THE UNIVERSITY
of EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Machine learning for inductive theorem proving

Yaqing Jiang

Doctor of Philosophy
Centre for Intelligent Systems and their Applications
School of Informatics
University of Edinburgh

2019

Abstract

Over the past few years, machine learning has been successfully combined with automated theorem provers (ATPs) to prove conjectures from various proof assistants. However, such approaches do not usually focus on inductive proofs. In this work, we explore a combination of machine learning, a simple Boyer-Moore model and ATPs as a means of improving the automation of inductive proofs in HOL Light. We evaluate the framework using a number of inductive proof corpora. In each case, our approach achieves a higher success rate than running ATPs or the Boyer-Moore tool individually. An attempt to add the support for non-recursive type to the Boyer-Moore waterfall model is made by looking at proof automation for finite sets. We also test the framework in a program verification setting by looking at proofs about sorting algorithms in Hoare Logic.

Lay Summary

Theorem proving systems can perform formal verification of mathematical proofs to prevent unjustified steps and errors in them. One of the major challenges in formal verification is induction, which is required when reasoning about recursion. This is encountered in structures such as the loops of programs or natural numbers where the properties of such infinite types can be proven in a finite number of steps. There have been systems about the automation of inductive proofs, among which figures the Boyer-Moore model which we use for our work. Performing proofs by induction is difficult because intermediate lemmas are often needed to be figured out and proven first for the main proof to proceed. Machine learning has been successfully used to select useful lemmas from existing proofs and, combined with automated theorem provers (ATPs), to prove conjectures in many areas, but with limited support for inductive proofs. Therefore, we explore a combination of machine learning, the Boyer-Moore model, and ATPs as a means of improving the automation of such proofs. We evaluate the framework using a number of inductive proofs previously performed by humans. Machine learning is used to learn from these proofs and select the pre-proven theorems for Boyer-Moore. In each case, our approach is able to prove more theorems than running ATPS or the Boyer-Moore tool individually. We also test the framework in a program verification setting by looking at proofs about sorting algorithms. We extract mathematical problems from the algorithm and use our system to prove them.

Acknowledgements

First, I would like to thank my supervisor Jacques Fleuriot for his guidance over the course of the PhD. Thanks also to Petros Papapanagiotou, especially for his insights into the Boyer-Moore system. I am very grateful for the help from Phil Scott on OCaml and HOL Light as well as the advice for this work. I also received a lot of feedback from Paul Jackson during my PhD.

My parents have supported me a lot. They funded me since the beginning of my fourth year. Though I was not able to see them very often and help them using my knowledge or strength with things such as repairing the computer as other sons, they never complained but comforted me.

This work was supported by the China Scholarship Council (CSC). Thanks to everyone who told me not to panic, though the effects never lasted long.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text. We confirm that appropriate credit has been given within the thesis where reference has been made to the work of others, and that this work has not been submitted for any other degree or professional qualification except as specified.

Chapter 5, 6 and 7 of this thesis include work from the following jointly-authored publication:

- Jiang Y., Papapanagiotou P., Fleuriot J. (2018) Machine Learning for Inductive Theorem Proving. In: Fleuriot J., Wang D., Calmet J. (eds) *Artificial Intelligence and Symbolic Computation. AISC 2018. Lecture Notes in Computer Science*, vol 11110. Springer, Cham

The research described in this publication was performed as part of the current work by myself, under the supervision of Jacques Fleuriot and Petros Papapanagiotou.

(*Yaqing Jiang*)

Table of Contents

1 Introduction **1**

1.1 Inductive theorem proving 1

1.2 Machine learning and lemma selection 2

1.3 Improving inductive theorem proving with machine learning in HOL Light 2

1.4 Structure of the thesis 4

2 Theorem proving **7**

2.1 Interactive Theorem Proving 7

2.1.1 HOL Light Syntax 8

2.1.2 Edinburgh LCF 9

2.1.3 Conversions 10

2.1.4 Tactics 11

2.1.5 Conversionals and tacticals 11

2.1.6 Clausal Form 12

2.2 ATP and SMT solvers 12

2.2.1 METIS 13

2.2.2 Vampire 13

2.2.3 E prover 14

2.3 Recursive data type and Induction 15

2.3.1 Induction in HOL Light 16

2.3.2 The failure of Cut Elimination 17

2.4 Using ATPs to find ITP proofs 18

2.4.1 Translation between ITPs and ATPs 19

2.5 Systems for induction 21

2.5.1 The Boyer-Moore Model 22

2.5.2 The heuristics 23
2.5.3 The Shell Principle 26
2.5.4 Induction procedure 27
2.5.5 Extensibility 28
2.5.6 Other systems for induction 29
2.6 Conclusion 31

3 Machine learning for theorem proving 33
3.1 Machine learning 33
3.1.1 Supervised learning 34
3.1.2 Unsupervised learning 37
3.1.3 Online learning 37
3.2 Lemma selection 38
3.2.1 Non-machine learning methods for lemma selection 38
3.3 Machine learning for theorem proving 39
3.3.1 Machine learning algorithms used for lemma selection 39
3.3.2 Feature extraction 40
3.3.3 Dependency tracking 43
3.3.4 A quick note on incremental learning 45
3.3.5 Experimental methodology 46
3.3.6 Machine learning for inductive theorem proving 46
3.4 Conclusion 47

4 Methodology 49
4.1 Research strategy 49
4.1.1 Investigating failures 51
4.2 Infrastructure 52
4.2.1 Proof library 53
4.2.2 Boyer-Moore waterfall analysis 54
4.3 Experiment data 59
4.3.1 List theory 59
4.3.2 Arithmetic 61
4.3.3 The IsaPlanner Benchmark 61
4.3.4 Lemmas for Hoare Logic 62
4.3.5 Data split 63
4.4 Proof metrics 63

4.4.1 Proof complexity 64
4.4.2 Methodology 65
4.5 Conclusion 67

5 Improving the Boyer-Moore Model 69
5.1 Improve the waterfall with experiment 69
5.1.1 Removing clausal form heuristic 69
5.1.2 Generalising variables 70
5.1.3 HOL Light's automated procedures 71
5.1.4 Forced induction 72
5.1.5 Summary 73
5.2 The Multi-waterfall model 73
5.3 Extending induction heuristic to support non-primitive recursive definitions 75
5.3.1 Motivation 75
5.3.2 Lexicographic induction and related techniques 76
5.3.3 Methodology 77
5.4 Improving the user interface 78
5.4.1 Tracing proof attempts by Boyer-Moore 78
5.4.2 Generating proof scripts 80
5.5 Conclusion 82

6 Combining Boyer-Moore with machine learning 85
6.1 Improving machine learning for lemma selection 85
6.2 Using *HOL(y)Hammer* to find inductive proofs 86
6.3 Adding lemma selection to waterfall 86
6.3.1 Development environment 87
6.3.2 Lemma selection for internal and external tools 88
6.3.3 Direct induction 91
6.3.4 Fixing the translation to TPTP 92
6.3.5 Intermediate evaluation 93
6.4 Machine learning for selecting induction variable 94
6.4.1 Methodology 95
6.5 Conclusion 98

7 Evaluation 99

7.1 Multi-waterfall model 99
7.1.1 Choice of datasets 99
7.1.2 Experiments 100
7.1.3 Metrics 101
7.1.4 Results 101
7.1.5 Examples 104
7.1.6 Proof metrics 105
7.2 Selecting induction variables with machine learning 108
7.2.1 Experiment settings 108
7.2.2 Results 108
7.3 Support of non-primitive recursive definitions 112
7.3.1 Dataset 112
7.3.2 Experiment settings 112
7.3.3 Results 113
7.3.4 Simplification with preselected lemmas 116
7.4 Generality of our approach 118

8 Induction for finite set **121**
8.1 Finite sets in HOL Light 121
8.2 Simplified shell 122
8.3 Reformulation of induction rule 122
8.4 Choosing suitable induction variable 123
8.5 Experiment 124
8.5.1 Test data and environment 124
8.5.2 Results 125
8.6 Conclusion 128

9 Application: verification of a sorting algorithm **131**
9.1 Data for evaluation 131
9.1.1 Selection sort 131
9.1.2 Hoare Logic 132
9.1.3 Goal for the verification of selection sort 133
9.2 Total correctness of nested loops 136
9.3 Experiments 138
9.3.1 Supporting library 138
9.3.2 Goal split & simplify 138

9.3.3 Prove subgoals by Boyer-Moore automatically 139
9.3.4 Examples 139
9.4 Using terms as features 144
9.4.1 Generating term features 145
9.4.2 Limitation 147
9.4.3 Evaluation of term feature 147
9.5 Conclusion 151

10 Conclusion **153**
10.1 Achievements and limitations 153
10.1.1 Lemma selection for Boyer-Moore Model 154
10.1.2 Machine learning for selecting induction variable 155
10.1.3 Proof metrics 155
10.1.4 Finite sets 155
10.2 Future work 156
10.2.1 Machine learning for induction 156
10.2.2 Extending the Boyer-Moore functionality 158
10.3 Concluding Remarks 159

A Appendix **161**
A.1 Subgoals from splitting the VC goals in Fig 9.2 161
A.2 Testing goal for list theory 167
A.2.1 List core 167
A.2.2 List Hilbert 172
A.2.3 Poly.ml 178
A.2.4 Isaplanner benchmark 186
A.2.5 Hoare logic 191

Bibliography **193**

List of Figures

1.1 Our approach . . . 3

2.1 HOL Light abstract syntax . . . 9
2.2 Inductive proof for ADD_SYM . . . 16
2.3 The workflow of a hammer . . . 19
2.4 TPTP abstract syntax . . . 20
2.5 Diagram of the Waterfall Model . . . 23
2.6 The Boyer-Moore Model . . . 24

3.1 Tree of Statement 3.11 . . . 42
3.2 Tracked identifier . . . 45
3.3 Tracked dependency . . . 46

4.1 Research strategy . . . 50
4.2 Major iterations of improving Boyer-Moore . . . 50
4.3 Table of tracked theorem identifiers . . . 54
4.4 Table of dependencies . . . 55
4.5 Table of results . . . 56
4.6 Visualisation example . . . 57
4.7 Visualisation example . . . 58
4.8 Proof of "*CARD_NUMSEG_1*" . . . 65
4.9 Parse tree of the proof of "CARD_NUMSEG_1" . . . 66

5.1 COPRIME_EXP_IMP and its CNF . . . 70
5.2 Two waterfalls in parallel . . . 74
5.3 Proof search with multi-waterfall . . . 75
5.4 Records of (sub)goals . . . 80
5.5 JSON of (sub)goals . . . 80

6.1 Combining hammers in waterfalls 90
6.2 Optimised 3 waterfalls 92
6.3 Function must_pred 93

7.1 Coverage of proven theorems by the different methods in Table 7.2 . . 103
7.2 User and Boyer-Moore proofs for DROP_DROP 105
7.3 User proof for LENGTH_REVERSE 105
7.4 Average proof size of theorems by different methods in Table 7.2 . . . 106
7.5 Proof size of theorems by different methods in Table 7.2 107
7.6 Comparison of induction steps between *IndMl* and *RecAna* for *List*(*hilbert*) 109
7.7 Comparison of induction steps between *Lex* and *RecAna* for *List*(*hilbert*) 114
7.8 Comparison of induction steps between *Lex* and *IndMl* for *List*(*hilbert*) 114

9.1 Selection sort algorithm 132
9.2 Evaluation of selection sort 134
9.3 Goal 2.2 140
9.4 Goal 5.4 (simplified) 141
9.5 Goal 4.2 143
9.6 Goal 4.2 viewed diagrammatically 143
9.7 Lemma ranking comparison among the methods in Table 9.3 149

List of Tables

2.1 HOL Light syntax 8
2.2 Shell for the natural number type 27
2.3 Shell for the list type 27

3.1 An example of training data 34
3.2 An example of training data for ranking 36
3.3 Matrix for tree in Figure 3.1 42

4.1 List theory datasets 60
4.2 Arithmetic (Gödel) dataset 61
4.3 *IsaPlanner benchmark* 62
4.4 Hoare Logic dataset characteristics 62
4.5 Summary of all datasets 63

5.1 Initial experiment for comparing MESON and METIS 72
5.2 Success rates comparison after applying initial fix to Boyer-Moore . . 73

6.1 Performance of higher order ATP 86
6.2 Number of proven lemmas with different numbers of lemmas sent to ATPs 88
6.3 Using REWRITE_CONV with lemma selection 90
6.4 Success rates comparison between two and three-waterfall model . . . 91
6.5 Performance comparison with fixed TPTP translation 93
6.6 Heuristic settings for three waterfalls 94
6.7 Success rates comparison after combining machine learning and multi-waterfall to Boyer-Moore 94

7.1 Success rate of *Ind simp* and the original Boyer-Moore. 101
7.2 Success rates of the different configurations of Boyer-Moore for *Poly* . 102

7.3 Success rates comparison between Boyer-Moore *Multi-waterfall* and ATP . . . 102
7.4 Success rate comparison of using different lemmas for *Poly* . . . 104
7.5 Number of proven theorems . . . 108
7.6 Success rate of RecAna and Lexicographic induction(on lemmas for Hoare logic) . . . 115
7.7 Success rate of RecAna and Lexicographic induction *IsaPlanner benchmark* . . . 115
7.8 Success rate after adding simplification . . . 117

8.1 Size of test data . . . 125
8.2 Success rates of the different systems on finite set libraries . . . 126

9.1 Libraries for the verification of selection sorted . . . 139
9.2 Results of running Boyer-Moore to prove subgoals from Fig 9.2 . . . 140
9.3 Methods for comparing different features . . . 148

Chapter 1

Introduction

Over the past few years, machine learning has been successfully combined with automated theorem provers to prove conjectures from proof assistants by selecting appropriate lemmas. One of the limitations of such approaches is that inductive proofs are usually not supported. In this work, we try to combine machine learning techniques with an existing model for inductive proofs and investigate possible ways of improving it. We begin by describing the main involved concepts in the next sections.

1.1 Inductive theorem proving

Recursive data types and function definitions are widely used in theorem proving to deal with concepts such as natural numbers and lists. Theorems about recursive definitions often require inductive proofs. However, most theorem provers do not perform well with goals that require inductive theorem proving. For such cases, the user is usually required to provide a skeleton of the proof while the automated tools handle smaller sub-goals, such as those generated by induction.

Automated methods for inductive theorem proving do exist. ACL2 (Kaufmann et al., 2000), for example, is a system that evolved from the so-called *Boyer Moore* approach (which we use in our current work) and is successfully being used for the formalization of industrial problems. However, due to the *failure of Cut Elimination* (see Section 2.3.2), such methods often require the manual provision of suitable lemmas to help with the inductive proof (for example as *hints* in ACL2). Identifying such lemmas is a major challenge and the system relies on human expertise and understanding of

the problem and its context.

Proof planning (Bundy et al., 2005) has been applied to guide the proof search mainly for inductive proofs, which is implemented as the system CLAM and the *IsaPlanner* tool (Dixon and Fleuriot, 2003) for the proof assistant Isabelle (Nipkow et al., 2002). Proof planning usually incorporates lemma discovery techniques, which try to automatically speculate intermediate lemmas based on *the productive use of failure*. In this, the cause of failure is analysed and used to direct the search process (Ireland, 1992; Ireland and Bundy, 1996). The lemma discovery techniques include, for example, generalisation, which was also incorporated in the original Boyer Moore prover, but have had relatively limited success.

1.2 Machine learning and lemma selection

Automated theorem provers (ATPs) like Vampire (Kovács and Voronkov, 2013) and E (Schulz, 2013) and satisfiability modulo theories (SMT) solvers like Z3 (De Moura and Bjørner, 2008) are increasingly being used to facilitate the development of large proof corpora in systems such as Isabelle and HOL Light.

In order to use such external tools effectively, machine learning (ML) infrastructures have been developed to automatically select hundreds of potentially relevant lemmas whenever the user tries to prove a goal automatically. Sledgehammer (Paulson and Blanchette, 2010) in Isabelle and HOL(y)Hammer (Kaliszyk and Urban, 2015) in HOL Light are examples of two such ML systems.

Hammers, the collective name that is sometimes used for the lemma selection systems, generally focus on the problems that can be proven in one step or on a sub-problem inside a big proof. For instance, they might be able to prove the subgoals generated from induction. This then requires users to provide the skeleton of the proofs.

1.3 Improving inductive theorem proving with machine learning in HOL Light

Our main hypothesis is that machine learning techniques can be used to improve the existing automated inductive theorem proving model in HOL Light.

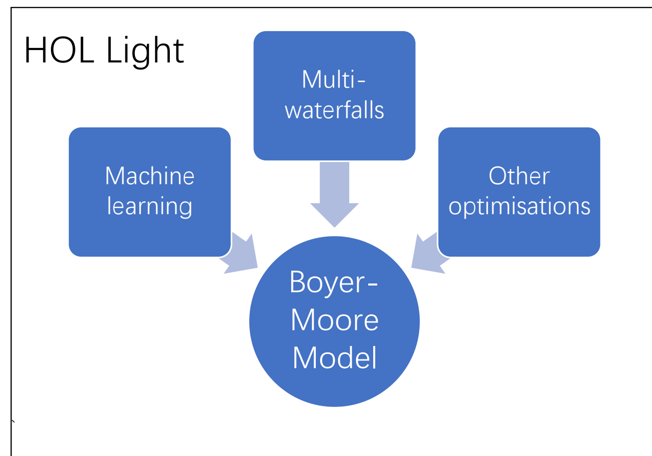


Figure 1.1: Our approach

Our investigation involves the integration of machine learning technology into the Boyer-Moore model and covers:

Selecting lemmas for proving subgoals: We investigate the potential use of machine learning to select lemmas that are available in big corpora to support automated inductive theorem proving. We aim to select suitable lemmas that can then be used to prove the subgoals that are generated after induction is applied or directly prove the goal without induction. According to *the failure of Cut Elimination* such lemmas are sometimes necessary for the inductive proof (see Section 2.3.2).

Selecting variable for induction scheme: We try to learn the selection of variable for induction from existing proofs.

The Boyer Moore implementation in HOL Light is used (see Chapter 2). During the development of our approaches various limitations of the Boyer Moore model were discovered, and attempts were made to solve them. Particularly, as we incorporate proof strategies that make use of machine learning techniques and ATPs within a Boyer-Moore style model, we decided to run these in parallel in a new environment we call a *multi-waterfall*. This can help circumvent the long waiting time that may arise due to large number of selected lemmas being provided to the Boyer Moore heuristics. More importantly, it allows various combination of proof steps, such as induction with simplification, and so on. Other fixes were also made, e.g. adjusting the heuristics in the model, which played an important role in supporting our proof techniques. These approaches are summarised in Fig 1.1.

We evaluate and compare our approaches with both Hammers and the Boyer Moore

model available in HOL Light. Libraries involving algorithm verifications were also manually created to test our approach on some formal verification problems. Proof metrics were also designed to measure the effort of proving theorems, instead of comparing the results just against success rate.

We consider the improved Boyer Moore system in HOL Light the main practical outcomes of our research. Along the way we also developed other tools such as the visualisation that helps us analyse the data.

Although our work is based on HOL Light, most of our ideas and approaches are generic and extensible. We believe that the multi-waterfall model, the improvements made to the Boyer Moore model as well as lemma selection process can also be used as a basis for the developments of automated inductive theorem proving tools in other proof systems.

1.4 Structure of the thesis

We organise the thesis as follows:

Overview of the theorem proving context (Chapter 2): We review aspects of theorem proving relevant to our work. In particular, inductive theorem proving and relevant techniques are introduced.

Machine learning methods and their applications in theorem proving (Chapter 3): Some relevant background information about machine learning techniques and existing work combining it with theorem proving are discussed.

Description of the methodology and datasets (Chapter 4): The general strategy and datasets used in our development of Boyer-Moore system are introduced in this chapter.

Improvements to the Boyer Moore Model (Chapter 5): We introduce the multi-waterfall model, adding support for more general recursive definitions, and providing the visualisation e.g. proof attempts of multi-waterfalls.

The Boyer Moore Model with machine learning methods (Chapter 6): We add lemma selection for use by ATPs as a heuristic, i.e. as a procedure of the Boyer Moore waterfall model. We investigate the use of machine learning methods to select

induction variables.

Evaluation and discussion (Chapter 7): The experiments and evaluation of the Boyer Moore Multi-waterfall model and other methods for proofs involving recursive types are discussed.

Induction for non-recursive types (Chapter 8): We investigate and add support for non recursive types and finite sets in Boyer-Moore multi-waterfall model.

Experiments with algorithm verification (Chapter 9): A corpus about sorting algorithm expressed in Hoare logic is created and used to investigate the potential usefulness of our approach for the formal verification of algorithms.

Conclusion and future work (Chapter 10): The thesis concludes with a summary of the achievements and limitations, and with pointers to future work.

Chapter 2

Theorem proving

We now give a brief overview of the theorem proving context for our work. We start with an overview of interactive theorem proving, followed by a description of some relevant automated theorem provers. We also introduce *hammer* systems, which use automated theorem provers to find proofs for interactive theorem proving. In addition, we review the *Boyer Moore* model for inductive theorem proving, on which our work is based. Finally, various systems for inductions are reviewed.

2.1 Interactive Theorem Proving

General purpose interactive theorem provers (ITPs) help users formalize proofs and check them automatically, typically in some form or other of higher-order logic. The proof often progresses by the user performing forward and backward reasoning steps. As is mentioned in Chapter 1, our research is based on *HOL Light*. It is one of the most popular ITPs and has been used to verify many famous mathematical theorems[1], including big projects such as the *Kepler Conjecture* (the *Flyspeck* project)(Hales, 2006) and along the way historical results such as the *Jordan Curve theorem*.

HOL Light comes with a simple logic core, which allows users to extend its functions while retaining soundness (Harrison, 1996a). It is also lightweight and the user interactions are implemented as Objective CAML (OCaml) (Leroy et al., 2014) functions, which allow us to modify their behaviours in a straightforward way. This makes the

[1] http://www.cs.ru.nl/~freek/100/

Symbol	Meaning
T	$\top$, Truth
F	$\bot$, Falsity
~	$\neg$, Negation
/\	$\wedge$, Conjunction
\/	$\vee$, Disjunction
==>	$\Longrightarrow$, Implication
<=>	$\Longleftrightarrow$, Logic equality
=> and <=	$\geq$ and $\leq$
?	$\exists$, Existential quantifier
!	$\forall$, Universal quantifier
[]	Empty list
CONS h t	List constructor (put element *h* in front of the list *t*)
SUC n	Successor of the natural number *n*

Table 2.1: HOL Light syntax

implementation and experimentation with HOL Light somewhat easier than in Isabelle, for instance, as we have direct access to the underlying proof engine (see Section 2.1.2)

Other ITPs such as Coq, Isabelle, HOL4 (Slind and Norrish, 2008), etc., mentioned in Chapter 1, may have differences in syntax, logics and purpose, but share many key features. Since they are not directly relevant to our work, we will focus on HOL Light and introduce its features. In what follows we refer to the formula we want to prove as a *goal* [2]. There are two ways of proving a goal: by *conversions* or creating a *goal stack* and reducing it with *tactics*.

2.1.1 HOL Light Syntax

We briefly introduce HOL Light's syntax in this section. It uses ASCII approximations for propositional connectives and quantifiers as shown in Table 2.1. The last two are the *constructors* for lists and natural numbers, which are given as recursive datatypes in HOL Light (see Section 2.3).

[2] Actually in the HOL Light manual, only the statement on the goalstack is called a *goal*, others are called terms, but we think it is easier here to just call both of these instances goals.

$$
\begin{array}{lll}
\tau \text{ (HOL Light types)} ::= & & \\
 & \mid bool \mid ind & \text{primitive types (boolean and individuals)} \\
 & \mid \alpha & \text{variable type} \\
 & \mid \tau \Rightarrow \tau & \text{function type} \\
 & term ::= & \\
 & \mid c & \text{constant} \\
 & \mid v & \text{variable (identifier)} \\
 & \mid f\ t & \text{function application} \\
 & \mid \lambda\, x.\, t & \text{function abstraction} \\
\end{array}
$$

$$theorem ::= term\ list \text{ (assumptions)} \vdash\ term \text{ (conclusion)}$$

Figure 2.1: HOL Light abstract syntax

The structure of HOL Light types, terms, and theorems is revealed in Fig 2.1. The basic component of HOL Light terms are variables and constants. Then function and predicate applications are both represented as combinations. Finally, lambda abstractions are pairs of bound variables and bodies, which are both terms. Quantified terms (e.g. $\forall x.\ P(x)$) are combinations of the quantifier, which is treated as a predicate, and lambda abstractions. Each theorem consists of a list of terms, which are assumptions and a term as the conclusion.

2.1.2 Edinburgh LCF

HOL Light is an LCF system (Gordon et al., 1979). In LCF, types (in theorem proving context), terms, formulae, and theorems are represented by types in the ML metalanguage (e.g. OCaml for HOL Light). The logical inference rules are implemented as ML functions.

The benefit of LCF systems is that all theorems have a special type (e.g. *thm* in HOL Light), which is protected (i.e. private type in OCaml) and can only be accessed by the functions from the *kernel*. New theorems must be created via the inference rules implemented as *kernel* functions, i.e. their proofs are verified by the *kernel*. This allows the user to guarantee that a theorem is proven by checking its type (*thm*), so there

is no need to store the whole proof of a theorem, which may be expensive storage-wise. HOL Light's *kernel* is small: 672 lines, which makes it practical for the user to investigate further and create trust in it.

In addition, with the underlying OCaml environment, one can compose custom procedures on top of the *kernel* functions to obtain higher-level proof procedures, which provides high extensibility and convenience.

With the features mentioned above, large corpora of proofs can be developed using the customised proof procedures known as *conversions* and *tactics*.

2.1.3 Conversions

A *conversion* transforms a goal t to an equational theorem of the form $\vdash t = t'$. We can also prove t with a series of conversions by reducing it to true. In our work, we mainly work with the following conversions:

Simplifiers These are available for rewriting in HOL Light. There are two main simplifiers: `SIMP_CONV` performs conditional rewriting while `REWRITE_CONV` does not. They use an efficient implementation of lexicographic term ordering, which can give normalisation under associative and commutative laws with appropriate rules (Harrison, 2016). However, looping or very long search time may still result, especially when too many rewrite rules are given. These methods will be used as the Simplify Heuristic in our Boyer-Moore implementation (see Section 2.5.2).

Decision procedures HOL Light provides decision procedures that can prove goals in some restricted domains. For example, `ARITH_RULE` solves linear arithmetic goals, and `TAUT` proves propositional tautologies. The latter is used as our Tautology Heuristic (see Section 2.5.2).

Automated procedures There are various automated proof procedures: *MESON* (Harrison, 1996b) and *METIS* (Hurd, 2003) try to prove goals using first-order reasoning based on techniques such as model elimination (Loveland, 1968) and resolution (see Section 2.2). They can also have lemmas as input. They are more powerful than simplifiers when doing proof search but can easily get stuck when a large number of lemmas are given. `SET_RULE` deals with goals about set theory by first simplifying them with set-theoretic definitions, followed by *MESON*,

which tries to prove any remaining goal.

2.1.4 Tactics

As mentioned, the other approach to theorem proving is to put the goal in a stack, like an agenda of problems for the user to solve. It is sometimes more natural to prove backwards, simplifying a goal or splitting it into simpler ones until they are all eliminated (proven), especially when the goal is complicated. A *tactic* takes in a goal (from the stack), processes it in the ways just mentioned, and adds the resulting subgoals to the goal stack. Meanwhile, it keeps track of the information that will be used to construct a proof of the original goal.

Many *tactics* have their corresponding *conversions*. For instance, `REWRITE_TAC` and `SIMP_TAC` correspond to `REWRITE_CONV` and `SIMP_CONV`, but instead of outputting a converted theorem, these two tactics attempt to return a simplified goal; `MESON_TAC` and `METIS_TAC` try to directly finish the current goal. Inductive proofs are usually complicated and involve the generation of subgoals, so they are usually accomplished with *tactics*.

In HOL Light, a theorem can be proven by *tactics* in two ways: Either with the `prove` function, which takes a formula and tactic as input and returns a theorem; Or with the `g` function to create a goal stack and then using the `e` function to apply *tactics* in succession. Normally, it is easier for the user to prove a goal with `g` and `e` functions interactively. After that, packaging its proof with the `prove` function and *tacticals* will make it more compact and readable, which is particularly suitable for the development of large libraries.

2.1.5 Conversionals and tacticals

When the user wants to apply several *conversions* in succession, it would be tedious to get the converted term from the theorem generated by each *conversion* i.e. get t' from $\vdash t = t'$ and then apply the next *conversion* to t' repeatedly. The *conversional THENC* can compose a series of *conversions* to produce a single *conversion* and directly apply it to t.

Similar to *conversionals*, *tacticals* help users to manipulate *tactics* and form a compact

proof. For instance, *THEN* connects *tactics* that are applied to the *goal stack* successively. A tactic may perform a goal split and result in multiple subgoals. In this case, the tactics after *THEN* will be applied to all subgoals, while *THENL* can take in a list of *tactics* and apply them to each subgoal respectively.

Note that although *tactics* are usually more useful to users during interactive proofs, it is easier to automate proofs with *conversions*. This is because they can prove theorems independently of the goalstack enabling easy composition to prove complicated lemmas in automated systems. Other conversions about induction will be introduced in Section 2.3.1 when we describe the topic in more detail.

2.1.6 Clausal Form

Many techniques used in our work depend on statements being in *clausal form*, so we briefly introduce it here. A clause is a disjunction of *literals*, i.e. $L_1 \vee L_2 \vee ... \vee L_n$. A literal L_i is an atomic formula or its negation.

For example, $m+n=0 \implies m=0 \wedge n=0$ becomes two clauses: $\neg(m+n=0) \vee m=0$ and $\neg(m+n=0) \vee n=0$ when converted to clausal form. Clauses have no quantifiers, and universally quantified variables become free variables. Existentially quantified variables are replaced by *Skolem* functions or constants (Harrison, 2009, Section 3.6) .

2.2 ATP and SMT solvers

Automatic theorem provers (ATPs) try to prove logical statements automatically. Most ATPs use *resolution* (2.1) and variants based on it, where ϕ is the *most general unifier* (*mgu*) for all Ps and P's. The *unifier* for two formulae P and P' is the substitution that has $P[\phi] \equiv P'[\phi]$ (Bundy, 1983, Section 5.1.2). Resolution in refutation systems is complete for first-order logic. Therefore, ATPs are mainly intended for first order problems.

$$\frac{(C \vee P_1 \vee ... \vee P_m) \quad (C' \vee \neg P_1' \vee ... \vee \neg P_n')}{(C \vee C')\phi} \tag{2.1}$$

In order to reduce the search space of resolution, paramodulation is designed, which treats equality as part of the logic language and is shown as (2.2), where ϕ is the *mgu*

of T and T'. *Superposition* extends paramodulation, which restricts the inference with respect to term ordering so that the proof search space is further reduced (Robinson and Wos, 1969).

$$\frac{C[T'] \quad T = S \vee D \ (or\ S = T \vee D)}{(C[S] \vee D)\phi} \tag{2.2}$$

The refutation approach tries to prove a conjecture by adding its negation to the assumptions and then checking the unsatisfiability of these formulae by deriving false (the empty clause) i.e. if these formulae are unsatisfiable then we can prove that the conjecture is the logical consequence of these lemmas and assumptions.

2.2.1 METIS

METIS is an ATP which can work as a first order proof procedure in ITPs (Hurd, 2003), and has been integrated in Isabelle, HOL4 and HOL Light. It runs different proof procedures such as model elimination (Loveland, 1968) and resolution and allows them to cooperate by sharing the clauses.

In addition, an “LCF-style” logical kernel is implemented to provide an interface that translates the proof found by these proof procedures to ITP proofs. This kernel includes five primitive inference rules that are complete for the first order logic it uses and also ensures that theorems can only be proven with these rules (or the procedure consisting of them) like the LCF kernel. Meanwhile, the ATP proof is a chain of these rules and can be easily translated by mapping the corresponding ITP procedures to these inference rules.

2.2.2 Vampire

Vampire is a powerful ATP (Kovács and Voronkov, 2013) and a winner of several CASC competitions (Sutcliffe, 2016). It is a theorem proving system that uses inference rules that include resolution, factoring, superposition, etc. It has a unique limited resource strategy that tries to adjust proof search to meet the time and memory limit, and is especially efficient for short time limits, which makes it suitable for use as an assistant for ITPs (Kovács and Voronkov, 2013). It has been used in systems like Sledgehammer (Paulson and Blanchette, 2010) and *HOL(y)Hammer* (Kaliszyk and

Urban, 2015). Redundancy elimination technique helps *Vampire* to reduce the search space, e.g. by removing clauses that are tautologies.

In addition, *Vampire* has other features such as consequence elimination, which removes the formulae that are implied by others in a set of formulae and program analysis, which automatically generates properties of loops for programs written in a subset of the programming language C.

2.2.3 E prover

E prover is a first order ATP that also uses approaches such as superposition and proof by refutation. It is highly flexible and has a number of search control strategies, which may significantly affect its performance (Schulz, 2013). Such strategies include the selection of term orderings, inference literals, and the order in which clauses are picked for processing. There are also automatic modes that analyse the problem and choose the strategies. *HOL(y)Hammer* includes a wrapper of E, *Epar*, which tries 14 of the aforementioned strategies successively with a short time slice for each one (Urban, 2013). It outperforms the old auto-mode of E 1.4 and has a performance close to Vampire 2.6 when tested with *Flyspeck* theorems (see Section 2.4) (Kaliszyk and Urban, 2014).

2.2.3.1 Z3

Satisfiability modulo theories (SMT) solvers are also automatic tools (Harrison, 2009, p.449–450). In SMT problems, constraints are given and the solvers check their satisfiability. They extend the boolean satisfiability (SAT) by adding first order theories such as equality reasoning, arithmetic, fixed-size bit-vectors, arrays, etc. Different parts of a problem containing these theories are processed separately with certain solvers and therefore are good at problems in domains such as arithmetic. Z3 developed at Microsoft Research (De Moura and Bjørner, 2008) is a state-of-the-art SMT solver, and is often used for software verification. It also deals with first order problems in TPTP syntax (Section 2.4.1) like *Vampire* and *E*. However, Z3 does not handle TPTP problems well, particularly when it is required to output the used lemmas after checking the validity of the problem (Kaliszyk and Urban, 2014), a feature which is indispensable for hammers (see Section 2.4).

There are many other ATPs, but we only cover the ones that are used in our project. In the rest of this thesis, both ATPs and SMT solvers will be referred to as ATPs for convenience. Modern ATPs are more powerful than HOL Light's *MESON* and *METIS* partly due to the compromise of these automated procedures on LCF support. For instance, an ATP can often accept hundreds of lemmas to try to prove a goal. Irrelevant lemmas can often be sent to ATPs together with the useful ones and a proof will still be found. Finally, ATPs will generate a proof, which indicates which lemmas are actually used. The number of lemmas used is often small enough that HOL Light can use them to reconstruct its own proof of the goal automatically to guarantee the proof accepted by the LCF kernel, which will be introduced in Section 2.4.1.

2.3 Recursive data type and Induction

Recursive data types are usually used in inductive theorem proving. For instance, the natural numbers can be represented and defined as $0, s(0), s(s(0)), ...$, where any numbers can be represented as the constant 0, called the *bottom object*, or by applying the *successor* function s, called the *constructor*, recursively to 0.

Inductive inference involves the use of particular logical rules to prove properties of recursive datatypes that are not otherwise provable (Bundy, 2001). The induction rule for natural numbers where $s(n)$ is the successor of n is given as (2.3):

$$\frac{P(0), \forall n.\ P(n) \implies P(s(n))}{\forall x.\ P(x)} \tag{2.3}$$

Induction is also applied to more general recursively defined data types in a process known as **structural induction**. A typical example is the list type, which will be often used in this thesis. A list is either empty ([]) or constructed by adding an element, usually called the *head* to another list, the *tail*, using the *cons* operation. The induction rule for lists can be given as (2.4):

$$\frac{P\ ([]), \forall\ head\ tail.P\ (tail) \implies P(cons\ head\ tail)}{\forall list.P\ (list)} \tag{2.4}$$

Applying the induction rule allows us to break a goal about a particular property P into new subgoals, which come from the hypotheses of the rule. They are known as "base"

```
let ADD_SYM = prove
 (`!m n. m + n = n + m`,
  INDUCT_TAC THEN ASM_REWRITE_TAC[ADD_CLAUSES]);;
```

(a) Statement and proof of ADD_SYM

```
`!n. 0 + n = n + 0`
```

(b) Subgoal from induction (base case)

```
   0 [`!n. m + n = n + m`] (induction hypothesis)
`!n. SUC m + n = n + SUC m` (induction conclusion)
```

(c) Subgoal from induction (step case)

Figure 2.2: Inductive proof for ADD_SYM

and "step" cases. The base case (e.g. $P([])$) claims that P holds for the minimal structure of the definition. The step case usually assumes that if P holds for any substructure, e.g. $P\ (tail)$, then P also holds for the whole structure, e.g. "$P\ (cons\ head\ tail)$".

2.3.1 Induction in HOL Light

In HOL Light, a recursive data type always comes with a corresponding induction rule. For natural numbers, for instance, (2.3) is in the form of the theorem `num_INDUCTION` shown as (2.5).

$$\forall P.\ P\ 0 \wedge (\forall n.\ P\ n \implies P\ (s\ n)) \implies (\forall n. P\ n) \tag{2.5}$$

An example of an inductive proof in HOL Light is that of `ADD_SYM` (the commutative property of addition for natural numbers), whose statement and proof are shown in Fig 2.2a. The proof contains two steps:

1. Perform induction on m and generate subgoals shown as Fig 2.2b and Fig 2.2c.
2. Prove the subgoals with rewriting. `ASM_REWRITE_TAC` writes the goal just like `REWRITE_TAC`, except that it also uses the assumption ($\forall n.\ m+n = n+m$), which is known as the induction hypothesis.

In the first step, the outermost variable in the goal is always matched with the induction rule, i.e. m in this example. This can be used to control the variable for induction. However, the user is responsible for choosing such variables, which can be tricky when there are two or more variables in the goal, e.g. to choose between m and n here. Users

also need to apply the induction rule corresponding to the variable type. For instance, `INDUCT_TAC` only performs induction on a natural number, and `LIST_INDUCT_TAC` is for lists. This is not always necessary in other systems like Isabelle and HOL4, where it can be automatically figured out based on the type of the variable.

Note that there could also be alternative induction rules, in addition to the one that corresponds to the type definition, even for the same data type. For instance, (2.6) is another induction rule for list type in the formalization of Hilbert's Foundations of Geometry (see Section 4.3.1) where (::) for readability denotes the *cons* operation as an infix operator. The choice of variable and rule for induction is called an *induction scheme*.

$$\forall P.\ P\ [] \wedge (\forall x.\ P\ [x] \wedge (\forall y\ ys.\ P\ (y :: ys) \implies P(x :: y :: ys)) \implies (\forall xs.\ P\ xs) \tag{2.6}$$

Aside from the difficulty in choosing a suitable *induction scheme*, users still need to prove the subgoals from induction or apply induction repeatedly. These are some of the reasons why the automation of induction is a challenging area.

2.3.2 The failure of Cut Elimination

There is a theoretical limitation on the automation of induction that arises due to the failure of Cut Elimination (Gentzen, 1969). The *cut rule* (2.7) allows us to prove the goal Δ with additional lemma ψ, which is called the *cut formula* and then to eliminate ψ by proving ψ from Γ. Since ψ can be any formula, using the *cut rule* results in infinite branches for proof search.

$$\frac{\psi, \Gamma \vdash \Delta \quad \Gamma \vdash \psi}{\Gamma \vdash \Delta} \tag{2.7}$$

The *Cut Elimination Theorem* shows that for first-order logic, the *cut rule* is redundant (Gentzen, 1969). However, Kreisel showed that *cut rule* is necessary for inductive theories (Kreisel, 1965). For this reason, sometimes extra lemmas required by an inductive proof may not be available and need to be proven by induction themselves. Lemma speculation has been used to handle this issue in systems such as λ*CLAM* and *IsaPlanner* (see Section 2.5.6).

2.4 Using ATPs to find ITP proofs

As already mentioned, in addition to various built-in proof procedures, ITPs now incorporate so-called *hammers*, which act as intermediates between powerful, external ATPs and internal proof procedures. With the help of machine learning, they allow users to reconstruct complex formal proofs within ITPs with just one click. They usually consists of four parts (Blanchette et al., 2016):

- A *lemma selection* module to filter relevant lemmas that can be used by ATPs from a large number of results available in the library. More details will be given in Section 3.3.1.
- A *translation module* that translates ITP problems to a first order syntax acceptable to ATPs, as described in Section 2.4.1.
- Links to external ATPs i.e. send goals to ATPs and get lemmas used by them if a proof is found.
- A *proof reconstruction* module that reconstructs the output of ATPs to corresponding ITP proofs.

The workflow of a typical hammer is shown as Fig 2.3. After (hundreds of) lemmas relevant to the goal are selected, they are translated, together with the goal, to an ATP problem (e.g. using the TPTP syntax, described in Section 2.4.1). ATPs are then called in parallel to find a proof. If a proof is found, the lemmas that they used are sent back to the ITP, which then attempts to reconstruct the proof internally using its own proof procedures.

Sledgehammer is the original tool that started the whole effort: it is integrated into the Isabelle proof assistant and carries out lemma selection using a combination of relevance filtering *MePo* (Meng and Paulson, 2009) and machine learning methods (Kühlwein et al., 2013), including *Naive Bayes* (see Section 3.3.1).

