git clone https://github.com/data61/PSL

Domain-Specific Language to Encode Induction Heuristics

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

Yutaka Nagashima
University of Innsbruck
Czech Technical University

Yutaka Ng
yutakang

Block or report user

CVUT, CTU, CIIRC

CZECH INSTITUTE OF INFORMATICS ROBOTICS AND CYBERNETICS CTU IN PRAGUE

git clone https://github.com/data61/PSL

Domain-Specific Language to Encode Induction Heuristics

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

git clone https://github.com/data61/PSL

Domain-Specific Language to Encode Induction Heuristics

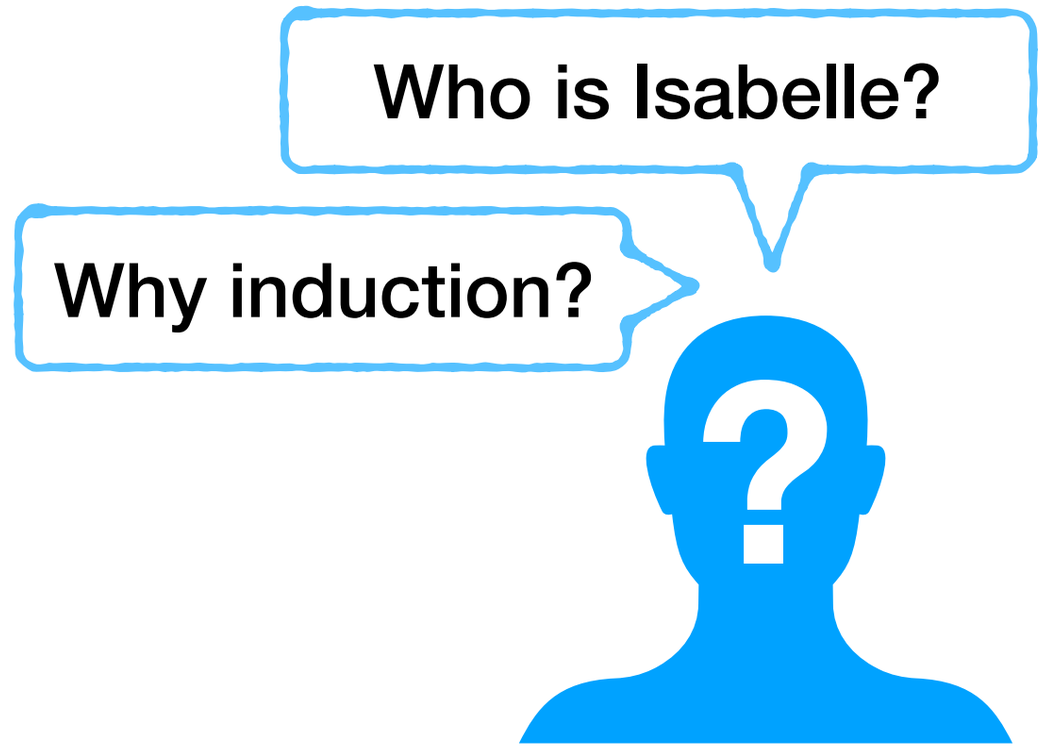
This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

Who is Isabelle?

git clone https://github.com/data61/PSL

Domain-Specific Language to Encode Induction Heuristics

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

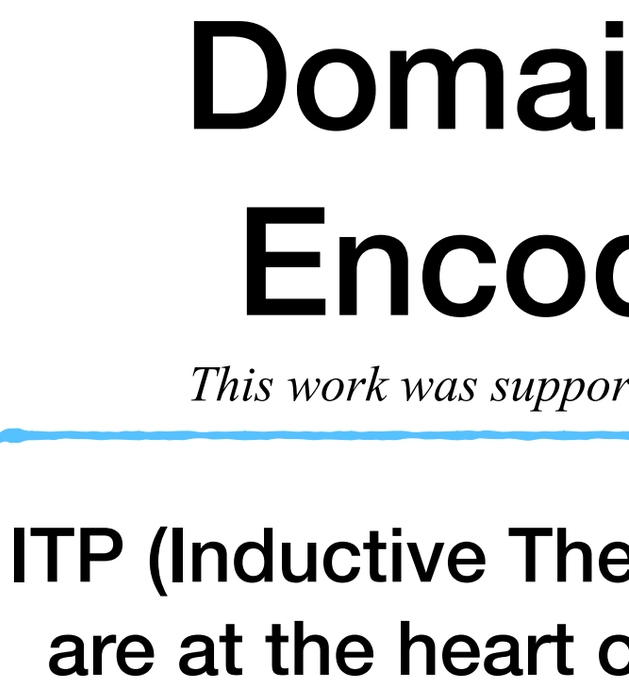


git clone https://github.com/data61/PSL

Domain-Specific Language to Encode Induction Heuristics

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

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



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

Who is Isabelle?

Why induction?

git clone https://github.com/data61/PSL

Domain-Specific Language to Encode Induction Heuristics

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

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

Who is Isabelle?

Why induction?

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

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

git clone https://github.com/data61/PSL

Domain-Specific Language to Encode Induction Heuristics

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

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

Who is Isabelle?

Why induction?

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

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

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

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

git clone https://github.com/data61/PSL

Domain-Specific Language to Encode Induction Heuristics

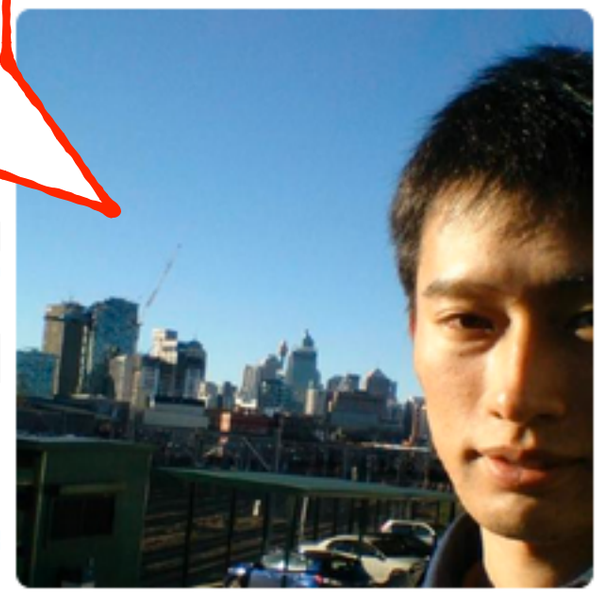
This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

ITP (Inductive Theorem Pro
are at the heart of many ve
reasoning tasks in computer science.

Challenge accepted!

?

