

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

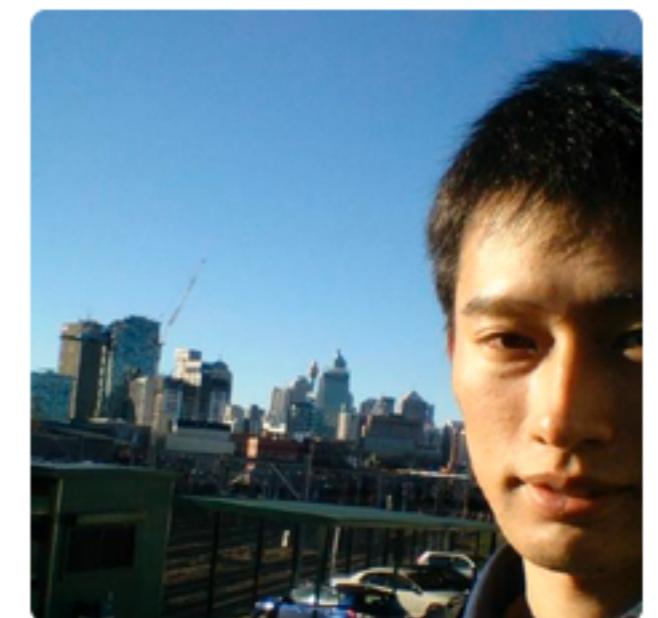
This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).



Yutaka Nagashima
University of Innsbruck
Czech Technical University



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



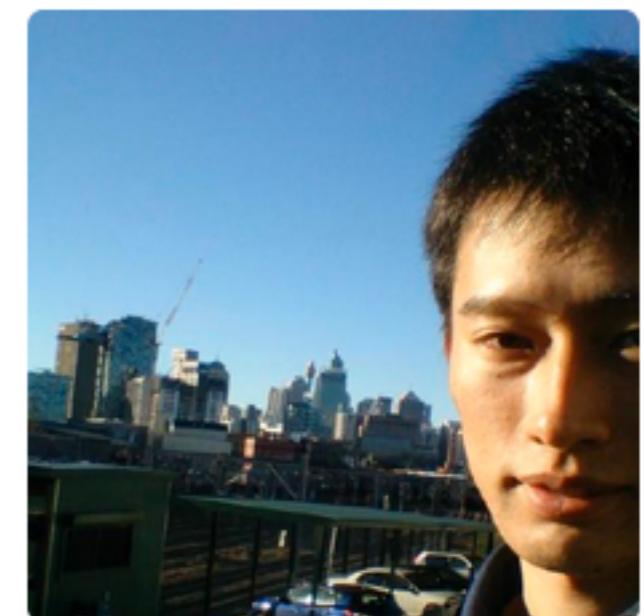
Yutaka Ng

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

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



Yutaka Nagashima
University of Innsbruck
Czech Technical University



Yutaka Ng

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

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



LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

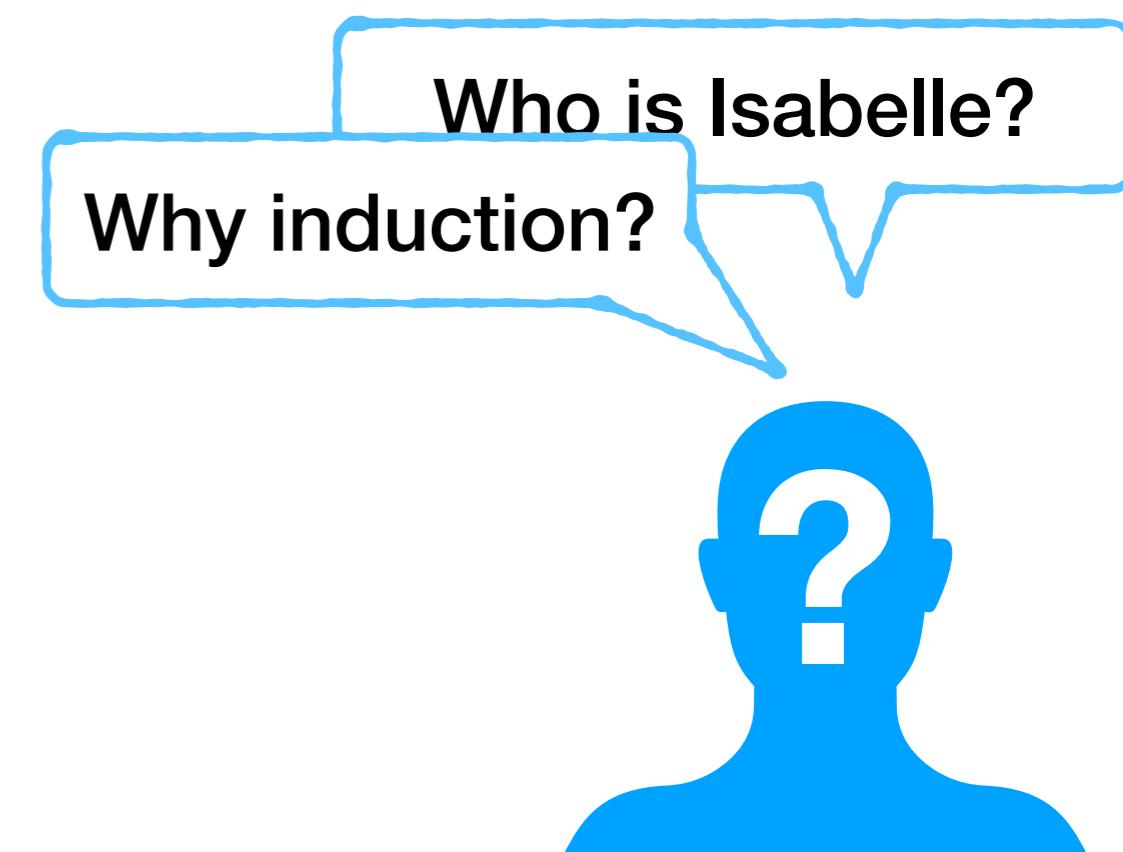
This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

Who is Isabelle?



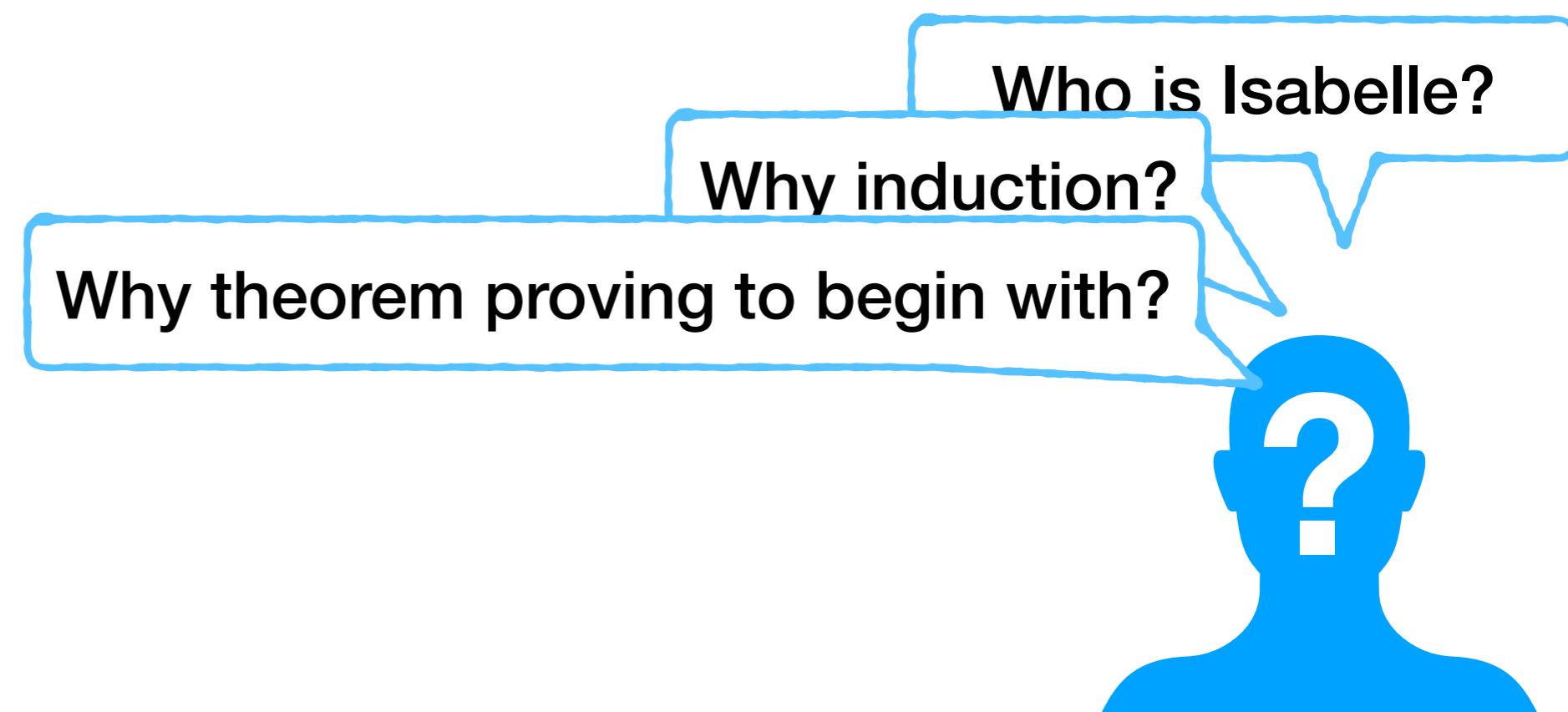
LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

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



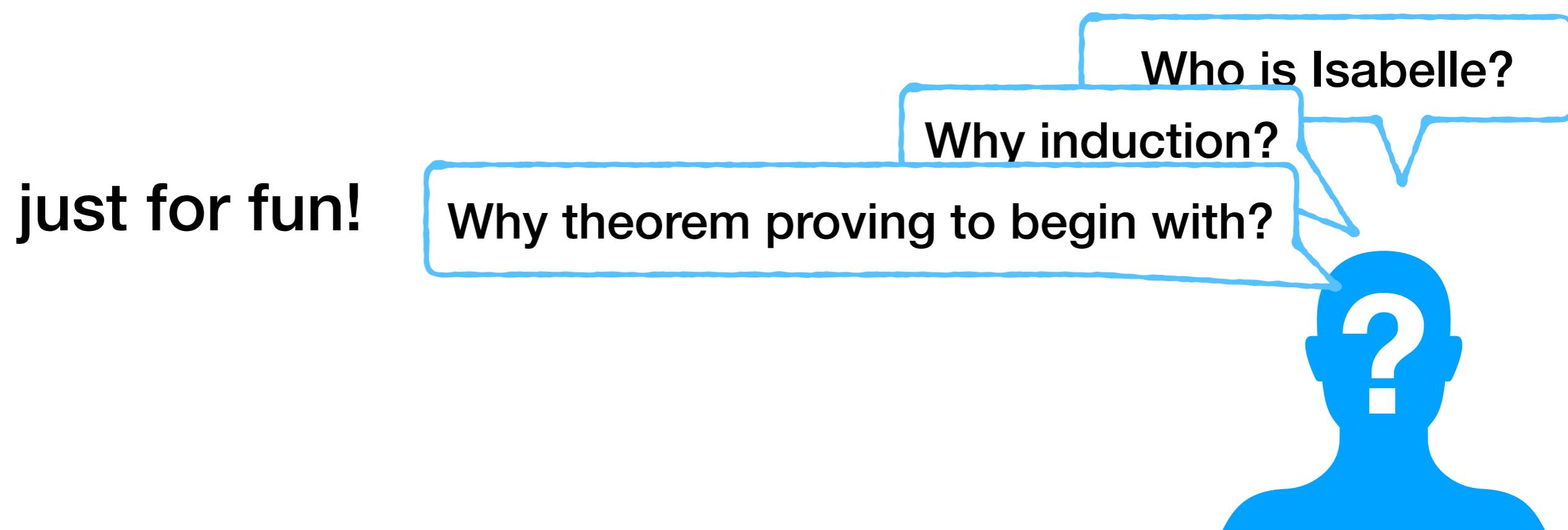
LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

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



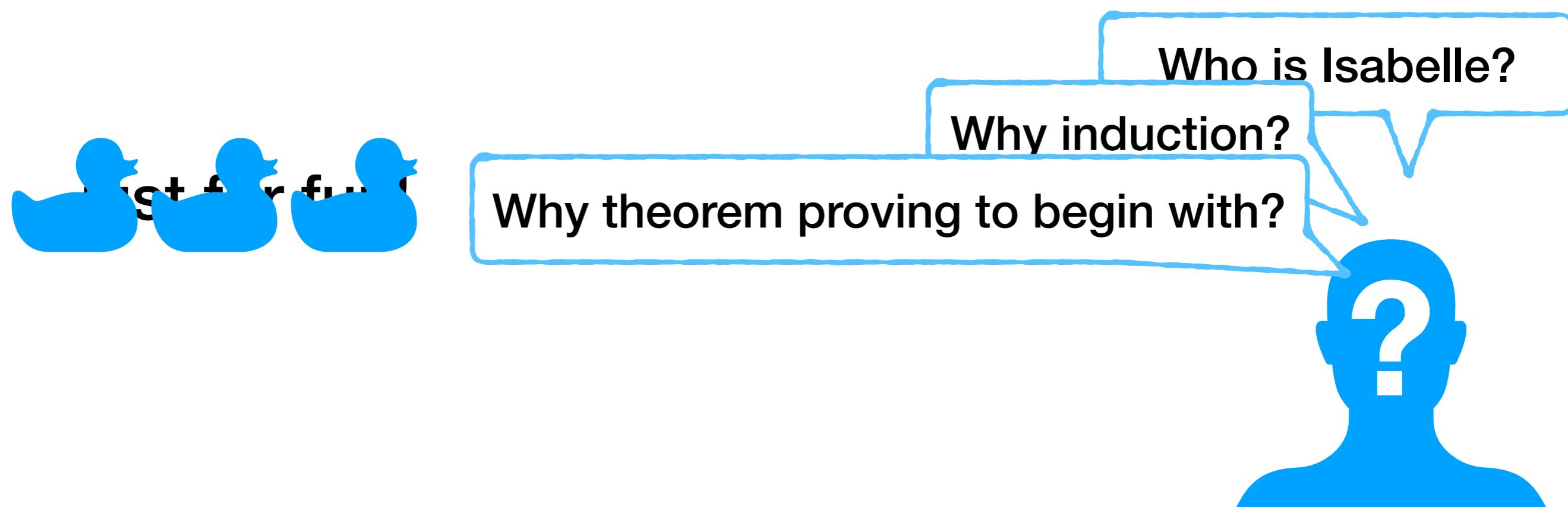
LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

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



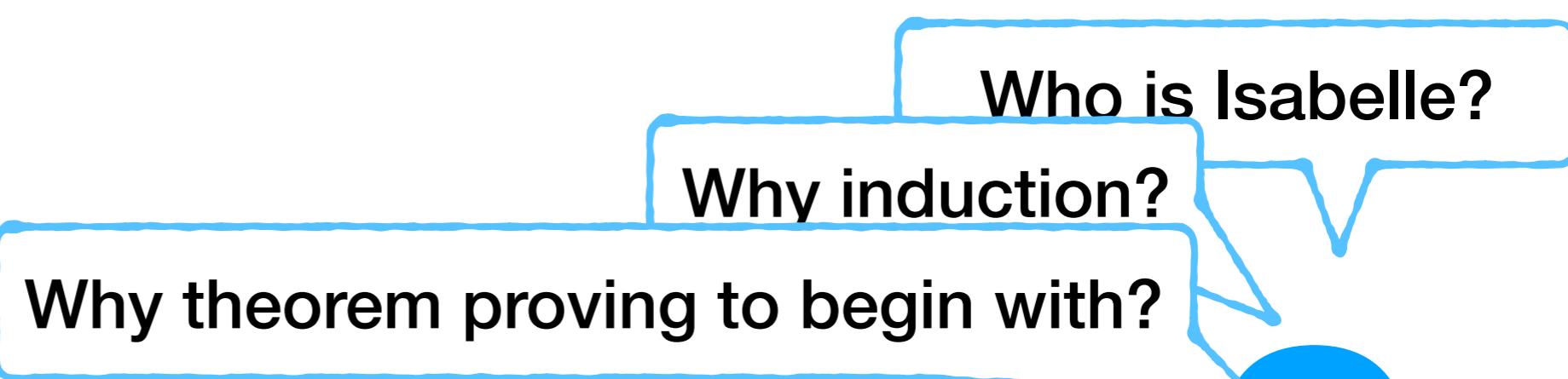
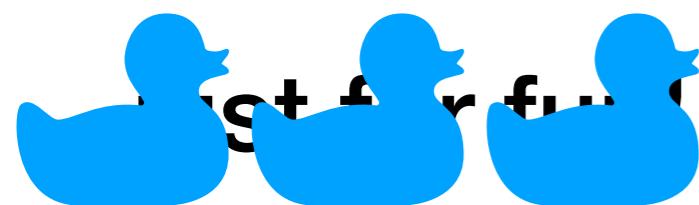
LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

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

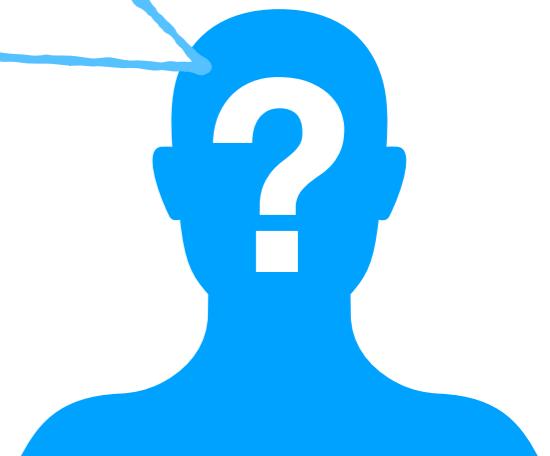


LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

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

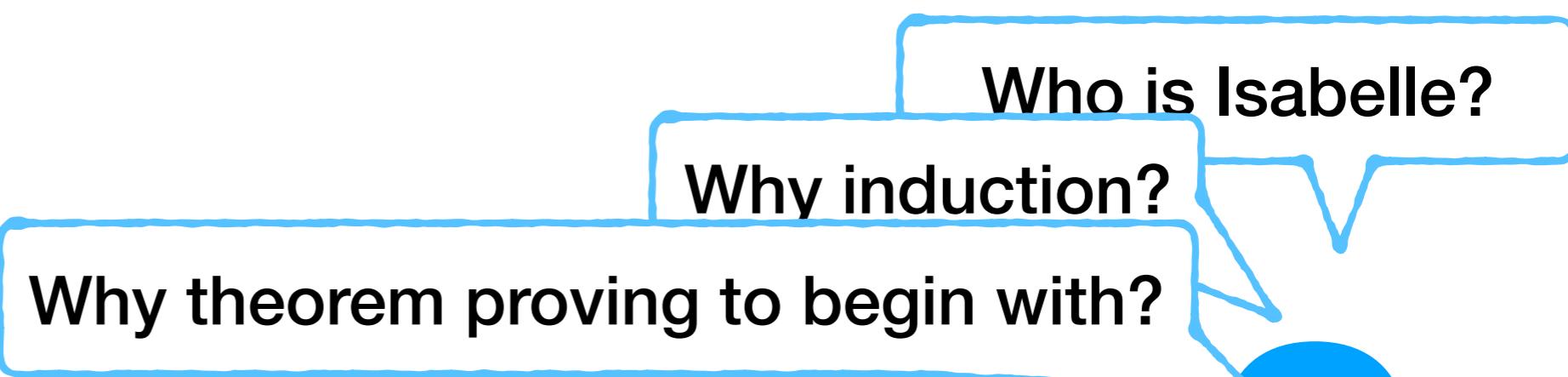
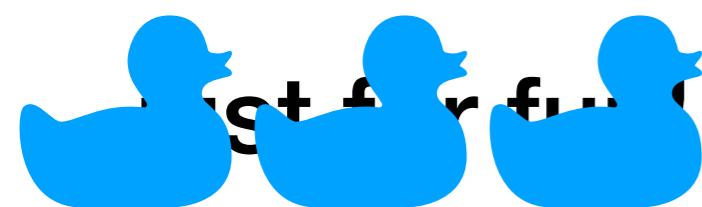


for complete formal verification!



LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

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

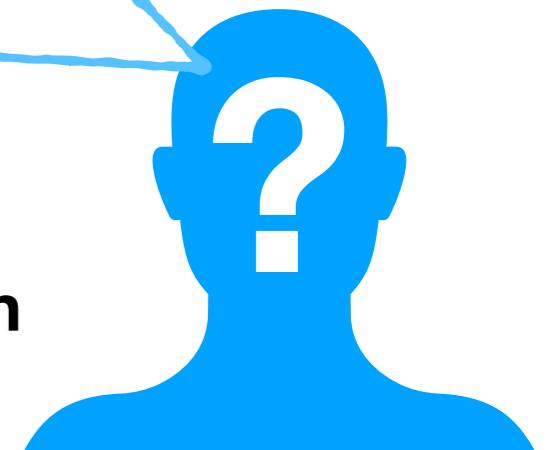


for complete formal verification!

developer

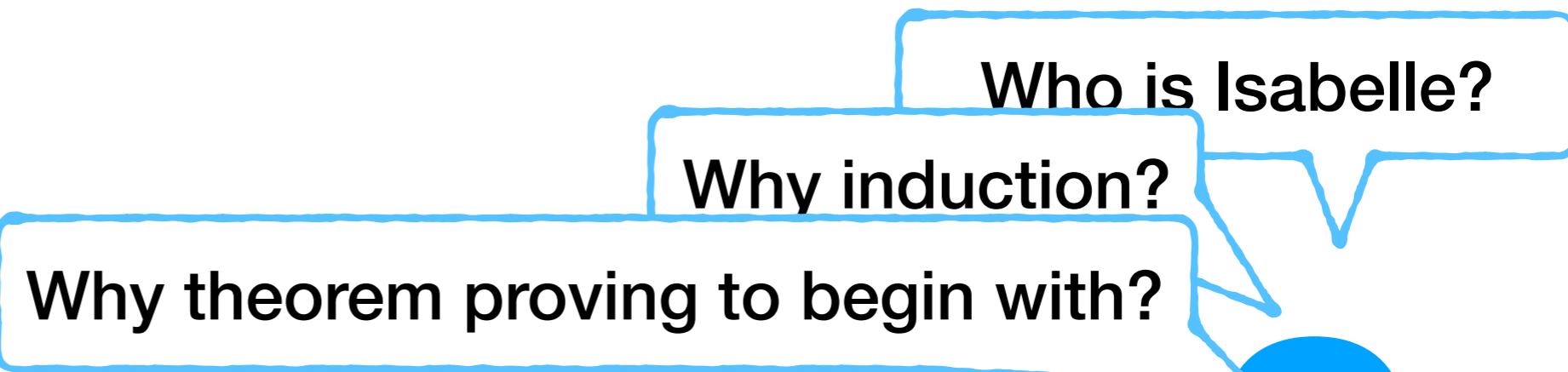
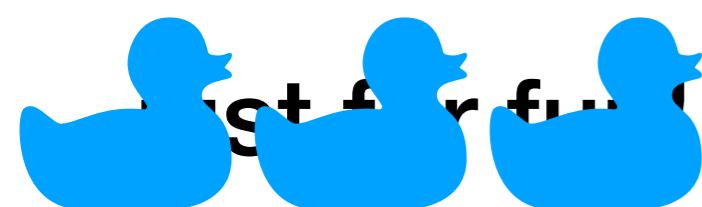
proof assistant / ITP

implementation



LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

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

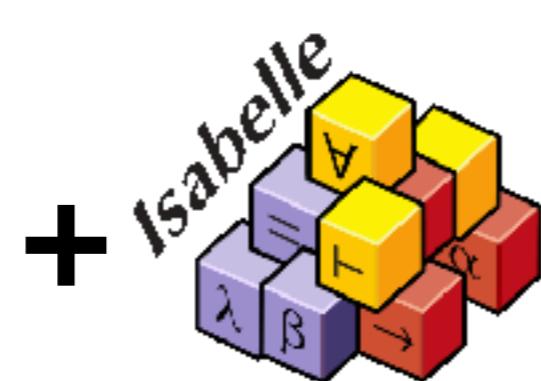


for complete formal verification!

developer



proof assistant / ITP



Gewin Klein et. al

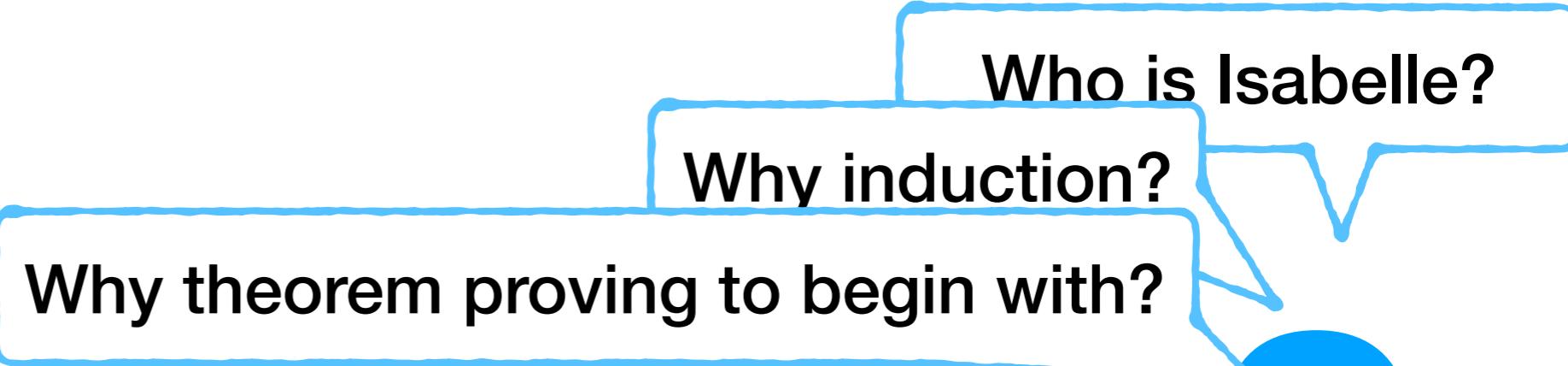
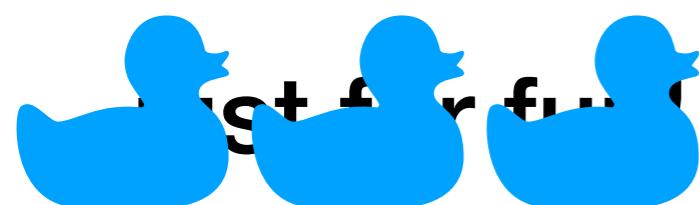
implementation



verified micro kernel,
seL4

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

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

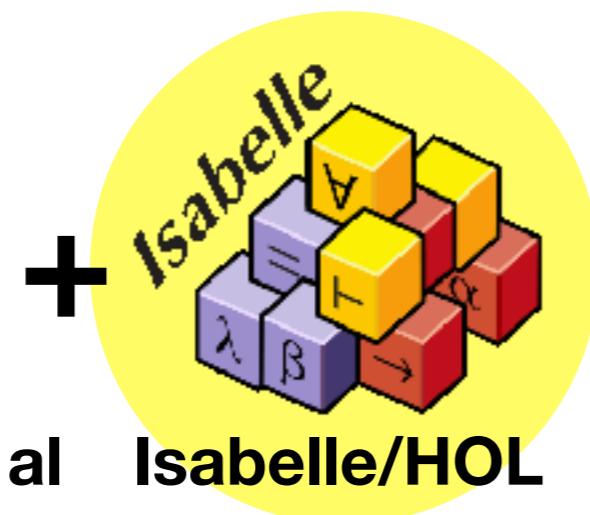


for complete formal verification!

developer



proof assistant / ITP



Gewin Klein et. al

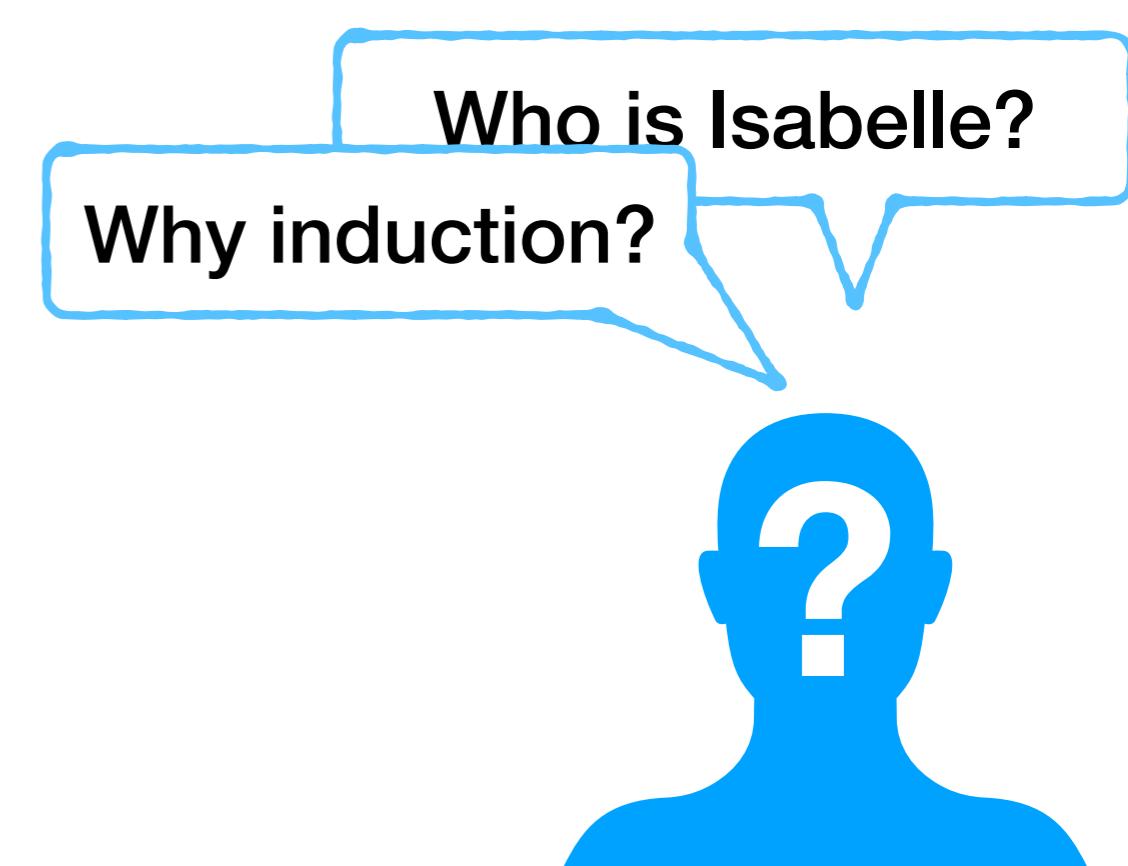
implementation



verified micro kernel,
seL4

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

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



LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

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

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



Who is Isabelle?
Why induction?



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

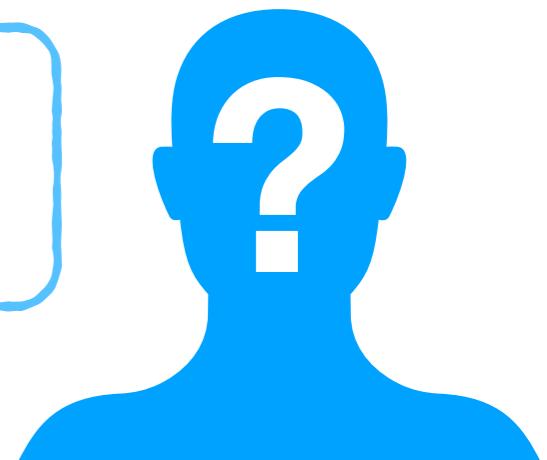
LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

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

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



Who is Isabelle?
Why induction?



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

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

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

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



Who is Isabelle?
Why induction?

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



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

Prof. Bernhard Gramlich

<https://www.logic.at/staff/gramlich/>

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

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

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

Who is Isabelle?
Why induction?

Why theorem proving to be

Challenge accepted!

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

spectacular breakthroughs
unrealistic, in view of the problems and the inherent inductive theorem pro



Prof. Bernhard Gramlich

<https://www.logic.at/staff/gramlich/>

Yutaka Ng
yutakang

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

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

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

The time has come!

Who is Isabelle?
Why induction?
Challenge accepted!

substantial
time.



spectacular breakthroughs are unrealistic, in view of the complexity of the problems and the inherent difficulties of solving inductive theorem pro-



Prof. Bernhard Gramlich

<https://www.logic.at/staff/gramlich/>

Yutaka Ng
[yutakang](https://twitter.com/Yutakang)

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

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

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

Who is Isabelle?

Why induction?

Challenge accepted!

The time has
come!

...or is coming
soon.

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



Prof. Bernhard Gramlich

<https://www.logic.at/staff/gramlich/>

Yutaka Ng
[yutakang](https://twitter.com/YutakangE)

<https://twitter.com/YutakangE>

<https://tinyurl.com/up7zhyk>

<https://twitter.com/YutakangE>

<https://tinyurl.com/up7zhyk>

What I did before APLAS2020...

<https://twitter.com/YutakangE>

<https://tinyurl.com/yup7zhyk>

What I did before APLAS2020...

DEMO!

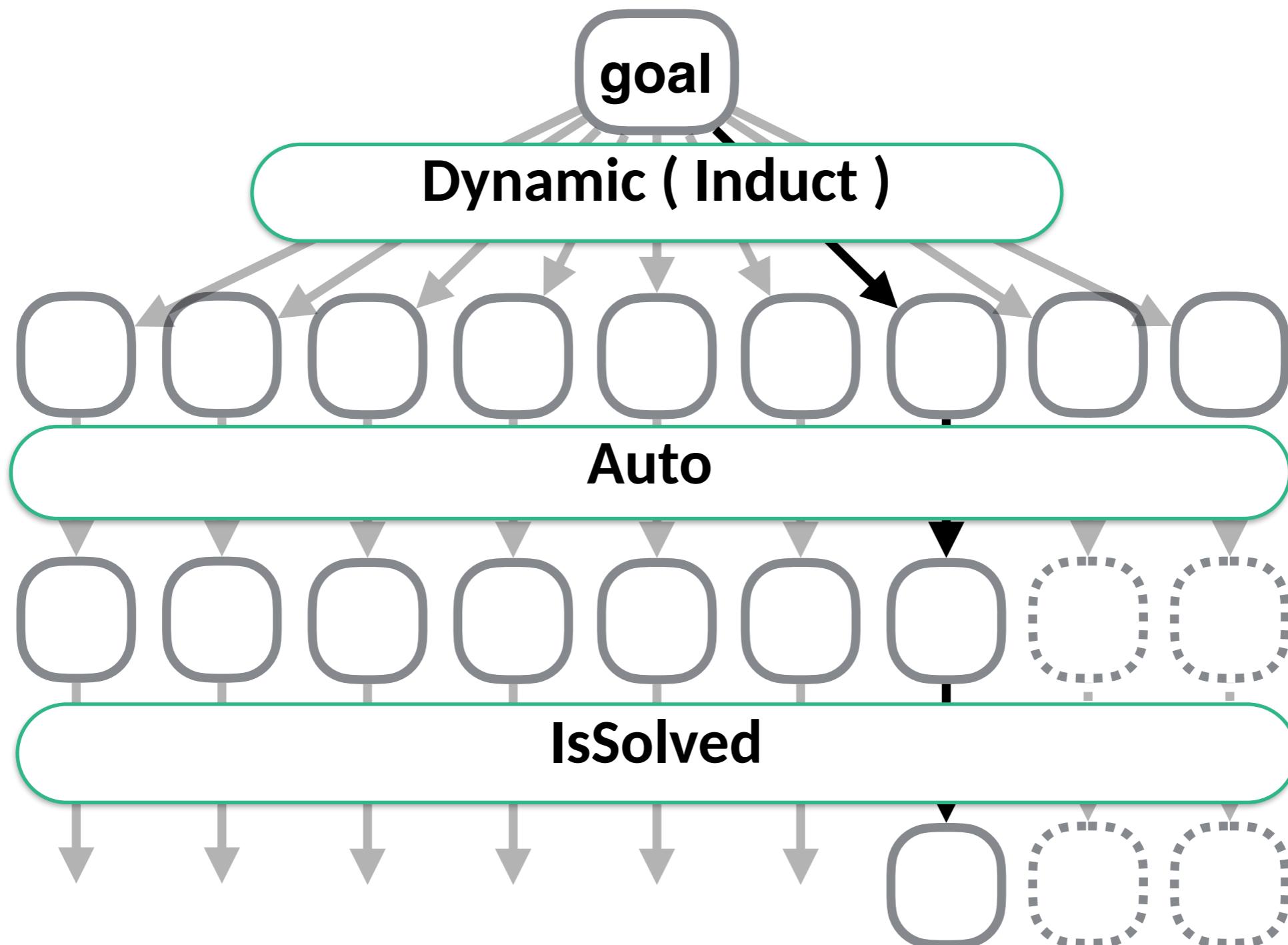
<https://twitter.com/YutakangE>

<https://tinyurl.com/up7zhyk>

What I did before APLAS2020...

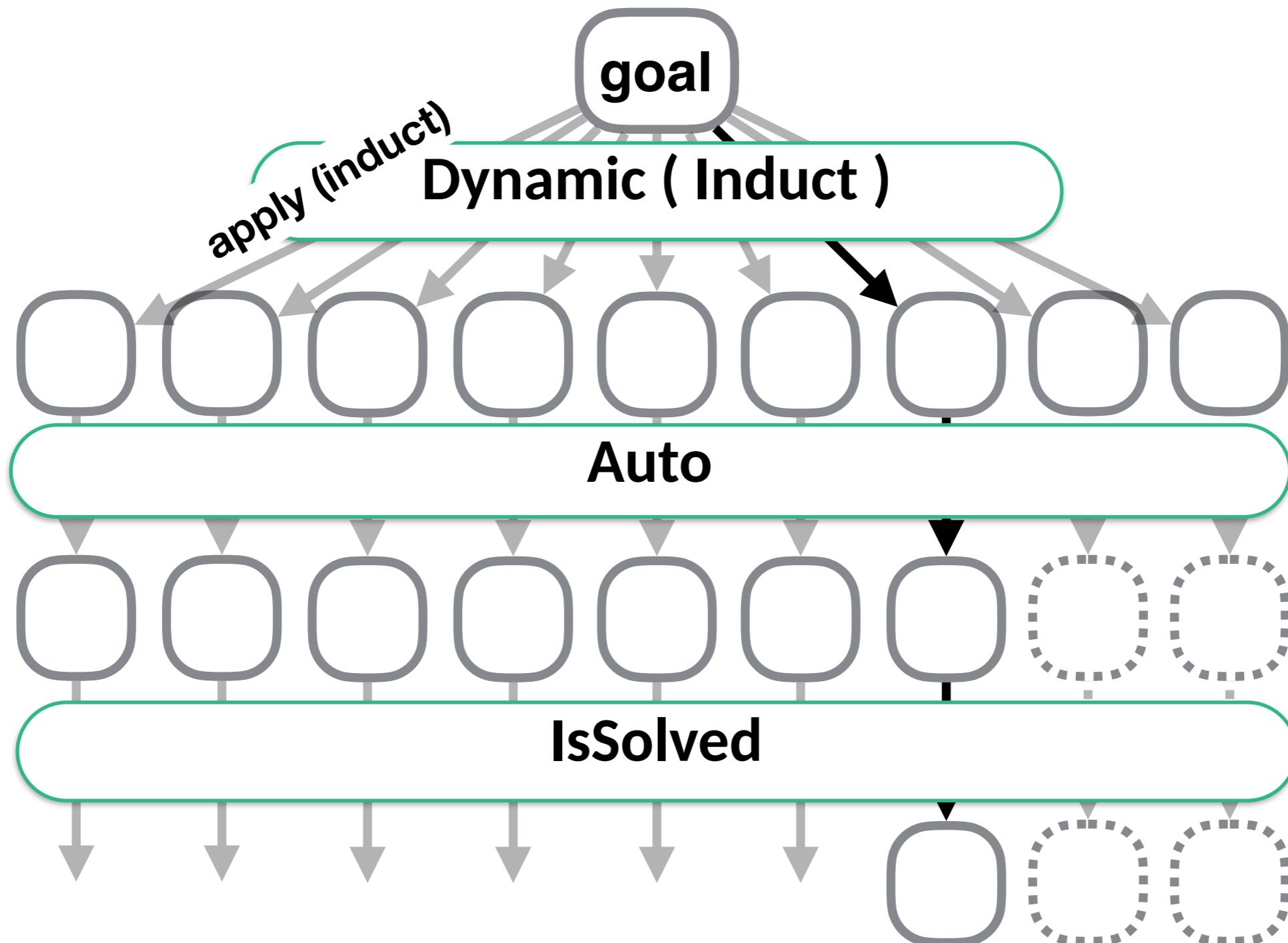
What I did before APLAS2020...

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



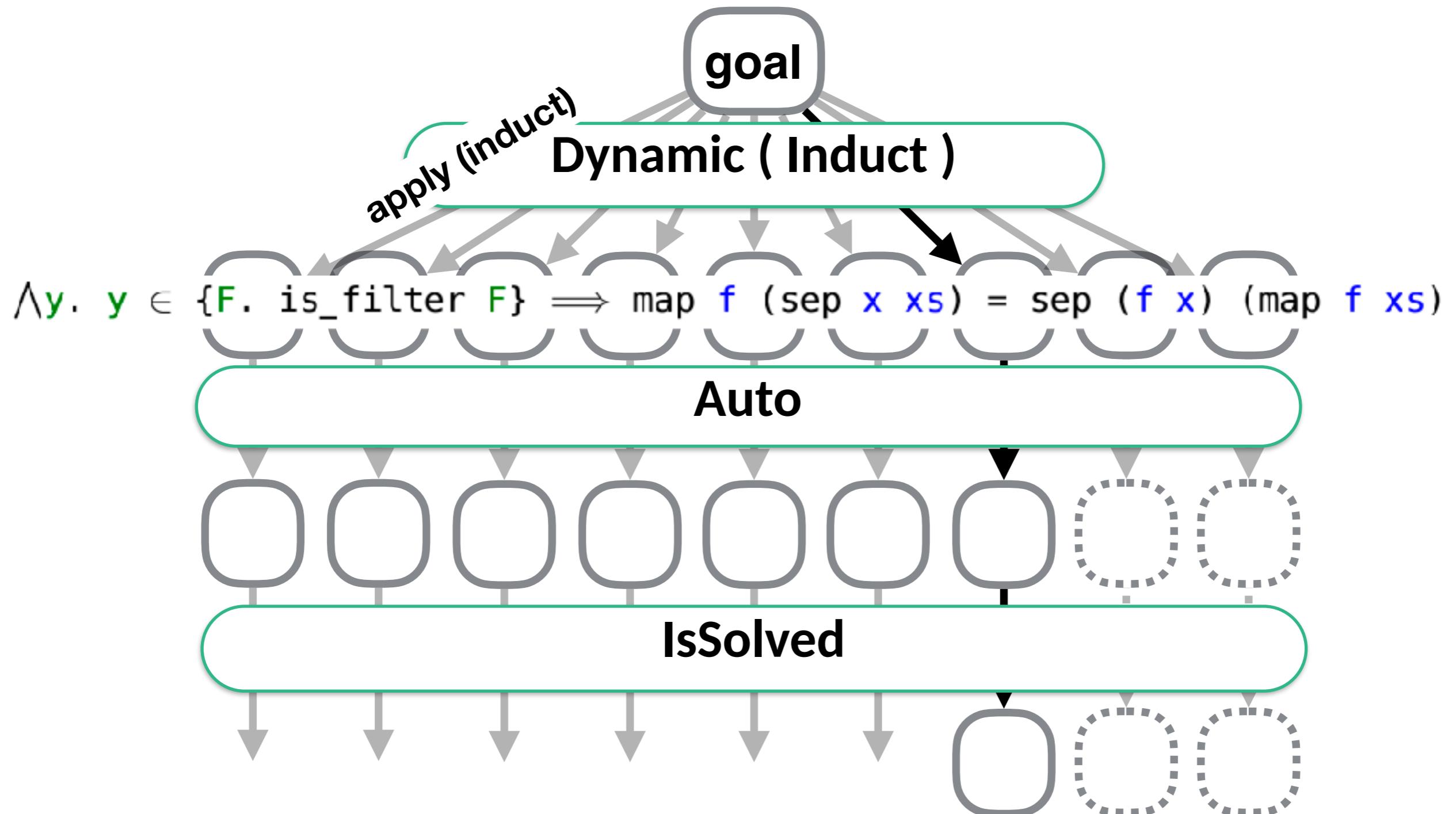
What I did before APLAS2020...

