Electronic Communications of the EASST
Volume 72 (2015)

Proceedings of the
15th International Workshop on
Automated Verification of Critical Systems (AVoCS 2015)

Conditional Lemma Discovery and Recursion Induction in Hipster

Irene Lobo Valbuena and Moa Johansson

15 pages

Guest Editors: Gudmund Grov, Andrew Ireland
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Conditional Lemma Discovery and Recursion Induction in Hipster

Irene Lobo Valbuena and Moa Johansson

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

Abstract: Hipster is a theory exploration tool for the proof assistant Isabelle/HOL. It automatically discovers lemmas about given recursive functions and datatypes and proves them by induction. Previously, only equational properties could be discovered. Conditional lemmas, for example required when reasoning about sorting, has been beyond the scope of theory exploration. In this paper we describe an extension to Hipster to also support discovery and proof of conditional lemmas.

We also present a new automated tactic, which uses *recursion induction*. Recursion induction follows the recursive structure of a function definition through its termination order, as opposed to structural induction, which follows that of the datatype. We find that the addition of recursion induction increases the number of proofs completed automatically, both for conditional and equational statements.

Keywords: theory exploration, automated induction, interactive theorem proving

1 Introduction

Theory exploration is a technique for automatically discovering new interesting lemmas in a formal mathematical theory development. These lemmas are intended to help constructing a richer background theory about the concepts at hand (e.g. functions and datatypes) which can be useful both to enhance the power of automation as well as being of use in interactive proofs [MBA07, JDB11, MMDB12]. Theory exploration has proved particularly useful for automation of inductive proofs [CJRS13]. This work builds on Hipster [JRSC14], an interactive theory exploration system for the proof assistant Isabelle/HOL [NPW02]. It can be used in two modes, either *exploratory mode* to generate a set of basic lemmas about given datatypes and functions, or in *proof mode*, where it assists the user by searching for missing lemmas needed to prove the current subgoal. To generate conjectures, Hipster uses as a backend the HipSpec system, a theory explorer for Haskell programs [CJRS13]. Proofs are then performed by specialised tactics in Isabelle/HOL. Hipster has been shown capable of discovering and proving standard lemmas about recursive functions, thus speeding up theory development in Isabelle. However, lemma discovery by theory exploration has previously been restricted to equational properties. In this paper we take the first steps towards lifting this restriction and exploring also conditional conjectures. Conditional lemmas are necessary if we for example want to prove properties about sorting algorithms. As an example, consider the proof of correctness for insertion sort:

```
theorem isortSorts: "sorted (isort xs)"
```

To prove this theorem by induction will in the step-case require a lemma telling us that if a list is sorted, it remains so after an additional element is inserted:

$$\texttt{lemma "sorted xs} \Longrightarrow \texttt{sorted (insert x xs)"}$$

Discovering this kind of conditional lemmas introduces a big challenge for theory exploration. First of all, the search space greatly increases: what statements should be picked as potentially interesting side-conditions to explore? Secondly, as our theory exploration system relies on generation of random test-cases, we also need to ensure that we perform tests where the condition evaluates to true, otherwise the system might miss some conditional equations (Example 2, p. 4).

As Hipster is designed as an interactive system, we avoid the first problem by asking the user to specify under which condition theory exploration should occur. In the example above, this would require the user to tell Hipster that the predicate `sorted` is an interesting pre-condition, in addition to which function symbols should be explored in the bodies of lemmas. The rest of the process is however automatic. We describe it in more detail in §3

The second contribution of this paper is a new automated tactic for *recursion induction* (see e.g. §3.5.4 of [NPW02]). Previously, Hipster only supported structural induction over the datatypes, but has now been extended with a new tactic that uses recursion induction, following the termination order of function definitions instead of the datatype. This has shown to be useful for many proofs that previously failed, but can also provide shorter proofs in some cases. The new recursion induction tactic is described in §3.2. It is used by Hipster during automated theory exploration, but can equally well be applied as a powerful regular tactic by a human user working in Isabelle.

2 Hipster

This section provides a description of how Hipster works and how its subsystem QuickSpec generates conjectures.

2.1 Theory Exploration in Hipster

Figure 1 gives an overview of the Hipster system. Starting from an Isabelle theory file that defines a set of datatypes and functions, the user calls Hipster on a list of functions about which she is interested in finding lemmas. The workings of Hipster can be divided up into three stages:

1) Generation of Haskell code.

2) Theory exploration in Haskell.

3) Proof in Isabelle.

