CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Proof output and machine learning for inductive theorem provers

Master's thesis in Computer Science

VICTOR LINDHÉ
NIKLAS LOGREN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

MASTER'S THESIS IN COMPUTER SCIENCE

Proof output and machine learning for inductive theorem provers

VICTOR LINDHÉ
NIKLAS LOGREN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Proof output and machine learning for inductive theorem provers

VICTOR LINDHÉ
NIKLAS LOGREN

Examiner:
Patrik Jansson, Department of Computer Science and Engineering
Supervisor:
Moa Johansson, Department of Computer Science and Engineering

Master's Thesis 2016:06
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2016

Proof output and machine learning for inductive theorem provers

VICTOR LINDHÉ
NIKLAS LOGREN
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Abstract

Automatic theorem provers have lately seen significant performance improvements by utilising knowledge from previously proven theorems using machine learning. HipSpec is an inductive theorem prover that has not yet explored this area, which is the primary motivation for this work.

We lay a foundation for supporting machine learning implementations within HipSpec. Firstly, a format for representing inductive proofs of theorems is designed. Secondly, a *persistent library* is implemented, which allows HipSpec to remember already-proven theorems in between executions. These extensions are vital for allowing machine learning, since they provide the machine learning algorithms with the necessary data.

This foundation is used to perform machine learning experiments on theorems from the TIP library, which is a collection of benchmarks for inductive theorem provers. We define several different feature extraction schemes for theorems, and test these using both supervised learning and unsupervised learning algorithms.

The results show that although no correlation between induction variables and term structure can be found, it is possible to utilise clustering algorithms in order to identify some theorems about tail-recursive functions.

Keywords: automated theorem proving, automated reasoning, theory exploration, machine learning

Acknowledgements

We want to thank our supervisor Moa Johansson for her dedication and unfailing support, and to Dan Rosén for his technical expertise and friendly manners.

Victor Lindhé and Niklas Logren, Gothenburg, June 2016

Contents

1 Introduction

When developing software in Haskell, it is common practice to verify the correctness of a program by writing properties about it that it should satisfy, and then automatically test these properties using QuickCheck [4]. An example of such a property is that reversing a list twice should yield the original list:

```
rev(rev(xs)) = xs
```

Having tested this property for many different lists *xs* gives the programmer some assurance that their definition of *rev* behaves as expected.

However, even if all tests pass, it is still possible that too few cases were tested and that the property is false. In contrast to this approach, proving a property offers much stronger guarantees: the programmer can be entirely convinced that the property holds for any case. This is where HipSpec enters the picture.

HipSpec is a system for automatically discovering and proving properties of Haskell programs, using induction [6]. It can automatically conjecture interesting properties of a program, as well as prove them by utilising an external theorem prover.

One aspect which differentiates HipSpec from other provers is its support for induction, which is often useful for proving properties about recursive data types and functions, commonly used in functional programming languages. Another aspect is that it uses a technique called *theory exploration* to invent auxillary lemmas which might be needed in the proof (explained in more detail in Chapter 2.3).

1.1 Goals

The purpose of this project is to extend HipSpec in four ways:

1. **Creating a proof output format:** Currently, HipSpec only returns whether or not a conjecture was proven without any additional data. In order to utilise machine learning on previously completed proofs, we need a way to store proofs. Therefore, we design a suitable format for representing proofs in HipSpec, and modify HipSpec to output proofs in this format.

2. **Developing a persistent library:** Currently, HipSpec does not store any knowledge permanently when it manages to prove something, and therefore, it cannot remember anything in between executions. This is a problem, since the machine learning engine needs to train itself on everything that HipSpec has ever proven. Therefore, we introduce a persistent library, which stores all functions, data types and lemmas the current instance of HipSpec ever encountered, as well as their proofs.

3. **Feature extraction on lemmas:** By this point, a library has been developed and filled with proofs of lemmas. Now, we need a way to extract features from lemmas, which are used for quantifying how similar two lemmas are. Thus, one challenge is to investigate what features of lemmas/theorems are useful for machine learning when attempting to learn how to recognise similar proofs. Different feature extraction schemes will be evaluated in combination with different machine learning engines, on how well they work for both supervised and unsupervised learning.

4. **Evaluate the possibilities of machine learning for HipSpec:** The final goal is to explore what kinds of possible improvements can be done to HipSpec using machine learning. This involves attempting to solve what we call the *induction selection problem*: selecting which induction scheme to use and which variable(s) to do induction on using supervised learning. In addition, we will also cluster similar lemmas and functions using unsupervised learning, in order to see if any common patterns can be found.

1.2 Motivation

When humans prove theorems we always rely on previous knowledge, and we often draw connections between different proofs that share a similar structure (reasoning by analogy). The main purpose of this project is to investigate whether this behaviour can be replicated by an automated theorem prover.

This requires defining a format for proofs. The proof output could additionally increase the user's confidence in the tool since it would be easier to check the proofs manually, and it could also be the starting point for implementing a proof checker.

That HipSpec keeps track of all things it has proved is essential for learning from previous proofs; furthermore, it will allow HipSpec to become smarter as it encounters more problems. The persistent library serves as a foundation for the machine learning implementation that will be examined in this project, as well as for any further work concerning machine learning in HipSpec.

Machine learning has seen many successful applications lately, and also in the field of theorem proving (see section 2.2.3). We want to bring some of that power also to HipSpec.

1.3 Contributions

This project has made the following concrete contributions:

- Defined and implemented a proof format for inductive proofs based on the TIP language
- Implemented a persistent library for HipSpec that stores lemmas, function definitions and data type definitions
- Implemented a machine learning and feature extraction framework for HipSpec

All changes to HipSpec, as well as a document detailing how the extensions are to be used can be found at `https://github.com/aommm/emna`, and extensions to the TIP library can be found at `https://github.com/aommm/tip-tools`.

1.4 Prerequisites

The reader is assumed to have basic knowledge of probability, as well as knowledge of logic and of formal proofs. It is also assumed that the reader is familiar with Haskell and functional programming. Furthermore, a background in Computer Science and programming languages is helpful for understanding the internal language of HipSpec and the new proof format.

2 Background and literature review

This project explores the areas of machine learning and theorem proving in general, and the inductive theorem prover HipSpec in particular. Therefore, this chapter aims to give an introduction to these subjects.

2.1 Machine learning

Machine learning is a field of study which examines the possibility of computers to learn instead of being explicitly programmed. This is most often achieved by means of software algorithms that analyse data and make decisions or predictions based on the data, rather than based on predefined instructions.

Now, we will briefly introduce the two main families of machine learning algorithms, and describe feature extraction, an important transformation most data has to go through in order to be usable in machine learning. We will also describe the specific algorithms *Bernoulli Naive Bayes* and *K-means clustering* that are used in this project, as well as a method called *K-fold cross-validation*, which is used for evaluating the performance of machine learning algorithms.

2.1.1 Supervised learning

Supervised learning algorithms are presented with *data* (input) and a set of *correct answers* (desired output), and try to learn how the input and output is correlated. Then, they can be provided with only input, and hopefully give the correct output.

More formally, the task of supervised learning is to find a function $g : X \to Y$, where X is the input space and Y the output space, given N training samples $\{(x_1, y_1), ..., (x_N, y_N)\}$. There are many algorithms for finding such a function, and in this project Bernoulli Naive Bayes will be used. For more details about how this is achieved, see section 2.1.4.

An example application of this is predicting housing prices. With supervised learning, we would give the computer information about some already sold houses (input), e.g. size in square metres, number of floors and city. We would also specify the prices that the houses were sold for (desired output). This would then be fed into some training algorithm (e.g. Bernoulli Naive Bayes) that outputs a function g, into which we could then input a new house and get a prediction of what the house would sell for.

2.1.2 Unsupervised learning

Unsupervised learning algorithms get only input and no desired output. This means that the algorithm should find similarities and patterns in the data without any additional guidance. Unsupervised learning is often useful when working with a large amount of data which is infeasible to label manually.

A specific kind of unsupervised learning is *clustering*, which partitions a data set into several groups. What a group (or cluster) is, is not formally defined, but depends on the particular clustering algorithm used. In general, though, the data points of a particular cluster are assumed to be more similar (in some way) to each other than to other data points. In this project, the K-means clustering algorithm is used, which is described in more detail in section 2.1.5.

An example: assume that we have a large amount of articles, and want to cluster them based on their topic (e.g. ”sports” or ”science”). Then, we simply feed the articles into an unsupervised learning algorithm which will attempt to cluster them by topic in a meaningful way. How the clusters are created depends heavily on the *feature extraction* used: how the important parts of the articles were circled out.

2.1.3 Feature extraction

Often, data is not suited for direct usage by machine learning algorithms. This can be the case if the data is in a nonsuitable format (i.e. a text string), or if it contains too much information, which could lead to timeouts or overfitting. This is why it is common to extract *features* from the original data, and train on these instead.

A feature is something thought to characterize data in a non-redundant way. For example, some features of a person are their gender, age and height. Data is then measured solely by the extracted features. This ensures that the machine learning algorithm is looking only at the relevant parts of the data.

Feature extraction is an important part of machine learning, and in real-world applications it is crucial that it is done right. It is by the feature extraction scheme that one defines what it means for two data points to be similar, and it is easy to imagine that different feature extraction schemes can yield vastly different results.

In the first example of predicting house prices, we would probably see drastically increased performance by factoring in the location of the house (for instance as ″distance from city centre″). In the second example of clustering articles, we might want to count word occurences for each article and use those as features (for instance ″football″:5 and ″derivative″:4). Additionally, in this case, neglecting to count common words such as ″and″ and ″then″ would probably improve the accuracy.

2.1.4 Bernoulli Naive Bayes

In order to solve the *induction selection problem* (choosing which variable to do induction on, defined in section 1.1), supervised learning is used with lemmas as the input and the induction variable(s) used as the desired output. This is described in more detail in section 3.4. More specifically, the input is obtained by running all lemmas through feature extraction, yielding one feature vector $x_1, \dots, x_n$ per lemma.

The algorithm that is used for supervised learning in this project is Bernoulli Naive Bayes. All Naive Bayes algorithms are based on a probabilistic model, and they assume that each pair of features are (probabilistically) independent from each other. Bernoulli Naive Bayes is a specific such algorithm, in which all features are binary, i.e. $x_i \in \{0, 1\}$. Now, we will go into detail about how this algorithm works.

