Thesis for the Degree of Licentiate of Philosophy

Theory Exploration and Inductive Theorem Proving

Dan Rosén

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
Göteborg, Sweden 2016

Theory Exploration and
Inductive Theorem Proving
Dan Rosén

Technical Report 151L
ISSN 1652-876X
Department of Computer Science and Engineering
Research group: Functional Programming

Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers
Göteborg, Sweden 2016

Abstract

We have built two state-of-the-art inductive theorem provers named *HipSpec* and *Hipster*. The main issue when automating proofs by induction is to discover essential helper lemmas. Our theorem provers use the technique *theory exploration*, which is a method to systematically discover interesting conclusions about a mathematical theory. We use the existing theory exploration system QuickSpec which conjectures properties for a program that seem to hold based on testing. The idea is to try to prove these explored conjectures together with the user-stated goal conjecture. By using this idea and connecting it with our previous work on *Hip*, the Haskell Inductive Prover, we were able to take new leaps in field of inductive theorem proving.

Additionally, we have developed a benchmark suite named *TIP*, short for Tons of Inductive Problems, with benchmark problems for inductive theorem provers, and a tool box for converting and manipulating problems expressed in the TIP format. There were two main reasons to this initiative. Firstly, the inductive theorem proving field lacked a shared benchmark suite as well as a format. Secondly, the benchmarks that have been used were outdated: all contemporary provers would solve almost every problem. We have so far added hundreds of new challenges to the TIP suite to encourage further research.

Contents

Acknowledgements

I would like to thank Koen Claessen for being a great supervisor, bombarding me with crazy ideas which keeps me enthusiastic about our research. I want to thank Moa Johansson for many reasons, such as pushing me to make this text actually resemble a thesis rather than unassorted ramblings. And I thank Jasmin Blanchette for being my discussion leader and the excellent and honest comments on this text. Nick Smallbone deserves to be showered with compliments for being such a kind friend while helping me in a myriad of ways. Irene Lobo Valbuena and Víctor López Juan should receive thanks for their helpful feedback on this text. I want to thank my family and my friends from in and out of the department for their encouragement and company, in particular Malin Ahlberg for her friendship during this time.

Introduction

Here is an example property of a functional program:

$$(1) \quad \forall\, xs \cdot sorted\ (sort\ xs)$$

It specifies the correctness of a sorting function. How can we automatically prove and discover such properties? This is the topic of this thesis.

There are several ways to determine the validity of property (1). It can be tested on some input lists to check that it always returns true. These input lists can be made by hand or generated randomly with, for example, QuickCheck [21]. The absence of a counterexample will give us some confidence that the property holds.

However, to be certain that the property holds for all inputs requires a proof. Since there are infinitely many lists induction is needed to prove (1). Induction gives us a way to mathematically reason about infinite structures. One way to prove (1) could be structural induction on the list *xs*. This can be done using pen and paper, but this is error-prone and tedious. Another alternative would be to work inside a proof assistant such as Isabelle [51], which checks each step of the proof for flawed reasoning. Writing a proof in an assistant, however, can also be tedious and laborious.

Our approach is to instead *automate* proofs by induction. We automate proofs by induction to make producing and maintaining proofs less time-consuming for specialists, as well as enabling non-specialists to use formal methods by lowering the bar to using it. We developed two state-of-the art automated inductive theorem provers: HipSpec and Hipster. They work on functional programs with definitions of recursive functions and algebraic data types, and optionally some property to be proved.

This thesis is a collection of four papers that describe our inductive theorem provers and our initiative to collect and standardise a benchmark suite for inductive theorem provers. The papers are:

1. *Automating Inductive Proofs using Theory Exploration*, describing HipSpec, our prover for Haskell, summarised in Section 3.
2. *Hipster: Integrating Theory Exploration in a Proof Assistant*, describing Hipster, our prover for Isabelle, summarised in Section 4.
3. *TIP: Tons of Inductive Problems*, describing the benchmark suite, summarised in Section 5.
4. *TIP: Tools for Inductive Provers*, describing tools for converting the benchmarks to different formats, summarised in Section 6.

This chapter introduces inductive proving with an example (Section 1) and theory exploration (Section 2). After the sections summarising the papers, this introductory chapter is concluded with related work (Section 7) and future work and conclusions (Section 8).

