CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Theory Exploration on Infinite Structures

Master's thesis in Computer Science – Algorithms, Languages, and Logic

SÓLRÚN HALLA EINARSDÓTTIR

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017

Theory Exploration on Infinite Structures

SÓLRÚN HALLA EINARSDÓTTIR

Department of Computer Science and Engineering
Formal Methods Division
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Theory Exploration on Infinite Structures
SÓLRÚN HALLA EINARSDÓTTIR

Supervisors: Moa Johansson, Department of Computer Science and Engineering
Johannes Åman Pohjola, Department of Computer Science and Engineering
Examiner: Wolfgang Ahrendt, Department of Computer Science and Engineering

Master's Thesis 2017
Department of Computer Science and Engineering
Formal Methods Division
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

Theory Exploration on Infinite Structures
SÓLRÚN HALLA EINARSDÓTTIR
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Hipster is a theory exploration system for the interactive theorem prover Isabelle/HOL which has previously been used to discover and prove inductive properties. In this thesis we present our extension to Hipster which adds the capability to discover and prove *coinductive* properties, allowing the exploration of infinite structures that Hipster could not handle before.

We have extended Hipster with a coinductive proof tactic, allowing it to discover and prove coinductive lemmas. As Hipster's theory exploration relies on generating terms and testing their equality, exploring infinite types whose equality cannot be determined presents a challenge. To solve this we have added support for observational equivalence to test the equivalence of infinite terms.

We have evaluated our extension on a number of examples and found that it is capable of proving a variety of coinductive theorems and discovering useful coinductive lemmas. To the best of our knowledge, Hipster is the first theory exploration system to be capable of handling infinite structures and discovering coinductive properties about them.

Keywords: Theory exploration, coinduction, automatic theorem proving, infinite structures, Hipster, Isabelle/HOL, QuickSpec, formal methods, functional programming

Acknowledgements

I would like to thank my supervisor Moa Johansson for suggesting this project and getting it off the ground, my co-supervisor Johannes Åman Pohjola for taking over my supervision and guiding me to the finish line, and my examiner Wolfgang Ahrendt for his patience and understanding in face of my continued procrastination.

I am grateful to those who gave their time to discuss the project with me and help me along when I was stuck, in particular Nick Smallbone and Koen Claessen.

I would like to express my great appreciation to Oskar Abrahamsson for his opposition, advice, and patience. I would also like to thank my officemate Maximilian Algehed.

Finally, I wish to thank my family and friends for their continuous support and encouragement throughout my studies these past two decades, especially my wonderful boyfriend Matthías Páll Gissurarson.

Sólrún Halla Einarsdóttir, Gothenburg, October 2017

Contents

Chapter 1

Introduction

Theory exploration is an automated technique for discovering new interesting mathematical properties for some given set of datatypes and functions. An interactive theorem prover or proof assistant is a software tool that assists users in the formulation of formal proofs, by a collaboration between the human user and automated techniques provided by the software.

Combining these two concepts, Hipster [1] is a theory exploration system for the interactive theorem prover Isabelle/HOL. It takes a set of functions and datatypes as input, automatically discovers equational properties about this input, and proves that the properties hold. These proofs are handled by an automatic proof procedure called a tactic.

The proof tactics previously implemented for Hipster use induction, simplification, and first-order reasoning, so it has only been used for exploration of theories for which those methods are sufficient. Until now, Hipster has mainly been used to discover and prove conjectures about recursive structures and functions, as they require induction to prove. This is useful in reasoning about functional programs as recursion is one of the elementary components of their construction.

However, functional programs may also contain (co)recursively defined "lazy" structures, where some instances of the structure may be infinitely large so that they can not be built from a base instance in a finite number of inductive steps. Examples of such structures are streams (infinite lists) and lazy lists (lists that may or may not be infinite). Induction does not suffice to prove conjectures about such types, but such proofs can be achieved by using a different proof technique, called coinduction [2, 3].

Induction relies on the assumption that the structure in question can be constructed from a base instance in a finite number of steps, and an inductive proof of a property shows that the property holds for all instances that can be constructed in the specified manner. Conversely, coinduction relies on the structure being "destructed" into simpler substructures in a specified way, and a coinductive proof of a property shows how that property holds for all instances that can be destructed in the specified manner.

Coinduction is not only useful to reason about lazy types in functional programs but has been used in various application areas to reason about structures that are cyclic or infinite. It has for instance been used to reason about concurrent processes, prove the soundness of type systems, formalize and reason about invariance properties of programs, and formulate database queries for nonstructured data [2].

We have developed a method of automatically discovering and proving coinductive properties about lazy types, using a notion of observational equivalence to perform theory exploration on infinite structures. We have implemented an extension to Hipster using this method, and our extension's performance and capabilities have been evaluated using a variety of examples.

To the best of our knowledge, Hipster is the first theory exploration system that is capable of handling infinite structures and discovering coinductive properties about them. The work presented in this thesis is a novel contribution to the intersection of theory exploration and functional programming in lazy languages. Coinductive reasoning is used in many application areas, as is mentioned above, and our work towards automating the discovery and proofs of coinductive properties may also prove useful there in the future.

Chapter 2

Background

In this chapter we give a brief description of the Hipster theory exploration system, the Isabelle proof assistant, the coinductive proof method and how coinduction works in Isabelle.

2.1 Hipster

Hipster [1] is a theory exploration system for Isabelle. It aims to automatically discover missing lemmas in a given theory. This facilitates theory development as it expands the collection of lemmas that can be used in automated and interactive proofs within the theory.

Hipster's lemma discovery procedure is parametrized by two proof tactics set by the user, one for routine reasoning and the other for difficult reasoning. Lemmas that can be proven by the routine tactic are assumed to be trivial and of little interest to the user, while those that require the difficult tactic are considered to be more interesting.

2.1.1 Architecture

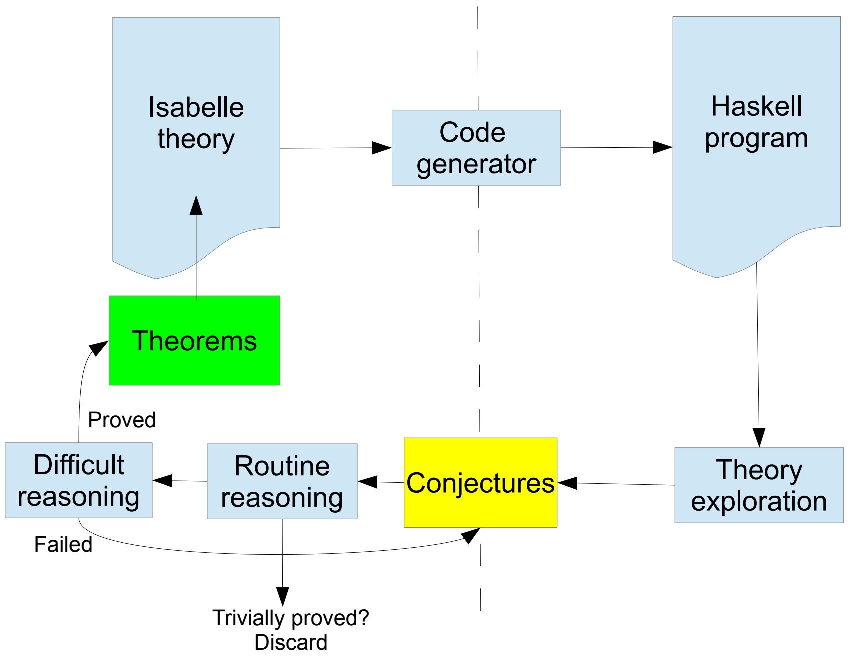
The current implementation of Hipster is broadly as follows [1]:

(i) Starting from an Isabelle/HOL theory, Hipster calls Isabelle/HOL's code generator to translate the given functions into a Haskell program. The Haskell file is translated to the TIP [4] format by the TIP tools `tip-ghc` translator. The TIP format and tools are introduced in section 2.1.3.

