


# Smart Induction for Isabelle/HOL (Tool Paper)

Yutaka Nagashima 

CIIRC, Czech Technical University in Prague

University of Innsbruck

Email: yutaka.nagashima@cvut.cz

**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.

## I. INTRODUCTION

Proof by induction lies at the heart of verification of computer programs that involve recursive data-structures, recursion, or iteration [1]. To facilitate proofs by induction, interactive theorem provers, such as Isabelle/HOL [2], Coq [3], and HOL4[4], 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 [1] 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 [1].

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 the `induct` tactic to attack that problem.

## II. PROOF BY INDUCTION IN ISABELLE/HOL

Given the following two simple reverse functions defined in Isabelle/HOL [2], how do you prove their equivalence [5]?

```
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 the `induct` tactic 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 the `induct` tactic 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 the `induct` tactic, 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?
- 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 [6]. Given an inductive problem, PSL produces various combinations of induction arguments for the `induct` tactic 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.

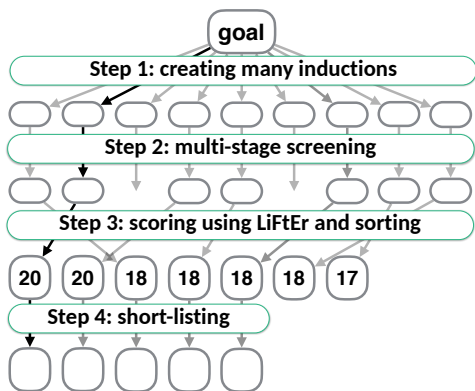


Fig. 1: The workflow of smart\_induct.

In this paper we present smart\_induct, a recommendation tool that addresses these criteria. smart\_induct is available at GitHub [7] together with our running example and the evaluation files discussed in Section IV.

The implementation of smart\_induct is specific to Isabelle/HOL; however, the underlying concept is transferable to other tactic-based proof assistants including HOL4 [4], Coq [3], and Lean [8]. 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 [9] and Imandra [10].

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 the induct tactic across problem domains without completing to a proof search.

### III. GENERATING AND FILTERING TACTICS

Fig. 1 illustrates the internal workflow of smart\_induct: when invoked by a user, the first step produces many variants of the induct tactic 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 [11] 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. 2. In this section, we explore details of Step 1 to Step 3.

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

smart\_induct inspects the given proof goal and produces a number of combinations of arguments for the induct tactic 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 the induct tactic. Then, for each member of the power set smart\_induct computes the permutation of the induction variables since

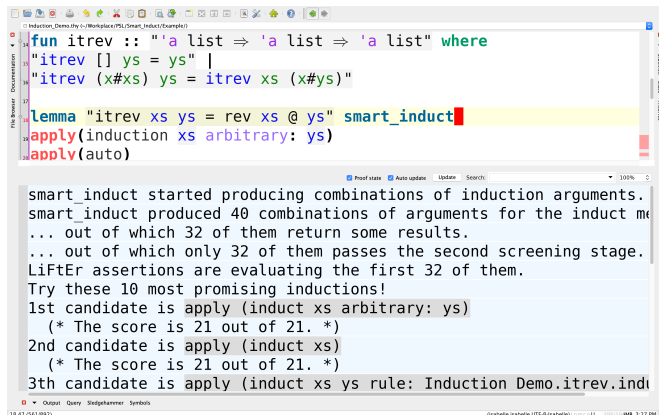


Fig. 2: The user-interface of smart\_induct.

the induct tactic 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.

#### B. 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 the induct tactic itself is not computationally expensive in most cases: we can apply the induct tactic to a proof goal and have intermediate sub-goals at a low cost. Therefore, in Step 2, smart\_induct applies the induct tactic 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 the induct tactics 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 the induct tactic, we also filter out those combinations of arguments, with which the induct tactic 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 the induct tactic 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.

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

Step 3 carefully investigates the remaining candidates using heuristics implemented in `LiFtEr` [11]. `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 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 [11] 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 the `induct` tactic. The domain of terms is the set of all sub-terms appearing in a given goal. The logical connectives ( $\forall$ ,  $\wedge$ ,  $\rightarrow$ , 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 the `induct` tactic.

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 1, 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,

---

### Program 1 A `LiFtEr` heuristic used in `smart_induct`.

---

