Lemma Discovery for Induction - A survey

Moa Johansson

Department of Computer Science and Engineering, Chalmers University of Technology
moa.johansson@chalmers.se

Abstract. Automating proofs by induction can be challenging, not least because proofs might need auxiliary lemmas, which themselves need to be proved by induction. In this paper we survey various techniques for automating the discovery of such lemmas, including both top-down techniques attempting to generate a lemma from an ongoing proof attempt, as well as bottom-up theory exploration techniques trying to construct interesting lemmas about available functions and datatypes, thus constructing a richer background theory.

1 Introduction

Induction is a proof method often needed to reason about repetition, for instance about recursive datatypes and functions in computer programs. However, automating proofs by induction in a theorem prover can be challenging as inductive proofs often need auxiliary lemmas, including both generalisations of the conjecture at hand as well as discovery of completely new lemmas. The lemmas themselves may also need induction to prove. On a theoretical level, this is captured by the *cut rule* of inference:

$$\frac{\Gamma, \psi \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi}$$

Reasoning backwards, this rule states we can prove the goal ϕ, given Γ, if it is possible to: 1) prove the goal ϕ from Γ with the extra assistance of a lemma ψ, and 2) lemma ψ can be proved from Γ. Note that this potentially introduces an infinite branching point in the search space, as ψ can be any formula. Furthermore, there is a risk that ψ cannot be proved from Γ, if it turns out to be an over-generalisation or otherwise invalid lemma. In some logics, the cut rule is redundant: we say that the logic allows *cut-elimination.* If so, there is no need to worry about having to introduce auxiliary lemmas. However, in logics allowing e.g. structural induction over recursive datatypes, cut-elimination is in general not possible. For these reasons, most major proof assistants such as Isabelle/HOL [37], ACL2 [29] and others, treat induction interactively, the human user takes the main responsibility for how to apply induction as well as inventing and proving any extra lemmas that might be needed. There are however several methods for automating lemma discovery in the context of inductive proofs, which we survey in this paper. For a more general history of the automation of mathematical induction, and a survey of some historical systems that have implemented a variety of methods for induction we refer to [35].

Various techniques for discovering lemmas have been explored since the early days of automating induction in the influential Boyer-Moore prover in the 1970's [3]. Initially, lemma discovery methods focused on generalisations of the conjecture at hand, guided by heuristics [2], while later methods also attempted to construct lemmas of other shapes, using information from failed proofs, introduction of meta-variables and higher-order unification [22]. These methods have in common that they work top-down, trying to derive a lemma from the conjecture we are trying to prove. The search space for top-down methods can be large, especially when more complex lemmas are required. A different approach is to instead proceed by generating lemmas bottom-up, using techniques for *theory exploration*. Here, candidate lemmas are constructed more freely given a set of functions and datatypes, with the intention of creating a richer background theory in which to prove subsequent theorems. This method has been successful in current state-of-the-art automated inductive provers [10].

The remainder of the paper is structured as follows: In Section 2 we survey some standard techniques for lemma discovery by generalisation which have been adapted and implemented in many provers since the early days of automated induction. In Section 3 we discuss primarily *proof critics*, techniques used to analyse failed proof attempts in various ways to come up with elaborate lemmas while avoiding over-generalisations. In Section 4 we then survey several systems for theory exploration, and their performance for automating inductive proofs. Lemma discovery by analysis of large mathematical libraries and machine learning is yet a relatively under-explored area, which we address in Section 5. Recently, there has also been work on integrating induction in first-order and SMT-solvers, which we discuss in Section 6.

Notation. In subsequent examples we will use the notation $x::xs$ for the list cons-operator and the symbol @ to denote list append. We use $t \mapsto s$ to denote simplification of a term t to s.

2 Lemma Discovery by Generalisation

The Boyer-Moore prover was one of the first systems to attempt to automate proofs by induction [3]. The prover was structured according to a *waterfall model* centred around a pool of open subgoals (see Figure 1). First a simplification procedure was applied, and if that did not prove the subgoal additional methods were attempted, e.g. trying to prove the goal using an assumption (such as applying an induction hypothesis) or attempting to generalise the goal. The last step of the waterfall was to apply a new induction, after which the resulting base-case(s) and step case(s) re-entered the waterfall from the beginning. This waterfall structure is still used by the decedents of the original Boyer-Moore prover, such as present day ACL2 [29], and has also been re-implemented in HOL Light [38].