How it works In abstract terms, say that we want to see which class the sample $x_1, \dots, x_n$ belongs to. What we want to do is to calculate, for each class y, the probability of the sample $x_1, \dots, x_n$ belonging to that class:

$$P(y|x_1, \dots, x_n)$$

This can be calculated from the training data e.g. using probability tables, and then we could classify our sample by choosing the class with the highest probability.

The problem with this approach is that it is often infeasible to calculate these class probabilities, when the number of features or samples are large.

In order to make this practical, Bayes' Rule from probability theory is used, which states that

$$P(y \mid x_1, \dots, x_n) = \frac{P(y)P(x_1, \dots x_n \mid y)}{P(x_1, \dots, x_n)}$$

$P(y)$ can be calculated from the training data, and the other expressions can be simplified further. Since the algorithm assumes that each pair of features are independent from one another, and since the nominator in the above expression is constant given the feature vector, the entire expression can be simplified to

$$P(y \mid x_1, \dots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

Where $\propto$ is read as "is proportional to". Now, we have constructed the Naive Bayes probability model. Note that this expression does not give an exact value of the probability for a class, but it can still be used in order to perform classification. The only thing that remains is how to calculate $P(x_i \mid y)$, i.e. the probability of a specific feature occuring in a class. When using Bernoulli Naive Bayes, recall that $x_i \in \{0, 1\}$. The probability is given by:

$$P(x_i \mid y) = P(i \mid y)x_i + (1 - P(i \mid y))(1 - x_i)$$

$P(i \mid y)$ denotes the probability of feature i being in class y, and it can easily be precomputed from the training data.

Finally, in order to actually choose a class for the feature vector, we could for example choose the class with the highest probability $\hat{y}$ (using the so-called *MAP* estimator):

$$\hat{y} = \arg\max_{y} P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

Motivation The Naive Bayes family of algorithms has been shown to work well for many problems, despite the apparently oversimplified assumption of independence [26]. It has been used successfully for e.g. text classification and spam filtering, as well as in theorem proving contexts. Furthermore, it scales well for a large number of samples, since many terms in the above equations can be precomputed.

Example Consider an example where a survey on happiness has been conducted. Each participant answered either 1 (yes) or 0 (no) to a number of questions. A feature vector then looks like

$$x = (rich, married, healthy)$$

Additionally, each participant answers $t = 1$ if they are content with their life or $t = 0$ if they are not. Say that we got 8 responses

$$t = 1 : (1; 1; 1); (0; 0; 1); (1; 1; 0)(1; 0; 1)$$
$$t = 0 : (0; 0; 0); (1; 0; 0); (0; 0; 1); (0; 1; 0)$$

And that we want to predict the contentedness of a new person who is not rich, but married and healthy. In other words, we want to calculate the probability of classifying the vector $x_{new} = [0, 1, 1]$ with the label $t_{new} = 1$ (in this simple case we can calculate the probability directly, rather than using proportionality).

$$P(t_{new} = 1 \mid x_{new}) = \frac{P(t_{new} = 1)P(x_{new} \mid t_{new} = 1)}{P(x_{new})}$$

Since we have an equal number of content and non-content responders, we can say that $P(t_{new} = 1) = P(t_{new} = 0) = 0.5$.

The other term in the numerator, the so-called *likelihood*, is more involved. If we use the Naive Bayes approach, we can assume that the vector components of each data point are independent, and hence we can multiply them to find the probability for the whole vector:

$$P(x_{new}|t_{new} = i) = \prod_{d=1}^{3} P(x_d^{new}|x_{new} = i)$$

where x_d^{new} is the d'th component of x_{new}

The question now is how each vector component x_d^{new} is distributed. We see that it can only obtain the value 0 or 1, and so we make the observation that it follows a *Bernoulli distribution*. To find the probabilities for $t_{new} = 1$ and x_i^{new}, we simply take the mean of all the data points' *i*th components. Hence, when $t_{new} = 1$:

$$\begin{aligned} x_1^{new} &\sim Bern(0.75) \\ x_2^{new} &\sim Bern(0.5) \\ x_3^{new} &\sim Bern(0.75) \end{aligned}$$

And when $t_{new} = 0$:

$$x_1^{new} \sim x_2^{new} \sim x_3^{new} \sim Bern(0.25)$$

The likelihood can then be computed using the probability mass function of the Bernoulli distribution:

$$\begin{aligned} P([0,1,1] \mid t_{new} = 1) &= 0.25 * 0.5 * 0.75 = 0.0938 \\ P([0,1,1] \mid t_{new} = 0) &= 0.75 * 0.25 * 0.25 = 0.0469 \end{aligned}$$

The nominator of the original expression can be rewritten using the law of total probability into

$$P(x_{new}|t_{new} = 0)P(t_{new} = 0) + P(x_{new}|t_{new} = 1)P(t_{new} = 1)$$

Hence, the final probability of the new person being content becomes

$$\begin{aligned} P(t_{new} = 1 \mid x_{new}) &= \frac{0.0938 * 0.5}{0.0469 * 0.5 + 0.0938 * 0.5} \\ &= 0.6667 \\ &\approx 67\% \end{aligned}$$

2.1.5 K-means clustering

For clustering, on the other hand, the K-means algorithm is used.

How it works This algorithm works by first creating K clusters. Each cluster is described by the mean μ_j of all the samples assigned to the cluster, also called the *centroid* of the cluster.

Typically, the centroids are initialised by picking K samples from the data set. Then, each sample is assigned to belong to the cluster with the closest centroid (where "closest" is taken to mean euclidian distance between the centroid and the sample). The centroids μ_j are then recomputed to be the mean of all samples assigned to the cluster.

This process, computing the centroids and then assigning samples to centroids, is repeated until the difference between the old centroids and the new centroids are less than a threshold. In other words, it is repeated until the centroids have converged. Now the algorithm is finished, and all samples have been assigned to a cluster.

Motivation The K-means clustering algorithm is a popular and widely used algorithm that is fast and easy to use [23, 24]. This algorithm was picked for its simplicity, because we wanted to focus on optimising the feature extraction rather than the algorithm.

2.1.6 K-fold cross-validation

K-fold cross-validation is a standard technique in machine learning for evaluating the performance of supervised learning algorithms. It will be used in this project for evaluating which machine learning parameters perform the best when solving the induction selection problem.

Its input is a set of labelled data. In our case, this is a set of lemmas where each lemma has the induction variable(s) it was proven with as a label.

It works by first partitioning the data into K folds. Then, 1 fold is assigned to be the *validation fold* and $K - 1$ folds are assigned to be *training folds*. A fresh instance of the supervised learning algorithm is then fitted on the data from the training folds. After this, each sample in the validation fold is classified using the algorithm, and the number of correct classifications are counted, and eventually a success ratio is calculated for all lemmas in the validation fold.

This entire process is repeated in total K times, with a new validation fold each time. Finally, the average success ratio of all repetitions is calculated, which becomes the overall performance ratio of the tested algorithm.

2.2 Theorem proving

Automated reasoning is the field of study which aims to investigate how computers can perform different kinds of reasoning automatically or semi-automatically. While the field of automated reasoning has seen many improvements since its inception, there is room for improvement, since computers are still behind humans when it comes to scientific thinking [16].

A subfield within automated reasoning is theorem proving, which aims to prove mathematical theorems using computers. This requires the software to implement mathematical reasoning and inference rules. Theorem proving software can today assist in analysing and exploring mathematical theories, and have been used successfully in several mathematical proofs, for example the Four-colour theorem and Kepler's conjecture [2] [9].

This chapter aims to give the reader an understanding of the field of theorem proving, which is the context in which this project is done. Firstly, we will give an overview of the different types of theorem provers that exist today, and a few examples of popular provers. Secondly, we will discuss the challenge of developing a proof output format for theorem provers. Finally, we will discuss prior applications of machine learning in theorem proving, and go into detail about different feature extraction schemes used by different provers.

2.2.1 Types of theorem provers

Theorem provers can (roughly speaking) be called either interactive theorem provers or automatic theorem provers. Their main difference lies in the way they search for proofs – in interactive provers, the search is guided by the human user, while in automated proofs, all decisions are made by the prover. Additionally, theorem provers that support induction are said to be inductive theorem provers. Here, we will describe these categories more in detail, and give examples of provers.

This report covers the system HipSpec, which is an automated inductive theorem prover. However, since techniques similar to what we want to accomplish has been implemented already in some interactive theorem provers, we will give a brief background to how these work as well.

Automatic Theorem Provers Automatic theorem provers can prove theorems without user interaction. They generally require two main inputs: (1) the conjecture we want to prove, and (2) the background theory consisting of potential premises for the conjecture. Examples of first-order automatic theorem provers include Z3, Vampire, Waldmeister and E [8, 17, 12, 21].

Interactive Theorem Provers When working with interactive theorem provers, the user guides the search for a proof by e.g. choosing which proof procedure to use, or by providing auxiliary lemmas. This makes the interactive provers more capable than the automatic ones, and more specifically they are better at producing complex proofs [14].

It is worth noting that most interactive provers also include automated tactics that can prove some things automatically. Hence, many interactive provers could also be considered as "semi-automatic" provers.

Examples of interactive theorem provers include Isabelle/HOL, Mizar and HOL Light [25, 20, 10]. Additionally, these systems often use a higher-order logic [11], which allows for expressing more properties than first-order logic.

Inductive Theorem Provers Inductive theorem prover is the name for provers that can prove things using induction. Many different induction principles could be supported, for example recursion induction, structural induction or mutual induction. Examples of inductive theorem provers include HipSpec and Hipster [15].

2.2.2 Proof output

Formats for the *input* of various kinds of provers have largely been standardized. However, no de-facto standards exist for the output (or *proof certificates*) of inductive theorem provers [3]. Different provers output different kinds of proofs, often containing data internal to the prover which is of little interest to others. Also, often crucial details are skipped which are obvious to the prover but not necessarily to a third-party, or no output is produced at all.

Böhme and Weber [3] propose some guidelines for designing a proof output format:

- Human-readable
- Proper level of theoretical detail
- Simple semantics
- State derived formulas explicitly
- Provide documentation

In this project, we aim to follow these guidelines as well as possible. While Böhme and Weber's main goal is to make proof formats checkable by an external proof checker, we will not implement a checker in this project. Our focus will be on storing the features of proofs which will be useful for machine learning and on helping users understand the output from HipSpec.

