Automating Inductive Proofs using Theory Exploration

Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone

Department of Computer Science and Engineering, Chalmers University of Technology
{koen,moa.johansson,danr,nicsma}@chalmers.se

Abstract. HipSpec is a system for automatically deriving and proving properties about functional programs. It uses a novel approach, combining theory exploration, counterexample testing and inductive theorem proving. HipSpec automatically generates a set of equational theorems about the available recursive functions of a program. These equational properties make up an algebraic specification for the program and can in addition be used as a background theory for proving additional user-stated properties. Experimental results are encouraging: HipSpec compares favourably to other inductive theorem provers and theory exploration systems.

1 Introduction

We are studying the problem of automatically proving algebraic properties of programs. Our aim is to build a tool that programmers can use to support software development. This paper describes current progress towards this goal, in particular addressing the problem of automating inductive proofs.

We work in a subset of the strongly typed functional programming language Haskell. Our subset consists of monomorphic, terminating programs without type classes or primitive types (like `Int`). The only data types are algebraic data types, functions and uninterpreted types. Removing these restrictions is ongoing work.

There are two key advantages of using Haskell as the input language. Firstly, a pure functional programming language is semantically simpler and thus easier to reason about than languages with side effects. Secondly, many Haskell programmers already use QuickCheck [5], a tool for property-based random testing, which means that many Haskell program are already annotated with formal properties (tested, but not proved).

The main obstacles one encounters when doing automated verification of functional programs are (1) when and how to apply induction, and (2) how to discover auxiliary lemmas or generalisations which may be required in inductive proofs. Let us look at a simple example. Consider the following Haskell program implementing the list reverse function in two different ways, `rev` and `qrev`. The latter uses a helper function `revacc` with an accumulating parameter which leads to a function with better time complexity. Their definitions are:

```
rev []     = []
rev (x:xs) = rev xs ++ [x]
```

```
revacc []     acc = acc
revacc (x:xs) acc = revacc xs (x:acc)

qrev xs = revacc xs []
```

A natural property one would like to verify is that the functions above produce the same result: $\forall$ `xs. rev xs = qrev xs`. Suppose we attempt to prove this by structural induction on `xs`. This will fail as the inductive hypothesis `rev as = qrev as` is too weak to prove `rev (a:as) = qrev (a:as)`. What is needed here is an additional lemma such as `rev xs++ys = revacc xs ys`, from which the original conjecture follows as a special case when `ys` happens to be the empty list. This is a typical example of the kind of generalisations which are required in proofs about functions with accumulator variables. One of the main challenges for inductive theorem provers is how to discover such lemmas automatically.

Current inductive theorem provers such as IsaPlanner [8], Zeno [19] and ACL2 [13] support a simple lemma discovery technique called *lemma calculation*, by which a new lemma is suggested by replacing some common subterm in a stuck goal by a variable. Although this technique works very well for many proofs, it is not enough for the above example, which cannot be automatically proved by these systems. The now defunct CLAM proof-planner had in addition a so-called *proof-critic* for discovering more complex generalisations [9], such as the one required in the example, but only if other basic lemmas were given by the user.

Our approach differs from the *top-down* manner in which the above systems work. Instead of waiting for the proof to somehow get stuck, we use *bottom-up* lemma discovery, or *theory exploration.* Our tool, called HipSpec, gets its name from its two subsystems which we developed previously: the automated inductive prover Hip [18], and the conjecture generation system QuickSpec [6]. Hip tries to prove a conjecture by enumerating all possible ways of doing structural induction over the free variables, and then calling an automated first-order prover to prove them. QuickSpec creates thousands of terms involving the functions of a given API, and computes equivalence classes over these terms by means of testing. Each pair of terms t_1, t_2 in an equivalence class gives rise to a conjecture $t_1 = t_2$.

HipSpec reads in a program, but besides trying to tackle any of the user-given properties, it asks QuickSpec to produce a list of conjectures about the program. HipSpec then sends these conjectures to Hip and those that are proved can be used as lemmas in subsequent proof-attempts. After this theory exploration phase, the properties stated by the programmer are tried, using all the proved lemmas as background theory.

There are several theory exploration systems which have been applied to discover theorems in inductive theories [12,15,16], but none have been fully integrated with an automated theorem prover in order to supply the prover with lemmas. Instead, these systems simply generate and prove a set of ‘interesting’ equations summarising the main properties about the program, which are then presented to the user. In fact, HipSpec may also be used in this manner without any user-stated properties.