1 Example proof

As an introductory example, we will look at the *rev–qrev* property about list reversal, a classic example in inductive theorem proving. This is a property about an efficient tail-recursive reversing function *qrev* (q for quick), and an inefficient but straightforward version *rev* using the list append function $+\!\!+$. Their definitions are:

$$
\begin{aligned}
&(+\!\!+) :: [a] \to [a] \to [a] \\
&[\,] \quad\quad +\!\!+\; ys = ys \\
&(x : xs) +\!\!+\; ys = x : (xs +\!\!+\; ys)
\end{aligned}
$$

$$
\begin{aligned}
&rev :: [a] \to [a] \\
&rev\ [\,] \quad\quad = [\,] \\
&rev\ (x : xs) = rev\ xs +\!\!+ [x]
\end{aligned}
$$

$$
\begin{aligned}
&qrev :: [a] \to [a] \to [a] \\
&qrev\ [\,] \quad\quad ys = ys \\
&qrev\ (x : xs)\ ys = qrev\ xs\ (x : ys)
\end{aligned}
$$

The *rev–qrev* property states that the functions produce the same result when the accumulator of *qrev* is initialised with the empty list:

$$(1) \quad \forall\, xs \cdot qrev\ xs\ [\,] = rev\ xs$$

Trying to prove (1) directly by structural induction on *xs* fails. We have to show $qrev\ as\ [a] = rev\ as +\!\!+ [a]$. No progress can be made because the induction hypothesis is $qrev\ as\ [\,] = rev\ as +\!\!+ [\,]$, which does not match the goal.

The way to go forward is to instead prove this generalisation lemma:

$$(2)\quad \forall\, xs\ ys \cdot qrev\ xs\ ys = rev\ xs +\!\!+ ys$$

In induction it is often easier to prove a more general statement like (2) rather than (1) as we get a stronger induction hypothesis. It is stronger since proving (2) by induction on *xs* allows us to requantify over *ys* in the step case. For clarity, the two proof obligations look like this:

- Base case ($xs = [\,]$):

$$\forall\, ys \cdot qrev\ [\,]\ ys = rev\ [\,] +\!\!+ ys$$

- Step case ($xs = a : as$):

$$\begin{aligned} & (\forall\, ys' \cdot qrev\ as\ ys' = rev\ as +\!\!+ ys') && \text{(IH)} \\ \Rightarrow\ & \forall\, ys \cdot qrev\ (a : as)\ ys = rev\ (a : as) +\!\!+ ys \end{aligned}$$

In the step case, we have indicated the induction hypothesis with IH. This is the formula preceding the implication. We call the formula after the arrow the *induction conclusion*. In the hypothesis we may instantiate the second argument of *qrev*, namely ys', freely. This is an important improvement from trying to prove (1), where this second argument to *qrev* had to be the empty list.

The base case is immediate. The step case can be proved as follows by instantiating ys' in the induction hypothesis with $(a : ys)$:

$$\begin{aligned} & qrev\ (a : as)\ ys \\ =\ & qrev\ as\ (a : ys) && \{qrev \text{ definition}\} \\ =\ & rev\ as +\!\!+ a : ys && \{\text{IH with } ys' \mapsto a : ys\} \\ =\ & rev\ as +\!\!+ ([a] +\!\!+ ys) && \{+\!\!+ \text{ definition}\} \\ =\ & (rev\ as +\!\!+ [a]) +\!\!+ ys && \{+\!\!+ \text{ associativity}\} \\ =\ & rev\ (a : as) +\!\!+ ys && \{rev \text{ definition}\} \end{aligned}$$

However, some important steps actually remain. When letting *ys* in (2) be the empty list we do not get (1) as hoped for, but this:

$$(1{*})\quad \forall\, xs \cdot qrev\ xs\ [\,] = rev\ xs +\!\!+ [\,]$$

Obviously another lemma is needed, namely the right identity property $\forall\, xs \cdot xs +\!\!+ [] = xs$. Simple as it may look, this also requires induction to be proved. Furthermore, we overlooked that we needed associativity of append in the proof of (2) above. This is also a lemma since it requires induction to be proved.

