CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Hopster: Automated discovery of mathematical properties in HOL

Master's thesis in Computer Science – algorithms, languages and logic

JUAN RICART

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Hopster: Automated discovery of mathematical properties in HOL

JUAN RICART

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2019

Hopster: Automated discovery of mathematical properties in HOL
JUAN RICART

Supervisor: Nicholas Smallbone, Department of Computer Science and Engineering
Examiner: Carl-Johan Seger, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Hopster: Automated discovery of mathematical properties in HOL
JUAN RICART
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

A new tactic for HOL is proposed that is suitable for proving equational properties of functional programs. HOL has numerous tactics that automate the process of proving a logical statement. Yet none incorporate the discovery of useful lemmas in their process, an essential activity when performing a manual proof.
The Hopster tactic is able to discover new lemmas via a technique known as theory exploration, implemented in a tool called QuickSpec. So far, theory exploration has been applied with increasing success for generating many useful properties when given a theory of considerable size. A dual to this scenario occurs when the theory is fairly well-developed and the user is interested in proving a specific new property. This work pays particular attention to the second scenario, when the user wishes to prove a specific goal using theory exploration. The tactic has been designed to support this case by automating the selection of which functions to explore. The design and testing of this selection process are the principal contributions of this work. The effectiveness and efficiency of the implementation are measured on a set of test cases to identify where the tactic performs well and where further improvements remain. It is hoped that Hopster serves as a useful tool that supports the development of verified functional programs.

Keywords: theory exploration, automated theorem proving, hopster, hol, quickspec

Acknowledgments

Anything of value in these pages is most certainly due to my supervisor Nick Smallbone. I can only take full credit for the mistakes that remain. I am grateful for the many hours spent together making this work a great learning experience for me.
Thanks to Magnus Myreen for his help and guidance during the preparation of the proposal for this thesis. Thank you for the sessions spent at your office where I picked many tricks in the use of the HOL4 proof assistant.
I am grateful to my examiner Carl Seger for his words of encouragement and good advice.
My journey to study abroad would not have been possible if not for the support I received from the scholarship program "Don Carlos Antonio López".
Thanks to my partner in crime, the beautiful Silvia Trigüis.
Lastly thanks to my family, you are everything to me.

Juan Ricart, Gothenburg, May 2019

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background

Prominent among software tools that assist in the development of mathematical proofs are interactive theorem provers, more commonly known as proof assistants. The main objective of proof assistants is to aid in the development of *formal mathematics* by verifying that proofs created by the user hold with absolute rigor. It can be argued that a practical application of proof assistants is to support the process of peer review that is commonplace in scientific publishing. An automatically verified proof not only makes explicit the reasoning which makes a proof correct, but it also increases the confidence on its correctness to a level higher than a manual peer review can.

Direct beneficiaries of tools such as proof assistants are software developers who can state the properties of their programs and use proof assistants to ensure they fulfill those properties. Still, proof assistants are not generally widespread among software developers, who choose not to specify and prove the properties of their programs. The reasons for this are numerous; primary among them is that writing rigorous proofs is still a tedious process that requires a lot of *manual* intervention. Thus, the opportunity exists to increase the participation of the computer in the development of a mathematical proof. In other words, to allow the possibility that some steps in the proof be done *automatically*.

In recent years, a technique that automatically discovers interesting properties from an existing theory or program has been developed which is known as theory exploration. Theory exploration automatically invents new and interesting concepts in mathematics or about functional programs. A theory exploration system takes as input a list of functions and datatypes, and automatically discovers new and interesting properties about them.

Efforts to integrate theory exploration systems with proof assistants have met with increasing success. A promising candidate among systems of this kind is Hipster [22], which is integrated with the Isabelle/HOL [27] proof assistant. The back end component of Hipster which automatically discovers new lemmas is a state-of-the-art theory exploration system known as QuickSpec [32].

Once Hipster provides QuickSpec with a set of constructs (datatypes and functions in the theory) to explore, it discovers equations which are *very likely* to hold. It accomplishes this by generating equations involving these constructs and then exhaustively testing them with many different random values. It is important to note that although the equations found by QuickSpec are likely to be true, they are

not formally proved. The job of proving the lemmas found is left to Hipster which relies on the facilities provided by Isabelle/HOL to perform this task.

The implementation of Hipster has seen improvements to such an extent that it has become a practical tool to discover lemmas in two different scenarios:

Exploratory mode At any point during the development of a theory the user should be able to select an arbitrary set of constructs on which to perform exploration. This mode is particularly useful at the outset of the development of a theory.

Proof mode Used to prove a specific conjecture. The set of constructs to explore should only encompass those that produce the missing lemmas which allow the specific goal to be proved. This mode describes the scenario where a user is stuck in an attempt to prove a conjecture.

The reason to distinguish between these two scenarios has to do with the number of constructs over which theory exploration is performed and how this affects the runtime. In its search for equational properties, QuickSpec explores all terms up to a certain size built from the constructs it is given. The runtime of QuickSpec is proportional to the number of terms it explores [32]. Therefore, its runtime will be *exponential* in the number of constructs selected for exploration.

In exploratory mode the number of constructs on which to explore is arbitrary and may well encompass all the constructs defined in a theory. This will have a negative impact on the performance of theory exploration. In contrast, proof mode can perform better given that the constructs are limited to include only those that produce relevant lemmas for a specific conjecture.

But which constructs are most likely to produce relevant lemmas after theory exploration in proof mode?. This remains an open question with many opportunities to explore different options. In its current implementation, Hipster delegates to the user the task of selecting the set of constructs to explore.

For instance, consider the theorem which states that reversing a list twice produces the original list. In Isabelle/HOL, this would be written as:

```
theorem reverse_twice : "reverse (reverse xs) = xs"
```

In a proof by induction on the length of the list, the base case can be trivially proven because it is enough to replace the calls to `reverse` by its definition. This is not the case for the inductive step:

```
reverse (reverse (x # xs)) = x # xs
```

At this point, Hipster can be invoked in proof mode and one could assume that without further knowledge the user would choose to discover lemmas involving the functions `reverse` and the list datatype constructors `(#)` and `[]`. `reverse` and the datatype constructors would be selected since they are the ones that directly appear on the conjecture to prove.

As it turns out, a possible lemma necessary to prove the inductive step is:

```
reverse (reverse (xs @ ys)) = reverse ys @ reverse xs
```

Alas, this lemma is not found among the set returned by QuickSpec. Had the user also included the append function `(@)`, the lemma would have been found immediately.

1.2 Statement of the problem

This work deals with functional programs that are expressed as theories in HOL. The axioms of the theory are regular sum-of-product types, which can be defined using HOL's `Datatype` command. Sum-of-product types can be defined recursively as follows [31]:

$$\begin{aligned} T(\alpha_1, \ldots, \alpha_p) = & C_1(t_{1,1}, \ldots, t_{1,m_1}) \\ & \mid \ldots \\ & \mid C_n(t_{n,1}, \ldots, t_{n,m_n}) \end{aligned}$$

The arguments $\alpha_1, \ldots, \alpha_p$ to the type T are type variables. The right-hand side is made up of one or more clauses. A clause is identified by its value constructor, denoted by C_i, which takes zero or more type specifications as arguments, denoted by $t_{i,j}$. Type specifications are either type variables or instances of sum-of-product types (including T itself).

Values of these types are manipulated by functions whose definitions follows the argument's recursive structure using pattern-matching. The `Define` facility is used to define these functions in HOL.

Functions are made up of conjunctions of equations. Each equation has a left-hand side with the name of the function and a pattern. The right-hand side is a term (a well-formed word according to the functions and datatypes defined). Each pattern may be a variable, a wildcard, or a value constructor.

It is to be noted that functions defined via `Define` are subject to an automatic proof of termination. Also, a partial function definition is completed with an extra equation which returns the singleton value `ARB`. Thus, all functions defined are assumed to be total and terminating. Many interesting functional programs can be constructed using functions with these properties.

Not only types and functions are expressed as equations but also the goal the user is trying to prove. Goals are expressed as equational identities between two well-formed terms (e.g. the goal in Section 1.1 where the left-hand side of the identity is `reverse (reverse (xs @ ys))` and the right-hand side is `reverse ys @ reverse xs`).

Proving a specific equational property from a given set of constructs (datatypes and functions) is a problem which has been extensively studied and is referred to as the "Word Problem". The problem treated in this work is a variation of the Word Problem where in addition it is assumed that an induction principle exists for each datatype in the set of constructs. For example, the induction principle for lists defined via the value constructors `[]` and `(::)` is:

$$\vdash \forall P.\, P\, [] \wedge (\forall t.\, P\, t \implies \forall h.\, P\, (h :: t)) \implies \forall l.\, P\, l$$

The Word Problem is known to be undecidable [23]. Short of providing a general procedure to solve this problem, a new tactic is proposed that is able to help the HOL user in many circumstances.

1.3 Aim of this work

Having set up the context in Section 1.2, below are the specific aims of this thesis work:

1. Integrate QuickSpec with HOL into a tool called Hopster that supports the development of verified functional programs. It should provide automatic generation of lemmas at the outset of a theory development (exploratory mode) and for a particular goal in a proof attempt (proof mode).
2. To consider the issues that arise when using theory exploration for the purpose of proving a specific goal. In particular, a) which functions (and datatypes) to choose to perform theory exploration on and b) the strategy used to prove the resulting list of conjectures returned after theory exploration.

A successful completion of this thesis project should considerably improve the way HOL can be used. On receiving suggestions of relevant lemmas sent in by Hopster, a user of HOL who is stuck trying to prove a particular goal should be able to fill in the missing steps required.

1.4 Related works

Hopster can be seen as part of a long tradition of computer tools with the mission to support the mathematical activity through theory exploration. The present Section provides a short overview of the most relevant systems that came before Hopster and thus have exerted an influence on its design and implementation.

Some of the systems considered below have certain functionality that goes beyond basic characteristics of a theory exploration system. Besides the features one would usually expect, such as building theories using the axiomatic method or the automatic generation of proofs, some systems also support rendering proofs in natural language or creating papers in a structured way for publication. In the summary for each system attention is paid exclusively with those aspects pertaining to theory exploration. These aspects are: 1. the selection of auxiliary functions to explore in order to prove a goal (i.e. proof mode support), 2. conjecture discovery and 3. the automated proving of the conjectures found.

1.4.1 Theorema

It is only fitting to begin by considering the *Theorema* system. The Theorema project was initiated by Buchberger [4], who originated the concept of theory exploration. According to this view, theorem provers paid special attention to proving an isolated theorem at a time whereas mathematicians usually work by exploring a whole theory. Thus a proof assistant that supports theory exploration should support the user in all phases of theory exploration. In particular, it should allow the definition of axioms and show the validity of propositions by identifying the proper proof methods for a particular theory and incrementally incorporate discovered lemmas [7]. Since the system was designed to encompass all activities of a mathematician Theorema is implemented in Mathematica, from which many of its features are leveraged to accomplish these goals.

The language in Theorema is a version of higher-order logic, meaning it extends predicate logic by letting variables range over sets. From this base, the user can decide how to build the theory (for instance, basing the theory on the axioms of Zermelo-Fraenkel set theory). The system works in a way similar to the HOL family of proof assistants where the object-level language for describing a theory is Theorema and the meta-level language, for extending the reasoners, is Mathematica.

The selection of auxiliary functions on which to explore is done manually by the user. Once the proof goal has been selected, the system presents the *knowledge browser*. This consists in a summary of all the formulas (theorems) available in each notebook (theory) that is open. These theorems are displayed in hierarchical structure according to the groupings the user set in the notebook, which facilitates the selection of all the formulas in a group [7].

Conjecture discovery in Theorema is available in the context of the automated synthesis of an algorithm from a specification. The specification is a pair of two conditions where one states the property that the input must fulfill and the other the respective property for the output. Once the specification has been set, an automatic attempt to derive the algorithm is attempted using algorithm schemes. If the proof fails, the information contained in the failing attempt is used to conjecture the specification of auxiliary algorithms that are needed for the proof to succeed [6]. An attempt is made of proving the auxiliary algorithms in a process called lazy thinking. This technique was first presented in [5].

Automated proving requires manual configuration by the user each time proving a goal is required. The system allows the selection of a specific proof strategy and to (de)activate each inference rule that is defined for it. Once this configuration is set the prover generates a proof object automatically. Proof objects are and-or trees where each interior node represents a conjunction or disjunction and has a status: success, failure or pending. The generation of the proof tree terminates when at least one path from the root to a leaf has all its nodes with a status of success, or a time/space limit has been reached [7].

1.4.2 MATHsAiD

The purpose of the MATHsAiD project is to build a software system to perform the automatic discovery of all the interesting lemmas and theorems from an initial theory [24]. As such, the main emphasis of the system is to synthesize new conjectures from an initial set of axioms and definitions supplied by the user (what is regarded in this work as exploratory mode). Proving a specific goal is not the main focus but rather to make sure the lemmas discovered achieve a high level of "interestingness". This allows the system to distinguish theorems that have great significance while at the same time not distracting the user with an unacceptable number of uninteresting theorems.

The logic used in user theories is customizable, but the default logic is a classical, untyped higher-order logic that is based on Gentzen's Natural Deduction [24]. The system is implemented in Amzi! Prolog which provides an interface to Java, which is used to build a graphical user interface in the Eclipse integrated development environment.

