# Artificial Intelligence and Domain-Specific Languages for Interactive Theorem Proving

**cumulative dissertation**

by

**Yutaka Nagashima**

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements
for the degree of academic degree

advisors: Assoc.-Prof. Dr. Cezary Kaliszyk

**Innsbruck, 5 December 2020**

universität innsbruck

cumulative dissertation

# Artificial Intelligence and Domain-Specific Languages for Interactive Theorem Proving

Yutaka Nagashima (11744794)
yutaka.nagashima@cvut.cz

5 December 2020

**advisors:** Assoc.-Prof. Dr. Cezary Kaliszyk

# Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

04. December. 2020
Datum


Unterschrift

# Abstract

As software systems are forming the essential infrastructures of human societies, we have to make them trustworthy. A prominent approach to building reliable systems is complete formal verification using proof assistants.

Proof assistants are software systems that facilitate interactive proof developments. In a complete formal verification project, software developers specify what they want in a proof assistant and prove that their implementation is correct in terms of the specification.

This method is known to be able to remove software failures, however, interactive proof developments are laborious processes that require domain-specific expertise in the proof assistant of our choice.

This dissertation demonstrates how to alleviate such burdensome processes for a particular proof assistant, Isabelle/HOL. Isabelle/HOL is one of the most commonly used proof assistants and comes with various sub-tools to expedite proof developments. Experienced Isabelle users know how to best exploit such sub-tools, whereas new Isabelle users are usually overwhelmed by the number of available sub-tools. Furthermore, even for experienced Isabelle users it is still necessary to carefully investigate the proof goal at hand to adjust the appropriate sub-tool for that goal.

I automate some of such onerous processes using artificial intelligence and domain-specific languages: domain-specific languages allow experienced users to encode their know-hows, and artificial intelligence tries to identify the promising sub-tool and recommends how to use the tool. Evaluations show that my meta-tools correctly predict experts' use of Isabelle's sub-tools, improving the user-experience of Isabelle/HOL.

# Acknowledgements

Cezary Kaliszyk supervised me for my PhD studies at the University of Innsbruck, while Josef Urban employed me as a junior researcher at the Czech Technical University in Prague. Gerwin Klein prepared me for academic work at Data61 and NICTA, where I had the pleasure to work with Yilun He and Pang Luo.

Makarius Wenzel documented how to program in Isabelle/ML, which immensely helped me implement my ideas. Jasmin Blanchette gave me detailed explanations on Mirabelle, an evaluation tool in Isabelle/HOL, to evaluate `try_hard`. Ekaterina Komendantskaya helped me improve my manuscript on `LiFtEr`. Many anonymous reviewers of my manuscripts provided me with constructive feedback.

This work was 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) and ERC Starting grant no. 714034 *SMART*.

My family caused enough distractions from research, with which I managed not to be overwhelmed by theorem proving.

# Contents

# Chapter 1

# Introduction

> Zu diesem Allen kommt, daß zu Papier gebrachte Gedanken überhaupt nichts weiter sind, als die Spur eines Fußgängers im Sande: man sieht wohl den Weg, welchen er genommen hat; aber um zu wissen, was er auf dem Wege gesehn, muß man seine eigenen Augen gebrauchen.
>
> *Arthur Schopenhauer*

## 1.1 Artificial Intelligence for Theorem Proving

This dissertation presents how I improved the user-interface of an interactive theorem prover, Isabelle/HOL, using artificial intelligence and domain-specific languages. Interactive proof assistants (ITPs) are software systems that assist human researchers to formulate mathematical concepts and prove them. Therefore, ITPs are also known as proof assistants. Isabelle/HOL is an ITP based on a classical higher-order logic and is known for its powerful sub-tools that expedite proof developments.

I improved the user-interface of Isabelle/HOL because proof automation for higher-order logic is an important problem of both theoretical interest and practical necessity.

On the theoretical side, theorem proving allows machines to fully capture the formal definitions of mathematical objects provided through user interaction. This forms a vibrant contrast to the conventional machine learning techniques that are often based on inductive reasoning: the conclusions of typical machine learning algorithms are based on statistical inference and their correctness is subject to probability, while the conclusions of mechanically proved theorems are in theory 100% correct under their formally stated assumptions.

On the practical side, theorem proving is becoming the norm of reliable systems programming. Researchers and engineers use ITPs to mechanically specify their system requirements and prove that their implementations are correct in terms of the specifications. ITPs are also forming the basis of formal scientific development. For example, well-known hard mathematical propositions, such as the four color theorem and the Kepler conjecture, have been mechanically proved in ITPs.

Despite the recent progress in proof automation, writing mechanical proofs still requires engineers' expertise and is labor intensive: theorem proving in higher-order logic is in general undecidable, and any deterministic algorithm quickly faces combinatorial explosion even for conceptually straight-forward conjectures. For this reason, one has to resort to heuristic solutions. Fortunately, many proof obligations require "shallow" heuristics in practice: since we already have many useful tools called *tactics* for writing proofs, we can prove many conjectures by specializing and combining tactics.

That is why this dissertation introduces approaches to improving ITPs' user-interface using AI and DSLs, so that we can build a system that is rigorous, expressive, and capable of autonomous reasoning by integrating the benefits of theorem proving and artificial intelligence.

## 1.2 Isabelle/HOL, the Target Proof Assistant

There are a number of proof assistants available. Arguably, Coq [TCdt] has the largest user base. Coq is based on an expressive formal language called the Calculus of Inductive Constructions [PM14]. There is also a group of proof assistants, called the HOL family, which includes HOL Light [Har96], HOL [SN08], HOL Zero [Ada16], PVS [ORS92], and Isabelle/HOL [NPW02]. These proof assistants are based on a variant of classical higher-order logic.

Originally, these proof assistants used to admit procedural proof scripts only: human engineers keep applying commands, called *tactics*, one by one to reduce proof goals into simpler sub-goals until no new sub-goals arise. The resulting proof scripts are, therefore, difficult to read and maintain.

Mizar [GKN10] addressed this problem by introducing a declarative style: it comes with reserved keywords that resemble English, such as `assume` and `by`. Using these keywords, human engineers develop proof scripts that are more intuitive to humans and easy to read and maintain.

Inspired by Mizar, Wenzel developed a proof language, Isar [Wen02], for Isabelle. Isar allows us to intertwine the procedural tactic-style proofs with the Mizar-like declarative proof style as well as definitions and specifications in one framework. Furthermore, the syntax of Isar is *extensible*: it provides an infrastructure to build other tools for Isabelle and integrate such tool as its sub-components [WW07].

I choose Isabelle/HOL as my target proof assistant for several reasons. Firstly, the extensible syntax of Isar lets me develop new tools as plug-ins. This allows users to easily install my tools using the standard Isabelle command and call the tools as parts of Isar. Thus, all tools presented in this dissertation are implemented as plug-ins, which one can use as parts of Isabelle/Isar.

Secondly, Isabelle/HOL comes with a number of sub-tools, such as proof tactics, Sledgehammer [BBP11] and counterexample finders [Bul12, Bla10]. These powerful sub-tools can handle non-trivial proof goals; however, each sub-tool has weakness as well as strength, and it requires Isabelle-specific expertise to decide how to use which sub-tool for what kind of problem. The tools presented in this dissertation address these issues,

so that Isabelle users can best exploit the already powerful Isabelle sub-tools.

Lastly, Isabelle/HOL comes with a large proof corpora called the Archive of Formal Proofs (AFP) [KNPT04]. The AFP is an online repository where Isabelle users submit their proof documents. All submitted proof documents are peer reviewed to maintain the quality of entries. Currently, the AFP hosts 548 articles written by in total 360 authors, with more than 148,000 lemmas and 2,584,000 lines of code. This large repository is useful to train my machine learning algorithms to learn how to write proof documents in Isabelle/HOL.

## 1.3 Need for Suitable Representations of Proof Heuristics

There are already a number of projects that attempt to improve proof assistants or automatic theorem provers using artificial intelligence. For example, Sledgehammer, a sub-tool of Isabelle/HOL itself, uses machine learning to select relevant facts [KBKU13].

One aspect that differentiates my work from other AI projects to improve theorem provers is that my approach exploits Isabelle users expert knowledge explicitly by choosing or developing suitable languages to represent their expertise.

Such representations of expertise have to satisfy the following two requirements:

1. they should not require a large number of similar data points, and
2. they should not be specific to particular problems.

From the perspective of theorem proving, these two criteria are the achievements of expressive logics: when formulating theorems and proving them in an expressive logic, engineers prefer to describe general concepts that can cover a large number of concrete cases. Such abstractions let engineers handle concrete cases as instances of the general concepts.

Consider, for example, the following formalisation of `list` from Isabelle's standard library.

```
datatype 'a list =
  Nil                    ("[]")
| Cons "'a" "'a list"    (infixr "#" 65)
```

In this formalisation, the `list` type constructor is defined using parametric polymorphism, and the concrete type of `list` depends on the type variable expressed as `'a`. For instance, we can instantiate `'a` with `char` to express lists of characters or with `int` to express lists of integers. This way, parametric polymorphism allows us to reason about lists in general instead of dealing with lists of particular types one by one. Based on this formalisation of the `list` constructor, the standard library defines a function to append two lists and the associative property of this function as follows:

```
primrec append :: "'a list ⇒ 'a list ⇒ 'a list" (infixr "@" 65) where
  append_Nil:  "[]     @ ys = ys"
```

```
| append_Cons: "(x#xs) @ ys = x # xs @ ys"

lemma associative: "(xs @ ys) @ zs = xs @ (ys @ zs)"
  apply(induct xs)
   apply auto
  done
```

where the variables (`xs`, `ys`, and `zs`) are universally quantified implicitly. That is, this `append` function is defined for all lists of any number of arbitrary elements of any type. Therefore, the only restriction that lists in `lemma associative` has to satisfy is that all elements in the lists under consideration have to be of the same type. For this reason, once we prove the generic associative property for lists as `lemma associative`, we can derive the associative properties of concrete lists as special cases of `lemma associative` as follows:

```
lemma "([1] @ [2]) @ [3] = [1] @ ([2] @ [3])"
  by (intro append_assoc)

lemma "([''a''] @ [''b'']) @ [''c''] = [''a''] @ ([''b''] @ [''c''])"
  by (intro append_assoc)
```

This way, when we use an expressive logic experienced scientists replace many similar theorems sharing the same nature with one general formula. And concise presentation of knowledge is what is desired in science, and that is why scientists prefer to use expressive logics.

But from the perspective of AI application to theorem proving, this expressiveness is what causes the aforementioned two criteria: since similar concrete cases are represented in one general formula, we should not have similar theorems and proofs in a high quality proof database. If two theorems share a certain nature, we should represent them in one general formula and treat the two theorems as instances of the general formula.

This expressiveness also leads to the second criterion about the desirable representation of heuristics. Since we can handle similar concrete cases as instances of a general formula, whenever we need support from AI tools, these tools have to provide recommendations for a new problem that is not similar to existing ones.

In our example, we already have the associative property for all lists. Therefore, we do not need AI support to prove the associative property for lists of integers or lists of strings. But we might need AI supports when we try to prove the associative property of new operations defined on new types (e.g. the plus operator) .

This means if we want to exploit human engineers' heuristics from the proof of associative property of the `append` function for lists to prove the associative property of a new operation, such as the plus operator on natural numbers, we have to choose a representation that is not specific to the `list` constructor or the `append` function.

## 1.4 Three Kinds of Representations for Proof Heuristics

For this reason, I employed three kinds of representations to encode proof heuristics in this dissertation. In this section, I introduce these three styles briefly. The details of each representation is expounded in the corresponding chapters.

**Proof Strategy Language.** Chapter 3 introduces `PSL`, Proof Strategy Language. Using `PSL`, Isabelle users can encode their rough idea about how to attack proof goals as a strategy. Strategies written in `PSL` are not specific to concrete proof goals but are abstract descriptions of procedural proof heuristics. When applied to a proof goal, `PSL`'s interpreter produces variants of proof tactics from a strategy and attempts to solve the goal.

For instance, if we apply a proof strategy about proof by induction to the aforementioned proof goal about the associative property of lists, `PSL`'s interpreter produces variants of the induction tactic that are parametrized by arguments such as `(induct xs)` and `(induct xs arbitrary:ys)`.

The advantage of this approach is that the interpreter produces a combination of tactics with which Isabelle can discharge a proof goal if `PSL`'s interpreter succeeds to complete a proof search based on a given strategy. The drawback of this approach is that when applied to difficult proof goals the search space specified by a strategy as a procedural heuristic tends to explode and the interpreter fails to complete a proof search.

**Elaborate Feature Extractor in Standard ML** To complement the weakness of `PSL`, Chapter 6 introduces `PaMpeR`. `PaMpeR` is a recommendation tool for proof tactics that does not rely on a proof search. `PaMpeR` learns which proof tactic to apply to what kind of proof goals from the Archive of Formal Proofs. However, when creating a database which matches proof tactic names and proof goals `PaMpeR` applies an elaborate feature extractor to the proof goals, converting each proof goal into an array of boolean values.

This feature extractor is hand-written in Isabelle's implementation language, Standard ML. And it only analyzes the concrete information defined in Isabelle's standard library and the meta-information related to proof goals. Strictly speaking, analyzing concrete information defined in Isabelle's standard library is an exception to the aforementioned second criterion. However, some language constructs defined in the standard library, such as the list type constructor, appear in many projects across problem domains. Therefore, I decided that it is valuable to have heuristics about concrete information defined in the standard library.

To acquire meta-information, `PaMpeR` mainly analyzes how each construct is defined in the underlying context. `PaMpeR`'s feature extractor, for example, checks which Isabelle keyword was used to define constants appearing in the proof goal. Our ongoing example about the associative property of list involves the `append` function, which is defined with the `primrec` keyword. This keyword is used to define new constants using primitive recursion. Therefore, when `PaMpeR`'s feature extractor processes this proof, it tags the name of proof tactic, `induct`, with the information that this proof goal involves a constant defined through primitive recursion.

Note that such feature extractor for the meta-information can analyze not only proof goals that are available today, but also new goals that will be developed in future using new constants and types that do not even exist yet.

The evaluation results in Chapter 6 show that `PaMpeR` can recommend certain proof tactics accurately without resorting to proof search. However, it does not recommend arguments of proof tactics. This is a serious limitation of `PaMpeR`, as the choice of arguments is critical to effective proof developments for some tactics, notably the `induct` tactic.

**`LiFtEr`: Logical Feature Extractor.** To address this problem for the `induct` tactic we developed our domain-specific language, `LiFtEr`. `LiFtEr` stands for logical feature extractor and offers logical connectives such as implication, conjunction, and existential and universal quantifiers. These connectives provide the abstraction that allows experienced users to encode induction heuristics without referring to concrete constructs, such as constants or variable names. Since induction heuristics written in `LiFtEr` do not depend on problem specific constructs, they can transcend problem domains. Chapter 4 provides more thorough explanations on `LiFtEr`. And Chapter 5 introduces an automatic recommendation tool for the arguments of the `induct` tactic, using induction heuristics encoded in `LiFtEr`.

## 1.5 Inductive Theorem Proving

As the previous section implies, a significant part of this dissertation explains approaches to automating proof by induction in Isabelle/HOL. The automation of proof by induction is one of the largest remaining challenge in mechanised theorem proving. In particular, the automation of inductive theorem proving of arbitrary problem domains is a known open question.

Traditionally, this problem was addressed mainly by the automated first-order theorem provers, in which one introduces induction axioms to handle proof by induction.

Inductive theorem provers are often based on logics expressive enough to handle proof by induction without introducing additional axioms. Isabelle/HOL is no exception here. In fact, Isabelle offers proof tactics, with which users can specify how to apply proof by induction intuitively. However, there are usually infinitely many ways to apply the induction tactic for a given inductive problem. What is worse, for many cases an application of the induction tactic can be both inappropriate and safe at the same time: an inappropriate application of the induction tactic would transform an inductive problem into a format that is more difficult to prove, yet if the application is safe the resulting new proof goals are still provable assuming the original inductive problem is provable. Therefore, when an application of induction tactic is both inappropriate and safe, counterexample finders cannot detect that the proof attempt has gone awry, causing the explosion of search space.

In this dissertation, I present a guided brute-force approach in Chapter 3 to automate relatively easy proof by induction in Isabelle/HOL, and Chapter 4 introduces a

domain-specific language, `LiFtEr`, to encode induction heuristics, so that we can mechanically check if an application of the induction tactic is likely to be appropriate or not independently of the safety of the application. `smart_induct` in Chapter 5 uses `LiFtEr` and recommends how one should use `induct` for a given inductive problem. Chapter 8 describes an approach to identifying an auxiliary lemma useful to prove a given inductive problem using abductive reasoning. In Chapter 9, I envision a strong inductive prover that takes advantages of the approaches presented in this dissertation.

# Chapter 2

# Contributions

Consider the following two definitions of the reverse function for lists presented in the Isabelle/HOL tutorial [NPW02]:

```
primrec append :: "'a list ⇒ 'a list ⇒ 'a list" (infixr "@" 65) where
  append_Nil : "    [] @ ys = ys"
| append_Cons: "(x#xs) @ ys = x # xs @ ys"

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

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

where `#` is the list constructor, `[x]` is a syntactic sugar for `x # []`, and `@` is defined as an infix operator for `append`. How would you prove the following equivalence in Isabelle/HOL?

```
lemma itrev_is_rev: "itrev xs [] = rev xs"
```

One way to prove this lemma is to prove a more generic lemma and use it as an auxiliary lemma to prove our final goal:

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

lemma itrev_is_rev: "itrev xs [] = rev xs"
  by (simp add: helper)
```

This small proof script reveals the following three key challenges of theorem proving in Isabelle/HOL:

- `Task A`: Proof authors choose adequate proof methods for each step in the proofs.

- `Task B`: Proof authors pass arguments to the method to handle each proof goal.
- `Task C`: Proof authors specify auxiliary lemmas to prove the original goal.

In my PhD studies, I automated some of these processes using artificial intelligence and domain-specific languages. My research contributions resulted in the following six papers compiled into this thesis.

- A proof strategy language and proof script generation for Isabelle/HOL presented as Chapter 3
- Goal-oriented conjecturing for Isabelle/HOL as Chapter 8
- PaMpeR: proof method recommendation system for Isabelle/HOL as Chapter 6
- Simple dataset for proof method recommendation in Isabelle/HOL as Chapter 7
- LiFtEr: Language to encode induction heuristics for Isabelle/HOL as Chapter 4
- Smart induction for Isabelle/HOL (tool paper) as Chapter 5

I am the first author and main contributor of these six publications. In the following, I expound how each publication addressed the aforementioned three challenges and what my contribution was for each publication.

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

### Publication Details

> Yutaka Nagashima and Ramana Kumar. A proof strategy language and proof script generation for Isabelle/HOL. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 528–545. Springer, 2017

In this chapter, I automated `Task A` and `Task B` for relatively easy proof goals by introducing `PSL`. `PSL` is a domain-specific language 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. I also present `PSL`'s monadic interpreter to show that the underlying idea of `PSL` is transferable to other ITPs.

### My Contributions

I came up with the idea of proof strategy language as a meta-language for Isabelle's tactic language and implemented the `PSL` interpreter. I conducted the evaluations of `PSL`'s default strategy, `try_hard` based on course work assignments and selected theory files from the Archive of Formal Proofs (AFP) [KNPT04]. I wrote most of the final version of the manuscript with support from Ramana Kumar.

## 2.2 LiFtEr: Language to encode induction heuristics for Isabelle/HOL

### Publication Details

> Yutaka Nagashima. LiFtEr: Language to encode induction heuristics for Isabelle/HOL. In *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings*, pages 266–287, 2019

In this chapter, I addressed `Task B` for relatively difficult inductive problems, which brute-force search tools cannot prove in a realistic timeout, by designing and implementing a domain-specific language, `LiFtEr`. `LiFtEr` allows experienced Isabelle users to encode their induction heuristics in a style independent of any problem domain. When new Isabelle users face an inductive problem, `LiFtEr`'s interpreter mechanically checks if a given combination of arguments to the `induct` tactic matches the heuristics with respects to the inductive problem. Later, `LiFtEr` was used to implement `smart_induct` [Nag20e], a recommendation tool for inductive problems in Isabelle/HOL.

### My Contributions

I was the sole author of this chapter: I came up with the idea of developing a domain-specific language resembling the first-order logic to encode induction heuristics for Isabelle/HOL, implemented its interpreter, and wrote the paper independently with the supports from Ekaterina Komendantskaya as the shepherd assigned by the program committee.

## 2.3 Smart Induction for Isabelle/HOL (Tool Description)

### Publication Details

> Yutaka Nagashima. Smart induction for Isabelle/HOL (tool paper). In *Proceedings of the 20th Conference on Formal Methods in Computer-Aided Design – FMCAD 2020*, 2020

In this chapter, I addressed `Task B` for relatively difficult inductive theorem proving in Isabelle/HOL, using `LiFtEr`. More specifically, I encoded 19 induction heuristics

in LiFtEr and implemented a recommendation tool, smart_induct. Given a problem in any problem domain. smart_induct suggests promising combinations of arguments to the induct tactic without completing a proof search. My in-depth evaluations demonstrated that smart_induct produces valuable recommendations across problem domains. Currently, smart_induct is an interactive tool; however, I expect that smart_induct can be used to narrow the search space of automatic inductive provers.

### My Contributions

I was the sole author of this chapter: I came up with the idea of developing a recommendation tool for the induct tactic, using LiFtEr in Isabelle/HOL, implemented and evaluated the tool, and wrote the paper independently.

## 2.4 PaMpeR: proof method recommendation system for Isabelle/HOL

### Publication Details

> Yutaka Nagashima and Yilun He. PaMpeR: proof method recommendation system for Isabelle/HOL. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 362–372, 2018

In this chapter, I built a proof method recommendation system, PaMpeR, to automate Task A in Isabelle/HOL. Given a proof state, PaMpeR recommends proof methods to discharge the proof goal and provides qualitative explanations as to why it suggests these methods. PaMpeR generates these recommendations based on existing hand-written proof corpora, thus transferring experienced users' expertise to new users. Our evaluation shows that PaMpeR correctly predicts experienced users' proof method invocations especially when it comes to special purpose proof methods.

### My Contributions

I designed PaMpeR and supervised the student, Yilun He, who implemented the first working prototype of PaMpeR. His first prototype had less than 20 assertions to extract features of proof obligations and relied on an external Python library for the machine learning implementation. And the first prototype showed a limited performance in an initial evaluation. Therefore, I added more than 90 assertions to make the feature extractor more expressive and implemented the machine learning algorithms in Isabelle/ML to remove the dependency to the external software. Even though Yilun He was the main developer of the first prototype, I became the main developer of the second prototype. I wrote the manuscript alone, which I presented at the conference.

## 2.5 Simple Dataset for Proof Method Recommendation in Isabelle/HOL (Dataset Description)

### Publication Details

> Yutaka Nagashima. Simple dataset for proof method recommendation in Isabelle/HOL. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings*, volume 12236 of *Lecture Notes in Computer Science*, pages 297–302. Springer, 2020

In this chapter, I present the dataset I used to build `PaMpeR`. This dataset contains data on over 400k proof method applications along with over 100 extracted features for each. The data format is simple, so that the dataset helps machine learning practitioners try out machine learning tools to address `Task A` without knowing domain expertise in logic.

### My Contributions

I produced the database using `PaMpeR`'s feature extractor from the Archive of Formal Proofs [KNPT04] and Isabelle's standard library. I also wrote the manuscript. I am the sole author of this chapter.

## 2.6 Goal-oriented conjecturing for Isabelle/HOL

### Publication Details

> Yutaka Nagashima and Julian Parsert. Goal-oriented conjecturing for Isabelle/HOL. In *Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings*, pages 225–231, 2018

In this chapter, I present an abductive reasoning framework to address `Task C`: discovering of useful auxiliary lemmas. Given a proof goal and its background context, my tool, `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 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.

### My Contributions

I designed the abductive reasoning framework and the framework to produce conjectures based on a proof obligation. I also implemented these. I wrote most of the manuscript except for Section 2.2 on conjecturing, which was written mainly by Julian Parsert.

## 2.7 Other Contributions Before my PhD Studies

In addition to the aforementioned contributions, I worked to develop a certifying compiler, Cogent, at Data61, CSIRO, formerly known as NICTA. This line of work resulted in the following three publications, which are not part of this dissertation.

### Publication Details (Functional Correctness of File Systems Code)

> Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby C. Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In Tom Conte and Yuanyuan Zhou, editors, *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 175–188. ACM, 2016

### My Contributions

Using Isabelle/HOL, I supported proving functional correctness of two main API functions of this file systems code written in the Cogent language. The main contributor to this proof was Sidney Amani, my contribution was limited to proving auxiliary lemmas Amani needed to complete the correctness proofs.

### Publication Details (Development of a Certifying Compiler for Cogent)

- Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby C. Murray, Gabriele Keller, and Gerwin Klein. A framework for the automatic formal verification of refinement from Cogent to C. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 323–340. Springer, 2016

- Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby C. Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: bringing down the cost of verification. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 89–102. ACM, 2016

### My Contributions

In this project, we developed a verified compiler for a restricted functional systems language with linear types (Cogent). I contributed to proof automation tools, written in ML, for the translation validation pass to C code in this verified compiler.

## 2.8 Other Contributions During my PhD Studies

In addition to the aforementioned contributions, I presented my plans to further improve Isabelle/HOL. These presentations resulted in the following two extended abstracts, which are not part of this dissertation.

### Publication Details

> Yutaka Nagashima. Towards evolutionary theorem proving for Isabelle/HOL. In Manuel López-Ibáñez, Anne Auger, and Thomas Stützle, editors, *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, pages 419–420. ACM, 2019

### My Contributions

I was the sole author of this extended abstract: I came up with the idea and presented it independently.

### Publication Details

> Yutaka NAGASHIMA. Towards united reasoning for automatic induction in Isabelle/HOL. *The 34th Annual Conference of the Japanese Society for Artificial Intelligence*, JSAI2020:3G1ES103–3G1ES103, 2020

### My Contributions

I was the sole author of this extended abstract: I came up with the idea and presented it independently.

# Chapter 3

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

## Publication Details

Yutaka Nagashima and Ramana Kumar. A proof strategy language and proof script generation for Isabelle/HOL. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 528–545. Springer, 2017

## 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.

## 3.1 Introduction

Currently, users of interactive theorem provers (ITPs) spend a lot of time iteratively interacting with their ITP to manually specialise and combine tactics. This time consuming process requires expertise in the ITP, making ITPs more esoteric than they should be. The integration of powerful automatic 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 [NPW02]. We provide `PSL` (available on GitHub [Nag16d]) as a language, so that its users can encode *proof strategies*, abstract descriptions of how

to attack proof obligations, based on their intuitions about a conjecture. When applied to a proof obligation, PSL's runtime system creates and combines several tactics based on the given proof strategy. This makes it possible to explore a larger search space than has previously been possible with conventional tactic languages, while utilising users' intuitions on the conjecture.

We developed PSL to use engineers' downtime: with PSL, we can run an automatic proof search for hours while we are attending meetings, sleeping, or reviewing papers. PSL makes such expensive proof search possible on machines with limited memory: PSL's runtime system truncates failed proof attempts as soon as it backtracks to minimise its memory usage.

Furthermore, PSL's runtime system attempts to generate efficient proof scripts from a given strategy by searching for the appropriate specialisation and combination of tactics for a particular conjecture without direct user interaction. Thus, PSL not only reduces the initial labour cost of theorem proving, but also keeps proof scripts interactive and maintainable by reducing the execution time of subsequent proof checking.

In Isabelle, Sledgehammer adopts a similar approach [BKPU16]. It exports a proof goal to various external ATPs and waits for them to find a proof. If the external provers find a proof, Sledgehammer tries to reconstruct an efficient proof script in Isabelle using hints from the ATPs. Sledgehammer is often more capable than most tactics but suffers from discrepancies between the polymorphic higher-order logic of Isabelle and the monomorphic first-order logic of most backend provers. While we integrated Sledgehammer as a sub-tool in PSL, PSL conducts a search using Isabelle tactics, thus avoiding the problems arising from the discrepancies and proof reconstruction.

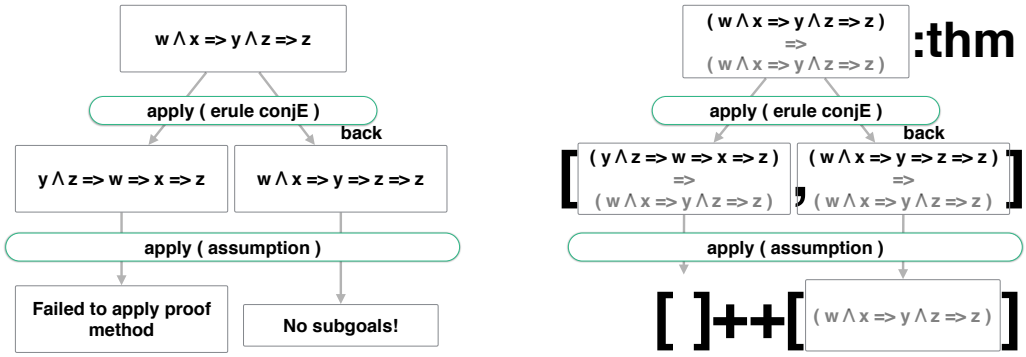
The underlying implementation idea in PSL is the monadic interpretation of proof strategies, which we introduce in Section 3.6. We expect this prover-agnostic formalization brings the following strengths of PSL to other ITPs such as Lean [dMKA+15] and Coq [TCdt]:

- runtime tactic generation based on user-defined procedures,
- memory-efficient large-scale proof search, and
- generation of efficient proof scripts for proof maintenance.

## 3.2 Background

Interactive theorem proving can be seen as the exploration of a search tree. Nodes of the tree represent proof states. Edges represent applications of tactics, which transform the proof state. Tactics are context sensitive: they behave differently depending on information stored in background proof contexts. These proof contexts contain such information as the constants defined and auxiliary lemmas proved prior to the current step. Since tactic behaviour depends on the proof context, it is hard to predict the shape of the search tree in advance.

The goal is to find a node representing a solved state: one in which the proof is complete. The search tree may be infinitely wide and deep, because there are endless variations of tactics that may be tried at any point. The goal for a PSL strategy is to




(a) External view (b) Internal view

Figure 3.1: External and internal view of proof search tree.

direct an automated search of this tree to find a solved state; `PSL` will reconstruct an efficient path to this state as a human-readable proof script.

Fig. 3.1a shows an example of proof search. At the top, the tactic `erule conjE` is applied to the proof obligation `w∧x` ⇒ `y∧z` ⇒ `z`. This tactic invocation produces two results, as there are two places to apply conjunction elimination. Applying conjunction elimination to `w∧x` returns the first result, while doing so to `y∧z` produces the second result. Subsequent application of proof by `assumption` can discharge the second result; however, `assumption` does not discharge the first one since the `z` in the assumptions is still hidden by the conjunction. Isabelle's proof language, *Isar*, returns the first result by default, but users can access the subsequent results using the keyword `back`.

Isabelle represents this non-deterministic behaviour of tactics using lazy sequences: tactics are functions of type `thm -> [thm]`, where `[·]` denotes a (possibly infinite) lazy sequence [Pau93]. Fig. 3.1 illustrates how Isabelle internally handles the above example where `++` stands for the concatenation of lazy sequences. Each proof state is expressed as a (possibly nested) implication which assumes proof obligations to conclude the conjecture. One may complete a proof by removing these assumptions using tactics. Tactic failure is represented as an empty sequence, which enables backtracking search by combining multiple tactics in a row [Wad85]. For example, one can write `apply(erule conjE, assumption)` using the sequential combinator `,` (comma) in *Isar*; this tactic traverses the tree using backtracking and discharges the proof obligation without relying on the keyword `back`.

The search tree grows wider when choosing between multiple tactics, and it grows deeper when tactics are combined sequentially. In the implementation language level, the tactic combinators in Isabelle include `THEN` for sequential composition (corresponding to `,` in *Isar*), `APPEND` for non-deterministic choice, `ORELSE` for deterministic choice, and `REPEAT` for iteration.

