Thesis for The Degree of Licentiate of Engineering

# Theory Exploration for Programs and Proofs

Sólrún Halla Einarsdóttir



Division of Computing Science
Department of Computer Science & Engineering
Chalmers University of Technology | University of Gothenburg
Gothenburg, Sweden, 2022

**Theory Exploration for Programs and Proofs**

Sólrún Halla Einarsdóttir



Department of Computer Science & Engineering
Division of Computing Science
Chalmers University of Technology and Gothenburg University
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Digitaltryck,
Gothenburg, Sweden 2022.

# Abstract

We have built two theory exploration systems, *Cohipster* and *RoughSpec*. Theory exploration is a method of automatically conjecturing properties about the functions and structures that appear in a computer program or a formalization of a mathematical theory.

Cohipster is a theory exploration system that discovers equational lemmas about corecursive functions in Isabelle/HOL and automatically searches for coinductive proofs for them. Coinduction and corecursion are the mathematical duals of induction and recursion and allow the specification of potentially infinite structures such as streams, and functions that operate on such structures. Cohipster is the first system to automatically discover and prove coinductive lemmas, and its design required the development of techniques for testing infinite structures as well as for automating coinductive proofs.

RoughSpec is a template-based theory exploration system for Haskell programs. RoughSpec allows users to specify what kinds of properties they are interested in finding by using templates. A template is an expression describing a family of properties, such as distributivity or commutativity, that have a particular shape. Limiting the search space to specific shapes of properties makes theory exploration more targeted and tractable than previous methods.



**Keywords**

Theory Exploration, Conjecture Generation, Property-Based Testing, Theorem Proving, Automated Reasoning, Artificial Intelligence, Functional Programming, Coinduction

# Acknowledgment

I'd like to thank my supervisor Moa Johansson for all the encouragement and guidance and for always reminding me to have fun in research. Thank you to my cosupervisor Nick Smallbone for always being available and willing to answer my technical questions and help me solve any problems I run into. Thank you to Aarne Ranta for your feedback and support as examiner. Thanks to Johannes Åman Pohjola for guidance in my MSc thesis research and collaboration on the Cohipster paper. And thank you Cezary Kaliszyk for agreeing to be discussion leader for this licentiate!

Thank you to my past and present colleagues at CSE who have made the workplace so fun and friendly, I hope we can soon return to a more "normal" work life and enjoy many coffee breaks, lunches, and after-work beers together.

Special thanks to my friends for their company, support, and commiseration during the past two years of pandemic and parental leave life.

Thank you to my family, in particular my mother Eyja, my father Einar and my sisters: Védís, Iðunn, Edda, and Una. I can feel your love and support in everything I do, and I hope the feeling is mutual.

Koen, I'm so happy and grateful to have you as my partner. Thank you for all your support, encouragement, and love. Sander and Vincent, thank you for your friendship and acceptance. Last but not least, thank you Pieter Snæbjörn for helping me put everything into perspective and filling every day with joy.

My PhD research is supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation.

# List of Publications

## Appended publications

This thesis is based on the following publications:

[A] Sólrún Halla Einarsdóttir and Moa Johansson and Johannes Åman Pohjola "Into the Infinite - Theory Exploration for Coinduction"
*Proceedings of AISC 2018.*

**Contribution:** I did all of the technical implementation work and carried out two of the three case studies described in the paper. The paper was written in collaboration with me as the main driver.

[B] Sólrún Halla Einarsdóttir and Nicholas Smallbone and Moa Johansson "Template-based Theory Exploration: Discovering Properties of Functional Programs by Testing"
*IFL '20.*

**Contribution:** I did all of the technical implementation work and carried out most of the case studies described in the paper. The paper was written in collaboration with me as the main driver.

## Other publications

The following publications were published during my PhD studies. However, they are not appended to this thesis, due to contents not related to the thesis.

[a] Maximilian Algehed, Patrik Jansson, Sólrún Halla Einarsdóttir, Alex Gerdes "Saint: an API-generic type-safe interpreter"
*International Symposium on Trends in Functional Programming (TFP) 2018*

[b] Patrik Jansson, Sólrún Halla Einarsdóttir, Cezar Ionescu "Examples and results from a BSc-level course on domain specific languages of mathematics"
*Proceedings of TFPIE 2018*

[c] Dhasarathy Parthasaraty, Karl Bäckstrom, Jens Henriksson, Sólrún Einarsdóttir "Controlled time series generation for automotive software-in-the-loop testing using GANs"
*2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*

# Contents

# Chapter 1

# Introduction

In this thesis we present extensions and applications of theory exploration. Theory exploration is an approach to automatically discovering interesting and useful properties about the functions and data structures that appear in computer programs and mathematical theories. In a programming context, knowing what properties hold about the functions and data structures in our program can help us with testing, debugging, verification, or extending the program. In a theorem proving context, automatically coming up with new conjectures and lemmas can help with proof automation or to guide an interactive proof process.

As an example, when asked to generate conjectures about addition and multiplication on natural numbers, our theory exploration system RoughSpec will produce the output shown in Figure 1.1 in less than half a second. The generated conjectures include identity, commutativity, distributivity, and associativity properties.

The tools we present have

[a] Allowed for the application of theory exploration to a new domain, with the first system to automatically discover and prove coinductive lemmas.

[b] Made theory exploration faster, more tractable, targeted and versatile, with a system that allows the user to specify what shapes of properties they want it to discover.

```
Searching for fix-point/id properties...
  1. x * 0 = 0
Searching for left-id-elem properties...
  2. 0 + x = x
  3. 1 * x = x
Searching for right-id-elem properties...
  4. x + 0 = x
  5. x * 1 = x
Searching for commutative properties...
  6. x + y = y + x
  7. x * y = y * x
Searching for operator-commutative properties...
  8. x + (y + z) = y + (x + z)
  9. x * (y * z) = y * (x * z)
Searching for distributive properties...
 10. x * (y + z) = (x * y) + (x * z)
Searching for associative properties...
 11. (x + y) + z = x + (y + z)
 12. (x * y) * z = x * (y * z)
```

Figure 1.1: Arithmetic properties discovered by RoughSpec.

## 1.1 Background

### Our approach to AI

Traditionally, artificial intelligence (AI) has been divided into two approaches. On the one hand there are symbolic methods which are explicitly embedded with symbolic representations of reasoning, and on the other data-driven machine learning methods that learn without explicit instructions, by using algorithms to recognize patterns in a large set of examples. These days, the buzz is centered around machine learning systems that have been achieving more and more impressive results in recent years. Training a machine learning algorithm to perform well most often requires a huge training data set and a significant amount of computing power. For example the Generative Pre-trained Transformer 3 (GPT-3) [1] is a recent deep learning model for natural language model that has been trained on a broad swath of internet data and is able to produce human-like text. The largest version of the model contains 175 billion parameters, was trained on a dataset of around 500 billion byte-pair-encoded tokens, and consumed several thousand petaflop/s-days of compute during pre-training.

The approach to AI described in this thesis is not machine learning but also not purely symbolic. It is in part data-driven but does not require us to provide a dataset for training and testing – rather we generate random test data using methods that are fast and efficient. In this way we generate new knowledge, using tools that can be run on a normal laptop and come up with results in seconds.

## QuickSpec and Hipster

The work presented in this thesis is built on top of two theory exploration systems developed at Chalmers, QuickSpec [2] and Hipster [3, 4].

### QuickSpec

QuickSpec [2] is a theory exploration system that discovers equational properties about Haskell programs. It takes in a set of functions given by the user, generates all correctly typed terms up to a given size limit, and evaluates them for a large set of randomly generated inputs, using the QuickCheck [5] property based testing tool. Terms that have the same value for every test input are then conjectured to be equal.

### Hipster

Hipster is a theory exploration system for Isabelle/HOL, built on top of QuickSpec. It uses QuickSpec to come up with lemmas about a set of functions given by the Isabelle user, and then uses proof tactics implemented in Isabelle to attempt to prove the generated lemma. It can discard a lemma as too trivial to be of interest if its proof is "too easy" or if it's a consequence of a previously discovered lemma. The definition of simple and complicated proof can be set by the user. For example, proofs that only require rewriting might be considered simple while proofs that require induction are considered to indicate an interesting lemma.

## Historical context and motivation

In 1955, AI pioneers Newell and Simon presented a specification of the logic theorist [6], a system that generated proofs for logic theorems using heuristic methods *"similar to those that have been observed in human problem solving activity"* that could successfully prove many theorems from *Principia Mathematica* [7]. Newell and Simon were very optimistic about the development of AI that would surpass human abilities in mathematics as well as other domains, as were many of their peers at the time. In 1958 they predicted that *"within ten years a digital computer will discover and prove an important new mathematical theorem."* [8].

This prediction certainly did not come true in the ten following years, and even now more than 60 years later it has not yet been realized in its entirety, although various qualified versions of the statement have come true. Why is this?

The early AI visionaries believed it would be straightforwardly possible to create systems that could simulate the reasoning abilities of a human, by programming them with the correct heuristic rules. This turned out not to be the case.

We believe that such a discovery requires a level of creativity and insight that AI has yet to achieve. Discovering a new theorem may require the ability to represent the same concept in many different ways, making generalizations from specific examples, or reasoning step-by-step by inventing new lemmas that hold in the current context. Furthermore, determining what kinds of

properties are interesting, useful, and worth knowing about is not obvious. Theory exploration can help provide the "creative" step of coming up with new lemmas that hold in a given context, paving the way to more complicated conjecturing.

When it comes to proving theorems, the proof may require new lemmas that are not yet known in the proof context, and theory exploration can help us find them. More generally, in a theorem proving context, whether or not we have a specific goal theorem to prove, learning additional properties that hold for the given set of axioms will advance our knowledge and help to prove more advanced theorems. Theory exploration tools that generate lemmas can complement automatic tools, making them more powerful. A human mathematician may also find a lemma generator useful in their proof development. This is especially true in the case of proofs by induction, which often require additional lemmas. For example using QuickSpec in combination with Vampire greatly improves its ability to prove benchmarks for inductive provers, as shown in [9]. The usefulness of QuickSpec in automating inductive proofs has also been demonstrated by the HipSpec system [10], and Hipster's abilities to guide interactive inductive proof development is discussed and illustrated in [3].

## 1.2 Literature survey

Following is an overview of systems that generate conjectures and lemmas using various different methods.

### Methods based on heuristics and symbolic reasoning

The AM system [11] was an AI system designed to develop new mathematical concepts, developed in the 1970s. Guided by a large body of heuristic rules, it incrementally extends its knowledge base, eventually rediscovering various mathematical concepts and theorems. For example, starting with knowledge about elementary concepts such as sets, operations, and equality, AM discovered concepts corresponding to natural numbers, multiplication, and factors, and conjectured known relationships including the unique factorization theorem.

The HR system [12] was designed to generate concepts and conjectures in mathematical domains. It used a model finder combined with heuristic search rules to generate conjectures of interest, starting from a set of axioms. Its applications included lemma generation as well as generating benchmark theorems to test automatic theorem provers. It displayed some ability to invent interesting new mathematical knowledge, for example when applied to number theory it discovered interesting and previously undefined sequences [13].

Buchberger [14] introduced the term "theory exploration" (earlier work used the term "theory formation") and his team implemented it in the Theorema [15] system. Theorema provides tools to assist the user in their theory exploration but does not automate the process, rather it is intended as a framework for mathematicians to more effectively work with an interactive theorem prover.

MATHsAiD [16] is an automated theorem-discovery tool designed to assist a working mathematician, which has mainly been applied in the context of abstract algebra. It takes a set of axioms and definitions provided by the user

and discovers new knowledge using a forward reasoning process, instantiating templates called theorem shells and generating theorems in parallel with their proofs. It uses a variety of heuristics constructed to reflect human mathematical intuition and knowledge. This system managed to rediscover a known but nontrivial theorem on Zariski spaces.