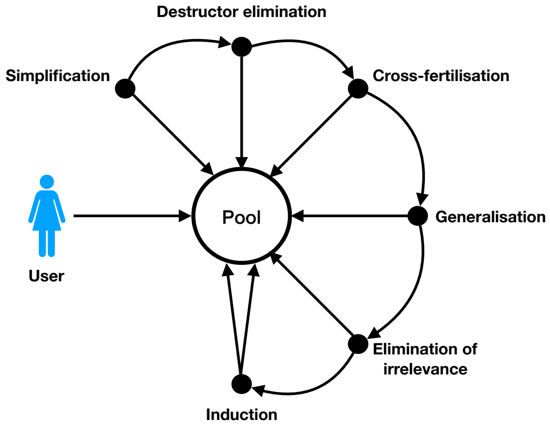

Fig. 1. The Boyer-Moore waterfall model. *Destructor elimination* is a heuristic used to remove so called destructor functions, making implied term structures explicit by replacing variables with terms. *Cross-fertilisation* is concerned with the application of an equational assumption (often an inductive hypothesis) as a rewrite rule. The *Generalisation* step will try to replace sub-terms by variables. The *Elimination of irrelevance* step is also a kind of generalisation which attempts to discard an unnecessary hypothesis.

The generalisation step in the waterfall may suggest candidate lemmas that could be useful for proving the original conjecture using several heuristics for replacement of some sub-term(s) with fresh variables. The main heuristic would pick the minimal non-variable common sub-term, i.e. occurring more than once, for generalisation. Generalisations which introduced a new variable in a position to which induction could potentially be applied were preferred, as such lemmas more likely would be provable by induction. However, deciding which sub-term(s) to generalise can be non-trivial. If the wrong one is picked the result might be an over-generalisation, producing a non-theorem. An overview of generalisation methods for induction can be found in [2], where it is also pointed out that generalisation methods are safest considered in conjunction with counter-example checking, to avoid wasting time proving non-theorems.

Example 1. Consider a small functional program implementing *insertionSort*:

```
sorted [] = True
sorted [x] = True
sorted (x::y::ys) = x <= y /\ sorted(y::ys)

insert x [] = [x]
insert x (y::ys) = if x <= y then (x::y::ys) else y::(insert x ys)
```

```
insertionSort [] = []
insertionSort (x::xs) = insert x (insertionSort xs)
```

Suppose we want to prove that it produces a sorted list: *sorted(insertionSort xs)*. Applying structural induction on xs results in:

$$\text{Base Case: } sorted(insertionSort\ []) \mapsto sorted([]) \mapsto True$$

$$\text{Step Case: } \underbrace{sorted(insertionSort\ xs)}_{ind.\ hyp.} \Longrightarrow \underbrace{sorted(insertionSort(x :: xs))}_{ind.\ concl.}$$

Applying one step of rewriting to the induction conclusion, using the definition of *insertionSort*, results in the new subgoal:

$$sorted(\mathbf{insertionSort\ xs}) \Longrightarrow sorted(insert\ x\ (\mathbf{insertionSort\ xs}))$$

Now, note that the sub-term **insertionSort xs** appear in both the induction hypothesis and in the conclusion. The Boyer-Moore generalisation heuristic would replace this sub-term with a fresh variable, producing the key lemma for proving our original conjecture:

$$sorted\ ys \Longrightarrow sorted(insert\ x\ ys)$$

However, it was observed that the lemma generation heuristics too often produced useless or false conjectures and the Boyer-Moore family of provers instead moved towards becoming interactive, where the user provides the appropriate lemmas. ACL2 still uses the waterfall model from Figure 1 including generalisation techniques, but rely on the user to steer the prover away from unproductive parts of the search space, such as trying to prove an over-generalisation.

Sophisticated generalisation methods have also been implemented in many other systems supporting proofs by induction, such as the INKA system [21]. These include replacement of common sub-terms with new variables as well as replacement of independent sub-terms. Aderhold further enhanced these methods in the VeriFun system [1], an interactive verification system for functional programs. These methods include common sub-term generalisation where induction is applicable to the new variable, a method for renaming variables apart, removing conditions and as well as techniques specific to systems applying destructor-style induction. Moreover, he includes a counter-example checker to filter out any over-generalisations. These methods were demonstrated to be useful in for instance the verification proofs of several sorting algorithms.

Zeno [42] is an automated inductive prover for proving properties about a subset of Haskell programs. It can generate lemmas by the common sub-term technique shown in Example 1, and applies a counter-example finder to avoid over-generalisations.