```

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

```

---

`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 1 is independent of any types or constants specific to `prf2`. Instead of handling specific constructs explicitly, Program 1 analyzes the structure of the goal with respect to the arguments passed to the `induct` tactic 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 the `induct` tactic 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.

### D. User-Interface

Fig. 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 [12], 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 the `induct` tactic to the right location in the document with one click.

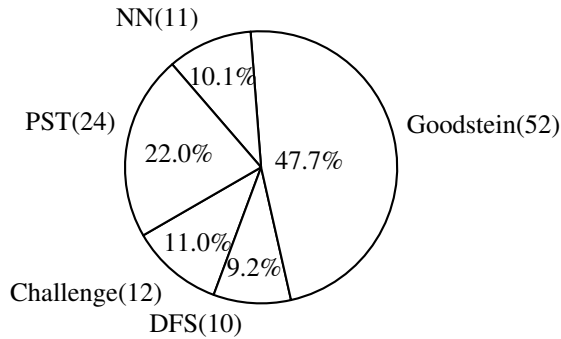


Fig. 3: Breakdown of the evaluation dataset.

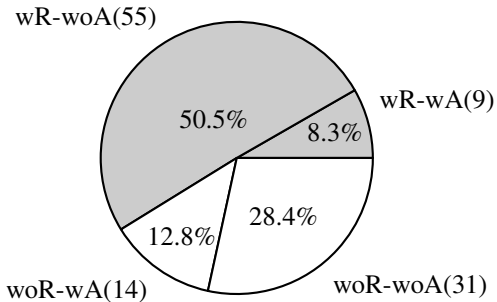


Fig. 4: Use of rule and arbitrary fields.

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.

#### IV. 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.

##### A. 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 [13]. In the following, we use the following short names to denote these files:

- 1) *Challenge* stands for `Challenge1A.thy`, which is a part of the solution for `VerifyThis2019`, a program verification competition associated with `ETAPS2019` [14],
- 2) *DFS* stands for `DFS.thy`, which is a formalisation of depth-first search [15],
- 3) *Goodstein* is for `Goodstein_Lambda.thy`, which is an implementation of the Goodstein function in lambda-calculus [16],
- 4) *NV* stands for `Nearest_Neighbors.thy`, which is from the formalisation of multi-dimensional binary search trees [17], and

TABLE I: 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%)

- 5) *PST* stands for `PST_RBT.thy`, which is from the formalisation of priority search tree [18].

As a whole these files contain 109 calls of the `induct` tactic. Fig. 3 shows the demographics of our dataset. For example, `NN(11)` 10.1% mean that `Nearest_Neighbor.thy` contains 11 invocations of the `induct` tactic, which accounts for 10.1% of all invocations of the `induct` tactic in our dataset.

Fig. 4, on the other hand, shows how often proof authors used the `rule` and `arbitrary` fields. In the labels of Fig. 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 the `induct` tactic 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 the `induct` tactic with an argument in the `rule` field.

This figure illustrates that in our dataset

- more than half of applications come with a rule, and
- applications of the `induct` tactic with a rule are less likely to involve generalisation.