## Methods based on term generation and testing

Graffiti [17] was a conjecture generation system for graph theory, developed in the 1980s. It attempted to generate conjectures of specific predetermined shapes, comparing them against a library of graphs such that a formula for which none of the library graphs provides a counterexample is considered a conjecture, and used various heuristics to trim away trivial and uninteresting conjectures.

IsaCosy [18] and IsaScheme [19] are theory exploration systems for Isabelle/HOL that use term generation and testing. IsaScheme makes use of user-provided templates (schemes) similarly to our approach in RoughSpec, presented in Paper B. The generated conjectures are then automatically refuted using Isabelle/HOL's counter-example finders, or proved using the IsaPlanner [20, 21] prover.

Similarly to QuickSpec, the TheSy [22] system uses comprehensive term generation. However rather using testing their system is purely symbolic, evaluating the terms on symbolic inputs to form conjectures. Like Hipster, it then uses an inductive theorem prover to attempt to prove the generated conjectures.

Speculate [23] is another theory exploration system from the functional programming community, designed to discover interesting properties about Haskell programs. It can discover inequalities and conditional properties in addition to equational properties.

## Methods based on machine learning

### Conjecture generation by neural networks

Urban and Jakubův [24] use neural methods to generate conjectures, training the transformer model GPT-2 on the Mizar Mathematical Library [**?**]. They generate novel and well-typed conjectures in the Mizar format but the output may also include duplicates from the training set as well as false statements.

Rabe et al. [25] use language modeling applied to mathematical formulas to automate logical reasoning. They attempt reasoning tasks such as generating a missing precondition for a given conditional statement, generating one side of an equation given the other side, and "free-form" conjecturing. Between 13-30% of statements their system generated were both provable and new while the remainder were either false or trivial consequences (ie. alpha-renamings or even exact copies) of statements from the training set.

### Machine-learning guiding human mathematicians

Recently Davies et al. [26] demonstrated how an AI system can be used to guide a human mathematician's intuition towards achieving novel mathematical

results. They used supervised learning and attribution techniques to verify whether an intuition a mathematician may have about the relationship between two objects is worth looking into and guidance as to how they may be related if so. This method was used to discover and prove a novel theorem in knot theory and to prove a theorem that resolves a well-known conjecture in representation theory.

In the knot theory example the mathematicians hypothesized a relationship between a set of geometric and algebraic invariants for knots and the machine learning system helped to see that a certain algebraic invariant could be predicted based on three of the geometric ones. In the representation theory example there was a conjecture stating that the KL (Kazhdan-Lusztig) polynomial of a pair of elements in a symmetric group can be calculated from their unlabelled Bruhat interval, which is a large directed graph. The ML system helped determine that in fact only a certain subgraph is needed and further examining subgraph edges deemed most significant by the ML system led to the discovery that the Bruhat interval can be decomposed into two parts and a proof that the KL polynomial can be computed from these two parts through a "beautiful formula."

This work has been criticized by [27] for overstating the role of the AI system in achieving the mathematical results. In particular, bold headlines and statements in popular science media about these results such as *"Researchers create AI that can create brand new math theorems"* [28]. *"AI is discovering patterns in pure mathematics that have never been seen before... We can add suggesting and proving mathematical theorems to the long list of what artificial intelligence is capable of."* [29] would have the reader believe that the AI discovered and proved the theorems all on its own, thus realizing Newell and Simon's vision from 1958. In fact the role of the AI was rather small– it guided the mathematicians towards discovering the new theorems, by identifying and analyzing correlation between different factors, and the AI system had no role in proving the theorems.

On the other hand, theory exploration systems such as ours can indeed suggest mathematical theorems, although usually not new or especially complicated ones.

## 1.3 Contributions of included papers

### Paper A: Cohipster

In Paper A we present an extension of theory exploration to coinductive theories, and a tool called Cohipster that can discover and prove coinductive properties.

Coinduction and corecursion are the mathematical duals of induction and recursion and allow the specification of potentially infinite structures, for example streams, and functions that operate on such structures. Cohipster is an extension of Hipster that discovers and proves equational properties about corecursive functions in Isabelle/HOL. Prior to this work, Hipster only had capabilities to discover properties about recursive functions and prove them by induction.

Cohipster takes a set of functions as an argument and generates conjectures of equational properties. It then attempts to prove the generated conjectures automatically, discarding those that are trivial to prove, presenting the conjectures it finds nontrivial proofs of to the user as lemmas, and any unproved conjectures are left for the user to try to prove or disprove. Developing this tool required the extending QuickSpec with methods for testing infinite values, as well as the development of methods to automate coinductive proofs in Isabelle.

For example when we call on Cohipster for a theory of infinite binary trees, containing a function *mirror* for reflecting a tree and a function *tmap* for mapping a function over the tree elements, Cohipster will discover the lemmas

$$mirror\,(mirror\ t) = t$$

and

$$tmap\,f\,(mirror\ t) = mirror\,(tmap\ f\ x)$$

Cohipster has conjectured these lemmas with the use of QuickSpec, generated formal machine-checked proofs in Isabelle, and determined that the lemmas are not too trivial to be of interest. This process takes just under a minute. The Isabelle user can then import these lemmas into their theory with a mouse-click.

Cohipster is the first theory exploration tool to be capable of handling infinite structures and discovering coinductive properties about them as well as generating coinductive proofs.

## Paper B: RoughSpec

In paper B we introduce the RoughSpec tool for template-based theory exploration. QuickSpec's exhaustive methods of term generation and testing make it ill-suited for exploring large numbers of functions at the same time and limited in the ability to discover properties containing large terms, as this can take a very long time and produce an overwhelming amount of output [2]. RoughSpec offers great improvements in these areas, while sacrificing completeness. While QuickSpec tests all properties up to a size limit, RoughSpec only considers properties matching templates defined by the user.

A template is an expression describing a family of properties such as commutativity or distributivity. We represent a template as a Haskell equation containing functions, variables, and *metavariables*. For example the following is a template for commutative properties (in our syntax, variables are written in uppercase, and a metavariable is written as a variable with a leading question mark): `?F X Y = ?F Y X`.

RoughSpec takes in a set of functions along with a set of templates, and generates conjectures about the functions that match one of the given templates. For each given template, RoughSpec instantiates the metavariables in the template with the functions in scope to create type-correct equations which are then tested.

For example, if we call RoughSpec using the above template along with addition and subtraction on integers it will come up with the conjecture `x+y = y+x` (and no further output), in less than 0.1 seconds. In the example shown at the beginning of the chapter in Figure 1.1 we used this commutativity template and an additional six templates describing identity properties, distributivity and associativity, shown in Figure 1.2:

```
?F(?X)   = ?X          ?F(?G(X))      = ?G(?F(X))
?F(?Y,X) = X           ?F(?G(X,Y))    = ?G(?F(X),?F(Y))
?F(X,?Y) = X           ?F(?F(X,Y),Z) = ?F(X,?F(Y,Z))
?F(X,Y)  = ?F(Y,X)
```

Figure 1.2: Templates used for the example in Figure 1.1.

Since we only generate terms and test conjectures that match the given templates, RoughSpec potentially has a much smaller search space than QuickSpec. It is well suited to exploring larger libraries of functions and performing fast and targeted searches for properties. RoughSpec solves the problem of determining what conjectures are interesting by letting the user specify what kinds of properties they are interested in finding. We have also combined RoughSpec with QuickSpec, using QuickSpec to perform a complete search for smaller term sizes, while using templates for larger, more complex properties, in order to leverage the strengths of both systems.

## 1.4 Future Work and Conclusions

We believe that theory exploration can be useful in many contexts and have many exciting ideas about further extensions and novel applications of our systems.

### Data-driven template generation

We have an ongoing project where we are collecting a library of equational properties and analyzing their shapes, extracting the templates that could be used to generate them. We intend to determine what makes a "good" template and learn which templates will be useful in which context.

### Better methods for discovering conditional properties

At this time our systems are mostly limited to discovering equational conjectures. QuickSpec has limited support for discovering conditional properties, discussed in [2], and Hipster has been extended with support for discovering conditional lemmas [30]. These methods require the user to provide predicates to be used as conditions, and can lead the generation of a large amount of potentially uninteresting output as QuickSpec's pruning methods become much more limited when conditions are involved. Improved abilities to discover conditional properties would enable many new applications of our tools.

RoughSpec could easily be used to generate conditional properties if we give it conditional property templates, or templates for conditions to be combined with equational property templates. In our work on to determine what templates are good, described above, we could also analyze collections of conditional properties and attempt to determine what conditions are useful in a given context.

## Theory exploration as a part of AI for mathematics

We would like to examine what role theory exploration tools such as ours can play in AI for mathematics. A hybrid system combining theory exploration, machine learning, and automated reasoning methods is an exciting prospect.

One example of an exciting goal in AI for mathematics is the IMO grand challenge [31]. The challenge is to create an AI system that can win a gold medal in the International Mathematical Olympiad, an elite mathematics competition for high school students. There is ongoing work on developing neural theorem provers targeting this domain of problems [32, 33] but they have achieved only moderate success. Solving the more challenging Olympiad problems requires great ingenuity and it is not at all clear how to go about designing an AI system that would be good at it. We believe the ability to generate lemmas could be of great use here. For instance, one of the of the major problem domains in these competitions is number theory, where the problem might introduce a property or relation on natural numbers and ask for a proof of a statement about the introduced concepts. In this context inventing inductive lemmas could be very useful coming up with a solution proof.

Another exciting goal would be to collaborate with mathematicians and assist them in the discovery of a new and important mathematical result. We envision a workflow for a mathematician coming up with and proving new theorems where a theory exploration tool for conjecture generation is used in collaboration with machine learning and automated reasoning. For example one might use a machine learning system trained on a large set of examples to identify correlations as in [26] and then make use of a conjecture generation system to further explore those relationships.

## Making theory exploration more usable and useful

Of course one of our goals is to reach more users and facilitate their ability to use our tools to do cool stuff.

**For mathematicians** We would like for our tools to be useful for mathematicians who are developing new theorems, either using a proof assistant or by hand. Automatic lemma discovery can save them time and effort and help them gain a better understanding of the structures and functions they define, and the discovered lemmas may be useful in the proofs of more advanced theorems.

This may require integration with more proof assistants or a more user-friendly interface.

**For programmers** The original intention of QuickSpec was to generate a specification for a functional program. It could be made more useful as a tool in a programmer's workflow and even be integrated into some kind of developer tool.

Currently, QuickSpec requires the user to explicitly type up a signature. A default signature for a given set of functions could be automatically generated using Template Haskell (indeed the interface between Hipster and QuickSpec already does something along those lines).

QuickSpec requires generators to produce test data, and it is not always straightforward to implement such a generator (as discussed by Lampropoulos et al. in [34]). The methods introduced by Mista et al. in [35] could be used to synthesize generators. The test data needs to be computable, or have a computable proxy that can be used instead (like we do to test infinite structures in Cohipster). This limits the applicability of QuickSpec to Haskell programs as many datatypes are difficult to generate test data for. The methods developed in paper A for testing infinite structures could be extended to test non-terminating programs.

# Chapter 2

# Paper A

**Into the Infinite - Theory Exploration for Coinduction**

**Sólrún Halla Einarsdóttir and Moa Johansson and Johannes Åman Pohjola**

***Proceedings of AISC 2018.***

# 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.

## 2.1 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 [36], Hoare logic for non-terminating programs [37], total functional programming in the presence of non-termination [38], 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 [39–41].

