Hipster: Integrating Theory Exploration in a Proof Assistant

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

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

Abstract. This paper describes Hipster, a system integrating theory exploration with the proof assistant Isabelle/HOL. Theory exploration is a technique for automatically discovering new interesting lemmas in a given theory development. Hipster can be used in two main modes. The first is *exploratory mode*, used for automatically generating basic lemmas about a given set of datatypes and functions in a new theory development. The second is *proof mode*, used in a particular proof attempt, trying to discover the missing lemmas which would allow the current goal to be proved. Hipster's proof mode complements and boosts existing proof automation techniques that rely on automatically selecting existing lemmas, by inventing new lemmas that need induction to be proved. We show example uses of both modes.

1 Introduction

The concept of theory exploration was first introduced by Buchberger [2]. He argues that in contrast to automated theorem provers that focus on proving one theorem at a time in isolation, mathematicians instead typically proceed by exploring entire theories, by conjecturing and proving layers of increasingly complex propositions. For each layer, appropriate proof methods are identified, and previously proved lemmas may be used to prove later conjectures. When a new concept (e.g. a new function) is introduced, we should prove a set of new conjectures which, ideally, "completely" relates the new with the old, after which other propositions in this layer can be proved easily by "routine" reasoning. Mathematical software should be designed to support this workflow. This is arguably the mode of use supported by many interactive proof assistants, such as Theorema [3] and Isabelle [17]. However, they leave the generation of new conjectures relating different concepts largely to the user. Recently, a number of different systems have been implemented to address the conjecture synthesis aspect of theory exploration [15,13,16,5]. Our work goes one step further by integrating the discovery and proof of new conjectures in the workflow of the interactive theorem prover Isabelle/HOL. Our system, called Hipster, is based on our previous work on HipSpec [5], a theory exploration system for Haskell programs. In that work, we showed that HipSpec is able to automatically discover many of the kind of equational theorems present in, for example, Isabelle/HOL's libraries for natural numbers and lists. In this article we show how similar techniques can be used to speed up and facilitate the development of new theories in Isabelle/HOL by discovering basic lemmas automatically.

Hipster translates Isabelle/HOL theories into Haskell and generates equational conjectures by testing and evaluating the Haskell program. These conjectures are then imported back into Isabelle and proved automatically. Hipster can be used in two ways: in *exploratory mode* it quickly discovers basic properties about a newly defined function and its relationship to already existing ones. Hipster can also be used in *proof mode*, to provide lemma hints for an ongoing proof attempt when the user is stuck.

Our work complements Sledgehammer [18], a popular Isabelle tool allowing the user to call various external automated provers. Sledgehammer uses *relevance filtering* to select among the available lemmas those likely to be useful for proving a given conjecture [14]. However, if a crucial lemma is missing, the proof attempt will fail. If theory exploration is employed, we can increase the success rate of Isabelle/HOL's automatic tactics with little user effort.

As an introductory example, we consider the example from section 2.3 of the Isabelle tutorial [17]: proving that reversing a list twice produces the same list. We first apply structural induction on the list `xs`.

```
theorem rev_rev : "rev(rev xs) = xs"
apply (induct xs)
```

The base case follows trivially from the definition of `rev`, but Isabelle/HOL's automated tactics `simp`, `auto` and `sledgehammer` all fail to prove the step case. We can simplify the step case to:

$$\texttt{rev(rev xs) = xs} \Longrightarrow \texttt{rev((rev xs) @ [x]) = x\#xs}$$

At this point, we are stuck. This is where Hipster comes into the picture. If we call Hipster at this point in the proof, asking for lemmas about `rev` and append (`@`), it suggests and proves three lemmas:

```
lemma lemma_a:  "xs @ [] = xs"
lemma lemma_aa : "(xs @ ys) @ zs = xs @ (ys @ zs)"
lemma lemma_ab : "(rev xs) @ (rev ys) = rev (ys @ xs)"
```

To complete the proof of the stuck subgoal, we need lemma `ab`. Lemma `ab` in turn, needs lemma `a` for its base case, and lemma `aa` for its step case. With these three lemmas present, Isabelle/HOL's tactics can take care of the rest. For example, when we call Sledgehammer in the step case, it suggests a proof by Isabelle/HOL's first-order reasoning tactic `metis` [11], using the relevant function definitions as well as `lemma_ab`:

```
theorem rev_rev : "rev(rev xs) = xs"
apply (induct xs)
apply simp
sledgehammer
by (metis rev.simps(1) rev.simps(2) app.simps(1) app.simps(2) lemma_ab)
```

The above example shows how Hipster can be used interactively in a stuck proof attempt. In exploratory mode, there are also advantages of working in an interactive setting. For instance, when dealing with large theories that would otherwise generate a very large search space, the user can instead incrementally explore different relevant sub-theories while avoiding a search space explosion. Lemmas discovered in each sub-theory can be made available when exploring increasingly larger sets of functions.

