HipSpec: Automating Inductive Proofs of Program Properties

Koen Claessen, Moa Johansson, Dan Rosén[*]and Nicholas Smallbone

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

Abstract

We present ongoing work on HipSpec, a system for automatically deriving and proving properties about functional programs. HipSpec uses a combination of theory exploration, counter-example testing and inductive theorem proving to automatically generate a set of equational theorems about recursive functions in a program, which are later used as a background theory for proving stated properties about a program. Initial experiments are encouraging; our initial HipSpec prototype already compares favourably to other, similar 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 while they are developing software and that supports them in the process. This paper describes our current progress made in this direction. In particular, we have worked on the problem of automating induction.

The context we are working in is strongly typed purely functional programming, notably using the programming language Haskell. There are two key advantages to using Haskell in this context: (1) pure functional programming is semantically simpler and thus easier to reason about than programming languages with side effects, (2) many Haskell programmers already use QuickCheck [5], a tool for property-based random testing, which means that many Haskell programs already have formal properties stated about them (albeit unproved, but tested).

The main obstacle one encounters when doing automatic verification is deciding where and how *induction* needs to be applied.

Let us look at a simple example. Consider the following Haskell program implementing the list reverse function in two different ways. The first is called `reverse`:

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

The second is called `qreverse`, and uses an accumulating parameter which leads to a function with better complexity:

