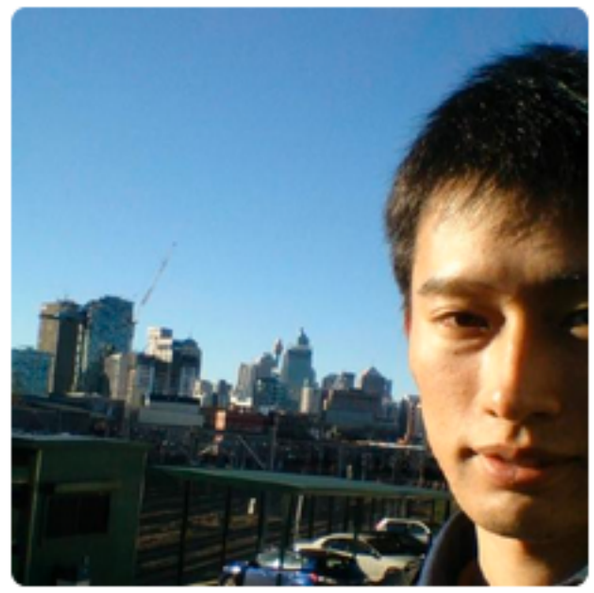
https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

Yutaka Nagashima
University of Innsbruck
Czech Technical University



Yutaka Ng

https://twitter.com/YutakangE

slides

https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

Yutaka Nagashima
University of Innsbruck
Czech Technical University

Yutaka Ng

https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

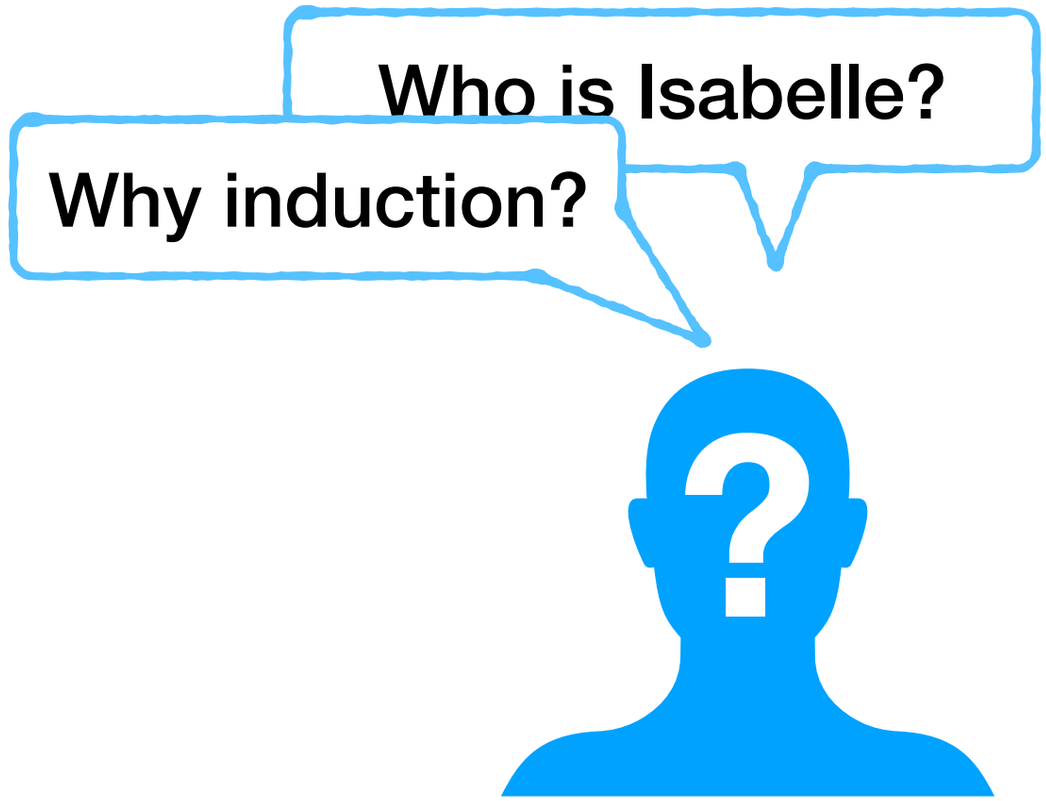
This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

Who is Isabelle?

https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

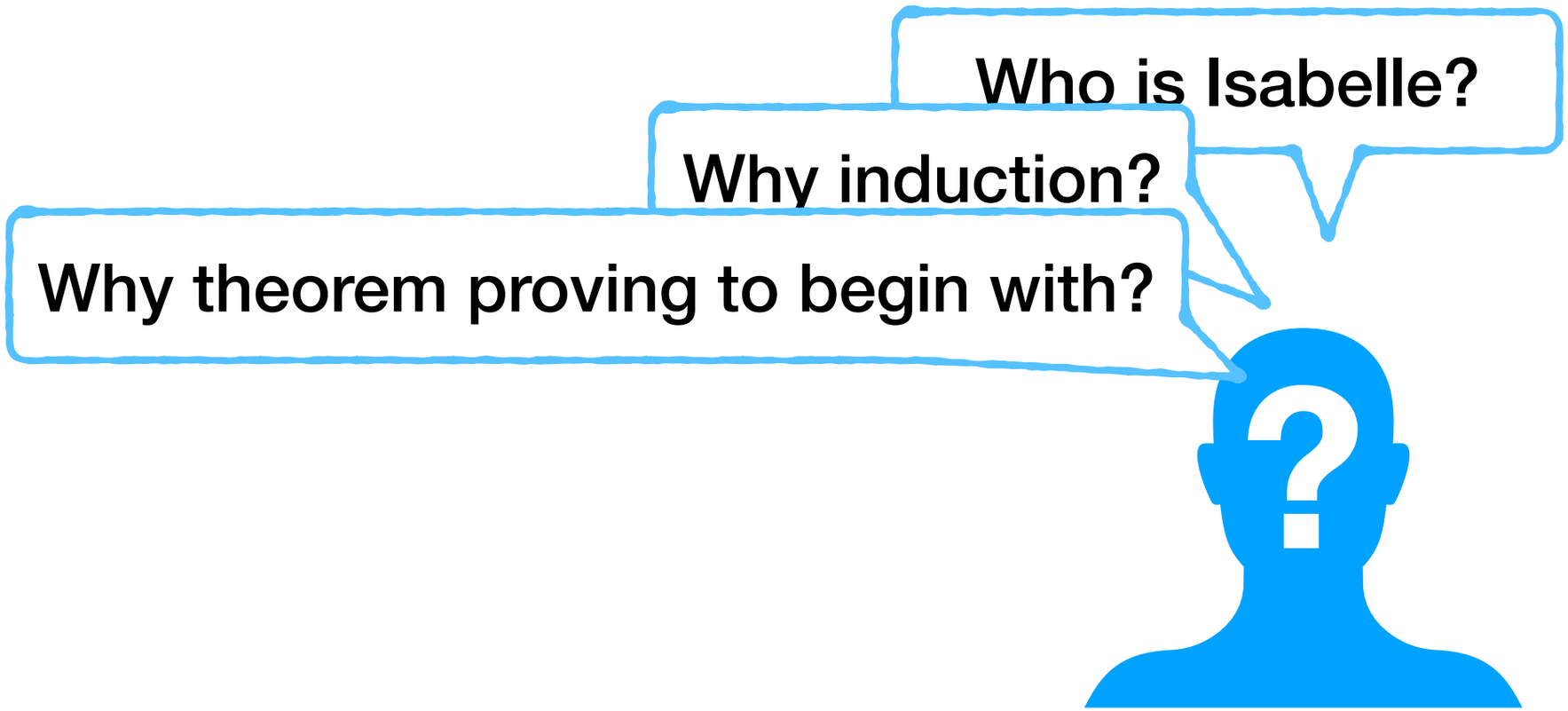
This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).



https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).



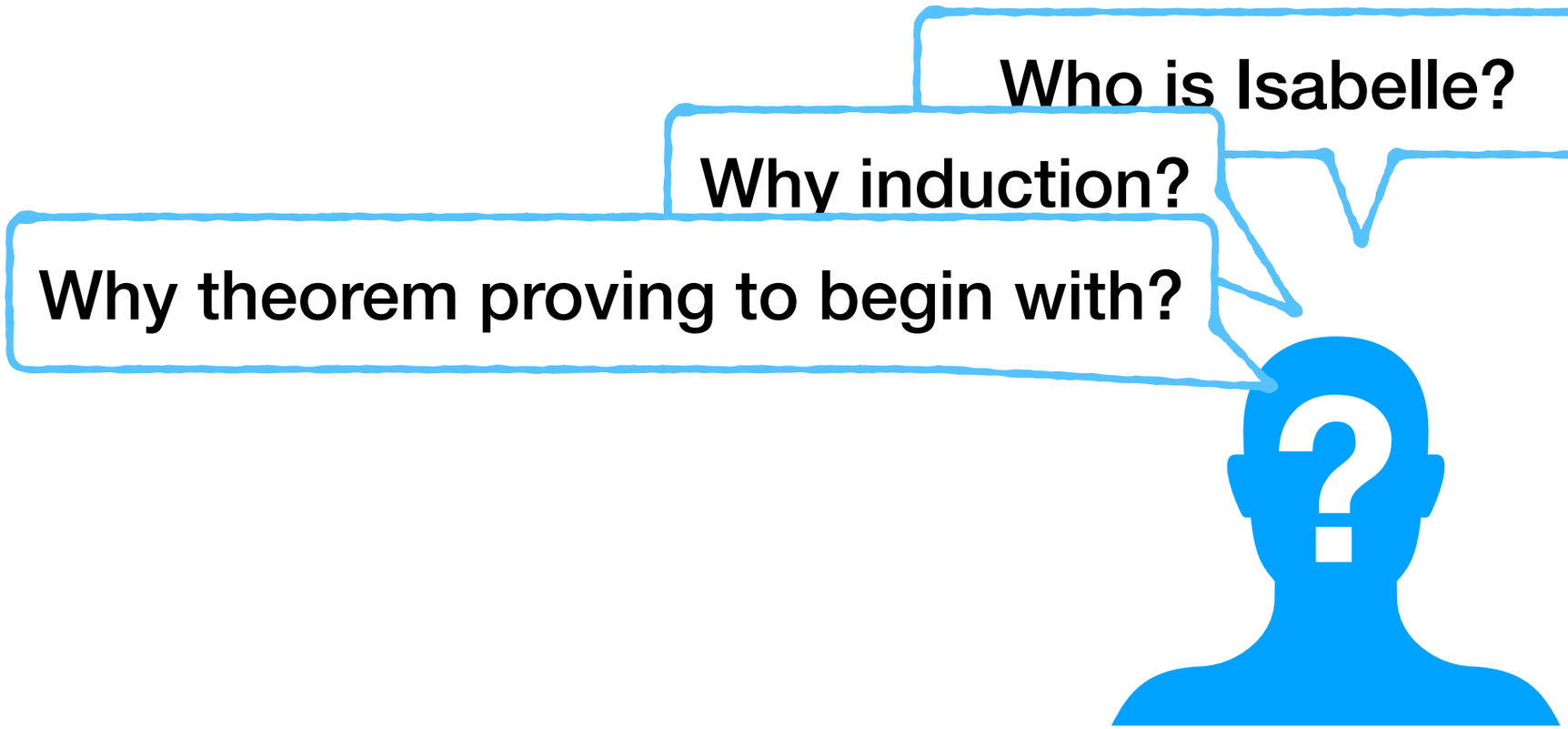
https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

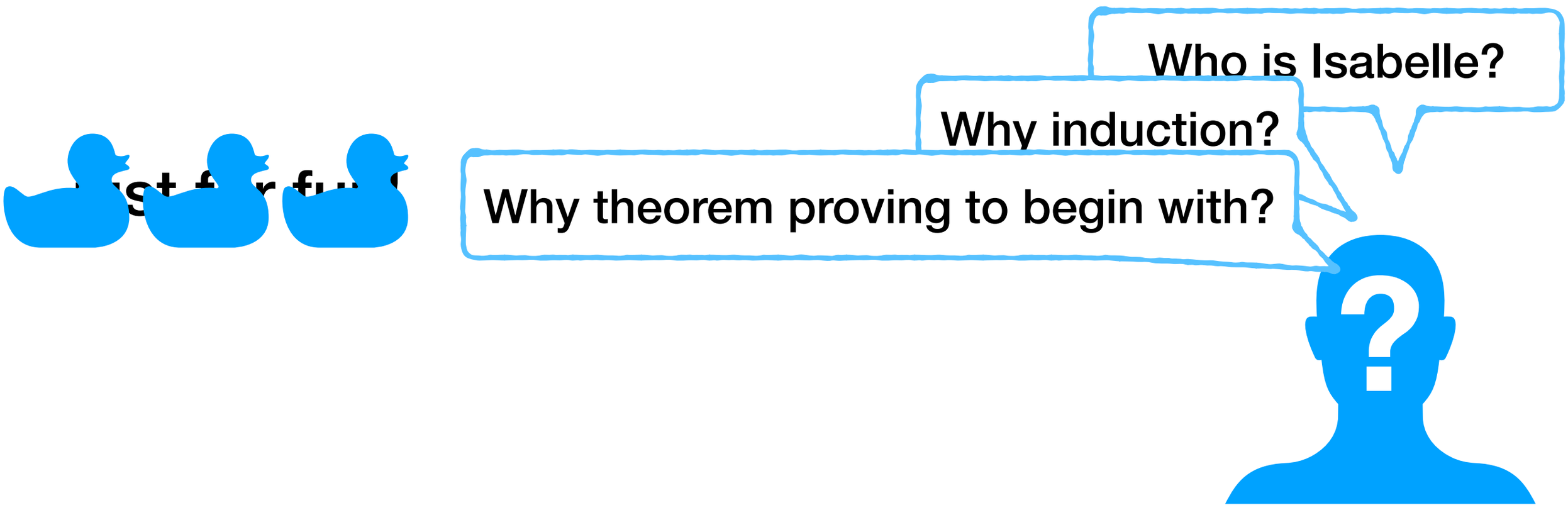
just for fun!



https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

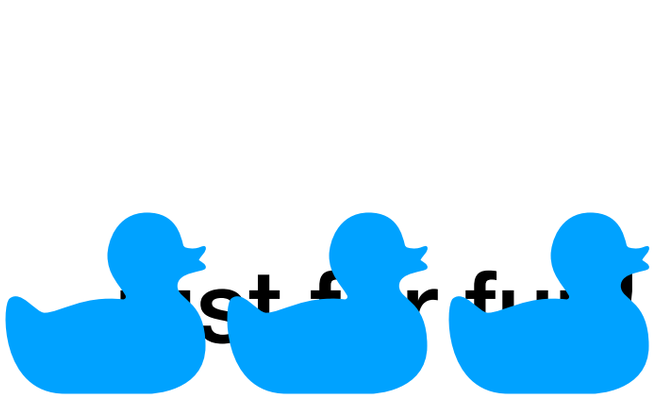
This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).



https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).



Who is Isabelle?

Why induction?

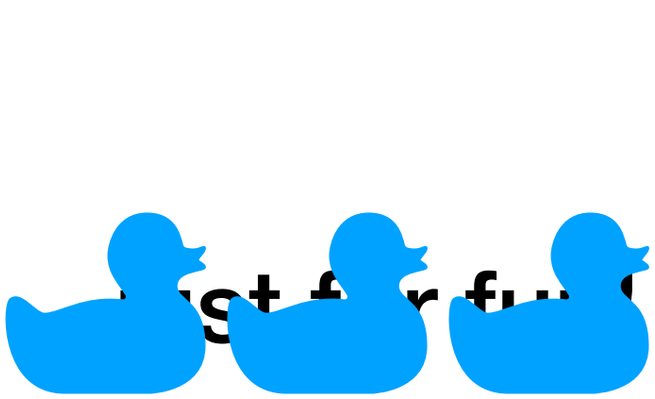
Why theorem proving to begin with?

for complete formal verification!

https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).



Who is Isabelle?

Why induction?

Why theorem proving to begin with?

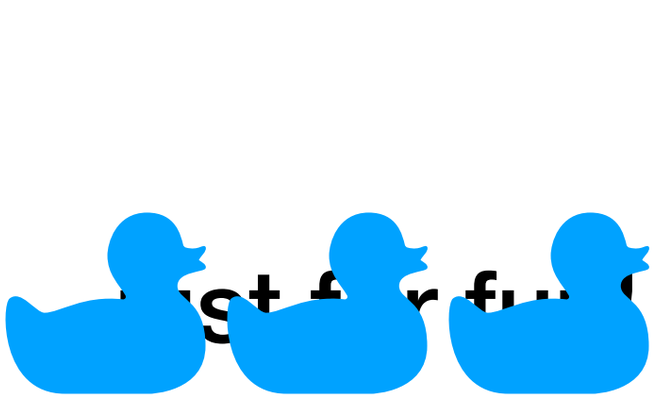
for complete formal verification!

developer **proof assistant / ITP** **implementation**

https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).



Who is Isabelle?

Why induction?

Why theorem proving to begin with?

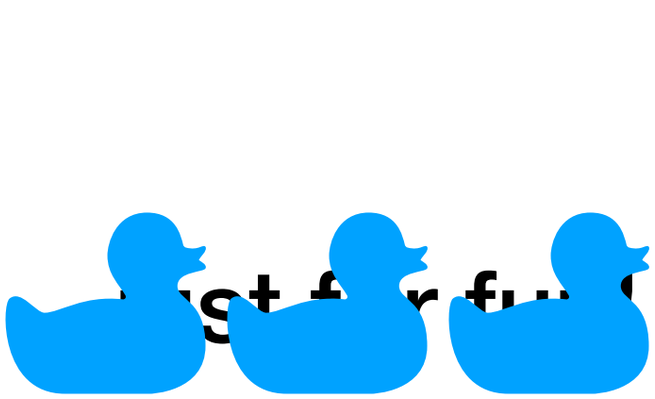
for complete formal verification!

developer	proof assistant / ITP		implementation
	+ 	=	
Gewin Klein et. al	Isabelle/HOL		verified micro kernel, seL4

https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

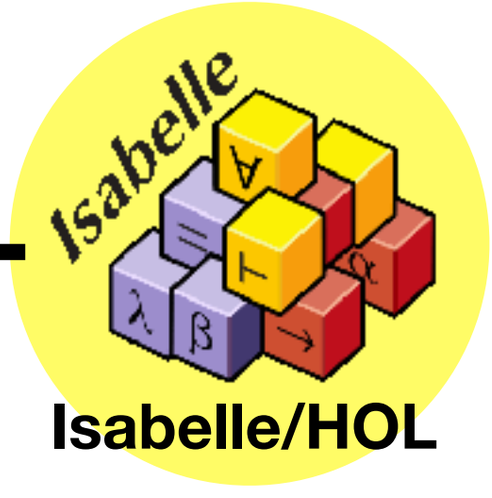


Who is Isabelle?

Why induction?

Why theorem proving to begin with?

for complete formal verification!

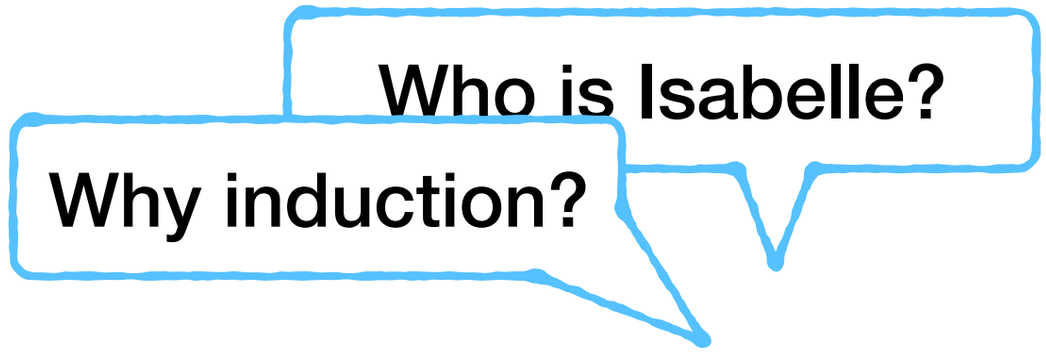
developer	proof assistant / ITP		implementation
		=	
Gewin Klein et. al	Isabelle/HOL		verified micro kernel, seL4

https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).



https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

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

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

Who is Isabelle?

Why induction?

https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

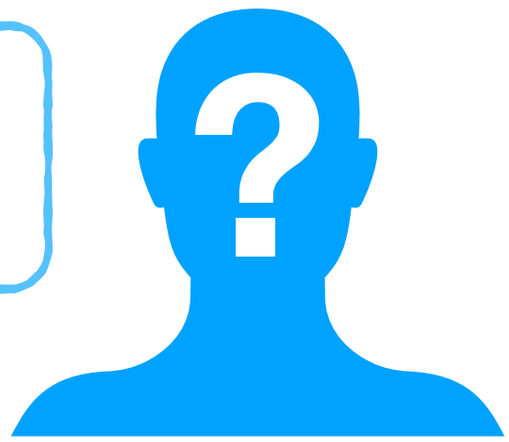
ITP (Inductive Theorem Proving) problems are at the heart of many verification and reasoning tasks in computer science.

Who is Isabelle?

Why induction?

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

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



https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

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

Who is Isabelle?

Why induction?

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

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

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

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

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

Who is Isabelle?

Why induction?