The article is organised as follows: In section 2 we give a brief overview of the HipSpec system which Hipster uses to generate conjectures, after which we describe Hipster in more detail in section 3, together with some larger worked examples of how it can be used, both in proof mode and exploratory mode. In section 4 we describe how we deal with partial functions, as Haskell and Isabelle/HOL differ in their semantics for these. Section 5 covers related work and we discuss future work in section 6.

2 Background

In this section we give a brief overview of the HipSpec system which we use as a backend for generating conjectures, and of Isabelle's code generator which we use to translate Isabelle theories to Haskell programs.

2.1 HipSpec

HipSpec is a state-of-the-art inductive theorem prover and theory exploration system for Haskell. In [5] we showed that HipSpec is able to automatically discover and prove the kind of equational lemmas present in Isabelle/HOL's libraries, when given the corresponding functions written in Haskell.

HipSpec works in two stages:

1. Generate a set of conjectures about the functions at hand. These conjectures are equations between terms involving the given functions, and have not yet been proved correct but are nevertheless extensively tested.
2. Attempt to prove each of the conjectures, using already proven conjectures as assumptions. HipSpec implements this by enumerating induction schemas, and firing off many proof obligations to automated first-order logic theorem provers.

The proving power of HipSpec comes from its capability to automatically discover and prove lemmas, which are then used to help subsequent proofs.

In Hipster we can not directly use HipSpec's proof capabilities (stage (2) above); we use Isabelle/HOL for the proofs instead. Isabelle is an LCF-style prover which means that it is based on a small core of trusted axioms, and proofs must be built on top of those axioms. In other words, we would have to reconstruct inside Isabelle/HOL any proof that HipSpec found, so it is easier to use Isabelle/HOL for the proofs in the first place.

The part of HipSpec we directly use is its conjecture synthesis system (stage (1) above), called QuickSpec [6]), which efficiently generates equations about a given set of functions and datatypes.

QuickSpec takes a set of functions as input, and proceeds to generate all type-correct terms up to a given limit (usually up to depth three). The terms may contain variables (usually at most three per type). These parameters are set heuristically, and can be modified by the user. QuickSpec attempts to divide the terms into equivalence classes such that two terms end up in the same equivalence class if they are equal. It first assumes that all terms of the same type are equivalent, and initially puts them in the same equivalence class. It then picks random ground values for the variables in the terms (using QuickCheck [4]) and evaluates the terms. If two terms in the same equivalence class evaluate to different ground values, they cannot be equal; QuickSpec thus breaks each equivalence class into new, smaller equivalence classes depending on what values their terms evaluated to. This process is repeated until the equivalence classes stabilise. We then read off equations from each equivalence class, by picking one term of that class as a representative and equating all the other terms to that representative. This means that the conjectures generated are, although not yet proved, fairly likely to be true, as they have been tested on several hundred different random values. The confidence increases with the number of tests, which can be set by the user. The default setting is to first run 200 tests, after which the process stops if the equivalence classes appear to have stabilised, i.e. if nothing has changed during the last 100 tests. Otherwise, the number of tests are doubled until stable.