Table I 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 the `induct` tactic 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 the `induct` tactics on compound terms or the `induct` tactics 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 I correspond to the invocations of the `induct` tactic that lie outside the scope of `smart_induct`. And such invocations amount to 7 (6.4%) out of 109.

##### B. 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 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

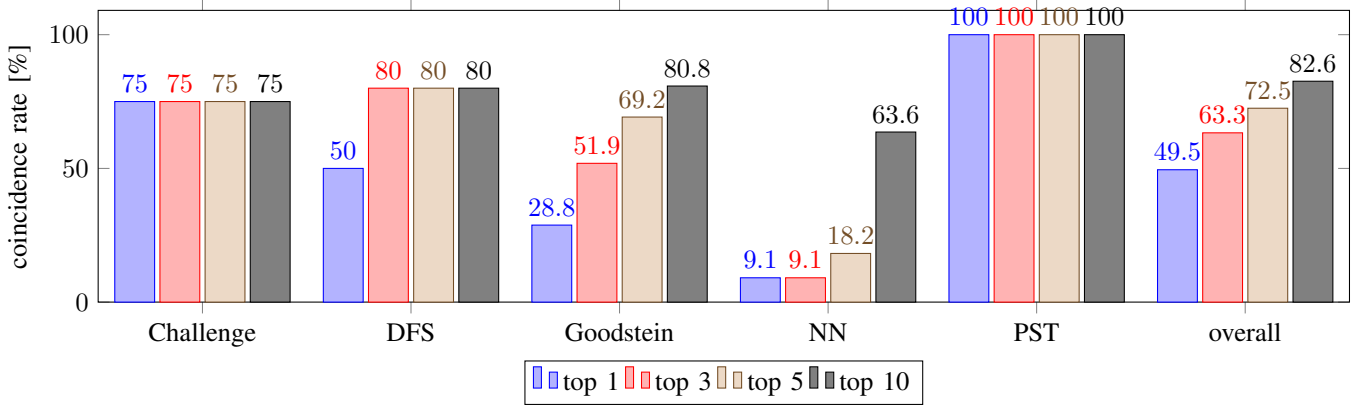


Fig. 5: Coincidence rates for each theory file.

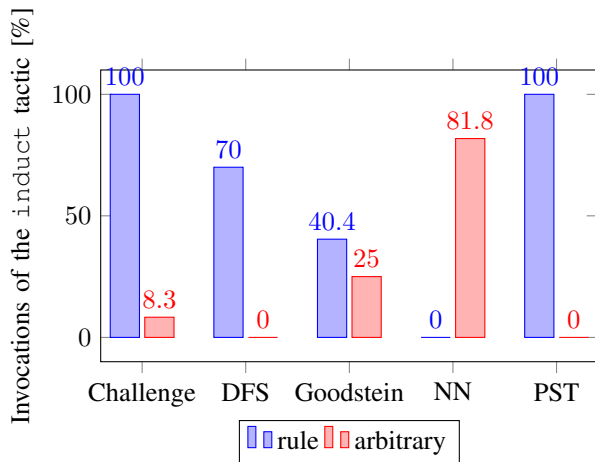


Fig. 6: Inductions with a rule or generalization.

the correctness of these combinations of induction arguments by completing the corresponding proof attempts; however, the necessary proof scripts that follow the `induct` tactic, 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 [13], which accepts Isabelle proof documents only after the peer-reviewing process by Isabelle experts.

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

A quick glance over Fig. 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 the `induct` tactic to develop each theory file in Fig. 6. In this figure each pair of bars presents how often the `induct` tactic 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 the `induct` tactic is applied with an argument in the `rule` field for 40.4% of times in Goodstein, and the `induct` tactic generalises a variable using the `arbitrary` field for 25.0% of times in the same file.

Together with Fig. 5, Fig. 6 shows that `smart_induct`

tends to show a higher coincidence rate for theory files with a high proportion of the `induct` tactics 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 the `induct` tactic involve generalisation while no application of the tactic has an argument in the `rule` field in Fig. 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. 7 where “w” and “wo” stand for “with” and “without”, respectively. For example, the leftmost group labelled with “w-rule-w-arb” represents the coincidence rates among the applications of the `induct` tactic 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 the `induct` tactic when the `induct` tactic has an argument in the `rule` field, which correspond to functional induction and rule inversion.

Interestingly, the two groups in the middle of Fig. 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 the `induct` tactics involving generalisation, `smart_induct` shows even *lower* coincidence rates for rule-based inductions as shown by the two leftmost groups in Fig. 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. 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. 8. In Fig. 8, the coincidence rate among top 1 is still as low as 9.1% since `smart_induct` often chooses 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-arb in Fig 7. The large discrepancies