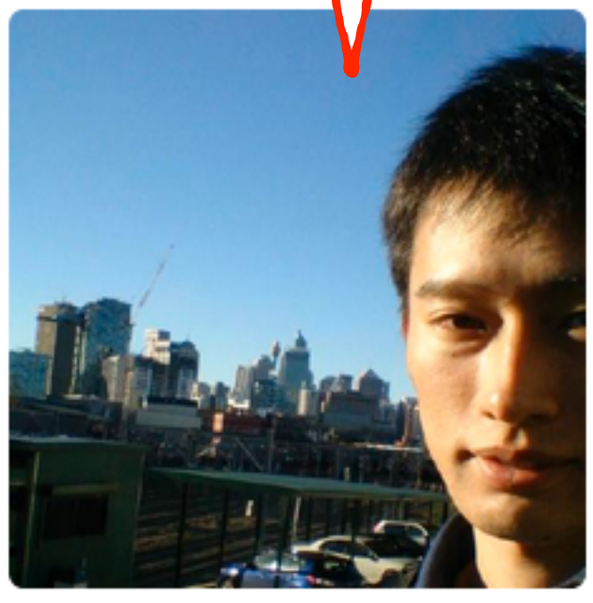
Challenge accepted!

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

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

spectacular breakthro

unrealistic, in view of the problems and the inherent inductive theorem pr



Yutaka Ng
yutakang

https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

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

Who is Isabelle?

Why induction?

Challenge accepted!

The time has come!

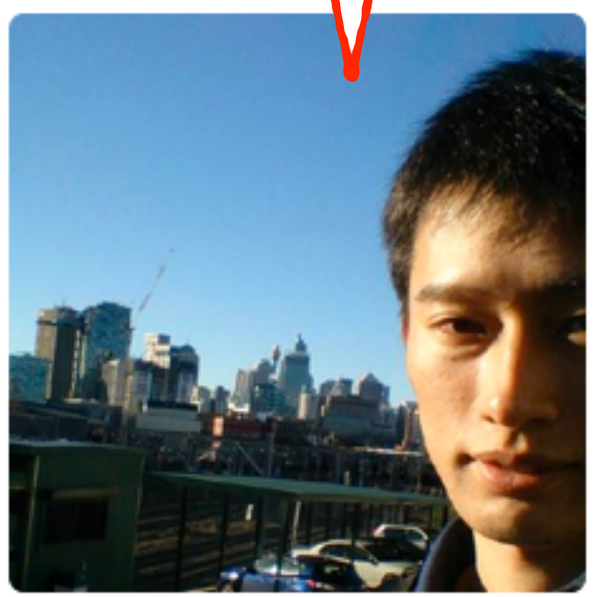
Prof. Bernhard Gramlich
https://www.logic.at/staff/gramlich/

bstantial

ke time.

spectacular breakthro

unrealistic, in view of the problems and the inherent inductive theorem pr



Yutaka Ng
yutakang

https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

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

Who is Isabelle?

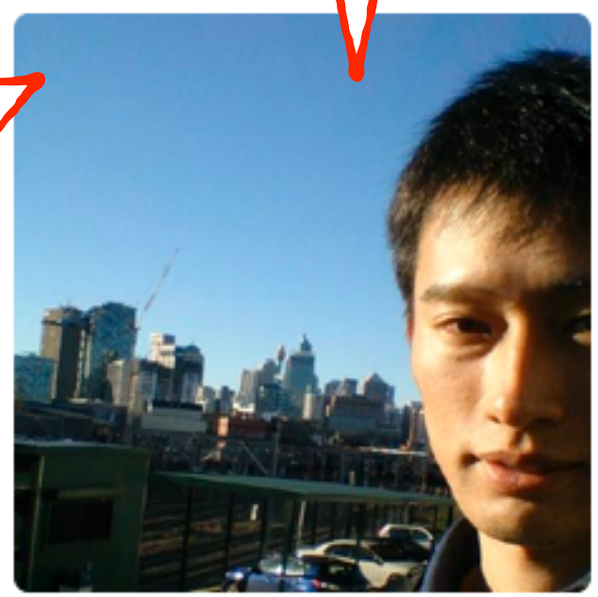
Why induction?

Challenge accepted!

The time has come!

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

...or is coming soon.



Yutaka Ng
yutakang

https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

What I did before APLAS2020…

https://twitter.com/YutakangE https://tinyurl.com/upZzhyk

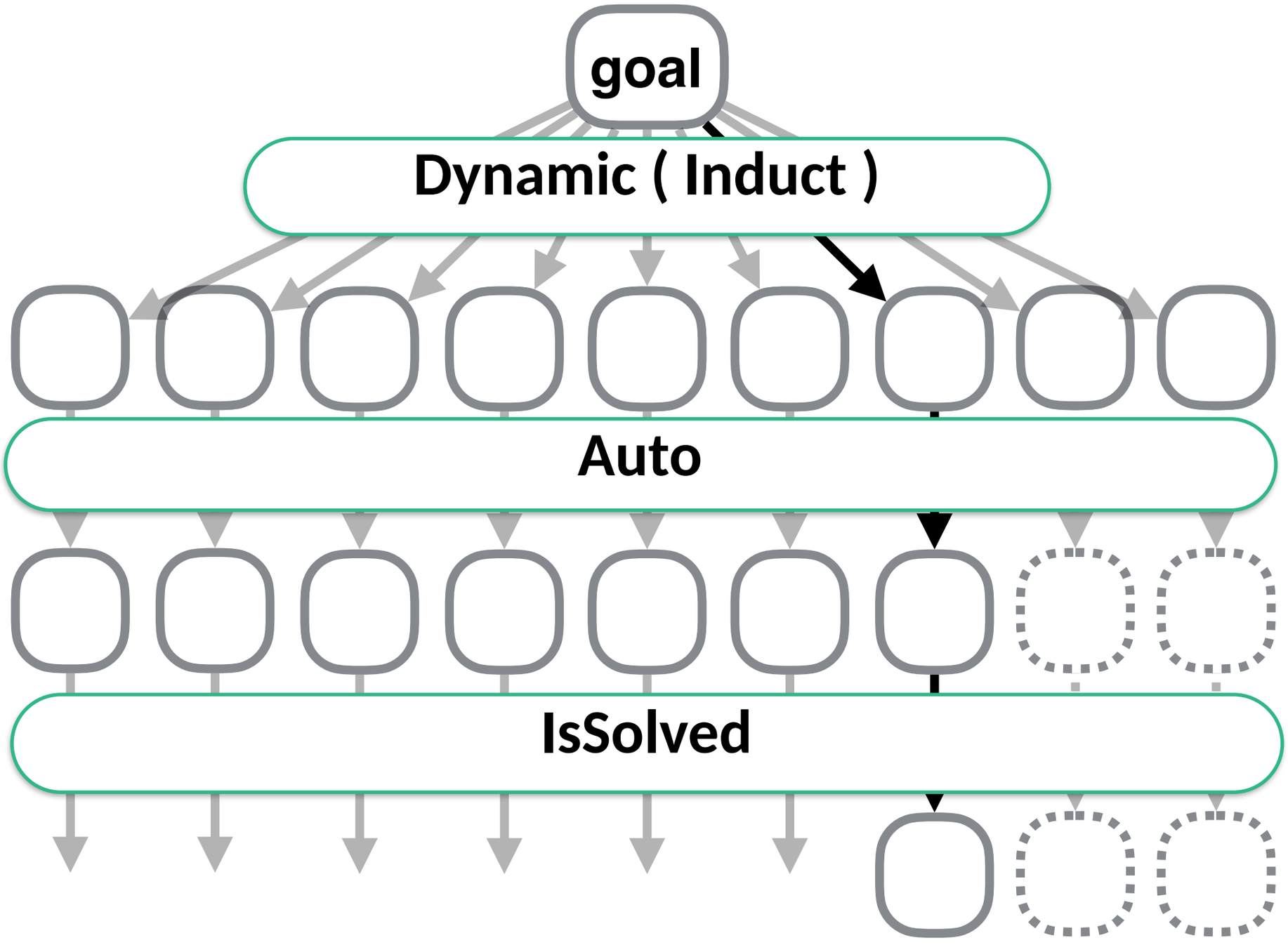
What I did before APLAS2020…

DEMO!