As an example, we ask QuickSpec to explore the theory with list append, `@`, the empty list, `[]`, and three list variables `xs`, `ys`, `zs`. Among the terms it will generate are `(xs @ ys) @ zs`, `xs @ (ys @ zs)`, `xs @ []` and `xs`. Initially, all four will be assumed to be in the same equivalence class. The random value generator for lists from QuickCheck might for instance generate the values: `xs` $\mapsto$ `[]`, `ys` $\mapsto$ `[a]` and `zs` $\mapsto$ `[b]`, where `a` and `b` are arbitrary distinct constants. Performing the substitutions of the variables in the four terms above and evaluating the resulting ground expressions gives us:

	Term	Ground Instance	Value
1	`(xs @ ys) @ zs`	`([] @ [a]) @ [b]`	`[a,b]`
2	`xs @ (ys @ zs)`	`[] @ ([a] @ [b])`	`[a,b]`
3	`xs @ []`	`[] @ []`	`[]`
4	`xs`	`[]`	`[]`

Terms 1 and 2 evaluate to the same value, as do terms 3 and 4. The initial equivalence class will therefore be split in two accordingly. After this, whatever variable assignments QuickSpec generates, the terms in each class will evaluate to the same value. Eventually, QuickSpec stops and the equations for associativity and right identity can be extracted from the resulting equivalence classes.

2.2 Code Generation in Isabelle

Isabelle/HOL's code generator can translate from Isabelle's higher-order logic to code in several functional programming languages, including Haskell [9,8]. Isabelle's higher-order logic is a typed λ-calculus with polymorphism and type-classes. Entities like constants, types and recursive functions are mapped to corresponding entities in the target language. For the kind of theories we consider in this paper, this process is straightforward. However, the code generator also supports user-given *code lemmas*, which allows it to generate code from non-executable constructs, e.g. by replacing sets with lists.

3 Hipster: Implementation and Use

We now give a description of the implementation of Hipster, and show how it can be used both in theory exploration mode and in proof mode, to find lemmas relevant for a particular proof attempt. An overview of Hipster is shown in figure 1. The source code and examples are available online[1].

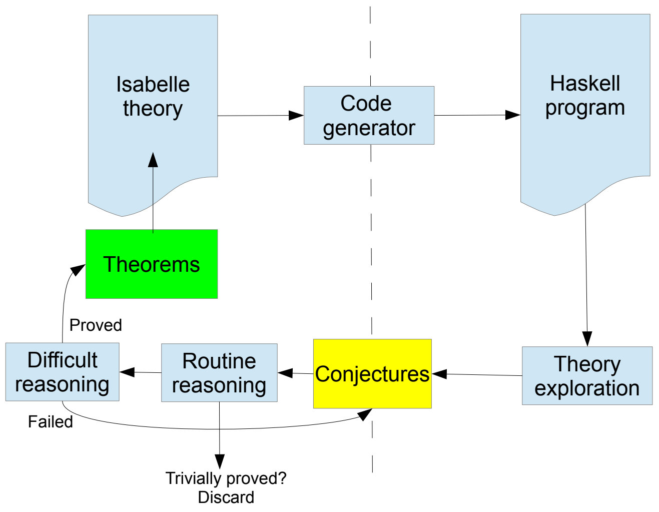

Fig. 1. Overview of Hipster

Starting from an Isabelle/HOL theory, Hipster calls Isabelle/HOL's code generator [9,8] to translate the given functions into a Haskell program. In order to use the testing framework from QuickCheck, as described in the previous section, we also post-process the Haskell file, adding *generators*, which are responsible for producing arbitrary values for evaluation and testing. A generator in Haskell's

[1] `https://github.com/moajohansson/IsaHipster`

QuickCheck is simply a function which returns a random ground value for a particular datatype [4]. In our case, the generators pick a random constructor and then recursively generate its arguments. To ensure termination, the generators are parametrised by a size; the generator reduces the size when it invokes itself recursively and when the size reaches zero, it always picks a non-recursive constructor.

Another important issue in the translation is the difference in semantics for partial functions between Isabelle/HOL and Haskell. In order for HipSpec not to miss equations that hold in Isabelle/HOL, but not in Haskell, we have to translate partial functions specially. This is explained in more detail in section 4.

Once the Haskell program is in place, we run theory exploration and generate a set of equational conjectures, which HipSpec orders according to generality. More general equations are preferred, as we expect these to be more widely applicable as lemmas. In previous work on HipSpec, the system would at this stage apply induction on the conjectures and send them off to some external prover. Here, we instead import them back into Isabelle as we wish to produce checkable LCF-style proofs for our conjectures.