As previously stated, there is no special support for proving a specific goal supplied by the user. Under this scenario, determining the auxiliary function definitions that will most likely yield useful lemmas to prove a specific goal needs to be considered manually.

Details on the procedure to discover new conjectures automatically is documented in [24]. As a rough overview, all conjectures generated have the form of a conditional proposition:

$$\theta \implies \rho(y_1, \ldots, \xi, \ldots, y_k)$$

They are comprised of a k-ary predicate, ρ, taken from the initial set of predicates provided and what is known as a term of interest, ξ, a term that has a type compatible with an argument of ρ. The terms of interest are generated exhaustively, following a set of schemes up until some resource limit is reached (such as the size of the term or the maximum nesting allowed). The rest of the arguments for ρ, if any, are filled with distinct variables y_i. Finally, the hypothesis θ is constructed using ξ.

θ is a conjunction of type declarations and non-type hypotheses. Type declarations are easily calculated for every variable in ξ, its type is identified, τ, then the declaration states $\tau(y_i)$. Non-type hypotheses are generated from each predicate ς in the theory different from ρ and for each combination of arguments that are type-compatible.

1.4.3 IsaScheme

As its name implies and following in the tradition of Theorema, IsaScheme embodies a scheme-based approach to theory exploration. A scheme is a term in higher-order logic whose purpose is to capture a certain pattern that has proven useful for discovering new interesting theorems or concepts in mathematics. A scheme has free variables which once instantiated with particular values turn the scheme into a new conjecture or even a new function definition. Whereas in Theorema the user is required to manually instantiate the free variables in a scheme, this is done automatically in IsaScheme [26].

The principal motivation for the ongoing development of IsaScheme is to assess the capabilities and limitations of the scheme based approach when applied to theory exploration and concept invention. Another motivation is to detail the different techniques and heuristics necessary to automate these processes. IsaScheme is developed on top of Isabelle/HOL, following the LCF tradition.

With regards to proof mode, IsaScheme does not provide special support for automatically choosing the datatypes and functions over which to perform exploration. Two initial sets of closed terms X and schemes S need to be provided manually. The X set represents the function definitions and the S set represents the schemes that are used to augment X with new function definitions and also generate new conjectures.

The procedure for generating new conjectures is as follows: the instantiation of a new conjecture happens when all the free variables in a scheme $s \in S$ are substituted with elements from $x_i \in X$. The substitutions are only possible if the free variables are type-compatible with their corresponding x_i. So, for every free variable v_i of

s, a domain is initialized with all the x_i whose type is unifiable with v_i. Then the free variables are processed in sequence, selecting any of the x_i at random. Every time a v_i is substituted with a x_i, the domains of the free variables which are still to be substituted need to be adjusted accordingly. This process continues until all free variables have been assigned a closed term at which point the schema has been instantiated into a new conjecture. This process of substituting the free variables of a scheme with closed terms is essentially solved using a constraint satisfaction algorithm.

Once a new conjecture has been generated it passes through a process of normalization using a terminating higher-order rewrite system which is supplied manually by the user. Then, the normal form of the conjecture is checked whether it is subsumed by any previously proved theorems or if it is trivially proved using simplification. If so, the conjecture is discarded. Otherwise, a counter-example check is applied to the conjecture to reduce failed proof attempts. Counter-example checking is performed by Isabelle/HOL via QuickCheck [10] or Nitpick [3]. Finally, if no counter-examples are found, a proof attempt is performed via IsaPlanner [14].

1.4.4 IsaCoSy

It is to be noted that some of the theory exploration tools presented thus far use a *top-down strategy* for the generation of new conjectures. Following this strategy, new conjectures are arrived at during attempts to prove a particular goal. A failed proof attempt provides data to guide the discovery of useful conjectures which are then independently proved, using techniques such as rippling and higher-order unification. Lemmas obtained in this way can be subsequently used for the proof of the original goal. In contrast, a *bottom-up strategy* does not rely on any information from failed proof attempts. It searches the available term space in a theory starting from the smallest possible terms (such as constant symbols), and gradually increasing their depth by composing them until new properties are found that hold for the theory.

It has been shown that a bottom-up strategy for theory exploration succeeds in finding lemmas that are not found using a top-down approach [12].

The **Isa**belle **Co**njecture **Sy**nthesis system, or IsaCoSy, generates conjectures using a bottom-up approach. IsaCoSy works with inductive theories, consisting of recursive datatypes and function definitions. From this initial set, IsaCoSy builds terms starting with the smallest term possible and then incrementally building larger ones [21].

An increase in the maximum depth for a term translates into an exponential growth in the total number of terms that can be generated. To successfully deal with this search space explosion, IsaCoSy only considers the generation of irreducible terms. Irreducible terms are those that do not appear in the left-hand side of any rewrite rule (function definitions). Encoding the restrictions that determine which terms are irreducible is done by means of constraints in the conjecture synthesis process.

The conjectures found are then further processed by a counter-example checker in order to quickly discard those that are false. An attempt to prove the remaining ones is carried out by the automated inductive prover IsaPlanner which tries a proof

by induction using the rippling heuristic.

1.4.5 HipSpec

HipSpec was intended for the study of automating the proof of algebraic properties of Haskell programs [11]. Specifically, it works on a subset of Haskell programs, which consists of monomorphic and terminating programs without type classes or primitive types (like `Int` or `Char`). Possible datatypes are algebraic data types, functions and uninterpreted types [12]. The name HipSpec originates from the two subsystems is comprised of: Quick**Spec** for generating new conjectures and **Hip** for proving them using induction.

Particular support for proof mode is not a goal of the system. The input consists of a Haskell source file. If the file includes properties about the datatypes and functions defined, then the system will attempt to prove those properties, in the style of an automated theorem prover. It will additionally return the set of lemmas it has discovered (some of which may have been necessary for the proof of the user stated properties) and conjectures it has not been able to prove but which have been exhaustively tested. The user has manual control over which datatypes and functions to explore by selecting which ones to include in the Haskell source file.

New conjectures are synthesized by means of QuickSpec. With the set of datatype constructors and functions defined, QuickSpec generates all possible terms up to a configurable depth. Then it calculates equivalence classes over these terms by means of testing. A pair of terms, t_1, t_2 that are in the same equivalence class form a new equality conjecture $t_1 = t_2$ [12].

The subsystem inside HipSpec in charge of proving the conjectures found is called Hip, the Haskell Inductive Prover [29]. As its name suggests, it attempts the proof by deciding how to perform structural induction. For this, it enumerates all the possible ways induction can be performed over the set of free variables in the conjecture. Subsequently, an attempt to prove the induction cases is performed by calling an external automated first-order prover [12].

1.4.6 Hipster

The first system to successfully integrate a bottom-up approach for conjecture synthesis with a proof assistant for the development of inductive theories is Hipster [22]. As previously mentioned, Hipster integrates QuickSpec with Isabelle/HOL.

The principal motivation for Hipster has been to harness the performance gains of HipSpec, in particular its theory exploration component: QuickSpec, after realizing it is a suitable choice for the generation of new conjectures in an interactive setting [20]. An interactive setting refers to proof mode, where the user is able to ask for suggestions of new lemmas, not at the outset of a new theory, but for a particular proof attempt. This is realized via the `Hipster.explore_goal` function which lets the user manually tailor the functions over which to perform theory exploration. The list of functions specified by the user are passed as an argument to `Hipster.explore_goal`.

Once the list of auxiliary functions on which to explore has been determined, the

process of conjecture discovery begins by translating the function definitions into a Haskell program. This step is performed by means of Isabelle/HOL's code generator. Then, the discovery of new equational laws is performed by QuickSpec combining these functions using a bottom-up strategy, as outlined above. All laws generated in this way are exhaustively tested by QuickCheck. The purpose of performing various test-cases with random values using QuickCheck is to ascertain that the new conjectures discovered are very likely to hold.

To manage the proof of the conjectures discovered, a loop is entered where an attempt to prove each conjecture is carried out. The more general conjectures are attempted first so that the resulting lemmas can be used to safely discard other conjectures which can now be proved using routine reasoning tactics. This helps to avoid cluttering the user with many lemmas which are simple consequences of the more general ones. Using this procedure, it may occur that proving a conjecture fails because its proof requires a lemma which in turn has yet to be proved. When this happens, the conjecture is put back into the list of conjectures to be proved and will be retried later; when an attempt to prove the rest of the conjectures has been made. In this way, the loop iterates until there is nothing left to prove or when all attempts to prove the conjectures in the list fail in one iteration, thus no more progress is possible. [20].

1.5 Organization of the text

The rest of this book is organized as follows: Chapter 2 provides an overview of the design of Hopster and how it establishes a bridge between the HOL proof assistant and the QuickSpec theory exploration tool. Chapter 3 goes into detail on different components necessary to make Hopster work. Chapter 4 evaluates the effectiveness and efficiency of the current implementation of Hopster by subjecting it to a variety of test cases. The evaluation records whether useful lemmas are found and the time it takes for Hopster to do so. Finally, Chapter 5 analyses the results obtained from the tests and closes the book with final remarks and suggestions for further work.

2
An overview

The design and implementation of this project is inspired by Hipster and, in general lines, can be described by the flowchart in Figure 2.1.

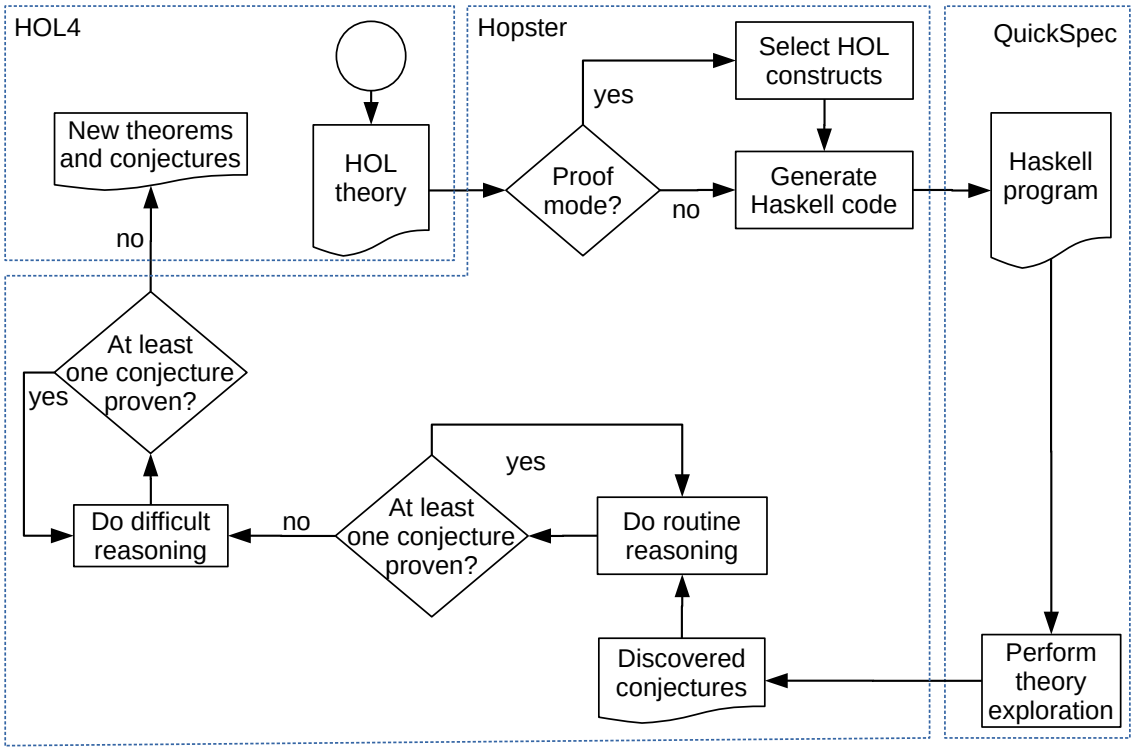

Figure 2.1: An overview of Hopster

As is shown, there are three components which interact in a theory exploration: HOL, QuickSpec and Hopster. Each of this components is described in the rest of this Chapter.

2.1 The HOL4 proof assistant

HOL4 is the latest iteration in a family of proof assistants which can trace its lineage as far back as the original Stanford LCF [15]. HOL4 is a proof assistant for **higher order logic**, hence the name HOL. Higher order logic can be interpreted as an extension of predicate calculus where[16]:

- Variables can range not only over concrete values but also over functions (from which stems the name *higher-order* to denote this logic)

- Every term in the logic has a type
- Predicates are functions whose return value is of type boolean. They are not special with respect to other functions

This logical foundation was chosen for its suitability to hardware verification. It has since been successfully applied to many different areas, which range from the development of mathematical theories to the verification of software (including the verification of a compiler for a large subset of Standard ML).

HOL4 is written in the Standard ML programming language and a session in the HOL system is a regular Standard ML top-level loop where the HOL libraries have been loaded. The objective of a session in HOL is to produce a *theory*.

A HOL theory, like any mathematical theory, contains definitions and axioms. Yet, the user that is developing a theory is mostly concerned with the creation of *theorems*. Theorems are represented in HOL as Standard ML values that have the type `thm`. This datatype is encapsulated in such a way that a HOL user can only create values of type `thm` by generating a proof. Proofs, in turn, consist of the repeated application of ML functions (which represent the rules of inference) to either axioms or previously proved theorems. By the successive application of these functions, it is guaranteed that a HOL user is only able to create valid theorems and the logic remains consistent.