What I did before APLAS2020…

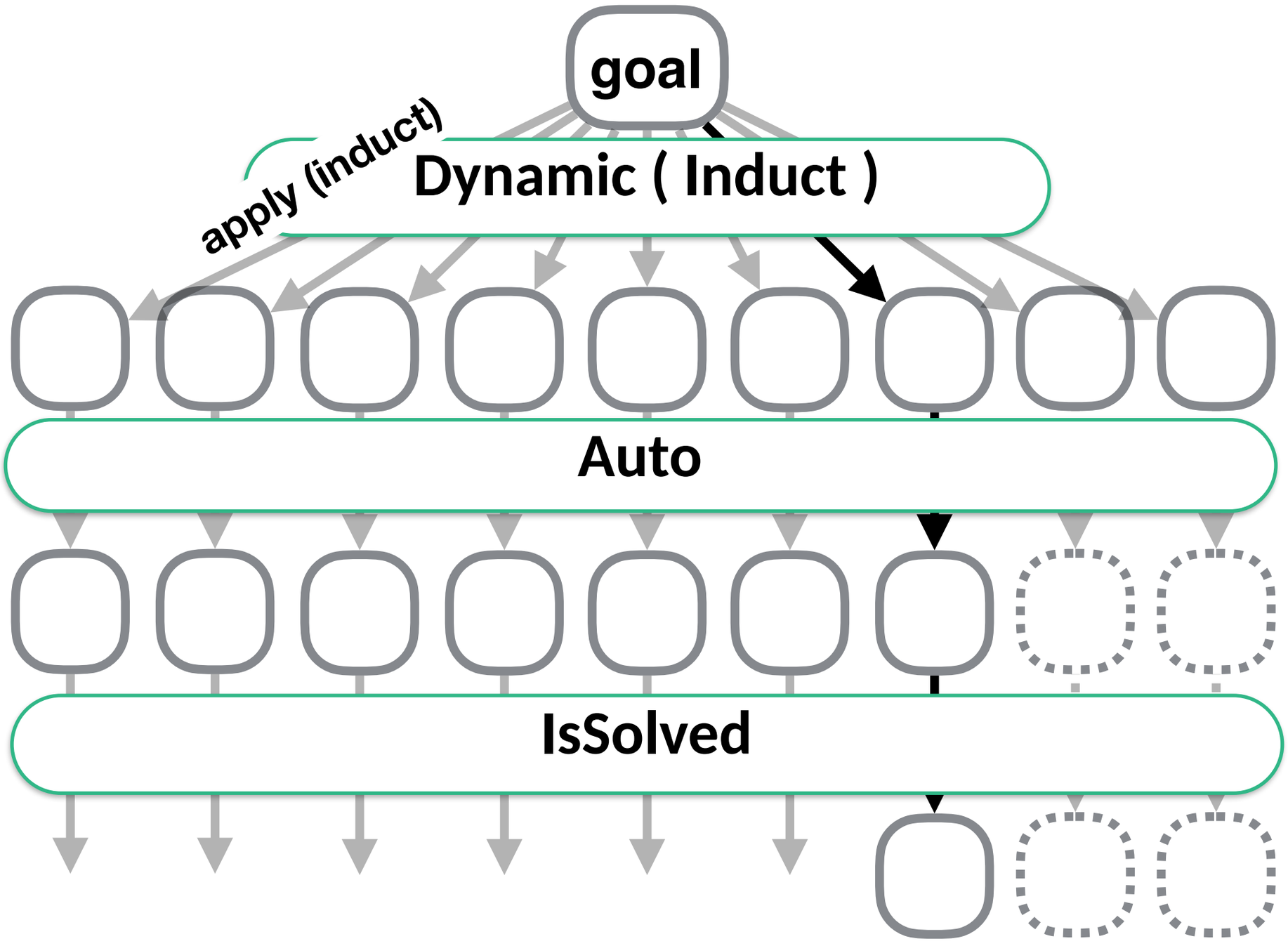
What I did before APLAS2020…

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



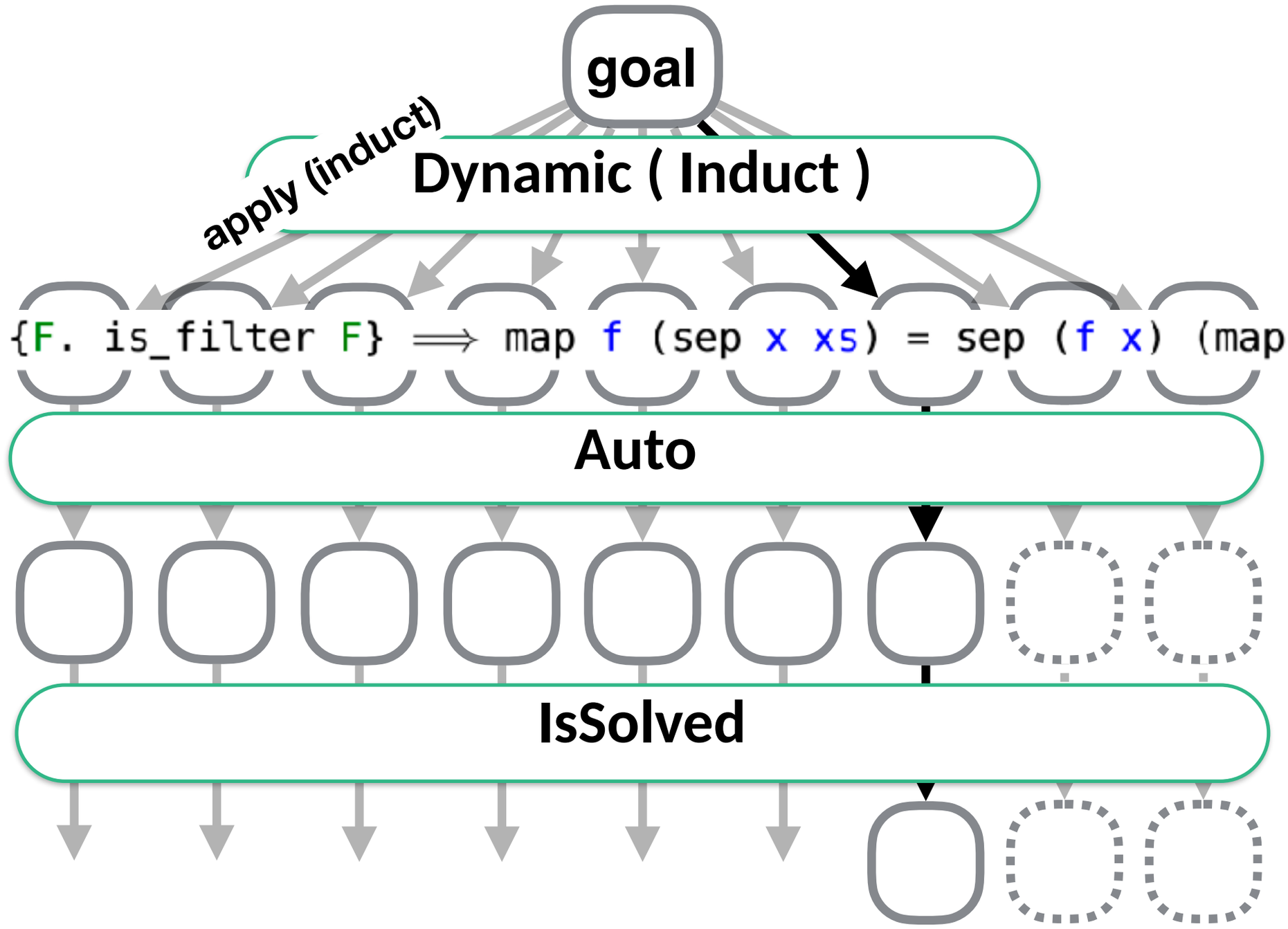
What I did before APLAS2020…

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



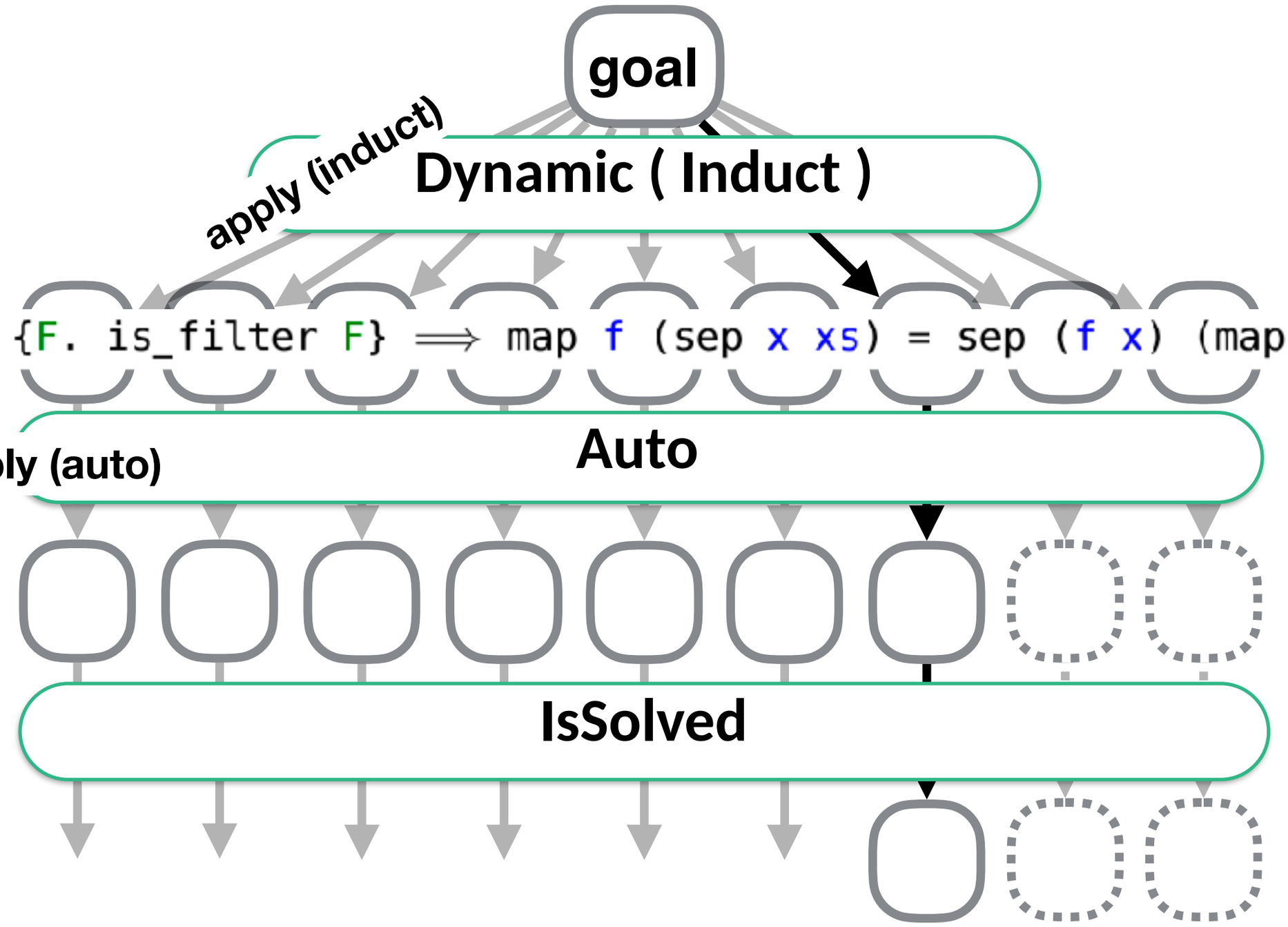
What I did before APLAS2020…

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



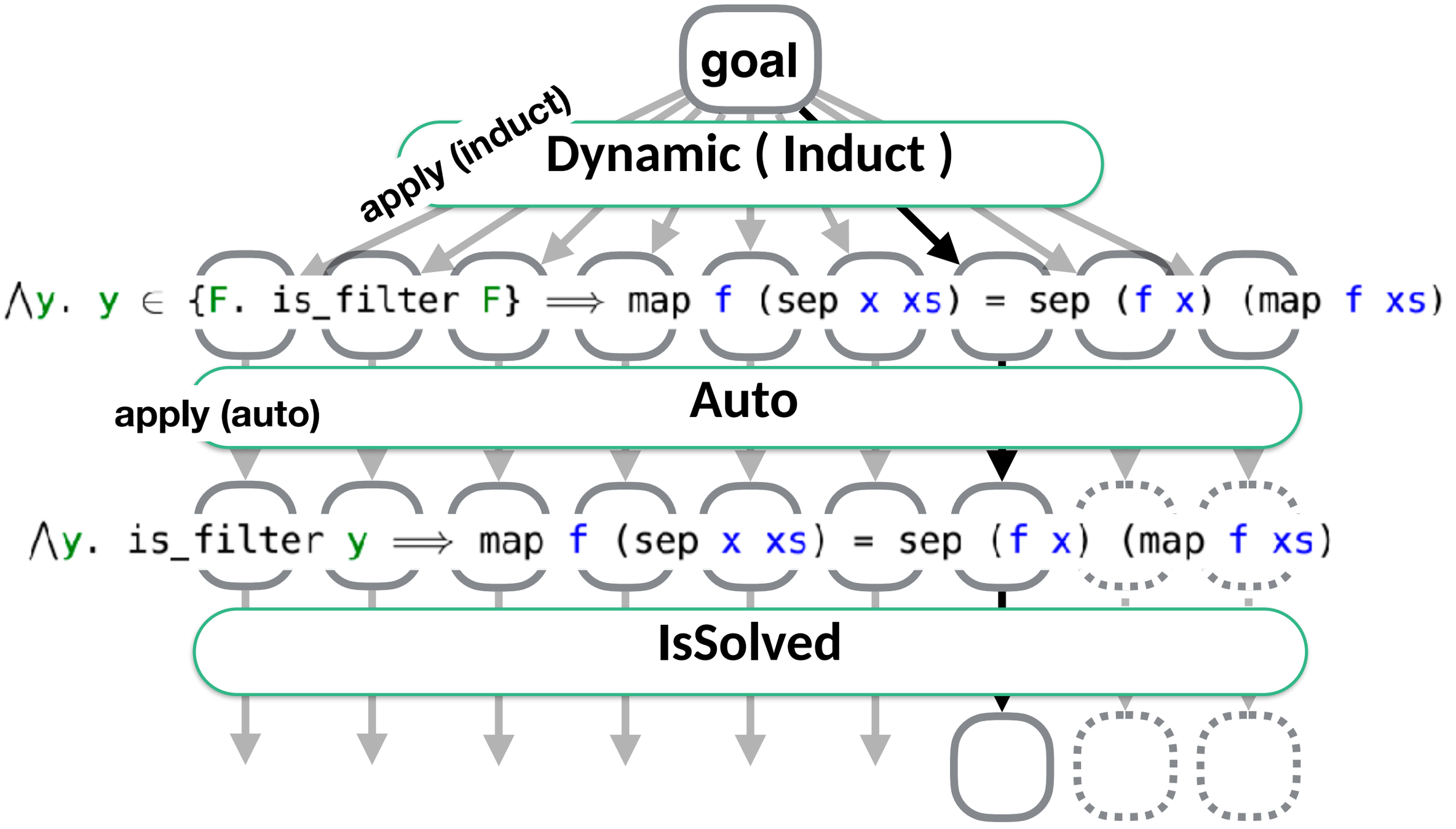
What I did before APLAS2020…

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



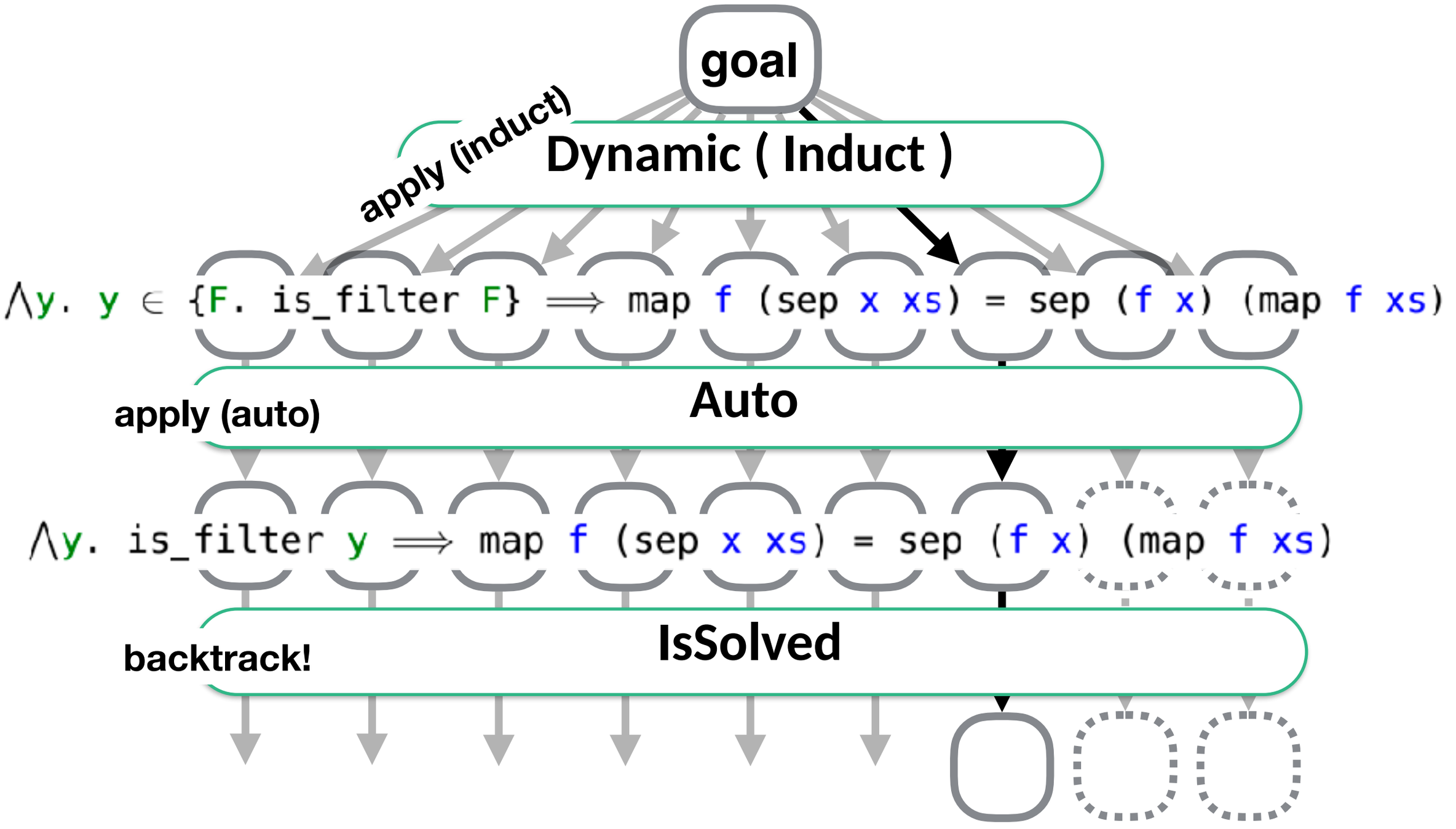
What I did before APLAS2020…

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



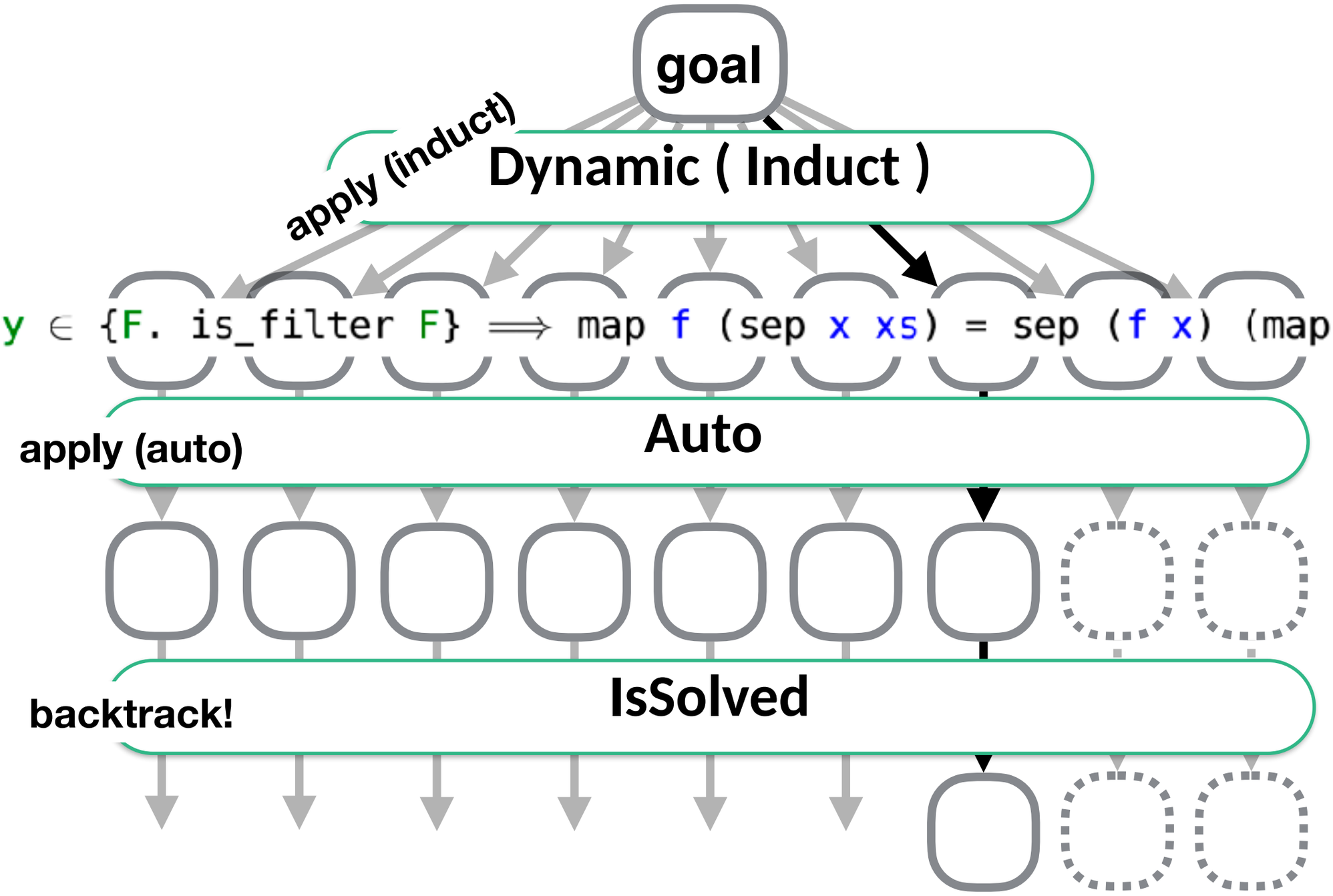
What I did before APLAS2020…

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



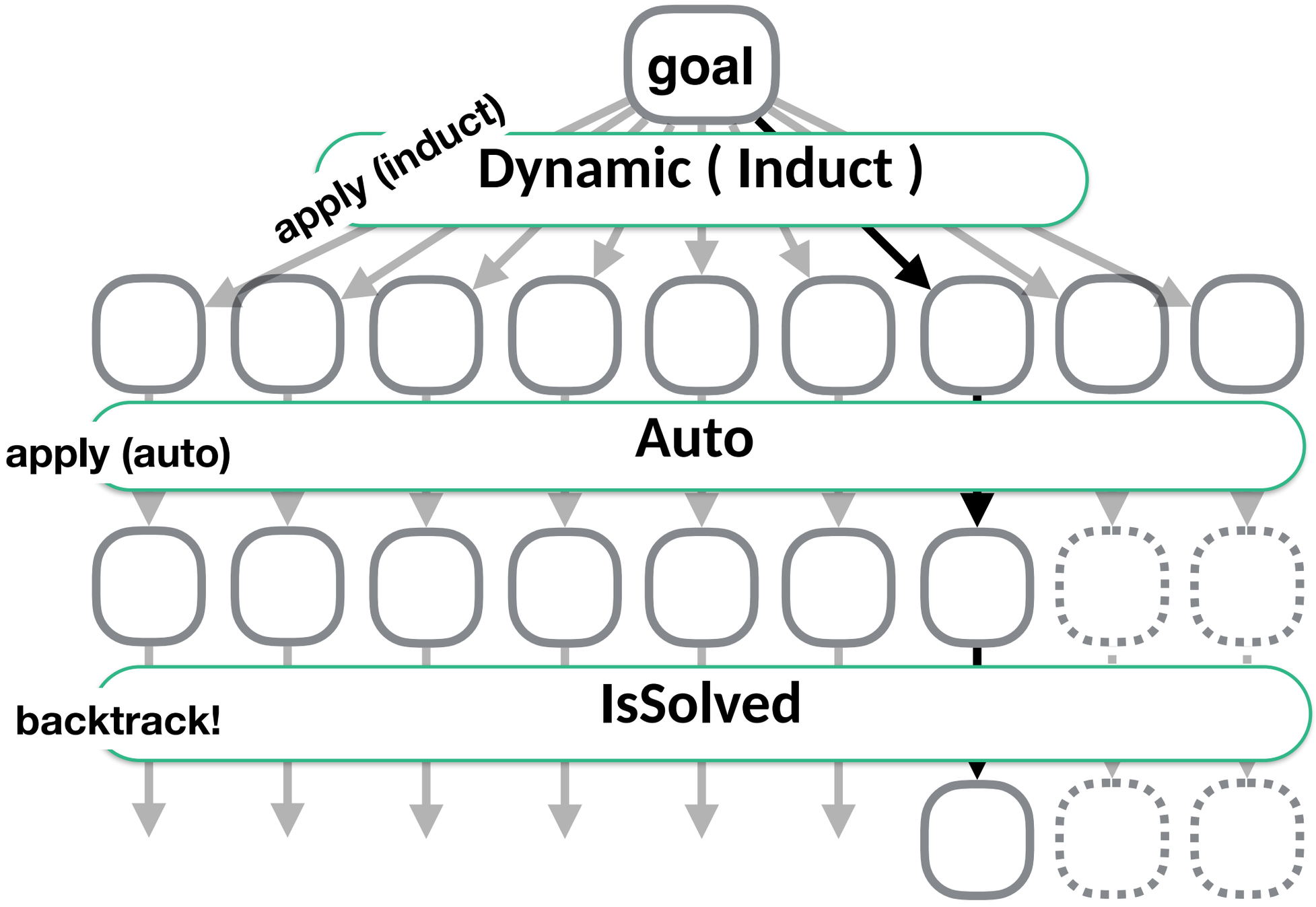
What I did before APLAS2020…

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



What I did before APLAS2020…

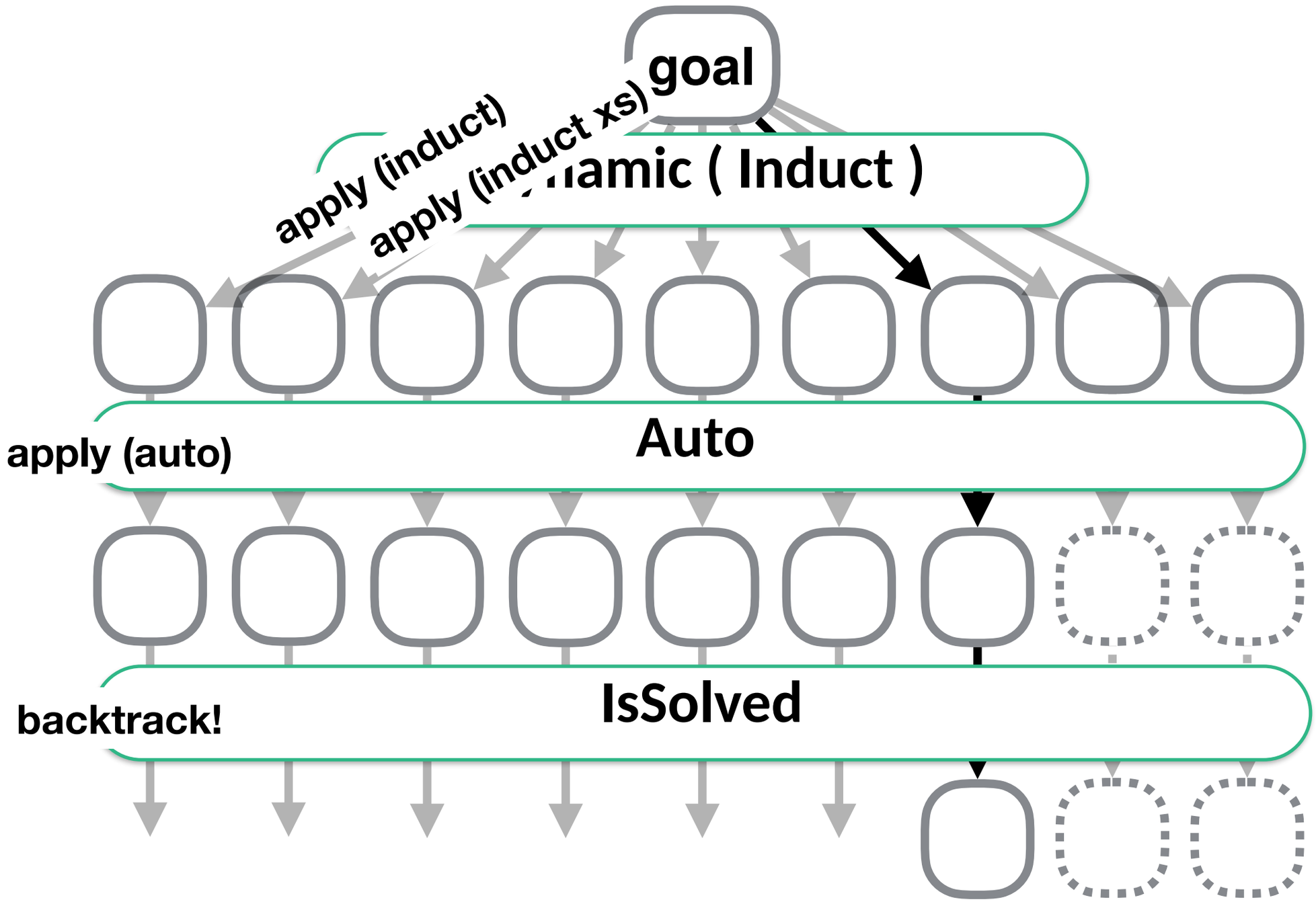
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