The proof procedure in Hipster is parametrised by two tactics, one for easy or *routine reasoning* and one for *difficult reasoning*. In the examples below, the routine reasoning tactic uses Isabelle/HOL's simplifier followed by first-order reasoning by Metis [11], while the difficult reasoning tactic performs structural induction followed by simplification and first-order reasoning. Metis is restricted to run for at most one second in both the routine and difficult tactic. If there are several possible variables to apply induction on, we may backtrack if the first choice fails. Both tactics have access to the theorems proved so far, and hence get stronger as the proof procedure proceeds through the list of conjectures.

As theory exploration produces rather many conjectures, we do not want to present them all to the user. We select the most interesting ones, i.e. those that are difficult to prove, and filter out those that follow only by routine reasoning. Depending on the theory and application we can vary the tactics for routine and difficult reasoning to suit our needs. If we want Hipster to produce fewer or more lemmas, we can choose a stronger or weaker tactic, allowing for flexibility.

The order in which Hipster tries to prove things matters. As we mentioned, it will try more general conjectures first, with the hope that they will be useful to filter out many more specific routine results. Occasionally though, a proof will fail as a not-yet-proved lemma is required. In this case, the failed conjecture is added back into the list of open conjectures and will be retried later, after at least one new lemma has been proved. Hipster terminates when it either runs out of open conjectures, or when it can not make any more progress, i.e. when all open conjectures have been tried since it last proved a lemma.

Below we give two typical use cases for Hipster. In both examples, Hipster has been instantiated with the routine and difficult reasoning tactics that we described above.

3.1 Exploring a Theory of Binary Trees

This example is about a theory of binary trees, with data stored at the leaves:

```
datatype 'a Tree =
    Leaf 'a
  | Node "'a Tree" "'a Tree"
```

Let us define some functions over our trees: `mirror` swaps the left and right subtrees everywhere, and `tmap` applies a function to each element in the tree.

```
fun mirror :: 'a Tree => 'a Tree
where
  mirror (Leaf x) = Leaf x
| mirror (Node l r) = Node (mirror r) (mirror l)

fun tmap :: ('a => 'b) => 'a Tree => 'b Tree
where
  tmap f (Leaf x) = Leaf (f x)
| tmap f (Node l r) = Node (tmap f l) (tmap f r)
```

Now, let us call theory exploration to discover some properties about these two functions. Hipster quickly finds and proves the two expected lemmas:

```
lemma lemma_a [thy_expl]: "mirror (tmap x y) = tmap x (mirror y)"
by (tactic {* Hipster_Tacs.induct_simp_metis . . .*})

lemma lemma_aa [thy_expl]: "mirror (mirror x) = x"
by (tactic {* Hipster_Tacs.induct_simp_metis . . . *})
```

The output produced by Hipster can be automatically pasted into the proof script by a mouseclick. Recall that Hipster discards all lemmas that can be proved by routine reasoning (here, without induction). The tactic `induct_simp_metis` appearing in the proof script output is the current instantiation of "difficult reasoning". Note that the lemmas discovered are tagged with the attribute `thy_expl`, which tells Hipster which lemmas it has discovered so far. If theory exploration is called several times, we can use these lemmas in proofs and avoid rediscovering the same things. The user can also inspect what theory exploration has found so far by executing the Isabelle command `thm thy_expl`.

Next, let us also define two functions extracting the leftmost and rightmost element of the tree:

```
fun rightmost :: 'a Tree => 'a
where
  rightmost (Leaf x) = x
| rightmost (Node l r) = rightmost r

fun leftmost :: 'a Tree => 'a
where
  leftmost (Leaf x) = x
| leftmost (Node l r) = leftmost l
```

Asking Hipster for lemmas about all the functions defined so far, it provides one additional lemma, namely:

```
lemma lemma_ab [thy_expl]: "leftmost (mirror x2) = rightmost x2"
by (tactic {* Hipster_Tacs.induct_simp_metis . . . *})
```

Finally, we define a function flattening trees to lists:

```
fun flat_tree :: 'a Tree => 'a list
where
  flat_tree (Leaf x) = [x]
| flat_tree (Node l r) = (flat_tree l) @ (flat_tree r)
```