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

qreverse xs = revacc xs []
```

A natural property one would like to state and verify of the above program is that reverse and qreverse implement the same function: $\forall$ `xs . reverse xs = qreverse xs`.

In previous work, we developed a tool called Hip, the Haskell Inductive Prover [15]. Hip reads in a Haskell program and a property, and tries to prove the property by various induction

[*] Corresponding Author

principles. For each induction principle, Hip generates a set of first-order proof obligations which are sent to an automatic first-order prover. Thus, Hip takes care of the inductive reasoning but delegates the pure first-order reasoning to an automatic prover. The version of Hip used in this paper only applies structural induction, so for the `reverse` property, it will try induction on `xs`. This will fail: the induction hypothesis `reverse as = qreverse as` is too weak to prove `reverse (a:as) = qreverse (a:as)`.

What is needed is one or more extra lemmas that make the proof go through, and can themselves be proven by induction. The challenge of automatic induction is to come up with such lemmas automatically. Most automated induction provers take a top-down approach: they start by trying to prove the inductive step, somehow get stuck, and use the stuck state to come up with a more general lemma that might help the proof of the inductive step go through.

Our approach differs in that we use a bottom-up approach. Our tool, called HipSpec, reads in a program and first, before even looking at the properties, makes a list of conjectures about the program. Each conjecture is then submitted to Hip to try to find an automatic induction proof. As soon as a conjecture is proved, it is given as a lemma to Hip, so that it may be used in proving more conjectures. When as many conjectures as possible have been proved, Hip tries to prove the actual properties stated by the programmer, using all the lemmas it has now proved.

How do we come up with the conjectures? We use another tool we developed in previous work, called QuickSpec [6]. QuickSpec generates conjectures in the form of algebraic equations about the functions in a given program. It uses random testing, so equations are sometimes false. However, we have a kind of completeness: QuickSpec will discover all true equations up to a given term depth. This completeness gives us a good chance of finding useful lemmas.

For our example, QuickSpec takes about 3 seconds to run. HipSpec feeds QuickSpec's equations into Hip, which proves them. Many of the equations are proved without induction, and thus cannot be useful lemmas; the equations that needed induction are shown below.

No	Conjecture		Lemmas used
(*1*)	`xs`	`= xs++[]`	
(*2*)	`xs++(ys++zs)`	`= (xs++ys)++zs`	
(*3*)	`revacc xs (ys++zs)`	`= revacc xs ys++zs`	
(*4*)	`revacc xs ys`	`= reverse xs++ys`	(*1*), (*3*)
(*5*)	`revacc ys xs++zs`	`= revacc (revacc xs ys) zs`	(*3*)

Now Hip attempts to prove the original property, using the lemmas above, and easily succeeds: the property follows directly from (*4*) and the definition of `qreverse`, and no induction is needed. Note that equation number (*5*) was proved but was never needed for proving the original property. Proving unnecessary things is an obvious risk of our approach.

Contributions We augment the automated induction landscape with a new method which uses a bottom-up theory exploration approach to find auxiliary lemmas. Our bottom-up approach combines our own earlier work on conjecture generation based on testing (QuickSpec) and induction principle enumeration (Hip). Our hypothesis is that algebraic equations constructed from terms up to a certain depth form a rich enough background theory for proving many algebraic properties about programs, and that a reasoning system for functional programs can be built on top of an automatic first-order theorem prover. The experimental results in this paper have so far confirmed this.

2 HipSpec

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

Hip is an automatic tool for proving user-stated equality properties about Haskell programs. This is done by compiling the program to first-order logic. Hip applies induction rules to the conjectures, yielding first-order proof obligations, which are proved using off-the shelf automated first-order theorem provers. 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 only apply structural induction; as a standalone tool, it also supports coinductive proof techniques such as fixed point induction. The focus of our work 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.

QuickSpec conjectures equations about a functional program by 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, and a depth limit. QuickSpec generates the set of all type-correct terms whose depth is less than the limit, built from the functions and variables given, which we call the universe. It then partitions the universe into equivalence classes by random testing: two terms will be in the same equivalence class unless a test case distinguished them. From this equivalence relation we can generate a vast set of equations about the tested program, which completely characterises the universe of terms. However, there will be very many equations, most of them redundant. Therefore, QuickSpec also includes a pruning phase, which leaves behind a minimal core from which the rest of the equations follow by pure equational reasoning; this minimal core is presented to the user. However, QuickSpec may prune out equations that are useful as lemmas in inductive proofs, so HipSpec must also consider the full set of equations.

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