Hipster uses Isabelle's code generator [HN10], to translate the theory to a Haskell program. Hipster then employs the theory exploration system HipSpec as a backend for generating conjectures. While HipSpec can be used also as a fully fledged theorem prover, Hipster only uses its conjecture generation subsystem QuickSpec [CSH10], and performs proofs inside Isabelle. Isabelle is an LCF-style prover, which means that it is based on a small core of trusted axioms, upon which subsequent proofs must be built. Therefore, any proofs found outside Isabelle, e.g. by HipSpec, would have to be reconstructed inside Isabelle anyway. Hence it is easier for Hipster to simply use Isabelle for proofs in the first place.

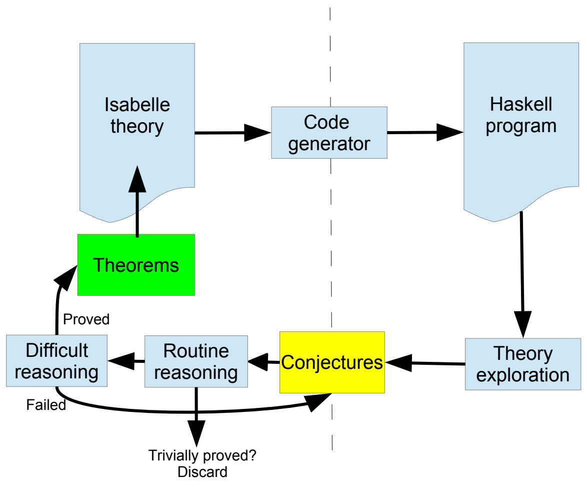

Figure 1: Overview of Hipster

Not all conjectures returned from QuickSpec are interesting. Hipster is parametrised by two tactics, which can be set by the user: one for *routine reasoning* and one for *difficult reasoning*. Conjectures solved by routine reasoning are deemed trivial and discarded, while those requiring more difficult reasoning are displayed to the user and included in the Isabelle theory so they can be used in subsequent proofs if necessary. In the context of this paper, routine reasoning is first-order equational reasoning and simplification, whilst difficult reasoning involves some kind of induction. If a conjecture is not immediately provable, Hipster will place it at the end of the list of open conjectures and will try it again if it has found some additional lemmas. Occasionally, Hipster might discover some conjecture which it does not manage to prove automatically, because not even its tactic for difficult reasoning is strong enough. Such an open conjecture would also be displayed to the user, who can then choose to perform an interactive proof in Isabelle, perhaps employing other tactics or lemmas than those currently available to Hipster.

2.2 Conjecture Generation in QuickSpec

QuickSpec takes as input a set of functions and variables (by default three per type), and generates all type-correct terms up to a given limit (by default depth three). The number of variables and term-depth limit can be adjusted by the user. QuickSpec then proceeds to divide the generated terms into equivalence classes, so that each equivalence class eventually represents a set of equations. Initially, all terms of the same type are in the same equivalence class. QuickSpec then uses QuickCheck [CH00], to generate random ground values for the variables in the terms, and evaluates the result. If two terms in an equivalence class turn out to evaluate differently, the equivalence class is split accordingly. The process is then repeated until the equivalence classes stabilise (after several hundred different random tests), which means that we usually have quite a high confidence in that the conjectures produced are probably true, even though they are not yet proved.

Example 1. As a small example, consider a theory exploration attempt where we have asked Hipster for lemmas about a function `isort` implementing insertion sort. Among the terms generated by QuickSpec are those in the table below. Initially, all terms are placed in the same equivalence class. Suppose QuickSpec generates the random value `xs` $\rightarrow$ `[3,1]`.

	Term	Ground Instance	Value
1	`isort xs`	`isort [3,1]`	`[1,3]`
2	`isort (isort xs)`	`isort (isort [3,1])`	`[1,3]`
3	`xs`	`[3,1]`	`[3,1]`

As not all terms evaluate to the same value, they should no longer be in the same equivalence class. We thus split the terms into two new equivalence classes: terms 1 and 2 evaluate to the same value and remain together, while term 3 is separated. After this, no subsequent tests further split these equivalence classes, and we can read off the equation: `isort(isort xs)` $=$ `isort xs`.

3 Conditional Lemmas and Recursion Induction

We now demonstrate how to employ Hipster interactively for theory exploration of conditional lemmas in the development of a theory.We first explain how conditional conjectures are generated in QuickSpec. We then explain our new automated induction tactic for recursion induction, and finally show how Hipster combines these in a case study proving the correctness of insertion sort.

3.1 Generating Conditional Conjectures

The support in QuickSpec for generating conditional conjectures (implications) is still rather basic. In this case, QuickSpec will in addition to the other input require the user to specify a predicate to use as the premise of an implication. Term generation proceeds as described above, but testing takes the given predicate into account. Here, we are only interested in tests with values that make the premise true, otherwise we may split the equivalence classes when they should not be split. QuickCheck uses special functions called *generators* to produce random values of a given type. If using QuickSpec directly in Haskell, the user can program special purpose generators that could be made to only produce values satisfying a given predicate. In Hipster, however, these generator functions are simpler as they have to be automatically derived together with the Haskell code. Tests not satisfying the premise are simply discarded during conditional exploration, which means that we typically must generate more tests than for equational conjectures. Also, the risk of some non-theorem slipping through is slightly higher, but as Hipster then attempts to prove all conjectures, such a statement would be caught in the proving phase. Automatically generating customised generator functions is further work.

