Automation of proof by induction in Isabelle/HOL using Domain-Specific Languages

LiFtEr: Logical Feature Extractor

SeLFiE: Semantic Logical Feature Extractor

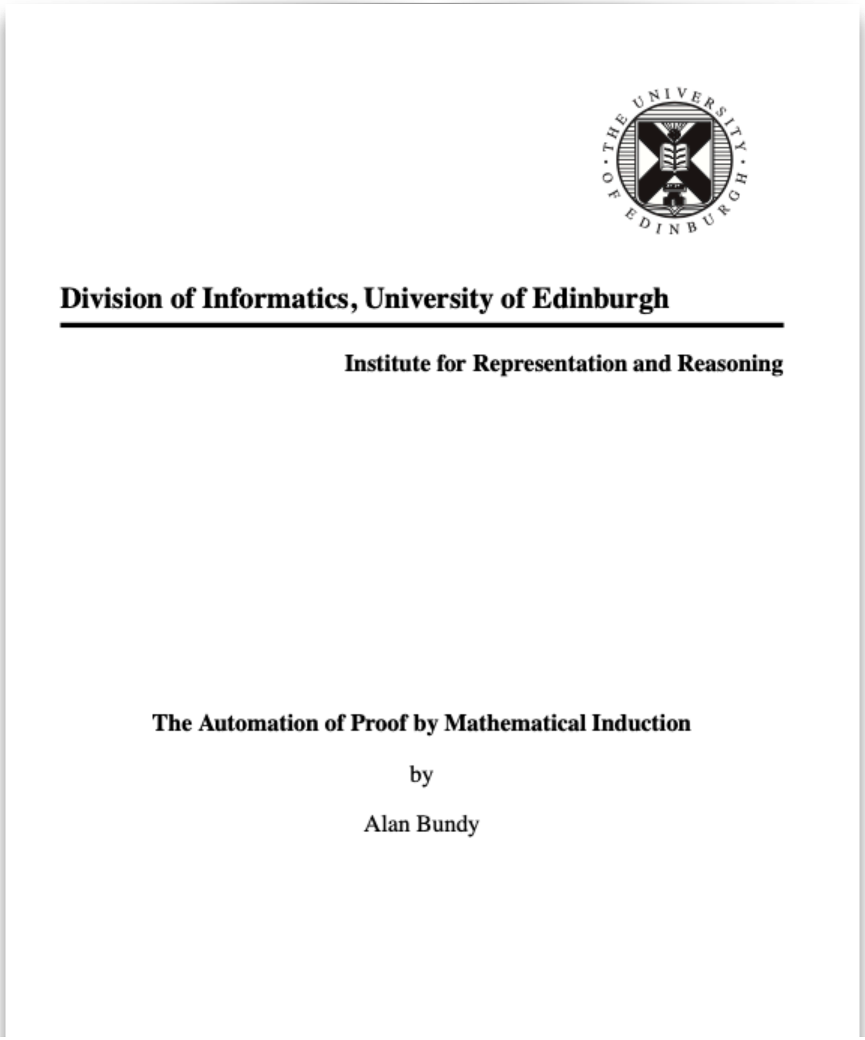
This work was supported by the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466).

universität
innsbruck

CZECH INSTITUTE
OF INFORMATICS
ROBOTICS AND
CYBERNETICS
CTU IN PRAGUE

Yutaka Nagashima, AITP, France, September 2020

Why proof by induction?



Division of Informatics, University of Edinburgh

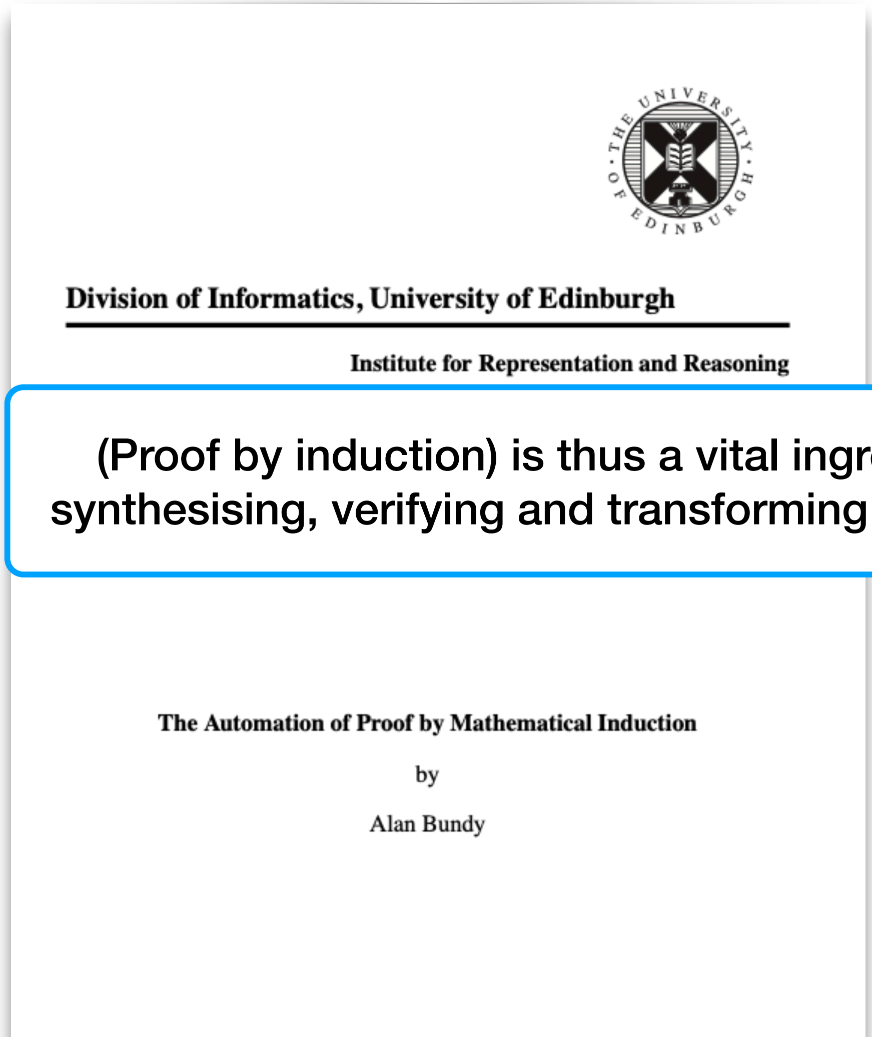
Institute for Representation and Reasoning

The Automation of Proof by Mathematical Induction

by

Alan Bundy

Why proof by induction?



Division of Informatics, University of Edinburgh

Institute for Representation and Reasoning

The Automation of Proof by Mathematical Induction

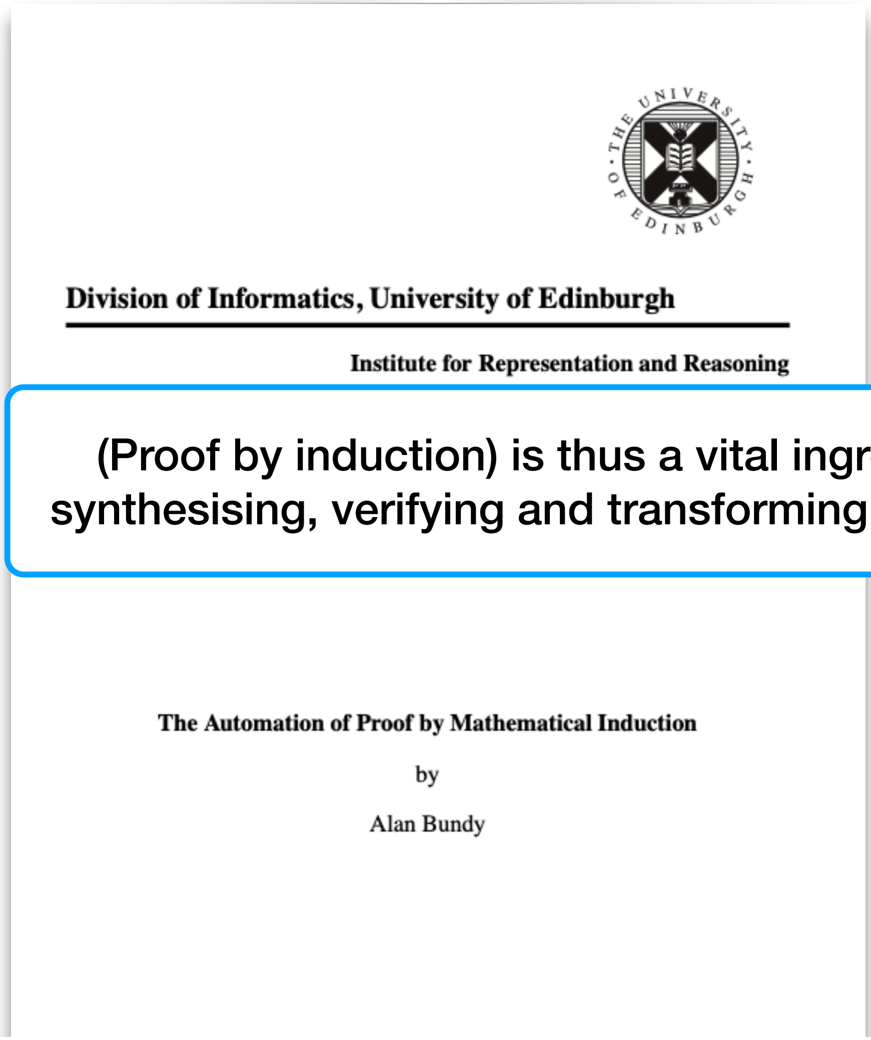
by

Alan Bundy

(Proof by induction) is thus a vital ingredient of formal methods for synthesising, verifying and transforming software and hardware. (1999)

Austintate
https://en.wikipedia.org/wiki/Alan_Bundy#/media/File:Alan.Bundy.Image.jpg
http://creativecommons.org/licenses/by-sa/3.0/

Why proof by induction?



Division of Informatics, University of Edinburgh

Institute for Representation and Reasoning

The Automation of Proof by Mathematical Induction

by

Alan Bundy

(Proof by induction) is thus a vital ingredient of formal methods for synthesising, verifying and transforming software and hardware. (1999)

Austintate
https://en.wikipedia.org/wiki/Alan_Bundy#/media/File:Alan.Bundy.Image.jpg
http://creativecommons.org/licenses/by-sa/3.0/

Why proof by induction?

German Research Center for Artificial Intelligence

JACOBS UNIVERSITY

UNIVERSITÄT DES SAARLANDES

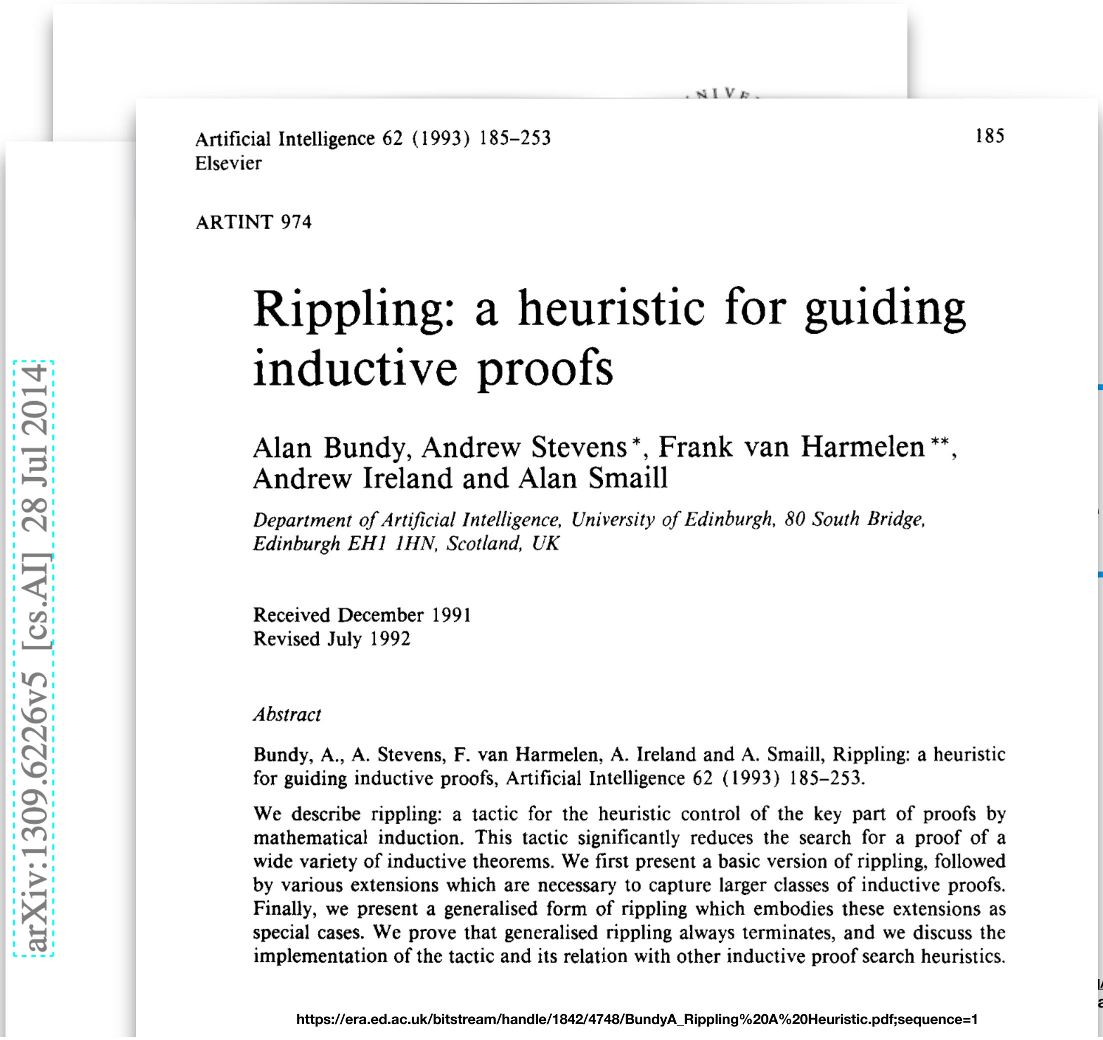
http://wirth.bplaced.net/seki.html

ISSN 1437-4447

of formal methods for
re and hardware. (1999)

Austintate
ia.org/wiki/Alan_Bundy#/media/File:Alan.Bundy.Image.jpg
://creativecommons.org/licenses/by-sa/3.0/

Why proof by induction?



arXiv:1309.6226v5 [cs.AI] 28 Jul 2014

Artificial Intelligence 62 (1993) 185–253
Elsevier

185

ARTINT 974

Rippling: a heuristic for guiding inductive proofs

Alan Bundy, Andrew Stevens*, Frank van Harmelen**, Andrew Ireland and Alan Smaill

Department of Artificial Intelligence, University of Edinburgh, 80 South Bridge, Edinburgh EH1 1HN, Scotland, UK

Received December 1991
Revised July 1992

Abstract

Bundy, A., A. Stevens, F. van Harmelen, A. Ireland and A. Smaill, Rippling: a heuristic for guiding inductive proofs, Artificial Intelligence 62 (1993) 185–253.

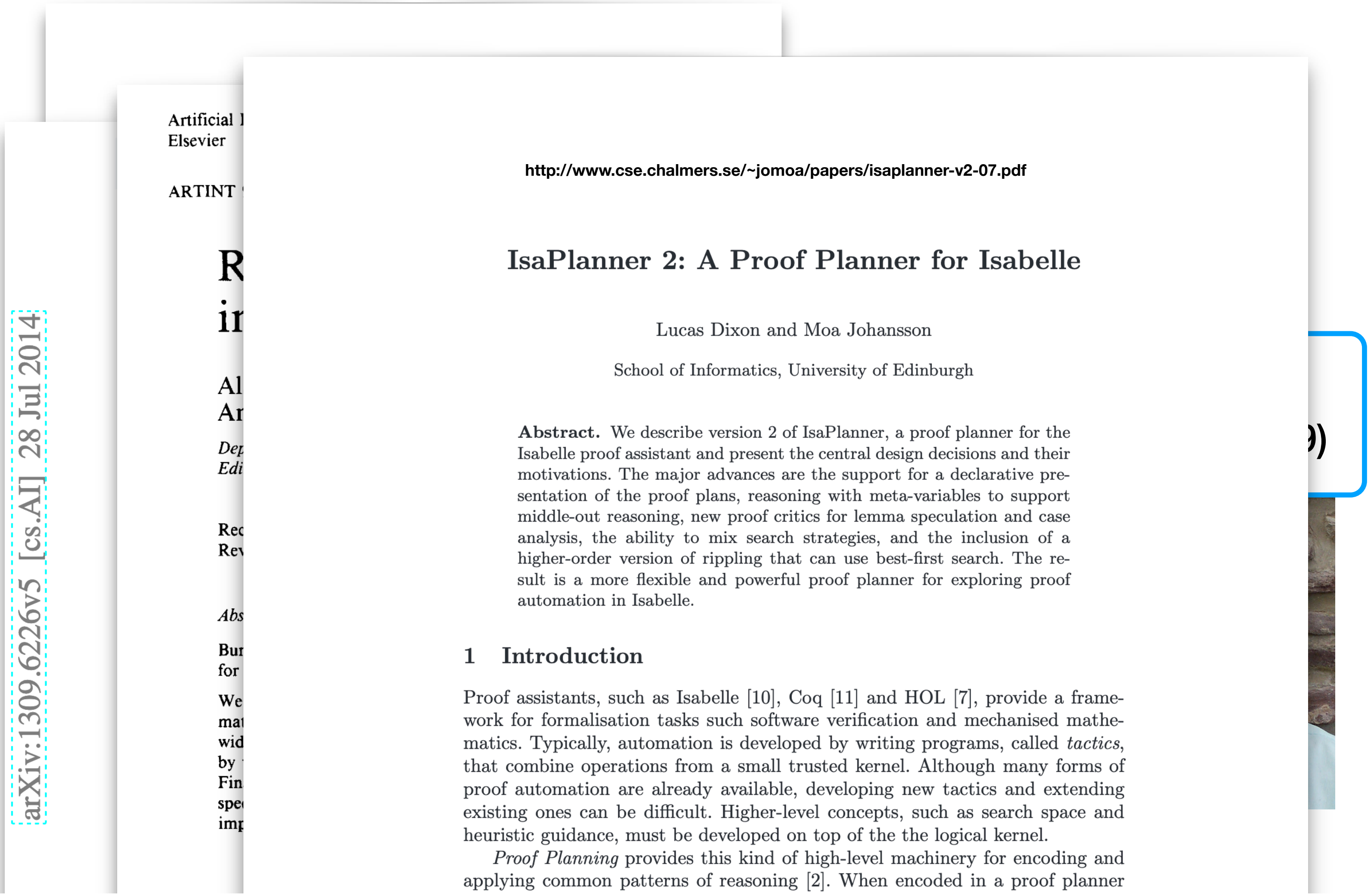
We describe rippling: a tactic for the heuristic control of the key part of proofs by mathematical induction. This tactic significantly reduces the search for a proof of a wide variety of inductive theorems. We first present a basic version of rippling, followed by various extensions which are necessary to capture larger classes of inductive proofs. Finally, we present a generalised form of rippling which embodies these extensions as special cases. We prove that generalised rippling always terminates, and we discuss the implementation of the tactic and its relation with other inductive proof search heuristics.

https://era.ed.ac.uk/bitstream/handle/1842/4748/BundyA_Rippling%20A%20Heuristic.pdf;sequence=1

formal methods for
and hardware. (1999)

Austintate
/wiki/Alan_Bundy#/media/File:Alan.Bundy.Image.jpg
ativecommons.org/licenses/by-sa/3.0/

Why proof by induction?



http://www.cse.chalmers.se/~jomoa/papers/isaplanner-v2-07.pdf

IsaPlanner 2: A Proof Planner for Isabelle

Lucas Dixon and Moa Johansson

School of Informatics, University of Edinburgh

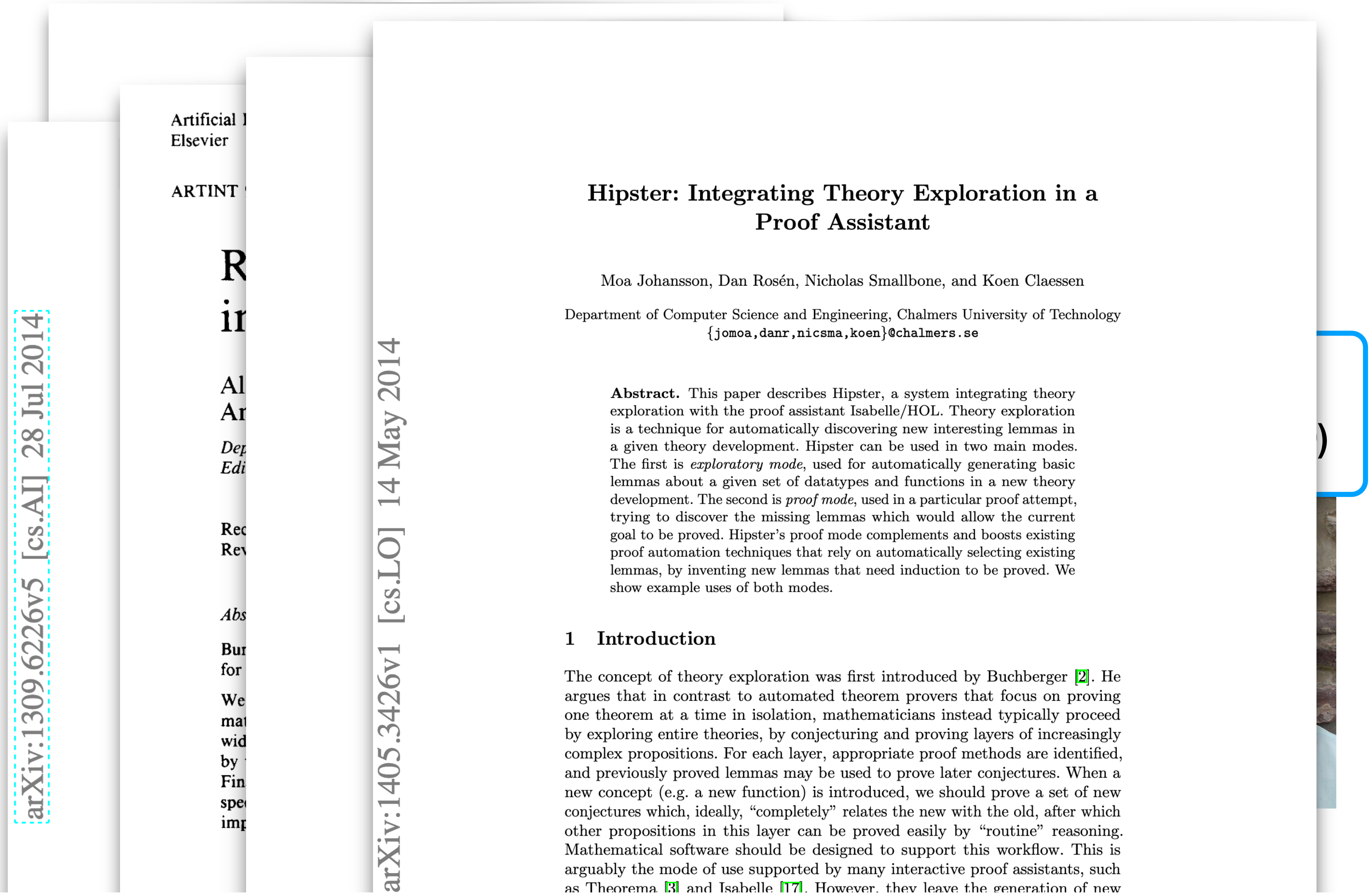
Abstract. We describe version 2 of IsaPlanner, a proof planner for the Isabelle proof assistant and present the central design decisions and their motivations. The major advances are the support for a declarative presentation of the proof plans, reasoning with meta-variables to support middle-out reasoning, new proof critics for lemma speculation and case analysis, the ability to mix search strategies, and the inclusion of a higher-order version of rippling that can use best-first search. The result is a more flexible and powerful proof planner for exploring proof automation in Isabelle.

1 Introduction

Proof assistants, such as Isabelle [10], Coq [11] and HOL [7], provide a framework for formalisation tasks such software verification and mechanised mathematics. Typically, automation is developed by writing programs, called *tactics*, that combine operations from a small trusted kernel. Although many forms of proof automation are already available, developing new tactics and extending existing ones can be difficult. Higher-level concepts, such as search space and heuristic guidance, must be developed on top of the the logical kernel.

Proof Planning provides this kind of high-level machinery for encoding and applying common patterns of reasoning [2]. When encoded in a proof planner

Why proof by induction?



arXiv:1309.6226v5 [cs.AI] 28 Jul 2014

Artificial I
Elsevier

ARTINT

arXiv:1405.3426v1 [cs.LO] 14 May 2014

Hipster: Integrating Theory Exploration in a Proof Assistant

Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen

Department of Computer Science and Engineering, Chalmers University of Technology
{jomoa,danr,nicsma,koen}@chalmers.se

Abstract. This paper describes Hipster, a system integrating theory exploration with the proof assistant Isabelle/HOL. Theory exploration is a technique for automatically discovering new interesting lemmas in a given theory development. Hipster can be used in two main modes. The first is *exploratory mode*, used for automatically generating basic lemmas about a given set of datatypes and functions in a new theory development. The second is *proof mode*, used in a particular proof attempt, trying to discover the missing lemmas which would allow the current goal to be proved. Hipster's proof mode complements and boosts existing proof automation techniques that rely on automatically selecting existing lemmas, by inventing new lemmas that need induction to be proved. We show example uses of both modes.

1 Introduction

The concept of theory exploration was first introduced by Buchberger [2]. He argues that in contrast to automated theorem provers that focus on proving one theorem at a time in isolation, mathematicians instead typically proceed by exploring entire theories, by conjecturing and proving layers of increasingly complex propositions. For each layer, appropriate proof methods are identified, and previously proved lemmas may be used to prove later conjectures. When a new concept (e.g. a new function) is introduced, we should prove a set of new conjectures which, ideally, "completely" relates the new with the old, after which other propositions in this layer can be proved easily by "routine" reasoning. Mathematical software should be designed to support this workflow. This is arguably the mode of use supported by many interactive proof assistants, such as Theorema [3] and Isabelle [17]. However, they leave the generation of new

Why proof by induction?

arXiv:1309.6226v5 [cs.AI] 28 Jul 2014

Artificial I
Elsevier

ARTINT

R
in

Al
Ar

Dep
Edi

Rec
Rev

Abs

Bur
for

We
mat
wid
by
Fin
spe
imp

arXiv:1405.3426v1 [cs.LO] 14 May 2014

arXiv:1606.02941v9 [cs.LO] 2 Mar 2017

https://doi.org/10.1007/978-3-319-63046-5_32

A Proof Strategy Language and Proof Script Generation for Isabelle/HOL

Yutaka Nagashima and Ramana Kumar

Data61, CSIRO / UNSW

Abstract. We introduce a language, PSL, designed to capture high level proof strategies in Isabelle/HOL. Given a strategy and a proof obligation, PSL's runtime system generates and combines various tactics to explore a large search space with low memory usage. Upon success, PSL generates an efficient proof script, which bypasses a large part of the proof search. We also present PSL's monadic interpreter to show that the underlying idea of PSL is transferable to other ITPs.

1 Introduction

Currently, users of interactive theorem provers (ITPs) spend too much time iteratively interacting with their ITP to manually specialise and combine tactics as depicted in Fig. 1a. This time consuming process requires expertise in the ITP, making ITPs more esoteric than they should be. The integration of powerful automated theorem provers (ATPs) into ITPs ameliorates this problem significantly; however, the exclusive reliance on general purpose ATPs makes it hard to exploit users' domain specific knowledge, leading to combinatorial explosion even for conceptually straight-forward conjectures.

To address this problem, we introduce PSL, a programmable, extensible, meta-tool based framework, to Isabelle/HOL [21]. We provide PSL (available on GitHub [17]) as a language, so that its users can encode *proof strategies*, abstract

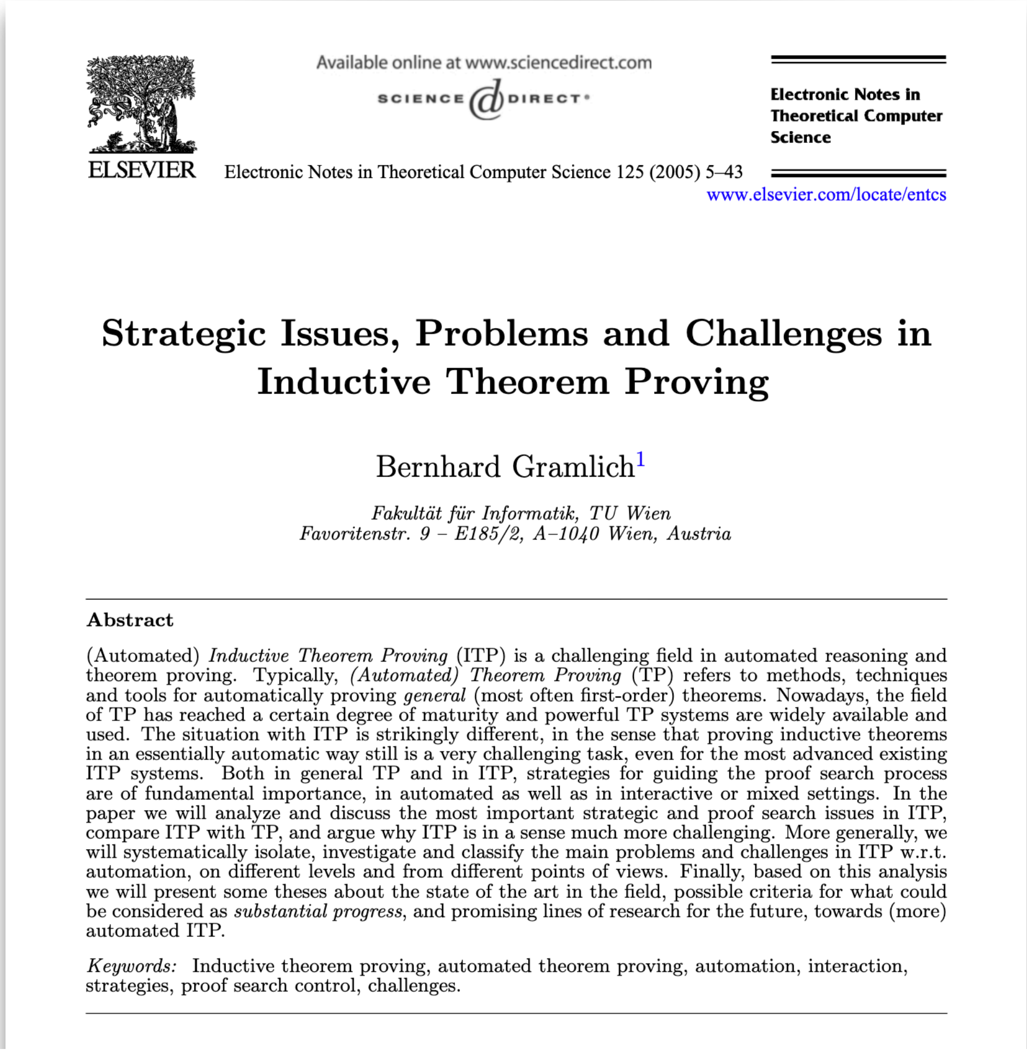
(a) Standard proof attempt

(b) Proof attempt with PSL

Proof by induction is hard!

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

Proof by induction is hard!



Available online at www.sciencedirect.com

SCIENCE DIRECT®

ELSEVIER

Electronic Notes in Theoretical Computer Science 125 (2005) 5–43

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

Bernhard Gramlich[1]

Fakultät für Informatik, TU Wien
Favoritenstr. 9 – E185/2, A–1040 Wien, Austria

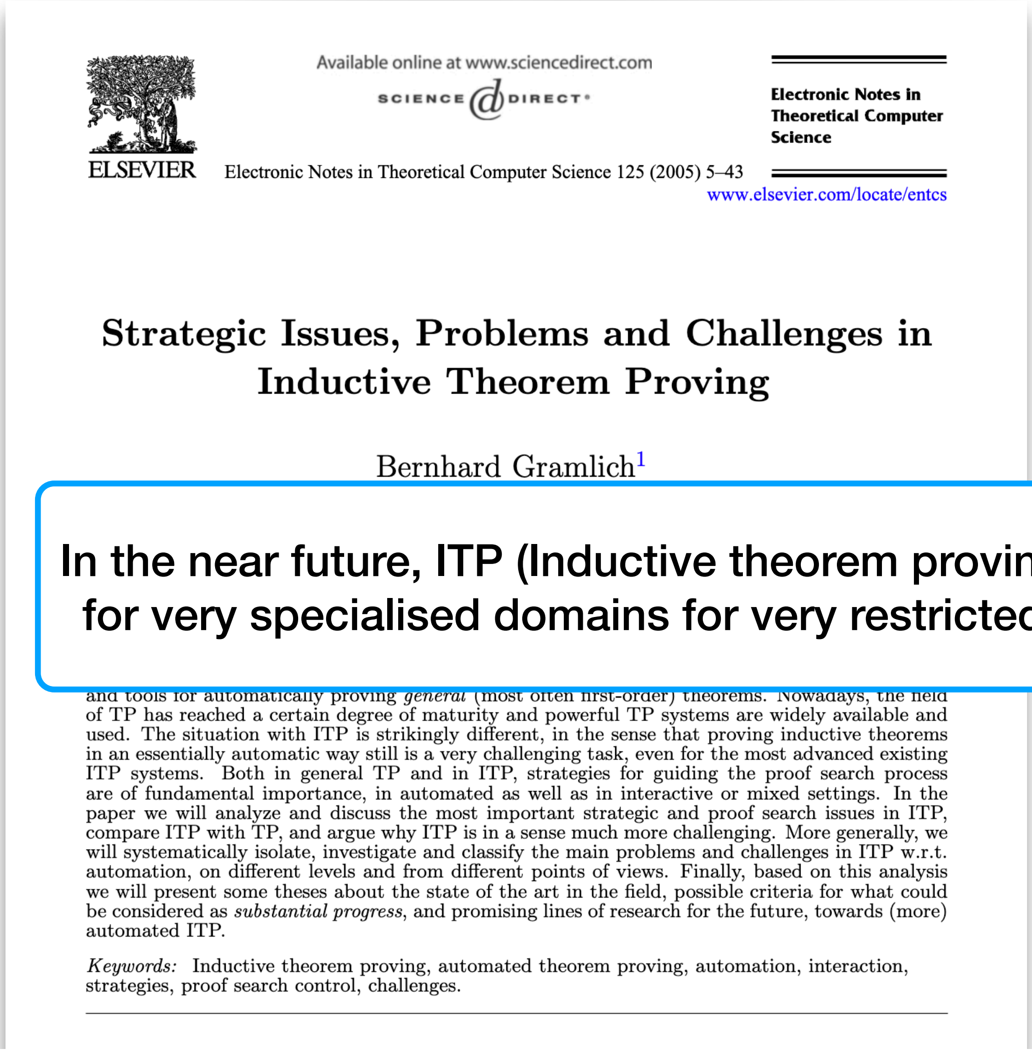
Abstract

(Automated) *Inductive Theorem Proving* (ITP) is a challenging field in automated reasoning and theorem proving. Typically, *(Automated) Theorem Proving* (TP) refers to methods, techniques and tools for automatically proving *general* (most often first-order) theorems. Nowadays, the field of TP has reached a certain degree of maturity and powerful TP systems are widely available and used. The situation with ITP is strikingly different, in the sense that proving inductive theorems in an essentially automatic way still is a very challenging task, even for the most advanced existing ITP systems. Both in general TP and in ITP, strategies for guiding the proof search process are of fundamental importance, in automated as well as in interactive or mixed settings. In the paper we will analyze and discuss the most important strategic and proof search issues in ITP, compare ITP with TP, and argue why ITP is in a sense much more challenging. More generally, we will systematically isolate, investigate and classify the main problems and challenges in ITP w.r.t. automation, on different levels and from different points of views. Finally, based on this analysis we will present some theses about the state of the art in the field, possible criteria for what could be considered as *substantial progress*, and promising lines of research for the future, towards (more) automated ITP.

Keywords: Inductive theorem proving, automated theorem proving, automation, interaction, strategies, proof search control, challenges.

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

Proof by induction is hard!



Available online at www.sciencedirect.com

SCIENCE DIRECT

ELSEVIER

Electronic Notes in Theoretical Computer Science 125 (2005) 5–43

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

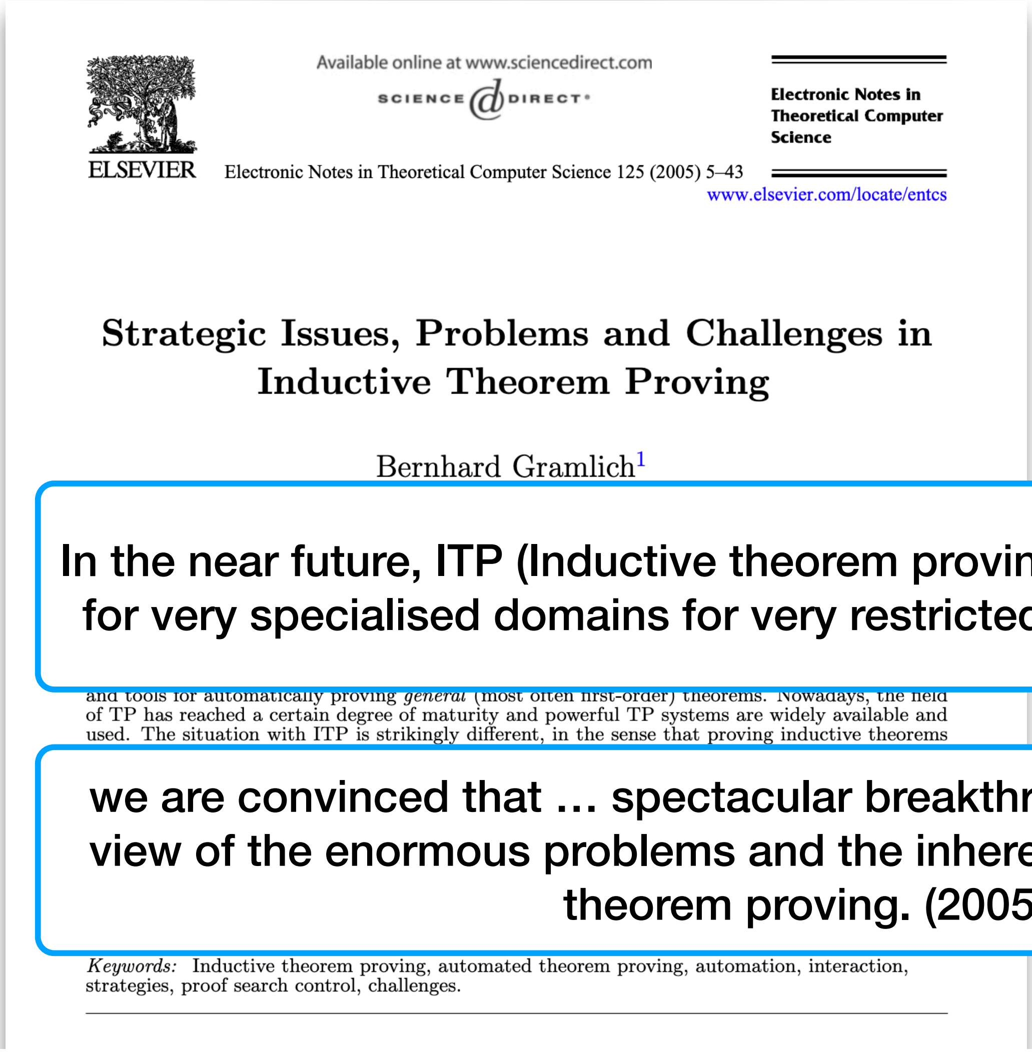
Bernhard Gramlich[1]

and tools for automatically proving *general* (most often first-order) theorems. Nowadays, the field of TP has reached a certain degree of maturity and powerful TP systems are widely available and used. The situation with ITP is strikingly different, in the sense that proving inductive theorems in an essentially automatic way still is a very challenging task, even for the most advanced existing ITP systems. Both in general TP and in ITP, strategies for guiding the proof search process are of fundamental importance, in automated as well as in interactive or mixed settings. In the paper we will analyze and discuss the most important strategic and proof search issues in ITP, compare ITP with TP, and argue why ITP is in a sense much more challenging. More generally, we will systematically isolate, investigate and classify the main problems and challenges in ITP w.r.t. automation, on different levels and from different points of views. Finally, based on this analysis we will present some theses about the state of the art in the field, possible criteria for what could be considered as *substantial progress*, and promising lines of research for the future, towards (more) automated ITP.

Keywords: Inductive theorem proving, automated theorem proving, automation, interaction, strategies, proof search control, challenges.

In the near future, ITP (Inductive theorem proving) will only be successful for very specialised domains for very restricted classes of conjectures.

Proof by induction is hard!



Available online at www.sciencedirect.com

SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science

ELSEVIER

Electronic Notes in Theoretical Computer Science 125 (2005) 5–43

www.elsevier.com/locate/entcs

Strategic Issues, Problems and Challenges in Inductive Theorem Proving

Bernhard Gramlich[1]

and tools for automatically proving *general* (most often first-order) theorems. Nowadays, the field of TP has reached a certain degree of maturity and powerful TP systems are widely available and used. The situation with ITP is strikingly different, in the sense that proving inductive theorems

Keywords: Inductive theorem proving, automated theorem proving, automation, interaction, strategies, proof search control, challenges.

In the near future, ITP (Inductive theorem proving) will only be successful for very specialised domains for very restricted classes of conjectures.

we are convinced that … spectacular breakthroughs are unrealistic, in view of the enormous problems and the inherent difficulty of inductive theorem proving. (2005)

Proof by induction is important.

Proof by induction is hard.

✓ Proof by induction is important.

Proof by induction is hard.

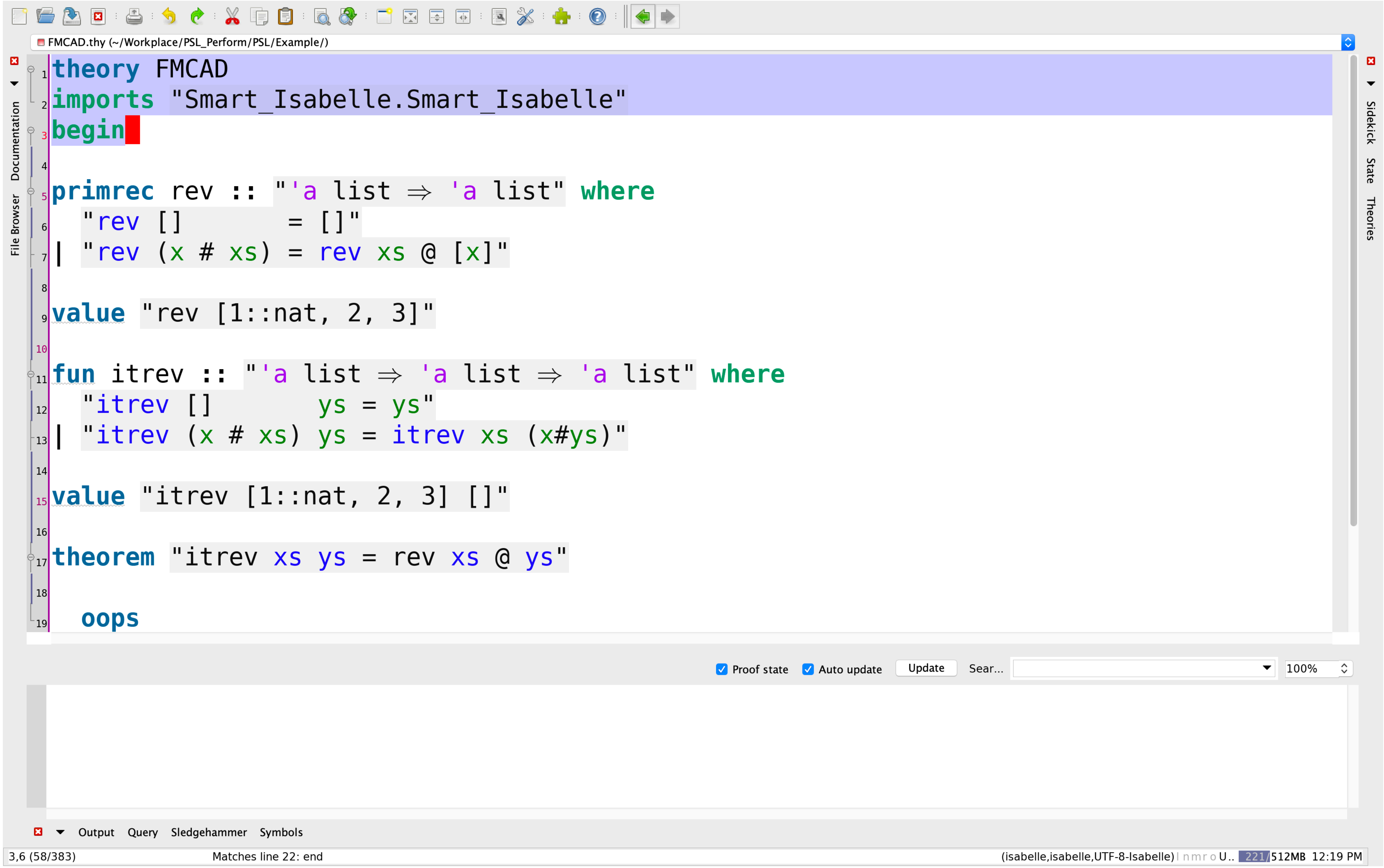
- ✓ Proof by induction is important.
- ✓ Proof by induction is hard.

- ✓ Proof by induction is important.
- ✓ Proof by induction is hard.

DEMO

proof by induction in Isabelle/HOL

The example theorem is taken from “Isabelle/HOL A Proof Assistant for Higher-Order Logic” Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36

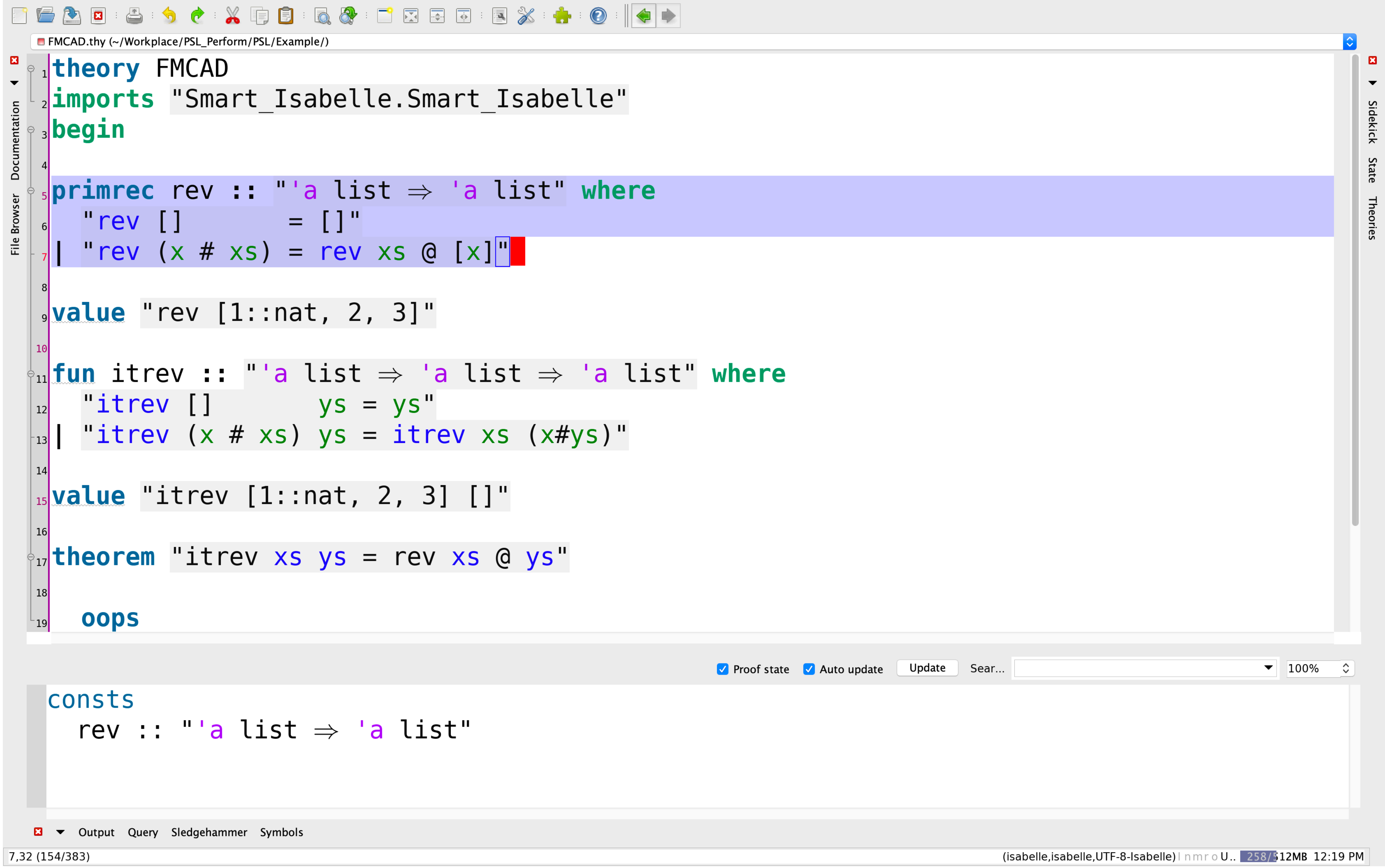


FMCAD.thy (~/Workplace/PSL_Perform/PSL/Example/)

```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"

  oops
```

☑ Proof state ☑ Auto update Update Sear... 100%

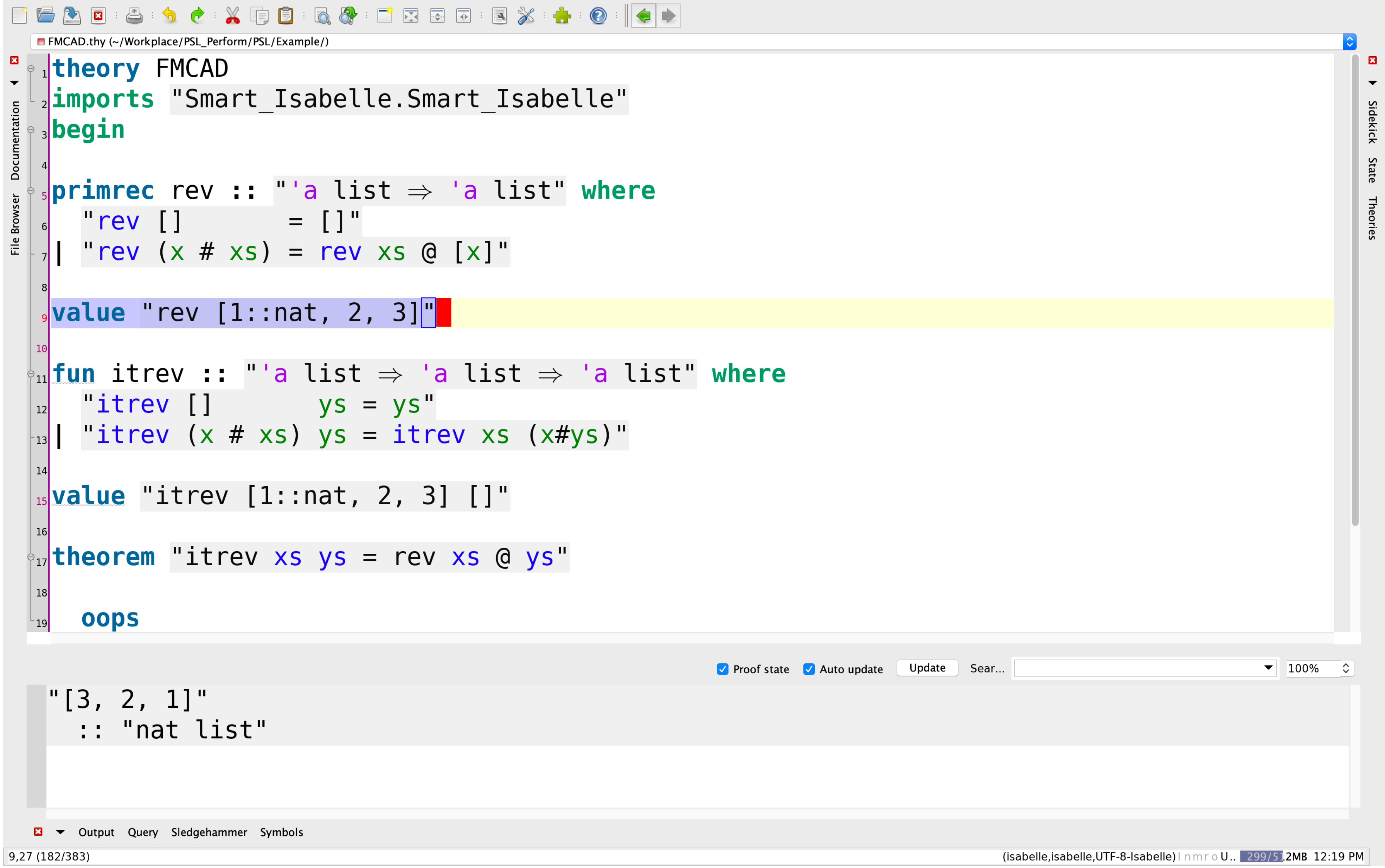
3,6 (58/383) Matches line 22: end (isabelle,isabelle,UTF-8-Isabelle) | nmro U.. 221/512MB 12:19 PM



```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"

  oops
```

```
consts
  rev :: "'a list ⇒ 'a list"
```

Output
Query
Sledgehammer
Symbols
7,32 (154/383)
(isabelle,isabelle,UTF-8-Isabelle) | n m r o U.. 258/512MB 12:19 PM



FMCAD.thy (~/Workplace/PSL_Perform/PSL/Example/)

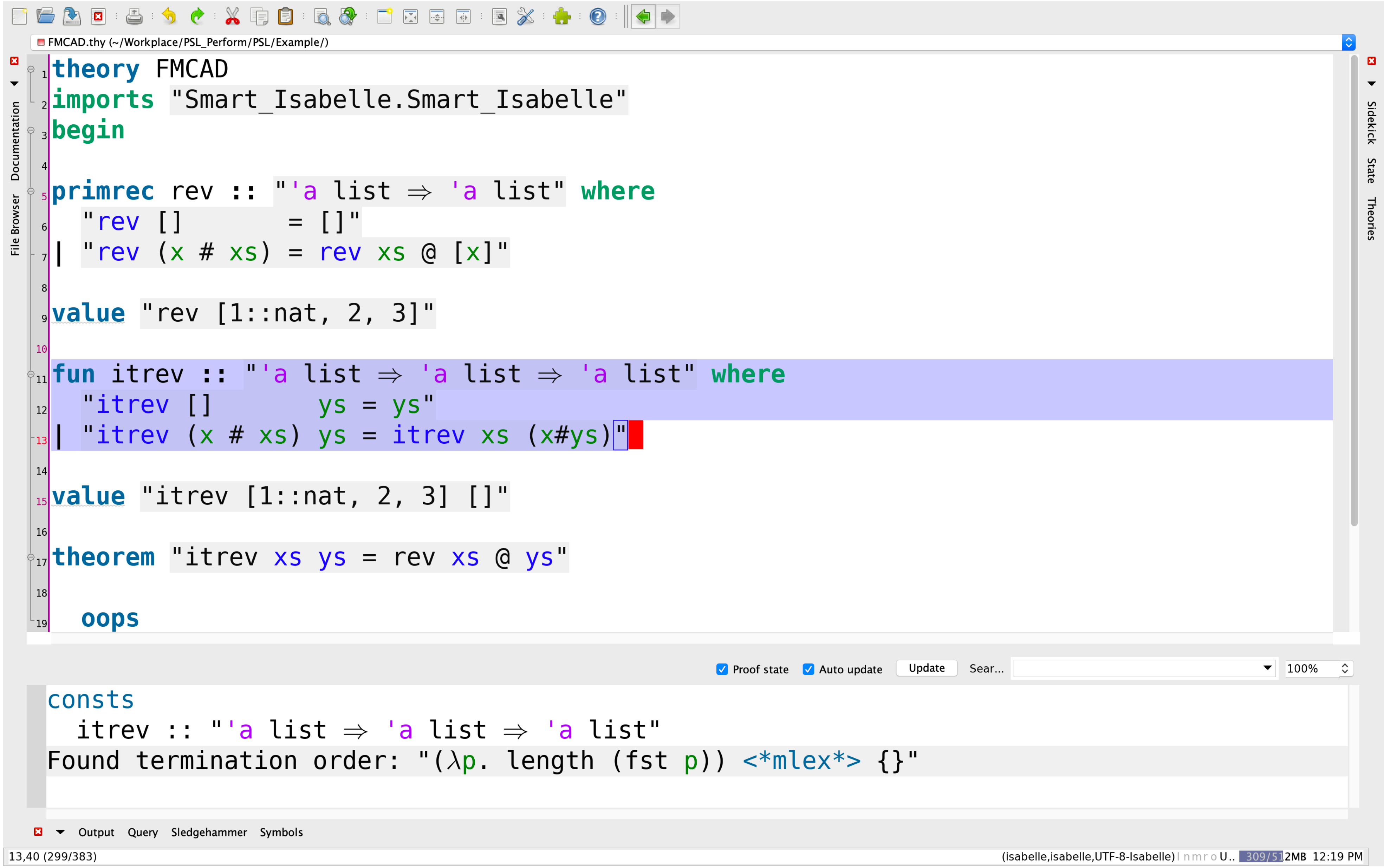
```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"

  oops
```

☑ Proof state ☑ Auto update Update Sear... 100%

```
"[3, 2, 1]"
  :: "nat list"
```

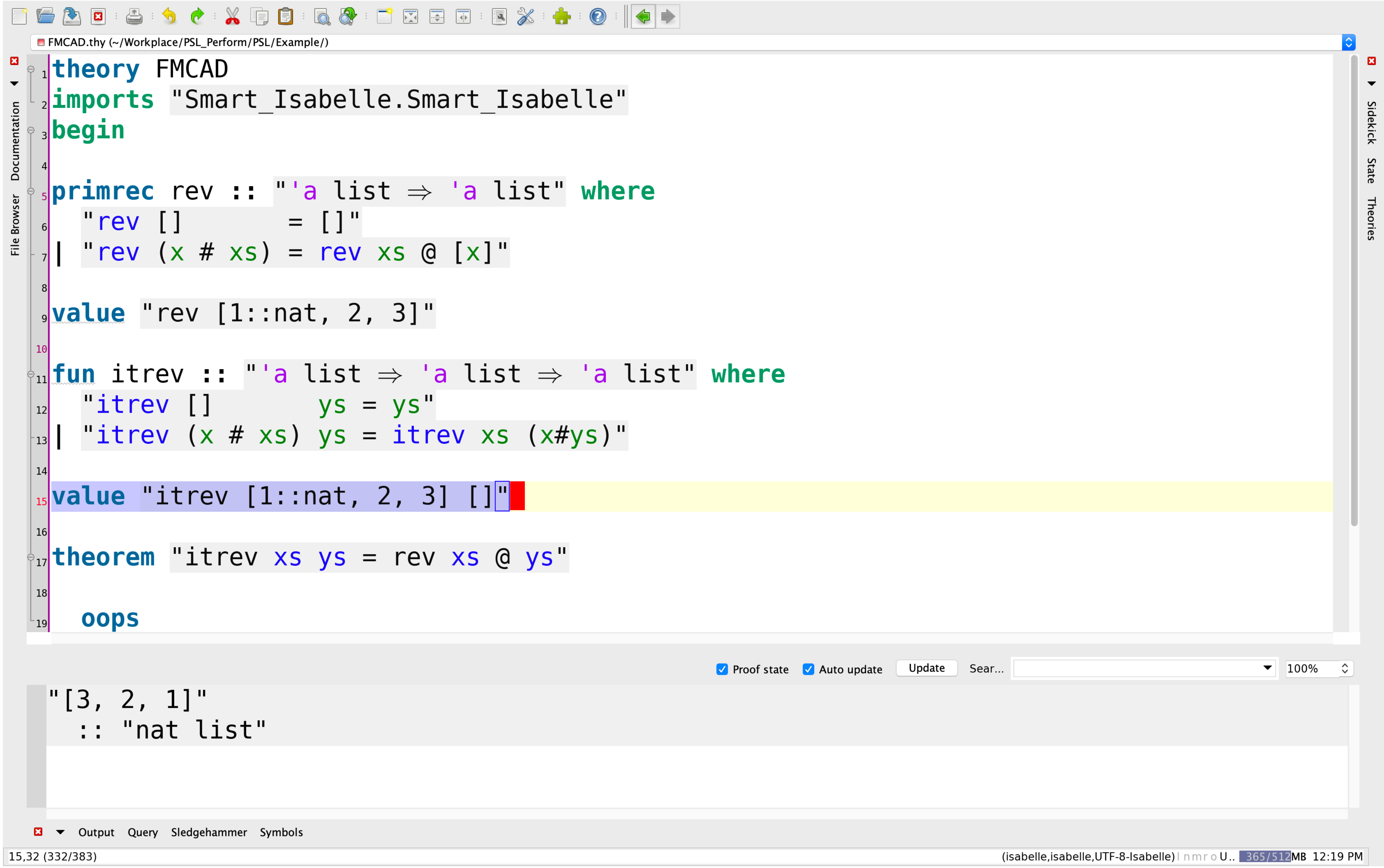
Output Query Sledgehammer Symbols

9,27 (182/383) (isabelle,isabelle,UTF-8-Isabelle) nmro U.. 299/512MB 12:19 PM



```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"

  oops
```

```
consts
  itrev :: "'a list ⇒ 'a list ⇒ 'a list"
Found termination order: "(λp. length (fst p)) <*mlex*> {}"
```



