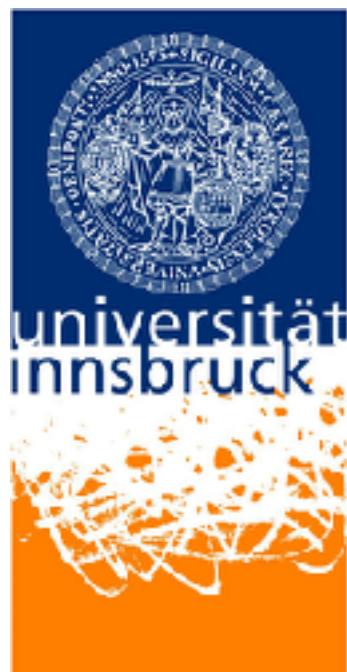
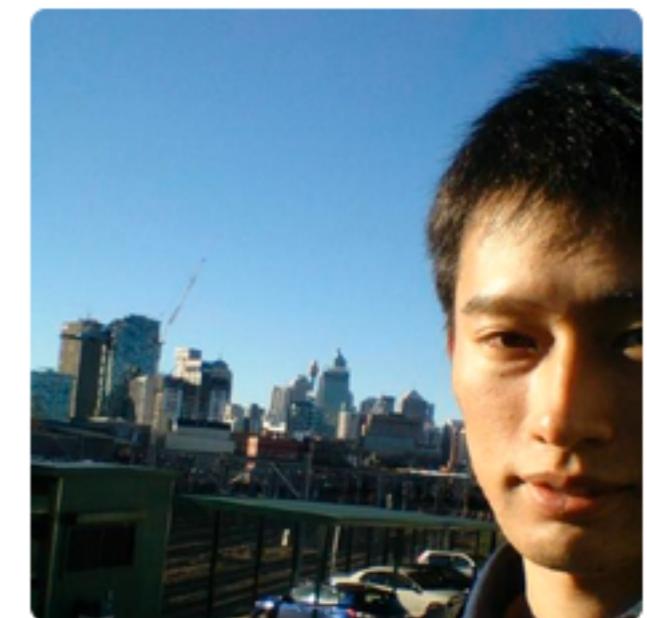


<https://twitter.com/YutakangE>

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL



Yutaka Nagashima
University of Innsbruck
Czech Technical University



**CZECH INSTITUTE
OF INFORMATICS
ROBOTICS AND
CYBERNETICS
CTU IN PRAGUE**

Yutaka Ng
yutakang

[Block or report user](#)

<https://twitter.com/YutakangE>

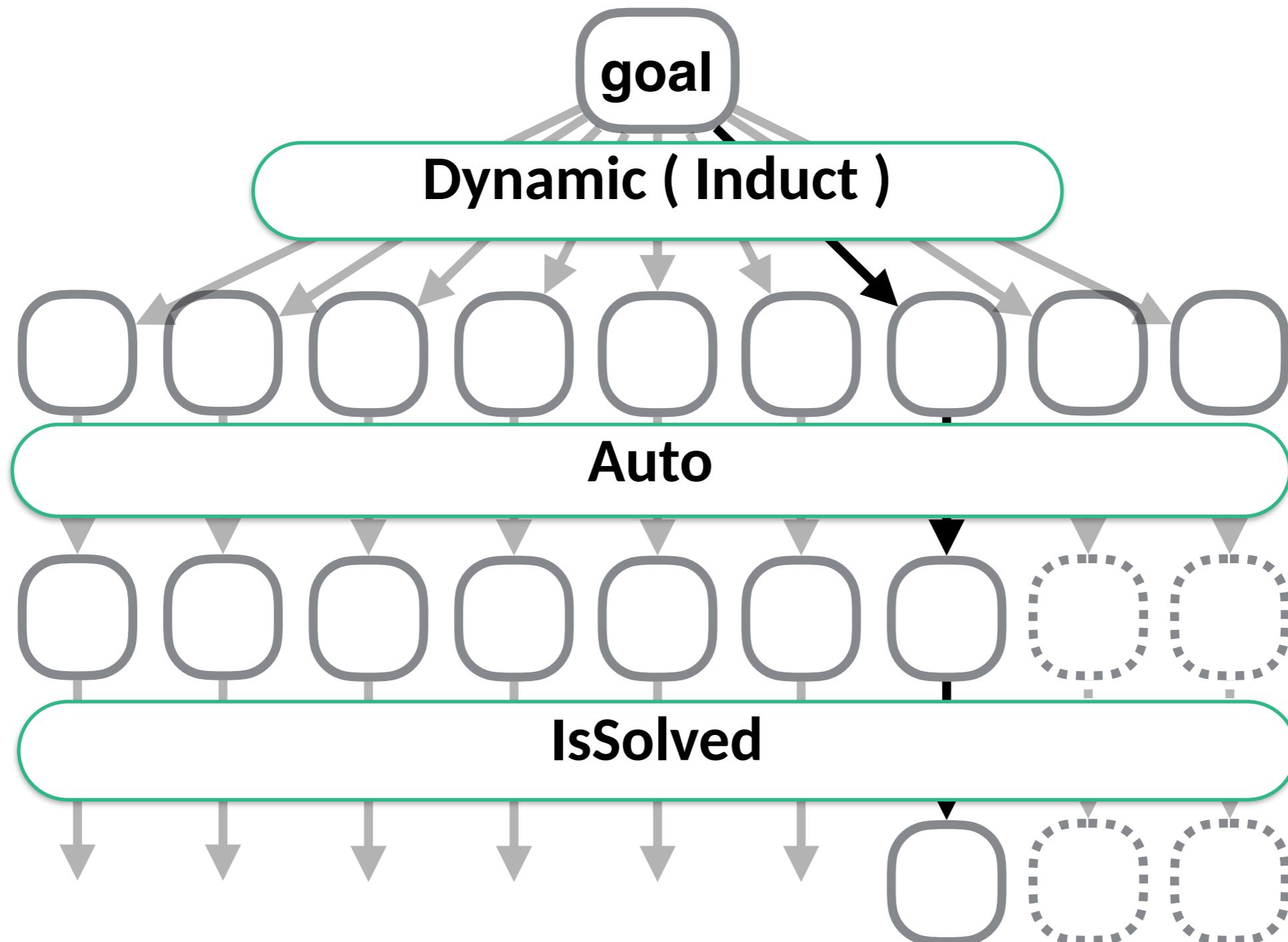
Previously at Master Seminar...

<https://twitter.com/YutakangE>

Previously at Master Seminar... I talked about PSL.

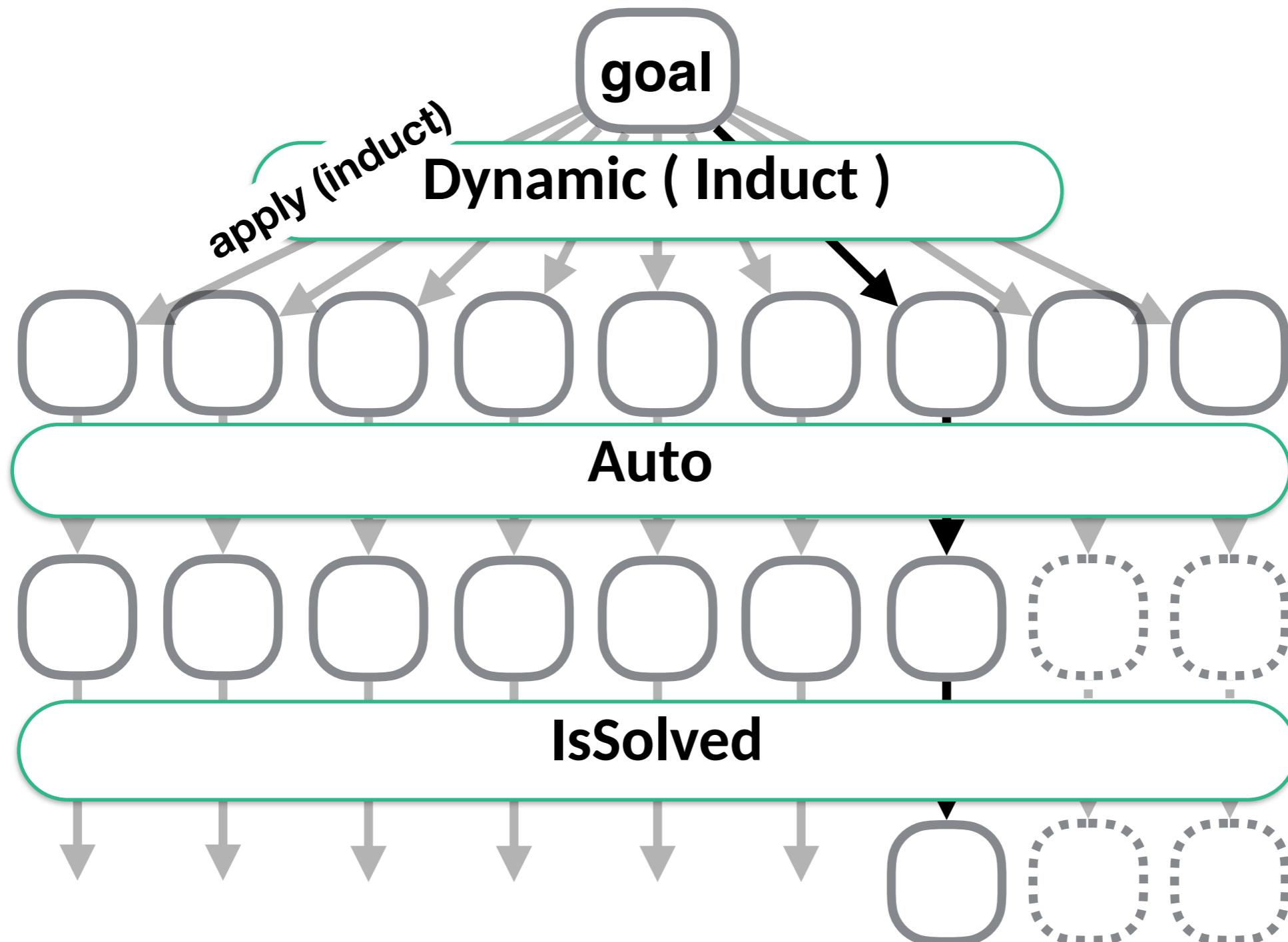
Previously at Master Seminar... I talked about PSL.

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



Previously at Master Seminar... I talked about PSL.

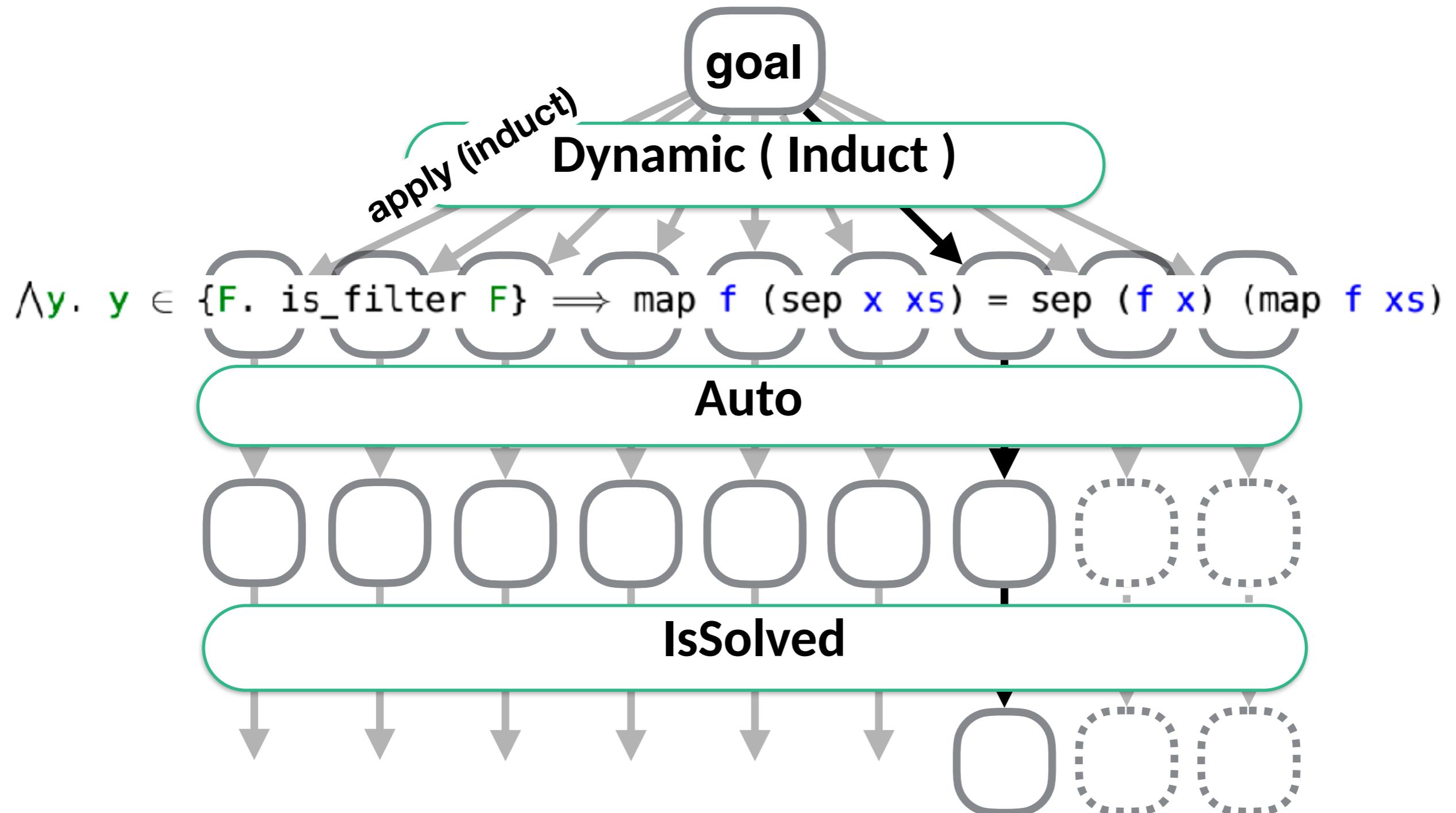
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



<https://twitter.com/YutakangE>

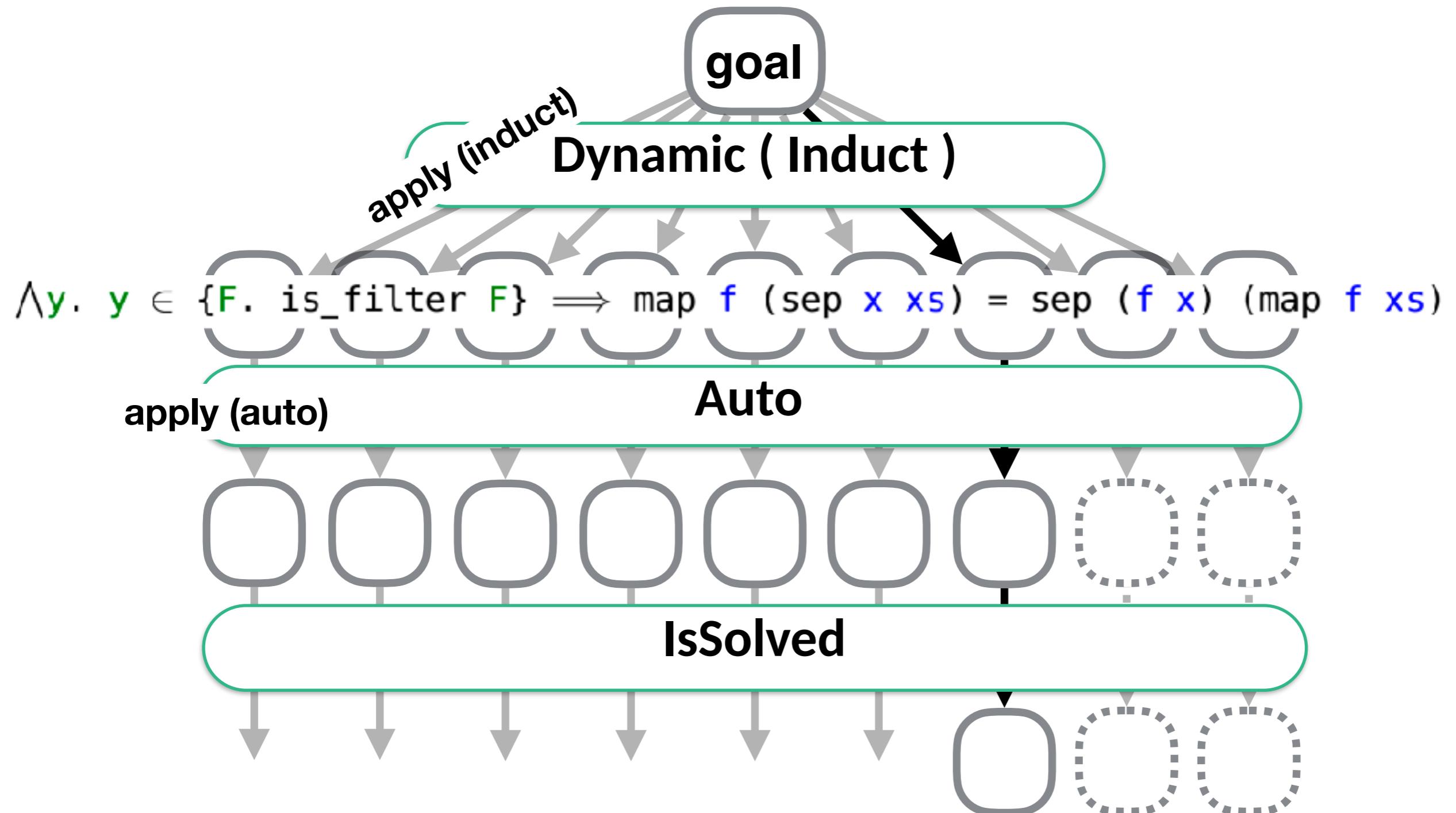
Previously at Master Seminar... I talked about PSL.

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



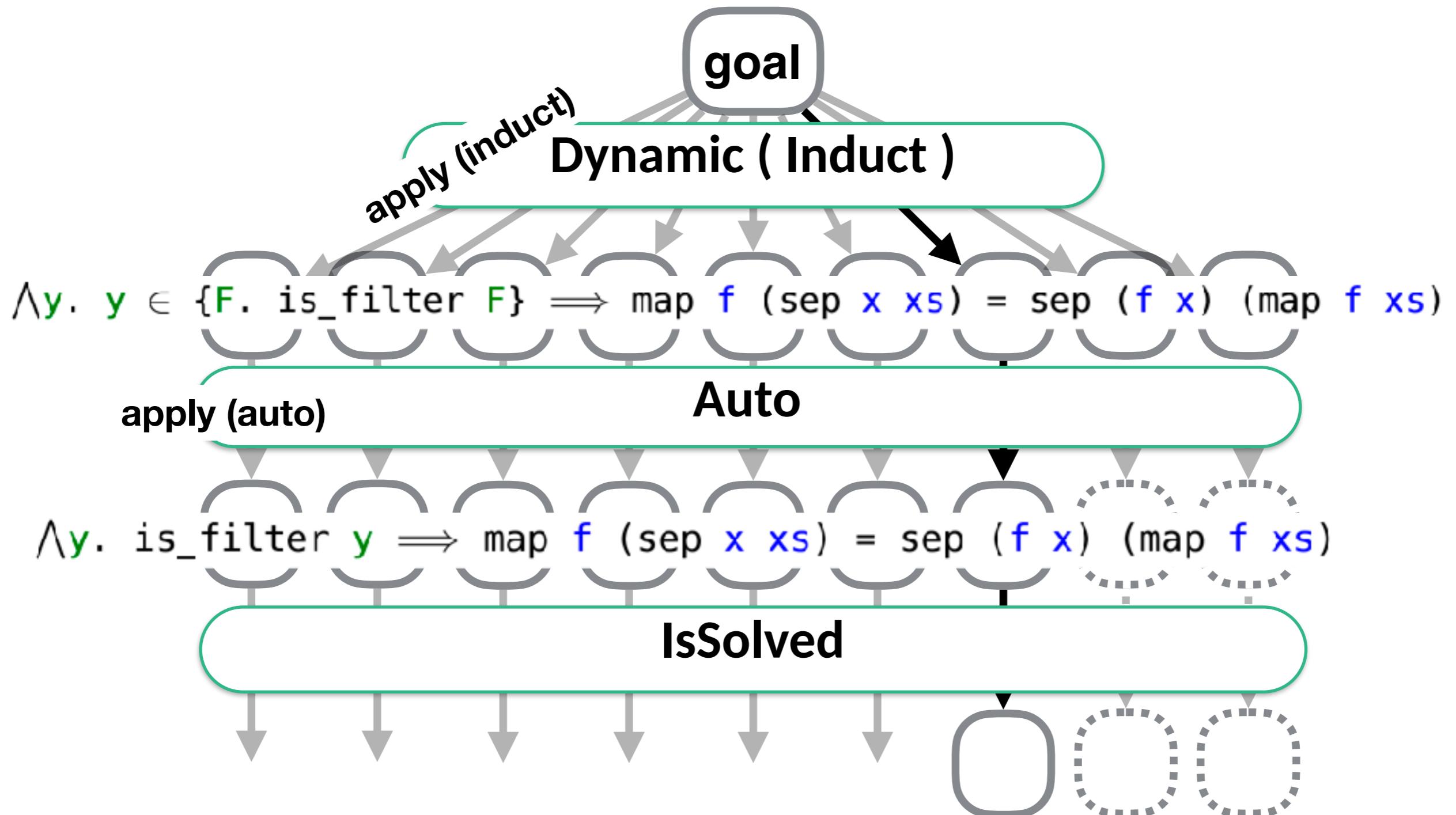
Previously at Master Seminar... I talked about PSL.