Isabelle/HOL comes with several default tactics such as `auto`, `simp`, `induct`, `rule`, and `erule`. When using tactics, proof authors often have to adjust tactics using *modifier*s for each proof obligation. `succeed` and `fail` are special tactics: `succeed` takes a value

of type `thm`, wraps it in a lazy sequence, and returns it without modifying the value. `fail` always returns an empty sequence.

## 3.3 Syntax of PSL

The following is the syntax of PSL. We made PSL's syntax similar to that of Isabelle's tactic language aiming at the better usability for users who are familiar with Isabelle's tactic language.

```
strategy := default | dynamic | special | subtool | compound
default  := Simp | Clarsimp | Fastforce | Auto | Induct
          | Rule | Erule | Cases | Coinduction | Blast
dynamic  := Dynamic (default)
special  := IsSolved | Defer | IntroClasses | Transfer
          | Normalization | Skip | Fail | User <string>
subtool  := Hammer | Nitpick | Quickcheck
compound := Thens [strategy] | Ors [strategy] | Alts [strategy]
          | Repeat (strategy) | RepeatN (strategy)
          | POrs [strategy] | PAlts [strategy]
          | PThenOne [strategy] | PThenAll [strategy]
          | Cut int (strategy)
```

The *default* strategies correspond to Isabelle's default tactics without arguments, while *dynamic* strategies correspond to Isabelle's default tactics that are specialised for each conjecture. Given a *dynamic* strategy and conjecture, the runtime system generates variants of the corresponding Isabelle tactic. Each of these variants is specialised for the conjecture with a different combination of promising arguments found in the conjecture and its proof context. It is the purpose of the PSL runtime system to select the right combination automatically.

*subtool* represents Isabelle tools such as Sledgehammer [PB10] and counterexample finders. The *compound* strategies capture the notion of tactic combinators: `Thens` corresponds to `THEN`, `Ors` to `ORELSE`, `Alts` to `APPEND`, and `Repeat` to `REPEAT`. `POrs` and `PAlts` are similar to `Ors` and `Alts`, respectively, but they admit parallel execution of sub-strategies. `PThenOne` and `PThenAll` take exactly two sub-strategies, combine them sequentially and apply the second sub-strategy to the results of the first sub-strategy in parallel in case the first sub-strategy returns multiple results. Contrary to `PThenAll`, `PThenOne` stops its execution as soon as it produces one result from the second sub-strategy. Users can integrate user-defined tactics, including those written in Eisbach [MWM14, MMW16], into PSL strategies using `User`. `Cut` limits the degree of non-determinism within a strategy.

In the following, we explain how to write strategies and how PSL's runtime system interprets strategies with examples.

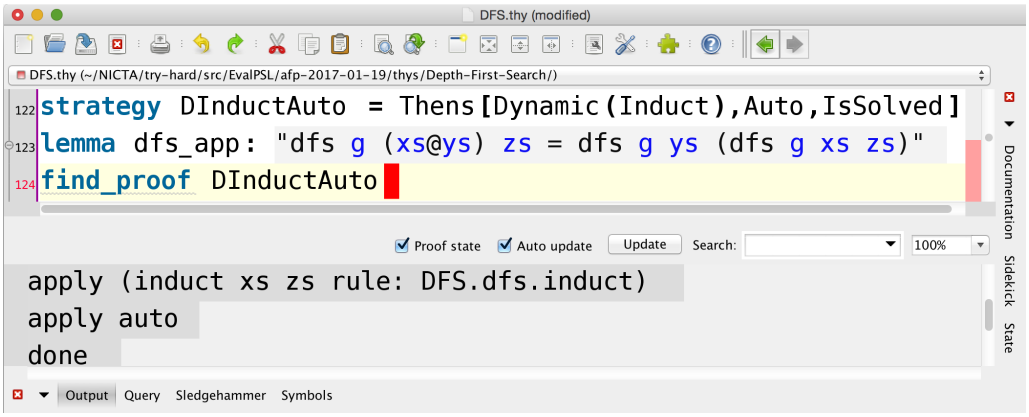



Figure 3.2: Screenshot for Example 1.

## 3.4 PSL by Example

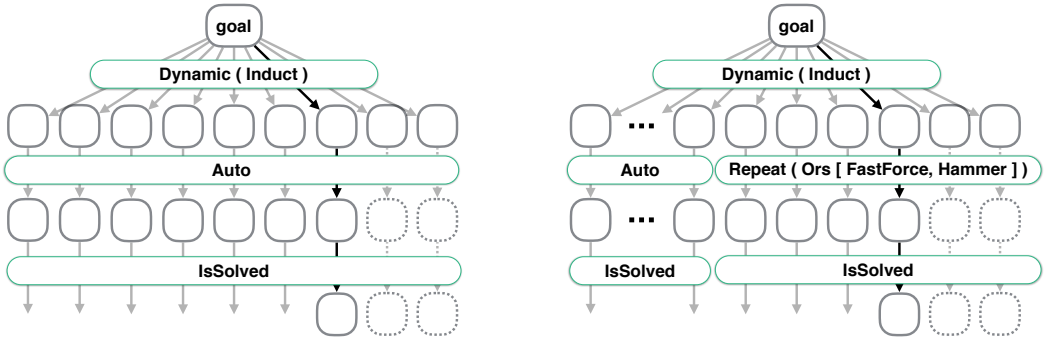
**Example 1.** For our first example, we take the following lemma from an entry [NM04a] in the Archive of Formal Proofs (AFP):

```
lemma dfs_app:  "dfs g (xs @ ys) zs = dfs g ys (dfs g xs zs)"
```

where `dfs` is a recursively defined function for depth-first search. As `dfs` is defined recursively, it is natural to expect that its proof involves some sort of mathematical induction. However, we do not know exactly how we should conduct mathematical induction here; therefore, we describe this rough idea as a proof strategy, `DInductAuto`, with the keyword `strategy`, and apply it to `dfs_app` with the keyword `find_proof` as depicted in Fig. 3.2. The `find_proof` command tells PSL's runtime system to interpret `DInductAuto`. For example, it interprets `Auto` as Isabelle's default tactic, `auto`.

The interpretation of `Dynamic(Induct)` is more involved: the runtime generates tactics using the information in `dfs_app` and its background context. First, PSL collects the free variables (noted in *italics* above) in `dfs_app` and applicable induction rules stored in the context. PSL uses the set of free variables to specify two things: on which variables instantiated tactics conduct mathematical induction, and which variables should be generalised in the induction scheme. The set of applicable rules are used to specify which rules to use. Second, PSL creates the powerset out of the set of all possible modifiers. Then, it attempts to instantiate a variant of the induct tactic for each subset of modifiers. Finally, it combines all the variants of `induct` with unique results using `APPEND`. In this case, PSL tries to generate 4160 induct tactics for `dfs_app` by passing several combinations of modifiers to Isabelle; however, Isabelle cannot produce valid induction schemes for some combinations, and some combinations lead to the same induction scheme. The runtime removes these, focusing on the 223 unique results. PSL's runtime combine these tactics with `auto` using `THEN`.

PSL's runtime interprets `IsSolved` as the `is_solved` tactic, which checks whether any




(a) `DInductAuto` (b) `some_induct2`

Figure 3.3: Proof search tree for `some_induct`

proof obligations are left or not. If obligations are left, `is_solved` behaves as `fail`, triggering backtracking. If not, `is_solved` behaves as `succeed`, allowing the runtime to stop the search. This is how `DInductAuto` uses `IsSolved` to ensure that no sub-goals are left before returning an efficient proof script. For `dfs_app`, `PSL` interprets `DInductAuto` as the following tactic:

```
(induct1 APPEND induct2 APPEND...)  THEN auto THEN is_solved
```

where `induct_n`s are variants of the induct tactic specialised with modifiers.

Within the runtime system, Isabelle first applies `induct1` to `dfs_app`, then `auto` to the resultant proof obligations. Note that each `induct` tactic and `auto` is deterministic: it either fails or returns a lazy sequence with a single element. However, combined together with `APPEND`, the numerous variations of `induct` tactics *en masse* are non-deterministic: if `is_solved` finds remaining proof obligations, Isabelle backtracks to the next `induct` tactic, `induct2` and repeats this process until either it discharges all proof obligations or runs out of the variations of `induct` tactics. The numerous variants of `induct` tactics from `DInductAuto` allow Isabelle to explore a larger search space than its naive alternative, `induct THEN auto`, does. Fig. 3.3a illustrates this search procedure. Each edge and curved square represents a tactic application and a proof state, respectively, and edges leading to no object stand for tactic failures. The dashed objects represent possible future search space, which `PSL` avoids traversing by using lazy sequences.

The larger search space specified by `DInductAuto` leads to a longer search time. `PSL` addresses this performance problem by tracing Isabelle's proof search: it keeps a log of successful proof attempts while removing backtracked proof attempts. The monadic interpretation discussed in Section 3.6 let `PSL` remove failed proof steps as soon as it backtracks. This minimises `PSL` memory usage, making it applicable to hours of expensive automatic proof search. Furthermore, since `PSL` follows Isabelle's execution model based on lazy sequences, it stops proof search as soon as it finds a specialisation and combination of tactics, with which Isabelle can pass the no-proof-obligation test

imposed by `is_solved`.

We still need a longer search time with PSL, but only once: upon success, PSL converts the log of successful attempts into an efficient proof script, which bypasses a large part of proof search. For `dfs_app`, PSL generates the following proof script from `DInductAuto`.

```
apply (induct xs zs rule: DFS.dfs.induct) apply auto done
```

We implemented PSL as an Isabelle theory; to use it, PSL users only have to import the relevant theory files to use PSL to their files. Moreover, we have integrated PSL into Isabelle/Isar, Isabelle's proof language, and Isabelle/jEdit, its standard editor. This allows users to define and invoke their own proof strategies inside their ongoing proof attempts, as shown in Figure 3.2; and if the proof search succeeds PSL presents a proof script in jEdit's output panel, which users can copy to the right location with one click. All generated proof scripts are independent of PSL, so users can maintain them without PSL.

**Example 2.** `DInductAuto` is able to pick up the right induction scheme for relatively simple proof obligations using backtracking search. However, in some cases even if PSL picks the right induction scheme, `auto` fails to discharge the emerging sub-goals. In the following, we define `InductHard`, a more powerful strategy based on mathematical induction, by combining `Dynamic(Induct)` with more involved sub-strategies to use external theorem provers.

```
strategy SolveAllG = Thens[Repeat(Ors[Fastforce,Hammer]),IsSolved]
strategy PInductHard = PThenOne[Dynamic(Induct),SolveAllG]
strategy InductHard = Ors[DInductAuto, PInductHard]
```

PSL's runtime system interprets `Fastforce` and `Hammer` as the `fastforce` tactic and Sledgehammer, respectively. Both `fastforce` and Sledgehammer try to discharge the first sub-goal only and return an empty sequence if they cannot discharge the sub-goal.

The repetitive application of Sledgehammer would be very time consuming. We mitigate this problem using `Ors` and `PThenOne`. Combined with `Ors`, PSL executes `PInductHard` only if `DInductAuto` fails. When `PInductHard` is called, it first applies `Dynamic(Induct)`, producing various induction schemes and multiple results. Then, `SolveAllG` tries to discharge these results in parallel. The runtime stops its execution when `SolveAllG` returns at least one result representing a solved state. We apply this strategy to the following conjecture, which states the two versions of depth-first search programs (`dfs2` and `dfs`) return the same results given the same inputs.

```
lemma "dfs2 g xs ys = dfs g xs ys"
```

Then, our machine with 28 cores returns the following script within 3 minutes:

```
apply (induct xs ys rule: DFS.dfs2.induct)
apply fastforce
```

```
apply (simp add: dfs_app)
done
```

Fig. 3.3b roughly shows how the runtime system found this proof script. The runtime first tried to find a complete proof as in Example 1, but without much success. Then, it interpreted `PInductHard`. While doing so, it found that induction on *xs* and *ys* using `DFS.dfs2.induct` leads to two sub-goals both of which can be discharged either by `fastforce` or Sledgehammer. For the second sub-goal, Sledgehammer found out that the result of *Example 1* can be used as an auxiliary lemma to prove this conjecture. Then, it returns an efficient proof script `(simp add: dfs_app)` to `PSL`, before `PSL` combines this with other parts and prints the complete proof script.

**Example 3.** In the previous examples, we used `IsSolved` to get a complete proof script from `PSL`. In Example 3, we show how to generate incomplete but useful proof scripts, using `Defer`. Incomplete proofs are specially useful when ITP users face numerous proof obligations, many of which are within the reach of high-level proof automation tools, such as Sledgehammer, but a few of which are not.

Without `PSL`, Isabelle users had to manually invoke Sledgehammer several times to find out which proof obligations Sledgehammer can discharge. We developed a strategy, `HamCheck`, to automate this time-consuming process. The following shows its definition and a use case simplified for illustrative purposes.

```
strategy HamCheck = RepeatN(Ors[Hammer,Thens[Quickcheck,Defer]])
lemma safe_trans:  shows
1:"ps_safe p s" and 2:"valid_tran p s s' c" and 3:"ps_safe p s'"
find_proof HamCheck
```

We made this example simple, so that two sub-goals, `1:"ps_safe p s"` and `3:"ps_safe p s'"`, are not hard to prove; however, they are still beyond the scope of commonly used tactics, such as `fastforce`.

Generally, for a conjecture and a strategy of the form of `RepeatN (strategy)`, `PSL` applies `strategy` to the conjecture as many times as the number of proof obligations in the conjecture. In this case, `PSL` applies `Ors [Hammer, Thens [Quickcheck, Defer]]` to `safe_trans` three times.

Note that we integrated `quickcheck` and `nitpick` into `PSL` as *assertion tactics*. Assertion tactics provide mechanisms for controlling proof search based on a condition: such a tactic takes a proof state, tests an assertion on it, then behaves as `succeed` or `fail` accordingly. We have already seen one of them in the previous examples: `is_solved`.

`Ors [Hammer, Thens [Quickcheck, Defer]]` first applies Sledgehammer. If Sledgehammer does not find a proof, it tries to find counterexamples for the sub-goal using `quickcheck`. If `quickcheck` finds no counterexamples, `PSL` interprets `Defer` as `defer_tac 1`, which postpones the current sub-goal to the end of the list of proof obligations.

In this example, Sledgehammer fails to discharge `2:"valid_tran p s s' c"`. When

Sledgehammer fails, PSL passes 2 to Thens [Quickcheck, Defer], which finds no counterexample to 2 and sends 2 to the end of the list; then, PSL continues working on the sub-goal 3 with Sledgehammer. The runtime stops its execution after applying Ors [Hammers, Thens [Quickcheck, Defer]] three times, generating the following proof script. This script discharges 1 and 3, but it leaves 2 as the meaningful task for human engineers, while assuring there is no obvious counterexamples for 2.

```
apply (simp add: state_safety ps_safe_def)
defer apply (simp add: state_safety ps_safe_def)
```

## 3.5 The default strategy: try_hard.

PSL comes with a default strategy, try_hard. Users can apply try_hard as a completely automatic tool: engineers need not provide their intuitions by writing strategies. Unlike other user-defined strategies, one can invoke this strategy by simply typing try_hard without find_proof inside a proof attempt. The lack of input from human engineers makes try_hard less specific to each conjecture; however, we made try_hard more powerful than existing proof automation tools for Isabelle by specifying larger search spaces presented.

We conducted a Judgement Day style evaluation [BN10b] of try_hard against selected theory files from the AFP, coursework assignments and exercises [BFW15], and Isabelle's standard library. Table 1, 2 and 3 show that given 300 seconds for each proof goal try_hard solves 1115 proof goals out of 1526, while Sledgehammer found proofs for 901 of them using the same computational resources and re-constructed proofs in Isabelle for 866 of them. This is a 14 percentage point improvement of proof search and a 16 percentage point increase for proof reconstruction. Moreover, 299 goals (20% of all goals) were solved only by try_hard within 300 seconds. They also show that a longer time-out improves the success ratio of try_hard, which is desirable for utilising engineers' downtime.

try_hard is particularly more powerful than Sledgehammer at discharging proof obligations that can be nicely handled by the following:

- mathematical induction or co-induction,
- type class mechanism,
- specific procedures implemented as specialised tactics (such as transfer and normalization), or
- general simplification rules (such as field_simps and algebra_simps) .

Furthermore, careful observation of PSL indicates that PSL can handle the so-called "hidden fact" problem in relevance filtering. Hidden facts are auxiliary lemmas that are useful to discharge a given proof obligation but are not obviously relevant. For example, a hidden fact may share no constants with the proof obligation, because it is related only via an intermediate fact. With PSL, a user can write a strategy that applies rewriting before relevance filtering to reveal more information. This information allows the relevance filter to find useful facts that were previously hidden. For example, the following strategy "massages" the given proof obligation before invoking the relevance filter of Sledgehammer: Thens [Auto, Repeat(Hammer), IsSolved].

Table 3.1: The number of automatically proved proof obligations from assignments. `TH` and `SH` stand for the number of obligations discharged by `try_hard` and Sledgehammer, respectively. `TH\SH` represents the number of goals to which `try_hard` found proofs but Sledgehammer did not. `POs` stands for the number of proof obligations in the theory file. $x(y)$ for `SH` means Sledgehammer found proofs for $x$ proof obligations, out of which it managed to reconstruct proof scripts in Isabelle for $y$ goals. We omit these parentheses when these numbers coincide. Note that all proofs of `PSL` are checked by Isabelle/HOL. Besides, Sledgehammer inside `PSL` avoids the `smt` proof method, as this method is not allowed in the Archive of Formal Proofs.

| assignments [BFW15] | POs | TH | SH | TH\SH | TH | SH | TH\SH |
|---|---|---|---|---|---|---|---|
| time out | - | 30s | 30s | 30s | 300s | 300s | 300s |
| assignment_1 | 19 | 17 | 14(13) | 4 | 18 | 14(13) | 5 |
| assignment_2 | 22 | 21 | 5 | 16 | 22 | 5 | 17 |
| assignment_3 | 52 | 30 | 27 | 8 | 35 | 27 | 10 |
| assignment_4 | 82 | 66 | 61 | 10 | 71 | 61 | 10 |
| assignment_5 | 64 | 36 | 41(39) | 6 | 55 | 44(42) | 17 |
| assignment_6 | 26 | 11 | 12(11) | 2 | 14 | 13(12) | 3 |
| assignment_8 | 52 | 36 | 45(39) | 1 | 40 | 46(39) | 0 |
| assignment_9 | 61 | 31 | 32(30) | 6 | 35 | 32(30) | 6 |
| assignment_11 | 26 | 14 | 15 | 1 | 20 | 17 | 3 |
| sum | 404 | 262 | 252(241) | 54 | 310 | 259(246) | 71 |

For 3 theories out of 35, `try_hard` discharged fewer proof obligations, even given 300 seconds of time-out. This is due to the fact that `PSL` uses a slightly restricted version of Sledgehammer internally for the sake of the integration with other tools and to avoid the `smt` method, which is not allowed in the AFP. In these files, Sledgehammer can discharge many obligations and other obligations are not particularly suitable for other sub-tools in `try_hard`. Of course, given high-performance machines, users can run both `try_hard` and Sledgehammer in parallel to maximise the chance of proving conjectures.

## 3.6 Monadic Interpretation of Strategy

The implementation of the tracing mechanism described in Section 3.4 is non-trivial: `PSL`'s tracing mechanism has to support arbitrary strategies conforming to its syntax. What is worse, the runtime behaviour of backtracking search is not completely predictable statically since `PSL` generates tactics at runtime, using information that is not available statically. Moreover, the behaviour of each tactic varies depending on the proof context and proof obligation at hand.

Table 3.2: The number of automatically proved proof obligations from exercises.

| exercises [BFW15] | POs | TH | SH | TH\SH | TH | SH | TH\SH |
|---|---|---|---|---|---|---|---|
| time out | - | 30s | 30s | 30s | 300s | 300s | 300s |
| exercise_1 | 15 | 12 | 8 | 4 | 12 | 8 | 4 |
| exercise_2 | 7 | 4 | 3 | 2 | 5 | 3 | 2 |
| exercise_3 | 42 | 27 | 26(25) | 5 | 29 | 27(26) | 5 |
| exercise_4 | 23 | 11 | 15 | 0 | 17 | 15 | 2 |
| exercise_5a | 13 | 9 | 11 | 0 | 11 | 11 | 0 |
| exercise_5b | 83 | 65 | 74 | 1 | 74 | 74 | 1 |
| exercise_6 | 4 | 1 | 2 | 0 | 1 | 3 | 0 |
| exercise_7a | 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| exercise_7b | 9 | 5 | 6 | 1 | 8 | 6 | 2 |
| exercise_8a | 10 | 7 | 7 | 1 | 7 | 7 | 1 |
| exercise_8b | 26 | 11 | 9 | 4 | 12 | 12 | 2 |
| exercise_9 | 31 | 14 | 17 | 3 | 19 | 17 | 3 |
| exercise_10 | 15 | 5 | 5(4) | 1 | 6 | 6(5) | 1 |
| exercise_11 | 10 | 4 | 6 | 0 | 9 | 6 | 3 |
| exercise_12 | 30 | 8 | 10 | 1 | 12 | 10 | 3 |
| sum | 321 | 183 | 199(197) | 23 | 222 | 205(203) | 29 |

Table 3.3: The number of automatically proved proof goals from AFP entries and Isabelle's standard libraries. Due to limited page space, we introduced the following short names for some theory files: `ES` for `Efficient_Sort.thy`, `CL` for `Coinductive_Language.thy`, `CFG` for `Context_Free_Grammar.thy`, and `HLTM` for `HOL/Library/Tree_Multiset.thy`.

| theory name | POs | TH | SH | TH\SH | TH | SH | TH\SH |
|---|---|---|---|---|---|---|---|
| time out | - | 30s | 30s | 30s | 300s | 300s | 300s |
| `DFS.thy` [NM04a] | 51 | 24 | 28 | 6 | 34 | 29 | 7 |
| `ES` [Ste11] | 75 | 27 | 28(26) | 8 | 33 | 31(28) | 9 |
| `List_Index.thy` [Nip10] | 105 | 48 | 72(70) | 12 | 67 | 75(72) | 14 |
| `Skew_Heap.thy` [Nip14] | 16 | 8 | 6(5) | 4 | 12 | 8(7) | 5 |
| `Hash_Code.thy` [Lam09] | 16 | 7 | 4 | 4 | 11 | 4 | 7 |
| `CoCallGraph.thy` [Bre15] | 141 | 88 | 78(71) | 29 | 104 | 79(73) | 33 |
| `CL` [Tra13] | 139 | 57 | 69(68) | 11 | 106 | 70(69) | 43 |
| `CFG` [Tra13] | 29 | 26 | 2 | 26 | 29 | 2 | 27 |
| `LTL.thy` [Sic16] | 97 | 56 | 61 | 15 | 78 | 65(62) | 15 |
| `HOL/Library/Tree.thy` | 124 | 93 | 70(68) | 32 | 101 | 73(70) | 32 |
| `HLTM` | 8 | 8 | 1 | 7 | 8 | 1 | 7 |
| sum | 801 | 442 | 419(404) | 154 | 583 | 437(417) | 199 |

**Program 1** Monad with zero and plus, and lazy sequence as its instance.

```
signature MONADOPLUS =
sig
  type 'a m0p;
  val return :  'a -> 'a m0p;
  val bind   :  'a m0p -> ('a -> 'b m0p) -> 'b m0p;
  val mzero  :  'a m0p;
  val ++     : ('a m0p * 'a  m0p) -> 'a m0p;
end;
structure Nondet : MONADOPLUS =
struct
  type 'a m0p     = 'a Seq.seq;
  val return      = Seq.single;
  fun bind xs f   = Seq.flat (Seq.map f xs);
  val mzero       = Seq.empty;
  fun (xs ++ ys)  = Seq.append xs ys;
end;
```

It is likely to cause code clutter to specify where to backtrack explicitly with references or pointers, whereas explicit construction of search tree [Nag16c] consumes too much memory space when traversing a large search space. Furthermore, both of these approaches deviate from the standard execution model of Isabelle explained in Section 3.2. This deviation makes the proof search and the efficient proof script generation less reliable. In this section, we introduce our monadic interpreter for `PSL`, which yields a modular design and concise implementation of `PSL`'s runtime system.

**Monads in Standard ML.** A monad with zero and plus is a constructor class [1] with four basic methods (`return`, `bind`, `mzero`, and `++`). As Isabelle's implementation language, Standard ML, does not natively support constructor classes, we emulated them using its module system [NO16]. Program 1 shows how we represent the type constructor, `seq`, as an instance of monad with zero and plus.

The body of `bind` for lazy sequences says that it applies `f` to all the elements of `xs` and concatenates all the results into one sequence. Attentive readers might notice that this is equivalent to the behaviour of `THEN` depicted in Fig. 3.1 and that of `Thens` shown in Fig. 3.3. In fact, we can define all of `THEN`, `succeed`, `fail`, and `APPEND`, using `bind`, `return`, `mzero`, and `++`, respectively.

[1] Constructor classes are a class mechanism on type constructors such as `list` and `option`, whereas type classes are a class mechanism on types such as `int` and `double`. Commonly used constructor classes include functor, applicative, monoid, and arrow.

**Program 2** The monadic interpretation of strategies.

```
interp :: core_strategy -> 'a -> 'a m0p
interp (Atom atom_str) n     = eval atom_str n
interp  Skip  n              = return n
interp  Fail  n              = mzero
interp (str1 Then str2) n    = bind (interp str1 n) (interp str2)
interp (str1 Alt  str2) n    = interp str1 n ++ interp str2 n
interp (str1 Or   str2) n    = let val result1 = interp str1 n
  in if (result1 != mzero) then result1 else interp str2 n end
interp (Rep str) n           = interp ((str THEN (Rep str)) Or Skip) n
interp (Comb (comb, strs)) n = eval_comb (comb, map interp strs) n
```

**Monadic Interpretation of Strategies.** Based on this observation, we formalised PSL's search procedure as a monadic interpretation of strategies, as shown in Program 2, where the type `core_strategy` stands for the internal representation of strategies. Note that `Alt` and `Or` are binary versions of `Alts` and `Ors`, respectively; `PSL` desugars `Alts` and `Ors` into nested `Alt`s and `Or`s. We could have defined `Or` as a syntactic sugar using `Alt`, `mzero`, `Fail`, and `Skip`, as explained by Martin *et al.* [MGW96]; however, we prefer the less monadic formalisation in Program 2 for better time complexity.

`eval` deals with all the atomic strategies, which correspond to *`default`*, *`dynamic`*, and *`special`* in the surface language. For the *`dynamic`* strategies, `eval` expands them into dynamically generated tactics making use of contextual information from the current proof state. `PSL` combines these generated tactics either with `APPEND` or `ORELSE`, depending on the nature of each tactic. `eval_comb` handles non-monadic strategy combinators, such as `Cut`. We defined the body of `eval` and `eval_comb` for each atomic strategy and strategy combinator separately using pattern matching. As is obvious in Program 2, `interp` separates the complexity of compound strategies from that of runtime tactic generation.

**Adding Tracing Modularly for Proof Script Generation.** We defined `interp` at the constructor class level, abstracting it from the concrete type of proof state and even from the concrete type constructor. When instantiated with lazy sequence, `interp` tries to return the first element of the sequence, working as depth-first search. This abstraction provides a clear view of how compound strategies guide proof search while producing tactics at runtime; however, without tracing proof attempts, `PSL` has to traverse large search spaces every time it checks proofs.

We added the tracing mechanism to `interp`, combining the non-deterministic monad, `Nondet`, with the writer monad. To combine multiple monads, we emulate monad transformers using ML functors: Program 3 shows our ML functor, `writer_trans`, which takes a module of `MONADOPLUS`, adds the logging mechanism to it, and returns a module equipped with both the capability of the base monad and the logging mechanism of the writer monad. We pass `Nondet` to `writer_trans` as the base monad to combine

**Program 3** The writer monad transformer as a ML functor.

```
functor writer_trans (structure Log:MONOID; structure Base:MONADOPLUS) =
struct
  type 'a m0p  = (Log.monoid * 'a) Base.m0p;
  fun return (m:'a) = Base.return (Log.mempty, m) : 'a m0p;
  fun bind (m:'a m0p) (func: 'a -> 'b m0p) : 'b m0p =
    Base.bind    m          (fn (log1, res1) =>
    Base.bind   (func res1) (fn (log2, res2) =>
    Base.return (Log.mappend log1 log2, res2)));
  val mzero = Base.mzero;
  val (xs ++ ys) = Base.++ (xs, ys);
end : MONADOPLUS;
```

the logging mechanism and the backtracking search based on non-deterministic choice. Observe Programs 1, 2 and 3 to see how `Alt` and `Or` truncate failed proof attempts while searching for a proof. The returned module is based on a new type constructor, but it is still a member of `MONADOPLUS`; therefore, we can re-use `interp` without changing it.

**History-Sensitive Tactics using the State Monad Transformer.** The flexible runtime interpretation might lead `PSL` into a non-terminating loop, such as `REPEAT succeed`. To handle such loops, `PSL` traverses a search space using iterative deepening depth-first search (IDDFS). However, passing around information about depth as an argument of `interp` as following quickly impairs its simplicity:

```
interp (t1 CSeq t2) level n = if level < 1 then return n else ...
interp (t1 COr  t2) level n = ...
```

where `level` stands for the remaining depth `interp` can proceed for the current iteration.

We implemented IDDFS without code clutter, introducing the idea of a *history-sensitive tactic*: a tactic that takes the log of proof attempts into account. Since the writer monad does not allow us to access the log during the search time, we replaced the writer monad transformer with the state monad transformer, with which the runtime keeps the log of proof attempt as the "state" of proof search and access it during search. By measuring the length of "state", `interp` computes the current depth of proof search at runtime.

The modular design and abstraction discussed above made this replacement possible with little change to the definition of `interp`: we only need to change the clause for `Atom`, providing a wrapper function, `iddfc`, for `eval`, while other clauses remain intact.

```
inter (CAtom atom_str) n = iddfc limit eval atom_str n
```

`iddfc limit` first reads the length of "state", which represents the number of edges to the node from the top of the implicit proof search tree. Then, it behaves as `fail` if the length exceeds `limit`; if not, it executes `eval atom_str n`. [2]

[2] In this sense, we implemented IDDFS as a tactic combinator.

## 3.7 Related Work

ACL2 [KMM00] is a functional programming language and mostly automated first-order theorem prover. ACL2 is known for the so-called waterfall model, which is essentially repeated application of various heuristics. Its users can guide proof search by supplying arguments called "hints", but the underlining operational procedure of the waterfall model itself is fixed. ACL2 does not produce efficient proof scripts after running the waterfall algorithm.

PVS [ORS92] provides a collection of commands called "strategies". Despite the similarity of the name to `PSL`, strategies in PVS correspond to tactics in Isabelle. The highest-level strategy in PVS, grind, can produce re-runnable proof scripts containing successful proof steps only. However, scripts returned by grind describe steps of much lower level than human engineers would write manually, while `PSL`'s returned scripts are based on tactics engineers use. Furthermore, grind is known to be useful to complete a proof that does *not* require induction, while `try_hard` is good at finding proofs involving mathematical induction.

SEPIA [GWR15] is an automated proof generation tool in Coq. Taking existing Coq theories as input, SEPIA first produces proof traces, from which it infers an extended finite state machine. Given a conjecture, SEPIA uses this model to search for its proof. The authors of SEPIA chose to use breadth-first search (BFS) to find shorter proofs. For `PSL` we could emulate the BFS strategy within the IDDFS framework. However, our experience tells us that the search tree tends to be very wide and some tactics, such as `induct`, need to be followed by other tactics to complete proofs. Therefore, we chose IDDFS for `PSL`. Both SEPIA and `PSL` off-load the construction of proof scripts to search and try to reconstruct efficient proof scripts. Compared to SEPIA, `PSL` allows users to specify their own search strategies to utilize the engineer's intuition, which enables `PSL` to return incomplete proof scripts, as discussed in Section 3.4.