between the numbers for NN in Fig. 5 and those in Fig. 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.

### C. Pruning.

Section III showed how `smart_induct` produces many candidates of the `induct` tactic 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. 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 the `induct` tactic.

Along the horizontal axis in Fig. 9, we sorted proofs by induction based on the 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.

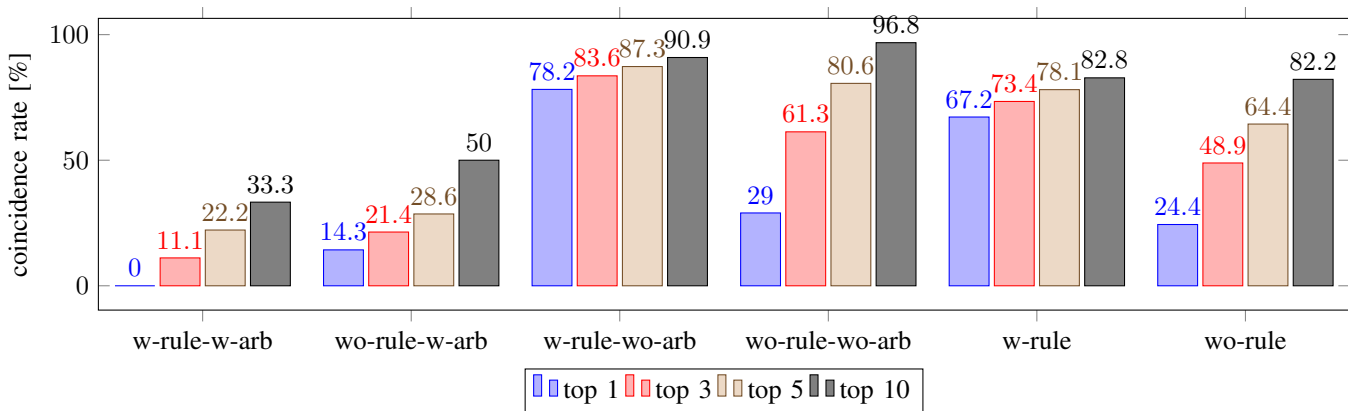


Fig. 7: Coincidence rates with regard to the rule and arbitrary fields.

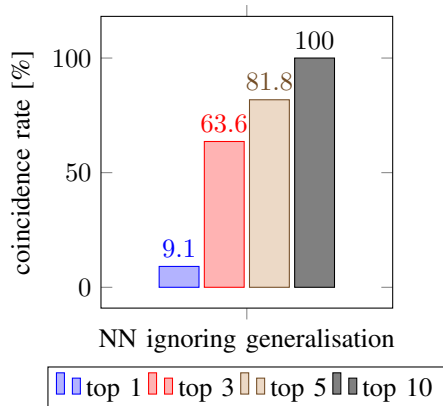


Fig. 8: Coincidence rates when ignoring arbitrary.

On the other hand, the number of candidates after Step 2 are mostly contained under 200 with a single exception of 592.

Fig. 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. 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.

#### D. Execution Time.

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

Fig. 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 IV-C, we filled circles for unsuccessful cases with black.

Similarly to Fig. 9, Fig. 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 IV-C. 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`.

## V. 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 [6], [19] and approaches based on evolutionary computation [20] or statistical machine learning [21].

*Related Work:* The most well-known approach for inductive problems is called the Boyer-Moore waterfall model [22]. This approach was invented for a first-order logic on Common Lisp. `ACL2` [23] 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

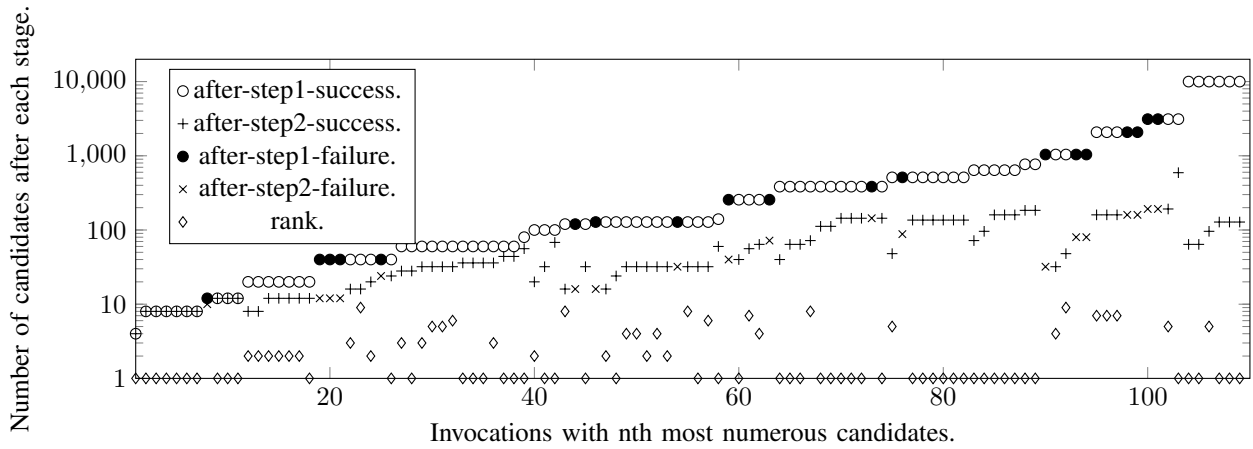


Fig. 9: Number of Candidates After Each Step.

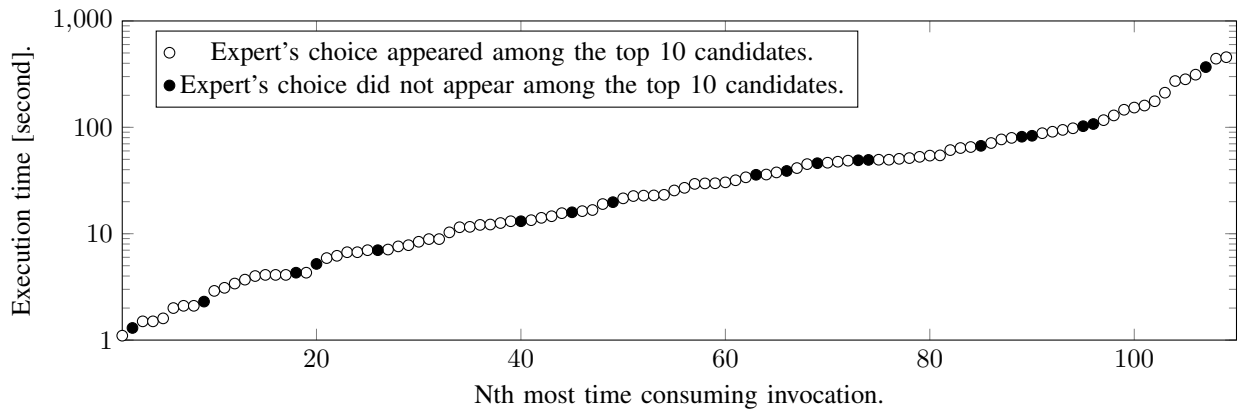


Fig. 10: Execution Time of `smart_induct`.

for and proceeds with the induction scheme with the highest hitting ratio [9], [24].

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 [25] in `HOL Light` [26]. 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 [27], [28], [29], [30], [31], [32] 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 the `induct` tactic effectively.

Sometimes it is not enough to apply the `induct` tactic 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 [33], [34] and top-down approach known as goal-oriented conjecturing [19]. 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` [35], [36]. 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` [37] for saturation-based automated first-order theorem proving [38], 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.

