https://twitter.com/YutakangE

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

Yutaka Nagashima
University of Innsbruck
Czech Technical University

Yutaka Ng
yutakang

Block or report user

https://twitter.com/YutakangE

Previously at Master Seminar…

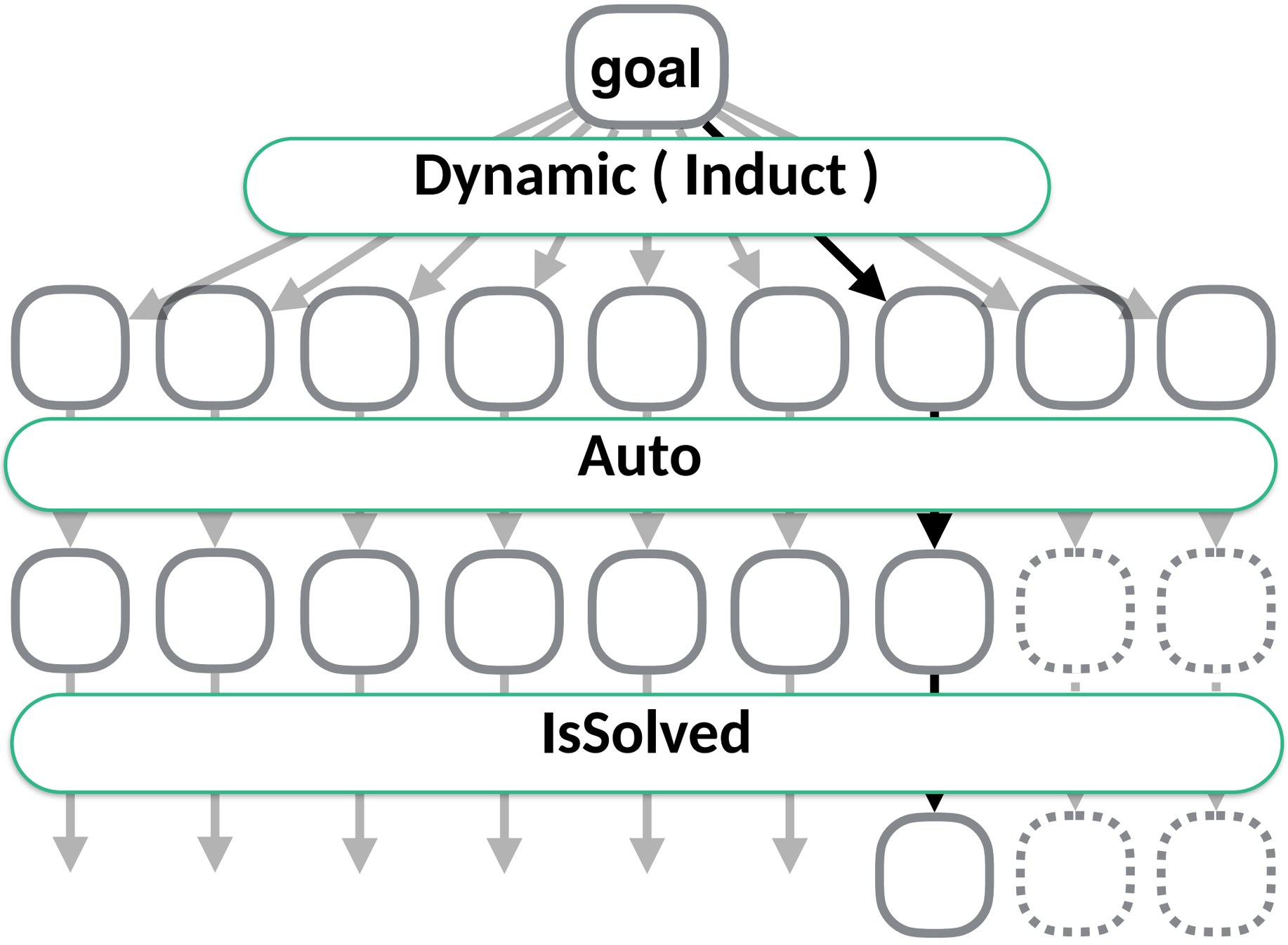
https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

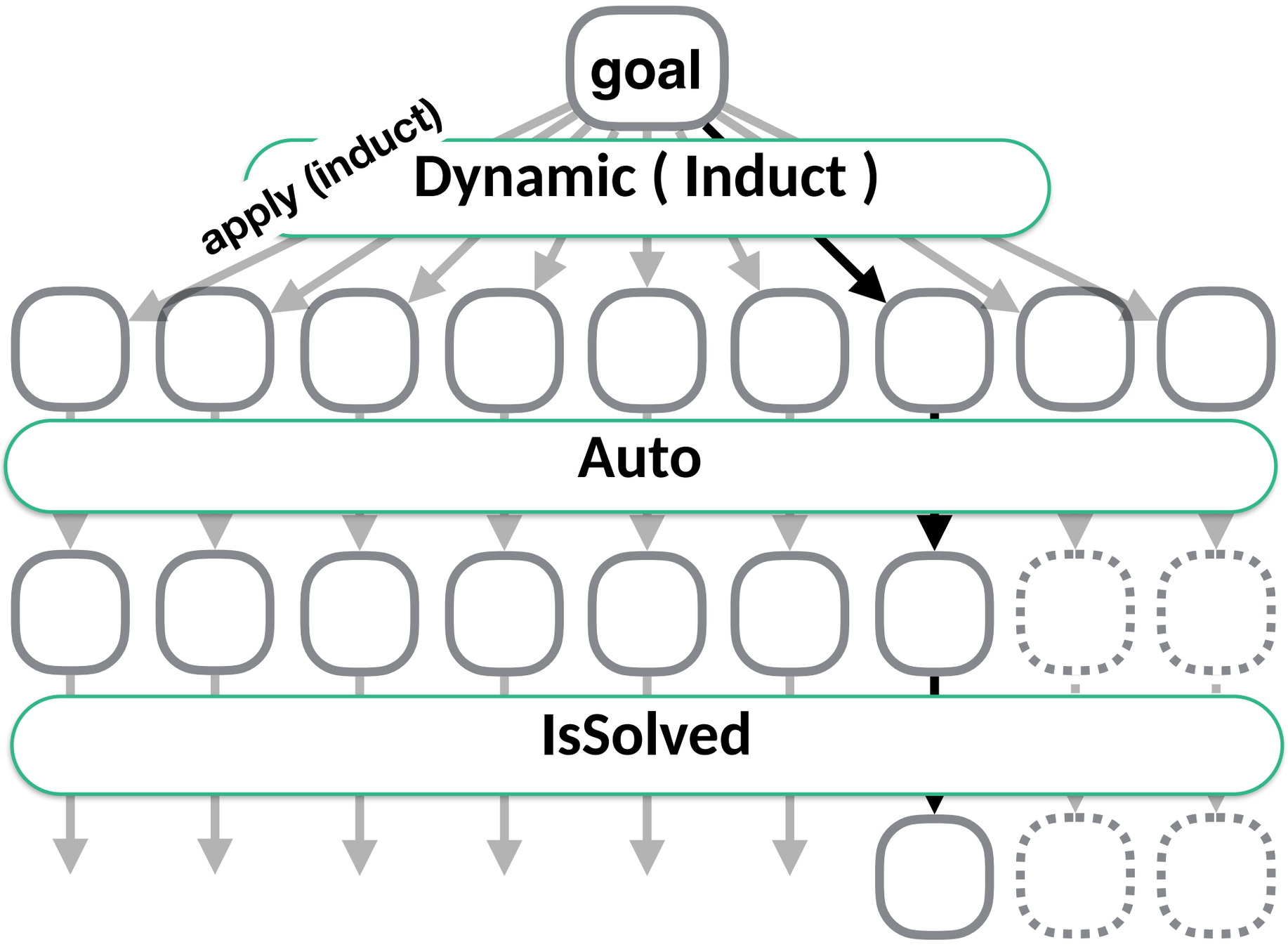
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

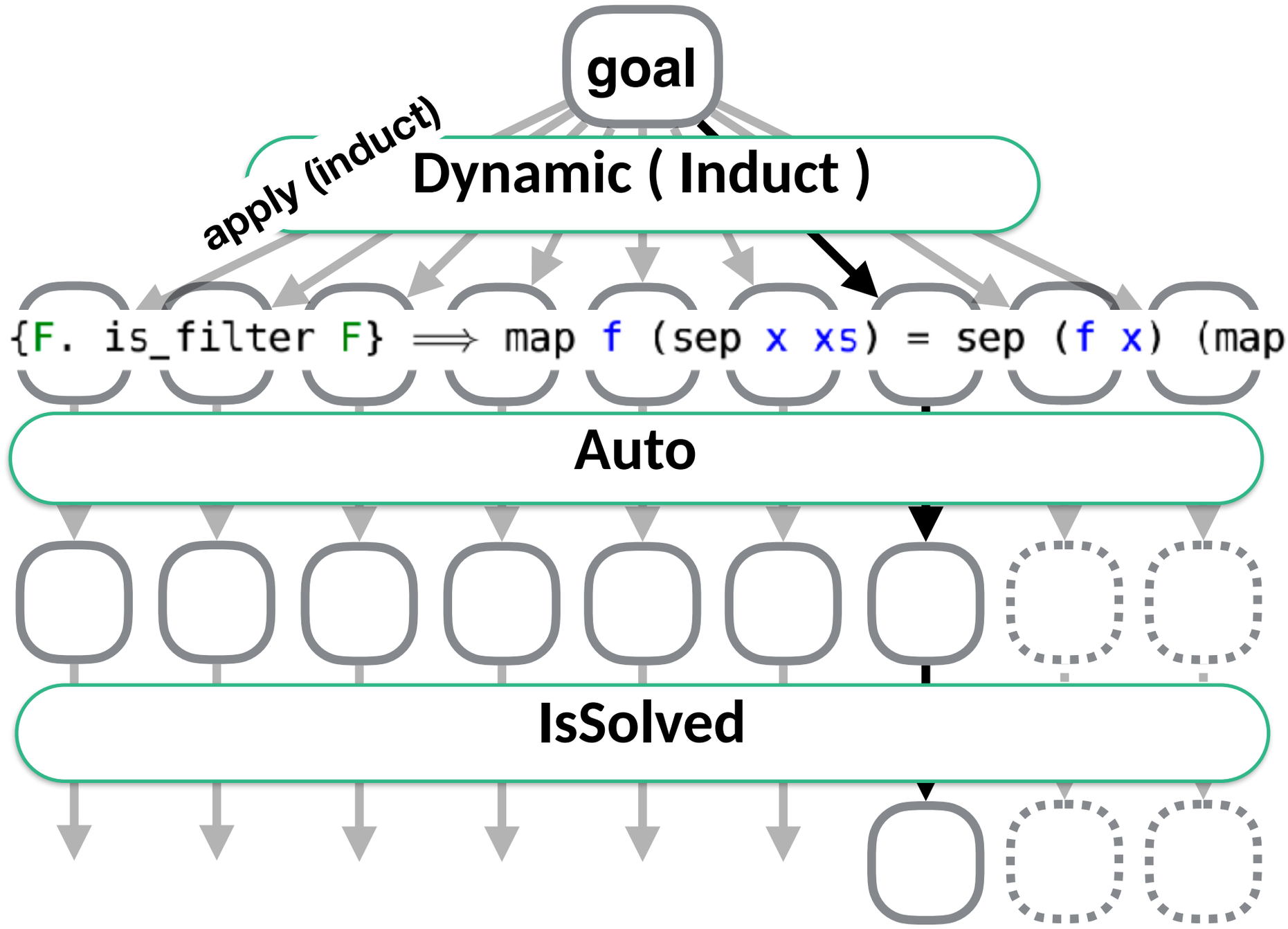
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

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

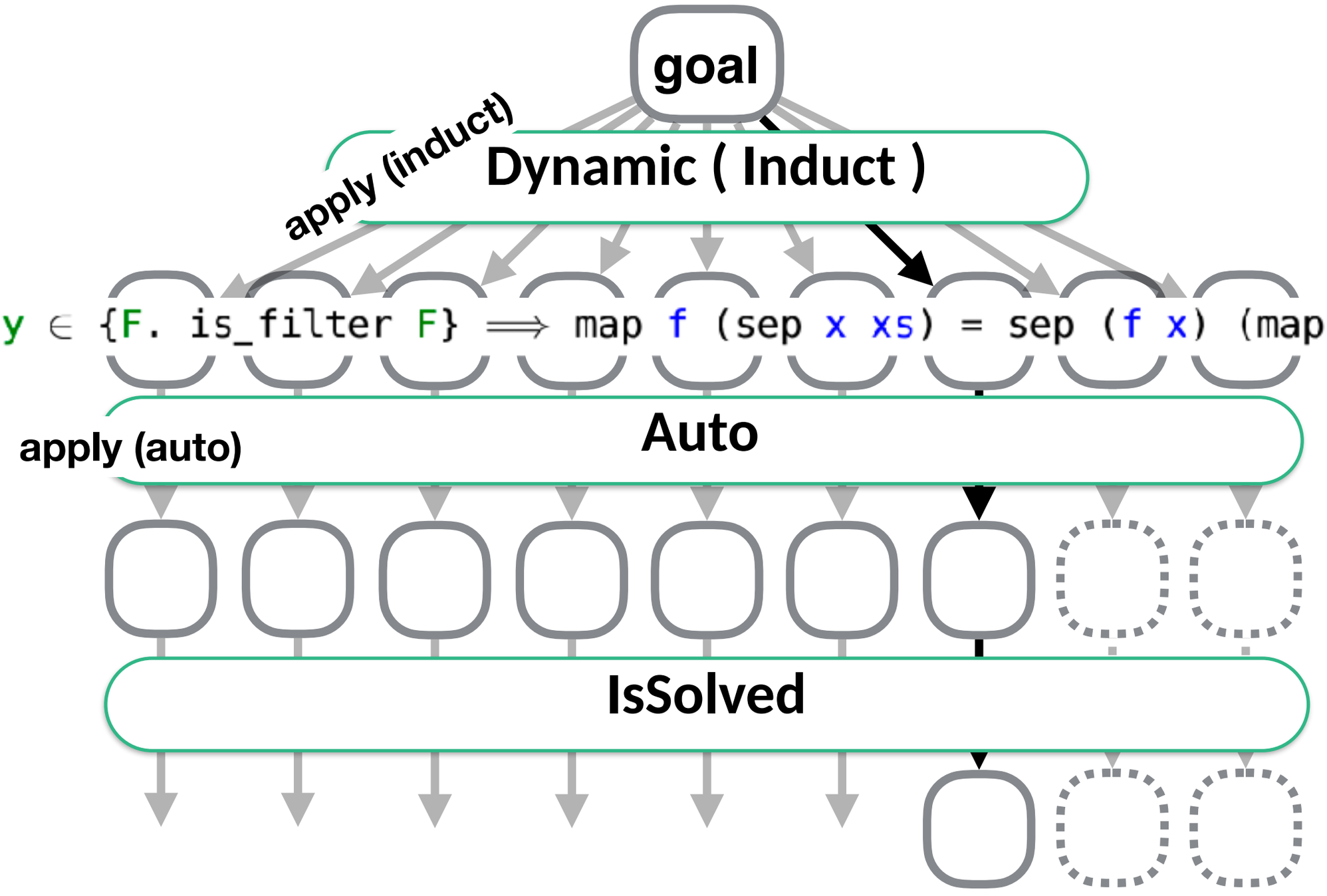



```
⋀y. y ∈ {F. is_filter F} ⟹ map f (sep x xs) = sep (f x) (map f xs)
```

https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

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

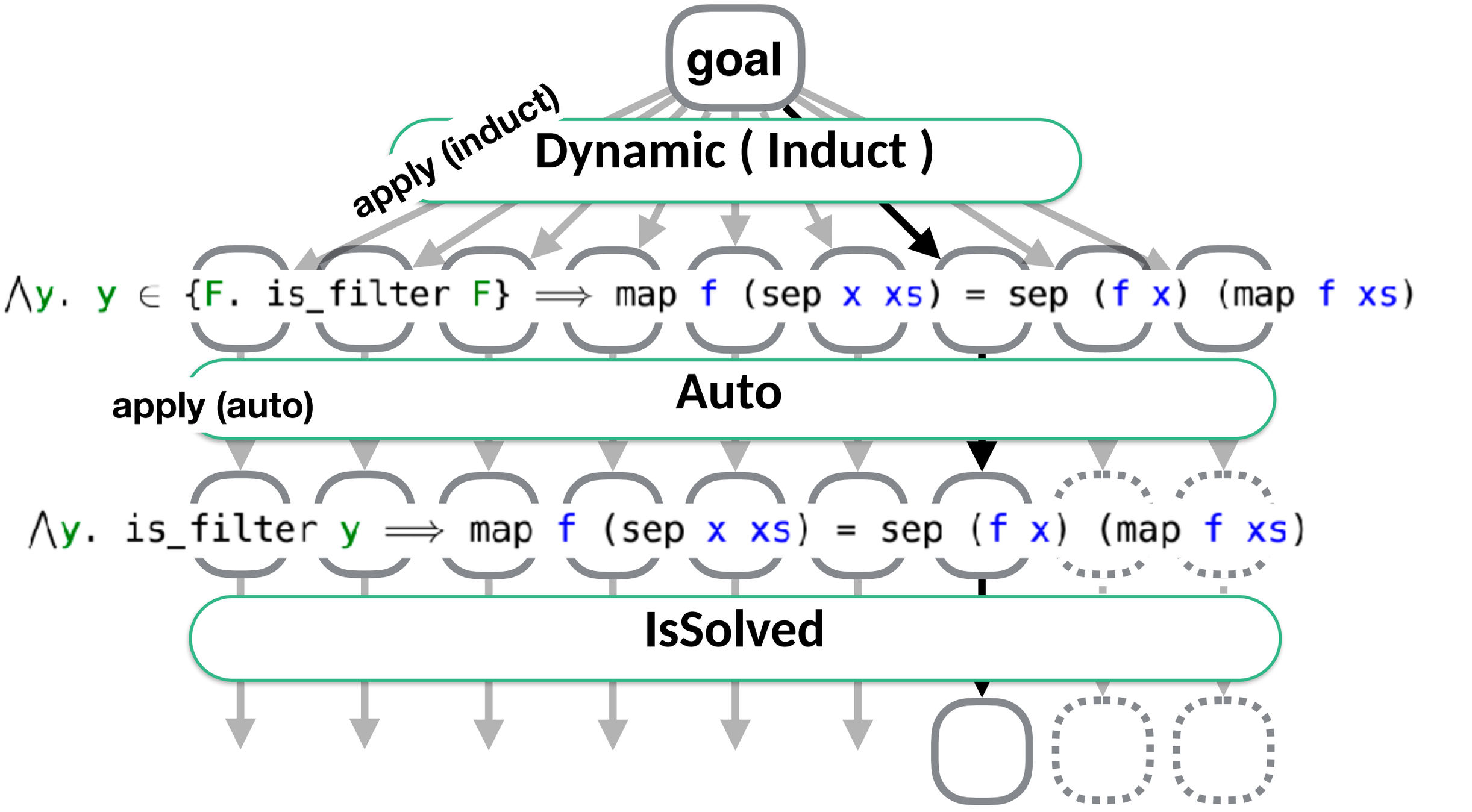



```
⋀y. y ∈ {F. is_filter F} ⟹ map f (sep x xs) = sep (f x) (map f xs)
```

https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

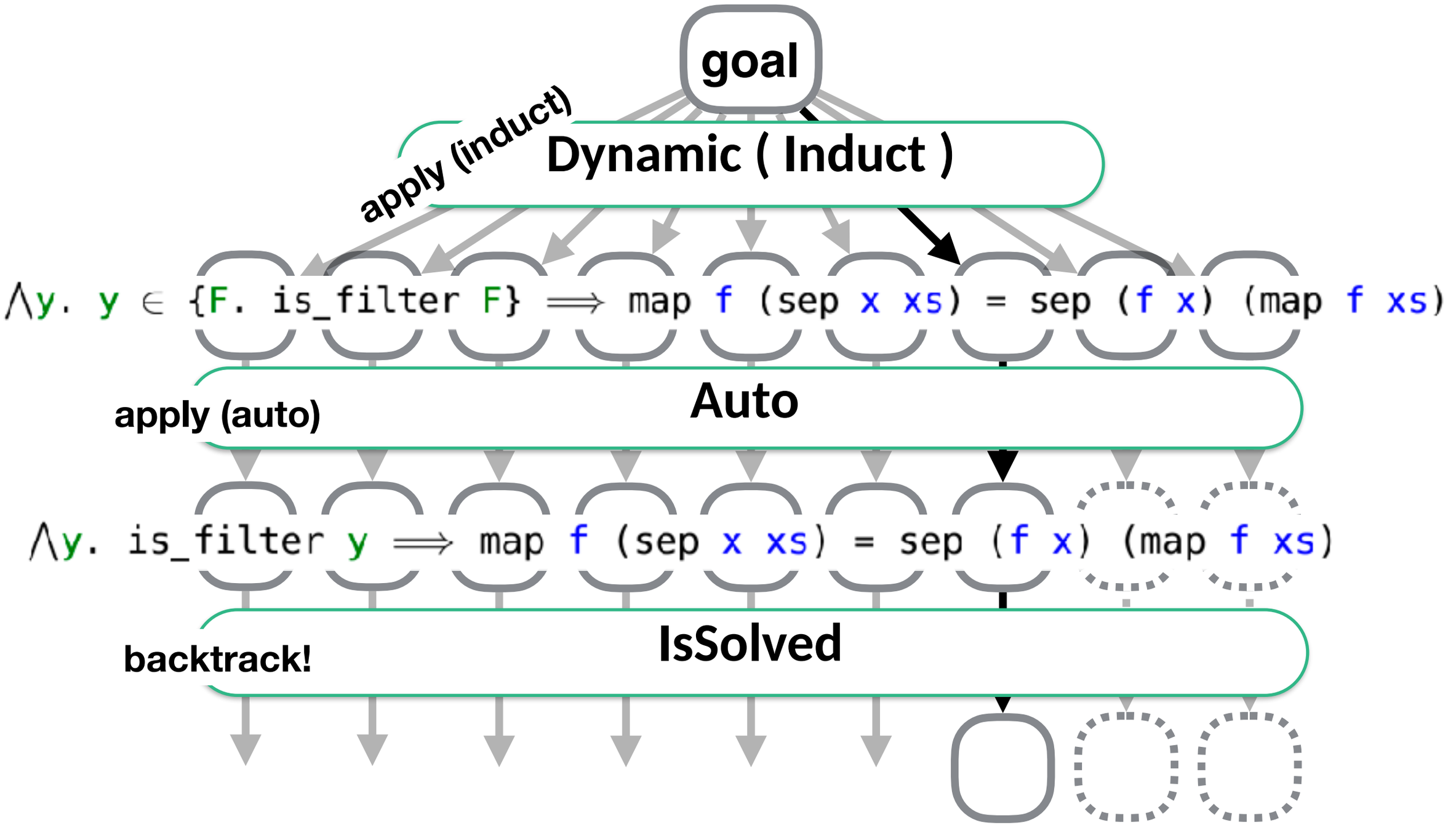
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