2.2.3 Machine learning in theorem proving

When humans prove theorems, we rely heavily upon data from previously completed proofs when proving new theorems. Not every proof is unique, and it is not uncommon that proofs share many similarities with one another. Therefore, it is not unreasonable to assume that machine learning could be applied in the domain of theorem proving, to find similar lemmas and aid in the proof search.

In fact, machine learning has already been integrated in some theorem proving systems, and these implementations will now be examined. Furthermore, how these systems deal with feature extraction from theorems warrants a closer look.

Isabelle/HOL and Sledgehammer Machine learning is employed in an extension to Isabelle/HOL called Sledgehammer [19], which builds proofs by invoking external theorem provers.

An important task for Sledgehammer is to find already proven facts relevant to the conjecture it is proving, which should be sent to the theorem prover. This is called the *premise selection problem* [16]. If the prover is sent too many facts, the execution takes too long time. On the other hand, if it gets too few facts, it won't have the information necessary for completing the proof. This fact selection is done by the so-called *relevance filter*, which selects known facts based on similarity to the conjecture Sledgehammer is attempting to prove.

One of the relevance filters available for Sledgehammer is MaSh, which uses machine learning for selecting premises (facts) for the proof goal [18, 1].

Feature extraction in MaSh MaSh extracts features based on symbols, types, type classes and the theory the fact belongs to, along with the facts used in the proof.

Additionally, features of the terms are extracted with a certain depth, in order to preserve parts of the formula's structure (such as subterms). For example, the lemma *rev(rev(xs)) = xs*, where *rev* is a function and *xs* is a variable of type list, would give the features *rev* and *rev(rev)* given a depth of 2. In addition to this, the theory *List* and the type *List.list* would also be features.

The way of extracting features in this project is very similar to that of MaSh. For more on how it is done in this project, see section 3.3.

2.3 Theory Exploration and Induction

This project aims to extend the HipSpec system, which is an automated theorem prover that works by invoking external theorem provers. Now, a brief introduction to HipSpec's main mechanisms will be given.

2.3.1 Theory Exploration

HipSpec uses a bottom-up approach when trying to prove user-stated conjectures, which are properties of a Haskell program that the user wants to prove. The first step of this approach is called *theory exploration*, which is more described in section 2.3.2 and 2.3.3. When using theory exploration, HipSpec builds a background theory with as many lemmas as possible before trying to prove the user conjectures. This has been shown to be efficient when it comes to finding the necessary lemmas for many inductive theorems [6].

This approach is the opposite of the conventional top-down approach where the prover first tries to prove the conjecture and, if it fails, tries to construct needed auxiliary lemmas and then makes a new attempt on the proof goal.

2.3.2 Using QuickSpec and how it works

When first started, HipSpec uses a program called QuickSpec to generate a set of conjectures, which are equational properties believed to be true by testing, but not yet proven [5]. This is what is called theory exploration, and now we will describe in more detail how it works.

As a first step, QuickSpec generates a universe of terms based on the functions and variables in a supplied program. As input, QuickSpec takes a Haskell program, a list of variables and functions (along with their types) and test generators for the involved types.

From this universe, it puts terms in different equivalence classes. Two terms are in the same equivalence class if they are the same for all test cases QuickSpec runs on them. In the beginning, all terms are in the same equivalence class. When discovering that two terms in the same class get different results from the same test case, the equivalence class is split. This continues until no split has happened for a couple of hundred tests – we then regard it as stable. Terms that are alone in their own equivalence classes are discarded.

The next step is to generate the equations from these equivalence classes. This is done by selecting one term t from a class, preferably as simple as possible, and stating that every other term t' equals that term, getting equations of the form $t = t'$. The selected term is called the *representative term*.

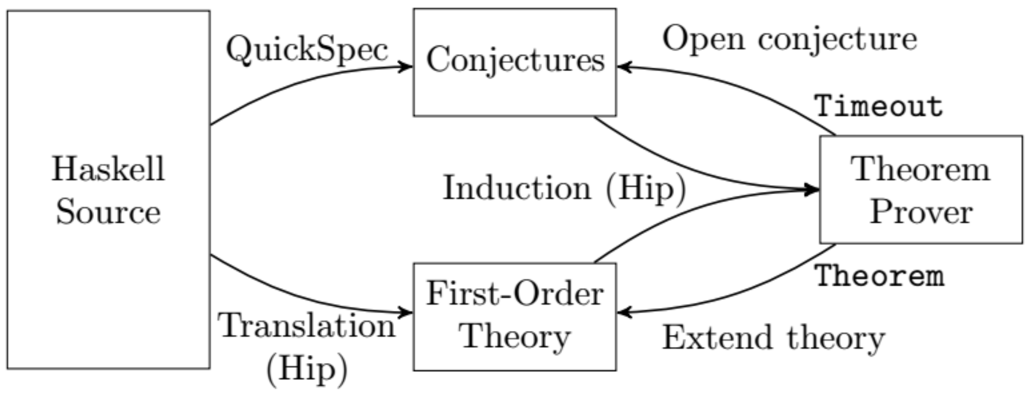
The final step of QuickSpec is called *depth optimisation*. Given two equal terms, one of them is selected to be the standard form of the expression. And given these standard forms, only terms with standard form subterms are generated, avoiding unecessary terms that are obvious from other terms.

2.3.3 The HipSpec loop

When QuickSpec is finished, HipSpec has a conjecture set which it will loop over, trying to prove each conjecture. Now we will describe this loop in more detail.

Selecting a conjecture to prove HipSpec begins the loop by selecting a conjecture **c**. How this conjecture is chosen is important for HipSpec's performance: some conjectures might require other conjectures to be proven

Figure 1: The main loop of HipSpec [6]



first, and thus should be picked late, while others can be proven immediately, and thus should be picked early.

HipSpec chooses conjectures by saying that simple equations have a priority over complicated ones. The simplicity of a term is defined by its size: the smaller, the simpler. If two terms have the same size, the more general term is picked: we pick the one with the greatest number of distinct variables. This is a reasonable heuristic, but one could imagine using machine learning in order to decide conjecture order (see section 6.1, Further work).

Proving the conjecture When HipSpec has selected a conjecture, it checks whether the conjecture follows from first order reasoning. If so, it is regarded as trivial, and HipSpec discards it.

Otherwise, it tries to prove the conjecture using induction using the subsystem Hip. Hip first enumerates all the different ways of doing structural induction on a conjecture. For each way of doing induction, Hip then invokes an external theorem prover for all subproblems of the induction (e.g. for the base case and the step case).

If the external prover succeeds for all subproblems, then Hip has successfully proven the conjecture. It is then added to the background theory and can be used in subsequent proof attempts. However, if the proof attempts fails (usually because of a timeout) the conjecture is added to the set of failed conjectures.

Reusing failed conjectures When the set of open conjectures is empty, HipSpec refills it with all the failed conjectures and repeats the loop if any new conjecture was proven in the last iteration. The purpose of this is to try to prove these conjectures with a larger background theory than it might have had available in the earlier attempts.

When all discovered conjectures are proven, or one iteration has been done without any new success, HipSpec is done with the theory exploration part. After this, HipSpec attempts to prove the user specified conjectures, using the background theory.

2.3.4 TIP: Ton of Inductive Problems

TIP (Tons of Inductive Problems) is a benchmark suite for inductive theorem provers [7], whose purpose is to provide a common test suite for many different provers. It consists of hundreds of problems, which are expressed in the TIP language (which is an extension of the SMT-LIB language).

3 Implementation

Here follows a description of the project's implementation. It covers design/implementation of a proof format, storing the proofs into a persistent library, extracting features from the library and finally using machine learning on the features in order to find the most probable induction order.

3.1 Proof output

The main design goal of the proof format is to keep it at a high abstraction level without too many details. In this way, it is more like a *proof plan* than a complete proof.

By leaving out low-level details, the representation becomes significantly simpler. Directly translating low-level proofs between different provers is very difficult, since they often have differences in their internal logic and in the inference rules they implement. Instead, the chosen high-level approach allows different provers to reconstruct the proof from the proof plan if required, following the same approach as Sledgehammer and Metis [19, 13].

Since HipSpec proves conjectures by induction, it is necessary to store the induction variable(s). Furthermore, since HipSpec works by invoking external theorem provers, it is necessary to store how the prover was invoked (the input arguments). It was decided to not store the output of the external provers, as the input is sufficient for running the prover again and reproducing the results.

Thus, our format includes these pieces of information:

- Induction variable(s) (stored as De Bruijn indices[1])
- Induction method (structural, recursion, ...)
- Lemmas used by prover
- External prover identifier (e.g. z3-4.4.0)
- Internal prover identifier (e.g. hipspec-0.1)

[1]The first occurring variable is renamed to 0, the second to 1 and so forth. For example, $\lambda x\ y\ .\ y$ would be translated to $\lambda 0\ 1\ .\ 1$

```
Proof.          Proof       ::= "(" LemmasUsed IndVars
    IndMethod Prover Prover ")";
IndVars.        IndVars     ::= "[" [Integer] "]";
IndStructural.  IndMethod   ::= "structural";
LemmasUsed.     LemmasUsed  ::= "[" [LemmaName] "]";
Prover.         Prover      ::= Symbol;
```

Figure 2: The BNFC grammar for a proof

```
([lemma-155, lemma-49, lemma-60] [0] structural z3-4.4.0
emna-0.1)
```

Figure 3: An example proof

The actual BNFC grammar can be seen in Figure 2. An example proof as rendered in this grammar can be seen in Figure 3.

3.2 Persistent library

HipSpec was extended to output proofs to a persistent library that keeps track of everything HipSpec has ever proven. This kind of functionality exists in many other theorem provers, and is essential for gathering data for use in machine learning.

Whenever HipSpec is invoked with the flag *–output=/path/to/library*, it loads the library from that path. If the proof attempt succeeds, the library is extended with new proofs: both of the original conjecture, as well as of any lemmas that were proven along the way.

The library keeps track of functions, data types and lemmas. Whenever a new lemma is added, any functions and data types ocurring in that lemma are also added. If two functions/data types share the same name, they are assumed to be equal, and the newest one will be discarded. In this implementation, it is the user's responsibility to make sure function names are unique. The reason for this is that checking function equality is difficult, and it is left as further work.

Ensuring that lemmas are not duplicated in the library required a little more work. Firstly, lemmas were extended to optionally have a name.