Let us return to the example property about `rev`. HipSpec calls QuickSpec, which within a few seconds conjectures a set of equations about the functions involved. HipSpec feeds these to Hip, which tries to prove them. Those that can be proved without induction are redundant and can be discarded; the lemmas needing induction are shown below[1]:

No	Conjecture		Proved using[2]
(*1*)	`xs++[]`	`= xs`	`xs`
(*2*)	`(xs++ys)++zs`	`= xs++(ys++zs)`	`xs`
(*3*)	`rev xs++rev ys`	`= rev (ys++xs)`	`ys`, (*1*), (*2*)
(*4*)	`revacc (revacc xs ys) []`	`= revacc ys xs`	`xs`
(*5*)	`revacc (revacc xs ys) zs`	`= revacc ys (xs++zs)`	`xs`
(*6*)	`revacc xs ys++zs`	`= revacc xs (ys++zs)`	`zs`, (*1*), (*2*), (*5*)
(*7*)	`revacc xs (rev ys)`	`= rev (ys++xs)`	`xs`, (*1*), (*3*), (*6*)

The original property is now easily proved: it follows directly from (*7*), letting `ys = []`, and the definition of `qrev`; induction is not even needed. Note that lemma (*4*) is not needed for proving the original property. Discovering some unnecessary lemmas is a (potentially disadvantageous) side-effect of the bottom-up approach.

Contributions. We augment the automated induction landscape with a new method which uses a bottom-up theory exploration approach to find auxiliary lemmas. This approach combines our own earlier work on conjecture generation based on testing (QuickSpec) and induction principle enumeration (Hip). By adding proof capabilities on top of QuickSpec we also get a system which can be used as a stand-alone theory exploration system.

Our hypothesis is that:

1. Algebraic equations constructed from terms up to a certain depth form a rich enough background theory for proving many algebraic properties about programs without specialised proof-critics.
2. A reasoning system for functional programs can be built on top of an automatic first-order theorem prover.
3. A system combining (1) and (2) can be used both as a theorem prover and as an efficient theory exploration system, producing background lemmas comparable to those appearing in human-created libraries.

The experimental results in this paper have so far confirmed this.

2 Implementation

Below we describe in more detail how Hip and QuickSpec work, and how they are combined in HipSpec.

[1] The variables are implicitly universally quantified over total and finite values.

[2] This column shows the induction variables and which lemmas were used.

2.1 Hip

Hip [18] is an automatic tool for proving user-stated equality or implicational properties about Haskell programs. Hip starts by compiling the definitions in the program at hand to first-order logic. For each property stated in the program, it systematically applies different induction rules, yielding first-order proof obligations, which are tested for validity using off-the shelf automated first-order theorem provers. If one proof obligation succeeds, the original conjecture was valid. Thus, the first-order prover takes care of non-inductive reasoning, while Hip adds inductive reasoning at the meta-level. In the context of HipSpec, Hip is configured to apply structural induction up to a given depth on one or more variables. Hip, however, also supports co-inductive proof techniques such as fixed point induction. The focus of our work in HipSpec is currently not on proving termination, so we restrict ourselves by allowing only well-founded definitions, and put the responsibility on the end user to enforce this policy for now.

2.2 QuickSpec

QuickSpec [6] conjectures equations about a functional program by means of testing. The user of QuickSpec provides a list of functions and their types, a random test data generator for each of the types involved, a set of variables (usually 2-3 per type), and a term depth limit (usually 3). QuickSpec starts by creating a set of terms, called the *universe*, consisting of all well-typed terms built from the functions and variables given, whose depth is within the given limit. It then partitions this universe into equivalence classes by running a finite number of random tests (usually 100); two terms will be in the same equivalence class if and only if they were equal for all tests. This equivalence relation in turn gives rise to a huge set of conjectured equations about the tested program (typically thousands or tens of thousands). For the sake of human users, QuickSpec also includes a final phase which prunes away equations that follow from simpler ones, leaving only a small core of equations from which all original equations follow. This core is usually presented to the user (usually 10-25 equations). However, when HipSpec uses QuickSpec to generate lemmas, it does not use the pruning phase, because valuable lemmas may be pruned away. For example, even when an equation E_1 implies a more complex equation E_2, we can not necessarily discard E_2, because E_2 may be provable by induction whereas E_1 may not be. In fact, E_2 may very well be needed as a lemma to prove E_1! So, HipSpec considers the full set of equations produced by QuickSpec before pruning.

2.3 HipSpec

HipSpec's operation is illustrated in Figure 1. We start by running QuickSpec on the program source file, which generates a list of conjectures. We also translate the program source code to a first-order theory using Hip.