Example 2. In Example 1, we showed how QuickSpec generated equational conjectures about the insertion sort function `isort`. We are furthermore interested in the case with the condition that the predicate `sorted` holds (for one variable). In this case, QuickSpec first performs one pass looking for plain equations, as in Example 1, then a second where it considers the condition `sorted xs`. In this second phase, QuickSpec performs a new exploration, this time requiring the predicate sorted xs to hold for all test values. Suppose we test with the sorted list: xs $\rightarrow$ [1,2] (other non-sorted values for xs would be discarded).

	Term	Ground Instance	Value
1	isort xs	isort([1,2])	[1,2]
2	isort (isort xs)	isort (isort [1,2])	[1,2]
3	xs	[1,2]	[1,2]

This time, all terms evaluate to the same value on all tests where the list is sorted, so all three terms remain in the same equivalence class. QuickSpec realises that there is no point producing the conjecture sorted xs $\Longrightarrow$ isort (isort xs) = xs, as this is subsumed by the non-conditional equation discovered in the first phase. It will however produce the additional conjecture sorted xs $\Longrightarrow$ isort xs = xs, which clearly only holds if the list is already sorted.

3.2 Automating Recursion Induction

A recursion induction scheme is derived from a function's recursive definition. Unlike structural induction, the recursion induction scheme corresponds to the originating definition, and hence, the cases considered in its simplification rules.

When defining a recursive function over an inductive datatype one might traverse arguments following a different pattern to the strictly structural one (the one arising from a datatype's definition). This pattern could be more specific, or even less so, than following the datatype.

For instance, take the functions on lists sorted and last:

```
fun sorted :: "Nat List ⇒ Bool" where
  "sorted [] = True"
| "sorted ([x]) = True"
| "sorted (x1 # (x2 # xs)) = (x1 ≤ x2 & sorted (x2 # xs))"

fun last :: "'a List ⇒ 'a" where
  "last ([x]) = x"
| "last (x # xs) = last xs"
```

From these definitions' structures one can derive a new induction principle. Structural induction on lists considers the base-case [] (the empty list) and step-case x # xs (a list with an element inserted at the front). In the case of sorted, cases are broken down into more detailed ones by including an additional base-case [x] (the singleton list) and restricting the step-case to lists with at least two elements x1 # x2 # xs. Meanwhile last is not defined for the case [] and hence partially defined, meaning the induction scheme it gives rise to is to be completed with such a case. This, in fact, results in the same recursion induction scheme derived from sorted:

SORTEDIND
$$\frac{P\,([]) \qquad \forall x\;\; P\,([x]) \qquad \forall x,\, y,\, xs\;\; (P\,(x\,\#\,xs) \implies P\,(y\,\#\,(x\,\#\,xs)))}{\forall x\;\; P\,(x)}$$

LASTIND
$$\frac{\forall x\;\; P\,([x]) \qquad \forall x,\, y,\, xs\;\; (P\,(x\,\#\,xs) \implies P\,(y\,\#\,(x\,\#\,xs))) \qquad P\,([])}{\forall x\;\; P\,(x)}$$

Induction determined by these schemata is called *recursion induction* or *computation induction*. They can isolate sub-units not represented in a datatype's structure as being atomic, such as lists with at least two elements in the scheme for sorted. Recursion induction hence provides an immediate and more specific structure for reasoning about other recursion patterns where a simple structural induction might fail to set appropriate base and step-cases for the induction to succeed.

Within Isabelle/HOL these schemata are automatically derived and proved as theorems from recursive functions' termination order, and hence guaranteed to be sound [KST+11].