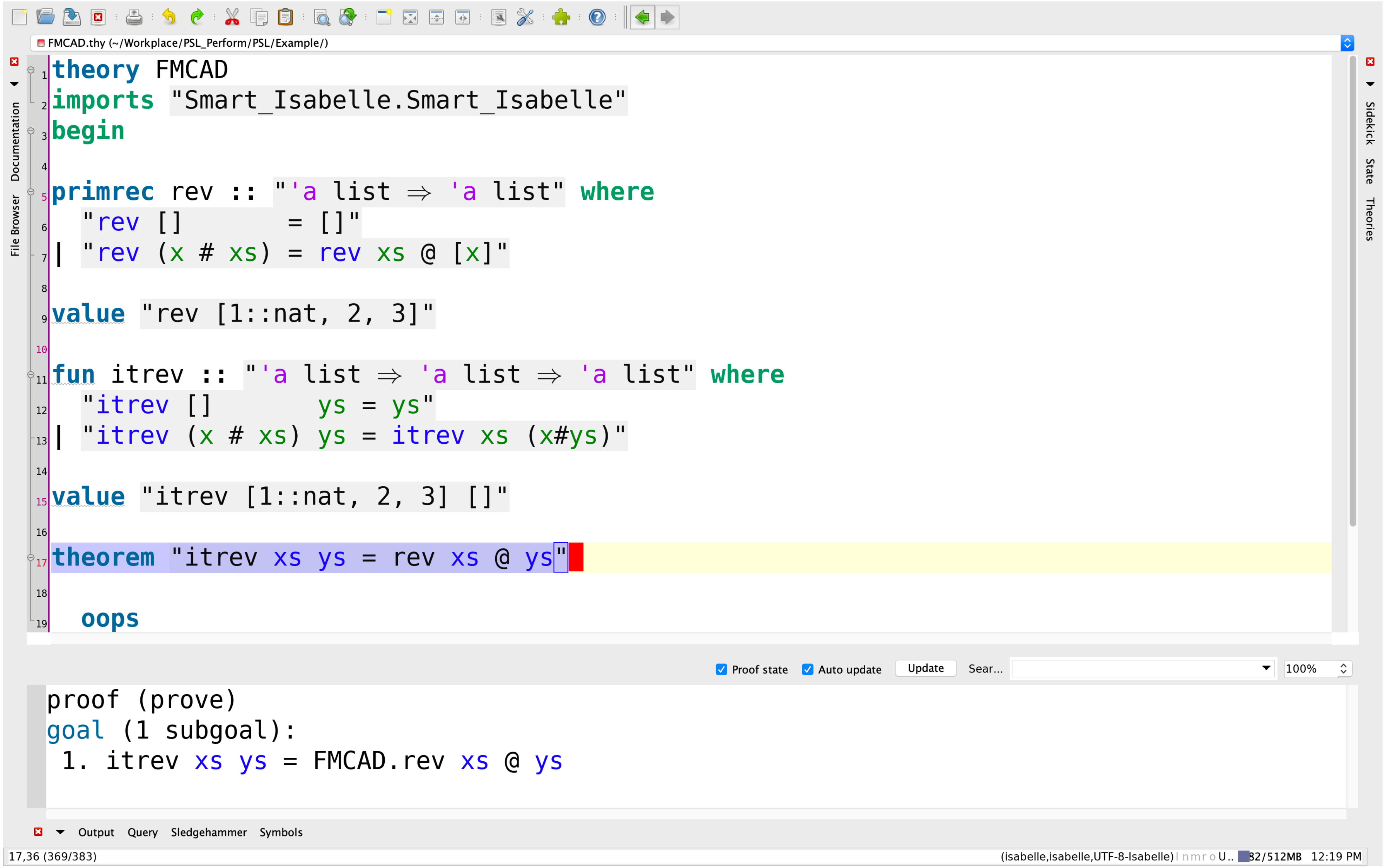
FMCAD.thy (~/Workplace/PSL_Perform/PSL/Example/)
File Browser
Documentation
Sidekick
State
Theories
Update
Sear...
100%
Output
Query
Sledgehammer
Symbols
(isabelle,isabelle,UTF-8-Isabelle) nmro U.. 365/512MB 12:19 PM

```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"

  oops
```

☑ Proof state ☑ Auto update

```
"[3, 2, 1]"
  :: "nat list"
```

15,32 (332/383)



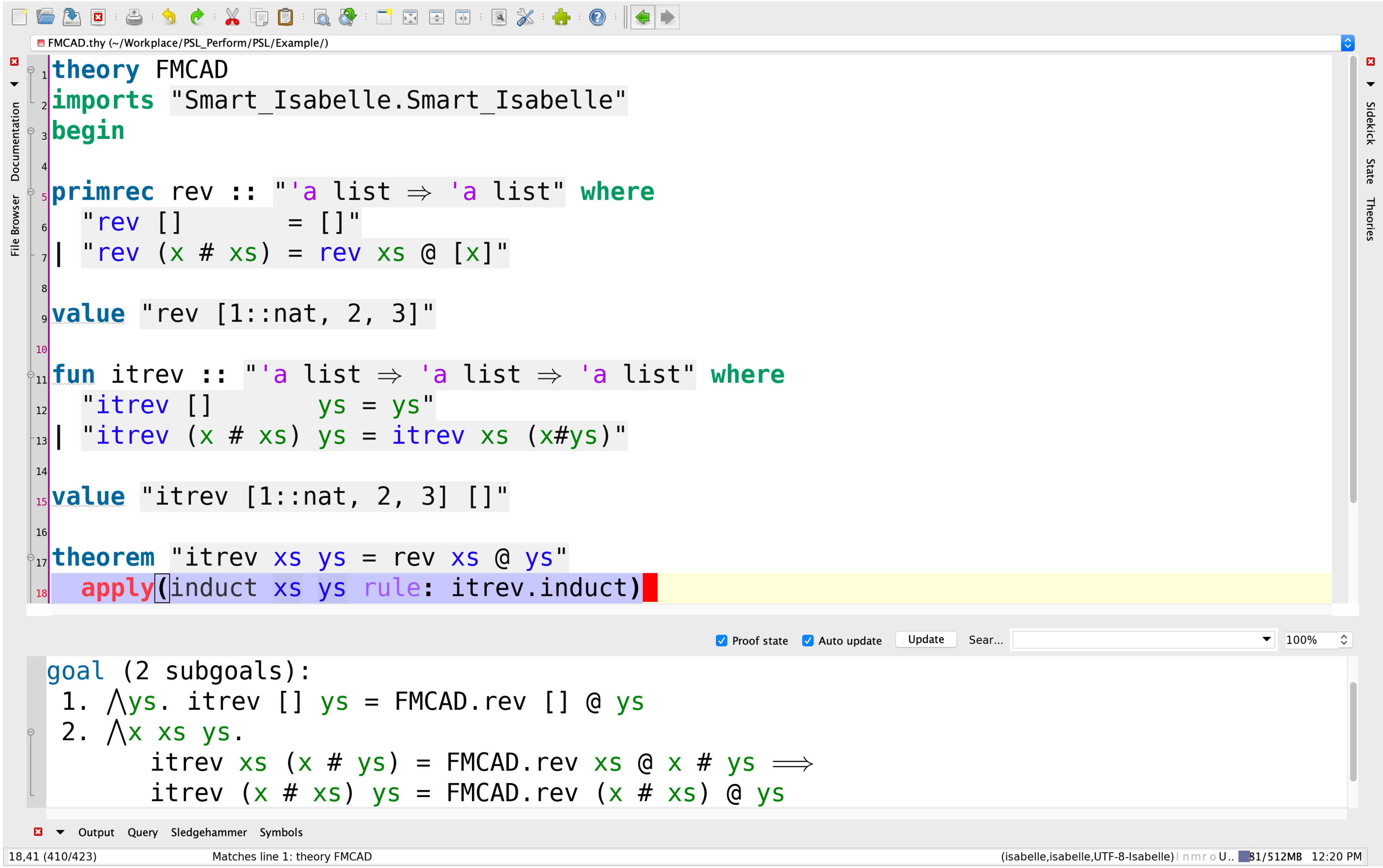
FMCAD.thy (~/Workplace/PSL_Perform/PSL/Example/)

```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"

  oops
```

☑ Proof state ☑ Auto update 100%

```
proof (prove)
goal (1 subgoal):
 1. itrev xs ys = FMCAD.rev xs @ ys
```

17,36 (369/383) (isabelle,isabelle,UTF-8-Isabelle) | n m r o U.. 82/512MB 12:19 PM

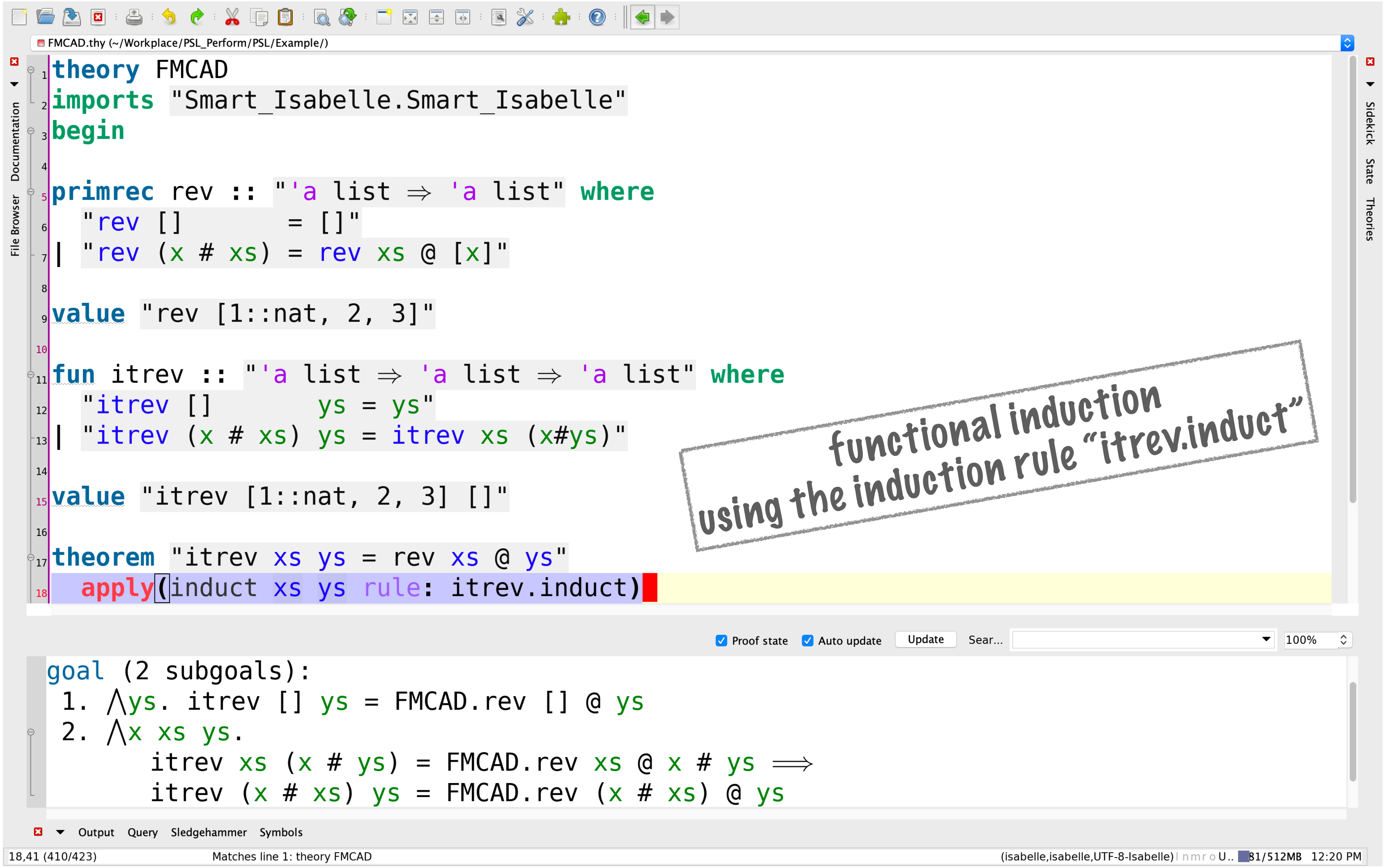


```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs ys rule: itrev.induct)
```

☑ Proof state ☑ Auto update

```
goal (2 subgoals):
 1. ⋀ys. itrev [] ys = FMCAD.rev [] @ ys
 2. ⋀x xs ys.
       itrev xs (x # ys) = FMCAD.rev xs @ x # ys ⟹
       itrev (x # xs) ys = FMCAD.rev (x # xs) @ ys
```

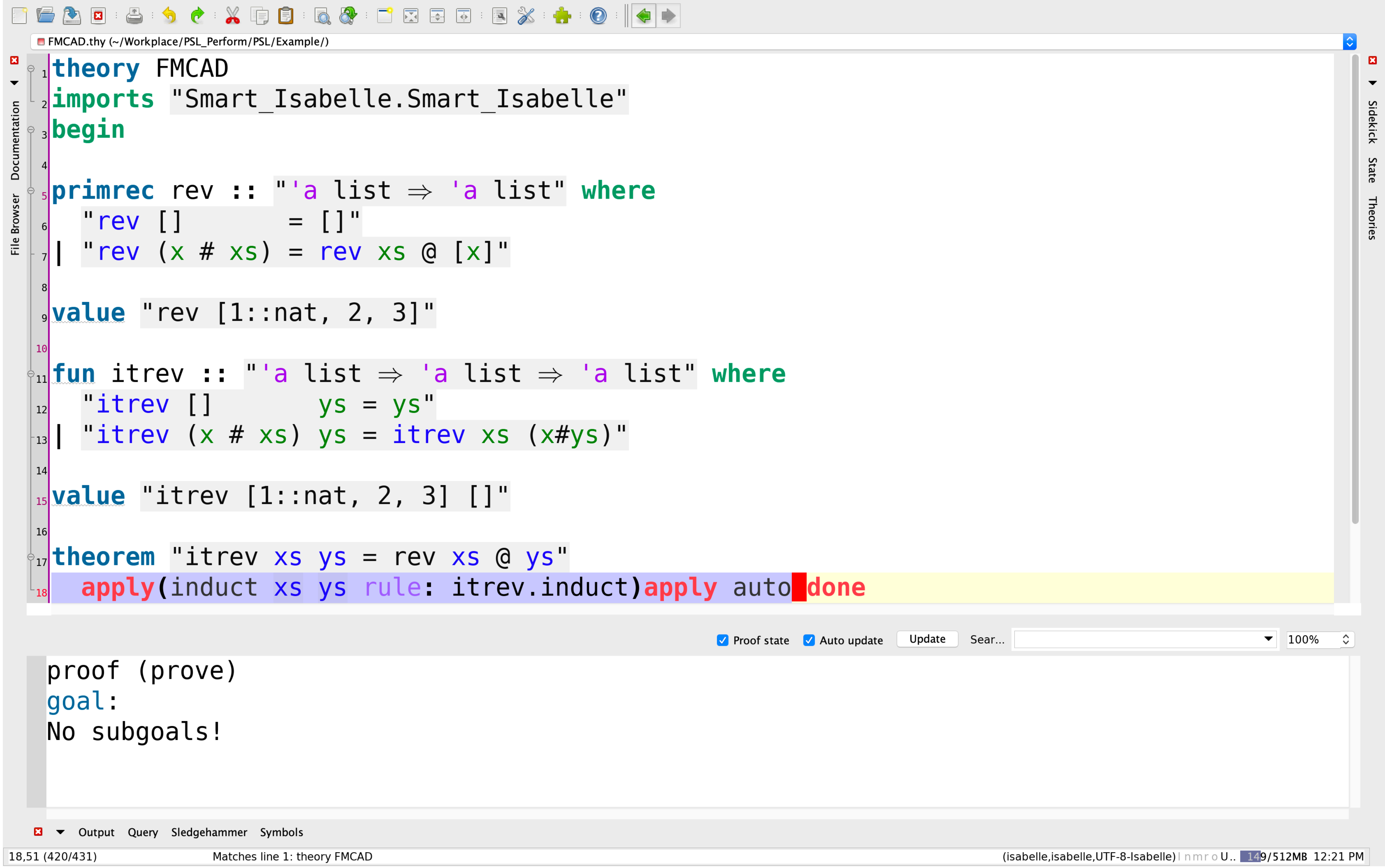
Output
Query
Sledgehammer
Symbols
18,41 (410/423)
Matches line 1: theory FMCAD
(isabelle,isabelle,UTF-8-Isabelle) | nmro U.. 81/512MB 12:20 PM


FMCAD.thy (~/Workplace/PSL_Perform/PSL/Example/)
Documentation
File Browser
Sidekick
State
Theories
☑ Proof state
☑ Auto update
Update
Sear...
100%
Output
Query
Sledgehammer
Symbols
18,41 (410/423)
Matches line 1: theory FMCAD
(isabelle,isabelle,UTF-8-Isabelle) | nmro U.. 81/512MB 12:20 PM

```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs ys rule: itrev.induct)
```

```
goal (2 subgoals):
 1. ⋀ys. itrev [] ys = FMCAD.rev [] @ ys
 2. ⋀x xs ys.
       itrev xs (x # ys) = FMCAD.rev xs @ x # ys ⟹
       itrev (x # xs) ys = FMCAD.rev (x # xs) @ ys
```

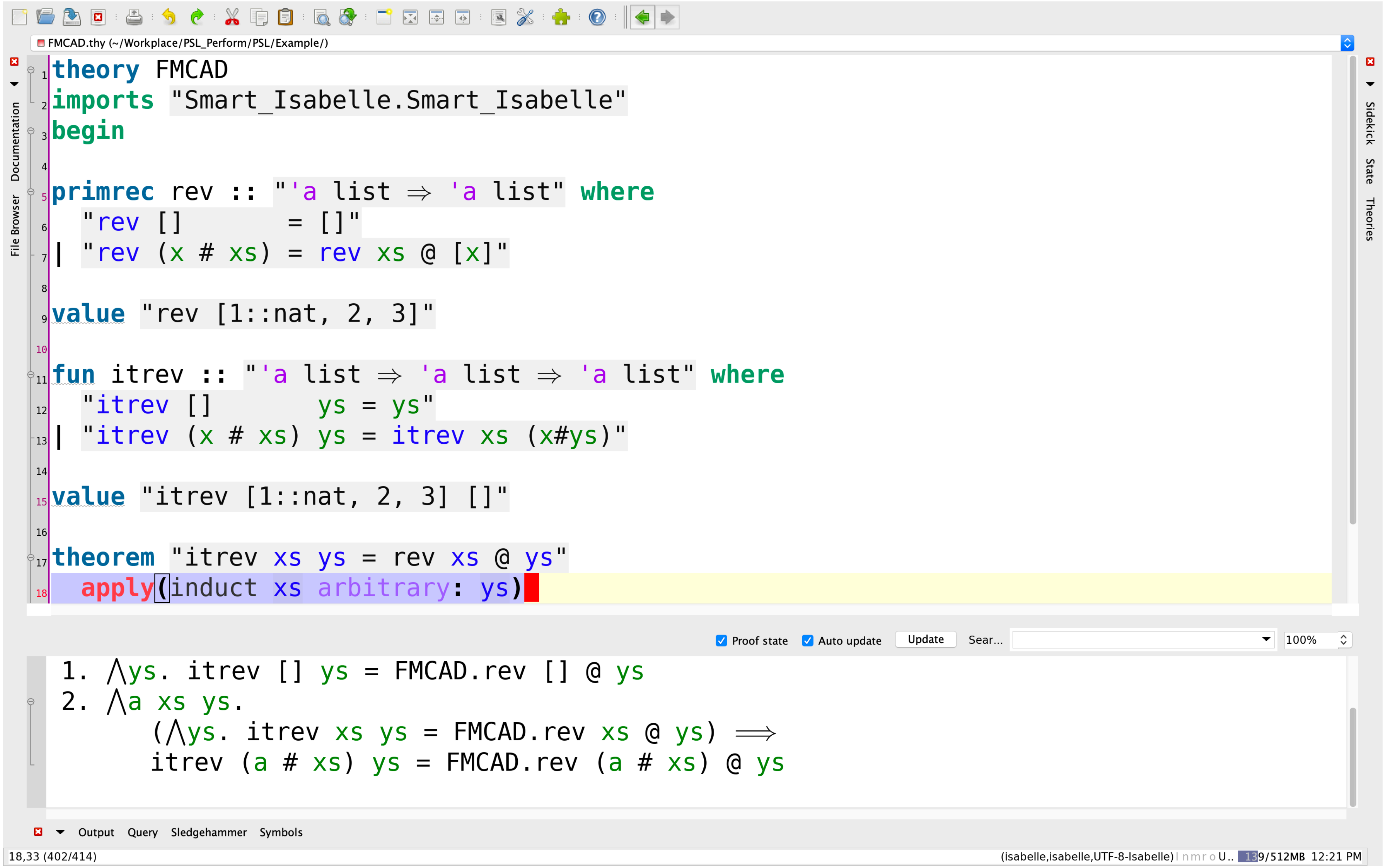
functional induction
using the induction rule “itrev.induct”



FMCAD.thy (~/Workplace/PSL_Perform/PSL/Example/)
Documentation
File Browser
Sidekick
State
Theories
☑ Proof state
☑ Auto update
Update
Sear...
100%
Output
Query
Sledgehammer
Symbols
18,51 (420/431)
Matches line 1: theory FMCAD
(isabelle,isabelle,UTF-8-Isabelle) | n m r o U.. 149/512MB 12:21 PM

```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

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

```
proof (prove)
goal:
No subgoals!
```



FMCAD.thy (~/Workplace/PSL_Perform/PSL/Example/)

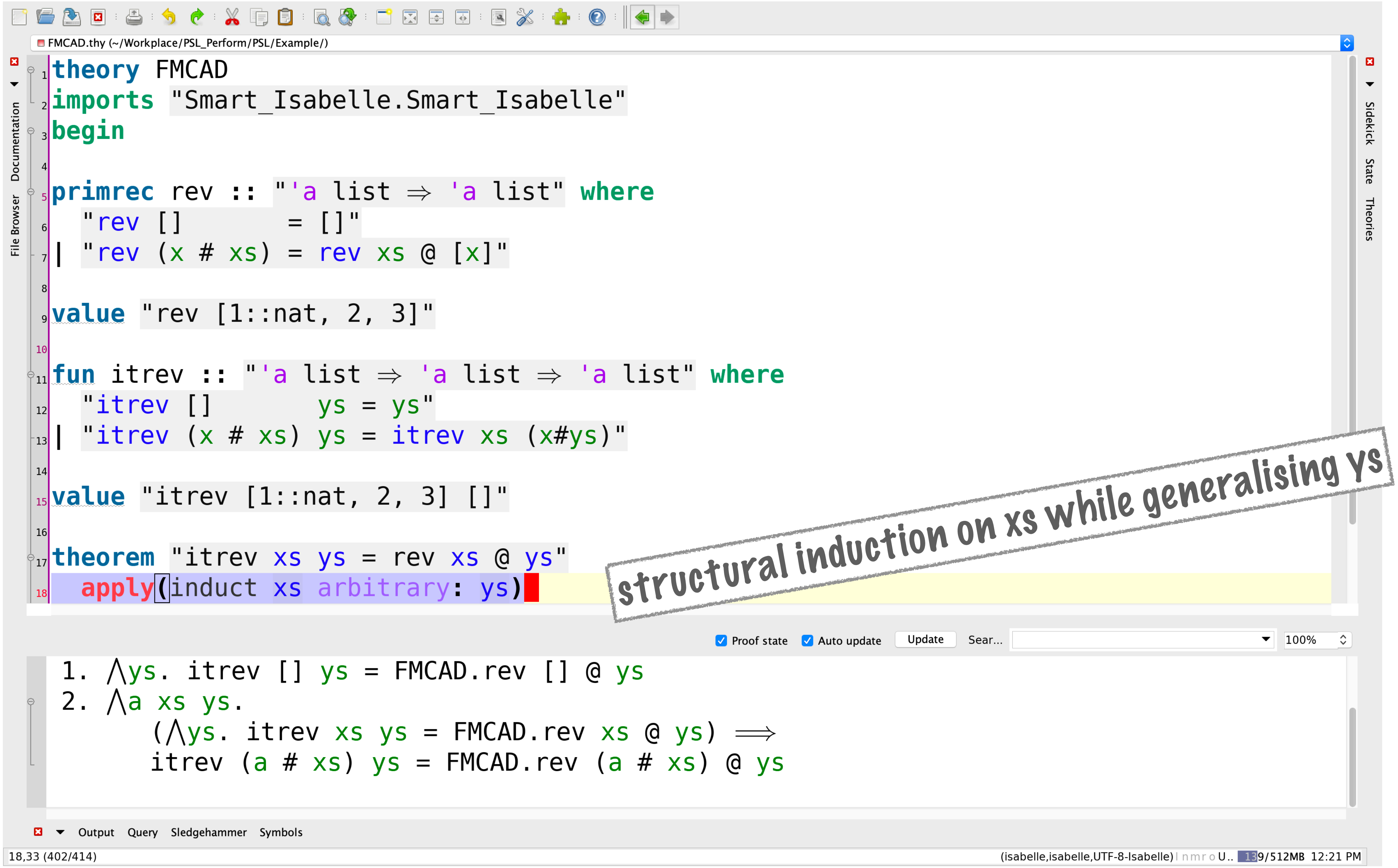
```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

☑ Proof state ☑ Auto update

```
1. ⋀ys. itrev [] ys = FMCAD.rev [] @ ys
2. ⋀a xs ys.
      (⋀ys. itrev xs ys = FMCAD.rev xs @ ys) ⟹
      itrev (a # xs) ys = FMCAD.rev (a # xs) @ ys
```

18,33 (402/414)

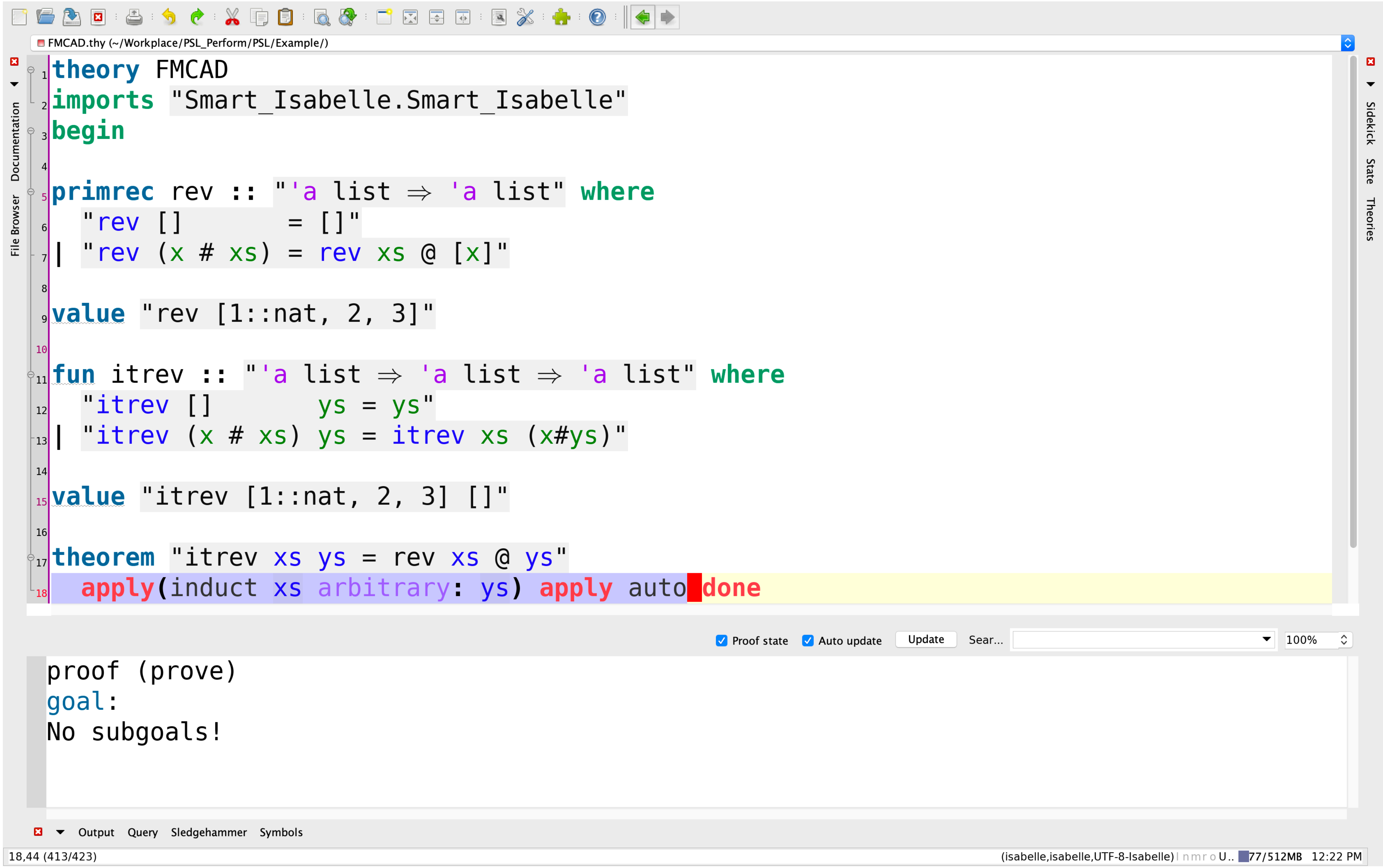
(isabelle,isabelle,UTF-8-Isabelle) | n m r o U.. 139/512MB 12:21 PM



```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

```
1. ⋀ys. itrev [] ys = FMCAD.rev [] @ ys
2. ⋀a xs ys.
      (⋀ys. itrev xs ys = FMCAD.rev xs @ ys) ⟹
      itrev (a # xs) ys = FMCAD.rev (a # xs) @ ys
```

structural induction on xs while generalising ys



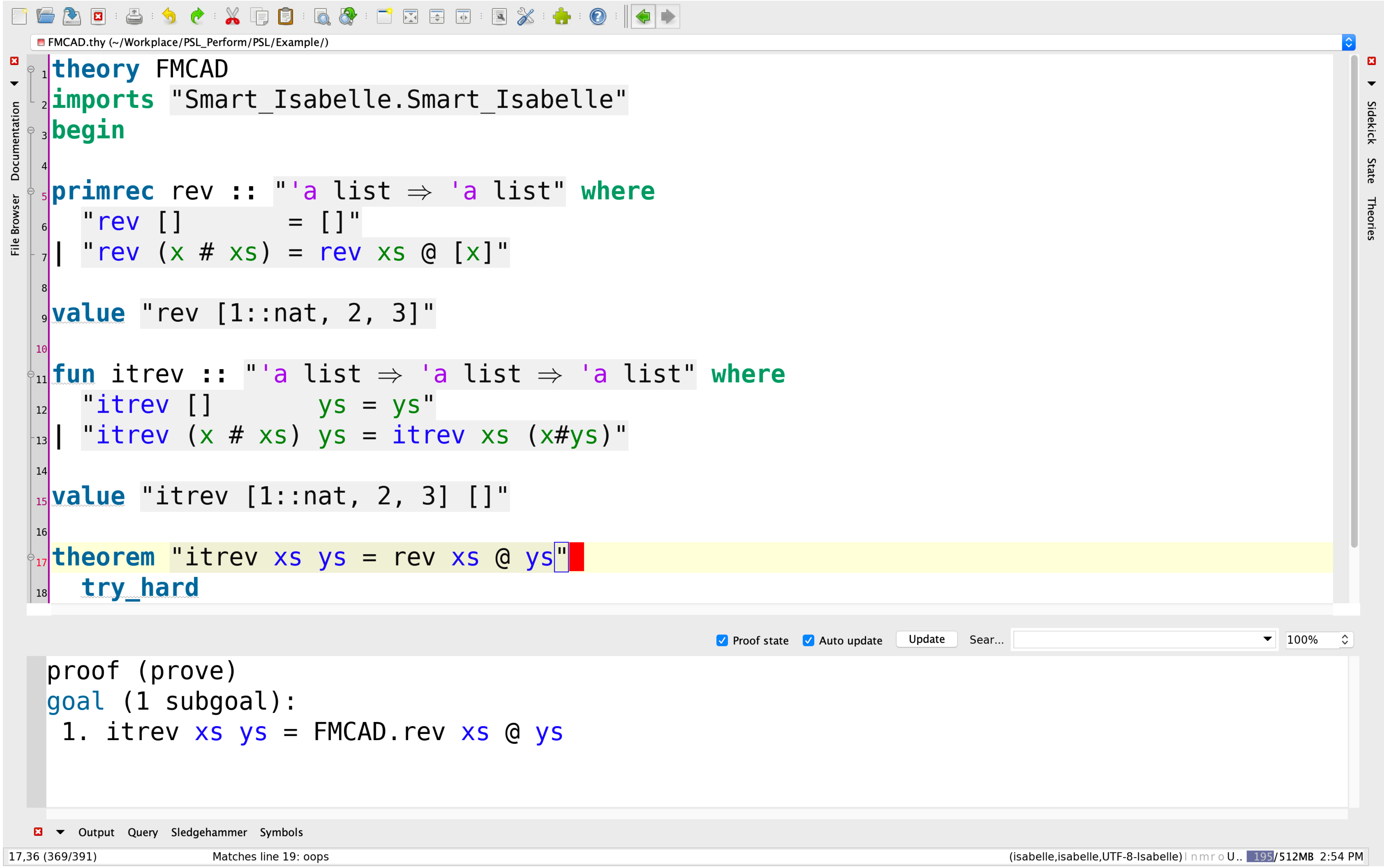
FMCAD.thy (~/Workplace/PSL_Perform/PSL/Example/)

```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys) apply auto done
```

☑ Proof state ☑ Auto update Update Sear... 100%

```
proof (prove)
goal:
No subgoals!
```

18,44 (413/423) (isabelle,isabelle,UTF-8-Isabelle) | nmro U.. 77/512MB 12:22 PM

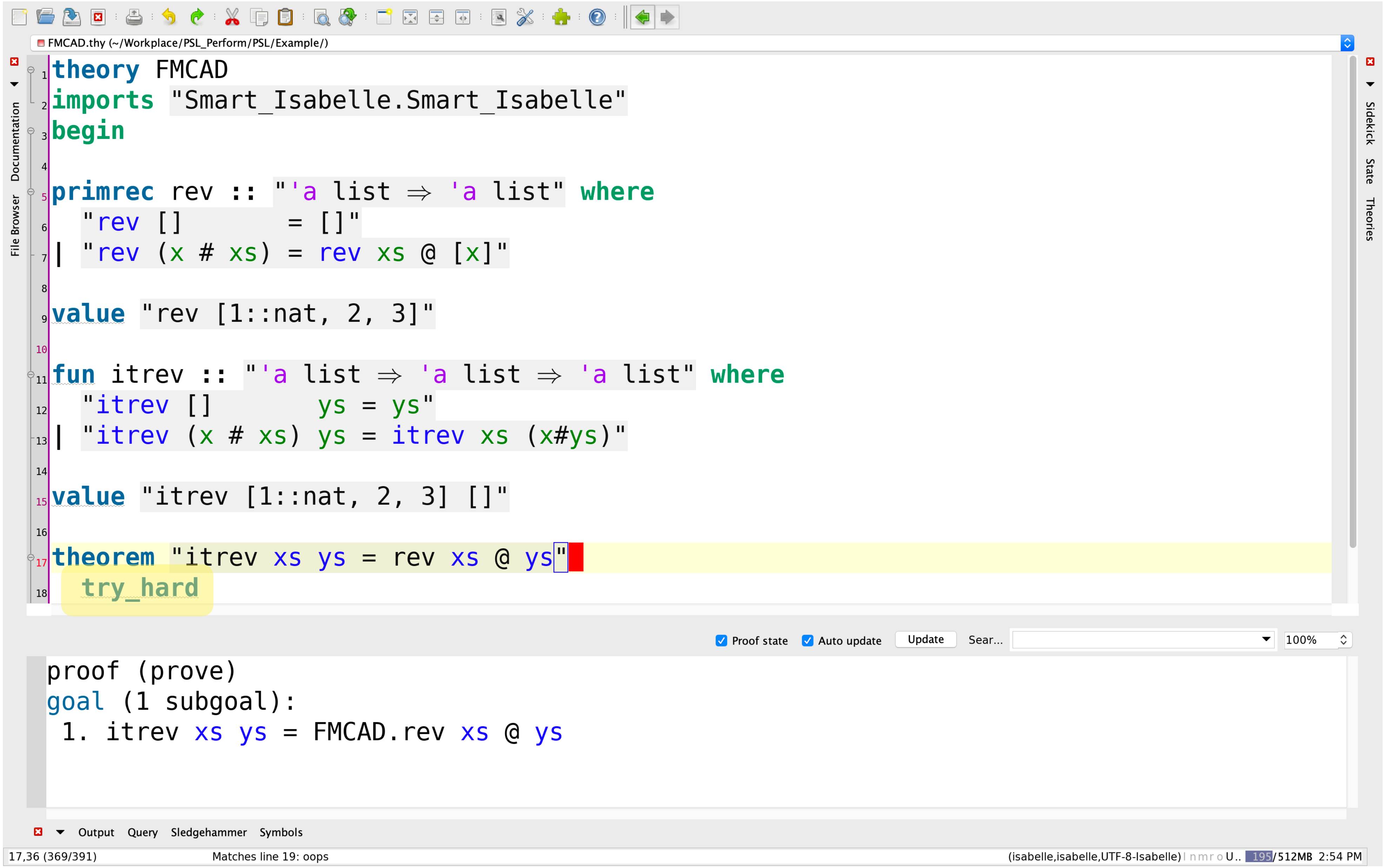


```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"
  try_hard
```

☑ Proof state ☑ Auto update

```
proof (prove)
goal (1 subgoal):
 1. itrev xs ys = FMCAD.rev xs @ ys
```

17,36 (369/391) Matches line 19: oops (isabelle,isabelle,UTF-8-Isabelle) nmro U.. 195/512MB 2:54 PM

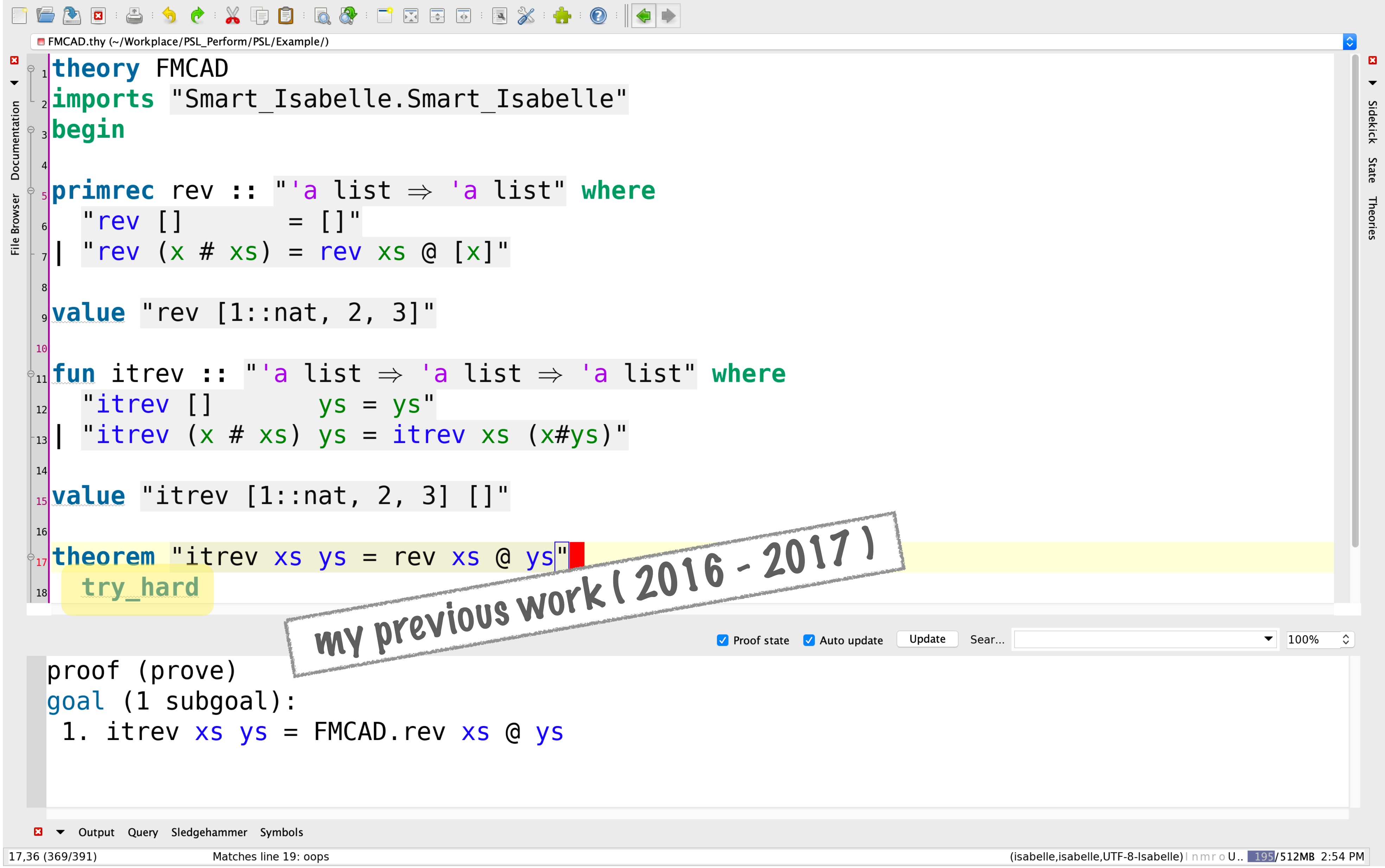


```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"
  try_hard
```

☑ Proof state ☑ Auto update

```
proof (prove)
goal (1 subgoal):
 1. itrev xs ys = FMCAD.rev xs @ ys
```

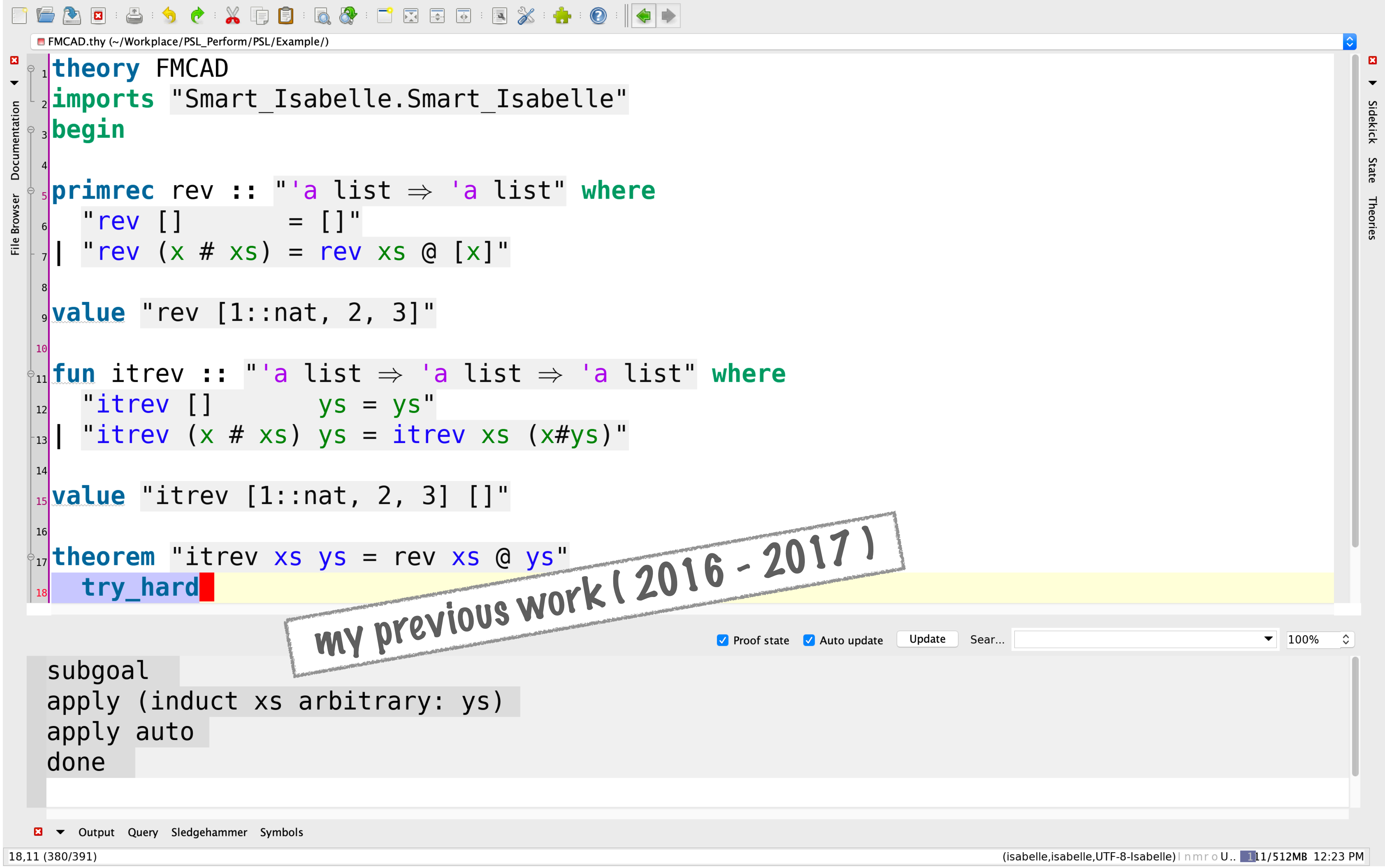
17,36 (369/391) Matches line 19: oops (isabelle,isabelle,UTF-8-Isabelle) | nmro U.. 195/512MB 2:54 PM



```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"
  try_hard
```

```
proof (prove)
goal (1 subgoal):
 1. itrev xs ys = FMCAD.rev xs @ ys
```

my previous work (2016 - 2017)



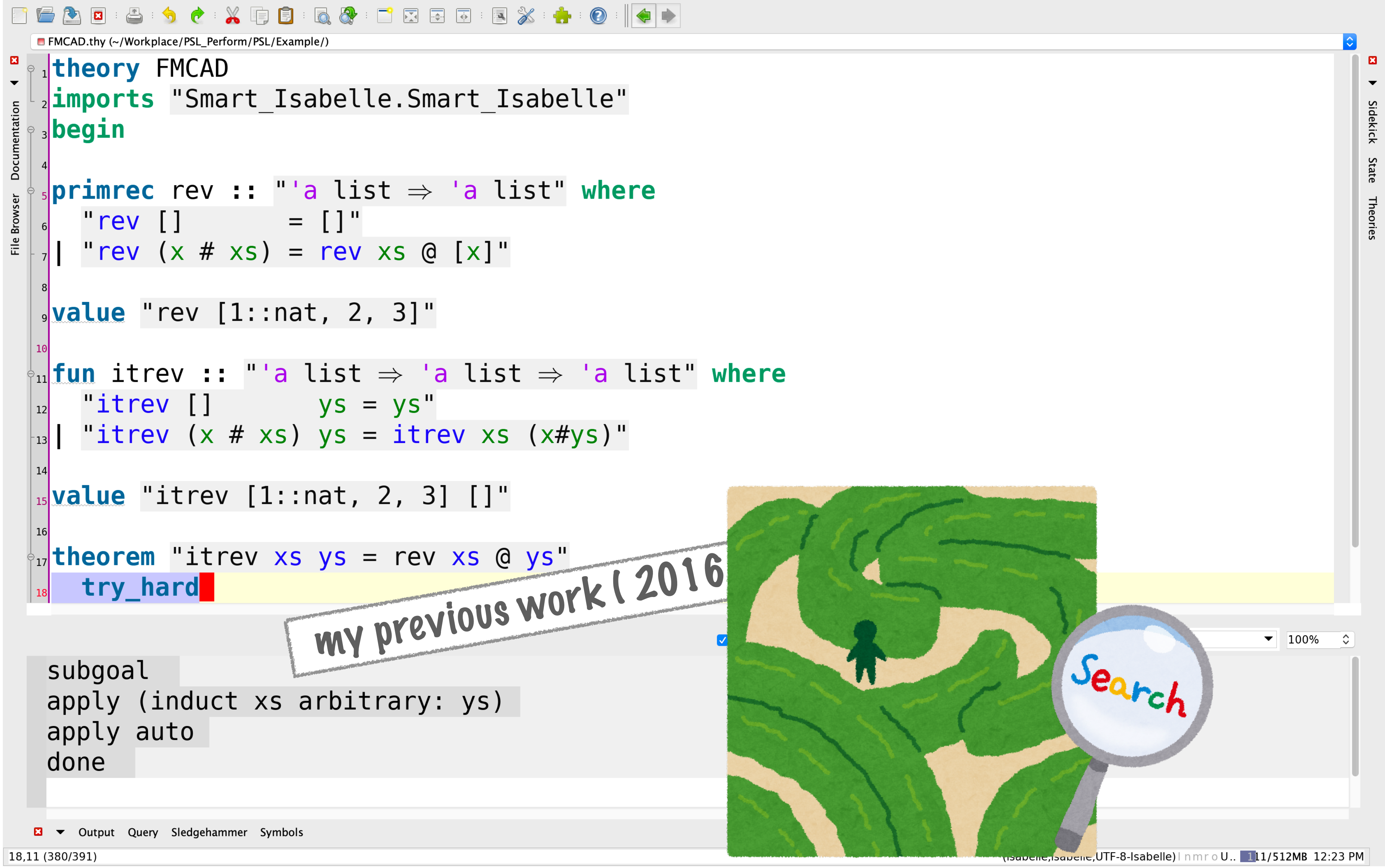
FMCAD.thy (~/Workplace/PSL_Perform/PSL/Example/)

```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"
  try_hard
```

☑ Proof state ☑ Auto update

```
subgoal
apply (induct xs arbitrary: ys)
apply auto
done
```

my previous work (2016 - 2017)



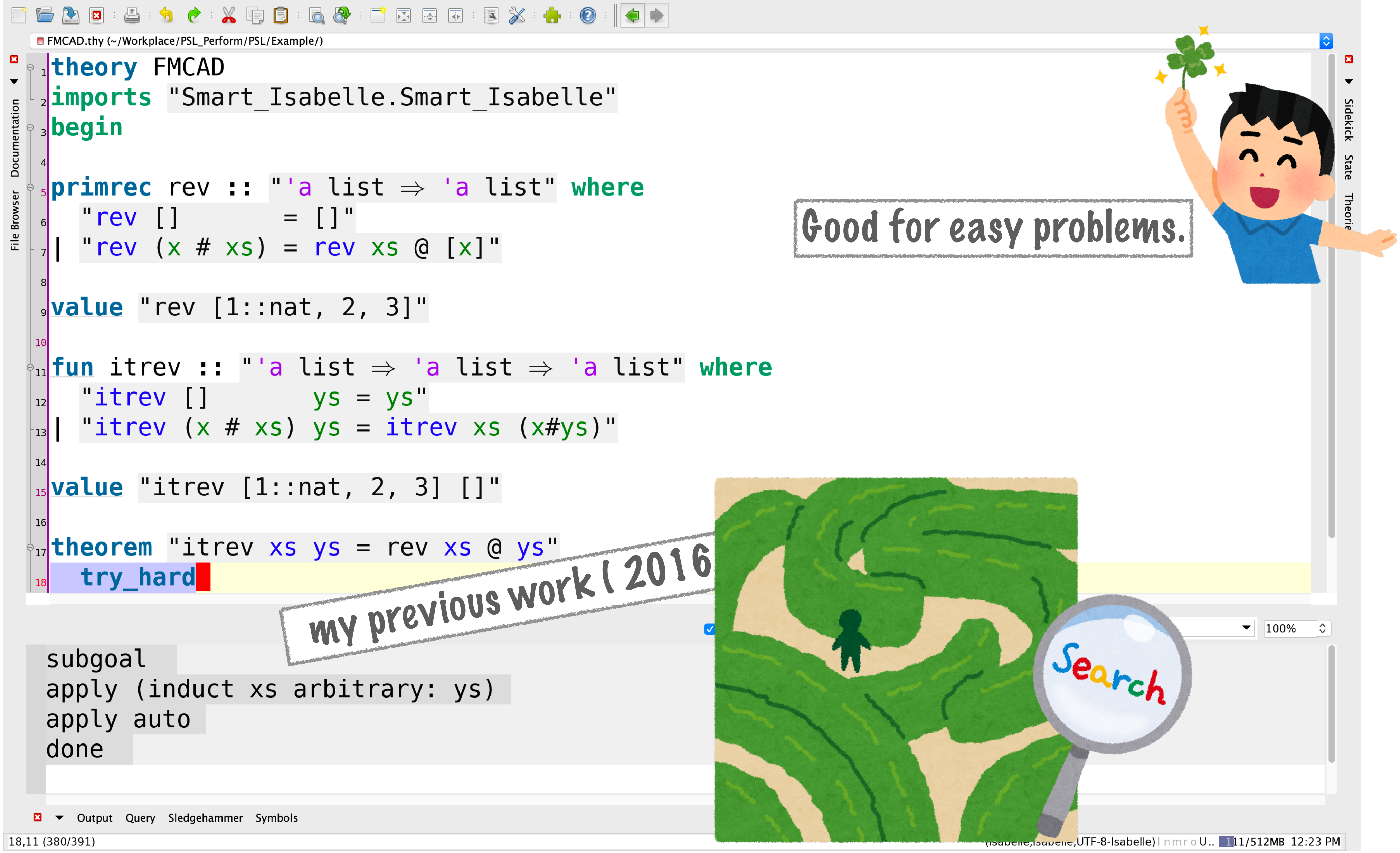
```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"
  try_hard
```

```
subgoal
apply (induct xs arbitrary: ys)
apply auto
done
```

18,11 (380/391)

(Isabelle,isabelle,UTF-8-Isabelle) nmro U.. 111/512MB 12:23 PM

my previous work (2016

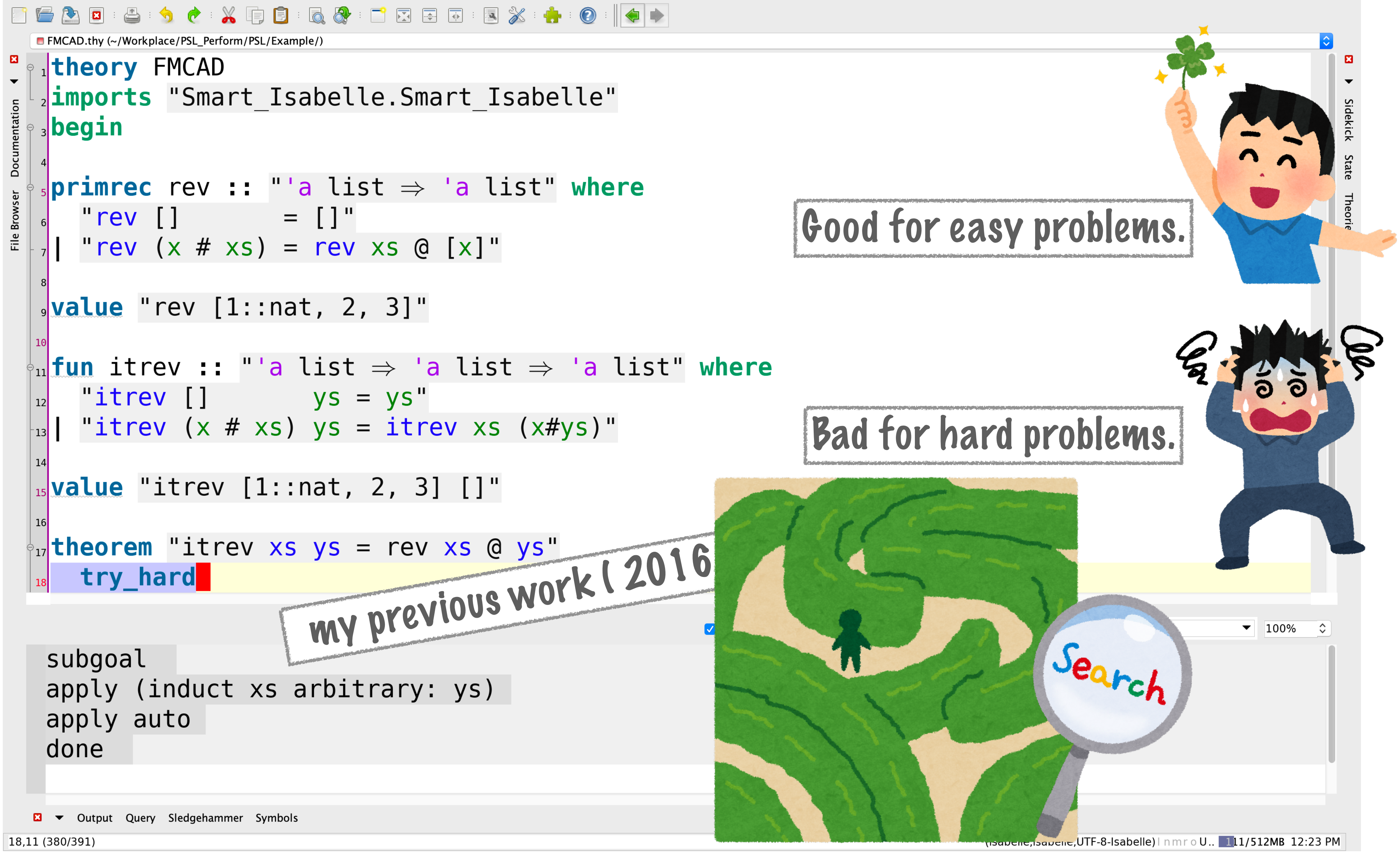

FMCAD.thy (~/Workplace/PSL_Perform/PSL/Example/)
Documentation
File Browser
Sidekick
State
Theorie
100%
Output
Query
Sledgehammer
Symbols
18,11 (380/391)
(Isabelle,isabelle,UTF-8-Isabelle) | nmro U.. 111/512MB 12:23 PM
Search

```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"
  try_hard
```

```
subgoal
apply (induct xs arbitrary: ys)
apply auto
done
```

Good for easy problems.

my previous work (2016


FMCAD.thy (~/Workplace/PSL_Perform/PSL/Example/)
Documentation
File Browser
Sidekick
State
Theories
100%
Output
Query
Sledgehammer
Symbols
18,11 (380/391)
(Isabelle,isabelle,UTF-8-Isabelle) nmro U.. 111/512MB 12:23 PM
Search

```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"
  try_hard
```

```
subgoal
apply (induct xs arbitrary: ys)
apply auto
done
```

my previous work (2016)

Good for easy problems.

Bad for hard problems.

Good news for automation.

Bad news for automation.

Good news for automation.

(For most cases) we only have to pass the right arguments to the induction tactic.

Bad news for automation.

Good news for automation.

(For most cases) we only have to pass the right arguments to the induction tactic.

Bad news for automation.

Names do not matter globally. Structures matter.

Good news for automation.

(For most cases) we only have to pass the right arguments to the induction tactic.

Bad news for automation.

Names do not matter globally. Structures matter.

All theorems must be different.

Good news for automation.

(For most cases) we only have to pass the right arguments to the induction tactic.

Bad news for automation.

Names do not matter globally. Structures matter.

All theorems must be different.

We should not have many similar theorems.

Good news for automation.

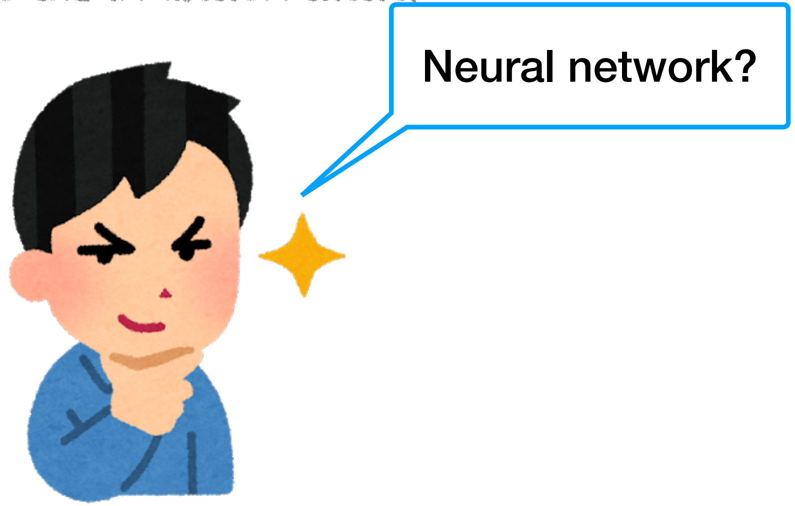
(For most cases) we only have to pass the right arguments to the induction tactic.

Bad news for automation.

Names do not matter globally. Structures matter.

All theorems must be different.

We should not have many similar theorems.



Good news for automation.

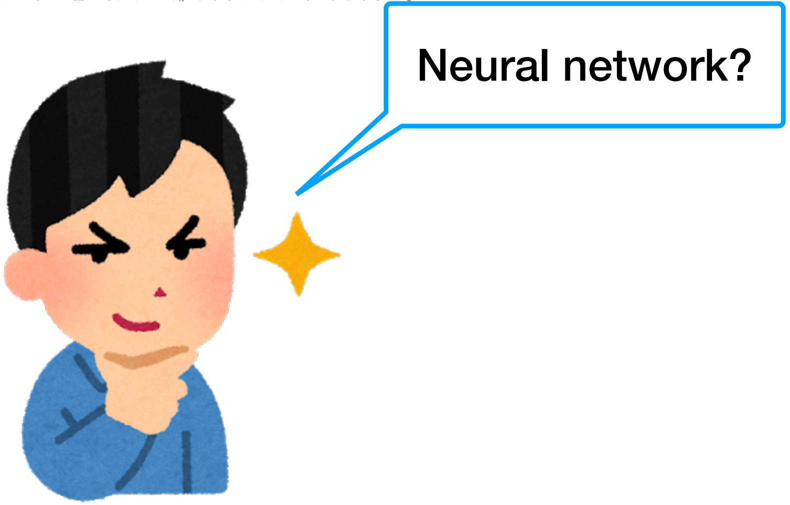
(For most cases) we only have to pass the right arguments to the induction tactic.

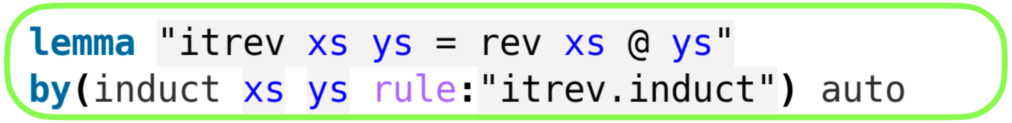
Bad news for automation.

Names do not matter globally. Structures matter.

All theorems must be different.

We should not have many similar theorems.





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

Good news for automation.

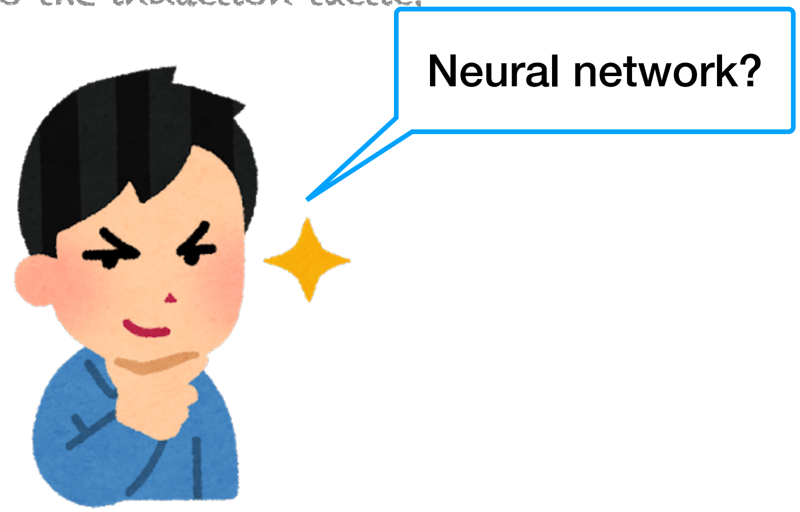
(For most cases) we only have to pass the right arguments to the induction tactic.

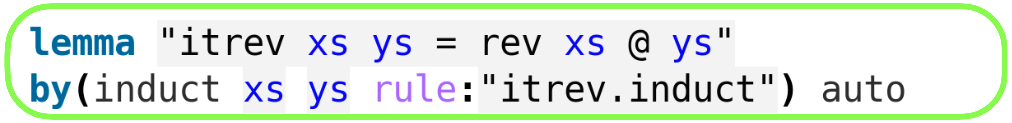
Bad news for automation.

Names do not matter globally. Structures matter.

All theorems must be different.

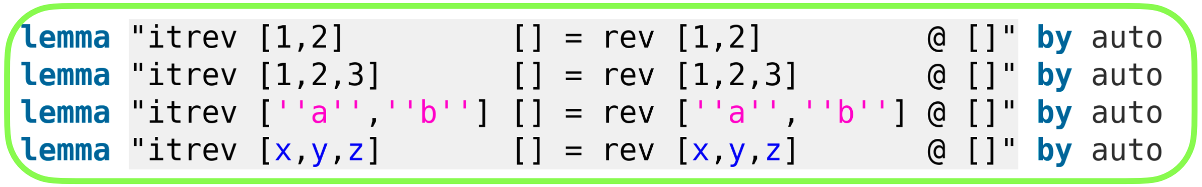
We should not have many similar theorems.