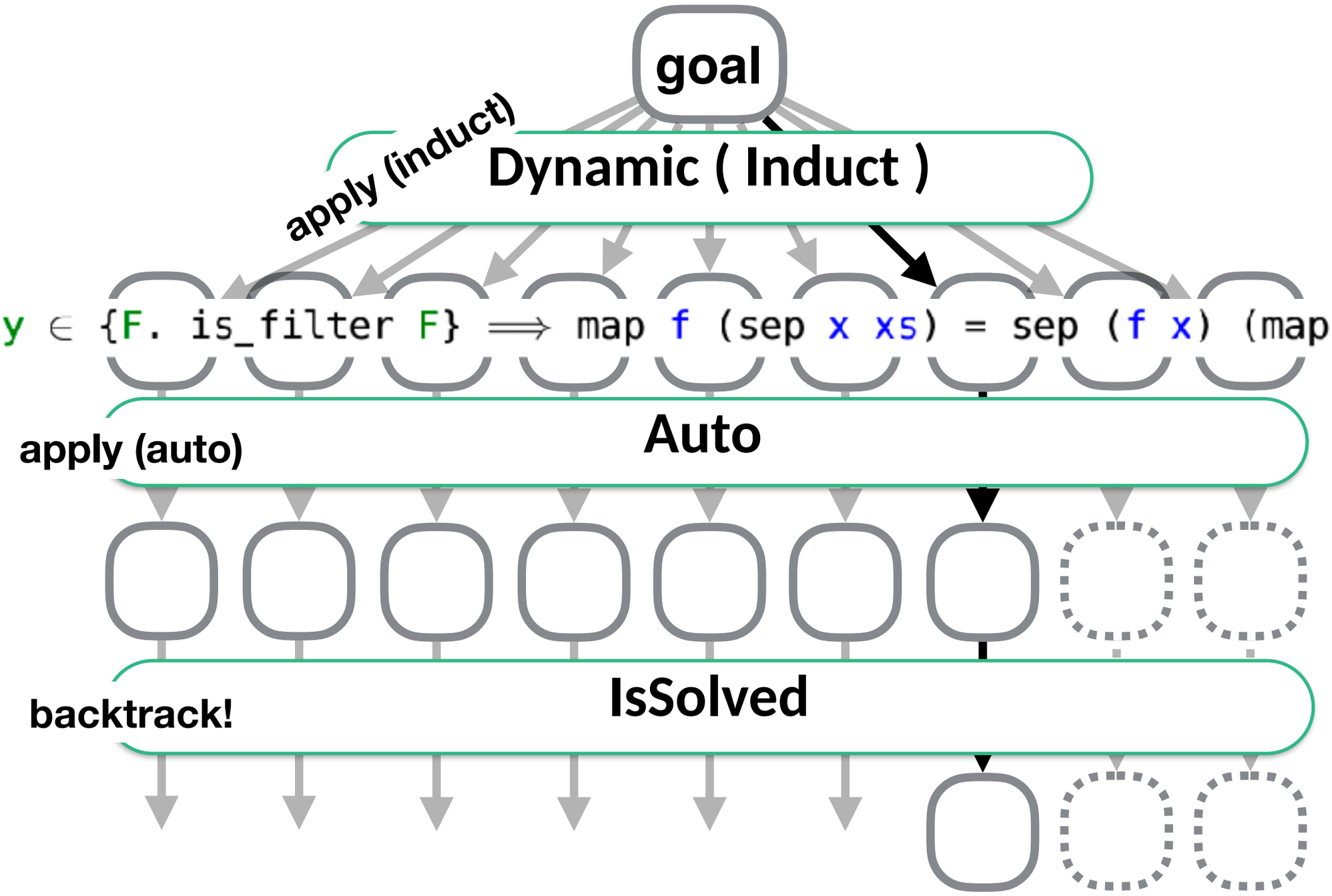
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

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

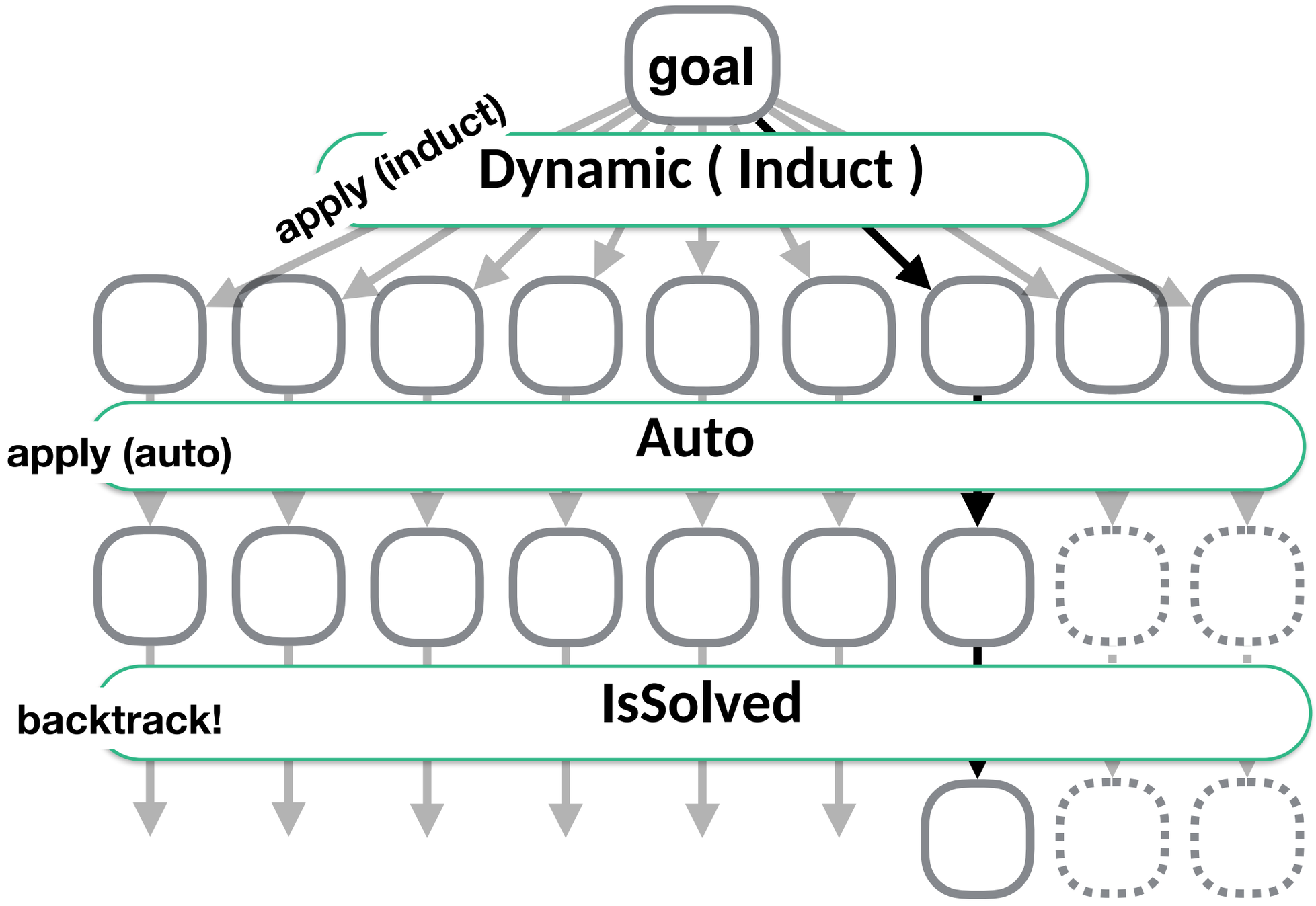



```
⋀y. y ∈ {F. is_filter F} ⟹ map f (sep x xs) = sep (f x) (map f xs)
```

https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

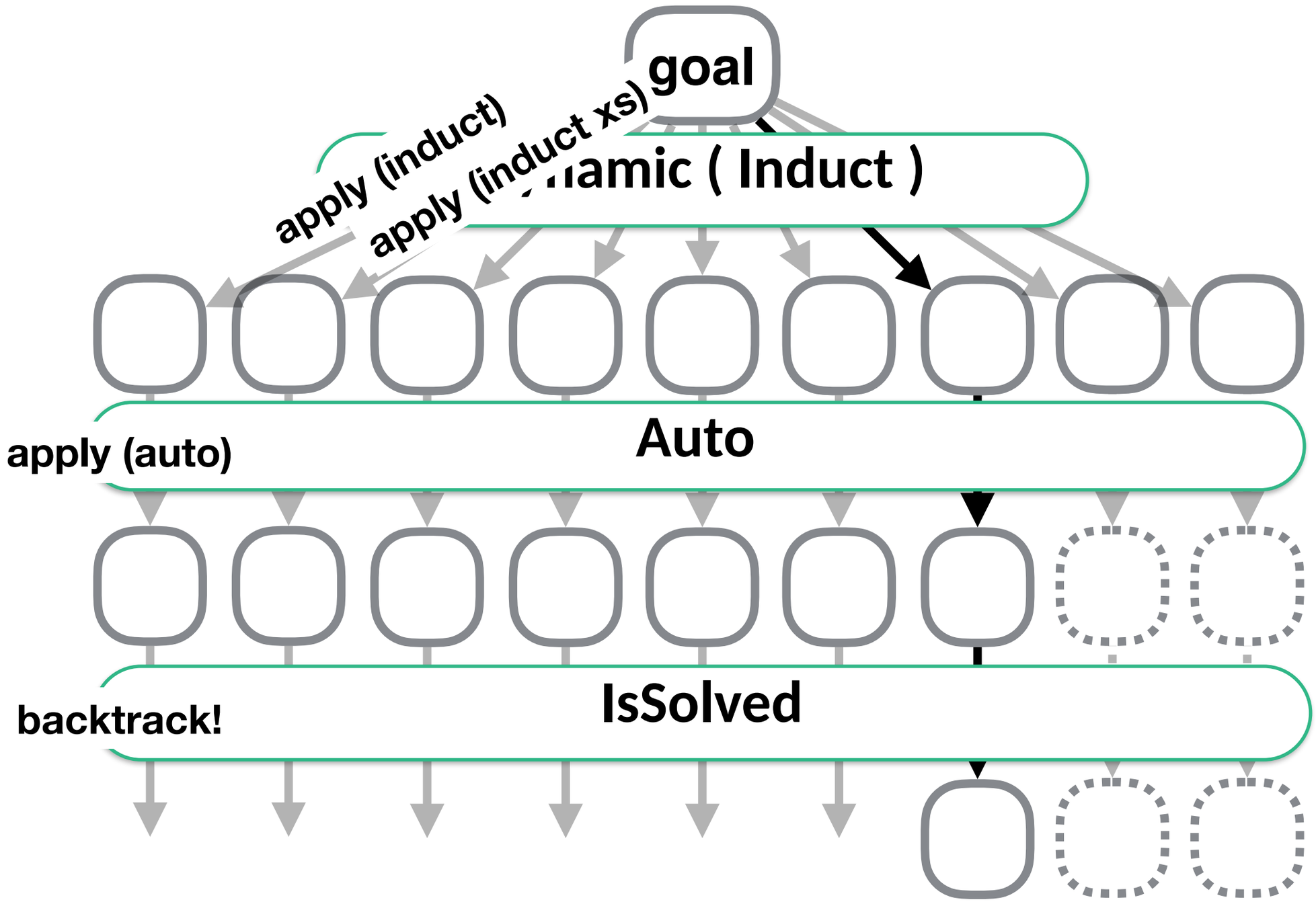
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

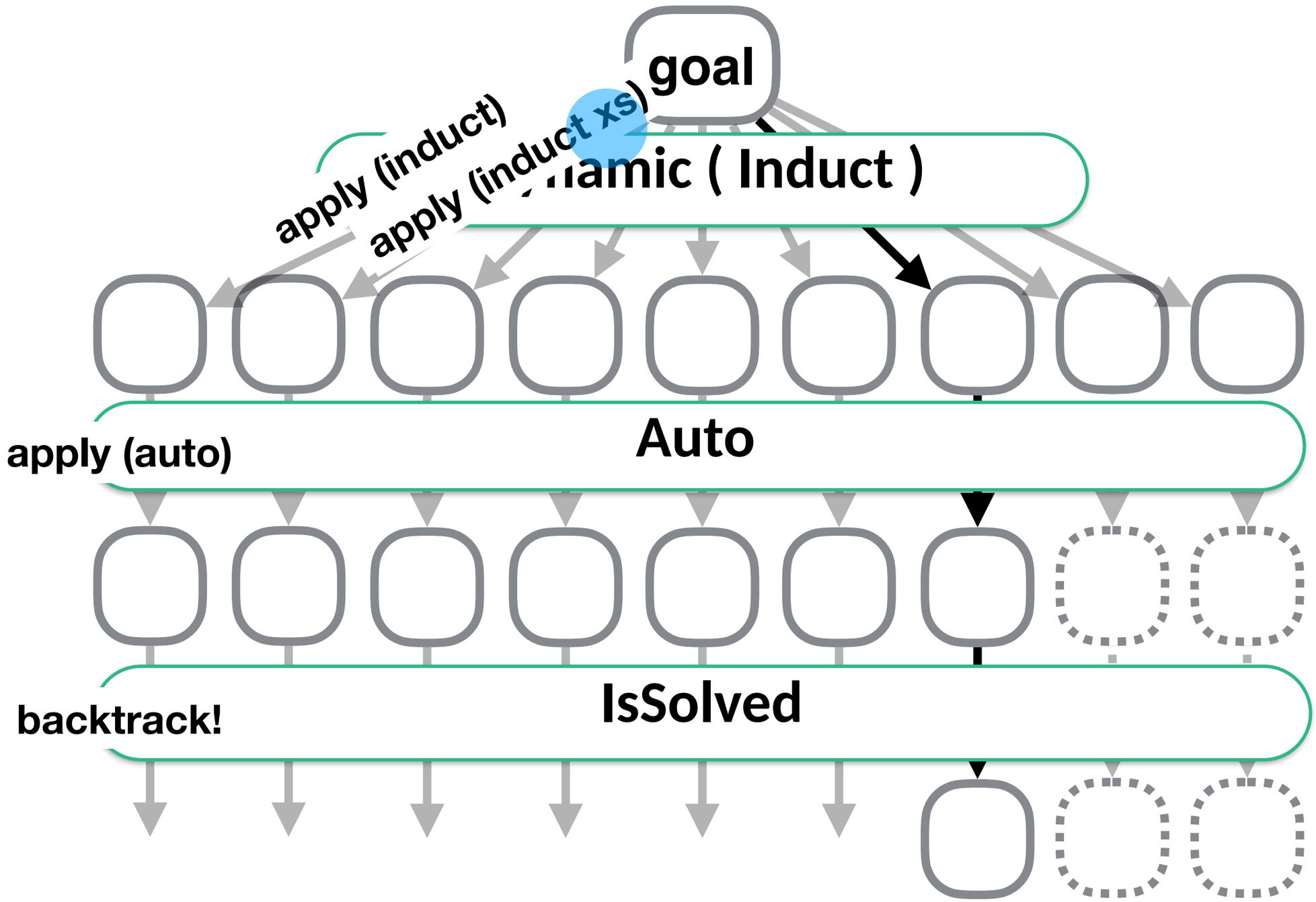
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

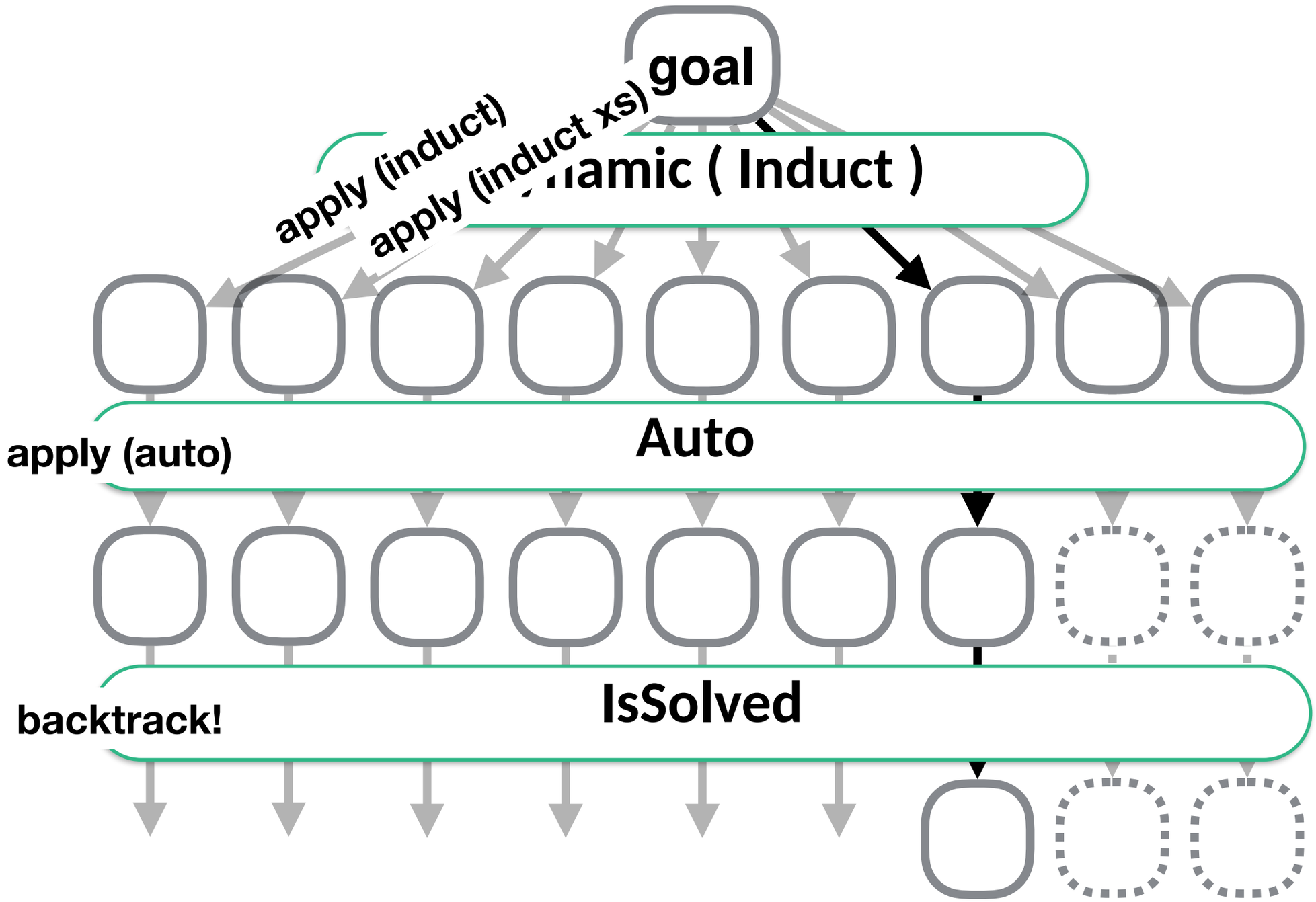
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