Example 3: Recursion Induction in a Proof. To exemplify the potential difference between recursion and structural induction, let us take the already introduced conditional lemma sorted xs $\Longrightarrow$ sorted (insert x xs). Applying structural induction on the list xs would produce the subgoals:

```
1. sorted [] ⟹ sorted (insert x [])
2. ⋀ y ys. (sorted ys ⟹ sorted (insert x ys)) ⟹ sorted (y # ys) ⟹
        sorted (insert x (y # ys))
```

Whilst sorted's recursion induction scheme would yield:

```
1. sorted [] ⟹ sorted (insert x [])
2. ⋀ y. sorted [y] ⟹ sorted (insert x [y])
3. ⋀ y1 y2 ys. (sorted (y2 # ys) ⟹ sorted (insert x (y2 # ys))) ⟹
        sorted (y1 # y2 # ys) ⟹ sorted (insert x (y1 # y2 # ys))
```

The latter set of subgoals leads to an immediate proof of the main lemma thanks to its steps mirroring the actual predicate definition, hence having a correspondence with its simplification rules. In contrast, the former, even though it intuitively looks immediate to prove, is not sufficiently generalised nor does it specify any intermediate result on inserting an element on a concrete non-empty list (in our case, the singleton list) which would enable to prove the second subgoal for any arbitrary list. Structural induction is in some way a weaker scheme and additional case-splits or lemmas would be required to close the proof employing it in our example.

A New Induction Tactic for Hipster

We have implemented a new automated tactic, called hipster_induct_schemes, for induction in Isabelle. This tactic searches not only for proofs by structural induction, but may also employ recursion induction when appropriate. It is designed for Hipster to use as its "difficult reasoning" component, but human users may of course also employ this tactic in interactive proofs.

The tactic first tries structural induction using the induction scheme associated with the datatype(s) of variables in the problem. If this fails, the tactic then tries recursion induction, using the induction schemes associated with the functions occurring in the problem. When instantiating recursion induction schemes with variables of the problem, more complete instantiations are considered first. This leaves less specific partial instantiations to be tried later. For each attempted induction, the tactic will apply Isabelle's simplifier followed by (if necessary) first-order reasoning using Isabelle's built in first-order prover Metis. Figure 2 shows an overview of the tactic.

The user can configure the tactic to specify how to select facts to be passed to the simplifier and to Metis. The default is the simplification rules from the relevant function definitions, the datatype

```
TRY
  Structural induction schemes(s)
     THEN simplification + F.O. reasoning
OTHERWISE_TRY
  Recursion induction sceheme(s)
     THEN simplification + F.O. reasoning
```

Figure 2: Overview of Hipsters new tactic.

case distinction rules which are automatically derived by Isabelle, and the lemmas discovered by theory exploration so far. However, if we pass too many facts to Metis, it becomes slower. Therefore, the user can configure Hipster to include fewer of the discovered lemmas if needed. Hipster also impose a timeout on simplification and first-order reasoning, which can be set by the user. The default timeout is 1 second for each proof attempt. As further work, we plan to experiment with using Sledgehammer instead [PB10], which calls powerful external first-order provers and reports back exactly which facts were needed in the proof. Metis can then reconstruct the proof quickly inside Isabelle's trusted kernel.

Example 4: Simultaneous Induction. A notable gain of the new tactic with recursion induction is that of having the capability of performing simultaneous induction, whereas previously only structural inductions on a single variable were performed by Hipster. Simultaneous induction schemata are those inducting over more than one variable at a time, whether those variables are of the same type or not. Such is the case for the list function zip's recursion induction scheme, which corresponds to parallel induction on two lists:

```
fun zip :: "'a list ⇒ 'b list ⇒ ('a × 'b) list" where
  "zip [] y = []"
| "zip (x # xs) [] = []"
| "zip (x # xs) (y # ys) = (x, y) # (zip xs ys)"
```

ZipInd

$$\frac{\forall ys\ \ P\,([],\, ys) \qquad \forall x,\, xs\ \ P\,(x\,\#\,xs,\, []) \qquad \forall x,\, y,\, xs,\, ys\ \ (P\,(xs,\, ys) \Longrightarrow P\,(x\,\#\,xs,\, y\,\#\,ys))}{\forall xs,\, ys\ \ P\,(xs,\, ys)}$$

This scheme, along with some initial theory exploration, allows theorems like the following to be proven automatically:

```
zip (xs @ ys) zs = (zip xs (take (len xs) zs)) @ (zip ys (drop (len xs) zs))
```

Or even the alternative related conditional lemma to be proven without prior exploration:

```
len xs = len ys ⟹ (zip xs ys) @ (zip zs ws) = zip (xs @ zs) (ys @ ws)
```

Neither of these lemmas were provable before, even having done exploration for all the occurring functions in them. Hipster's prior structural induction approach could not capture in a scheme the relation between two variables. In these two cases, zip traverses its arguments taking steps on both at the same time, a pattern we can only capture with some form of simultaneous induction. Instead of synthesising a series of possible simultaneous structural induction schemata, recursion induction gives us an immediate choice which is also closer to the problem at hand.