```
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

Good news for automation.

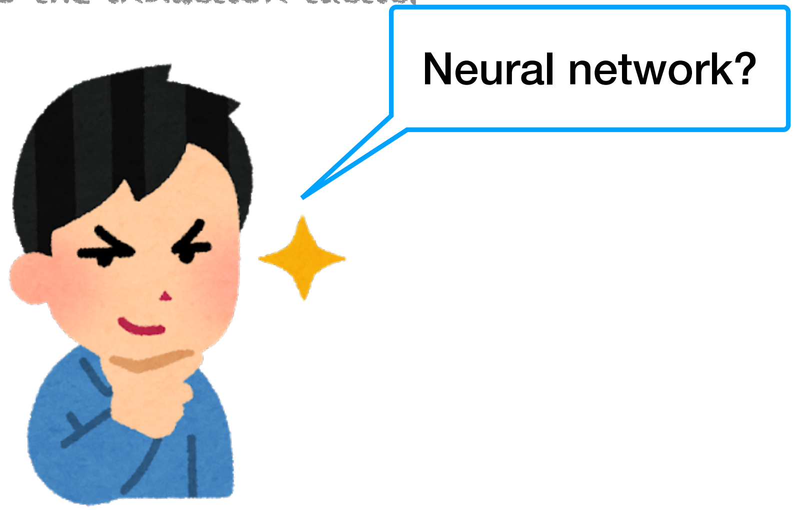
(For most cases) we only have to pass the right arguments to the induction tactic.

Bad news for automation.

Names do not matter globally. Structures matter.

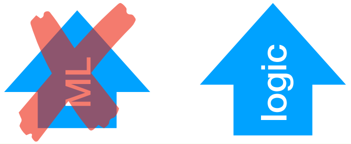
All theorems must be different.

We should not have many similar theorems.



```
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

Good news for automation.

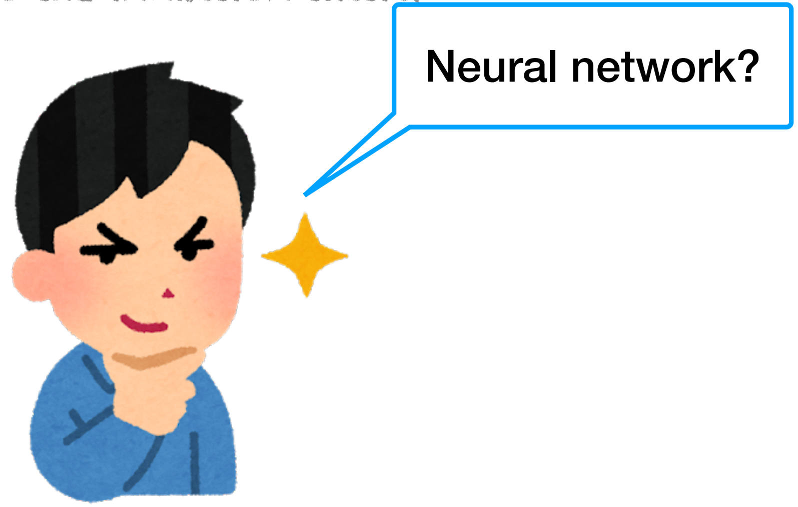
(For most cases) we only have to pass the right arguments to the induction tactic.

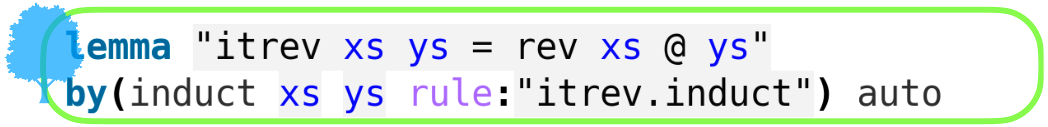
Bad news for automation.

Names do not matter globally. Structures matter.

All theorems must be different.

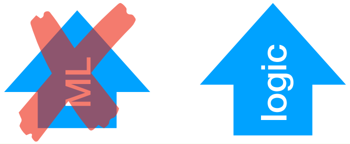
We should not have many similar theorems.



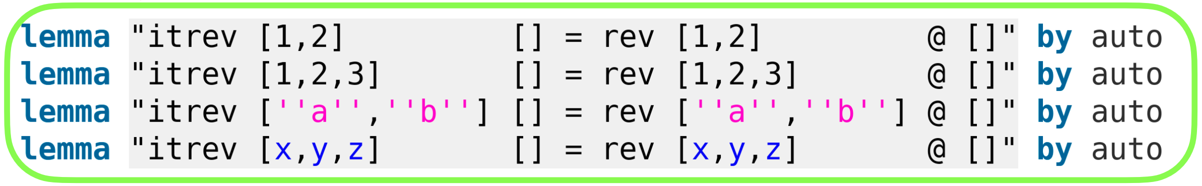


```
lemma "itrev xs ys = rev xs @ ys"
by(induct xs ys rule:"itrev.induct") auto
```

<- one abstract representation

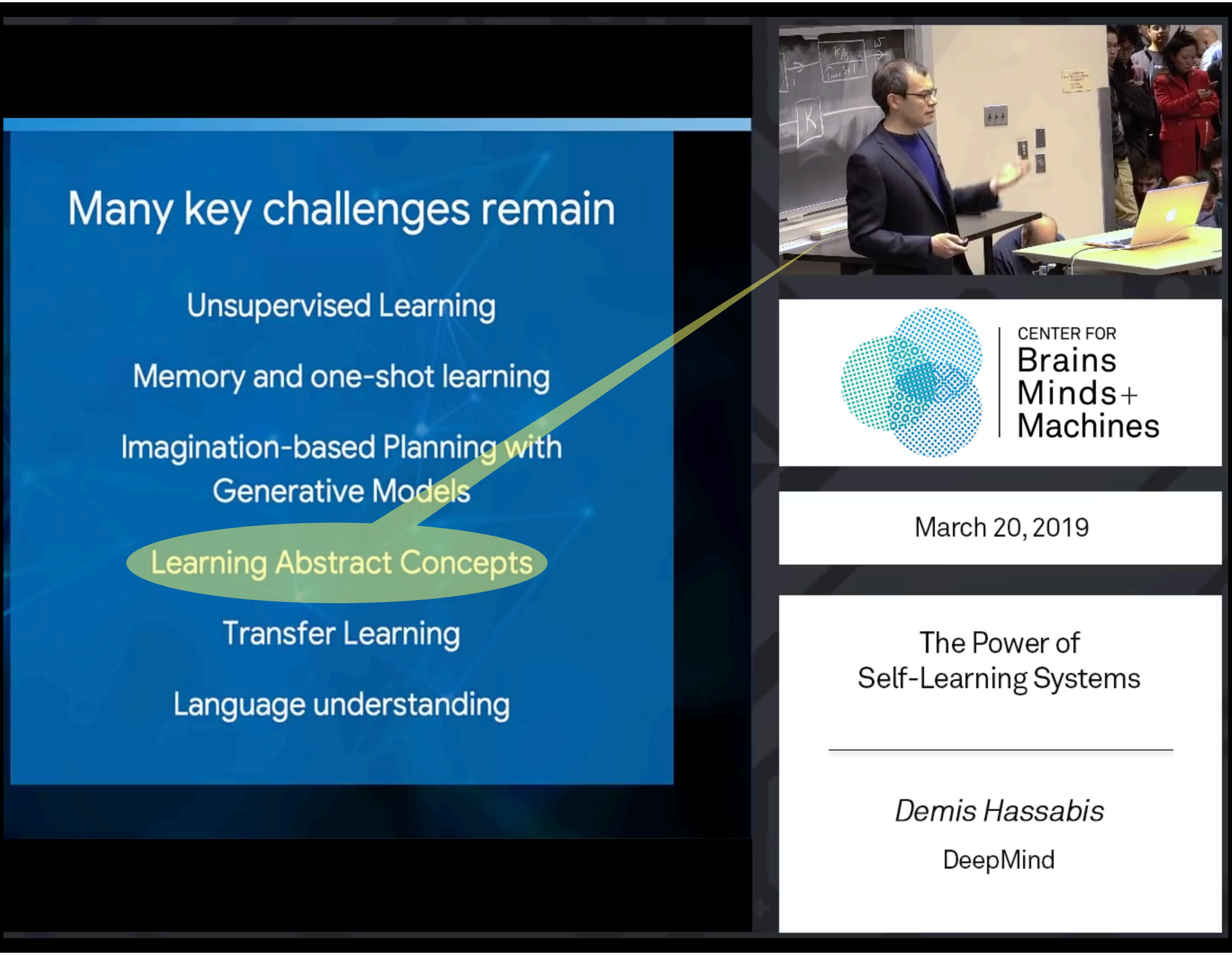


<- abstraction using expressive logic



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

<- many concrete cases



Many key challenges remain

Unsupervised Learning

Memory and one-shot learning

Imagination-based Planning with Generative Models

Learning Abstract Concepts

Transfer Learning

Language understanding

CENTER FOR Brains Minds + Machines

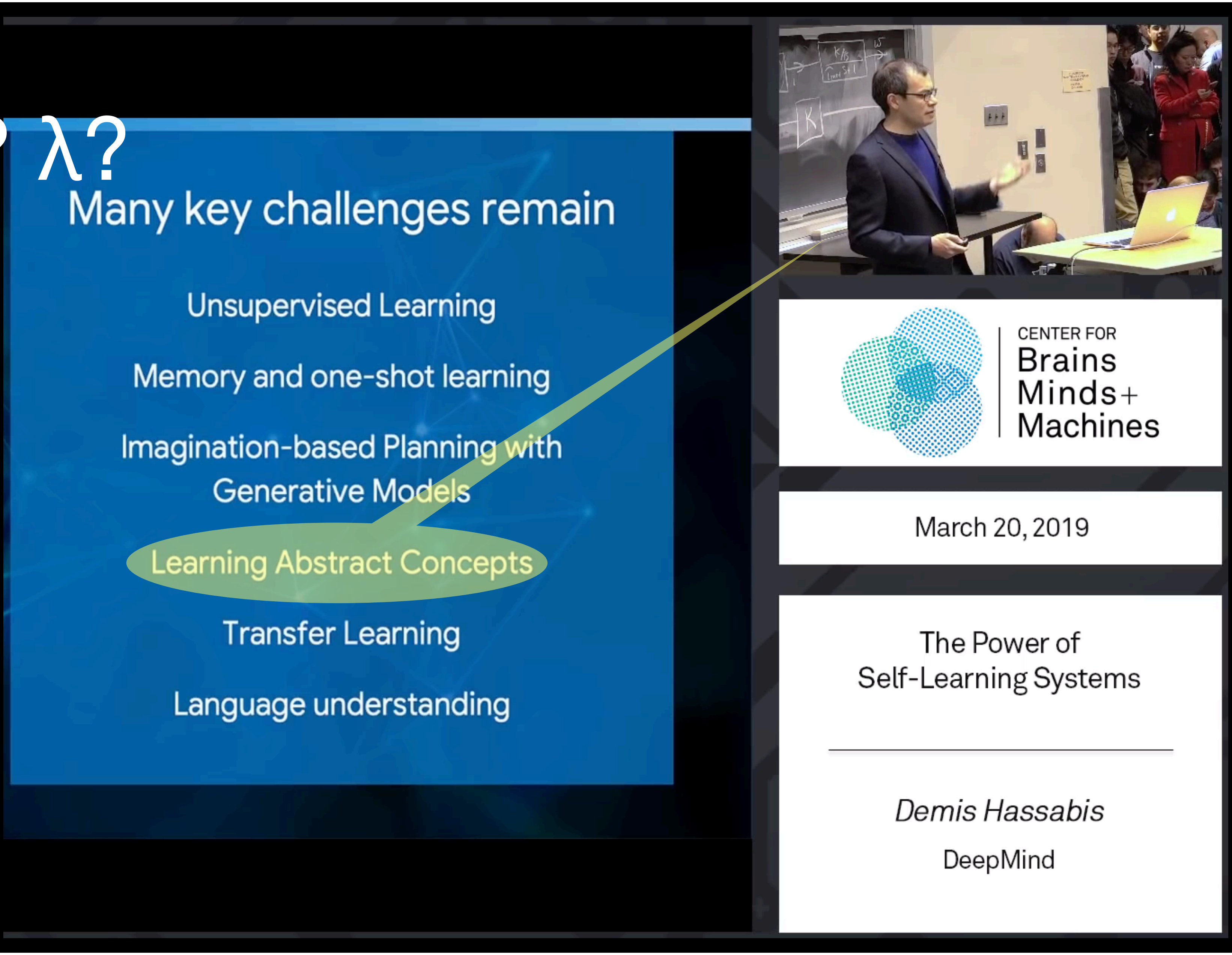
March 20, 2019

The Power of Self-Learning Systems

Demis Hassabis

DeepMind

∀? λ?



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

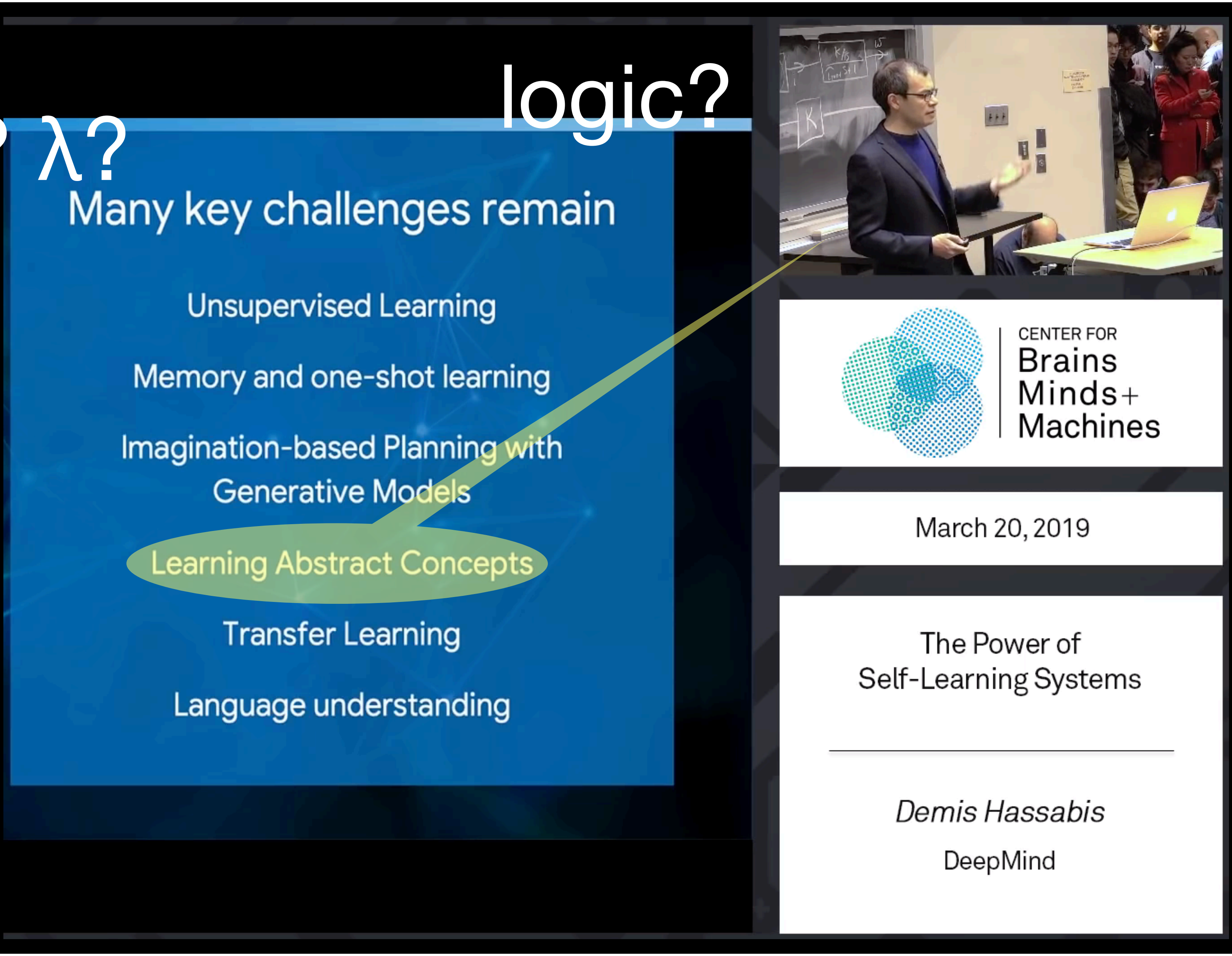
The Power of Self-Learning Systems

Demis Hassabis

DeepMind

logic?

∀? λ?



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

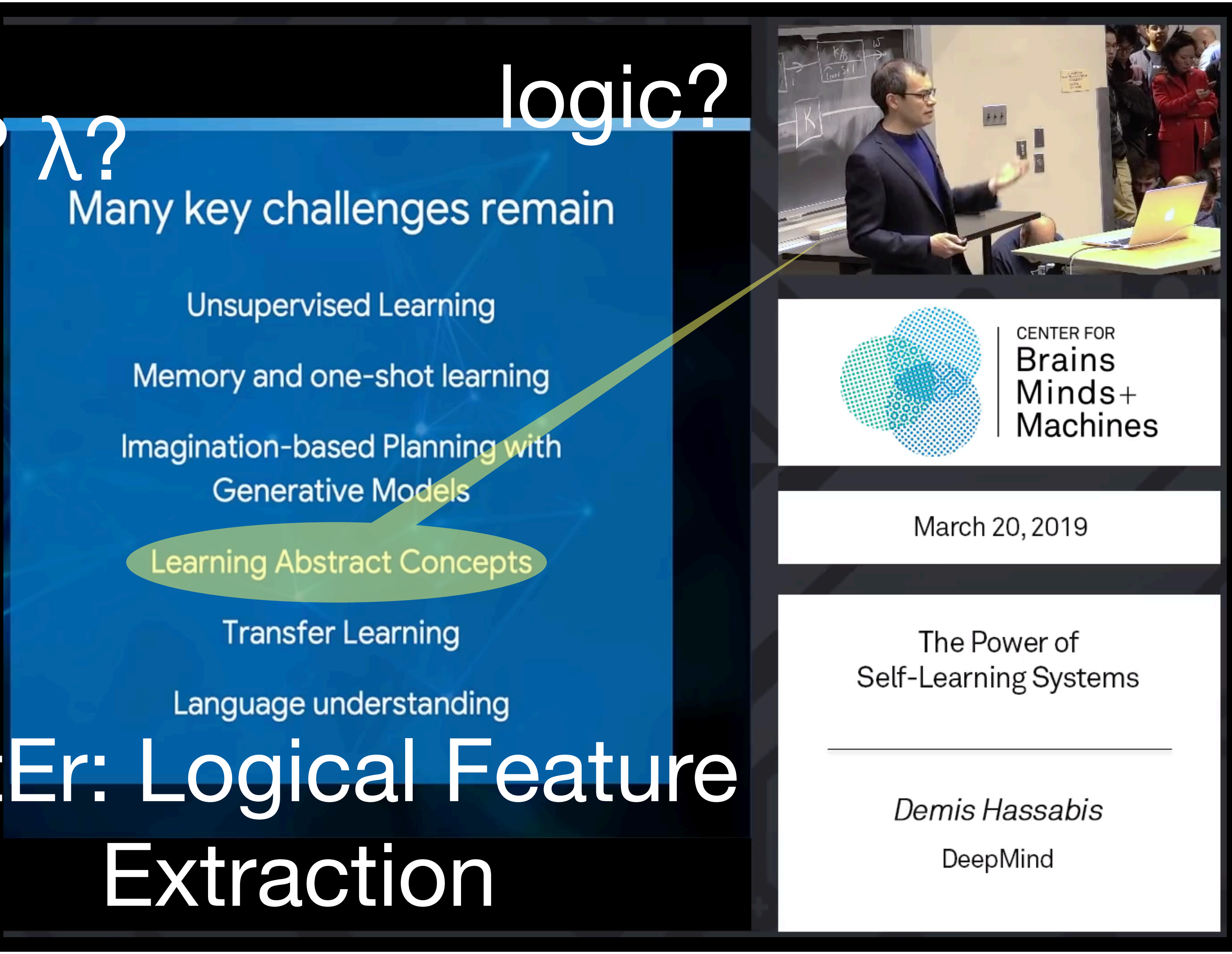
The Power of
Self-Learning Systems

Demis Hassabis

DeepMind

logic?

∀? λ?



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-Learning Systems

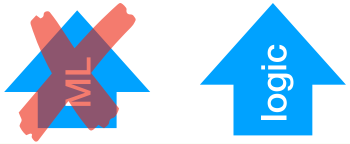
Demis Hassabis

DeepMind

LiFtEr: Logical Feature Extraction

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

<- one abstract representation



<- abstraction using expressive logic

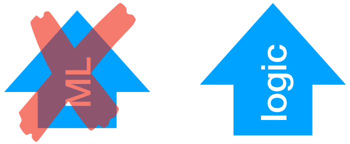
```
lemma "itrev [1,2]         [] = rev [1,2]         @ []" by auto
lemma "itrev [1,2,3]       [] = rev [1,2,3]       @ []" by auto
lemma "itrev [''a'',''b''] [] = rev [''a'',''b''] @ []" by auto
lemma "itrev [x,y,z]       [] = rev [x,y,z]       @ []" by auto
```

<- many concrete cases

Grand Challenge: Abstract Abstraction

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

<- one abstract representation



<- abstraction using expressive logic

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

<- many concrete cases

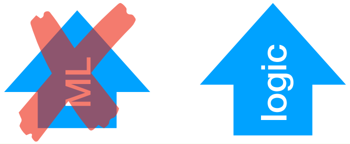
Grand Challenge: Abstract Abstraction

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

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

<- small dataset about different domains

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

<- one abstract representation

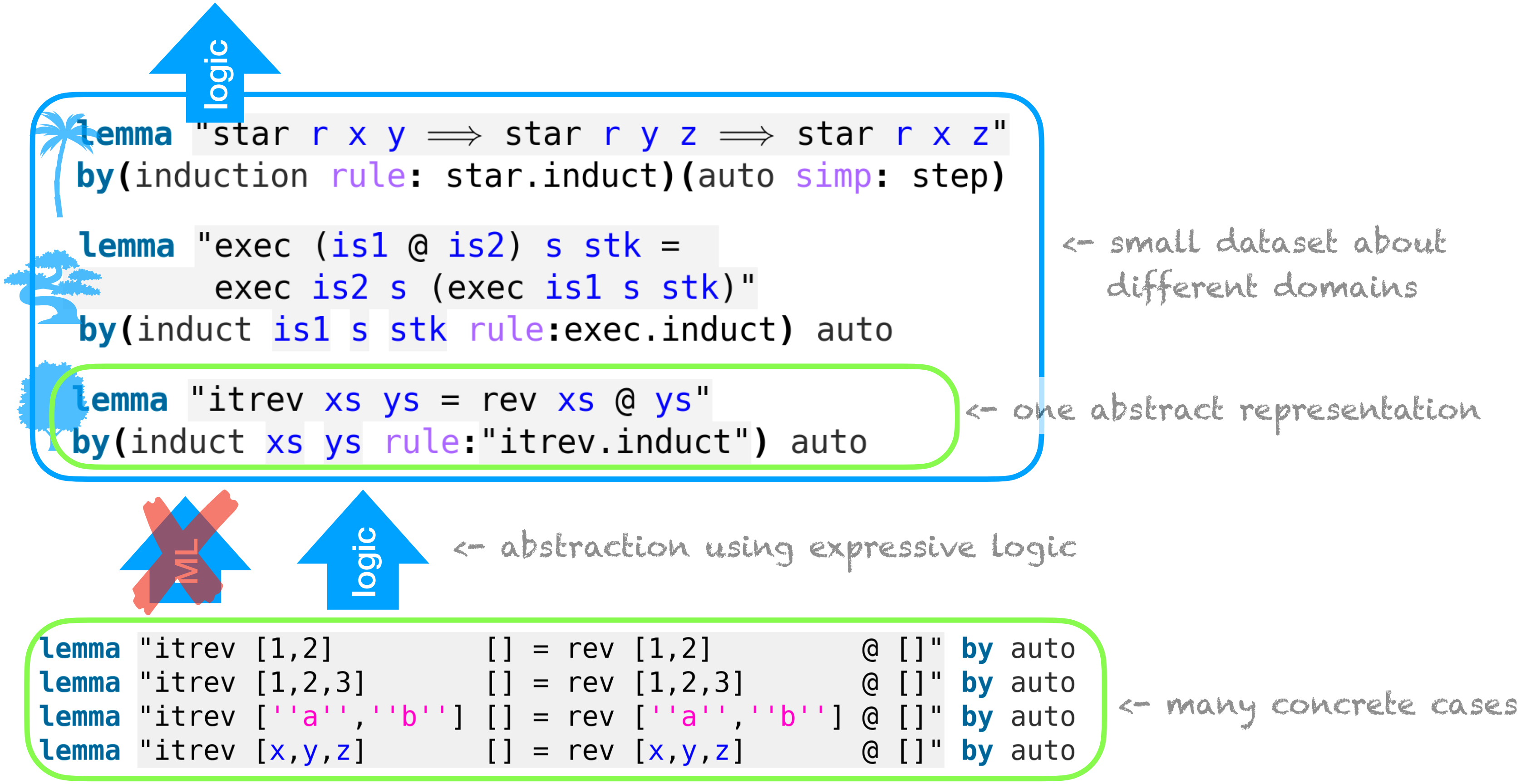


<- abstraction using expressive logic

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

<- many concrete cases

Grand Challenge: Abstract Abstraction



```
lemma "star r x y ⟹ 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

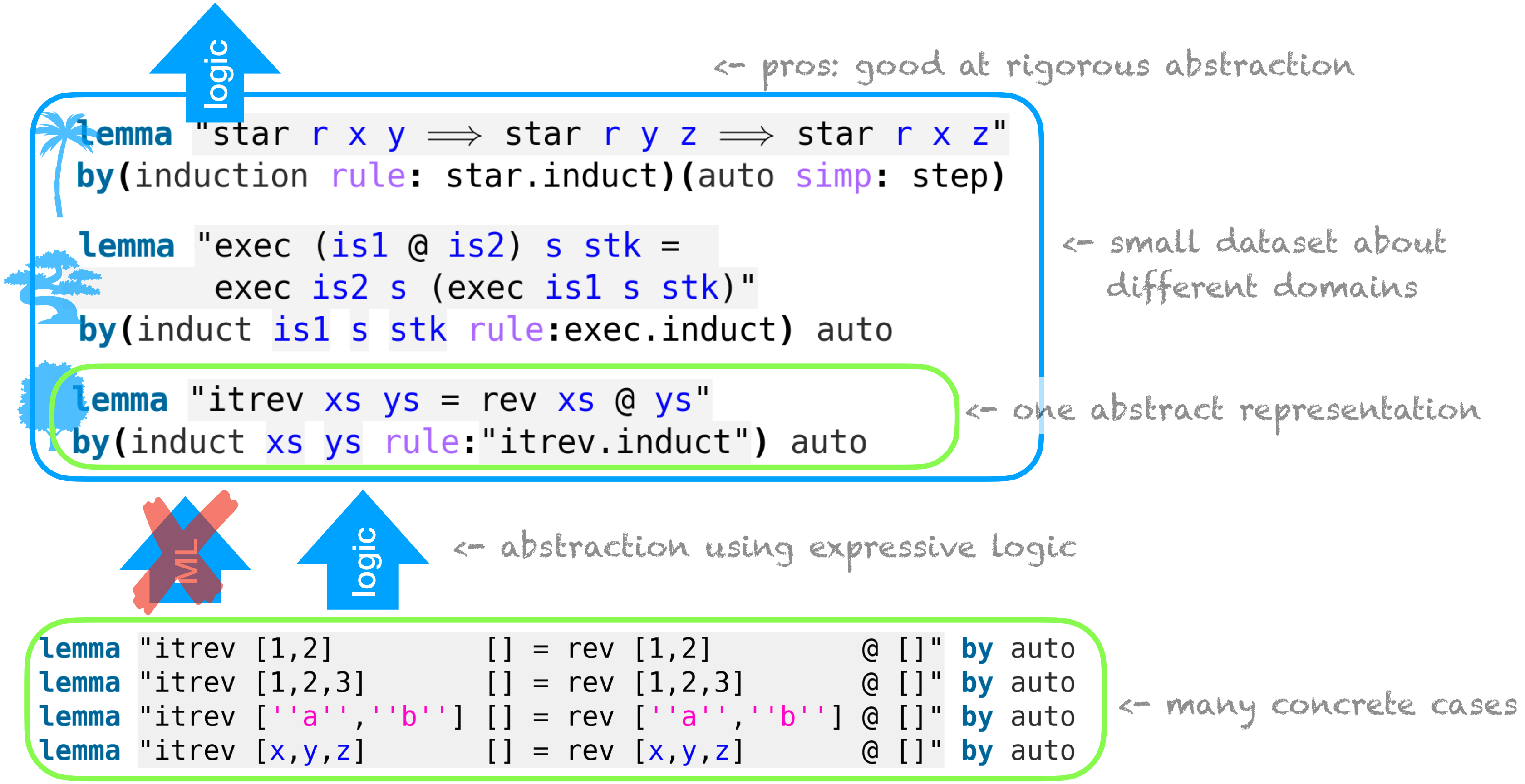
<- one abstract representation

<- 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

Grand Challenge: Abstract Abstraction



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

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

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

<- pros: good at rigorous abstraction

<- small dataset about different domains

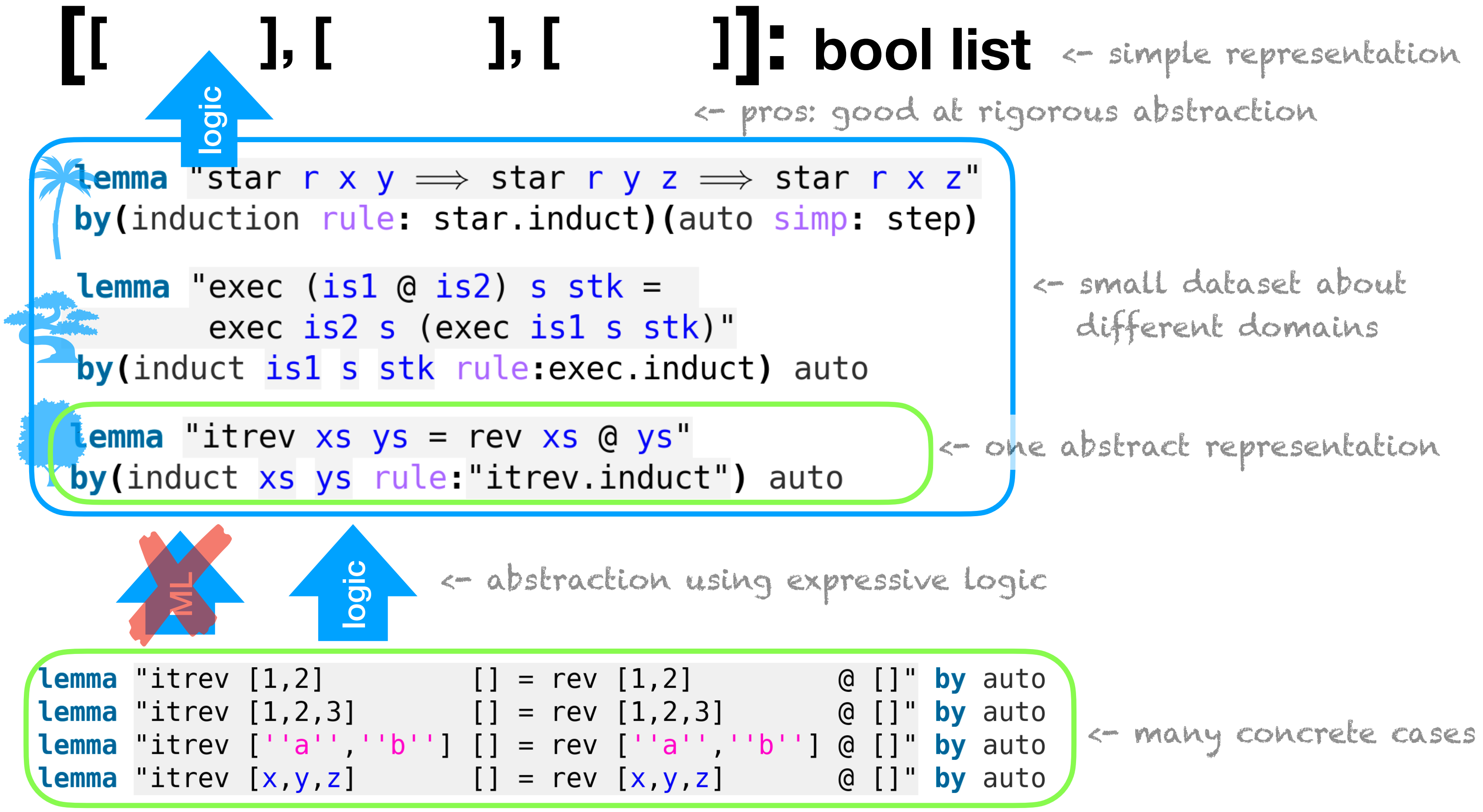
<- one abstract representation

<- 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

Grand Challenge: Abstract Abstraction



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

<- pros: good at rigorous abstraction

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

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

<- small dataset about different domains

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

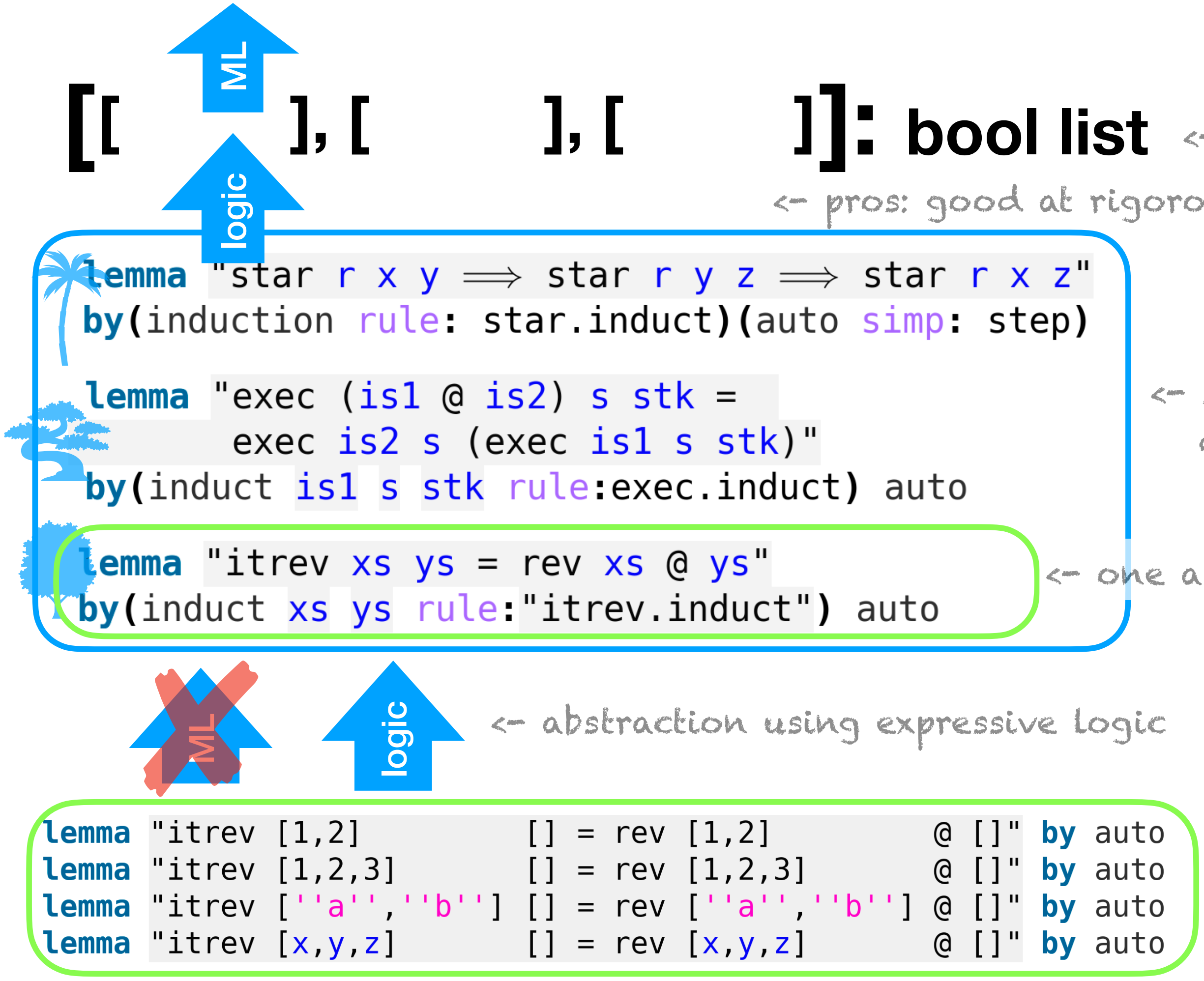
<- one abstract representation

<- abstraction using expressive logic

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

<- many concrete cases

Grand Challenge: Abstract Abstraction



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

<- pros: good at rigorous abstraction

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

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

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

<- small dataset about different domains

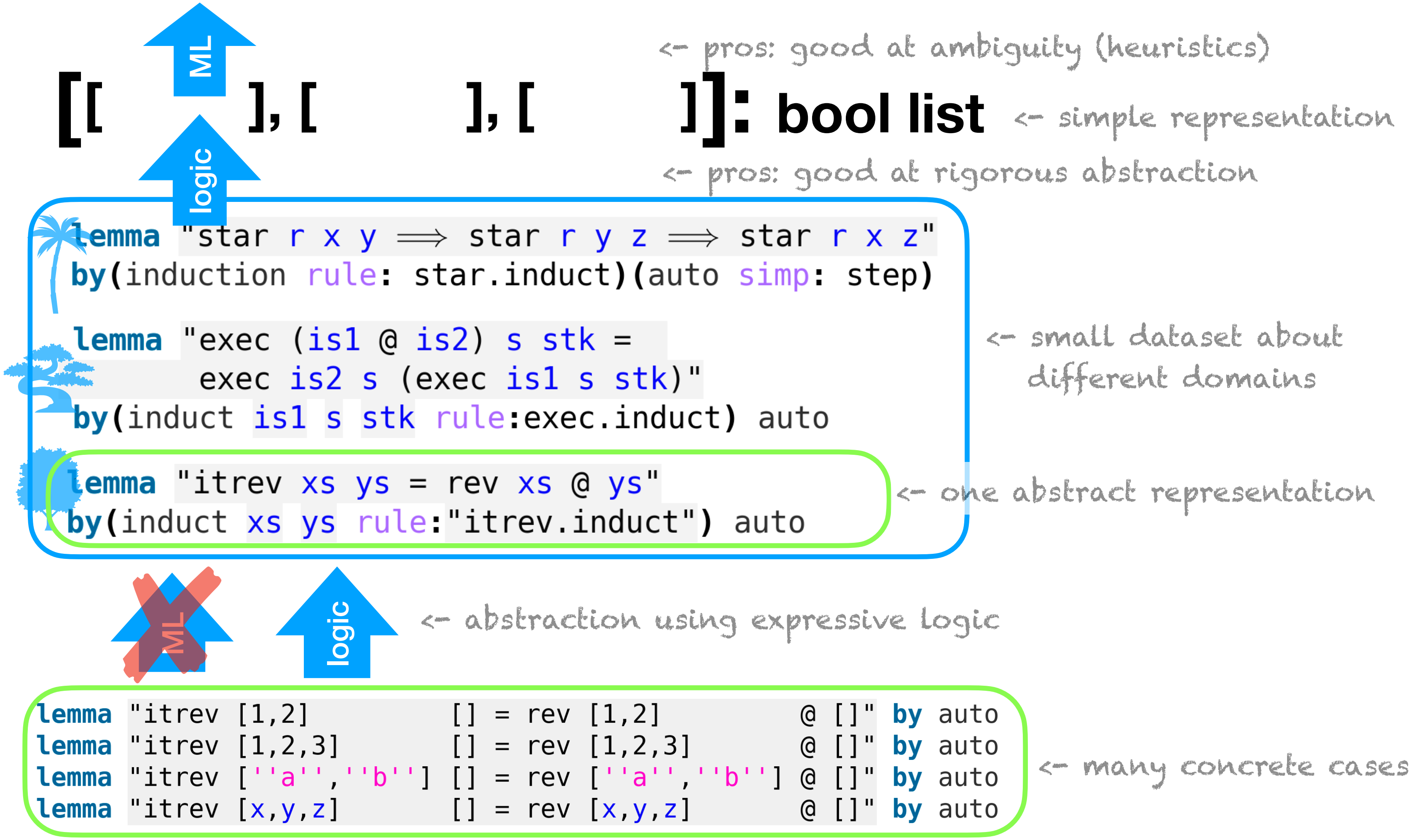
<- one abstract representation

<- abstraction using expressive logic

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

<- many concrete cases

Grand Challenge: Abstract Abstraction



<- pros: good at ambiguity (heuristics)

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

<- pros: good at rigorous abstraction

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

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

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

<- small dataset about different domains

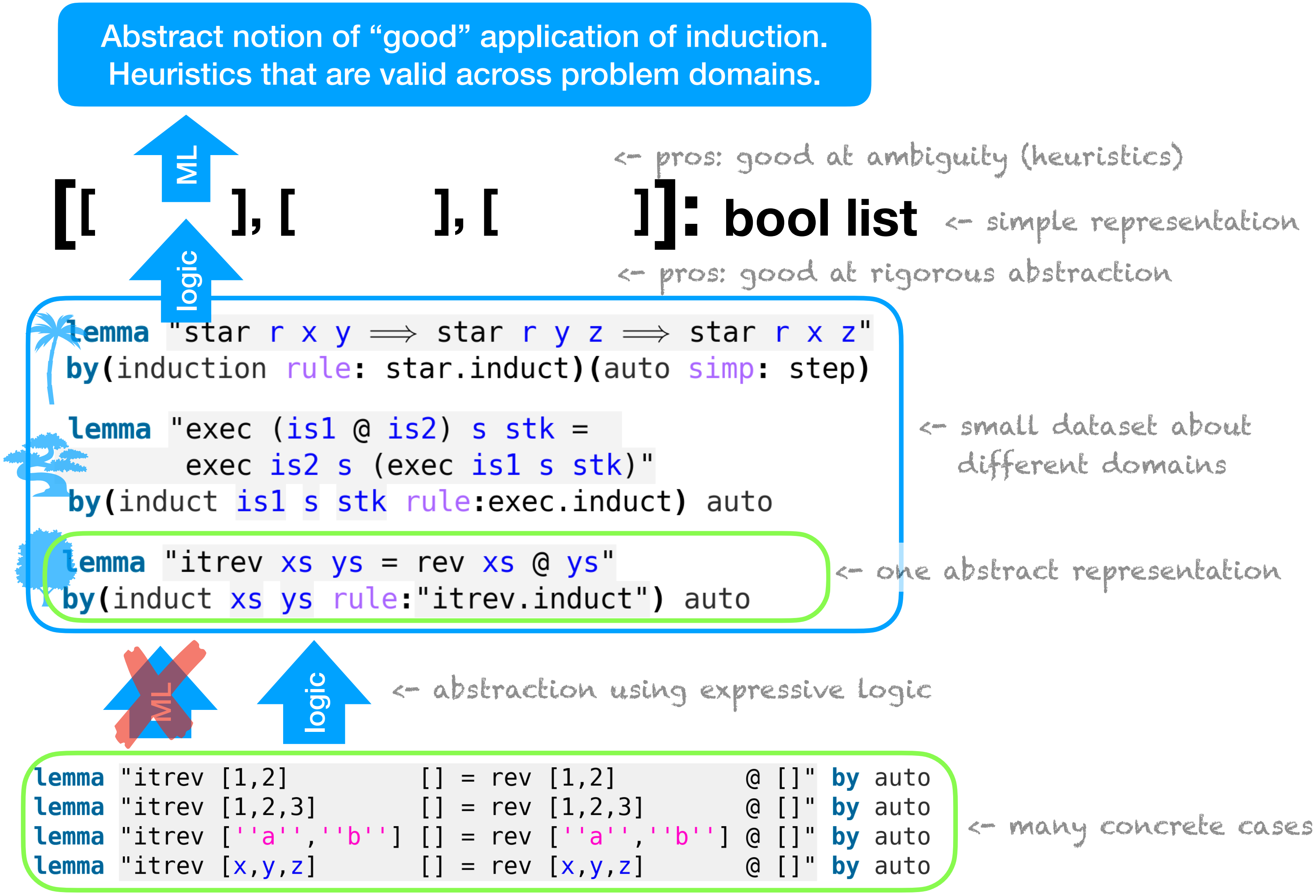
<- one abstract representation

<- abstraction using expressive logic

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

<- many concrete cases

Grand Challenge: Abstract Abstraction



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

<- pros: good at ambiguity (heuristics)

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

<- pros: good at rigorous abstraction

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

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

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

<- small dataset about different domains

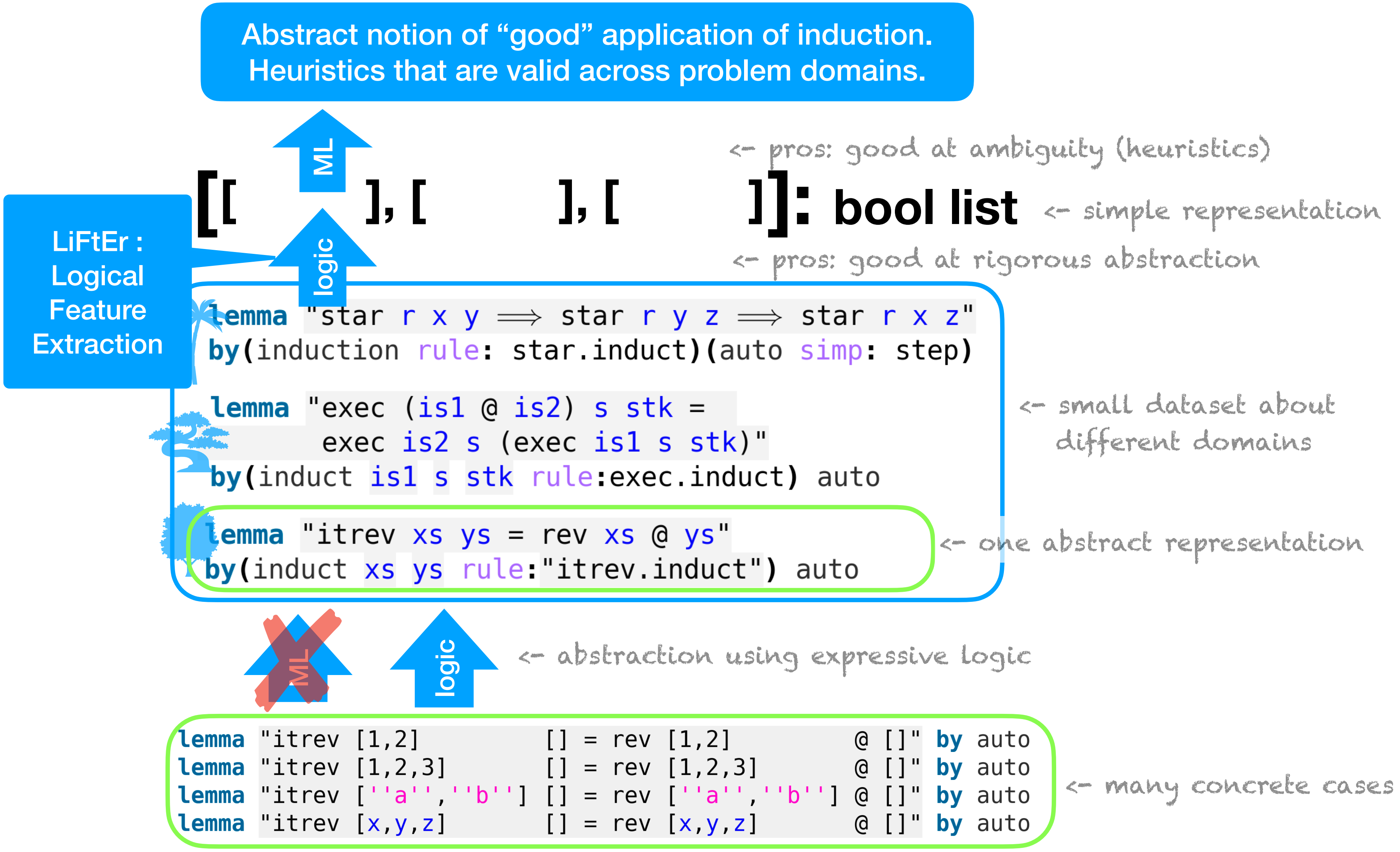
<- one abstract representation

<- abstraction using expressive logic

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

<- many concrete cases

Grand Challenge: Abstract Abstraction



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

<- pros: good at ambiguity (heuristics)

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

<- pros: good at rigorous abstraction

LiFtEr : Logical Feature Extraction

```
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

Example Heuristic in LiFtEr (in Abstract Syntax)

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

Example Heuristic in LiFtEr (in Abstract Syntax)

implication

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

Example Heuristic in LiFtEr (in Abstract Syntax)

implication

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

Example Heuristic in LiFtEr (in Abstract Syntax)

implication

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

Example Heuristic in LiFtEr (in Abstract Syntax)

implication

variable for auxiliary lemmas

variable for terms

conjunction

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

Example Heuristic in LiFtEr (in Abstract Syntax)

implication

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

variable for auxiliary lemmas

variable for terms

variable for term occurrences

conjunction

Example Heuristic in LiFtEr (in Abstract Syntax)

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

implication

variable for auxiliary lemmas

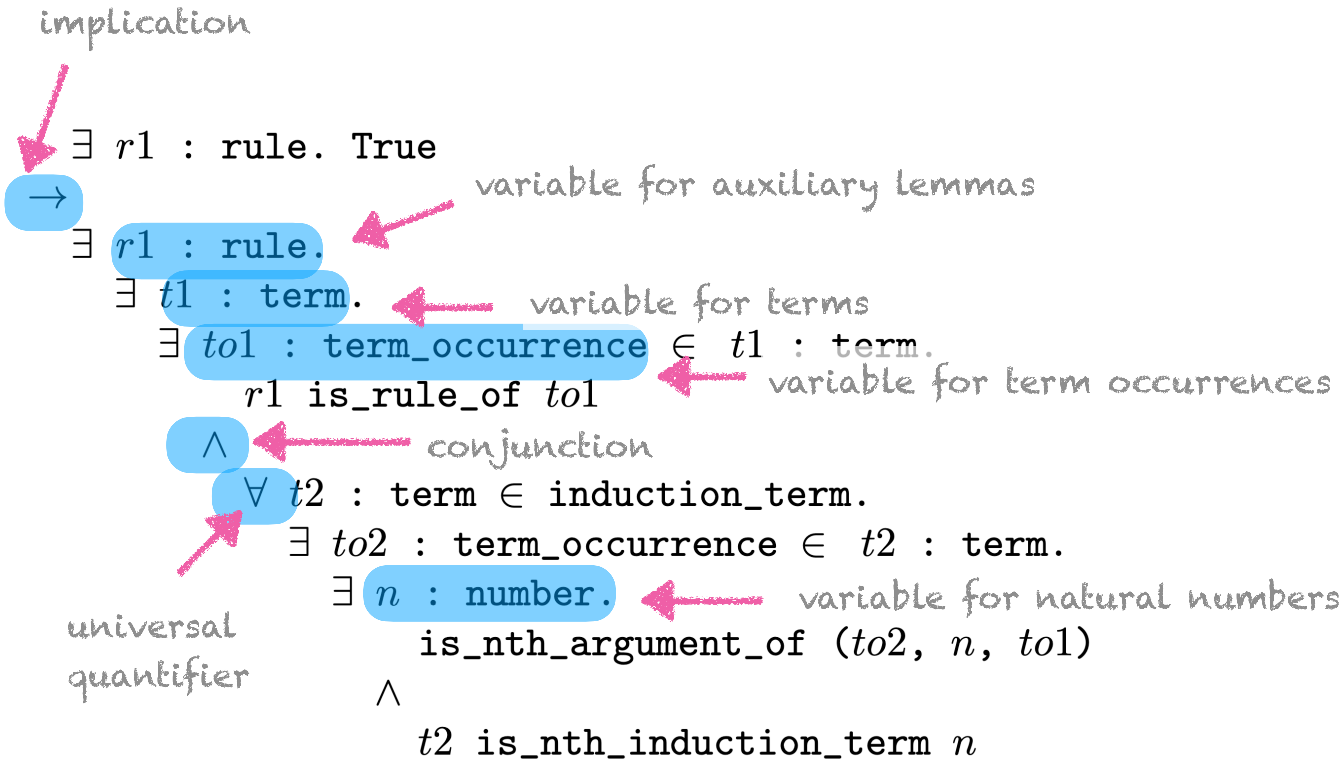
variable for terms

variable for term occurrences

conjunction

variable for natural numbers

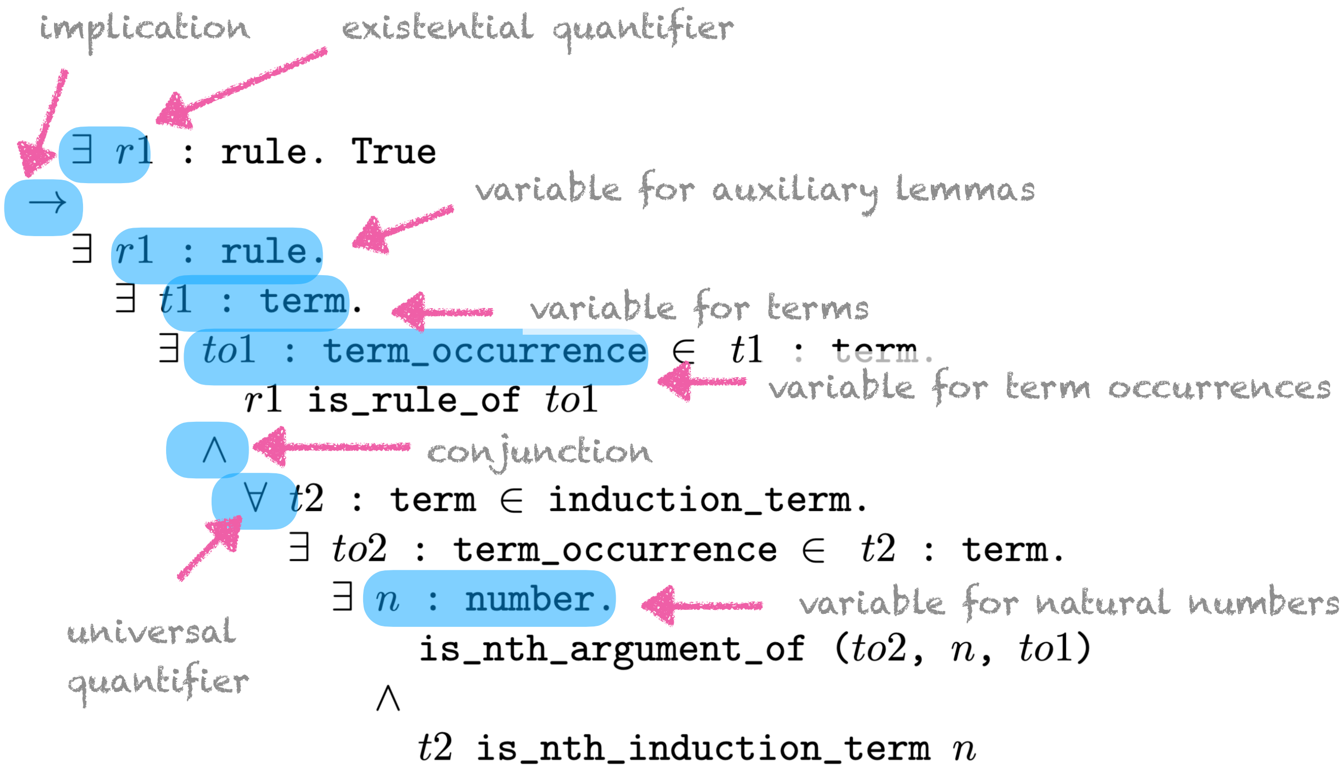
Example Heuristic in LiFtEr (in Abstract Syntax)



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

implication
variable for auxiliary lemmas
variable for terms
variable for term occurrences
conjunction
universal quantifier
variable for natural numbers

Example Heuristic in LiFtEr (in Abstract Syntax)

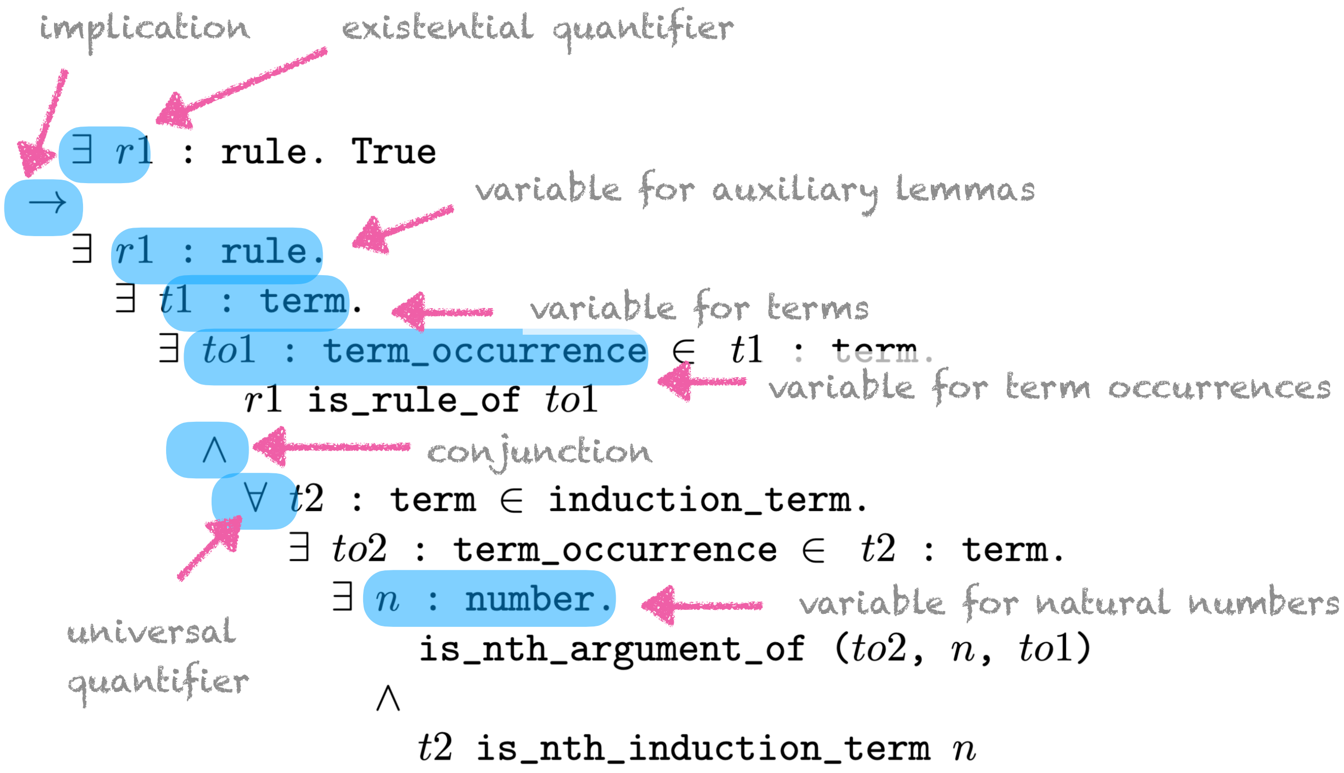