Heavily inspired by Sledgehammer, HOL Light now includes *HOL(y)Hammer*, which also uses machine learning for lemma selection. In our work, we incorporate elements from its latest released version[3], such as its *feature extraction* algorithm (see Section 3.3.2). In Kaliszyk and Urban's experiment with 14185 *Flyspeck* theorems, 39% of the theorems were proven with a combination of 14 different machine learn-

[3] http://cl-informatik.uibk.ac.at/software/hh/hh-0.13.tgz

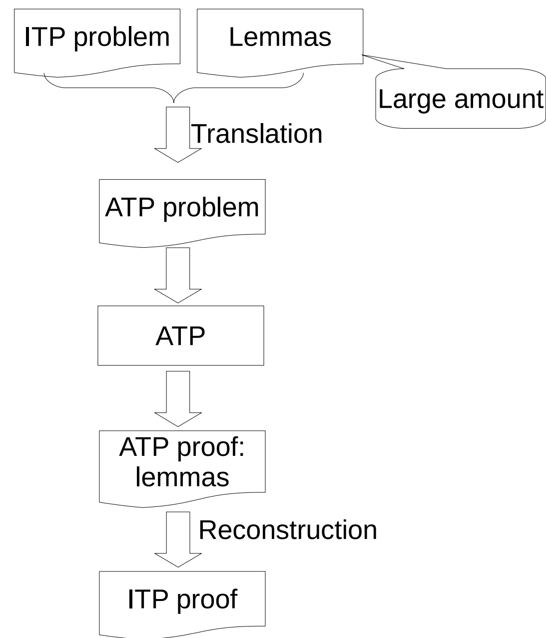


Figure 2.3: The workflow of a hammer

ing methods (some are the same methods with different parameters) and three ATPs (Vampire, Z3 and a modified version of E) within 30 seconds (Kaliszyk and Urban, 2014).

2.4.1 Translation between ITPs and ATPs

In order to use ATPs to solve ITP problems, translations are required. Usually, several ATPs may be used to generate the proof and they each have their own particular input format, so a unified format can avoid the need for multiple translators. In *HOL(y)Hammer*, HOL Light formulae are translated to the "Thousands of Problems for Theorem Provers" (TPTP) format (Sutcliffe, 2009), which is supported by many ATPs. The TPTP syntax is shown as the BNF in Fig 2.4 [4], where only the entries used in the translation are given.

For instance, the formula $\forall x.\ x < x+1$ translated to TPTP syntax by *HOL(y)Hammer* is:

```
![X]: p(s(bool,l_(s(num,X),
  s(num,p_(s(num,X),
            s(num,numeral(s(num,bit1(s(num,u_0))))))))))
```

[4] http://tptp.cs.miami.edu/TPTP/SyntaxBNF.html

```
%----FOF formulae.
<fof_formula>           ::= <fof_logic_formula> | <fof_sequent>
<fof_logic_formula>     ::= <fof_binary_formula> | <fof_unary_formula> |
                            <fof_unitary_formula>
<fof_binary_formula>    ::= <fof_binary_nonassoc> | <fof_binary_assoc>
<fof_binary_nonassoc>   ::= <fof_unit_formula> <nonassoc_connective>
                            <fof_unit_formula>
<fof_binary_assoc>      ::= <fof_or_formula> | <fof_and_formula>
<fof_or_formula>        ::= <fof_unit_formula> <vline> <fof_unit_formula> |
                            <fof_or_formula> <vline> <fof_unit_formula>
<fof_and_formula>       ::= <fof_unit_formula> & <fof_unit_formula> |
                            <fof_and_formula> & <fof_unit_formula>
<fof_unary_formula>     ::= <unary_connective> <fof_unit_formula> |
                            <fof_infix_unary>
%----Atomic formula
<fof_unit_formula>      ::= <fof_unitary_formula> | <fof_unary_formula>
<fof_unitary_formula>   ::= <fof_quantified_formula> | <fof_atomic_formula> |
                            (<fof_logic_formula>)
<fof_quantified_formula> ::= <fof_quantifier> [<fof_variable_list>] :
                            <fof_unit_formula>
<fof_variable_list>     ::= <variable> | <variable>,<fof_variable_list>
<fof_atomic_formula>    ::= <fof_plain_atomic_formula> |
                            <fof_defined_atomic_formula> |
                            <fof_system_atomic_formula>
<fof_plain_atomic_formula> ::= <fof_plain_term>
<fof_plain_atomic_formula> :== <proposition> | <predicate>(<fof_arguments>)
%----FOF terms.
<fof_plain_term>        ::= <constant> | <functor>(<fof_arguments>)
%----Arguments recurse back to terms (this is the FOF world here)
<fof_arguments>         ::= <fof_term> | <fof_term>,<fof_arguments>
<fof_term>              ::= <fof_function_term> | <variable>
<fof_function_term>     ::= <fof_plain_term> | <fof_defined_term> |
                            <fof_system_term>
%----Connectives
<fof_quantifier>        ::= ! | ?
<nonassoc_connective>   ::= <=> | => | <= | <~> | ~<vline> | ~&
```

Figure 2.4: TPTP abstract syntax

As can be seen TPTP uses a prefix notation so, for example, the variable ‘x:num‘ and the predicate $a < b$ are represented as “s(num,X)” and “p(s(bool,l_(s(num,a),s(num,b))))” respectively. Terms are represented recursively taking type information as arguments (i.e. num), see `<fof_arguments>` in Fig 2.4.

In order to let ATPs support some aspects of higher order logic, a suitable translation to first order logic is necessary before translation to TPTP syntax. For instance, a translation for EQ_EXT from HOL Light where functions are universally quantified over is (Kaliszyk and Urban, 2014):

```
EQ_EXT: !f g. (!x. f x = g x) ==> f = g
EQ_EXT(Translated):!f g. (!x. happ f x = happ g x) ==> f = g
```

In the original theorem, f and g are polymorphic typed functions. After the translation, they become arguments and the only function is the newly introduced constant $happ$. The theorem is then in first order, and is ready to be translated to TPTP syntax afterwards. The changes in syntax (Fig 2.1 vs Fig 2.4) and other steps such as the aforementioned conversion to first order logic make it challenging to translate a HOL Light problem to an ATP acceptable version.

Note that all the lemmas are labelled (with their names) before they are sent to ATPs. As long as the ATPs find a proof, these labels will be sent back for the reconstruction of ITP proofs. This means that only lemma names are available for reconstruction. Therefore the reconstruction is usually a group of attempts with the methods in ITPs, such as the *conversions* and *tactics* for simplification in HOL Light. For instance, *HOL(y)Hammer* uses REWRITE_TAC, SIMP_TAC, and MESON_TAC [5] for reconstruction. METIS_TAC was added later when building our own system. We remark here that none of these *tactics* can guarantee the success of reconstruction. The main reason is that *tactics* are still slower than ATPs and may not find the proof within the time constraints usually imposed during interactive theorem proving.

2.5 Systems for induction

Originally the Boyer-Moore model covered some of the key components of an automated theorem prover for inductive proofs (Boyer and Moore, 1979). It focused on

[5]The *HOL(y)Hammer* version we acquired was for the old version of HOL Light, where METIS was not available.

recursive data types and functions. Many of its techniques are still in use for research on automated inductive proofs, e.g. *ACL2*, which was mentioned in Chapter 1.

Our work uses a Boyer-Moore system implemented in HOL-Light (Papapanagiotou and Fleuriot, 2018), which is itself a reimplementation of Boulton's work in HOL90 (Boulton, 1992). An important advantage of this implementation in HOL Light, particularly in comparison with more sophisticated evolutions of the Boyer-Moore approach on ACL2, is that it is lightweight with simple structures and easy, direct access to the inner workings. This makes it relatively easy to manipulate and adjust the Boyer-Moore waterfalls and heuristics, and analyze the effects of machine learning thoroughly.

2.5.1 The Boyer-Moore Model

The Boyer-Moore Model revolves around the notion of a *waterfall*, as shown in Fig. 2.5. In this, conjectures (or proof goals) are poured at the top and then fall through a series of procedures, called *heuristics*. Each heuristic in the waterfall tries to either prove or simplify the goal and has one of the following outcomes:

- The goal may be proven, in which case it "evaporates".
- The goal may be simplified or transformed, and sometimes split into small subgoals. In this case, the transformed goal (or subgoals) is poured from the top of waterfall again.
- The goal or subgoal is disproved, where the whole proof fails.
- The goal cannot be processed. For instance, a heuristic tries to simplify the goal, but there are no relevant rules to apply. In this case, the goal remains unchanged, and the heuristic has "failed".

Induction is applied when all heuristics have failed, and the goals have trickled down to the pool at the bottom of the waterfall. Applying the appropriate induction rule results in new sub-goals, i.e. the base and step cases of the induction rule, which are then poured over new waterfalls again. This process is repeated recursively until all subgoals are proven, as shown in Fig. 2.6, in which case a proof of the original goal is reconstructed and proven. If any subgoal is determined to be unprovable, the original goal cannot be proven in this model. Although very rare, the induction procedure may

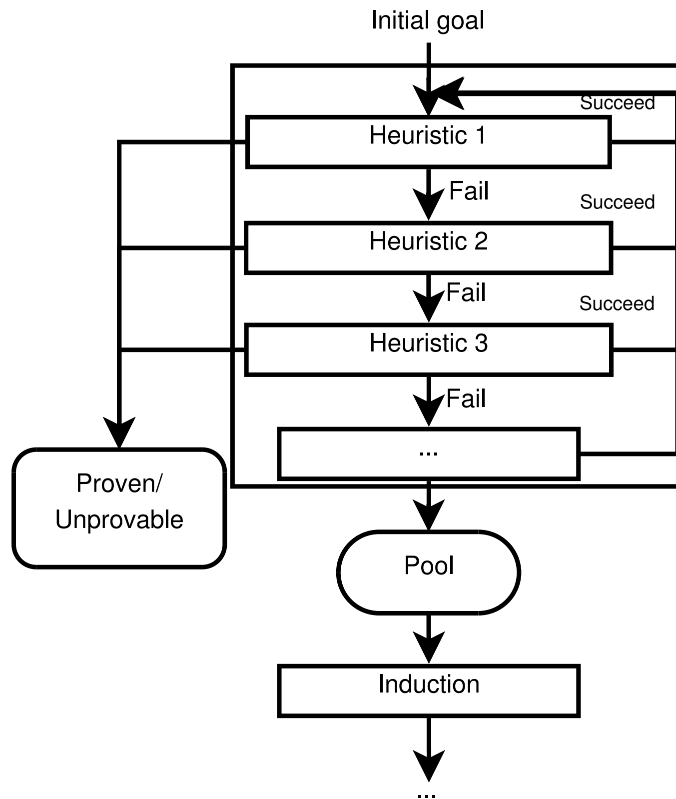


Figure 2.5: Diagram of the Waterfall Model

be unable to proceed, e.g. fail to find a variable for induction, which also makes the model fail to prove the goal.

In some cases, this model may not terminate. Firstly, looping can result from endlessly pouring the subgoals on top of the waterfall , although some of these situations have been solved (Papapanagiotou and Fleuriot, 2018). Secondly, the process may get stuck at some heuristic, e.g. those that involve simplifiers.

2.5.2 The heuristics

All heuristics have the same input and output structure, so the organisation of waterfalls can easily be adjusted in a modular way. We now introduce the heuristics in the Boyer-Moore Model together with some of the additional heuristics from the HOL Light implementation (Papapanagiotou and Fleuriot, 2018) (Papapanagiotou, 2007).

The Clausal Form heuristic This transforms the goal to Clausal Normal Form (CNF) (see Section 2.1.6), which other Boyer-Moore heuristics take advantage of.

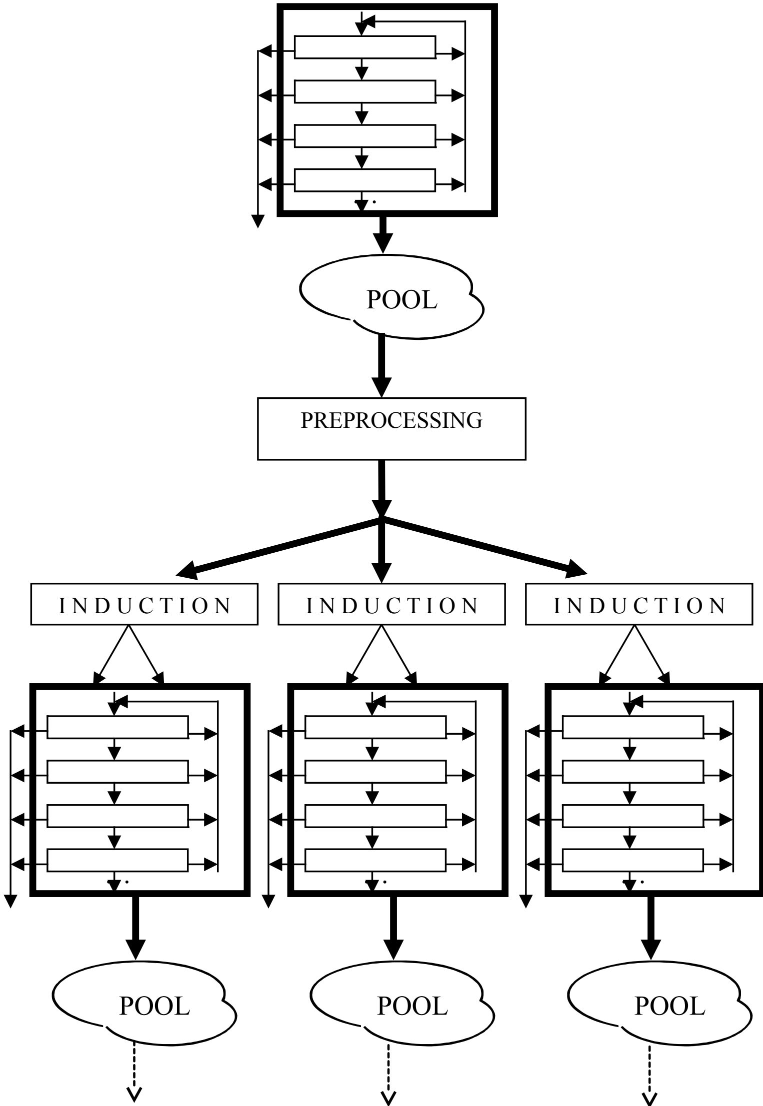


Figure 2.6: The Boyer-Moore Model

The Simplify heuristic This applies rewriting to the goal in order to simplify or prove it using function definitions and rewrite rules. This is implemented by the conditional rewriting function `SIMP_CONV` in HOL Light (see Section 2.1.3). Note that termination is not guaranteed and depends on the selection of rewrite rules.

The Substitution heuristic This simplifies the goal in clausal form by eliminating negations of equalities between a variable and a term. For instance, suppose in the clause 2.8, x does not occur in L_1, L_2, L_3, or t. This heuristic simplifies it to (2.9) by substituting t for x in $P(x)$ and F can then be removed.

$$L_1 \vee \neg(x = t) \vee L_2 \vee P(x) \vee L_3 \tag{2.8}$$

$$L_1 \vee F \vee L_2 \vee P(t) \vee L_3 \tag{2.9}$$

The Equality heuristic This is similar to the **Substitution heuristic**, but instead of looking for equality between a variable and a term, the term that is **not** an *explicit value template* is eliminated. An explicit value template is a non-variable term composed of only the bottom objects or constructor applications (to bottom objects or variables) e.g. 0 and $s(0)$. For instance, $n \times 0$ is not an explicit value template, so $\neg(n \times 0 = 0) \vee (n \times 0) + 0 = 0$ is simplified to $F \vee 0 + 0 = 0$ and finally to $0 + 0 = 0$.

The Generalization heuristic This attempts to propose a stronger goal than the current one, which may be easier to prove. One of the approaches is to eliminate minimal common subterm. For instance, in $(m \times n) + n = n + (m \times n)$, $m \times n$ occurs on both sides of the equation, so the equation can be generalised to $n' + n = n + n'$, which implies the original goal, does not affect its validity, but is simpler. Another approach is to generalise variables apart. For example, (2.10) (i.e. the length of the list from appending two same lists x is equal to summing up the lengths of two lists x) requires generalisation, because x appears in both recursion and non-recursion position, which interferes with *recursion analysis* (see Section 2.5.4). It can be generalised to (2.11) where x is renamed so that the variable for induction can be chosen. However, there is a risk that such a process over-generalises the goal and makes it unprovable.

$$LENGTH\ (APPEND\ x\ x) = LENGTH\ x + LENGTH\ x \tag{2.10}$$

$$LENGTH\ (APPEND\ x'\ x) = LENGTH\ x' + LENGTH\ x \tag{2.11}$$

The Irrelevance heuristic This tries to eliminate irrelevant literals from the goal in clausal form. For instance, with the generalization heuristic, (2.12) is generalised to (2.13) where $p = []$ is no longer relevant because it has no common variables with other literals. In this case, it can be replaced by F, i.e. removed from the clause (see (2.14)). Unfortunately, this heuristic is also unsafe and may remove literals that are actually relevant.

$$\begin{aligned} p = [] \vee REVERSE\ (APPEND\ (REVERSE\ p)\ [a]) \\ = CONS\ a\ (REVERSE\ (REVERSE\ p)) \end{aligned} \tag{2.12}$$

$$p = [] \vee REVERSE\ (APPEND\ l\ [a]) = CONS\ a\ (REVERSE\ l) \tag{2.13}$$

$$\cancel{F \vee} REVERSE\ (APPEND\ l\ [a]) = CONS\ a\ (REVERSE\ l) \tag{2.14}$$

The Tautology heuristic This is a propositional tautology checker which is added in the HOL Light implementation. Note that the atomic formulae do not have to be propositional, i.e. $a = 0 \vee \neg(a = 0)$ can also be proven.

The Setify heuristic This eliminates redundant literals, which often result from applying the Simplify heuristic e.g. $A \vee B \vee A$ is simplified to $A \vee B$. This is added in the HOL Light implementation because `SIMP_CONV` does not perform such elimination.

The MESON heuristic The HOL Light implementation of Boyer Moore has `MESON` as a heuristic. However, note that it can probably run for a long time without finding a proof when given rewrite rules and definitions.

As mentioned, many heuristics rely on having formulae in clausal form. Therefore, the Boyer-Moore Model takes *quantifier-free* formula as input and converts it to CNF before trying other heuristics.

2.5.3 The Shell Principle

The *Shell Principle* was originally used to introduce new recursive datatypes in the Boyer-Moore Model (Boyer and Moore, 1979). Although HOL Light supports recursive data types, the shell properties are stored as theorems separately, i.e. not bound directly to the data types as part of the Boyer-Moore tool. The Shell Principle is still needed in the HOL Light implementation of the Boyer Moore Model to give access to important properties of data types that are needed to run the waterfall in a systematic

Name	"num"
Type variable(s)	[](none)
Bottom Object	0
Constructor	SUC
Induction rule	$\vdash \forall P.\ P\ 0 \wedge (\forall n.\ P\ n \implies P\ (SUC\ n)) \implies (\forall n.\ P\ n)$

Table 2.2: Shell for the natural number type

Name	"list"
Type variable(s)	[A]
Bottom Object	[]
Constructor	CONS
Induction rule	$\vdash \forall P.\ P\ [] \wedge (\forall a_0\ a_1.\ P\ a_1 \implies P\ (CONS\ a_0\ a_1)) \implies (\forall x.\ P\ x)$

Table 2.3: Shell for the list type

way. Some important properties that are frequently used are shown in Table 2.2 and Table 2.3 for natural numbers and lists respectively.

2.5.4 Induction procedure

Boyer-Moore uses an additional procedure at the pool of the waterfall to:

1. Choose the appropriate variable for induction.
2. Apply induction rule based on the type of the selected variable and generate subgoals.

The choice of induction variable is based on the definitions of recursive functions and is known as *recursion analysis* (Boyer and Moore, 1979). This prefers the variable that occurs in a recursion argument position for each recursive function. For instance, the definition of addition is shown as (2.15), and its first argument is recursive. As shown in the proof in Fig 2.2a, the first argument is often used for induction.

$$(\forall n.\ 0+n=n) \wedge (\forall m\ n.\ s(m)+n=s(m+n)) \tag{2.15}$$

In order to perform *recursion analysis*, it is clear that recursive function definitions are required. Note that in the HOL Light implementation, this heuristic only supports

primitive recursive definitions (Boulton, 1992), which means there can be only one recursion argument position in a function definition, like (2.15). The induction rule is provided by the shell, so it is fixed for each data type.

2.5.4.1 Destructor-style and constructor-style definition

The definition shown in (2.15) is called *constructor-style*, where constructor is used in the recursion argument position. In some Boyer-Moore implementations such as ACL2, *destructor-style* definition is used. For instance, the definition of addition can be (2.16), where $p(n)$ is the predecessor function defined as (2.17). Most recursive function definitions in HOL Light are *constructor-style*.

$$x + y = IF\ x = 0\ THEN\ y\ ELSE\ s(p(x) + y) \tag{2.16}$$

$$p(n) = \begin{cases} 0, & n = 0 \\ m, & n = s(m) \end{cases} \tag{2.17}$$

2.5.5 Extensibility

Based on the above, the system configuration can be tailored to deal with different problems. The most common customizations are the following (Papapanagiotou and Fleuriot, 2018):

- As shells are an essential part of the Boyer-Moore model, when we want the system to deal with a particular datatype, the corresponding shell must be prepared, with all the properties provided, e.g. like in Table 2.3.
- The rewrite rules for the Simplify heuristic can be chosen by the user to improve its effectiveness towards proving the subgoals.
- The order and combination of heuristics can also be adjusted for different situations. For instance, some heuristics are unsafe and may render the goal more complicated or result in an infinite loop. Apart from this, new heuristics can be made and added, which may help the system to deal with certain kinds of problems.

2.5.6 Other systems for induction

There are some other systems for the automation of inductive proofs.

CLAM (and the higher-order version λCLAM) are tools for automated theorem proving and particularly in inductive theorem proving. They both use the Proof Planning (Bundy, 1988, 1996) technique, which provides a global control strategy proof search. One of its main technique is rippling (Bundy et al., 2005), which guides the rewriting process. Rippling is based on the observation that the parts of the induction conclusion which differ from the induction hypothesis (*wave-fronts*) can *ripple-out* of the induction conclusion so that the remaining part is identical to the induction hypothesis. Sometimes, additional lemmas need to be introduced, with methods like lemma speculations and generalisations.

IsaPlanner (Dixon and Fleuriot, 2003) is a generic framework for proof planning in Isabelle. It allows proof planning to take advantages of the powerful tactics in Isabelle such as the simplifier. Dynamic rippling (Smaill and Green, 1996) is used in *IsaPlanner*, so it is suitable for higher order logic (Dixon and Fleuriot, 2004). The combination with induction and lemma speculation improves its ability to automate proofs in Isabelle. It can also generate Isar proof scripts.

HipSpec (Claessen et al., 2012), is a system that derives and proves properties about the functional programming language Haskell automatically. It uses a bottom-up approach to try and generate theorems from the source file (the definitions in a program) and use them to prove more complicated properties about the program. As a theorem prover, when tested against the *IsaPlanner benchmark* (see Section 4.3.3), it also has a competitive performance (80 of the 85 theorems proven) compared with ACL2 Sedan (Chamarthi et al., 2011) (74 proven), and Zeno (Sonnex et al., 2012) (82 proven), which is also an automated proof system for Haskell.

CVC4 with induction Reynolds and Kuncak developed an approach to extend SMT solvers with induction support and implemented it in CVC4 (Reynolds and Kuncak, 2015). It performs *inductive strengthening* on the conjecture about recursive datatypes. It also has a *subgoal generation module* which generates and proves relevant subgoals which are required to prove the conjecture. When tested with 933 benchmarks (331 theorems in three encodings) constructed from the benchmarks for *IsaPlanner*, *Clam*, *Hipspec*, etc., it improved the success rate of CVC4

from 17% to 67% after adding induction support and up to 78% with the subgoal generation module.

Superposition with induction A system that supports structural induction as an extension to superposition-based provers also exists (Cruanes, 2017). Strategies such as heuristics for selecting induction variable and generalisation are used, which allow the prover to do induction, while retaining the performance in first-order reasoning. When testing with the *Tons of Inductive Problems* (TIP) benchmark (Claessen et al., 2015) (484 theorems) and given 30 seconds for each problem, it had a similar performance to CVC4 with induction support (Reynolds and Kuncak, 2015) (139 vs 138 proven), but a little weaker than the special mode of CVC4 (160 proven), which uses a bottom-up approach similar to HipSpec.

ACL2 A Computational Logic for Applicative Common Lisp (ACL2) is both a logic and programming language and a theorem prover made in Common LISP (Kaufmann et al., 2000). It is the latest system based on the Boyer-Moore model with a number of improvements and has been successfully applied in industry e.g. the verification of microcode programs for complex arithmetic processors and the kernel of the floating-point division operation on the AMD microprocessor (Brock et al., 1996). It allows users to create program functions, and prove properties on them in untyped first-order logic where all variables are indicated as universally quantified.

ACL2 proves theorems with a high degree of automation. Meanwhile, it allows users to advise the theorem proving process with "hints", which are intermediate lemmas like the *cut formula* (see Section 2.3.2) to help ACL2 resolve failures. As expected, it is sometimes challenging for the user to provide such hints.

Compared with the HOL Light implementation that we use in our investigation, ACL2 is more powerful and has more support for special proof features e.g. its induction procedure allows non-primitive recursive function definitions. However, the HOL Light implementation allows higher order logic, is easier for modification (see Section 2.5), and aims to improve automated inductive proof support within in the general settings of a proof assistant.

2.6 Conclusion

In this chapter, we gave the context for our work i.e. interactive theorem proving. Theorem provers i.e. ITPs and ATPs as well as other background knowledge involved in our work were introduced. After that, techniques about proof by induction and its automation were also reviewed. The Boyer-Moore model for automated inductive theorem proving was introduced. We also reviewed other automated systems for inductive theorem proving. In the next chapter, some machine learning techniques and their applications in theorem proving are introduced.

Chapter 3

Machine learning for theorem proving

Since our work is about combining machine learning techniques with theorem proving, in this chapter, we first introduce relevant machine learning methods (Section 3.1) and then discuss the lemma selection problem (Section 3.2), followed by the techniques used for lemma selection (Section 3.3).

3.1 Machine learning

Machine learning is about data analysis. A simplified example of the type of data that is used for training is shown in Table 3.1. Each row represents a piece of data called an instance, which is usually collected from a single test or item. The two key elements of the data are:

Feature In each instance from the data set, the features $F_1, F_2, ..., F_n$ form a vector of values which represent the properties of the object. For instance, in Table 3.1, the features are represented as a vector of binary digits.

Label In each instance there is sometimes also one or more outputs or response variables, called labels. In particular, when a classification task is used, as in this thesis, they are also called *class labels*. Table 3.1 is an example that contains two classes (0 and 1), which yields a binary classification problem.

There are two kinds of tasks in machine learning: *supervised* and *unsupervised* learning. A supervised learning task tries to learn the "mapping" from *features* to *labels*. This procedure, called the *training process*, is "supervised" by the training set, where

Features					Label
F_1	F_2	F_3	...	F_n	
0	1	1	...	0	1
0	0	1	...	0	1
0	1	1	...	0	0
...	...	...	...	...	...
1	1	1	...	0	0

Table 3.1: An example of training data

each instance contains such a mapping. After the training, *classifiers* are obtained and can be used to *predict* the *label* for new item based on its *features*.

3.1.1 Supervised learning

In what follows we give a brief overview of supervised learning algorithms, which are the main approaches used for theorem proving:

Naive Bayes is a probabilistic learning algorithm based on Bayes' theorem with a strong (naive) independence assumption between the features (Zhang, 2004), i.e. in our case that the value of each feature is independent of any other feature for the same goal. It is a generative approach which models the probability based on Bayes' theorem, and the classification is done by choosing the class with a higher probability. The theorem for a two-class case (A and B) is shown as (3.1). "$P(Y), Y \in \{A, B\}$" is called the *prior*, i.e. the probability of label Y to appear in an instance, which is computed as $P(Y) = n/m$, where m is the total number of instances, and n is the number with label Y. $P(f|Y), Y \in \{A, B\}$ is the *likelihood*, which is computed as (3.2), where $Y \in \{A, B\}$. Data in Table 3.1 can be used as an example, if we have f_i fit to F_i by having $f_i \in \{0, 1\}$ and $A, B = 1, 0$. $P(f_i|Y)$ is computed as (3.3) where m is the total number of instances labelled Y, and n is the number of the ones that also have feature $f_i = 1$. Note that a more efficient form, in (3.4), with log likelihood, is more commonly used, and a minimal value σ is used to avoid extreme, i.e. $P(f_i|Y) = 0$.

$$P(A|f) = \frac{P(f|A)P(A)}{P(f|A)P(A) + P(f|B)P(B)} \tag{3.1}$$

$$P(f|Y) = \prod_{i=1}^{n} P(f_i|Y) \tag{3.2}$$

$$P(f_i|Y) = (n/m)^{f_i}(1-n/m)^{1-f_i} \tag{3.3}$$

$$\log P(f|Y) = \sum_{P(f_i|Y)\neq 0} \ln P(f_i|Y) + \sum_{P(f_i|Y)=0} \sigma \tag{3.4}$$

Naive Bayes has advantages when it comes to training: the training process, explicitly shown in (3.4), can be quickly computed (compared with other algorithms), and when a new instance is added to the data, only small changes need to be done to update $P(Y)$ and $P(f_i|Y)$, instead of recalculating with all the previous data.

***k*-Nearest Neighbours** (k-NN) algorithm considers features as vectors in a high-dimensional space, and computes the Euclidean distances between each of them (Cover and Hart, 1967). When predicting for a new item, based on the distance, k nearest instances (neighbours) can be picked out. Different strategies can then be applied: use the mode (the value that appears most often) of the neighbours' class labels as the label for the new item or use the weighted average to estimate the label. For instance, in a two-class problem, 1 and 0 are assigned to the neighbours labelled A and B respectively. A weighted average is assigned to the new item, where the weights are usually the distance to the neighbour and can also be customised by the user. The label for the new item will be A if the average value is closer to 1, but B if closer to 0.

k-NN requires no training by default, so the data can be directly used for prediction. However, all the training data have to be kept, which can be a big space consumption and, as their size grows, the prediction will extremely slow down. Tree-based data structures can accelerate the search process, and therefore the prediction. However, this will require extra training time as a result (Bishop, 2006, p.127). Another issue is the choice of k, which is very important, but may strongly depend on the training data.

Regularized Least Squares with Kernel methods Regularized Least Squares (RLS) classification also considers features as high-dimensional vectors, and try to find a model f that fits all the vectors by minimising the error function like (3.5)(Rifkin et al., 2003). It contains the RLS part, i.e. $(y_i - f(x_i))^2$, where x_i and y_i are the *feature* and *label* we just mentioned. The *regularizer* $\|f\|_k^2$ helps reduce *over-fitting*, which means that the classifier works better on known data

Features				Labels				
F_1	F_2	F_3	...	L_1	L_2	L_3	...	L_m
0	1	1	...	1	0	0	...	0
0	0	1	...	1	1	0	...	0
0	1	1	...	0	0	1	...	1
...	...	...	...	...	...	...	...	...
1	1	1	...	0	1	1	...	1

Table 3.2: An example of training data for ranking

while worse on new data. Similar to k-NN, the value $f(x)$ can be used for prediction, based on its difference from 1 and 0.

$$\min_f \sum_{i=1}^{n} (y_i - f(x_i))^2 + \lambda \|f\|_k^2 \tag{3.5}$$

The minimizer of (3.5) has the form (3.6) (Rifkin et al., 2003), where $K(x,x')$ is the kernel. Kernel methods allow the user to change the learning model by using a different kernel, while still use the minimizer (3.6) (Bishop, 2006, p.292).

$$f^*(x) = \sum_{i=1}^{n} c_i K(x, x_i) \tag{3.6}$$

3.1.1.1 Learning to rank

Sometimes, a ranking is preferred to absolute values. For instance, if we want to predict the **most** relevant lemmas rather than only the relevant ones. In this case, we want to have a *rank* of labels. To be more precise, the *label ranking* task is to find a total rank of the multiple labels $L_1, L_2, ...$, instead of their values.

A common solution is to rank by the output values. For instance, all the machine learning algorithms in this section return either $P(Y_i|f)$, which is a probability of having label Y_i with feature f, or $f_i(X)$, which is a value computed from feature vector X. Here Y_i and $f_i(X)$ are the output values for L_i. In both cases, labels can be ranked by comparing their values. Assume that there are m labels for ranking, e.g. the data in Table 3.2. The training procedure works as:

1. For each label L_i, train a classifier (c_i) with all features (i.e. all the columns under *Features*) and the labels from one column L_i. Therefore, m classifiers ($c_1, c_2, ...$)

are trained separately, with all the features shared but label information independent.

2. When predicting for a new item with feature f_{new}, each classifier c_i makes prediction with f_{new} independently and get m output values: $v_i = c_i(f_{new})$.

3. The label L_i has a rank based on its v_i.

3.1.2 Unsupervised learning

In unsupervised learning, no *label* is given, and the task is to discover "interesting structure" in the data (Murphy, 2012). Unsupervised learning is occasionally used in the theorem proving field for clustering via the *k-Means* algorithm.

K-Means clustering (Arthur and Vassilvitskii, 2007) considers features as vectors in high-dimensional space (like some supervised learning methods we just mentioned). The process of clustering (shown as (3.7)) is to find k "*centres*" (represented as the vector $\mu_{\mathbf{i}}$) in the space, so that each vector point ($\mathbf{x}$) is assigned to the cluster that contains its nearest *centre* (i.e. S_i), and the sum of all (squared) distances between the *centres* and the vectors is minimised. The k value can be customised by the user.

$$\min_S \sum_{i=1}^{k} \sum_{x \in S_i} ||\mathbf{x} - \mu_{\mathbf{i}}||^2 \tag{3.7}$$

3.1.3 Online learning

If the machine learning approach is performed with a *batch* of data, it is said to be *offline*. However, when the data come as a *stream*, i.e. as separate data points, it is better if we can update the *classifiers* right after new data come, so that the *classifiers* are always available with the updated information. In this case, we need to perform *online learning* (Murphy, 2012, Section 8.5). Another situation where we need *online learning* is when the batch of training data is too large to hold in main memory and perform an *offline* training, they need to be split and learnt as a stream. Thanks to the mechanism of *Naive Bayes* and *k-NN*, *online learning* is supported for them and can thus be applied to lemma selection in theorem proving.

3.2 Lemma selection

Lemma selection is an important component of hammers as they provide the pre-proved results that may be used by the external ATPs to lead to a proof (see Section 2.4).

Target of lemma selection. For a given goal and a group of candidate lemmas, rank all the lemmas based on their likelihood of being used to prove the goal. (For machine learning approaches) Previous goals and their lemmas can be provided, so that the relevance between the lemmas and the goals can be learnt.

3.2.1 Non-machine learning methods for lemma selection

Before machine learning approaches were applied in lemma selection, there was some work already on lemma selection.

MePo is a filter developed by Meng and Paulson, used in Sledgehammer, which filtering the lemmas based on their relevance (Meng and Paulson, 2009). This approach computes the *relevance distance* between two clauses, based on a group of relevant symbols, where symbol could refer to a constant, function or predicate. The use of *MePo* results in a relatively high success proof rate and can often set users free from hand-selection of suitable lemmas.

The key idea is to compute the relevance of a lemma to a goal based on the proportion of symbols in both the lemma and the goal to the total number of symbols in the lemma. With 7000 known theorems as candidates, it had an increase in proof success rate from 10 to 20% after lemma selection on their problem sets (Meng and Paulson, 2009). Other ideas that are paid attention to in this approach included:

1. The iterative selection of lemmas, where in each round, the relevance between the new lemma and the selected ones is considered.
2. The rarity of symbols, which assigns higher weights to uncommon symbols.

There are other non-machine learning methods which we briefly mention. *SInE* (SUMO Inference Engine) is a symbol based lemma selection algorithm for ATPs, which has been implemented in Vampire (Hoder and Voronkov, 2011). *SRASS* (Semantic Relevance Axiom Selection System) is also a lemma selection method for ATP guidance

which is based on syntactic relevance (Sutcliffe and Puzis, 2007). *APRILS* (Automated Prophesier of Relevance Incorporating Latent Semantics) is a method that applies latent semantic analysis (LSA), and is used in an ATP meta-system called Divvy (Roederer et al., 2009). These methods are more intended for the selection within ATPs, e.g. the order of applying lemmas in resolution, so are not directly related to our work and will not be described further here.

3.3 Machine learning for theorem proving

In this section, we first introduce the application of machine learning algorithms in theorem proving. After that we explain how features and dependencies are generated from HOL Light proofs, and then represented as the training data in Section 3.1. Finally, the related work about combining machine learning for inductive theorem proving is introduced. In this section, we are using a similar way to define the terminology used in lemma selection (e.g. the **feature** and **dependency matrix**) as in the work by (Alama et al., 2014).

3.3.1 Machine learning algorithms used for lemma selection

In this section, the application of the machine learning algorithms mentioned in Section 3.1.1 to lemma selection will be reviewed.

Naive Bayes is most commonly used in lemma selection. In *HOL(y)Hammer* (Kaliszyk and Urban, 2014), the implementation of Naive Bayes used for lemma selection is from SNoW (Sparse Network of Winnows) (Carlson et al., 1999). In Sledgehammer, *Mash* initially used an implementation in Python and later on in Standard ML (Kühlwein et al., 2013). These implementations are *sparse naive Bayes*, where only the features that appear in the current goal being learnt are considered (i.e. for (3.3) $P(f_i|Y) = (n/m)^{f_i}$), so they are faster than standard Naive Bayes. This method was tested in both Sledgehammer and *HOL(y)Hammer*. In the experiment for *HOL(y)Hammer*, Naive Bayes was able to prove up to 29.4% *Flyspeck* theorems with different settings (e.g. the choice of features) (Kaliszyk and Urban, 2014). In Sledgehammer, the experiment by Kühlwein et al. was to compare the results from reproving the theorems in three formalizations in the Isabelle distribution and the *Archive of Formal Proofs* between

using the advice from *MePo* and *Mash*. *Mash* solved many more goals than *MePo* alone (when comparing the peak value of the proof rate curve[1], it was 44.8% vs. 38.2%) (Kühlwein et al., 2013).

Mesh is a combination of *Mash* and *MePo* in Sledgehammer (Kühlwein et al., 2013). It uses the weighted average of the outputs (the relevance of the lemma i.e. the value for ranking) from *MePo* and *Mash*. When reproving the theorems in the *Judgment Day benchmarks* (Böhme and Nipkow, 2010), Mesh solved more goals than *MePo* and *Mash* alone (69.8% vs. 65.6% and 63.0%).

k-NN was explored for lemma selection. When introducing inverse document frequency (IDF) as weights, *k*-NN had improvement (from 39.0% to 45.5% Flyspeck problems solved) overall coverage by the combinations of different machine learning methods (Kaliszyk and Urban, 2013).

Kernel methods were used in the MOR (multi-output ranking) with Gaussian kernel (Kühlwein et al., 2011). In MOR-CG (multi-output ranking conjugate gradient) (Kühlwein et al., 2012), a linear kernel was applied and conjugate-gradient was used for optimisation, which accelerated the algorithm. When testing on the MPTP2078 benchmark [2], which has 2078 theorems, it increased the proven theorems from 788 (by *Naive Bayes*) to 824 (Alama et al., 2014). We refer the interested reader to the cited papers for more technical details.

3.3.2 Feature extraction

Statements of theorems and goals are characterised by features during machine learning. In this section, various approaches to generating features for different systems are reviewed.

[1]It is difficult to decide the best number of lemmas that ATPs require to find a proof. ATPs usually run faster with fewer lemmas, while more lemmas need to be selected so that all the ones required to prove the goal may be included. In this case, a curve was drawn to show the results with different number of lemmas sent to ATPs

[2]http://wiki.mizar.org/twiki/bin/view/Mizar/MpTP2078

3.3.2.1 HOL Light

Below is an example of feature extraction in HOL Light for the theorem "`EVEN_OR_-ODD`" (3.8) using the methods of *HOL(y)Hammer* (Kaliszyk and Urban, 2014).

$$\forall\, n.\, EVEN\; n \vee ODD\; n \tag{3.8}$$

Given (3.8), the features in (3.9) are extracted as strings.

$$\begin{aligned}&\text{“}num\text{”}, \text{“}fun\text{”}, \text{“}bool\text{”}, \text{“}ODD\text{”}, \text{“}EVEN\text{”},\\ &\text{“}Anum\text{”}, \text{“}EVEN\ Anum\text{”}, \text{“}ODD\ Anum\text{”}\end{aligned} \tag{3.9}$$

These are generated using several approaches. First, all the constants (e.g. "*ODD*", "*EVEN*") and data types (e.g. "num") are collected except for logical connectives and quantifiers (e.g. "$\vee$", "$\forall$"). Then subterms of the statement (e.g. "*EVEN Anum*" and "*ODD Anum*") are collected where variables are normalised in several ways. For instance, in the default method, the identifier of a variable "*n*" is normalised to "*A*", and the feature is then the identifier "*A*" followed by the type of "*n*" (i.e. "$EVEN\ n : num$ is normalised to "*EVEN Anum*", where ": *num*" means the type of "*n*" is natural number). Structure information is kept as subterms (e.g. "*EVEN Anum*"), which provides more information for learning (Kühlwein et al., 2013).