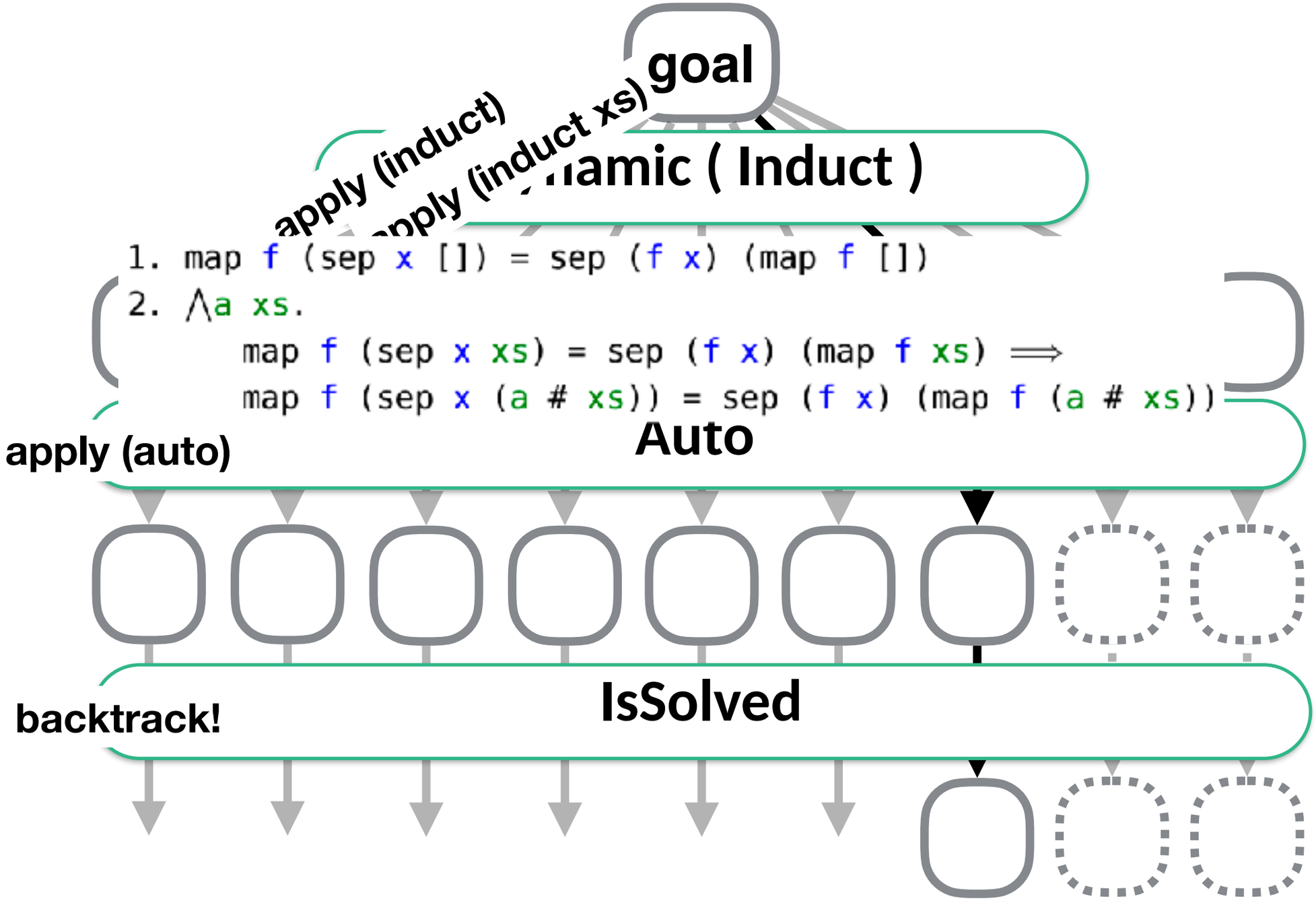
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

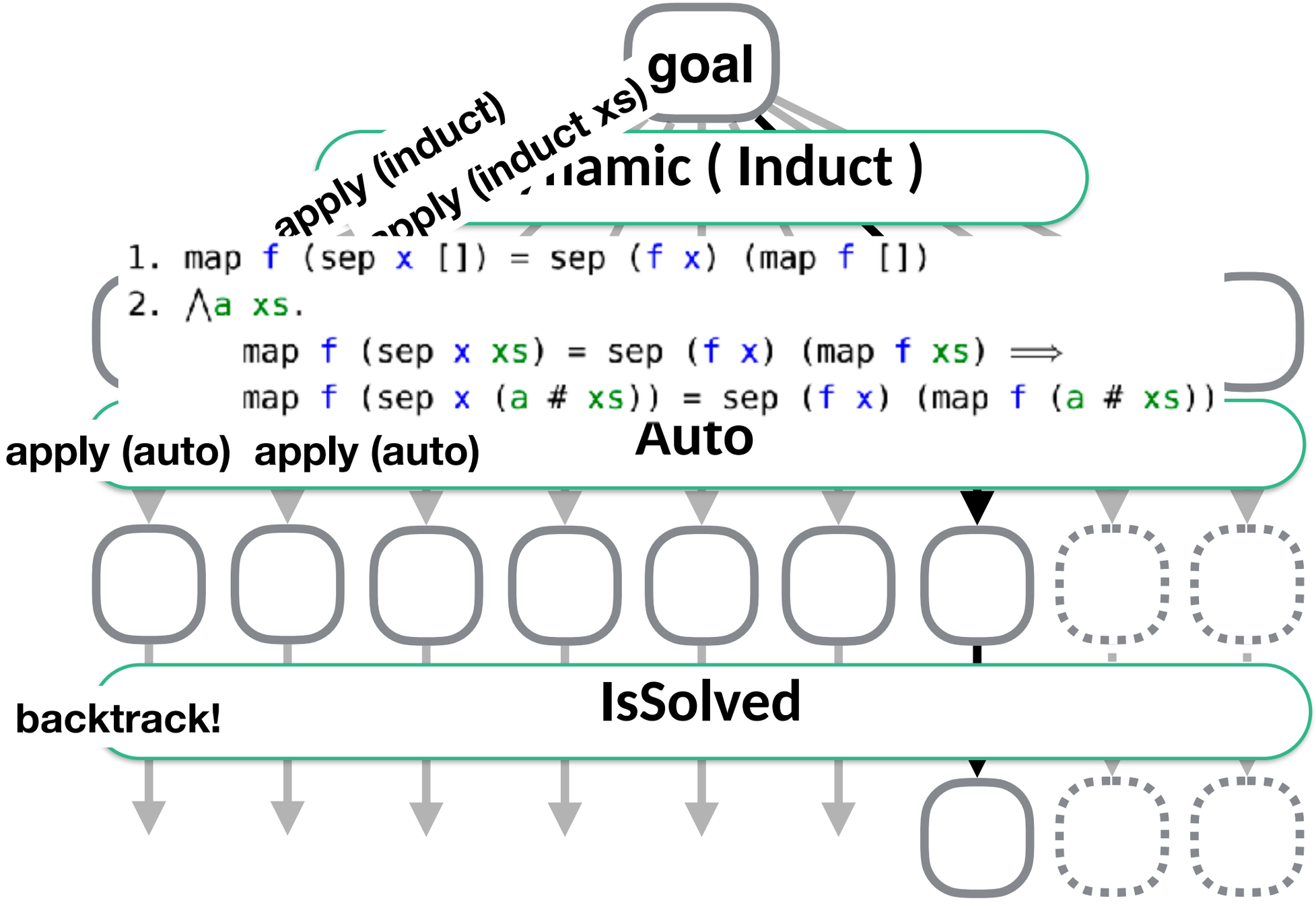
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

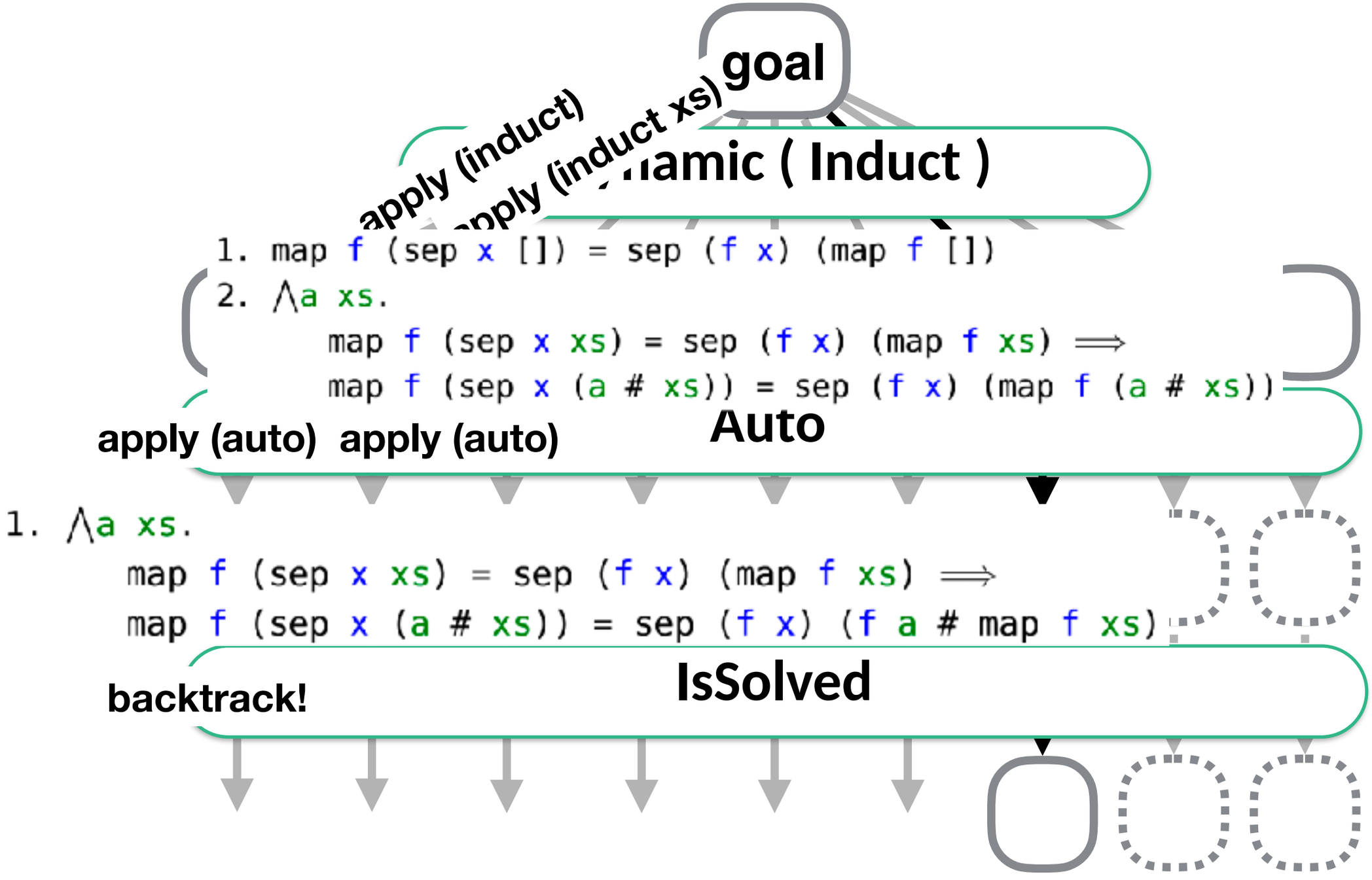
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

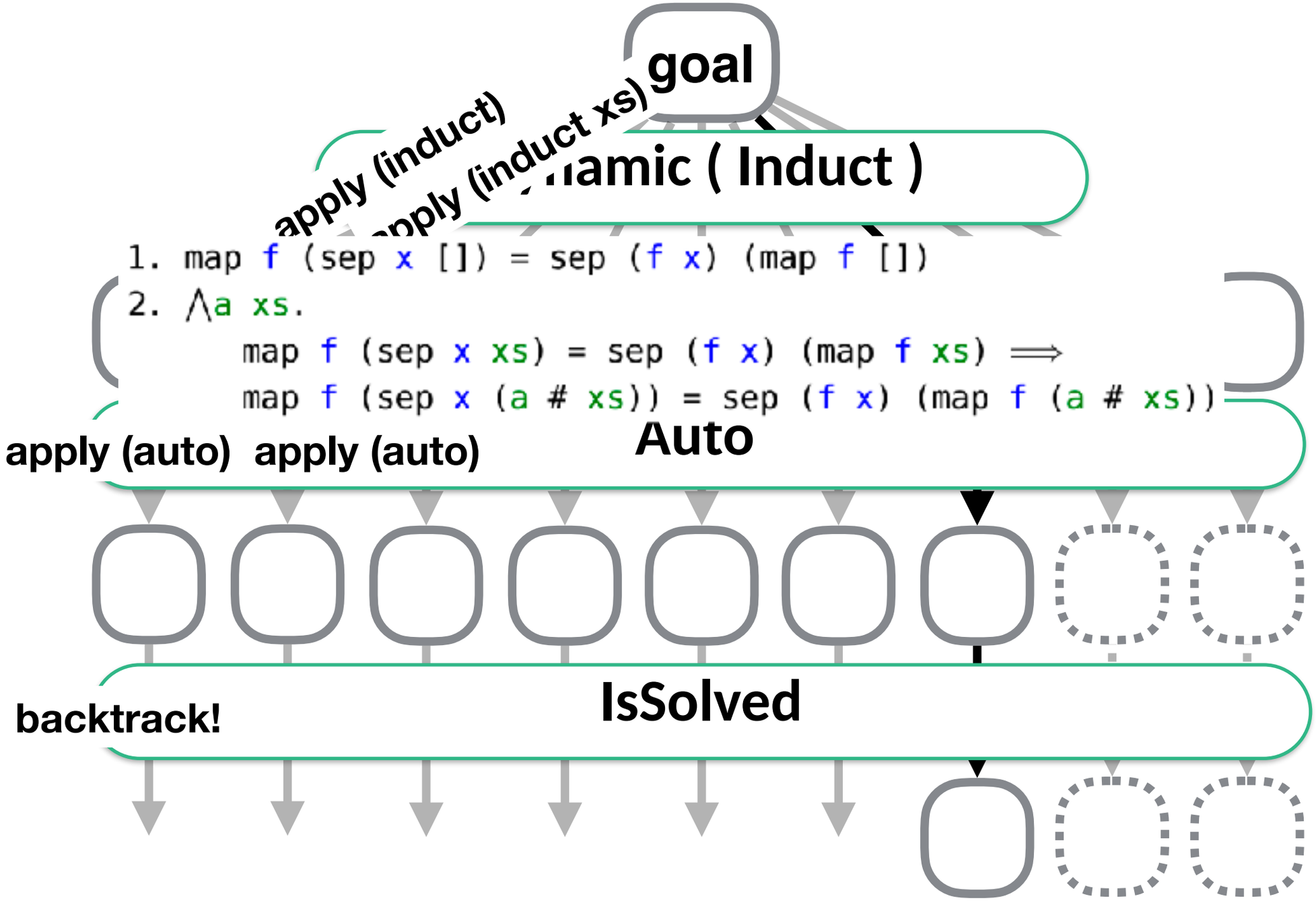
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

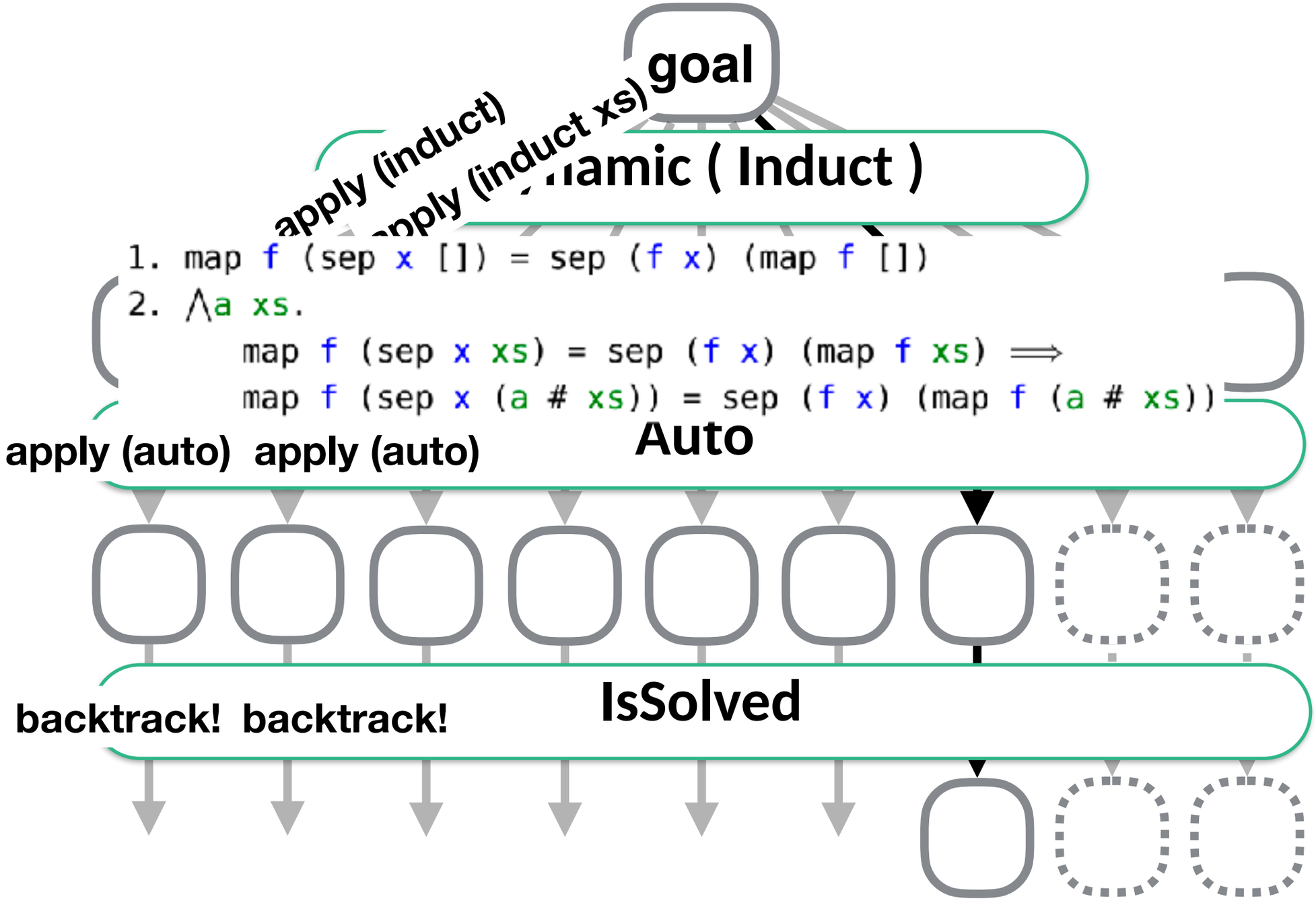
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

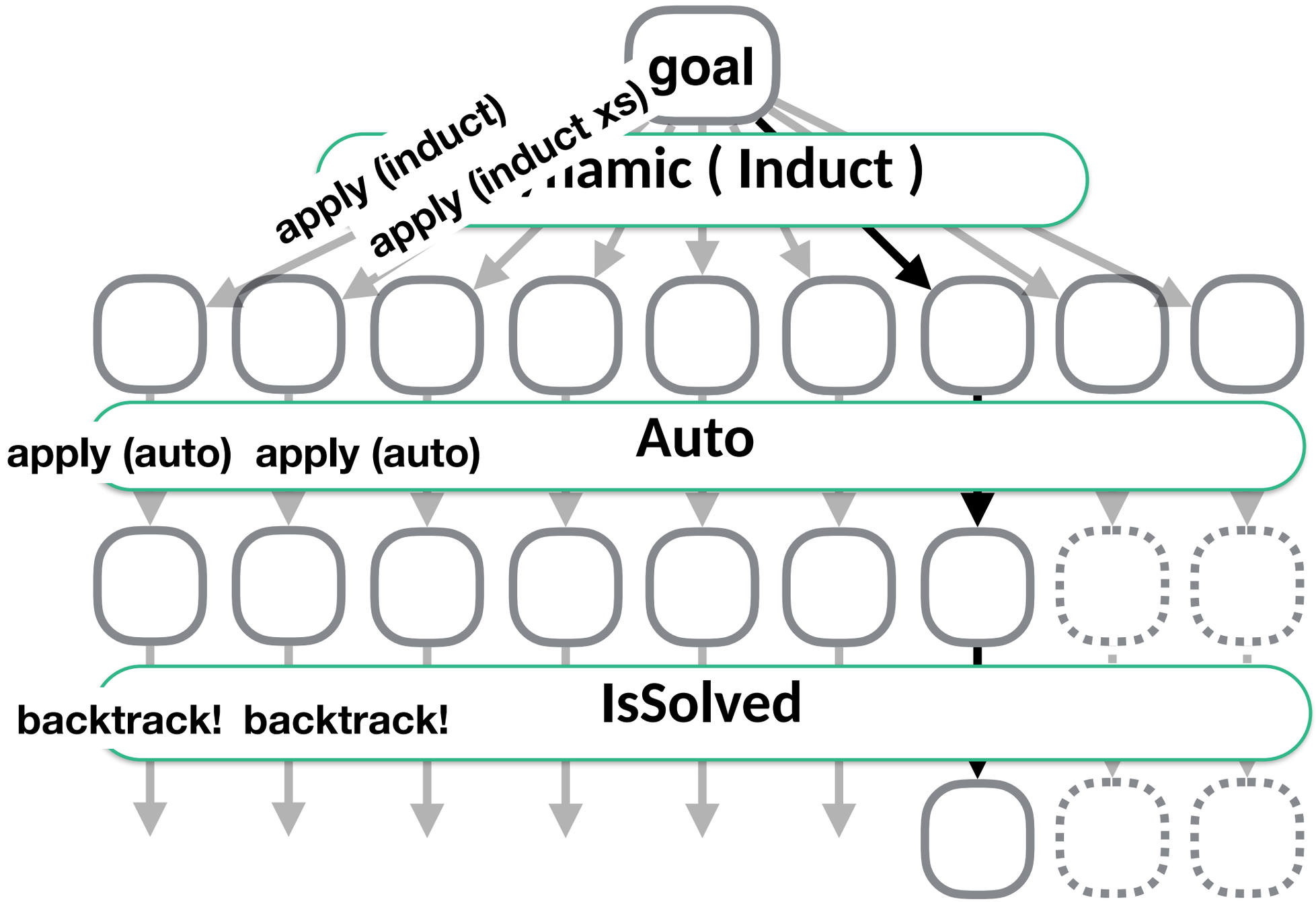
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

