Faster Smarter Proof by Induction in Isabelle/HOL

Yutaka Nagashima
Yale-NUS College, National University of Singapore
University of Innsbruck
Czech Institute of Informatics, Robotics, and Cybernetics, Czech Technical University in Prague
yutaka@yale-nus.edu.sg

Abstract

We present `sem_ind`, a recommendation tool for proof by induction in Isabelle/HOL. Given an inductive problem, `sem_ind` produces candidate arguments for proof by induction, and selects promising ones using heuristics. Our evaluation based on 1,095 inductive problems from 22 source files shows that `sem_ind` improves the accuracy of recommendation from 20.1% to 38.2% for the most promising candidates within 5.0 seconds of timeout compared to its predecessor while decreasing the median value of execution time from 2.79 seconds to 1.06 seconds.

1 Introduction

As our society grew reliant on software systems, the trustworthiness of such systems became essential. One approach to develop trustworthy systems is complete formal verification using proof assistants. In a complete formal verification, we specify the desired properties of our systems and prove that our implementations are correct in terms of the specifications using software tools, called *proof assistants*.

In many verification projects, proof by induction plays a critical role. To facilitate proof by induction, modern proof assistants offer sub-tools, called *tactic*s. For example, Isabelle [Nipkow *et al.*, 2002] comes with the `induct` tactic. Using the `induct` tactic, human proof authors can apply proof by induction simply by passing appropriate arguments instead of manually developing induction principles. When choosing such arguments, proof engineers have to answer the following three questions:

- On which terms do they apply induction?
- Which variables do they pass to the `arbitrary` field to generalise them?
- Which induction rule do they pass to the `rule` field?

For example, Program 1 defines the append function (`@`) and two reverse functions (`rev1` and `rev2`) and presents two ways to prove their equivalence by applying the `induct` tactic. Note that `[]`, `#`, and `[x]` represent the empty list, the list constructor, and the syntactic sugar for `x # []`, respectively.

Program 1 Equivalence of two reverse functions

```
@ :: α list ⇒ α list ⇒ α list
  []       @ ys = ys
| (x # xs) @ ys = x # (xs @ ys)

rev1 :: α list ⇒ α list
  rev1 []       = []
| rev1 (x # xs) = rev1 xs @ [x]

rev2 :: α list ⇒ α list ⇒ α list
  rev2 []       ys = ys
| rev2 (x # xs) ys = rev2 xs (x # ys)

theorem rev2 xs ys = rev1 xs @ ys
 apply(induct xs ys rule: rev2.induct)
by auto

theorem rev2 xs ys = rev1 xs @ ys
 apply(induct xs arbitrary: ys) by auto
```

The first proof script applies computation induction by passing `rev2.induct` to the `rule` field. `rev2.induct` is a customised induction rule, which Isabelle automatically derives from the definition of `rev2`. The subsequent application of `auto` discharges all sub-goals produced by this induction.

The second proof script applies structural induction on `xs` while generalising `ys`. This application of structural induction results in the following base case and induction step:

```
base case: rev2 [] ys = rev1 [] @ ys
induction step:
 (∀ys. rev2 xs ys = rev1 xs @ ys) ⟶
 rev2 (a # xs) ys = rev1 (a # xs) @ ys
```

where $\forall$ and $\longrightarrow$ represent the universal quantifier and implication, respectively. Using the associative property of `@`, the subsequent application of `auto` firstly transformed the induction step to the following intermediate goal internally:

```
(∀ys. rev2 xs ys = rev1 xs @ ys) ⟶
rev2 xs (a # ys) = rev1 xs @ (a # ys)
```

Since `ys` was generalised in the induction hypothesis, `auto` proved `rev2 xs (a # ys) = rev1 (xs @ (a # ys))` by considering it as a concrete case of the induction hypothesis. If we remove ys from the arbitrary field, the subsequent application of auto leaves the induction step as follows:

```
rev2 xs ys = rev1 xs @ ys ⟶
rev2 xs (a # ys) = rev1 xs @ (a # ys)
```

In other words, auto cannot make use of the induction hypothesis since the conclusion of induction step share the *same* ys. Experienced human researchers can judge that this application of the induct tactic was not appropriate. However, it is also true that this induction step is still provable. For this reason, counter-example finders, such as Nitpick [Blanchette and Nipkow, 2010] and Quickcheck [Bulwahn, 2012], cannot detect that this induct tactic without generalisation is not appropriate for this problem. This is why engineers still have to carefully examine inductive problems to answer the aforementioned three questions when using the induct tactic.

This issue is not specific to Isabelle: other proof assistants, such as Coq [The Coq development team, 2021], HOL4 [Slind and Norrish, 2008], and HOL Light [Harrison, 1996], offer similar tactics for inductive theorem proving, and it is human engineers who have to specify the arguments for such tactics. This issue is not trivial either: in a summary paper from 2005, Gramlich listed generalisation as one of *the main problems and challenges* of inductive theorem proving while predicting that *substantial progress in inductive theorem proving will take time* due to *the enormous problems and the inherent difficulty of inductive theorem proving* [Gramlich, 2005].