From these symbolic features, a **feature matrix** is then generated. The feature matrix *Fea* is defined by (3.10). Each row in the feature matrix is the feature vector of a statement, say s_i, i.e. $Fea(s_i) = (Fea(i,1), Fea(i,2), ...)$. This corresponds to the feature (i.e. left) part of Table 3.1 and Table 3.2.

$$Fea(i,j) \;=\; \begin{cases} 1 & \text{if (string) feature } f_j \text{ appears in statement } s_i \\ 0 & \text{otherwise} \end{cases} \tag{3.10}$$

3.3.2.2 ACL2

In the work of (Heras et al., 2013), features based on the structures of terms are used rather than strings. For instance, (3.11) is a definition in ACL2.

```
(defthm fact-fact-tail (implies (natp n) (equal (fact-tail n) (fact n))))
```
(3.11)

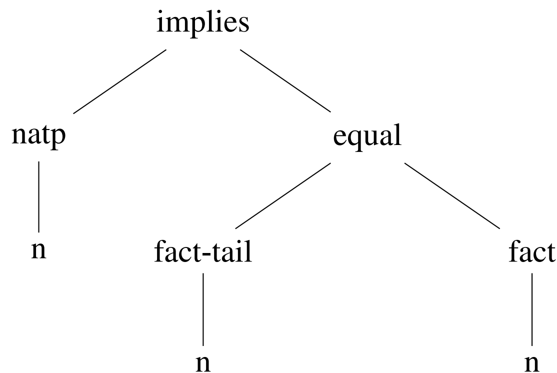


Figure 3.1: Tree of Statement 3.11

	variables	arity 0	arity 1	arity 2
td0	0	0	0	[implies]
td1	0	0	[natp]	[equal]
td2	[n]	0	[fact-tail]::[fact]	0
td3	[n]::[n]	0	0	0

Table 3.3: Matrix for tree in Figure 3.1

The statement in (3.11) is represented in Figure 3.1. The tree is transformed into a matrix, which is shown in Table 3.3. This table sorts the nodes by tree-depth (td). The nodes at the same depth are split according to the arities of the operators. This is a compact way of representing the tree, compared with adjacency matrices. According to the authors, with arities up to 5 and tree-depth up to 7, the matrix is sufficient for all libraries considered in that paper, i.e. the JVM library (Moore, 2003) and the *List* library (Kaufmann and Moore, 2004). Finally, all symbols in Table 3.1, i.e. the symbols of the statements in (3.11) are converted to rational numbers based on the number and order of unique variables in the statement, the termination function ACL2 assigned to a function definition, etc. (Heras et al., 2013).

Compared with the feature extraction from *HOL(y)Hammer*, a notable change is that the rational number is used instead of a binary digit (see (3.10)). However, the biggest difference is that the structure information (e.g. the term tree, and the variable order) is given more attention than when just using function and constant names (e.g. the string “EVEN” in (3.9)).

3.3.2.3 ATPs

Bridges for his part used totally different features (static and dynamic ones) in his approach for ATPs (Bridge, 2010). Static features such as the size and structure of the clauses were used instead of function and variable names. For instance, the features could be separated into two main types: “Fraction of clauses that are unit/horn/ground clauses” and “Maximum/Average clause length/depth/weight”. Dynamic features, which were generated when running the prover with a heuristic were also used. The idea of extracting features from running the prover was creative, although the improvement was not significant. There was also a problem that it was difficult to pick a heuristic to generate the features before knowing which heuristic would be the best. In the paper, the heuristic that was most often selected by E in auto mode was used, and it was used for only a short time to avoid introducing a bias.

3.3.3 Dependency tracking

We refer to the lemmas used to proven a theorem as its dependencies. Dependency information is necessary for predicting whether a theorem might be used again (i.e. to predict new dependencies). This information provides the class label for machine learning mentioned (see Section 3.1.1). The procedure that involves gathering dependency information is called dependency tracking. This is mainly done by automatically processing the proof, either via a patched kernel in systems like HOL Light (Kaliszyk and Urban, 2014), or a background process for systems like Sledgehammer in Isabelle (Kühlwein et al., 2013).

Specifically, for each manual proof, we can get $l_1, l_2, ... \vdash t$, where $l_1, l_2, ...$ are the lemmas, i.e. dependencies, and t is the theorem. The **dependency matrix** is then organised as (3.12). This corresponds to the label (i.e. right) part of Table 3.2. If we fit both the **feature matrix** and **dependency matrix** to Table 3.2, each row (i.e. an instance) exactly corresponds to a whole data point collected from one manual proof, the features are extracted from the statement of the theorem, and the labels are from the proof script.

$$Dep(i,j) = \begin{cases} 1 & \text{if } t_i \text{ depends on } l_j \\ 0 & \text{otherwise} \end{cases} \tag{3.12}$$

The dependencies form a graph which is built according to the following requirements:

- When the theorem is a conjunction, it is better to split it (Kaliszyk and Urban, 2014). This is helpful for ATPs to search and generate proofs, especially when many theorems are conjoined (e.g. ARITH in HOL Light is a conjunction of 108 theorems).
- Each theorem is also considered to depend on itself, i.e. $\forall i.\ Dep(i,i) = 1$ (Kaliszyk and Urban, 2014). This is because most machine learning approaches only select theorems that have been used in a proof. We want all theorems to be used in at least one proof so that they can be selected, otherwise, a theorem just being proven will never be selected for the next proof.

The dependency tracking tool that comes with *HOL(y)Hammer* follows the following procedure (Kaliszyk and Urban, 2014):

1. Load all the libraries for dependency tracking, and get the set T of all the **visible** theorems, which means a theorem is accessible via its identifier.
2. Reload the libraries, and construct the dependency list for each theorem with the ones in T

This approach results in a problem: a theorem T_i may be recorded as depending on theorem T_j, which is proven after T_i. This is because T_j may be generated while proving T_i, but not stored as a theorem at this time. However, if T_j is stored later then a seemingly impossible dependency is recorded.

In order to have dependency information that consists with the HOL Light proof script, we started to use another dependency tracking tool *HOListic*, which was developed as part of the *Proofpeer* project[3].

In addition to an accurate tracking of dependencies, it also collects meta data of theorems. Fig 3.2 is an example of the meta information about theorem "ADD", i.e. the file and line it locates, and its id tracked. Note that there are two id records. One is *source id*, which identifies a theorem in the source file, and the information is collected as the data structure in Fig 3.2. *Tracked id* is derived from source. For instance, "ADD" is tracked as two theorems, because it is a conjunction (see its definition in Fig 3.2). Fig 3.3 focuses on the individual theorem, and has information about its dependencies,

[3]http://www.proofpeer.net/

```
ADD in source file:
  (!n. 0 + n = n) /\
  (!m n. SUC m + n = SUC (m + n))
identifier data:
{
    "source_id" : 203,
    "name" : "ADD",
    "location":
      {
        "filename" : "arith.ml",
        "line" : 46
      },
    "tracked_ids" : [236,237]
}
```

Figure 3.2: Tracked identifier

i.e. a list of *tracked ids* of other theorems, constant symbols, etc. These meta data are organised as JSON files and we discuss their visualisation in Section 4.2.1.

Some theorems are no longer **visible** where their identifiers are reused for other theorems. For instance, one may name a theorem T "A" and then name another theorem T' "A" again, making only T' is **visible** by referring to "A" from this point onwards. However, such theorems are still tracked when using *HOListic*. Therefore, lemma selection with such dependency information may select lemmas that appear no longer accessible for a proof. The solution is to rename the affected theorems, to retain the visibility of the original theorems. This avoids the problem that *HOL(y)Hammer*'s tracking mechanism suffers from.

3.3.4 A quick note on incremental learning

When developing a new theory accumulatively, the training data updates rapidly, i.e. new proofs and lemmas are added every time a new theorem is proven. This requires the *training procedure*, feature extraction, and dependency tracking to be fast. As mentioned in Section 3.1.3, some machine learning algorithms can support *online learning*. SNoW, which is used as part of *HOL(y)Hammer*, has such an option (Carlson et al., 1999). In principle, this should be efficient, but we will outline a problem when it comes to using it later.

```
dependency data
{
 "src_id": 203,
 "tracked_dependencies": [228, 140, 105, 5, 4, 3, 2, 1],
 "type_constants": ["bool", "fun", "num"],
 "new_type_constants": [], "as": ["conjunct", "0"],
 "stringified": "|- !n. 0 + n = n",
 "source_code_theorem_dependencies": [197],
 "tracking_id": 236,
 "new_constants": ["+"],
 "source_code_tactic_dependencies": [],
 "constants": ["!", "=", "+", "NUMERAL", "_0"]
}
```

Figure 3.3: Tracked dependency

3.3.5 Experimental methodology

When evaluating a lemma selection method, we need to make sure that the proof of the conjecture (or goal) for testing has not been learnt during the training. We follow the approach to evaluate *HOL(y)Hammer* (Kaliszyk and Urban, 2014):

Suppose the *dependency matrix* **Dep** and **feature matrix Fea** have been obtained from the data set for evaluation. Theorems are evaluated in order according to **Dep**. When theorem T_i, i.e. the theorem corresponding to the ith row of **Dep** is to be tested, the procedure mentioned in 3.1.1.1 is used, and:

- The training data are the submatrices $Dep[1,2,...,i-1;\ 1,2,...,i-1]$ and $Fea[1,2,...i-1;\ 1,2,...,j]$, where j is the number of features that have been extracted up to theorem T_{i-1}, i.e. the subset of the proofs that are "known" to T_i.
- The testing data, which is the feature vector (i.e. f_{new}) is $Fea[i;1,2,...,k]$, where k is the number of features up to T_i.

3.3.6 Machine learning for inductive theorem proving

There has been related work that has attempted to combine machine learning techniques with inductive theorem proving.

Machine learning for ACL2 has resulted in several extensions, collectively known as

ACL2(ml) (Heras et al., 2013; Heras and Komendantskaya, 2014):

Theorem clustering using the feature extraction method shown in Table 3.3 to cluster theorems based on their similarity. This extension detects the redundancies, and can help users reuse previous theorems and definitions.

Lemma analogy tries to construct a lemma that helps ACL2 to complete a proof. For a theorem that requires auxiliary lemmas (called *target theorem*, TT), *Analogy Mapping* firstly looks for a *source theorem* (ST) that has a similar structure as TT with *theorem clustering*. Then the *Source Lemma* (SL), which has assisted the proof of ST, will be reconstructed as a lemma for TT by a process called *Term Tree Mutation*.

Machine learning for HipSpec (Lindhé and Logren, 2016) The investigation involves several extensions to *HipSpec* (see Section 2.5.6), with machine learning techniques being applied to two aspects:

Selecting the induction variable. Features are extracted as strings, like (3.9), but are separated into different groups. For instance, *function* and *lemma* features are extracted from (Haskell) functions (see Section 2.5.6) and the lemmas. *Symbolic features* are directly from constant names, and *abstract features* are from generalisation, e.g. a function is generalised to *Func*. The induction variables used in proofs are converted to *class labels* for training, based on their order, e.g. if the first, second, or third, ... variable in a statement is used for induction, the label will be "0", "1", or "2", ... respectively. When testing with 96 proven theorems using *Naive Bayes*, 72.47% of the selection was correct.

Clustering similar lemmas This is used to find the common pattern of certain definitions (e.g. tail-recursive function), with the same features used for selecting induction variables. *Abstract* features had better performance on grouping lemmas about tail-recursive functions.

3.4 Conclusion

Machine learning methods and its application to lemma selection were reviewed in this chapter. The general notions underlying hammer systems were introduced as well as some existing work on the application of machine learning methods applied to induc-

tive theorem proving.

In the next chapter, we describe the methodology and the main development and test sets used in this work.

Chapter 4

Methodology

In this chapter, the general strategy used to develop our Boyer-Moore implementation and to combine it with machine learning methods is introduced in Section 4.1. After that, the infrastructure and data sets for our development are introduced in Section 4.2 and Section 4.3. Finally, some of the existing approaches for measuring proof "complexity" are discussed in Section 4.4.

4.1 Research strategy

As mentioned in Section 1.3, our hypothesis is whether the automation of inductive theorem proving in HOL Light can be improved by applying machine learning techniques.

This hypothesis will be tested using the Boyer-Moore Model in HOL Light (see Section 2.5) and evaluated on its ability to prove theorems that require proof by induction. Our empirical approach follows the general strategy summarised in Fig 4.1. We make choices about new methods to incorporate in the Boyer-Moore Model based on the results, e.g. we pick the method that proves the most theorems. We now give some more details about the steps involved:

Implementation of new methods We:

- Start with the original implementation of Boyer-Moore in HOL Light (see Section 2.5).
- Derive new methods inspired by proof failures.

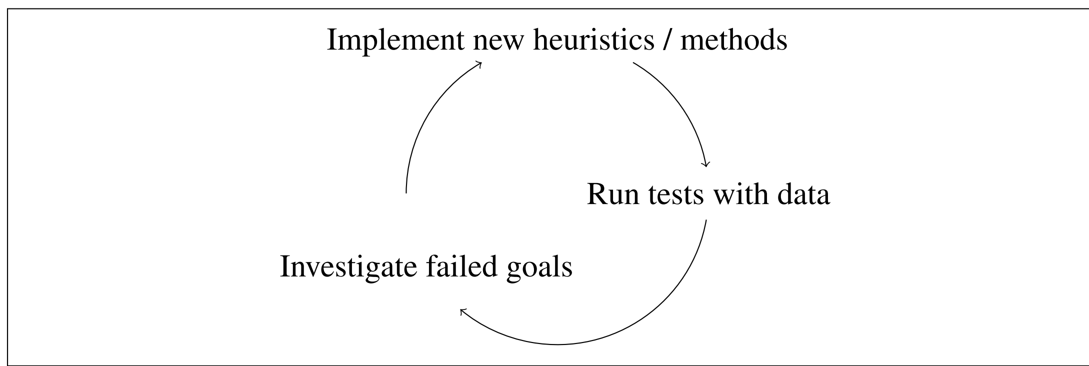


Figure 4.1: Research strategy

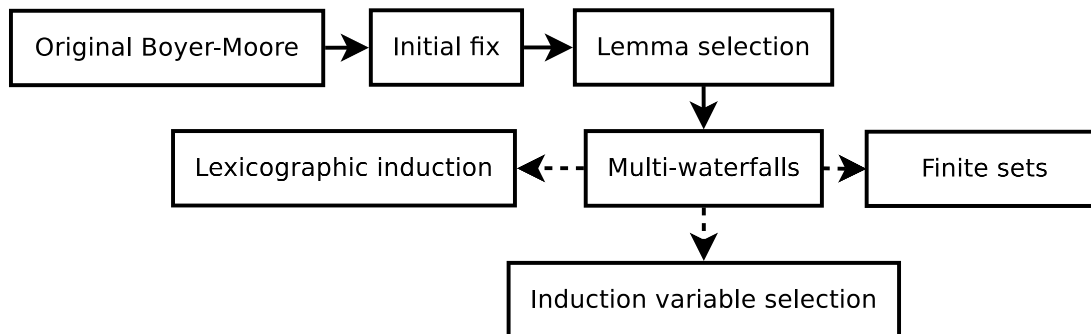


Figure 4.2: Major iterations of improving Boyer-Moore

- Try to incorporate potentially useful machine learning approaches into the Boyer-Moore model.

Run tests with data We test the new implementation with the development data sets in Section 4.3.

Investigate failed goals We look for issues or bugs in our methods, with the approaches described in Section 4.1.1.

During our investigation, we did a number of iterations and the major ones are shown in Fig 4.2. The *initial fix* involved the improvements made to the original Boyer-Moore, which will be discussed in Section 5.1. Then machine learning approaches were applied and optimised. These are introduced in Section 6.3 and Section 5.2 respectively. In Section 6.4, we describe how we investigate machine learning methods to choose variables for induction and in Section 5.3 briefly discuss lexicographic induction. We also discuss a special implementation that supports induction for finite sets in Chapter 8.

4.1.1 Investigating failures

As Fig 4.1 shows, our strategy tests the new methods and is then followed by the most important part: investigating the failures i.e. the theorems that are not proven in the experiment. The cause for failure ranges from problems in the original implementation to the new issues and limitations of any added methods. When running Boyer-Moore, the goal (or subgoals) is passing through various heuristics in the waterfall and induction procedures repeatedly, so such issues are sometimes hard to find. We try to locate them in the following ways:

Classify the theorems Although we are eager to go through the theorems that are not proven in each experiment, it is more efficient to group them as follows than to see all the tens of failed examples:

- The theorems that could be proven before the new method is applied but now are not. These problems can usually be found by comparing the changes to the system.
- Easy theorems: If a theorem has an easy manual proof, we expect the system to prove it with only a few heuristics. Therefore, the problem within the system can be found by testing it with such heuristics. Meanwhile, the failure of easy theorems may relate to fundamental issues which also affects complicated theorems, so we need to deal with them first. Some of our initial fixes (see Section 5.1) are from these theorems.
- Theorems in the same domain, e.g. containing the same function definition, or having a similar structure. We often start from the easiest theorem among them.

Target the failed procedure For each theorem not proven by Boyer-Moore, the next step is to detect the procedure that failed. However, it is sometimes difficult (maybe even impossible) to answer "why was this theorem not proven?", because there may be different proofs for the same theorem and we do not know why none of them were found by Boyer-Moore. Instead, in each case, we focus our investigation on the proof script for the manual proof. We compare it to the steps Boyer-Moore took and detect where the latter could have done better. We look at the manual proofs as follows:

- The proof skeleton, i.e. whether the system chose the correct induction

scheme. If not, the problem or limitation of the induction procedure is then looked into.

- For theorems whose proofs do not involve induction or the induction has been correctly applied, we focus on the waterfall, i.e. the heuristics that make insufficient progress towards proving the goal or subgoals. We are particularly interested in the failures in lemma selection. The dependencies from the manual proof are used:

  - If some of the tracked dependencies are missing from the list of automatically selected lemmas, it is a failure of lemma selection. However, rather than adjust the lemma selection approach for a single example, we look into more examples and compare the performance between the new and old implementations.

  - If no dependencies are missing, the simplify heuristic (see Section 2.5.2) and hammer (see Section 6.3) are looked into. There could be a problem with the ATPs, the translation of the goal (from ITP to ATP) or the reconstruction (see Section 2.4).

Although this is a general, systematic strategy, we adapt our investigation in each case, depending on the context of the proof. There are two particular types of cases that we choose not to investigate further. First, theorems with complicated proofs may not be found using our implementation, e.g. because some lemmas required in a proof are proven locally (and not named or stored) rather than cut into the proof as pre-proven theorems, so they cannot be learnt and suggested by our implementation. Second, ATPs are tried as black boxes and although efforts have been made to improve the translation from ITP to ATP problems, we do not directly investigate ATP-related failures.

4.2 Infrastructure

In order to support the systematic analysis of our results and investigate the proof attempts of our implementation, we built a number of tools. Some of the more salient ones are described next.

4.2.1 Proof library

A common demand during this research was to look up a theorem in the source file by its name and then see its statement and proof. Thanks to dependency tracking (see Section 3.3.3), each theorem is recorded with key metadata, including the file it belongs to, its dependencies, etc. There is a similar demand for the analysis of the results when we need to look up the output for each theorem from Boyer-Moore and compare the results between different settings (i.e. iterations in Section 4.1)

A web-based, graphical user interface was created using *DataTables*[1], such that it allows searching and filtering of the tracked theorem metadata. This includes three tables: *identifier* table, *dependency* table, and *result* table.

As shown in Fig 4.3, the theorems are organised in an *identifier* table, which supports filtering using properties of each theorem: its name (identifier), source file, or id. Hyperlinks also allow the user to directly jump to the line in the source file. For instance, `AND_CLAUSES` is in the file theorems.ml and is tracked as five conjuncts.

The *dependency* table shown in Fig 4.4 contains the tracked dependencies, the constants in the statements, etc. It is a very wide table, so buttons are added to hide unused fields. For example, we can learn from table that theorem with id 103 depends on 1, 2, 3, 4, 102.

The *result* table created is shown in Fig 4.5. Each line shows the outputs from Boyer-Moore for a certain theorem, such as the status (whether it is proven), number of the induction steps used, the last subgoal that the system succeeded/failed to prove etc. Meanwhile, some information from the source file is also listed for convenience, e.g. whether induction is used in the manual proof ("Source induction"). For the theorems that were proven, its data record can be expanded to show the subgoals and the heuristics used to prove them (see `ALL_MEM` in the Fig 4.5).

At the top of the table, various options are available for changing the view on the data. For instance, the first drop-down list is used to choose the test data set. With the drop-down lists on the second line detailing the approach used in the experiment and the options available in the third line. We can compare different methods, e.g. find the theorems proven by one method, but not by another.

[1] https://datatables.net/

hilbert/lists.ml ▾ Load

Table for theorem_idents

Show 10 ▾ entries Search:

src_id	name	file	tracked_ids
30	EXISTS_SIMP	/theorems.ml	38
31	EQ_IMP	/theorems.ml	39
32	EQ_CLAUSES	/theorems.ml	40,41,42,43
33	NOT_CLAUSES_WEAK	/theorems.ml	44,45
34	AND_CLAUSES	/theorems.ml	46,47,48,49,50
35	OR_CLAUSES	/theorems.ml	51,52,53,54,55
36	IMP_CLAUSES	/theorems.ml	56,57,58,59,60
37	EXISTS_UNIQUE_THM	/theorems.ml	61
38	EXISTS_REFL	/theorems.ml	62
39	EXISTS_UNIQUE_REFL	/theorems.ml	63
Searc	Search name	Search file	Search tracked_ids

Showing 31 to 40 of 2,565 entries Previous 1 2 3 4 5 … 257 Next

Figure 4.3: Table of tracked theorem identifiers

4.2.2 Boyer-Moore waterfall analysis

As mentioned in Section 4.1, an important step in our main strategy is to look into failures. The data tables in Fig 4.5 do not show the subgoals Boyer-Moore failed to prove. This is due to the difficulty of tracking the processes when running our *Multi-waterfall model* (see Section 5.2). Meanwhile, even when all the proof attempts (e.g. the subgoals generated and the waterfalls tried) were collected, it is still difficult to find the ones that caused problems. To solve this, we track and visualise the proof attempts (see Section 5.4.1) and organise them as shown in Fig 4.6 and Fig 4.7.

Trees that show the proof attempts provide a useful, intuitive visualisation of proof search. However, such trees can be very big. Fig 4.6 shows an example where the goal cannot be proven, so induction is applied repeatedly causing infinite branching. It is difficult to locate and focus on where Boyer-Moore gets stuck by scrolling a big graph, so an interactive tree graph [2] implemented using the **D3.js** library[3] was also used. This solves this issue: Any parent node can be expanded or contracted with the tree below

[2]http://bl.ocks.org/d3noob/8375092
[3]https://d3js.org/

Table for tracked_theorems

Use buttons to show or hide columns.

Source code tactic looks like proofs (for convenience), but they are different. e.g. a step like REWRITE_TAC[SYM SUC;SUC] is tracked as REWRITE_TAC[SUC].

Toggle column: Src_id - tracked_id_deps - source_code_deps - Constants - Stringified

Show 10 entries Search:

src_id	as	tracked_id	tracked_dependencies
76	EXISTS_UNIQUE	100	1,2,3,4,12,14,20,56,57,58,59,60,62,70,89,99
77	ETA_AX	101	
78	EQ_EXT	102	1,2,3,4,56,57,58,59,60,101
79	FUN_EQ_THM	103	1,2,3,4,102
80	SELECT_AX	104	
81	EXISTS_THM	105	1,2,3,4,5,101,102,104
82	SELECT_REFL	106	1,2,3,4,5,12,16,105
83	SELECT_UNIQUE	107	1,2,3,4,12,101,106
84	EXCLUDED_MIDDLE	108	1,2,3,4,5,6,8,12,16,40,41,42,43,44,51,52,53,54,55,56,57,58,59,60,105
85	BOOL_CASES_AX	109	1,2,3,4,6,7,8,40,41,42,43,44,45,51,52,53,54,55,108
Searc	Search as	Search trac	Search tracked_dependencies

Showing 101 to 110 of 2,902 entries Previous 1 … 10 11 12 … 291 Next

Figure 4.4: Table of dependencies

list_core ▾
3w_defsel ▾ - ori ▾ - ori ▾

Load | Sub | Sub2 | Intersect
Remove first elements from the 1st result
Remove first elements from the 2nd result

Table

Tracked dependencies
Show 10 ▾ entries

name	Status	Total steps	Induciton steps	Source induction	I node	N node	Success/Failed
ALL_APPEND	true	1	0	1	12	23	ALL P (APPEND l1 l2) <=> ALL P l1 /\ AL
ALL_conjunct0	is def	0	0	0	4	6	DEF
ALL_conjunct1	is def	0	0	0	4	6	DEF
ALL_EL	true	1	0	0	9	20	(!i. i < LENGTH l ==> P (EL i l)) <=> ALL
ALL_FILTER	time out	-1	-1	1	17	32	Fail
ALL_IMP	true	4	1	1	14	27	(!P Q. (!x. MEM x a1 /\ P x ==> Q x) /\ ALL MEM x (CONS a0 a1) /\ P x ==> Q x) /\ AL (CONS a0 a1)
ALL_MAP	time out	-1	-1	1	13	24	Fail
ALL_MEM	true	4	1	1	12	24	(!P. (!x. MEM x a1 ==> P x) <=> ALL P a1) ==> P x) <=> ALL P (CONS a0 a1))

Clause (0) :	(!P. (!x. MEM x a1 ==> P x) <=> ALL P a1) ==> ((!x. MEM x (CONS a0 a1) ==> P x) <=> ALL P (CONS a0 a1))#,
End heuristic:	hh:ALL_conjunct1,MEM_conjunct1
Clause (1) :	(!x. MEM x [] ==> P x) <=> ALL P []#,
End heuristic:	hh:ALL_conjunct0,MEM_conjunct0

name	Status	Total steps	Induciton steps	Source induction	I node	N node	Success/Failed
ALL_MP	time out	-1	-1	1	13	25	Fail
ALL_T	true	4	1	1	7	14	ALL (\x. T) a1 ==> ALL (\x. T) (CONS a0 a
Search name	Search St	Search Tc	Search In	Search Sc	**1457(1457 total)**	**2817(2817 total)**	Search Success/Failed Clause

Showing 11 to 20 of 141 entries Previous 1

Figure 4.5: Table of results

ind:Theorems proven by FINITE_INDUCT.
strong: Theorems proven by FINITE_INDUCT_STRONG
ind_tracked_dep FINITE_UNION_IMP Load
Open File Open File1

(FINITE {} ∧ FINITE t ==> FINITE ({} UNION t)) ==> FINITE {x} ∧ FINITE t ==> FINITE ({x} UNION t)

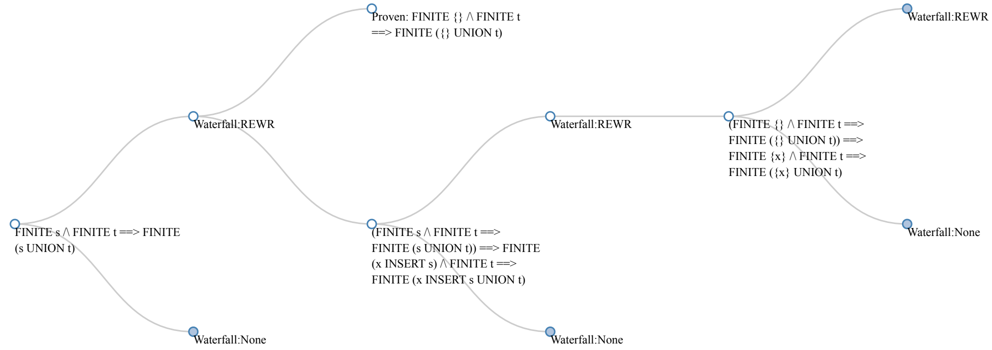


Figure 4.6: Visualisation example

it by click. When a branch is not needed, it can be hidden.

In Fig 4.6, the tree shows the proof attempts for `FINITE_UNION_IMP` where it could not be proven due to the lack of a lemma (see Section 8.5.2). There are two kinds of nodes: the nodes named “Waterfall” indicate the waterfall being run (among the multiple waterfall, see Section 5.2), and the other show the subgoals. “Proven” indicates the proven subgoals. Fig 4.7 displays a completed proof which also shows the variable used for each induction step.

When using this tree graph to investigate the failure, we can gradually expand the subgoals not proven by Boyer-Moore and investigate the reason for failure. For instance, we first check whether the subgoal is provable. If the goal is unprovable then there might be a bug, or some heuristics are over-strengthening the goal e.g. the generalisation heuristic. After that, we inspect the proof search applied to the subgoal e.g. whether induction should be applied, or whether the correct variable is used for induction. This can be achieved by comparing with the manual proof or checking whether the lemmas selected from machine learning are enough to simplify the subgoal.

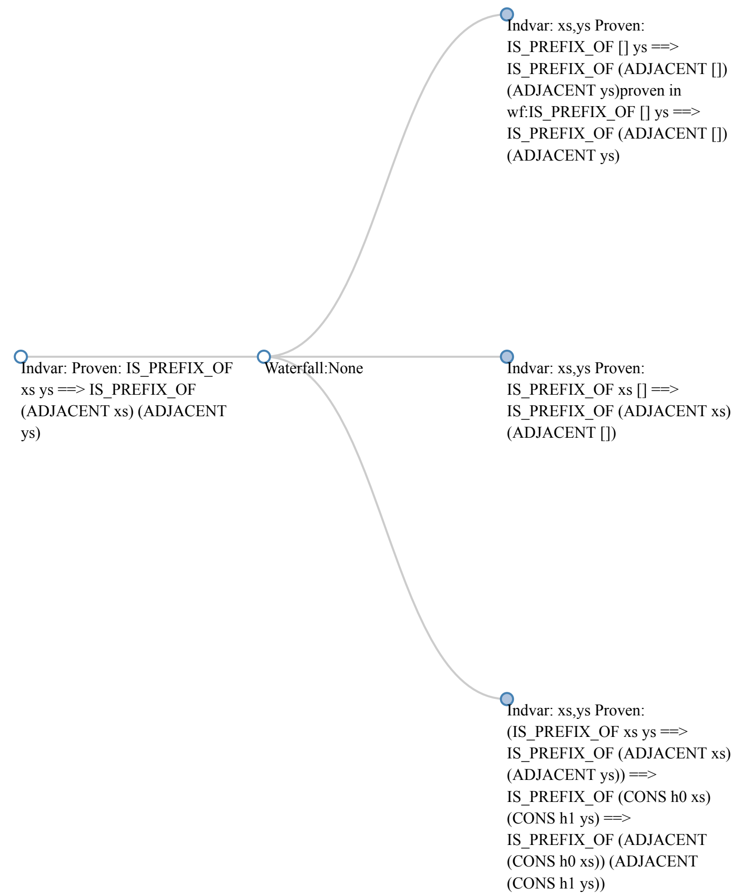


Figure 4.7: Visualisation example

4.3 Experiment data

We use the following data sets for the implementation and evaluation of our Boyer-Moore approach: List theories, used mainly for the Boyer-Moore Model experiments (Section 4.3.1), An arithmetic theory used in the initial experiments for the Boyer-Moore Model (Section 4.3.2), and some libraries ported from Isabelle, and mainly for the evaluation of support for non-primitive recursive definitions (Section 4.3.3 and Section 4.3.4). We describe our datasets in more detail next.

4.3.1 List theory

We use the following corpora in our Boyer-Moore Model experiments (see Appendix A.2 also):

The core list library in HOL Light, which is loaded by default in HOL Light and we will refer to as *List*(*core*). It contains the basic definitions of list theory, such as the head and tail, length, reverse, etc. Most proofs in it do not contain many lemmas, i.e. they are generally proven using only definitions. Since it is a file loaded by HOL Light in a very early stage, there are only round 700 theorems available for selection. This amount is relatively small as we usually select more than two hundred lemmas for the ATPs (see Section 6.3), so we need extra libraries. See Appendix A.2.1 for the list of theorems.

Hilbert List library This is used in the formalization of Hilbert's Foundations of Geometry (Scott, 2015). We refer to this as *List*(*hilbert*). It has more definitions and theorems about lists, such as `TAKE` and `DROP` i.e. the functions that keep and remove the first n elements of a list respectively. These definitions are non-primitive recursive (see Section 5.3). It also has a customised induction scheme, which performs a two-step induction. If we use a fixed induction scheme provided by Boyer-Moore (see Section 2.3.1), two induction steps will be required, which is thus more challenging. See Appendix A.2.2 for these theorems.

Polynomials library This contains properties about real polynomials represented as lists of coefficients. We refer to this as *Poly*. This will only be used for evaluation.

In this, a polynomial "$a_1 + a_2x + a_3x^2...$" is defined as "$poly\ [a_1;a_2;a_3...]\ x =$

	Definitions	Theorems	Inductive
List(*core*)	44	97	73 (75.26%)
List(*hilbert*)	22	115	80 (69.57%)
Poly	20	123	67 (54.47%)

Table 4.1: List theory datasets

$a_1 + x(a_2 + x(a_3 + ...))$", where $[a_1;a_2;a_3]$ is the list of coefficients. The formal definition of *poly* is given in (4.1). An example of polynomial operation is (4.2). It calculates the addition of polynomials e.g. $(a_1 + a_2x + a_3x^2...) + (b_1 + b_2x + b_3x^2...) = (a_1 + b_1) + (a_2 + b_2)x + (a_3 + b_3)x^2...$ as the entry-wise sum of the coefficient lists: $[a_1;a_2;...] + +[b_1;b_2;...]$ where "++" stands for the addition of polynomials. See Appendix A.2.3 for these theorems.

$$poly\ []\ x = 0 \wedge poly\ (CONS\ h\ t)\ x = h + x * (poly\ t\ x) \tag{4.1}$$

$$\begin{aligned}[] + +l = l \wedge (CONS\ h\ t) + +l = \\ (if\ l = []\ then\ (CONS\ h\ t)\ else\ (CONS\ (h + HD\ l)\ (t + +TL\ l))\end{aligned} \tag{4.2}$$

The sizes of the datasets are shown in Table 4.1. Note that conjunctions have been split, meaning that a theorem (or definition) $P \wedge Q$ is automatically split into P and Q as separate goals (or definitions).

Note that the number of inductive proofs is a lower bound, obtained by tallying the proofs containing the string "`INDUCT`". In our current datasets, we did not observe any inductive proofs that were not captured in this way, but this is not necessarily true for other libraries. Since induction can be applied in various ways in HOL Light (e.g. by matching different induction rules), it is somewhat difficult to automatically determine the exact number of inductive proofs. However, we believe that our approach is pretty accurate.

	Definitions	Theorems	Inductive
Arithmetic	351	754	109 (14.46%)

Table 4.2: Arithmetic (Gödel) dataset

4.3.2 Arithmetic

When we started our investigation, the arithmetic in HOL Light core library[4] was used for the development of our Boyer-Moore implementation. It contains theorems about basic arithmetic over the natural numbers. The Boyer-Moore implementation already contains definitions and lemmas, and can be directly used to prove the theorems. Therefore, it was suitable for our initial experiment. However, the theorems are generally easy and the theorems about linear arithmetic can usually be proven with HOL Light's decision procedure `ARITH_RULE` (see Section 2.1). This library will be referred to as *Arith(core)*.

Another library about arithmetic, called *Arithmetic*[5] was also considered. It contains definitions and theorems about Gödel's incompleteness theorem and the Sigma-1 completeness of Robinson's arithmetic. This library rebuilds first order logic to provide a language for arithmetic, so it has its own recursive type for numbers and formulae. Note that the symbols are different from the default syntax in Section 2.1.1 to avoid conflict.

After further investigation, we found it unsuitable for our experiments, because it only contains a relatively small proportion of inductive proofs, (see Table 4.2). In addition, many of the inductive proofs are not about recursive data types, so the Boyer-Moore model is not generally applicable. Therefore, it was not used for later experiments and evaluations.

4.3.3 The IsaPlanner Benchmark

The *IsaPlanner benchmark* consists of theorems about natural numbers, lists, and binary trees (Johansson et al., 2010).

The test data is self-contained i.e. all definitions are included, independent from the

[4] https://github.com/jrh13/hol-light/blob/master/arith.ml
[5] https://github.com/jrh13/hol-light/tree/master/Arithmetic

	Definitions	Theorems
Natural Numbers	9	24
Lists	22	62
Binary Trees	5	1
Total	36	87

Table 4.3: *IsaPlanner benchmark*

Topic	Definitions	Theorems	Inductive
TAKE, DROP	0	4	3
List update	3	20	13
Sorted	2	9	3
Total	5	33	19

Table 4.4: Hoare Logic dataset characteristics

corresponding types and definitions in HOL Light. No other lemmas are provided and the ones from HOL Light cannot be used, therefore lemma selection and ATPs are not helpful. For this reason, this corpus was only used in the Boyer-Moore evaluation that does not involve machine learning techniques. See Appendix A.2.4 for the list of theorems.

4.3.4 Lemmas for Hoare Logic

A group of definitions and lemmas were ported from Isabelle proof assistant[6] for the experiments in Chapter 9. Some of the main characteristics of this dataset are shown in Table 4.4. Here *List update* means the updating of one element of a list i.e. $list_update\ l\ i\ n$ replaces the ith element of list l with n, as shown in (4.3). See Appendix A.2.5 for the list of theorems.

$$
\begin{aligned}
&list_update\ []\ i\ n = []\wedge \\
&list_update\ (cons\ h\ t)\ 0\ n = cons\ n\ t\wedge \\
&list_update\ (cons\ h\ t)\ (s(i))n = cons\ h\ (list_update\ t\ i\ n)
\end{aligned}
\tag{4.3}
$$

[6]https://isabelle.in.tum.de/library/HOL/HOL/List.html

Library	Definitions	Theorems	Inductive	Dev	Eval
List(*core*)	44	97	73	*BM*	
List(*hilbert*)	22	115	80	*BM*	*Lex*, *IndMl*
poly	20	123	67		*BM*, *IndMl*
Arithmetic	351	754	109	*BM*	
IsaPlanner (Sec. 4.3.3)	36	87	-		*Lex*
Hoare Logic (Sec. 4.3.4)	5	33	19		*Lex*

Table 4.5: Summary of all datasets

All the data sets in this section are summarised in Table 4.5. "Dev" or "Eval" means the data set is used for the development or evaluation respectively for the three main tasks: improving Boyer-Moore ("*BM*"), machine learning for selecting induction variables ("*IndMl*"), and introducing lexicographic induction ("*Lex*"). The *Isaplanner benchmark* does not contain proofs, so we could not determine the number of inductive proofs.

4.3.5 Data split

When using the aforementioned data, we need to make sure that no proofs or lemmas used for training are used for testing. We follow the approach mentioned in Section 3.3.5 and split the data such that training is done on the first n theorems in a library and testing is on the $(n+1)th$ theorem.

4.4 Proof metrics

An automated theorem proving system may achieve a high proof rate, but the difficulty of the proven theorems is sometimes not clear. Therefore, in addition to calculating the success rate, we are interested in the question "how much do automated tools like *HOL(y)Hammer*, or our Boyer-Moore based system help the user?". The answer should be from the user's point of view, thus we propose that the difficulty of a goal is correlated to the complexity of its manual proof (rather than dependencies).

A direct way of analysing the manual proof is to count the number of lines. However, the number of lines cannot be used as a criterion in HOL Light. An arbitrary number of tactic applications can be written in a single line, so the number of lines of proof

can vary according to the users' proof style and how its steps are packaged together. In order to measure the "complexity" of a proof in a more reliable way, we decided to count the number of nodes in the parse tree of the proof instead.

4.4.1 Proof complexity

There has been some research on characterising notions of proof complexity when it comes to ITP proofs. In some of the approaches, the results from ATPs are used for comparison as shown below. In this context, we call proofs from ITP libraries *manual proofs* to contrast them to the *automatic* proofs generated by ATPs.

Number of lemmas in the proof. Here, this refers to the number of necessary lemmas gathered from dependency tracking in manual proofs (see Section 3.3.3), or directly from ATP proofs. This was used for comparing the manual proofs and ATP proofs (Alama et al., 2012) (Kaliszyk and Urban, 2014). The limitation of this approach is that the tactics in HOL Light can result in irrelevant lemmas. For example, VEC3_TAC tactic results in 121 lemmas when tracking the dependencies (Kaliszyk and Urban, 2014). However, even if all the lemmas in the tactic are relevant, the user only needs to remember the tactic, rather than a group of lemmas, which suggests that the number of lemmas may not reflect the actual difficulty of the goal for humans.

Number of manual proof lines. The number of proof steps (lines) was used as a metric of measuring proof complexity in Mizar. The assumption was that "the Mizar weak refutational checker enforces a relatively uniform degree of derivational complexity on all Mizar proof steps" (Alama et al., 2012), which is close to the number of proof lines. For ATP proofs, the ATP inference lines (i.e. ATP proof steps) were used, and there was a conversion ratio proposed for the convenience of comparing between ATP and Mizar proof (Alama et al., 2012).

In Mizar and Isabelle with Isar syntax, the proofs are well structured, so the number of lines of proof are not strongly affected by the proof style, and can be viewed as correlated to the difficulty of the goal. However, this is not the case in HOL Light: The proof steps are not separated by lines and are often one line long. Therefore, the number of lines does not seem to be a uniformly reliable metric either.

```
prove
(`!n. CARD(1..n) = n`,
REWRITE_TAC[CARD_NUMSEG] THEN ARITH_TAC)
```

Figure 4.8: Proof of "*CARD_NUMSEG_1*"

Transitive recursions. Some dependencies of a theorem may also have their own dependencies. For instance, if $A \vdash B$ and $B,\ C \vdash D$, we can replace B with A and get $A,\ C \vdash D$. This process can be recursively repeated until all the dependencies are definitions or axioms. In Mizar, Alama et al. recursively obtained the dependencies of the lemmas until they had the transitive closure of all the lemmas used in a proof, and count the number of lemmas and proof lines (i.e. the previous two metrics) by summing up the values (e.g. the number of proof lines) in the transitive closure. The motivation was to "capture the mathematician's intuition of proof complexity as the set of the **proofs that need to be understood**" (Alama et al., 2012).