```
∃ r1 : rule. True
→
  ∃ r1 : rule.
    ∃ t1 : term.
      ∃ to1 : term_occurrence ∈ t1 : term.
          r1 is_rule_of to1
      ∧
        ∀ t2 : term ∈ induction_term.
          ∃ to2 : term_occurrence ∈ t2 : term.
            ∃ n : number.
                is_nth_argument_of (to2, n, to1)
     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

implication
existential quantifier
variable for auxiliary lemmas
variable for terms
variable for term occurrences
conjunction
universal quantifier
variable for natural numbers

Example Heuristic in LiFtEr (in Abstract Syntax)

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



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

implication

existential quantifier

variable for auxiliary lemmas

variable for terms

variable for term occurrences

conjunction

universal quantifier

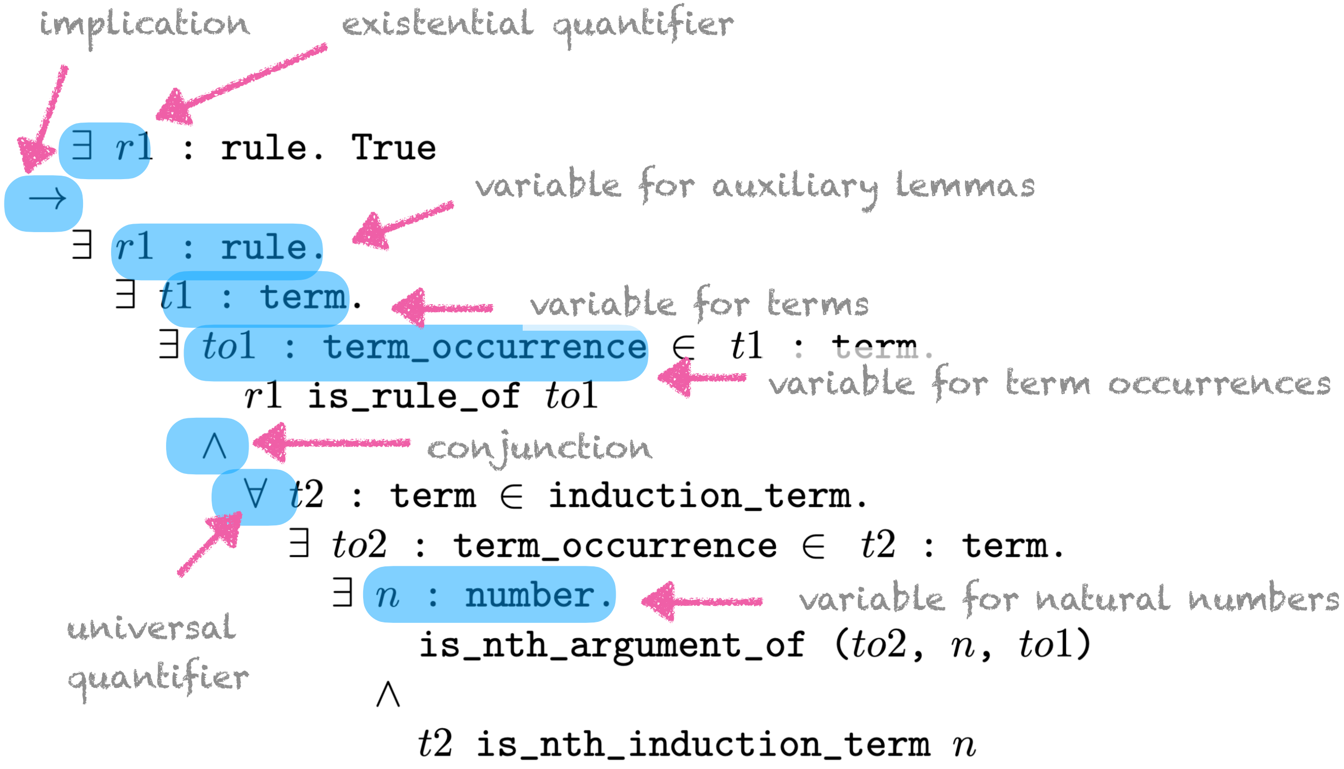
variable for natural numbers

Example Heuristic in LiFtEr (in Abstract Syntax)

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

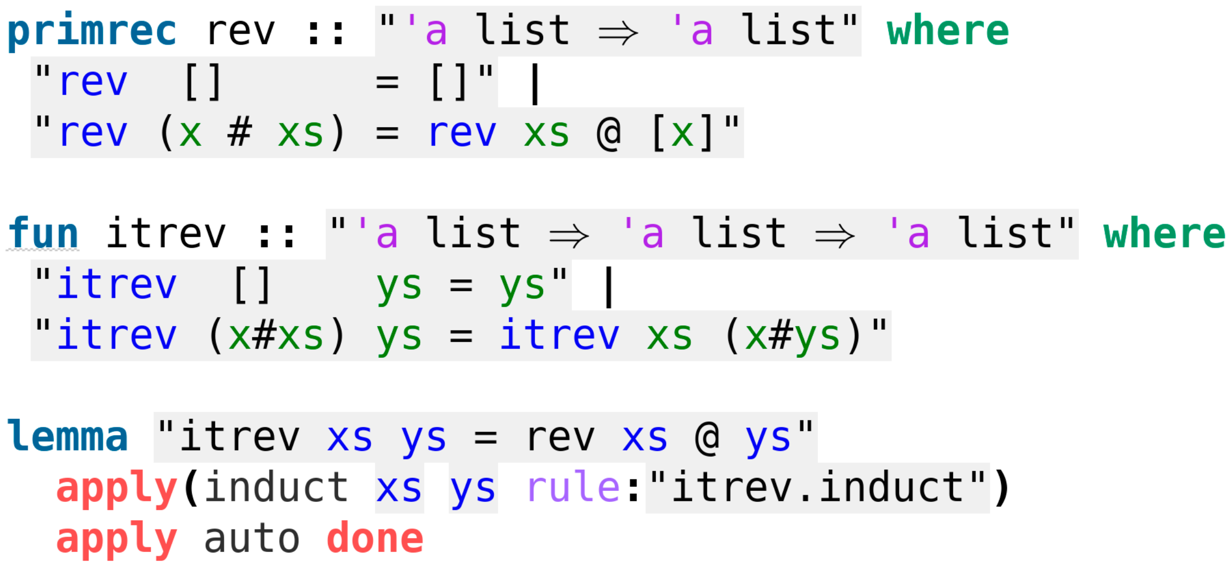
should be <u>true</u> if induction is <u>good</u>

should be <u>false</u> if induction is <u>bad</u>



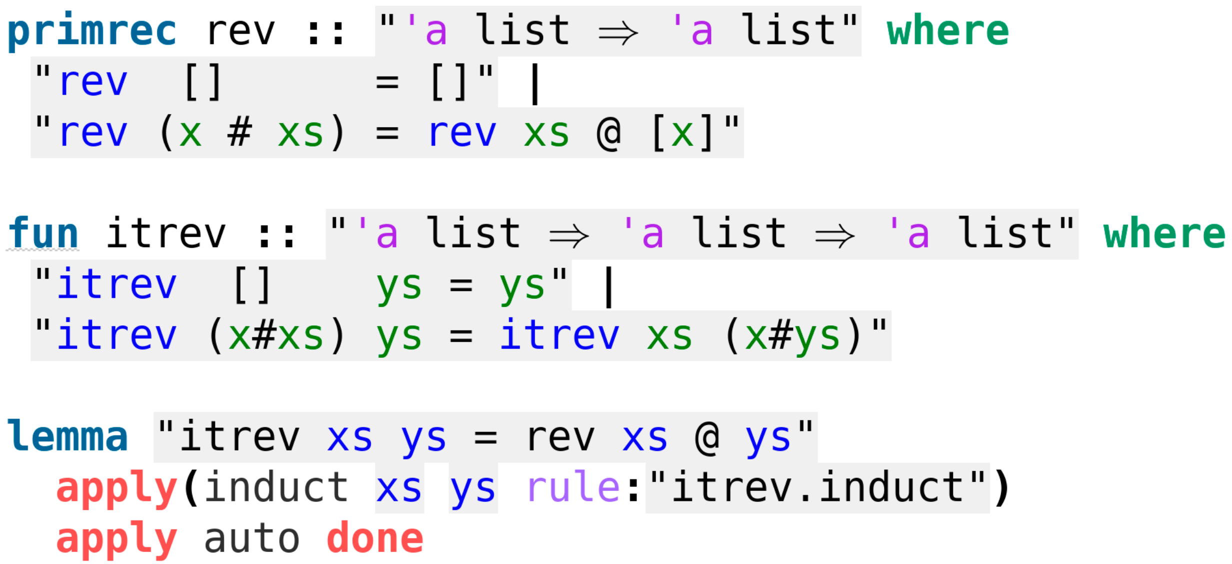
```
  ∃ r1 : rule. True
→
   ∃ r1 : rule.
     ∃ t1 : term.
       ∃ to1 : term_occurrence ∈ t1 : term.
           r1 is_rule_of to1
         ∧
           ∀ t2 : term ∈ induction_term.
             ∃ to2 : term_occurrence ∈ t2 : term.
               ∃ n : number.
                   is_nth_argument_of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

implication
existential quantifier
variable for auxiliary lemmas
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
```

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



```
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
```

good induction ->

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

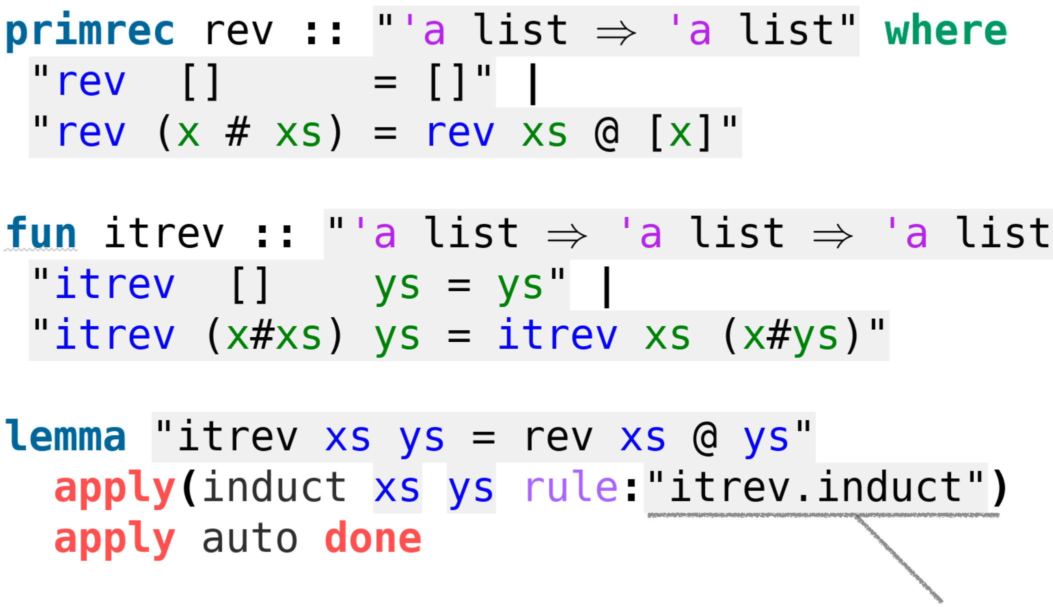
```
primrec rev :: "'a list ⇒ 'a list" where
 "rev  []       = []" |
 "rev (x # xs) = rev xs @ [x]"

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
```

good induction ->

r1

(r1 = itrev.induct)

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



```
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
```

good induction ->

r1

(r1 = itrev.induct)

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

```
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
```

good induction ->

r1

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

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

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

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

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

to1

good induction ->

r1

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

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

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

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

to1

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

good induction ->

r1

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

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

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

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

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

to1

good induction ->

r1

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

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

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

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

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

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

to1

good induction ->

r1

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

(r1 = itrev.induct)

(t1 = itrev)

(to1 = itrev)

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

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

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

to1

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

good induction ->

r1

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

(r1 = itrev.induct)

(t1 = itrev)

(to1 = itrev)

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

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

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

to1

good induction ->

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

r1

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

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

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

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

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

to1

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

good induction ->

t2

r1

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

(r1 = itrev.induct)

(t1 = itrev)

(to1 = itrev)

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

(t2 = xs and ys)

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

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

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

to1

to2

good induction ->

t2

r1

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

(r1 = itrev.induct)

(t1 = itrev)

(to1 = itrev)

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

(t2 = xs and ys)

(to2 = xs and ys)

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

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

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

to1

to2

good induction ->

t2

r1

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

(r1 = itrev.induct)

(t1 = itrev)

(to1 = itrev)

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

(t2 = xs and ys)

(to2 = xs and ys)

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

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

to1

to2

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

good induction ->

t2

r1

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

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

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

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

to1

first

to2

good induction ->

t2

first

r1

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

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

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

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

when t2 is xs (n = 1) ?

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

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

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

to1 first to2

good induction ->

t2 first r1

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

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

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

(t2 = xs ✓ and ys)
(to2 = xs and ys)

when t2 is xs (n = 1) 👍

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

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

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

to1

first

second

to2

good induction ->

t2

first

second

r1

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

(r1 = itrev.induct)

(t1 = itrev)

(to1 = itrev)

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

(t2 = xs and ys)

(to2 = xs and ys)

when t2 is xs (n = 1)

when t2 is ys (n = 2) ?

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

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

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

to1 first second to2

good induction ->

t2 first second r1

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

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

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

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

when t2 is xs (n = 1) 👍

when t2 is ys (n = 2) ?👍

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

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

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

to1 first second to2

good induction ->

t2 second first r1

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

(r1 = itrev.induct)

(t1 = itrev)

(to1 = itrev)

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

✓ ✓

(t2 = xs and ys)

(to2 = xs and ys)

when t2 is xs (n = 1) ?

when t2 is ys (n = 2) ?

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

fun itrev :: "'a list ⇒ 'a list ⇒ 'a list" where
                                s (x#ys)"
```

Heuristic correctly returns true to the good induction.

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

good induction ->

t2

first

second

r1

$\exists\ r1$: rule. True

$\rightarrow$

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

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

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

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

when t2 is xs (n = 1) ?

when t2 is ys (n = 2) ?

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

fun itrev :: "'a list ⇒ 'a list ⇒ 'a list" where
s (x#ys)"

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

Heuristic correctly returns true to the good induction.

Success!

good induction

r1

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

v.induct)

(t1 = itrev)

(to1 = itrev)

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

(t2 = xs ✓ and ys ✓)

(to2 = xs and ys)

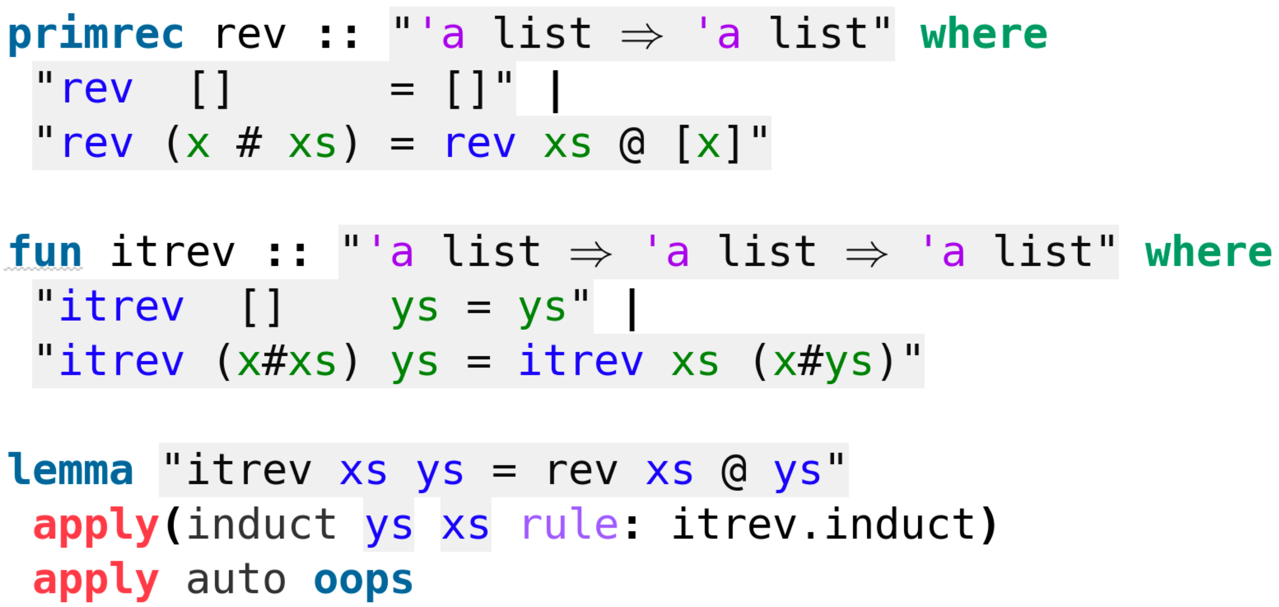
when t2 is xs (n = 1) ?

when t2 is ys (n = 2) ?

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

the same LiFtEr heuristic

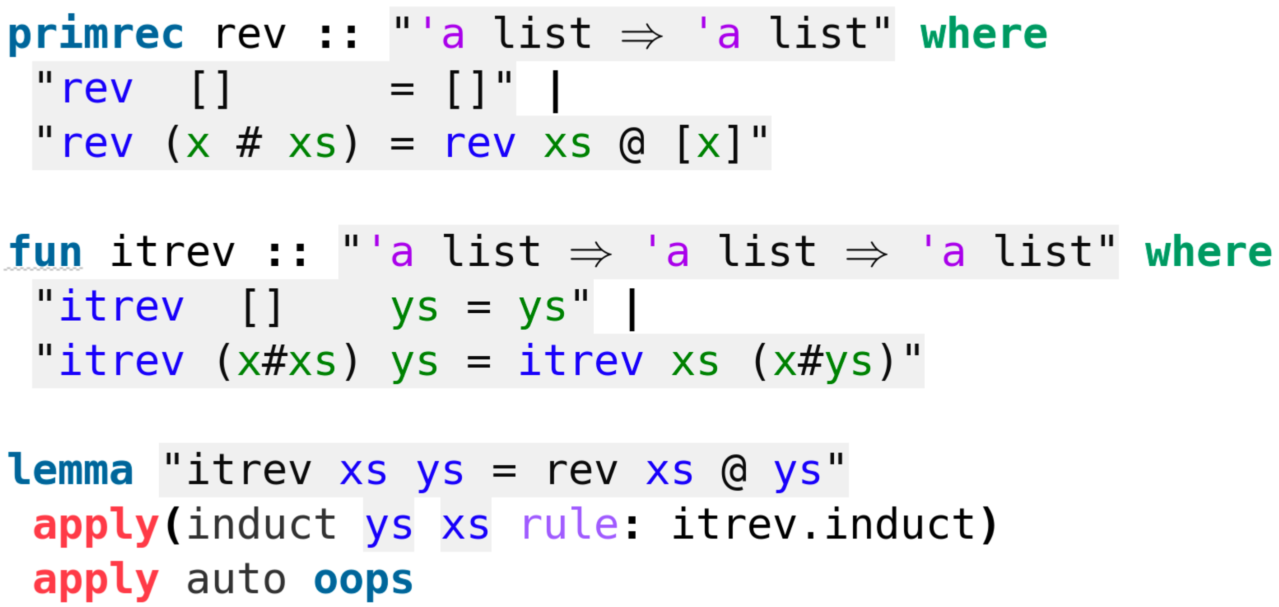
```
  ∃ r1 : rule. True
→
  ∃ r1 : rule.
    ∃ t1 : term.
      ∃ to1 : term_occurrence ∈ t1 : term.
          r1 is_rule_of to1
        ∧
          ∀ t2 : term ∈ induction_term.
            ∃ to2 : term_occurrence ∈ t2 : term.
              ∃ n : number.
                  is_nth_argument_of (to2, n,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



```
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 ys xs rule: itrev.induct)
 apply auto oops
```

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

the same LiFtEr heuristic



```
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 ys xs rule: itrev.induct)
 apply auto oops
```

same lemma ->

the same LiFtEr heuristic

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

```
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)"
```

same lemma ->

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

bad induction ->

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

the same LiFtEr heuristic

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

```
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)"
```

same lemma ->

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

bad induction ->

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

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

the same LiFtEr heuristic

```
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)"
```

same lemma ->

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

bad induction ->

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

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

the same LiFtEr heuristic

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

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

Heuristic correctly returns <u>false</u> to the <u>bad</u> induction.

```
                                        s (x#ys)"
```

same lemma ->

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

bad induction ->

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

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

the same LiFtEr heuristic

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

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

Heuristic correctly returns false to the bad induction.

```
s (x#ys)"
```

same lemma ->

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

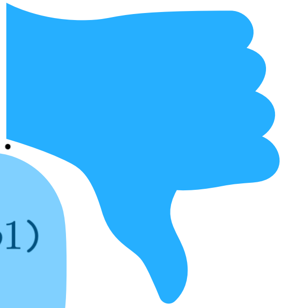
bad induction -

```
v.induct)
```

Success!

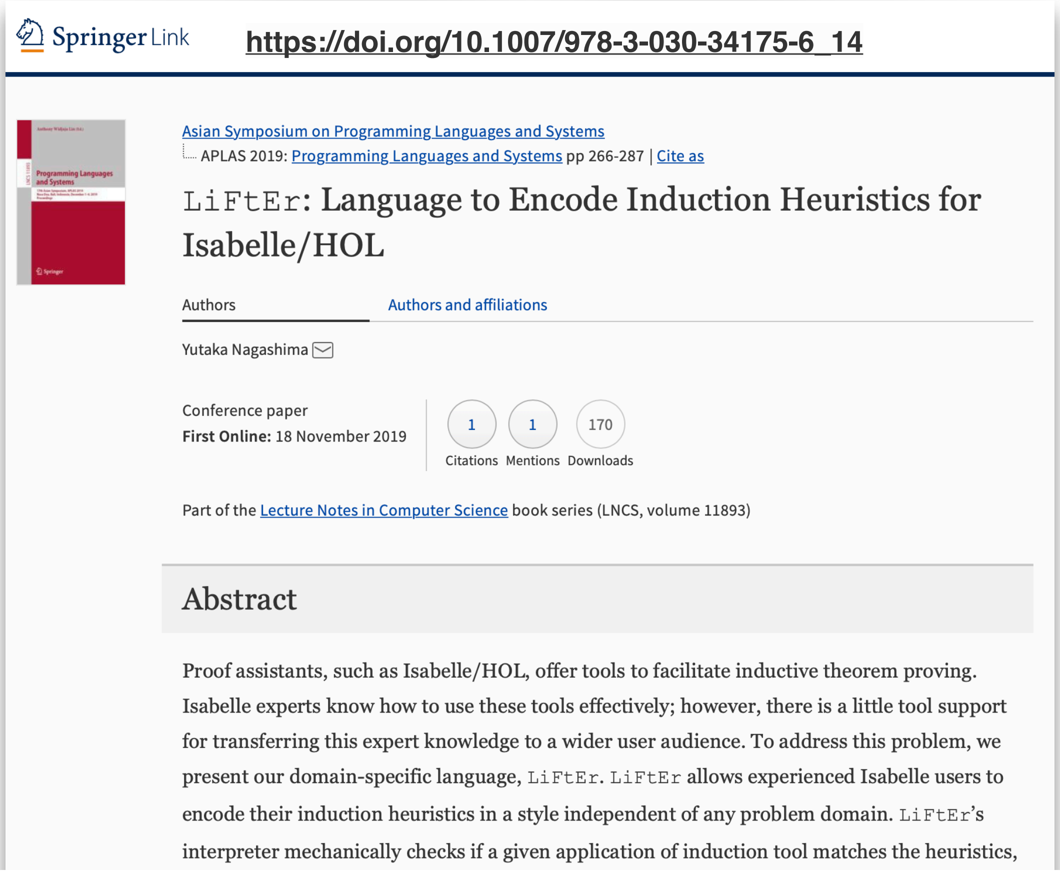
same LiFtEr heuristic

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



LiFtEr at APLAS2019 and Smart Induct at FMCAD2020 (nest week)

LiFtEr at APLAS2019 and Smart Induct at FMCAD2020 (nest week)



Springer Link https://doi.org/10.1007/978-3-030-34175-6_14

Asian Symposium on Programming Languages and Systems

APLAS 2019: Programming Languages and Systems pp 266-287 | Cite as

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

Yutaka Nagashima

Conference paper

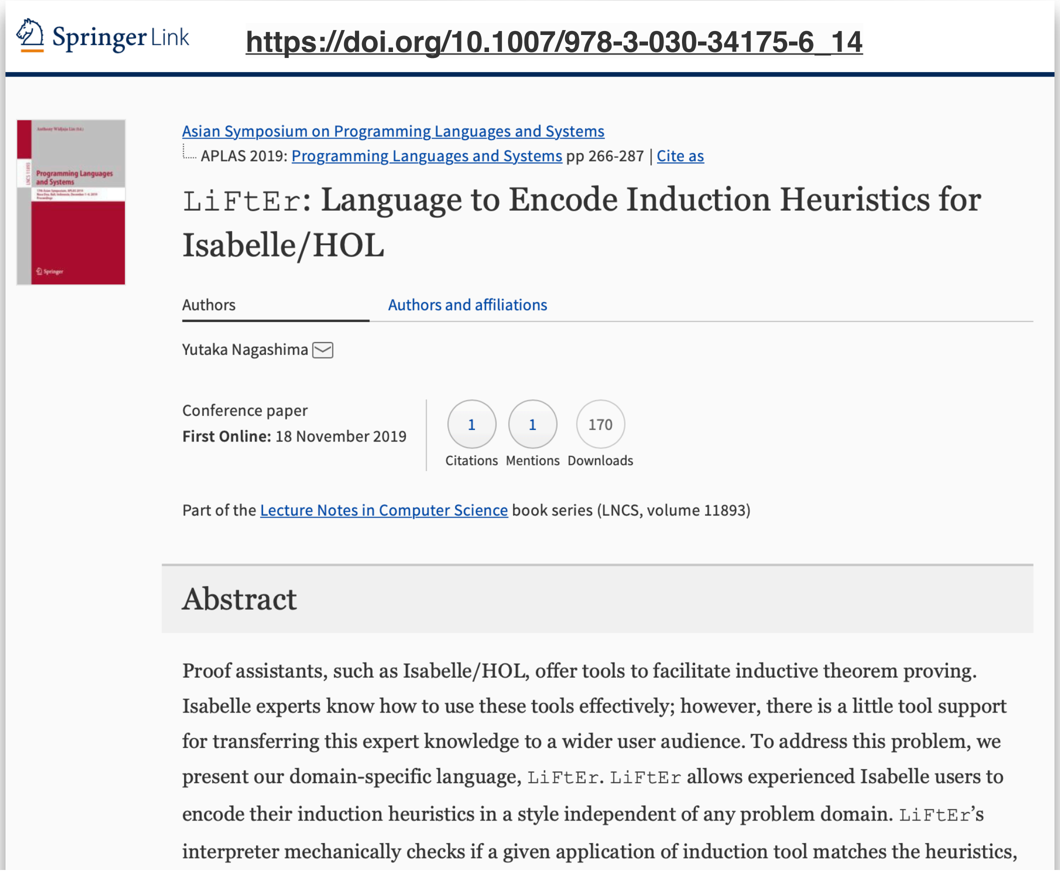
First Online: 18 November 2019

Part of the Lecture Notes in Computer Science book series (LNCS, volume 11893)

Abstract

Proof assistants, such as Isabelle/HOL, offer tools to facilitate inductive theorem proving. Isabelle experts know how to use these tools effectively; however, there is a little tool support for transferring this expert knowledge to a wider user audience. To address this problem, we present our domain-specific language, LiFtEr. LiFtEr allows experienced Isabelle users to encode their induction heuristics in a style independent of any problem domain. LiFtEr's interpreter mechanically checks if a given application of induction tool matches the heuristics,

LiFtEr at APLAS2019 and Smart Induct at FMCAD2020 (nest week)



Springer Link https://doi.org/10.1007/978-3-030-34175-6_14

Asian Symposium on Programming Languages and Systems

APLAS 2019: Programming Languages and Systems pp 266-287 | Cite as

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

Yutaka Nagashima

Conference paper

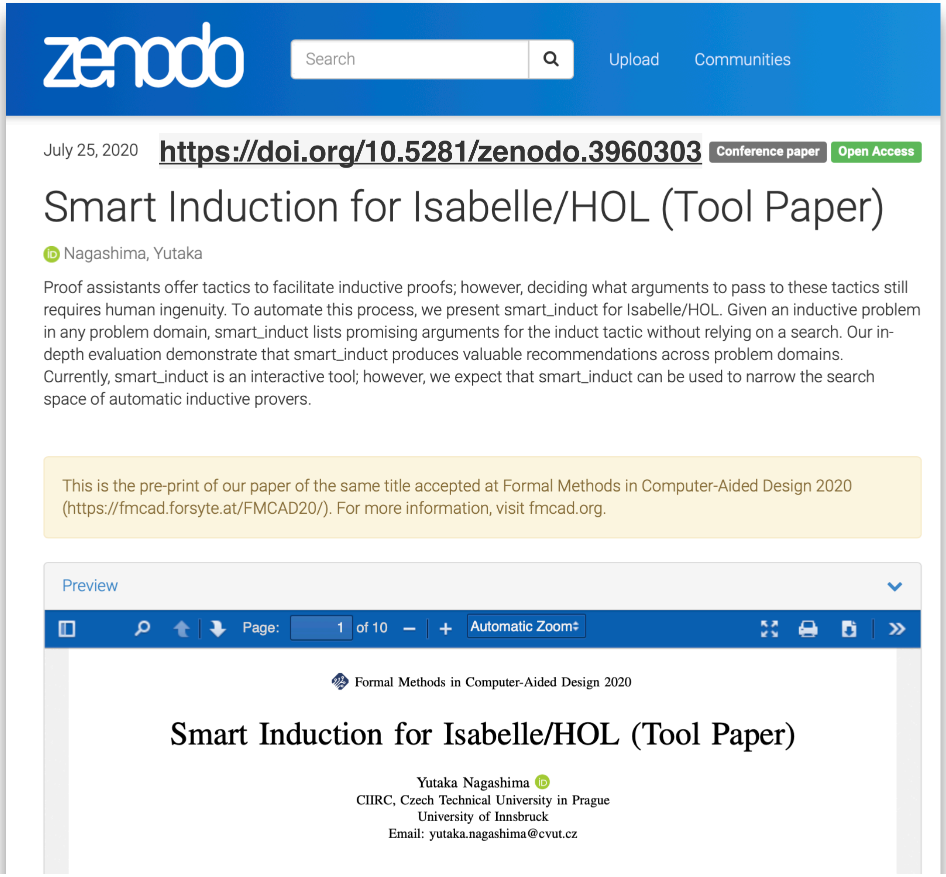
First Online: 18 November 2019

1	1	170
Citations	Mentions	Downloads

Part of the Lecture Notes in Computer Science book series (LNCS, volume 11893)

Abstract

Proof assistants, such as Isabelle/HOL, offer tools to facilitate inductive theorem proving. Isabelle experts know how to use these tools effectively; however, there is a little tool support for transferring this expert knowledge to a wider user audience. To address this problem, we present our domain-specific language, LiFtEr. LiFtEr allows experienced Isabelle users to encode their induction heuristics in a style independent of any problem domain. LiFtEr's interpreter mechanically checks if a given application of induction tool matches the heuristics,



July 25, 2020 https://doi.org/10.5281/zenodo.3960303 Conference paper Open Access

Smart Induction for Isabelle/HOL (Tool Paper)

Nagashima, Yutaka

Proof assistants offer tactics to facilitate inductive proofs; however, deciding what arguments to pass to these tactics still requires human ingenuity. To automate this process, we present smart_induct for Isabelle/HOL. Given an inductive problem in any problem domain, smart_induct lists promising arguments for the induct tactic without relying on a search. Our in-depth evaluation demonstrate that smart_induct produces valuable recommendations across problem domains. Currently, smart_induct is an interactive tool; however, we expect that smart_induct can be used to narrow the search space of automatic inductive provers.

This is the pre-print of our paper of the same title accepted at Formal Methods in Computer-Aided Design 2020 (https://fmcad.forsyte.at/FMCAD20/). For more information, visit fmcad.org.

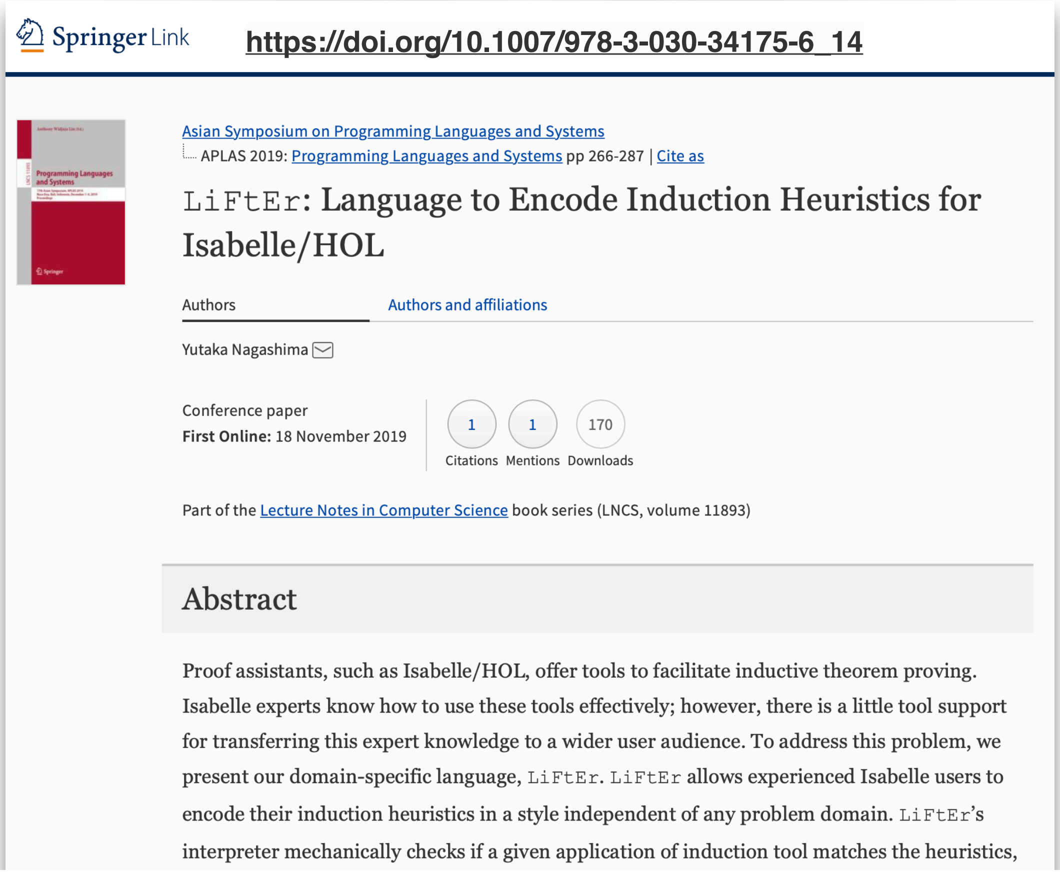
Formal Methods in Computer-Aided Design 2020

Smart Induction for Isabelle/HOL (Tool Paper)

Yutaka Nagashima
CIIRC, Czech Technical University in Prague
University of Innsbruck
Email: yutaka.nagashima@cvut.cz

LiFtEr at APLAS2019 and Smart Induct at FMCAD2020 (nest week)

On which variables to apply induction



Springer Link https://doi.org/10.1007/978-3-030-34175-6_14

Asian Symposium on Programming Languages and Systems
APLAS 2019: Programming Languages and Systems pp 266-287 | Cite as

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

Authors | Authors and affiliations

Yutaka Nagashima

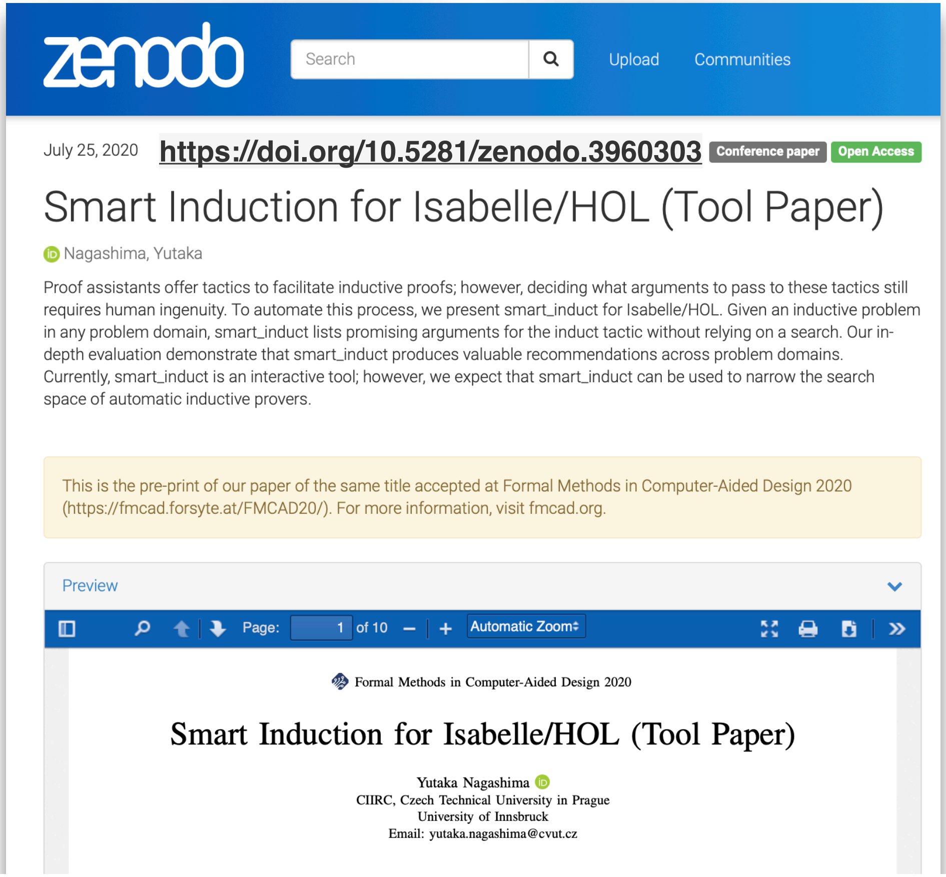
Conference paper
First Online: 18 November 2019

1 Citations · 1 Mentions · 170 Downloads

Part of the Lecture Notes in Computer Science book series (LNCS, volume 11893)

Abstract

Proof assistants, such as Isabelle/HOL, offer tools to facilitate inductive theorem proving. Isabelle experts know how to use these tools effectively; however, there is a little tool support for transferring this expert knowledge to a wider user audience. To address this problem, we present our domain-specific language, LiFtEr. LiFtEr allows experienced Isabelle users to encode their induction heuristics in a style independent of any problem domain. LiFtEr's interpreter mechanically checks if a given application of induction tool matches the heuristics,



July 25, 2020 https://doi.org/10.5281/zenodo.3960303 Conference paper Open Access

Smart Induction for Isabelle/HOL (Tool Paper)

Nagashima, Yutaka

Proof assistants offer tactics to facilitate inductive proofs; however, deciding what arguments to pass to these tactics still requires human ingenuity. To automate this process, we present smart_induct for Isabelle/HOL. Given an inductive problem in any problem domain, smart_induct lists promising arguments for the induct tactic without relying on a search. Our in-depth evaluation demonstrate that smart_induct produces valuable recommendations across problem domains. Currently, smart_induct is an interactive tool; however, we expect that smart_induct can be used to narrow the search space of automatic inductive provers.

This is the pre-print of our paper of the same title accepted at Formal Methods in Computer-Aided Design 2020 (https://fmcad.forsyte.at/FMCAD20/). For more information, visit fmcad.org.

Formal Methods in Computer-Aided Design 2020

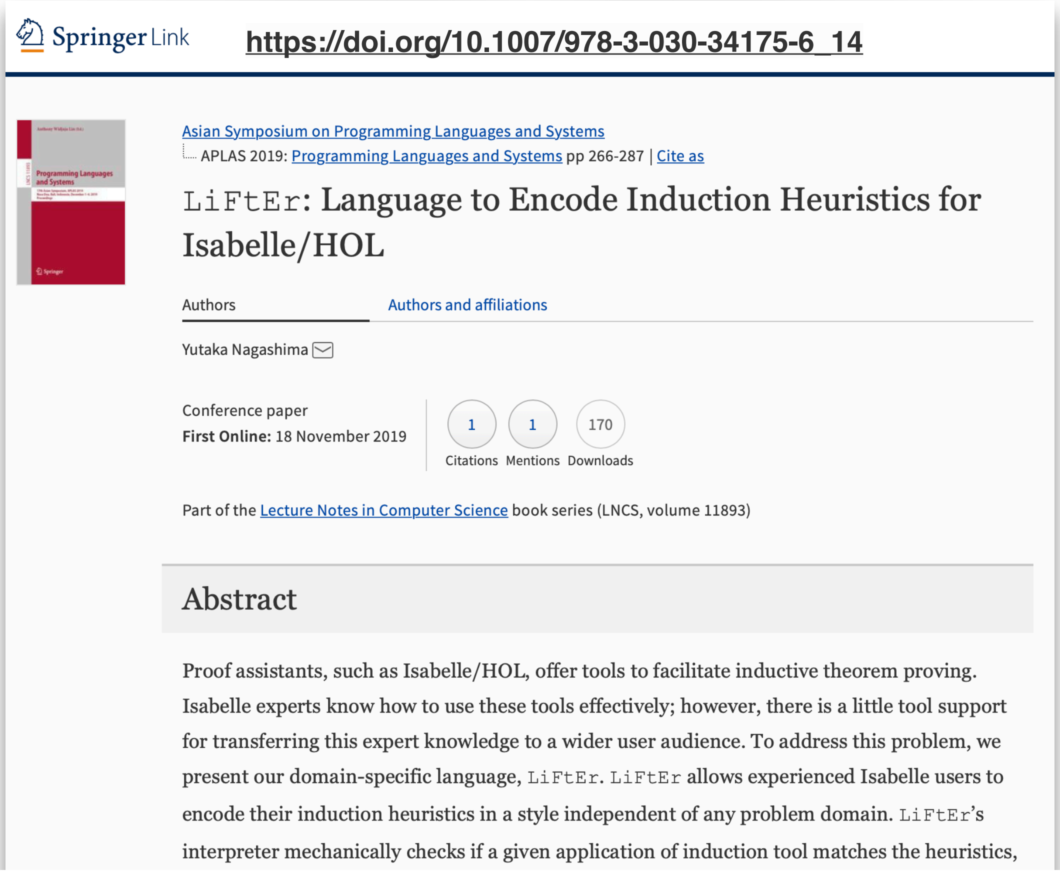
Smart Induction for Isabelle/HOL (Tool Paper)

Yutaka Nagashima
CIIRC, Czech Technical University in Prague
University of Innsbruck
Email: yutaka.nagashima@cvut.cz

LiFtEr at APLAS2019 and Smart Induct at FMCAD2020 (nest week)

On which variables to apply induction

Variable generalisation



https://doi.org/10.1007/978-3-030-34175-6_14

Asian Symposium on Programming Languages and Systems

APLAS 2019: Programming Languages and Systems pp 266-287 | Cite as

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

Yutaka Nagashima

Conference paper

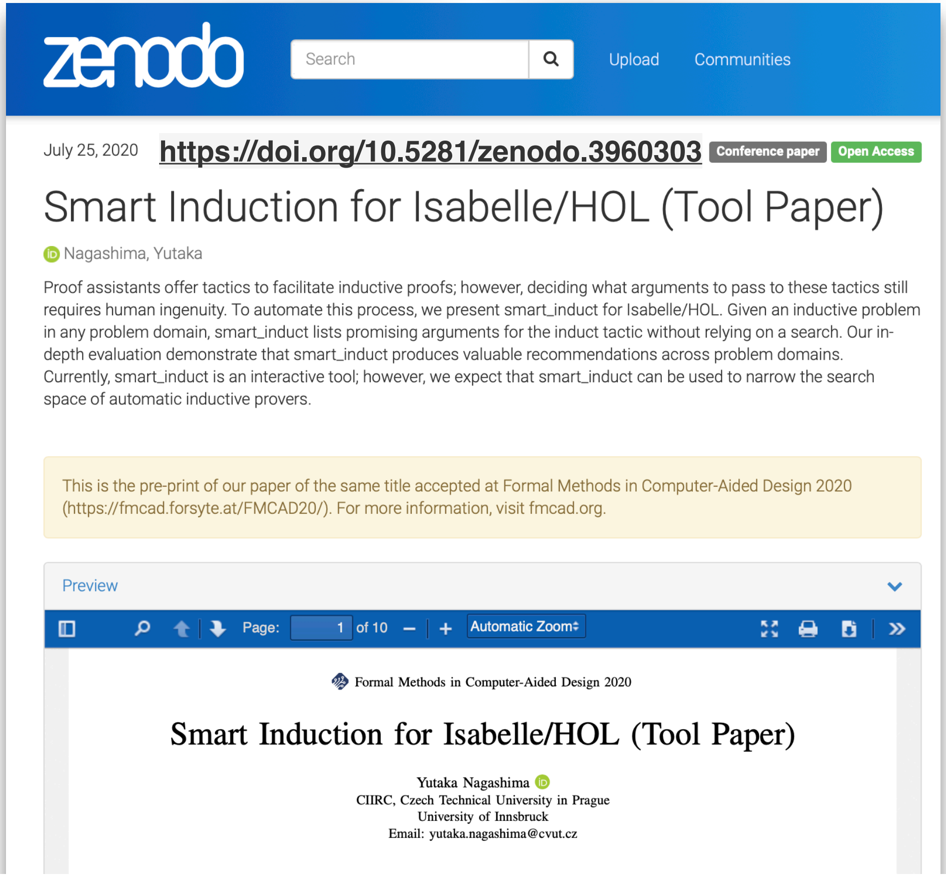
First Online: 18 November 2019

1	1	170
Citations	Mentions	Downloads

Part of the Lecture Notes in Computer Science book series (LNCS, volume 11893)

Abstract

Proof assistants, such as Isabelle/HOL, offer tools to facilitate inductive theorem proving. Isabelle experts know how to use these tools effectively; however, there is a little tool support for transferring this expert knowledge to a wider user audience. To address this problem, we present our domain-specific language, LiFtEr. LiFtEr allows experienced Isabelle users to encode their induction heuristics in a style independent of any problem domain. LiFtEr's interpreter mechanically checks if a given application of induction tool matches the heuristics,



July 25, 2020 https://doi.org/10.5281/zenodo.3960303 Conference paper Open Access

Smart Induction for Isabelle/HOL (Tool Paper)

Nagashima, Yutaka

Proof assistants offer tactics to facilitate inductive proofs; however, deciding what arguments to pass to these tactics still requires human ingenuity. To automate this process, we present smart_induct for Isabelle/HOL. Given an inductive problem in any problem domain, smart_induct lists promising arguments for the induct tactic without relying on a search. Our in-depth evaluation demonstrate that smart_induct produces valuable recommendations across problem domains. Currently, smart_induct is an interactive tool; however, we expect that smart_induct can be used to narrow the search space of automatic inductive provers.

This is the pre-print of our paper of the same title accepted at Formal Methods in Computer-Aided Design 2020 (https://fmcad.forsyte.at/FMCAD20/). For more information, visit fmcad.org.

Formal Methods in Computer-Aided Design 2020

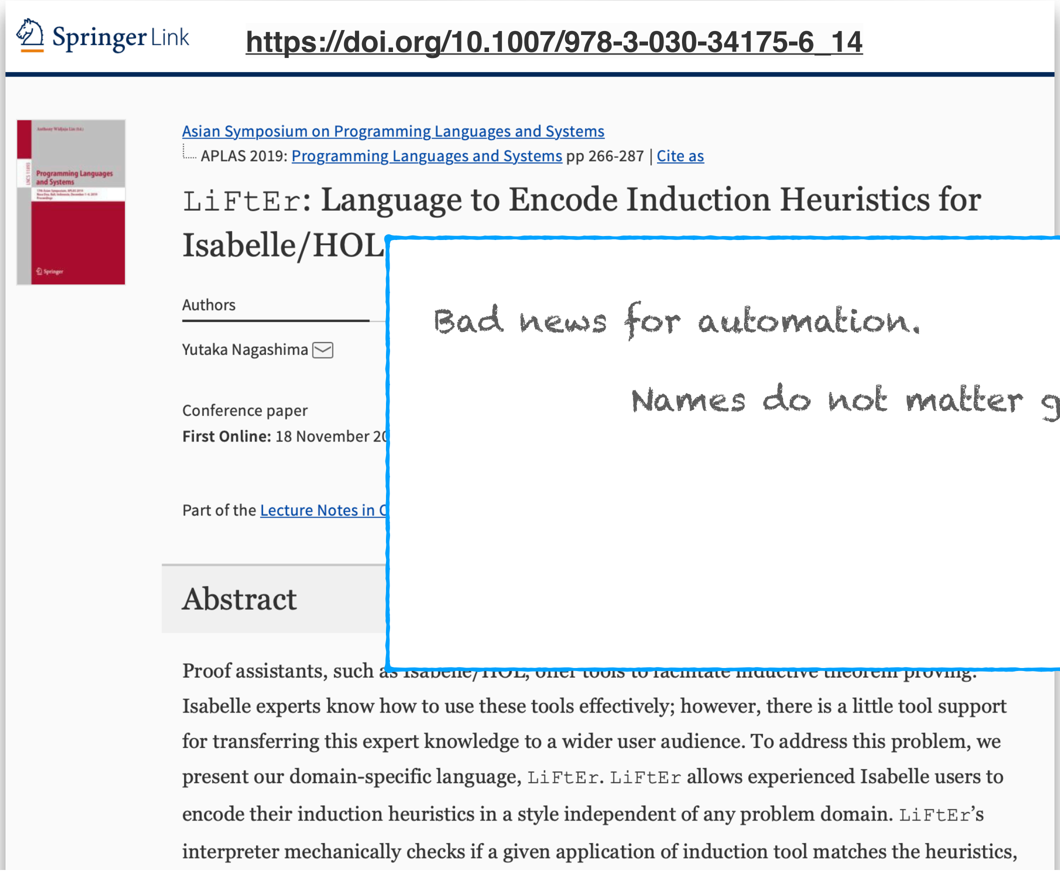
Smart Induction for Isabelle/HOL (Tool Paper)

Yutaka Nagashima
CIIRC, Czech Technical University in Prague
University of Innsbruck
Email: yutaka.nagashima@cvut.cz

LiFtEr at APLAS2019 and Smart Induct at FMCAD2020 (nest week)

On which variables to apply induction

Variable generalisation



Springer Link https://doi.org/10.1007/978-3-030-34175-6_14

Asian Symposium on Programming Languages and Systems
APLAS 2019: Programming Languages and Systems pp 266-287 | Cite as

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

Authors

Yutaka Nagashima

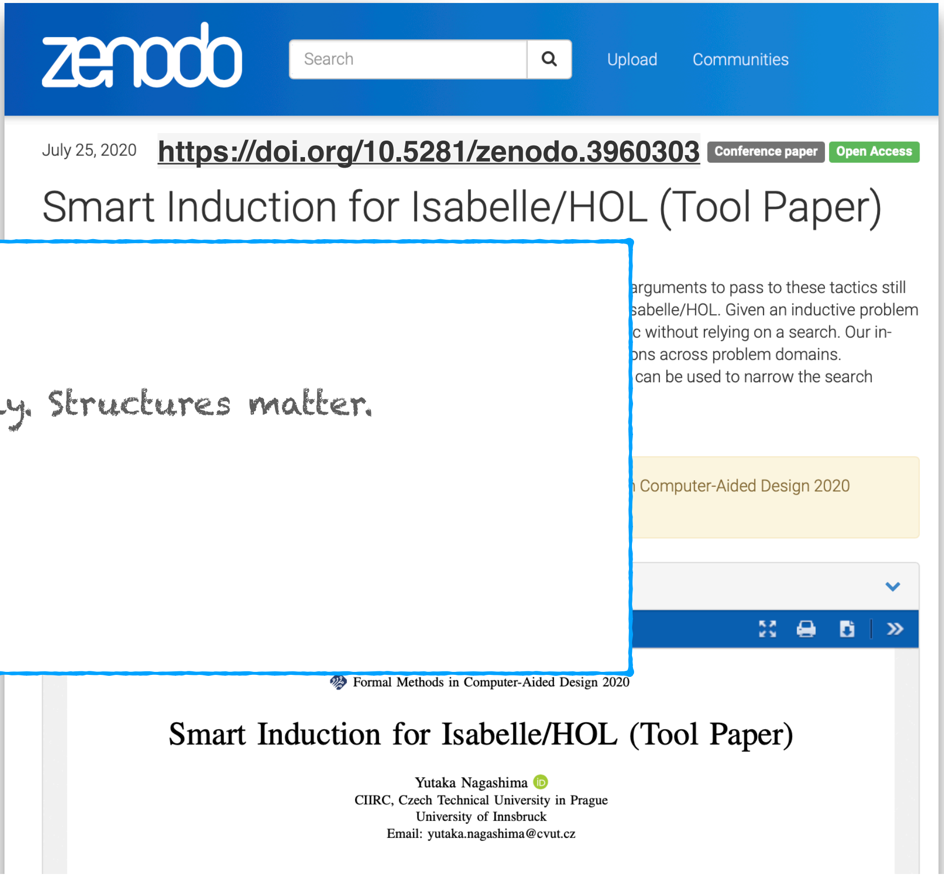
Conference paper

First Online: 18 November 20

Part of the Lecture Notes in C

Abstract

Proof assistants, such as Isabelle/HOL, offer tools to facilitate inductive theorem proving. Isabelle experts know how to use these tools effectively; however, there is a little tool support for transferring this expert knowledge to a wider user audience. To address this problem, we present our domain-specific language, LiFtEr. LiFtEr allows experienced Isabelle users to encode their induction heuristics in a style independent of any problem domain. LiFtEr's interpreter mechanically checks if a given application of induction tool matches the heuristics,



zenodo Search Upload Communities

July 25, 2020 https://doi.org/10.5281/zenodo.3960303 Conference paper Open Access

Smart Induction for Isabelle/HOL (Tool Paper)

arguments to pass to these tactics still
sabelle/HOL. Given an inductive problem
c without relying on a search. Our in-
ons across problem domains.
can be used to narrow the search

Computer-Aided Design 2020

Formal Methods in Computer-Aided Design 2020

Smart Induction for Isabelle/HOL (Tool Paper)

Yutaka Nagashima
CIIRC, Czech Technical University in Prague
University of Innsbruck
Email: yutaka.nagashima@cvut.cz

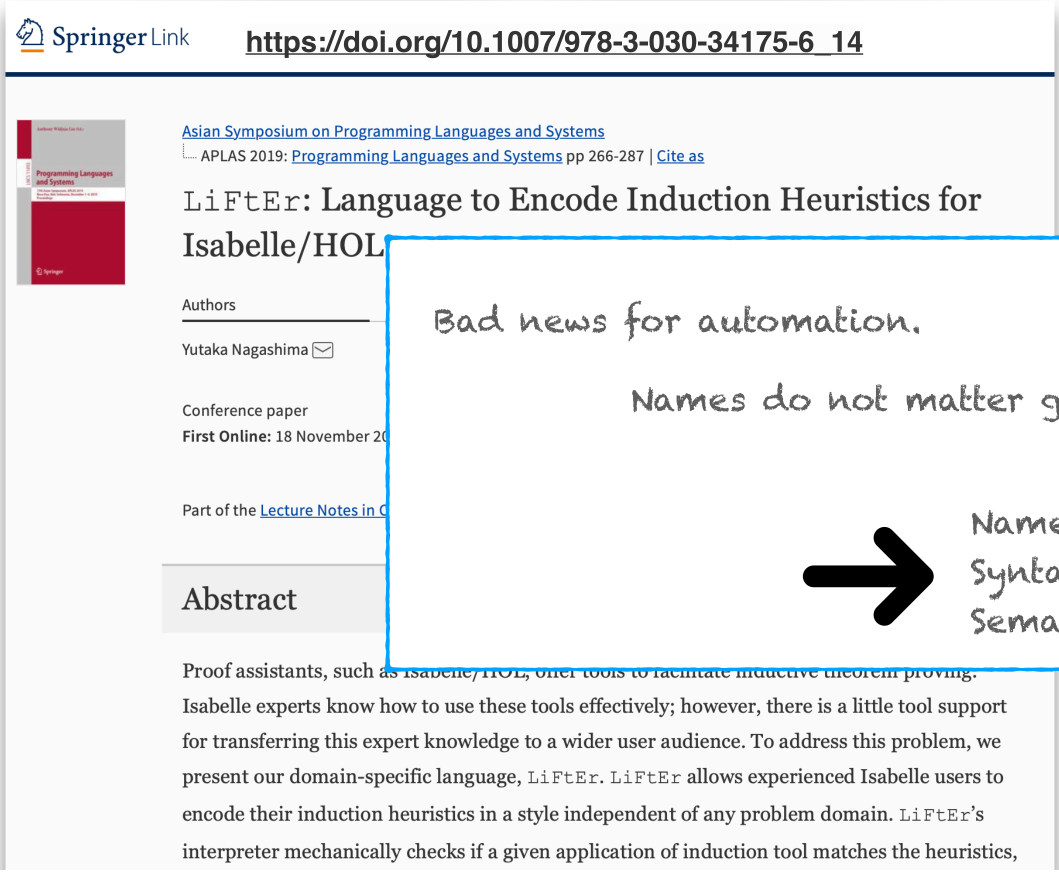
Bad news for automation.

Names do not matter globally. Structures matter.

LiFtEr at APLAS2019 and Smart Induct at FMCAD2020 (nest week)

On which variables to apply induction

Variable generalisation



Springer Link https://doi.org/10.1007/978-3-030-34175-6_14

Asian Symposium on Programming Languages and Systems
APLAS 2019: Programming Languages and Systems pp 266-287 | Cite as

LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

Authors

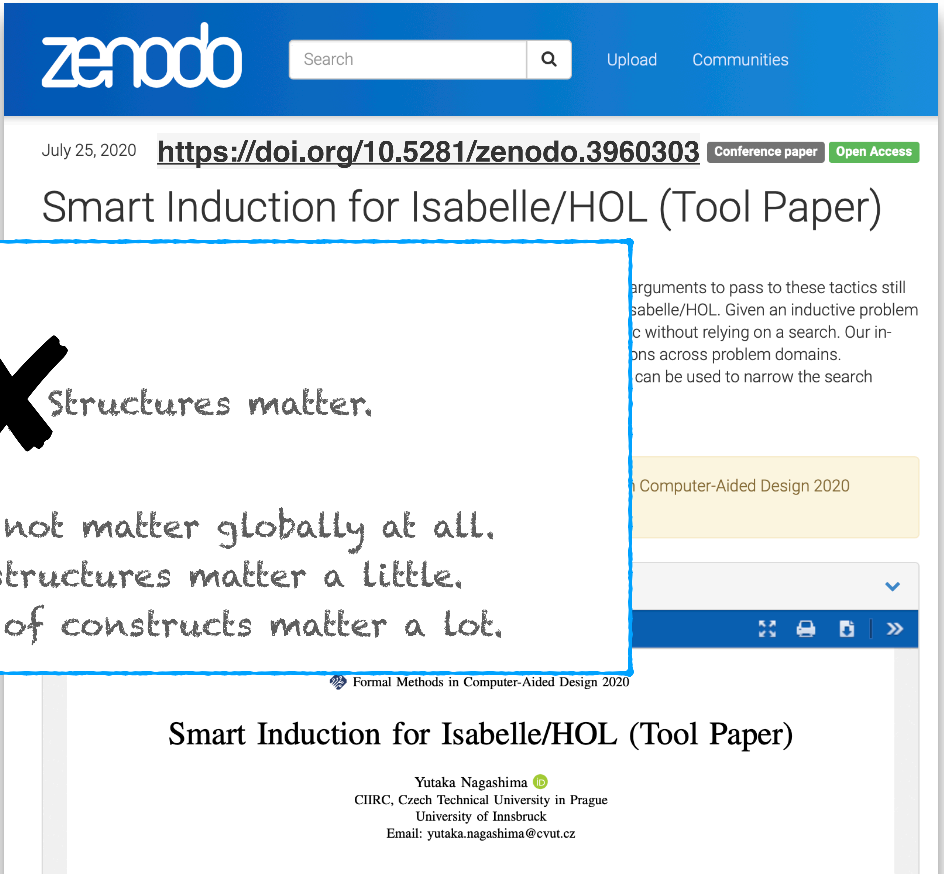
Yutaka Nagashima

Conference paper
First Online: 18 November 20

Part of the Lecture Notes in C

Abstract

Proof assistants, such as Isabelle/HOL, offer tools to facilitate inductive theorem proving. Isabelle experts know how to use these tools effectively; however, there is a little tool support for transferring this expert knowledge to a wider user audience. To address this problem, we present our domain-specific language, LiFtEr. LiFtEr allows experienced Isabelle users to encode their induction heuristics in a style independent of any problem domain. LiFtEr's interpreter mechanically checks if a given application of induction tool matches the heuristics,



July 25, 2020 https://doi.org/10.5281/zenodo.3960303 Conference paper Open Access

Smart Induction for Isabelle/HOL (Tool Paper)

arguments to pass to these tactics still
sabelle/HOL. Given an inductive problem
c without relying on a search. Our in-
ons across problem domains.
can be used to narrow the search

Computer-Aided Design 2020

Formal Methods in Computer-Aided Design 2020

Smart Induction for Isabelle/HOL (Tool Paper)

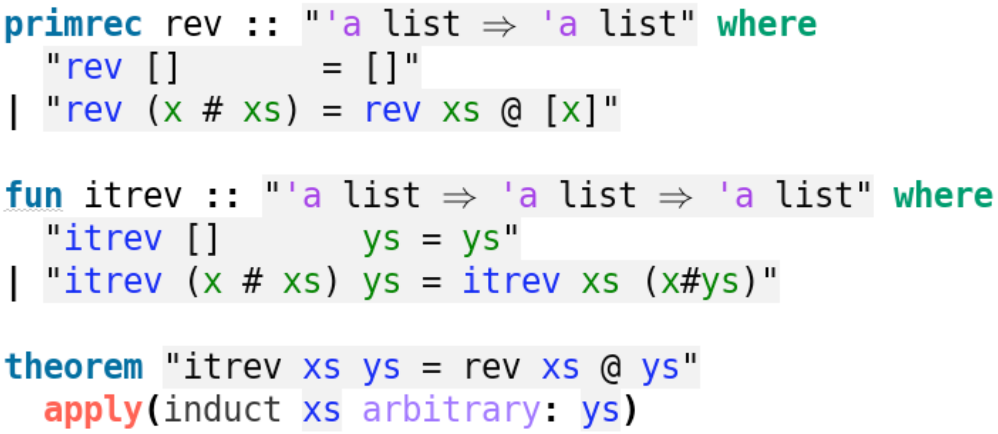
Yutaka Nagashima
CIIRC, Czech Technical University in Prague
University of Innsbruck
Email: yutaka.nagashima@cvut.cz

Bad news for automation.