Previously, we built smart_induct, which suggests arguments of the induct tactic in Isabelle/HOL. Our evaluation showed that smart_induct predicts on which variables Isabelle experts apply the induct tactic for some inductive problems. Unfortunately, smart_induct has the following limitations:

L1. It tends to take too long to produce recommendations.

L2. It cannot recommend induction on compound terms or induction on multiple occurrences of the same variable.

L3. It is bad at predicting variable generalisations.

L4. Its evaluation is based on a small dataset with 109 inductive problems.

We overcame these problems with sem_ind, a new recommendation tool for the induct tactic. Similarly to smart_induct, sem_ind suggests what arguments to pass to the the induct tactic for a given inductive problem. Our overall contribution is that

> we built a system that predicts how one should apply proof by induction in Isabelle/HOL both quickly and accurately.

Even though we built sem_ind for Isabelle/HOL, our approach is transferable to other proof assistants based on tactics: no matter what proof assistants we use, we need an architecture that aggressively removes less promising candidates to address L1 (presented in Section 2), a procedure to construct promising induction candidates without missing out too many good ones to address L2 (presented in Section 3), and domain-agnostic heuristics that analyse not only the syntactic structures of inductive problems but the definitions of relevant constants to address L3 (presented in Section 4). Finally, Section 5 justifies our claims through extensive evaluations based on 1095 inductive problems, addressing L4.

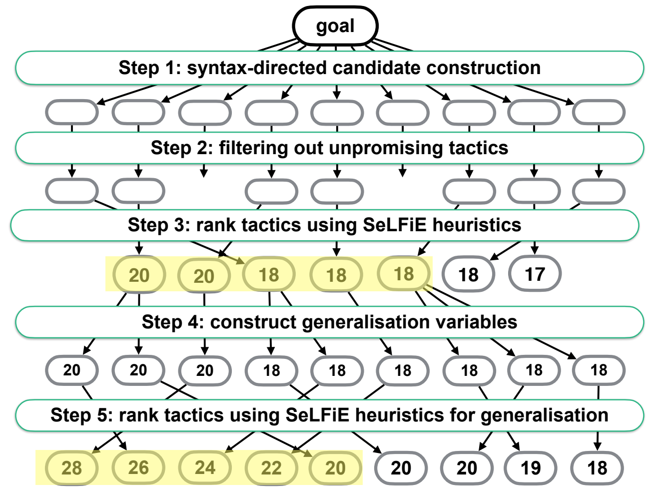

Figure 1: The overview of sem_ind.

2 The Overall Architecture

Figure 1 illustrates the overall architecture of sem_ind, consisting of 5 steps to produce and select candidate tactics as follows.

Step 1. sem_ind produces a set of sequences of induction terms and induction rules for the induct tactic from a given inductive problem. The aim of this step is to produce a small number of candidates intelligently, so that it covers most promising sequences of induction terms and induction rules while avoiding a combinatorial blowup. We expound the algorithm to achieve this goal in Section 3.

Step 2. sem_ind applies the induct tactic with the sequences of arguments produced in Step 1 to discard less promising candidates. sem_ind decides a sequence of arguments is unpromising if the sequence satisfies any of the following conditions:

- the induct tactic fails with an error message, or
- one of the resulting intermediate goals is identical to the original goal itself.

Step 3. sem_ind applies 36 pre-defined heuristics written in SeLFiE [Nagashima, 2020a], which we explain in Section 4 using our running example. These heuristics judge the validity of induction terms and induction rules with respect to the proof goal and the relevant definitions. Each heuristic is implemented as an assertion on inductive problems and arguments of the induct tactic, and each assertion is tagged with a value. If an assertion returns True for a sequence of arguments, sem_ind gives the tagged value to the sequence. sem_ind sums up such points from the 36 heuristics to compute the score for each sequence. Based on these scores,

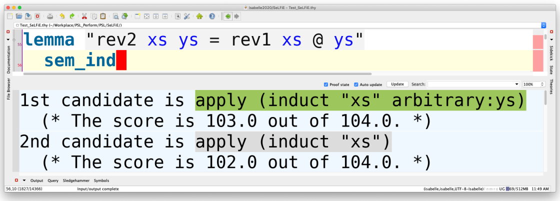

Figure 2: The user-interface of `sem_ind`.

`sem_ind` sorts sequences of arguments from Step 2 and selects the five most promising sequences for further processing.