(ii) Theory exploration is performed by the QuickSpec system [5], introduced in section 2.1.2 which is called via the TIP tool `tip-spec`.

(iii) The conjectures found in (ii) are imported back to Isabelle, using the translation functionality from TIP tools. Then attempts are made to prove the conjectures, first using the selected routine tactic and then the selected hard tactic if the routine one doesn't suffice. Conjectures that can be proved by the routine tactic are discarded as trivial, while those that require the hard tactic can be copied into the Isabelle theory file as lemmas. Conjectures that cannot be proved by either of the two tactics are presented to the user without a proof, and the user can then attempt to prove them manually.

Figure 2.1: An overview of Hipster's architecture (reprinted from [1])



The Hipster software consists mainly of ML code files, which are imported into a common Isabelle theory file. This theory is then imported into Isabelle theory files where the user wishes to make calls to Hipster. The current implementation can be found on github [1].

2.1.2 QuickSpec

QuickSpec [5] is a theory exploration system that discovers equational properties of Haskell programs. It takes a set of functions as input and generates all type-correct terms up to a given size limit, and then attempts to divide the terms into equivalence classes. Random ground values are found for the variables in the terms using QuickCheck [6] and splits terms that evaluate to different values into different classes. This is repeated until the equivalence classes stabilize.

Then equations are formed from each equivalence class by equating each term in the class to a chosen representative term. These generated conjectures are not necessarily true but likely to be so, having been tested on several hundred different random variables. In the case of Hipster these conjectures are then imported into Isabelle and not presented as lemmas unless they can be proven, so if an untrue conjecture is found the attempt to prove it will simply fail.

2.1.3 TIP

The TIP (Tons of Inductive Problems) suite of benchmarks for inductive theorem provers [4] aims to provide a standard benchmark suite for inductive theorem provers. Its benchmark problems are expressed in the TIP format [7], which is

[1]`https://github.com/moajohansson/IsaHipster/tree/TIP-Isabelle2016`

an extended variant of SMT-LIB [8], the details of which are unimportant for the purposes of this thesis.

The TIP tools are tools for working with the TIP suite. They currently include the `tip-ghc` tool, which translates Haskell files to the TIP format, the `tip` tool, which translates TIP files to various other formats, including Isabelle, and the `tip-spec` tool which performs QuickSpec exploration to invent conjectures about a TIP file. Hipster makes use of the `tip-ghc` tool to translate the theory being explored to the TIP format after it has been translated from Isabelle to Haskell by Isabelle's code generator. It then uses the `tip-spec` tool to perform theory exploration, after which the discovered conjectures are translated to Isabelle by the `tip` tool.

2.2 Isabelle

Isabelle [9] is a generic proof assistant that assists users in the formalization of mathematical proofs. It has a central meta-logical framework, Isabelle/Pure [10], which can be instantiated to a broad range of object-logics, such as set theory or sequent calculi, following the idea of natural deduction [11].

Isabelle/HOL [12], the instance of Isabelle that Hipster is built around, is the most mature and widely used instance of Isabelle. It provides a higher-order logic theorem proving environment and includes powerful specification tools and a large library of built-in theories.

2.2.1 Layers

The regular Isabelle user interacts with the upper layer of Isabelle *theories*, or modules, which contain types, functions and theorems written in a notation similar to conventional mathematical notation. The proofs of theorems are written in the structured proof language Isar [13], (Intelligible Semi-Automated Reasoning), which is designed to read like traditional mathematical proof texts.

Underlying this top layer is the lower layer, which is implemented in Isabelle/ML, a dialect of Standard ML. New proof procedures and extensions can be written in ML and accessed from the top layer. Hipster is mainly implemented in Isabelle/ML but meant to be used when developing Isabelle theories at the top layer.

2.2.2 Syntax notes

The Isabelle/HOL syntax consists of an *outer syntax*, the Isabelle theory language, and an *inner syntax*, the HOL syntax. Types and formulas written in the inner syntax are embedded in the outer syntax; in some cases they must be enclosed in quotation marks.

Shorter arrows $\Rightarrow$ are used to denote function types in the inner syntax, while longer arrows $\Longrightarrow$ represent implication in the outer syntax.

2.2.3 Tactics, goals and proving

Tactics in Isabelle are theorem proving functions written in ML that can be used for automated proving of theorems in an Isabelle theory.

A tactic is a function that transforms a goal to zero or more subgoals, where a goal represents the conjecture that is to be proved based on the currently known facts and assumptions. Further tactics can then be applied to refine these subgoals, and when all the subgoals have been solved a proof of the original theorem can be constructed by going backwards through the steps that have been taken [11].

The following example is taken from section 2.2.2 in [14]. Suppose we have the following definitions of natural numbers and addition:

```
datatype nat = 0 | Suc nat
```

```
fun add :: "nat => nat => nat" where
"add 0 n = n"|
"add (Suc m) n = Suc (add m n)"
```

Now suppose we want to prove that **add** $m\ 0 = m$ (Note that the definition above is right associative so this does not follow trivially from the definition). In Isabelle we can use the `lemma` command to state a conjecture and start a proof of it:

```
lemma add_0_right: "add m 0 = m"
```

At this point the goal of the proof is simply **add** $m\ 0 = m$. Now suppose we apply a tactic for induction over m, which can be done with the command `apply(induction m)` in Isabelle/HOL. Then the goal will be refined to the following subgoals:

1. **add** $0\ 0 = 0$
2. $\forall m.$ **add** $m\ 0 = m \Longrightarrow$ **add** $(Suc\, m)\ 0 = Suc\, m$

These subgoals are simple enough to be solved by simplification based on the definition of addition shown above and the induction hypothesis.

Isabelle/HOL includes various pre-written tactics that users can make use of, including tactics for induction, as shown in the example above, and coinduction, as shown in section 2.4.1. In our coinduction tactic, discussed in Chapter 3, we make use of the built-in coinduction tactic, the simplification tactic *simp*, and the automated proof construction tool Sledgehammer.

The *simp* tactic simplifies the assumptions and conclusion of the current goal. It replaces expressions that occur on the left hand side of the available simplification rules with the corresponding right hand side expressions, continuing with these replacements for as long as possible. The user can add theorems to the pool of available simplification rules, otherwise they are implicitly declared by type and function definitions.

Sledgehammer [15] is invoked from within Isabelle/HOL and calls on several external automatic theorem provers (ATPs) that run for up to 30 seconds searching for a proof. It automatically makes use of relevant lemmas from all those available, and if it is successful in finding a proof, it tries to generate a proof command that can be inserted into the theory.

