https://twitter.com/YutakangJ

AI for Isabelle/HOL

Yutaka Nagashima
University of Innsbruck
Czech Technical University

universität innsbruck

CZECH INSTITUTE OF INFORMATICS ROBOTICS AND CYBERNETICS CTU IN PRAGUE

AI for Isabelle/HOL

Yutaka Nagashima
University of Innsbruck
Czech Technical University

universität
innsbruck

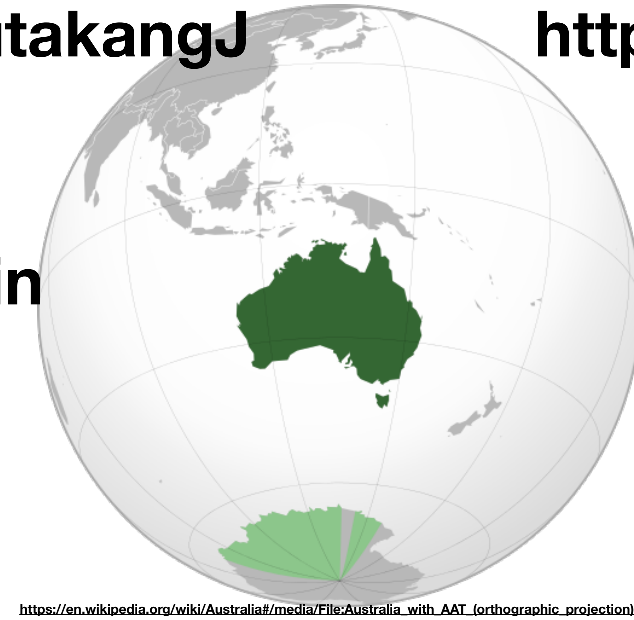
CZECH INSTITUTE
OF INFORMATICS
ROBOTICS AND
CYBERNETICS
CTU IN PRAGUE

https://twitter.com/YutakangJ

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

2013 ~ 2017

with Dr. Gerwin Klein



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

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

Isabelle HOL

(seL4)

Security. Performance. Proof.

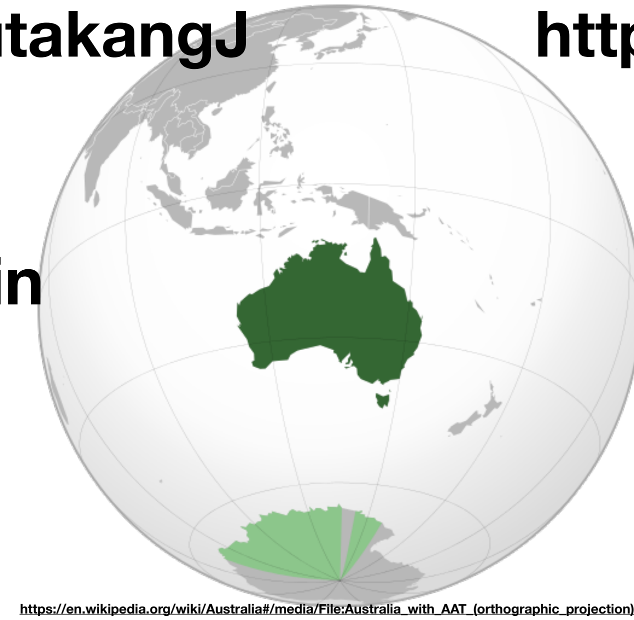
https://twitter.com/YutakangJ

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

2013 ~ 2017

with Dr. Gerwin Klein

pre-PhD



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

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

Isabelle HOL

(seL4 Security. Performance. Proof.)

https://twitter.com/YutakangJ

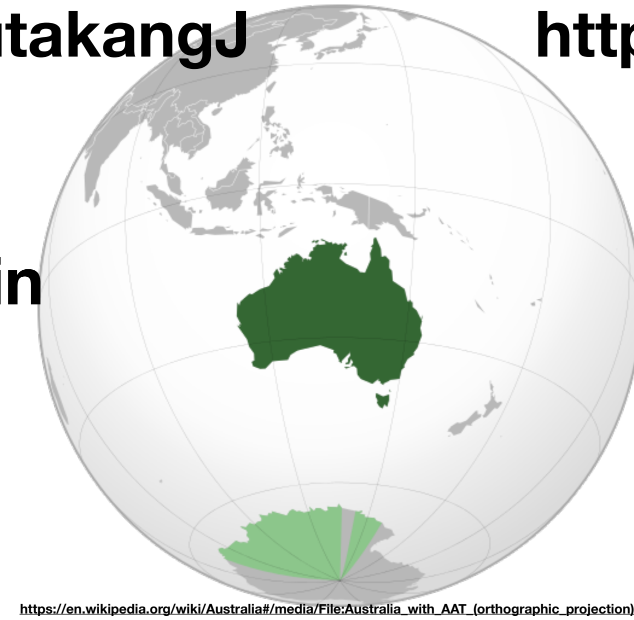
https://github.com/data61/PSL/slide/2019_ps.pdf

2013 ~ 2017

with Dr. Gerwin Klein

pre-PhD

PhD in AI for theorem proving



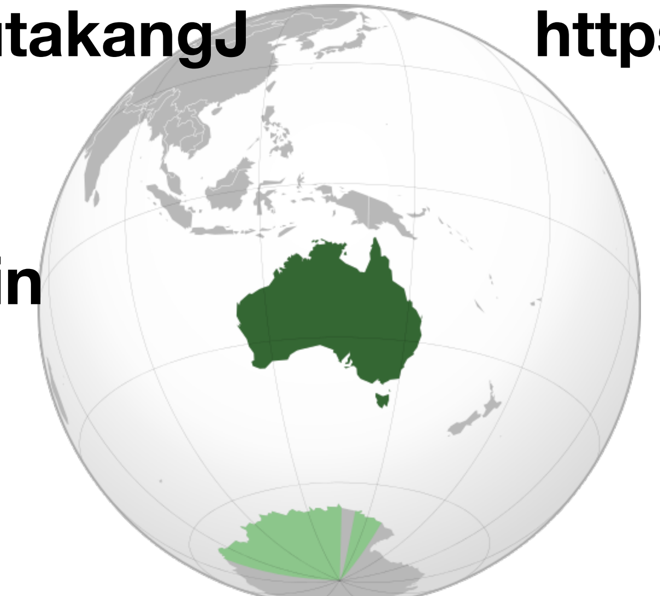
https://en.wikipedia.org/wiki/Australia#/media/File:Australia_with_AAT_(orthographic_projection).svg

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

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

2013 ~ 2017

with Dr. Gerwin Klein



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

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

(seL4 Security. Performance. Proof.)

pre-PhD

PhD in AI for theorem proving

2017 ~ 2018

with Prof. Cezary Kaliszyk

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

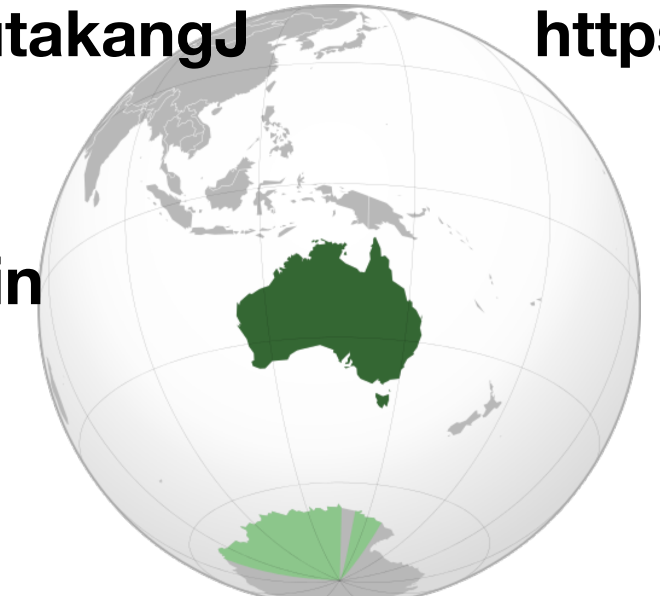
http://cl-informatik.uibk.ac.at/users/cek/

https://twitter.com/YutakangJ

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

2013 ~ 2017

with Dr. Gerwin Klein



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

(seL4)

Security. Performance. Proof.

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

pre-PhD

PhD in

AI for theorem proving

2017 ~ 2018

with Prof. Cezary Kaliszyk

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

http://cl-informatik.uibk.ac.at/users/cek/

2018 ~ 2020

with Dr. Josef Urban

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

http://ai4reason.org/members.html

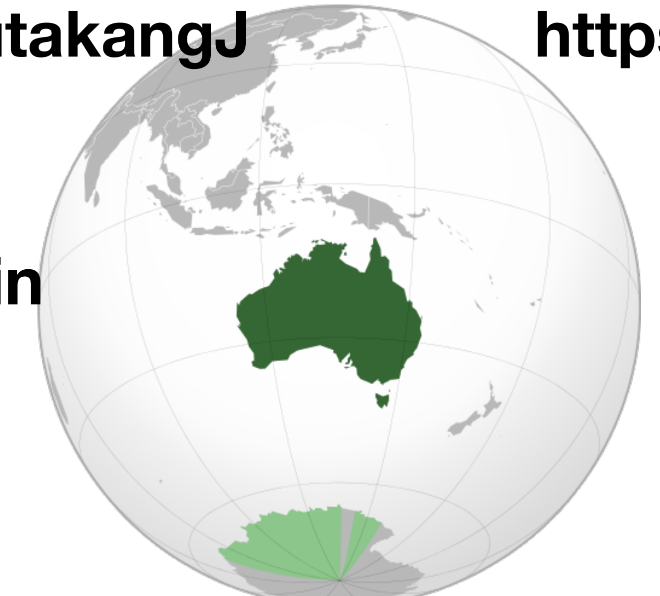
https://twitter.com/YutakangJ

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

2013 ~ 2017

with Dr. Gerwin Klein

pre-PhD



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

Isabelle HOL (seL4 Security. Performance. Proof.)

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

PhD in
AI for theorem proving

2017 ~ 2018
2020 ~ 2021?

with Prof. Cezary Kaliszyk

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

Isabelle HOL

http://cl-informatik.uibk.ac.at/users/cek/

2018 ~ 2020

with Dr. Josef Urban

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

Isabelle HOL

http://ai4reason.org/members.html

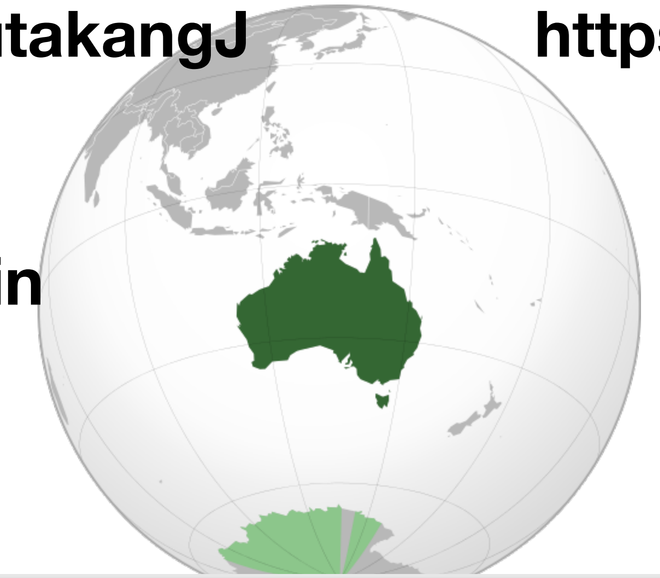
https://twitter.com/YutakangJ

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

2013 ~ 2017

with Dr. Gerwin Klein

pre-PhD



(seL4)

Registration is now closed.

http://aitp-conference.org/2019/

Background

Large-scale semantic processing and strong computer assistance of mathematics and science is our inevitable future. New combinations of AI and reasoning methods and tools deployed over large mathematical and scientific corpora will be instrumental to this task. The AITP conference is the forum for discussing how to get there as soon as possible, and the force driving the progress towards that.

Topics

- AI and big-data methods in theorem proving and mathematics
- Collaboration between automated and interactive theorem proving
- Common-sense reasoning and reasoning in science
- Alignment and joint processing of formal, semi-formal, and informal libraries
- Methods for large-scale computer understanding of mathematics and science
- Combinations of linguistic/learning-based and semantic/reasoning methods

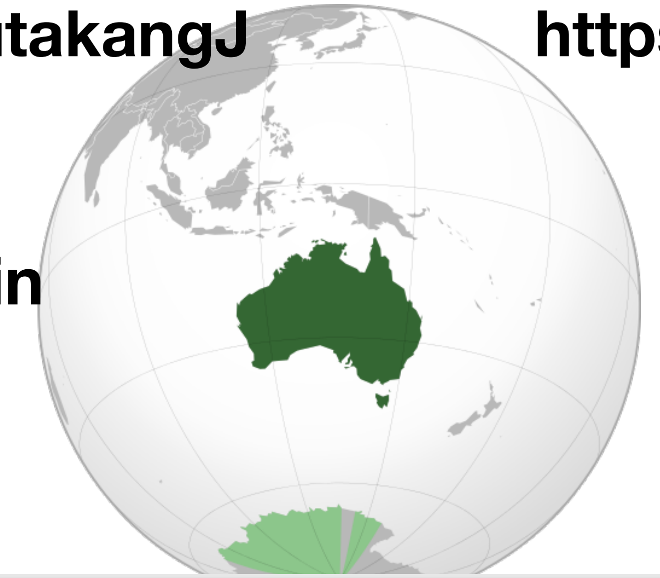
https://twitter.com/YutakangJ

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

2013 ~ 2017

with Dr. Gerwin Klein

pre-PhD



Isabelle HOL

(seL4)

Registration is now closed.

http://aitp-conference.org/2019/

Background

Large-scale semantic processing and strong computer assistance of mathematics and science is our inevitable future. New combinations of AI and reasoning methods and tools deployed over large mathematical and scientific corpora will be instrumental to this task. The AITP conference is the forum for discussing how to get there as soon as possible, and the force driving the progress towards that.

?

Topics

- AI and big-data methods in theorem proving and mathematics
- Collaboration between automated and interactive theorem proving
- Common-sense reasoning and reasoning in science
- Alignment and joint processing of formal, semi-formal, and informal libraries
- Methods for large-scale computer understanding of mathematics and science
- Combinations of linguistic/learning-based and semantic/reasoning methods

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

Isabelle/HOL architecture

Isabelle/HOL architecture

ML (Poly/ML)

Isabelle/HOL architecture

Meta-logic

ML (Poly/ML)

Isabelle/HOL architecture

HOL

Meta-logic

ML (Poly/ML)

Isabelle/HOL architecture

Isar

HOL

Meta-logic

ML (Poly/ML)

Isabelle/HOL architecture

PIDE / jEdit

Isar

HOL

Meta-logic

ML (Poly/ML)

Isabelle/HOL architecture

PIDE / jEdit

Isar

HOL

Meta-logic

ML (Poly/ML)

Isabelle/HOL architecture

PIDE / jEdit

Isar

HOL

Meta-logic

ML (Poly/ML)

You can access all the layers!
:)

Isabelle/HOL architecture

PIDE / jEdit

Isar

HOL

Meta-logic

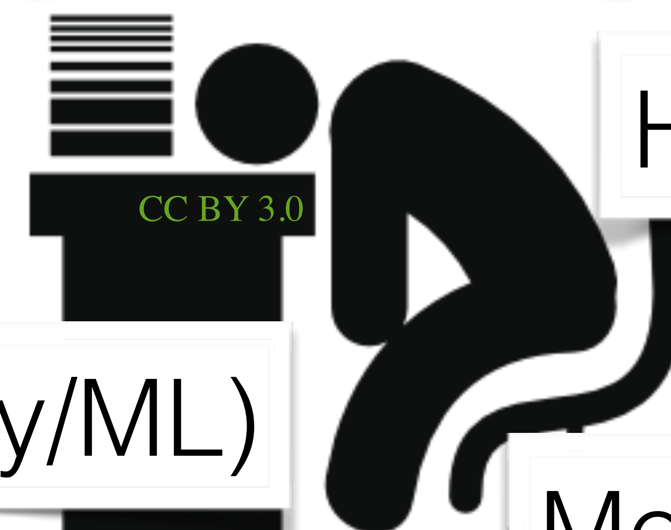
ML (Poly/ML)

You can access all the layers!
:)

They come all together!
: (

PIDE / jEdit

Isar

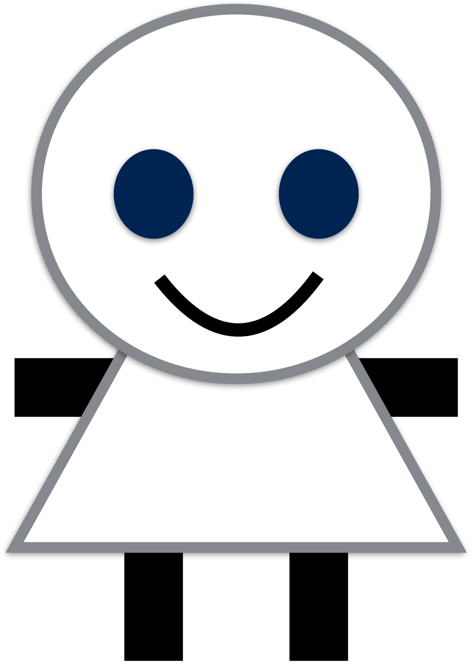


HOL

ML (Poly/ML)

Meta-logic

Interactive theorem proving with Isabelle/HOL

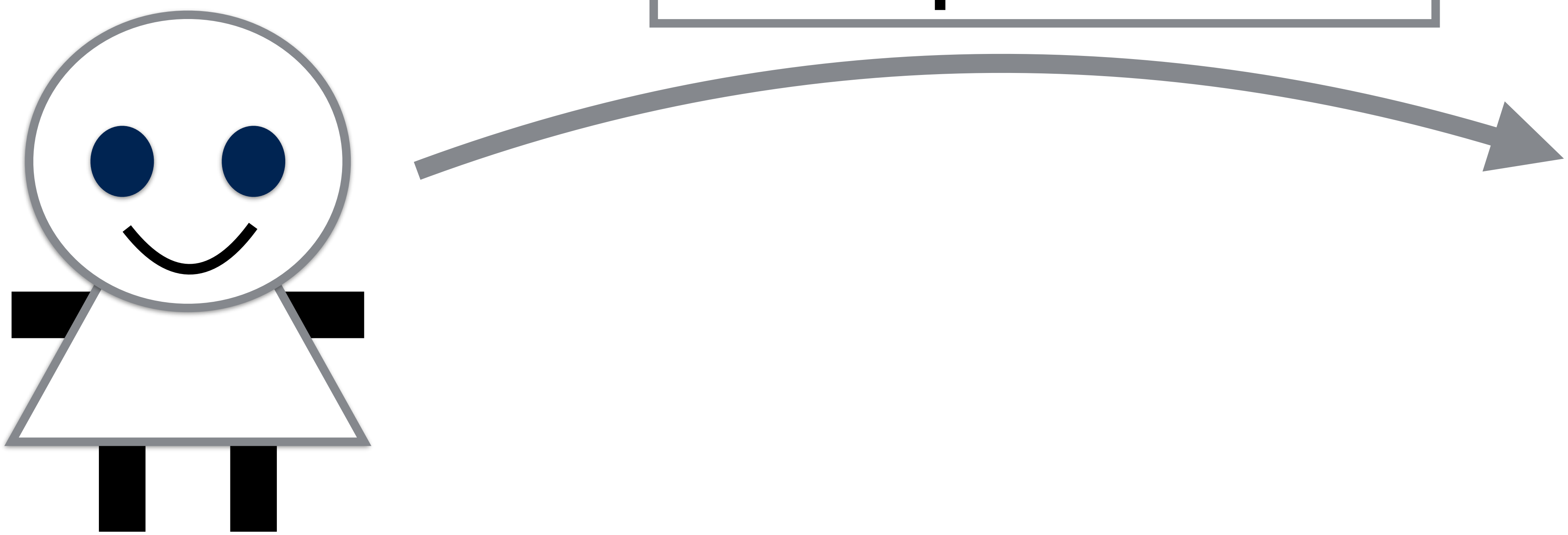


https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

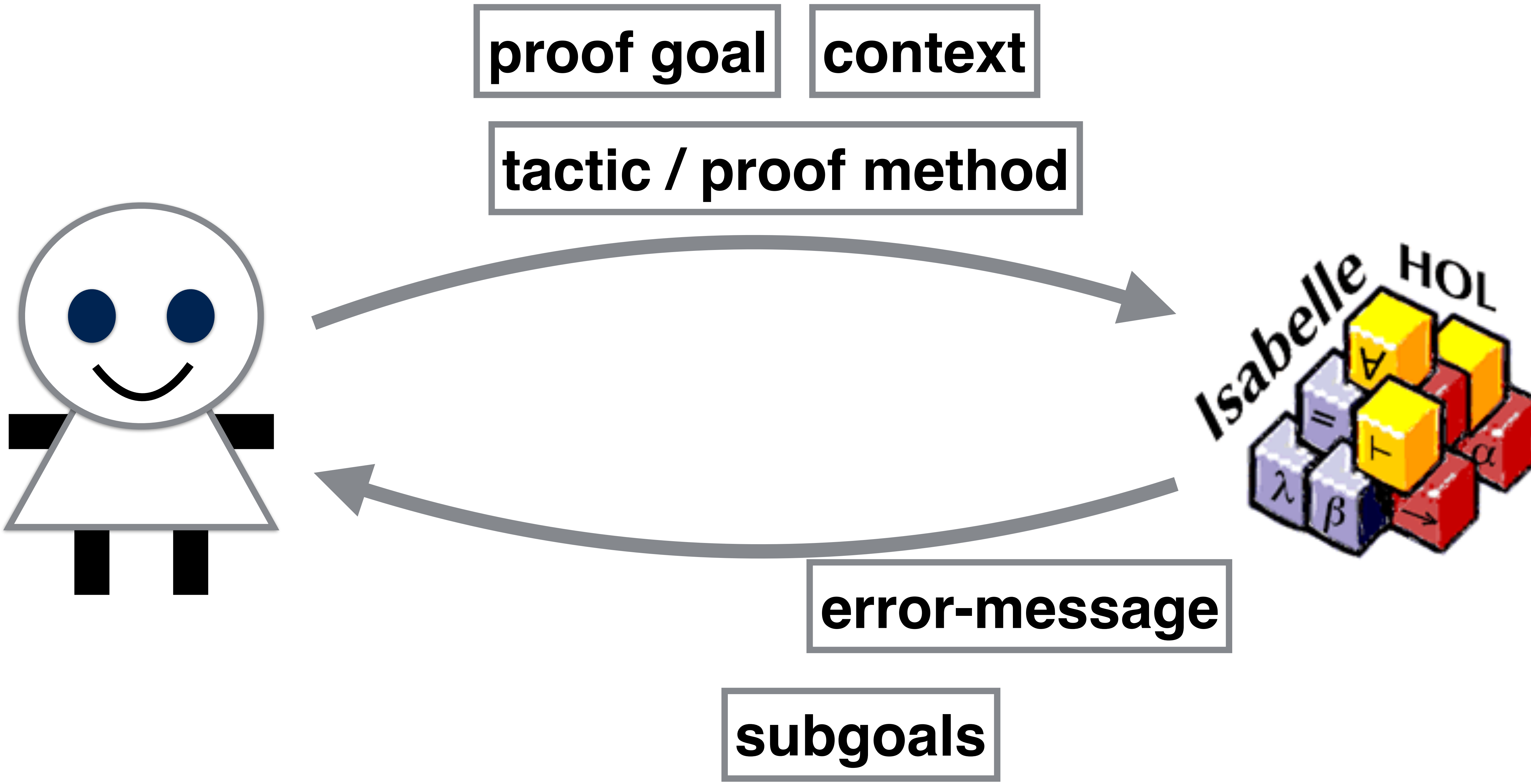
Interactive theorem proving with Isabelle/HOL

proof goal **context**

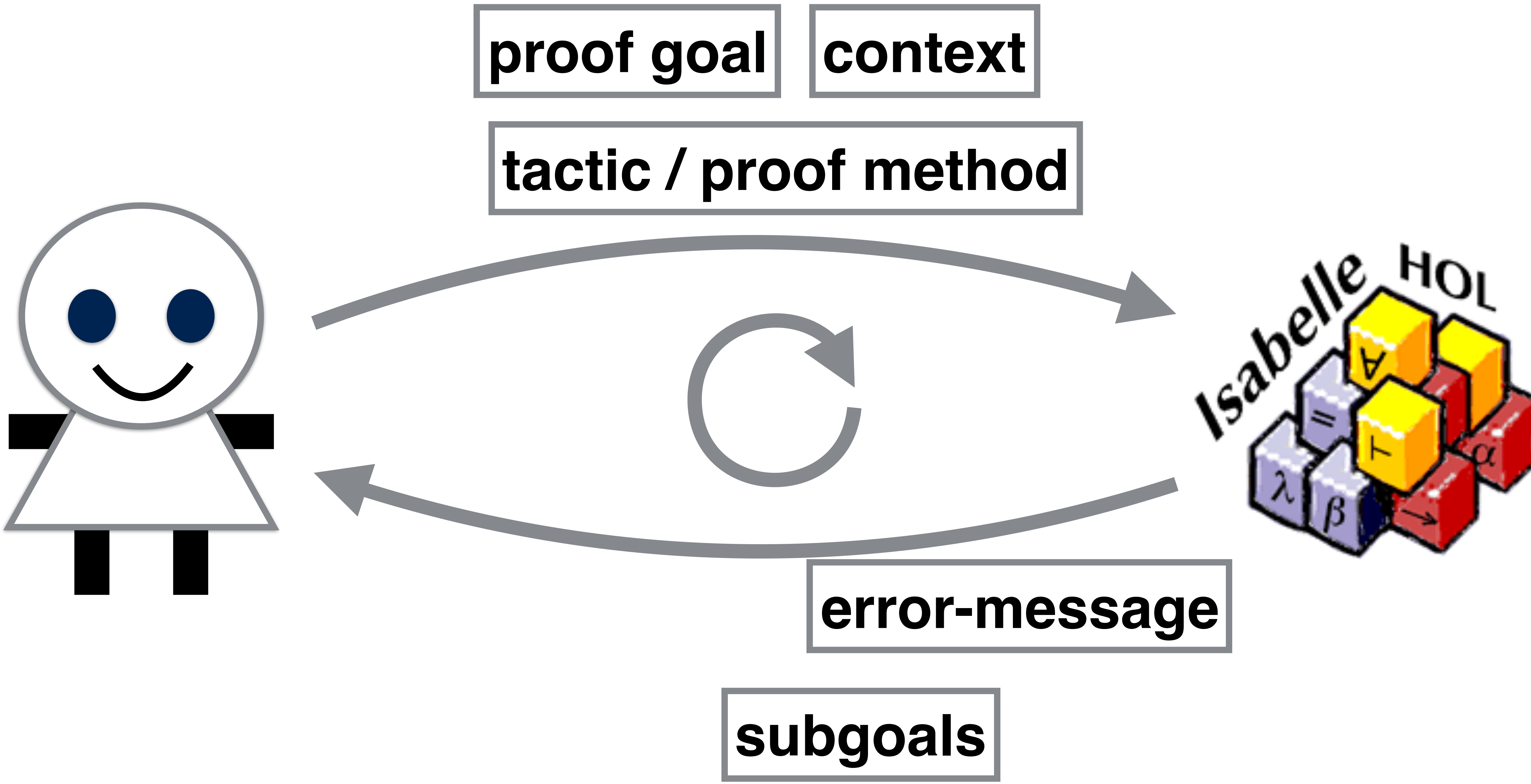
tactic / proof method



Interactive theorem proving with Isabelle/HOL



Interactive theorem proving with Isabelle/HOL

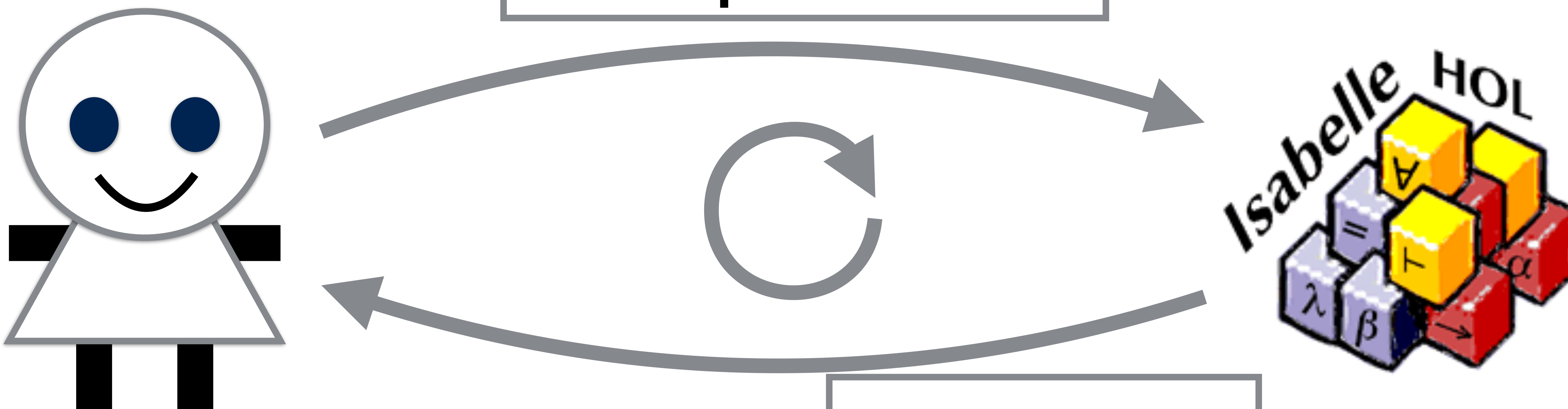


https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

Interactive theorem proving with Isabelle/HOL

proof goal **context**

tactic / proof method

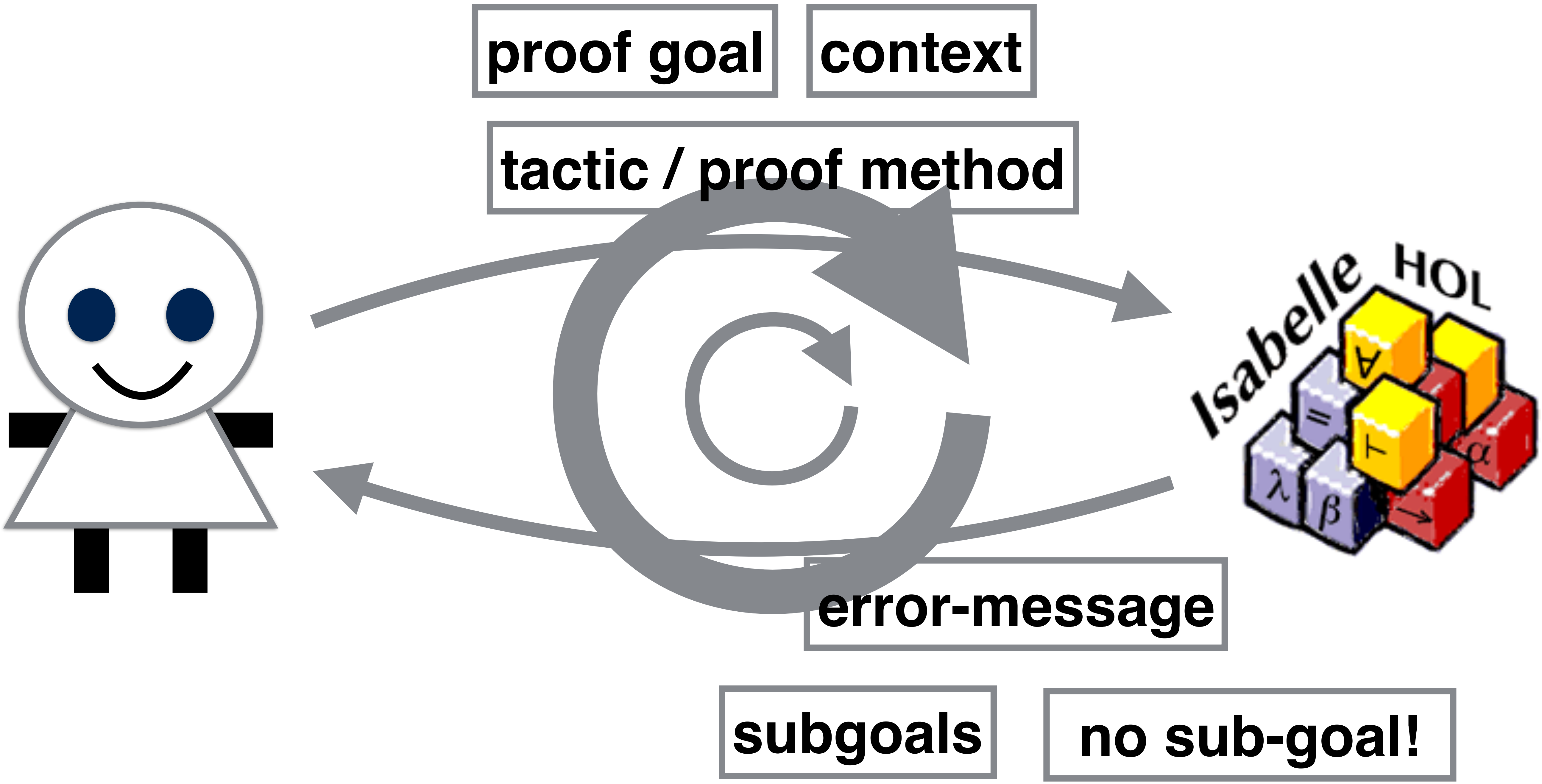


error-message

subgoals **no sub-goal!**

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

Interactive theorem proving with Isabelle/HOL

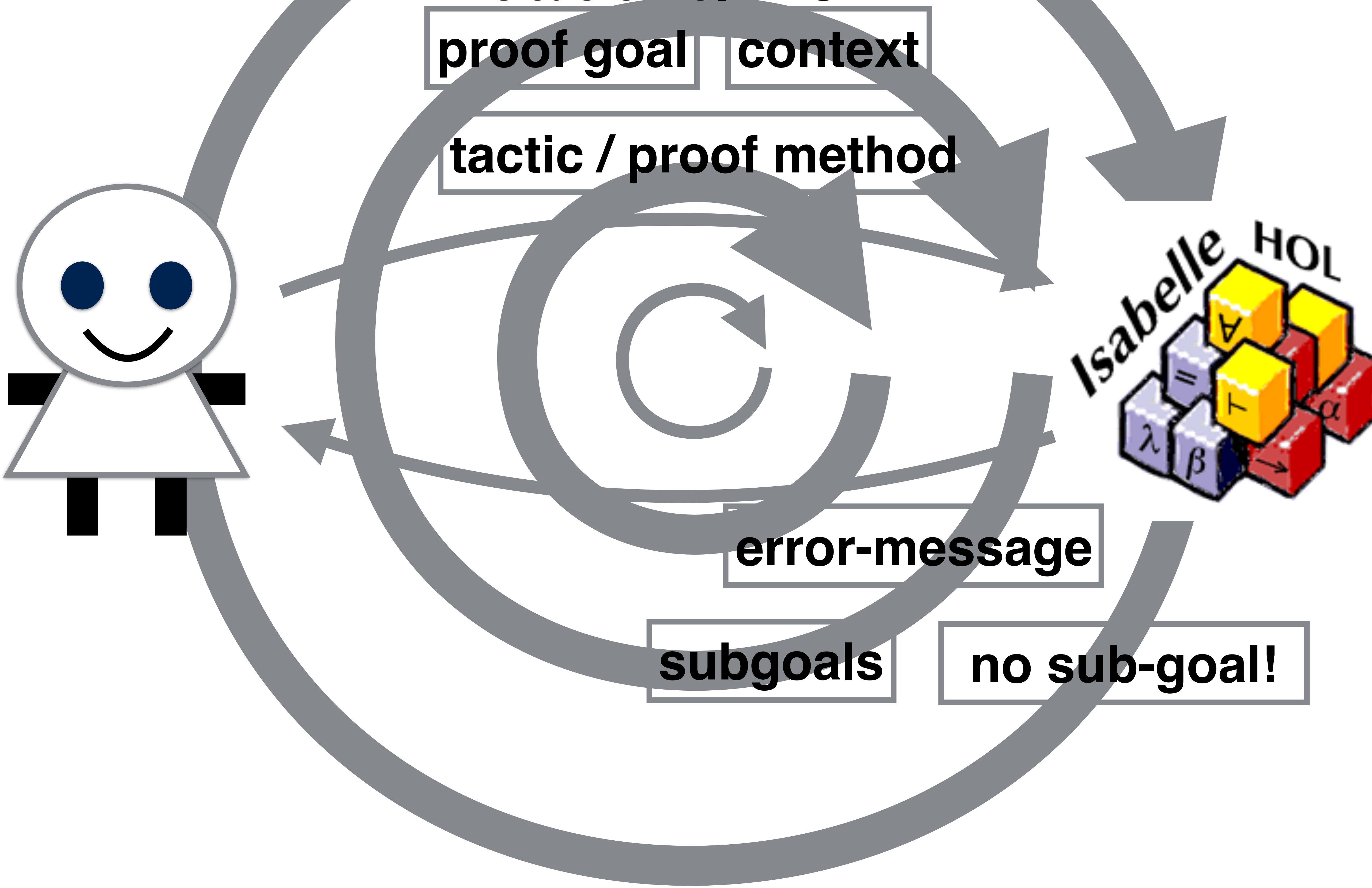


https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

Interactive theorem proving with Isabelle/HOL

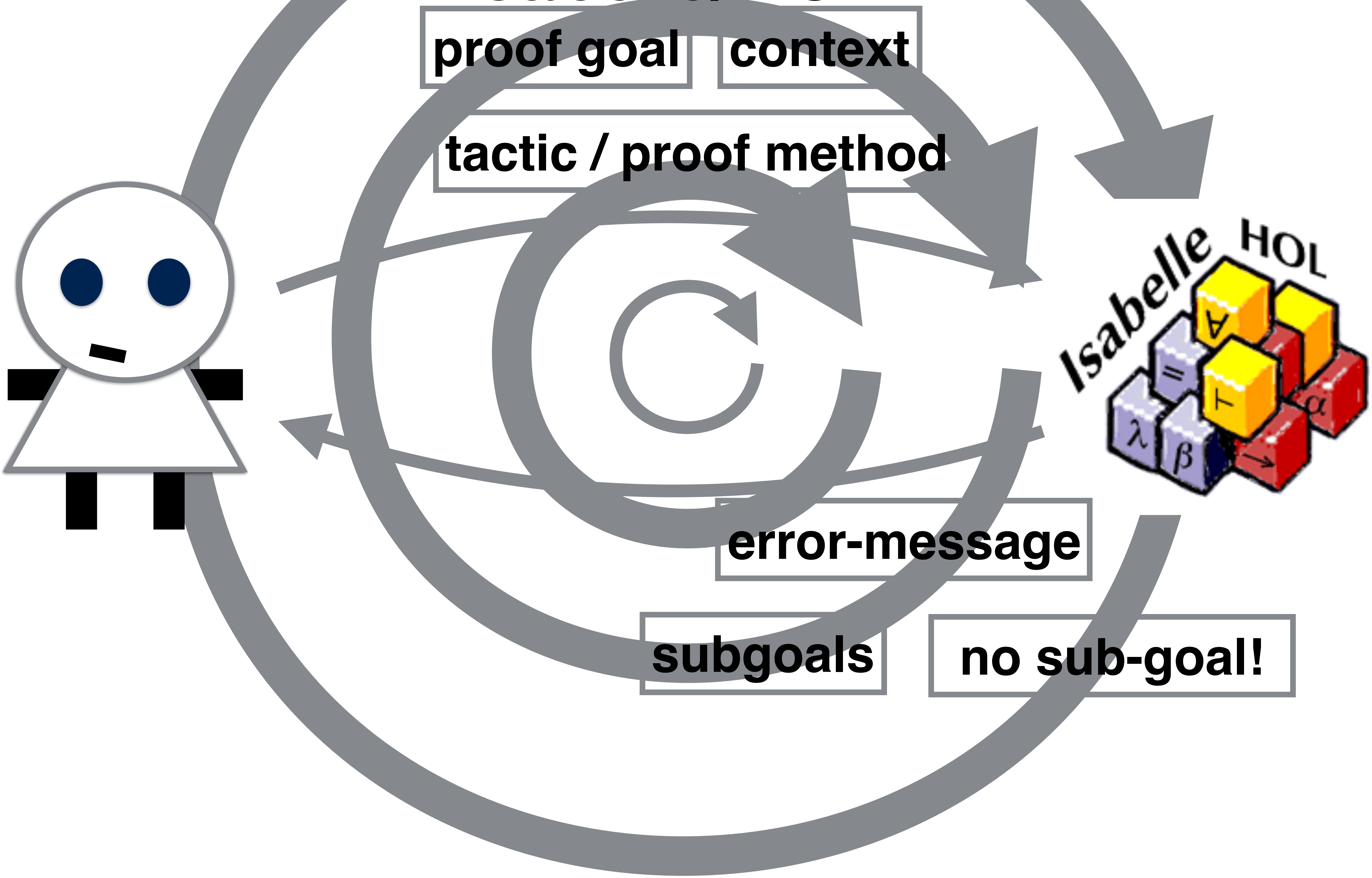
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

