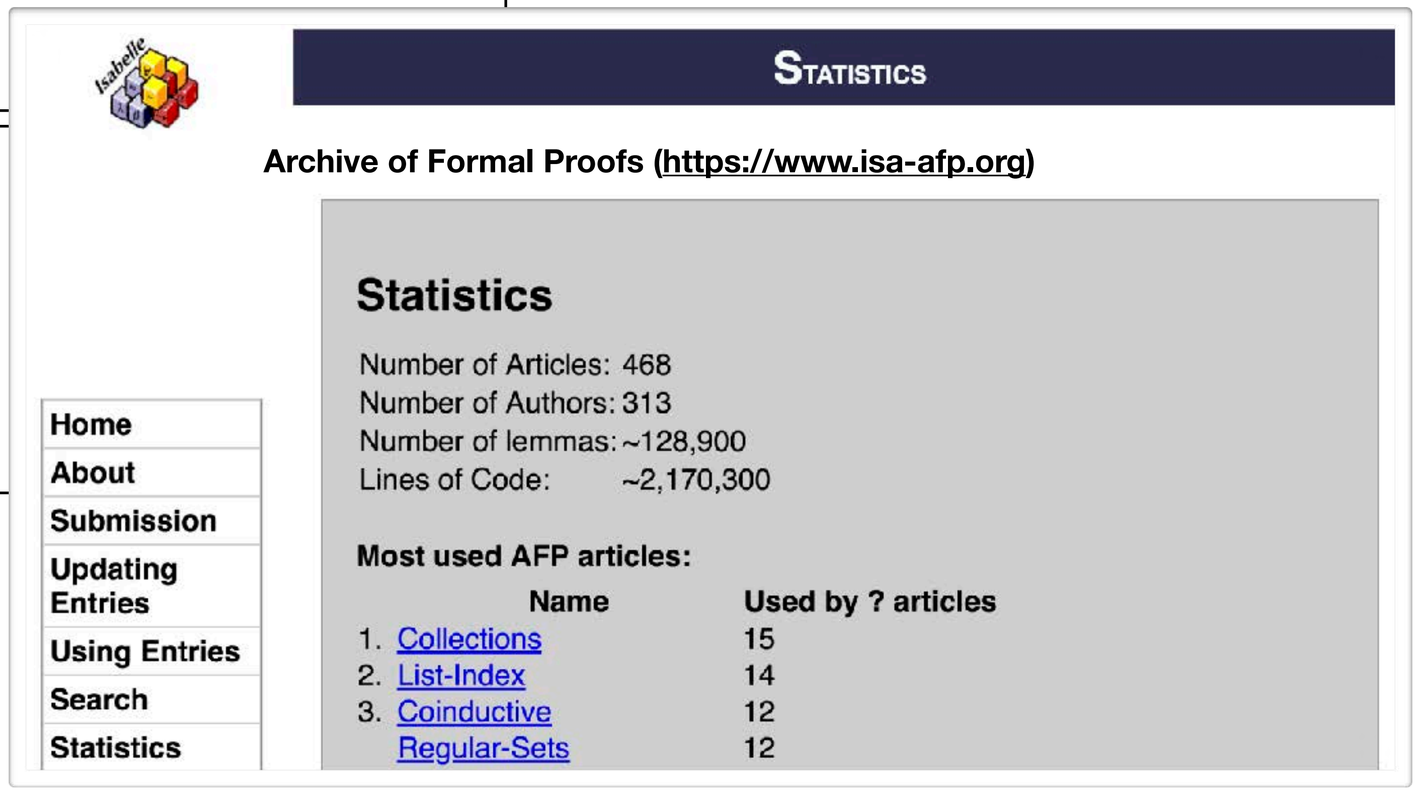
How does PaMpeR work?

recommendation phase

proof state



proof engineer



STATISTICS

Archive of Formal Proofs (https://www.isa-afp.org)

Home
About
Submission
Updating Entries
Using Entries
Search
Statistics

Statistics

Number of Articles: 468
Number of Authors: 313
Number of lemmas: ~128,900
Lines of Code: ~2,170,300

Most used AFP articles:

	Name	Used by ? articles
1.	Collections	15
2.	List-Index	14
3.	Coinductive	12
	Regular-Sets	12

preparation phase

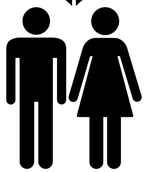
large proof corpora

AFP and standard library

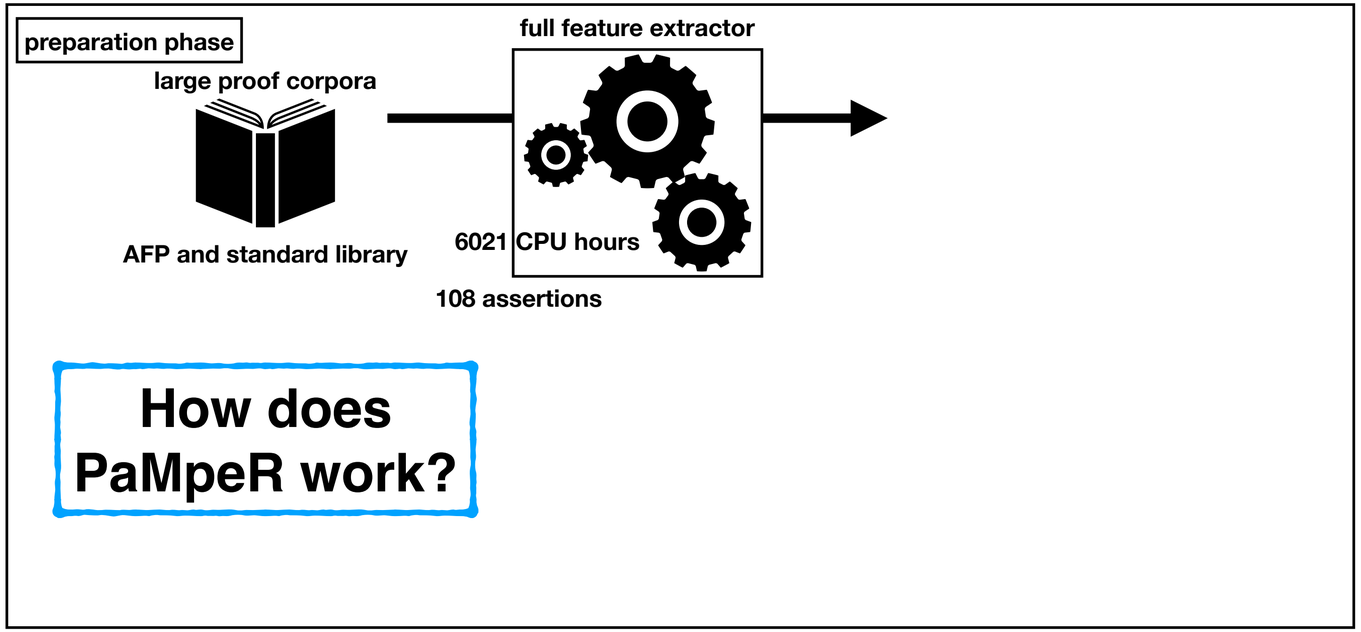
How does PaMpeR work?

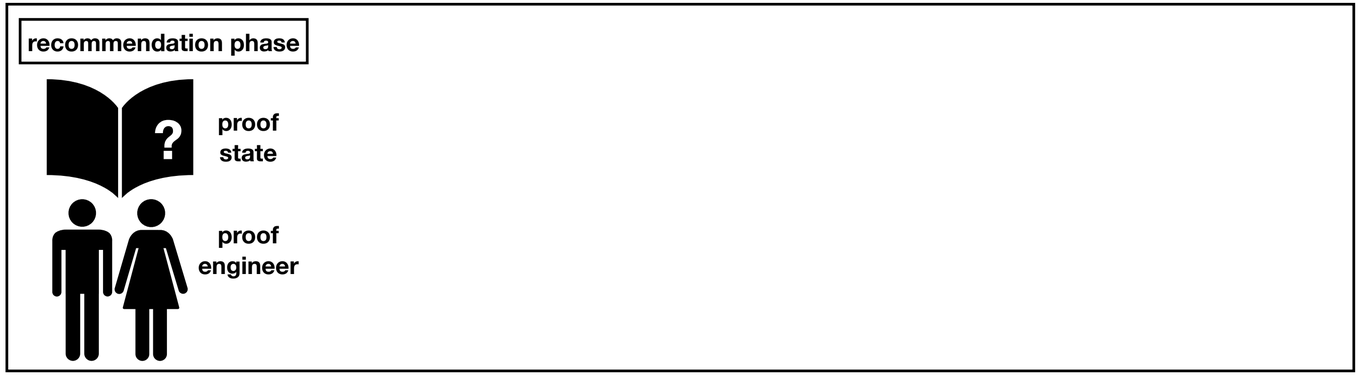
recommendation phase

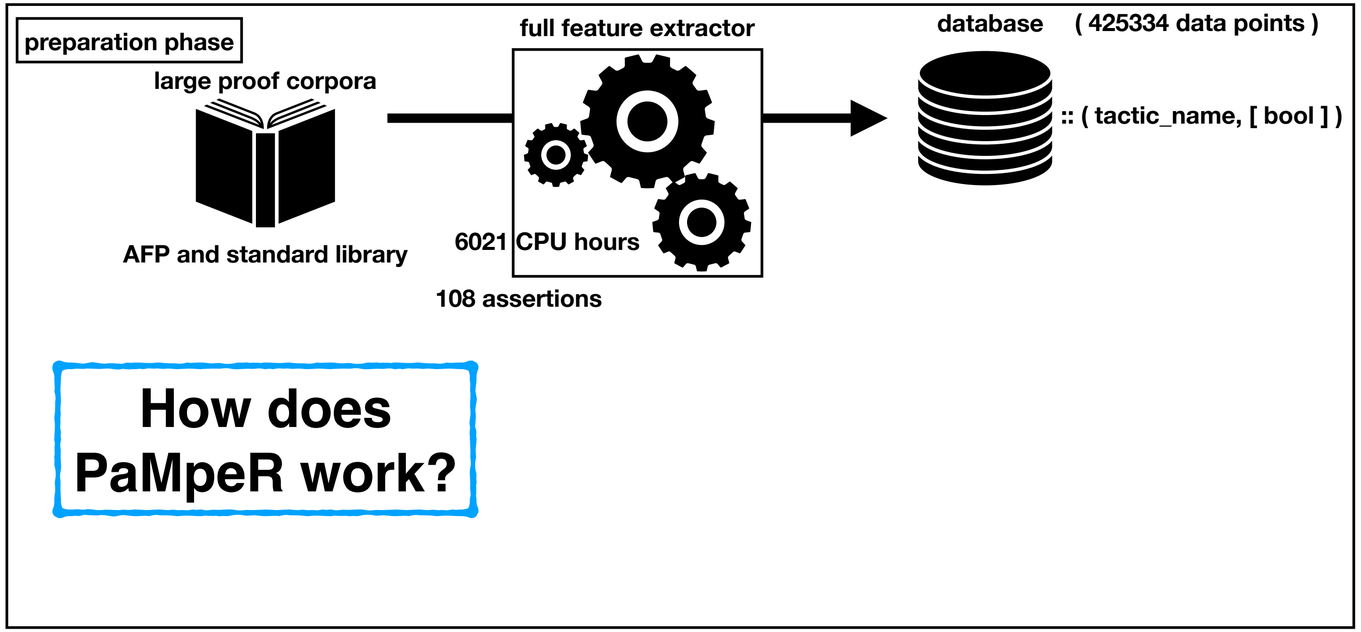
proof state

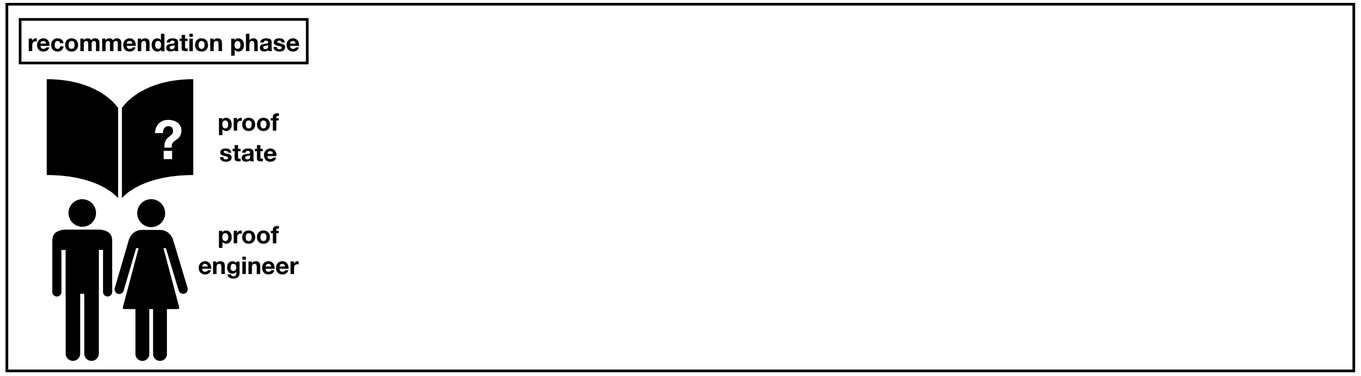


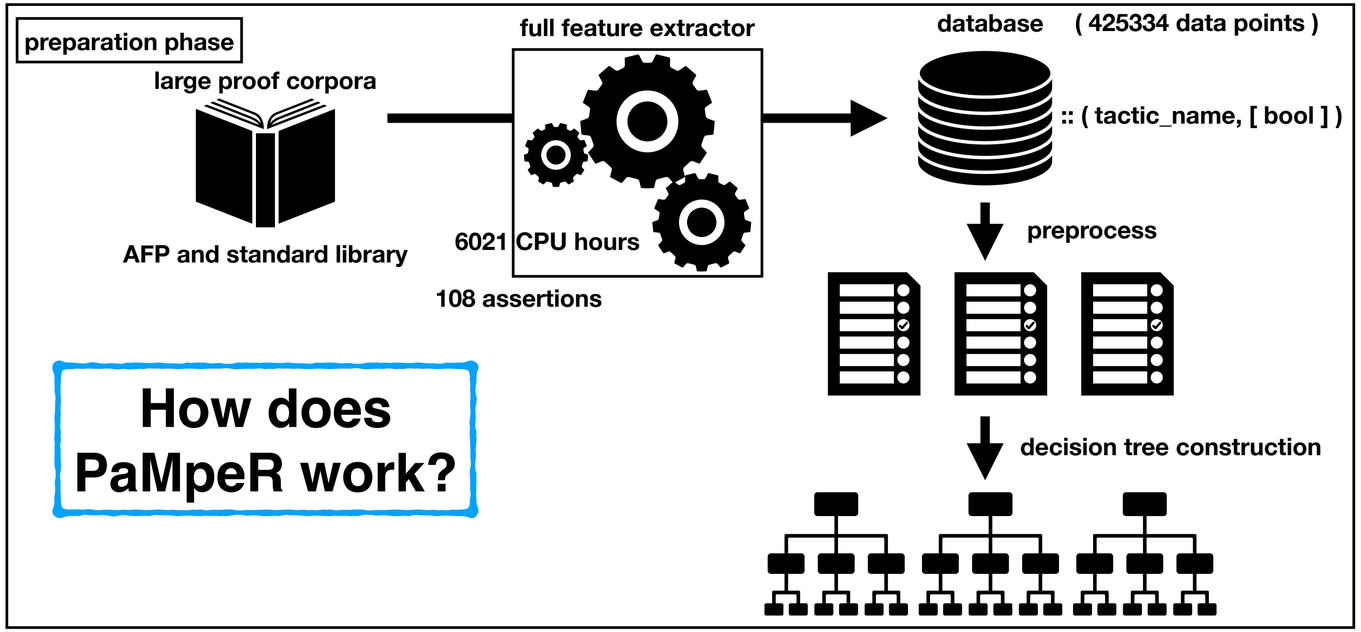
proof engineer

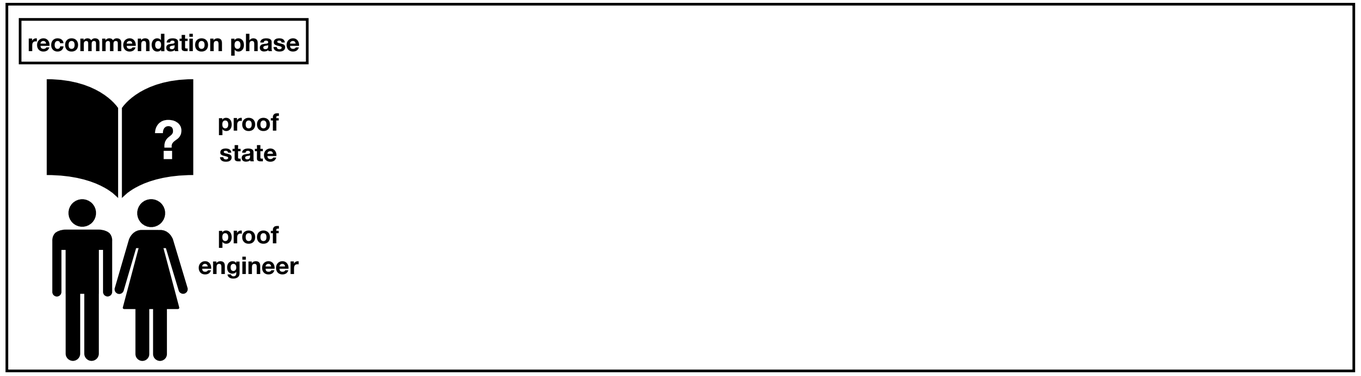

preparation phase
large proof corpora
AFP and standard library
full feature extractor
6021 CPU hours
108 assertions
How does PaMpeR work?

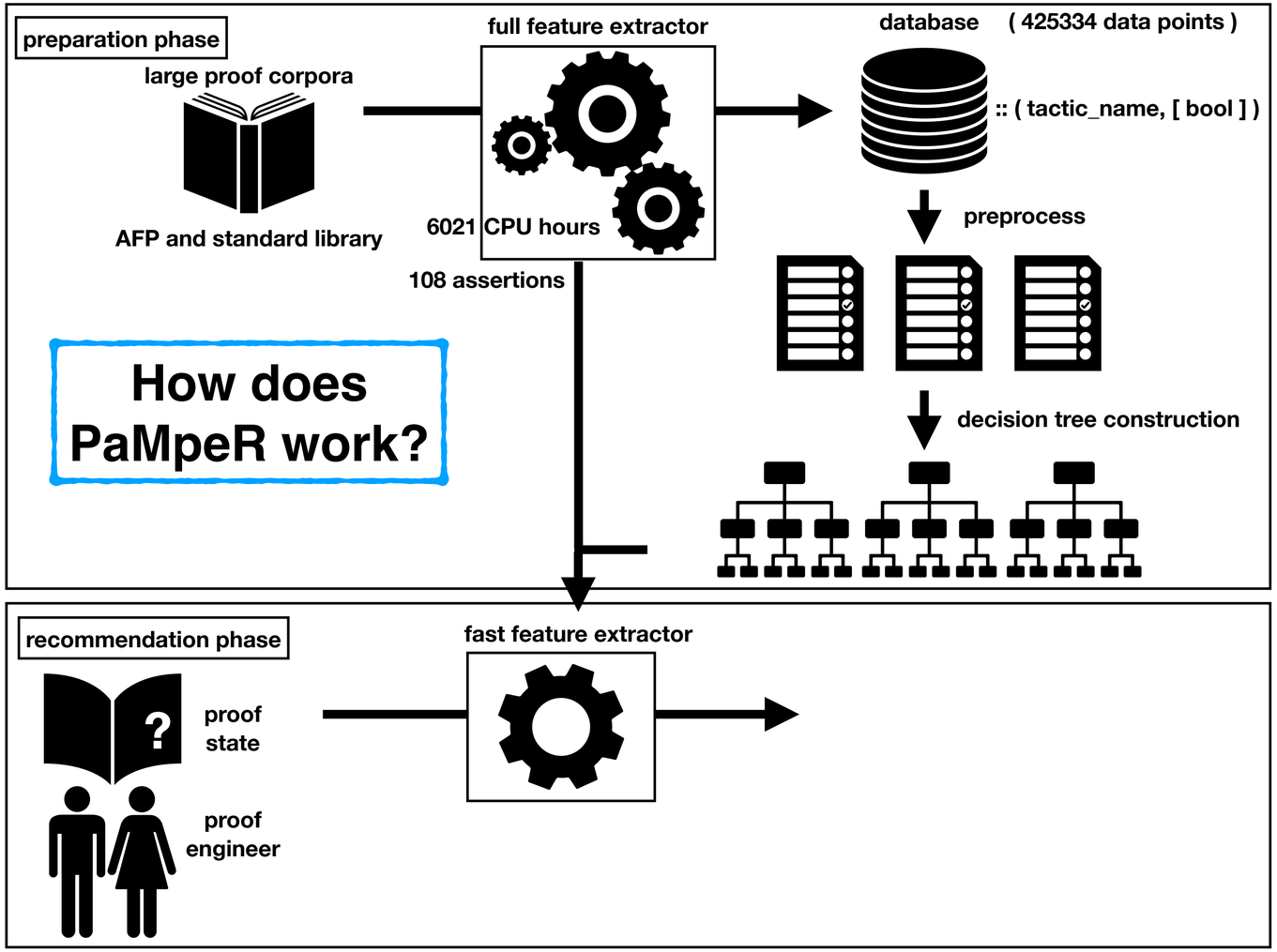

recommendation phase
?
proof state
proof engineer

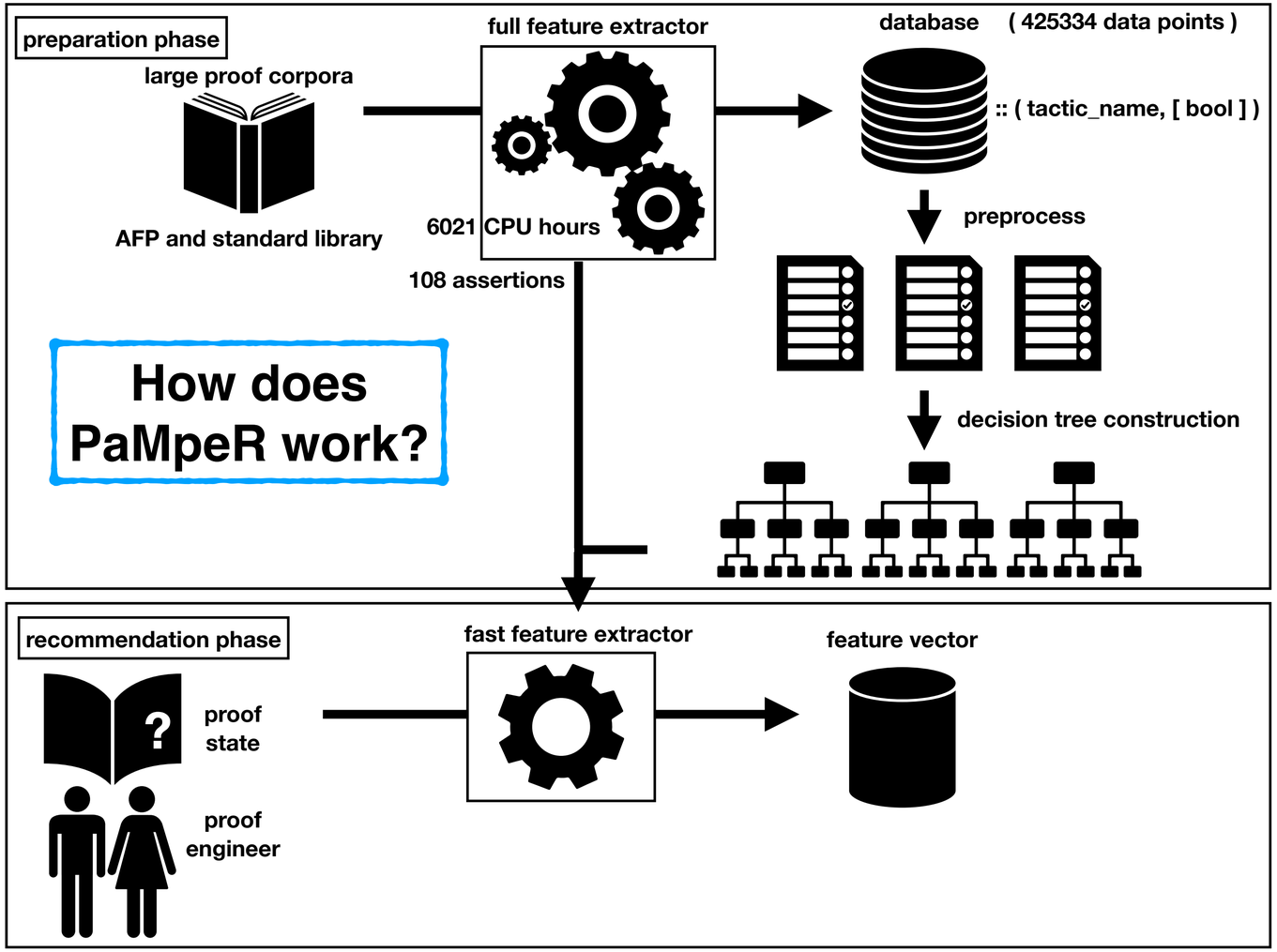

preparation phase
large proof corpora
AFP and standard library
full feature extractor
6021 CPU hours
108 assertions
database
(425334 data points)
:: (tactic_name, [bool])
How does PaMpeR work?

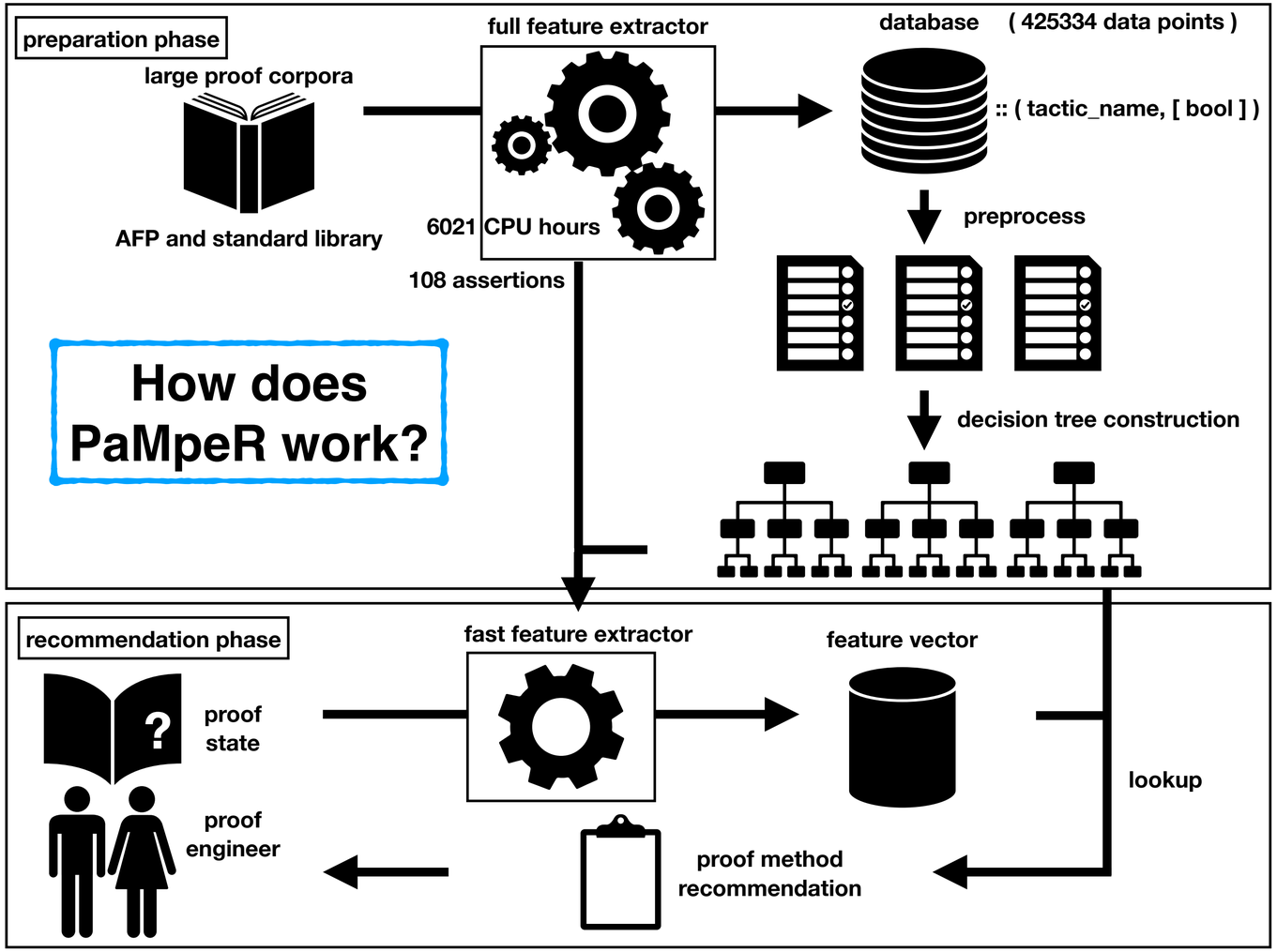

recommendation phase
?
proof state
proof engineer

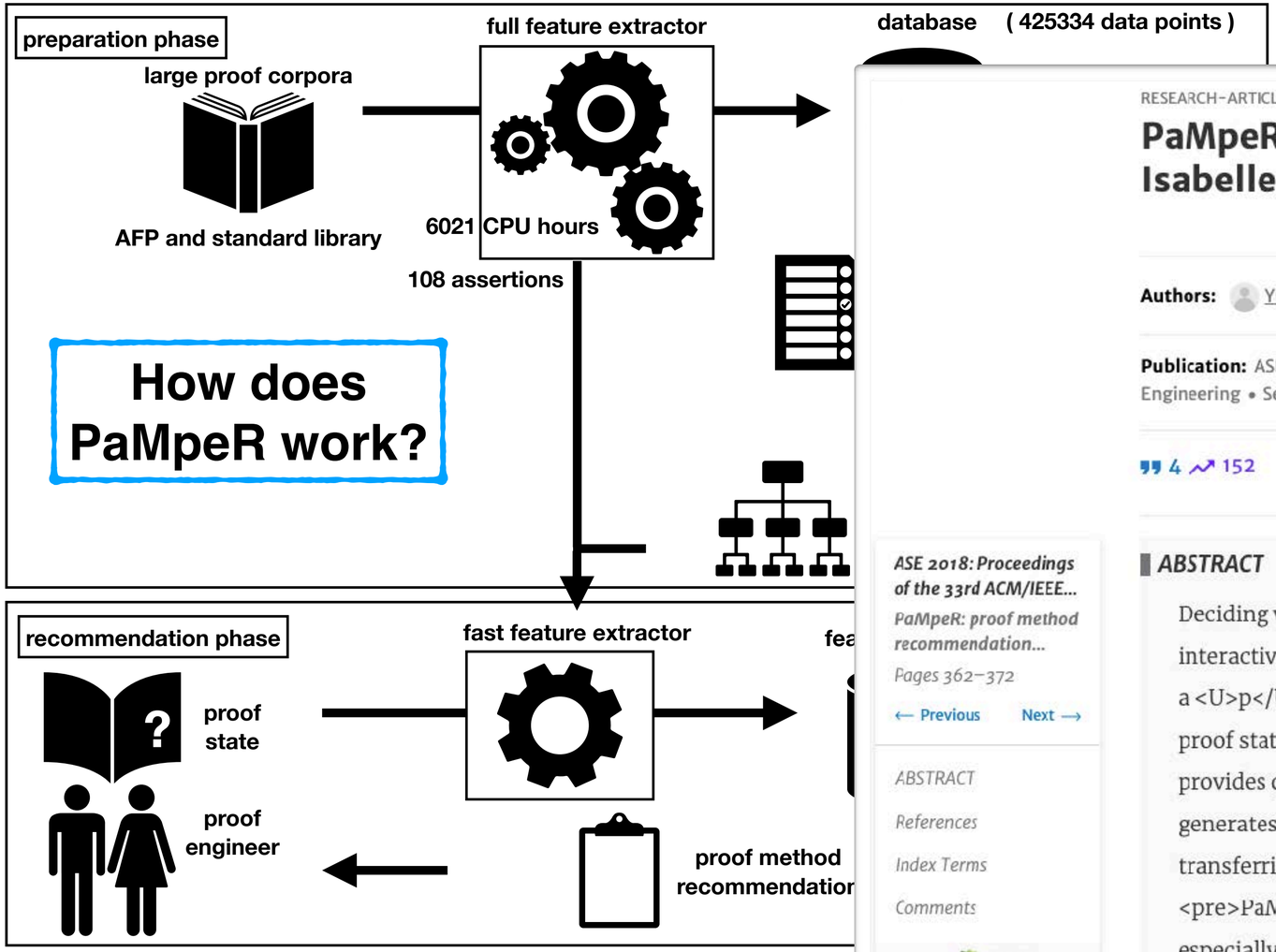

preparation phase
large proof corpora
AFP and standard library
full feature extractor
6021 CPU hours
108 assertions
database
(425334 data points)
:: (tactic_name, [bool])
preprocess
decision tree construction
How does PaMpeR work?


recommendation phase
proof state
proof engineer


preparation phase
large proof corpora
AFP and standard library
full feature extractor
6021 CPU hours
108 assertions
database
(425334 data points)
:: (tactic_name, [bool])
preprocess
decision tree construction
How does PaMpeR work?
recommendation phase
proof state
proof engineer
fast feature extractor


preparation phase
large proof corpora
AFP and standard library
full feature extractor
6021 CPU hours
108 assertions
database
(425334 data points)
:: (tactic_name, [bool])
preprocess
decision tree construction
How does PaMpeR work?
recommendation phase
proof state
proof engineer
fast feature extractor
feature vector


preparation phase
large proof corpora
AFP and standard library
full feature extractor
6021 CPU hours
108 assertions
database
(425334 data points)
:: (tactic_name, [bool])
preprocess
decision tree construction
How does PaMpeR work?
recommendation phase
proof state
proof engineer
fast feature extractor
feature vector
lookup
proof method recommendation



RESEARCH-ARTICLE

PaMpeR: proof method recommendation system for Isabelle/HOL

Authors: Yutaka Nagashima, Yilun He Authors Info & Affiliations

Publication: ASE 2018: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering • September 2018 • Pages 362–372 • https://doi.org/10.1145/3238147.3238210

4 152 Get Access

ASE 2018: Proceedings of the 33rd ACM/IEEE...
PaMpeR: proof method recommendation...
Pages 362–372
← Previous Next →
ABSTRACT
References
Index Terms
Comments

ACM DIGITAL LIBRARY

ABSTRACT

Deciding which sub-tool to use for a given proof state requires expertise specific to each interactive theorem prover (ITP). To mitigate this problem, we present <pre>PaMpeR</pre>, a <U>p</U>roof <U>m</U>ethod <U>r</U>ecommendation system for Isabelle/HOL. Given a proof state, <pre>PaMpeR</pre> recommends proof methods to discharge the proof goal and provides qualitative explanations as to why it suggests these methods. <pre>PaMpeR</pre> generates these recommendations based on existing hand-written proof corpora, thus transferring experienced users' expertise to new users. Our evaluation shows that <pre>PaMpeR</pre> correctly predicts experienced users' proof methods invocation especially when it comes to special purpose proof methods.