To summarise, lemma discovery by generalisation can work very well, as we saw in Example 1, but can also be expensive, as it will increase the size of the search space of the prover and may over-generalise and produce false statements. To avoid the prover wasting time trying to prove false conjectures, it can be beneficial to combine generalisation with counter-example checking. For these reasons, it is not always obvious when generalisation should be applied. In [31], experimental evaluation suggested that generalisation was difficult to control and often produced over-generalisations if attempted before applying induction. Better results were obtained if generalisation was deferred until induction had been tried and failed to complete the proof. In the next section, we will survey techniques for lemma discovery which are applied only when no other options remain. We will also consider the discovery of lemmas that cannot be found by the generalisation methods described so far, such as the following:

Example 2. In this example, we consider a proof requiring a lemma with some "extra" term structure introduced, which could not be found just by sub-term generalisation.

```
len [] = 0
len (x::xs) = Suc(len xs)

rotate 0 xs = xs
rotate (Suc n) [] = []
rotate (Suc n) (x::xs) = rotate n (xs @ [x])
```

Consider proving that rotating a list around its length results in the same list as you started with: $\mathit{rotate}\ (\mathit{len}\ \mathit{xs})\ \mathit{xs} = \mathit{xs}$. Applying structural induction on xs results in:

$$\text{Base Case: } \mathit{rotate}\ (\mathit{len}\ [])\ [] = [] \mapsto [] = [] \mapsto \mathit{True}$$

$$\text{Step Case: } \underbrace{\mathit{rotate}\ (\mathit{len}\ \mathit{xs})\ \mathit{xs} = \mathit{xs}}_{\mathit{ind.hyp.}} \Longrightarrow \underbrace{\mathit{rotate}\ (\mathit{len}\ (x :: \mathit{xs}))\ (x :: \mathit{xs}) = (x :: \mathit{xs})}_{\mathit{ind.concl.}}$$

Rewriting to the induction conclusion using the definitions of rotate and len results in:

$$\mathit{rotate}\ (\mathit{len}\ \mathit{xs})\ \mathit{xs} = \mathit{xs} \Longrightarrow \mathit{rotate}\ (\mathit{len}\ \mathit{xs})\ (\mathit{xs}\ @\ [x]) = (x :: \mathit{xs})$$

At this point, common sub-term generalisation will not help us find the missing lemma. What is required is a lemma which both has an extra variable, as well as some additional term-structure compared to our original conjecture:

$$\mathit{rotate}\ (\mathit{len}\ \mathit{ys})\ (\mathit{ys}\ @\ \mathit{zs}) = \mathit{zs}\ @\ \mathit{ys}$$

Our conjecture is a special case of this lemma, with zs happening to be the empty list. Figuring out what the structure of this lemma is a difficult task, and we discuss methods for doing so in sections 3 and 4.

3 Lemma discovery from failed proofs

Proof-planning was introduced by Bundy [6], and motivated by the need to control proof search for inductive proofs in the NuPRL system [13]. The heuristics of the

Boyer-Moore prover were used as inspiration. It was observed that the Boyer-Moore prover could prove a large number of theorems by induction using the same heuristics, but it was never made explicit exactly how and in which order these heuristics had been applied in a particular proof. A proof-plan was thus suggested as an explicit description of a *strategy* to solve a particular family of proofs, such as proofs by induction. Proof-plans could then be composed by *methods*, each a declarative wrapper for a *tactic*[1] including explicit heuristic information about under what condition the tactic should be applied, what effects it would have if successful and so on. Other motivations for proof-plans were the desire to produce human readable proof descriptions and, importantly, to deal with failure of methods by attempting to recover and patch the proof. The idea of *proof-planning critics* was introduced to handle such failures [22]. *Rippling* became an important method in proof planning for guiding rewriting in step-cases of inductive proofs towards being able to apply the inductive hypothesis [7]. Rippling uses annotations (called *wave-fronts*) on the rewrite rules and sub-goals to keep track of which parts match the inductive hypothesis and which ones do not, and steers rewriting towards minimising these differences. The rippling method fails if some crucial lemma is missing. Depending on the which one of the pre-conditions of the rippling method had failed (see chapter 3 of [7] for details), one of several lemma discovery critics would be triggered:

Lemma Calculation. If an inductive proof failed after the inductive hypothesis had been applied, this critic would apply common sub-term generalisation (similar to what was described in Section 2) to the remaining goal, and attempt to prove the resulting conjecture as a lemma. This simple critic often worked well in practice.

Generalisation. The generalisation techniques described in Section 2 could not deal with more complicated generalisations, such as Example 2. Rippling would in these cases fail, as its heuristics would realise that it would not be possible to apply the inductive hypothesis without introducing a generalisation of the conjecture, containing some extra new universally quantified variable (a *sink* in rippling terminology). To figure out where and how such a new variable should be introduced, a schematic lemma, containing some higher-order meta-variables would be constructed. The system would then apply *middle-out reasoning* [20], trying to prove the schematic lemma by induction and expecting to instantiate the meta-variables by subsequent rewrites and the eventual application of the inductive hypothesis. The lemma from Example 2 can be found in this manner, although it requires that some lemmas about the append-functions are already present (see [7], section 3.7.2).