Interactive theorem proving with Isabelle/HOL

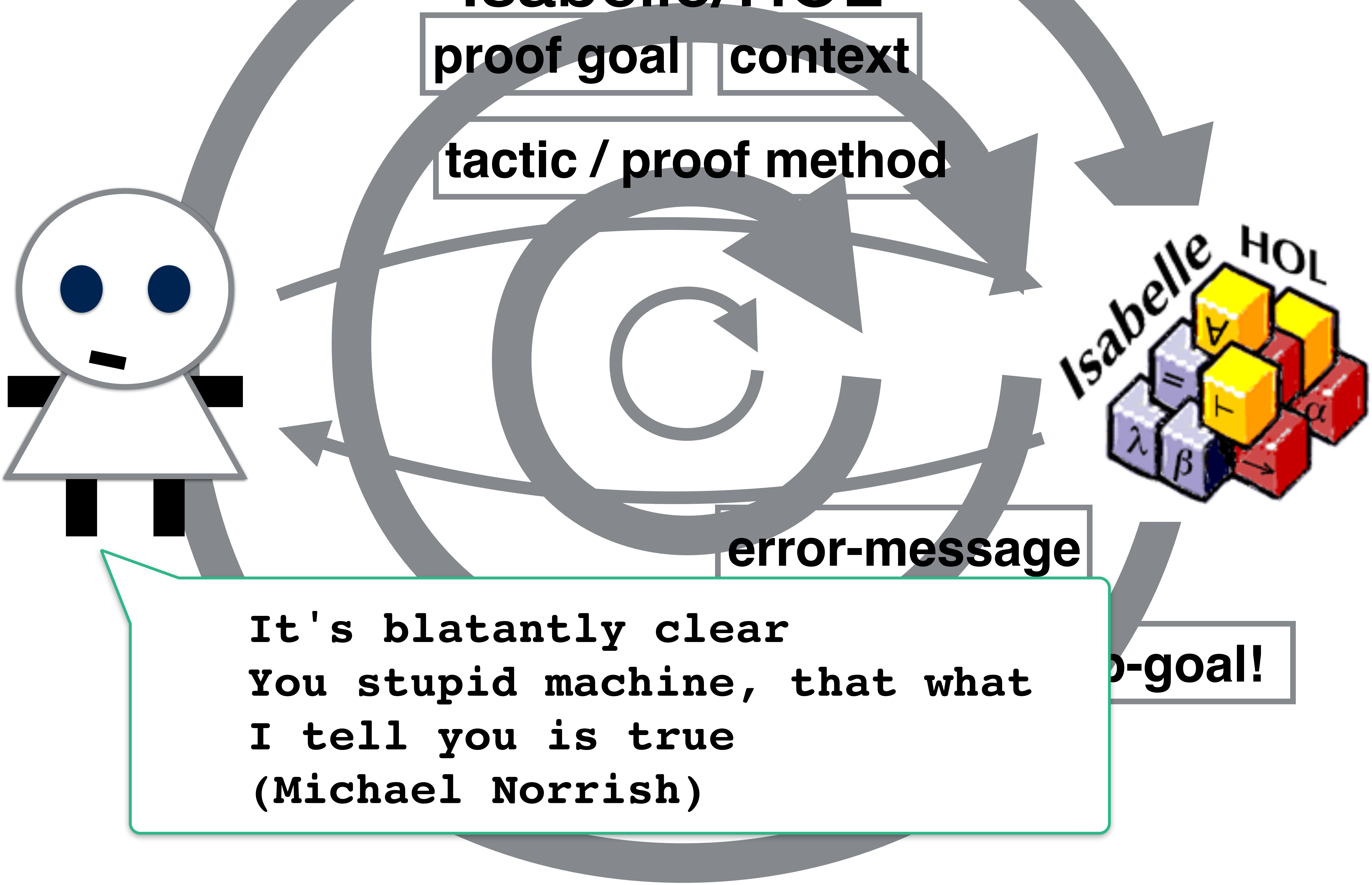


https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

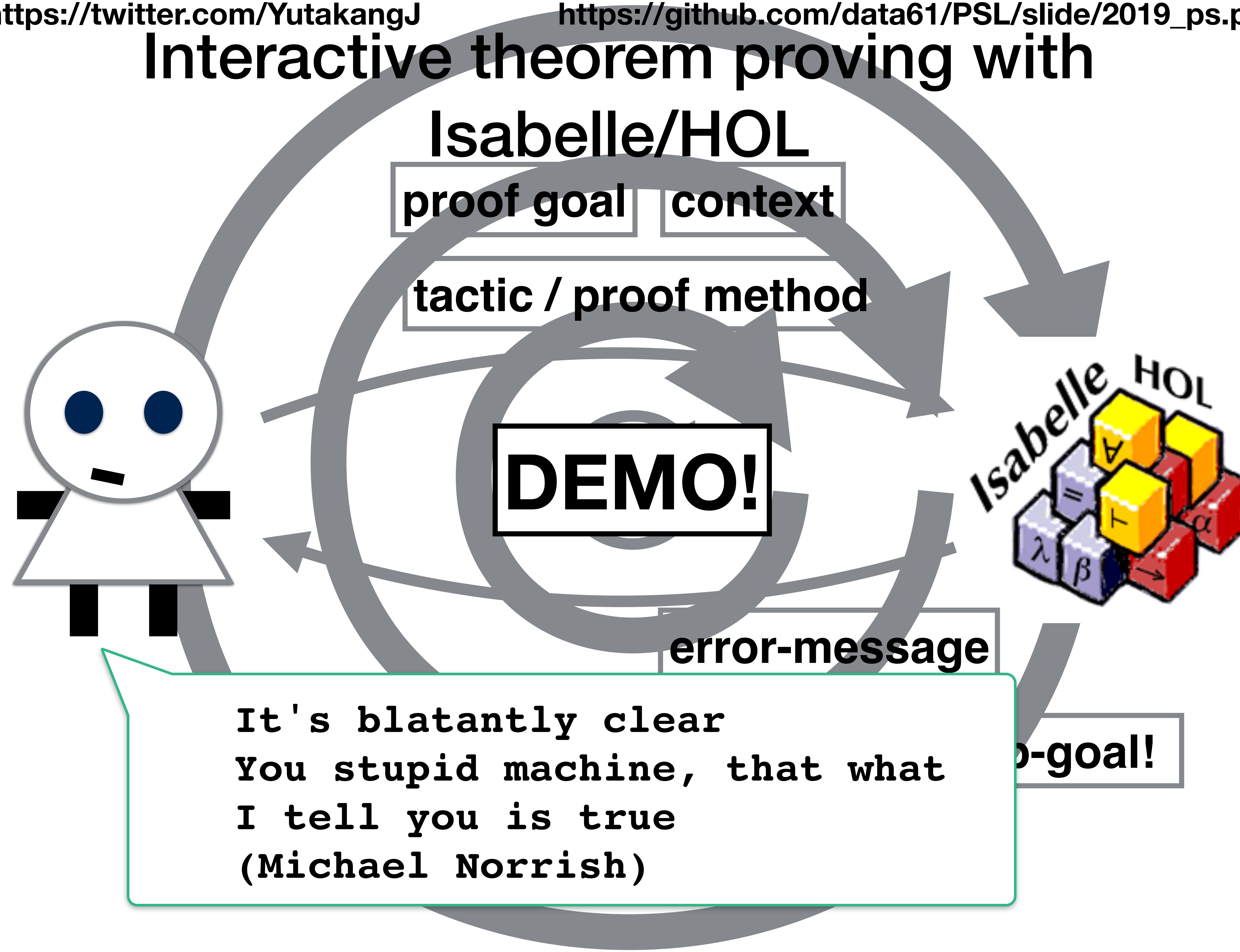
Interactive theorem proving with Isabelle/HOL



Interactive theorem proving with Isabelle/HOL



Interactive theorem proving with Isabelle/HOL



Example proof at Data61

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

```
lemma performPageTableInvocationUnmap_ccorres:
  "ccorres (K (K \<bottom>) \<currency> dc) (liftxf errstate id (K ()) ret__unsigned_long_')
       (invs' and cte_wp_at' (diminished' (ArchObjectCap cap) \<circ> cteCap) ctSlot
              and (\<lambda>_. isPageTableCap cap))
       (UNIV \<inter> \<lbrace>ccap_relation (ArchObjectCap cap) \<acute>cap\<rbrace> \<inter> \<lbrace>\<acute>ctSlot
       []
       (liftE (performPageTableInvocation (PageTableUnmap cap ctSlot)))
       (Call performPageTableInvocationUnmap_'proc)"
  apply (simp only: liftE_liftM ccorres_liftM_simp)
  apply (rule ccorres_gen_asm)
  apply (cinit lift: cap_' ctSlot_')
   apply csymbr
   apply (simp del: Collect_const)
   apply (rule ccorres_split_nothrow_novcg_dc)
      apply (subgoal_tac "capPTMappedAddress cap
                           = (\<lambda>cp. if to_bool (capPTIsMapped_CL cp)
                              then Some (capPTMappedASID_CL cp, capPTMappedAddress_CL cp)
                              else None) (cap_page_table_cap_lift capa)")
       apply (rule ccorres_Cond_rhs)
        apply (simp add: to_bool_def)
        apply (rule ccorres_rhs_assoc)+
        apply csymbr
        apply csymbr
        apply csymbr
        apply csymbr
        apply (ctac add: unmapPageTable_ccorres)
          apply csymbr
          apply (simp add: storePTE_def swp_def)
          apply (ctac add: clearMemory_setObject_PTE_ccorres[unfolded dc_def])
         apply wp
        apply (simp del: Collect_const)
```

taken from:
https://github.com/seL4/seL4

Example proof at Data61

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

impressive!

interesting?

```
lemma performPageTableIn[illegible]ocationUnmap_ccorres:
  "ccorres (K (K \[illegible]) \<currency> dc) ([illegible]ate id (K ()) ret__unsigned_long_')
       (invs[illegible] (diminishe[illegible]ap cap) \<circ> cteCap) ctSlot
       [illegible]lambda>_. is[illegible]p))
       [illegible]nter> \<lbrace>cca[illegible]on (ArchObjectCap cap) \<acute>cap\<rbrace> \<inter> \<lbrace>\<acute>ctSlot

       (liftE (performPageTableInvocation (PageTableUnmap cap ctSlot)))
       (Call performPageTableInvocationUnmap_'proc)"
  apply (simp only: liftE_liftM ccorres_liftM_simp)
  apply (rule ccorres_gen_asm)
  apply (cinit lift: cap_' ctSlot_')
   apply csymbr
   apply (simp del: Collect_const)
   apply (rule ccorres_split_nothrow_novcg_dc)
      apply (subgoal_tac "capPTMappedAddress cap
                          = (\<lambda>cp. if to_bool (capPTIsMapped_CL cp)
                             then Some (capPTMappedASID_CL cp, capPTMappedAddress_CL cp)
                             else None) (cap_page_table_cap_lift capa)")
       apply (rule ccorres_Cond_rhs)
        apply (simp add: to_bool_def)
        apply (rule ccorres_rhs_assoc)+
        apply csymbr
        apply csymbr
        apply csymbr
        apply csymbr
        apply (ctac add: unmapPageTable_ccorres)
          apply csymbr
          apply (simp add: storePTE_def swp_def)
          apply (ctac add: clearMemory_setObject_PTE_ccorres[unfolded dc_def])
         apply wp
        apply (simp del: Collect_const)
```

taken from:
https://github.com/seL4/seL4

Example proof at Data61

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

impressive!

~~interesting?~~

```
lemma performPageTableInvocationUnmap_ccorres:
  "ccorres (K (K \[illegible]) \<currency> dc) (\[illegible]ate id (K ()) ret__unsigned_long_')
       (invs[illegible] (diminishe[illegible]ap cap) \<circ> cteCap) ctSlot
       [illegible]ambda>_. is[illegible]p))
       [illegible]nter> \<lbrace>cc[illegible]on (ArchObjectCap cap) \<acute>cap\<rbrace> \<inter> \<lbrace>\<acute>ctSlot

       (liftE (performPageTableInvocation (PageTableUnmap cap ctSlot)))
       (Call performPageTableInvocationUnmap_'proc)"
  apply (simp only: liftE_liftM ccorres_liftM_simp)
  apply (rule ccorres_gen_asm)
  apply (cinit lift: cap_' ctSlot_')
   apply csymbr
   apply (simp del: Collect_const)
   apply (rule ccorres_split_nothrow_novcg_dc)
      apply (subgoal_tac "capPTMappedAddress cap
                           = (\<lambda>cp. if to_bool (capPTIsMapped_CL cp)
                              then Some (capPTMappedASID_CL cp, capPTMappedAddress_CL cp)
                              else None) (cap_page_table_cap_lift capa)")
       apply (rule ccorres_Cond_rhs)
        apply (simp add: to_bool_def)
        apply (rule ccorres_rhs_assoc)+
        apply csymbr
        apply csymbr
        apply csymbr
        apply csymbr
        apply (ctac add: unmapPageTable_ccorres)
          apply csymbr
          apply (simp add: storePTE_def swp_def)
          apply (ctac add: clearMemory_setObject_PTE_ccorres[unfolded dc_def])
         apply wp
        apply (simp del: Collect_const)
```

taken from:
https://github.com/seL4/seL4

Example proof at Data61

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

impressive!

~~interesting?~~

```
lemma performPageTableInvocationUnmap_ccorres:
  "ccorres (K (K \        ) \<currency> dc) (            ate id (K ()) ret__unsigned_long_')
       (invs'        (diminishe              ap cap) \<circ> cteCap) ctSlot
                    ambda>_. is                    p))
              inter> \<lbrace>cc              on (ArchObjectCap cap) \<acute>cap\<rbrace> \<inter> \<lbrace>\<acute>ctSlot

       (liftE (performPageTableInvocation (PageTableUnmap cap ctSlot)))
       (Call performPageTableInvocationUnmap_'proc)"
  apply (simp only: liftE_liftM ccorres_liftM_simp)
  apply (rule ccorres_gen_asm)
  apply (cinit lift: cap_' ctSlot_')
   apply csymbr
   apply (simp del: Collect_const)
   apply (rule ccorres_split_nothrow_novcg_dc)
      apply (subgoal_tac "capPTMappedAddress cap
                           = (\<lambda>cp. if to_bool (capPTIsMapped_CL cp)
                              then Some (capPTMappedASID_CL cp, capPTMappedAddress_CL cp)
                              else None) (cap_page_table_cap_lift capa)")
                     ccorres_Cond_rhs)
                    mp add: to_bool_def)
                      ccorres_rhs_assoc)+
                     br
             ply    ymbr
             ply    symbr
         apply csymbr
         apply (ctac add: unmapPageTable_ccorres)
           apply csymbr
           apply (simp add: storePTE_def swp_def)
           apply (ctac add: clearMemory_setObject_PTE_ccorres[unfolded dc_def])
          apply wp
         apply (simp del: Collect_const)
```

taken from:
https://github.com/seL4/seL4

Example proof at Data61

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

```
lemma performPageTableInvocationUnmap_ccorres:
  "ccorres (K (K \ ) \<currency> dc) ( ate id (K ()) ret__unsigned_long_')
       (invs  (diminishe ap cap) \<circ> cteCap) ctSlot
           ambda>_. is p))
           nter> \<lbrace>cc on (ArchObjectCap cap) \<acute>cap\<rbrace> \<inter> \<lbrace>\<acute>ctSlot

       (liftE (performPageTableInvocation (P
       (Call performPageTableInvocationU
  apply (simp only: liftE_liftM ccorr
  apply (rule ccorres_gen_asm)
  apply (cinit lift: cap_' ctSlot_')
   apply csymbr
   apply (simp del: Collect_const)
   apply (rule ccorres_split_nothrow_novcg_
      apply (subgoal_tac "capPT dress cap
                        = (\< >cp. if to_bool (capPTIsMapped_CL cp)
                            en Some (capPTMappedASID_CL cp, capPTMappedAddress_CL cp)
                             else None) (cap_page_table_cap_lift capa)")
                ccorres_Cond_rhs)
               mp add: to_bool_def)
                ccorres_rhs_assoc)+
                  br
         ply  symbr
         ply  symbr
        apply csymbr
        apply (ctac add: unmapPageTable_ccorres)
          apply csymbr
          apply (simp add: storePTE_def swp_def)
          apply (ctac add: clearMemory_setObject_PTE_ccorres[unfolded dc_def])
         apply wp
        apply (simp del: Collect_const)
```

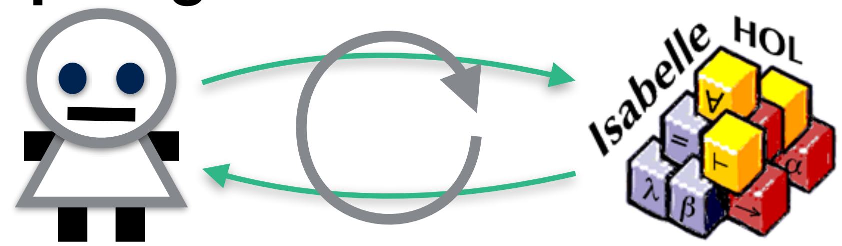
impressive!

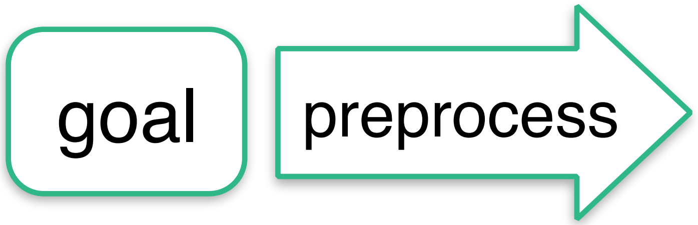
~~interesting?~~

これを人間が書くべきなの…？
どうにかしろよ、人工知能。

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

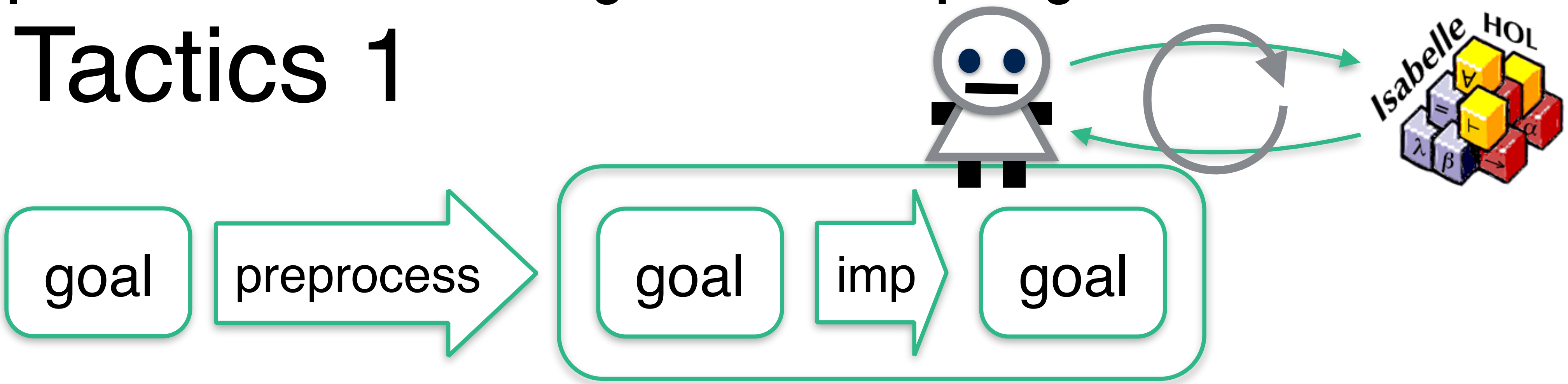
Tactics 1





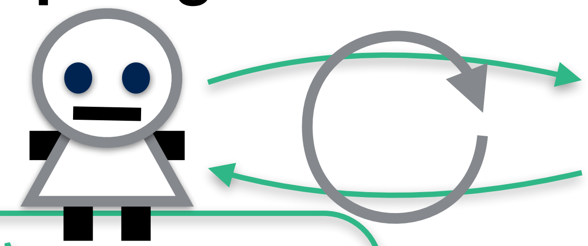
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

Tactics 1



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

Tactics 1



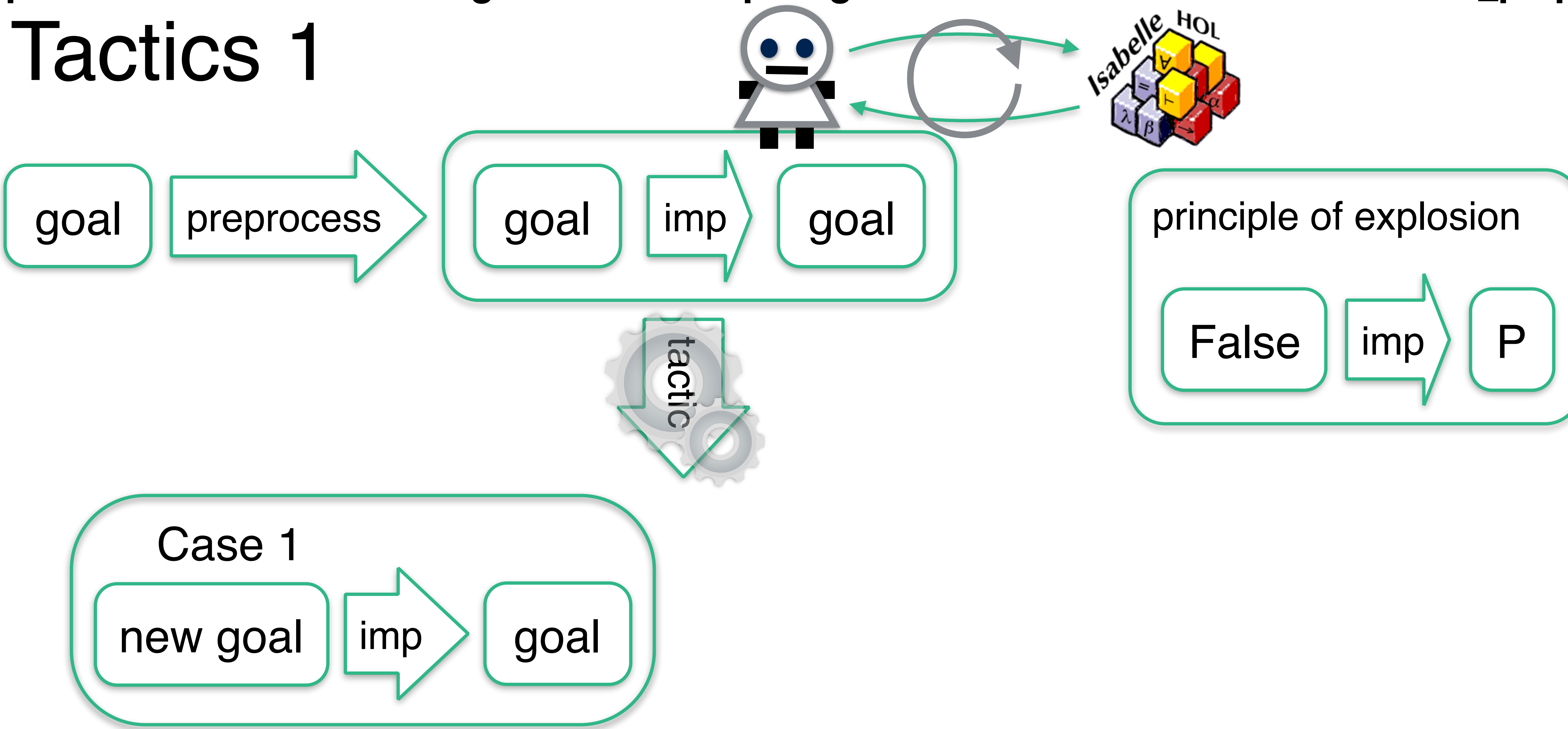
goal preprocess goal imp goal

principle of explosion

False imp P

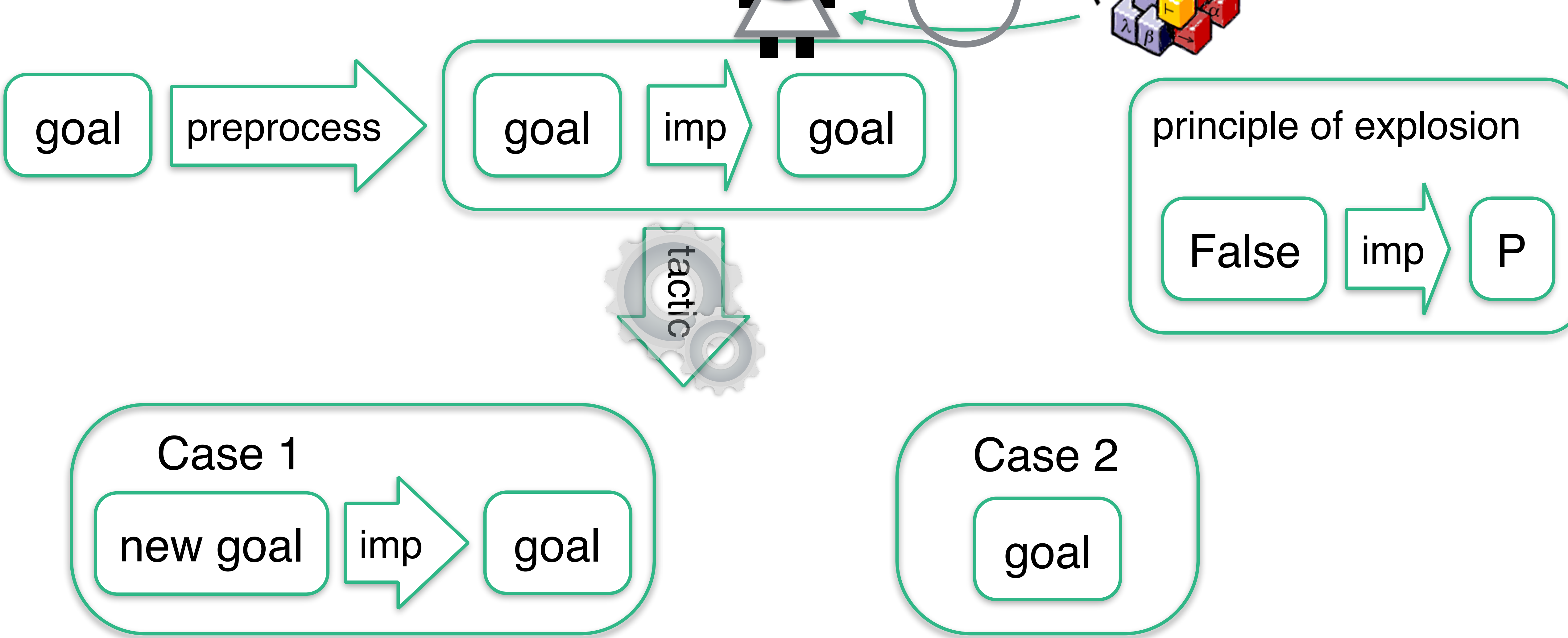
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

Tactics 1



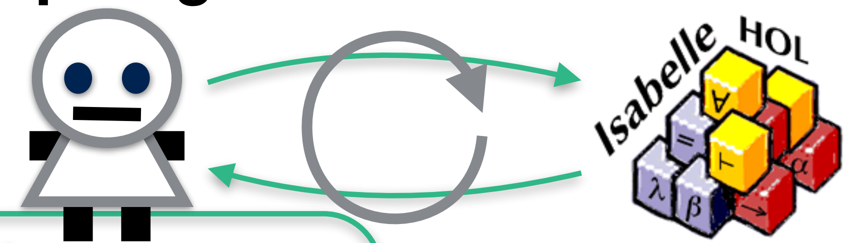
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

Tactics 1



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

Tactics 1



goal → preprocess → goal imp goal

principle of explosion

False imp P

tactic

Case 1

new goal imp goal

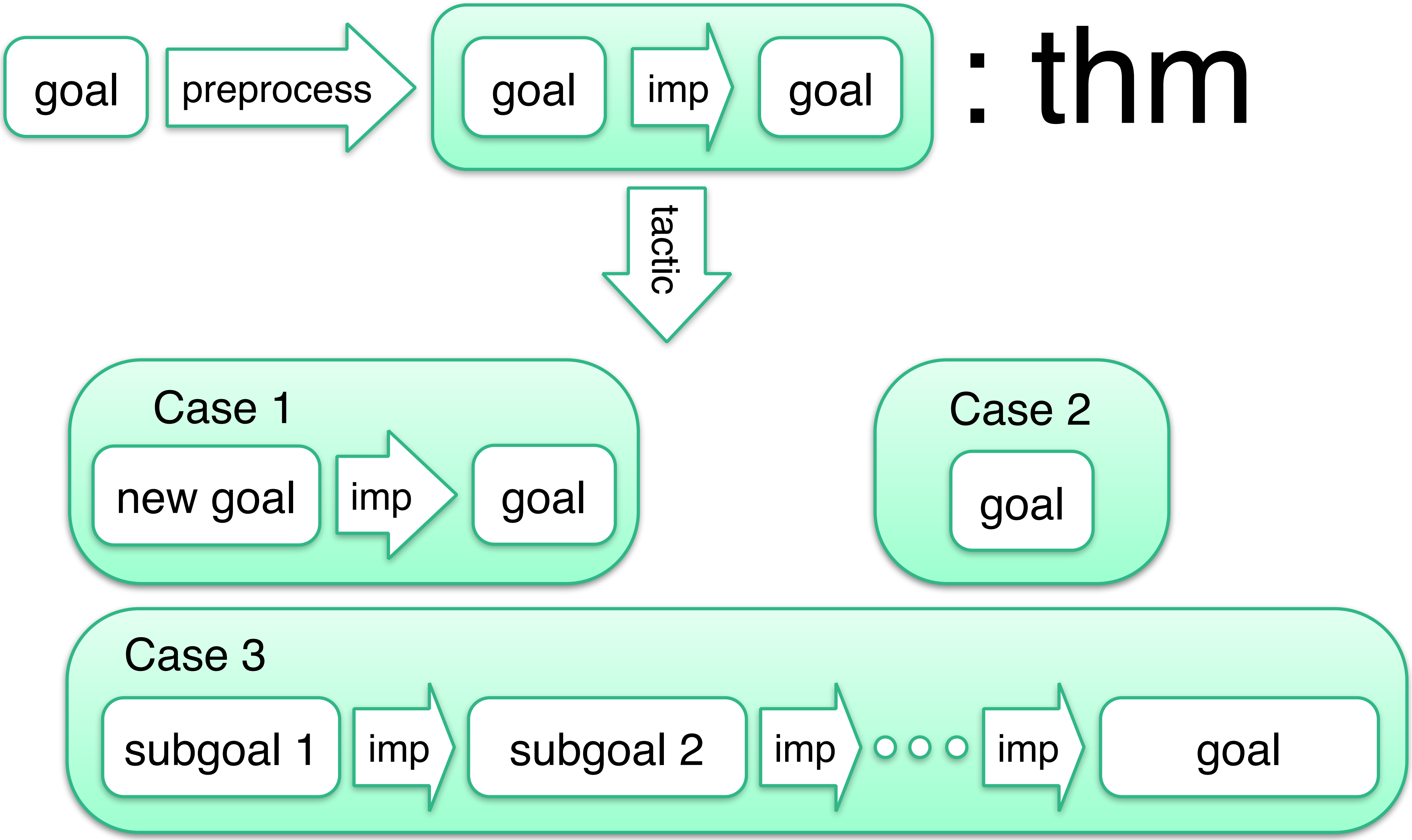
Case 2

goal

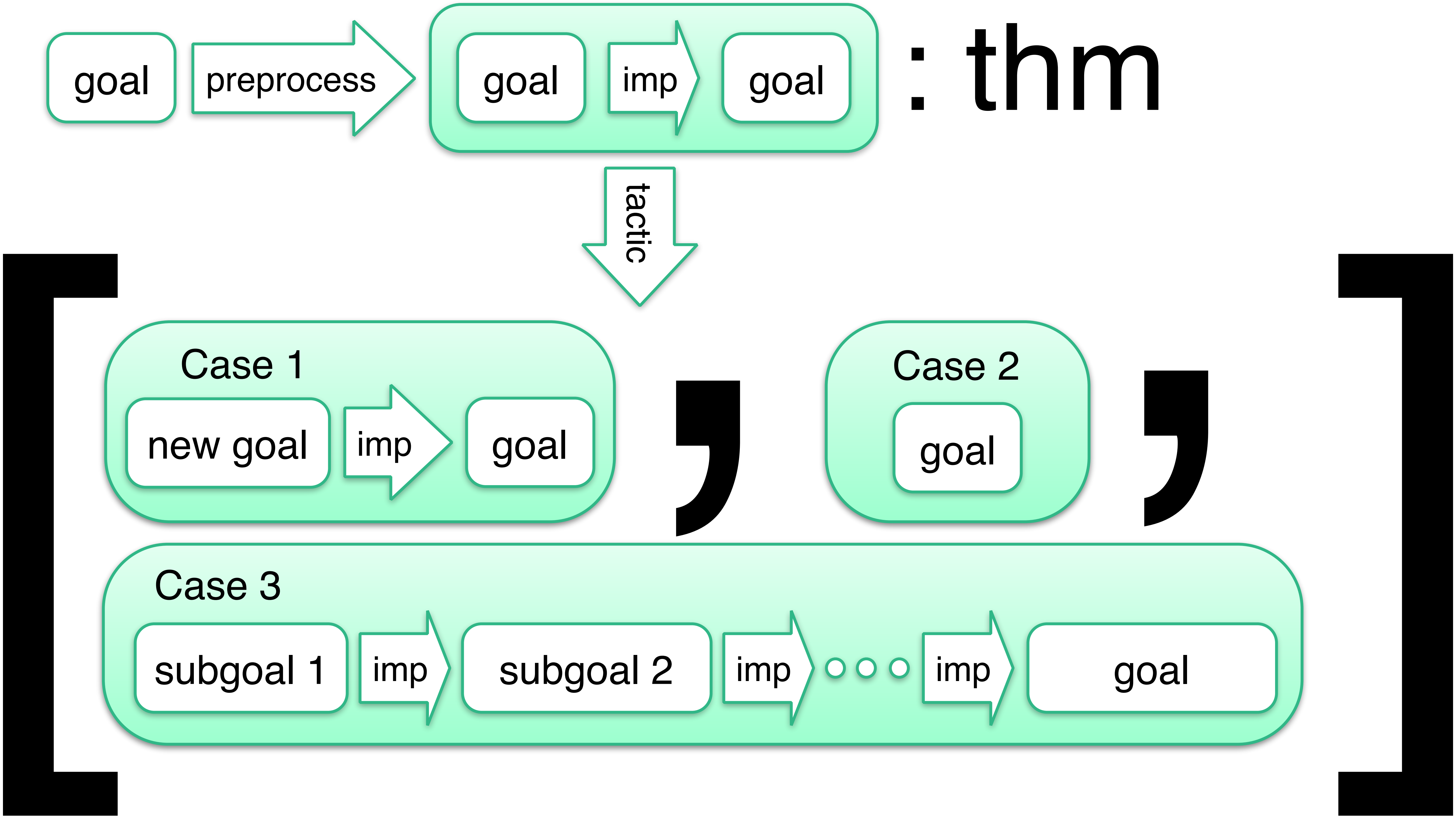
Case 3

subgoal 1 imp subgoal 2 imp ○ ○ ○ imp goal

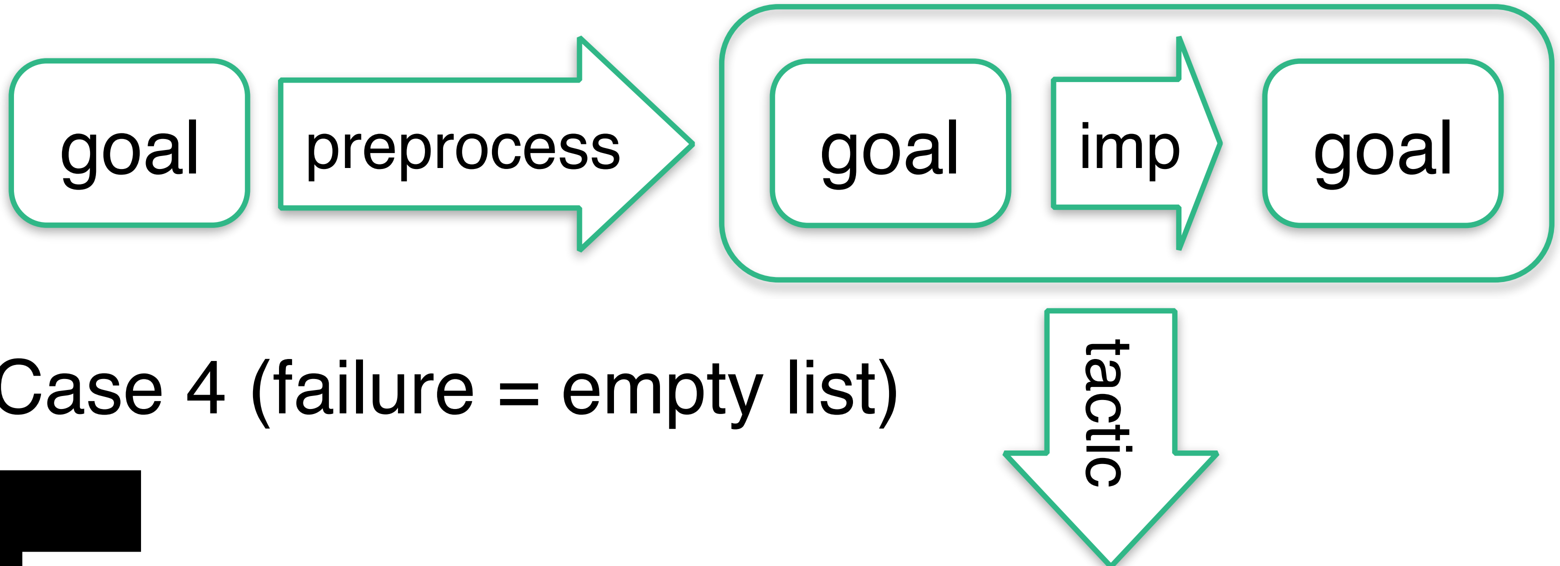
Tactics 2


goal
preprocess
goal
imp
goal
: thm
tactic
Case 1
new goal
imp
goal
Case 2
goal
Case 3
subgoal 1
imp
subgoal 2
imp
imp
goal

Tactics 2


goal
preprocess
goal
imp
goal
: thm
tactic
Case 1
new goal
imp
goal
Case 2
goal
Case 3
subgoal 1
imp
subgoal 2
imp
imp
goal

Tactics 2



Case 4 (failure = empty list)

[ ]

Tactics 3

$$(w \land x \Rightarrow y \land z \Rightarrow z)$$

Tactics 3

(w ∧ x => y ∧ z => z)

=>

(w ∧ x => y ∧ z => z)

Tactics 3

$$(w \land x \Rightarrow y \land z \Rightarrow z)$$

$$\Rightarrow$$

$$(w \land x \Rightarrow y \land z \Rightarrow z)$$

our original goal

our current proof obligation

Tactics 3

(w ∧ x => y ∧ z => z)

=>

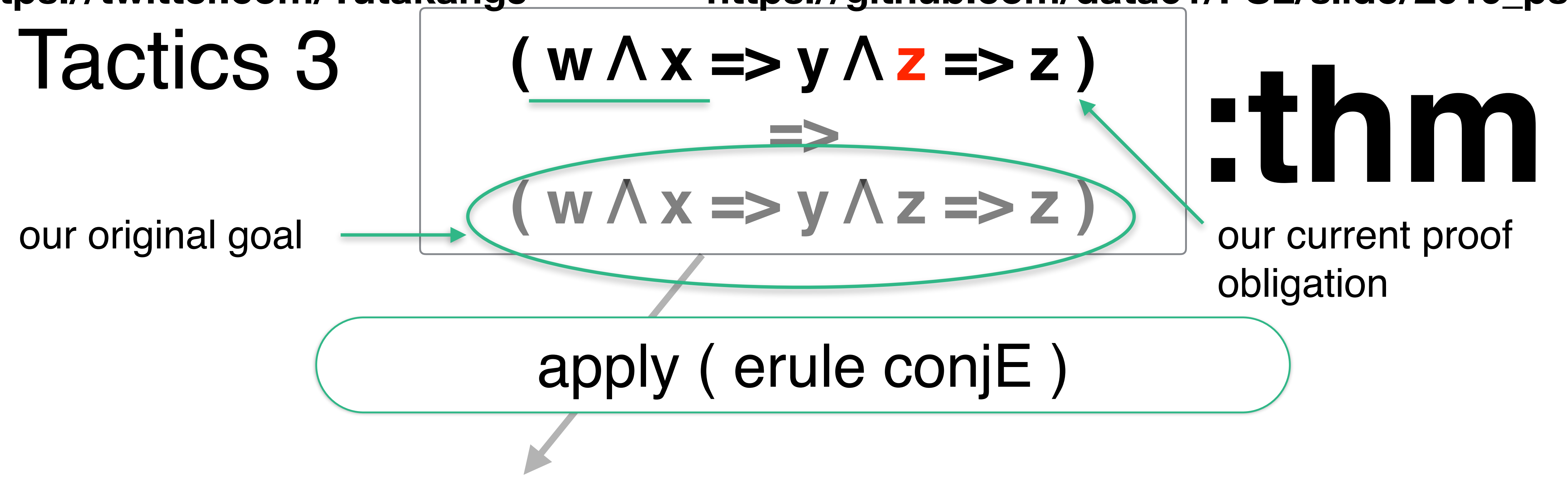
(w ∧ x => y ∧ z => z)

:thm

our original goal

our current proof obligation

Tactics 3


(w ∧ x => y ∧ z => z)
=>
(w ∧ x => y ∧ z => z)
:thm
our original goal
our current proof obligation
apply (erule conjE)

Tactics 3

(w ∧ x => y ∧ z => z)
=>
(w ∧ x => y ∧ z => z)

:thm

our original goal

our current proof obligation

apply (erule conjE)

[(y ∧ z => w => x => z)
=>
(w ∧ x => y ∧ z => z) ,]

Tactics 3

(w ∧ x => y ∧ z => z)
=>
(w ∧ x => y ∧ z => z)

:thm

our original goal

our current proof obligation

apply (erule conjE)

[(y ∧ z => w => x => z)
=>
(w ∧ x => y ∧ z => z) ,]

apply (assumption)

Tactics 3

(w ∧ x => y ∧ z => z)
=>
(w ∧ x => y ∧ z => z)

:thm

our original goal

our current proof obligation

apply (erule conjE)

[(y ∧ z => w => x => z)
=>
(w ∧ x => y ∧ z => z) ,]

apply (assumption)

[ ]

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

Tactics 3

(w ∧ x => y ∧ z => z)

=>

(w ∧ x => y ∧ z => z)

:thm

our original goal

our current proof obligation

apply (erule conjE)

back

[(y ∧ z => w => x => z)

=>

(w ∧ x => y ∧ z => z) **,]**

apply (assumption)

[ ]

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

Tactics 3

(w ∧ x => y ∧ z => z)
=>
(w ∧ x => y ∧ z => z)

:thm

our original goal

our current proof obligation

apply (erule conjE)

back

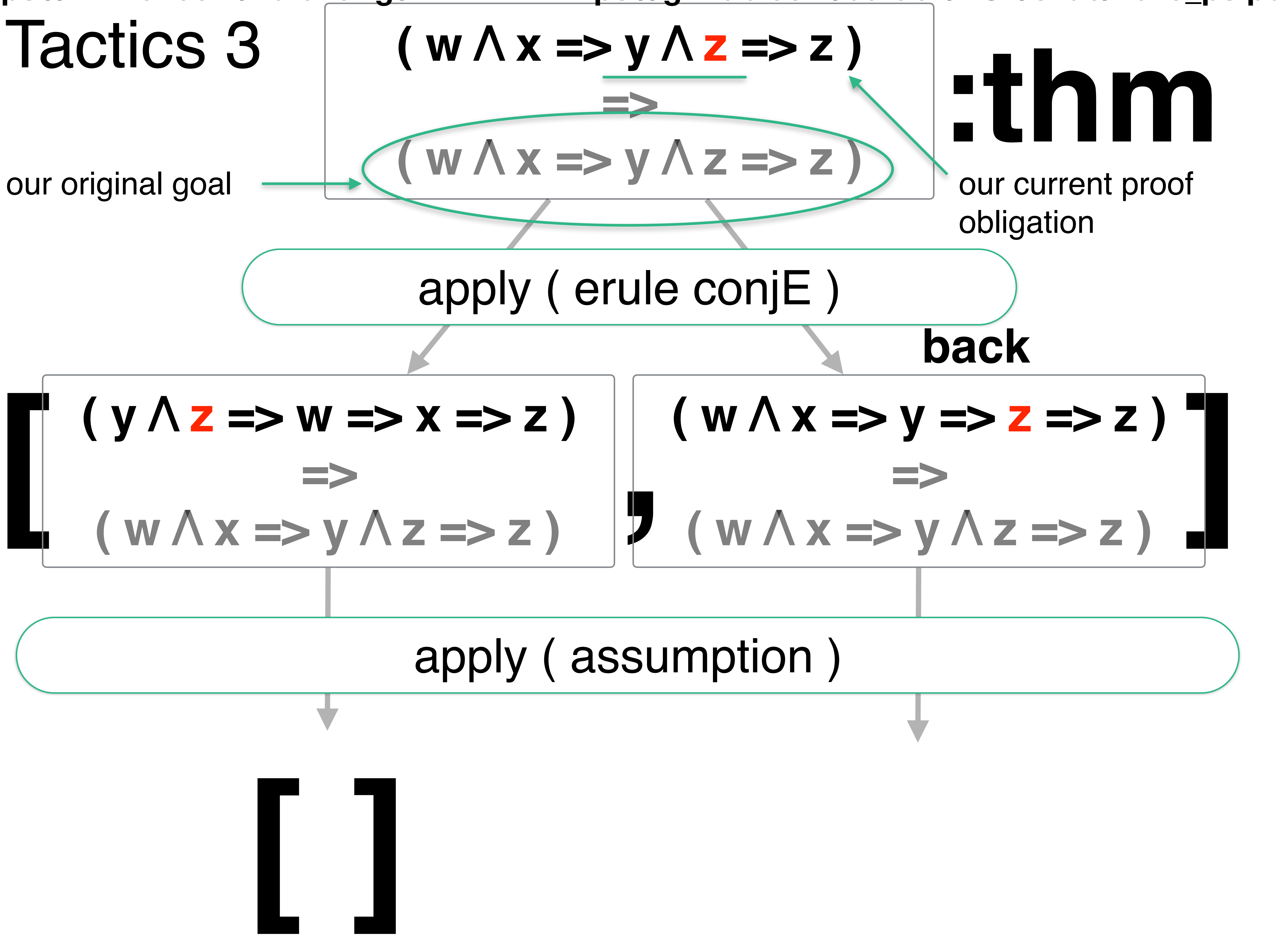
[(y ∧ z => w => x => z)
=>
(w ∧ x => y ∧ z => z) ,
(w ∧ x => y => z => z)
=>
(w ∧ x => y ∧ z => z)]

apply (assumption)