Some conclusions have been made based on these metrics: hammers tend to provide proofs with fewer lemmas than manual ones, because they "learn" new lemmas faster than humans, i.e. newly proven lemmas are likely to be used by hammers. Such lemmas have larger transitive lemma numbers and proof lines (Alama et al., 2012), which indicates that they tend to have more complicated statements than basic definitions and theorems (Kaliszyk and Urban, 2014).

4.4.2 Methodology

HOL Light is written in OCaml (see Section 2.1), and the proof structures can be obtained from the parse tree of OCaml. For instance, in each assignment of a theorem, such as "*let A = B*" (A is a theorem, and B is its proof), we process the parse tree and count the number of nodes in it.

A simple example is the theorem "*CARD_NUMSEG_1*" (a theorem stating that the cardinality of the set $\{1,2,3,...n\}$ is n), given below. Its proof as found in the HOL Light library is shown in Fig 4.8.

The corresponding OCaml parse tree is shown in Fig. 4.9, where labels like "*Texp_apply*" are constructors used by OCaml, and names in brackets are the identifiers and constants

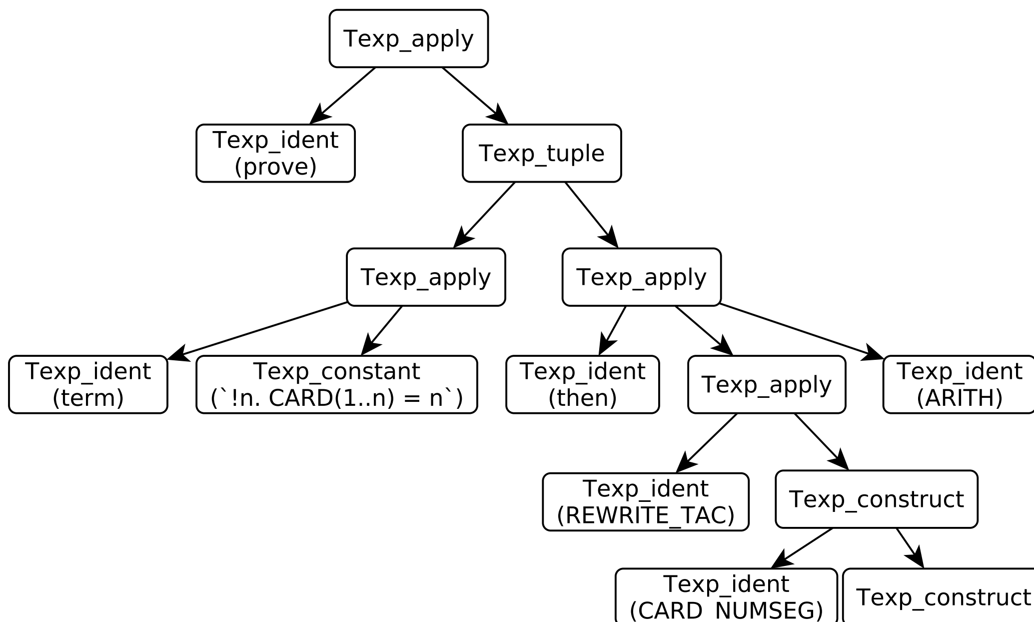


Figure 4.9: Parse tree of the proof of ”CARD_NUMSEG_1”

(e.g. the statement of the goal) in the manual proof. We introduce two metrics based on the parse tree:

1. *I* metric: Counting the number of identifiers, i.e. all the items visible in the proof script. According to Fig 4.8, our method extracts the following nodes from this proof:

   “*prove*”, “*A TERM*”, “*REWRITE_TAC*”,
   “*CARD_NUMSEG*”, “*THEN*”, “*ARITH_TAC*”

   There are six nodes, which is consistent with the proof size. Note that *I* metric actually counts the number of leaf nodes, such as “*Texp_ident*”.

2. *N* metric: Counting the total number of nodes in the parse tree. There are 14 nodes in Fig 4.9.

We believe that the *I* metric can actually reflect the number of steps in a proof. Each step here can be:

1. A tactic. For users, more complicated goals usually require more steps, or

2. The number of lemmas used by tactics, which we believe is related to the effort by users to think of or look for them.

These two aspects can reflect a user's effort in figuring out a proof. In both *I* and *N* metric, only the tactic itself is considered, which we believe to be what the user actually has in mind, rather than the dependencies of the tactic. For instance, if we use the number of lemmas from tracking dependencies in this manual proof (Fig 4.8) as a proof metric, there will be 124 lemmas (mostly from *ARITH_TAC*). Intuitively, humans just need to remember the *tactic* name, rather than all its dependencies. For the same reason, we do not gather lemmas recursively (see Section 4.4.1). On the other hand, the *I* and *N* metrics are also more precise than using the line count for proofs, which would be one in this example.

There are some issues while applying this method across the whole library (i.e. the manual proof scripts in HOL Light). These two proof metrics currently cannot deal with theorems in the following situations:

1. Sometimes theorems are proven as a group (e.g. "*let A, B=...*" will assign two theorems proven in one proof on the right hand side to *A* and *B* respectively). We have not figured out a way of assigning the nodes to each theorem.

2. Theorems may be proven several times, or assigned different names, so a theorem may have multiple proofs. It is difficult to choose a proof for such theorems legitimately. For instance, after defining a theorem *A*, one may define theorem "*let B = A*". In this case, *B* is proven by just copying *A*, which does not correspond to its complexity, because *B* and *A* should have the same complexity.

Nevertheless, we believe that we have a good way of characterising proof differently, especially when compared with relatively coarse ones such as the number of lines in a proof. We will apply these metrics in our evaluation to compare the theorems proven by *HOL(y)Hammer* and our Boyer-Moore implementation in Section 7.1.6.

4.5 Conclusion

We introduced an empirical methodology aimed at improving the Boyer-Moore tool and its supporting infrastructure in HOL Light. The datasets used for the evaluation of our approach as well as a metric for quantifying proof "complexity" were also provided.

In the following chapters, we will carry out our investigation of the Boyer-Moore

model and its combination with machine learning based on these concepts.

Chapter 5

Improving the Boyer-Moore Model

5.1 Improve the waterfall with experiment

In this section, the improvements made to the Boyer-Moore model with the strategy described in Section 4.1 are examined. Some of the datasets in Section 4.3.1 are used for the development and improvement. In particular, *List*(*core*) and *List*(*hilbert*) are mainly used for the development. *Arithmetic* and *Arith(core)* were also used in initial experiments.

5.1.1 Removing clausal form heuristic

During our initial experiments, some goals became unprovable by Boyer-Moore after the CNF heuristic was applied. For instance, when testing with the theorem `NOT_EVEN` (a natural number is not even, if and only if it is odd) in *Arith(core)*, the CNF heuristic splits $\neg EVEN\ x \iff ODD\ x$ into two clauses: $EVEN\ x \vee ODD\ x$ and $\neg EVEN\ x \vee \neg ODD\ x$. In the manual proof, `NOT_EVEN` is actually proven independently and then used to prove the two clauses, so it is unreasonable to expect Boyer-Moore to now prove them as subgoals for `NOT_EVEN`. This is an indication that the CNF heuristic does not always make progress in the right direction towards a proof.

Moreover, the CNF heuristic breaks goals that contain if-and-only-if into subgoals containing implication, leading to a generation of a number of subgoals that is exponential to the number of equalities encountered in the original goal. Therefore, removing it can significantly reduce the total number of subgoals.

```
Statement: coprime(a,b) ==> coprime(a EXP n,b EXP n)
CNF: ~coprime (a,b) \/ coprime (a EXP n,b EXP n)
```

Figure 5.1: COPRIME_EXP_IMP and its CNF

It is worth noting that removing the CNF heuristic directly affects some of the heuristics that come later in the waterfall e.g. the **Substitution heuristic**, which rely on CNF (see Section 2.5.2). However, `SIMP_CONV` in the **Simplify heuristic** performs conditional rewriting (see Section 2.5.2) and works better without CNF because the implication in the theorem is removed in CNF. For instance, `COPRIME_EXP_IMP` in *Arithmetic*, whose statement and its CNF is shown in Fig 5.1, could be proven without CNF by `SIMP_CONV` and lemmas. Therefore, despite the side-effect, our experiments showed a significant overall improvement in the performance of Boyer-Moore without the CNF heuristic. For example, the original Boyer-Moore implementation can prove 47% of the theorems in *List*(*core*), whereas removing the CNF heuristic allows it to prove 60% (see Table 5.1).) Although some goals were no longer provable by Boyer-Moore without the CNF heuristic, we considered the CNF heuristic detrimental to the Boyer-Moore model in HOL Light and removed it.

5.1.2 Generalising variables

While testing with *List*(*hilbert*), the theorem `TAKE_DROP` could not be proven after most fixes had been done and even after lemma selection was added to the Boyer-Moore model (see Section 6.3). This states that appending the sub-list that takes the first n elements of a given list xs to the sub-list that drops the first n elements of xs yields xs (5.1).

$$\forall\, n\, xs.\, APPEND\ (TAKE\ n\ xs)\ (DROP\ n\ xs) = xs \tag{5.1}$$

We carried out a comparison with the manual proof. Both manual proof and Boyer-Moore performed induction on xs. The manual induction produced the subgoal (5.2) for induction step, but Boyer-Moore produced (5.3), which it could not prove.

$$\begin{aligned} &\forall\, n.\, APPEND\ (TAKE\ n\ t)(DROP\ n\ t) = t \\ &\vdash \forall\, n.\, APPEND\ (TAKE\ n\ (CONS\ h\ t))\ (DROP\ n\ (CONS\ h\ t)) = CONS\ h\ t \end{aligned} \tag{5.2}$$

$$\begin{aligned}&APPEND\ (TAKE\ n\ t)(DROP\ n\ t) = t\\ &\implies APPEND\ (TAKE\ n\ (CONS\ h\ t))\ (DROP\ n\ (CONS\ h\ t)) = CONS\ h\ t\end{aligned} \tag{5.3}$$

We explain this issue with a general and easy example. When applying induction to a formula with more than one universally quantified variable, only one is typically selected for induction, and the others are not affected (Bundy, 2001). For example, applying induction on variable n in the formula $\forall\ n\ m.\ Q(n,m)$ yields the following step case:

$$\forall\ n.\ (\forall m.\ Q(n,m)) \implies (\forall m.\ Q(s(n),m)) \tag{5.4}$$

Notice that the other universally quantified variable m remains unaffected (Bundy, 2001). However, in Boyer-Moore the input formula is always *quantifier-free* (see Section 2.5.2), so the step case generated is the following instead:

$$Q(n,m) \implies Q(s(n),m) \tag{5.5}$$

This subgoal may be more difficult to prove in certain cases compared to its general counterpart (5.4), since m is the same in the induction hypothesis and conclusion. Our solution is to firstly universally quantify all free variables other than the one for induction. Applying induction then yields the same subgoal as (5.4), which is shown as (5.6), though we then remove the quantifiers again to fit to the quantifier-free environment of Boyer-Moore.

$$(\forall m.\ Q(n,m)) \implies Q(s(n),m') \tag{5.6}$$

Generally, this fixes the problem in (5.5) and allows induction on multiple variables successively. It gives the flexibility to instantiate m to not only m' in (5.6). However, this remains an open issue when to just match m to m' and yield a simpler goal.

5.1.3 HOL Light's automated procedures

During early experiments, we identified (sub)goals that could be proven by HOL Light's automated model elimination procedure MESON. For instance, the theorem `list_CASES` from *List*(*core*) has the subgoal shown in (5.7), which is a tautology. Such subgoals with existential quantifiers are usually difficult for Boyer-Moore because the heuristics such as the **Substitution heuristic** and the **Tautology heuristic**

Arithmetic	METIS	MESON	METIS ∩ MESON
Original Boyer-Moore	131	135	129
Removing CNF	161	167	160
List(*core*)	METIS	MESON	METIS ∩ MESON
Original Boyer-Moore	48	46	46
Removing CNF	58	58	57

Table 5.1: Initial experiment for comparing MESON and METIS

only deal with the literal in the goal (see Section 2.5.2); The **Simplify heuristic** requires suitable lemmas to eliminate such quantifiers. On the other hand, HOL Light's automated procedures MESON and METIS can easily handle such goals.

$$a_1 = [] \vee (\exists\, h\, t.\ a_1 = CONS\ h\ t) \implies (\exists\, h\, t. a_0 = h \wedge a_1 = t) \tag{5.7}$$

Some experiments summarised in Table 5.1[1] showed that METIS and MESON had similar performance, so we decided to only add MESON as a heuristic in the waterfall.

5.1.4 Forced induction

As mentioned previously (Section 2.5.4), the induction heuristic in Boyer-Moore can perform recursion analysis only for primitive recursive function definitions. This means the original Boyer-Moore fails to perform induction when faced with non-primitive recursive functions as it is unable to choose an appropriate variable. We address this problem by forcing Boyer-Moore to pick the first free variable with a recursive type for induction if no other suitable selection is found by the original heuristic. Random picking was avoided to make the results reproducible. So, for example, when trying to prove `TAKE_DROP` (5.1), our strategy will pick n for induction. In the later part of the thesis, we also consider the use of machine learning techniques (see Section 6.4) as well as the *Lexicographic induction* (see Section 5.3) as a more sophisticated mechanism for the selection of variables for induction.

[1] we used all lemmas tracked from manual proofs as rewrite rules, so the results were higher than those in Section 7.1

	Total		Induction	
	List(*core*)	*List*(*hilbert*)	*List*(*core*)	*List*(*hilbert*)
Original	41.24%	14.78%	36.99%	8.75%
Initial	52.58%	20.00%	45.21%	17.50%

Table 5.2: Success rates comparison after applying initial fix to Boyer-Moore

5.1.5 Summary

When the optimisations and changes mentioned in the previous sections are combined and tested, the results shown in Table 5.2 are obtained. We can already observe some improvements in proof automation for the two libraries *List*(*core*) and *List*(*hilbert*).

5.2 The Multi-waterfall model

The original setup of the waterfall works in a serial, monolithic way. Each heuristic is tried sequentially in a static order. However, certain proofs may require different configurations or strategies for different subgoals. Moreover, some of the Boyer-Moore heuristics may naturally get stuck during a proof. For example, certain combinations of rewrite rules may cause the Simplify heuristic to loop endlessly. This is particularly important in the context of automated lemma selection where we have less control over looping rewrite rule sets. Using a different configuration might help unlock and make progress with the proof.

In order to achieve a more flexible implementation that does not rely on a single configuration, we introduce a *Multi-waterfall model*. In this, we run multiple waterfalls with different configurations in parallel and with a preset timeout. We then have the following possible outcomes:

1. One of the waterfalls succeeds and the corresponding (sub)goal is proven. The proof of the (sub)goal is reconstructed and propagated upwards (as in the standard waterfall model), ensuring the soundness of the overall proof.

2. One of the waterfalls completes, having generated new subgoals that reached their pools. In this case, we apply induction to all unproven goals as in the standard waterfall model (see Section 2.5.1). We then apply the same set of

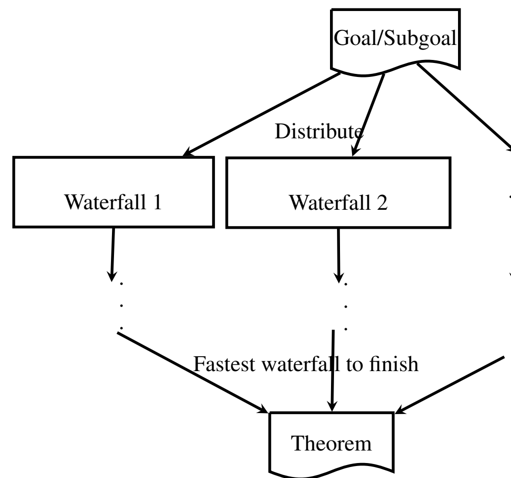


Figure 5.2: Two waterfalls in parallel

multiple waterfalls to each of the new subgoals generated by induction.

3. All the waterfalls determine the goal is unprovable, or the timeout is reached. In this case, the whole branch of proof search fails and is discarded.

An example of this model is shown as Fig 5.2. The timeout applied to each waterfall ensures that any waterfalls that take too long are assumed to have failed and are forcibly stopped and their corresponding branches abandoned. This allows the other waterfalls running in parallel to still potentially make progress towards the proof.

An example search tree with two waterfalls is shown in Fig. 5.3. The waterfalls are run in parallel on the same goal. When a waterfall finishes, we apply induction to any unproven subgoals in its pool, constructing new subgoals indicated by the dashed arrows. We then start new waterfalls for each generated subgoal until all subgoals are proven or deemed unprovable.

A full proof can be reconstructed by tracking all successful waterfalls in a branch. This means a proof may be found by a chain of different waterfalls. In Fig. 5.3, for example, the proof is reconstructed by the waterfalls enclosed in the marked area. Notice that both types of waterfalls were used to make progress on or prove different subgoals.

In our implementation, we spawn the waterfalls for a particular goal using threaded concurrency. If a waterfall fully proves a goal (such as Waterfall 1” in Fig. 5.3), the other waterfalls working on the same goal (such as Waterfall 2”) and their children no longer need to run and are forcibly stopped in order to release system resources. Waterfalls could be tried sequentially instead, for example in a machine with limited

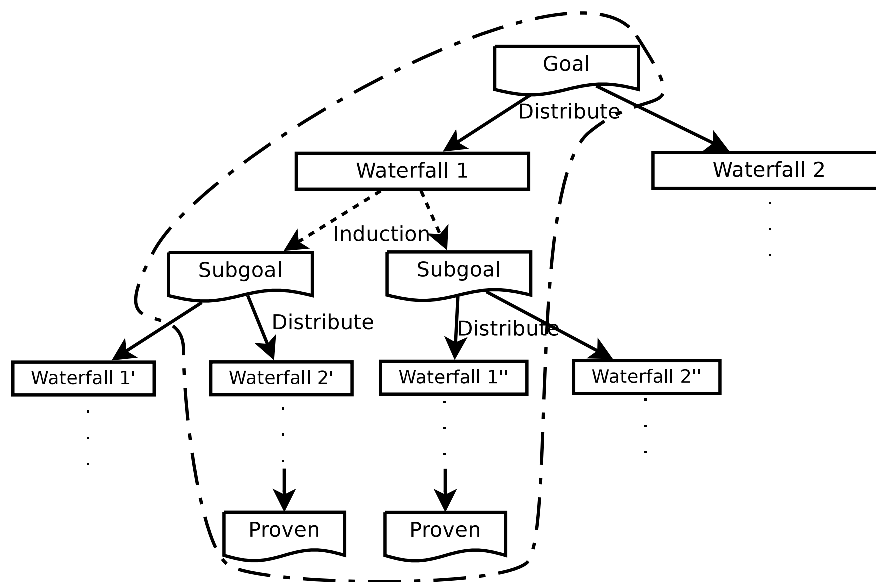


Figure 5.3: Proof search with multi-waterfall

capabilities for parallel computation or memory capacity, but this would dramatically increase the times for a proof to complete. For instance, the user will need to wait for different waterfalls to timeout for each and every subgoal.

5.3 Extending induction heuristic to support non-primitive recursive definitions

5.3.1 Motivation

As mentioned in Section 5.1.4, non-primitive recursive definitions are not supported by the HOL Light version of Boyer-Moore. One solution is to use machine learning approaches to select the induction variable. Meanwhile, heuristics for selecting induction variables for goals containing general recursive definitions do exist (Kaufmann et al., 2000, Chapter 15). We mainly adapt the lexicographic induction approach, independently from the machine learning approach, with the following motivations:

- Supporting non-primitive recursive definitions.
- Comparing its performance with our machine learning approaches for selecting induction variable.

5.3.2 Lexicographic induction and related techniques

Here we look at some of the induction heuristics described by (Boyer and Moore, 1979). Some heuristics may not be applicable to HOL Light (because, for example, they refer to systems with destructive style definitions, see Section 2.5.4.1), so only applicable heuristics are introduced.

Lexicographic induction For primitive recursive definition, there is only one recursion argument position, so the induction rule for the recursive type can be directly used. For non-primitive recursive definition, the recursion position is not fixed. For instance, in the definition of `TAKE` shown in (5.8) (s is the successor function), both n and xs are at the recursion argument in *TAKE n xs*, and the tuple (n,xs) has the *lexicographic relation*. This relation forms an induction rule (5.9).

$$
\begin{aligned}
&TAKE\ 0\ xs = [] \wedge \\
&TAKE\ n\ [] = [] \wedge \\
&TAKE\ (s(n))\ (cons\ x\ xs) = cons\ x\ (TAKE\ n\ xs)
\end{aligned}
\tag{5.8}
$$

$$
\begin{aligned}
\forall P.\ &(\forall l : (A)list.\ P\ 0\ l) \wedge \\
&(\forall n : num.\ P\ n\ []) \wedge \\
&(\forall n\ h\ t.\ P\ n\ t \implies P\ (s(n))\ (cons\ h\ t)) \\
&\implies (\forall n\ l.\ P\ n\ l)
\end{aligned}
\tag{5.9}
$$

Merging induction schemes When there is more than one induction scheme (see Section 2.3.1) suggested, the schemes should be merged. For instance, in the goal $a < b \wedge b < c \implies a < c$, about natural numbers, both $\{a,b\}$ and $\{b,c\}$ are considered for induction[2], based on the general recursive definition of $<$. This heuristic will merge these schemes and apply induction on $\{a,b,c\}$, i.e. all three variables. Note that merging heuristic is only applied under certain conditions.

Tie breaking rules Tie breaking rules are necessary when there is more than one induction scheme remaining. Such rules are choosing the scheme

- that has the maximum number of variables,

[2]The general recursive definition of $<$ is: $(x < 0 \iff F) \wedge (0 < s(y) \iff T) \wedge (s(x) < s(y) \iff x < y)$

- whose variables are all in unflawed position, i.e. a variable is always in the recursion argument position (see Section 2.5.4)

There are more tie breaking rules, but they are rarely applicable. Often an arbitrary pick (e.g. the first scheme) is used after all tie breaking rules failed.

5.3.3 Methodology

These heuristics need to be implemented in Boyer-Moore for HOL Light. The implementation of **Merging induction schemes** and **Tie breaking rules** are straightforward. It takes more work to have **Lexicographic induction** in HOL Light: A new function for adding definitions is required, which is able to figure out more than one induction position from the definition. The detection of the recursive argument is achieved by extending the existing Boyer-Moore implementation, which looks for the application of a constructor (e.g. s(n)). We make it accept more than one such arguments.

A lexicographic relation L can be constructed by two relations R and S :

$$\begin{aligned} L\ (a_1,b_1)\ (a_2,b_2) \iff & R\ a_1\ a_2 \vee \\ & a_1 = a_2 \wedge S\ b_1\ b_2 \end{aligned} \tag{5.10}$$

Various induction rules can be derived based on such relation. Unfortunately, HOL Light does not provide a corresponding induction rule for a recursive function definition like Isabelle (Nipkow, 2013, Section 2.3.4). In our current implementation, all variables recur simultaneously with the relation shown as (5.11), so the induction rule is like (5.9). In the future, a sophisticated method should be used, which detects the decrease of the measure for the variable tuples and provides more induction rules.

$$(n,xs) \prec (n',xs') \iff n' = s(n) \wedge xs' = cons\ h\ xs \tag{5.11}$$

In HOL Light, induction requires instantiating the induction rule, so induction rules for lexicographic relations need to be generated and proven. This is challenging because the induction rule is no longer limited to the one that comes with the type definitions, but are based on any combinations of variables. For instance, the induction rule for the combination of a natural (n) and a list (l) is shown in (5.9). This rule consists of two

base cases, where each variable is in their base case respectively, followed by one step case.

This rule can be generated automatically based on the induction scheme. The induction rule is proven by applying induction on the first variable and then rewriting all the remaining variables. For instance, in order to prove (5.9), we firstly strip the quantifier from P and match the implication consequent with the induction rule for natural number (see (2.5) in Section 2.3.1), which is the type of n and get (5.12). (5.12) is very similar to (5.9) except for the underlined part. We then rewrite the consequent of this part with the rule (5.13), which can be derived from the induction rule for lists (i.e. the type of l) and get (5.14). Finally, we can prove that (5.9) is a consequence of (5.14) by simplification.

$$\vdash (\forall l.\, P\; 0\; l) \wedge \underline{(\forall n.(\forall l. P\; n\; l) \implies (\forall l.\, P\; (s(n))\; l))} \implies (\forall\, n\; l.\, P\; n\; l) \tag{5.12}$$

$$\vdash (\forall l.\, P\; l) \iff P\; [] \wedge\; (\forall h\; t.\, P\; (cons\; h\; t)) \tag{5.13}$$

$$\begin{aligned} &\vdash (\forall l.\, P\; 0\; l) \wedge \\ &\quad \underline{(\forall n.(\forall l.\, P\; n\; l) \implies P\; (s(n))\; [] \wedge (\forall h\; t.\, P\; (s(n))\; (cons\; h\; t)))} \\ &\quad \implies (\forall\, n\; l.\, P\; n\; l) \end{aligned} \tag{5.14}$$

5.4 Improving the user interface

5.4.1 Tracing proof attempts by Boyer-Moore

It is useful to track the proof attempts by the Boyer-Moore tool and to visualise the search tree as was shown in Fig. 5.3. As mentioned in Section 4.2.2, this is necessary so that we can detect when Boyer-Moore failed to prove a goal and come up with potential improvements. It also shows how induction steps are applied and what subgoals are proven, which gives a sketch of the proof by Boyer-Moore.

There are two main issues when tracking the information in diagrammatic form:

- Waterfalls are often stopped by the systems when running the Multi-waterfall model, due to the time out or the success of another waterfall in parallel (see Section 5.2). In this case, there is no time for the waterfall being stopped to record its progress.

- The waterfalls run asynchronously. Therefore, even though the progress records of stopped waterfalls are collected, it is difficult to organise them chronologically i.e. in the order in which these waterfalls were created.

To solve these issues, the following solutions are used:

1. Any subgoal is recorded before it is sent to a waterfall, and recorded again when it is proven.
   - This ensures that any subgoal being tried by a waterfall is recorded and labelled with the information about its position in the proof i.e. the order of the subgoal, which is obtained using a *time stamp*.
   - We can tell that a subgoal is proven if it is recorded twice.
2. All records are collected together in the main process, which will never be stopped.

Each time information is recorded about a subgoal, a *time stamp* is added of the form shown as (5.15), where *gid* and *wid* are the ids of the subgoal and waterfall respectively. For instance, a goal g_1 is recorded initially with time stamp $[(0,x)]$ where x is unknown, because it has not been sent to a waterfall. g_1 is then sent to multiple waterfalls. Assume that the second waterfall cannot prove g_1, induction is applied to it, and this generates two subgoals, these will have time stamps $[(0,1);(0,x)]$ and $[(0,1);(1,x)]$ respectively (assuming ids start at 0).

$$[(gid_0,wid_0);(gid_1,wid_1);...(gid_n,wid_n] \tag{5.15}$$

Each goal g is first recorded with $\{stamp;g\}$ before sending it to the waterfall, where $stamp$ is its time stamp. If g is proven, a second record will be created as $\{stamp;terms\}$, where $stamp$ is the same, but $terms$ is the list of subgoals proven within the waterfall. This is because a goal may be split into several subgoals, which are all proven within the waterfall. When processing these records, two of them with the same time stamp represent one proven goal, so are merged and labelled as "proven". An example of the processed records is shown in Fig 5.4 where # is used to separate the time stamp and the (sub)goal, -1 stands for the aforementioned x, and the variable for induction is also recorded. Note that the last subgoal was not proven, although it is recorded.

These records are then processed to a JSON file as shown in Fig 5.5. The time stamps

```
0,-1# Indvar:  Proven: ALL p (REVERSE xs) <=> ALL p xs
0,2;0,-1# Indvar: xs Proven: ALL p (REVERSE []) <=> ALL p []
0,2;1,-1# Indvar: xs Proven: (!p. ALL p (REVERSE a1) <=> ALL p a1)
          ==> (ALL p (REVERSE (CONS a0 a1)) <=> ALL p (CONS a0 a1))
0,2;1,2;0,-1#(!p. ALL p (REVERSE []) <=> ALL p []) ==> (ALL p (REVERSE [a0]) <=> ALL p [a0])
```

Figure 5.4: Records of (sub)goals

```
[{"node":
  {"name": "Indvar:  Proven: ALL p (REVERSE xs) <=> ALL p xs",
   "children": [
     {"name": "Waterfall:None",
      "children": [
        {"name": "Indvar: xs Proven: ALL p (REVERSE []) <=> ALL p []"},
        {"name": "Indvar: xs Proven: (!p. ALL p (REVERSE a1) <=> ALL p a1)\n
                  ==> (ALL p (REVERSE (CONS a0 a1)) <=> ALL p (CONS a0 a1))",
         "children": [
           {"name": "Waterfall:None",
            "children": [
              {"name": "(!p. ALL p (REVERSE []) <=> ALL p [])\n
                          ==> (ALL p (REVERSE [a0]) <=> ALL p [a0])"}]
           }]}]}]}, "name": "ALL_REVERSE", "tracking_id": 2806}]
```

Figure 5.5: JSON of (sub)goals

are turned into tree-structured data. There are two kinds of children nodes (i.e. *children* in the file): the ones that show all subgoals, and the others that show the different outcomes from running multiple waterfalls i.e. the node contain the *name* attribute "Waterfall:x" ($x = None$ means the waterfall that directly performs induction). This file is then be used for visualisation, as shown in Fig 4.7 of Section 4.2.2.

5.4.2 Generating proof scripts

We also wanted our Boyer-Moore tool to generate proof scripts for proven theorems. This is because:

- It takes significantly more resources (time and CPU threads) to prove the theorem using Boyer-Moore (which includes searching with multiple waterfalls) than loading any proof script that has been generated and saved.
- The proof scripts can be used to track dependencies for lemma selection.

In order to generate the proof script, information such as lemmas used for rewriting, induction variable selection, etc. need to be tracked. This can be achieved by modifying the waterfall heuristics to record the information used in the proof.

To store the Boyer-Moore steps, several data structures were used:

The data structure (5.16) used to store the steps from the waterfall heuristics keeps track of the following information: For basic heuristics, like the *Tautology heuristic*, only the heuristic name is required; For heuristics like *Holy Hammer*, extra information such as the tactic and the lemma names used for proof reconstruction are stored.

$$
\begin{aligned}
&type\ meta_heu = \\
&\qquad Taut(Tautology\ heuristic) \\
&\qquad |\ Clausal(Clausal\ form\ heuristic) \\
&\qquad |\ Setify(Setify\ heuristic) \\
&\qquad |\ Subst(Substitution\ heuristic) \\
&\qquad |\ Equality(Equality\ heuristic) \\
&\qquad |\ Gen(Generalization\ heuristic) \\
&\qquad |\ Nb\ of\ string\ list(Simplify\ heuristic\ with\ lemmas) \\
&\qquad |\ Hammer\ of\ string * stringlist(Holy\ Hammer\ with\ tactic\ and\ lemmas) \\
&\qquad |\ Auto\ of\ string(Automated\ procedures) \\
&\qquad |\ Inst(Induction\ related\ operation) \\
&\qquad |\ Exception
\end{aligned}
\tag{5.16}
$$

The main data structure (5.17) that can fully record Boyer-Moore proofs has the following characteristics: pfs stores the list of waterfall heuristics in the order that they were applied using the type that is defined in (5.16); *indvar* stores the list of induction variables; *sbg* recursively stores the proofs for all the subgoals.

$$
\begin{aligned}
type\ meta_proof = \{ & \\
& pfs : meta_heu\ list; \\
& indvar : term\ list; \\
& sbg : meta_proof\ list\}
\end{aligned}
\tag{5.17}
$$

Most proof steps can be directly recorded while running Boyer-Moore. One exception is the Simplify Heuristic where the subgoal is often rewritten with a large number of lemmas (e.g. 256), but only a few of them are actually required for a successful proof. In order to avoid an extremely long proof script with all 256 lemmas in each rewrite step and in order to more accurately record dependencies, we need to pick a minimal list of used rewrite lemmas. A compromise solution is to repeatedly remove lemmas for rewriting, comparing the result with the original one. If removing a lemma results in a different result, the lemma is added back. We note here that this is similar to the approach used in Isabelle's Sledgehammer when it tries to minimise the lemmas needed to reconstruct a proof found by one of the ATPs. In the future, a more efficient approach which modifies the tactic such as `REWRITE_CONV` should be used. Note that since there is a timeout restriction for Boyer-Moore multi-waterfalls, such solution would have to be done in a second run of Boyer-Moore. In that second run, we would only need to go through the successful waterfalls, which can be figured out by the time stamps, and there is no need to set a timeout, because we already know the proof will be successful.

The reconstruction of proof scripts replaces the proof steps tracked from Boyer-Moore with corresponding HOL Light tactics. An additional tactic `IND_MP_TAC` is added to generalise variables that are not used for induction (see Section 5.1.2), because there is no HOL Light tactic that has this effect. As mentioned in Section 3.3.3, lemma selection will suggest a specific conjunct of a conjunctive theorem, so a function `conj` is also added for specifying the conjunct needed for the proof. One example of a generated proof is shown in Fig. 7.2 (see Section 7.1.5).

5.5 Conclusion

We investigated a number of approaches towards improving Boyer-Moore model in this chapter. This involved changes to the waterfall and the heuristics within it. Multi-waterfalls were proposed as a means of unblocking the proof search using parallelism. An initial approach using lexicographic induction was implemented for the support of non-primitive recursive definitions. We also added some extra functions to Boyer-Moore to facilitate our investigation e.g. by generating proof scripts and novel tracing support for inductive proof attempts.

In the next chapter, machine learning techniques will be applied based on the improvements in this chapter.

Chapter 6

Combining Boyer-Moore with machine learning

In this chapter, we describe our approach to combining machine learning techniques with the Boyer-Moore model. We first look at possible ways of improving the machine learning for lemma selection in Section 6.1. We also examine how good *HOL(y)Hammer* is at proving inductive theorems.

6.1 Improving machine learning for lemma selection

Lemma selection is one of the key aspect of investigating the application of machine learning to theorem proving. It is also very important for the multi-waterfall model because it can help select intermediate lemmas for proving subgoals. In this section, we will briefly introduce our attempts at improving it.

Our first investigation was to try *logistic regression* (Walker and Duncan, 1967), which in some early experiments (outside the scope of this thesis) looked promising. However, its training time proved to be too time-consuming and was not investigated further.

	Total		Induction	
	Arithmetic	*List*(*core*)	*Arithmetic*	*List*(*core*)
Satallax	196	6	4	3
BM	167	58	7	39
First order ATP	379	5	19	0

Table 6.1: Performance of higher order ATP

6.2 Using *HOL(y)Hammer* to find inductive proofs

In order to gain an idea regarding the effectiveness of ATPs on inductive proofs, we decided to run them, along with *Satallax* (Brown, 2012), a recent winner of *the CADE ATP System Competition(CASC)* (Sutcliffe, 2016) for higher order problems, on some of our datasets.

The results are shown as Table 6.1. Neither first order ATPs nor Satallax did well on *List*(*core*), which contains mostly inductive problems. ATPs proved many more theorems than Boyer-Moore(BM) in *Arithmetic*, because this contains quite a lot of first order theorems. We also noticed that Satallax had worse performance in general than first order ATPs, but it did prove a few more inductive theorems.

According to the results, it is not likely that the inductive proofs can be found with only ATPs.

6.3 Adding lemma selection to waterfall

We now consider the application of machine learning inside Boyer-Moore to select intermediate lemmas for proving subgoals from induction. The settings we used for the experiments are introduced in Section 6.3.1, followed by a description of the approach developed in Section 6.3.2. Some changes that were motivated by our experiments are then mentioned in Section 6.3.3 and Section 6.3.4.

6.3.1 Development environment

In this section, the datasets mentioned in Section 5.1 are used to discover and fix the issues resulting from combining Boyer-Moore with lemma selection.

Our framework will only consider the incremental learning algorithm, namely *Sparse Naïve Bayes*, which can be incrementally trained and has been showed to be effective in *HOL(y)Hammer* (Kaliszyk and Urban, 2014) and Sledgehammer (Kühlwein et al., 2013).

However, although *HOL(y)Hammer* supports *Sparse Naïve Bayes* via SNoW, this was found to be defective with respect to its incremental learning, so we ported an optimised version from Isabelle *Mash*[1] and adapted it to HOL Light.

When we run multiple waterfalls in parallel, there will be a number of ATPs running simultaneously leaving few threads for the waterfalls, so we decided to use only two ATPs: Vampire 4.1[2] and Epar (a wrapper of E included in HOL(y)Hammer) (Urban, 2013). Z3, although included in *HOL(y)Hammer*, was not used because in our experiments, most theorems proven by it are covered by the other two.

We set the timeout for each waterfall to 30 seconds, which is a reasonable time that a user would wait for the system as well as the default timeout of Sledgehammer and HOL(y)Hammer (Kühlwein et al., 2013; Kaliszyk and Urban, 2014).

The number of lemmas to be sent to ATPs can be hard to figure out. A balance needs to be struck between choosing a value that is high enough to make it likely that all lemmas needed for the proof have been chosen and one that is low enough to prevent the ATPs from getting stuck in their search for a proof. Because of this tension, the number of proven theorems tends to increase with the number of lemmas sent to ATPs initially and then decrease, as was shown for *HOL(y)Hammer* (Kaliszyk and Urban, 2014). For our investigation, we also ran some experiments to determine a suitable value for how many lemmas to select. For this, we used *HOL(y)Hammer* with the *Jordan Curve Theorem* library[3]. In the *HOL(y)Hammer* source code, the available parameters for lemma selection were 64 and 128. In our early experiments, we found that our chosen ATPs could easily handle 128 chosen lemmas and prove more theorems than with 64.

[1] https://github.com/seL4/isabelle/blob/master/src/HOL/Tools/Sledgehammer/sledgehammer_mash.ML

[2] http://www.cs.miami.edu/ tptp/CASC/J8/

[3] https://github.com/jrh13/hol-light/tree/master/Jordan

Number of lemmas sent to ATPs	128	256	384
Number of proven theorems	1417	1564	1523

Table 6.2: Number of proven lemmas with different numbers of lemmas sent to ATPs

Therefore, we decided to set 128 as our base value and then test on multiples of 128 (i.e. 128, 256, and 384).

According to Table 6.2, the number of proven theorems was the highest when 256 lemmas were sent to ATPs. We could probably find a more precise value with further experiments, and different lemma selection methods may have different peak values. However, given that the number of proven theorems were actually quite close for the values above, and the optimal value also depends on the corpora for evaluation, we decided to set 256 as the value for all the methods in the current work.

Such parameters cannot be optimized globally as each goal may require different values (the user could tinker with the values in an interactive setting). We believe that the current settings are reasonable for the automated evaluation of our implementation, and further optimisations can be tested in future experiments.

In order to run multiple waterfalls in parallel, a multi-core machine was used with 2 *Intel(R) Xeon(R) CPU E5-2690 v2 @ 3.00GHz* (40 threads in total) with 64GB RAM. Note that the actual CPU load varies for different problems and is relatively low in most cases.

6.3.2 Lemma selection for internal and external tools

A straightforward way to apply lemma selection in the Boyer-Moore model is to select rewrite rules for the Simplify heuristic or, more generally, any heuristic that requires relevant lemmas.

The lemmas are usually used in the following two situations:

- As rewrite rules in the Simplify heuristic. A selection of suitable lemmas may directly lead to a proof by rewriting without further induction steps.
- For automatically tracking definitions for selecting the suitable induction variable (see Section 2.5.1). Although with the new dependency tracking method, all definitions are clearly marked, we noticed a decrease of the success rate while

pouring all definitions into Boyer-Moore. This is because Boyer-Moore uses all definitions as rewrite rules. In this case, all the functions will be recursively expanded so many lemmas involving them cannot be used.

The approaches to selecting these lemmas automatically are:

1. Train a classifier *C* on the proofs that are encountered before the current goal (see Section 3.3.5).

2. Run Boyer-Moore with *C*

    - At the beginning, *C* selects the definitions for the induction procedure.

    - In the heuristics that requires rewrite rules (e.g. simplify heuristic), *C* is also used for selecting relevant lemmas for the current goal/subgoal. Lemma selection is applied to different subgoals individually.