HipSpec maintains three sets of equations: *active conjectures*, which we still need to consider, *failed conjectures*, which we have already tried to prove but

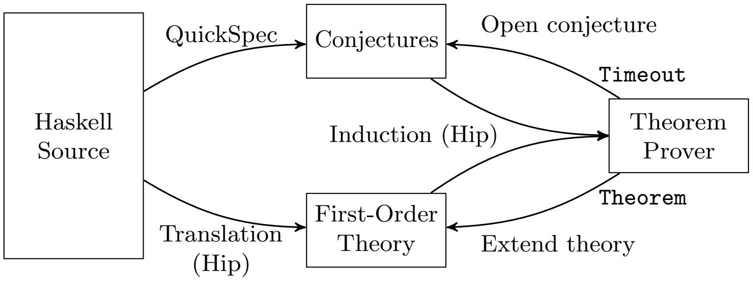

Fig. 1. An overview of HipSpec.

failed, and *lemmas*, which we have managed to prove. The *first-order theory* in Figure 1 consists of Hip's translation of our program plus the current set of lemmas. Initially the active conjectures consist of all equations that QuickSpec found (even those that would have been removed by pruning), and the failed conjecture set and lemma set are empty.

The main loop works as follows:

1. Pick a conjecture c from the active conjecture set (using a heuristic described below).
2. Check if c follows from the lemmas found so far by equational reasoning only. If so, discard c, and re-iterate.
3. Otherwise, ask Hip to prove the conjecture by induction, using definitions and previously proved lemmas as background theory.
4. If Hip succeeds, move c to the lemma set, and move some failed conjectures back to the active conjectures (based on a heuristic described below).
5. If Hip does not succeed within a set timeout, move c to the failed conjectures.

The loop ends when the active conjecture set is empty.

Picking the conjecture The performance of HipSpec completely depends on one heuristic: which active conjecture to try to prove next. Our current heuristics are rather crude; more sophisticated techniques are further work.

Our basic strategy is to prove simpler equations before more complicated ones. We define simplicity as follows. A smaller term is simpler than a bigger term; if two terms have the same size, the term with more variables is simpler (because it might be more general). For example, `(x+y)+z=x+(y+z)` is simpler than `(x+x)+y=x+(x+y)`. The simplicity of an equation $t_1 = t_2$ is determined by whichever of t_1 and t_2 is the most complex.

We also take into account the call graph of the program. For example, if we are proving properties about the natural numbers, we prove as much as possible about `+` before starting on `*`, since `*` calls `+`. More precisely, when choosing which conjecture to prove next, we pick the one whose call graph is the smallest; if two conjectures have the same size call graph, we pick the simplest one.

Discarding trivial consequences It is quite expensive to send every conjecture to Hip to be proved, when we may have thousands of them. Luckily, QuickSpec has a lightweight theorem prover based on congruence closure. This prover can efficiently answer questions of the form "given these lemmas, can I prove this equation?", replying either "yes" or "don't know".

Whenever we pick a conjecture, we check if this prover can prove it from the current lemmas without induction. If so, we just discard it. This filters out most trivial conjectures that are provable without induction.

Re-activating failed conjectures When we prove a lemma, we sometimes move some failed conjectures back to the active conjectures. HipSpec's rule is to wait until the set of active conjectures is empty and then move *all* failed conjectures back to the active set, provided that at least one new lemma was proved since last attempting the conjecture. This guarantees termination.

We have experimented with more elaborate heuristics in this step, eagerly adding failed conjectures back. These heuristics can help in certain examples, but so far none have been sufficiently general. Perhaps surprisingly, the simple method described above works well for all examples in this article. More sophisticated heuristics are further work.

3 Examples

This section gives examples of successful proofs and their related theory explorations, as well as an example showing some current limitations of our approach.

3.1 Rotating the length a of list

This simple property of the `rotate` function is surprisingly difficult to prove[3]:

```
prop_rotate xs = rotate (length xs) xs =:= xs
```

The `rotate` function takes a natural number `n` and returns the list resulting from removing the `n` first elements and appending them to the end. Rotating a list by its length returns the original list. Although this property is very simple to state it is surprisingly hard to prove by mathematical induction, as it requires a generalised version to be proved, which implies `prop_rotate`. This generalisation itself can be proved by induction.

Given the standard definitions of append, length and Peano numbers with successor `S` and zero `Z`, and the below definition of `rotate`, HipSpec finds and proves such a generalisation, and uses it to prove `prop_rotate`:

```
rotate Z     xs     = xs
rotate (S n) []     = []
rotate (S n) (x:xs) = rotate n (xs ++ [x])
```

[3] Here, `=:=` is HipSpec's notation for equality.