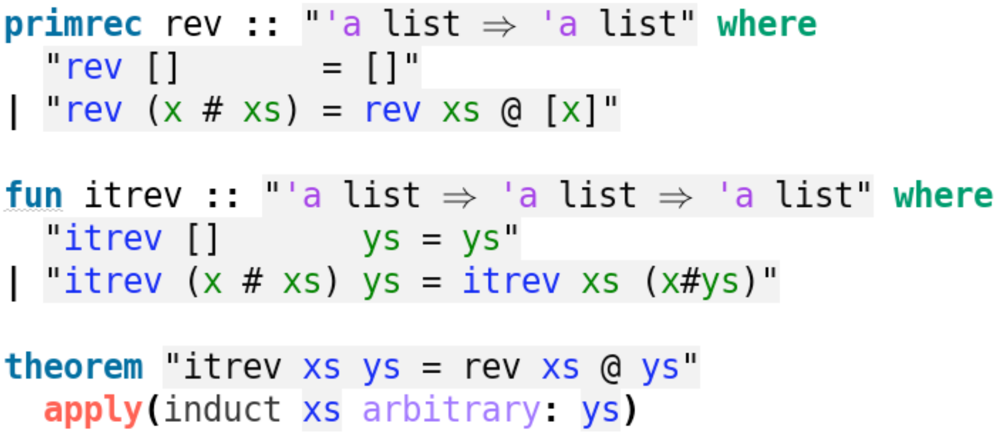
Names do not matter global ✗ Structures matter.

→

Names do not matter globally at all.
Syntactic structures matter a little.
Semantics of constructs matter a lot.

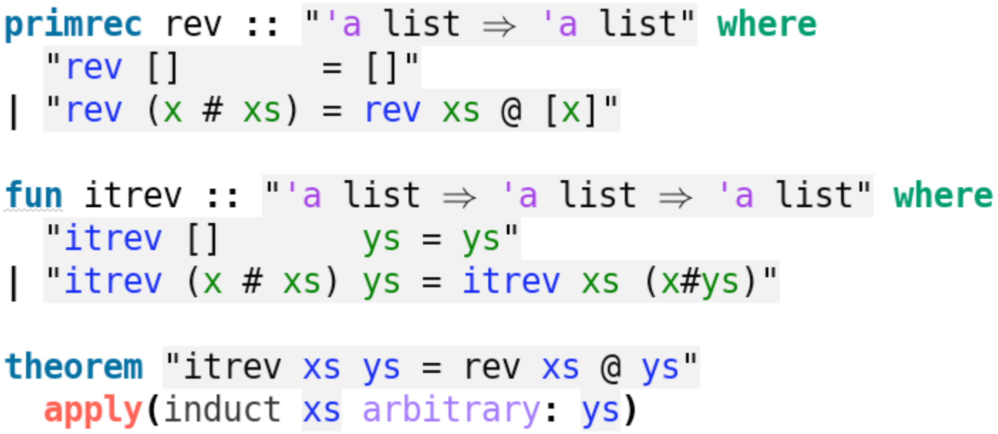


```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []        = []"
| "rev (x # xs) = rev xs @ [x]"

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

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```



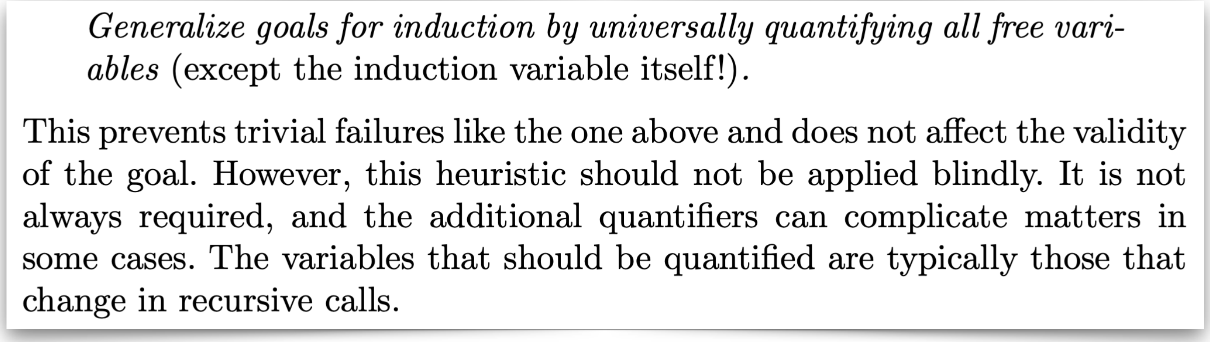
```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []        = []"
| "rev (x # xs) = rev xs @ [x]"

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

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

alternative good proof by induction with generalisation



```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

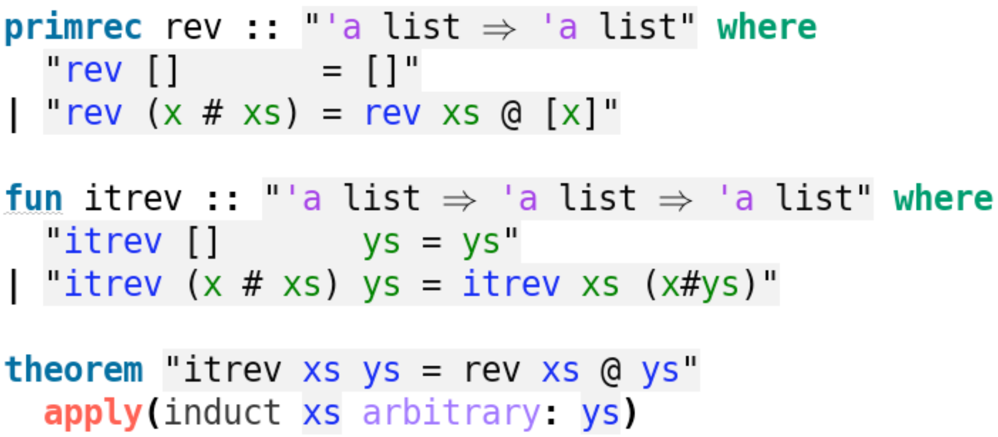
alternative good proof by induction with generalisation



> *Generalize goals for induction by universally quantifying all free variables* (except the induction variable itself!).

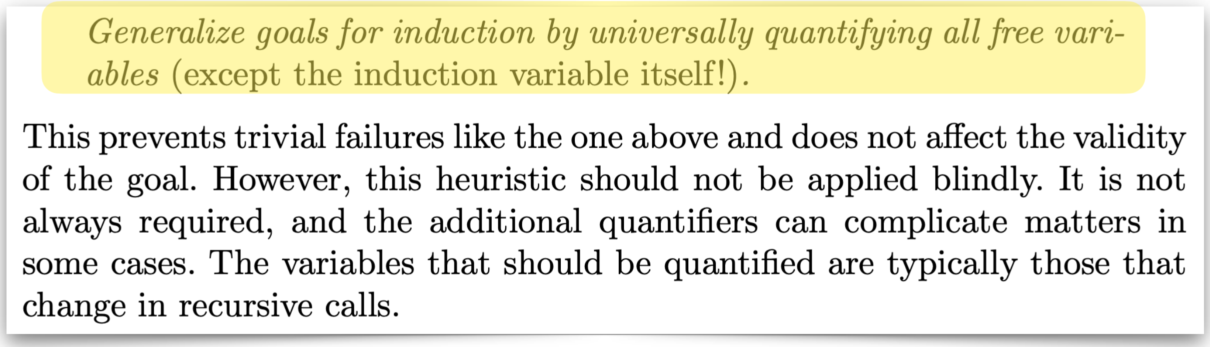
This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

"Isabelle/HOL A Proof Assistant for Higher-Order Logic"
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36



```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

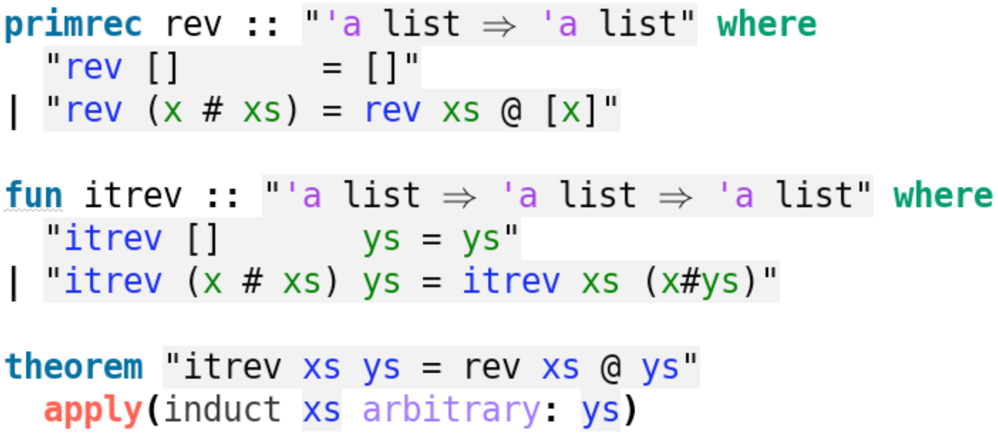
alternative good proof by induction with generalisation



Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

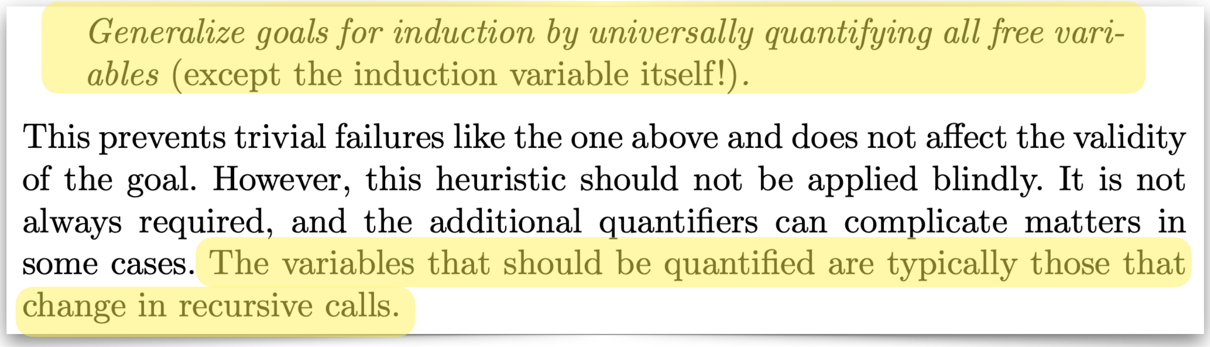
This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

“Isabelle/HOL A Proof Assistant for Higher-Order Logic”
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36



```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

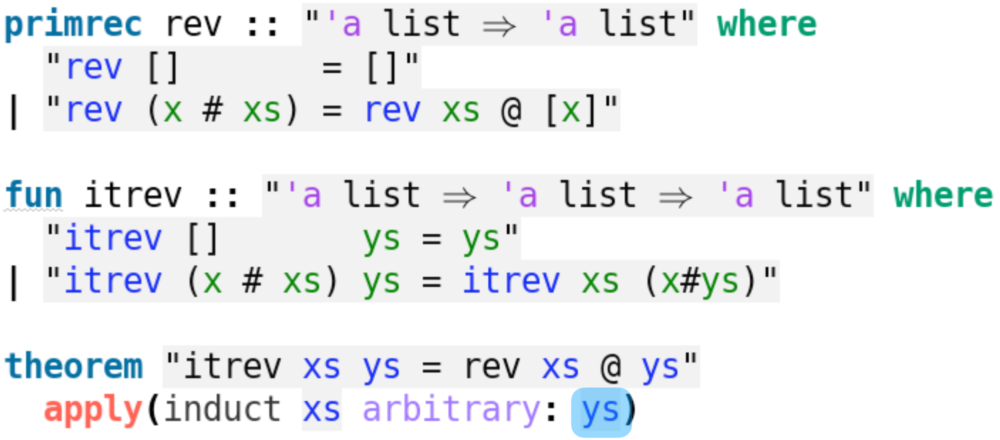
alternative good proof by induction with generalisation



Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

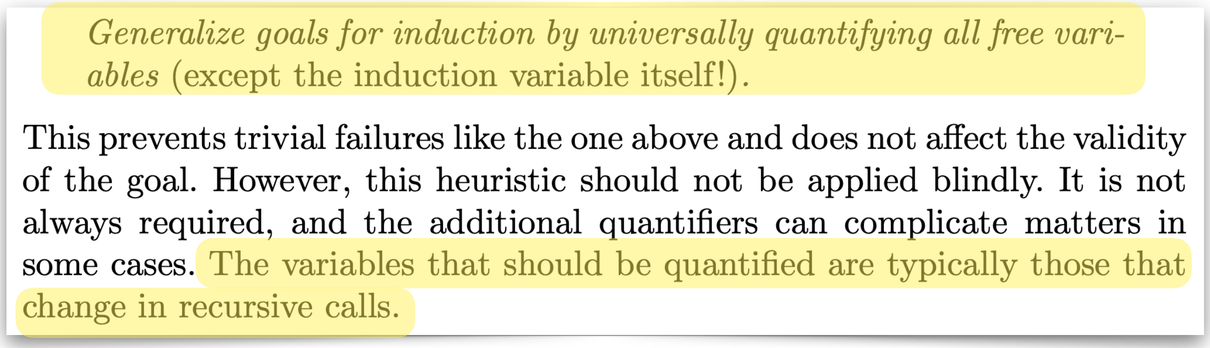
This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

"Isabelle/HOL A Proof Assistant for Higher-Order Logic"
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36



```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

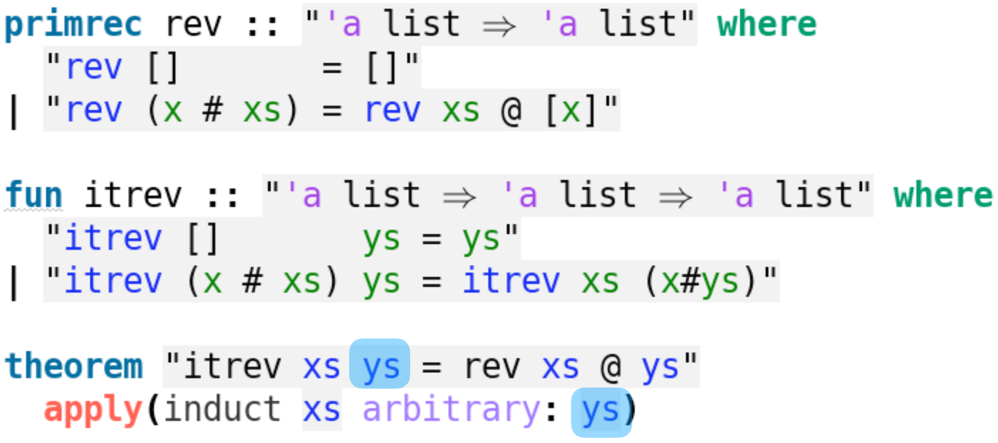
alternative good proof by induction with generalisation



Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

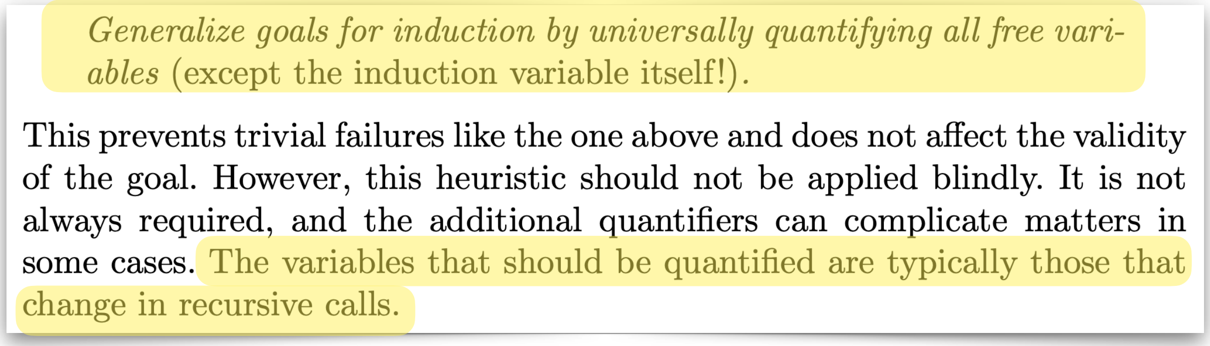
This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

“Isabelle/HOL A Proof Assistant for Higher-Order Logic”
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36



```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

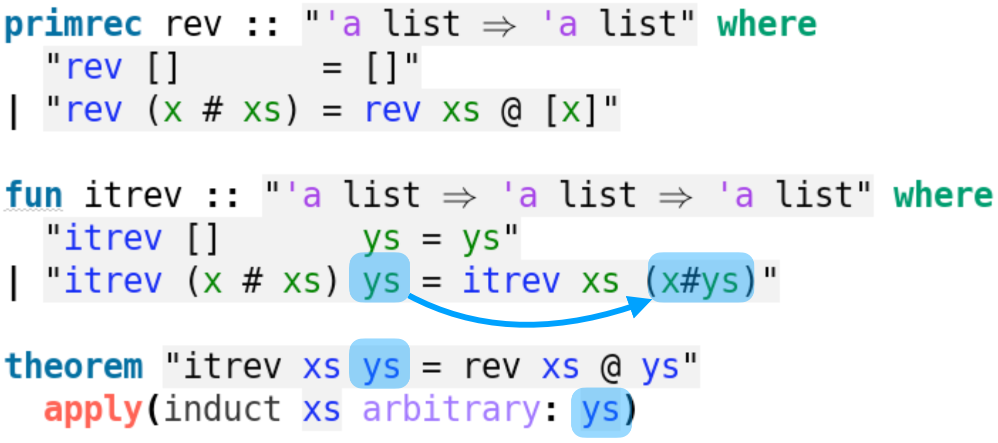
alternative good proof by induction with generalisation



Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

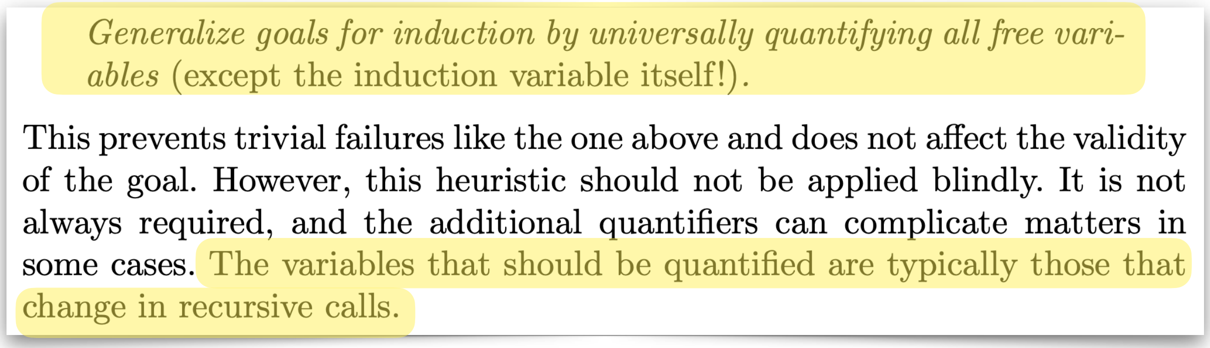
This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

"Isabelle/HOL A Proof Assistant for Higher-Order Logic"
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36



```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

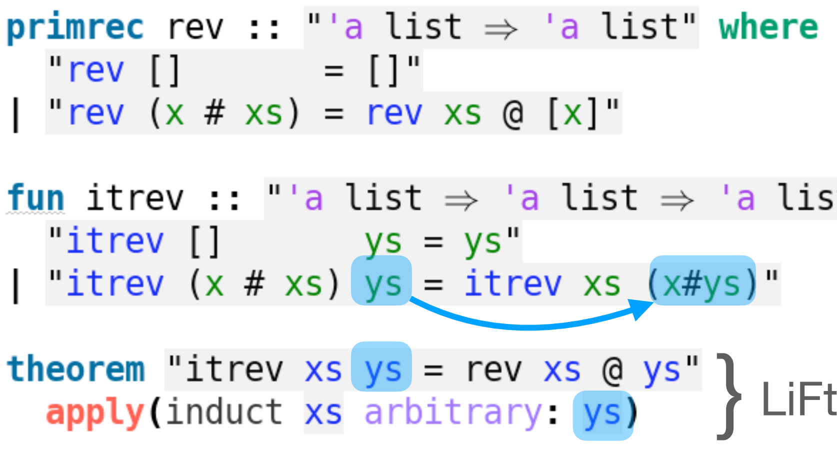
alternative good proof by induction with generalisation



Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

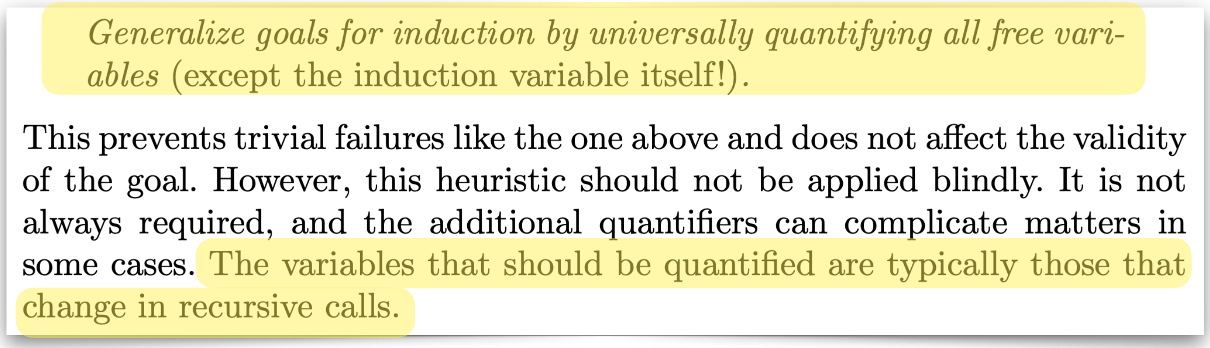
"Isabelle/HOL A Proof Assistant for Higher-Order Logic"
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36



```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

} LiFtEr

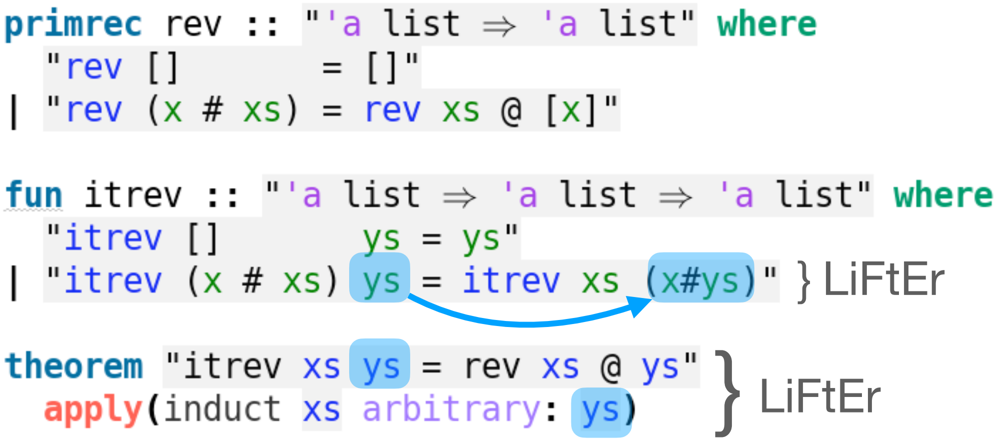
alternative good proof by induction with generalisation



Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

"Isabelle/HOL A Proof Assistant for Higher-Order Logic"
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36

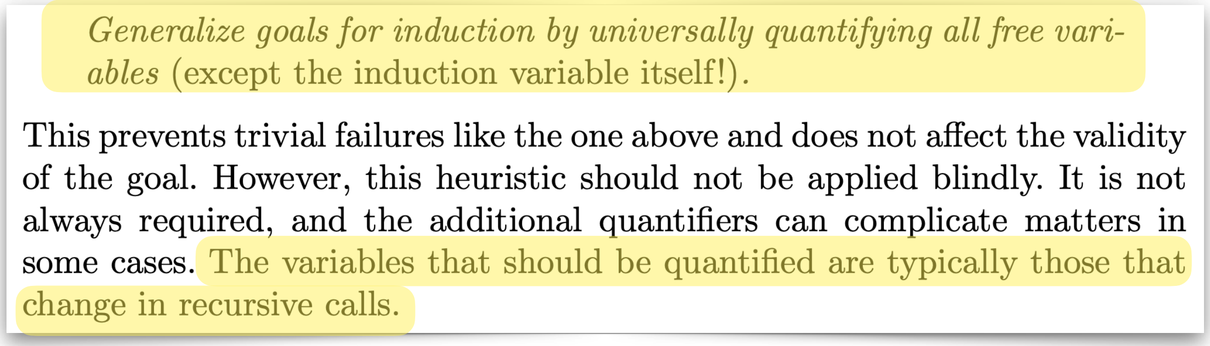


```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []        = []"
| "rev (x # xs) = rev xs @ [x]"

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

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

} LiFtEr

} LiFtEr

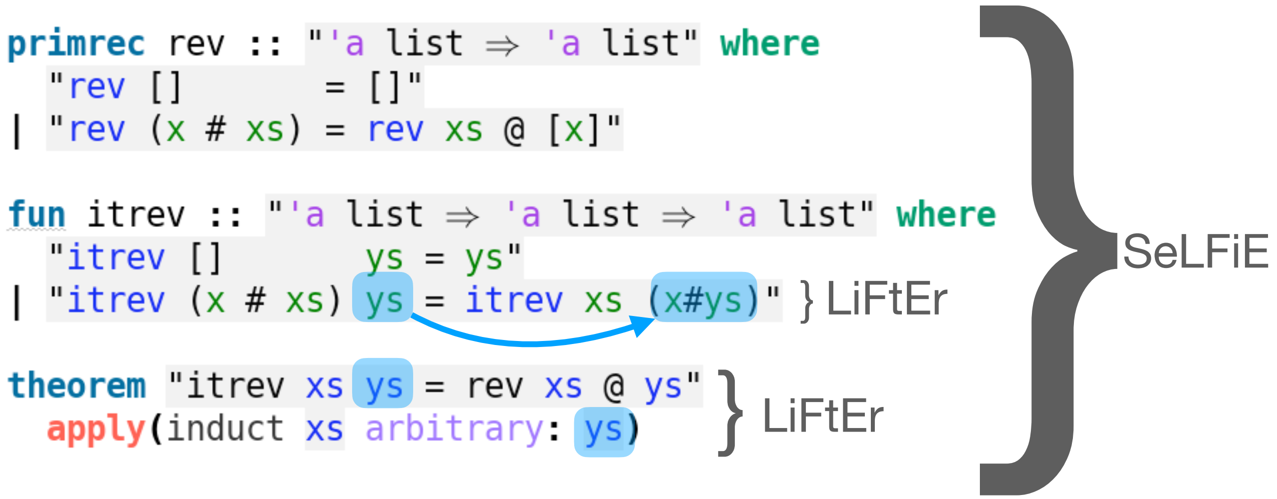
alternative good proof by induction with generalisation



Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

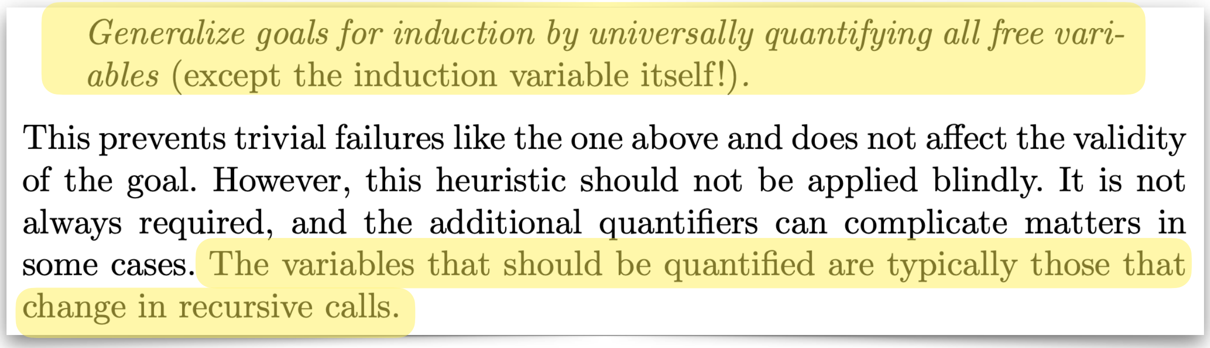
This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

"Isabelle/HOL A Proof Assistant for Higher-Order Logic"
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36



```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

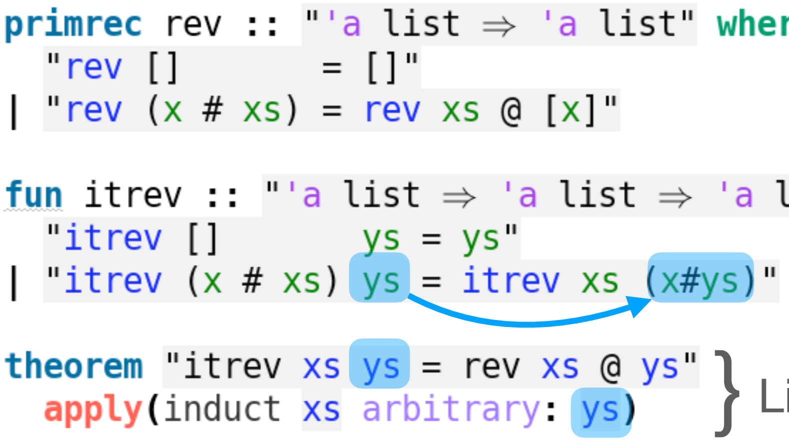
alternative good proof by induction with generalisation



Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

“Isabelle/HOL A Proof Assistant for Higher-Order Logic”
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36



```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

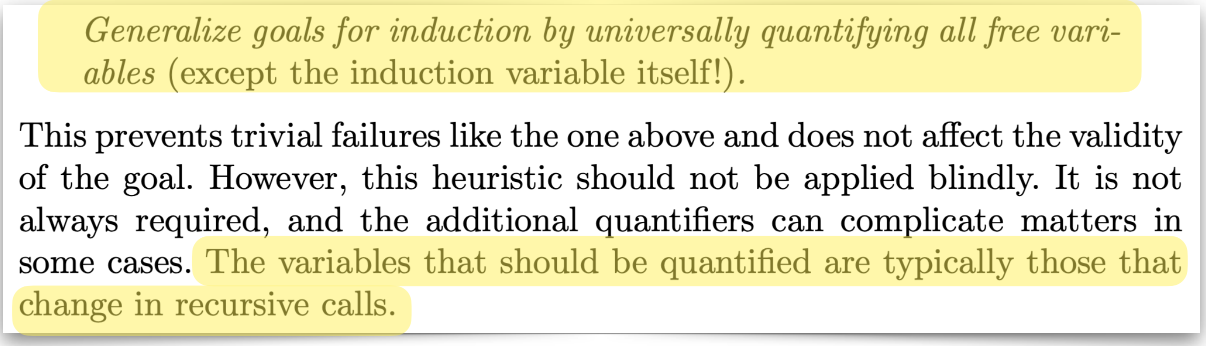
} LiFtEr

} LiFtEr

SeLFiE

} outer part for syntactic analysis

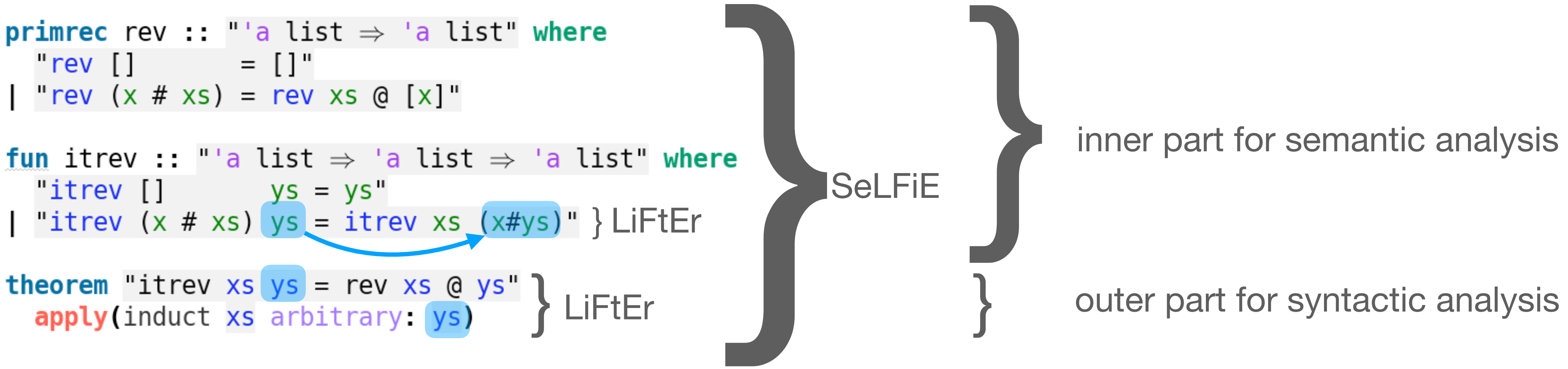
alternative good proof by induction with generalisation



Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

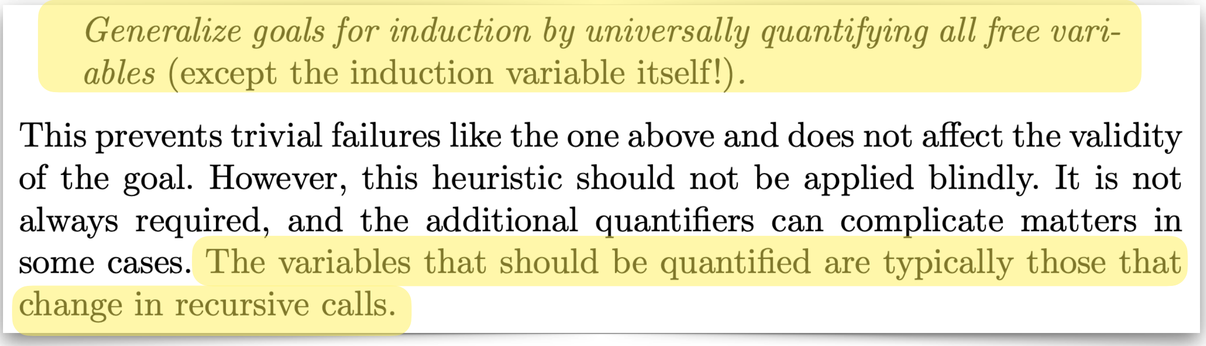
"Isabelle/HOL A Proof Assistant for Higher-Order Logic"
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36



```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

LiFtEr
LiFtEr
SeLFiE
inner part for semantic analysis
outer part for syntactic analysis

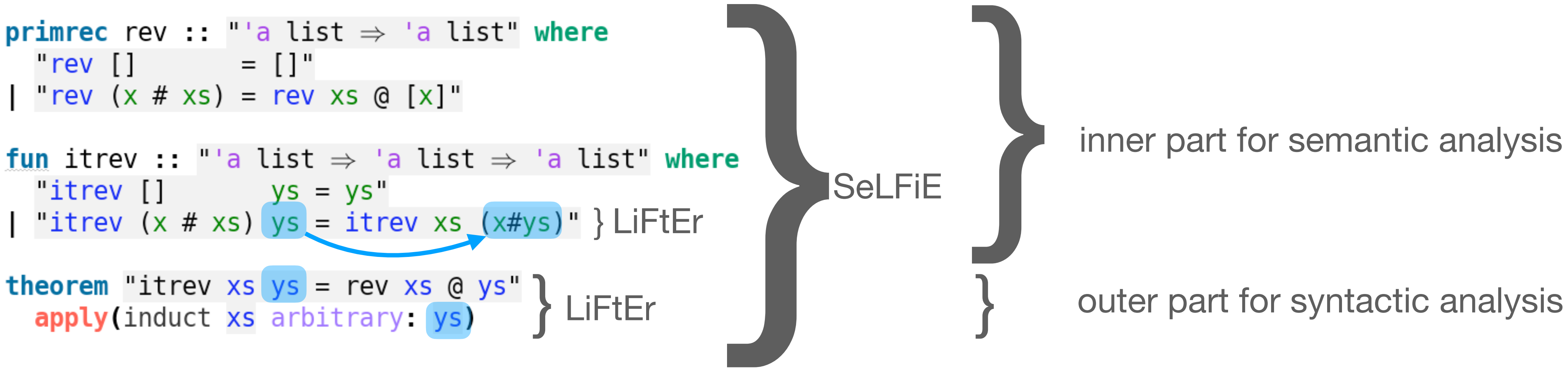
alternative good proof by induction with generalisation



Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

"Isabelle/HOL A Proof Assistant for Higher-Order Logic"
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36



```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

} LiFtEr

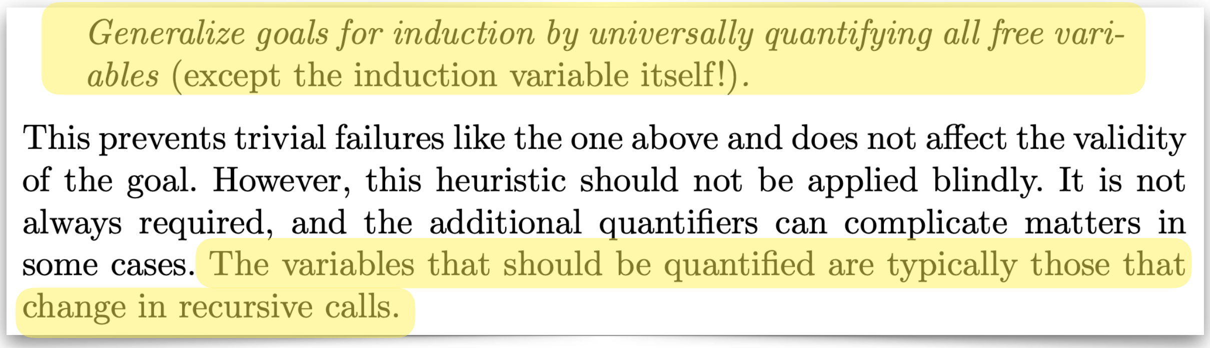
} LiFtEr

SeLFiE

inner part for semantic analysis

outer part for syntactic analysis

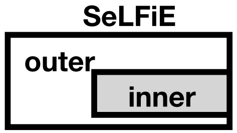
alternative good proof by induction with generalisation

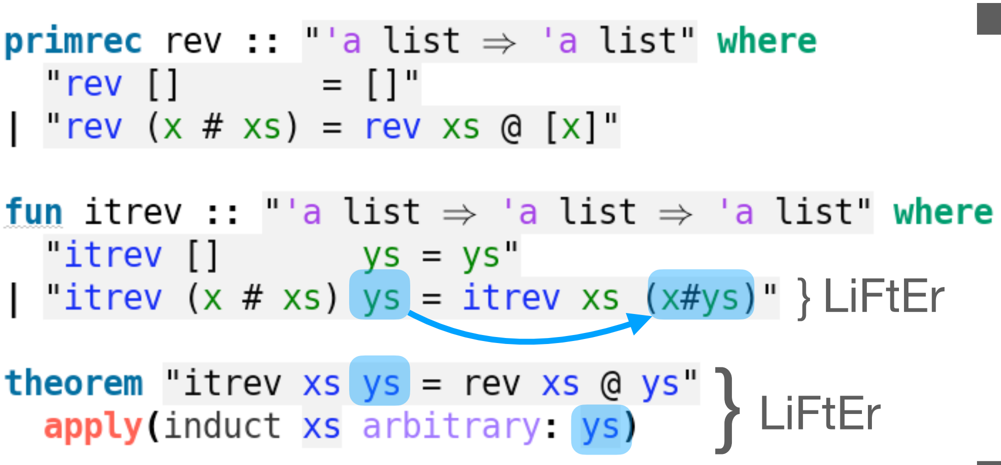


Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

"Isabelle/HOL A Proof Assistant for Higher-Order Logic"
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36





```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

} LiFtEr

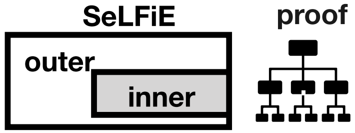
} LiFtEr

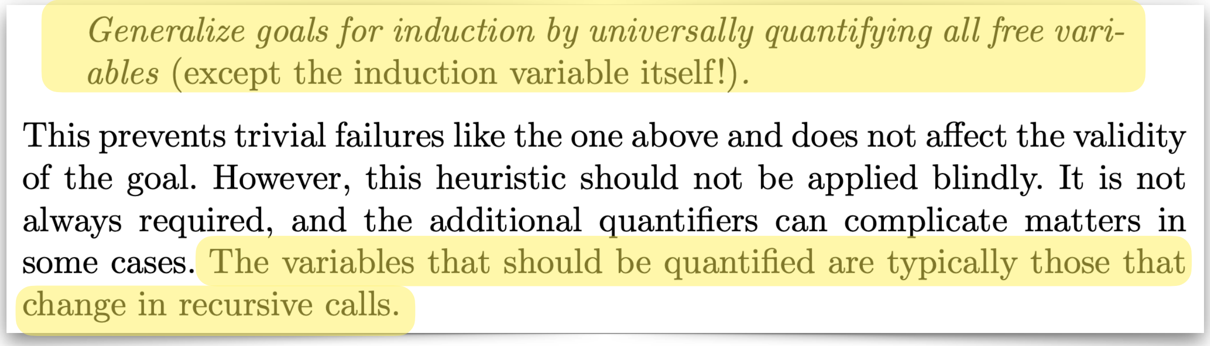
SeLFiE

inner part for semantic analysis

outer part for syntactic analysis

alternative good proof by induction with generalisation

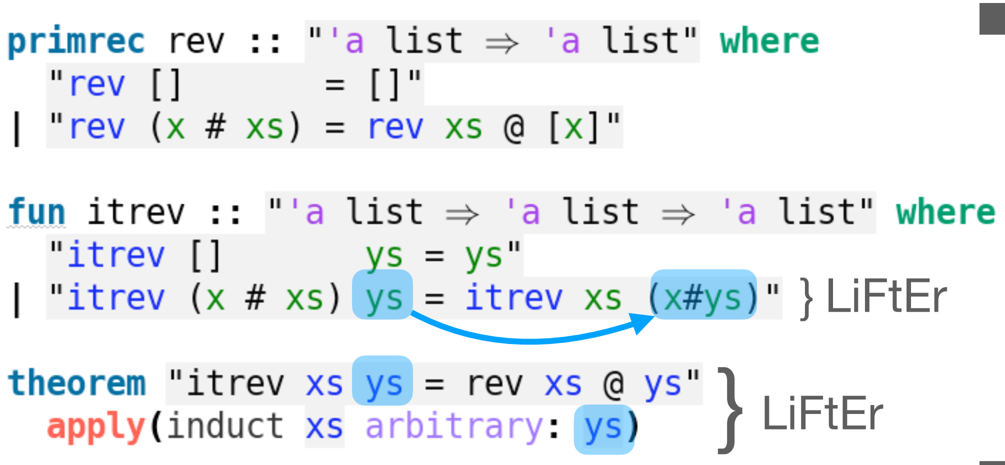




Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

"Isabelle/HOL A Proof Assistant for Higher-Order Logic"
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36



```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

} LiFtEr

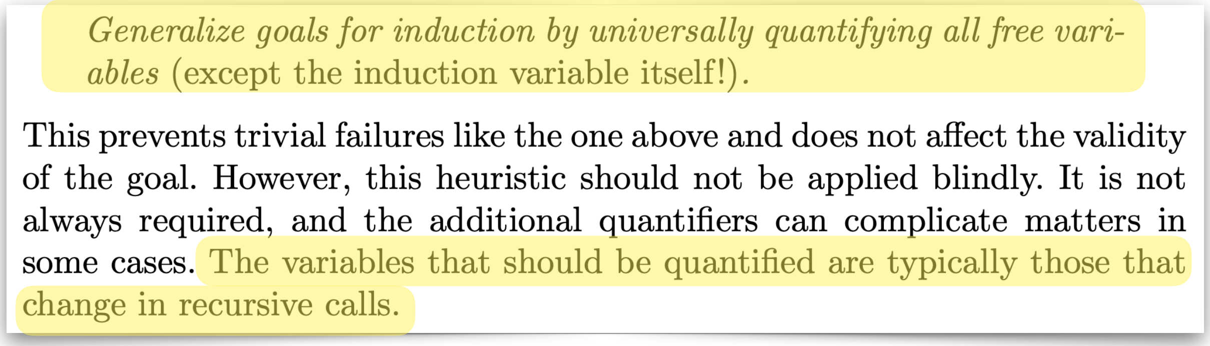
} LiFtEr

SeLFiE

inner part for semantic analysis

outer part for syntactic analysis

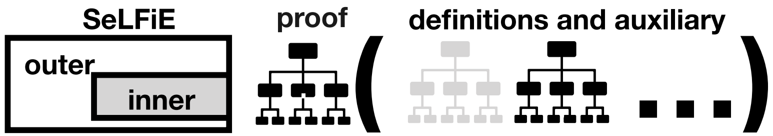
alternative good proof by induction with generalisation

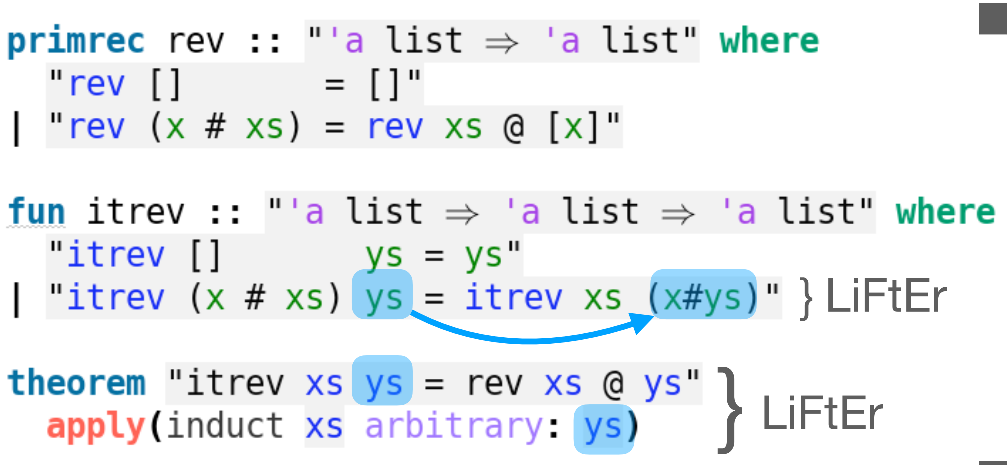


Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

"Isabelle/HOL A Proof Assistant for Higher-Order Logic"
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36





```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

} LiFtEr

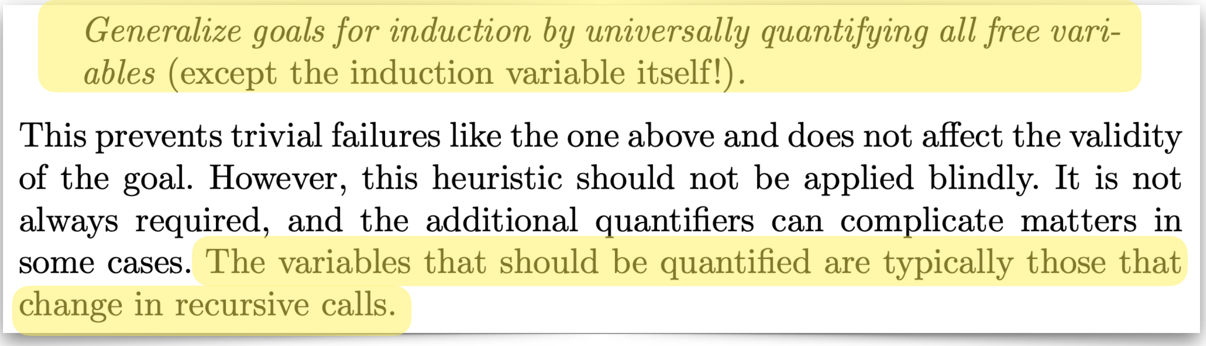
} LiFtEr

SeLFiE

inner part for semantic analysis

outer part for syntactic analysis

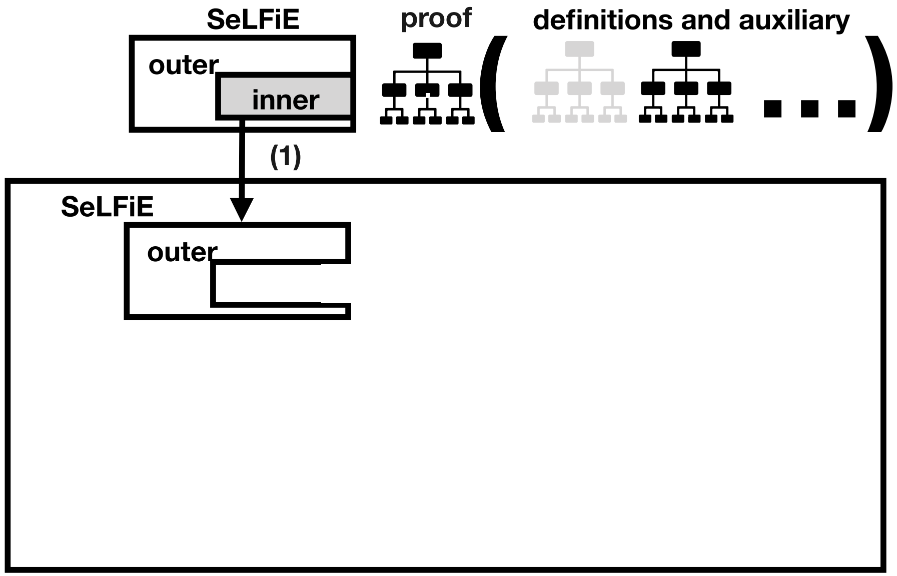
alternative good proof by induction with generalisation

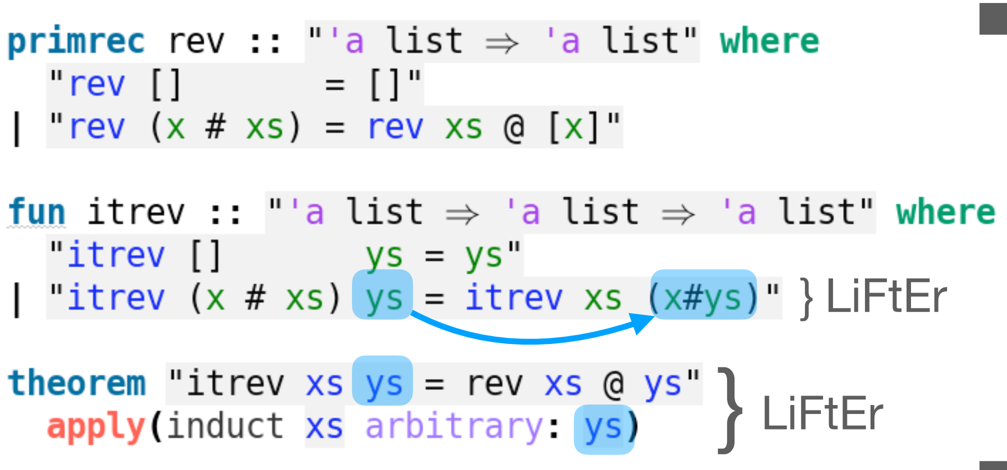


Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

"Isabelle/HOL A Proof Assistant for Higher-Order Logic"
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36





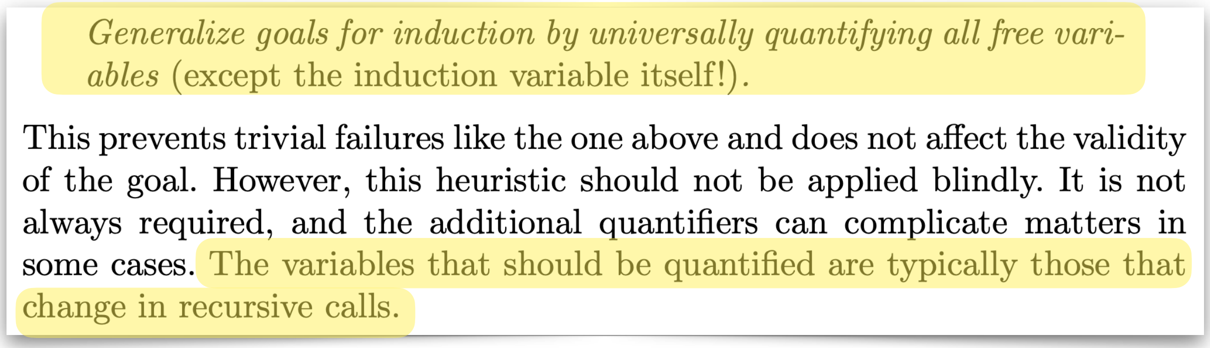
```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []        = []"
| "rev (x # xs) = rev xs @ [x]"

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

theorem "itrev xs ys = rev xs @ ys"  } LiFtEr
  apply(induct xs arbitrary: ys)
```

SeLFiE

inner part for semantic analysis

outer part for syntactic analysis

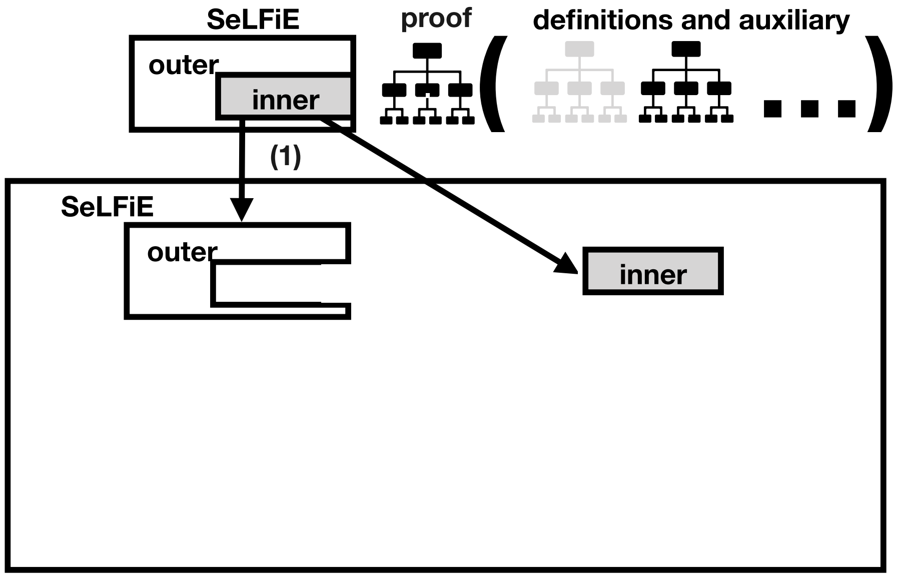
alternative good proof by induction with generalisation

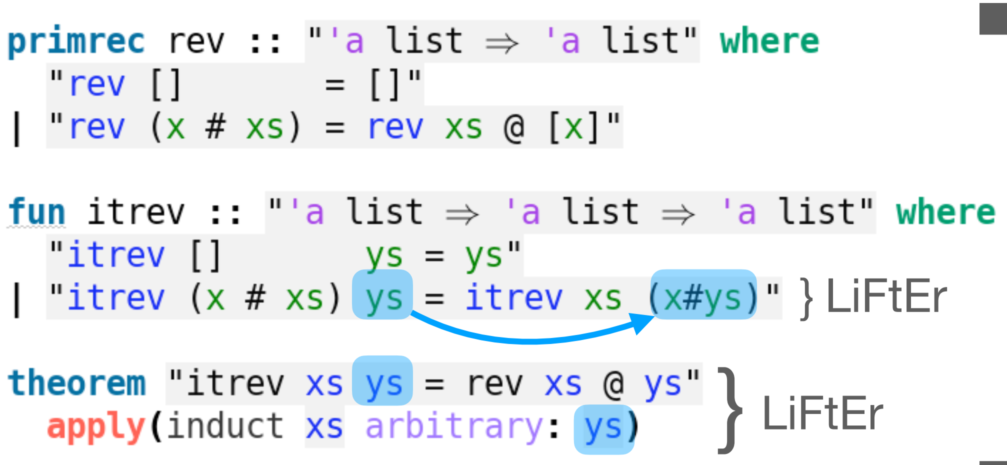


Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

“Isabelle/HOL A Proof Assistant for Higher-Order Logic”
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36





```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

} LiFtEr

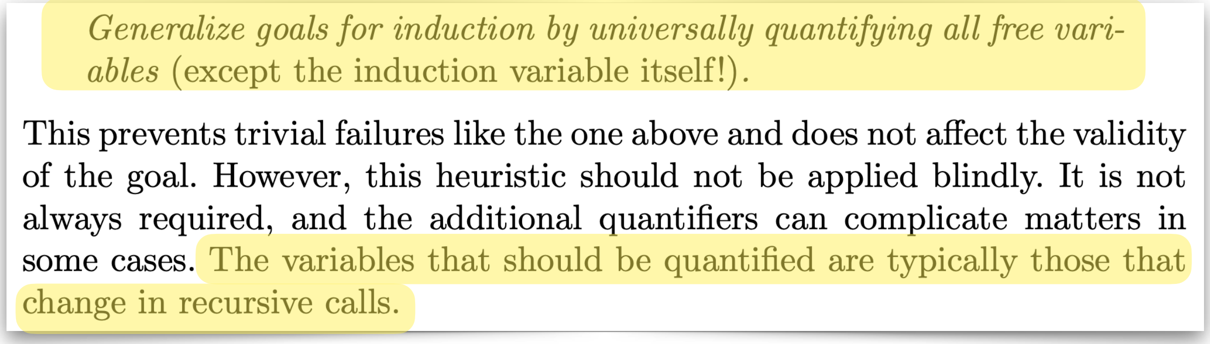
} LiFtEr

SeLFiE

inner part for semantic analysis

outer part for syntactic analysis

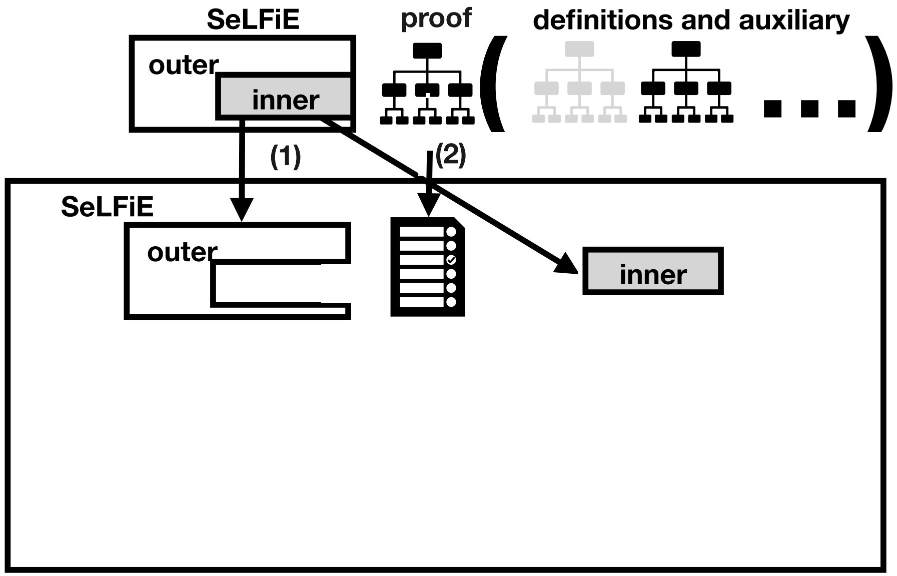
alternative good proof by induction with generalisation