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



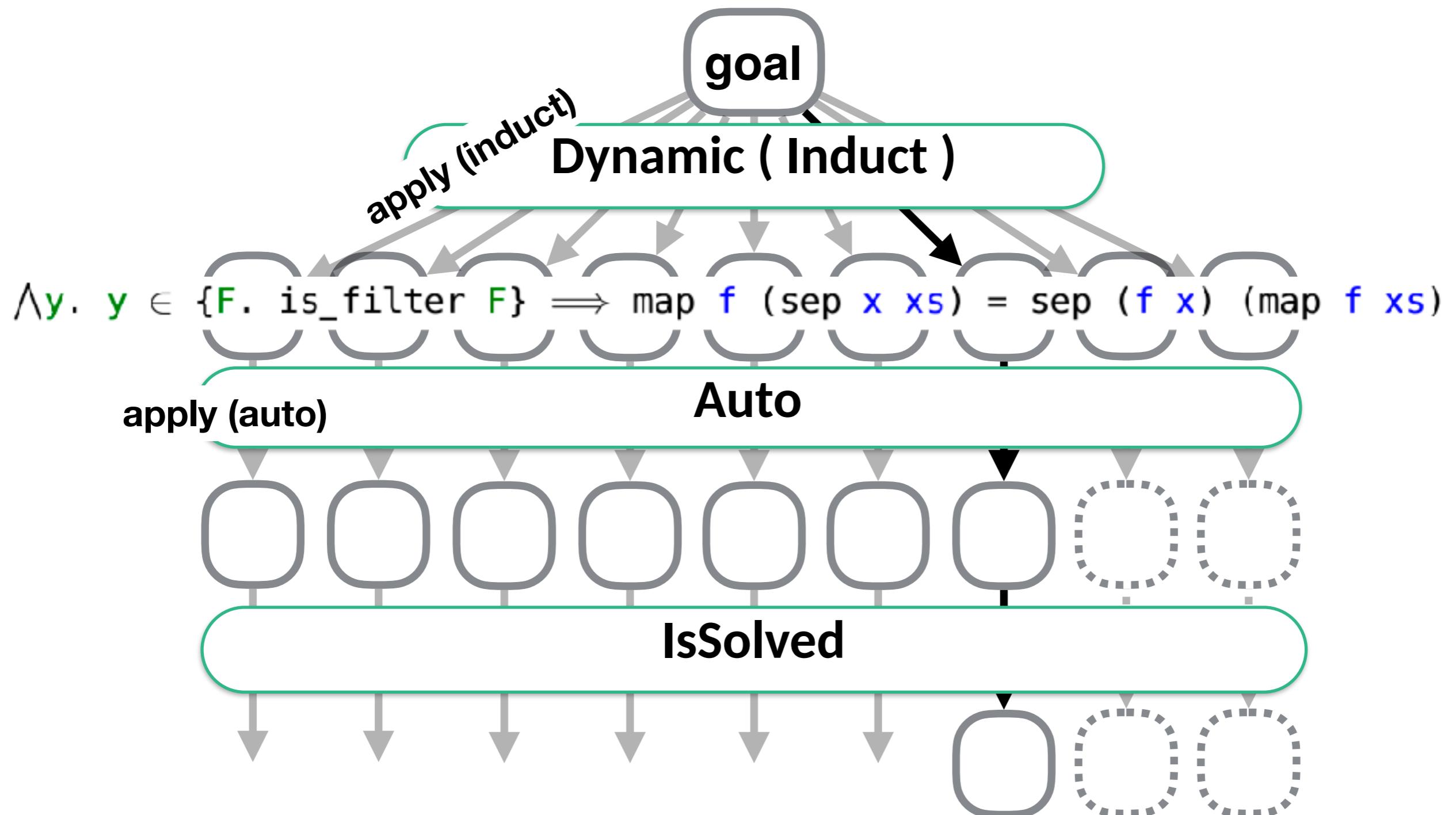
What I did before APLAS2020...

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



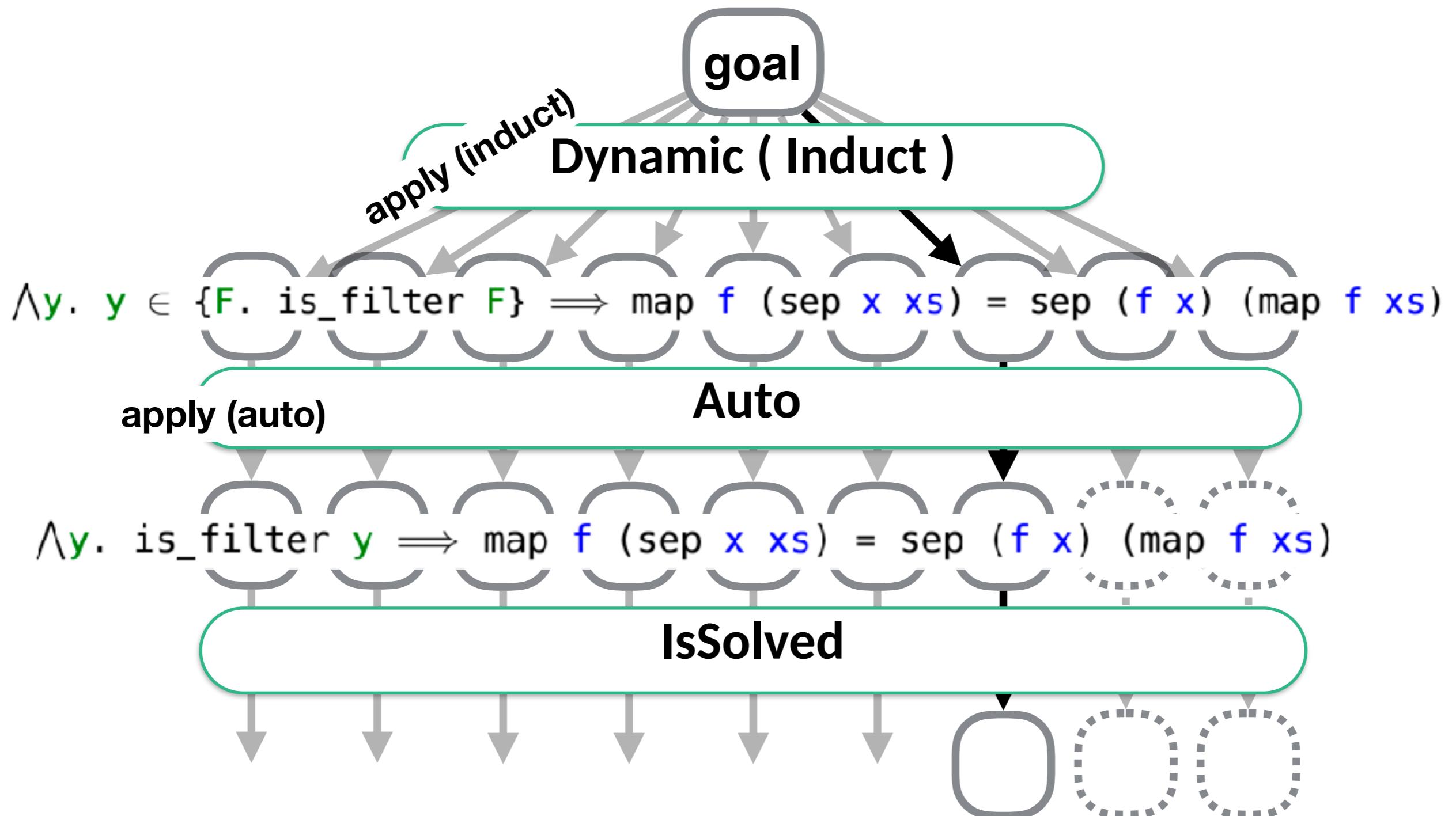
What I did before APLAS2020...

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



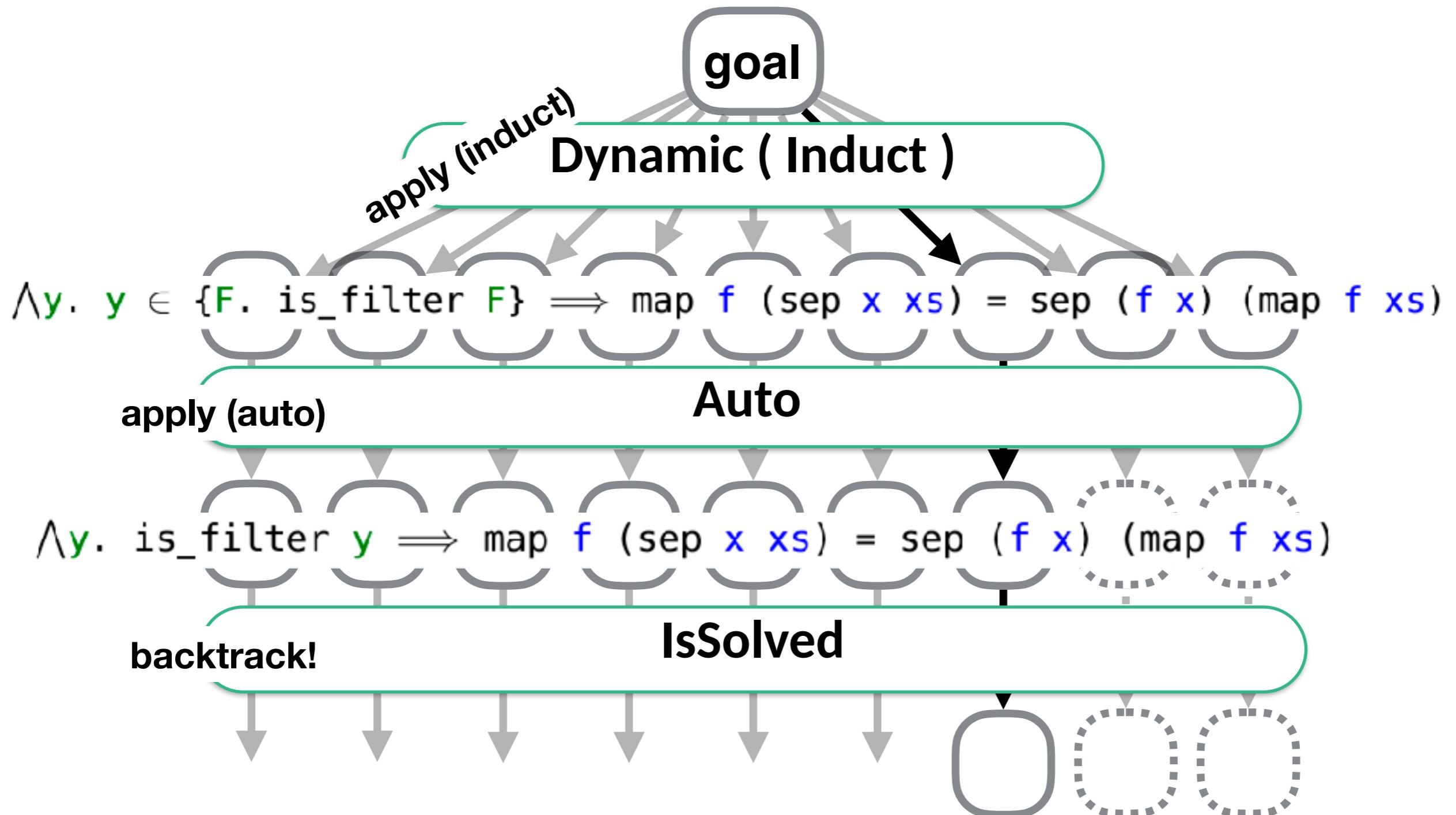
What I did before APLAS2020...

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



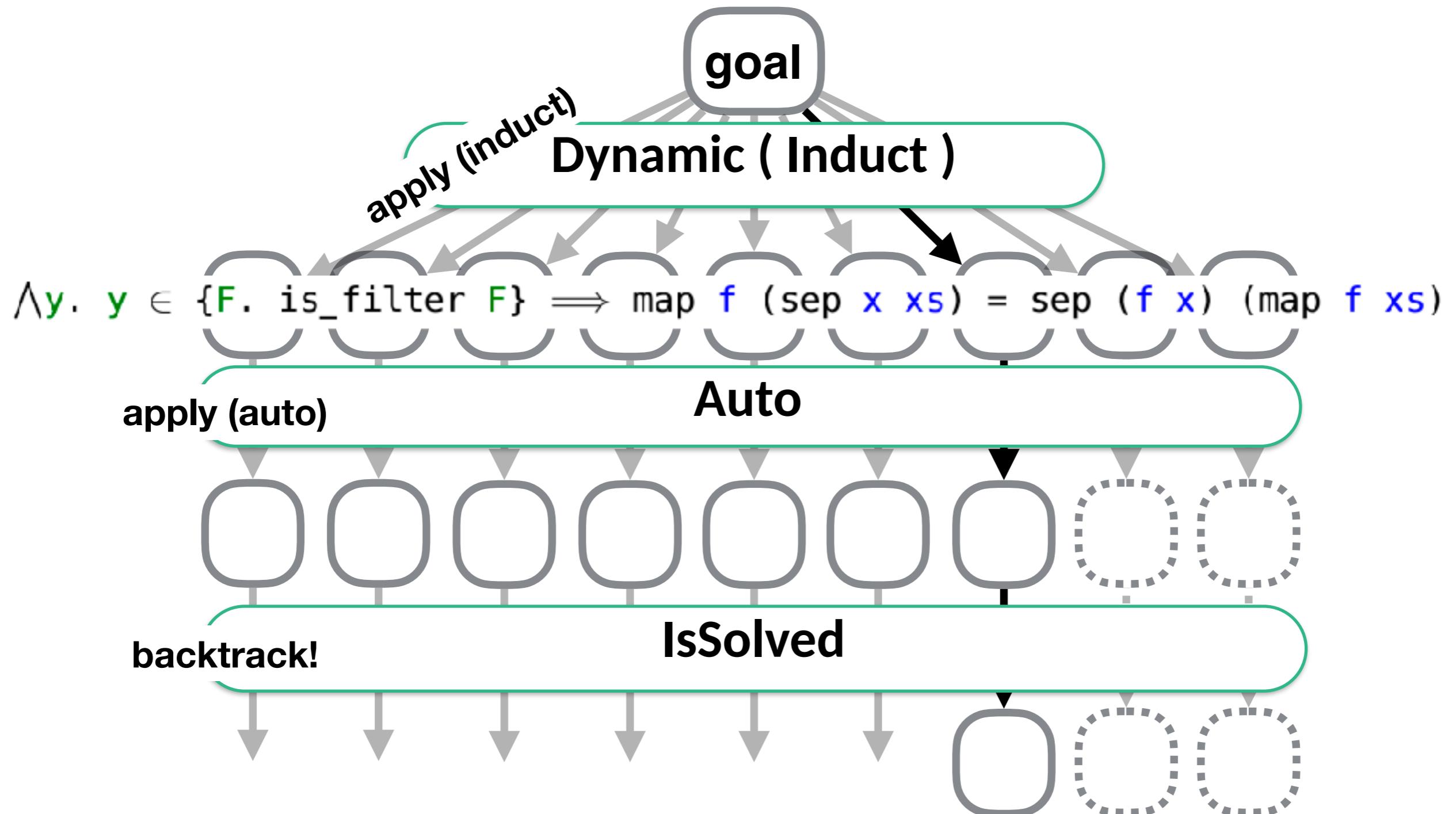
What I did before APLAS2020...

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



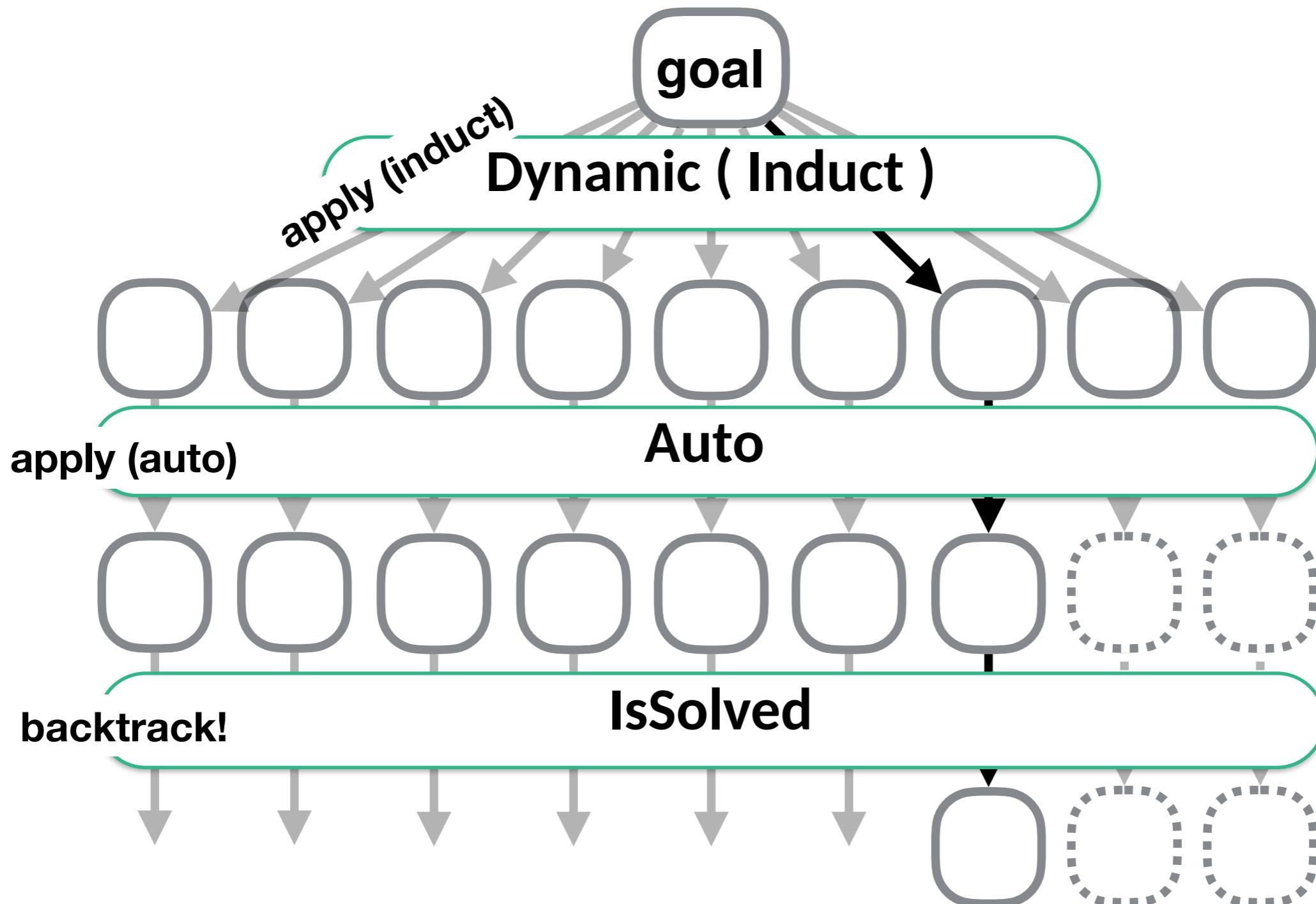
What I did before APLAS2020...

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



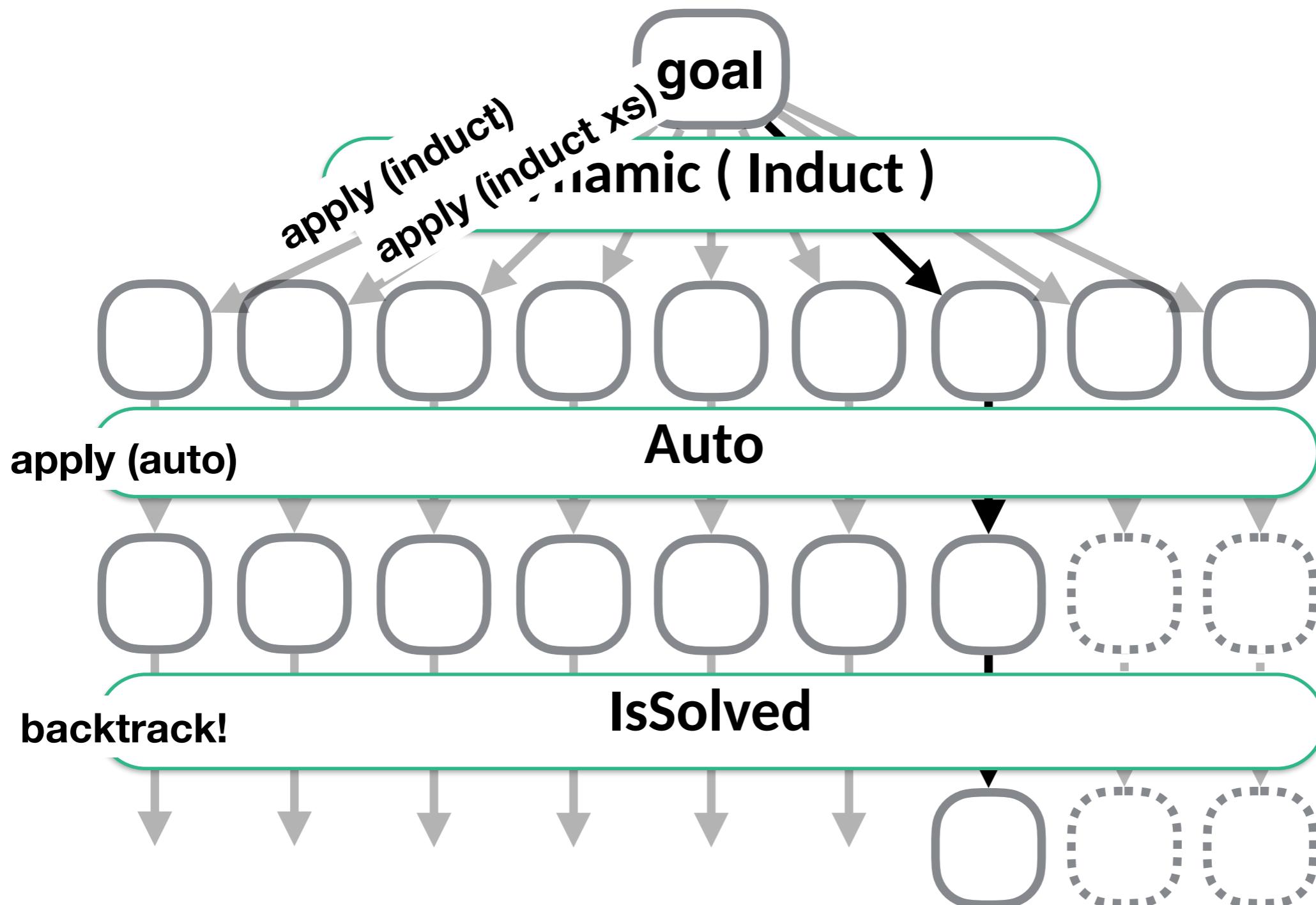
What I did before APLAS2020...