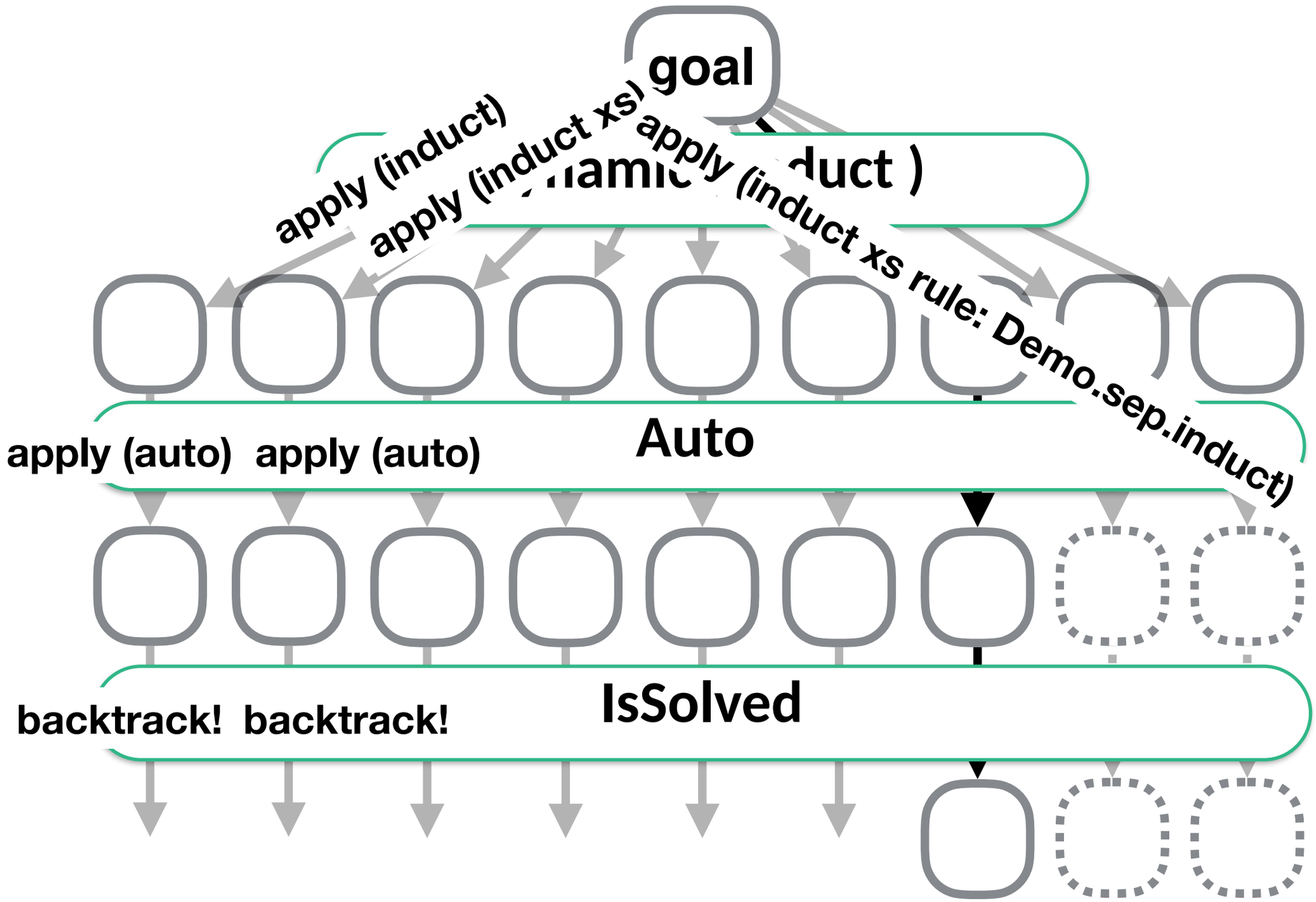
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

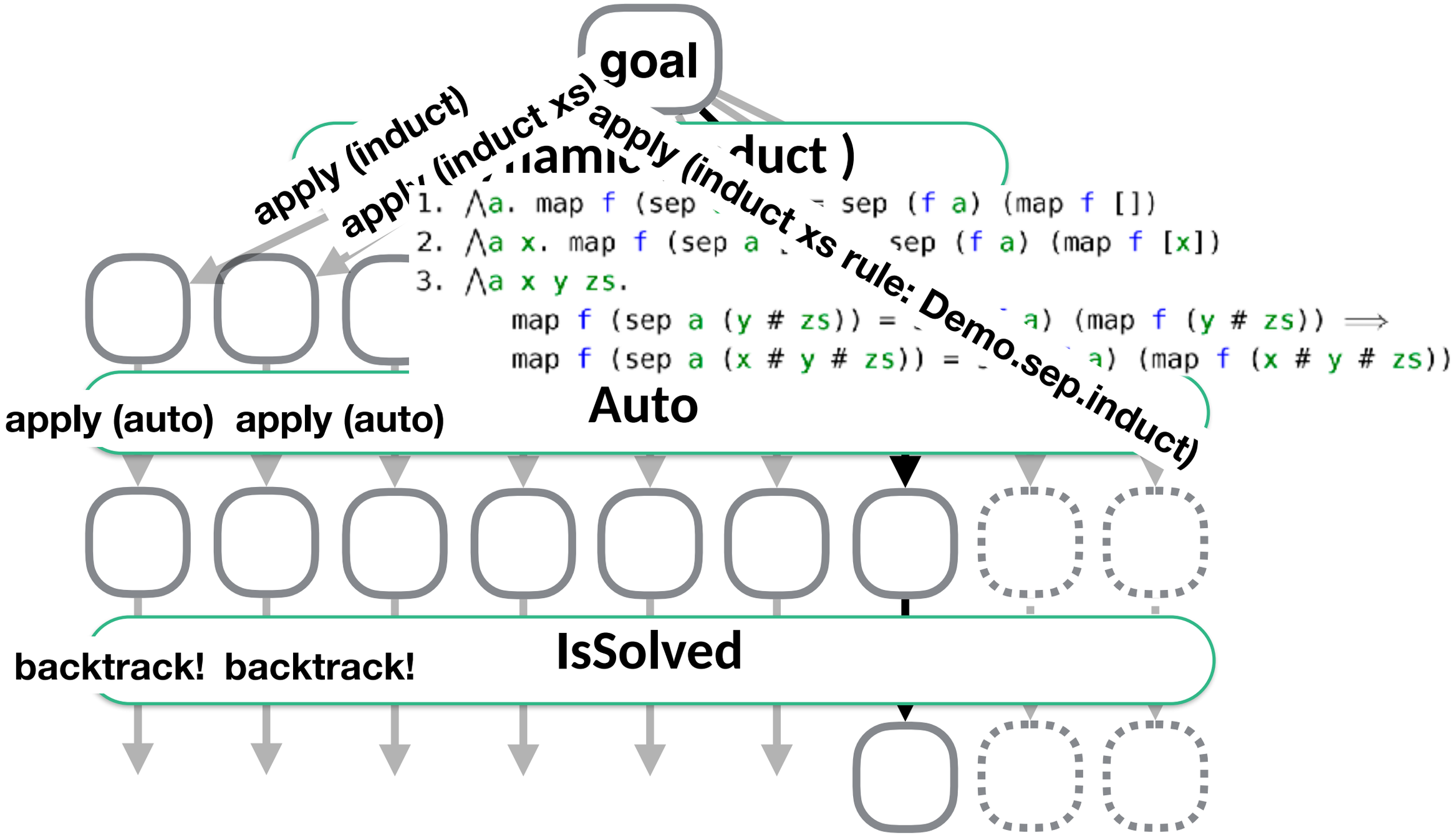
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

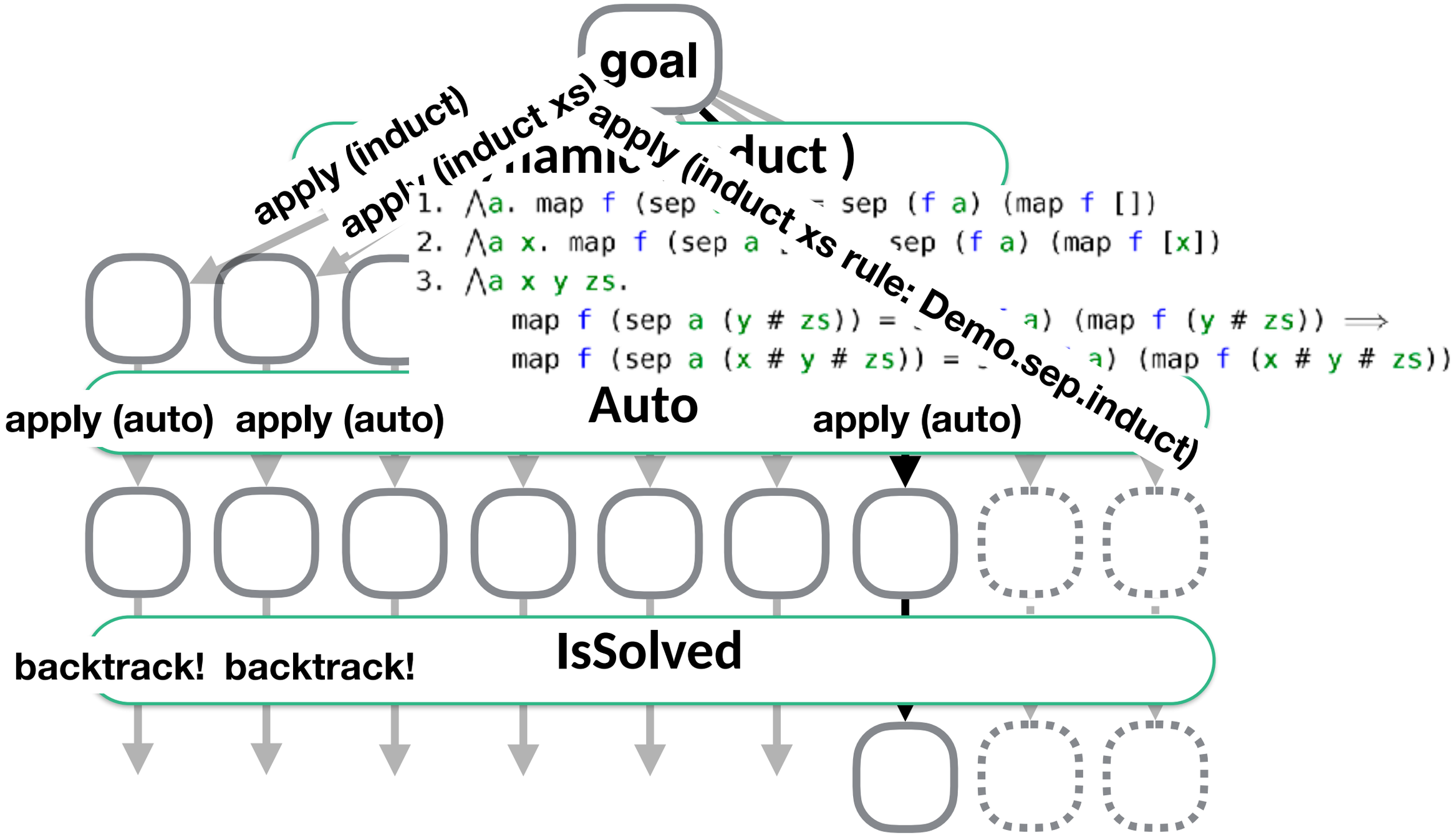
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

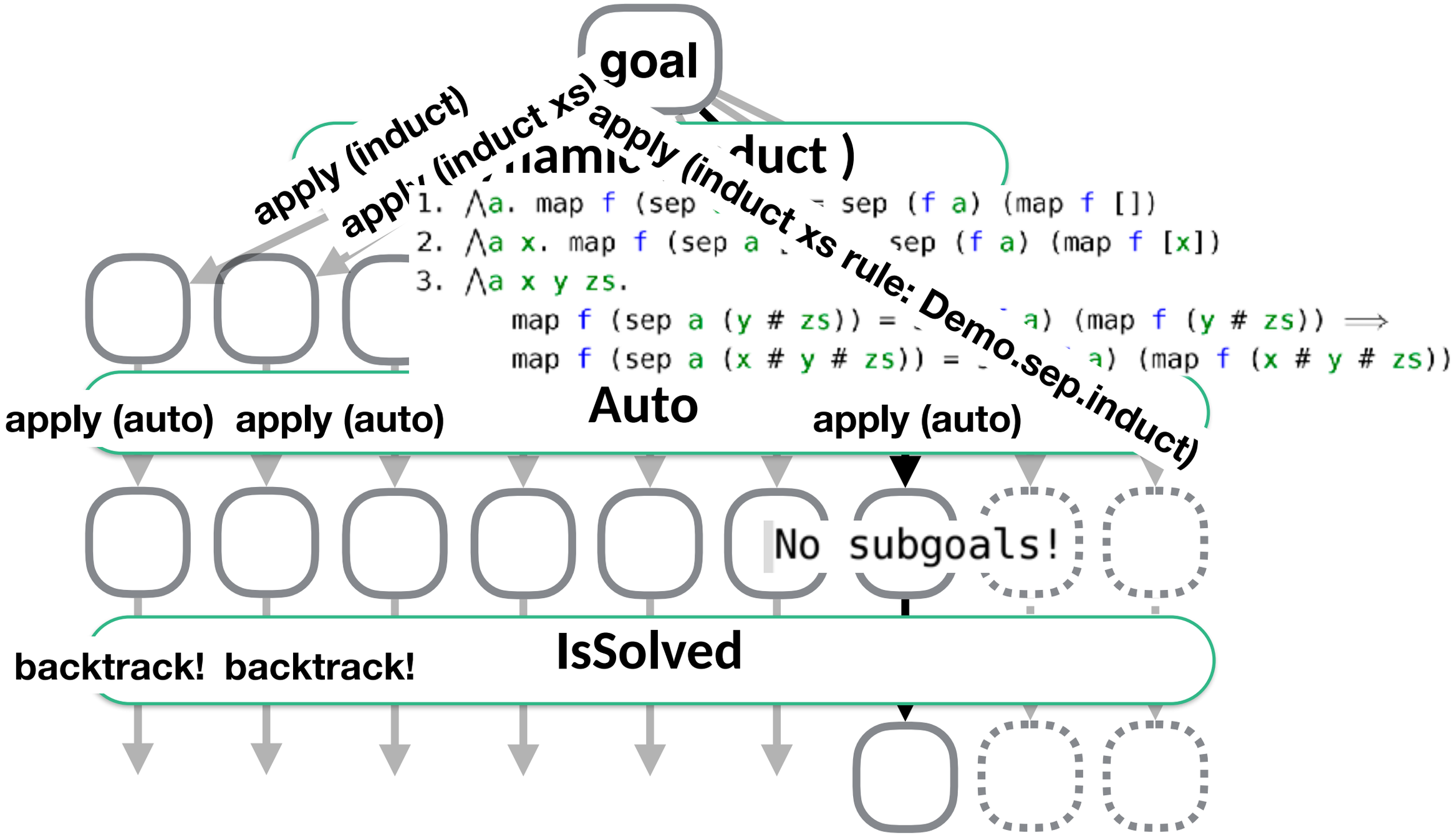
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

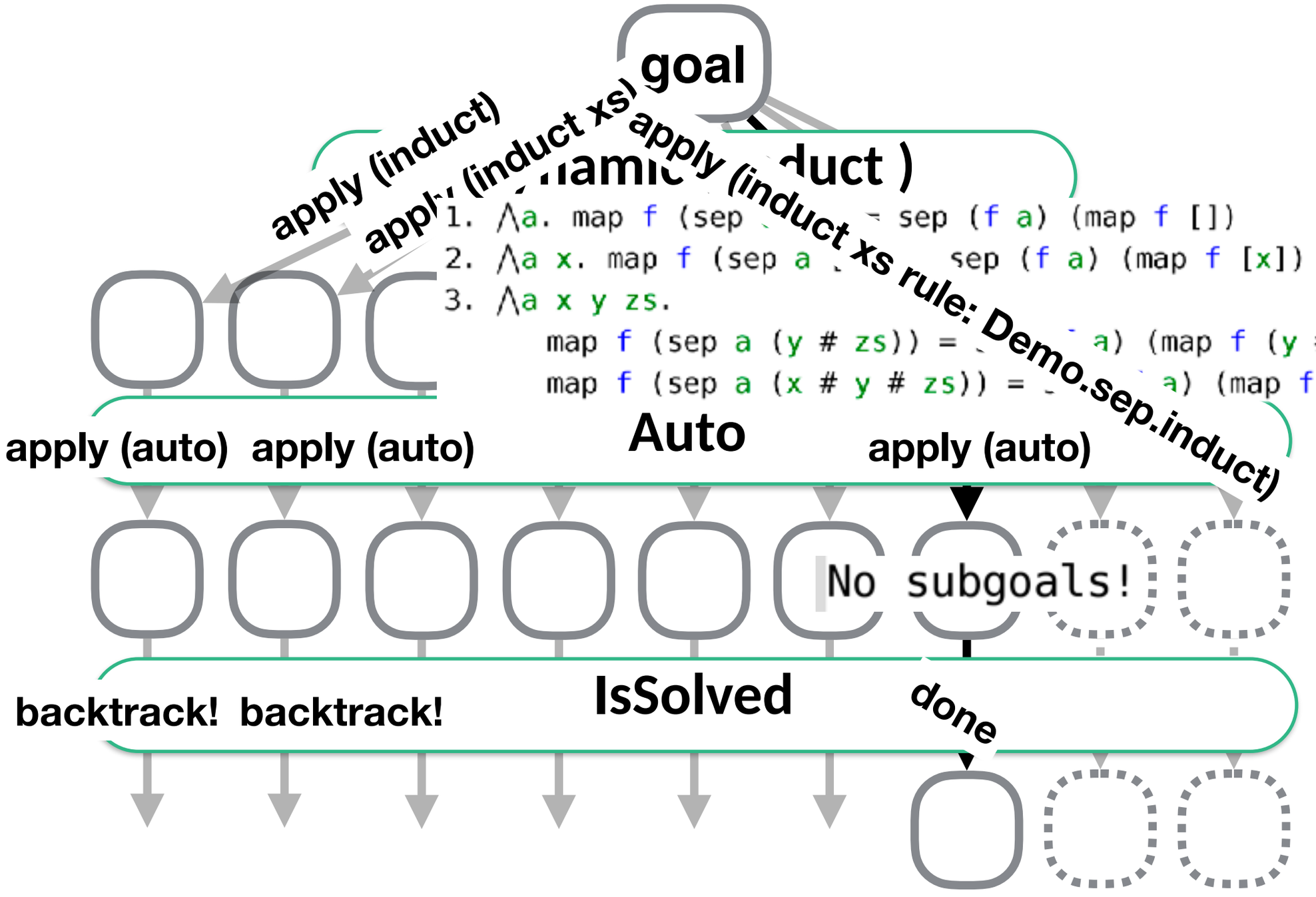
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

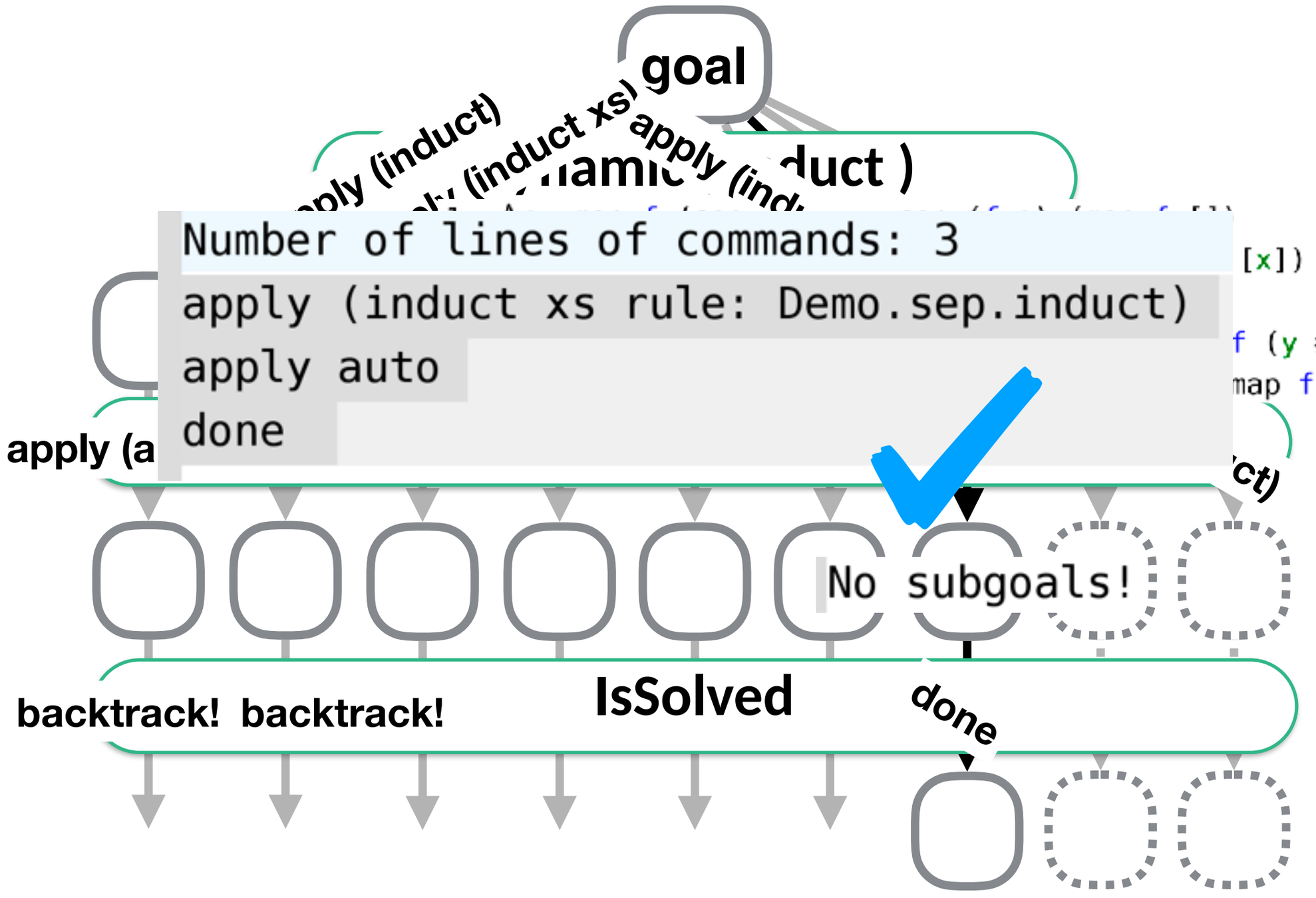
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

Previously at Master Seminar… I talked about PSL.