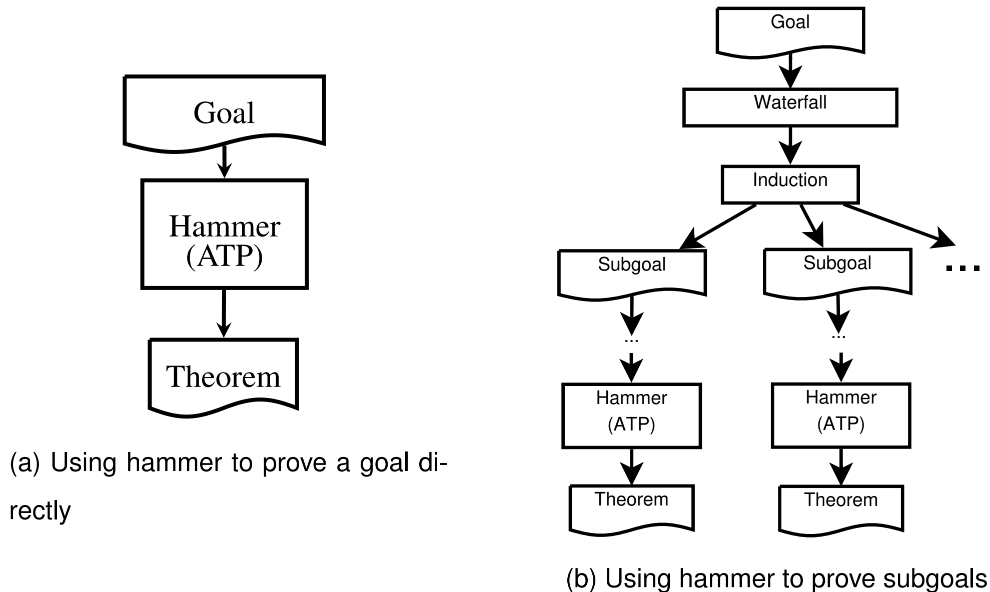
Note that these approaches are different from those in ACL2(ml). The difference in machine learning approaches (i.e. supervised vs unsupervised) and feature extraction have been explained in Section 3.3. In addition, we apply lemma selection on each subgoal independently, while ACL2(ml) generally applies its search only at the beginning on the whole goal. Such fine grained changes are possible thanks to the simplicity and accessibility of our HOL Light test bed (in contrast to the complicated structure of ACL2).

The main issue with lemma selection in this context is that the number of selected lemmas must be bounded. The larger the rewrite rule set, the more likely it is that the Simplify heuristic will loop. Selecting fewer lemmas means that key lemmas may be classified as 'not relevant enough' and not be selected.

Rewrite in simplify heuristic One way to mitigate the problem with the number of lemmas that can be tackled by the simplifier is by replacing the conditional `SIMP_-CONV` in the HOL Light Boyer-Moore implementation with the simpler `REWRITE_CONV`, which can deal with more rewrite rules. One experiment was used to evaluate its performance by sending different numbers of lemmas to it, as shown in Table 6.3. When testing with *List*(*core*), sending fewer lemmas (i.e. 64) allowed Boyer-Moore to prove more theorems, which indicated that more lemmas are slowing down `REWRITE_CONV`. However, with *Arithmetic*, sending 128 lemmas had better results, which means that some relevant lemmas were not selected within the top 64. Judging by the number of lemmas for selection in these two libraries, which is around 750 vs around 2800,

	List(core)	*Arithmetic*
Sending 128	47	72
Sending 64	56	62

Table 6.3: Using REWRITE_CONV with lemma selection



(a) Using hammer to prove a goal directly

(b) Using hammer to prove subgoals

Figure 6.1: Combining hammers in waterfalls

`REWRITE_CONV` is not suitable to support lemma selection when there are more lemmas for selection. The same problem was observed with `MESON` and `METIS` as they could not handle large sets of lemmas, and timed out. For that reason, `MESON` is currently used on its own (see Section 5.1.3).

In contrast, ATPs are good at handling large numbers of lemmas in more ways than just simplification. We take advantage of this by adding a modified version of HOL(y) Hammer (see Section 6.3.1) as a heuristic that can directly prove a (sub)goal. We call this heuristic the *ATP heuristic*.

There could be two ways to prove inductive goals with *HOL(y)Hammer*:

- The goal is directly proven by an ATP through *HOL(y)Hammer*, as is shown in Fig 6.1a, without induction, or
- After induction is applied, the subgoals are proven by hammers, as is shown in Fig 6.1b.

Two waterfalls in parallel We can fit these two strategies into the multi-waterfall

	List(*core*)	*List*(*hilbert*)
two-waterfall	32.99%	57.39%
three-waterfall	57.73%	63.48%

Table 6.4: Success rates comparison between two and three-waterfall model

model, and run two waterfalls in parallel: one with and one without *HOL(y)Hammer*. The simplify heuristic is removed in the waterfall containing hammer, because simplification is already incorporated in the ATP.

6.3.3 Direct induction

It is quite common in manual inductive proofs for the reasoning to begin with induction before any simplification or other proof steps. In the Boyer-Moore model such proofs may get stuck at the ATP or Simplify heuristics and eventually time out and fail, whereas applying induction directly could help unlock the proof. Moreover, some goals in our initial experiments were being rewritten to a form that caused Boyer-Moore to either choose a wrong variable for induction or have more complicated subgoals after induction (for example because complex definitions were expanded unnecessarily) and fail.

For these reasons, we constructed a new configuration of the waterfall with no heuristics, but instead induction is applied directly. Including this in our multi-waterfall model (see Section 5.2) enables proofs where induction is applied directly and another waterfall that first uses heuristics to try to prove the subgoals.

To investigate the necessity of this third waterfall, a comparison between two and three-waterfall setting was done, and the results are shown in Table 6.4. The detailed settings are shown in Table 6.6 in Section 6.3.5. A big improvement can be noticed according to the table, and in *List*(*core*), the success rate with two-waterfalls was worse than the Boyer-Moore with our initial changes from Section 5.1.5 (see Table 6.7), therefore this third waterfall is necessary.

Note that the waterfall containing the Simplify heuristic is still useful in some examples, because sometimes a goal cannot be proven immediately by *HOL(y)Hammer*, but a simplified version may be proven after induction. The simplify heuristic is also useful after induction has been applied.

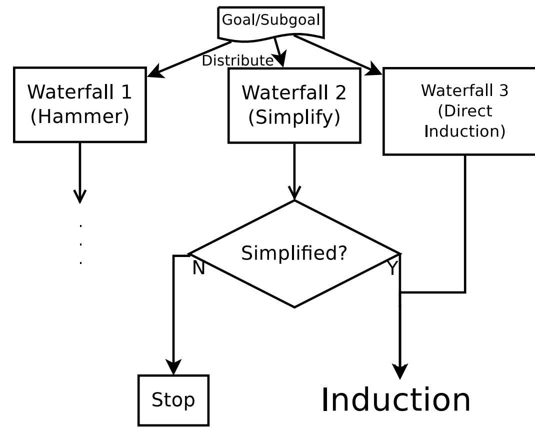


Figure 6.2: Optimised 3 waterfalls

However, there can be a redundancy if the waterfall with the simplifier fails to process the goal. For this reason, we stop the waterfall for simplification if the goal remains unchanged. Induction is only applied if the goal is actually simplified in the waterfall. This is shown as Fig 6.2, which is an example of a 3-waterfall setting, including waterfalls for simplification and direct induction.

6.3.4 Fixing the translation to TPTP

While *HOL(y)Hammer* provides a translation tool from a HOL Light goal to TPTP format, it still has limitations. We addressed and tried to fix the following issues, found during our experiment with the Boyer-Moore Model:

Utilise the TPTP built-in constants and functions TPTP has its own syntax $true and $false for the constants T and F in HOL Light (see Section 2.1.1). Translating these constants to TPTP syntax may help ATPs because they do not need lemmas like $(x \iff T) \iff x$ to eliminate such constants. If lemma selection fails to find such lemmas, the proofs can still be found by ATPs. Ifthenelse expressions can also be converted to the corresponding TPTP format, for the same reason. This is helpful with definitions like BUTLAST (6.1).

$$\begin{aligned} &BUTLAST\ [] = [] \wedge \\ &BUTLAST\ (CONS\ h\ t) = if\ t = []\ then\ []\ else\ CONS\ h\ (BUTLAST\ t) \end{aligned} \tag{6.1}$$

Translate equality There are two kinds of equalities in both TPTP and HOL Light syntax, one of which is term equality and the other is logical equality, and

```
let must_pred tm =
  is_forall tm || is_exists tm || is_conj tm || is_disj tm or
  is_imp tm || is_eq tm || is_neg tm || is_abs tm;;
```

Figure 6.3: Function must_pred

	Number of proven
No translation fix	68
With translation fix	71
Common	63

Table 6.5: Performance comparison with fixed TPTP translation

will be referred to as "=" and " $\iff$ " here. The original translation module in *HOL(y)Hammer* prefers "=" to " $\iff$ ". It has a function *must_pred* shown as Fig 6.3 to filter out non-atomic formulae, and " $\iff$ " is applied only when an argument at one side is a non-atomic formula. This approach can reduce the number of clauses during resolution (see Section 2.2) and thus reduce the proof search time. However, it is necessary to use " $\iff$ " to translate T and F correctly.

An experiment for this fix using multi-waterfall model (three waterfalls) on *List*(*hilbert*) is shown in Table 6.5. There is an improvement after changing the translation. Note that however that some of the originally proven theorems are now missing.

6.3.5 Intermediate evaluation

We compared the Boyer-Moore multi-waterfall with lemma selection and the version with our initial fix. A three-waterfall was used, with its settings shown in Table 6.6 (see Section 7.1). More specifically, we used a waterfall with the ATP heuristic, a standard waterfall with the Simplify and MESON heuristics, and a waterfall with direct induction (see Section 6.3.3). The results are shown as Table 6.7. The improvement after applying these approaches can be observed, particularly in *List*(*hilbert*). This indicates that the original Boyer-Moore's built-in lemmas are enough to prove theorems in *List*(*core*), so only small benefits can be gained. The significant improvements for *List*(*hilbert*) demonstrate that lemma selection is effective for corpora that contain more difficult theorems and a larger variety of useful lemmas.

Heuristic	Waterfall 1	Waterfall 2	Waterfall 3
Simplify		$\times$	
MESON		$\times$	
Other Heuristics	$\times$	$\times$	
HOL(y)Hammer	$\times$		
Induction	$\times$	$\times$	$\times$

Table 6.6: Heuristic settings for three waterfalls

	Total		Induction	
	List(*core*)	*List*(*hilbert*)	*List*(*core*)	*List*(*hilbert*)
Initial	52.58%	20.00%	45.21%	17.50%
Multi-waterfall	57.73%	63.48%	46.58%	62.50%

Table 6.7: Success rates comparison after combining machine learning and multi-waterfall to Boyer-Moore

6.4 Machine learning for selecting induction variable

Induction variable selection, i.e. selecting suitable variables, is an important step of induction. We decided to investigate the application of machine learning methods to the selection of induction variables because recursion analysis only suggests induction variables based on primitive recursive definitions. Advanced techniques, e.g. Lexicographic Relation Induction (Boyer and Moore, 1979) have not been implemented in our Boyer-Moore implementation (Note that this investigation was done before some of the methods described in Section 5.3 were implemented). The current induction procedure is insufficient sometimes, particularly when the definitions in *List*(*hilbert*) are non-primitive recursive e.g. 5.8 in Section 5.3. The induction procedure cannot figure out the recursive argument for such definitions. Therefore, it might be useful to learn induction variable selection from existing inductive proofs.

In addition, we are also curious to know whether it is possible to learn patterns and predict induction variables from induction procedures in manual proofs and how a machine learning approach compared to the existing induction heuristics.

6.4.1 Methodology

The learning approach for induction variable selection follows the same framework as lemma selection, i.e. processing the manual proofs to get *features* and *class labels*, training on them, and predicting for new items (Section 3.1). However, the *features* and *class labels* need to be adjusted.

6.4.1.1 Induction variable tracking

Tracking induction variables is different from dependency tracking because in lemma selection, all lemmas learnt from previous proofs are available for a new proof. However, the induction variable in previous proofs may not exist in a new statement. We can only select the induction variable within the free variables in the statement to be proven.

Moreover, the total number of variables in a statement is not fixed, so it is difficult to make a prediction like "the ith variable is for induction", because i can always exceed the range of prediction. For instance, in the proofs for training, if we train with the proofs whose statements contain at most three variables, when trying to prove a new goal whose statement has five variables, the fourth and fifth variable will never be considered. Thus, the approach described in (Lindhé and Logren, 2016), for instance, which is based on such an indexing idea would not be suitable for our work.

We suggest a new approach of tracking induction variable, where variables are indexed by their positions in functions i.e. argument positions in each non-trivial subterm of the statement. For instance, in the statement $m+(n+p) = (m+n)+p$, m is indexed by its position in "$m+n$", "$m+(n+p)$" and "$(m+n)+p$". Below are the detailed approaches:

First, training data is gathered from manual proofs. We look for induction steps in the proof. In each induction step, the goal statement looks like "$\forall v_i.\ P\ v_1\ v_2 ... v_n$" where the outermost variable v_i is used for induction:

1. A record is generated for each variable $v_x \in \{v_1\ v_2 ... v_n\}$ that is in recursive type (i.e. suitable for induction), so n records are generated when there are n such variables.

2. Each record is in the form $(s, v_x, label)$ where s is the goal statement, and $label$

is T when $v_x = v_i$, otherwise F i.e. *label* is true (T) or false (F) according to the claim "v_x is the induction variable for s".

After this process, the manual proofs are converted to triples which consist of $(statement, variable, label)$. Note that the $statement, variable$ pairs will be used to generate features and replaced by $feature$ (see Section 6.4.1.3), so the training datasets will contain instances like $(feature, label)$ where *label* is a binary i.e. a binary classification problem.

6.4.1.2 Induction variable predicting

We then train on the dataset obtained in Section 6.4.1.1 and get a classifier C, which can be used to predict the probability $P(label = T) = C(feature)$. For a goal statement g, all variables in the goal can be selected based on their probability of being used for induction with the following steps:

1. Get all free variables **FVs** from g that have recursive type, assuming there are n such variables.

2. Generate $feature_i$ for each $x_i \in$ **FVs** with the approach in Section 6.4.1.3 so that n features are generated.

3. Compute $P(label_i = T) = C(feature_i)$

4. Pick x_i with the highest $P(label_i = T)$

6.4.1.3 Feature extraction

We now introduce our approach for generating features. This process is based on the feature extraction for lemma selection. It generates features shared globally, from the local induction information represented as $(statement, variable)$ pairs, which will be referred to as s, v for short. The procedure is:

1. Get the free variables **FVs** with recursive type from s, i.e. all s, v generated from the same statements share the same **FVs**

2. Track the features from s in the same way as for lemma selection (see Section 3.3.2), except that for the terms or subterms that contain x_i in **FVs**, whose type is t:

- If $x_i = v$, normalise (i.e. rename) it to “Bt”
- Otherwise, normalise x_i to “At” i.e. the same way as lemma selection

Here is an example for the conjecture s: “$m + SUC\ n = SUC\ (m+n)$” m was used for induction in the manual proof:

First, there are two inductive variable m and n, so the two records generated are: (s,m,T) and (s,n,F).

Then the generated features are:

Variable	Feature	Label
m	“Bnum + SUC Anum”, “SUC (Bnum + Anum)”,...	T
n	“Anum + SUC Bnum”, “SUC (Anum + Bnum)”,...	F

Note that only “Feature” and “Label” are used for training, and the “Variable” column is just used to show the relation between variables and data records.

Regarding the evaluation, this approach has a similar issue as lemma selection. When trying to select the induction variable for a goal i.e. a theorem for testing, only the inductive proofs before it (i.e. the proofs that are “known” to the theorem) are used for training. For this reason, each record is labelled with the id of the theorem whose proof it is tracked from. When training, only the data with ids smaller than the test theorem are used for training.

6.4.1.4 Synthetic proof

A limitation of this approach above is that the definition cannot be learnt before it is used in a proof, which is also mentioned in Section 7.2.2. We try to solve this by adding a “synthetic” proof for training:

1. Whenever a new recursive function f is defined, we figure out its recursion argument v_i by recursion analysis.
2. A statement “$\forall v_i.\ f\ v_1\ v_2 ... v_n = f\ v_1\ v_2 ... v_n$” is added for machine learning, which indicates v_i is used for induction. This is a synthetic proof that has the same form as the tracked statement in Section 6.4.1.1. For instance, the second argument is at the recursion position for addition, so its synthetic proof is $\forall v_1.\ v_0 + v_1 = v_0 + v_1$.

This approach can be viewed as the incorporation of recursion analysis. However, the improvement was small: only one additional theorem `BREAK_ACC_APPEND_conjunct1` (6.2) was proven after this change for the dataset *List*(*hilbert*), because the definition of `BREAK_ACC` (*BREAK_ACC p l acc* is a primitive function recurs on *l*) was learnt immediately after the definition and before testing this theorem. Without this knowledge, *ys* is selected according to the definition of `APPEND`. This theorem was only proven when the dependencies tracked from its manual proof were provided to Boyer-Moore multi-waterfalls, so the results were not updated in Section 7.2.2.

$$\forall p\, xs\, y\, ys.\ SND\ (BREAK_ACC\ p\ xs\ (APPEND\ ys\ [y])) = SND\ (BREAK_ACC\ p\ xs\ ys) \tag{6.2}$$

6.5 Conclusion

The application of machine learning to inductive theorem proving was discussed in this chapter. The attempt to use *HOL(y)Hammer* with a higher-order ATP did not bring much success. Lemma selection was then integrated in Boyer-Moore model with multi-waterfalls, which helped to unblock the proof search. An approach to using machine learning to select induction variables was also described.

In the next chapter, we evaluate these methods on our various corpora.

Chapter 7

Evaluation

In this chapter, our approaches from Chapter 5 and Chapter 6 are evaluated. First, the *multi-waterfall* model with lemma selection integrated is evaluated in Section 7.1 and the machine learning approach for selecting induction variable is in Section 7.2. After that, in Section 7.3, the implementation of selecting induction variable for goals involving non-primitive recursive definitions i.e. *Lexicographic induction* is tested and compared with the existing approach in Boyer-Moore as well as the machine learning approach. Finally, the generality of our approach is discussed in Section 7.4.

7.1 Multi-waterfall model

We evaluate our Boyer-Moore Multi-waterfall with machine learning for lemma selection in this section.

7.1.1 Choice of datasets

In order to evaluate our work, we use proven theorems about recursively-defined data types, which are shown in Section 4.3.1. We note here that the *IsaPlanner benchmark* is unsuitable in our case for the following reasons:

1. Many of the definitions use case-expressions, which are not currently supported by HOL Light. After removing case-expression in the translation from Isabelle to HOL Light, many test theorems are part of the recursive definitions of the

corresponding functions and so can be proven trivially, which also happens in the benchmark[1].

2. In the evaluation of HipSpec, 67 theorems were proven without using any auxiliary lemmas, and more than 10 were proven using only rewriting (Claessen et al., 2012). Therefore, lemma selection would not have any impact in these examples.

7.1.2 Experiments

In practice, many users create inductive proofs which apply induction at the beginning followed by simplifications using rewriting to the subgoals. This strategy generally works well in a number of interactive theorem provers may be very useful especially in some theorem provers, so is preferred to dedicated automated proof system for induction such as Boyer-Moore.

In order to show that the Boyer-Moore model is a good starting point for inductive theorem proving, a comparison between it and a simple "induction then rewriting" proof strategy was therefore made. Such a strategy is commonly used in manual proofs for a large number of (relatively simple) inductive theorems. We will refer to it as *Ind simp*. Since this strategy has no heuristic about choosing induction variable, the induction procedure from Boyer-Moore was applied.

We then performed the following experiments using the methods described in Chapter 5:

1. *Original*: Running the original Boyer-Moore implementation as a baseline.
2. *Initial*: Running Boyer-Moore with the changes from Section 5.1.
3. *Multi-waterfall*: Running the multi-waterfall model described in Section 5.2, using the three waterfalls shown in Table 6.6, see Section 6.3.5.
4. *ATP*: The combination of lemma selection with the ATP heuristic *outside* Boyer-Moore, i.e. without induction, so that we evaluate and compare the performance of ATPs on inductive proofs independently.

Note that in the experiments without lemma selection (*Ind simp*, *Original* and *Initial*),

[1]https://github.com/tip-org/benchmarks/tree/master/original/isaplanner

	List(*core*)	*List*(*hilbert*)	*Poly*
Ind simp	24.74%	13.04%	8.94%
Original	41.24%	14.78%	13.01%

Table 7.1: Success rate of *Ind simp* and the original Boyer-Moore.

the built-in rewrite rules and definitions in Boyer-Moore are used. We also use the definitions from our *List*(*hilbert*) and *Poly* datasets.

7.1.3 Metrics

For each dataset, we compute the success rate as n/m where n is the number of theorems proven and m is the number of theorems in it. We also compute the *inductive* success rate in the same way for the subset of inductive theorems in each dataset. All results are given as percentages.

7.1.4 Results

The comparison between the original implementation of Boyer-Moore and *Ind simp* is shown in Table 7.1. *Ind simp* has a lower success rate. Note that the *Ind simp* evaluated is already enhanced with the induction procedure from Boyer-Moore, so this approach can be seen to be generally weaker than Boyer-Moore. Boyer-Moore only failed on two theorems proven by *Ind simp* mainly due to the issue with the CNF heuristic mentioned in Section 5.1. Therefore, this provides a state of the art to build on and a higher bar over which to show improvement.

The results of the experiments with the evaluation data set *poly* are shown in Table 7.2, which are consistent with the experiments on our development sets (see Section 5.1.5 and Section 6.3.5). *Initial* generally outperformed *Original*, which was still able to prove some theorems that *Initial* failed on though, due to the failure of some heuristics that rely on CNF. Similar to *List*(*hilbert*), *Multi-waterfall* leads to major improvements (see Section 6.3.5), because it contains many difficult theorems and quite a few useful lemmas.

Table 7.3 is the comparison between the Boyer-Moore *Multi-waterfall* and ATPs. ATPs performed relatively poorly on inductive theorems (which significantly affected their

	Total	Induction
Original	13.01%	11.94%
Initial	14.63%	13.43%
Multi-waterfall	40.65%	37.31%

Table 7.2: Success rates of the different configurations of Boyer-Moore for *Poly*

	Total			Induction		
	List(*core*)	*List*(*hilbert*)	*Poly*	*List*(*core*)	*List*(*hilbert*)	*Poly*
Multi-waterfall	57.73%	63.48%	40.65%	46.58%	62.50%	37.31%
ATP	25.77%	36.52%	24.39%	5.48%	30.00%	10.45%

Table 7.3: Success rates comparison between Boyer-Moore *Multi-waterfall* and ATP

total success rate as well). However, ATPs had a relatively good success rate on *List*(*hilbert*). This shows that with appropriate lemma selection, ATPs can sometimes be useful for problems that were inductively proved manually, indicating that ultimately these are not inductive theorems.

Fig. 7.1 shows a Venn diagram representation of the theorems proven by *Initial*, *Multi-waterfall* and *ATP*, demonstrating the percentage of theorems that could only be proven by some of the methods, but not the others. *Multi-waterfall* could prove many theorems that none of the other methods could. This reveals the enhanced potential of combining lemma selection and Boyer-Moore in a multi-pronged, parallel proof strategy.

In *List*(*core*), *Multi-waterfall* failed to prove some theorems that were proven by *Initial*. This is mainly due to the lack of conditional rewriting (see Section 6.3). Moreover, some theorems were proven by *ATP* but not *Multi-waterfall*, because *Multi-waterfall* requires *quantifier-free* goals as input and the quantifiers in each test statement are removed. This affects how the goals are translated to the ATP format variables (they are translated to constants by *HOL(y)Hammer*), and thus impacts the performance of ATPs integrated in Boyer-Moore.

Examining failed proofs in *Multi-waterfall*, we discovered that some theorems are proven with special tactics: a tactic can eliminate conditional expressions e.g. (7.10); Or a case split on a list l can produce two subgoals, one with $l = []$ and the other $\neg l = []$; Or a tactic can specialise the variable in the goal. In some cases the wrong variable was chosen for induction, particularly when two or more induction steps are used in

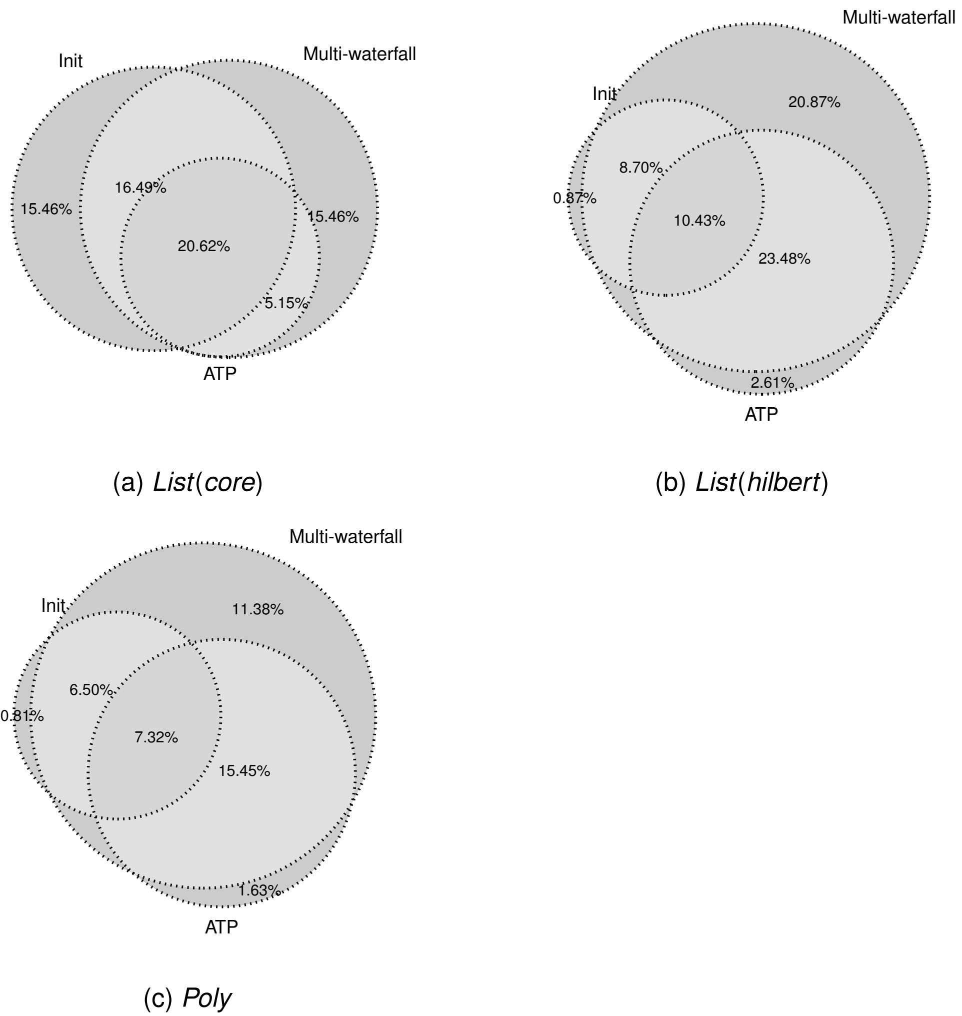


Figure 7.1: Coverage of proven theorems by the different methods in Table 7.2

	Total	Induction
Multi-waterfall (lemma selection)	40.65%	37.31%
Multi-waterfall (tracked dependencies)	51.2%	49.3%

Table 7.4: Success rate comparison of using different lemmas for *Poly*

a proof which is very common in *List*(*hilbert*) and *Poly* and there are non-primitive recursive definitions in *List*(*hilbert*) (see Section 4.3.1).

Other failed proofs can be attributed to lemma selection issues. For instance, if we give Boyer-Moore multi-waterfall directly with the dependencies tracked from manual proofs, the success rate can be further improved. The results for *Poly* is shown in Table 7.4. Apart from missing key lemmas, another issue is that irrelevant lemmas slow down ATPs (see the example of `LENGTH_REVERSE` in Section 7.1.5).

7.1.5 Examples

An inductive theorem proven by *Multi-waterfall* is `DROP_DROP` from *List*(*hilbert*) shown in Fig. 7.2. It is worth comparing the manual proof to the one generated by Boyer-Moore. With the push of a button, a theorem with a complex manual proof containing three induction steps can be proven by *Multi-waterfall* automatically in only two induction steps. The corresponding proof script for the new proof is automatically generated and verified in HOL Light. Also note that the ATPs used with HOL(y)Hammer were unable to find a proof by themselves, neither when supplied with the same lemmas used in *Multi-waterfall* nor with machine-learned selection of 256 lemmas.

An example of a case where there was a failure to find an automatic proof is for the `LENGTH_REVERSE` theorem shown in Figure 7.3. It has a short manual proof with only one induction step and was proven by *Initial*, but not by *Multi-waterfall*. Further investigation showed that when trying to prove a particular subgoal, although lemma selection included the 6 lemmas that were sufficient for the proof, ATPs still failed to find it (even after being allowed to run for 60 seconds, i.e. double the time). In our later experiments, a list of 13 theorems (including definitions and common rewrite rules for lists) can easily prove many subgoals when used on their own, but not as part of a large selection, see Section 7.3.4. This shows that a small group of carefully picked lemmas can be more effective than a large number of automatically selected lemmas.

DROP_DROP: $\forall n, m, xs$: DROP $(n+m)$=DROP n (DROP m xs)

Manual proof:

```
INDUCT_TAC THEN REWRITE_TAC [ADD_CLAUSES;DROP]
   THEN INDUCT_TAC THEN ASM_REWRITE_TAC [LENGTH;ADD_CLAUSES;DROP]
   THEN LIST_INDUCT_TAC THEN ASM_REWRITE_TAC [LENGTH;ADD_CLAUSES;DROP]
   THEN REWRITE_TAC [GSYM ADD] THEN ASM_REWRITE_TAC [DROP;ADD_CLAUSES]
```

Proof generated by Boyer-Moore *Multi-waterfall*:

```
REPEAT GEN_TAC THEN REWRITE_TAC[conj 0 ADD_AC] THEN
IND_MP_TAC [`xs:(a)list`] list_INDUCT THEN CONJ_TAC THEN
CONV_TAC (REPEATC (DEPTH_FORALL_CONV RIGHT_IMP_FORALL_CONV)) THEN
(REPEAT GEN_TAC) THENL [REWRITE_TAC[conj 1 DROP];
IND_MP_TAC [`m:num`] num_INDUCTION THEN CONJ_TAC THEN
CONV_TAC (REPEATC (DEPTH_FORALL_CONV RIGHT_IMP_FORALL_CONV)) THEN
(REPEAT GEN_TAC) THENL [REWRITE_TAC [conj 0 DROP;conj 0 ADD];
SIMP_TAC[conj 1 ADD;conj 2 DROP];];]
```

Figure 7.2: User and Boyer-Moore proofs for DROP_DROP

LENGTH_REVERSE: $\forall xs$. LENGTH (REVERSE xs) = LENGTH xs

Manual proof:

```
LIST_INDUCT_TAC THEN ASM_REWRITE_TAC
   [LENGTH;REVERSE;LENGTH_APPEND] THEN ARITH_TAC
```

Figure 7.3: User proof for LENGTH_REVERSE

This explains why *Multi-waterfall* failed to prove some theorems that *Initial* proved.

7.1.6 Proof metrics

We apply the proof metrics from Section 4.4.2 to the results in Section 7.1.4. According to our initial experiments, the *I* and *N* metrics seem largely undifferentiable, so only the *I* metric is used. The average number of nodes is shown in Fig 7.4. *Multi-waterfall* proves theorems with more nodes in proofs than the *ATPs* for *List*(*core*) and *Poly*, which means it has proven more complicated theorems under our metrics. *All* stands for the average number of nodes in the manual proofs of all the theorems in each corpus, which is still higher than *Multi-waterfall*, particularly in *Poly*. This is because more than ten proofs in *Poly* have a hundred or more nodes and were not proven

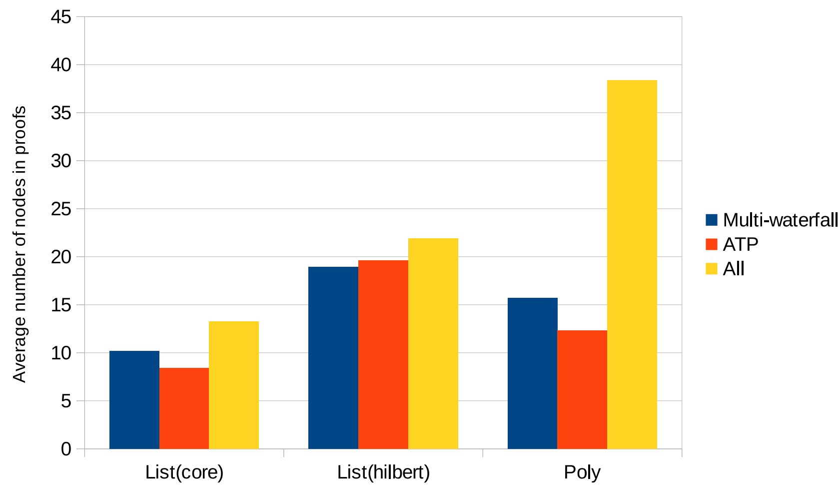
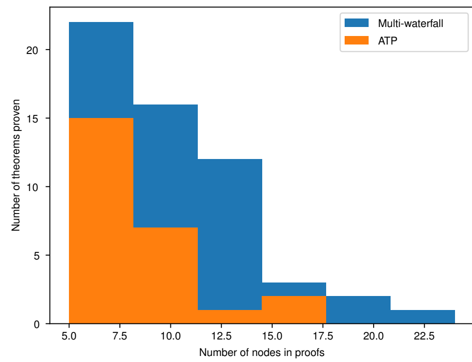


Figure 7.4: Average proof size of theorems by different methods in Table 7.2

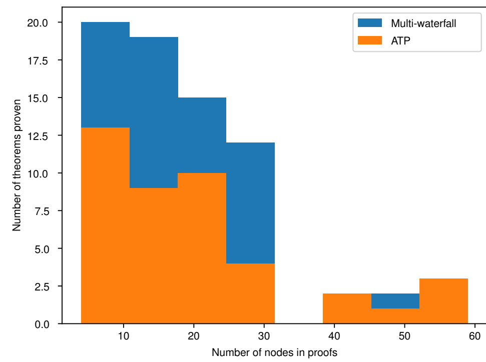
by either of the systems. As expected, this is an indication that automated systems are still unable to accomplish proofs with high complexity at the level that humans can.

In order to investigate why *ATPs* find proofs containing more nodes in *List*(*hilbert*), consider the histograms in Fig 7.5. These show the number of theorems proved by the two methods, with different proof sizes (Note that *ATP* is fully covered by *Multi-waterfall* and is put in front, so *Multi-waterfall* cannot be seen when they have the identical value). According to Fig 7.5, *Multi-waterfall* has proven more theorems with more nodes in manual proofs (i.e. more than average) in *List*(*core*) and *Poly*. In *List*(*hilbert*), *Multi-waterfall* proves more theorems with relatively fewer nodes in proofs. For instance, many proofs are like "LIST_INDUCT_TAC THEN REWRITE_-TAC[...]", which are very short, but *ATPs* could not prove. On the other hand, some theorems with a lot of nodes in proofs were proven by hammers, giving the *ATPs* a higher average number of nodes for proofs.

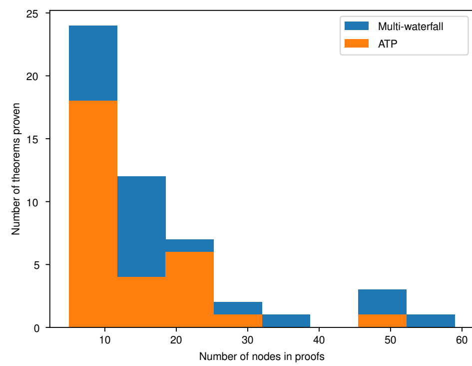
Our metrics failed to demonstrate the complexity of some proofs. First, though Boyer-Moore cannot sometimes find proofs containing a lot of nodes, the proofs that have fewer nodes but are inductive can still be challenging. In addition, sometimes the theorems with more nodes in manual proofs can be proven by *ATP*, because they can find alternative proofs, which means they may not be as difficult as their proofs indicated. Despite these limitations, these metrics reflected the complexity of most manual proofs, so we believe they are good enough in the general case.



(a) *List*(*core*)



(b) *List*(*hilbert*)



(c) *Poly*

Figure 7.5: Proof size of theorems by different methods in Table 7.2

List(*hilbert*)	Proven	Proven (Ind)
IndMl	72	49
RecAna	73	50

Poly	Proven	Proven (Ind)
IndMl	48	23
RecAna	50	25

Table 7.5: Number of proven theorems

7.2 Selecting induction variables with machine learning

In this section, the performance of using machine learning methods to select induction variables is evaluated.

7.2.1 Experiment settings

We compare recursion analysis, referred to as *RecAna*, which is originally used by Boyer-Moore with our machine learning approach, *IndMl*.

The *Multi-waterfall* setting in Section 7.1.2 is used. *List*(*hilbert*) and *Poly* provide the test data (see Table 4.5 in Section 4.3), because they contain many multi-step induction proofs.

7.2.2 Results

The results, shown in Table 7.5, are very close, so the number of proven theorems is used instead of the percentage. According to the results, *IndMl* proved one and two fewer theorems in *List*(*hilbert*) and *Poly* respectively. After further investigation, we found that the two approaches generally proved the same theorems (71 and 48 in common respectively).

To have an in-depth comparison between the two approaches for selecting induction variables, eight theorems, on which the two methods had different selection of induction variables, are examined. The number of induction steps used in each proof is

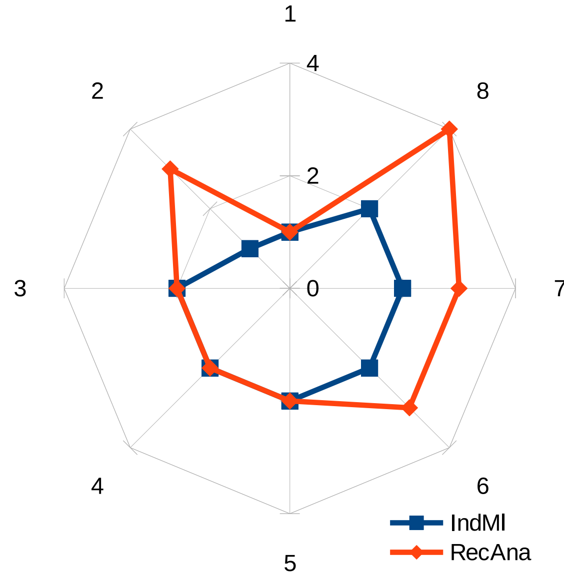


Figure 7.6: Comparison of induction steps between *IndMl* and *RecAna* for *List*(*hilbert*)

shown in Fig 7.6 for *List*(*hilbert*). *IndMl* always used equal (in five proofs) or fewer (in three proofs) induction steps on the same theorem and use no more than two induction steps on all proofs, which indicates that it has a better selection of induction variables. There are 32 theorems where both methods have chosen identical induction variable(s), which also shows that it is possible to have a heuristic similar to recursion analysis by learning from human proofs.

The theorem `IS_PREFIX_OF_EXISTS_APPEND` shown as (7.1a), i.e. if *xs* is the prefix of *ys*, there exists a list *zs* so that *ys* is the result of appending *xs* and *zs* was proven only with *Indml*. This was because the definition (7.1b) of *IS_PREFIX_OF* is non-primitive recursive, *RecAna* could not work properly and picked *xs* for induction repeatedly. *Indml* had induction on *xs* and *ys* in succession (by learning from an early proof) and proved it.

$$IS_PREFIX_OF\ xs\ ys \iff (\exists zs.\ APPEND\ xs\ zs = ys) \tag{7.1a}$$

$$\begin{aligned}
&(\forall ys.\ IS_PREFIX_OF\ []\ ys \iff T) \land \\
&(\forall x\ xs.\ IS_PREFIX_OF\ (x :: xs)[] \iff F \land \\
&(\forall x\ y\ xs\ ys.\ IS_PREFIX_OF\ (x :: xs)\ (x :: ys) \iff \\
&\qquad x = y \land IS_PREFIX_OF\ xs\ ys
\end{aligned} \tag{7.1b}$$

One of the theorems that *IndMl* failed to prove (but proved by *RecAna*) is `TAKE_TAKE`

with the subgoal (7.2). *IndMl* erroneously chose m for induction. However, given the definition of addition, n should have been chosen. This is because in the training data (i.e. the HOL Light proofs), m is used slightly more often for induction than n (15 vs 14). This means the machine learning approach will select m for induction when encountering the term $n+m$, if there is no other information to help the selection (e.g. m and n do not occur elsewhere).

$$\begin{aligned}(\forall m\, n.\, TAKE\ (n+m)\ a_1 = APPEND\ (TAKE\ n\ a1)\ (TAKE\ m\ (DROP\ n\ a_1)))\\ \Longrightarrow TAKE\ (n+m)\ (CONS\ a_0\ a_1) =\\ APPEND\ (TAKE\ n\ (CONS\ a_0\ a_1))\ (TAKE\ m\ (DROP\ n\ (CONS\ a_0\ a_1)))\end{aligned} \tag{7.2}$$

IndMl also failed to prove `APPEND_EQ` (7.3) i.e. if two lists xs and us has the same length, the outcomes of appending them with ys and vs respectively are equal, if and only if $xs = us$ and $ys = vs$. *RecAna* applied induction on us and xs in succession, while *IndMl* did this in reverse order i.e. xs and then us, which led to a subgoal essentially the same as *RecAna*. However, ATPs failed to find a proof with all required lemmas selected. The failure is due to irrelevant lemmas that slow down the ATPs rather than the selection of the induction variable.

$$\begin{aligned}LENGTH\ xs = LENGTH\ us \Longrightarrow\\ (APPEND\ xs\ ys = APPEND\ us\ vs \iff xs = us \wedge ys = vs)\end{aligned} \tag{7.3}$$

In *Poly*, the two approaches almost always select the same induction variables. One exception is `POLY_ADD_SYM`: $\forall x\ y.\ x++y = y++x$ (++ is the addition of lists i.e. the commutativity of list addition), where the positions of the arguments are symmetric, so it does not matter whether we start with x or y.

IndMl also failed to prove the theorem `NOT_POLY_MUL_NIL` (7.4) (the product of two non-zero polynomials is non-zero) where ** is the multiplication operation for polynomials . *IndMl* made the wrong choice between p_2 and a_1 in the subgoals after performing induction on p_1 twice as shown by (7.5) where ## is the scalar multiplication for a single number and a list. According to the definitions of ## and **, p_2 is in the recursive position for ## (though non-recursive position for **), and is the only choice for *RecAna*.