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



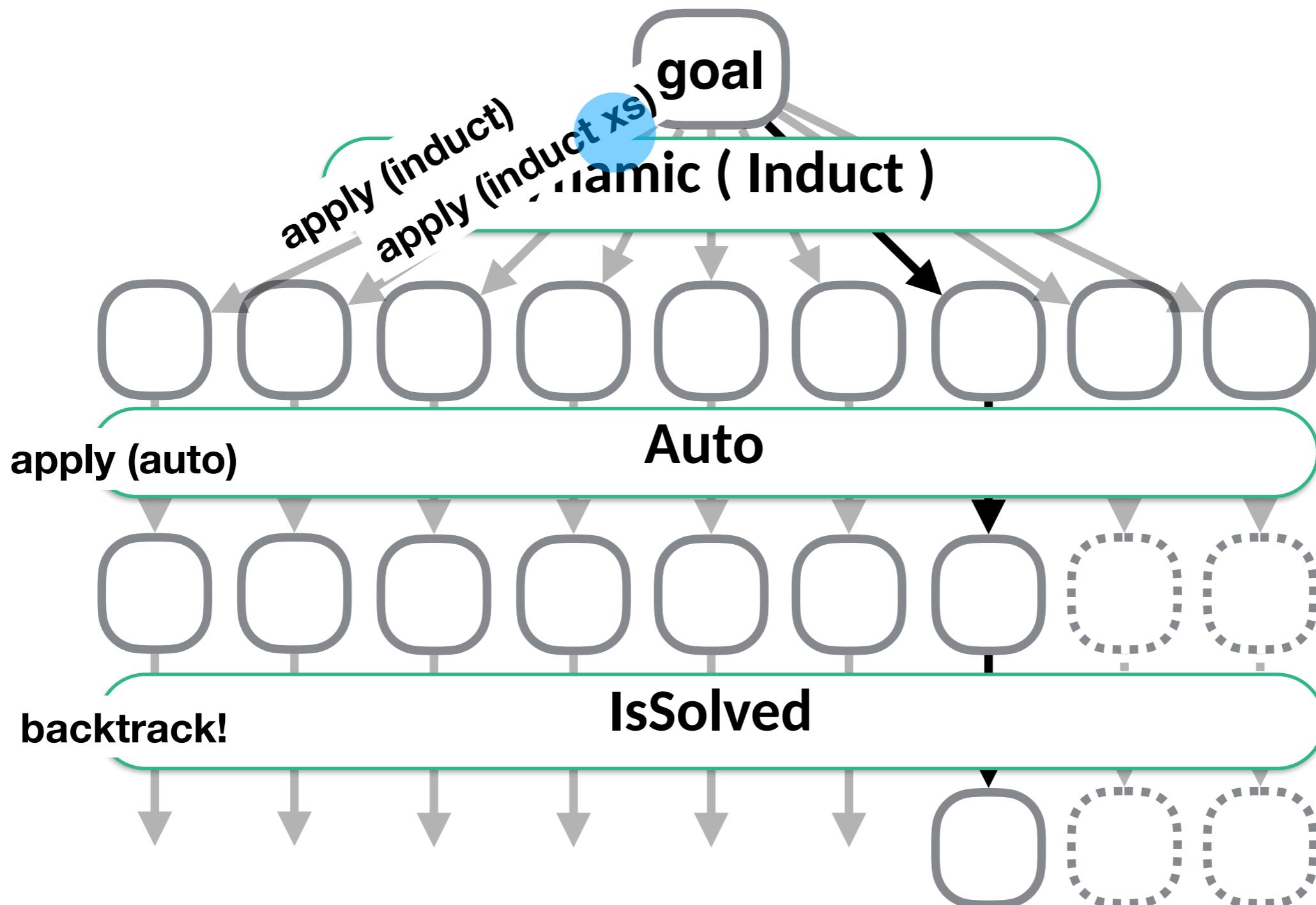
What I did before APLAS2020...

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



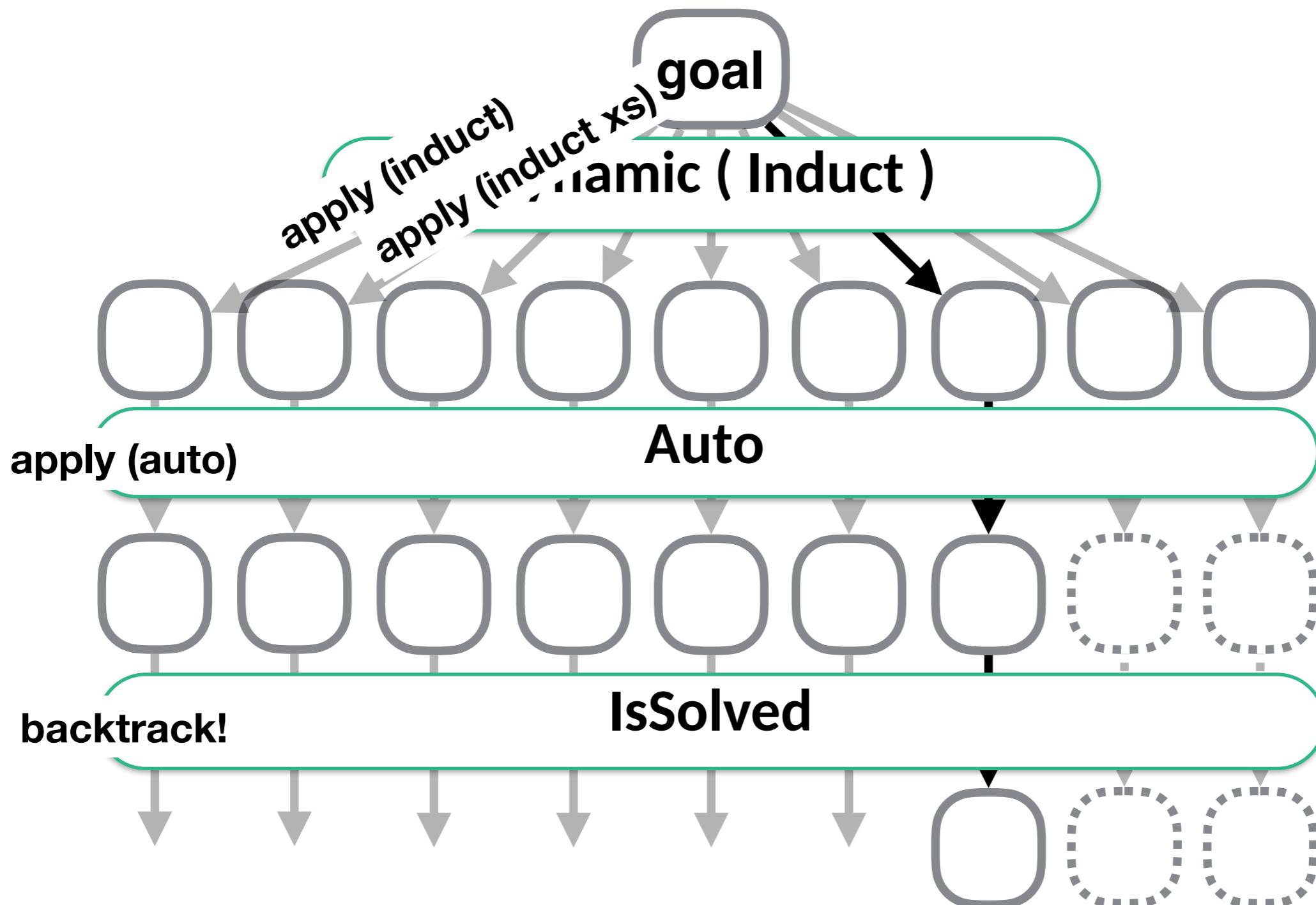
What I did before APLAS2020...

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



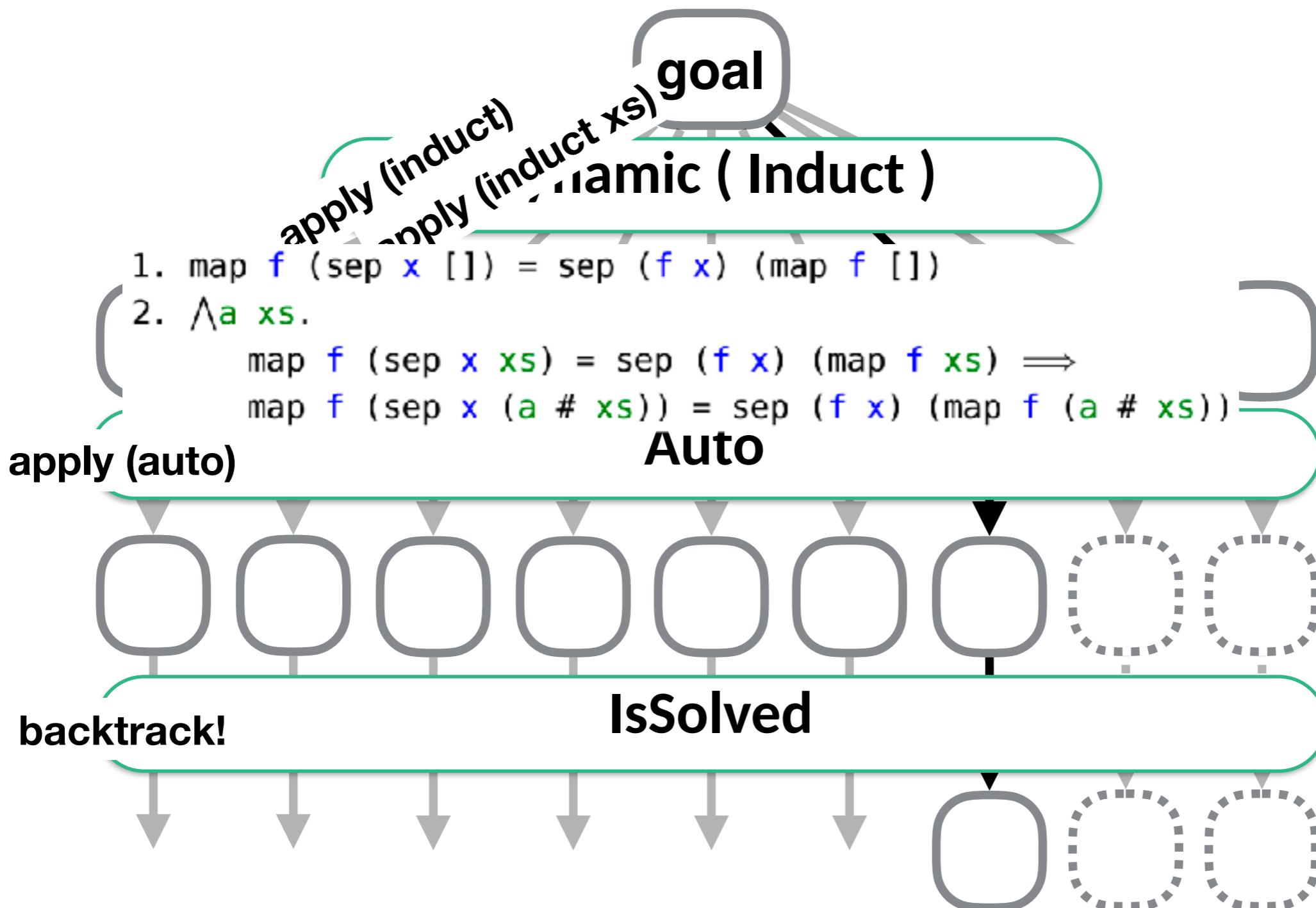
What I did before APLAS2020...

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



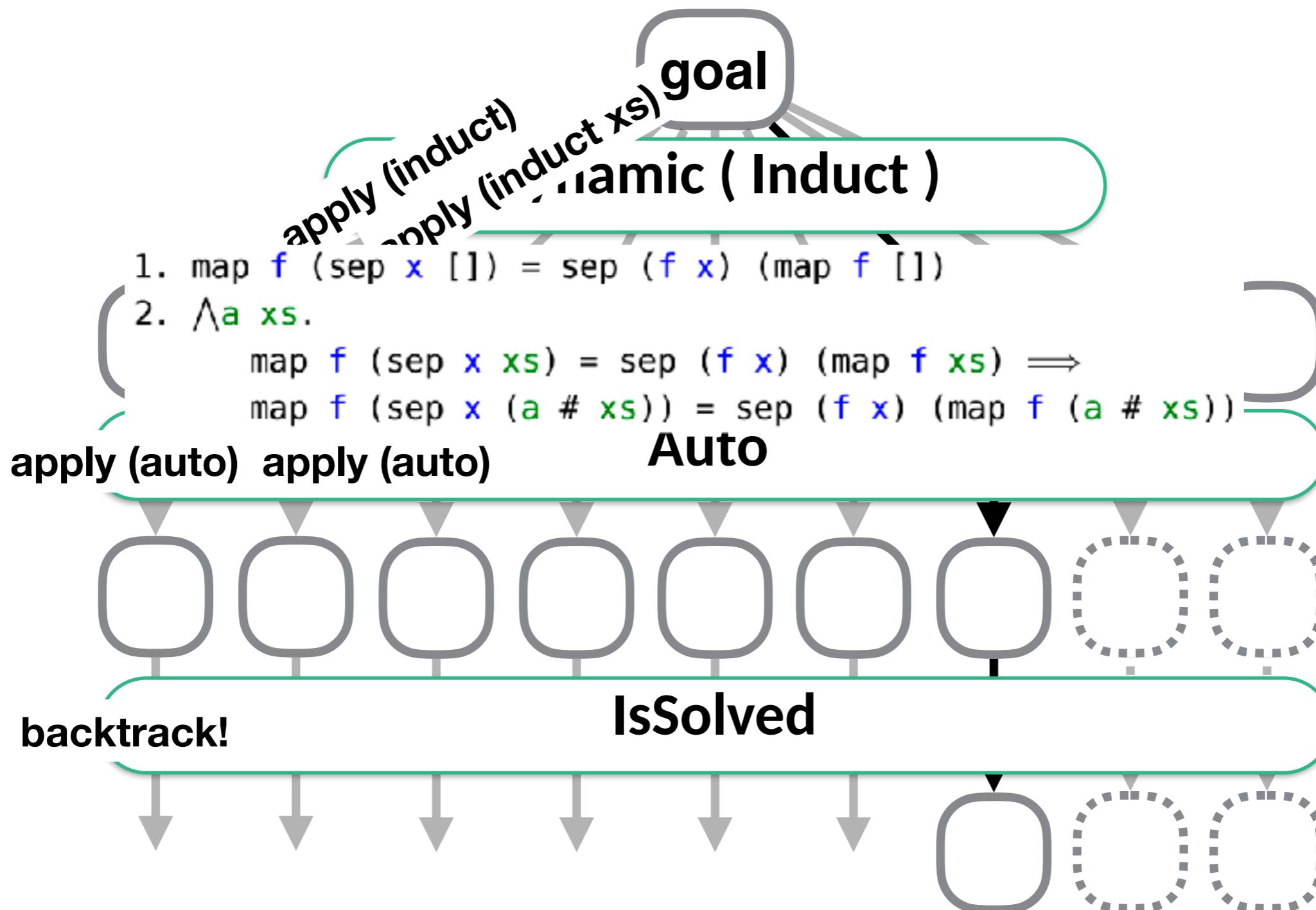
What I did before APLAS2020...

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



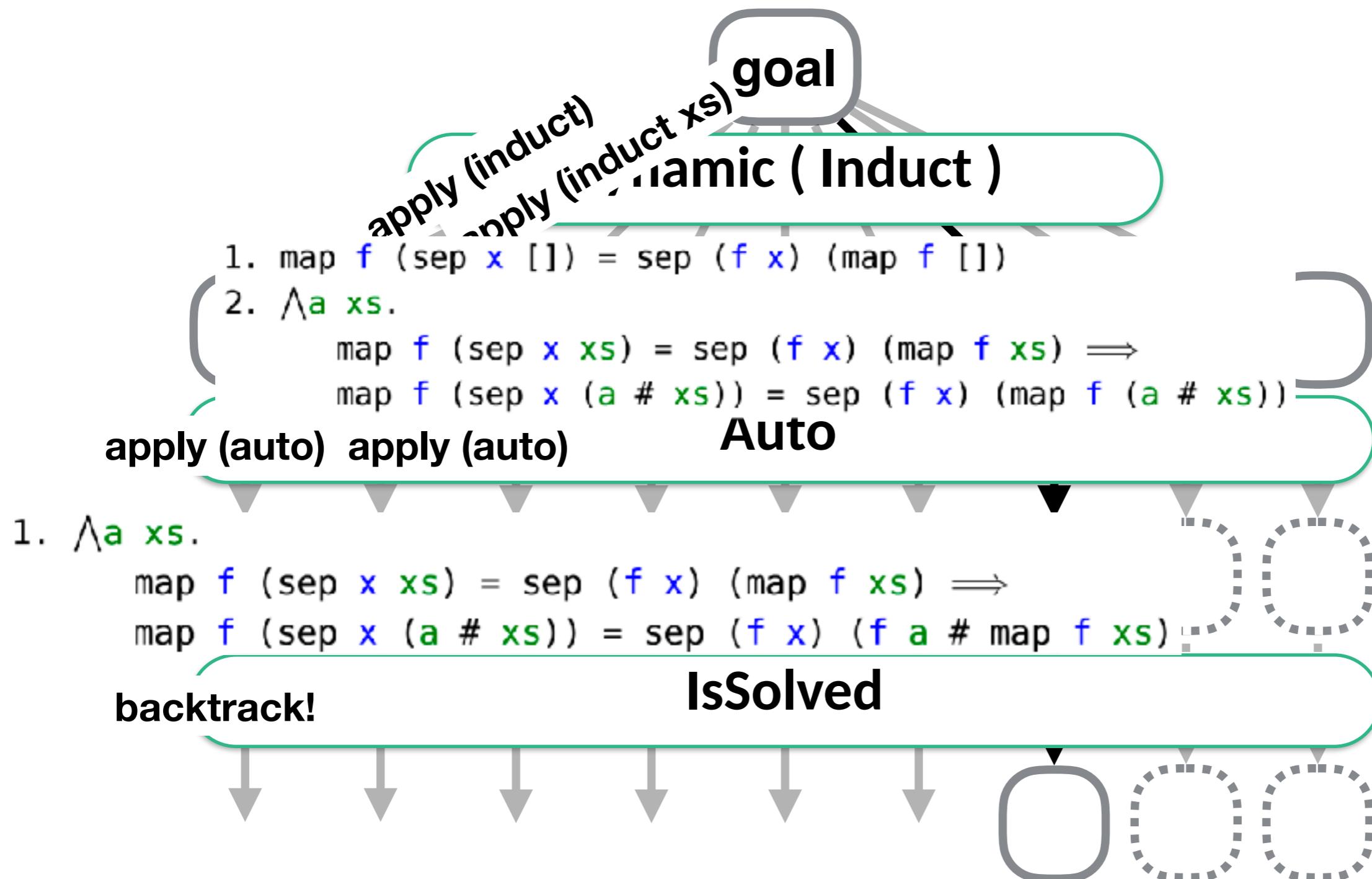
What I did before APLAS2020...

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



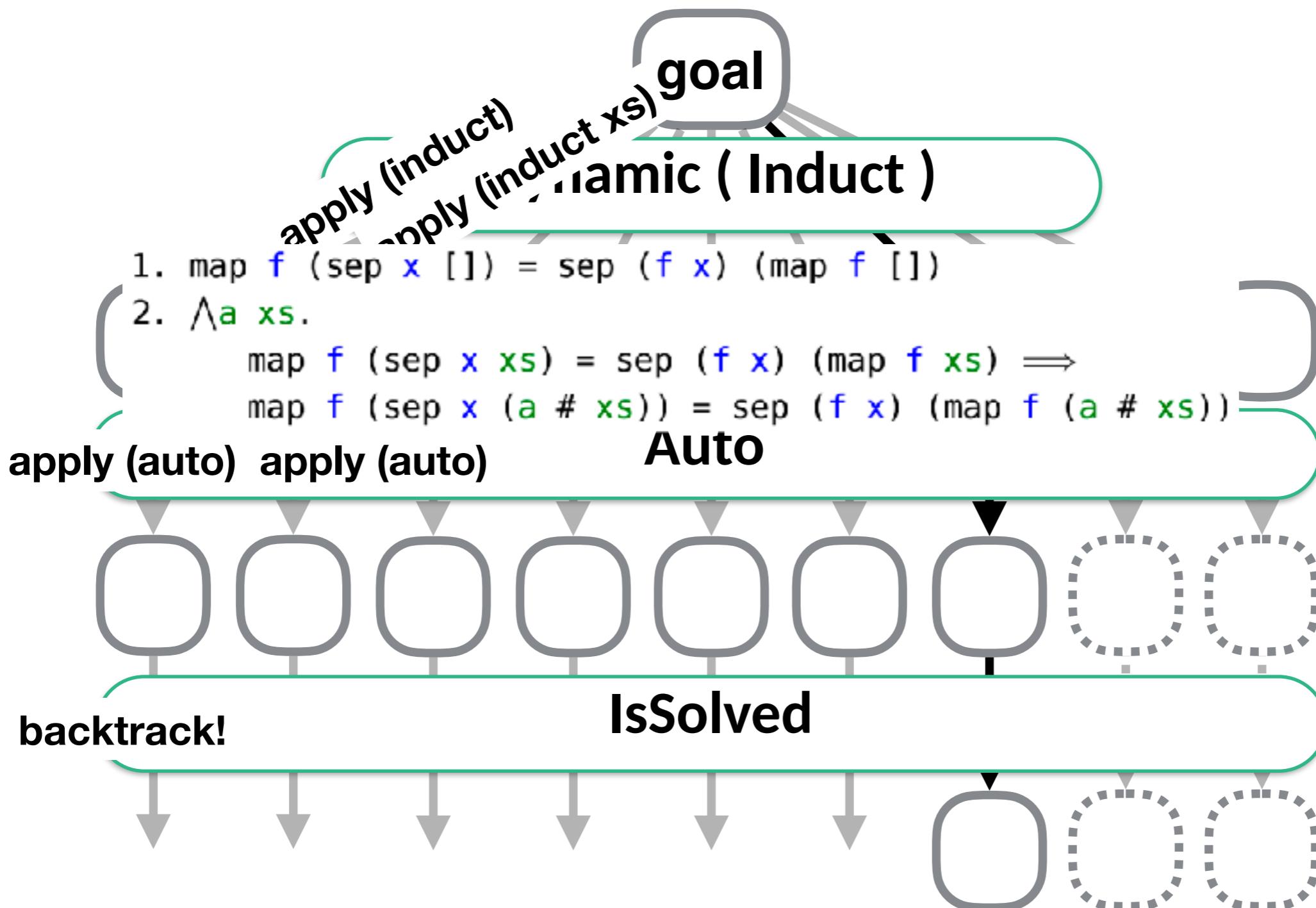
What I did before APLAS2020...