References

1. 2018. PSL with PGT: CICM2018 for Isabelle2017. (2018). https://github. com/data61/PSL/releases/tag/v0.1.1 To use PaMpeR, one first needs to install Isabelle/HOL, which is distributed at https://isabelle.in.tum.de/. PaMpeR: Proof Method Recommendation System for Isabelle/HOL ASE '18, September 3–7, 2018, Montpellier, France

Proof Method Recommendation (PaMpeR)

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

```
5  primrec rev1::"'a list ⇒ 'a list" where
6    "rev1  []       = []"
7  | "rev1 (x # xs) = rev1 xs @ [x]"
8
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 theorem "rev2 xs ys = rev1 xs @ ys"
14   which_method
```

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

```
Promising methods for this proof goal are:
  simp with expectation of 0.411909082795
  auto with expectation of 0.159332679097
  rule with expectation of 0.0874798467373
  induction with expectation of 0.0613706657985
  metis with expectation of 0.052603838226
  induct with expectation of 0.0510162518838
```

Proof Method Recommendation (PaMpeR)

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"
  which_method
```

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

```
Promising methods for this proof goal are:
  simp with expectation of 0.411909082795
  auto with expectation of 0.159332679097
  rule with expectation of 0.087479846737
  induction with expectation of 0.061370665798
  metis with expectation of 0.052603838226
  induct with expectation of 0.051016251883
```

Arguments of tactics?

Path to automatic induction

Path to automatic induction

Step 1: Tactic recommendation.

Path to automatic induction

Step 1: Tactic recommendation.

I know I should apply induction.
But I want to know how I should apply induction.

Path to automatic induction

Step 1: Tactic recommendation.

Step 2: Tactic argument recommendation.

I know I should apply induction.
But I want to know how I should apply induction.

Path to automatic induction

Step 1: Tactic recommendation.

Step 2: Tactic argument recommendation.

- a: Guided backtracking search with tracing (PSL).
- b: Logical Feature Extraction (LiFtEr).
- c: smart_induct using LiFtEr.
- d: Semantic-aware Logical Feature Extraction (SeLFiE).
- e: sem_ind using SeLFiE.
- f: conjecturing using abductive-reasoning.

Step 3: Putting things together.

I know I should apply induction.
But I want to know how I should apply induction.

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/)

```
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
```

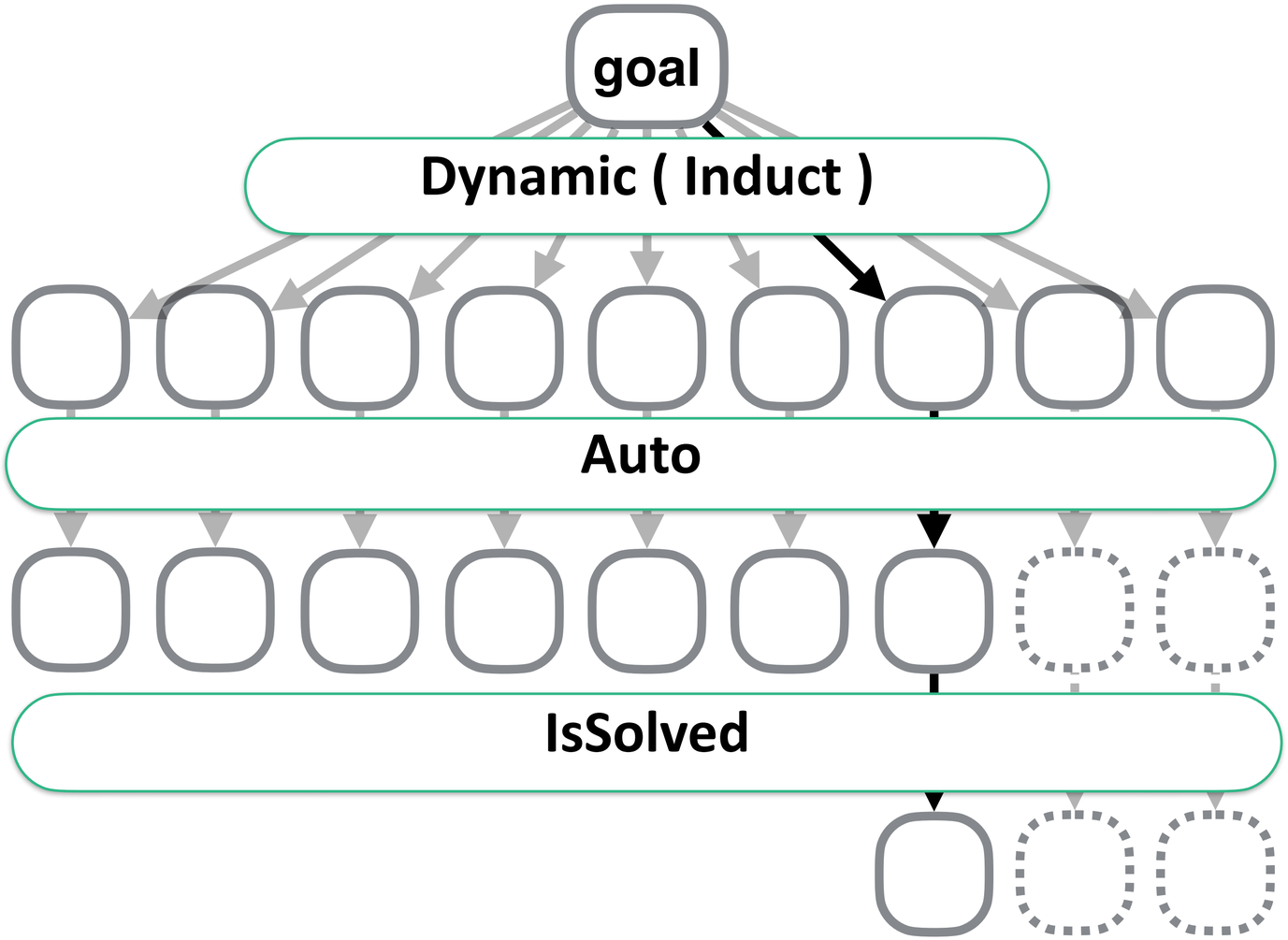
Proof state | Auto update | Update | Search: | 100%

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

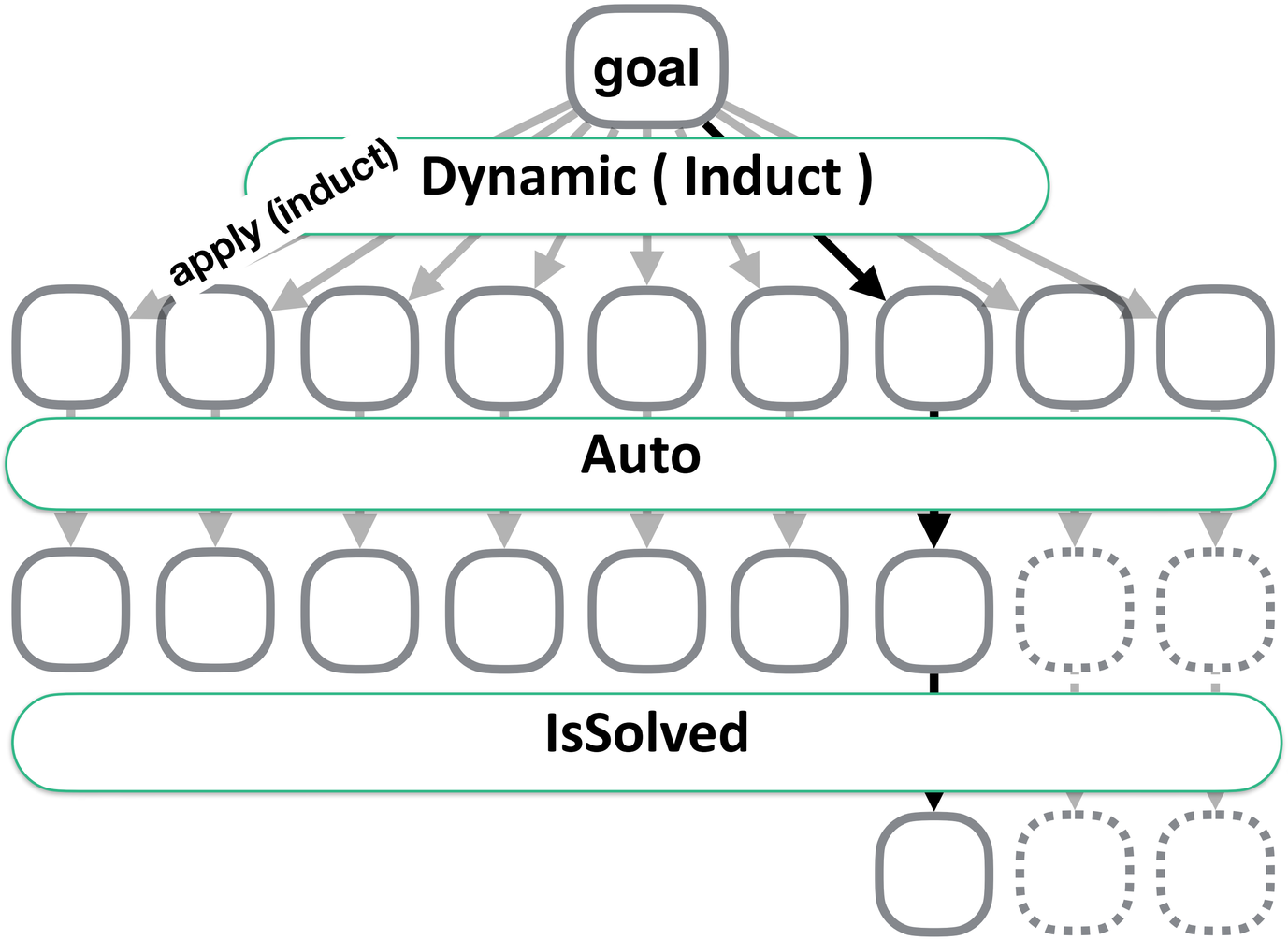
Output | Query | Sledgehammer | Symbols

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

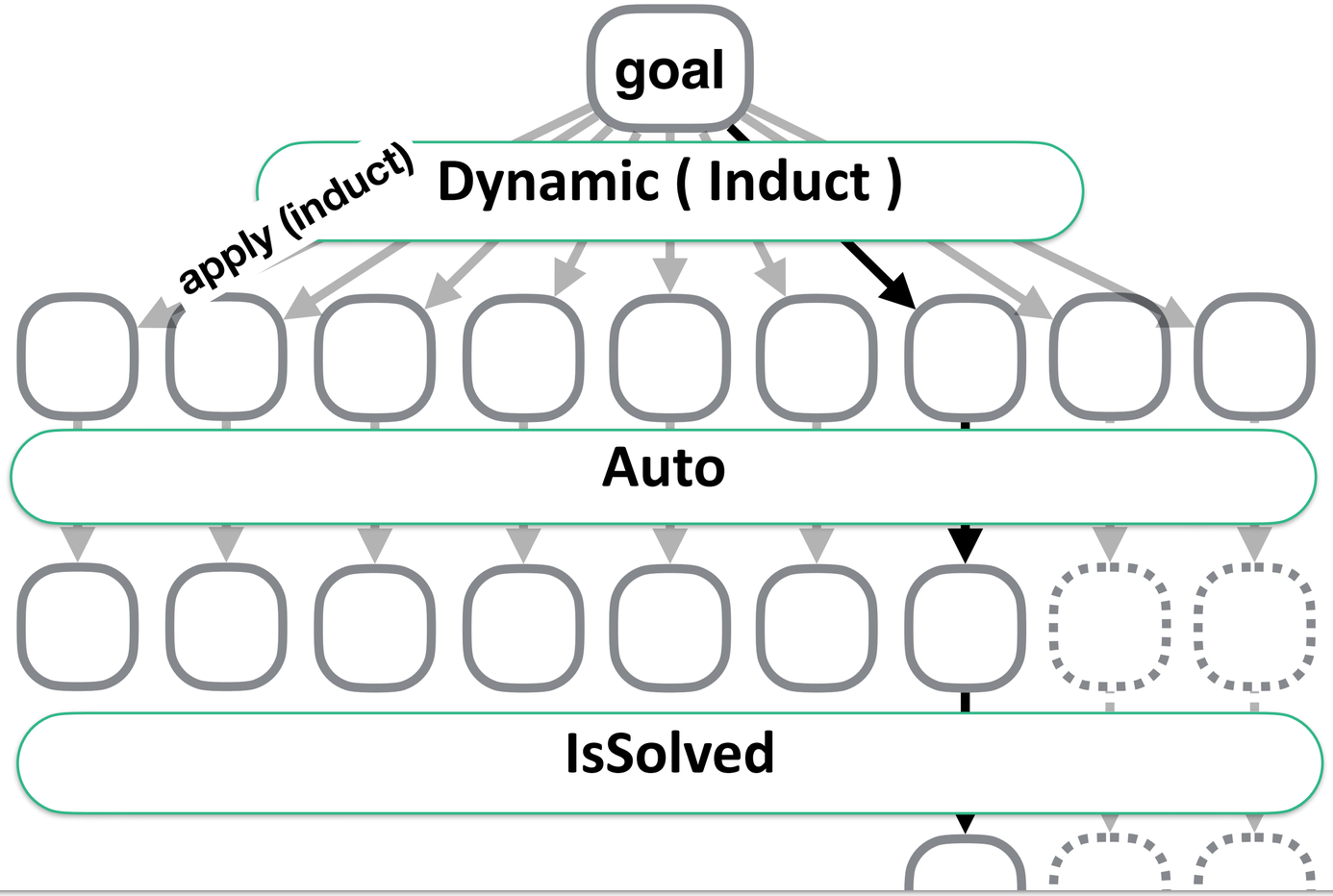
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

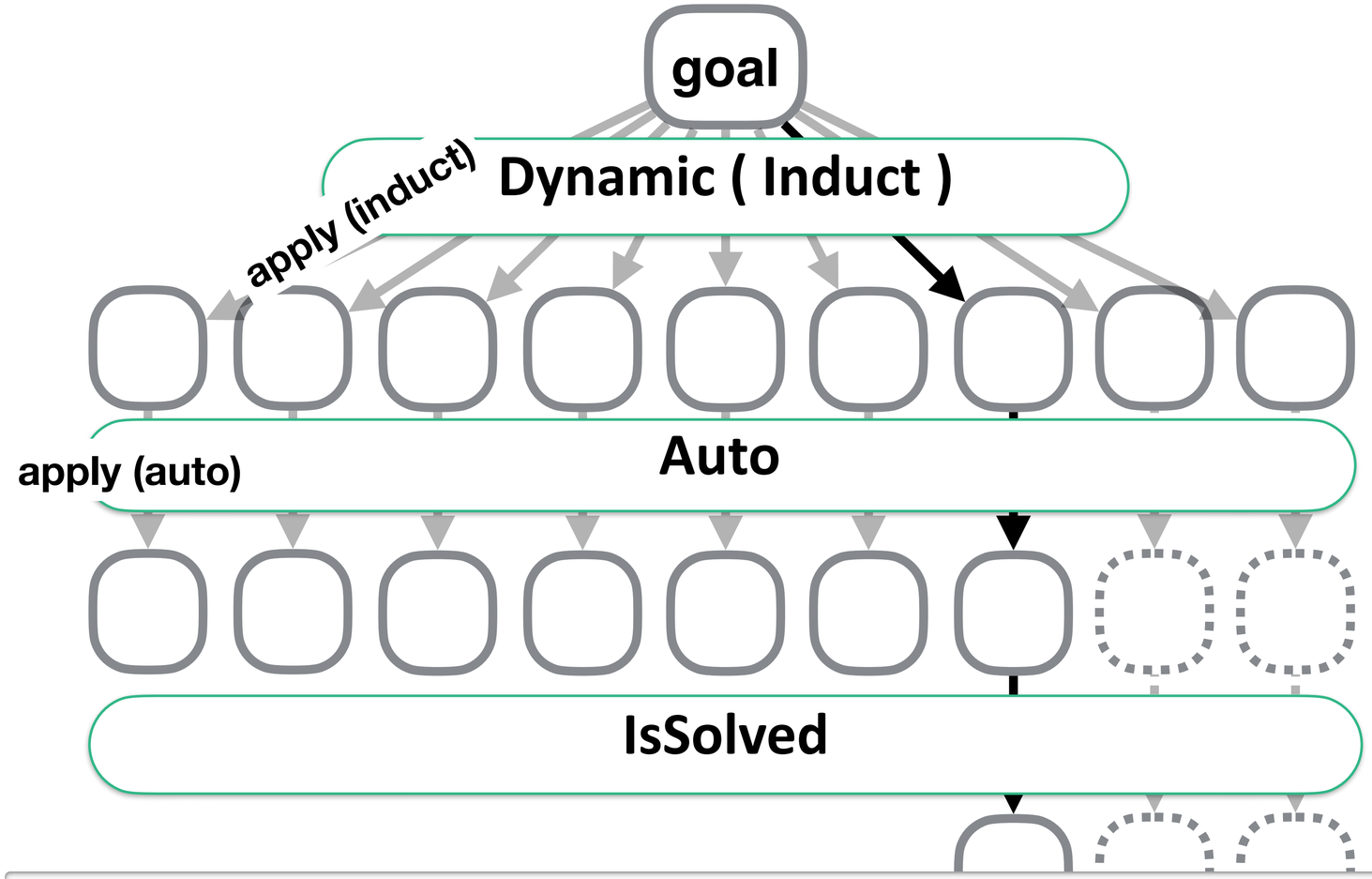


```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



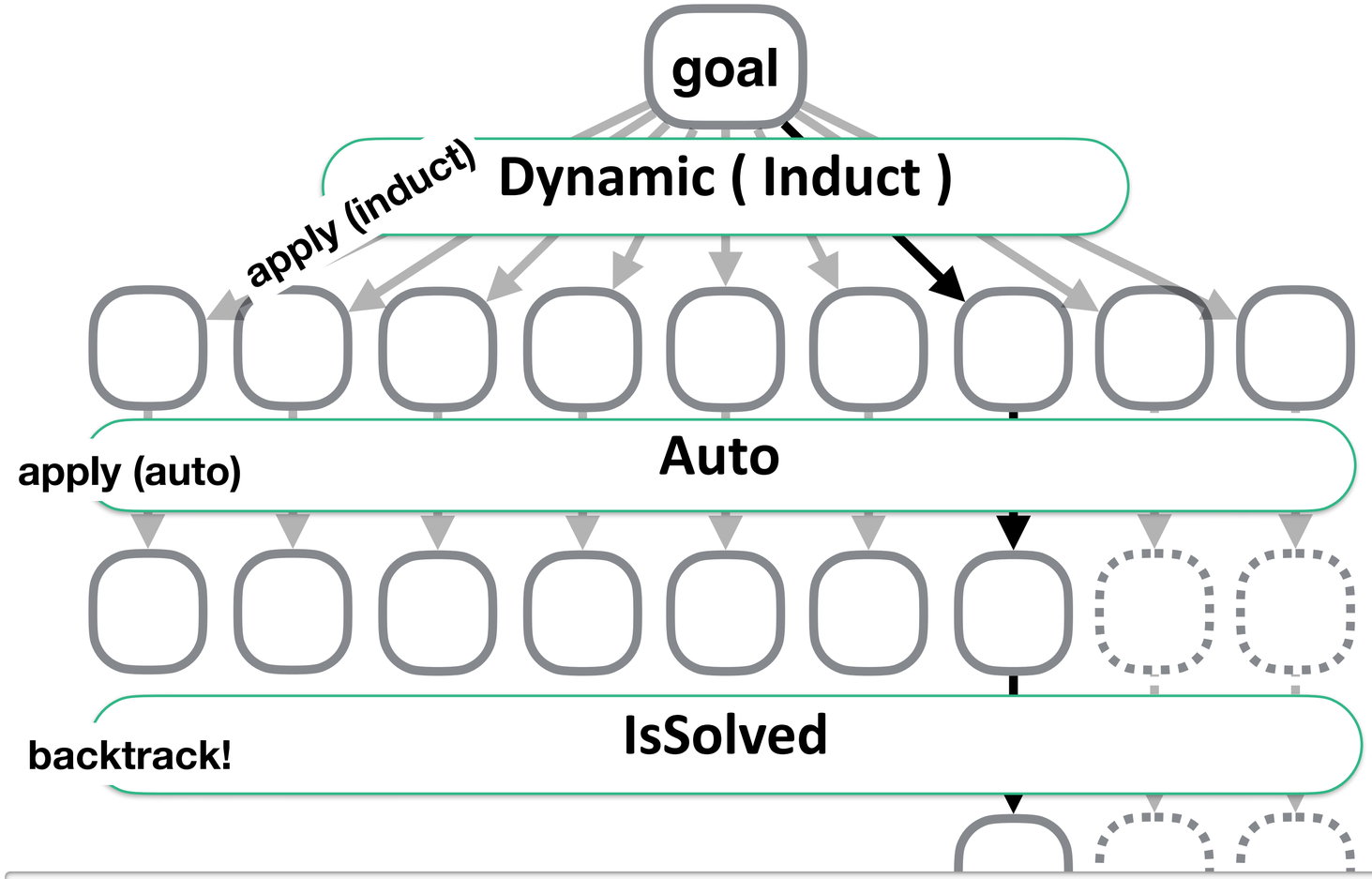

```
goal (1 subgoal):
 1. ⋀y. y ∈ {F. is_filter F} ⟹ rev2 xs ys = rev1 xs @ ys
```

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



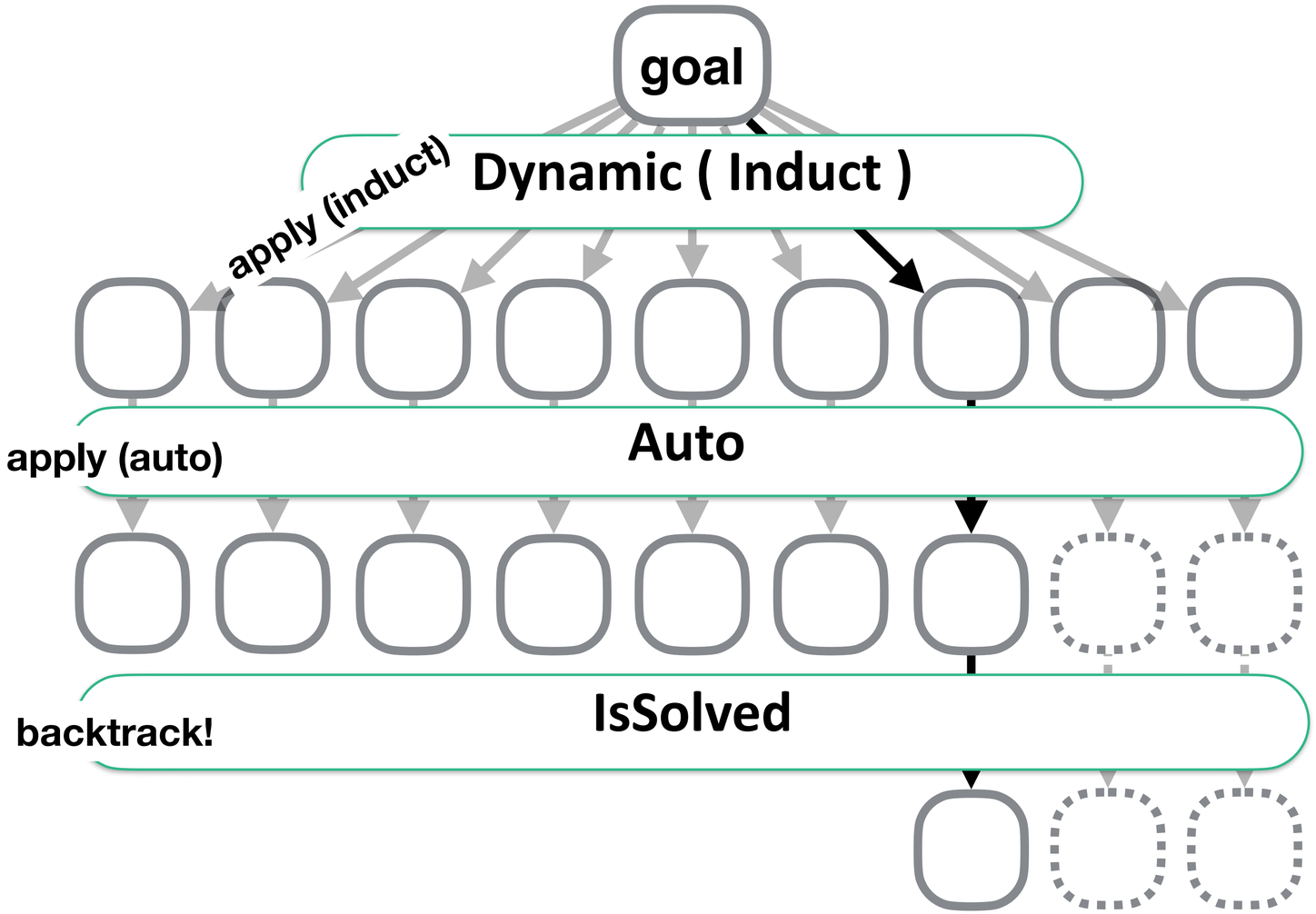

```
goal (1 subgoal):
 1. ⋀y. is_filter y ⟹ rev2 xs ys = rev1 xs @ ys
```

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

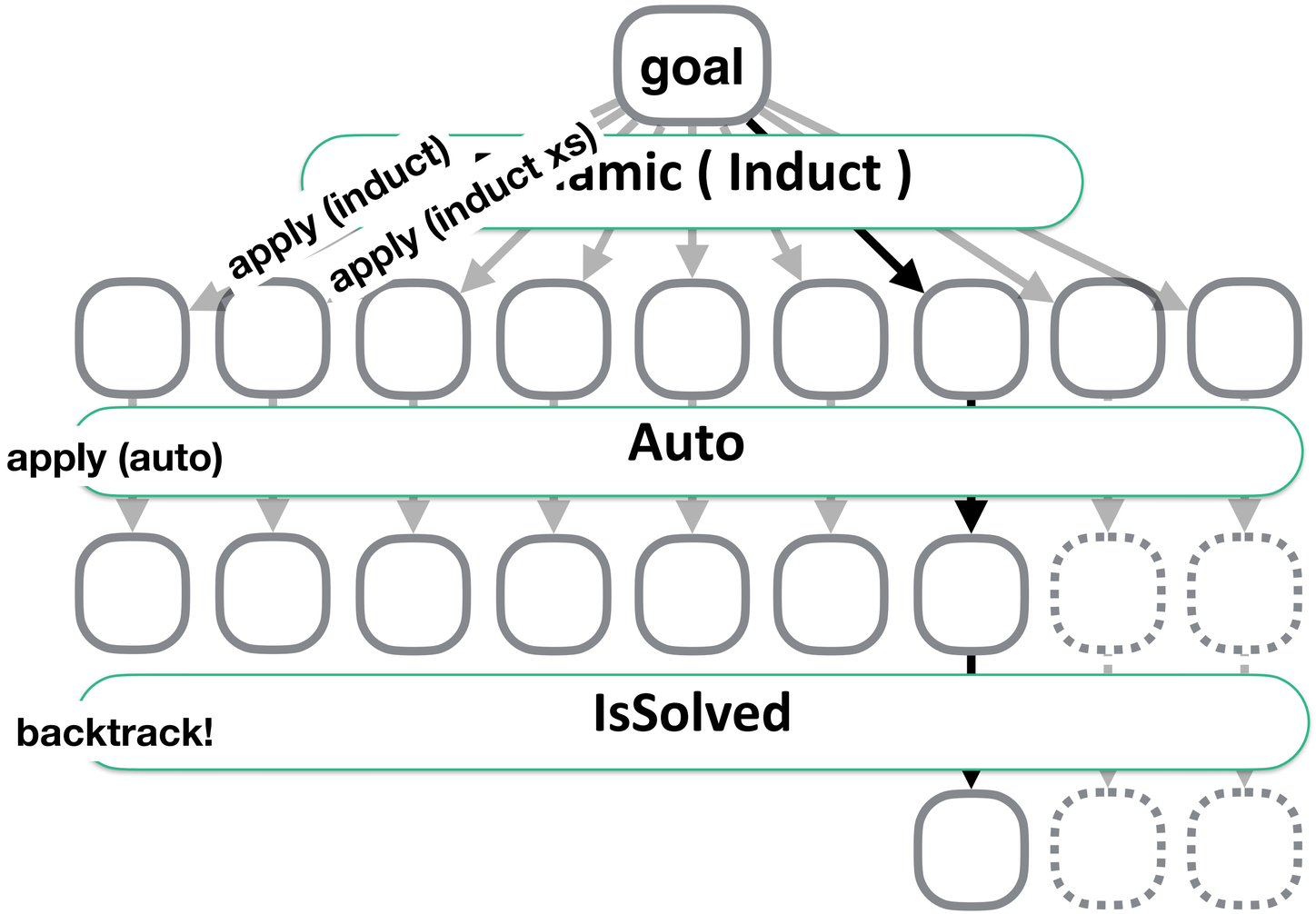



```
goal (1 subgoal):
 1. ⋀y. is_filter y ⟹ rev2 xs ys = rev1 xs @ ys
```

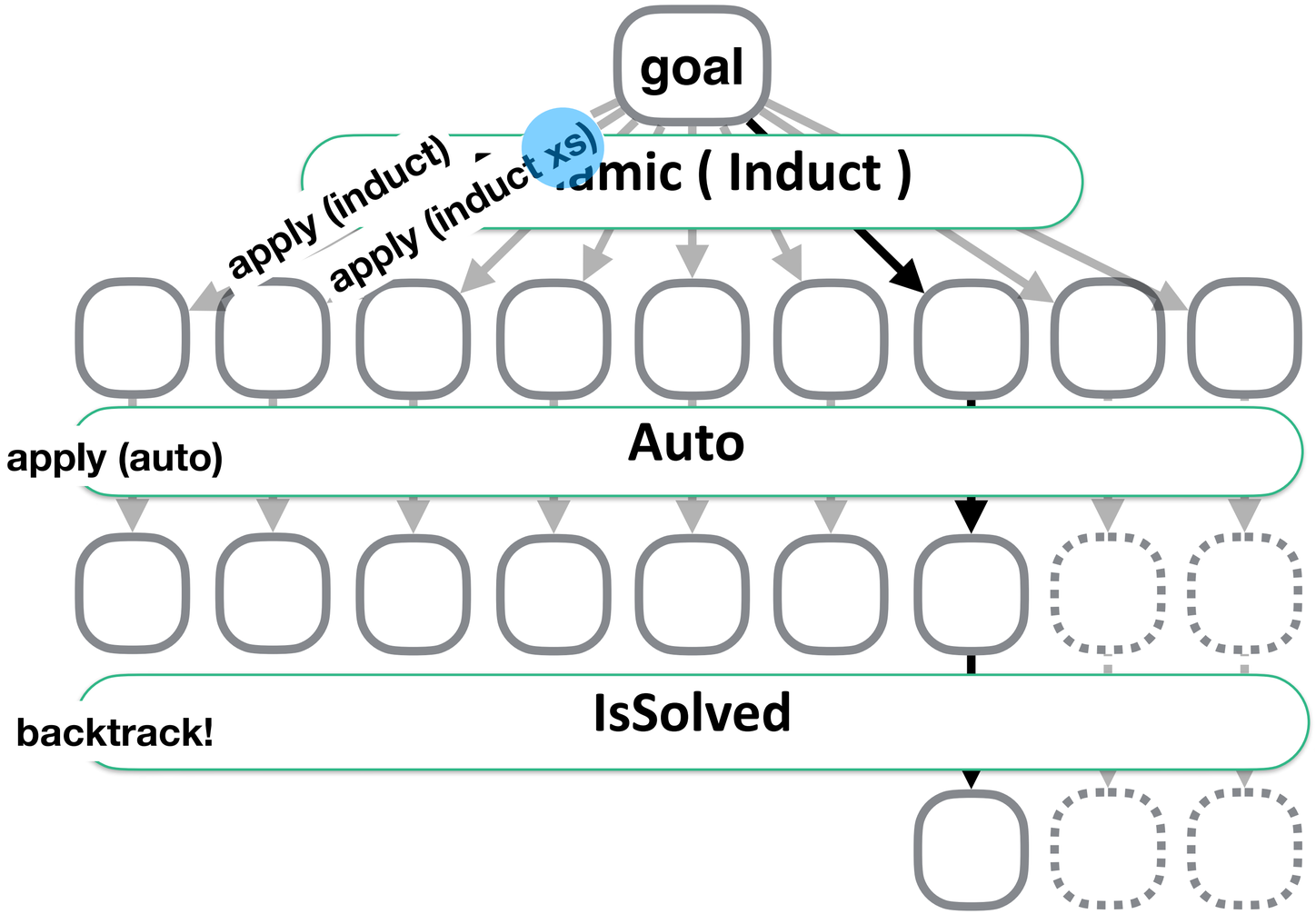
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



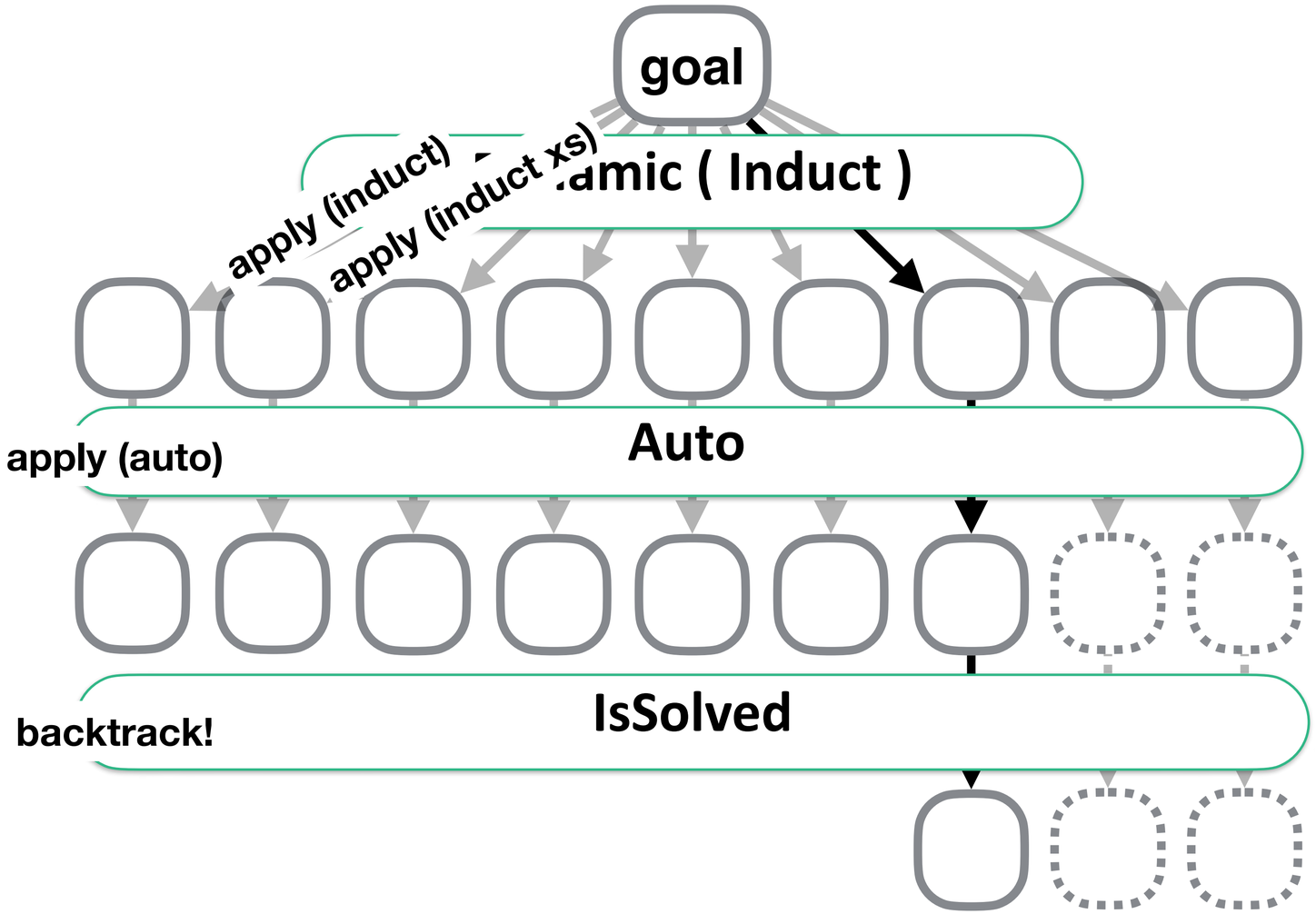
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



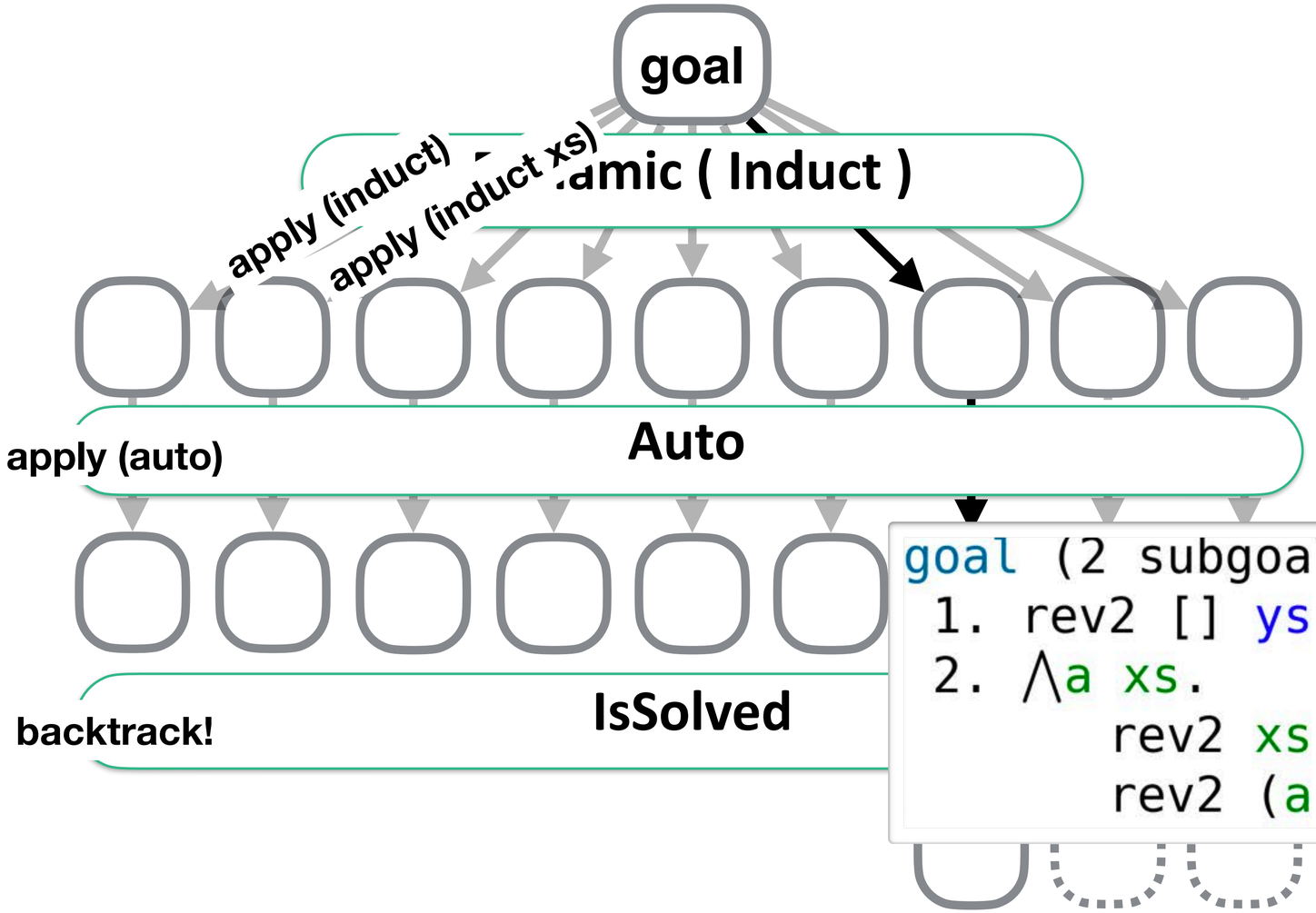
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