Here is a summary of the entire proof of (1) using associativity (a) and right identity (ri). The formulas are implicitly $\forall$-quantified.

$$\begin{array}{lllll}
(ri) & xs +\!\!+ [] & = xs & & \text{by induction on } xs \\
(a) & xs +\!\!+ (ys +\!\!+ zs) & = (xs +\!\!+ ys) +\!\!+ zs & & \text{by induction on } xs \\
(2) & qrev\ xs\ ys & = rev\ xs +\!\!+ ys & & \text{by induction on } xs \\
 & & & & \text{and using } (a) \\
(1) & qrev\ xs\ [] & = rev\ xs & & \text{using } (ri) \text{ and } (2)
\end{array}$$

This example illustrates the crucial dependencies on lemmas which is typical in proofs by induction. Suitable lemmas must be invented. They must be proved by induction themselves, which might in turn require another set of lemmas, and so on. The major difficulty in inductive theorem proving is discovering lemmas. Our theorem provers invent lemmas using a technique called theory exploration, described in the next section.

2 Theory exploration

A theory exploration system is a program that invents new concepts and properties about a problem or theory. One way to see it is as a theorem prover that has no particular goal to prove, but rather wants to create an intuition of interesting truths that can be established from the axioms. In our setting of inductive theorem proving, we do (usually) have a goal property to be proved, but we use theory exploration to find additional properties that hold in the current set of axioms *together* with induction. This is a way to try to figure out which lemmas to use. As we saw in the previous section, finding the right lemmas is the cornerstone problem of inductive reasoning. Our theorem provers do theory exploration using the already existing system QuickSpec [23].

QuickSpec-style theory exploration

QuickSpec does theory exploration by enumerating terms and testing them against each other to find candidate conjectures. Its input is an executable program. To describe its operation, we will continue the example from Section 1 and give QuickSpec the program with definitions of *rev*, *qrev* and $+\!\!+$, and tell QuickSpec to build well-typed terms from these functions, the list constructors and free variables. Some example terms that it would generate are:

(1) $\mathit{qrev}\ \mathit{xs}\ \mathit{ys}$
(2) $\mathit{rev}\ \mathit{xs} +\!\!+ \mathit{ys}$
(3) $\mathit{xs} +\!\!+ \mathit{ys}$
(4) $\mathit{qrev}\ (\mathit{rev}\ \mathit{xs})\ \mathit{ys}$
(5) $\mathit{rev}\ (\mathit{rev}\ \mathit{xs}) +\!\!+ \mathit{ys}$

Of course, many more terms would be generated. The default is to generate terms up to depth three. QuickSpec proceeds by generating random values for the variables using QuickCheck, and then evaluating the terms. Suppose $\mathit{xs} \mapsto [1,2]$ and $\mathit{ys} \mapsto [3,4]$. The terms above evaluate to the following under these assignments:

(1)	$\mathit{qrev}\ \mathit{xs}\ \mathit{ys}$	$= \mathit{qrev}\ [1,2]\ [3,4]$	$= [2,1,3,4]$
(2)	$\mathit{rev}\ \mathit{xs} +\!\!+ \mathit{ys}$	$= \mathit{rev}\ [1,2] +\!\!+ [3,4]$	$= [2,1,3,4]$
(3)	$\mathit{xs} +\!\!+ \mathit{ys}$	$= [1,2] +\!\!+ [3,4]$	$= [1,2,3,4]$
(4)	$\mathit{qrev}\ (\mathit{rev}\ \mathit{xs})\ \mathit{ys}$	$= \mathit{qrev}\ (\mathit{rev}\ [1,2])\ [3,4]$	$= [1,2,3,4]$
(5)	$\mathit{rev}\ (\mathit{rev}\ \mathit{xs}) +\!\!+ \mathit{ys}$	$= \mathit{rev}\ (\mathit{rev}\ [1,2]) +\!\!+ [3,4]$	$= [1,2,3,4]$

Based on the evaluation the terms are divided into two equivalence classes, as indicated by the separating line. The testing goes on until the equivalence classes seem to be stable: that none of them has been further divided for some pre-defined amount of tests. Upon reaching this state, equations are extracted from the classes. The following equations can be extracted from the second equivalence class with terms (3), (4) and (5):

$$\forall\, \mathit{xs}\ \mathit{ys} \cdot \mathit{xs} +\!\!+ \mathit{ys} = \mathit{qrev}\ (\mathit{rev}\ \mathit{xs})\ \mathit{ys}$$
$$\forall\, \mathit{xs}\ \mathit{ys} \cdot \mathit{xs} +\!\!+ \mathit{ys} = \mathit{rev}\ (\mathit{rev}\ \mathit{xs}) +\!\!+ \mathit{ys}$$
$$\forall\, \mathit{xs}\ \mathit{ys} \cdot \mathit{qrev}\ (\mathit{rev}\ \mathit{xs})\ \mathit{ys} = \mathit{rev}\ (\mathit{rev}\ \mathit{xs}) +\!\!+ \mathit{ys}$$

We can now *conjecture* that these equations hold *universally*, even though they have only been tested it on finitely many values. Based on experimental results, for many programs there are no or very few erroneous conjectures. Nevertheless, there is no guarantee of truth so a theorem prover cannot assume them without first proving them.