Step 4. After deciding induction terms and induction rules for the `induct` tactic in Step 3, `sem_ind` adds arguments for the arbitrary field to the sequences of arguments passed from Step 3. Firstly, `sem_ind` collects free variables in the proof goal that are not induction terms for each sequence from Step 3. Then, it constructs the powerset of such free variables and uses each set in the powerset as the arguments to the `arbitrary` field of the `induct` tactic. For example, if `sem_ind` receives `(induct xs)` from Step 3 for our running example of list reversal, it produces `{{}, {ys}}` as the powerset because `xs` and `ys` are the only free variables in the goal and `xs` appears as the induction term. This powerset leads to the following two `induct` tactics: `(induct xs)`, and `(induct xs arbitrary:ys)`.

Step 5. For each remaining sequence, `sem_ind` applies 8 pre-defined SeLFiE heuristics to judge the validity of generalisation. Again, each heuristic is tagged with a value, which is used to compute the final score for each candidate: for each sequence, `sem_ind` adds the score from the generalisation heuristics to the score from Step 3 to decide the final score for each sequence. Based on these final scores, `sem_ind` sorts sequences of arguments from Step 5 and presents the 10 most promising sequences in the Output panel of Isabelle/jEdit, the default proof editor for Isabelle/HOL [Wenzel, 2012].

We developed `sem_ind` entirely within the Isabelle ecosystem without any dependency to external tools. This allows for the easy installation process of `sem_ind`: to use `sem_ind`, users only have to download the relevant Isabelle files from our public GitHub repository[1] and install `sem_ind` using the standard Isabelle command. The seamless integration into the Isabelle proof language, Isar [Wenzel, 2002], lets users invoke `sem_ind` within their ongoing proof development and copy a recommended `induct` tactic to the right location with one click as shown in Figure 2.

3 Syntax-Directed Candidate Construction

In general, the `induct` tactic may take multiple induction terms and induction rules in one invocation. However, it is rarely necessary to pass multiple induction rules to the `rule` filed. Therefore, `sem_ind` passes up-to-one induction rule to the `induct` tactic.

[1]https://github.com/data61/PSL

On the other hand, it is often necessary to pass multiple induction terms to the `induct` tactic, and the order of such induction terms is important to apply the `induct` tactic effectively. Moreover, it is sometimes indispensable to pass the same induction term multiple times to the `induct` tactic, so that each of them corresponds to a distinct occurrence of the same term in the proof goal. What is worse, induction terms do not have to be variables: they can be compound terms such as function applications.

Enumerating all possible sequences of induction terms leads to a combinatorial explosion. To avoid such combinatorial explosion, `sem_ind` produces sequences of induction terms and induction rules taking a syntax-directed approach, which traverses the syntax tree representing the proof goal while collecting plausible sequences of induction terms and rules as follows.

Step 1-A. The collection starts at the root node of the syntax tree with an empty set of sequences of induction arguments.

Step 1-B. If the current node is a function application, `sem_ind` takes arguments to the function, produces a set of lists of such arguments while preserving their order. This set of lists represents candidates for induction terms. If the function in this function application is a constant with a relevant induction rule stored in the proof context, `sem_ind` produces candidate `induct` tactics with and without this rule for the `rule` field. For example, if the current node is `rev2 xs ys`, Step 1-B produces `(induct xs)` and `(induct xs ys rule:rev2.induct)`, as well as other candidates such as `(induct xs ys)` and `(induct xs rule:rev2.induct)`.

Step 1-C. If any sub-terms of the current node is a compound term, `sem_ind` moves down to such sub-terms in the syntax tree and repeats S1-b to collect more candidates for induction arguments.

Step 1-D. `sem_ind` finishes Step 1 when it reaches the leaf nodes in all branches of the syntax tree.

This syntax-directed argument construction avoids a combinatorial explosion at the cost of missing out some effective sequences of induction arguments. One notable example is the omission of simultaneous induction, which is essential to tackle inductive problems with mutually recursive functions. Our evaluation results in Section 5 show that despite the omission of such cases `sem_ind` manages to recommend correct induction arguments for most of the cases that appear in day-to-day theorem proving.

In principle, this smart construction of candidate `induct` tactics ignores handcrafted induction rules: in Step 1-B `sem_ind` collects induction rules derived automatically by Isabelle when defining relevant functions. For example, `sem_ind` picks up `rev2.induct` when seeing `rev2` in our running example since `rev2.induct` is an induction principle Isabelle automatically derived when defining `rev2`. This constraint is unavoidable since we cannot predict what induction rules proof engineers will manually develop in the future for problem domains that may not even exist yet.