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



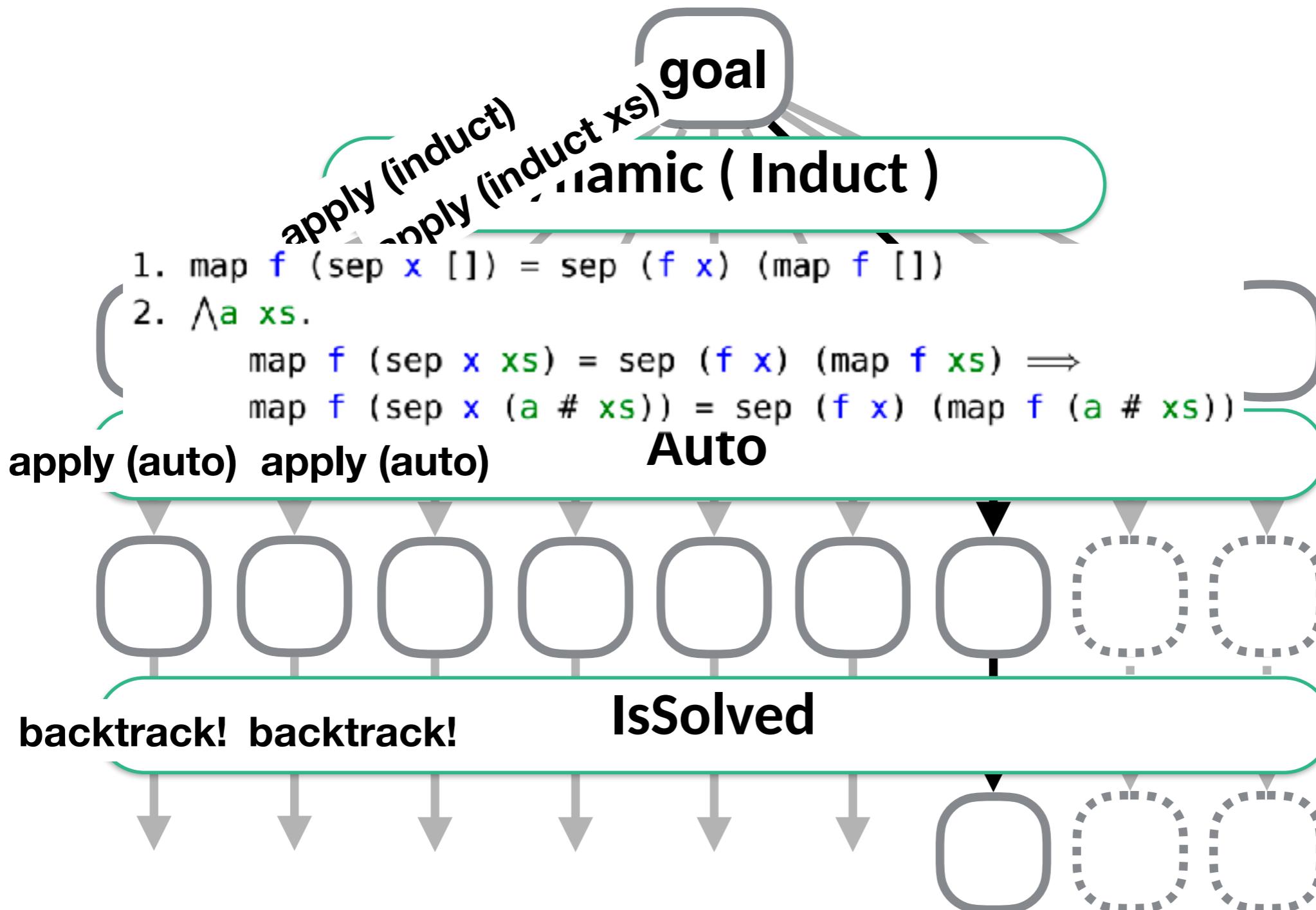
What I did before APLAS2020...

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



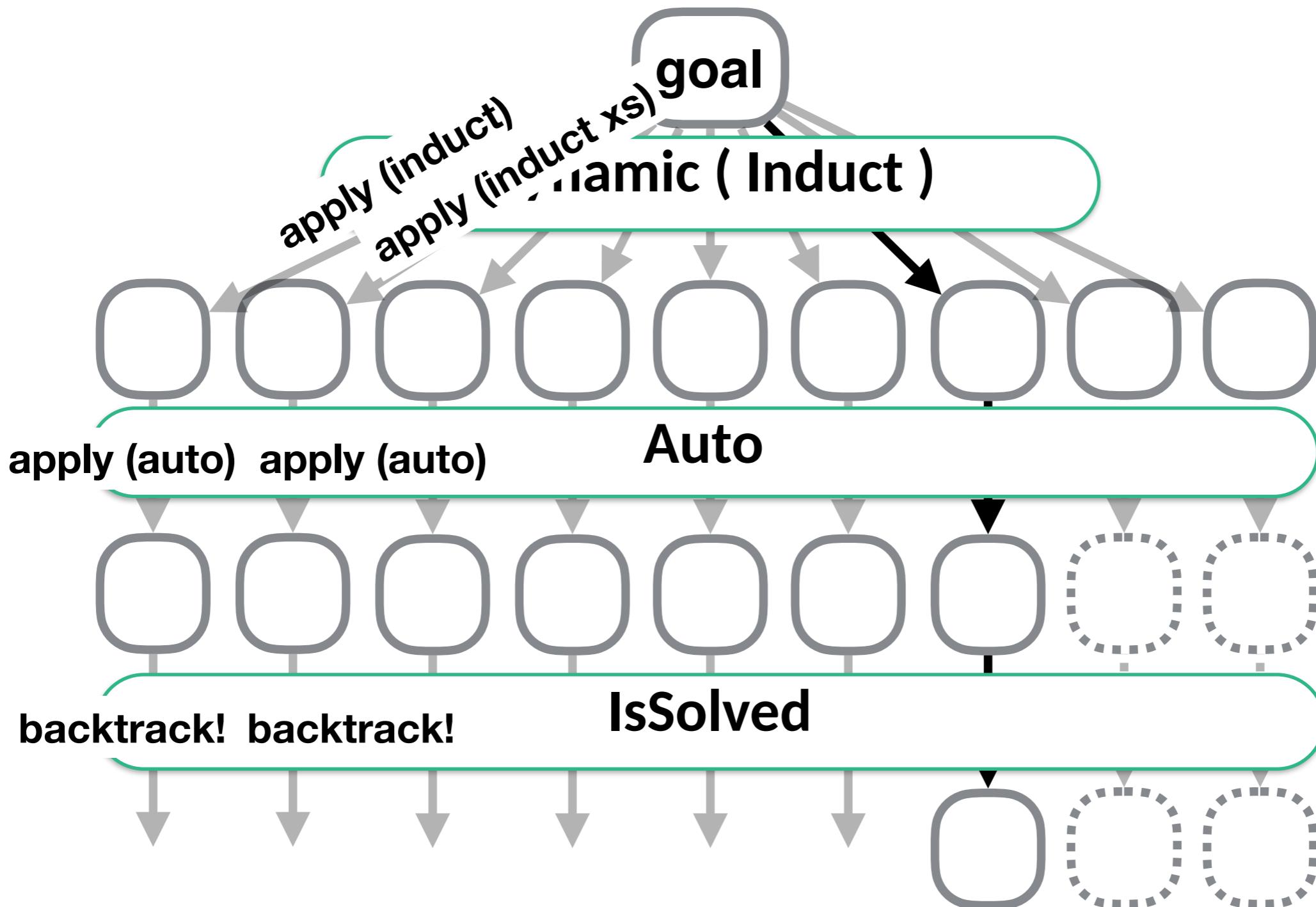
What I did before APLAS2020...

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



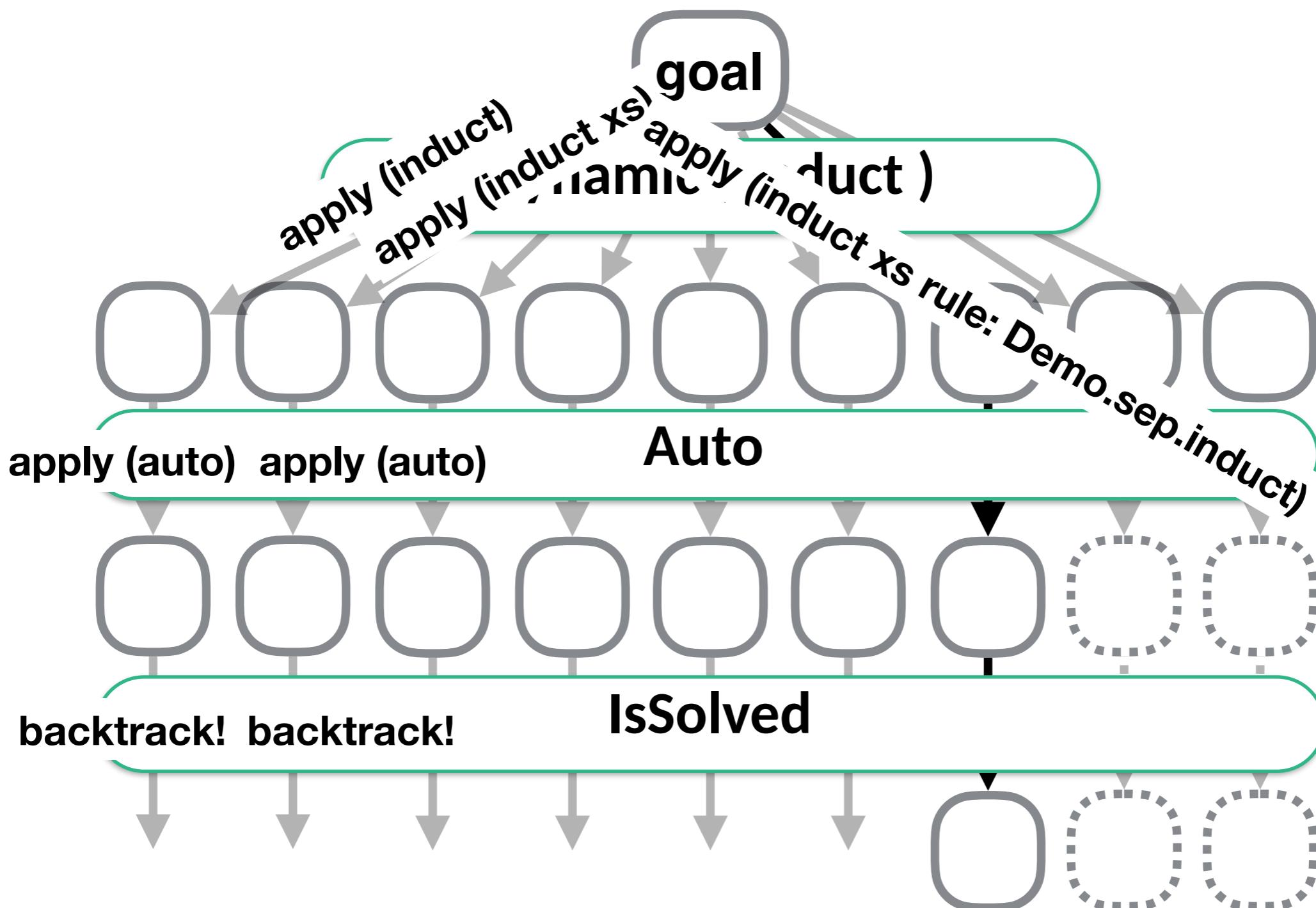
What I did before APLAS2020...

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



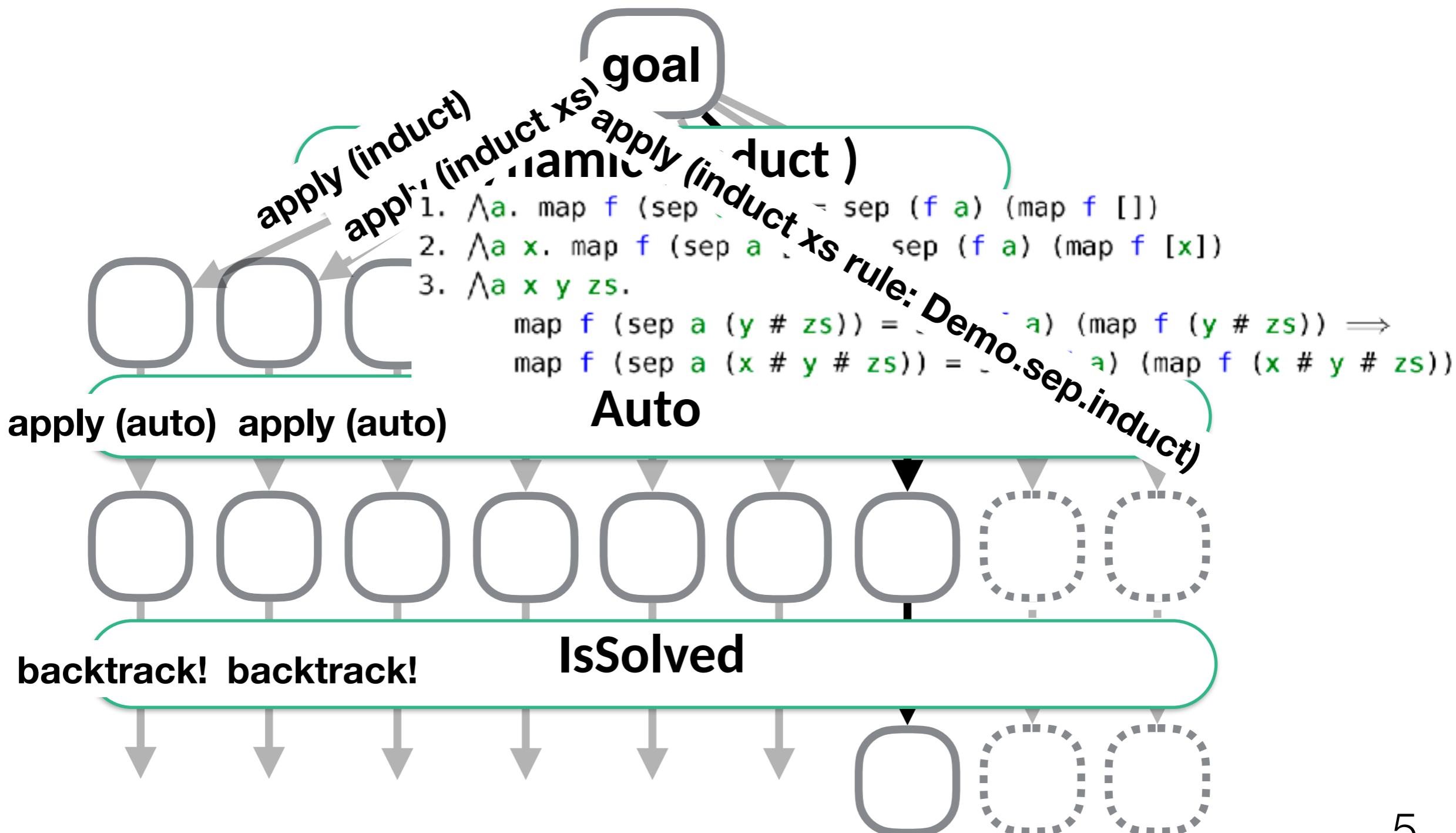
What I did before APLAS2020...

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



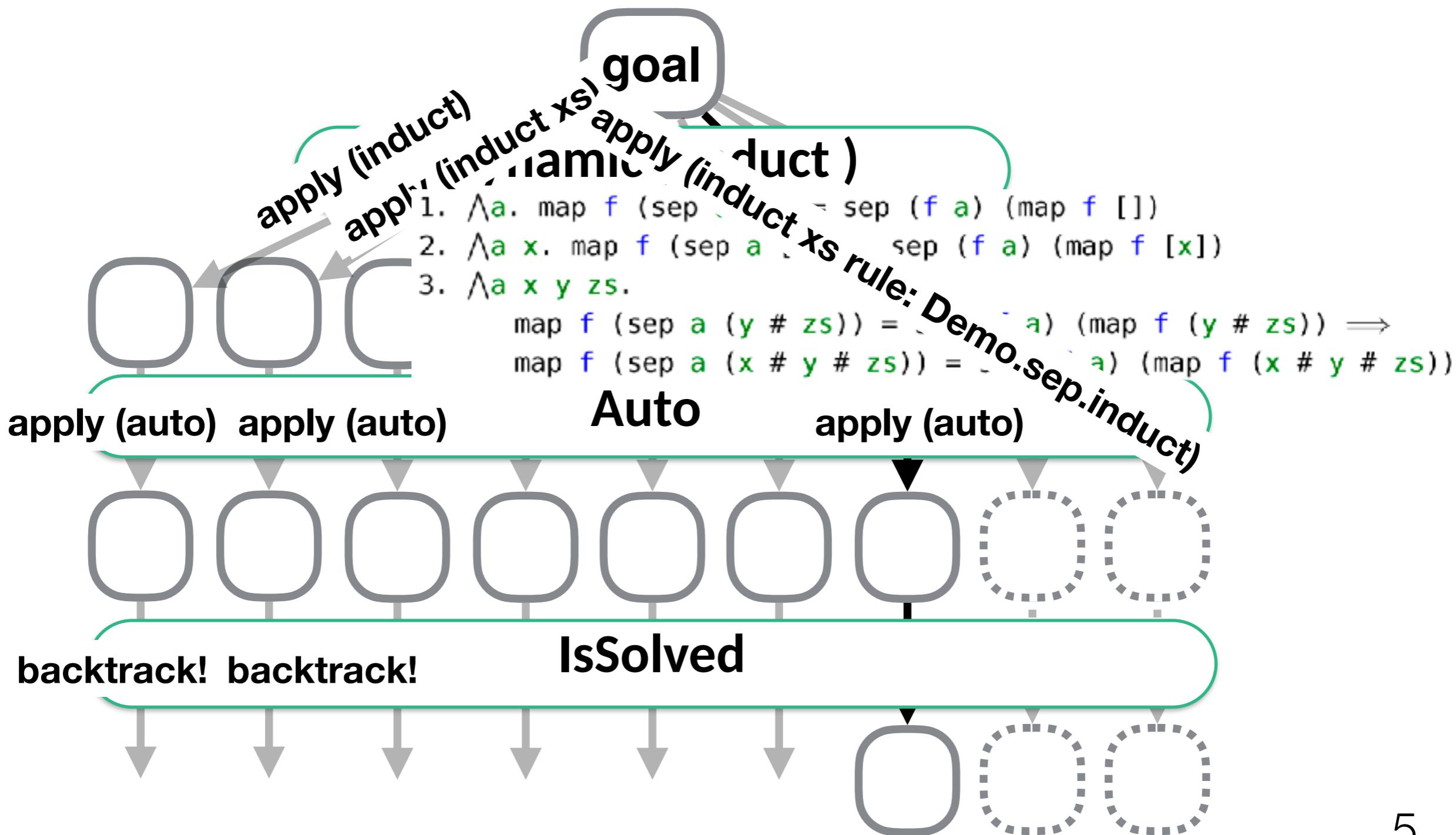
What I did before APLAS2020...