3.3 Interactive Case Study: Insertion Sort

We here showcase Hipster's handling of conditional lemmas via the proof of correctness for the theorem `sorted (isort ts)`. For it, we assume the less-or-equal operator $\leq$ for naturals (and no prior, additional lemmas), and the function definitions:

```
fun sorted :: "Nat List ⇒ Bool" where
  "sorted [] = True"
| "sorted ([x]) = True"
| "sorted (x1 # (x2 # xs)) = (x1 ≤ x2 & sorted (x2 # xs))"

fun insert :: "Nat ⇒ Nat List ⇒ Nat List" where
  "insert x [] = [x]"
| "insert x1 (x2 # xs) = (if (x1 ≤ x2) then x1 # (x2 # xs)
                                         else x2 # (insert x1 xs))"

fun isort :: "Nat List ⇒ Nat List" where
  "isort [] = []"
| "isort (x # xs) = insert x (isort xs)"
```

Running exploration from the simpler components is the first step, considering both equational and conditional lemmas, since we have two predicates involved in the definiens of functions in the final theorem. The following command invokes conditional exploration for $\leq$:

```
hipster_cond ≤
```

which, along with conditional exploration for its negation, results in 10 discovered and proven lemmas, 6 of which are conditionals (we present the vital lemmas towards the final proof) and all require recursion induction:

```
lemma lemma_ac [thy_expl]: "x ≤ y ⟹ x ≤ (S y) = True"
by (hipster_induct_schemes ≤.simps Nat.exhaust)

lemma lemma_ad [thy_expl]: "y ≤ x ⟹ (S x) ≤ y = False"
by (hipster_induct_schemes ≤.simps Nat.exhaust)
(...)
lemma lemma_ai [thy_expl]: "(¬ (x ≤ y)) ⟹ x ≤ Z = False"
by (hipster_induct_schemes ≤.simps Nat.exhaust)
(...)
```

Hipster automatically generates this output. For each case, the `lemma` command makes the statement to be proven and is followed by a tactic application via the `by` command, here using Hipster's recursion induction tactic `hipster_induct_schemes`, which employs recursion induction where necessary. To enable the completion of the proof, exploration provides it with the automatically generated Isabelle rules for simplification of function definitions, such as ≤.simps, and datatype case distinction rules, such as Nat.exhaust.

With a new exploration considering the functions about sorting itself and (potentially) taking sorted as a side-condition for which to discover lemmas, Hipster discovers and proves both the conditional auxiliary lemma required and the goal theorem. Note that the exploration command takes as its first argument the predicate with which to construct side-conditions:

```
hispter_cond sorted isort insert
(...)
lemma isortInvariant [thy_expl]: "sorted ys ⟹ sorted (insert x ys) = True"
by (hipster_induct_schemes sorted.simps isort.simps insert.simps)
(...)
theorem isortSorts [thy_expl]: "sorted (isort x) = True"
by (hipster_induct_schemes sorted.simps isort.simps insert.simps)
```

During this last exploration, other interesting lemmas are discovered, all of which can be now proven automatically by using the sub-lemma about insert's invariant isortInvariant:

```
lemma isortFixes [thy_expl]: "sorted x ⟹ isort x = x"
by (hipster_induct_schemes sorted.simps isort.simps insert.simps)

lemma insertComm [thy_expl]: "insert x (insert y z) = insert y (insert x z)"
by (hipster_induct_schemes insert.simps)
```

Invoking the recursion induction tactic hipster_induct_schemes once proves all of the statements above, simplifying the interaction with the proof assistant. Particularly, the crucial lemma isortInvariant is proven applying sorted's associated recursion induction scheme, highlighting once again the need for support of conditional lemmas in automated inductive proving and the possibilities recursion induction brings towards proof automation.

4 Evaluation

In this section we present an evaluation of Hipster's automated tactics, an analysis which had not been performed for Hipster to the same extent priorly.

Keeping in mind evaluation of automated tools for interactive theorem proving necessarily has to consider some degree of interaction, two forms of evaluation have been carried out[1]:

- case studies on algebraic data types and operations on them; in particular focusing on inductive theories for natural numbers and lists
- evaluation on problems from TIP (Tons of Inductive Problems) [CJRS15], a set of benchmarks and challenge problems for inductive theorem provers.

From TIP, we evaluate Hipster over two sets of problems employed in previous works on inductive theorem proving: Johansson, Dixon and Bundy's work on case-analysis for rippling

[1] Source code for Hipster, examples presented and benchmarks are available online: https://github.com/moajohansson/IsaHipster

[JDB10] (we denote it *case-analysis* [2]), and prior work by Ireland and Bundy on employing proof failure to guide lemma discovery and patch inductive proofs [IB96] (we denote it *prod-failure* [3]). We now present these results and compare them with other tools' reported results.

4.1 Method

To evaluate performance on TIP, each problem is analysed individually, in isolation from others, to assess how far Hipster can go from bare definitions. Theory explorations were only run whenever the problem was not provable by the induction tactic directly, i.e. when the problem was missing helping lemmas. Explorations were first performed on the individual functions appearing in the problem definition, jointly with their auxiliary functions. These were followed by explorations on groups of said functions if required, leaving conditional exploration as the last exploration to be run before defining the problem as non-provable by Hipster.