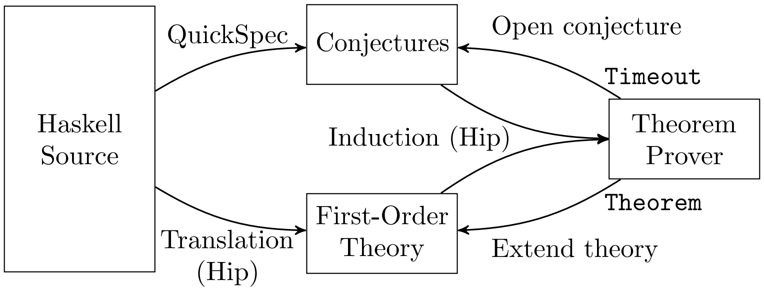

Figure 1: An overview of HipSpec.

HipSpec maintains two sets of equations: *conjectures*, which we suspect are true but have not proved, and *lemmas*, which we have proved. The *first-order theory* in Figure 1 consists of Hip's translation of our program plus the current set of lemmas. Initially the conjectures consist of *all* equations that QuickSpec found—even those that would be removed by pruning. Our main loop is as follows:

1. Pick a conjecture from the conjecture set.
2. If the conjecture follows by equational reasoning only, discard it. Otherwise, ask Hip to prove the conjecture by induction, using previously proved lemmas and definitions as background theory.
3. If Hip succeeds within a timeout, move the conjecture to the lemma set.

The loop ends either when all conjectures have been proved, or all conjectures have been tried and failed since the last time the lemma set changed.

Picking a conjecture The performance of HipSpec completely depends on one heuristic: which conjecture to try to prove next. Our current heuristics are rather crude; more sophisticated techniques are further work. We define the *desirability* of an equation by the following total order:

- An equation that is in QuickSpec's list of equations surviving pruning is more desirable than an equation that is not.
- Otherwise, a simpler equation is more desirable than a complicated equation, using a simplicity metric that is built into QuickSpec.

We maintain a set of *failed* conjectures, ones that we have tried and failed to prove, and a set of *active* conjectures, that we will try to prove. Our basic heuristic is as follows:

- Always pick the most desirable conjecture in the active set.
- If the active set is empty, move all failed conjectures to the active set.
  (The main loop above will terminate when this process is no longer productive.)

The definition above roughly means that we first try to prove the set of equations that QuickSpec would normally present to the user, then all the equations that QuickSpec's pruning algorithm discarded, in order of simplicity.

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". Adding new lemmas is somewhat expensive, but we can do this incrementally as we prove them.

Whenever we pick a conjecture, we check if this prover can prove it from the current lemmas. If so, we just discard it. This filters out most trivial conjectures.

Candidates With the heuristic above, if we fail to prove a conjecture, we will never retry it until we have tried *all* the active conjectures. This is bad: we would like to retry the conjecture once we have proved a relevant lemma. The question is: which lemmas are relevant?

We observe that, if we are given an equation F and a relevant lemma E, sometimes F implies E, given the lemmas we have already proved. For example, to prove the law `x*y = y*x` we need the lemma `x+(x*y) = x*S y`: if we are given the definition of multiplication `S x*y = y+(x*y)`, this lemma follows from `x*y = y*x`.[1]

Accordingly, we define the following heuristic. If we have just proved equation E in background Γ, and we have a failed equation F such that $\Gamma, F \vdash E$ *without induction*, then we move F to the active set and try to prove it next. In the example above, E is `x+(x*y) = x*S y`, F is `x*y = y*x`, and we must have already proved `S x*y = y+(x*y)`, which will be in Γ; then $\Gamma, F \vdash E$, suggesting that we should try to prove `x*y = y*x` again. The heuristic is somewhat unreliable, but can be implemented efficiently using QuickSpec's built-in equational reasoning as mentioned above.

[1] Another example is `x+y = y+x`: we need the lemma `x+0 = 0+x`, a special case of commutativity. Yet another is `reverse (reverse xs ++ ys) = reverse ys ++ xs`, needed to prove `reverse (reverse xs) = xs`.

3 Examples

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. It states that the sum of the n first cubes is the nth triangle number squared:

$$\sum_{k=1}^{n} k^3 = \left(\sum_{k=1}^{n} k\right)^2$$

The `tri` function below calculates the triangle numbers and the sum of cubes by `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[2]:

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

When HipSpec is given the plus, multiplication, `tri` and `cubes`, it generates and proves the properties listed in Figure 2 below. The properties are listed in the order they were proved.

No	Conjecture	Lemmas used
(1)	`x       = x+Z`	
(2)	`Z       = x*Z`	
(3)	`S (x+y) = x+S y`	
(4)	`x+y     = y+x`	(1), (3)
(5)	`x+(y+z) = y+(x+z)`	(4)
(6)	`x*(y+z) = (x*z)+(x*y)`	(4), (5)
(7)	`x*(y+y) = (x+x)*y`	(4), (5)
(8)	`x       = x*S Z`	
(9)	`x+(x*y) = x*S y`	(1), (3), (5)
(10)	`x*y     = y*x`	(2), (9)
(11)	`x*(y*z) = z*(x*y)`	(4), (6), (10)
(12)	`x+(x*x) = tri x+tri x`	(3), (4), (5), (8), (9)
(13)	`cubes x = tri x*tri x`	(12) and more

Figure 2: Properties proved about the theory with natural number addition, multiplication, triangle numbers (`tri`) and sum of cubes (`cubes`). `Z` represents zero and `S` successor.

HipSpec first proves that the naturals form a commutative semiring. Only induction on one variable was used in this test. This means that, for instance, to prove (*4*), commutativity of addition, then (*1*) and (*3*) are required. Some superfluous lemmas are proved, for example (*7*), which is never used and follows from commutativity (*10*) and distributivity (*6*). In (*12*), the well-known identity, $\sum_{k=1}^{n} k = n(n+1)/2$ is proved. From this lemma HipSpec proves Nicomachus' Theorem.

3.1 Binary Arithmetic

Peano arithmetic is not very realistic. A better way to implement the naturals is as a list of bits:

```
data Bits = Zero | ZeroAnd Bits | OneAnd Bits
```

[2] Where `=:=` is HipSpec notation for equality

Here, the constructor `ZeroAnd` *n* represents the number $2n$, while `OneAnd` *n* represents $2n+1$. Thus 6 can be represented as `ZeroAnd (OneAnd (OneAnd Zero))`. Arithmetic operations can be defined in the usual messy way:

```
succ Zero        = OneAnd Zero
succ (ZeroAnd x) = OneAnd x
succ (OneAnd x)  = ZeroAnd (succ x)

plus Zero xs = xs
plus xs Zero = xs
plus (ZeroAnd xs) (ZeroAnd ys) = ZeroAnd (plus xs ys)
plus (ZeroAnd xs) (OneAnd ys)  = OneAnd (plus xs ys)
plus (OneAnd xs)  (ZeroAnd ys) = OneAnd (plus xs ys)
plus (OneAnd xs)  (OneAnd ys)  = ZeroAnd (succ (plus xs ys))
```

We can specify `succ` and `plus` by relating them to the corresponding functions on Peano numbers by means of a function `toNat :: Bits -> Nat`:

```
toNat Zero        = Z
toNat (ZeroAnd x) = toNat x + toNat x
toNat (OneAnd x)  = S (toNat x + toNat x)

prop_succ n = toNat (succ n) =:= S (toNat n)
prop_plus m n = toNat (plus m n) =:= toNat m + toNat n
```

HipSpec is able to prove both `prop_succ` and `prop_plus`, along with the lemmas about Peano arithmetic from the last section, and these generated lemmas:

Conjecture		
`plus xs ys`	=	`plus ys xs`
`s (plus xs ys)`	=	`plus xs (s ys)`
`S (toNat xs)`	=	`toNat (s xs)`
`ZeroAnd (plus xs xs)`	=	`plus (ZeroAnd xs) (plus xs xs)`
`plus xs (plus ys zs)`	=	`plus zs (plus xs ys)`
`toNat (plus xs ys)`	=	`toNat ys+toNat xs`
`toNat (ZeroAnd xs)`	=	`toNat (plus xs xs)`

3.2 Lists with Efficient Append

Another representation of lists is as a type with three constructors: the empty list, a singleton list and the concatenation of two lists, with the invariant that a list has no internal `Nil` nodes.

```
data List a = Nil | Unit a | Append (List a) (List a)
```

We can efficiently concatenate such lists as well as convert them to normal lists:

```
Nil +++ xs  = xs
xs  +++ Nil = xs
xs  +++ ys  = Append xs ys

toList Nil            = []
toList (Unit x)       = [x]
toList (Append xs ys) = toList xs ++ toList ys
```

Let us implement a function to reverse such lists, followed by some properties:

```
reverseL Nil            = Nil
reverseL (Unit x)       = Unit x
reverseL (Append xs ys) = Append (reverseL ys) (reverseL xs)

prop_append xs ys = toList (xs +++ ys) =:= toList xs ++ toList ys
prop_reverse xs = toList (reverseL xs) =:= reverse (toList xs)
```

HipSpec proves both properties. The first requires the lemma `xs ++ [] = xs`, and the second requires `reverse (xs ++ ys) = reverse ys ++ reverse xs`, which are automatically generated and proved as before.

4 Evaluation

We have evaluated HipSpec on two test suites from the inductive theorem proving literature. For fairness, we did not allow Hip to use equations from the test suite as lemmas when proving other properties: we only use the lemmas that are automatically generated by HipSpec. As the test suites contain many unrelated functions, we split them up by hand into independent parts. This was quite mechanical and we hope to automate it in the future. The results of our evaluation are available online[3].

Test Suite A This consists of 50 theorems about lists and natural numbers, and was used to evaluate techniques for discovering auxiliary lemmas and generalisations in the now-defunct CLAM prover [8]. There are no function definitions provided with the test suite so we have implemented them in Haskell in a standard fashion. Of the 50 properties, 38 are equational and so within the scope of HipSpec. Of these, HipSpec proves 36. The two that we cannot prove are:

No	Conjecture
T14	`ordered (isort xs) = True`
T50	`count x (isort xs) = count x xs`

Both require conditional lemmas, which HipSpec cannot yet discover: T14 requires `ordered xs` → `ordered (insert x xs)`, and T50 needs `x /= y` → `count x (insert y ys) = count x ys`.

Test Suite B This consists of 85 theorems about lists, natural numbers and binary trees [9]. It originates from IsaPlanner and was later translated into Haskell[4] as the test suite for the Zeno prover [16]. 71 of the properties in the test suite are equational. Of these, HipSpec proves 67. However, only 15 require any lemmas at all: 52 are proved without theory exploration, and 10 are even proved without induction[5]. The four unproved properties are:

No	Conjecture
53	`count n xs                  = count n (sort xs)`
66	`len (filter p xs) <= len xs = True`
68	`len (delete n xs) <= len xs = True`
78	`sorted (sort xs)            = True`

There is a slight overlap of the two test suites: property 53 and 78 above are the same as T50 and T14 respectively from Test Suite A. Properties 66 and 68 in Test Suite B, unproved by HipSpec, need the conditional lemma expressing transitivity of `<=` to be proved.

Test Suite B has been evaluated on IsaPlanner[6], Zeno, ACL2s [4] (by the authors of Zeno) and Dafny [12]. Dafny proves 45, IsaPlanner proves 47, ACL2s proves 74 and Zeno proves 82, failing on three properties. Two of these are equational, namely number 72 and 74:

[3] `http://www.cse.chalmers.se/~koen/hip/`

[4] `http://www.doc.ic.ac.uk/~ws506/tryzeno/comparison/`

[5] This is because these problems were originally designed only for evaluating rippling in IsaPlanner on problems involving if- and case-expressions, which are expressed as higher-order functions in IsaPlanner's logic.

[6] `http://dream.inf.ed.ac.uk/projects/lemmadiscovery/results/case-analysis-rippling.txt`

No	Conjecture
72	`rev (drop i xs) = take (len xs - i) (rev xs)`
74	`rev (take i xs) = drop (len xs - i) (rev xs)`

None of the other tools can prove the two properties above, but HipSpec can. Both properties have similar proofs; let us look at number 72. HipSpec discovers the five lemmas below:

No	Conjecture
1	`len (drop x xs) = len xs-x`
2	`len xs          = len (rev xs)`
3	`xs              = take x xs++drop x xs`
4	`rev (ys++xs)    = rev xs++rev ys`
5	`xs              = take (len xs) (xs++ys)`

With these lemmas, the proof proceeds by equational reasoning as follows:

```
rev (drop x xs)                                                    = {5}
take (len (rev (drop x xs))) (rev (drop x xs) ++ rev (take x xs))  = {2}
take (len (drop x xs)) (rev (drop x xs) ++ rev (take x xs))        = {1}
take (len xs - x) (rev (drop x xs) ++ rev (take x xs))             = {4}
take (len xs - x) (rev (take x xs ++ drop x xs))                   = {3}
take (len xs - x) (rev xs)
```

Execution time We can roughly partition the running time in three:

- First, QuickSpec explores the theory by testing. This generally takes just a few seconds.
- Second, we try to prove QuickSpec's conjectures. This is where most of the time is spent, and unfortunately much of the time is spent in failed proof attempts where we have not yet proved the required lemmas. Each failed proof attempt will take at most one second, that being the timeout we gave the first-order prover.
- Finally, we prove the user-stated properties. In our evaluation this means the properties in the test suite. For the theorems we managed to prove this is almost instantaneous: QuickSpec already invents the properties itself and we prove them in the previous stage.

The total runtime varied according to the testsuite. Typical runtimes were on the order of one minute. However, some of the larger parts took as long as half an hour.

5 Related Work

Inductive theories are undecidable, and proofs are thus often performed interactively, for instance using proof assistants such as ACL2 [11] and Isabelle[14]. In particular, the discovery of auxiliary lemmas or generalisations has to be performed by a human user.

Proof-planning is an automated technique which exploits the fact that certain families of proofs share a similar structure [2]. For instance, all inductive proofs consists of one or more base-cases followed by step-cases. Rippling is a commonly used heuristic for guiding rewriting in the step-case in proof-planning [3]. If some step in the proof-plan fails, *proof-critics* may be employed to analyse the failure and perhaps suggest a missing lemma or generalisation [8]. Proof critics for different kinds of failures of rippling were implemented in the now defunct CLAM proof-planner. Using these critics, CLAM succeeds in finding generalisations for proofs involving accumulator variables, such as the example from §1: ∀ `xs . reverse xs = qreverse xs`. IsaPlanner is a current proof-planner built on top of the Isabelle proof assistant [7]. It supports rippling and critics for lemma discovery, but not for finding generalisations and can thus not prove properties about functions defined using accumulator variables. The main difference between our work and the proof critics approach is that critics work top-down, trying to derive a missing lemma from a particular failed proof attempt. HipSpec on the other hand, works bottom-up, first it tries theory exploration to construct lemmas about available functions, then it tackles the user defined properties of interest. While proof-critics are tied to react to a particular failure pattern of a particular proof-plan, theory exploration techniques can be used in a more flexible manner, in combination with various provers or proof techniques. However, theory exploration may also come up with irrelevant lemmas, while the critics approach is more targeted towards a particular lemma suitable for the current proof attempt.

Proof assistants often have large libraries of previously proved lemmas for the user to build further upon. The theory exploration systems IsaCoSy and IsaScheme were developed for automating the creation of such lemma libraries for inductive theories in Isabelle [10, 13]. Both systems use IsaPlanner to prove conjectures which pass counter-example checking, but differ in the heuristics they use to generate conjectures. IsaCoSy and IsaScheme make one pass at generating a background theory, if some conjecture cannot be proved, it is left open. However, as IsaPlanner has a lemma calculation critic, simple lemmas can be derived on-the-fly. HipSpec on the other hand, relies only on the lemmas it has previously discovered and proved. This allows failed proofs to be revisited later, after suitable lemmas has been proved.

Zeno is an automated theorem prover for a subset of Haskell [16]. It performs proofs by structural induction and rewriting and case analysis. Like IsaPlanner, it can discover simple lemmas by applying common subterm generalistion to subgoals to which no rewriting steps apply, followed by a new induction on the resulting conjecture.