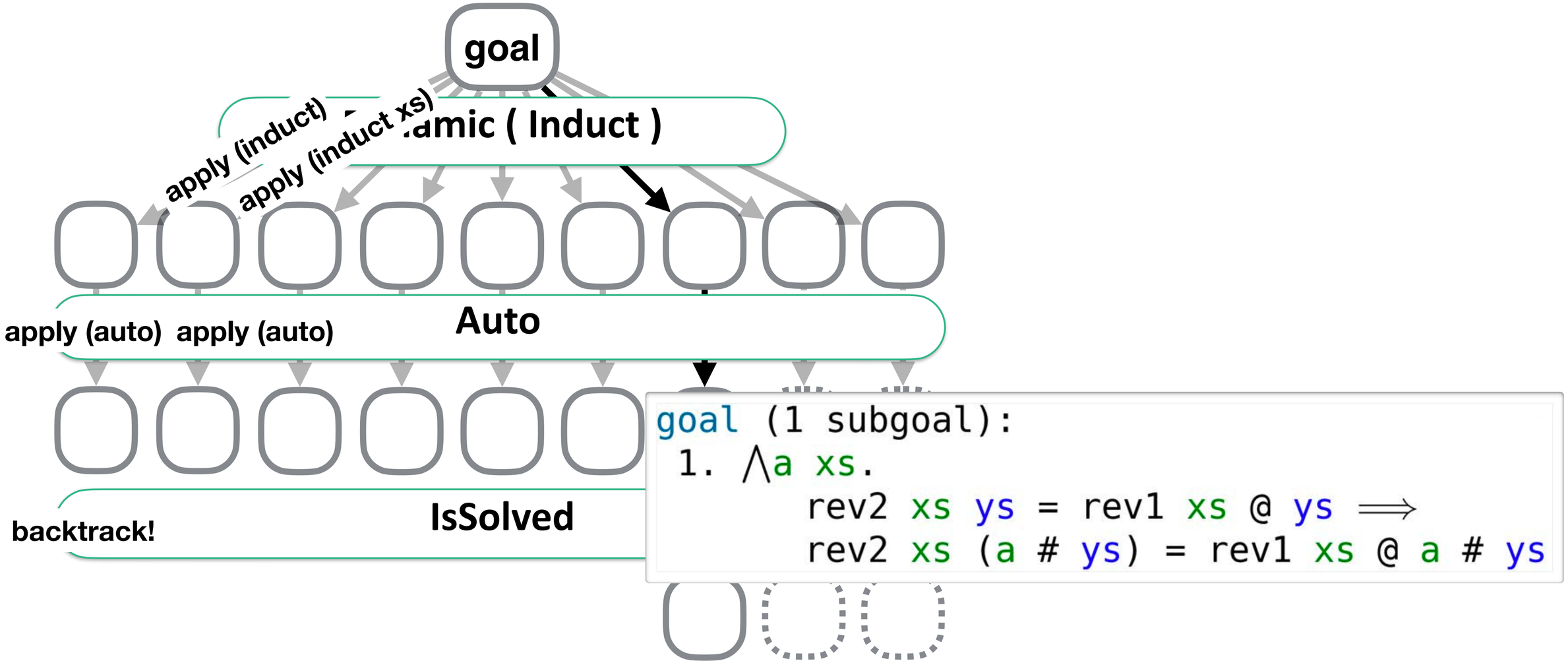


```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```

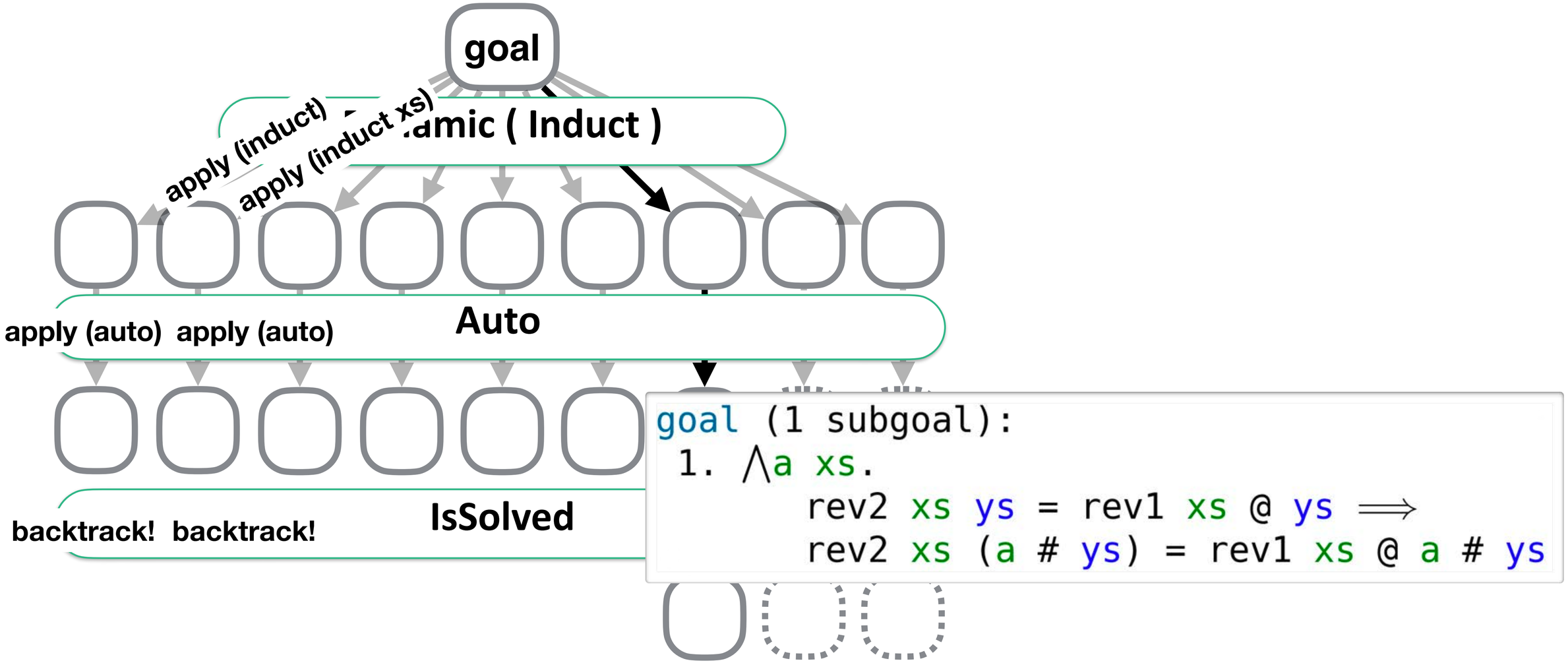



```
goal (2 subgoals):
 1. rev2 [] ys = rev1 [] @ ys
 2. ⋀a xs.
       rev2 xs ys = rev1 xs @ ys ⟹
       rev2 (a # xs) ys = rev1 (a # xs) @ ys
```

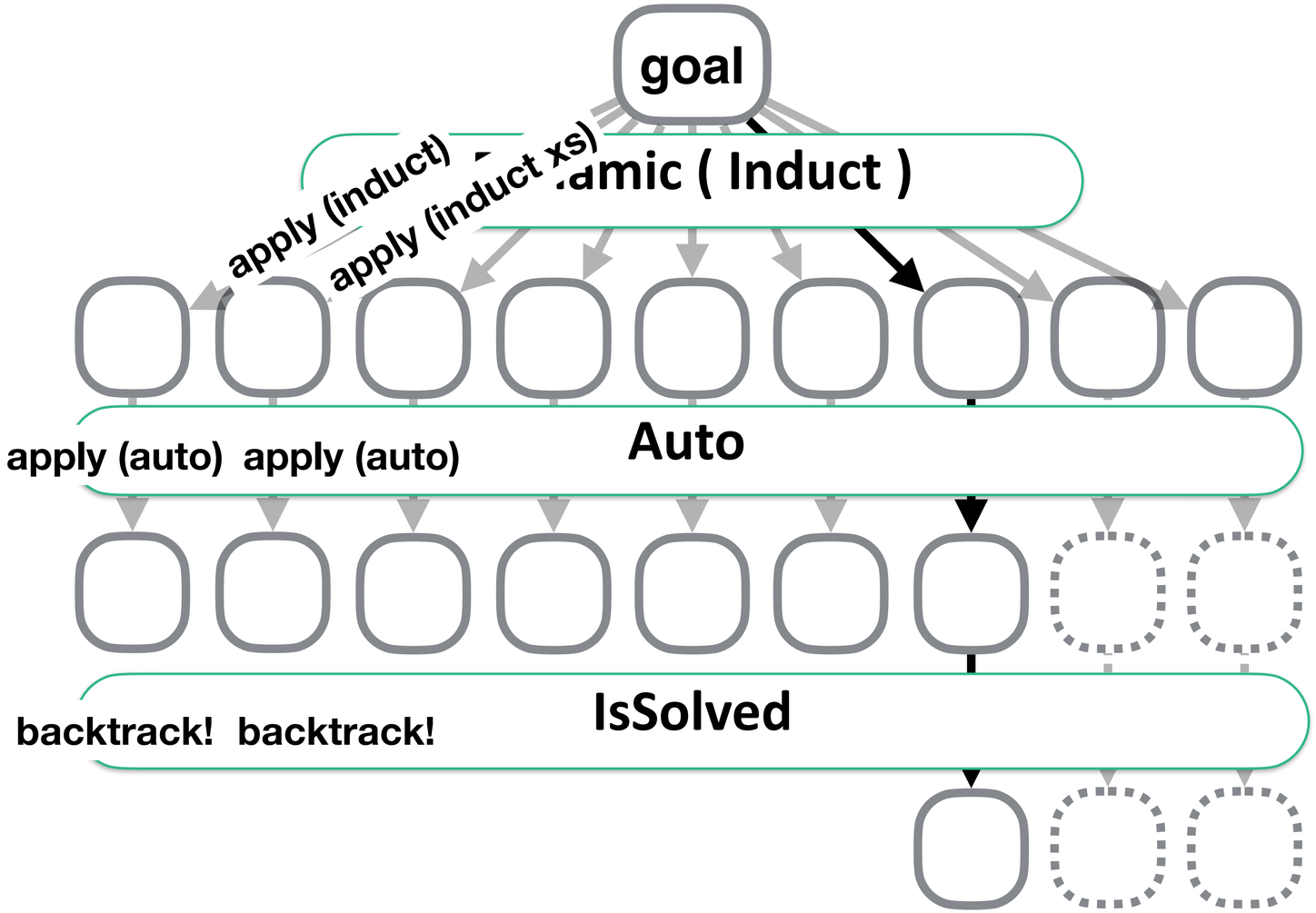
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



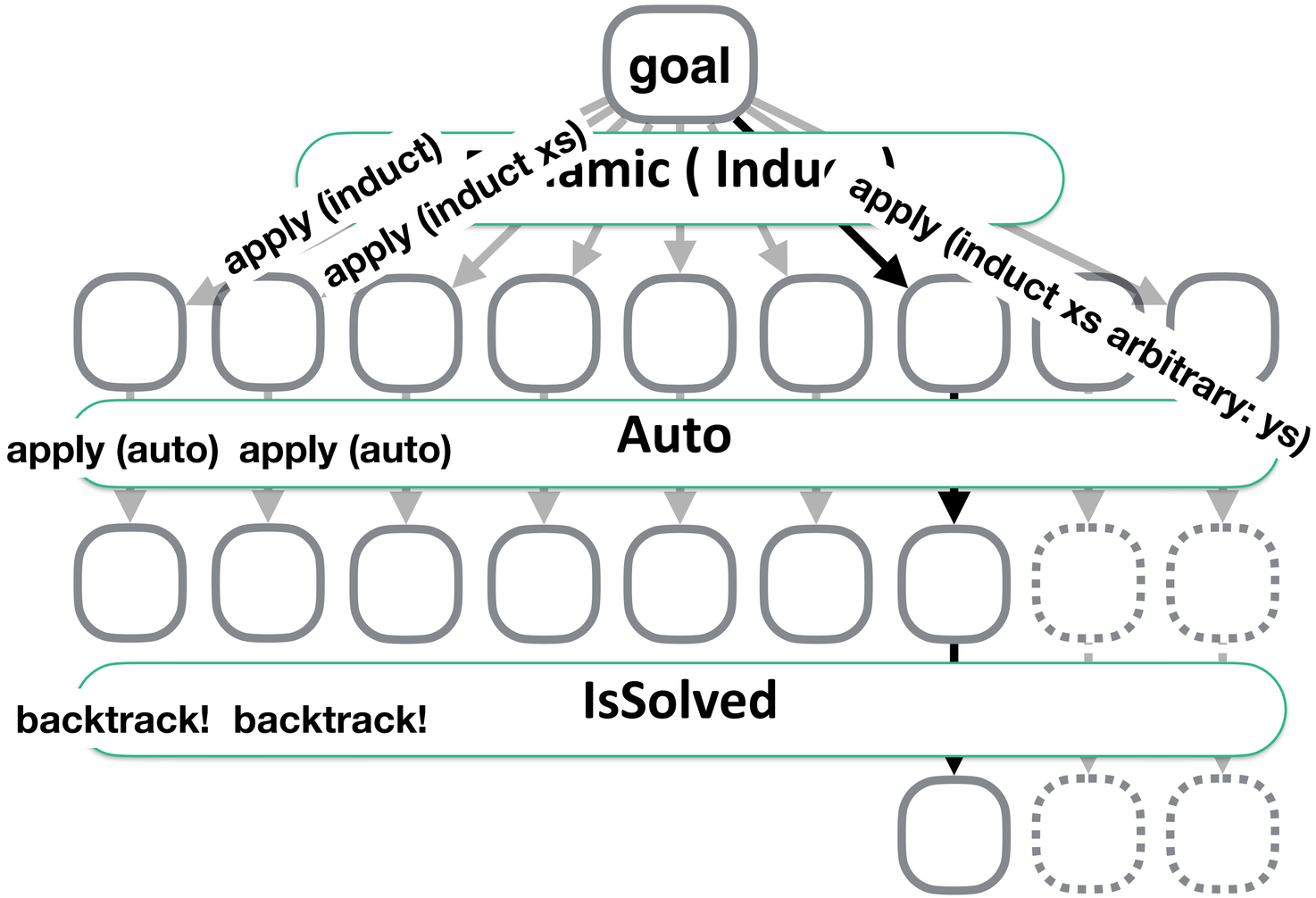
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



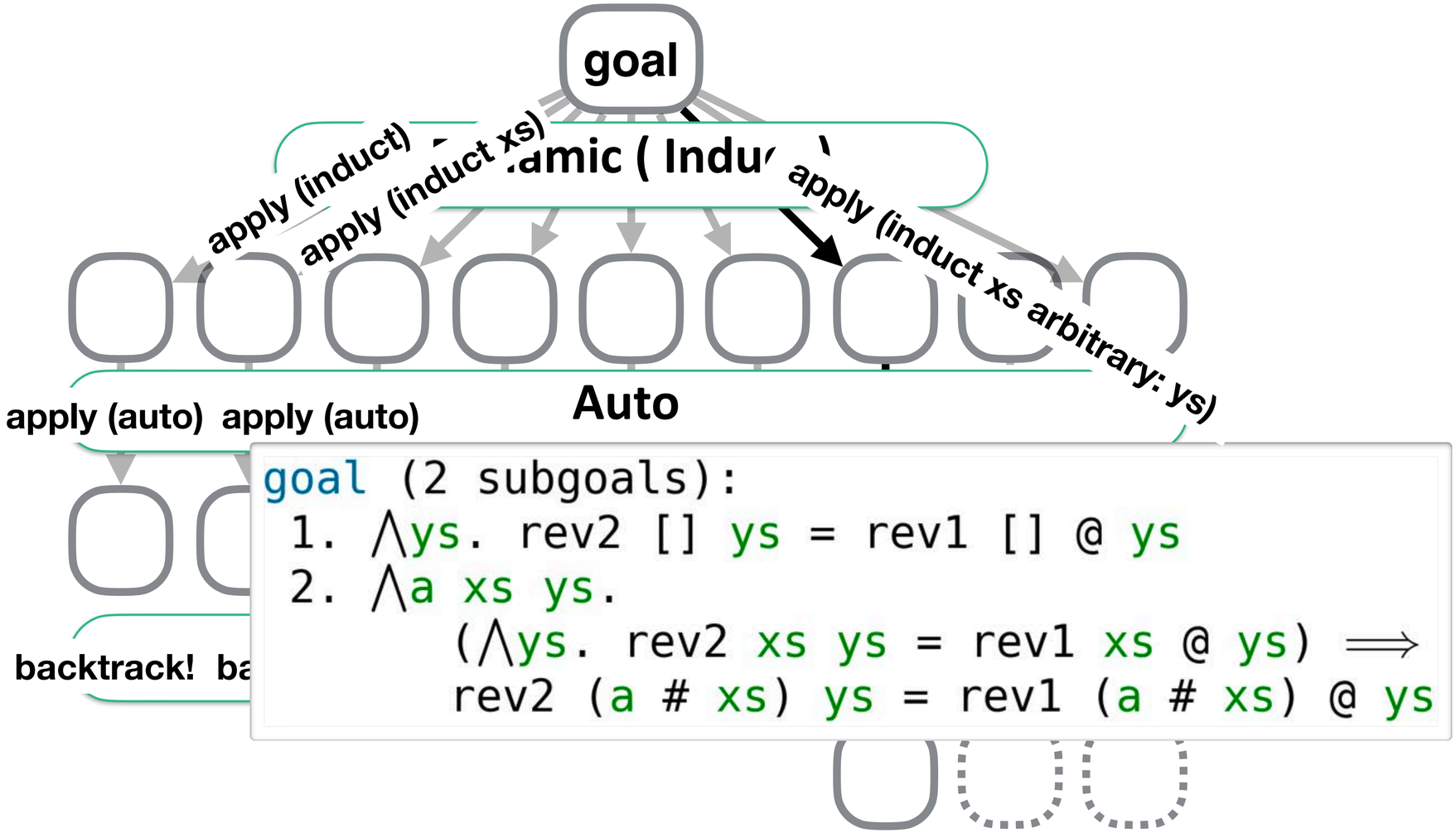
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



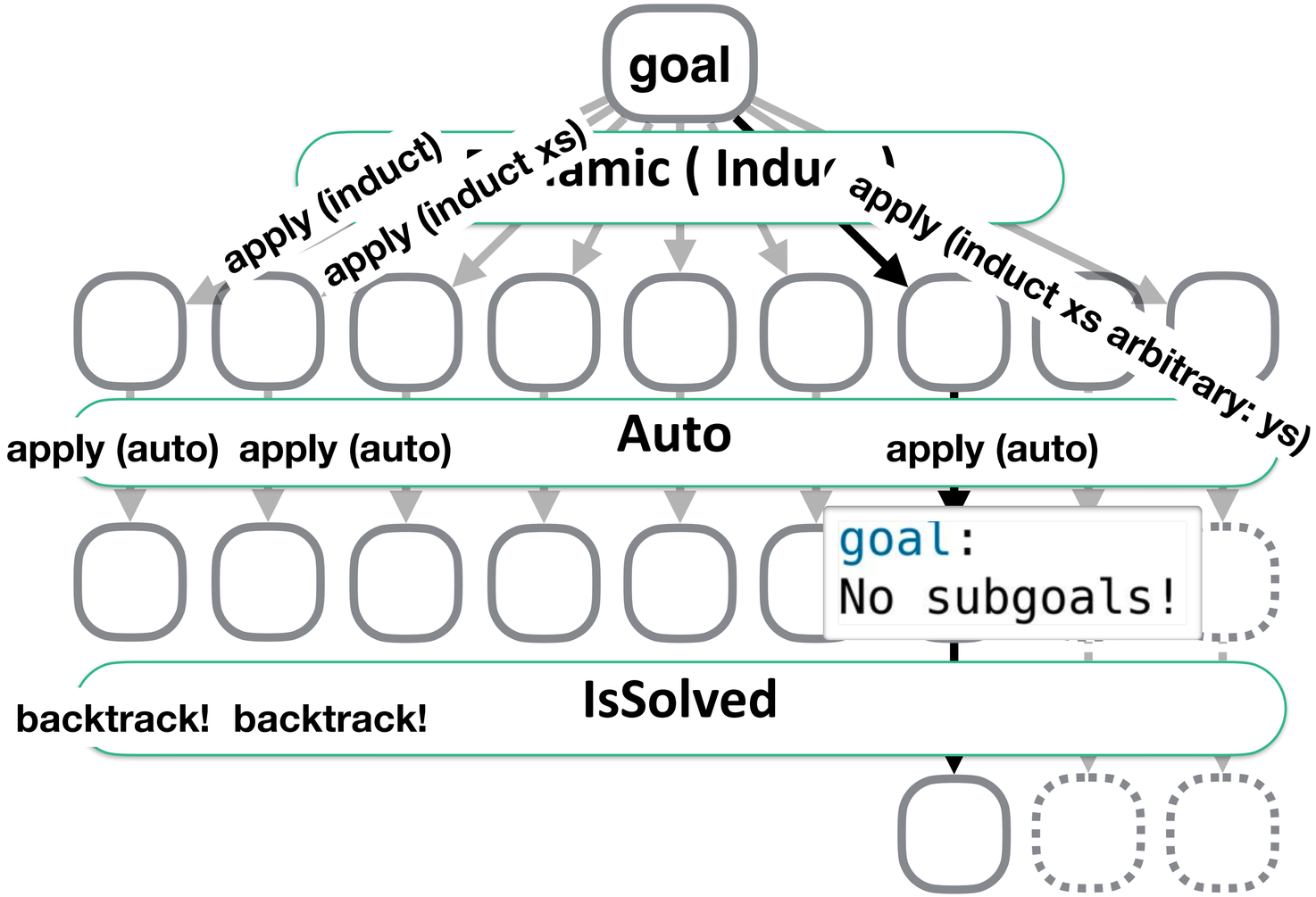
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



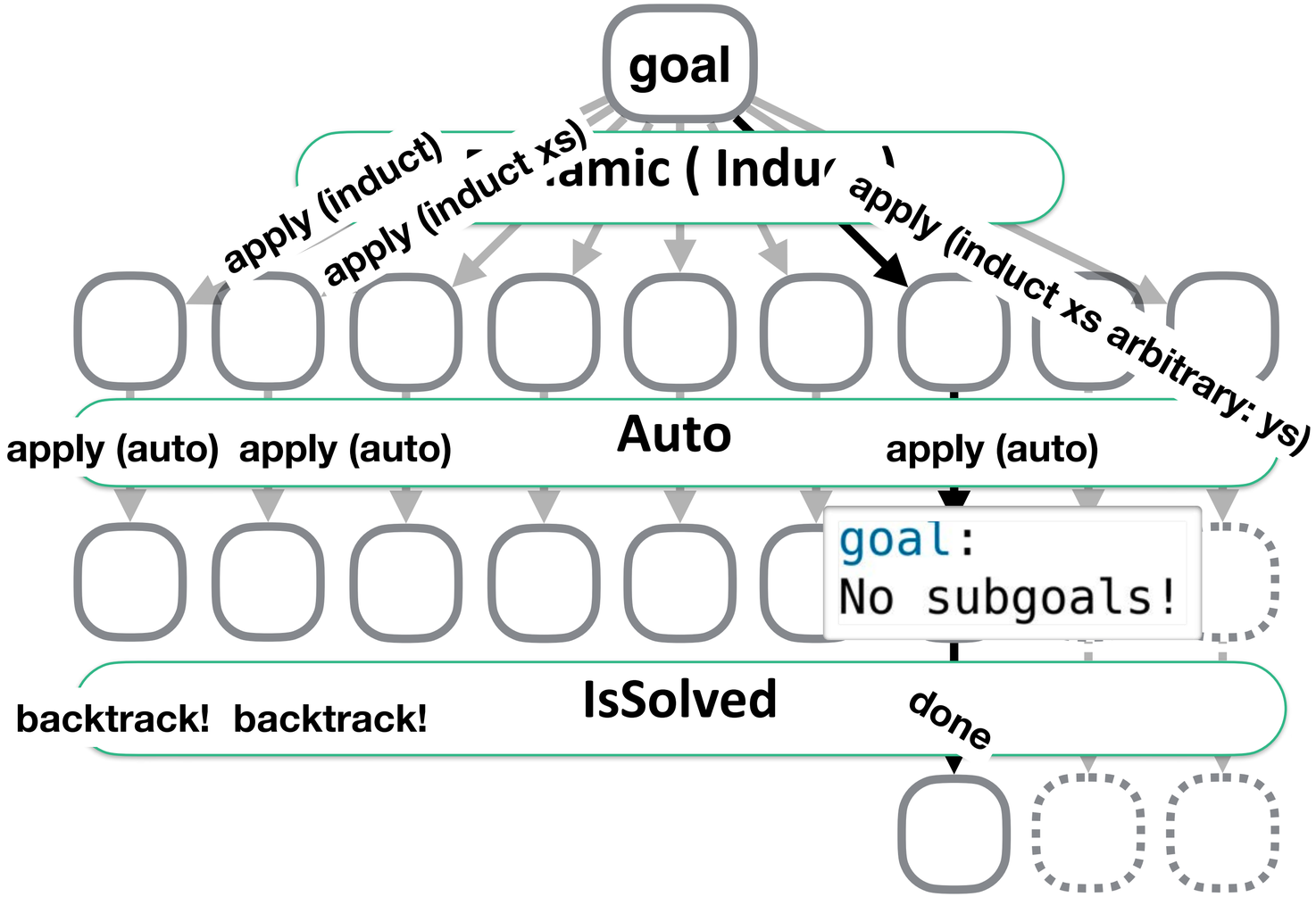
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



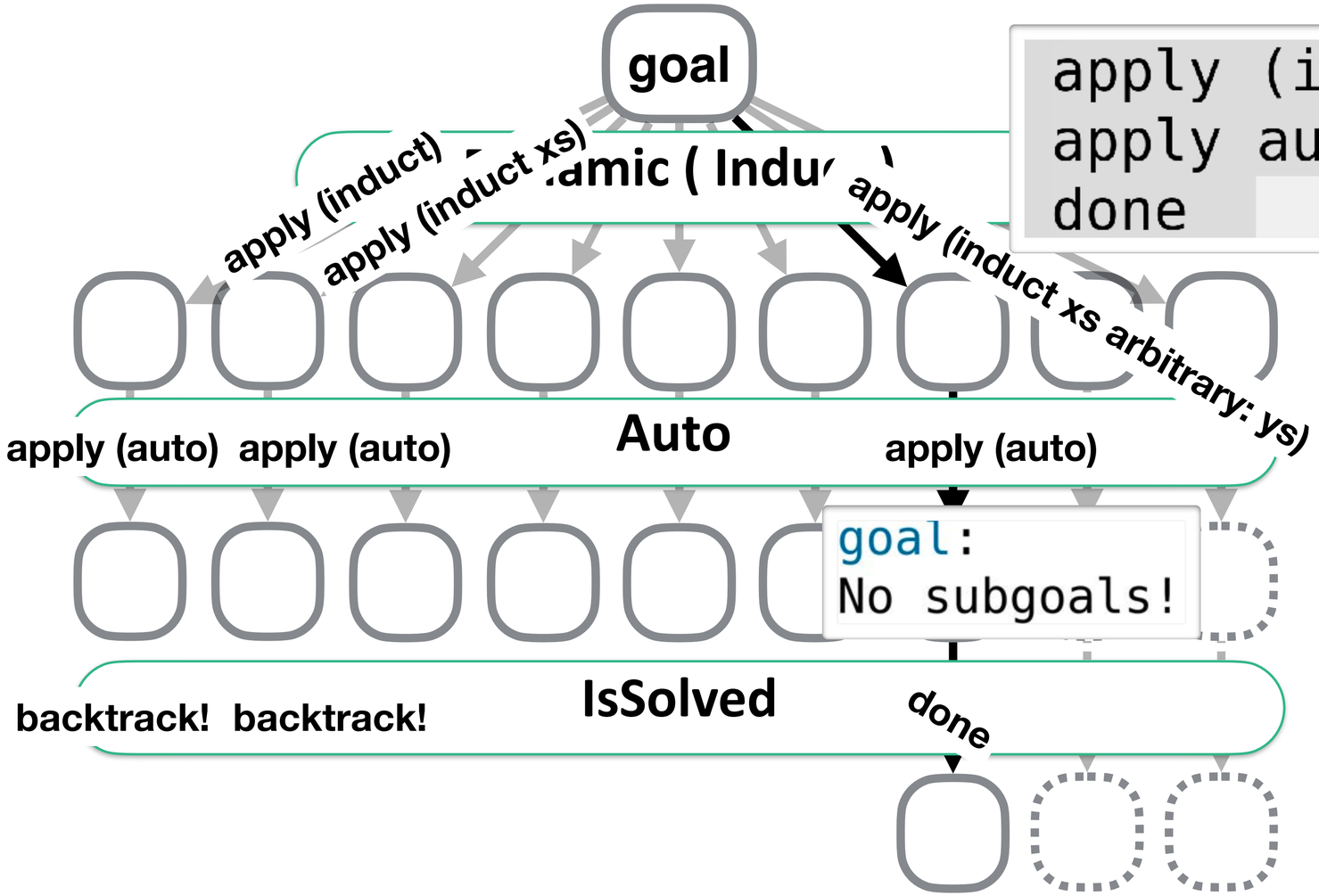
```
theorem "rev2 xs ys = rev1 xs @ ys"
  find_proof DInd(*Thens [Dynamic (Induct), Auto, IsSolved]*)
```



```
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]*)
```




```
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) 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*)

```
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) 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) 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

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*)

Indeed, auto returned quickly in our case.

What if subsequent steps are more involved?

We have to identify promising arguments for induction tactics

<u>without</u> relying on a search!

Machine learning?

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
```

~~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
```

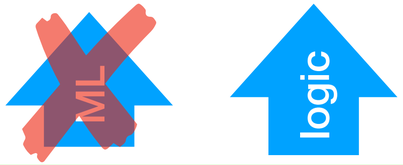
<- many concrete cases

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
```



<- 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
```

~~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**

```
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
```

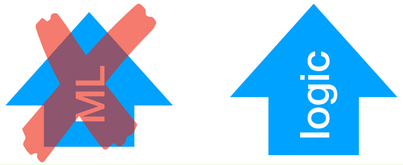
~~ML~~ logic <- abstraction using expressive logic

```
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

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

<- one abstract representation



<- abstraction using expressive logic

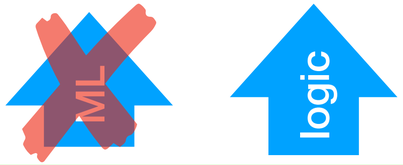
```
lemma "rev2 [1,2]          [] = rev1 [1,2]          @ []" by auto
lemma "rev2 [1,2,3]        [] = rev1 [1,2,3]        @ []" by auto
lemma "rev2 [''a'',''b''] [] = rev1 [''a'',''b''] @ []" by auto
lemma "rev2 [x,y,z]        [] = rev1 [x,y,z]        @ []" by auto
```

<- many concrete cases

Grand Challenge: Abstract Abstraction

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

<- one abstract representation



<- 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

Grand Challenge: Abstract Abstraction

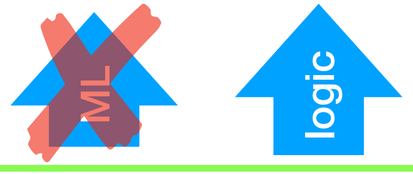
```
lemma "star r x y ⟹ star r y z ⟹ star r x z"
by(induction rule: star.induct)(auto simp: step)

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

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

<- small dataset about different domains

<- one abstract representation



<- 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

Grand Challenge: Abstract Abstraction

Abstract notion of “good” application of induction.
Heuristics that are valid across problem domains.

<- even more abstract

```
lemma "star r x y ⟹ star r y z ⟹ star r x z"
by(induction rule: star.induct)(auto simp: step)

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

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

<- small dataset about different domains

<- one abstract representation

~~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

It is difficult to write induction heuristics.

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

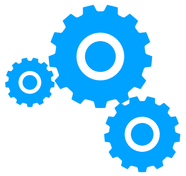
It is difficult to write induction heuristics.

```
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



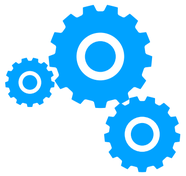
It is difficult to write induction heuristics.

```
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.

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

Simply because xs and ys are arguments of rev2?

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.

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

Simply because xs and ys are arguments of rev2?

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

experts
(= me)

Simply because is1 s stk are arguments of exec?

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
```

Simply because xs and ys are arguments of rev2?

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

experts
(= me)

Simply because is1 s stk are arguments of exec?

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
```

Simply because xs and ys are arguments of rev2?

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

experts
(= me)

Simply because is1 s stk are arguments of exec?

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
```

Simply because xs and ys are arguments of rev2?

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

experts
(= me)

Simply because is1 s stk are arguments of exec?

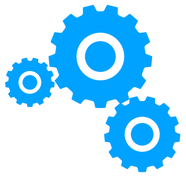
goal

?

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
```

Simply because xs and ys are arguments of rev2?

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

experts
(= me)

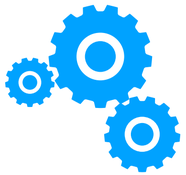
I can't refer to terms w/o knowing proof goals.

Simply because is1 s stk are arguments of exec?

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!

Many key challenges remain

Unsupervised Learning

Memory and one-shot learning

Imagination-based Planning with Generative Models

Learning Abstract Concepts

Transfer Learning

Language understanding

CENTER FOR
Brains
Minds+
Machines

March 20, 2019

The Power of
Self-Learning Systems

Demis Hassabis

DeepMind

https://cbmm.mit.edu/video/power-self-learning-systems

Many key challenges remain

Unsupervised Learning

Memory and one-shot learning

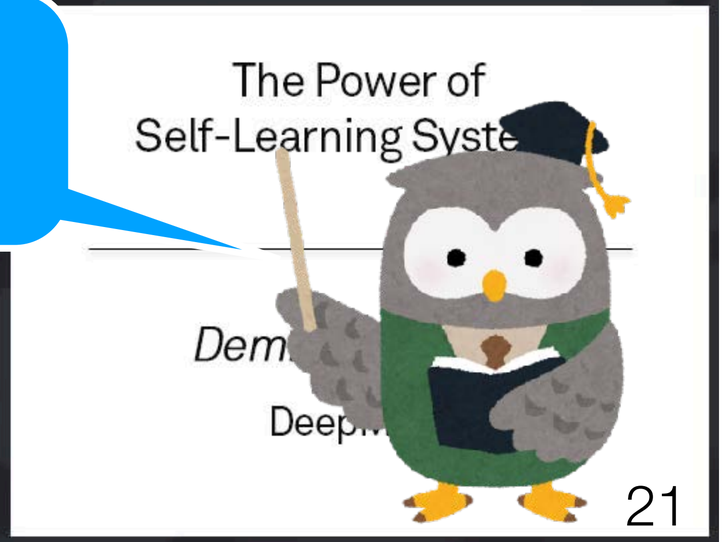
Imagination-based Planning with Generative Models

Learning Abstract Concepts

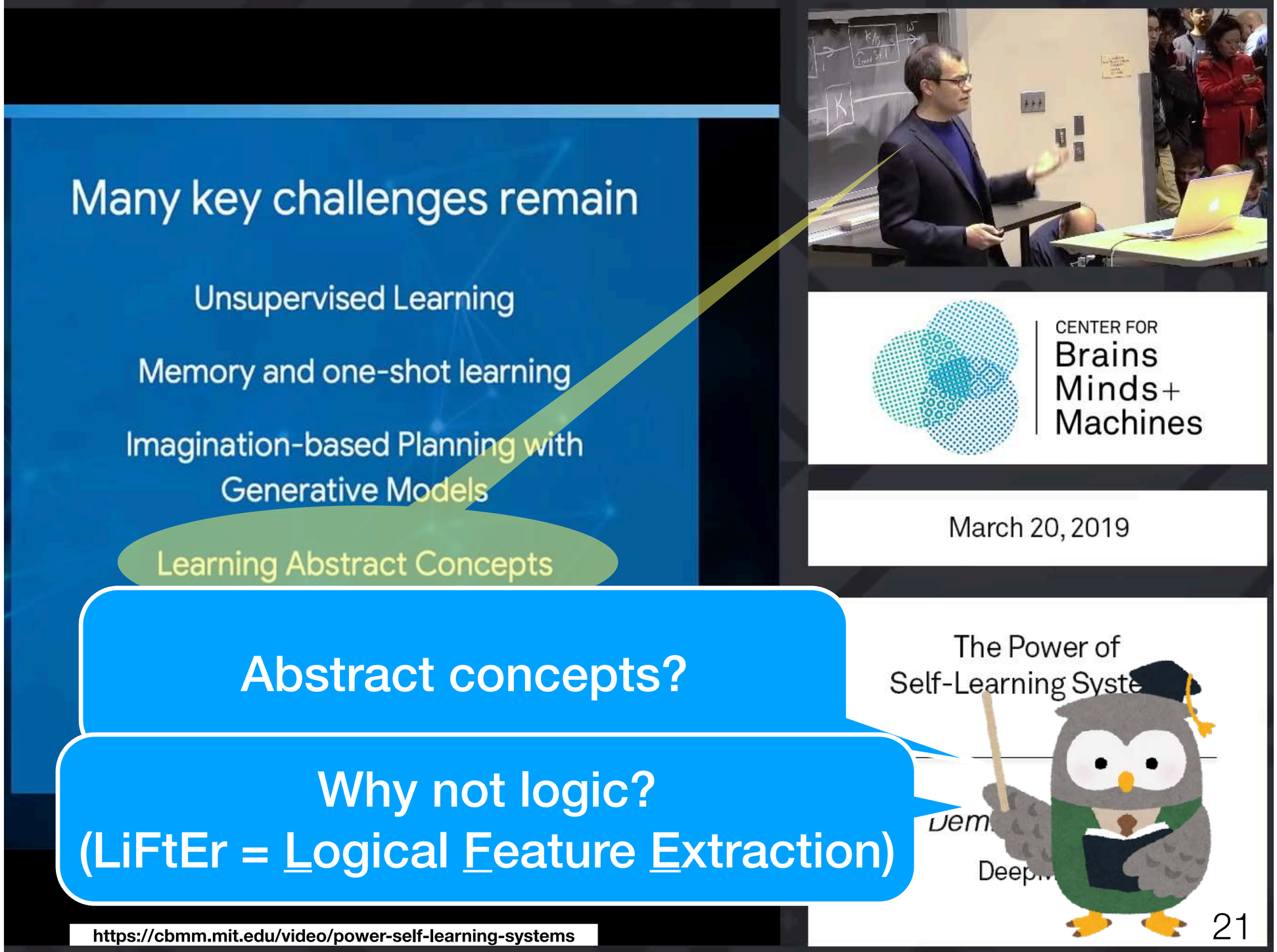
Abstract concepts?