In this paper, we extend a technique, called *theory exploration* [14], and present a tool that automatically discovers and proves equational properties about corecursive functions in the proof assistant Isabelle/HOL [42], 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:

```
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* [43]. 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 [2], which heuristically generates type-correct terms and uses automated testing to invent interesting candidate lemmas. We give a brief introduction to QuickSpec in Section 2.2, along with a lightweight introduction to coinduction.

Earlier versions of Hipster [3, 4] 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 2.3 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 2.5. Results are encouraging: we can discover and prove many well-known and useful properties. Similar theory exploration systems can be found in the literature [10, 16, 18, 19], 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.[1]

## 2.2 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

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

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 2.1:

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

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 [44] or [45].

## Isabelle/HOL

Isabelle/HOL is an interactive proof assistant for higher-order logic [42]. 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 [46]. 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.

## Hipster

The architecture of the Hipster system is shown in Figure 2.1. 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 [5]. 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 [2]. 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

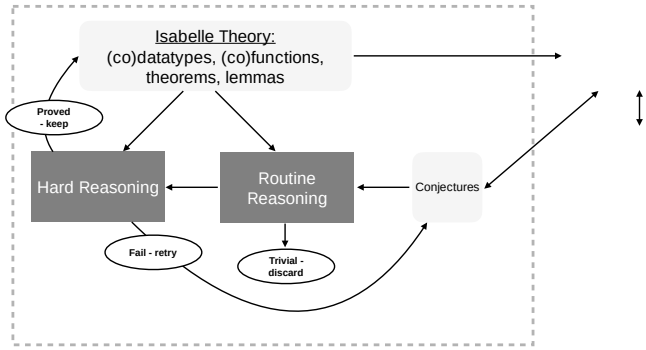



Figure 2.1: The architecture of the Hipster system.

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.

## 2.3 Testing infinite structures

Recall from Section 2.2 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 [2]. 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* [47, 48]. 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 2.1, 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 generate a new type in place of the undefined value $\bot$ 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 reduces $n$ by *(n-1)/#constructors* in each recursive call. For *OStream*, this is simply $(n-1)$, while for e.g. binary trees it is $(n-1)/2$ for each branch.

## 2.4 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 [49]; 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 2.1:

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

Note that the (weak) coinduction principle shown in Section 2.2 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 2.5.1.

The recent AmiCo definitional package by Blanchette et al. [50] 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 [46], 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 LHS and RHS of the conjecture is 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 2.5).

## 2.5 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 2.3, is effective in discovering coinductive properties and that our automated coinduction tactic, described in Section 2.4, 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[2].

### 2.5.1 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 [51] in the archive of formal proofs[3], 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 [4]. 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 2.1 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

[2] `https://github.com/moajohansson/IsaHipster/tree/master/benchmark/AISC18`
[3] `https://www.isa-afp.org/`
[4] `http://isabelle.in.tum.de/library/HOL/HOL-Library/Extended\_Nat.html`

| **cohipster** args | Expl | Expl+Proof | # properties | # lib. 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 2.1: An overview of the results of exploring our lazy list theory.

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, # lib. 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 2.4.

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 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.

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 2.1, 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 2.1 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.

### 2.5.2 Case Study: Stream Laws

We already saw in Section 2.1 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 [52], 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 2.2. Lemmas whose precision (shown in column Prec.), recall (shown in column Rec.) and time (column T) have been explored together by invoking Hipster with every function mentioned in the lemmas; e.g., to search for laws 7-9 we invoke **cohipster** `map zip fst snd`. The column Disc. contains an X for lemmas that were discovered and – for those that were not. 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 2.4..

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

| | Property | Disc. | Prec. | Rec. | T |
|---|---|---|---|---|---|
| 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 2.2: An overview of the stream properties discovered and proved by Hipster. Lemmas in gray are not in scope.

up-to friendly contexts. The 21% overall precision can improved by using a more powerful routine tactic, such as simplification interleaved with stream expansion.

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.

### 2.5.3 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 2.3. 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*<br>+ 3 more proved by Sledgehammer | coinduction+simp |
| *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*<br>*tsum (tsum x y) z = tsum x (tsum y z)*<br>*mirror (tsum y (mirror x)) = tsum x (mirror y)*<br>*tsum (mirror x) (mirror y) = mirror (tsum x y)*<br>+ 2 more proved by Sledgehammer | coinduction+smt<br>coinduction+smt<br>coinduction+smt<br>Sledgehammer (using above lemmas) |

Table 2.3: 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 Tree 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 that take the length of the list into account is future work.

| **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)*<br>*tsum y x = tsum x y*<br>*tsum (tsum x y) z = tsum x (tsum y z)*<br>+ 4 more unproved about tsum/mirror | Sledgehammer<br>no<br>no |

Table 2.4: 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.

As can be seen in Table 2.4, 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.

## 2.6 Related Work

There is substantial recent work on making Isabelle/HOL more expressive for working with codatatypes and corecursive functions [39, 50]. 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 [53] Leino and Moskal present a method for automated reasoning about coinductive properties in the Dafny verifier. CIRC [54] 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.

IsaCoSy [18] and IsaScheme [19] are other theory exploration systems for Isabelle/HOL, both of which focus on the discovery and proof of inductive properties. MATHsAiD [16] 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 [10].

## 2.7 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, 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 behavioural equivalences, or discovering Hoare triples about non-terminating

programs. This requires 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.

# Chapter 3

# Paper B

**Template-based Theory Exploration: Discovering Properties of Functional Programs by Testing**

**Sólrún Halla Einarsdóttir and Nicholas Smallbone and Moa Johansson**

***IFL '20.***

# Abstract

We present RoughSpec, a template-based extension of the theory exploration tool QuickSpec. QuickSpec uses testing to automatically discover equational properties about functions in a Haskell program. These properties can help the user understand the program or be used as a source of possible lemmas in proofs of the program's correctness. In RoughSpec, the user supplies templates, which describe families of laws such as associativity and distributivity, and we only consider properties that match the templates. This restriction limits the search space and ensures that only relevant properties are discovered. In this way, we sacrifice broad search for more direction towards desirable property patterns, which makes theory exploration tractable and scalable. We also combine RoughSpec with QuickSpec, using QuickSpec to perform a complete search for smaller term sizes, while using templates for larger, more complex properties, in order to leverage the strengths of both systems.

## 3.1 Introduction

One strength of functional programming is that programs are easy to reason about. Pure functions often obey simple formal specifications which, as long as the programmer writes them down, are a great help in programming. A formal specification can be proved correct, automatically tested with a tool such as QuickCheck [5] or SmallCheck [55], or simply read in order to understand a code base.

Many functional programmers already specify their code, by writing e.g. QuickCheck properties, but many do not. Can those who do *not* specify their code also reap the benefits of formal specification? The answer is *yes*: given a piece of code, we can automatically infer properties about it.

A tool that infers properties from code is called a *theory exploration system*. Two theory exploration systems for Haskell are QuickSpec [2] and Speculate [23]. These tools take as input a collection of Haskell functions and, through testing, discover formal properties which can be expressed using those functions. For example, given the list functions `++`, `reverse`, and `map`, QuickSpec discovers a total of five laws, all of them well-known and useful:

```
reverse (reverse xs) = xs
map f (reverse xs) = reverse (map f xs)
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
reverse xs ++ reverse ys = reverse (ys ++ xs)
map f xs ++ map f ys = map f (xs ++ ys)
```

Both tools work in a similar way. They take as input a *signature*, which describes the set of functions we would like to explore. Very roughly, they potentially consider *all* possible properties, up to some size limit, which can be built from the given functions (and some variables). They (1) build a set of terms from the given functions, (2) use automatic testing to check which terms appear to be equal and build equations from the equal terms,[1] resulting in a set of equational properties which are likely to be true, (3) remove any redundant properties (a property is redundant if it can be derived from other discovered properties), and (4) report all the non-redundant discovered properties. Because they explore all possible properties, the generated specification is *complete* (up to the size limit).

This approach works well on small sets of functions. Completeness means that we get an expressive specification, and discarding redundant properties keeps the specification short. When given only a few functions, QuickSpec and Speculate typically produce clear, crisp and useful specifications, like the one above. We have found that studying the output of QuickSpec is a great help in understanding an unfamiliar API.

Unfortunately, this approach breaks down when exploring large APIs: a *complete* theory exploration system simply finds too many laws. In a benchmark running QuickSpec on about 30 list functions [2], over 500 laws were found! The QuickSpec user is unlikely to bother reading all 500. Many are unenlightening, such as this one:

```
map (f x) (take (succ 0) xs) = zipWith f (scanl g x []) xs
```

[1] Speculate also discovers non-equational properties.

This law is found, not because it was interesting, but because it was true and because QuickSpec did not consider it to be redundant. When we explore large APIs, we often get huge numbers of uninteresting laws. Furthermore, the search space is huge so the tools often take a while to run: exploring the 30 list functions took about two hours. These problems arise precisely because QuickSpec and Speculate are complete—unless a law is redundant, it will get discovered and printed out, interesting or not.

### 3.1.1 RoughSpec

We have developed a new theory exploration system, RoughSpec. Like QuickSpec and Speculate, it takes as input a set of Haskell functions (which we call the *signature*), and uses testing to find properties that seem to hold. The difference is that RoughSpec is *incomplete*: it does not try to find all true properties.

Instead, the user gives a set of *templates*, expressions which describe a family of laws such as associativity or distributivity. RoughSpec searches only for instances of these templates. In this way, the user can specify what kind of properties they would find interesting, and RoughSpec searches only for these properties.

A template is a Haskell equation containing functions, variables and *metavariables*. For example, here is a template which represents commutativity (note that in our syntax, variables are written in uppercase, and a metavariable is written as a variable with a leading question mark):

```
?F X Y = ?F Y X
```

When a template contains metavariables, RoughSpec *instantiates* them with functions drawn from the signature, tests the resulting equations, and reports any that appear to hold. In this case, RoughSpec will search for functions `?F` such that `?F X Y = ?F Y X` for all `X` and `Y` — that is, for commutative functions.

Here are some more examples of templates. They describe: (1) associativity; (2) an invertible function; (3) distributivity; (4) and (5) a function having an identity element; and (6) a list homomorphism:

[a] `?F (?F X Y) Z = ?F X (?F Y Z)`

[b] `?F (?G X) = X`

[c] `?F (?G X) (?G Y) = ?G (?F X Y)`

[d] `?F X ?E = X`

[e] `?F ?E X = X`

[f] `?F (X ++ Y) = ?G (?F X) (?F Y)`

In (4) and (5), `?E` is a metavariable, and will be replaced by *constants* drawn from the signature, while `X` is a universally-quantified variable. Note too that apart from metavariables and variables, templates may also mention specific functions, such as `++` in (6).

When we run RoughSpec on a signature of five list functions `++`, `reverse`, `map`, `sort` and `nub`, using the templates (1)–(3) above as well as commutativity, we get the following output:

```
Searching for commutativity properties...
  1. sort (xs ++ ys) = sort (ys ++ xs)
Searching for associativity properties...
  2. (xs ++ ys) ++ zs = xs ++ (ys ++ zs)
  3. sort (sort (xs ++ ys) ++ zs) =
     sort (xs ++ sort (ys ++ zs))
  4. nub (nub (xs ++ ys) ++ zs) =
     nub (xs ++ nub (ys ++ zs))
Searching for inverse function properties...
  5. reverse (reverse xs) = xs
Searching for distributivity properties...
  6. map f xs ++ map f ys = map f (xs ++ ys)
  7. sort (sort xs ++ sort ys) = sort (xs ++ ys)
  8. nub (nub xs ++ nub ys) = nub (xs ++ ys)
```

Each property is tagged with the name of the template that generated it. For example, the first law is an instance of commutativity, `?F X Y = ?F Y X`, with `?F = \xs ys -> sort (xs ++ ys)`. (Section 3.2 describes the strategy RoughSpec uses to instantiate metavariables.) We see that `++` is associative (2), that `reverse` is its own inverse (5), and that `map` distributes over `++` (6). We also see that composing `++` with `sort` or `nub` produces a binary operation which satisfies many properties in its own right.

By adding more templates, we can find more laws. For example, adding the template `?F (?G X) = ?G (?F X)` produces the law
`map f (reverse xs) = reverse (map f xs)`.
We have not found all of the important list laws (for example, the law `reverse (xs++ys) = reverse ys ++ reverse xs`), but have produced a useful and short subset.

The templates we have used so far represent well-known properties and apply to a wide range of APIs. The goal of RoughSpec is that the user can start with a "standard" set of templates, and find an incomplete, but useful set of properties for their program. Then they can find more detailed properties by adding templates that are tailored to their domain. By putting the user in charge of choosing templates, we aim to keep the output small and easy to understand.

Our current implementation of RoughSpec is built on top of QuickSpec [2], but the ideas are not QuickSpec-specific, and would work equally well on top of a different theory exploration system such as Speculate [23]. The only difference is that as Speculate also discovers inequalities and conditional laws, a RoughSpec built on top of Speculate would naturally support inequalities and conditions in templates.

Next, we describe how RoughSpec works, and evaluate it on some larger examples.

## 3.2 How it works

To use RoughSpec, the user inputs the templates they are interested in, along with the functions they want to explore, in a *signature* [2]. The user must also supply QuickCheck [5] test data generators for any non-standard types they want to test. Figure 3.1 shows an example of a simple signature, consisting of the functions `reverse`, `++` and `length`, and the template `?F (?G X Y) = ?F (?G Y X)`. Note that in our current implementation, functions are written uncurried, but for readability we write them curried in the main text of this article.