2.3 Coinduction

Consider lists with elements of type `a`, defined by:
`List a = Nil | Cons a (List a)`
We can reason about such lists using structural induction. For instance, if we want to prove that two list functions `F` and `G` are equivalent, that is, `F xs = G xs` for all lists `xs` of type `List a`, we do the following:

1. Prove that
   `F Nil = G Nil`
2. Prove that if `F ys = G ys` for some list `ys`, then
   `F (Cons y ys) = G (Cons y ys)` for any `y` of type `a`.

We can then prove, by structural induction, that `F` and `G` are equivalent.

Induction relies on the assumption that every instance of this list type can be constructed from the empty list `Nil` in a finite number of steps, by using the `Cons` constructor to add elements. But what if the `Cons` is applied an infinite number of times? Such a list is no longer reachable from `Nil` and the inductive reasoning is no longer valid.

We must use another approach to prove properties of such potentially infinite lists. Instead of the inductive approach, which is based on each data element being formed recursively by “constructor” operations (in this case `Nil` and `Cons`), we can use a coinductive approach, which uses so-called “destructor” operations that tell us what can be observed about data elements. An example of such destructors for lists are the `head` and `tail` operations. We can then prove by coinduction that an equation holds for all lists, finite or infinite, by showing that it is satisfied for a list with any possible `head` and `tail` values.

We can prove that two list functions `F` and `G` are equivalent using coinduction in the following manner:

1. Prove that `F xs = Nil <=> G xs = Nil`
2. Otherwise, prove that `head (F xs) = head (G xs)`
   and there exists a list `ys` of type `List a` such that
   `tail (F xs) = F ys` and `tail (G xs) = G ys`

An example of such a proof is shown in section 2.4.1, where we prove coinductively that `lappend xs Nil = xs`

Coinduction is the mathematical dual of structural induction, relying on deconstructing structures from the top down instead of constructing them from the bottom up as induction does. Interested readers can find a more detailed introduction to coinduction in [2] or [3].

2.4 Codatatypes and coinduction in Isabelle/HOL

Isabelle/HOL has separate definitional commands for datatypes and codatatypes [16]. The keyword `datatype` implies that the defined type only contains finite values, while the keyword `codatatype` implies that the type contains infinite values as well as finite ones. There is built-in support for reasoning about codatatypes using corecursion and coinduction.

Isabelle/HOL generates a number of characteristic theorems for every type that is defined using one of these keywords. These theorems describe various properties that hold for the type in question, which can then be used in reasoning involving that type. Both datatypes and codatatypes generate similarly defined free constructor theorems and functorial theorems, but while datatypes generate *inductive* theorems which state properties relating to their inductive nature, codatatypes generate *coinductive* theorems which state properties relating to their coinductive nature. A further overview of these characteristic theorems can be found in [17].

2.4.1 Isabelle/HOL's coinduction tactic

We will present Isabelle/HOL's built-in coinduction tactic by showing an example of its use. Consider the following Isabelle/HOL definition of a lazy list:

```
codatatype (lset:'a) Llist =
  lnull: LNil
  | LCons (lhd:'a) (ltl: "'a Llist ")
where
 "ltl LNil = LNil"
```

The theorem `Llist.coinduct` is automatically generated based on the above definition and has the following form:

$$\begin{aligned}
&R\ ls\ ls' \Longrightarrow \\
&(\forall\ l1\ l2. \\
&\ R\ l1\ l2 \Longrightarrow lnull\ l1 = lnull\ l2 \\
&\qquad\quad \wedge (\neg\, lnull\ l1 \rightarrow \neg\, lnull\ l2 \rightarrow lhd\ l1 = lhd\ l2 \\
&\qquad\quad \wedge\ R\ (ltl\ l1)\ (ltl\ l2))) \\
&\qquad\quad \Longrightarrow ls = ls'
\end{aligned}$$

Here R is an unknown witness relation and ls and ls' are unknown lazy lists. The theorem above states that if R holds for ls and ls', and if for an arbitrary but fixed pair $l1$ and $l2$ we have that if R holds for $l1$ and $l2$ then:

1. If $l1$ is empty then so is $l2$, and vice versa.
2. If neither $l1$ nor $l2$ is empty then their respective heads are equal and R holds for their respective tails.

then ls and ls' must be equal.

Note that this rule states that two lazy lists must be equal to each other if they satisfy certain requirements. We can therefore use it to prove the equality of two lazy lists by coinduction, namely by proving that the lists satisfy those requirements.

We define the function `lappend` to append one lazy list to another:

```
primcorec lappend :: "'a Llist ⇒ 'a Llist ⇒ 'a Llist" where

"lnull xs ⟹ lnull ys ⟹ lnull (lappend xs ys)"
| "lhd (lappend xs ys) = lhd (if lnull xs then ys else xs)"

| "ltl (lappend xs ys) = (if lnull xs then ltl ys else lappend (ltl xs) ys)"
```

We would like to prove that appending an empty list to the lazy list `xs` results in `xs` itself, stated in the following lemma:

```
lemma lappend_LNil: "lappend xs LNil = xs"
```

The starting goal of the proof state is then simply

$$lappend\ xs\ LNil = xs.$$

We then apply Isabelle/HOL's *coinduction* tactic by writing:

```
apply(coinduction arbitrary: xs)
```

We set xs as arbitrary which causes xs to be universally quantified in the coinduction hypothesis, so the hypothesis is stated to hold for any arbitrary (but fixed) value of xs, rather than for a known value of xs. The *coinduction* tactic has an optional parameter to specify which variables should be set as arbitrary. The *coinduction* tactic has another optional parameter to specify the coinduction rule on which the proof should be based. If no rule is specified it uses the generated coinduction rule `t.coinduct` for some codatatype t that is used in the statement of the goal.

After this application the goal of the proof state has become the following:

$$\begin{aligned}\forall xs.\ & lnull\ (lappend\ xs\ LNil) = lnull\ xs\\ & \wedge (\neg\, lnull\ (lappend\ xs\ LNil) \rightarrow \neg\, lnull\ xs\\ & \rightarrow\ lhd\ (lappend\ xs\ LNil) = lhd\ xs\\ & \wedge (\exists xsa.\ ltl\ (lappend\ xs\ LNil) = lappend\ xsa\ LNil \wedge ltl\ xs = xsa))\end{aligned}$$

If we look back at the coinduction theorem `Llist.coinduct` shown on the previous page, we see that this goal corresponds to the requirements which that rule says are sufficient to prove that two lazy lists are equal. Here the witness relation R could be defined as

$$R(xs, ys) = (xs = lappend\ ys\ LNil)$$

This goal is easy enough to be proved by simplification based on the definition of `lappend`. After a call to *simp* the goal of the proof state is empty, meaning the lemma has been fully proved.

Chapter 3

A tactic for coinduction

In this chapter we describe an automated tactic for coinductive proofs and the implementation of such a tactic which we have added to Hipster.

As is described in section 2.4.1, Isabelle/HOL already contains a coinduction tactic that performs coinduction when applied by the user. After applying this tactic the user must decide how to proceed to complete the proof after the coinductive step.

The ability to automatically prove lemmas in one step without user involvement is crucial in lemma discovery by theory exploration, as then the process of attempting to prove the discovered conjectures, and adding those that have appropriately difficult proofs to the theory being explored, can be automated. We would therefore like to extend Hipster with an automated tactic for proving coinductive lemmas. This tactic must perform coinduction automatically and then complete the proof by proving all remaining subgoals. 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.