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



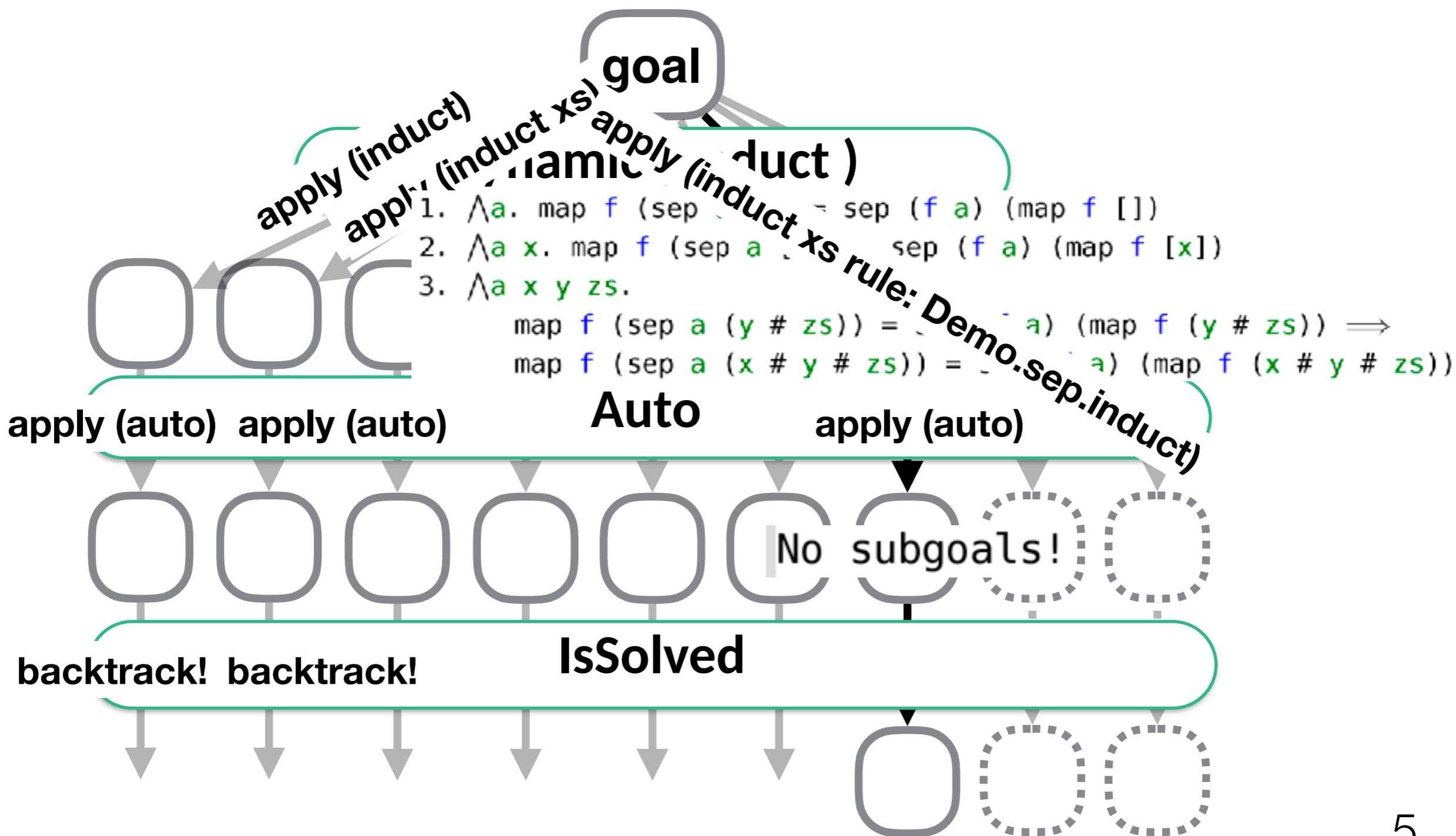
What I did before APLAS2020...

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



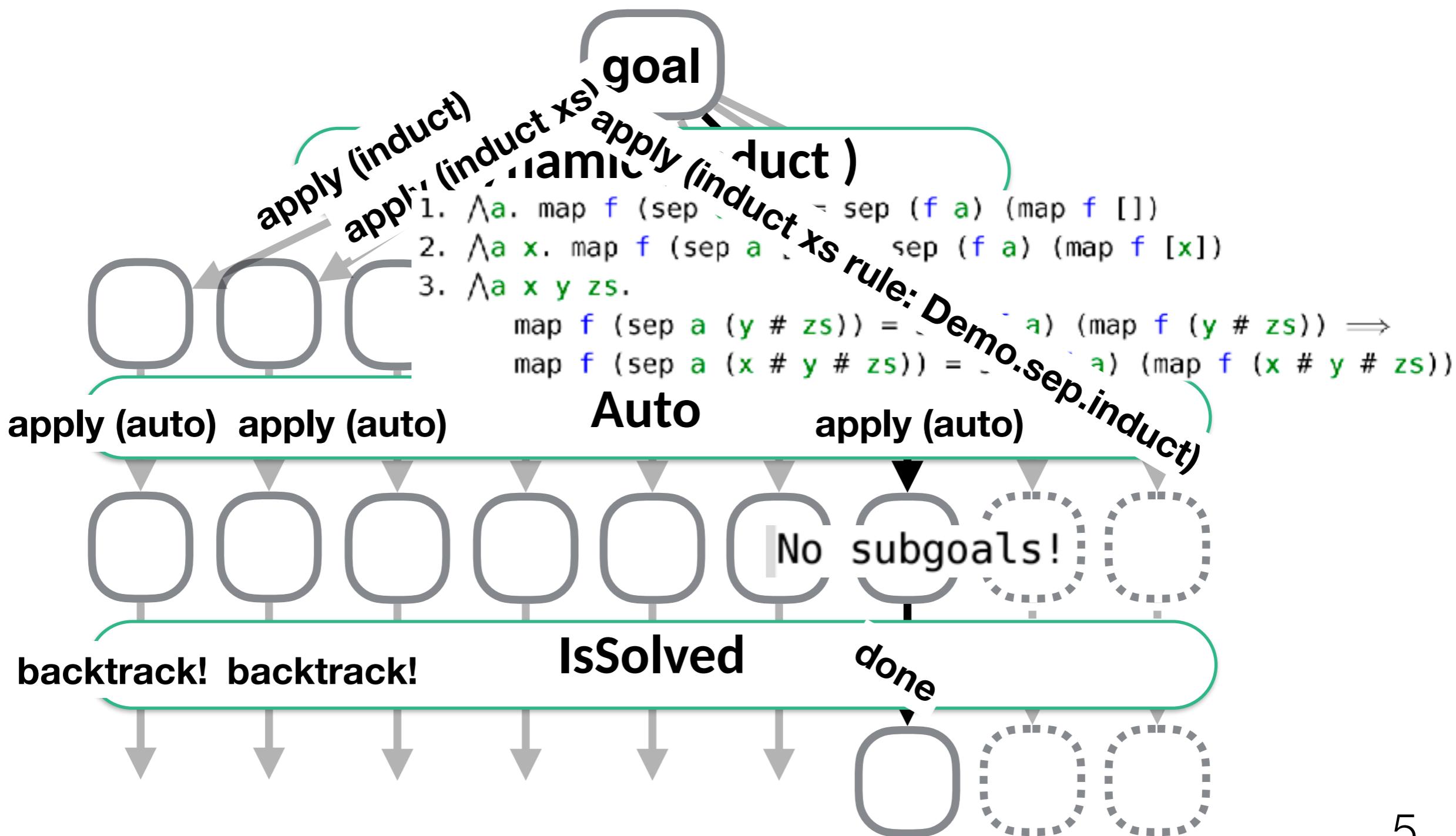
What I did before APLAS2020...

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



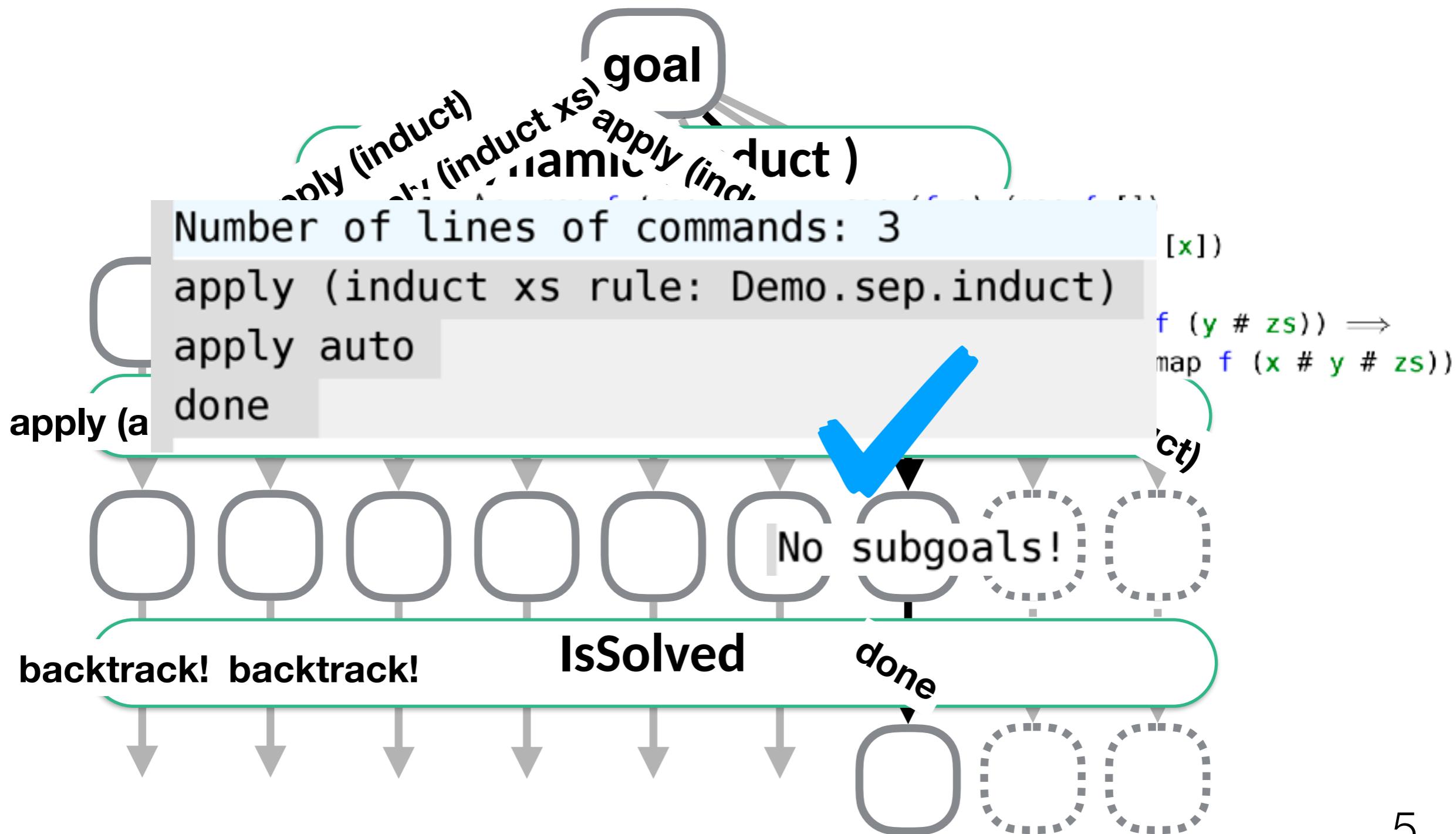
What I did before APLAS2020...

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



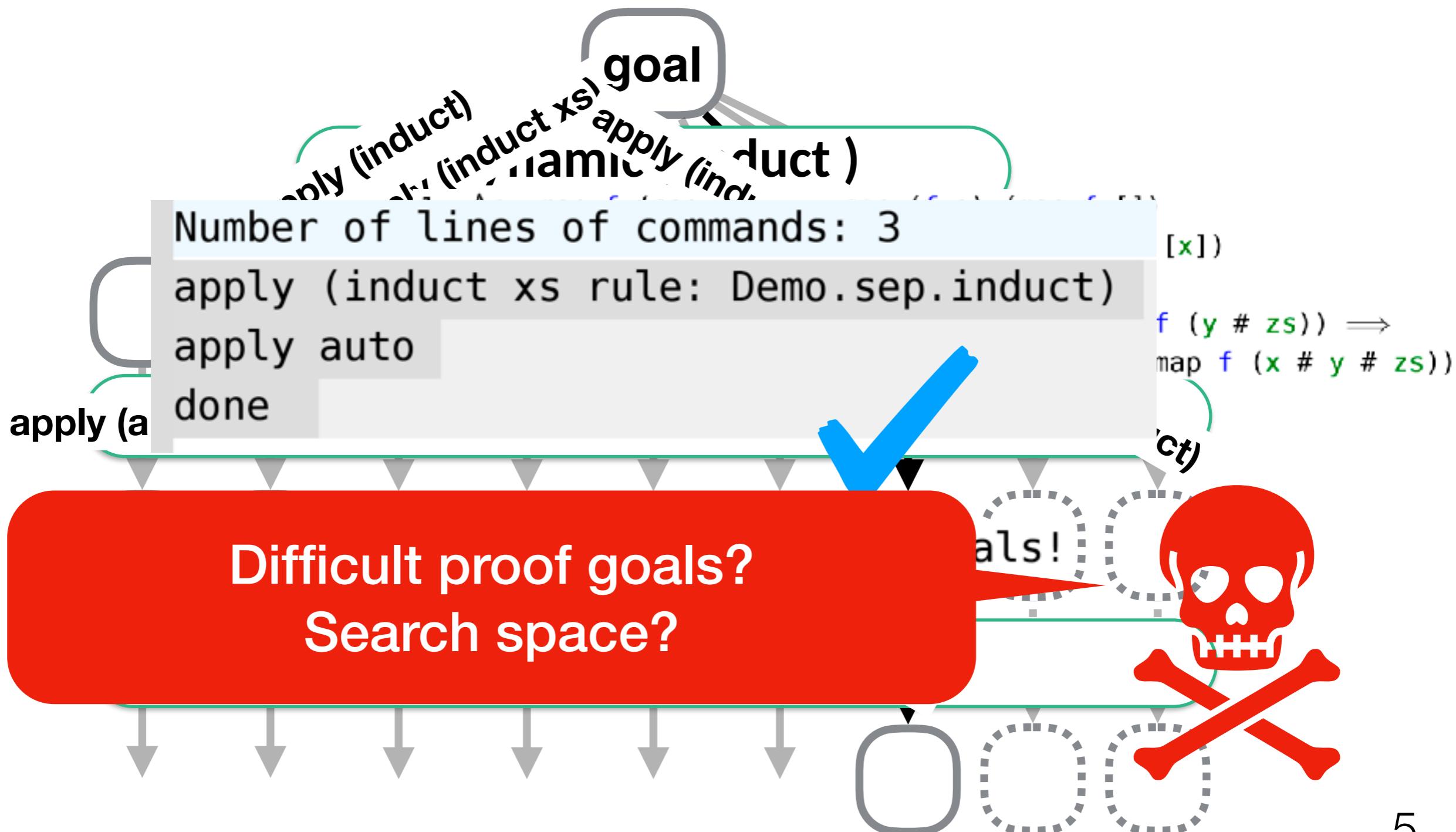
What I did before APLAS2020...

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



What I did before APLAS2020...

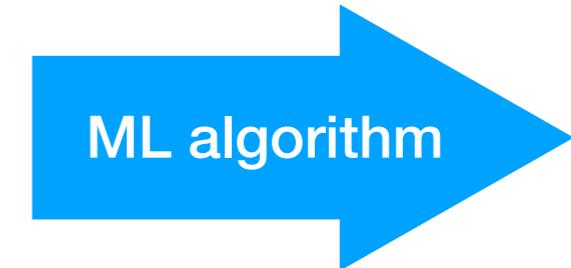
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"  
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



<https://twitter.com/YutakangE>

<https://tinyurl.com/up7zhyk>

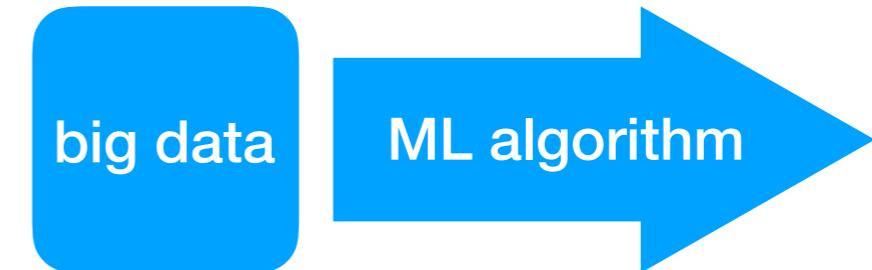
Introduction to Machine Learning in 10 seconds



<https://twitter.com/YutakangE>

<https://tinyurl.com/up7zhyk>

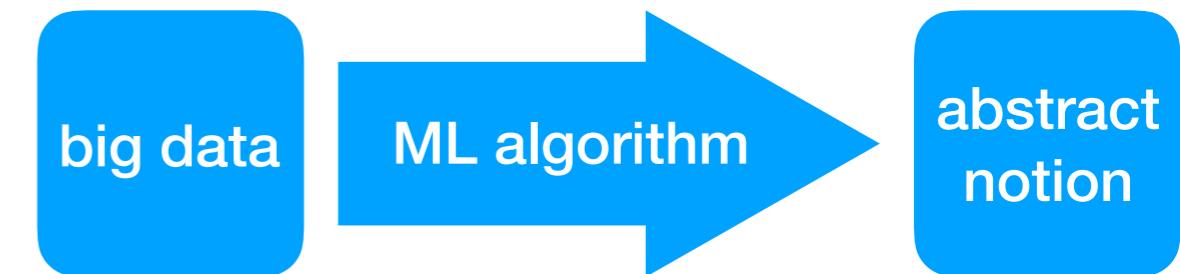
Introduction to Machine Learning in 10 seconds



<https://twitter.com/YutakangE>

<https://tinyurl.com/up7zhyk>

Introduction to Machine Learning in 10 seconds

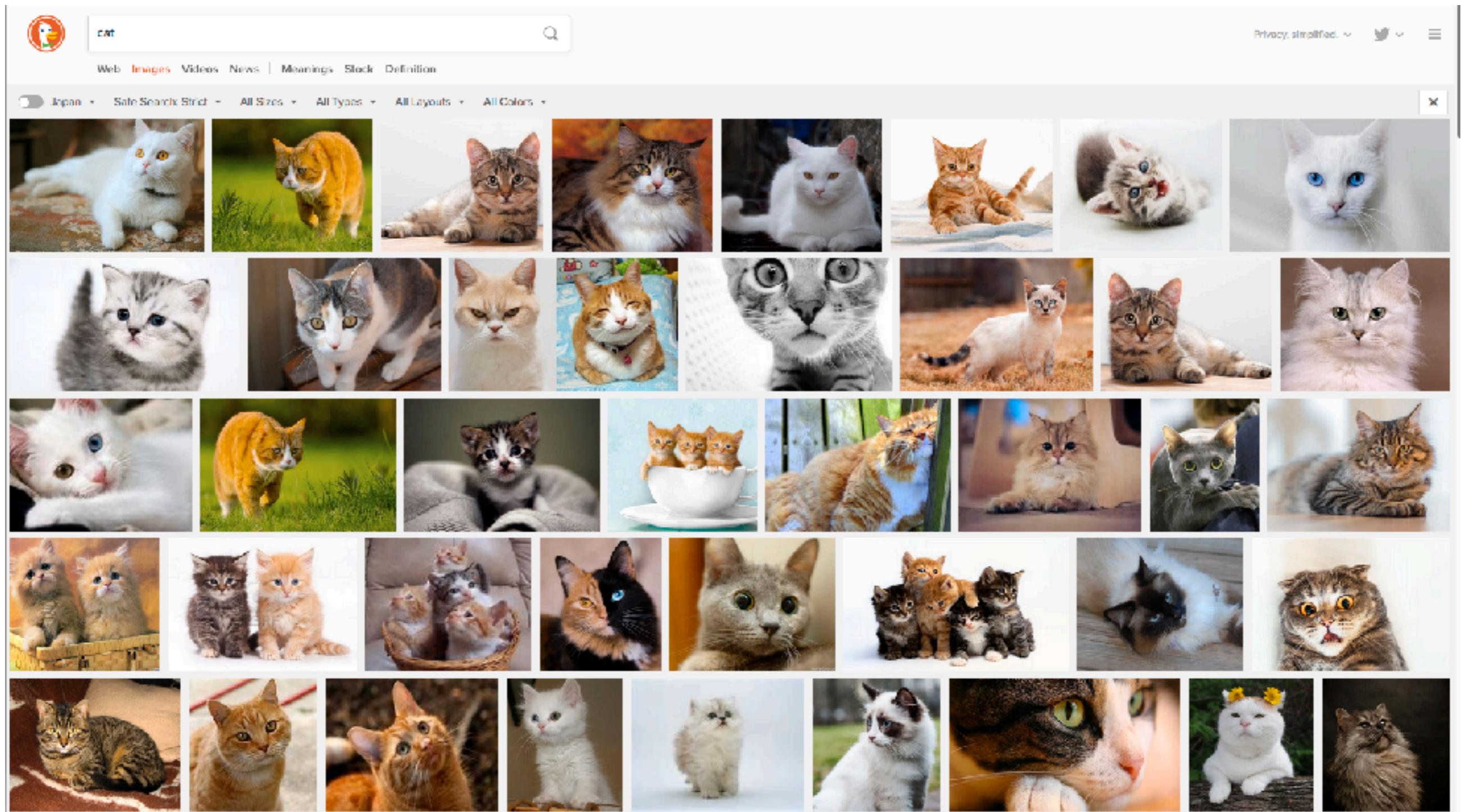


Introduction to Machine Learning in 10 seconds

big data

ML algorithm

abstract notion

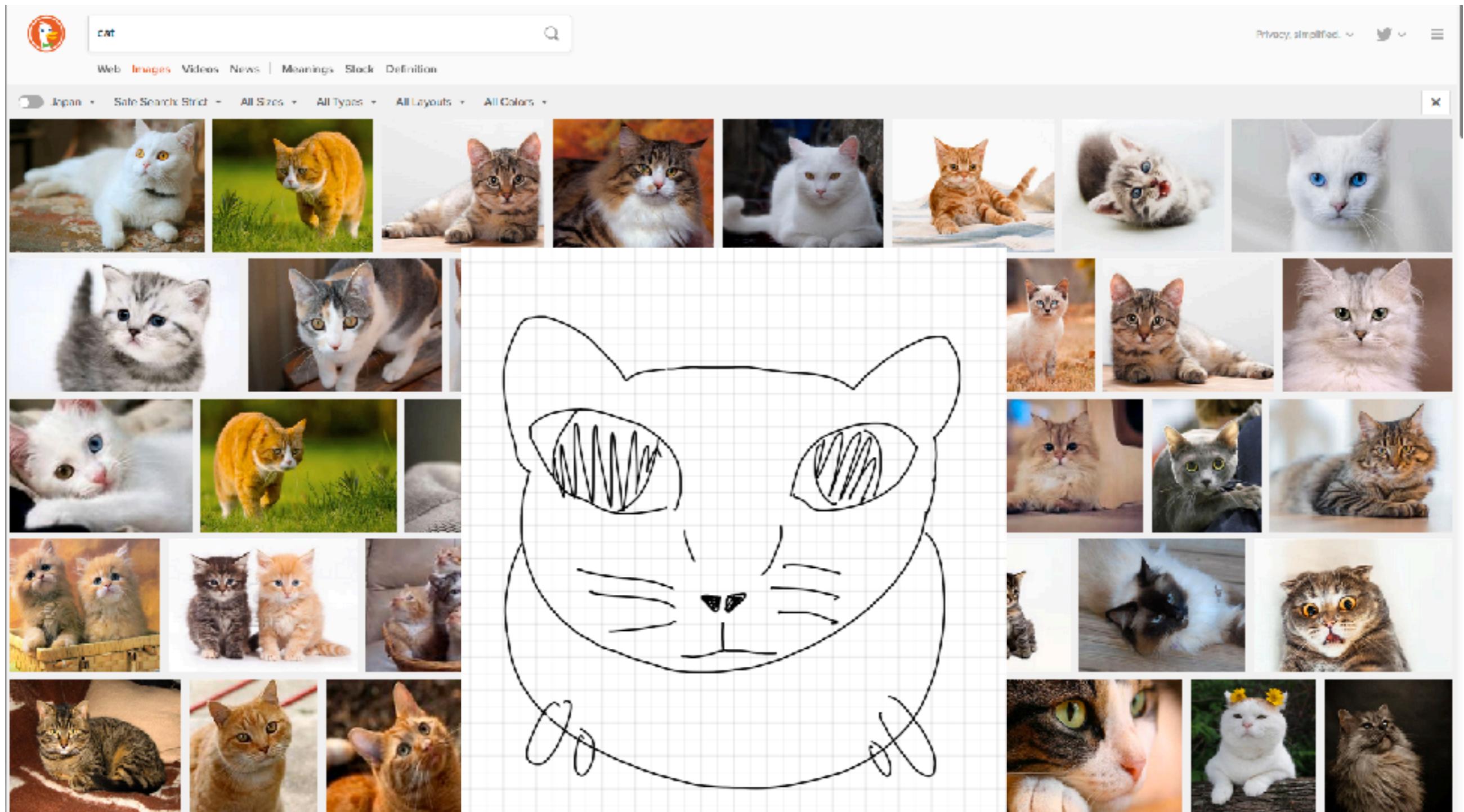


Introduction to Machine Learning in 10 seconds

big data

ML algorithm

abstract notion



ML for Inductive Theorem Proving the BAD

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

← one abstract representation

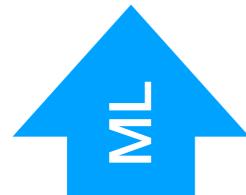
```
lemma "itrev [] = rev [] @ []" by auto
lemma "itrev [1,2] = rev [1,2] @ []" by auto
lemma "itrev [1,2,3] = rev [1,2,3] @ []" by auto
lemma "'a', 'b'" [] = rev ['a', 'b'] @ []" by auto
lemma "[x,y,z]" [] = rev [x,y,z] @ []" by auto
```