Martin *et al.* first discussed a monadic interpretation of tactics for their language, *Angel*, in an unpublished draft [MG02]. We independently developed `interp` with the features discussed above, lifting the framework from the tactic level to the strategy level to traverse larger search spaces. The two interpreters for different ITPs turned out to be similar to each other, suggesting our approach is not specific to Isabelle but can be used for other ITPs.

Similar to Ltac [Del00] in Coq, Eisbach [MWM14, MMW16] is a framework to write *proof method*s in Isabelle. Proof methods are the *Isar* syntactic layer of tactics. Eisbach does not generate methods dynamically, trace proof attempts, nor support parallelism natively. Eisbach is good when engineers already know how to prove their conjecture, while `try_hard` is good when they want to find out how to prove it.

IsaPlanner [DF03] offers a framework for encoding and applying common patterns of reasoning in Isabelle, following the style of proof planning [Bun88]. IsaPlanner addresses the performance issue by a memoization technique, on the other hand `try_hard` strips off backtracked steps while searching for a proof, which Isabelle can check later without `try_hard`. While IsaPlanner works on its own data structure *reasoning state*, `try_hard` managed to minimize the deviation from Isabelle's standard execution model using

constructor classes.

## 3.8 Conclusions

PSL improves proof automation in higher-order logic, allowing us to exploit both the engineer's intuition and various automatic tools. The simplicity of the design is our intentional choice: we reduced the process of interactive proof development to the well-known dynamic tree search problem and added new features (efficient-proof script generation and IDDFS) by safely abstracting the original execution model and employing commonly used techniques (monad transformers).

We claim that our approach enjoys significant advantages. Despite the simplicity of the design, our evaluations indicate that PSL reduces the labour cost of ITP significantly. The conservative extension to the original model lowers the learning barrier of PSL and makes our proof script generation reliable by minimising the deviation. The meta-tool approach makes the generated proof script independent of PSL, separating the maintenance of proof scripts from that of PSL; furthermore, by providing a common framework for various tools we supplement one tool's weakness (e.g. induction for Sledgehammer) with other tools' strength (e.g. the `induct` tactic), while enhancing their capabilities with runtime tactic generation. The parallel combinators reduce the labour-intensive process of interactive theorem proving to embarrassingly parallel problems. The abstraction to the constructor class and reduction to the tree search problem make our ideas transferable: other ITPs, such as Lean and Coq, handle inter-tactic backtracking, which is best represented in terms of MONADOPLUS.

#### Acknowledgements

We thank Jasmin C. Blanchette for his extensive comments that improved the evaluation of `try_hard`. Pang Luo helped us for the evaluation. Leonardo de Moura, Daniel Matichuk, Kai Engelhardt, and Gerwin Klein provided valuable comments on an early draft of this paper. We thank the anonymous reviewers for useful feedback, both at CADE-26 and for previous versions of this paper at other conferences. This work was partially funded by the ERC Consolidator grant 649043 - AI4REASON.

## Appendix: Details of the Evaluation

All evaluations were conducted on a Linux machine with Intel (R) Core (TM) i7-600 @ 3.40GHz and 32 GB memory. For both tools, we set the time-out of proof search to 30 and 300 seconds for each proof obligation.

Prior to the evaluation, the relevance filter of Sledgehammer was trained on 27,041 facts and 18,695 non-trivial Isar proofs from the background libraries imported by theories under evaluation for both tools. Furthermore, we forbid Sledgehammer inside PSL from using the `smt` method for proof reconstruction, since the AFP does not permit this method.

Note that `try_hard` does not use parallel strategy combinators which exploit parallelism. The evaluation tool does not allow `try_hard` to use multiple threads either. Therefore, given the same time-out, `try_hard` and Sledgehammer enjoy the same amount of computational resources, assuring the fairness of the evaluation results.

The evaluation tool [Nag16b] and results [Nag16a] are available at our websites. We provide the evaluation tool and results in the following websites:

- `http://ts.data61.csiro.au/Downloads/cade26_evaluation/`
- `http://ts.data61.csiro.au/Downloads/cade26_results/`

# Chapter 4

# LiFtEr: Language to Encode Induction Heuristics for Isabelle/HOL

## Publication Details

Yutaka Nagashima. LiFtEr: Language to encode induction heuristics for Isabelle/HOL. In *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings*, pages 266–287, 2019

## 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, thus automating the knowledge transfer loop.

## 4.1 Introduction

Consider the following reverse functions, `rev` and `itrev`, presented in a tutorial of Isabelle/HOL [NPW02]:

```
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)"
```

where `#` is the list constructor, and `@` appends two lists. How do you prove the following lemma?

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

Since both `rev` and `itrev` are defined recursively, it is natural to imagine that we can handle this problem by applying induction. But how do you apply induction and why? What induction heuristics do you use? In which language do you describe those heuristics?

Modern proof assistants (PAs), such as Isabelle/HOL [NPW02], are forming the basis of trustworthy software. Klein *et al.*, for example, verified the correctness of the seL4 micro-kernel in Isabelle/HOL [KAE+10], whereas Leroy developed a certifying C compiler, CompCert, using Coq [Ler09]. Despite the growing number of such complete formal verification projects, the limited progress in proof automation still keeps the cost of proof development high, thus preventing the wide spread adoption of complete formal verification.

A noteworthy approach in proof automation for proof assistants is hammer tools [BKPU16]. Sledgehammer, for example, exports proof goals in Isabelle/HOL to various external automated theorem provers (ATPs) to exploit the state-of-the-art proof automation of these backend provers; however, the discrepancies between the polymorphic higher-order logic of Isabelle/HOL and the monomorphic first-order logic of the backend provers severely impair Sledgehammer's performance when it comes to inductive theorem proving (ITP).

This is unfortunate for two reasons. Firstly, many Isabelle users chose Isabelle/HOL precisely because its higher-order logic is expressive enough to specify mathematical objects and procedures involving recursion without introducing new axioms. Secondly, induction lies at the heart of mathematics and computer science. For instance, induction is often necessary for reasoning about natural numbers, recursive data-structures, such as lists and trees, computer programs containing recursion and iteration [Bun01].

This is why ITP remains as a long-standing challenge in computer science, and its automation is much needed. Facing the limited automation in ITP, Gramlich surveyed the problems in ITP and presented the following prediction in 2005 [Gra05]:

> in the near future, ITP will only be successful for very specialized domains for very restricted classes of conjectures. ITP will continue to be a very challenging engineering process.

We address this conundrum with our domain-specific language, `LiFtEr`. `LiFtEr` allows experienced Isabelle users to encode their induction heuristics in a style independent of problem domains. `LiFtEr`'s interpreter mechanically checks if a given application of induction is compatible with the induction heuristics written by experienced users. Our research hypothesis is that:

> *it is possible to encode valuable induction heuristics for Isabelle/HOL in* `LiFtEr` *and these heuristics can be valid across diverse problem domains, because* `LiFtEr` *allows for meta-reasoning on applications of induction methods,*

> *without relying on concrete proof goals, their underlying proof states, nor concrete applications of induction methods.*

We developed `LiFtEr` as an Isabelle theory and integrated `LiFtEr` into Isabelle's proof language, Isabelle/Isar, and its proof editor, Isabelle/jEdit. This allows for an easy installation process: to use `LiFtEr`, users only have to import the relevant theory files into their theory files, using Isabelle's `import` keyword. Our working prototype is available at GitHub [Nag].

The important difference of `LiFtEr` from other tactic languages, such as Eisbach [MWM14, MMW16] and Ltac [Del00], is that `LiFtEr` itself is not a tactic language but a language to write how one should use Isabelle's existing proof method for induction. To the best of our knowledge, `LiFtEr` is the first language in which one can write how to use a tactic by mechanically analyzing the structures of proof goals in a style independent of any problem domain.

## 4.2 Induction in Isabelle/HOL

To handle inductive problems, modern proof assistants offer tools to apply induction. For example, Isabelle comes with the `induct` proof method and the `induction` method [1]. Nipkow *et al.* proved our ongoing example as follows [NK14]:

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

Namely, they applied structural induction on `xs` while generalizing `ys` before applying induction by passing the string `ys` to the `arbitrary` field. The resulting sub-goals are:

```
1. !!ys. itrev [] ys = rev [] @ ys
2. !!a xs ys. (!!ys. itrev xs ys = rev xs @ ys) ==>
              itrev (a # xs) ys = rev (a # xs) @ ys
```

where `!!` is the universal quantifier and `==>` is the implication in Isabelle's meta-logic. Due to the generalization, the `ys` in the induction hypothesis is quantified within the hypothesis, and it is differentiated from the `ys` that appears in the conclusion. Had Nipkow *et al.* omitted `arbitrary: ys`, the first sub-goal would be the same, but the second sub-goal would have been:

```
 2. !!a xs. itrev xs ys = rev xs @ ys ==>
            itrev (a # xs) ys = rev (a # xs) @ ys
```

Since the same `ys` is shared by the induction hypothesis and the conclusion, the subsequent application of `auto` fails to discharge this sub-goal.

It is worth noting that in general there are multiple equivalently appropriate combinations of arguments to prove a given inductive problem. For instance, the following proof snippet shows an alternative proof script for our example:

[1] Proof methods are the Isar syntactic layer of LCF-style tactics.

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

Here we passed the `itrev.induct` rule to the `rule` field of the `induct` method and proved the lemma by recursion induction [2] over `itrev`. This rule was derived by Isabelle automatically when we defined `itrev`, and it states the following:

```
(!!ys. P [] ys) ==>
(!!x xs ys. P xs (x # ys) ==> P (x # xs) ys) ==>
P a0 a1
```

Essentially, this rule states that to prove a property `P` of `a0` and `a1` we have to prove it for two cases where `a0` is the empty list and the list with at least two elements. When the `induct` method takes this rule and `xs` and `ys` as induction variables, Isabelle produces the following sub-goals:

```
 1. !!ys. itrev [] ys = rev [] @ ys
 2. !!x xs ys. itrev xs (x # ys) = rev xs @ x # ys ==>
               itrev (x # xs) ys = rev (x # xs) @ ys
```

where the two sub-goals correspond to the two clauses in the definition of `itrev`.

There are other lesser-known techniques to handle difficult inductive problems using the `induct` method, and sometimes users have to develop useful auxiliary lemmas manually; however, for most cases the problem of how to apply induction boils down to the the following three questions:

- On which terms do we apply induction?
- Which variables do we generalize?
- Which rule do we use for recursion induction?

Isabelle experts resort to induction heuristics to answer such questions and decide what arguments to pass to the `induct` method; however, such reasoning still requires human engineers to carefully investigate the inductive problem at hand. Moreover, Isabelle experts' induction heuristics are sparsely documented across various documents, and there was no way to encode their heuristics as programs. For the wide spread adoption of complete formal verification, we need a program language to encode such heuristics and the system to check if an invocation of the `induct` method written by an Isabelle novice complies with such heuristics. We developed `LiFtEr`, taking these three groups of questions as a design space.

[2] Recursion induction is also known as functional induction or computation induction.

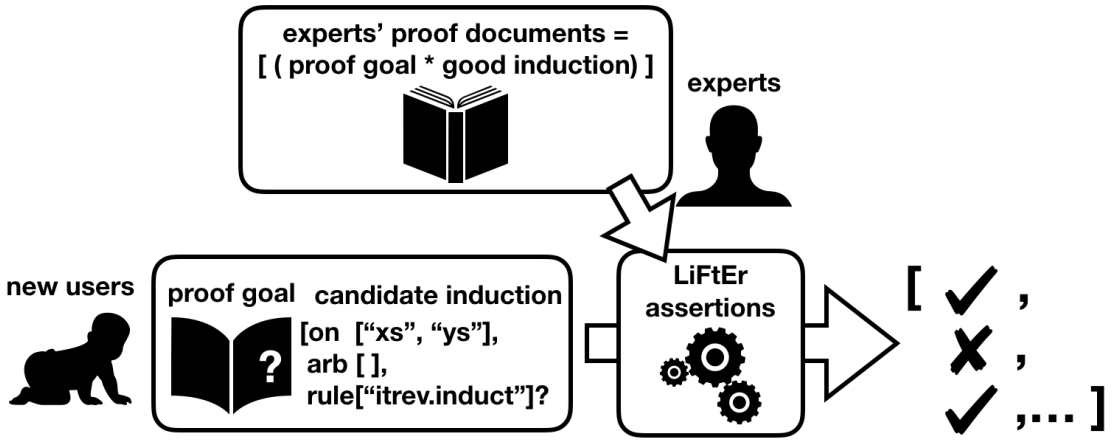



Figure 4.1: The Workflow of LiFtEr.

## 4.3 Overview and Syntax of LiFtEr

We designed LiFtEr to encode induction heuristics as assertions on invocations of the induct method in Isabelle/HOL. An assertion written in LiFtEr takes the pair of a proof goal with its underlying proof state and arguments passed to the induct method. When one applies a LiFtEr assertion to an invocation of the induct method, LiFtEr's interpreter returns a boolean value as the result of the assertion applied to the proof goals and their underlying proof state.

The goal of a LiFtEr programmer is to write assertions that implement reliable heuristics. A heuristic encoded as a LiFtEr assertion is reliable when it satisfies the following two properties:

1. The LiFtEr interpreter is likely to evaluate the assertion to True when the arguments of the induct method are appropriate for the given proof goal.

2. The LiFtEr interpreter is likely to evaluate the assertion to False when the arguments are inappropriate for the goal.

Fig. 4.1 illustrates the workflow of LiFtEr. Firstly, Isabelle experts encode the gist of promising applications of induction based on experts' proofs. Note that the heuristics encoded in LiFtEr become applicable to problem domains that the experts users have not even encountered at the time of writing the assertions.

When new Isabelle users are facing an inductive problem and are unsure if their application of induction is a valid approach or not, they can apply LiFtEr assertions written by experts using the assert_LiFtEr keyword to their proof goal and their candidate arguments.

LiFtEr's interpreter checks if the pair of new users' proof goal and candidate arguments to the induct method is compatible with the experts' heuristics. If the interpreter evaluates the pair to True, Isabelle prints "Assertion succeeded." in the Output panel of Isabelle/jEdit [Wen12]. If the interpreter evaluates the pair to False, Isabelle

highlights the `assert_LiFtEr` in red and prints "`Assertion failed.`" in the Output panel.

**Program 4** The Abstract Syntax of `LiFtEr`.

*assertion* := *atomic* | *connective* | *quantifier* | ( *assertion* )
*type* := `term` | `term_occurrence` | `rule` | `number`
*modifier_term* := `induction_term` | `arbitrary_term`
*quantifier* := $\exists x$ : *type*. *assertion*
| $\forall x$ : *type*. *assertion*
| $\exists x$ : `term` $\in$ *modifier_term* . *assertion*
| $\forall x$ : `term` $\in$ *modifier_term* . *assertion*
| $\exists x$ : `term_occurrence` $\in y$ : `term` . *assertion*
| $\forall x$ : `term_occurrence` $\in y$ : `term` . *assertion*
*connective* := `True` | `False` | *assertion* $\vee$ *assertion* | *assertion* $\wedge$ *assertion*
*assertion* $\rightarrow$ *assertion* | $\neg$ *assertion*
*pattern* := `all_only_var` | `all_constructor` | `mixed`
*atomic* :=
*rule* `is_rule_of` *term_occurrence*
| *term_occurrence* `term_occurrence_is_of_term` *term*
| `are_same_term` ( *term_occurrence* , *term_occurrence* )
| *term_occurrence* `is_in_term_occurrence` *term_occurrence*
| `is_atomic` *term_occurrence*
| `is_constant` *term_occurrence*
| `is_recursive_constant` *term_occurrence*
| `is_variable` *term_occurrence*
| `is_free_variable` *term_occurrence*
| `is_bound_variable` *term_occurrence*
| `is_lambda` *term_occurrence*
| `is_application` *term_occurrence*
| *term_occurrence* `is_an_argument_of` *term_occurrence*
| *term_occurrence* `is_nth_argument_of` *term_occurrence*
| *term* `is_nth_induction_term` *number*
| *term* `is_nth_arbitrary_term` *number*
| `pattern_is` ( *number* , *term_occurrence* , *pattern* )
| `is_at_deepest` *term_occurrence*
| ...

Program 4 shows the essential part of `LiFtEr`'s abstract syntax. `LiFtEr` has four types of variables: `number`, `rule`, `term`, and `term_occurrence`. A value of type `number` is a natural number from 0 to the maximum of the following two numbers: the number of terms appearing in the proof goals at hand, and the maximum arity of constants appearing in the proof goals. A value of type `rule` corresponds to a name of an auxiliary lemma passed to the `induct` method as an argument in the `rule` field.

The difference between `term` and `term_occurrence` is crucial: a value of `term` is a term appearing in proof goals, whereas a value of `term_occurrence` is an *occurrence* of such terms. It is important to distinguish terms and term occurrences because the `induct` method in Isabelle/HOL only allows its users to specify induction terms but it does not allow us to specify on which occurrences of such terms we intend to apply induction.

The connectives, $\wedge$, $\vee$, $\neg$, and $\rightarrow$ correspond to conjunction, disjunction, negation, and implication in the classical logic, respectively; and $\rightarrow$ admits the principle of explosion.

`LiFtEr` has four essential quantifiers and two quantifiers as syntactic sugar. As is often the case, $\forall$ quantifies over variables universally, and $\exists$ stands for the existence of a variable it binds. Again, it is important to notice the difference between the quantifiers over `term` and the ones over `term_occurrence`. For example, $\forall$`_. ` $\in$ `term` quantifies all sub-terms appearing in the proof goals, whereas $\forall$`_. ` $\in$ `term_occurrence` quantifies all *occurrences* of such sub-terms. Quantified variables restricted to `induction_term` by the membership function $\in$ are quantified over all terms passed to the `induct` method as induction terms, while quantified variables restricted to `arbitrary_term` are quantified over all terms passed to the `induct` method as arguments in the `arbitrary` field.

Some atomic assertions judge properties of term occurrences, and some judge the syntactic structure of proof goals with respect to certain terms, their occurrences, or certain numbers. While most atomic assertions work on the syntactic structures of proof goals, `Pattern` provides a means to describe a limited amount of semantic information of proof goals since it checks how terms are defined. Section 4.4 explains the meaning of important atomic assertions through `LiFtEr`'s standard heuristics.

Attentive readers may have noticed that `LiFtEr`'s syntax does not cover any user-defined types or constants. This absence of specific types and constants is our intentional choice to promote induction heuristics that are valid across various problem domains: it encourages `LiFtEr` users to write heuristics that are not specific to particular data-types or functions. And `LiFtEr`'s interpreter can check if an application of the `induct` method is compatible with a given `LiFtEr` heuristic even if the proof goal involves user-defined data-types and functions even though such types and functions are unknown to the `LiFtEr` developer or the author of the heuristic but come into existence in the future only after developing `LiFtEr` and such heuristic.

## 4.4 `LiFtEr`'s Standard Heuristics

This section presents `LiFtEr`'s standard heuristics and illustrates how to use those atomic assertions and quantifiers to encode induction heuristics.

### 4.4.1 Heuristic 1: Induction terms should not be constants.

Let us revise the first example lemma about the equivalence of two reverse functions, `itrev` and `rev`. One naive induction heuristic would be *"any induction term should not*

*be a constant"* [3] In `LiFtEr`, we can encode this heuristic as the following assertion [4]:

```
∀ t1 : term ∈ induction_term.
  ∃ to1 : term_occurrence.
      ( to1  term_occurrence_is_of_term t1 )
    ∧
      ¬ ( is_constant to1 )
```

Note the use of quantifiers over `induction_term`s and `term_occurrence`s: when `LiFtEr` handles induction terms, `LiFtEr` treats them as terms, but it is often necessary to analyze the *occurrences* of these terms in the proof goal to decide how to apply induction. In our example lemma, `xs` is a variable, which appears twice: once as the first argument of `itrev`, and once as the first argument of `rev`. With this in mind, the above assertion reads as follows:

> for all induction terms, named $t1$, there exists a term occurrence, named $to1$, such that $to1$ is an occurrence of $t1$ and $to1$ is not a constant.

Now we compare this heuristic with the model proof by Nipkow *et al.*

The only induction term, `xs`, has two occurrences in the proof goal both as variables. Therefore, if we apply this `LiFtEr` assertion to the model proof, `LiFtEr`'s interpreter acknowledges that the model proof complies with the induction heuristic defined above.

Fig. 4.2 shows the user interface of `LiFtEr`. In the second line where the cursor is staying, `LiFtEr`'s interpreter executes the aforementioned reasoning and concludes that the model proof by Nipkow *et al.* is compatible with this heuristic, printing "`Assertion succeeded.`" in the Output panel. On the contrary, the fourth line applies the same heuristic to another possible combination of arguments to the `induct` method (`induct itrev arbitrary: ys`) and concludes that this candidate induction is not compatible with our heuristic because `itrev` is a constant. `LiFtEr` also highlights this line in red to warn the user.

It is a common practice to analyze occurrences of specific terms when describing induction heuristics. Therefore, we introduced two pieces of syntactic sugar to avoid boilerplate code: ∃_ `: term_occurrence` ∈ _ `: term` and ∀_ `: term_occurrence` ∈ _ `: term`. Both of these quantify over term occurrences of a particular term rather than all term occurrences in the proof goal at hand. Using ∃_ `: term_occurrence` ∈ _ `: term`, we can shrink the above assertion from 5 lines to 3 lines as follows:

```
∀ t1 : induction_term.
  ∃ to1 : term_occurrence ∈ t1 : term.
     ¬  ( is_constant  to1 )
```

In English, this reads as follows:

[3] This *naive heuristic* is not very reliable: there are cases where the `induct` method takes terms involving constants and apply induction appropriately by automatically introducing induction variables. See Concrete Semantics [NK14] for more details.

[4] For better readability we omit parentheses where the binding of terms is obvious from indentation.

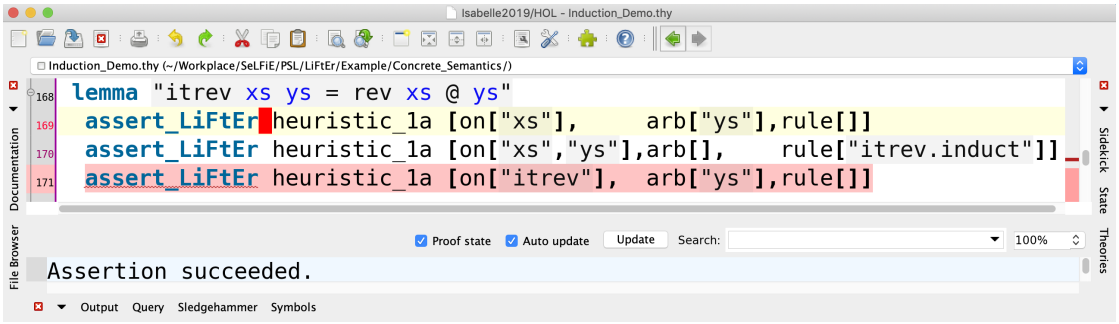



Figure 4.2: The User Interface of LiFtEr.

> for all induction terms, named $t1$, there exists an occurrence of $t1$, named $to1$, such that $to1$ is not a constant.

### 4.4.2 Heuristic 2. Induction terms should appear at the bottom of syntax trees.

Not applying induction on a constant would sound a plausible heuristic, but such heuristic is not very useful.

In this sub-section, we encode an induction heuristic that analyzes not only the properties of the induction terms but also the location of their occurrences within the proof goal at hand. When attacking inductive problems with many variables, it is sometimes a good attempt to apply induction on variables that appear at the bottom of the syntax tree representing the proof goal. We encode such heuristic using `is_at_deepest` as the following LiFtEr assertion:

$$\forall\ t1\ :\ \texttt{induction\_term}.$$
$$\quad \exists\ to1\ :\ \texttt{term\_occurrence} \in t1\ :\ \texttt{term}.$$
$$\qquad \texttt{is\_atomic}\ to1\ \rightarrow\ \texttt{is\_at\_deepest}\ to1$$

In English, this assertion reads as follows:

> for all induction terms, named $t1$, there exists an occurrence of $t1$, named $to1$, such that if $to1$ is an atomic term then $to1$ lies at the deepest layer in the syntax tree that represents the proof goal.

We used the infix operator, $\rightarrow$, to add the condition that we consider only the induction terms that are atomic terms. An atomic term is either a constant, free variable, schematic variable, or variable bound by a lambda abstraction. We added this condition because it makes little sense to check if the induction term resides at the bottom of the syntax tree when an induction term is a compound term: such compound terms have sub-terms at lower layers.

LiFtEr's interpreter acknowledges that the model solution provided by Nipkow *et al.* complies with this heuristic when applied to this lemma: there is only one induction

term, `xs`, and `xs` appears as an argument of `rev` on the right-hand side of the equation in the lemma at the lowest layer of this syntax tree.

### 4.4.3 Heuristic 3. All induction terms should be arguments of the same occurrence of a recursively defined function.

Probably, it is more meaningful to analyze where induction terms reside in the proof goal with respect to other terms in the goal. More specifically, one heuristic for promising application of induction would be *"apply induction on terms that appear as arguments of the same occurrence of a recursively defined function"*. We encode this heuristic using `LiFtEr`'s atomic assertions, `is_recursive_constant` and `is_an_argument_of`, as follows:

```
∃ t1 : term.
  ∃ to1 : term_occurrence ∈ t1 : term.
    ∀ t2: term ∈ induction_term.
      ∃ to2 : term_occurrence ∈ t2 : term.
          is_recursive_constant to1  ∧  to2 is_an_argument_of to1
```

where `is_recursive_constant` checks if a constant is defined recursively or not, and `is_an_argument_of` takes two term occurrences and checks if the first one is an argument of the second one.

Note that using `is_recursive_constant` this assertion checks not only the syntactic information of the proof goal at hand, but it also extracts an essential part of the semantic information of constants appearing in the goal, by investigating how these constants are defined in the underlying proof context. As a whole, this assertion reads as follows:

> there exists a term, named $t1$, such that there exists an occurrence of $t1$, named $to1$, such that for all induction terms, named $t2$, there exists an occurrence of $t2$, named $to2$, such that $to1$ is defined recursively and $to2$ appears as an argument of $to1$.

Attentive readers may have noticed that we quantified over induction terms within the quantification over $to1$, so that this induction heuristic checks if all induction terms occur as arguments of the same constant.

The `LiFtEr` interpreter confirms that the model proof is compatible with this heuristic as well: the constant, `itrev`, is defined recursively and has an occurrence that takes the only induction variable `xs` as the first argument.

### 4.4.4 Heuristic 4. One should apply induction on the nth argument of a function where the nth parameter in the definition of the function always involves a data-constructor.

The previous heuristic checks if all induction terms are arguments of the same occurrence of a recursively defined function. Sometimes we can even estimate on which arguments

of such function we should apply induction by inspecting the definition of the function more carefully.

We introduce two constructs to support this style of reasoning: `is_nth_argument_of` and `pattern_is`. `is_nth_argument_of` takes a term occurrence, a number, and another term occurrence, and it checks if the first term occurrence is the $n$th argument of the second term occurrence where counting starts at 0. `pattern_is` takes a number, a term occurrence, one of three *patterns*: `all_only_var`, `all_constructor`, and `mixed`. Each of such patterns describes how the term is defined.

For example, `pattern_is (`$n$`, `$to$`, all_only_var)` denotes that the $n$th parameter is always a variable on the left-hand side of the definition of the term that has the term occurrence, $to$. Likewise, `all_constructor` denotes the case where the corresponding parameter of the definition of a particular constant always involves a data-constructor, whereas `mixed` denotes that the corresponding parameter is a variable in some clauses but involves a data-constructor in other clauses. With these atomic assertions in mind, we write the following `LiFtEr` assertion:

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

This roughly translates to the following English sentence:

> if there is no argument in the `rule` field in the `induct` method, then there exists a recursively defined constant, $t1$, with an occurrence, $to1$, such that for all induction terms $t2$, there exists an occurrence, $to2$, of $t2$, such that there exists a number, $n$, such that the $n$th parameter involves a data-constructor in all the clauses of the definition of $t1$, and $to2$ appears as the $n$th argument of $to1$ in the proof goal.

Note that we added `¬ (∃r1 : rule. True)` to focus on the case where the `induct` method does not take any auxiliary lemma in the `rule` field since this heuristic is known to be less reliable if there is an auxiliary lemma passed to the `induct` method.

`LiFtEr`'s interpreter confirms that Nipkow's proof about `itrev` and `rev` conforms to this heuristic: there exists an occurrence of `itrev`, such that `itrev` is recursively defined and for the only induction term, `xs`, there is an occurrence of `xs` on the left-hand side of

the proof goal, such that `itrev`'s first parameter involves data-constructor in all clauses of its definition, and this occurrence of `xs` appears as the first argument of the occurrence of `itrev` in the goal [5].

### 4.4.5 Heuristic 5. Induction terms should appear as arguments of a function that has a related `.induct` rule in the `rule` field.

When the `induct` method takes an auxiliary lemma in the `rule` field that Isabelle automatically derives from the definition of a constant, it is often true that we should apply induction on terms that appear as arguments of an occurrence of such constant.

See, for example, our alternative proof, `alt_prf`, for our ongoing example theorem. When Nipkow *et al.* defined the `itrev` function with the `fun` keyword, Isabelle automatically derived the auxiliary lemma `itrev.induct`, and the occurrence of `itrev` on the left-hand side of the equation takes `xs` and `ys` as its arguments. Furthermore, the alternative proof passes `xs` and `ys` to the `rule` field in the same order they appear as the arguments of the occurrence of `itrev` in the proof goal.

We introduce `is_rule_of` to relate a term occurrence with an auxiliary lemma passed to the `rule` field. `is_rule_of` takes a term occurrence and an auxiliary lemma in the `rule` field of the `induct` method, and it checks if the rule was derived by Isabelle at the time of defining the term. Moreover, we introduce `is_nth_induction_term`, which allows us to specify the order of induction terms passed to the `induct` method: `is_nth_induction_term` takes a term and a number, and it checks if the term is passed to the `induct` method as the $n$th induction term. Using these constructs, we can encode the aforementioned heuristic as follows:

```
  ∃ 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
```

As a whole this `LiFtEr` assertion checks if the following holds:

> if there exists a rule, $r1$, in the `rule` field of the `induct` method, then there exists a term $t1$ with an occurrence $to1$, such that $r1$ is derived by Isabelle

[5] Note that in reality the counting starts at 0 internally. Therefore, "the first argument" in this English sentence is processed as the 0th argument within `LiFtEr`.

> when defining $t1$, and for all induction terms $t2$, there exists an occurrence $to2$ of $t2$ such that, there exists a number $n$, such that $to2$ is the $n$th argument of $to1$ and that $t2$ is the $n$th induction terms passed to the `induct` method.

Our alternative proof is compatible with this heuristic: there is an argument, `itrev.induct`, in the `rule` field, and the occurrence of its related term, `itrev`, in the proof goal takes all the induction terms, `xs` and `ys`, as its arguments in the same order.

### 4.4.6 Heuristic 6. Generalize variables in induction terms.

Isabelle's `induct` method offers the `arbitrary` field, so that users can specify which terms to be generalized in induction steps; however, it is known to be a hard problem to decide which terms to generalize.

Of course LiFtEr cannot provide you with a decision procedure to determine which terms to generalize, but it allows us to describe heuristics to identify variables that are likely to be generalized by experienced Isabelle users. For example, experienced users know that it is usually a bad idea to pass induction terms themselves to the `arbitrary` field. We also know that it is often a good idea to generalize variables appearing within induction terms if induction terms are compound terms.