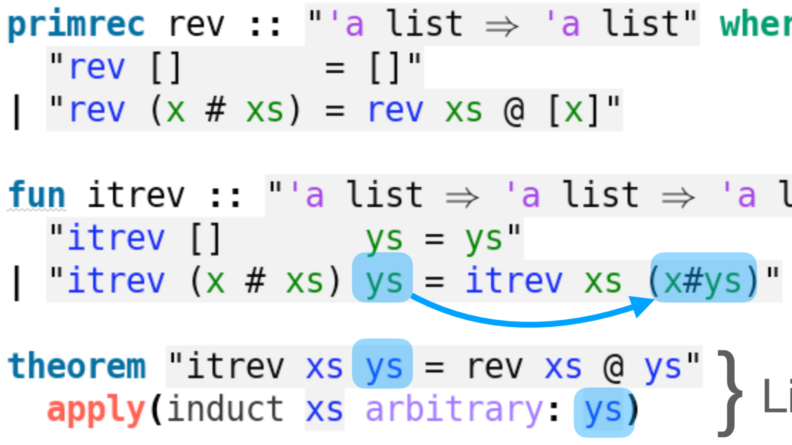


Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

"Isabelle/HOL A Proof Assistant for Higher-Order Logic"
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36





```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

} LiFtEr

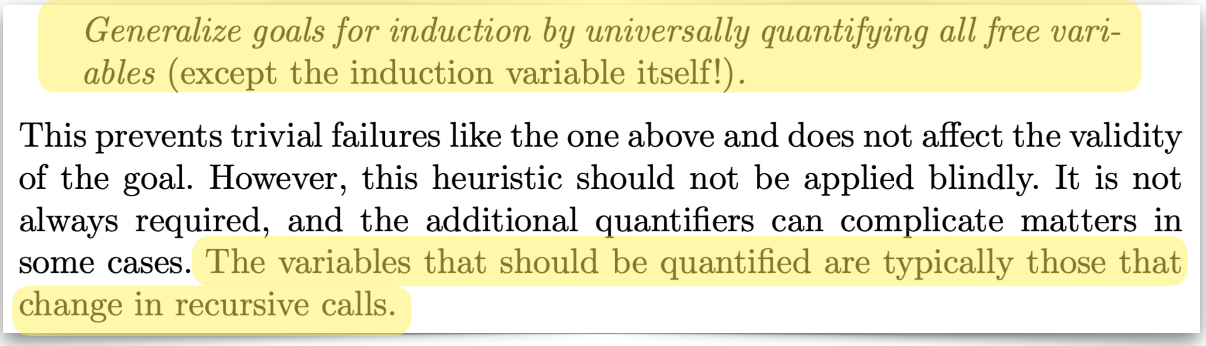
} LiFtEr

SeLFiE

inner part for semantic analysis

outer part for syntactic analysis

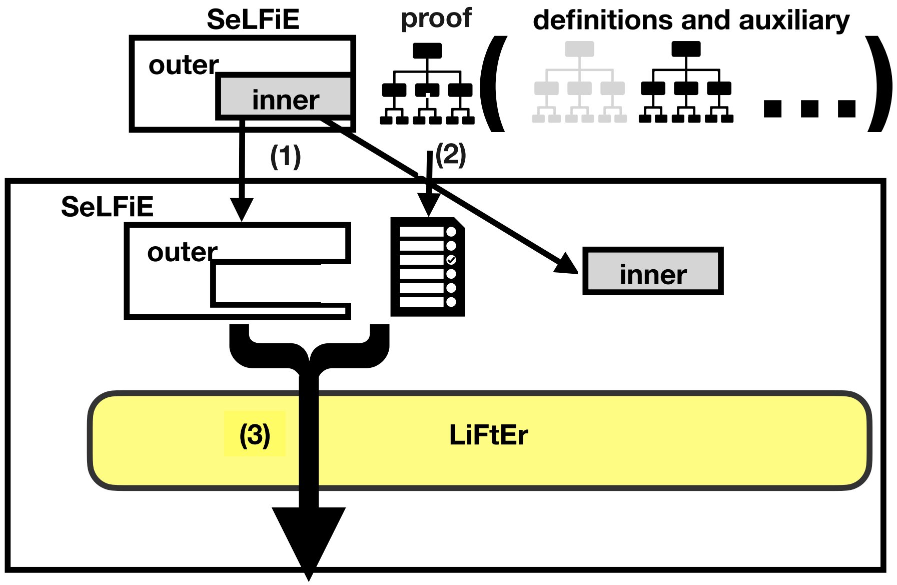
alternative good proof by induction with generalisation

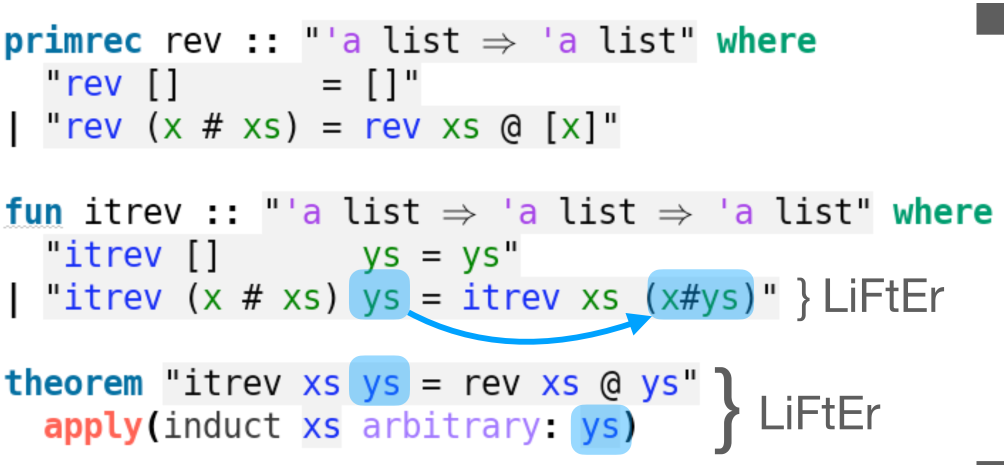


Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

"Isabelle/HOL A Proof Assistant for Higher-Order Logic"
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36





```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

} LiFtEr

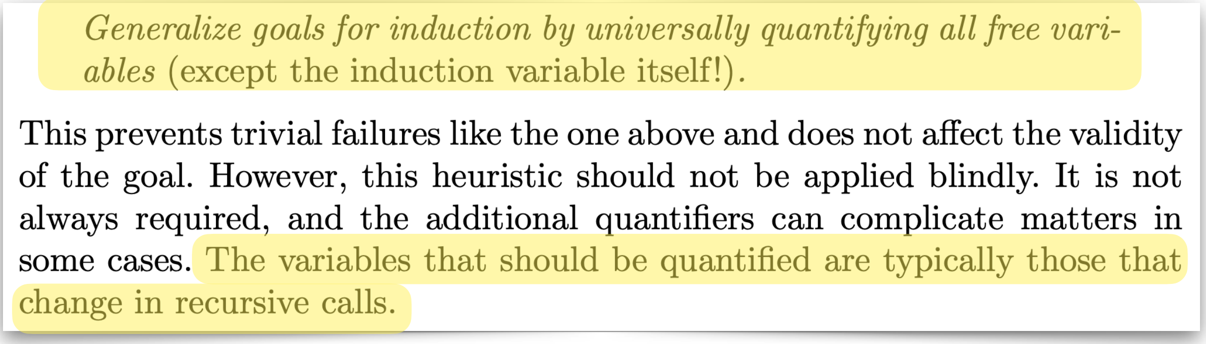
} LiFtEr

} SeLFiE

} inner part for semantic analysis

} outer part for syntactic analysis

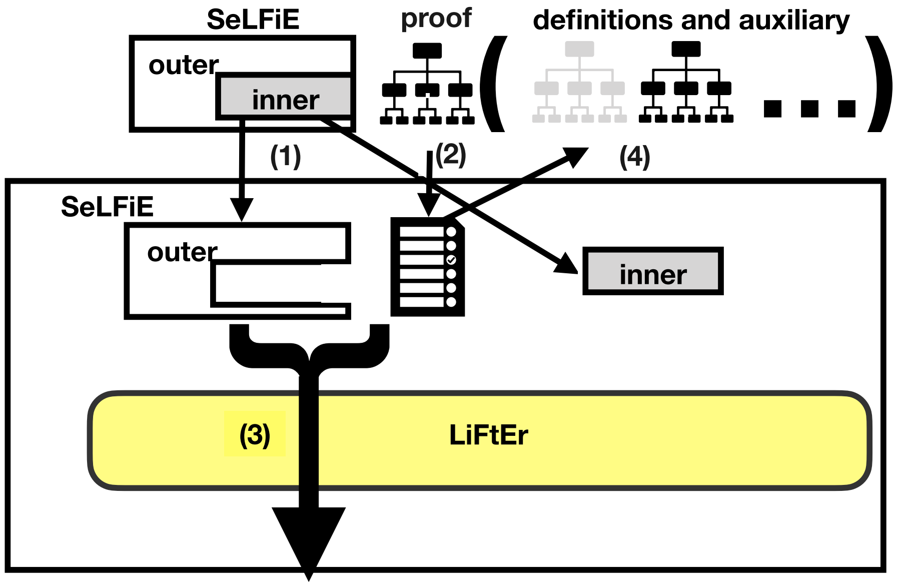
alternative good proof by induction with generalisation

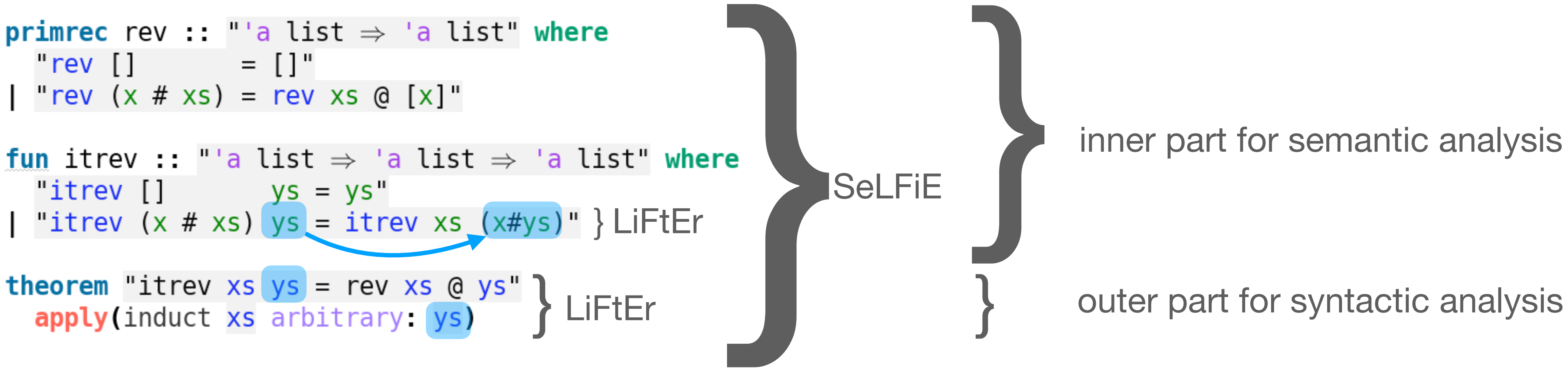


Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

"Isabelle/HOL A Proof Assistant for Higher-Order Logic"
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36

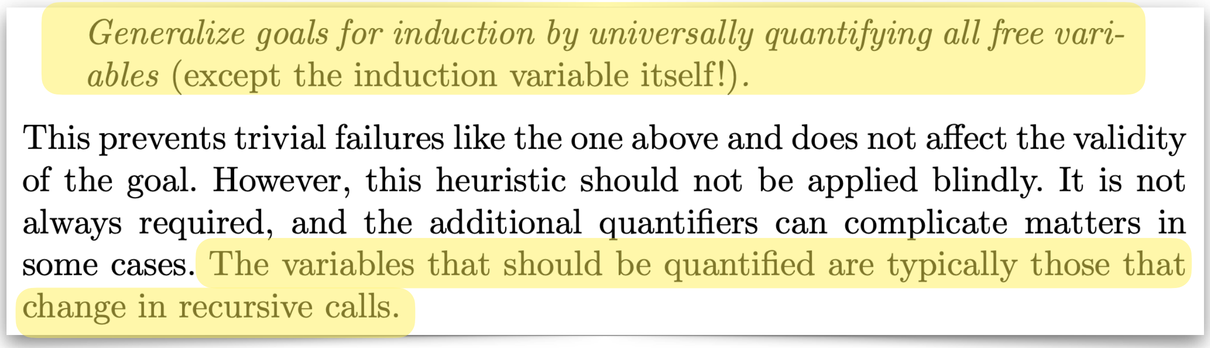




```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []        = []"
| "rev (x # xs) = rev xs @ [x]"

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

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

LiFtEr
LiFtEr
SeLFiE
inner part for semantic analysis
outer part for syntactic analysis

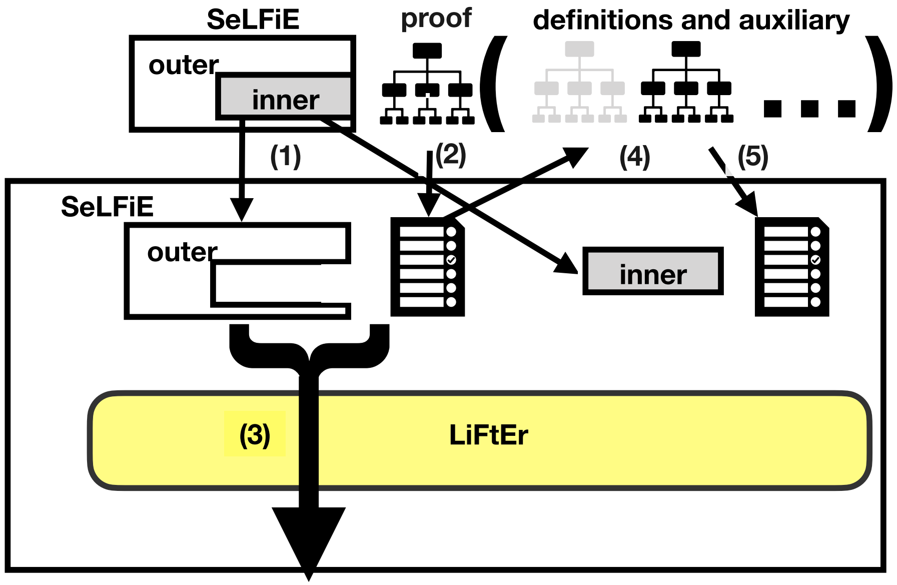
alternative good proof by induction with generalisation

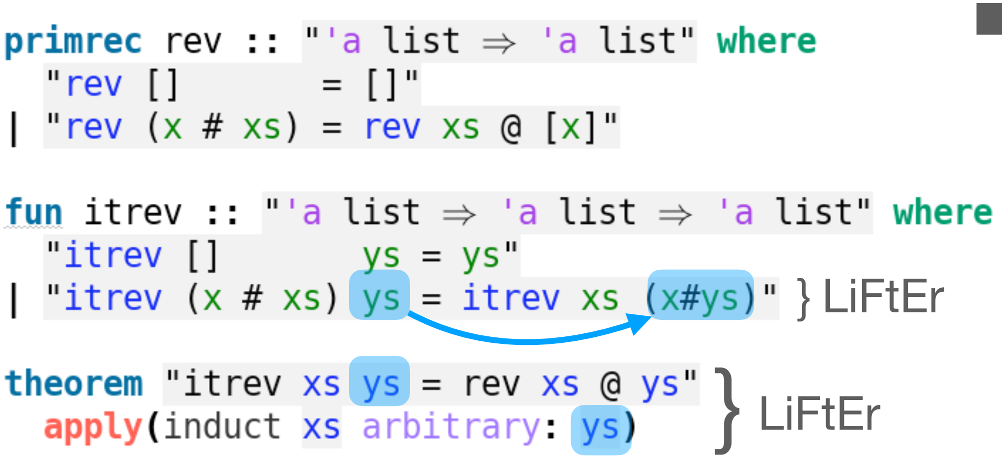


Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

“Isabelle/HOL A Proof Assistant for Higher-Order Logic”
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36





```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

} LiFtEr

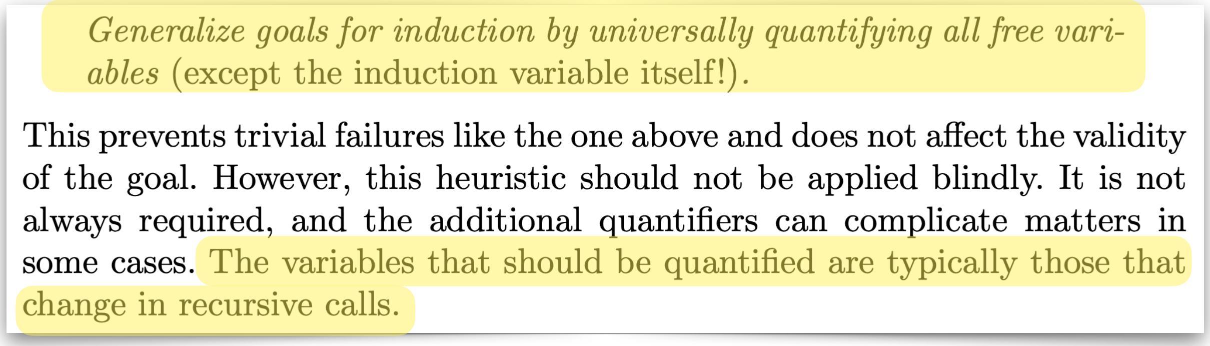
} LiFtEr

SeLFiE

inner part for semantic analysis

outer part for syntactic analysis

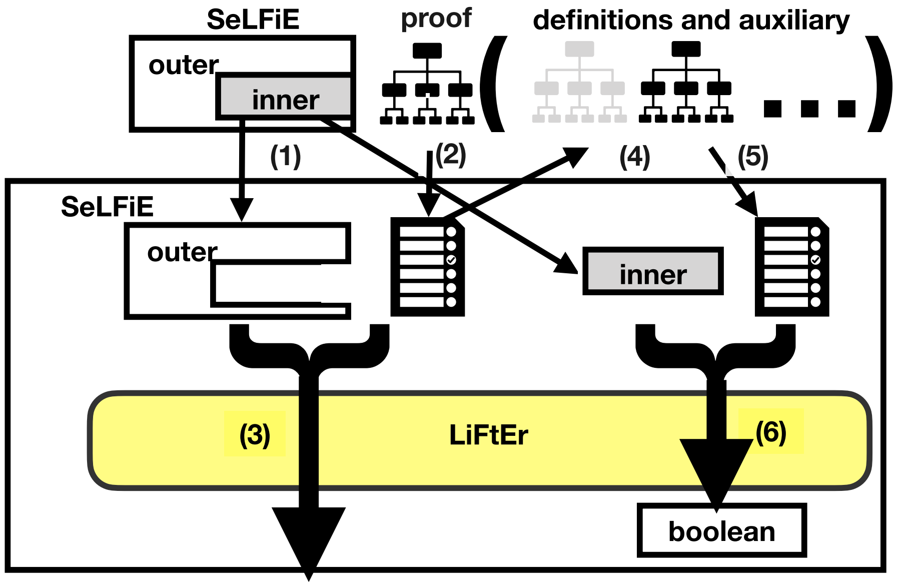
alternative good proof by induction with generalisation

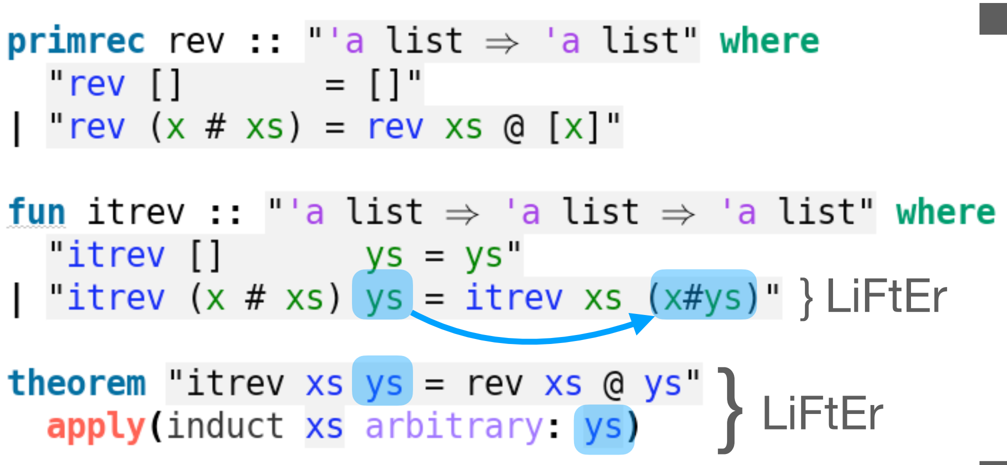


Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

“Isabelle/HOL A Proof Assistant for Higher-Order Logic”
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36





```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

} LiFtEr

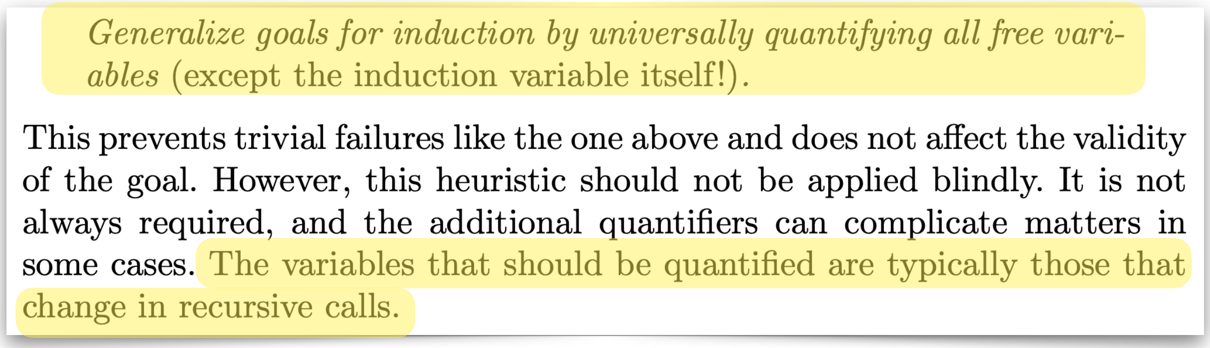
} LiFtEr

SeLFiE

inner part for semantic analysis

outer part for syntactic analysis

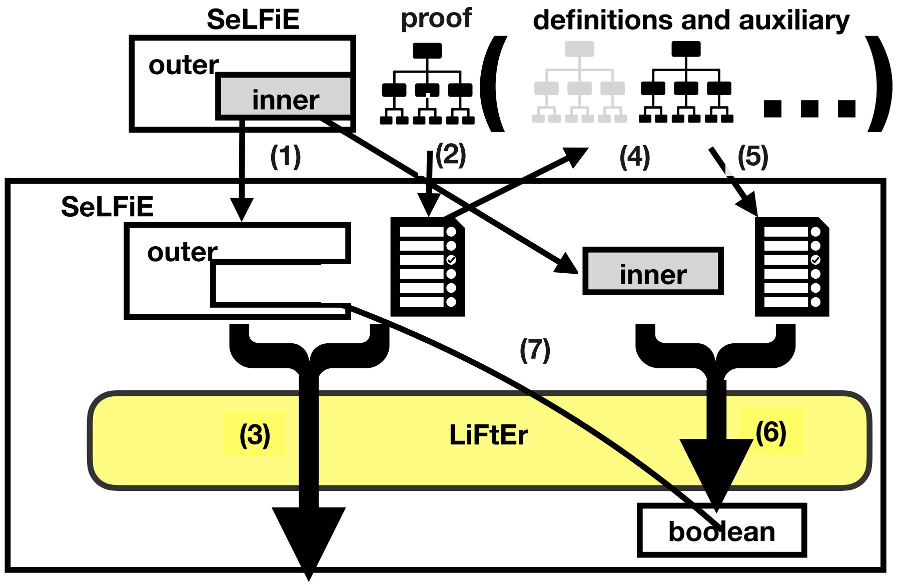
alternative good proof by induction with generalisation

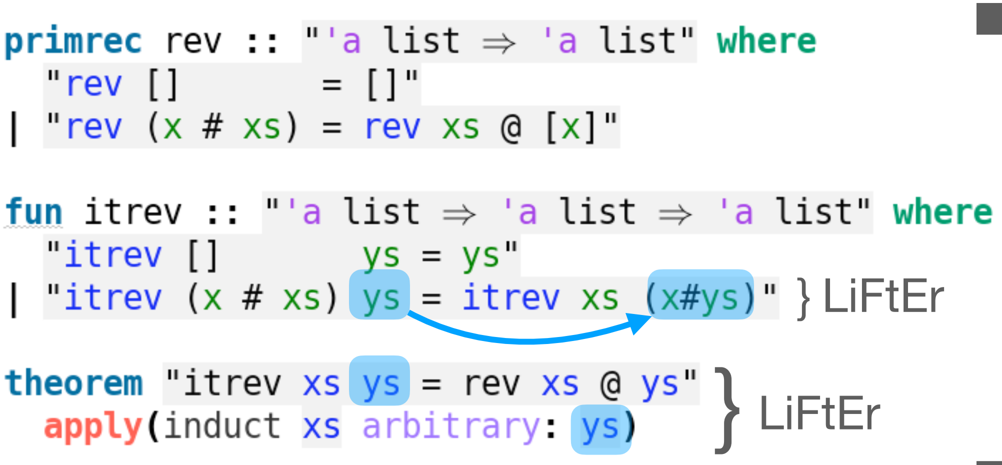


Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

“Isabelle/HOL A Proof Assistant for Higher-Order Logic”
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36





```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

} LiFtEr

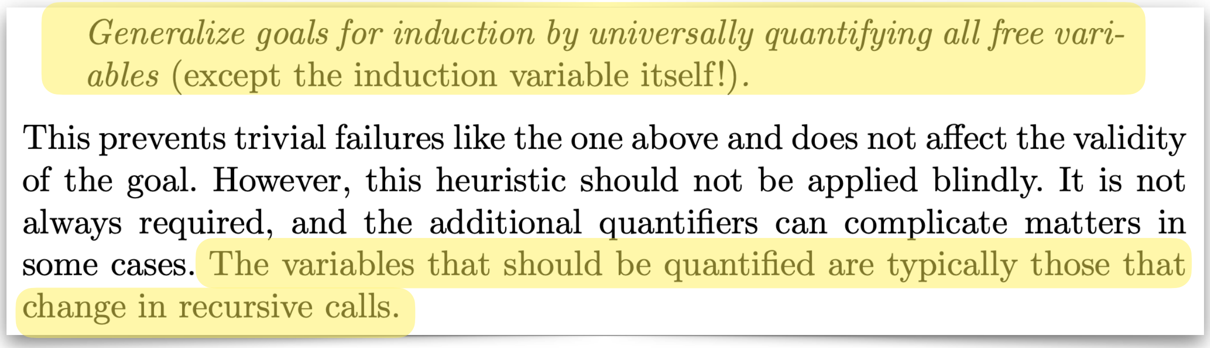
} LiFtEr

} SeLFiE

} inner part for semantic analysis

} outer part for syntactic analysis

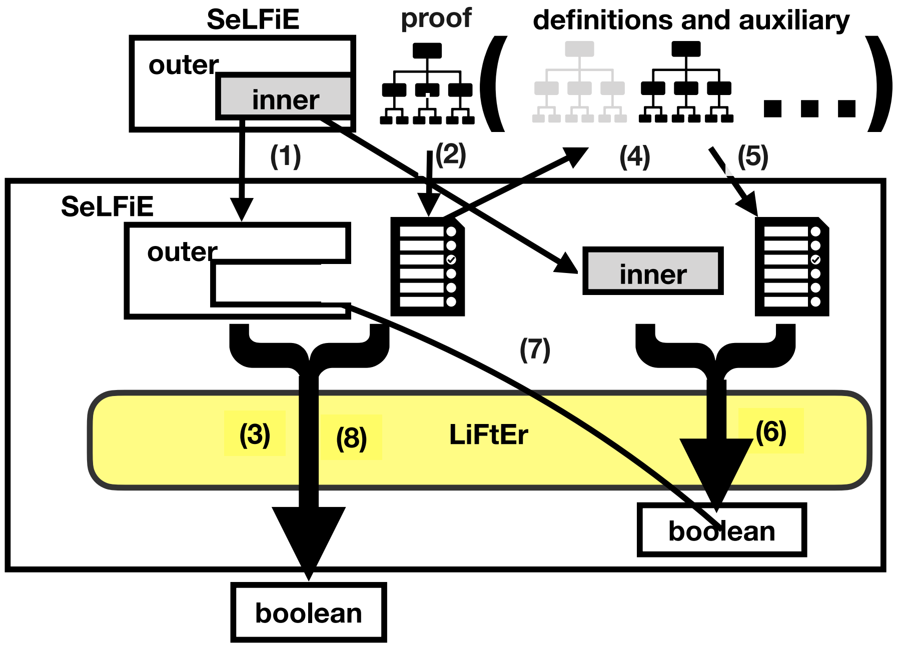
alternative good proof by induction with generalisation

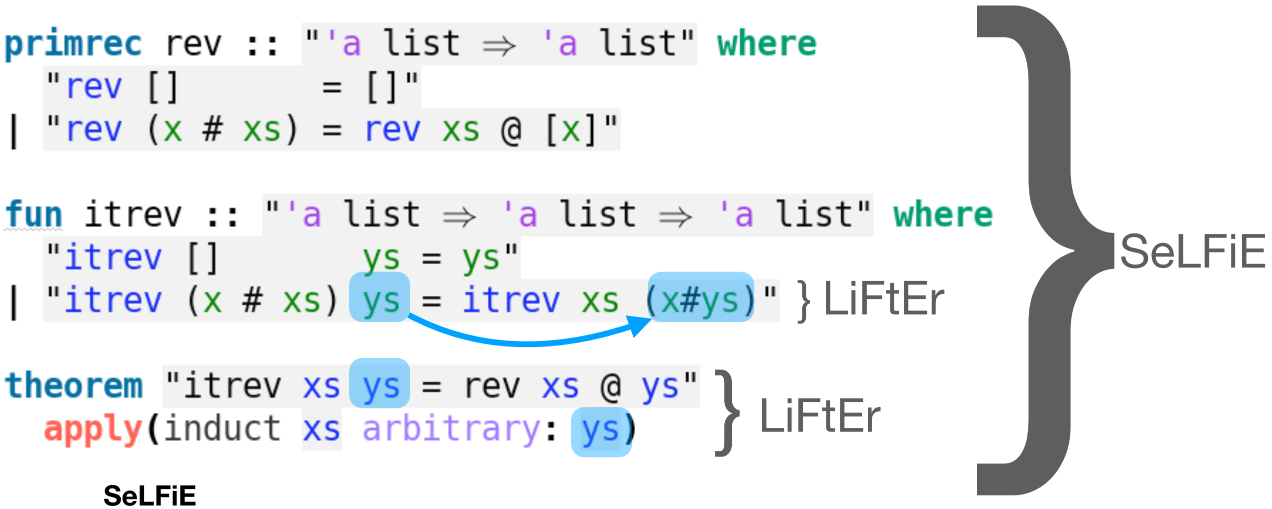


Generalize goals for induction by universally quantifying all free variables (except the induction variable itself!).

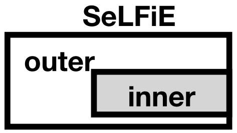
This prevents trivial failures like the one above and does not affect the validity of the goal. However, this heuristic should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that should be quantified are typically those that change in recursive calls.

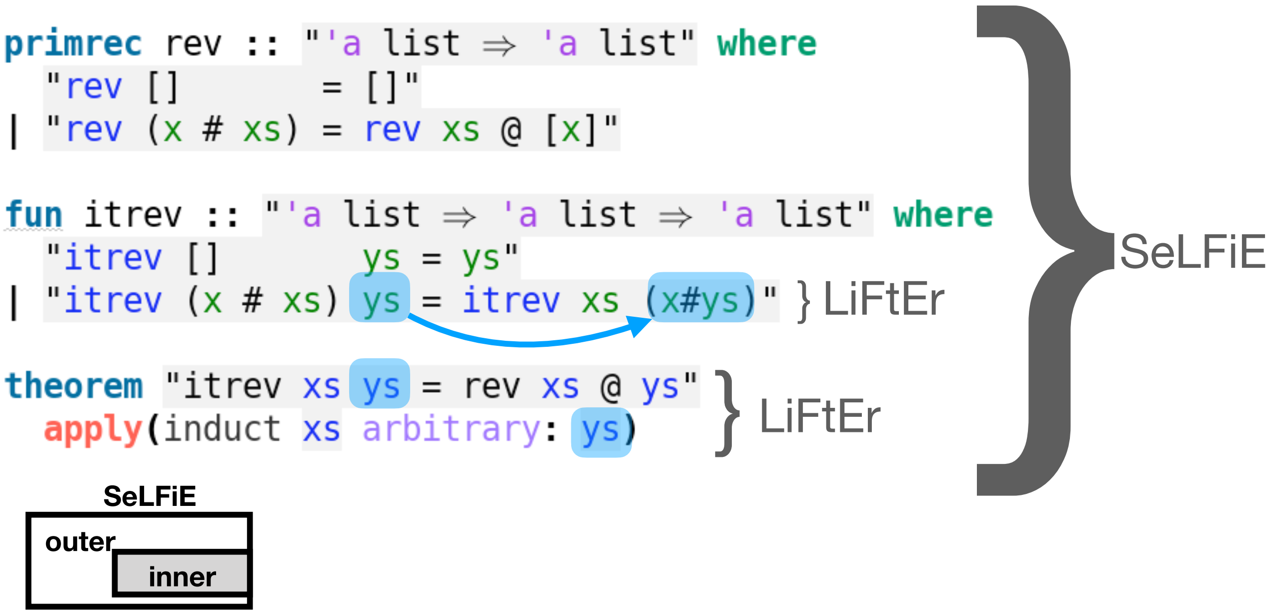
“Isabelle/HOL A Proof Assistant for Higher-Order Logic”
Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36





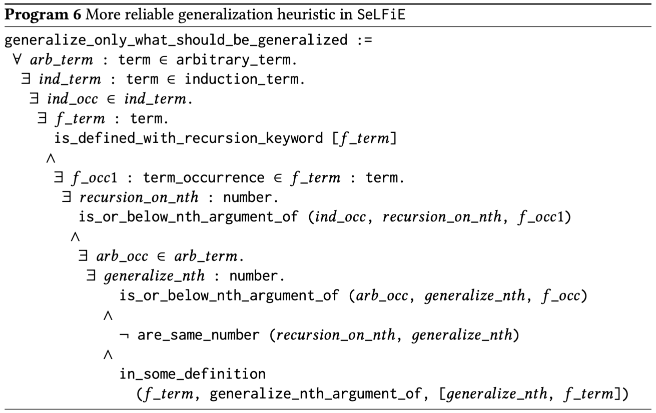
```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []        = []"
| "rev (x # xs) = rev xs @ [x]"

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

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```





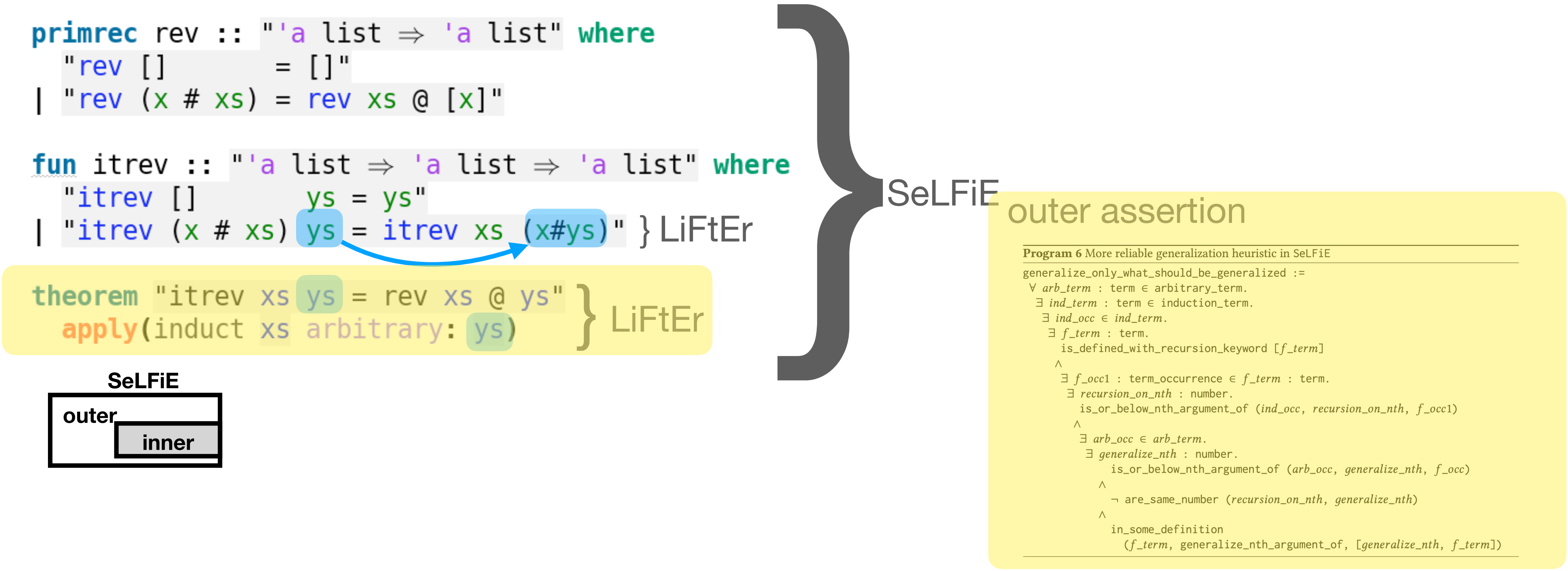
```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []        = []"
| "rev (x # xs) = rev xs @ [x]"

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

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

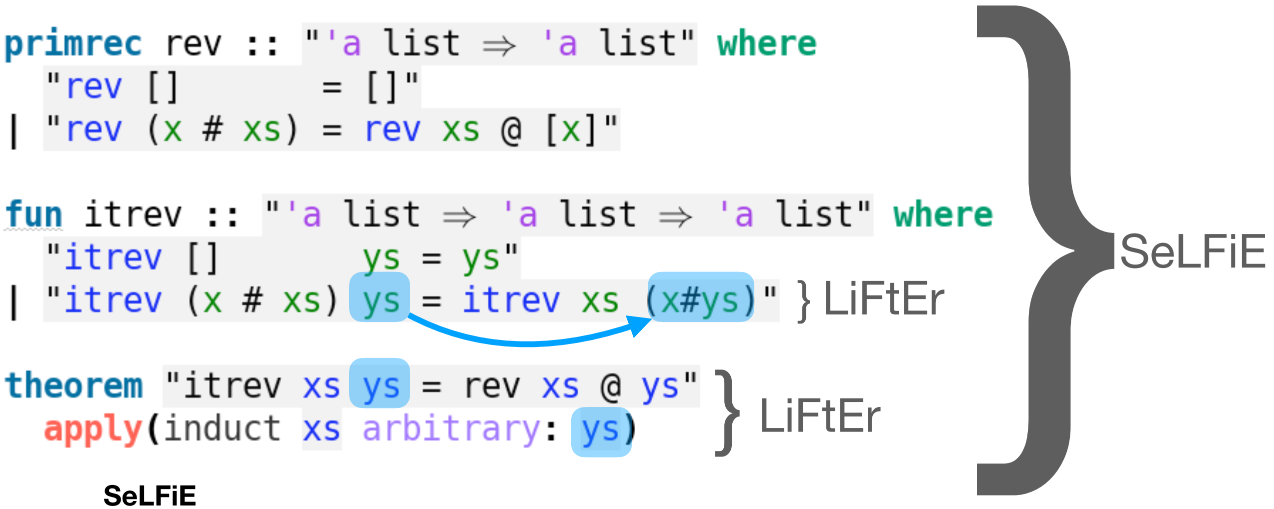
outer assertion



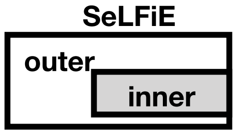
Program 6 More reliable generalization heuristic in SeLFiE

```
generalize_only_what_should_be_generalized :=
 ∀ arb_term : term ∈ arbitrary_term.
  ∃ ind_term : term ∈ induction_term.
   ∃ ind_occ ∈ ind_term.
    ∃ f_term : term.
      is_defined_with_recursion_keyword [f_term]
     ∧
      ∃ f_occ1 : term_occurrence ∈ f_term : term.
       ∃ recursion_on_nth : number.
         is_or_below_nth_argument_of (ind_occ, recursion_on_nth, f_occ1)
        ∧
         ∃ arb_occ ∈ arb_term.
          ∃ generalize_nth : number.
            is_or_below_nth_argument_of (arb_occ, generalize_nth, f_occ)
           ∧
            ¬ are_same_number (recursion_on_nth, generalize_nth)
           ∧
            in_some_definition
              (f_term, generalize_nth_argument_of, [generalize_nth, f_term])
```

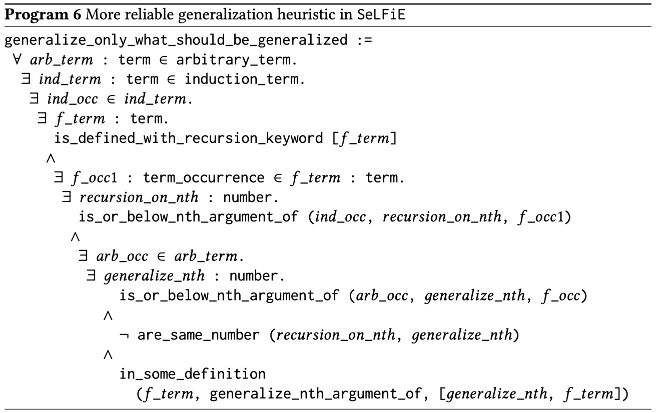




```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []        = []"
| "rev (x # xs) = rev xs @ [x]"

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

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

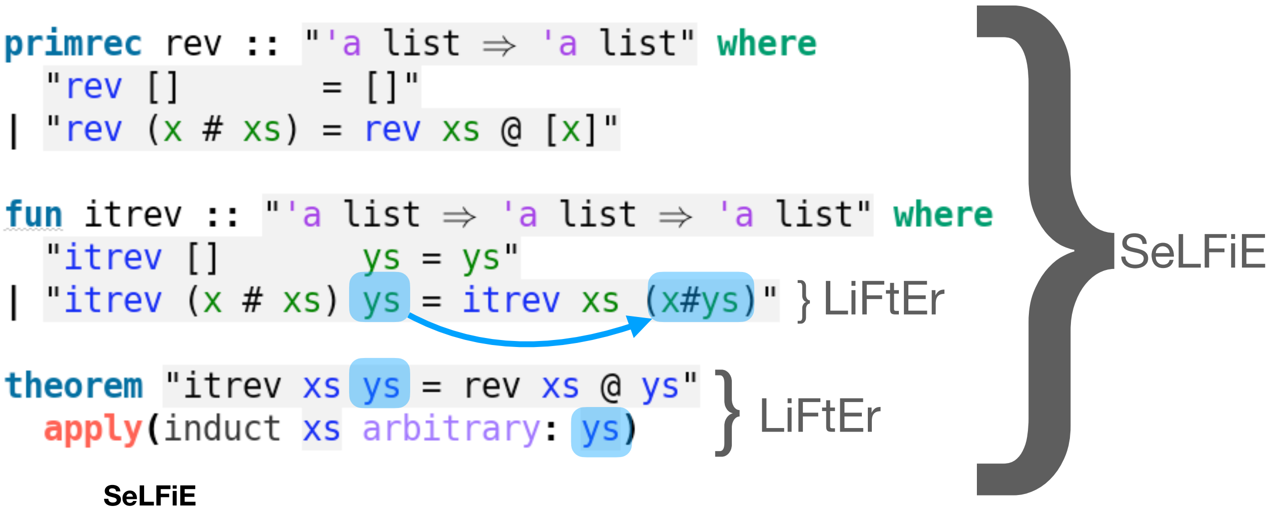


outer assertion



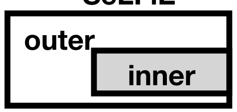
Program 6 More reliable generalization heuristic in SeLFiE

```
generalize_only_what_should_be_generalized :=
 ∀ arb_term : term ∈ arbitrary_term.
  ∃ ind_term : term ∈ induction_term.
   ∃ ind_occ ∈ ind_term.
    ∃ f_term : term.
      is_defined_with_recursion_keyword [f_term]
     ∧
      ∃ f_occ1 : term_occurrence ∈ f_term : term.
       ∃ recursion_on_nth : number.
         is_or_below_nth_argument_of (ind_occ, recursion_on_nth, f_occ1)
        ∧
         ∃ arb_occ ∈ arb_term.
          ∃ generalize_nth : number.
              is_or_below_nth_argument_of (arb_occ, generalize_nth, f_occ)
            ∧
              ¬ are_same_number (recursion_on_nth, generalize_nth)
            ∧
              in_some_definition
                (f_term, generalize_nth_argument_of, [generalize_nth, f_term])
```

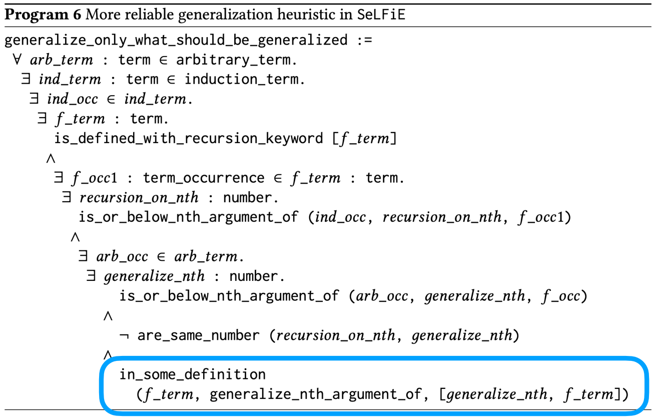


```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []        = []"
| "rev (x # xs) = rev xs @ [x]"

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

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

SeLFiE



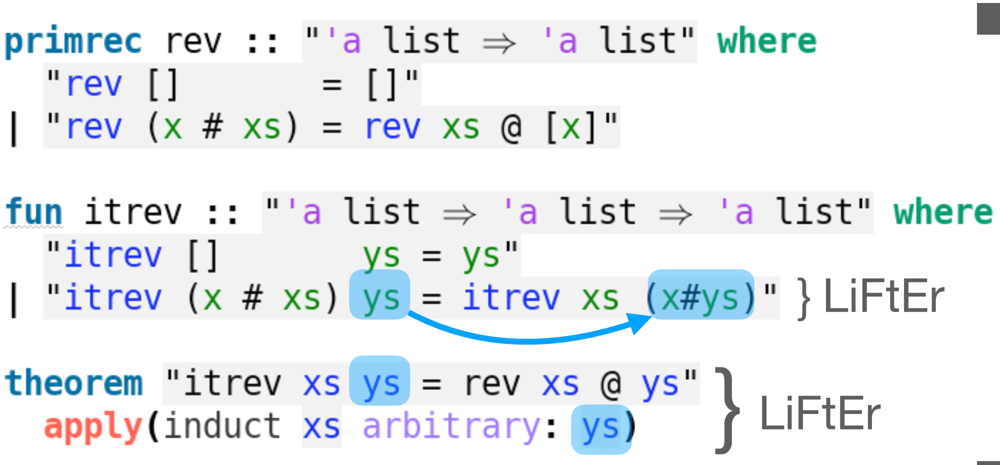
outer assertion



Program 6 More reliable generalization heuristic in SeLFiE

```
generalize_only_what_should_be_generalized :=
 ∀ arb_term : term ∈ arbitrary_term.
  ∃ ind_term : term ∈ induction_term.
   ∃ ind_occ ∈ ind_term.
    ∃ f_term : term.
      is_defined_with_recursion_keyword [f_term]
     ∧
      ∃ f_occ1 : term_occurrence ∈ f_term : term.
       ∃ recursion_on_nth : number.
         is_or_below_nth_argument_of (ind_occ, recursion_on_nth, f_occ1)
        ∧
         ∃ arb_occ ∈ arb_term.
          ∃ generalize_nth : number.
              is_or_below_nth_argument_of (arb_occ, generalize_nth, f_occ)
            ∧
              ¬ are_same_number (recursion_on_nth, generalize_nth)
            ∧
              in_some_definition
                (f_term, generalize_nth_argument_of, [generalize_nth, f_term])
```

in_some_definition (
f_term,
generalized_nth_argument_of,
[genearlize_nth, f_term])

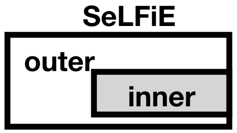


```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []        = []"
| "rev (x # xs) = rev xs @ [x]"

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

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

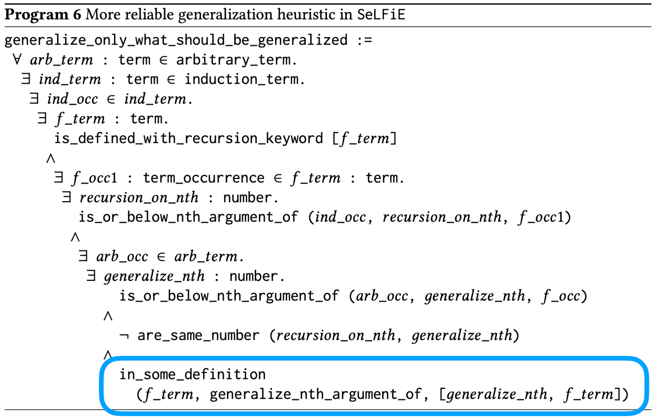
} LiFtEr

} LiFtEr

SeLFiE



outer assertion

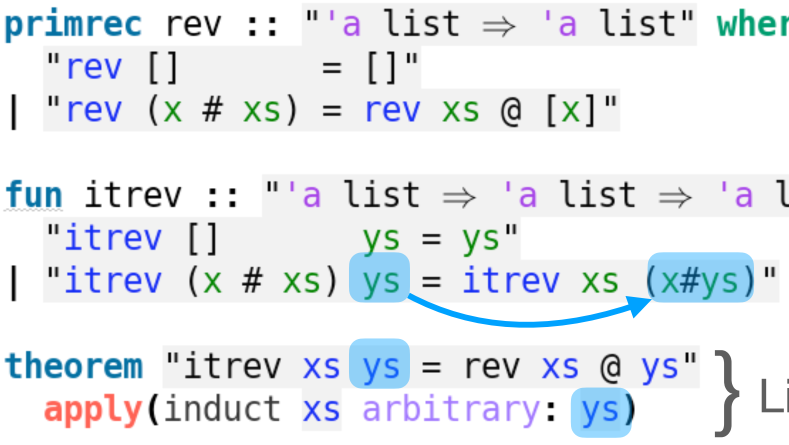


Program 6 More reliable generalization heuristic in SeLFiE

```
generalize_only_what_should_be_generalized :=
 ∀ arb_term : term ∈ arbitrary_term.
  ∃ ind_term : term ∈ induction_term.
   ∃ ind_occ ∈ ind_term.
    ∃ f_term : term.
      is_defined_with_recursion_keyword [f_term]
     ∧
      ∃ f_occ1 : term_occurrence ∈ f_term : term.
       ∃ recursion_on_nth : number.
         is_or_below_nth_argument_of (ind_occ, recursion_on_nth, f_occ1)
        ∧
         ∃ arb_occ ∈ arb_term.
          ∃ generalize_nth : number.
              is_or_below_nth_argument_of (arb_occ, generalize_nth, f_occ)
            ∧
              ¬ are_same_number (recursion_on_nth, generalize_nth)
            ∧
              in_some_definition
                (f_term, generalize_nth_argument_of, [generalize_nth, f_term])
```

in_some_definition (
f_term,
generalized_nth_argument_of,
[genearlize_nth, f_term])

<- key to look up the defining clauses

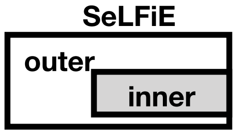


```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []         = []"
| "rev (x # xs) = rev xs @ [x]"

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

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

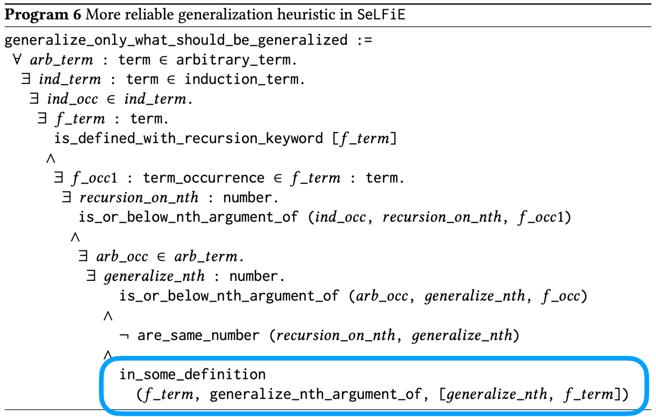
} LiFtEr

} LiFtEr

} SeLFiE



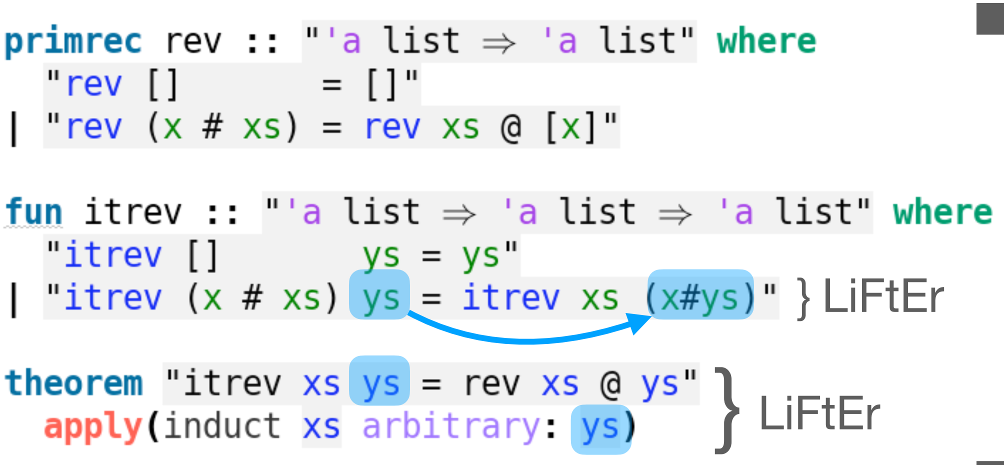
outer assertion



Program 6 More reliable generalization heuristic in SeLFiE

```
generalize_only_what_should_be_generalized :=
 ∀ arb_term : term ∈ arbitrary_term.
  ∃ ind_term : term ∈ induction_term.
   ∃ ind_occ ∈ ind_term.
    ∃ f_term : term.
      is_defined_with_recursion_keyword [f_term]
     ∧
      ∃ f_occ1 : term_occurrence ∈ f_term : term.
       ∃ recursion_on_nth : number.
         is_or_below_nth_argument_of (ind_occ, recursion_on_nth, f_occ1)
        ∧
         ∃ arb_occ ∈ arb_term.
          ∃ generalize_nth : number.
              is_or_below_nth_argument_of (arb_occ, generalize_nth, f_occ)
             ∧
              ¬ are_same_number (recursion_on_nth, generalize_nth)
             ∧
              in_some_definition
                (f_term, generalize_nth_argument_of, [generalize_nth, f_term])
```

in_some_definition (
f_term, <- key to look up the defining clauses
generalized_nth_argument_of, <- name of inner_assertion
[genearlize_nth, f_term])

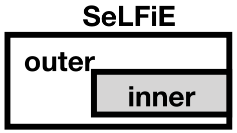


```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []        = []"
| "rev (x # xs) = rev xs @ [x]"

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

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

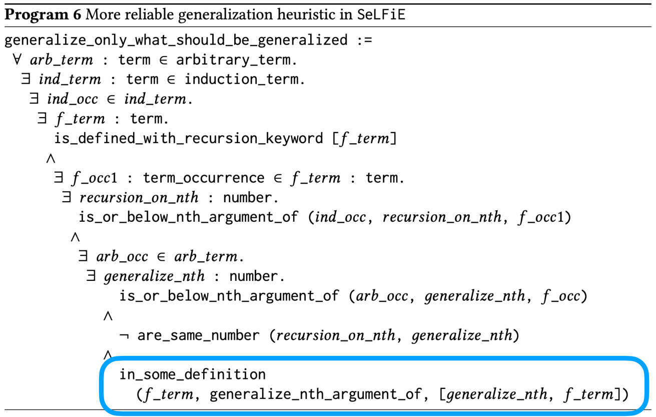
} LiFtEr

} LiFtEr

} SeLFiE



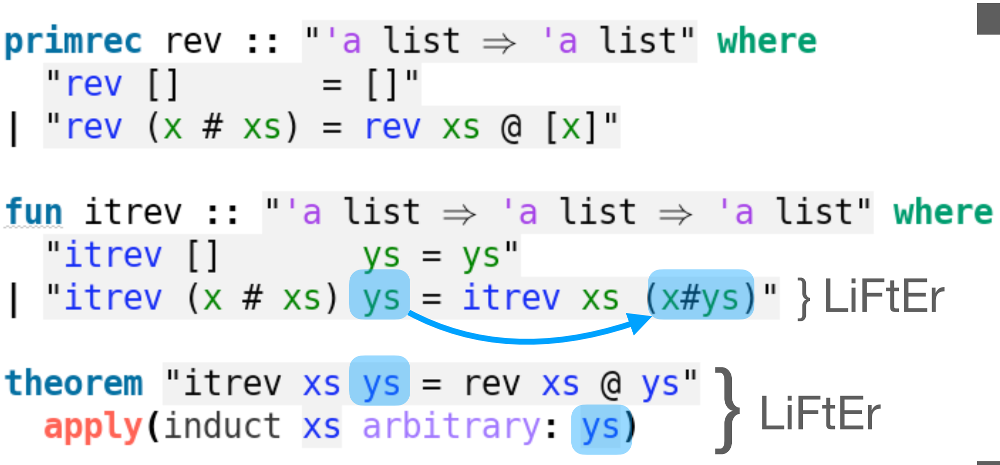
outer assertion



Program 6 More reliable generalization heuristic in SeLFiE

```
generalize_only_what_should_be_generalized :=
 ∀ arb_term : term ∈ arbitrary_term.
  ∃ ind_term : term ∈ induction_term.
   ∃ ind_occ ∈ ind_term.
    ∃ f_term : term.
      is_defined_with_recursion_keyword [f_term]
     ∧
      ∃ f_occ1 : term_occurrence ∈ f_term : term.
       ∃ recursion_on_nth : number.
         is_or_below_nth_argument_of (ind_occ, recursion_on_nth, f_occ1)
        ∧
         ∃ arb_occ ∈ arb_term.
          ∃ generalize_nth : number.
              is_or_below_nth_argument_of (arb_occ, generalize_nth, f_occ)
             ∧
              ¬ are_same_number (recursion_on_nth, generalize_nth)
             ∧
              in_some_definition
                (f_term, generalize_nth_argument_of, [generalize_nth, f_term])
```

in_some_definition (
f_term, <- key to look up the defining clauses
generalized_nth_argument_of, <- name of inner_assertion
[genearlize_nth, f_term]) <- arguments from outer-to-inner

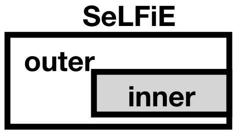


```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []        = []"
| "rev (x # xs) = rev xs @ [x]"

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

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

} LiFtEr

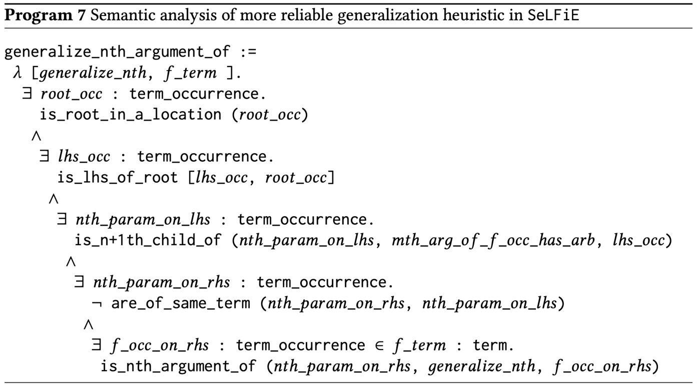
} LiFtEr

SeLFiE



inner assertion

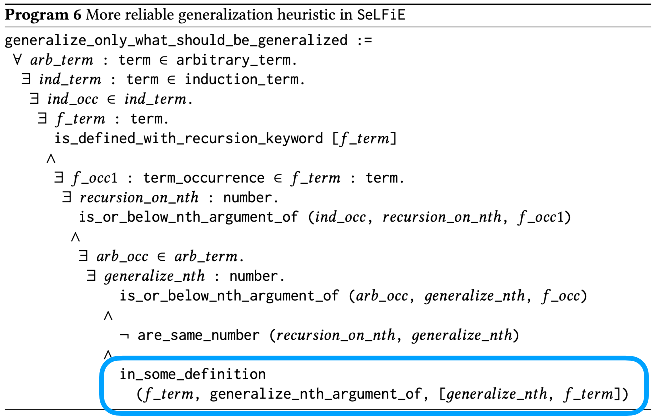
(= generalized_nth_argument_of)



Program 7 Semantic analysis of more reliable generalization heuristic in SeLFiE

```
generalize_nth_argument_of :=
 λ [generalize_nth, f_term ].
  ∃ root_occ : term_occurrence.
    is_root_in_a_location (root_occ)
   ∧
    ∃ lhs_occ : term_occurrence.
      is_lhs_of_root [lhs_occ, root_occ]
     ∧
      ∃ nth_param_on_lhs : term_occurrence.
        is_n+1th_child_of (nth_param_on_lhs, mth_arg_of_f_occ_has_arb, lhs_occ)
       ∧
        ∃ nth_param_on_rhs : term_occurrence.
          ¬ are_of_same_term (nth_param_on_rhs, nth_param_on_lhs)
         ∧
          ∃ f_occ_on_rhs : term_occurrence ∈ f_term : term.
           is_nth_argument_of (nth_param_on_rhs, generalize_nth, f_occ_on_rhs)
```