Secondly, HipSpec was extended to check whether a lemma already exists before adding it. This involves both looking at the name of the lemma, and comparing the lemma's body with all other lemmas in the library.

In order to be able to test equality of lemmas, a trick is needed. When a lemma is added to the library, we begin by assigning new names to all (local) identifiers in the lemma's body. (For example, consider $L_1 = \lambda q\ w.q$ and $L_2 = \lambda a\ b.a$, which are identical except for their variable names - since they are both renamed using the same scheme, they would both be renamed to $\lambda x\ y.x$). Then, the lemmas' bodies are compared. Since all lemmas in the library are reduced according to the same scheme, it is guaranteed that alpha equivalent terms will be identified.

3.3 Feature extraction

Several different schemes were implemented for extracting features from lemmas. The feature extraction module loads a persistent library from a file, and then runs one or several feature extraction schemes on all lemmas contained therein.

Both so-called symbolic and abstract features can be collected, as well as some metadata regarding the structure of a lemma or its functions. All features are binary (either present or non-present), and are represented as strings.

3.3.1 Overview

There are four basic feature extraction schemes. There are also four analytical schemes, one for each basic scheme. Schemes can be used in any combination, as long as at least one scheme is used. The basic schemes are:

- **ls**: Symbolic features from lemmas. Used by analytical scheme **als**.
- **la**: Abstract features from lemmas. Used by analytical scheme **ala**.
- **fs**: Symbolic features from functions. Used by analytical scheme **afs**.
- **fa**: Abstract features from functions. Used by analytical scheme **afa**.

Given an extraction depth, the four basic schemes extract features from the body of a lemma. The analytical schemes use the features from their corresponding basic scheme to build new features, with metadata about the lemma.

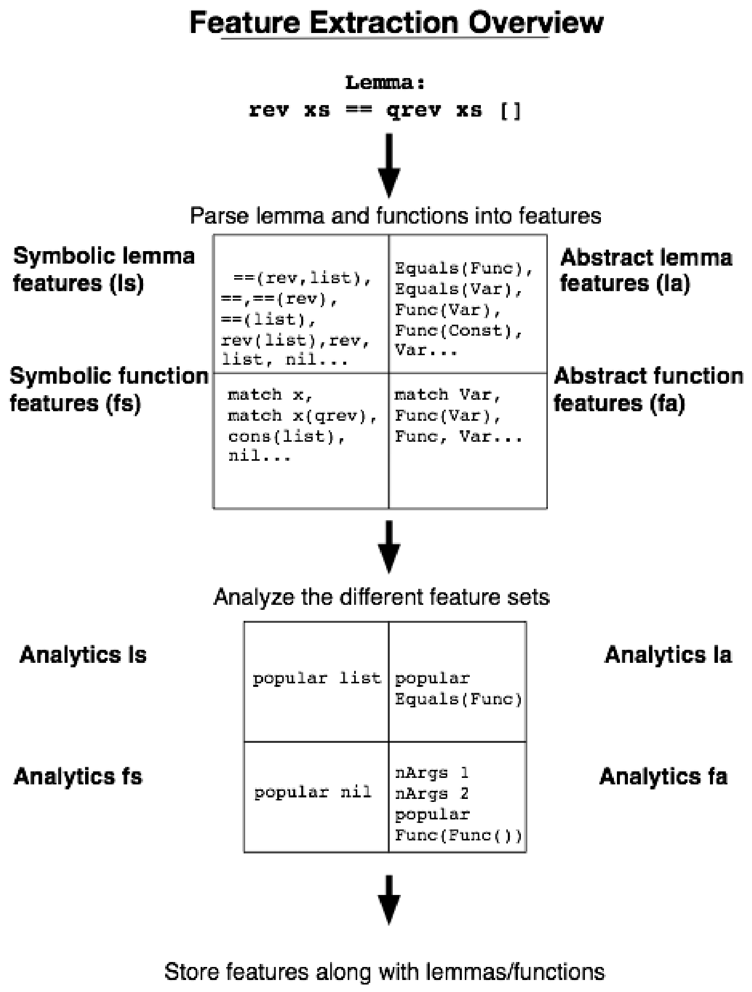

Feature Extraction Overview
Lemma:
rev xs == qrev xs []
Parse lemma and functions into features
Symbolic lemma features (ls)
==(rev,list), ==,==(rev), ==(list), rev(list),rev, list, nil...
Abstract lemma features (la)
Equals(Func), Equals(Var), Func(Var), Func(Const), Var...
Symbolic function features (fs)
match x, match x(qrev), cons(list), nil...
Abstract function features (fa)
match Var, Func(Var), Func, Var...
Analyze the different feature sets
Analytics ls
popular list
Analytics la
popular Equals(Func)
Analytics fs
popular nil
Analytics fa
nArgs 1 nArgs 2 popular Func(Func())
Store features along with lemmas/functions

Figure 4: Overview of how the basic schemes and the analytics schemes interact with each other.

Lemma and function features *Lemma features* are extracted from the bodies of lemmas, while *function features* are extracted from function definitions.

Function features are necessary for seeing that lemmas are talking about similar functions. Since induction often is made on the arguments of functions, how the functions are defined might affect the proofs. Therefore, the assumption here is that similar function features might implicate similar proofs.

Symbolic and abstract features *Symbolic features* are the symbols that can be directly read from a lemma or a function. For example, the lemma *rev xs == qrev xs []* contain symbols such as *rev*, *qrev*, == and *list* (which is the type of the variable *xs*). If for example depth 2 is used, symbolic features such as *rev(list)* or *==(rev,qrev)* would occur. Our hypothesis is that symbolic features are useful to find lemmas with the same components. See 3.3.2 for more detail.

Abstract features are built up the same way as symbolic features, but use more abstract notations. The purpose is to communicate term structure rather than the specific components of a lemma or function. For example, given the lemma example from earlier, *rev* would instead be denoted as *Func* and *list* would be *Var*. Depth is also used here, so instead of *rev(list)*, the feature would be *Func(Var)*.

The structural information given by abstract features could potentially be useful in finding lemmas similar in structure, but using different functions and types, for example different lemmas on tail recursion. See 3.3.3 for more detail.

Analytical schemes Every basic scheme has an analytical scheme which can be used to generate metadata from the set of features of that basic scheme. This metadata may contains information about the most popular feature, and some other features depending on chosen scheme. See 3.3.4 for more detail.

The working hypothesis here is that different metadata features could be useful in finding lemmas and functions that differ a lot, but have some important structural features in common.

Inverse document frequency *Term Frequency–Inverse Document Frequency* (TF-IDF) was used for assigning weights. TF-IDF is a technique for assigning feature weights on a feature-by-feature basis, based on how often the feature occurs in all samples. This means that popular features, such as *Var* or *Func* which occur in virtually every lemma, are assigned less weight.

3.3.2 Symbolic features

The symbolic features of a lemma concern the concrete symbols occuring in it. For example, the lemma *rev xs == qrev xs []* contains the concrete symbols *rev*, *qrev*, *xs*, *[]* and ==.

To begin with, a lemma is converted into a big tree of all its terms. For example, the lemma *rev xs == qrev xs []*, where *xs* and *[]* are of type *list*, would generate the tree seen in figure 5.

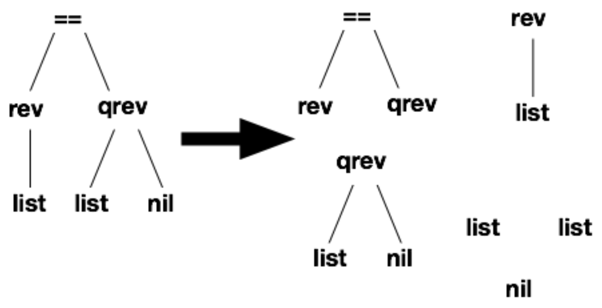
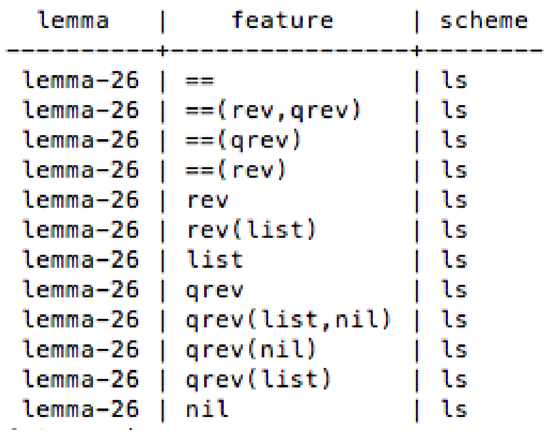


Figure 5: A term tree for *rev xs == qrev xs []*, and how it is converted into 6 smaller subtrees of depth 1-2. Note how two trees are identical – these duplicates will later be removed.

As we can see, the variable *xs* is replaced by its type *list*. This tree is then split up into all possible subtrees with a maximum depth, in this example of depth 2. Note that this is similar to how feature extraction is done in Sledgehammer [1].

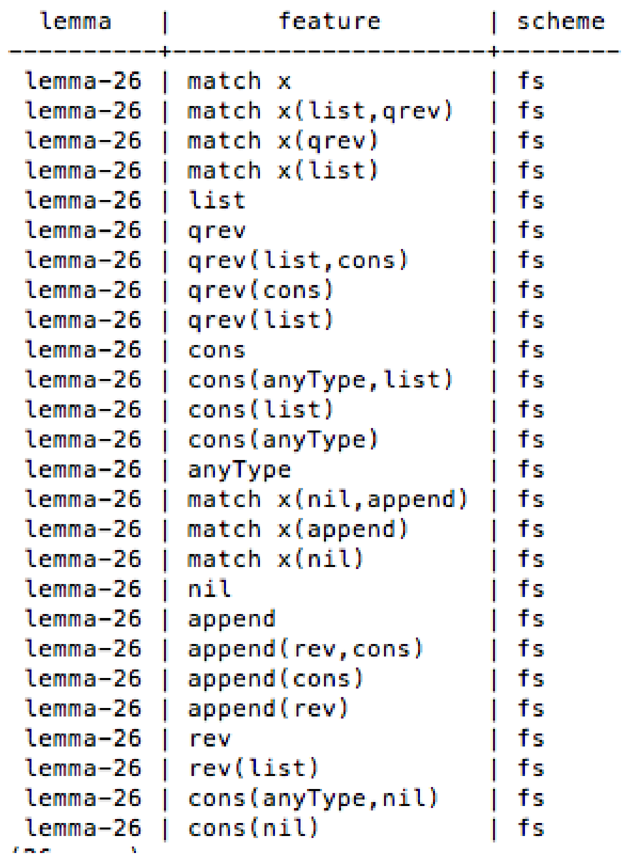
When having a set of trees, we extract features from each one of them. Each feature is represented as a string with the either the type or the function name. A function application will not only be represented alone, but also with combinations of the arguments. For the subtrees above, we can see what features are generated in Figure 6.



```
  lemma    |     feature      | scheme
-----------+-----------------+--------
 lemma-26 | ==               | ls
 lemma-26 | ==(rev,qrev)     | ls
 lemma-26 | ==(qrev)         | ls
 lemma-26 | ==(rev)          | ls
 lemma-26 | rev              | ls
 lemma-26 | rev(list)        | ls
 lemma-26 | list             | ls
 lemma-26 | qrev             | ls
 lemma-26 | qrev(list,nil)   | ls
 lemma-26 | qrev(nil)        | ls
 lemma-26 | qrev(list)       | ls
 lemma-26 | nil              | ls
```

Figure 6: The resulting symbolic lemma features. As we can see, no duplicates remain here.