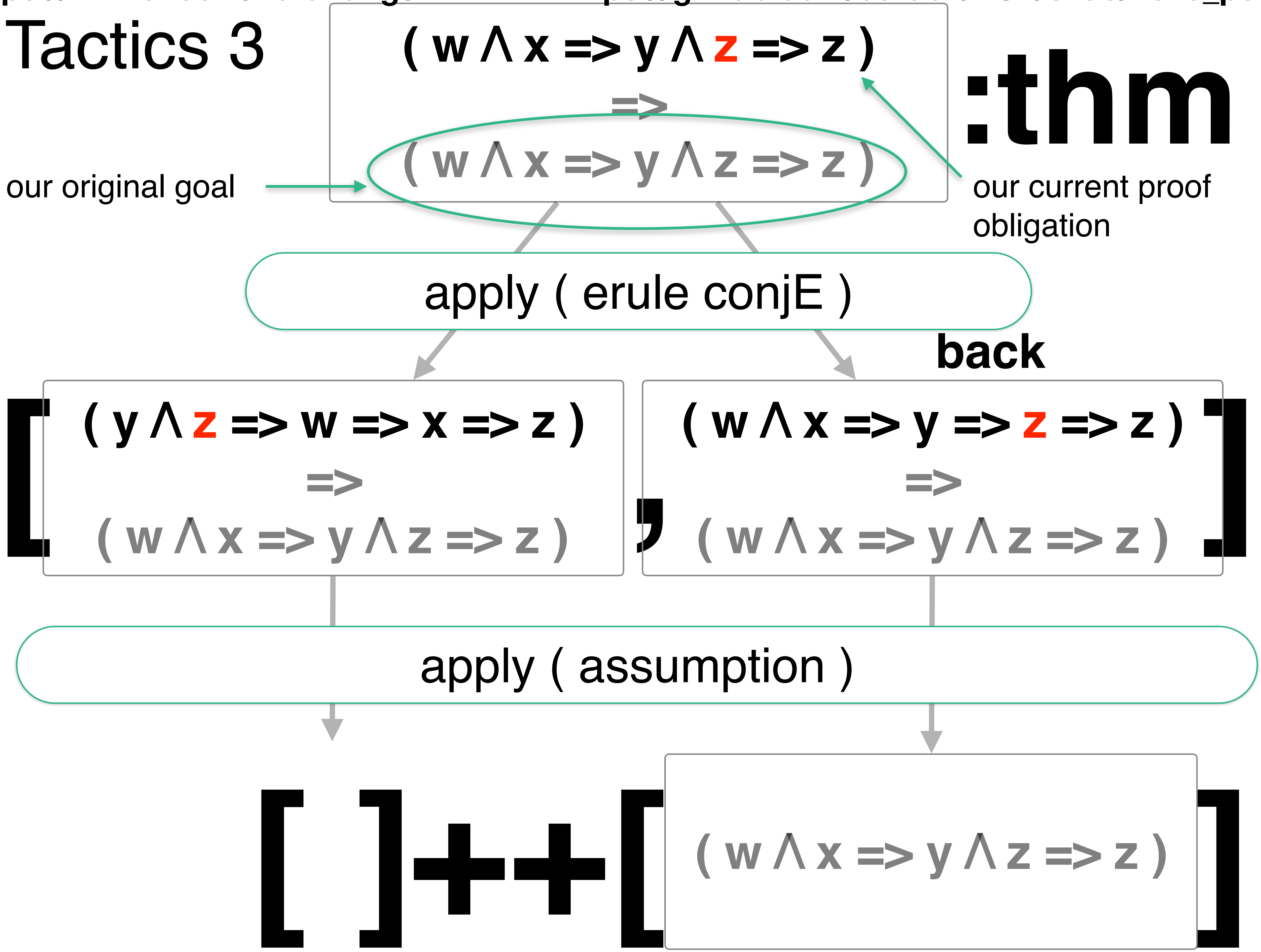
[ ]

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

Tactics 3



Tactics 3


(w ∧ x => y ∧ z => z)
=>
(w ∧ x => y ∧ z => z)
:thm
our original goal
our current proof obligation
apply (erule conjE)
back
[(y ∧ z => w => x => z) => (w ∧ x => y ∧ z => z) , (w ∧ x => y => z => z) => (w ∧ x => y ∧ z => z)]
apply (assumption)
[ ] ++ [(w ∧ x => y ∧ z => z)]

Tactics 3

:thm

(w ∧ x => y ∧ z => z)

=>

(w ∧ x => y ∧ z => z)

our original goal

our current proof obligation

apply (erule conjE)

[(y ∧ z => w => x => … ; … y => …]

apply (rule conjE, assumption)

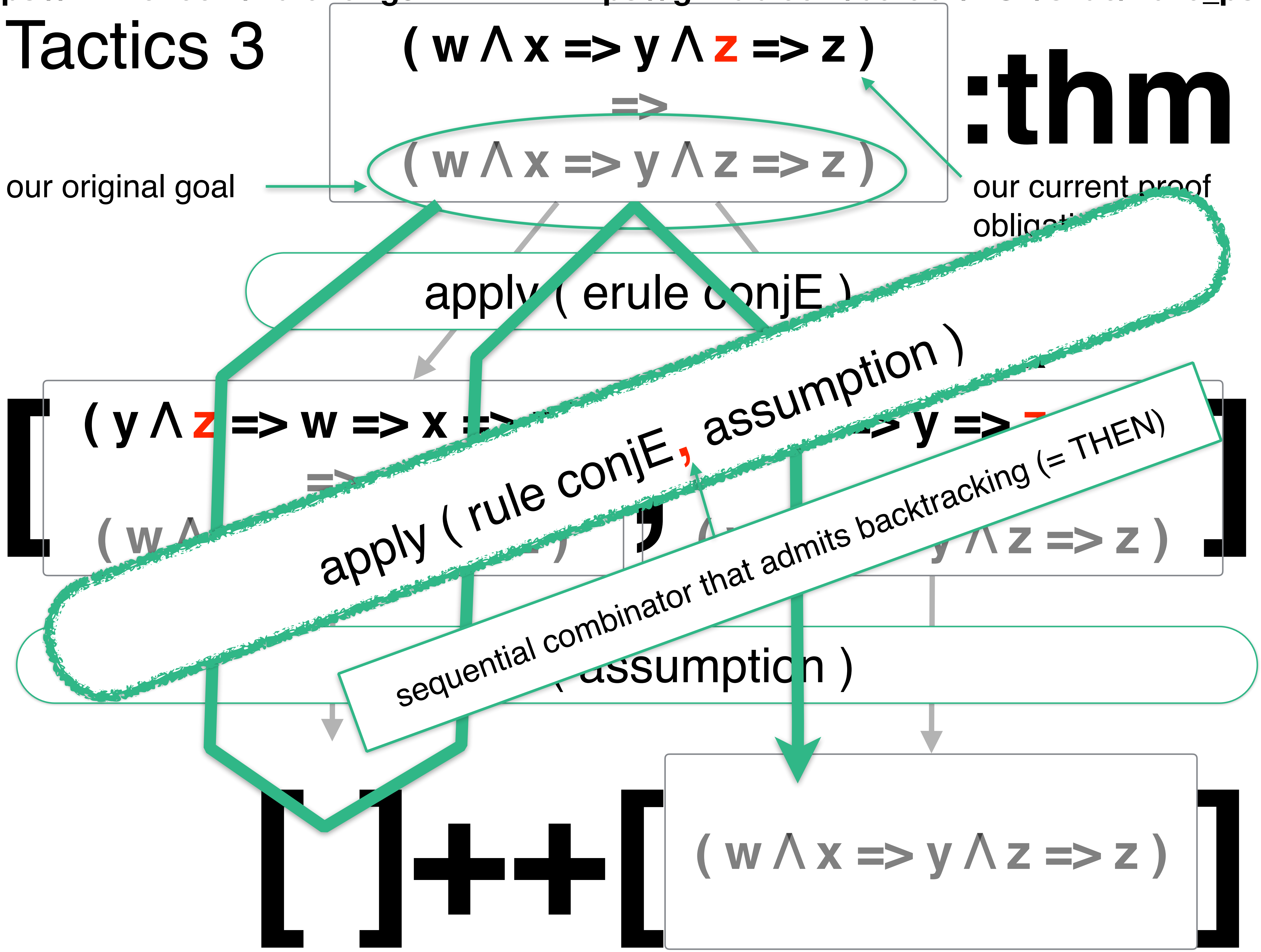
sequential combinator that admits backtracking (= THEN)

(assumption)

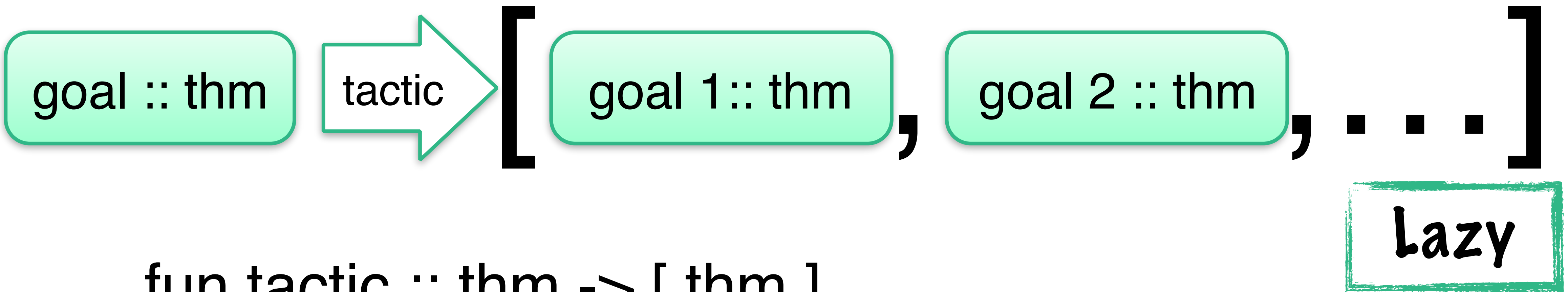
[ ]++[(w ∧ x => y ∧ z => z)]

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

Tactics 3

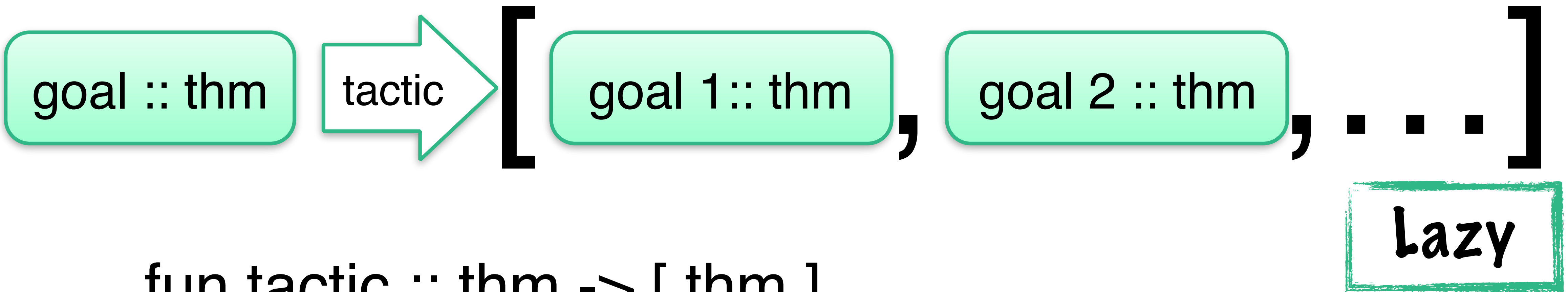


Tactics 4

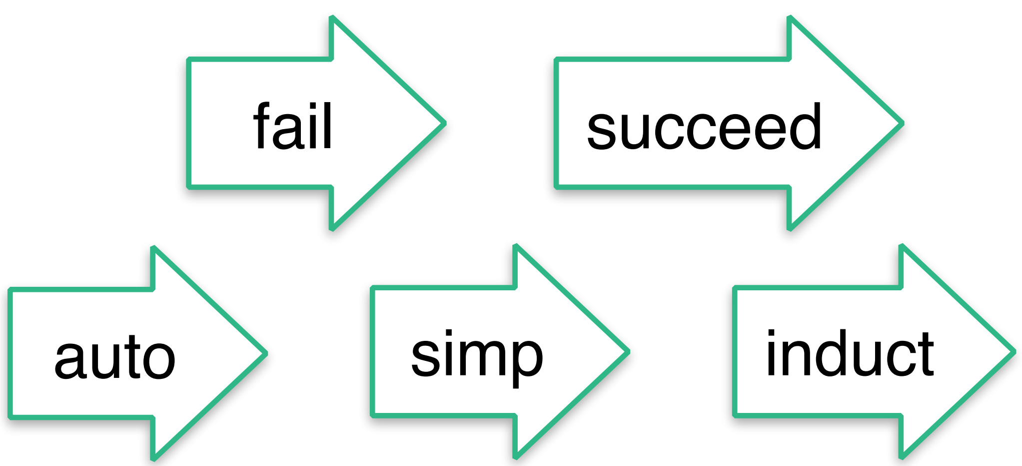


fun tactic :: thm -> [thm]

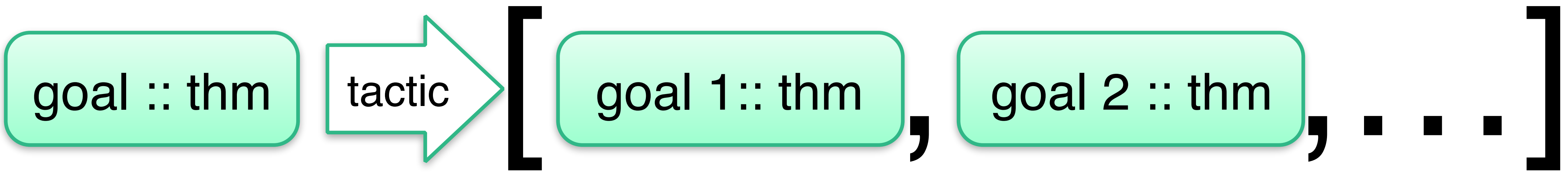
Tactics 4



fun tactic :: thm -> [thm]

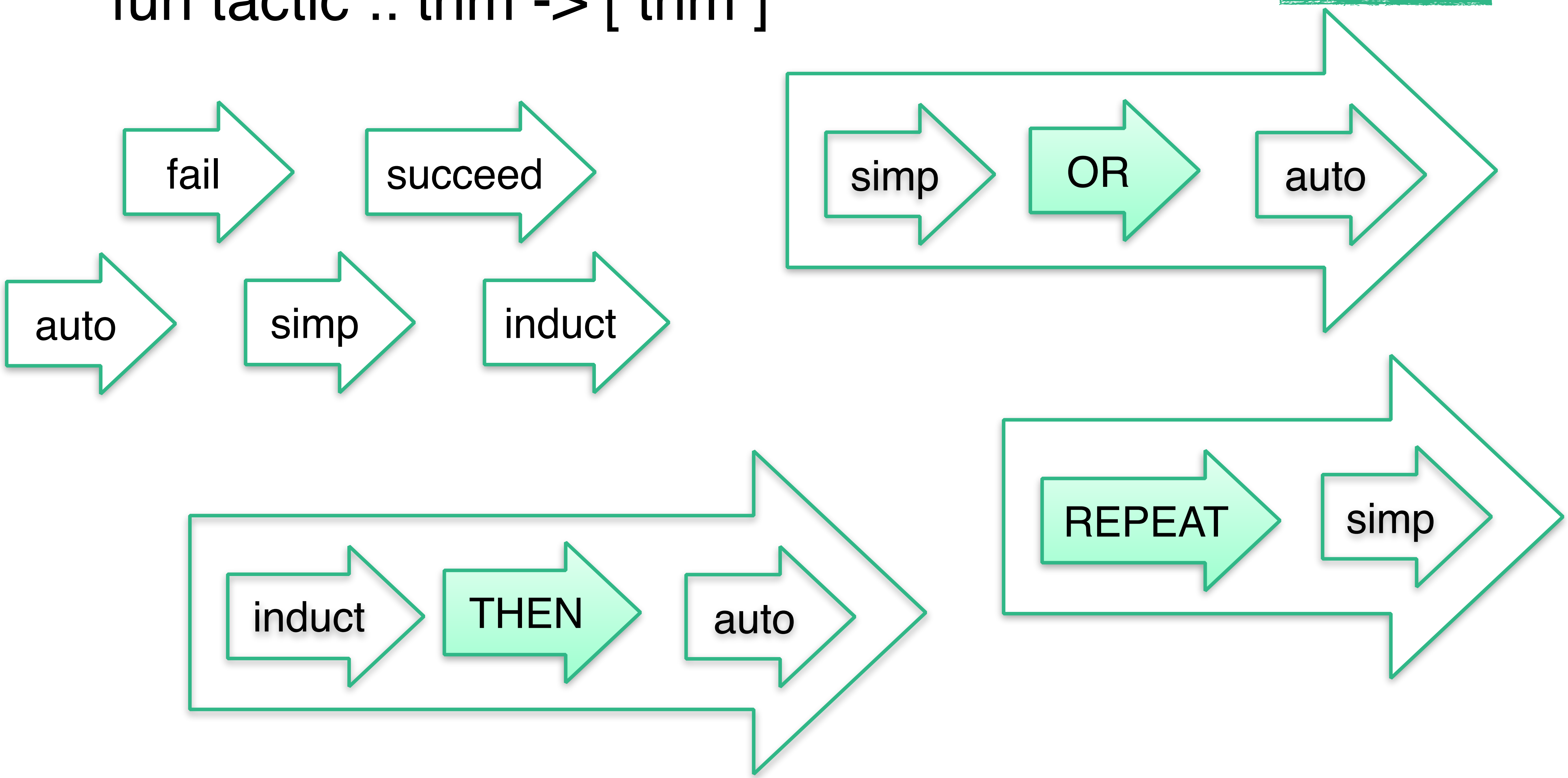


Tactics 4

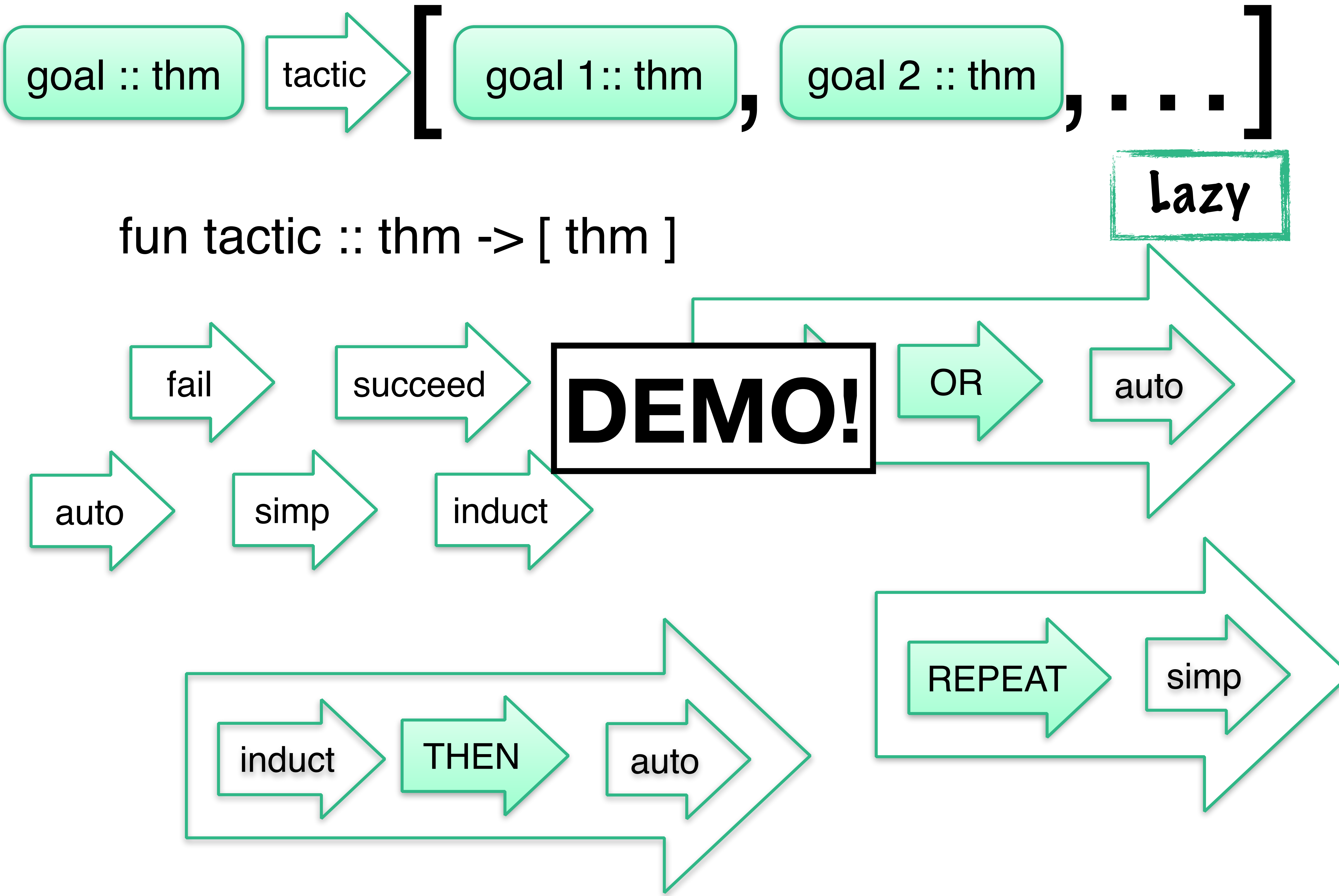


Lazy

fun tactic :: thm -> [thm]



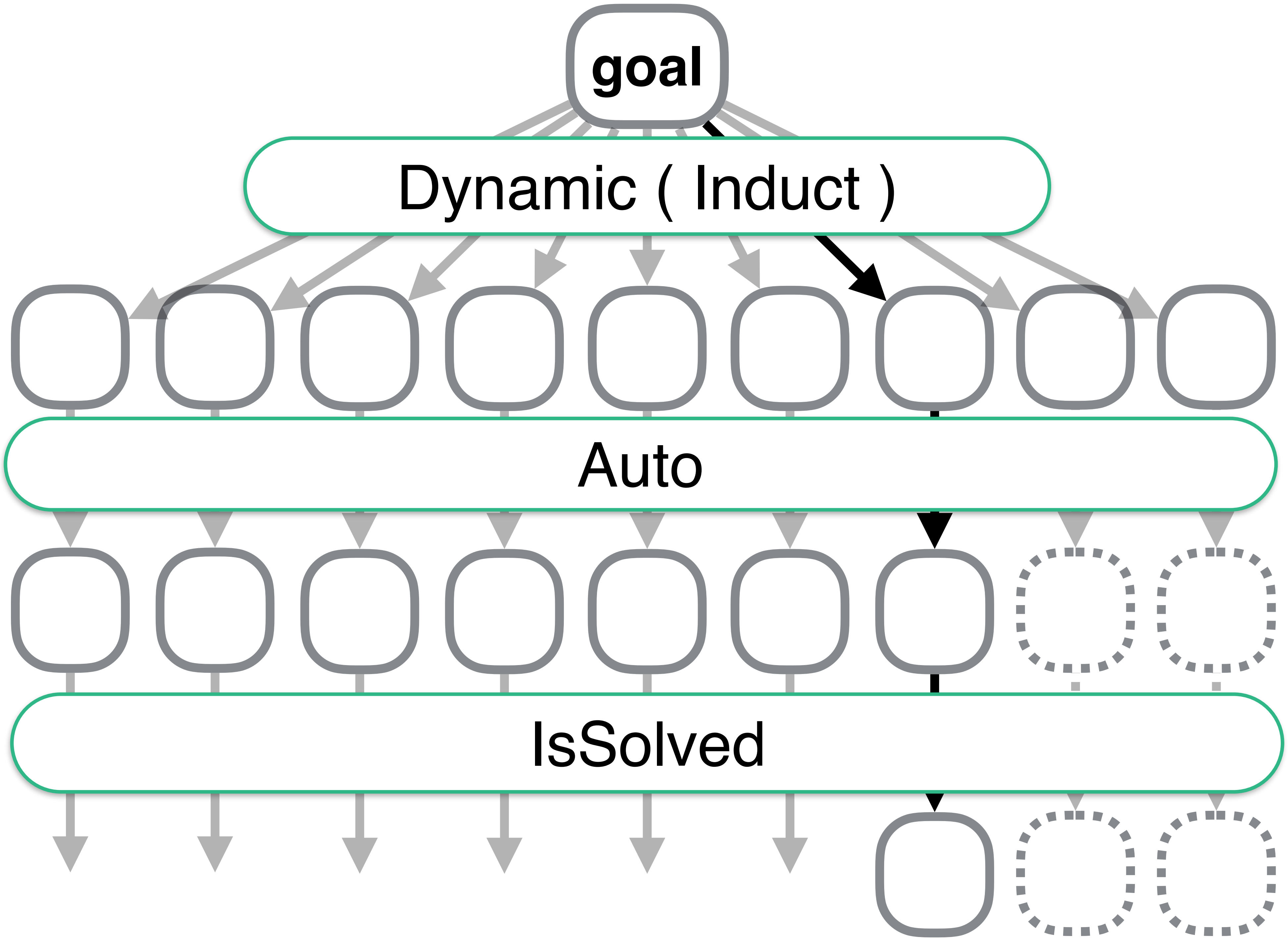
Tactics 4


goal :: thm
tactic
[goal 1:: thm , goal 2 :: thm , . . .]
Lazy
fun tactic :: thm -> [thm]
fail
succeed
DEMO!
OR
auto
auto
simp
induct
induct
THEN
auto
REPEAT
simp

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

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
```

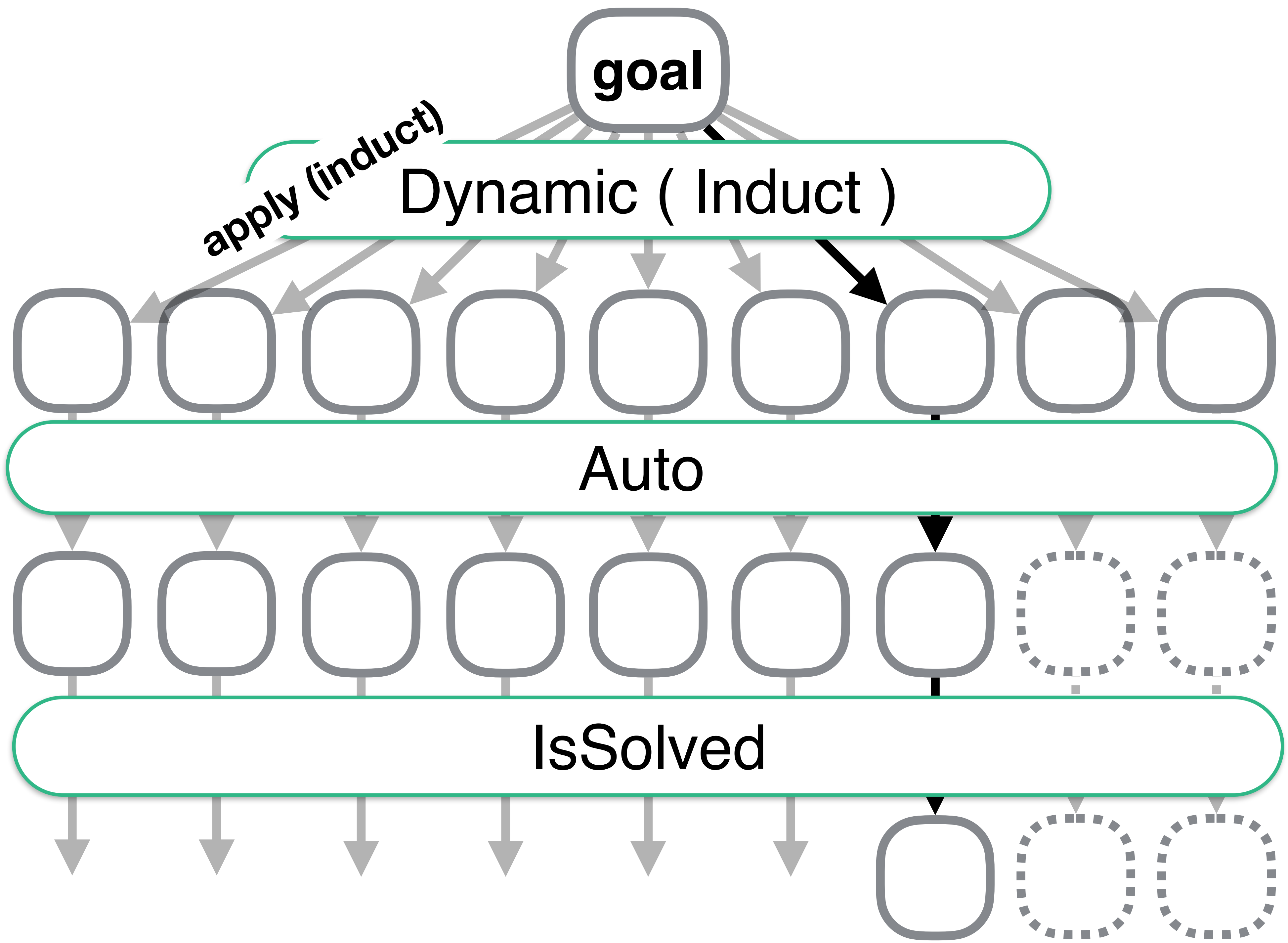
```
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