## REFERENCES

- [1] B. Gramlich, “Strategic issues, problems and challenges in inductive theorem proving,” *Electr. Notes Theor. Comput. Sci.*, vol. 125, no. 2, pp. 5–43, 2005. [Online]. Available: <https://doi.org/10.1016/j.entcs.2005.01.006>
- [2] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL - a proof assistant for higher-order logic*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2283.
- [3] The Coq development team, “The Coq proof assistant.” [Online]. Available: <https://coq.inria.fr>
- [4] K. Slind and M. 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*, 2008, pp. 28–32. [Online]. Available: [https://doi.org/10.1007/978-3-540-71067-7\\_6](https://doi.org/10.1007/978-3-540-71067-7_6)
- [5] T. Nipkow and G. Klein, *Concrete Semantics - With Isabelle/HOL*. Springer, 2014. [Online]. Available: <https://doi.org/10.1007/978-3-319-10542-0>
- [6] Y. Nagashima and R. Kumar, “A proof strategy language and proof script generation for Isabelle/HOL,” in *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, ser. Lecture Notes in Computer Science, L. de Moura, Ed., vol. 10395. Springer, 2017, pp. 528–545. [Online]. Available: [https://doi.org/10.1007/978-3-319-63046-5\\_32](https://doi.org/10.1007/978-3-319-63046-5_32)
- [7] Y. Nagashima, “data61/psl.” [Online]. Available: <https://github.com/data61/PSL/releases/tag/0.1.7-alpha>
- [8] L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. 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*, 2015, pp. 378–388. [Online]. Available: [https://doi.org/10.1007/978-3-319-21401-6\\_26](https://doi.org/10.1007/978-3-319-21401-6_26)
- [9] R. S. Boyer and J. S. Moore, *A computational logic handbook*, ser. Perspectives in computing. Academic Press, 1979, vol. 23.
- [10] G. O. Passmore, S. Cruanes, D. Ignatovich, D. Aitken, M. Bray, E. Kagan, K. Kanishev, E. Maclean, and N. Mometto, “The imandra automated reasoning system (system description),” *CoRR*, vol. abs/2004.10263, 2020. [Online]. Available: <https://arxiv.org/abs/2004.10263>
- [11] Y. 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*, 2019, pp. 266–287. [Online]. Available: [https://doi.org/10.1007/978-3-030-34175-6\\_14](https://doi.org/10.1007/978-3-030-34175-6_14)
- [12] M. Wenzel, “The Isabelle/Isar reference manual,” 2011.
- [13] G. Klein, T. Nipkow, L. Paulson, and R. Thiemann, *The Archive of Formal Proofs*, 2004. [Online]. Available: <https://www.isa-afp.org/>
- [14] P. Lammich and S. Wimmer, “Verifythis 2019 – polished isabelle solutions,” *Archive of Formal Proofs*, Oct. 2019, <http://isa-afp.org/entries/VerifyThis2019.html>, Formal proof development.