Lemma Speculation. If rippling got stuck before the inductive hypothesis could be applied, the lemma speculation critic would be triggered. Similarly to the generalisation critic, an underspecified lemma would be constructed and applied to the goal: the right-hand side chosen to match some suitable sub-term to rewrite, and the left-hand side consisting of a higher-order meta-variable expected to be instantiated by middle-out reasoning as before. Note

[1] a small program executing one or several proof steps automatically.

that there might be several possible such schematic lemmas, and that higher-order unification is required to complete the instantiation of the lemma which may require a fair bit of search.

These proof-planning critics for rippling were implemented in the OYSTER-CLAM system [8,22]. The original paper on proof critics in CLAM also contained a set of benchmarks of theorems needing inductive proofs and lemmas (including examples 1 and 2). These were later digitalised and made available to developers of inductive theorem provers in the TIP benchmark library [11], and are commonly used to evaluate inductive theorem provers.

An advantage of proof-planning was to clarify exactly when a critic should be triggered, minimising the risk of producing over-generalisations and allowing for more complex lemmas to be automatically synthesised by using middle-out reasoning and higher-order unification. A similar lemma discovery method, based on the instantiation of meta-variables in schematic lemmas was also proposed for the RRL system [28]. Here, the instantiation of the schematic lemma was guided by constraints and not rippling. The lemma calculation and lemma speculation critics were also later implemented in IsaPlanner system [15,16]. However, experimental evaluation of lemma speculation in IsaPlanner was largely negative, the critic was found to be rarely applicable, and the complexities of the many options of how and where to introduce meta-variables could lead to a large search space for higher-order unification [24].

In the context of lemma speculation, we also mention the work by Sonnex on the Elea system [43]. This system takes quite a different approach to automating induction inspired by ideas from functional programming and super-compilation. Elea supports lemma discovery when otherwise stuck by introducing meta-variables and attempting to synthesise functions of a special shape, so called fold-functions, which restricts the search space.

Nagashima and Parsert presented a conjecture generation method for Isabelle/HOL based on generalisation and mutation of stuck subgoals [36]. Here, existing rewrite rules are used to suggest mutations of the goal, and counter-example checking filters out non-theorems. However, no evaluation was provided comparing it to other lemma discovery techniques.

4 Bottom-up lemma discovery: theory exploration

The lemma discovery techniques covered so far have all tried to somehow construct a missing lemma from an ongoing proof attempt of a given conjecture. A different approach is that of *theory exploration*: starting from the bottom up in a new theory given the set of available symbols (such as functions and datatypes) what are the basic, interesting lemmas? In the context of automating induction, the system should try to find and prove as many useful lemmas as possible, and then try to tackle harder proofs in this richer theory, hoping that the key lemmas have already been discovered. The term theory exploration was first coined by Buchberger [4] to describe the workflow of a human mathematician starting a new theory, and how it differs from that of an automated theorem prover, which proves theorems in isolation. Instead, he argues, mathematical software should support an exploratory workflow, by which new concepts are introduced and their relationship to existing concepts explored before proofs of complex theorems are attempted. This is largely how interactive theorem provers are used, and motivated the design of the Theorema system by Buchberger's group [5]. Theorema introduced the concept of *knowledge schemas* representing some interesting mathematical knowledge, and which could be instantiated in new theories, but did not automate the process.

In the following sections we survey theory exploration systems that have been applied to inductive theories. All system have in common that they first generate terms and/or equations followed by some form of evaluation on concrete values, or counter-example checking. Finally, they attempt automated inductive proofs, using previously discovered and proved lemmas if needed. However, they differ in heuristics for how they generate the conjectures, how to evaluated them, and how to judge their interestingness.

4.1 MATHsAiD

MATHsAiD [33], was primarily designed for theory exploration in algebra, but has also been applied to simple inductive theories about natural numbers [32]. MATHsAiD first heuristically constructs a set of potential left-hand sides, called *terms of interest* starting with smaller terms and using some heuristics such as specifically looking for common properties like associativity, commutativity and distributivity. Next, MATHsAiD selects a variable in each term of interest, and instantiates it with some concrete value "TWO" (e.g. for natural numbers, this would be $suc(suc\ 0)$, for lists a list of two elements). It then applies (bounded) forward reasoning from each potential left-hand side term, using function definitions as rewrite rules, until arriving at a different term, also containing an instance of "TWO". At this point, a candidate equation can be constructed by replacing "TWO" by a variable again. MATHsAiD successfully discovered and proved basic lemmas about addition and multiplication in Peano arithmetic, but its wider application to inductive theories was not explored.