← many concrete cases

ML for Inductive Theorem Proving the BAD

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

← one abstract representation



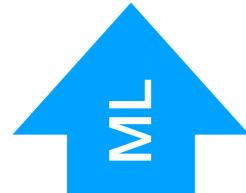
```
lemma "itrev [1,2] [] = rev [1,2] @ []" by auto
lemma "itrev [1,2,3] [] = rev [1,2,3] @ []" by auto
lemma "itrev [''a'', ''b''] [] = rev [''a'', ''b''] @ []" by auto
lemma "itrev [x,y,z] [] = rev [x,y,z] @ []" by auto
```

← many concrete cases

ML for Inductive Theorem Proving the BAD

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

← one abstract representation



```
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
```

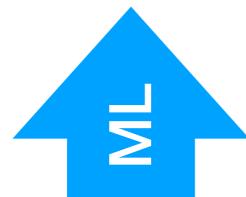
← many concrete cases

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
```

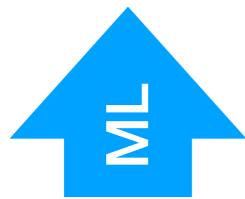
← one abstract representation

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

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



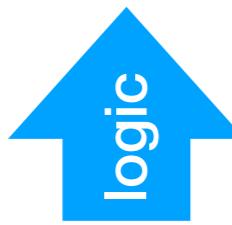
```
lemma "itrev [1,2] [] = rev [1,2] @ []" by auto  
lemma "itrev [1,2,3] [] = rev [1,2,3] @ []" by auto  
lemma "itrev [''a'', ''b''] [] = rev [''a'', ''b''] @ []" by auto  
lemma "itrev [x,y,z] [] = rev [x,y,z] @ []" by auto
```

← 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
```

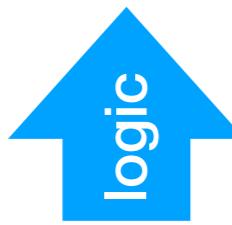
<- many concrete cases

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

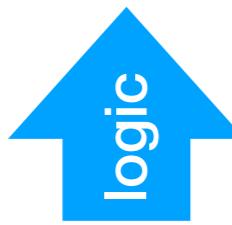
ML for Inductive Theorem Proving the BAD

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

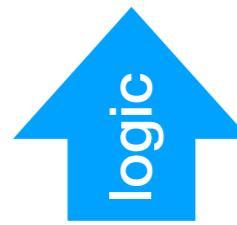
polymorphism

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

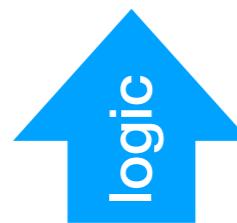
type class

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

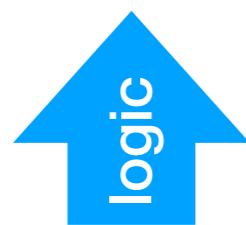
universal quantifier

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

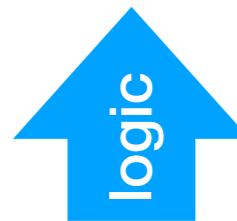
lambda abstraction

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

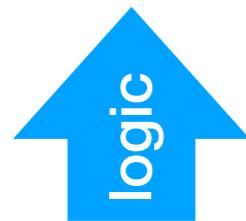
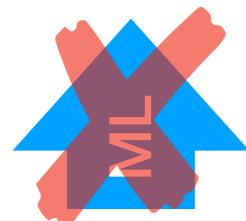
concise formula that can cover
many concrete cases

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
```

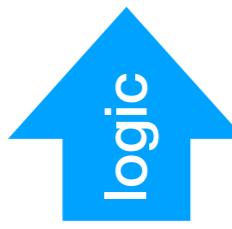
← many concrete cases

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

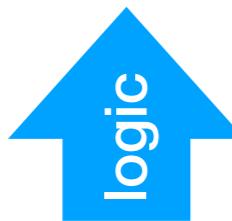
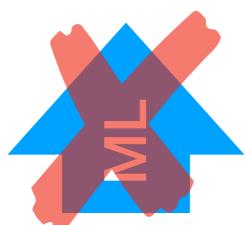
 **lemma** "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

 **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

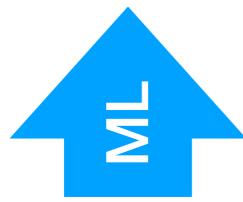


← abstraction using expressive logic

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



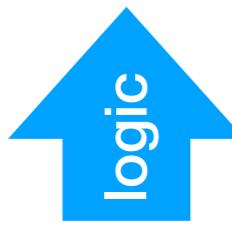
 **Lemma** "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

 **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

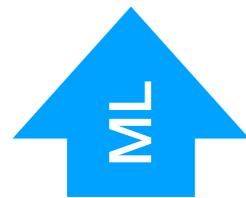
lemma "itrev [1,2] [] = rev [1,2] @ []" **by** auto
lemma "itrev [1,2,3] [] = rev [1,2,3] @ []" **by** auto
lemma "itrev [''a'', ''b''] [] = rev [''a'', ''b''] @ []" **by** auto
lemma "itrev [x,y,z] [] = rev [x,y,z] @ []" **by** auto

← many concrete cases

Grand Challenge: Abstract Abstraction

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

<- even more abstract



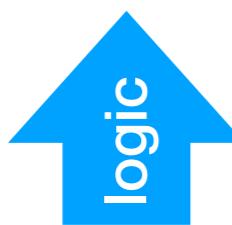
 **Lemma** "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

<- 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

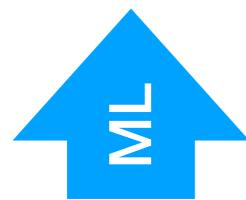
lemma "itrev [1,2] [] = rev [1,2] @ []" **by** auto
lemma "itrev [1,2,3] [] = rev [1,2,3] @ []" **by** auto
lemma "itrev [''a'', ''b''] [] = rev [''a'', ''b''] @ []" **by** auto
lemma "itrev [x,y,z] [] = rev [x,y,z] @ []" **by** auto

<- 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)



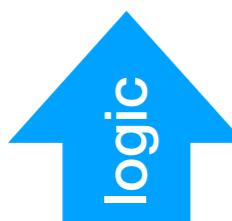
 **Lemma** "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

<- 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

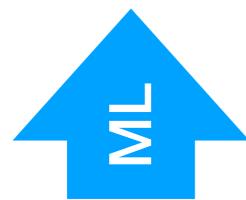
lemma "itrev [1,2] [] = rev [1,2] @ []" **by** auto
lemma "itrev [1,2,3] [] = rev [1,2,3] @ []" **by** auto
lemma "itrev [''a'', ''b''] [] = rev [''a'', ''b''] @ []" **by** auto
lemma "itrev [x,y,z] [] = rev [x,y,z] @ []" **by** auto

<- many concrete cases

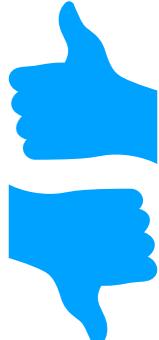
Grand Challenge: Abstract Abstraction

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

← even more abstract



← pros: good at ambiguity (heuristics)



← cons: bad at reasoning & abstraction

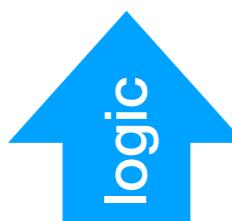
 **Lemma** "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

← 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

Many key challenges remain

Unsupervised Learning

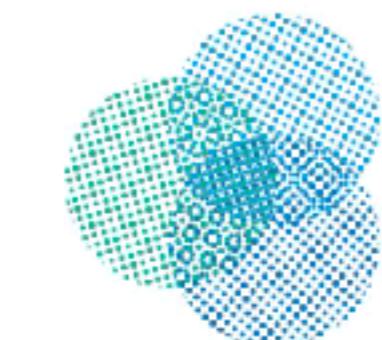
Memory and one-shot learning

Imagination-based Planning with
Generative Models

Learning Abstract Concepts

Transfer Learning

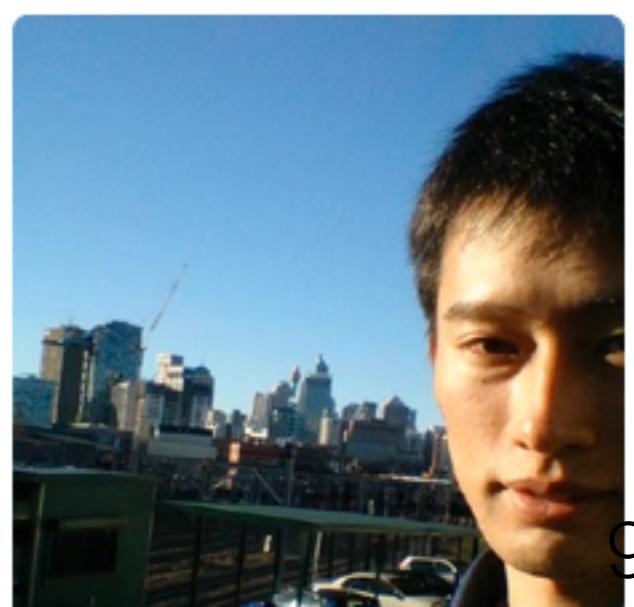
Language understanding



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

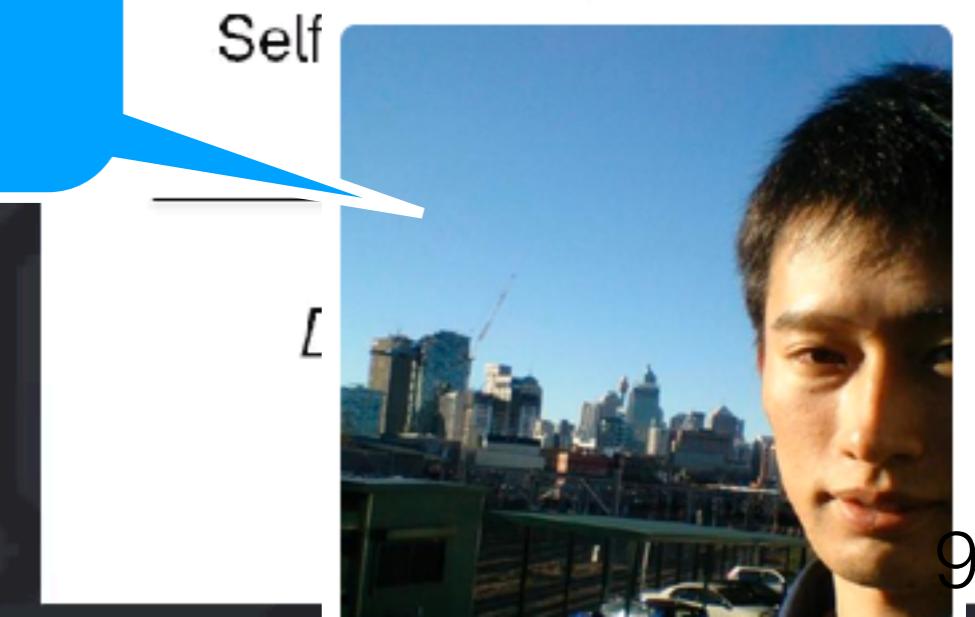
Learning Abstract Concepts

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 = Logical Feature Extraction)



March 20, 2019

The Power of
Self



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

Big Picture

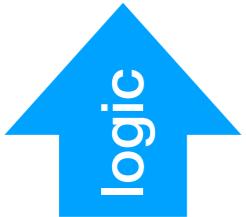
 **lemma** "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

 **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 \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

← 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

ML

[[], []]

Lifter

<- 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



[[] , [] , []] : 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

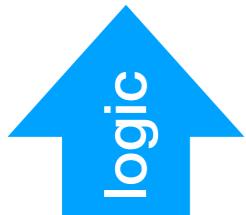


```
lemma "exec (is1 @ is2) s stk =  
      exec is2 s (exec is1 s stk)"  
by(induct is1 s stk rule:exec.induct) auto
```



```
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 "'a', 'b'" [] = rev ['a', 'b'] @ []" by auto  
lemma "[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

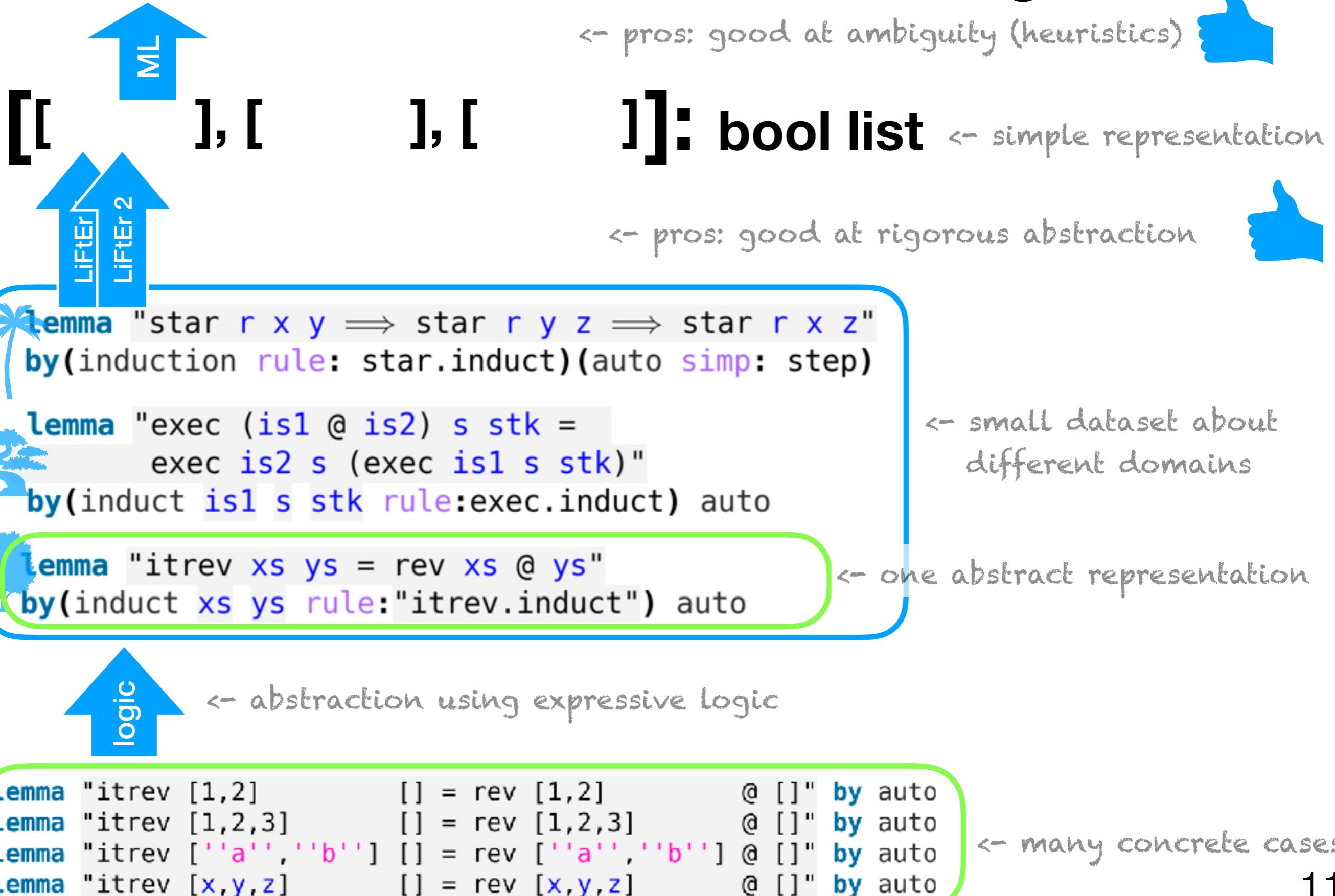
<- abstraction using expressive logic

 **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

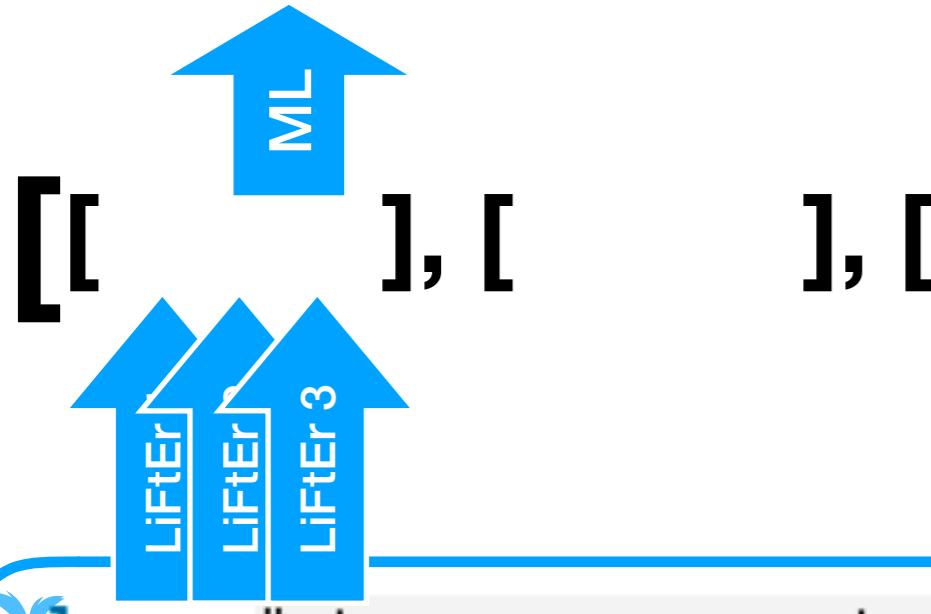
Abstract notion of “good” application of induction.
Heuristics that are valid across problem domains.

