Into the Infinite - Theory Exploration for Coinduction*

Sólrún Halla Einarsdóttir[1], Moa Johansson[1], Johannes Åman Pohjola[2]

[1] Chalmers University of Technology, Gothenburg, Sweden.
{slrn, moa.johansson}@chalmers.se
[2] Data61 / CSIRO, Sydney, Australia. johannes.amanpohjola@data61.csiro.au

Abstract. Theory exploration is a technique for automating the discovery of lemmas in formalizations of mathematical theories, using testing and automated proof techniques. Automated theory exploration has previously been successfully applied to discover lemmas for inductive theories, about recursive datatypes and functions. We present an extension of theory exploration to coinductive theories, allowing us to explore the dual notions of corecursive datatypes and functions. This required development of new methods for testing infinite values, and for proof automation. Our work has been implemented in the Hipster system, a theory exploration tool for the proof assistant Isabelle/HOL.

1 Introduction

Coinduction and corecursion are dual notions to induction and recursion that admit the specification of potentially infinite structures and functions that operate on them. Their many applications in theoretical computer science include, to name a few: defining and verifying behavioral equivalence of processes [21], Hoare logic for non-terminating programs [23], total functional programming in the presence of non-termination [29], and accounting for lazy data in functional languages like Haskell. Recently, support for coinduction in proof assistants has matured significantly, with powerful definitional packages and reasoning tools [5, 6, 1].

In this paper, we extend a technique, called *theory exploration* [7], and present a tool that automatically discovers and proves equational properties about corecursive functions in the proof assistant Isabelle/HOL [24], a widely used interactive theorem proving system featuring both automated and interactive proof techniques. The purpose of theory exploration is to automate the discovery of basic lemmas when, for instance, developing a new theory. The human user can then focus on inventing and proving more complex conjectures, using the automatically generated background lemmas. As an appetizer, consider this simple example of an Isabelle theory:

* The final authenticated version is available at https://doi.org/10.1007/978-3-319-99957-9_5

```
codatatype (sset: 'a) Stream = SCons (shd: 'a) (stl: "'a Stream")

primcorec smap :: "('a ⇒ 'b) ⇒ 'a Stream ⇒ 'b Stream" where
  "smap f xs = SCons (f (shd xs)) (smap f (stl xs))"

primcorec siterate :: "('a ⇒ 'a) ⇒ 'a ⇒ 'a Stream" where
  "siterate f a = SCons a (siterate f (f a))"

cohipster smap siterate — tell Hipster to explore these functions
```

The theory above defines the codatatype `Stream` of infinite sequences, the function `smap` that maps a function onto every element of a stream, and the function `siterate` that given a function f and an initial element x generates the sequence $f(x), f(f(x)), f(f(f(x))), \ldots$. The verbatim output of our tool, Hipster, is as follows:

```
lemma lemma_a [thy_expl]: "smap y (siterate y z) = siterate y (y z)"
  by (coinduction arbitrary: y z rule: Stream.coinduct_strong)
    auto

lemma lemma_aa [thy_expl]: "SCons (y z) (smap y x2) = smap y (SCons
z x2)"
  by (coinduction arbitrary: x2 y z rule: Stream.coinduct_strong)
    simp

lemma lemma_ab [thy_expl]: "smap z (SCons y (siterate z x2)) = SCons
(z y) (siterate z (z x2))"
  by (coinduction arbitrary: x2 y z rule: Stream.coinduct_strong)
    (simp add: lemma_a)
```

This Isabelle snippet, when pasted into the theory (simply by a mouse-click), proves the discovered laws about `smap` and `siterate` by coinduction. The first lemma, `lemma_a`, may appear familiar as it describes the *map-iterate property* [3]. The whole process of generation and proof took Hipster less than 10 seconds on a regular laptop computer. Moreover, the generated proofs are formal proofs, machine-checked down to the axioms of higher-order logic.

Note that at no point did the user need to supply the conjectures or proofs. Hipster uses a specialized conjecture discovery subsystem, called QuickSpec [28], which heuristically generates type-correct terms and uses automated testing to invent interesting candidate lemmas. We give a brief introduction to QuickSpec in Section 2, along with a lightweight introduction to coinduction.

Earlier versions of Hipster [16, 14] supported only induction and recursive datatypes. The main difference when we also treat codatatypes is in the testing phase, when conjectures are generated. Naively testing and evaluating terms for equivalence cannot be done in the same way as for regular datatypes, since instances of a codatatype like `Stream` are infinite, so testing would not terminate. Our solution to this conundrum is that for testing purposes, we generate step-indexed *observer functions* for the codatatypes under consideration. These operate

on a copy of the codatatype with an extra nullary constructor, that we return when the step-index reaches 0. The step-indexing guarantees that testing will terminate. Section 3 describes this in more detail, along with our approach to coinductive proof exploration.

We evaluate our tool by testing it on several examples of codatatypes and corecursive functions in Section 5. Results are encouraging: we can discover and prove many well-known and useful properties. Similar theory exploration systems can be found in the literature [20, 15, 22, 9], but ours is the first system capable of discovering and proving properties of coinductive types and corecursive functions. We integrate inductive and coinductive reasoning, so that in a theory featuring both recursion and corecursion, both inductive and coinductive proofs can be discovered even when one depends on the other. The source code and examples are available online.[3]

2 Background

We give a brief introduction of coinduction for readers unfamiliar with the concept, followed by an introduction to the proof assistant Isabelle/HOL and the Hipster theory exploration system.