All lemmas needed to prove the *rev*–*qrev* property from Section 1 are found by QuickSpec. In fact, more properties than needed are discovered. One such redundant conjecture is $\forall\, xs \cdot rev\ (rev\ xs) = xs$. This over-generation can be a drawback of this approach of theory exploration. The theory exploration that QuickSpec offers is not goal-oriented. Rather, it tries to discover everything that is true about the program, regardless of what would be useful in proving the conjecture.

3 HipSpec

This section summarises the paper 1, *Automating Inductive Proofs using Theory Exploration*.

HipSpec is an inductive theorem prover based on theory exploration that sprung out of a fusion of two systems:

1. **Hip** [57] (the Haskell Inductive Prover), which proves Haskell properties by translating the program to first-order logic, by applying induction on the property, and using off-the shelf theorem provers to discharge the proof obligations.

2. **QuickSpec** [23], introduced in the previous section, is used to discover extra properties that are proved and then used as lemmas.

A schematic overview of the HipSpec architecture and its proving loop can be seen in Figure 1. HipSpec maintains a set of open conjectures, initialised with the user-stated properties and conjectures from QuickSpec. We also keep a first-order theory of the translated program as well as a background theory of already proved lemmas.

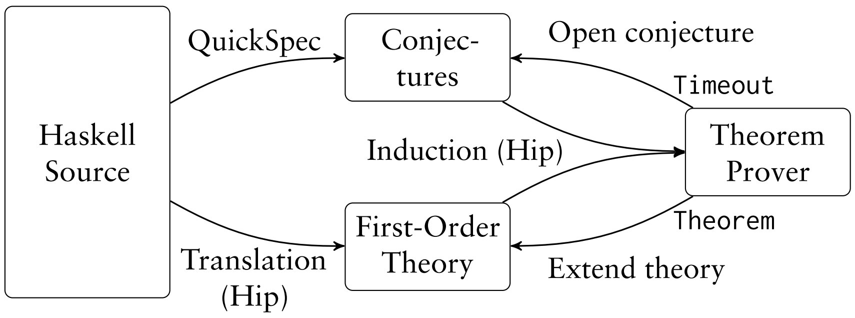

Figure 1: An overview of HipSpec.

The loop begins by carefully choosing one of the open conjectures, applying induction to it and asking a theorem prover to prove it using the current first-order theory. A success promotes the conjecture to a lemma which is added to the background theory. Failure is indicated by a timeout (since provers will in general only terminate on theorems), in which case the conjecture is left open and can be retried later. HipSpec terminates by either giving up if no progress is made in the current theory or by succeding when the user-stated properties are proved.

In the evaluation of HipSpec, we showed that it could prove more properties than any other prover on the difficult benchmark suite from Ireland et al. [35]. In particular, it was the first prover to fully automate the *rotate–length* challenge from [18].

4 Hipster

This section summarises the paper 2, *Hipster: Integrating Theory Exploration in a Proof Assistant.*

Hipster is HipSpec's sibling and works similarly to it. But instead of starting from a Haskell program, it works within the proof assistant Isabelle [51], and operates on an Isabelle theory with function definitions and data types. The Isabelle code is translated to Haskell using the Isabelle code generator [29]. Hipster automatically adds QuickCheck generators for arbitrary values (which QuickSpec needs) to this code and transforms the program to handle Isabelle functions that are not total such as taking the head of a list. After this post-processing of the generated code, theory exploration is provided by QuickSpec. Its conjectures are fed back into a loop similar to HipSpec's. Hip is not used; we instead use an induction tactic followed by first-order reasoning using for example simplification or Isabelle's trustworthy prover Metis [33]. All proofs are carried out in Isabelle, which guarantees that Hipster is sound.