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



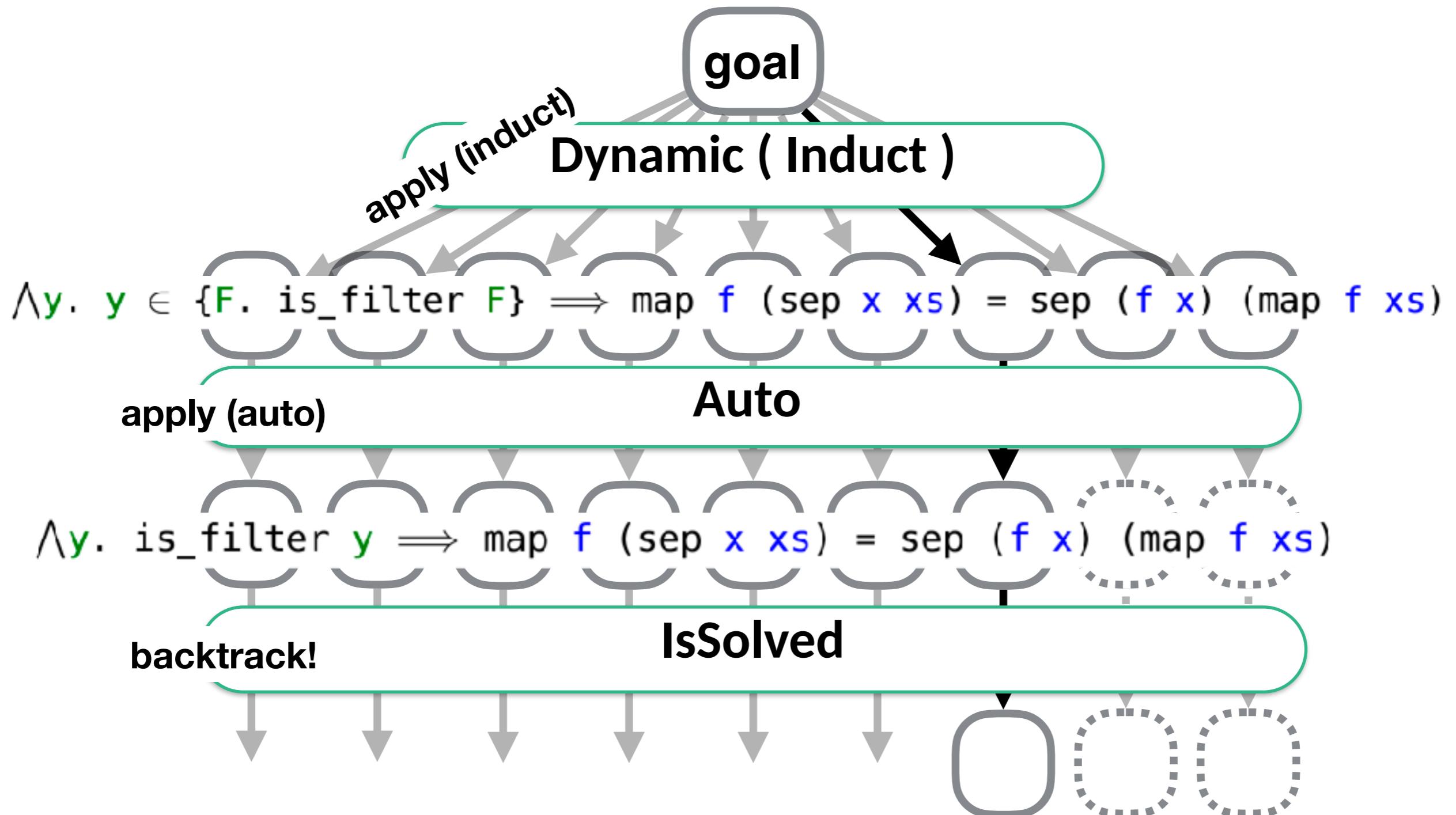
Previously at Master Seminar... I talked about PSL.

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



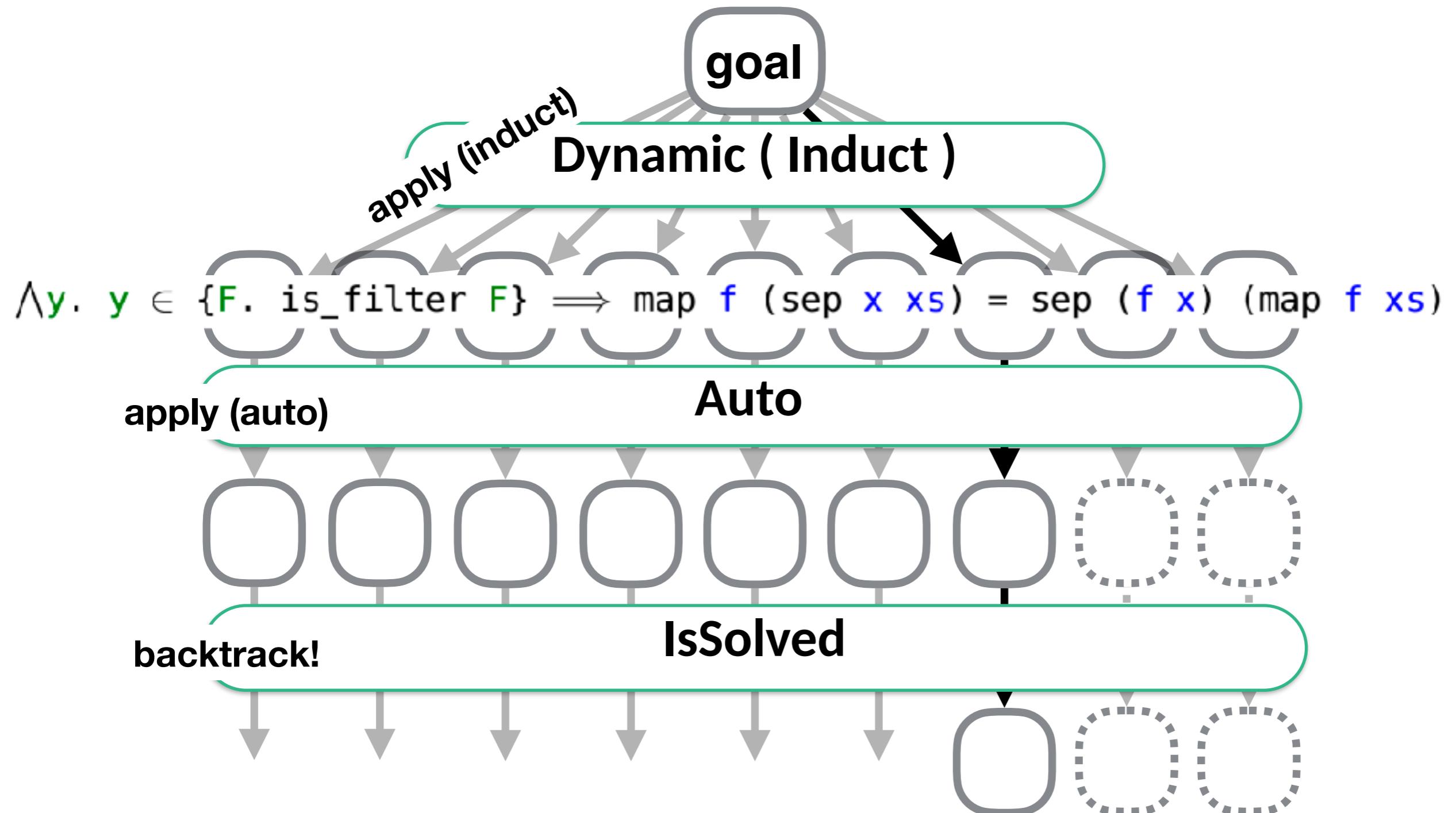
Previously at Master Seminar... I talked about PSL.

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



Previously at Master Seminar... I talked about PSL.

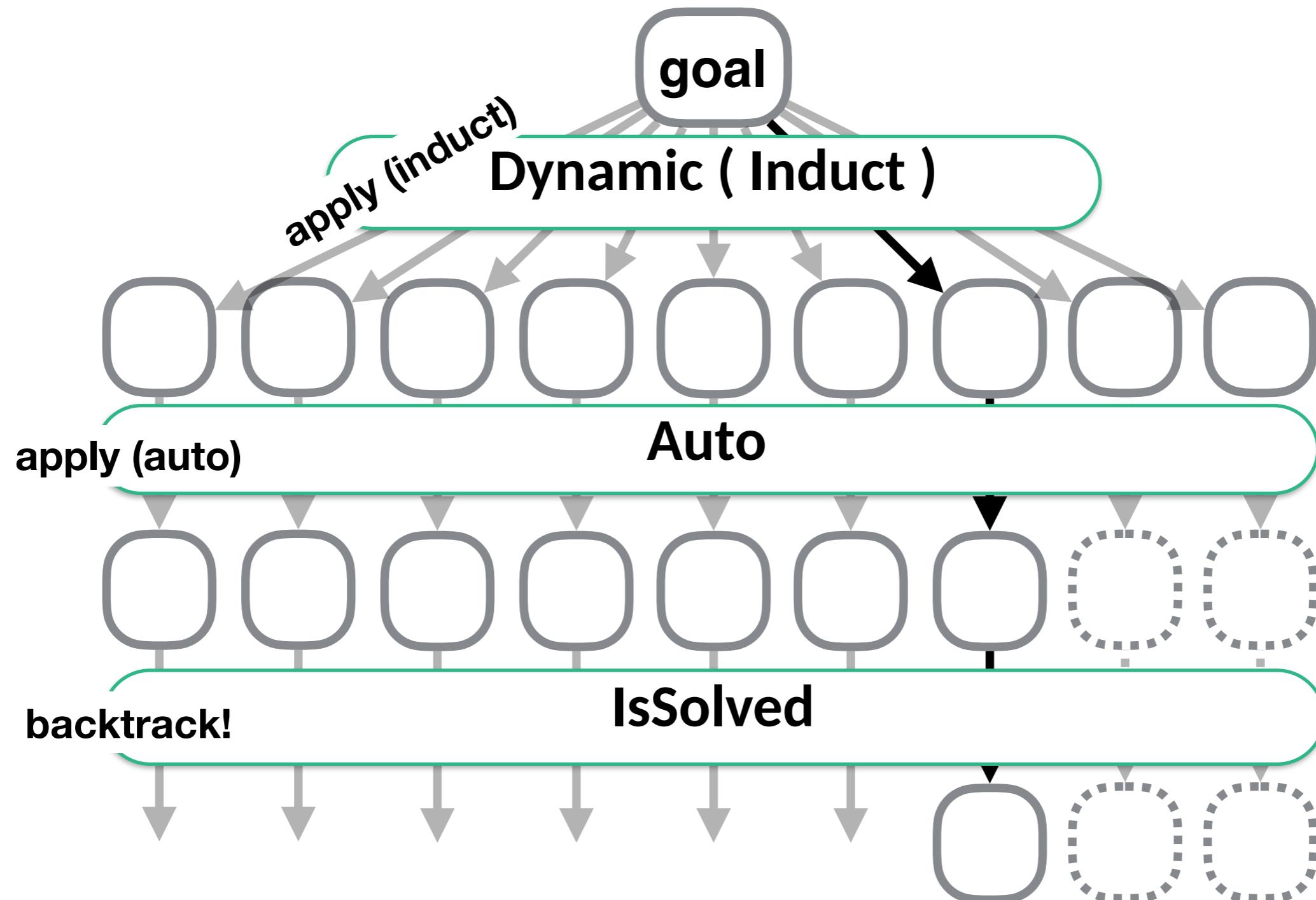
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



<https://twitter.com/YutakangE>

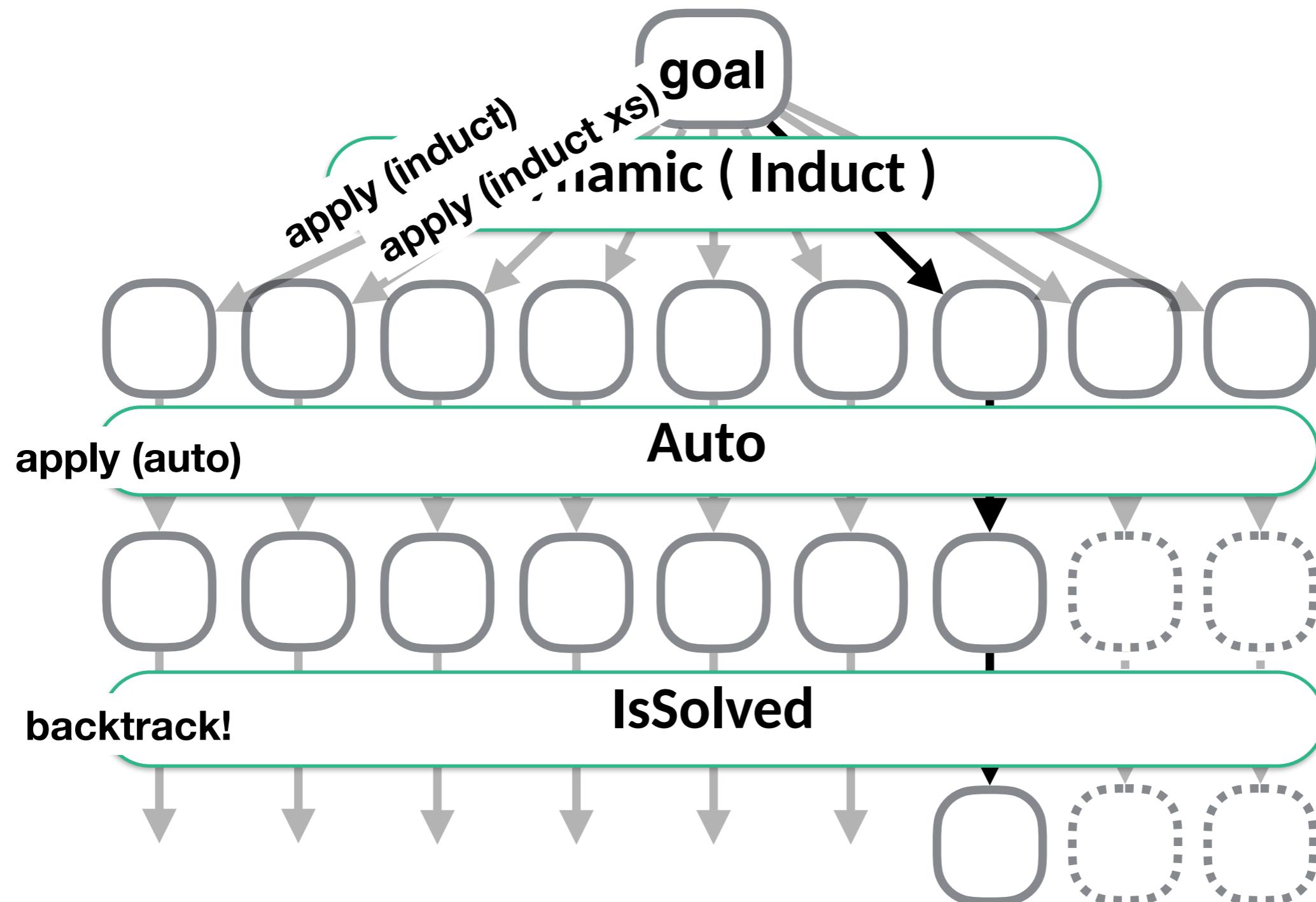
Previously at Master Seminar... I talked about PSL.

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



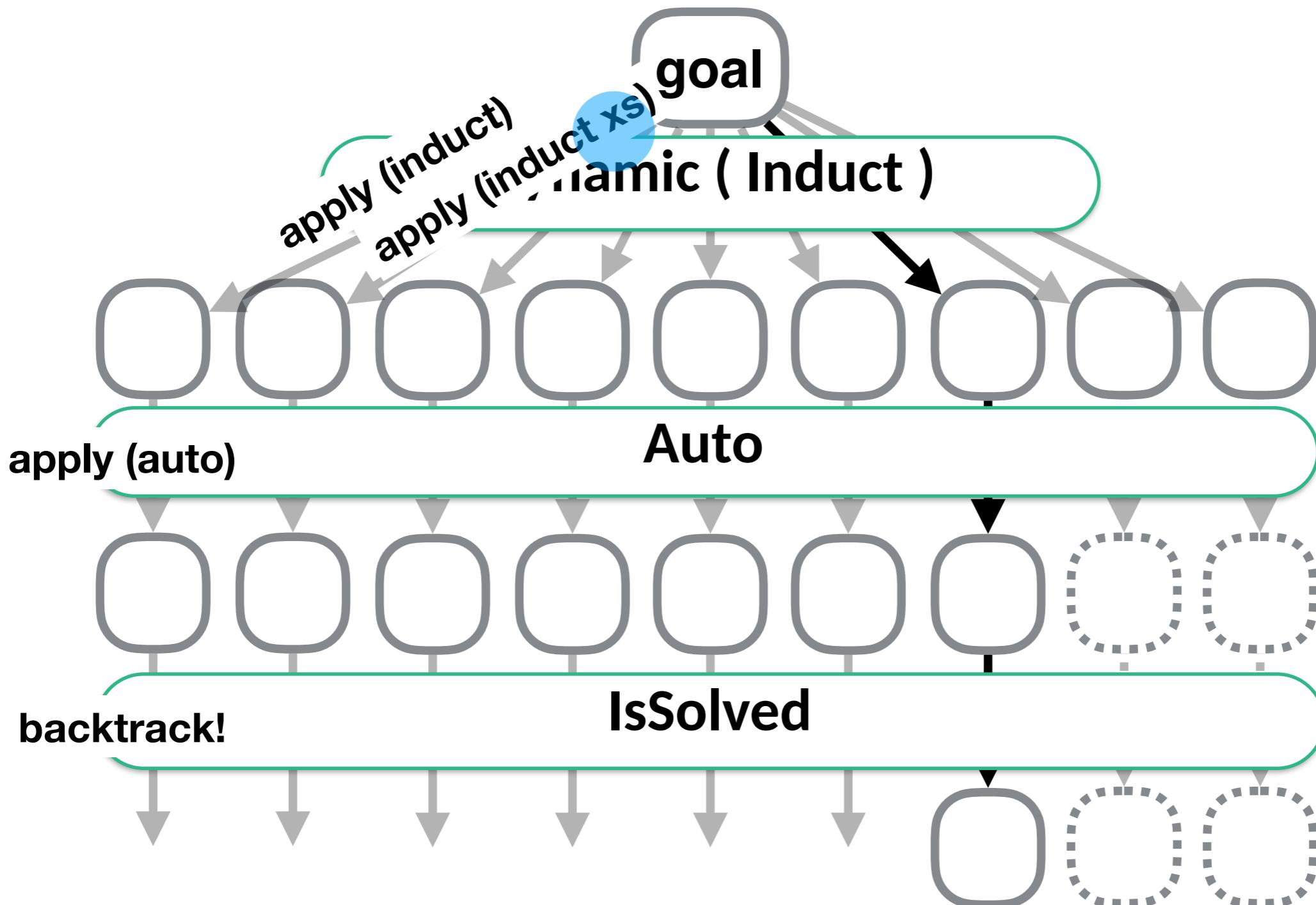
Previously at Master Seminar... I talked about PSL.

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



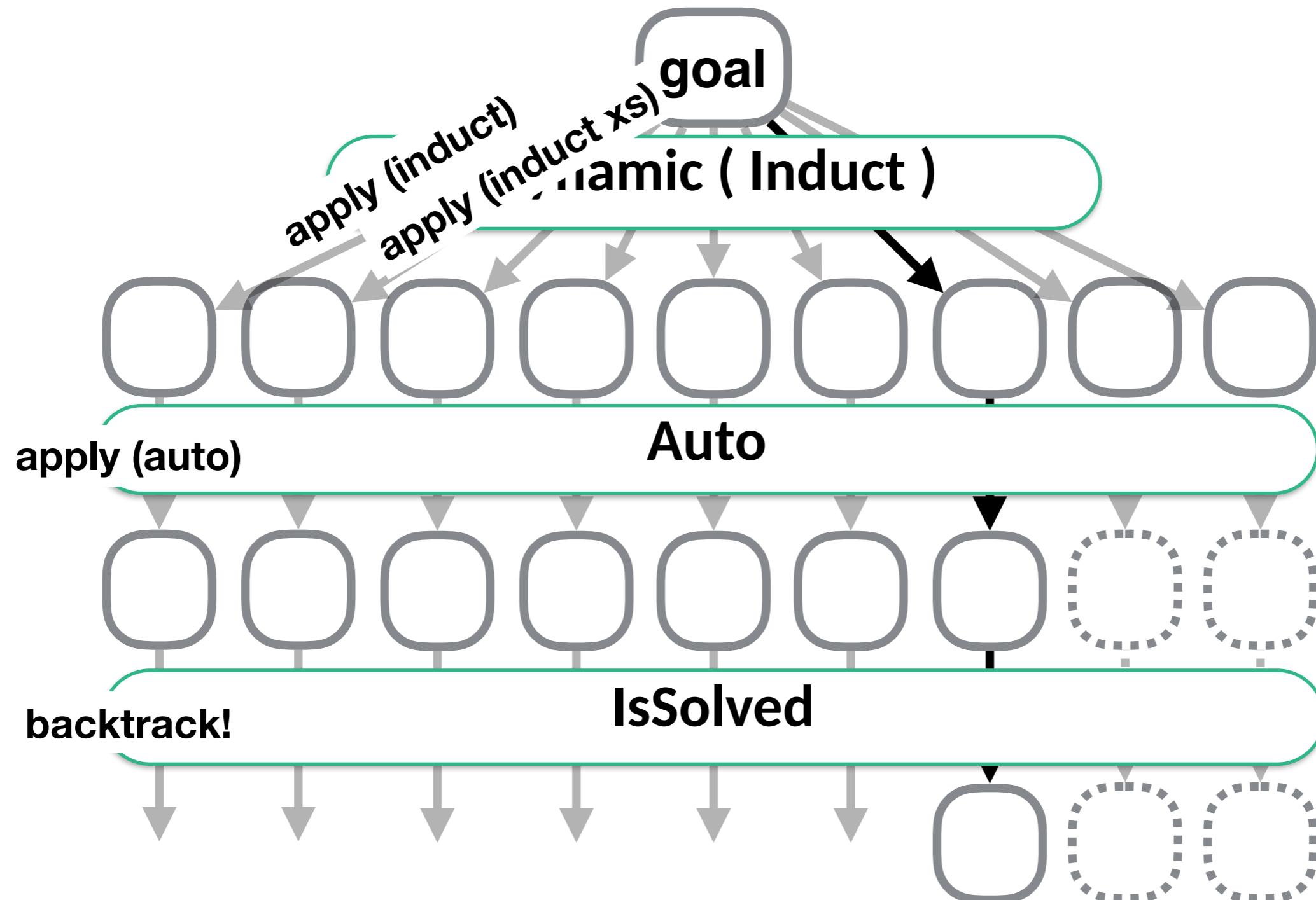
Previously at Master Seminar... I talked about PSL.