Symbolic function features are generated the same way as for lemmas. There are, however, some types of features not occuring in lemmas which occur in functions. One example of this is pattern matching on arguments, specifying different function bodies. This is represented by the *match* feature. See Figure 7 for the function features from *rev xs == qrev xs [].*



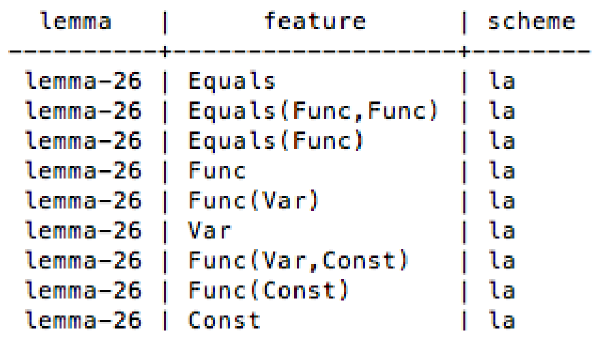
```
  lemma   |        feature        | scheme
----------+-----------------------+--------
 lemma-26 | match x               | fs
 lemma-26 | match x(list,qrev)    | fs
 lemma-26 | match x(qrev)         | fs
 lemma-26 | match x(list)         | fs
 lemma-26 | list                  | fs
 lemma-26 | qrev                  | fs
 lemma-26 | qrev(list,cons)       | fs
 lemma-26 | qrev(cons)            | fs
 lemma-26 | qrev(list)            | fs
 lemma-26 | cons                  | fs
 lemma-26 | cons(anyType,list)    | fs
 lemma-26 | cons(list)            | fs
 lemma-26 | cons(anyType)         | fs
 lemma-26 | anyType               | fs
 lemma-26 | match x(nil,append)   | fs
 lemma-26 | match x(append)       | fs
 lemma-26 | match x(nil)          | fs
 lemma-26 | nil                   | fs
 lemma-26 | append                | fs
 lemma-26 | append(rev,cons)      | fs
 lemma-26 | append(cons)          | fs
 lemma-26 | append(rev)           | fs
 lemma-26 | rev                   | fs
 lemma-26 | rev(list)             | fs
 lemma-26 | cons(anyType,nil)     | fs
 lemma-26 | cons(nil)             | fs
```

Figure 7: The resulting symbolic function features, from the functions *rev* and *qrev*.

3.3.3 Abstract features

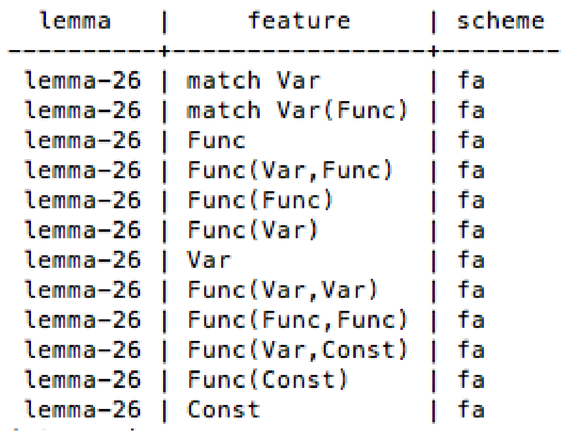
Abstract features for functions and lemmas are generated using the same tree-based approach as symbolic features. However, the strings generated do not contain any concrete symbols from the lemma/function, but rather an abstract representation of it, in order to only show its structure.

For example, the lemma *rev xs == qrev xs []* as we saw in a previous section, would, given a depth of 2, generate the abstract features shown in Figure 8. The abstract features of the functions of the same lemma are shown in Figure 9.



```
  lemma    |       feature       | scheme
-----------+---------------------+--------
 lemma-26  | Equals              | la
 lemma-26  | Equals(Func,Func)   | la
 lemma-26  | Equals(Func)        | la
 lemma-26  | Func                | la
 lemma-26  | Func(Var)           | la
 lemma-26  | Var                 | la
 lemma-26  | Func(Var,Const)     | la
 lemma-26  | Func(Const)         | la
 lemma-26  | Const               | la
```

Figure 8: The resulting abstract lemma features from *rev xs == qrev xs []*.



```
  lemma    |      feature       | scheme
-----------+--------------------+--------
 lemma-26  | match Var          | fa
 lemma-26  | match Var(Func)    | fa
 lemma-26  | Func               | fa
 lemma-26  | Func(Var,Func)     | fa
 lemma-26  | Func(Func)         | fa
 lemma-26  | Func(Var)          | fa
 lemma-26  | Var                | fa
 lemma-26  | Func(Var,Var)      | fa
 lemma-26  | Func(Func,Func)    | fa
 lemma-26  | Func(Var,Const)    | fa
 lemma-26  | Func(Const)        | fa
 lemma-26  | Const              | fa
```

Figure 9: The resulting abstract function features from *rev xs == qrev xs []*.

Different functions or data types are not distinguished in the abstract feature extraction schemes, leading to fewer features compared to symbolic feature extraction.

The abstract symbol *Func* is used for named function application. There are two other cases relating to functions. Firstly, when a type signature for a function is the only information stated in a lemma or function. For example, *filter(a:=>Boolean, list)*, which takes a function from type *a* to type Bool and a list, would be represented by *Func(FuncType, Var)*. Secondly, when lambda functions are used in a lemma or function then they are represented by the *Lambda* feature.

3.3.4 Analytical features

Analytical features contain metadata that is derived by analysing symbolic and abstract features for a lemma or function. Different analytical schemes are run over different sets of features, so a set of symbolic lemma features are not analysed the same way as a set of symbolic function features, for example.

One feature that is run for every analytical scheme is **popular**, which is the most popular feature generated (before removing duplicates). This is dependent on depth. For example, in $rev(rev(rev(xs))) == rev(xs)$, the most popular 1-depth symbolic feature is *rev*.

Lemmas containing the same most popular feature are more similar, and are more probable to have more similar proof data. This feature is prefixed with *abstract* when in an abstract analytical scheme.

Analysing symbolic features When analysing symbolic lemma features, the following structural information is made features:

- **associative**: If the lemma is stating the associativity property. The rule for this is that the lemma fulfills the condition of being **f a (f b c) == f (f a b) c**.
- **commutative**: If the lemma is stating the commutativity property. The rule for this is that the lemma fulfills the condition of being **f a b == f b a**.

The motivation for having these features is that lemmas on commutativity and associativity are generally very similar, which might indicate similar proofs.

A note on extraction depth When running the module, the user can select the depth which is used when building subtrees as seen in section 3.3.2. Some analytical results are independent of this depth. For example, when exploring the inner function depth of a lemma, its body is explored independently from the abstract features generated. However, other analytical results such as length of features are affected by the depth.

Analysing abstract features Abstract features are analysed in a similar way to symbolic features. Abstract lemma features are only analysed with the most popular feature. However, abstract function features are also analysed with the two following features:

- **nArgs**: the number of arguments of a function. The number of arguments of functions tells us something about functions, and the lemma using them, from a structural point of view. If functions contain more arguments, this might indicate another way approach to proving a lemma.
- **tailRecursive**: in this project, a function is defined as tailRecursive if a function has a body, for some pattern, which is **f args = f otherargs** (meaning that the top-level function of its own body must be itself), and that at least one of the arguments in the body is a function application.

3.4 Machine learning for induction variable selection

We have described how to populate a library with successful lemma proofs and to extract features and store them in a database. Now, it is time to extend HipSpec to learn from the data, and use the knowledge to classify new lemmas.

Learning To learn from the data, the idea is to use supervised learning on lemmas in order to divide them into classes. More specifically, the variable(s) that the HipSpec did induction on when it proved the lemma is used as the class. Hence, we will divide the set of lemmas into the classes ”first variable”, ”second variable, ”first and second variable”, and so on.

Since the induction variable(s) are the classes, it is straightforward to find out which induction variable to use for a new lemma: simply extract features from the lemma, and see which induction variable it is most similar to (by classifying).

An implementation of Bernoulli Naive Bayes from the scikit-learn package [22] is used. This is fed with all previously proven lemmas, and the induction variable(s) they were proven with. The classifier is fitted to the data, and then saved to disk.

Classifying Whenever HipSpec tries to prove a conjecture, it should now find the induction variable order. This is done in the following way:

1. Extract features from the conjecture
2. Load the trained classifier from disk
3. Classify the conjecture by feeding the features into the classifier

The output is an ordering of the ways of doing induction, for example "[[1], [0], [0,1]]", or "[second, first, first and second]". HipSpec then executes the external theorem provers in this order. This can be seen as adding a step to the main HipSpec loop which asks for the induction variables, as seen in Figure 10.