```
simpleSig = [
  con "reverse" (reverse :: [A] -> [A]),
  con "++" ((++) :: [A] -> [A] -> [A]),
  con "length" (length :: [A] -> Int),
  template "nest-commute" "?F(?G(X,Y))=?F(?G(Y,X))"
  ]
```

Figure 3.1: A signature containing some list functions and a template for properties about nested composition of functions being commutative in two variables.

As described in Section 3.1, the templates are expressed in a simple term language consisting of:

- metavariables, which represent *holes to be filled* with a function or constant symbol, and are written as a question mark followed by a name;
- variables, which are universally quantified, and are written as a name starting with a capital letter; and
- functions drawn from the signature.

Candidate properties are generated by attempting to fill the holes in a template using function symbols from the signature, making sure the generated equations are well typed. For our example template above, filling in the holes using the functions `length`, `reverse`, and `++` gives two candidate properties:

```
length (xs ++ ys) = length (ys ++ xs)        (cp1)
reverse (xs ++ ys) = reverse (ys ++ xs)      (cp2)
```

The generated candidate properties are then tested using QuickCheck [5]. If no counterexamples are found the property is presented to the user as a law. In our example, *cp*1 passes this phase and is presented to the user, while *cp*2 fails and is discarded.

### 3.2.1 Expanding templates

In the algorithm described above, each hole in a template can be filled only with precisely one of the function symbols in scope. This means that each template matches a rather narrow class of properties. As we shall see, many properties that we would intuitively consider to be in the same category are

not instances of the same template, so the user is forced to write many similar templates to capture a category of properties in full generality.

To help the user avoid the tedious work of typing up a set of nearly-identical templates, and to improve the expressiveness of the discovered properties, we have implemented some automated "expansion" of user input templates, which augments the user-provided templates by automatically adding variants of them.

#### 3.2.1.1 Nested functions

Consider the property

```
length (xs ++ ys) = length (ys ++ xs)
```
(*cp*1)

discovered in our example above. In that example, we discovered the property using a template that specifically described the composition of two function symbols being commutative. Suppose we had a more general template for commutativity, i.e. `?F X Y = ?F Y X`. What if we want such a template to cover properties like *cp*1, rather than having to type up a specialized "composition of two functions" commutativity template?

In order to do this we have implemented an extension allowing a hole to be filled by the composition of two function symbols. We replace a given hole in our template with two holes representing an outer function applied to an inner function applied to the original hole's arguments. That is, a hole of the form `?F e1...en` turns into `?G (?F e1...en)`. This allows us to discover the property *cp*1 using the commutativity template `?F X Y = ?F Y X`. It also allows us to use a general template for identity functions, `?F X = X`, to discover the property `reverse (reverse xs) = xs`, that is, that `reverse . reverse` is the identity function.

#### 3.2.1.2 Partial application

Suppose we extend our example signature from Figure 3.1 by adding the function `map` and a distributivity template

```
?F (?G X Y) = ?G (?F X) (?F Y),
```
(*d*1)

describing a function `?F` distributing over a two-argument function `?G`.

We would like to discover the property

```
map f (xs ++ ys) = map f xs ++ map f ys,
```
(*dmap*)

describing how `map` distributes over `++`. However, since our template holes can only be filled using precisely one function symbol (or, following Section 3.2.1.1, two nested function symbols), this template does not cover the desired property. Instead we would need a more complex template like

```
?F X (?G Y Z) = ?G (?F X Y) (?F X Z),
```

with an extra variable `X` for `map`'s first argument.

In order to avoid needing a variety of complicated templates when our signatures contain functions with varying numbers of arguments, we allow a template hole to be filled with a partially applied function. We replace a given

hole in our template with a hole applied to a number of fresh variables, limited by the maximum arity of the functions in scope. By doing so our desired property *dmap* is now covered by the template $d1$.

In combination with our nested function expansion described above, this also allows us to discover properties such as

```
map f (concat (xss ++ yss)) =
  map f (concat xss) ++ map f (concat yss),
```

by adding the `concat` function to our signature, using the same template $d1$.

This method considers all possible partially-applied functions when filling a hole. In practice we found this to give rise to some rather confusing properties when binary operators were involved. For instance, suppose we extend our example signature with a template `?F (?G X) = ?F X`, which describes pairs of functions `?F` and `?G` where the result of `?F` is preserved when we apply `?G` to its argument. This gives rise to nice properties such as

```
length (reverse xs) = length xs
length (map f xs) = length xs,
```

but also such properties as

```
length (xs ++ reverse ys) = length (xs ++ ys),
```

where the hole `?F` has been filled by the function `length . (xs ++)`.

We find properties about partially applied functions such as `xs ++` rather confusing and uninteresting, and therefore decided to restrict this expansion. Our restriction is that if a function is a binary operator (specifically, it has two arguments, both of which have the same type) we do not allow a partial application of it to fill a hole.

#### 3.2.1.3 Restricting expansion

Expanding templates automatically is a delicate balance. In moderation, it produces interesting properties that users want to see, and that intuitively match the given template. If we expand templates too much, we may generate irrelevant properties, overwhelm the user with output or increase the running time of our tool. As can be seen from the special treatment of binary operators in Section 3.2.1.2, we have implemented some ad hoc restrictions to our expansion heuristics to prevent them from producing properties we found less interesting. In general, which expansions are appropriate and how they should be used seems to depend on the context, the kinds of functions being explored and the user's priorities. In the future, we plan to give the user more control over the expansion process by making the language for describing templates more expressive. For example, the user should be able to specify which functions they are comfortable seeing partially applied.

### 3.2.2 Pruning

There is one last ingredient in RoughSpec's algorithm. Suppose we run the algorithm so far on a signature consisting of the functions `reverse`, `++`, `length` and `map`, and two templates from earlier:

- `?F X = X`, expressing that `?F` is the identity function.
- `?F (?G X) = ?F X`, expressing that the result of `?F` is unchanged when we apply `?G` to its argument.

We are presented with the following output:

```
Searching for identity properties...
 1. reverse (reverse xs) = xs
Searching for preserve properties...
 2. length (reverse xs) = length xs
 3. length (map f xs) = length xs
 4. length (reverse (reverse xs)) = length (reverse xs)
 5. length (reverse (map f xs)) = length (reverse xs)
 6. length (map f (reverse xs)) = length (map f xs)
 7. length (map f (map g xs)) = length (map f xs)
 8. reverse (reverse (reverse xs)) = reverse xs
 9. (++) (reverse (reverse xs)) = (++) xs
10. length (reverse (reverse xs)) = length xs
11. length (reverse (map f xs)) = length xs
12. length (map f (reverse xs)) = length xs
13. length (map f (map g xs)) = length xs
14. map f (reverse (reverse xs)) = map f xs
```

Some of these properties appear to be redundant. For instance, property 4 is an instance of property 2, in which `xs` has been replaced by `reverse xs`. Surely our user isn't interested in seeing a property that's just a more specific instance of a previously discovered property?

To avoid cluttering the output with redundant properties, RoughSpec includes a *pruning* phase, which aims to remove discovered properties that are trivial consequences of earlier properties. The pruning phase removes a property if:

- It is an *instance* of an earlier property. In this case, properties 4 and 8 will be pruned away, as they are instances of properties 2 and 1, respectively.
- It can be obtained applying the same function to both sides of an earlier property. For example, property 10 can be obtained by applying the `length` function to both sides of property 1.

More formally, a property is removed if it is of the form $t\sigma = u\sigma$ or $C[t] = C[u]$, where $t = u$ is a previously-discovered property, $\sigma$ is a substitution and $C$ is a context. In our example, we will discard properties 4, 8, 9, 10, and 14, so that a total of 9 properties remain.

This still leaves us with some redundant properties. For example, property 5 follows by equational reasoning from properties 2 and 3. If we were also to prune away properties that follow by equational reasoning from earlier properties, we would be left with only the three properties 1–3 above. Unfortunately, it is not a good idea to remove all properties that follow by equational reasoning from earlier properties, because the removed properties are often *non-trivial* consequences of the discovered properties, and may well be interesting to the user. The problem is how to add some form of equational reasoning, to remove

more redundant properties, but without removing redundant properties too aggressively.

Here is our solution. Because the templates given to RoughSpec describe the exact shapes of properties that the user is interested in, we should be careful not to go too far in pruning away properties matching those desired shapes. We therefore distinguish between two kinds of properties, depending on whether *template expansion* (Section 3.2.1) was needed to discover the property:

- If the property was found without using expansion, we use the simple pruning algorithm described above, which removes a property if it is a trivial consequence of a single existing property.
- If the property was found using expansion, we use the more powerful pruning algorithm from QuickSpec [2], which removes the property if it follows by equational reasoning from the earlier properties.

For instance, property 11 is pruned away as it follows by equational reasoning from properties 2 and 3, and was generated from an expanded template. However, if we added the template `?F (?G (?H F X)) = ?F X` to our signature, then property 11 would precisely match an input template, so pruning by equational reasoning would be disabled for that property, and it would no longer be pruned away. On the other hand, property 4 will always be pruned away, even if we add a template exactly matching it, as it is a direct instance of property 1.

With this last improvement added, RoughSpec produces the following three properties, and all others are pruned away:

```
Searching for identity properties...
  1. reverse (reverse xs) = xs
Searching for preserve properties...
  2. length (reverse xs) = length xs
  3. length (map f xs) = length xs
```

As properties discovered earlier are used to prune away ones that are discovered later, the order in which the templates are input makes a difference to which properties we output. It is usually a good idea to start with smaller and/or more general templates and move on to larger and/or more specific ones, as smaller properties are more likely to be useful in pruning larger ones, but the user may also want to put the templates they find most relevant first.

### 3.2.3 Libraries of default templates

We have compiled a small set of default templates capturing very common properties which we found useful in our case studies:

```
identity: ?F X = X
fixpoint: ?F ?X = ?X
left-id-elem: ?F ?Y X = X
right-id-elem: ?F X ?Y = X
cancel: ?F (?G X) = ?F X
commutative: ?F X Y = ?F Y X
commuting-functions: ?F (?G X) = ?G (?F X)
```

```
distributivity: ?F (?G X Y) = ?G (?F X) (?F Y)
homomorphism: ?F (?G X) (?G Y) = ?G (?H X Y)
associativity: ?F (?F X Y) Z = ?F X (?F Y Z)
```

These capture standard algebraic properties, and can be imported into a signature as a means to quickly run a first pass of exploration on a new theory without having to define any templates yourself. The user can then, if need be, simply extend this set with more specific templates.

## 3.3 Case studies

The following examples demonstrate theory exploration using our template-based approach and discuss what kinds of templates we have found to be useful. We compare our results to theory exploration with QuickSpec on the same sets of functions. The code is available at `https://github.com/solrun/quickspec`, in the `template-examples/ifl2020` directory, along with detailed experiment output. All experiments described in this paper were performed on a ThinkPad X260 laptop with a 2.5GHz Intel i7-6500U processor and 16GB of RAM running 64-bit Linux.

**Runtimes and memory use** Exploring the large library of list functions in Section 3.3.3 took just under 8 minutes and reached a maximum heap residency of 700 MB. All other examples ran in under 20s with a maximum heap residency under 100 MB.

### 3.3.1 Pretty Printing

This case study shows how RoughSpec can be useful in understanding an unfamiliar library. Suppose we are using Hughes's pretty-printing library [56] for the first time. We are presented with an intimidating array of combinators:

```
empty :: Doc
text :: String -> Doc
nest :: Int -> Doc -> Doc
(<>) :: Doc -> Doc -> Doc
(<+>) :: Doc -> Doc -> Doc
($$) :: Doc -> Doc -> Doc
hcat :: [Doc] -> Doc
hsep :: [Doc] -> Doc
vcat :: [Doc] -> Doc
sep :: [Doc] -> Doc
fsep :: [Doc] -> Doc
```

The library documentation explains that `Doc` represents a pretty-printed document, `empty` is an empty document, `text` prints a string verbatim, and `nest` indents an entire document by a given number of spaces. The remaining functions combine multiple documents into one:

- `<>`, `<+>` and `$$` typeset two documents beside one another, beside one another with a space in between, or one above the other, respectively.

- `hcat`, `hsep` and `vcat` are variants of `<>`, `<+>` and `$$` that take a *list* of documents.
- `sep` and `fsep` choose whichever of `<+>` and `$$` gives the prettiest output.

We may now feel happy going off and writing some pretty printers. But there are still questions unanswered:

- What is the difference between `empty` and `text ""`?
- If I am indenting a multi-line document, should I apply `nest` to each line individually or to the whole document?
- Does it matter if I use `<>` or `hcat`, `<+>` or `hsep`, `$$` or `vcat`?
- Why is there no analogue of `<>` for `sep` and `fsep`?

These are the kinds of questions a formal specification of the pretty-printing library would answer. Let us see if RoughSpec can help us.

We start with the list of default templates from 3.2.3. We reproduce RoughSpec's output verbatim. It finds the following 46 laws:

```
Searching for identity properties...
  1. hcat (unit x) = x
  2. hsep (unit x) = x
  3. vcat (unit x) = x
  4. sep (unit x) = x
  5. fsep (unit x) = x
Searching for fixpoint properties...
 6. nest x empty = empty
 7. nest x (hcat []) = hcat []
 8. nest x (hsep []) = hsep []
 9. nest x (vcat []) = vcat []
 10. nest x (sep []) = sep []
 11. nest x (fsep []) = fsep []
Searching for left-id-elem properties...
 12. nest 0 x = x
 13. empty <> x = x
 14. empty <+> x = x
 15. empty $$ x = x
 16. hcat [] <> x = x
 17. hsep [] <> x = x
 18. vcat [] <> x = x
 19. sep [] <> x = x
 20. fsep [] <> x = x
 21. hcat [] <+> x = x
 22. hsep [] <+> x = x
 23. vcat [] <+> x = x
 24. sep [] <+> x = x
 25. fsep [] <+> x = x
 26. hcat [] $$ x = x
 27. hsep [] $$ x = x
```

```
 28. vcat [] $$ x = x
 29. sep [] $$ x = x
 30. fsep [] $$ x = x
Searching for right-id-elem properties...
 31. x <> empty = x
 32. x $$ empty = x
 33. x <+> empty = x
 34. x <> text [] = x
Searching for cancel properties...
 35. length (unit (nest x y)) = length (unit y)
Searching for commutative properties...
Searching for commuting-functions properties...
 36. nest x (nest y z) = nest y (nest x z)
Searching for distributivity properties...
 37. nest x (y <> z) = nest x y <> nest x z
 38. nest x (y $$ z) = nest x y $$ nest x z
 39. nest x (y <+> z) = nest x y <+> nest x z
Searching for homomorphism properties...
 40. text xs <> text ys = text (xs ++ ys)
 41. hcat xs <> hcat ys = hcat (xs ++ ys)
 42. vcat xs $$ vcat ys = vcat (xs ++ ys)
 43. hsep xs <+> hsep ys = hsep (xs ++ ys)
Searching for associative properties...
 44. (x <> y) <> z = x <> (y <> z)
 45. (x $$ y) $$ z = x $$ (y $$ z)
 46. (x <+> y) <+> z = x <+> (y <+> z)
```

Laws 7–11 are curious. They are all rather similar, and do not look very interesting. In fact, each of these laws contains a term (such as `hsep []` or `vcat []`) which is actually equal to `empty`. Once we know that, we see that these laws are trivial restatements of law 6. The same issue occurs with laws 16–30, which are trivial restatements of 13–15. The problem is that there was no template which allowed RoughSpec to discover laws such as `hsep [] = empty`.

We shall see an automatic fix for this problem in Section 3.4.1. For now, we fix it by adding the template `?F ?X = ?Y`. This template finds 12 laws, including `hsep [] = empty` and its companions, and now laws 7–11 as well as laws 16–30 are pruned away as they follow from 6 and 13–15 respectively. Law 35 is also pruned away as our new template finds the simpler and subsuming property `length (unit x) = length (unit y)` We are left with a total of 25 laws: 1–6, 12–15, 31–34, and 36–46 above.

Together, these laws answer most of the questions we posed above. The difference between `empty` and `text ""` is that `empty` acts as an identity for the other operators:

```
empty <>  x = x     x <>  empty = x
empty <+> x = x     x <+> empty = x
empty $$  x = x     x $$  empty = x
```

On the other hand, `text ""` mostly does not, only satisfying one identity law:

```
x <> text "" = x
```

If we want to find out why `text ""` is not an identity element, we can now use QuickCheck to find a revealing counterexample (or indeed read Hughes [56] for an explanation).

As for whether one should indent each line separately or the whole document at once, it doesn't matter, because `nest` distributes over `$$`:

```
nest x (y $$ z) = nest x y $$ nest x z
```

Another distributivity law tells us that we can freely choose to typeset a long string in one go, or split it up into smaller pieces:

```
text xs <> text ys = text (xs ++ ys)
```

The `<>`, `<+>` and `$$` operators are associative:

```
(x <> y) <> z   = x <> (y <> z)
(x <+> y) <+> z = x <+> (y <+> z)
(x $$ y) $$ z   = x $$ (y $$ z)
```

and `hcat`, `vcat` and `hsep` appear to be those operators folded over a list:

```
hcat xs <> hcat ys = hcat (xs ++ ys)
vcat xs $$ vcat ys = vcat (xs ++ ys)
hsep xs <+> hsep ys = hsep (xs ++ ys)
```

Therefore, it doesn't matter whether one uses e.g. `<>` or `hcat`—they are equivalent.

Associativity of course means that we can write e.g. `x <> y <> z` without worrying about bracketing. We might wonder whether the same applies to sequences of mixed operators, e.g. `x <> y <+> z`. To find out we can add another template:

```
mixed-associativity: ?G (?F X Y) Z = ?F X (?G Y Z)
  -- as infix: (X `?F` Y) `?G` Z = X `?F` (Y `?G` Z)
```

This reveals that, indeed, a whole host of expressions can be freely rebracketed:

```
nest x y <> z = nest x (y <> z)
(x $$ y) <> z = x $$ (y <> z)
(x <+> y) <> z = x <+> (y <> z)
nest x y <+> z = nest x (y <+> z)
(x <> y) <+> z = x <> (y <+> z)
(x $$ y) <+> z = x $$ (y <+> z)
```

Finally, we come to the question of why there is no two-argument version of `sep` and `fsep`. Given what we learnt above, we might suspect that these operators are not associative. To test this, we can add two new functions to the signature:

```
sep2, fsep2 :: Doc -> Doc -> Doc
sep2 x y = sep [x, y]
fsep2 x y = fsep [x, y]
```

Indeed, no new associativity law appears.[2] Nor is it the case that e.g. `fsep2 (fsep xs) (fsep ys) = fsep (xs ++ ys)`. In fact, no interesting laws of any kind appear.

The laws that `hsep` and family satisfy are very useful when programming. When we want to typeset a list of documents horizontally, we can either use `hsep`, `<+>` or a mixture (e.g. we may write `hsep xs <+> hsep ys` instead of `hsep (xs++ys)`). By contrast, when using `sep` or `fsep`, we must carefully collect all documents into a list and only then apply the combinator. In this case, the *lack of a nice specification* is itself useful information: it warns us that we should take care when using these combinators!

**Summary** RoughSpec performed well on the pretty-printing library. It produced a manageable number of equations, all of them simple and easily understood. Despite their simplicity, they answered important questions about how to use the library—the questions listed at the top of this section. We believe that even simple properties, such as associativity and distributivity laws, are a great help in understanding how to use a new library. Finally, we got good results from a "standard" set of templates and were able to improve the output by adding our own.

The one hiccup in RoughSpec's performance was laws 7–11 and 16–30. We were forced to add a template specifically to prune away these laws. In fact, another instance of the same problem occurred: `sep` and `fsep` only differ on lists of at least three elements, which means that `sep2 = fsep2`. QuickSpec discovers this law instantly, but RoughSpec failed to find it as there was no template of the form `?X = ?Y`. Instead, laws about this function appear twice—once with `sep2` and once with `fsep2`.

In both cases, we have two laws containing syntactically different terms *that are actually equal*—for example, `hcat []` and `hsep []`. RoughSpec ought to detect that the terms are equal, and avoid generating duplicate laws. One option is to gather all the terms used to instantiate metavariables, divide them into equivalence classes by testing, and keep only the representative of each equivalence class. Section 3.4 describes an alternative solution to this problem.

**Comparison with QuickSpec** As reported in [2], QuickSpec does well given the combinators `text`, `nest`, `<>`, `<+>` and `$$`, finding a complete specification that matches the one given by Hughes [56]. Unfortunately, when we add `hcat` and friends, QuickSpec finds many complicated, unimportant-looking laws, for example:

```
40. fsep (xs ++ [empty] ++ ys) = fsep (xs ++ ys)
41. hcat (xs ++ [empty] ++ ys) = hcat (xs ++ ys)
42. hsep (xs ++ [empty] ++ ys) = hsep (xs ++ ys)
43. hcat (xs ++ [hcat ys] ++ zs) = hcat (xs ++ ys ++ zs)
44. hsep (xs ++ [hsep ys] ++ zs) = hsep (xs ++ ys ++ zs)
45. fsep (xs ++ [x $$ (y $$ z)]) = fsep xs $$ (x $$ (y $$ z))
46. fsep (xs ++ [x $$ x] ++ ys) = fsep xs $$ ((x $$ x) $$ fsep ys)
```

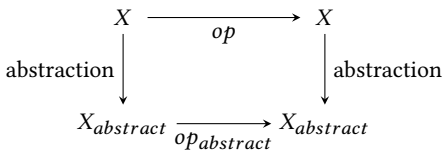
[2] Exercise for the reader: reading the documentation of the `pretty` library, it seems reasonable that `fsep2` could be associative. Why is it not?

What's more, QuickSpec now takes several minutes to run, while RoughSpec takes 15 seconds. In short, RoughSpec copes much better than QuickSpec once the combinators no longer have a simple specification.

### 3.3.2 Model-based properties

In [57], Hughes compared five different methods of defining properties for QuickCheck testing, and found the most effective to be *model-based* testing, which revealed all the bugs in his test programs and required only a small number of properties to be written.

Model-based testing is based on the approach to proving the correctness of data representations introduced by Hoare in [58]. The data representation is related to an appropriate abstract representation using an *abstraction function*. For each operation $op : X \to X$, an abstract implementation $op_{abstract} : X_{abstract} \to X_{abstract}$ is defined, and the following diagram is proven to commute:




The correctness of the data representation and operations in question then follows from the (presumably simpler) correctness proofs for the abstract data and operations.

In model-based testing, we define the same kind of abstract model of the data structure being tested, then *test* that the above diagram commutes. In [57], bugs in the implementation of concrete operations are found to cause counterexamples to such properties.

Since we can include specific function symbols from the exploration scope in our templates, we can use RoughSpec to search only for properties that relate two operations via a given abstraction function, with a template along the lines of:

```
?F (abstraction X) = abstraction (?G X).
```

#### 3.3.2.1 Binary search trees

In [57], Hughes uses binary search trees as an example and defines five model-based properties relating the tree operations to operations on a list of key-value pairs with *toList* as an abstraction function.