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



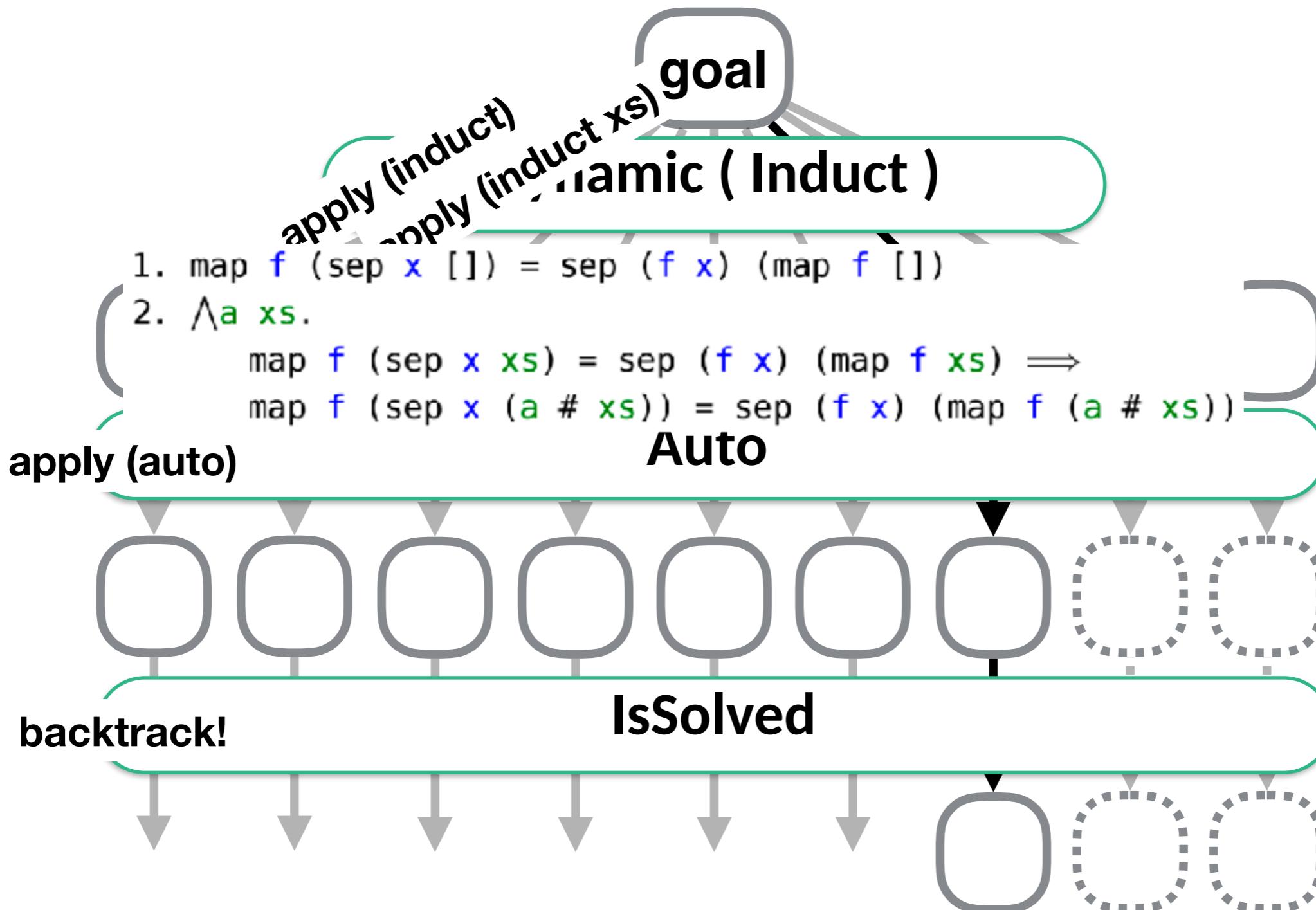
Previously at Master Seminar... I talked about PSL.

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



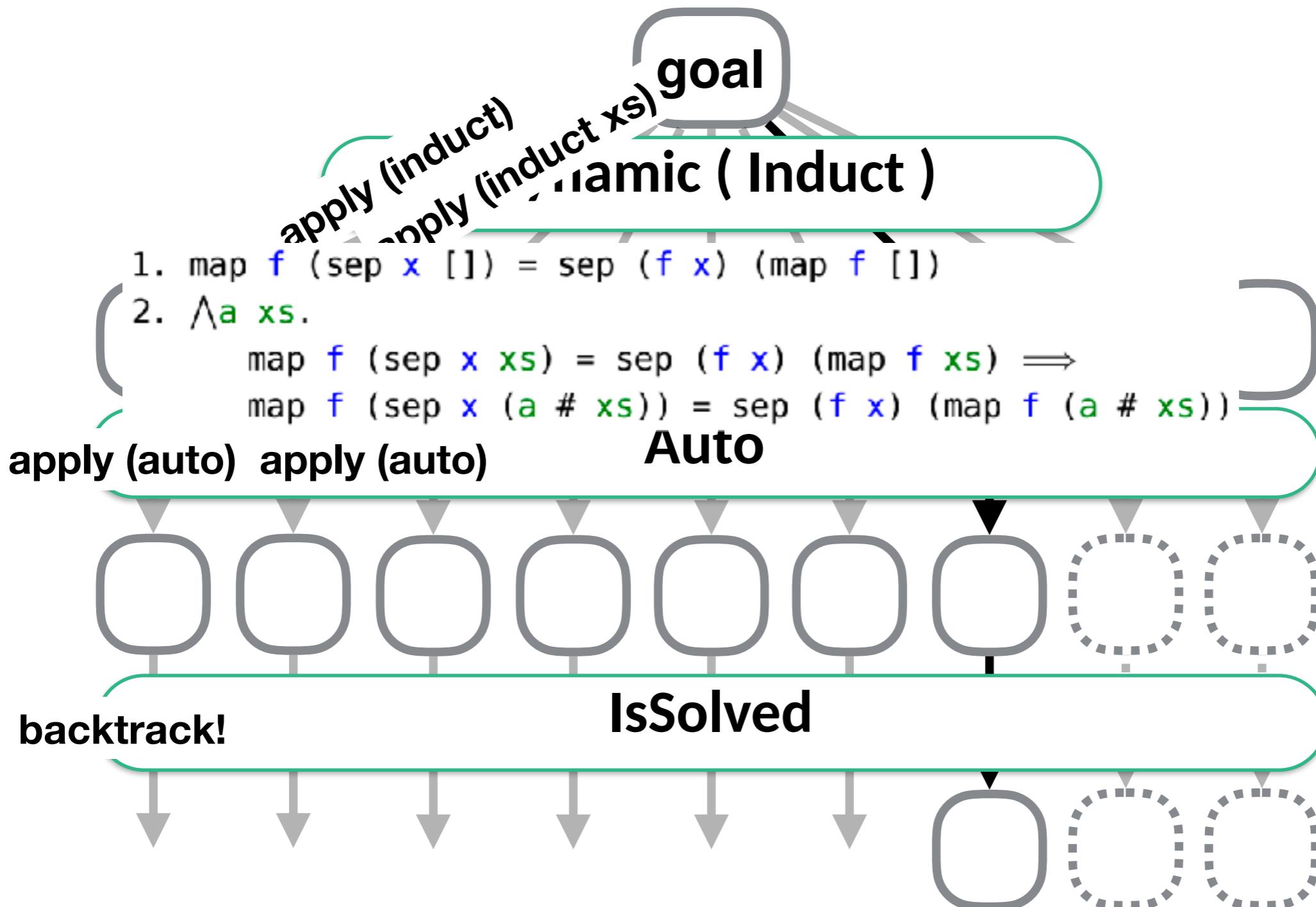
Previously at Master Seminar... I talked about PSL.

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



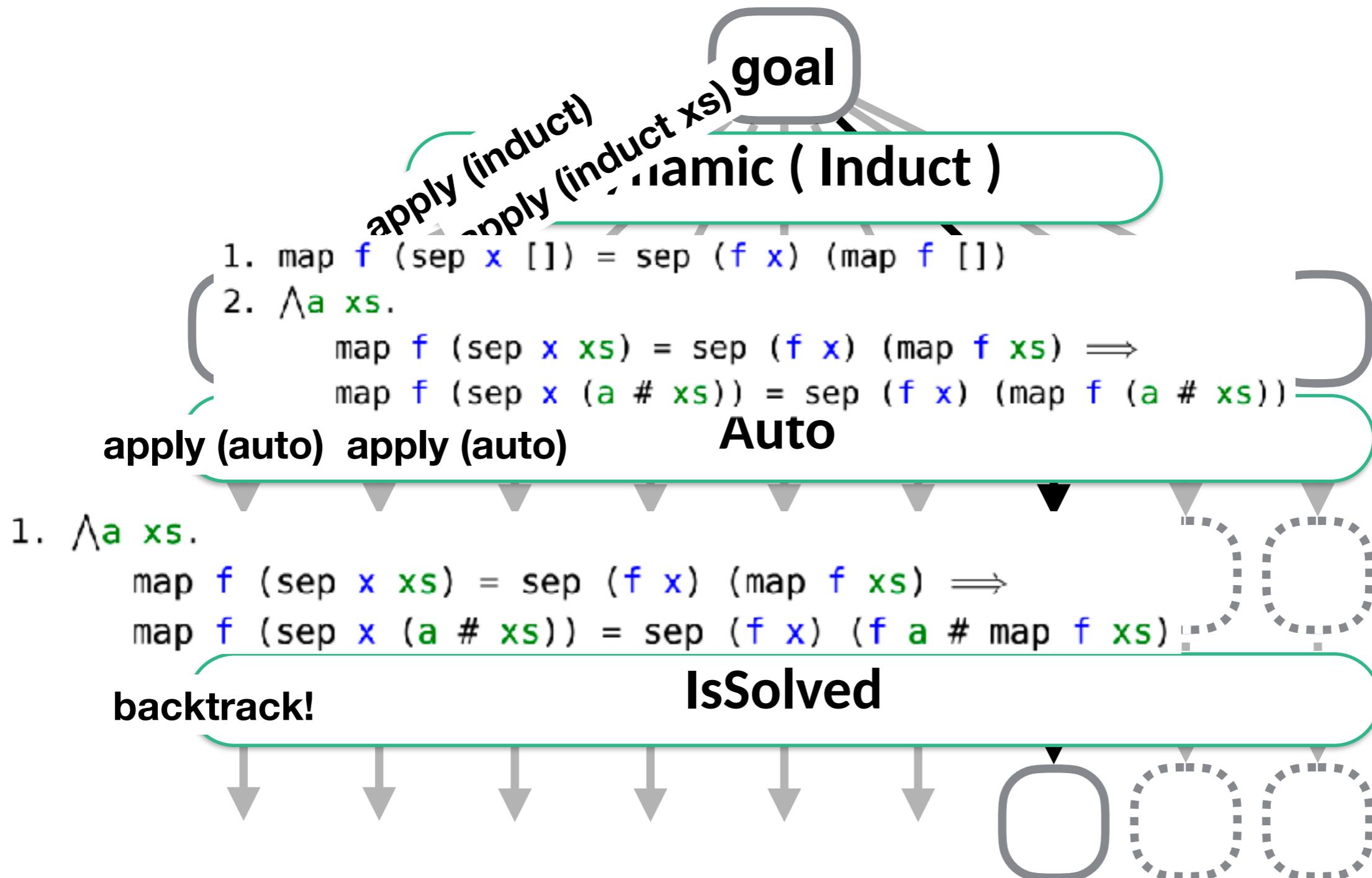
Previously at Master Seminar... I talked about PSL.

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



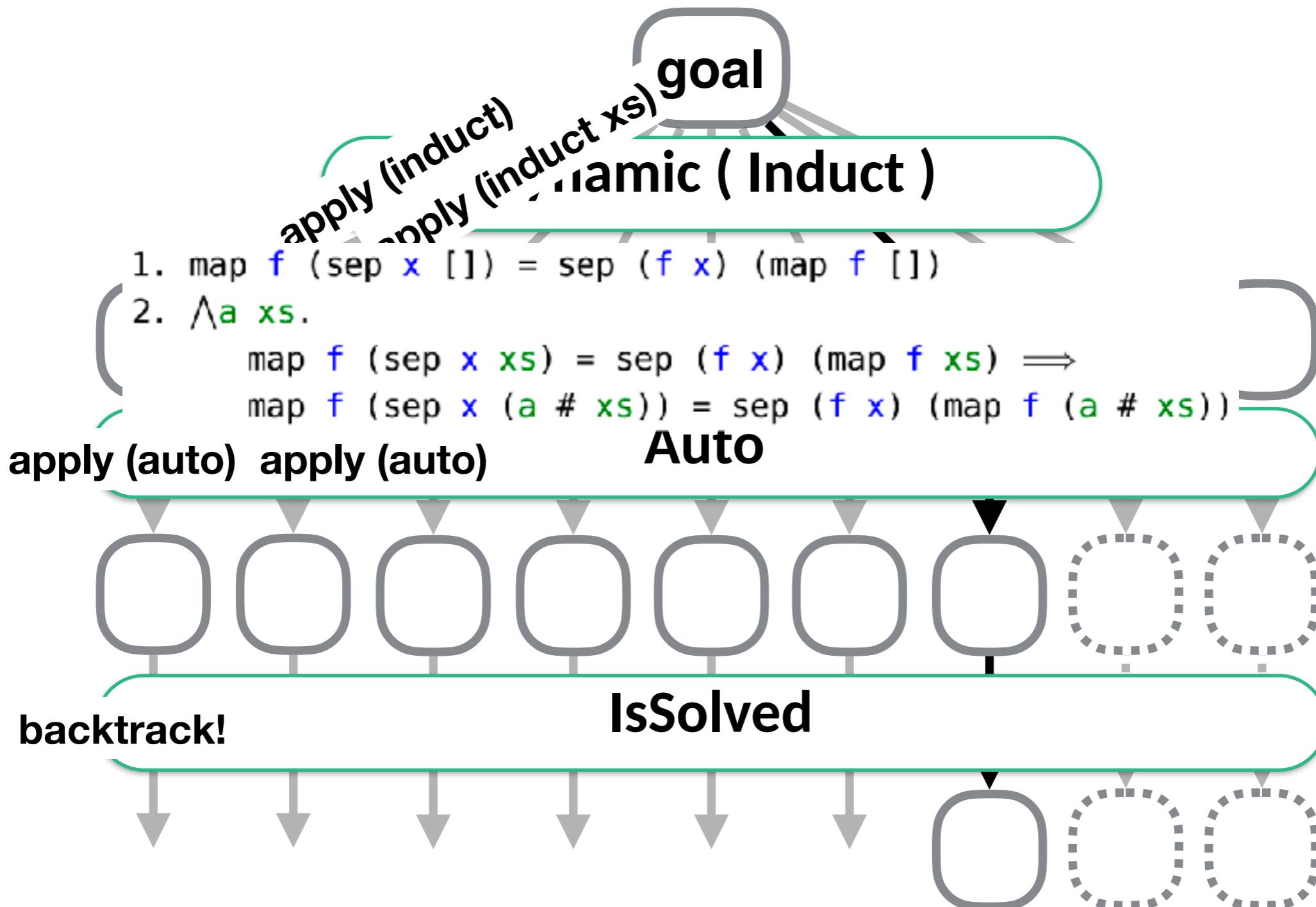
Previously at Master Seminar... I talked about PSL.

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



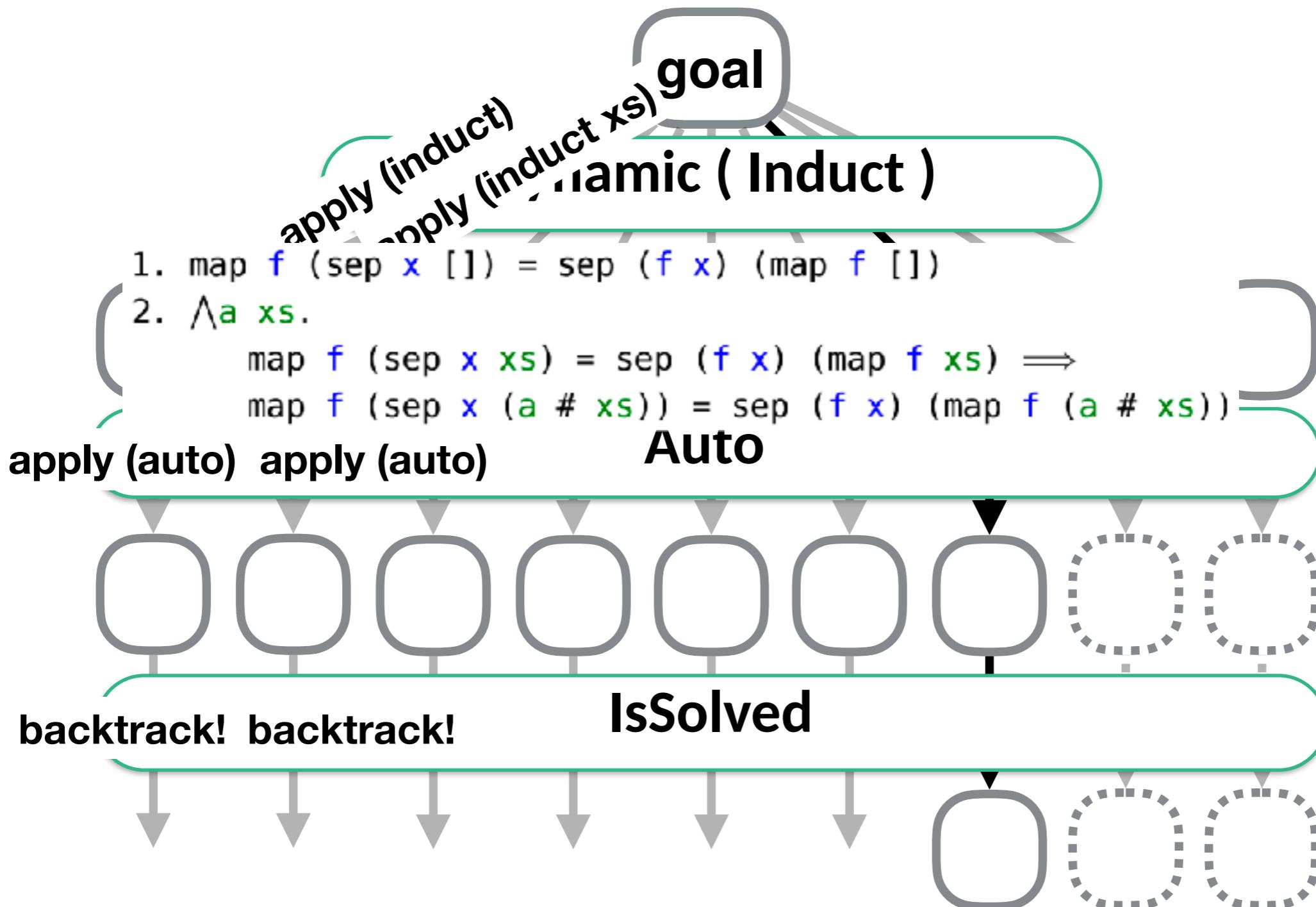
Previously at Master Seminar... I talked about PSL.

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



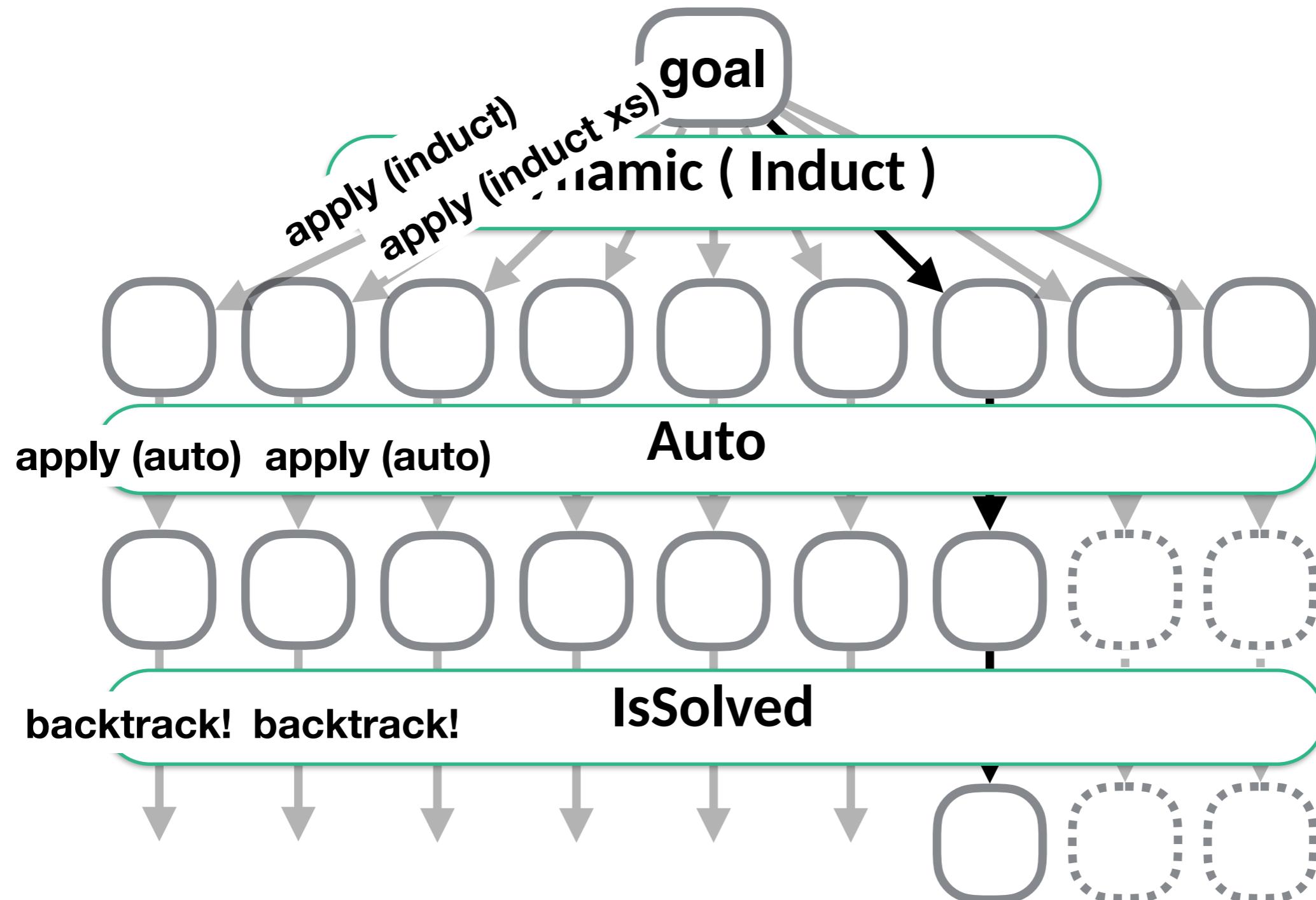
Previously at Master Seminar... I talked about PSL.