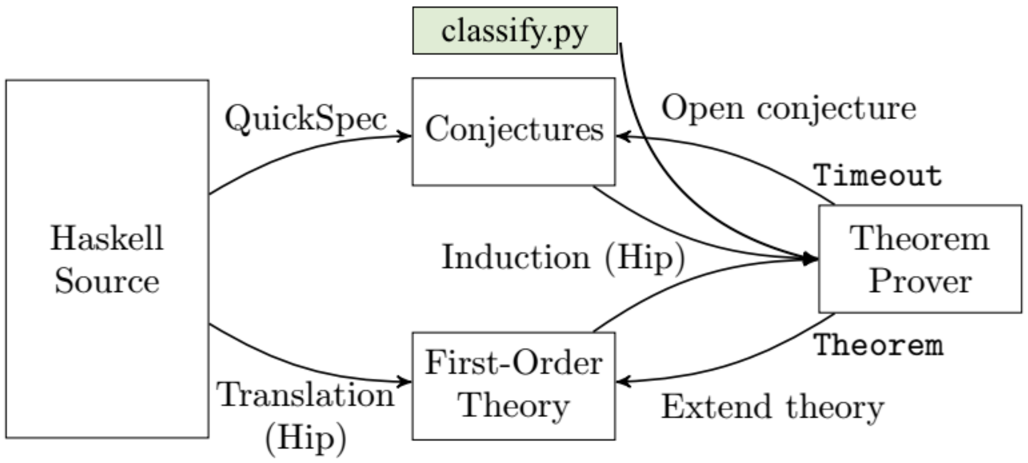


Figure 10: The main HipSpec loop now also invokes *classify.py* when proving a conjecture to get the induction ordering. Original picture from [6], but modified with showing *classify.py*.

4 Results and evaluation

This chapter describes the results of this project, and discusses possible sources of improvement. The machine learning experiments and their results are also presented and discussed in detail.

4.1 Evaluating the proof output

The proof output format did become very abstract, more akin to a proof plan than a proof. We recall the criteria specified by Böhme and Weber [3] (see section 2.2.2). Although these criteria are better suited for more low-level proofs, some of them are nonetheless useful for evaluating our format.

The format is human-readable, in particular because lemma names were also added to HipSpec, and thus could be used when referring to lemmas. A possible improvement for readability, though, would be to store induction variables as strings rather than De Bruijn indices (e.g. ”xs” rather than ”0”).

Another criterion, defined by ourselves, was that the proof plans should allow the user to reproduce a more detailed proof if needed. This is accomplished by storing which prover was used and which arguments it was invoked with. Worth noting is that we do not store the concrete proof obligations the external prover proved (the actual base case and step case(s)), but these can easily be derived given the expression and the induction variable(s).

4.2 Evaluating HipSpec’s performance

After HipSpec had been extended to choose induction variable based on which class the current lemma was predicted to belong to, it was time to run it on many lemmas and see how it fared.

A framework was built that runs HipSpec on every *.smt*2-file in a directory. After each run, it writes to a log file where it reports how long it took to run, whether it succeeded or failed, and whether it was actually proved using the predicted induction variable.

Halfway through the project, though, we realised that HipSpec executes all possible induction attempts for a lemma in parallel. This means that the induction order does not matter, and thus that a speedup would not result from solving the induction selection problem. This fact rendered this evaluation method unusable, so other methods were tried as well.

4.3 Evaluation of supervised learning

Different feature extraction schemes were evaluated by how well they performed at predicting the induction variables during supervised learning. A framework was built which enumerates all possible machine learning configurations and then tests each one using 5-fold cross-validation. Feature extraction schemes and depth were varied.

After running all configurations, the framework prints all of them to the screen, sorted by success rate. Now, the actual experiments performed within this framework will be described.

4.3.1 Guessing induction variable

This test tried to predict which induction variable lemmas were proven with (e.g. first, second, first and second, ...). Because many lemmas in the data set were proven without induction (see 4.5.2), these were filtered out beforehand and therefore not included in the test.

Setup As sample data, 96 lemmas proven by induction were used. These were assigned the induction variable(s) they were proven with as labels, with 66 of the lemmas having just the first variable as induction variable. Bernoulli Naive Bayes was used as algorithm, and inverse document frequency was used to give different feature weights. Different combinations of all 8 schemes were tried at each depth level from 1 to 3.

Results The best result was using depth 2 with only the *als* scheme, at 72.47 percent. See Table 1 for the top 20 best results.

Accuracy	Depth	Schemes
0.7247	2	als
0.7161	2	ala, afs, als
0.7161	2	afs, als
0.7055	2	ala, als
0.7055	1	la, afa, als
0.6987	2	ala, afa, afs, als
0.6974	2	afa, afs, als
0.6968	1	la, ala
0.6968	1	la, als
0.6955	3	fa, ls, ala, afa, afs, als
0.6955	3	fa, ls, ala, afa, afs
0.6955	3	fa, ls, ala, afa, als
0.6955	3	fa, ls, ala, afa
0.6955	3	fa, ls, ala, afs, als
0.6955	3	fa, ls, ala, afs
0.6955	3	fa, ls, ala, als
0.6955	3	fa, ls, ala
0.6955	3	fa, ls, afa, afs, als
0.6955	3	fa, ls, afa, afs
0.6955	3	fa, ls, afa, als

Table 1: The 20 best results out of 765 combinations.

4.3.2 First or other induction variable

Following the previous experiment, it would be interesting to see if better estimations would come from just having two classes, given the small sample size available. All lemmas which used another induction variable combination than just the first variable were grouped together as one class.

Setup As sample data, 96 lemmas proven by induction were used. These were assigned either *variable 0* or *other variable combination*. Bernoulli Naive Bayes was used as algorithm, and inverse document frequency was used to give different feature weights. Different combinations of all 8 schemes were tried at each depth level from 1 to 3.

Results The best result was using depth 2 with only the *als* scheme, at 71.89 percent. See Table 2 for the top 20 best results.

Accuracy	**Depth**	**Schemes**
0.7189	2	als
0.7089	2	fa, afs, als
0.7089	2	afs, als
0.7089	1	la, afs
0.7084	3	als
0.6984	2	ala, als
0.6984	1	fa, la, afa, afs
0.6984	1	fa, la, afs, als
0.6984	1	fa, la, afs
0.6984	1	la, als
0.6984	1	la
0.6979	3	fa, ls, ala, als
0.6979	3	fa, ls, als
0.6884	3	ala, als
0.6879	3	fa, la, ls, ala, afa, afs, als
0.6879	3	fa, la, ls, ala, afa, als
0.6879	3	fa, la, ls, ala, als
0.6879	3	fa, la, ls, afa, afs, als
0.6879	3	fa, la, ls, afa, als
0.6879	3	fa, la, ls, afa

Table 2: The 20 best results out of 765 combinations.

4.3.3 Discussion on supervised learning experiments

The best results for both experiments are almost exactly equal to guessing three more lemmas correctly, compared to just guessing the first variable for every lemma which would give 68.75 percent correct.

Despite some small success, the results are by far not good enough to really show that the induction variable can be estimated by these feature extraction schemes. It is worth noting that both of these experiments presupposes that it is possible to correctly predict a lemma's induction needs based on its similarity to other lemmas. This may be a faulty assumption: it is possible that no meaningful correlation exists between induction variables and term structure.

In other words, this evaluation method may not be sufficient to assess a feature extraction scheme's general fitness – if this method reports unsatisfactory results, it is still possible that the feature extraction scheme may be useful for other tasks. This is why another kind of validation was also performed.

4.4 Evaluation of unsupervised learning

An unsupervised clustering algorithm was also run for the different feature extraction schemes, and the resulting clusters were examined by hand. The results of this method are harder to quantify, but it has the advantage that it is not dependent upon the induction variable selection problem.

In all clustering tests, we used 50 lemmas from the *prod* benchmarks. The K-means algorithm was used to generate clusters, and inverse document frequency was used to weight the features. When comparing the resulting clusters, they were checked manually to see if any patterns could be discerned between lemmas in the same cluster.

4.4.1 Clustering lemmas using function features

Setup The parameters *scheme*, *depth* and *clusters* were varied according to Table 3.

Number	Scheme	Depth	Clusters
1	fa	1	2
2	fa	2	2
3	fa	3	2
4	fa	1	8
5	fa	2	8
6	fa	3	8
7	fa	1	20
8	fa	2	20
9	fa	3	20
10	fs	1	2
11	fs	2	2
12	fs	3	2
13	fs	1	8
14	fs	2	8
15	fs	3	8
16	fs	1	20
17	fs	2	20
18	fs	3	20

Table 3: Experimental combinations for lemma clustering using function features. Highlighted rows indicate interesting results.

Results While most combinations showed no or non-conclusive patterns, some the highlighted combinations (with the color green, numbers 4, 7 and 8) from Table 3 consistently formed the following cluster (where lemmas on tail-recursion are bold):

- **revflat xs = qrevflat xs []**
- **rev (qrev xs []) = xs**
- length (xs++xs) = double (length xs)
- **qrev (qrev xs []) [] = xs**
- **rotate (length xs) xs = xs**
- **fac n = qfac n b**

- **(n*m) = mult2 n m Z**
- (n‘elem‘ys) => (n‘elem‘(xs++ys))

Yellow highlighting indicate similar clustering but not quite as good. Combination number 5 formed the above cluster with an extra lemma.

Orange highlighting indicate stable, consistent clustering but not as good as yellow or green. Combinations 13-15 formed the following stable cluster, while the other lemmas on tail-recursion were shattered among other lemmas:

- **qrev (qrev xs []) [] = xs**
- **rotate (length xs) xs = xs**
- **fac n = qfac n b**
- **(n*m) = mult2 n m Z**
- (n‘elem‘ys) => (n‘elem‘(xs++ys))

Discussion The above results indicate that abstract function features are better for grouping tail-recursive lemmas than symbolic function features. However, we found no clear explanation to why this was the case.

4.4.2 Clustering lemmas using lemma features

Setup The parameters *scheme*, *depth* and *clusters* were varied according to Table 4.

Number	Scheme	Depth	Clusters
1	la	1	2
2	la	2	2
3	la	3	2
4	la	1	8
5	la	2	8
6	la	3	8
7	la	1	20
8	la	2	20
9	la	3	20
10	ls	1	2
11	ls	2	2
12	ls	3	2
13	ls	1	8
14	ls	2	8
15	ls	3	8
16	ls	1	20
17	ls	2	20
18	ls	3	20

Table 4: Experimental combinations for lemma clustering using function features. Highlighted rows indicate interesting results.