2.2 QuickSpec and the TIP tools

QuickSpec is a theory exploration system capable of scanning a Haskell program and automatically discovering its specification [32]. The specification of the program is expressed as equational laws that the program's functions and constants are likely to satisfy. It may be noted that although the equational laws are tested exhaustively by QuickSpec, they are not proven correct.

The practical uses for a theory exploration tool such as QuickSpec are numerous, among which could be cited [32]:

Documentation to help understand program libraries for the cases where the documentation is missing

Lemma discovery to discover interesting new lemmas about a mathematical theory. This is the application which this thesis work concerns

The TIP (Tools for Inductive Problems) tools [30] is a suite of utilities that sit on top of QuickSpec providing various support for theorem provers. `tip-ghc` translates a Haskell program into TIP's internal format, `tip-spec` runs QuickSpec to perform theory exploration and `tip` translates the list of discovered conjectures to the syntax required by various proof assistants, among them HOL4.

TIP also refers to a set of standard benchmarks to test the performance of inductive theorem provers [13].

2.3 Hopster

Hopster is the object of this thesis work. It defines a library which acts as an interface between the HOL4 proof assistant and the QuickSpec theory exploration

system. A general understanding of what Hopster does can be gleaned by following the flowchart illustrated in Figure 2.1. The original design for this procedure is based on the description of Hipster and is documented in [22]. Notes are added in places where Hopster deviates from the design of Hipster and in Chapter 3 an in-depth view is provided of those features which are not present in Hipster and which constitute original contributions of this work.

For a user of HOL4 wishing to perform theory exploration, the starting point is a HOL theory loaded in the proof assistant. At this point theory exploration may be invoked in one of the two modes described: *exploration mode* or *proof mode*. Exploration mode takes all the constructs defined in the theory with the objective of discovering new lemmas, whereas proof mode is used to discover which lemmas are relevant to just one particular sub-goal in an open theorem. Hopster provides particular support for proof mode by automatically selecting the HOL constructs that are relevant to the particular sub-goal at hand. The selection is based on three heuristics that a) take into account local information which can be extracted from the goal b) global information which can be extracted from the theory and c) a general procedure which is applicable to any goal and theory.

Both modes of operation require the translation of constructs of HOL4 into a format suitable to be processed by QuickSpec: either Haskell or the TIP format [30]. In Hipster this step is accomplished by calling Isabelle/HOL's code generator which can translate Isabelle's higher-order logic to code in several functional programming languages, including Haskell. In contrast, HOL4 does not have this component, so a suitable translation step from HOL constructs to Haskell has been developed and documented on Chapter 3.

Having the Haskell file in place, theory exploration is invoked to generate a list of equational conjectures. Although theory exploration can potentially produce many conjectures Hopster keeps them all eventually presents them the user. In contrast, Hipster does not return every conjectures which is discovered; only those that are deemed most interesting (i.e. those that are difficult to prove) while the others are filtered out.

Before attempting to prove each conjecture, the list is sorted by QuickSpec according to generality: more general equations are given priority, as it is expected they can be more widely applicable as lemmas for proving the other conjectures.

Once the list of conjectures is sorted by generality, QuickSpec translates them into HOL's format and an attempt to prove them begins. The proof procedure is broken down into two stages. Both stages are similar except for the tactic used to prove each conjecture. The first stage specializes in proving easy conjectures, therefore it is known as the *routine reasoning*. Routine reasoning uses HOL4's simplification tactic to turn a conjecture into a theorem. The second stage is called *difficult reasoning* and its tactic performs structural induction followed by first-order reasoning.

Each stage works through the list of conjectures one at a time. If a conjecture is proven it is added to a list of known lemmas and the loop continues processing the next. If it cannot prove it then nothing happens and the loop considers the next conjecture as before. At any moment, both stages have access to the lemmas proved so far. So, every new conjecture that is proved increases the power of these procedures as they loop through the list of conjectures. For this reason, if at the

end of one iteration at least one conjecture was proved, the procedure is repeated again with the list of conjectures which still remain to be proved. If no new lemma was added to the list by the end of an iteration, the process ends.

Once the proof procedure finishes, the list of theorems and unproven conjectures that remain are returned to the user of HOL4 (it is deemed that unproven conjectures are still of interest to the user since they have been thoroughly tested, so are left for the user to attempt to prove them manually). This terminates the procedure of invoking Hopster inside a session of HOL4.

3

The specifics of theory exploration in HOL

The exploration of a theory with Hopster, as described in Section 2.3, requires the successful inter-relation of a few components which, as a whole, make up the Hopster module:

- A heuristic that selects the HOL constructs which are more likely to produce relevant lemmas when proving a particular conjecture, for the case when theory exploration is invoked in proof mode.
- A code generator that translates the elements of HOL logic into a format suitable to be processed by the TIP tools, such as a Haskell program file.
- A pretty-printer for QuickSpec that outputs the discovered lemmas in HOL format.
- The proof loop which imports the lemmas found into the current theory and tries to prove them using suitable tactics.

Each of these four components has been developed exclusively for Hopster and is an original contribution of this work. This Chapter documents in detail the design considerations that went into their implementation.

3.1 The selection of HOL constructs for theory exploration

The selection of which datatypes and functions to explore (when the exploration is made in proof mode) is based on three heuristics. The first heuristic leverages information that pertains to the particular goal the user is trying to prove. The second heuristic uses information available in the current theory, when the theory is fairly well developed. Finally, the third heuristic provides a schema that synthesizes new functions irrespective of the goal or theory.

3.1.1 Local information from the goal

The most readily available information in a goal is the list of types and functions which occur in the statement of the conjecture. It is reasonable to expect that these types and functions will also occur in any lemmas that are necessary to prove the conjecture. This idea is used in the selection of which constructs to explore and is referred to as the "local heuristic".

Continuing with the example goal shown on Section 1.1, in HOL syntax:

```
∀ (xs : 'a list). reverse (reverse xs) = xs
```

It involves the `'a list` type and the function `reverse`. A convenient lemma to use when proving the conjecture above is:

```
∀ xs ys. reverse (append xs ys) = append (reverse ys) (reverse xs)
```

The lemma states how `reverse` distributes over `append` and it is clear that `'a list` and `reverse` are involved in the statement of this lemma. So, it would be a good idea to include them among the functions and datatypes to explore. However, including only `'a list` and `reverse` will not be enough for QuickSpec to discover this lemma. It must also include `append`.

In fact, the sole inclusion of `'a list` and `reverse` will produce an error on a theory exploration tool such as QuickSpec. The reason for this can be gleaned by examining the definition of `reverse`:

```
val reverse_def = Define `
  reverse Nil = Nil /\
  reverse (Cons x xs) = append (reverse xs) (Cons x Nil)`
```

As QuickSpec generates different terms it exhaustively tests that each pair is indeed equal. Following the example, after generating simple terms such as `xs` it will proceed to generate terms involving `reverse`, such as `reverse (reverse xs)`. To test the equation `reverse (reverse xs) = xs` for different values of `xs` it needs to evaluate `reverse`. Its definition depends on `append`, but the definition for this function is missing.

Therefore, when selecting types and functions which directly appear in a goal, their definitions should be traversed as well in a recursive manner until all functions and types which are necessary for testing are included in the call to QuickSpec.

3.1.2 Global information from the theory

A well developed theory, with a large number of function definitions and lemmas, offers an opportunity to derive another heuristic. The functions which are referenced the most in the theory's lemmas are likely candidates to appear in any missing lemma which one hopes to discover through theory exploration. This heuristic is named the "global heuristic".

Obtaining the most referenced functions is done by querying the theorems stored in the current theory, via `DB.thms`. A call to `DB.thms` will return a list with the names of all the theorems, definitions, and axioms that are stored in the theory whose name is passed as an argument. This list is then checked to see how many times each operator is referred to in the the actual definitions, axioms and theorems. A ranking is produced sorted by which operator appears the most. Then the top two operators from this ranking are selected for theory exploration.

3.1.3 Synthesizing new functions

Contrary to the global heuristic, a theory in its initial stages of development will lack sufficient lemmas from which to select the functions to explore. Furthermore,

there is the possibility that the necessary lemma refers to a function which belongs to a different theory.

Consider a theory of lists in its early stages of development where the `append` function is missing but a version of `reverse` exists which does not require `append`:

```
val rev_def = Define `
  rev Nil ys = ys /\
  rev (Cons x xs) ys = rev xs (Cons x ys)`
```

After writing the definition above the user sets the following goal to check `rev`'s correctness:

```
∀ xs. rev (rev xs Nil) Nil = xs
```

A manual proof attempt reveals that one possibility is to use induction on `xs` and the following two lemmas:

```
∀ xs ys zs. rev (append xs ys) zs = rev ys (rev xs zs)
∀ xs ys zs. append (rev xs ys) zs = rev xs (append ys zs)
```

The problem is that both lemmas refer to `append`, a function which will not be selected for exploration either by the local or global heuristics.

To accommodate this scenario one more heuristic has been developed which is based around the concept of a scheme. As is used in IsaScheme and Theorema, a scheme is a term in higher-order logic whose objective is to capture a particular recurring pattern that is observed in the definitions of various functions in a theory. The scheme contains holes that need to be filled in order to convert the scheme into the definition of a particular function. The holes are represented as universally quantified variables and the process of filling the holes is made by instantiating the variables within the scheme.

Below is an example of a scheme:

```
∀ (f : 'a -> 'b -> 'b).
  ∃ fn. !x xs ys. (fn Nil ys = ys) /\
                  (fn (Cons x xs) ys = f x (fn xs ys))
```

The existentially quantified variable `fn` denotes the name of the function to be defined. The `f` hole needs to be instantiated to complete the definition. Fortunately QuickSpec is able to generate and compare terms of function type [32]. If, in continuing with the example above, the scheme were among the set of functions to explore then QuickSpec would find that the hole can be filled with `Cons` which is type-compatible with `f`. With this, the scheme is instantiated to the `append` that was required and the necessary lemmas are discovered.

From this informal explanation and example, a formal definition for a scheme is given, adapted from the *definitional scheme* in [26]: A term is designated as a scheme if it has the form $\forall\bar{x}\exists\bar{f}\ \wedge_{i=1}^{n} l_i = r_i$, where the head of each $l_i \in \bar{f}$. The notation $\bar{x}$ denotes an arbitrary number of x, likewise for $\bar{f}$. The defining equations for the function are denoted by $l_1 = r_1, \ldots, l_n = r_n$.

The scheme presented above is one example of a scheme which is valid according to this definition. Virtually an unlimited number of schemes exist which conform to the definition given. Valid schemes can be created based on the user's experience with a particular theory. However, in order to fully automate the process of theory

exploration in proof mode, this work focuses on the applicability of one particular scheme: the fold operator on regular datatypes.

Previous studies have shown that a fold operation is not exclusive to lists. By using concepts from category theory, a fold operator can be defined for any regular datatype [25, 31]. The idea for the "fold heuristic" is to derive the fold operator for the types involved in the goal. More specifically, those types that correspond to the universally quantified variables in the goal. This is done in order to limit the number of different fold operators to explore.

It is important to note that although the process of deriving a fold operator can be made fully automatic in HOL4 (in particular by starting with the primitive recursive definition for the datatype obtained with `TypeBase.axiom_of`), for the purposes of the evaluation tests this derivation has been manually performed for each datatype that was tested.

3.2 The code generator

Higher Order Logic (or HOL), as implemented in HOL4, is a version of the simple theory of types due to Church [9]. Thus, the HOL language can be described by means of two different syntactic categories: types and terms. There is a Hopster module dedicated to the translation of HOL types and terms into types and functions of the Haskell programming language. This translation allows for the exploration of HOL types and terms using QuickSpec.

Although in what follows the textual representation of the different HOL constructs is described using BNF notation, the general procedure for their translation only deals with a parsed representation of them in memory. The procedure traverses this representation in the manner of a recursive descent parser, recursively applying destructors to break up a given construct into its constituents and translating each of these in turn.

The output of the translation process is a `Doc`-ument datatype wherein the types and terms (functions) in Haskell syntax are contained. The `Doc` datatype is defined as part of a pretty-printing library which is due to Hughes [17] which was ported to the Standard ML programming language.

Inter-process communication between the Hopster module and QuickSpec is done through files in the operating system. The resulting `Doc` is rendered to a file which is then passed on to the TIP tools by executing the following command:

```
> tip-ghc Explore.hs | tip-spec | tip --hopster > tip-stdout.txt
```

`Explore.hs` is the external file which Hopster writes after rendering the `Doc` to a textual representation. Since the discovered lemmas are printed to standard output, the command above redirects this output to a file so it can be read by Hopster once theory exploration is finished.