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



Previously at Master Seminar... I talked about PSL.

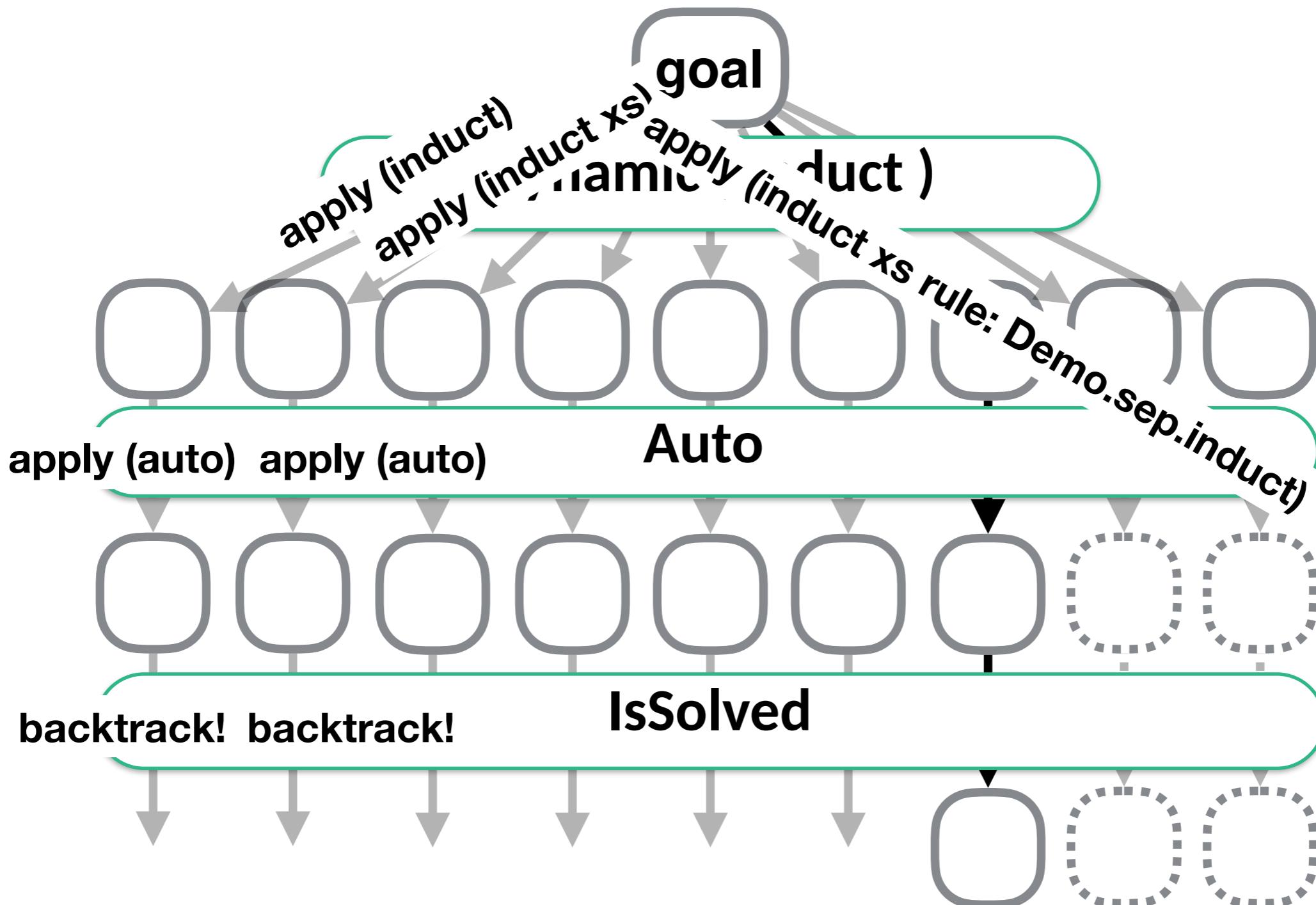
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



<https://twitter.com/YutakangE>

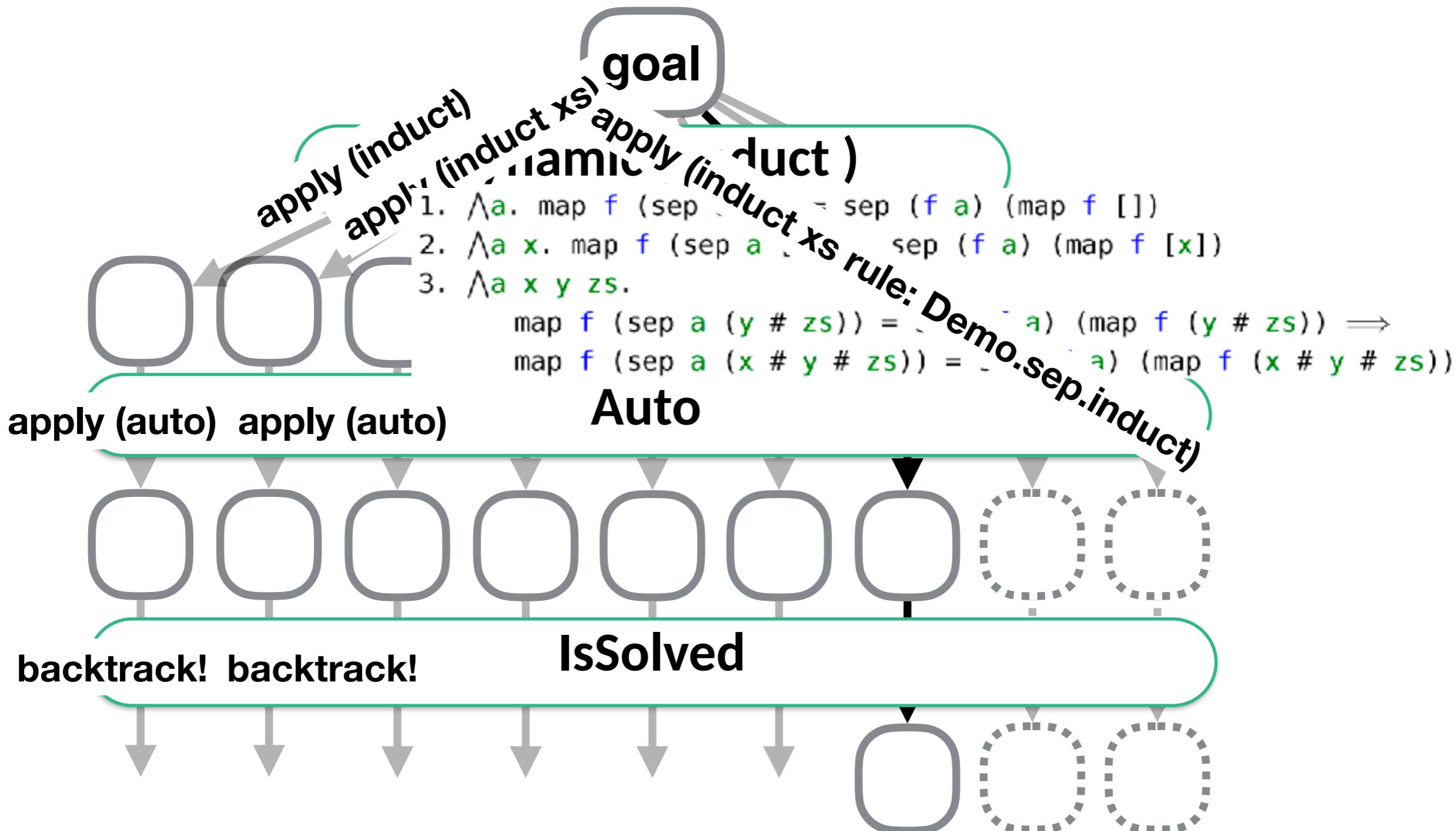
Previously at Master Seminar... I talked about PSL.

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



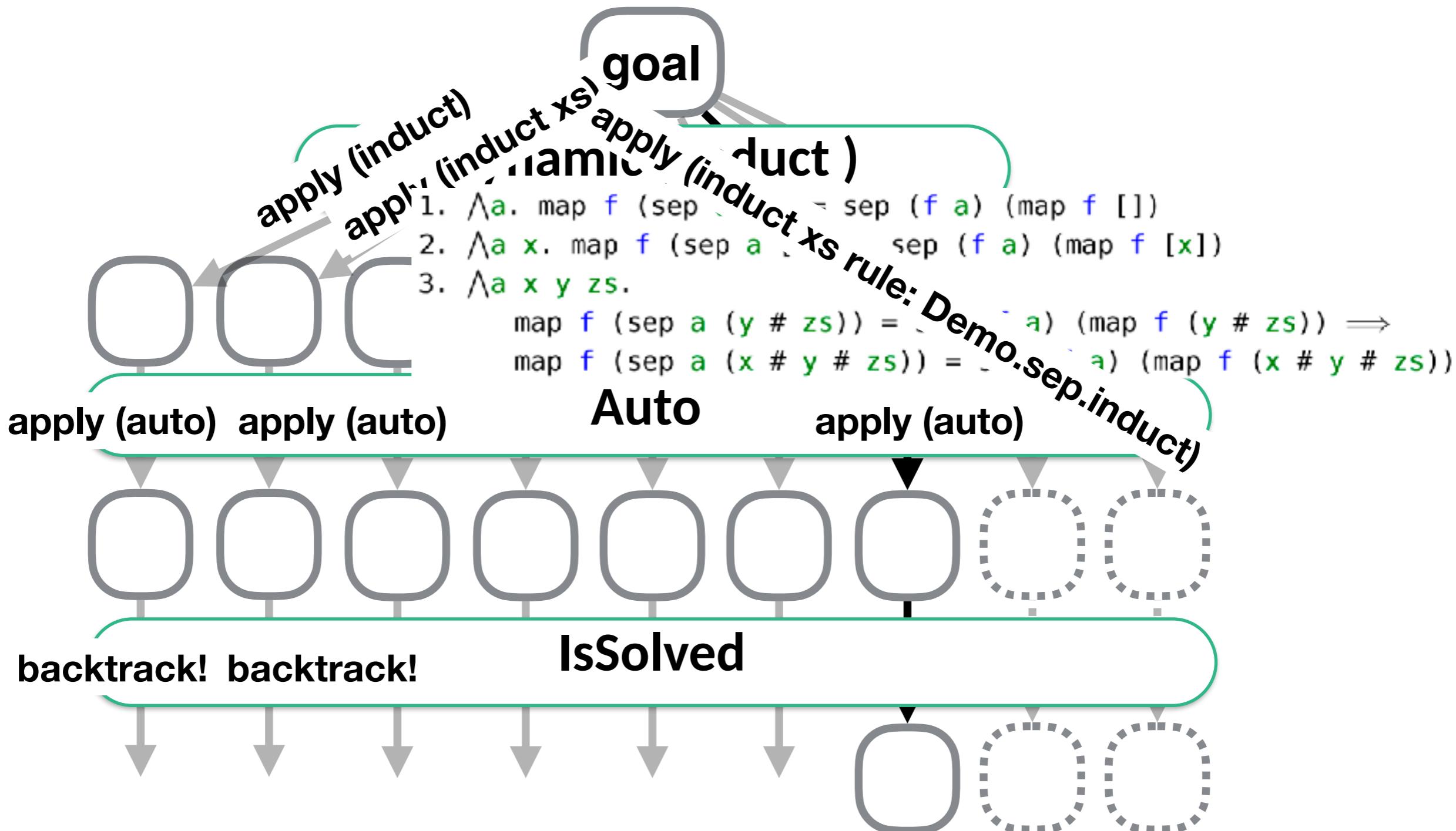
Previously at Master Seminar... I talked about PSL.

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



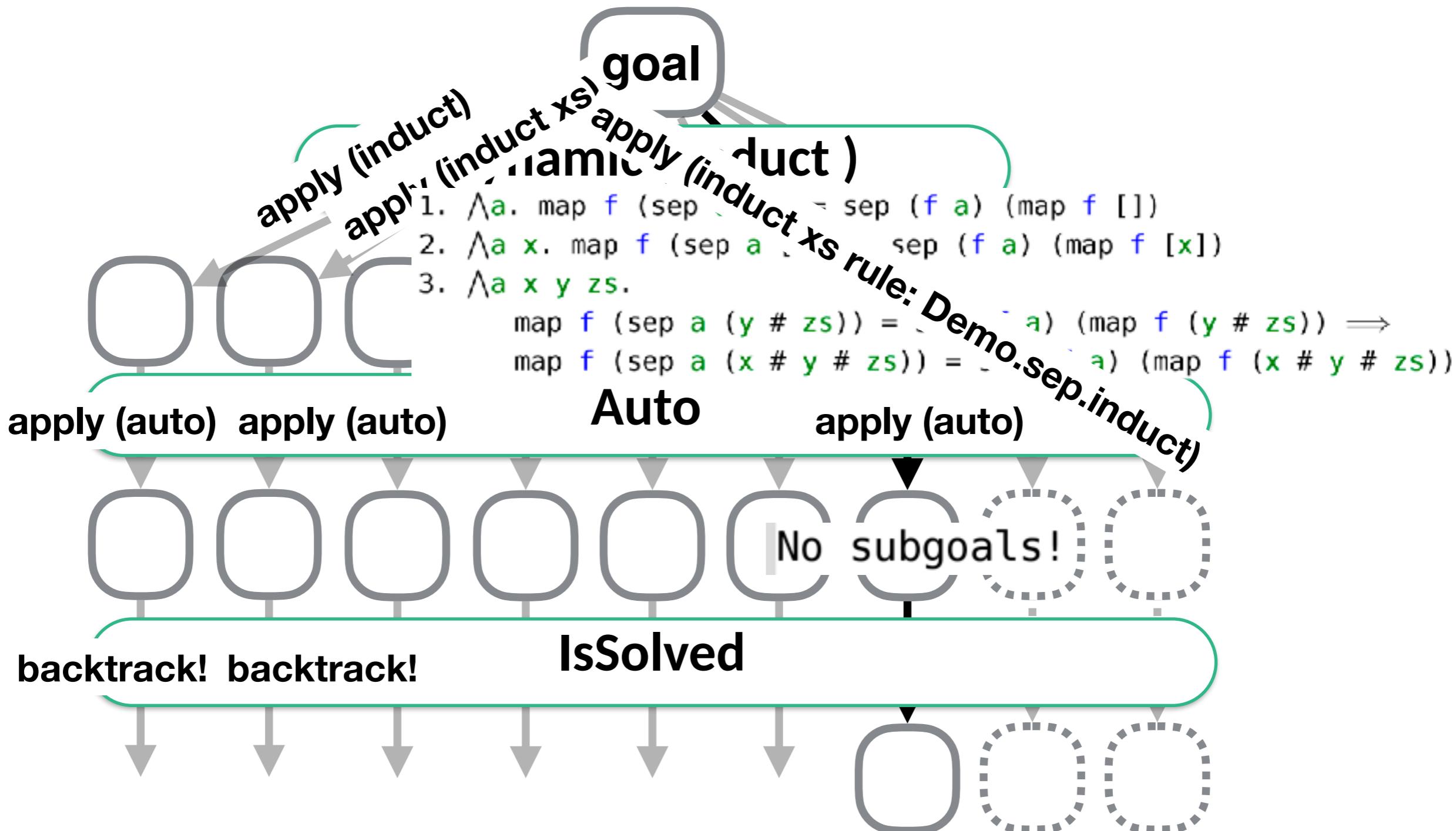
Previously at Master Seminar... I talked about PSL.

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



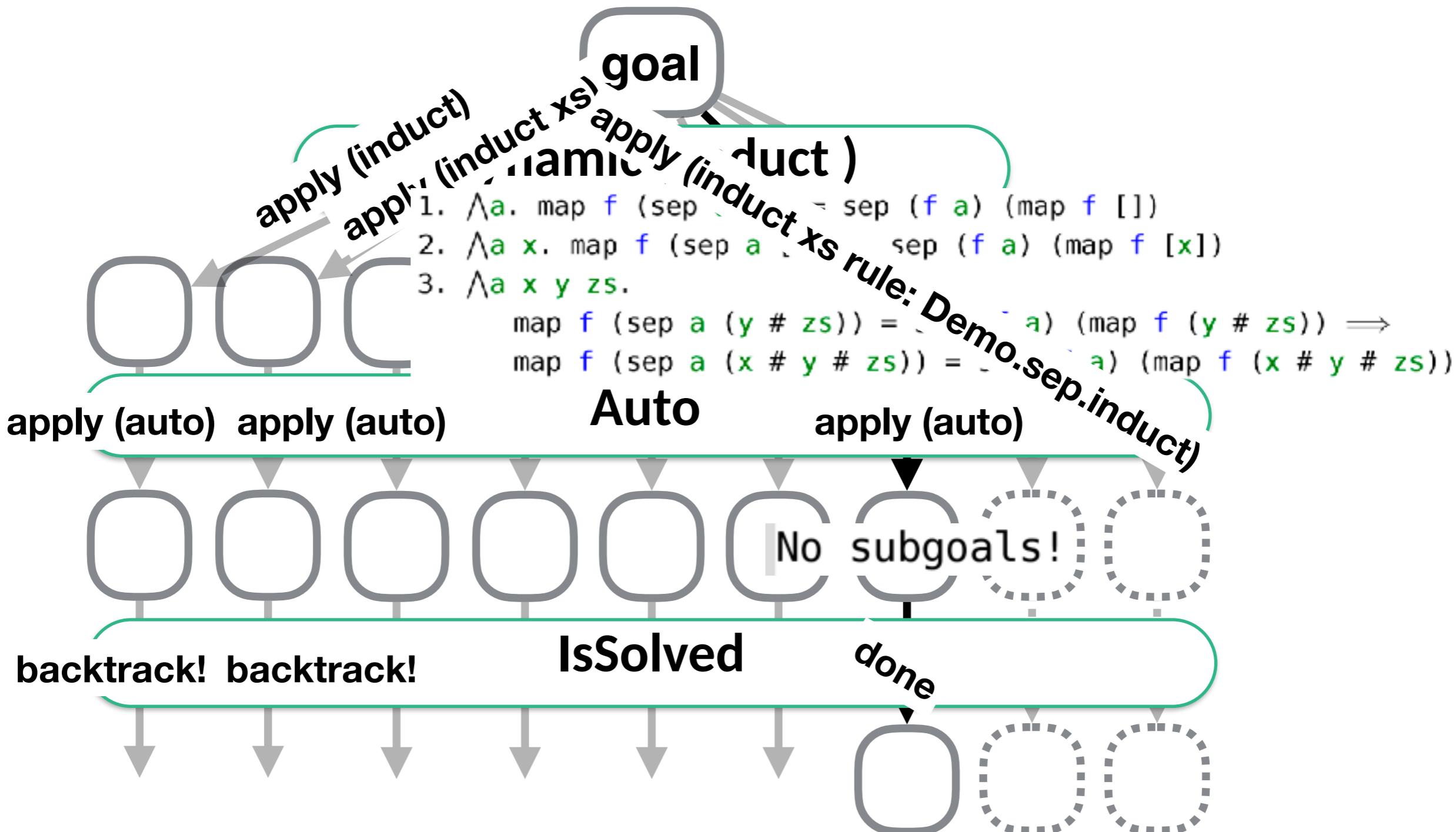
Previously at Master Seminar... I talked about PSL.