Results Combination 1 consistently created the following cluster (tail-recursive lemmas are bold):

- length (xs++ys) = length (ys++xs)
- half (length (xs++ys)) = half (length (ys++xs))
- even (n+m) = even (m+n)
- even (length (xs++ys)) = even (length ys+length xs)
- **rev xs = qrev xs []**
- **revflat xs = qrevflat xs []**
- length (xs++xs) = double (length xs)

- **qrev (qrev xs []) [] = xs**
- **rotate (length xs) xs = xs**
- **fac n = qfac n b**
- **(n*m) = mult2 n m Z**
- (n‘elem‘ys) => (n‘elem‘(xs++ys))

Combination 2 created the same cluster, but also with additional tail-recursive lemma **rev (qrev xs []) = xs**.

Orange combinations indicate that clustering of some of the tail-recursive lemmas were successful, but they were generally less than six. The tail-recursive vs. non-tail-recursive 1-to-1 ratio in these clusters maintained relatively stable for all highlighted combinations.

Discussion These results indicates that abstract lemma features are notably better at clustering lemmas on tail-recursion. This goes in line with the previous results from the experiments in section 4.4.1, where the abstract features were slightly better than symbolic features.

For abstract lemma features, the feature *Const* were occurring almost only in the tail-recursive lemmas, which might have helped the clustering in this experiment, but despite this, it did not cluster as well as with abstract function features.

Further comparing the results to section 4.4.1, this suggests that function features are better to cluster and isolate lemmas on tail-recursion. This might depend on more information about the functions and that tail-recursive functions distinguish themselves in some way.

4.4.3 Clustering lemmas using analytical features

Setup After experimenting with symbols and abstract features, we tested clustering using the analytical extraction schemes, which create features based on the symbolic features or abstract features. The parameters *scheme*, *depth* and *clusters* were varied according to Table 5.

Number	Scheme	Depth	Clusters
1	ala	1	2
2	ala	2	2
3	ala	3	2
4	ala	1	8
5	ala	2	8
6	ala	3	8
7	ala	1	20
8	ala	2	20
9	ala	3	20
10	als	1	2
11	als	2	2
12	als	3	2
13	als	1	8
14	als	2	8
15	als	3	8
16	als	1	20
17	als	2	20
18	als	3	20
19	afa	1	2
20	afa	2	2
21	afa	3	2
22	afa	1	8
23	afa	2	8
24	afa	3	8
25	afa	1	20
26	afa	2	20
27	afa	3	20
28	afs	1	2
29	afs	2	2
30	afs	3	2
31	afs	1	8
32	afs	2	8
33	afs	3	8
34	afs	1	20
35	afs	2	20
36	afs	3	20

Table 5: Experimental combinations for lemma clustering using function features. Highlighted rows indicate interesting results.

Results Using combinations 22, 23 and 26 consistently formed the following cluster (lemmas on tail-recursion are bold):

- length (xs++ys) = length (ys++xs)
- half (length (xs++ys)) = half (length (ys++xs))
- even (n+m) = even (m+n)
- even (length (xs++ys)) = even (length ys+length xs)
- **revflat xs = qrevflat xs []**
- **rev (qrev xs []) = xs**
- length (xs++xs) = double (length xs)
- **qrev (qrev xs []) [] = xs**
- **rotate (length xs) xs = xs**
- **fac n = qfac n b**
- **(n*m) = mult2 n m Z**
- (n‘elem‘ys) => (n‘elem‘(xs++ys))

Combinations number 6 and 9 consistently formed a similar cluster:

- half (length (xs++ys)) = half (length (ys++xs))
- even (n+m) = even (m+n)
- even (length (xs++ys)) = even (length ys+length xs)
- **rev xs = qrev xs []**
- **qrev (qrev xs []) [] = xs**
- **rotate (length xs) xs = xs**

- **fac n = qfac n b**
- **(n*m) = mult2 n m Z**
- (n‘elem‘ys) => (n‘elem‘(xs++ys))

Discussion The cluster formed by combinations 22, 23 and 26 is the same as the one formed by abstract lemma features in section 4.4.2, despite analysing abstract function features in this case. In addition, both presented clusters here contain some additional lemmas like the clusters from section 4.4.2 also do. These results might indicate that analytic schemes cluster lemmas in similar ways as lemma features, with the same drawbacks such as worse isolation in comparison to function feature clustering.

It is worth noting that only abstract analytic schemes, *ala* and *afa*, seem useful when clustering tail-recursive lemmas. This is in line with previous experiments, in section 4.4.1 and section 4.4.2. Here, it is also reasonable to assume that the feature **tailRecursive** from the *afa* scheme is helping.

4.4.4 Clustering functions

Setup 31 functions, the same used in the 50 earlier clustering lemmas, were used as input data to a K-means clustering algorithm. Inverse document frequency was used to weight the features. The parameters *scheme*, *depth* and *clusters* were varied according to Table 6.

Number	Scheme	Depth	Clusters
1	fa	1	2
2	fa	2	2
3	fa	3	2
4	fa	1	8
5	fa	2	8
6	fa	3	8
7	fa	1	20
8	fa	2	20
9	fa	3	20
10	fs	1	2
11	fs	2	2
12	fs	3	2
13	fs	1	8
14	fs	2	8
15	fs	3	8
16	fs	1	20
17	fs	2	20
18	fs	3	20

Table 6: Experimental combinations for function clustering. Highlighted rows mark where the function *exp*, *mult* and *rotate* where consistently clustered together.

Results Many of the experiments consistently grouped the function *exp*, *rotate* and *mult* together, with varying number of other lemmas accompanying them. No clear pattern for tail-recursive functions were found.

Discussion Again, the abstract feature scheme seems more consistent and reliable. However, we did not observe the same patterns for tail-recursive functions here as we did with their lemmas.

4.5 Discussion of results

There may be several contributing factors to the lack of satisfying machine learning results. The size and quality of the data led to a small sample size, in combination to potentially too many features. These potential factors are discussed in this section.

4.5.1 Size of data

In general, having had more data would have been beneficial. Contrary to what the name suggests, TIP (Tons of Inductive Problems) contains as of yet only 385 lemmas, which may not be enough to discover useful correlations.

Additionally, the part of TIP that could be proven by HipSpec was further restricted to 229 lemmas. The reason for this is that the development version of HipSpec/QuickSpec was used, which cannot prove as many lemmas as the stable version. This resulted in fewer lemmas being available for supervised learning, since our supervised learning required induction variable(s) as labels.

4.5.2 Incorrect data

Another problem with the supervised learning data is that right now, HipSpec ”mislabels” some lemmas as not having used induction despite them having done so. This is because of how theory exploration works.

Given a user-stated lemma, the theory exploration might find a generalized variant of that lemma, and prove it using induction. Then, the original user-stated lemma would technically be proven without induction, since the induction took place in the proof of the generalized lemma.

This sort of ”indirect induction” is not captured in the current model; hence, only 96/229 lemmas are proven by induction from our point of view, and thus only 96 lemmas are available as training data for the supervised learning evaluation. Modelling this indirect induction in some fashion would probably increase the amount of available data.

4.5.3 Feature extraction

In addition to having too little data, it is possible that we had too many features. For some of the test runs, it turned out that we had more features than data points; that is, the dimensionality was higher than the sample size. This is generally a warning sign within machine learning, and might have affected the results. In these cases, it could be helpful to perform

Principal Component Analysis (PCA) in order to reduce the number of dimensions, but this was regrettably skipped in this work due to lack of time.

Another relevant observation is that all of the lemma features used in this work are binary – i.e., either present or non-present. We suspect that this might also impact the results, and be further motivation for using a dimension reducing technique such as PCA.

5 Related work

Machine learning has been used successfully in several applications. Here we will cover two improvements that have been done to theorem provers, MeSh and ACL2(ml), how they differed from us and how they performed in comparison.

5.1 MeSh

The fact selector of Sledgehammer, MeSh, uses machine learning in order to solve the premise selection problem, i.e. the task of filtering out relevant lemmas for the current proof goal [1]. How MeSh does feature extraction is described in section 2.2.3, and it was a big inspiration for the feature extraction done in this project.

The results of MeSh seem promising. In its evaluation, it is reported to be more performant than older methods, both when comparing results directly from the supervised learning, and when used in real-life testing of running times.

Hence, one could say that they were successful in solving the premise selection problem. In contrast, we tried to solve the induction selection problem but did not succeed. This could stem from a number of reasons. Firstly, Isabelle contains a much larger theory than TIP, and thus they had more data at their disposal. Secondly, and more importantly, it is not certain that the induction selection problem even is solvable (see 4.3.3).

5.2 ACL2(ml)

Another related project is ACL2(ml), which is an extension to the ACL2 prover [11]. It first uses unsupervised learning to cluster lemmas, and then these clusters are used for generating new lemmas that are (hopefully) useful for the current proof goal.

The feature extraction done in ACL2(ml) is quite involved: for example, it represents term trees with adjacency matrices and it uses recurrent clustering. In contrast, the features in this project are merely strings.

Additionally, the ACL2(ml) project uses *dense matrices* as input to their machine learning algorithm. This refers to feature matrices, and it means that they reduce the total number of features to a small amount which is done using a custom-built technique. This project, on the other hand, uses *sparse matrices* for representing features which is much simpler.

ACL2(ml) did achieve some success with their clustering: their approach scaled well when more data was added (the principal clusters remained the same), and lemmas within clusters generally had similar proofs. Compared to us, they had access to more data, and they had a more robust way of evaluating the performance of their clustering.

5.2.1 Clustering comparison with HipSpec

In the paper on ACL2(ml) [11], 996 lemmas were clustered using a much more complicated feature extraction method than that of this project. A working hypothesis of this project was that it could be made much simpler.

Their experiments showed that the following three lemmas about function equivalence of recursive and tail-recursive functions were consistently grouped together:

- **fact-tail n = fact n**
- **fib-tail n = fib n**
- **power-tail n = power n**

We added these three lemmas and their functions to our 50 lemmas and 31 functions, and tested clustering similar to the previously described experiments.

We noted, as for lemma clustering, that all scheme combinations generally clustered the above lemmas together in the *larger cluster*, and not together with the other lemmas on tail recursion. Besides that, the clusters were very similar to how they formed without these added lemmas and functions.

Considering these results, we can not conclude that our feature extraction works as well as ACL2(ml). Therefore, the hypothesis was not verified.

6 Further work and conclusions

This chapter describes possible future extensions to HipSpec made possible by this report and summarizes its main contributions.

6.1 Further work

This project has laid the ground work for future implementations in HipSpec using machine learning. Here, we suggest suitable additions that can now be implemented, and improvements to the work done in this project.