- [15] T. Nishihara and Y. Minamide, “Depth first search,” *Archive of Formal Proofs*, Jun. 2004, <http://isa-afp.org/entries/Depth-First-Search.html>, Formal proof development.
- [16] B. Felgenhauer, “Implementing the goodstein function in lambda-calculus,” *Archive of Formal Proofs*, Feb. 2020, [http://isa-afp.org/entries/Goodstein\\_Lambda.html](http://isa-afp.org/entries/Goodstein_Lambda.html), Formal proof development.
- [17] M. Rau, “Multidimensional binary search trees,” *Archive of Formal Proofs*, May 2019, [http://isa-afp.org/entries/KD\\_Tree.html](http://isa-afp.org/entries/KD_Tree.html), Formal proof development.
- [18] P. Lammich and T. Nipkow, “Priority search trees,” *Archive of Formal Proofs*, Jun. 2019, [http://isa-afp.org/entries/Priority\\_Search\\_Trees.html](http://isa-afp.org/entries/Priority_Search_Trees.html), Formal proof development.
- [19] Y. Nagashima and J. Parsert, “Goal-oriented conjecturing for Isabelle/HOL,” in *Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings*, 2018, pp. 225–231. [Online]. Available: [https://doi.org/10.1007/978-3-319-96812-4\\_19](https://doi.org/10.1007/978-3-319-96812-4_19)
- [20] Y. Nagashima, “Towards evolutionary theorem proving for Isabelle/HOL,” in *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2019, Prague, Czech Republic, July 13-17, 2019, 2019*, pp. 419–420. [Online]. Available: <https://doi.org/10.1145/3319619.3321921>
- [21] *Towards Machine Learning Mathematical Induction*, 2018. [Online]. Available: <http://arxiv.org/abs/1812.04088>
- [22] J. S. Moore, “Computational logic : structure sharing and proof of program properties,” Ph.D. dissertation, University of Edinburgh, UK, 1973. [Online]. Available: <http://hdl.handle.net/1842/2245>
- [23] *Symbolic Simulation: An ACL2 Approach*, 1998. [Online]. Available: [https://doi.org/10.1007/3-540-49519-3\\_22](https://doi.org/10.1007/3-540-49519-3_22)
- [24] J. S. Moore and C. Wirth, “Automation of mathematical induction as part of the history of logic,” *CoRR*, vol. abs/1309.6226, 2013. [Online]. Available: <http://arxiv.org/abs/1309.6226>
- [25] Y. Jiang, P. Papapanagiotou, and J. 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*, 2018, pp. 87–103. [Online]. Available: [https://doi.org/10.1007/978-3-319-99957-9\\_6](https://doi.org/10.1007/978-3-319-99957-9_6)
- [26] J. 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*, 1996, pp. 265–269. [Online]. Available: <https://doi.org/10.1007/BFb0031814>
- [27] Y. Nagashima and Y. 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, 2018*, pp. 362–372. [Online]. Available: <https://doi.org/10.1145/3238147.3238210>