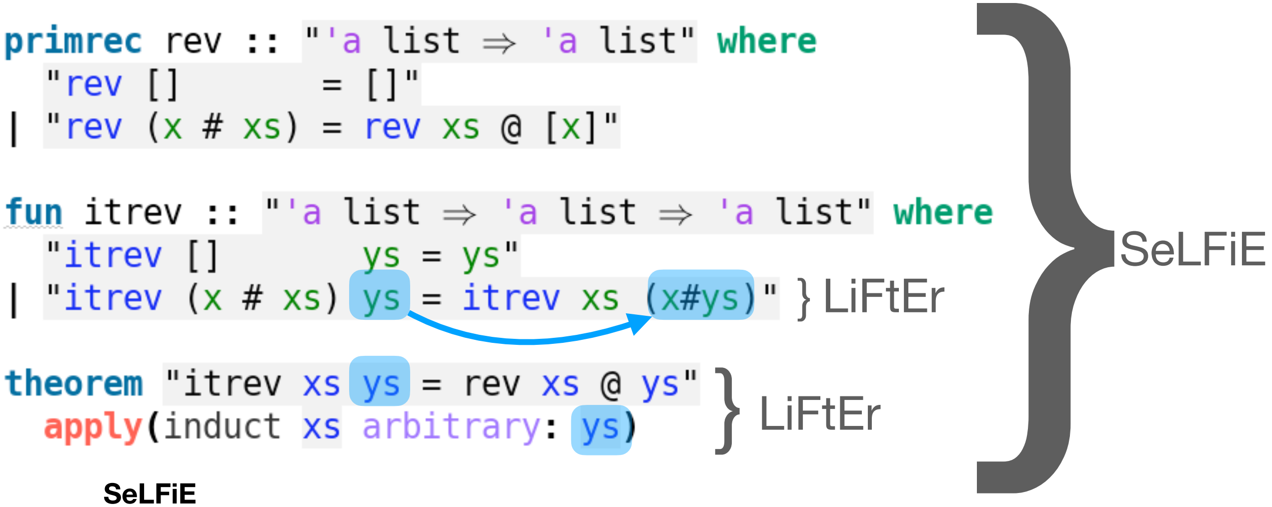
outer assertion



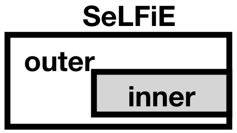
Program 6 More reliable generalization heuristic in SeLFiE

```
generalize_only_what_should_be_generalized :=
 ∀ arb_term : term ∈ arbitrary_term.
  ∃ ind_term : term ∈ induction_term.
   ∃ ind_occ ∈ ind_term.
    ∃ f_term : term.
      is_defined_with_recursion_keyword [f_term]
     ∧
      ∃ f_occ1 : term_occurrence ∈ f_term : term.
       ∃ recursion_on_nth : number.
         is_or_below_nth_argument_of (ind_occ, recursion_on_nth, f_occ1)
        ∧
         ∃ arb_occ ∈ arb_term.
          ∃ generalize_nth : number.
            is_or_below_nth_argument_of (arb_occ, generalize_nth, f_occ)
           ∧
            ¬ are_same_number (recursion_on_nth, generalize_nth)
           ∧
            in_some_definition
              (f_term, generalize_nth_argument_of, [generalize_nth, f_term])
```

in_some_definition (
f_term, <- key to look up the defining clauses
generalized_nth_argument_of, <- name of inner_assertion
[genearlize_nth, f_term]) <- arguments from outer-to-inner

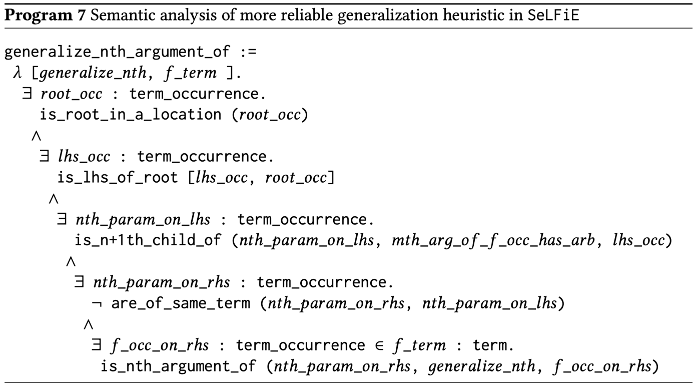


```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []        = []"
| "rev (x # xs) = rev xs @ [x]"

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

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```



inner assertion

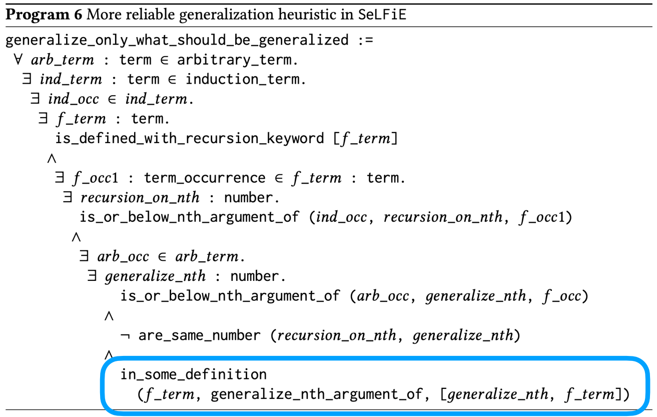
(= generalized_nth_argument_of)



Program 7 Semantic analysis of more reliable generalization heuristic in SeLFiE

```
generalize_nth_argument_of :=
 λ [generalize_nth, f_term ].
  ∃ root_occ : term_occurrence.
    is_root_in_a_location (root_occ)
   ∧
    ∃ lhs_occ : term_occurrence.
      is_lhs_of_root [lhs_occ, root_occ]
     ∧
      ∃ nth_param_on_lhs : term_occurrence.
        is_n+1th_child_of (nth_param_on_lhs, mth_arg_of_f_occ_has_arb, lhs_occ)
       ∧
        ∃ nth_param_on_rhs : term_occurrence.
          ¬ are_of_same_term (nth_param_on_rhs, nth_param_on_lhs)
         ∧
          ∃ f_occ_on_rhs : term_occurrence ∈ f_term : term.
           is_nth_argument_of (nth_param_on_rhs, generalize_nth, f_occ_on_rhs)
```

outer assertion



Program 6 More reliable generalization heuristic in SeLFiE

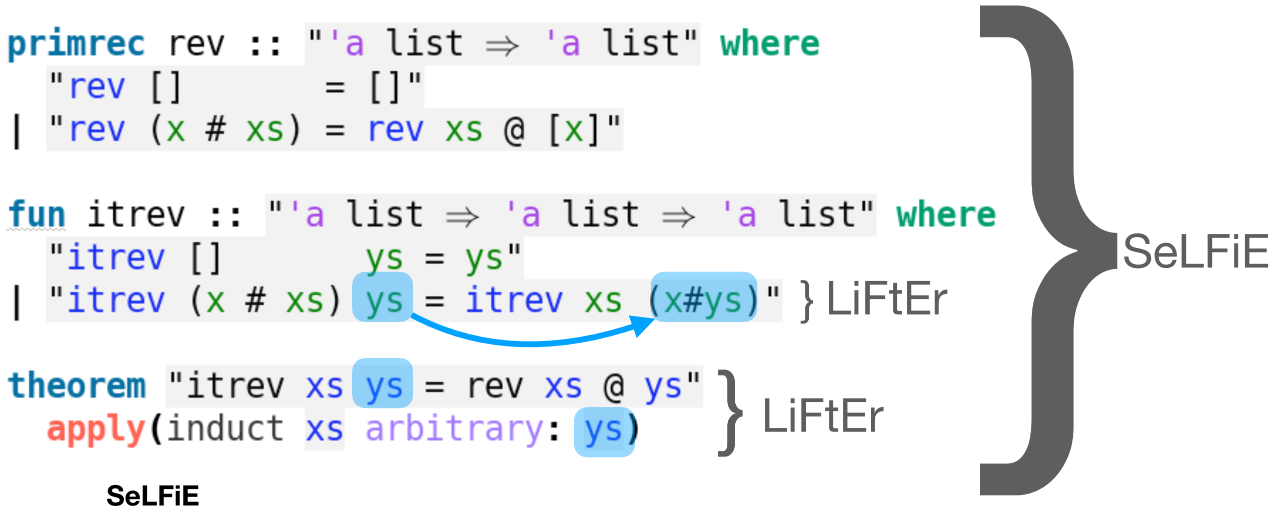
```
generalize_only_what_should_be_generalized :=
 ∀ arb_term : term ∈ arbitrary_term.
  ∃ ind_term : term ∈ induction_term.
   ∃ ind_occ ∈ ind_term.
    ∃ f_term : term.
      is_defined_with_recursion_keyword [f_term]
     ∧
      ∃ f_occ1 : term_occurrence ∈ f_term : term.
       ∃ recursion_on_nth : number.
         is_or_below_nth_argument_of (ind_occ, recursion_on_nth, f_occ1)
        ∧
         ∃ arb_occ ∈ arb_term.
          ∃ generalize_nth : number.
            is_or_below_nth_argument_of (arb_occ, generalize_nth, f_occ)
           ∧
            ¬ are_same_number (recursion_on_nth, generalize_nth)
           ∧
            in_some_definition
              (f_term, generalize_nth_argument_of, [generalize_nth, f_term])
```

in_some_definition (

f_term, <- key to look up the defining clauses

generalized_nth_argument_of, <- name of inner_assertion

[genearlize_nth, f_term])<- arguments from outer-to-inner



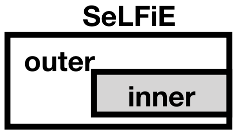
```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []        = []"
| "rev (x # xs) = rev xs @ [x]"

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

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

outer assertion

Program 6 More reliable generalization heuristic in SeLFiE

```
generalize_only_what_should_be_generalized :=
 ∀ arb_term : term ∈ arbitrary_term.
  ∃ ind_term : term ∈ induction_term.
   ∃ ind_occ ∈ ind_term.
    ∃ f_term : term.
      is_defined_with_recursion_keyword [f_term]
     ∧
      ∃ f_occ1 : term_occurrence ∈ f_term : term.
       ∃ recursion_on_nth : number.
         is_or_below_nth_argument_of (ind_occ, recursion_on_nth, f_occ1)
        ∧
         ∃ arb_occ ∈ arb_term.
          ∃ generalize_nth : number.
            is_or_below_nth_argument_of (arb_occ, generalize_nth, f_occ)
           ∧
            ¬ are_same_number (recursion_on_nth, generalize_nth)
           ∧
            in_some_definition
              (f_term, generalize_nth_argument_of, [generalize_nth, f_term])
```

in_some_definition (
f_term, <- key to look up the defining clauses
generalized_nth_argument_of, <- name of inner_assertion
[genearlize_nth, f_term])<- arguments from outer-to-inner

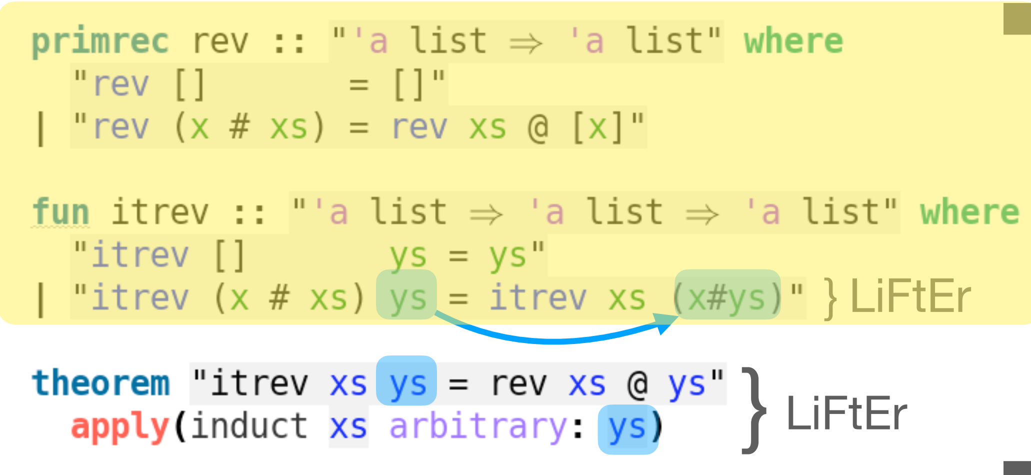


inner assertion

(= generalized_nth_argument_of)

Program 7 Semantic analysis of more reliable generalization heuristic in SeLFiE

```
generalize_nth_argument_of :=
 λ [generalize_nth, f_term ].
  ∃ root_occ : term_occurrence.
    is_root_in_a_location (root_occ)
   ∧
    ∃ lhs_occ : term_occurrence.
      is_lhs_of_root [lhs_occ, root_occ]
     ∧
      ∃ nth_param_on_lhs : term_occurrence.
        is_n+1th_child_of (nth_param_on_lhs, mth_arg_of_f_occ_has_arb, lhs_occ)
       ∧
        ∃ nth_param_on_rhs : term_occurrence.
          ¬ are_of_same_term (nth_param_on_rhs, nth_param_on_lhs)
         ∧
          ∃ f_occ_on_rhs : term_occurrence ∈ f_term : term.
           is_nth_argument_of (nth_param_on_rhs, generalize_nth, f_occ_on_rhs)
```



```
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)"
```

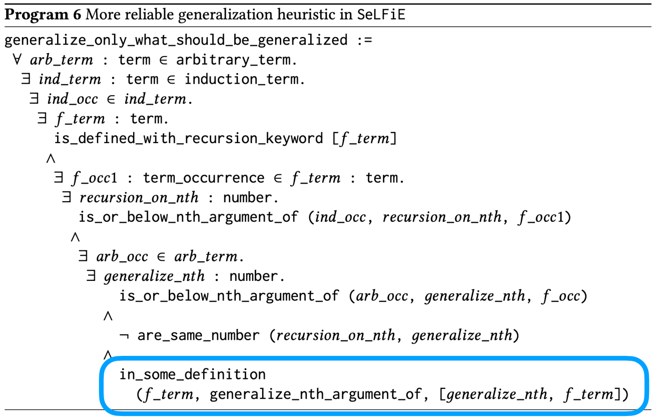
} LiFtEr

```
theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

} LiFtEr

SeLFiE

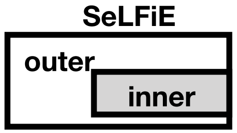
outer assertion



Program 6 More reliable generalization heuristic in SeLFiE

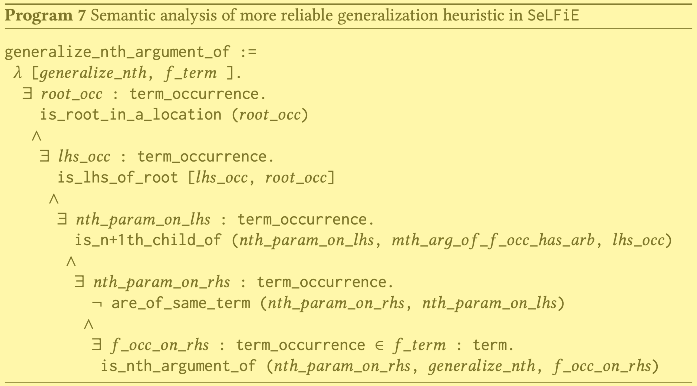
```
generalize_only_what_should_be_generalized :=
 ∀ arb_term : term ∈ arbitrary_term.
  ∃ ind_term : term ∈ induction_term.
   ∃ ind_occ ∈ ind_term.
    ∃ f_term : term.
      is_defined_with_recursion_keyword [f_term]
     ∧
      ∃ f_occ1 : term_occurrence ∈ f_term : term.
       ∃ recursion_on_nth : number.
         is_or_below_nth_argument_of (ind_occ, recursion_on_nth, f_occ1)
        ∧
         ∃ arb_occ ∈ arb_term.
          ∃ generalize_nth : number.
              is_or_below_nth_argument_of (arb_occ, generalize_nth, f_occ)
             ∧
              ¬ are_same_number (recursion_on_nth, generalize_nth)
             ∧
              in_some_definition
                (f_term, generalize_nth_argument_of, [generalize_nth, f_term])
```

in_some_definition (
f_term, <- key to look up the defining clauses
generalized_nth_argument_of, <- name of inner_assertion
[genearlize_nth, f_term]) <- arguments from outer-to-inner



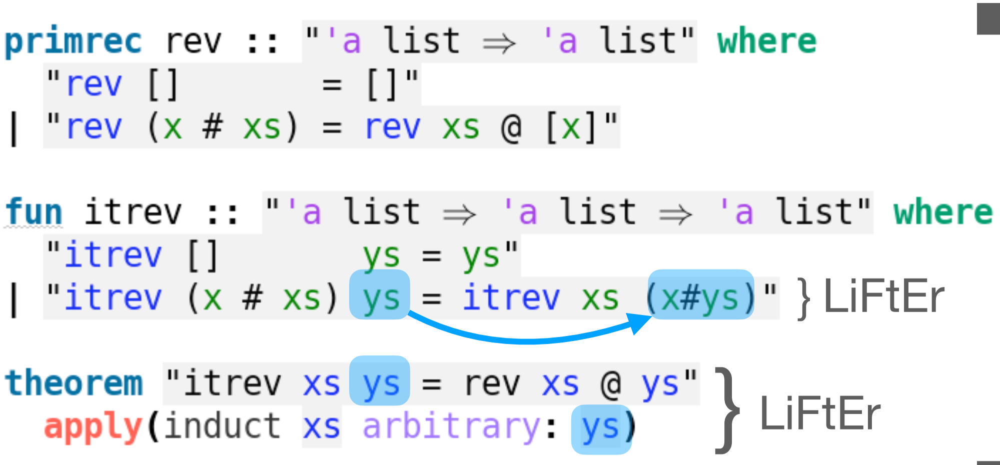
inner assertion

(= generalized_nth_argument_of)



Program 7 Semantic analysis of more reliable generalization heuristic in SeLFiE

```
generalize_nth_argument_of :=
 λ [generalize_nth, f_term ].
  ∃ root_occ : term_occurrence.
    is_root_in_a_location (root_occ)
   ∧
    ∃ lhs_occ : term_occurrence.
      is_lhs_of_root [lhs_occ, root_occ]
     ∧
      ∃ nth_param_on_lhs : term_occurrence.
        is_n+1th_child_of (nth_param_on_lhs, mth_arg_of_f_occ_has_arb, lhs_occ)
       ∧
        ∃ nth_param_on_rhs : term_occurrence.
          ¬ are_of_same_term (nth_param_on_rhs, nth_param_on_lhs)
         ∧
          ∃ f_occ_on_rhs : term_occurrence ∈ f_term : term.
           is_nth_argument_of (nth_param_on_rhs, generalize_nth, f_occ_on_rhs)
```



```
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)" } LiFtEr

theorem "itrev xs ys = rev xs @ ys" 
  apply(induct xs arbitrary: ys)      } LiFtEr
```

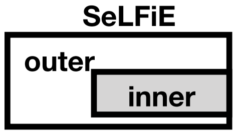
} SeLFiE

outer assertion

Program 6 More reliable generalization heuristic in SeLFiE

```
generalize_only_what_should_be_generalized :=
 ∀ arb_term : term ∈ arbitrary_term.
  ∃ ind_term : term ∈ induction_term.
   ∃ ind_occ ∈ ind_term.
    ∃ f_term : term.
      is_defined_with_recursion_keyword [f_term]
     ∧
      ∃ f_occ1 : term_occurrence ∈ f_term : term.
       ∃ recursion_on_nth : number.
         is_or_below_nth_argument_of (ind_occ, recursion_on_nth, f_occ1)
        ∧
         ∃ arb_occ ∈ arb_term.
          ∃ generalize_nth : number.
            is_or_below_nth_argument_of (arb_occ, generalize_nth, f_occ)
           ∧
            ¬ are_same_number (recursion_on_nth, generalize_nth)
           ∧
            in_some_definition
              (f_term, generalize_nth_argument_of, [generalize_nth, f_term])
```

in_some_definition (
f_term, <- key to look up the defining clauses
generalized_nth_argument_of, <- name of inner_assertion
[genearlize_nth, f_term]) <- arguments from outer-to-inner

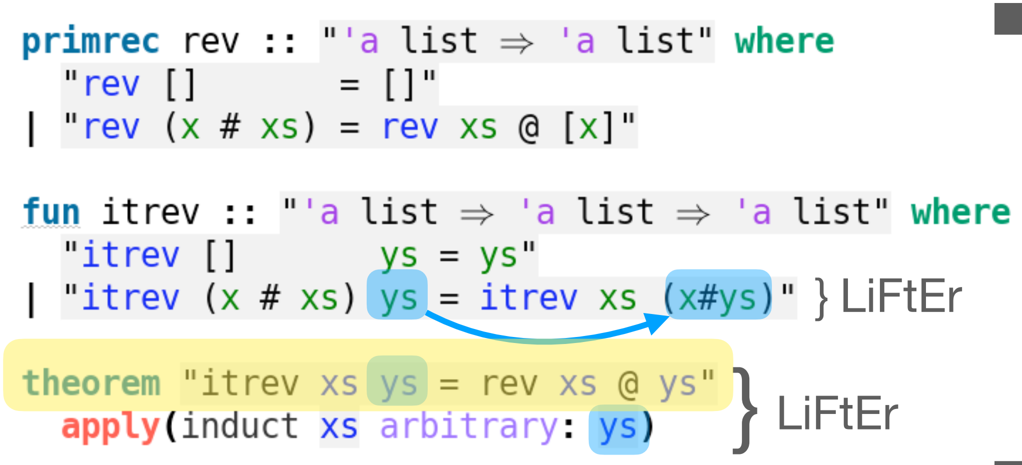


inner assertion

(= generalized_nth_argument_of)

Program 7 Semantic analysis of more reliable generalization heuristic in SeLFiE

```
generalize_nth_argument_of :=
 λ [generalize_nth, f_term ].
  ∃ root_occ : term_occurrence.
    is_root_in_a_location (root_occ)
   ∧
    ∃ lhs_occ : term_occurrence.
      is_lhs_of_root [lhs_occ, root_occ]
     ∧
      ∃ nth_param_on_lhs : term_occurrence.
        is_n+1th_child_of (nth_param_on_lhs, mth_arg_of_f_occ_has_arb, lhs_occ)
       ∧
        ∃ nth_param_on_rhs : term_occurrence.
          ¬ are_of_same_term (nth_param_on_rhs, nth_param_on_lhs)
         ∧
          ∃ f_occ_on_rhs : term_occurrence ∈ f_term : term.
           is_nth_argument_of (nth_param_on_rhs, generalize_nth, f_occ_on_rhs)
```



```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

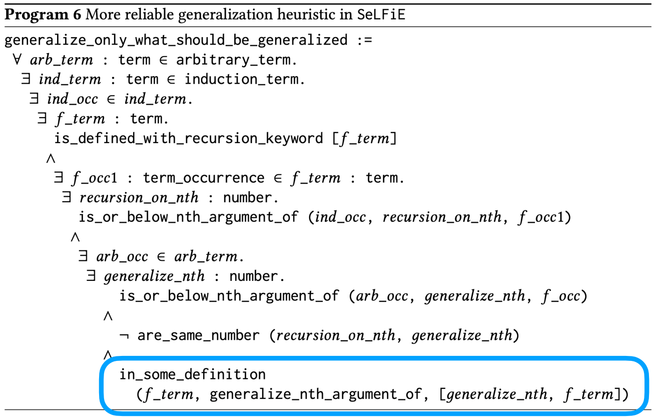
} LiFtEr

} LiFtEr

} SeLFiE

xs is the first argument of itrev.
If we apply induction on xs
should we generalise ys, which is the second argument of itrev?

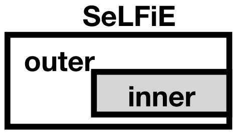
outer assertion



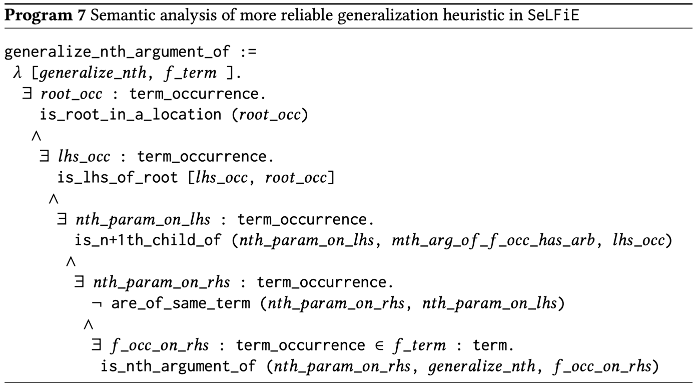
Program 6 More reliable generalization heuristic in SeLFiE

```
generalize_only_what_should_be_generalized :=
 ∀ arb_term : term ∈ arbitrary_term.
  ∃ ind_term : term ∈ induction_term.
   ∃ ind_occ ∈ ind_term.
    ∃ f_term : term.
      is_defined_with_recursion_keyword [f_term]
     ∧
      ∃ f_occ1 : term_occurrence ∈ f_term : term.
       ∃ recursion_on_nth : number.
         is_or_below_nth_argument_of (ind_occ, recursion_on_nth, f_occ1)
        ∧
         ∃ arb_occ ∈ arb_term.
          ∃ generalize_nth : number.
             is_or_below_nth_argument_of (arb_occ, generalize_nth, f_occ)
            ∧
             ¬ are_same_number (recursion_on_nth, generalize_nth)
            ∧
             in_some_definition
               (f_term, generalize_nth_argument_of, [generalize_nth, f_term])
```

in_some_definition (
f_term, <- key to look up the defining clauses
generalized_nth_argument_of, <- name of inner_assertion
[genearlize_nth, f_term]) <- arguments from outer-to-inner

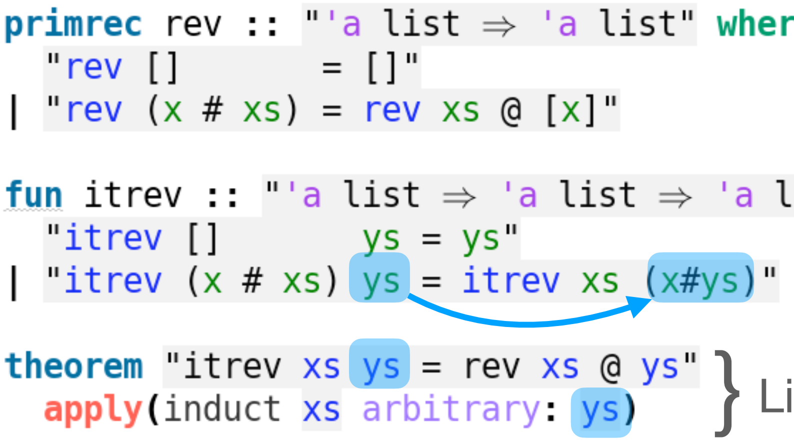


inner assertion
(= generalized_nth_argument_of)



Program 7 Semantic analysis of more reliable generalization heuristic in SeLFiE

```
generalize_nth_argument_of :=
 λ [generalize_nth, f_term ].
  ∃ root_occ : term_occurrence.
    is_root_in_a_location (root_occ)
   ∧
    ∃ lhs_occ : term_occurrence.
      is_lhs_of_root [lhs_occ, root_occ]
     ∧
      ∃ nth_param_on_lhs : term_occurrence.
        is_n+1th_child_of (nth_param_on_lhs, mth_arg_of_f_occ_has_arb, lhs_occ)
       ∧
        ∃ nth_param_on_rhs : term_occurrence.
          ¬ are_of_same_term (nth_param_on_rhs, nth_param_on_lhs)
         ∧
          ∃ f_occ_on_rhs : term_occurrence ∈ f_term : term.
           is_nth_argument_of (nth_param_on_rhs, generalize_nth, f_occ_on_rhs)
```



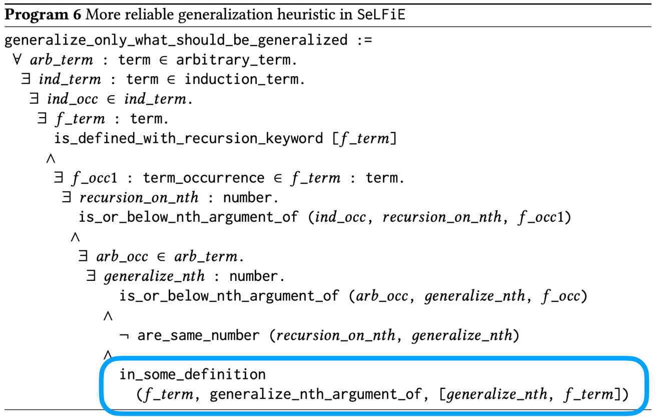
```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []        = []"
| "rev (x # xs) = rev xs @ [x]"

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

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

} LiFtEr

} LiFtEr

SeLFiE outer assertion

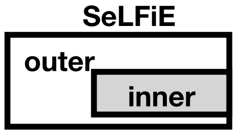
xs is the first argument of itrev.
If we apply induction on xs
should we generalise ys, which is the second argument of itrev?



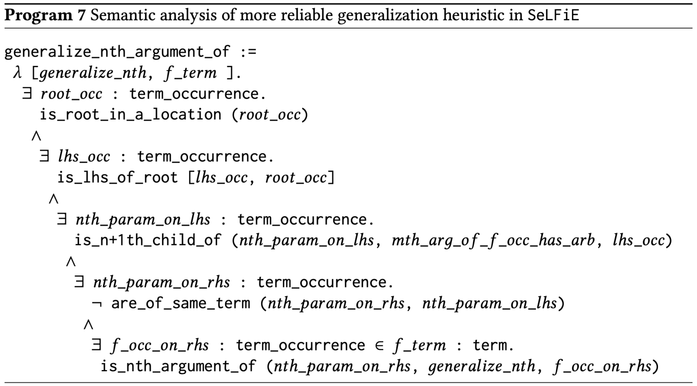
Program 6 More reliable generalization heuristic in SeLFiE

```
generalize_only_what_should_be_generalized :=
 ∀ arb_term : term ∈ arbitrary_term.
  ∃ ind_term : term ∈ induction_term.
   ∃ ind_occ ∈ ind_term.
    ∃ f_term : term.
      is_defined_with_recursion_keyword [f_term]
     ∧
      ∃ f_occ1 : term_occurrence ∈ f_term : term.
       ∃ recursion_on_nth : number.
         is_or_below_nth_argument_of (ind_occ, recursion_on_nth, f_occ1)
        ∧
         ∃ arb_occ ∈ arb_term.
          ∃ generalize_nth : number.
              is_or_below_nth_argument_of (arb_occ, generalize_nth, f_occ)
             ∧
              ¬ are_same_number (recursion_on_nth, generalize_nth)
             ∧
              in_some_definition
                (f_term, generalize_nth_argument_of, [generalize_nth, f_term])
```

in_some_definition (
f_term, <- key to look up the defining clauses
generalized_nth_argument_of, <- name of inner_assertion
[genearlize_nth, f_term]) <- arguments from outer-to-inner

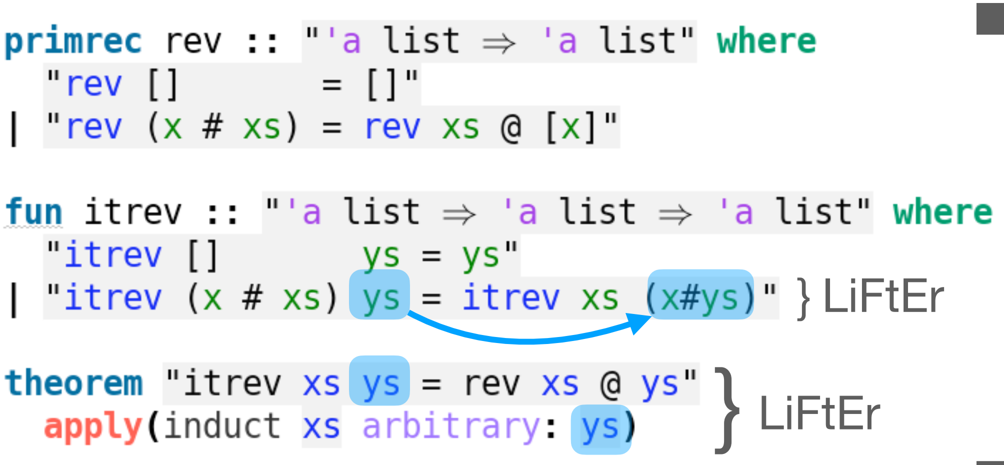


inner assertion
(= generalized_nth_argument_of)



Program 7 Semantic analysis of more reliable generalization heuristic in SeLFiE

```
generalize_nth_argument_of :=
 λ [generalize_nth, f_term ].
  ∃ root_occ : term_occurrence.
    is_root_in_a_location (root_occ)
   ∧
    ∃ lhs_occ : term_occurrence.
      is_lhs_of_root [lhs_occ, root_occ]
     ∧
      ∃ nth_param_on_lhs : term_occurrence.
        is_n+1th_child_of (nth_param_on_lhs, mth_arg_of_f_occ_has_arb, lhs_occ)
       ∧
        ∃ nth_param_on_rhs : term_occurrence.
          ¬ are_of_same_term (nth_param_on_rhs, nth_param_on_lhs)
         ∧
          ∃ f_occ_on_rhs : term_occurrence ∈ f_term : term.
           is_nth_argument_of (nth_param_on_rhs, generalize_nth, f_occ_on_rhs)
```



```
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)"

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

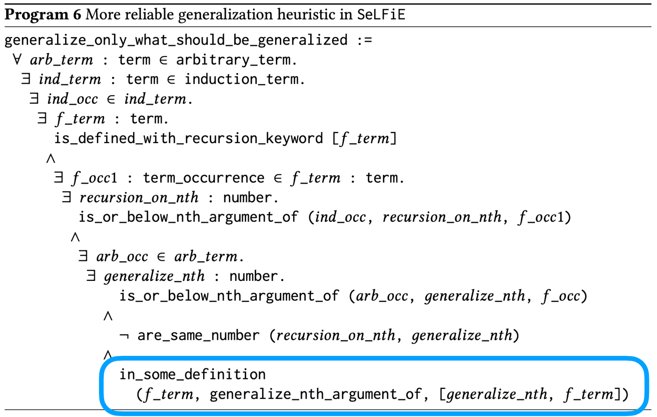
} LiFtEr

} LiFtEr

SeLFiE

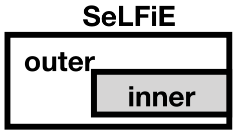
xs is the first argument of itrev.
If we apply induction on xs
should we generalise ys, which is the second argument of itrev?

outer assertion



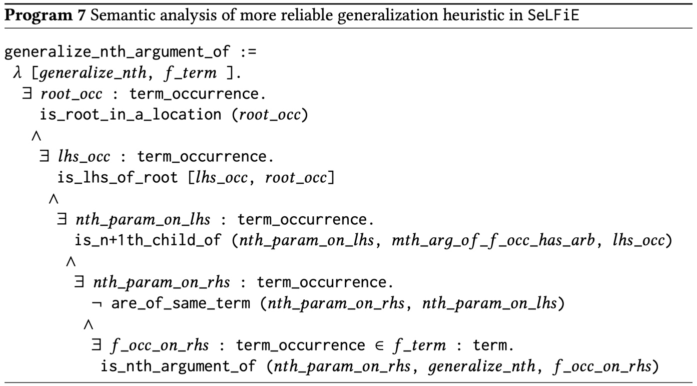
Program 6 More reliable generalization heuristic in SeLFiE

```
generalize_only_what_should_be_generalized :=
 ∀ arb_term : term ∈ arbitrary_term.
  ∃ ind_term : term ∈ induction_term.
   ∃ ind_occ ∈ ind_term.
    ∃ f_term : term.
      is_defined_with_recursion_keyword [f_term]
     ∧
      ∃ f_occ1 : term_occurrence ∈ f_term : term.
       ∃ recursion_on_nth : number.
         is_or_below_nth_argument_of (ind_occ, recursion_on_nth, f_occ1)
        ∧
         ∃ arb_occ ∈ arb_term.
          ∃ generalize_nth : number.
              is_or_below_nth_argument_of (arb_occ, generalize_nth, f_occ)
             ∧
              ¬ are_same_number (recursion_on_nth, generalize_nth)
             ∧
              in_some_definition
                (f_term, generalize_nth_argument_of, [generalize_nth, f_term])
```



[2, itrev]

inner assertion
(= generalized_nth_argument_of)



Program 7 Semantic analysis of more reliable generalization heuristic in SeLFiE

```
generalize_nth_argument_of :=
 λ [generalize_nth, f_term ].
  ∃ root_occ : term_occurrence.
    is_root_in_a_location (root_occ)
   ∧
    ∃ lhs_occ : term_occurrence.
      is_lhs_of_root [lhs_occ, root_occ]
     ∧
      ∃ nth_param_on_lhs : term_occurrence.
        is_n+1th_child_of (nth_param_on_lhs, mth_arg_of_f_occ_has_arb, lhs_occ)
       ∧
        ∃ nth_param_on_rhs : term_occurrence.
          ¬ are_of_same_term (nth_param_on_rhs, nth_param_on_lhs)
         ∧
          ∃ f_occ_on_rhs : term_occurrence ∈ f_term : term.
           is_nth_argument_of (nth_param_on_rhs, generalize_nth, f_occ_on_rhs)
```

in_some_definition (
f_term, <- key to look up the defining clauses
generalized_nth_argument_of, <- name of inner_assertion
[genearlize_nth, f_term]) <- arguments from outer-to-inner

```
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)"
```

} LiFtEr

```
theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```

} LiFtEr

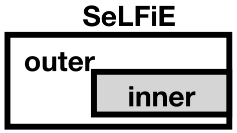
SeLFiE

xs is the first argument of itrev.
If we apply induction on xs
should we generalise ys, which is the second argument of itrev?

outer assertion

Program 6 More reliable generalization heuristic in SeLFiE

```
generalize_only_what_should_be_generalized :=
 ∀ arb_term : term ∈ arbitrary_term.
  ∃ ind_term : term ∈ induction_term.
   ∃ ind_occ ∈ ind_term.
    ∃ f_term : term.
      is_defined_with_recursion_keyword [f_term]
     ∧
      ∃ f_occ1 : term_occurrence ∈ f_term : term.
       ∃ recursion_on_nth : number.
         is_or_below_nth_argument_of (ind_occ, recursion_on_nth, f_occ1)
        ∧
         ∃ arb_occ ∈ arb_term.
          ∃ generalize_nth : number.
              is_or_below_nth_argument_of (arb_occ, generalize_nth, f_occ)
            ∧
              ¬ are_same_number (recursion_on_nth, generalize_nth)
            ∧
              in_some_definition
                (f_term, generalize_nth_argument_of, [generalize_nth, f_term])
```



[2, itrev]

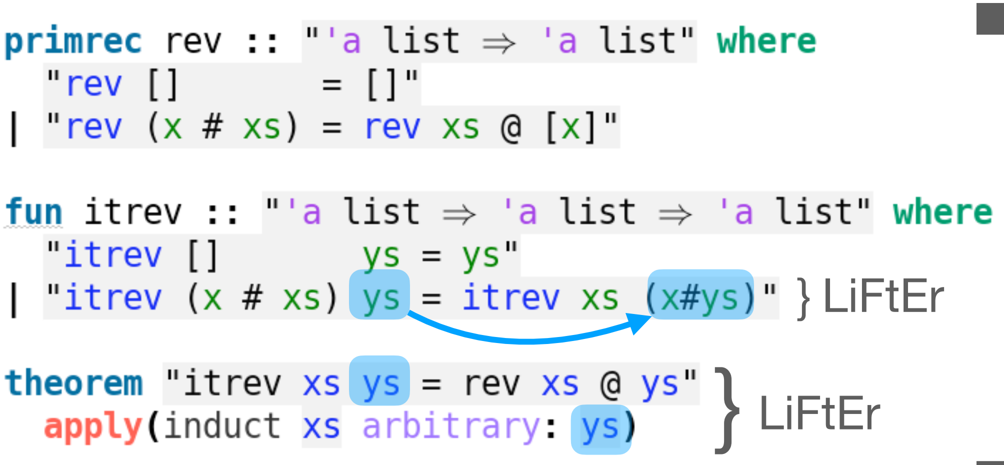
inner assertion
(= generalized_nth_argument_of)

Program 7 Semantic analysis of more reliable generalization heuristic in SeLFiE

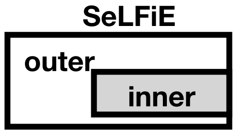
```
generalize_nth_argument_of :=
 λ [generalize_nth, f_term ].
  ∃ root_occ : term_occurrence.
    is_root_in_a_location (root_occ)
   ∧
    ∃ lhs_occ : term_occurrence.
      is_lhs_of_root [lhs_occ, root_occ]
     ∧
      ∃ nth_param_on_lhs : term_occurrence.
        is_n+1th_child_of (nth_param_on_lhs, mth_arg_of_f_occ_has_arb, lhs_occ)
       ∧
        ∃ nth_param_on_rhs : term_occurrence.
          ¬ are_of_same_term (nth_param_on_rhs, nth_param_on_lhs)
         ∧
          ∃ f_occ_on_rhs : term_occurrence ∈ f_term : term.
           is_nth_argument_of (nth_param_on_rhs, generalize_nth, f_occ_on_rhs)
```

in_some_definition (
f_term, <- key to look up the defining clauses
generalized_nth_argument_of, <- name of inner_assertion
[genearlize_nth, f_term]) <- arguments from outer-to-inner

Yes. In the second clause defining itrev, the second argument changes from the LHS to RHS.

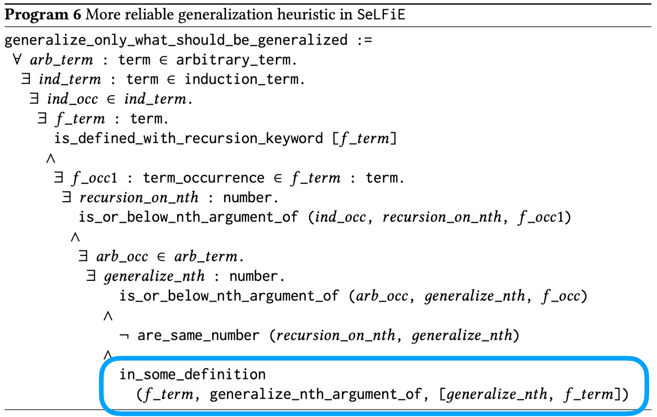


```
primrec rev :: "'a list ⇒ 'a list" where
  "rev []        = []"
| "rev (x # xs) = rev xs @ [x]"

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

theorem "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys)
```



xs is the first argument of itrev.
If we apply induction on xs
should we generalise ys, which is the second argument of itrev?

outer assertion

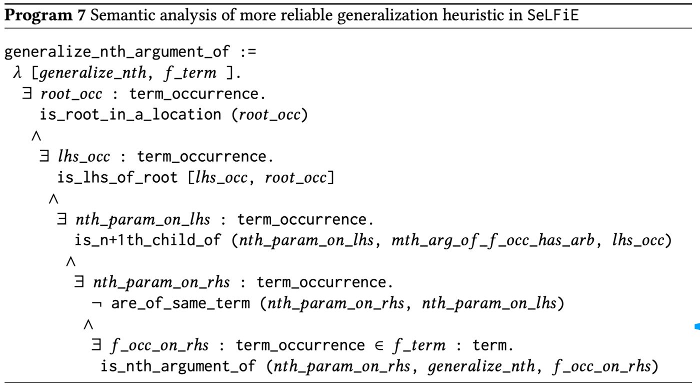


Program 6 More reliable generalization heuristic in SeLFiE

```
generalize_only_what_should_be_generalized :=
 ∀ arb_term : term ∈ arbitrary_term.
  ∃ ind_term : term ∈ induction_term.
   ∃ ind_occ ∈ ind_term.
    ∃ f_term : term.
      is_defined_with_recursion_keyword [f_term]
     ∧
      ∃ f_occ1 : term_occurrence ∈ f_term : term.
       ∃ recursion_on_nth : number.
         is_or_below_nth_argument_of (ind_occ, recursion_on_nth, f_occ1)
        ∧
         ∃ arb_occ ∈ arb_term.
          ∃ generalize_nth : number.
              is_or_below_nth_argument_of (arb_occ, generalize_nth, f_occ)
             ∧
              ¬ are_same_number (recursion_on_nth, generalize_nth)
             ∧
              in_some_definition
                (f_term, generalize_nth_argument_of, [generalize_nth, f_term])
```

[2, itrev]

inner assertion
(= generalized_nth_argument_of)

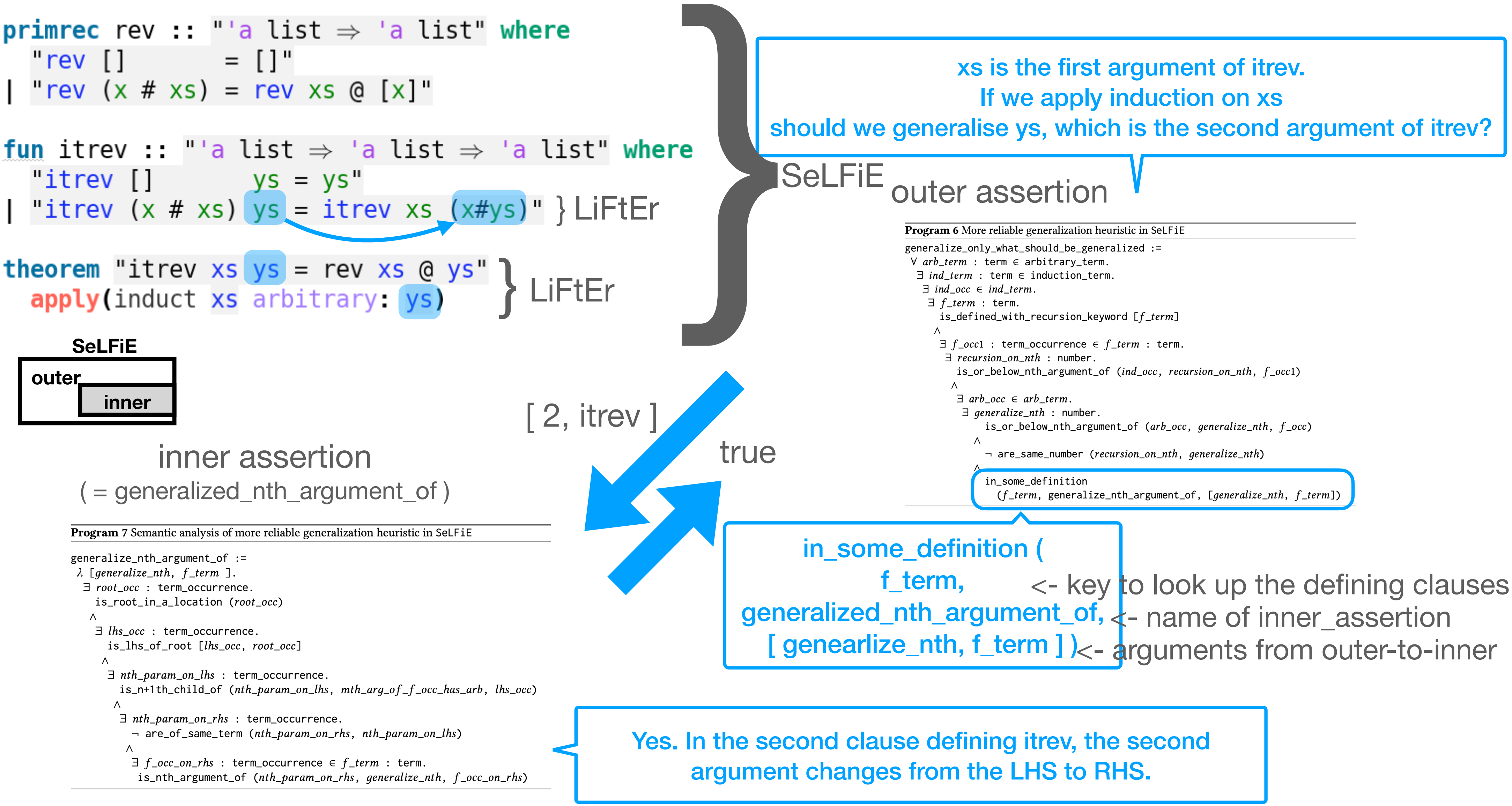


Program 7 Semantic analysis of more reliable generalization heuristic in SeLFiE

```
generalize_nth_argument_of :=
 λ [generalize_nth, f_term ].
  ∃ root_occ : term_occurrence.
    is_root_in_a_location (root_occ)
   ∧
    ∃ lhs_occ : term_occurrence.
      is_lhs_of_root [lhs_occ, root_occ]
     ∧
      ∃ nth_param_on_lhs : term_occurrence.
        is_n+1th_child_of (nth_param_on_lhs, mth_arg_of_f_occ_has_arb, lhs_occ)
       ∧
        ∃ nth_param_on_rhs : term_occurrence.
          ¬ are_of_same_term (nth_param_on_rhs, nth_param_on_lhs)
         ∧
          ∃ f_occ_on_rhs : term_occurrence ∈ f_term : term.
           is_nth_argument_of (nth_param_on_rhs, generalize_nth, f_occ_on_rhs)
```

in_some_definition (
f_term, <- key to look up the defining clauses
generalized_nth_argument_of, <- name of inner_assertion
[genearlize_nth, f_term]) <- arguments from outer-to-inner

Yes. In the second clause defining itrev, the second argument changes from the LHS to RHS.



```
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)" } LiFtEr

theorem "itrev xs ys = rev xs @ ys" } LiFtEr
  apply(induct xs arbitrary: ys)
```

SeLFiE

outer assertion

xs is the first argument of itrev.
If we apply induction on xs
should we generalise ys, which is the second argument of itrev?

SeLFiE

outer

inner

Program 6 More reliable generalization heuristic in SeLFiE

```
generalize_only_what_should_be_generalized :=
 ∀ arb_term : term ∈ arbitrary_term.
  ∃ ind_term : term ∈ induction_term.
   ∃ ind_occ ∈ ind_term.
    ∃ f_term : term.
      is_defined_with_recursion_keyword [f_term]
     ∧
      ∃ f_occ1 : term_occurrence ∈ f_term : term.
       ∃ recursion_on_nth : number.
         is_or_below_nth_argument_of (ind_occ, recursion_on_nth, f_occ1)
        ∧
         ∃ arb_occ ∈ arb_term.
          ∃ generalize_nth : number.
            is_or_below_nth_argument_of (arb_occ, generalize_nth, f_occ)
           ∧
            ¬ are_same_number (recursion_on_nth, generalize_nth)
           ∧
            in_some_definition
              (f_term, generalize_nth_argument_of, [generalize_nth, f_term])
```

[2, itrev]

true

inner assertion

(= generalized_nth_argument_of)

Program 7 Semantic analysis of more reliable generalization heuristic in SeLFiE

```
generalize_nth_argument_of :=
 λ [generalize_nth, f_term ].
  ∃ root_occ : term_occurrence.
    is_root_in_a_location (root_occ)
   ∧
    ∃ lhs_occ : term_occurrence.
      is_lhs_of_root [lhs_occ, root_occ]
     ∧
      ∃ nth_param_on_lhs : term_occurrence.
        is_n+1th_child_of (nth_param_on_lhs, mth_arg_of_f_occ_has_arb, lhs_occ)
       ∧
        ∃ nth_param_on_rhs : term_occurrence.
          ¬ are_of_same_term (nth_param_on_rhs, nth_param_on_lhs)
         ∧
          ∃ f_occ_on_rhs : term_occurrence ∈ f_term : term.
           is_nth_argument_of (nth_param_on_rhs, generalize_nth, f_occ_on_rhs)
```

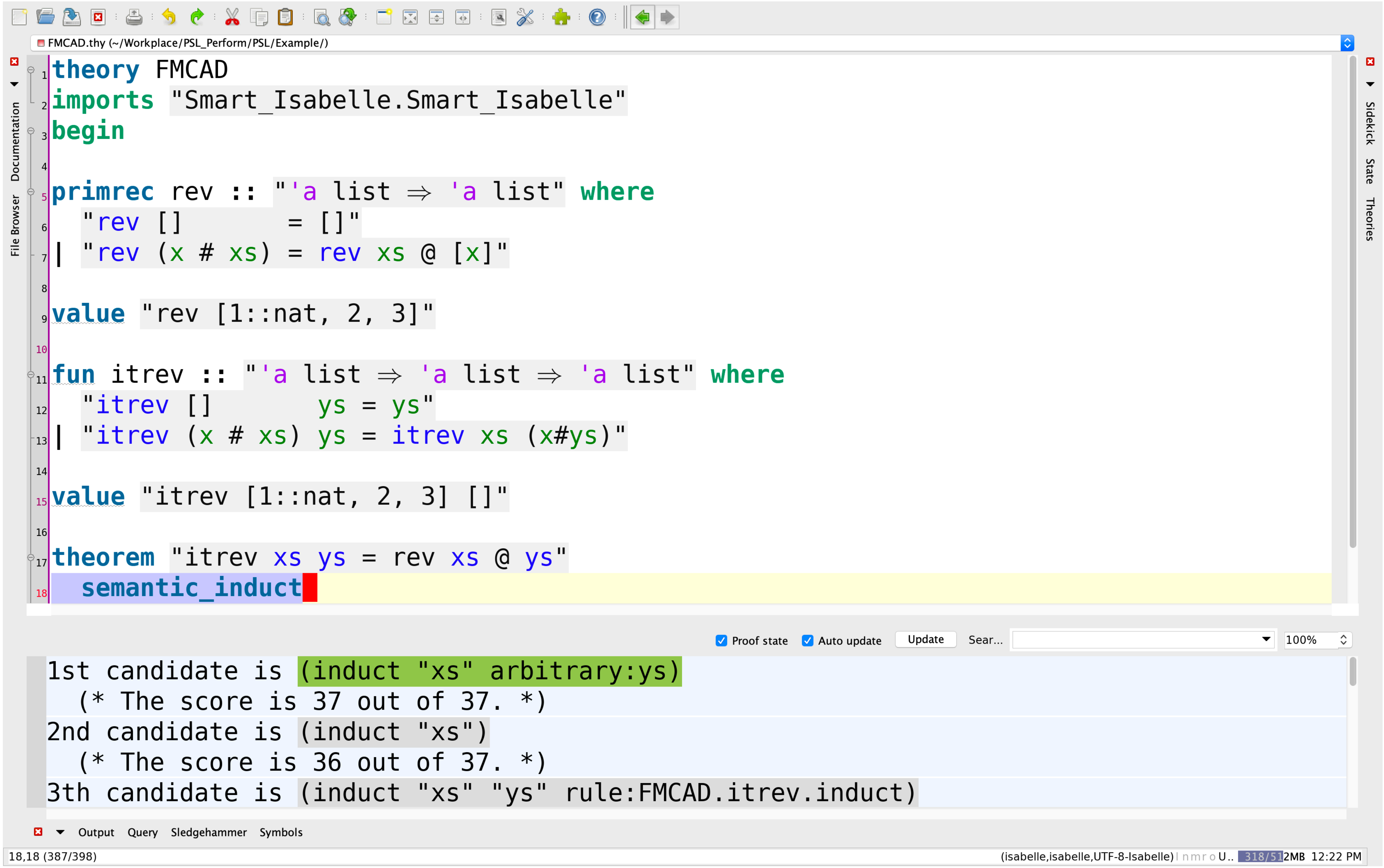
in_some_definition (
f_term, <- key to look up the defining clauses
generalized_nth_argument_of, <- name of inner_assertion
[genearlize_nth, f_term]) <- arguments from outer-to-inner

Yes. In the second clause defining itrev, the second argument changes from the LHS to RHS.

DEMO

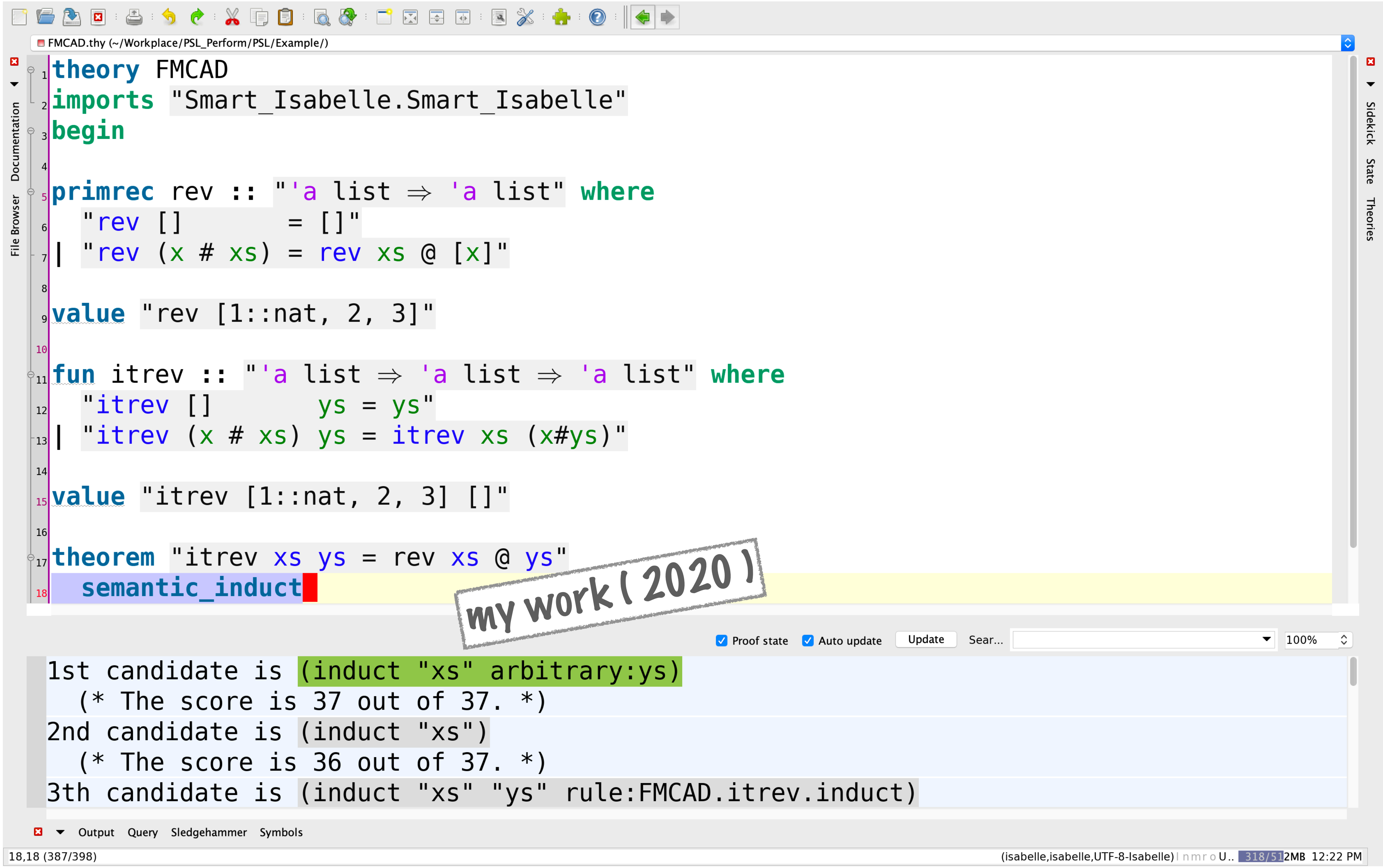
semantic_induct

The example theorem is taken from “Isabelle/HOL A Proof Assistant for Higher-Order Logic” Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel page 36



```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"
  semantic_induct
```

```
1st candidate is (induct "xs" arbitrary:ys)
  (* The score is 37 out of 37. *)
2nd candidate is (induct "xs")
  (* The score is 36 out of 37. *)
3th candidate is (induct "xs" "ys" rule:FMCAD.itrev.induct)
```

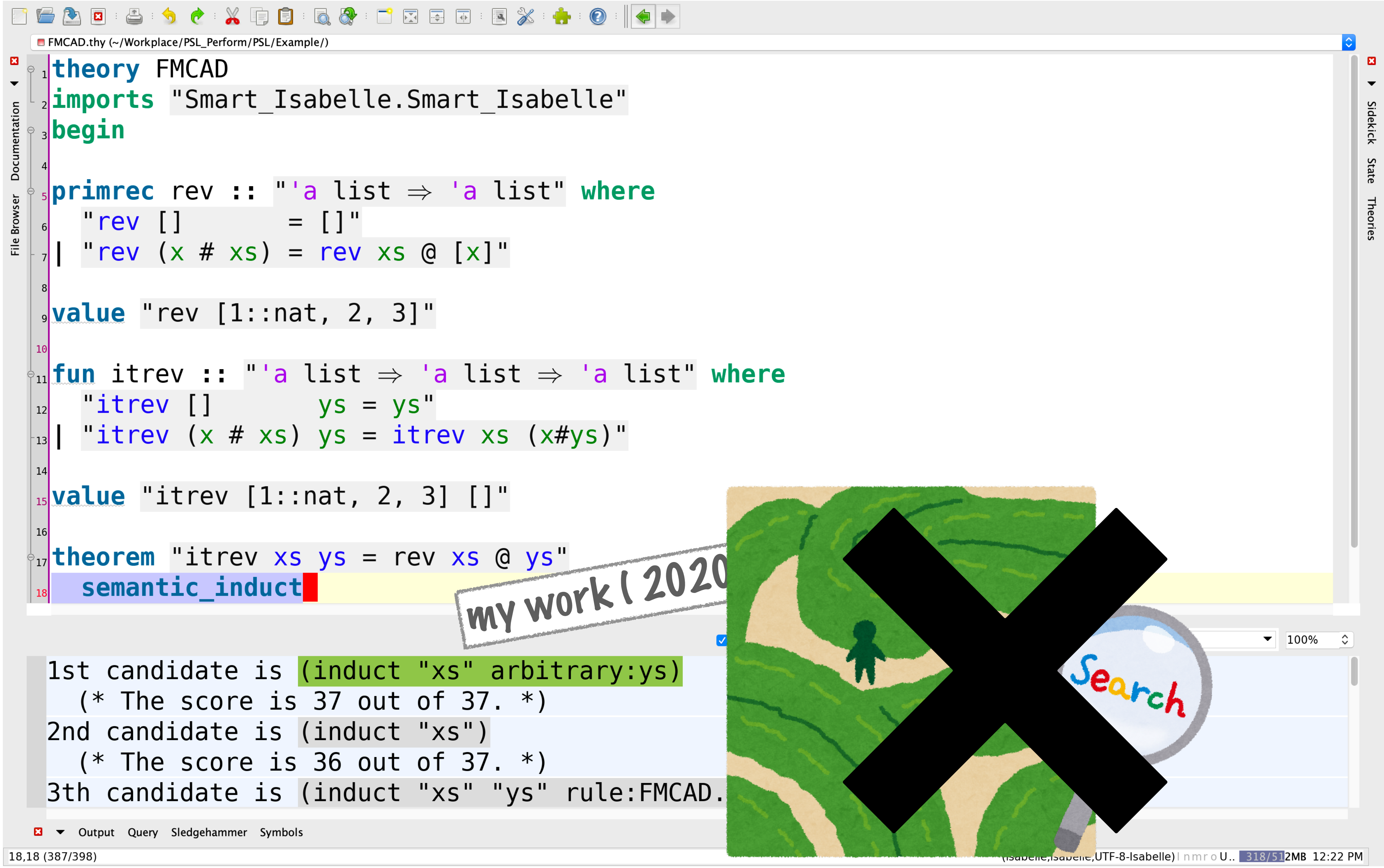