One interesting aspect of working inside a proof assistant is that it enables interacting in a natural way with the theory exploration system. For example, one can use Hipster when stuck in a failed proof attempt. It is also possible to use Hipster in exploratory mode, without any user-stated properties, only function definitions, and Hipster will find and prove properties about them, effectively building a background theory to be used for human-created proofs.

5 Benchmarks for inductive theorem provers

This section summarises paper 3, *TIP: Tons of Inductive Problems.*

To compare provers or configurations of a prover one needs to have a set of benchmarks. For evaluation of provers in the last years the benchmarks have come from two sources, 86 problems from Johansson et al. [37] and 50 problems from Ireland et al. [35]. The benchmarks from [37] were written in Isabelle, and those from [35] were provided only as conjectures without function definitions.

To use these benchmark suites to evaluate new provers, their developers had to translate the problems into their formats. This was done by hand, a tedious process with little guarantee that everyone ends up with the same function definitions. As a research community this is a big problem: we were lacking a *shared* set of benchmarks. There was not even an appropriate format available that could communicate benchmarks.

We designed a simple format for expressing inductive problems named *TIP*, short for *Tons of Inductive Problems*, to signal the intent to gather a vast corpus of problems. We translated the problems from [37] and [35] into this format, and gave definitions in the case they were missing. The format itself is based on the well-established SMT-LIB format [5], which already included data types and recursive functions definitions, two necessary features. On top of SMT-LIB we added two extensions: rank-1 polymorphism and first-class functions, including lambda expressions.

Basing it on an already established format such as SMT-LIB has immediate advantages over starting from scratch: it saves time, it is already well-thought out and already supported in many systems. Further, there are many builtin-theories of SMT-LIB waiting to be used in TIP when the need arises within inductive theorem proving. So far benchmarks are only using integer theories, if any. Basing TIP on another format than SMT-LIB was judged to not be as well-suited in comparison: TPTP [59] has no way of expressing data types, Why3 [27] requires more effort to parse and enforces structural termination of all functions, and Isabelle [51] is also more difficult than SMT-LIB to parse, and typically expresses pattern matching not by case-expressions but by left-hand side fall-through pattern matching which is more complicated to manipulate.

We noticed another issue with the problems from [37] and [35]: only a small fraction are not solved by the state-of-the-art provers, making them much too easy. For instance, many of the problems do not need any auxiliary lemmas, and all function definitions are structurally recursive. To advance and keep interest up in this field, more challenges are needed. As part of our effort of the TIP benchmark suite, we also added hundreds of new problems with varying difficulty beyond the capabilities of the provers of today. To solve these benchmarks, provers will need to use stronger induction techniques, handle modules with many more functions, invent lemmas with conditionals and synthesise functions to express lemmas.

6 Bridging different formats and provers

This section summarises paper 4, *TIP: Tools for Inductive Provers*.

It is perhaps unrealistic to assume that all provers will start supporting TIP directly. In particular for unmaintained provers. How can we evaluate provers that do not support TIP natively? We provide a toolbox, the *TIP tools*, for solving these incompabilities. It includes transformations that translate them into logically equivalent (or weaker) versions. Some important transformations are defunctionalisation for removing higher-order functions, monomorphisation for instantiating polymorphic definitions, and replacing data types with first-order axiomatisations.

In fact, these are essentially the transformations that were developed for Hip and HipSpec. This is because HipSpec already had to translate to other logics to use the back end theorem provers. However, with the tools these transformations are modularised and can be used independently of HipSpec. The TIP tools can already output to a handful of formats, including TPTP, WhyML, Isabelle/HOL, Waldmeister, simplified SMT-LIB and Haskell, and adding a new printer is a small effort.

In the long run, it is better if most inductive provers have built-in support for the same format. However, for many problems it will not be important to have sophisticated (complete) ways of dealing with for example polymorphism and higher-order functions. In these (common) cases, a solver could use our transformations to simplify the problem. This allows prover developers to focus on specific areas: such as in this example first-order simply-sorted problems. By providing these transformations modularly, we provide opportunities for code reuse.

We also supply transformations that call QuickSpec and apply induction, which are taken from HipSpec. Using all these components, the TIP tools can be used as a foundation for new experimental inductive theorem provers.

7 Related work

Lemma invention has been identified as the key problem in inductive theorem proving. There are several ways to invent lemmas. The main contemporary approaches can be divided into two: top-down approaches that examine a proof state syntactically, and bottom-up theory exploration.