We can encode the former heuristic using `are_same_term`, which checks if two terms are the same term or not. For instance, we can write the following:

```
∀ t1 : term ∈ arbitrary_term.
  ¬ (∃ t2 : term ∈ induction_term. are_same_term (t1, t2))
```

By now, it should be easy to see that this assertion checks if the following holds:

> for all terms in the `arbitrary` field, there is no induction term of the same term in the `induct` method.

The latter heuristic involves the description of the term structure constituting the proof goal. For this purpose we use `is_in_term_occurrence` to check if a term occurrence resides within another term occurrence. With this construct, we can encode the latter heuristic as follows:

```
∃ t1 : term ∈ induction_term.
  ∃ to1 : term_occurrence ∈ t1 : term.
    ∀ t2 : term.
      ∃ to2 : term_occurrence ∈ t2 : term.
          ( to2 is_in_term_occurrence to1 ∧ is_free_variable to2 )
        →
          ∃ t3 : term ∈ arbitrary_term. are_same_term (t2, t3)
```

Again, we used the implication (_ → _) to avoid applying this generalization heuristics to the cases without compound induction terms.

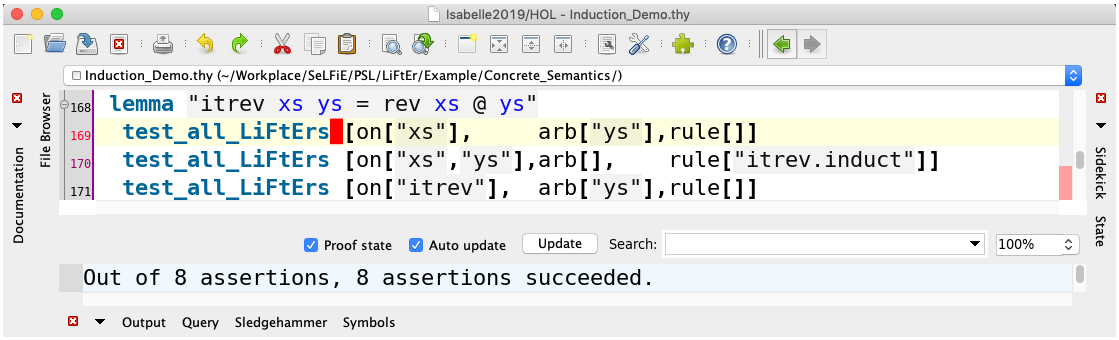



Figure 4.3: The `test_all_LiFtErs` command.

### 4.4.7 Apply all assertions using the `test_all_LiFtErs` command.

In this section we have written eight assertions (two assertions from each of Section 4.4.1 and Section 4.4.6). To exploit all the available `LiFtEr` assertions, we developed the `test_all_LiFtErs` command. The `test_all_LiFtErs` command first takes a combination of induction arguments to the `induct` method. Then, it applies all the available `LiFtEr` assertions to the pair of the combination of arguments and the proof goal at hand. Finally, it counts how many assertions return `True`. For example, the second line in Fig. 4.3 executed the eight available assertions to the combination of arguments (`[on["xs"], arb["ys"], rule[]]`) and the proof goal. The output panel shows the result: `Out of 8 assertions, 8 assertions succeeded.` This indicates that the model proof by Nipkow is indeed a good solution in terms of all the heuristics we discussed in this section.

## 4.5 Induction Heuristics Across Problem Domains

In Section 4.4 we wrote eight assertions in `LiFtEr`. When writing these eight assertions, we emphasized that none of them is specific to the data structure `list` or the function `itrev` appearing in the proof goal. In this section we demonstrate that the `LiFtEr` assertions written in Section 4.4 are applicable across domains, taking an inductive problem from a completely different domain as an example. The following code is the formalization of a simple stack machine from Concrete Semantics [NK14]:

```
type_synonym vname = string
type_synonym val   = int
type_synonym state = "vname => val"
datatype instr     = LOADI val | LOAD vname | ADD
type_synonym stack = "val list"

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

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

`exec1` defines how the stack machine in a certain state transforms a given stack into a new one by executing one instruction, whereas `exec` specifies how the machine executes a series of instructions one by one. Nipkow *et al.* proved the following lemma using structural induction.

```
lemma exec_append_model_prf[simp]:
  "exec (is1 @ is2) s stk = exec is2 s (exec is1 s stk)"
  apply(induct is1 arbitrary: stk) by auto
```

This lemma states that executing a concatenation of two lists of instructions in a state to a stack produces the same stack as executing the first list of the instructions first in the same state to the same stack and executing the second list again in the same state again but to the resulting new stack. As in the case with the equivalence of two reverse functions, there is also an alternative proof based on recursion induction:

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

where `exec.induct` is automatically derived by Isabelle when defining `exec`. Now we check if the heuristics from Section 4.4 correctly recommend these proofs.

**Heuristic 1.** Both `exec_append_model_prf` and `exec_append_alt_prf` comply with this heuristic. For example, `is1` is the only induction term in `exec_append_model_prf`, and it has occurrences in the proof goal, where it occurs as a variable.

**Heuristic 2.** `exec_append_model_prf` complies with the second heuristic: its only induction term, `is1`, occurs at the bottom of the syntax tree as a variable, which is an atomic term. `exec_append_alt_prf` also complies with this heuristic: `is1`, `s`, and `stk` as the arguments of the inner `exec` on the right-hand side of the equation are all atomic terms at the deepest layer of the syntax tree.

**Heuristic 3.** Both proof scripts comply with this heuristic. For example, the inner occurrence of `exec` on the right-hand side of the equation takes all the induction terms of the alternative proof (namely, `is1`, `s`, and `stk`) as its arguments.

**Heuristic 4.** This heuristic works for both proof scripts, but it explains the model answer particularly well: it has a recursively defined constant, `exec`, and the inner occurrence of `exec` on the right-hand side of the equation has an occurrence that takes the only induction term `is1` as its first argument, and the first parameter of `exec` always involve a data-constructor in the definition of `exec`.

**Heuristic 5.** This heuristic also works for both proof scripts, but it fits particularly well with the alternative answer: the rule `exec.induct` is derived by Isabelle when defining `exec`, while `exec` has an occurrence as part of the third argument of another `exec` on the right-hand side of the equation, and this inner occurrence of `exec` takes all the induction terms (`is1`, `s`, and `stk`) in the same order.

**Heuristic 6.** None of our proofs involve induction on a compound term, making the second assertion in Section 4.4.6 rather irrelevant, whereas the first assertion in Section 4.4.6 explains the model answer well: the only generalized term, `stk`, does not appear as an induction term.

## 4.6 Real World Example

In Section 4.4 and Section 4.5, we introduced simple `LiFtEr` assertions applied to smaller problems. For example, all induction terms in the examples were variables, even though Isabelle's `induct` method can induct on non-atomic terms.

Program 5 is a more challenging proof about a formalization of an imperative language, IMP2 [LW19a], from the Archive of Formal Proofs [KNPT04]. Due to the space constraints, we refrain ourselves from presenting the complete formalization of IMP2 but focus on the essential part of the proof document.

In this project, Lammich *et al.* proved the equivalence between IMP2's big-step semantics and small-step semantics. `smalls_seq` in Program 5 is an auxiliary lemma useful to prove the equivalence. The proof of `smalls_seq` appears to be somewhat similar to that of `alt_prf` in Section 4.2 and `exec_append_alt_prf` in Section 4.5: `smalls_seq`'s proof uses the auxiliary lemma `small_steps.induct`, which Isabelle derived automatically when Lammich *et al.* defined `small_steps`. Furthermore, the three induction terms, $\pi$, `(c, s)`, and `Some (c', s')`, are the arguments of one occurrence of `small_steps`.

The difference from the preceding examples is the generalization of four free variables appearing in induction terms: in Program 5, `c` and `s` appear within `(c, s)`, while `c'` and `s'` appear within `Some (c', s')`. As we discussed in Section 4.4.6, when applying induction on non-atomic terms in Isabelle/HOL it is often a good idea to generalize free variables appearing within such non-atomic induction terms.

To encode such heuristic, we strengthened Example 5 in Section 4.4 using the `is_in_term_occurrence` assertion. Program 6 checks if any induction term is non-atomic and contains a free variable, all such free variables are generalized in the `arbitrary` field. Note that `LiFtEr`'s interpreter evaluates the universal quantifier over *to*3 to `True` when all induction terms are atomic, since *to*3 `term_occurrence_is_of_term` *t*3 is guarded by ¬ `( is_atomic` *to*2 `)`, making this assertion valid even for the cases where induction terms are atomic variables.

**Program 5** A Proof about the Semantics of an Imperative Language, IMP2.

```
datatype com =
  SKIP                              (*No-op*)
(*Assignment*)
| AssignIdx vname aexp aexp         (*Assign to index in array*)
| ArrayCpy vname vname              (*Copy whole array*)
| ArrayClear vname                  (*Clear array*)
| Assign_Locals "vname => val"      (*Assign all local variables*)
(*Block*)
| Seq    com  com                   (*Sequential composition*)
| ...

fun small_step :: "program => com × state => (com × state) option" where
  "small_step π ((AssignIdx x i a, s) =
     Some (SKIP, s(x := (s x)(aval i s := aval a s)))"
| "small_step π (ArrayCpy x y, s)    = Some (SKIP, s(x := s y))"
| "small_step π (ArrayClear x, s)    = Some (SKIP, s(x := (λ_. 0)))"
| "small_step π (Assign_Locals l, s) = Some (SKIP, <l|s>)"
| "small_step π (SKIP ;; c, s)       = Some (c, s)"
| "small_step π (c1 ;; c2, s)        = (case small_step π (c1, s) of
     Some (c1', s') => Some (c1' ;; c2, s') | _ => None)"
| ...

inductive small_steps ::
  "program => com × state => (com × state) option => bool" where
  "small_steps π cs (Some cs)"
| "small_step π cs = None ⟶ small_steps π cs None"
| "small_step π cs = Some cs1 ⟶
   small_steps π cs1 cs2 ⟶ small_steps π cs cs2"

lemma smalls_seq:
  "small_steps π (c, s) (Some (c', s')) ⟹
   small_steps π (c ;; cx, s) (Some (c';; cx, s'))"
  apply (induct π "(c, s)" "Some (c', s')"
         arbitrary: c s c' s' rule: small_steps.induct)
  apply (auto dest: small_seq intro: small_steps.intros)
  by (metis option.simps(1) prod.simps(1)
            small_seq small_step.simps(31) small_steps.intros(3))
```

**Program 6** An Assertion for the Generalization of Variables in Induction Terms.

```
  ∃ 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.
                ∃ n1 : number.
                    is_nth_argument_of (to2, n1, to1)
                  ∧
                    t2 is_nth_induction_term n1
              ∧
                ∀ to3 : term_occurrence.
                      ¬ ( is_atomic to2 )
                    ∧
                      is_free_variable to3
                    ∧
                      to3 is_in_term_occurrence to2
                  →
                    ∃ t3 : arbitrary_term.
                    to3 term_occurrence_is_of_term t3
```

## 4.7 Conclusions, Related and Future Work

ITP has been considered as a very challenging task. To address this issue, we presented `LiFtEr`. `LiFtEr` is a domain-specific language in the sense that we developed `LiFtEr` to encode induction heuristics; however, heuristics written in `LiFtEr` are often not specific to any problem domains. To the best of our knowledge, `LiFtEr` is the first programming language developed to capture induction heuristics across problem domains, and its interpreter is the first system that executes meta-reasoning on interactive inductive theorem proving.

The recent development in proof automation for higher-order logic takes the meta-tool approach. Gauthier *et al.*, for example, developed an automated tactic prover, TacticToe, on top of HOL4 [GKU17]. TacticToe leanrs how human engineers used tactics and applies the knowledge to execute a tactic based Monte Carlo tree search. To automate proofs in Coq [TCdt], Komendantskaya *et al.* developed ML4PG [KH17]. ML4PG uses recurrent clustering to mine a proof database and attempts to find a tactic-based proof for a given proof goal. Both of them try to identify useful lemmas or hypotheses as arguments of a tactic; however, they do not identify promising terms as arguments of a tactic even

though identifying such terms is crucial to apply induction effectively.

The most well-known approach for ITP is called the Boyer-Moore waterfall model [Moo73]. This approach was invented for a first-order logic on Common Lisp. Most waterfall provers attempt to apply six proof techniques (simplification, destructor elimination, cross-fertilization, generalization, elimination of irrelevance, and induction) in a fixed order, store the resulting sub-goals in a pool, and keep applying these techniques until the pool becomes empty.

ACL2 [Moo98] is the most commonly used waterfall model based prover, which has achieved industrial-scale proofs [KM97]. When deciding how to apply induction, ACL2 computes a score, called *hitting ratio*, to estimate how good each induction scheme is for the term which it accounts for and proceeds with the induction scheme with the highest hitting ratio [BM79, MW13].

Compared to the hitting ratio used in the waterfall model, `LiFtEr`'s atomic assertions let us analyze the structures of proof goals directly while `LiFtEr`'s quantifiers let us keep `LiFtEr` assertions non-specific to any problem. While ACL2 produces many induction schemes and computes their hitting ratios, `LiFtEr` assertions do not directly produce induction schemes but analyze the given proof goal and the arguments passed to the `induct` method, re-using Isabelle's existing tool to (implicitly) produce induction principles. We consider `LiFtEr`'s approach to be a reasonable choice, since it extends the usability of the already well-developed proof assistant, Isabelle/HOL, while avoiding to reinvent the mechanism to produce induction principle.

Furthermore, the choice of Isabelle/HOL as the host system of `LiFtEr` allowed us to take advantage of human interaction more aggressively both from Isabelle experts and new Isabelle users: Isabelle experts can encode their own heuristics since `LiFtEr` is a language, and new Isabelle users can inspect the results of `LiFtEr` assertions and decide how to attack their proof goals instead of following the fixed order of six proof techniques as in the waterfall model.

Heras *et al.* used ML4PG learning method to find patterns to generalize and transfer inductive proofs from one domain to another in ACL2 [HKJM13]. Jiang *et al.* followed the waterfall model and ran multiple waterfalls [JPF18] to automate ITP in HOL light [Har96]. However, when deciding induction variables, they naively picked the first free variable with recursive type and left the selection of appropriate induction variables as future work.

To determine induction variables automatically, we developed a proof strategy language `PSL` and its default proof strategy, `try_hard` for Isabelle/HOL [NK17]. `PSL` tries to identify useful arguments for the `induct` method by conducting a depth-first search. Sometimes it is not enough to pass arguments to the `induct` method, but users have to specify necessary auxiliary lemmas before applying induction. To automate such labor-intensive work, `PGT` [NP18], a new extension to `PSL`, produces many lemmas by transforming the given proof goal while trying to identify a useful one in a goal-oriented manner.

The drawback of `PSL` and `PGT` is that they cannot produce recommendations if they fail to complete a proof search: when the search space becomes enormous, neither `PSL` and `PGT` gives any advice to Isabelle users.

`PaMpeR` [NH18a], on the other hand, recommends which proof method is likely to be useful to a given proof goal, using a supervised learning applied to the Archive of Formal Proofs [KNPT04]. The key of `PaMpeR` was its feature extractor: `PaMpeR` first applies 108 assertions to each invocation of proof methods and converts each pair of a proof goal with its context and the name of proof method applied to that goal into an array of boolean values of length 108 because this simpler format is amenable for machine learning algorithms to analyze. The limitation of `PaMpeR` is, unlike `PSL`, it cannot recommend which arguments in the `induct` method to tackle a given proof goal.

Taking the same approach as `PaMpeR`, we attempted to build a recommendation tool, `MeLoId` [Nag18], to automatically suggest promising arguments for the `induct` method without completing a proof: we wrote many assertions in Isabelle/ML. Unfortunately, encoding induction heuristics as assertions directly in Isabelle/ML caused an immense amount of code-clutter, and we could not encode even the human-friendly notion of depth in syntax tree since multi-arity functions are represented as curried functions in Isabelle. Therefore, we developed `LiFtEr`, expecting that `LiFtEr` serves as a language for feature extraction.

We hope that when combined into the supervised learning framework of `MeLoId`, assertions written in `LiFtEr` extract the essence of induction in Isabelle/HOL in a cross-domain style and produce a useful database for machine learning algorithms, so that new Isabelle users can have the recommendation of promising arguments for the `induct` method in a fully automatic way.

# Chapter 5

# Smart Induction for Isabelle/HOL (Tool Paper)

## Publication Details

Yutaka Nagashima. Smart induction for Isabelle/HOL (tool paper). In *Proceedings of the 20th Conference on Formal Methods in Computer-Aided Design – FMCAD 2020*, 2020

## Abstract

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.

## 5.1 Introduction

Proof by induction lies at the heart of verification of computer programs that involve recursive data-structures, recursion, or iteration [Gra05]. To facilitate proofs by induction, interactive theorem provers, such as Isabelle/HOL [NPW02], Coq [TCdt], and HOL[SN08], offers tactics. Yet, it requires prover specific expertise to be familiar with such tactics, and human developers have to manually investigate each inductive problem to decide how to apply such tactics.

Unfortunately, the automation of proof by induction is considered as a long standing challenge in computer science, for which Gramlich [Gra05] presented the following conjecture in 2005:

> in the near future, inductive theorem proving will only be successful for very specialised domains for very restricted classes of conjectures. Inductive theorem proving will continue to be a very challenging engineering process [Gra05].

We challenge his conjecture with `smart_induct`, a recommendation tool for proof by induction in Isabelle/HOL. Given an inductive problem in *any* domain, `smart_induct` suggests how one should apply `induct` to attack that problem.

## 5.2 Proof by Induction in Isabelle/HOL

Given the following two simple reverse functions defined in Isabelle/HOL [NPW02], how do you prove their equivalence [NK14]?

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

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

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

where `#` is the list constructor, and `@` appends two lists. Using `induct` of Isabelle/HOL, we can prove this inductive problem in multiple ways:

```
lemma prf1: "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys) by auto
lemma prf2: "itrev xs ys = rev xs @ ys"
  apply(induct xs ys rule:itrev.induct)
  by auto
```

`prf1` applies structural induction on `xs` while generalising `ys` before applying induction by passing `ys` to the `arbitrary` field. It is worth noting that `induct` determines the default induction principle in `prf1` from the induction term, `xs`. On the other hand, `prf2` applies functional induction (also known as computation induction) on `itrev` by the induction principle, `itrev.induct`, to the `rule` field.

There are other lesser-known techniques to handle difficult inductive problems using `induct`, and sometimes users have to develop useful auxiliary lemmas manually; however, for most cases the problem of how to apply induction boils down to the the following three questions:

- On which terms to apply induction?

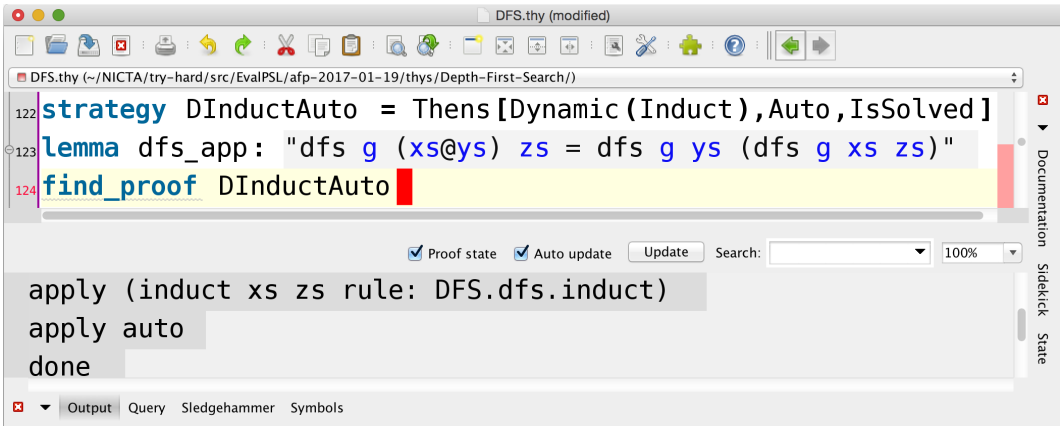



Figure 5.1: The workflow of `smart_induct`.

- Which variables to generalise using the `arbitrary` field?
- Which rule to use for functional induction or rule inversion (as known as rule induction) in the `rule` field?

To answer these questions automatically, we previously developed a proof strategy language, `PSL` [NK17]. Given an inductive problem, `PSL` produces various combinations of induction arguments for `induct` and conducts an extensive proof search based on a given strategy. If `PSL` completes a proof search, it identifies the appropriate combination of arguments for the problem and presents the combination to the user; however, when the search space becomes enormous, `PSL` cannot find a proof within a realistic timeout and fails to provide any recommendation, even if `PSL` produces the right combination of induction arguments. For further automation of proof by induction, we need a tool that satisfies the following two criteria:

- The tool suggests right induction arguments without completing a proof search.
- The tool suggests right induction arguments for any inductive problems.

In this paper we present `smart_induct`, a recommendation tool that addresses these criteria. `smart_induct` is available at GitHub [Nag] together with our running example and the evaluation files discussed in Section 5.4.

The implementation of `smart_induct` is specific to Isabelle/HOL; however, the underlying concept is transferable to other tactic-based proof assistants including HOL4 [SN08], Coq [TCdt], and Lean [dMKA+15]. We developed `smart_induct` as an interactive tool, but one can take its approach to narrow the search space for automatic inductive provers, such as ACL2 [BM79] and Imandra [PCI+20].

To the best of our knowledge `smart_induct` is the first recommendation tool that uses a logic to analyze the syntactic structures of proof goals and advises how to apply `induct` across problem domains without completing to a proof search.

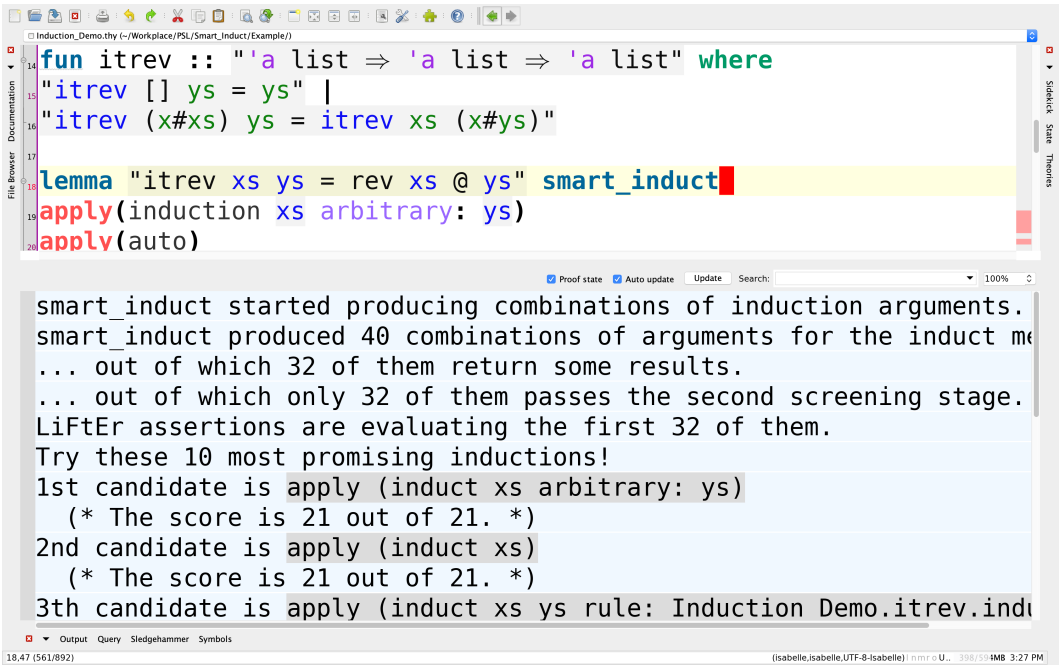



Figure 5.2: The user-interface of `smart_induct`.

## 5.3 Generating and Filtering Tactics

Fig. 5.1 illustrates the internal workflow of `smart_induct`: when invoked by a user, the first step produces many variants of `induct` with different combinations of arguments. Secondly, the multi-stage screening step filters out less promising combinations of induction arguments. Thirdly, the scoring step evaluates each combination to a natural number using logical feature extractors implemented in `LiFtEr` [Nag19a] and reorder the combinations based on their scores. Lastly, the short-listing step takes the best 10 candidates and prints them in the Output panel of Isabelle/jEdit as shown in Fig. 5.2. In this section, we explore details of Step 1 to Step 3.

### 5.3.1 Step 1: Creation of Many Induction Tactics.

`smart_induct` inspects the given proof goal and produces a number of combinations of arguments for `induct` taking the following procedure: `smart_induct` collects variables and constants appearing in the goal. If a constant has an associated induction rule in the underlying proof context, `smart_induct` also collects that rule. From these variables and induction rules, `smart_induct` produces the power set of combinations of arguments for `induct`. Then, for each member of the power set `smart_induct` computes the permutation of the induction variables since `induct` behaves differently for different orders of induction variables. Finally, `smart_induct` produces a tactic for each well-typed permutation of induction variables for each member of the power set.

In our example, `smart_induct` picks up `xs` and `ys` as variables and `itrev` and `rev`

as constants, from which it finds `itrev.induct` as an induction rule, which Isabelle derived automatically when defining `itrev`. From these variables and rule, `smart_induct` produces 40 combinations of induction arguments.

If the size of this set is enormous, we cannot store all the produced induction tactics in our machines. Therefore, `smart_induct` produces this set as a lazy sequence and takes only the first 10,000 combinations for further processing.

### 5.3.2 Step 2: Multi-Stage Screening.

10,000 is still a large number, and feature extractors used in Step 3 often involve nested traversals of nodes in the syntax tree representing a proof goal, leading to high computational costs. Fortunately, the application of `induct` itself is not computationally expensive in most cases: we can apply `induct` to a proof goal and have intermediate sub-goals at a low cost. Therefore, in Step 2, `smart_induct` applies `induct` to the given proof goal using the various combinations of arguments from Step 1 and filter out some of them through the following two stages.

**Stage 1 focuses on `inducts` that return some results** in the first stage, `smart_induct` filters out those combinations of induction arguments, with which Isabelle/HOL does not produce an intermediate goal. Since we have no known theoretical upper bound for the computational cost for `induct`, we also filter out those combinations of arguments, with which `induct` does not return a result within a pre-defined timeout. In our running example, this stage filters out 8 combinations out of 40.

**Stage 2 discards `induct` tactics that return unpromising results** taking the results from the previous stage, Stage 2 scans both the original goal and the newly introduced intermediate sub-goals at the same time to further filter out less promising combinations. More concretely, this stage filters out all combinations of arguments if they satisfy any of the following conditions.

- Some of newly introduced sub-goals are identical to each other.
- A newly introduced sub-goal contains a schematic variable even though the original first sub-goal did not contain a schematic variable.

In our example, Stage 2 does not filter out any combination. Note that these tests on the original goal and resulting sub-goals do not involve nested traversals of nodes in the syntax tree representing goals. For this reason, the computational cost of this stage is often lower than that of Step 3.

### 5.3.3 Step 3: Scoring Induction Arguments using `LiFtEr`.

Step 3 carefully investigates the remaining candidates using heuristics implemented in `LiFtEr` [Nag19a]. `LiFtEr` is a domain-specific language to encode induction heuristics in a style independent of problem domains. Given a proof goal and combination of induction

**Program 7** A `LiFtEr` heuristic used in `smart_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
```

arguments, the `LiFtEr` interpreter mechanically checks if the combination is appropriate for the goal in terms of a heuristic written in `LiFtEr`. The interpreter returns `True` if the combination is compatible with the heuristic and `False` if not. We illustrated the details of `LiFtEr` in our previous work [Nag19a] with many examples. In this paper, we focus on the essence of `LiFtEr` and show one example heuristic used in `smart_induct`.

`LiFtEr` supports four types of variables: natural numbers, induction rules, terms, and term occurrences. An induction rule is an auxiliary lemma passed to the `rule` field of `induct`. The domain of terms is the set of all sub-terms appearing in a given goal. The logical connectives ($\vee$, $\wedge$, $\rightarrow$, and $\neg$) correspond to the connectives in the classical logic. `LiFtEr` offers atomic assertions, such as `is_rule_of`, to examine the property of each atomic term. Quantifiers bring the power of abstraction to `LiFtEr`, which allows `LiFtEr` users to encode induction heuristics that can transcend problem domains. Quantification over `term` can be restricted to the induction terms used in `induct`.

We encoded 19 heuristics in `LiFtEr` for `smart_induct` and assign weights to these heuristics. Some of them examine a combination of induction arguments in terms of functional induction or rule inversion, whereas others check the combination for structural induction. Program 7, for example, encodes a heuristic for functional induction. In English this heuristic reads as follows:

> if there exists a rule, $r1$, in the `rule` field of the `induct` tactic, then there exists a term $t1$ with an occurrence $to1$, such that $r1$ is derived by Isabelle when defining $t1$, and for all induction terms $t2$, there exists an occurrence $to2$ of $t2$ such that, there exists a number $n$, such that $to2$ is the $n$th argument of $to1$ and that $t2$ is the $n$th induction terms passed to the `induct` tactic.

If we apply this heuristic to our running example, `prf2`, the `LiFtEr` interpreter returns `True`: there is an argument, `itrev.induct`, in the `rule` field, and the occurrence of its

related term, `itrev`, in the proof goal takes all the induction terms, `xs` and `ys`, as its arguments in the same order.

Attentive readers may have noticed that Program 7 is independent of any types or constants specific to `prf2`. Instead of handling specific constructs explicitly, Program 7 analyzes the structure of the goal with respect to the arguments passed to `induct` in an abstract way using quantified variables and logical connectives. This power of abstraction let `smart_induct` evaluate whether a given combination of arguments to `induct` is appropriate for a user-defined proof goal consisting of user-defined types and constants, even though such constructs are not available to the `smart_induct` developers. In fact, none of the `LiFtEr` heuristics used in `smart_induct` relies on constructs specific to any problem domain except for one heuristic, which involves a heuristic about `Set.member`. We developed this particular heuristic for conjectures involving `Set.member` since `Set.member` appears in the standard library of Isabelle/HOL and is used by many Isabelle users.

In Step 3, `smart_induct` applies these heuristics to the results from Step 2. For each heuristic, `smart_induct` gives certain predefined points to each combination of `induct` arguments if the `LiFtEr` interpreter returns `True` for that combination. Then, `smart_induct` reorders these combinations based on their scores and presents the most promising combinations to the user in Step 4.

### 5.3.4 User-Interface

Fig. 5.2 shows a screenshot of Isabelle/jEdit interface with `smart_induct`. The seamless integration into Isabelle's ecosystem makes `smart_induct` easy to install and easy to use: `smart_induct` is free from any dependency to external tools except for Isabelle/HOL itself, and we have incorporated `smart_induct` into Isabelle/Isar [Wen11], Isabelle's proof language, and Isabelle/jEdit, its standard editor. This allows Isabelle users to invoke `smart_induct` by typing `smart_induct` within their proof document and to copy a recommended use of `induct` to the right location in the document with one click.

Since `smart_induct` is a meta-tool to use Isabelle's default induction tactic, once `smart_induct` has been called and the tactic inserted, one can remove the `smart_induct` call.

## 5.4 Evaluation

We evaluated `smart_induct` by measuring its performance. We conducted all evaluations using a MacBook Pro (15-inch, 2019) with 2.6 GHz Intel Core i7 6-core memory 32 GB 2400 MHz DDR4.