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

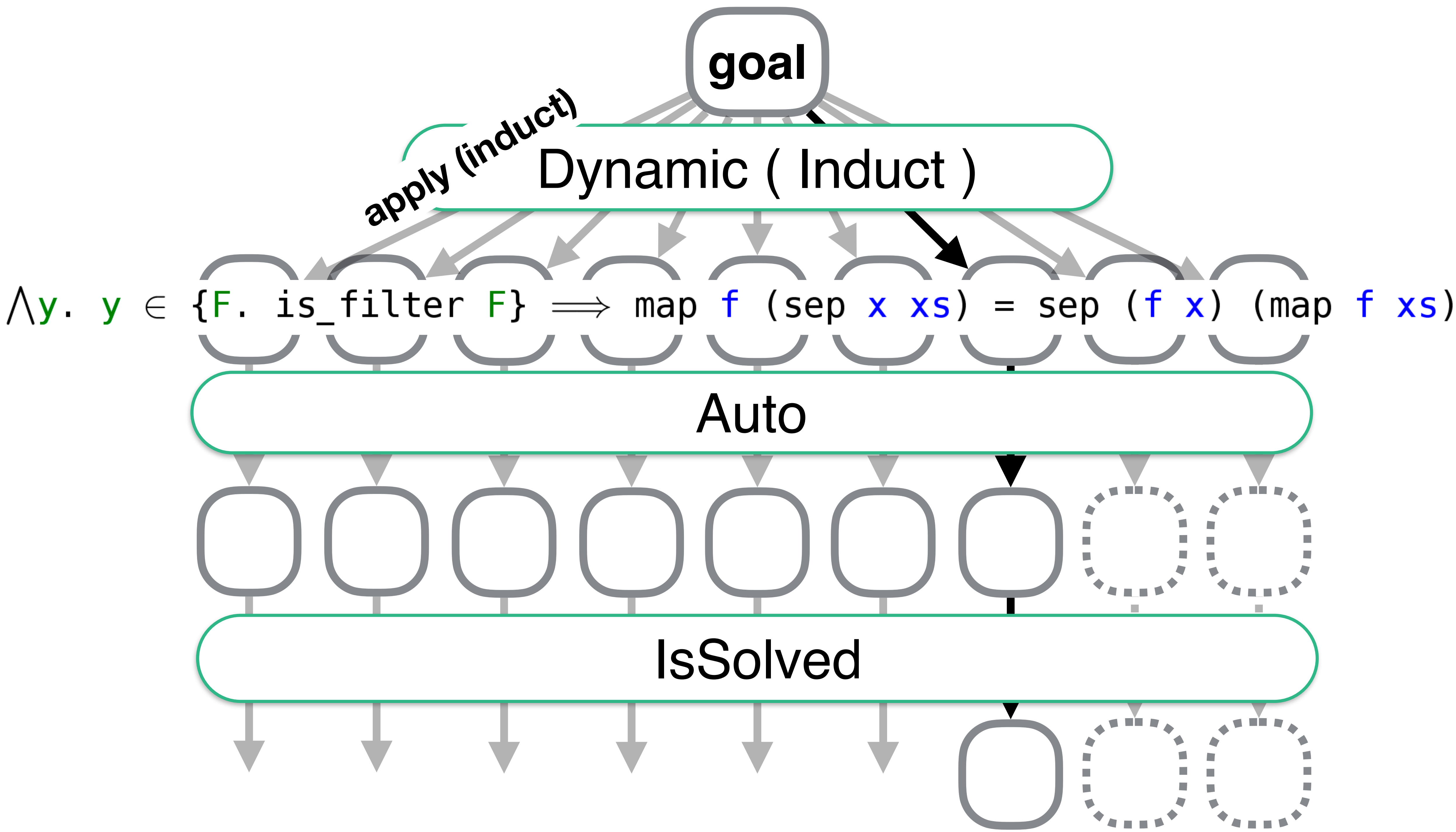
```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
```

```
find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



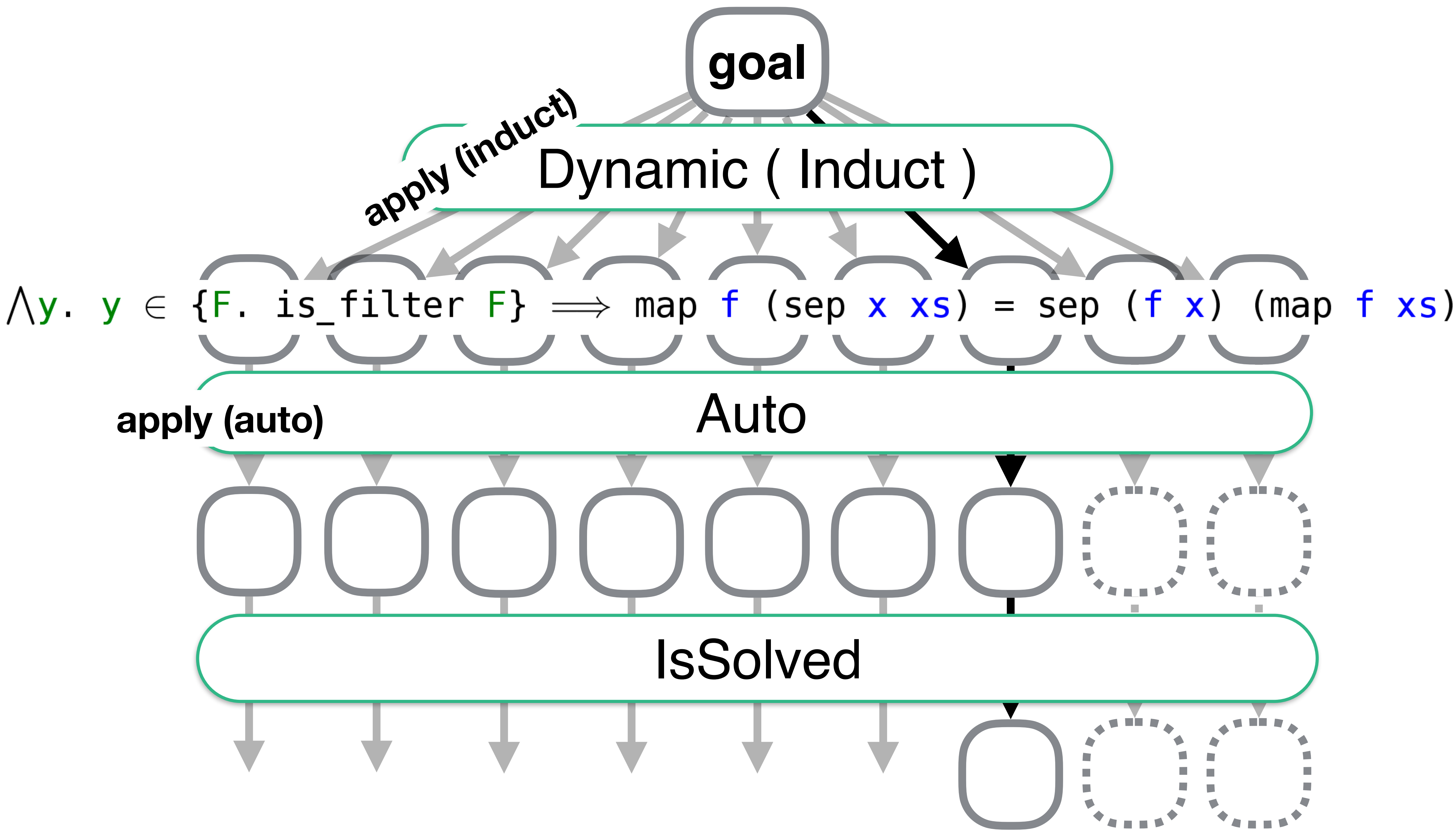
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"

find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



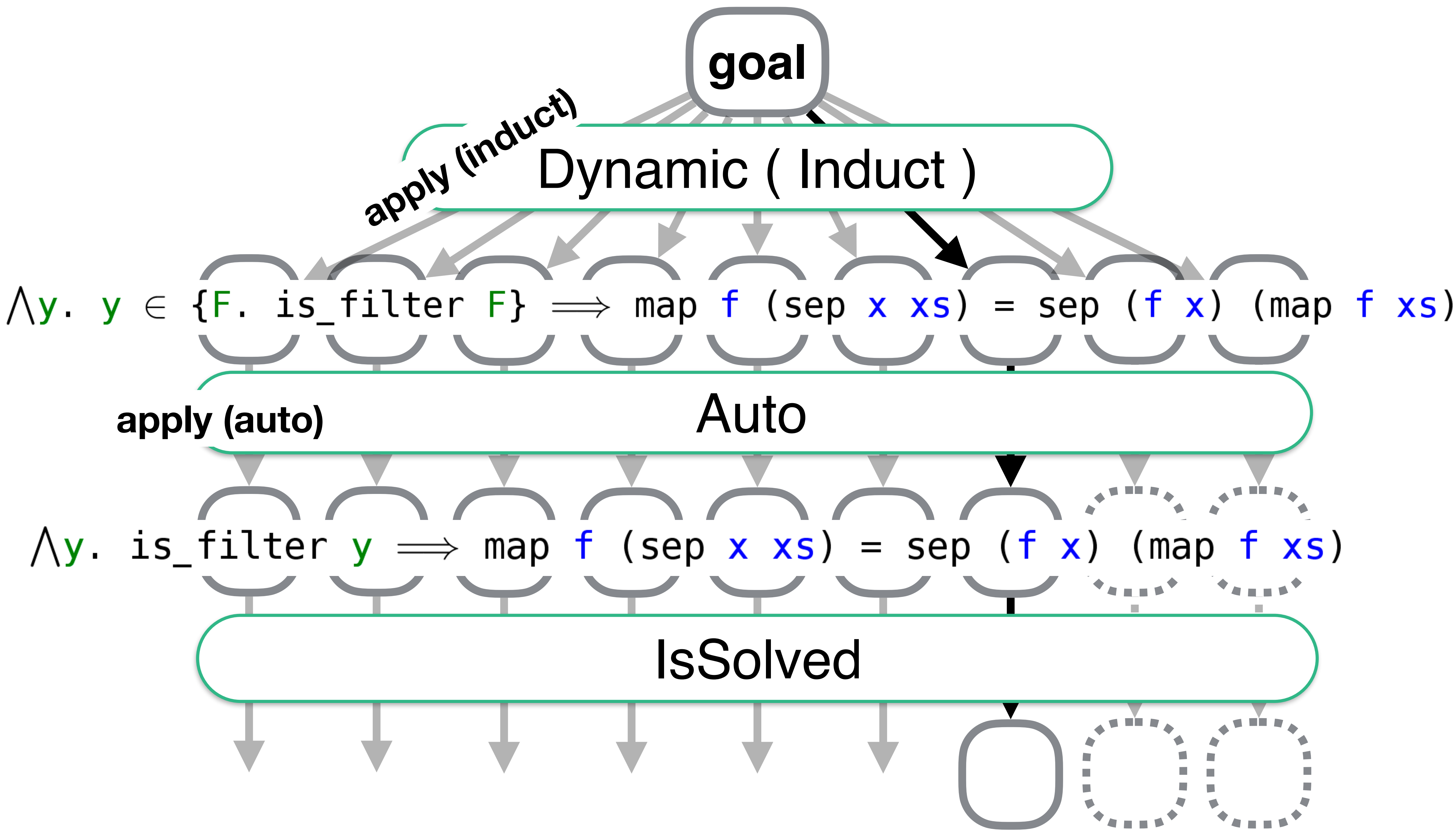
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"

find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



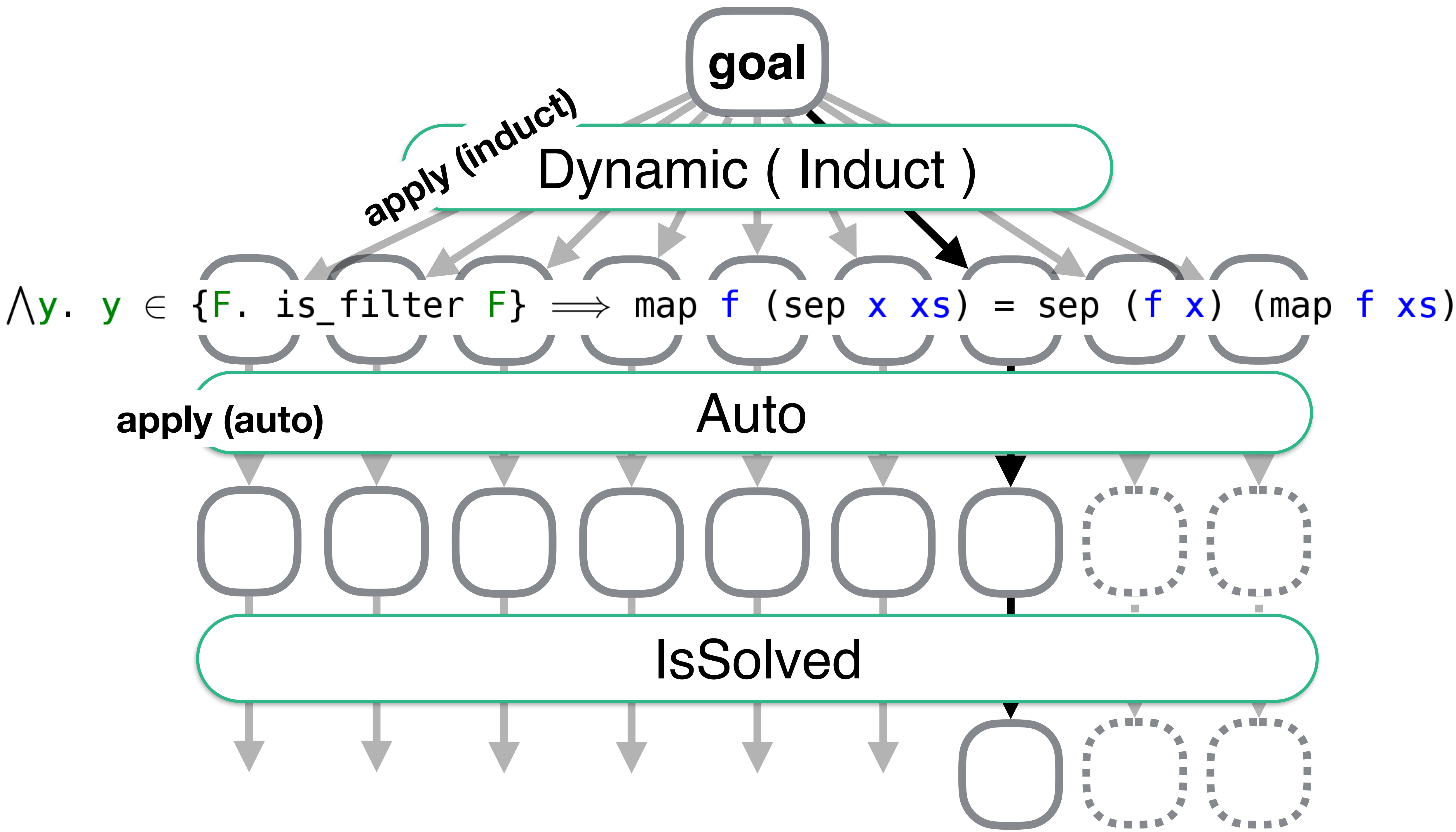
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"

find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



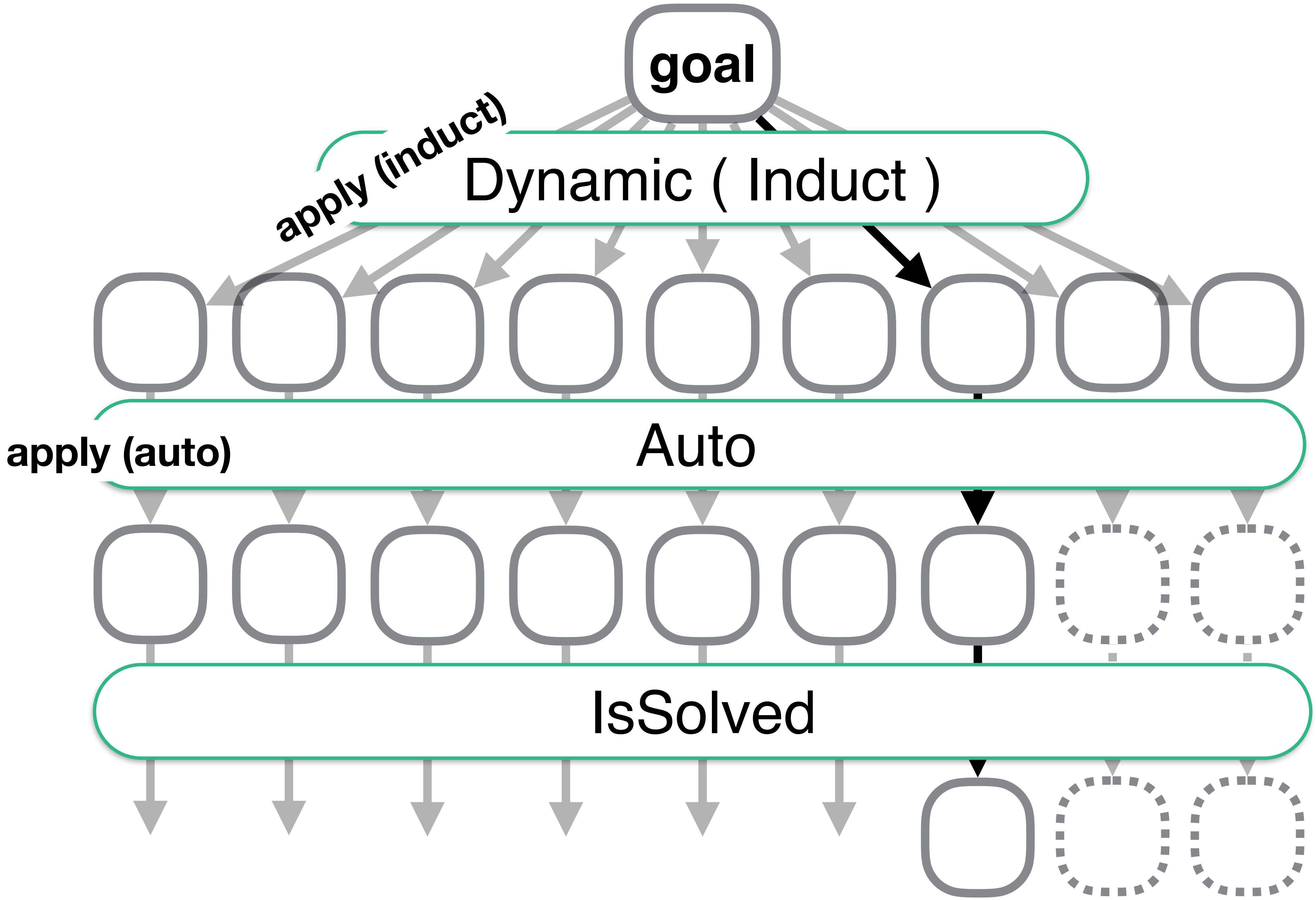
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"

find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



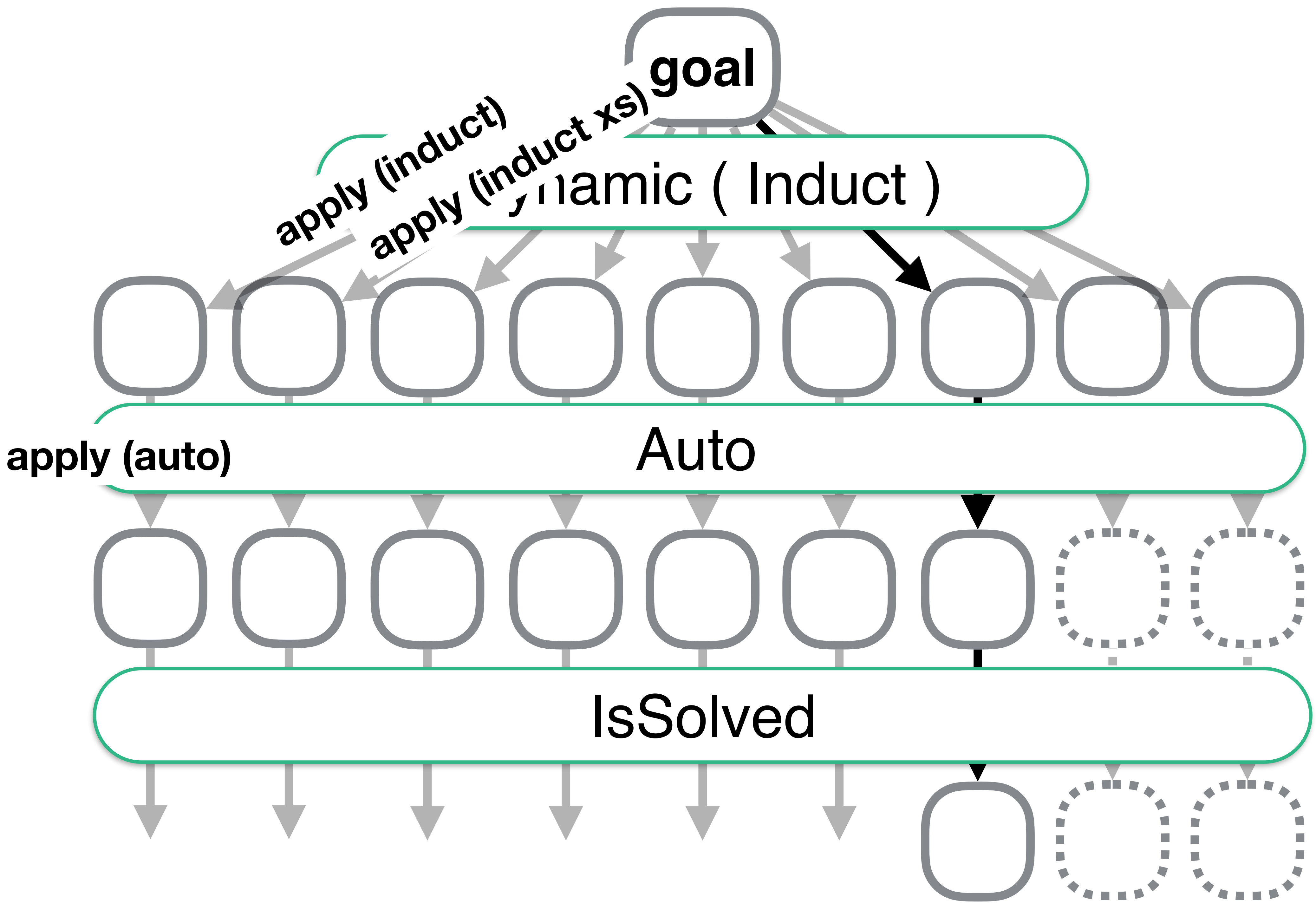
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"

find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



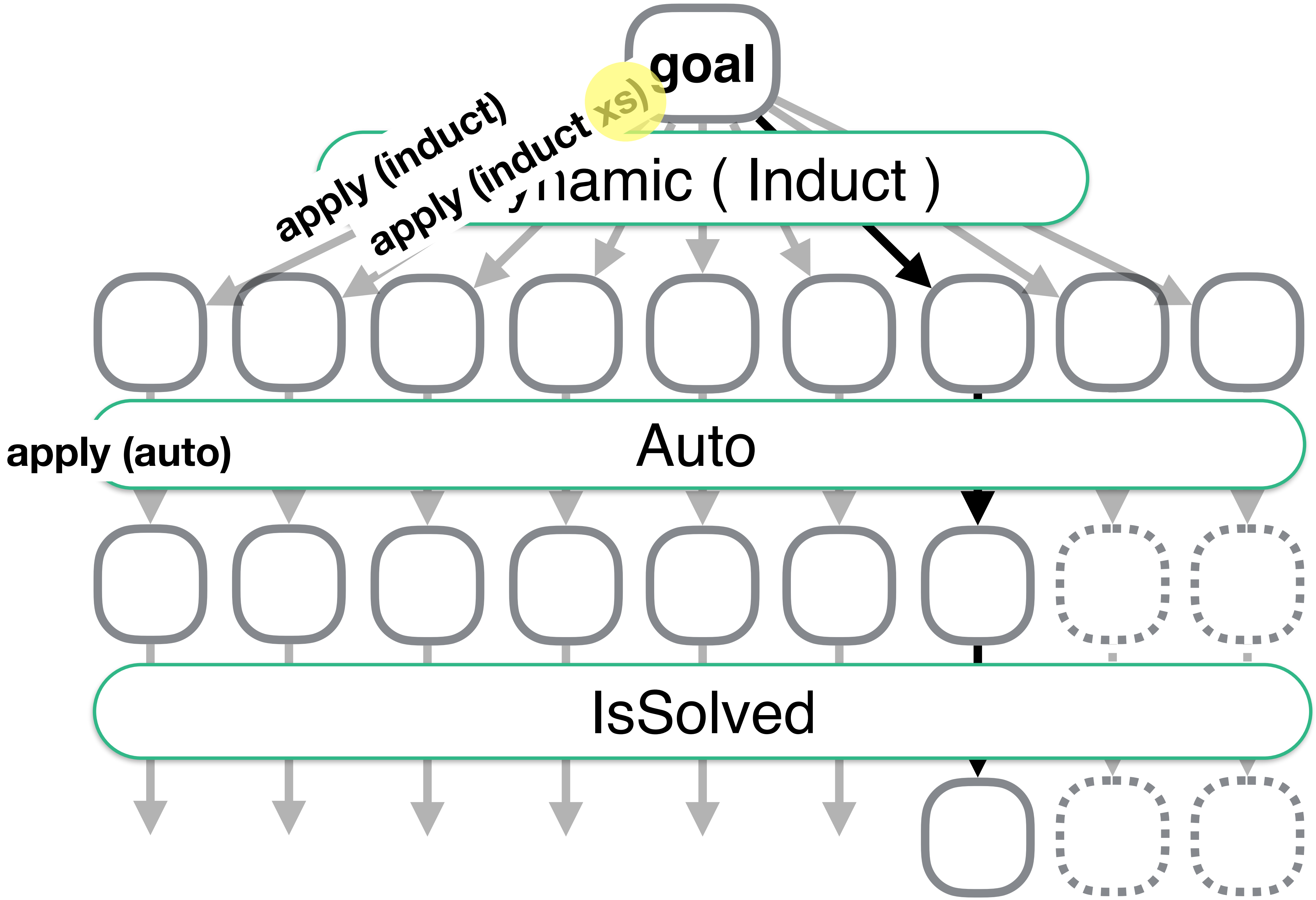
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"

find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



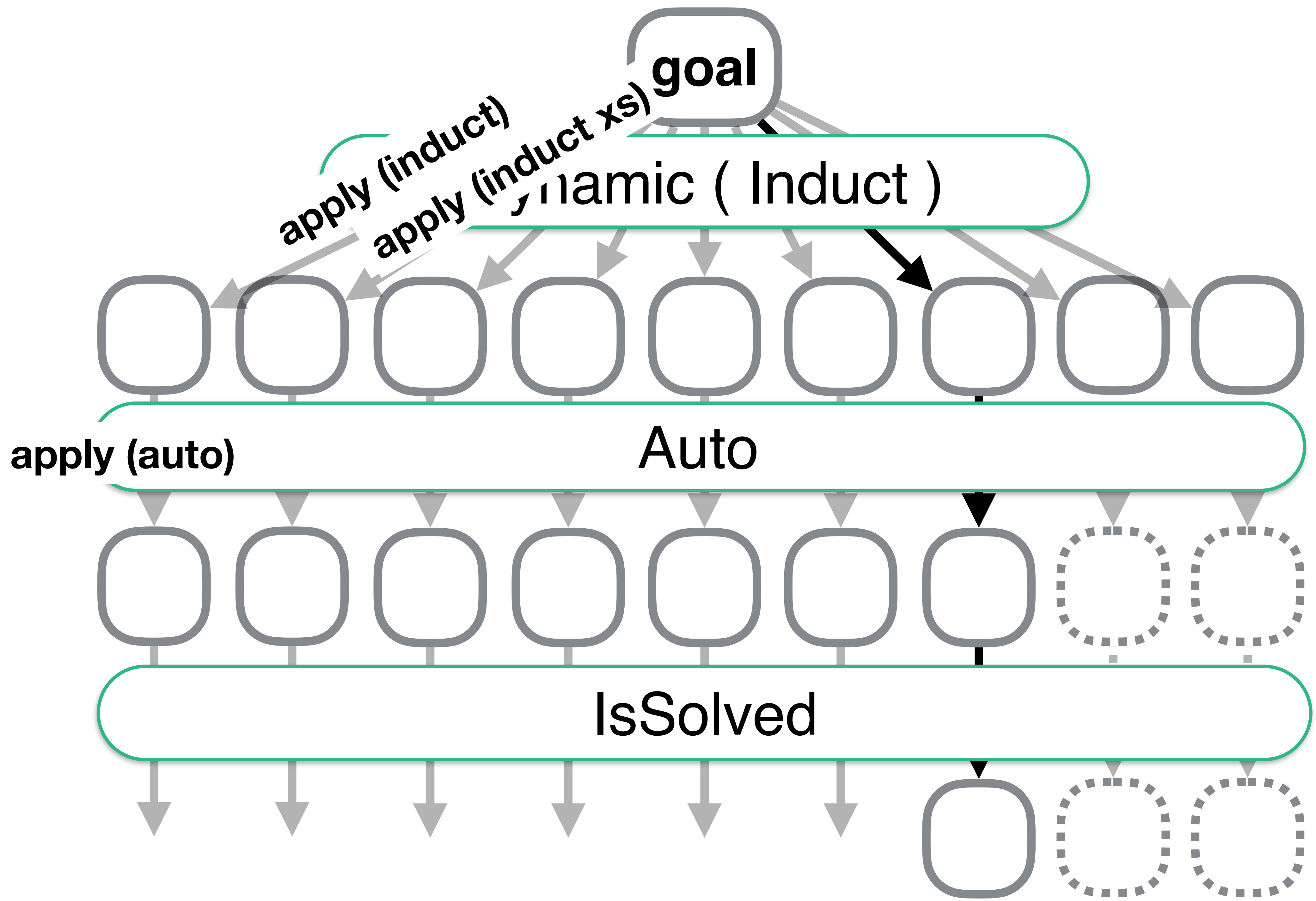
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

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

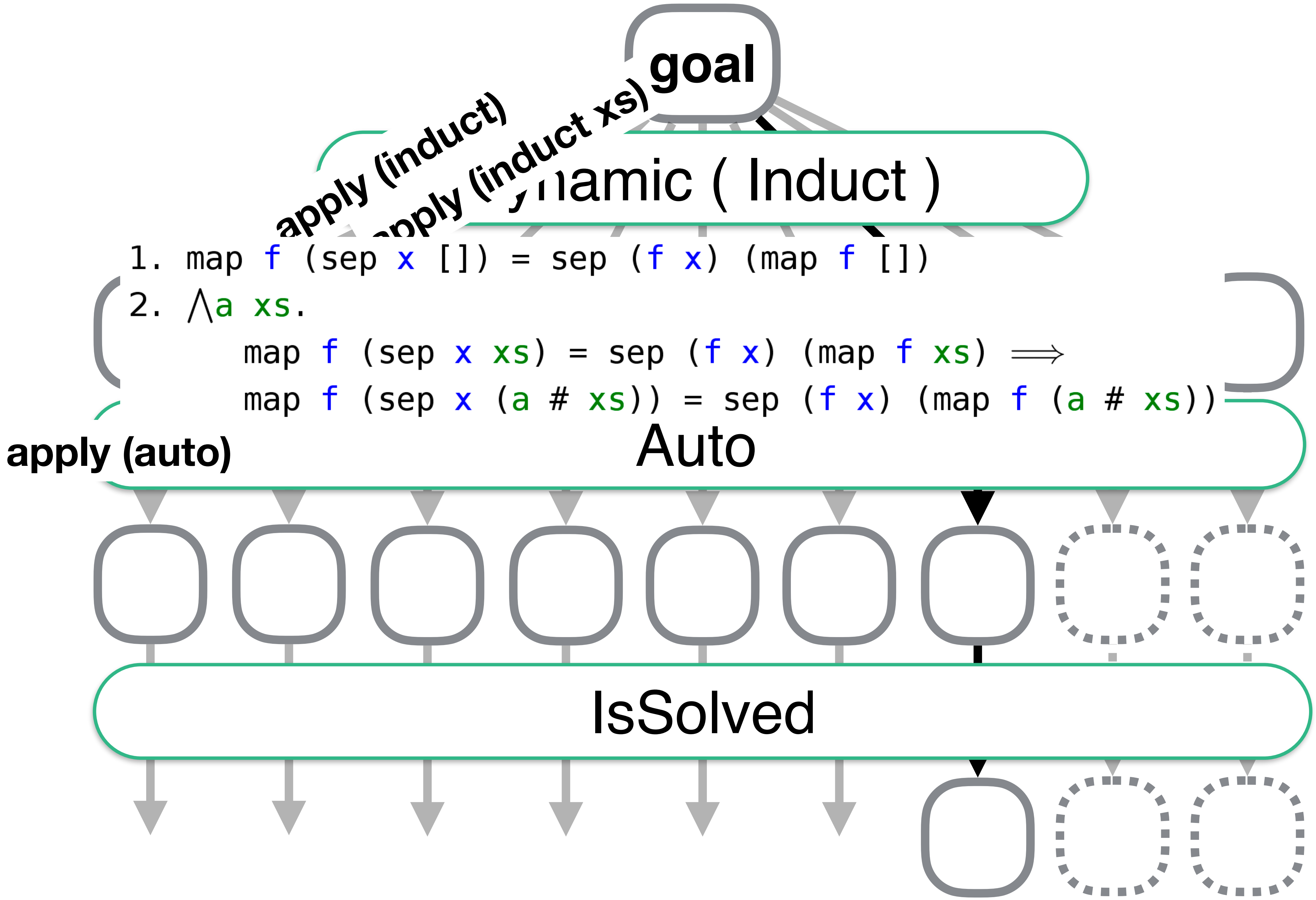


https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

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

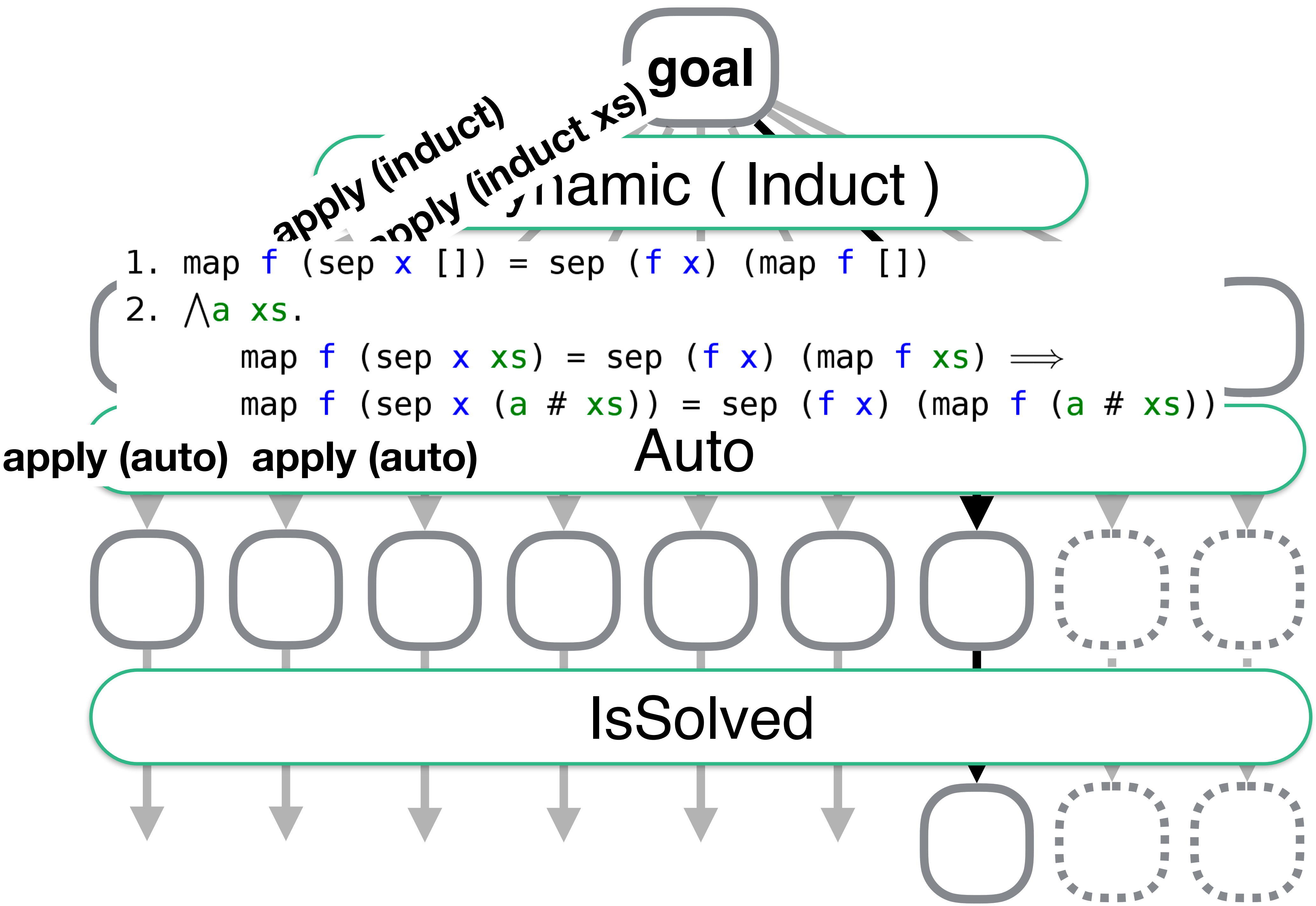


```
lemma "map f (sep x xs) = sep (f x) (map f xs)"

find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



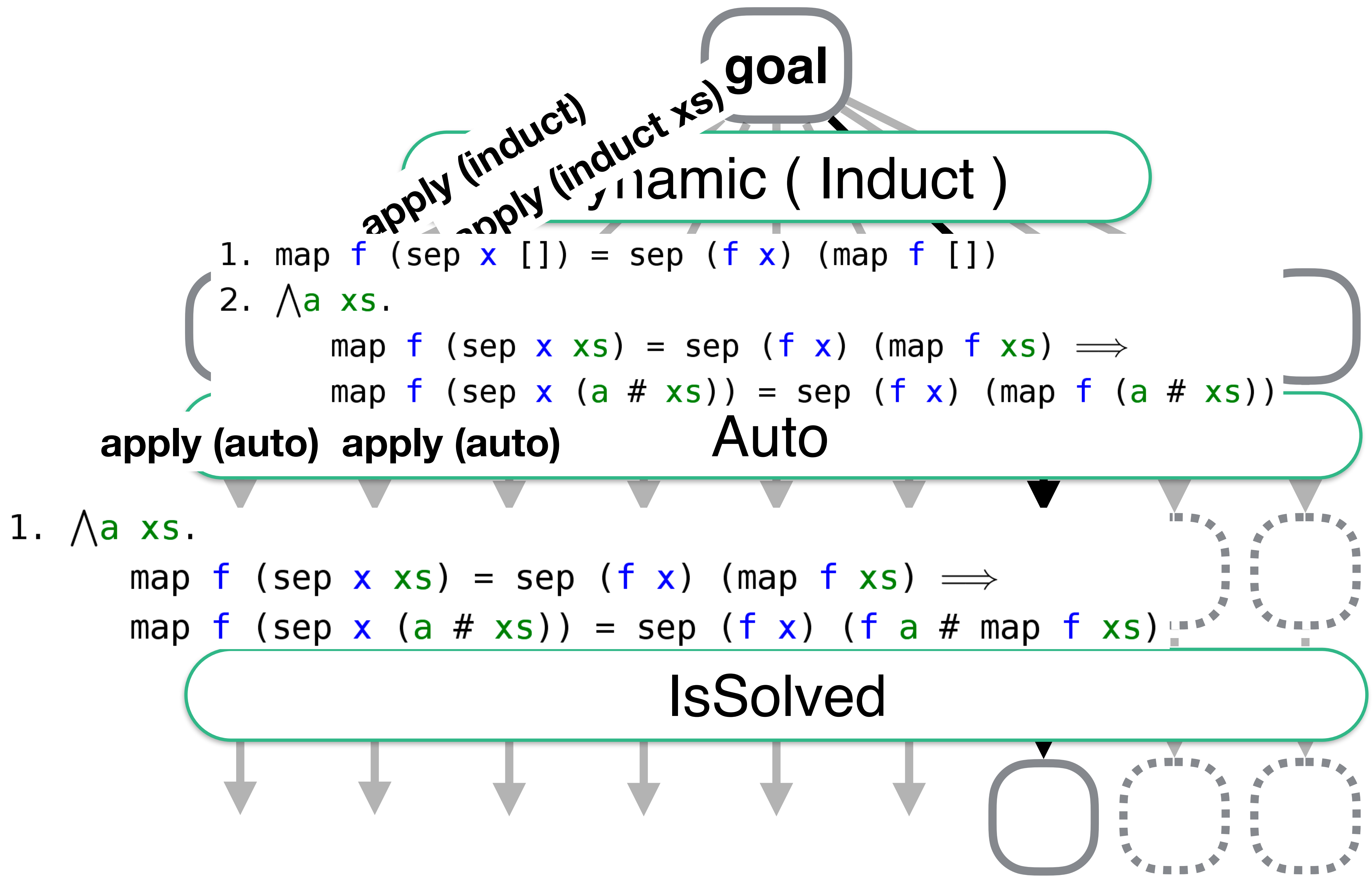
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

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



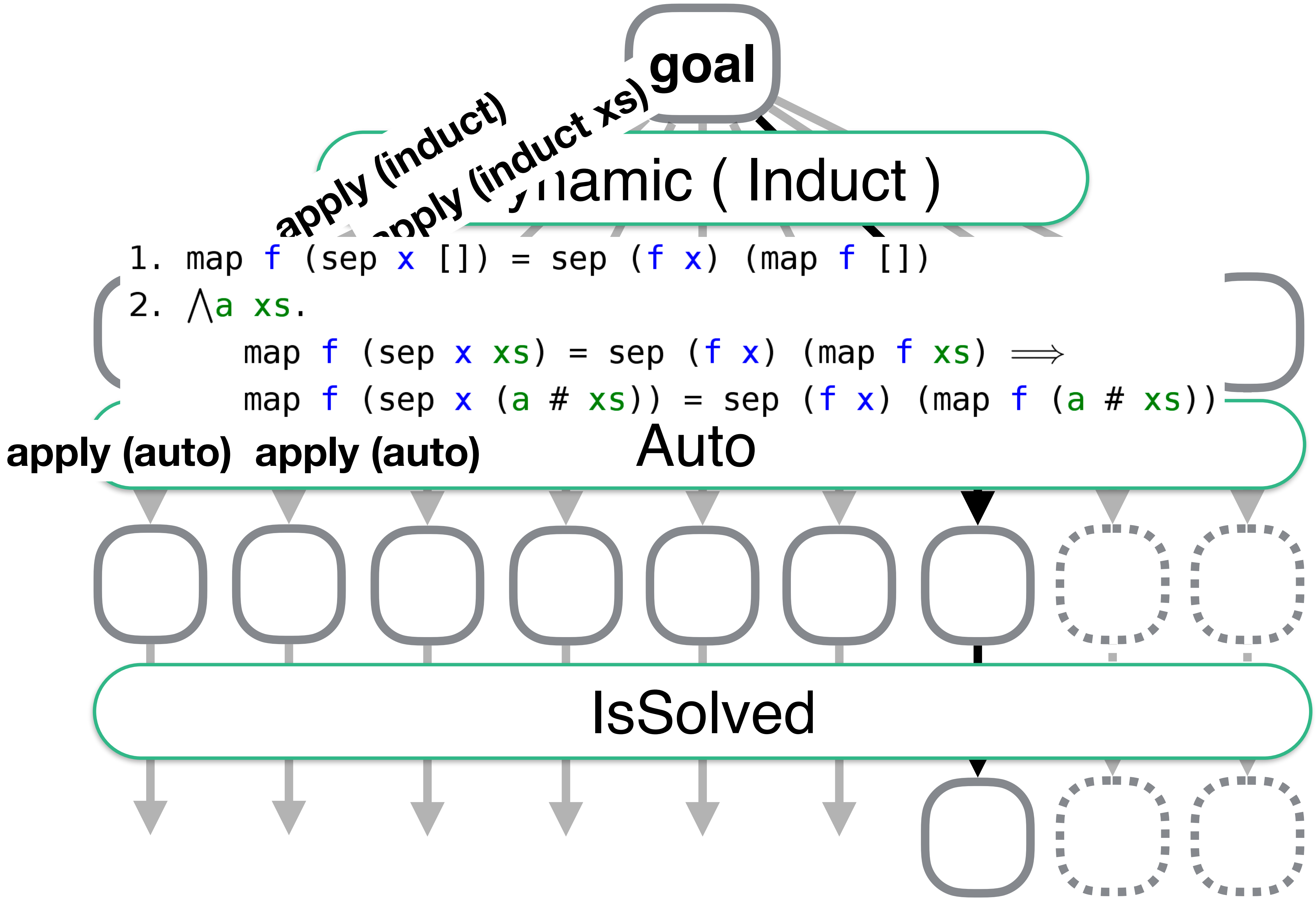
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"

find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



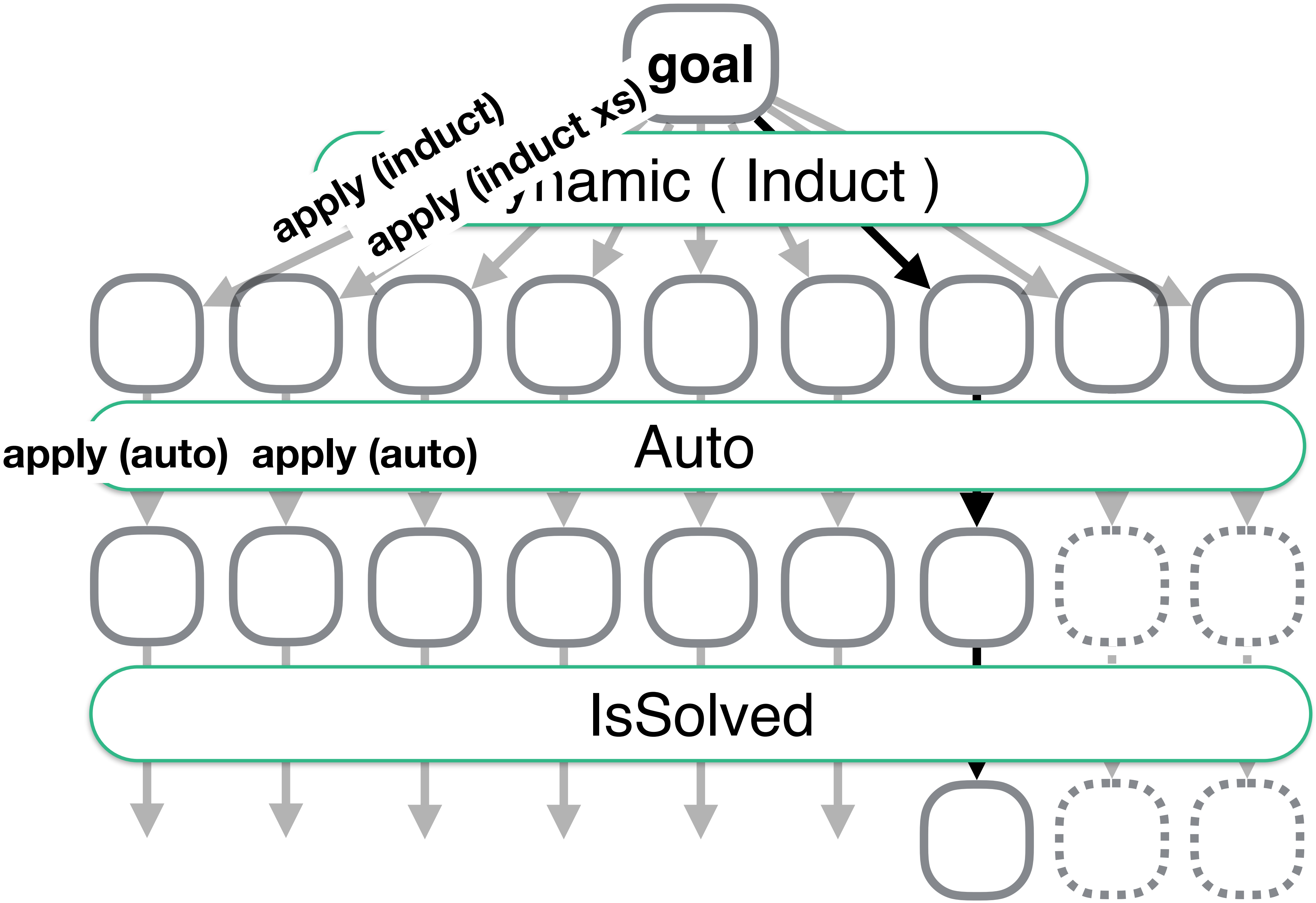
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"

find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



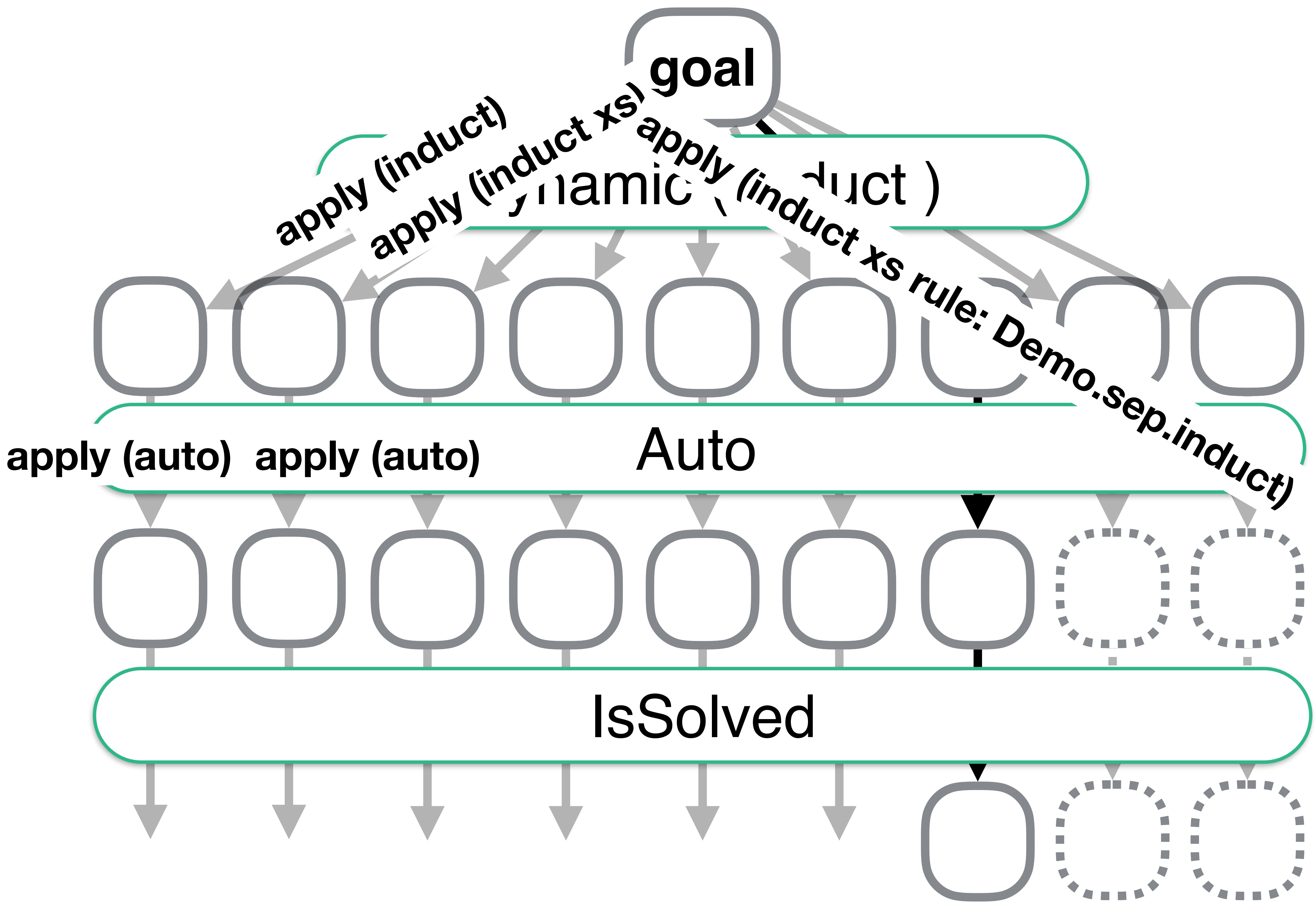
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

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



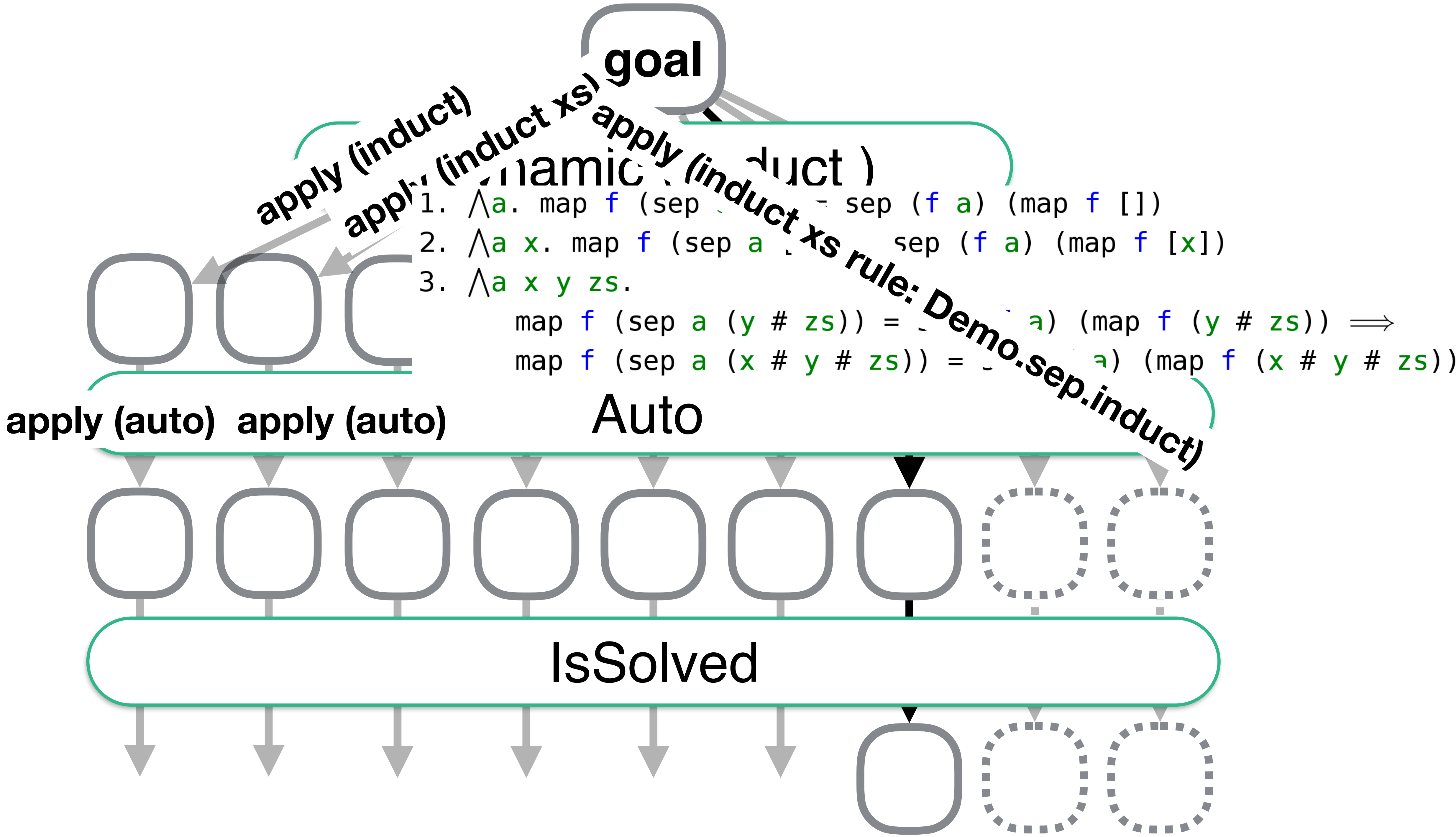
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