We can now ask Hipster to explore the relationships between the functions on trees and the corresponding functions on lists, such as `rev`, `map` and `hd`. Hipster produces four new lemmas and one open conjecture:

```
lemma lemma_ac [thy_expl]: "flat_tree (tmap x y) = map x (flat_tree y)"
by (tactic {* Hipster_Tacs.induct_simp_metis . . . *})

lemma lemma_ad [thy_expl]: "map x (rev xs) = rev (map x xs)"
by (tactic {* Hipster_Tacs.induct_simp_metis . . . *})

lemma lemma_ae [thy_expl]: "flat_tree (mirror x) = rev (flat_tree x)"
by (tactic {* Hipster_Tacs.induct_simp_metis . . . *})

lemma lemma_af [thy_expl]: "hd (xs @ xs) = hd xs"
by (tactic {* Hipster_Tacs.induct_simp_metis . . . *})

lemma unknown: "hd (flat_tree x) = leftmost x"
oops
```

Lemmas `ad` and `af` are perhaps not of much interest, as they only relate functions on lists. In fact, lemma `ad` is already in Isabelle/HOL's list library, but is not labelled as a simplification rule, which is why Hipster rediscovers it. Lemma `af` is a variant of a conditional library-lemma: `xs` $\neq$ `[]` $\Longrightarrow$ `hd(xs @ ys) = hd xs`. Observe that lemma `af` holds due to the partiality of `hd`. Hipster can not discover conditional lemmas, so we get this version instead.

In addition to the four lemmas which have been proved, Hipster also outputs one interesting conjecture (labelled `unknown`) which it fails to prove. To prove this conjecture, we need a lemma stating that, as the trees store data at the leaves, `flat_tree` will always produce a non-empty list: `flat_tree t` $\neq$ `[]`. As this is not an equation, it is not discovered by Hipster.

This example shows that Hipster can indeed find most of the basic lemmas we would expect in a new theory. The user has to provide the constants Hipster should explore, and the rest is fully automatic, thus speeding up theory development. Theory exploration in this example takes just a few seconds, no longer than it takes to run tools like Sledgehammer. Even if Hipster fails to prove some properties, they may still be interesting, and the user can choose to prove them interactively.

Exploring with different tactics. To illustrate the effects of choosing a slightly different tactic for routine and difficult reasoning, we also experimented with an instantiation using only Isabelle/HOL's simplifier as routine reasoning and induction followed by simplification as difficult reasoning. The advantage of this instantiation is that the simplifier generally is faster than Metis, but less powerful. However, for this theory, it turns out that the simplifier is sufficient to prove the same lemmas as above. Hipster also suggests one extra lemma, namely `rightmost (mirror x) = leftmost x`, which is the dual to lemma `ab` above. When we used Metis, this lemma could be proved without induction, by routine reasoning, and was thus discarded. Using only the simplifier, difficult reasoning and induction is required to find a proof, and the lemma is therefore presented to the user.

3.2 Proving Correctness of a Small Compiler

The following example is about a compiler to a stack machine for a toy expression language[2]. We show how theory exploration can be used to unblock a proof on which automated tactics otherwise fail due to a missing lemma.

Expressions in the language are built from constants (`Cex`), values (`Vex`) and binary operators (`Bex`):

```
type_synonym 'a binop = 'a => 'a => 'a

datatype ('a, 'b) expr =
    Cex 'a
  | Vex 'b
  | Bex "'a binop" "('a,'b) expr" "('a,'b) expr"
```

The types of values and variables are not fixed, but given by type parameters `'a` and `'b`. To evaluate an expression, we define a function `value`, parametrised by an environment mapping variables to values:

```
fun value :: ('b => 'a) => ('a,'b) expr => 'a
where
    value env (Cex c) = c
  | value env (Vex v) = env v
  | value env (Bex f e1 e2) = f (value env e1) (value env e2)
```

A program for our stack machine consists of four instructions:

```
datatype ('a, 'b) program =
    Done
  | Const 'a "('a, 'b) program"
  | Load 'b "('a, 'b) program"
  | Apply "'a binop" "('a, 'b) program"
```

A program is either empty (`Done`), or consists of one of the instructions `Const`, `Load` or `Apply`, followed by the remaining program. We further define a function `sequence` for combining programs:

[2] This example is a slight variation of that in §3.3 in the Isabelle tutorial [17].

```
fun sequence :: ('a, 'b) program => ('a, 'b) program => ('a, 'b) program
where
    sequence Done p = p
  | sequence (Const c p) p' = Const c (sequence p p')
  | sequence (Load v p) p' = Load v (sequence p p')
  | sequence (Apply f p) p' = Apply f (sequence p p')
```