3.1 Automatically determining parameters

As shown in section 2.4.1, Isabelle/HOL's coinduction tactic has parameters to set which variables are arbitrary, meaning that they are universally quantified in the goal statement. It also has an optional parameter to specify which coinduction rule to apply.

3.1.1 Arbitrary variables

In many cases, such as the proof shown in section 2.4.1, universally quantifying the correct variables in the coinduction hypothesis can make it possible to prove a formerly unprovable goal. In order to automatically prove our goal we must automatically determine which variables should be universally quantified. We can solve this by simply finding all free variables in the current proof state, which is fairly simple to do with the help of built-in functions in Isabelle/HOL, and declaring them all as arbitrary. This makes the proof goal statement more complicated and verbose than may be necessary, but less or equally difficult to prove than the same goal with less of the variables universally quantified. A goal that can be proved to based on a hypothesis that holds for some fixed value of variable v must also be provable when the hypothesis holds for all possible values of v.

3.1.2 Choice of coinduction rule

The built-in *coinduction* tactic also has an optional parameter to specify what coinduction rule should be used for the proof. We also let the user specify what rule they would like to use, as an optional parameter to our tactic which is then passed along as a parameter to the *coinduction* tactic.

If the user does not specify what rule to use, we would like to have a default rule that can be applied prove as wide a range of lemmas as possible. As was discussed in section 2.4, every codatatype `t` has a characteristic coinduction rule `t.coinduct`. In addition to this, every codatatype `t` also has a characteristic strong coinduction rule, `t.coinduct_strong`.

For example, here is the theorem `Llist.coinduct_strong` for the lazy list type `Llist` defined in in section 2.4.1:

$$\begin{aligned}
&R\ ls\ ls' \Longrightarrow\\
&(\forall\ l1\ l2.\\
&\ R\ l1\ l2 \Longrightarrow lnull\ l1 = lnull\ l2\\
&\qquad\qquad \wedge (\neg\, lnull\ l1 \rightarrow \neg\, lnull\ l2 \rightarrow lhd\ l1 = lhd\ l2\\
&\qquad\qquad \wedge\ (R\ (ltl\ l1)\ (ltl\ l2)\\
&\qquad\qquad \vee\ ltl\ l1 = ltl\ l2')))\\
&\qquad\qquad \Longrightarrow ls =\ ls'
\end{aligned}$$

If we compare this to the theorem `Llist.coinduct`, which is shown in section 2.4.1, we see that they are the same except for the second-to last line, $\vee\, ltl\ l1 = ltl\ l2'$, which is not included in `Llist.coinduct`. Since using `Llist.coinduct` for a proof requires a proof of

$$R\ (ltl\ l1)\ (ltl\ l2)$$

while *llist.coinduct_strong* requires a proof of

$$(R\ (ltl\ l1)\ (ltl\ l2) \vee\ ltl\ l1 = ltl\ l2'),$$

we see that `Llist.coinduct_strong` is more general than `Llist.coinduct`.

Some theorems that cannot be proved with the default `t.coinduct` rule can be proved by instead using the stronger rule `t.coinduct_strong`, for instance strong coinduction is needed to prove the associativity of the lazy list append lapp,

$$lapp\ (lapp\ xs\ ys)\ zs = lapp\ xs\ (lapp\ ys\ zs)$$

Since *t.coinduct_strong* is more general than `t.coinduct`, but includes the requirements from `t.coinduct` in disjunction with others, `t.coinduct_strong` can be used to prove any lemma where `t.coinduct` is sufficient. Therefore, we choose the rule `t.coinduct_strong`, for the first codatatype `t` found in the proof context, as our default coinduction rule. In an equational conjecture, such as those discovered by QuickSpec, the first codatatype in the proof context is the outermost codatatype in the equation. This is the type of the terms we want to prove are equal and is therefore the appropriate type to perform coinduction over.

3.2 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*, as described in section 2.2.3. If this does not suffice it uses Isabelle's automated proof construction tool Sledgehammer, also described in section 2.2.3, 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.

3.3 Outcome

We have designed and implemented an automated coinductive proof tactic for equational lemmas in Isabelle/HOL. Our tactic performs coinduction, automatically determining the appropriate parameters, and then continues to attempt to prove the remaining subgoals of the conjecture in question. Our tactic can be used to prove a variety of coinductive lemmas, as is further demonstrated and discussed in Chapter 5. We can now, for instance, prove the lemma from section 2.4.1 automatically with a call to our tactic.

Chapter 4

Theory exploration

In this chapter we describe how we can use the notion of observational equivalence, along with the take lemma, to perform theory exploration on codatatypes with no finite instances. We then show how a user can explore an Isabelle theory using Hipster, discovering lemmas that are proved using the tactic described in Chapter 3.

4.1 Testing infinite structures

As is described in chapter 2.1, Hipster calls on QuickSpec, via `tip-spec`, to come up with conjectures using QuickCheck-based testing. Since QuickSpec can only check the equality of finite terms, we cannot test truly infinite structures to generate conjectures, only finite instances of the types being considered.

When the codatatype being considered has no finite instances, as in the case of streams, QuickSpec cannot check the equality of any of the generated terms, since that would take an infinite amount of time due to their infinite size. When we first attempted to use Hipster for theory exploration on streams it timed out and resulted in an error.

Since we cannot directly test the equality of streams by comparing them element for element as we can do for finite lists, we must use some other way to determine whether two streams are equal. The take lemma [18] states that two streams xs and ys are equal if $take_n(xs) = take_n(ys)$ for all natural numbers n, where $take_n$ is a function that returns a list containing the n first elements of a stream. We make use of this idea to come up with conjectures about the equality of streams, using observation functions as described in section 4.1.1.

4.1.1 Observational equivalence in QuickSpec

QuickSpec has support for observational equivalences to deal with types, such as those that have no finite instances, that cannot be directly compared [5]. The user can define a method of observing such a type and state that two values of the type are equivalent if all such observations make them equal.

More specifically: For any type T the user can supply an observation 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 observation function using the random *Obs* and comparing the resulting values of type *Res*.

For instance, we can define an observation 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 observation 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.

4.1.2 Observational equivalence in tip-spec and Hipster

Support for observational equivalence had not previously been added to `tip-spec`, so we implemented changes to the `tip-spec` code base to allow Hipster to make use of observation functions. We added optional parameters to `tip-spec` calls representing the type, T, that needs an observation function, the observable type, *Res*, we use to represent it, and an observation function with type $Int \to T \to Res$. We use integers in place of the *Obs* type from section 4.1.1, since when observing an infinite structure we can use a positive integer to determine how large of a finite substructure to observe. Looking this function and types up in the scope of the code being explored, `tip-spec` can now define an observation function accepted by QuickSpec and pass that function as a parameter to its QuickSpec call.

We have also added support for observation functions in Hipster. The user can now define an observation function in their Isabelle theory file. The names of this function, the type that needs an observation function and the type that can be used to observe it are then specified by the user in their Hipster call. This information is passed along to `tip-spec` and QuickSpec and handled there as is described above. An example of an observation function definition and corresponding Hipster call is shown in section 4.2.1

4.2 Exploring a theory

To perform theory exploration without observer functions, the user calls the command `hipster` followed by the names of the functions the user wishes to discover properties about. When several function names are given they are explored jointly which can lead to the discovery of properties that describe relations between different functions in addition to properties of individual functions.