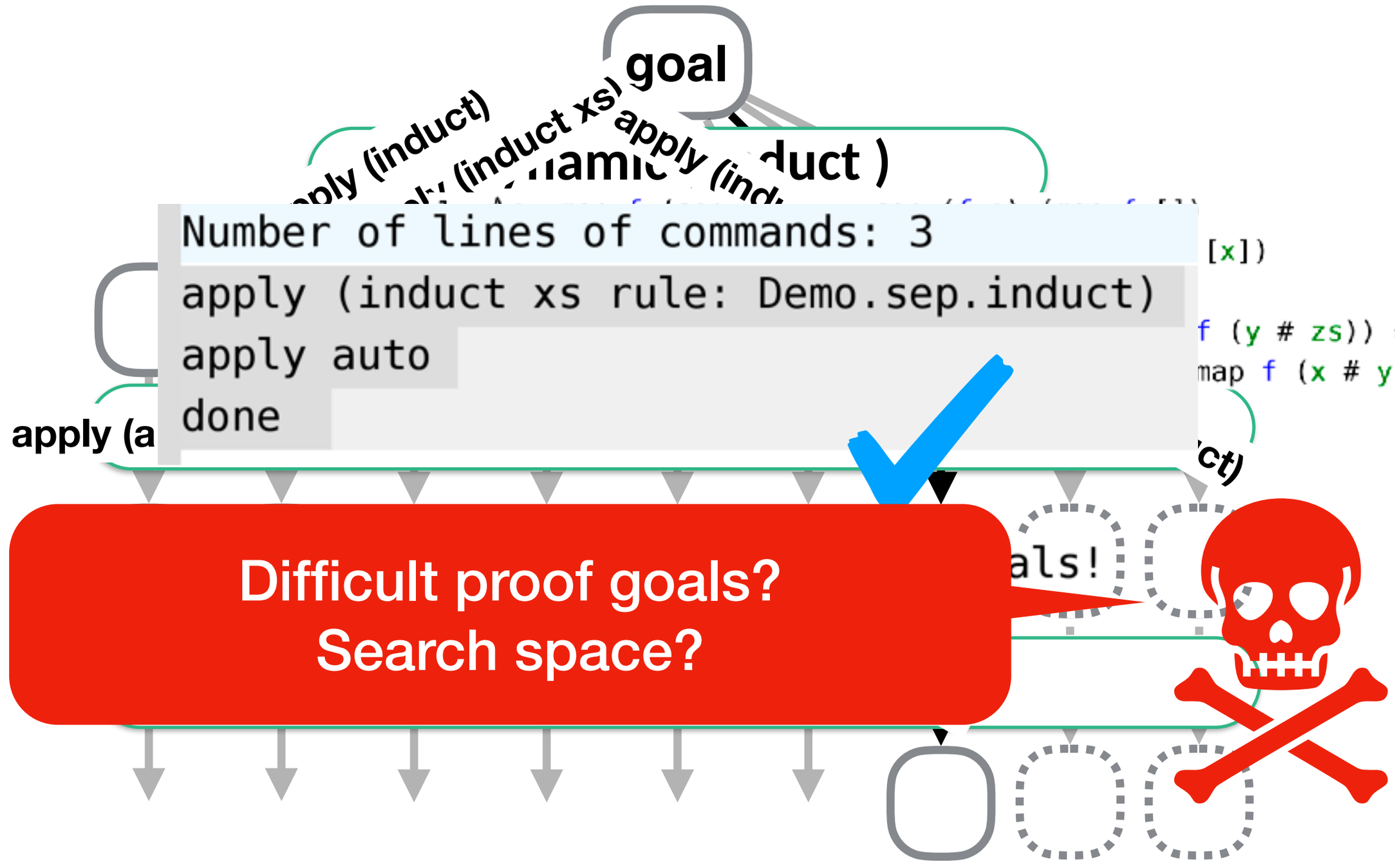
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE

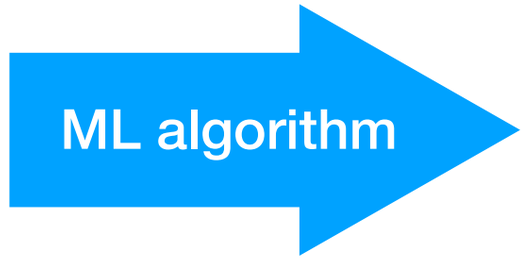
Previously at Master Seminar… I talked about PSL.

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



https://twitter.com/YutakangE

Introduction to Machine Learning in 10 seconds



https://duckduckgo.com/?q=cat&t=ffab&iar=images&iax=images&ia=images

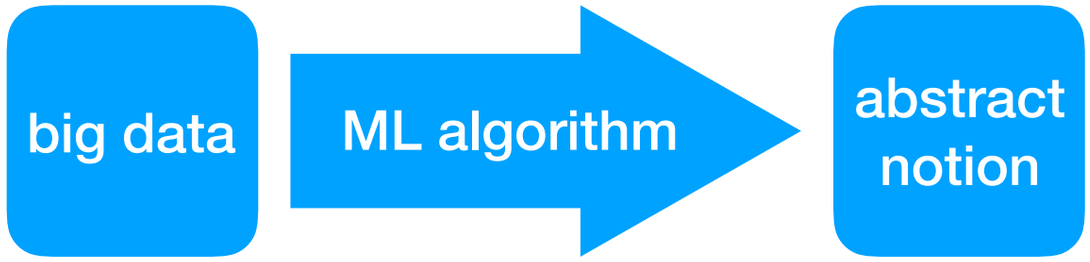
https://twitter.com/YutakangE

Introduction to Machine Learning in 10 seconds

https://duckduckgo.com/?q=cat&t=ffab&iar=images&iax=images&ia=images

https://twitter.com/YutakangE

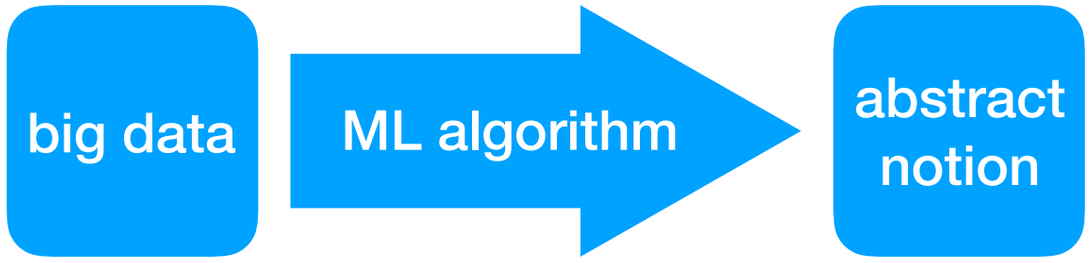
Introduction to Machine Learning in 10 seconds

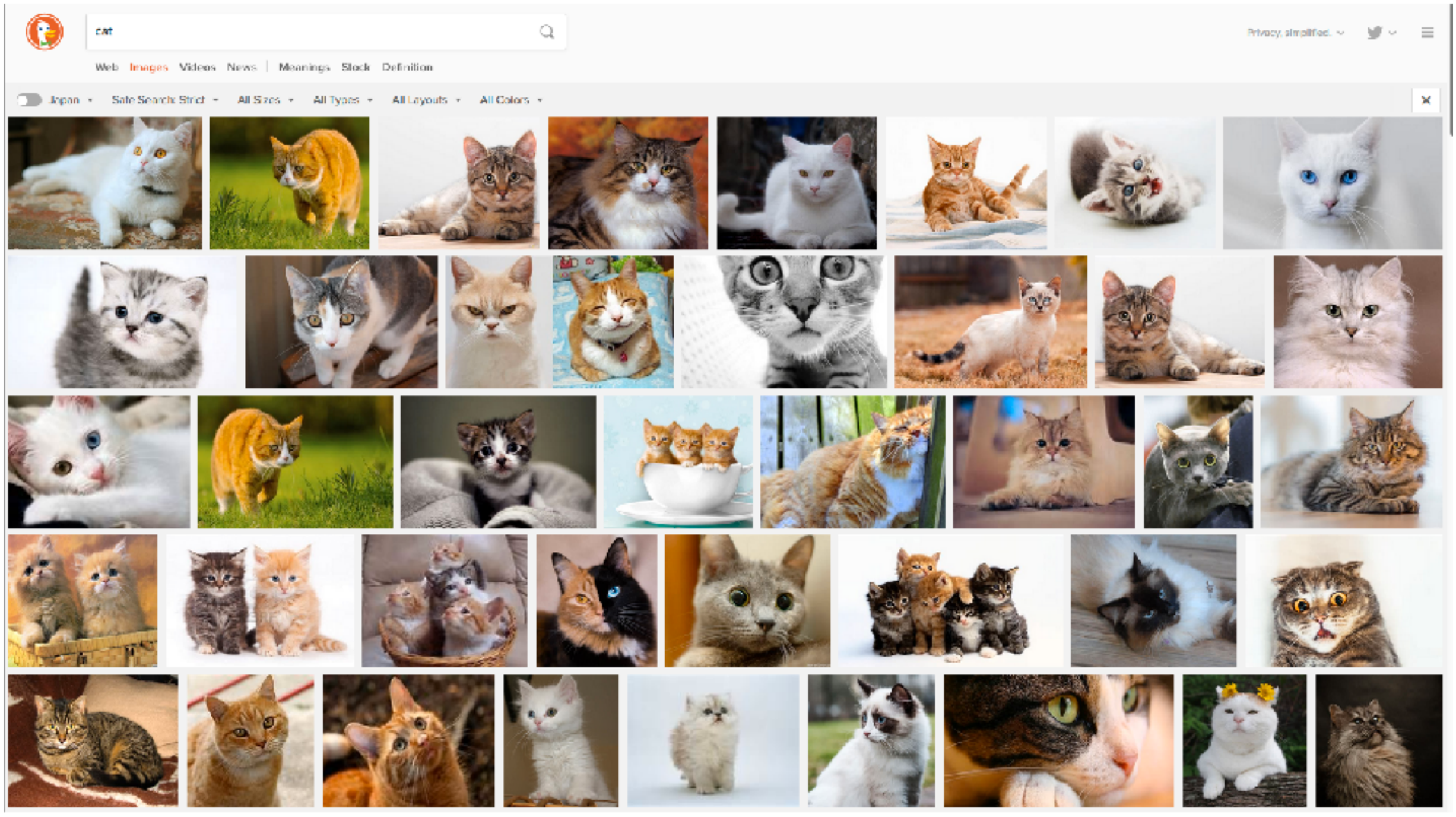


https://duckduckgo.com/?q=cat&t=ffab&iar=images&iax=images&ia=images

https://twitter.com/YutakangE

Introduction to Machine Learning in 10 seconds

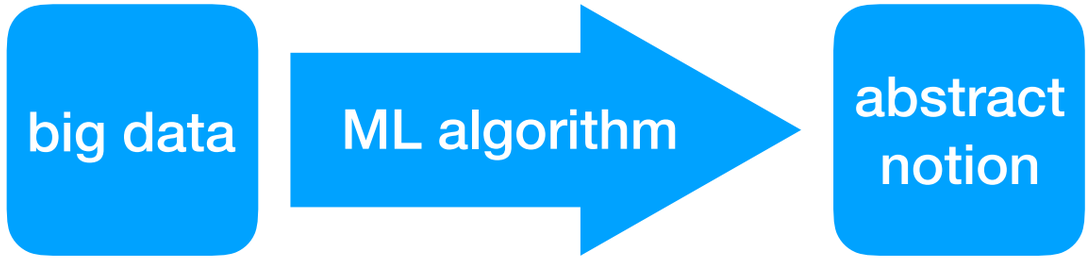


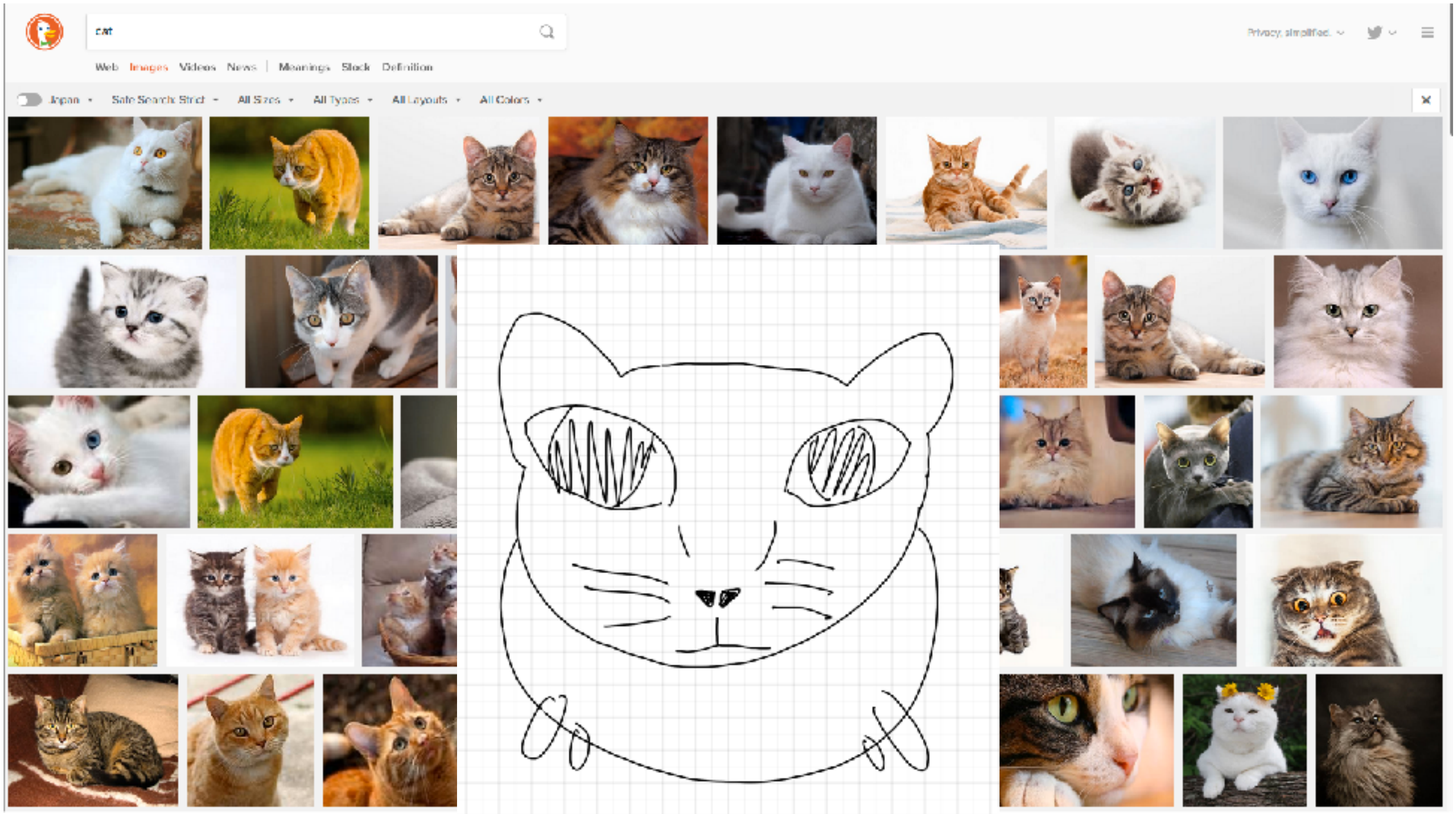


https://duckduckgo.com/?q=cat&t=ffab&iar=images&iax=images&ia=images

https://twitter.com/YutakangE

Introduction to Machine Learning in 10 seconds





https://duckduckgo.com/?q=cat&t=ffab&iar=images&iax=images&ia=images

https://twitter.com/YutakangE

ML for Inductive Theorem Proving the BAD

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

<- one abstract representation

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

<- many concrete cases

https://twitter.com/YutakangE

ML for Inductive Theorem Proving the BAD

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

<- one abstract representation

ML

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

<- many concrete cases

https://twitter.com/YutakangE

ML for Inductive Theorem Proving the BAD

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

<- one abstract representation

ML

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

<- many concrete cases

https://twitter.com/YutakangE

ML for Inductive Theorem Proving the BAD

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

<- one abstract representation

```
Failed to apply proof methodΔ:
goal (1 subgoal):
 1. itrev xs ys = rev xs @ ys
```

ML

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

<- many concrete cases

https://twitter.com/YutakangE

ML for Inductive Theorem Proving the BAD

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

<- one abstract representation

```
Failed to apply proof methodΔ:
goal (1 subgoal):
 1. itrev xs ys = rev xs @ ys
```

ML

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

<- many concrete cases

https://twitter.com/YutakangE

ML for Inductive Theorem Proving the BAD

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

<- one abstract representation


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

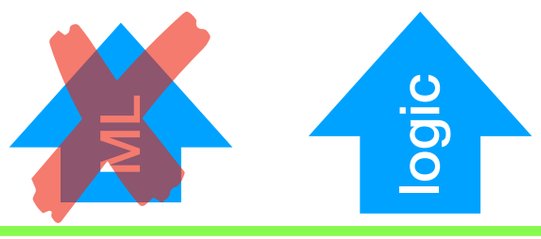
<- many concrete cases

https://twitter.com/YutakangE

ML for Inductive Theorem Proving the BAD

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

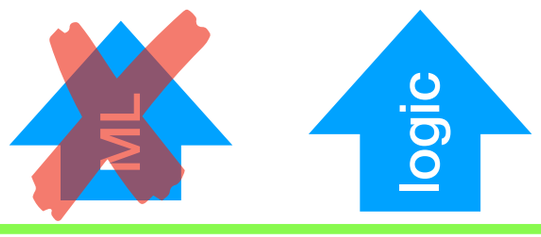
https://twitter.com/YutakangE

ML for Inductive Theorem Proving the BAD

polymorphism

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

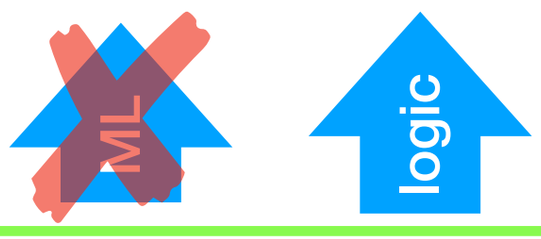
https://twitter.com/YutakangE

ML for Inductive Theorem Proving the BAD

polymorphism **type class**

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

https://twitter.com/YutakangE

ML for Inductive Theorem Proving the BAD

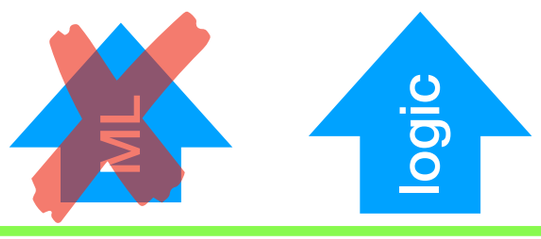
polymorphism

type class

universal quantifier

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

https://twitter.com/YutakangE

ML for Inductive Theorem Proving the BAD

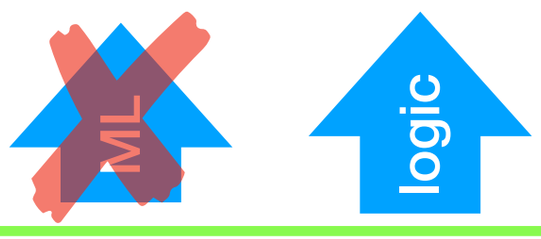
Higher-Order functions

polymorphism **type class**

universal quantifier

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

https://twitter.com/YutakangE

ML for Inductive Theorem Proving the BAD

Higher-Order functions

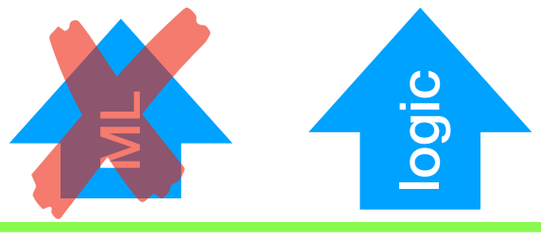
polymorphism **type class**

universal quantifier

lambda abstraction

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

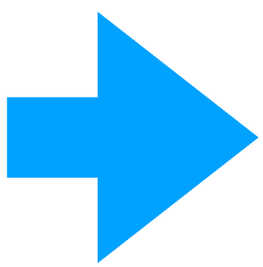
https://twitter.com/YutakangE

ML for Inductive Theorem Proving the BAD

Higher-Order functions

polymorphism **type class**

universal quantifier

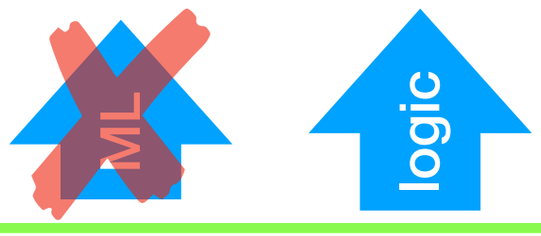


concise formula that can cover many concrete cases

lambda abstraction

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

https://twitter.com/YutakangE

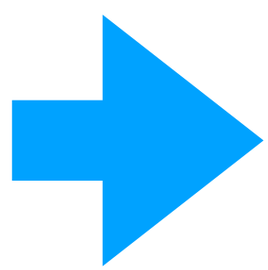
ML for Inductive Theorem Proving the BAD