Big Picture



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



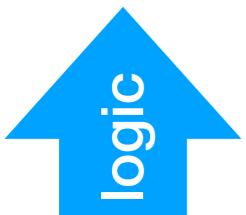
Lemma "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

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



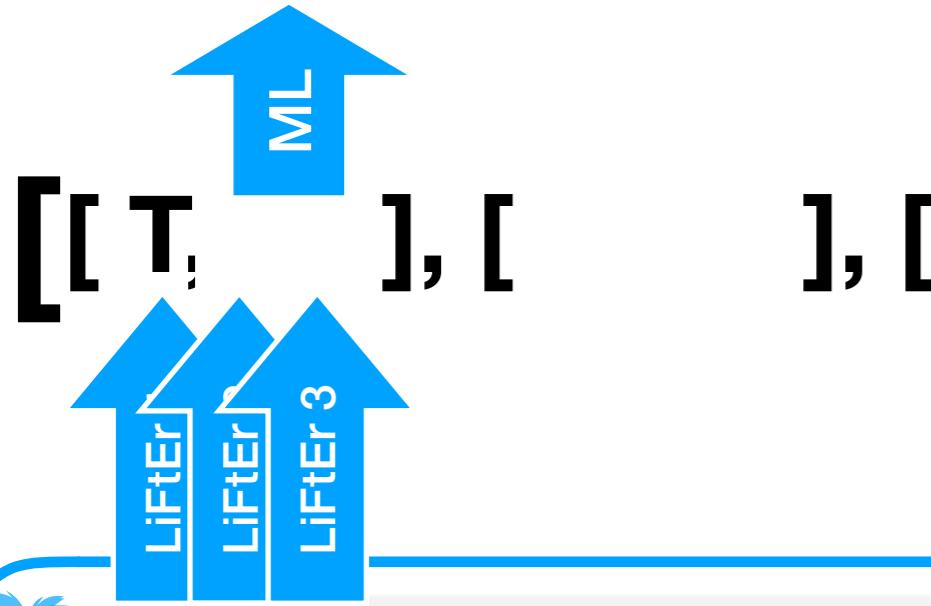
← abstraction using expressive logic

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



`<- simple representation`

`<- pros: good at rigorous abstraction`



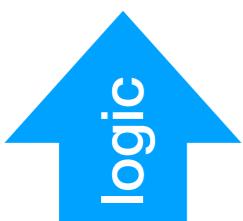
Lemma "star r x y \Rightarrow star r y z \Rightarrow star r x z"
by(induction rule: star.induct)(auto simp: step)

`<- 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



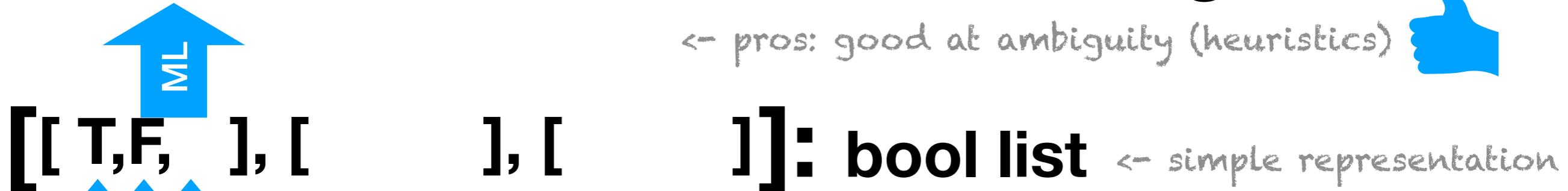
`<- abstraction using expressive logic`

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



 **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

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

<- small dataset about different domains

<- 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

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

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

← pros: good at ambiguity (heuristics)



]]: bool list

← simple representation

LiftEr
LiftEr
LiftEr 3
LiftEr 1
LiftEr 2
LiftEr 3

Lemma "star r
by(induction r

star r y z \Rightarrow star r x z"
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

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) suct) auto
by(induct is1 s stk rule:e)

← 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
by(induction r)

← pros: good at rigorous abstraction



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

← small dataset about
different domains

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

← one abstract representation

logic

← abstraction using expressive logic

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

← many concrete cases

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

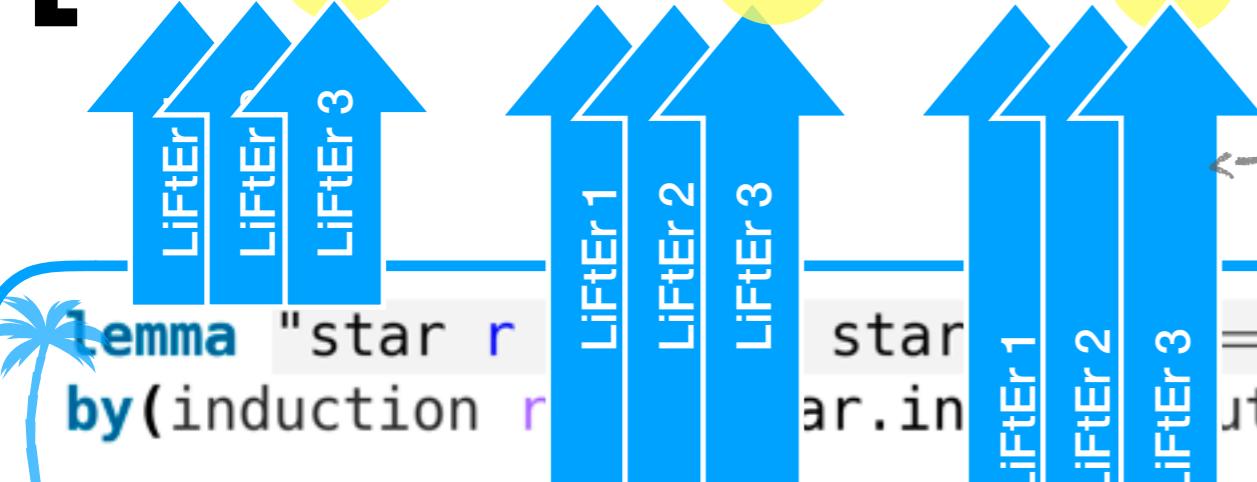
Big Picture

ML

← pros: good at ambiguity (heuristics)

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

← simple representation



← pros: good at rigorous abstraction

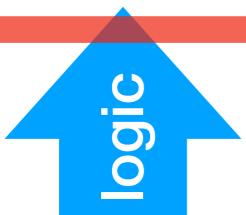


lemma "exec (is1 @ is2) s
exec is2 s (exec is
by(induct is1 s stk rule:e
uct) auto

← small dataset about
different domains

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

← one abstract representation



← abstraction using expressive logic

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

← many concrete cases

Abstract Syntax of LiFtEr

```
assertion := atomic | connective | quantifier | ( assertion )
type := term | term_occurrence | rule | number
modifier_term := induction_term | arbitrary_term
quantifier :=  $\exists x : type . assertion$ 
             |  $\forall x : type . assertion$ 
             |  $\exists x : term \in modifier\_term . assertion$ 
             |  $\forall x : term \in modifier\_term . assertion$ 
             |  $\exists x : rule . assertion$ 
             |  $\exists x : term\_occurrence \in y : term . assertion$ 
             |  $\forall x : term\_occurrence \in y : term . assertion$ 
connective := True | False | assertion  $\vee$  assertion | assertion  $\wedge$  assertion
            | assertion  $\rightarrow$  assertion |  $\neg$  assertion
pattern := all_only_var | all_constructor | mixed
```

atomic :=

```
rule is_rule_of term_occurrence
| term_occurrence term_occurrence_is_of_term term
| are_same_term ( term_occurrence , term_occurrence )
| term_occurrence is_in_term_occurrence term_occurrence
| is_atomic term_occurrence
| is_constant term_occurrence
| is_recursive_constant term_occurrence
| is_variable term_occurrence
| is_free_variable term_occurrence
| is_bound_variable term_occurrence
| is_lambda term_occurrence
| is_application term_occurrence
| term_occurrence is_an_argument_of term_occurrence
| term_occurrence is_nth_argument_of term_occurrence
```

Abstract Syntax of LiFtEr

assertion := atomic | connective | quantifier | (assertion)

type := term | term_occurrence | rule | number

modifier_term := induction_term | arbitrary_term

quantifier := $\exists x : type . assertion$

| $\forall x : type . assertion$

| $\exists x : term \in modifier_term . assertion$

| $\forall x : term \in modifier_term . assertion$

| $\exists x : rule . assertion$

| $\exists x : term_occurrence \in y : term . assertion$

| $\forall x : term_occurrence \in y : term . assertion$

connective := True | False | assertion \vee assertion

 | assertion \rightarrow assertion | \neg assertion

pattern := all_only_var | all_constructor

atomic :=

 rule is_rule_of term_occurrence

 | term_occurrence term_occurrence term_occurrence

 | are_same_term (term_occurrence , term_occurrence)

 | term_occurrence is_in_term_occurrence term_occurrence

 | is_atomic term_occurrence

 | is_constant term_occurrence

 | is_recursive_constant term_occurrence

 | is_variable term_occurrence

 | is_free_variable term_occurrence

 | is_bound_variable term_occurrence

 | is_lambda term_occurrence

 | is_application term_occurrence

 | term_occurrence is_an_argument_of term_occurrence

 | term_occurrence is_nth_argument_of term_occurrence

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
 $r1 \text{ is_rule_of } to1$ ←
 \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
 $t2 \text{ is_nth_induction_term } n$

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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



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

variable for auxiliary lemmas

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

variable for terms

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

$r1 \text{ is_rule_of } to1$

variable for term occurrences

\wedge

conjunction

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

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

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

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

variable for natural numbers

\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

$t2 \text{ is_nth_induction_term } n$

universal quantifier

Example Assertion in LiFtEr (in Abstract Syntax)

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

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

$t2 \text{ is_nth_induction_term } n$

universal
quantifier

<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

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

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

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

$r1 \text{ is_rule_of } to1$

\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

(r1 = itrev.induct)

(t1 = itrev)

<https://twitter.com/YutakangE>

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

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

to1

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

```
apply(induct xs ys rule:"itrev.induct")
apply auto done
```

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

\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

$t2 \text{ is_nth_induction_term } n$

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

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)"
```

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

$t2 \text{ is_nth_induction_term } n$

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

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

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

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

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

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

($t1 = \text{itrev}$)

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

($to1 = \text{itrev}$)

r1 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

$t2 \text{ is_nth_induction_term } n$

$r1$

<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

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

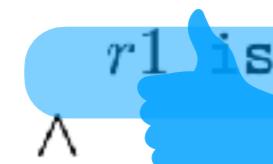
($r1 = \text{itrev.induct}$)

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

($t1 = \text{itrev}$)

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

($to1 = \text{itrev}$)

 $r1 \text{ is_rule_of } to1$ 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

$t2 \text{ is_nth_induction_term } n$

```
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 $to1 (= \text{itrev})$.



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

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

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

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



$t2 \text{ is_nth_induction_term } n$

$r1$

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

($to1 = \text{itrev}$)

($t1 = \text{itrev}$)

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

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

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

$t2 \text{ is_nth_induction_term } n$

$r1$

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

($to1 = \text{itrev}$)

($to1 = \text{itrev}$)

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

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

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

r1

(r1 = itrev.induct)

(t1 = itrev)

(to1 = itrev)

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

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

to1

to2

lemma "itrev xs ys = rev xs @ ys"

apply(induct xs ys rule:"itrev.induct")
apply auto done

t2

r1

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

→

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

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

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

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

^

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

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

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

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

^

$t2 \text{ is_nth_induction_term } n$

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

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

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

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

```
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

$t2 \text{ is_nth_induction_term } n$

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

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

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

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

<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 $to1 (= \text{itrev})$.



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

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

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

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



$t2 \text{ is_nth_induction_term } n$

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

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

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

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

```
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
```

$\exists r_1 : \text{rule}. \text{True}$

\rightarrow

$\exists r_1 : \text{rule}.$

$\exists t_1 : \text{term}.$

$\exists t_{01} : \text{term_occurrence} \in t_1 : \text{term}.$

$r_1 \text{ is_rule_of } t_{01}$ True! $r_1 (= \text{itrev.induct})$ is a lemma about $t_{01} (= \text{itrev})$.

\wedge

$\forall t_2 : \text{term} \in \text{induction_term}.$

$\exists t_{02} : \text{term_occurrence} \in t_2 : \text{term}.$

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

$\text{is_nth_argument_of } (t_{02}, n, t_{01})$

\wedge

$t_2 \text{ is_nth_induction_term } n$

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

($t_1 = \text{itrev}$)

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

($t_2 = \text{xs}$ and ys)

($t_{02} = \text{xs}$ and ys)

when t_2 is xs ($n = 1$)?

```
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
```

$\exists r_1 : \text{rule}. \text{True}$

\rightarrow

$\exists r_1 : \text{rule}.$

$\exists t_1 : \text{term}.$

$\exists t_{01} : \text{term_occurrence} \in t_1 : \text{term}.$

$r_1 \text{ is_rule_of } t_{01}$ True! $r_1 (= \text{itrev.induct})$ is a lemma about $t_{01} (= \text{itrev})$.

\wedge

$\forall t_2 : \text{term} \in \text{induction_term}.$

$\exists t_{02} : \text{term_occurrence} \in t_2 : \text{term}.$

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

$\text{is_nth_argument_of } (t_{02}, n, t_{01})$

\wedge

$t_2 \text{ is_nth_induction_term } n$

t_{01}

t_{02}

t_2

r_1

t_{01}

t_{02}

t_2

r_1

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

($t_1 = \text{itrev}$)

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

($t_2 = xs$ and ys)

($t_{02} = xs$ and ys)

when t_2 is xs ($n = 1$)

```
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

$\exists r_1 : \text{rule}. \text{True}$

\rightarrow

$\exists r_1 : \text{rule}.$

$\exists t_1 : \text{term}.$

$\exists t_{01} : \text{term_occurrence} \in t_1 : \text{term}.$

$r_1 \text{ is_rule_of } t_{01}$ True! $r_1 (= \text{itrev.induct})$ is a lemma about $t_{01} (= \text{itrev})$.

\wedge

$\forall t_2 : \text{term} \in \text{induction_term}.$

$\exists t_{02} : \text{term_occurrence} \in t_2 : \text{term}.$

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

$\text{is_nth_argument_of } (t_{02}, n, t_{01})$

\wedge

$t_2 \text{ is_nth_induction_term } n$

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

($t_1 = \text{itrev}$)

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

($t_2 = xs \text{ and } ys$)

($t_{02} = xs \text{ and } ys$)

when t_2 is xs ($n = 1$)

when t_2 is ys ($n = 2$) ?

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

```
fun itrev :: "'a list ⇒ 'a list ⇒ 'a list" where
```

```
  "itrev [] ys" = ys |
```

```
  "itrev (x#xs) ys" = itrev xs (x#ys)"
```

to1 → first second to2

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

```
apply(induct xs ys rule:"itrev.induct")
```

```
apply auto done
```

t2
first
second

r1

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

→

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

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

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

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



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

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

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

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

^

$t2 \text{ is_nth_induction_term } n$

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

($t1 = \text{itrev}$)

($to1 = \text{itrev}$)

($t2 = xs$ 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)"
```

t_{01} → t_{02}

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

```
apply(induct xs ys rule:"itrev.induct")
```

```
apply auto done
```

t_2 → t_1
first → second
second → first

r_1

$\exists r_1 : \text{rule}. \text{True}$

→

$\exists r_1 : \text{rule}.$

$\exists t_1 : \text{term}.$

$\exists t_{01} : \text{term_occurrence} \in t_1 : \text{term}.$

$r_1 \text{ is_rule_of } t_{01}$ True! $r_1 (= \text{itrev.induct})$ is a lemma about $t_{01} (= \text{itrev})$.

^

$\forall t_2 : \text{term} \in \text{induction_term}.$

$\exists t_{02} : \text{term_occurrence} \in t_2 : \text{term}.$

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

$\text{is_nth_argument_of} (t_{02}, n, t_{01})$

^

$t_2 \text{ is_nth_induction_term } n$

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

($t_1 = \text{itrev}$)

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

($t_2 = xs$ and ys)

($t_{02} = xs$ and ys)

when t_2 is xs ($n = 1$)



when t_2 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 : rule. True  
→  
exists r1 : rule.  
  exists t1 : term.  
    exists to1 : term_occurrence ∈ t1 : term.  
      r1 is_rule_of to1  
      ∧  
      ∀ t2 : term ∈ induction_term.  
        exists to2 : term_occurrence ∈ t2 : term.  
          exists n : number.  
            is_nth_argument_of (to2, n, to1)  
            ∧  
            t2 is_nth_induction_term n
```

new types ->

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

the same LIFTER assertion



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

\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

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

(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

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

r1

→

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

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

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

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

r1 is_rule_of to1

^

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

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

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

is_nth_argument_of (to2, n, to1)

^

t2 is_nth_induction_term n

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

to1

r1

→

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

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

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

$r1 \text{ is_rule_of } to1$

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

to1

r1

→

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

(r1 = exec.induct)

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

(t1 = exec)

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

(to1 = exec)

$r1 \text{ is_rule_of } to1$

\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

to1

r1

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

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

($t1 = \text{exec}$)

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

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

\wedge

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

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

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

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

\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

to1

r1

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

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

($t1 = \text{exec}$)

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

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

\wedge

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

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

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

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

\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

to1

r1

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

(r1 = exec.induct)

(t1 = exec)

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

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



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

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

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

is_nth_argument_of (to2, n, to1)

\wedge

t2 is_nth_induction_term n

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

to1

r1

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

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

($t1 = \text{exec}$)

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

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

\wedge

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

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

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

is_nth_argument_of (t2, n, to1)

\wedge

t2 is_nth_induction_term n

new types ->

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

new constants ->

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

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

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

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

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

b2

to1

r1

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

(r1 = exec.induct)

(t1 = exec)

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

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

^

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

(t2 = is1, s, and stk)

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

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

is_nth_argument_of (to2, n, to1)

^

t2 is_nth_induction_term n

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**

→

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

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

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

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

^

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

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

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

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

^

$t2 \text{ is_nth_induction_term } n$

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 $to1 (= \text{exec})$.



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

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

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

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



$t2 \text{ is_nth_induction_term } n$

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 \rightarrow

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

new constants \rightarrow

```
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" | first
  "exec (i#is) s stk = exec is s (exec1 i s stk)"
```

~~new lemma \rightarrow lemma "exec (is1 @ is2) s \text{ stk} = \text{exec is2 s} (\text{exec is1 s \text{ stk})}"~~

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

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

apply auto done

→

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

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

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

(r1 = exec.induct)

(E1 = exec)

(b01 = exec)

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

8

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

(t2 = is1, s, and stk)

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

(to2 = is1, s, and stk)

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

is_nth_argument_of (to2, n, to1)

when t_2 is $i s_1$ ($n \rightarrow 1$)?

△

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



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

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

($t1 = \text{exec}$)

($\text{to1} = \text{exec}$)

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

first

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

($\text{to1} = \text{exec}$)

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

\wedge

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

($t2 = \text{is1, s, and stk}$)
($\text{to2} = \text{is1, s, and stk}$)

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

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

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

when $t2$ 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



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

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

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

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



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

($t1 = \text{exec}$)

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

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



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

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

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

($to2 = is1, s, \text{and } stk$)

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

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

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



$t2 \text{ is_nth_induction_term } n$

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



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

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

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

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



first
second (r1 = exec.induct)

(t1 = exec)

(t01 = exec)

\wedge

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

(t2 = is1, s, and stk)

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

(t02 = is1, s, and stk)

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

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

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



$t2 \text{ is_nth_induction_term } n$

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



→

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

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

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

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



first third
second (r1 = exec.induct)

(t1 = exec)

(to1 = exec)

r1

^

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

(t2 = is1, s, and stk)

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

(to2 = is1, s, and stk)

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

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

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

^

$t2 \text{ is_nth_induction_term } n$

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



→

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

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

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

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



first third

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

($t1 = \text{exec}$)

($to1 = \text{exec}$)

$r1$

^

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

($t2 = is1, s, \text{and } stk$)
($to2 = is1, s, \text{and } stk$)

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

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

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

^

$t2 \text{ is_nth_induction_term } n$

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



→

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

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

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

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

^

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

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

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

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

^

$t2 \text{ is_nth_induction_term } n$

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

($to2 = is1, s, \text{and } stk$)

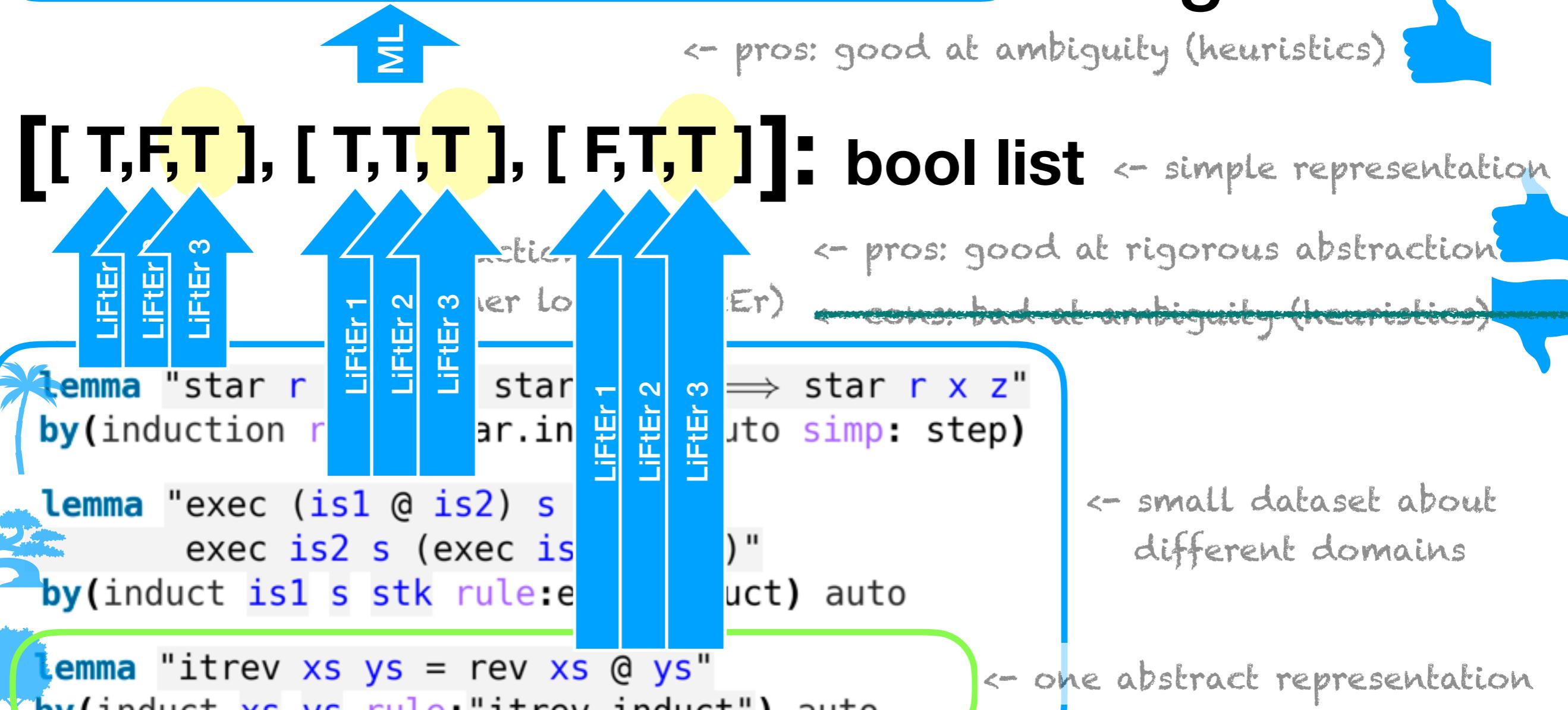
when $t2$ is $is1$ ($n \rightarrow 1$)

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



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 "'a', 'b'" [] = rev ['a', 'b'] @ []" by auto
lemma "[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,F,T], [T,T,T], [F,T,T]]: bool list

WIP

← pros: good at ambiguity (heuristics)

LiftEr 1 LiftEr 2 LiftEr 3
LiftEr 1 LiftEr 2 LiftEr 3
LiftEr 1 LiftEr 2 LiftEr 3
LiftEr 1 LiftEr 2 LiftEr 3

Lemma "star r by(induction r)"

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

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

← pros: good at rigor

← cons: bad at ambiguity (heuristics)



← 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

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

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

\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

\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)"
```

← 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
```

← 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

WIP

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" auto
by(induct xs ys rule:"itrev.induct") 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

WIP

lemma "star r x y \Rightarrow star r x z"
by(induction rule: star.in)

\Rightarrow star r x z"
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

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

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

<- relevant definitions

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

fin.