6.1.1 More and better data

In general, we judge that we had too little data. This could be alleviated by TIP growing in size, including lemmas from more branches of mathematics. It would also be possible to add more data sources (i.e. Isabelle theories) for the machine learning.

Additionally, when the newest HipSpec version has matured, more data will be available for supervised learning. For induction classification only 96 lemmas were used, but this figure could be increased much if HipSpec managed to prove more lemmas (and if cases of ”indirect induction” are modelled, see 4.5.2).

6.1.2 Conjecture ordering

Now that the persistent library is in place, other problems besides induction selection can be tackled. One such problem is to decide the order in which to prove conjectures within the HipSpec loop.

This order is important, since there generally can be many dependencies between the conjectures. Some of the conjectures might be easy to prove, and thus should be proven early, while other conjectures are more complicated and require other lemmas as background theory, and thus should be proven late.

The conjecture order is currently decided analytically (smaller conjectures are attempted before bigger conjectures). This could potentially be improved by using machine learning to decide the order, to make theory exploration run faster.

6.1.3 Theory exploration

We recall that HipSpec uses QuickSpec to generate conjectures from relevant function definitions. It could be possible to utilise machine learning in order to make QuickSpec discover more relevant lemmas.

When invoking the newest version of QuickSpec, it is possible to supply a *schema* that roughly specifies the shape that conjectured lemmas ought to take. It could be possible to use machine learning in order to generate such a schema, by first finding similar theorems, and then looking at the lemmas they use in their proofs.

6.1.4 Utilising clustering

The results from clustering are not only of theoretical interest, but may also be of practical use. For example, our results indicate that it is relatively easy to cluster lemmas that talk about tail recursive functions. This could possibly be used when proving such lemmas, since proofs of lemmas about tail recursive functions usually begin by proving a generalisation of the lemma.

Another potential usage of clustering is to use a cluster to find premises for a proof of a lemma. If a lemma is clustered together with other lemmas, it might mean that some proof similarities exist. In Sledgehammer, a k-nearest neighbour algorithm is used to find other lemmas similar to the proof goal [1]. It would be interesting to test these feature extraction schemes for a similar case.

One could also, in the way done in ACL2(ml), use the clusters as a basis for symbolically generating potential premises.

6.2 Conclusion

The goals of this project was to lay the groundwork for utilising machine learning in HipSpec and to evaluate how machine learning might be used.

The groundwork consisted of defining a high-level format for proofs. Furthermore, a persistent library was implemented, where HipSpec stores everything it succeeds to prove (in the aforementioned proof format).

These changes allow HipSpec to remember what has happened in the past, and they allow the user to collect data regarding proven lemmas.

This is fundamental functionality if one wants to leverage previously proven lemmas in new proof attempts, for example through machine learning.

On the machine learning side, a module for feature extraction from lemmas was implemented, as well as several different feature extraction schemes. This module is now used internally in HipSpec, but can also easily be invoked as a standalone module, for future machine learning implementations.

Finally, several experiments with supervised and unsupervised learning were performed. We did not succeed in solving the induction selection problem, either because of lack of data or because no correlation between induction variables exists. However, one experiment with unsupervised learning succeeded in clustering lemmas on tail recursive functions. This shows that using machine learning within theorem proving could be successful, and that further work is warranted.

All in all, a solid foundation has been built that enables future work on many other machine learning applications in addition to the ones tried here. Also, our results suggest that there is cause for hope for machine learning within theorem proving.

References

[1] Jasmin Christian Blanchette; David Greenaway; et al. “A learning-based fact selector for isabelle/HOL”. In: *Journal of Automated Reasoning* (2016). Pending publication.

[2] Kenneth I Appel and Wolfgang Haken. *Every planar map is four colorable*. Vol. 98. American mathematical society Providence, RI, 1989.

[3] Sascha Böhme and Tjark Weber. “Designing proof formats: A user’s perspective”. In: *Proof eXchange for Theorem Proving* (2011), pp. 27–32.

[4] Koen Claessen and John Hughes. “ICFP ’00 Proceedings of the fifth ACM SIGPLAN international conference on Functional programming”. In: ACM, 2000, pp. 268–279.

[5] Koen Claessen, Nicholas Smallbone, and John Hughes. “QuickSpec: Guessing formal specifications using testing”. In: *Tests and Proofs*. Springer, 2010, pp. 6–21.

[6] Koen Claessen et al. “Automating inductive proofs using theory exploration”. In: *Automated Deduction–CADE-24*. Springer Berlin Heidelberg, 2013, pp. 392–406.

[7] Koen Claessen et al. “TIP: Tons of Inductive Problems”. In: *Proceedings of the Conference on Intelligent Computer Mathematics (CICM) 2015*. 2015, pp. 333–337.

[8] Leonardo De Moura and Nikolaj Bjørner. “Z3: An efficient SMT solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[9] Thomas Hales et al. “A formal proof of the Kepler conjecture”. In: *arXiv preprint arXiv:1501.02155* (2015).

[10] John Harrison. “HOL Light: A tutorial introduction”. In: *Formal Methods in Computer-Aided Design*. Springer. 1996, pp. 265–269.

[11] Jónathan Heras et al. “Proof-pattern recognition and lemma discovery in ACL2”. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer. 2013, pp. 389–406.

[12] Thomas Hillenbrand et al. “Waldmeister-high-performance equational deduction”. In: *Journal of Automated Reasoning* 18.2 (1997), pp. 265–270.

[13] Joe Hurd. "First-order proof tactics in higher-order logic theorem provers". In: *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports* (2003), pp. 56–68. URL: http://www.gilith.com/research/papers/metis.pdf.

[14] 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.

[15] Moa Johansson et al. "Hipster: integrating theory exploration in a proof assistant". In: *Intelligent Computer Mathematics*. Springer, 2014, pp. 108–122.

[16] Cezary Kaliszyk, Josef Urban, and Jirı Vyskocil. "Efficient semantic features for automated reasoning over large theories". In: *IJCAI*. Vol. 15. 2015, pp. 3084–3090.

[17] Laura Kovács and Andrei Voronkov. "First-order theorem proving and Vampire". In: *Computer Aided Verification*. Springer. 2013, pp. 1–35.

[18] Daniel Kühlwein et al. "MaSh: machine learning for Sledgehammer". In: *Interactive Theorem Proving*. Springer, 2013, pp. 35–50.

[19] Lawrence C Paulson and Jasmin Christian Blanchette. "Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers". In: *IWIL-2010* (2010).

[20] Piotr Rudnicki. "An overview of the Mizar project". In: *Proceedings of the 1992 Workshop on Types for Proofs and Programs*. 1992, pp. 311–330.

[21] Stephan Schulz. "System Description: E 1.8". In: *Proc. of the 19th LPAR, Stellenbosch*. Ed. by Ken McMillan, Aart Middeldorp, and Andrei Voronkov. Vol. 8312. LNCS. Springer, 2013.

[22] *Scikit-learn*. Mar. 29, 2016. URL: http://scikit-learn.org/stable/.

[23] *Scikit-learn Clustering*. May 21, 2016. URL: http://scikit-learn.org/stable/modules/clustering.html#k-means.

[24] *Scikit-learn KMeans*. May 4, 2016. URL: http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.

[25] Tobias Nipkow; Lawrence C Paulson; Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002. URL: `http://isabelle.in.tum.de/doc/tutorial.pdf`.

[26] Harry Zhang. “The optimality of naive Bayes”. In: *AA* 1.2 (2004), p. 3. URL: `http://www.cs.unb.ca/~hzhang/publications/FLAIRS04ZhangH.pdf`.

A Readme for this project's ML extensions

A.1 Prerequisites

- Python 2.7+
- PostgreSQL

A.2 batch.py

```
python batch.py /path/to/directory /path/to/library.lib
```

Runs 'emna' on all problems in a directory. Useful for building a .lib file, or for evaluating how emna performs. Outputs useful information for each problem into './batch.log'.

A.3 Choose induction variables by machine learning

When you have a .lib file that you want to use in order to make predictions of induction variables, run:

1. extractFeatures
2. learn

Subsequent invocations of 'emna' will use the generated classifier to choose induction order.

A.3.1 extractFeatures.hs

```
extractFeatures /path/to/library.lib depth
feature-extraction-schemes
```

Extracts features from the given library into the PostgreSQL database. Will wipe the database beforehand, so use with care! Looks for the environment variables 'HS_DB_NAME', 'HS_DB_HOST', 'HS_DB_USERNAME' and 'HS_DB_PASSWORD'.

A note on filtering and induction variables:

- In extractFeatures.hs, you can select whether or not to filter away lemmas with no induction variable. This is done by the function filterNonInductiveLemmas.

- In FeatureExtraction.hs, in the function insertLemmas, you can choose if you will group all non-first-variable-only-lemmas as one group by using the statement which returns either [0] or [1].
- Make sure to recompile after these changes before running any tests.

A.3.2 extractFunctionFeatures.hs

Works like extractFeatures.hs, but only with fs, fa, afa and afs extraction schemes.

A.3.3 create_classifier.py

```
python create_classifier.py [/path/to/data]
```

Creates 'classifier.pkl' from the data currently in the PostgreSQL database.

A.3.4 use_classifier.py

Used only internally by HipSpec, there should be no need to call this directly.

```
python use_classifier.py stringifiedFeatures [/path/to/data]
```

Reads 'classifier.pkl' and classifies 'stringifiedFeatures' by it, returning the most probable classes on stdout.

A.4 Evaluating feature extraction schemes

A.4.1 supervised.py

```
python supervised.py
```

Runs cross-validation of supervised learning. Evaluates many different configurations and prints a table of results to screen.

The script varies feature extraction schemes, depths and ML engines. To change which combinations it tries, you should edit the script.

A.4.2 unsupervised.py

```
python unsupervised.py number_of_clusters path/to/lib.tiplib
depth [feature extraction schemes]
```

Runs an unsupervised clustering algorithm, given some feature extractions schemes and depth, and prints the resulting clusters to screen.

Example invocation:

```
python scripts/unsupervised.py 8 ./data/lib.tiplib 4 la ls
```

A.5 Other files

A.5.1 db.py

Used only internally by other scripts, there should be no need to call this directly.

Connects to the local PostgreSQL database, and supplies utility functions for reading lemmas/classes/features from it.

A.5.2 tables.sql

Contains definitions of PostgreSQL tables.