The lemmas for which HipSpec needed induction are in Figure 2. Lemma (*8*) is the required generalisation, from which it proves `prop_rotate`, which follows as a special case when `ys` is the empty list. Notice that lemma (*8*) itself requires lemmas (*1*) and (*2*). A number of additional lemmas are also discovered, which are not of use in this particular proof, but could well be useful in other proofs. The whole process of theory exploration and the proof of `prop_rotate` took 17 seconds, with less than a second spent in QuickSpec and the rest of the time spent in various proofs of the generated equations.

No	Conjecture		Proved by
(*1*)	`xs++[]`	`= xs`	`xs`
(*2*)	`(xs++ys)++zs`	`= xs++(ys++zs)`	`xs`
(*3*)	`rotate n (rotate m xs)`	`= rotate m (rotate n xs)`	`n`, `m`
(*4*)	`rotate (S n) (rotate m xs)`	`= rotate (S m) (rotate n xs)`	`xs`, (*3*)
(*5*)	`rotate n [x]`	`= [x]`	`n`
(*6*)	`length (xs++ys)`	`= length (ys++xs)`	`xs`, `ys`
(*7*)	`length (rotate n xs)`	`= length xs`	`n`, (*6*)
(*8*)	`rotate (length xs) (xs++ys)`	`= ys++xs`	`xs`, (*1*), (*2*)
(*9*)	`rotate (length xs) xs`	`= xs`	(*8*)

Fig. 2. Properties generated and proved about the theory of lists with `++`, `rotate`, and `length`. The third column shows which induction variables and lemmas were used.

As this proof requires both generalisation and lemma discovery it was identified in 2005 as an automated reasoning challenge beyond the capabilities of state-of-the-art reasoning systems ([3], p. 77). We are not aware of any other theorem provers which prove this theorem fully automatically, without the help of user-supplied lemmas.

3.2 Nicomachus' Theorem

Using Peano arithmetic, with standard definitions of addition and multiplication recursively on the first argument, we will try to get HipSpec to prove Nicomachus' Theorem. This states that the sum of the n first cubes is the nth triangle number squared: $\sum_{k=1}^{n} k^3 = \left(\sum_{k=1}^{n} k\right)^2$. We define two functions: `tri` calculates triangle numbers and `cubes n` calculates the sum of the first `n` cubes.

```
tri Z     = Z               cubes Z     = Z
tri (S n) = tri n + S n     cubes (S n) = cubes n + (S n*S n*S n)
```

Using these definitions, Nicomachus' theorem is stated as follows:

```
prop_Nicomachus x = cubes x =:= tri x * tri x
```

When HipSpec is given the definitions of plus, multiplication, `tri` and `cubes`, it generates and proves (by induction) the properties listed in Figure 3 below, which takes 10 seconds. The properties are listed in the order they were proved.

No	Conjecture	Lemmas used	Induction on
(*1*)	`x+y           = y+x`		`x`, `y`
(*2*)	`x+(y+z)       = (y+x)+z`	(*1*)	`z`
(*3*)	`x*y           = y*x`	(*2*)	`x`, `y`
(*4*)	`x*(y*z)       = (y*x)*z`	(*1*), (*2*), (*3*)	`x`, `y`
(*5*)	`x*(y+y)       = y*(x+x)`	(*1*), (*2*), (*3*), (*4*)	`y`
(*6*)	`(x*y)+(x*z) = x*(y+z)`	(*1*), (*2*), (*3*)	`z`
(*7*)	`tri x*(y+y) = (x*y)*S x`	(*1*), (*2*), (*3*), (*4*), (*6*)	`x`
(*8*)	`tri x+tri x = x+(x*x)`	(*1*), (*2*), (*3*)	`x`
(*9*)	`tri x*tri x = cubes x`	(*1*), (*2*), (*3*), (*6*), (*8*)	`x`

Fig. 3. Properties proved about the theory with natural number addition, multiplication, triangle numbers (`tri`) and sum of cubes (`cubes`).

In (*8*) the well-known identity $\sum_{k=1}^{n} k = n(n+1)/2$ is proved, using previously proved lemmas. From this lemma HipSpec proves Nicomachus' Theorem in (*9*). Due to the order in which HipSpec ends up proving the conjectures in this example, some unnecessary lemmas are included in figure 3, e.g. (*5*) and (*7*).

3.3 Insertion sort produces a sorted list

Currently, QuickSpec can only generate equational lemmas. To prove that, for example, insertion sort produces a sorted list requires conditional lemmas. We state this property as `prop_sorted xs = sorted (isort xs) =:= True`.