Higher-Order functions

polymorphism

type class

universal quantifier



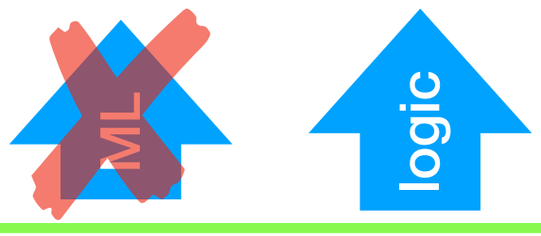
concise formula that can cover many concrete cases

different proof for general case

lambda abstraction

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

https://twitter.com/YutakangE

ML for Inductive Theorem Proving the BAD

Higher-Order functions

polymorphism

type class

universal quantifier

lambda abstraction

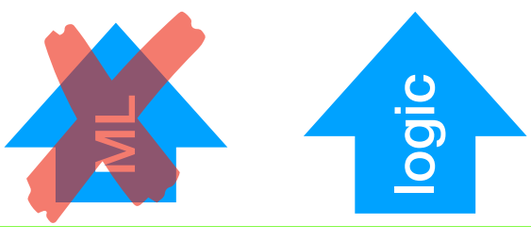
concise formula that can cover many concrete cases

different proof for general case

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

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

<- one abstract representation



<- abstraction using expressive logic

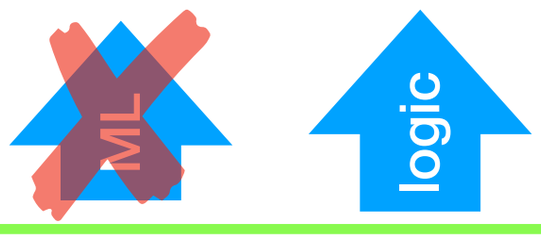
```
lemma "itrev [1,2]         [] = rev [1,2]         @ []" by auto
lemma "itrev [1,2,3]       [] = rev [1,2,3]       @ []" by auto
lemma "itrev [''a'',''b''] [] = rev [''a'',''b''] @ []" by auto
lemma "itrev [x,y,z]       [] = rev [x,y,z]       @ []" by auto
```

<- many concrete cases

https://twitter.com/YutakangE

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

<- one abstract representation



<- abstraction using expressive logic

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

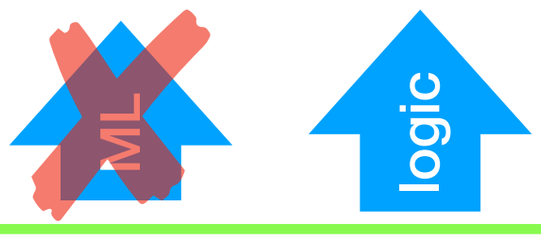
<- many concrete cases

https://twitter.com/YutakangE

Grand Challenge: Abstract Abstraction

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

https://twitter.com/YutakangE

Grand Challenge: Abstract Abstraction

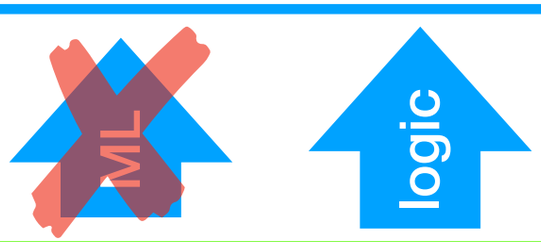
```
lemma "star r x y ⟹ star r y z ⟹ star r x z"
by(induction rule: star.induct)(auto simp: step)

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

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

<- small dataset about different domains

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

https://twitter.com/YutakangE

Grand Challenge: Abstract Abstraction

ML

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

<- small dataset about different domains

<- one abstract representation

~~ML~~

logic

<- abstraction using expressive logic

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

<- many concrete cases

https://twitter.com/YutakangE

Grand Challenge: Abstract Abstraction

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

<- even more abstract

ML

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

<- small dataset about different domains

<- one abstract representation

~~ML~~

logic

<- abstraction using expressive logic

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

<- many concrete cases

https://twitter.com/YutakangE

Grand Challenge: Abstract Abstraction

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

<- even more abstract

<- pros: good at ambiguity (heuristics)

ML

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

<- small dataset about different domains

<- one abstract representation

ML

logic

<- abstraction using expressive logic

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

<- many concrete cases

https://twitter.com/YutakangE

Grand Challenge: Abstract Abstraction

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

<- even more abstract

ML

<- pros: good at ambiguity (heuristics)

<- cons: bad at reasoning & 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 "itrev xs ys = rev xs @ ys"
by(induct xs ys rule:"itrev.induct") auto
```

<- small dataset about different domains

<- one abstract representation

ML

logic

<- abstraction using expressive logic

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

<- many concrete cases

https://twitter.com/YutakangE

Many key challenges remain

Unsupervised Learning

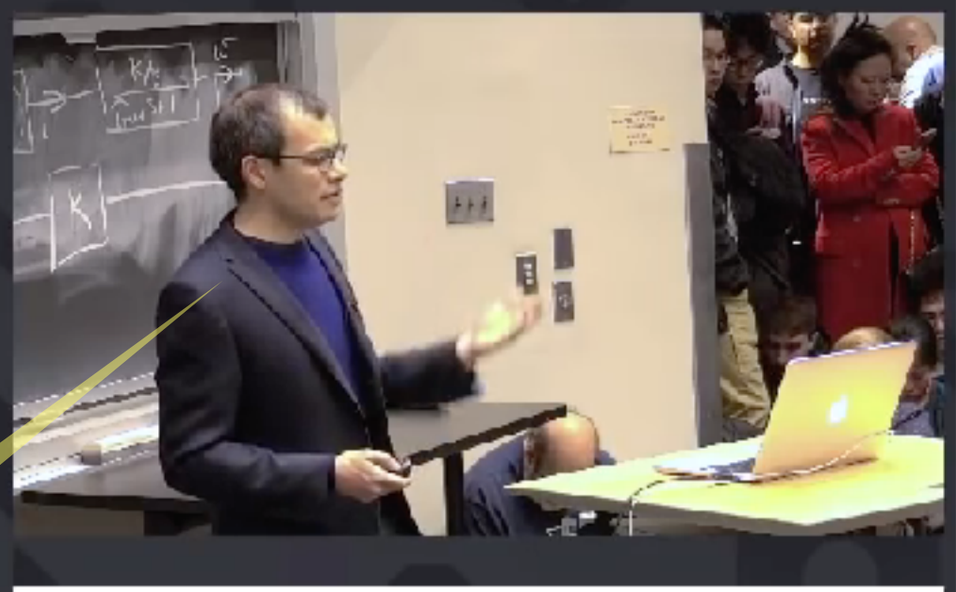
Memory and one-shot learning

Imagination-based Planning with Generative Models

Learning Abstract Concepts

Transfer Learning

Language understanding

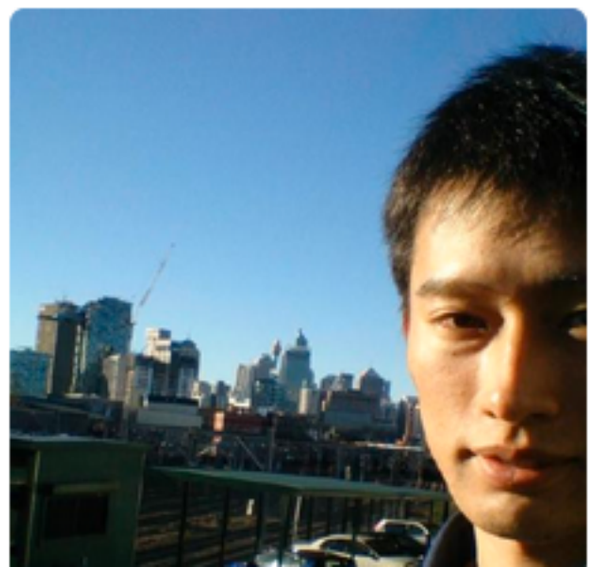


CENTER FOR
Brains
Minds+
Machines

March 20, 2019

The Power of

Self



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

https://twitter.com/YutakangE

Many key challenges remain

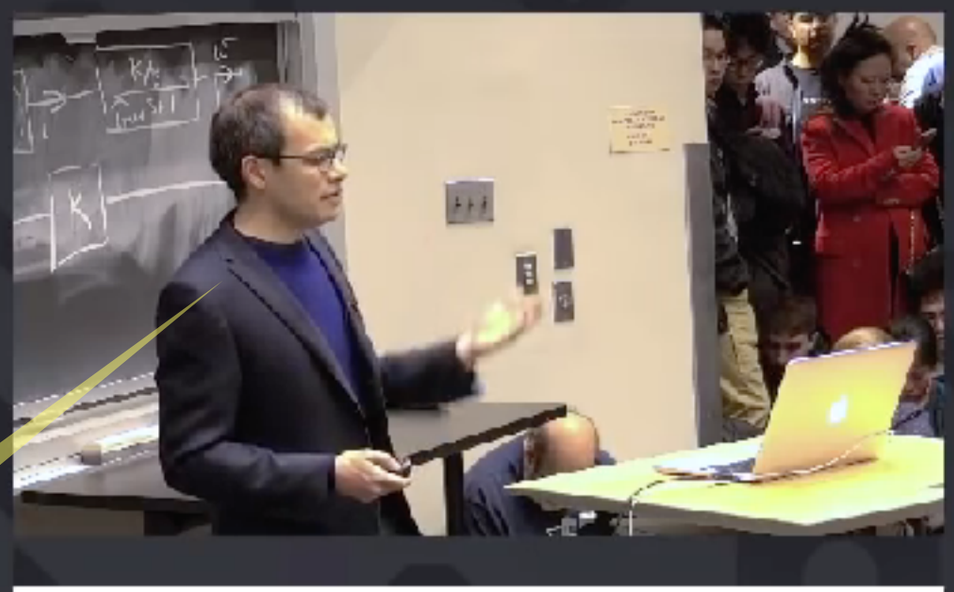
Unsupervised Learning

Memory and one-shot learning

Imagination-based Planning with Generative Models

Learning Abstract Concepts

Abstract concepts?

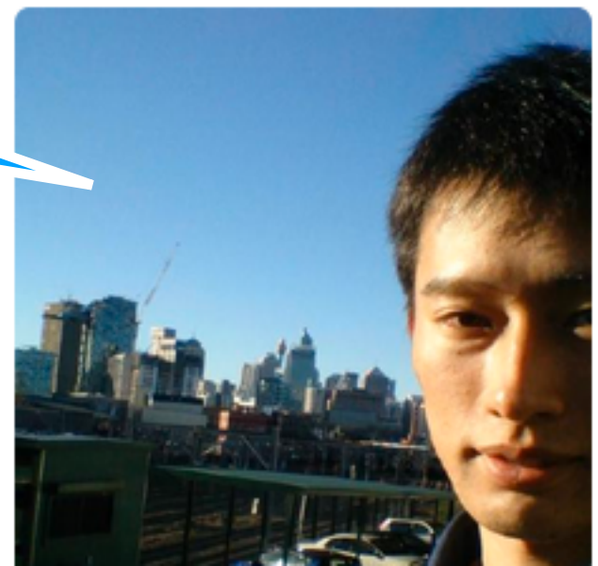


CENTER FOR
Brains
Minds+
Machines

March 20, 2019

The Power of

Self



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

https://twitter.com/YutakangE

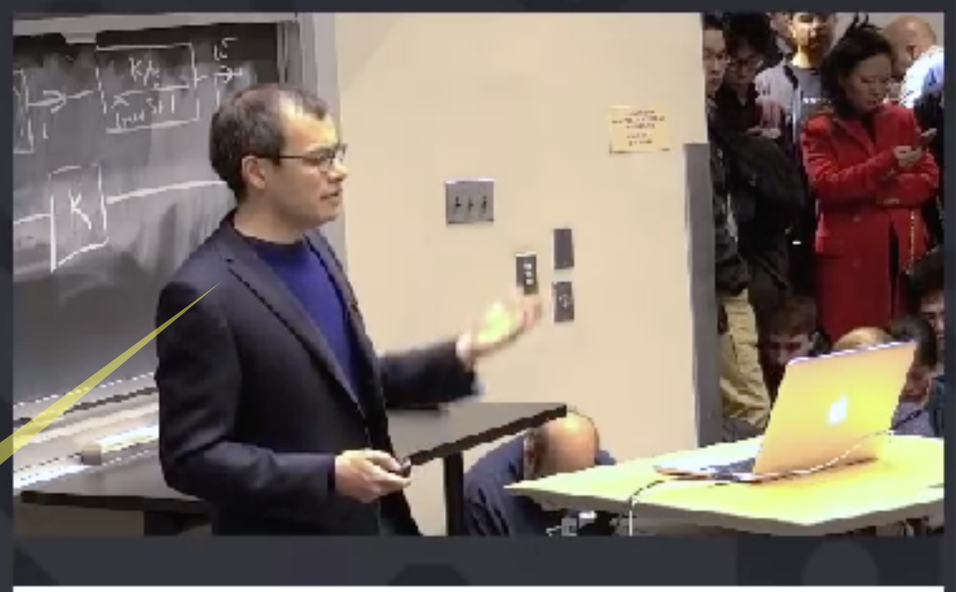
Many key challenges remain

Unsupervised Learning

Memory and one-shot learning

Imagination-based Planning with Generative Models

Learning Abstract Concepts



CENTER FOR
Brains
Minds+
Machines

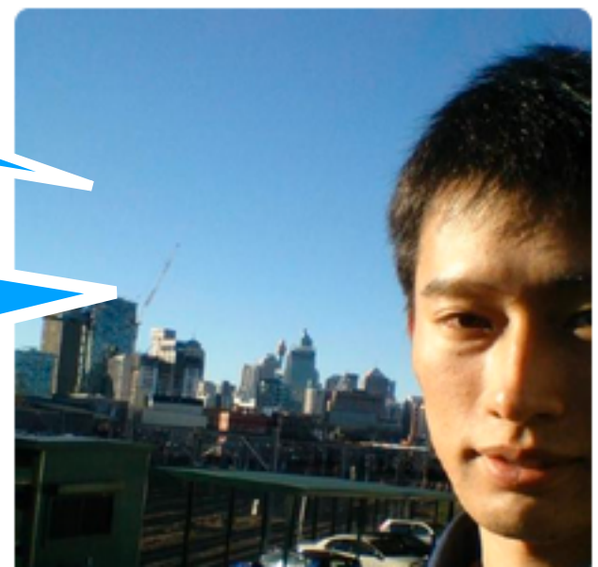
March 20, 2019

Abstract concepts?

why not logic? (LiFtEr)

The Power of

Self

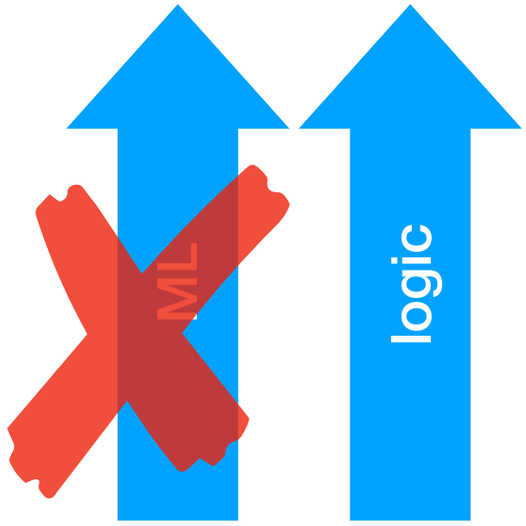


https://cbmm.mit.edu/video/power-self-learning-systems

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

<- even more abstract

Big Picture



abstraction using
another logic (LiFtEr)

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

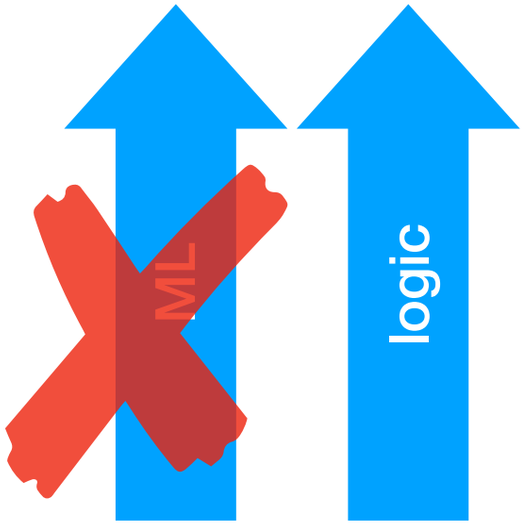
```
lemma "itrev [1,2]         [] = rev [1,2]         @ []" by auto
lemma "itrev [1,2,3]       [] = rev [1,2,3]       @ []" by auto
lemma "itrev [''a'',''b''] [] = rev [''a'',''b''] @ []" by auto
lemma "itrev [x,y,z]       [] = rev [x,y,z]       @ []" by auto
```

<- many concrete cases

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

<- even more abstract

Big Picture



abstraction using
another logic (LiFtEr)

<- pros: good at rigorous abstraction

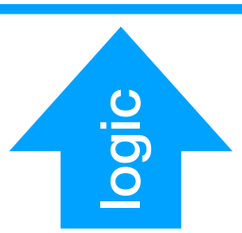
```
lemma "star r x y ⟹ star r y z ⟹ star r x z"
by(induction rule: star.induct)(auto simp: step)

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

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

<- small dataset about different domains

<- one abstract representation



<- abstraction using expressive logic

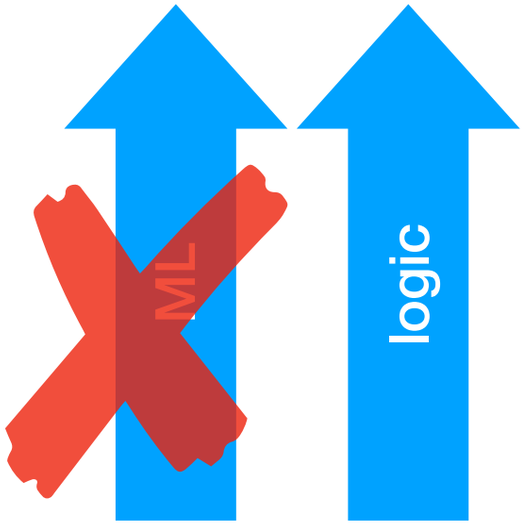
```
lemma "itrev [1,2]         [] = rev [1,2]         @ []" by auto
lemma "itrev [1,2,3]       [] = rev [1,2,3]       @ []" by auto
lemma "itrev [''a'',''b''] [] = rev [''a'',''b''] @ []" by auto
lemma "itrev [x,y,z]       [] = rev [x,y,z]       @ []" by auto
```

<- many concrete cases

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

<- even more abstract

Big Picture



abstraction using another logic (LiFtEr)

<- pros: good at rigorous abstraction
<- cons: bad at ambiguity (heuristics)

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Big Picture

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

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

LiFtEr

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Big Picture

LiFtEr

<- pros: good at rigorous 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 "itrev xs ys = rev xs @ ys"
by(induct xs ys rule:"itrev.induct") auto
```

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

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

ML

LiFtEr

<- pros: good at rigorous 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
```

<- small dataset about different domains

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

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

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

ML

<- pros: good at ambiguity (heuristics)

LiFtEr

<- pros: good at rigorous 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
```

<- small dataset about different domains

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

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

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

ML

<- pros: good at ambiguity (heuristics)

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

LiFtEr

<- pros: good at rigorous 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 "itrev xs ys = rev xs @ ys"
by(induct xs ys rule:"itrev.induct") auto
```

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

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

ML

<- pros: good at ambiguity (heuristics)

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

<- pros: good at rigorous 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 "itrev xs ys = rev xs @ ys"
by(induct xs ys rule:"itrev.induct") auto
```

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

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

ML

<- pros: good at ambiguity (heuristics)

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

LiFtEr 1

<- pros: good at rigorous 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 "itrev xs ys = rev xs @ ys"
by(induct xs ys rule:"itrev.induct") auto
```

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

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

ML

<- pros: good at ambiguity (heuristics)

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

LiFtEr

LiFtEr 2

<- pros: good at rigorous 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 "itrev xs ys = rev xs @ ys"
by(induct xs ys rule:"itrev.induct") auto
```

<- small dataset about different domains

<- one abstract representation

logic

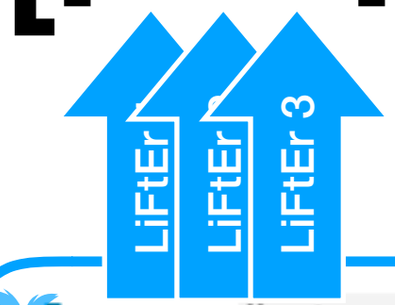
<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

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



[[ ], [ ], [ ]]: bool list

<- pros: good at ambiguity (heuristics)