CENTER FOR
Brains
Minds+
Machines

March 20, 2019



https://cbmm.mit.edu/video/power-self-learning-systems



Abstract concepts?

Why not logic?
(LiFtEr = Logical Feature Extraction)

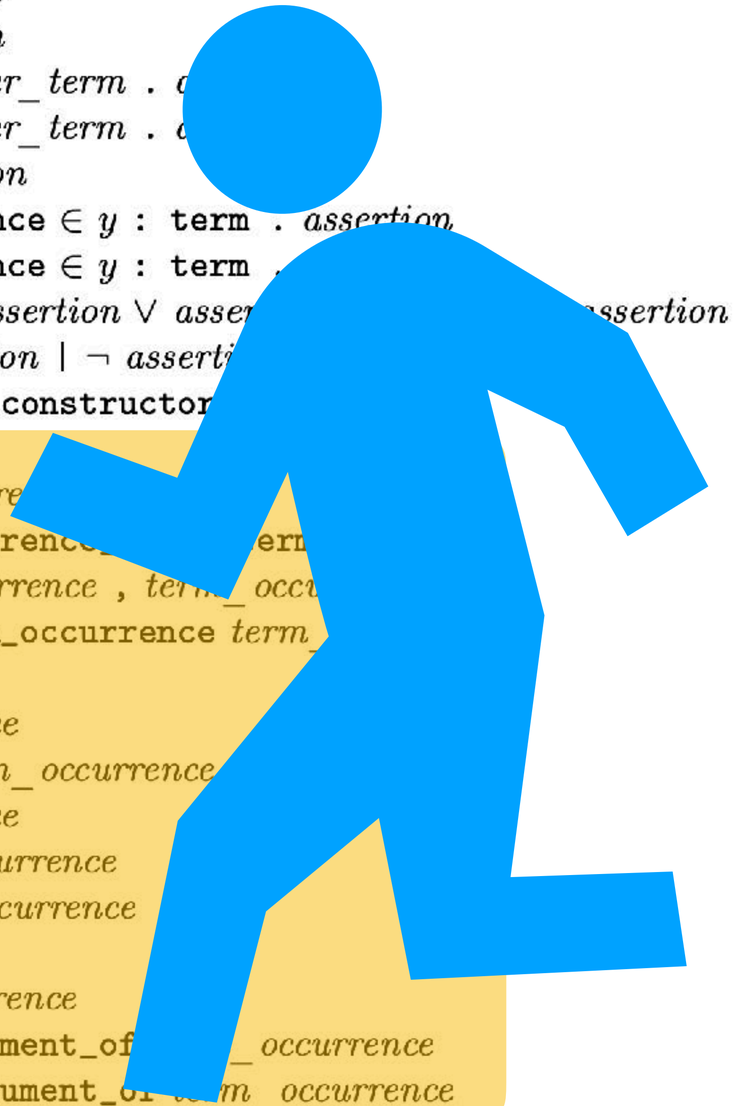
https://cbmm.mit.edu/video/power-self-learning-systems

Abstract Syntax of LiFtEr

```
assertion := atomic | connective | quantifier | ( assertion )
type := term | term_occurrence | rule | number
modifier_term := induction_term | arbitrary_term
quantifier := ∃x : type . assertion
            | ∀x : type . assertion
            | ∃x : term ∈ modifier_term . assertion
            | ∀x : term ∈ modifier_term . assertion
            | ∃x : rule . assertion
            | ∃x : term_occurrence ∈ y : term . assertion
            | ∀x : term_occurrence ∈ y : term . assertion
connective := True | False | assertion ∨ assertion | assertion ∧ assertion
              assertion → assertion | ¬ 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* $\vee$ *asser* *assertion*
assertion $\rightarrow$ *assertion* | $\neg$ *asserti*
pattern := `all_only_var` | `all_constructor`
atomic :=
rule `is_rule_of` *term_occurre*
| *term_occurrence* `term_occurrence` *term*
| `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` *term_occurrence*



Example Assertion in LiFtEr (in Abstract Syntax)

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

conjunction

Example Assertion in LiFtEr (in Abstract Syntax)

implication

variable for auxiliary lemmas

conjunction

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

variable for auxiliary lemmas

variable for terms

conjunction

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

variable for auxiliary lemmas

variable for terms

variable for term occurrences

conjunction

Example Assertion in LiFtEr (in Abstract Syntax)

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

(r1 = rev2.induct)

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

```
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

(r1 = rev2.induct)

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

```
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

```
∃ 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 = rev2.induct)
(t1 = rev2)

```
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
```

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 = rev2.induct)
(t1 = rev2)
(to1 = rev2)

```
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
```

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 = rev2.induct)
(t1 = rev2)
(to1 = rev2)

```
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
```

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 = rev2.induct)
(t1 = rev2)
(to1 = rev2)

True! r1 (= rev2.induct) is a lemma about to1 (= rev2).

```
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
```

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 = rev2.induct)
(t1 = rev2)
(to1 = rev2)

True! r1 (= rev2.induct) is a lemma about to1 (= rev2).

```
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
```

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 = rev2.induct)
(t1 = rev2)
(to1 = rev2)

True! r1 (= rev2.induct) is a lemma about to1 (= rev2).

```
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
```

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 = rev2.induct)
(t1 = rev2)
(to1 = rev2)

True! r1 (= rev2.induct) is a lemma about to1 (= rev2).

```
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
```

to1

t2

r1

```
∃ r1 : rule. True
→
∃ r1 : rule.                                        ( r1 = rev2.induct )
  ∃ t1 : term.                                      ( t1  = rev2 )
    ∃ to1 : term_occurrence ∈ t1 : term.            ( to1 = rev2 )
      r1 is_rule_of to1
```

True! r1 (= rev2.induct) is a lemma about to1 (= rev2).

```
    ∧
      ∀ 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 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
```

to1 to2 t2 r1

```
∃ r1 : rule. True
→
∃ r1 : rule.                                        ( r1 = rev2.induct )
  ∃ t1 : term.                                      ( t1  = rev2 )
    ∃ to1 : term_occurrence ∈ t1 : term.            ( to1 = rev2 )
      r1 is_rule_of to1
    ∧
      ∀ 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
```

True! r1 (= rev2.induct) is a lemma about to1 (= rev2).

```
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
```

to1 to2 t2 r1

```
∃ r1 : rule. True
→
∃ r1 : rule.                                        ( r1 = rev2.induct )
  ∃ t1 : term.                                      ( t1  = rev2 )
    ∃ to1 : term_occurrence ∈ t1 : term.            ( to1 = rev2 )
      r1 is_rule_of to1
    ∧
      ∀ 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
```

True! r1 (= rev2.induct) is a lemma about to1 (= rev2).

```
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
```

to1 to2 t2 r1

```
∃ r1 : rule. True
→
∃ r1 : rule.                                        ( r1 = rev2.induct )
  ∃ t1 : term.                                      ( t1  = rev2 )
    ∃ to1 : term_occurrence ∈ t1 : term.            ( to1 = rev2 )
      r1 is_rule_of to1
    ∧
      ∀ 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
```

True! r1 (= rev2.induct) is a lemma about to1 (= rev2).

```
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)"
```

to1 first to2

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

t2 first r1

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

(r1 = rev2.induct)
(t1 = rev2)
(to1 = rev2)

True! r1 (= rev2.induct) is a lemma about to1 (= rev2).

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

when t2 is xs (n = 1) ?

```
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)"
```

to1 first to2

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

t2 first r1

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

(r1 = rev2.induct)
(t1 = rev2)
(to1 = rev2)

True! r1 (= rev2.induct) is a lemma about to1 (= rev2).

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

when t2 is xs (n = 1)

```
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
```

to1 first second to2

t2 first second 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 = rev2.induct)
(t1 = rev2)
(to1 = rev2)

True! r1 (= rev2.induct) is a lemma about to1 (= rev2).

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

when t2 is xs (n = 1)

when t2 is ys (n = 2) ?

```
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
```

to1 first second to2

t2 first second 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 = rev2.induct)
(t1 = rev2)
(to1 = rev2)

True! r1 (= rev2.induct) is a lemma about to1 (= rev2).

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

when t2 is xs (n = 1)

when t2 is ys (n = 2) ?

```
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)"
```

to1 first second to2

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

t2 first second r1

$\exists$ $r1$: rule. True

$\rightarrow$

$\exists$ $r1$: rule. (r1 = rev2.induct)

$\exists$ $t1$: term. (t1 = rev2)

$\exists$ $to1$: term_occurrence $\in$ $t1$: term. (to1 = rev2)

$r1$ is_rule_of $to1$ True! r1 (= rev2.induct) is a lemma about to1 (= rev2).

$\wedge$

$\forall$ $t2$: term $\in$ induction_term. (t2 = xs and ys)

$\exists$ $to2$: term_occurrence $\in$ $t2$: term. (to2 = xs and ys)

$\exists$ n : number.

is_nth_argument_of ($to2$, n, $to1$) when t2 is xs (n = 1)

$\wedge$

$t2$ 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
```

the same LiFtEr assertion

$\exists\ r1$: **rule**. **True**

$\rightarrow$

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

new types ->

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

the same LiFtEr assertion

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

new types ->

```
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)"
```

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"
```

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)"
```

the same LiFtEr assertion

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
```

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
```

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

```
∃ 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)

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

$\exists\ r1$: rule. True

$\rightarrow$

$\exists\ r1$: rule.

(r1 = exec.induct)

$\exists\ t1$: term.

$\exists\ to1$: term_occurrence $\in$ $t1$: term.

$r1$ is_rule_of $to1$

$\wedge$

$\forall\ t2$: term $\in$ induction_term.

$\exists\ to2$: term_occurrence $\in$ $t2$: term.

$\exists\ n$: number.

is_nth_argument_of ($to2$, n, $to1$)

$\wedge$

$t2$ is_nth_induction_term n

new types ->

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

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

```
∃ 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)

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 ->

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

$\exists\ r1$: rule. True

$\rightarrow$

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

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

new types ->

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

new constants ->

```
fun execl :: "instr ⇒ state ⇒ stack ⇒ stack" where
 "execl (LOADI n) _  stk      =  n        # stk" |
 "execl (LOAD x)  s  stk      =  s(x)     # stk" |
 "execl  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 (execl 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
```

to1

r1

$\exists\ r1$: rule. True

$\rightarrow$

$\exists\ r1$: rule. (r1 = exec.induct)

$\exists\ t1$: term. (t1 = exec)

$\exists\ to1$: term_occurrence $\in$ $t1$: term. (to1 = exec)

$r1$ is_rule_of $to1$

$\wedge$

$\forall\ t2$: term $\in$ induction_term.

$\exists\ to2$: term_occurrence $\in$ $t2$: term.

$\exists\ n$: number.

is_nth_argument_of ($to2$, n, $to1$)

$\wedge$

$t2$ is_nth_induction_term n

new types ->

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

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)"
```

to1

a model proof ->

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

r1

$\exists\ r1\ :\ \texttt{rule.}\ \texttt{True}$

$\rightarrow$

$\exists\ r1\ :\ \texttt{rule.}$ (r1 = exec.induct)

$\exists\ t1\ :\ \texttt{term.}$ (t1 = exec)

$\exists\ to1\ :\ \texttt{term_occurrence} \in t1\ :\ \texttt{term.}$ (to1 = exec)

$r1\ \texttt{is_rule_of}\ to1$ True! r1 (= exec.induct) is a lemma about to1 (= exec).

$\wedge$

$\forall\ t2\ :\ \texttt{term} \in \texttt{induction_term.}$

$\exists\ to2\ :\ \texttt{term_occurrence} \in t2\ :\ \texttt{term.}$

$\exists\ n\ :\ \texttt{number.}$

$\texttt{is_nth_argument_of}\ (to2,\ n,\ to1)$

$\wedge$

$t2\ \texttt{is_nth_induction_term}\ n$

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)"
```

to1

a model proof ->

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

r1

$\exists\ r1$: rule. True

$\rightarrow$

$\exists\ r1$: rule. (r1 = exec.induct)

$\exists\ t1$: term. (t1 = exec)

$\exists\ to1$: term_occurrence $\in$ $t1$: term. (to1 = exec)

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

$\wedge$

$\forall\ t2$: term $\in$ induction_term.

$\exists\ to2$: term_occurrence $\in$ $t2$: term.

$\exists\ n$: number.

is_nth_argument_of ($to2$, n, $to1$)

$\wedge$

$t2$ is_nth_induction_term n

new types ->

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

new constants ->

```
fun execl :: "instr ⇒ state ⇒ stack ⇒ stack" where
 "execl (LOADI n) _  stk      =  n        # stk" |
 "execl (LOAD x)  s  stk      =  s(x)     # stk" |
 "execl  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 (execl 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
```

to1

r1

$\exists$ $r1$: rule. True

$\rightarrow$

$\exists$ $r1$: rule. (r1 = exec.induct)

$\exists$ $t1$: term. (t1 = exec)

$\exists$ $to1$: term_occurrence $\in$ $t1$: term. (to1 = exec)

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

$\wedge$

$\forall$ $t2$: term $\in$ induction_term.

$\exists$ $to2$: term_occurrence $\in$ $t2$: term.

$\exists$ n : number.

is_nth_argument_of ($to2$, n, $to1$)

$\wedge$

$t2$ is_nth_induction_term n

new types ->

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

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
```

to1

r1

$\exists$ $r1$: rule. True

$\rightarrow$

$\exists$ $r1$: rule. (r1 = exec.induct)

$\exists$ $t1$: term. (t1 = exec)

$\exists$ $to1$: term_occurrence $\in$ $t1$: term. (to1 = exec)

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

$\wedge$

$\forall$ $t2$: term $\in$ induction_term.

$\exists$ $to2$: term_occurrence $\in$ $t2$: term.

$\exists$ n : number.

is_nth_argument_of ($to2$, n, $to1$)

$\wedge$

$t2$ is_nth_induction_term n

new types ->

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

new constants ->

```
fun execl :: "instr ⇒ state ⇒ stack ⇒ stack" where
 "execl (LOADI n) _  stk      =  n        # stk" |
 "execl (LOAD x)  s  stk      =  s(x)     # stk" |
 "execl  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 (execl 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 t01 r1

$\exists$ r1 : rule. True

$\rightarrow$

$\exists$ r1 : rule. (r1 = exec.induct)

$\exists$ t1 : term. (t1 = exec)

$\exists$ to1 : term_occurrence $\in$ t1 : term. (to1 = exec)

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

$\wedge$

$\forall$ t2 : term $\in$ induction_term. (t2 = is1, s, and stk)

$\exists$ to2 : term_occurrence $\in$ t2 : term.

$\exists$ n : number.

is_nth_argument_of (to2, n, to1)

$\wedge$

t2 is_nth_induction_term n

new types ->

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

new constants ->

```
fun execl :: "instr ⇒ state ⇒ stack ⇒ stack" where
 "execl (LOADI n) _  stk      =  n        # stk" |
 "execl (LOAD x)  s  stk      =  s(x)     # stk" |
 "execl  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 (execl 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 t01 t02 r1

$\exists$ $r1$: rule. True

$\rightarrow$

$\exists$ $r1$: rule. (r1 = exec.induct)

$\exists$ $t1$: term. (t1 = exec)

$\exists$ $to1$: term_occurrence $\in$ $t1$: term. (to1 = exec)

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

$\wedge$

$\forall$ $t2$: term $\in$ induction_term. (t2 = is1, s, and stk)

$\exists$ $to2$: term_occurrence $\in$ $t2$: term. (to2 = is1, s, and stk)

$\exists$ n : number.

is_nth_argument_of ($to2$, n, $to1$)

$\wedge$

$t2$ is_nth_induction_term n

new types ->

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

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 t01 r1 t02

$\exists\ r1$: rule. True

$\rightarrow$

$\exists\ r1$: rule. (r1 = exec.induct)

$\exists\ t1$: term. (t1 = exec)

$\exists\ to1$: term_occurrence $\in$ $t1$: term. (to1 = exec)

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

$\wedge$

$\forall\ t2$: term $\in$ induction_term. (t2 = is1, s, and stk)

$\exists\ to2$: term_occurrence $\in$ $t2$: term. (to2 = is1, s, and stk)

$\exists\ n$: number.

is_nth_argument_of ($to2$, n, $to1$)

$\wedge$

$t2$ is_nth_induction_term n

new types ->

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

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 t01 t02 r1

$\exists\ r1 : \texttt{rule}.\ \texttt{True}$

$\rightarrow$

$\exists\ r1 : \texttt{rule}.$ (r1 = exec.induct)

$\exists\ t1 : \texttt{term}.$ (t1 = exec)

$\exists\ to1 : \texttt{term_occurrence} \in t1 : \texttt{term}.$ (to1 = exec)

$r1\ \texttt{is_rule_of}\ to1$ True! r1 (= exec.induct) is a lemma about to1 (= exec).

$\wedge$

$\forall\ t2 : \texttt{term} \in \texttt{induction_term}.$ (t2 = is1, s, and stk)

$\exists\ to2 : \texttt{term_occurrence} \in t2 : \texttt{term}.$ (to2 = is1, s, and stk)

$\exists\ n : \texttt{number}.$

$\texttt{is_nth_argument_of}\ (to2,\ n,\ to1)$

$\wedge$

$t2\ \texttt{is_nth_induction_term}\ n$

new types ->

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

new constants ->

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

first

t2 t01

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
```

t02

first r1

$\exists$ $r1$: rule. True

$\rightarrow$

$\exists$ $r1$: rule. (r1 = exec.induct)

$\exists$ $t1$: term. (t1 = exec)

$\exists$ $to1$: term_occurrence $\in$ $t1$: term. (to1 = exec)

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

$\wedge$

$\forall$ $t2$: term $\in$ induction_term. (t2 = is1, s, and stk)

$\exists$ $to2$: term_occurrence $\in$ $t2$: term. (to2 = is1, s, and stk)

$\exists$ n : number.

is_nth_argument_of ($to2$, n, $to1$) when t2 is is1 (n -> 1) ?

$\wedge$

$t2$ 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)"
```

first

t2 t01

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
```

t02

first 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 = 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 execl :: "instr ⇒ state ⇒ stack ⇒ stack" where
 "execl (LOADI n) _  stk      =  n        # stk" |
 "execl (LOAD x)  s  stk      =  s(x)     # stk" |
 "execl  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 (execl i s stk)"
```

first second

t2 to1

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
```

to2

r1

first second

$\exists$ r1 : rule. True

$\rightarrow$

$\exists$ r1 : rule. (r1 = exec.induct)

$\exists$ t1 : term. (t1 = exec)

$\exists$ to1 : term_occurrence $\in$ t1 : term. (to1 = exec)

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

$\wedge$

$\forall$ t2 : term $\in$ induction_term. (t2 = is1, s, and stk)

$\exists$ to2 : term_occurrence $\in$ t2 : term. (to2 = is1, s, and stk)

$\exists$ n : number.

is_nth_argument_of (to2, n, to1) when t2 is is1 (n -> 1) ?

$\wedge$ when t2 is s (n -> 2) ?

t2 is_nth_induction_term n

new types ->

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

new constants ->

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

first second t2 to1 to2

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 first second

$\exists$ r1 : rule. True

$\rightarrow$

$\exists$ r1 : rule. (r1 = exec.induct)

$\exists$ t1 : term. (t1 = exec)

$\exists$ to1 : term_occurrence $\in$ t1 : term. (to1 = exec)

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

$\wedge$

$\forall$ t2 : term $\in$ induction_term. (t2 = is1, s, and stk)

$\exists$ to2 : term_occurrence $\in$ t2 : term. (to2 = is1, s, and stk)

$\exists$ n : number.

is_nth_argument_of (to2, n, to1) when t2 is is1 (n -> 1) ?

$\wedge$ when t2 is s (n -> 2) ?

t2 is_nth_induction_term n

new types ->

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

new constants ->

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

first second third

t2 t1

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 t1 t2 r1

first second third

$\exists$ $r1$: rule. True

$\rightarrow$

$\exists$ $r1$: rule. (r1 = exec.induct)

$\exists$ $t1$: term. (t1 = exec)

$\exists$ $to1$: term_occurrence $\in$ $t1$: term. (to1 = exec)

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

$\wedge$

$\forall$ $t2$: term $\in$ induction_term. (t2 = is1, s, and stk)

$\exists$ $to2$: term_occurrence $\in$ $t2$: term. (to2 = is1, s, and stk)

$\exists$ n : number.

is_nth_argument_of ($to2$, n, $to1$) when t2 is is1 (n -> 1) ?

$\wedge$ when t2 is s (n -> 2) ?

$t2$ is_nth_induction_term n when t2 is stk (n -> 3) ?

new types ->

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

new constants ->

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

first second third

t2 t1

new lemma ->

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

a model proof ->

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

t2

first second third r1

$\exists$ $r1$: rule. True

$\rightarrow$

$\exists$ $r1$: rule. (r1 = exec.induct)

$\exists$ $t1$: term. (t1 = exec)

$\exists$ $to1$: term_occurrence $\in$ $t1$: term. (to1 = exec)

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

$\wedge$

$\forall$ $t2$: term $\in$ induction_term. (t2 = is1, s, and stk)

$\exists$ $to2$: term_occurrence $\in$ $t2$: term. (to2 = is1, s, and stk)

$\exists$ n : number.

is_nth_argument_of ($to2$, n, $to1$) when t2 is is1 (n -> 1) ?

$\wedge$ when t2 is s (n -> 2) ?

$t2$ is_nth_induction_term n when t2 is stk (n -> 3) ?

new types ->

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

new constants ->

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

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

first third second

t2 to1

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
```

to2

r1

first second third

$\exists$ $r1$: rule. True

$\rightarrow$

$\exists$ $r1$: rule. (r1 = exec.induct)

$\exists$ $t1$: term. (t1 = exec)

$\exists$ $to1$: term_occurrence $\in$ $t1$: term. (to1 = exec)

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

$\wedge$

$\forall$ $t2$: term $\in$ induction_term. (t2 = is1, s, and stk)

$\exists$ $to2$: term_occurrence $\in$ $t2$: term. (to2 = is1, s, and stk)

$\exists$ n : number.

is_nth_argument_of ($to2$, n, $to1$) when t2 is is1 (n -> 1) ?

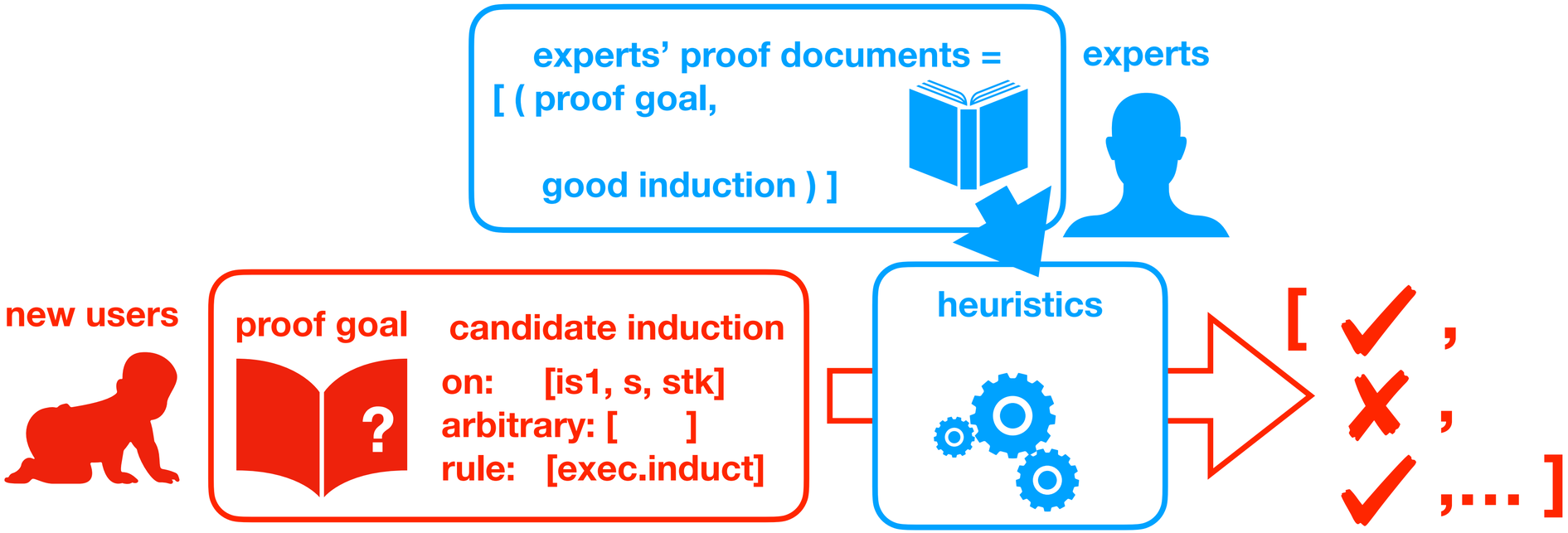
$\wedge$ when t2 is s (n -> 2) ?

$t2$ is_nth_induction_term n when t2 is stk (n -> 3) ?

It's hard 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
```




```
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's hard 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
```

Candidate induction?

experts' proof documents =
[(proof goal,

good induction)]

experts

new users

proof goal

candidate induction

on: [is1, s, stk]
arbitrary: [ ]
rule: [exec.induct]

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’s hard 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
```

Candidate induction?

experts’ proof documents = [(proof goal,

experts

Can’t you tell me what arguments to pass to the induct tactic?

new users

proof goal

candidate induction

on: [is1, s, stk]
arbitrary: [ ]
rule: [exec.induct]