In order to prove `prop_sorted` we need the conditional lemma `sorted xs ==> sorted (insert x xs)`, where `insert` is the sorted list insertion function used by `isort`, but HipSpec only can only discover and prove the somewhat peculiar equations (lemmas *1-4*) in Figure 4. HipSpec also discovers, but fails to prove, some additional properties (conjectures *5-9*). For example, property (*5*), which states that `insert` is commutative in its first argument. These equations are not proved because they require conditional lemmas.

Although not proved, QuickSpec has tested these equations and not found a counterexample. Hence, even a failed proof attempt may at least give some insight into the properties of the program. The runtime for this example was 8 seconds.

No	Conjecture		Induction on
(*1*)	`x<=x`	`= True`	`x`
(*2*)	`x<=S x`	`= True`	`x`
(*3*)	`S x<=x`	`= False`	`x`
(*4*)	`insert y (x:[])`	`= insert x (y:[])`	`x`, `y`
(*5*)	`insert x (insert y xs)`	`= insert y (insert x xs)`	
(*6*)	`sorted (insert x xs)`	`= sorted xs`	
(*7*)	`isort (insert x xs)`	`= isort (x:xs)`	
(*8*)	`sorted (isort xs)`	`= True`	
(*9*)	`isort (isort xs)`	`= isort xs`	

Fig. 4. Results for the theory of insertion sort. Properties *1-4* were proved, while properties *5-9* were not, as they require conditional lemmas.

4 Evaluation

HipSpec has two modes of use. Firstly, it can be used as an automated induction to prove user-given conjectures using theory exploration to find necessary lemmas. In this case, the individual lemmas that are discovered in the background and used in the proofs are of less interest for the user, since the focus is on proving the user supplied properties automatically. Theory exploration is treated more like a black box.

Secondly, HipSpec can be used in a more speculative manner, as a standalone theory exploration system. In this case, the user expects HipSpec to discover and prove a set of basic equational properties about the given program. Here it becomes important not to swamp the user with trivial or overly complicated equations. Rather, we wish to present the user with a concise set of elegant equations summarising the main properties, much like the libraries in proof assistants such as Isabelle. The hope is that these may be useful in later interactive reasoning or as an algebraic specification of the program.

The examples come from the theorem proving literature and assume terminating functions over total values. We used Z3 [7] as a backend for HipSpec in these experiments. As the program is translated to a first order theory, we did not use any of Z3's built-in theories or decision procedures, but we did exploit its support for types and constructor functions. The source code for HipSpec and all experimental results are available online [1,2].

4.1 HipSpec as a Theorem Prover

HipSpec was evaluated on two test suites from the inductive theorem proving literature. The test suites consist of conjectures about natural numbers, lists and binary trees. As they feature a large number of unrelated functions, HipSpec was run separately for each property. This reduces the number of generated equations because HipSpec will ignore any function that is not (directly or indirectly) reachable from the property. It also means that HipSpec cannot use already-proved properties from the test suite to prove later ones. Thus, the order of the properties in the test suite does not matter: they are proved independently.

HipSpec was configured to give a timeout of 1 second for each individual proof obligation sent to the prover, and to allow induction on up to two variables simultaneously using one-step structural induction.

Test Suite A consists of 85 conjectures with both first- and higher-order functions about lists, natural numbers and binary trees [10]. These were originally formalised for the IsaPlanner system in Isabelle's HOL and have since been translated into other formalisms to compare the Zeno and ACL2 Sedan provers [19,4] and the Dafny system [14]. As these systems use different logics we note that the functions are not defined in exactly the same way in the different experiments. This test suite was originally designed for evaluating IsaPlanner's rippling heuristic in the presence of if- and case-expressions, which are expressed as higher-order functions in Isabelle, and cause trouble for IsaPlanner's syntax-based rippling heuristic. Hence, from a lemma discovery point of view, many proofs are rather easy: 67 theorems can be proved without extra lemmas, and 12 do not require induction. The results for the different provers on the 85 conjectures are summarised below:

HipSpec	Zeno [19]	ACL2s [4]	IsaPlanner [10]	Dafny [14]
80	82	74	47	45

HipSpec performs well, with the majority of failures being due to proofs requiring conditional lemmas, as HipSpec only is able to generate equations. For one property (number 81), we had to configure HipSpec to use induction on three variables; this is counted as a success in the table above. Zeno performs best, failing only on three examples, two fewer than HipSpec. However, HipSpec can prove two theorems that Zeno cannot: `rev (drop i xs) = take (len xs-i) (rev xs)` and `rev (take i xs) = drop (len xs-i) (rev xs)`.