4.2 IsaCoSy

IsaCoSy was the first theory exploration system designed specifically for discovering basic inductive lemmas that would be useful in more difficult subsequent proofs [25]. As the name suggests, it was built on top of the Isabelle/HOL proof assistant [37], and used IsaPlanner [15], to prove discovered candidate conjectures. The key idea to restrict the search-space of possible conjectures was to only generate new terms that were irreducible, i.e. terms not possible to reduce further by rewriting using any equations proved so far. IsaCoSy would generate equational terms starting from the minimal size of left- and right-hand sides (single constants, variables) and generate all possible irreducible type-correct terms of that size. Next, the equations would be filtered through a counter-example finder and those surviving passed on to IsaPlanner for proof. IsaCosy then generated constraints from any theorems found to avoid generating any reducible terms in the next iteration when the term size was increased (up to a maximum size given by the user). IsaCosy demonstrated high recall on theories of lists and natural numbers from Isabelle's library. Although precision was lower, one could argue that the lemmas suggested would be reasonable additions to the library. IsaCoSy also performed better at finding lemmas required by difficult inductive proofs than IsaPlanner's lemma speculation critic [24]. On the downside, the runtimes could be very long, primarily due to the non-optimised implementation and the many calls to Isabelle's counter-example finder.

4.3 IsaScheme

IsaScheme was another theory exploration system for Isabelle/HOL [34], using the above-mentioned idea of user-defined schemas from Theorema [5], to generate conjectures and functions, but automating the process of instantiating them. Conjecture schemas would typically capture common patterns such as associativity, distributivity etc. After the instantiation of a schema, IsaScheme would check that the resulting conjecture did not follow trivially from known facts (similar to IsaCoSy's irreducibility heuristic), then pass it on to a counter-example checker and finally for proof using IsaPlanner. IsaScheme furthermore included a Knuth-Bendix completion pass, as lemmas could be discovered in the "wrong" order - a new conjecture might be a generalisation of a previous one (which then was discarded) and which also ensured that the lemmas discovered formed a terminating set of rewrite rules.

4.4 QuickSpec and HipSpec/Hipster

QuickSpec [12,41] was originally designed to be a system for automatically inventing and testing equational specifications of Haskell programs. While QuickSpec itself does not perform any proofs, it is considerably faster at generating conjectures compared to other systems like e.g. IsaCoSy and IsaScheme. QuickSpec does not generate whole equations at once, but rather just terms which would make candidate left- and right hand sides, up to a given maximum size. All terms are initially placed in a single equivalence class. Next, QuickSpec calls Haskell's testing tool QuickCheck [9], to generate random values for all variables in terms (variables were assumed to be shared), followed by evaluation and splitting of the equivalence class(es) accordingly. After many rounds of testing when equivalence classes have stabilised, equations can be extracted. This way, it is never necessary to counter-example check individual equalities, testing and equation-generation is integrated.

The HipSpec system [10], integrated QuickSpec in an inductive theorem prover, and successfully proved most of the difficult theorems from the CLAM-critics benchmark set available in the TIP library [22,11]. HipSpec itself was a rather lightweight prover, it simply applied induction to conjectures and passed the resulting proof obligations to an external automated prover (first-order or SMT-solver). It did however achieve state-of-the-art results by first letting QuickSpec come up with a set of candidate lemmas about the functions occurring in the problem, proving these, and then tackling the main conjectures. This included a fully automatic proof of the rotate-length lemma from Example 2, with all required lemmas found by theory exploration.

Hipster [26,23] is a sister-system to HipSpec, which also use QuickSpec for conjecture generation but conducts formally checked proofs in Isabelle/HOL. As QuickSpec treats the program as a black box, it might re-discover equations representing e.g. function definitions or equations already present in Isabelle's library, which are unnecessary to present to an Isabelle user (but might be interesting if exploring a program for which the source code is not available). To judge which conjectures are likely interesting to a human user, Hipster is therefore parametrised by two tactics: one for *routine reasoning* and one for *hard reasoning*. The idea is that lemmas that are proved by the routine tactic, are somewhat trivial and therefore discarded and not displayed to the user, while lemmas requiring the hard reasoning tactic are judged interesting and returned. Common configurations are to use Isabelle's simplifier (rewriting) or Sledgehammer (a method for calling external first-order automated provers [39]) as routine tactics, and some form of induction as the hard reasoning tactic, but any tactics could be used. Hipster is under ongoing development, and is currently employing the most recent version of QuickSpec [41]. Hipster has added capabilities for conditional lemma discovery which were lacking from HipSpec, as well as support for co-induction [17]. Hipster can thus discover and prove the required lemmas from both Example 1 (insertion sort) and Example 2 (rotate-length).