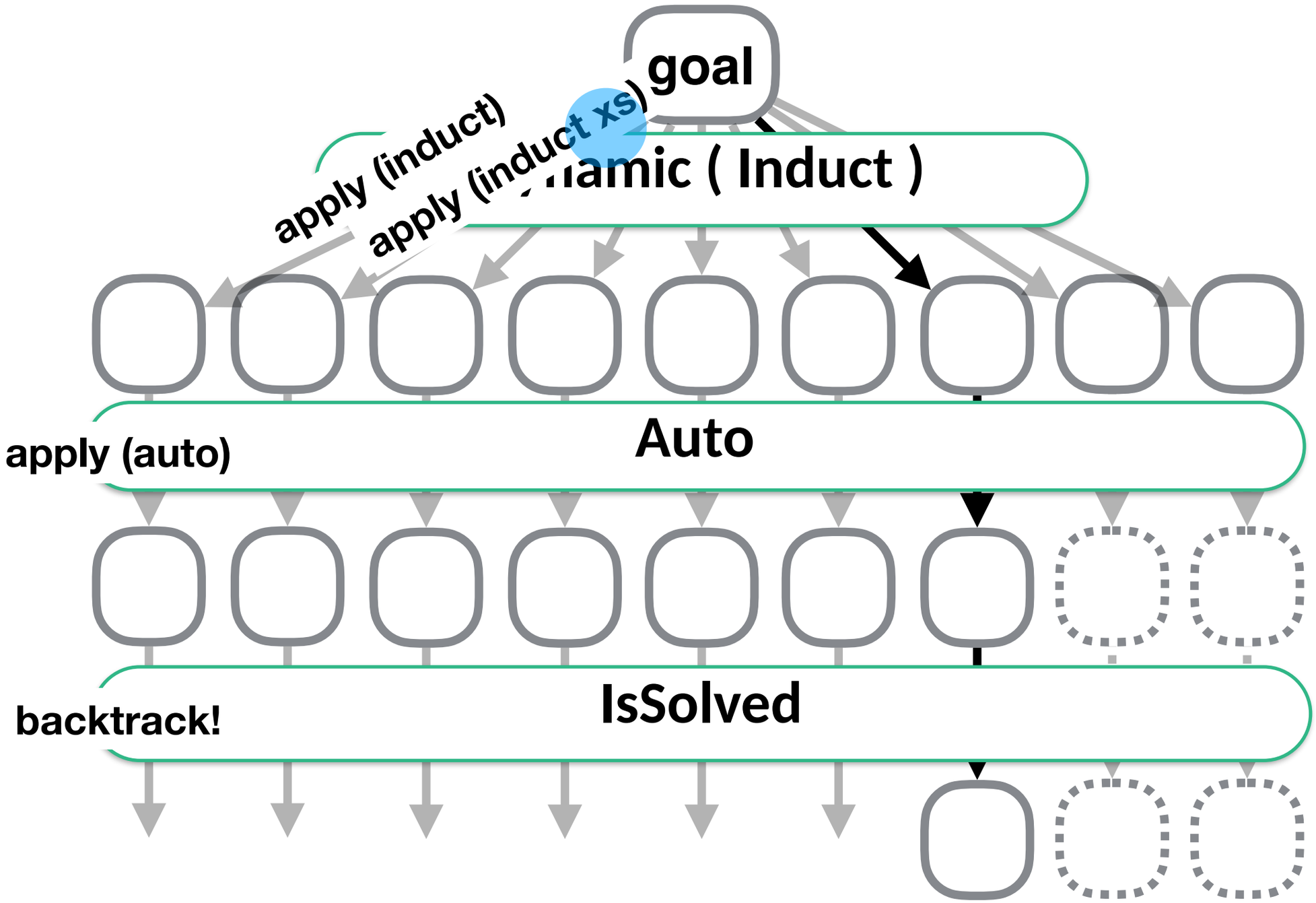
What I did before APLAS2020…

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



What I did before APLAS2020…

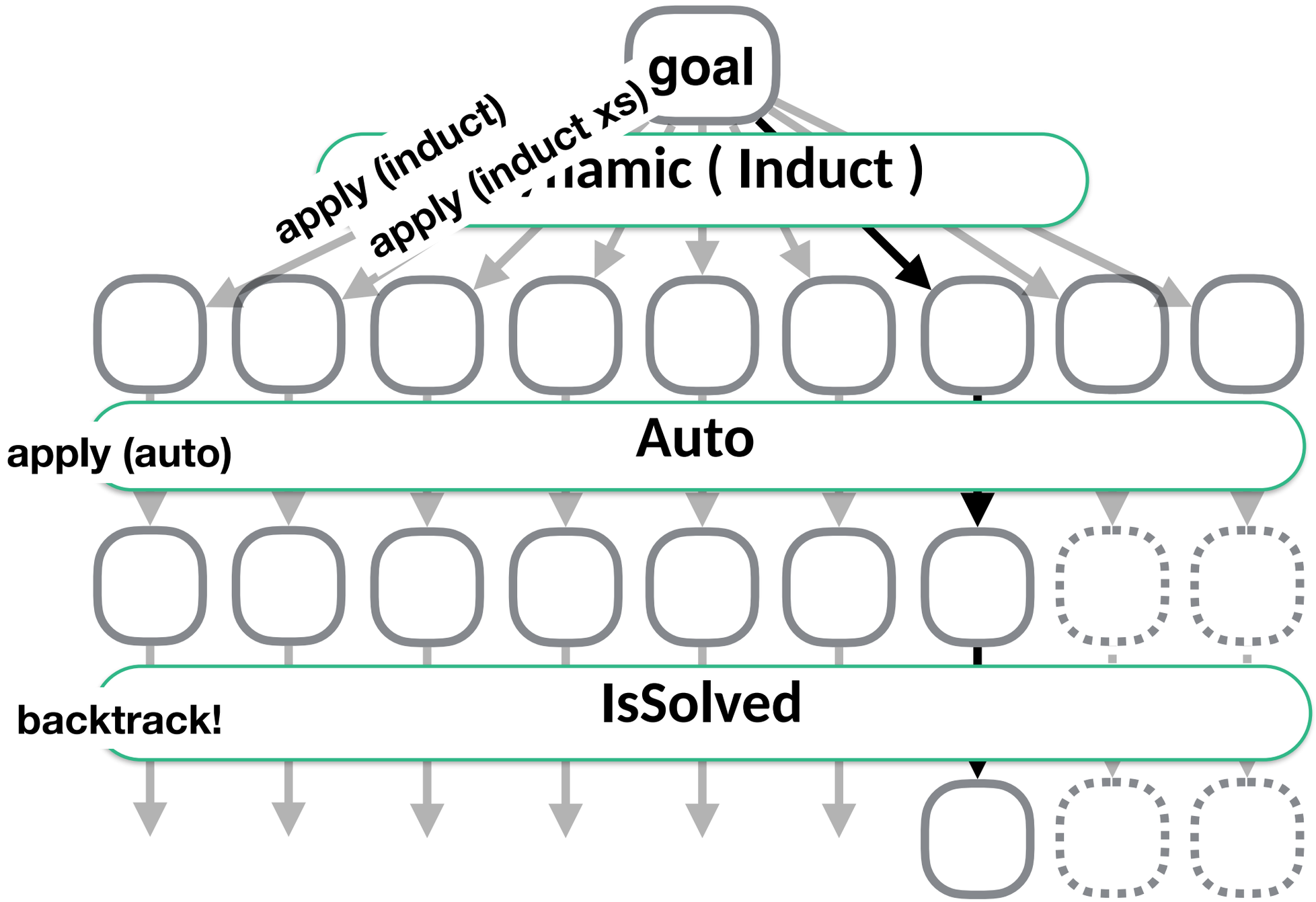
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