4.2.1 Examples

For instance, suppose we have the lazy list type `Llist` as seen in section 2.4, along the append function `lappend`, also shown in section 2.4 and the map function `lmap` as shown below:

```
primcorec lmap :: "('a => 'b) => 'a Llist => 'b Llist" where
 "lmap f xs = (case xs of LNil => LNil
             | LCons x xs => LCons (f x) (lmap f xs))"
```

We can then invoke theory exploration on these functions with Hipster by typing

```
hipster lappend lmap
```

This produces a set of lemmas involving either or both of these functions:

```
lemma lemma_a [thy_expl]: "lmap y (LCons z LNil) = LCons (y z) LNil"
  apply (coinduction  arbitrary: y z
         rule: ExploreExample.Llist.coinduct_strong)
  by simp

lemma lemma_aa [thy_expl]: "LCons (y z) (lmap y x) = lmap y (LCons z x)"
  apply (coinduction  arbitrary: x y z
         rule: ExploreExample.Llist.coinduct_strong)
  by simp

lemma lemma_ab [thy_expl]: "lappend y LNil = y"
  apply (coinduction  arbitrary: y
         rule: ExploreExample.Llist.coinduct_strong)
  by simp

lemma lemma_ac [thy_expl]: "lappend LNil y = y"
  apply (coinduction  arbitrary: y
         rule: ExploreExample.Llist.coinduct_strong)
  by simp

lemma lemma_ad [thy_expl]: "ltl (lappend y y) = lappend (ltl y) y"
  apply (coinduction  arbitrary: y
         rule: ExploreExample.Llist.coinduct_strong)
  apply simp
  by (smt Llist.collapse(1) Llist.sel(2) lappend.disc_iff(2)
      lappend.simps(3) lappend.simps(4))

lemma lemma_ae [thy_expl]:
  "lappend (LCons y z) x2 = LCons y (lappend z x2)"
  apply (coinduction  arbitrary: x2 y z
         rule: ExploreExample.Llist.coinduct_strong)
  by simp

lemma lemma_af [thy_expl]: "lappend (lappend y z) x = lappend y (lappend z x)"
  apply (coinduction  arbitrary: x y z
         rule: ExploreExample.Llist.coinduct_strong)
  apply simp
  by auto
```

```
lemma lemma_ag [thy_expl]: "ltl (lappend y (ltl y)) = lappend (ltl y) (ltl y)"
  apply (coinduction  arbitrary: y
         rule: ExploreExample.Llist.coinduct_strong)
  apply simp
  by (metis ExploreExample.lemma_ab Llist.collapse(1) Llist.sel(2))

lemma lemma_ah [thy_expl]: "ltl (lmap y z) = lmap y (ltl z)"
  apply (coinduction  arbitrary: y z
         rule: ExploreExample.Llist.coinduct_strong)
  apply simp
  by (smt Llist.case_eq_if Llist.disc(1) Llist.disc_eq_case(1) Llist.sel(1)
      Llist.simps(5) lmap.code ltl_def)

lemma lemma_ai [thy_expl]:
  "ltl (lappend z (lmap y z)) = lappend (ltl z) (lmap y z)"
  apply (coinduction  arbitrary: y z
         rule: ExploreExample.Llist.coinduct_strong)
  apply simp
by (smt ExploreExample.lemma_ab lappend.disc_iff(2) lappend.simps(3)
    lappend.simps(4) lmap.ctr(1) lmap.disc(2))

lemma lemma_aj [thy_expl]:
  "lappend (lmap y z) (lmap y x2) = lmap y (lappend z x2)"
  apply (coinduction  arbitrary: x2 y z
         rule: ExploreExample.Llist.coinduct_strong)
  apply simp
  by (smt Llist.case_eq_if lappend.disc_iff(2) lappend.simps(3)
      lappend.simps(4))

lemma lemma_ak [thy_expl]:
"ltl (lappend (lmap y z) (ltl z)) = lappend (lmap y (ltl z)) (ltl z)"
apply (coinduction  arbitrary: y z rule: ExploreExample.Llist.coinduct_strong)
apply simp
by (smt ExploreExample.lemma_aa ExploreExample.lemma_ab Llist.case_eq_if
    Llist.collapse(2) Llist.sel(1) Llist.sel(2) lemma_ah lmap.ctr(1)
    lmap.disc(2) lnull_def)

lemma unknown [thy_expl]:
"ltl (lappend (lmap y z) z) = lappend (lmap y (ltl z)) z"
oops
```

The construction of the proofs to these lemmas is based on the coinduction tactic described in Chapter 3 and is further described in section 4.2.2.

If an observer function is to be used to explore the command `hipster_obs` is used instead, followed by, in order:

1. The name of the type, T, requiring an observer function.
2. The name of the type, Res, that can be used to observed.
3. The name of the observer function with type $Int \to T \to Res$.
4. The names of the functions the user wishes to explore.

For instance, suppose we have the following definition for streams:

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

Along with the following definition of a list and an observer function that extracts a list of a given length from the end of a stream.

```
datatype 'a Lst =
  Emp
  | Cons "'a" "'a Lst"

fun obsStream :: "int => 'a Stream => 'a Lst" where
"obsStream n s = (if (n <= 0) then Emp
                  else Cons (shd s) (obsStream (n - 1) (stl s)))"
```

Suppose we want to discover lemmas involving the following map and iterate functions:

```
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
  "shd (siterate f x) = x"
| "stl (siterate f x) = siterate f (f x)"
```

We can then invoke Hipster with the `hipster_obs` command as follows, note the order of arguments which is in accordance with the list above.

```
hipster_obs Stream Lst obsStream smap siterate
```

Hipster then provides the following lemmas:

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

lemma lemma_am [thy_expl]:
"smap z (SCons y (siterate z x2)) = SCons (z y) (siterate z (z x2))"
  apply (coinduction  arbitrary: x2 y z
         rule: ExploreExample.Stream.coinduct_strong)
  by (simp_all add: lemma_al)
```

4.2.2 Proof loop

During the theory exploration for the examples in the previous section, we had the routine tactic set to use Isabelle/HOL's simplifier followed by first-order reasoning by Metis [19], while the hard tactic was the coinduction tactic with Sledgehammer for subgoals as described in Chapter 3.

Those of the discovered conjectures that could be proven by the routine tactic were discarded as trivial and therefore uninteresting to the user. Examples of such conjectures for the examples in section 4.2.1 are

`lappend y LNil = LNil`
which follows trivially from the definition of `lappend`, and
`shd (smap z (siterate y x2)) = z x2`
which follows trivially from `lemma_al`.

Previously discovered lemmas are made available to the tactics used in the proof attempts of following conjectures by the use of the label `thy_expl`. For instance, note that the proof of `lemma_ae` makes use of `lemma_ad`. Hipster attempts to prove each of the conjectures found by QuickSpec using first the routine tactic and then the hard one. QuickSpec attempts to order the conjectures it outputs such that the most general ones are listed first with more specific instances listed later [5]. This way, if a more general conjecture is provable without considering more specific instances, we avoid performing redundant proofs and presenting the user with redundant properties.

Further attempts are made to prove conjectures that could not be proven on the first attempt once attempts have been made to prove all the others, in case a conjecture can be proved by using the proof of another. Hipster cycles through making proof attempts for all unproved conjectures until no new proofs are found. At the end of this process Hipster outputs the found lemmas and their proofs for the user to paste into their theory. Lemmas that could not be proven are also output so that the user can inspect them and make further proof attempts, an example of this is the last listed lemma, `unknown`, from the exploration of `lappend` and `lmap` in section 4.2.1.

Chapter 5

Evaluation

We have gathered a collection of codatatypes along with corecursive functions and coinductive theorems to evaluate our tactic as well as our theory exploration method.

5.1 Evaluation data

Most of our evaluation problems were found in the Coinductive library [20] in the archive of formal proofs [1]. We have chosen lemmas from the `Coinductive_Stream` and `Coinductive_List` theories that are equational and whose proofs use coinduction. These theories contain various definitions and proofs concerning streams and lazy lists respectively. We have also considered some theorems from examples shown in the Certified Functional (Co)programming with Isabelle/HOL Tutorial [2] which was colocated with CADE-26 in August 2017. These theorems described some properties of addition on extended natural numbers, pointwise addition and multiplication on streams of naturals, and transforming infinite trees to lazy lists and vice versa.

5.2 Evaluation procedure

For each of the theorems we follow the following procedure:

1. We make up a file containing definitions of the types and functions involved. We try to avoid using types and functions from theories in Isabelle/HOL (such as the `Main` theory) and instead use our own definitions. In this way Isabelle's predefined properties do not affect our outcome and we can better see Hipster's capacity in dealing with newly defined theories. For instance, we have defined our own map functions instead of using Isabelle's built-in method for declaring a map function after a type definition, as that method automatically provides certain properties.
2. We put forth the given theorem and attempt to prove it using our automated tactic as described in Chapter 3.

[1] `https://www.isa-afp.org/`

[2] `http://matryoshka.gforge.inria.fr/cade26-tutorial/`

3. We attempt to perform exploration with Hipster on all the functions involved in the theorem statement to see if the theorem is found during exploration.
4. If the exploration in the previous step is successful but the desired theorem is not found we again attempt to prove it, to see if it can now be proved using a lemma found during exploration.
5. If the exploration in step 3 fails for some reason and we still have not proved the theorem, we perform exploration on smaller subsets of the involved functions to check if that produces lemmas sufficient to prove the theorem.

5.3 Results

The results of our evaluation are shown below. CoList denotes our collection of theorems from the `Coinductive_List` theory, CoStream the theorems from the `Coinductive_Stream` theory, and CoTut the theorems from the (Co)programming tutorial. Further information on which theorems gave rise to which results can be found in Appendix A.

	CoList	CoStream	CoTut	Total
Total # of theorems	18	16	15	49
Directly provable	12	13	5	30
Provable after exploration	4	0	4	8
Discovered in exploration	6	0	3	9
Fully explorable	11	4	11	25
Requires observer	3	16	9	28

Table 5.1: Hipster's performance on the evaluation problems. The meanings of labels are explained below.

Directly provable A theorem is directly provable by our automated tactic if it can be proved right away without any lemmas having been found first.

Provable after exploration A theorem is provable after exploration if it was not directly provable but is provable by our automated tactic after lemmas have been discovered by theory exploration hit Hipster.

Discovered in exploration Some of the theorems we considered were discovered as lemmas when we performed theory exploration on the functions involved.

Fully explorable A theorem is fully explorable if we can run simultaneous theory exploration on all of the functions involved without running into any issues. Some of the theorems we considered were not fully explorable, for reasons discussed in section 5.4.

Requires observer A theorem requires an observer to perform exploration if one of the functions involved always returns an infinite structure. For instance, any function that returns a stream requires an observer for exploration.

5.3.1 Discussion of results

Out of the 49 theorems we considered, 30 could be proven directly with our automated coinduction tactic, and 7 of the 19 that couldn't be proven directly could be proved using lemmas discovered in theory exploration. We therefore managed to prove 38 out of the 49 theorems. Among those 11 theorems that we still could not prove after exploration there were 7 which were not fully explorable, which may have prevented us from discovering lemmas that could aid us in proving the theorems.

Out of the remaining four theorems, two are taken from the `Coinductive_List` theory from [20] and state that `take` and `takeWhile` distribute over `map` on lazy lists. The original theory file also contains various non-coinductive lemmas that the proofs of the coinductive lemmas may depend on, which are missing from our theory. The proofs of the two theorems in question in the `Coinductive_List` theory both depend on such non-coinductive lemmas.

The other two are from the (Co)programming tutorial, and state that converting a lazy list to a right- or left-leaning tree and then converting this tree back to a lazy list returns the original list. In the proofs shown for these theorems in the tutorial, one uses a non-coinductive lemma in its proof and the other uses a lemma with a conditional formulation, both of which are beyond our current ability to discover with exploration.

5.4 Limitations

During the evaluation some limitations of our method and its implementation were exposed, as is described in this section.

5.4.1 Limitations of our proof tactic

When performing theory exploration as part of our evaluation, we had, as in section 4.2.2, the routine tactic set to use simplification and Metis, while the hard tactic was our coinduction tactic as described in Chapter 3.

When a discovered conjecture cannot be proved using the routine tactic, our method then attempts to prove it using coinduction. However, if there is no coinduction rule corresponding to the type being considered, the proof loop will crash.

This means that if we attempt to run theory exploration on a non-corecursive function using this method, it will probably crash during the proof loop. However, some of the theorems we considered in our evaluation involved a combination of corecursive and non-corecursive functions, so we could not perform theory exploration on all of the functions involved in those cases. Possible solutions to this are discussed in section 6.2.

5.4.2 Limitations of observer method

Currently, our support for observational equivalence, as discussed in section 4.1, is limited in that it only allows the user to specify one observer function to observe one type in each call to Hipster. However, in some cases we may be considering more than one type that requires an observer.

For instance, we attempted exploration on functions that connect streams and lazy lists,

```
primcorec llist_of :: "'a Stream => 'a Llist"
where "llist_of s = LCons (shd s) (llist_of (stl s))"
primcorec stream_of :: "'a Llist => 'a Stream"
where "stream_of xs = SCons (lhd xs) (stream_of (ltl xs))"
```

In this case, although there are finite instances of lazy lists, the function `llist_of` will always return an infinite list when applied to a stream. Therefore we would need an observer for lazy lists in addition to one for streams in order to perform joint exploration on the functions and perhaps discover lemmas such as