Proof critics Proof critics [35] are heuristics concerning what to do when the proof is stuck (no more rewrite rules apply). Broadly speaking, these are lemma discovery techniques in top-down provers. Rippling [18] is a heuristic specifically designed to guide rewriting of the induction conclusion to the hypothesis, with a specific measure that ensure that this process terminates. Critics are used in case the proof gets stuck rather than ending up in the induction hypothesis. Examples of rippling provers are CLAM [35], INKA [6, 34] and IsaPlanner [26]. The names of the critics below are from [35].

The most straightforward critic is *lemma calculation*. This is applicable when the proof is stuck after the induction hypothesis is applied. Another round of induction can then be applied. Common subterms are generalised if there are any. For example, in the step case of the proof of $rev\ (rev\ xs) = xs$, after applying the induction hypothesis on the left hand side, the goal is stuck at $rev\ (rev\ xs \mathbin{+\!\!+} [x]) = x : rev\ (rev\ xs)$. The $rev\ xs$ subterms are now are generalised to ys, obtaining $rev\ (ys \mathbin{+\!\!+} [x]) = x : rev\ ys$. This new equation can be proved by induction on ys.

Lemma calculation is the main lemma discovery technique in Zeno [58] as well as ACL2 [40] and its predecessor Nqthm (also known as the Boyer/Moore Theorem Prover) [13]. Zeno, which like HipSpec operates on terminating and finite Haskell programs, does not support inventing or using any lemmas besides those found by this critic.

The two stronger critics *lemma speculation* and the *generalisation critic* augment the conjecture or the inductive conclusion with “holes” represented by higher-order meta variables to be solved by higher-order unification. Both techniques are implemented in CLAM. The generalisation critic can find generalisations for tail recursive accumulator functions, such as the generalisation in the *qrev–rev* property (but only when appropriate lemmas about list append are manually provided.) Lemma speculation can find lemmas such as $(xs \mathbin{+\!\!+} [y]) \mathbin{+\!\!+} zs = xs \mathbin{+\!\!+} y{:}zs$ from a stuck inductive step of $rev\ (rev\ xs \mathbin{+\!\!+} ys) = rev\ ys \mathbin{+\!\!+} xs$. In the evaluation of implementing these critics in IsaPlanner [38], it was concluded that lemma speculation is seldom applicable and that a theory exploration system would find the lemmas with less effort.

HipSpec never applies a new round of induction inside an inductive proof, which is what the lemma calculation and lemma speculation critics do. Theoretically, this is not necessary as all induction can be on "the top level" (as in the Peano axioms), but it can be more natural and provide opportunities to syntactically discover lemmas during proof search.

Bottom-up theory exploration Theory exploration was defined in the context of the interactive system Theorema [16, 17]. It has since been used to explore mathematical theories in MATHsAiD [48]. In the context of automated inductive theorem proving, theory exploration was first implemented in IsaCoSy [39]. HipSpec, and especially Hipster because it is also based in Isabelle, are in many ways a continuation of IsaCoSy. However, IsaCoSy's theory exploration is not as efficient.

Theory exploration inside SMT The SMT solver CVC4 has support for structural induction over datatypes [55]. Induction is integrated into its SMT loop: the quantifier instantiation is modified to also instantiate induction schemas and generate lemmas to be added to the theory. This is done by enumerating equations in a similar way to QuickSpec. The search space is pruned by filtering heuristics: for example, it tries to falsify candidate lemmas by checking for counterexamples in the current ground model. This way of searching for counterexamples also works when functions are not fully defined, unlike QuickSpec which needs executable functions. Although only induction on data types is supported, it can be used in conjunction with the rest of its SMT theories.

Schematic theory exploration In IsaScheme [50], a predefined set of property schemas are instantiated with the function symbols in the program to create conjectures. For example, there is a schema about distributivity, so for each pair of binary operators it is conjectured that they distribute over each other. Invalid conjectures are filtered out by searching for counter examples. One drawback is that it only works at conjecturing properties in these particular simple and elegant shapes. This makes completeness suffer since some important lemmas do not have a simple shape. On the other hand, it is efficient when lemmas do have these exact shapes.

Non-structural induction The semi-automatic theorem prover Dafny [44] does all inductive reasoning using only strong induction. There are default orders given for builtin-types and the order for datatypes is size. However, the order can be chosen manually by the user. Dafny does not have any lemma discovery techniques.