### 5.4.1 Database for evaluation.

As our evaluation target, we chose five Isabelle theory files with many inductive problems developed by various researchers from the Archive of Formal Proofs [KNPT04]. In the following, we use the following short names to denote these files:

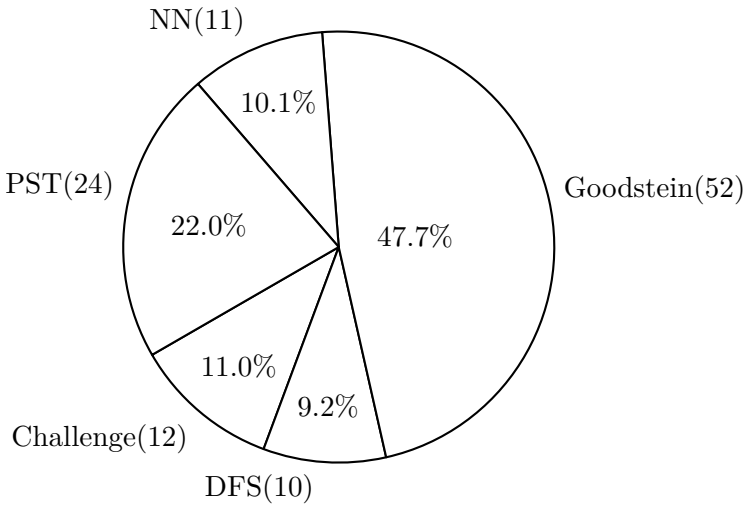



Figure 5.3: Breakdown of the evaluation dataset.

1. *Challenge* stands for `Challenge1A.thy`, which is a part of the solution for VerifyThis2019, a program verification competition associated with ETAPS2019 [LW19b],
2. *DFS* stands for `DFS.thy`, which is a formalisation of depth-first search [NM04b],
3. *Goodstein* is for `Goodstein_Lambda.thy`, which is an implementation of the Goodstein function in lambda-calculus [Fel20],
4. *NN* stands for `Nearest_Neighbors.thy`, which is from the formalisation of multi-dimensional binary search trees [Rau19], and
5. *PST* stands for `PST_RBT.thy`, which is from the formalisation of priority search tree [LN19].

As a whole these files contain 109 calls of `induct`. Fig. 5.3 shows the demographics of our dataset. For example, NN(11) 10.1% mean that `Nearest_Neighbor.thy` contains 11 invocations of `induct`, which accounts for 10.1% of all invocations of `induct` in our dataset.

Fig. 5.4, on the other hand, shows how often proof authors used the `rule` and `arbitrary` fields. In the labels of Fig. 5.4, "w" and "wo" stand for "with" and "without", respectively; whereas "R" and "A" stand for "Rule" and "Arbitrary". For example, "wR-woA(55) 50.5%" represents that among the 109 applications of `induct` 55 of them have an argument in the `rule` field but have no argument in the `arbitrary` field, and this amounts to 50.5%. We greyed the area corresponding to the applications of `induct` with an argument in the `rule` field.

This figure illustrates that in our dataset

- more than half of applications come with a `rule`, and

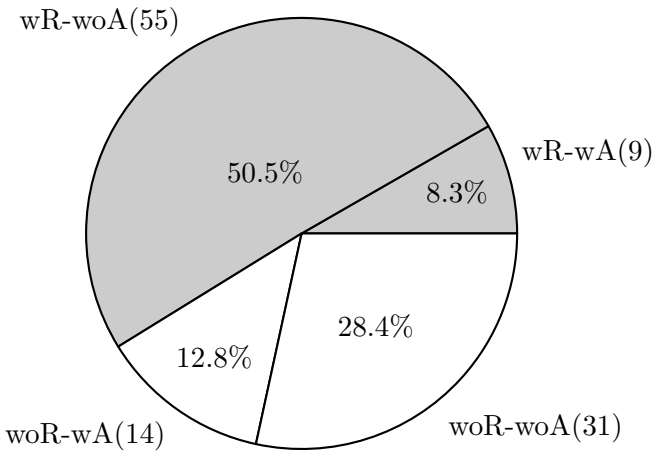



Figure 5.4: Use of `rule` and `arbitrary` fields.

Table 5.1: Scope of `smart_induct`.

| - | w/ handwritten rule | w/o handwritten rule |
|---|---|---|
| w/ compound term | 1 (0.9%) | 1 (0.9%) |
| w/o compound term | 5 (4.6%) | 102 (93.6%) |

- applications of `induct` with a `rule` are less likely to involve generalisation.

Table 5.1 shows how many proofs by induction in the evaluation dataset reside within the scope of `smart_induct`. For example, 102(93.6%) for "w/o compound term" and "w/o handwritten rule" means the following: for 102 proofs by induction out of 109, developers of this dataset used `induct` without applying induction on a compound term nor using an induction rule in the `rule` field that was conjectured and proved manually by a human developer.

These 102 proofs by induction are the only ones that reside within the scope of `smart_induct` because Step 1 of `smart_induct` does not create `induct`s on compound terms or `induct`s with induction principles that were not derived by Isabelle automatically when defining a constant appearing in the proof goal at hand.

Conversely, the remaining three entries in Table 5.1 correspond to the invocations of `induct` that lie outside the scope of `smart_induct`. And such invocations amount to 7 (6.4%) out of 109.

### 5.4.2 Coincidence Rate.

The most important aspect of this tool would be the accuracy of its recommendation. Unfortunately, it is in general not possible to measure if a combination of induction

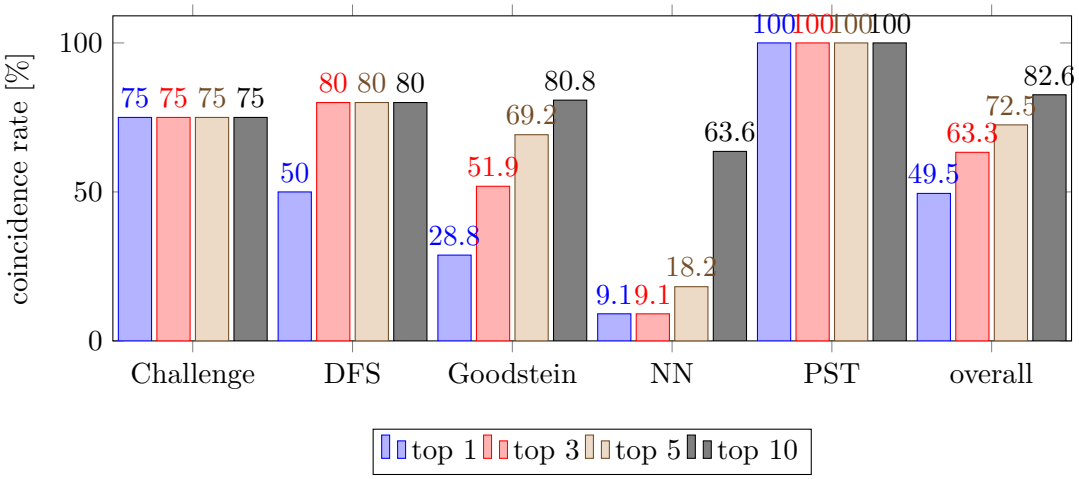



Figure 5.5: Coincidence rates for each theory file.

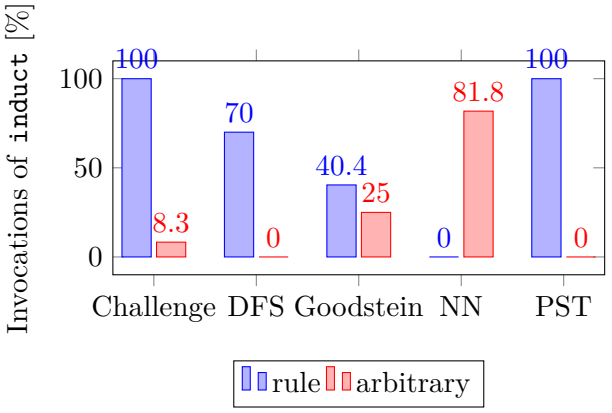



Figure 5.6: Inductions with a `rule` or generalization.

arguments is correct for a goal because many proofs by induction can be valid for one inductive problem. For our running example, we have two proofs, `prf1` and `prf2`, and both of them are equally good. In this particular case, we can confirm the correctness of these combinations of induction arguments by completing the corresponding proof attempts; however, the necessary proof scripts that follow `induct`, in general, can be arbitrarily long, and for this reason it is not possible to mechanically check whether a combination of induction arguments is correct or not.

Since we cannot directly measure the true success rate of `smart_induct`, we evaluated the trustworthiness of `smart_induct`'s recommendations using *coincidence rates*: we counted how often its recommendation coincides with the choices of Isabelle experts. Since we often have multiple equally valid combinations of induction arguments for a given proof goal, we should regard a coincidence rate as a conservative estimate of true success rate.

On the other hand, we can safely consider our coincidence rates as the lower bound for the true success rates since we collected our evaluation targets from the Archive of Formal Proofs [KNPT04], which accepts Isabelle proof documents only after the peer-reviewing process by Isabelle experts.

Fig. 5.5 shows coincidence rates for each theory file and the entire dataset separately. The four bars for each theory file represent the corresponding success rates among top $n$ recommendations, where $n$ is 1, 3, 5, and 10 from left to right. For example, top 3 for Goodstein is 51.9%. This means the following: when `smart_induct` recommends three most promising combinations induction arguments to 52 inductive problems in `Goodstein_Lambda.thy`, for 51.9% out of 52 problems in this file one of the three combinations of induction arguments recommended by `smart_induct` *coincides* with the choice of human proof author.

As mentioned earlier, we should regard a coincidence rate as a conservative estimate of true success rate. Therefore, 51.9% mentioned above should be interpreted as following: `smart_induct`'s recommendation coincides with the choice of experienced Isabelle user for 51.9% of times when it is allowed to recommend three combinations of arguments, but the real success rate of `smart_induct`'s recommendation can be higher than 51.9%.

Notably the rightmost group of bars in Fig. 5.5 shows that `smart_induct` can recommend the choice of human engineer as the most promising application of `induct` for at least roughly half of the cases (49.5%).

A quick glance over Fig. 5.5 would give the impression that `smart_induct`'s performance depends heavily on problem domains: `smart_induct` demonstrated the perfect result for PST, whereas the coincidence rate for NN remains at 18.2% for top_5.

However, a closer investigation of the results reveals that the different coincidence rates come from the style of induction rather than domain specific items such as the types or constructs appearing in goals.

To corroborate this claim, we illustrate how each proof author used `induct` to develop each theory file in Fig. 5.6. In this figure each pair of bars presents how often `induct` comes with an argument in the `rule` field and `arbitrary` field, respectively. For example, the left bar for Goodstein is 40.4% whereas its right bar is 25.0%. This means that `induct` is applied with an argument in the `rule` field for 40.4% of times in Goodstein, and `induct` generalises a variable using the `arbitrary` field for 25.0% of times in the same file.

Together with Fig. 5.5, Fig. 5.6 shows that `smart_induct` tends to show a higher coincidence rate for theory files with a high proportion of `induct`s with an argument in the `rule` field and a lower proportion of the tactics with generalisation using the `arbitrary` field. NN and PST are two extreme examples: In NN, 81.8% of applications of `induct` involve generalisation while no application of the tactic has an argument in the `rule` field in Fig. 5.6, and `smart_induct`'s coincidence rates are lowest for NN. On the contrary, PST has no application involving generalisation while all applications use the `rule` field, and `smart_induct`'s showed the perfect result for PST.

To further investigate how the style of induction affects the coincidence rate of `smart_induct`, we measured coincidence rates based on the use of the `rule` and `arbitrary` fields in Fig. 5.7 where "w", "wo", and "a" stand for "with", "without", and "arbitrary",

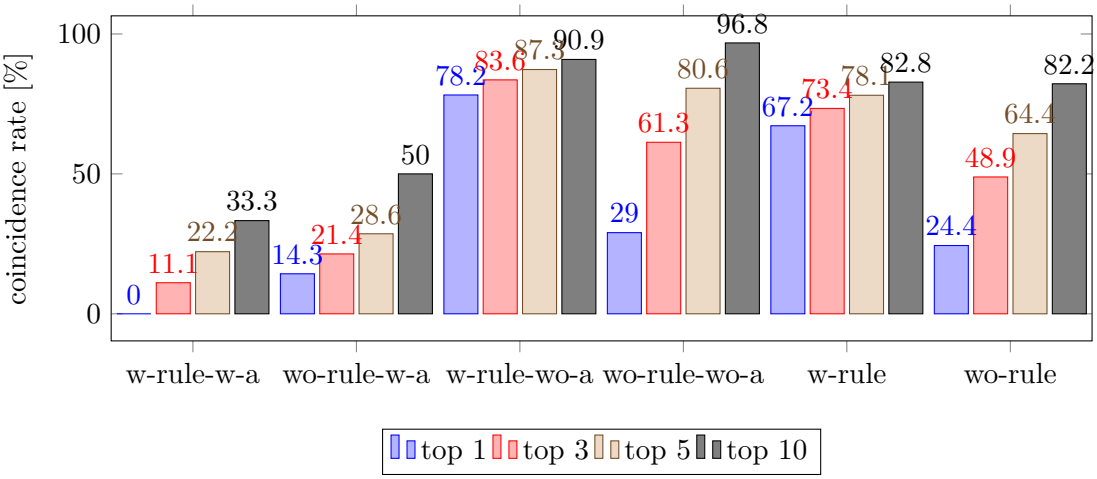



Figure 5.7: Coincidence rates with regard to the `rule` and `arbitrary` fields.

respectively. For example, the leftmost group labelled with "w-rule-w-a" represents the coincidence rates among the applications of `induct` that have arguments in both the `rule` and `arbitrary` fields.

The two right most groups of bars represent the coincidence rates based on the use of `rule` field regardless of the use of the `arbitrary` field. These two groups show that `smart_induct` tends to perform better in predicting how human engineers use `induct` when `induct` has an argument in the `rule` field, which correspond to functional induction and rule inversion.

Interestingly, the two groups in the middle of Fig. 5.7 show that if we focus on the cases without generalisation we can see that the trend among the gaps between the coincidence rates for rule-based inductions (function induction and rule inversion) and the corresponding rates for structural inductions is less clear: we have a wider gap for "top 1", but narrower gaps for "top 3" and "top 5". And for "top 10" we even have a lower coincidence rate for rule-based inductions. Moreover, if we focus on `induct`s involving generalisation, `smart_induct` shows even *lower* coincidence rates for rule-based inductions as shown by the two leftmost groups in Fig. 5.7; even though `smart_induct` overall tends to show *higher* coincidence rates for rule-based inductions.

This seemingly paradoxical phenomenon is best explained by Fig. 5.4, which shows that rule-based inductions less often involve generalisation (14.0%) than structural induction (31.1%) in the dataset: it is still difficult for `smart_induct` to predict which variable to generalise, especially for rule-based inductions, but rule-based inductions tend not to involve variable generalisation to begin with.

To investigate how far generalisation of variables leads to poor coincidence rates, we computed the coincidence rates for `NN` again based on a different criterion: this time we ignored the `arbitrary` fields and took only induction terms and arguments in the `rule` into consideration to measure coincidence rates presented in Fig. 5.8. In Fig. 5.8, the coincidence rate among top 1 is still as low as 9.1% since `smart_induct` often chooses

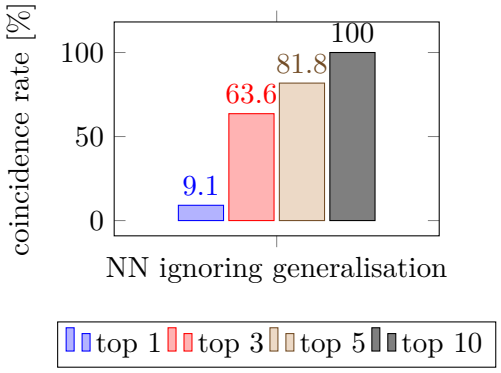



Figure 5.8: Coincidence rates when ignoring `arbitrary`.

a rule-based induction for the most promising candidate, but the overall trend is much better and similar to the rates for w-rule-wo-a in Fig 5.7. The large discrepancies between the numbers for NN in Fig. 5.5 and those in Fig. 5.8 show that even for the most problematic theory file, NN, which contains many structural inductions `smart_induct` is often able to predict on which variables experts apply induction, but it fails to predict which variables to generalise.

The limited performance in predicting experts' use of the `arbitrary` field stems from `LiFtEr`'s limited capability to examine semantic information of proof goals. Even though `LiFtEr` offers quantifiers, logical connectives, and atomic assertions to analyze the syntactic structure of a goal in an abstract way, `LiFtEr` does not offer enough supports to analyze the semantics of the goal. For more accurate prediction of variable generalisation, `smart_induct` needs a language to analyze not only the structure of a goal itself but also the structure of the definitions of types and constants appearing in the goal abstractly.

### 5.4.3 Pruning.

Section 5.3 showed how `smart_induct` produces many candidates of `induct` and prunes less promising ones step by step. We measured how each of these steps contributes to the production of recommendations by counting how many candidates are produced and pruned at each step.

Fig. 5.9 illustrates how many candidates `smart_induct` produced at each step for each proof by induction. The vertical axis denotes the number of candidates after each step for the corresponding proof by induction. White circles and "`+`"es represent the number of remaining candidates for invocations of `smart_induct` when the choice of induction arguments by human authors coincides with one of the 10 most promising combinations recommended by `smart_induct`. For such *successful* cases, we also used a white diamond to depict the corresponding "rank" given by `smart_induct`. For example, if `smart_induct` gives a rank of 3, this means `smart_induct` recommended the choice of human engineer as the third most promising combination of arguments to `induct`.

Along the horizontal axis in Fig. 5.9, we sorted proofs by induction based on the

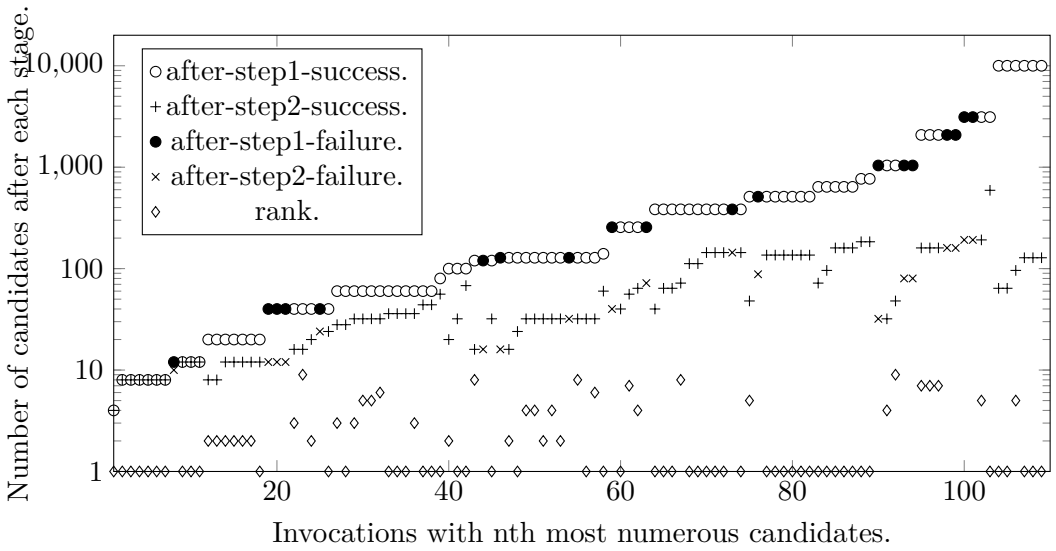



Figure 5.9: Number of Candidates After Each Step.

number of candidates after Step 1. For example, at the right-end of the horizontal axis, we have a circle, a plus, and a diamond. This means for the proof by induction represented by these three points Step 1 produced 10,000 candidates, and Step 2 pruned them down to 128 candidates, and Step 3 ranked the choice of human engineer as the most promising candidate.

On the other hand, black circles and "`x`"es represent the number of candidates for *failed* cases where the choice of induction arguments by human authors did not appear among the top 10 recommendations by `smart_induct`.

One can see that black circles are broadly distributed along the horizontal axis, indicating that the number of initial candidates after Step 1 does not have a strong influence on the accuracy of `smart_induct`.

The use of the logarithmic scale for the vertical axis makes it clear that the number of candidates after Step 1 differs wildly. On the other hand, the number of candidates after Step 2 are mostly contained under 200 with a single exception of 592.

Fig. 5.9 also shows that we had 6 cases where Step 1 reached its upper limit, 10,000. Interestingly, all these cases are successful and 5 of them have the rank of 1. From this, we can judge that the pre-defined upper limit of 10,000 is a descent compromise, which excludes some possible combinations of induction arguments without seriously damaging the coincidence rates of `smart_induct`.

Finally the wide gaps between each "`+`" and its corresponding diamond in Fig. 5.9 indicate that `smart_induct`'s heuristics written in `LiFtEr` effectively nailed down the combination of induction arguments used by human engineers out of many plausible options.

### 5.4.4 Execution Time.

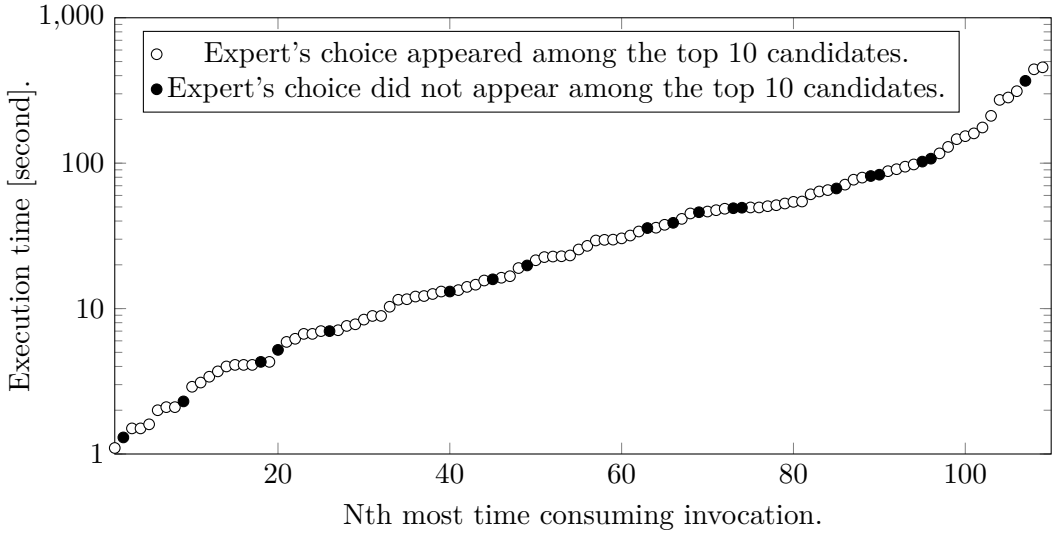



Figure 5.10: Execution Time of `smart_induct`.

For `smart_induct` to be useful, it has to be able to provide valuable recommendations within a realistic time out.

Fig. 5.10 illustrates the distribution of `smart_induct`'s execution time necessary to produce recommendations. The vertical axis represents the execution times in second for each data point, which are sorted along the horizontal axis. As is the case in Section 5.4.3, we filled circles for unsuccessful cases with black.

Similarly to Fig. 5.9, Fig. 5.10 also shows that the unsuccessful cases are spread along the horizontal axis, meaning there is no clear correlation between execution time and the accuracy of recommendation.

We again used the logarithmic scale for the vertical axis. This means that execution times vary largely for different proofs by induction, even though the numbers of candidates after Step 2 are mostly kept below 200, as we saw in Section 5.4.3, This is because the computational cost for each `LiFtEr` heuristic in Step 3 depends on the syntactic structure of each inductive problem, `smart_induct`'s execution time varies for different problems.

The overall median value is 25.5 seconds, which means `smart_induct` can produce a recommendation within 25.5 seconds for half of the problems. In the future we plan to identify and discard less valuable heuristics in Step 3 to speed up `smart_induct`.

## 5.5 Conclusion

We presented `smart_induct`, a recommendation tool for proof by induction in Isabelle/HOL. Our evaluation showed `smart_induct`'s excellent performance in recommending how to apply functional induction and rule inversion and good performance at

identifying induction variables for structural induction for various inductive problems across problem domains. This partially refutes Gramlich's bleak conjecture from 2005. However, recommendation of variable generalisation remains as a challenging task.

It remains as an open question how far we can improve the accuracy and speed of `smart_induct` by combining it with search based systems [NK17, NP18] and approaches based on evolutionary computation [Nag19b] or statistical machine learning [Nag18].

**Related Work** The most well-known approach for inductive problems is called the Boyer-Moore waterfall model [Moo73]. This approach was invented for a first-order logic on Common Lisp. ACL2 [Moo98] is a commonly used waterfall model based prover. When deciding how to apply induction, ACL2 computes a score, called *hitting ratio*, to estimate how good each induction scheme is for the term which it accounts for and proceeds with the induction scheme with the highest hitting ratio [BM79, MW13].

Instead of computing the hitting ratios, `smart_induct` analyzes the structures of proof goals directly using `LiFtEr`. While ACL2 produces many induction schemes and computes their hitting ratios, `smart_induct` does not directly produce induction schemes but analyzes the given proof goal, the arguments passed to the `induct` tactic, and the emerging sub-goals.

Jiang *et al.* ran multiple waterfalls [JPF18] in HOL Light [Har96]. However, when deciding induction variables, they naively picked the first free variable with a recursive type and left the selection of appropriate induction variables as future work.

Machine learning applications to tactic-based provers [NH18a, Nag20c, BLR$^+$19, GKU17, BUG20a, BUG20b] focus on selections of tactics, and the selections of tactic arguments are restricted to premise selections for general-purpose tactics; even though one often has to choose terms for induction arguments to use `induct` effectively.

Sometimes it is not enough to apply `induct` to discharge an inductive problem in Isabelle/HOL but we have to conjecture useful auxiliary lemmas, which we can use to prove the original problem effectively. There are two schools to automate such conjecturing step: bottom-up approach known as theory exploration [Buc00, JRSC14, Joh17, Joh19, EJP18] and top-down approach known as goal-oriented conjecturing [NP18]. For both cases, conjectured lemmas themselves are often inductive problems, which one has to prove by applying proof by induction. For this reason, we plan to achieve complementary strengths by incorporating `smart_induct` into a conjecturing tool.

There was a series of attempts to automate proof by induction in Isabelle/HOL in the style of *rippling* [BSvH$^+$93, BBHI05]. Compared to their approach, we built `smart_induct` on top of the default `induct` tactic, which allowed us to exploit the widely used existing framework for proof by induction in Isabelle/HOL and made the resulting proof scripts maintainable without `smart_induct`.

Reger *et al.* incorporated lightweight automated induction into Vampire [RV19] for saturation-based automated first-order theorem proving [KV13], while we built `smart_induct` for Isabelle/HOL, a tactic-based interactive theorem prover for higher-order logic.

## Acknowledgment

This work was 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) and by NII under NII-Internship Program 2019-2nd call.

# Chapter 6

# PaMpeR: Proof Method Recommendation System for Isabelle/HOL

## Publication Details

Yutaka Nagashima and Yilun He. PaMpeR: proof method recommendation system for Isabelle/HOL. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 362–372, 2018

## Abstract

Deciding which sub-tool to use for a given proof state requires expertise specific to each interactive theorem prover (ITP). To mitigate this problem, we present `PaMpeR`, a proof method recommendation system for Isabelle/HOL. Given a proof state, `PaMpeR` recommends proof methods to discharge the proof goal and provides qualitative explanations as to why it suggests these methods. `PaMpeR` generates these recommendations based on existing hand-written proof corpora, thus transferring experienced users' expertise to new users. Our evaluation shows that `PaMpeR` correctly predicts experienced users' proof methods invocation especially when it comes to special purpose proof methods.

## 6.1 Introduction

Do you know when to use the proof method[1] called `intro_classes` in Isabelle? What about `uint_arith`? Can you tell when `fastforce` tends to be more powerful than `auto`? If you are an Isabelle expert, your answer is "*Sure.*" But if you are new to Isabelle, your answer might be "*No. Do I have to know these Isabelle specific details?*"

Interactive theorem provers (ITPs) are forming the basis of reliable software engineering. Klein *et al.* proved the correctness of the seL4 micro-kernel in Isabelle/HOL [KAE+10]. Leroy developed a certifying C compiler, CompCert, using Coq [Ler09]. Kumar *et al.*

[1] Proof methods are tools used to discharge proof goals in Isabelle. They are similar to *tactic*s in other LCF-style provers.

built a verified compiler for a functional programming language, CakeML, in HOL4 [KMNO14, TMK+19]. In mathematics, mathematicians are replacing their pen-and-paper proofs with mechanised proofs to avoid human-errors in their proofs: Hales *et al.* mechanically proved the Kepler conjecture using HOL-light and Isabelle/HOL [HAB+17], whereas Gonthier *et al.* finished the formal proofs of the four colour theorem in Coq [Gon07]. In theoretical computer science, Paulson proved Gödel's incompleteness theorems using Nominal Isabelle [Pau15].

To facilitate efficient proof developments in such large scale verification projects, modern ITPs are equipped with many sub-tools, such as proof methods and tactics. For example, Isabelle/HOL comes with 160 proof methods defined in its standard library. These sub-tools provide useful automation for interactive theorem proving; however, it still requires ITP specific expertise to pick up the right proof method to discharge a given proof goal.

This paper presents our novel approach to proof method recommendation and its implementation, `PaMpeR`. The implementation is available at GitHub [Nag]. Our research hypothesis is that:

> it is possible to advise which proof methods are useful to a given proof state, based only on the meta-information about the state and information in the standard library. Furthermore, we can extract advice by applying machine learning algorithms to existing large proof corpora.

The paper is organized as follows: Section 6.2 explains the basics of Isabelle/HOL and provides the overview of `PaMpeR`. Section 6.3 expounds how `PaMpeR` transforms the complex data structures representing proof states to simple data structures that are easier to handle for machine learning algorithms. Section 6.4 shows how our machine learning algorithm constructs regression trees from these simple data structures. Section 6.5 demonstrates how users can elicit recommendations from `PaMpeR`. Section 6.6 presents our extensive evaluation of `PaMpeR` to assess the accuracy of `PaMpeR`'s recommendations. Section 6.7 discusses the strengths and limitations of the current implementation and the future work that might improve `PaMpeR`'s performance further or provide even more detailed evaluation of the current implementation. Section 6.8 compares our work with other attempts of applying machine learning and data mining to interactive theorem proving.

## 6.2 Background and Overview of `PaMpeR`

### 6.2.1 Background

Isabelle/HOL is an interactive theorem prover, mostly written in Standard ML. The consistency of Isabelle/HOL is carefully protected by isolating its logical kernel using the module system of Standard ML. *Isabelle/Isar* [Wen02] (*Isar* for short) is a proof language used in Isabelle/HOL. Isar provides a human-friendly interface to specify and discharge proof goals. Isabelle users discharge proof goals by applying *proof methods*, which are the Isar syntactic layer of LCF-style tactics.

Each proof goal in Isabelle/HOL is stored within a *proof state*, which also contains locally bound theorems for proof methods (*chained facts*) and the background *proof context* of the proof goal, which includes local assumptions, auxiliary definitions, and lemmas proved prior to the the current step. Proof methods are in general sensitive not only to proof goals but also to their chained facts and background proof contexts: they behave differently based on information stored in proof state. Therefore, when users decide which proof method to apply to a proof goal, they often have to take other information in the proof state into consideration.

Isabelle comes with many Isar keywords to define new types and constants, such as `datatype`, `codatatype`, `primrec`, `primcorec`, `inductive`, and `definition`. For example, the `fun` command is used for general recursive definitions.

These keywords not only let users define new types or constants, but they also automatically derive auxiliary lemmas relevant to the defined objects behind the user-interface and register them in the background proof context where each keyword is used. For example, Nipkow *et al.* defined a function, `sep`, using the `fun` keyword in an old Isabelle tutorial [NPW02] as follows:

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

Intuitively, this function inserts the first argument between any two elements in the second argument. Following this definition, Isabelle automatically derives the following auxiliary lemma, `sep.induct`, and registers it in the background proof context as well as other four automatically derived lemmas:

```
sep.induct: (!!a. ?P a [])
  ==> (!!a x. ?P a [x])
  ==> (!!a x y zs. ?P a (y # zs)
  ==> ?P a (x # y # zs))
  ==> ?P ?a0.0 ?a1.0
```

where variables prefixed with `?`, such as `?a0.0`, are schematic variables, `!!` is the meta-logic universal quantifier, `==>` is the meta-logic implication[2]. Isabelle also attaches unique names to these automatically derived lemmas following certain naming conventions hard-coded in Isabelle's source code. In this example, the full name of this lemma is `fun0.sep.induct`, which is a concatenation of the theory name (`fun0`), the delimiter (`.`), the name of the constant defined (`sep`), followed by a hard-coded postfix (`.induct`), which represents the kind of this derived lemma.