3.2.1 Lexical matters

Variable names in HOL can contain any character that is valid in a Standard ML string, although this degree of freedom is often discouraged. Generally, only special

forms for names known as identifiers are used. Identifiers can be classified into two sorts:

- A sequence of alphanumeric characters which start with a letter
- A sequence formed by any combination of symbols

From these considerations, the accepted forms for identifiers in the code generator are defined in the following BNF grammar below.

```
varid ::= <alpha>
      |   <alpha> <alphanumeric>
      |   <symbolic>

tyvarid ::= ' <alpha>
        |   ' <alpha> <alphanumeric>

alphanumeric ::= <alphanum> | <alphanum> <alphanumeric>

alphanum ::= <alpha> | <digit>

alpha ::= <small> | <large>
small ::= a | b | ... | z
large ::= A | B | ... | Z

digit ::= 1 | 2 | ... | 9

symbolic ::= <symbol> | <symbol> <symbolic>

symbol ::= # | ? | +  | * | / | \ | = | < | >
             | & | \% | @ | ! | : | | | - | ^ | ‘
```

Even though the above grammar defines a reduced set of characters accepted for the different names of HOL constructs (non-terminals `varid` and `tyvarid`), a slight difficulty remains. The difficulty stems from Haskell names being governed by even stricter rules (for instance, Haskell's value constructor names should begin with an uppercase letter). To solve this problem, the code generator transforms HOL names to syntactically correct Haskell names during translation.

In order to facilitate importing the discovered conjectures back into HOL, the TIP tools accept specially formatted comments in the generated Haskell file. The original HOL names are annotated in this way, one name per line. This allows the `tip` tool to rename the variable and type names again prior to sending the discovered conjectures back to HOL. Below, an example function definition for `reverse` is shown on which theory exploration is to be performed. Attached to this definition, annotations for the different names of variables and types used in the original HOL are included.

```
  {-# ANN type List (Name "list") #-}
  data List a = NIL | CONS a (List a)

  {-# ANN reverse (Name "REVERSE") #-}
  {-# ANN append (Name "APPEND") #-}
```

```
reverse [] = []
reverse (h : t) = append (reverse t) [h]
```

The identifier `reverse` will be renamed back to `REVERSE` when the discovered conjectures are returned. The same applies to the identifiers `List` and `append`.

3.2.2 Translation of HOL types

HOL types are defined with the same syntax as for Standard ML. Their grammar is presented below.

```
type ::= : <base>
     |   : <base> <type>

base ::= <tyvarid>
     |   <typeid>

typeid ::= <alpha>
       |   <alpha> <alphanumeric>
```

Moreover, types recognized by the code generator should be registered in the HOL system `TypeBase`. The `TypeBase` is a database that keeps track of many features of a new type that is introduced in HOL. Of interest for Hopster's code generator are the value constructors of each type registered. Types are registered in the `TypeBase` when they are created with the `Datatype` directive.

3.2.3 Translation of HOL terms

There are only four possibilities for HOL terms: variables, constants, abstractions or applications. Every possible term is composed of these four basic elements. Moreover, special syntactic support has been added for specially recognized terms. In this way, HOL supports features present in most functional programming languages, such as: let expressions, conditionals, pattern matching, infix operators and more. This specially recognized terms are supported in the form of syntactic sugar, so that they ultimately get translated into combinations of the four elementary forms.

Although translating the four basic forms for terms would have been enough for the purposes of this thesis work, extra effort has been put towards recognizing wider syntactic forms that have a direct translation to Haskell. Notable terms of HOL logic that are not recognized by the translator are $\exists$-quantification and ϵ-terms.

The result of the translator is a Haskell file that is more easily readable than it would otherwise be. It is hoped that the extra readability in the translated code proves helpful in a number of practical ways:

- To users of HOL4 who wish to perform theory exploration, the generated file is readily available for debugging purposes.
- The generated code may be used for a different purpose than as input to perform theory exploration. For instance, providing a testing facility for HOL definitions that executes them with noticeable speed-ups.

For similar reasons, care has been taken not to clutter the output Haskell file with an excess of parentheses.

A partial grammar for terms recognized by the Haskell code generator in Hopster is shown below.

```
defn ::= <eqn>
     |   (<eqn>) /\ <spec>

eqn ::= <lhs> = <rhs>

lhs ::= <alphanumeric> <pats>

pats ::= <pat>
     |   <pat> <pats>

pat ::= _                  (* wildcard *)
    |   <varid>            (* 0-ary constructor *)
    |   (<varid> <pats>)   (* n-ary constructor *)

rhs ::= <varid>
    |   <number>
    |   <string>
    |   ()                 (* unit value *)
    |   <condition>
    |   <list>
    |   <binop>
    |   <pair>
    |   <let>
    |   <abs>
    |   <fail>
    |   <case>
    |   <constructor>
    |   <combination>
```

As in most functional programming languages, the top-level terms recognized are function definitions. Function definitions consist of a conjunction of equations (non-terminal `defn`). Each equation (non-terminal `eqn`) represents the definition of a function. The left-hand side of the equation (non-terminal `lhs`) is phrased using pattern-matching in the same style as in Standard ML. The right-hand side of the equation (non-terminal `rhs`) represents the body of a definition.

3.3 Importing the discovered conjectures

Pretty-printing the conjectures found in HOL syntax is included in the TIP tools and is due to Nicholas Smallbone. Consequently, Hopster leverages the parser for terms in HOL4 to import them. The parser for terms can be invoked by calling `Parser.Term`.

3.4 Turning conjectures into lemmas

Section 2.3 presented an overview of the tactic used to prove the set of conjectures discovered by theory exploration. It noted that it was composed of two stages (tactics themselves). The first one was for *routine* reasoning and its results are fed to the second, meant for *difficult* reasoning. Both stages iterate over the set of conjectures in the same manner but differ in the tactic used to prove each individually. Details on the tactic used for each stage are documented below.

3.4.1 A tactic for routine reasoning

The strategy for recognizing easy to prove conjectures is based on the simplification tactic of HOL4. The simplification tactic, `SIMP_TAC`, adds all the equational lemmas which have been found thus far to the simpset. The simpset is the set of equational theorems used to rewrite the conclusion of the goal. Then, it rewrites the conclusion of the goal using the new simpset.

Simplification is an operation which never fails, but it may diverge. The reason for this is that the rewriting process is applied to a goal recursively, to an arbitrary depth. For certain combinations of rewriting theorems and goals this recursive process may trigger a sequence of rewrites which does not terminate. This divergent rewriting behavior is the result of a term `t` being rewritten (immediately or eventually) to a term that contains `t` as a sub-term.

In order to avoid this possibility, the `Ntimes` directive is employed on each equational lemma discovered. For every lemma `th` returned by QuickSpec, `Ntimes th n` is evaluated, which sets that `th` is to be used at most `n` times in a rewriting process. In its current implementation, Hopster sets the parameter `n` to be equal to ten.

3.4.2 A tactic for difficult reasoning

Conjectures which resist a proof attempt on the stage for routine reasoning are considered interesting. Proving them may yield lemmas which are inherently interesting in themselves. Moreover, interesting lemmas may turn out to be useful for proving a specific goal that prompts the user to perform theory exploration.

The tactic for proving an interesting conjecture is itself based on a combination of two other tactics: induction and first order reasoning.

Induction is performed on the universally quantified variables of the goal. Therefore, goals are expected to be of the form:

```
∀ x. t
```

where `x` is a universally quantified variable, and `t` is an arbitrary term with no free variables other than `x`.

A concrete example is a conjecture which states that the number of elements in a list is equal to the number of elements if the list where reversed:

```
∀ xs. length xs = length (reverse xs)
```

HOL's induction tactic `Induct_on` is passed one of the universally quantified variables in the goal: `xs` is the only possibility for the example shown. It splits the

conjecture into as many subgoals as value constructors for the type of `xs`. `xs` being a list splits the goal into two subgoals: the base case (when `xs` is empty) and the inductive case (when `xs` has at least one element):

```
length (reverse []) = length []

∀ h t. length (h::t) = length (reverse (h::t))
```

If `Induct_on` is successful, the `metis_tac` tactic is applied to each of the subgoals produced. `metis_tac` is a tactic which implements automated theorem proving for first order logic with equality [18]. Attempting to prove a satisfiable first-order formula as unsatisfiable using `metis_tac` may result in a non-terminating computation. To overcome this, the parameter `metisTools.limit` is set to restrict the execution of `metis_tac` to a maximum of ten seconds.

The example above dealt with a goal quantified by a single variable. The actual tactic in Hopster for goals with an arbitrary number of universally quantified variables is composed of a series of nested induction tactics. Each induction is performed over one of the universally quantified variables of the goal. To illustrate this process a second example is given.

Assume that the goal is to prove the associativity of the `append` function on lists:

```
∀ x y z. append x (append y z) = append (append x y) z
```

For each variable, a tactic is constructed which performs induction on it followed by first order reasoning:

```
Induct_on `x` >> metis_tac
Induct_on `y` >> metis_tac
Induct_on `z` >> metis_tac
```

`Induct_on` and `metis_tac` are joined using HOL's `>>` operator. `>>` takes two tactics as an argument T_1 (i.e. induction) and T_2 (i.e. first order reasoning). It first applies T_1 to the goal, if T_1 does not completely prove the goal then T_2 is applied to all the subgoals generated. If T_1 proves the goal then T_2 is not applied.

Once a tactic for each variable has been calculated in this way, all possible combinations of them are produced. This is done so that the order in which the variables are processed does not determine whether the goal is proved or not.

```
Induct_on `x` >> metis_tac
>> Induct_on `y` >> metis_tac
   >> Induct_on `z` >> metis_tac

Induct_on `y` >> metis_tac
>> Induct_on `x` >> metis_tac
   >> Induct_on `z` >> metis_tac

Induct_on `y` >> metis_tac
>> Induct_on `z` >> metis_tac
   >> Induct_on `x` >> metis_tac

...
```

At this stage there is a tactic for every possible way induction can be performed over the set of variables. Finally, all combinations are turned into a single tactic by joining them using HOL's `FIRST_PROVE`.

```
FIRST_PROVE [
  Induct_on `x` >> metis_tac
  >> Induct_on `y` >> metis_tac
     >> Induct_on `z` >> metis_tac,

  Induct_on `y` >> metis_tac
  >> Induct_on `x` >> metis_tac
     >> Induct_on `z` >> metis_tac,

  Induct_on `y` >> metis_tac
  >> Induct_on `z` >> metis_tac
     >> Induct_on `x` >> metis_tac,

  ...,

  metis_tac ]
```

`FIRST_PROVE` tries each tactic in its argument list in sequence until one of them successfully proves the goal. Notice that a solitary `metis_tac` is appended at the end of the list. This is to account for the case where the goal is not quantified at all. This is ultimately the tactic used to prove an interesting goal in Hopster.

4

Results

As stated in Section 1.3, the objective of Hopster is to support the development of verified software by making HOL4 able to automatically prove properties of programs. To measure its success, the evaluation criteria are based on registering two metrics which directly impact the usability of Hopster. These metrics are the number of useful lemmas found and the running time.

The response time has been chosen to identify test cases where Hopster performs well. Just as important, it also serves to identify test cases where, although successful in proving a goal, it takes an impractical amount of time to do so. On the other hand, the number of useful lemmas found has been chosen to record the payoff for the investment in time necessary for Hopster to run. In other words, the number of useful lemmas found measures the effectiveness of Hopster while the response (or running) time measures its efficiency.

Besides registering whether Hopster completely solves the original goal submitted by the user, it may be the case that the goal is not proven but a number of useful lemmas are returned. Also, it could be that Hopster does not solve the goal nor provide useful lemmas, but the conjectures found during theory exploration turn out to be useful. All these possibilities are accounted for and the data registered in Tables 5.1 and 5.2 from which an evaluation of the results is derived.

A number of test cases have been surveyed with the aim of subjecting Hopster to a number of different scenarios. These test cases can be classified based on the theory to which they belong. Some concern the theory of lists, while others deal with trees and yet others concern the natural numbers. There is one final test case that is a more elaborate example where an attempt is made at proving the correctness of a compiler for the language of propositional logic.

The test cases for theories such as lists and trees may be regarded as small and artificial. Nonetheless, they have shown to be challenging not only for the aspects of discovering new conjectures but also for proving by induction the conjectures found [8].

All measurements were carried out on a computer with an Intel(R) Core(TM) i5-7200U CPU with 2.50 GHz and 8 GB of RAM.

4.1 The theory of lists

4.1.1 Reversing a list twice

The starting point is the motivating example developed in Section 1: the discovery of a proof that reversing a list twice gives back the original list. This can be stated as a goal in HOL:

```
∀ (xs : 'a list). reverse (reverse xs) = xs
```

To start, note that `reverse` is defined in HOL by means of `append` (concatenation of two lists):

```
val REVERSE_DEF = Define `
(reverse [] = []) /\
(reverse (h::t) = (reverse t) ++ [h])`
```

Exploring the datatypes and functions determined by the local heuristic (the `'a list` datatype and the functions `reverse` and consequently `append`) is enough to yield a satisfactory result. The process takes approximately 10 seconds to execute producing a total of 22 lemmas, among which the goal that was sought after. No further intervention from the user is required.

Using the three heuristics also proves the goal with the only difference that it takes considerably more time. This is mainly due to the additional number of functions on which lemma discovery is performed. In this case the datatypes to be explored are lists and natural numbers. The functions are: `reverse` and `append` from the local heuristic, `length`, `append` and `map` from the global heuristic and a version of `foldr` for the list datatype, as per the fold heuristic. In total, there are 2 datatypes and 5 functions to explore.