Coinduction. Coinduction is the mathematical dual of structural induction, relying on deconstructing structures top-down instead of constructing them bottom-up as induction does. Consider lists with elements of type `a`, defined by: `List a = Nil | Cons a (List a)`.
The inductive reading of this declaration is that it specifies everything that can be constructed from the empty list `Nil` in a finite number of steps, by using the `Cons` constructor to add elements. The coinductive reading is that it specifies everything that is either `Nil` or can be decomposed ("destructed") into a head and a tail, where the tail is either `Nil` or something that can be destructed into another head and tail, and so on. The latter reading encompasses not only `Nil`-terminated lists, but also infinite lists built from `Cons` only. We say that the first reading defines a *datatype* while the second defines a *codatatype*.

Since codata need not bottom out in a base case, proof by induction does not apply; instead we resort to the dual notion of coinduction, which allows us to prove equalities between elements x, y of a codatatype by exhibiting a *candidate relation* R such that $x\ R\ y$ and R is closed under destruction. For example, here is the coinduction principle for the `Stream` type introduced in Section 1:

$$\frac{R\ s\ s' \qquad \forall s_1, s_2.\ \dfrac{R\ s_1\ s_2}{shd\ s_1 = shd\ s_2 \wedge R\ (stl\ s_1)\ (stl\ s_2)}}{s = s'}$$

In words: to show that $s = s'$, we must prove that for all pairs s_1, s_2 related by R, s_1 and s_2 have the same heads and R-related tails. Interested readers can find a more detailed introduction to coinduction in [27] or [13].

[3] https://github.com/moajohansson/IsaHipster

Isabelle/HOL. Isabelle/HOL is an interactive proof assistant for higher-order logic [24]. Users write definitions and proofs in *theory files*, which are checked by running them through Isabelle's small trusted logical kernel to ensure each step in a proof is correct. More complex proof techniques, called *tactics*, can be built up using combinations of basic inference rules from the trusted kernel. Isabelle is an *interactive system*, meaning that there are both automated and semi-automated tactics available. An example of the former is the *simplifier*, which performs equational reasoning automatically. An example of the latter is Isabelle's (co)induction tactics, which applies a (maybe user given) induction rule to a subgoal while leaving it to the user how to prove the resulting subgoals. Sledgehammer is a useful tool in Isabelle which allows outsourcing proofs to fully automated external first-order (FO) or SMT-solvers [25]. When the external provers report back, the proof is reconstructed inside Isabelle's trusted kernel. In our work on Hipster, we combine Isabelle's interactive tactics with Sledgehammer to provide automation for (co)inductive proofs.

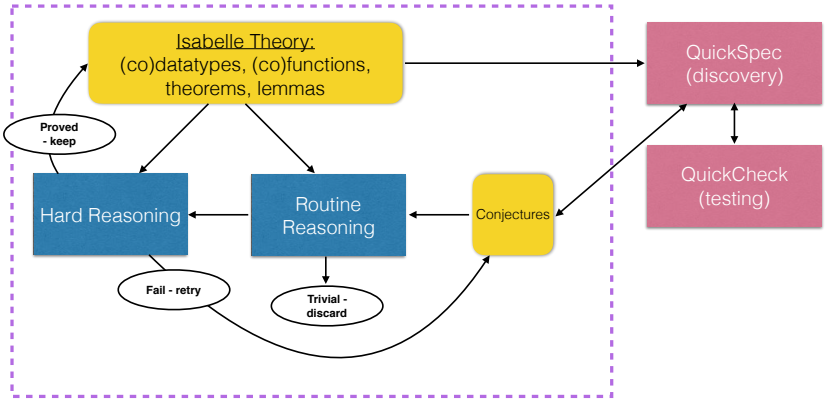

Fig. 1. The architecture of the Hipster system.

Hipster. The architecture of the Hipster system is shown in Figure 1. Hipster outsources conjecture generation to the external tool QuickSpec. QuickSpec generates type-correct terms in order of size, up to a given limit. At each step, it evaluates the terms on randomly generated test data, using the property-based testing tool QuickCheck [8]. Based on the results of testing, terms are divided into equivalence classes from which equational conjectures are extracted. For a full description of QuickSpec's conjecture generation algorithm and its heuristics we refer the reader to [28]. The conjectures produced by QuickSpec are then read back into the Isabelle/HOL environment for proof. The conjectures have been thoroughly tested at this point, so we have quite good reasons to believe they may actually be true. However, not all of them might be considered interesting

by a human. In particular, statements that have trivial proofs are rarely exciting. Hipster therefore takes two reasoning strategies as parameters: *routine reasoning* (often just rewriting), and *hard reasoning* (for instance coinduction). Depending on the exact configuration of the routine and hard reasoning strategies, we can tweak Hipster to produce slightly different output: the conjectures that follow from using only routine reasoning are discarded, while those proved by the hard reasoning strategy are reported back to the user. Whenever Hipster proves a lemma, it may use it in subsequent proofs. This means that during exploration, its automated proof strategies become more powerful as more lemmas are found. Should some conjecture fail to be proved by either of the proof strategies, it is also presented to the user, who can try a manual proof.