Why in



we are convinced that subst
progress in ITP will take ti

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

spectacular breakt **Yutaka Ng**
unrealistic, in view of yutakang
problems and the inhe Block or report user
inductive theorer CVUT, CTU, CIIRC

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

git clone https://github.com/data61/PSL

Domain-Specific Language to Encode Induction Heuristics

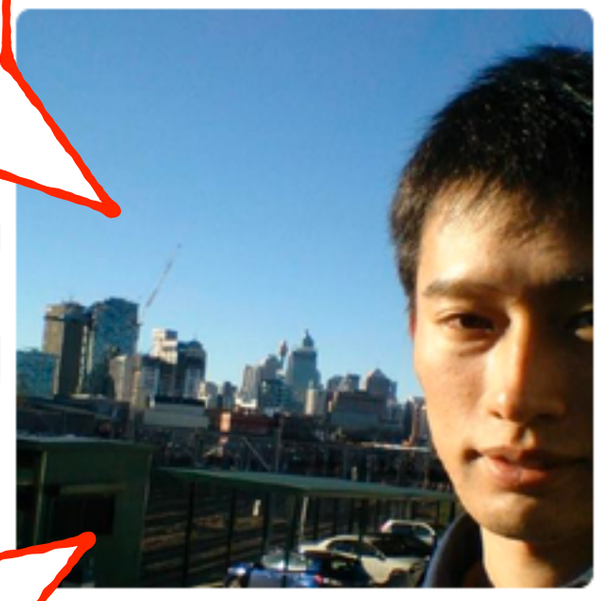
This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

ITP (Inductive Theorem Pro
are at the heart of many ve
reasoning tasks in computer science.

Challenge accepted!

?

Why in



we are convinced that subst
progress in ITP will take ti

Let AI solve it!

Yutaka Ng
yutakang

Block or report user

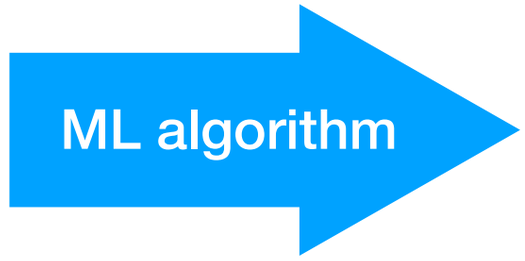
CVUT, CTU, CIIRC

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

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

git clone https://github.com/data61/PSL

Introduction to Machine Learning in 10 seconds



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

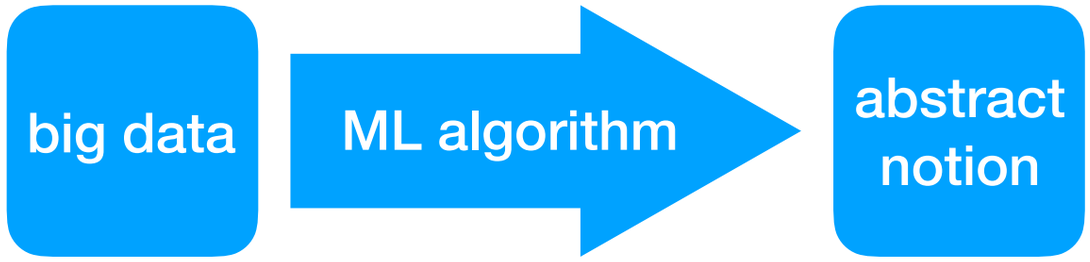
git clone https://github.com/data61/PSL

Introduction to Machine Learning in 10 seconds

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

git clone https://github.com/data61/PSL

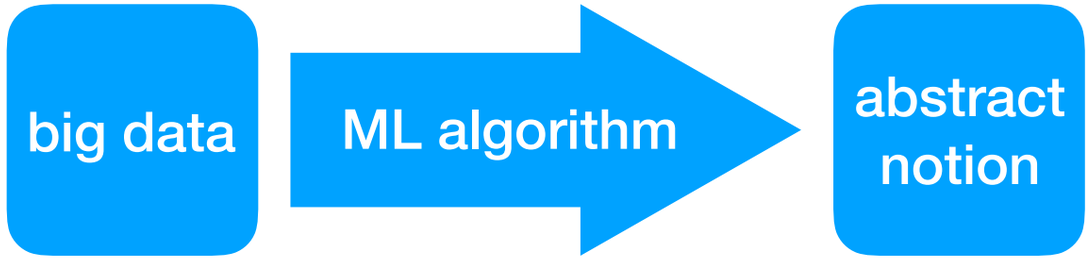
Introduction to Machine Learning in 10 seconds