```
stream_of (llist_of s) = s
```

Chapter 6

Conclusion

6.1 Related Work

6.1.1 (Co)programming in Isabelle/HOL

There is substantial recent work on making Isabelle/HOL more expressive for working with codatatypes and corecursive functions. In 2014, Blanchette et al. [16] introduced a new package for defining types in Isabelle/HOL, including the `codatatype` command and the `primcorec` command for defining primitively corecursive functions. In [21] they present the notion of *friends*, which allow users to define more kinds of corecursive functions with greater ease than was previously possible.

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. The recent developments in this area also suggest directions for future extensions and improvements of our work, as discussed in section 6.2.

6.1.2 Automated coinductive proofs

There has been prior work on automating coinductive proofs and reasoning, although the techniques are nowhere near as sophisticated or widely used as those available for induction.

In [22] Leino and Moskal present a method for automated reasoning about coinductive properties in the Dafny verifier. CIRC [23] 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.

The CoVeCe (Coinduction for Verification and Certification) project [1] is currently working towards using coinduction to improve upon the state of the art in verification and certification. In [24] Pous presents a new theory encompassing parameterized coinduction [25], which he claims is well suited for automated proofs, and second-order reasoning.

We have not made a proper comparison of our automated coinductive tactic with those available in other systems, but this could make for interesting future work.

[1] `https://perso.ens-lyon.fr/damien.pous/covece/`

However, none of the other systems has the theory exploration capabilities that are the focus of our work.

6.1.3 Other theory exploration systems

IsaCoSy [26] and IsaScheme [27] are other theory exploration systems for Isabelle/HOL, both of which focus on the discovery and proof of inductive properties. MATHsAiD [28] 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.

None of these tools is capable of discovering and proving coinductive properties, and to the best of our knowledge Hipster is the first theory exploration system to have such capabilities. Hipster is well suited to coinductive lemma discovery because it uses QuickSpec for theory exploration. Since Quickspec is implemented in Haskell, a lazy language, and has support for observational equivalence, it allows the generation and checking of infinite structures as test data, making it possible to discover properties for such structures.

6.2 Discussion and future work

While developing and evaluating our extension to Hipster we ran into various issues worth discussing, which also provide interesting ideas for possible directions of future work.

6.2.1 Defining corecursive functions in Isabelle

When gathering the examples used for evaluation in Chapter 5 our selection was mostly limited to primitively corecursive functions, which can be defined using the `primcorec` keyword in Isabelle/HOL. A primitive corecursive function is limited in that its definition must have exactly one constructor around a corecursive call. Our reasons for limiting ourselves to such functions were mainly the simplicity and ease of defining functions using the `primcorec` keyword and the fact that our examples from the Coinductive library were all primitively corecursive.

More complex corecursive functions can be defined using the methods introduced in [21], and it would be interesting to see how Hipster handles such functions. More evaluation on examples of examples that are not primitively corecursive would give us a better idea of Hipster's effectiveness in coinductive lemma discovery, and likely expose flaws that we could work towards eliminating.

6.2.2 Combining inductive and coinductive tactics

One limitation of our implementation is its lack of ability to handle functions that aren't corecursive, as is discussed in 5.4.1. The ability to explore structures that mix inductive and coinductive types and discover properties that relate recursive and corecursive functions would greatly improve Hipster's versatility and applicability.

In some cases, inductive lemmas are required to prove coinductive properties. As a simple example, when exploring streams of natural numbers, we must have a proof of the associativity of addition on the naturals in order to prove the associativity of element-wise addition of two streams. Currently, Hipster's proof loop will crash if it is set to use a coinductive proof tactic and it can't find a coinduction rule corresponding to the types being considered. Our attempts at exploration involving zipping of both lazy lists and streams ran into problems due to the underlying type of pairs, which is not coinductive.

As Hipster's previous development has been focused on induction it already contains effective inductive tactics for exploration and proofs, which could be combined with our coinductive tactic to create a more powerful combined tactic. For this we would need to automatically determine whether induction or coinduction is appropriate in each case by checking whether the types involved are codatatypes or finite datatypes.

6.2.3 Improving observation method

Another limitation we have noticed in our implementation is the fact that we currently only allow one observer function, to observe one type, in each call to Hipster, as is discussed in section 5.4.2. Since we may need more than one observer to explore certain theories, the ability to send more than one observer to hipster would be beneficial. Support for multiple observers already exists in QuickSpec and `tip-spec`. Extending Hipster to do so as well would mainly entail making modifications to how the parameters denoting the observer function and relevant types are passed from Hipster to `tip-spec`.

Another extension that would make Hipster more user friendly for coinductive exploration is automatically generating appropriate observer functions and types where needed rather than having the user define them and including them in their `hipster_obs` call. One solution to this is the automation of the following general technique to define an observer function and observer type, an example of which is the `obsStream` observer function for streams as shown in section 4.2.1.

1. We determine that a codatatype T requires an observer if one of the two following properties is true:
   - T has no nullary constructor in its definition.
   - T is the return type of a corecursive function with no base case.
2. If a codatatype, T, needs an observer, we begin by defining an observer datatype, *Res* that has constructors corresponding to T's constructors, along with an additional nullary constructor if T does not have one.
3. We then define an observation function *obsFun* with type $Int \to T \to Res$ which uses the integer parameter as fuel. *obsFun* $n\ x$ then returns the nullary constructor of *Obs* if $n \leq 0$, and otherwise recursively constructs a structure of type *Obs* using the destructors of T, decrementing the integer parameter in the recursive call. For instance, consider the function `obsStream` as shown in section 4.2.1:

```
fun obsStream :: "int => 'a Stream => 'a Lst" where
"obsStream n s = (if (n <= 0) then Emp
                    else Cons (shd s) (obsStream (n - 1) (stl s)))"
```

Here, `obsStream` returns the empty list `Emp` if it runs out of fuel, and otherwise recursively constructs a list using the destructors `shd` and `stl`.

As can be seen, this technique is fairly generic, which leads us to believe it is possible to automate. A worthwhile step towards such automation could be to alert the user when an observer is needed. In the current implementation a missing observer will simply cause Hipster to time out and display a confusing error message. A simple first step might be to catch timeout errors and inform the user that a missing observer may be the cause of the timeout, along with displaying relevant information about how to use observers. A more advanced solution would check the user's code for codatatypes with no nullary constructors and warn the user that this type requires an observer. A codatatype with a nullary constructor may also need an observer if it is the output type of a corecursive function with no base case, so ideally we would also identify such functions and warn the user that observers are needed to explore them.

6.3 Final remarks

In this thesis we introduced our extension to Hipster, an interactive theory exploration system for Isabelle/HOL, which gives it new capabilities of discovering properties of infinite structures and proving them using coinduction. To the best of our knowledge, Hipster is the first theory exploration system to have such capabilites.

Our main contributions were firstly extending Hipster with a coinductive proof tactic, allowing it to discover and prove coinductive lemmas, and secondly extending Hipster and the underlying tool `tip-spec` to support observation functions to allow for exploration of types that have no finite instances. The use of such observation functions allows us much more versatility in the exploration of infinite structures.

We have evaluated our work on a number of examples and seen that our proof tactic is powerful in its ability to prove a variety of theorems, and that our theory exploration method can discover useful and interesting lemmas. Our results confirm that Hipster is well suited for theory exploration on infinite structures. Its lazy Haskell back-end with support for observational equivalence facilitates the discovery of properties of such structures, while the support for reasoning about such structures in its Isabelle/HOL front-end facilitates the proof of those properties.

Bibliography

[1] M. Johansson, D. Rosén, N. Smallbone, and K. Claessen, "Hipster: Integrating theory exploration in a proof assistant," in *Proceedings of the Conference on Intelligent Computer Mathematics (CICM) 2014*, pp. 108–122, Springer, 2014.

[2] D. Sangiorgi, *Introduction to Bisimulation and Coinduction.* New York, NY, USA: Cambridge University Press, 2011.

[3] B. Jacobs and J. Rutten, "A tutorial on (co)algebras and (co)induction," *EATCS Bulletin*, vol. 62, pp. 222–259, 1997.

[4] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone, "Tip: Tons of inductive problems," in *Proceedings of the Conference on Intelligent Computer Mathematics (CICM) 2015*, pp. 333–337, Springer, 2015.

[5] N. Smallbone, M. Johansson, K. Claesson, and M. Algehed, "Quick specifications for the busy programmer," *Journal of Functional Programming*, vol. 27, 2017.

[6] K. Claesson and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," in *Proceedings of ICFP*, pp. 268–279, 2000.

[7] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone, "The tip format." `http://tip-org.github.io/format.html`.

[8] C. B. an Aaron Stump and C. Tinelli, "The smt-lib standard – version 2.0," in *In Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (SMT '10)*, 2010.

[9] M. Wenzel, L. Paulson, and T. Nipkow, "The isabelle framework (invited tutorial)," in *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, pp. 33–38, 2008.

[10] L. C. Paulson, "The foundation of a generic theorem prover," *Journal of Automated Reasoning*, vol. 5, no. 3, pp. 363–397, 1989.

[11] L. C. Paulson, "Isabelle: The next 700 theorem provers," *CoRR*, vol. cs.LO/9301106, 1993.

[12] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL.* Springer, 2002. Latest online version December 12 2016. `http://isabelle.in.tum.de/dist/Isabelle2016-1/doc/tutorial.pdf`.

[13] M. Wenzel, *Isar — A Generic Interpretative Approach to Readable Formal Proof Documents*, pp. 167–183. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999.

[14] T. Nipkow, "Programming and proving in isabelle/hol." `http://isabelle.in.tum.de/dist/Isabelle2016-1/doc/prog-prove.pdf`, 2016. Updated with every new version of Isabelle/HOL.

[15] 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 the 8th International Workshop on the Implementation of Logics (IWIL-2010)*, 2010.

[16] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel, *Truly Modular (Co)datatypes for Isabelle/HOL*, pp. 93–110. Springer International Publishing, 2014.

[17] J. Biendarra, J. C. Blanchette, M. Desharnais, L. Panny, A. Popescu, and D. Traytel, “Defining (co)datatypes and primitively (co)recursive functions in isabelle/hol.” `http://isabelle.in.tum.de/dist/Isabelle2016-1/doc/datatypes.pdf`, 2016. Updated with every new version of Isabelle/HOL.

[18] R. Bird and P. Wadler, *An Introduction to Functional Programming.* Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1988.

[19] J. Hurd, “First-order proof tactics in higher-order logic theorem provers,” in *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports*, pp. 56–68, 2003.

[20] A. Lochbihler, “Coinductive,” *Archive of Formal Proofs*, Feb. 2010. `http://isa-afp.org/entries/Coinductive.html`, Formal proof development.

[21] J. C. Blanchette, A. Bouzy, A. Lochbihler, A. Popescu, and D. Traytel, *Friends with Benefits*, pp. 111–140. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017.

[22] R. Leino and M. Moskal, “Co-induction simply: Automatic co-inductive proofs in a program verifier,” tech. rep., July 2013.

[23] D. Lucanu, E.-I. Goriac, G. Caltais, and G. Roşu, *CIRC: A Behavioral Verification Tool Based on Circular Coinduction*, pp. 433–442. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009.

[24] D. Pous, “Coinduction all the way up,” in *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS ’16, (New York, NY, USA), pp. 307–316, ACM, 2016.

[25] C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis, “The power of parameterization in coinductive proof,” *SIGPLAN Not.*, vol. 48, pp. 193–206, Jan. 2013.

[26] M. Johansson, L. Dixon, and A. Bundy, “Conjecture synthesis for inductive theories,” *Journal of Automated Reasoning*, vol. 47, pp. 251–289, Oct 2011.

[27] O. Montano-Rivas, R. McCasland, L. Dixon, and A. Bundy, “Scheme-based theorem discovery and concept invention,” *Expert systems with applications*, vol. 39, no. 2, pp. 1637–1646, 2012.

[28] R. L. McCasland, A. Bundy, and P. F. Smith, “Mathsaid: Automated mathematical theory exploration,” *Applied Intelligence*, Jun 2017.

Appendix A

Evaluation theorem list

Here the theorems we used to perform evaluation on are listed by name along with information about the evaluation results. Further information about the evaluation can be found in Chapter 5. Dir. prove stands for directly provable, Prov. expl. stands for provable after exploration, Disc stands for discovered in exploration, Fully expl. stands for fully explorable and Obs. stands for requires observer. Explanations of these concepts can be found in section 5.3. The source code for these theorems can be found on github [1], where each theorem is contained in a file with a name matching the theorem name.

[1]`https://github.com/moajohansson/IsaHipster/tree/master/benchmark/she`

Name	Source	Dir. prov.	Prov. expl.	Disc	Fully expl.	Obs.
`lappend_LNil2`	CoList	X	–	X	X	
`lappend_assoc`	CoList	X	–	X	X	
`lmap_lappend_distrib`	CoList	X	–	X	X	
`lappend_inf`	CoList	X	–			
`lprefix_antisym`	CoList	X	–			
`llength_lmap`	CoList		X	X	X	
`llength_lappend`	CoList	X	–		X	
`ltake_lmap`	CoList				X	
`lmap_literates`	CoList		X	X		X
`lappend_literates`	CoList	X	–	X	X	X
`ltake_lzip`	CoList	X	–		X	
`lzip_literates`	CoList	X	–			
`lzip_lmap`	CoList		X			
`ltakeWhile_lmap`	CoList				X	
`ltakeWhile_K_True`	CoList	X	–			
`lzip_ltakeWhile_fst`	CoList	X	–			
`lzip_ltakeWhile_snd`	CoList	X	–			
`ltakeWhile_repeat`	CoList	X	–		X	X
`smap_unfold_stream`	CoStream	X	–		X	X
`unfold_stream_ltl_unroll`	CoStream	X	–		X	X
`unfold_stream_id`	CoStream	X	–		X	X
`szip_iterates`	CoStream	X	–			X
`szip_smap1`	CoStream	X	–			X
`szip_smap2`	CoStream	X	–			X
`szip_smap`	CoStream	X	–			X
`smap_fst_szip`	CoStream	X	–			X
`smap_snd_szip`	CoStream	X	–			X
`szip_sconst1`	CoStream	X	–			X
`szip_sconst2`	CoStream					X
`stream_of_llist_llist_of_stream2`	CoStream	X	–			X
`llist_of_stream_unfold_stream2`	CoStream					X
`lmap_llist_of_stream2`	CoStream					X
`llist_of_stream_siterates2`	CoStream	X	–			X
`lzip_llist_of_stream2`	CoStream	X	–			X
`Eplus_zero`	CoTut	X	–	X	X	
`Eplus_assoc`	CoTut	X	–	X	X	
`Eplus_commut`	CoTut		X	X	X	
`Pls_assoc`	CoTut	X	–		X	X
`Pls_ac`	CoTut	X	–		X	X
`Pls_comm`	CoTut	X	–		X	X
`Pls_assoc_enat`	CoTut		X		X	X
`Pls_ac_enat`	CoTut		X		X	X
`Pls_comm_enat`	CoTut		X		X	X
`Prd_distribR`	CoTut					X
`Prd_comm`	CoTut					X
`Prd_assoc`	CoTut					X
`llist_of_tree_of`	CoTut				X	
`llist_of_tree_of2`	CoTut				X	
`chop_tmap`	CoTut					