The newest version of Hipster [47] has a tactic for instantiating the automatically derived Isabelle recursion-induction schemas [41]. It has been evaluated on structurally recursive functions and is stronger than using induction on one variable. It is not yet thoroughly evaluated on non-structurally recursive functions.

Proof by consistency Without using an induction schemas, a theory together with a conjecture can sometimes be shown to be consistent, effectively showing that it is an inductive theorem. This approach is called *proof by consistency* or *inductionless induction.* For examples using Knuth-Bendix completion for showing consistency, including showing *rev* being an involution, see [32]. Unfortunately, the technique seldom works since showing that a theory is consistent is very difficult.

Contract-style induction Annotating functions with pre- and post-conditions gives rise to an inductive technique. The idea is to make a well-formedness check for each function. The pre-condition is assumed, and the post-condition has to be proved. Further, the pre-condition must hold at each recursive call, and then the post-condition is asserted, analogous to an induction hypothesis. This is the technique used in for example Leon [60], Liquid Haskell [36] (with contracts expressed in the type system as *refinement types*), Dafny [43] (as an alternative to quantifiers), and HALO [62] (as fix-point induction). Some drawbacks are that not all properties can be expressed this way, and it is unclear how to combine several contracts for the same function.

Other techniques The higher-order prover Agsy [45, 46] can find inductive proofs using the search technique *narrowing* over Agda [52] terms. Proofs are structural over inductive families, but those can encode for example well-founded induction.

In *cyclic proofs* [14], proofs are allowed to contain cycles as long as the goal to be proved decreases in each cycle according to some well-founded ordering. This has been implemented in the Cyclist prover [15] which allows induction over inductive predicates. It does not offer any solutions to lemma discovery.

The optimization technique *equality saturation* [61] has been adapted to functional programming as an inductive theorem prover named GraphSC [28], where proving amounts to bisimulation. The experimental version Pirate of the superposition prover SPASS has been extended with support for induction over data types [65]. Machine learning has been used to find lemmas by looking for analogies to existing lemmas and function definitions [31].

HipSpec allows induction on several variables, with strictly smaller subterms as induction hypotheses. There are more sophisticated ways to compute and combine induction hypotheses [63, 64] or they can be generated lazily [54].

8 Future work and conclusions

This section outlines some unsolved questions in doing proofs by induction. Additionally, these are some issues that must be addressed in order to progress on the benchmark suite.

Conditional properties

Many important properties are *conditional*: they have preconditions. QuickSpec is currently optimised to conjecture equations only. An example of a conditional property is that inserting an element at the correct position in it preserves the list being sorted:

$$\forall\, xs \cdot \mathit{sorted}\ xs \Rightarrow \mathit{sorted}\ (\mathit{insert}\ x\ xs)$$

Other examples include injectivity, ordering properties such as symmetry, asymmetry, antisymmetry, transitivity, totality and functions that are only well-behaved when some invariant is assumed on their arguments: *sorted* is an invariant preserved by *insert*.

The latest version of Hipster [47] used a modified version of QuickSpec which allowed the user to interactively choose one unary or binary predicate as a precondition, such as *sorted* or $<$. The data generators were not adapted to satisfy the preconditions, leading to many false conjectures unless the number of tests was greatly increased. Generating values that makes the precondition true, such as sorted lists, is partly what makes conditional theory exploration difficult. Another reason is that the search space is huge with many uninteresting truths.

Beyond structural induction

So far, we have only focused on structural induction. This works well for reasoning about functions that are structurally recursive, but many interesting functions are not. One example is quicksort, and a non-inplace version can be written tersely in Haskell in this way:

$$\begin{aligned} &\mathit{qsort}\ []\ &&= []\\ &\mathit{qsort}\ (x : xs) &&= \mathit{qsort}\ (\mathit{filter}\ (< x)\ xs) +\!\!+ [x]\\ & &&\quad +\!\!+\ \mathit{qsort}\ (\mathit{filter}\ (\geqslant x)\ xs) \end{aligned}$$

Attempting to prove a property about *qsort* using structural induction is hopeless, since the induction hypothesis will talk about the tail of the list whilst the recursive calls are through *filter*.