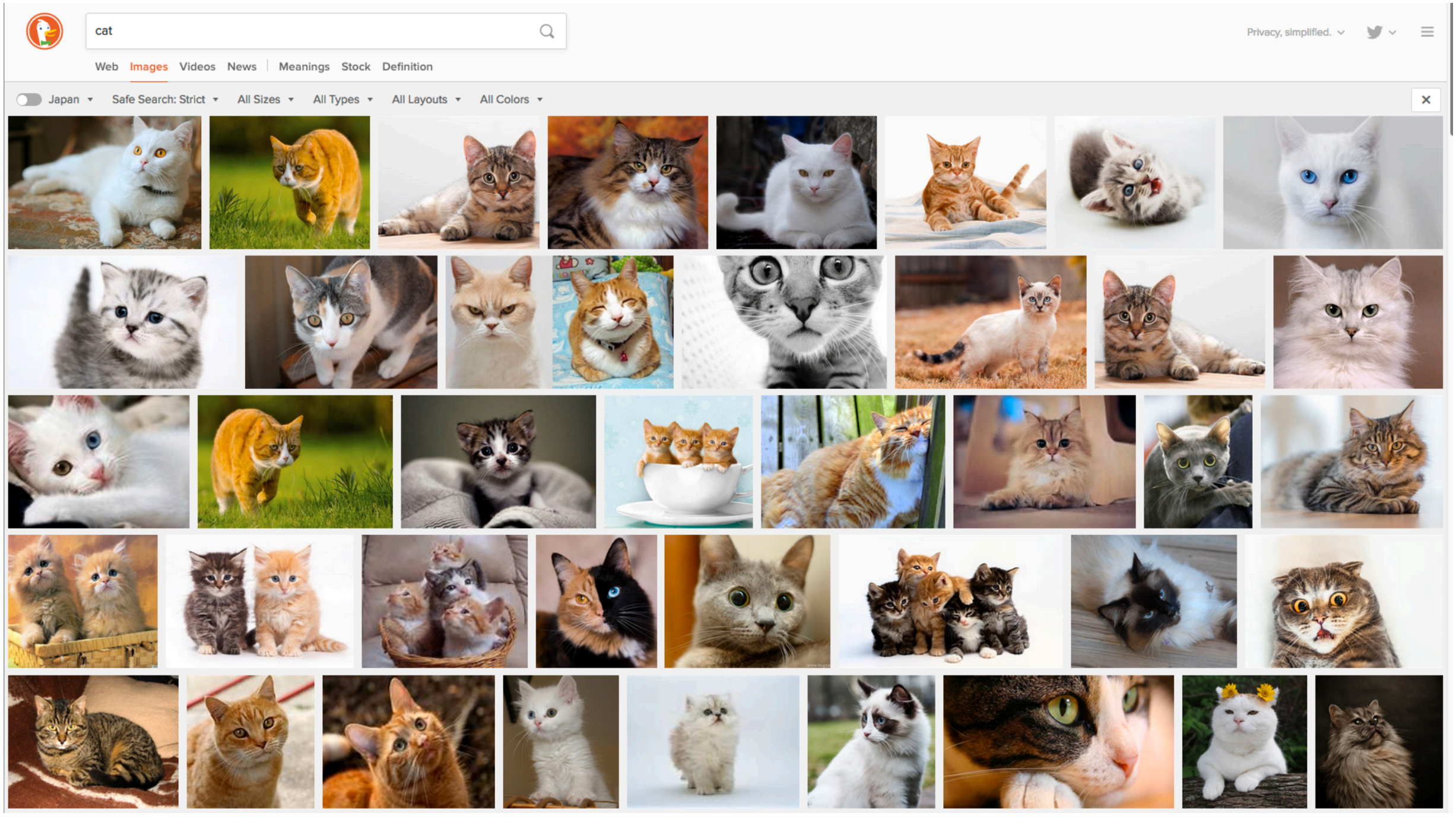


https://duckduckgo.com/?q=cat&t=ffab&iar=images&iax=images&ia=images

git clone https://github.com/data61/PSL

Introduction to Machine Learning in 10 seconds

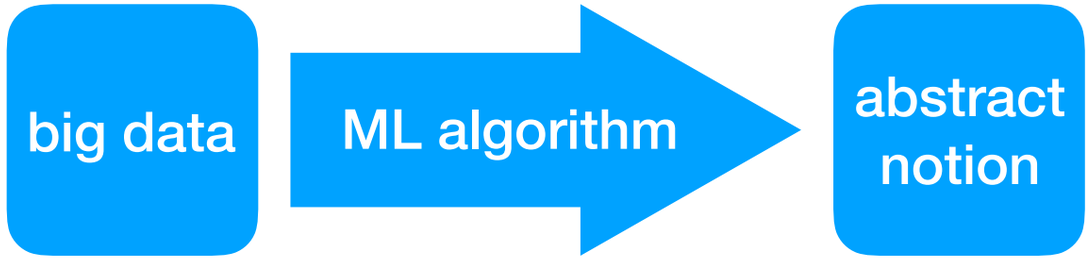


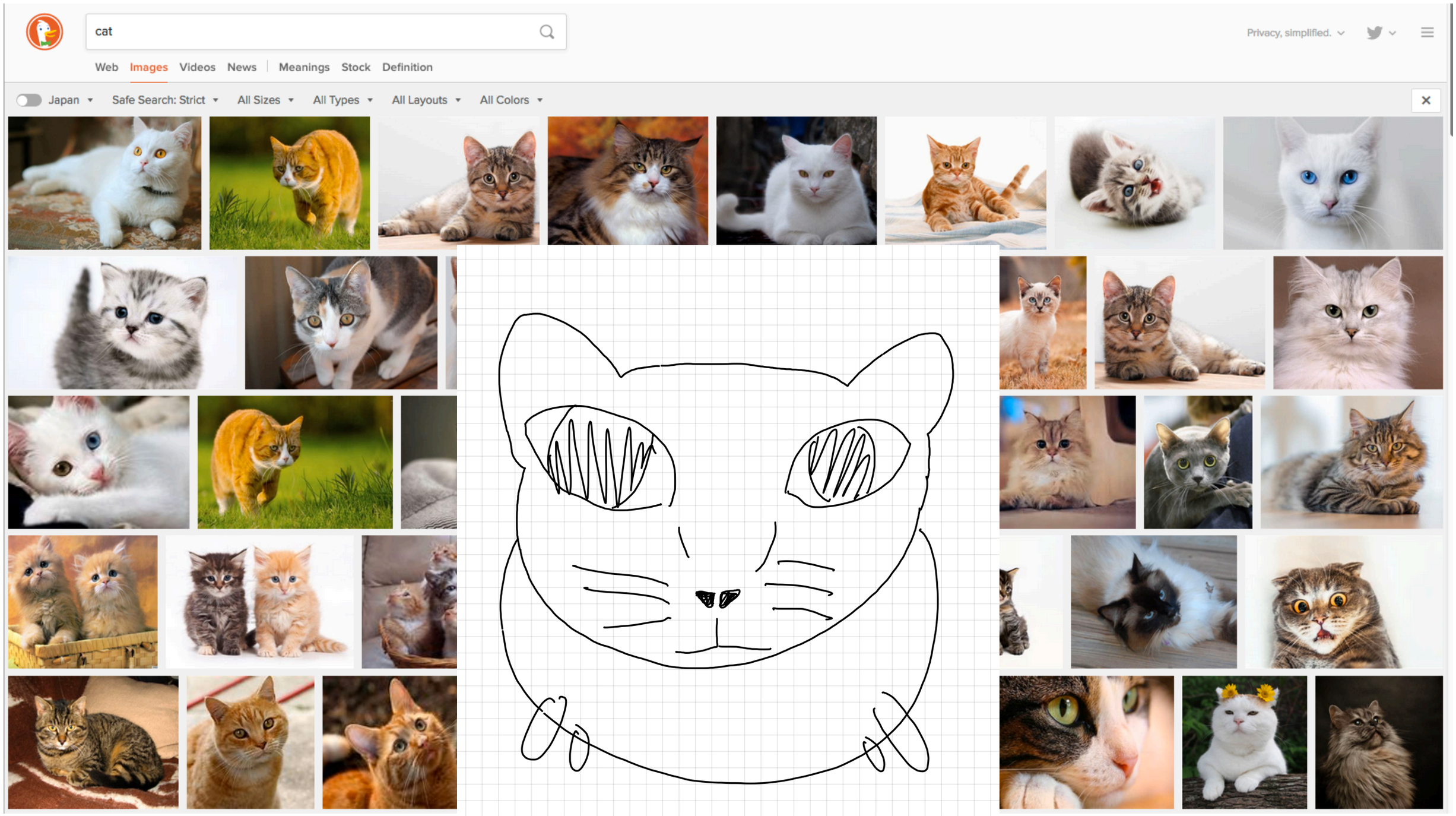


https://duckduckgo.com/?q=cat&t=ffab&iar=images&iax=images&ia=images

git clone https://github.com/data61/PSL

Introduction to Machine Learning in 10 seconds





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

git clone https://github.com/data61/PSL

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
```

git clone https://github.com/data61/PSL

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

git clone https://github.com/data61/PSL

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

git clone https://github.com/data61/PSL

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

git clone https://github.com/data61/PSL

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

git clone https://github.com/data61/PSL

ML for Inductive Theorem Proving the BAD

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

<- one abstract representation

ML

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

<- many concrete cases

git clone https://github.com/data61/PSL

ML for Inductive Theorem Proving the BAD

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

<- one abstract representation

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

ML

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

<- many concrete cases

git clone https://github.com/data61/PSL

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

git clone https://github.com/data61/PSL

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
```