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's hard 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
```

Candidate induction?

experts' proof documents = [(proof goal,

experts

Can't you tell me what arguments to pass to the induct tactic?

new users

proof goal

candidate induction

on: [is1, s, stk]

arbitrary: [ ]

heuristics

[✓ ,

✗ ,

✓ ,...]

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

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

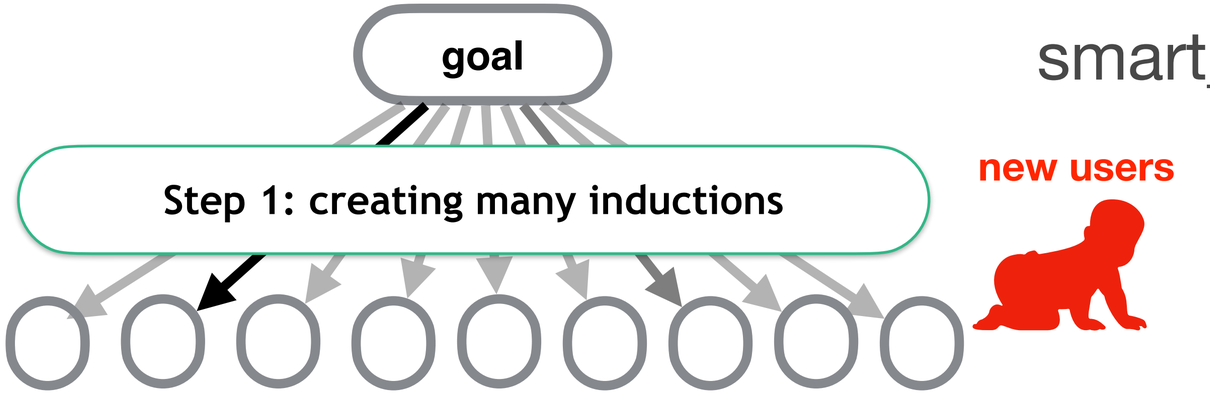
Algorithmic approaches work well, but not always.

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



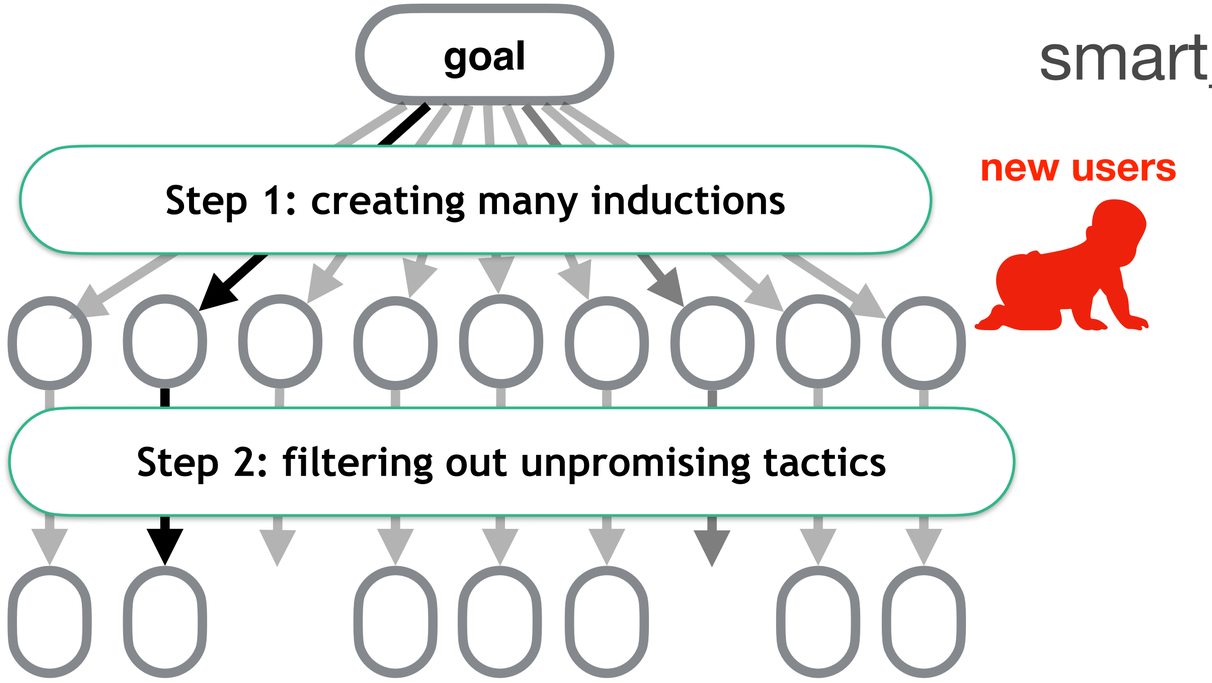
Algorithmic approaches work well, but not always.

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



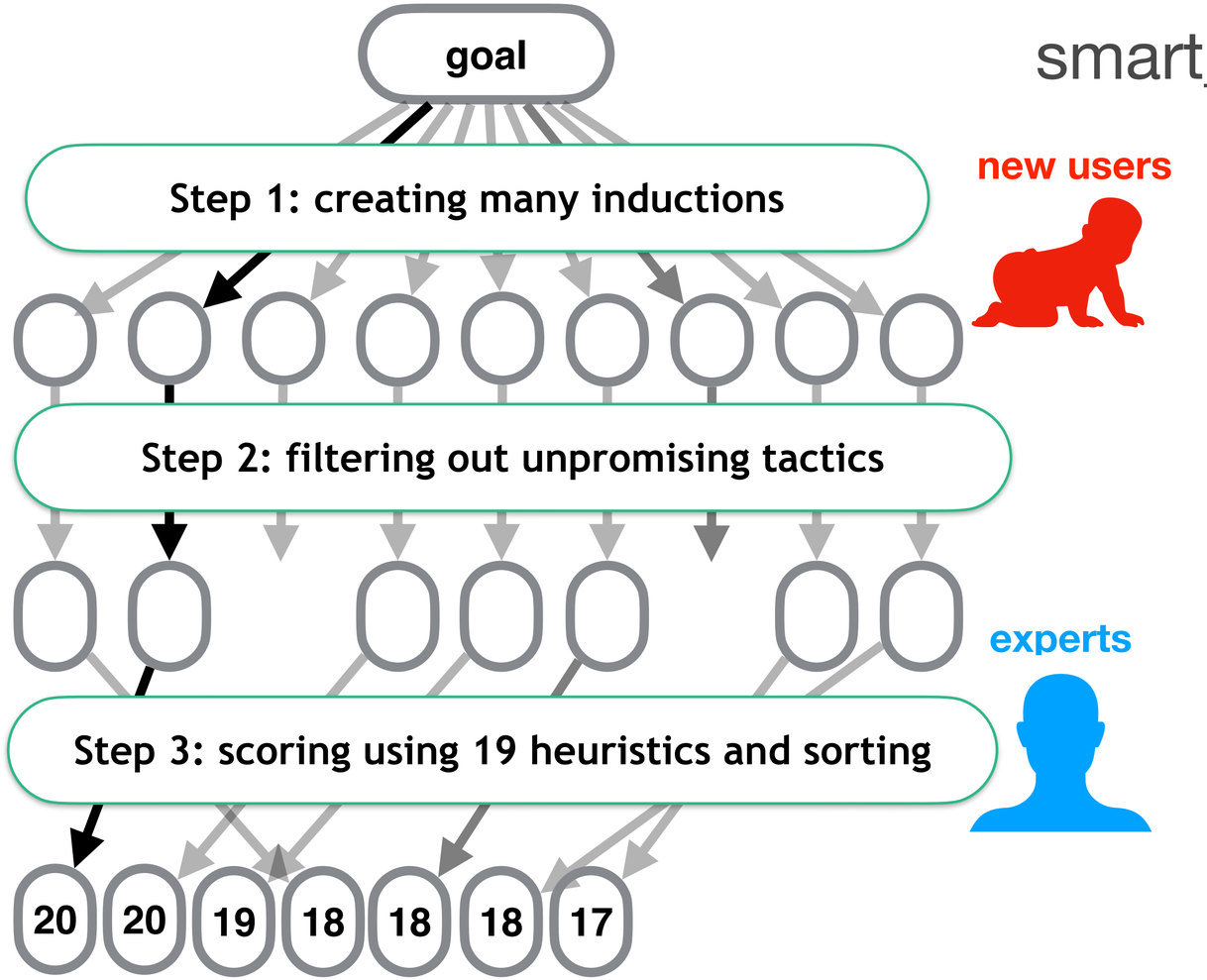
Algorithmic approaches work well, but not always.

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



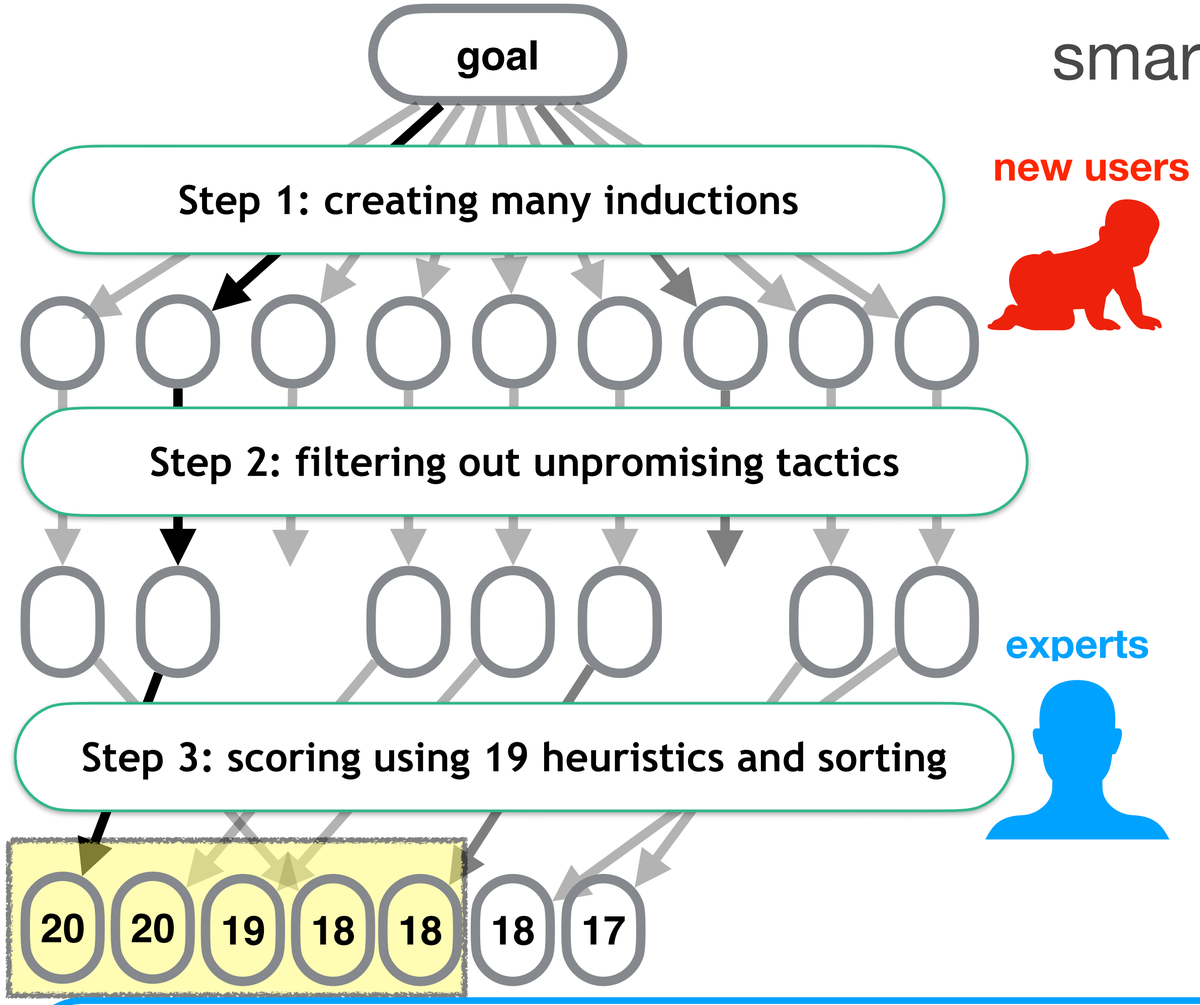
Algorithmic approaches work well, but not always.

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



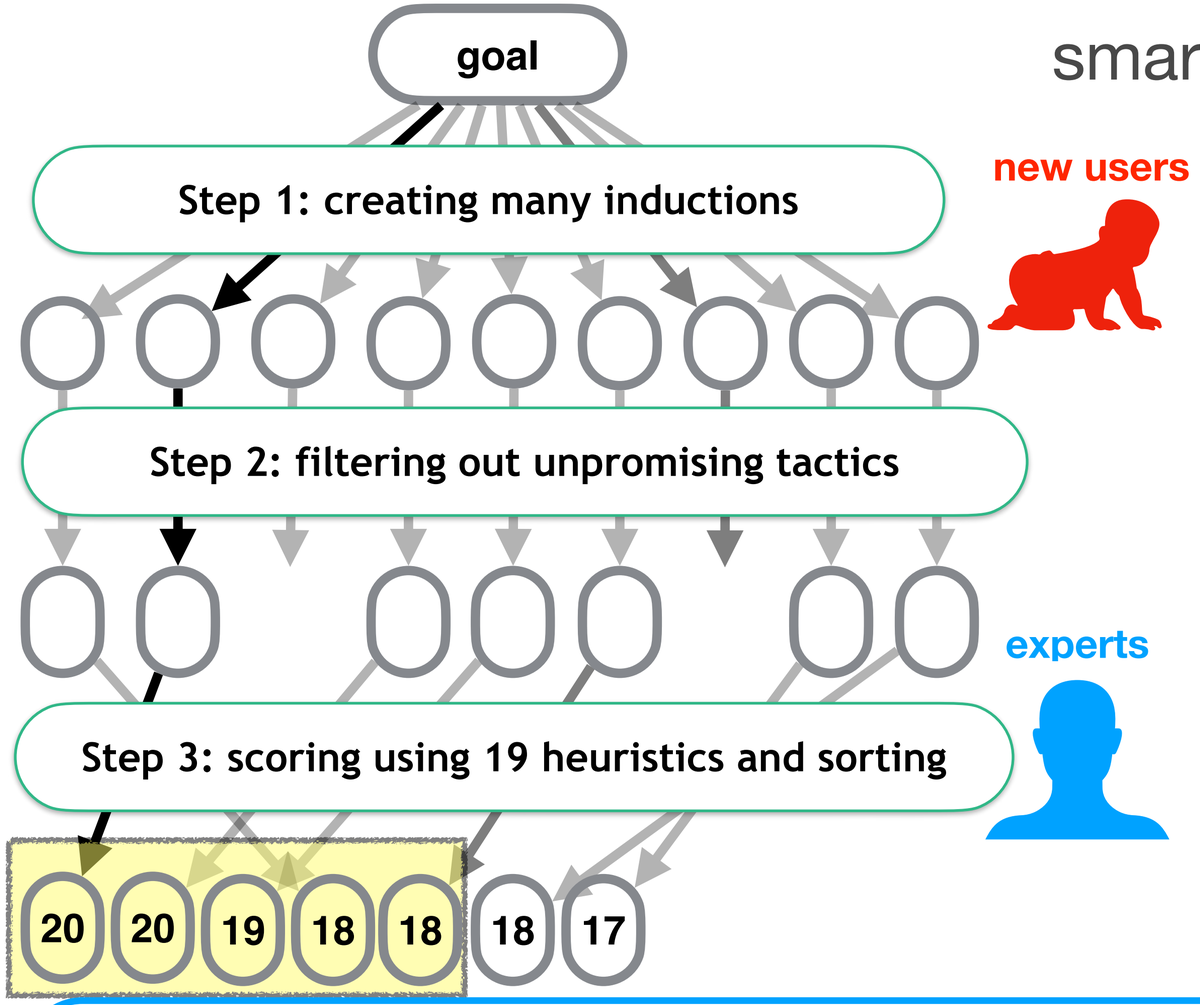
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

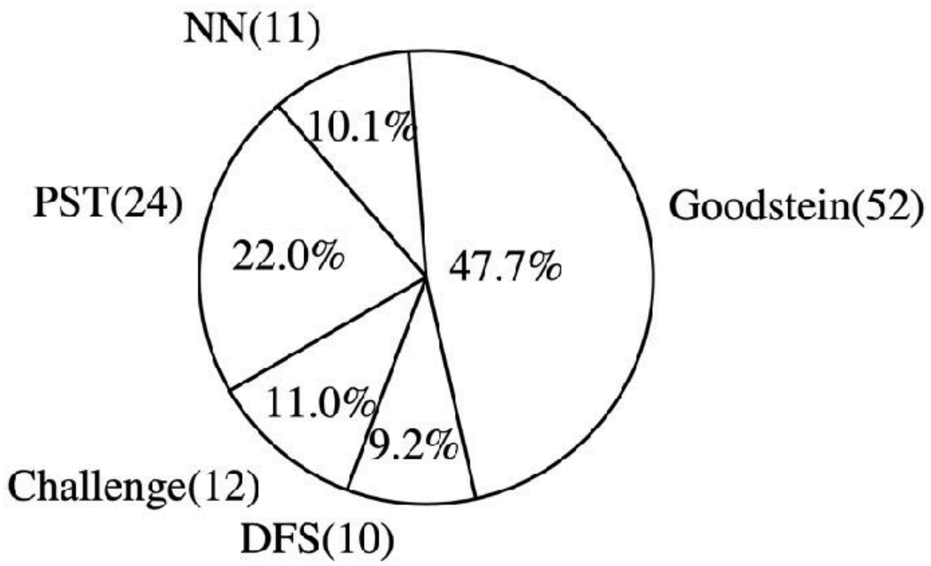
Abstract reasoning is often necessary.

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)



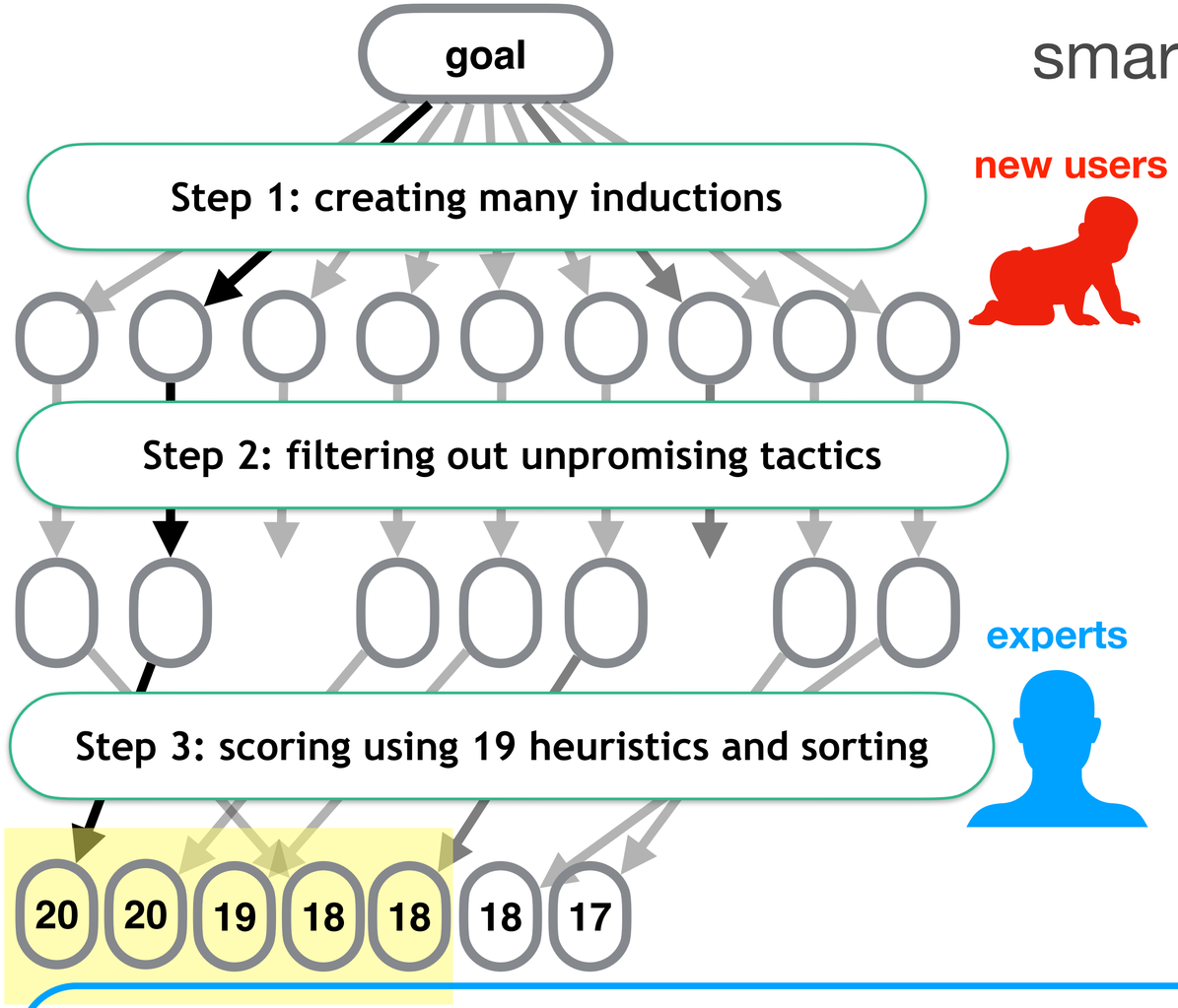
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

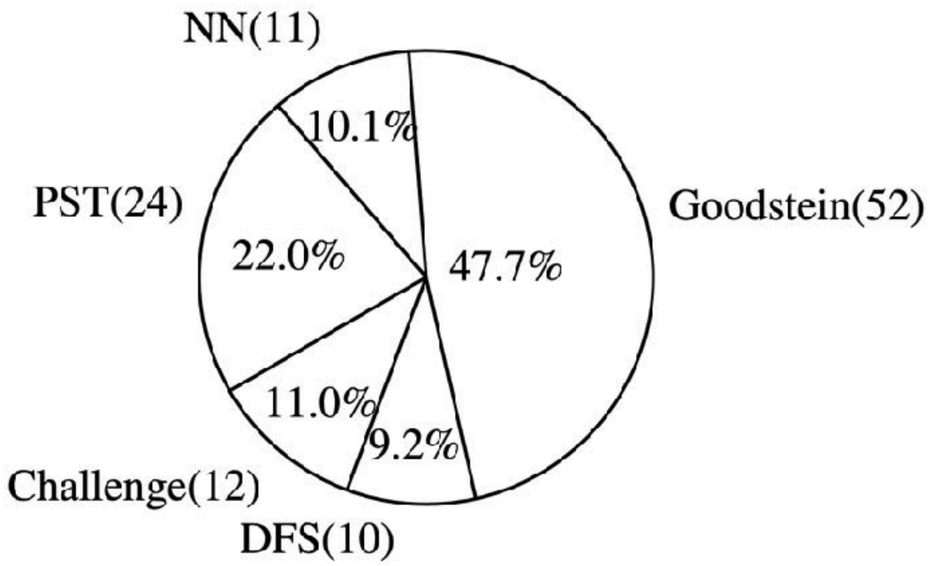
Abstract reasoning is often necessary.

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

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.

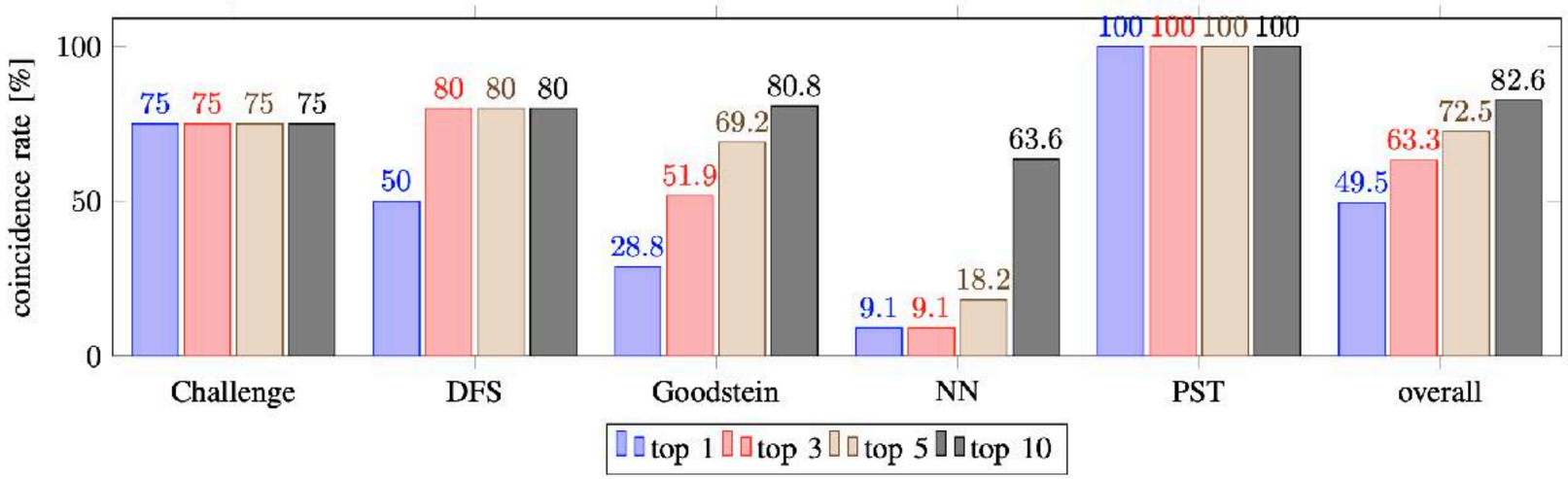
Algorithmic approaches work well, but not always.

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



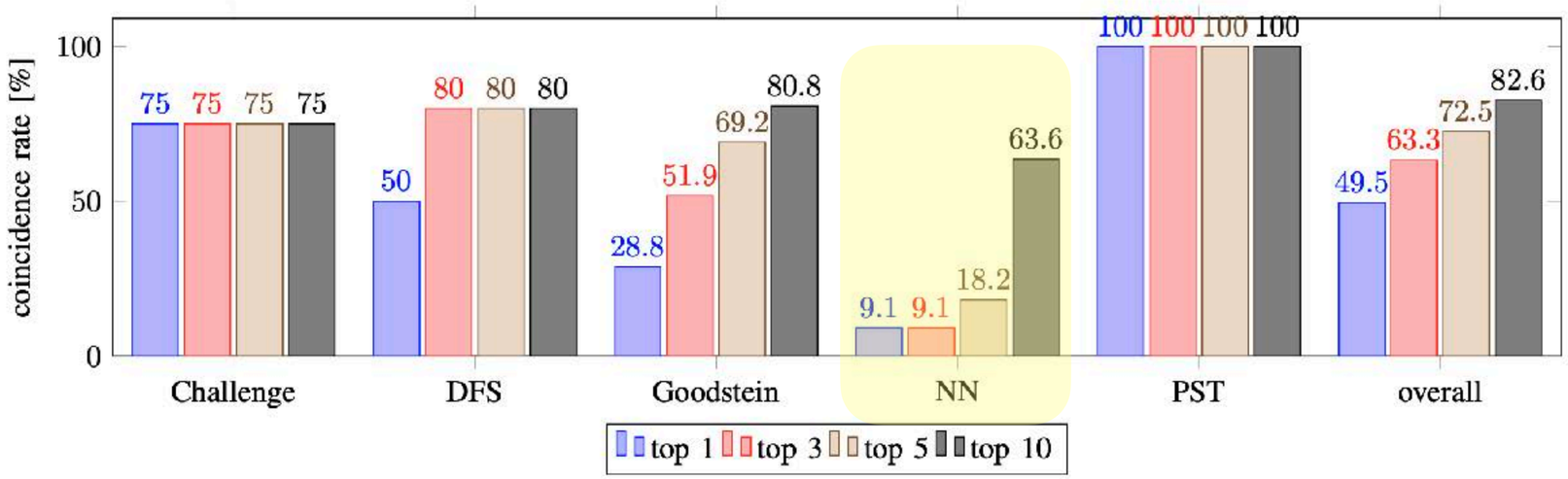
Algorithmic approaches work well, but not always.

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



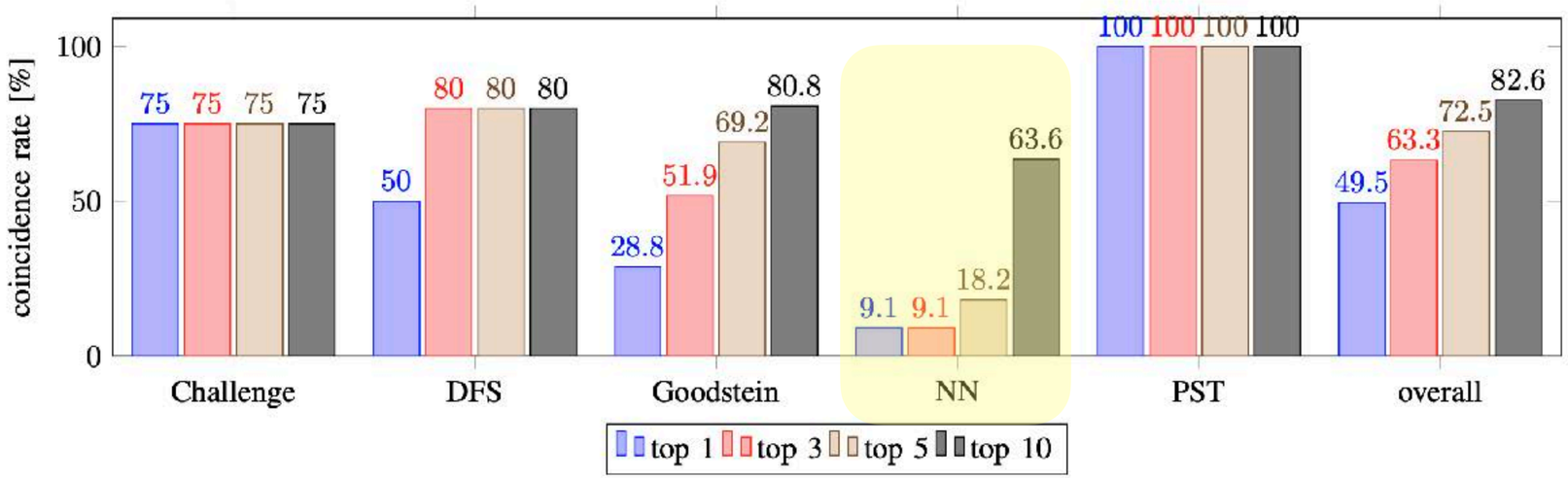
Algorithmic approaches work well, but not always.

There are many good ways to prove one theorem.

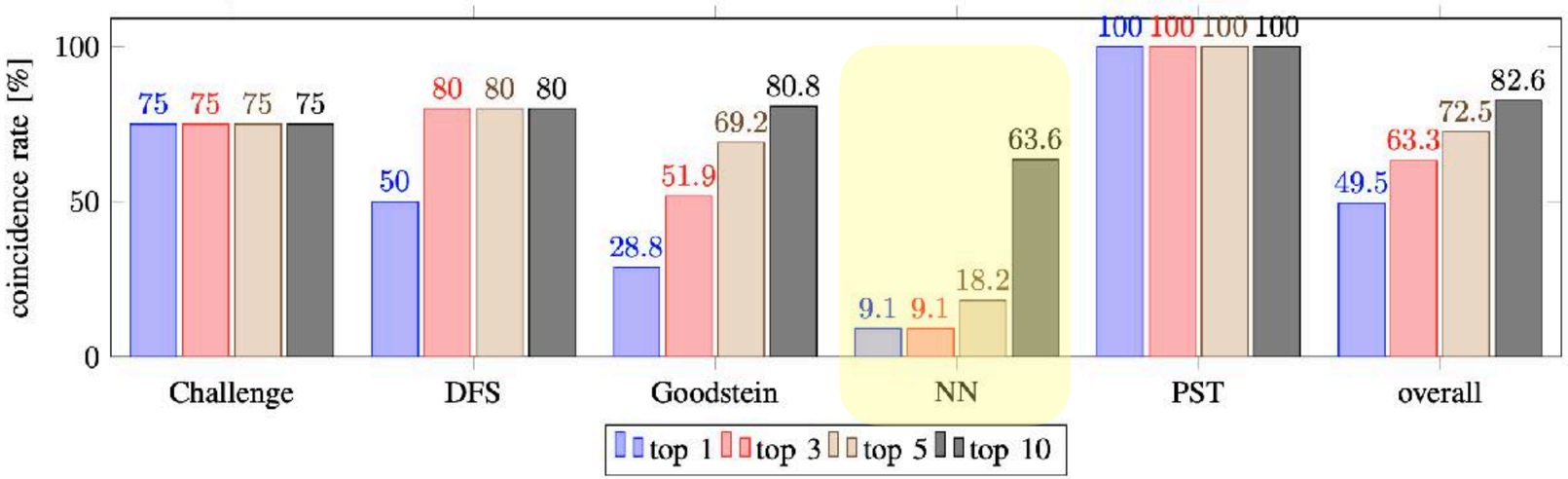
Abstract reasoning is often necessary.

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

Coincidence rates

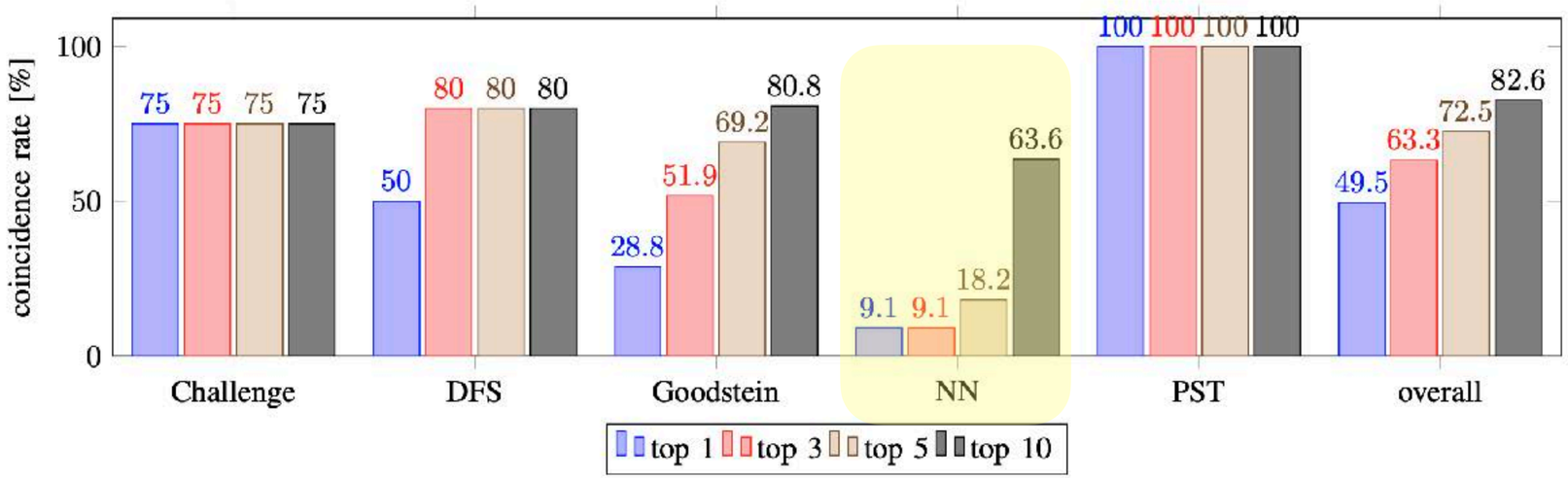


Coincidence rates



Is smart_induct Domain dependent?

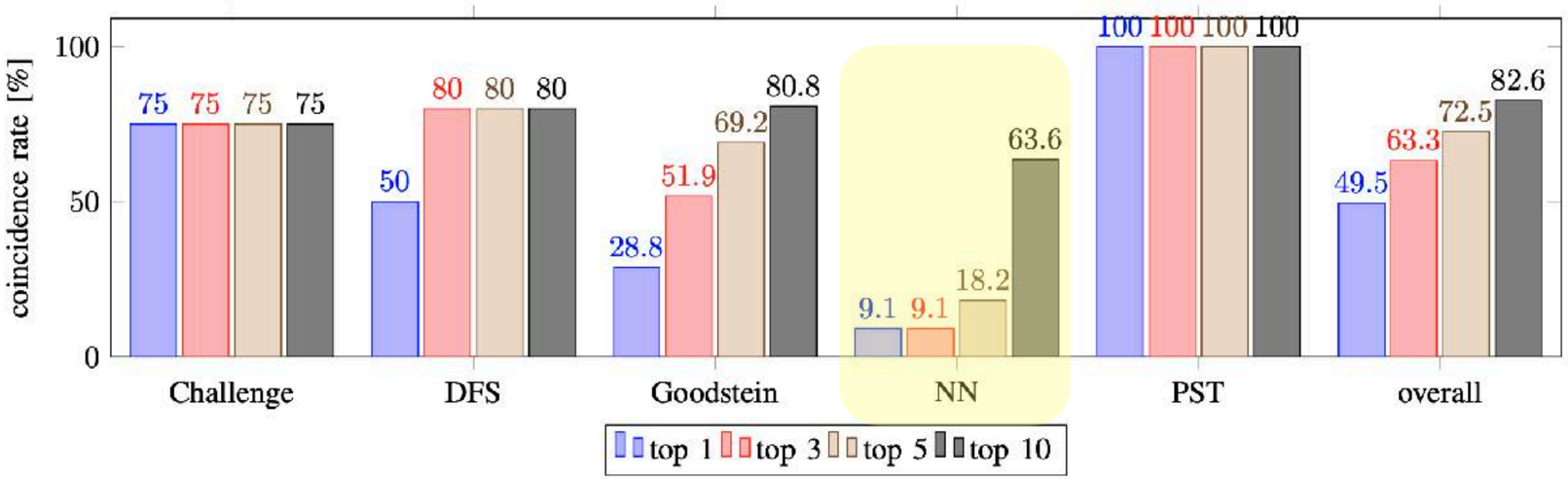
Coincidence rates

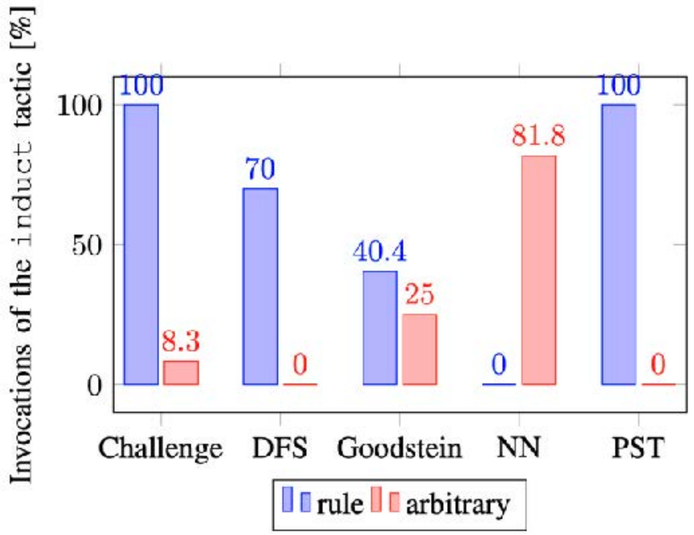


Is smart_induct Domain dependent?

smart_induct is bad at generalisation heuristics.

Coincidence rates

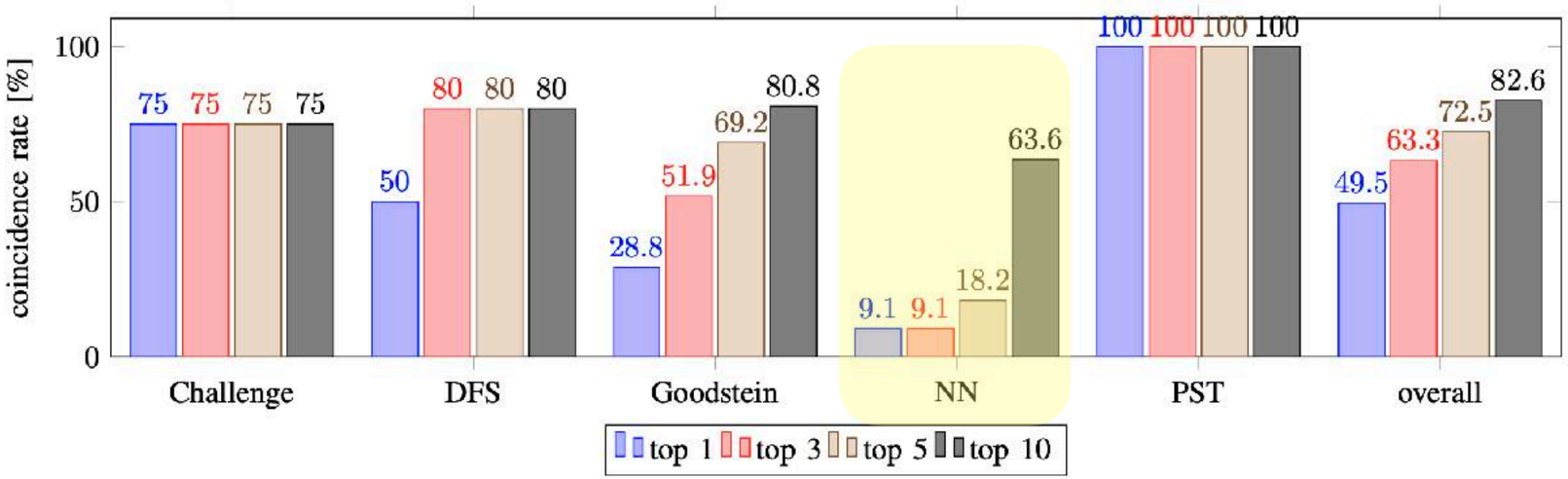


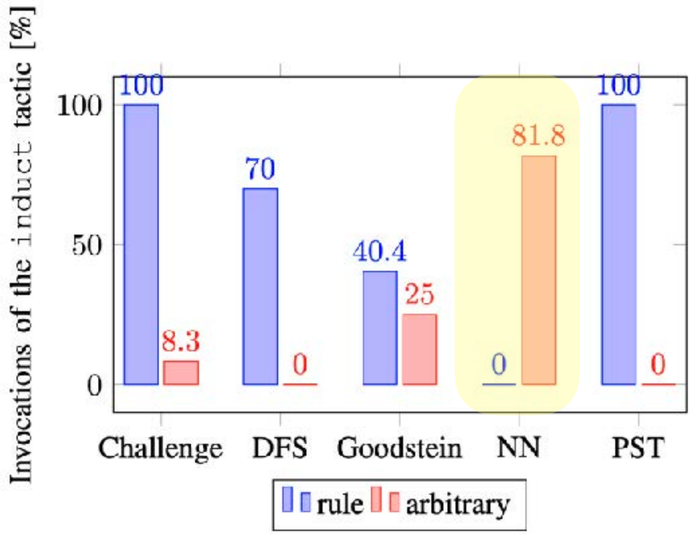


Is smart_induct Domain dependent?

smart_induct is bad at generalisation heuristics.

Coincidence rates

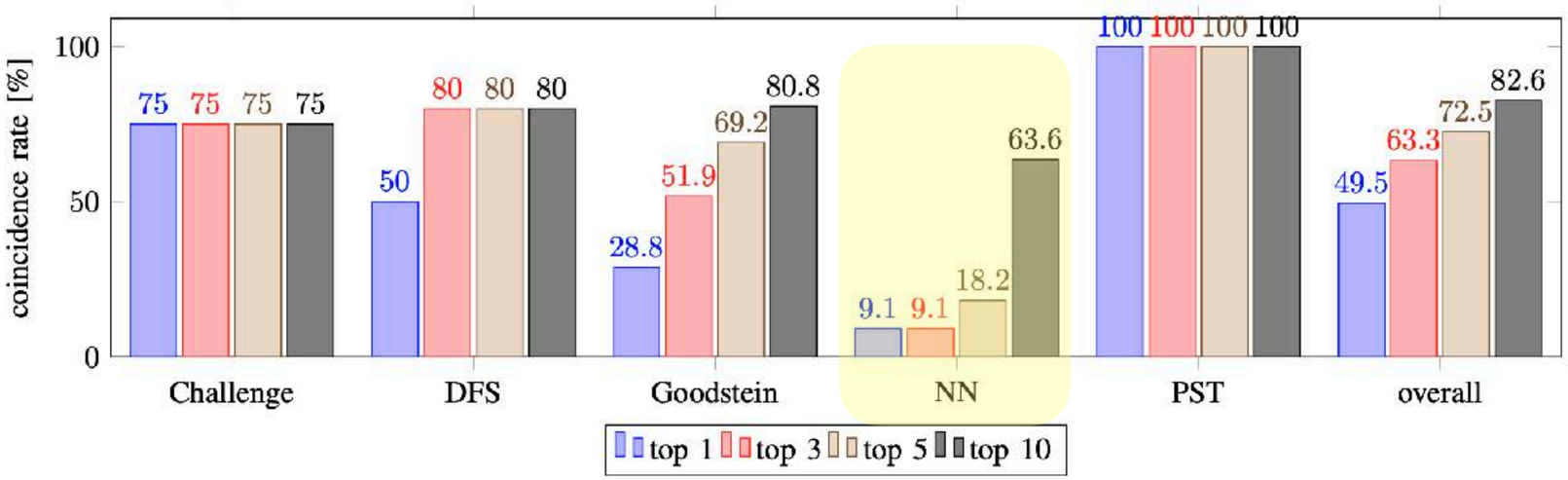


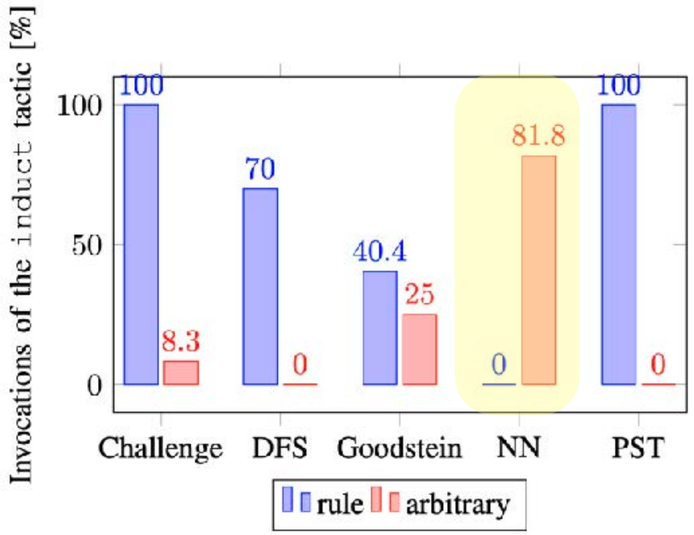


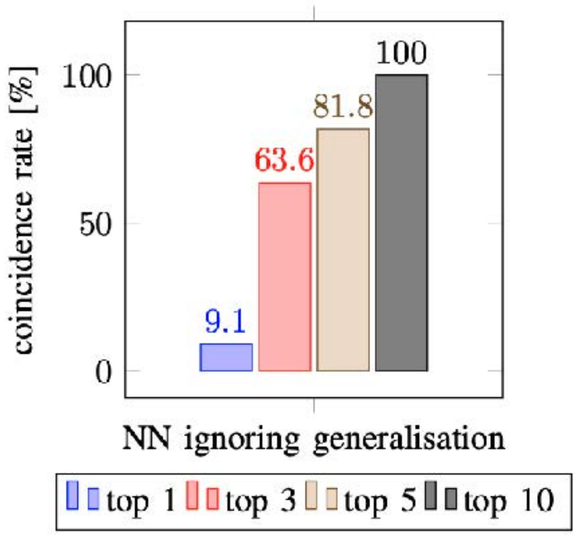
Is smart_induct Domain dependent?

smart_induct is bad at generalisation heuristics.

Coincidence rates







Is smart_induct Domain dependent?

smart_induct is bad at generalisation heuristics.

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. 😱

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. 😱

5: can be quite slow. 😱

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. 😱

... because of LiFtEr

5: can be quite slow. 😱

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. 😱

... because of LiFtEr

5: can be quite slow. 😱

... because of too many branches

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. 😱

... because of LiFtEr → we need semantic awareness!

5: can be quite slow. 😱

... because of too many branches

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

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

Output | Query | Sledgehammer | Symbols

15,13 (351/361) Input/output complete (isabelle,isabelle,UTF-8-Isabelle) 94/512MB 1 error(s) 1:47 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 ys rule: rev2.induct)
  apply auto done
```

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

```
proof (prove)
goal:
```

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.

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

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

fun rev2::"'a list ⇒ 'a list ⇒ 'a lis
  "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 state

```
proof (prove)
goal:
```

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

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

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

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

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

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

Output Query Sledgehammer Symbols

15,13 (344/353)

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

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

fun rev2::"'a list ⇒ 'a list ⇒ 'a lis
  "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 state

```
proof (prove)
goal:
```

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

15,1

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

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

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

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

generalisation!