3 Testing Infinite Structures

Recall from Section 2 that Hipster's conjecture generation subsystem, QuickSpec, relies on being able to test terms on randomly generated values. When a codatatype has no finite instances, as in the case of streams, QuickSpec cannot directly check the equality of any of the generated terms, since that would take an infinite amount of time due to their infinite size. Thus testing will not work.

When an Isabelle user invokes Hipster on a coinductive theory, an *observer type* and *observer function* are generated for every type under consideration. These types and functions ensure that QuickSpec only tests a (randomly chosen) finite prefix of any infinite values, using support for *observational equivalence*. This allows Hipster to discover lemmas about codatatypes without finite instances.

Observational Equivalence in QuickSpec. When used interactively through its Haskell interface, QuickSpec supports observational equivalence to deal with types that for instance have no finite instances, and thus cannot be directly compared [28]. Note that in this case, the user must define a function for observing such a type and state that two values of the type are equivalent if all such observations make them equal. We have extended this functionality by developing a method to *automatically generate* observer functions for the codatatypes being explored and added it to the interface between Hipster and QuickSpec.

More specifically, observer functions are used as follows: For any type T, QuickSpec can be given an observer function of type $Obs \to T \to Res$, where Obs can be any type that QuickSpec can generate random data for, and Res any type that can be compared for equality. QuickSpec will then include a random value of type Obs as part of each test case, and will compare values of type T by applying this observer function using the random value of type Obs and comparing the resulting values of type Res. For instance, we can define an observer function for streams:

$$obsStream :: Int \to Stream \to List,$$

where $obsStream\ n\ s$ returns a list containing the first n elements of the stream s. If we supply this observer function to QuickSpec it will generate a random integer n for each test case where streams are to be observed, and assume that two streams are equal if their first n elements are equal in every case.

Generating Observer Functions. For Hipster, we want to relieve the user of having to define the observer function by hand, and instead generate it automatically. Our method of generating observer functions is inspired by the *Approximation Lemma* [2, 12]. Here, a so called *approximation function*, *approx*, is defined in the same way as the recursive identity function for a given type, except that it has an additional numeric argument which is decremented at each recursive call. The lemma states that $a = b$ if $approx\ n\ a = approx\ n\ b$ for all values of n. For the `Stream` type introduced in Section 1, the approximation function is defined as:

$$approx\ (n+1)\ xs = SCons\ (shd\ xs)\ (approx\ n\ (stl\ xs))$$

The function is undefined for $n = 0$ and therefore returns a partial structure, for instance, if *zeroes* is a stream of zeroes then $approx\ 1\ zeroes$ evaluates to the partial stream $SCons\ 0\ \bot$, where $\bot$ represents an undefined value.

To make our solution practical we, instead of using the undefined value $\bot$, generate a new type that has the same structure as the type being observed, but with an additional nullary constructor. For example, the generated observation type for a stream is:

$$OStream\ a = OSCons\ a\ (OStream\ a)\ \mid\ NullConsStream$$

We then generate an observer function for a given type T with an observer type $ObsT$ in the following manner:

$$\begin{aligned} obsFunT &:: Nat \to T \to ObsT \\ obsFunT\ 0\ \_ &= NullConsT \\ obsFunT\ n\ t &= approx'\ n\ t \end{aligned}$$

where $approx'$ is like the recursive identity function for T except that it replaces each constructor occurring in t with the equivalent constructor for $ObsT$, and the *fuel* parameter n is decremented at every recursive call, ensuring we will only attempt to observe a finite prefix. As an example, an observer function for streams using the observer type from above is shown below:

$$\begin{aligned} &obsFunStream :: Nat \to Stream\ a \to OStream\ a \\ &obsFunStream\ 0\ \_ = NullConsStream \\ &obsFunStream\ n\ (SCons\ x\ xs) = OSCons\ x\ (obsFunStream\ (n-1)\ xs) \end{aligned}$$

Some care needs to be taken when decrementing the numeric fuel argument which determines how much more of the structure should be observed, as using $n - 1$ in every step results in testing being too slow for structures with larger branching factors, such as trees. For now, we use a heuristic measure which decrements n to *n/#constructors -1* in each recursive call. For *OStream*, this is simply $(n - 1)$, while for e.g. binary trees, defined:

$$Tree\ a = TNode\ a\ (Tree\ a)\ (Tree\ a)$$

with an observer type defined:

$$OTree\ a = OTNode\ a\ (OTree\ a)\ (OTree\ a)\ \mid\ NullConsTree$$

the fuel counter is decremented to $n/2-1$ for each branch, as seen in the observer function definition below:

$$\begin{aligned}
&obsFunTree :: Nat \to Tree\ a \to OTree\ a \\
&obsFunTree\ 0\ _\ =\ NullConsTree \\
&obsFunStream\ n\ (TNode\ x\ l\ r)\ = \\
&\quad OTNode\ x\ (obsFunTree\ (n/2-1)\ l)\ (obsFunTree\ (n/2-1)\ r)
\end{aligned}$$

4 Automating Proofs of Coinductive Lemmas

Isabelle/HOL features a built-in `coinduction` tactic that applies a coinduction principle to a goal, with the candidate relation instantiated to be the singleton relation containing the equation in the conclusion. After applying this tactic the user must decide how to finish the proof after the coinductive step. However, the ability to automatically prove lemmas without user involvement is crucial in lemma discovery by automated theory exploration. Therefore we have extended Hipster with an automated tactic for proving coinductive lemmas. In order to do this, we must automatically determine the parameters for our call to Isabelle/HOL's coinduction tactic, and then automate the subgoal proofs.