Program execution is modelled by the function `exec`, which given a store for variables and a program, returns the values on the stack after execution.

```
fun exec :: ('b => 'a) => ('a,'b) program => 'a list => 'a list
where
    exec env Done stack = stack
  | exec env (Const c p) stack = exec env p (c#stack)
  | exec env (Load v p) stack = exec env p ((env v)#stack)
  | exec env (Apply f p) stack =
     exec env p ((f (hd stack) (hd(tl stack)))#(tl(tl stack)))
```

We finally define a function `compile`, which specifies how expressions are compiled into programs:

```
fun compile :: ('a,'b) expr => ('a,'b) program
  where
    compile (Cex c) =  Const c Done
  | compile (Vex v) =  Load v Done
  | compile (Bex f e1 e2) =
     sequence (compile e2) (sequence (compile e1) (Apply f Done))"
```

Now, we wish to prove correctness of the compiler, namely that executing a compiled expression indeed results in the value of that expression:

```
theorem "exec env (compile e) [] = [value env e]"
```

If we try to apply induction on `e`, Isabelle/HOL's simplifier solves the base-case but neither the simplifier or first-order reasoning by Sledgehammer succeeds in proving the step-case. At this stage, we can apply Hipster's theory exploration tactic. It will generate a set of conjectures, and interleave proving these with trying to prove the open sub-goal. Once Hipster succeeds in finding a set of lemmas which allow the open goal to be proved by routine reasoning, it presents the user with a list of lemmas it has proved, in this case:

```
Try first proving lemmas:

lemma lemma_a: "sequence x Done = x"
by (tactic {* Hipster_Tacs.induct_simp_metis . . . *})

lemma lemma_aa: "exec x y (sequence z x1) xs = exec x y x1 (exec x y z xs)"
by (tactic {* Hipster_Tacs.induct_simp_metis . . . *})

lemma lemma_ab: "exec x y (compile z) xs = value x y z # xs"
by (tactic {* Hipster_Tacs.induct_simp_metis . . . *})
```

Our theorem is a trivial instance of `lemma_ab`, whose proof depends on `lemma_aa`. Hipster takes about twenty seconds to discover and prove the lemmas. Pasting them into our proof script we can try Sledgehammer on our theorem again. This time, it succeeds and suggests the one-line proof:

```
theorem "exec env (compile e) [] = [value env e]"
by (metis lemma_ab)
```

4 Dealing With Partial Functions

Isabelle is a logic of total functions. Nonetheless, we can define apparently partial functions, such as `hd`:

```
fun hd :: 'a list => 'a where
  hd (x#xs) = x
```

How do we reconcile `hd` being partial with Isabelle functions being total? The answer is that in Isabelle, `hd` is total, but the behaviour of `hd []` is unspecified: it returns some arbitrary value of type `'a`. Meanwhile in Haskell, `head` is partial, but the behaviour of `head []` is specified: it crashes. We must therefore translate *partially defined* Isabelle functions into *total but underspecified* Haskell functions.

Hipster uses a technique suggested by Jasmin Blanchette [1] to deal with partial functions. Whenever we translate an Isabelle function that is missing some cases, we need to add a default case, like so:

```
hd :: [a] -> a
hd (x:xs) = x
hd [] = ???
```

But what should we put for the result of `hd []`? To model the notion that `hd []` is unspecified, whenever we evaluate a test case we will pick a *random* value for `hd []`. This value will vary from test case to test case but will be consistent within one run of a test case. The idea is that, if an equation involving `hd` in Haskell always holds, for all values we could pick for `hd []`, it will also hold in Isabelle, where the value of `hd []` is unspecified.

Suppose we define the function `second`, which returns the second element of a list, as

```
second (x#y#xs) = y
```

It might seem that we should translate `second`, by analogy with `hd`, as

```
second :: [a] -> a
second (x:y:xs) = y
second _ = ???
```

and pick a random value of type `a` to use in the default case. But this translation is wrong! If we apply our translated `second` to a single-element list, it will give the same answer regardless of which element is in the list, and HipSpec will discover the lemma `second [x] = second [y]`. This lemma is certainly not true of our Isabelle function, which says nothing about the behaviour of `second` on single-element lists, and Hipster will fail to prove it.