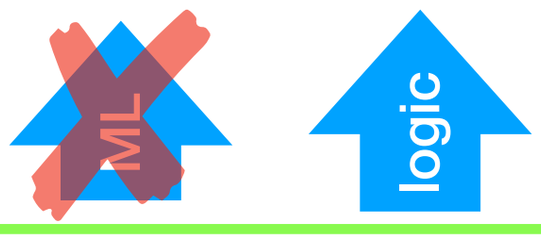
<- many concrete cases

git clone https://github.com/data61/PSL

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

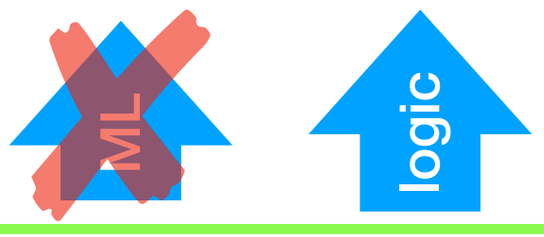
git clone https://github.com/data61/PSL

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

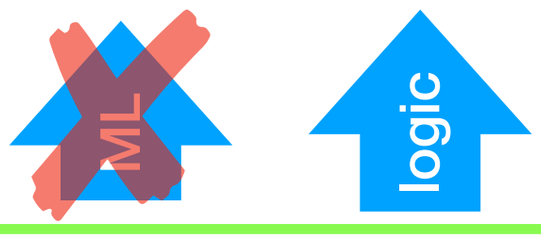
git clone https://github.com/data61/PSL

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

git clone https://github.com/data61/PSL

ML for Inductive Theorem Proving the BAD

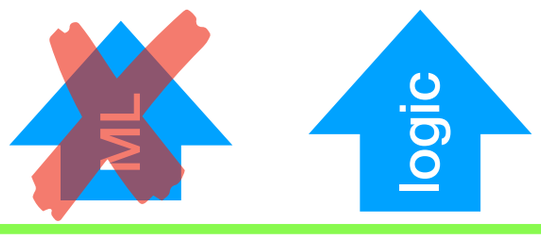
polymorphism

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

git clone https://github.com/data61/PSL

ML for Inductive Theorem Proving
the BAD

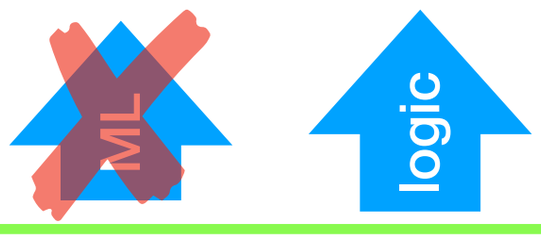
Higher-Order functions

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

git clone https://github.com/data61/PSL

ML for Inductive Theorem Proving the BAD

Higher-Order functions

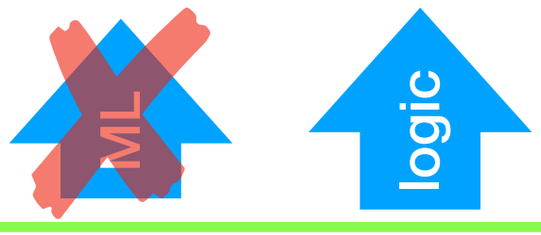
polymorphism **type class**

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

git clone https://github.com/data61/PSL

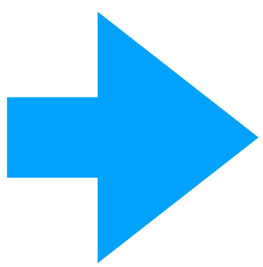
ML for Inductive Theorem Proving the BAD

Higher-Order functions

polymorphism

type class

universal quantifier

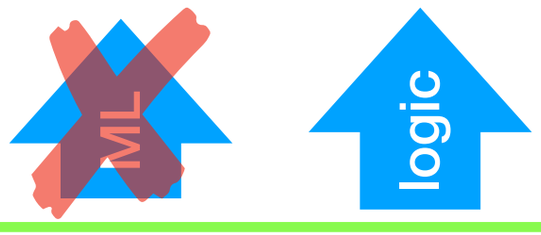


concise formula that can cover many concrete cases

lambda abstraction

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

git clone https://github.com/data61/PSL

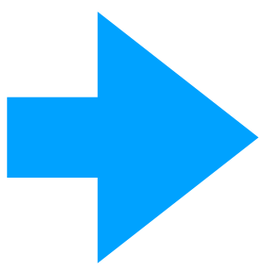
ML for Inductive Theorem Proving the BAD

Higher-Order functions

polymorphism

type class

universal quantifier



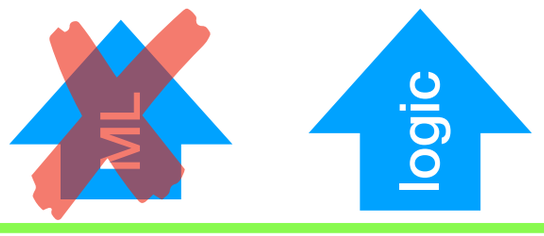
concise formula that can cover many concrete cases

different proof for general case

lambda abstraction

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

git clone https://github.com/data61/PSL

ML for Inductive Theorem Proving the BAD

Higher-Order functions

polymorphism

type class

universal quantifier

lambda abstraction

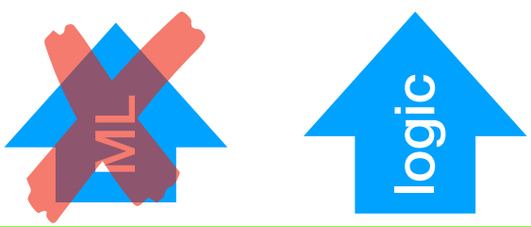
concise formula that can cover many concrete cases

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

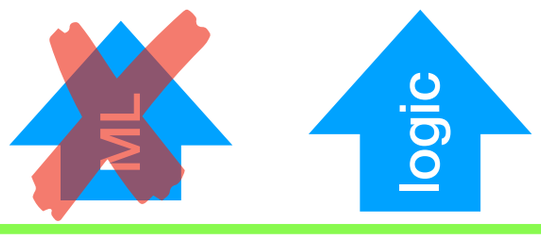
```
lemma "itrev [1,2]         [] = rev [1,2]         @ []" by auto
lemma "itrev [1,2,3]       [] = rev [1,2,3]       @ []" by auto
lemma "itrev [''a'',''b''] [] = rev [''a'',''b''] @ []" by auto
lemma "itrev [x,y,z]       [] = rev [x,y,z]       @ []" by auto
```

<- many concrete cases

git clone https://github.com/data61/PSL

```
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
```