5 Machine Learning and Lemmas by Analogy

Many proof assistants have large libraries with already formalised mathematics, including many common lemmas. Heras et al. [19] demonstrated a small prototype system for ACL2 where machine learning was used for identifying similarities between a new conjecture and existing ones in the library. From such a similar library fact, one could examine its proof and, if any lemmas had been used, extract a lemma schema from it. Next, a restricted form of theory exploration searched for new lemmas appropriate for the current case. This worked well for examples of the kind of lemmas needed in inductive proofs of the equivalence between recursive functions and their tail-recursive counterparts, but has not been more widely applied or evaluated.

Gauthier et al. also experiment with conjecturing lemmas based on statistical analysis and machine learning from the Mizar mathematical library [18], although they do not consider lemmas for inductive theories and proofs.

Exploring the use of machine learning seems a promising direction for further work. In particular, it could potentially help reducing the search space for theory exploration to specifically target a sub-space where we are more likely to find a lemma which is useful for a particular proof attempt at hand. The theory exploration systems described in Section 4 typically search broadly for lemmas, meaning that they typically also discover and prove a lot of extra things, which might be undesirable if speed is an issue, or if the term size of the lemma required is large.

6 Induction in first-order provers and SMT solvers

More recently, there has been work in integrating induction also in automated superposition based first-order provers [14,44], and SMT-solvers [30,40].

The induction method implemented in the SMT-solver CVC4 employs local theory exploration integrated in the DPLL(T) engine to generate extra lemmas during a proof attempt [40]. The lemma generation module enumerates terms, similarly to QuickSpec, up to a given maximum size. It then heuristically chooses a subset of these candidates (typically around 3) depending on the current context, which will enter the proof search. These heuristic filters include removing reducible terms, similarly to the heuristics used in IsaCoSy, as well as generating ground instances of terms (similar to how QuickSpec used QuickCheck) to detect if any such assignment in the current context falsifies any speculated equations. The performance of CVC4 is comparable to that of other inductive provers mentioned, but it does not quite reach the numbers proved by HipSpec on the proof-critics benchmarks from the TIP library [11,22], which require slightly more difficult lemmas.

Wand developed an extension to superposition calculus to include a type system and induction over datatypes for his PhD thesis [44], which was implemented in the Pirate system. Pirate supports several generalisation techniques for conjecturing lemmas from stuck subgoals, and thus belong to the category of top-down lemma discovery methods. Pirate is reported to perform similarly to HipSpec on the TIP-benchmarks.

Cruanes extended the superposition prover Zipperposition with support for induction [14]. The prover uses an architecture supporting interleaving several inductive proof attempts simultaneously, in the same saturation loop, anticipating that proving lemmas will be needed in many inductive proofs. Once a lemma has been proved, it can be used automatically as a normal axiom. While this system itself only supports some simpler generalisations as a lemma discovery method, it is argued that it could easily be integrated with other techniques such as theory exploration. The capability for interleaving several proof attempts seems as if it could be very useful for theory exploration, where the system often gets a list of conjectures to prove, where some will need others as lemmas.

7 Summary

Lemma discovery is crucial in all but the simplest inductive proofs, and many methods have been implemented. They fall primarily into three groups: Generalisations, proof critics and theory exploration. Variants of generalisation techniques have proved useful in many contexts, in particular those based on replacing (common) sub-terms with new variables, which are easy to implement and often work well. However, there is a risk of over-generalisation so these techniques are safest employed in conjunction with counter-example checking or user interaction. The rippling-based proof critics discussed in Section 3 are today largely obsolete, as most modern inductive provers have abandoned rippling in favour of more general automated rewriting techniques, which do a good enough job without requiring quite as many annotations and heuristics as rippling. Furthermore, theory exploration based techniques can find also the more difficult lemmas which were previously requiring advanced proof critics. Theory exploration has many advantages, it is not dependent on any particular proof technique or prover and it can be run once when a new theory is initiated to provide basic lemmas, after which many harder conjectures from standard benchmark suites are provable. One potential downside is that to find very large lemmas, the search space might also grow rapidly. Theory exploration is also not very good at finding complex conditional lemmas, as it is difficult to generate random ground values which satisfies arbitrary conditions automatically. Furthermore, it can be time consuming to explore functions that have high computational complexity, as evaluation of ground instances then takes a long time. As mentioned in Section 5, one possibility is to exploit existing libraries and machine learning for guiding and restricting lemma discovery.