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



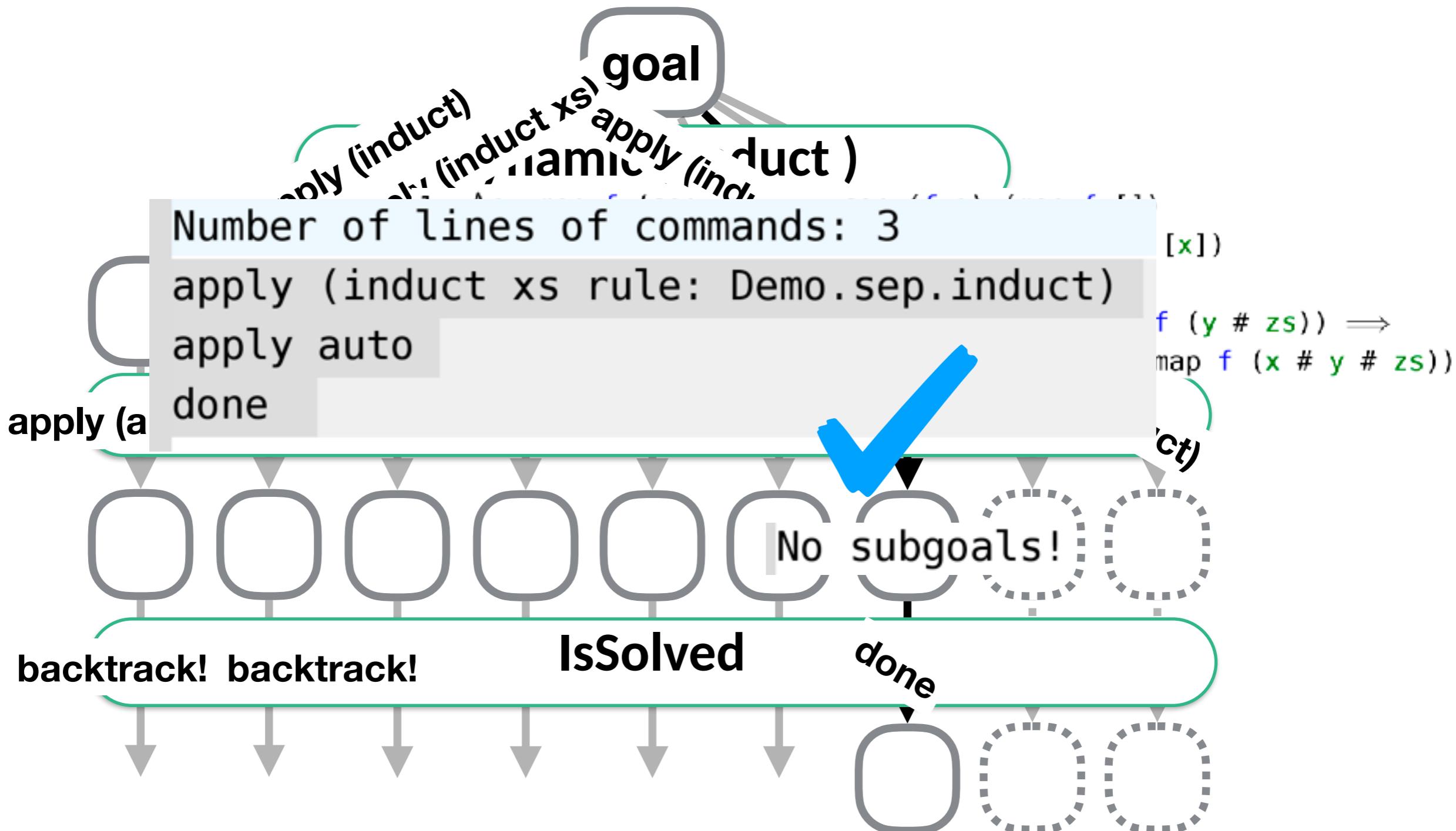
Previously at Master Seminar... I talked about PSL.

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



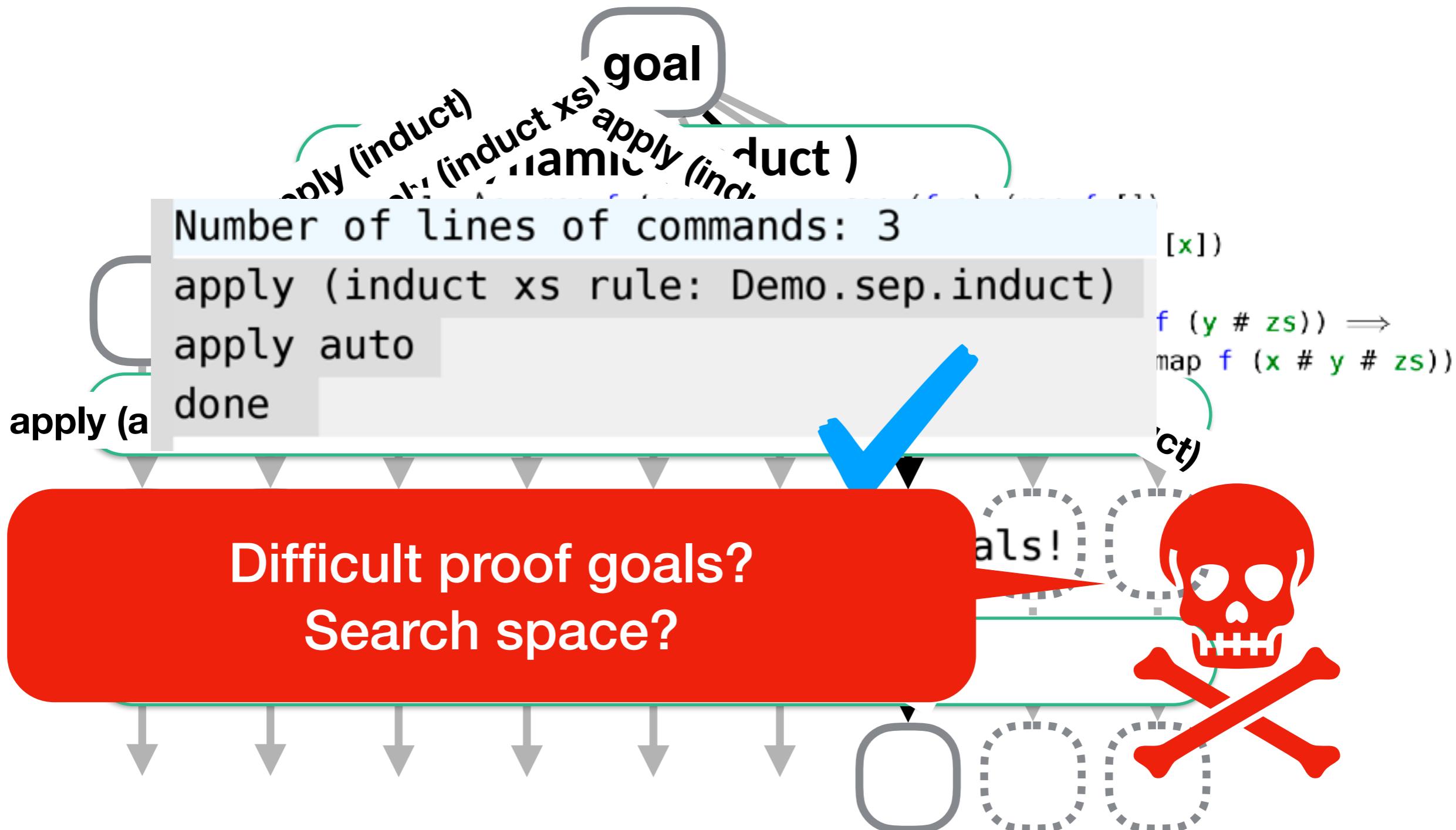
Previously at Master Seminar... I talked about PSL.

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



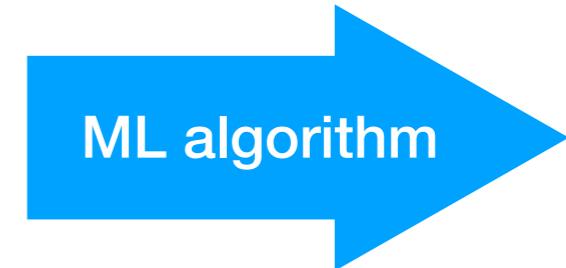
Previously at Master Seminar... I talked about PSL.

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



<https://twitter.com/YutakangE>

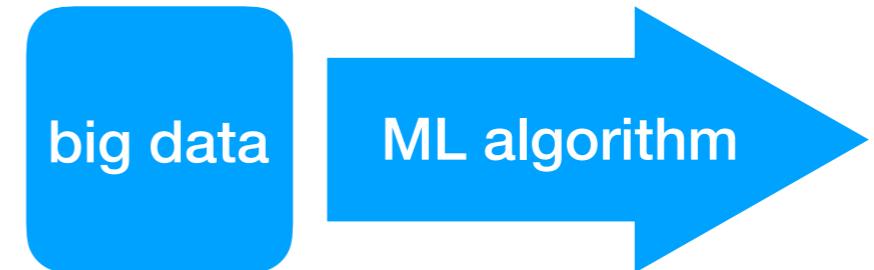
Introduction to Machine Learning in 10 seconds



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

<https://twitter.com/YutakangE>

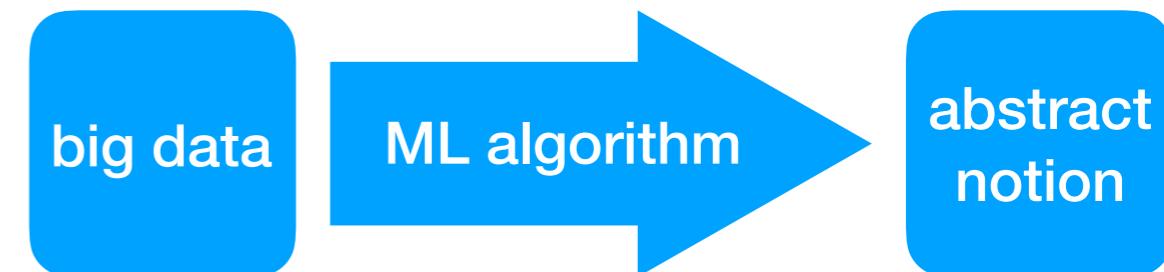
Introduction to Machine Learning in 10 seconds



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

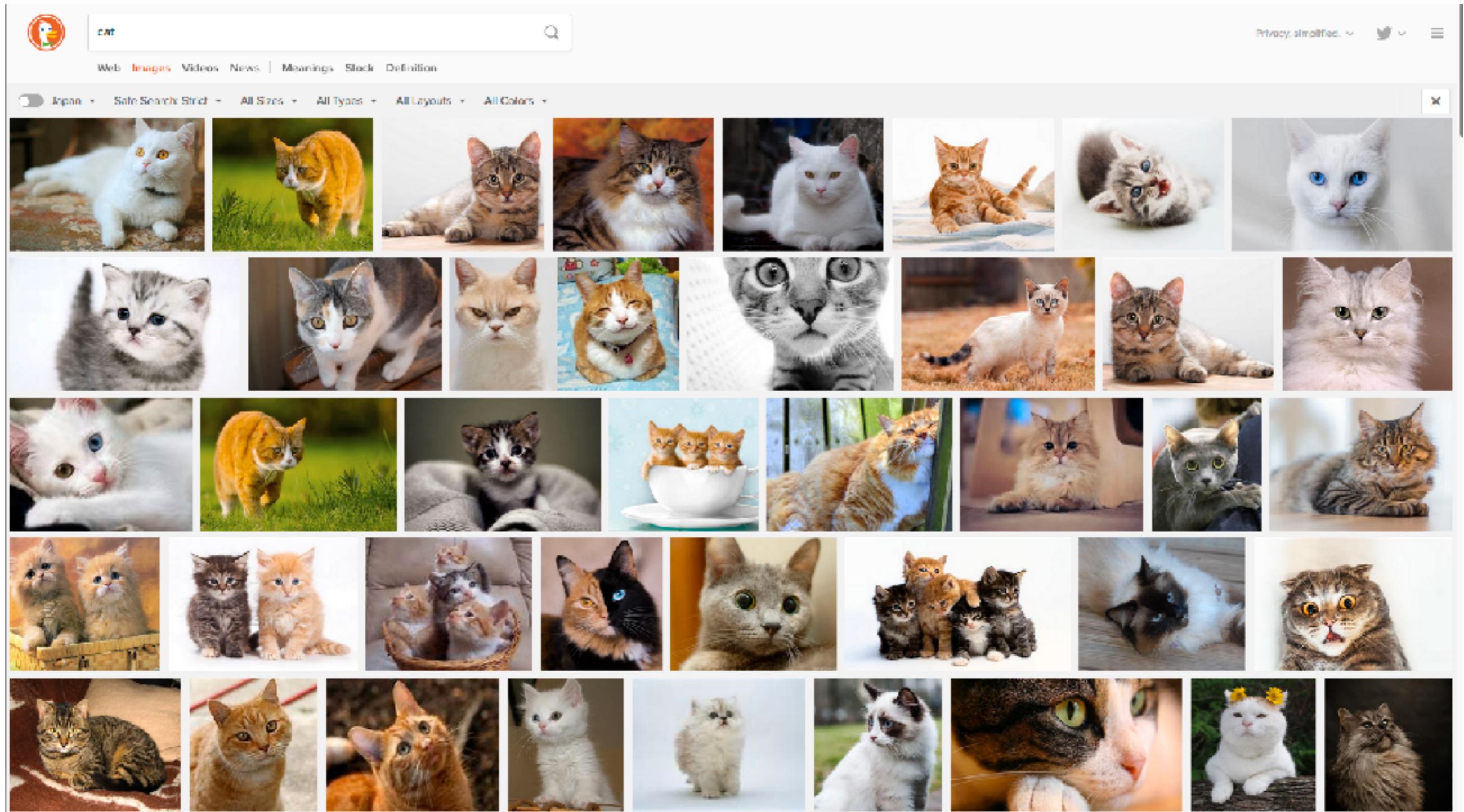
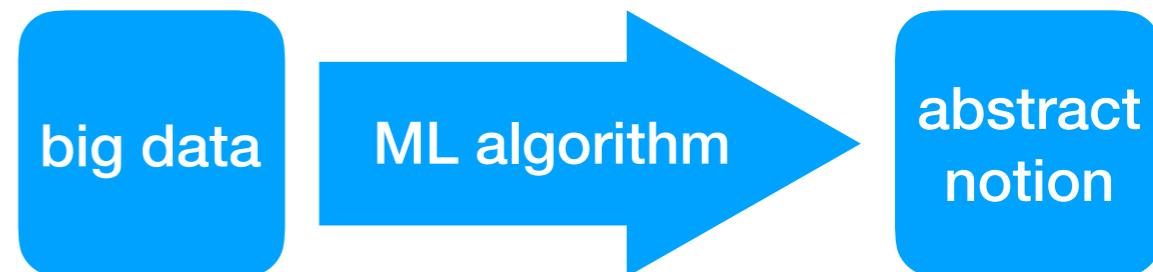
<https://twitter.com/YutakangE>

Introduction to Machine Learning in 10 seconds



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

Introduction to Machine Learning in 10 seconds

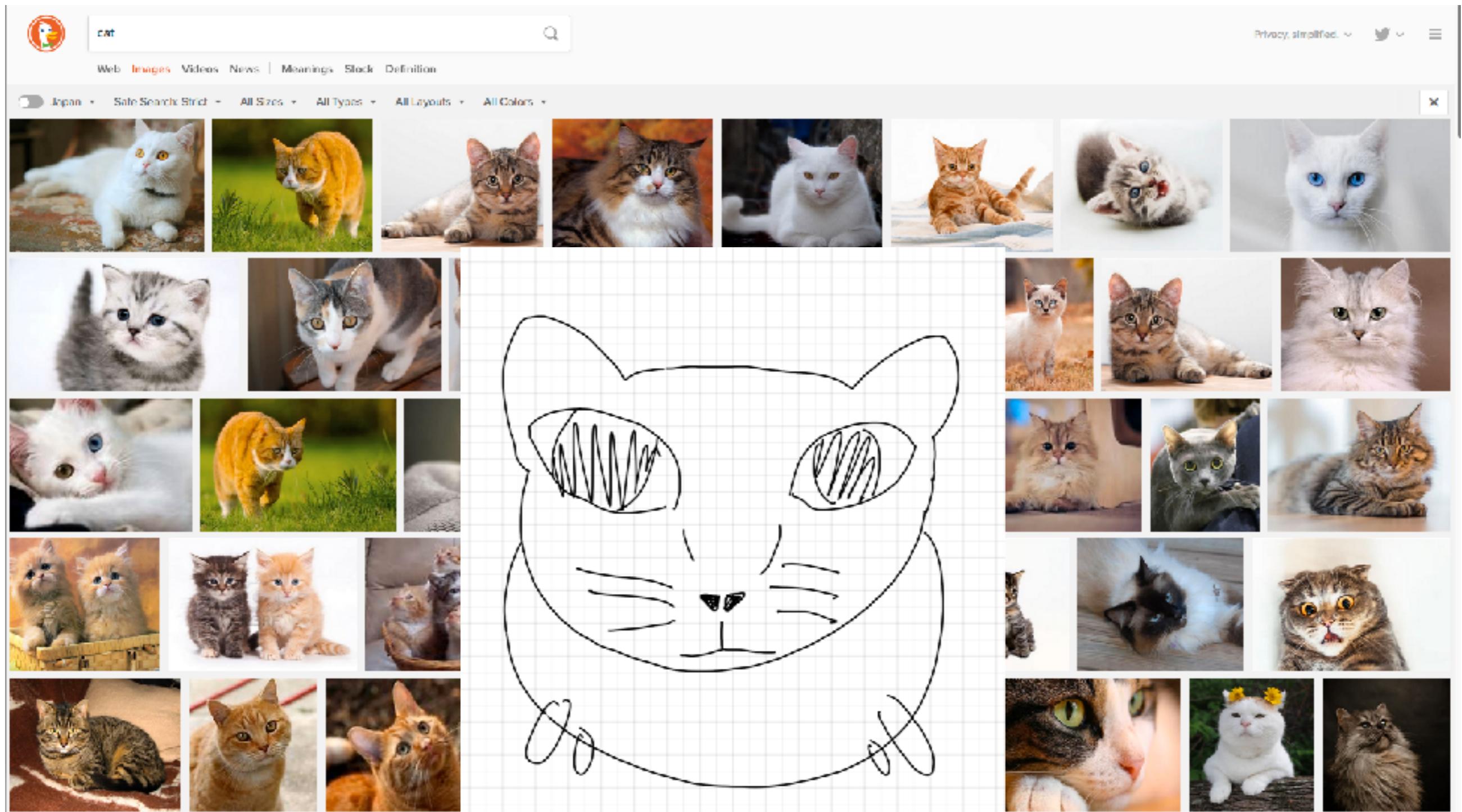


Introduction to Machine Learning in 10 seconds

big data

ML algorithm

abstract notion



ML for Inductive Theorem Proving the BAD

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

← one abstract representation

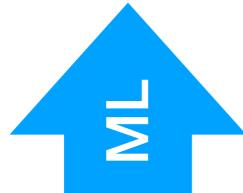
```
lemma "itrev [1,2] [] = rev [1,2] @ []" by auto
lemma "itrev [1,2,3] [] = rev [1,2,3] @ []" by auto
lemma "itrev [''a'', ''b''] [] = rev [''a'', ''b''] @ []" by auto
lemma "itrev [x,y,z] [] = rev [x,y,z] @ []" by auto
```

← many concrete cases

ML for Inductive Theorem Proving the BAD

Lemma "itrev xs ys = rev xs @ ys"

← one abstract representation



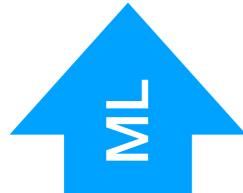
```
lemma "itrev [] = rev [] @ []" by auto
lemma "itrev [1,2] = rev [1,2] @ []" by auto
lemma "itrev [1,2,3] = rev [1,2,3] @ []" by auto
lemma "'a', 'b'" [] = rev ['a', 'b'] @ []" by auto
lemma "[x,y,z]" [] = rev [x,y,z] @ []" by auto
```

← many concrete cases

ML for Inductive Theorem Proving the BAD

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

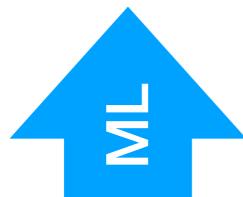
← many concrete cases

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



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

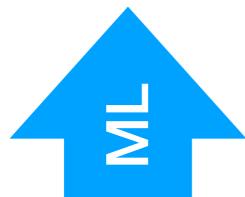
```
lemma "itrev xs ys = rev xs @ ys" by auto  
by(induct xs ys rule:"itrev.induct") auto
```

← one abstract representation

Failed to apply proof method△:

goal (1 subgoal):

1. itrev xs ys = rev xs @ ys



```
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

```
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

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
```

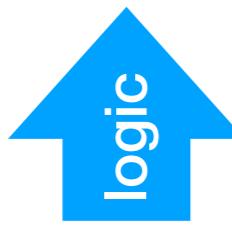
← many concrete cases

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

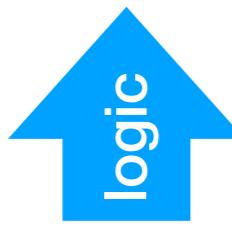
ML for Inductive Theorem Proving the BAD

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

ML for Inductive Theorem Proving the BAD

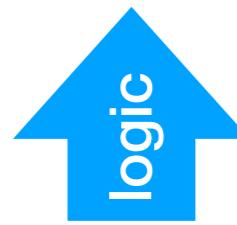
polymorphism

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

ML for Inductive Theorem Proving the BAD

Higher-Order functions

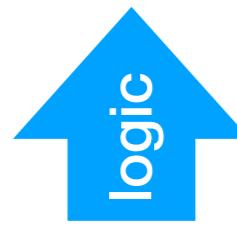
polymorphism

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

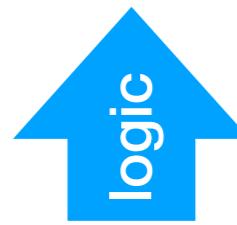
type class

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

ML for Inductive Theorem Proving the BAD

Higher-Order functions

polymorphism

type class

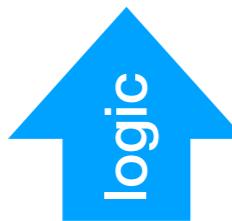
universal quantifier

lambda abstraction

concise formula that can cover
many concrete cases