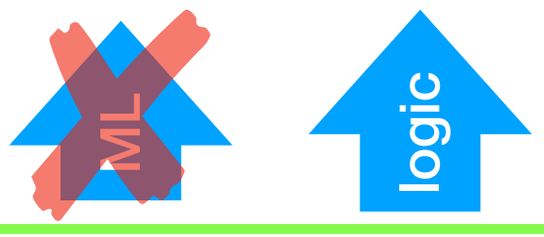
<- many concrete cases

git clone https://github.com/data61/PSL

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

git clone https://github.com/data61/PSL

Grand Challenge: Abstract Abstraction

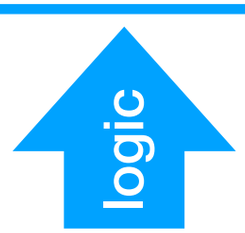
```
lemma "star r x y ⟹ star r y z ⟹ star r x z"
by(induction rule: star.induct)(auto simp: step)

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

<- small dataset about different domains

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

git clone https://github.com/data61/PSL

Grand Challenge: Abstract Abstraction

ML

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

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

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

<- small dataset about different domains

<- one abstract representation

~~ML~~ logic

<- abstraction using expressive logic

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

<- many concrete cases

git clone https://github.com/data61/PSL

Grand Challenge: Abstract Abstraction

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

<- even more abstract

ML

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

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

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

<- small dataset about different domains

<- one abstract representation

~~ML~~ logic

<- abstraction using expressive logic

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

<- many concrete cases

git clone https://github.com/data61/PSL

Grand Challenge: Abstract Abstraction

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

<- even more abstract

<- pros: good at ambiguity (heuristics)

ML

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

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

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

<- small dataset about different domains

<- one abstract representation

~~ML~~

logic

<- abstraction using expressive logic

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

<- many concrete cases

git clone https://github.com/data61/PSL

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

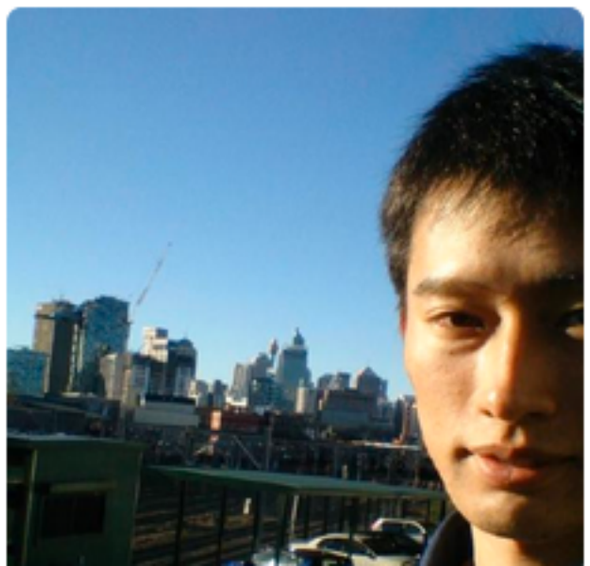
https://cbmm.mit.edu/video/power-self-learning-systems

CENTER FOR
Brains
Minds+
Machines

March 20, 2019

The Power of

Self



Many key challenges remain

Unsupervised Learning

Memory and one-shot learning

Imagination-based Planning with Generative Models

Learning Abstract Concepts

Transfer Learning

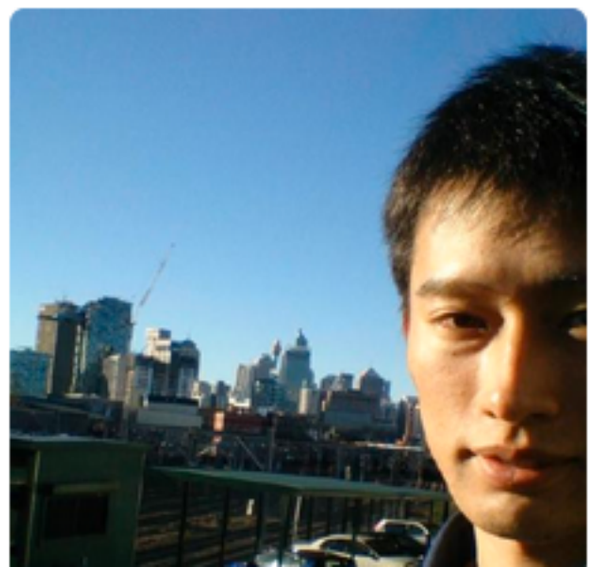
Language understanding

CENTER FOR
Brains
Minds+
Machines

March 20, 2019

The Power of

Self



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

We're working on.

Many key challenges remain

Unsupervised Learning

Memory and one-shot learning

Imagination-based Planning with Generative Models

Learning Abstract Concepts

Transfer Learning

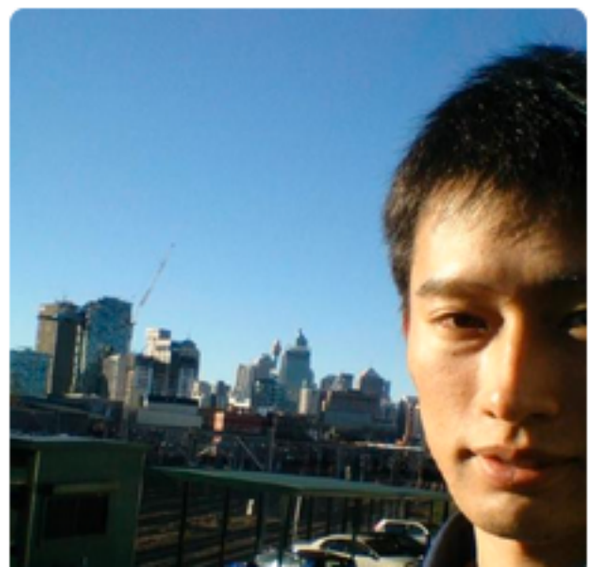
Language understanding

CENTER FOR
Brains
Minds+
Machines

March 20, 2019

The Power of

Self



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

We're working on.

CENTER FOR
Brains
Minds+
Machines

March 20, 2019

Many key challenges remain

Unsupervised Learning

Memory and one-shot learning

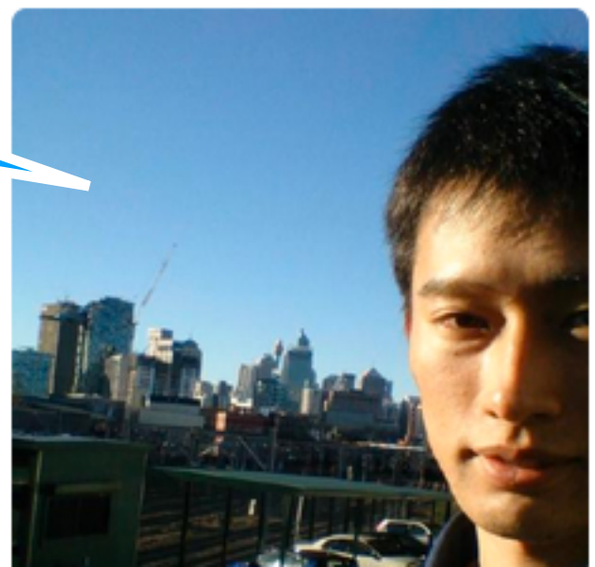
Imagination-based Planning with Generative Models

Learning Abstract Concepts

We already know a framework good for abstract concepts.

The Power of

Self



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

We're working on.

CENTER FOR
Brains
Minds+
Machines

March 20, 2019

Many key challenges remain

Unsupervised Learning

Memory and one-shot learning

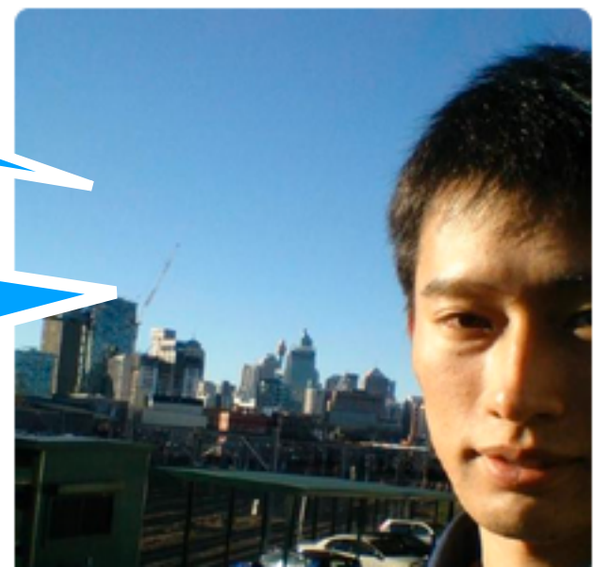
Imagination-based Planning with Generative Models

Learning Abstract Concepts

We already know a framework good for abstract concepts.

It is called logic!

The Power of Self



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

git clone https://github.com/data61/PSL

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

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

git clone https://github.com/data61/PSL

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

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

git clone https://github.com/data61/PSL

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

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

git clone https://github.com/data61/PSL

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

git clone https://github.com/data61/PSL

Logic about Proofs to Abstract Abstraction

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

<- even more abstract

abstraction using another logic (LiFtEr)

<- pros: good at rigorous abstraction

<- cons: bad at ambiguity (heuristics)

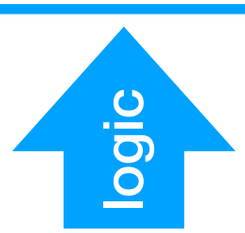
```
lemma "star r x y ⟹ star r y z ⟹ star r x z"
by(induction rule: star.induct)(auto simp: step)