The notable exceptions to this principle are induction rules manually developed in Isabelle's standard library: in Step 1-B

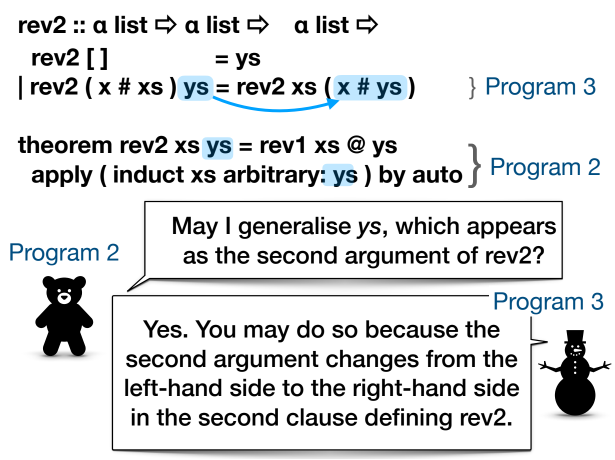

Figure 3: Definition-aware generalisation heuristic as a dialogue.

Program 2 Syntactic analysis for generalisation in SeLFiE

```
∀ arb_term : term ∈ arbitrary_term.
 ∃ f_term : term.
  ∃ f_occ : term_occ ∈ f_term.
   ∃ arb_occ ∈ arb_term.
    ∃ generalise_nth : number.
     is_or_below_nth_argument_of
       (arb_occ, generalise_nth, f_occ)
     ∧
     ∃def
       (f_term,
        generalise_nth_argument_of,
        [generalise_nth, f_term])
```

the smart construction algorithm collects some manually developed induction rules from the standard library if the rules seem to be relevant to the inductive problem at hand. This optimisation is reasonable: some concepts in the standard library, such as lists and natural numbers, are used in many projects and have useful induction rules handcrafted by Isabelle experts.

4 Induction and Generalisation Heuristics

We now have a closer look at heuristics used in Step 3 and Step 5. To produce accurate recommendations quickly, heuristics for `sem_ind` have to satisfy the following two criteria.

C1: The heuristics should be applicable to a wide range of problem domains, some of which do not exist yet.

C2: They should be able to analyse not only the syntactic structures of the inductive problems at hand but also the definitions of relevant constants in terms of how such constants are used within the inductive problems.

To satisfy the above criteria, we choose SeLFiE [Nagashima, 2020a] as our implementation language to encode heuristics. SeLFiE is is a meta-language to encode heuristics for inductive theorem proving as assertions. A SeLFiE

Program 3 Definitional analysis for generalisation in SeLFiE

```
generalise_nth_argument_of :=
λ [generalise_nth, f_term].
 ∃ lhs_occ : term_occ.
  is_left_hand_side (lhs_occ)
 ∧
  ∃ nth_param_on_lhs : term_occ.
    is_nth_argument_of
     (nth_param_on_lhs, generalise_nth,
      lhs_occ)
   ∧
    ∃ nth_param_on_rhs : term_occ.
      ¬ are_of_same_term
       (nth_param_on_rhs, nth_param_on_lhs)
     ∧
      ∃ f_occ_on_rhs : term_occ ∈ f_term.
       is_nth_argument_of
         (nth_param_on_rhs,
          generalise_nth,
          f_occ_on_rhs)
```

assertion takes a pair of arguments to the `induct` tactic and an inductive problem with relevant definitions. The assertion should return `True` if the choice of argument of the `induct` tactic is compatible with the heuristic.

The exact definitions of our 44 heuristics are not informative or possible due to the page limit. Therefore, instead of presenting each heuristic, this section introduces one simple generalisation heuristic written in SeLFiE to demonstrate how we address the above two criteria using SeLFiE.

Program 2 and 3 define the following generalisation heuristic introduced by Nipkow *et al.*[Nipkow *et al.*, 2002][2]:

> (Variable generalisation) should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that need to be quantified are typically those that change in recursive calls.

Figure 3 illustrates how this heuristic justifies the generalisation of `ys` in our running example as an informal dialogue between Program 2 and Program 3. In this dialogue, Program 2 analyses the syntactic structure of the proof goal in terms of the arguments of the `induct` tactic, whereas Program 3 analyses the definition of `rev2` in terms of how `rev2` is used in the goal. Note that Program 2 and Program 3 realise this dialogue through a *definitional quantifier*, $\exists_{def}$. With this dialogue in mind, we now formally interpret the two programs for our running example.

Program 2 checks for all generalised variable, *arb_term*, if there exists a function, *f_term*, its occurrence, *f_occ*, an occurrence of the generalised variable, *arb_occ*, and a natural number, *generalise_nth*, that satisfy the conjunction. Since our running example has only one generalised variable, `ys`, if we choose

[2]In this explanation we simplified the heuristic to focus on the essence of SeLFiE. The corresponding heuristic we used for `sem_ind` involves optimisations and handling of corner cases.

- `rev2` for *f_term*,
- the only occurrence of `rev2` in the proof goal for *f_occ*,
- the occurrence of `ys` on the left-hand side of the equation in the proof goal for *arb_occ*,
- 2 for *generalise_nth*,

we can satisfy the first conjunct for all *arb_term*s because in the proof goal *ys* appears as the second argument of the only occurrence of `rev2`.