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

Output Query Sledgehammer Symbols

15,13 (344/353)

It’s hard to write induction heuristics.

① blue = what I can see before releasing smart_induct

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

experts’ proof documents =
[(proof goal,

good induction)]

experts

new users

proof goal

candidate induction

on: [is1, s, stk]
arbitrary: [ ]
rule: [exec.induct]

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’s hard to write induction heuristics.

① blue = what I can see before releasing smart_induct

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

experts’ proof documents =
[(proof goal,
relevant definitions,
good induction)]

experts

new users

proof goal

candidate induction
on: [is1, s, stk]
arbitrary: [ ]
rule: [exec.induct]

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

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-lsabelle) 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)

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

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>.

↓ for semantic analysis

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

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>.

↓ 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])
```

∃ over definitions

← in some defining clause of

← rev2, check if

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>.

↓ 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])
```

∃ 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

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

↓ 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

∃ 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

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

↓ 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

↓ 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:

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. 😱

... because of LiFtEr → SeLFiE

5: can be quite slow. 😱

... because of too many branches

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. 😱

... because of LiFtEr → SeLFiE

5: can be quite slow. 😱

... because of too many branches

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. 😱

... because of LiFtEr → SeLFiE

5: can be quite slow. 😱

... because of too many branches

→ fewer branches

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. 😱

... because of LiFtEr → SeLFiE

5: can be quite slow. 😱

... because of too many branches

→ fewer branches

(1) two step candidate construction
(2) syntax directed candidate construction

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. 😱

... because of LiFtEr → SeLFiE

5: can be quite slow. 😱

... because of too many branches

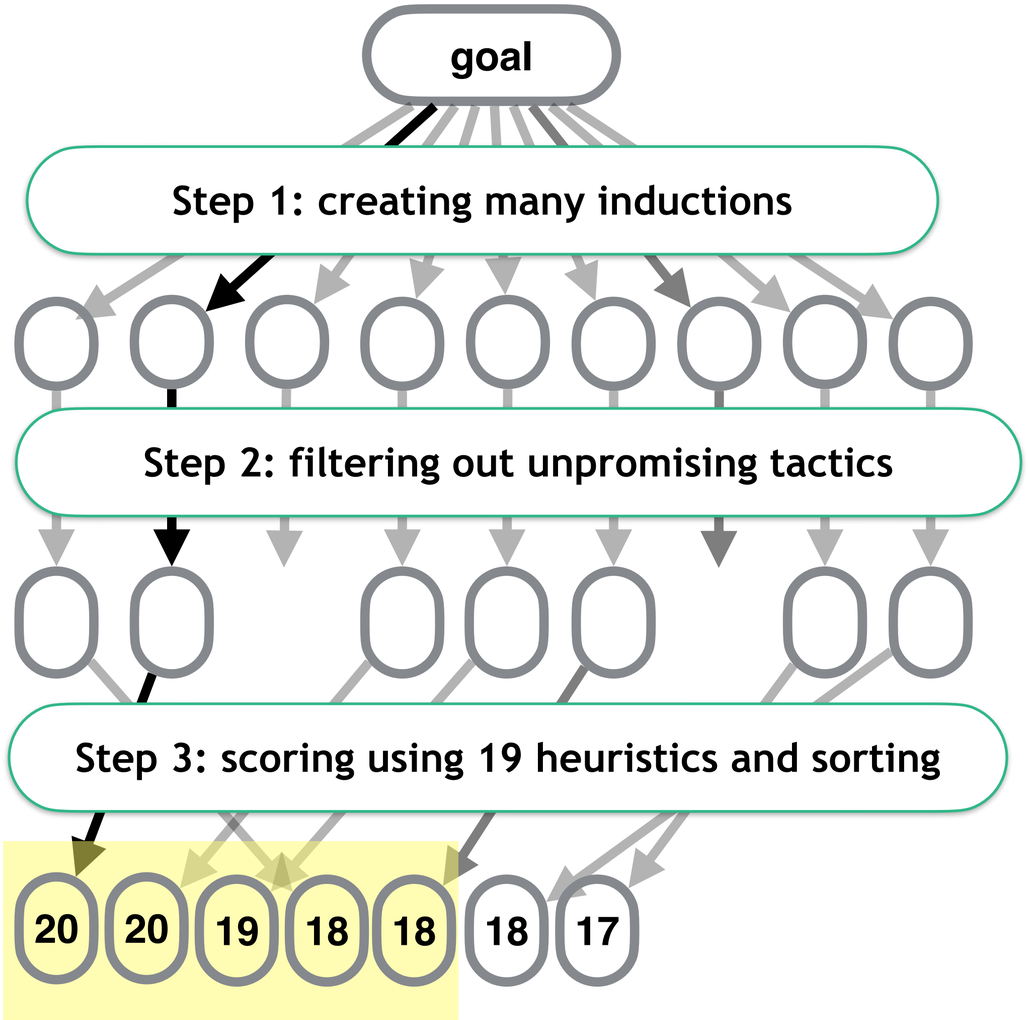
→ fewer branches

(1) two step candidate construction

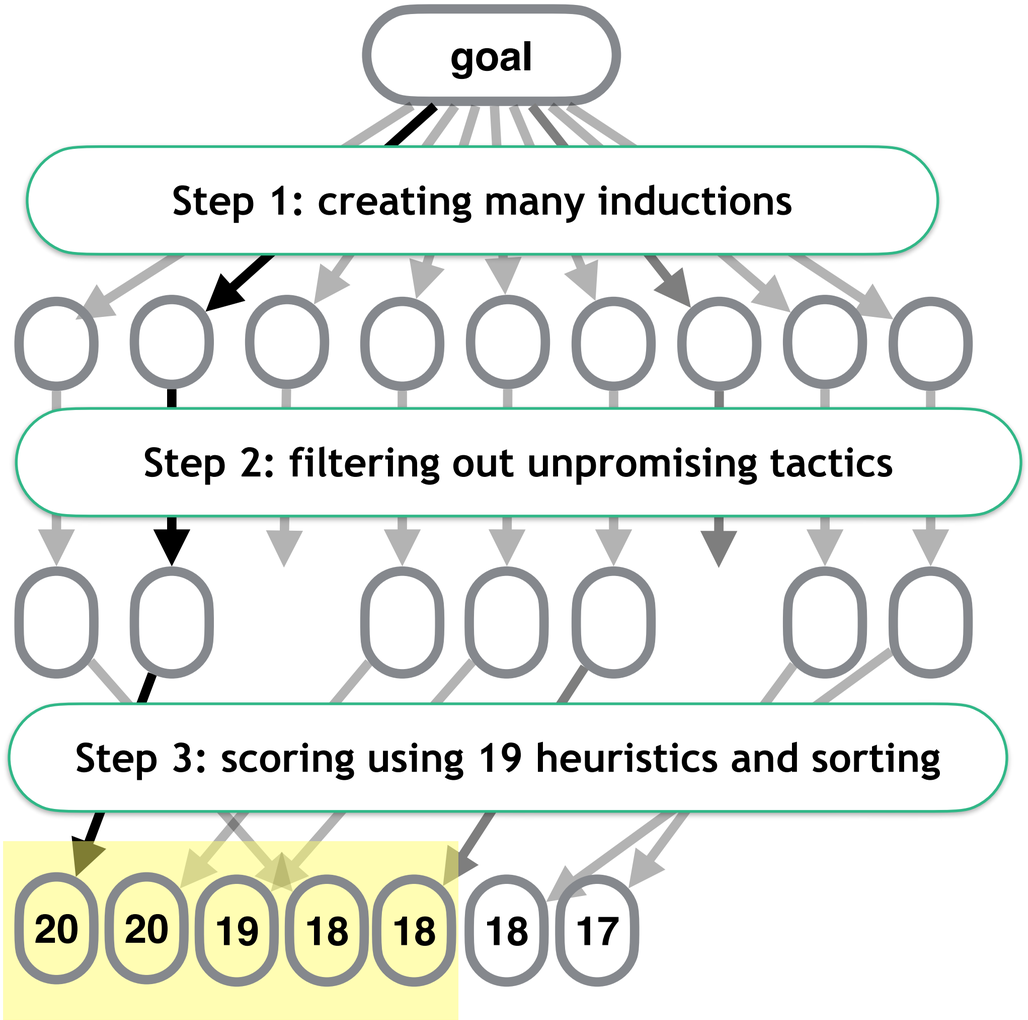
(2) syntax directed candidate construction 😵

smart_induct (based on LiFtEr)

sem_ind (based on SeLFiE)

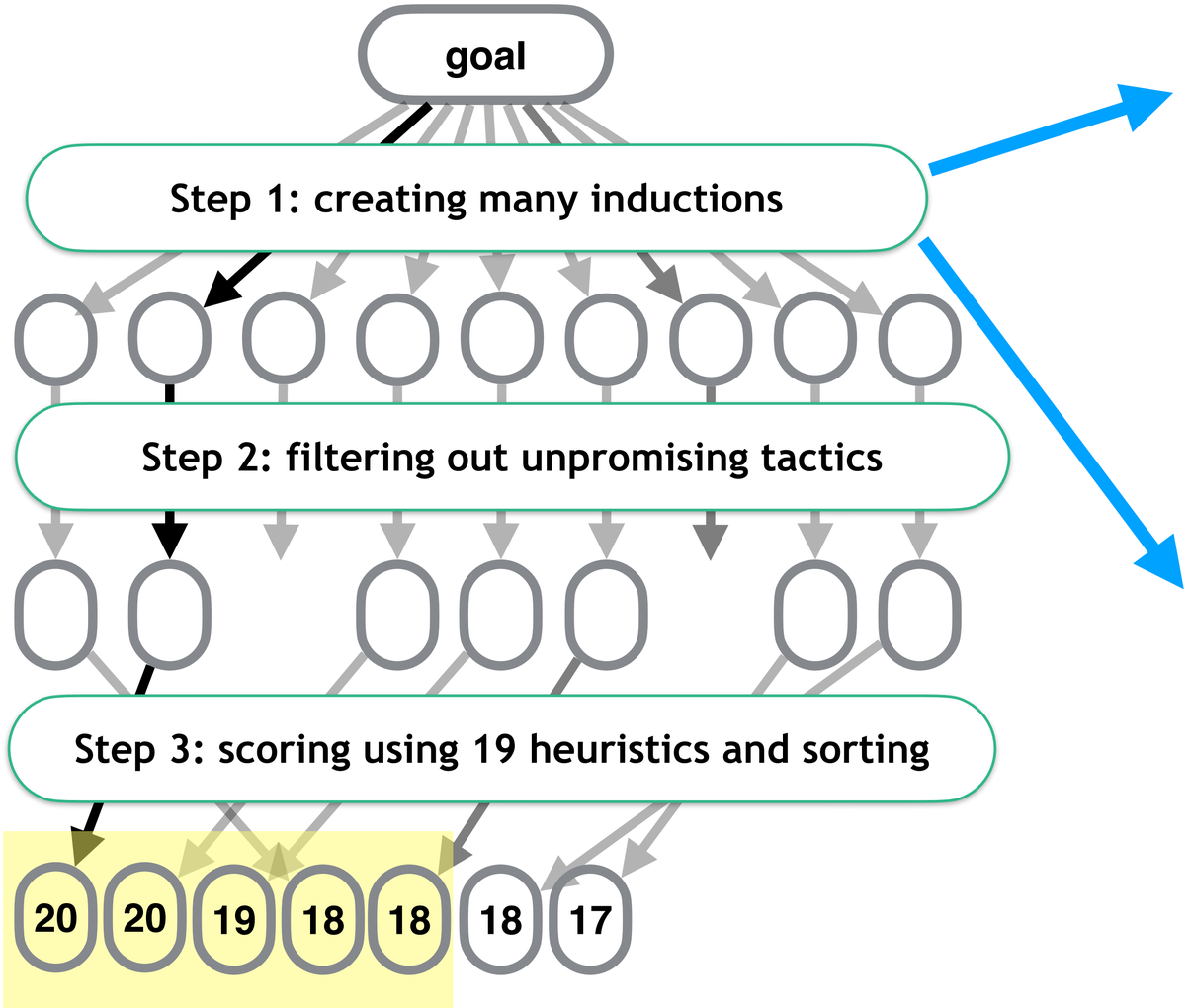


smart_induct (based on LiFtEr)

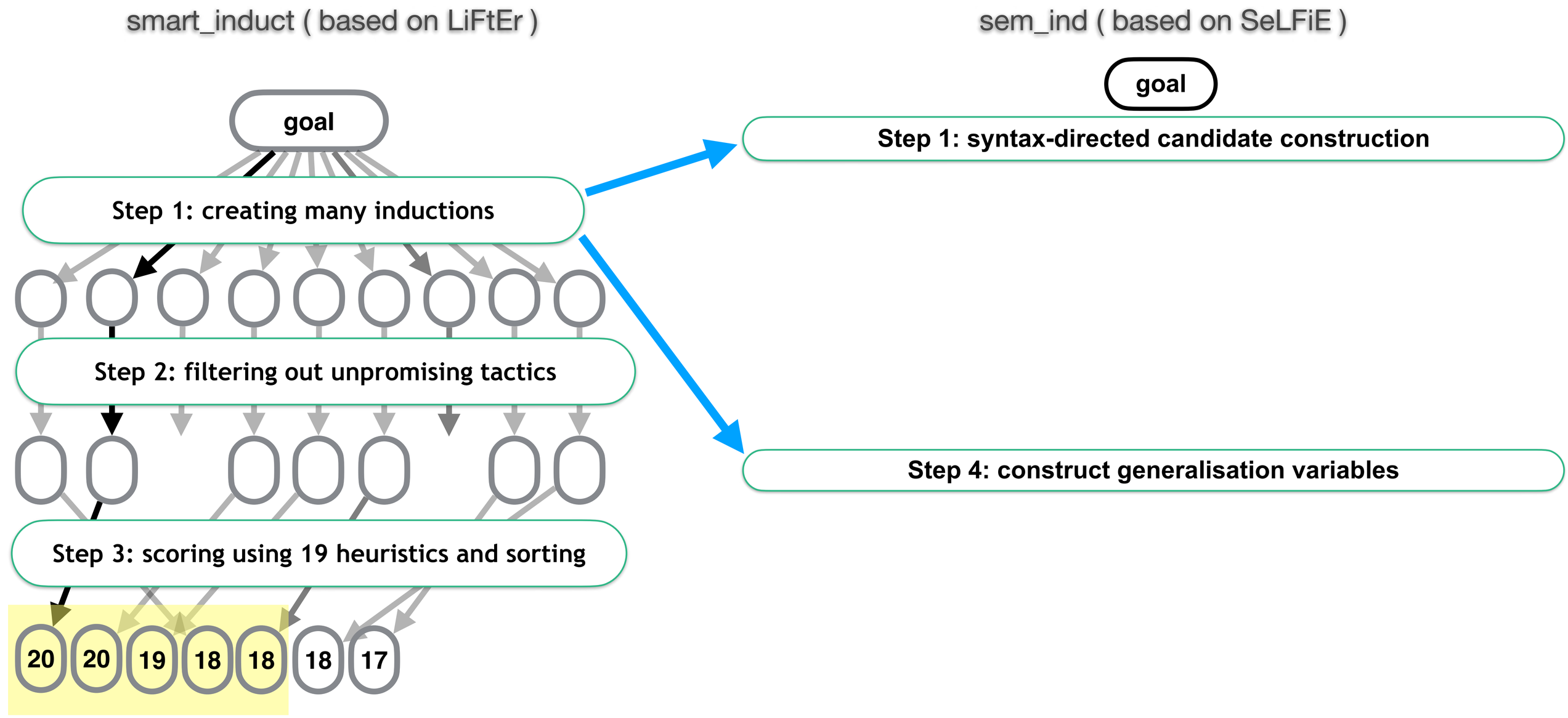


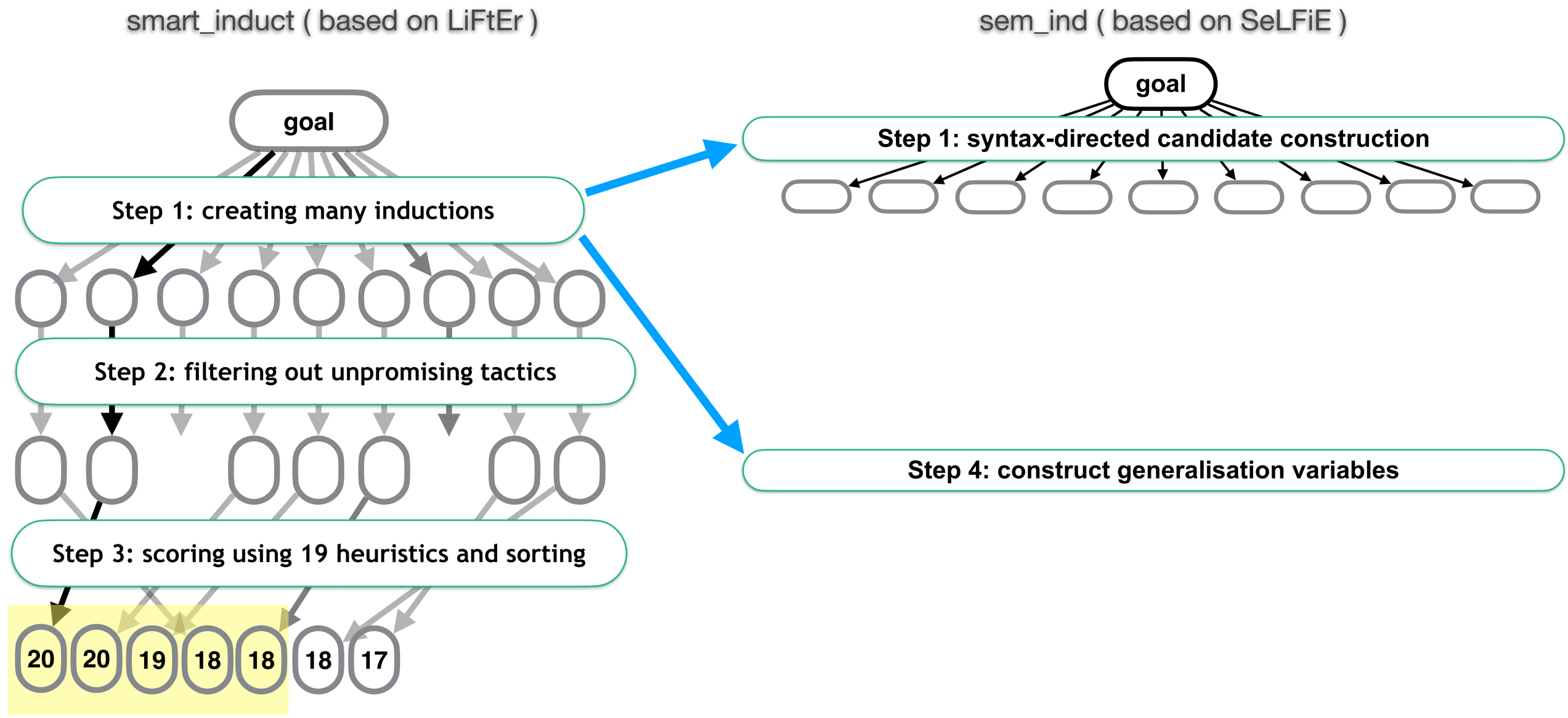
sem_ind (based on SeLFiE)

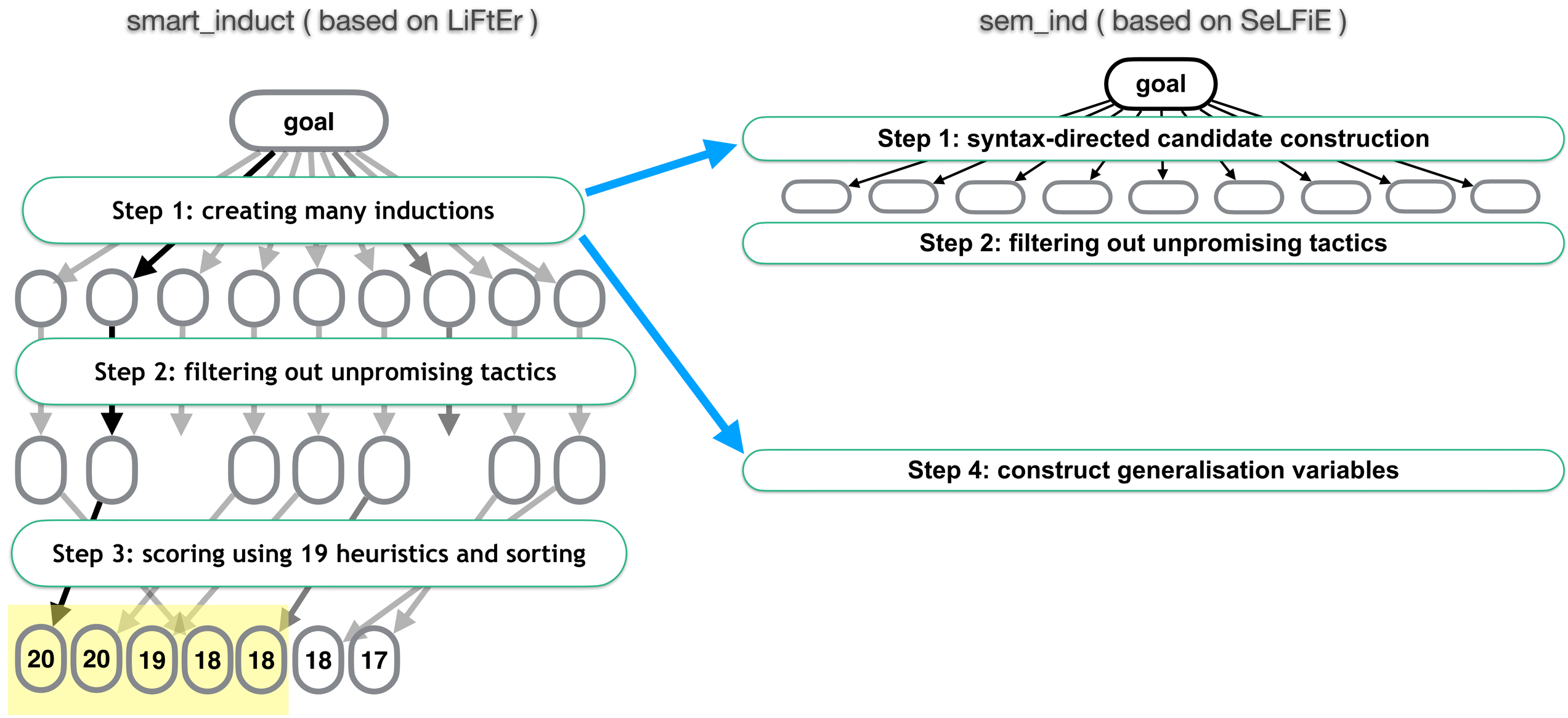
smart_induct (based on LiFtEr)

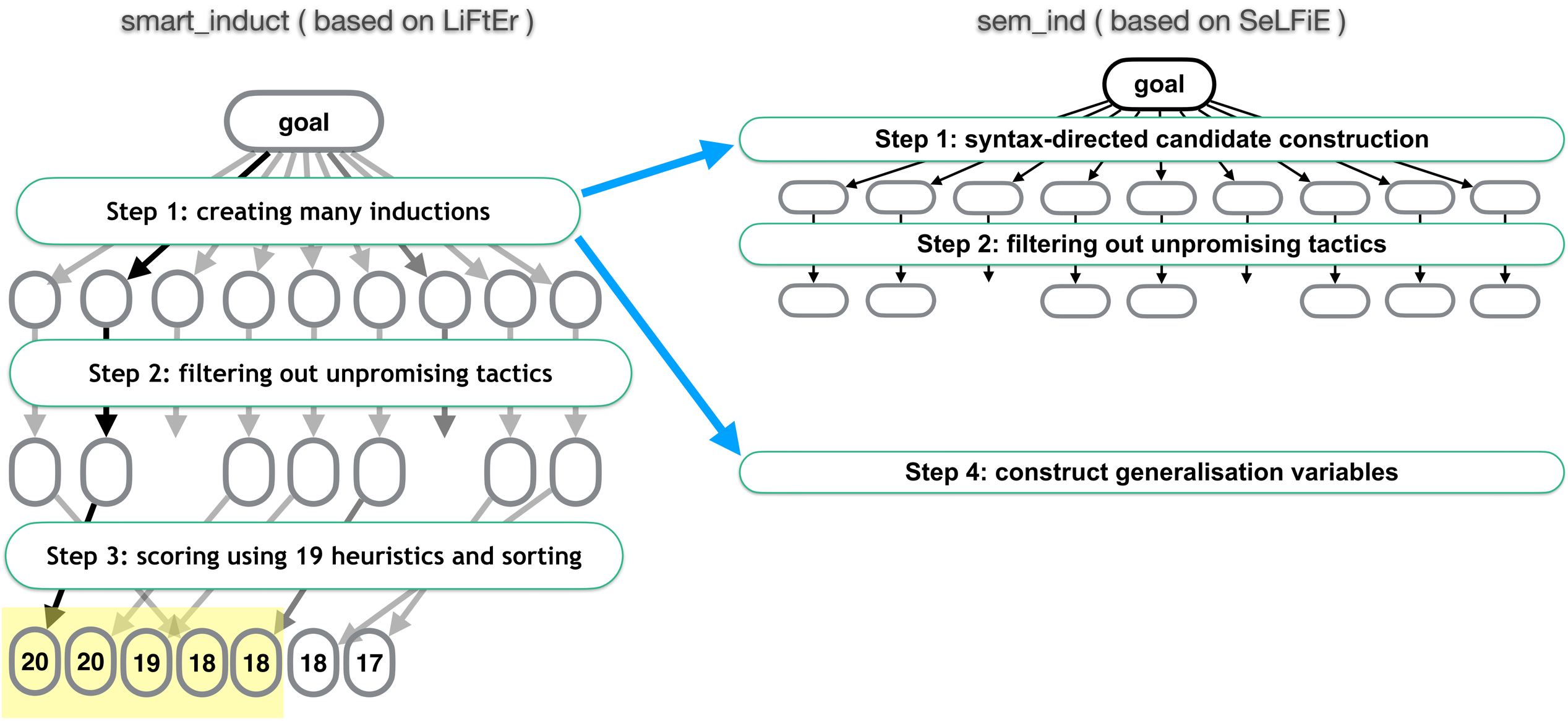


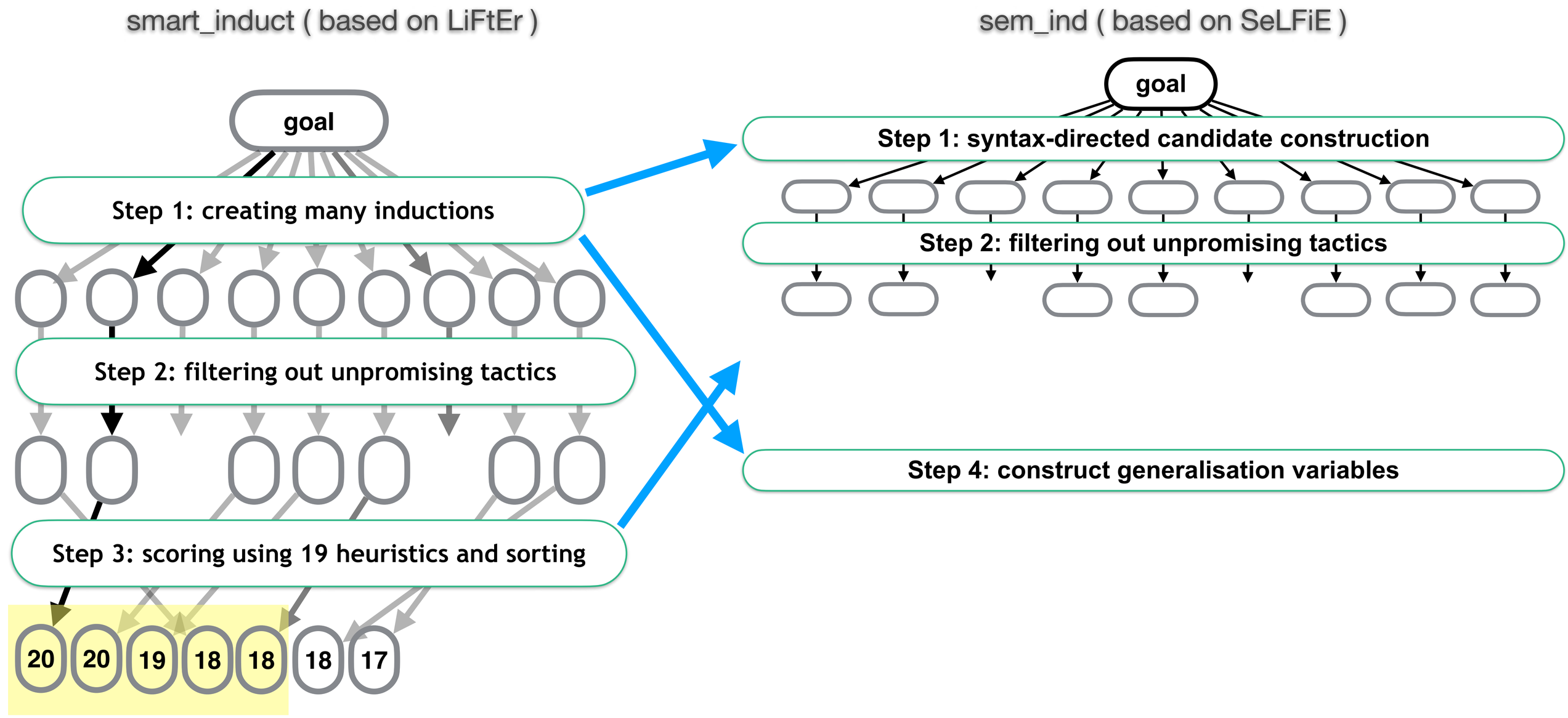
sem_ind (based on SeLFiE)

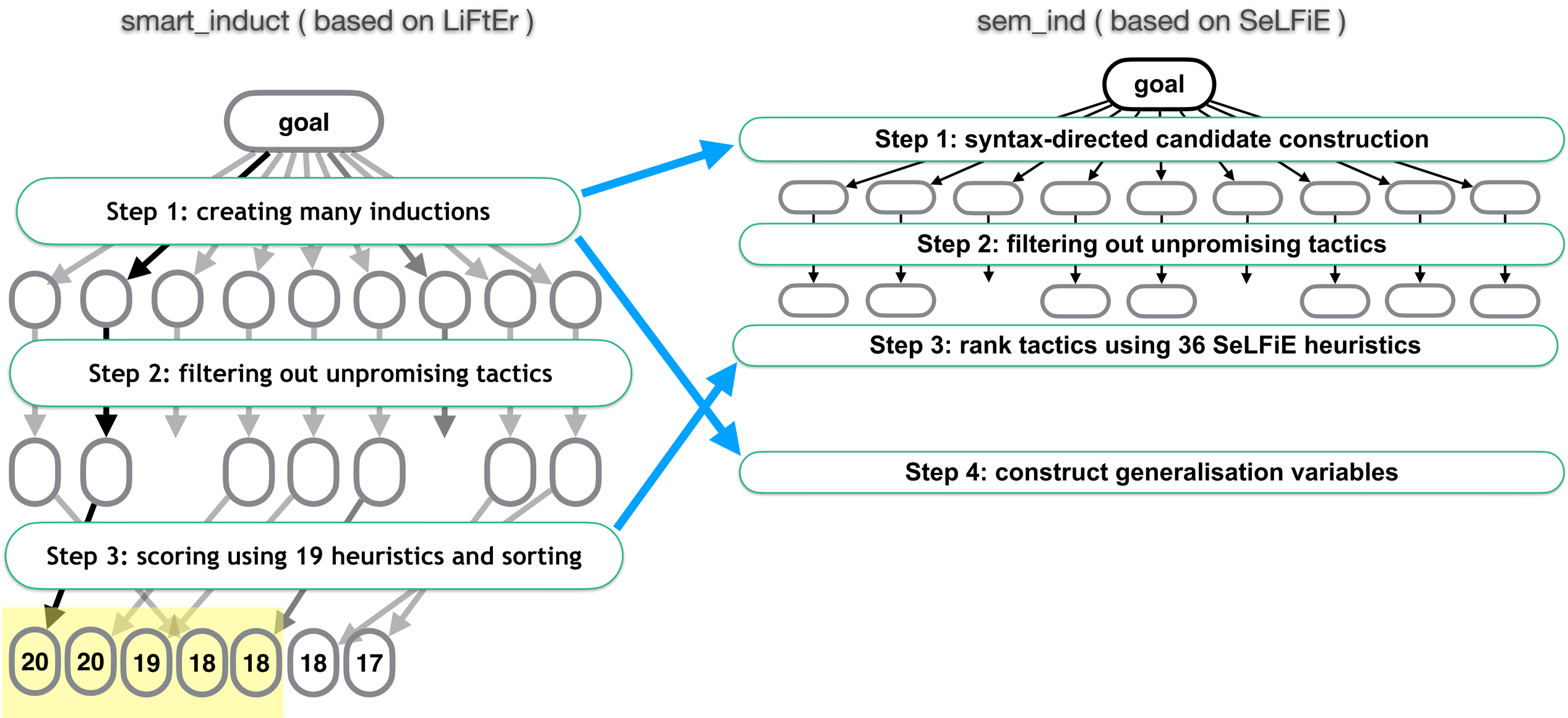

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 4: construct generalisation variables

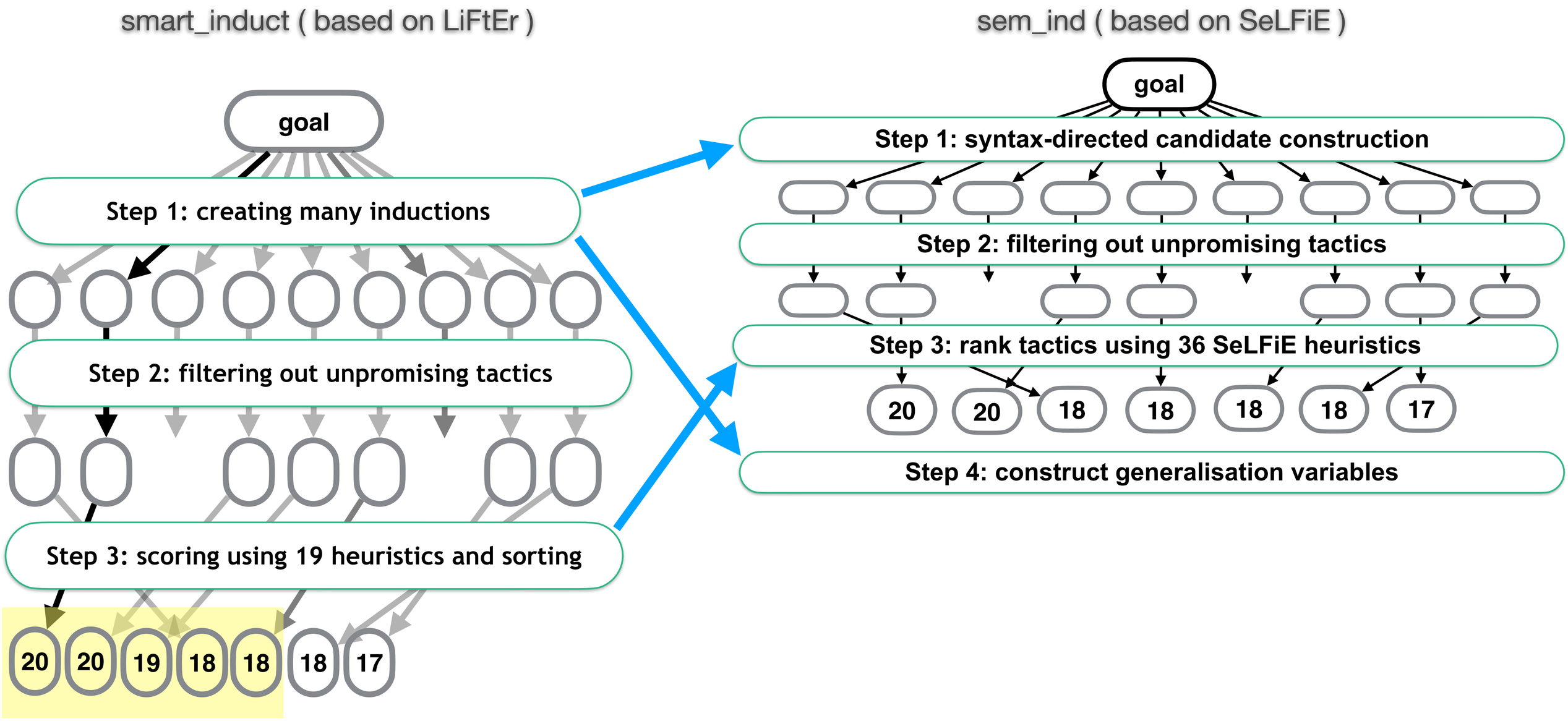

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 4: construct generalisation variables