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



<- 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://github.com/data61/PSL

Big Picture

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

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.

https://github.com/data61/PSL

Big Picture

abstraction using another logic (LiFtEr)

<- pros: good at rigorous abstraction

<- cons: bad at ambiguity (heuristics)

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

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

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

<- small dataset about different domains

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

https://github.com/data61/PSL

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.

https://github.com/data61/PSL

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.

https://github.com/data61/PSL

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.

https://github.com/data61/PSL

Big Picture

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

LiFtEr 2

LiFtEr

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.

https://github.com/data61/PSL

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.

https://github.com/data61/PSL

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.

https://github.com/data61/PSL

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

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

https://github.com/data61/PSL

Big Picture

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

LiFtEr 3 LiFtEr LiFtEr 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 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.

https://github.com/data61/PSL

Big Picture

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

LiFtEr 3 LiFtEr LiFtEr 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.

https://github.com/data61/PSL

Big Picture

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

LiFtEr 3 LiFtEr LiFtEr LiFtEr 1 LiFtEr 2

<- pros: good at rigorous abstraction

<- cons: bad at ambiguity (heuristics)

<- abstraction using other logic (LiFtEr)

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

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

<- 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.

https://github.com/data61/PSL

Big Picture

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

LiFtEr 3 LiFtEr LiFtEr LiFtEr 1 LiFtEr 2

<- pros: good at rigorous abstraction

<- cons: bad at ambiguity (heuristics)

<- abstraction using other logic (LiFtEr)

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

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

<- 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.

https://github.com/data61/PSL

Big Picture

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

LiFtEr LiFtEr LiFtEr 3 — LiFtEr 1 LiFtEr 2 LiFtEr 3

<- 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 r rule:star.induct)(auto simp: step)

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

<- 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.

https://github.com/data61/PSL

Big Picture

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

LiFtEr LiFtEr LiFtEr 3 — LiFtEr 1 LiFtEr 2 LiFtEr 3

<- pros: good at rigorous abstraction

<- cons: bad at ambiguity (heuristics)

<- abstraction using
<- logic (LiFtEr)

```
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
```

<- 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.

https://github.com/data61/PSL

Big Picture

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

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

<- 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    =
       exec is2 s (exec is    stk)"
by(induct is1 s stk rule:e     .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.

https://github.com/data61/PSL

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

<- 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                =
       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.

https://github.com/data61/PSL

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

<- 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.

https://github.com/data61/PSL

Big Picture

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

LiFtEr LiFtEr LiFtEr

<- pros: good at rigorous abstraction

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

LiFtEr 1 LiFtEr 2 LiFtEr 3

LiFtEr 1 LiFtEr 2

```
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
```

<- 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.

https://github.com/data61/PSL

Big Picture

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

LiFtEr LiFtEr LiFtEr

<- pros: good at rigorous abstraction

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

LiFtEr 1 LiFtEr 2 LiFtEr 3

LiFtEr 1 LiFtEr 2 LiFtEr 3

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

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

<- small dataset about different domains

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

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

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

https://github.com/data61/PSL

Big Picture

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

LiFtEr LiFtEr LiFtEr

<- 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
```

LiFtEr 1 LiFtEr 2 LiFtEr 3

<- 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://github.com/data61/PSL

Big Picture

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

ML

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

LiFtEr LiFtEr LiFtEr

LiFtEr 1 LiFtEr 2 LiFtEr 3

LiFtEr 1 LiFtEr 2 LiFtEr 3

<- pros: good at rigorous abstraction

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

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

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

<- small dataset about different domains

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

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

https://github.com/data61/PSL

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
```

<- 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://github.com/data61/PSL

Big Picture

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

ML

<- pros: good at ambiguity (heuristics)

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

LiFtEr LiFtEr LiFtEr 3

LiFtEr 1 LiFtEr 2 LiFtEr 3

<- pros: good at rigorous abstraction

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

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

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

<- small dataset about different domains

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

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

https://github.com/data61/PSL

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)

this talk

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

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

<- small dataset about different domains

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

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

Example Assertion in LiFtEr

```
val example_assertion =
  Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
          Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
         And
         (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

Example Assertion in LiFtEr

implication

```
val example_assertion =
 Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
          Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
         And
         (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

Example Assertion in LiFtEr

implication

conjunction

```
val example_assertion =
 Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
          Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
         And
         (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

Example Assertion in LiFtEr

implication

variable for auxiliary lemmas

```
val example_assertion =
 Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
          Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
         And
         (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

conjunction

Example Assertion in LiFtEr

implication

variable for auxiliary lemmas

variable for terms

conjunction

```
val example_assertion =
 Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
          Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
         And
         (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

Example Assertion in LiFtEr

implication

variable for auxiliary lemmas

variable for terms

variable for term occurrences

conjunction

```
val example_assertion =
 Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
          Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
         And
         (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

Example Assertion in LiFtEr

implication

variable for auxiliary lemmas

variable for terms

variable for term occurrences

conjunction

variable for natural numbers

```
val example_assertion =
 Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
          Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
         And
         (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

Example Assertion in LiFtEr

implication

variable for auxiliary lemmas

variable for terms

variable for term occurrences

conjunction

variable for natural numbers

universal quantifier

```
val example_assertion =
 Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
         Some_Numb (Numb 1,
          Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
         And
         (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

Example Assertion in LiFtEr

implication

existential quantifier

variable for auxiliary lemmas

```
val example_assertion =
 Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
          Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
         And
         (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

variable for terms

variable for term occurrences

conjunction

universal quantifier

variable for natural numbers

```
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
```

```
val example_assertion =
  Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
          Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
         And
         (Trm 2 Is_Nth_Ind Numb 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
```

Rule 1

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1))))))))));
```

(Rule 1 = itrev.induct)

```
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
```

Rule 1

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

(Rule 1 = itrev.induct)

```
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
```

Rule 1

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

(Rule 1 = itrev.induct)

```
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
```