In the second conjunct, Program 2 uses $\exists_{def}$ to ask Program 3 if there exists a clause defining `rev2` that satisfies the condition specified in Program 3. Formally speaking, Program 3 examines if there is a term occurrence, *nth_param_on_lhs*, such that *nth_param_on_lhs* is the *generalise_nth* argument on the left-hand side in the equation defining *f_term*, but an occurrence of *f_term* on the right-hand side has a different term for its second argument.

In the second clause defining `rev2`, the second argument of `rev2` is `ys` on the left-hand side, while the second argument of `rev2` is `x#ys` on the right-hand side. Therefore, Program 3 returns `True` to $\exists_{def}$ in Program 2, with which Program 2 confirms that the candidate arguments of the `induct` tactic satisfy Nipkow's heuristic.

Attentive readers may have noticed that Program 2 and Program 3 satisfy the aforementioned two criteria. They satisfy C1 because they refer to problem specific constants and arguments, such as `rev2` and `ys`, abstractly using quantifiers, so that they can be applicable to other inductive problems. They also satisfy C2 because Program 3 analyses the definitions of relevant constants, such as `rev2`, while Program 2 analyses the syntactic structures of the problem. This is why `sem_ind` achieves higher coincidence rates compared to its predecessor, `smart_induct` [Nagashima, 2020b], reported in Section 5.

In total, we implemented 44 heuristics in SeLFiE. 36 of them are induction heuristics and 8 of them are generalisation heuristics. We adopted some heuristics from `smart_induct`, and we newly implemented others based on literature and our expertise. As discussed above, SeLFiE allows us to encode heuristics that transcend problem domains using quantifiers. At the same time, however, some basic concepts, such as lists, sets and natural numbers, appear in a wide range of verification projects. Therefore, we developed 20 SeLFiE heuristics that explicitly refer to concrete constants or manually derived induction rules from the standard library to improve the accuracy of recommendations for problems involving such commonly used concepts. Unlike other parts of this paper, these 20 heuristics involve expertise specific to Isabelle/HOL.

5 Evaluation

We evaluated `sem_ind` against `smart_induct`. Our focus is to measure the accuracy of recommendations and execution time necessary to produce recommendations. All evaluations are conducted on a MacBook Pro (15-inch, 2019) with 2.6 GHz Intel Core i7 6-core memory 32 GB 2400 MHz DDR4.

Unfortunately, it is, in general, not possible to decide whether a given application of the `induct` tactic is right for a given problem. In particular, even if we can finish a proof search after applying the `induct` tactic, this does not guarantee that the arguments passed to the `induct` tactic are a good choice. For example, it is possible to prove our motivating example by applying `(induct ys)`; however, the necessary proof script following this application of the `induct` tactic becomes unnecessarily lengthy.

tool	top 1	top 3	top 5	top 10
`sem_ind`	38.2%	59.3%	64.5%	72.7%
`smart_induct`	20.1%	42.8%	48.5%	55.3%

Table 1: Overall coincidence rates within 5.0 seconds of timeout.

tool	top 1	top 3	top 5	top 10
`sem_ind`	54.5%	63.6%	72.7%	72.7%
`smart_induct`	0.0%	0.0%	0.0%	9.1%

Table 2: Coincidence rates for `Nearest_Neighbors.thy`.

Therefore, we adopt *coincidence rates* as the surrogate for success rates to approximate the accuracy of `sem_ind`'s recommendations: we measure how often recommendations of `sem_ind` *coincide* with the choice of human engineers. Since there are often multiple equally valid sequences of induction arguments for a given inductive problem, we should regard coincidence rates as conservative estimates of true success rates.

As our evaluation target, we use 22 Isabelle theory files with 1,095 applications of the `induct` tactic from the Archive of Formal Proofs (AFP) [Klein *et al.*, 2004]. The AFP is an online repository of formal proofs in Isabelle/HOL. Each entry in the AFP is peer-reviewed by Isabelle experts prior to acceptance, which ensures the quality of our target theory files. Therefore, if `sem_ind` achieves higher coincidence rates for our target theory files, we can say that `sem_ind` produces good recommendations for many problems. To the best of our knowledge, this is the most diverse dataset used to measure recommendation tools for proof by induction. For example, when Nagashima evaluated `smart_induct` they used only 109 invocations of the `induct` tactic from 5 theory files, all of which are included in our dataset.

5.1 Coincidence Rates within 5.0 Seconds

Table 1 shows coincidence rates of both `sem_ind` and `smart_induct` within 5.0 seconds of timeout.