When users want to prove conjectures about `sep`, they can specify their conjectures using Isar keywords such as `lemma` and `theorem`. The Isar commands, `apply` and `by`, allow

[2]Isabelle/HOL is a specialization of Isabelle for Higher-Order Logic (HOL) formalized in Isabelle's meta-logic. Therefore, it has two versions of universal quantifier and implication: one in the meta-logic and the other one in HOL

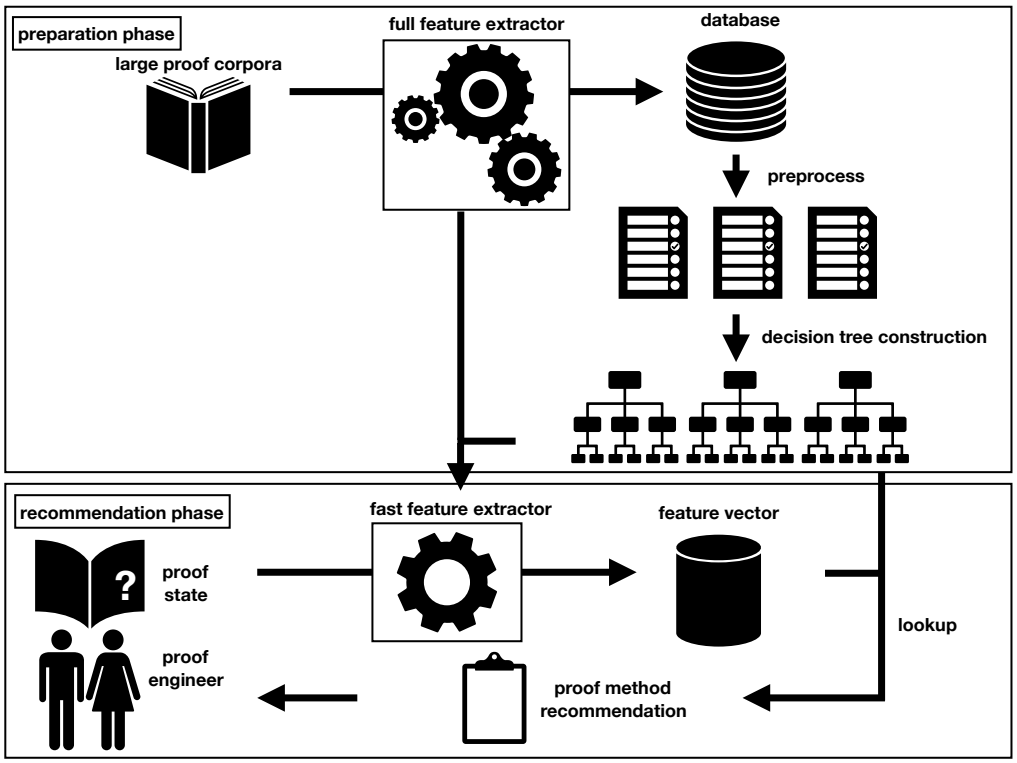



Figure 6.1: Proof attempt with PaMpeR.

users to apply proof methods to these proof goals. In the above example, Nipkow *et al.* proved the following lemma about `map` and `sep` using the automatically derived auxiliary lemma, `sep.induct`, as an argument to the proof method `induct_tac` as following:

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
 apply(induct_tac x xs rule: sep.induct)
 apply simp_all done
```

where `simp_all` is a proof method that executes simplification to all sub-goals and `done` is another Isar command used to conclude a proof attempt.

Isabelle provides a plethora of proof methods, which serve as ammunitions when used by experienced Isabelle users; however, new Isabelle users sometimes spend hours or days trying to prove goals using proof methods sub-optimal to their problems without knowing Isabelle has already specialized methods that are optimized for their goals.

### 6.2.2 Overview of PaMpeR

Figure 6.1 illustrates the overview of PaMpeR. The system consists of two phases: the upper half of the figure shows PaMpeR's preparation phase, and the lower half shows its recommendation phase.

In the preparation phase, PaMpeR's feature extractor converts the proof states in existing proof corpora such as the Archive of Formal Proofs (AFP) [KNPT04] into a database. This database describes which proof methods have been applied to what kind of proof state, while abstracting proof states as arrays of boolean values. This abstraction is a many-to-one mapping: it may map multiple distinct proof states into to the same array of boolean values. Therefore, each array represents a group of proof states sharing certain properties.

PaMpeR first preprocesses this database and generates a database for each proof method. Then, PaMpeR applies a regression algorithm to each database and creates a regression tree for each proof method. This regression algorithm attempts to discover combinations of features useful to recommend which proof method to apply. Each tree corresponds to a certain proof method, and each node in a tree corresponds to a group of proof states, and the value tagged to each leaf node shows how likely it is that the method represented by the tree is applied to these proof states according to the proof corpora used as training sample.

For the recommendation phase, PaMpeR offers three commands, `which_method`, `why_method`, and `rank_method`. The `which_method` command first abstracts the state into a vector of boolean values using PaMpeR's feature extractor. Then, PaMpeR looks up the regression trees and presents its recommendations in Isabelle/jEdit's output panel. If you wonder why PaMpeR recommends certain methods, for example `auto`, to your proof state, type `why_method auto`. Then, PaMpeR tells you why it recommended `auto` to the proof state in jEdit's output panel. If you are curious how PaMpeR ranks a certain method, let us say `intro_classes`, type `rank_method intro_classes`. This command shows `intro_classes`'s rank given by PaMpeR in comparison to other proof methods. In the following, we describe these steps in detail.

## 6.3 Processing Large Proof Corpora

The key component of PaMpeR is its feature extractor: the extractor converts proof goals, chained facts, and proof contexts into arrays of boolean values by applying assertions to them.

### 6.3.1 Representing a Proof State as an Array of Boolean Values

Currently we employ 108 assertions manually written in Isabelle's implementation language, Standard ML, based on our expertise in Isabelle/HOL. Table 6.1 shows selected assertions we used in PaMpeR. Most of these assertions fall into two categories: assertions about proof goals themselves, and assertions about the relation between proof goals and information stored in the corresponding proof context.

Note that PaMpeR's assertions do not directly rely on any user-defined constants because PaMpeR's developers cannot access concrete definitions of user-defined constants when developing PaMpeR. For example, we can check if the first proof goal has a constant defined in the `Set.thy` file in Isabelle/HOL, but we cannot check if that sub-goal has a constant defined in the proof script that some user developed after we released PaMpeR.

However, by investigating how Isabelle/HOL works, we implemented assertions that can check the meta-information of proof goal even without knowing their concrete specifications when developing `PaMpeR`. For example, the lemma presented in Section 6.2.1 has a function, `sep`, which was defined with the `fun` keyword. `PaMpeR`'s feature extractor checks if the underlying proof context contains a lemma of name `sep.elims`. If the context has such a lemma, `PaMpeR` infers that a user defined `sep` using either the `fun` keyword or the `function` keyword, rather than other keywords such as `primcorec` or `definition`.

We wrote some assertions to reflect our own expertise in Isabelle/HOL. One example is the assertion that checks if the proof goal or chained facts involve the constant, `Filter.eventually`, defined in Isabelle's standard library. We developed such an assertion because we knew that the proof method called `eventually_elim` can handle many proof goals involving this constant. But in some cases we were not sure which assertion can be useful to decide which method to use. For example, we have assertions to check if a proof goal has constants defined in `Set.thy`, `Int.thy`, or `List.thy` as these theory files define commonly used concepts in theorem proving. But their effects to proof method selection were unclear until we conducted an extensive evaluation described in Section 6.6.

More importantly, we did not know numerical estimates on which assertion is more useful than others when developing these assertions. For instance, we guessed that the assertion to check the use of the constant `Filter.eventually` to be useful to recommend the use of the `eventually_elim` method, but we did not have means of comparing the accuracy of this guess with other hints prior to this project. To obtain numerical assessments for proof method prediction, we applied the multi-output regression algorithm described in Section 6.4.

The evaluation in Section 6.6 corroborates that it is possible to derive meaningful advice about proof methods. This implies that some parts of the expertise necessary to select appropriate proof methods are based on the meta-information about proof states or the information available within Isabelle's standard library, and our assertion-based feature extractor preserves some essence of proof states while converting them into simpler format.

### 6.3.2 Database Extraction from Large Proof Corpora

The first step of the preparation phase is to build a database from existing proof corpora. We modified the proof method application commands, `apply` and `by`, in Isabelle and implemented a logging mechanism to build the database. The modified `apply` and `by` take the following steps to generate the database:

1. apply assertions to the current proof state,
2. represent the proof state as an array of boolean values,
3. record which method is used to that array,
4. apply the method as the standard `apply` or `by` command, accordingly.

This step requires a slight modification to the Isabelle source code to allow us to overwrite the definition of these command. This way, we build its database by running the target proof scripts.

The current version of `PaMpeR` available at our website [Nag] is based on the database extracted from Isabelle's standard library and the AFP, but the database extraction mechanism is not specific to this library. In case users prefer to optimise `PaMpeR`'s recommendation for their own proof scripts, they can take the same approach following the instructions at our website [Nag], even though this process tends to require significant computational resources.

This overwriting of `apply` and `by` is the only modification we made to Isabelle's source code, and we did so only to build the database for our machine learning algorithm. As long as users choose to use the off-the-shelf default learning results, they can use `PaMpeR` without ever modifying Isabelle's source code. In that case, they only have to include the theory file `PaMpeR/PaMpeR.thy` into their own theory file using the Isar keyword `import` just as a normal theory file to use `PaMpeR`.

Note that logging mechanism ignores the `apply` commands that contain composite proof methods to avoid data pollution. When multiple proof methods are combined within a single command, the naive logging approach would record proof steps that are backtracked to produce the final result. One exemplary data point in an extracted database would look as the following:

```
induct, [1,0,0,1,0,0,0,0,1,0,0,1,0,...]
```

where `induct` is the name of method applied to this proof state and the $n$th element in the list shows the result of the $n$th assertion of the feature extractor when applied to the proof state.

The default database construction from Isabelle standard library and the AFP took about 6,021 hours 43 minutes of CPU time, producing a database consisting of 425,334 unique data points. We used three multi-core server machines[3] to reduce the clock time necessary to obtain this dataset. Unfortunately, this database is heavily imbalanced: some proof methods are used far more often than others. We discuss how this imbalance influenced the quality of `PaMpeR`'s recommendation in Section 6.6.

## 6.4 Machine Learning Databases

In this section, we explain the multi-output regression tree construction algorithm we implemented in Standard ML for `PaMpeR`. We chose a multi-output algorithm because there are in general multiple valid proof methods for each proof goal, and we chose a regression algorithm rather than classification algorithm because we would like to provide numerical estimates about how likely each method would be useful to a given proof goal. We chose a regression tree construction algorithm [BFOS84] because this simple

[3] One of them has 2 Intel(R) Xeon(R) CPUs E5-2698 v3 @ 2.30GHz with 16 cores for each and with hyperthreading, the other two have 2 Intel(R) Xeon(R) CPUs E5-2690 v4 @ 2.60GHz with 14 cores for each with hyperthreading.

algorithm allows us to produce qualitative explanations as to why `PaMpeR` recommends certain methods and it works well for small datasets for rarely used methods as shown in Section 6.6.The comparisons of various machine learning algorithms remain as our future work.

### 6.4.1 Preprocess the Database

We first preprocess the database generated in Section 6.3.2. This process produces a separate database for each proof method from the raw database, which describes the use of most proof methods appearing in the target proof corpora.

Among the class of problem transformation methods for multi-output regression problems, this straightforward approach is called single-target method: it first transforms a single multi-output problem into several single-target problems, then applies a regression algorithm to each of them separately, then combines the results of each regression algorithm to build a single predictor for the original multi-output problem.

For example, if our preprocessor finds the example line discussed in Section 6.3.2, it considers that an ideal user represented by the proof corpora decided to use the `induct` method but not other methods, such as `auto` or `coinduction`, and produces the following line in the database for `induct`:

```
used, [1,0,0,1,0,0,0,0,1,0,0,1,0,...]
```

And the preprocessor adds the following line in the databases for other proof methods appearing in the proof corpora:

```
not, [1,0,0,1,0,0,0,0,1,0,0,1,0,...]
```

Note that the resulting databases do not always represent a provably correct choice of proof methods but conservative estimates. In principle, there could be multiple equally valid proof methods for a single proof state, but existing proof corpora describe only one way of attacking it. For example, Nipkow *et al.* applied the `induct_tac` method to the lemma in Section 6.2.1, but we can prove this lemma with another method for mathematical induction (`induction`) as follows:

```
lemma "map f (sep x xs) = sep (f x) (map f xs)"
 apply(induction x xs rule: sep.induct)
 apply simp_all done
```

For this reason, this preprocessing may misjudge some methods to be inappropriate to a proof state represented by a feature vector in some cases. Unfortunately, exploring all the possible combinations of proof methods for each case is computationally infeasible: some proof methods work well only when they are followed by other proof methods or they are applied with certain arguments, and the combination of these proof methods and arguments explodes quickly.

On the other hand, we can reasonably expect that the proof method appearing in our training sample is the right choice to the proof state represented by the feature vector, since Isabelle mechanically checks the proof scripts. Furthermore, we built the default recommendation using Isabelle's standard library, which was developed by experienced Isabelle developers, and the AFP, which accepts new proofs only after peer reviews by Isabelle experts. This allowed us to avoid low quality proof scripts that Isabelle can merely process but are inappropriate. Therefore, we consider the approximation `PaMpeR`'s preprocessor makes to be a realistic point of compromise and show the effectiveness of this approach in Section 6.6.

### 6.4.2 Regression Tree Construction

After preprocessing, we apply our regression tree construction algorithm to each created database separately. We implemented our tree construction algorithm from scratch in Standard ML for better flexibility and tool integration.

In general, the goal of the regression tree construction is to partition the feature space described in each database into partitions of sub-spaces that lead to the minimal Residual Sum of Squares (RSS)[4] while avoiding over-fitting. Intuitively, RSS denotes the discrepancy between the data and estimation based on a model. The RSS in our problem is defined as follows:

$$RSS = \sum_{j=1}^{J} \sum_{i \in R_j} (used_i - \widehat{used}_{R_j})^2 \tag{6.1}$$

where $R_j$ stands for the $j$th sub-space, to which certain data points (represented as lines in database) belong. The value of $used_i$ is 1.0 if the data point represented by the subscript $i$ says the method was applied to the feature vector, and it is 0.0 if the data point represented by the subscript $i$ says otherwise. $\widehat{used}_{R_j}$ is the average value of $used$ among the data points pertaining to the sub-space $R_j$.

Computing the RSS for every possible partition of the database under consideration is computational infeasible. Therefore, `PaMpeR`'s tree construction takes a top-down, greedy approach, called *recursive binary splitting* [JWHT13].

In recursive binary splitting, we start constructing the regression tree from the root node, which corresponds to the entire dataset for a given method. First, we select a feature in such a way we can achieve the greatest reduction in RSS at this particular step. We find such feature by computing the reduction of the RSS by each feature by one level. For each feature, we split the database into two sub-spaces, $R_{used}(j)$ and $R_{not}(j)$ as follows:

$$\begin{aligned} &R_{used}(j) = \{used | used_j = 1.0\} \text{ and} \\ &R_{not}(j) = \{used | used_j = 0.0\} \end{aligned} \tag{6.2}$$

where $j$ stands for the number representing each feature. Then, for each feature represented by $j$, we compute the following value:

[4] RSS is also known as the sum of squared residuals (SSR).

$$\sum_{i:x_i \in R_{used}(j)} (used_i - \widehat{used}_{R_{used}(j)})^2 + \sum_{i:x_i \in R_{not}(j)} (used_i - \widehat{used}_{R_{not}(j)})^2 \tag{6.3}$$

and choose the feature $j$ that minimizes this value.

Second, we repeat this partition procedure to each emerging sub-node of the regression tree under construction until the depth of tree hits our pre-defined upper limit, three.

After reaching the maximum depth, we compute the average value of $used(j)$ in the corresponding sub-space $R$ for each leaf node. We consider this value as the expectation that the method is useful to proof states abstracted to the combination of feature values to that leaf node.

`PaMpeR` records these regression trees in a text file, so that users can avoid the computationally intensive data extraction and regression tree construction processes unless they want to optimize the learning results based on their own proof corpora.

Note that if we add more assertions to our feature extractor in future, the complexity of this algorithm increases linearly with the number of assertions given a fixed depth of regression tree, since the partition only takes the best step at each level instead of exploring all the combinations of partitions.

## 6.5 Recommendation Phase

Once finishing building regression trees for each proof method appeared in the given proof corpora, one can extract recommendations from `PaMpeR`. When imported to users' theory file, `PaMpeR` automatically reads these trees using the `read_regression_trees` command in `PaMpeR/PaMpeR.thy`.

`PaMpeR` provides three new commands to provide two kinds of information: the `which_method` command tells which proof methods are likely to be useful for a given proof state; the `why_method` command takes a name of proof method and tells why `PaMpeR` would recommend the proof method for the proof state; the `rank_method` command shows the rank of a given method to the proof state in comparison to other proof methods. In the following, we explain how these three commands produce recommendations from the regression trees produced in the preparation phase.

### 6.5.1 Faster Feature Extractor

Before applying the machine learning algorithm, we were not sure which assertion produces valuable features, but after applying the machine learning algorithm, we can judge which assertions are not useful, by checking which features are used to branch each regression tree. The `build_fast_feature_extractor` command in `PaMpeR/ PaMpeR.thy` constructs a faster feature extractor from the regression trees built in the preparation phase and the full feature extractor to reduce the waiting time of `PaMpeR`'s users. It

builds the faster feature extractor by removing assertions that do not result in a branch in the regression trees.

### 6.5.2 The `which_method` Command

When users invoke the `which_method` command, `PaMpeR` applies the faster feature extractor to convert the ongoing proof state into a feature vector, which consists of those features that are deemed to be important to make a recommendation. The speed of this faster feature vector depends on both the regression trees and what each proof state contains. As a rule of thumb, if the proof goal has less terms, it tends to spend less time.

Then, `PaMpeR` looks up the corresponding node in each regression tree and decides the expectation that the method is the right choice for the proof state represented by the feature vector. `PaMpeR` computes this value for each proof method it encountered in the training proof corpora, by looking up a node in each regression tree. Finally, `PaMpeR` compares these expectations and shows the 15 most promising proof methods with their expectations in Isabelle/jEdit's output panel. In the on-going example from Section 6.2.1, a user can know which method to use by typing the `which_method` command as follows:

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

Then, `PaMpeR` shows the following message in the output panel for the top 15 methods[5]:

```
Promising methods for this proof goal are:
  simp with expectation of 0.4119
  auto with expectation of 0.1593
  rule with expectation of 0.0874
  induction with expectation of 0.06137
  metis with expectation of 0.05260 ...
```

Attentive readers might have noticed that `PaMpeR`'s recommendations are not identical to the model answer provided by Nipkow *et al.* This, however, does not immediately mean `PaMpeR`'s recommendation is not valuable: in fact, `PaMpeR` recommended the `induction` method at the fourth place out of 239 proof methods, and `induction` is also a valid method for this proof goal as discussed in Section 6.4.1.

### 6.5.3 The `why_method` Command

Our rather straightforward machine learning algorithm makes `PaMpeR`'s recommendation *explainable*. If you wonder why `PaMpeR` recommends a certain method, for example `case_tac`, to your proof goal, type `why_method case_tac` in the proof script. `PaMpeR` first checks features used to evaluate the expectation for the method and their feature values. Second, `PaMpeR` shows qualitative explanations tagged to both these features and their values in jEdit's output. If you wonder why `PaMpeR` recommended `induction` in the above example, type the following:

[5]Note that we truncated the message due to the space restriction here.

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

Then, you will see this message in jEdit's output panel:

```
Because it is not true that the context has locally
defined assumptions.
Because the underlying proof context has a recursive
simplification rule related to a constant appearing in
the first subgoal.
```

The first reason corresponds to the first branching at the root node in the regression tree for the `induction` method, and the second reason corresponds to the second branching in the tree. In this case, `PaMpeR` found that the proof goal involves the constant, `sep`, and the underlying proof context contains a simplification rule, `sep.simps(3)`, which involves a recursive call of `sep` as following:

```
sep.simp(3):
 sep ?a (?x # ?y # ?zs) = ?x # ?a # sep ?a (?y # ?zs)
```

### 6.5.4 The `rank_method` Command

Sometimes users already have a guess as to which proof method would be useful to their proof state, but they want to know how `PaMpeR` ranks the proof method in mind. Continuing with the above example, if you want to know how `PaMpeR` ranks `conduction` for this proof state, type the following:

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

Then, `PaMpeR` warns you:

```
coinduction 123 out of 239
```

indicating that `PaMpeR` does not consider `coinduction` to be the right choice for this proof goal, before you waste your time on emerging sub-goals appearing after applying `coinduction`.

## 6.6 Evaluation

We conducted a cross-validation to assess the accuracy of `PaMpeR`'s `which_method` command. For this evaluation, we used Isabelle's standard library and the AFP as follows: First, we extracted a database from these proof corpora. This database consists of 425,334 data points. Second, we randomly chose 10% of data points in this database to create the evaluation dataset. Third, we built regression trees from the remaining 90%. There is no overlap between the evaluation dataset and training dataset. Then,

we applied regression trees to each data point in the evaluation dataset and counted how often PaMpeR's recommendation coincides with the proof methods chosen by human proof authors.

Since there are often multiple equally valid proof methods for each proof state, it is only reasonable to expect that `which_method` should be able to recommend the proof method used in the evaluation dataset as one of the most important methods for each proof method invocation. Therefore, for each proof method, we measured how often each proof method used in the evaluation dataset appears among the top $n$ methods in PaMpeR's recommendations.

Table 6.2 shows the results for the 15 proof methods that are most frequently used in the training data in the descending order.

For example, the top row for `simp` should be interpreted as following: The `simp` method was used 102,441 times in the training data. This amounts to 26.8% of all proof method invocations in the training data that are recorded by PaMpeR. In the evaluation dataset, `simp` was used 11,385 times, which amounts to 26.8% of proof method invocations in the evaluation dataset that are recorded by PaMpeR. For 58% out of 11,385 `simp` invocations in the evaluation dataset, PaMpeR predicted that `simp` is the most promising method for the corresponding proof states. For 98% out of 11,385 `simp` invocations in the evaluation dataset, PaMpeR recommended that `simp` is either the most promising method or the second most promising method for the corresponding proof states.

Note that the numbers presented in this table are not the success rates of PaMpeR's recommendation but its conservative estimates. Assume PaMpeR recommends `simp` as the most promising method and `auto` as the second most promising method to a proof goal, say `pg`, in the evaluation dataset, but the human proof author of `pg` chose to apply `auto` to this proof goal. This does not immediately mean that PaMpeR failed to recommend `auto` in the first place, because both `simp` and `auto` might be equally suitable for `pg`. Therefore, the 58% for `simp` mentioned above should be interpreted as follows: PaMpeR's recommendation coincides with the choice of experienced Isabelle user for 58% of times where human engineers applied `simp` when PaMpeR is allowed to recommend only one proof method, but the real success rate of PaMpeR's recommendation can be higher than 58% for these cases. To avoid the confusion with *success rate*, we introduce the term, *coincidence rate*, for this measure. Appendix of our technical report [NH18b] presents three tables to provide the complete list of the evaluation results.

The overall results of this evaluation are as follows: PaMpeR learnt 239 methods from Isabelle's standard library and the AFP: 160 of them are defined within Isabelle's standard library, and the others are user-defined proof methods specified in the AFP entries.

Out of the 239 proof methods PaMpeR learnt from the training dataset, 171 proof methods appeared in the evaluation dataset. Out of these 171 proof methods within the evaluation dataset, 133 methods are defined in Isabelle's standard library, and 38 methods were defined by the AFP authors.

The distribution of proof method usage is heavily imbalanced. The three most frequently used proof methods (`simp`, `auto`, and `rule`) account for 59.1% of all data points in the training dataset, and the ten most frequently used methods account for 79.2% in the training dataset. Similarly in the evaluation dataset, the top three methods account for

58.9%, and the top ten methods for 79.1%.

Fig. 2 illustrates this imbalance, in which the horizontal axis represents the rank of method usage for a proof method and the vertical axis stands for the number of methods invocations for that proof method. For instance, the square located at the top-left corner denotes that the most frequently used proof method in the training dataset (`simp`) is used 102,441 times. And the circle located at $(6, 1093)$ denotes that the sixth most frequently used method in the evaluation dataset (`fastforce`) is used 1,093 times in the evaluation dataset. With the use of logarithmic scale on the vertical axis, this figure presents the serious imbalance of proof method invocations occurring in Isabelle's standard library and the AFP.

Fig. 3 summarises the overall performance of `PaMpeR`. In this figure the horizontal axis represents the number of proof methods `PaMpeR` is allowed to recommend (15 by default), whereas the vertical axis represents the number of proof methods, for which `PaMpeR` achieves certain coincidence rates.

For example, the square at $(3, 23)$ means that `PaMpeR` can achieve 50% of coincidence rate for 23 methods if `PaMpeR` is allowed to recommend three most promising methods. Similarly, `PaMpeR` achieves 50% of coincidence rate for 58 methods when recommending 10 methods and for 72 methods when recommending 15 methods.

The number of methods `PaMpeR` that achieved the four coincident rates (25%, 50%, 75%, and 90%) reached a plateau when `PaMpeR` is allowed to recommend about 60 proof methods.

Overall, `PaMpeR`'s recommendations tend to coincide with human engineers' choice when Isabelle has only one method that is suitable for the proof goal at hand, whereas `PaMpeR`'s recommendations tend to differ from human engineers' choice when there are multiple equally valid proof methods for the same goal. For example, `PaMpeR`'s coincidence rates are low for less commonly used general-purpose methods, such as `safe`, `clarimp`, `best`, `bestsimp` because multiple general purpose proof methods can often handle the same proof goal equally well.

A careful observation at the raw evaluation results provided in the Appendix of the technical report reveals that `PaMpeR` provides valuable recommendations when proof states are best handled by special purpose proof methods, such as `unfold_locales`, `transfer`, `eventually_elim`, `standard`, and so on.

`PaMpeR`'s regression tree construction does not severely suffer from the imbalance among proof method invocation, even though class imbalances often cause problems in other domains such as fraud detection and medical diagnosis [HG09]. The complete evaluation results in Appendix of the report show that `PaMpeR` achieved 50% of coincidence rate for 34 proof methods that appear less than 0.1% of times in the training dataset.

The reason the imbalance did not cause serious problems to `PaMpeR` is that some of these rarely used methods are specialised proof methods, for which we can write assertions that can abstract the essence of the problem very well. Another reason is the fact that commonly used proof methods tend to hold up each other's share, since they address similar problems, lowering expectations for commonly used general purpose methods where both specialised methods and general purpose methods can discharge proof goals.

On the other hand, `PaMpeR` did not produce valuable recommendations to some special

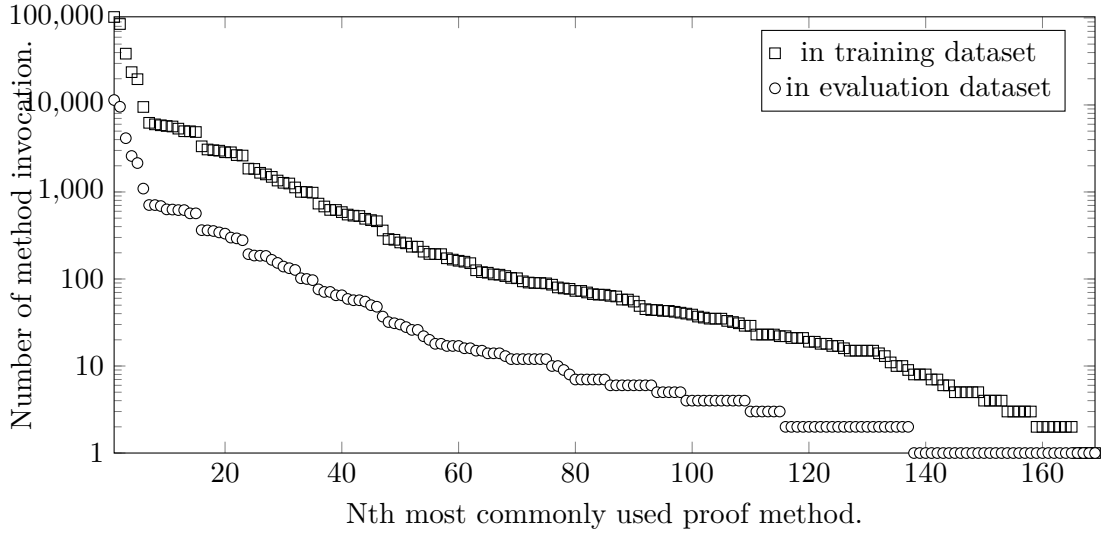



Figure 6.2: Method usage in large proof corpora.

purpose proof methods, such as `vector` and `normalization`, for which we did not manage to develop assertions that capture the properties shared by the proof goals that these methods can handle well. Writing suitable assertions for these remain as our future work.

Some of the proof methods appearing in our evaluation dataset are clearly outside the scope of `PaMpeR`. For example, `cartouche`, `tactic`, `ml_tactic`, `rotate_tac` do not have much semantic meaning: `tactic` is simply an interface between Isabelle's source code language, Standard ML, and Isabelle's proof language, Isar, whereas `rotate_tac` simply rotates the order of premises when a proof goal has multiple premises. Another good example of proof methods outside the scope of `PaMpeR` is the `my_simp` method. This method was defined in the standard library to test the domain specific language, *Eisbach*, for writing new proof methods: `my_simp` is simply a synonym of `simp` and nobody is expected to use `my_simp`. Predicting such methods is not a very meaningful task for `PaMpeR`.

To our surprise, Table V in Appendix of our technical report [NH18b] shows that `PaMpeR`'s recommendation achieved 50% of coincidence rate for 12 methods out of 38 user-defined proof methods defined outside Isabelle's standard library appearing in the evaluation dataset when `PaMpeR` is allowed to provide 15 most promising proof methods, even though `PaMpeR`'s developers did not know anything about these proof methods at the time of development. This suggests that one does not need to know the problem specific information about proof goals to predict the use of some user-defined proof methods. For example, `PaMpeR` achieves 100% of coincidence rate for `sepref` when allowed to recommend only four methods, by checking if the first sub-goal has a schematic variable and if the first sub-goal has variables of type record.

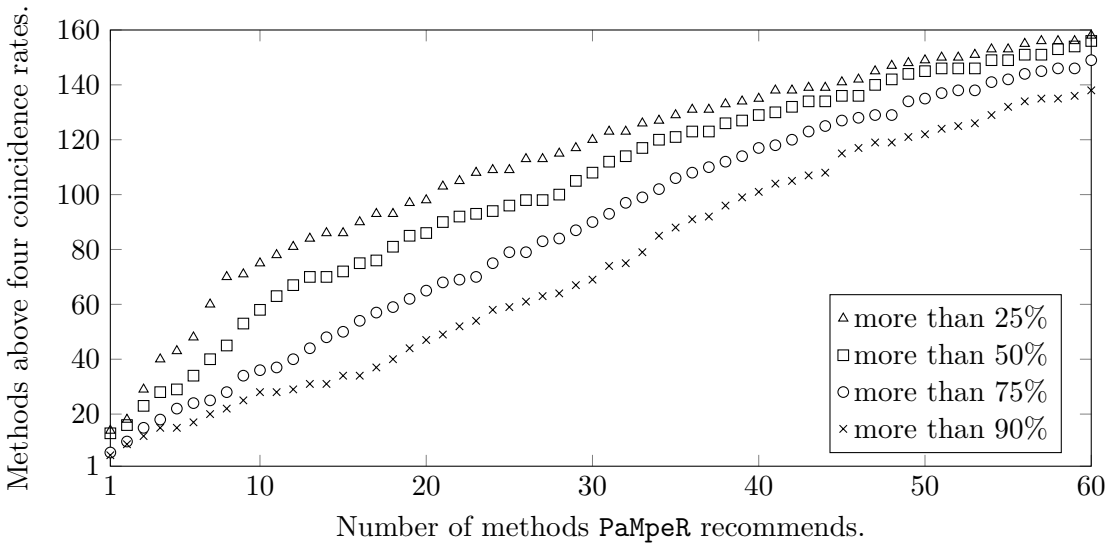



Figure 6.3: Coincidence rate for PaMpeR.

## 6.7 Discussion and Future Work

Prior to PaMpeR, Isabelle had the `print_methods` command, which merely lists the proof methods defined in the corresponding proof context in alphabetical order ignoring the properties of the proof goal at hand. Therefore, new Isabelle/HOL users have to go through various documentations and the archive of mailing lists to learn how to prove lemmas in Isabelle/HOL independently.

Choosing the right methods was a difficult task for new ITP users especially when they should choose special-purpose proof methods, since new users tend not to know even the existence of those rarely used proof methods. Some proof methods are strongly related to certain definitional mechanisms in Isabelle. Therefore, when Isabelle experts use such definitional mechanisms, they can often guess which proof methods they should use later. But this is not an easy task for new users. And this problem is becoming severer nowadays, since large scale theorem proving projects are slowly becoming popular and new ITP users often have to take over proof scripts developed by others and they also have to discharge proof goals specified by others. PaMpeR addressed this problem by systematically transferring experienced users' knowledge to less experienced users. We plan to keep improving PaMpeR by incorporating other Isabelle users intuitions as assertions.

Our manually written feature extractor may seem to be naive compared to the recent success in machine learning research: in some problem domains, such as image recognition and the game of Go, deep neural networks extract features of the subject matters via expensive training. Indeed, others have applied deep neural networks to theorem proving, but without much success [ISA+16, LISK17].

The two major problems of automatic feature extraction for theorem proving is the lack of enormous database needed to train deep neural networks and the expressive nature of the underlying language, i.e. logic. The second problem, the expressive nature of logic, contributes to the first problem: self-respecting proof engineers tend to replace multiple similar propositions with one proposition from which one can easily conclude similar propositions, aiming at a succinct presentation of the underlying concept.

What is worse, when working with modern ITPs, it is often not enough to reason about a proof goal, but one also has to take its proof context into consideration. A proof context usually contains numerous auxiliary lemmas and nested definitions, and each of them is a syntax tree, making the effective automatic feature extraction harder.

Furthermore, whenever a proof author defines a new constant or prove a new lemma, Isabelle/HOL changes the underlying proof context, which affects how one should attack proof goals defined within this proof context. And proof authors do add new definitions because they use ITPs as specification tools as well as tools for theorem proving. Some of these changes are minor modifications to proof states that do not severely affect how to attack proof goals in the following proof scripts, but in general changing proof contexts results in, sometimes unexpected, problems.

For this reason, even though the ITP community has large proof corpora, we essentially deal with different problems in each line of proof corpus. For example, even the AFP has 396 articles consisting of more than 100,000 lemmas, only 4 articles are used by more than 10 articles in the AFP, indicating that many authors work on their own specifications, creating new problems. This results in an important difference between theorem proving in an expressive logic and other machine learning domains, such as image recognition where one can collect numerous instances of similar objects.

We addressed this problem with human-machine cooperation, the philosophy that underpins ITPs. Even though it is hard to extract features automatically, experienced ITP users know that they can discharge many proof goals with shallow reasoning. We encoded experienced Isabelle users' expertise as assertions to simulate their shallow reasoning. Since these assertions are carefully hand-written in Isabelle/ML, they can extract features of proof states (including proof goal, chained facts, and its context) despite the above mentioned problems.

Currently `PaMpeR` recommends only which methods to use and shows why it suggests that method. This is enough for special purpose methods that do not take parameters. For other methods, such as `induct`, it is often indispensable to pass the correct parameters to guide methods. If you prefer to know which arguments to pass to the proof method `PaMpeR` recommends, we would invite you to use `PSL` [NK17], the proof strategy language for Isabelle/HOL, which attempts to find the right combination of arguments via an iterative deepening depth first search based on rough ideas about which method to use. If you want to have those rough ideas, use `PaMpeR`.

`PaMpeR` constructs regression trees of a fixed height. We set the height to a small number, three, to avoid over-fitting. It might be possible that advanced pruning methods can improve the accuracy of `PaMpeR`'s recommendation. Furthermore, since we developed 108 assertions based on our limited expertise, it is likely that we have missed out information valuable to recommend proof methods when abstracting proof states using assertions.

Our cross-validation showed that some simple assertions, such as checking the existence of certain constants in proof goals, turned out to be useful. Therefore, it might be possible to find more useful assertions by systematically enumerating more assertions of this kind to check the existence of other constants appearing in proof corpora. Unfortunately, the database construction based on 108 assertions already consumed serious computational resources, and database construction based on generated assertions remains as our future work due to the limitation of resources currently available to `PaMpeR`'s developers.

When conducting the cross-evaluation, we focused on the coincidence rate for each method. It would be worthwhile to compare the results of `PaMpeR`'s overall coincidence rate for all methods with the corresponding overall coincidence rate that would be produced by a naive system that recommends proof methods in order of their frequency in training data without constructing decision trees.

Finally, our choice of machine learning algorithm is not final. We currently use regression tree construction algorithm based on a problem transformation method because the straightforward algorithm lets us produce qualitative explanations of `PaMpeR`'s recommendation; however, other machine learning algorithms might lead to higher coincidence rates. The comparison of various machine learning algorithms on the dataset remains as our future work.

## 6.8 Conclusion and Related Work

We presented the design and implementation of `PaMpeR`. In the preparation phase, `PaMpeR` learns which method to use from existing proof corpora using regression tree construction algorithm. In the recommendation phase, `PaMpeR` recommends which proof methods to use to a given proof goal and explains why it suggests that method. Our evaluation showed that `PaMpeR` tends to provide valuable recommendations especially for specialised proof methods, which new Isabelle users tend not to be aware of. We also identified problems that arise when applying machine learning to proof method recommendation and proposed our solution to them.

**Related Work** ML4PG [KH17] extends a proof editor, Proof General, to collect proof statistics about shapes of goals, sequence of applied tactics, and proof tree structures. It also clusters the gathered data using machine learning algorithms in MATLAB and Weka and provides proof hints during proof developments. Based on learning, ML4PG lists similar proof goals proved so far, from which users can infer how to attack the proof goal at hand, while `PaMpeR` directly works on proof methods. Compared to ML4PG, `PaMpeR`'s feature extractor is implemented within Isabelle/ML, which made it possible to investigate not only proof goals themselves but also their surrounding proof context.

Gauthier *et al.* developed TacticToe for HOL4 [GKU17]. It selects proved lemmas similar to the current proof goal using premise selection and applies tactics used to these similar goals to discharge the current proof goal. Compared to TacticToe, the abstraction via assertions allows `PaMpeR` to provide valuable recommendations even when similar goals do not exist in the problem domain.

Several people applied machine learning techniques to improve the so-called Hammer-style tools. For Isabelle/HOL, both MePo [MP09] and MaSh [KBKU13] decreased the quantity of facts passed to the automatic provers while increasing their quality to improve Sledgehammer's performance. Their approaches attempt to choose facts that are likely to be useful to the given proof goal, while `PaMpeR` suggests proof methods that are likely to be useful to the goal.

MePo judges the relevance of facts by checking the occurrence of symbols appearing in proof goals and available facts, while MaSh computes the relevance using sparse naive Bayes and k Nearest Neighbours. They detect similarities between proof goals and available facts by checking mostly formalization-specific information and only two piece of meta information, while `PaMpeR` discards most of problem specific information and focus on meta information of proof goals: the choice of relevant fact is a problem specific question, while the choice of proof method largely depends on which Isabelle's subsystem is used to specify a proof goal.

The original version of MaSh was using machine learning libraries in Python, and Blanchette *et al.* ported them from Python to Standard ML for better efficiency and reliability. Similarly, an early version of `PaMpeR` was also using a Python library [PVG+11] until we implemented the regression tree construction algorithm in Standard ML for better tool integration and flexibility. Both MaSh and `PaMpeR` record learning results in persistent states outside the main memory, so that users can preserve the learning results even after shutting down Isabelle.

Blanchette *et al.* analysed the AFP, looking at sizes and dependencies for theory files [BHMN15]. Matichuk *et al.* investigated the seL4 proofs and two articles in the AFP to find the relationship between the size of statement and the size of proof [MMA+15]. None of them analysed the occurrence of proof methods in their target proof corpora nor developed a recommendation system based on their results. Moreover, `PaMpeR`'s database construction is more active compared to their work: it applies 108 hand-written assertions to analyse the properties of not only each proof goal but also the relationship between each goal and its background context and chained facts.

## Acknowledgement

The authors would like to the anonymous referees at ITP2016, ITP2017, and ASE2018 for their valuable comments and helpful suggestions. This work was 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)

Table 6.1: Selected Assertions.

- Assertions about proof goals themselves
  - constants defined in Isabelle's standard library
    - check if the first goal has the `BNF_Def.rel_fun` constant or the `Fun.map_fun` constant
    - check if the first proof goal has `Orderings.ord_class.less_eq`, `Orderings.ord_class.less`, or `Groups.plus_class.plus`.
    - check if the first goal and its chained facts have `Filter.eventually`
  - constants defined in Isabelle's standard library at certain locations in the first proof goal
    - check if the outermost constant of the first goal is the meta-logic universal quantifier
    - check if the first goal has the HOL existential quantifier but not as the outermost constant
  - terms of certain types defined in Isabelle's standard library
    - check if the first goal has a term of type `Word.word`
    - check if the first goal has a schematic variable
  - existence of constants defined in certain theory files
    - check if the first goal has a constant defined in the `Nat` theory
    - check if the first goal has a constant defined in the `Real` theory
    - check if the first goal has a constant defined in the `Set` theory
- Assertions about the relation between proof goals and proof contexts.
  - types defined with a certain Isar keyword
    - check if the goal has a term of a type defined with the `datatype` keyword
    - check if the goal has a term of a type defined with the `codatatype` keyword
    - check if the goal has a term of a type defined with the `record` keyword
  - constants defined with a certain Isar keyword
    - check if the goal has a constant defined with the `lift_definition` keyword
    - check if the goal has a constant defined with the `primcorec` keyword
    - check if the goal has a constant defined with the `inductive` keyword or `inductive_set` keyword.

Table 6.2: Evaluation of PaMpeR on 15 most frequently used proof methods.

| proof method | training | % | evaluation | % | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| simp | 102441 | 26.8 | 11385 | 26.8 | 58 | 98 | 99 | 99 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| auto | 85097 | 22.2 | 9527 | 22.4 | 60 | 94 | 98 | 99 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| rule | 38856 | 10.2 | 4150 | 9.8 | 3 | 15 | 86 | 99 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| blast | 23814 | 6.2 | 2590 | 6.1 | 0 | 26 | 26 | 35 | 84 | 95 | 99 | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| metis | 19771 | 5.2 | 2149 | 5.1 | 0 | 0 | 13 | 72 | 84 | 89 | 93 | 96 | 98 | 99 | 100 | 100 | 100 | 100 | 100 |
| fastforce | 9477 | 2.5 | 1093 | 2.6 | 0 | 0 | 0 | 0 | 5 | 54 | 70 | 81 | 89 | 93 | 96 | 96 | 97 | 98 | 98 |
| force | 6232 | 1.6 | 708 | 1.7 | 0 | 0 | 0 | 0 | 1 | 9 | 22 | 32 | 40 | 51 | 66 | 77 | 84 | 89 | 94 |
| clarsimp | 5984 | 1.6 | 628 | 1.5 | 0 | 12 | 14 | 14 | 20 | 29 | 39 | 49 | 54 | 57 | 62 | 64 | 66 | 67 | 73 |
| cases | 5842 | 1.5 | 689 | 1.6 | 0 | 0 | 1 | 16 | 16 | 20 | 34 | 54 | 70 | 80 | 86 | 91 | 93 | 95 | 96 |
| erule | 5732 | 1.5 | 707 | 1.7 | 0 | 0 | 15 | 38 | 44 | 53 | 64 | 70 | 76 | 82 | 85 | 87 | 91 | 92 | 93 |
| subst | 5655 | 1.5 | 619 | 1.5 | 0 | 0 | 19 | 19 | 19 | 20 | 22 | 28 | 45 | 58 | 69 | 77 | 82 | 86 | 90 |
| rule_tac | 5342 | 1.4 | 631 | 1.5 | 0 | 14 | 32 | 34 | 44 | 45 | 46 | 47 | 50 | 51 | 52 | 52 | 53 | 57 | 63 |
| intro | 4988 | 1.3 | 619 | 1.5 | 0 | 0 | 5 | 18 | 24 | 39 | 46 | 47 | 48 | 48 | 49 | 57 | 69 | 77 | 84 |
| simp_all | 4982 | 1.3 | 568 | 1.3 | 0 | 0 | 0 | 1 | 3 | 6 | 15 | 21 | 26 | 33 | 45 | 60 | 70 | 78 | 83 |
| induct | 4884 | 1.3 | 568 | 1.3 | 0 | 0 | 0 | 1 | 27 | 45 | 49 | 50 | 50 | 51 | 56 | 62 | 71 | 77 | 79 |

# Chapter 7

# Simple Dataset for Proof Method Recommendation in Isabelle/HOL

## Publication Details

Yutaka Nagashima. Simple dataset for proof method recommendation in Isabelle/HOL. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings*, volume 12236 of *Lecture Notes in Computer Science*, pages 297–302. Springer, 2020

## Abstract

Recently, a growing number of researchers have applied machine learning to assist users of interactive theorem provers. However, the expressive nature of underlying logics and esoteric structures of proof documents impede machine learning practitioners, who often do not have much expertise in formal logic, let alone Isabelle/HOL, from achieving a large scale success in this field. In this data description, we present a simple dataset that contains data on over 400k proof method applications along with over 100 extracted features for each in a format that can be processed easily without any knowledge about formal logic. Our simple data format allows machine learning practitioners to try machine learning tools to predict proof methods in Isabelle/HOL without requiring domain expertise in logic.

## 7.1 Introduction

As our society relies heavily on software systems, it has become essential to ensure that our software systems are trustworthy. Interactive theorem provers (ITPs), such as Isabelle/HOL [NPW02], allow users to specify desirable functionalities of a system and prove that the corresponding implementation is correct in terms of the specification.

A crucial step in developing proof documents in ITPs is to choose the right tool for a proof goal at hand. Isabelle/HOL, for example, comes with more than 100 proof methods.

Proof methods are sub-tools inside Isabelle/HOL. Some of these are general purpose methods, such as `auto` and `simp`. Others are special purpose methods, such as `intro_classes` and `intro_locales`. The Isabelle community provides various documentations [NPW02] and on-line supports to help new Isabelle users learn when to use which proof methods.

Previously, we developed `PaMpeR` [NH18a], a proof method recommendation tool for Isabelle/HOL. Given a proof goal specified in a proof context, `PaMpeR` recommends a list of proof methods likely to be suitable for the goal. `PaMpeR` learns which proof method to recommend to what kind of proof goal from proof documents in Isabelle's standard library and the Archive of Formal Proofs [KNPT04].

The key component of `PaMpeR` is its elaborate feature extractor. Instead of applying machine learning algorithms to Isabelle's proof documents directly, `PaMpeR` first applies 113 assertions to the pair of a proof goal and its underlying context. Each assertion checks a certain property about the pair and returns a boolean value. Some assertions check if a proof goal involves certain constants or types defined in the standard library. Others check the meta-data of constants and types appearing in a goal. For example, one assertion checks if the goal has a term of a type defined with the `codatatype` keyword.

When developing `PaMpeR`, we applied these 113 assertions to the proof method invocations appearing in the proof documents and constructed a dataset consisting of 425,334 unique data points.

Note that this number is strictly smaller than all the available proof method invocations in Isabelle2020 and the Archive of Formal Proofs in May 2020, from which we can find more than 900k proof method invocations. One obvious reason for this gap is the ever growing size of the available proof documents. The other reason is that we are intentionally ignoring compound proof methods while producing data points. We decided to ignore them because they may pollute the database by introducing proof method invocations that are eventually backtracked by Isabelle. Such backtracking compound methods may reduce the size of proof documents at the cost of introducing backtracked proof steps, which are not necessary to complete proofs. Since we are trying to recommend proof methods appropriate to complete a proof search, we should not include data points produced by such backtracked steps.

We trained `PaMpeR` by constructing regression trees [BFOS84] from this dataset. Even though our tree construction is based on a fixed height and we did not take advantage of modern development of machine learning research, our cross evaluation showed `PaMpeR` can correctly predict experts' choice of proof methods for many cases. However, decision tree construction based on a fixed height is an old technique that tends to cause overfitting and underfitting. We expect that one can achieve better performance by applying other algorithms to this dataset.

In the following we present the simple dataset we used to train `PaMpeR`. Our aim is to provide a dataset that is publicly available at Zenodo [Nag20d] and easily usable for machine learning practitioners without backgrounds in theorem proving, so that they can exploit the latest development of machine learning research without being hampered by technicalities of theorem proving.

## 7.2 The PaMpeR Dataset

Each data point in the dataset consists of the following three entries:

- the location of a proof method invocation,
- the name of the proof method used there,
- an array of 0s and 1s expressing the proof goal and its context.

The following is an example data point:

```
Functors.thy119 simp 1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,...
```

This data point describes that in the theory file named `Functors.thy`, a proof author applied the `simp` method in line 119 to a proof goal represented by the sequence of `1`s and `0`s where `1` indicates the corresponding assertion returns true while `0` indicates the otherwise.

This dataset has important characteristics worth mentioning. Firstly, this dataset is heavily imbalanced in terms of occurrences of proof methods. Some general purpose methods, such as `auto` and `simp`, appear far more often than other lesser known methods: each of `auto` and `simp` accounts more than 25% of all proof method invocations in the dataset, whereas no proof methods account for more than 1% of invocations except for the 15 most popular methods.

Secondly, this dataset only serves to learn what proof methods to apply, but it does not describe how to apply a proof method. None of our 113 assertions examines arguments passed to proof methods. For some proof methods, notably the `induct` method, the choice of arguments is the hardest problem to tackle, whereas some methods rarely take arguments at all. We hope that users can learn what arguments to pass to proof methods from the use case of these methods in existing proof documents once they learn which methods to apply to their goal.

Thirdly, it is certainly possible that PaMpeR's feature extractor misses out certain information essential to accurately recommend some methods. This dataset was not built to preserve the information in the original proof documents: we built the dataset, so that we can effectively apply machine learning algorithms to produce recommendations.

Finally, this dataset shows only one way to prove a given goal, ignoring alternative possible approaches to prove the same goal. Consider the following goal: `"True` $\vee$ `False"`. Both `auto` or `simp` can prove this goal equally well; however, if this goal appeared in our dataset our dataset would show only the choice of the proof author, say `auto`, ignoring alternative proofs, say `simp`.

One might guess that we could build a larger dataset that also includes alternative proofs by trying to complete a proof using various methods, thus converting this problem into a multi-label problem. That approach would suffer from two problems. Firstly, there are infinitely many ways to apply methods since we often have to apply multiple proof methods in a sequence to prove a conjecture. Secondly, some combinations of methods are not appropriate even though they can finish a proof in Isabelle. For example, the following is an alternative proof for the aforementioned proposition:

```
lemma "True ∨ False" apply(rule disjI1) apply auto done
```

This is a valid proof script, with which Isabelle can check the correctness of the conjecture; however, the application of the `rule` method is hardly appropriate since the subsequent application of the `auto` method can discharge the proof without the preceding `rule`. For these reasons we take the proof methods chosen by human proof authors as the correct choice while ignoring other possibilities.

## 7.3 Overview of 113 Assertions

The 113 assertions we used to build the dataset roughly fall into the following two categories:

1. assertions that check terms and types appearing in the first sub-goal, and
2. assertions that check how such terms and types are defined in the underlying proof context.

The first kind of assertions directly check the presence of constructs defined in the standard library. For example, the 56th assertion checks if the first sub-goal contains `Filter.eventually`, which is a constant defined in the standard library since the presence of this constant may be a good indicator to recommend the special purpose proof method called `eventually_elim`. A possible limitation of these assertions is that these assertions cannot directly check the presence of user-defined constructs because such constructs may not even exist when we develop the feature extractor.

The second kind of assertions address this issue by checking how constructs appearing in the first sub-goal are defined in the proof context. For example, the 13th assertion checks if the first sub-goal involves a constant that has one of the following related rules: the `code` rule, the `ctr` rule, and the `sel` rule.

These related rules are derived by Isabelle when human engineers define new constants using the `primcorec` keyword, which is used to define primitively corecursive functions. Since this assertion checks how constants are defined in the background context, it can tell that the proof goal at hand is a coinductive problem. Therefore, if this assertion returns `true`, maybe the special purpose method called `coinduct` would be useful, since it is developed for coinductive problems. The advantage of this assertions is that it can guess if a problem is a coinductive problem or not, even though we did not have that problem at hand when developing the assertion.

Due to the page limit, we expound the further details of the 113 assertions in our accompanying Appendix [Nag20a].

## 7.4 The Task for Machine Learning Algorithms

The task for machine learning algorithms is to predict the name of a promising proof method from the corresponding array of boolean values. Since we often have multiple equivalently suitable methods for a given proof goal, this learning task should be seen as

a multi-output problem: given an array of boolean values machine learning algorithms should return multiple candidate proof methods rather than only one method. Furthermore, this problem should be treated as a regression problem rather than a classification problem, so that users can see numerical estimates about how likely each method is suitable for a given goal.

## 7.5 Conclusion and Related Work

We presented our dataset for proof method recommendation in Isabelle/HOL. Its simple data format allows machine learning practitioners to try out various algorithms to improve the performance of proof method recommendation.

Kaliszyk *et al.* presented HolStep [KCS17], a dataset based on proofs for HOL Light [Har96]. They developed the dataset from a multivariate analysis library [Har13] and the proof of the Kepler conjecture [HAB+17]. They built HolStep for for various tasks, which does not include proof method prediction. While their dataset explicitly describes the text representations of conjectures and dependencies of theorems and constants, our dataset presents only the essential information about proof documents as an array of boolean values.

Blanchette *et al.* mined the Archive of Formal Proofs [KNPT04] and investigated the nature of proof developments, such as the size and complexity of proofs [BHMN15]. Matichuk *et al.* also studied the Archive of Formal Proofs to understand leading indicators of proof size [MMA+15]. Neither of their projects aimed at suggesting how to write proof documents: to the best of our knowledge we are the first to mine a large repository of ITP proofs using hand crafted feature extractors.

Our dataset does not contain information useful to predict what arguments to pass to each method. Previously we developed, `smart_induct` [Nag20e], to address this problem for the `induct` method in Isabelle/HOL, using a domain-specific language for logical feature extraction [Nag19a].

Recently a number of researchers have developed meta-tools that exploit existing proof methods and tactics and brought stronger proof automation to ITPs [GKU17, NK17, NP18, BLR+19, KH17, GWR15]. We hope that our dataset helps them improve the performance of such meta-tools for Isabelle/HOL.

# Chapter 8

# Goal-Oriented Conjecturing

## Publication Details

Yutaka Nagashima and Julian Parsert. Goal-oriented conjecturing for Isabelle/HOL. In *Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings*, pages 225–231, 2018

## Abstract

We present PGT, a Proof Goal Transformer for Isabelle/HOL. Given 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 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.

## 8.1 Introduction

Consider the following two reverse functions defined in literature [NPW02]:

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

How would you prove their equivalence `"itrev xs [] = rev xs"`? Induction comes to mind. However, it turns out that Isabelle's default proof methods, `induct` and `induct_tac`, are unable to handle this proof goal effectively.

Previously, we developed PSL [NK17], a programmable, meta-tool framework for Isabelle/HOL. With PSL one can write the following strategy for induction:

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

PSL's Dynamic keyword creates variations of the induct method by specifying different combinations of promising arguments found in the proof goal and its background proof context. Then, DInd combines these induction methods with the general purpose proof method, auto, and is_solved, which checks if there is any proof goal left after applying auto. As shown in Fig. 8.1a, PSL keeps applying the combination of a specialization of induct method and auto, until either auto discharges all remaining sub-goals or DInd runs out of the variations of induct methods as shown in Fig. 8.1a.

This approach works well only if the resulting sub-goals after applying some induct are easy enough for Isabelle's automated tools (such as auto in DInd) to prove. When proof goals are presented in an automation-unfriendly way, however, it is not enough to set a certain combination of arguments to the induct method. In such cases engineers have to investigate the original goal and come up with auxiliary lemmas, from which they can derive the original goal.

In this paper, we present PGT, a novel design and prototype implementation[1] of a conjecturing tool for Isabelle/HOL. We provide PGT as an extension to PSL to facilitate the seamless integration with other Isabelle sub-tools. Given a proof goal, PGT produces a series of conjectures that might be useful in discharging the original goal, and PSL attempts to identify the right one while searching for a proof of the original goal using those conjectures.

## 8.2 System Description

### 8.2.1 Identifying Valuable Conjectures via Proof Search

To automate conjecturing, we added the new language primitive, Conjecture to PSL. Given a proof goal, Conjecture first produces a series of conjectures that might be useful in proving the original theorem, following the process described in Section 8.2.2. For each conjecture, PGT creates a subgoal_tac method and inserts the conjecture as the premise of the original goal. When applied to "itrev xs [] = rev xs", for example, Conjecture generates the following proof method along with 130 other variations of the subgoal_tac method:

```
apply (subgoal_tac "!!Nil. itrev xs Nil = rev xs @ Nil")
```

where !! stands for the universal quantifier in Isabelle's meta-logic. Namely, Conjecture introduced a variable of name Nil for the constant []. Applying this method to the goal results in the following two new sub-goals:

```
1. (!!Nil. itrev xs Nil = rev xs @ Nil) ==> itrev xs [] = rev xs
```

[1] available at Github https://github.com/data61/PSL/releases/tag/v0.1.1. The example of this paper appears in PSL/PGT/Example.thy.

```
2. !!Nil. itrev xs Nil = rev xs @ Nil
```

`Conjecture` alone cannot determine which conjecture is useful for the original goal. In fact, some of the generated statements are not even true or provable. To discard these non-theorems and to reduce the size of `PSL`'s search space, we combine `Conjecture` with `Fastforce` (corresponding to the `fastforce` method) and `Quickcheck` (corresponding to Isabelle's sub-tool *quickcheck* [Bul12]) sequentially as well as `DInd` as follows:

```
strategy CDInd = Thens [Conjecture, Fastforce, Quickcheck, DInd]
```

Importantly, `fastforce` does not return an intermediate proof goal: it either discharges the first sub-goal completely or fails by returning an empty sequence. Therefore, whenever `fastforce` returns a new proof goal to a sub-goal resulting from `subgoal_tac`, it guarantees that the conjecture inserted as a premise is strong enough for Isabelle to prove the original goal. In our example, the application of `fastforce` to the aforementioned first sub-goal succeeds, changing the remaining sub-goals to the following:

```
1. !!Nil. itrev xs Nil = rev xs @ Nil
```

However, `PSL` still has to deal with many non-theorems: non-theorems are often strong enough to imply the original goal due to the principle of explosion. Therefore, `CDInd` applies `Quickcheck` to discard easily refutable non-theorems. The atomic strategy `Quickcheck` returns the same sub-goal only if Isabelle's sub-tool quickcheck does not find a counter example, but returns an empty sequence otherwise.

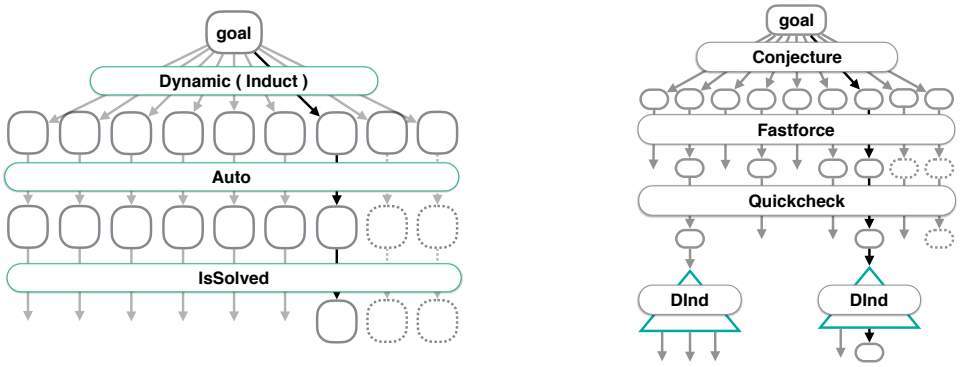
Now we know that the remaining conjectured goals are strong enough to imply the original goal and that they are not easily refutable. Therefore, `CDInd` applies its sub-strategy `DInd` to the remaining sub-goals and it stops its proof search as soon as it finds the following proof script, which will be printed in Isabelle/jEdit's output panel.

```
apply (subgoal_tac "!!Nil. itrev xs Nil = rev xs @ Nil")
apply fastforce apply (induct xs) apply auto done
```

Fig. 8.1b shows how `CDInd` narrows its search space in a top-down manner. Note that `PSL` lets you use other Isabelle sub-tools to prune conjectures. For example, you can use both *nitpick* [Bla10, BN10a] and quickcheck: `Thens [Quickcheck, Nitpick]` in `CDInd`. It also let you combine `DInd` and `CDInd` into one: `Ors [DInd, CDInd]`.

### 8.2.2 Conjecturing

Section 8.2.1 has described how we identify useful conjectures. Now, we will focus on how `PGT` creates conjectures in the first place. `PGT` introduced both automatic conjecturing (`Conjecture`) and automatic generalization (`Generalize`). Since the conjecturing functionality uses generalization, we will only describe the former. We now walk through the




(a) Search tree of `DInd` (b) Search tree of `CDInd`

Figure 8.1: `PSL`'s proof search with/without `PGT`.

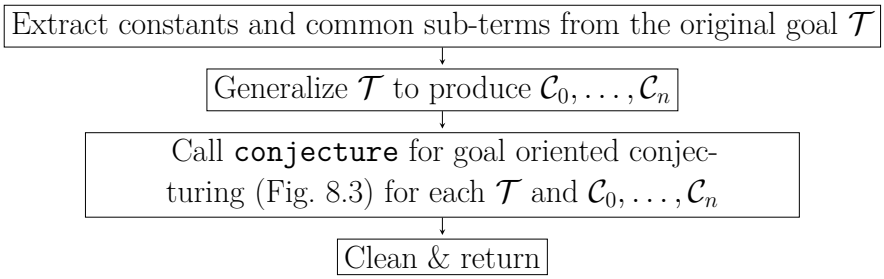



Figure 8.2: The overall workflow of `Conjecture`.

main steps that lead from a user defined goal to a set of potentially *useful* conjectures, as illustrated in Fig. 8.2. We start with the extraction of constants and sub-terms, continue with generalization, goal oriented conjecturing, and finally describe how the resulting terms are sanitized.

**Extraction of Constants and Common Sub-terms.** Given a term representation $\mathcal{T}$ of the original goal, `PGT` extracts the constants and sub-terms that appear multiple times in $\mathcal{T}$. In the example from Section 8.1, `PGT` collects the constants `rev`, `itrev`, and `[]`.

**Generalization.** Now, `PGT` tries to generalize the goal $\mathcal{T}$. Here, `PGT` alone cannot determine over which constant or sub-terms it should generalize $\mathcal{T}$. Hence, it creates a generalized version of $\mathcal{T}$ for each constant and sub-term collected in the previous step. For `[]` in the running example, `PGT` creates the following generalized version of $\mathcal{T}$: `!!Nil. itrev xs Nil = rev xs`.

**Goal Oriented Conjecturing.** This step calls the function `conjecture`, illustrated in Fig. 8.3, with the original goal $\mathcal{T}$ and each of the generalized versions of $\mathcal{T}$ from the previous step $(\mathcal{C}_0, \ldots, \mathcal{C}_n)$. The following code snippet shows part of `conjecture`:

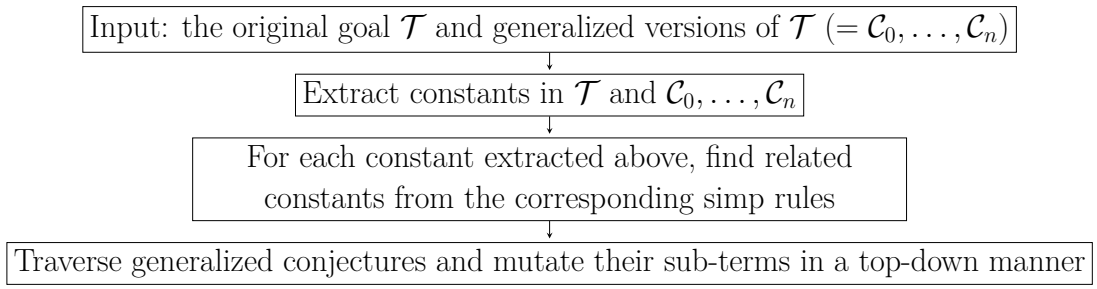



Figure 8.3: The workflow of the `conjecture` function.