Rule 1

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

(Rule 1 = itrev.induct)

```
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
```

Term 1

Rule 1

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,                          ( Rule 1      = itrev.induct )
  Some_Trm (Trm 1,                           ( Trm 1       = itrev )
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 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
```

Term 1

Rule 1

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

(Rule 1 = itrev.induct)

(Trm 1 = 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
```

Term_Occ 1

Term 1

Rule 1

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,                          ( Rule 1      = itrev.induct )
  Some_Trm (Trm 1,                           ( Trm 1       = itrev )
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,        ( Trm_Occ 1 = itrev )
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 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)"
```

Term_Occ 1

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

Term 1

Rule 1

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

(Rule 1 = itrev.induct)
(Trm 1 = itrev)
(Trm_Occ 1 = 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
```

Term_Occ 1

Term 1

Rule 1

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,                          ( Rule 1      = itrev.induct )
  Some_Trm (Trm 1,                           ( Trm 1       = itrev )
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,        ( Trm_Occ 1 = itrev )
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 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)"
```

Term_Occ 1

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

Term 1

Rule 1

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

(Rule 1 = itrev.induct)
(Trm 1 = itrev)
(Trm_Occ 1 = 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
```

Term_Occ 1

Term 1

Rule 1

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

(Rule 1 = itrev.induct)
(Trm 1 = itrev)
(Trm_Occ 1 = 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
```

Term_Occ 1

Term 2

Term 1

Rule 1

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,                                  ( Rule 1      = itrev.induct )
  Some_Trm (Trm 1,                                   ( Trm 1       = itrev )
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,                ( Trm_Occ 1 = itrev )
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,                                 ( Trm 2       = xs and ys )
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 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
```

Term_Occ 1 · Term_Occ 2 · Term 2 · Term 1 · Rule 1

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,                                  ( Rule 1      = itrev.induct )
  Some_Trm (Trm 1,                                   ( Trm 1       = itrev )
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,                ( Trm_Occ 1 = itrev )
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,                                 ( Trm 2       = xs and ys )
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,             ( Trm_Occ 2 = xs and ys )
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 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
```

Term_Occ 1 — Term_Occ 2 — Term 2 — Term 1 — Rule 1

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,                          ( Rule 1      = itrev.induct )
  Some_Trm (Trm 1,                           ( Trm 1       = itrev )
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,        ( Trm_Occ 1 = itrev )
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,                         ( Trm 2       = xs and ys )
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,     ( Trm_Occ 2 = xs and ys )
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 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
```

Term_Occ 1 · Term_Occ 2 · Term 2 · Term 1 · Rule 1

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,                          ( Rule 1      = itrev.induct )
  Some_Trm (Trm 1,                           ( Trm 1       = itrev )
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,        ( Trm_Occ 1 = itrev )
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,                                ( Trm 2       = xs and ys )
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,            ( Trm_Occ 2 = xs and ys )
       Some_Numb (Numb 1,
        Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
       And
       (Trm 2 Is_Nth_Ind Numb 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
```

Term_Occ 1 — first — Term_Occ 2

Term 2 — first — Term 1 — Rule 1

```
val example_assertion =
  Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

(Rule 1 = itrev.induct)
(Trm 1 = itrev)
(Trm_Occ 1 = itrev)
(Trm 2 = xs and ys)
(Trm_Occ 2 = 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
```

Term_Occ 1 — first — second — Term_Occ 2

Term 2 — second — Term 1 — Rule 1

first

(Rule 1 = itrev.induct)
(Trm 1 = itrev)
(Trm_Occ 1 = itrev)
(Trm 2 = xs and ys)
(Trm_Occ 2 = xs and ys)

```
val example_assertion =
  Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

```
val example_assertion =
  Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
          Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
         And
         (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

```
val example_assertion =
  Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And                                          <- the same LiFtEr assertion
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
          Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
         And
         (Trm 2 Is_Nth_Ind Numb 1))))))))));
```

new types ->

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

```
val example_assertion =
  Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
          Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
         And
         (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

<- the same LiFtEr assertion

new types ->

new constants ->

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

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

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

```
val example_assertion =
  Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
          Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
         And
         (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

<- the same LiFtEr assertion

new types ->

new constants ->

new lemma ->
a model proof ->

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

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

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

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

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

<- the same LiFtEr assertion

new types ->

new constants ->

new lemma ->
a model proof ->

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

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

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

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

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

new types ->

new constants ->

new lemma ->
a model proof ->

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

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

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

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

Rule 1

(Rule 1 = exec.induct)

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

new types ->

new constants ->

new lemma ->
a model proof ->

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

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

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

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

Rule 1

(Rule 1 = exec.induct)

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

new types ->

new constants ->

new lemma ->
a model proof ->

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

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

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

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

Rule 1

(Rule 1 = exec.induct)

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

new types ->

new constants ->

new lemma ->
a model proof ->

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

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

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

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

Rule 1

(Rule 1 = exec.induct)

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

new types ->

new constants ->

new lemma ->
a model proof ->

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

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

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

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

Term 1 Rule 1

(Rule 1 = exec.induct)

(Trm 1 = exec)

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

new types ->

new constants ->

new lemma ->
a model proof ->

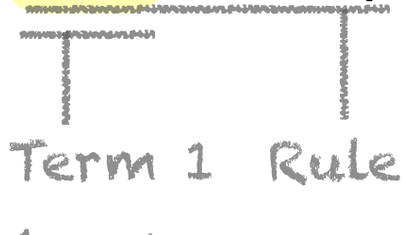
```
datatype     instr = LOADI val | LOAD vname | ADD
type_synonym stack = "val list"

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

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

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

```
val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```



Term 1 Rule 1

(Rule 1 = exec.induct)

(Trm 1 = exec)

new types ->

new constants ->

new lemma ->
a model proof ->

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

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

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

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

val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

Term_Occ 1

Term 1 Rule 1

(Rule 1   = exec.induct)

(Trm 1   = exec)

(Trm_Occ 1 = exec)

new types ->

new constants ->

new lemma ->
a model proof ->

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

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

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

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

val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

Term_Occ 1

Term 1 Rule 1

(Rule 1 = exec.induct)

(Trm 1 = exec)

(Trm_Occ 1 = exec)

new types ->

new constants ->

new lemma ->
a model proof ->

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

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

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

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

val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

Term_Occ 1

Term 1 Rule 1

(Rule 1 = exec.induct)

(Trm 1 = exec)

(Trm_Occ 1 = exec)

new types ->

new constants ->

new lemma ->
a model proof ->

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

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

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

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

val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

Term_Occ 1

Term 1 Rule 1

(Rule 1 = exec.induct)

(Trm 1 = exec)

(Trm_Occ 1 = exec)

new types ->

new constants ->

new lemma ->
a model proof ->

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

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

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

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

val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1))))))))));
```

Term_Occ 1

Term 1 Rule 1

(Rule 1   = exec.induct)

(Trm 1   = exec)