Test Suite B consists of 50 theorems about lists and natural numbers and was previously used to demonstrate proof-critics in the CLAM prover [9], which is unfortunately no longer maintained. As opposed to Test Suite A, most theorems here do require auxiliary lemmas, generalisations, case-splits or non-standard inductions. CLAM proves 41 of the 50 theorems fully automatically. The remaining 9 theorems were proved interactively. They require generalisation (including the `rev` example from §1 and the `rotate` example from §3.1), for which CLAM needed the help of some user-supplied lemmas. Again, HipSpec was not given any auxiliary lemmas. Fully automatically, it proved 44 theorems, including 6 of the 9 theorems which CLAM proved with the help of user-supplied lemmas.

We managed to prove 3 further theorems (properties 33–35) by adjusting HipSpec's settings. These three properties concern accumulating versions of multiplication, factorial and exponentiation. Because we are using Peano arithmetic, these functions return large results, and the testing phase used too much memory: we supplied a flag that causes QuickSpec to compare results up to some size bound, so results that are too large will be considered equal. There were also too many conjectures, so we added a flag to limit the *size* of the generated terms. We did *not* have to give any lemmas by hand. In total, HipSpec proved 47 theorems, including the 9 for which CLAM needed user-supplied lemmas.

We also tested Zeno on these examples: it can prove 21, but not any of the ones requiring complex generalisations.

Finally, we remark that the bottom-up approach taken by HipSpec is naturally a bit slower than IsaPlanner and Zeno, which typically perform proofs in less than a second. Most successful proof attempts are very fast, with the long runtimes arising from cases with a lot of failed proof attempts.

For test suite A, all properties required less than a minute on a normal desktop computer [1]. The vast majority required less than 15 seconds, and most 1–2 seconds. For test suite B, the 44 successful properties required at most 15 seconds, most of them 1–2 seconds. The three properties for which we needed to tweak the settings ranged from 30 seconds to 40 minutes. Of the three failed properties, two took about five minutes before giving up, the third 8 seconds.

As mentioned, HipSpec may also discover some superfluous lemmas not strictly required for the proof of the user-stated property. In these examples, there are very few such lemmas and the theorem prover's performance was not notably affected by these being added to the theory.

4.2 HipSpec as a Theory Exploration System

In these experiments HipSpec is given a program as an input, without any user-properties stated. The aim is to present the user with a concise set of equational properties that have been discovered and proved. We exploit the pruning algorithm already implemented in QuickSpec to achieve this. QuickSpec was originally built as a standalone system for suggesting algebraic specifications of programs using testing. When used on its own, it prunes the many equations it generates by heuristically ordering them and removing those that trivially follow from previous ones. We refer to [6] for a detailed description of this pruning algorithm. When HipSpec is used in theory exploration mode, it first attempts to prove as many conjectures as we can, just as in the theorem-proving mode. Then it takes the set of the conjectures that it proved, or that trivially follow from what it proved, and applies the pruning algorithm to this set. As a result, HipSpec often produces a smaller and more concise set of lemmas than it does when used in theorem-prover mode. The final list of equations does not depend on what order we proved things in, or on what needed induction, only on what the theory implies.

We have applied HipSpec to some simple theories from the theory exploration literature [12,16], one about natural numbers, with `+` and `*`, and three small theories about lists: 1) `append`, `reverse` and `length`, 2) `append`, `reverse` and `map` and 3) `append`, `foldl` and `foldr`. The theorems produced are presented in Figure 5. HipSpec generates these theorems much faster than IsaCoSy and IsaScheme: it takes only between 6-12 seconds for each theory (full results available online [1]), while IsaCoSy and IsaScheme may require hours. We expect this to be due to the congruence closure reasoning of QuickSpec, which reduces the search space and integrates counterexample checking in the term generation phase.

We also perform the same precision-recall analysis as in [12,16] to assess the quality of the generated theories using Isabelle's libraries[4] as reference. This experiment assumes that the Isabelle library is so well-designed that it contains exactly all interesting properties and nothing more. The results are summarised in Table 1, where *recall* measures how many of the theorems in the library were also produced by HipSpec, and *precision* measures how many of the theorems HipSpec produced were also in the library, i.e. how well it avoids producing "superfluous" theorems.