With these three heuristics, it takes around 23 seconds for Hopster to complete. The bulk of its time is spent on the proof loop, trying to prove 35 conjectures returned by QuickSpec.

Now, consider a different definition for `reverse`, called `rev`. This is a more efficient implementation of `reverse` which does not incur the quadratic runtime cost due to `append` present in the previous implementation:

```
val rev_def = Define `
(rev [] acc = acc) /\
(rev (h::t) acc = rev t (h::acc))`
```

The same goal can be expressed, albeit slightly rewritten, in terms of `rev`:

```
∀ (xs : 'a list). rev (rev xs []) [] = xs
```

When theory exploration is performed the process takes 19 seconds to complete. In this case 23 lemmas are returned. Among them, a more general version of the goal is proved:

```
∀ xs ys. rev (rev ys xs) [] = rev xs ys
```

Thus, the proof of the original goal consists of instantiating the theorem above, substituting `xs` for the empty list and rewriting the right-hand side of the equation in accordance to the definition of `reverse`.

For both test cases, `reverse` and `rev`, Hopster is successful in proving the desired result. It is interesting to see that, as noted in Section 1, Hipster is not able to prove the goal for the case of `rev`. This is due to the fact that its default tactic only performs induction on the leading universally quantified variable: `xs`. A manual attempt at a proof reveals that induction should be performed on `ys` instead.

4.1.2 The length of a list after a drop

Once more, the goal involves a few functions that work on lists. It states roughly that the length of a list after some elements have been discarded is equal to the number of elements that remain. In HOL notation:

```
∀ xs ys. length (drop (length ys) (xs ++ ys)) = length xs
```

`length` calculates the number of elements in a list, `drop` discards the specified number of elements from the front of a list and `append` is the same as in Section 4.1.1.

Hopster takes a total time of 246 seconds to complete and although the goal was indeed among the set of 42 conjectures found, it remains unproven. Yet, a successful strategy for proving this goal exists and involves induction on `ys`. This results in two subgoals, one for the base case and the other for the inductive case:

$$\forall \texttt{xs.}\ \ \texttt{length (drop (length []) (append xs [])) = length xs}$$

$$\frac{\forall \texttt{xs.}\ \ \texttt{length (drop (length ys) (xs ++ ys)) = length xs}}{\forall \texttt{y xs.}\ \ \texttt{length (drop (length (y ::\ \ ys)) (xs ++ (y ::\ \ ys))) = length xs}}$$

The base case could proceed by expanding the definitions on the left-hand side until the term equals the one the right, as follows:

```
  ∀ xs. length (drop (length []) (xs ++ [])) = length xs
≡ ∀ xs. length (drop 0 (xs ++ [])) = length xs
≡ ∀ xs. length (xs ++ []) = length xs
≡ ∀ ...
```

At this point none of the definitions apply. A cursory look suggests the following lemma:

```
∀ xs. xs ++ [] = xs
```

This simple lemma is indeed discovered and proved by Hopster. Applying it to the derivation above gives the desired result.

The inductive case follows similarly:

```
  ∀ y xs. length (drop (length (y :: ys)) (xs ++ (y :: ys)))
≡ ∀ y xs. length (drop (Succ (length ys)) (xs ++ (y :: ys)))
≡ ∀ ...
```

Once more, none of the definitions apply. After careful consideration, this auxiliary lemma emerges:

```
∀ n xs y ys. length (drop n (xs ++ (y :: ys))) =
             length (drop n ((y :: xs) ++ ys))
```

It more-or-less states that the length of a list after dropping some elements is independent of which specific elements are dropped. Although this lemma only involves a subset of the functions explored, it is not among the conjectures found by Hopster. As it happens, the size of the terms exceeds the default maximum set by QuickSpec. QuickSpec exhaustively generates all possible terms with a maximum size of 7 in its search for equational conjectures.

Fortunately, this size can be configured via the `-s` option. By allowing terms of size up to 8, the lemma is found and proven. With this in place the derivation can proceed culminating in the proof of the original goal. Running Hopster with this setting takes over 11 minutes to complete. The necessary lemma is found and consequently the goal is successfully proved.

4.2 The theory of trees

4.2.1 Flattening trees

In this test case we explore a theory of binary trees of the form:

```
Datatype `hoptree = Leaf | Node hoptree 'a hoptree`
```

This datatype defines binary trees whose interior nodes carry a value of type `'a`.

An attempt is made to prove the following goal:

```
∀ t. inorder (mirror t) = reverse (inorder t)
```

The goal states that mirroring a tree and then flattening it produces a list which is equivalent to flattening the tree and then reversing the resulting list.

Flattening a tree refers to the operation of producing a list from a tree, such that a value is an element of the list if and only if it is also an element of the tree. The order of the elements in the resulting list is the property that distinguishes the different flattening methods.

In this particular case, the list resulting from `inorder` has the value at the root between the lists that result from flattening the left and right subtrees. The function `mirror` recursively switches its left and right subtrees, thus performing a mirroring effect. `reverse` has been treated in Section 4.1.

In a sense, this goal is analogous to the one presented in Section 4.1.1, only this time involving to trees. From the perspective of theory exploration, it deals with a superset of the types and functions which were explored in the case of lists. Another thing that relates this test case to that of reversing lists is that the lemma

```
∀ xs. reverse (reverse xs) = xs
```

is useful for proving the goal.

Hopster takes approximately 311 seconds to generate 251 lemmas and 9 conjectures. As was the case presented in Section 4.1.1, the goal is found and proved in a reasonable time.

4.2.2 Functional arrays

A long standing problem in functional programming languages is how to support the concept of arrays[1, 28]. An array is a data structure able to store a set of elements where each element is accessible by using an index. Imperative languages leverage how memory is laid out in current computer architectures and set a contiguous region where the elements are stored one after the other. An index into the array is used as an offset to the memory address where the array is stored. Once an element is located, it can simply be updated in-place, replacing the previously stored value with a new one. On common computer architectures these operations can be performed in constant time.

Efficient arrays are not straight-forward to implement in functional programming languages. In particular, updating an element in-place may break the principle of referential transparency. Many different methods have been created to overcome this so called array updating problem. Here, a naive implementation is developed where a binary tree is used to store the elements. When an element of the array is updated a whole new tree is generated.

In Figure 4.1 a graphical representation of an array of size seven is shown:

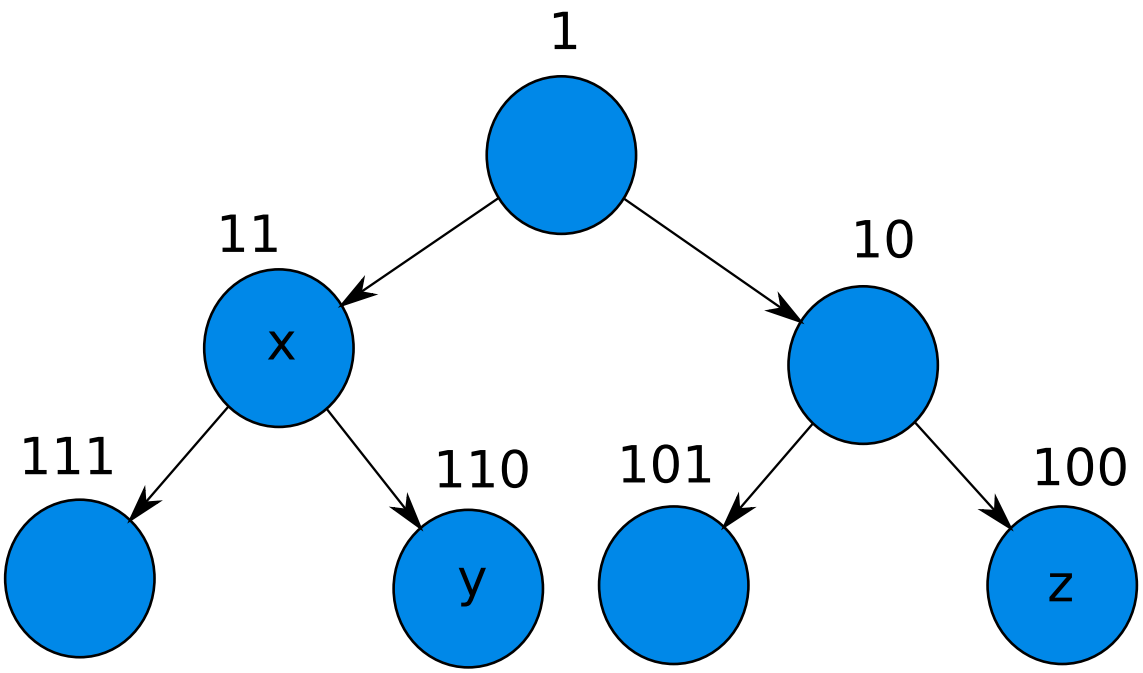

Figure 4.1: Graphical representation of a functional array

The nodes of the tree hold the values of the array. Figure 4.1 depicts a tree holding three elements: x, y and z. Each node in the tree is annotated by an index, written in binary form. Consequently, the trees are balanced.

To lookup an element in the array, the index is examined (after conversion into binary) starting from its least significant digit upwards. If the digit under consideration is 1 the left child node is selected, otherwise the right. This process continues until the most significant digit is reached at which point the node under consideration is said to hold the element that was sought.

A binary tree like the one described is represented in HOL thus:

```
Datatype `array = leaf | node array ('a maybe) array`
```

The values on the nodes of the trees have a `maybe` type, whose values are defined as:

```
Datatype `maybe = None | Just 'a`
```

If a node of the tree (i.e. a position in the array) is empty, it is signified by `None` otherwise the value is contained within a `Just`.

A verification that this implementation of arrays is correct requires a proof of the following property:

```
∀ v t n. lookup (update v t n) n = Just v
```

There are 4 datatypes involved during theory exploration: `array` and `maybe` as was mentioned above, `list` and `bool` to represent the numbers as as list of digits in binary and lastly `oddeven` which is part of a theory of natural numbers of the form 1, $2n + 1$ or $2n + 2$:

```
Datatype `oddeven = One | Odd oddeven | Even oddeven`
```

This representation was chosen for its ease of transformation into a list of binary digits.

The functions selected by the local heuristic are `lookup` and `update` as mentioned on the goal, `num2boolList` the function to transform the index into a list of binary digits and some auxiliary functions. The global heuristic is not applicable since the theory was prepared to prove this particular goal, no additional lemmas exist. The fold heuristic adds two fold functions, one for `array`s and another for `oddeven` numbers. In total there are 11 functions to explore.

The overall process takes more than 10.5 hours to complete, a considerable time. Nonetheless, the goal and its necessary lemma are successfully proved among the set of 373 conjectures discovered.

4.3 The theory of natural numbers

The next two goals are based on an axiomatic formalization of the natural numbers known as Peano arithmetic. It postulates the existence of the number 0 (represented as `Zero` in HOL) and a successor function (`Succ`) which is used to generate the rest of the numbers. It can be defined in HOL thus:

```
Datatype `nat = Zero | Succ nat`;
```

Based on this representation, the addition of two `nat` numbers is simple enough to define:

```
val add_def = Define `
  add Zero n = n /\
  add (Succ m) n = Succ (add m n)`;
```

Subtraction, on the other hand, is not as simple. Depending on the arguments, the result may be lower than zero which is not member of the set of natural numbers.

Thus, the definition is extended so that subtracting any natural number from `Zero` yields `Zero`:

```
val sub_def = Define `
  sub m Zero = m /\
  sub Zero n = Zero /\
  sub (Succ m) (Succ n) = sub m n`;
```

`sub` is an example of a function which is defined recursively over both of its arguments. This fact makes proving goals which involve `sub` be not as straightforward, as the next Sections show.

4.3.1 Subtracting a bigger number from a smaller one

Consider the following goal:

```
∀ m n. sub n (add n m) = Zero
```

It is a statement of the fact that subtracting a number bigger or equal from another results in `Zero`.

The total number of datatypes to explore is one (the Peano numbers) which involves two value constructors (`Zero` and `Succ`). Meanwhile, the number of functions comprises: `add` and `sub` from the local heuristic; `add`, `leq` (the lower or equal than operator) and `mul` (the multiplication operator) from the global heuristic; and (`foldnum` (the fold over natural numbers) from the fold heuristic. In total, 9 functions over which to perform theory exploration.

A manual proof attempt shows it is possible to prove the validity of the goal by structural induction on `n`, provided that the following lemma is discovered and proved:

```
∀ n. sub Zero n = Zero
```

The lemma refers to `sub` which is already selected among the set of functions to explore. Furthermore, the size of the terms on each side of the equation is lower than seven, the maximum allowed by QuickSpec in its default settings. This provides strong indication that the lemma will be found during theory exploration. Its successful proof implies the validity of the original goal.

Hopster takes a total time of approximately 1.6 hours to complete. Note that the global and fold heuristics do not provide any useful lemmas. So, their inclusion only contributes in delaying further the running time of Hopster, in large part because of the higher number of conjectures that are discovered with the impact on the execution of the proof loop. A version of Hopster with only the local heuristic is able to find a proof of the goal in under 11 minutes.

4.3.2 Subtracting twice

The following goal provides a law that governs expressions made up of two consecutive subtractions:

```
∀ m n p. sub (sub m n) p = sub m (add n p)
```

The law states that subtracting two number in succession is equal to a single subtraction of their sum.