<- simple representation

<- pros: good at rigorous abstraction

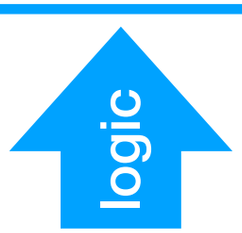
```
lemma "star r x y ⟹ star r y z ⟹ star r x z"
by(induction rule: star.induct)(auto simp: step)

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

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

<- small dataset about different domains

<- one abstract representation



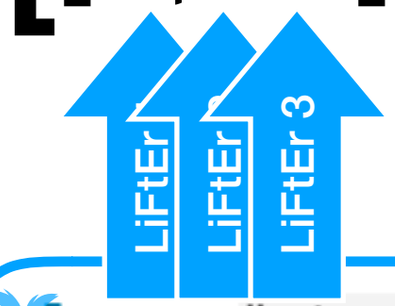
<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

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



[[T,], [ ], [ ]]: bool list

<- pros: good at ambiguity (heuristics)

<- simple representation

<- pros: good at rigorous abstraction

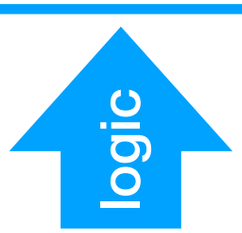
```
lemma "star r x y ⟹ star r y z ⟹ star r x z"
by(induction rule: star.induct)(auto simp: step)

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

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

<- small dataset about different domains

<- one abstract representation



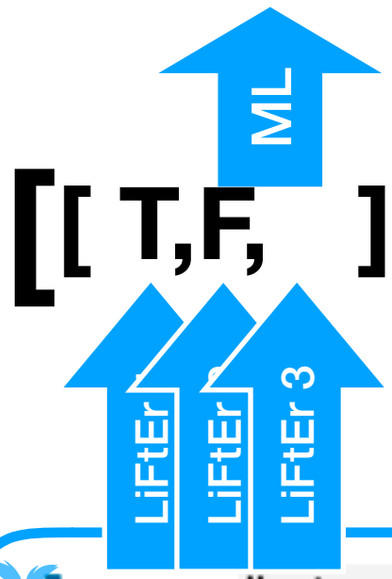
<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

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



[[T,F,], [ ], [ ]]: bool list

<- pros: good at ambiguity (heuristics)

<- simple representation

<- pros: good at rigorous abstraction

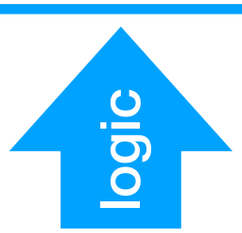
```
lemma "star r x y ⟹ star r y z ⟹ star r x z"
by(induction rule: star.induct)(auto simp: step)

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

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

<- small dataset about different domains

<- one abstract representation



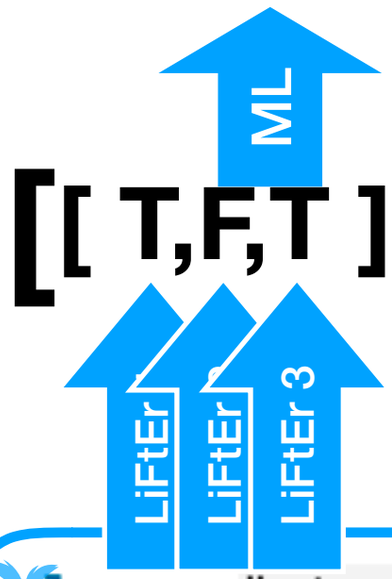
<- abstraction using expressive logic

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

<- many concrete cases

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

Big Picture



<- pros: good at ambiguity (heuristics)

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

<- pros: good at rigorous 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 "itrev xs ys = rev xs @ ys"
by(induct xs ys rule:"itrev.induct") auto
```

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Big Picture

<- pros: good at ambiguity (heuristics)

ML

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

LiFtEr 3 LiFtEr 2 LiFtEr 1 LiFtEr 3 LiFtEr 2 LiFtEr 1

<- pros: good at rigorous abstraction

```
lemma "star r x y ∧ star r y z ⟹ star r x z"
by(induction r 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 "itrev xs ys = rev xs @ ys"
by(induct xs ys rule:"itrev.induct") auto
```

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

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

ML

<- pros: good at ambiguity (heuristics)

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

LiFtEr

LiFtEr

LiFtEr 3

LiFtEr 1

LiFtEr 2

LiFtEr 3

<- pros: good at rigorous abstraction

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

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

<- small dataset about different domains

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

<- one abstract representation

logic

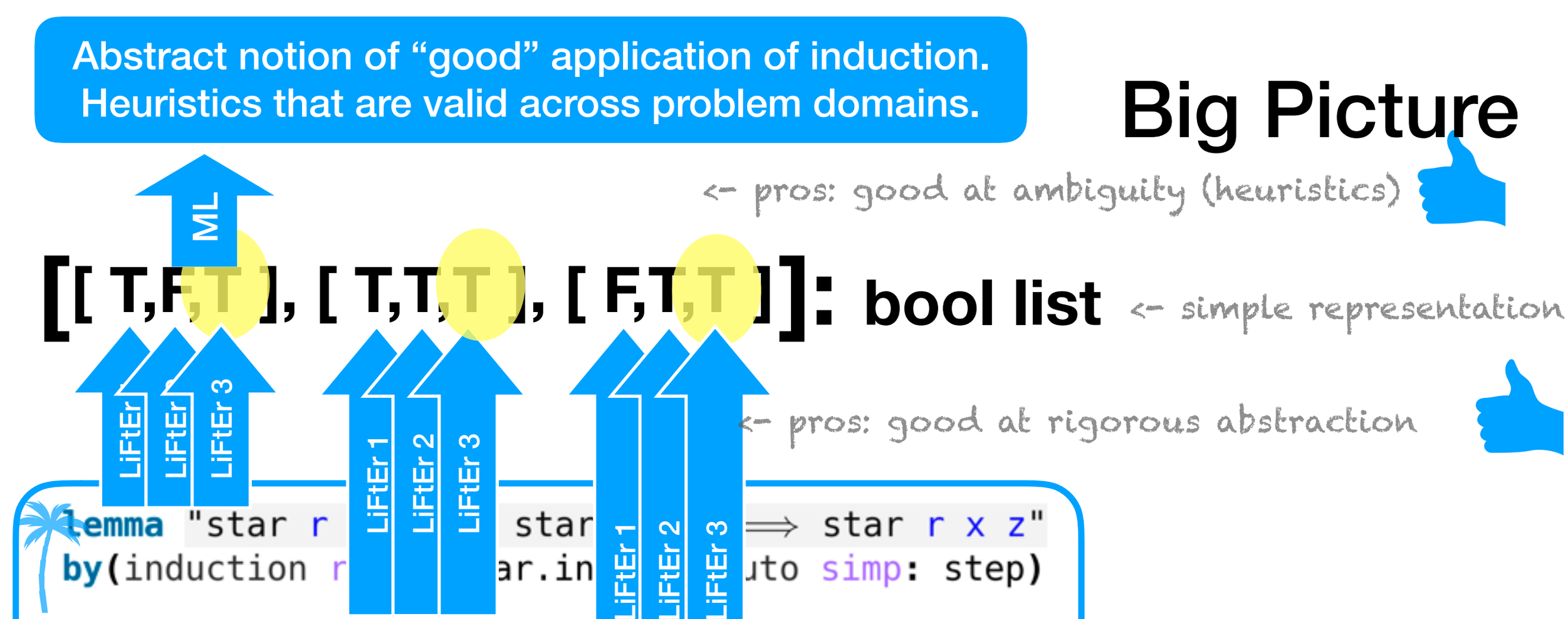
<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

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



Big Picture

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

ML

<- pros: good at ambiguity (heuristics)

[[T,F,T], [T,T,T], [F,T,T]]: bool list

<- simple representation

LiFtEr

LiFtEr

LiFtEr 3

<- pros: good at rigorous abstraction

LiFtEr 1

LiFtEr 2

LiFtEr 3

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

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

example?

<- small dataset about different domains

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

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

https://twitter.com/YutakangE

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
```

https://twitter.com/YutakangE

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
```

https://twitter.com/YutakangE

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

https://twitter.com/YutakangE

Example Assertion in LiFtEr (in Abstract Syntax)

implication

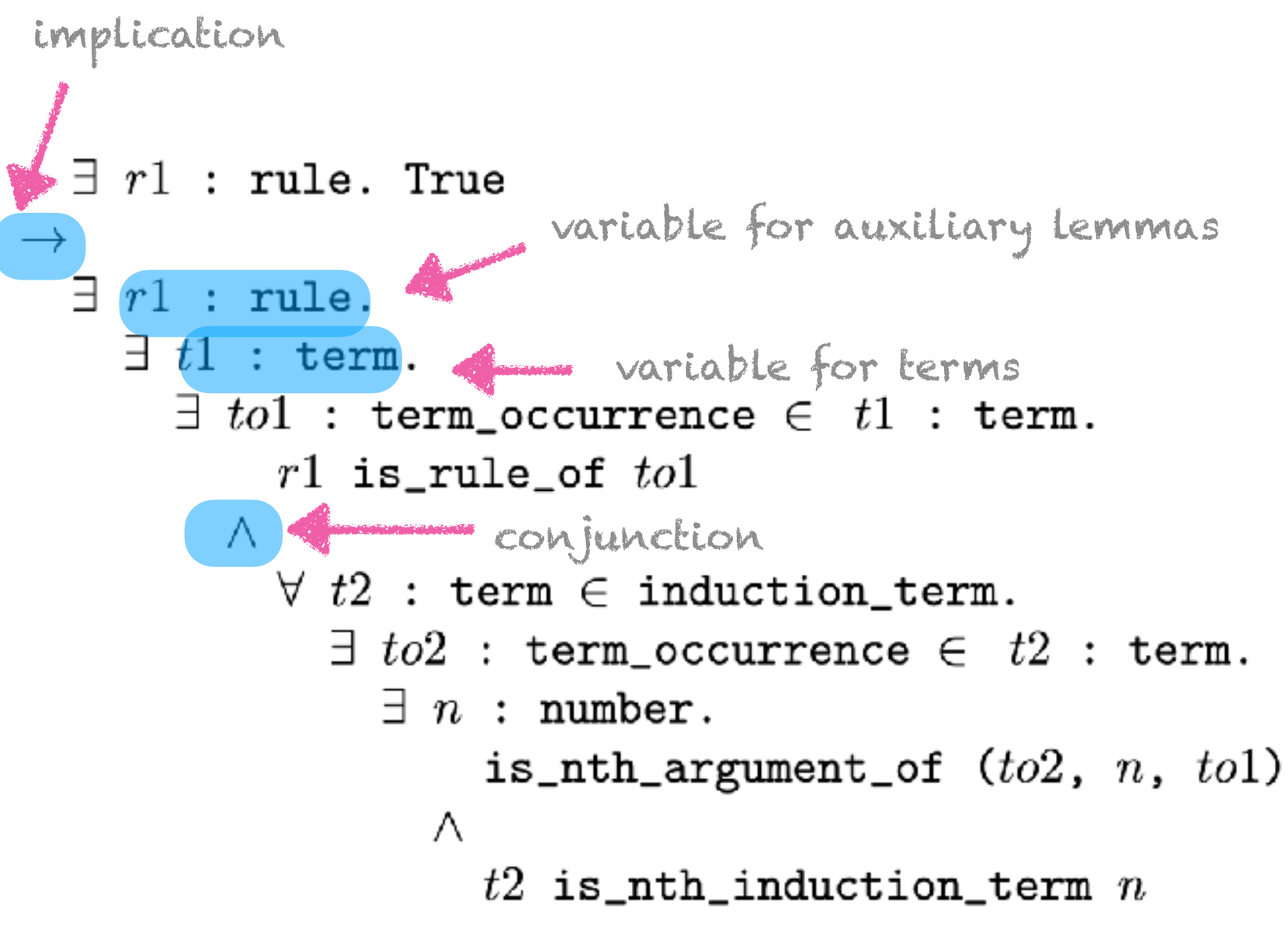
```
∃ r1 : rule. True
→
   ∃ r1 : rule.
     ∃ t1 : term.
       ∃ to1 : term_occurrence ∈ t1 : term.
           r1 is_rule_of to1
         ∧
           ∀ t2 : term ∈ induction_term.
             ∃ to2 : term_occurrence ∈ t2 : term.
               ∃ n : number.
                   is_nth_argument_of (to2,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

variable for auxiliary lemmas

conjunction

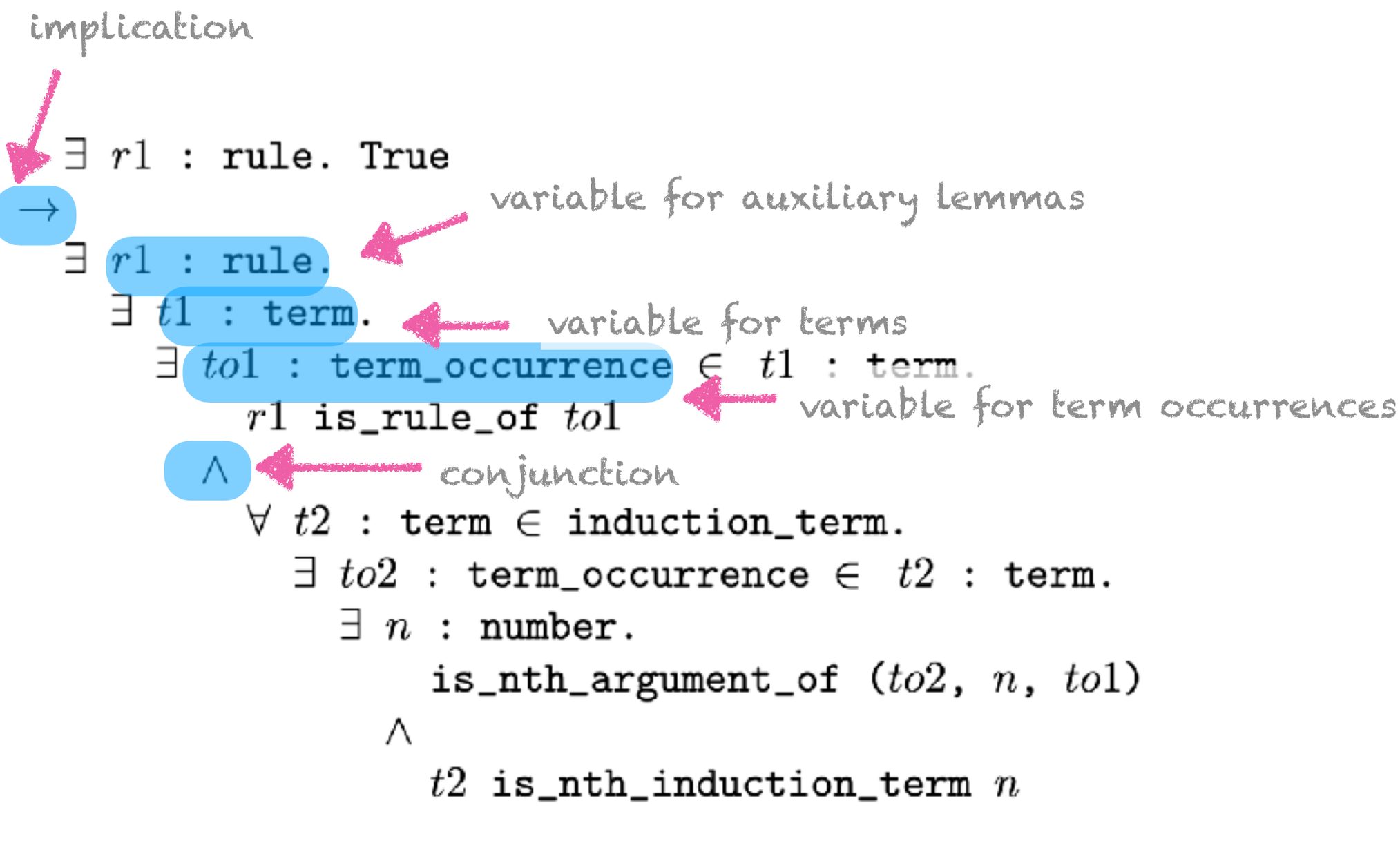
https://twitter.com/YutakangE

Example Assertion in LiFtEr (in Abstract Syntax)



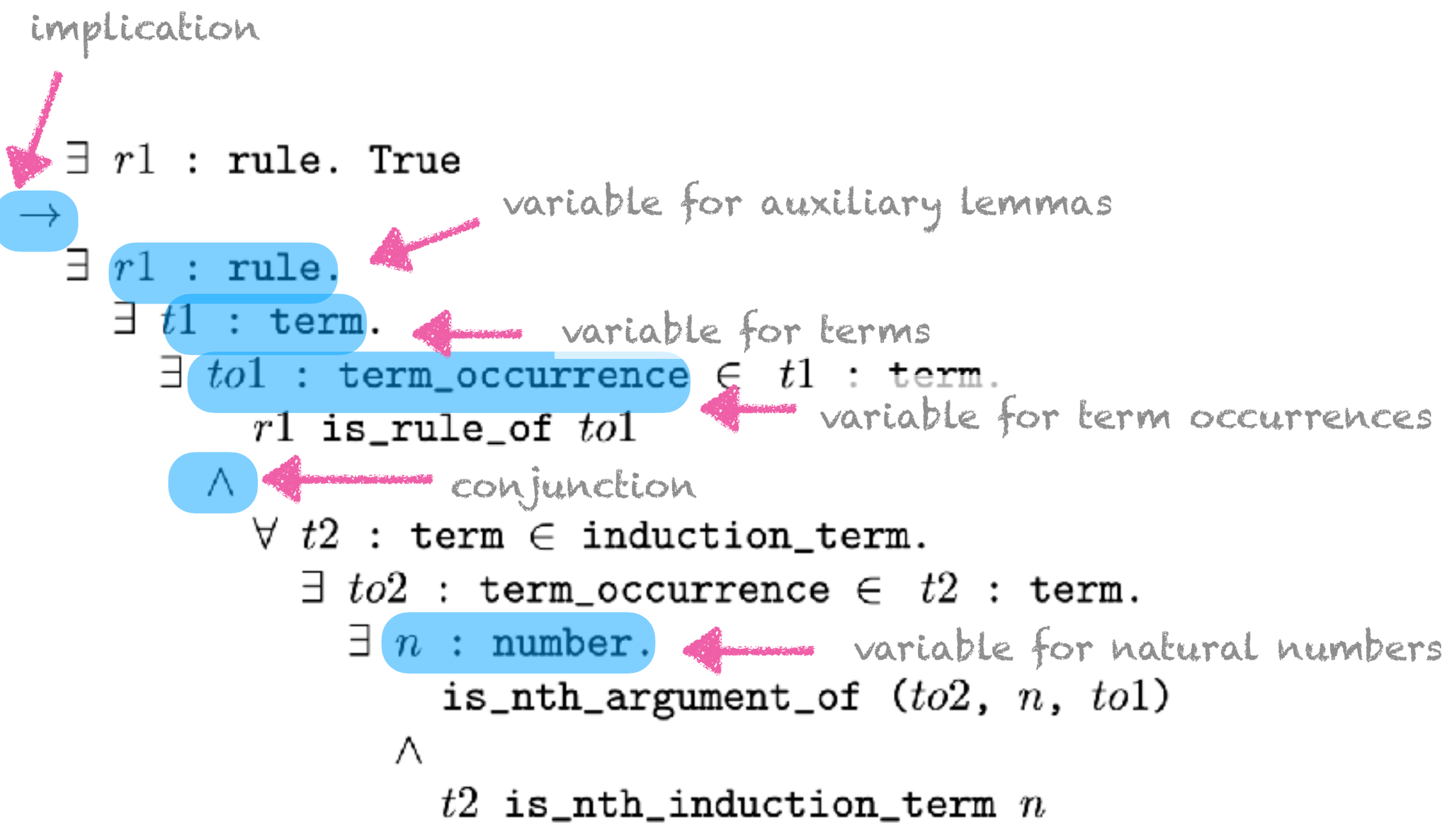
https://twitter.com/YutakangE

Example Assertion in LiFtEr (in Abstract Syntax)



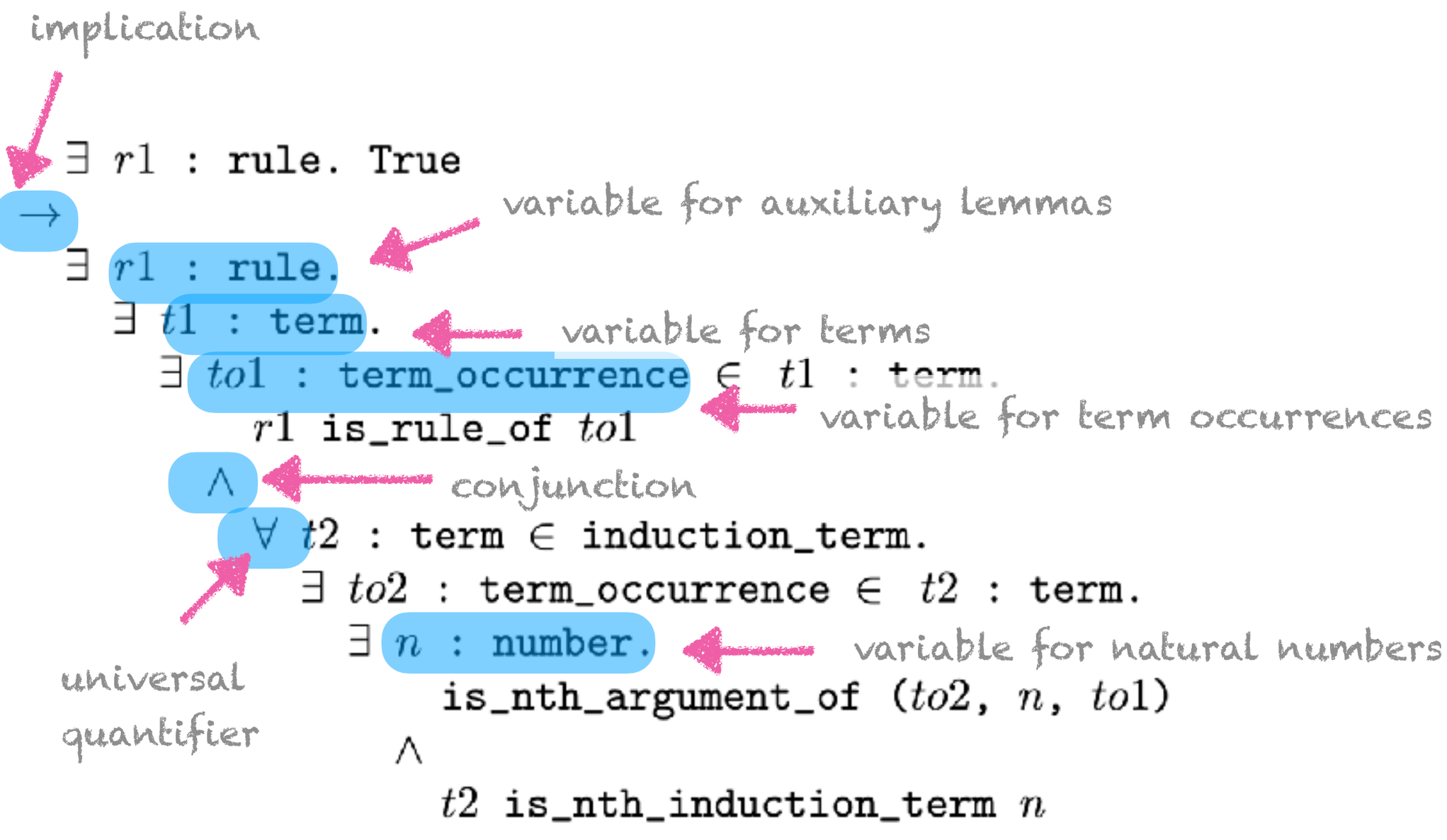
https://twitter.com/YutakangE

Example Assertion in LiFtEr (in Abstract Syntax)



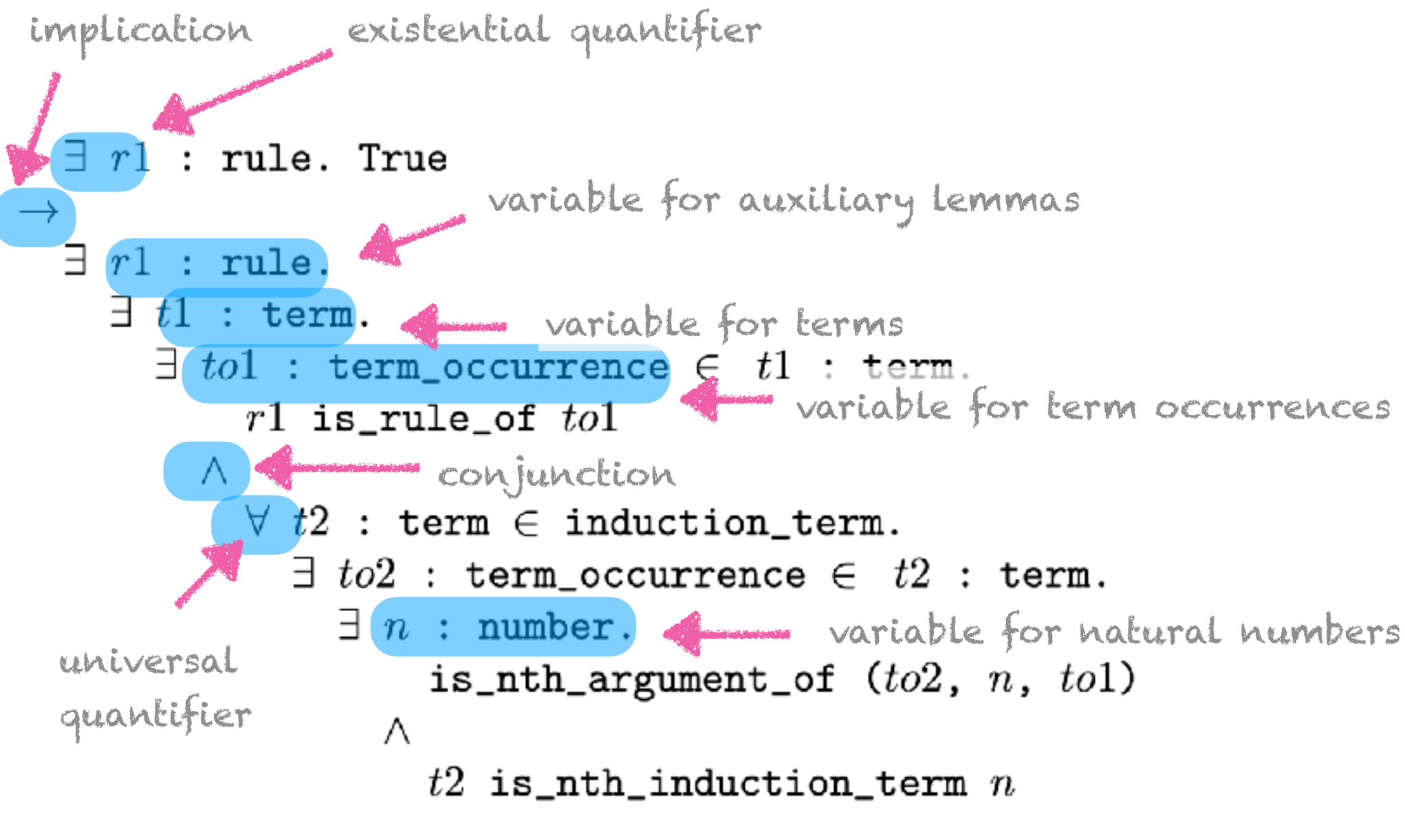
https://twitter.com/YutakangE

Example Assertion in LiFtEr (in Abstract Syntax)



https://twitter.com/YutakangE

Example Assertion in LiFtEr (in Abstract Syntax)



https://twitter.com/YutakangE

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

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

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

$\exists$ $r1$: 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

https://twitter.com/YutakangE

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

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

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