However, *IndMl* found the feature "B(real)list ** A(real)list" (and "A(real)list** B(real) list", because the variable considered for induction is normalised to "B" (see Section 6.4.1.3)) from the term "$a_1 ** p_2$" in the induction hypothesis. The variable a_1 is at the recursive position in "$a_1 ** p_2$", so *IndMl* suggested that a_1 is in a "better" choice than p_2 based on previous proofs. Conversely, *RecAna* only looks into the induction conclusion i.e. the subterms without quantifiers. This reveals a flaw in the current version of *IndMl*, namely that the (string) features tracked from induction hypothesis are treated equally as those in the conclusion. In future work, different strings should be assigned to the features to distinguish them.

$$\forall p_1\ p_2.\ \neg(p_1 = []) \wedge \neg(p_2 = []) \implies \neg((p_1 ** p_2) = []) \tag{7.4}$$

$$\begin{aligned}
&(\forall a_0\ p_2.\ (\forall p_2.\ \neg(a_1 = []) \wedge \neg(p_2 = []) \implies \neg(a_1 ** p_2 = [])) \implies \\
&\quad (if\ a_1 = []\ then\ a_0\ \#\#\ p_2\ else \\
&\quad\quad a_0\ \#\#\ p_2\ ++\ CONS\ (\&_0)\ (a_1\ **\ p_2)) = [] \implies p_2 = []) \\
&\implies (\forall p_2. \neg(CONS\ a_0\ a_1 = []) \wedge \neg(p_2 = []) \implies \neg(CONS\ a_0\ a_1 ** p_2 = [])) \\
&\quad \implies (if\ CONS\ a_0\ a_1 = []\ then\ a_0'\ \#\# p_2\ else \\
&\quad\quad a_0'\ \#\#\ p_2\ ++\ CONS\ (\&_0)\ (CONS\ a_0\ a_1\ **\ p_2)) = [] \implies p_2 = []
\end{aligned} \tag{7.5}$$

Like all machine learning methods, the performance strongly depends on the data. This results in some issues: Firstly, the training data may be misleading as mentioned, e.g. in the case of `TAKE_TAKE`. Another limitation is that for a new function definition, there is no available information about how to choose an induction variable from its arguments before a manual proof that has induction involving this function has been carried out. Therefore, *IndMl* cannot work properly the first time it encounters the function. An alternative way would be to use recursion analysis as a supplement. This is investigated in Section 6.4.1.4 although we have not made a notable progress. Finally, the training data we have access to is relatively small, because only inductive proofs can be used. When more training data are available, the selection may become more accurate.

In our evaluation, *IndMl* achieved a close performance as *RecAna* by just learning from manual proofs. Although it has not proven more theorems, better performance may be

achieved if the issues mentioned above are addressed.

7.3 Support of non-primitive recursive definitions

In this section, **Lexicographic induction** is evaluated (see Section 5.3).

7.3.1 Dataset

List(*hilbert*) and the *Lemmas for Hoare Logic* (see Section 4.3.4) are used for evaluation, since they contain non-primitive recursive definitions (see Section 4.3.1). The *IsaPlanner benchmark* was also used (see Section 4.3.3), as it has many non-primitive recursive definitions, so it fits the evaluation here.

7.3.2 Experiment settings

In the first test with *List*(*hilbert*), the *Multi-waterfall* setting in Section 7.1.2 is used again except that **Lexicographic induction**, which will be referred to as *Lex* is used for selecting induction variable.

When testing with the *Lemmas for Hoare Logic*, we tried to simulate the process of building a new library with Boyer-Moore multi-waterfalls. If Boyer-Moore successfully proved a theorem, its generated proof is used for training instead of the manual proof. We believe this approach is closer to how Boyer-Moore might be used in practice, because if the theorem can be automatically proven, the user will not create a manual proof. The proofs generated by Boyer-Moore (see Section 5.4.2) are given to the dependency tracking system to obtain the dependencies for training.

The *IsaPlanner benchmark* was ported from Isabelle. In order to make the translation as close as possible to the original version, conditional expressions (i.e. “if then else”) in definitions and goals were kept. This caused problems in HOL Light, especially after the **clausal form heuristic** was removed (see Section 5.1) as it used to handle such expressions, so a function that eliminates conditional expressions was added as a heuristic for this evaluation. Note that a single waterfall with **simplify heuristic** (i.e. Waterfall 2 in Table 6.6) was used for this experiment as, based on our experiments, the multi-waterfall approach was problematic (worse results were achieved than with

a single waterfall for reasons which, unfortunately, we could not figure out). So there was no need to run other waterfalls because no lemma selection was involved (see Section 4.3.3).

7.3.3 Results

Lex and *RecAna* had identical success rates on *List*(*hilbert*) and proved almost the same theorems, except two. In particular, `IS_PREFIX_OF_EXISTS_APPEND` was proven after applying *Lex*. The reason was the same as mentioned in Section 7.2.2: `IS_PREFIX_OF` is non-primitive recursive and *Lex* successfully picked both *xs* and *ys* for induction. `APPEND_EQ` was only proven by *RecAna*. This was due to the limitation of ATPs again, see the discussion in Section 7.2.2 for the same theorem.

The same comparison of induction steps (as in Fig 7.6) is done for the 11 theorems proven with different induction schemes in *List*(*hilbert*), as shown in Fig 7.7. Theorems proven with *Lex* use equal or fewer inductions steps, except the 11th, `IS_PREFIX_OF_ADJACENT`. No induction was used by *RecAna* (or *IndMl*) because *HOL(y)Hammer* waterfall found a proof before an inductive proof was found by other waterfalls. With *Lex*, inductions were successfully applied and combined with simplifications to prove the theorem before *HOL(y)Hammer*. This proof is similar to its manual one, which used three induction steps with rewrites. Therefore, *Lex* appears better when it comes to the number of inductive steps.

In the comparison between *Lex* and *IndMl* with *List*(*hilbert*), the same theorems are proven, except that *IndMl* failed to prove `TAKE_TAKE` (see Section 7.2.2). The comparison for the induction steps used by the two approaches is shown in Fig 7.8. In most cases, both *Lex* and *IndMl* use one to two induction steps, and *Lex* often uses fewer induction steps. This is because *Lex* applies induction on two variables at the same time, while for the same problem, *IndMl* needs to perform two induction steps.

The only example where *IndMl* uses fewer induction steps (1 vs 3) is `EL_TAIL`, (the 10th theorem). Its statement (7.6) (if the list *xs* is longer than $(n+1)$ then the $(n+1)th$ element of the tail of *xs* is equal to the $(n+2)th$ element of *xs*). Both *Lex* and *RecAna* chose *n* for induction, because *xs* is not in a recursive argument position for *EL*. After further investigation, it was found that *IndMl* discovered the common pattern from a previous proof, whose statement (7.7) is quite close to (7.6). They have the

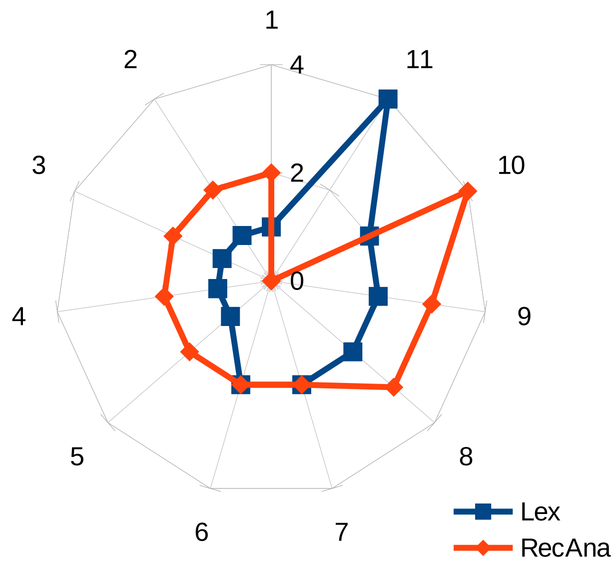


Figure 7.7: Comparison of induction steps between *Lex* and *RecAna* for *List*(*hilbert*)

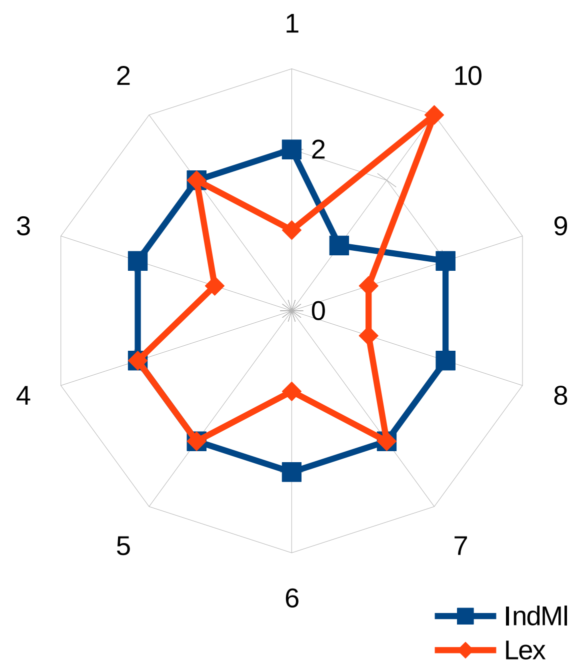


Figure 7.8: Comparison of induction steps between *Lex* and *IndMl* for *List*(*hilbert*)

	Total	Induction
RecAna	42%	32%
Lex	61%	47%

Table 7.6: Success rate of RecAna and Lexicographic induction(on lemmas for Hoare logic)

Method	Success rate
RecAna	40.2%
Lex	57.5%

Table 7.7: Success rate of RecAna and Lexicographic induction *IsaPlanner benchmark*

common features "$EL\ (SUC\ Anum)\ B(A)list$" and "$SUC\ Anum < LENGTH\ B(A)list$". Therefore, *IndMl* picked the same variable l (i.e. xs in `EL_TAIL`) for induction as the previous proof.

$$\forall\, n\, xs.\ SUC\ n < LENGTH\ xs \implies EL\ n\ (TAIL\ xs) = EL\ (SUC\ n)\ xs \tag{7.6}$$

$$SUC\ n < LENGTH\ l \implies EL\ (SUC\ n)\ (MAP\ f\ l) = f(EL\ (SUC\ n)\ l) \tag{7.7}$$

The results on the *lemmas for Hoare logic* are shown in Table 7.6. There is a notable increase of success rate after using *Lex*. Note that the increase is not only for inductive theorem, because Boyer-Moore also proved some theorems by induction whose manual proofs did not involve induction.

Lex has a better performance on the *IsaPlanner benchmark*, according to Table 7.7. The increase is due to the success of *Lex* on non-primitive definitions. The performance is close to IsaPlanner (57.5% vs 54%), but worse than HipSpec and ACL2s (92% and 85%) Claessen et al. (2013). This is partly because HipSpec and ACL2s are more powerful, modern systems for inductive proofs, whereas the Boyer-Moore implementation in HOL Light (as discussed before) is relatively primitive. More importantly, techniques like *lemma speculation* have not been fully applied to our Boyer-Moore implementation.

Lex might have a better performance if HOL Light could use a different style of definition. Specifically, definitions in HOL Light (core library) are always primitive recursive and additional definitions are needed in some cases to achieve this. For instance, consider the function EL that returns the nth elements of a list in HOL Light in (7.8), where HD and TL return the head and tail of a list. This definition is only recursive on the argument n, so l is usually not considered for induction (see `EL_TAIL` from Section 7.3.3). However, a different definition (7.9) (as used in Isabelle) is not primitive recursive and for that *Lex* works better. Another issue is that some definitions in HOL Light are not recursive at all e.g. the definition of minimum (7.10) where no recursive argument position can be found.

$$\begin{aligned} EL\,0\,l =& HD\,l \wedge \\ EL\,(SUC\,n)\,l =& EL\,n\,(TL\,l) \end{aligned} \tag{7.8}$$

$$EL\,n\,(x :: xs) = \begin{cases} x, & n = 0 \\ EL\,k\,xs, & n = SUC\,k \end{cases} \tag{7.9}$$

$$\forall m\,n.\min\ m\,n = if\ m \leq n\ then\ m\ else\ n \tag{7.10}$$

One solution could be to use more suitable formulations: the alternative definitions (e.g. (7.9)) can then be proven as lemmas and used instead of the original definition in HOL Light. If all definitions were replaced in this way, *Lex* should work better.

7.3.4 Simplification with preselected lemmas

During our investigation of failed subgoals, we found many theorems were not proven because Boyer-Moore failed to prove the subgoals even though the right induction scheme was chosen (e.g. `LENGTH_REVERSE` in Section 7.1.5). Many of them can be proven by rewriting with a list of common lemmas, but when using lemma selection, such lemmas are sent together with more than two hundred others, which slows down ATPs. Such lemmas consist of definitions (e.g. `LENGTH`, `TAKE`, `DROP`[2]) and basic arithmetic and list theorems (7.11a)-(7.11g). Since the group of lemmas is small,

[2]and other definitions: APPEND, EL, HD, TL

	Total		Induction	
	Without *Simp*	With *Simp*	Without *Simp*	With *Simp*
RecAna	42%	36%	32%	26%
Lex	61%	70%	47%	58%

Table 7.8: Success rate after adding simplification

conditional rewrite functions in HOL Light (i.e. `SIMP_CONV`) can be used. This then becomes a new heuristic, which we will refer to as *Simp*.

$$\text{NOT_SUC} :\vdash \forall n.\ \neg(SUC\ n = 0) \tag{7.11a}$$

$$\text{LT_0} :\vdash \forall n.\ 0 < SUC\ n \tag{7.11b}$$

$$\text{LT_conjunct0} :\vdash \forall m.\ m < 0 \iff F \tag{7.11c}$$

$$\text{LT_SUC} :\vdash \forall m\ n.\ SUC\ m < SUC\ n \implies m < n \tag{7.11d}$$

$$\text{SUC_INJ} :\vdash \forall m\ n.\ SUC\ m = SUC\ n \iff m = n \tag{7.11e}$$

$$\text{CONS_11} :\vdash \forall h_1\ h_2\ t_1\ t_2 = CONS\ h_2\ t_2 \iff h_1 = h_2 \wedge t_1 = t_2 \tag{7.11f}$$

$$\text{LE_SUC} :\vdash \forall m\ n.\ SUC\ m \le SUC\ n \iff m \le n \tag{7.11g}$$

Experiments were done using *RecAna* and *Lex* on *Lemmas for Hoare logic* (Section 7.3). It turns out that the special *Simp* approach has different effects on different induction heuristics: it does worse with *RecAna* and better with *Lex*, as seen in Table 7.8.

After a further investigation, the differences were found to be mainly from the second induction step: After the first induction step, the subgoal from the step cases are like "$P\ n \implies P\ (SUC\ n)$" for the natural number n, and "$P\ l \implies P\ (h :: l)$" for the list l. In both cases, n and l are no longer in recursion positions, so *RecAna* will look for other variables (or pick variable arbitrarily). In this case, *Simp* may sometimes eliminate these constructors and simplify the term back to "$P\ n$" (or "$P\ l$") where n (or l) is still in recursion position.

On the other hand, *Lex* is free from this issue because firstly it usually combines two induction steps, and there is usually a lexicographic induction on all the relevant variables without a second induction step. In addition, it also has smarter choices of induction

variables, for instance, it considers non-primitive recursive definitions and does not usually choose induction variables arbitrarily.

The combination of *Lex* and *Simp* does improve the success rate (from 61% to 70% and 47% to 58% respectively). This shows the potential for improving Boyer-Moore to prove more subgoals and thus the original goal from which they originate. However, it is important to figure out the strategy of choosing these lemmas, especially when adding new recursive types. Some of the theorems are included in the shell i.e. `NOT_SUC`, `SUC_INJ`, and `CONS_11`. Alternatively, the user can help to select such lemmas.

7.4 Generality of our approach

We believe that our approach is a generic solution to combining machine learning with automated inductive theorem proving. This is because:

- The Boyer-Moore implementation we use (see Section 2.5) is adaptable for most HOL proving systems:
  - The Shell Principle can be implemented in any systems that support recursive datatypes.
  - The functions (i.e. *conversions*) integrated as the Boyer-Moore heuristics are also available in other systems. For instance, the simplifier `SIMP_CONV` and the tautology checker `TAUT` in HOL Light are covered by the tactic `simp` in Isabelle. Automated procedures like `MESON` and `METIS` are also available in systems such as Isabelle and HOL.
- The machine learning approaches we use are generic:
  - The machine learning algorithms are available as external software independent from theorem provers and can also be implemented as internal functions.
  - The functionality offered by hammers are generic and not tied to particular systems or logics. The techniques involved in building such systems e.g. dependency tracking, feature extraction, etc. can be developed for most modern interactive theorem proving systems, especially LCF-style ones.

- The selection of induction variables requires the tracking of induction application. Our approach in HOL Light may differ from some ITPs, but it should be achievable to track the variable and statement for induction in any ITPs.
- Our multi-waterfall model is generic and simply involved running several versions of our tool in parallel in a coordinated fashion. We believe that having more parallelism in theorem proving is becoming essential. In some respect, our approach can be viewed as a secondary contribution towards effective parallelism in mechanised theorem proving.

Chapter 8

Induction for finite set

As discussed previously, the Boyer-Moore model only deals with recursive types. It is interesting (and useful) to investigate whether it can be extended to perform other kinds of induction. For this purpose, the finite set theory in HOL Light was used.

8.1 Finite sets in HOL Light

In HOL Light, sets are defined as predicates, e.g. the set membership is defined as $\forall x.\, x \in P \iff P\, x$, where P is a set on the left and a predicate on the right. Finiteness is introduced as a precondition.

$$finite\ \emptyset \wedge (\forall x\, S.\ finite\ S \implies finite\ (x\ insert\ S)) \tag{8.1}$$

The corresponding induction rule `FINITE_INDUCT` is (8.2). In addition, the rule `FINITE_-INDUCT_STRONG` (8.3) is also used.

$$\forall P.\ P\, \emptyset \wedge (\forall x\, S.\ P\, S \implies P\ (x\ insert\ S)) \implies (\forall A.\ finite\, A \implies P\, A) \tag{8.2}$$

$$\begin{aligned} \forall P.\ P\, \emptyset \wedge (\forall x\, S.\ P\, S \wedge \neg(x \in S) \wedge finite\ S \implies P\ (x\ insert\ S)) \\ \implies (\forall A.\ finite\, A \implies P\, A) \end{aligned} \tag{8.3}$$

Reasoning about finite sets and primitive recursive types differs in the following aspects:

- The property of "finite set" is not determined by the type only, but by the combination of the type (i.e. $? \to bool$) and a precondition (i.e. $finite\ S$).
- The induction rule is not derived from type definition, but from the definition of the function.

These aspects also apply to various other theories e.g. the theories about reflexive, symmetric and transitive closures in HOL Light. Therefore, we believe that the solutions from our investigation in finite sets are more generally applicable.

8.2 Simplified shell

Since a recursive data type is not used to define finite sets, the normal Boyer-Moore shell is no longer applicable. As an initial investigation, a simpler shell is used to provide the basic functionality:

- Type information. In the original shell in Boyer-Moore, the type was stored as strings. In the simpler shell, HOL types are used instead (i.e. the type of terms in HOL Light). This is because the type name of ($? \to bool$) is "fun" i.e. function, which is used by all function types in HOL Light, but only the ones that has a return type *bool* is applicable here.
- The induction rules, i.e. (8.2) or (8.3) in this case.

8.3 Reformulation of induction rule

We found it necessary to reformulate the induction rule from the one in (8.2) to the one shown in (8.6), because:

- It is preferable to have the consequent of the induction rule as "$\forall x.\ P\ x$" instead of "$\forall x.\ finite\ x \implies P\ x$". This is more general and thus more likely to match the goal we want to prove.
- The precondition $finite\ x$ is often entailed rather than given explicitly in the assumptions or preconditions of the goal. For instance, when given the definition of `HAS_SIZE` (8.4), the goal "$S\ HAS_SIZE\ 0 \implies s = \emptyset$" is actually reasoning about finite sets.

$$\forall S\, n.\ S\ HAS_SIZE\ n \iff finite\ S \wedge CARD\ S = n \tag{8.4}$$

(8.5) is derived from (8.2) and moves the consequent to the antecedent:

$$\begin{aligned}\forall P.\ P\ \emptyset \wedge (\forall x\, S.\ P\ S \implies P\ (x\ insert\ S)) \wedge \\ (\forall S.\ (finite\ S \implies P\ S) \implies P\ S) \implies \forall S.\ P\ S\end{aligned} \tag{8.5}$$

This can then be simplified to (8.6). The same reformulation can be applied to (8.3) as well.

$$\forall P.\ P\ \emptyset \wedge (\forall x\, S.\ P\ S \implies P\ (x\ insert\ S)) \wedge (\forall S.\ finite\ S \vee P\ S) \implies \forall S.\ P\ S \tag{8.6}$$

When applying (8.6), "$\forall S.\ finite\ S \vee P\ S$" is generated as an extra subgoal, compared with (8.2). In practice, we expect "$finite\ S$" to be discharged, rather than "$P\ S$". This procedure allows Boyer-Moore to make every effort (through the waterfall or maybe by induction) to eliminate "$finite\ S$". For instance, the definition in (8.4) contains "$finite\ S$" and can be used for elimination when the goal has "$S\ HAS_SIZE\ n$".

8.4 Choosing suitable induction variable

Since there are no heuristics for choosing a suitable induction variable, a naive approach would be to pick the variable that has the set type (i.e. $? \rightarrow bool$) and the property of being finite. For instance, in the formula (8.7) (from the theorem `FINITE_-SUBSET`), T already has the property *finite*, so it seems more promising to induct on it.

$$\forall S\, T.\ finite\ T \wedge S \subset T \implies finite\ S \tag{8.7}$$

On the other hand, once we know a variable has the property *finite*, there is no need to prove the subgoal like $(\forall S.\ finite\ S \vee P\ S)$ in (8.6). For any goal with statement $g = P\ v_1\ v_2 ...\ v_n$ (after removing all universal quantifiers) we:

1. Find all the free variables with set type in g, referred to as fvs e.g. $fvs = \{S, T\}$ for (8.7).

2. Find one of the v_i in fvs where the lemma "$l = \vdash \forall v_i.\ finite\ v_i \vee P\ v_1\ v_2 ...\ v_i\ ...$" can be proven. This is achieved by running multiple waterfalls on all $v \in fvs$ and picking the v for which l holds.

   For instance, for (8.7), T is chosen, and the lemma (8.8) is generated.

$$\vdash \forall T.\ finite\ T \vee (finite\ T \wedge S \subset T \implies finite\ S) \tag{8.8}$$

3. l can be reformulated as l' (8.9) where the precondition P' is simplified so that the formula $finite\ v_i$ inside $(\forall v_i.\ P\ v_1\ v_2\ ...\ v_i\ ...)$ is discharged (if there is any such formula). For instance, (8.8) is reformulated to (8.10). The antecedent of l' can be used to match the induction rule (i.e. (8.2) or (8.3)) and the consequent is the goal (with v_i generalised).

$$l' = \vdash (\forall v_i.\ finite\ v_i \implies P'\ v_1\ v_2\ ...\ v_i\ ...) \implies (\forall v_i.\ P\ v_1\ v_2\ ...\ v_i\ ...) \tag{8.9}$$

$$\begin{gathered}\vdash (\forall T.\ finite\ T \implies S \subset T \implies finite\ S) \implies \\ (\forall T.\ finite\ T \wedge S \subset\ T \implies finite\ T)\end{gathered} \tag{8.10}$$

4. Boyer-Moore is used to prove antecedent of l' and then prove g

8.5 Experiment

The above approach for reasoning about finite sets are evaluated in this section.

8.5.1 Test data and environment

There are two files involving induction for finite sets in the HOL Light core library:

sets.ml[1] is about basic set theory. It will be referred to as *set(basic)*. It contains the operations and properties of finite sets such as union, subset, membership, cardinality, etc. For instance, the union of the sets S and T is defined by (8.14).

iterate.ml[2] contains:

- Generic iterated operations i.e. $iterate\ op\ S\ f$ iteratively applies the binary operator op to $(f\ x)$ for each $x \in S$ with an initial value that is the neutral (identity) element of op (e.g. 0 is the neutral element for addition).

	Definitions	Theorems	Induction	Induction (Set)
set(basic)	20	284	40	23
set(iterated)	8	294	36	26

Table 8.1: Size of test data

- Sums over natural and rational numbers i.e. the sum of natural number is defined as $nsum = iterator\ (+)$.

For instance, the theorem about summing up a finite set of natural numbers (8.11) where f has type $? \to num$ (*num* stands for natural number) is as follows:

$$
\begin{aligned}
&(\forall f.\ nsum\ \emptyset\ f = 0) \wedge \\
&\quad (\forall x\ f\ s.\ finite(s) \\
&\qquad \Longrightarrow (nsum\ (x\ insert\ s)\ f = \\
&\qquad\qquad if\ x \in s\ then\ nsum\ s\ f\ else\ f(x) + nsum\ s\ f))
\end{aligned}
\tag{8.11}
$$

This file will be referred to as *set(iterated)*.

The induction rules `FINITE_INDUCT` and `FINITE_INDUCT_STRONG` are defined in *set (basic)*, so only the theorems proven afterwards are used. The general characteristics of these datasets are shown in Table 8.1 where *Induction* means proven by induction and *Induction (Set)* means involving induction over finite set. We use `FINITE_INDUCT_-STRONG` as it seems more generally applicable than `FINITE_INDUCT`.

The same settings as for the Boyer-Moore Multi-waterfalls in Section 6.3.5 are used, except that the automated procedure `SET_RULE` that attempts to prove set-theoretic lemmas in HOL Light is used instead of the `MESON` heuristic. `SET_RULE` tries to simplify the goal with lemmas in set theory and actually integrates `MESON` as the last procedure, so we believe that it is more appropriate than using `MESON` directly.

8.5.2 Results

The success rates for the two libraries are shown in Table 8.2. These are relatively low compared with previous experiments. There are many reasons: The proofs in these libraries are complex e.g. variables sometimes need to be instantiated with specific values, lemmas need to be cut into the proofs, etc. ATPs were also tried with these datasets. The two approaches have a similar success rate overall, mainly because they

	Total		Induction		Induction (Set)	
	set(basic)	*set(iterated)*	*set(basic)*	*set(iterated)*	*set(basic)*	*set(iterated)*
Multi-waterfall	38%	24%	15%	3%	17%	4%
ATP	37%	20%	8%	3%	4%	4%

Table 8.2: Success rates of the different systems on finite set libraries

proved many theorems in common without induction. *Multi-waterfalls* has a better performance on *set(basic)* by performing induction on finite sets. Both of them proved only a few theorems with inductive proofs in *set(iterated)*. *Multi-waterfalls* has a moderate success rate because some theorems are proven by a simplification followed by `SET_RULE`.

A theorem proven by Boyer-Moore with induction for finite set is `FINITE_UNION_IMP` (8.12) i.e. the union of two finite sets is finite. Interestingly, the manual proof begins with the reformulation of the statement to match the induction rule shown as (8.13), which is achieved automatically with our approach. A subgoal was also proposed in the manual proof, which is identical to a previously proven theorem and found by lemma selection.

$$\forall S\, T.\ finite\ S \wedge finite\ T \implies finite\ (S \cup T) \tag{8.12}$$

$$\forall S.\ finite\ s \implies (\forall T.\ finite\ T \implies finite\ (S \cup T)) \tag{8.13}$$

Boyer-Moore happened to pick the same variable S as the manual proof. It actually makes no difference if T is chosen, because the union of set is defined as (8.14) which is symmetric for S and T. In addition to this, we also found that Boyer-Moore often picked the same variable as the manual proof, though sometimes by chance, so the approach of choosing a set variable for induction is currently sufficient.

$$\forall S\, T.\ S \cup T = \{x | x \in S \vee x \in T\} \tag{8.14}$$

A more complicated theorem `INF_FINITE_LEMMA` (8.15) (any non-empty finite set of real numbers has an infimum) was proven with the help of lemma selection: after induction applied, a one-step proof was found while the manual proof has three steps.

This example shows the ability of combining machine learning and the HOL Light implementation of Boyer-Moore, higher order theorems can be proven.

$$\forall S.\ finite\ S \land \neg(s = \emptyset) \implies \exists b : real.\ b \in S \land \forall x.\ x \in S \implies b \leq x \tag{8.15}$$

One failed example is `NSUM_CONST` (8.16) i.e. the sum of the natural number c (from $\lambda\ n.\ c$) in finite set S is equivalent to the cardinality of the S times c (see the description for *nsum* in Section 8.5.1). The induction step is shown in (8.17) where c is universally quantified by our induction procedure (see Section 5.1.2). This however is undesired, not only because the goal is easier to prove when c is identical in the induction hypothesis and conclusion, but also because *HOL(y)Hammer* translates the λ-expressions containing them to different functions. Without the universal quantifier for c, this goal can be proven by *HOL(y)Hammer*.

$$\forall c\ S.\ finite\ S \implies nsum\ S\ (\lambda\ n.\ c) = (card\ S) * c \tag{8.16}$$

$$\begin{aligned}(\forall c.\ finite\ S \implies nsum\ S\ (\lambda\ n.\ c) = card\ S * c) \land \\ \neg(x \in S) \land\ finite\ S \implies \\ finite\ (x\ insert\ S) \implies nsum\ (x\ insert\ S)\ (\lambda\ n.\ c) = card\ (x\ insert\ S) * c\end{aligned} \tag{8.17}$$

Specifically, *HOL(y)Hammer* assigns $f_1 = \lambda\ c\ n.\ c$ and $f_2 = \lambda\ n.\ c$ in order to translate the goal to first order (see Section 2.4.1), so (8.17) is translated to (8.18) where the first two antecedents are added after the translation. *HOL(y)Hammer* chooses a different translation probably to reduce the nested *happ* formula, which leads to a more complicated goal. However, this goal cannot be proven unless ATPs figure out that $f_1\ c = f_2$.

This issue may be fixed by changing the translation of *HOL(y)Hammer* and keeping only f_1. However, a more delicate heuristic could be developed which decides whether the variable not used for induction should be universally quantified instead of the approach in Section 5.1.2.

$$\begin{aligned}\forall f_1.\ (\forall c\ n.\ happ\ (happ\ f_1\ c)\ n = c) \implies \\ (\forall f_2.\ (\forall n.\ happ\ f_2 = c) \implies \\ (\forall c.\ finite\ S \implies nsum\ S\ (f_1\ c) = card\ S * c) \wedge \\ \neg(x \in S) \wedge\ finite\ S \implies \\ finite\ (x\ insert\ S) \implies nsum\ (x\ insert\ S)\ f_2 = card\ (x\ insert\ S) * c)\end{aligned} \tag{8.18}$$

There are situations where we need to choose the induction variable between a finite set and recursive datatype. For instance, `HAS_SIZE_FUNSPACE` (8.19), which states that the set of all the functions which map the elements in a set S to T has size n^m where n and m are the cardinalities of S and T respectively (the definition of `HAS_SIZE` is shown as (8.4)). With our approach, Boyer-Moore picked T for induction. However, the manual proof chose m, i.e. induction over natural number. A heuristic is required to choose the induction variable between these two kinds of variables.

$$\begin{aligned}\forall d\ n\ T\ m\ S.S\ HAS_SIZE\ m \wedge T\ HAS_SIZE\ n \implies \\ \{f | (\forall x.\ x \in S \implies f(x) \in T) \wedge (\forall x.\ \neg(x \in S) \implies (f(x) = d))\}\ HAS_SIZE\ (n^m)\end{aligned} \tag{8.19}$$

There are more theorems not proven because there are subgoals that are cut into the manual proofs. Unlike the proof for `FINITE_UNION_IMP`, such subgoals are usually not proven as named lemmas. In addition, `EXISTS_TAC`, which requires the user to specify a value for an existential variable and `COND_CASES_TAC`, which eliminates the conditional expressions (i.e. *if then else*) in the goal are not supported by our Boyer-Moore implementation.

8.6 Conclusion

In this chapter, we investigated induction for finite sets as a means of performing induction on non-recursive datatypes. We had to reformulate induction rules for such sets so that they can be used by the existing Boyer-Moore implementation.

However, our experiments with these approaches achieved limited results. This was partly due to the difficult test data e.g. because many proofs contain subgoals cut in by the user. Other issues such as choosing a suitable variable for induction, particularly-between finite sets and other recursive types.

Chapter 9

Application: verification of a sorting algorithm

Program verification is a common use of inductive theorem proving. So, we decided to investigate the performance of our approach by formally verifying the selection sort algorithm for lists.

9.1 Data for evaluation

This sorting algorithm is simple but important in program design. There are no algorithms formalised about sorting with Hoare logic in the ITP libraries such as Isabelle, HOL Light and HOL4 that are suitable for our experiment [1]. For this reason, an imperative version of *selection sort* is formalised using *Hoare logic*[2] (Hoare, 1969)

9.1.1 Selection sort

Selection sort keeps a sorted sublist in the back (right) of the list, which is initially empty and the unsorted sublist in the head. In each iteration, the biggest element is picked from the unsorted sublist and added to the front of the sorted sublist. Finally, all the elements are moved to the sorted sublist and the list is in ascending order.

[1] Although there are similar algorithms, they define lists using the notion of a heap, which is not a recursive type.

[2] https://github.com/jrh13/hol-light/blob/master/Examples/prog.ml

```
i=SIZE
while (1<i){
  j=i-1;
  k=j;
  while(0<=j){
    if(a[k]<=a[j])k=j;
    j=j-1;
  }
  tmp=a[k];
  a[k]=a[i-1];
  a[i-1]=tmp;
  i=i-1;
}
```

Figure 9.1: Selection sort algorithm

The program for investigation in Fig 9.1 sorts a list a of size $SIZE$ in ascending order. In each iteration of the outer loop, the biggest value is put in front of the sorted sublist in the tail.

This algorithm involves the recursive datatype list and recursive function definitions such as the property of being sorted and taking/droping elements from the list, so its verification requires proof by induction. Meanwhile, this algorithm is challenging because two nested loops are used and various changes to the list are made within each iteration of loop. We also address an issue regarding termination checks for the nested while loops in Section 9.2.

9.1.2 Hoare Logic

Hoare logic is a formal system with axioms and inference rules for verifying properties of imperative programs (Hoare, 1969). The formulas of Hoare logic are *Hoare triples* shown as (9.1) where P and Q are the precondition and postcondition respectively, and C is a (program) command. This formula claims that if P is true before the execution of C then Q is true after the execution of C.

$$\{P\}\ C\ \{Q\} \tag{9.1}$$

The proof rules for Hoare logic are shown in (9.2a)-(9.2e) (Nipkow and Klein, 2014) where *SKIP* means do nothing. These rules can be used to prove the partial correctness and, if termination is also proven, the total correctness of programs. The rules below are for partial correctness proofs:

$$\frac{}{\{P\}\ SKIP\ \{P\}} \tag{9.2a}$$

$$Assignment: \frac{}{\{P[a/x]\}\ x := a\ \{P\}} \tag{9.2b}$$

$$Sequence: \frac{\{P_1\}\ c_1\ \{P_2\} \quad \{P_2\}\ c_2\ \{P_3\}}{\{P_1\}\ c_1;\ c_2\ \{P_3\}} \tag{9.2c}$$

$$Conditional: \frac{\{P \wedge b\}\ c_1\{Q\} \quad \{P \wedge \neg b\}\ c_2\ \{Q\}}{\{P\}\ IF\ b\ THEN\ c_1\ ELSE\ c_2\ \{Q\}} \tag{9.2d}$$

$$While\ loop: \frac{\{P \wedge b\}\ c\ \{P\}}{\{P\}\ WHILE\ b\ DO\ c\ \{P \wedge \neg b\}} \tag{9.2e}$$

In practice, *verification conditions* (VCs) are generated based on these proof rules such that we can prove the Hoare triples by discharging VCs. VCs can be automatically generated by VC generators such as `VC_TAC` in HOL Light. However, the user still needs to provide *invariants* (i.e. P in (9.2e)) for the verification of while loops. Invariants remain true after each iteration of the loop.

9.1.3 Goal for the verification of selection sort

The HOL Light formalisation of a selection sort in Hoare logic is shown in Fig 9.2:

It is a mixture of assertions and program commands, which is parsed as a goal in HOL Light. The sorting starts from the tail to the head of the list. We chose this order because it is consistent with the recursive definition of lists i.e. *cons* an element to another list. The program is similar to Fig 9.1 except that $j-1$ is used instead of j. This is because if we use j, the condition in the inner loop is changed to $0 \leq j$, which is always true and so the loop never terminates. Note that $\forall m\ n.\ n > m \implies m - n = 0$ in HOL Light.

We provide an explanation for the code shown in Fig 9.2.

Definitions: LENGTH LENGTH *lst* returns the length of the list *lst*

TAKE TAKE *i lst* returns the first *i* elements of the list *lst*

```
   {T}
     var i,j,k,lst,tmp;
       i := LENGTH lst;
       while SUC 0 < i do
         [invariant  SORTED (DROP i lst) /\ (i=LENGTH lst \/
            (!x. MEM x (TAKE i lst)  ==> ALL ((<=)x) (DROP i lst))/\i<LENGTH lst) ;
          measure i]
           (j:=i;
            k:= j-1;
            while 0 < j  do [ invariant SUC 0 < i /\ SORTED (DROP i lst)/\ (j=i\/
                           ALL ((>=) ( EL k lst)) (DROP j (TAKE i lst))) /\
                          (i=LENGTH lst \/ (!x. MEM x (TAKE i lst)  ==>
                           ALL ((<=)x) (DROP i lst)) /\i<LENGTH lst)/\j-1<=k/\k<i;
                                         measure j]
            (if EL k lst <= EL (j-1) lst then (k := (j-1));
             j:= j-1);
         tmp := EL k lst;
         lst := ASG lst k (EL (i-1) lst);
         lst := ASG lst (i-1) tmp;
         i := i - 1)
     end
{SORTED lst}
```

Figure 9.2: Evaluation of selection sort

DROP DROP i lst returns the list lst after removing the first i elements

nth EL i lst returns the ith element of lst

MEM MEM x lst is true iff x is a member of lst:

$$(MEM\ x\ [] \iff T) \wedge (MEM\ x\ (CONS\ h\ t) \iff x = h \wedge MEM\ x\ t) \quad (9.3)$$

ALL ALL P lst is true iff P holds for all elements in lst:

$$(ALL\ P\ [] \iff T) \wedge (ALL\ P\ (CONS\ h\ t) \iff P\ h \wedge ALL\ P\ t) \quad (9.4)$$

SORTED There are different definitions for sorted, here we use a recursive definition:

$$\begin{aligned}(SORTED\ [] \iff T) \wedge \\ (SORTED\ (CONS\ h\ t) \iff (ALL\ ((\leq)h)\ t) \wedge SORTED\ t)\end{aligned} \quad (9.5)$$

List update ASG lst i k means $lst[i] := k$ (i.e. *list_update* in Isabelle. Here ASG stands assignment for short). Since in-place list update is not supported in our Hoare logic environment, and it is usually implemented by complicated approaches such as arrays or heaps, we simply assign the updated list to the original one.

Program command The commands such as ":=" and "while" are the Hoare logic ones. The conditional $if\ b\ then\ c$ is the same as $IF\ b\ THENc\ ELSE\ SKIP$, so is covered by (9.2d).

Precondition and Postcondition The expression "{`T`} ... {`SORTED lst`}" in 9.2 is the Hoare triple where `T` is just $True$ as a precondition i.e. no precondition is required in our case and the postcondition is the statement we want to prove: {`SORTED lst`}, i.e. lst is sorted after the algorithm. Note that for a full verification of the sorting algorithm, we also need to prove that the sorted list is a permutation of the original one. This aspect is left as future work.

Invariant The first part of the invariant is for the outer loop, which says that the tail of the list (after ith element) is always sorted, and the elements in the tail (sorted sublist) is always larger any elements in the front (unsorted sublist), after each loop. The second invariant is for the inner loop, which says that the invariant from the outer loop holds, and the kth element is the biggest one in the sublist from jth to ith.

Measure In the outer and inner loop, i and j decrease by one each loop respectively

9.2 Total correctness of nested loops

When directly generating verification condition goals for the algorithm in Fig 9.2, the termination of the outer loop is unprovable. i.e. we cannot prove that i decreases at each iteration. The reason will be explained in this section.

The rule for verifying the total correctness of while loops is shown in (9.6) (Nipkow and Klein, 2014) ($\vdash_t$ stands for total correctness).

$$\frac{\forall n.\ \vdash_t \{\lambda\, s.P\ s \wedge b \wedge n = f\ s\}c\{\lambda\, s.P\ s \wedge f\ s < n\}}{\vdash_t \{P\}WHILE\ b\ DO\ c\{\lambda\, s.P\ s \wedge \neg b\}} \tag{9.6}$$

where f is the *measure* function for checking the termination and s is the state. The global variable n stores the value of f in order to detect its decrease.