For this test case, `add` is defined recursively on the second argument, thus:

```
val add_def = Define `
  add m Zero = m /\
  add m (Succ n) = Succ (add m n)`;
```

A possible proof of this goal reveals a different structure from the one in the preceding Section. The overall tactic employed is by induction on m. Proving the base case requires the same auxiliary lemma:

```
∀ n. sub Zero n = Zero
```

The inductive step can proceed by a case analysis on n and is reliant on two other lemmas:

```
∀ n. add Zero n = n
∀ n. add (Succ m) n = Succ (add m n)
```

Both equations amount to the definition of `add` by recursion on the first argument.

Hopster takes slightly more than 1.5 hours to prove the goal. Surprisingly, the two lemmas mentioned are not discovered. Instead, QuickSpec discovers that `add` is commutative and the lemmas follow from this property and `add`'s own definition. In total 539 lemmas were found from a set of 606 conjectures found during theory exploration.

4.4 The correctness of a small compiler

The last test case offers an example where Hopster is not able to completely prove a goal because a required lemma involves a function which has not been selected for exploration by the three heuristics. The goal is to prove the correctness of a compiler that translates propositional logic into instructions for a simple stack-based abstract machine.

The language of propositional logic, as defined below, is a small extension to one presented in [19]:

```
val _ = Datatype `
lang = Const bool
     | Var var
     | Not lang
     | And lang lang
     | Or lang lang
     | Implies lang lang`
```

The set of terms is defined recursively and it can be described with only a few syntactic forms. There are constants, variables, a unary operation that negates a propositional term and three binary operations that correspond to conjunction, disjunction and implication in propositional logic. Constant values can be either true or false (`T` or `F` as defined in HOL's `min` theory). Variables are defined as an enumeration of one-letter values:

```
val _ = Datatype `
var = X | Y | Z`
```

The proof of correctness for the compiler is done in accordance to the operational semantics approach. This approach dictates that at least two different abstract machines be provided each describing a different way for how terms in the language evaluate to values (i.e. the semantics of the language).

By necessity, one of the abstract machines should be straight-forward to construct. Its instructions should correspond directly to the possible terms of the language, as opposed to being based on instructions for some low-level machine. For our specific test case, this simple abstract machine is an interpreter that defines how to reduce any a propositional formula to a Boolean value:

```
val eval_def = Define `
eval env (Const b) = b /\
eval env (Var x) = env x /\
eval env (Not e) = ~ (eval env e) /\
eval env (And e1 e2) =
    (eval env e1 /\ eval env e2) /\
eval env (Or e1 e2) =
    (eval env e1 \/ eval env e2) /\
eval env (Implies e1 e2) =
    (eval env e1 ==> eval env e2)`
```

To evaluate an arbitrary propositional formula the values for all its variables need to be known. For this reason, `eval` is parameterized with an environment that maps variables to values. Other than this, the interpreter is a straight-forward mapping of terms in the propositional logic language to HOL Boolean values.

The reason that the interpreter needs to be simple is that it is trusted to be correct. Then, the correctness of a more elaborate abstract machine (i.e. one that operates with a different instruction set, closer to a real machine) is established by proving that any term in the language produces the same value for both abstract machines.

In contrast to the interpreter, a definition for a different abstract machine is presented whose programs are not terms in the language of propositional formulas, but a sequence of boolean-valued instructions held on a stack. The set of possible instructions is defined by:

```
val _ = Datatype `
instr = PUSH bool
      | LOAD var
      | NOT
      | AND
      | OR
      | IMPL`
```

A program for this abstract machine is thus represented as a list of these instructions. Given an environment to determine concrete values for variables and a stack,

any program can be run to yield a final stack:

```
val exec_def = Define `
exec env [] s = Just s /\
exec env (PUSH b :: is) s = exec env is (b :: s) /\
exec env (LOAD x :: is) s = exec env is (env x :: s)  /\
exec env (NOT :: is) (b :: s) = exec env is (deny b :: s) /\
exec env (AND :: is)
         (b1 :: b2 :: s) = exec env is (conj b1 b2 :: s) /\
exec env (OR :: is)
         (b1 :: b2 :: s) = exec env is (disj b1 b2:: s) /\
exec env (IMPL :: is)
         (b1 :: b2 :: s) = exec env is (impl b1 b2 :: s) /\
exec _ _ _ = None`
```

In order to make the function total, the result has a type `'a maybe`:

```
val _ = Datatype `
maybe = None | Just 'a`;
```

If the program runs successfully, a stack is returned wrapped inside the value constructor `Just`. It may be the case where an operation, requires more arguments than the number of elements pushed onto the stack. In such cases, the program is in an error state and the value `None` is returned.

At last, we can define the compilation process that compiles the language of propositional logic into a program for this stack-based abstract machine:

```
val comp_def = Define `
comp (Const b) = [PUSH b] /\
comp (Var x) = [LOAD x] /\
comp (Not e) = comp e ++ [NOT] /\
comp (And e1 e2) = comp e2 ++ comp e1 ++ [AND] /\
comp (Or e1 e2) = comp e2 ++ comp e1 ++ [OR] /\
comp (Implies e1 e2) = comp e2 ++ comp e1 ++ [IMPL]`
```

With everything in place, the goal of proving the correctness of the compiler can be stated as follows:

```
∀ p env. exec env (comp p) [] = Just (eval env p)
```

Although for the purposes of proving it, a more general version is preferred:

```
∀ p env s. exec env (comp p) s = Just (eval env p :: s)
```

That is, the goal is generalized to an arbitrary stack. The original goal follows by simply instantiating `s` to the empty list.

A manual proof attempt uses the same strategy as the case study on correct compilers in [33]. The goal is disposed of by induction on the size of `p`, an instance of a formula in propositional logic. The two base cases, when the term is just a constant or a variable, can be solved by rewriting the calls to `comp`, `exec` and `eval` by their definitions. On the other hand, the four inductive cases corresponding to negation, conjunction, disjunction and implication cannot be disposed of so easily. They all follow the same difficulty and pattern of proof. The case for conjunction

can be stated as:

$$\frac{\begin{array}{c}\forall \texttt{env s. exec env (comp p) s = Just (eval env p :: s)} \\ \forall \texttt{env s. exec env (comp p') s = Just (eval env p' :: s)}\end{array}}{\forall \texttt{env s. exec env (comp (And p p')) s = Just (eval env (And p p')::s)}}$$

So, assuming the property is true for two formulas, `p` and `p'`, a proof is shown for a formula bigger than both of them: that which corresponds to their conjunction. A pen-and-paper proof could proceed by transforming the left side of the equality into the right side by expanding the definitions:

```
  ∀ env s. exec env (comp (And e1 e2)) s
≡ ∀ env s. exec env (comp e1 ++ comp e2 ++ [AND]) s
≡ ∀ ...
```

At this point the expression cannot be expanded any further and an auxiliary lemma is required. After some consideration, the lemma which states how `exec` distributes over the append operator `++` presents itself:

```
∀ env x y s. exec env (x ++ y) s = exec env y (exec env x s)
```

Executing code of the form `x ++ y` is equal to executing `x` and then executing y with the resulting stack. Although the general idea is promising, the above formula is not correct. This is because the result of `exec` is not a stack, but "maybe a stack": `(inst list) maybe`. The lemma that is needed should express that `exec env y` is to be run, only if `exec env x` runs successfully. This is exactly how the monadic bind operator works. Its definition for `'a maybe` types is:

```
val bind_def = Define `
bind None (f : 'a -> 'b maybe) = None /\
bind (Just x) f = f x`
```

So the required lemma may be expressed as:

```
∀ env x y s. exec env (x ++ y) s
             = bind (exec env x s) (exec env y)
```

The lemma itself may be proven with induction on `x` followed by rewriting using the definitions of the functions involved. The proof of the case for conjunction (and the other inductive cases) proceeds by simple expansion of the definitions of the functions involved. Thus, having proved the base and inductive cases, the original goal follows also.

With the aid of the lemma above, only induction and rewriting are necessary to show the validity of the goal. It would seem reasonable to assume that Hopster would be able to find and prove it. Unfortunately this is not the case because the necessary lemma is not discovered due to the fact that the bind operator is not among the set of functions selected for exploration.

Six datatypes are explored: `lang`, `var`, `instr`, `'a list`, `bool` and `'a maybe`.

The local heuristic determines eight functions: `exec`, `comp`, `eval` from the current theory presented. The functions `deny`, `conj`, `disj` and `impl` use in the evaluator, coming from the theory of booleans. Lastly, `append` from the theory of lists. The global heuristic is not applicable for this theory since it has been built for the sole purpose of proving the stated goal. The fold heuristic includes the function `foldlang`

which represents the fold for expressions of propositional logic. In total there are nine functions over which to perform theory exploration.

This test case which takes the longest to complete, with a total of 6 hours and 14 minutes. Some 1,019 conjectures are discovered. The goal (the general version with an arbitrary stack) is found among the conjectures discovered. But, as mentioned earlier, the proof loop is not able to prove it. The reason for this is that the necessary lemma is never discovered. This in turn is due to the fact that `bind` is not among the set of explored functions.

5

Conclusion

5.1 Evaluation

The success of the implementation depends on its performance as measured by the two metrics chosen. The first metric measures the number of useful lemmas found and is known as the effectiveness metric for it is a measure of how close Hopster gets to solving a goal. The second metric measures the running time and is referred to as the efficiency metric.

The Tables below feature four versions of Hopster. Each version is identified by a two-letter code. If the first letter is an **H** then it means that all the three heuristics for the selection of constructs to explore are used, otherwise a letter **N** is used which indicates that only the local heuristic is employed. The second letter may be either **P** or **N**. When **P** is used it means that QuickSpec reduces the number of conjectures returned by pruning those for which the terms on each side of the equality can be made equal using rewriting alone. This feature is implemented in QuickSpec (passing the `-p` argument to `tip-spec`) and pruning happens before Hopster imports the conjectures back to run the routine for easy reasoning. If instead of **P** an **N** is used then all conjectures found by QuickSpec are returned. In summary, the four versions are:

Hopster NN only the local heuristic is used. All conjectures found by QuickSpec are passed to the proof loop.

Hopster HN all heuristics as used. All conjectures found are passed to the proof loop. This is the version of Hopster for which the test cases were developed in Chapter 4.

Hopster NP only the local heuristic is used. Conjectures found via theory exploration are pruned.

Hopster HP all heuristics are used. Conjectures found via theory exploration are pruned.

Table 5.1 collects data corresponding to the effectiveness metric. Each column shows data for one version of Hopster. There are nine test cases shown and each is annotated with the number of lemmas that are needed to prove the goal. This number was obtained by performing a manual proof using induction and equational reasoning. Appendix B documents the goal that is proved for each test case, for easy reference.

For each test case and Hopster version, the following information is recorded:

Goal's lemmas records the number of lemmas found that are useful for proving the goal. When this number is less than the number of lemmas needed, it

means that the goal was not proven for that particular version of Hopster.
Total lemmas counts the total number of lemmas which were discovered an proven.
Total conjectures counts the total number of conjectures which were discovered.

Table 5.1: Experimental data for the effectiveness metric

Effectiveness/Hopster version	Hopster NN	Hopster HN	Hopster NP	Hopster HP
`REVERSE`: 3 lemmas				
Goal's lemmas	3	3	3	3
Total lemmas	22	35	8	16
Total conjectures	22	35	8	16
`REV`: 1 lemma				
Goal's lemmas	1	1	1	1
Total lemmas	6	23	5	16
Total conjectures	6	23	5	16
`LENGTH_DROP`: 2 lemmas				
Goal's lemmas	2	2	1	1
Total lemmas	84	89	13	17
Total conjectures	93	100	15	18
`MIRROR`: no lemmas				
Goal's lemmas	0	0	0	0
Total lemmas	7	119	4	37
Total conjectures	7	225	4	39
`INORDER_MIRROR`: 3 lemmas				
Goal's lemmas	3	3	3	3
Total lemmas	32	251	17	49
Total conjectures	32	260	17	52
`ARRAY`: 1 lemma				
Goal's lemmas	1	1	1	1
Total lemmas	42	291	16	80
Total conjectures	63	373	26	118
`SUB_BIGGER`: 1 lemma				
Goal's lemmas	1	1	1	1
Total lemmas	234	539	27	84
Total conjectures	240	606	28	114
`SUB_TWICE`: 2 lemmas				
Goal's lemmas	2	2	2	2
Total lemmas	234	539	26	81
Total conjectures	240	606	28	114
`COMPILER`: 2 lemmas				
Goal's lemmas	1	1	1	1
Total lemmas	1004	1404	498	696
Total conjectures	1019	1503	505	714

An observation which applies to all versions of Hopster is that the combination of theory exploration with the proof loop is indeed effective for proving many properties of a functional program. It follows and is shown in the rows under "Goal's lemmas" that almost all required lemmas were discovered and proved as well. Hopster NN and Hopster HN, that do not prune conjectures, have a success rate of 92%. On the

other hand, Hopster NP and Hopster HP have a slightly lower success rate of 83%. The difference stems from that Hopster NP and Hopster HP are unable to prove a lemma for `LENGTH_DROP`. None of the versions is able to prove `COMPILER`.