```
1. find x t = findList x (toList t)
2. insertList x (toList t) = toList (insert x t)
3. deleteKeyList x (toList t) = toList (delete x t)
4. toList nil = []
5. toList (union t t1) =
   sort (unionList (toList t) (toList t1))
```

Running RoughSpec on a signature containing the relevant functions and three templates describing model-based properties, we discover precisely these five properties and two additional properties, shown below, in just over 0.3 seconds.

```
6. toList (delete x nil) = []
7. toList nil = sort []
```

Note that properties 6 and 7 are both equivalent to property 4 above. However, since RoughSpec only searches for properties matching the given templates and nothing else, it won't discover that `sort [] = []` or that `delete x nil = nil`, even though these are properties a human might find rather trivial and obvious and would want the pruner to know about. This can be improved by running QuickSpec up to a small term size to provide background information, as discussed in Section 3.4.

Due to the different shapes of the desired properties we need three different templates to discover them all.

```
?F Y (toList X) = ?G Y X
toList ?X = ?Y
toList (?G X Y) = ?F (toList X) (toList Y)
```

With a more expressive template language, as discussed in Section 3.2.1, we could manage with fewer, more general templates. For example, all three templates are instances of the general shape
`toList (?F X1...Xn) = ?G (toList X1) ... (toList Xn)`,
which captures the properties used in model-based testing.

**Comparison with QuickSpec** QuickSpec discovers 28 properties about the functions in our signature, among them the five model-based properties. This takes between 10 and 11 seconds, significantly longer than RoughSpec's 0.3 seconds.

### 3.3.3 A large library of list functions

Section 4.2 of [2] describes a stress-test where QuickSpec was used to find properties about a set of 33 Haskell functions on lists. This took standard QuickSpec 42 minutes and resulted in 398 properties when limited to terms of size 7 or less, and hit a time limit of 2 hours when the size was increased to 8. As described in the Introduction, many of the laws found by QuickSpec were not interesting. This illustrates how running QuickSpec on larger theories scales poorly with regard to run-time and may produce an overwhelming amount of output. When we ran the most recent version of QuickSpec on this set of functions it ran out of memory and did not manage to produce any properties.

In contrast, running RoughSpec on this set of functions we can tailor the templates we use to properties we are interested in discovering and produce a more manageable amount of output in a much shorter time. The list of functions is shown in Figure 3.2. The last six functions are declared as *background* functions. Background functions may appear in properties, but a discovered property must contain at least one non-background function.

```
length    :: [A] -> Int
sort      :: [Int] -> [Int]
scanr     :: (A -> B -> B) -> B -> [A] -> [B]
(>>=)     :: [A] -> (A -> [B]) -> [B]
reverse   :: [A] -> [A]
(>=>)     :: (A -> [B]) -> (B -> [C]) -> A -> [C]
(:)       :: A -> [A] -> [A]
break     :: (A -> Bool) -> [A] -> ([A], [A])
filter    :: (A -> Bool) -> [A] -> [A]
scanl     :: (B -> A -> B) -> B -> [A] -> [B]
zipWith   :: (A -> B -> C) -> [A] -> [B] -> [C]
concat    :: [[A]] -> [A]
zip       :: [A] -> [B] -> [(A, B)]
usort     :: [Int] -> [Int]
sum       :: [Int] -> Int
(++)      :: [A] -> [A] -> [A]
map       :: (A -> A) -> [A] -> [A]
foldl     :: (A -> A -> A) -> A -> [A] -> A
takeWhile :: (A -> Bool) -> [A] -> [A]
foldr     :: (A -> A -> A) -> A -> [A] -> A
drop      :: Int -> [A] -> [A]
dropWhile :: (A -> Bool) -> [A] -> [A]
span      :: (A -> Bool) -> [A] -> ([A], [A])
unzip     :: [(A, B)] -> ([A], [B])
[]        :: [A]
partition :: (A -> Bool) -> [A] -> ([A], [A])
take      :: Int -> [A] -> [A]

Background functions:
(,)  :: A -> B -> (A, B)
fst  :: (A, B) -> A
snd  :: (A, B) -> B
(+)  :: Int -> Int -> Int
0    :: Int
succ :: Int -> Int
```

Figure 3.2: A library of list functions.

Running RoughSpec on this set of functions with the library of 10 default templates presented in Section 3.2.3, we discover 184 properties in just under 8 minutes. The properties include many useful laws, such as distributivity-like properties:

```
length xs + length ys = length (xs ++ ys)
concat xss ++ concat yss = concat (xss ++ yss)
sum xs + sum ys = sum (xs ++ ys)
```

Template expansion results in more complex properties. The second property below has size 11, much larger than QuickSpec was able to discover:

```
take x (takeWhile p (zip xs ys)) =
  takeWhile p (zip (take x xs) (take x ys))
take x (zipWith f xs (zipWith g ys zs)) =
  zipWith f xs (zipWith g (take x ys) (take x zs))
```

These two properties are given as examples of distributivity (`take` is distributed over the rest of the expression). The user may not consider these laws interesting, which suggests that having a more expressive template language is important. Nonetheless, the laws discovered are better than those found by QuickSpec, and we are able to discover them in a fraction of the time. This demonstrates that RoughSpec is much better suited than QuickSpec to exploring large libraries of functions, and that it makes theory exploration tractable on such libraries that were previously infeasible to explore.

### 3.3.4 A window manager

The xmonad window manager [59] is a tiling window manager for X, written in Haskell. We take an implementation of a simple window manager with multiple virtual workspaces containing stacks of screens, in the style of xmonad, as shown in [60], and demonstrate how RoughSpec is useful for finding properties about this data structure.

The data structure `StackSet` represents a set of workspaces each containing a stack of screens, and there are a number of operators to construct and manipulate `StackSet`s, shown below:

```
empty   :: Natural -> StackSet a
current :: StackSet a -> Natural
view    :: Natural -> StackSet a -> StackSet a
peek    :: StackSet a -> a
rotate  :: Ordering -> StackSet a -> StackSet a
push    :: a -> StackSet a -> StackSet a
shift   :: Natural -> StackSet a -> StackSet a
insert  :: a -> Natural -> StackSet a -> StackSet a
delete  :: a -> StackSet a -> StackSet a
index   :: Natural -> StackSet a -> [a]
```

The operator `empty` creates a `StackSet` containing a given number of empty workspaces, `current` returns the current workspace, `view` sets the current workspace, `peek` extracts the screen on top of the current workspace's stack,

`rotate` cycles the current screen stack up or down, `push` inserts an screen on top of the current stack, `shift` moves the screen on top of the current stack to the top of stack $n$, `insert` inserts a screen on top of stack $n$, `delete` deletes a given screen, and `index` extracts the workspace at a given index.

Running RoughSpec with the set of default templates described in Section 3.2.3 we discover the following 23 properties:

```
Searching for identity properties...
Searching for fix-point properties...
  1. current (empty 0) = 0
  2. view x (empty 1) = empty 1
  3. rotate o (empty 1) = empty 1
  4. shift x (empty 1) = empty 1
  5. delete x (empty 1) = empty 1
Searching for left-id-elem properties...
  6. rotate EQ s = s
  7. current (empty x) + y = y
Searching for right-id-elem properties...
  8. peek (push x s) = x
Searching for cancel properties...
  9. current (rotate o s) = current s
 10. current (push x s) = current s
 11. current (shift x s) = current s
 12. current (delete x s) = current s
 13. current (insert x y s) = current s
 14. current (view x (rotate o s)) = current (view x s)
 15. current (view x (push y s)) = current (view x s)
 16. current (view x (shift y s)) = current (view x s)
 17. current (view x (delete y s)) = current (view x s)
 18. current (view x (insert y z s)) = current (view x s)
Searching for commutative properties...
Searching for op-commute properties...
 19. view x (delete y s) = delete y (view x s)
 20. rotate o (rotate o' s) = rotate o' (rotate o s)
 21. delete x (delete y s) = delete y (delete x s)
 22. view x (insert y z s) = insert y z (view x s)
 23. shift x (push y (view z s)) = view z (shift x (push y s))
Searching for 2-distributive properties...
Searching for homomorphism properties...
Searching for associative-3 properties...
```

We may note that our fix-point properties, 1–5, look rather specific, referring to `empty 0` and `empty 1`, and wonder whether they are perhaps valid for `empty n` for more values of $n$. Property 7, `current (empty x) + y = y`, implies that `current (empty x) = 0`, subsuming property 1, but in a verbose and unclear manner. However, the more interesting property `current (empty x) = 0` does not fit any of our templates. Testing with QuickCheck reveals that properties 3–5 hold for `empty n` for all values of $n > 0$, not just `empty 1`, while property 2 is specific to $n = 1$ (if and only if we have just one workspace in our set, the result of setting a given one of the workspaces

as the current one must always be the same). This demonstrates that even when RoughSpec does not generate the most general valid properties for our program, it can help us to gain a better understanding of the functions in our program and come up with more general properties to test and create a more complete specification. However, adding support for conditions such as $n > 0$ could enable us to generate more elegant and useful properties.

The properties shown above, discovered by RoughSpec using our list of default templates, provide us with some insight into how this window manager works. However, it clearly does not provide a complete specification and some operations hardly appear in the properties discovered. If we add a template `?F X = ?F (?F X)` describing idempotency we discover 15 further properties. The first 4, 24–27 shown below, tell us that `view`, `push`, `delete`, and `insert` are idempotent, which is interesting and useful information.

```
24. view x s = view x (view x s)
25. push x s = push x (push x s)
26. delete x s = delete x (delete x s)
27. insert x y s = insert x y (insert x y s)
```

Properties 28–39, however, describe the idempotency of various combinations of the previously mentioned functions (along with the function `shift` in the case of 31 and 33), for example:

```
33. shift x (push y s) =
      shift x (push y (shift x (push y s)))
38. insert x y (delete z s) =
      insert x y (delete z (insert x y (delete z s)))
```

These properties look rather complicated and uninteresting. They are generated due to our template expansions, and once again indicate that in some cases we should limit this expansion. As before, it may be useful to have a more descriptive template language where the user can indicate whether or not a given template should be expanded.

**Comparison with hand-written properties and QuickSpec** On this example, QuickSpec takes 12 seconds, while RoughSpec takes 7 seconds.

In [60] the author defines a set of 18 QuickCheck test properties forming a specification for this window manager. Of these 18, five are conditional and therefore cannot be found by our current implementation of RoughSpec.

The author defines an invariant for `StackSet`s stating that the current workspace index is within bounds (between 0 and the size of the set), and that the set does not contain any duplicate screens. They test that this invariant holds for all `StackSet`s generated by the QuickCheck generator (`invariant s`) and also that it still holds after applying various operations.