```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"
  semantic_induct
```

my work (2020)

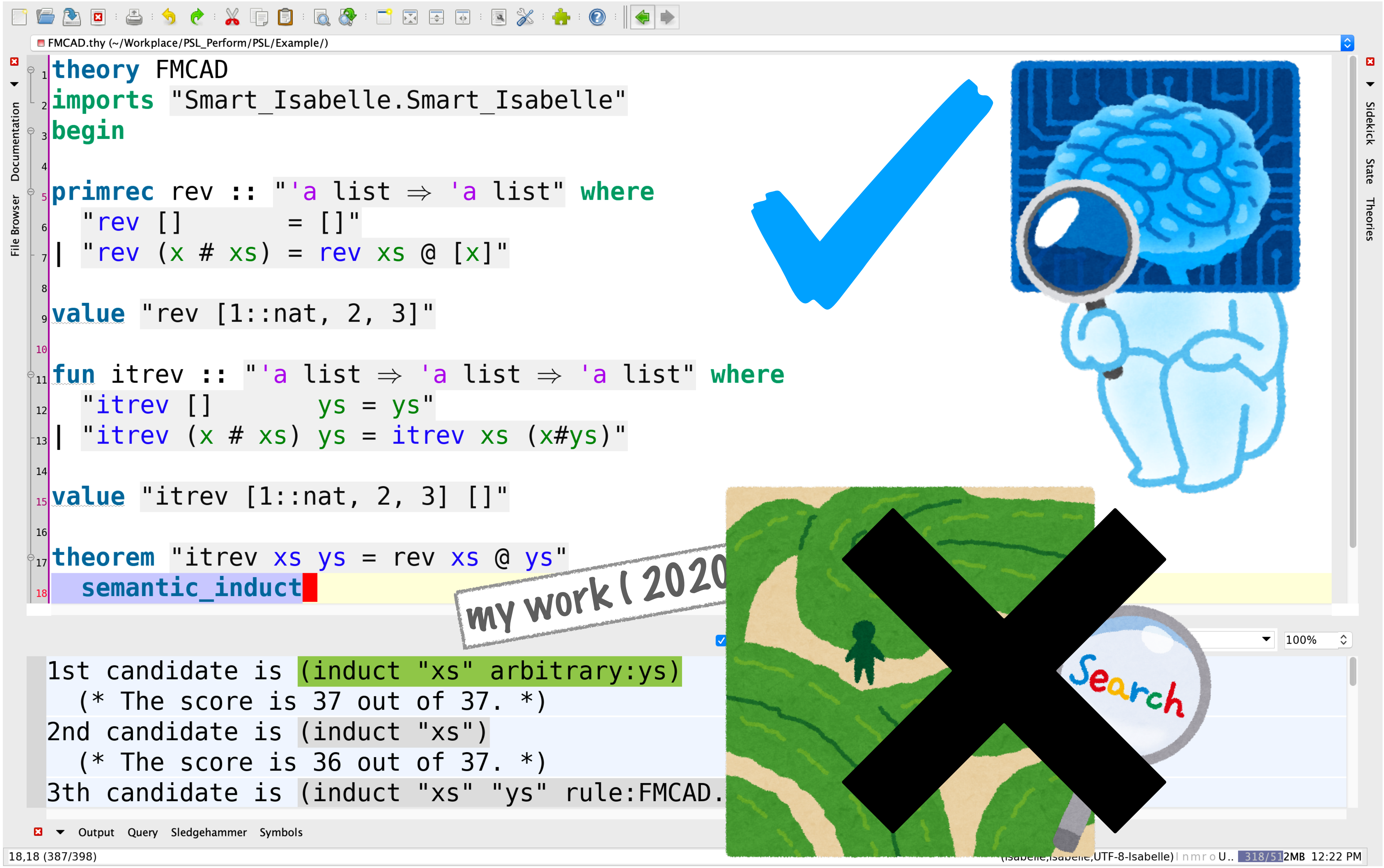
```
1st candidate is (induct "xs" arbitrary:ys)
  (* The score is 37 out of 37. *)
2nd candidate is (induct "xs")
  (* The score is 36 out of 37. *)
3th candidate is (induct "xs" "ys" rule:FMCAD.itrev.induct)
```

Output
Query
Sledgehammer
Symbols
18,18 (387/398)
(isabelle,isabelle,UTF-8-Isabelle)| n m r o U.. 318/512MB 12:22 PM



```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"
  semantic_induct
```

```
1st candidate is (induct "xs" arbitrary:ys)
  (* The score is 37 out of 37. *)
2nd candidate is (induct "xs")
  (* The score is 36 out of 37. *)
3th candidate is (induct "xs" "ys" rule:FMCAD.
```



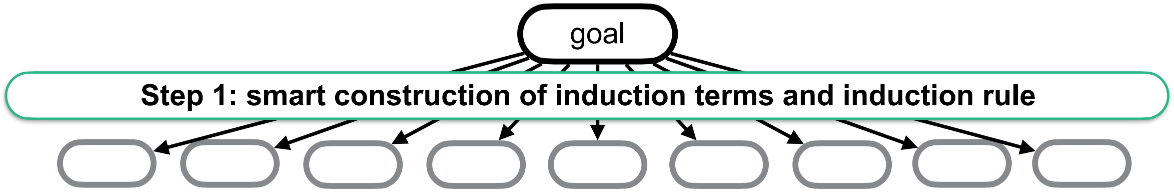
```
theory FMCAD
imports "Smart_Isabelle.Smart_Isabelle"
begin

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

value "rev [1::nat, 2, 3]"

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

value "itrev [1::nat, 2, 3] []"

theorem "itrev xs ys = rev xs @ ys"
  semantic_induct
```

```
1st candidate is (induct "xs" arbitrary:ys)
  (* The score is 37 out of 37. *)
2nd candidate is (induct "xs")
  (* The score is 36 out of 37. *)
3th candidate is (induct "xs" "ys" rule:FMCAD.
```

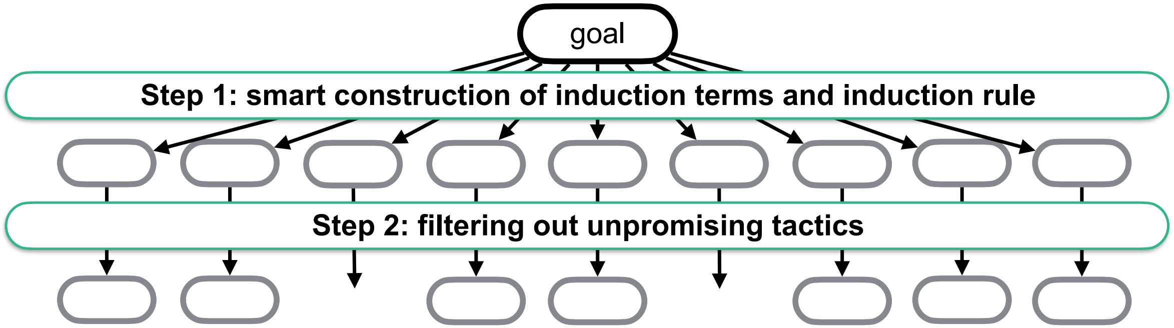
Build semantic_induct using SeLFiE

goal

Build semantic_induct using SeLFiE


goal
Step 1: smart construction of induction terms and induction rule

Build semantic_induct using SeLFiE

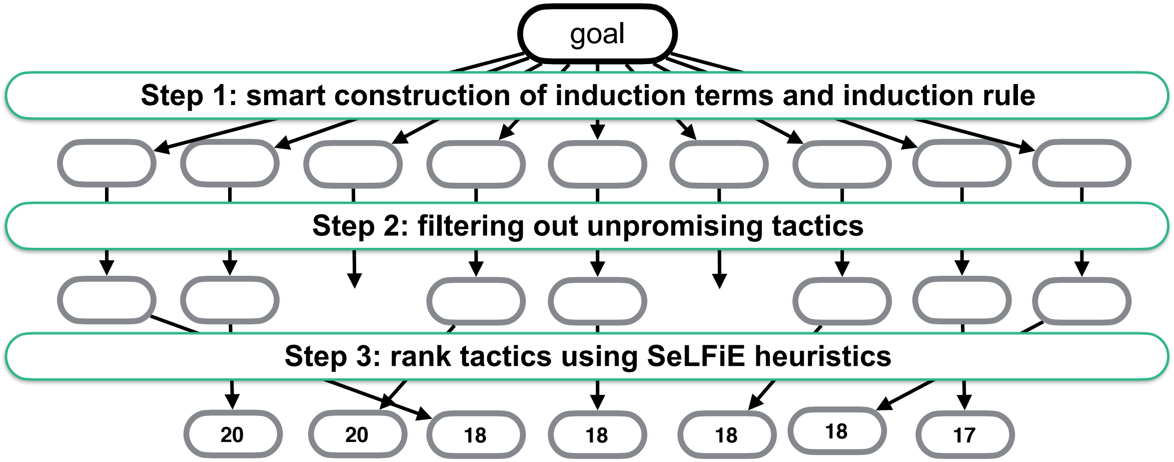


goal

Step 1: smart construction of induction terms and induction rule

Step 2: filtering out unpromising tactics

Build semantic_induct using SeLFiE

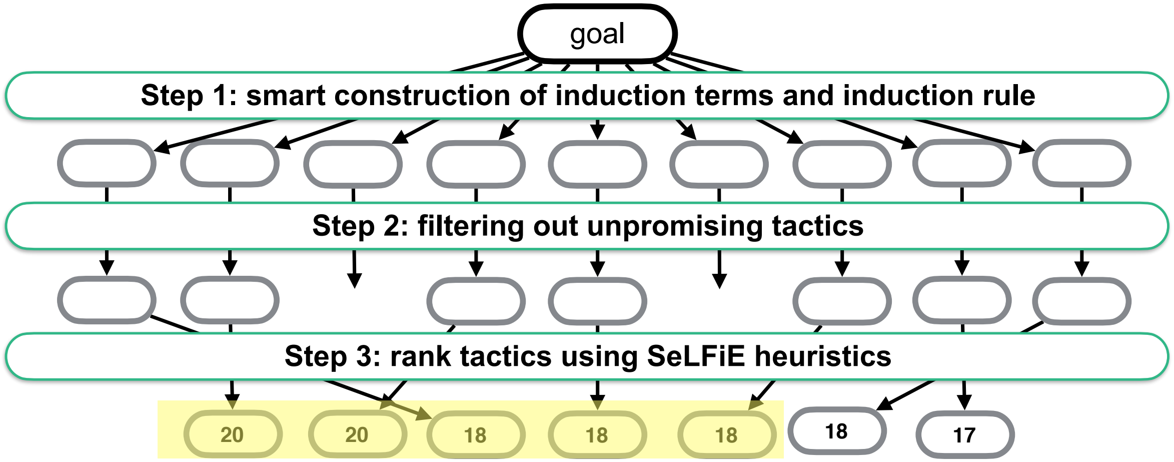


Step 1: smart construction of induction terms and induction rule

Step 2: filtering out unpromising tactics

Step 3: rank tactics using SeLFiE heuristics

Build semantic_induct using SeLFiE



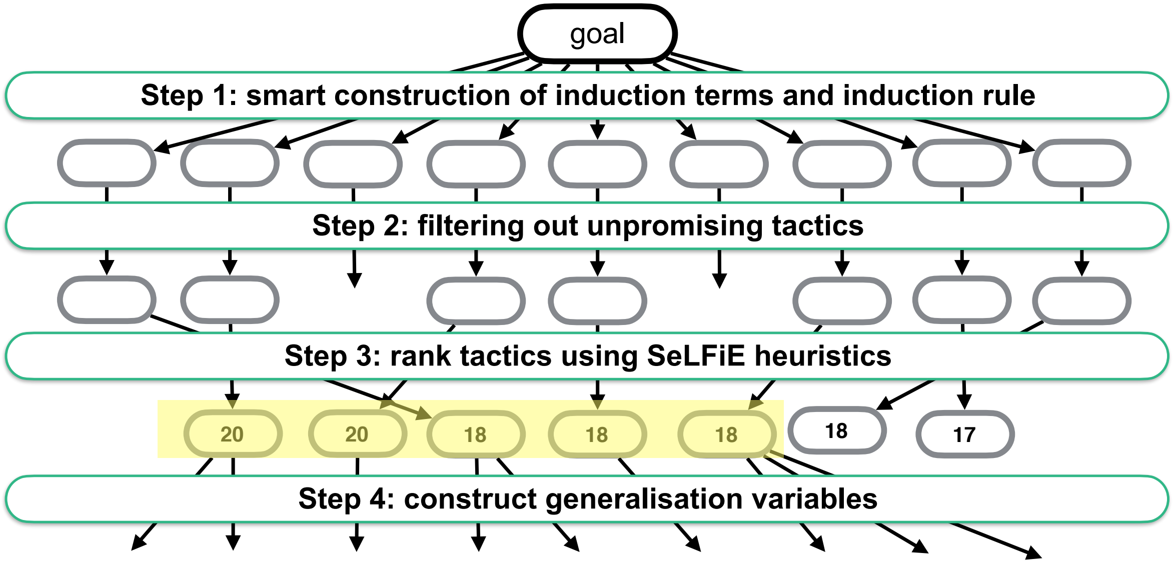
goal

Step 1: smart construction of induction terms and induction rule

Step 2: filtering out unpromising tactics

Step 3: rank tactics using SeLFiE heuristics

Build semantic_induct using SeLFiE



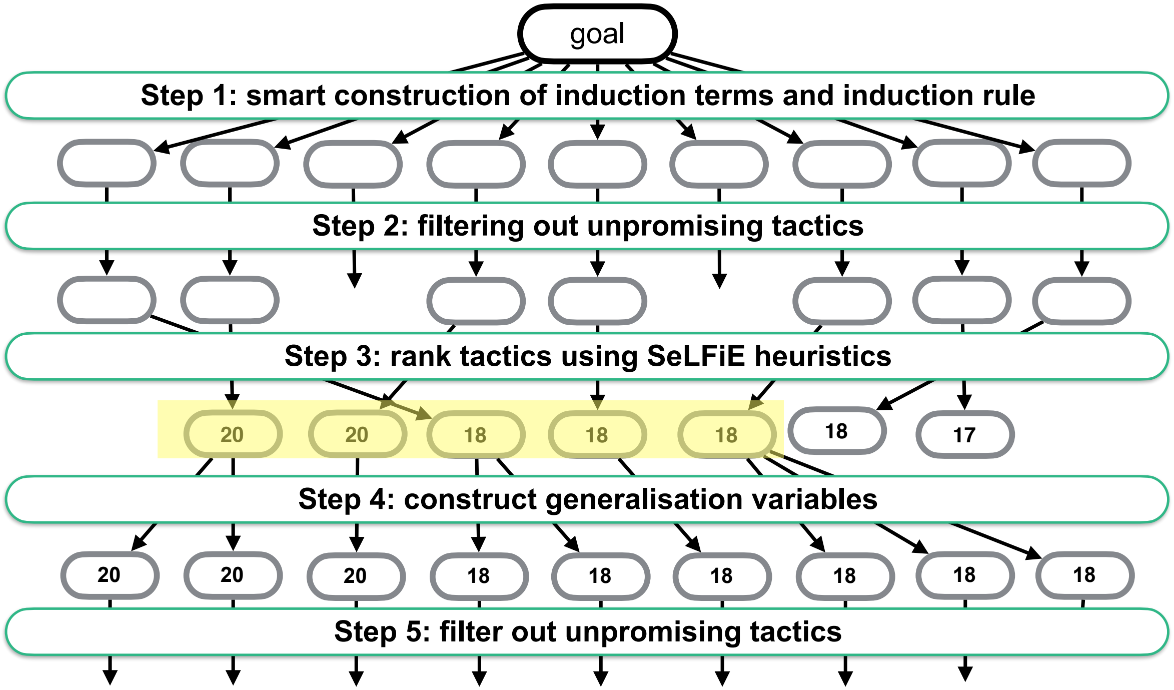
Step 1: smart construction of induction terms and induction rule

Step 2: filtering out unpromising tactics

Step 3: rank tactics using SeLFiE heuristics

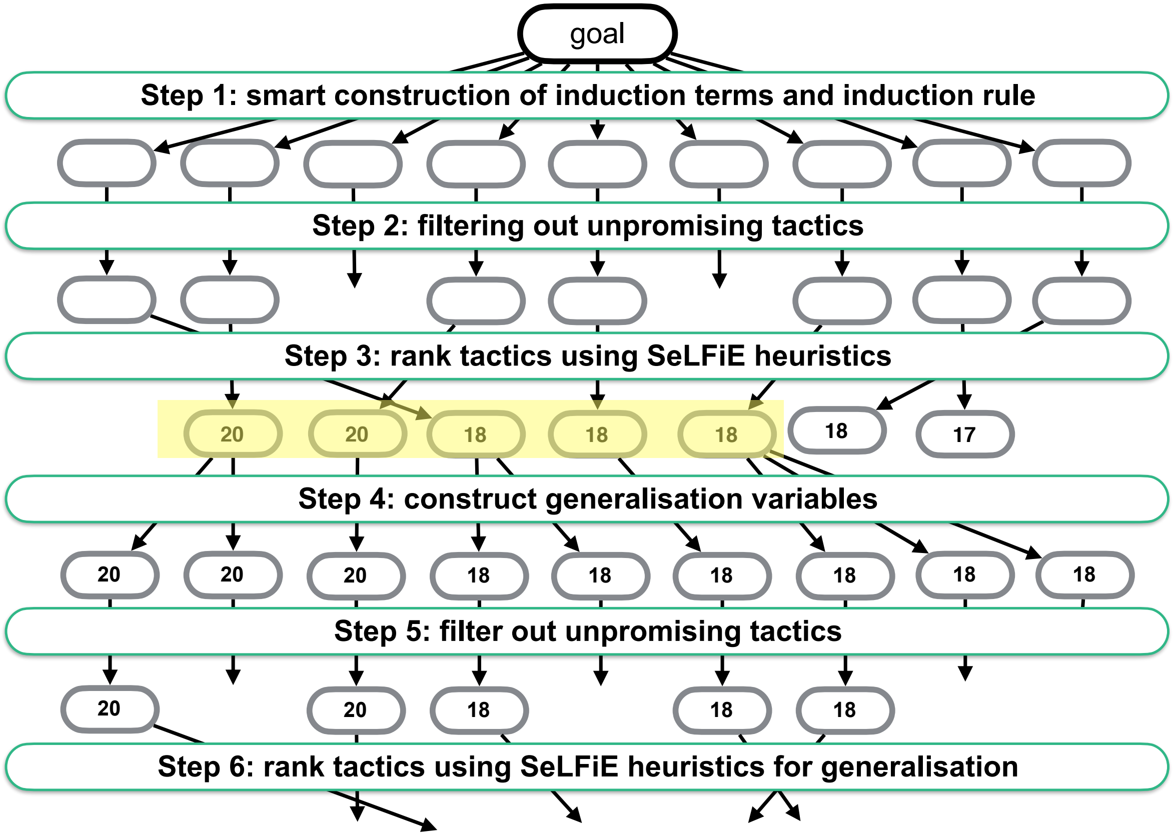
Step 4: construct generalisation variables

Build semantic_induct using SeLFiE

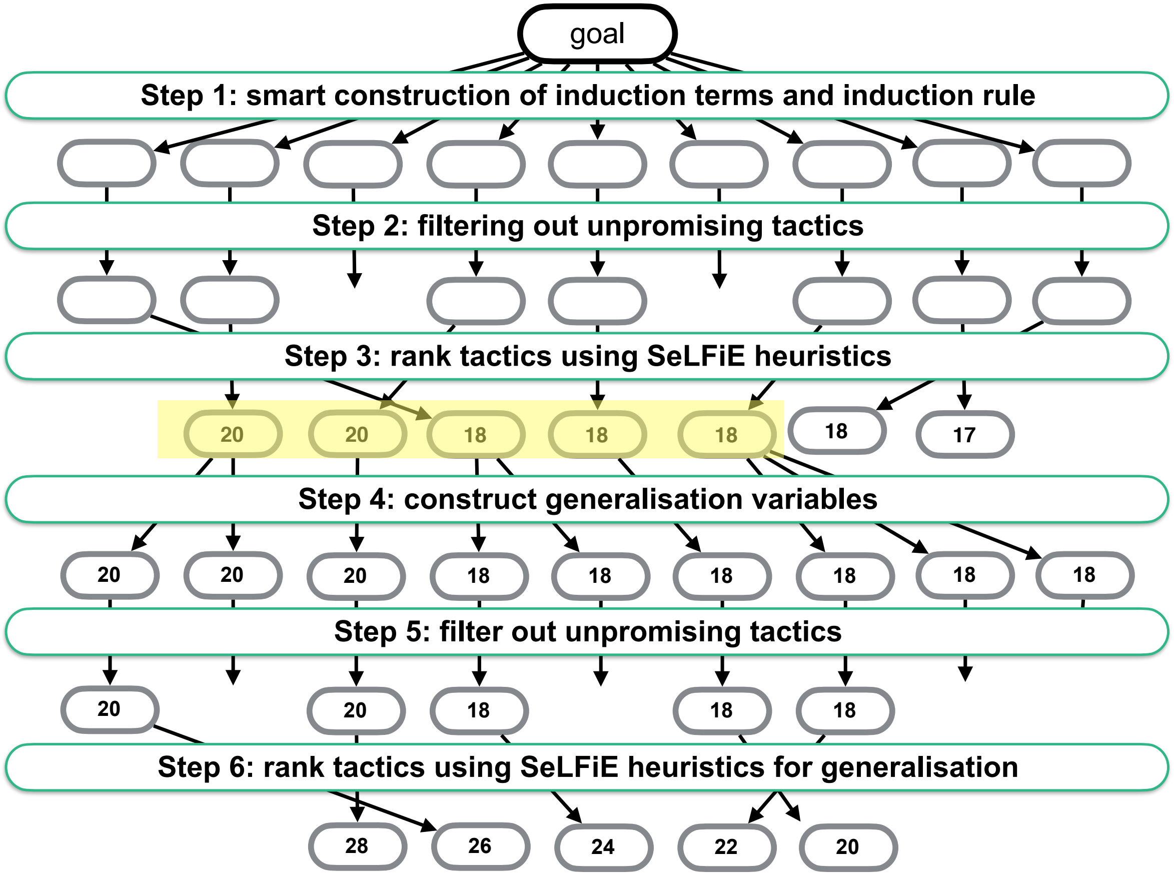


goal
Step 1: smart construction of induction terms and induction rule
Step 2: filtering out unpromising tactics
Step 3: rank tactics using SeLFiE heuristics
20
20
18
18
18
18
17
Step 4: construct generalisation variables
20
20
20
18
18
18
18
18
18
Step 5: filter out unpromising tactics

Build semantic_induct using SeLFiE

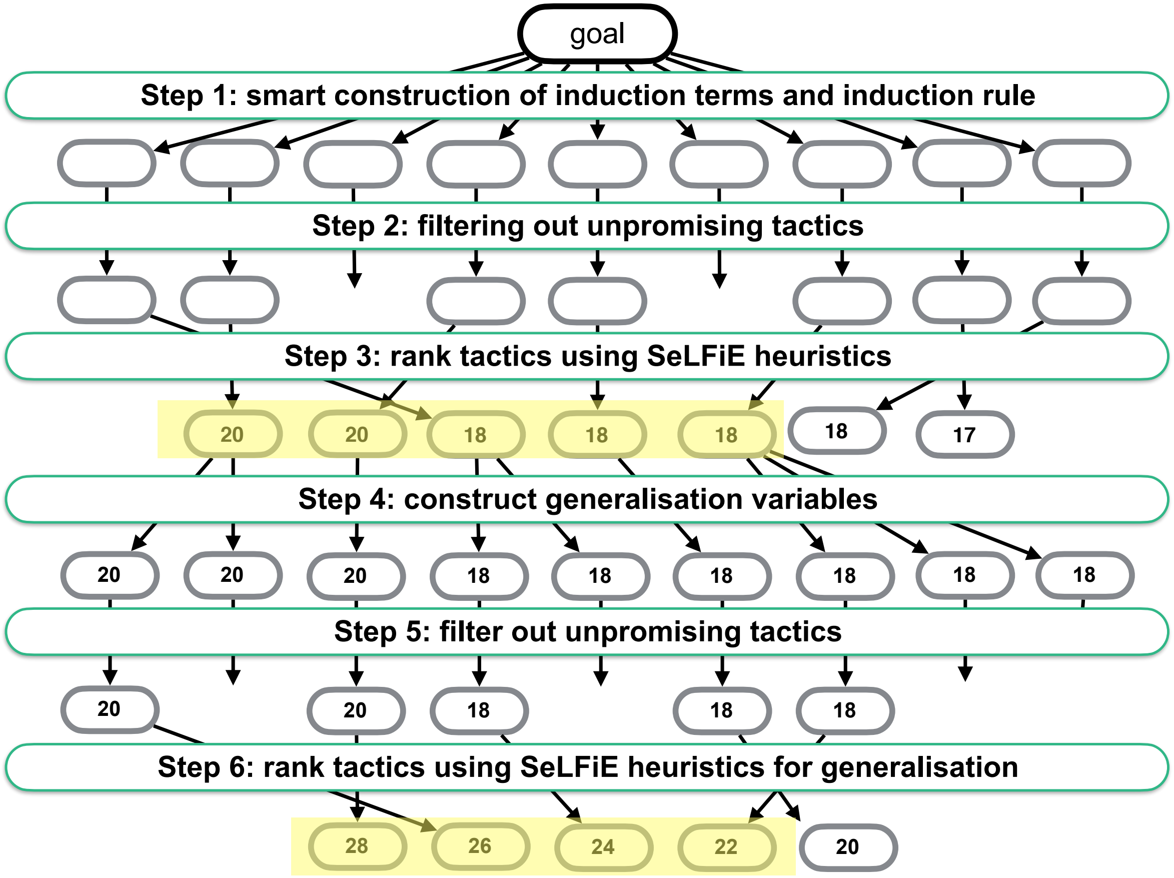

goal
Step 1: smart construction of induction terms and induction rule
Step 2: filtering out unpromising tactics
Step 3: rank tactics using SeLFiE heuristics
20
20
18
18
18
18
17
Step 4: construct generalisation variables
20
20
20
18
18
18
18
18
18
Step 5: filter out unpromising tactics
20
20
18
18
18
Step 6: rank tactics using SeLFiE heuristics for generalisation

Build semantic_induct using SeLFiE



goal
Step 1: smart construction of induction terms and induction rule
Step 2: filtering out unpromising tactics
Step 3: rank tactics using SeLFiE heuristics
20
20
18
18
18
18
17
Step 4: construct generalisation variables
20
20
20
18
18
18
18
18
18
Step 5: filter out unpromising tactics
20
20
18
18
18
Step 6: rank tactics using SeLFiE heuristics for generalisation
28
26
24
22
20

Build semantic_induct using SeLFiE



Step 1: smart construction of induction terms and induction rule

Step 2: filtering out unpromising tactics

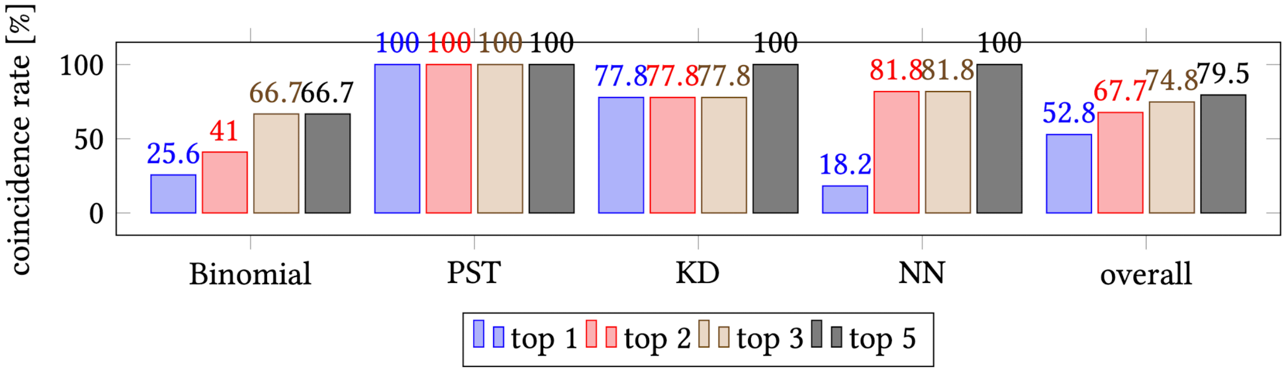
Step 3: rank tactics using SeLFiE heuristics

Step 4: construct generalisation variables

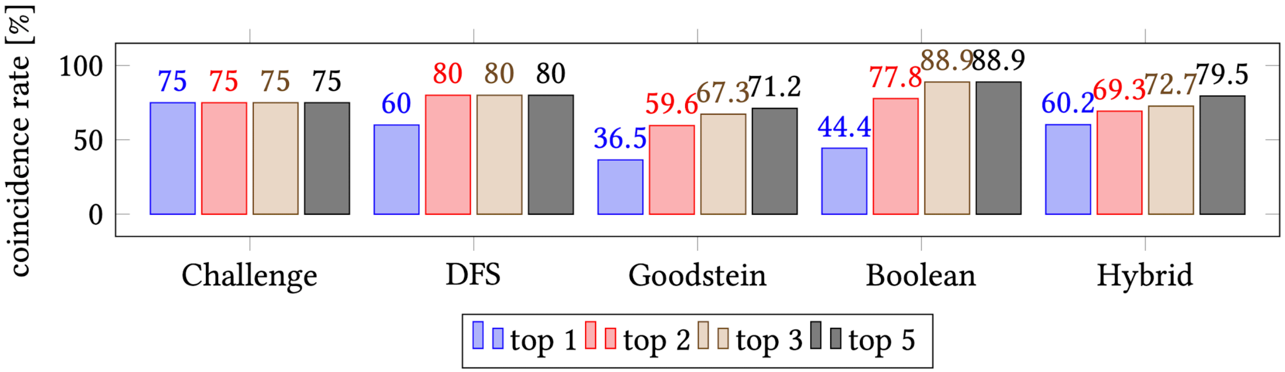
Step 5: filter out unpromising tactics

Step 6: rank tactics using SeLFiE heuristics for generalisation

recommendation using SeLFiE



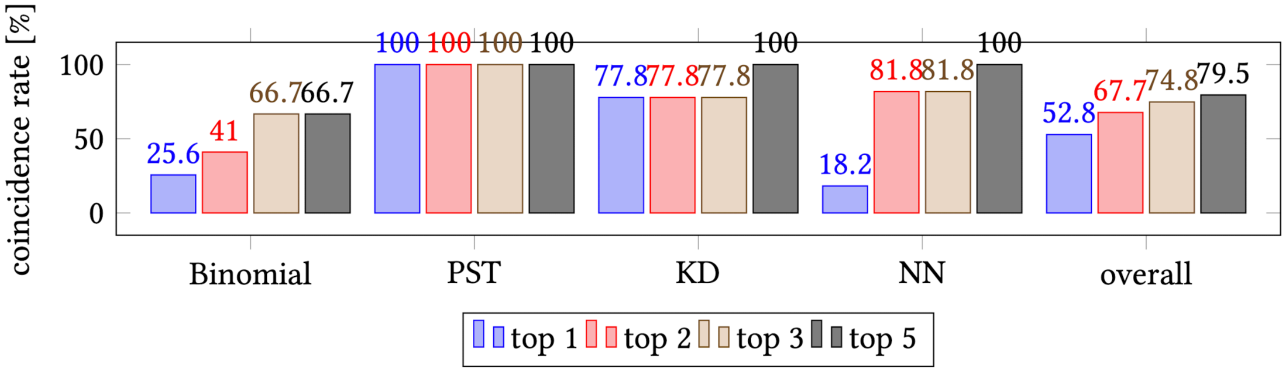
(b) Coincidence rates of semantic_induct for each theory file (part 1).



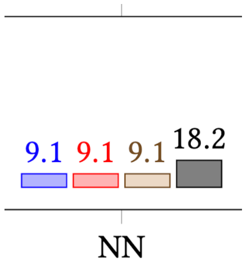
(d) Coincidence rates of semantic_induct for each theory file (part 2).

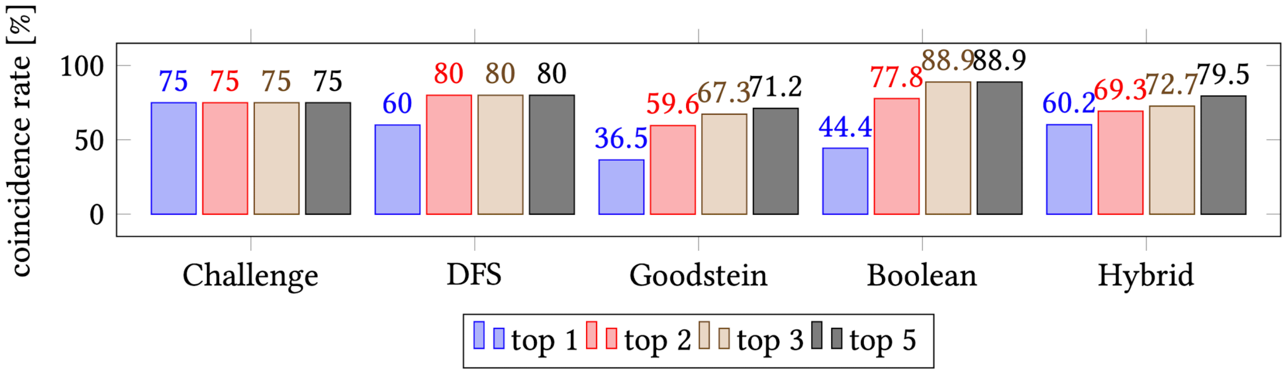
recommendation using SeLFiE

recommendation using LiFtEr



(b) Coincidence rates of semantic_induct for each theory file (part 1).

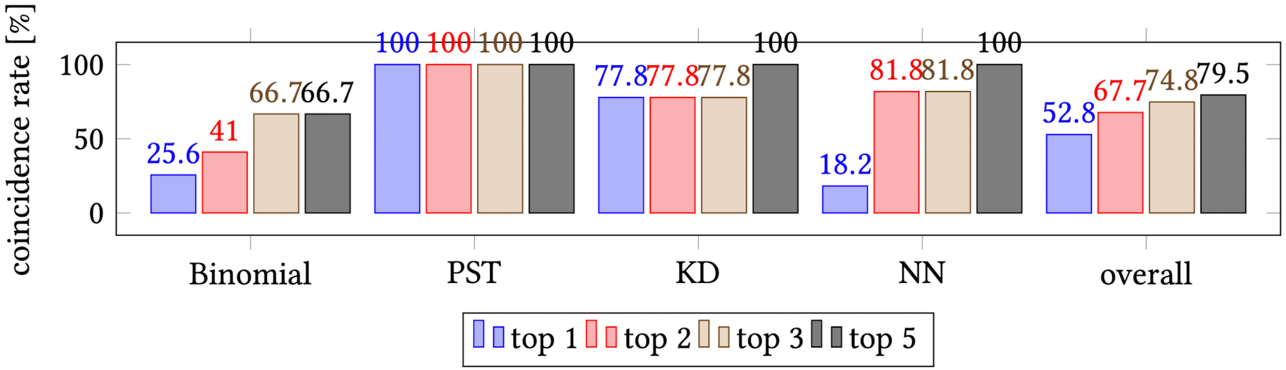


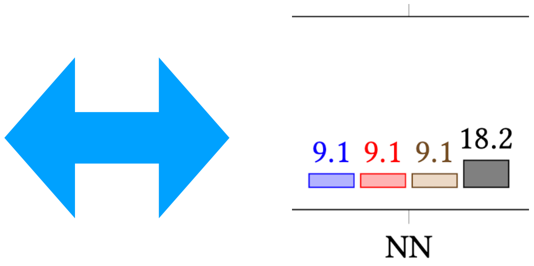


(d) Coincidence rates of semantic_induct for each theory file (part 2).

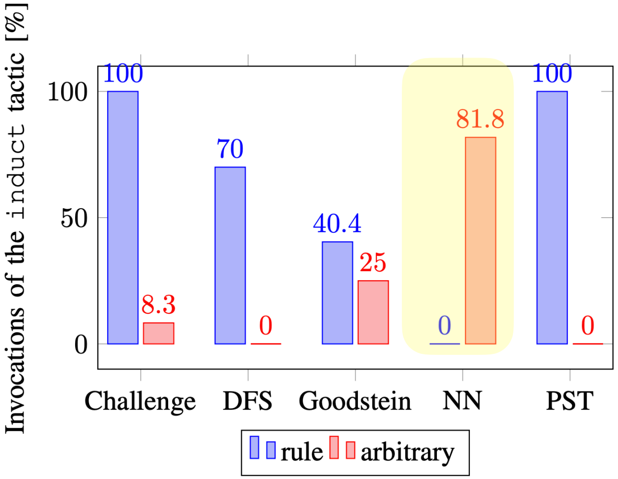
recommendation using SeLFiE

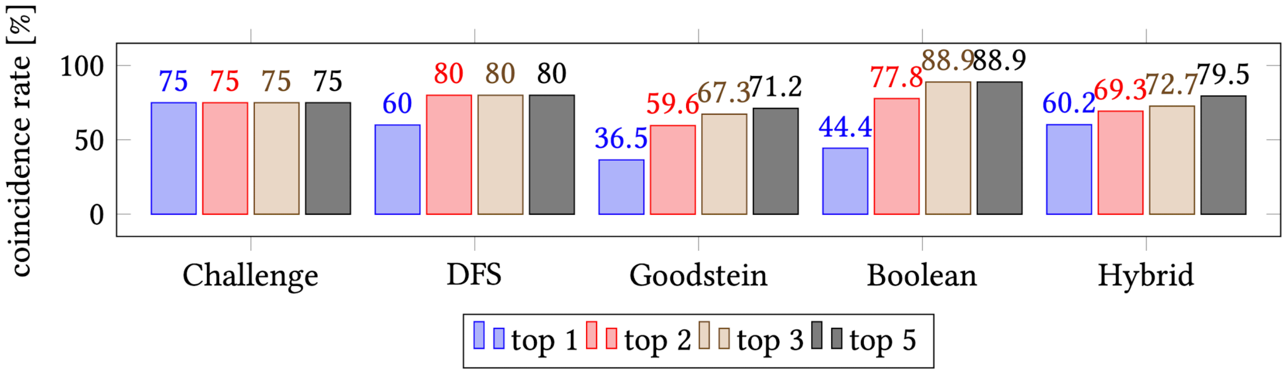
recommendation using LiFtEr





(b) Coincidence rates of semantic_induct for each theory file (part 1).



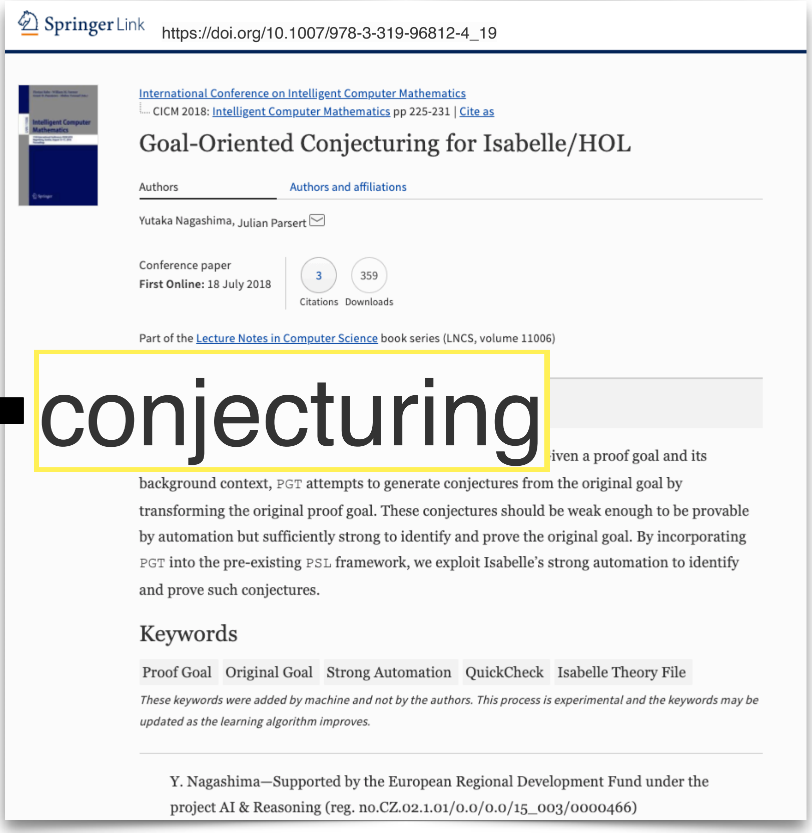


(d) Coincidence rates of semantic_induct for each theory file (part 2).

Future work

SeLFiE

Future work



Springer Link https://doi.org/10.1007/978-3-319-96812-4_19

International Conference on Intelligent Computer Mathematics

CICM 2018: Intelligent Computer Mathematics pp 225-231 | Cite as

Goal-Oriented Conjecturing for Isabelle/HOL

Authors Authors and affiliations

Yutaka Nagashima, Julian Parsert

Conference paper
First Online: 18 July 2018

3 Citations 359 Downloads

Part of the Lecture Notes in Computer Science book series (LNCS, volume 11006)

iven a proof goal and its background context, PGT attempts to generate conjectures from the original goal by transforming the original proof goal. These conjectures should be weak enough to be provable by automation but sufficiently strong to identify and prove the original goal. By incorporating PGT into the pre-existing PSL framework, we exploit Isabelle's strong automation to identify and prove such conjectures.

Keywords

Proof Goal Original Goal Strong Automation QuickCheck Isabelle Theory File

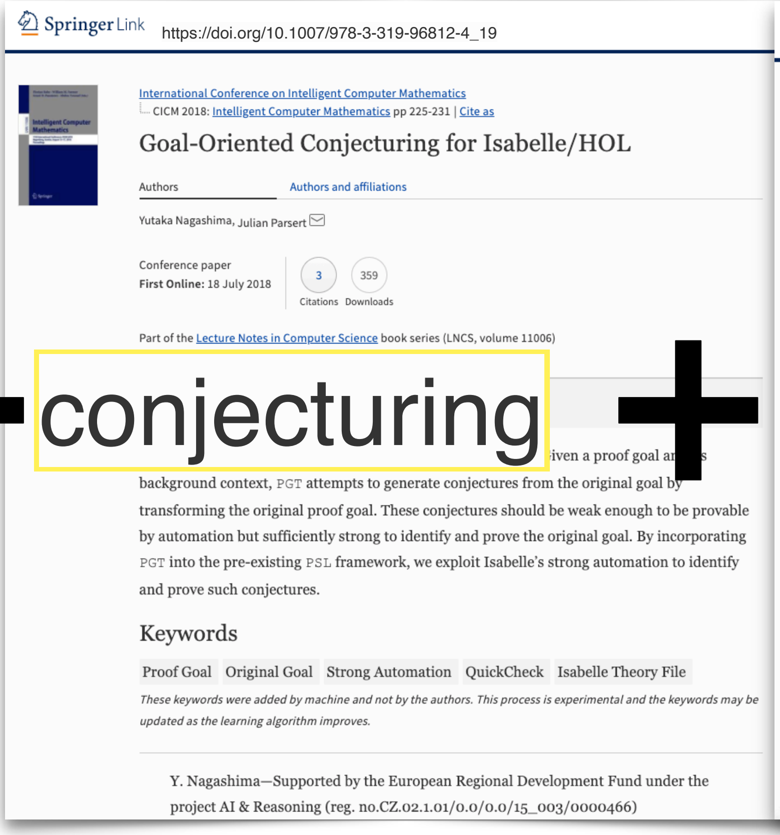
These keywords were added by machine and not by the authors. This process is experimental and the keywords may be updated as the learning algorithm improves.

Y. Nagashima—Supported by the European Regional Development Fund under the project AI & Reasoning (reg. no.CZ.02.1.01/0.0/0.0/15_003/0000466)

SeLFiE + conjecturing

Future work

SeLFiE + conjecturing + proof search



Springer Link https://doi.org/10.1007/978-3-319-96812-4_19

International Conference on Intelligent Computer Mathematics
CICM 2018: Intelligent Computer Mathematics pp 225-231 | Cite as

Goal-Oriented Conjecturing for Isabelle/HOL

Authors | Authors and affiliations

Yutaka Nagashima, Julian Parsert

Conference paper
First Online: 18 July 2018

3 Citations | 359 Downloads

Part of the Lecture Notes in Computer Science book series (LNCS, volume 11006)

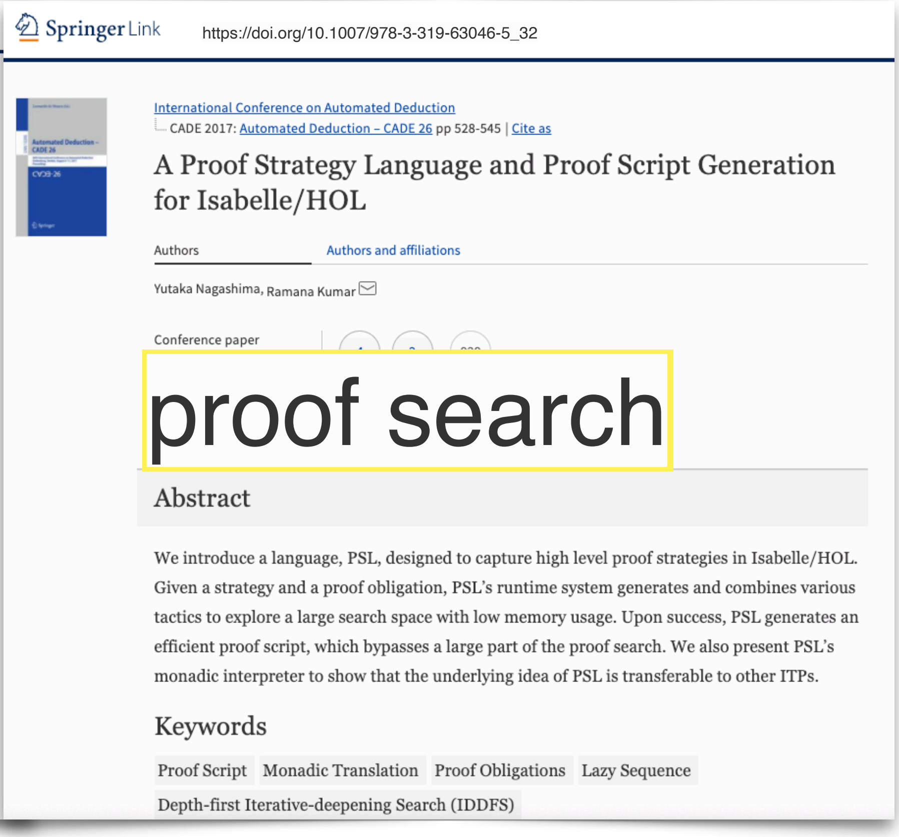
iven a proof goal ar s background context, PGT attempts to generate conjectures from the original goal by transforming the original proof goal. These conjectures should be weak enough to be provable by automation but sufficiently strong to identify and prove the original goal. By incorporating PGT into the pre-existing PSL framework, we exploit Isabelle's strong automation to identify and prove such conjectures.

Keywords

Proof Goal | Original Goal | Strong Automation | QuickCheck | Isabelle Theory File

These keywords were added by machine and not by the authors. This process is experimental and the keywords may be updated as the learning algorithm improves.

Y. Nagashima—Supported by the European Regional Development Fund under the project AI & Reasoning (reg. no.CZ.02.1.01/0.0/0.0/15_003/0000466)



Springer Link https://doi.org/10.1007/978-3-319-63046-5_32

International Conference on Automated Deduction
CADE 2017: Automated Deduction – CADE 26 pp 528-545 | Cite as

A Proof Strategy Language and Proof Script Generation for Isabelle/HOL

Authors | Authors and affiliations

Yutaka Nagashima, Ramana Kumar

Conference paper

Abstract

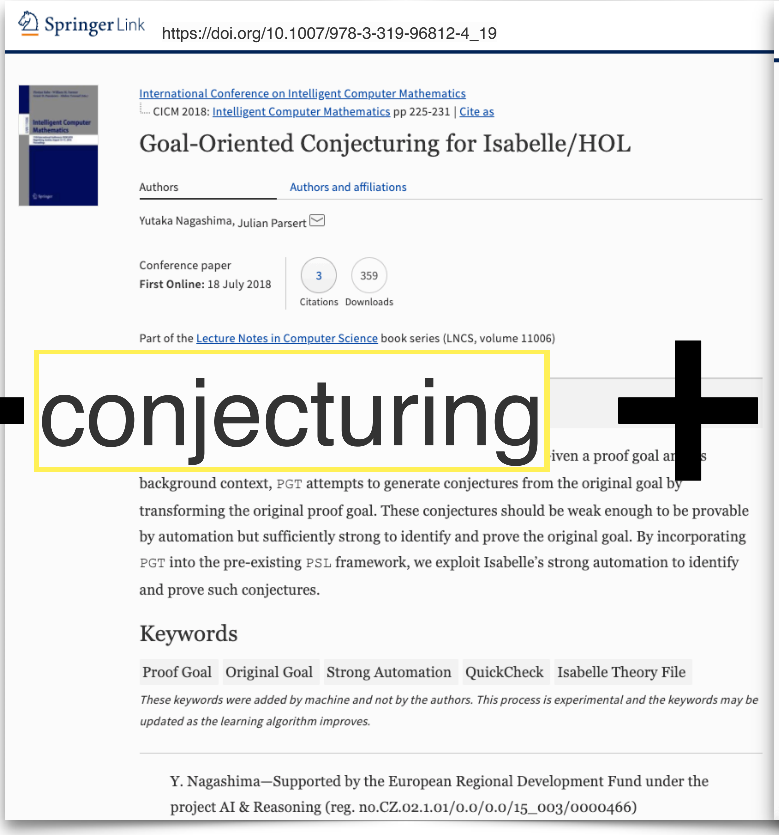
We introduce a language, PSL, designed to capture high level proof strategies in Isabelle/HOL. Given a strategy and a proof obligation, PSL's runtime system generates and combines various tactics to explore a large search space with low memory usage. Upon success, PSL generates an efficient proof script, which bypasses a large part of the proof search. We also present PSL's monadic interpreter to show that the underlying idea of PSL is transferable to other ITPs.

Keywords

Proof Script | Monadic Translation | Proof Obligations | Lazy Sequence | Depth-first Iterative-deepening Search (IDDFS)

Future work

SeLFiE + conjecturing + proof search



Springer Link https://doi.org/10.1007/978-3-319-96812-4_19

International Conference on Intelligent Computer Mathematics
CICM 2018: Intelligent Computer Mathematics pp 225-231 | Cite as

Goal-Oriented Conjecturing for Isabelle/HOL

Authors | Authors and affiliations

Yutaka Nagashima, Julian Parsert

Conference paper
First Online: 18 July 2018

3 Citations | 359 Downloads

Part of the Lecture Notes in Computer Science book series (LNCS, volume 11006)

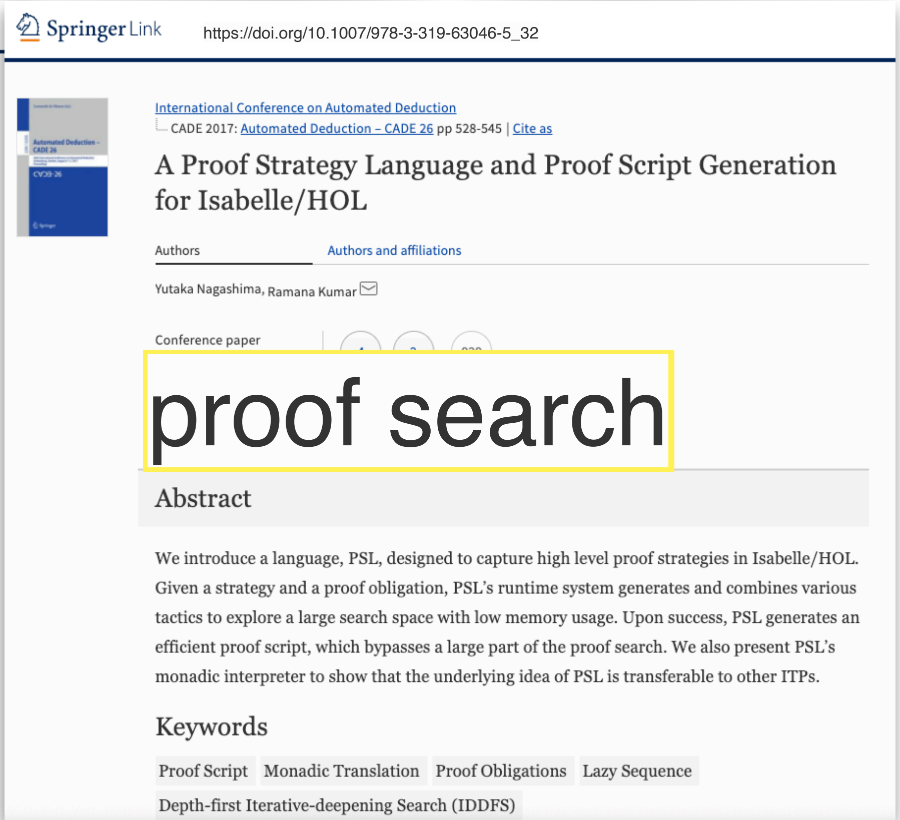
iven a proof goal an s background context, PGT attempts to generate conjectures from the original goal by transforming the original proof goal. These conjectures should be weak enough to be provable by automation but sufficiently strong to identify and prove the original goal. By incorporating PGT into the pre-existing PSL framework, we exploit Isabelle's strong automation to identify and prove such conjectures.

Keywords

Proof Goal | Original Goal | Strong Automation | QuickCheck | Isabelle Theory File

These keywords were added by machine and not by the authors. This process is experimental and the keywords may be updated as the learning algorithm improves.

Y. Nagashima—Supported by the European Regional Development Fund under the project AI & Reasoning (reg. no.CZ.02.1.01/0.0/0.0/15_003/0000466)



Springer Link https://doi.org/10.1007/978-3-319-63046-5_32

International Conference on Automated Deduction
CADE 2017: Automated Deduction – CADE 26 pp 528-545 | Cite as

A Proof Strategy Language and Proof Script Generation for Isabelle/HOL

Authors | Authors and affiliations

Yutaka Nagashima, Ramana Kumar

Conference paper

Abstract

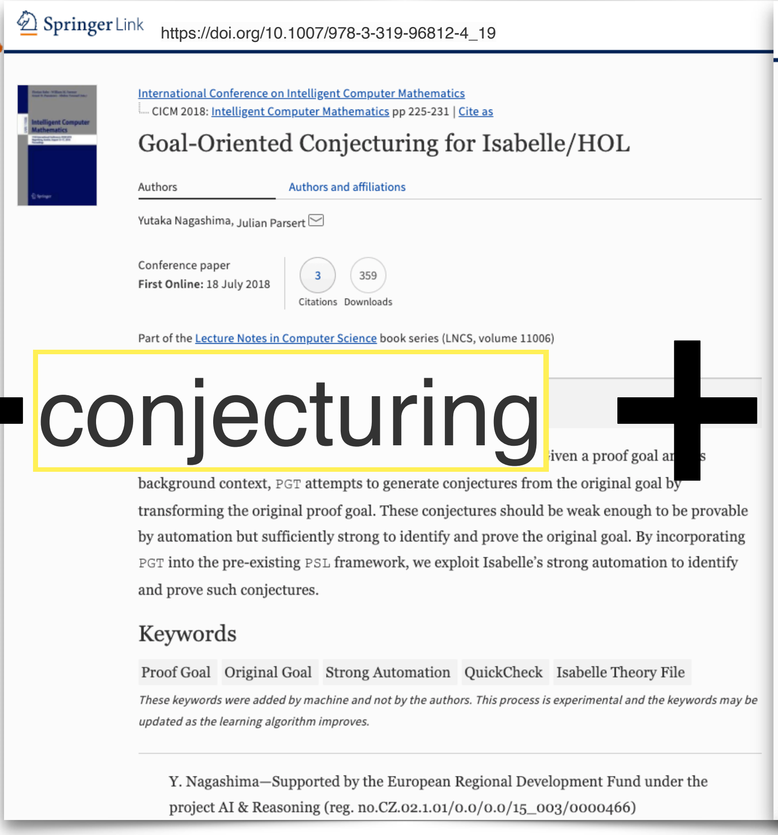
We introduce a language, PSL, designed to capture high level proof strategies in Isabelle/HOL. Given a strategy and a proof obligation, PSL's runtime system generates and combines various tactics to explore a large search space with low memory usage. Upon success, PSL generates an efficient proof script, which bypasses a large part of the proof search. We also present PSL's monadic interpreter to show that the underlying idea of PSL is transferable to other ITPs.

Keywords

Proof Script | Monadic Translation | Proof Obligations | Lazy Sequence | Depth-first Iterative-deepening Search (IDDFS)

==> fully automatic inductive prover in Isabelle/HOL

Future work



Springer Link https://doi.org/10.1007/978-3-319-96812-4_19

International Conference on Intelligent Computer Mathematics
CICM 2018: Intelligent Computer Mathematics pp 225-231 | Cite as

Goal-Oriented Conjecturing for Isabelle/HOL

Authors Authors and affiliations

Yutaka Nagashima, Julian Parsert ✉

Conference paper
First Online: 18 July 2018

3 Citations 359 Downloads

Part of the Lecture Notes in Computer Science book series (LNCS, volume 11006)

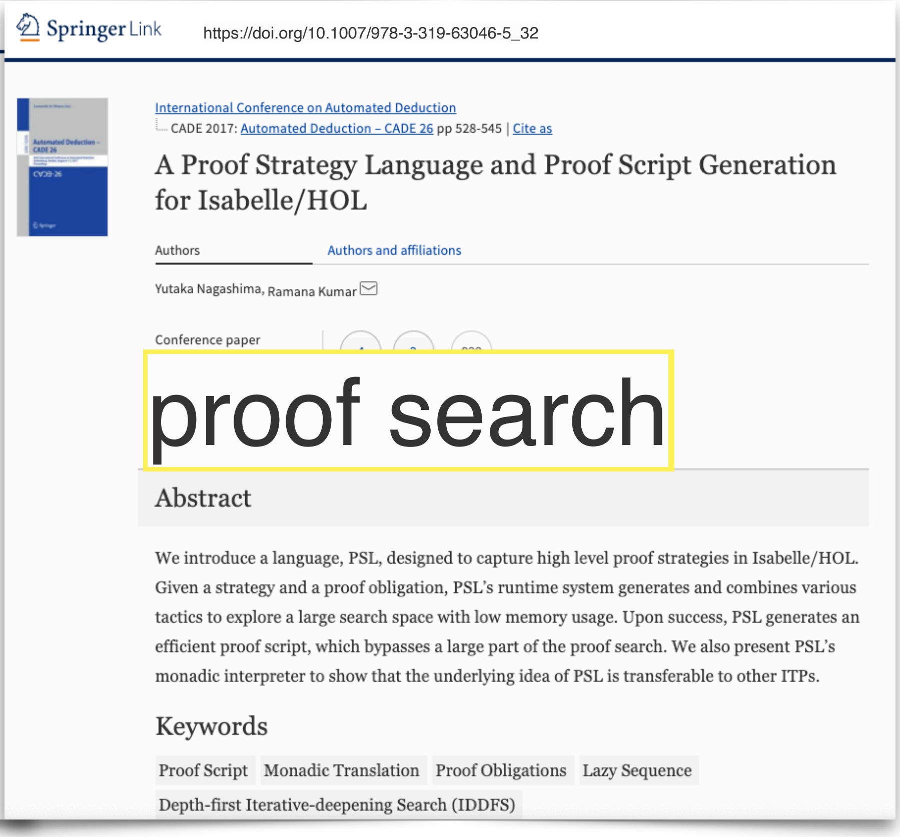
iven a proof goal ar s background context, PGT attempts to generate conjectures from the original goal by transforming the original proof goal. These conjectures should be weak enough to be provable by automation but sufficiently strong to identify and prove the original goal. By incorporating PGT into the pre-existing PSL framework, we exploit Isabelle's strong automation to identify and prove such conjectures.

Keywords

Proof Goal Original Goal Strong Automation QuickCheck Isabelle Theory File

These keywords were added by machine and not by the authors. This process is experimental and the keywords may be updated as the learning algorithm improves.

Y. Nagashima—Supported by the European Regional Development Fund under the project AI & Reasoning (reg. no.CZ.02.1.01/0.0/0.0/15_003/0000466)



Springer Link https://doi.org/10.1007/978-3-319-63046-5_32

International Conference on Automated Deduction
CADE 2017: Automated Deduction – CADE 26 pp 528-545 | Cite as

A Proof Strategy Language and Proof Script Generation for Isabelle/HOL

Authors Authors and affiliations

Yutaka Nagashima, Ramana Kumar ✉

Conference paper

Abstract

We introduce a language, PSL, designed to capture high level proof strategies in Isabelle/HOL. Given a strategy and a proof obligation, PSL's runtime system generates and combines various tactics to explore a large search space with low memory usage. Upon success, PSL generates an efficient proof script, which bypasses a large part of the proof search. We also present PSL's monadic interpreter to show that the underlying idea of PSL is transferable to other ITPs.

Keywords

Proof Script Monadic Translation Proof Obligations Lazy Sequence Depth-first Iterative-deepening Search (IDDFS)

SeLFiE + conjecturing + proof search

==> fully automatic inductive prover in Isabelle/HOL