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



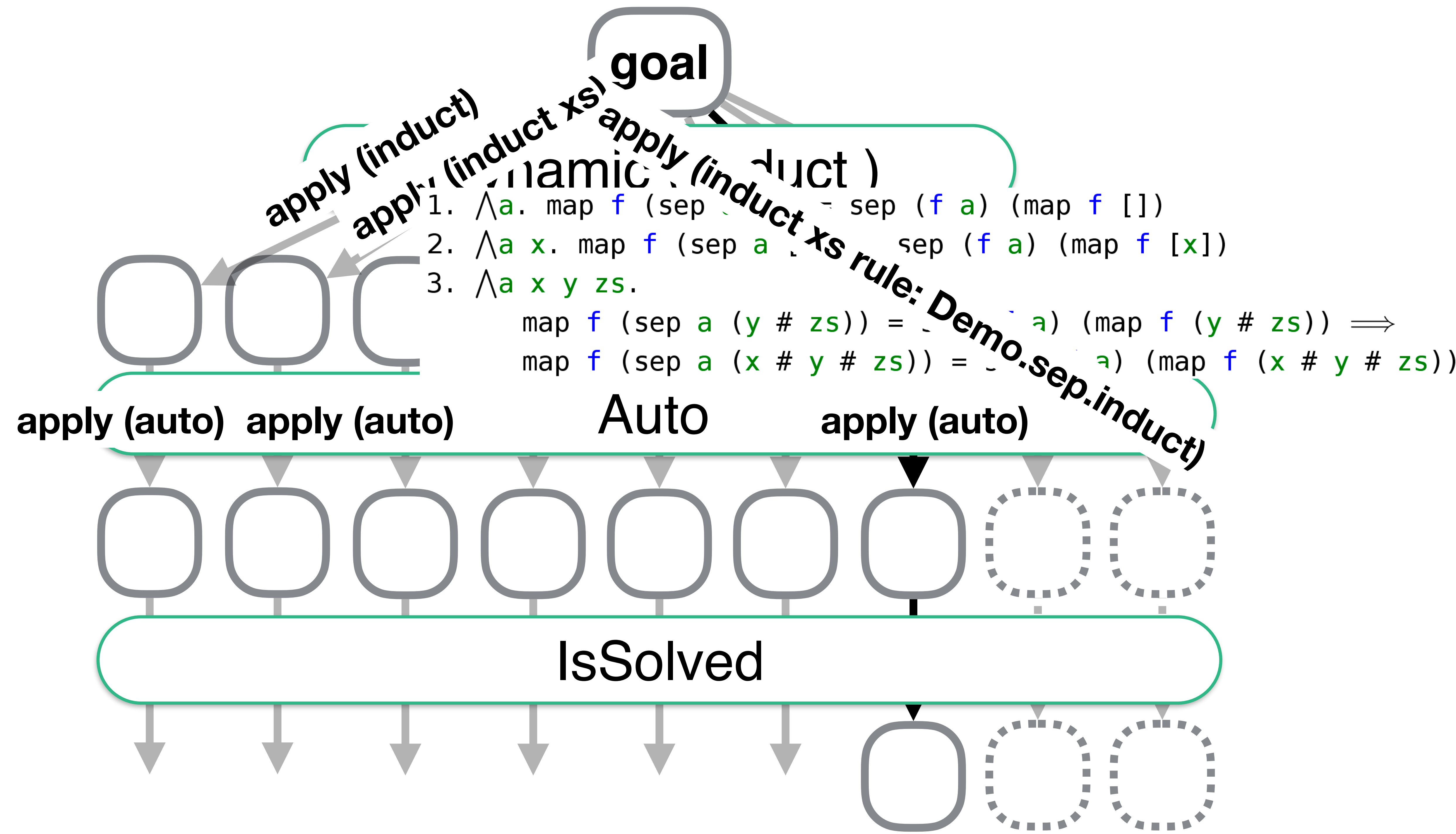
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

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



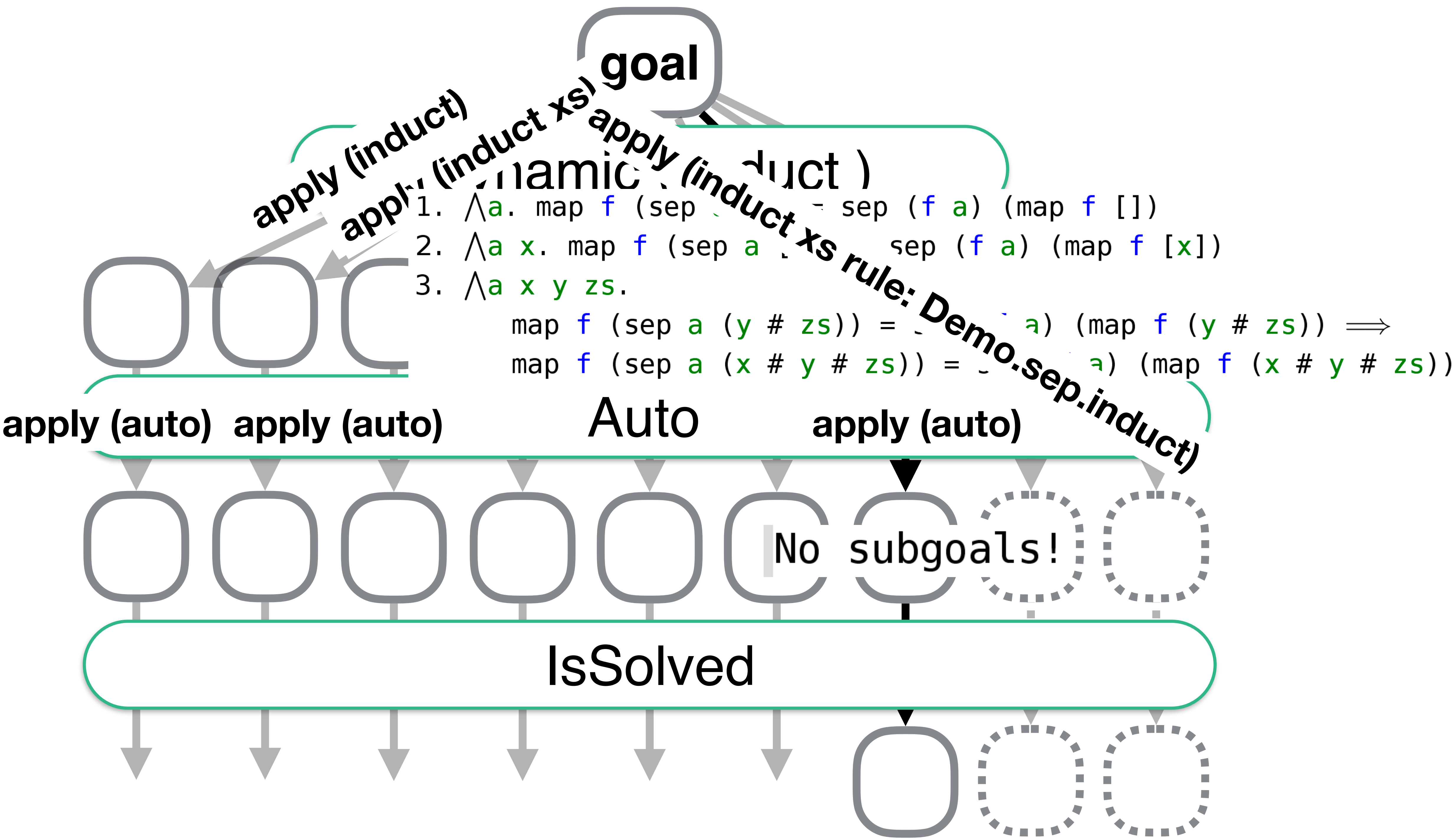
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

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



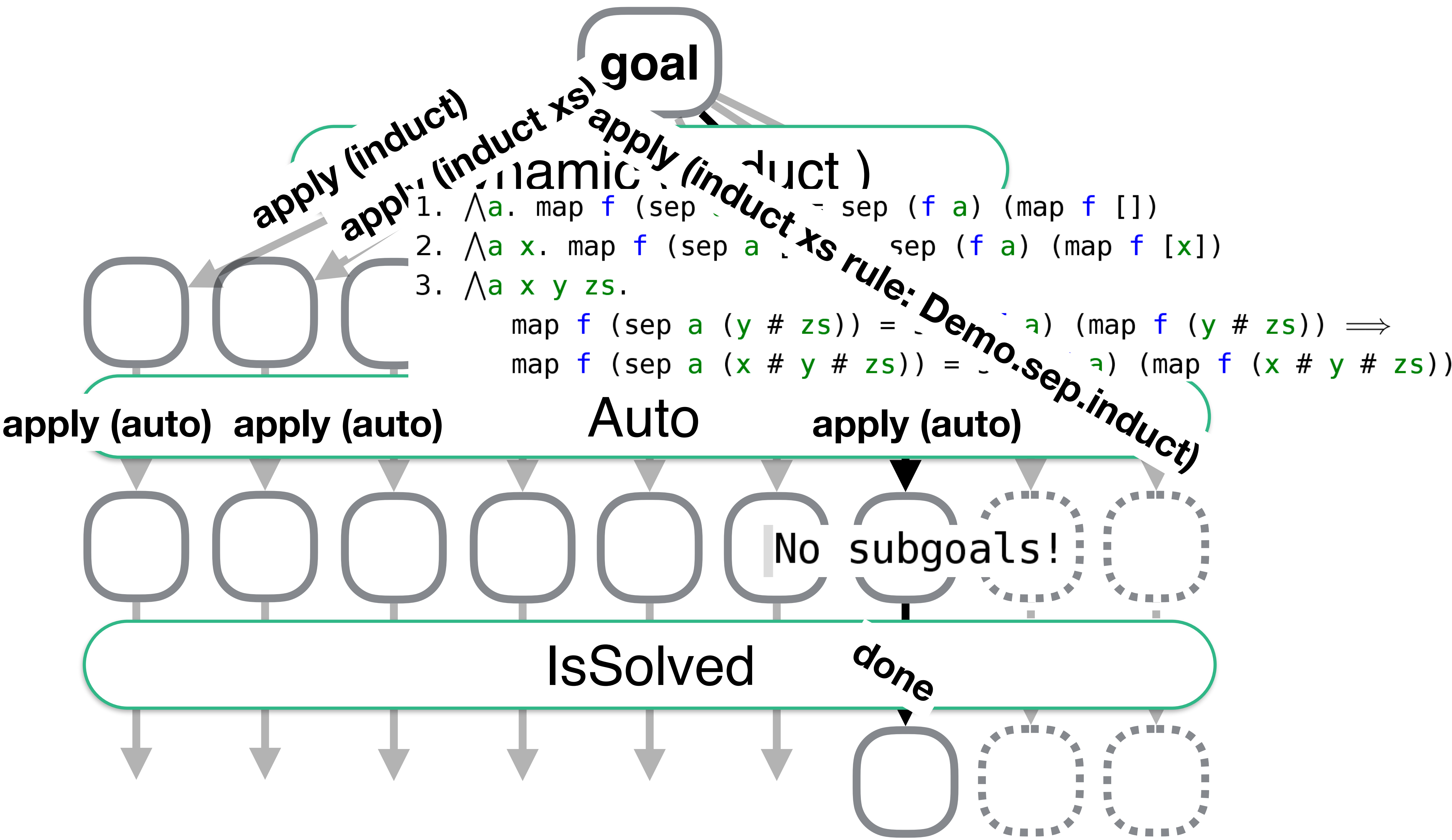
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"

find_proof DInd(*= Thens [Dynamic (Induct), Auto, IsSolved]*)
```



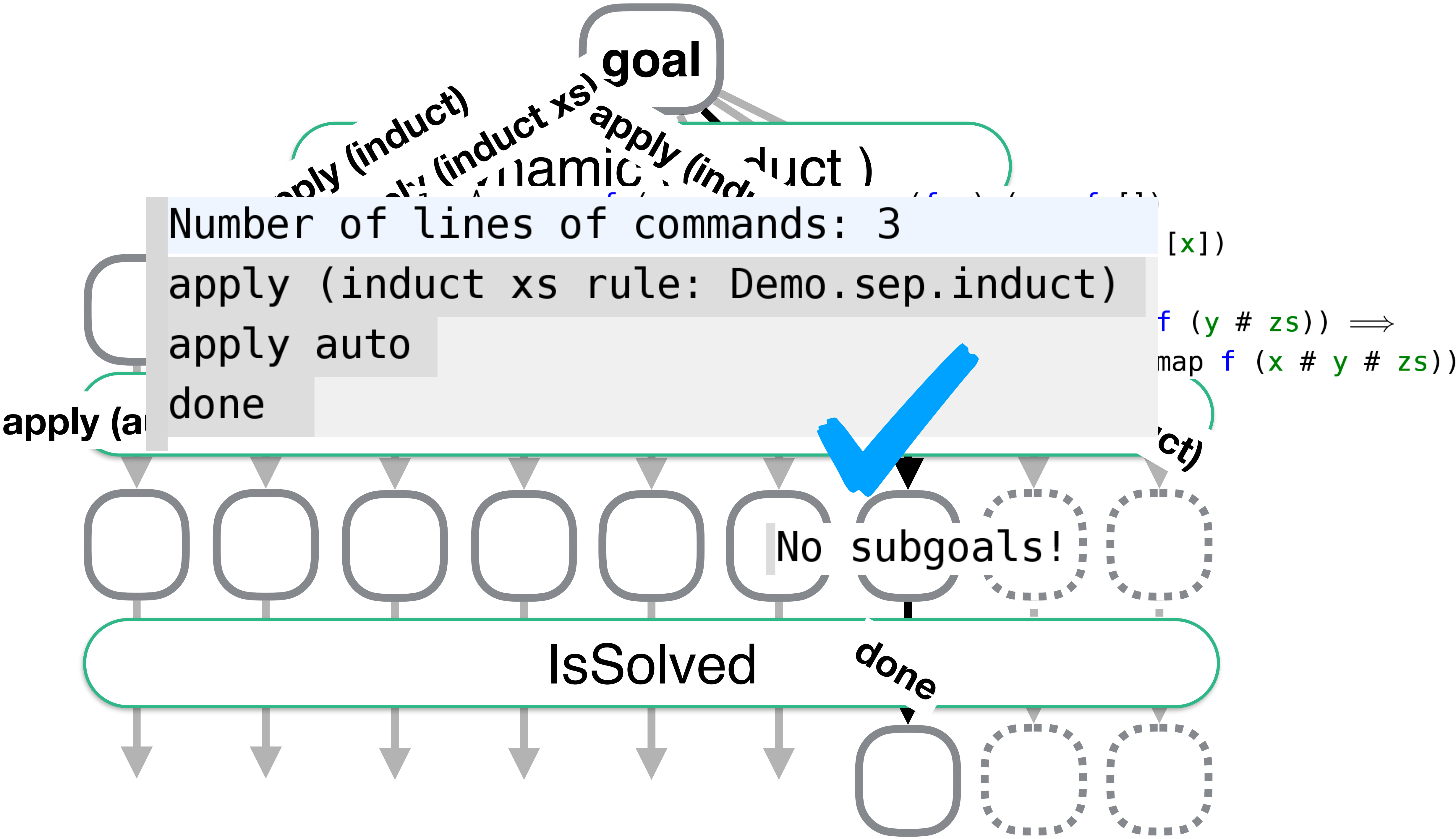
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

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



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

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



Try_Hard: the default strategy

```
strategy Try_Hard =
Ors [Thens [Subgoal, Basic],
      Thens [DInductTac, Auto_Solve],
      Thens [DCaseTac, Auto_Solve],
      Thens [Subgoal, Advanced],
      Thens [DCaseTac, Solve_Many],
      Thens [DInductTac, Solve_Many] ]
```