```
1. invariant x
2. invariant $ view n x
3. invariant $ rotate n x
4. invariant $ push n x
5. invariant $ delete n x
6. invariant $ shift n x
7. invariant $ insert n i x
```

Note that having discovered property 1 above, both RoughSpec and QuickSpec would prune away properties 2–6, deeming them redundant. However they can be useful as test properties when we don't trust the test data generator to provide good coverage of the range of potential values.

If we add the `invariant` function to our signature, QuickSpec will discover the property `invariant s`. Of the 13 equational properties defined in [60], QuickSpec discovers 4.

RoughSpec, on the other hand, discovers the equivalent but more clunky property `invariant s && x = x`, although it will discover `invariant s` if we add an invariant template `?F ?X = True`, which is perhaps a reasonable template to include when we have boolean-valued functions in our exploration scope. Three of the equational properties defined in [60] are found by RoughSpec with the idempotency template and one of the remaining ones is `current (empty n) = 0`, where RoughSpec discovered the equivalent but less elegant `current (empty x) + y = y`. The problem is similar to the one we encountered in the pretty-printing and binary search examples: RoughSpec instantiates the schema `?E + X = X` with both `?E = current (empty x)` and `?E = 0`, but does not notice that these two terms are equal.

## 3.4 A hybrid approach

RoughSpec and QuickSpec's approaches seem to be complementary. For large APIs, QuickSpec is slow, and often produces an overwhelming amount of output. By contrast, RoughSpec runs quickly, and produces a moderate number of laws. The laws it finds are easy to understand, because they follow standard patterns, and can be targeted to the user's interests.

On the other hand, RoughSpec does not usually find a complete specification. Even when testing lists, RoughSpec failed to find the law
`reverse (xs ++ ys) = reverse ys ++ reverse xs`,
as it did not match any of the provided templates. This is by design but is nonetheless a weakness.

There is another problem. If RoughSpec is given very general templates, our premise of limiting the search space may no longer hold. For example, consider a template `?F X = ?G X` searching for equivalent functions. This template could produce interesting and useful properties, for instance stating that different sorting functions produce the same output for a given input. However, if our signature contains many functions that have the same type we will produce a large number of candidate properties and testing them will take a long time (and probably most will be falsified). Meanwhile, QuickSpec will discover relevant properties of this shape much more quickly.

### 3.4.1 Hybrid tool

To solve these problems, this section introduces a *hybrid* approach. The idea is to run QuickSpec to find all *small* laws, running with a small size limit, and then run RoughSpec to find interesting laws beyond that size limit.

We have implemented a tool that combines QuickSpec and RoughSpec. The user can call `roughSpecWithQuickSpec k s` on their signature `s`, where `k` is an integer parameter specifying up to which term size QuickSpec should

go. This prompts QuickSpec to run on the signature up to the specified term size and print out the properties discovered. Then RoughSpec is run, using the templates supplied, but with knowledge of the properties discovered by QuickSpec. The pruner removes any properties subsequently discovered by RoughSpec that follow from those discovered by QuickSpec. This allows us to discover elegant and useful smaller properties using QuickSpec that would be cumbersome and time-consuming to find with RoughSpec as they would require a too-general template. It also helps us prune away redundant and uninteresting properties discovered by RoughSpec. Below we examine the output of the new hybrid tool on the examples we introduced in Section 3.3.

### 3.4.2 Pretty Printing

Our pretty-printing library example from Section 3.3.1 provides a good illustration of how combining QuickSpec and RoughSpec helps us find better properties.

There, RoughSpec found the undesired properties 7–11 and 16–30, which described semantically equivalent properties of various terms that were all equal to `empty`. In order to remove these properties, we were forced to add a template `?F ?X = ?Y`. This template has a very general shape and can hardly be said to describe a family of interesting properties. Very many true properties as well as falsifiable candidate properties are likely to match this template, so including it is likely to harm RoughSpec's performance. In the case of our pretty-printing signature, adding this template nearly doubles the runtime, taking it from 16 seconds with just the default templates to 28 seconds. However, running our hybrid tool on the pretty-printing signature with QuickSpec up to size 2, we get the same benefits but with a runtime of 14.5 seconds. QuickSpec discovers precisely the 5 laws stating that `hcat []`, `hsep []`, `vcat []`, `sep []` and `fsep []` are equal to `empty`, and properties 7–11 and 16–30 are pruned away. As a result, the number of properties discovered by RoughSpec goes down from 46 to 26.

### 3.4.3 Binary Search Trees

In the binary search tree example of Section 3.3.2, RoughSpec generated two redundant properties:

```
6. toList (delete x nil) = []
7. toList nil = sort []
```

These properties followed from the other discovered properties and the laws `delete x nil = nil` and `sort [] = []`, but these last two laws were not an instance of any of our templates so were not discovered by RoughSpec. This is the same problem that we encountered with the pretty-printing example of Section 3.3.1.

If we run our hybrid tool with QuickSpec up to term size 3, QuickSpec discovers, among other laws, that `sort [] = []` and `delete x nil = nil`, allowing RoughSpec to prune away its redundant properties. RoughSpec then discovers four properties: the first five from Section 3.3.2 minus `toList nil = []`, which is now discovered by QuickSpec. (It would be useful to report that this last

property matches one of the user-supplied templates, but this is future work.) RoughSpec's runtime goes up from 0.3 to 0.8 seconds, which still compares favourably to the 11 seconds needed by QuickSpec. Here we pay a small price in runtime to obtain better quality output.

### 3.4.4 List library

We performed two experiments using the combined tool on the large list library from Section 3.3.3, allowing QuickSpec to explore terms of size up to 2 and 3 respectively.

When we ran QuickSpec up to term size 2, QuickSpec discovered 7 properties, some of which would otherwise have been discovered by RoughSpec but a few of which RoughSpec had previously missed, for instance `length [] = 0`, `sum [] = 0` (as those don't fit any of our templates), and `concat [] = []` (which RoughSpec's fix-point template does not discover due to the empty list having a different type on the left hand side of the equation than the right hand side). These properties are useful for pruning away some of the redundant properties discovered by RoughSpec, such as `length [] + x = x`. The hybrid tool now discovers 177 properties instead of the previous 184, with 4 additional properties being discovered by QuickSpec instead, and 3 being pruned away. However, even running QuickSpec up to a small term size takes a significant amount of time on this set of functions due to the amount of terms of different types QuickSpec generates and enumerates, and our runtime went from 7.3 minutes to 8.6 minutes.

When we ran QuickSpec up to term size 3, QuickSpec discovered 43 properties and RoughSpec 146 (compared to the original 184) and the runtime was 8.75 minutes. Since adding QuickSpec to the mix increases the runtime quite a bit and (while QuickSpec also discovers a number of properties RoughSpec would have found anyway), our combination of tools does not provide the improvements we hoped for in this case. It seems that the benefits of using QuickSpec decrease as the number of different functions in the signature increase.

Alternatively, running QuickSpec only on the background functions or a limited subset of the library functions could be more helpful to make our pruner smarter without taking too much time.

### 3.4.5 Window manager

When we run our hybrid tool with QuickSpec set up to term size 3, QuickSpec discovers two properties:

```
1. rotate EQ s = s
2. current (empty x) = 0
```

This allows RoughSpec to prune away the less elegant properties `current (empty 0) = 0` and `current (empty x) + y = y` (property 1 would have been discovered by RoughSpec anyway), otherwise producing the same output as previously and taking 7 seconds instead of the previous 6.5. When we add the `invariant` function as described in Section 3.3.4, QuickSpec also discovers the property `invariant s`, allowing RoughSpec to prune away the equivalent but less nice `invariant s && x = x`.

### 3.4.6 Summary

Combining QuickSpec and RoughSpec reduces the number of templates needed to discover small, basic properties. These properties are often useful for pruning away other, larger (and to a human, less elegant) properties. What's more, the combination solves the problem we repeatedly encountered in Section 3.3, where RoughSpec produced redundant laws as a result of failing to notice that two terms are equal. There is often a small overhead in runtime compared to using RoughSpec on its own, which is to be expected, as the search space is bigger when we allow QuickSpec to explore small terms. The larger the signature is, the bigger this overhead is. Still, the combination of QuickSpec and RoughSpec often produces higher-quality laws than either tool manages on its own.

## 3.5 Related work

Apart from QuickSpec [2] and Speculate [23], which we described in the Introduction, there are also theory exploration tools for mathematics [16,18,19]. Below we describe several which support templates or schemas:

- Buchberger [14] introduced the idea of schema-based theory exploration and his team implemented it in the Theorema [15] system. Theorema provides tools to assist the user in their theory exploration but does not automate the process. The user must provide the schemas (but can store them in a schema library for easier reuse), manually perform substitutions to instantiate the schemas with terms, and conduct proofs interactively.

- IsaScheme [19] is a schema-based theory exploration system for Isabelle/HOL. Users provide the schemas as well as a set of terms to instantiate the schemas with, but the instantiation is performed automatically. The conjectures generated by instantiation are then automatically refuted using Isabelle/HOL's counter-example finders, or proved using the IsaPlanner [20, 21] prover.

- MATHsAiD [16] is an automated theorem-discovery tool which has mainly been applied in the context of abstract algebra. It uses a combination of several exploration techniques, one of them being schema instantiation, which is used for a limited set of lemmas/theorems. The schemas used by MATHsAiD are predefined and built-in to the system and include, for example, reflexivity and transitivity.

None of the above systems has the same flexibility in combining scheme-based and free-form equational exploration as our system. We allow the user to customize the templates used, as well as tune exploration to decide which part of the search space should be explored thoroughly by QuickSpec, and which parts should be explored by a template-based strategy.

Very recently, there has been interest in applying neural networks and techniques developed for natural language processing also for tasks in automated theorem proving. Rabe et al. [61], present an experiment using a language model for various mathematical reasoning tasks, including the generation of new conjectures. They evaluate their tool on tasks such as generating a missing

precondition for a given conditional statement, generating the opposite side of an equation, given one side, as well as "free-form" conjecturing. While the model does produce some true statements, the majority these are not new, but for instance exact copies or alpha renamings of statements from the training set. This is of course a very different approach compared to our work, as our tool is designed to produce only statements that are apparently true (by means of testing), and only statements that are *different* from what has been previously seen, ensured by including simple equational reasoning in the generation process.

## 3.6 Future work

There are many avenues of future work we would like to explore.

RoughSpec currently supports only equations as templates. We would like to allow templates to be arbitrary formulas, such as a template for monotonicity, `X <= Y ==> ?F X <= ?F Y`. There is no deep reason why RoughSpec only supports equations—only that it re-uses code from QuickSpec [2], which itself only supports equations. We plan to lift this restriction, which is not hard but requires a bit of engineering.

We are currently extending RoughSpec to discover conditional equations. In our approach, the user specifies a set of equational templates and a set of condition templates, and the tool discovers which combinations of conditions make each equation true. That is, the user does not have to know the exact condition for each template, but only which conditions are interesting. The main idea is that given a candidate equation and a large set of candidate conditions, we can test if *all conditions together* imply the equation; if they do, we can remove conditions one by one and thus obtain a minimal set of conditions. An important question would be whether we can supply a useful set of "standard" condition templates, as we do for equations.

In the experiments described in this paper we have used hand-written templates provided by the user or by a library of default templates. We would like to further explore using data-driven methods to learn good templates for a given context. We will investigate using machine learning to extract common patterns from proof libraries, learning common lemma shapes given properties of the theorem we want to prove (c.f. [62]), as well as exploiting type-class laws and other algebraic properties.

QuickSpec has been used to discover lemmas in a theorem proving context [4], and we believe our extension could also be useful here, using templates relevant for the theorem we would like to prove. We will also investigate extracting templates from failed proof attempts, similar to critics in proof planning [63].

We currently use a set of heuristics to expand templates. Template expansion is important in order to captures a wide variety of laws, but it sometimes goes too far. For example, given the template `?F (?G X) = ?G (?F X)`, both `?F` and `?G` can be replaced by a nested function, resulting in laws of the form `f (g (h (i x))) = h (i (f (g x)))`. To reduce the use of heuristics, we would like to define an *expressive* template language, in which the user can say precisely what sort of laws they want, for example, to forbid the use of nested functions in the template above, or alternatively allow for expansion as iterative

deepening, up to a specified limit. As another example, it should be possible to define a template that captures a general distributivity law `f (g x1) (g x2)...(g xn) = g (f x1...xn)` for $n$-ary functions, without specialising it to a particular $n$. Doing so requires designing a small set of combinators for building templates.

Our tool could be made more user-friendly by not requiring the user to explicitly type up a signature. A default signature for a given set of functions could be automatically generated using Template Haskell.

## 3.7 Conclusion

We have presented RoughSpec, a theory exploration tool in which the user specifies which kinds of properties are interesting. It generates specifications which are short, and easy to understand, but not necessarily complete. It can be used both to produce a rough specification of how a set of functions behaves, and to target specific families of laws that the user is interested in. It also scales well to large APIs. It is complementary to QuickSpec, which uses a complete strategy for conjecture generation.

By combining the two tools, we get the benefits of both: QuickSpec deals with smaller conjectures (of which there are fewer), while RoughSpec handles larger conjectures, resulting in a system that is both fast and outputs a manageable number of largely interesting properties.

**Acknowledgements** This work was partially supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP), funded by the Knut and Alice Wallenberg Foundation, and by the Swedish Research Council (VR) grants 2016-06204, *Systematic testing of cyber-physical systems (SyTeC)* and 2014-04849, *Learning and exploration in automated reasoning*.

# Bibliography


[1] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

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

[3] M. Johansson, D. Rosén, N. Smallbone, and K. Claessen, "Hipster: Integrating theory exploration in a proof assistant," in *Proceedings of CICM*. Springer, 2014, pp. 108–122.

[4] M. Johansson, "Automated theory exploration for interactive theorem proving: An introduction to the Hipster system," in *Proceedings of ITP*, ser. LNCS, vol. 10499. Springer, 2017, pp. 1–11.

[5] K. Claessen and J. Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs," in *Proceedings of ICFP*, 2000, pp. 268–279.

[6] A. Newell and H. Simon, "The logic theory machine–a complex information processing system," *IRE Transactions on Information Theory*, vol. 2, no. 3, pp. 61–79, 1956.

[7] A. N. Whitehead and B. A. W. Russell, *Principia mathematica; 2nd ed.* Cambridge: Cambridge Univ. Press, 1927. [Online]. Available: https://cds.cern.ch/record/268025

[8] H. A. Simon and A. Newell, "Heuristic problem solving: The next advance in operations research," *Operations Research*, vol. 6, no. 1, pp. 1–10, 1958. [Online]. Available: http://www.jstor.org/stable/167397

[9] M. Johansson and N. Smallbone, "Conjectures, tests and proofs: An overview of theory exploration," in *Proceedings of the 9th International Workshop on Verification and Program Transformation, VPT@ETAPS 2021, Luxembourg, Luxembourg, 27th and 28th of March 2021*, ser. EPTCS, A. Lisitsa and A. P. Nemytykh, Eds., vol. 341, 2021, pp. 1–16. [Online]. Available: https://doi.org/10.4204/EPTCS.341.1

[10] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone, "Automating inductive proofs using theory exploration," in *Proceedings of CADE*, ser. LNCS, vol. 7898. Springer, 2013, pp. 392–406.

[11] D. B. Lenat, "AM, an artificial intelligence approach to discovery in mathematics as heuristic search," Ph.D. dissertation, Stanford University, 1976.

[12] S. Colton, "The HR program for theorem generation," in *Automated Deduction—CADE-18*, A. Voronkov, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 285–289.

[13] ——, "Refactorable numbers - a machine invention," *Journal of Integer Sequences*, vol. 2, 1999. [Online]. Available: https://cs.uwaterloo.ca/journals/JIS/colton/joisol.html

[14] B. Buchberger, "Theory exploration with Theorema," *Analele Universitatii Din Timisoara, ser. Matematica-Informatica*, vol. 38, no. 2, pp. 9–32, 2000.

[15] B. Buchberger, A. Craciun, T. Jebelean, L. Kovács, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger, "Theorema: Towards computer-aided mathematical theory exploration," *Journal of Applied Logic*, vol. 4, pp. 470–504, 12 2006.

[16] R. L. McCasland, A. Bundy, and P. F. Smith, "MATHsAiD: Automated mathematical theory exploration," *Applied Intelligence*, Jun 2017. [Online]. Available: https://doi.org/10.1007/s10489-017-0954-8

[17] S. Fajtlowicz, "On conjectures of graffiti," *Discrete Math.*, vol. 72, no. 1–3, p. 113–118, dec 1988. [Online]. Available: https://doi.org/10.1016/0012-365X(88)90199-9

[18] M. Johansson, L. Dixon, and A. Bundy, "Conjecture synthesis for inductive theories," *Journal of Automated Reasoning*, vol. 47, no. 3, pp. 251–289, Oct 2011. [Online]. Available: https://doi.org/10.1007/s10817-010-9193-y

[19] 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.

[20] L. Dixon and J. D. Fleuriot, "Isaplanner: A prototype proof planner in isabelle," in *Automated Deduction – CADE-19*, vol. 2741. Springer Berlin Heidelberg, 07 2003, pp. 279–283.

[21] L. Dixon and M. Johansson, "Isaplanner 2: A proof planner for isabelle," 2007.

[22] E. Singher and S. Itzhaky, "Theory exploration powered by deductive synthesis," in *Computer Aided Verification*, A. Silva and K. R. M. Leino, Eds. Cham: Springer International Publishing, 2021, pp. 125–148.

[23] R. Braquehais and C. Runciman, "Speculate: discovering conditional equations and inequalities about black-box functions by reasoning from test results," in *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, 2017, pp. 40–51.

[24] J. Urban and J. Jakubův, "First neural conjecturing datasets and experiments," in *Intelligent Computer Mathematics*, C. Benzmüller and B. Miller, Eds. Cham: Springer International Publishing, 2020, pp. 315–323.

[25] M. N. Rabe, D. Lee, K. Bansal, and C. Szegedy, "Mathematical reasoning via self-supervised skip-tree training," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=YmqAnY0CMEy

[26] A. Davies, P. Veličković, L. Buesing, S. Blackwell, D. Zheng, N. Tomašev, R. Tanburn, P. Battaglia, C. Blundell, A. Juhász, M. Lackenby, G. Williamson, D. Hassabis, and P. Kohli, "Advancing mathematics by guiding human intuition with AI," *Nature*, vol. 600, no. 7887, pp. 70–74, Dec 2021. [Online]. Available: https://doi.org/10.1038/s41586-021-04086-x

[27] E. Davis, "Deep learning and mathematical intuition: A review of (Davies et al. 2021)," 2021.

[28] K. Spalding. (2021) Researchers create AI that can invent brand new math theorems. [Online]. Available: https://www.iflscience.com/editors-blog/researchers-create-ai-that-can-invent-brand-new-math-theorems/

[29] D. Nield. (2021) AI is discovering patterns in pure mathematics that have never been seen before. [Online]. Available: https://www.sciencealert.com/ai-is-discovering-patterns-in-pure-mathematics-that-have-never-been-seen-before

[30] I. L. Valbuena and M. Johansson, "Conditional lemma discovery and recursion induction in Hipster," *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, vol. 72, 2015.

[31] "IMO grand challenge." [Online]. Available: https://imo-grand-challenge.github.io/

[32] S. Polu, J. M. Han, K. Zheng, M. Baksys, I. Babuschkin, and I. Sutskever. (2022) Formal mathematics statement curriculum learning. [Online]. Available: https://cdn.openai.com/papers/Formal_Mathematics_Statement_Curriculum_Learning__ICML_2022.pdf

[33] K. Zheng, J. M. Han, and S. Polu, "Minif2f: a cross-system benchmark for formal olympiad-level mathematics," 2021.

[34] L. Lampropoulos, D. Gallois-Wong, C. Hriţcu, J. Hughes, B. C. Pierce, and L.-y. Xia, "Beginner's luck: A language for property-based generators," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 114–129. [Online]. Available: https://doi.org/10.1145/3009837.3009868

[35] A. Mista, A. Russo, and J. Hughes, "Branching processes for QuickCheck generators," in *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, ser. Haskell 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–13. [Online]. Available: https://doi.org/10.1145/3242744.3242747

[36] R. Milner, *Communication and Concurrency*. Prentice-Hall, Inc., 1989.

[37] K. Nakata and T. Uustalu, "A Hoare logic for the coinductive trace-based big-step semantics of While," in *Programming Languages and Systems*. Springer, 2010, pp. 488–506.

[38] D. A. Turner, "Total functional programming," *J. UCS*, vol. 10, no. 7, pp. 751–768, 2004.

[39] J. C. Blanchette, J. Hölzl, A. Lochbihler, L. Panny, A. Popescu, and D. Traytel, "Truly modular (co)datatypes for Isabelle/HOL," in *Proceedings of ITP*, G. Klein and R. Gamboa, Eds. Springer International Publishing, 2014, pp. 93–110.

[40] 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)*, June 2017, pp. 1–12.

[41] A. Abel and B. Pientka, "Well-founded recursion with copatterns and sized types," *J. Funct. Program.*, vol. 26, p. e2, 2016. [Online]. Available: https://doi.org/10.1017/S0956796816000022

[42] 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.

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

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

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

[46] 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.

[47] R. Bird, *Introduction to Functional Programming*, 2nd ed. Pearson Education, 1998.

[48] G. Hutton and J. Gibbons, "The generic approximation lemma," *Information Processing Letters*, vol. 79, p. 2001, 2001.

[49] D. Pous, "Coinduction all the way up," in *Proceedings of LICS*. New York, NY, USA: ACM, 2016, pp. 307–316. [Online]. Available: http://doi.acm.org/10.1145/2933575.2934564

[50] J. C. Blanchette, A. Bouzy, A. Lochbihler, A. Popescu, and D. Traytel, "Friends with benefits," in *Proceedings of ESOP 2017*. Springer, 2017, pp. 111–140. [Online]. Available: https://doi.org/10.1007/978-3-662-54434-1_5

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

[52] R. Hinze, "Concrete stream calculus: An extended study," *J. Funct. Program.*, vol. 20, no. 5-6, pp. 463–535, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1017/S0956796810000213

[53] R. Leino and M. Moskal, "Co-induction simply: Automatic co-inductive proofs in a program verifier," Microsoft Research, Tech. Rep., July 2013. [Online]. Available: https://www.microsoft.com/en-us/research/publication/co-induction-simply-automatic-co-inductive-proofs-in-a-program-verifier/

[54] D. Lucanu, E.-I. Goriac, G. Caltais, and G. Roşu, "CIRC: A behavioral verification tool based on circular coinduction," in *Proceedings of CALCO 2009*. Springer, 2009, pp. 433–442. [Online]. Available: https://doi.org/10.1007/978-3-642-03741-2_30

[55] C. Runciman, M. Naylor, and F. Lindblad, "SmallCheck and Lazy Small-Check: automatic exhaustive testing for small values," in *Proceedings of the first ACM SIGPLAN symposium on Haskell*, 2008, pp. 37–48.

[56] J. Hughes, "The Design of a Pretty-printing Library," in *Advanced Functional Programming*, J. Jeuring and E. Meijer, Eds. Springer Verlag, LNCS 925, 1995, pp. 53–96.

[57] ——, "How to specify it," in *Trends in Functional Programming*, W. J. Bowman and R. Garcia, Eds. Cham: Springer International Publishing, 2020, pp. 58–83.

[58] C. A. R. Hoare, "Proof of correctness of data representations," in *Language Hierarchies and Interfaces*, F. L. Bauer, E. W. Dijkstra, A. Ershov, M. Griffiths, C. A. R. Hoare, W. A. Wulf, and K. Samelson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1976, pp. 183–193.

[59] D. Stewart and S. Sjanssen, "Xmonad," in *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, ser. Haskell '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 119. [Online]. Available: https://doi.org/10.1145/1291201.1291218

[60] D. Stewart, "Roll your own window manager: Part 1: Defining and testing a model," May 2007. [Online]. Available: https://donsbot.wordpress.com/2007/05/01/roll-your-own-window-manager-part-1-defining-and-testing-a-model/

[61] M. N. Rabe, D. Lee, K. Bansal, and C. Szegedy, "Mathematical reasoning via self-supervised skip-tree training," 2020.

[62] J. Heras, E. Komendantskaya, M. Johansson, and E. Maclean, "Proof-pattern recognition and lemma discovery in ACL2," in *Proceedings of LPAR*, 2013.

[63] A. Ireland and A. Bundy, "Productive use of failure in inductive proof," *Journal of Automated Reasoning*, vol. 16, pp. 79–111, 1996.