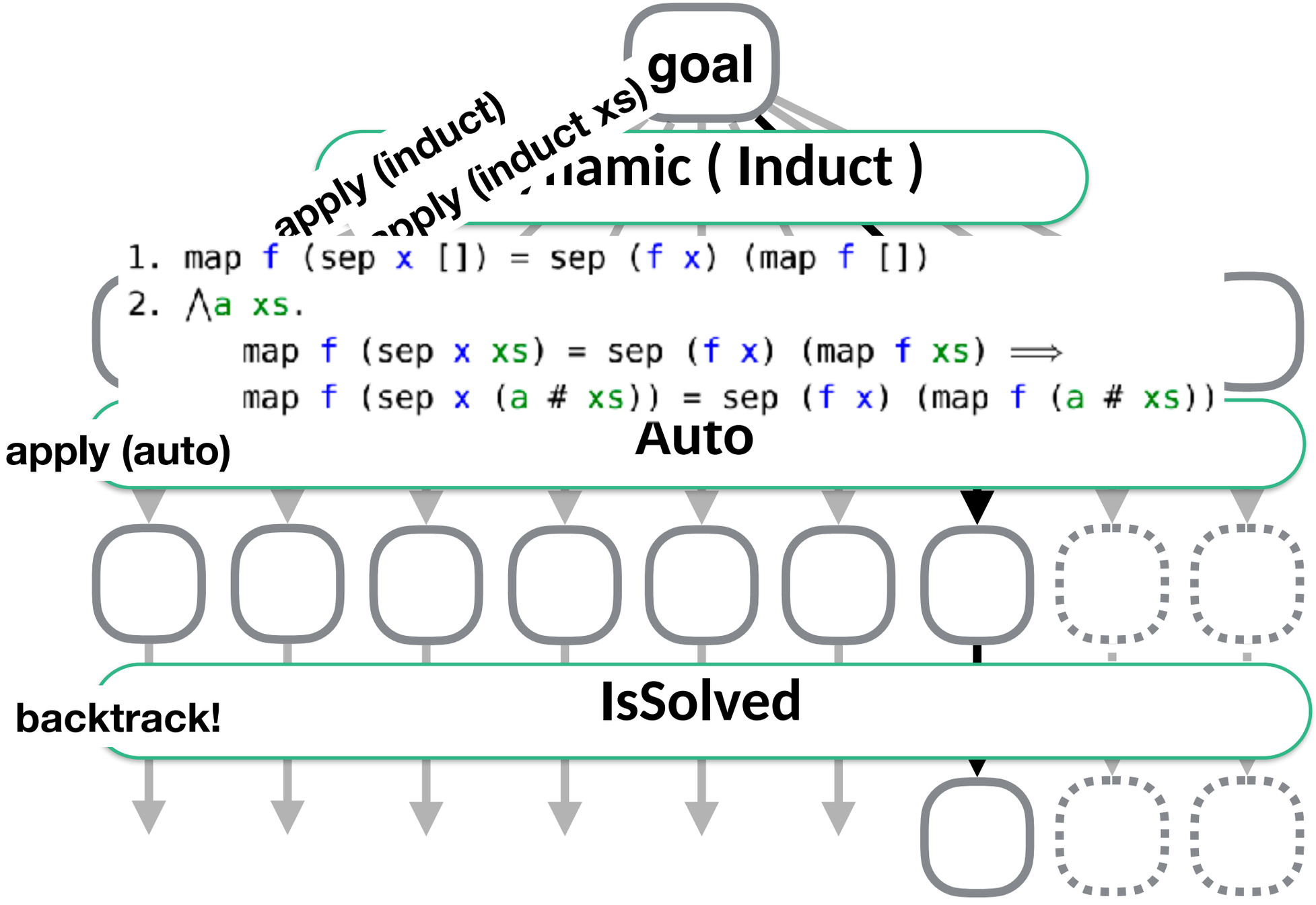
What I did before APLAS2020…

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



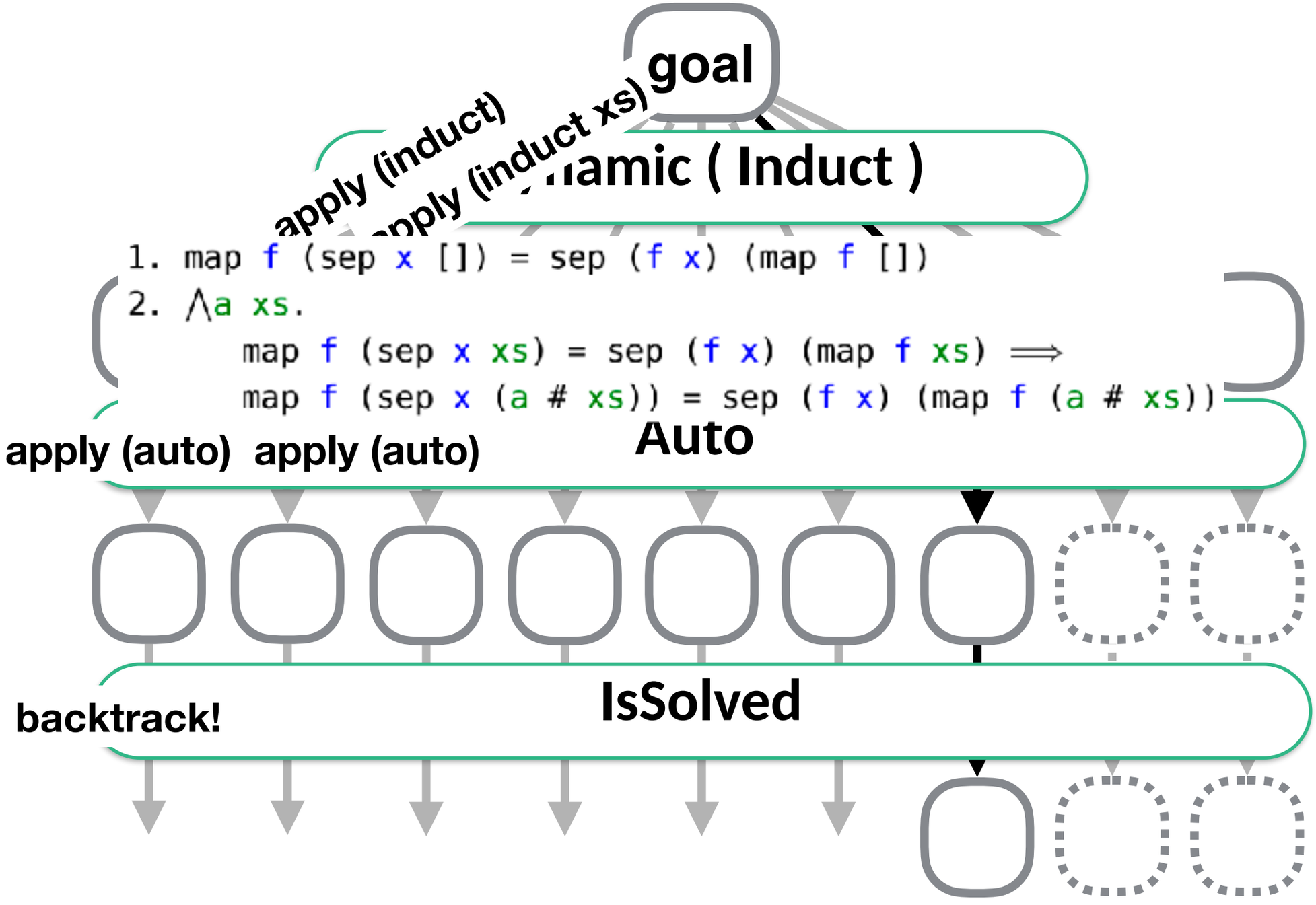
What I did before APLAS2020…

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



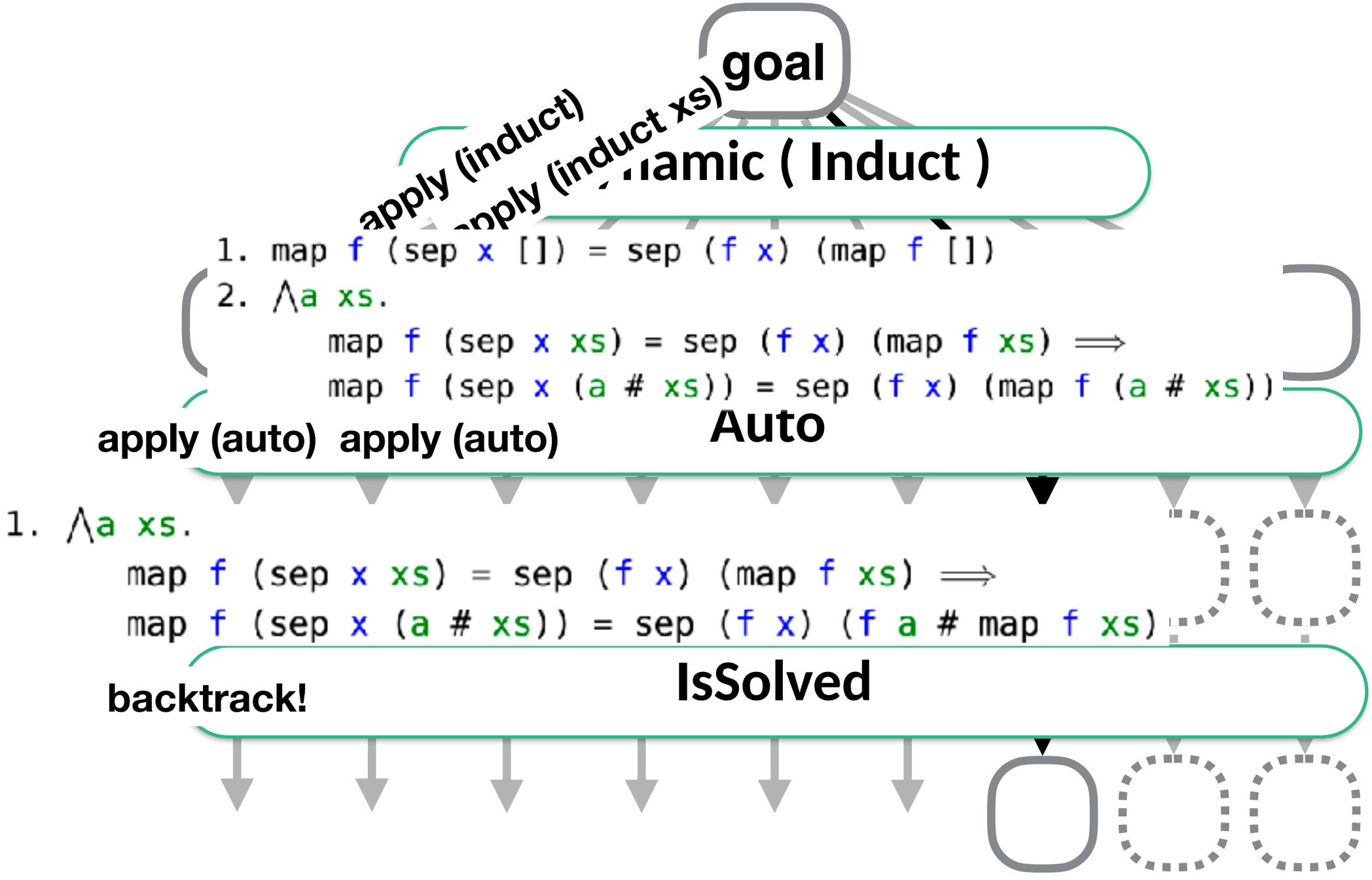
What I did before APLAS2020…

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



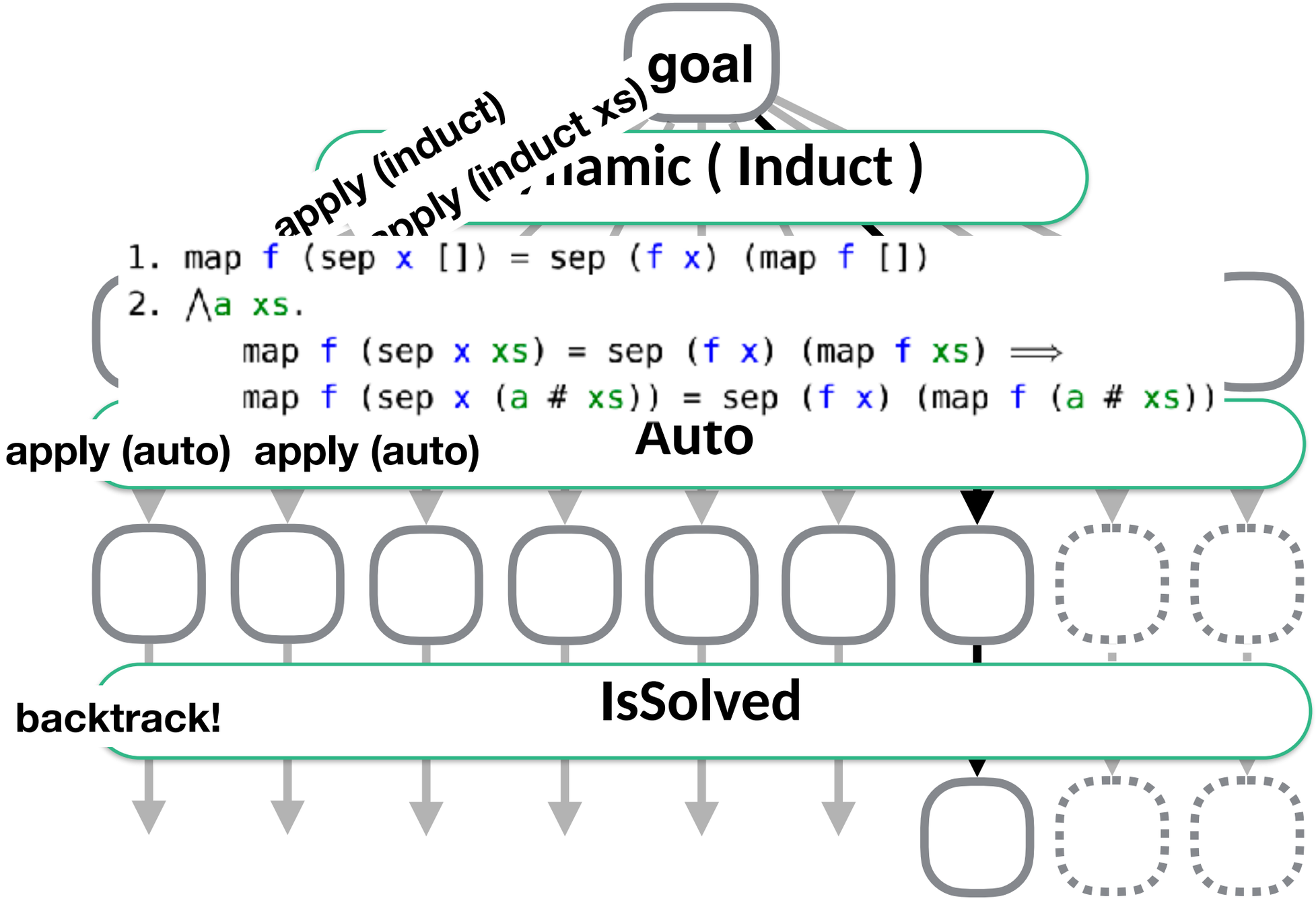
What I did before APLAS2020…

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



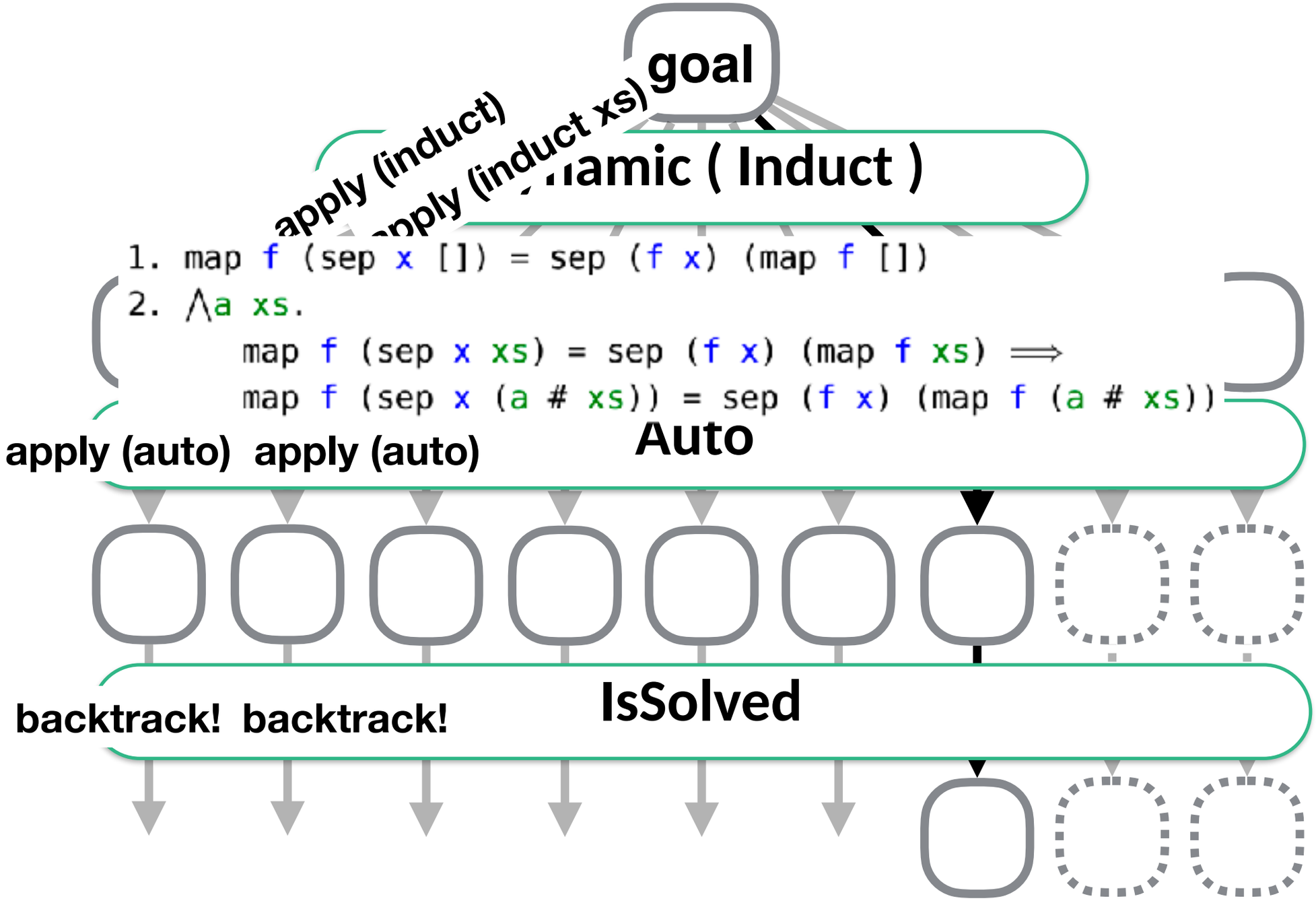
What I did before APLAS2020…

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



What I did before APLAS2020…

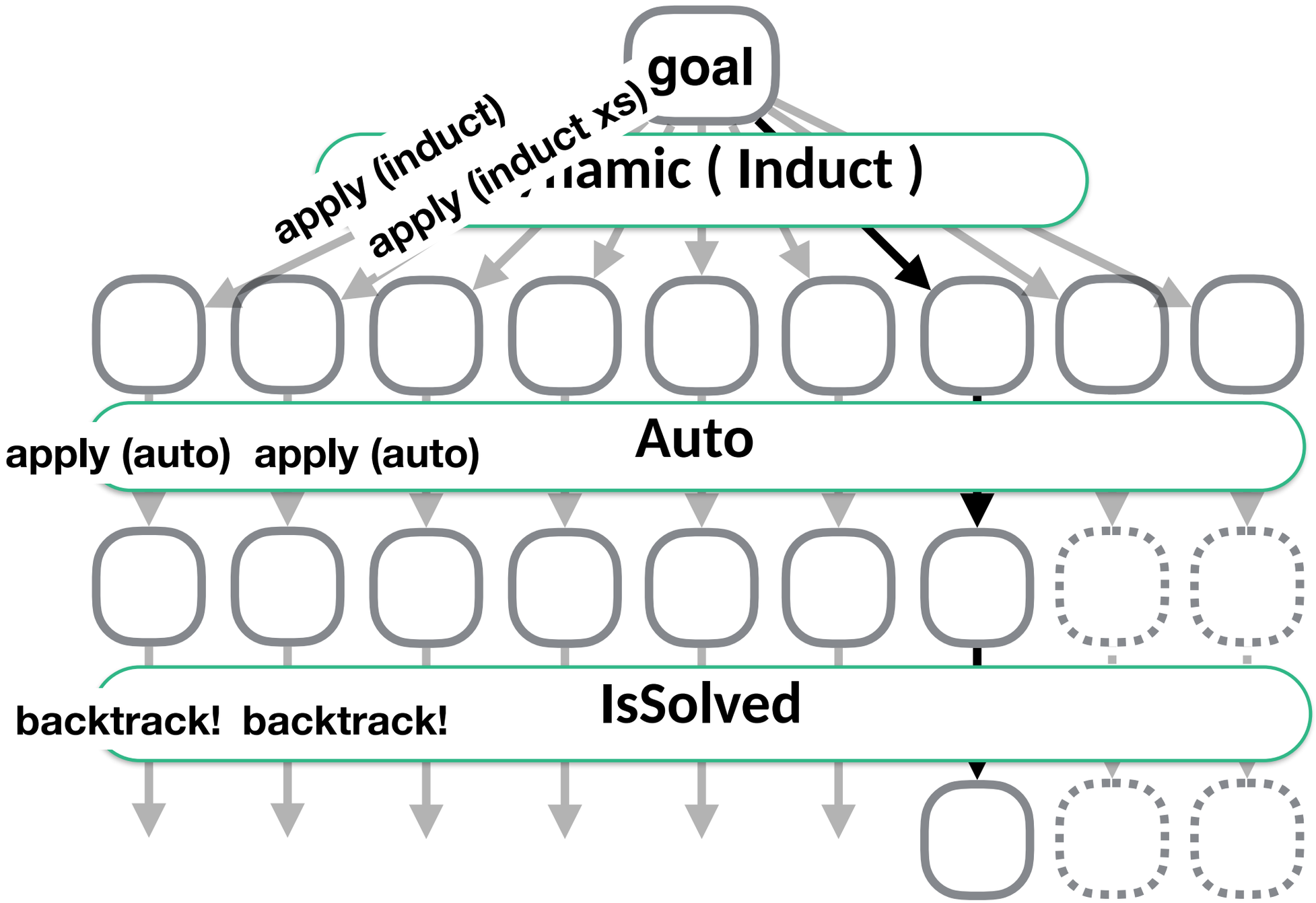
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

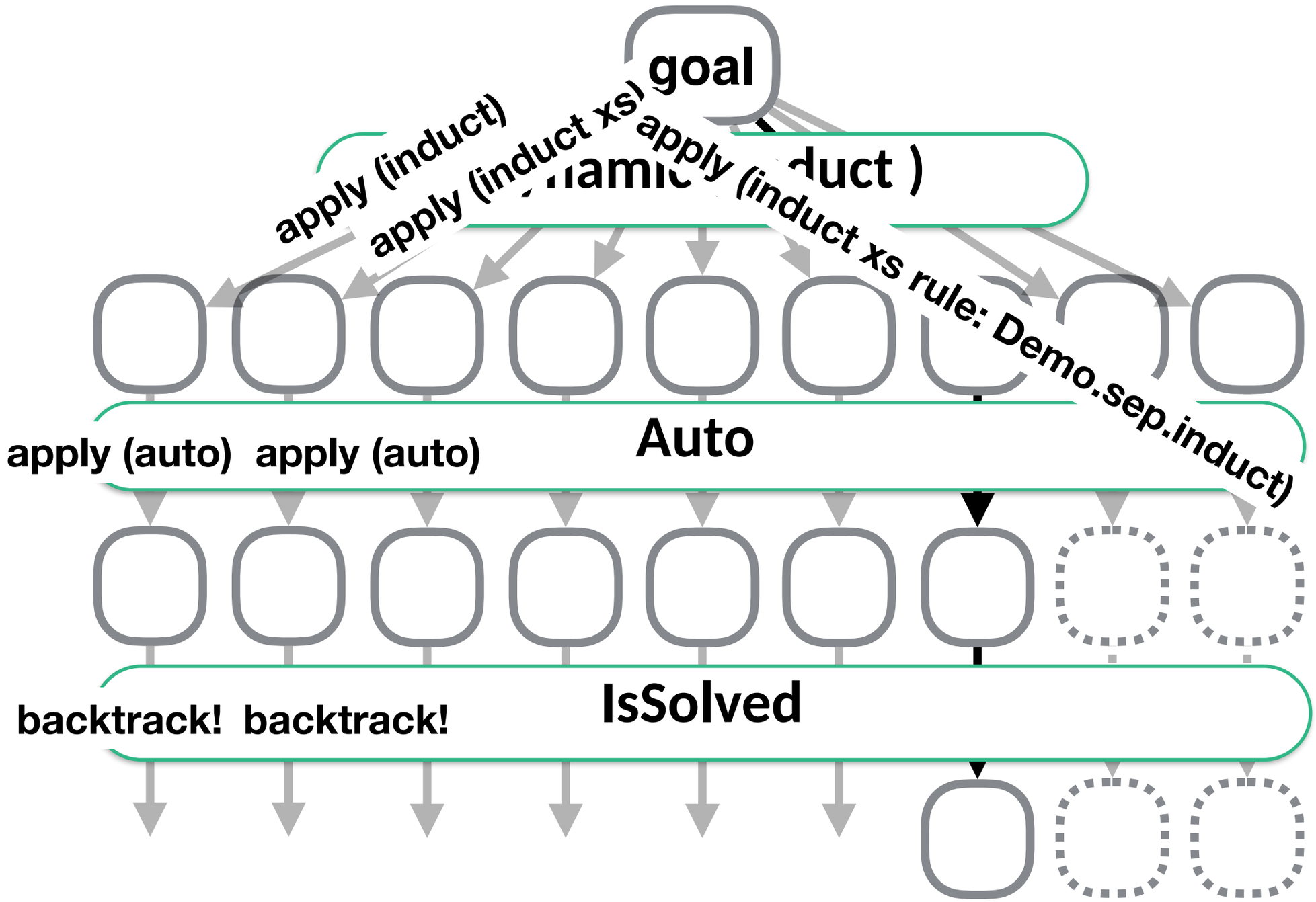
What I did before APLAS2020…

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



What I did before APLAS2020…

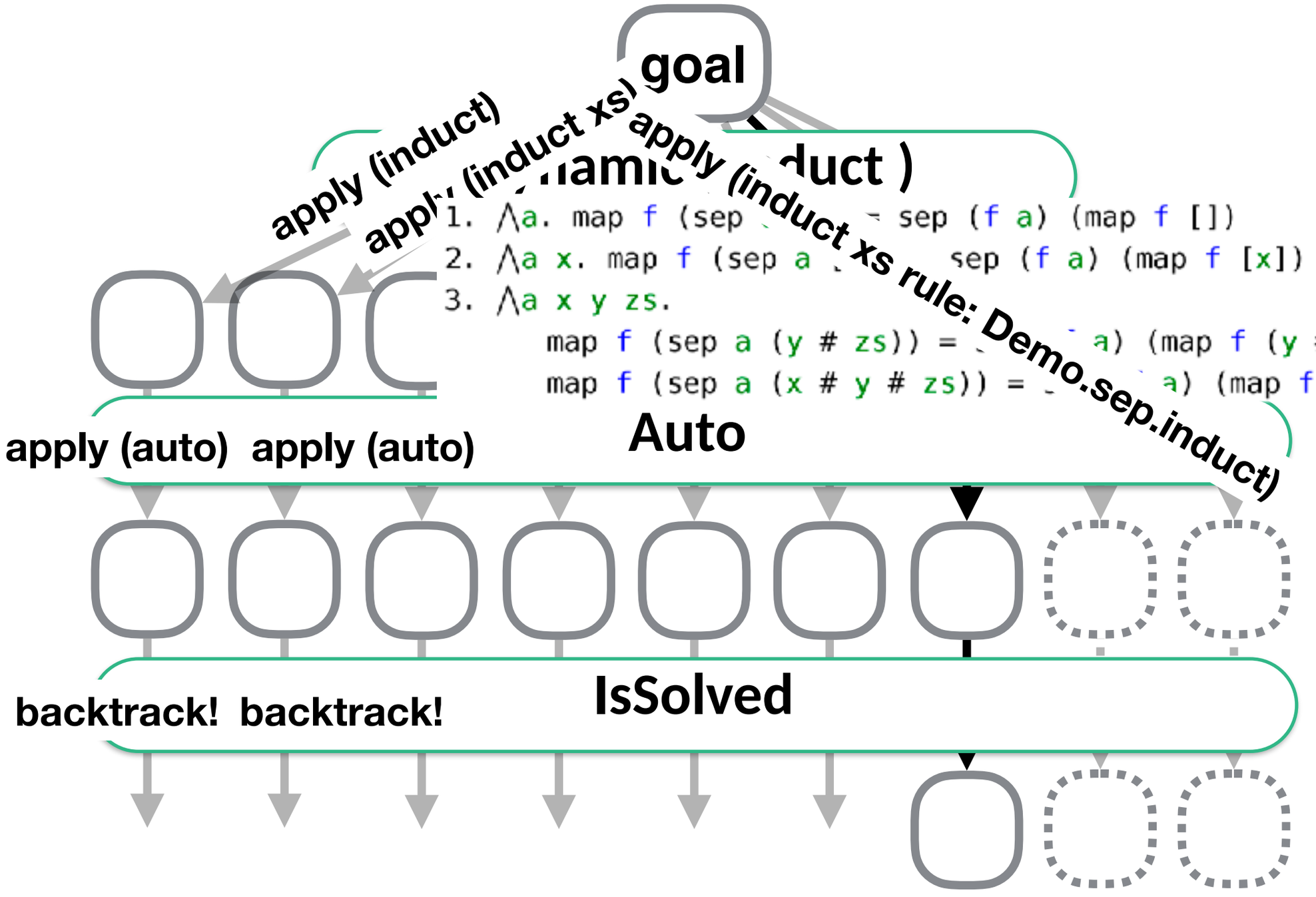
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