Automatically Determining Parameters. Isabelle/HOL's coinduction tactic has parameters to set which variables are *arbitrary*, meaning that they appear universally quantified in the candidate relation (and hence existentially quantified in the conclusion of the resulting subgoal). It also has an optional parameter to specify which coinduction rule to use.

Our default setting is to set all free variables in the current goal as arbitrary. This yields weaker proof obligations, at the expense of introducing existential quantifiers in the goal, which is sometimes less automation-friendly since it may require guessing an instantiation to discharge the goal. Our experience is that setting at least some variables to arbitrary is necessary for all but the most trivial of proofs; for the rest, the goal statements are simple enough that the extra existentials do not cause any difficulty in practice.

The built-in `coinduction` tactic also has an optional parameter to specify what coinduction rule should be used for the proof. We must again make a tradeoff between one that can be applied to prove as wide a range of lemmas as possible, such as coinduction up-to the codatatype's companion function [26]; and one that yields simple and automation-friendly subgoals, such as the (weak) coinduction principle associated with the datatype.

For reasoning about functions defined with primitive corecursion, we find that the strong coinduction principle generated by the datatype package works well in practice. It allows one to close the proof by proving either equality or membership

in the candidate relation. For example, here is the strong coinduction principle for the `Stream` type defined in Section 1:

$$\frac{R\ s\ s' \qquad \forall s_1, s_2\ \dfrac{R\ s_1\ s_2}{shd\ s_1 = shd\ s_2 \wedge\ (R\ (stl\ s_1)\ (stl\ s_2) \vee\ stl\ s_1 = stl\ s_2)))}}{s = s'}$$

Note that the (weak) coinduction principle shown in Section 2 differs by omitting the right-hand side $stl\ s1 = stl\ s2$ of the disjunction. The extra disjunction is lightweight enough not to confuse the simplifier, and the equality has very important consequences: it allows equations that have previously been proven by coinduction to be re-used in the proof, without having to include them in the candidate relation. This allows us to automatically prove, e.g., the associativity of append on lazy lists as seen in Section 5.1.

The recent AmiCo definitional package by Blanchette et al. [4] allows a form of non-primitive corecursion where corecursive calls may be guarded by *friends* in addition to constructors. A friend is a function that consumes at most one constructor to produce a constructor. For functions with friend-guarded corecursive calls, the strong coinduction rule often results in an unsuccessful proof attempt: terms on the shape required by the candidate relation tend to occur as arguments to friends rather than at top-level. Fortunately, the AmiCo package generates a coinduction principle up-to friendly contexts covering precisely this use case. Hence we prioritize such coinduction principles over the strong coinduction principle whenever they are relevant, i.e., whenever the goal state mentions a function symbol defined using non-primitive corecursion.

Proving Subgoals. After applying coinduction, Hipster's `simp_or_sledgehammer` tactic is applied to the current proof state in an attempt to prove the remaining subgoals and conclude the proof of the lemma. This tactic first attempts to complete the proof using Isabelle's automatic simplification procedure `simp`. If this does not suffice it uses Isabelle's automated proof construction tool Sledgehammer [25] to attempt to construct a proof. Since Sledgehammer is quite powerful, this tactic is sufficient to conclude the proofs of a wide range of lemmas.

Mixed Induction and Coinduction In practice, theories are neither purely inductive nor purely coinductive — coinductive definitions of datatypes and functions may use auxiliary inductive definitions, and vice versa. In order to cope with such theories, it is important that we integrate Hipster's inductive and coinductive functionality. For conjecture discovery, this integration comes for free since Isabelle's code generator maps both data and codata to identical Haskell code.

For proof search, we must decide whether to tackle our conjectures using induction, coinduction or both. For this, we use a simple heuristic that appears to work well in practice: if the conjecture contains a free variable whose type has an induction principle, we invoke the inductive proof search procedure; if the left- and right-hand sides of the conjecture are of a type that has a coinduction

principle, we invoke the coinductive proof search; if both, we try both and keep the first successful proof attempt. This architecture allows us to find proofs of inductive lemmas that require coinductive auxiliary lemmas, such as the fact that `append` distributes over the `to_llist` function on finite lists (see Section 5).

5 Evaluation and Results

We apply Hipster to several theories of common codatatypes found in the literature: lazy lists, extended natural numbers, streams, and two kinds of infinite trees. Our goal is to demonstrate how a user can invoke Hipster to discover useful lemmas in their coinductive theory development, showing that our method for testing infinite structures, as described in Section 3, is effective in discovering coinductive properties and that our automated coinduction tactic, described in Section 4, is effective in proving those properties.

We restrict each Hipster call to a small number of functions, to explore how those functions relate to each other, rather than exploring all the functions in a theory at once. This is how we envision typical users will interact with the tool, since in practice it tends to yield quicker and more relevant results.