The corresponding VC generated by the `VC_TAC` in HOL Light is:

$$\begin{aligned} vc\ \{P\}\ WHILE\ b\ [I]\ DO\ c\{Q\} = & \\ & (\forall s.\ P\ s \implies I\ s) \wedge (\forall s.\ \neg b \wedge I\ s \implies Q\ s) \wedge \\ & (\forall n.\ vc\ \{\lambda s.\ I\ s \wedge b \wedge n = f\ s\} \\ & \quad c \\ & \quad \{\lambda s.\ I\ s \wedge f\ s < n\}) \end{aligned} \tag{9.7}$$

Where I is the invariant given by the user and vc is a function that is recursively invoked to generate the VC from the loop body c. n is the variable for the termination checking and must be passed through the VC from c. However, when c contains other loops, (9.7) will be used again, the n from outer loop is contained in P and Q, and will never be eliminated. For instance, the VC generated from the inner loop in Fig 9.2 contains $i = n$ in P and $i - 1 < n$ in Q, but they are in different conjuncts according to (9.7). Such variable n cannot be given as invariant by the user, which is the only connection between P and Q. Therefore, the termination of outer loop cannot be verified.

An existing approach that solves the issue of verifying the termination is to fix the value of the outer loop variable (Schirmer, 2008). It requires the user to specify the

variable that remains unchanged in the inner loop. This also avoids repeating the outer-loop invariant that involves such variable, as we did in Section 9.1.3 (copying invariant from outer loop to inner loop).

However, this approach assumes that there are variable that never changed in the inner loop, which may not be always true and requires the user to pick such variables. So we come up with a new approach, by introducing a *Hoare quadruple*, which we have not found to exist already in the literature. The quadruple has an extra fixed part and is just some additional syntax sugar over the usual *Hoare triple* (on the right hand side) added for convenience:

$$\{p\}c\{q\}\{fix\} \iff \{p\}c\{\lambda s.\ q\ s \wedge\ fix\ s\} \tag{9.8}$$

The *Hoare triple* in (9.7) is converted to a *Hoare quadruple* for the outer loop:

$$\begin{aligned} vc\ \{P\}\ WHILE\ b\ [I]\ DO\ c\{Q\} = \\ & (\forall s. P\ s \implies I\ s) \wedge (\forall s. \neg b \wedge I\ s \implies Q\ s) \wedge \\ & (\forall n.\ vc\ \{\lambda s. I\ s \wedge b \wedge n = f\ s\} \\ & \quad c \\ & \quad \{\lambda s. I\ s\} \underbrace{\{\lambda s. f\ s < n\}}_{fix}) \end{aligned} \tag{9.9}$$

Then for the inner loop where there is already a *Hoare quadruple* shown as (9.10), the two fixed parts can be eliminated when c does not have loop or passed to the loop inside c again. The VC remains in the form of *Hoare quadruple*.

$$\begin{aligned} vc\ \{P\}\ WHILE\ b\ [I]\ DO\ c\{Q\}\{fix\} = \\ & (\forall s. P\ s \implies I\ s) \wedge (\forall s. \neg b \wedge I\ s \implies Q\ s) \wedge \\ & (\forall n.\ vc\ \{\lambda s. I\ s \wedge \mathbf{fix}\ x \wedge b \wedge n = f\ s\} \\ & \quad c \\ & \quad \{\lambda s. I\ s\}\{\lambda s. f\ s < n\ \wedge \mathbf{fix}\ x\}) \end{aligned} \tag{9.10}$$

Most rules for generating VCs in HOL Light can be reused by simply rewriting *Hoare quadruple* back to *triple*. The only exception is that the VC from the sequence rule shown as (9.11) (derived from (9.2c)) needs to be changed to (9.12):

$$vc\ \{P\}(c_1;c_2)\{Q\} = (vc\ \{P\}c_1\{R\}) \wedge (vc\ \{R\}c_2\{Q\}) \tag{9.11}$$

$$\begin{aligned} & vc\ \{P\}(c_1;c_2)\{Q\}\{fix\} = \\ & \quad (vc\ \{P\}c_1\{R\}\{fix'\}) \wedge (vc\ \{\lambda s.R\ s \wedge fix'\ s\}c_2\{Q\}\{fix\}) \end{aligned} \tag{9.12}$$

Where R is still figured out with the existing VC generating approach i.e. from (9.11), but fix' is adjusted:

- When c_2 is not a (while) loop: fix' is figured out in the same way as R using (9.11) but fix is used instead of Q in the formula
- When c_2 is a loop: $fix' = fix$ so that the information from outer loop in fix is passed to fix through the inner loop.

Compared with the existing one, our method focus on passing the measure variable (i.e. n in the fix part), while leaving the rest part unchanged. Users do not have to pick the fixed variable, but give the invariant as usual. Some of the invariant from outer loop needs to be repeated in the inner loop, which makes this method a little tedious. However, if stronger assumptions are given in the inner loop, the user does not have to copy the identical invariant from the outer loop to the inner loop, so this also allows some flexibility.

9.3 Experiments

9.3.1 Supporting library

Our proof about selection sort involves definitions about list operations (take, drop, etc.). There were no theorems about list update or sorted in HOL Light libraries and unsurprisingly our initial experiment that tried to prove the VC goals without such lemmas did not achieve good results. Therefore, the relevant lemmas were ported from Isabelle. The libraries finally used are mentioned in Table 9.1 .

9.3.2 Goal split & simplify

There are six VCs for the goal in Fig 9.2. When they cannot be proven by Boyer-Moore automatically, a few strategies are tried to emulate how the user would split the goals to easier ones. The most frequently used strategy is to use the HOL Light tactic to split connectives (e.g. $\wedge$, $\Longrightarrow$, and etc.) in the conclusion (i.e. `STRIP_TAC`), then apply the

Name	Description
List(*hilbert*)	See Appendix A.2.2 for the list of theorems
Lemmas for Hoare Logic	Theorems ported from Isabelle about list update, sorted, etc. see Appendix A.2.5 for the list of theorems

Table 9.1: Libraries for the verification of selection sorted

decision procedure for solving linear arithmetic problems (i.e. `ARITH_TAC`) to all the split goals, which will eliminate the trivial goals, e.g. verifying the termination of the loops (they are usually easy arithmetic problems about loop counters).

A goal may still be unproven after the split. Some methods to make the goal easier and help Boyer-Moore to prove it are tried:

- The user performs some additional proof steps and then try Boyer-Moore again.
- The user comes up with some lemmas, proves them with Boyer-Moore, and then tries Boyer-Moore with these lemmas on the goal at hand.

9.3.3 Prove subgoals by Boyer-Moore automatically

The subgoals split from Fig 9.2 are in Appendix A.1. The results of running Boyer-Moore directly to prove the subgoals is shown in Table 9.2. The six VC goals have been split in the way mentioned in Section 9.3.2, except for the first and third, which can be proven by Boyer-Moore without splitting. ✓and $\times$ stands for whether the goal can be proven by Boyer-Moore automatically. We use the *Multi-waterfall* setting in Section 7.1.2. According to the table, many goal could not be proven automatically, and we investigate the reasons in Section 9.3.4.

9.3.4 Examples

Fig 9.3 shows the goal 2.2, which was proven by Boyer-Moore automatically. Note that i can only be 0 or 1 in this goal and a case split can prove it. Case split is not supported in Boyer-Moore, but it can achieve the same effects by doing induction on i twice. Three induction steps were used (one on lst, and two on i), which shows the potential Boyer-Moore's power for solving complicated problems.

Goal	Proven	Induction step
1	✓	0
2.1	✓	1
2.2	✓	3
3	✓	1
4.1	×	-
4.2	×	-
4.3	×	-
4.4	×	-
5.1	×	-
5.2	×	-
5.3	×	-
5.4	×	-
6.1	✓	2
6.2	✓	2
6.3	×	-
6.4	×	-

Table 9.2: Results of running Boyer-Moore to prove subgoals from Fig 9.2

```
  0 [`~(SUC 0 < i)`]
  1 [`SORTED (DROP i lst)`]
  2 [`!x. MEM x (TAKE i lst) ==> ALL ((<=) x) (DROP i lst)`]
  3 [`i < LENGTH lst`]

`SORTED lst`
```

Figure 9.3: Goal 2.2

```
 0 [`SUC 0 < i`]
 1 [`SORTED (DROP i lst)`]
 2 [`ALL ((>=) (EL k lst)) (DROP j (TAKE i lst))`]
 3 [`!x. MEM x (TAKE i lst) ==> ALL ((<=) x) (DROP i lst)`]
 4 [`i < LENGTH lst`]
 5 [`j - 1 <= k`]
 6 [`k < i`]
 7 [`i - 1 < X`]
 8 [`0 < j`]
 9 [`X' = j`]
10 [`EL k lst <= EL (j - 1) lst`]

`ALL ((>=) (EL (j - 1) lst)) (DROP (j - 1) (TAKE i lst))`
```

Figure 9.4: Goal 5.4 (simplified)

A goal which Boyer-Moore failed to prove (goal 5.4) is shown in Fig 9.4. The main idea of here is embodied by (9.13) (if $a \leq b$ and a is larger than all elements in l, then b is larger than all elements in l) which can be proven by our system. A transformation is required: *cons* "EL $(j-1)$ *lst*" "(DROP j (TAKE i *lst*))" giving us "(DROP (j - 1) (TAKE i *lst*))" (in the conclusion) i.e. (9.14). However, it is difficult for Boyer-Moore to figure this out.

$$\forall a\, b.\ a \leq b \implies ALL\ ((\geq)a)\ l \implies ALL\ ((\geq)b)\ l \tag{9.13}$$

It is still possible to use Boyer-Moore to complete this proof. For instance, this goal could be proven in the following way:

- Prove the lemma (9.14), which can be automatically dealt with by Boyer-Moore (*lst* here has variable type A instead of natural number, which is its type in the goal, because Boyer-Moore cannot prove the goal with the type instantiated. This is discussed in Section 9.4). It then needs to be instantiate (i.e. j is instantiated with $j-1$)

$$\begin{aligned} &\forall j\ lst.\ j < LENGTH\ lst \implies \\ &\quad DROP\ j\ lst = CONS\ (EL\ j\ lst)\ (DROP\ (SUC\ j)\ lst) \end{aligned} \tag{9.14}$$

- Prove the subgoal EL $(j-1)$ (TAKE i lst)=EL$(j-1)$ lst, which Boyer-Moore failed to prove, so the user need to provide extra lemmas for this subgoal.
- Prove the preconditions, for instance: $j-1 <$ LENGTH (TAKE i lst) and 0$< j \implies$ SUC$(j-1)$=j. These preconditions can be automatically proven by Boyer-Moore.

From this example, Boyer-Moore shows its ability to prove some lemmas automatically for the user. However, a big effort is required to prove the goal, which are usually unnoticeable when testing with corpora where the theorems are given in their general form e.g. for the evaluation in Chapter 7. For instance, the difficulties of a VC goal like Goal 5.4 are:

- The user needs to figure out the subgoals and (the statements of) the lemmas e.g. (9.14).
- There is a mixture of arithmetic and other problems. e.g. the user has to prove the precondition "0$< j \implies$ SUC$(j-1)$=j" so that (9.14) can be used, which is easy arithmetic, but difficult (or tedious) to figure out.
- There are irrelevant assumptions that makes Boyer-Moore select wrong induction variables and lemmas. For instance, X (for termination checking) is totally irrelevant here, but may be chosen for induction. The assumption "SORTED (DROP i lst)" is not used, but can result in irrelevant lemmas about "SORTED" during lemma selection.
- Some variables e.g. lst has its type instantiated, i.e. list of natural numbers here, while the lemmas in the library may have type list of unspecified variables. This can affect lemma selection (see Section 9.4).

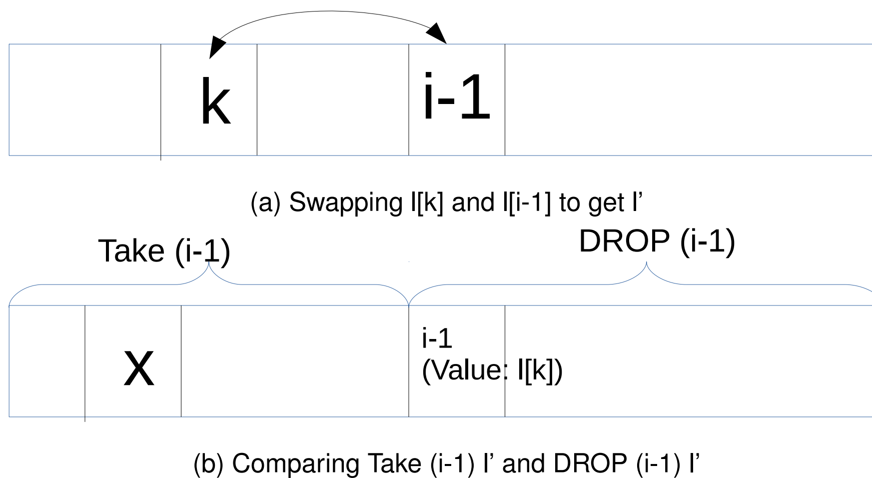
Goal 4.2 is even more complicated for Boyer-Moore shown as Fig 9.5. Its first conjunct is: After swapping the max element $l[k]$ with the last unsorted $l[i\text{-}1]$, i.e. l'= (ASG (ASG lst k (EL $(i-1)$ lst)) $(i-1)$ (EL k lst)) (Fig 9.6a), any element x in TAKE $(i-1)$ l' is not larger than any y in DROP $(i-1)$ l'(Fig 9.6b).

There are many cases to split. For instance, the swap may not happen (i.e. $k = i-1$), x may be swapped (i.e. x is l'[k]), where different assumptions should be used. This is complicated for both Boyer-Moore and the users.

During the investigation of the subgoals, we found Boyer-Moore could not prove the

```
Goal 4.2:
  0 [`~(0 < j)`]
  1 [`SUC 0 < i`]
  2 [`SORTED (DROP i lst)`]
  3 [`ALL ((>=) (EL k lst)) (DROP j (TAKE i lst))`]
  4 [`i = LENGTH lst`]
  5 [`j - 1 <= k`]
  6 [`k < i`]
`(!x. MEM x
      (TAKE (i - 1) (ASG (ASG lst k (EL (i - 1) lst)) (i - 1) (EL k lst)))
      ==> ALL ((<=) x)
          (DROP (i - 1)
          (ASG (ASG lst k (EL (i - 1) lst)) (i - 1) (EL k lst))))/\
 i - 1 < LENGTH (ASG (ASG lst k (EL (i - 1) lst)) (i - 1) (EL k lst))`
```

Figure 9.5: Goal 4.2



(a) Swapping l[k] and l[i-1] to get l'

(b) Comparing Take (i-1) l' and DROP (i-1) l'

Figure 9.6: Goal 4.2 viewed diagrammatically

conjecture we suggested in (9.15a) with the three required lemmas shown below selected from the corpora. The conjecture is finally simplified with these lemmas to (9.15e) which can be solved by the decision procedure for linear arithmetic. Instead, Boyer-Moore reproved this conjecture directly from definitions (with either lexicographic induction or machine learning to select the induction variable because `TAKE` and `DROP` are non-primitive recursive). The system failed to use the lemmas because:

- When trying to use the simplify heuristic (to get (9.15e), it does not rewrite (9.15b) from right to left as desired.
- When trying to use *HOL(y)Hammer* to prove it, (9.15e) is required as a lemma (or the more general one: $\forall m\, n.\ (\min\ m\ n) - m = 0$).

$$\forall l\ i.\ DROP\ i\ (TAKE\ i\ l) = [] \tag{9.15a}$$

$$\text{LENGTH_EQ_NIL} : \forall l.\ LENGTH\ l = 0 \iff l = [] \tag{9.15b}$$

$$\text{LENGTH_DROP} : \forall xs\ n.\ LENGTH\ (DROP\ n\ xs) = LENGTH\ xs - n \tag{9.15c}$$

$$\text{LENGTH_TAKE} : \forall xs\ n.\ LENGTH\ (TAKE\ n\ xs) = \min\ n\ (LENGTH\ xs) \tag{9.15d}$$

$$\forall l\ i.\ (\min\ i\ (LENGTH\ l)) - i = 0 \tag{9.15e}$$

This issue might be solved if more lemmas about minimum were made available for selection (HOL Light has few such lemmas). We ported some additional lemmas from HOL4, but no improvements were observed. We also attempted to use Z3 to help with the problem but no improvement was observed.

9.4 Using terms as features

The string-based approach of generating features in *HOL(y)Hammer* (see Section 3.3.2) has a limitation that the connection between the instantiated terms and the original one may be lost during the feature extraction. For instance, the type of l in (9.15a) is instantiated with natural number in order to prove our VC goals i.e. from $l : A$ to $l : num$. However, this causes a mismatch between the feature "LENGTH A(num)list" in the conjecture and the feature "LENGTH A(A)list" from the theorem (e.g. (9.15d)). In this case, lemma selection will not be able to find the corresponding more general lemmas.

In an attempt to solve this problem, we tried to use the term as features instead of the string so that the more general term “LENGTH l:(A)list” subsumes the specific one “LENGTH l:(num)list”. The main difference between these two approaches are:

- When using string as features, the feature vector is generated by comparing the strings i.e. if string str can be extracted from a statement s, then s has features str.
- If terms are used as features instead of strings, the aforementioned instantiation relation is used instead. i.e. if a term tm can be extracted from a statement s, for any tm' where tm is its instantiation, s has the feature tm'. This approach will be referred to as *term feature* .

9.4.1 Generating term features

Many steps involved in the generation of term features are similar to those for string features, except that the term is not converted to a string in the last step. However, there are several changes that are needed when extracting and using term features. These are described next.

9.4.1.1 Two round extraction

The first difference is that the features of each term changes dynamically. For example, suppose there were two statements for feature extraction and (9.16) was processed before (9.17) the term feature term “LENGTH l:(A)list” would be only extracted as a feature for (9.17)However, (9.16) should also has this feature. In this case, two round of lemma extraction is required:

1. In the first round, terms are extracted from each statements and collected as a set.
2. In the second round, features corresponding to each statement are generated by looking for instantiations between the terms extracted from it and the ones already collected in the set.

Note that when having incremental learning (see Section 3.3.4), we need to redo both of the rounds, which makes it more complicated than the string feature approach.

$$LENGTH\ (REVERSE\ (l : num\ list)) = LENGTH\ l \tag{9.16}$$

$$LENGTH\ (REVERSE\ (l : A\ list)) = LENGTH\ l \tag{9.17}$$

9.4.1.2 Feature Normalisation

The second difference is in feature normalisation (see Section 3.3.2). The normalisation to the same variable and type name may cause problem. For instance, the term "f (x:A)" is normalised to "A:(A→A) A:A" (Identifiers "f" and "x" are renamed to A. All the variable types are also renamed to A, so f is normalised to "A:(A→A)" and "x" is to "A:A"). However, the extracted term feature fails to match the original term, because it assumes f to have the same return type as x, which is too strong. A better solution is to generalise "f x" to "A:(A2→A1) A:A2". Note that we do not to rename the variable names differently where the normalisation should be "A1:(A2→A1) A2:A2". In practice, only changing the type name is enough for HOL Light to match the two terms.

The detailed feature extraction procedure for a library of statements **S** is:

1. Round 1: Create an empty feature set **F**. For each $s_i \in \mathbf{S}$:
   (a) Extract features $F_i = f_{i1}, f_{i2}, ...$ in the same way as *HOL(y)Hammer* from s_i (Section 3.3.2), except that the term is not converted to string, and normalised in the way mentioned in this section.
   (b) Add $f_{i1}, f_{i2}, ...$ to **F**
2. Label features in **F**, so that $\mathbf{F} = f_1, f_2, ...$
3. Round 2: Create an empty *feature matrix*, *Fea* with size $|\mathbf{S}| \times |\mathbf{F}|$. For each s_i:
   (a) Extract feature F_i as Round 1.
   (b) Update *Fea* with (9.18).

$$Fea(i,j) \ = \ \begin{cases} 1 & \text{if } \exists f \in F_i, f'_j \in \mathbf{F}.\ f \text{is an instantiation of} f'_j \\ 0 & \text{otherwise} \end{cases} \tag{9.18}$$

Fea can then be used for lemma selection in the same way as the *feature matrix* generated with string features (see Section 3.3.2).

9.4.2 Limitation

The time needed is the biggest weakness of the term feature approach, compared with string feature. This is because: First, the two round feature extraction takes extra time. Second, computing the instantiation of term features is slower than comparing string features. Moreover, its implementation is more complicated compared with string feature, particularly when adding the support for incremental learning, which still requires extra implementations and has not been done yet.

9.4.3 Evaluation of term feature

We evaluated this approach with *HOL(y)Hammer* but obtained only a relatively small improvement compared with string features. We believe the reason is that the theorems for evaluation are from libraries, particularly the supporting libraries, which usually have only the most general forms i.e. polymorphic types are used rather than theorems with specific (instantiated) types like (9.16). In this case, a "real world" problem such as the VC goals in this chapter is more suitable for evaluation.

The statement (9.19) is a goal derived from Goal 5.2 in Appendix A.1 , which says that when j is less than the length of a list *lst* of natural numbers, the outcome of taking first $j+1$ elements and then removing first j elements is equal to the list that contains only the jth element of lst. This goal was only proven after replacing the type *num* to the generic type A. This seems to support our hypothesis that the lemmas available for selection are only about lists with generic types, and some of the string feature (e.g. "$Anumlist$") from the term of the goal (e.g. "*lst* : *num list*") do not match the feature (e.g. "$AAlist$") from the lemmas (e.g. "*lst* : *A list*"), which leads to the poor performance of lemma selection.

$$j < LENGTH\ (lst : num\ list) \implies DROP\ j\ (TAKE\ (SUC\ j)\ lst) = [EL\ j\ lst] \quad (9.19)$$

To see whether term features solve this issue, lemma selection is performed on the two

Abbrev	Explanation
str:A	using string feature and change *lst* type to *A*
str:num	using string feature and instantiate *lst* with type *num*
term:A	using term feature and change *lst* type to *A*
term:num	using term feature and instantiate *lst* with type *num*

Table 9.3: Methods for comparing different features

subgoals (9.20) and (9.21)[3] obtained from performing induction on the goal (9.19) with the settings shown in Table 9.3, i.e. by comparing whether string or term features are used; whether the polymorphic type "A" or the specific type "num" is used, and then outputting the rank of all the lemmas required to prove this subgoal for comparison.

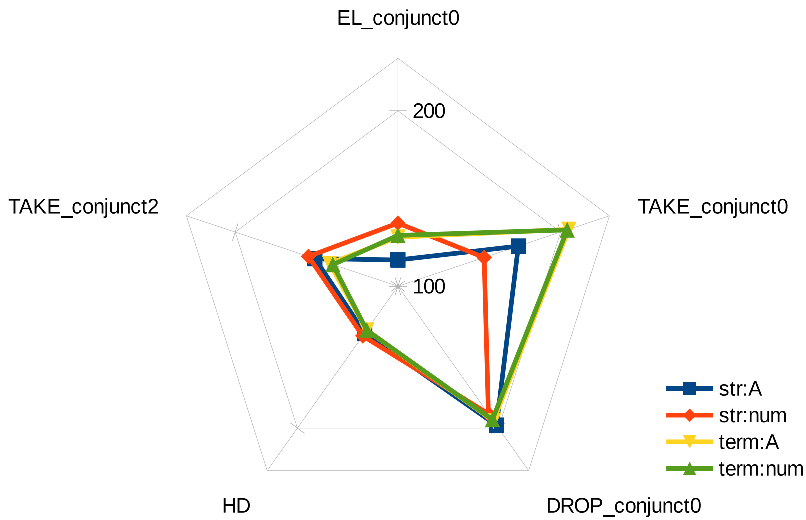
$$\begin{aligned}&(0 < LENGTH\ a_1 \implies DROP\ 0\ (TAKE\ (SUC\ 0)\ a_1) = [EL\ 0\ a_1])\\ &\implies 0 < LENGTH\ (CONS\ a_0\ a_1)\\ &\implies DROP\ 0\ (TAKE\ (SUC\ 0)\ (CONS\ a_0\ a_1)) = [EL\ 0\ (CONS\ a_0\ a_1)]\end{aligned} \tag{9.20}$$

$$\begin{aligned}&(\forall\ n.\\ &\quad (\forall\ lst.n < LENGTH\ lst \implies DROP\ n\ (TAKE\ (SUC\ n)\ lst) = [EL\ n\ lst]) \implies\\ &\quad SUC\ n < LENGTH\ a_1 \implies DROP\ (SUC\ n)\ (TAKE\ (SUC\ (SUC\ n))\ a_1) = [EL\ (SUC\ n)\ a_1])\\ &\implies (\forall\ lst.n < LENGTH\ lst \implies DROP\ n\ (TAKE\ (SUC\ n)\ lst) = [EL\ n\ lst])\\ &\implies SUC\ n < LENGTH\ (CONS\ a_0\ a_1)\\ &\implies DROP\ (SUC\ n)\ (TAKE\ (SUC\ (SUC\ n))\ (CONS\ a_0\ a_1)) = [EL\ (SUC\ n)\ (CONS\ a_0\ a_1)]\end{aligned} \tag{9.21}$$

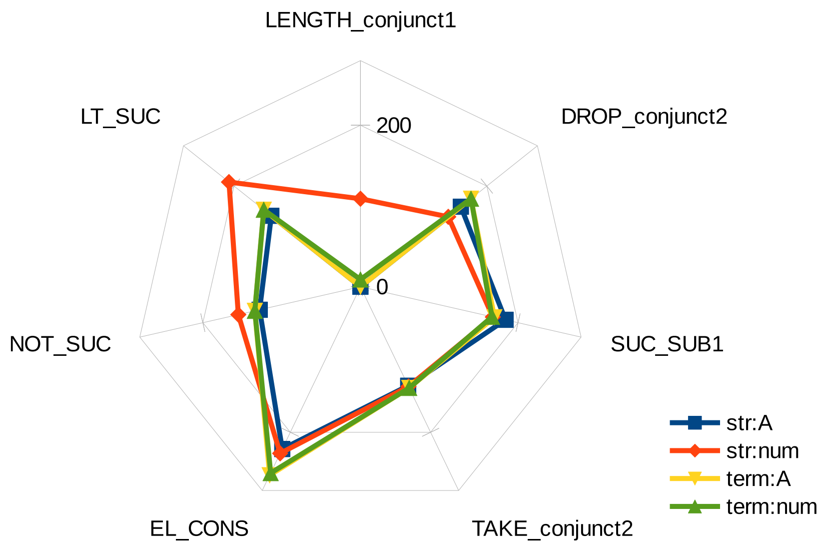
In this experiment, it turns out that all lemmas can be found without changing the type of *lst* (i.e. both *str:A* and *str:num* have all lemmas within top 256, which is the number of selected lemmas). Therefore, we believe that the failure was due to the irrelevant lemmas that slow down ATPs (as discussed in Section 7.3.4).

According to Fig 9.7, most lemmas have similar rankings in different settings. Term features generally provide a better ranking than string features, particularly *str:num*. Moreover, they also give a more consistent ranking: the rankings using term features do not change when the type of *lst* instantiated types.

[3]The other subgoals are proven with fewer than three lemmas



(a) Lemma rankings for the subgoal (9.20)



(b) Lemma rankings for the subgoal (9.21)

Figure 9.7: Lemma ranking comparison among the methods in Table 9.3

The only lemma where term features give a much poorer ranking than string feature is "TAKE_conjunct0". String features give similar rankings for the lemmas "TAKE_-conjunct0" and "TAKE_conjunct2", which are the first and third conjunct in (9.22), because only the string "TAKE" is found in common between the goal and the two lemmas. Term features give "TAKE_conjunct0" a significantly lower rank, by matching term feature "TAKE (SUC A) (CONS A A)". This is actually consistent with our intuition: Only "TAKE_conjunct2" is desired; "TAKE_conjunct0" is only needed after rewriting with "TAKE_conjunct2" (the term then becomes "$TAKE\ 0\ a_1$"). This means that term features do actually give more accurate selection here, or at least, it is not a mistake that the "TAKE_conjunct0" has a poor ranking.

$$\begin{aligned} &TAKE\ 0\ xs = [] \wedge \\ &TAKE\ n\ [] = [] \wedge \\ &TAKE\ (s(n))\ (cons\ x\ xs) = cons\ x\ (TAKE\ n\ xs) \end{aligned} \tag{9.22}$$

This example reveals one of the limitation of current lemma selection method that some of the features are actually not contained in the goal, but in the lemmas. It is not enough to consider lemmas related to the goal but also other lemmas, which was an issue that was looked out for MePo. for *MePo* (Meng and Paulson, 2009).

We also noticed that when trying to prove the second conjunct in Goal 4.2 (see Fig 9.5), term features approach was able to find the common (term) feature "LENGTH (ASG A A A)" and "ASG A A A" between the goal and the lemma `length_list_update` shown as (9.23), while the string features could only find the common string "ASG". Unfortunately, since the incremental feature extraction has not been implemented (see Section 9.4.2), the lemmas about list_update have not been learnt for selection and thus the evaluation is not done for this example.

$$\forall l\ i\ n.\ LENGTH\ (ASG\ l\ i\ n) = LENGTH\ l \tag{9.23}$$

The experiments in this section have shown the some promising results of using term as features. However, despite our expectations, it does not seem as convenient as string features (see Section 9.4.2) and has not shown improvements in our evaluation.

9.5 Conclusion

In this chapter, we test our approaches by looking at the verification of the selection sort algorithm. An issue that arises when proving the termination of nested loops is addressed and solved. Using Boyer-Moore to prove the VC goals in an interactive settings (i.e. split the goal into subgoals), it had rather limited success as many interactions were required during the proof process. Enhancing our Boyer-Moore tool with notions of hints, as in ACL2, may be a worthwhile avenue for future work.

Chapter 10

Conclusion

In this thesis, we investigated the application of machine learning techniques to inductive theorem proving. In this final chapter, we summarise the achievements and the limitations of our work. We then provide some directions for future work.

10.1 Achievements and limitations

Our work is focused on the combination of machine learning methods with inductive theorem proving tools. Existing state-of-the-art proof assistant tools, such as *HOL(y)Hammer* which uses generic first order ATPs, are shown to perform poorly on inductive problems. Therefore, we base our work on the Boyer-Moore system which is tailored towards the automation of inductive proofs. Our main aim is to show that it is possible to improve the automation of inductive proofs in HOL Light. Changes were made to improve the HOL Light implementation of Boyer-Moore for inductive theorem proving. These lead to the creation of a multi-waterfall Boyer-Moore model that incorporates machine learning techniques and various other enhancements.

Other potential applications of machine learning were also investigated, such as selecting suitable variables for induction. Induction for non-recursive data types was also explored e.g. finite sets in HOL Light.

Proof metrics were proposed for measuring the difficulty or "complexity" of the proofs that departed from unreliable criteria such as the length of the proof script.

We evaluated the potential of our Boyer-Moore Multi-waterfall system on various

statements and also applied it to the verification of a sorting algorithm in a Hoare Logic setting.

We breakdown our achievements and limitations from these investigations in the next sections.

10.1.1 Lemma selection for Boyer-Moore Model

The success of an inductive proof procedure depends on two main conditions: the correct choice of induction variable and the selection of the right lemmas while avoiding the irrelevant ones. This becomes dramatically more difficult when a (sub)goal is not proven within the waterfall and induction steps are repeatedly applied.

The biggest challenge we faced was to find a proper way to deal with the large number of lemmas suggested for selection: First, the waterfalls often get stuck when dealing with these lemmas whether with the HOL Light simplifier or *HOL(y)Hammer*. Our empirical *Multi-waterfall Model* (with three waterfalls in our particular case) was used to unblock such waterfalls and was shown effective in proving quite a few inductive theorems that *HOL(y)Hammer* and the original Boyer-Moore implementation failed to prove (see Section 7.1.4). However, sometimes the selected lemmas still slowed down ATPs and made them fail to find the proof within the time-out limit (see Section 7.3.4). We also faced issues in the translation from ITP to ATP problems, which we tried to address (see Section 6.3.4).

Although the hammer systems face similar issues to the above when it comes to lemma selection, they are one-shot procedures that either prove a conjecture or fail there and then. In our case though, the Boyer-Moore still has further actions in the form of the heuristics in the waterfall and the induction procedure at the end. For instance, example (9.15a) in Section 9.3.4 could be proven only if the right rewrite direction was picked.

We investigated term features as a way of getting over our perceived limitations of string features, whereby the latter mismatched instantiated version of the statements from which they were extracted. It has some good results in some examples but a limited improvement compared with string features in general.

To summarize, we have shown that machine learning techniques were able to successfully select lemmas for the Boyer-Moore Model and improved its ability to find inductive proofs.

10.1.2 Machine learning for selecting induction variable

We investigated and developed machine learning methods to select the induction variable. The performance achieved was similar to that of *recursion analysis*, with small differences resulting from the training data (see Section 7.2.2).

This aspect of our research seems to indicate that a good old-fashion heuristic cannot massively be improved upon by a data-driven approach.

10.1.3 Proof metrics

We found that our Boyer-Moore Multi-waterfall model proved theorems whose proofs have more nodes, which indicates that more complicated theorems are proven (see Section 7.1.6). However, the metric we use i.e. the number of nodes counted in manual proof may not always reflect the complexity of the theorem. This was because inductive proofs are sometimes short, even though they are difficult to automate. Moreover, the user sometimes finds a more complicated proof. In order to improve this metric, the difficulty of the automation of the proof should also be considered by looking into the special proof steps (e.g. induction) in it.

10.1.4 Finite sets

We attempted to extend our work to induction over finite sets, which involves non-recursive data types. We tackled the challenge by adapting the Boyer-Moore system (see Section 8.3 and Section 8.4). The system can prove some theorems with induction successfully, but the results were not as good as the experiments on our other data sets. Some of the problems arose because subgoals cut in by the user and special *tactics* are frequently used in proofs (see Chapter 8). More test theorems whose proofs are more likely to be automated may also be helpful for the evaluation of our approach.

10.2 Future work

10.2.1 Machine learning for induction

Future work about improving the application of machine learning to inductive theorem proving should involve more accurate selection of intermediate lemmas and a better way of filtering them to cut down irrelevant ones before calling the ATPs.

10.2.1.1 Tweaking parameters

As mentioned in Section 6.3.1, there are many parameters in our system e.g. the length of the time-outs (for ATP and the whole Boyer-Moore) and the number of lemmas selected for ATPs. We use a fixed value for them in this work. A better performance may be achieved by using different settings. Some parameters may need to be adjusted dynamically e.g. depending on the problem at hand, particularly the number of lemmas sent for ATPs, which is relevant to the number of lemmas available for selection. It may also help to use a time-slice for scheduling the running of the multiple waterfalls so that multi-thread machine is no longer required to run a number of waterfalls in parallel.

10.2.1.2 Efficient Multi-waterfall

Redundant search for a proof may occur when running multiple waterfalls on the same goal, leading to unnecessary steps. In order to reduce such redundancy, collaboration between parallel waterfalls could be considered. For instance, a shared cache could be used like a warehouse of already-seen subgoals, as a means of communication between parallel proof search. If a subgoal has been processed (proven, disproven, or unprovable) in one waterfall, it could be added to the warehouse so that other waterfalls with the same configuration do not need to try it again.

10.2.1.3 Lemma selection

In addition to selecting relevant lemmas, it is also important to filter out irrelevant ones. As mentioned in Section 10.1.1, some lemmas slow down the ATPs. Manual filtering has been made for *HOL(y)Hammer*, which only removes lemmas about logic

symbols such as "¬", "∧", "∀" etc. (Kaliszyk and Urban, 2014). A more systematic approach that might automatically filter out the lemmas that are not helpful for ATPs may improve the performance of the hammer systems.

Different approaches to utilising the lemmas from selection may be considered. We have tried the simplifiers and automated procedures in HOL Light (e.g. `SIMP_CONV` and `MESON`), but they are less powerful compared to ATPs in terms of supporting a large number of lemmas. *Rippling analysis* may help use these lemmas with its powerful ability to control the rewrite direction (see the example in Section 9.3.4).

We have investigated the approach of using *terms as features* instead of strings (see Section 9.4). Currently, further effort is required in order to deploy this in systems such as Boyer-Moore effectively. For instance, incrementally adding features to the training data. One of our test dataset *poly* involves instantiated list type (to the real numbers), i.e. many list variables in this corpus are lists of real numbers rather than polymorphic values, which may yield better results.

SMT solvers may be able to play a more important role. For instance, CVC4 now has a support for induction, so its combination with machine learning could be investigated. However, a different translation would be needed, which directly map the recursive types including the natural numbers for arithmetic to the SMT syntax (rather than the current TPTP syntax) so that they are treated accordingly.

10.2.1.4 Other applications of machine learning

The possibility of learning and predicting the variable for induction has been investigated. Currently it only selects a single variable, because all the proofs for training in HOL Light perform induction on a single variable. This needs to be extended to allow the selection of multiple variables.

It is also possible to select the induction rule e.g. whether a two-step induction rule in Section 2.3.1 should be used. This can also be useful for induction on non-recursive types (e.g. the finite set, see Section 8.1) where the approach of analysing the recursion of arguments does not work.

Other aspects of combining machine learning techniques and induction may also be investigated. Lemma speculation including generalisation may be looked into. Machine learning approaches have been applied to support some of these approaches such as

looking for the theorem with a similar structure and construct the intermediate lemma by analogy (Heras et al., 2013). If proper data are available where the conjecture and its corresponding intermediate lemma (e.g. cut formula) are provided for training, it may be possible to predict intermediate lemmas for new conjectures.

10.2.2 Extending the Boyer-Moore functionality

Other features can be added to improve the Boyer-Moore Model, which may also help improve the performance of machine learning methods.

First, more heuristics can be added to Boyer-Moore waterfall to handle certain kinds of problems. For instance, a heuristic could be added to eliminate conditional expressions resulting from definitions (e.g. (10.1a), see Section 7.1.2).

A better solution is to dynamically construct a waterfall for problems in different domains. For example, we could attempt to employ machine learning techniques to select the appropriate heuristics that can maximize both the effectiveness and efficiency of the waterfall.

$$\forall m\, n.\, \min\ m\ n = if\ m \leq n\ then\ m\ else\ n \tag{10.1a}$$

$$\begin{aligned} \forall m\, n.\ & \min\ 0\ n \wedge \\ & \min\ s(m)\ 0 = 0 \wedge \\ & \min\ s(m)\ s(n) = s(\min\ m\ n) \end{aligned} \tag{10.1b}$$

Some definitions can be replaced with alternative versions (see Section 7.3.3). It would be useful if the Boyer-Moore system can keep a different function definition (10.1b) that helps the recursion analysis particularly for the non-recursive ones e.g. (10.1a). Such definitions are used to both selecting induction variable and simplifying the goal.

It will also be more convenient if the Boyer-Moore system could automatically add corresponding shells for new recursive type definitions. The required information is available but require automation: Most attributes in the shell can be obtained directly from HOL Light e.g. the induction rule is returned by the type defining function and other theorems can be produced by HOL Light functions.

10.3 Concluding Remarks

In this thesis we presented a combination of machine learning techniques with the Boyer-Moore implementation in HOL-Light. The latter has been greatly improved and can now prove some complicated theorems with the lemmas selected by machine learning. This approach can be further improved if the lemmas can be better used e.g. by controlling the rewrite direction and removing the problematic ones that slow down the system. In general, machine learning is helpful for the automation of inductive proofs due to its ability to exploit the human knowledge from existing manual proofs. According to our evaluation, although machine learning sometimes cannot compete with carefully selected human heuristics, it can definitely improve the automation of inductive proofs in a general proof assistant such as HOL Light.