For example, the coincidence rate of `sem_ind` is 38.2% for *top 1*. This means that the sequences of induction arguments used by human researchers appear as the most promising sequences recommended by `sem_ind` for 38.2% of the uses of the `induct` tactic. On the other hand, the coincidence rate of `smart_induct` is 20.1% for *top 1*. This means that `sem_ind` achieved a 90.0% increase of the coincidence rate for the most promising candidates. Overall, Table 1 indicates that `sem_ind` consistently outperforms `smart_induct` when they can suggest multiple sequences.

We leave the coincidence rates for each theory file in the accompanying technical appendix [Nagashima, 2021] but

tool	0.2s	0.5s	1.0s	2.0s	5.0s
`sem_ind`	8.8%	24.7%	47.8%	69.8%	86.8%
`smart`	0.0%	3.5%	16.9%	38.3%	70.2%

Table 3: Return rates for five timeouts.

present coincidence rates for a representative theory file in Table 2. This theory file contains 11 proofs by induction, many of which involve generalisation. Previously, we reported low coincidence rates of `smart_induct` for this file and concluded that we could not achieve higher rates because of the domain-specific language we used to encode heuristics [Nagashima, 2020b]: this language, called `LiFtEr` [Nagashima, 2019a], did not allow us to analyse definitions of relevant constants, even though such definitions often carry the essential information to decide what variables to generalise.

As shown in Table 2, our evaluation confirmed low coincidence rates of `smart_induct` for this file but showed significantly higher rates of `sem_ind`. That is, `sem_ind` managed to predict experts' use of generalisation accurately since `sem_ind` uses SeLFiE to analyse the definitions of relevant constants as shown in Section 4.

5.2 Return Rates for 5 Timeouts

`sem_ind` achieves the higher accuracy by analysing not only the syntactic structures of inductive problems but also the definitions of constants relevant to the problems. Inevitably, this requires larger computational resources: the SeLFiE interpreter has to examine not only the syntax tree representing proof goals but also the syntax trees representing the definitions of relevant constants. However, thanks to the syntax-directed candidate construction algorithm presented in Section 3 and aggressive pruning strategy presented in Section 2, `sem_ind` provides recommendations faster than `smart_induct` does.

This performance improvement is presented in Table 3, which shows how often `sem_ind` and `smart_induct` return recommendations within certain timeouts. For example, the return rate of `sem_ind` is 8.8% for 0.2 seconds. This means that `sem_ind` returns recommendations for 8.8% of proofs by induction within 0.2 seconds. On the other hand, the return rate of `smart_induct` is 0.0% for 0.2 seconds.

Table 3 shows that for all theory files `sem_ind` produces more recommendations than `smart_induct` does for all timeouts specified in this evaluation, proving the superiority of `sem_ind` over `smart_induct` in terms of the execution time necessary to produce recommendations. In fact, the median values of execution time for these 1,095 problems are 1.06 seconds for `sem_ind` and 2.79 seconds for `smart_induct`. That is to say, `sem_ind` achieved 62% of reduction in the median value of execution time.

6 Related Work

Boyer and Moore invented the waterfall model [Moore, 1973] for inductive theorem proving for a first-order logic on Common Lisp [Jr., 1982]. In the original waterfall model, a prover tries to apply any of the following six techniques: simplification, destructor elimination, cross-fertilization, generalisation, elimination of irrelevance, and induction to emerging sub-goals until it solves all sub-goals.

The most well-known prover based on the waterfall model is ACL2 [Moore, 1998]. To decide how to apply induction, ACL2 computes a score, called *hitting ratio*, based on a fixed formula [Boyer and Moore, 1979; Moore and Wirth, 2013] to estimate how good each induction scheme is. Instead of computing a hitting ratio, we use SeLFiE to encode our induction heuristics as assertions. While ACL2 produces many induction schemes and computes the corresponding hitting ratios, `sem_ind` produces a small number of promising sequences of induction terms and rules.

For Isabelle/HOL, we developed a proof strategy language, PSL [Nagashima and Kumar, 2017]. PSL's interpreter discharges easy induction problems by conducting expensive proof searches, and its extension to abductive reasoning tries to identify auxiliary lemmas useful to prove inductive problems [Nagashima and Parsert, 2018]. While our abductive reasoning mechanism took a top-down approach, Johansson *et al.* took a bottom-up approach [Johansson *et al.*, 2014] based on the idea of theory exploration.

There are ongoing attempts to extend saturation-based superposition provers with induction: Cruanes presented an extension of typed superposition that can perform structural induction [Cruanes, 2017], while Reger *et al.* incorporated lightweight automated induction [Reger and Voronkov, 2019] to the Vampire prover [Kovács and Voronkov, 2013] and Hajdú *et al.* extended it to cover induction with generalisation [Hajdú *et al.*, 2020]. A straightforward comparison to their approaches is difficult as their provers are based on less expressive logics and different proof calculi. However, we argue that one advantage of `sem_ind` over their approaches is that `sem_ind` never introduces axioms that risk the consistency of Isabelle/HOL. Furthermore, our evaluation consists of a wider range of problem domains written by experienced Isabelle users based on their diverse interests: Hajdú's evaluation involved a number of inductive problems, but the problem domains were limited to lists, natural numbers, and trees.