The evaluation was performed with Isabelle 2017 using Isabelle/jEdit, on a ThinkPad X260 laptop with a 2.5GHz Intel i7-6500U processor and 16GB of RAM running 64-bit Linux. The Isabelle theory files used to attain these results are available online[4].

5.1 Case Study: Lazy Lists and Extended Natural Numbers

In this section we demonstrate the results attained when using Hipster to explore a theory of lazy lists (lists of potentially infinite length). We define some common functions for this type: *lappend* to append two lazy lists, a map function *lmap*, *iterates* which generates a lazy list by iteratively applying a function to an element, *llist_of* which maps a standard Isabelle/HOL list to a lazy list, *llength* which returns the length, and *ltake* which takes a given number of elements. We also define a codatatype *ENat* for extended natural numbers (natural numbers of potentially infinite size) and an addition function *eplus* on *ENat*s.

We check which of the lemmas we discover are stated and proved in the Coinductive library [18] in the archive of formal proofs[5], which is a collection of formalizations about coinductive types and functions. For the extended naturals we refer to the *Extended_Nat* theory from the Isabelle/HOL library [6]. Since the lemmas in these libraries have been collected and hand-proved by Isabelle experts, we conclude that they must be interesting and/or useful for Isabelle theory development.

Table 1 shows the results of exploration on this theory. The column args shows the names of the functions explored in the particular Hipster call, Expl is

[4] https://github.com/moajohansson/IsaHipster/tree/master/benchmark/AISC18
[5] https://www.isa-afp.org/
[6] http://isabelle.in.tum.de/library/HOL/HOL-Library/Extended_Nat.html

the amount of time (in seconds) spent in exploration and testing, Expl+Proof is the amount of time (in seconds) spent in exploration, testing, and proving, # properties shows the number of properties Hipster discovers, # library lemmas shows how many of those properties are lemmas stated and proved in the libraries mentioned above. For these experiments, Hipster's routine tactic was configured to only do simplification, and the hard tactic was our automated coinduction and induction tactic as described in Section 4.

In our 13 calls to Hipster, we discover 33 coinductive or inductive properties. Of these 33 properties, 13 are stated and proved as lemmas in Isabelle libraries, leading us to believe that they are of interest to Isabelle users. Of the other 20, most are rather trivial consequences of function definitions and/or other discovered lemmas, which our routine tactic does not suffice to prove. Some of the discovered properties may however be interesting to users despite not appearing in the libraries, for instance that $llength(lappend\ xs\ ys) = llength(lappend\ ys\ xs)$.

The discovered properties include the associativity of append, $lappend\ (lappend\ x\ y)\ z = lappend\ x\ (lappend\ y\ z)$, and that mapping preserves length, $llength\ (lmap\ f\ x) = llength\ x$. The exploration involving $llist\_of$, which maps a standard list to a lazy list, results in lemmas showing the correspondence between our lazy list functions and Isabelle/HOL's built-in list functions, for example $lmap\ f\ (llist\_of x) = llist\_of\ (map\ f x)$. The previous lemma is proved by induction, demonstrating Hipster's capabilities in exploring mixed inductive and coinductive theories.

cohipster args	Expl	Expl+Proof	# properties	# library lemmas
lappend	2.5s.	25s.	4	2
lmap	3.2s.	7s.	3	0
lappend lmap	4.1s.	17s.	1	1
llist_of lappend append	4.9s.	28s.	1	1
llist_of lmap map	4.9s.	21s.	1	1
llength	2.1s.	2s.	1	0
llength lmap	4.0s.	11s.	1	1
eplus	2.9s.	39s.	4	3
llength lappend eplus	5.2s.	87s.	5	1
ltake	4.1s.	76s.	7	0
ltake lmap	5.7s.	23s.	2	1
lmap iterates	4.2s.	18s.	2	1
lappend iterates	4.6s.	15s.	1	1

Table 1. An overview of the results of exploring our lazy list theory.

All of the discovered properties are proved by our automated proof tactic, except for the commutativity of *eplus*. This was due to our rather short timeout for Sledgehammer, which was just set to 10s. in this experiment. If we allow a 30s. timeout (which is the standard when Sledgehammer is used interactively), a proof is found. As can be seen from Table 1, the time it takes for Hipster to discover and prove properties varies between 2-90 seconds. As all calls took less

than 90 seconds to complete, and most took less than a minute, we can see that the user does not have to wait very long for Hipster to come up with lemmas for their functions. We believe that for most Isabelle users, making a call to Hipster would be much faster than writing down and proving the same lemmas manually, not to mention coming up with them. In Table 1 we also compare the runtime of the calls: most of the time is spent trying to prove properties (we give each call to Sledgehammer a timeout limit of 10 seconds), while the time to discover and test the properties is just a few seconds. There is however a configuration option in Hipster for very impatient users to only do exploration, leaving the proofs to the user altogether.

5.2 Case Study: Stream Laws

We already saw in Section 1 that Hipster can discover and prove the *map-iterate* property for streams. In this section, our aim is to quantify the degree to which Hipster discovers stream equations that a human would find interesting. That is of course subjective, but for the purposes of this section we operationalize "interesting" as being any of the 18 laws of Hinze's Stream Calculus [11], which according the the author *"provides an account of the most important properties of streams"*. Of the 18 laws given by Hinze, three are beyond the scope of Hipster's current capabilities: lambda-expressions are not supported, nor are conditional statements with term depth > 1 in the antecedent. The remaining 15 are all equational statements. With respect to these 15 laws, we analyze Hipster's precision (percentage of the lemmas we find that are among Hinze's laws) and recall (percentage of Hinze's laws that we find).