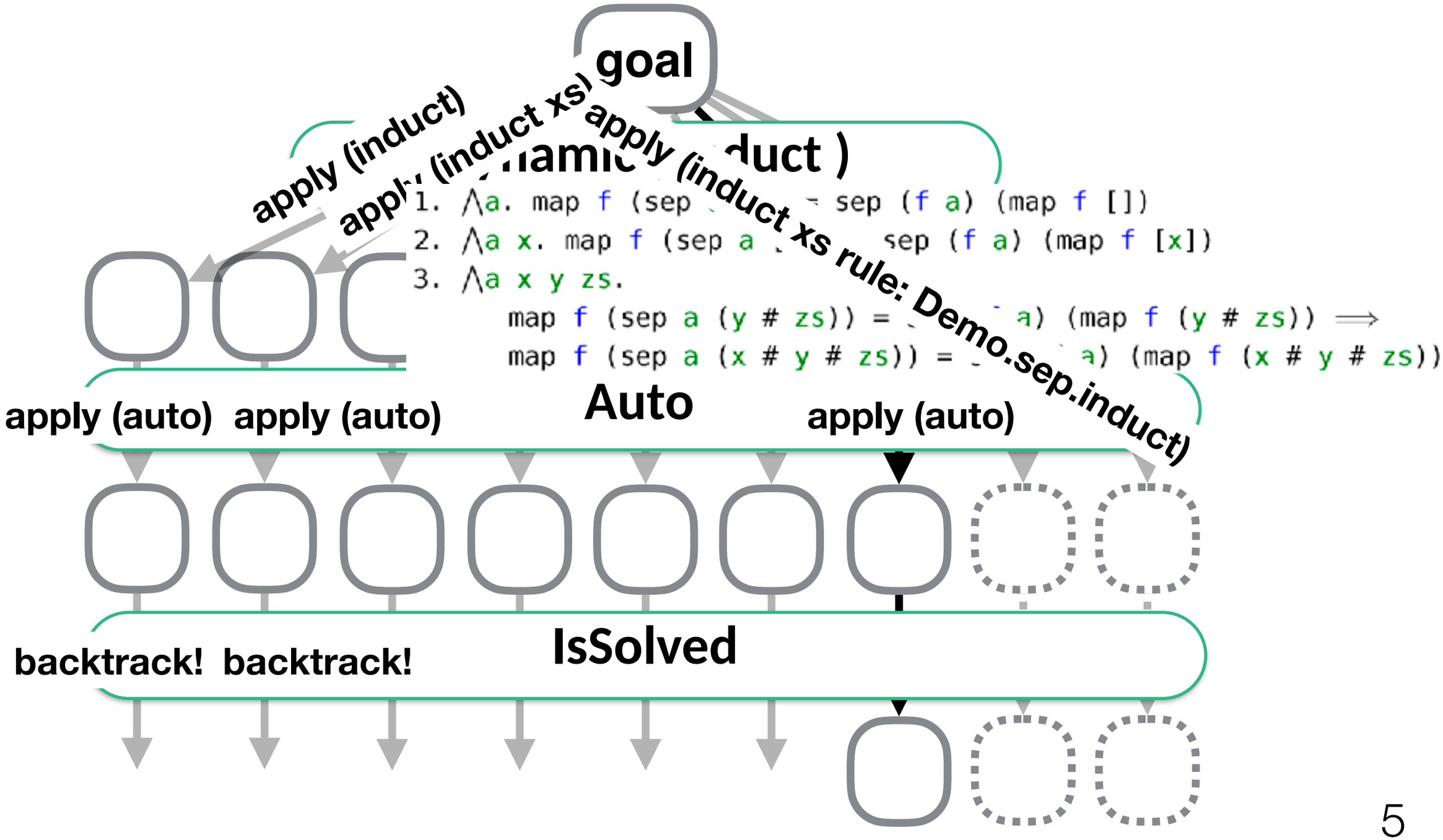
What I did before APLAS2020…

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



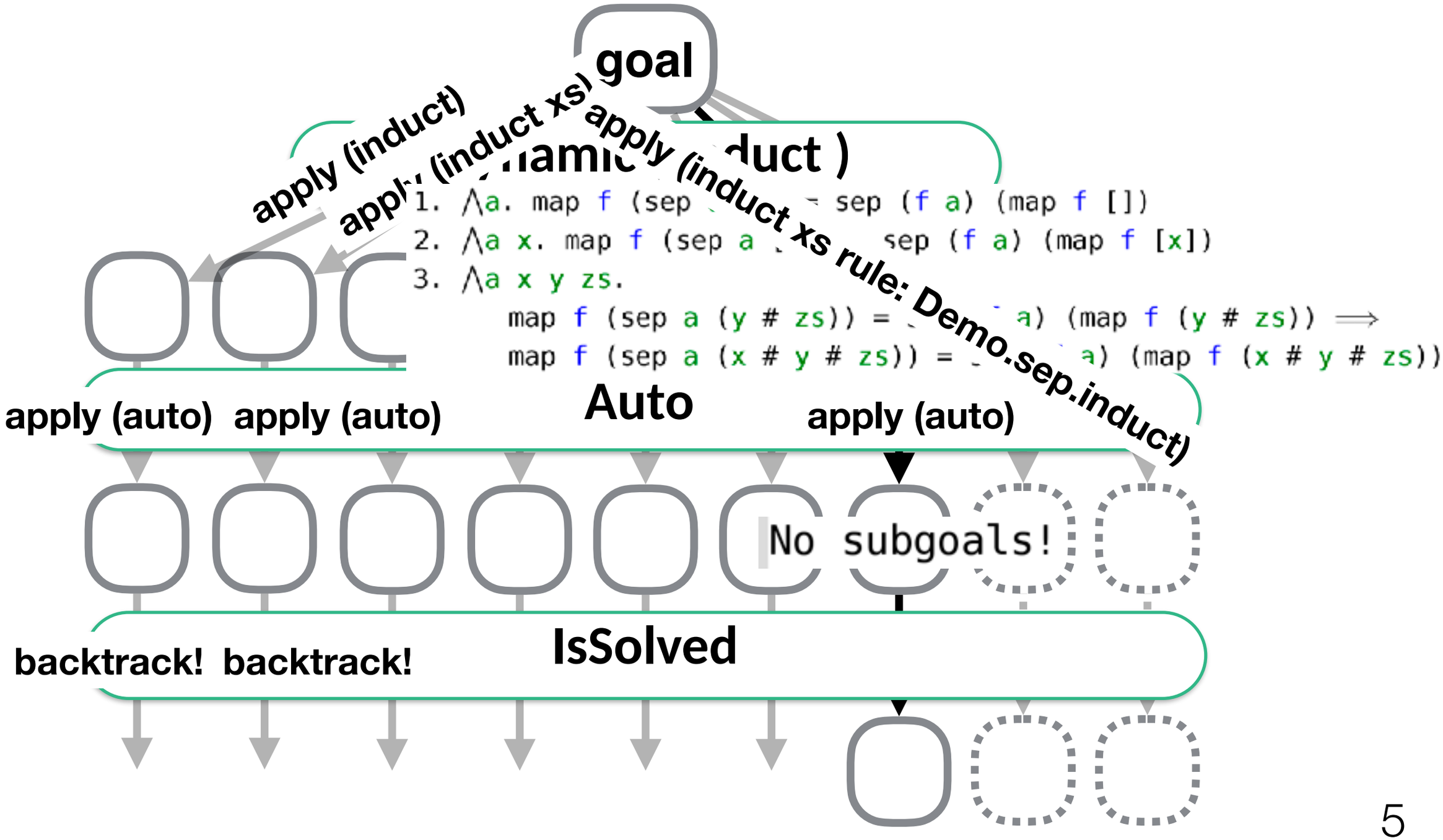
What I did before APLAS2020…

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



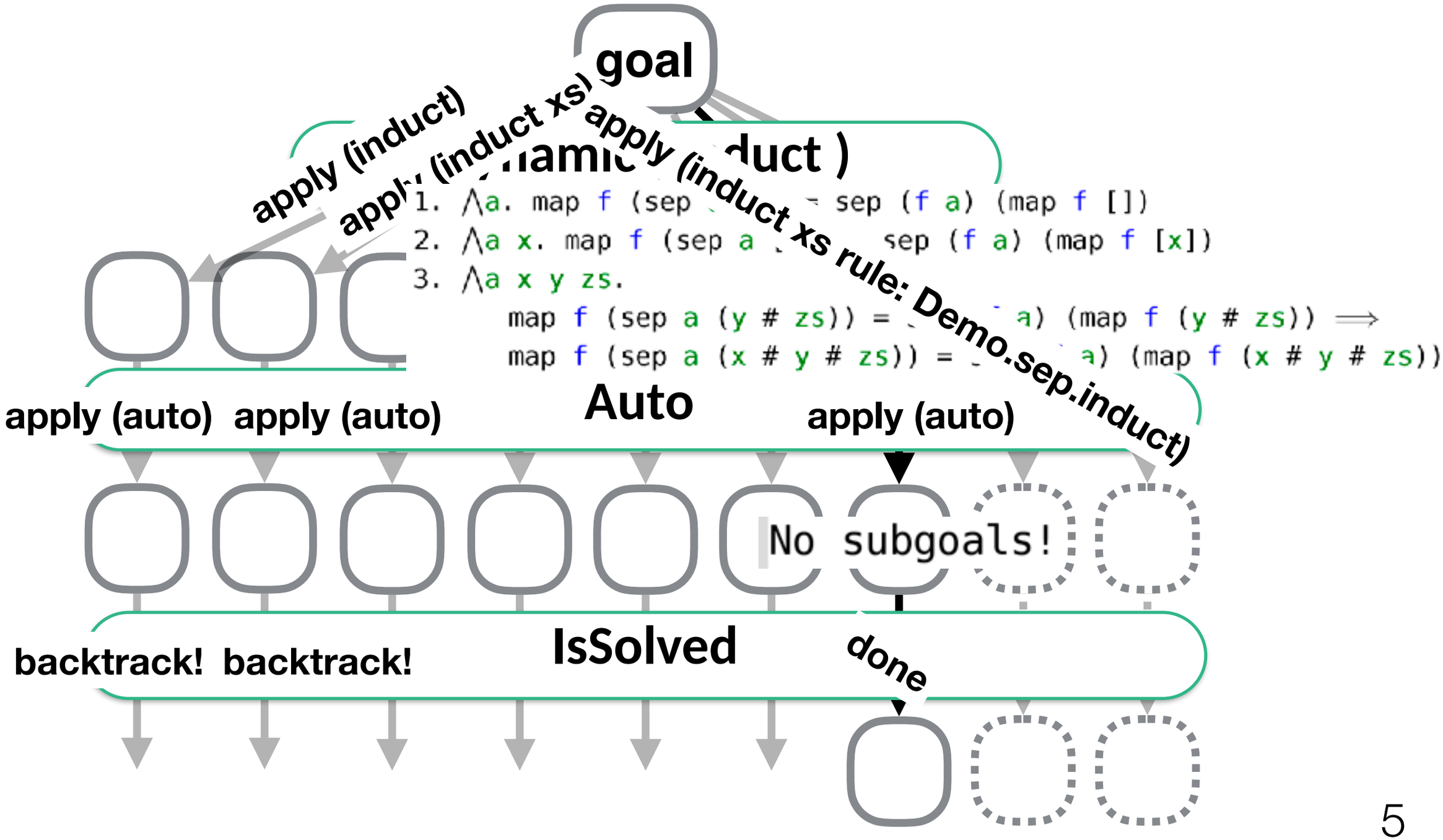
What I did before APLAS2020…

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



What I did before APLAS2020…

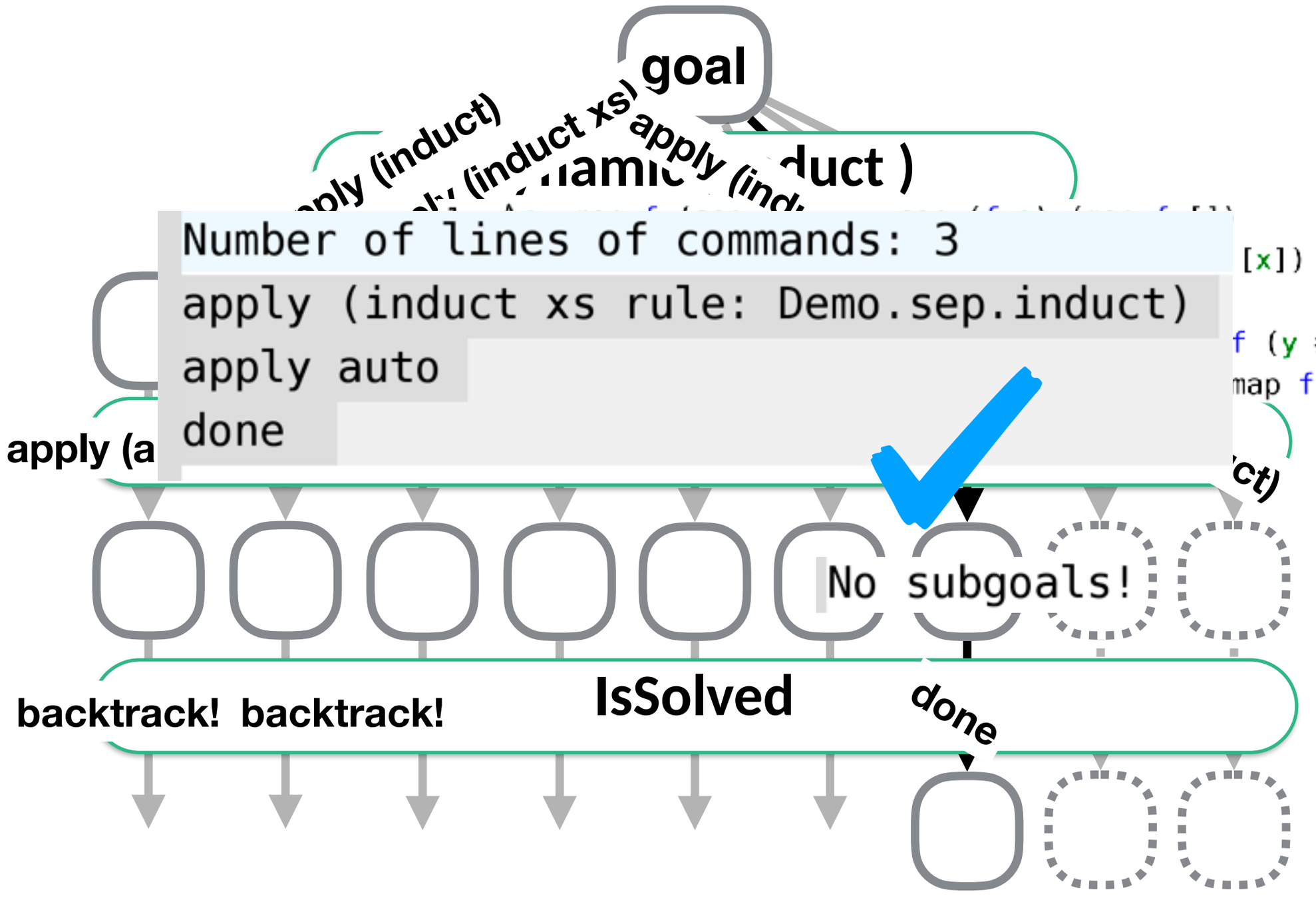
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



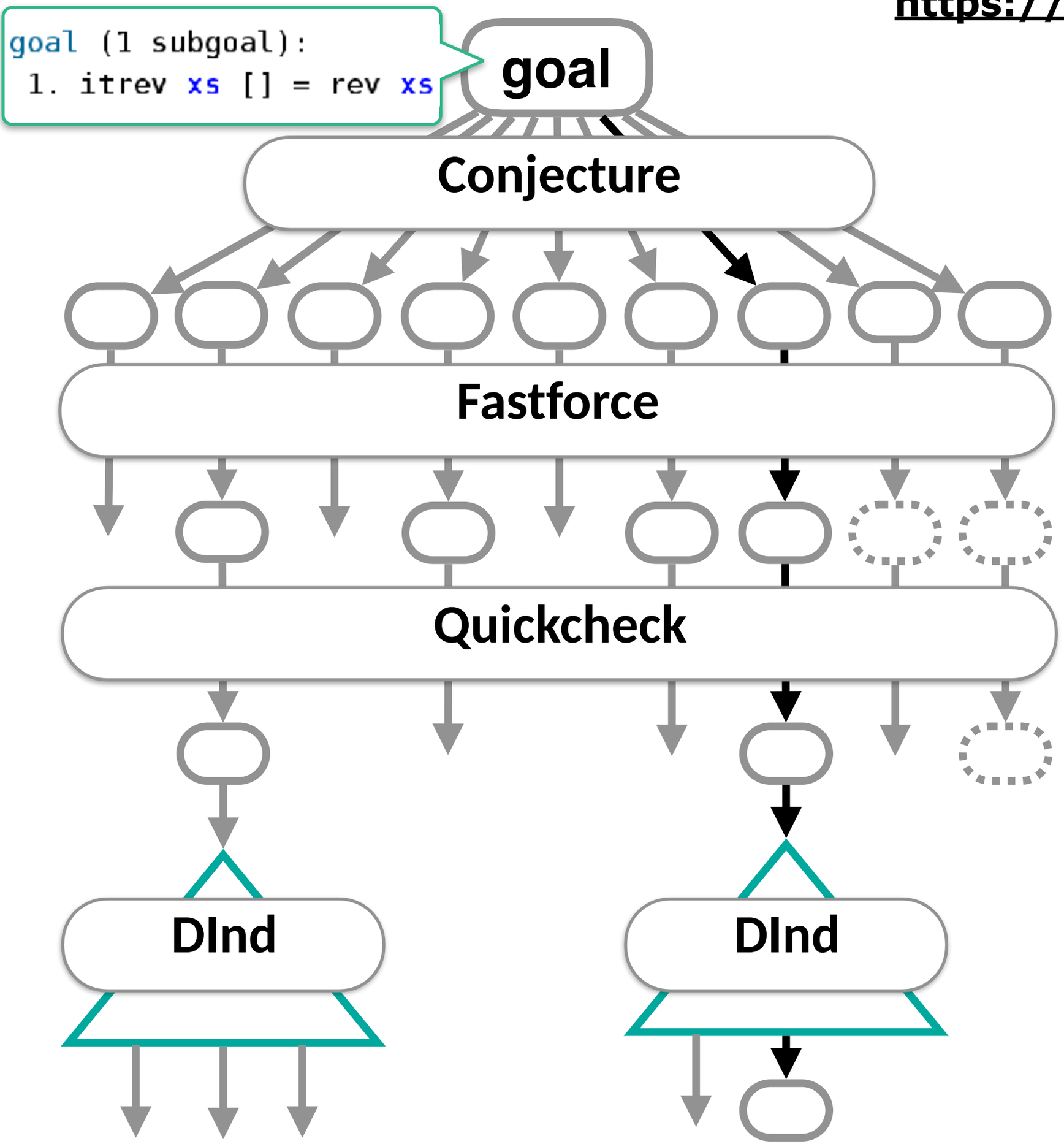
https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

What I did before APLAS2020…

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



https://tinyurl.com/up7zhyk

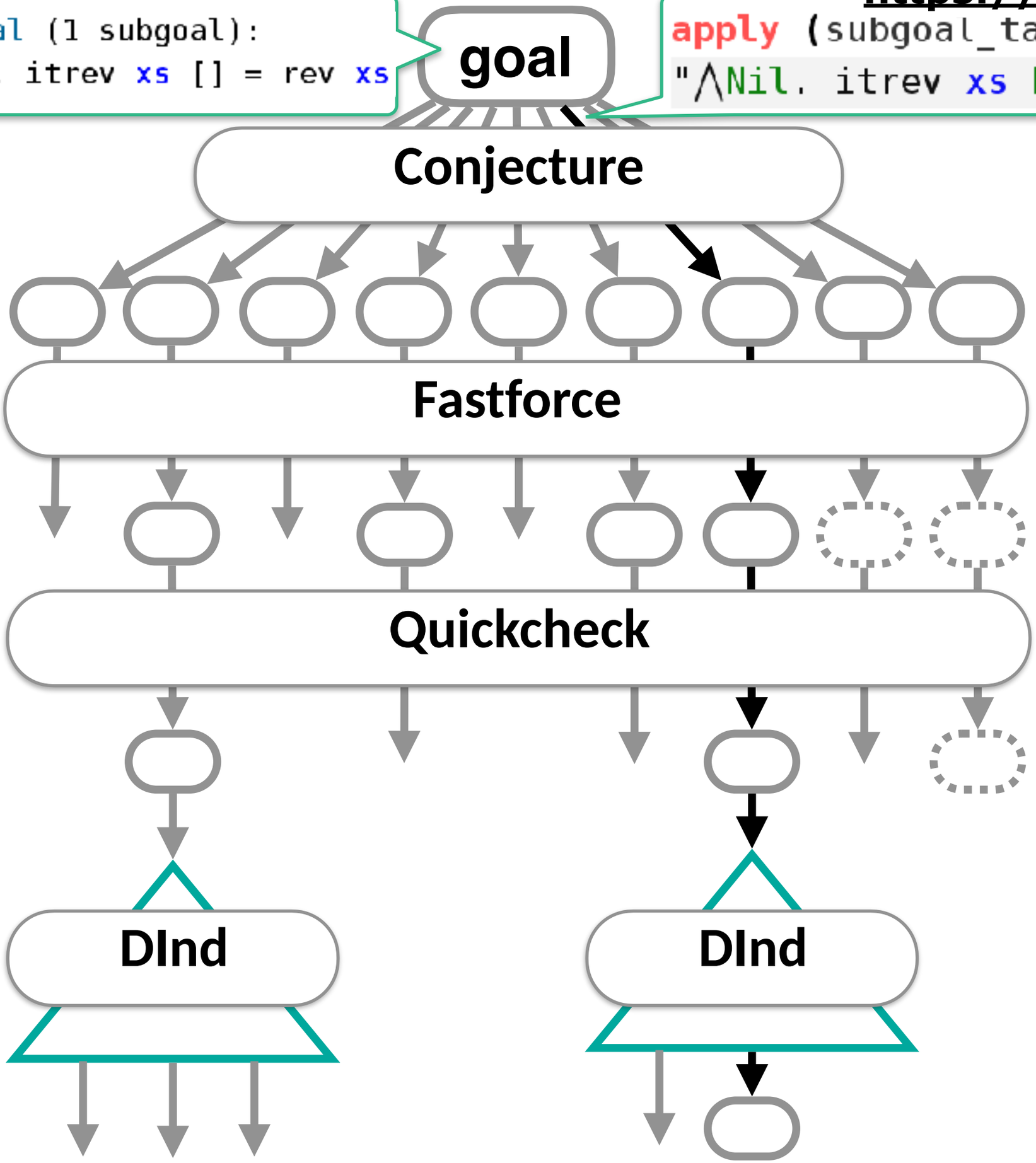


https://tinyurl.com/up7zhyk

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

goal

```
apply (subgoal_tac
"⋀Nil. itrev xs Nil = rev xs @ Nil")
```



https://tinyurl.com/up7zhyk

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

goal

```
apply (subgoal_tac
"⋀Nil. itrev xs Nil = rev xs @ Nil")
```

```
goal (2 subgoals):
 1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
    itrev xs [] = rev xs
 2. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Conjecture

Fastforce

Quickcheck

DInd

DInd

https://tinyurl.com/up7zhyk

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

goal

```
apply (subgoal_tac
"⋀Nil. itrev xs Nil = rev xs @ Nil")
```

```
goal (2 subgoals):
 1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
    itrev xs [] = rev xs
 2. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Conjecture

Fastforce

```
apply fastforce
```

Quickcheck

DInd

DInd

https://tinyurl.com/up7zhyk

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

goal

```
apply (subgoal_tac
"⋀Nil. itrev xs Nil = rev xs @ Nil")
```

```
goal (2 subgoals):
 1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
    itrev xs [] = rev xs
 2. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Conjecture

Fastforce

```
apply fastforce
```

```
goal (1 subgoal):
 1. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Quickcheck

DInd

DInd

https://tinyurl.com/up7zhyk

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

goal

```
apply (subgoal_tac
"⋀Nil. itrev xs Nil = rev xs @ Nil")
```

Conjecture

```
goal (2 subgoals):
 1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
    itrev xs [] = rev xs
 2. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Fastforce

```
apply fastforce
```

```
goal (1 subgoal):
 1. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Quickcheck

```
goal (1 subgoal):
 1. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

DInd

DInd

https://tinyurl.com/up7zhyk

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

goal

```
apply (subgoal_tac
"⋀Nil. itrev xs Nil = rev xs @ Nil")
```

Conjecture

```
goal (2 subgoals):
 1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
    itrev xs [] = rev xs
 2. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Fastforce

```
apply fastforce
```

```
goal (1 subgoal):
 1. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Quickcheck

```
goal (1 subgoal):
 1. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

DInd

DInd

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

https://tinyurl.com/up7zhyk

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

goal

```
apply (subgoal_tac
"⋀Nil. itrev xs Nil = rev xs @ Nil")
```

Conjecture

```
goal (2 subgoals):
 1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
    itrev xs [] = rev xs
 2. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Fastforce

```
apply fastforce
```

```
goal (1 subgoal):
 1. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Quickcheck

```
goal (1 subgoal):
 1. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

DInd

DInd

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

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

https://tinyurl.com/up7zhyk

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

goal

```
apply (subgoal_tac
"⋀Nil. itrev xs Nil = rev xs @ Nil")
```

Conjecture

```
goal (2 subgoals):
 1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
    itrev xs [] = rev xs
 2. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Fastforce

```
apply fastforce
```

```
Number of lines of commands: 5
apply (subgoal_tac "⋀Nil. itrev xs Nil = rev xs @ Nil")
apply fastforce
apply (induct xs)
apply auto
done
```

Nil

Nil

DInd

DInd

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

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

https://tinyurl.com/up7zhyk

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

goal

```
apply (subgoal_tac
"⋀Nil. itrev xs Nil = rev xs @ Nil")
```

```
goal (2 subgoals):
 1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
    itrev xs [] = rev xs
 2. ⋀Nil. itrev xs Nil = rev xs @ Nil
```

Conjecture

Fastforce

```
apply fastforce
```

```
Number of lines of commands: 5
apply (subgoal_tac "⋀Nil. itrev xs Nil = rev xs @ Nil")
apply fastforce
apply (induct xs)
apply auto
done
```

Nil

Nil

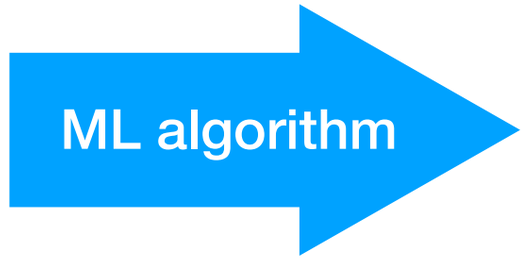
Difficult proof goals?
Search space?

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

https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

Introduction to Machine Learning in 10 seconds



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

https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

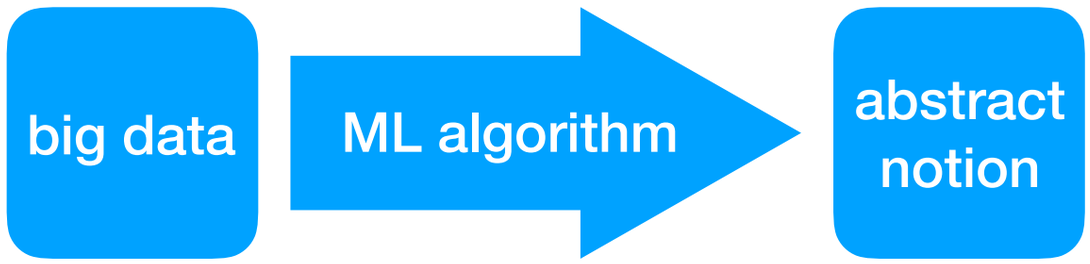
Introduction to Machine Learning in 10 seconds