```
lemma "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

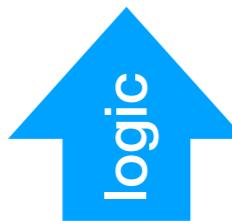
lambda abstraction

concise formula that can cover
many concrete cases

different proof for general case

```
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

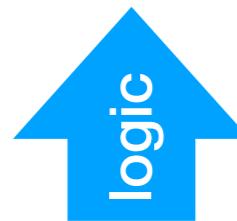
concise formula that can cover
many concrete cases

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

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

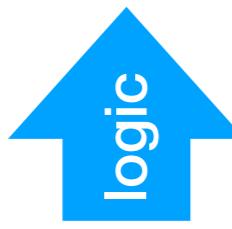
← many concrete cases

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

Grand Challenge: Abstract Abstraction

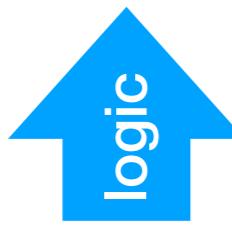
 **lemma** "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

 **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

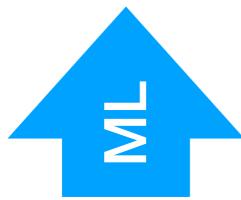


← abstraction using expressive logic

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



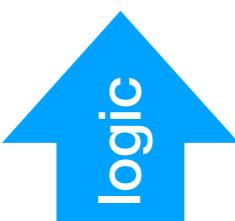
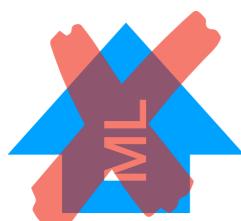
 **Lemma** "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

 **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



← abstraction using expressive logic

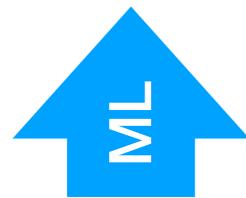
lemma "itrev [1,2] [] = rev [1,2] @ []" **by** auto
lemma "itrev [1,2,3] [] = rev [1,2,3] @ []" **by** auto
lemma "itrev [''a'', ''b''] [] = rev [''a'', ''b''] @ []" **by** auto
lemma "itrev [x,y,z] [] = rev [x,y,z] @ []" **by** auto

← many concrete cases

Grand Challenge: Abstract Abstraction

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

← even more abstract



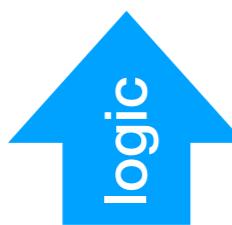
 **Lemma** "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

← small dataset about different domains

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

← one abstract representation

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



← abstraction using expressive logic

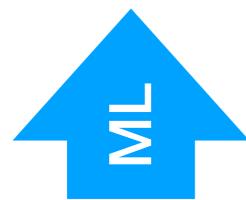
lemma "itrev [1,2] [] = rev [1,2] @ []" **by** auto
lemma "itrev [1,2,3] [] = rev [1,2,3] @ []" **by** auto
lemma "itrev [''a'', ''b''] [] = rev [''a'', ''b''] @ []" **by** auto
lemma "itrev [x,y,z] [] = rev [x,y,z] @ []" **by** auto

← many concrete cases

Grand Challenge: Abstract Abstraction

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

<- even more abstract



<- pros: good at ambiguity (heuristics)



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

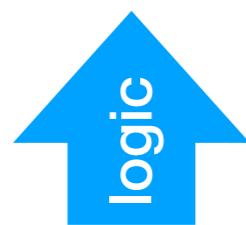
<- small dataset about
different domains



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

<- one abstract representation

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



<- abstraction using expressive logic

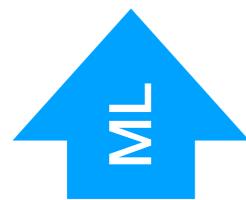
```
lemma "itrev [1,2] [] = rev [1,2] @ []" by auto  
lemma "itrev [1,2,3] [] = rev [1,2,3] @ []" by auto  
lemma "itrev [''a'', ''b''] [] = rev [''a'', ''b''] @ []" by auto  
lemma "itrev [x,y,z] [] = rev [x,y,z] @ []" by auto
```

<- many concrete cases

Grand Challenge: Abstract Abstraction

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

← even more abstract



← 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)
```

← small dataset about
different domains



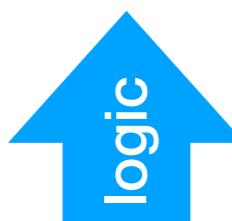
```
lemma "exec (is1 @ is2) s stk =  
      exec is2 s (exec is1 s stk)"  
by(induct is1 s stk rule:exec.induct) auto
```

← small dataset about
different domains



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

← one abstract representation



← 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

Many key challenges remain

Unsupervised Learning

Memory and one-shot learning

Imagination-based Planning with
Generative Models

Learning Abstract Concepts

Transfer Learning

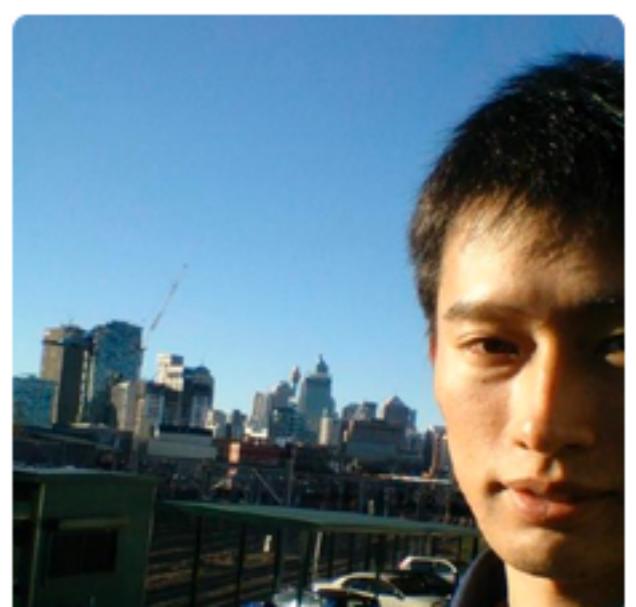
Language understanding



CENTER FOR
Brains
Minds+
Machines

March 20, 2019

The Power of
Self



Many key challenges remain

Unsupervised Learning

Memory and one-shot learning

Imagination-based Planning with
Generative Models

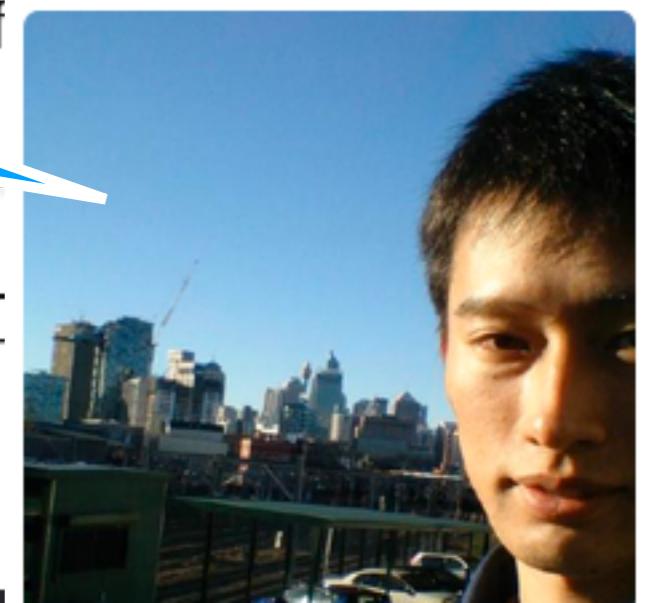
Learning Abstract Concepts

Abstract concepts?



March 20, 2019

The Power of
Self



Many key challenges remain

Unsupervised Learning

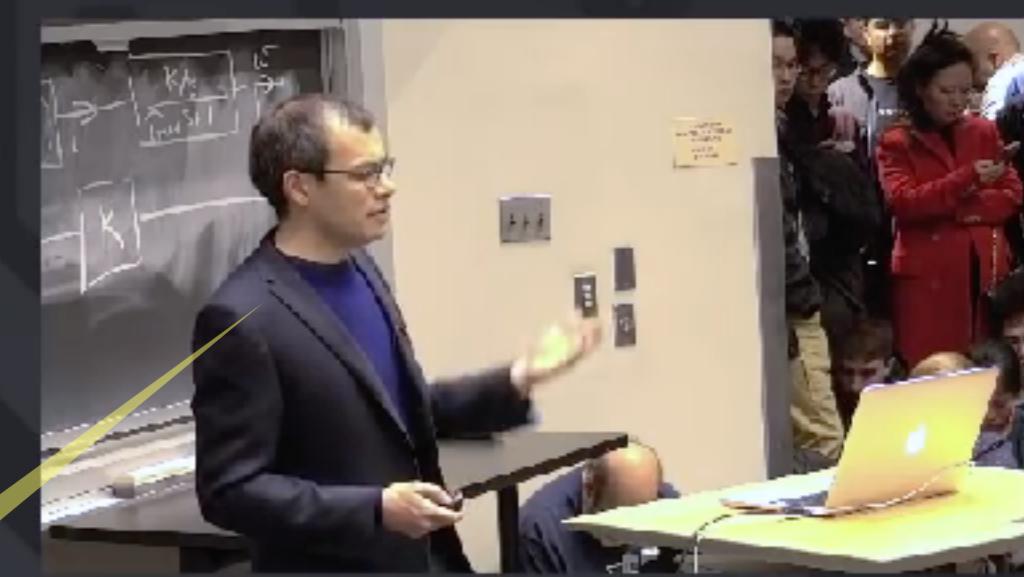
Memory and one-shot learning

Imagination-based Planning with
Generative Models

Learning Abstract Concepts

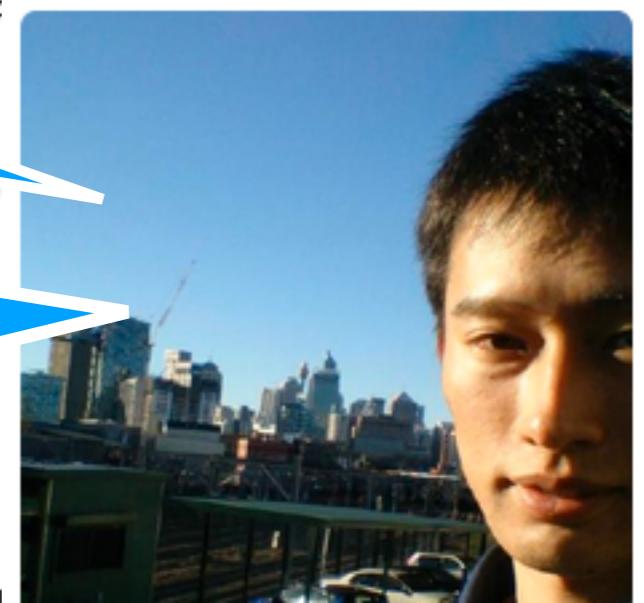
Abstract concepts?

why not logic? (LiFtEr)



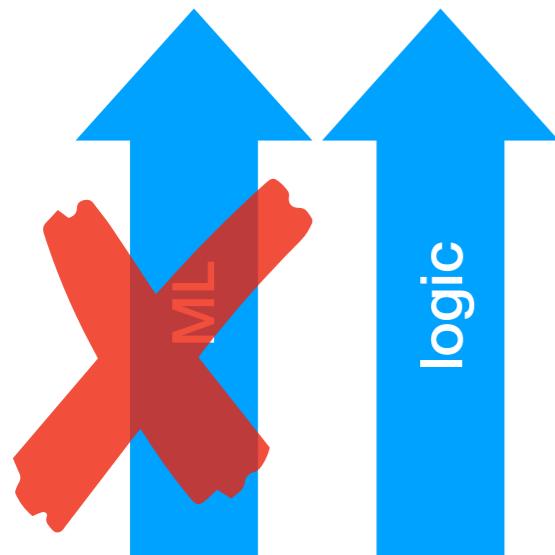
March 20, 2019

The Power of
Self



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

← even more abstract
Big Picture



abstraction using
another logic (LIFTer)

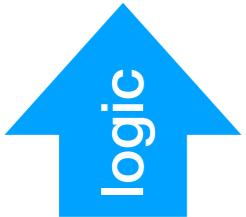
 **lemma** "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

 **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



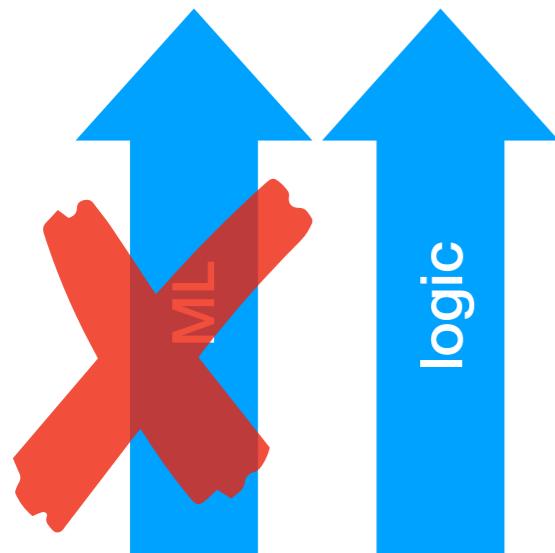
← abstraction using expressive logic

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

← many concrete cases

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

← even more abstract
Big Picture



abstraction using
another logic (LIFTer)

← pros: good at rigorous abstraction



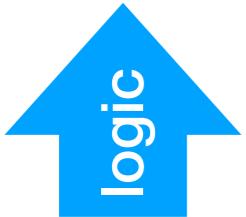
 **lemma** "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

← small dataset about
different domains

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

← one abstract representation

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



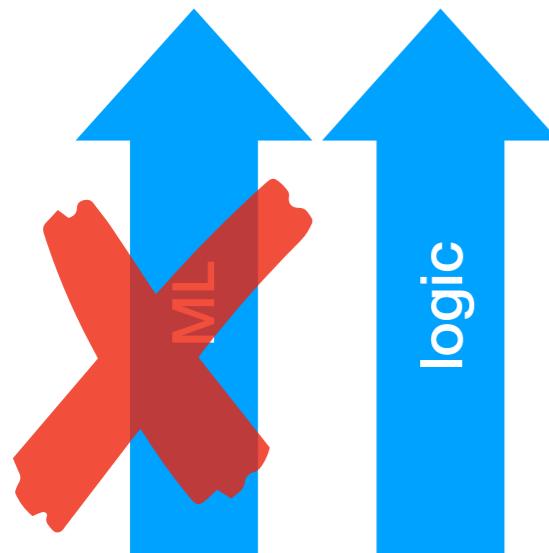
← abstraction using expressive logic

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

← many concrete cases

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

← even more abstract
Big Picture



abstraction using
another logic (LIFTEr)

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



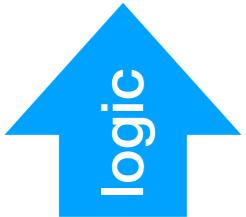
 **Lemma** "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

← small dataset about
different domains

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

← one abstract representation

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



← abstraction using expressive logic

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

← many concrete cases

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

Big Picture

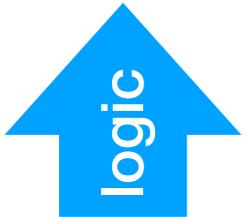
 **lemma** "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

 **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



← 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

 **Lemma** "star r x y \Rightarrow star r y z \Rightarrow 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

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 \Rightarrow star r y z \Rightarrow 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

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

Big Picture

ML

Lifter

← pros: good at rigorous abstraction



Lemma "star r x y ==> star r y z ==> star r x z"
by(induction rule: star.induct)(auto simp: step)

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

← small dataset about
different domains

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

← one abstract representation

logic

← abstraction using expressive logic

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

← many concrete cases

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

Big Picture

ML

← pros: good at ambiguity (heuristics)



Lifter

← pros: good at rigorous abstraction



 **Lemma** "star r x y \Rightarrow star r y z \Rightarrow 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

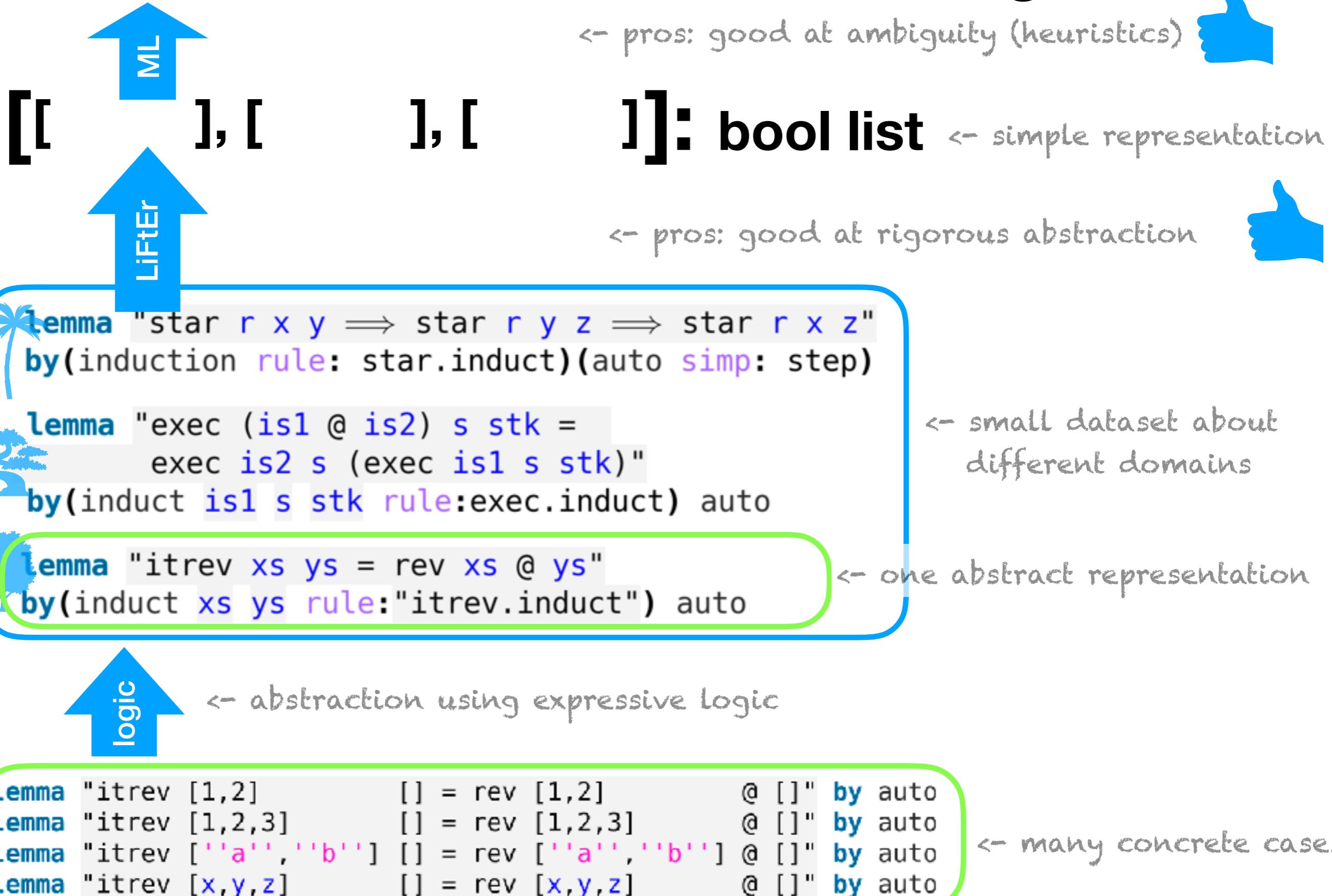
← abstraction using expressive logic

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

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

Big Picture



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

Big Picture



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

<- 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)
```

<- small dataset about different domains

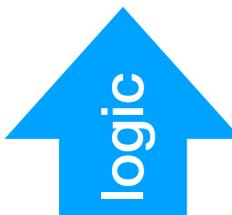