Similarly to `sem_ind`, TacticToe [Gauthier *et al.*, 2017; Gauthier *et al.*, 2021] for HOL4, Tactician [Blaauwbroek *et al.*, 2020] for Coq, and PaMpeR [Nagashima and He, 2018] for Isabelle are meta-tactic tools seamlessly integrated in proof assistants' ecosystems; however, none of them logically analyse inductive problems or predict arguments of the `induct` tactic accurately. Unlike these tools, `sem_ind` presents accurate recommendations without relying on statistical machine learning.

Despite the growing interest in deep learning for theorem proving, [Kaliszyk *et al.*, 2017; Bansal *et al.*, 2019; Yang and Deng, 2019; Jakubuv *et al.*, 2020; Paliwal *et al.*, 2020; Chvalovský, 2019; Sekiyama and Suenaga, 2018; Piotrowski and Urban, 2020; Loos *et al.*, 2017; Li *et al.*, 2021], we mindfully avoided deep learning since we have only a limited number of inductive problems available. Instead of deep learning, we used SeLFiE's quantifiers to encode our heuristics in a domain-agnostic style. To the best of our knowledge, no project based on deep learning has managed to predict arguments to the `induct` tactic accurately.

7 Conclusion

We presented `sem_ind`, a recommendation tool for proof by induction. `sem_ind` constructs candidate `induct` tactics for a given inductive problem while avoiding combinatorial explosion, and it selects promising candidates by filtering out unpromising candidates and scoring remaining ones. To give scores to each remaining candidate, we encoded 36 heuristics in SeLFiE to decide on which terms and with which rules we should apply the `induct` tactic, as well as 8 SeLFiE heuristic to decide which variables to generalise.

Our evaluation based on 1,095 inductive problems from 22 theory files showed that compared to the existing tool, `smart_induct`, `sem_ind` achieves a 90.0% increase of the coincidence rate from 20.1% to 38.2% for the most promising candidate, while achieving a 62.0% decrease of the median value of execution time. In particular, `sem_ind` surpassed the accuracy of the existing tool by a wide margin for inductive problems involving variable generalisation.

Currently, `sem_ind` uses manually specified weights for heuristics. It remains as our future work to optimise such weights using evolutionary computation [Nagashima, 2019b] and to integrate `sem_ind` into a larger AI tool for Isabelle/HOL [Nagashima, 2020c].

Acknowledgements

We thank the anonymous reviewers for useful feedback both at TACAS2020 and IJCAI2021. This work has been supported by the following grants:

- the grant of Singapore NRF National Satellite of Excellence in Trustworthy Software Systems (NSoE-TSS),
- the European Regional Development Fund under the project AI & Reasoning (reg.no.CZ.02.1.01/0.0/0.0/15_003/0000466),
- the ERC starting grant no. 714034, and
- NII under NII-Internship Program 2019-2nd call.

References

[Bansal *et al.*, 2019] Kshitij Bansal, Sarah Loos, Markus 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*, 2019.

[Blaauwbroek *et al.*, 2020] Lasse Blaauwbroek, Josef Urban, and Herman 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*, 2020.

[Blanchette and Nipkow, 2010] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *Interactive Theorem Proving, First International Conference, ITP Edinburgh*, 2010.

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

[Bulwahn, 2012] Lukas Bulwahn. The new quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. In *Certified Programs and Proofs - Second International Conference, CPP*, 2012.

[Chvalovský, 2019] Karel Chvalovský. Top-down neural model for formulae. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA*. OpenReview.net, 2019.

[Cruanes, 2017] Simon Cruanes. Superposition with structural induction. In Clare Dixon and Marcelo Finger, editors, *Frontiers of Combining Systems - 11th International Symposium, FroCoS 2017, Brasília*, 2017.

[Gauthier *et al.*, 2017] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. TacticToe: Learning to reason with HOL4 tactics. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun*, 2017.

[Gauthier *et al.*, 2021] Thibault Gauthier, Cezary Kaliszyk, Josef Urban, Ramana Kumar, and Michael Norrish. TacticToe: Learning to prove with tactics. *J. Autom. Reason.*, 65(2):257–286, 2021.

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

[Hajdú *et al.*, 2020] Márton Hajdú, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. Induction with generalization in superposition reasoning. In *Intelligent Computer Mathematics - 13th International Conference, CICM, Bertinoro*. Springer, 2020.

[Harrison, 1996] John Harrison. HOL light: A tutorial introduction. In *Formal Methods in Computer-Aided Design, First International Conference, FMCAD, Palo Alto*, 1996.