```
strategy Basic =
  Ors [
     Auto_Solve,
     Blast_Solve,
     FF_Solve,
     Thens [IntroClasses, Auto_Solve],
     Thens [Transfer, Auto_Solve],
     Thens [Normalization, IsSolved],
     Thens [DInduct, Auto_Solve],
     Thens [Hammer, IsSolved],
     Thens [DCases, Auto_Solve],
     Thens [DCoinduction, Auto_Solve],
     Thens [Auto, RepeatN(Hammer), IsSolved],
     Thens [DAuto, IsSolved]]
```

16 percentage point performance improvement compared to sledgehammer

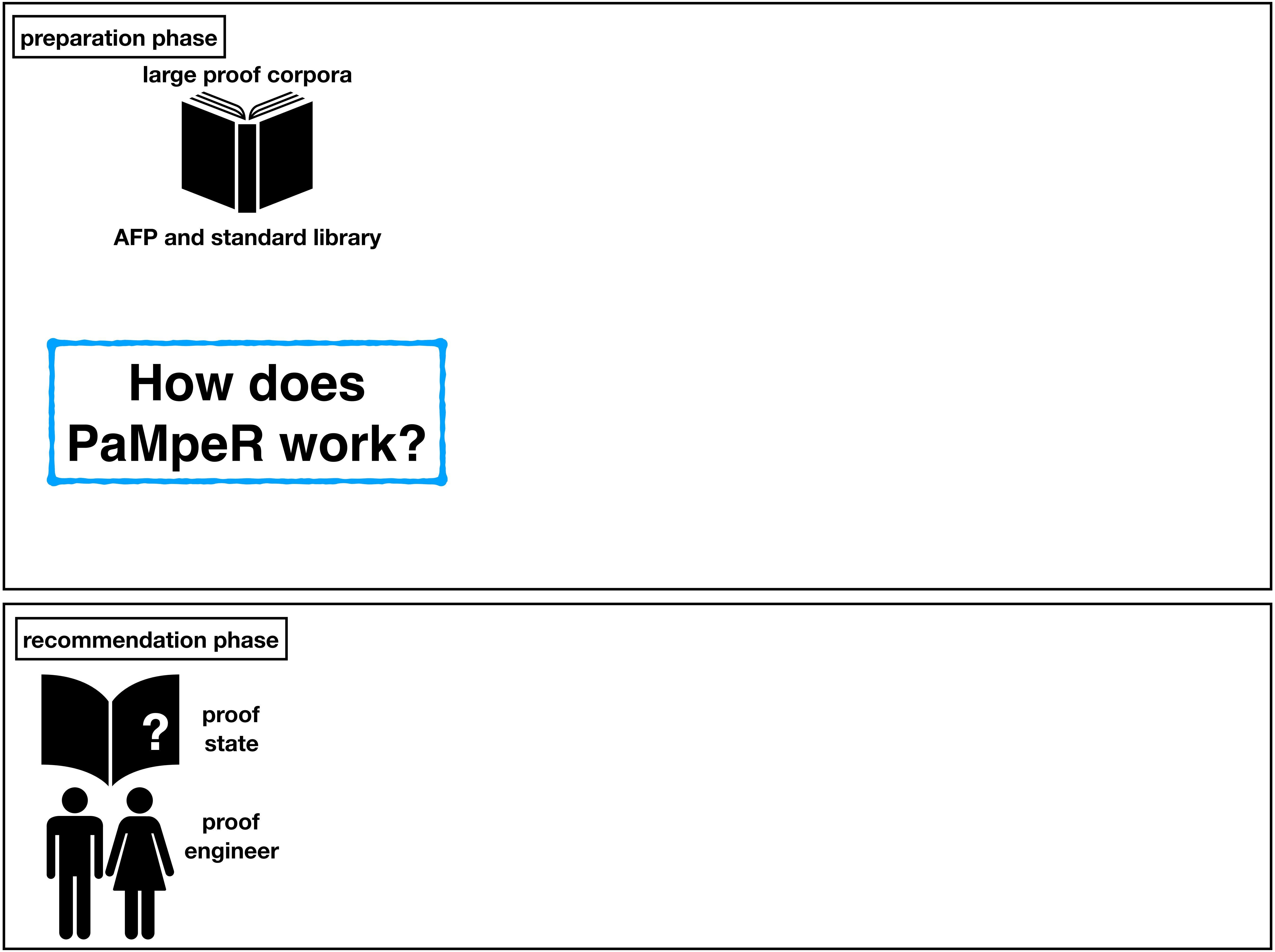
but the search space explodes

PaMpeR: Proof Method Recommendation

preparation phase

How does PaMpeR work?

recommendation phase


preparation phase
large proof corpora
AFP and standard library
How does PaMpeR work?
recommendation phase
proof state
proof engineer

preparation phase

large proof corpora

AFP and standard library

Statistics

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

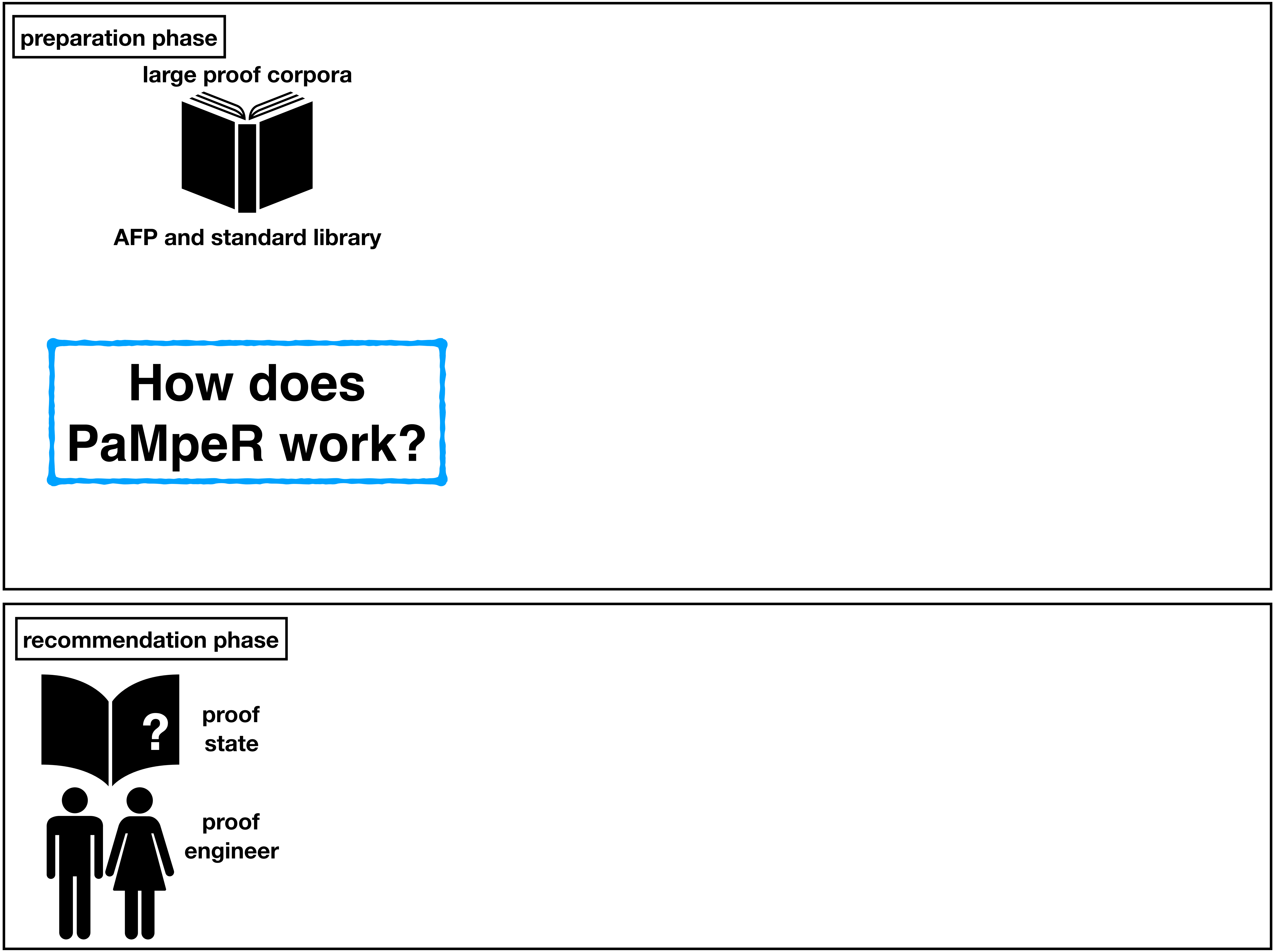
- Home
- About
- Submission
- Updating Entries
- Using Entries
- Search

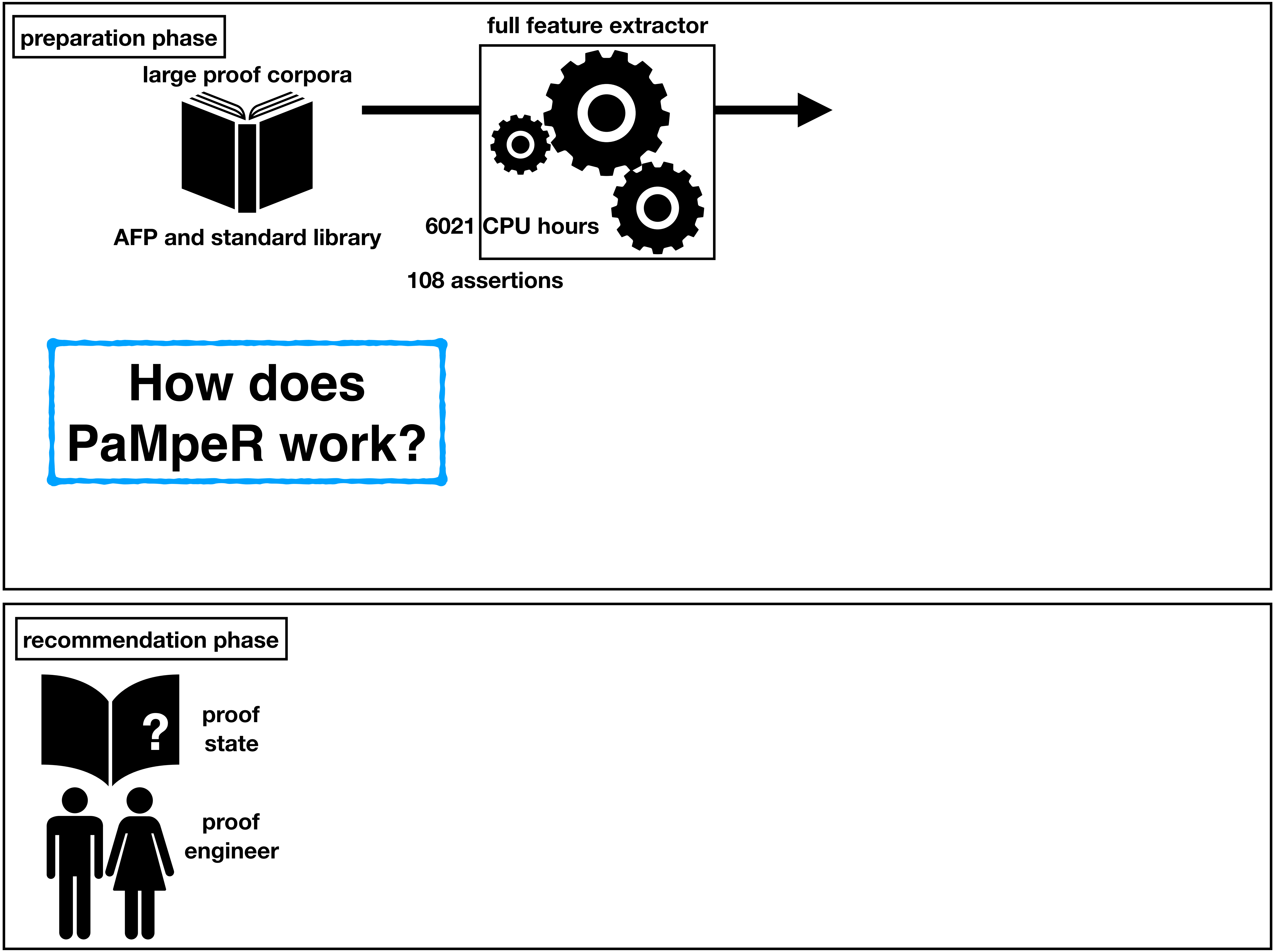
Statistics

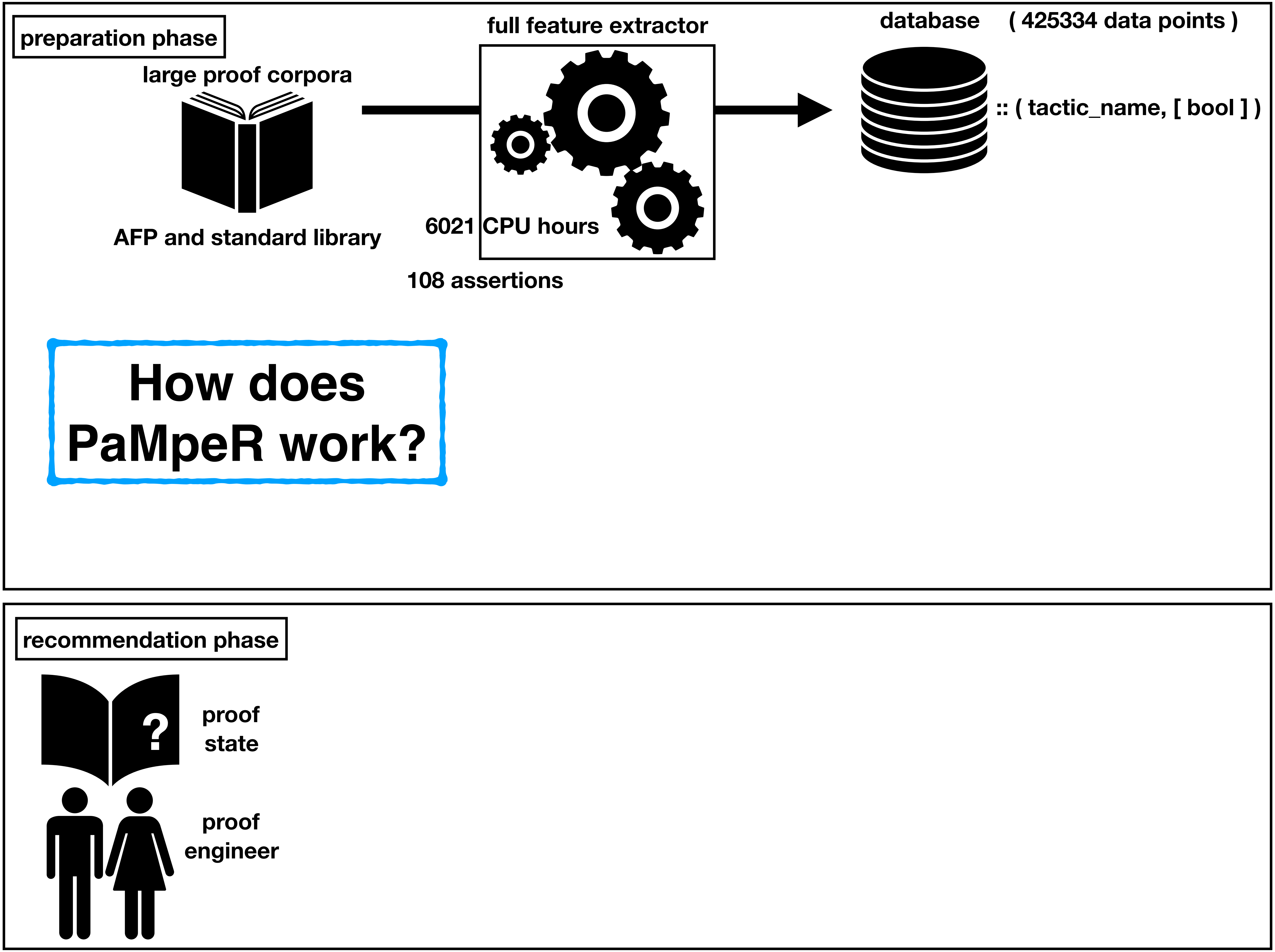
Number of Articles:	468
Number of Authors:	313
Number of lemmas:	~128,900
Lines of Code:	~2,170,300

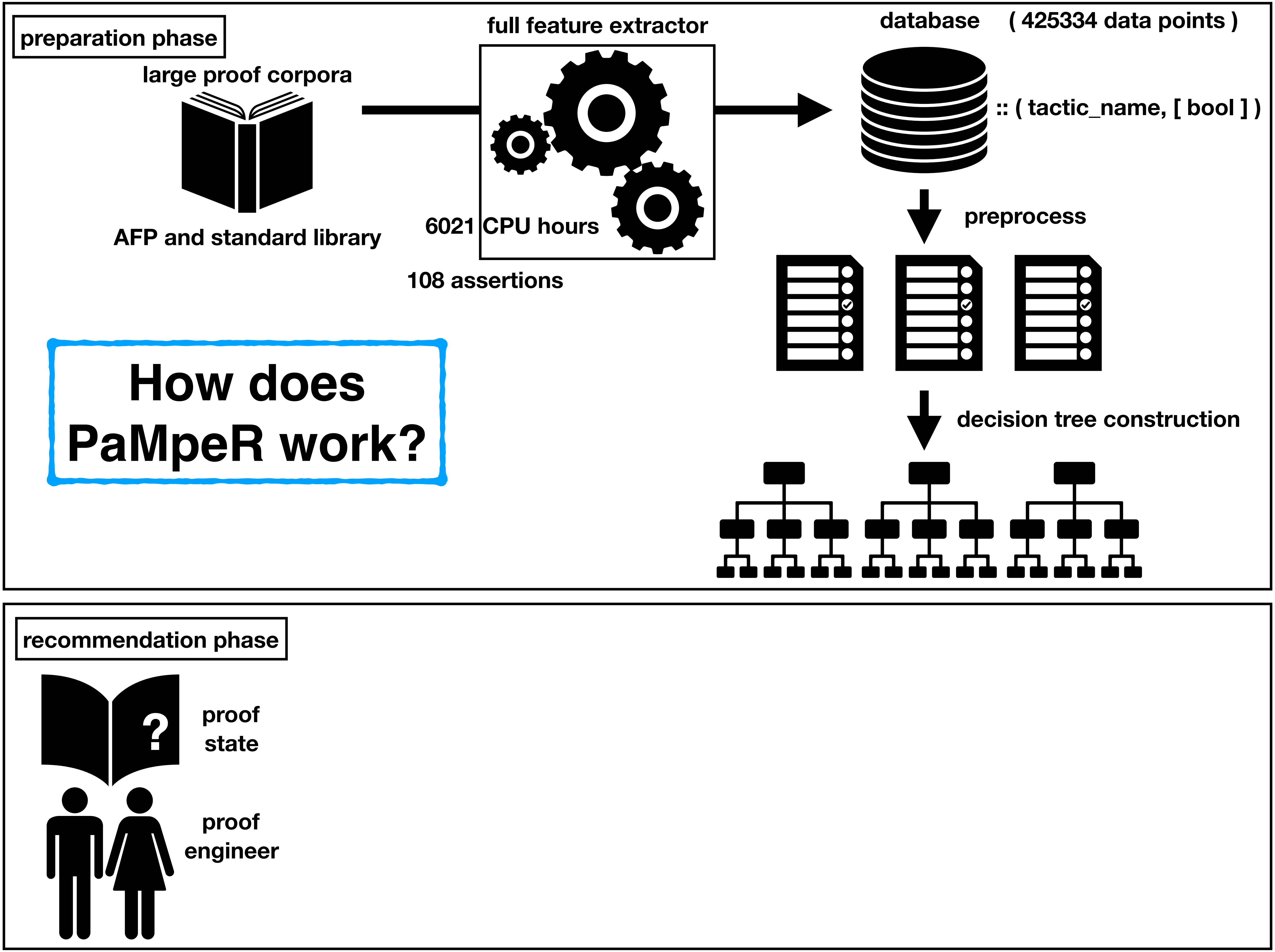
Most used AFP articles:

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

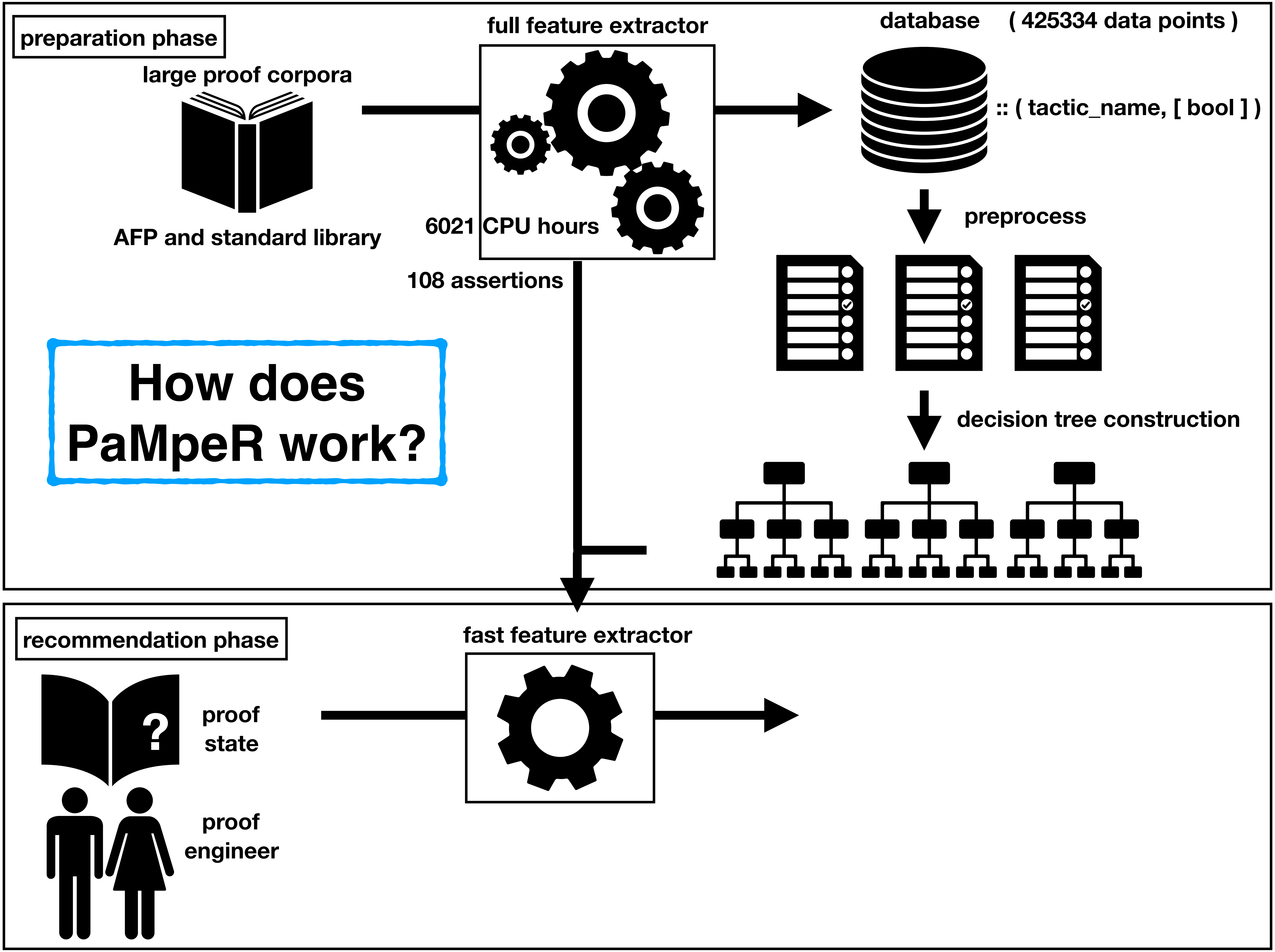

preparation phase
large proof corpora
AFP and standard library
How does PaMpeR work?
recommendation phase
proof state
proof engineer


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

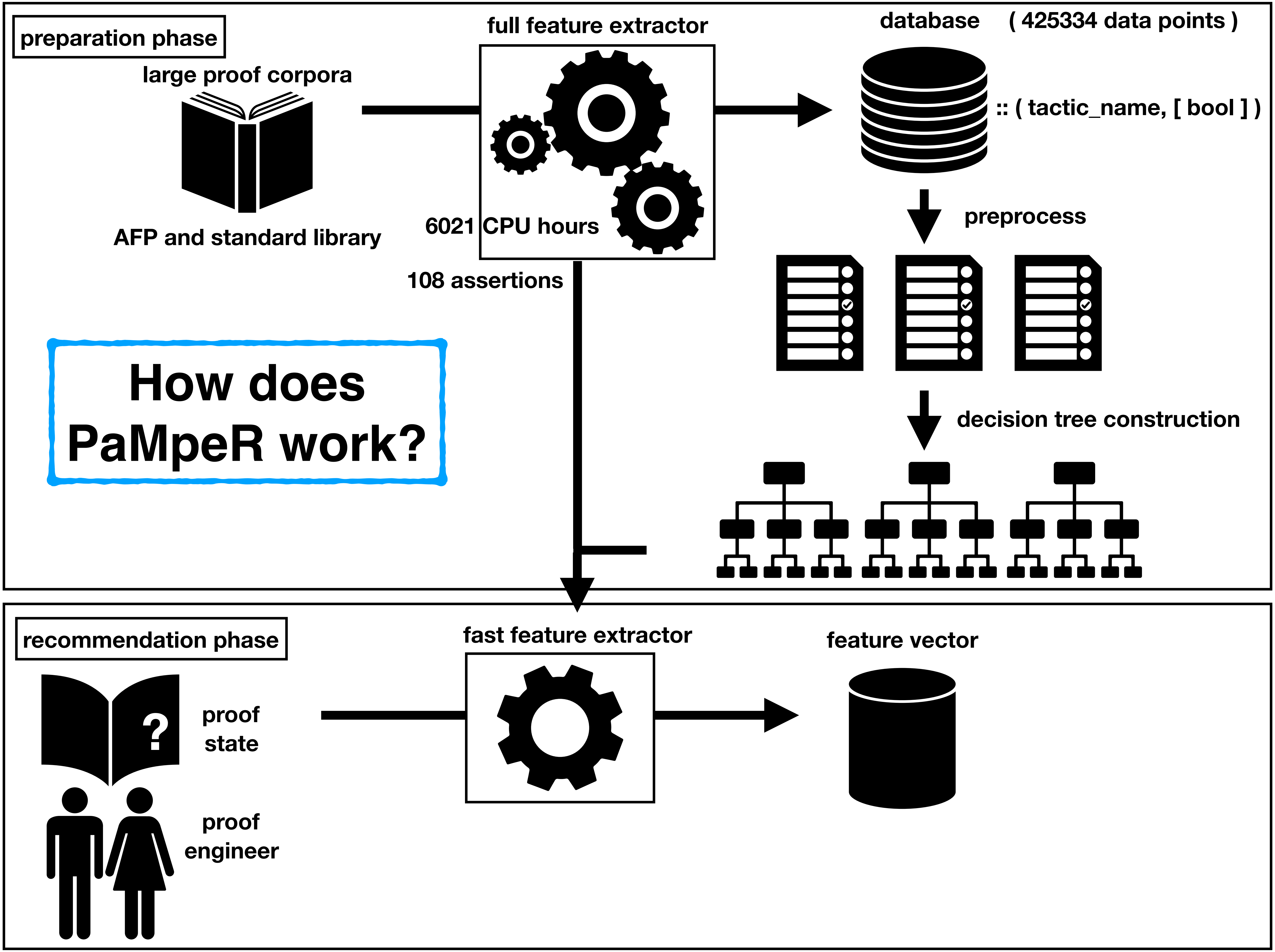

preparation phase
large proof corpora
AFP and standard library
full feature extractor
6021 CPU hours
108 assertions
database
(425334 data points)
:: (tactic_name, [bool])
How does PaMpeR work?
recommendation phase
?
proof state
proof engineer



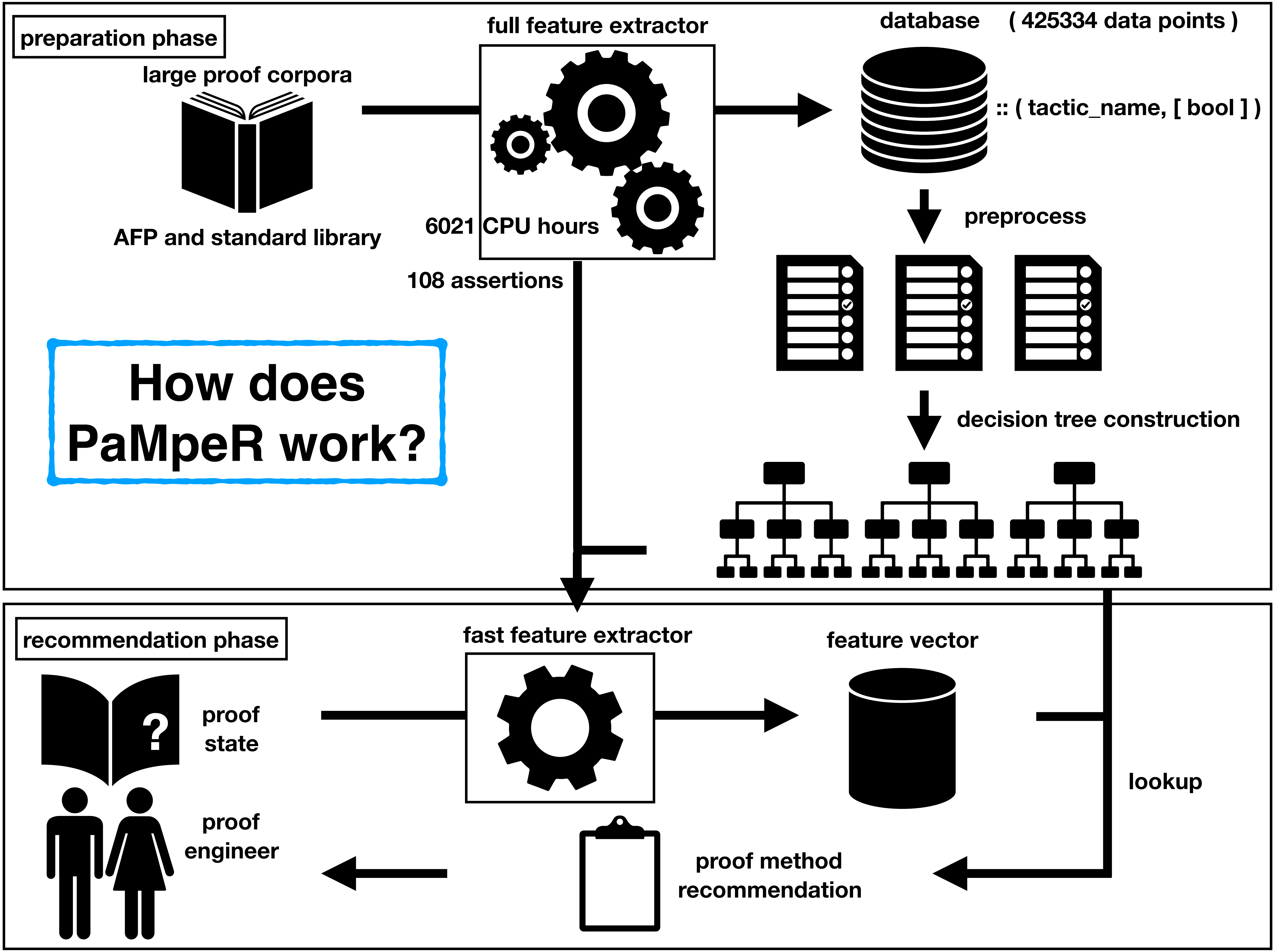
preparation phase
large proof corpora
AFP and standard library
full feature extractor
6021 CPU hours
108 assertions
database
(425334 data points)
:: (tactic_name, [bool])
preprocess
decision tree construction
How does PaMpeR work?
recommendation phase
proof state
proof engineer

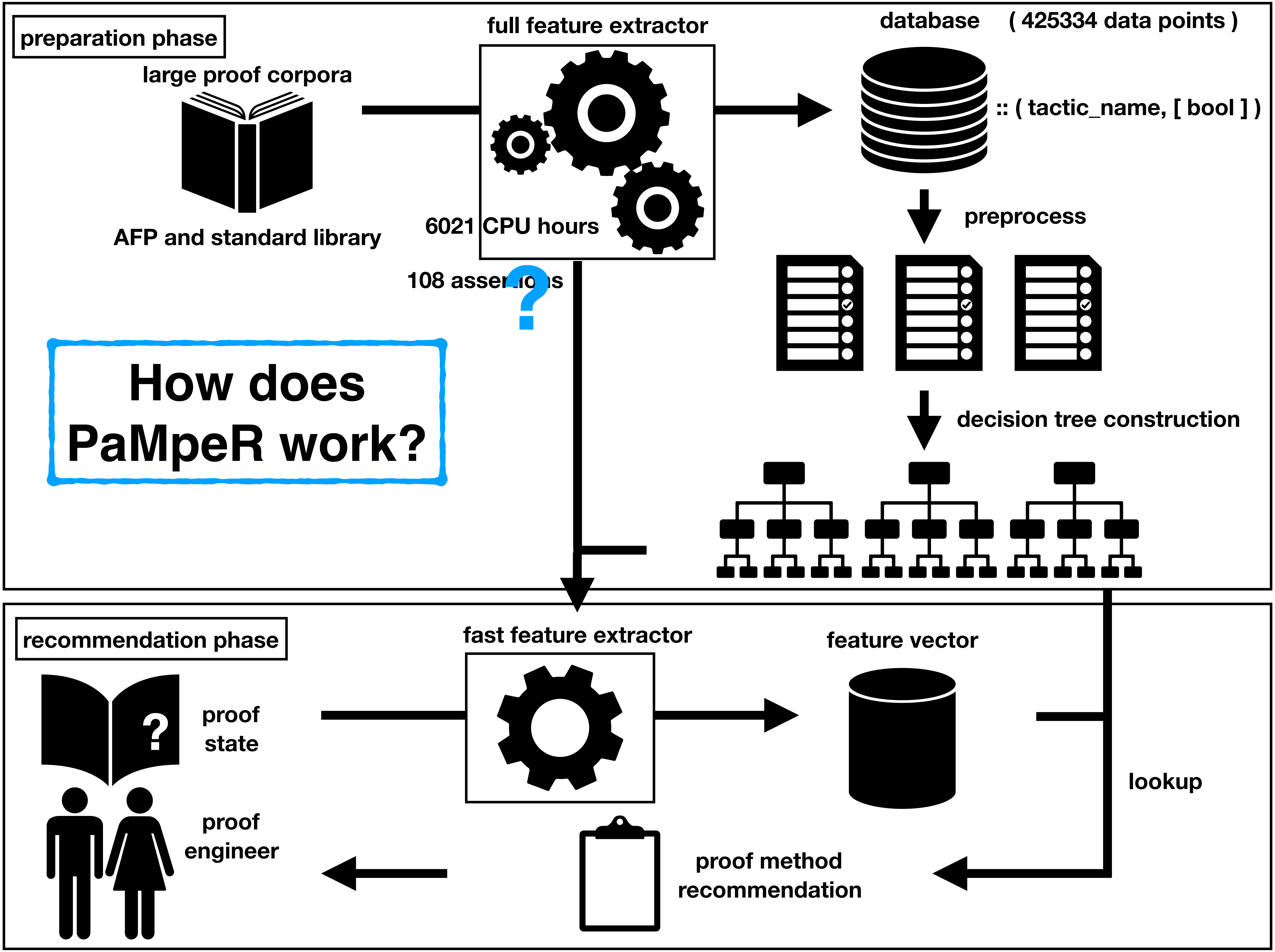


preparation phase
large proof corpora
AFP and standard library
full feature extractor
6021 CPU hours
108 assertions
database
(425334 data points)
:: (tactic_name, [bool])
preprocess
decision tree construction
How does PaMpeR work?
recommendation phase
proof state
proof engineer
fast feature extractor



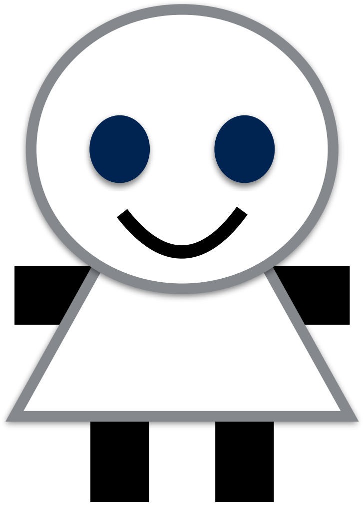
preparation phase
large proof corpora
AFP and standard library
full feature extractor
6021 CPU hours
108 assertions
database
(425334 data points)
:: (tactic_name, [bool])
preprocess
decision tree construction
How does PaMpeR work?
recommendation phase
proof state
proof engineer
fast feature extractor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


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


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

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

ITP2018 review

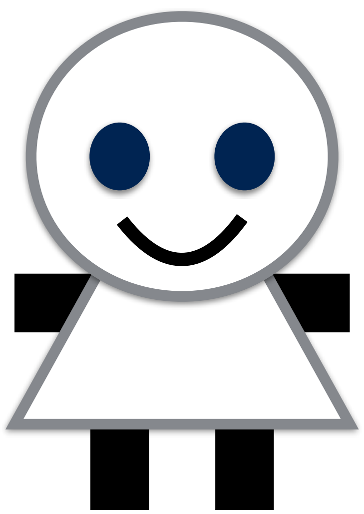


anonymous reviewer

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

ITP2018 review

`Proof Method Recommendation, PaMpeR!`

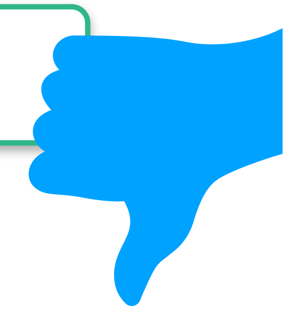


anonymous
reviewer

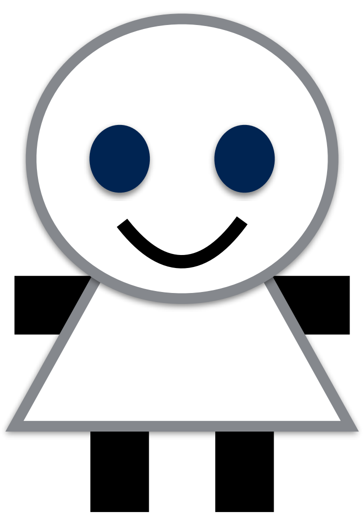
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

ITP2018 review

Proof Method Recommendation, PaMpeR!



I have doubts about various approaches proposed in the paper.

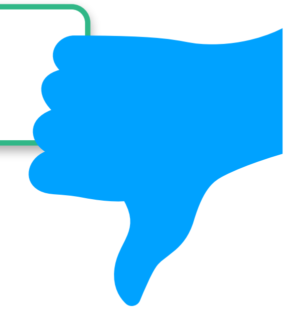


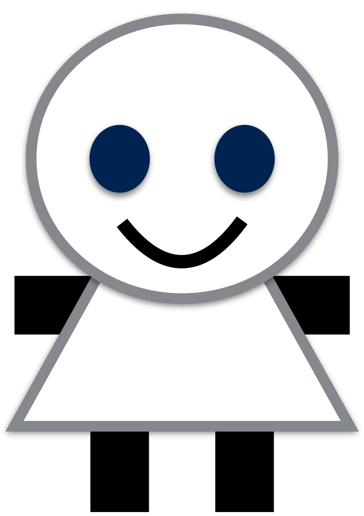
anonymous reviewer

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

ITP2018 review

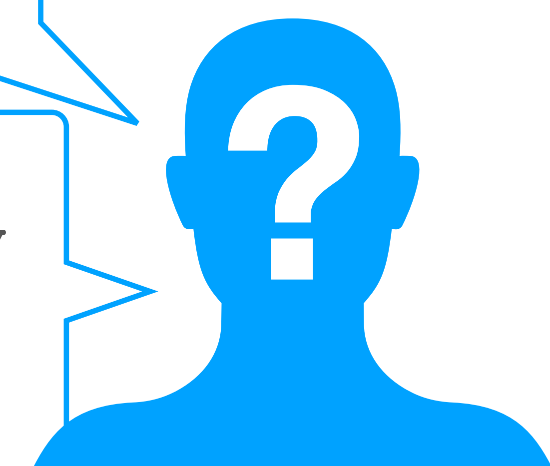
Proof Method Recommendation, PaMpeR!





I have doubts about various approaches proposed in the paper.

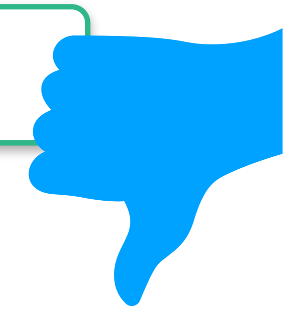
New users of Isabelle are facing many challenges from
- writing their first definitions,
- **stating suitable theorem statements….**

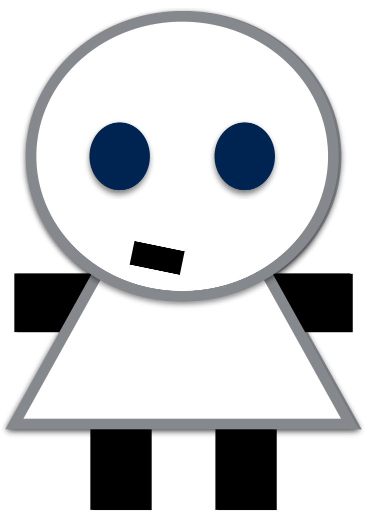


anonymous reviewer

ITP2018 review

Proof Method Recommendation, PaMpeR!

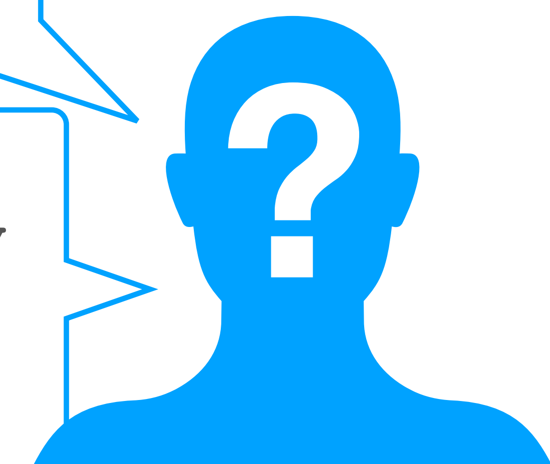




I have doubts about various approaches proposed in the paper.

New users of Isabelle are facing many challenges from

- writing their first definitions,
- **stating suitable theorem statements….**



anonymous reviewer

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

ITP2018 review

Proof Method Recommendation, PaMpeR!

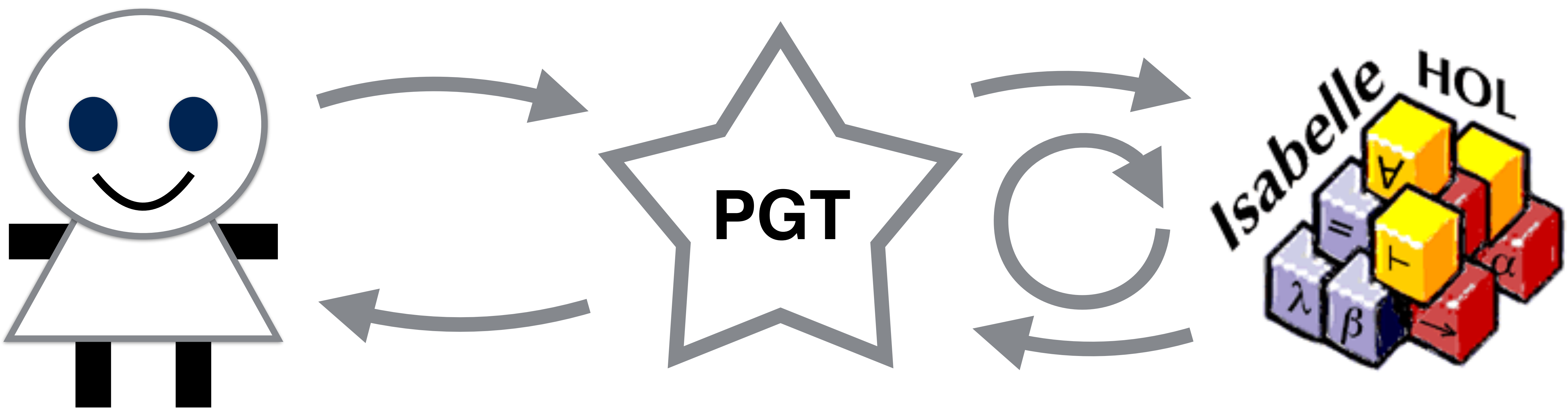
I have doubts about various approaches proposed in the paper.

New users of Isabelle are facing many challenges from
- writing their first definitions,
- **stating suitable theorem statements….**

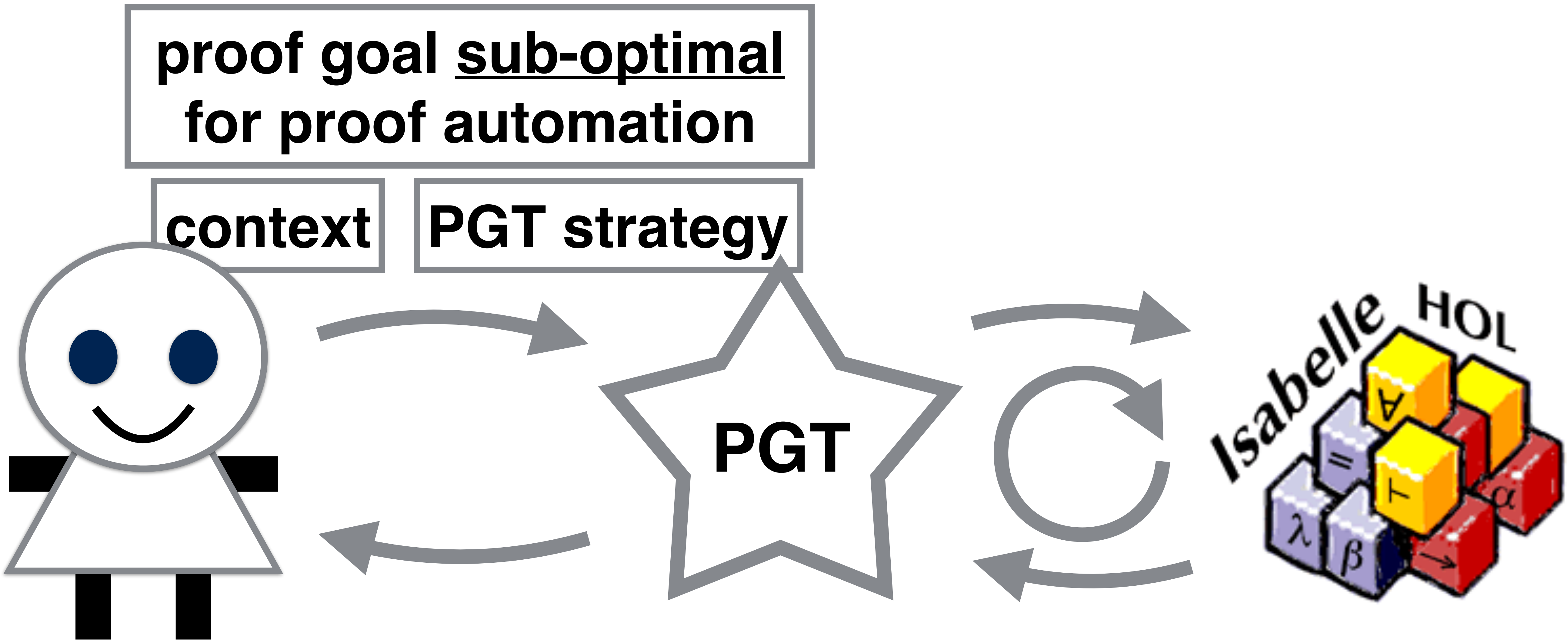
anonymous reviewer

Proof Goal Transformer, PGT!

PSL with PGT

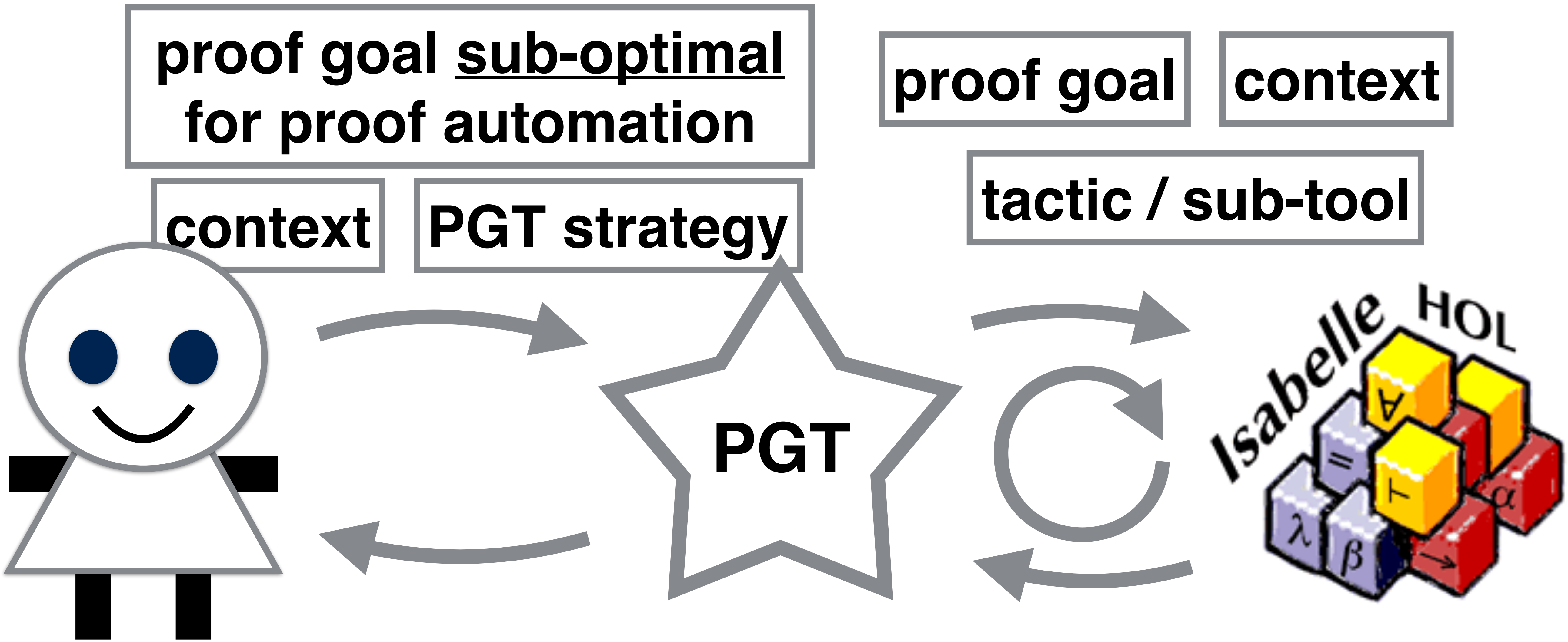


PSL with PGT


proof goal sub-optimal
for proof automation
context
PGT strategy
PGT
Isabelle
HOL

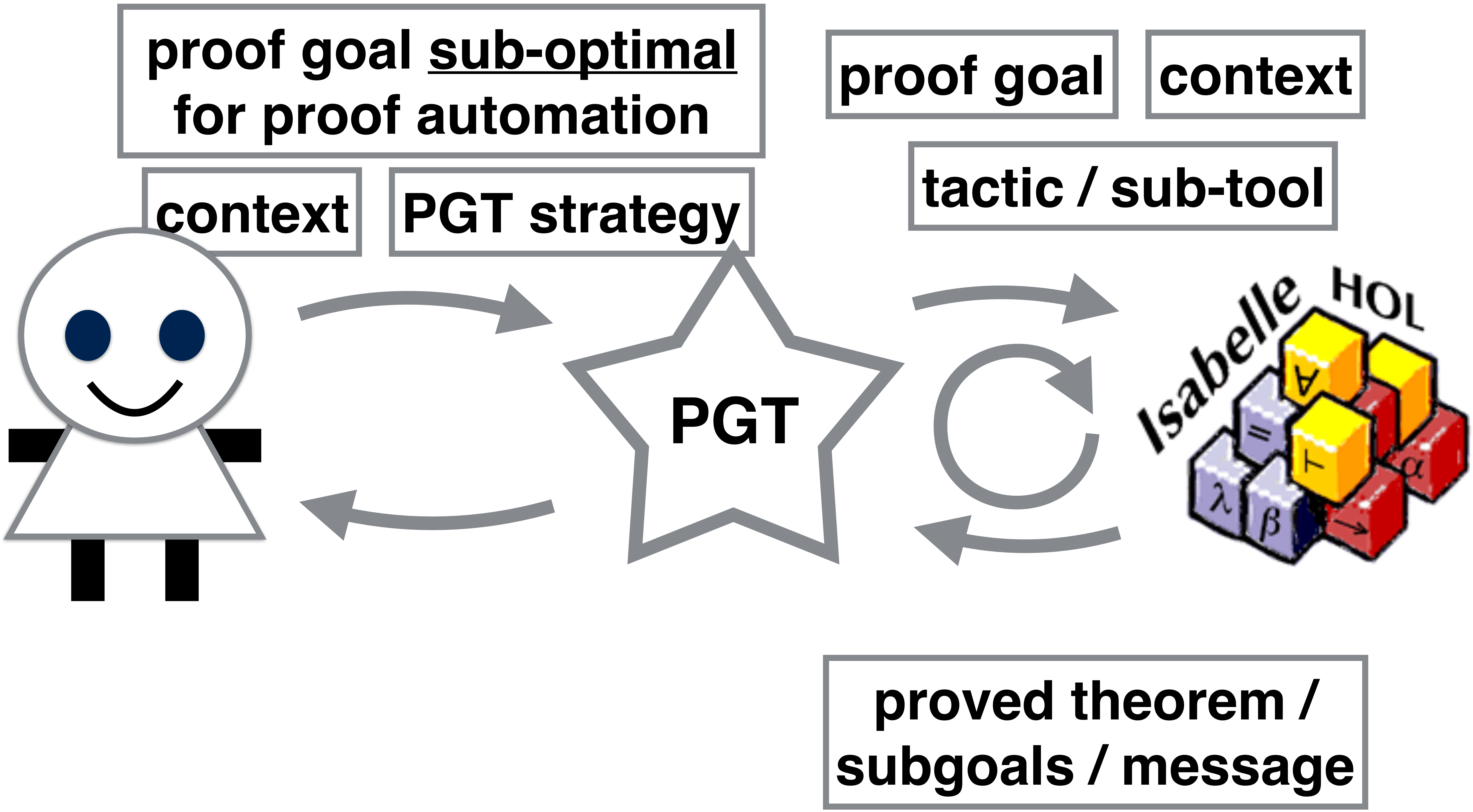
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

PSL with PGT



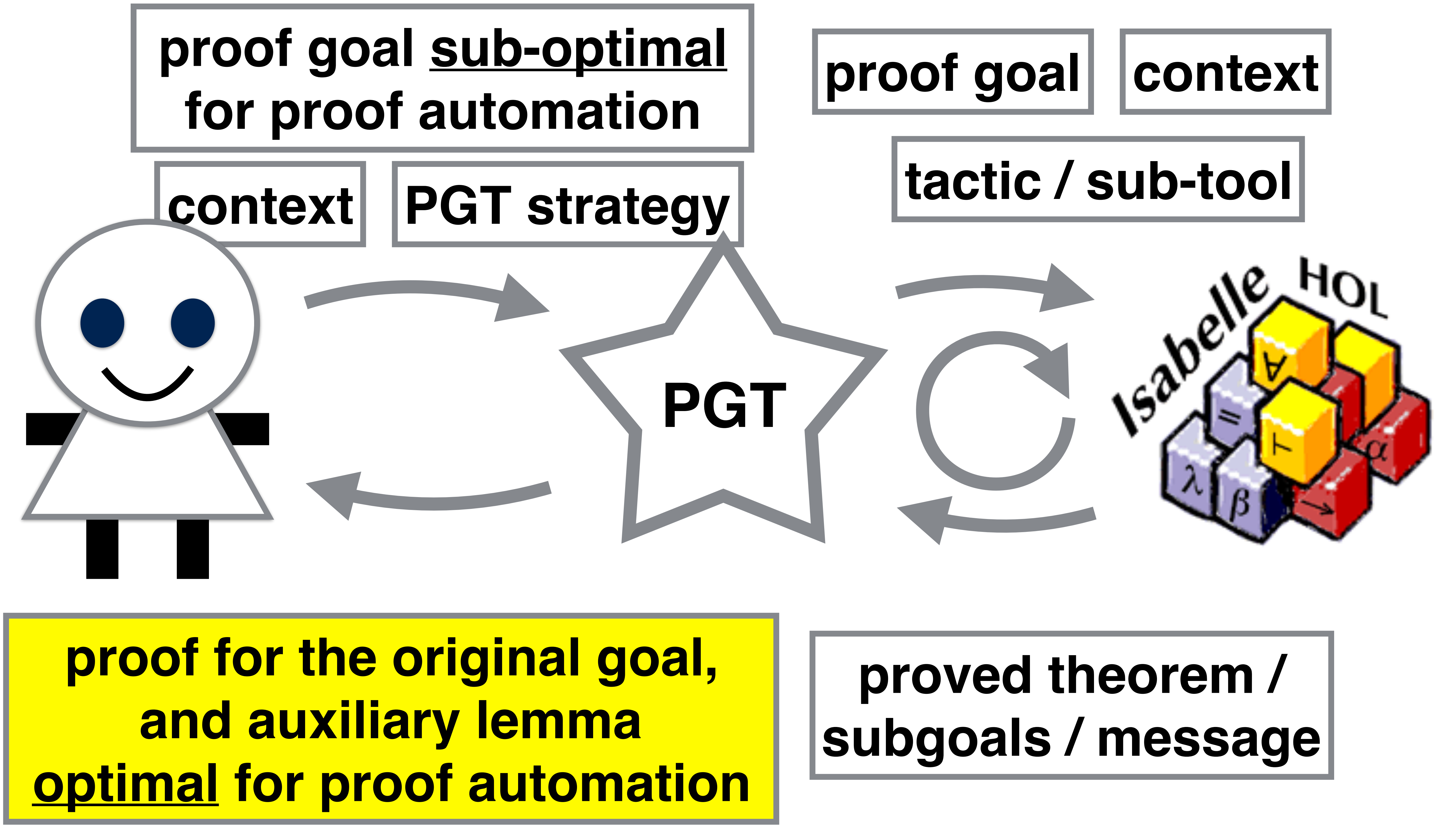
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

PSL with PGT



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

PSL with PGT



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

PSL with PGT

proof goal <u>sub-optimal</u> for proof automation

context

PGT strategy

proof goal

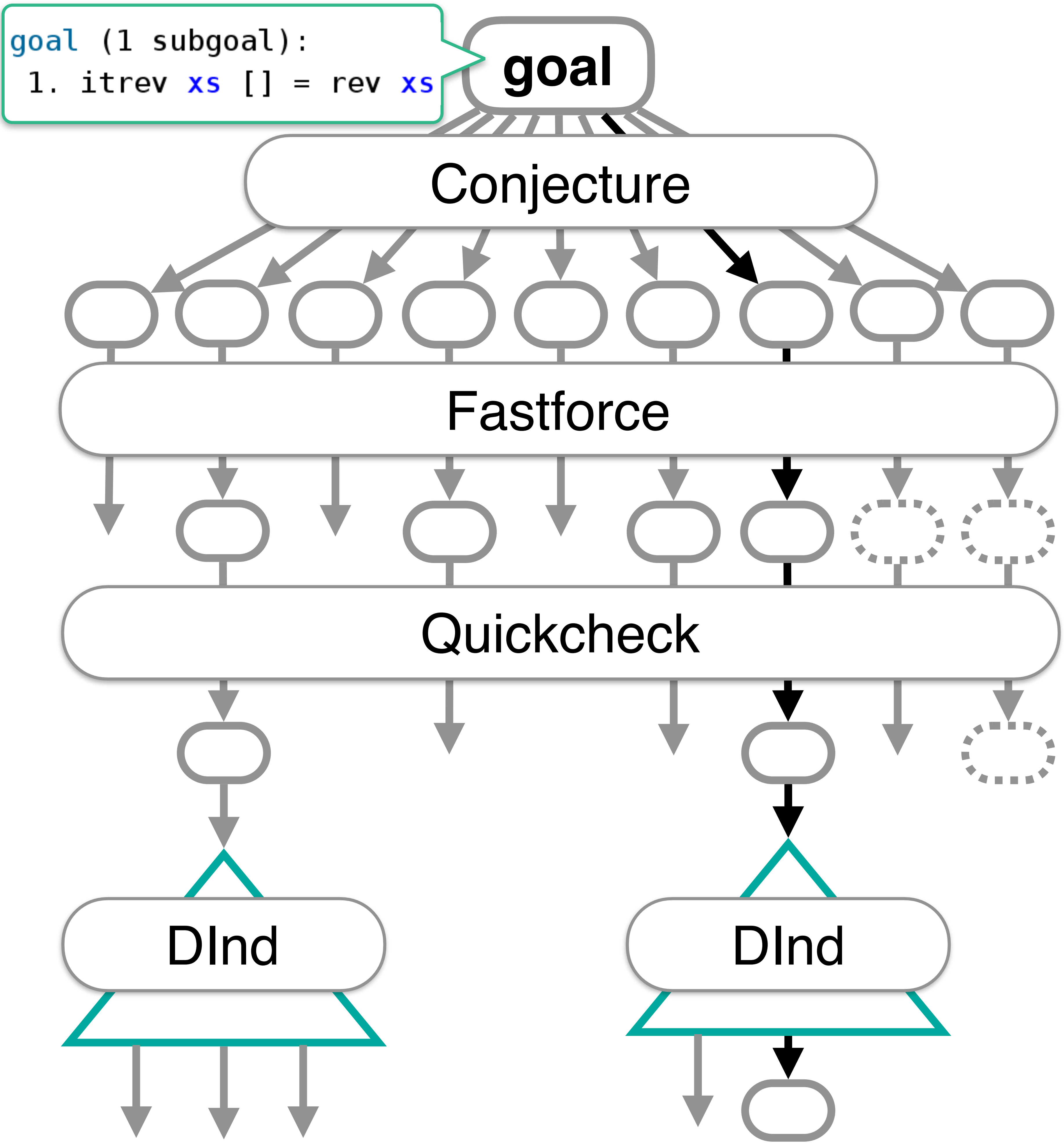
context

tactic / sub-tool

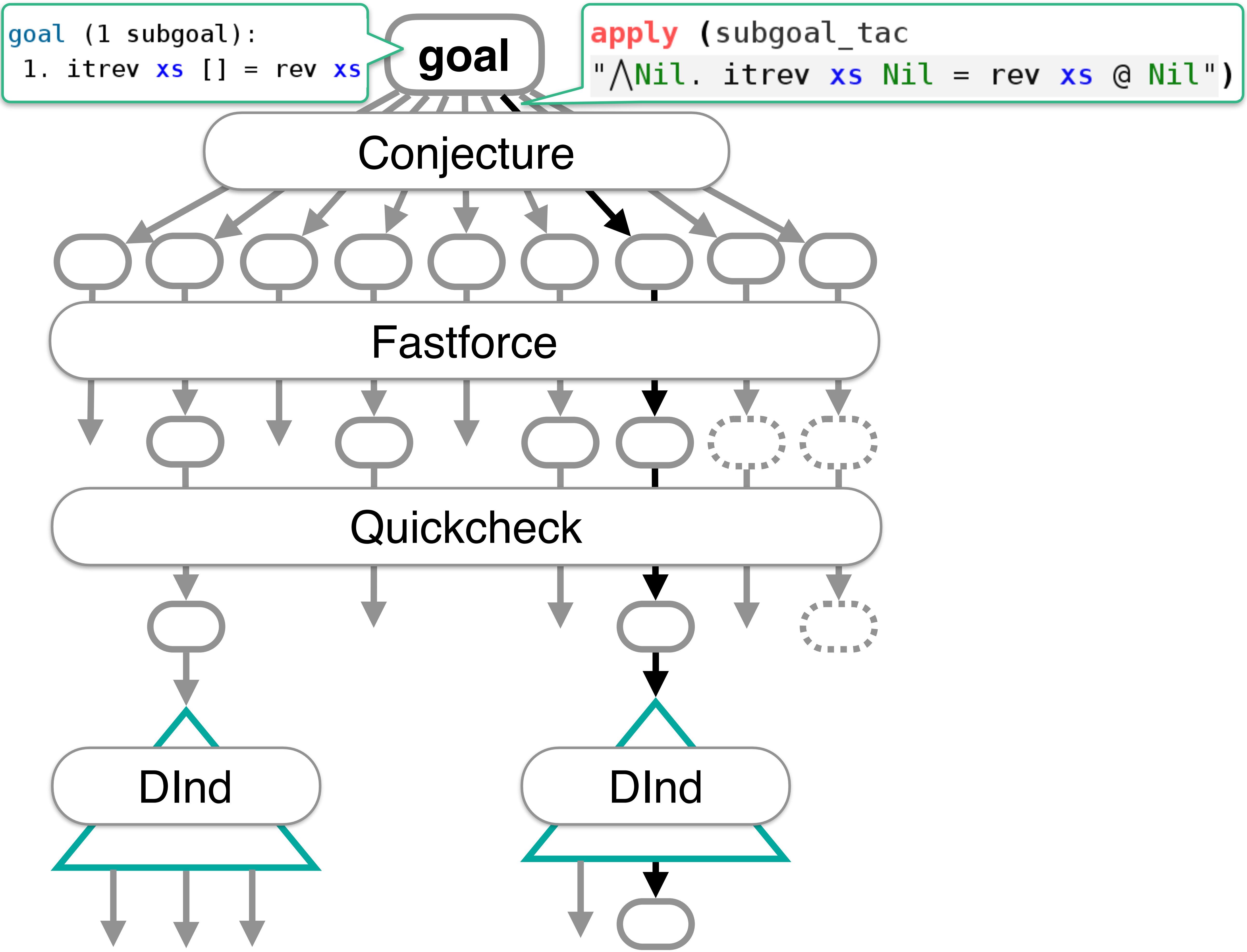
proof for the original goal, and auxiliary lemma <u>optimal</u> for proof automation

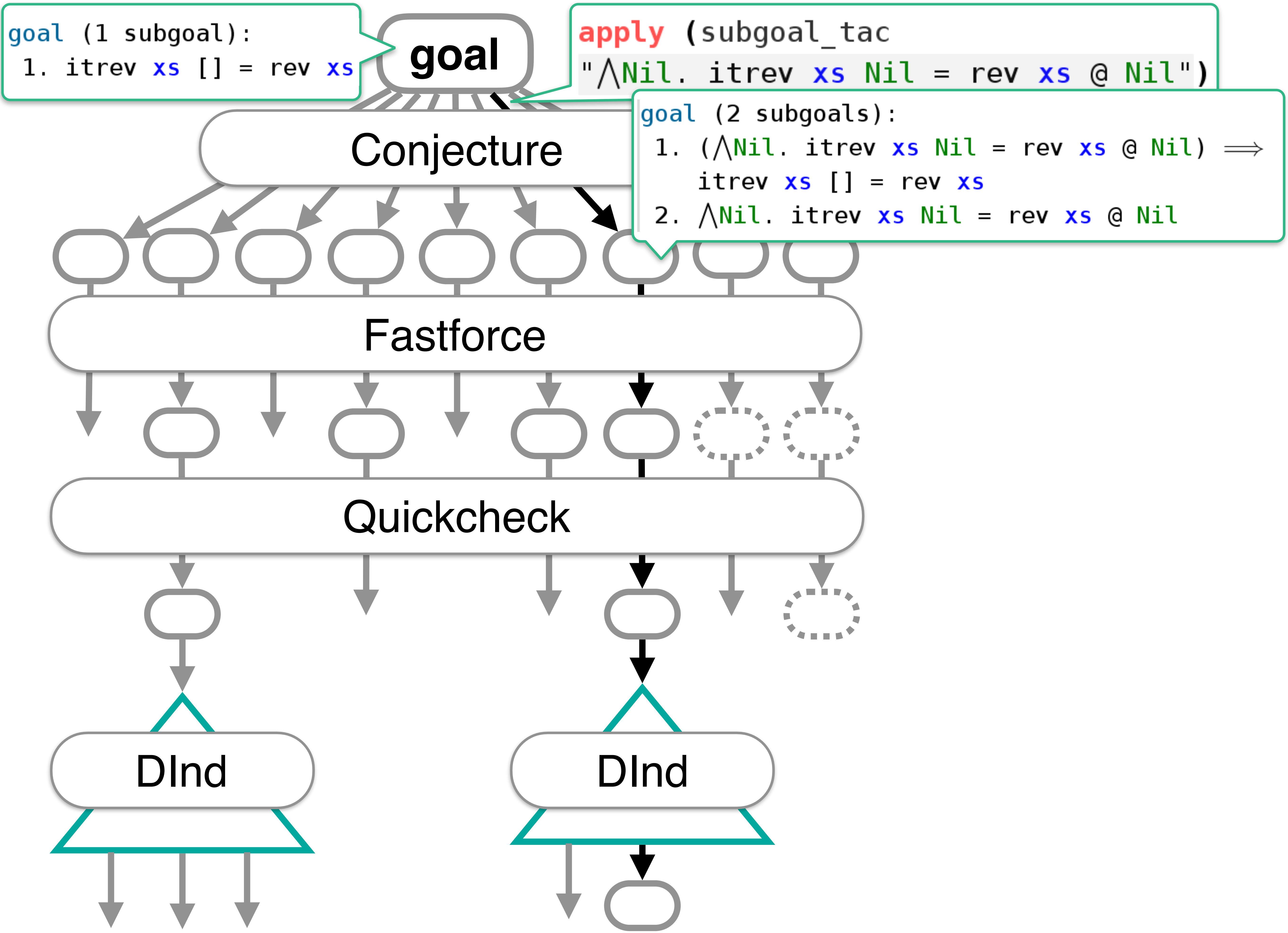
proved theorem / subgoals / …ssage

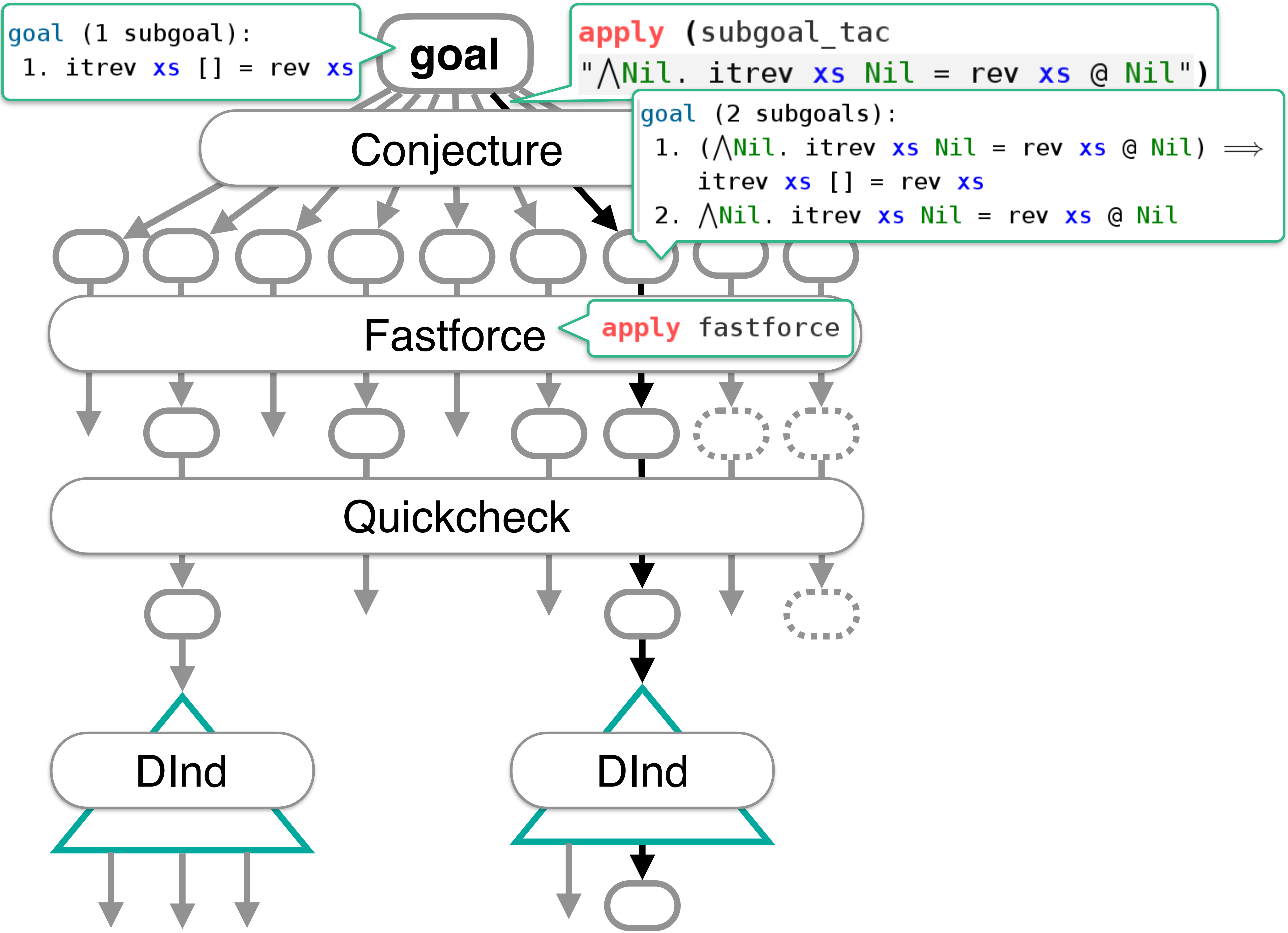
DEMO!

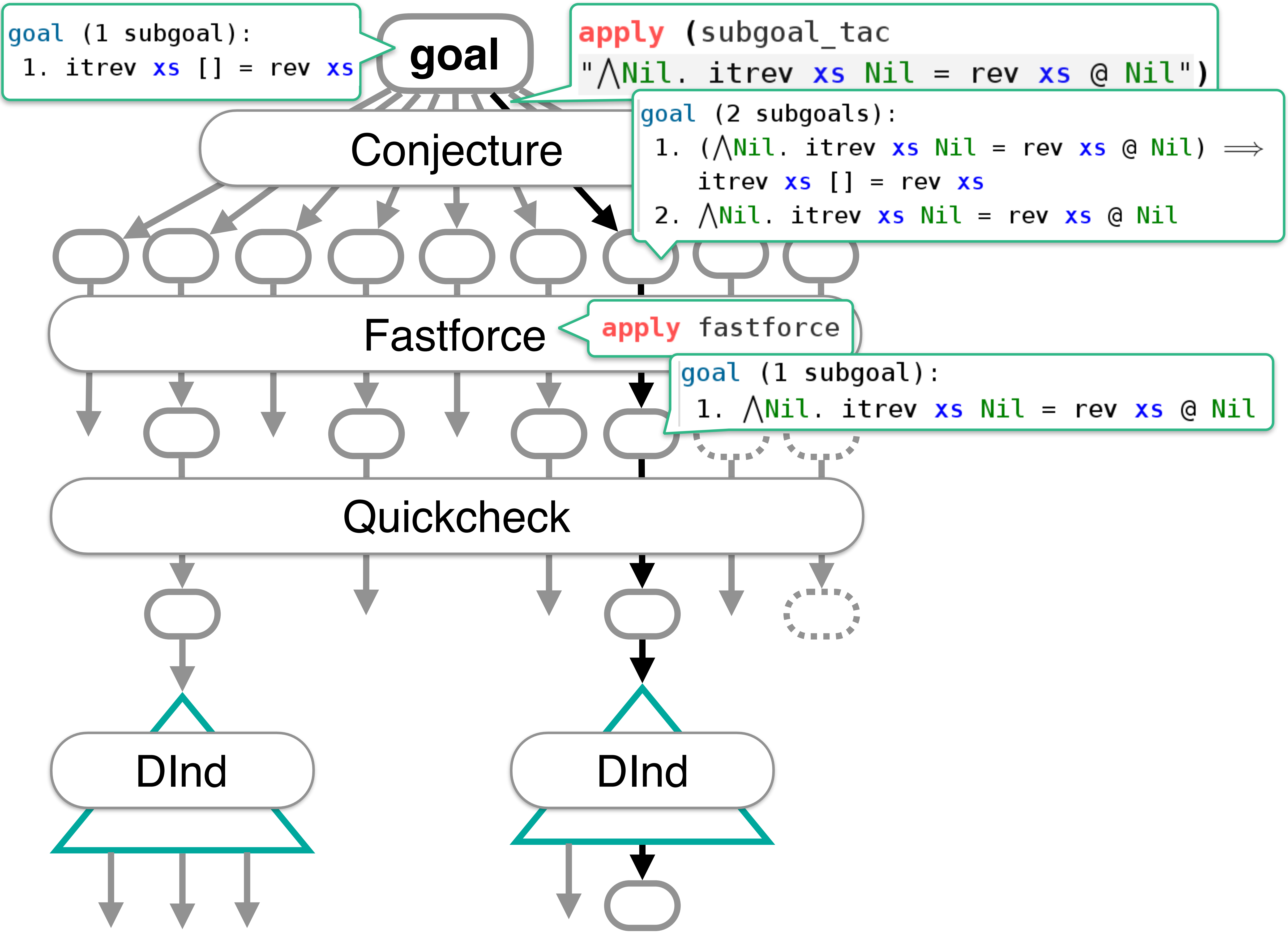


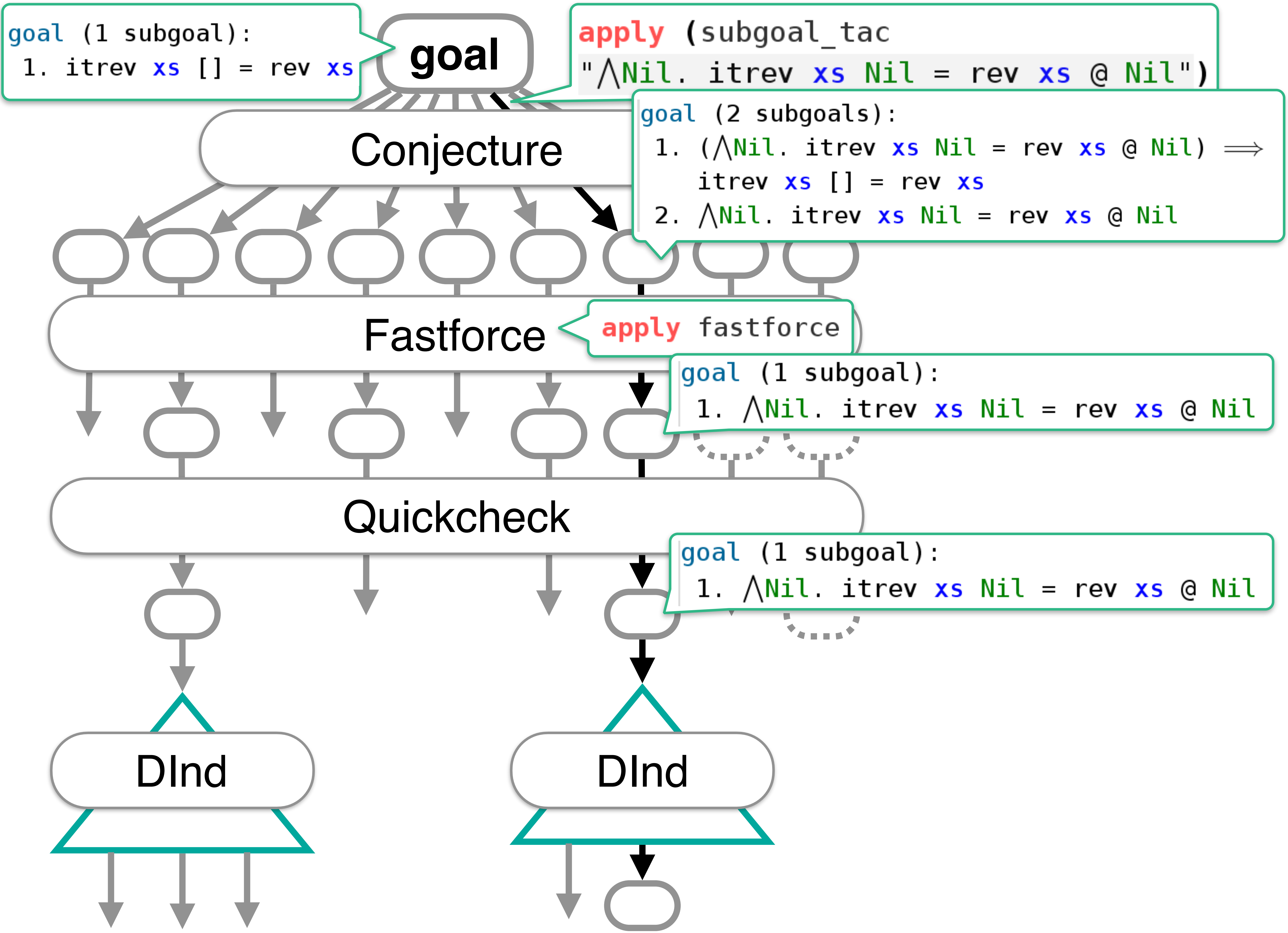
goal (1 subgoal):
1. itrev xs [] = rev xs
goal
Conjecture
Fastforce
Quickcheck
DInd
DInd


goal (1 subgoal):
1. itrev xs [] = rev xs
goal
apply (subgoal_tac
"⋀Nil. itrev xs Nil = rev xs @ Nil")
Conjecture
Fastforce
Quickcheck
DInd
DInd

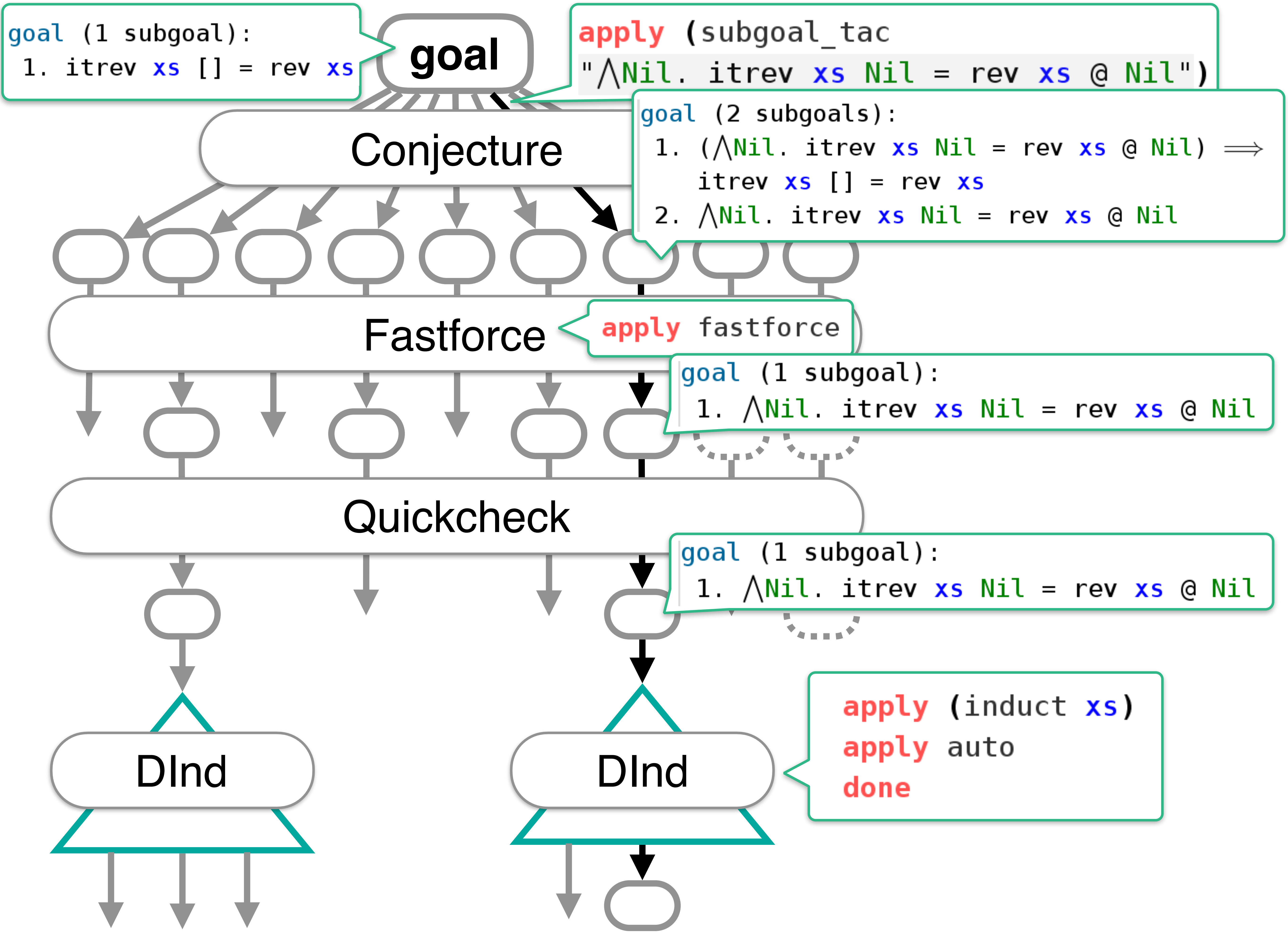

goal (1 subgoal):
1. itrev xs [] = rev xs
goal
apply (subgoal_tac
"⋀Nil. itrev xs Nil = rev xs @ Nil")
goal (2 subgoals):
1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
itrev xs [] = rev xs
2. ⋀Nil. itrev xs Nil = rev xs @ Nil
Conjecture
Fastforce
Quickcheck
DInd
DInd


goal (1 subgoal):
1. itrev xs [] = rev xs
goal
apply (subgoal_tac
"⋀Nil. itrev xs Nil = rev xs @ Nil")
Conjecture
goal (2 subgoals):
1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
itrev xs [] = rev xs
2. ⋀Nil. itrev xs Nil = rev xs @ Nil
Fastforce
apply fastforce
Quickcheck
DInd
DInd

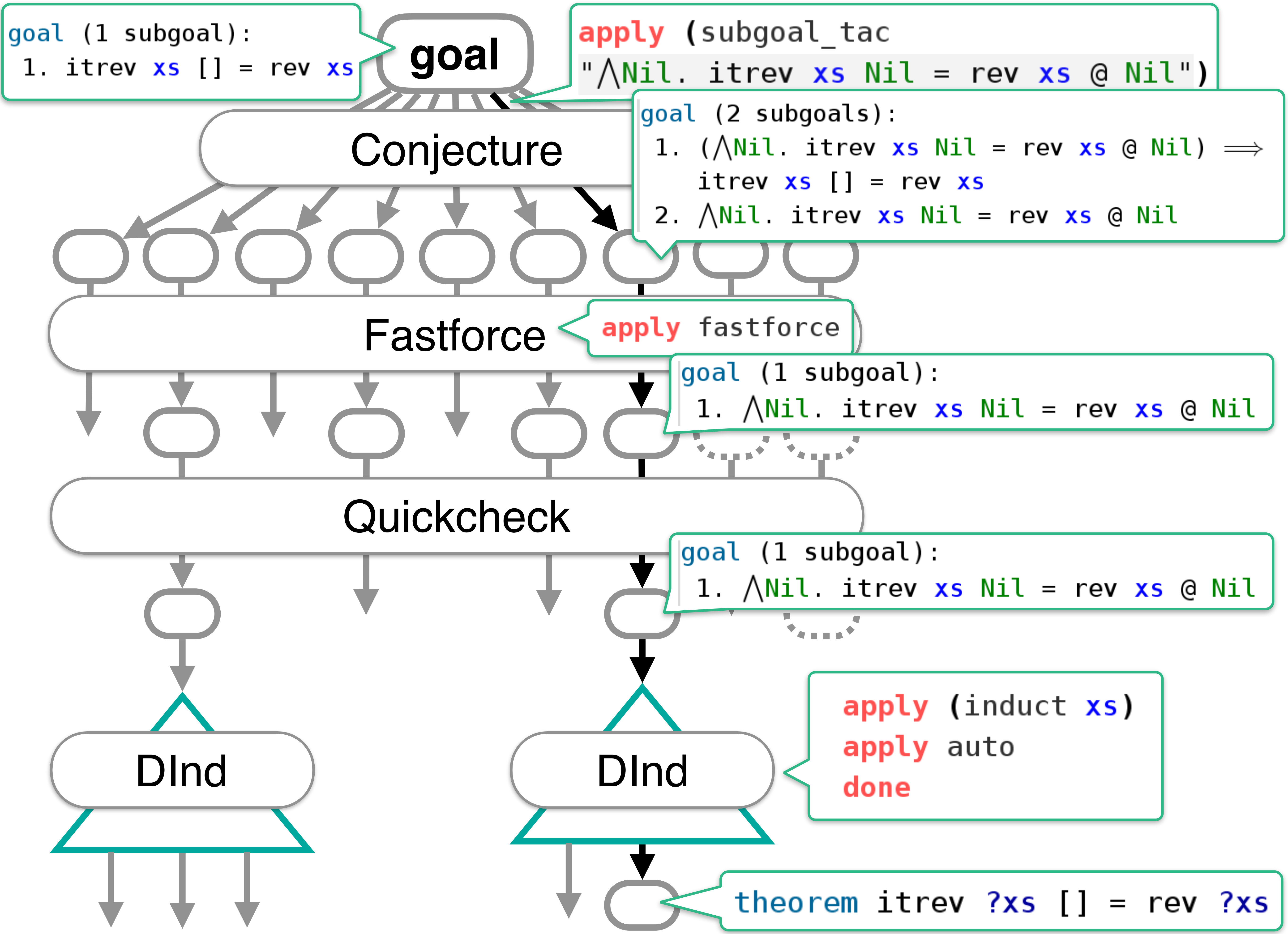

goal (1 subgoal):
1. itrev xs [] = rev xs
goal
apply (subgoal_tac
"⋀Nil. itrev xs Nil = rev xs @ Nil")
Conjecture
goal (2 subgoals):
1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
itrev xs [] = rev xs
2. ⋀Nil. itrev xs Nil = rev xs @ Nil
Fastforce
apply fastforce
goal (1 subgoal):
1. ⋀Nil. itrev xs Nil = rev xs @ Nil
Quickcheck
DInd
DInd



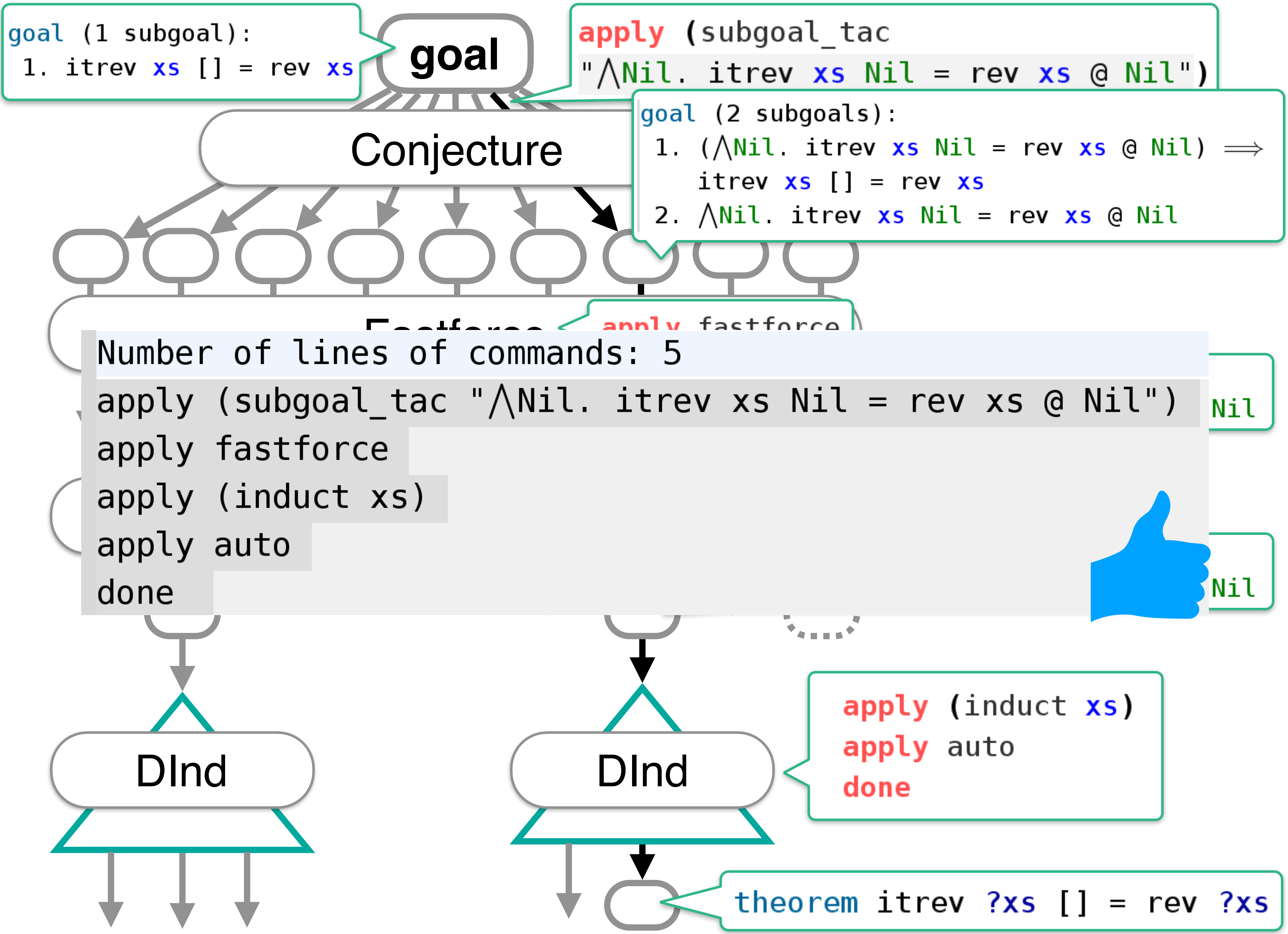
goal (1 subgoal):
1. itrev xs [] = rev xs
goal
apply (subgoal_tac "⋀Nil. itrev xs Nil = rev xs @ Nil")
Conjecture
goal (2 subgoals):
1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹ itrev xs [] = rev xs
2. ⋀Nil. itrev xs Nil = rev xs @ Nil
Fastforce
apply fastforce
goal (1 subgoal):
1. ⋀Nil. itrev xs Nil = rev xs @ Nil
Quickcheck
goal (1 subgoal):
1. ⋀Nil. itrev xs Nil = rev xs @ Nil
DInd
DInd



goal (1 subgoal):
1. itrev xs [] = rev xs
goal
apply (subgoal_tac
"⋀Nil. itrev xs Nil = rev xs @ Nil")
Conjecture
goal (2 subgoals):
1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
itrev xs [] = rev xs
2. ⋀Nil. itrev xs Nil = rev xs @ Nil
Fastforce
apply fastforce
goal (1 subgoal):
1. ⋀Nil. itrev xs Nil = rev xs @ Nil
Quickcheck
goal (1 subgoal):
1. ⋀Nil. itrev xs Nil = rev xs @ Nil
DInd
DInd
apply (induct xs)
apply auto
done



goal (1 subgoal):
1. itrev xs [] = rev xs
goal
apply (subgoal_tac "⋀Nil. itrev xs Nil = rev xs @ Nil")
goal (2 subgoals):
1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹ itrev xs [] = rev xs
2. ⋀Nil. itrev xs Nil = rev xs @ Nil
Conjecture
Fastforce
apply fastforce
goal (1 subgoal):
1. ⋀Nil. itrev xs Nil = rev xs @ Nil
Quickcheck
goal (1 subgoal):
1. ⋀Nil. itrev xs Nil = rev xs @ Nil
DInd
DInd
apply (induct xs)
apply auto
done
theorem itrev ?xs [] = rev ?xs


goal (1 subgoal):
1. itrev xs [] = rev xs
goal
apply (subgoal_tac
"⋀Nil. itrev xs Nil = rev xs @ Nil")
Conjecture
goal (2 subgoals):
1. (⋀Nil. itrev xs Nil = rev xs @ Nil) ⟹
itrev xs [] = rev xs
2. ⋀Nil. itrev xs Nil = rev xs @ Nil
Number of lines of commands: 5
apply (subgoal_tac "⋀Nil. itrev xs Nil = rev xs @ Nil")
apply fastforce
apply (induct xs)
apply auto
done
DInd
DInd
apply (induct xs)
apply auto
done
theorem itrev ?xs [] = rev ?xs

Success story

PSL can find how to apply induction for easy problems.

Success story

PSL can find how to apply induction for easy problems.

PaMpeR recommends which proof methods to use.

Success story

PSL can find how to apply induction for easy problems.

PaMpeR recommends which proof methods to use.

PGT produces useful auxiliary lemmas.

Success story

PSL can find how to apply induction for easy problems.

CADE2017

PaMpeR recommends which proof methods to use.

PGT produces useful auxiliary lemmas.

Success story

PSL can find how to apply induction for easy problems.

PaMpeR recommends which proof methods to use.

PGT produces useful auxiliary lemmas.

Success story

PSL can find how to apply induction for easy problems.

PaMpeR recommends which proof methods to use.

PGT produces useful auxiliary lemmas.

Too good to be true?

PSL can find how to apply induction for easy problems.

PaMpeR recommends which proof methods to use.

PGT produces useful auxiliary lemmas.

Too good to be true?

PSL can find how to apply induction for easy problems

only if PSL completes a proof search

PaMpeR recommends which proof methods to use.

PGT produces useful auxiliary lemmas.

only if PSL with PGT completes a proof search

Too good to be true?

PSL can find how to apply induction for easy problems

only if PSL completes a proof search

PaMpeR recommends which proof methods to use.

but PaMpeR does not recommend arguments for proof methods

PGT produces useful auxiliary lemmas.

only if PSL with PGT completes a proof search

Too good to be true?

PSL can find how to apply induction for easy problems

only if PSL completes a proof search

PaMpeR recommends which proof methods to use.

but PaMpeR does not recommend arguments for proof methods

PGT produces useful auxiliary lemmas.

only if PSL with PGT compl

proof search

Recommend how to apply induction without completing a proof.

Too good to be true?

PSL can find how to apply induction for easy problems

only if PSL completes a proof search

PaMpeR recommends which proof methods to use.

but PaMpeR does not recommend arguments for proof methods

PGT produces useful auxiliary lemmas.

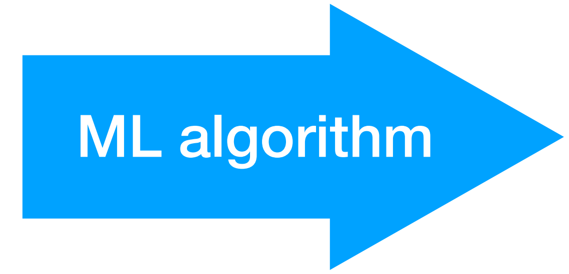
only if PSL with PGT compl proof search

Recommend how to apply induction without completing a proof.

MeLoId: Machine Learning Induction

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

Introduction to Machine Learning in 10 seconds



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

Introduction to Machine Learning in 10 seconds

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

Introduction to Machine Learning in 10 seconds

big data

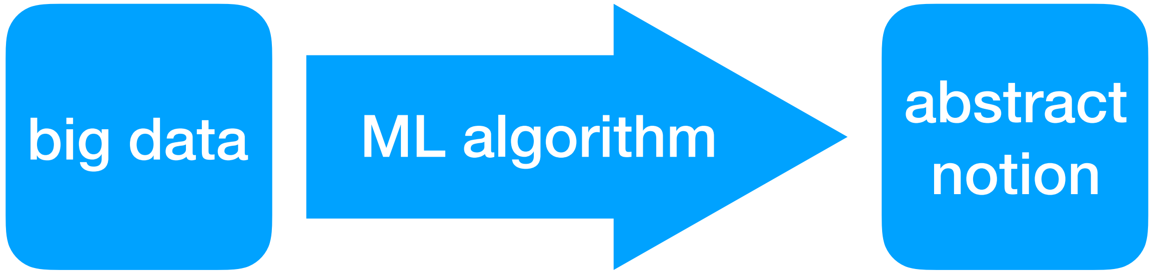
ML algorithm

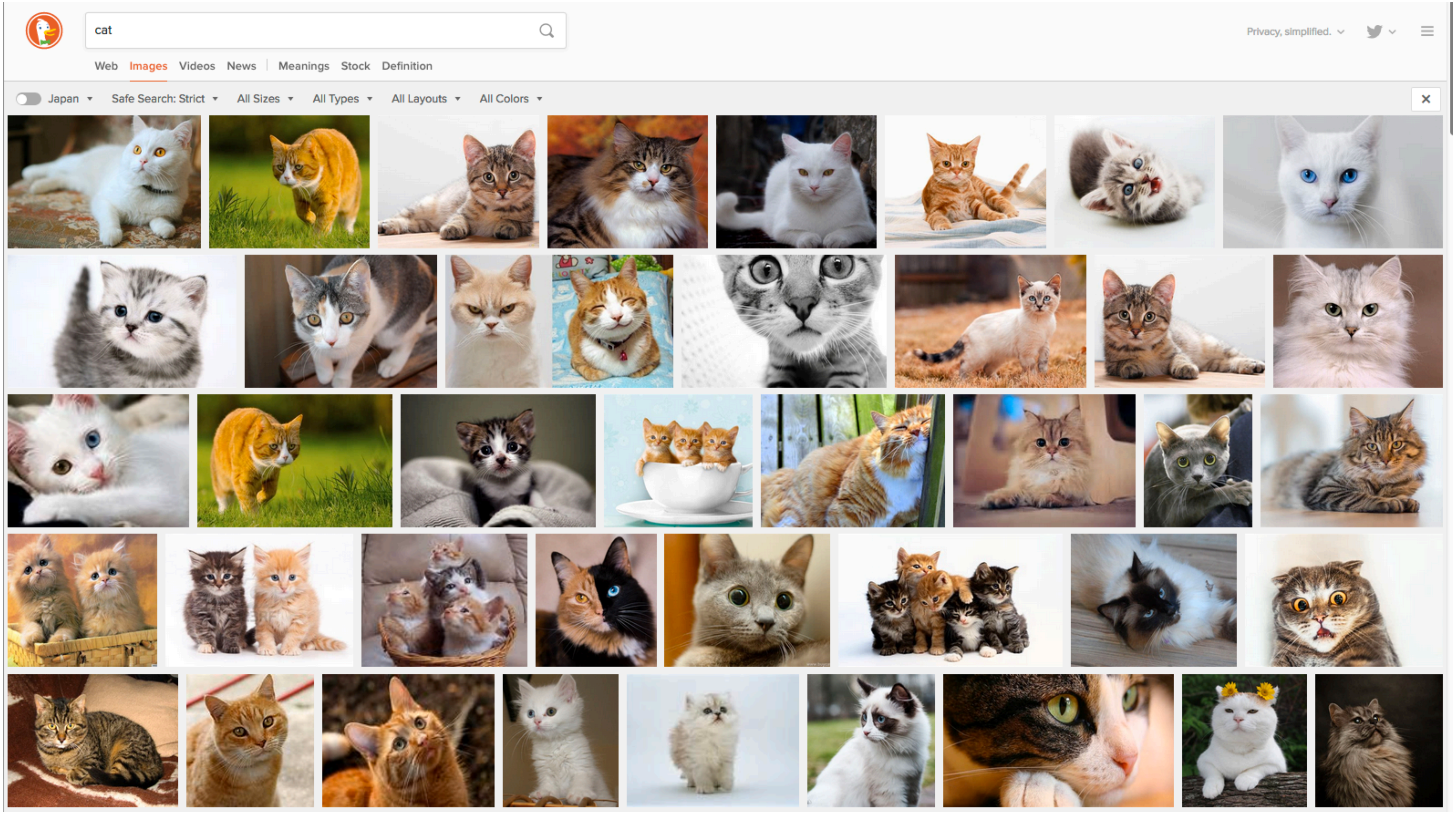
abstract notion

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

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

Introduction to Machine Learning in 10 seconds

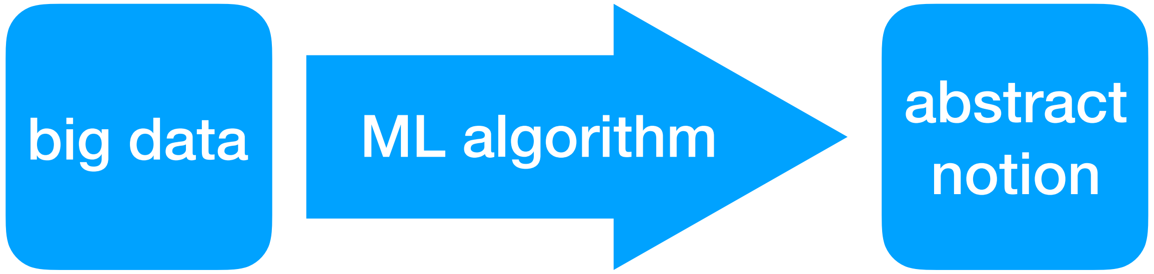


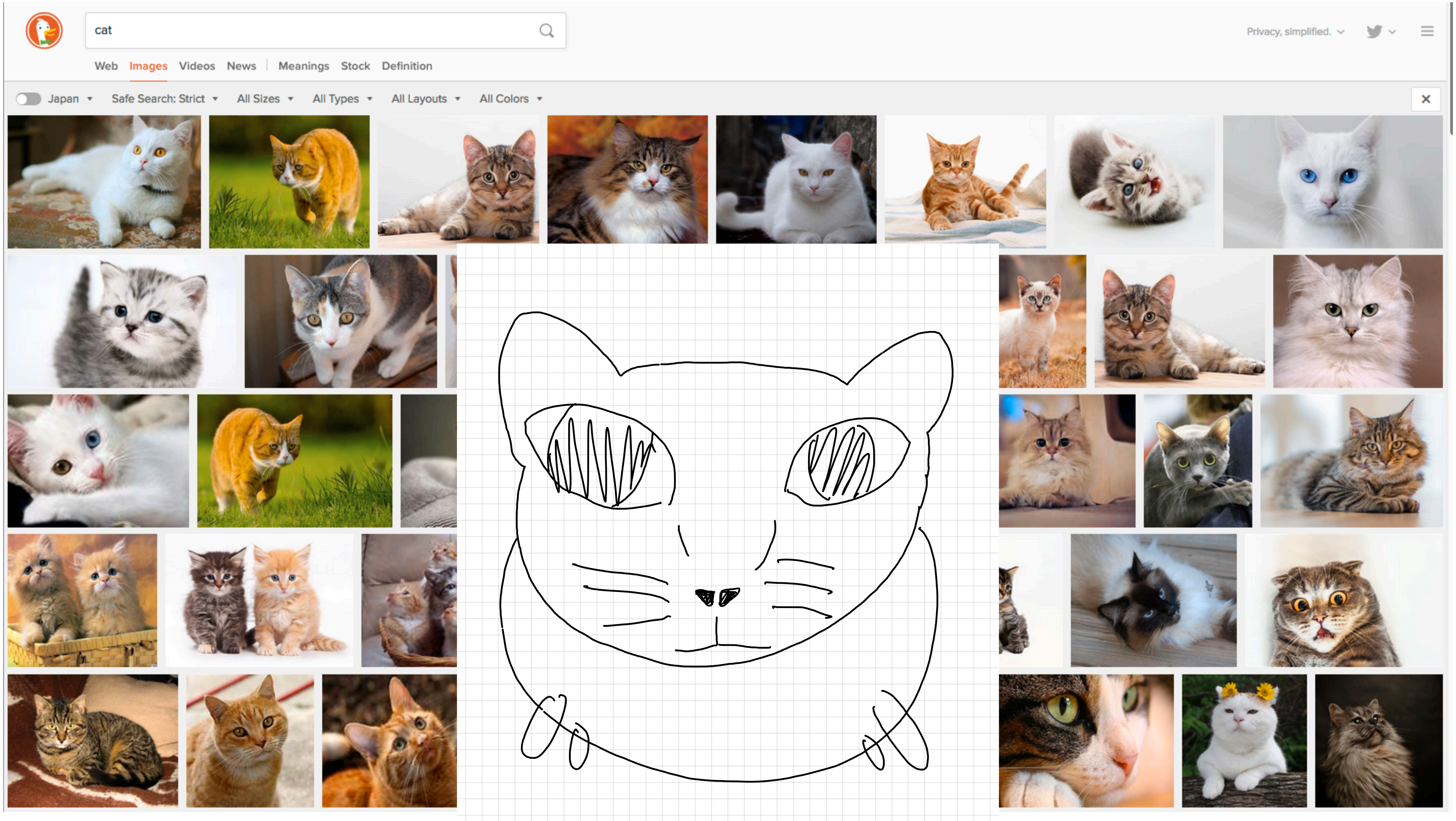


https://duckduckgo.com/?q=cat&t=ffab&iar=images&iax=images&ia=images

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

Introduction to Machine Learning in 10 seconds





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

ML for Inductive Theorem Proving
the BAD