Strengths and weaknesses:

Generalisation

+ Can quickly and effectively find the right lemma, if correct generalisation found (see Example 1).
- Might over-generalise and produce non-theorems unless coupled with a good counter-example finder, or included in an interactive environment where a human can catch such cases.
- Not always clear when to apply generalisation. Should it be applied before or after attempting induction, and if so, should one defer until after the induction hypothesis has been applied (lemma calculation) or allow more eager generalisation, such as the Boyer-Moore provers?

Proof Critics

+ Clear under which conditions the critic is supposed to be applied.
+ Reduces risk of generating over-generalisations.
+ Can find (some) lemmas that are not generalisations of terms (see Example 2).
- Relies heavily on rippling heuristic, might need work to transfer to other context.
- Some critics rely on middle-out reasoning and higher-order unification which may lead to a rather large search space.

Theory Exploration

+ Bottom-up, can explores theory to find lemmas up-front, not (only) after failed proof attempt. Automatically finds and proves both lemmas required for Examples 1 and 2.
+ Not reliant on any particular proof-technique.

- +/- Relies on evaluation of random ground values for variables in terms. Often fast, but can be computationally heavy if highly complex functions given to the system.
- - So far limited to simple conditional lemmas, as automated generation of random values satisfying arbitrary conditions is a non-trivial problem.
- - Scalability. Can be difficult to search for very large lemmas featuring many different functions as the search space then increase a lot. Term sizes up to 7-9 on each side of an equality is usually OK though.

Finally, we note that automated lemma discovery seems to fit in nicely as a complement to the recent success of *hammers* in interactive theorem proving systems. Such systems, e.g. Sledgehammer [39] for Isabelle and HOLyHammer for HOL Light [27], use machine learning to select a subset of all the available facts in the provers library, and sends them to an external and powerful first-order prover or SMT-solver. If a proof is found, the external prover reports back which lemmas it used and the interactive theorem prover reconstructs it using its internal trusted tactics, without having to re-do all the search. However, in a new theory, the key facts might not yet be there, why lemma discovery techniques could be helpful, especially if the hammer had access to a first-order prover supporting induction, as discussed in Section 6.

References

1. Markus Aderhold. Improvements in formula generalization. In *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction*, LNCS, pages 231–246, Berlin, Heidelberg, 2007. Springer-Verlag.
2. Raymond Aubin. *Mechanizing structural induction.* PhD thesis, University of Edinburgh, 1976.
3. Robert S. Boyer and J S. Moore. *A Computational Logic.* ACM Monographs in Computer Science, 1979.
4. Bruno Buchberger. Theory exploration with Theorema. *Analele Universitatii Din Timisoara, ser. Matematica-Informatica*, 38(2):9–32, 2000.
5. Bruno Buchberger, Adrian Creciun, Tudor Jebelean, Laura Kovacs, Temur Kutsia, Koji Nakagawa, Florina Piroi, Nikolaj Popov, Judit Robu, Markus Rosenkranz, and Wolfgang Windsteiger. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, 4(4):470 – 504, 2006. Towards Computer Aided Mathematics.
6. Alan Bundy. The use of explicit plans to guide inductive proofs. In *Proceedings of CADE*, pages 111–120, 1988.
7. Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: meta-level guidance for mathematical reasoning.* Cambridge University Press, 2005.
8. Alan Bundy, Frank van Harmelen, Christian Horn, and Alan Smail. The OYSTER-CLAM system. In *International Conference on Automated Deduction*, volume 449 of *LNCS*, 1990.
9. Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of ICFP*, pages 268–279, 2000.

10. Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In *Proceedings of the Conference on Auomtated Deduction (CADE)*, volume 7898 of *LNCS*, pages 392–406. Springer, 2013.
11. Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. TIP: Tons of inductive problems. In *Conference on Intelligent Computer Mathematics*, volume 9150 of *LNCS*, pages 333–337. Springer, 2015.
12. Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: Guessing formal specifications using testing. In *Proceedings of TAP*, pages 6–21, 2010.
13. R.L. Constable, S.F. Allen, and H.M. Bromley. *Implementing Mathematics with the nuPRL development system.* Prentice Hall, 1986.
14. Simon Cruanes. Superposition with structural induction. In *Frontiers of Combining Systems, 11th International Symposium.* Springer, 2017.
15. Lucas Dixon and Jacques D. Fleuriot. Higher order rippling in IsaPlanner. In *Proceedings of TPHOLs*, volume 3223 of *LNCS*, pages 83–98, 2004.
16. Lucas Dixon and Moa Johansson. IsaPlanner 2: A proof planner in Isabelle, 2007. DReaM Technical Report (System description).
17. Sólrún Halla Einarsdottir, Moa Johansson, and Johannes Åman Pohjala. Into the infinite – theory exploration for coinduction. In *Conference on Interactive Theorem Proving.* Springer, 2017.
18. Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. Initial experiments with statistical conjecturing over large formal corpora. In *Joint Proceedings of the FM4M, MathUI, and ThEdu Workshops, Doctoral Program, and Work in Progress at the Conference on Intelligent Computer Mathematics 2016 (CICM-WiP 2016)*, volume 1785 of *CEUR*, pages 219–228. CEUR-WS.org, 2016.
19. Jonathan Heras, Ekaterina Komendantskaya, Moa Johansson, and Ewen Maclean. Proof-pattern recognition and lemma discovery in ACL2. In *Proceedings of LPAR*, volume 8312 of *LNCS*, pages 389–406. Springer-Verlag, 2013.
20. Jane Hesketh. *Using middle out reasoning to guide inductive theorem proving.* PhD thesis, University of Edinburgh, 1992.
21. Birgit Hummel. An investigation of formula generalization heuristics for inductive proofs. Interner Bericht Nr, 6/87, Universität Karlsruhe, 1987.
22. Andrew Ireland and Alan Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16:79–111, 1996.
23. Moa Johansson. Automated theory exploration for interactive theorem proving. In *Conference on Interactive Theorem Proving.* Springer, 2017.
24. Moa Johansson, Lucas Dixon, and Alan Bundy. Dynamic rippling, middle-out reasoning and lemma discovery. In S. Siegler and N. Wasser, editors, *Induction, Verification and Termination Analysis: Festschrift for Christoph Walther*, volume 6463 of *LNAI*, pages 102–116. Springer, 2010.
25. Moa Johansson, Lucas Dixon, and Alan Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 47(3):251–289, 2011.
26. Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. Hipster: Integrating theory exploration in a proof assistant. In *Conference on Intelligent Computer Mathematics*, volume 8543 of *LNCS*, pages 108–122. Springer, 2014.
27. Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with flyspeck. *Journal of Automated Reasoning*, 53(2):173–213, 2014.
28. Deepak Kapur and M. Subramaniam. Lemma discovery in automating induction. In M. A. McRobbie and J. K. Slaney, editors, *Automated Deduction — Cade-13*, pages 538–552, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

29. Matt Kaufmann, Manolios Panagiotis, and J Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.
30. K. Rustan M. Leino. Automating induction with an SMT solver. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 315–331, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
31. E. Maclean. Generalisation as a critic. Master's thesis, University of Edinburgh, 1999.
32. Roy McCasland, Alan Bundy, and Serge Autexier. Automated discovery of inductive theorems. In R. Matuszewski and P. Rudnicki, editors, *From Insight to Proof: Festschrift in Honor of A. Trybulec.* 2007.
33. Roy McCasland, Alan Bundy, and Patrick Smith. MATHsAiD: Automated mathematical theory exploration. *Applied Intelligence*, 47(3):585–606, 2017.
34. Omar Montano-Rivas, Roy McCasland, Lucas Dixon, and Alan Bundy. Scheme-based theorem discovery and concept invention. *Expert Systems with Applications*, 39(2):1637–1646, 2012.
35. J Strother Moore and Claus-Peter Wirth. Automation of mathematical induction as part of the history of logic. *IfCoLog Journal of Logics and their Applications*, 4(5), 2014. SEKI-Report SR-2013-02.
36. Yutaka Nagashima and Julian Parsert. Goal-oriented conjecturing for isabelle/hol. In *11th International Conference, CICM 2018, Proceedings*, pages 225–231, 2018.
37. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
38. Petros Papapanagiotou and Jaques Fleuriot. The Boyer-Moore waterfall model revisited. https://arxiv.org/pdf/1808.03810.pdf, 2011.
39. Lawrence C Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. *IWIL-2010*, 2010.
40. Andrew Reynolds and Viktor Kuncak. Induction for SMT solvers. In Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 80–98, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
41. Nicholas Smallbone, Moa Johansson, Claessen Koen, and Maximilian Algehed. Quick specifications for the busy programmer. *Journal of Functional Programming*, 2017.
42. Willam Sonnex, Sophia Drossopoulou, and Susan Eisenbach. Zeno: An automated prover for properties of recursive datatypes. In *Proceedings of TACAS*, pages 407–421. Springer, 2012.
43. William Sonnex. Fixed point promotion: taking the induction out of automated induction. Technical Report UCAM-CL-TR-905, University of Cambridge, Computer Laboratory, March 2017.
44. Daniel Wand. *Superposition: Types and Induction.* PhD thesis, Saarland University, 2017.