```
fun cnjcts t = flat (map (get_cnjct generalisedT t) consts)
fun conj (trm as Abs (_,_,subtrm)) = cnjcts trm @ conj subtrm
 |  conj (trm as App (t1,t2)) = cnjcts trm @ conj t1 @ conj t2
 |  conj trm = cnjcts trm
```

For each $\mathcal{T}$ and $\mathcal{C}_i$ for $0 \leq i \leq n$, `conjecture` first calls `conj`, which traverses the term structure of each $\mathcal{T}$ or $\mathcal{C}_i$ in a top-down manner. In the running example, `PGT` takes some $\mathcal{C}_k$, say `!!Nil. itrev xs Nil = rev xs`, as an input and applies `conj` to it.

For each sub-term the function `get_cnjct` in `cnjcts` creates new conjectures by replacing the sub-term (`t` in `cnjcts`) in $\mathcal{T}$ or $\mathcal{C}_i$ (`generalisedT`) with a new term. This term is generated from the sub-term (`t`) and the constants (`consts`). These are obtained from simplification rules that are automatically derived from the definition of a constant that appears in the corresponding $\mathcal{T}$ or $\mathcal{C}_i$.

In the example, `PGT` first finds the constant `rev` within $\mathcal{C}_k$. Then, `PGT` finds the simp-rule (`rev.simps(2)`) relevant to `rev` which states, `rev (?x # ?xs) = rev ?xs @ [?x]`, in the background context. Since `rev.simps(2)` uses the constant `@`, `PGT` attempts to create new sub-terms using `@` while traversing in the syntax tree of `!!Nil. itrev xs Nil = rev xs` in a top-down manner.

When `conj` reaches the sub-term `rev xs`, `get_cnjct` creates new sub-terms using this sub-term, `@` (an element in `consts`), and the universally quantified variable `Nil`. One of these new sub-terms would be `rev xs @ Nil`[2]. Finally, `get_cnjct` replaces the original sub-term `rev xs` with this new sub-term in $\mathcal{C}_k$, producing the conjecture: `!!Nil. itrev xs Nil = rev xs @ Nil`.

Note that this conjecture is not the only conjecture produced in this step: `PGT`, for example, also produces `!!Nil. itrev xs Nil = Nil @ rev xs`, by replacing `rev xs` with `Nil @ rev xs`, even though this conjecture is a non-theorem. Fig. 8.4 illustrates the sequential application of generalization in the previous paragraph and goal oriented conjecturing described in this paragraph.

**Clean & Return** Most produced conjectures do not even type check. This step removes them as well as duplicates before passing the results to the following sub-strategy (`Then`

[2]Note that `Nil` is a universally quantified variable here.

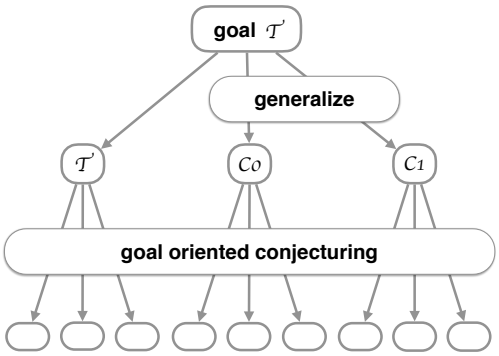



Figure 8.4: PSL's sequential generalization and goal oriented conjecturing.

`[Fastforce, Quickcheck, DInd]` in the example).

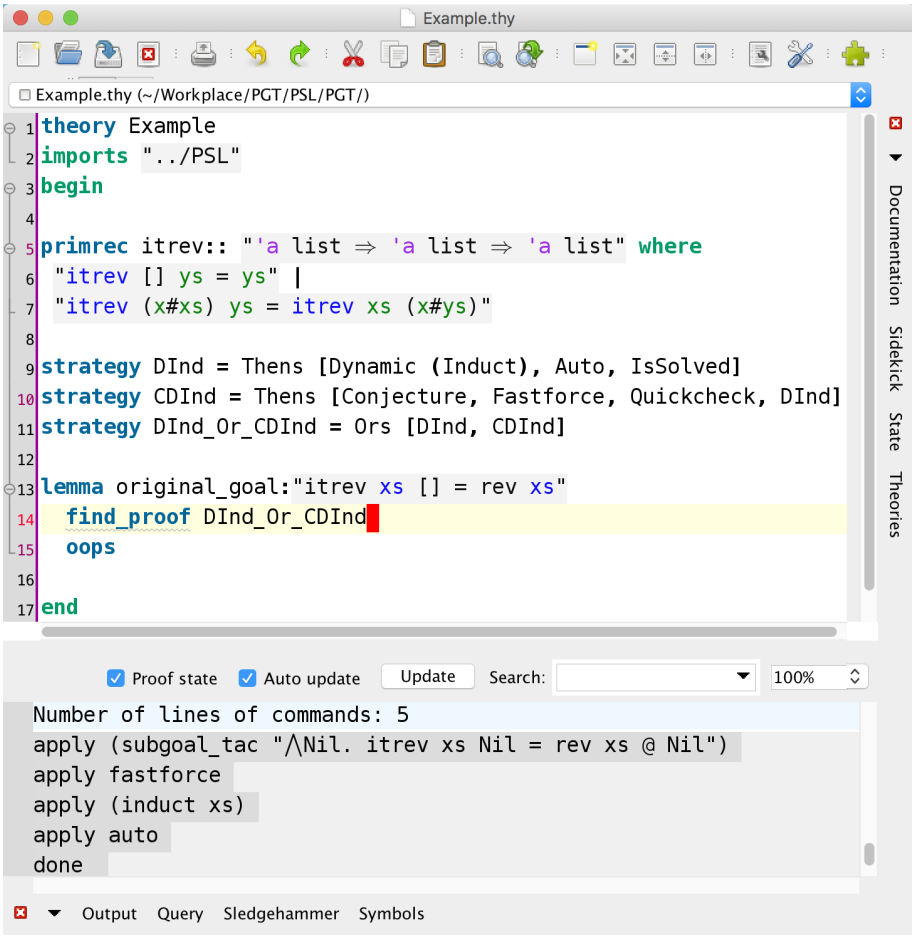
## 8.3 Conclusion

We presented an automatic conjecturing tool PGT and its integration into PSL. Currently, PGT tries to generate conjectures using previously derived simplification rules as hints. We plan to include more heuristics to prioritize conjectures before passing them to subsequent strategies.

Most conjecturing tools for Isabelle, such as *IsaCoSy* [JDB11] and *Hipster* [JRSC14], are based on the bottom-up approach called *theory exploration* [Buc00]. The drawback is that they tend to produce uninteresting conjectures. In the case of IsaCoSy the user is tasked with pruning these by hand. Hipster uses the difficulty of a conjecture's proof to determine or measure its usefulness. Contrary to their approach, PGT produces conjectures by mutating original goals. Even though PGT also produces unusable conjectures internally, the integration with PSL's search framework ensures that PGT only presents conjectures that are indeed useful in proving the original goal. Unlike Hipster, which is based on a Haskell code base, PGT and PSL are an Isabelle theory file, which can easily be imported to any Isabelle theory. Finally, unlike Hipster, PGT is not limited to equational conjectures.

Gauthier *et al.* described conjecturing across proof corpora [GK15]. While PGT creates conjectures by mutating the original goal, Gauthier *et al.* produced conjectures by using statistical analogies extracted from large formal libraries [GKU16].

# Appendix




Screenshot of Isabelle/HOL with PGT.

# Chapter 9

# Conclusion

## 9.1 Summary

This dissertation introduced various artificial intelligence (AI) approaches to assist theorem proving in Isabelle/HOL.

What makes my approaches unique compared to other AI-based theorem proving projects is the deliberate choice of representations for meta-reasoning in Isabelle/HOL.

Chapter 3 introduced the proof strategy language, `PSL`, to encode procedural heuristics as strategies. As discussed in Chapter 8, this language is also used to specify abductive reasoning to identify valuable auxiliary lemmas. The main drawback of `PSL` is that this language cannot assist proof engineers if the interpreter does not complete a proof search.

To complement this weakness, Chapter 6 presented `PaMpeR`, a tool that suggests which proof method to use without completing a proof search. Even though `PaMpeR` relies on a simple machine learning algorithm, known as regression tree construction, our evaluation showed that it produces accurate recommendations for many methods because `PaMpeR` transforms proof databases into a simple data format explained in Chapter 7 before building regression trees. `PaMpeR`, however, does not recommend what arguments to pass to each proof methods.

`smart_induct` in Chapter 5 addressed this problem for the `induct` method: it suggests what arguments to pass to the `induct` method without completing a proof search. For `smart_induct` to produce valuable recommendations across problem domains, the heuristics used in `smart_induct` were encoded in a domain-specific language, `LiFtEr`, introduced in Chapter 4.

## 9.2 Towards a Stronger Automation for Proof by Induction

I conclude this dissertation by identifying some of the remaining challenges to develop a stronger proof automation for inductive theorem prover.

### 9.2.1 From `LiFtEr` to `SeLFiE`

The evaluation results shown in this dissertation proved the overall validity of my approaches. But they also showed weaknesses. In particular, the evaluation of `smart_induct` in Chapter 5 revealed that `smart_induct` tends to be to be too slow for smooth user-experience or to guide a proof search as a part of an automatic prover. Furthermore,

the evaluation also showed that the recommendations of `smart_induct` tend to be unreliable when appropriate inductive proofs involve variable generalisation.

We consider that the poor accuracy of `smart_induct`'s recommendation stems from the limited capacity of its implementation language, `LiFtEr`: heuristics written in `LiFtEr` are not able to analyse semantics of proof goals, even though it is often necessary to analyse how each constant in a proof goal is defined to decide which variables to generalise before applying induction.

What is much needed to improve `smart_induct`'s speed and accuracy is a new domain-specific language (DSL) that satisfies the following three criteria.

- The DSL preserves the domain-agnostic nature of `LiFtEr`.
- The DSL allows experts to encode heuristics that analyse not only syntactic structures of proof goals but also the semantics of relevant constants.
- The DSL's interpreter can quickly analyse a given combination of arguments to the `induct` tactic in terms of a given heuristic.

To satisfy these criteria, I started working on `SeLFiE`, which is a promising candidate to replace `LiFtEr` [Nag20b]. I expect that `SeLFiE`'s semantic-awareness and its fast interpreter improve both the accuracy and speed of `smart_induct`.

### 9.2.2 Towards United Reasoning

Another remaining challenge is that the approaches presented in this dissertation are implemented as separate tools, each of which takes advantage of a different style of reasoning.

- `PSL` in Chapter 3 enhances the deductive reasoning of Isabelle/HOL by allowing human engineers to provide procedural guidance for proof search.
- `PGT` in Chapter 8 extends `PSL` with abductive reasoning: it explicitly produces hypotheses, which upon success serve as auxiliary lemmas to prove the original goal.
- `PaMpeR` in Chapter 6 and `smart_induct` in Chapter 5 enrich Isabelle/HOL with inductive reasoning: they suggest next promising moves based on an available database or human expertise.

Each of them provides unique strength; however, none of them can automatically prove difficult inductive problems all by itself. What is needed beyond this dissertation is the framework to integrate such different styles of reasoning, so that they can complement the weakness of one reasoning style with the strength of other styles. I envision that such framework, which I call *united reasoning* [NAG20f], will realise a stronger automation tool for inductive theorem proving.

# Bibliography


[Ada16] Mark Adams. HOL zero's solutions for pollack-inconsistency. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2016.

[AHC+16] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby C. Murray, Gerwin Klein, and Gernot Heiser. Cogent: Verifying high-assurance file system implementations. In Tom Conte and Yuanyuan Zhou, editors, *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pages 175–188. ACM, 2016.

[BBHI05] Alan Bundy, David A. Basin, Dieter Hutter, and Andrew Ireland. *Rippling - meta-level guidance for mathematical reasoning*, volume 56 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 2005.

[BBP11] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending sledgehammer with SMT solvers. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, pages 116–130, 2011.

[BFOS84] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[BFW15] Jasmin Blanchette, Mathias Fleury, and Daniel Wand. Concrete Semantics with Isabelle/HOL, 2015.

[BHMN15] Jasmin Christian Blanchette, Maximilian P. L. Haslbeck, Daniel Matichuk, and Tobias Nipkow. Mining the archive of formal proofs. In Manfred Kerber, Jacques Carette, Cezary Kaliszyk, Florian Rabe, and Volker Sorge, editors, *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*, volume 9150 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2015.

[BKPU16] Jasmin Blanchette, Cezary Kaliszyk, Lawrence Paulson, and Josef Urban. Hammering towards qed. *Journal of Formalized Reasoning*, 9(1):101–148, 2016.

[Bla10] Jasmin Christian Blanchette. Nitpick: A counterexample generator for isabelle/hol based on the relational model finder kodkod. In Andrei Voronkov, Geoff Sutcliffe, Matthias Baaz, and Christian G. Fermüller, editors, *Short papers for 17th International Conference on Logic for Programming, Artificial intelligence, and Reasoning, LPAR-17-short, Yogyakarta, Indonesia, October 10-15, 2010*, volume 13 of *EPiC Series in Computing*, pages 20–25. EasyChair, 2010.

[BLR+19] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher order logic theorem proving. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, pages 454–463, 2019.

[BM79] Robert S. Boyer and J. Strother Moore. *A computational logic handbook*, volume 23 of *Perspectives in computing*. Academic Press, 1979.

[BN10a] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2010.

[BN10b] Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement day. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2010.

[Bre15] Joachim Breitner. The safety of call arity. *Archive of Formal Proofs*, February 2015. `http://isa-afp.org/entries/Call_Arity.shtml`, Formal proof development.

[BSvH+93] Alan Bundy, Andrew Stevens, Frank van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: A heuristic for guiding inductive proofs. *Artif. Intell.*, 62(2):185–253, 1993.

[Buc00] Bruno Buchberger. Theory exploration with Theorema, 2000.

[BUG20a] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. Tactic learning and proving for the Coq proof assistant. In Elvira Albert and Laura Kovács, editors, *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73 of *EPiC Series in Computing*, pages 138–150. EasyChair, 2020.

[BUG20b] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. The tactician - A seamless, interactive tactic learner and prover for Coq. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings*, volume 12236 of *Lecture Notes in Computer Science*, pages 271–277. Springer, 2020.

[Bul12] Lukas Bulwahn. The new quickcheck for isabelle - random, exhaustive and symbolic testing under one roof. In Chris Hawblitzel and Dale Miller, editors, *Certified Programs and Proofs - Second International Conference, CPP 2012, Kyoto, Japan, December 13-15, 2012. Proceedings*, volume 7679 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2012.

[Bun88] Alan Bundy. The use of explicit plans to guide inductive proofs. In *9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 23-26, 1988, Proceedings*, pages 111–120, 1988.

[Bun01] Alan Bundy. The automation of proof by mathematical induction. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 845–911. Elsevier and MIT Press, 2001.

[Del00] David Delahaye. A tactic language for the system coq. In *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, pages 85–95, 2000.

[DF03] Lucas Dixon and Jacques D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In *Automated Deduction - CADE-19, 19th International Conference on Automated Deduction Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, pages 279–283, 2003.

[dMKA+15] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 378–388, 2015.

[EJP18] Sólrún Halla Einarsdóttir, Moa Johansson, and Johannes Åman Pohjola. Into the infinite - theory exploration for coinduction. In Jacques D. Fleuriot, Dongming Wang, and Jacques Calmet, editors, *Artificial Intelligence and Symbolic Computation - 13th International Conference, AISC 2018, Suzhou, China, September 16-19, 2018, Proceedings*, volume 11110 of *Lecture Notes in Computer Science*, pages 70–86. Springer, 2018.

[Fel20] Bertram Felgenhauer. Implementing the goodstein function in lambda-calculus. *Archive of Formal Proofs*, February 2020. `http://isa-afp.org/entries/Goodstein_Lambda.html`, Formal proof development.

[GK15] Thibault Gauthier and Cezary Kaliszyk. Sharing HOL4 and HOL light proof knowledge. *CoRR*, abs/1509.03527, 2015.

[GKN10] Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. Mizar in a nutshell. *J. Formalized Reasoning*, 3(2):153–245, 2010.

[GKU16] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. Initial experiments with statistical conjecturing over large formal corpora. In Andrea Kohlhase, Paul Libbrecht, Bruce R. Miller, Adam Naumowicz, Walther Neuper, Pedro Quaresma, Frank Wm. Tompa, and Martin Suda, editors, *Joint Proceedings of the FM4M, MathUI, and ThEdu Workshops, Doctoral Program, and Work in Progress at the Conference on Intelligent Computer Mathematics 2016 co-located with the 9th Conference on Intelligent Computer Mathematics (CICM 2016), Bialystok, Poland, July 25-29, 2016*, volume 1785 of *CEUR Workshop Proceedings*, pages 219–228. CEUR-WS.org, 2016.

[GKU17] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. TacticToe: Learning to reason with HOL4 tactics. In Thomas Eiter and David Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC Series in Computing*, pages 125–143. EasyChair, 2017.

[Gon07] Georges Gonthier. The four colour theorem: Engineering of a formal proof. In Deepak Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007.

[Gra05] Bernhard Gramlich. Strategic issues, problems and challenges in inductive theorem proving. *Electr. Notes Theor. Comput. Sci.*, 125(2):5–43, 2005.

[GWR15] Thomas Gransden, Neil Walkinshaw, and Rajeev Raman. SEPIA: search for proofs using inferred automata. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 246–255. Springer, 2015.

[HAB+17] THOMAS HALES, MARK ADAMS, GERTRUD BAUER, TAT DAT DANG, JOHN HARRISON, LE TRUONG HOANG, CEZARY KALISZYK, VICTOR MAGRON, SEAN MCLAUGHLIN, TAT THANG NGUYEN, and et al. A formal proof of the kepler conjecture. *Forum of Mathematics, Pi*, 5:e2, 2017.

[Har96] John Harrison. HOL light: A tutorial introduction. In *Formal Methods in Computer-Aided Design, First International Conference, FMCAD '96, Palo*

*Alto, California, USA, November 6-8, 1996, Proceedings*, pages 265–269, 1996.

[Har13] John Harrison. The HOL light theory of Euclidean Space. *J. Autom. Reasoning*, 50(2):173–190, 2013.

[HG09] Haibo He and Edwardo A. Garcia. Learning from imbalanced data. *IEEE Trans. Knowl. Data Eng.*, 21(9):1263–1284, 2009.

[HKJM13] Jónathan Heras, Ekaterina Komendantskaya, Moa Johansson, and Ewen Maclean. Proof-pattern recognition and lemma discovery in ACL2. In *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, pages 389–406, 2013.

[ISA+16] Geoffrey Irving, Christian Szegedy, Alexander A. Alemi, Niklas Eén, François Chollet, and Josef Urban. Deepmath - deep sequence models for premise selection. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2235–2243, 2016.

[JDB11] Moa Johansson, Lucas Dixon, and Alan Bundy. Conjecture synthesis for inductive theories. *J. Autom. Reasoning*, 47(3):251–289, 2011.

[Joh17] Moa Johansson. Automated theory exploration for interactive theorem proving: - an introduction to the hipster system. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2017.

[Joh19] Moa Johansson. Lemma discovery for induction - A survey. In Cezary Kaliszyk, Edwin C. Brady, Andrea Kohlhase, and Claudio Sacerdoti Coen, editors, *Intelligent Computer Mathematics - 12th International Conference, CICM 2019, Prague, Czech Republic, July 8-12, 2019, Proceedings*, volume 11617 of *Lecture Notes in Computer Science*, pages 125–139. Springer, 2019.

[JPF18] Yaqing Jiang, Petros Papapanagiotou, and Jacques D. Fleuriot. Machine learning for inductive theorem proving. In *Artificial Intelligence and Symbolic Computation - 13th International Conference, AISC 2018, Suzhou, China, September 16-19, 2018, Proceedings*, pages 87–103, 2018.

[JRSC14] Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. Hipster: Integrating theory exploration in a proof assistant. In Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban, editors,

*Intelligent Computer Mathematics - International Conference, CICM 2014, Coimbra, Portugal, July 7-11, 2014. Proceedings*, volume 8543 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2014.

[JWHT13] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning*. Springer-Verlag New York, 2013.

[KAE+10] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

[KBKU13] Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. MaSh: Machine learning for sledgehammer. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2013.

[KCS17] Cezary Kaliszyk, François Chollet, and Christian Szegedy. Holstep: A machine learning dataset for higher-order logic theorem proving. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[KH17] Ekaterina Komendantskaya and Jónathan Heras. Proof mining with dependent types. In *Intelligent Computer Mathematics - 10th International Conference, CICM 2017, Edinburgh, UK, July 17-21, 2017, Proceedings*, pages 303–318, 2017.

[KM97] Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Trans. Software Eng.*, 23(4):203–213, 1997.

[KMM00] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*. Springer, Boston, MA, Norwell, MA, USA, 2000.

[KMNO14] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. Cakeml: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192. ACM, 2014.

[KNPT04] Gerwin Klein, Tobias Nipkow, Larry Paulson, and Rene Thiemann. *The Archive of Formal Proofs*. 2004.

[KV13] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 1–35, 2013.

[Lam09] Peter Lammich. Collections framework. *Archive of Formal Proofs*, November 2009. `http://isa-afp.org/entries/Collections.shtml`, Formal proof development.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[LISK17] Sarah M. Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. In Thomas Eiter and David Sands, editors, *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, volume 46 of *EPiC Series in Computing*, pages 85–105. EasyChair, 2017.

[LN19] Peter Lammich and Tobias Nipkow. Priority search trees. *Archive of Formal Proofs*, June 2019. `http://isa-afp.org/entries/Priority_Search_Trees.html`, Formal proof development.

[LW19a] Peter Lammich and Simon Wimmer. IMP2 - simple program verification in isabelle/hol. *Archive of Formal Proofs*, 2019, 2019.

[LW19b] Peter Lammich and Simon Wimmer. Verifythis 2019 – polished isabelle solutions. *Archive of Formal Proofs*, October 2019. `http://isa-afp.org/entries/VerifyThis2019.html`, Formal proof development.

[MG02] Andrew Martin and Jeremy Gibbons. A monadic interpretation of tactics, 2002.

[MGW96] A. P. Martin, Paul H. B. Gardiner, and Jim Woodcock. A tactic calculus-abridged version. *Formal Asp. Comput.*, 8(4):479–489, 1996.

[MMA+15] Daniel Matichuk, Toby C. Murray, June Andronick, D. Ross Jeffery, Gerwin Klein, and Mark Staples. Empirical study towards a leading indicator for cost of formal software verification. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 722–732. IEEE Computer Society, 2015.

[MMW16] Daniel Matichuk, Toby C. Murray, and Makarius Wenzel. Eisbach: A proof method language for isabelle. *J. Autom. Reasoning*, 56(3):261–282, 2016.

[Moo73] J. Strother Moore. *Computational logic : structure sharing and proof of program properties*. PhD thesis, University of Edinburgh, UK, 1973.

[Moo98] J. Strother Moore. Symbolic simulation: An ACL2 approach. In *Formal Methods in Computer-Aided Design, Second International Conference, FMCAD '98, Palo Alto, California, USA, November 4-6, 1998, Proceedings*, pages 334–350, 1998.

[MP09] Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *J. Applied Logic*, 7(1):41–57, 2009.

[MW13] J Strother Moore and Claus-Peter Wirth. Automation of mathematical induction as part of the history of logic. *CoRR*, abs/1309.6226, 2013.

[MWM14] Daniel Matichuk, Makarius Wenzel, and Toby C. Murray. An isabelle proof method language. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 390–405. Springer, 2014.

[Nag] Yutaka Nagashima. data61/psl.

[Nag16a] Yutaka Nagashima. Evaluation Results., 2016.

[Nag16b] Yutaka Nagashima. Evaluation Tool., 2016.

[Nag16c] Yutaka Nagashima. Keep failed proof attempts in memory. In *Isabelle Workshop 2016*, Nancy, France, aug 2016.

[Nag16d] Yutaka Nagashima. PSL, 2016.

[Nag18] Yutaka Nagashima. Towards machine learning mathematical induction. *CoRR*, abs/1812.04088, 2018.

[Nag19a] Yutaka Nagashima. LiFtEr: Language to encode induction heuristics for Isabelle/HOL. In *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings*, pages 266–287, 2019.

[Nag19b] Yutaka Nagashima. Towards evolutionary theorem proving for Isabelle/HOL. In Manuel López-Ibáñez, Anne Auger, and Thomas Stützle, editors, *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, pages 419–420. ACM, 2019.

[Nag20a] Yutaka Nagashima. Appendix to "Simple Dataset for Proof Method Recommendation in Isabelle/HOL (Dataset Description)", May 2020.

[Nag20b] Yutaka Nagashima. Faster smarter induction in isabelle/hol with selfie. *CoRR*, abs/2009.09215, 2020.

[Nag20c] Yutaka Nagashima. Simple dataset for proof method recommendation in Isabelle/HOL. In Christoph Benzmüller and Bruce R. Miller, editors, *Intelligent Computer Mathematics - 13th International Conference, CICM 2020, Bertinoro, Italy, July 26-31, 2020, Proceedings*, volume 12236 of *Lecture Notes in Computer Science*, pages 297–302. Springer, 2020.

[Nag20d] Yutaka Nagashima. Simple Dataset for Proof Method Recommendation in Isabelle/HOL, May 2020.

[Nag20e] Yutaka Nagashima. Smart induction for Isabelle/HOL (tool paper). In *Proceedings of the 20th Conference on Formal Methods in Computer-Aided Design – FMCAD 2020*, 2020.

[NAG20f] Yutaka NAGASHIMA. Towards united reasoning for automatic induction in Isabelle/HOL. *The 34th Annual Conference of the Japanese Society for Artificial Intelligence*, JSAI2020:3G1ES103–3G1ES103, 2020.

[NH18a] Yutaka Nagashima and Yilun He. PaMpeR: proof method recommendation system for Isabelle/HOL. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 362–372, 2018.

[NH18b] Yutaka Nagashima and Yilun He. Pamper: Proof method recommendation system for isabelle/hol. *CoRR*, abs/1806.07239, 2018.

[Nip10] Tobias Nipkow. List index. *Archive of Formal Proofs*, February 2010. `http://isa-afp.org/entries/List-Index.shtml`, Formal proof development.

[Nip14] Tobias Nipkow. Skew heap. *Archive of Formal Proofs*, August 2014. `http://isa-afp.org/entries/Skew_Heap.shtml`, Formal proof development.

[NK14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014.

[NK17] Yutaka Nagashima and Ramana Kumar. A proof strategy language and proof script generation for Isabelle/HOL. In Leonardo de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 528–545. Springer, 2017.

[NM04a] Toshiaki Nishihara and Yasuhiko Minamide. Depth first search. *Archive of Formal Proofs*, June 2004. `http://isa-afp.org/entries/Depth-First-Search.shtml`, Formal proof development.

[NM04b] Toshiaki Nishihara and Yasuhiko Minamide. Depth first search. *Archive of Formal Proofs*, June 2004. `http://isa-afp.org/entries/Depth-First-Search.html`, Formal proof development.

[NO16] Yutaka Nagashima and Liam O'Connor. Close encounters of the higher kind - emulating constructor classes in standard ML, sep 2016.

[NP18] Yutaka Nagashima and Julian Parsert. Goal-oriented conjecturing for Isabelle/HOL. In *Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings*, pages 225–231, 2018.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - a proof assistant for higher-order logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[OCR+16] Liam O'Connor, Zilin Chen, Christine Rizkallah, Sidney Amani, Japheth Lim, Toby C. Murray, Yutaka Nagashima, Thomas Sewell, and Gerwin Klein. Refinement through restraint: bringing down the cost of verification. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 89–102. ACM, 2016.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, pages 748–752, London, UK, UK, 1992. Springer-Verlag.

[Pau93] Lawrence C. Paulson. The foundation of a generic theorem prover. *CoRR*, cs.LO/9301105, 1993.

[Pau15] Lawrence C. Paulson. A mechanised proof of gödel's incompleteness theorems using nominal isabelle. *J. Autom. Reasoning*, 55(1):1–37, 2015.

[PB10] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska, editors, *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2010.

[PCI+20] Grant O. Passmore, Simon Cruanes, Denis Ignatovich, Dave Aitken, Matt Bray, Elijah Kagan, Kostya Kanishev, Ewen Maclean, and Nicola Mometto. The imandra automated reasoning system (system description). In *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, pages 464–471, 2020.

[PM14] Christine Paulin-Mohring. Introduction to the calculus of inductive constructions. 11 2014.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[Rau19] Martin Rau. Multidimensional binary search trees. *Archive of Formal Proofs*, May 2019. `http://isa-afp.org/entries/KD_Tree.html`, Formal proof development.

[RLN+16] Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby C. Murray, Gabriele Keller, and Gerwin Klein. A framework for the automatic formal verification of refinement from Cogent to C. In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 323–340. Springer, 2016.

[RV19] Giles Reger and Andrei Voronkov. Induction in saturation-based proof search. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 477–494. Springer, 2019.

[Sic16] Salomon Sickert. Linear temporal logic. *Archive of Formal Proofs*, March 2016. `http://isa-afp.org/entries/LTL.shtml`, Formal proof development.

[SN08] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, pages 28–32, 2008.

[Ste11] Christian Sternagel. Efficient mergesort. *Archive of Formal Proofs*, November 2011. `http://isa-afp.org/entries/Efficient-Mergesort.shtml`, Formal proof development.

[TCdt] The Coq development team. The Coq proof assistant.

[TMK+19] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. The verified cakeml compiler backend. *J. Funct. Program.*, 29:e2, 2019.

[Tra13] Dmitriy Traytel. A codatatype of formal languages. *Archive of Formal Proofs*, November 2013. `http://isa-afp.org/entries/Coinductive_Languages.shtml`, Formal proof development.

[Wad85] Philip Wadler. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*, pages 113–128, 1985.

[Wen02] Markus Wenzel. *Isabelle, Isar - a versatile environment for human readable formal proof documents*. PhD thesis, Technical University Munich, Germany, 2002.

[Wen11] Makarius Wenzel. The isabelle/isar reference manual, 2011.

[Wen12] Makarius Wenzel. Isabelle/jEdit - A prover IDE within the PIDE framework. In *Intelligent Computer Mathematics - 11th International Conference*, pages 468–471, 2012.

[WW07] Makarius Wenzel and Burkhart Wolff. Building formal method tools in the isabelle/isar framework. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2007.