```
∃ r1 : rule. True
```

→ r1

(r1 = itrev.induct)

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

https://twitter.com/YutakangE

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

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

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

∃ r1 : rule. True

→ r1

∃ r1 : rule.

(r1 = itrev.induct)

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

https://twitter.com/YutakangE

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

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

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

```
∃ r1 : rule. True
```

→ r1

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

(r1 = itrev.induct)

https://twitter.com/YutakangE

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

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

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

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

t1 r1

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

https://twitter.com/YutakangE

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

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

to1

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

t1 r1

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

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

https://twitter.com/YutakangE

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

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

to1

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

t1 r1

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

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

https://twitter.com/YutakangE

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

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

to1

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

t1 r1

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

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

https://twitter.com/YutakangE

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

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

to1

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

t1 r1

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

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

https://twitter.com/YutakangE

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

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

to1

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

t1 r1

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

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

https://twitter.com/YutakangE

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

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

to1

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

t1 r1

```
∃ r1 : rule. True
```

→

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

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

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

https://twitter.com/YutakangE

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

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

to1

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

t2

t1 r1

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

https://twitter.com/YutakangE

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

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

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

to1 to2 t2 t1 r1

∃ r1 : rule. True

→

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

https://twitter.com/YutakangE

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

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

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

to1 to2 t2 t1 r1

∃ r1 : rule. True

→

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

https://twitter.com/YutakangE

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

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

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

to1 to2 t2 t1 r1

∃ r1 : rule. True

→

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

https://twitter.com/YutakangE

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

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

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

to1 first to2 t2 first t1 r1

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

https://twitter.com/YutakangE

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

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

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

to1 first to2 t2 first t1 r1

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

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

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

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

when t2 is xs (n = 1)

https://twitter.com/YutakangE

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

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

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

to1 first second to2 t2 second first t1 r1

∃ r1 : rule. True

→

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

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

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

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

when t2 is xs (n = 1) ?

when t2 is ys (n = 2) ?

https://twitter.com/YutakangE

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

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

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

to1 first second to2

t2 second t1 r1

first

∃ r1 : rule. True

→

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

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

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

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

when t2 is xs (n = 1)

when t2 is ys (n = 2)

https://twitter.com/YutakangE

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

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

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

to1 first second to2 t2 second first t1 r1

∃ r1 : rule. True

→

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

$\exists$ $r1$: rule. True

$\rightarrow$

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

the same LiFtEr assertion

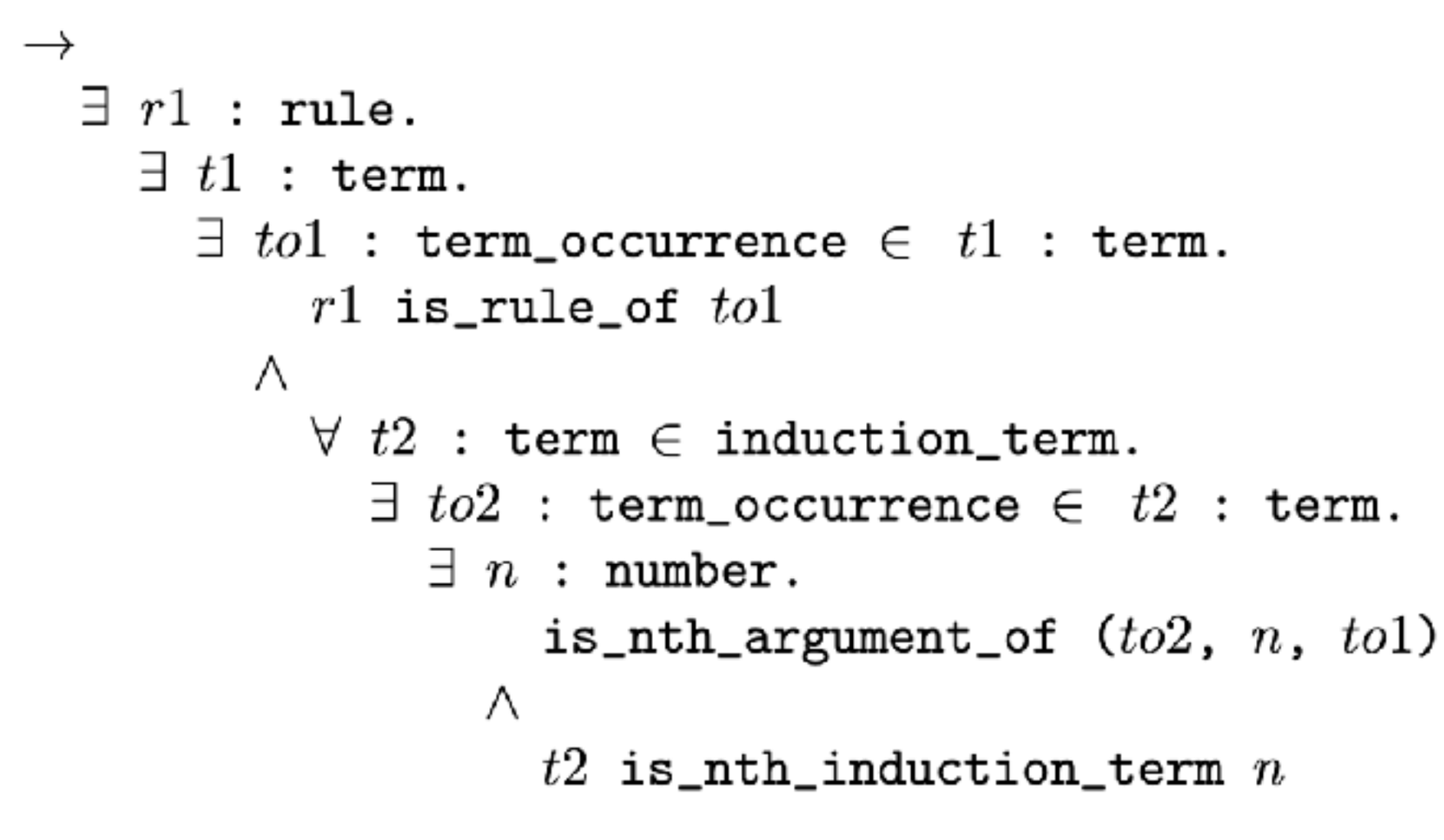
```
∃ r1 : rule. True
→
  ∃ r1 : rule.
    ∃ t1 : term.
      ∃ to1 : term_occurrence ∈ t1 : term.
          r1 is_rule_of to1
        ∧
          ∀ t2 : term ∈ induction_term.
            ∃ to2 : term_occurrence ∈ t2 : term.
              ∃ n : number.
                  is_nth_argument_of (to2, n, to1)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

new types ->

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

the same LiFtEr assertion

```
∃ r1 : rule. True
```



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

new types ->

new constants ->

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

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

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

the same LiFtEr assertion

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

new types ->

new constants ->

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

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

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

the same LiFtEr assertion

new lemma ->

a model proof ->

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

∃ r1 : rule. True

→

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

new types ->

new constants ->

new lemma ->

a model proof ->

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

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

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

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

∃ r1 : rule. True

→

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

new types ->

new constants ->

new lemma ->

a model proof ->

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

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

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

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

∃ r1 : rule. True

r1

(r1 = exec.induct)

→

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

new types ->

new constants ->

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

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

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

new lemma ->
a model proof ->

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

∃ r1 : rule. True

→

r1

(r1 = exec.induct)

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

new types ->

new constants ->

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

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

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

new lemma ->

a model proof ->

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

∃ r1 : rule. True

→

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

r1

(r1 = exec.induct)

new types ->

```
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

→

t1 r1

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

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

new types ->

new constants ->

new lemma ->

a model proof ->

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

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

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

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

to1

t1 r1

∃ r1 : rule. True

→

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

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

new types ->

new constants ->

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

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

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

new lemma ->

a model proof ->

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

to1

t1 r1

∃ r1 : rule. True

→

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

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

new types ->

new constants ->

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

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

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

new lemma ->

a model proof ->

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

to1

t1 r1

∃ r1 : rule. True

→

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

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

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

new types ->

new constants ->

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

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

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

new lemma ->

a model proof ->

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

to1

t1 r1

∃ r1 : rule. True

→

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

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

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

new types ->

new constants ->

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

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

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

new lemma ->

a model proof ->

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

to1

t1 r1

∃ r1 : rule. True

→

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

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

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

new types ->

new constants ->

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

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

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

new lemma ->
a model proof ->

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

to1

t1 r1

∃ r1 : rule. True

→

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

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

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

new types ->

new constants ->

new lemma ->

a model proof ->

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

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

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

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

t2 t01 t1 r1

∃ r1 : rule. True

→

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

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

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

(t2 = is1, s, and stk)

new types ->

new constants ->

new lemma ->

a model proof ->

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

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

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

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

t2 t01 t02 t1 r1

∃ r1 : rule. True

→

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

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

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

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

new types ->

new constants ->

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

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

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

new lemma ->
a model proof ->

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

t2 t01 t02 t1 r1

∃ r1 : rule. True

→

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

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

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

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

new types ->

new constants ->

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

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

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

new lemma ->

a model proof ->

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

t2 t01 t02 t1 r1

∃ r1 : rule. True

→

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

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

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

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

new types ->

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

new constants ->

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

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

new lemma ->

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

a model proof ->

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

first

t2 t01 first t01 t02

∃ r1 : rule. True

→

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

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

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

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

when t2 is is1 (n -> 1) ?

new types ->

new constants ->

new lemma ->

a model proof ->

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

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

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

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

first

t2

to1

to2

t1

r1

first

∃ r1 : rule. True

→

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

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

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

∧

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

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

when t2 is is1 (n -> 1)

new types ->

new constants ->

new lemma ->

a model proof ->

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

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

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

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

first / second / t2 / to1 / t1 / r1 / to2

∃ r1 : rule. True

→

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

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

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

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

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

new types ->

new constants ->

new lemma ->

a model proof ->

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

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

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

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

first, second, t2, to1, t1, r1, to2

∃ r1 : rule. True

→

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

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

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

∧

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

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

when t2 is is1 (n -> 1)

when t2 is s (n -> 2)

new types ->

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

new constants ->

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

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

new lemma ->

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

a model proof ->

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

∃ r1 : rule. True

→

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

first second third t2

first second third to1

t1 r1

to2

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

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

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

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

new types ->

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

new constants ->

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

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

new lemma ->

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

a model proof ->

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

∃ r1 : rule. True

→

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

first, second, third, t1, r1, to1, to2, t2

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

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

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

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

new types ->

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

new constants ->

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

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

new lemma ->

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

a model proof ->

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

∃ r1 : rule. True

→

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

first second third t2

first second third to1

t1 r1

to2

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

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

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

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

Big Picture

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

ML

<- pros: good at ambiguity (heuristics)

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

LiFtEr LiFtEr LiFtEr 3 LiFtEr 1 LiFtEr 2 LiFtEr 3 LiFtEr 1 LiFtEr 2 LiFtEr 3

<- pros: good at rigorous abstraction

~~<- cons: bad at ambiguity (heuristics)~~

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

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

<- small dataset about different domains

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

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

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

ML

[[T,F,T], [T,T,T], [F,T,T]]: bool list

<- pros: good at ambiguity (heuristics)

<- pros: good at rigor

~~<- cons: bad at ambiguity (heuristics)~~

WIP

LiFtEr

LiFtEr

LiFtEr 3

LiFtEr 1

LiFtEr 2

LiFtEr 3

LiFtEr 1

LiFtEr 2

LiFtEr 3

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

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

<- small dataset about different domains

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

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

https://twitter.com/YutakangE

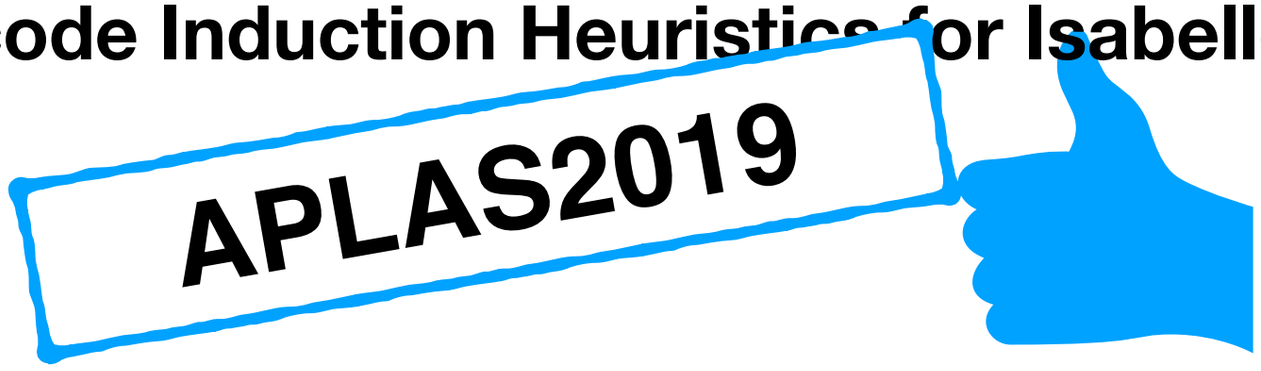
3 secrets behind LiFtEr

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

https://twitter.com/YutakangE

3 secrets behind LiFtEr

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL



https://twitter.com/YutakangE

3 secrets behind LiFtEr

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

APLAS2019

Secret 1

Many induction heuristics are simple.

https://twitter.com/YutakangE

3 secrets behind LiFtEr

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

APLAS2019

Secret 1

Many induction heuristics are simple.

Secret 2

Proofs can be more abstract than theorems.

https://twitter.com/YutakangE https://github.com/data61/PSL/slide/2019_ps.pdf

3 secrets behind LiFtEr

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

APLAS2019

Secret 1

Many induction heuristics are simple.

Secret 2

Proofs can be more abstract than theorems.

Secret 3

LiFtEr is not good enough because …

LiFtEr almost ignores the definitions of constants.

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

LiFtEr LiFtEr LiFtEr

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

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

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

<- small dataset about different domains

<- one abstract representation

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

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

LiFtEr LiFtEr LiFtEr

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

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

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

<- small dataset about different domains

<- one abstract representation

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

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

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

<- relevant definitions

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

LiFtEr LiFtEr LiFtEr

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

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

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

<- small dataset about different domains

<- one abstract representation

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

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

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

<- relevant definitions

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

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

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

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

LiFtEr LiFtEr LiFtEr

<- small dataset about different domains

<- one abstract representation

LiFtEr 2 LiFtEr 2

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

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

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

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

<- relevant definitions

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

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

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

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

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

LiFtEr LiFtEr LiFtEr

<- small dataset about different domains

<- one abstract representation

LiFtEr 2 LiFtEr 2

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

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

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

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

<- relevant definitions

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

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

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

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

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

LiFtEr LiFtEr LiFtEr

<- small dataset about different domains

<- one abstract representation

LiFtEr 2 LiFtEr 2

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

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

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

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

<- relevant definitions

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

WIP

[[T,F,T], [T,T,T], [F,T,T]]: bool list <- s

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

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

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

LiFtEr LiFtEr LiFtEr

<- small dataset about different domains

<- one abstract representation

LiFtEr 2 LiFtEr 2

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

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

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

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

<- relevant definitions

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

[T,F,T], [T,T,T], [F,T,T]: bool list <- su

WIP

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

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

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

LiFtEr LiFtEr LiFtEr

<- small dataset about different domains

<- one abstract representation

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

LiFtEr 2 LiFtEr 2

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

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

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

<- relevant definitions

fin.