We must allow the default case to produce a different result for different arguments. We therefore translate `second` as

```
second :: [a] -> a
second (x:y:xs) = y
second xs = ??? xs
```

where `???` is a random *function* of type `[a] -> a`. (QuickCheck can generate random functions.) As before, whenever we evaluate a test case, we instantiate `???` with a new random function[3]. This second translation mimics Isabelle's semantics: any equation that holds in Haskell no matter how we instantiate the `???` functions also holds in Isabelle.

In Hipster, we first use Isabelle/HOL's code generator to translate the theory to Haskell. Then we transform *every* function definition, whether it is partial or not, in the same way we transformed `second` above. If a function is already total, the added case will simply be unreachable. This avoids having to check functions for partiality. The extra clutter introduced for total functions is not a problem as we neither reason about nor show the user the generated program.

5 Related Work

Hipster is an extension to our previous work on the HipSpec system [5], which was not designed for use in an interactive setting. HipSpec applies structural induction to conjectures generated by QuickSpec and sends off proof obligations to external first-order provers. Hipster short-circuits the proof part and directly imports the conjectures back into Isabelle. This allows for more flexibility in the choice of tactics employed, by letting the user control what is to be considered routine and difficult reasoning. Being inside Isabelle/HOL provides the possibility to easily record lemmas for future use, perhaps in other theory developments. It gives us the possibility to re-check proofs if required, as well as increased reliability as proofs have been run through Isabelle's trusted kernel. As Hipster uses HipSpec for conjecture generation, any difference in performance (e.g. speed, lemmas proved) will depend only on what prover backend is used by HipSpec and what tactic is used by Hipster.

There are two other theory exploration systems available for Isabelle/HOL, IsaCoSy [13] and IsaScheme [16]. They differ in the way they generate conjectures, and both discover similar lemmas as HipSpec/Hipster. A comparison between HipSpec, IsaCoSy and IsaScheme can be found in [5], showing that all three

[3] To avoid having to retranslate the Isabelle theory every time we evaluate a test case, in reality we parametrise the generated program on the various `???` functions. That way, whenever we evaluate a test case, we can cheaply change the default cases.

systems manage to find largely the same lemmas on theories about lists and natural numbers. HipSpec does however outperform the other two systems on speed. IsaCoSy ensures that terms generated are non-trivial to prove by only generating irreducible terms, i.e. conjectures that do not have simple proofs by equational reasoning. These are then filtered through a counter-example checker and passed to IsaPlanner for proof [7]. IsaScheme, as the name suggests, follows the scheme-based approach first introduced for algorithm synthesis in Theorema [3]. IsaScheme uses general user-specified schemas describing the shape of conjectures and then instantiates them with available functions and constants. It combines this with counter-example checking and Knuth-Bendix completion techniques in an attempt to produce a minimal set of lemmas.

Unfortunately, the counter-example checking in IsaCoSy and IsaScheme is often too slow for use in an interactive setting. Unlike IsaCoSy, Hipster may generate reducible terms, but thanks to the equivalence class reasoning in QuickSpec, testing is much more efficient, and conjectures with trivial proofs are instead quickly filtered out at the proof stage. None of our examples takes more than twenty seconds to run.

Neither IsaCoSy or IsaScheme has been used to generate lemmas in stuck proof attempts, but only in fully automated exploratory mode. Starting from stuck proof attempts allows us to reduce the size of the interesting background theory, which promises better scalability.

Proof planning critics have been employed to analyse failed proof attempts in automatic inductive proofs [12]. The critics use information from the failure in order to try to speculate a missing lemma top-down, using techniques based on rippling and higher-order unification. Hipster (and HipSpec) takes a less restricted approach, instead constructing lemmas bottom-up, from the symbols available. As was shown in our previous work [5], this succeeds in finding lemmas that the top-down critics based approach fails to find, at the cost of possibly also finding a few extra lemmas as we saw in the example in section 3.2.

6 Further Work

The discovery process is currently limited to equational lemmas. We plan to extend the theory exploration to also take side conditions into account. If theory exploration is called in the middle of a proof attempt, there may be assumptions associated with the current sub-goal, which could be a useful source of side conditions. For example, if we are proving a lemma about sorting, there will most likely be an assumption involving the “less than” operator; this suggests that we should look for equations that have “less than” in a side condition. Once we have a candidate set of side conditions, it is easy to extend QuickSpec to find equations that assume those conditions.