HipSpec performs very well: for the lists, it generates all theorems in Isabelle's library, plus theorem *L3* in Figure 5, which is the closest we can get to the useful lemma `length (xs ++ ys) = length xs + length ys` since we did not include the `+` operator in the program. For the natural numbers, HipSpec fails to generate three of the library theorems: the standard formulations of associativity are missing (instead HipSpec generates two variants in theorems *N5* and *N6*) and

[4] `http://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/library/HOL/`

Natural Numbers			
N1.	`x+y = y+x`	N6.*	`x*(y*z) = y*(x*z)`
N2.	`x*y = y*x`	N7.	`x+S y = S (x+y)`
N3.	`x+Z = x`	N8.	`x*S y = x+(x*y)`
N4.	`x*Z = Z`	N9.*	`x*(y+y) = y*(x+x)`
N5.*	`x+(y+z) = y+(x+z)`	N10.	`(x*y)+(x*z) = x*(y+z)`

Lists	
L1.	`xs++[] = xs`
L2.	`(xs++ys)++zs = xs++(ys++zs)`
L3.*	`length (xs++ys) = length (ys++xs)`
L4.	`length (rev xs) = length xs`
L5.	`rev (rev xs) = xs`
L6.	`rev xs++rev ys = rev (ys++xs)`
L7.	`map f xs++map f ys = map f (xs++ys)`
L8.	`map f (rev xs) = rev (map f xs)`
L9.	`foldl f (foldl f x xs) ys = foldl f x (xs++ys)`
L10.	`foldr f (foldr f x xs) ys = foldr f x (ys++xs)`

Fig. 5. Theory exploration results: theorems generated by HipSpec. Theorems marked by $*$ were not in Isabelle's library.

the theorem `S x + y = x + S y` is excluded. However, all three can be trivially derived by equational reasoning from the theorems HipSpec does produce.

	HipSpec	IsaCoSy [12]	IsaScheme [16]	Isabelle
#Thms Naturals	10	16	16*	12
Precision	80%	63%	100%*	-
Recall	73%	83%	46%*	-
#Thms Lists	10	24	13	9
Precision	90%	38%	70%	-
Recall	100%	100%	100%	-

Table 1. Theory Exploration results. Note that IsaScheme was evaluated on a natural number theory also including exponentiation [16].

5 Related Work

Inductive theories do not allow cut-elimination and are thus undecidable. In practice, this means that auxiliary lemmas (themselves requiring an inductive proof) may be required to complete a proof. Inductive theorem provers which support some form of automated lemma discovery, such as ACL2's induction tactic [4], CLAM [9], IsaPlanner [8] and Zeno [19], use a *top-down* approach by which lemmas are discovered from failed proof-attempts. HipSpec differ from all of these in its *bottom-up* theory exploration approach. HipSpec automatically tries to discover a background theory for the relevant functions, building up something like the human-created lemma libraries available for interactive provers such as Isabelle [17] or ACL2 [13]. Experimental evaluation shows that HipSpec's bottom-up approach compares well in terms of finding the right lemmas. Some types of lemmas are difficult to discover in the top-down approach, for instance the generalised version needed to prove the theorem `rev xs = qrev xs []`, and many other similar theorems featuring accumulator variables. While the CLAM system could discover the `rev/qrev` generalisation given some other basic lemmas, HipSpec discovers it all automatically. Zeno, IsaPlanner and ACL2 do not support this type of lemma discovery at all, and thus fail on theorems of this kind. In HipSpec, there is always a risk of discovering extra irrelevant lemmas too. However, these may perhaps be useful in other proofs.

Both CLAM and IsaPlanner are based on the rippling heuristic for guiding rewriting of the step-case towards the inductive hypothesis. The advantage of rippling is that it guarantees termination of rewriting, and that rewrite rules may be used both ways around if need be. Rippling is a syntax-based heuristic, which may cause problems for instance on conjectures where a lot of case-analysis is required, as highlighted by Test Suite A in §4 where HipSpec, Zeno and ACL2 performed better than the rippling-based IsaPlanner. HipSpec relies on an off-the-shelf prover as backend which has no termination guarantee like rippling-based provers. Instead termination is enforced by using a timeout, which means that there is a risk of missing proofs which just take a little bit too long. When special-purpose rippling-based provers fail, the user may inspect the final proof state to see where the proof got stuck. HipSpec cannot currently.

While most other provers have some form of built-in rewriting tactics, HipSpec and the program verifier Dafny [14] instead send proof obligations to external automated provers. Like HipSpec, Dafny applies induction on the meta-level and passes the resulting proof obligations to the theorem prover Z3, which was also used as a backend for HipSpec in the experiments in this article. Dafny does not, however, support automated lemma discovery, so auxiliary lemmas must be supplied by the user. The obvious advantage is that off-the-shelf automated provers are often very fast and powerful. However, as the provers are treated as black boxes we do not get a readable proof, or any information if a proof fails. IsaPlanner checks proof steps in Isabelle and can produce readable output of complete or partial proofs. Zeno can output proofs in Isabelle format, which can then be re-checked in the proof assistant, ensuring correctness. Readable and checkable proofs are further work in HipSpec.