```
lemma "exec (is1 @ is2) s stk =  
      exec is2 s (exec is1 s stk)"  
by(induct is1 s stk rule:exec.induct) auto
```



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

<- 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

ML

[[], []]

LiftEr¹

<- pros: good at ambiguity (heuristics)



]]: bool list

<- simple representation



<- pros: good at rigorous abstraction

 **Lemma** "star r x y \Rightarrow star r y z \Rightarrow 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

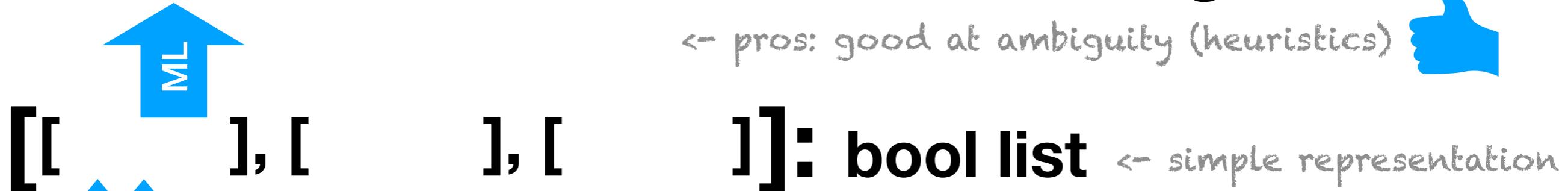
<- abstraction using expressive logic

 **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

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

Big Picture



Lemma "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

← pros: good at rigorous abstraction

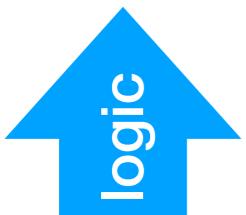


lemma "exec (is1 @ is2) s stk =
exec is2 s (exec is1 s stk)"
by(induct is1 s stk rule:exec.induct) auto

← small dataset about
different domains

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

← one abstract representation



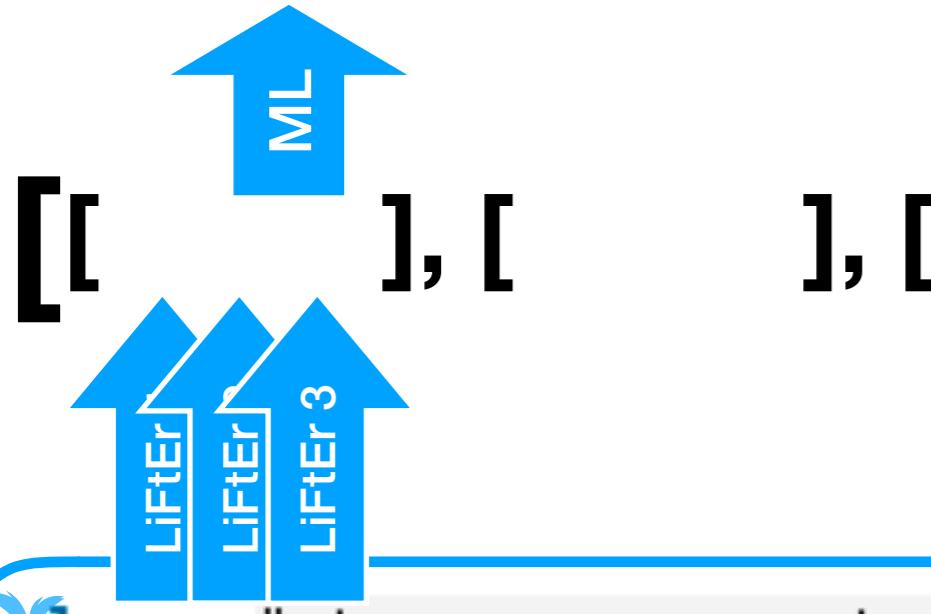
← abstraction using expressive logic

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

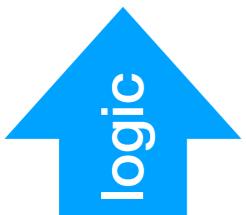
 **Lemma** "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

← small dataset about different domains

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

← one abstract representation

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



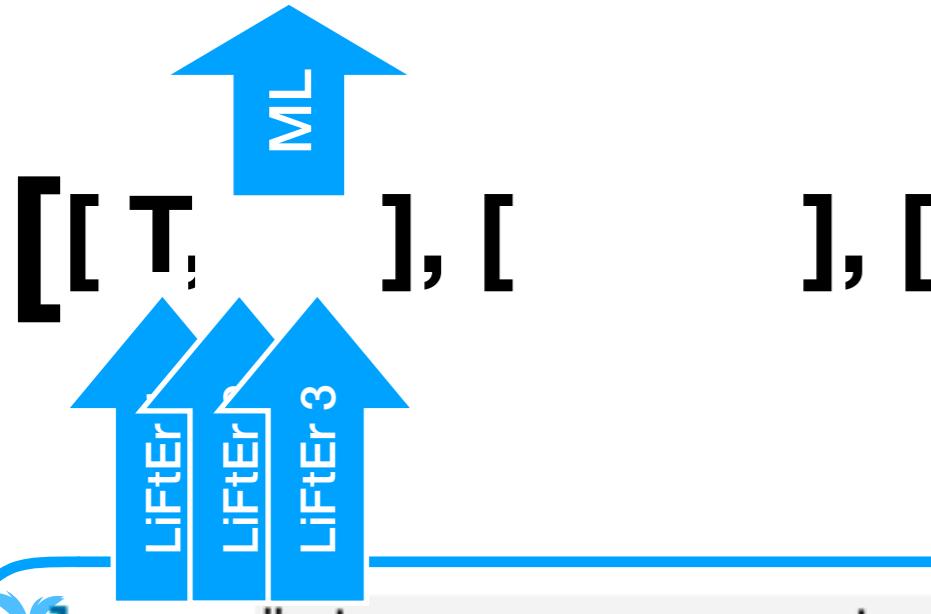
← abstraction using expressive logic

 **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)



← simple representation

← pros: good at rigorous abstraction



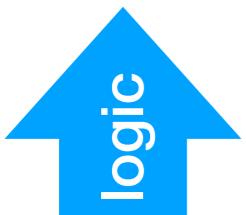
 **Lemma** "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

← small dataset about different domains

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

← one abstract representation

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



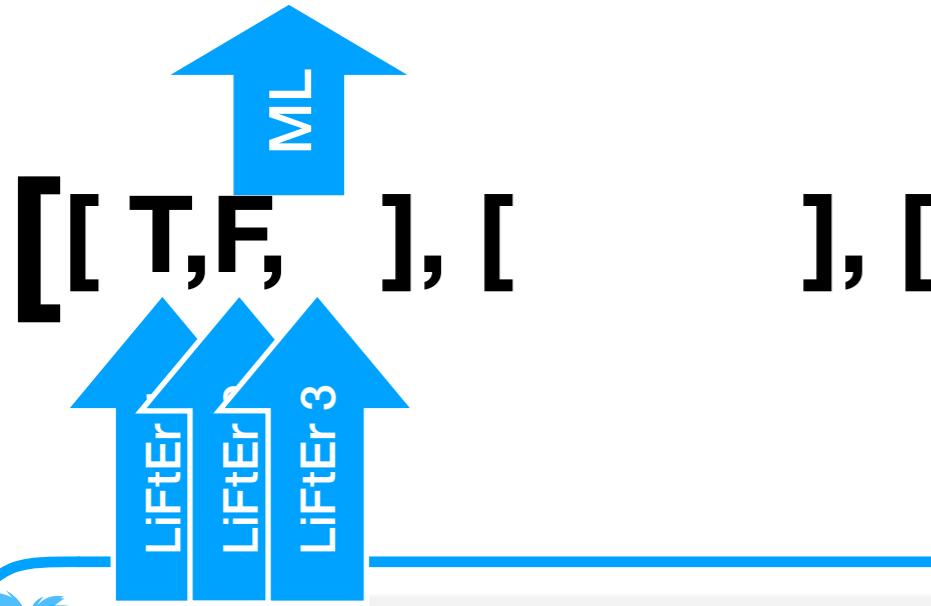
← abstraction using expressive logic

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

← simple representation

← pros: good at rigorous abstraction



Lemma "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

← small dataset about different domains

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

← one abstract representation

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



← 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

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

<- pros: good at ambiguity (heuristics)



LiftEr
LiftEr
LiftEr 3

<- simple representation



<- pros: good at rigorous abstraction

Lemma "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

<- small dataset about different domains

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

<- one abstract representation

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

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

ML

[[T,F,T], [T,T,T], [

← pros: good at ambiguity (heuristics)



]]: bool list

← simple representation



Lemma "star r
by(induction r

LiftEr 1

LiftEr 2

LiftEr 3

star r y z ==> star r x z"
ar.induct)(auto simp: step)

← pros: good at rigorous abstraction

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

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

Big Picture

ML

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

← pros: good at ambiguity (heuristics)



← simple representation

LiftEr
LiftEr
LiftEr 3
LiftEr 1
LiftEr 2
LiftEr 3
star r
ar.in
star r x z
simp: step

← pros: good at rigorous abstraction



Lemma "star r by(induction r)"
lemma "exec (is1 @ is2) s
exec is2 s (exec is1 s)"
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 notion of “good” application of induction.
Heuristics that are valid across problem domains.

Big Picture

ML

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

← pros: good at ambiguity (heuristics)



← simple representation



← pros: good at rigorous abstraction

Lemma "star r
by(induction r)"

LiftEr 1
LiftEr 2
LiftEr 3

LiftEr 1
LiftEr 2
LiftEr 3

LiftEr 1
LiftEr 2
LiftEr 3

star r x z"
auto simp: step)

lemma "exec (is1 @ is2) s
exec is2 s (exec is1 s)"
by(induct is1 s stk rule:euct) 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 notion of “good” application of induction.
Heuristics that are valid across problem domains.

Big Picture

ML

← pros: good at ambiguity (heuristics)



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

← simple representation



← pros: good at rigorous abstraction

Lemma "star r
by(induction r

LiftEr 1
LiftEr 2
LiftEr 3

LiftEr 1
LiftEr 2
LiftEr 3

LiftEr 1
LiftEr 2
LiftEr 3

star
ar.in
star r x z"
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 ys rule:"itrev.induct") auto

example?

← 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

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



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

\rightarrow

$\exists r1 : \text{rule}.$
 $\exists t1 : \text{term}.$
 $\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$
 $r1 \text{ is_rule_of } to1$

\wedge

$\forall t2 : \text{term} \in \text{induction_term}.$
 $\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$
 $\exists n : \text{number}.$
 $\text{is_nth_argument_of } (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

↓

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

→

$\exists r1 : \text{rule}.$
 $\exists t1 : \text{term}.$
 $\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$
 $r1 \text{ is_rule_of } to1$

\wedge ↪ conjunction

$\forall t2 : \text{term} \in \text{induction_term}.$
 $\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$
 $\exists n : \text{number}.$
 $\text{is_nth_argument_of} (to2, n, to1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

↓
→ $\exists r1 : \text{rule}. \text{True}$ variable for auxiliary lemmas

$\exists r1 : \text{rule}.$ ←

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge ← conjunction

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

↓
 $\exists r1 : \text{rule}. \text{True}$ → variable for auxiliary lemmas
 →
 $\exists r1 : \text{rule}.$ ←
 $\exists t1 : \text{term}.$ ← variable for terms
 $\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$
 $r1 \text{ is_rule_of } to1$
 \wedge ← conjunction
 $\forall t2 : \text{term} \in \text{induction_term}.$
 $\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$
 $\exists n : \text{number}.$
 $\text{is_nth_argument_of } (to2, n, to1)$
 \wedge
 $t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

↓
 $\exists r1 : \text{rule}. \text{True}$ → variable for auxiliary lemmas
 $\exists r1 : \text{rule}.$ ←
 $\exists t1 : \text{term}.$ ← variable for terms
 $\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$ ← variable for term occurrences
 $r1 \text{ is_rule_of } to1$ ← conjunction
 \wedge ←
 $\forall t2 : \text{term} \in \text{induction_term}.$
 $\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$
 $\exists n : \text{number}.$
 $\text{is_nth_argument_of } (to2, n, to1)$
 \wedge
 $t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

↓
 $\exists r1 : \text{rule}. \text{True}$ → variable for auxiliary lemmas
→
 $\exists r1 : \text{rule}.$ ←
 $\exists t1 : \text{term}.$ ← variable for terms
 $\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$ ← variable for term occurrences
 $r1 \text{ is_rule_of } to1$ ←
Λ ← conjunction
 $\forall t2 : \text{term} \in \text{induction_term}.$
 $\exists to2 : \text{term_occurrence} \in t2 : \text{term}.$
 $\exists n : \text{number}.$ ← variable for natural numbers
 $\text{is_nth_argument_of } (to2, n, to1)$
Λ
 $t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

\rightarrow variable for auxiliary lemmas

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$ variable for terms

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$ variable for term occurrences

\wedge conjunction

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

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

$\exists n : \text{number}.$ variable for natural numbers

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

\wedge

$t2 \text{ is_nth_induction_term } n$

universal
quantifier

Example Assertion in LiFtEr (in Abstract Syntax)

implication existential quantifier

$\exists r1 : \text{rule}. \text{True}$ variable for auxiliary lemmas

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$ variable for terms

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$ variable for term occurrences

$r1 \text{ is_rule_of } to1$

\wedge conjunction

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

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

$\exists n : \text{number}.$ variable for natural numbers

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

\wedge

$t2 \text{ is_nth_induction_term } n$

<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
```

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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

<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
```

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

→

$\exists r1 : \text{rule}.$

($r1 = \text{itrev.induct}$)

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

r1

<https://twitter.com/YutakangE>

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

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

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

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

\rightarrow

$\exists r1 : \text{rule}.$

($r1 = \text{itrev.induct}$)

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

r1

<https://twitter.com/YutakangE>

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

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

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

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

\rightarrow

$\exists r1 : \text{rule}.$

($r1 = \text{itrev.induct}$)

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

$r1$

<https://twitter.com/YutakangE>

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

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

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

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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists tol : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } tol$

\wedge

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

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

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

$\text{is_nth_argument_of} (to2, n, tol)$

\wedge

$t2 \text{ is_nth_induction_term } n$



<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
```

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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

T

t1

r1

(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
```

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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

T
t1

r1

(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
```

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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

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

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

T

t1

r1

(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
```

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

\rightarrow

$\exists r1 : \text{rule}.$

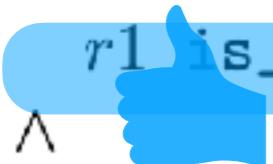
$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

($r1 = \text{itrev.induct}$)

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

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

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

<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
```

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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

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



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

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

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

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



$t2 \text{ is_nth_induction_term } n$



($r1 = \text{itrev.induct}$)
($t1 = \text{itrev}$)
($to1 = \text{itrev}$)

<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
```

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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

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

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$



($r1 = \text{itrev.induct}$)

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

<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
```

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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

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

\wedge

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

$(t2 = xs \text{ and } ys)$

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

t2

T

t1

r1

$(r1 = \text{itrev.induct})$

$(t1 = \text{itrev})$

$(to1 = \text{itrev})$

<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)"

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

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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

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

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

($r1 = \text{itrev.induct}$)

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

T

t1

r1

($t2 = \text{xs and ys}$)

($to2 = \text{xs and ys}$)

<https://twitter.com/YutakangE>

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

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

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

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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

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



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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

($r1 = \text{itrev.induct}$)

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

T

t1

r1

($t2 = \text{xs and ys}$)

($to2 = \text{xs and ys}$)

<https://twitter.com/YutakangE>

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

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

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

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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

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

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

($r1 = \text{itrev.induct}$)

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

T

t1

r1

($t2 = \text{xs and ys}$)

($to2 = \text{xs and ys}$)

<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

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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

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

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

($r1 = \text{itrev.induct}$)

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

T

t1

r1

T

t1

($t2 = \text{xs}$ and ys)

($to2 = \text{xs}$ and ys)

when $t2$ is xs ($n = 1$) ?

<https://twitter.com/YutakangE>

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

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

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

t₀₁

first

t₀₂

∃ r₁ : rule. True

→

∃ r₁ : rule.

∃ t₁ : term.

∃ t₀₁ : term_occurrence ∈ t₁ : term.

r₁ is_rule_of t₀₁ True! r₁ (= itrev.induct) is a lemma about t₀₁ (= itrev).

^

∀ t₂ : term ∈ induction_term.

∃ t₀₂ : term_occurrence ∈ t₂ : term.

∃ n : number.

is_nth_argument_of (t₀₂, n, t₀₁)

^

t₂ is_nth_induction_term n

(r₁ = itrev.induct)

(t₁ = itrev)

(t₀₁ = itrev)

(t₂ = xs and ys)

(t₀₂ = xs and ys)

when t₂ is xs (n = 1)

<https://twitter.com/YutakangE>

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

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

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

to1

first second to2

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

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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

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



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

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

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

is_nth_argument_of (to2, n, to1)



t2 is_nth_induction_term n

(r1 = itrev.induct)

(t1 = itrev)

(to1 = itrev)

(t2 = xs and ys)

(to2 = xs and ys)

when t2 is xs (n = 1)

when t2 is ys (n = 2) ?

<https://twitter.com/YutakangE>

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

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

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

to1

first second to2

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

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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

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



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

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

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

is_nth_argument_of (to2, n, to1)



t2 is_nth_induction_term n

(r1 = itrev.induct)

(t1 = itrev)

(to1 = itrev)

(t2 = xs and ys)

(to2 = xs and ys)

when t2 is xs (n = 1)

when t2 is ys (n = 2)

<https://twitter.com/YutakangE>

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

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

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

to1

first second to2

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

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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

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

\wedge

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

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

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

is_nth_argument_of (to2, n, to1)

\wedge

t2 is_nth_induction_term n

(r1 = itrev.induct)

(t1 = itrev)

(to1 = itrev)

(t2 = xs and ys)

(to2 = xs and ys)

when t2 is xs (n = 1)

when t2 is ys (n = 2)

```
 $\exists r1 : \text{rule}. \text{True}$ 
```

\rightarrow

```
 $\exists r1 : \text{rule}.$ 
```

```
 $\exists t1 : \text{term}.$ 
```

```
 $\exists to1 : \text{term\_occurrence} \in t1 : \text{term}.$ 
```

```
 $r1 \text{ is\_rule\_of } to1$ 
```

\wedge

```
 $\forall t2 : \text{term} \in \text{induction\_term}.$ 
```

```
 $\exists to2 : \text{term\_occurrence} \in t2 : \text{term}.$ 
```

```
 $\exists n : \text{number}.$ 
```

```
 $\text{is\_nth\_argument\_of} (to2, n, to1)$ 
```

\wedge

```
 $t2 \text{ is\_nth\_induction\_term } n$ 
```

the same LIFTER assertion



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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

new types ->

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

the same LIFTER assertion



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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

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"
```

the same LIFTER assertion

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



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

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

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) s stk = n # stk" |  
  "exec1 (LOAD x) s stk = s(x) # stk" |  
  "exec1 ADD s (j#i#stk) = (i + j) # stk"
```

the same LIFTER assertion

```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
  "exec [] s 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)
 $\exists r1 : \text{rule}. \text{True}$ apply auto done

→

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

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)

$\exists r1 : \text{rule}. \text{True}$ apply auto done

→

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

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)

$\exists r1 : \text{rule. } \text{True }$ apply auto done

r1

→

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

$\exists t1 : \text{term. }$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term. }$

$r1 \text{ is_rule_of } to1$

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

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)

$\exists r1 : \text{rule}. \text{True}$ apply auto done

r1

$\exists r1 : \text{rule}.$

(r1 = exec.induct)

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

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)

$\exists r1 : \text{rule}. \text{True}$ apply auto done

r1

$\rightarrow \exists r1 : \text{rule}.$

(r1 = exec.induct)

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

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)

$\exists r1 : \text{rule}. \text{True}$ apply auto done



→

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists tol : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } tol$

\wedge

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

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

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

$\text{is_nth_argument_of} (to2, n, tol)$

\wedge

$t2 \text{ is_nth_induction_term } n$

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

new types ->

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

new constants ->

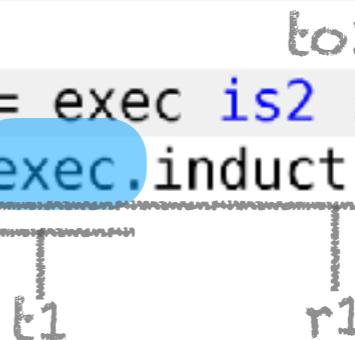
```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

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

new lemma -> lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> apply(induct is1 s stk rule:exec.induct)

$\exists r1 : \text{rule}. \text{True}$ apply auto done



→

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists tol : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } tol$

\wedge

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

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

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

$\text{is_nth_argument_of} (to2, n, tol)$

\wedge

$t2 \text{ is_nth_induction_term } n$

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

new types ->

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

new constants ->

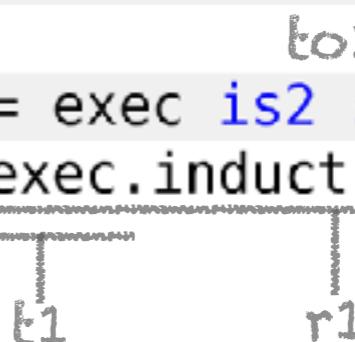
```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

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

new lemma -> lemma "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> apply(induct is1 s stk rule:exec.induct)

$\exists r1 : \text{rule}. \text{True}$ apply auto done



$\rightarrow \exists r1 : \text{rule}.$

($r1 = \text{exec.induct}$)

($t1 = \text{exec}$)

($to1 = \text{exec}$)

$\exists t1 : \text{term}.$

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

$r1 \text{ is_rule_of } to1$

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

new types ->

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

new constants ->

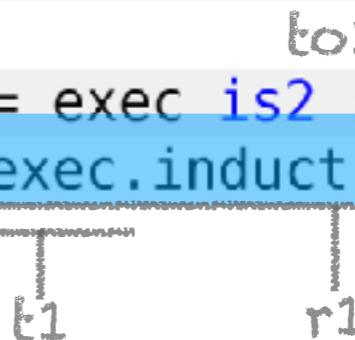
```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
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)

$\exists r1 : \text{rule}. \text{True}$ apply auto done



→

$\exists r1 : \text{rule}.$

($r1 = \text{exec.induct}$)

$\exists t1 : \text{term}.$

($t1 = \text{exec}$)

$\exists to1 : \text{term_occurrence} \in t1 : \text{term}.$

($to1 = \text{exec}$)

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

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

new types ->

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

new constants ->

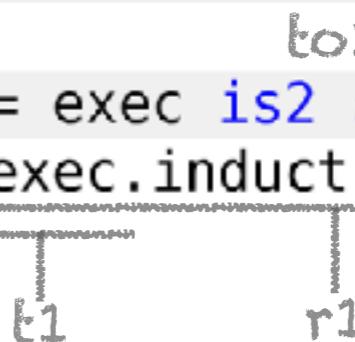
```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
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)

$\exists r1 : \text{rule}. \text{True}$ apply auto done



$\rightarrow \exists r1 : \text{rule}.$

($r1 = \text{exec.induct}$)

($t1 = \text{exec}$)

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

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

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

new types ->

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

new constants ->

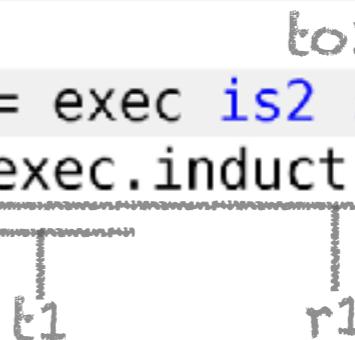
```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
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)

$\exists r1 : \text{rule}. \text{True}$ apply auto done



→

$\exists r1 : \text{rule}.$

($r1 = \text{exec.induct}$)

($t_1 = \text{exec}$)

$\exists to1 : \text{term_occurrence} \in t_1 : \text{term}. (to1 = \text{exec})$

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



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

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

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

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



$t2 \text{ is_nth_induction_term } n$

new types ->

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

new constants ->

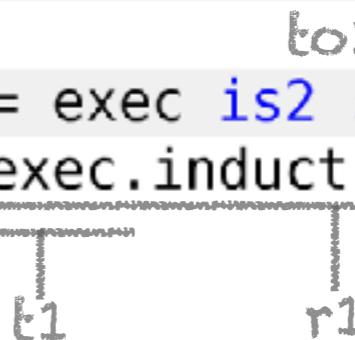
```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
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)

$\exists r1 : \text{rule}. \text{True}$ apply auto done



→

$\exists r1 : \text{rule}.$

($r1 = \text{exec.induct}$)

($t1 = \text{exec}$)

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

$r1 \text{ is_rule_of } t1$ True! $r1 (= \text{exec.induct})$ is a lemma about $t1 (= \text{exec})$.

^

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

$\exists t2 : \text{term_occurrence} \in t2 : \text{term}.$

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

is_nth_argument_of (t2, n, t1)

^

t2 is_nth_induction_term n

new types ->

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

new constants ->

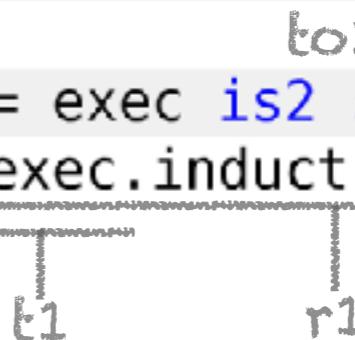
```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
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)**

$\exists r1 : \text{rule}. \text{True}$ **apply auto done**



$\rightarrow \exists r1 : \text{rule}.$

($r1 = \text{exec.induct}$)

($t1 = \text{exec}$)

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

$r1 \text{ is_rule_of } t1$ True! $r1 (= \text{exec.induct})$ is a lemma about $t1 (= \text{exec})$.