Dafny is a program verifier with a simple automated induction tactic [12]. It uses heuristics for selecting properties which may require proof by induction. Like HipSpec, it then applies induction on the meta-level and pass the resulting proof obligations to the SMT solver Z3. Dafny does not support automated lemma discovery, so auxiliary lemmas must be supplied by the user.

6 Conclusion and Future Work

We have introduced HipSpec, a tool that uses a form of theory exploration to build a background theory about a Haskell program, in order to automatically prove stated properties by induction. In the experimental evaluation, the tool compares favourably against existing similar systems.

The main difference with most existing methods is that we use a bottom-up approach when building the background theory. A possible advantage over top-down, syntactic methods is that our approach will not get stuck as easily. A possible drawback is that we might generate far too many irrelevant lemmas, and waste time proving them. Our initial experiments show that this is often not the case; it is relatively cheap to generate and prove the conjectures. Choosing appropriate algorithms and data structures to discharge lemmas provable without induction quickly plays a key role in this.

For future work, we will explore heuristics for in which order to try to prove conjectures. We should exploit syntactic information, such as the call graph. When proving a conjecture fails, we should postpone trying to prove similar conjectures. There is a world of heuristics here.

We plan to extend HipSpec to the full Haskell language, in particular by allowing non-terminating functions and exceptions. Both QuickSpec and Hip already deal with exceptions. Hip can also deal with non-terminating functions; what we have not completely worked out is how to practically distinguish between lemmas that hold for all values of a datatype (including partial values), and lemmas that only hold for completely defined, total values.

The properties we deal with right now are universally quantified equalities between expressions. We would like to extend these to other kinds of properties, in particular conditional properties. The main difficulty is getting QuickSpec to come up with conditional equations automatically.

Hip supports several induction principles apart from structural induction, notably Plotkin's fixed point induction; we would like to support these in HipSpec.

Finally, for HipSpec to be usable by programmers, we need to give useful feedback when we cannot prove a property. This feedback should help the programmer to decide e.g. if a lemma should be added by hand or if QuickSpec should be given more function definitions. We plan to incorporate ideas from our earlier work on non-standard counter-examples here [1].

References

[1] Jasmin Blanchette and Koen Claessen. Generating counterexamples for structural inductions by exploiting nonstandard models. In *LPAR*. Springer LNCS, 2010.

[2] Alan Bundy. The use of explicit plans to guide inductive proofs. In *CADE*, pages 111–120, 1988.

[3] Alan Bundy, David Basin, Dieter Hutter, and Andrew Ireland. *Rippling: meta-level guidance for mathematical reasoning*. Cambridge University Press, New York, NY, USA, 2005.

[4] Harsh Raju Chamarthi, Peter Dillinger, Panagiotis Manolios, and Daron Vroon. The ACL2 sedan theorem proving system. In *TACAS*, pages 291–295, 2011.

[5] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, 2000.

[6] Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: Guessing formal specifications using testing. In *TAP*, pages 6–21, 2010.

[7] L. Dixon and J. D. Fleuriot. Higher order rippling in IsaPlanner. In *Theorem Proving in Higher Order Logics*, volume 3223 of *LNCS*, pages 83–98, 2004.

[8] Andrew Ireland and Alan Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16:16–1, 1995.

[9] Moa Johansson, Lucas Dixon, and Alan Bundy. Case-analysis for rippling and inductive proof. In *ITP*, pages 291–306, 2010.

[10] Moa Johansson, Lucas Dixon, and Alan Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 47(3):251–289, 2011.

[11] Matt Kaufmann, Manolios Panagiotis, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[12] K. Rustan Leino. Automating induction with an SMT solver. To appear at VMCAI 2012.

[13] 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.

[14] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[15] Dan Rosén. Proving Equational Haskell Properties using Automated Theorem Provers, 2012. MSc. Thesis, University of Gothenburg.

[16] Willam Sonnex, Sophia Drossopoulou, and Susan Eisenbach. Zeno: A tool for the automatic verification of algebraic properties of functional programs. Technical report, Imperial College London, February 2011.