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

https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

Introduction to Machine Learning in 10 seconds

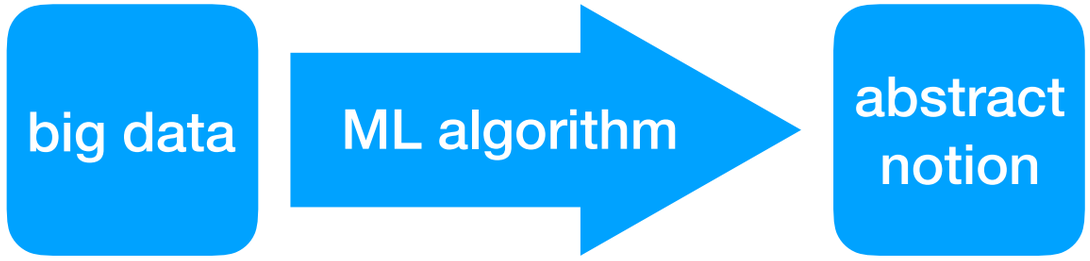


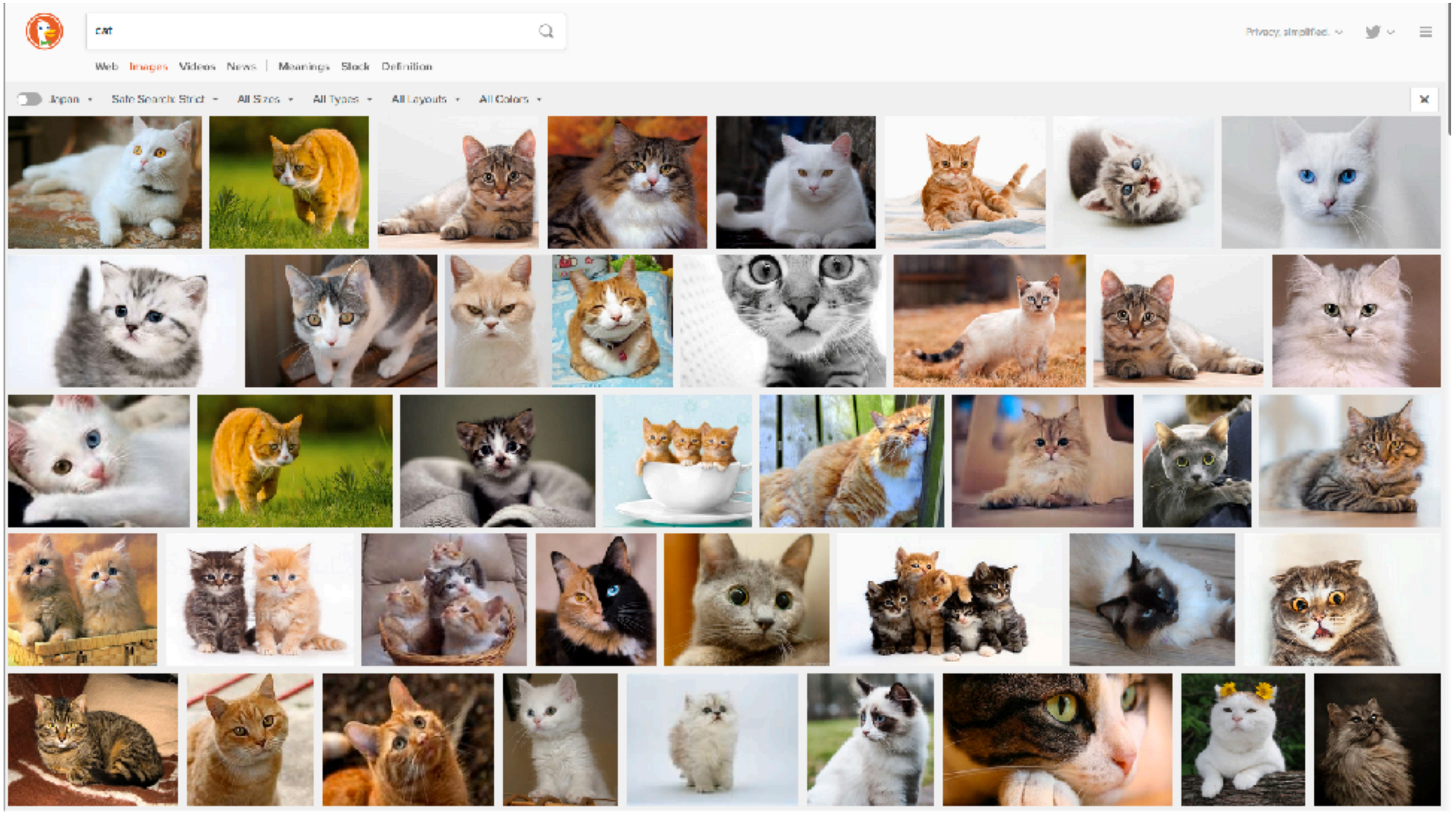
https://duckduckgo.com/?q=cat&t=ffab&iar=images&iax=images&ia=images

https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

Introduction to Machine Learning in 10 seconds



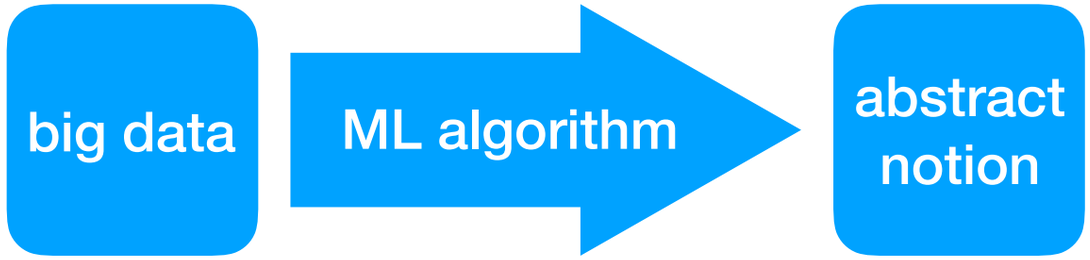


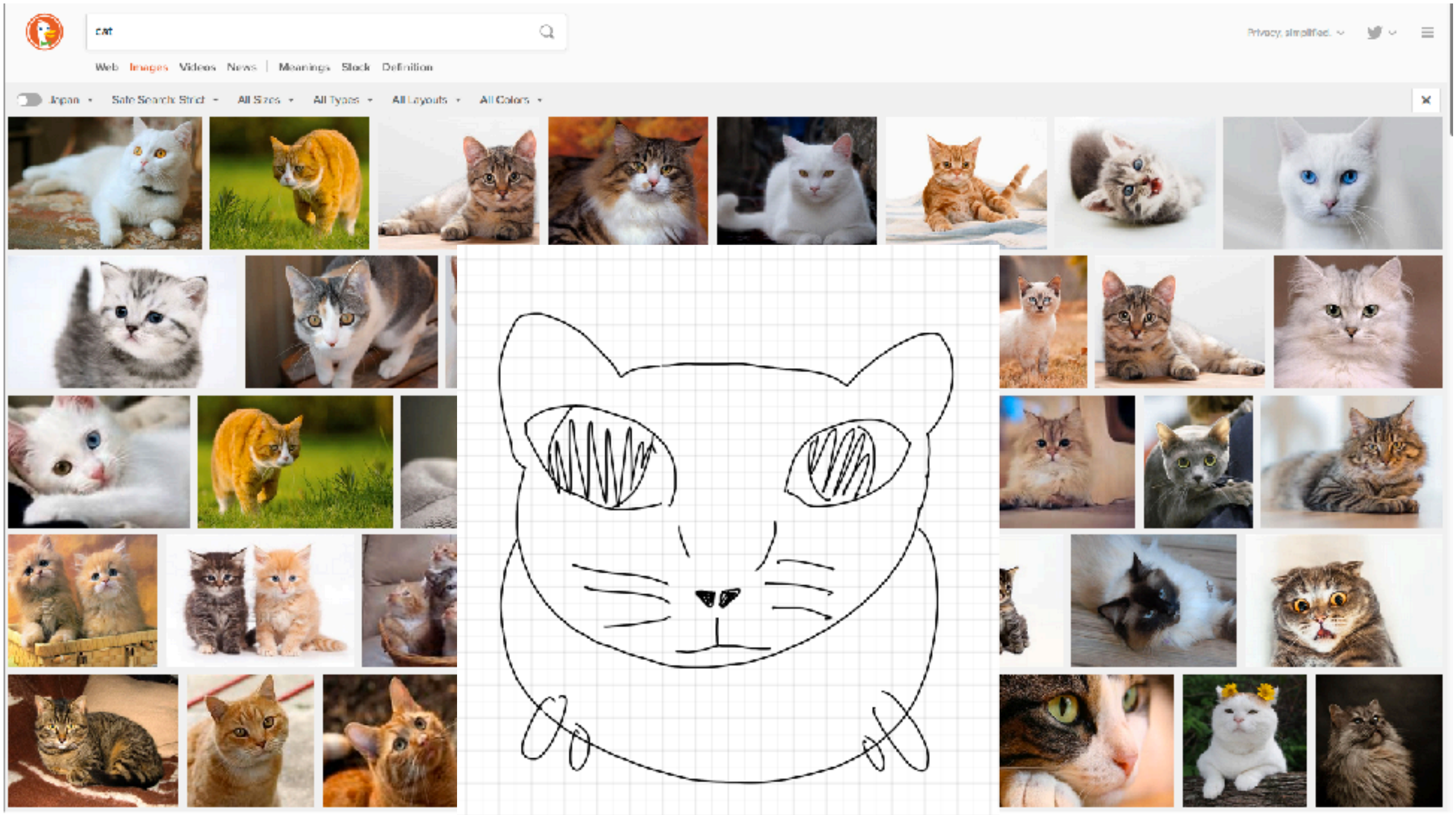
https://duckduckgo.com/?q=cat&t=ffab&iar=images&iax=images&ia=images

https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

Introduction to Machine Learning in 10 seconds





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

https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

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 https://tinyurl.com/up7zhyk

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 https://tinyurl.com/up7zhyk

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 https://tinyurl.com/up7zhyk

ML for Inductive Theorem Proving the BAD

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

<- one abstract representation

```
Failed to apply proof method⌂:
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 https://tinyurl.com/up7zhyk

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 method⌂:
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 https://tinyurl.com/up7zhyk

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
```

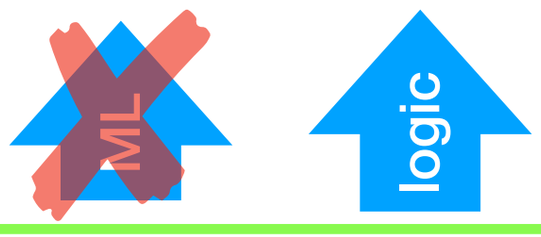
<- many concrete cases

https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

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
```

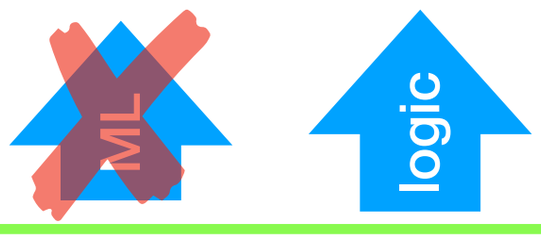
<- many concrete cases

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

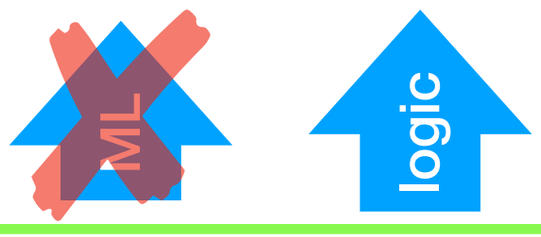
https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

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 https://tinyurl.com/up7zhyk

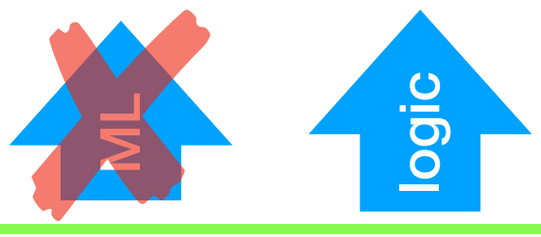
ML for Inductive Theorem Proving the BAD

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 https://tinyurl.com/up7zhyk

ML for Inductive Theorem Proving the BAD

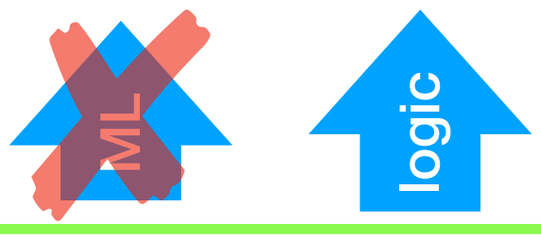
Higher-Order functions

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

ML for Inductive Theorem Proving the BAD

Higher-Order functions

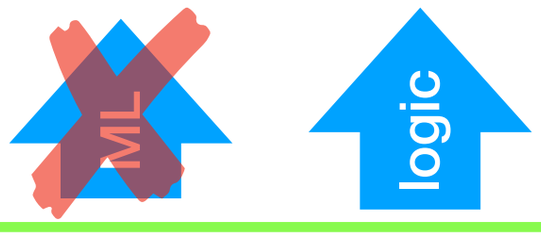
polymorphism **type class**

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

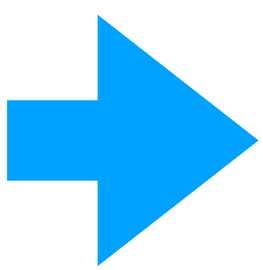
https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

ML for Inductive Theorem Proving the BAD

Higher-Order functions

polymorphism **type class**

universal quantifier

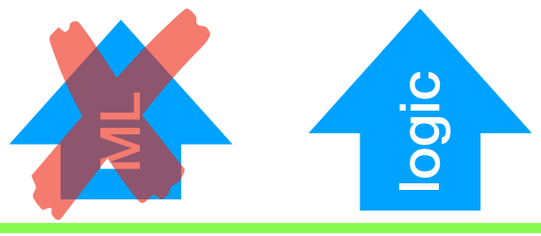


concise formula that can cover many concrete cases

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

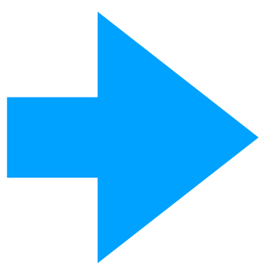
https://tinyurl.com/up7zhyk

ML for Inductive Theorem Proving the BAD

Higher-Order functions

polymorphism **type class**

universal quantifier



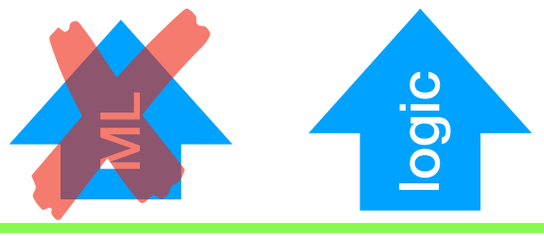
concise formula that can cover many concrete cases

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

https://tinyurl.com/up7zhyk

ML for Inductive Theorem Proving the BAD

Higher-Order functions

polymorphism

type class

universal quantifier

lambda abstraction

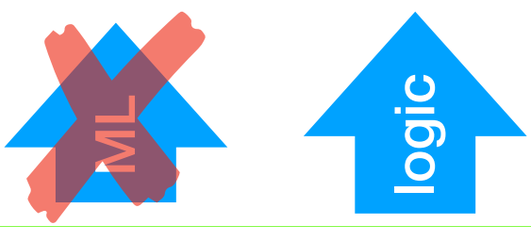
concise formula that can cover many concrete cases

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
```

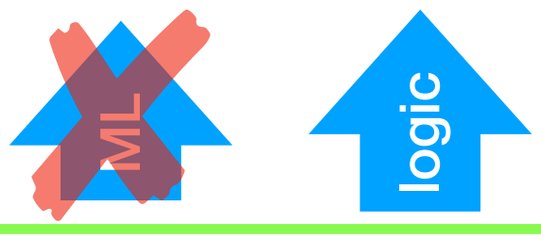
<- many concrete cases

https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

```
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
```

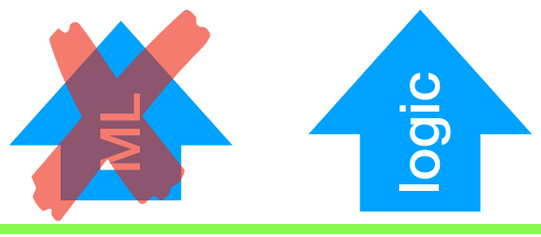
<- many concrete cases

https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

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 https://tinyurl.com/up7zhyk

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

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

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 https://tinyurl.com/up7zhyk

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 https://tinyurl.com/up7zhyk

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 (crossed out)

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 https://tinyurl.com/up7zhyk

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

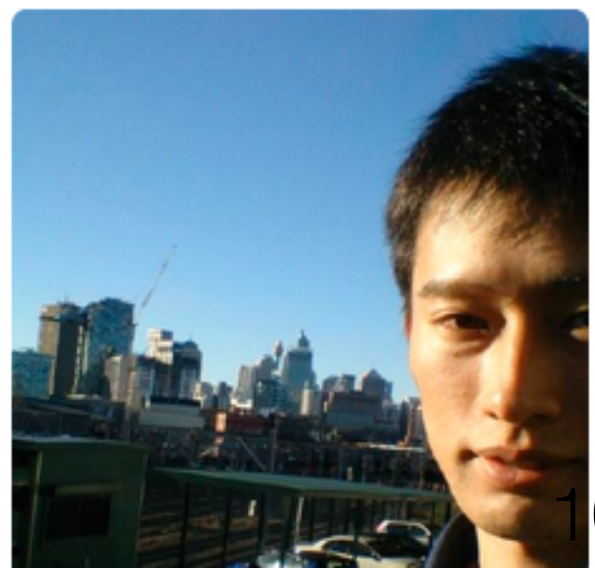
Language understanding

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

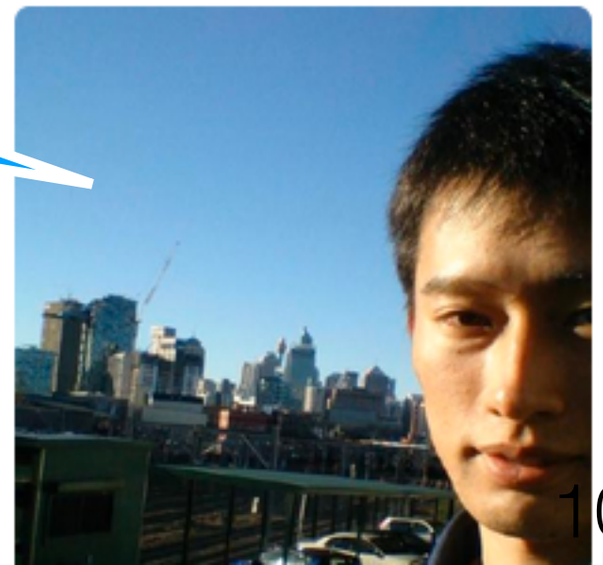
Abstract concepts?

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

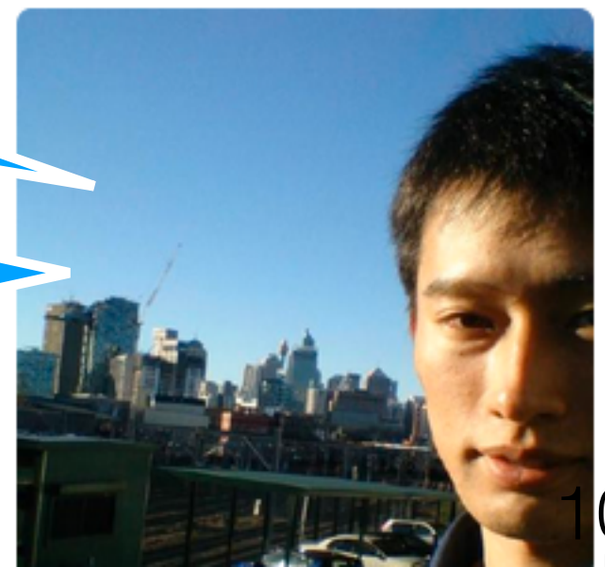
March 20, 2019

Abstract concepts?

Why not logic?
(LiFtEr = Logical Feature Extraction)

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.

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

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)

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

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

Big Picture

<- pros: good at ambiguity (heuristics)

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

<- pros: good at rigorous abstraction

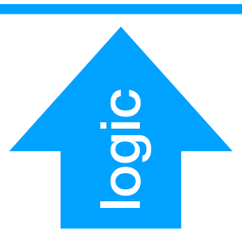
```
lemma "star r x y ⟹ star r y z ⟹ star r x z"
by(induction rule: star.induct)(auto simp: step)

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



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

<- pros: good at rigorous abstraction

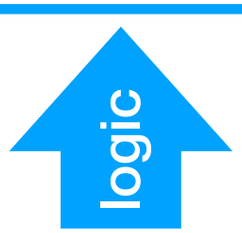
```
lemma "star r x y ⟹ star r y z ⟹ star r x z"
by(induction rule: star.induct)(auto simp: step)

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



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

LiFtEr

LiFtEr

LiFtEr 3

<- 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

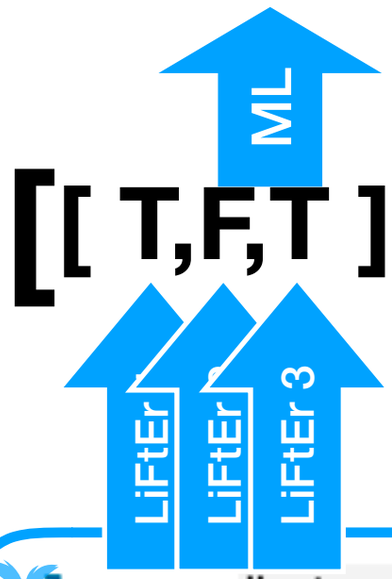
<- abstraction using expressive logic

```
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)

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

<- pros: good at rigorous abstraction

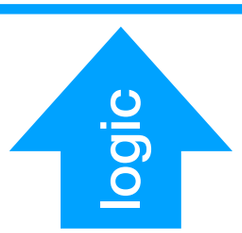
```
lemma "star r x y ⟹ star r y z ⟹ star r x z"
by(induction rule: star.induct)(auto simp: step)

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



<- 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], [ ]]: bool list

<- pros: good at ambiguity (heuristics)

<- simple representation

<- pros: good at rigorous abstraction

LiFtEr

LiFtEr

LiFtEr 3

LiFtEr 1

LiFtEr 2

LiFtEr 3

```
lemma "star r    star r y z ⟹ star r x z"
by(induction r      ar.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.

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