HipSpec is the only system which can be used both as an inductive theorem prover and as a theory exploration system. The IsaCoSy and IsaScheme theory explorers were developed for automating the creation of lemma libraries for inductive theories in Isabelle [12,16]. Both systems use IsaPlanner to prove conjectures that pass counterexample checking, but differ in the heuristics they use to generate conjectures. Experiments in which the outputs of IsaCoSy were *manually* fed back to IsaPlanner have been successfully performed [11]. However, neither is fully integrated with the theorem prover: IsaPlanner cannot call either of these *automatically* while proving user-given properties, In contrast, HipSpec is fully automatic. Both IsaCoSy and IsaScheme are considerably slower than HipSpec, although all three systems produce similar sets of lemmas.

6 Conclusion and Further Work

HipSpec is an automated inductive theorem prover and a theory exploration system. It takes a novel bottom-up approach to lemma discovery by using theory exploration to first build a richer background theory in which user-given properties are proved. In experimental evaluation, HipSpec performs very well in comparison with other systems: in particular, it succeeds in proving theorems about tail-recursive functions that require generalisations, which no other system can prove fully automatically without user-supplied lemmas. HipSpec also performs very well as a standalone theory exploration system, producing sets of lemmas with high precision and recall when compared to Isabelle's libraries. Furthermore, it does so in seconds rather than hours like previous systems.

Ultimately, we would like to use HipSpec in a tool for automatically proving properties of Haskell programs, making it usable by "normal" programmers, much like the popular QuickCheck tool [5]. In order to extend HipSpec to the full Haskell language we need to add support also for infinite and lazy data-structures and non-terminating functions in QuickSpec and in HipSpec's property language. The Haskell-to-FOL translation system HALO [20] already supports this, and Hip supports co-inductive reasoning and fixpoint induction. The theory-exploration machinery does however need to be extended to record which lemmas hold for all values of a type (including partial ones) and which ones only hold for completely-defined total values.

Another area of further work is providing user feedback from failed proofs, and producing checkable proofs. It could interesting to experiment with a different prover backend, from which information about failed proof attempts can be reclaimed, rather than treating the prover as a black box.

References

1. HipSpec evaluation results. `http://www.cse.chalmers.se/~danr/hipspec`.
2. HipSpec source code repository. `http://www.github.com/danr/hipspec`.
3. Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: meta-level guidance for mathematical reasoning.* Cambridge University Press, 2005.
4. Harsh Raju Chamarthi, Peter Dillinger, Panagiotis Manolios, and Daron Vroon. The ACL2 Sedan theorem proving system. In *Proceedings of TACAS*, pages 291–295, 2011.
5. Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of ICFP*, pages 268–279, 2000.
6. Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: Guessing formal specifications using testing. In *Proceedings of TAP*, pages 6–21, 2010.

7. Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of TACAS*, TACAS'08/ETAPS'08, pages 337–340. Springer-Verlag, 2008.
8. L. Dixon and J. D. Fleuriot. Higher order rippling in IsaPlanner. In *Proceedings of TPHOLs*, volume 3223 of *LNCS*, pages 83–98, 2004.
9. Andrew Ireland and Alan Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16:79–111, 1996.
10. Moa Johansson, Lucas Dixon, and Alan Bundy. Case-analysis for rippling and inductive proof. In *Proceedings of ITP*, pages 291–306, 2010.
11. Moa Johansson, Lucas Dixon, and Alan Bundy. Lemma discovery and middle-out reasoning for automated inductive proof. 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.
12. Moa Johansson, Lucas Dixon, and Alan Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 47(3):251–289, 2011.
13. Matt Kaufmann, Manolios Panagiotis, and J Strother Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, 2000.
14. K. Rustan Leino. Automating induction with an SMT solver. In *Proceedings of VMCAI*. Springer, 2012.
15. 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.
16. 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.
17. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
18. Dan Rosén. Proving Equational Haskell Properties using Automated Theorem Provers, 2012. MSc. Thesis, University of Gothenburg.
19. 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.
20. Dimitrios Vytiniotis, Dan Rosén, Simon Peyton Jones, and Koen Claessen. HALO: Haskell to logic through denotational semantics. In *Proceedings of POPL'13*. ACM, 2013.