`COMPILER` remains unproved because a required lemma involves the `bind` operator which is not selected for exploration. Since `bind` is not present in the statement of the goal, it is not among the functions selected by the local heuristic. This precludes the discovery of the lemma for versions of Hopster that rely exclusively on the local heuristic. These are Hopster NN and Hopster NP.

If `COMPILER` were universally quantified by a variable with `'a maybe` type then the global heuristic could have selected `bind`, provided it was among the most used functions in the theory. Finally, `bind` may be expressed as a fold for `'a maybe` thus:

```
fold f None m = bind m f
```

so the lemma can be expressed as:

```
∀ env x y s. exec env (x ++ y) s
             = fold (exec env y) None (exec env x s)
```

But then again, there is no universally quantified variable in the goal with `'a maybe` type. Thus, the `fold` for `'a maybe` is not selected for exploration. This precludes the discovery of the lemma by versions of Hopster which rely on all the heuristics, namely Hopster HN and Hopster HP.

The other test case that remains unproven is `LENGTH_DROP`. The goal is not proved on versions of Hopster that prune conjectures which follow from those previously found. The necessary lemma:

```
∀ n xs y ys. length (drop n (xs ++ (y :: ys))) =
             length (drop n ((y :: xs) ++ ys))
```

is found by QuickSpec, but is subsequently discarded. This is because it follows from three conjectures which are found before:

```
length [] = zero
∀ m n xs. drop (Succ m) (n :: xs) = drop m xs
∀ n xs. drop (Succ (Succ (Succ n))) xs = []
```

The first two equations are valid. Indeed, both are definitional clauses of `length` and `drop`. The third one is interesting, because it is false.

As part of the process that decides whether to discard a newly generated conjecture, QuickSpec augments the set of current equations with others in a process known as Knuth-Bendix completion [32]. An informal explanation of the Knuth-Bendix completion algorithm is found in Section 5.2.2. For `LENGTH_DROP`, the three equations above are augmented with an extra one:

```
∀ n xs. drop n xs = []
```

So, put together these four equations are enough to show that the lemma follows by rewriting alone:

```
  ∀ n xs y ys. length (drop n (xs ++ (y :: ys)))
≡ length []
≡ zero
```

```
  ∀ n xs y ys. length (drop n ((y :: xs) ++ ys))
≡ length []
≡ zero
```

Therefore the required lemma is discarded. Later, the proof loop is unable to prove the goal. In particular because as was shown above, the third equation found by QuickSpec is false. This test case shows that any benefit of pruning should be weighed against the loss in effectiveness when there are false equations among the set of equations discovered.

The columns "Total lemmas" and "Total conjectures" show where the versions of Hopster differ. Hopster HN is able to discover and prove more conjectures than Hopster NN. The same happens when comparing Hopster HP to Hopster NP. The global and the fold heuristics give Hopster HN and Hopster HP better exploratory power. But, for the purposes of proving a specific goal, the local heuristic by itself is sufficient. No test case has been found where the local heuristic fails to select the appropriate functions but the global and/or fold heuristics does.

The global heuristic is based on the assumption that for any conjecture the most occurring functions in the theories involved are likely to play a role in its proof. It may be the case that this only happens for the functions that directly appear in the statement of the goal, making the local heuristic sufficient while canceling any contribution that the global heuristic may bring. This observation may also apply to the fold heuristic. Additionally, lemmas expressed using fold are generally bigger than they would otherwise be. For instance, the lemma for `COMPILER` shown above has its term on right-hand side of the equation with a size of 9. Using `bind` the term has a size of 8.

Still, the global and fold heuristics bring extra exploratory power without deterring the effectiveness when proving a specific goal. So, *barring any degradation in performance* there is evidence to support their inclusion.

Table 5.2 measures data that corresponds to the efficiency metric. The layout is the same as before: There is a column for each of the four different versions of Hopster. Additionally, the table is split in nine sections for each of the nine test cases.

Inside each test case, the rows have the following meaning:

Routine reasoning refers to the time (in seconds) it takes to prove easy conjectures.

Difficult reasoning is the time (in seconds) it takes to finish the proof loop.

QuickSpec is the time (in seconds) to perform theory exploration.

Total Time (in seconds) is the running time of the whole process.

Table 5.2: Experimental data for the efficiency metric

Efficiency/Hopster version	Hopster NN	Hopster HN	Hopster NP	Hopster HP
		`REVERSE`		
Routine reasoning	0.078	0.099	0.029	0.040
Difficult reasoning	3.636	13.792	2.793	8.717
QuickSpec	6.882	9.655	5.302	8.766
Total time	10.596	23.546	8.124	17.523
		`REV`		
Routine reasoning	0.032	0.066	0.022	0.041
Difficult reasoning	0.183	10.112	0.110	8.830
QuickSpec	5.984	8.826	5.326	8.535
Total time	6.199	19.004	5.458	17.406
		`LENGTH_DROP`		
Routine reasoning	0.200	0.258	0.070	0.080
Difficult reasoning	259.688	652.990	168.014	171.630
QuickSpec	8.554	19.912	7.234	10.282
Total time	268.242	673.160	175.318	181.992
		`MIRROR`		
Routine reasoning	0.039	0.642	0.018	0.102
Difficult reasoning	0.108	107.515	0.023	26.340
QuickSpec	5.818	12.857	5.265	8.891
Total time	5.965	121.014	5.306	35.333
		`INORDER_MIRROR`		
Routine reasoning	0.100	0.776	0.043	0.133
Difficult reasoning	6.831	294.583	5.892	42.006
QuickSpec	7.648	15.708	6.866	10.457
Total time	14.579	311.067	12.801	52.596
		`ARRAY`		
Routine reasoning	0.207	1.479	0.071	0.527
Difficult reasoning	3294.581	38013.575	2351.083	2177.946
QuickSpec	11.785	49.822	8.094	35.930
Total time	3306.573	38064.876	2359.248	3257.286
		`SUB_BIGGER`		
Routine reasoning	0.758	2.003	0.073	0.317
Difficult reasoning	643.506	5748.521	111.692	2177.946
QuickSpec	10.842	28.196	6.673	17.064
Total time	655.106	5779.720	118.434	2195.327
		`SUB_TWICE`		
Routine reasoning	0.590	2.141	0.072	0.324
Difficult reasoning	655.190	5666.405	122.045	1949.877
QuickSpec	11.230	28.068	6.692	17.378
Total time	667.010	5696.614	128.809	1967.579
		`COMPILER`		
Routine reasoning	3.850	5.501	1.850	2.609
Difficult reasoning	6801.134	22256.299	4095.048	4659.049
QuickSpec	46.678	187.682	36.661	157.420
Total time	6851.662	22449.482	4133.559	4819.078

In all versions of Hopster the proof loop takes the largest portion of the total time expended in proving a goal. At the lower end when the number of conjectures to prove is small (such as for the case of `REVERSE` and `REV`) the proof loop accounts for 20% of the total time for Hopster NN, 55% for Hopster HN, 18.5% for the case of Hopster NP and 50% for Hopster HP; roughly half the time for versions of Hopster which use all the heuristics while around 20% for those that only rely on the local heuristic. As the number of conjectures increases the proof loop quickly takes 98% of the time in all versions.

In cases such as `COMPILER` the running time proves excessive. As mentioned in Section 4.4 Hopster HN takes approximately 6 hours to complete. This time is reduced significantly in Hopster NN, taking just under 2 hours to complete. The same effect can be observed between Hopster HP and Hopster NP. So, even though the global and fold heuristics were at first considered successful under the effectiveness metric, the gains in exploratory power should be contrasted with the penalty they exact on efficiency.

In conclusion, the version of Hopster which strikes the best balance between efficiency and effectiveness for proving a single goal is Hopster NP. For this scenario, the contribution of the global and fold heuristics is unwarranted. Their impact is negative because of the extra time incurred. Interesting lemmas were not discovered by their use alone and the local heuristic seems enough to provide the necessary lemmas needed to prove a goal.

5.2 Suggestions for further work

It follows from the previous Section that improving the running time of the proof loop will have the greatest impact on the overall running time of Hopster. The increase in performance could in turn allow the exploration of larger numbers of functions and datatypes. The ultimate consequence of this, is that the exploration of more functions and datatypes is more likely to forge the necessary lemmas.

5.2.1 Detecting impossible conjectures

The set of conjectures returned by QuickSpec can be divided into two subsets: one comprised by conjectures which can be proven and the other by those conjectures that cannot. Membership to one of these subsets is determined by the strategy used to prove them. In the case of Hopster, we name this strategy the proof loop. The effectiveness of the proof loop rests partly on the lemmas that have already been proven at any one time.

Among the members of the set of conjectures which cannot be proven there is a subset of conjectures which would not be proven even if all others were to be used as lemmas. Let us call this subset the “impossible conjectures”. This Section considers a possible improvement based on the quick detection of impossible conjectures.

While the aim of Hopster is to arrive at a proof of as many conjectures as possible, investing time in detecting impossible conjectures could drastically improve the running time of the proof loop. Impossible conjectures would never be proven and on each iteration they are incurring the highest time.

This feature could be implemented as a preprocessing stage, before the routine for difficult reasoning initiates. For instance, it could be launched after the detection of easy to prove conjectures, a process described in Section 3.4.1.

The process of detecting "impossible conjectures" is illustrated in Algorithm 1.

Algorithm 1 Algorithm for the detection of "impossible conjectures"

Input: a list of conjectures C and a list of function definitions F
Output: a list of provable conjectures P and a list of "impossible conjectures" I

```
M = [ ]
for c such that c ∈ C do
  Add to M the pair c and mk_thm(x) such that x ∈ C ∧ x ≠ c
end for
I = [ ]
P = [ ]
for m such that m ∈ M do
  (goal, lemmas) = m
  L = lemmas ∪ F
  tac = FIRST_PROVE [EASY_TAC L, HARD_TAC L]
  if goal is not proven using tac then
    Add to I
  else
    Add to P
  end if
end for
return (P, I)
```

The process takes two arguments: the set of conjectures found via theory exploration and the definitions of the functions mentioned in the conjectures. The function definitions are required by the tactic in its attempt to prove the conjectures by rewriting the terms mentioned in them. The result of the process is a pair comprised by the conjectures which were proved on one hand, and the impossible conjectures on the other. Once integrated into Hopster, the conjectures that were proved conform the set that is passed to the routine for difficult reasoning while the impossible conjectures are separated and presented only at the end, as interesting conjectures found.

The first step in the process considers each conjecture in turn. It pairs each conjecture with all the rest, which are turned into lemmas by means of the `mk_thm` function. `mk_thm` when applied to a goal is able to construct a theorem from an arbitrary goal. Its use is discouraged in general since it can result in the creation of theorems that lead to contradictions but its use in this procedure (and as part of Hopster) is safe as none of the conjectures proven are used as lemmas in subsequent steps. Each pair of conjecture/lemmas is stored in a new set named M.

Then an attempt to prove each conjecture in M follows using the HOL tactic `FIRST_PROVE [EASY_TAC` L`, HARD_TAC` L`]`. `EASY_TAC` is the tactic used for easy reasoning, as described in Section 3.4.1. `HARD_TAC` is the tactic used for difficult reasoning, as described in Section 3.4.2. Both tactics are combined by `FIRST_PROVE`

which tries applying each tactic it is passed in sequence until one proves the goal. If the conjecture is proven it is added to the set P of conjectures which are provable; otherwise it is added to the set I of impossible conjectures. The process ends returning the sets I and P.

In essence, Algorithm 1 resembles one iteration of the proof loop. It is hoped that the gains in performance on each iteration of the proof loop will make up for the time it takes to run this process.

5.2.2 Knuth-Bendix completion

Instead of proposing a separate stage, the following suggestion attempts to improve on the routine for difficult reasoning. The improvement consists in augmenting the set of equational lemmas (the function definitions selected by the heuristics) with other equations such that the set as a whole becomes a confluent term-rewriting system. The process which is able to do this is known as the Knuth-Bendix completion algorithm. If this algorithm succeeds the resulting set of equations can be used to prove whether the goal is valid much faster than the equational reasoning performed by `metis_tac`.

The Knuth-Bendix completion algorithm starts with a set of equations between terms and augments it with more equations until the set turns into a confluent term-rewriting system. A rewriting system is a set of equations where a term on one side of every equation (i.e. the left-hand side) is considered more "complex" than the other (the right-hand side). Moreover, a rewriting system is said to be confluent if given a term t for which two (or more) reduction rules apply (i.e. it has the form of the left-hand side terms of two reductions) leading to different terms t_1 and t_2. When this happens t_1 and t_2 eventually converge to a common term s by successive application of other reduction rules [2].

For an informal explanation of how the algorithm works consider the equation $a^{-1}(ab) = b$. This equation is a reduction, therefore anywhere a term is found with the form $a^{-1}(ab)$ it can safely be replaced by b. The algorithm produces new reductions by analyzing the initial set of reductions in pairs, $\alpha_1 = \beta_1$, $\alpha_2 = \beta_2$ and creating a new term which has the form of α_1 and one of its subterms has the form of α_2. So the subterm can be replaced by β_2, and if the result is equated to β_1 a new reduction is formed. Following the initial example, let us consider the reduction $a^{-1}(ab) = b$ with itself so that $\alpha_1 = \alpha_2 = a^{-1}(ab)$ and $\beta_1 = \beta_2 = b$. The term $((x^{-1})^{-1}(x^{-1}(xy))$ keeps the form of α_1 but it also has a subterm $x^{-1}(xy)$ with the form of α_2. So, $((x^{-1})^{-1}(x^{-1}(xy))$ is equal to $(x^{-1})^{-1}y$ by replacing the subterm with β_2, but is also equal to xy by replacing the whole term with β_1 giving a new reduction $(x^{-1})^{-1}y = xy$ [23]. The new reduction can be added as a theorem in HOL by a simple application of the rewriting tactic `REWRITE_TAC`.