\wedge

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

($t2 = \text{is1, s, and stk}$)

$\exists t2 : \text{term} \in t2 : \text{term}.$

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

$\text{is_nth_argument_of} (t2, n, t1)$

\wedge

$t2 \text{ is_nth_induction_term } n$

new types ->

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

new constants ->

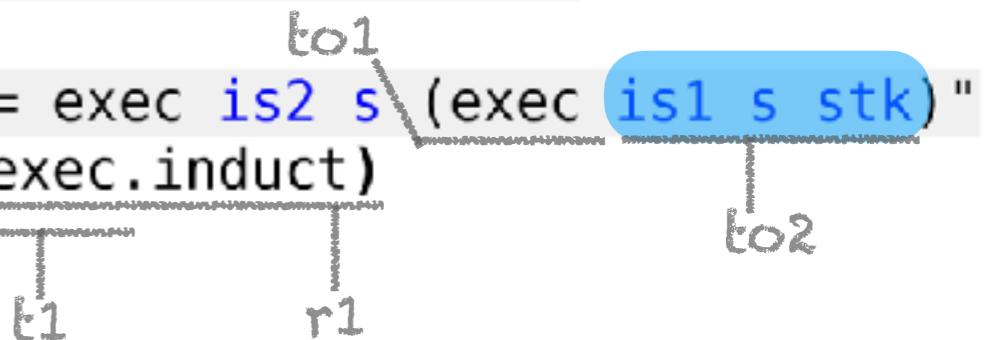
```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) _ s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
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)**

$\exists r1 : \text{rule}. \text{True}$ **apply auto done**



$\rightarrow \exists r1 : \text{rule}.$

($r1 = \text{exec.induct}$)

($t1 = \text{exec}$)

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

$\exists r1 : \text{rule_of } t1$ True! $r1 (= \text{exec.induct})$ is a lemma about $t1 (= \text{exec})$.



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

($t2 = \text{is1, s, and stk}$)

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

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

$\text{is_nth_argument_of } (t2, n, t1)$



$t2 \text{ is_nth_induction_term } n$

new types ->

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

new constants ->

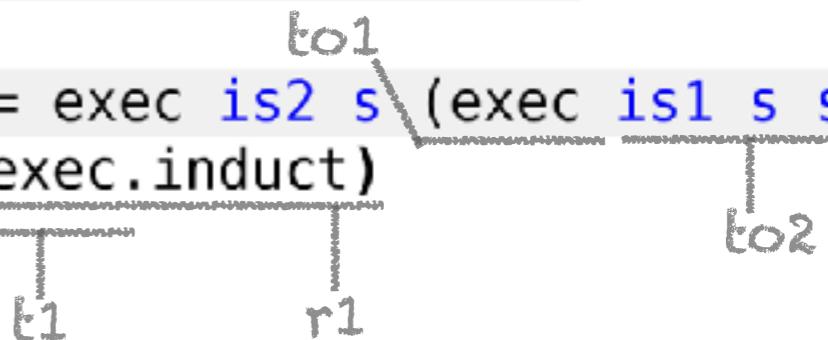
```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
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)**

$\exists r1 : \text{rule}. \text{True}$ **apply auto done**



$\rightarrow \exists r1 : \text{rule}.$

($r1 = \text{exec.induct}$)

($t1 = \text{exec}$)

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

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



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

($t2 = \text{is1, s, and stk}$)

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

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

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



$t2 \text{ is_nth_induction_term } n$

new types ->

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

new constants ->

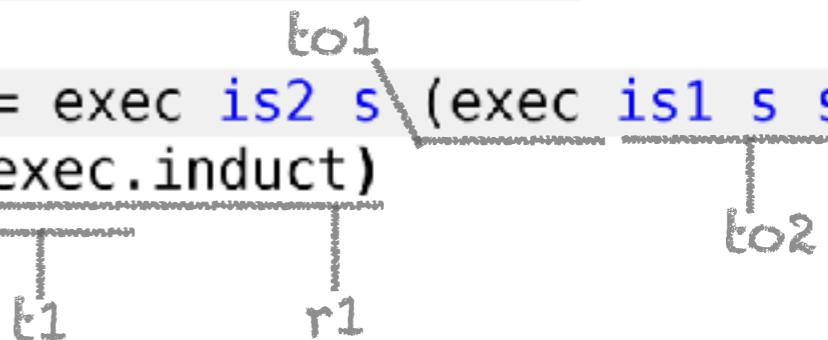
```
fun exec1 :: "instr ⇒ state ⇒ stack ⇒ stack" where  
"exec1 (LOADI n) _ stk = n # stk" |  
"exec1 (LOAD x) s stk = s(x) # stk" |  
"exec1 ADD _ (j#i#stk) = (i + j) # stk"
```

```
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)**

$\exists r1 : \text{rule}. \text{True}$ **apply auto done**



$\rightarrow \exists r1 : \text{rule}.$

($r1 = \text{exec.induct}$)

($t1 = \text{exec}$)

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

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



$\wedge \forall t2 : \text{term} \in \text{induction_term}.$

($t2 = \text{is1, s, and stk}$)

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

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

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



$t2 \text{ is_nth_induction_term } n$

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)**

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

apply auto done

$\frac{}{\begin{array}{c} t_1 \\ r_1 \end{array}}$

$\frac{}{\begin{array}{c} t_1 \\ t_2 \end{array}}$

first

to1

to2

$\exists r1 : \text{rule}.$

($r_1 = \text{exec.induct}$)

($t_1 = \text{exec}$)

($t_{01} = \text{exec}$)

$\exists t1 : \text{term}.$

first

$\exists t01 : \text{term_occurrence} \in t1 : \text{term}.$

($t_{01} = \text{exec}$)

$r1 \text{ is_rule_of } t01 \quad \text{True! } r1 (= \text{exec.induct}) \text{ is a lemma about } t01 (= \text{exec}).$

\wedge

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

($t_2 = \text{is1, s, and stk}$)

$\exists t02 : \text{term_occurrence} \in t2 : \text{term}.$

($t_{02} = \text{is1, s, and stk}$)

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

$\text{is_nth_argument_of } (t02, n, t01)$

when t_2 is is1 ($n \rightarrow 1$) ?

\wedge

$t2 \text{ is_nth_induction_term } n$

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)~~

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

apply auto done

t_1

t_{01}

t_{02}

$\exists r1 : \text{rule}.$

($r1 = \text{exec.induct}$)

($t_1 = \text{exec}$)

($t_{01} = \text{exec}$)

$\exists t1 : \text{term}.$

$r1 \text{ is_rule_of } t1 \quad \text{True! } r1 (= \text{exec.induct}) \text{ is a lemma about } t_{01} (= \text{exec}).$

^ 

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

($t_2 = \text{is1, s, and stk}$)

$\exists t2 : \text{term_occurrence} \in t2 : \text{term}.$

($t_{02} = \text{is1, s, and stk}$)

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

is_nth_argument_of (t_{02}, n, t_{01})

when t_2 is is1 ($n \rightarrow 1$) 

^

$t_2 \text{ is_nth_induction_term } n$

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)**

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

apply auto done

\vdash

t_1

r_1

first

second ($r_1 = \text{exec.induct}$)

($t_1 = \text{exec}$)

($t_1 = \text{exec}$)

$\exists t_1 : \text{term}. \quad \exists r_1 : \text{rule}. \quad \exists t_1 : \text{term}. \quad \exists r_1 : \text{rule}. \quad \text{True! } r_1 (= \text{exec.induct}) \text{ is a lemma about } t_1 (= \text{exec}).$

\wedge

$\forall t_2 : \text{term} \in \text{induction_term}. \quad \exists t_2 : \text{term} \in \text{induction_term}. \quad \exists r_2 : \text{rule}. \quad \text{True! } r_2 (= \text{exec.induct}) \text{ is a lemma about } t_2 (= \text{exec}).$

($t_2 = \text{is1, s, and stk}$)

($t_2 = \text{is1, s, and stk}$)

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

is_nth_argument_of (t_2, n, r_2)

when t_2 is is1 ($n \rightarrow 1$)

\wedge

t2 is_nth_induction_term n

when t_2 is s ($n \rightarrow 2$)?

new types ->

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

new constants ->

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

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

new lemma -> **lemma** "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> **apply(induct is1 s stk rule:exec.induct)**

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

apply auto done

\vdash

t_1

r_1

$\exists r1 : \text{rule}.$

first

$\exists t1 : \text{term}.$

second

$\exists t01 : \text{term_occurrence} \in t1 : \text{term}. (t01 = \text{exec})$

$r1 \text{ is_rule_of } t01$ True! $r1 (= \text{exec.induct})$ is a lemma about $t01 (= \text{exec})$.

\wedge

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

$(t2 = is1, s, \text{and } stk)$

$\exists t02 : \text{term_occurrence} \in t2 : \text{term}. (t02 = is1, s, \text{and } stk)$

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

$\text{is_nth_argument_of } (t02, n, t01)$

when $t2$ is $is1$ ($n \rightarrow 1$)

\wedge

$t2 \text{ is_nth_induction_term } n$

when $t2$ is s ($n \rightarrow 2$)

new types ->

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

new constants ->

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

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

new lemma -> **lemma** "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"

a model proof -> **apply(induct is1 s stk rule:exec.induct)**

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

apply auto done

\rightarrow

$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

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

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

\wedge

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

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

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

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

\wedge

$t2 \text{ is_nth_induction_term } n$

when $t2$ is $is1 (n \rightarrow 1)$

when $t2$ is $s (n \rightarrow 2)$

when $t2$ is $stk (n \rightarrow 3)$?

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"
```

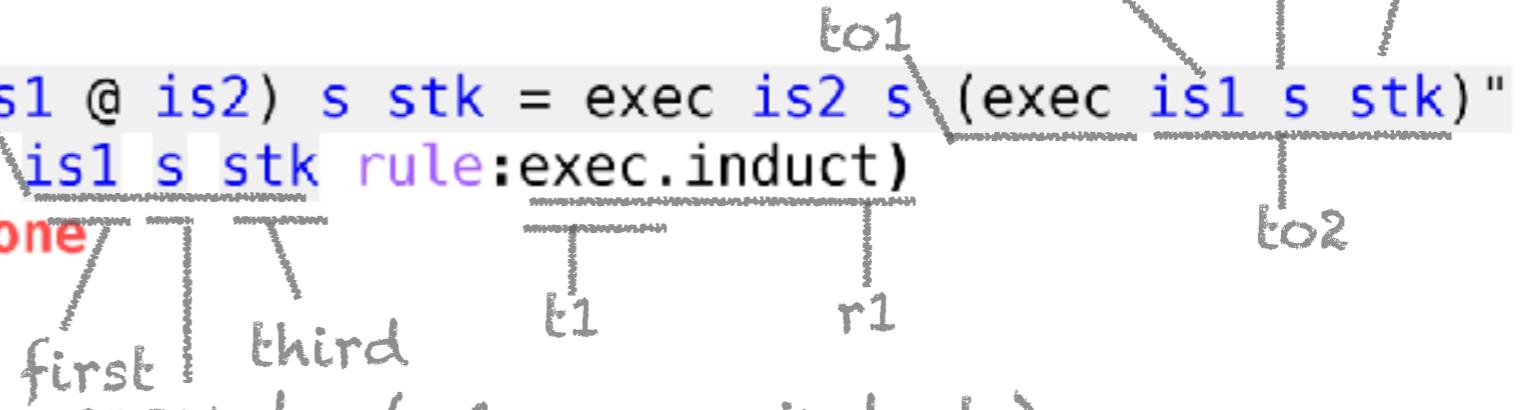
```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

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)**

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

apply auto done



$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

$\exists t01 : \text{term_occurrence} \in t1 : \text{term}. (t01 = \text{exec})$

r1 is_rule_of t01 True! r1 (= exec.induct) is a lemma about t01 (= exec).



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

$\exists t02 : \text{term_occurrence} \in t2 : \text{term}. (t02 = \text{is1, s, and stk})$

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

is_nth_argument_of (t02, n, t01)



t2 is_nth_induction_term n

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"
```

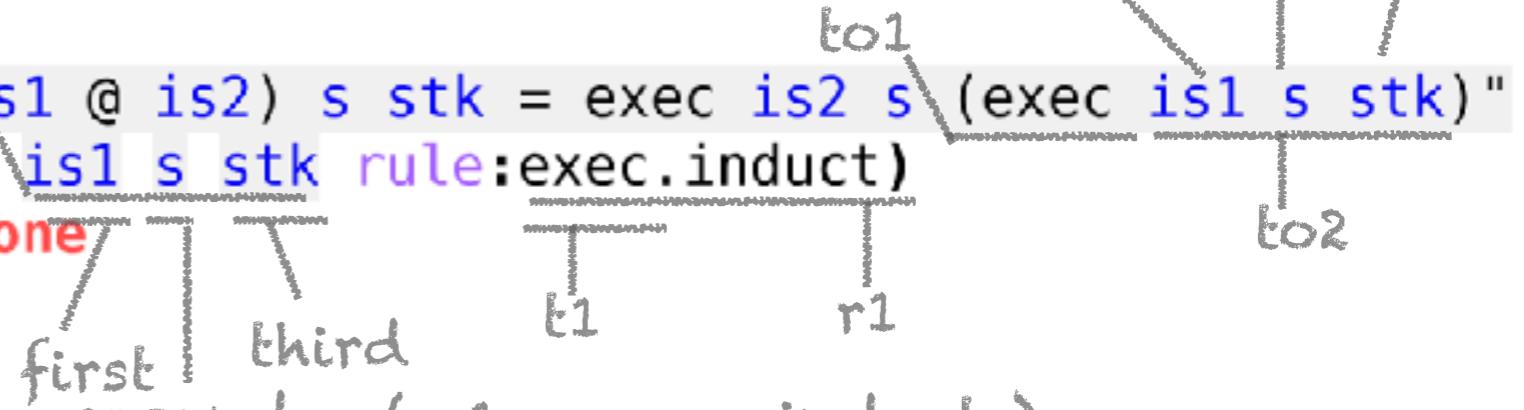
```
fun exec :: "instr list ⇒ state ⇒ stack ⇒ stack" where  
"exec [] _ stk = stk" |  
"exec (i#is) s stk = exec is s (exec1 i s stk)"
```

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)**

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

apply auto done



$\exists r1 : \text{rule}.$

$\exists t1 : \text{term}.$

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

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

\wedge

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

($t2 = \text{is1, s, and stk}$)

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

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

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

when $t2$ is is1 ($n \rightarrow 1$)

\wedge

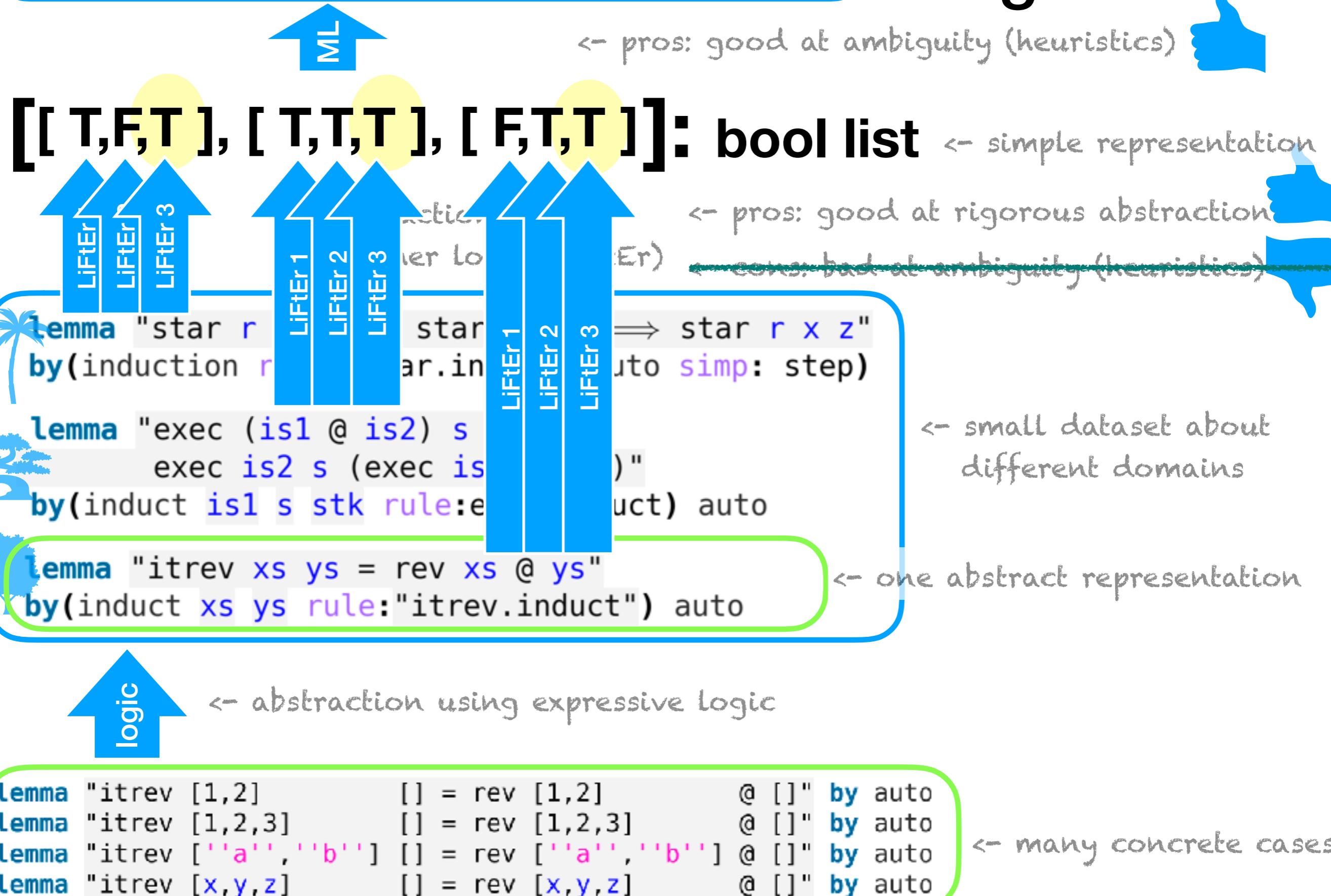
$t2 \text{ is_nth_induction_term } n$

when $t2$ is s ($n \rightarrow 2$)

when $t2$ is stk ($n \rightarrow 3$)

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

Big Picture



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

Big Picture

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

WIP

<- pros: good at ambiguity (heuristics)



The diagram illustrates the process of generating a word form from a lemma. It starts with a lemma "star r" and proceeds through three stages of lifting:

- LIFTER 1**: The lemma "star r" is processed by LIFTER 1, resulting in the prefix "star".
- LIFTER 2**: The prefix "star" is processed by LIFTER 2, resulting in the suffix ".in".
- LIFTER 3**: The suffix ".in" is processed by LIFTER 3, resulting in the final word "auto".

Each stage is represented by a blue arrow pointing upwards, indicating the progression of the lifting process.

\leftarrow pros: good at rigor

Er

~~→ goes back at ambiguity (heuristics)~~

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

\leftarrow small dataset about
different domains

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

<- one abstract representation

A blue arrow pointing upwards.

\leftarrow abstraction using expressive logic

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

← many concrete cases

<https://twitter.com/YutakangE>

3 secrets behind LiFtEr

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

3 secrets behind LiFtEr

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL



3 secrets behind LiFtEr

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

Secret 1

APLAS2019



Many induction heuristics are simple.

3 secrets behind LiFtEr

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL



Secret 1

Many induction heuristics are simple.

Secret 2

Proofs can be more abstract than theorems.

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

\leftarrow simple representation

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

\leftarrow small dataset about different domains

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

\leftarrow one abstract representation

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

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

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

\leftarrow simple representation

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

\leftarrow small dataset about different domains

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

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

\leftarrow 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]"
```

\leftarrow relevant definitions

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

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

← simple representation

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

← small dataset about different domains

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

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

← 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]"
```

← relevant definitions

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

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

\leftarrow simple representation

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

\leftarrow small dataset about different domains

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

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

\leftarrow one abstract representation

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

\leftarrow 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]"
```

\leftarrow relevant definitions

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

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

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

← simple representation

```
lemma "star r x y ==> star r x z"  
by(induction rule: star.in  
    auto simp: step)
```

← small dataset about
different domains

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

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

← one abstract representation

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

← relevant definitions

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

LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

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

<- simple representation

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

<- small dataset about different domains

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

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

<- one abstract representation

lemma "itrev xs ys = rev xs @ ys"
by(induct xs ys rule:"itrev.induct") auto "semantics" of constants (rev and itrev)

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

<- relevant definitions

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

LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

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

WIP

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

lemma "exec (is1 @ is2) s ==> exec is2 s (exec is1 s)"
by(induct is1 s stk rule: exec.induct) 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" auto
by(induct xs ys rule:"itrev.induct") auto "semantics" of constants (rev and itrev)

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

<- relevant definitions

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

LiFtEr₂ (proof goal * induction arguments) -> bool
* relevant definitions

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

WIP

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

lemma "exec (is1 @ is2) s ==> exec is2 s (exec is1 s)"
by(induct is1 s stk rule: exec.induct) 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" auto
by(induct xs ys rule:"itrev.induct") auto "semantics" of constants (rev and itrev)

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

<- relevant definitions

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

fin.