As already specified, conditional lemma discovery is limited to explore a single predicate at a time to define side-conditions. For the present evaluation this has sufficed.

Additionally, to test Hipster's capacity when working on strictly newly defined theories, no assumptions nor properties from theories in Isabelle/HOL were considered during proof search. As an example, natural numbers are not Isabelle/HOL's, but redefined. Hence, predefined notions of orderings and other properties do not play a part in proofs obscuring the results of Hipster's actual work. In this way, we only consider as the base starting point a set of definitional statements, aligning with the purpose of proving based on structure and construction of programs.

4.2 Results

The following set of tables summarises statistics on the two sets of the benchmarks, with respect to the number of problems solved. Columns *EQ* and *COND* do so for problems defined by an equational and a conditional theorem respectively.

	Case-analysis		Prod-failure		**Total**
	EQ	COND	EQ	COND	
Total number of benchmarks	71	14	38	12	**135**
Number of problems solved	71	13	35	12	**131**

Table 1: Total number of problems solved.

Automation Table 2 shows the number of problems with automated solutions out of those which were solved. Full automation is understood as solving a problem only with discovered lemmas about the function symbols involved in the target theorem and Hipster's automated recursion induction. Partially automated problems are those for which additional related functions of a datatype's theory were provided to exploration for completion.

[2] Case-analysis problems: https://github.com/tip-org/benchmarks/tree/master/benchmarks/isaplanner

[3] Prod-failure problems: https://github.com/tip-org/benchmarks/tree/master/benchmarks/prod

	Case-analysis		Prod-failure		**Total**
	EQ	COND	EQ	COND	
Fully automated	67	13	29	12	**121**
Partially automated	4	0	6	0	**10**

Table 2: Automation of problems solved.

	Case-analysis		Prod-failure	
	EQ	COND	EQ	COND
No additional lemmas	38	10	1	8
Only equational lemmas	27	2	32	1
Equational and conditional lemmas	6	1	2	3

Table 3: Number of problems requiring discovery of auxiliary lemmas.

Overall, the rate of fully automated provability on the benchmark set is 90% ; considering partially automated problems as well, the overall rate is 97%.

A number of theorems (problems 52, 53, 72, 74 from *case-analysis*; and 2, 4, 5, 20, 22, 23 from *prod-failure*) required one of the following two similar lemmas:

```
len (x @ y) = len (y @ x)
count z (x @ y) = count z (y @ x)
```

These two lemmas are not automatically proven in a first instance (neither by structural nor recursion induction). Each of them in turn needs an auxiliary lemma which is not discovered.

Nonetheless, their proof can be partially automated. In both cases, one can observe that the outermost function applied, `len` and `count` respectively, acts as a relator function between two datatypes. Furthermore, these will in fact act as relator functions between list concatenation `@` and addition for natural numbers `plus`. Since `plus` does not occur in the problems to be proven, it is not added to the exploration directly. Adding `plus` interactively, Hipster discovers and proves automatically the lemmas:

```
len (x @ y) = plus (len x) (len y)
count z (x @ y) = plus (count z x) (count z y)
```

Along with the commutative law for `plus`, also discovered and proven automatically, they enable the automation of the two pending proofs without further intervention. And so, the corresponding TIP problems are solved as well.

These two cases seem to indicate that recursion induction may not suffice when a non-commutative operation nested within another has commuting arguments on both sides of an equality. At least not in the absence of smaller related lemmas corresponding to subgoals. This seems reasonable: the structure of the terms at each side of the equality will differ upon induction.

Theory exploration Just over half of the problems required prior lemma discovery, showcasing the benefit of theory exploration. In Table 3 we show the number of solved problems which required prior theory exploration and specify how many required further conditional lemmas.

A smaller subset of problems were provable with the aid of conditional exploration, namely those involving functions defined in terms of some predicate.

Recursion induction Whereas recursion induction was not necessary as often as theory exploration (whether for the main theorem or auxiliary lemmas), its impact is still notable. Some problems would not be provable employing only Hipster's prior structural induction approach. In Table 4, problems solved by structural induction are those for which both the main theorem and any required auxiliary lemma only needed structural induction. Those solved by recursion induction required it for the main theorem's proof or any of its helping lemmas.

	Case-analysis		Prod-failure	
	EQ	COND	EQ	COND
Structural induction	38	7	30	11
Recursion induction	33	6	5	1

Table 4: Number of problems solved with both kinds of induction.