The parameters for Hipster, e.g. the number of QuickSpec generated tests, the runtime for Metis and so on, are largely based on heuristics from development and previous experience. There is probably room for fine-tuning these heuristics and possibly adapt them to the theory we are working in. We plan to add and experiment with additional automated tactics in Hipster. Again, we expect that different tactics will suit different theories.

Another interesting area of further work in theory exploration is reasoning by analogy. In the example in section 3.1, theory exploration discovers lemmas about `mirror` and `tmap` which are analogous to lemmas about lists and the functions `rev` and `map`. Machine learning techniques can be used to identify similar lemmas [10], and this information could then be used to for instance suggest new combinations of functions to explore, new connections between theories or directly suggest additional lemmas about trees by analogy to those on lists.

7 Summary

Hipster integrates lemma discovery by theory exploration in the proof assistant Isabelle/HOL. We demonstrated two typical use cases of how this can help and speed up theory development: by generating interesting basic lemmas in *exploration mode* or as a lemma suggestion mechanism for a stuck proof attempt in *proof mode*. The user can control what is discovered by varying the background theory, and by varying Hipster's "routine reasoning" and "difficult reasoning" tactics; only lemmas that need difficult reasoning (e.g. induction) are presented.

Hipster complements tools like Sledgehammer: by discovering missing lemmas, more proofs can be tackled automatically. Hipster succeeds in automating inductive proofs and lemma discovery for small, but non-trivial, equational Isabelle/HOL theories. The next step is to increase its scope, to conditional equations and to bigger theories: our goal is a practical automated inductive proof tool for Isabelle/HOL.

Acknowledgements The third author's research was supported by the EU FP7 Collaborative project *PROWESS*, grant number 317820.

References

1. Jasmin Christian Blanchette. Personal communication, 2013.
2. Bruno Buchberger. Theory exploration with Theorema. *Analele Universitatii Din Timisoara, ser. Matematica-Informatica*, 38(2):9–32, 2000.
3. Bruno Buchberger, Adrian Creciun, Tudor Jebelean, Laura Kovacs, Temur Kutsia, Koji Nakagawa, Florina Piroi, Nikolaj Popov, Judit Robu, Markus Rosenkranz, and Wolfgang Windsteiger. Theorema: Towards computer-aided mathematical theory exploration. *Journal of Applied Logic*, 4(4):470 – 504, 2006. Towards Computer Aided Mathematics.
4. Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of ICFP*, pages 268–279, 2000.
5. Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In *Proceedings of the Conference on Auomtated Deduction (CADE)*, volume 7898 of *LNCS*, pages 392–406. Springer, 2013.

6. Koen Claessen, Nicholas Smallbone, and John Hughes. QuickSpec: Guessing formal specifications using testing. In *Proceedings of TAP*, pages 6–21, 2010.
7. Lucas Dixon and Jacques D. Fleuriot. Higher order rippling in IsaPlanner. In *Proceedings of TPHOLs*, volume 3223 of *LNCS*, pages 83–98, 2004.
8. Florian Haftmann and Lukas Bulwahn. Code generation from Isabelle/HOL theories. `http://isabelle.in.tum.de/doc/codegen.pdf`, 2013.
9. Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In Matthias Blume, Naoki Kobayashi, and Germn Vidal, editors, *Functional and Logic Programming*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010.
10. Jonathan Heras, Ekaterina Komendantskaya, Moa Johansson, and Ewen Maclean. Proof-pattern recognition and lemma discovery in ACL2. In *Proceedings of LPAR*, volume 8312 of *LNCS*, pages 389–406. Springer-Verlag, 2013.
11. Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA)*, number NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, 2003.
12. Andrew Ireland and Alan Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16:79–111, 1996.
13. Moa Johansson, Lucas Dixon, and Alan Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 47(3):251–289, 2011.
14. Daniel Kuhlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. MaSh: Machine learning for Sledgehammer. In *Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 35–50. Springer, 2013.
15. Roy L. McCasland, Alan Bundy, and Patrick F. Smith. Ascertaining mathematical theorems. *Electronic Notes in Theoretical Computer Science*, 151(1):21 – 38, 2006.
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. Lawrence C Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. *IWIL-2010*, 2010.