```
lemma "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
```

ML for Inductive Theorem Proving
the BAD

```
lemma "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

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

ML for Inductive Theorem Proving
the BAD

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

<- one abstract representation

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

<- many concrete cases

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

ML for Inductive Theorem Proving
the BAD

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

<- one abstract representation

ML

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

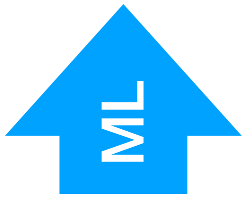
<- many concrete cases

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

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
```

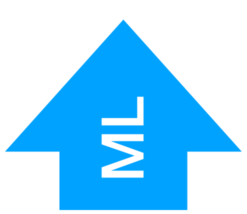
<- many concrete cases

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

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

ML for Inductive Theorem Proving
the BAD

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

<- one abstract representation

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

ML

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

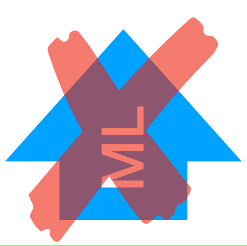
<- many concrete cases

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

ML for Inductive Theorem Proving
the BAD

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

<- one abstract representation




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

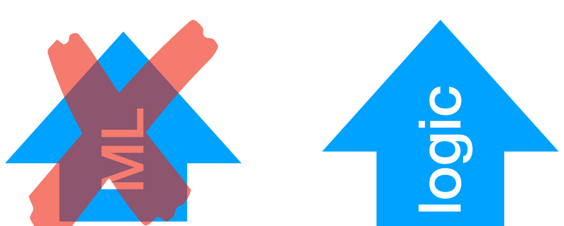
<- many concrete cases

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

ML for Inductive Theorem Proving
the BAD

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

<- one abstract representation



<- abstraction using expressive logic

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

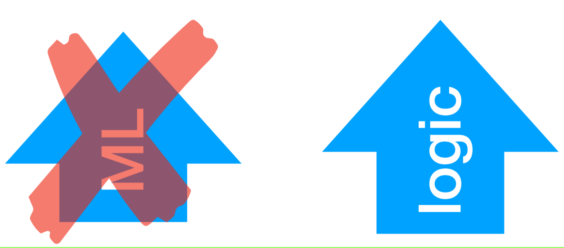
<- many concrete cases

ML for Inductive Theorem Proving
the BAD

polymorphism

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

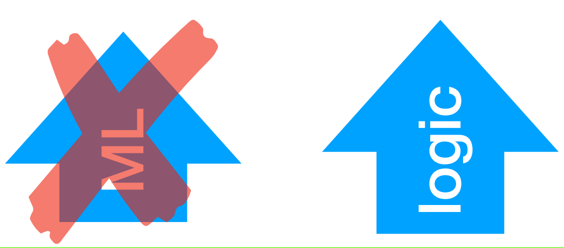
https://twitter.com/YutakangJ https://github.com/data61/PSL/slide/2019_ps.pdf

ML for Inductive Theorem Proving
the BAD

polymorphism **type class**

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

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

ML for Inductive Theorem Proving
the BAD

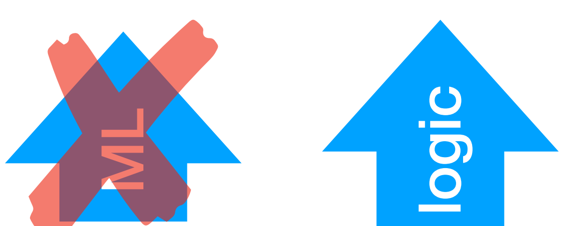
polymorphism

type class

universal quantifier

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

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

ML for Inductive Theorem Proving
the BAD

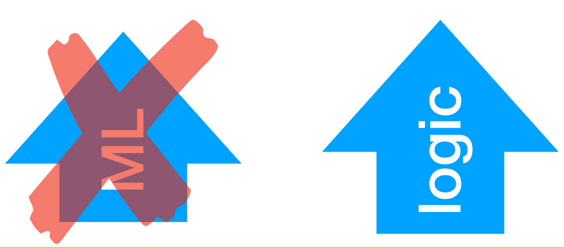
Higher-Order functions

polymorphism **type class**

universal quantifier

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

ML for Inductive Theorem Proving
the BAD

Higher-Order functions

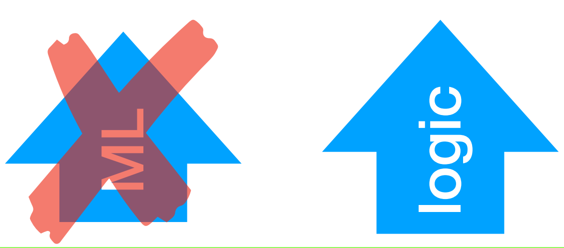
polymorphism **type class**

universal quantifier

lambda abstraction

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

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

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

~~ML~~

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

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

~~ML~~

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

ML for Inductive Theorem Proving
the BAD

Higher-Order functions

polymorphism **type class**

universal quantifier

lambda abstraction

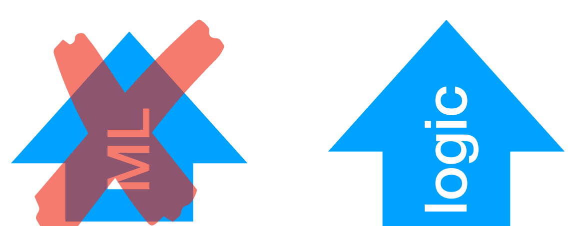
concise formula that can cover many concrete cases

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



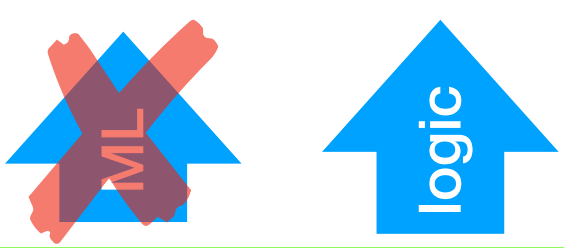
<- abstraction using expressive logic