(Trm_Occ 1 = exec)

new types ->

new constants ->

new lemma ->
a model proof ->

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

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

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

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

val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1))))))))));
```

Term 2

Term_Occ 1

Term 1 Rule 1

(Rule 1 = exec.induct)

(Trm 1 = exec)

(Trm_Occ 1 = exec)

(Trm 2 = is1, s, and stk)

new types ->

new constants ->

new lemma ->
a model proof ->

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

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

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

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

val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

Term 2
Term_Occ 1
Term_Occ 2

Term 1 Rule 1
(Rule 1   = exec.induct)
(Trm 1   = exec)
(Trm_Occ 1 = exec)
(Trm 2   = is1, s, and stk)
(Trm_Occ 2 = is1, s, and stk)

new types ->

new constants ->

new lemma ->
a model proof ->

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

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

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

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

val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1))))))))));
```

Term 2
Term_Occ 1
Term_Occ 2

Term 1 Rule 1
(Rule 1 = exec.induct)
(Trm 1 = exec)
(Trm_Occ 1 = exec)
(Trm 2 = is1, s, and stk)
(Trm_Occ 2 = is1, s, and stk)

new types ->

new constants ->

new lemma ->
a model proof ->

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

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

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

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

val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
        Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
       And
       (Trm 2 Is_Nth_Ind Numb 1))))))))));
```

Term 2
Term_Occ 1
Term_Occ 2

Term 1 Rule 1
(Rule 1   = exec.induct)
(Trm 1   = exec)
(Trm_Occ 1 = exec)

(Trm 2   = is1, s, and stk)
(Trm_Occ 2 = is1, s, and stk)

new types ->

new constants ->

new lemma ->

a model proof ->

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

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

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

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

val example_assertion =
 Some_Rule (Rule 1, True)
Imply
 Some_Rule (Rule 1,
  Some_Trm (Trm 1,
   Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
    (Rule 1 Is_Rule_Of Trm_Occ 1)
    And
    (All_Ind (Trm 2,
     (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
       Some_Numb (Numb 1,
        Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
       And
       (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

Term 2 — first

Term_Occ 1 — first

Term_Occ 2

Term 1 Rule 1

(Rule 1 = exec.induct)

(Trm 1 = exec)

(Trm_Occ 1 = exec)

(Trm 2 = is1, s, and stk)

(Trm_Occ 2 = is1, s, and stk)

new types ->

new constants ->

new lemma ->
a model proof ->

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

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

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

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

val example_assertion =
  Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1)))))))));
```

Term 2
first
second
Term_Occ 1
first
second
Term_Occ 2
Term 1 Rule 1
(Rule 1 = exec.induct)
(Trm 1 = exec)
(Trm_Occ 1 = exec)
(Trm 2 = is1, s, and stk)
(Trm_Occ 2 = is1, s, and stk)

new types ->

new constants ->

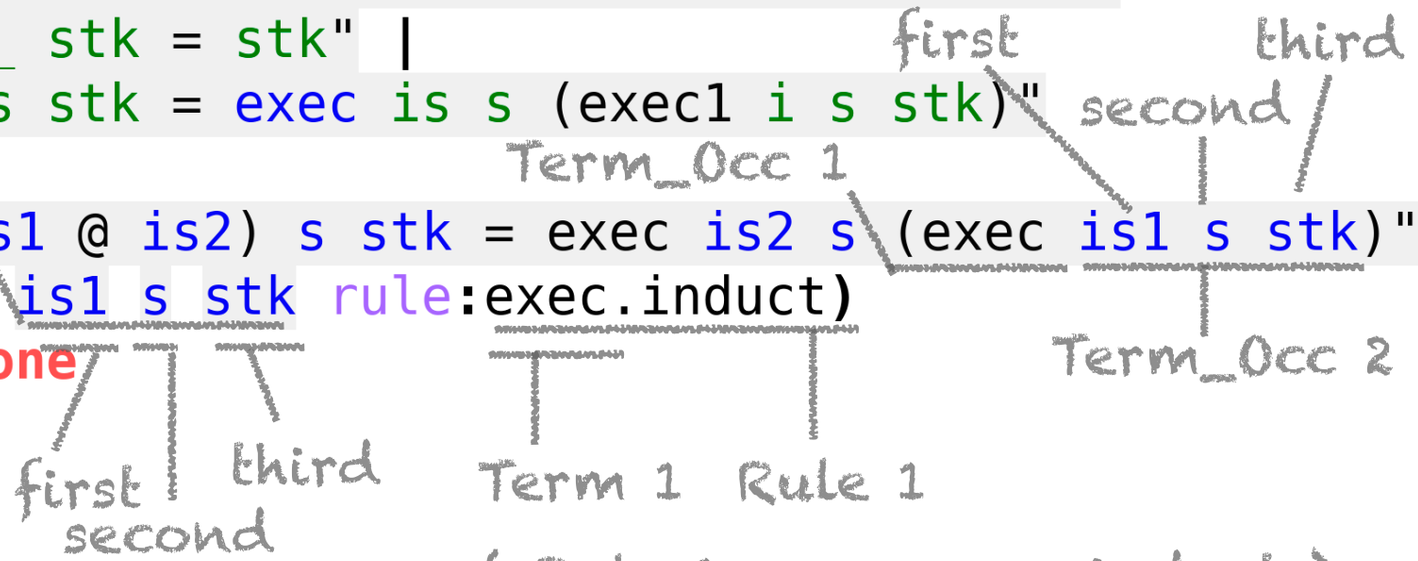
new lemma ->
a model proof ->

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

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

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

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




```
val example_assertion =
  Some_Rule (Rule 1, True)
 Imply
  Some_Rule (Rule 1,
   Some_Trm (Trm 1,
    Some_Trm_Occ_Of (Trm_Occ 1, Trm 1,
     (Rule 1 Is_Rule_Of Trm_Occ 1)
     And
     (All_Ind (Trm 2,
      (Some_Trm_Occ_Of (Trm_Occ 2, Trm 2,
        Some_Numb (Numb 1,
         Is_Nth_Arg_Of (Trm_Occ 2, Numb 1, Trm_Occ 1)
        And
        (Trm 2 Is_Nth_Ind Numb 1))))))))));
```

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

https://github.com/data61/PSL

Big Picture

ML

<- pros: good at ambiguity (heuristics)

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

LiFtEr LiFtEr LiFtEr 3

<- pros: good at rigorous abstraction

~~<- 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
```

LiFtEr 1 LiFtEr 2 LiFtEr 3

<- 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://github.com/data61/PSL

Big Picture

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

ML

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

<- pros: good at ambiguity (heuristics)

<- pros: good at rigoro… …tion

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

?

LiFtEr

LiFtEr

LiFtEr 3

LiFtEr 1

LiFtEr 2

LiFtEr 3

LiFtEr 1

LiFtEr 2

LiFtEr 3

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

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

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://github.com/data61/PSL

Big Picture

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

ML

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

<- pros: good at ambiguity (heuristics)

<- pros: good at rigoro… …on

?

LiFtEr

LiFtEr

LiFtEr 3

LiFtEr 1

LiFtEr 2

LiFtEr 3

LiFtEr 1

LiFtEr 2

LiFtEr 3

17:40 @ ML workshop

```
lemma "star r                star
by(induction r                ar.in                ut

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

<- small dataset about different domains

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

<- one abstract representation

logic

<- abstraction using expressive logic

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

<- many concrete cases

git clone https://github.com/data61/PSL