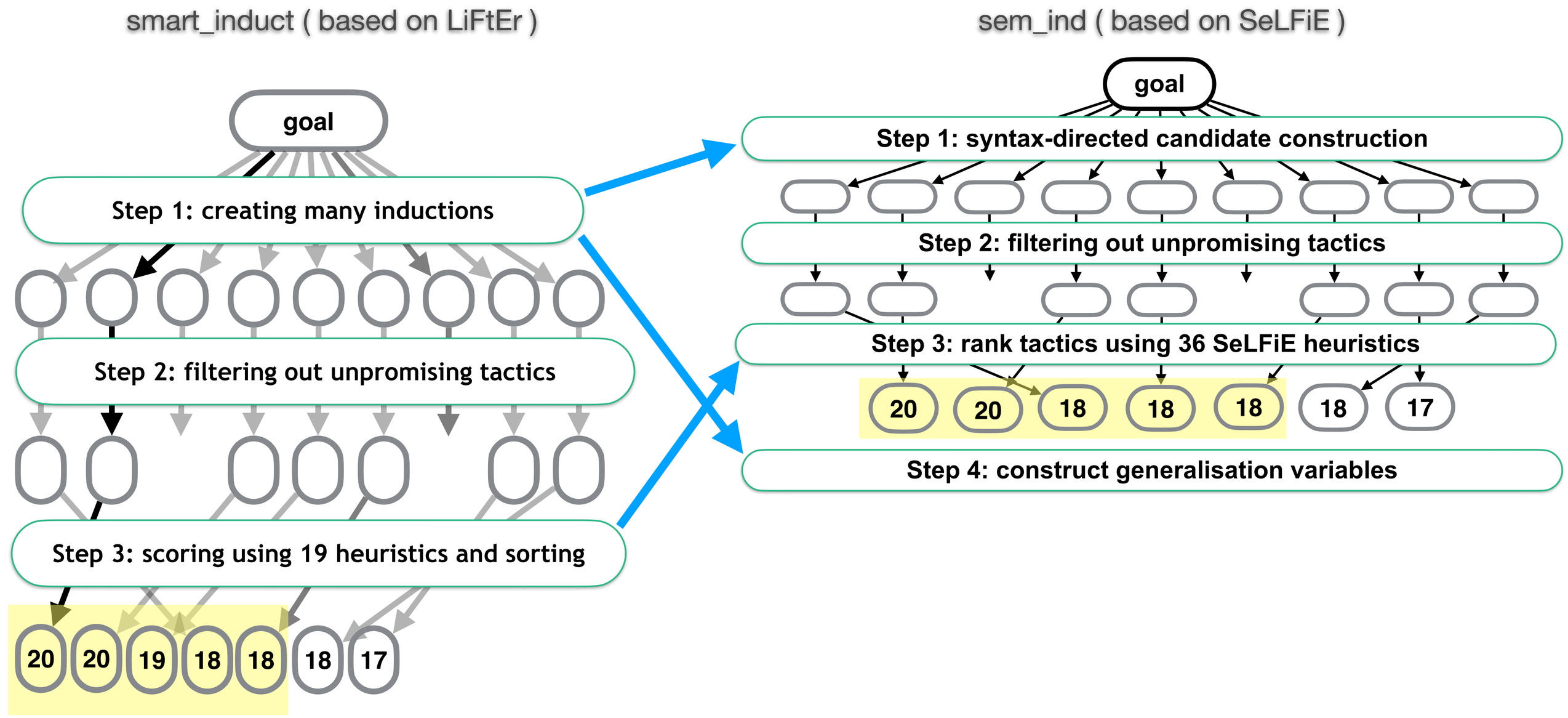

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step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

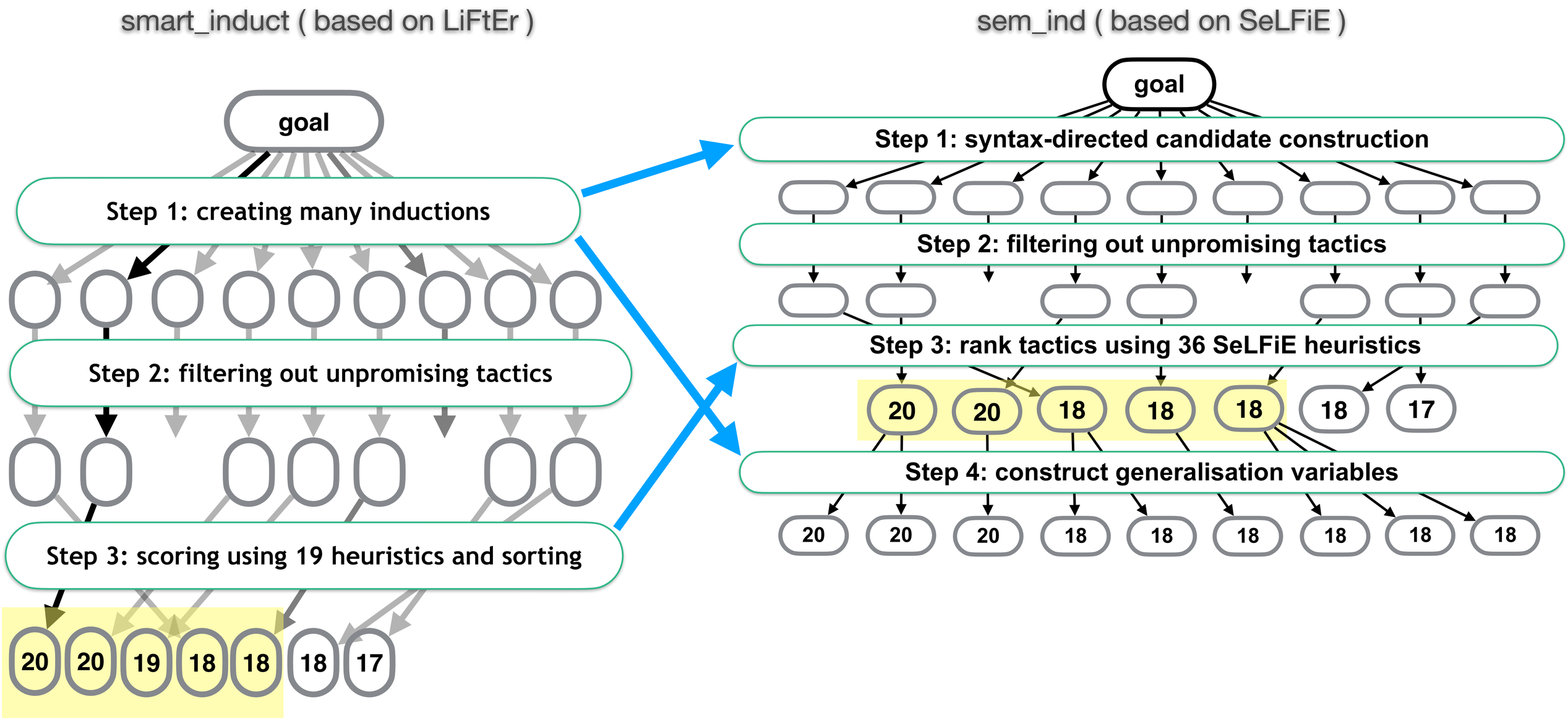

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step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

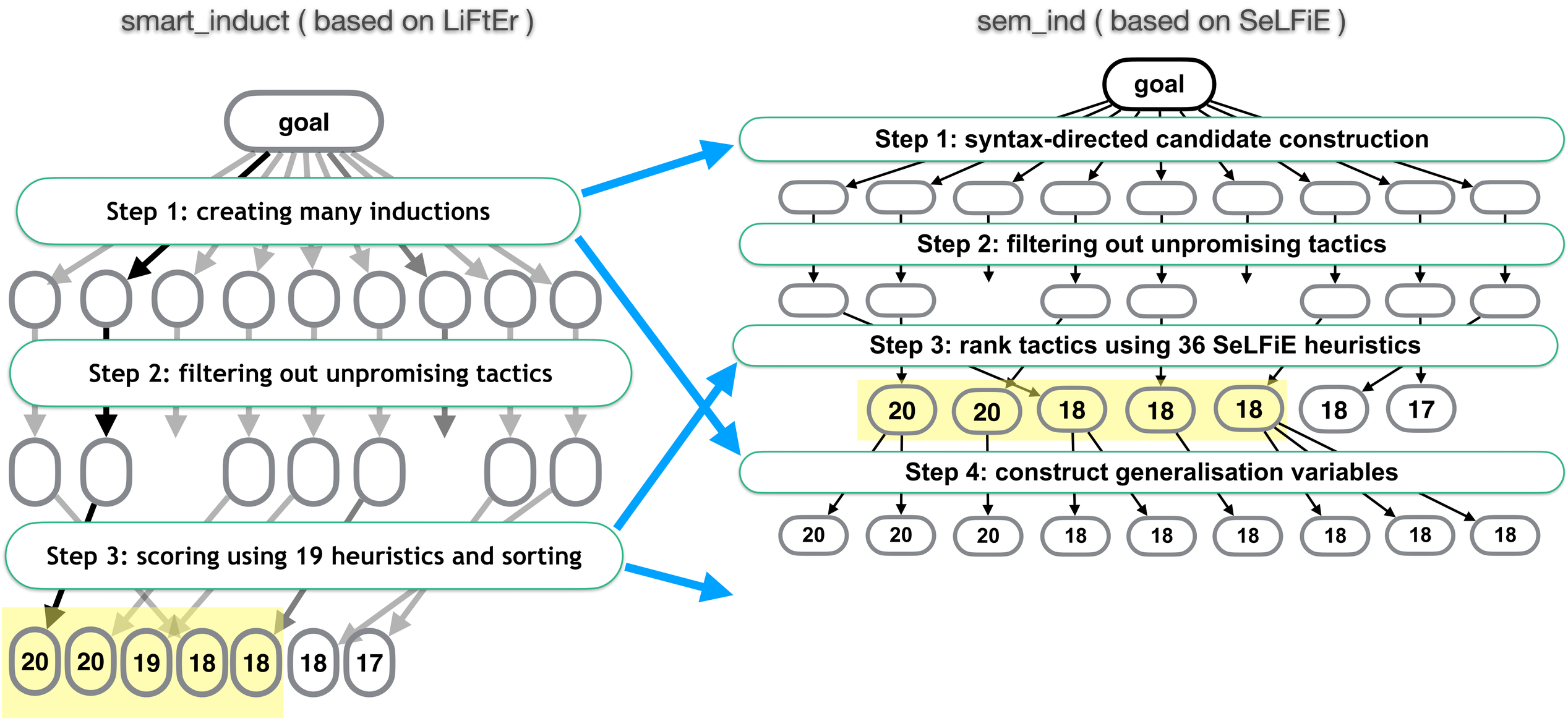

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step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

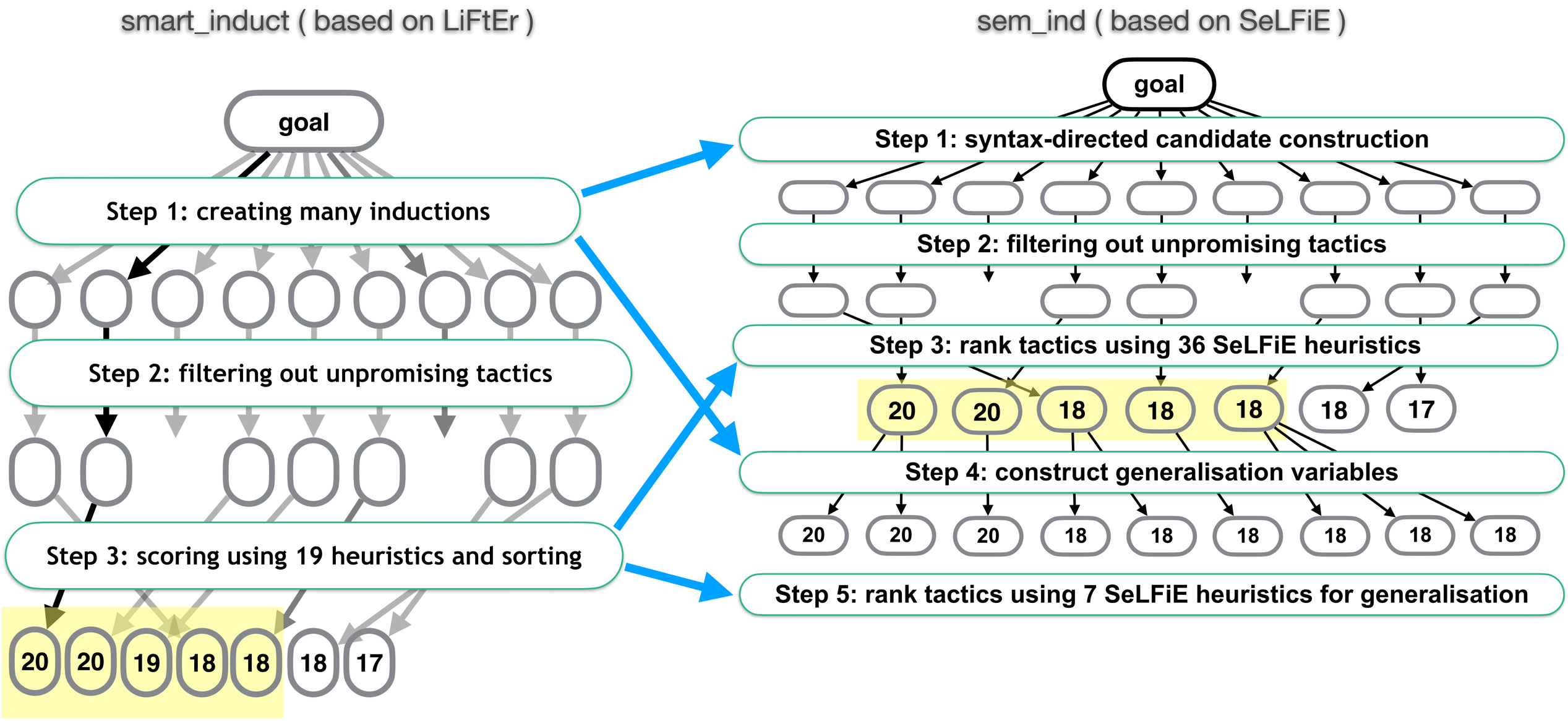

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step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

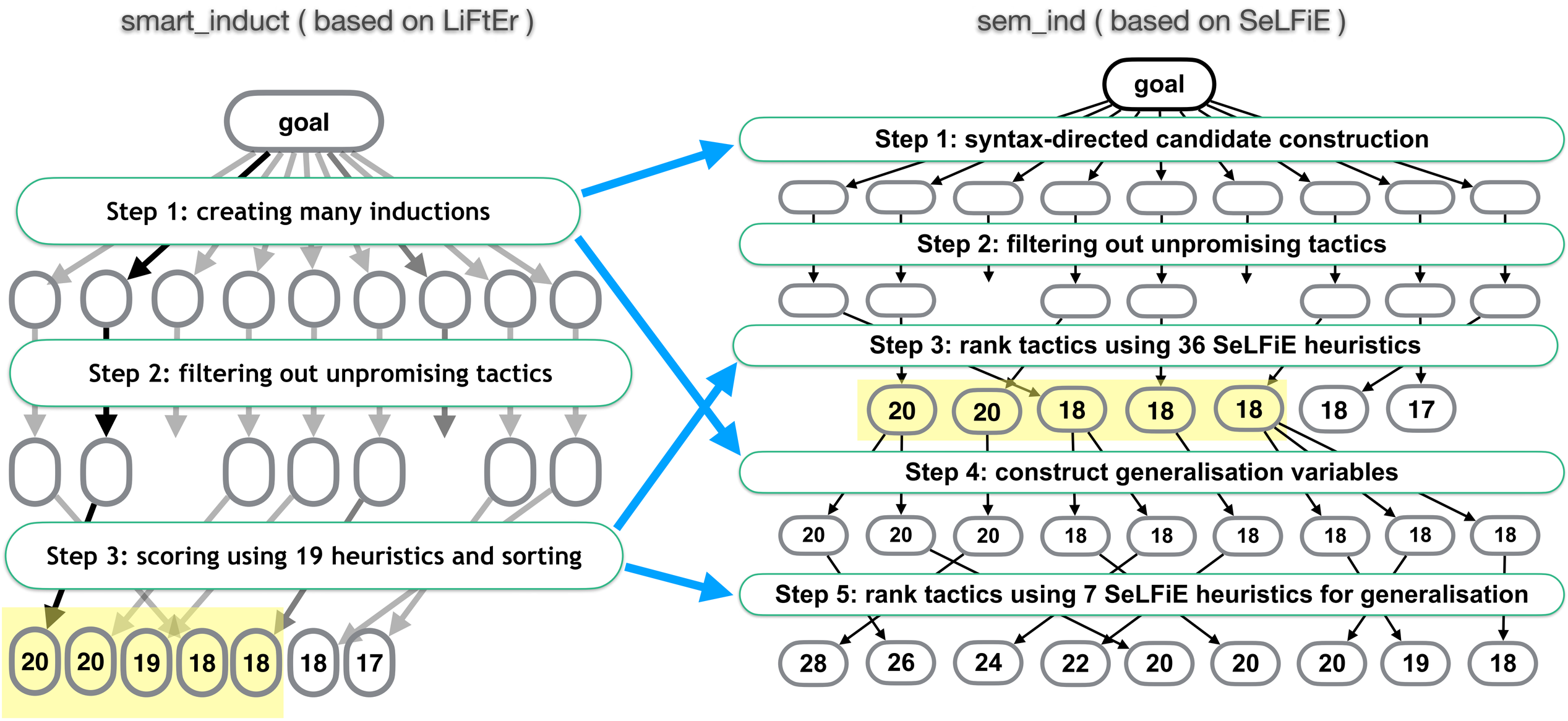

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step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

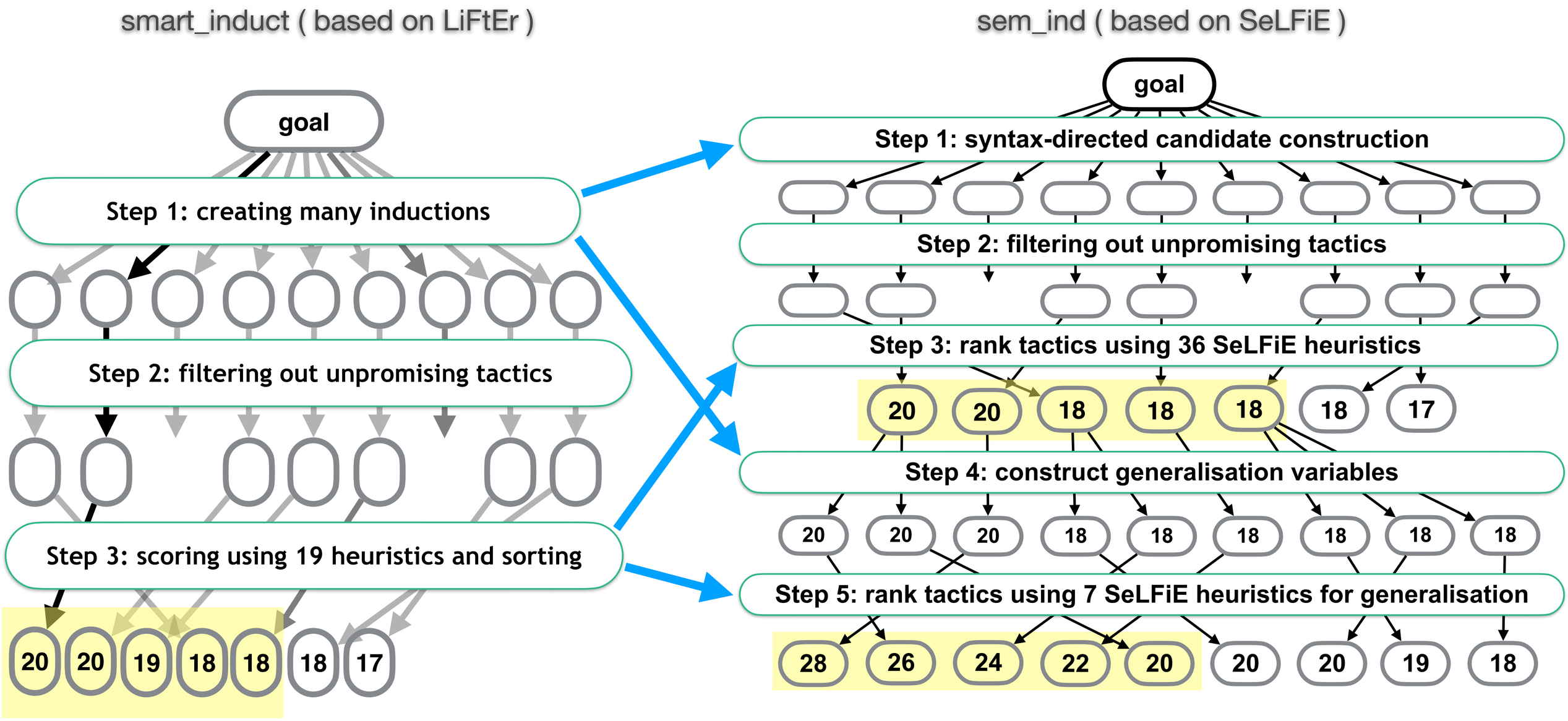

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step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

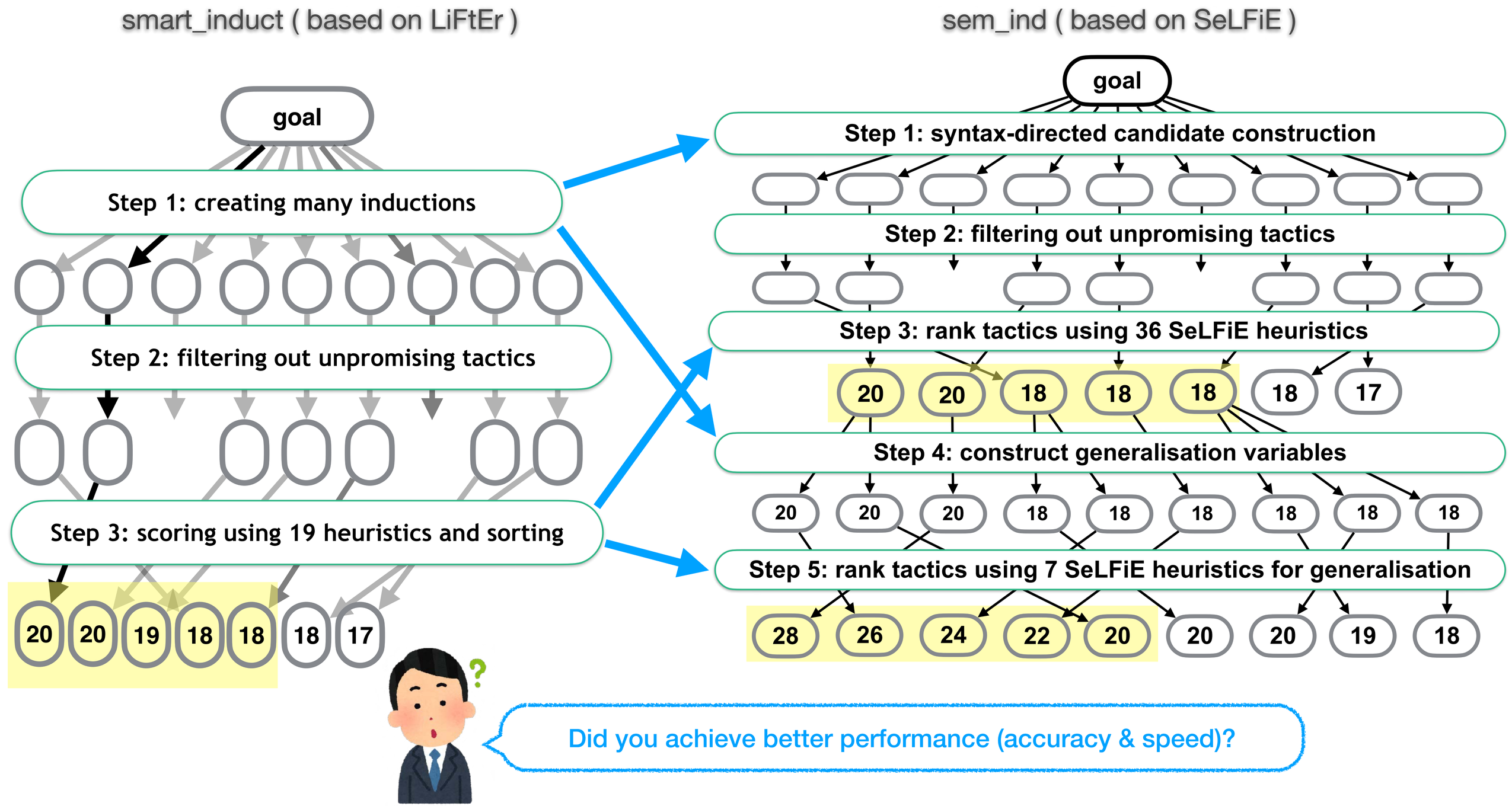

smart_induct (based on LiFtEr)
goal
Step 1: creating many inductions
Step 2: filtering out unpromising tactics
Step 3: scoring using 19 heuristics and sorting
20
20
19
18
18
18
17
sem_ind (based on SeLFiE)
goal
Step 1: syntax-directed candidate construction
Step 2: filtering out unpromising tactics
Step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

Return rates (speed)
based on 1095 inductive problems from the Archive of Formal Proofs

tool	0.2s	0.5s	1.0s	2.0s	5.0s
smart	0.0%	3.5%	16.9%	38.3%	70.2%

Return rates (speed)
based on 1095 inductive problems from the Archive of Formal Proofs

tool	0.2s	0.5s	1.0s	2.0s	5.0s
😱 `smart`	0.0%	3.5%	16.9%	38.3%	70.2%

Return rates (speed)
based on 1095 inductive problems from the Archive of Formal Proofs

tool	0.2s	0.5s	1.0s	2.0s	5.0s
sem_ind	8.8%	24.7%	47.8%	69.8%	86.8%
😱 smart	0.0%	3.5%	16.9%	38.3%	70.2%

Return rates (speed)
based on 1095 inductive problems from the Archive of Formal Proofs

tool	0.2s	0.5s	1.0s	2.0s	5.0s
😄 sem_ind	8.8%	24.7%	47.8%	69.8%	86.8%
😱 smart	0.0%	3.5%	16.9%	38.3%	70.2%

smart_induct

1: does not rely on a proof search. 😄

2: is domain-agnostic. 😄

3: can predict induction terms. 😄

4: cannot predict variable generalisation. 😱

... because of LiFtEr

5: can be quite slow. 😱

... because of too many branches

smart_induct → sem_ind

1: does not rely on a proof search. 😄

2: is domain-agnostic. 😄

3: can predict induction terms. 😄

4: cannot predict variable generalisation. 😱

... because of LiFtEr

5: can be quite slow. 😱

... because of too many branches

smart_induct → sem_ind

1: does not rely on a proof search. 😄

2: is domain-agnostic. 😄

3: can predict induction terms. 😄

4: cannot predict variable generalisation. 😱 → 😄

... because of LiFtEr → SeLFiE

5: can be quite slow. 😱

... because of too many branches

smart_induct → sem_ind

1: does not rely on a proof search. 😄

2: is domain-agnostic. 😄

3: can predict induction terms. 😄

4: cannot predict variable generalisation. 😱 → 😄

... because of LiFtEr → SeLFiE

5: can be quite slow. 😱 → 😄

... because of too many branches → fewer branches

smart_induct → sem_ind

1: does not rely on a proof search. 😄

2: is domain-agnostic. 😄

3: can predict induction terms. 😄

4: cannot predict variable generalisation. 😱 → 😄

... because of LiFtEr → SeLFiE

5: can be quite slow. 😱 → 😄

... because of too many branches → fewer branches

Conjecturing beyond variable generalisation?

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

```
5  primrec rev1::"'a list ⇒ 'a list" where
6    "rev1  []       = []"
7  | "rev1 (x # xs) = rev1 xs @ [x]"
8
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 theorem "rev2 xs ys = rev1 xs @ ys"
14   apply(induct xs arbitrary: ys)
15   apply auto done
```

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
```

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

```
proof (
goal:
No subg
```

Why should I present the equivalence of two reverse functions like this?

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
```

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

```
proof (
goal:
No subc
```

Why should I present the equivalence of two reverse functions like this?

A more intuitive presentation would be…

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" w
  "rev1  []       = []"
| "rev1 (x # xs) = rev1 xs @ [x]"

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

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

```
proof (
goal:
No subg
```

Why should I present the equivalence of two reverse functions like this?

A more intuitive presentation would be…

Output Query Sledgehammer Symbols

15,13 (344/353)

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

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

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

lemma "rev2 xs [] = rev1 xs"
  oops
```

Proof state

```
proof (prove)
goal (1 subgoal):
 1. rev2 xs [] = rev1 xs
```

Output Query Sledgehammer Symbols

13,30 (298/462)

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

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

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

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

```
proof (
goal:
No subg
```

Why should I present the equivalence of two reverse functions like this?

A more intuitive presentation would be…

Output Query Sledgehammer Symbols

15,13 (344/353)

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

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

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

lemma "rev2 xs [] = rev1 xs"
  oops
```

Proof state

```
proof (prove)
goal (1 subgoal):
```

Conjecturing beyond variable generalisation!

Output Query Sledgehammer Symbols

13,30 (298/462)

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

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

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

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

Proof (
goal:
No subg

Why should I present the equivalence of two reverse functions like this?

A more intuitive presentation would be…

Output Query Sledgehammer Symbols

15,13 (344/353)

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

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

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

lemma "rev2 xs [] = rev1 xs"
  oops
```

Proof state

proof (prove)
goal (1 subgoal):

Conjecturing beyond variable generalisation!

Output Query Sledgehammer Symbols

13,30 (298/462)

What are good conjectures?

What are good conjectures?

good = useful and provable

What are good conjectures?

good = useful and provable

$$\frac{\text{Conjecture} \to \text{Goal} \qquad \text{Conjecture}}{\text{Goal}}$$

What are good conjectures?

good = useful and provable

$$\frac{\text{Conjecture} \to \text{Goal} \qquad \text{Conjecture}}{\text{Goal}}$$

How do you identify such conjectures?

What are good conjectures?

good = useful and provable

$$\frac{\text{Conjecture} \rightarrow \text{Goal} \qquad \text{Conjecture}}{\text{Goal}}$$

How do you identify such conjectures?

using abductive-reasoning in PSL

What are good conjectures?

good = useful and provable

at least not easily refutable

$$\frac{\text{Conjecture} \rightarrow \text{Goal} \qquad \boxed{\text{Conjecture}}}{\text{Goal}}$$

How do you identify such conjectures?

using abductive-reasoning in PSL

What are good conjectures?

good = useful and provable

easier to prove

at least not easily refutable

$$\frac{\text{Conjecture} \rightarrow \text{Goal} \qquad \boxed{\text{Conjecture}}}{\text{Goal}}$$

How do you identify such conjectures?

using abductive-reasoning in PSL

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)"


lemma "rev2 xs [] = rev1 xs"
  oops
```

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

Output | Query | Sledgehammer | Symbols

15,7 (300/457) (isabelle,isabelle,UTF-8-Isabelle) 148/512MB 1 error(s) 5:22 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)"
strategy CDInd = Thens [Conjecture, Fastforce, Quickcheck,
                        Dynamic (Induct), Auto, IsSolved]
lemma "rev2 xs [] = rev1 xs"
  oops
```

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

Output | Query | Sledgehammer | Symbols

15,8 (416/573) Input/output complete (isabelle,isabelle,UTF-8-Isabelle) 332/512MB 1 error(s) 5:24 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)"
strategy CDInd = Thens [Conjecture, Fastforce, Quickcheck,
                        Dynamic (Induct), Auto, IsSolved]
lemma "rev2 xs [] = rev1 xs"
  find_proof 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...
Testing conjecture with Quickcheck-exhaustive...
```

Output | Query | Sledgehammer | Symbols

15,19 (427/584) (isabelle,isabelle,UTF-8-Isabelle) 57/512MB 1 error(s) 5:22 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)"
strategy CDInd = Thens [Conjecture, Fastforce, Quickcheck,
                        Dynamic (Induct), Auto, IsSolved]
lemma "rev2 xs [] = rev1 xs"
  find_proof CDInd
```

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