- [28] Y. Nagashima, “Simple dataset for proof method recommendation in isabelle/hol,” in *Intelligent Computer Mathematics*, C. Benzmüller and B. Miller, Eds. Cham: Springer International Publishing, 2020, pp. 297–302.
- [29] K. Bansal, S. M. Loos, M. N. Rabe, C. Szegedy, and S. 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, 2019*, pp. 454–463. [Online]. Available: <http://proceedings.mlr.press/v97/bansal19a.html>
- [30] T. Gauthier, C. Kaliszzyk, and J. Urban, “TacticToe: Learning to reason with HOL4 tactics,” in *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017*, ser. EPIc Series in Computing, T. Eiter and D. Sands, Eds., vol. 46. EasyChair, 2017, pp. 125–143. [Online]. Available: <http://www.easychair.org/publications/paper/340355>
- [31] L. Blaauwbroek, J. Urban, and H. Geuvers, “Tactic learning and proving for the Coq proof assistant,” in *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, ser. EPIc Series in Computing, E. Albert and L. Kovács, Eds., vol. 73. EasyChair, 2020, pp. 138–150. [Online]. Available: <https://easychair.org/publications/paper/JLdB>
- [32] L. Blaauwbroek et al., “The Tactician,” in *Intelligent Computer Mathematics*, C. Benzmüller and B. Miller, Eds. Cham: Springer International Publishing, 2020, pp. 271–277.
- [33] B. Buchberger, “Theory exploration with theorema,” 2000.
- [34] M. Johansson, D. Rosén, N. Smallbone, and K. Claessen, “Hipster: Integrating theory exploration in a proof assistant,” in *Intelligent Computer Mathematics CICM 2014*.
- [35] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill, “Rippling: A heuristic for guiding inductive proofs,” *Artif. Intell.*, vol. 62, no. 2, pp. 185–253, 1993. [Online]. Available: [https://doi.org/10.1016/0004-3702\(93\)90079-Q](https://doi.org/10.1016/0004-3702(93)90079-Q)
- [36] A. Bundy, D. A. Basin, D. Hutter, and A. Ireland, *Rippling - meta-level guidance for mathematical reasoning*, ser. Cambridge tracts in theoretical computer science. Cambridge University Press, 2005, vol. 56.
- [37] G. Reger and A. Voronkov, “Induction in saturation-based proof search,” in *Automated Deduction - CADE 27 - 27th International*

*Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, ser. Lecture Notes in Computer Science, P. Fontaine, Ed., vol. 11716. Springer, 2019, pp. 477–494. [Online]. Available: [https://doi.org/10.1007/978-3-030-29436-6\\_28](https://doi.org/10.1007/978-3-030-29436-6_28)

[38] L. Kovács and A. Voronkov, “First-order theorem proving and Vampire,”

in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 1–35. [Online]. Available: [https://doi.org/10.1007/978-3-642-39799-8\\_1](https://doi.org/10.1007/978-3-642-39799-8_1)