First, we will briefly recapitulate the relevant notation. *pure* x denotes a stream where every element is x. $\diamond$ is lifted function application, defined by the observations $hd(f \diamond x) = (hd\ f)\ (hd\ x)$ and $tl(f \diamond x) = (tl\ f) \diamond (tl\ x)$. The interleaving of two streams x, y is written $x \curlyvee y$. Tabulation, written *tabulate* f, is the stream whose n:th element is $f(n)$. Lookup, written *lookup* $s\ n$, is the n:th element of stream s. *zip* $x\ y$ merges two streams into a stream of pairs. *recurse* is defined by the observations $hd(recurse\ f\ a) = a$ and $tl(recurse\ f\ a) = map\ f\ (recurse\ f\ a)$. Unfolding satisfies $hd(unfold\ g\ f\ a) = g\ a$ and $tl(unfold\ g\ f\ a) = unfold\ g\ f\ (f\ a)$.

The results are shown in Table 2. The lemmas' precision, recall and time have been explored together by invoking Hipster with every function mentioned in each lemma; e.g., to search for laws 7-9 we invoke **cohipster** `map zip fst snd`. We also report total precision and recall over all such invocations at the bottom. For these experiments, Hipster has been configured to use a Sledgehammer timeout of 10s, a routine tactic that does only simplification, and a hard tactic that tries coinduction and induction, in each case followed by simplification or sledgehammer, as described in Section 4.

We see that in total, Hipster discovers 9 out of the 15 properties in scope, i.e. 60% recall. Note in particular property 13, where Hipster discovers a proof by induction, and property 14, where Hipster discovers a proof by coinduction up-to friendly contexts. The 21% overall precision can be improved by using a more powerful routine tactic, such as simplification interleaved with stream expansion.

	Property	Found	Precision	Recall	Time
1	$pure\ id \diamond u = u$	X			
2	$pure(\circ) \diamond u \diamond v \diamond w \diamond u = u$	–	22%	67%	44s.
3	$pure\ f \diamond pure\ x = pure\ (f\ x)$	X			
4	$u \diamond pure\ x = pure\ (\lambda f.\ f\ x) \diamond u$				
5	$map\ id\ x = x$	X	50%	100%	29s.
6	$map\ (f \circ g)\ x = map\ f\ (map\ g\ x)$	X			
7	$map\ fst\ (zip\ s\ t) = s$	–			
8	$map\ snd\ (zip\ s\ t) = t$	–	0%	0%	255s.
9	$zip\ (map\ fst\ p)\ (map\ snd\ p) = p$	–			
10	$pure\ a \curlyvee pure\ a = pure\ a$	X	25%	50%	18s.
11	$(s_1 \diamond s_2) \curlyvee (t_1 \diamond t_2) = (s_1 \curlyvee t_1) \diamond (s_2 \curlyvee t_2)$	–			
12	$map\ f\ (tabulate\ g) = tabulate\ (f\ \circ\ g)$	X	100%	100%	87s.
13	$f(lookup\ t\ x) = lookup\ (map\ f\ t)\ x$	X	33%	100%	57s.
14	$recurse\ f\ a = iterate\ f\ a$	X	33%	100%	73s.
15	$map\ h \circ iterate\ f_1 = iterate\ f_2 \circ h \Longleftarrow h \circ f_1 = f_2 \circ h$				
16	$unfold\ hd\ tl\ x = x$	–	0%	0%	21s.
17	$unfold\ g\ f \circ h = unfold\ g'\ f' \Longleftarrow g \circ h = g' \wedge f \circ h = h \circ f'$				
18	$map\ h\ (unfold\ g\ f\ x) = unfold\ (h \circ g)\ f\ x$	X	50%	100%	18s.
			21%	**60%**	**602s.**

Table 2. An overview of the stream properties discovered and proved by Hipster. Lemmas in gray are not in scope.

The properties that are in scope, but not discovered, are all attributable to QuickSpec's heuristics for restricting the search space. Properties involving variables denoting streams of functions such as Property 2 cannot be tested, and instantiation of type variables is restricted in ways that rule out, e.g., conjectures where fst occurs as an argument to *map*. It seems difficult to lift these restrictions in ways that do not make the search space intractable — this would be an interesting direction for future work.

5.3 Case Study: Infinite Trees

We have experimented with two different kinds of corecursive trees: A codatatype representing an infinitely deep binary tree, and another representing an infinitely deep *rose tree*, with arbitrary branching at each node. The purpose here is to demonstrate Hipster on a different kind of codatatype than the previous case-studies. Hipster was configured to use simplification as the *routine tactic*, and as the *hard tactic*, either just Sledgehammer or coinduction followed by Sledgehammer.