Overall, there seems to be a trade-off between using weaker induction schemes (structural induction) and reducing the number and complexity of needed auxiliary lemmas. Structural induction was always attempted first by the tactic, meaning theorems solved via recursion induction (around a third of the benchmarks) would have not been solved otherwise, at least not with the degree of exploration carried out.

The results suggest recursion induction can save on exploration time. It provides appropriate induction patterns that avoid the need for sub-lemmas about specific constructor combinations.

4.3 Comparison

Other inductive provers have also been evaluated on these test suites, serving as a good point of comparison. The following table collects the number of problems solved by some of them in comparison with Hipster; note that we compare on problems for which other provers have available data. Plain figures correspond to fully automated solutions and those in parentheses (x) indicate number of successful proofs after some adaptation of settings. In total, *case-analysis* has 85 problems whilst *prod-failure* has 50.

	Hipster		HipSpec		Zeno	IsaPlanner	CVC4	Pirate	
Case-analysis	80	(84)	80		82	47	80	85	
Prod-failure	41	(47)	44	(47)	21	-	40		(47)

The already mentioned HipSpec uses theory exploration, structural induction and external first-order provers to prove properties about functional programs [CJRS13]. Zeno is a tool for proving equational inductive properties of Haskell programs [SDE12]. CVC4's approach to inductive proving is built on SMT solving whilst Pirate is built on first-order prover SPASS, both with a top-down approach in conjecture generation [RK15, WW]. IsaPlanner is a proof planning tool for Isabelle based on rippling [DJ07, JDB10].

In comparison to other (automated) inductive provers, the new Hipster is the only one (to the best of our knowledge) to employ recursion induction. As results show, its performance is on par to other state-of-the-art tools'. Additionally, unlike these tools, Hipster produces formal, certified proofs.

To be noted is that the failing problems for Hipster in the benchmark set *prod-failure* (problems 33-35) differ from those HipSpec and Pirate fail at (with the exception of 33 in Pirate's case). These three problems involve definitions for multiplication, factorial and exponentiation operations for Peano numerals with accumulator arguments. Particularly, HipSpec employed adjusted settings for lemma discovery in these three cases: the generators for random values of datatypes are manually defined. As already pointed out in §3.1, Hipster derives generators automatically, which means the simplicity of these could lead to inefficiencies when it comes to generating values of larger sizes. Hipster has not been evaluated with adjusted settings at the HipSpec/QuickSpec level and hence the exploration phase was not feasible to perform for these problems due to memory usage during testing in QuickSpec. With similar settings to HipSpec's, problems 33-35 are likely to be solvable in Hipster too.

5 Related Work

The work on lemma discovery for inductive proofs has mainly focused on equational lemmas, for instance in the theory exploration systems IsaScheme and IsaCoSy [MMDB12, JDB11], which also work on Isabelle/HOL theories. IsaScheme requires the user to provide *term schemas*, which are then automatically filled in with available symbols. IsaCoSy only generates irreducible terms, and uses an internal constraint language to avoid generating anything that could be reduced by a known equation. These systems focused more on automation, while Hipster is designed to be useable in an interactive theory development. Hipster is faster, and now also supports conditional theory exploration where the user specifies an interesting condition. Conditional lemma discovery has also been missing from the IsaPlanner system, which uses *proof critics* to deduce lemmas from failed proof attempts [DJ07, JDB10].

Theory exploration systems rely on having an automated prover at hand to prove generated conjectures. In the context of inductive theories, most other automated provers supporting induction such as IsaPlanner, Zeno, HipSpec, Dafny and CVC4 [DJ07, SDE12, CJRS13, Lei12, RK15] only support structural induction. Hipster now also provides an automated tactic for recursion induction by exploiting Isabelle's automated derivation of such induction schemata. It can both be used in theory exploration and as a stand-alone automated tactic.

The use of recursion induction and the fact that Hipster produces LCF-style re-checkable proofs is also the main difference between Hipster and its sister system HipSpec [CJRS13], with which Hipster shares its conjecture generation component. HipSpec does instead rely on external first-order provers to solve the proof obligations arising in the step- and base-cases for inductive proofs, and does not produce checkable proofs.

6 Conclusion and Further Work

Generation of conditional lemmas in theory exploration is a challenging problem, not least as it is difficult for a tool to automatically assess which side conditions are interesting. Hipster is an interactive theory exploration system, and gets around this obstacle by relying on the user to decide which predicates are deemed interesting as conditions. In this paper we have also presented a new automated tactic for recursion induction, which improves the level of proof automation of discovered conjectures in Hipster. It can also be used as a powerful stand-alone induction tactic in Isabelle. Further work on the proving side includes experimenting with different heuristics for choosing which function's recursion induction scheme is most likely to produce a proof, as well as extending Hipster with tactics that can handle mutual- and co-induction automatically.