```
apply (subgoal_tac "⋀Nil. rev2 xs Nil = rev1 xs @ Nil")
apply fastforce
apply (induct xs)
apply auto
done
```

Output | Query | Sledgehammer | Symbols

15,19 (427/584) (isabelle,isabelle,UTF-8-Isabelle) 321/512MB 1 error(s) 5:21 PM

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.

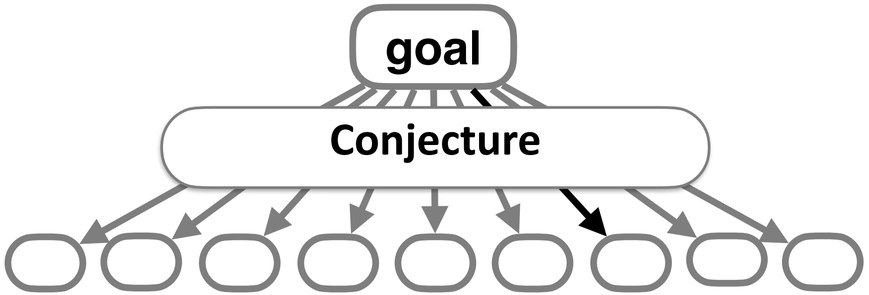
goal

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.

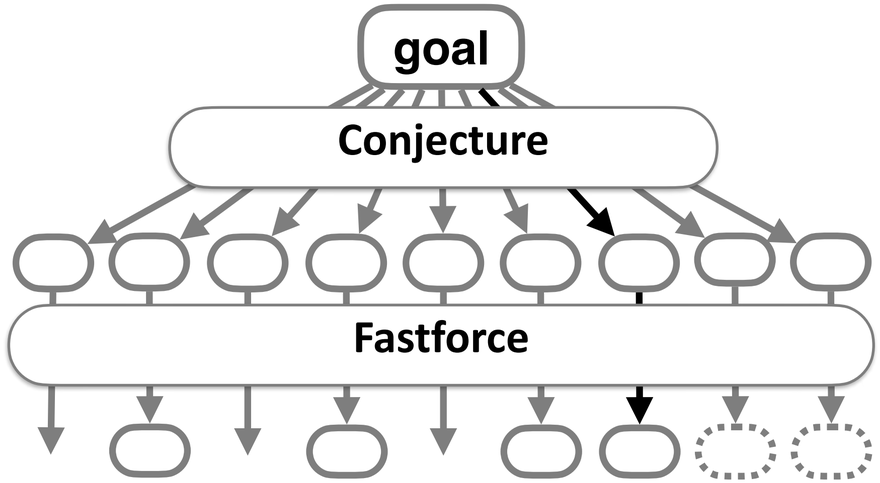


Algorithmic approaches work well, but not always → heuristics.

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.

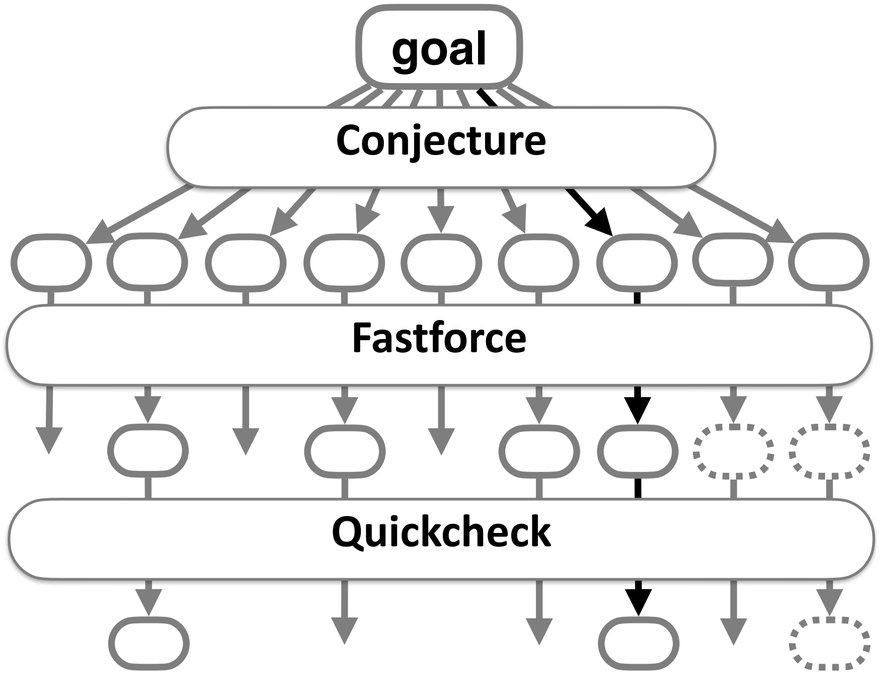


Algorithmic approaches work well, but not always → heuristics.

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.

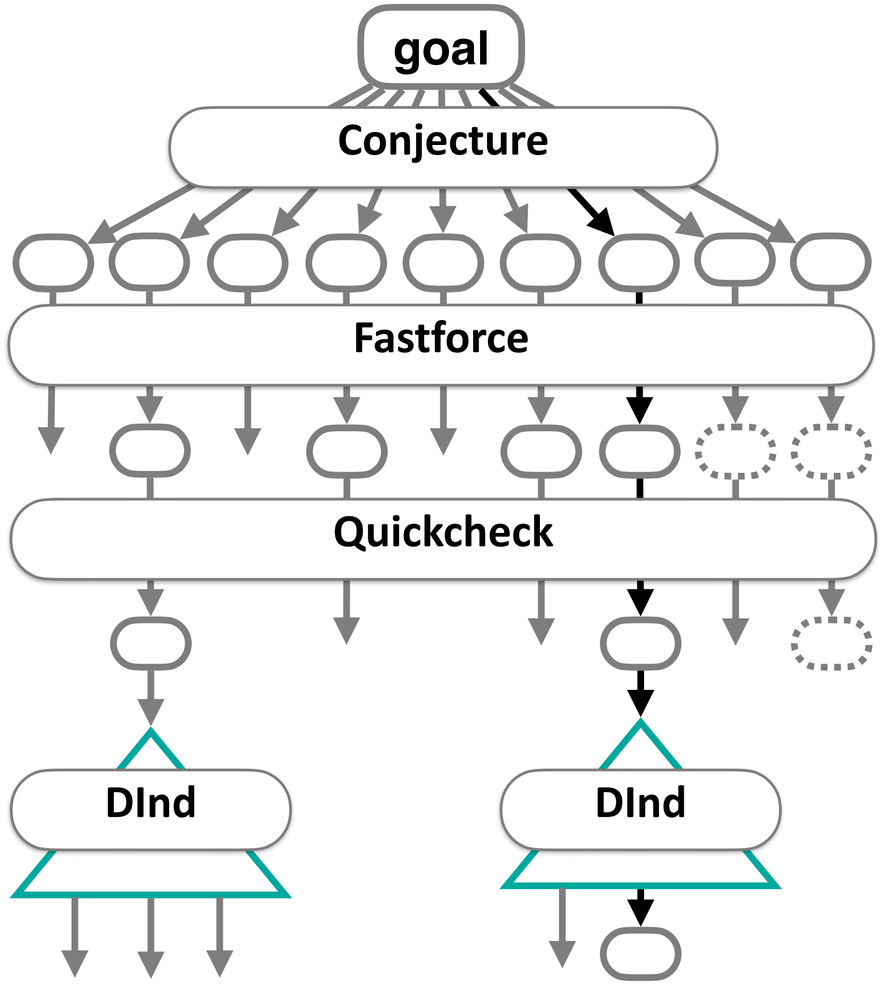


Algorithmic approaches work well, but not always → heuristics.

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.

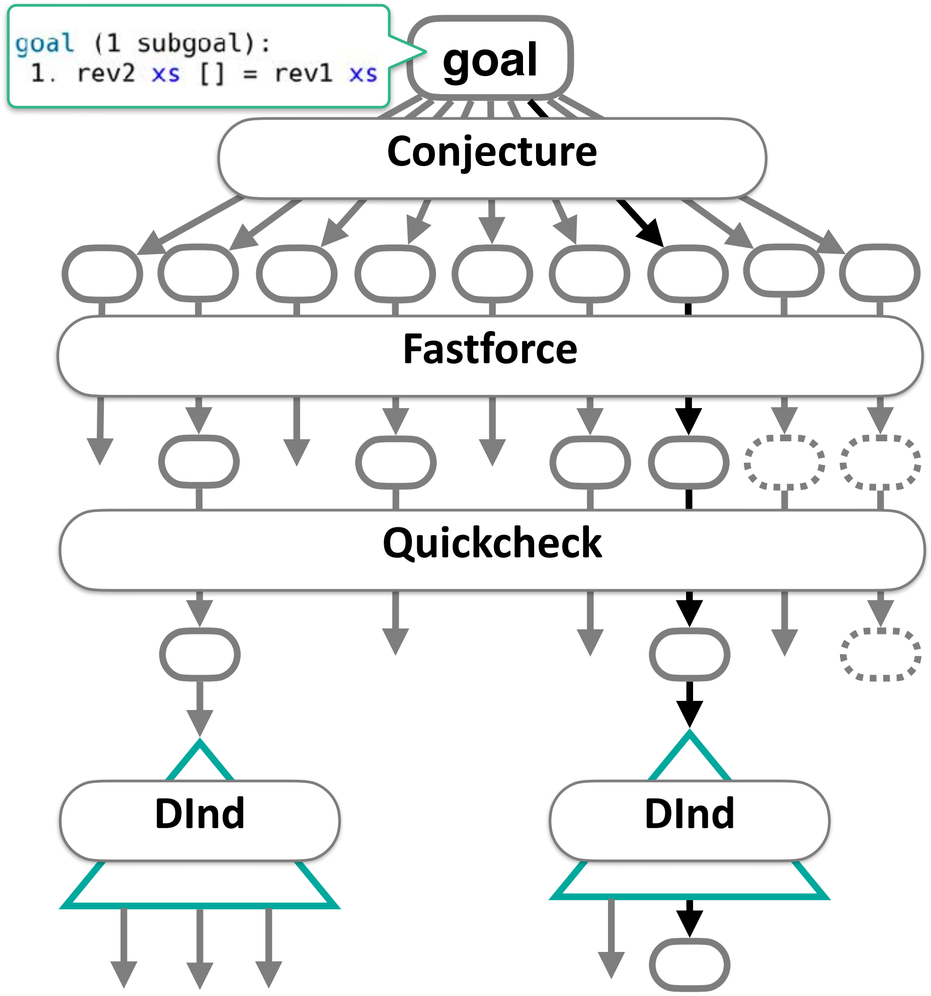


Algorithmic approaches work well, but not always → heuristics.

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.

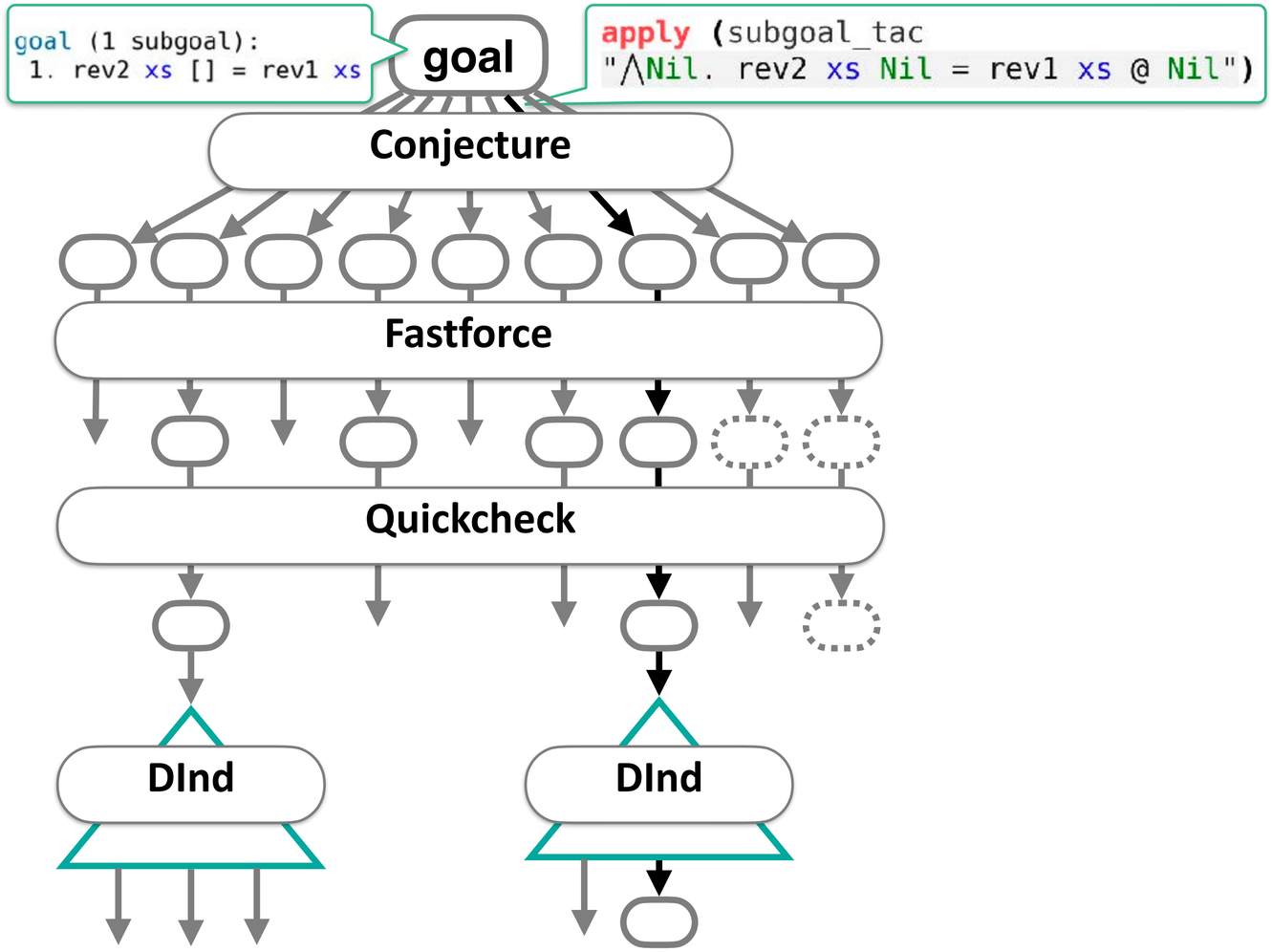


Algorithmic approaches work well, but not always → heuristics.

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.

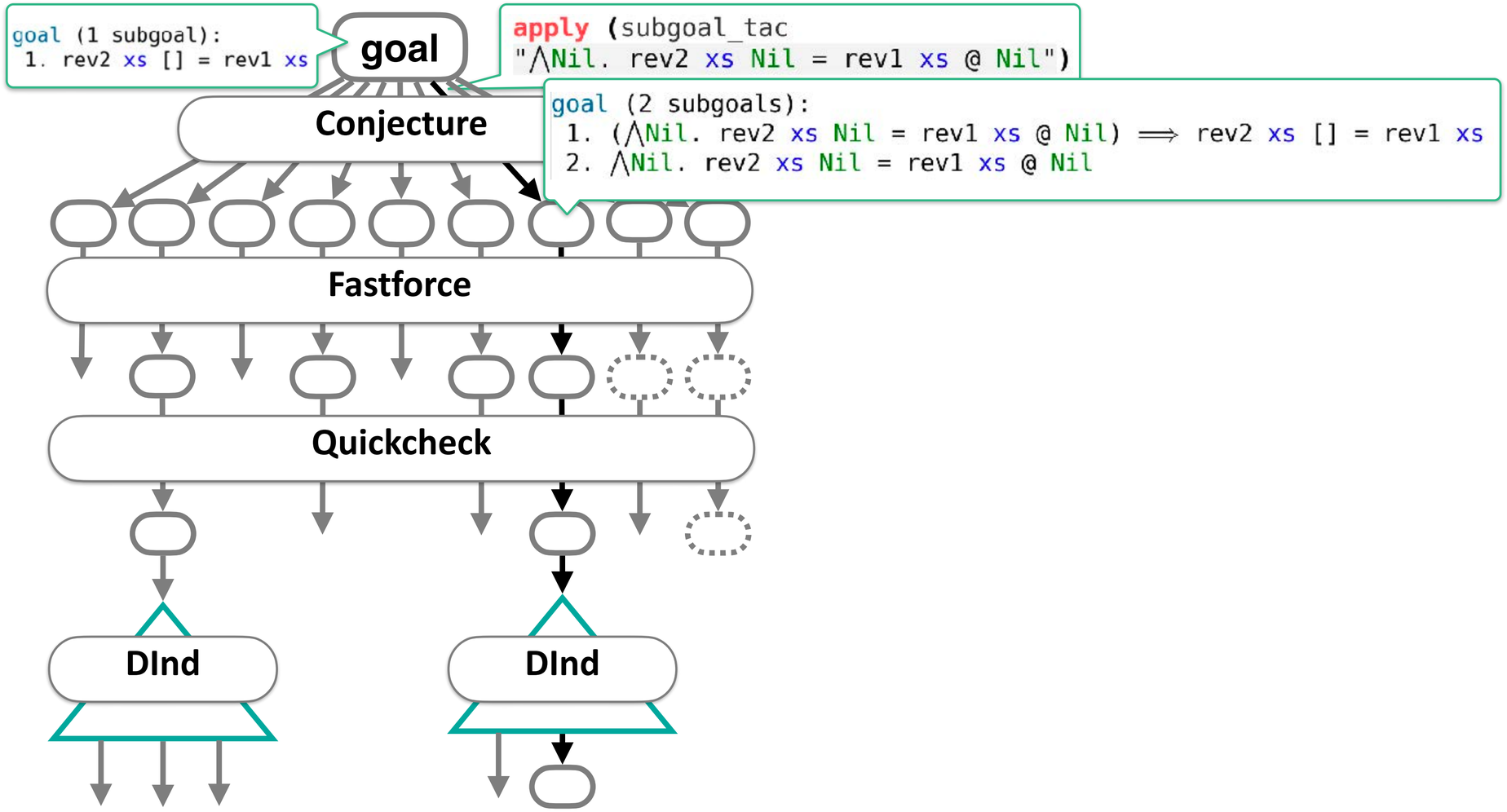


Algorithmic approaches work well, but not always → heuristics.

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.

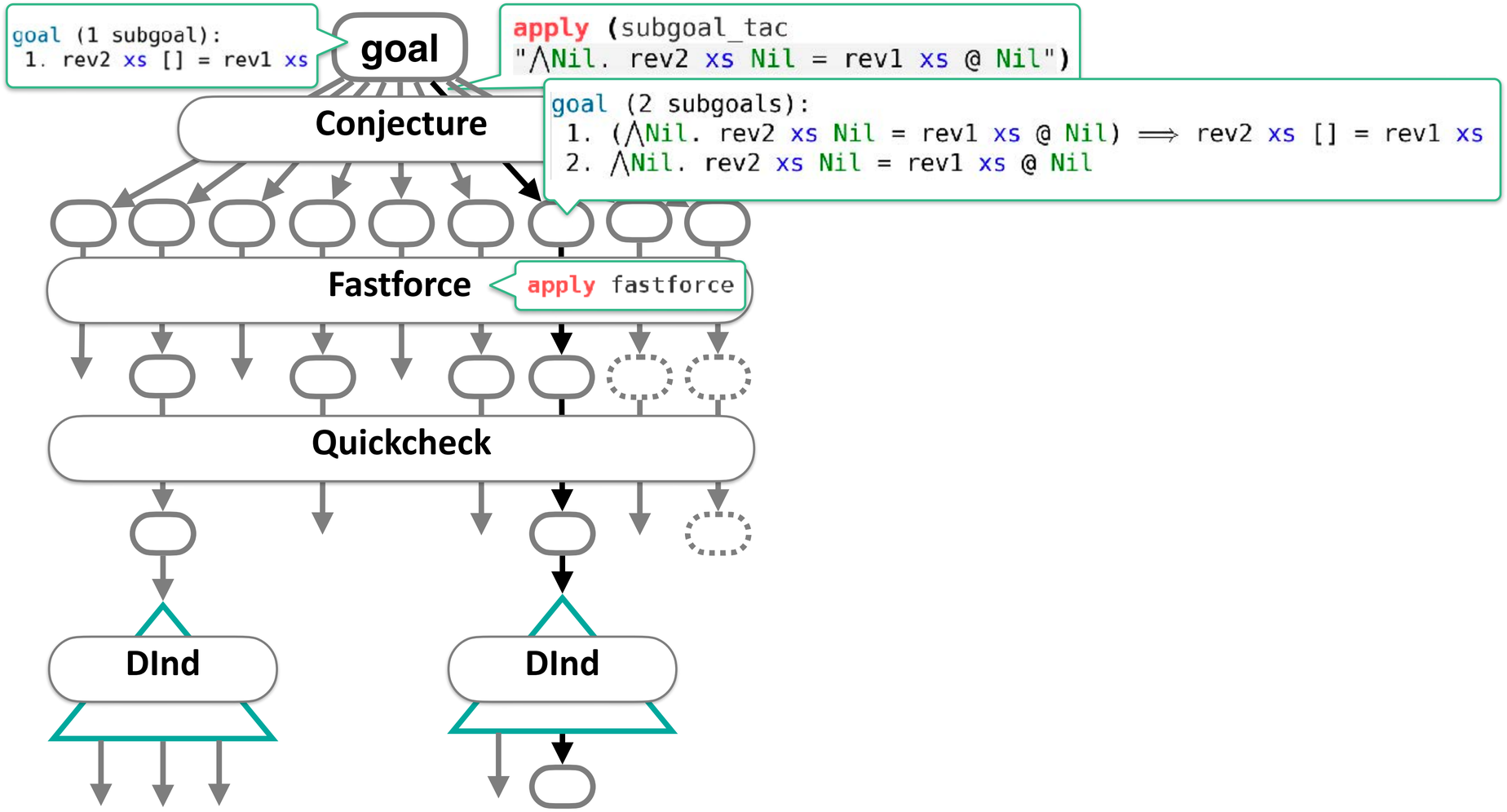


Algorithmic approaches work well, but not always → heuristics.

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.

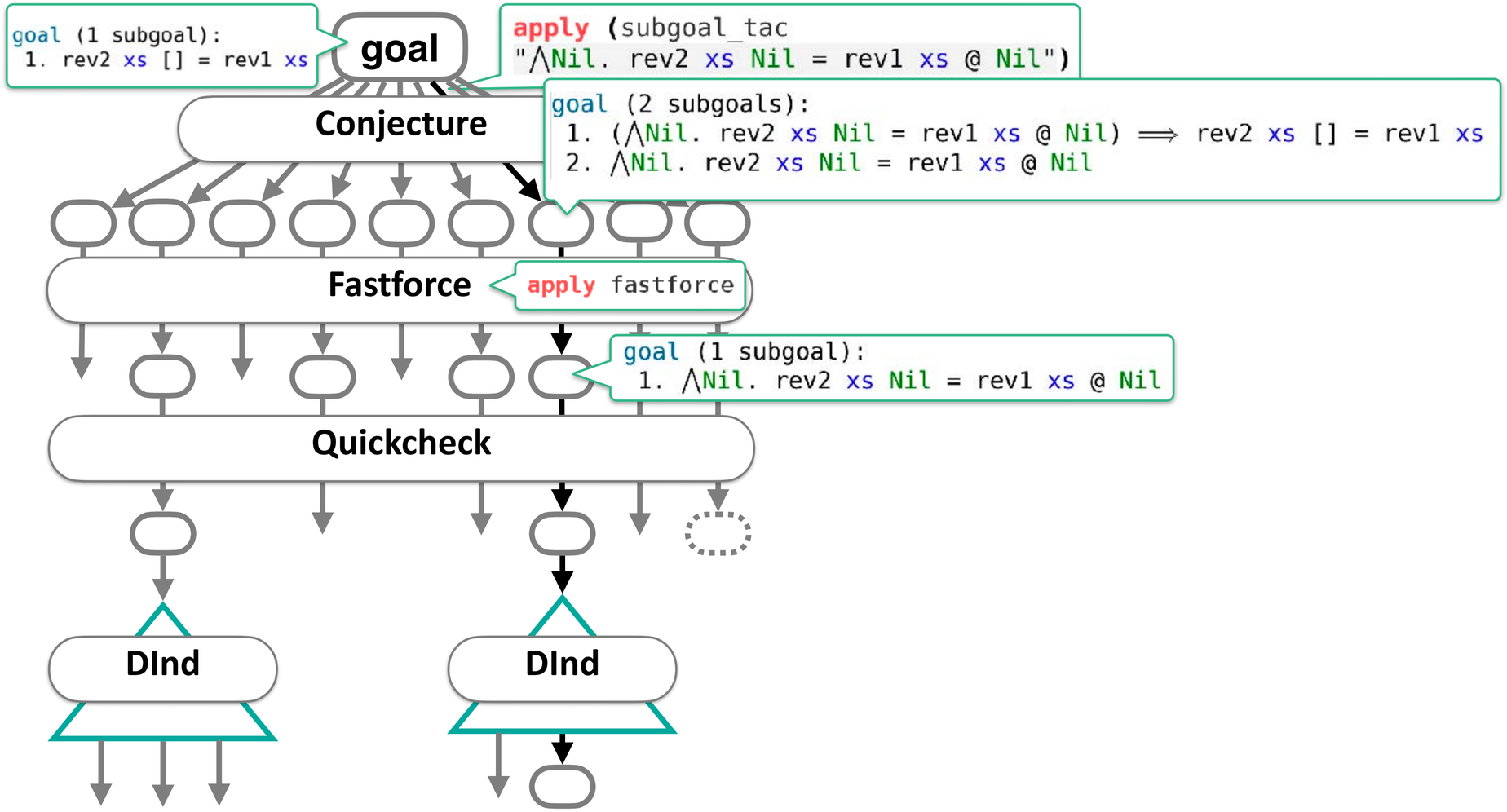


Algorithmic approaches work well, but not always → heuristics.

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.

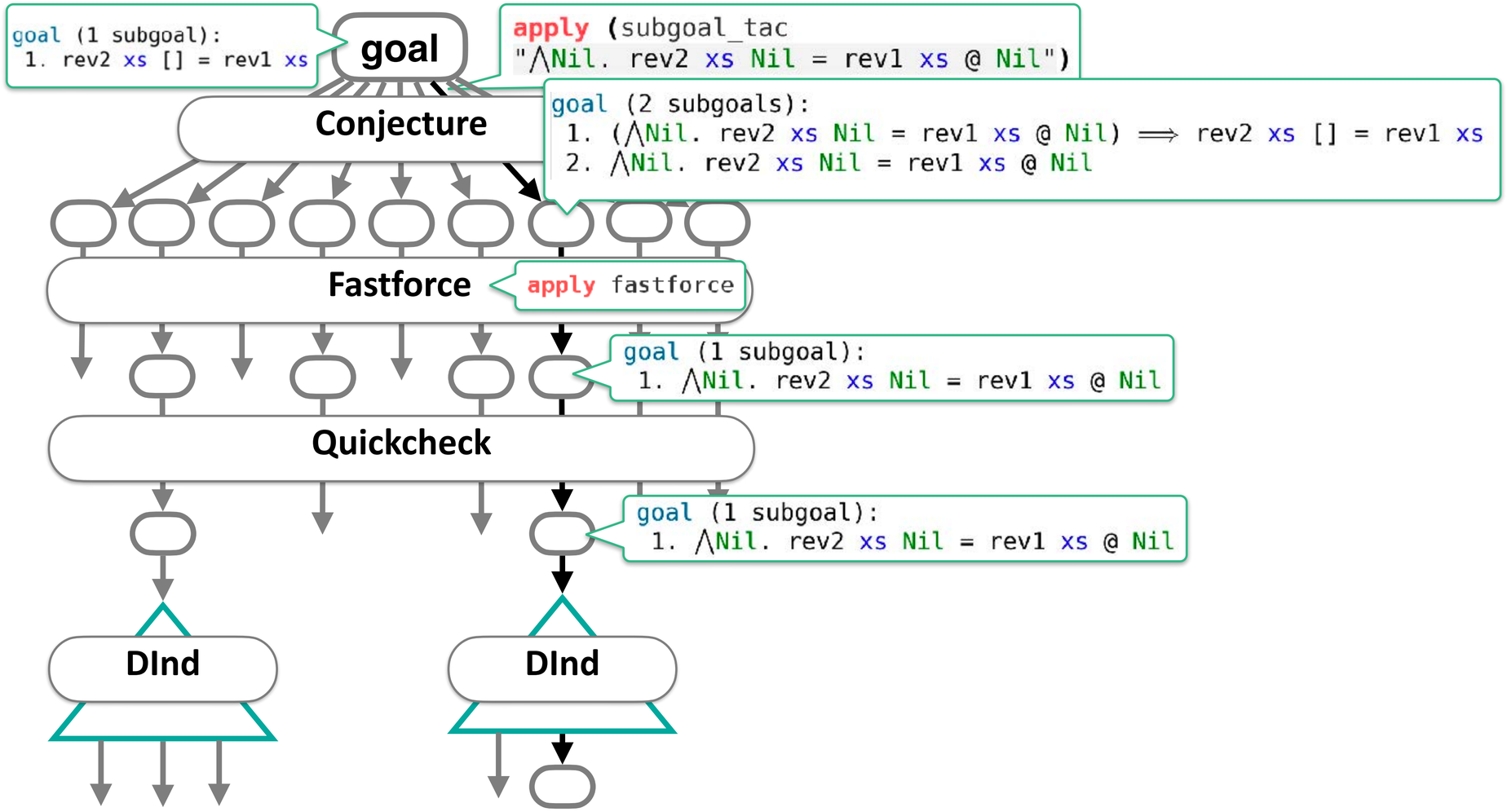


Algorithmic approaches work well, but not always → heuristics.

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.

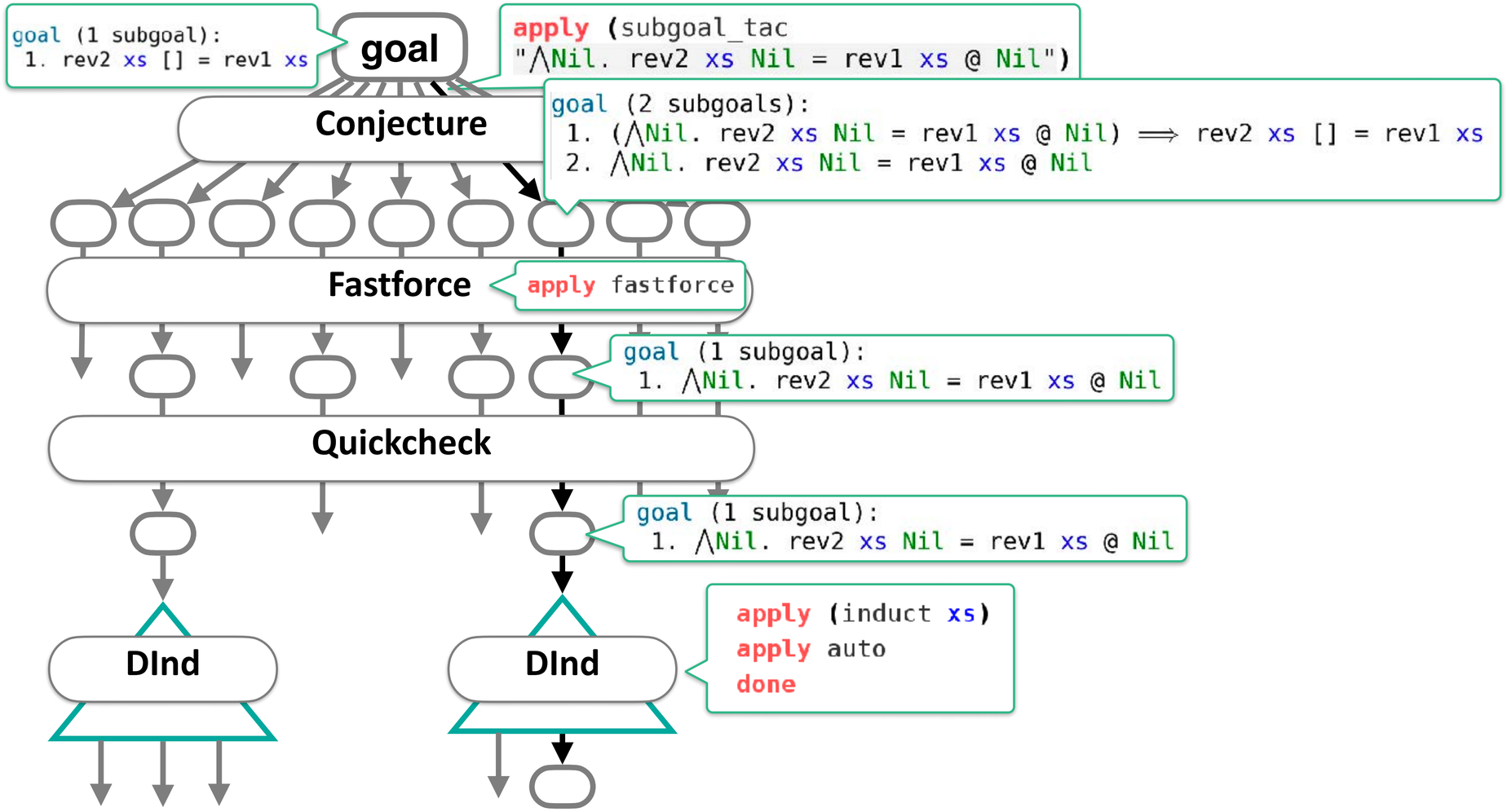


Algorithmic approaches work well, but not always → heuristics.

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.

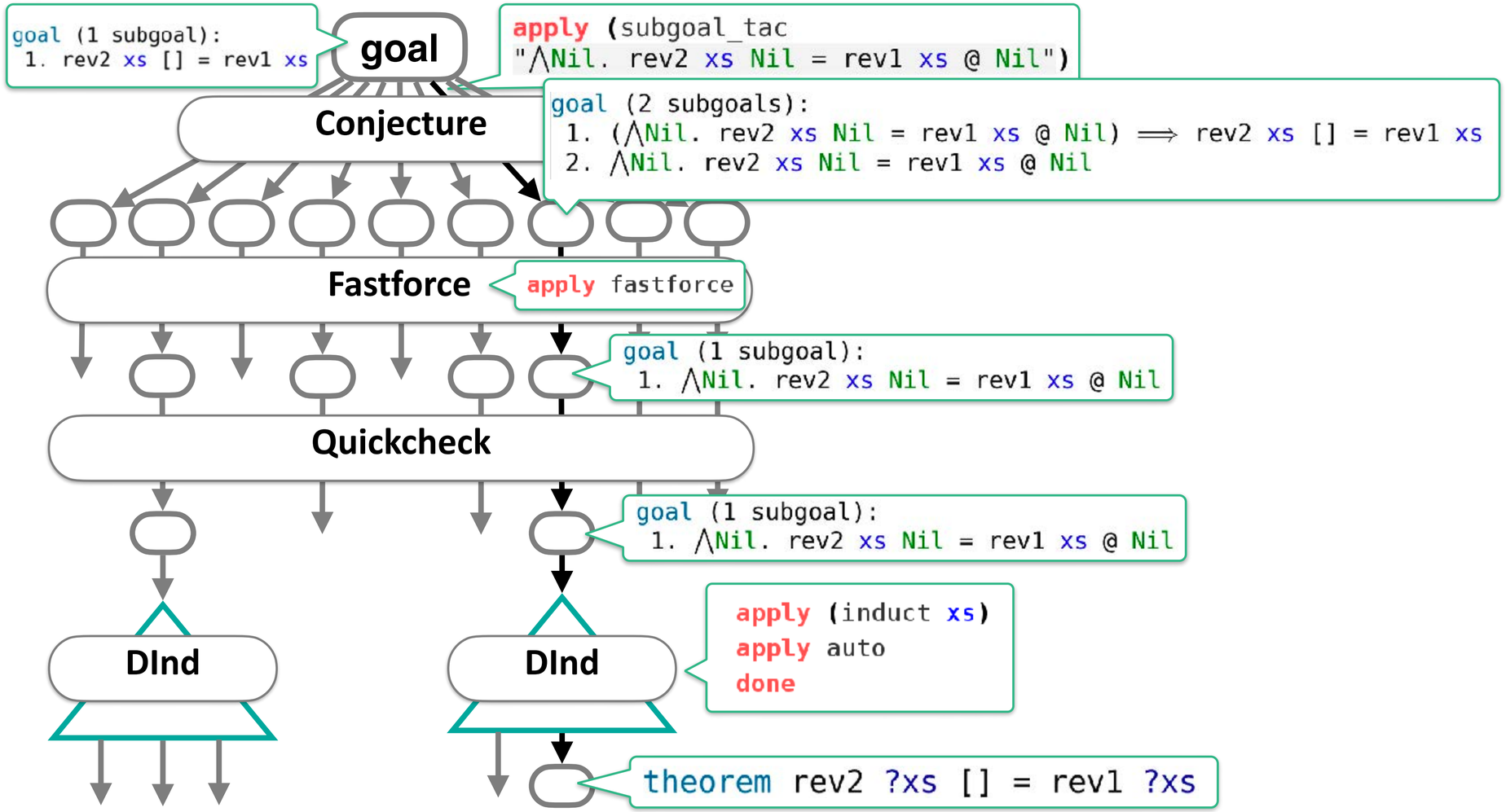


Algorithmic approaches work well, but not always → heuristics.

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.

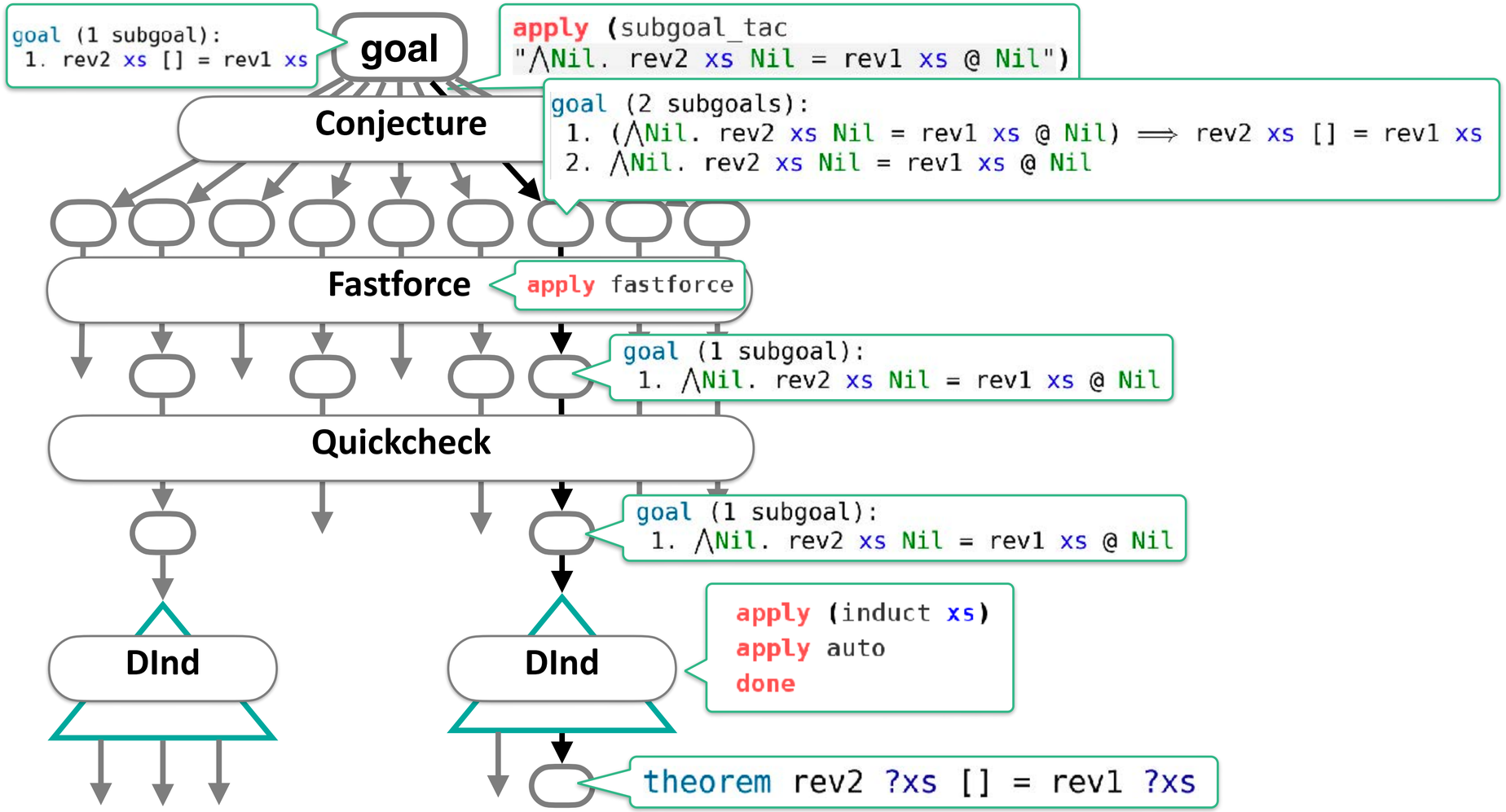


Algorithmic approaches work well, but not always → heuristics.

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.




```
apply (subgoal_tac "⋀Nil. rev2 xs Nil = rev1 xs @ Nil")
apply fastforce
apply (induct xs)
apply auto
done
```

Path to automatic induction

Step 1: Tactic recommendation.

Step 2: Tactic argument recommendation.

Path to automatic induction

Step 1: Tactic recommendation.

Step 2: Tactic argument recommendation.

a: Guided backtracking search with tracing (PSL).

Path to automatic induction

Step 1: Tactic recommendation.

Step 2: Tactic argument recommendation.

- a: Guided backtracking search with tracing (PSL).
- b: Logical Feature Extraction (LiFtEr).

Path to automatic induction

Step 1: Tactic recommendation.

Step 2: Tactic argument recommendation.

- a: Guided backtracking search with tracing (PSL).
- b: Logical Feature Extraction (LiFtEr).
- c: smart_induct using LiFtEr.

Path to automatic induction

Step 1: Tactic recommendation.

Step 2: Tactic argument recommendation.

- a: Guided backtracking search with tracing (PSL).
- b: Logical Feature Extraction (LiFtEr).
- c: smart_induct using LiFtEr.
- d: Semantic-aware Logical Feature Extraction (SeLFiE).

Path to automatic induction

Step 1: Tactic recommendation.

Step 2: Tactic argument recommendation.

- a: Guided backtracking search with tracing (PSL).
- b: Logical Feature Extraction (LiFtEr).
- c: smart_induct using LiFtEr.
- d: Semantic-aware Logical Feature Extraction (SeLFiE).
- e: sem_ind using SeLFiE.

Path to automatic induction

Step 1: Tactic recommendation.

Step 2: Tactic argument recommendation.

- a: Guided backtracking search with tracing (PSL).
- b: Logical Feature Extraction (LiFtEr).
- c: smart_induct using LiFtEr.
- d: Semantic-aware Logical Feature Extraction (SeLFiE).
- e: sem_ind using SeLFiE.
- f: conjecturing using abductive-reasoning.

Path to automatic induction

Step 1: Tactic recommendation.

Step 2: Tactic argument recommendation.

- a: Guided backtracking search with tracing (PSL).
- b: Logical Feature Extraction (LiFtEr).
- c: smart_induct using LiFtEr.
- d: Semantic-aware Logical Feature Extraction (SeLFiE).
- e: sem_ind using SeLFiE.
- f: conjecturing using abductive-reasoning.

Step 3: Putting things together.

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%

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%

speedup [times]	occurrence
$x < 1.0$	3.0 (2.4%)
$1.0 \leq x < 5.0$	64.0 (50.8%)
$5.0 \leq x < 10.0$	44.0 (34.9%)
$10.0 \leq x < 15.0$	9.0 (7.1%)
$15.0 \leq x < 20.0$	3.0 (2.4%)
$20.0 \leq x$	3.0 (2.4%)

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!