```
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

<- 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
```

<- 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

Abstract Syntax of LiFtEr

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

Abstract Syntax of LiFtEr

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

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

Example Assertion in LiFtEr (in Abstract Syntax)

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

$\rightarrow$

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

variable for auxiliary lemmas

variable for terms

conjunction

Example Assertion in LiFtEr (in Abstract Syntax)

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

implication

variable for auxiliary lemmas

variable for terms

variable for term occurrences

conjunction

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
```

implication

variable for auxiliary lemmas

variable for terms

variable for term occurrences

conjunction

variable for natural numbers

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
```

implication

variable for auxiliary lemmas

variable for terms

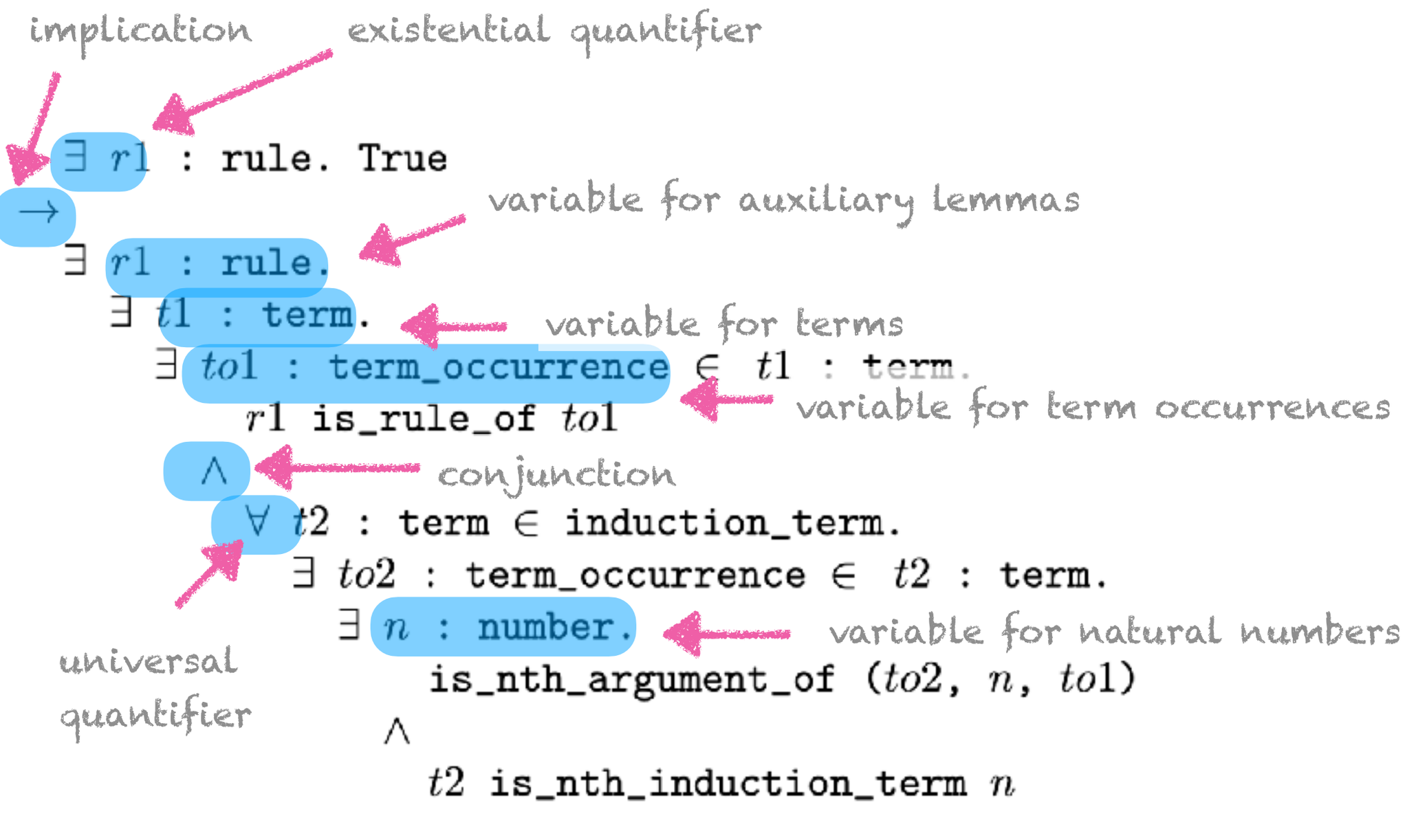
variable for term occurrences

conjunction

universal quantifier

variable for natural numbers

Example Assertion in LiFtEr (in Abstract Syntax)



Example Assertion in LiFtEr (in Abstract Syntax)

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

implication

existential quantifier

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

variable for auxiliary lemmas

variable for terms

variable for term occurrences

conjunction

universal quantifier

variable for natural numbers

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
```

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 = 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.
  ∃ 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)

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
```

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)

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
```

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
```

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
```

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
```

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
```

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
```

r1

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

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

t2

r1

∃ r1 : rule. True

→

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

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 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 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 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)"
```

to1 first to2

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

t2 r1

first

```
∃ 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 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 second to2 t2 second first r1

∃ r1 : rule. True

→

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

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 · r1

$\exists$ *r1* : rule. True

$\rightarrow$

$\exists$ *r1* : rule. (r1 = itrev.induct)
  $\exists$ *t1* : term. (t1 = itrev)
    $\exists$ *to1* : term_occurrence $\in$ *t1* : term. (to1 = itrev)
      r1 is_rule_of *to1* True! r1 (= itrev.induct) is a lemma about to1 (= itrev).
    $\wedge$
      $\forall$ *t2* : term $\in$ induction_term. (t2 = xs and ys)
        $\exists$ *to2* : term_occurrence $\in$ *t2* : term. (to2 = xs and ys)
          $\exists$ *n* : number.
            is_nth_argument_of (*to2*, *n*, *to1*) when t2 is xs (n = 1)
          $\wedge$
            t2 is_nth_induction_term *n* when t2 is ys (n = 2)

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 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 ->

```
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

new types ->

new constants ->

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

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

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

new lemma ->

a model proof ->

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

∃ r1 : rule. True

→

r1

(r1 = exec.induct)

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

new types ->

new constants ->

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

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

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

new lemma ->

a model proof ->

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

∃ r1 : rule. True

→

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

r1

(r1 = exec.induct)

new types ->

new constants ->

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

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

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

new lemma ->

a model proof ->

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

∃ r1 : rule. True

→

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

r1

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

new types ->

new constants ->

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

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

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

new lemma ->
a model proof ->

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

to1

r1

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

∃ r1 : rule. True

→

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

new types ->

new constants ->

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

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

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

new lemma ->

a model proof ->

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

to1

r1

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

∃ r1 : rule. True

→

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

new types ->

new constants ->

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

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

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

new lemma ->

a model proof ->

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

to1

r1

∃ r1 : rule. True

→

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

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

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

new types ->

new constants ->

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

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

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

new lemma ->
a model proof ->

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

to1

r1

∃ r1 : rule. True

→

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

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

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

new types ->

new constants ->

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

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

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

new lemma ->
a model proof ->
∃ r1 : rule. True

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

to1

r1

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

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

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

new types ->

new constants ->

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

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

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

new lemma ->

a model proof ->

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

to1

r1

∃ r1 : rule. True

→

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

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

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

new types ->

new constants ->

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

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

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

new lemma ->
a model proof ->

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

t2 t01

∃ r1 : rule. True

→

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

r1

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

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

(t2 = is1, s, and stk)

new types ->

new constants ->

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

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

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

new lemma ->
a model proof ->

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

t2 t01 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

(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 r1

$\exists$ r1 : rule. True

$\rightarrow$

$\exists$ $r1$: rule. (r1 = exec.induct)
$\exists$ $t1$: term. (t1 = exec)
$\exists$ $to1$: term_occurrence $\in$ $t1$: term. (to1 = exec)
r1 is_rule_of $to1$ True! r1 (= exec.induct) is a lemma about to1 (= exec).
$\wedge$
$\forall$ $t2$: term $\in$ induction_term. (t2 = is1, s, and stk)
$\exists$ $to2$: term_occurrence $\in$ $t2$: term. (to2 = is1, s, and stk)
$\exists$ n : number.
is_nth_argument_of ($to2$, n, $to1$)
$\wedge$
$t2$ is_nth_induction_term n

new types ->

new constants ->

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

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

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

new lemma ->
a model proof ->

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

t2 t01 t02 r1

$\exists$ r1 : rule. True

$\rightarrow$

$\exists$ $r1$: rule. (r1 = exec.induct)
$\exists$ $t1$: term. (t1 = exec)
$\exists$ $to1$: term_occurrence $\in$ $t1$: term. (to1 = exec)
$r1$ is_rule_of $to1$ True! r1 (= exec.induct) is a lemma about to1 (= exec).
$\wedge$
$\forall$ $t2$: term $\in$ induction_term. (t2 = is1, s, and stk)
$\exists$ $to2$: term_occurrence $\in$ $t2$: term. (to2 = is1, s, and stk)
$\exists$ n : number.
is_nth_argument_of ($to2$, n, $to1$)
$\wedge$
$t2$ is_nth_induction_term n

new types ->

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

new constants ->

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

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

new lemma ->

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

a model proof ->

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

t2 first

to1 first

r1

to2

∃ r1 : rule. True

→

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

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

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

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

when t2 is is1 (n -> 1) ?

new types ->

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

new constants ->

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

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

new lemma ->

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

a model proof ->

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

first, t2, to1, r1, to2

∃ r1 : rule. True

→

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

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

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

∧

∀ t2 : term ∈ induction_term.

(t2 = is1, s, and stk)

∃ to2 : term_occurrence ∈ t2 : term.

(to2 = is1, s, and stk)

∃ n : number.

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

∧

t2 is_nth_induction_term n

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
```

first
second
t2
to1
to2
r1

$\exists$ r1 : rule. True

$\rightarrow$

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

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

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

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

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

new types ->

new constants ->

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

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

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

first second

new lemma ->

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

t2 to1 to2

a model proof ->

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

r1

first second

$\exists$ r1 : rule. True

$\rightarrow$

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

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

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

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

$\wedge$

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

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

$\exists$ n : number.

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

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

$t2$ is_nth_induction_term n

new types ->

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

new constants ->

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

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

new lemma ->

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

a model proof ->

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

∃ r1 : rule. True

→

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

first second third — t2
first second third — to1
r1 — to1
first second third — to2

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

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

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

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

new types ->

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

new constants ->

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

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

new lemma ->

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

a model proof ->

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

∃ r1 : rule. True

→

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

first second third — t2

first second third — to1

r1 — to2

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

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

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

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

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

to2

r1

(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)

WIP

<- pros: good at rigor

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

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

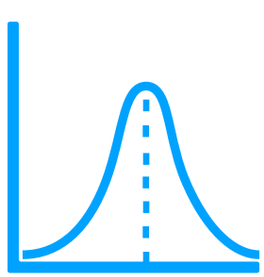
<- abstraction using expressive logic

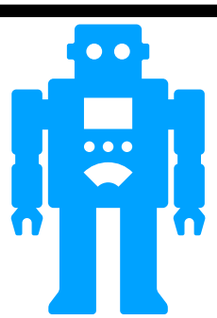
```
lemma "itrev [1,2]         [] = rev [1,2]         @ []" by auto
lemma "itrev [1,2,3]       [] = rev [1,2,3]       @ []" by auto
lemma "itrev [''a'',''b''] [] = rev [''a'',''b''] @ []" by auto
lemma "itrev [x,y,z]       [] = rev [x,y,z]       @ []" by auto
```

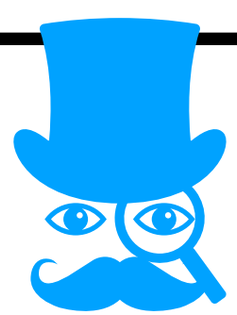
<- many concrete cases

https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk



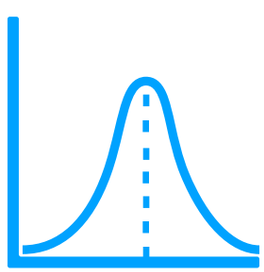


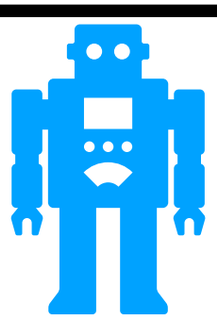


https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

Isabelle's proof methods
(deductive reasoning)



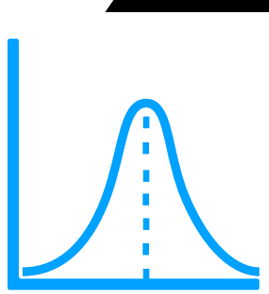


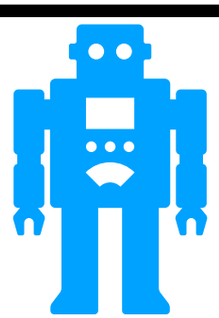
https://twitter.com/YutakangE

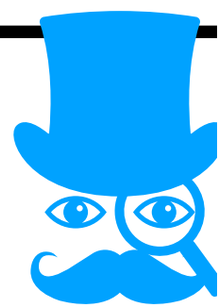
https://tinyurl.com/up7zhyk

Isabelle's proof methods
(deductive reasoning)

goal-oriented conjecturing
(abductive reasoning)





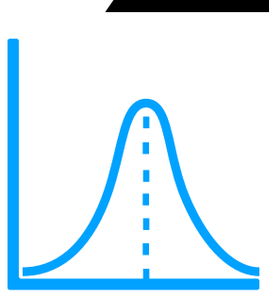


https://twitter.com/YutakangE

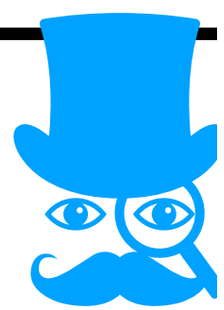
https://tinyurl.com/up7zhyk

Isabelle's proof methods
(deductive reasoning)

machine learning induction
(inductive reasoning)



goal-oriented conjecturing
(abductive reasoning)



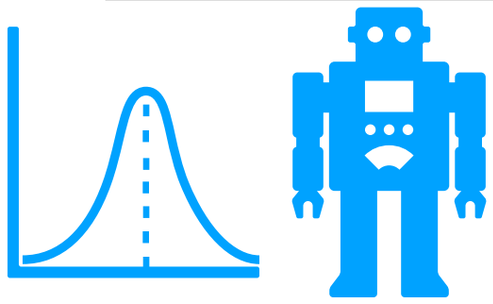
https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

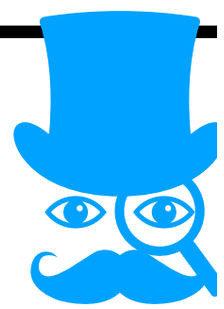
Isabelle's proof methods
(deductive reasoning)

united reasoning for automatic induction

machine learning induction
(inductive reasoning)



goal-oriented conjecturing
(abductive reasoning)



https://twitter.com/YutakangE

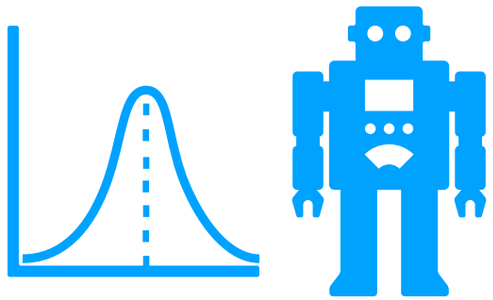
https://tinyurl.com/up7zhyk

Isabelle's proof methods
(deductive reasoning)

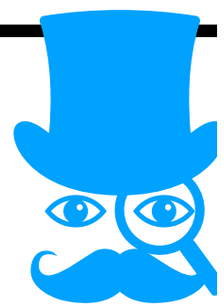
united reasoning for automatic induction

depth first search -> best first search

machine learning induction
(inductive reasoning)



goal-oriented conjecturing
(abductive reasoning)



https://twitter.com/YutakangE

https://tinyurl.com/up7zhyk

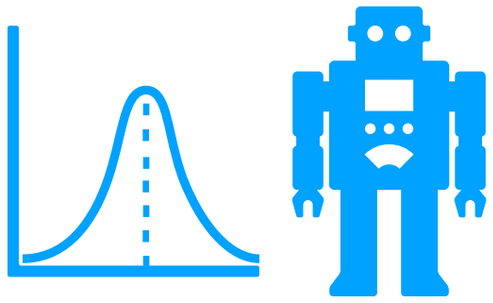
Isabelle's proof methods
(deductive reasoning)

united reasoning for automatic induction

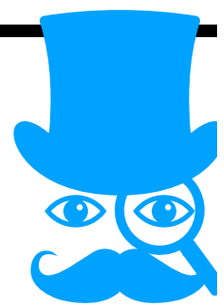
fin.

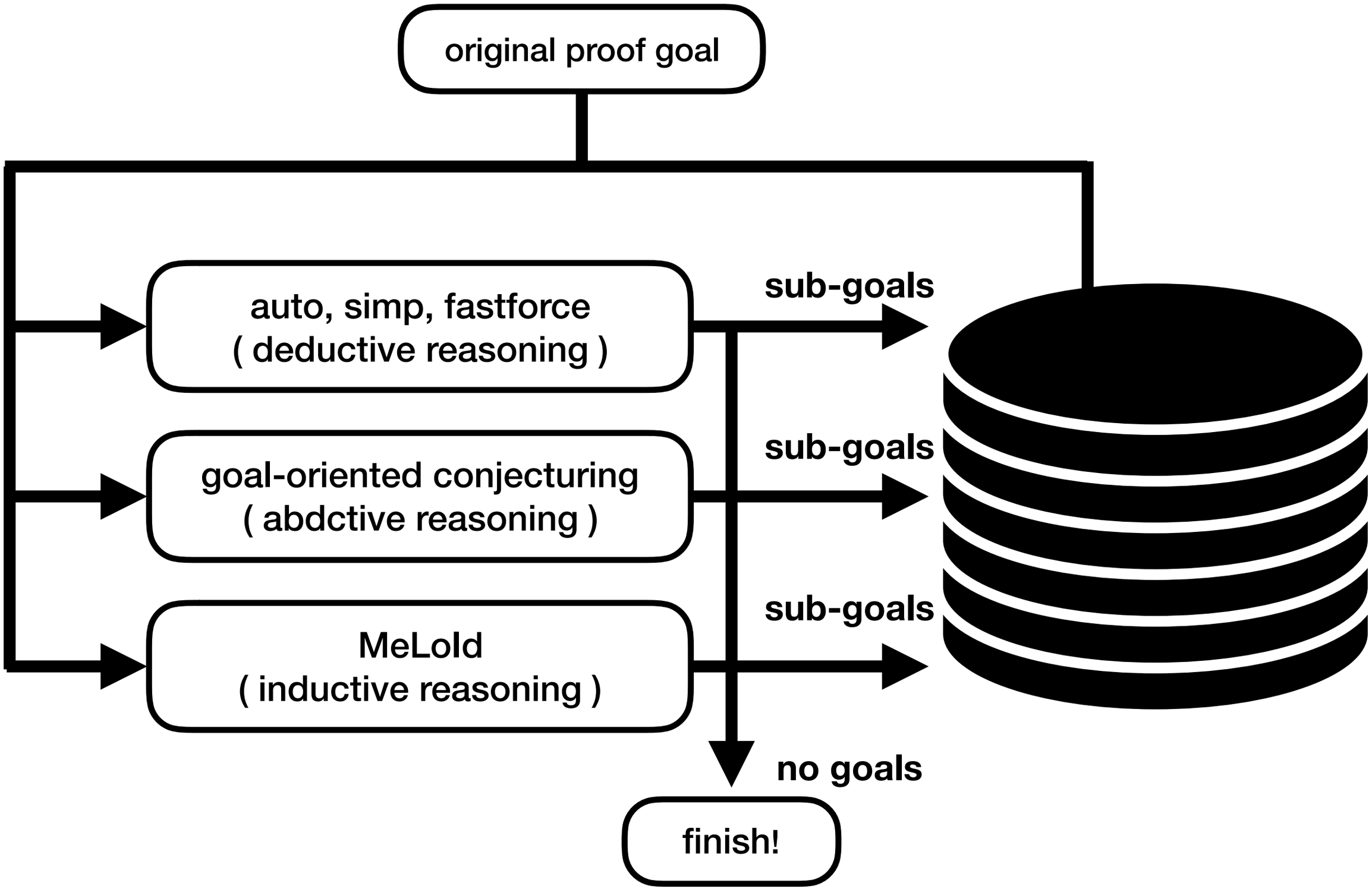
depth first search -> best first search

machine learning induction
(inductive reasoning)



goal-oriented conjecturing
(abductive reasoning)




original proof goal
auto, simp, fastforce
(deductive reasoning)
goal-oriented conjecturing
(abdctive reasoning)
MeLoId
(inductive reasoning)
sub-goals
sub-goals
sub-goals
no goals
finish!

https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

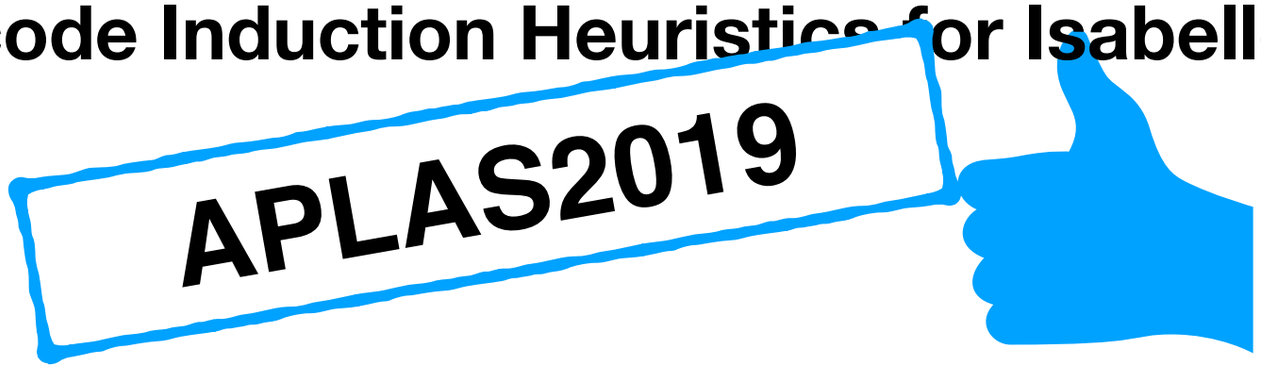
3 secrets behind LiFtEr

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

3 secrets behind LiFtEr

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL



https://twitter.com/YutakangE https://tinyurl.com/up7zhyk

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 https://tinyurl.com/up7zhyk

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://tinyurl.com/up7zhyk

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**

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

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                ist" 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 <- st

```
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.