Infinite Binary Trees. We define an infinite depth binary tree as follows:

```
codatatype 'a Tree = Node (lt: "'a Tree") (lab: 'a) (rt: "'a Tree")
```

We defined three functions over this codatatype: *mirror* (which switches the left and right branches of each node), *tmap* which applies a function to each label in

the tree and *tsum* which sums the labels of a tree of natural numbers. A summary of the results is given in Table 3. Hipster discovers the expected properties about the given functions (associativity, distributivity etc.) as well as a few additional properties which perhaps are of less interest. We note that these are presented to the user as Isabelle's simplifier is a rather weak tactic in this context, while another choice for the routine tactic would have pruned out more properties.

cohipster args	Expl	Expl+Proof	Properties Discovered	Proved
mirror	3.4s.	39s.	*mirror (mirror y) = y*	coinduction+simp
			+ 3 more proved by Sledgehammer	
mirror tmap	4.3s.	35s.	*tmap z (mirror x) = mirror (tmap z x)*	coinduction+smt
mirror tsum	6.1s.	112s.	*tsum y x = tsum x y*	coinduction+smt
			tsum (tsum x y) z = tsum x (tsum y z)	coinduction+smt
			mirror (tsum y (mirror x)) = tsum x (mirror y)	coinduction+smt
			tsum (mirror x) (mirror y) = mirror (tsum x y)	Sledgehammer (using above lemmas)
			+ 2 more proved by Sledgehammer	

Table 3. Overview of properties discovered about infinite depth binary trees. Due to space restrictions mainly properties proved by coinduction are listed, full results are available online.

Rose Trees: a Nested Codatatype. We also conducted an experiment with a nested codatatype representing arbitrarily branching *rose trees*:

```
codatatype 'a RoseTree = Node (lab: 'a) (sub: "'a RoseTree list")
```

We defined functions *mirror* (reversing the list of subtrees), *tmap* (mapping a function over the labels of each node) and *tsum* (summing the labels of a tree of natural numbers). Note that unlike for the infinite binary trees, *mirror* and *tmap* are not corecursive.

For this theory, we noticed that the runtimes varied a great deal from run to run of the same command. For example, in a series of runs of Hipster on the function mirror only, the runtime varied from as little as 21 seconds to as much as 125 seconds. This is due to how our observer function interacts with the random length lists being generated for the branches at each node. It decreases its fuel linearly in this case, so if the list is long observing each child tree recursively is time-consuming. Implementing smarter observer functions, for instance taking length of the list of a node's child trees into account to only observe an appropriately small subtree of each child, is future work.

As can be seen in Table 4, only a few properties are proved automatically (by Sledgehammer, no coinduction needed). This is because our automated coinduction tactic is not flexible enough to deal with nested datatypes. We believe a customized tactic, also able to perform some form of nested induction over the list of branches, would do a better job, but such domain specific tactics are left as further work at this stage.

cohipster args	Expl+Proof	Properties Discovered	Proved
mirror	29s.	*mirror (mirror y) = y*	Sledgehammer
mirror tmap	102s.	*tmap z (mirror x) = mirror (tmap z x)*	Sledgehammer
mirror tsum	597s.	*tsum (mirror x) (mirror x) = mirror (tsum x x)*	Sledgehammer
		tsum y x = tsum x y	no
		tsum (tsum x y) z = tsum x (tsum y z)	no
		+ 4 more unproved about tsum/mirror	

Table 4. Overview of properties discovered about rose trees. Note that timings here are from one sample run, and can vary quite a lot due to randomness in testing.

6 Related Work

There is substantial recent work on making Isabelle/HOL more expressive for working with codatatypes and corecursive functions [5, 4]. Our extension to Hipster can help Isabelle/HOL users who want to program with these new methods discover and prove new properties about their theories.

There has been prior work on automating coinductive proofs and reasoning. In [17] Leino and Moskal present a method for automated reasoning about coinductive properties in the Dafny verifier. CIRC [19] is a tool for automated inductive and coinductive theorem proving which uses circular coinductive reasoning. It has been successfully used to prove many properties of infinite structures such as streams and infinite binary trees. However, none of the other systems has the theory exploration capabilities of Hipster.

In the setting of resolution for Horn clause logic with coinductive entailment, Fu et al. [10] present a method for automatically generating appropriate candidate lemmas for proving such entailments. The application is to devise a method for e.g. type class resolution in Haskell that is stronger than cycle detection. Whereas Hipster uses testing to generate candidate lemmas, Fu et al. uses the structure of partial proof attempts. Given a partially unfolded resolution tree, the candidate lemma that gets generated states that the root of the tree is entailed by the conjunction of all leaves that mention fewer symbols than the root. This is also unlike Hipster in that Hipster strives for lemmas that will be generally useful for any further theory development using the types and functions under consideration, whereas Fu et al. are interested in finding which lemmas, were they true, could be used to prove a particular sequent.

IsaCoSy [15] and IsaScheme [22] are other theory exploration systems for Isabelle/HOL, both of which focus on the discovery and proof of inductive properties. MATHsAiD [20] is a tool for automated theorem discovery, aimed at aiding mathematicians in exploring mathematical theories. It can discover and prove theorems whose proofs consist of logical and transitive reasoning as well as induction. Hipster is the first theory exploration system capable of discovering and proving coinductive properties. Furthermore, it is considerably faster than IsaCoSy and IsaScheme thanks to using QuickSpec as a backend [9].