[Jakubuv *et al.*, 2020] Jan Jakubuv, Karel Chvalovský, Miroslav Olšák, Bartosz Piotrowski, Martin Suda, and Josef Urban. ENIGMA anonymous: Symbol-independent inference guiding machine (system description). In *Automated Reasoning - 10th International Joint Conference, IJCAR Paris, France*. Springer, 2020.

[Johansson *et al.*, 2014] Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. Hipster: Integrating theory exploration in a proof assistant. In *Intelligent Computer Mathematics - International Conference, CICM Coimbra, Portugal*, Lecture Notes in Computer Science. Springer, 2014.

[Jr., 1982] Guy L. Steele Jr. An overview of common lisp. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming, LFP Pittsburgh*, 1982.

[Kaliszyk *et al.*, 2017] 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, Toulon*. OpenReview.net, 2017.

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

[Kovács and Voronkov, 2013] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In *Computer Aided Verification - 25th International Conference, CAV*, 2013.

[Li *et al.*, 2021] Wenda Li, Lei Yu, Yuhuai Wu, and Lawrence C. Paulson. IsarStep: a benchmark for high-level mathematical reasoning. In *International Conference on Learning Representations*, 2021.

[Loos *et al.*, 2017] Sarah M. Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. In *LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana*, EPiC Series in Computing. EasyChair, 2017.

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

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

[Moore, 1998] J. Strother Moore. Symbolic simulation: An ACL2 approach. In *Formal Methods in Computer-Aided Design, Second International Conference, FMCAD '98, Palo Alto*, 1998.

[Nagashima and He, 2018] 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, Montpellier*, 2018.

[Nagashima and Kumar, 2017] Yutaka Nagashima and Ramana Kumar. A proof strategy language and proof script generation for Isabelle/HOL. In *26th International Conference on Automated Deduction, Gothenburg*, 2017.

[Nagashima and Parsert, 2018] Yutaka Nagashima and Julian Parsert. Goal-oriented conjecturing for Isabelle/HOL. In *Intelligent Computer Mathematics - 11th International Conference, CICM, Hagenberg*, 2018.

[Nagashima, 2019a] Yutaka Nagashima. LiFtEr: Language to encode induction heuristics for Isabelle/HOL. In *Programming Languages and Systems - 17th Asian Symposium, APLAS, Nusa Dua, Bali, Indonesia*, 2019.

[Nagashima, 2019b] Yutaka Nagashima. Towards evolutionary theorem proving for Isabelle/HOL. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO, Prague, Czech Republic*, 2019.

[Nagashima, 2020a] Yutaka Nagashima. SeLFiE: Modular semantic reasoning for induction in Isabelle/HOL. *CoRR*, abs/2010.10296, 2020.

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

[Nagashima, 2020c] Yutaka Nagashima. Towards united reasoning for automatic induction in Isabelle/HOL. In *The Japanese Society for Artificial Intelligence 34th Annual Conference, JSAI, online*, volume https://doi.org/10.11517/pjsai.JSAI2020.0_3G1ES103, 2020.

[Nagashima, 2021] Yutaka Nagashima. Appendix to "faster smarter proof by induction in Isabelle/HOL". *Zenodo, https://doi.org/10.5281/zenodo.4743750*, 2021.

[Nipkow *et al.*, 2002] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - a proof assistant for higher-order logic*. Springer, 2002.

[Paliwal *et al.*, 2020] Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI*. AAAI Press, 2020.

[Piotrowski and Urban, 2020] Bartosz Piotrowski and Josef Urban. Stateful premise selection by recurrent neural networks. In *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain*, EPiC Series in Computing, 2020.

[Reger and Voronkov, 2019] Giles Reger and Andrei Voronkov. Induction in saturation-based proof search. In *CADE - 27th International Conference on Automated Deduction, Natal*, 2019.

[Sekiyama and Suenaga, 2018] Taro Sekiyama and Kohei Suenaga. Automated proof synthesis for the minimal propositional logic with deep neural networks. In *Programming Languages and Systems - 16th Asian Symposium, APLAS, Wellington, New Zealand*. Springer, 2018.

[Slind and Norrish, 2008] Konrad Slind and Michael Norrish. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs Montreal*, 2008.

[The Coq development team, 2021] The Coq development team. The Coq proof assistant, https://coq.inria.fr. https://coq.inria.fr, 2021. Accessed: 2021-05-31.

[Wenzel, 2002] Markus Wenzel. *Isabelle, Isar - a versatile environment for human readable formal proof documents*. PhD thesis, Technical University Munich, Germany, 2002.

[Wenzel, 2012] Makarius Wenzel. Isabelle/jEdit - A prover IDE within the PIDE framework. In *Intelligent Computer Mathematics - 11th International Conference*, 2012.

[Yang and Deng, 2019] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In *Proceedings of the 36th International Conference on Machine Learning*, 2019.