Once the new reduction is constructed it is judged whether to include it in the original set of reductions. If the two terms that constitute the new reduction can be proven equal as a consequence of applying the reduction rules then it is discarded otherwise the new reduction is added to the set. When this happens, the set has a new member and new pairs can be analyzed so the process iterates. In the previous example, the two terms in the reduction $(x^{-1})^{-1}y = xy$ cannot be proven equal

(they are in normal form with respect to the original set of reduction rules) so the equation is considered interesting and added to the set.

It may be that the Knuth-Bendix algorithm fails when for a new reduction it is impossible to determine which of its constituent terms is more "complex" than the other. For example, if a commutativity rule is arrived at, such as $x + y = y + x$, it may be impossible to determine which of the terms in either side is more "complex". Both terms are the same up to the renaming of variables. Furthermore, the algorithm is not guaranteed to terminate in which case it will keep adding new reductions indefinitely. To ensure termination the algorithm could be stopped when the generated terms reach a certain size or after a maximum amount of time is reached.

Whether the process fails or it is stopped for diverging, the original set of equations is still augmented with new reduction rules. This increments the power of `metis_tac`, the tactic used for equational reasoning.

If the algorithm terminates successfully then the resulting set of reduction rules can be used to prove any goal provided its terms are built with the same functions and datatypes as the terms in the set of reductions. Since the first heuristic makes sure to include all the functions and datatypes mentioned in the goal, the goal is in fact a term that fulfills this requirement. Therefore the goal can be proved without resorting to the most time consuming component of Hopster: the tactic for equational reasoning.

5.3 Final remarks

This work set out to integrate the technique of theory exploration (as provided by the TIP tools) with the theorem prover HOL4. The work involved the translation of constructs of the HOL object language into a language suitable for QuickSpec, a search of proper heuristics to select the most suitable functions that yield useful lemmas via theory exploration, and a study of the strategy used for proving the conjectures discovered.

Initial results show the proposed method to be effective for problems of moderate size. For the instances where the method is unsatisfactory, the principal causes were identified and followed with suggestions for further improvements.

It is to be noted that the exploratory power of QuickSpec results in many interesting conjectures, among which, the ones necessary for proving a specific goal using induction. Furthermore, HOL4 is a suitable theorem prover capable of turning the set of conjectures discovered into useful lemmas. Upon further improvements in its running time, the combination of HOL4 and QuickSpec emerges as a useful tool for the construction of verified functional programs.

Bibliography

[1] Annika Aasa, Sören Holmström, and Christina Nilsson. An efficiency comparison of some representations of purely functional arrays. *BIT Numerical Mathematics*, 28(3):489–503, Sep 1988.

[2] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge university press, 1999.

[3] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *International conference on interactive theorem proving*, pages 131–146. Springer, 2010.

[4] B. Buchberger. Mathematica: Doing Mathematics by Computer? In A. Miola and M. Temperini, editors, *Advances in the Design of Symbolic Computation Systems*, pages 2–20. Springer Vienna, 1997. RISC Book Series on Symbolic Computation.

[5] Bruno Buchberger. Algorithm invention and verification by lazy thinking. In *Proceedings of SYNASC*, pages 2–26, 2003.

[6] Bruno Buchberger and Adrian Craciun. Algorithm synthesis by lazy thinking: Using problem schemes. In *Proceedings of SYNASC*, pages 90–106, 2004.

[7] Bruno Buchberger, Tudor Jebelean, Temur Kutsia, Alexander Maletzky, and Wolfgang Windsteiger. Theorema 2.0: Computer-Assisted Natural-Style Mathematics. *JFR*, 9(1):149–185, 2016.

[8] Alan Bundy. The automation of proof by mathematical induction. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*. MIT Press, 04 1999.

[9] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

[10] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[11] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Hipspec: Automating inductive proofs of program properties. In *ATx/WInG@ IJCAR*, pages 16–25, 2012.

[12] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In *International Conference on Automated Deduction*, pages 392–406. Springer, 2013.

[13] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. TIP: Tons of inductive problems. In *Conferences on Intelligent Computer Mathematics*, pages 333–337. Springer, 2015.

[14] Lucas Dixon and Jacques Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In *International Conference on Automated Deduction*, pages 279–283. Springer, 2003.

[15] Mike Gordon. From LCF to HOL: a short history. In *Proof, Language, and Interaction*, pages 169–186, 2000.

[16] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem-Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.

[17] John Hughes. The design of a pretty-printing library. In *International School on Advanced Functional Programming*, pages 53–96. Springer, 1995.

[18] Joe Hurd. First-order proof tactics in higher-order logic theorem provers. pages 56–68.

[19] G. Hutton. *Programming in Haskell*. Programming in Haskell. Cambridge University Press, 2016.

[20] Moa Johansson. Theory exploration for interactive theorem proving. In *4th International Workshop on Artificial Intelligence for Formal Methods (AI4FM 2013). Ed. by Grov, G., Maclean, E. and Freitas, L.(cit. on p. 5)*, 2013.

[21] Moa Johansson, Lucas Dixon, and Alan Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 47(3):251–289, 2011.

[22] Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. Hipster: Integrating theory exploration in a proof assistant. In *Intelligent Computer Mathematics*, pages 108–122. Springer, 2014.

[23] Donald E Knuth and Peter B Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983.

[24] R.L. McCasland, A. Bundy, and P.F. Smith. Mathsaid: Automated mathematical theory exploration. *Applied Intelligence*, 47(3):585–606, 2017.

[25] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

[26] Omar Montano-Rivas, Roy McCasland, Lucas Dixon, and Alan Bundy. Scheme-based theorem discovery and concept invention. *Expert Systems with Applications*, 39(2):1637–1646, 2012.

[27] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[28] Chris Okasaki. Purely functional random-access lists. In *FPCA*, volume 95, page 8695. Citeseer, 1995.

[29] Dan Rosén. *Proving Equational Haskell Properties using Automated Theorem Provers*. PhD thesis, MSc. Thesis, University of Gothenburg, 2012.

[30] Dan Rosén and Nicholas Smallbone. TIP: Tools for inductive provers. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 219–232. Springer, 2015.

[31] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 233–242. ACM, 1993.

[32] Nicholas Smallbone, Moa Johansson, Koen Claessen, and Maximilian Algehed. Quick specifications for the busy programmer. *Journal of Functional Programming*, 27, 2017.

[33] Niki Vazou, Joachim Breitner, Rose Kunkel, David Van Horn, and Graham Hutton. Theorem proving for all: Equational reasoning in liquid haskell (functional pearl). *SIGPLAN Not.*, 53(7):132–144, September 2018.

A

Example run for REVERSE

This appendix shows an example of Hopster in action. It is similar to the scenario as that described in Section 4.1.1, i.e. that reversing a list twice results in the original list [1].

The HOL theory file would contain all datatypes and functions related to lists. Here, only one datatype and a pair of functions are shown:

```
open HolKernel Parse boolLib bossLib;

val _ = new_theory "list";

val _ = Datatype `
list = Nil | Cons 'a list`;

val APPEND = Define `
append Nil ys = ys /\
append (Cons x xs) ys = Cons x (append xs ys)`;

val REVERSE_DEF = Define `
reverse Nil = Nil /\
reverse (Cons x xs) = append (reverse xs) (Cons x Nil)`;

val _ = export_theory ()
```

In the command below, the `Hopster.explore` function is called with two lists as arguments: a list of datatypes and another of the functions to explore. The datatype is the list of elements of any type (`: 'a list`). The functions are the reverse (`REVERSE_DEF`) and append (`APPEND`) operations.

```
- Hopster.explore [``:'a list``] [("list","REVERSE_DEF"),
                                  ("list","APPEND")];
== Functions ==
Nil :: (Arbitrary a, Enumerable a, Ord a) => List a
Cons :: (Arbitrary a, Enumerable a, Ord a) =>
        a -> List a -> List a

== Functions ==
append :: (Arbitrary a, Enumerable a, Ord a) =>
```

[1] The output has been slightly reformatted to fit on the page

```
           List a -> List a -> List a
reverse :: (Arbitrary a, Enumerable a, Ord a) =>
           List a -> List a

== Laws ==
1. reverse Nil = Nil
2. append x Nil = x
3. append Nil x = x
4. reverse (reverse x) = x
5. reverse (Cons x Nil) = Cons x Nil
6. append (Cons x y) z = Cons x (append y z)
7. append (append x y) z = append x (append y z)
8. append (reverse x) (reverse y) = reverse (append y x)
9. reverse (Cons x (reverse y)) = append y (Cons x Nil)
10. reverse (append x (reverse y)) = append y (reverse x)
11. reverse (append (reverse x) y) = append (reverse y) x
12. append (reverse (Cons x y)) z = append (reverse y) (Cons x z)
13. append (reverse x) (Cons y Nil) = reverse (Cons y x)
14. reverse (Cons x (Cons y Nil)) = Cons y (Cons x Nil)
15. reverse (append x (Cons y Nil)) = Cons y (reverse x)
16. reverse (append x (Cons y z)) =
    append (reverse z) (Cons y (reverse x))
17. append (reverse (append x y)) z =
    append (reverse y) (append (reverse x) z)
18. append x (Cons y (Cons z Nil)) =
    reverse (Cons z (Cons y (reverse x)))
19. append x (reverse (Cons y z)) =
    reverse (Cons y (append z (reverse x)))
20. append x (Cons y (reverse z)) =
    reverse (append z (Cons y (reverse x)))
21. reverse (append x (append y (reverse z))) =
    append z (reverse (append x y))
22. append (append (reverse x) y) (reverse z) =
    reverse (append z (append (reverse y) x))

>
```

The code generator produces the following Haskell file:

```
module Explore where

import Prelude ()

data List a = NIL | CONS a (List a)

reverse [] = []
reverse (h : t) = append (reverse t) [h]
```

```
append [] l = l
append (h : l1) l2 = h : append l1 l2
```

Finally, the result of theory exploration (the 22 conjectures listed above) are proved and converted into HOL theorems.

```
** Conjectures found **
** Lemmas found **
⊢ ∀y z. append (reverse z) (Cons y Nil) = reverse (Cons y z)
⊢ ∀y z x2. append (Cons y z) x2 = Cons y (append z x2)
⊢ ∀y. reverse (Cons y Nil) = Cons y Nil
⊢ ∀y. append Nil y = y
⊢ reverse Nil = Nil
⊢ ∀y. reverse (reverse y) = y
⊢ ∀y z x2.
append (append (reverse x2) z) (reverse y) =
reverse (append y (append (reverse z) x2))
⊢ ∀y z x2.
reverse (append z (append x2 (reverse y))) =
append y (reverse (append z x2))
⊢ ∀y z x2.
append x2 (Cons z (reverse y)) =
reverse (append y (Cons z (reverse x2)))
⊢ ∀y z x2.
append x2 (reverse (Cons y z)) =
reverse (Cons y (append z (reverse x2)))
⊢ ∀y z x2.
append x2 (Cons z (Cons y Nil)) =
reverse (Cons y (Cons z (reverse x2)))
⊢ ∀y z x2.
append (reverse (append z y)) x2 =
append (reverse y) (append (reverse z) x2)
⊢ ∀y z x2.
reverse (append x2 (Cons z y)) =
append (reverse y) (Cons z (reverse x2))
⊢ ∀y z. reverse (append z (Cons y Nil)) = Cons y (reverse z)
⊢ ∀y z. reverse (Cons z (Cons y Nil)) = Cons y (Cons z Nil)
⊢ ∀y z x2.
append (reverse (Cons z y)) x2 =
append (reverse y) (Cons z x2)
⊢ ∀y z. reverse (append (reverse z) y) = append (reverse y) z
⊢ ∀y z. reverse (append z (reverse y)) = append y (reverse z)
⊢ ∀y z. reverse (Cons z (reverse y)) = append y (Cons z Nil)
⊢ ∀y z. append (reverse z) (reverse y) = reverse (append y z)
⊢ ∀y z x2. append (append y z) x2 = append y (append z x2)
⊢ ∀y. append y Nil = y
```

B

Test cases and their goals

Table B.1: Goals solved for each test case

Name	Goal
REVERSE	∀xs. reverse (reverse xs) = xs
REV	∀xs. rev (rev xs []) [] = xs
LENGTH_DROP	∀xs ys. length (drop (length ys) (xs ++ ys)) = length xs
MIRROR	∀t. mirror (mirror t) = t
INORDER_MIRROR	∀t. inorder (mirror t) = reverse (inorder t)
ARRAY	∀v t n. lookup (update v t n) n = Just v
SUB_BIGGER	∀m n. sub n (add n m) = Zero
SUB_TWICE	∀m n p. sub (sub m n) p = sub m (add n p)
COMPILER	∀p env. exec env (comp p) [] = Just (eval env p)