7 Conclusion

We have extended the theory exploration system Hipster with the capabilities to discover and prove not only inductive lemmas, but also lemmas in coinductive theories involving potentially infinite types such as streams, lazy lists and trees. We have shown that the system can discover and prove many standard lemmas about these codatatypes. This goes beyond the capabilities of previous theory exploration systems, that do not consider coinduction at all.

In the long term, we envision that invoking a theory exploration system such as Hipster will be a natural first step for the working proof engineer when developing a new theory. This nicely complements tools like Isabelle's Sledgehammer. In a new theory, Sledgehammer is unlikely to be of much help until we have proven at least some basic lemmas, which is exactly what theory exploration can automate.

There are many interesting directions for further work. As seen in the case study on rose trees, we would benefit from specialized observation functions and proof methods for nested (co-)datatypes. The case studies in this paper are mostly in the domain of lazy data in the style of functional programming. It would be interesting to explore if we can extend our work to other uses of coinduction. For example, discovering algebraic laws about coinductively defined behavioral equivalences, or discovering Hoare triples about non-terminating programs. This would require developing a technique to test *relations* as opposed to functions.

Acknowledgements. The authors would like to thank Nicholas Smallbone for technical assistance with QuickSpec. The first author was partially supported by the GRACeFUL project, grant agreement No 640954, which has received funding from the European Union's Horizon 2020 research and innovation program.

References

1. A. Abel and B. Pientka. Well-founded recursion with copatterns and sized types. *J. Funct. Program.*, 26:e2, 2016.
2. R. Bird. *Introduction to Functional Programming.* Pearson Education, 2nd edition, 1998.
3. R. Bird and P. Wadler. *An Introduction to Functional Programming.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1988.
4. J. C. Blanchette, A. Bouzy, A. Lochbihler, A. Popescu, and D. Traytel. Friends with benefits. In *Proceedings of ESOP 2017*, pages 111–140. Springer, 2017.
5. J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel. Truly modular (co)datatypes for Isabelle/HOL. In G. Klein and R. Gamboa, editors, *Proceedings of ITP*, pages 93–110. Springer International Publishing, 2014.
6. J. C. Blanchette, F. Meier, A. Popescu, and D. Traytel. Foundational nonuniform (co)datatypes for higher-order logic. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, June 2017.
7. B. Buchberger. Theory exploration with Theorema. *Analele Universitatii Din Timisoara, ser. Matematica-Informatica*, 38(2):9–32, 2000.
8. K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of ICFP*, pages 268–279, 2000.

9. K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating inductive proofs using theory exploration. In *Proceedings of CADE*, volume 7898 of *LNCS*, pages 392–406. Springer, 2013.
10. P. Fu, E. Komendantskaya, T. Schrijvers, and A. Pond. Proof relevant corecursive resolution. In *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, pages 126–143, 2016.
11. R. Hinze. Concrete stream calculus: An extended study. *J. Funct. Program.*, 20(5-6):463–535, Nov. 2010.
12. G. Hutton and J. Gibbons. The generic approximation lemma. *Information Processing Letters*, 79:2001, 2001.
13. B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.
14. M. Johansson. Automated theory exploration for interactive theorem proving: An introduction to the Hipster system. In *Proceedings of ITP*, volume 10499 of *LNCS*, pages 1–11. Springer, 2017.
15. M. Johansson, L. Dixon, and A. Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 47(3):251–289, Oct 2011.
16. M. Johansson, D. Rosén, N. Smallbone, and K. Claessen. Hipster: Integrating theory exploration in a proof assistant. In *Proceedings of CICM*, pages 108–122. Springer, 2014.
17. R. Leino and M. Moskal. Co-induction simply: Automatic co-inductive proofs in a program verifier. Technical report, Microsoft Research, July 2013.
18. A. Lochbihler. Coinductive. *Archive of Formal Proofs*, Feb. 2010. http://isa-afp.org/entries/Coinductive.html, Formal proof development.
19. D. Lucanu, E.-I. Goriac, G. Caltais, and G. Roşu. CIRC: A behavioral verification tool based on circular coinduction. In *Proceedings of CALCO 2009*, pages 433–442. Springer, 2009.
20. R. L. McCasland, A. Bundy, and P. F. Smith. MATHsAiD: Automated mathematical theory exploration. *Applied Intelligence*, Jun 2017.
21. R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., 1989.
22. O. Montano-Rivas, R. McCasland, L. Dixon, and A. Bundy. Scheme-based theorem discovery and concept invention. *Expert systems with applications*, 39(2):1637–1646, 2012.
23. K. Nakata and T. Uustalu. A Hoare logic for the coinductive trace-based big-step semantics of While. In *Programming Languages and Systems*, pages 488–506. Springer, 2010.
24. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL*. Springer, 2002. Latest online version http://isabelle.in.tum.de/dist/Isabelle2017/doc/tutorial.pdf.
25. L. C. Paulson and J. C. Blanchette. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In *Proceedings of IWIL-2010*, 2010.
26. D. Pous. Coinduction all the way up. In *Proceedings of LICS*, pages 307–316, New York, NY, USA, 2016. ACM.
27. D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA, 2011.
28. N. Smallbone, M. Johansson, K. Claessen, and M. Algehed. Quick specifications for the busy programmer. *Journal of Functional Programming*, 27, 2017.
29. D. A. Turner. Total functional programming. *J. UCS*, 10(7):751–768, 2004.