Hipster has various configuration options for adjusting which of the discovered lemmas are passed to its tactics in subsequent proofs. For example, in larger theories, with many explorations, we may not want to pass all discovered lemmas to Isabelle's Metis tactic, as too many lemmas might slow down the proof process. We plan to experiment with combining Hipster's tactics with the relevance filtering ideas used in Sledgehammer [KBKU13]. Another item of further work is to extend Hipster to produce structured proofs in Isabelle's Isar language, instead of just a one-line application of Hipster's custom tactics. This will be easier to read for a human user, and can be more streamlined, not needing to repeat the search done in the automatic proof found by Hipster's powerful tactics.

Bibliography

[CH00] K. Claessen, J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of ICFP*. Pp. 268–279. 2000.

[CJRS13] K. Claessen, M. Johansson, D. Rosén, N. Smallbone. Automating Inductive Proofs using Theory Exploration. In *Proceedings of the Conference on Automated Deduction (CADE)*. LNCS 7898, pp. 392–406. Springer, 2013.

[CJRS15] K. Claessen, M. Johansson, D. Rosén, N. Smallbone. TIP: Tons of Inductive Problems. In *Proceedings of the Conference on Intelligent Computer Mathematics (CICM)*. LNCS 9150. Springer, 2015.

[CSH10] K. Claessen, N. Smallbone, J. Hughes. QuickSpec: Guessing Formal Specifications Using Testing. In *Proceedings of TAP*. Pp. 6–21. 2010.

[DJ07] L. Dixon, M. Johansson. IsaPlanner 2: A Proof Planner in Isabelle. 2007. DReaM Technical Report (System description).
http://dream.inf.ed.ac.uk/projects/isaplanner/docs/isaplanner-v2-07.pdf

[HN10] F. Haftmann, T. Nipkow. Code Generation via Higher-Order Rewrite Systems. In Blume et al. (eds.), *Functional and Logic Programming*. LNCS 6009, pp. 103–117. Springer, 2010.

[IB96] A. Ireland, A. Bundy. Productive Use of Failure in Inductive Proof. *Journal of Automated Reasoning* 16:79–111, 1996.

[JDB10] M. Johansson, L. Dixon, A. Bundy. Case-Analysis for rippling and inductive proof. In *Proceedings of ITP*. Pp. 291–306. 2010.

[JDB11] M. Johansson, L. Dixon, A. Bundy. Conjecture Synthesis for Inductive Theories. *Journal of Automated Reasoning* 47(3):251–289, 2011.

[JRSC14] M. Johansson, D. Rosén, N. Smallbone, K. Claessen. Hipster: Integrating Theory Exploration in a Proof Assistant. In *Proceedings of the Conference on Intelligent Computer Mathematics (CICM)*. LNCS 8543, pp. 108–122. Springer, 2014.

[KBKU13] D. Kuhlwein, J. C. Blanchette, C. Kaliszyk, J. Urban. MaSh: Machine Learning for Sledgehammer. In *Interactive Theorem Proving*. LNCS 7998, pp. 35–50. Springer, 2013.

[KST$^+$11] A. Krauss, C. Sternagel, R. Thiemann, C. Fuhs, J. Giesl. Termination of Isabelle Functions via Termination of Rewriting. In Eekelen et al. (eds.), *Interactive Theorem Proving*. Lecture Notes in Computer Science 6898, pp. 152–167. Springer Berlin Heidelberg, 2011.

[Lei12] K. R. Leino. Automating Induction with an SMT Solver. In *Proceedings of VMCAI*. Springer, 2012.

[MBA07] R. McCasland, A. Bundy, S. Autexier. Automated discovery of inductive theorems. In Matuszewski and Rudnicki (eds.), *From Insight to Proof: Festschrift in Honor of A. Trybulec*. 2007.

[MMDB12] O. Montano-Rivas, R. McCasland, L. Dixon, A. Bundy. Scheme-based theorem discovery and concept invention. *Expert Systems with Applications* 39(2):1637–1646, 2012.

[NPW02] T. Nipkow, L. C. Paulson, M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002. Latest online version 25 May 2015. http://isabelle.in.tum.de/doc/tutorial.pdf

[PB10] L. C. Paulson, J. C. Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. *IWIL-2010*, 2010.

[RK15] A. Reynolds, V. Kuncak. Induction for SMT Solvers. In *Proceedings of VMCAI*. 2015.

[SDE12] W. Sonnex, S. Drossopoulou, S. Eisenbach. Zeno: An automated prover for properties of recursive datatypes. In *Proceedings of TACAS*. Pp. 407–421. Springer, 2012.

[WW] D. Wand, C. Weidenbach. Automatic Induction inside Superposition. https://people.mpi-inf.mpg.de/~dwand/datasup/draft.pdf.