```
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

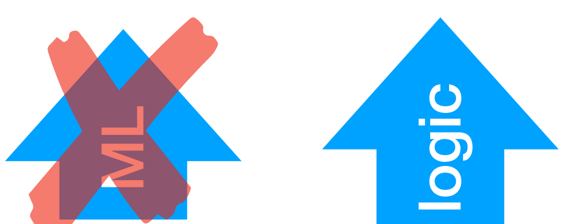
```
lemma "itrev [1,2]         [] = rev [1,2]         @ []" by auto
lemma "itrev [1,2,3]       [] = rev [1,2,3]       @ []" by auto
lemma "itrev [''a'',''b''] [] = rev [''a'',''b''] @ []" by auto
lemma "itrev [x,y,z]       [] = rev [x,y,z]       @ []" by auto
```

<- 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
```

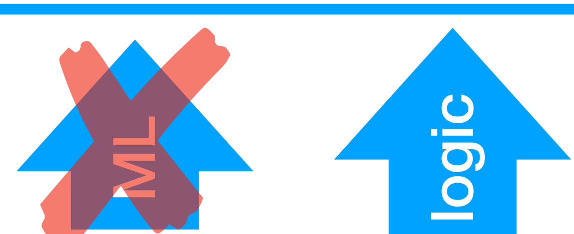
<- many concrete cases

Grand Challenge: Abstract Abstraction

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

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

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

<- small dataset about different domains

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

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

Grand Challenge: Abstract Abstraction

ML

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

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

<- small dataset about different domains

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

<- one abstract representation

~~ML~~ logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Grand Challenge: Abstract Abstraction

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

<- even more abstract

ML

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

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

<- small dataset about different domains

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

<- one abstract representation

~~ML~~ logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Grand Challenge: Abstract Abstraction

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

<- even more abstract

ML

<- pros: good at ambiguity (heuristics)

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

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

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

<- small dataset about different domains

<- one abstract representation

~~ML~~ logic

<- abstraction using expressive logic

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

<- many concrete cases

Grand Challenge: Abstract Abstraction

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

<- even more abstract

ML

<- pros: good at ambiguity (heuristics)

<- cons: bad at reasoning & abstraction

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

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

<- small dataset about different domains

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

<- one abstract representation

~~ML~~ logic

<- abstraction using expressive logic

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

<- many concrete cases

Many key challenges remain

Unsupervised Learning

Memory and one-shot learning

Imagination-based Planning with Generative Models

Learning Abstract Concepts

Transfer Learning

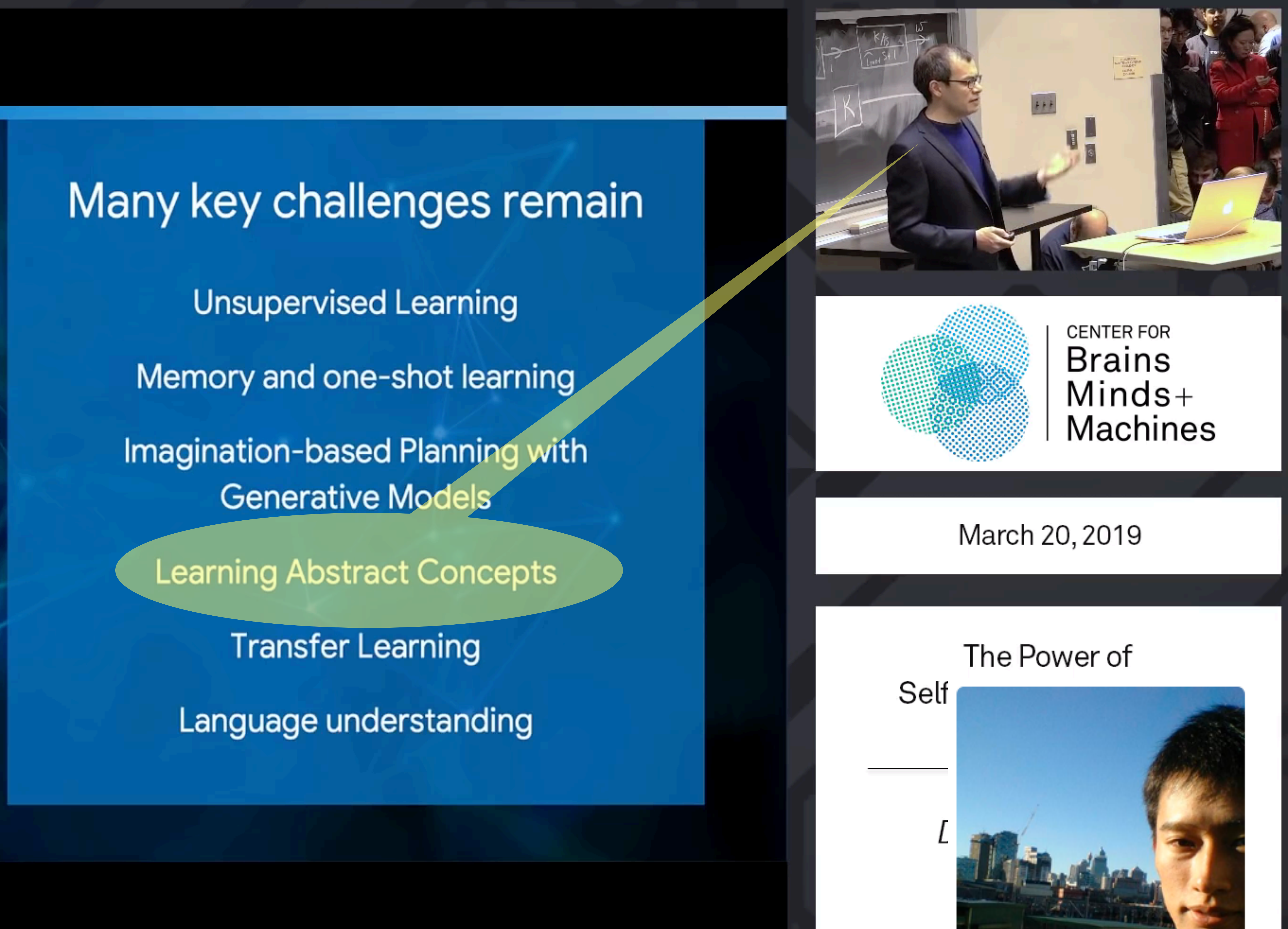
Language understanding

CENTER FOR
Brains
Minds+
Machines

March 20, 2019

The Power of
Self

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



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

Abstract conceptsといえば

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

Many key challenges remain

Unsupervised Learning

Memory and one-shot learning

Imagination-based Planning with Generative Models

Learning Abstract Concepts

CENTER FOR
Brains
Minds+
Machines

March 20, 2019

The Power of
Self

Abstract conceptsといえば

論理でしょう。

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

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

Logic about Proofs to Abstract Abstraction

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

<- even more abstract

ML

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

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

<- 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

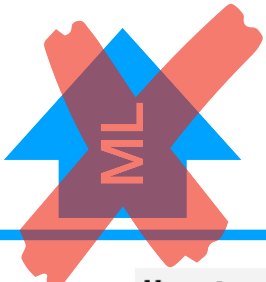
```
lemma "itrev [1,2]         [] = rev [1,2]         @ []" by auto
lemma "itrev [1,2,3]       [] = rev [1,2,3]       @ []" by auto
lemma "itrev [''a'',''b''] [] = rev [''a'',''b''] @ []" by auto
lemma "itrev [x,y,z]       [] = rev [x,y,z]       @ []" by auto
```

<- many concrete cases

Logic about Proofs to Abstract Abstraction

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

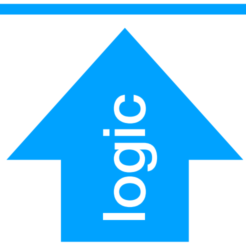
<- even more abstract




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

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

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

<- small dataset about different domains

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

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

Logic about Proofs to Abstract Abstraction

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

<- even more abstract

~~ML~~ logic

abstraction using
another logic (LiFtEr)

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

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

<- small dataset about
different domains

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

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Logic about Proofs to Abstract Abstraction

**Abstract notion of "good" application of induction.
Heuristics that are valid across problem domains.**

<- even more abstract

~~ML~~ logic

abstraction using another logic (LiFtEr)

<- pros: good at rigorous abstraction

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Logic about Proofs to Abstract Abstraction

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

<- even more abstract

~~ML~~ logic

abstraction using another logic (LiFtEr)

<- pros: good at rigorous abstraction

<- cons: bad at ambiguity (heuristics)

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

Logic about Proofs to Abstract Abstraction

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

<- even more abstract

~~ML~~ logic

abstraction using another logic (LiFtEr)

<- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

abstraction using
another logic (LiFtEr)

<- pros: good at rigorous abstraction

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

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

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

<- 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 <- simple representation

abstraction using another logic (LiFtEr) <- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Big Picture

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

LiFtEr 1

abstraction using another logic (LiFtEr)

<- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Big Picture

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

LiFtEr 1

abstraction using another logic (LiFtEr)

<- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Big Picture

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

LiFtEr
LiFtEr 2

abstraction using another logic (LiFtEr)

<- pros: good at rigorous abstraction

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

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

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

<- 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

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

LiFtEr LiFtEr 2

abstraction using another logic (LiFtEr)

<- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Big Picture

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

LiFtEr LiFtEr LiFtEr 3

abstraction using another logic (LiFtEr) <- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Big Picture

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

LiFtEr LiFtEr LiFtEr 3

abstraction using another logic (LiFtEr) <- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

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

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

LiFtEr LiFtEr LiFtEr 3 LiFtEr 1

abstraction using another logic (LiFtEr)

<- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Big Picture

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

LiFtEr LiFtEr LiFtEr 3 LiFtEr 1

abstraction using another logic (LiFtEr)

<- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Big Picture

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

LiFtEr 1 LiFtEr 2 LiFtEr 3 LiFtEr LiFtEr

…raction using other logic (LiFtEr) <- pros: good at rigorous abstraction

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

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

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

LiFtEr 1 LiFtEr 2 LiFtEr 3 LiFtEr LiFtEr

…raction using other logic (LiFtEr)

<- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Big Picture

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

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

...ction using ...ier logic (LiFtEr) <- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Big Picture

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

LiFtEr 1 LiFtEr 2 LiFtEr 3

...ction using ...er logic (LiFtEr)

<- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

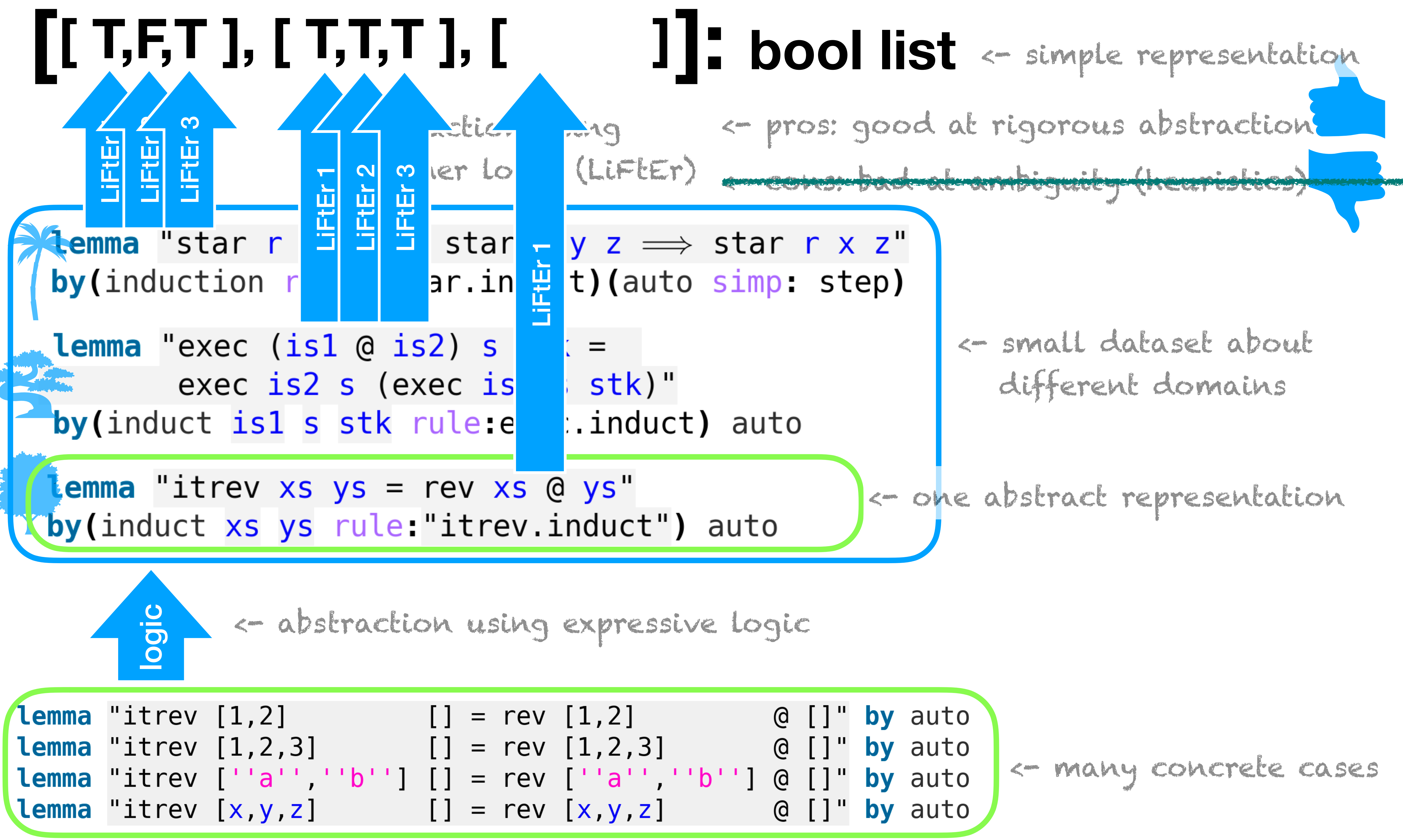
<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

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



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

Big Picture

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

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

...ctic... ...ng
...ier lo... (LiFtEr)

<- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Big Picture

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

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

...ctio... ...er lo... ...iFtEr)

<- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Big Picture

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

LiFtEr 1 LiFtEr 2 LiFtEr 3

<- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Big Picture

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

LiFtEr 1 LiFtEr 2 LiFtEr 3

<- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

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

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

LiFtEr 1 LiFtEr 2 LiFtEr 3

<- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

Big Picture

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

LiFtEr 1 LiFtEr 2 LiFtEr 3

<- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

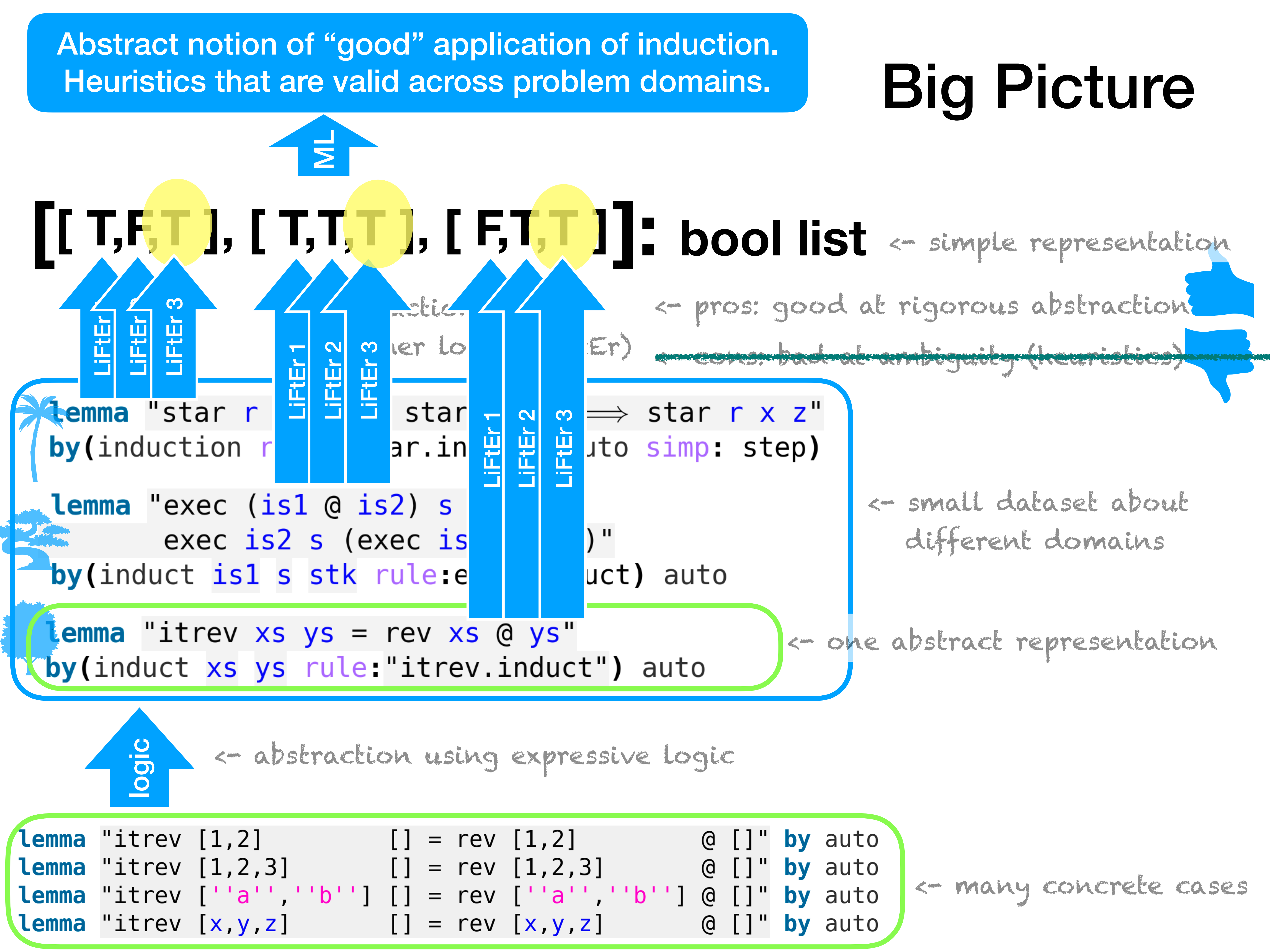
<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases


Abstract notion of "good" application of induction.
Heuristics that are valid across problem domains.
Big Picture
ML
[[T,F,T], [T,T,T], [F,T,T]]: bool list
<- simple representation
<- pros: good at rigorous abstraction
~~<- cons: bad at ambiguity (heuristics)~~
LiFtEr 1
LiFtEr 2
LiFtEr 3
lemma "star r ... star ... ⟹ star r x z"
by(induction r ... ar.in... uto simp: step)
lemma "exec (is1 @ is2) s ...
exec is2 s (exec is ...)"
by(induct is1 s stk rule:e... uct) auto
lemma "itrev xs ys = rev xs @ ys"
by(induct xs ys rule:"itrev.induct") auto
<- small dataset about different domains
<- one abstract representation
logic
<- abstraction using expressive logic
lemma "itrev [1,2] [] = rev [1,2] @ []" by auto
lemma "itrev [1,2,3] [] = rev [1,2,3] @ []" by auto
lemma "itrev [''a'',''b''] [] = rev [''a'',''b''] @ []" by auto
lemma "itrev [x,y,z] [] = rev [x,y,z] @ []" by auto
<- many concrete cases

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

Big Picture

ML

<- pros: good at ambiguity (heuristics)

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

LiFtEr 1 LiFtEr 2 LiFtEr 3

<- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

Big Picture

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

ML

<- pros: good at ambiguity (heuristics)

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

<- pros: good at rigorous abstraction

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

LiFtEr 1 LiFtEr 2 LiFtEr 3

LiFtEr 1 LiFtEr 2 LiFtEr 3

LiFtEr 1 LiFtEr 2 LiFtEr 3

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

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

具体例?

<- 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

Example Assertion in LiFtEr (in Abstract Syntax)

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

$\exists\ r1$: rule. True

$\rightarrow$ variable for auxiliary lemmas

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

Example Assertion in LiFtEr (in Abstract Syntax)

implication

$\exists\ r1$: rule. True

$\rightarrow$

variable for auxiliary lemmas

$\exists\ r1$: rule.

$\exists\ t1$: term. variable for terms

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

$r1$ is_rule_of $to1$

$\wedge$ conjunction

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

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

$\exists\ n$: number.

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

$\wedge$

$t2$ is_nth_induction_term n

Example Assertion in LiFtEr (in Abstract Syntax)

implication

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

variable for auxiliary lemmas

variable for terms

variable for term occurrences

conjunction

Example Assertion in LiFtEr (in Abstract Syntax)

implication

$\exists\ r1$: rule. True

$\rightarrow$

variable for auxiliary lemmas

$\exists\ r1$: rule.

$\exists\ t1$: term. variable for terms

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

variable for term occurrences

$r1$ is_rule_of $to1$

$\wedge$ conjunction

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

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

$\exists\ n$: number. variable for natural numbers

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

$\wedge$

$t2$ is_nth_induction_term n

Example Assertion in LiFtEr (in Abstract Syntax)

implication

$\exists\, r1$: `rule. True`

$\rightarrow$

variable for auxiliary lemmas

$\exists\, r1$: `rule.`

$\exists\, t1$: `term.`

variable for terms

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

variable for term occurrences

$r1$ `is_rule_of` $to1$

$\wedge$

conjunction

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

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

$\exists\, n$: `number.`

variable for natural numbers

universal quantifier

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

$\wedge$

$t2$ `is_nth_induction_term` n

Example Assertion in LiFtEr (in Abstract Syntax)

implication existential quantifier

$\exists\ r1$: rule. True

$\rightarrow$ variable for auxiliary lemmas

$\exists\ r1$: rule.

$\exists\ t1$: term. variable for terms

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

$r1$ is_rule_of $to1$ variable for term occurrences

$\wedge$ conjunction

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

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

$\exists\ n$: number. variable for natural numbers

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

universal quantifier

$\wedge$

$t2$ 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
```

$\exists\ r1$: rule. True

$\rightarrow$

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

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

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

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

r1

(r1 = itrev.induct)

$\exists\ r1$: rule. True

$\rightarrow$

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

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

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

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

r1

(r1 = itrev.induct)

$\exists\ r1$: rule. True

$\rightarrow$

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

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

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

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

r1

(r1 = itrev.induct)

$\exists\ r1$: rule. True

$\rightarrow$

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

```
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
```

t1 r1

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

$\exists\ r1$: rule. True

$\rightarrow$

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

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

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

fun itrev :: "'a list ⇒ 'a list ⇒ 'a list" wher
 "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

t1 r1

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

$\exists\ r1$: rule. True

$\rightarrow$

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

```
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

t1 r1

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

$\exists\ r1$: rule. True

$\rightarrow$

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

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

```
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 t1 r1

$\exists$ $r1$: rule. True

$\rightarrow$

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

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

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

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

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

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

to1

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

t1 r1

$\exists\ r1$: rule. True

$\rightarrow$

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

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

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

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

to1

t1 r1

$\exists\, r1$: rule. True

$\rightarrow$

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

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

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

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

to1

t2

t1

r1

$\exists\ r1$: rule. True

$\rightarrow$

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

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

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

(t2 = 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
```

to1 to2 t2 t1 r1

$\exists\ r1$: rule. True

$\rightarrow$

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

(r1 = itrev.induct)
(t1 = itrev)
(to1 = itrev)
True! r1 (= itrev.induct) is a lemma about to1 (= itrev).
(t2 = xs and ys)
(to2 = 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
```

to1 to2 t2 t1 r1

$\exists\ r1$: rule. True

$\rightarrow$

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

(r1 = itrev.induct)
(t1 = itrev)
(to1 = itrev)
True! r1 (= itrev.induct) is a lemma about to1 (= itrev).
(t2 = xs and ys)
(to2 = 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)"
```

to1 to2

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

t2 t1 r1

$\exists\ r1$: rule. True

$\rightarrow$

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

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

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

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

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

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

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

to1 first to2

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

t2 first t1 r1

$\exists\, r1$: rule. True

$\rightarrow$

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

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

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

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

True for 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

t2 first second t1 r1

$\exists\ r1$: rule. True

$\rightarrow$

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

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

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

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

True for xs (n = 1)!

True for ys (n = 2)!

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

the same LiFtEr assertion

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

$\rightarrow$

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

new types ->

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

the same LiFtEr assertion

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

new types ->

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

new constants ->

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

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

the same LiFtEr assertion

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

new types ->

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

new constants ->

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

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

the same LiFtEr assertion

new lemma ->

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

a model proof ->

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

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

new types ->

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

new constants ->

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

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

new lemma ->

a model proof ->

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

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

$$\rightarrow$$

$$\begin{array}{l}
\exists\ r1 : \texttt{rule}. \\
\quad \exists\ t1 : \texttt{term}. \\
\quad\quad \exists\ to1 : \texttt{term_occurrence} \in t1 : \texttt{term}. \\
\quad\quad\quad r1\ \texttt{is_rule_of}\ to1 \\
\quad\quad\quad \wedge \\
\quad\quad\quad \forall\ t2 : \texttt{term} \in \texttt{induction_term}. \\
\quad\quad\quad\quad \exists\ to2 : \texttt{term_occurrence} \in t2 : \texttt{term}. \\
\quad\quad\quad\quad\quad \exists\ n : \texttt{number}. \\
\quad\quad\quad\quad\quad\quad \texttt{is_nth_argument_of}\ (to2,\ n,\ to1) \\
\quad\quad\quad\quad\quad\quad \wedge \\
\quad\quad\quad\quad\quad\quad t2\ \texttt{is_nth_induction_term}\ n
\end{array}$$

new types ->

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

new constants ->

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

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

new lemma ->

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

a model proof ->

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

r1

$\exists\ r1$: rule. True

$\rightarrow$

$\exists\ r1$: rule.

(r1 = exec.induct)

$\exists\ t1$: term.

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

$r1$ is_rule_of $to1$

$\wedge$

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

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

$\exists\ n$: number.

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

$\wedge$

$t2$ is_nth_induction_term n

new types ->

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

new constants ->

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

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

new lemma ->

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

a model proof ->

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

r1

(r1 = exec.induct)

$\exists$ $r1$: rule. True

$\rightarrow$

$\exists$ $r1$: rule.
  $\exists$ $t1$: term.
    $\exists$ $to1$: term_occurrence $\in$ $t1$: term.
      $r1$ is_rule_of $to1$
      $\land$
      $\forall$ $t2$: term $\in$ induction_term.
        $\exists$ $to2$: term_occurrence $\in$ $t2$: term.
          $\exists$ n : number.
            is_nth_argument_of ($to2$, n, $to1$)
            $\land$
            $t2$ 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 -> a model proof ->

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

r1

(r1 = exec.induct)

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

$\rightarrow$

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

new types ->

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

new constants ->

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

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

new lemma ->

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

a model proof ->

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

t1 r1

(r1 = exec.induct)

(t1 = exec)

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

$$\rightarrow$$

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

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)"
```

to1

new lemma ->

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

a model proof ->

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

t1 r1

$\exists\ r1$: rule. True

$\rightarrow$

$\exists\ r1$: rule.

$\exists\ t1$: term.

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

$r1$ is_rule_of $to1$

$\wedge$

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

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

$\exists\ n$: number.

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

$\wedge$

$t2$ is_nth_induction_term n

(r1 = exec.induct)

(t1 = exec)

(to1 = exec)

new types ->

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

new constants ->

```
fun 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 ->

to1

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

a model proof ->

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

t1 r1

$\exists\ r1$: rule. True

$\to$

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

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

new types ->

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

new constants ->

```
fun 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)"
```

to1

new lemma ->

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

a model proof ->

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

t1 r1

$\exists$ $r1$: rule. True

$\rightarrow$

$\exists$ $r1$: rule.

$\exists$ $t1$: term.

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

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

$\wedge$

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

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

$\exists$ n : number.

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

$\wedge$

$t2$ is_nth_induction_term n

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

new types ->

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

new constants ->

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

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

to1

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

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

t1 r1

$\exists\ r1$: rule. True

$\rightarrow$

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

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

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

new types ->

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

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

new constants ->

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

to1

a model proof ->

t1 r1

$\exists$ $r1$: rule. True

$\rightarrow$

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

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

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

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)"
```

t2 to1

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

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

`apply auto done`

t1 r1

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

$\rightarrow$

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

$\exists$ $t1$: **term.**

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

$r1$ **is_rule_of** $to1$

$\wedge$

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

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

$\exists$ n : **number.**

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

$\wedge$

$t2$ **is_nth_induction_term** n

(r1 = exec.induct)

(t1 = exec)

(to1 = exec)

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

(t2 = is1, s, and stk)

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)"
```

t2 to1

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

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

`apply auto done`

to2

t1 r1

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

$\rightarrow$

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

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

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

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

new types ->

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

new constants ->

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

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

new lemma ->

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

a model proof ->

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

t2, to1, to2, t1, r1

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

$\rightarrow$

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

$\quad \exists\ t1 : \texttt{term}.$

$\quad\quad \exists\ to1 : \texttt{term\_occurrence} \in\ t1 : \texttt{term}.$

$\quad\quad\quad r1\ \texttt{is\_rule\_of}\ to1$

$\quad\quad \wedge$

$\quad\quad\quad \forall\ t2 : \texttt{term} \in \texttt{induction\_term}.$

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

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

$\quad\quad\quad\quad\quad\quad \texttt{is\_nth\_argument\_of}\ (to2,\ n,\ to1)$

$\quad\quad\quad\quad\quad \wedge$

$\quad\quad\quad\quad\quad\quad t2\ \texttt{is\_nth\_induction\_term}\ n$

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

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

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

new types ->

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

new constants ->

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

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

new lemma ->

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

t2 to1 to2

a model proof ->

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

t1 r1

$\exists$ $r1$: rule. True

$\rightarrow$

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

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

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

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

new types ->

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

new constants ->

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

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

first

t2 to1

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

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

`apply auto done`

to2

t1 r1

first

$\exists$ $r1$: rule. True

$\rightarrow$

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

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

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

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

True for is1 (n -> 1)!

new types ->

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

new constants ->

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

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

first second t2 to1

new lemma ->

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

a model proof ->

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

to2 first second t1 r1

∃ $r1$: rule. True

$\rightarrow$

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

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

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

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

True for is1 (n -> 1)!
True for ys (n -> 2)!

new types ->

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

new constants ->

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

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

new lemma ->

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

a model proof ->

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

first   second   third   t2   to1   first   second   third   to2   t1   r1

$\exists\ r1$: rule. True

$\rightarrow$

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

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

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

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

True for is1 (n -> 1)!
True for ys (n -> 2)!
True for stk (n -> 3)!

Big Picture

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

ML

<- pros: good at ambiguity (heuristics)

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

LiFtEr 1 LiFtEr 2 LiFtEr 3

<- pros: good at rigorous abstraction

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

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

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

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

<- small dataset about different domains

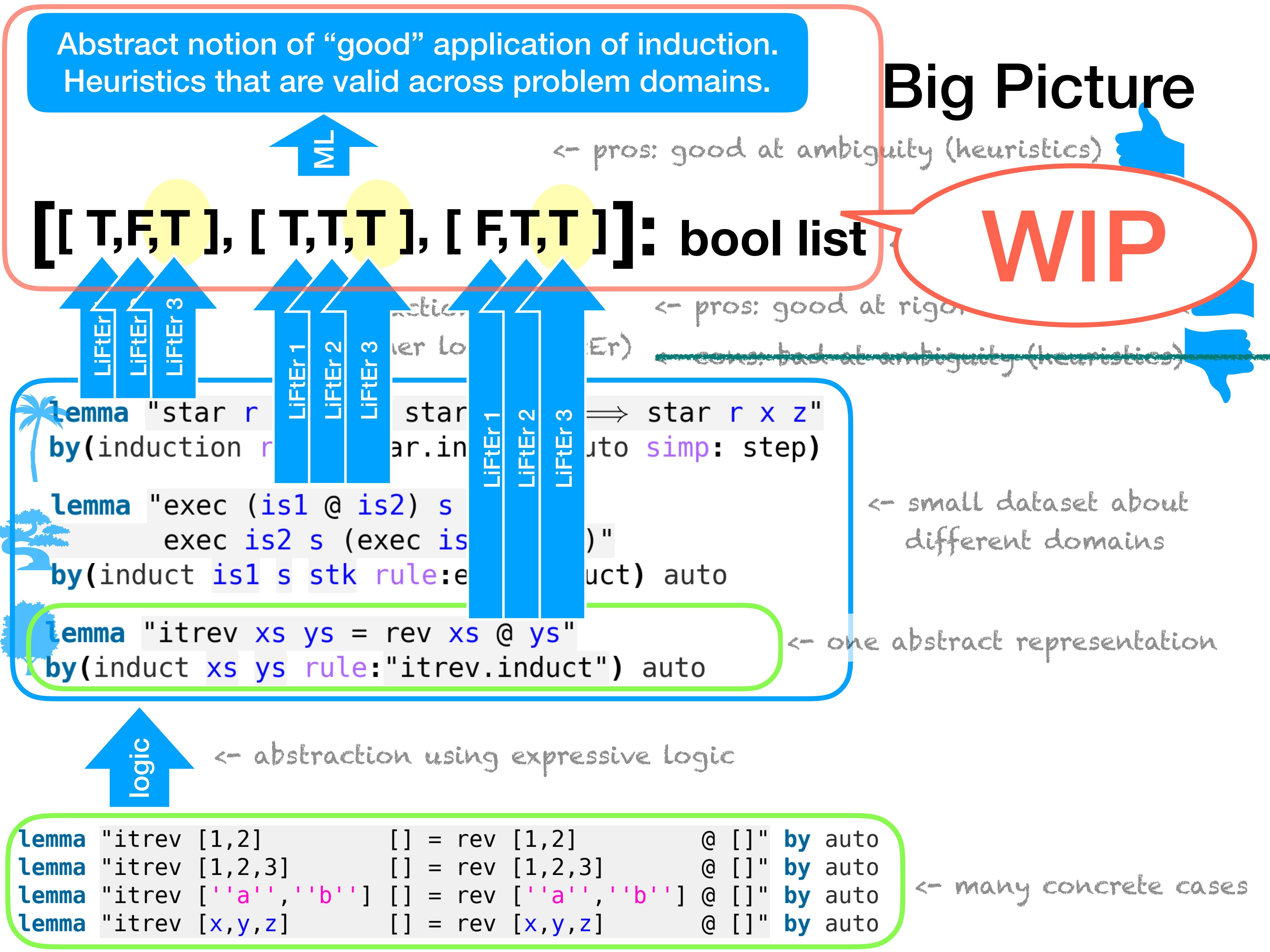
<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases


Abstract notion of “good” application of induction.
Heuristics that are valid across problem domains.
Big Picture
ML
<- pros: good at ambiguity (heuristics)
[[T,F,T], [T,T,T], [F,T,T]]: bool list
WIP
~~cons: bad at ambiguity (heuristics)~~
LiFtEr 1
LiFtEr 2
LiFtEr 3
lemma "itrev xs ys = rev xs @ ys"
by(induct xs ys rule:"itrev.induct") auto
<- small dataset about different domains
<- one abstract representation
logic
<- abstraction using expressive logic
lemma "itrev [1,2] [] = rev [1,2] @ []" by auto
lemma "itrev [1,2,3] [] = rev [1,2,3] @ []" by auto
lemma "itrev [''a'',''b''] [] = rev [''a'',''b''] @ []" by auto
lemma "itrev [x,y,z] [] = rev [x,y,z] @ []" by auto
<- many concrete cases

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

Registration is now closed.

http://aitp-conference.org/2019/

Background

Large-scale semantic processing and strong computer assistance of mathematics and science is our inevitable future. New combinations of AI and reasoning methods and tools deployed over large mathematical and scientific corpora will be instrumental to this task. The AITP conference is the forum for discussing how to get there as soon as possible, and the force driving the progress towards that.

Topics

- AI and big-data methods in theorem proving and mathematics
- Collaboration between automated and interactive theorem proving
- Common-sense reasoning and reasoning in science
- Alignment and joint processing of formal, semi-formal, and informal libraries
- Methods for large-scale computer understanding of mathematics and science
- Combinations of linguistic/learning-based and semantic/reasoning methods

Feature extractor?

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
```

Feature extractor?

```
fun sep::"'a ⇒ 'a list ⇒ 'a list" where
  "sep a [] = []" |
  "sep a [x] = [x]" |
  "sep a (x#y#zs) = x # a # sep a (y#zs)"
```

automatically proves and saves many auxiliary lemmas in the context

sep.simps, sep.induct, sep.elims, etc.

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
```

Feature extractor?

```
fun sep::"'a ⇒ 'a list ⇒ 'a list" where
  "sep a [] = []" |
  "sep a [x] = [x]" |
  "sep a (x#y#zs) = x # a # sep a (y#zs)"
```

automatically proves and saves many auxiliary lemmas in the context

sep.simps, sep.induct, sep.elims, etc.

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
```

assertion 27: if the outermost constant is the HOL equality?

Feature extractor?

```
fun sep::"'a ⇒ 'a list ⇒ 'a list" where
  "sep a [] = []" |
  "sep a [x] = [x]" |
  "sep a (x#y#zs) = x # a # sep a (y#zs)"
```

automatically proves and saves many auxiliary lemmas in the context

sep.simps, sep.induct, sep.elims, etc.

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
```

assertion 27: if the outermost constant is the HOL equality? ✓

Feature extractor?

```
fun sep::"'a ⇒ 'a list ⇒ 'a list" where
  "sep a [] = []" |
  "sep a [x] = [x]" |
  "sep a (x#y#zs) = x # a # sep a (y#zs)"
```

automatically proves and saves many auxiliary lemmas in the context

sep.simps, sep.induct, sep.elims, etc.

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
```

assertion 27: if the outermost constant is the HOL equality?

assertion 32: if the outermost constant is the HOL existential quantifier?

Feature extractor?

```
fun sep::"'a ⇒ 'a list ⇒ 'a list" where
  "sep a [] = []" |
  "sep a [x] = [x]" |
  "sep a (x#y#zs) = x # a # sep a (y#zs)"
```

automatically proves and saves many auxiliary lemmas in the context

sep.simps, sep.induct, sep.elims, etc.

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
```

assertion 27: if the outermost constant is the HOL equality?

assertion 32: if the outermost constant is the HOL existential quantifier?

Feature extractor?

```
fun sep::"'a ⇒ 'a list ⇒ 'a list" where
  "sep a [] = []" |
  "sep a [x] = [x]" |
  "sep a (x#y#zs) = x # a # sep a (y#zs)"
```

automatically proves and saves many auxiliary lemmas in the context

sep.simps, sep.induct, sep.elims, etc.

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
```

assertion 27: if the outermost constant is the HOL equality? ✓

assertion 32: if the outermost constant is the HOL existential quantifier? ✗

assertion 93: if the goal has a term of type "real"?

Feature extractor?

```
fun sep::"'a ⇒ 'a list ⇒ 'a list" where
  "sep a [] = []" |
  "sep a [x] = [x]" |
  "sep a (x#y#zs) = x # a # sep a (y#zs)"
```

automatically proves and saves many auxiliary lemmas in the context

sep.simps, sep.induct, sep.elims, etc.

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
```

assertion 27: if the outermost constant is the HOL equality? ✓

assertion 32: if the outermost constant is the HOL existential quantifier? ✗

assertion 93: if the goal has a term of type "real"? ✗

Feature extractor?

```
fun sep::"'a ⇒ 'a list ⇒ 'a list" where
  "sep a [] = []" |
  "sep a [x] = [x]" |
  "sep a (x#y#zs) = x # a # sep a (y#zs)"
```

automatically proves and saves many auxiliary lemmas in the context

sep.simps, sep.induct, sep.elims, etc.

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
```

assertion 27: if the outermost constant is the HOL equality? ✓

assertion 32: if the outermost constant is the HOL existential quantifier? ✗

assertion 93: if the goal has a term of type “real”? ✗

assertion 10: the context has a related recursive simplification rule?

Feature extractor?

```
fun sep::"'a ⇒ 'a list ⇒ 'a list" where
  "sep a [] = []" |
  "sep a [x] = [x]" |
  "sep a (x#y#zs) = x # a # sep a (y#zs)"
```

automatically proves and saves many auxiliary lemmas in the context

sep.simps, sep.induct, sep.elims, etc.

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
```

assertion 27: if the outermost constant is the HOL equality? ✓

assertion 32: if the outermost constant is the HOL existential quantifier? ✗

assertion 93: if the goal has a term of type “real”? ✗

assertion 10: the context has a related recursive simplification rule?

Feature extractor?

```
fun sep::"'a ⇒ 'a list ⇒ 'a list" where
  "sep a [] = []" |
  "sep a [x] = [x]" |
  "sep a (x#y#zs) = x # a # sep a (y#zs)"
```

automatically proves and saves many auxiliary lemmas in the context

sep.simps, sep.induct, sep.elims, etc.

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
```

assertion 27: if the outermost constant is the HOL equality? ✓

assertion 32: if the outermost constant is the HOL existential quantifier? ✗

assertion 93: if the goal has a term of type “real”? ✗

assertion 10: the context has a related recursive simplification rule? ✓

Feature extractor?

```
fun sep::"'a ⇒ 'a list ⇒ 'a list" where
  "sep a [] = []" |
  "sep a [x] = [x]" |
  "sep a (x#y#zs) = x # a # sep a (y#zs)"
```

automatically proves and saves many auxiliary lemmas in the context

sep.simps, sep.induct, sep.elims, etc.

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
```

assertion 27: if the outermost constant is the HOL equality? ✓

assertion 32: if the outermost constant is the HOL existential quantifier? ✗

assertion 93: if the goal has a term of type “real”? ✗

assertion 10: the context has a related recursive simplification rule? ✓

assertion 58: the context has a constant defined with the “fun” keyword?

Feature extractor?

```
fun sep::"'a ⇒ 'a list ⇒ 'a list" where
  "sep a [] = []" |
  "sep a [x] = [x]" |
  "sep a (x#y#zs) = x # a # sep a (y#zs)"
```

automatically proves and saves many auxiliary lemmas in the context

sep.simps, sep.induct, sep.elims, etc.

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
```

assertion 27: if the outermost constant is the HOL equality? ✓

assertion 32: if the outermost constant is the HOL existential quantifier? ✗

assertion 93: if the goal has a term of type "real"? ✗

assertion 10: the context has a related recursive simplification rule? ✓

assertion 58: the context has a constant defined with the "fun" keyword? ✓

Feature extractor?

```
fun sep::"'a ⇒ 'a list ⇒ 'a list" where
  "sep a [] = []" |
  "sep a [x] = [x]" |
  "sep a (x#y#zs) = x # a # sep a (y#zs)"
```

automatically proves and saves many auxiliary lemmas in the context

sep.simps, sep.induct, sep.elims, etc.

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
```

assertion 27: if the outermost constant is the HOL equality?

assertion 32: if the outermost constant is the HOL existential quantifier?

assertion 93: if the goal has a term of type "real"?

assertion 10: the context has a related recursive simplification rule?

assertion 58: the context has a constant defined with the "fun" keyword?

resulting feature vector: [...,1,...,1,...0,...,1,...0,...]

10th 27th 32nd 58th 93rd