One technique to prove properties about this is to use *recursion-induction* [41], which creates an inductive schema based on the recursive structure of a function, which is sound only if the function terminates. Here is the recursion-induction schema generated from *qsort*:

$$\frac{\mathsf{P}([])\quad \forall\, y\ ys \cdot \mathsf{P}(\mathit{filter}\ ({<}y)\ ys) \wedge \mathsf{P}(\mathit{filter}\ (\geqslant y)\ ys) \Rightarrow \mathsf{P}(y : ys)}{\forall\, xs \cdot \mathsf{P}(xs)}\ \textsc{qsort}$$

Another method is to use the *strong induction* over natural numbers. A *decreases measure* can be used, such as *length* for lists. Here is the strong induction schema built from *length*:

$$\frac{\forall\, zs \cdot (\forall\, ys \cdot \mathit{length}\ ys < \mathit{length}\ zs \Rightarrow \mathsf{P}(ys)) \Rightarrow \mathsf{P}(zs)}{\forall\, xs \cdot \mathsf{P}(xs)}\ \textsc{length}$$

The LENGTH schema implies the recursion-induction schema QSORT since *length* (*filter* *p* *xs*) < *length* (*x* : *xs*) (though this is an inductive truth). The QSORT schema is incomparable with structural induction, whilst the LENGTH schema implies also structural induction, making it the most versatile of the three.

Neither of these techniques are yet available in HipSpec. Research questions include when to use which function schema, and what decreases measure to use. Above we used *length*, but any function f returning a natural number can be used. (Proof: assume we have an infinite descending chain $a_1, a_2, \ldots$ in the domain of f. Then $f\ a_1, f\ a_2, \ldots$ is an infinite descending sequence of natural numbers, which is absurd.) In the most general case any well-founded ordering can be used.

Synthesis of new functions to express lemmas

Theory exploration using only the functions available in the program has its limitations, since lemmas can only be expressed by using these very functions. An example of a property that cannot be proved when a function is missing is the *rotate–length* property. The standard definitions are given in Paper 1, but here we will define *rotate* using a *snoc* function that adds one element to the end of a list:

$$
\begin{aligned}
&snoc :: a \to [a] \to [a]\\
&snoc\ x\ []\qquad\ \ = [x]\\
&snoc\ x\ (y : ys) = y : snoc\ x\ ys
\end{aligned}
$$

This is how the alternative *rotate* function is defined using *snoc*:

$$
\begin{aligned}
&rotate :: Nat \to [a] \to [a]\\
&rotate\ Zero\qquad xs\qquad\ = xs\\
&rotate\ (Succ\ n)\ []\qquad\ \ = []\\
&rotate\ (Succ\ n)\ (x : xs) = rotate\ n\ (snoc\ x\ xs)
\end{aligned}
$$

The *rotate–length* property is:

$$\forall\ xs \cdot rotate\ (length\ xs)\ xs = xs$$

This cannot be proved without lemmas. One way to prove it is by the following generalisation, which however requires the use of $+\!\!+$:

$$\forall\ xs\ ys \cdot rotate\ (length\ xs)\ (xs +\!\!+ ys) = ys +\!\!+ xs$$

Without adding any new functions to the program (such as $+\!\!+$), this crucial generalisation cannot be expressed, and I conjecture that no lemmas exist which prove the property under this assumption. This is thus an example where a function must be synthesised for an inductive proof to be found.

We do not currently have any function synthesis component in HipSpec, and it is unclear how to synthesise functions in a goal-oriented way. Naturally, $+\!\!+$ could always be added if lists occur, but it is not a very satisfactory heuristic.

Proofs

HipSpec does not yet produce proofs. Being able to transport proofs produced by inductive theorem provers into other systems would be very beneficial. One example is to import them into interactive theorem provers since making interactive proofs can be laborious and time-consuming. Inductive provers could give a proof sketch with information such as induction variables and lemmas used, and let a routine prover in the target system fill in the details. Ideally for portability, tools should output proofs or stubs in a common format based on the TIP format.

We believe that being able to leverage the strengths of inductive theorem provers into other systems is key for broader success and adoption of this research area.

8.1 Conclusions

We developed two inductive theorem provers, HipSpec and Hipster. They both use theory exploration for lemma discovery, which was identified as the main issue in inductive theorem proving. We advanced the state of the art in inductive theorem proving by automatically proving important properties such as *rev–qrev* and *rotate–length*. We have identified key areas for further improvements, including theory exploring conditional equations and using more sophisticated induction techniques. To be able to scientifically evaluate upcoming inductive theorem provers, we created a benchmark suite TIP with hundreds of new challenge problems and an accompanying tool box for converting it to different formats.