Appendix A

Appendix

A.1 Subgoals from splitting the VC goals in Fig 9.2

```
Goal 1:
`i = LENGTH lst
 ==> SORTED (DROP i lst) /\
     (i = LENGTH lst \/
      (!x. MEM x (TAKE i lst) ==> ALL ((<=) x) (DROP i lst)) /\
      i < LENGTH lst)`

Goal 2.1:
  0 [`~(SUC 0 < i)`]
  1 [`SORTED (DROP i lst)`]
  2 [`i = LENGTH lst`]

`SORTED lst`

Goal 2.2:
  0 [`~(SUC 0 < i)`]
  1 [`SORTED (DROP i lst)`]
  2 [`!x. MEM x (TAKE i lst) ==> ALL ((<=) x) (DROP i lst)`]
  3 [`i < LENGTH lst`]
```

```
`SORTED lst`

Goal 3(easy goal: can be proven by SIMP_TAC[] THEN ARITH_TAC):
`SORTED (DROP i lst) /\
 (i = LENGTH lst \/
  (!x. MEM x (TAKE i lst) ==> ALL ((<=) x) (DROP i lst)) /\ i < LENGTH lst) /\
 SUC 0 < i /\
 X = i
 ==> SUC 0 < i /\
     SORTED (DROP i lst) /\
     (i = LENGTH lst \/
      (!x. MEM x (TAKE i lst) ==> ALL ((<=) x) (DROP i lst)) /\
      i < LENGTH lst) /\
     i - 1 <= i - 1 /\
     i - 1 < i /\
     i - 1 < X`

Goal 4.1:
  0 [`~(0 < j)`]
  1 [`SUC 0 < i`]
  2 [`SORTED (DROP i lst)`]
  3 [`ALL ((>=) (EL k lst)) (DROP j (TAKE i lst))`]
  4 [`i = LENGTH lst`]
  5 [`j - 1 <= k`]
  6 [`k < i`]
  7 [`i - 1 < X`]

`SORTED (DROP (i - 1) (ASG (ASG lst k (EL (i - 1) lst)) (i - 1) (EL k lst)))`

Goal 4.2:
  0 [`~(0 < j)`]
  1 [`SUC 0 < i`]
  2 [`SORTED (DROP i lst)`]
  3 [`ALL ((>=) (EL k lst)) (DROP j (TAKE i lst))`]
  4 [`i = LENGTH lst`]
```

```
  5 [`j - 1 <= k`]
  6 [`k < i`]
  7 [`i - 1 < X`]

`i - 1 = LENGTH (ASG (ASG lst k (EL (i - 1) lst)) (i - 1) (EL k lst)) \/
 (!x. MEM x
      (TAKE (i - 1) (ASG (ASG lst k (EL (i - 1) lst)) (i - 1) (EL k lst)))
      ==> ALL ((<=) x)
          (DROP (i - 1)
          (ASG (ASG lst k (EL (i - 1) lst)) (i - 1) (EL k lst)))) /\
 i - 1 < LENGTH (ASG (ASG lst k (EL (i - 1) lst)) (i - 1) (EL k lst))`

Goal 4.3:
  0 [`~(0 < j)`]
  1 [`SUC 0 < i`]
  2 [`SORTED (DROP i lst)`]
  3 [`ALL ((>=) (EL k lst)) (DROP j (TAKE i lst))`]
  4 [`!x. MEM x (TAKE i lst) ==> ALL ((<=) x) (DROP i lst)`]
  5 [`i < LENGTH lst`]
  6 [`j - 1 <= k`]
  7 [`k < i`]
  8 [`i - 1 < X`]

`SORTED (DROP (i - 1) (ASG (ASG lst k (EL (i - 1) lst)) (i - 1) (EL k lst)))`

Goal 4.4:
  0 [`~(0 < j)`]
  1 [`SUC 0 < i`]
  2 [`SORTED (DROP i lst)`]
  3 [`ALL ((>=) (EL k lst)) (DROP j (TAKE i lst))`]
  4 [`!x. MEM x (TAKE i lst) ==> ALL ((<=) x) (DROP i lst)`]
  5 [`i < LENGTH lst`]
  6 [`j - 1 <= k`]
  7 [`k < i`]
  8 [`i - 1 < X`]
```

```
`i - 1 = LENGTH (ASG (ASG lst k (EL (i - 1) lst)) (i - 1) (EL k lst)) \/
 (!x. MEM x
      (TAKE (i - 1) (ASG (ASG lst k (EL (i - 1) lst)) (i - 1) (EL k lst)))
      ==> ALL ((<=) x)
          (DROP (i - 1)
          (ASG (ASG lst k (EL (i - 1) lst)) (i - 1) (EL k lst)))) /\
 i - 1 < LENGTH (ASG (ASG lst k (EL (i - 1) lst)) (i - 1) (EL k lst))`

Goal 5.1:
  0 [`SUC 0 < i`]
  1 [`SORTED (DROP i lst)`]
  2 [`j = i`]
  3 [`i = LENGTH lst`]
  4 [`j - 1 <= k`]
  5 [`k < i`]
  6 [`i - 1 < X`]
  7 [`0 < j`]
  8 [`X' = j`]
  9 [`EL k lst <= EL (j - 1) lst`]

`j - 1 = i \/ ALL ((>=) (EL (j - 1) lst)) (DROP (j - 1) (TAKE i lst))`

Goal 5.2:
  0 [`SUC 0 < i`]
  1 [`SORTED (DROP i lst)`]
  2 [`j = i`]
  3 [`!x. MEM x (TAKE i lst) ==> ALL ((<=) x) (DROP i lst)`]
  4 [`i < LENGTH lst`]
  5 [`j - 1 <= k`]
  6 [`k < i`]
  7 [`i - 1 < X`]
  8 [`0 < j`]
  9 [`X' = j`]
 10 [`EL k lst <= EL (j - 1) lst`]
```

```
`j - 1 = i \/ ALL ((>=) (EL (j - 1) lst)) (DROP (j - 1) (TAKE i lst))`

Goal 5.3:
  0 [`SUC 0 < i`]
  1 [`SORTED (DROP i lst)`]
  2 [`ALL ((>=) (EL k lst)) (DROP j (TAKE i lst))`]
  3 [`i = LENGTH lst`]
  4 [`j - 1 <= k`]
  5 [`k < i`]
  6 [`i - 1 < X`]
  7 [`0 < j`]
  8 [`X' = j`]
  9 [`EL k lst <= EL (j - 1) lst`]

`j - 1 = i \/ ALL ((>=) (EL (j - 1) lst)) (DROP (j - 1) (TAKE i lst))`

Goal 5.4:
  0 [`SUC 0 < i`]
  1 [`SORTED (DROP i lst)`]
  2 [`ALL ((>=) (EL k lst)) (DROP j (TAKE i lst))`]
  3 [`!x. MEM x (TAKE i lst) ==> ALL ((<=) x) (DROP i lst)`]
  4 [`i < LENGTH lst`]
  5 [`j - 1 <= k`]
  6 [`k < i`]
  7 [`i - 1 < X`]
  8 [`0 < j`]
  9 [`X' = j`]
 10 [`EL k lst <= EL (j - 1) lst`]

`j - 1 = i \/ ALL ((>=) (EL (j - 1) lst)) (DROP (j - 1) (TAKE i lst))`

Goal 6.1:
  0 [`SUC 0 < i`]
  1 [`SORTED (DROP i lst)`]
```

```
  2 [`j = i`]
  3 [`i = LENGTH lst`]
  4 [`j - 1 <= k`]
  5 [`k < i`]
  6 [`i - 1 < X`]
  7 [`0 < j`]
  8 [`X' = j`]
  9 [`~(EL k lst <= EL (j - 1) lst)`]

`j - 1 = i \/ ALL ((>=) (EL k lst)) (DROP (j - 1) (TAKE i lst))`

Goal 6.2:
  0 [`SUC 0 < i`]
  1 [`SORTED (DROP i lst)`]
  2 [`j = i`]
  3 [`!x. MEM x (TAKE i lst) ==> ALL ((<=) x) (DROP i lst)`]
  4 [`i < LENGTH lst`]
  5 [`j - 1 <= k`]
  6 [`k < i`]
  7 [`i - 1 < X`]
  8 [`0 < j`]
  9 [`X' = j`]
 10 [`~(EL k lst <= EL (j - 1) lst)`]

`j - 1 = i \/ ALL ((>=) (EL k lst)) (DROP (j - 1) (TAKE i lst))`

Goal 6.3:
  0 [`SUC 0 < i`]
  1 [`SORTED (DROP i lst)`]
  2 [`ALL ((>=) (EL k lst)) (DROP j (TAKE i lst))`]
  3 [`i = LENGTH lst`]
  4 [`j - 1 <= k`]
  5 [`k < i`]
  6 [`i - 1 < X`]
  7 [`0 < j`]
```

```
  8 [`X' = j`]
  9 [`~(EL k lst <= EL (j - 1) lst)`]

`j - 1 = i \/ ALL ((>=) (EL k lst)) (DROP (j - 1) (TAKE i lst))`

Goal 6.4:
  0 [`SUC 0 < i`]
  1 [`SORTED (DROP i lst)`]
  2 [`ALL ((>=) (EL k lst)) (DROP j (TAKE i lst))`]
  3 [`!x. MEM x (TAKE i lst) ==> ALL ((<=) x) (DROP i lst)`]
  4 [`i < LENGTH lst`]
  5 [`j - 1 <= k`]
  6 [`k < i`]
  7 [`i - 1 < X`]
  8 [`0 < j`]
  9 [`X' = j`]
 10 [`~(EL k lst <= EL (j - 1) lst)`]

`j - 1 = i \/ ALL ((>=) (EL k lst)) (DROP (j - 1) (TAKE i lst))`
```

A.2 Testing goal for list theory

A.2.1 List core

```
HD:|- HD (CONS h t) = h
TL:|- TL (CONS h t) = t
APPEND_conjunct0:|- !l. APPEND [] l = l
APPEND_conjunct1:|- !h t l. APPEND (CONS h t) l = CONS h (APPEND t l)
REVERSE_conjunct0:|- REVERSE [] = []
REVERSE_conjunct1:|- REVERSE (CONS x l) = APPEND (REVERSE l) [x]
LENGTH_conjunct0:|- LENGTH [] = 0
LENGTH_conjunct1:|- !h t. LENGTH (CONS h t) = SUC (LENGTH t)
MAP_conjunct0:|- !f. MAP f [] = []
MAP_conjunct1:|- !f h t. MAP f (CONS h t) = CONS (f h) (MAP f t)
```

```
LAST:|- LAST (CONS h t) = (if t = [] then h else LAST t)
BUTLAST_conjunct0:|- BUTLAST [] = []
BUTLAST_conjunct1:|- BUTLAST (CONS h t) = (if t = [] then [] else CONS h (BUTLAST t))
REPLICATE_conjunct0:|- REPLICATE 0 x = []
REPLICATE_conjunct1:|- REPLICATE (SUC n) x = CONS x (REPLICATE n x)
NULL_conjunct0:|- NULL [] <=> T
NULL_conjunct1:|- NULL (CONS h t) <=> F
ALL_conjunct0:|- ALL P [] <=> T
ALL_conjunct1:|- ALL P (CONS h t) <=> P h /\ ALL P t
EX_conjunct0:|- EX P [] <=> F
EX_conjunct1:|- EX P (CONS h t) <=> P h \/ EX P t
ITLIST_conjunct0:|- ITLIST f [] b = b
ITLIST_conjunct1:|- ITLIST f (CONS h t) b = f h (ITLIST f t b)
MEM_conjunct0:|- MEM x [] <=> F
MEM_conjunct1:|- MEM x (CONS h t) <=> x = h \/ MEM x t
ALL2_DEF_conjunct0:|- ALL2 P [] l2 <=> l2 = []
ALL2_DEF_conjunct1:|- ALL2 P (CONS h1 t1) l2 <=>
   (if l2 = [] then F else P h1 (HD l2) /\ ALL2 P t1 (TL l2))
ALL2_conjunct0:|- ALL2 P [] [] <=> T
ALL2_conjunct1:|- ALL2 P (CONS h1 t1) [] <=> F
ALL2_conjunct2:|- ALL2 P [] (CONS h2 t2) <=> F
ALL2_conjunct3:|- ALL2 P (CONS h1 t1) (CONS h2 t2) <=>
                  P h1 h2 /\ ALL2 P t1 t2
MAP2_DEF_conjunct0:|- MAP2 f [] l = []
MAP2_DEF_conjunct1:|- MAP2 f (CONS h1 t1) l =
                    CONS (f h1 (HD l)) (MAP2 f t1 (TL l))
MAP2_conjunct0:|- MAP2 f [] [] = []
MAP2_conjunct1:|- MAP2 f (CONS h1 t1) (CONS h2 t2) =
                    CONS (f h1 h2) (MAP2 f t1 t2)
EL_conjunct0:|- EL 0 l = HD l
EL_conjunct1:|- EL (SUC n) l = EL n (TL l)
FILTER_conjunct0:|- FILTER P [] = []
FILTER_conjunct1:|- FILTER P (CONS h t) =
                    (if P h then CONS h (FILTER P t) else FILTER P t)
ASSOC:|- ASSOC a (CONS h t) = (if FST h = a then SND h else ASSOC a t)
```

```
ITLIST2_DEF_conjunct0:|- ITLIST2 f [] l2 b = b
ITLIST2_DEF_conjunct1:|- ITLIST2 f (CONS h1 t1) l2 b =
                         f h1 (HD l2) (ITLIST2 f t1 (TL l2) b)
ITLIST2_conjunct0:|- ITLIST2 f [] [] b = b
ITLIST2_conjunct1:|- ITLIST2 f (CONS h1 t1) (CONS h2 t2) b =
                         f h1 h2 (ITLIST2 f t1 t2 b)
ZIP_DEF_conjunct0:|- ZIP [] l2 = []
ZIP_DEF_conjunct1:|- ZIP (CONS h1 t1) l2 = CONS (h1,HD l2) (ZIP t1 (TL l2))
ZIP_conjunct0:|- ZIP [] [] = []
ZIP_conjunct1:|- ZIP (CONS h1 t1) (CONS h2 t2) = CONS (h1,h2) (ZIP t1 t2)
PAIRWISE_conjunct0:|- PAIRWISE r [] <=> T
PAIRWISE_conjunct1:|- PAIRWISE r (CONS h t) <=> ALL (r h) t /\ PAIRWISE r t
list_of_seq_conjunct0:|- list_of_seq s 0 = []
list_of_seq_conjunct1:|- list_of_seq s (SUC n) =
                         APPEND (list_of_seq s n) [s n]
NOT_CONS_NIL:|- !h t. ~(CONS h t = [])
LAST_CLAUSES_conjunct0:|- LAST [h] = h
LAST_CLAUSES_conjunct1:|- LAST (CONS h (CONS k t)) = LAST (CONS k t)
APPEND_NIL:|- !l. APPEND l [] = l
APPEND_ASSOC:|- !l m n. APPEND l (APPEND m n) = APPEND (APPEND l m) n
REVERSE_APPEND:|- !l m. REVERSE (APPEND l m) = APPEND (REVERSE m) (REVERSE l)
REVERSE_REVERSE:|- !l. REVERSE (REVERSE l) = l
CONS_11:|- !h1 h2 t1 t2. CONS h1 t1 = CONS h2 t2 <=> h1 = h2 /\ t1 = t2
list_CASES:|- !l. l = [] \/ (?h t. l = CONS h t)
LIST_EQ:|- !l1 l2.
       l1 = l2 <=>
       LENGTH l1 = LENGTH l2 /\ (!n. n < LENGTH l2 ==> EL n l1 = EL n l2)
LENGTH_APPEND:|- !l m. LENGTH (APPEND l m) = LENGTH l + LENGTH m
MAP_APPEND:|- !f l1 l2. MAP f (APPEND l1 l2) = APPEND (MAP f l1) (MAP f l2)
LENGTH_MAP:|- !l f. LENGTH (MAP f l) = LENGTH l
LENGTH_EQ_NIL:|- !l. LENGTH l = 0 <=> l = []
LENGTH_EQ_CONS:|- !l n. LENGTH l = SUC n <=>
               (?h t. l = CONS h t /\ LENGTH t = n)
MAP_o:|- !f g l. MAP (g o f) l = MAP g (MAP f l)
MAP_EQ:|- !f g l. ALL (\x. f x = g x) l ==> MAP f l = MAP g l
```

```
ALL_IMP:|- !P Q l. (!x. MEM x l /\ P x ==> Q x) /\ ALL P l ==> ALL Q l
NOT_EX:|- !P l. ~EX P l <=> ALL (\x. ~P x) l
NOT_ALL:|- !P l. ~ALL P l <=> EX (\x. ~P x) l
ALL_MAP:|- !P f l. ALL P (MAP f l) <=> ALL (P o f) l
ALL_T:|- !l. ALL (\x. T) l
MAP_EQ_ALL2:|- !l m. ALL2 (\x y. f x = f y) l m ==> MAP f l = MAP f m
ALL2_MAP:|- !P f l. ALL2 P (MAP f l) l <=> ALL (\a. P (f a) a) l
MAP_EQ_DEGEN:|- !l f. ALL (\x. f x = x) l ==> MAP f l = l
ALL2_AND_RIGHT:|- !l m P Q. ALL2 (\x y. P x /\ Q x y) l m <=> ALL P l /\ ALL2 Q l m
ITLIST_APPEND:|- !f a l1 l2. ITLIST f (APPEND l1 l2) a = ITLIST f l1 (ITLIST f l2 a)
ITLIST_EXTRA:|- !l. ITLIST f (APPEND l [a]) b = ITLIST f l (f a b)
ALL_MP:|- !P Q l. ALL (\x. P x ==> Q x) l /\ ALL P l ==> ALL Q l
AND_ALL:|- !l. ALL P l /\ ALL Q l <=> ALL (\x. P x /\ Q x) l
EX_IMP:|- !P Q l. (!x. MEM x l /\ P x ==> Q x) /\ EX P l ==> EX Q l
ALL_MEM:|- !P l. (!x. MEM x l ==> P x) <=> ALL P l
LENGTH_REPLICATE:|- !n x. LENGTH (REPLICATE n x) = n
EX_MAP:|- !P f l. EX P (MAP f l) <=> EX (P o f) l
EXISTS_EX:|- !P l. (?x. EX (P x) l) <=> EX (\s. ?x. P x s) l
FORALL_ALL:|- !P l. (!x. ALL (P x) l) <=> ALL (\s. !x. P x s) l
MEM_APPEND:|- !x l1 l2. MEM x (APPEND l1 l2) <=> MEM x l1 \/ MEM x l2
MEM_MAP:|- !f y l. MEM y (MAP f l) <=> (?x. MEM x l /\ y = f x)
FILTER_APPEND:|- !P l1 l2. FILTER P (APPEND l1 l2) =
                           APPEND (FILTER P l1) (FILTER P l2)
FILTER_MAP:|- !P f l. FILTER P (MAP f l) = MAP f (FILTER (P o f) l)
MEM_FILTER:|- !P l x. MEM x (FILTER P l) <=> P x /\ MEM x l
EX_MEM:|- !P l. (?x. P x /\ MEM x l) <=> EX P l
MAP_FST_ZIP:|- !l1 l2. LENGTH l1 = LENGTH l2 ==> MAP FST (ZIP l1 l2) = l1
MAP_SND_ZIP:|- !l1 l2. LENGTH l1 = LENGTH l2 ==> MAP SND (ZIP l1 l2) = l2
LENGTH_ZIP:|- !l1 l2. LENGTH l1 = LENGTH l2 ==> LENGTH (ZIP l1 l2) = LENGTH l2
MEM_ASSOC:|- !l x. MEM (x,ASSOC x l) l <=> MEM x (MAP FST l)
ALL_APPEND:|- !P l1 l2. ALL P (APPEND l1 l2) <=> ALL P l1 /\ ALL P l2
MEM_EL:|- !l n. n < LENGTH l ==> MEM (EL n l) l
MEM_EXISTS_EL:|- !l x. MEM x l <=> (?i. i < LENGTH l /\ x = EL i l)
ALL_EL:|- !P l. (!i. i < LENGTH l ==> P (EL i l)) <=> ALL P l
ALL2_MAP2:|- !l m. ALL2 P (MAP f l) (MAP g m) <=>
```

```
                 ALL2 (\x y. P (f x) (g y)) l m
AND_ALL2:|- !P Q l m. ALL2 P l m /\ ALL2 Q l m <=>
                    ALL2 (\x y. P x y /\ Q x y) l m
ALL2_ALL:|- !P l. ALL2 P l l <=> ALL (\x. P x x) l
APPEND_EQ_NIL:|- !l m. APPEND l m = [] <=> l = [] /\ m = []
APPEND_LCANCEL:|- !l1 l2 l3. APPEND l1 l2 = APPEND l1 l3 <=> l2 = l3
APPEND_RCANCEL:|- !l1 l2 l3. APPEND l1 l3 = APPEND l2 l3 <=> l1 = l2
LENGTH_MAP2:|- !f l m. LENGTH l = LENGTH m ==>
                     LENGTH (MAP2 f l m) = LENGTH m
MAP_EQ_NIL:|- !f l. MAP f l = [] <=> l = []
INJECTIVE_MAP:|- !f. (!l m. MAP f l = MAP f m ==> l = m) <=>
                  (!x y. f x = f y ==> x = y)
SURJECTIVE_MAP:|- !f. (!m. ?l. MAP f l = m) <=> (!y. ?x. f x = y)
MAP_ID:|- !l. MAP (\x. x) l = l
MAP_I:|- MAP I = I
BUTLAST_APPEND:|- !xs ys. ~(ys = []) ==> BUTLAST (APPEND xs ys) =
                           APPEND xs (BUTLAST ys)
APPEND_BUTLAST_LAST:|- !l. ~(l = []) ==> APPEND (BUTLAST l) [LAST l] = l
LAST_APPEND:|- !p q. LAST (APPEND p q) = (if q = [] then LAST p else LAST q)
LENGTH_TL:|- !l. ~(l = []) ==> LENGTH (TL l) = LENGTH l - 1
EL_APPEND:|- !k l m.
       EL k (APPEND l m) =
       (if k < LENGTH l then EL k l else EL (k - LENGTH l) m)
EL_TL:|- !n. EL n (TL l) = EL (n + 1) l
EL_CONS:|- !n h t. EL n (CONS h t) = (if n = 0 then h else EL (n - 1) t)
LAST_EL:|- !l. ~(l = []) ==> LAST l = EL (LENGTH l - 1) l
HD_APPEND:|- !l m. HD (APPEND l m) = (if l = [] then HD m else HD l)
CONS_HD_TL:|- !l. ~(l = []) ==> l = CONS (HD l) (TL l)
EL_MAP:|- !f n l. n < LENGTH l ==> EL n (MAP f l) = f (EL n l)
MAP_REVERSE:|- !f l. REVERSE (MAP f l) = MAP f (REVERSE l)
ALL_FILTER:|- !P Q l. ALL P (FILTER Q l) <=> ALL (\x. Q x ==> P x) l
APPEND_SING:|- !h t. APPEND [h] t = CONS h t
MEM_APPEND_DECOMPOSE_LEFT:|- !x l. MEM x l <=>
                             (?l1 l2. ~MEM x l1 /\ l = APPEND l1 (CONS x l2))
MEM_APPEND_DECOMPOSE:|- !x l. MEM x l <=> (?l1 l2. l = APPEND l1 (CONS x l2))
```

```
PAIRWISE_APPEND:|- !P xs ys.
       PAIRWISE P (APPEND xs ys) <=>
       PAIRWISE P xs /\ (!x. MEM x xs ==> ALL (P x) ys) /\ PAIRWISE P ys
PAIRWISE_MAP:|- !R f l. PAIRWISE R (MAP f l) <=> PAIRWISE (\x y. R (f x) (f y)) l
PAIRWISE_IMPLIES:|- !R R' l.
       PAIRWISE R l /\ (!x y. MEM x l /\ MEM y l /\ R x y ==> R' x y)
       ==> PAIRWISE R' l
PAIRWISE_TRANSITIVE:|- !R x y l.
       (!x y z. R x y /\ R y z ==> R x z)
       ==> (PAIRWISE R (CONS x (CONS y l)) <=>
            R x y /\ PAIRWISE R (CONS y l))
LENGTH_LIST_OF_SEQ:|- !s n. LENGTH (list_of_seq s n) = n
EL_LIST_OF_SEQ:|- !s m n. m < n ==> EL m (list_of_seq s n) = s m
LIST_OF_SEQ_EQ_NIL:|- !s n. list_of_seq s n = [] <=> n = 0
MONO_ALL:|- (!x. P x ==> Q x) ==> ALL P l ==> ALL Q l
MONO_ALL2:|- (!x y. P x y ==> Q x y) ==> ALL2 P l l' ==> ALL2 Q l l'
char_INDUCT:|- !P. (!a0 a1 a2 a3 a4 a5 a6 a7. P (ASCII a0 a1 a2 a3 a4 a5 a6 a7))
       ==> (!x. P x)
char_RECURSION:|- !f. ?fn. !a0 a1 a2 a3 a4 a5 a6 a7.
                fn (ASCII a0 a1 a2 a3 a4 a5 a6 a7) =
                f a0 a1 a2 a3 a4 a5 a6 a7
```

A.2.2 List Hilbert

```
NOT_EMPTY_EXISTS:|- !xs. ~(xs = []) <=> (?h t. xs = CONS h t)
ONE_ONE_INDUCT:|- !f g P. (!x. g (f x) = x) /\ (!x. P (g x)) ==> (!x. P x)
LIST_INDUCT_APPEND:|- !P. P [] /\ (!x xs. P xs ==>
                                   P (APPEND xs [x])) ==> (!xs. P xs)
LIST_INDUCT2:|- !P. P [] /\
       (!x. P [x] /\ (!y ys. P (CONS y ys) ==> P (CONS x (CONS y ys))))
       ==> (!xs. P xs)
APPEND_EQ:|- !xs ys us vs.
       LENGTH xs = LENGTH us
       ==> (APPEND xs ys = APPEND us vs <=> xs = us /\ ys = vs)
```

```
EL_EQ:|- !xs ys.
       LENGTH xs = LENGTH ys
       ==> ((!n. n < LENGTH xs ==> EL n xs = EL n ys) <=> xs = ys)
EL_EQ_IMP:|- !xs ys.
       LENGTH xs = LENGTH ys /\ (!n. n < LENGTH xs ==> EL n xs = EL n ys)
       ==> xs = ys
HEAD_conjunct0:|- HEAD [] = []
HEAD_conjunct1:|- HEAD (CONS x xs) = [x]
TAIL_conjunct0:|- TAIL [] = []
TAIL_conjunct1:|- TAIL (CONS x xs) = xs
ADJACENT:|- !xs. ADJACENT xs = ZIP (BUTLAST xs) (TAIL xs)
BREAK_ACC_conjunct0:|- BREAK_ACC p [] acc = REVERSE acc,[]
BREAK_ACC_conjunct1:|- BREAK_ACC p (CONS x xs) acc =
   (if p x then REVERSE acc,CONS x xs else BREAK_ACC p xs (CONS x acc))
IS_PREFIX_OF_conjunct0:|- !ys. IS_PREFIX_OF [] ys <=> T
IS_PREFIX_OF_conjunct1:|- !x xs. IS_PREFIX_OF (CONS x xs) [] <=> F
IS_PREFIX_OF_conjunct2:|- !x y xs ys.
       IS_PREFIX_OF (CONS x xs) (CONS y ys) <=> x = y /\ IS_PREFIX_OF xs ys
TAKE_conjunct0:|- TAKE 0 xs = []
TAKE_conjunct1:|- TAKE n [] = []
TAKE_conjunct2:|- TAKE (SUC n) (CONS x xs) = CONS x (TAKE n xs)
DROP_conjunct0:|- DROP 0 xs = xs
DROP_conjunct1:|- DROP n [] = []
DROP_conjunct2:|- DROP (SUC n) (CONS x xs) = DROP n xs
BREAK:|- BREAK p xs = BREAK_ACC p xs []
LENGTH_TAIL:|- !xs. LENGTH (TAIL xs) = PRE (LENGTH xs)
EL_TAIL:|- !n xs. SUC n < LENGTH xs ==> EL n (TAIL xs) = EL (SUC n) xs
TAIL_APPEND:|- !xs ys. ~(xs = []) ==> TAIL (APPEND xs ys) = APPEND (TAIL xs) ys
LENGTH_REVERSE:|- !xs. LENGTH (REVERSE xs) = LENGTH xs
REVERSE_EQ_NIL:|- !xs. REVERSE xs = [] <=> xs = []
TAIL_REVERSE:|- !xs. TAIL (REVERSE xs) = REVERSE (BUTLAST xs)
REVERSE_EQ:|- !xs ys. REVERSE xs = REVERSE ys <=> xs = ys
APPEND_EQ_2:|- !xs ys vs us.
       LENGTH ys = LENGTH vs
       ==> (APPEND xs ys = APPEND us vs <=> xs = us /\ ys = vs)
```

```
HD_REVERSE:|- !xs. ~(xs = []) ==> HD (REVERSE xs) = LAST xs
LAST_REVERSE:|- !xs. ~(xs = []) ==> LAST (REVERSE xs) = HD xs
BUTLAST_APPEND:|- !xs ys. ~(ys = []) ==>
                          BUTLAST (APPEND xs ys) = APPEND xs (BUTLAST ys)
BUTLAST_LENGTH:|- !xs. LENGTH (BUTLAST xs) = PRE (LENGTH xs)
BUTLAST_REVERSE:|- !xs. BUTLAST (REVERSE xs) = REVERSE (TAIL xs)
MEM_BUTLAST:|- !x xs. ~(xs = []) ==>
                       (MEM x xs <=> LAST xs = x \/ MEM x (BUTLAST xs))
MEM_HD:|- !xs. ~(xs = []) ==> MEM (HD xs) xs
MEM_LAST:|- !xs. ~(xs = []) ==> MEM (LAST xs) xs
MEM_REVERSE:|- !p xs. MEM p (REVERSE xs) <=> MEM p xs
ALL_REVERSE:|- ALL p (REVERSE xs) <=> ALL p xs
EX_REVERSE:|- EX p (REVERSE xs) <=> EX p xs
ZIP_APPEND:|- !xs ys us vs.
       LENGTH xs = LENGTH us /\ LENGTH ys = LENGTH vs
       ==> ZIP (APPEND xs ys) (APPEND us vs) = APPEND (ZIP xs us) (ZIP ys vs)
ZIP_REVERSE:|- !xs ys.
       LENGTH xs = LENGTH ys
       ==> REVERSE (ZIP xs ys) = ZIP (REVERSE xs) (REVERSE ys)
MEM_REVERSE1:|- !xs x. MEM x (REVERSE xs) <=> MEM x xs
ZIP_SWAP:|- !xs ys.
       LENGTH xs = LENGTH ys ==> ZIP xs ys = MAP (\(x,y). y,x) (ZIP ys xs)
APPEND_HEAD_TAIL:|- !xs. ~(xs = []) ==> APPEND [HD xs] (TAIL xs) = xs
ADJACENT_APPEND:|- !xs ys.
       ~(xs = []) /\ ~(ys = [])
       ==> ADJACENT (APPEND xs ys) =
           APPEND (ADJACENT xs) (APPEND [LAST xs,HD ys] (ADJACENT ys))
ADJACENT_CONS:|- !x y t. ADJACENT (CONS x (CONS y t)) =
                         CONS (x,y) (ADJACENT (CONS y t))
ADJACENT_CLAUSES_conjunct0:|- !x y t. ADJACENT [] = []
ADJACENT_CLAUSES_conjunct1:|- !x y t. ADJACENT [x] = []
ADJACENT_CLAUSES_conjunct2:|- !x y t. ADJACENT (CONS x (CONS y t)) =
                                      CONS (x,y) (ADJACENT (CONS y t))
ADJACENT_MEM2:|- !xs x y. MEM (x,y) (ADJACENT xs) ==> MEM x xs /\ MEM y xs
EL_IS_PREFIX_OF:|- !xs ys n. IS_PREFIX_OF xs ys /\ n < LENGTH xs
```

```
                              ==> EL n xs = EL n ys
LENGTH_IS_PREFIX_OF:|- !xs ys. IS_PREFIX_OF xs ys ==> LENGTH xs <= LENGTH ys
IS_PREFIX_OF_APPEND:|- !xs ys. IS_PREFIX_OF xs (APPEND xs ys)
IS_PREFIX_OF_EXISTS_APPEND:|- !xs ys zs. IS_PREFIX_OF xs ys <=>
                                         (?zs. APPEND xs zs = ys)
IS_PREFIX_OF_ADJACENT:|- !xs ys. IS_PREFIX_OF xs ys ==>
                                   IS_PREFIX_OF (ADJACENT xs) (ADJACENT ys)
TAKE_ADJACENT:|- !xs n. TAKE n (ADJACENT xs) = ADJACENT (TAKE (SUC n) xs)
DROP_ADJACENT:|- !xs n. DROP n (ADJACENT xs) = ADJACENT (DROP n xs)
MEM_IS_PREFIX_OF:|- !xs ys. IS_PREFIX_OF xs ys ==> (!x. MEM x xs ==> MEM x ys)
MEM_ZIP_SWAP:|- !xs ys x y.
       LENGTH xs = LENGTH ys
       ==> (MEM (x,y) (ZIP xs ys) <=> MEM (y,x) (ZIP ys xs))
MEM_ADJACENT_REVERSE:|- !xs x y. MEM (y,x) (ADJACENT (REVERSE xs))
                                  <=> MEM (x,y) (ADJACENT xs)
APPEND_CONS_NOT_NIL:|- !xs ys y. ~(APPEND xs (CONS y ys) = [])
HEAD_TAIL:|- !xs. APPEND (HEAD xs) (TAIL xs) = xs
BREAK_ACC_APPEND_conjunct0:|- !p xs y ys.
       FST (BREAK_ACC p xs (APPEND ys [y])) =
       CONS y (FST (BREAK_ACC p xs ys))
BREAK_ACC_APPEND_conjunct1:|- !p xs y ys.
           SND (BREAK_ACC p xs (APPEND ys [y])) = SND (BREAK_ACC p xs ys)
BREAK_ACC_CONS_conjunct0:|- !p xs y.
            FST (BREAK_ACC p xs [y]) = CONS y (FST (BREAK_ACC p xs []))
BREAK_ACC_CONS_conjunct1:|- !p xs y.
            SND (BREAK_ACC p xs [y]) = SND (BREAK_ACC p xs [])
BREAK_CONS_conjunct0:|- !p x xs.
       FST (BREAK p (CONS x xs)) =
       (if p x then [] else CONS x (FST (BREAK p xs)))
BREAK_CONS_conjunct1:|- !p x xs.
       SND (BREAK p (CONS x xs)) =
       (if p x then CONS x xs else SND (BREAK p xs))
BREAK_APPEND_conjunct0:|- !p xs.
            EX p xs ==> FST (BREAK p (APPEND xs ys)) = FST (BREAK p xs)
BREAK_APPEND_conjunct1:|- !p xs.
```

```
       EX p xs
       ==> SND (BREAK p (APPEND xs ys)) = APPEND (SND (BREAK p xs)) ys
BREAK_FST_ALL:|- !p xs. ALL (\x. ~p x) (FST (BREAK p xs))
BREAK_SND_EX:|- !p xs. EX p xs ==>
                 ~(SND (BREAK p xs) = []) /\ p (HD (SND (BREAK p xs)))
APPEND_BREAK:|- !p xs. APPEND (FST (BREAK p xs)) (SND (BREAK p xs)) = xs
APPEND_HD_TL:|- !xs. ~(xs = []) ==> CONS (HD xs) (TL xs) = xs
ALL_SUBSET:|- !P xs. ALL P xs <=> set_of_list xs SUBSET P
TAKE_APPEND:|- !xs ys. TAKE (LENGTH xs) (APPEND xs ys) = xs
DROP_APPEND:|- !xs ys. DROP (LENGTH xs) (APPEND xs ys) = ys
APPEND_TAKE:|- !n xs ys. APPEND xs (TAKE n ys) =
                         TAKE (LENGTH xs + n) (APPEND xs ys)
TAKE_LEFT_APPEND:|- !xs ys. TAKE (LENGTH xs) (APPEND xs ys) = xs
TAKE_DROP:|- !n xs. APPEND (TAKE n xs) (DROP n xs) = xs
LENGTH_TAKE:|- !xs n. LENGTH (TAKE n xs) = MIN n (LENGTH xs)
LENGTH_DROP:|- !xs n. LENGTH (DROP n xs) = LENGTH xs - n
LENGTH_ADJACENT:|- !xs. LENGTH (ADJACENT xs) = PRE (LENGTH xs)
EL_ADJACENT:|- !n xs. SUC n < LENGTH xs ==>
                      EL n (ADJACENT xs) = EL n xs,EL (SUC n) xs
MEM_EL_ADJACENT:|- !n xs. SUC n < LENGTH xs ==>
                      MEM (EL n xs,EL (SUC n) xs) (ADJACENT xs)
ADJACENT_EQ_NIL:|- !xs. ADJACENT xs = [] <=> xs = [] \/ (?x. xs = [x])
ADJACENT_EQ_CONS:|- !x xs ys.
       SUC 0 < LENGTH xs /\ SUC 0 < LENGTH ys /\ ADJACENT xs = ADJACENT ys
       ==> ADJACENT (CONS x xs) = ADJACENT (CONS x ys)
ADJACENT_EQ:|- !xs ys.
       SUC 0 < LENGTH xs /\ SUC 0 < LENGTH ys
       ==> (ADJACENT xs = ADJACENT ys <=> xs = ys)
REVERSE_EL:|- !xs n. n < LENGTH xs ==>
                       EL n xs = EL (LENGTH xs - SUC n) (REVERSE xs)
DROP_EL:|- !xs m n. n < LENGTH xs - m ==> EL n (DROP m xs) = EL (m + n) xs
TAKE_EL:|- !xs m n. n < MIN m (LENGTH xs) ==> EL n (TAKE m xs) = EL n xs
DROP_EQ_NIL:|- !xs n. LENGTH xs <= n <=> DROP n xs = []
TAKE_ID:|- !xs n. LENGTH xs <= n <=> TAKE n xs = xs
MEM_TAKE:|- !n x xs. MEM x (TAKE n xs) ==> MEM x xs
```

```
MEM_DROP:|- !n x xs. MEM x (DROP n xs) ==> MEM x xs
ITER_conjunct0:|- ITER 0 f x = x
ITER_conjunct1:|- ITER (SUC n) f x = f (ITER n f x)
ITER_INJ:|- !n. ITER n SUC 0 = n
ITER_ADD:|- !n m f x. ITER (n + m) f x = ITER n f (ITER m f x)
ITER_SUC:|- !n f x. ITER (SUC n) f x = ITER n f (f x)
ITER_IND:|- !n P f x. P x /\ (!x. P x ==> P (f x)) ==> P (ITER n f x)
INDUCT_ITER:|- !f g. f 0 = b /\ (!n. f (SUC n) = g (f n))
                               ==> (!n. ITER n g b = f n)
ITER_MAP:|- !f n x g h. (!y. f (g y) = h (f y)) ==>
                                f (ITER n g x) = ITER n h (f x)
ITER_MAP2:|- !f g n. (!x. f x = g x) ==> ITER n f x = ITER n g x
MAXIMUM:|- !ns b n. MEM n ns ==> n <= ITLIST MAX ns b
PAIRWISE_APPEND:|- !P xs ys.
       PAIRWISE P (APPEND xs ys) <=>
       PAIRWISE P xs /\ (!x. MEM x xs ==> ALL (P x) ys) /\ PAIRWISE P ys
ALL_DISTINCT_CARD:|- !xs. PAIRWISE (\x y. ~(x = y)) xs <=>
                      CARD (set_of_list xs) = LENGTH xs
CHOOSE_PREFIX:|- !n xs. n <= LENGTH xs ==>
                      (?ys zs. xs = APPEND ys zs /\ LENGTH ys = n)
CHOOSE_SUFFIX:|- !n xs. n <= LENGTH xs ==>
                      (?ys zs. xs = APPEND ys zs /\ LENGTH zs = n)
BREAK_LAST:|- !p xs.
       BREAK_LAST p xs =
       (let ys,zs = BREAK p (REVERSE xs) in REVERSE zs,REVERSE ys)
LAMBDA_PAIRED:|- !f p. (\(x,y). f x y) p = f (FST p) (SND p)
APPEND_BREAK_LAST:|- !p xs.
         APPEND (FST (BREAK_LAST p xs)) (SND (BREAK_LAST p xs)) = xs
BREAK_LAST_SND_ALL:|- !p xs. ALL (\x. ~p x) (SND (BREAK_LAST p xs))
BREAK_LAST_FST_EX:|- !p xs.
       EX p xs
       ==> ~(FST (BREAK_LAST p xs) = []) /\ p (LAST (FST (BREAK_LAST p xs)))
list_CASES_APPEND:|- !xs. xs = [] \/ (?pre last. xs = APPEND pre [last])
MEM_IS_INFIX:|- !x xs. MEM x xs <=> (?ys zs. xs = APPEND ys (CONS x zs))
MEM_ADJACENT_IS_INFIX:|- !x y xs.
```

```
       MEM (x,y) (ADJACENT xs) <=>
       (?ys zs. xs = APPEND ys (CONS x (CONS y zs)))
ADJACENT_MIDDLE:|- !xs ys m.
       ADJACENT (APPEND xs (CONS m ys)) =
       APPEND (ADJACENT (APPEND xs [m])) (ADJACENT (CONS m ys))
MAP_FST_ADJACENT:|- !Ps. MAP FST (ADJACENT Ps) = BUTLAST Ps
PAIRWISE_INFIX_EQ:|- !xs ws x.
       PAIRWISE (\x y. ~(x = y)) (APPEND xs (CONS x ys))
       ==> APPEND xs (CONS x ys) = APPEND ws (CONS x zs)
       ==> xs = ws /\ ys = zs
rotation:|- !xs ys. rotation xs ys <=> (?n. xs = APPEND (DROP n ys) (TAKE n ys))
APPEND_DROP:|- !n xs ys. n <= LENGTH xs ==>
                 APPEND (DROP n xs) ys = DROP n (APPEND xs ys)
DROP_LENGTH:|- !n xs. LENGTH xs <= n ==> DROP n xs = []
rotation_LENGTH_EQ:|- rotation xs ys ==> LENGTH xs = LENGTH ys
DROP_DROP:|- !n m xs. DROP (n + m) xs = DROP n (DROP m xs)
TAKE_TAKE:|- !n m xs. TAKE (n + m) xs = APPEND (TAKE n xs) (TAKE m (DROP n xs))
DROP_APPEND2:|- !n xs ys.
       DROP n (APPEND xs ys) = APPEND (DROP n xs) (DROP (n - LENGTH xs) ys)
TAKE_APPEND2:|- !n xs ys.
       TAKE n (APPEND xs ys) = APPEND (TAKE n xs) (TAKE (n - LENGTH xs) ys)
rot_of:|- !xs ys. rot_of xs ys <=> (?us vs. xs = APPEND us vs /\ ys = APPEND vs us)
CONS_EQ_EXISTS_CONS:|- !h t xs. xs = CONS h t ==> (?ys. xs = CONS h ys)
APPEND_lemma:|- !as bs xs ys.
       APPEND as bs = APPEND xs ys /\ LENGTH as <= LENGTH xs
       ==> (?es. APPEND as bs = APPEND as (APPEND es ys))
rot_of_refl:|- !xs. rot_of xs xs
rot_of_sym:|- !xs ys. rot_of xs ys <=> rot_of ys xs
```

A.2.3 Poly.ml

```
poly_conjunct0:|- poly [] x = &0
poly_conjunct1:|- poly (CONS h t) x = h + x * poly t x
POLY_CONST:|- !c x. poly [c] x = c
```

```
POLY_X:|- !c x. poly [&0; &1] x = x
poly_add_conjunct0:|- [] ++ l2 = l2
poly_add_conjunct1:|- CONS h t ++ l2 =
   (if l2 = [] then CONS h t else CONS (h + HD l2) (t ++ TL l2))
poly_cmul_conjunct0:|- c ## [] = []
poly_cmul_conjunct1:|- c ## CONS h t = CONS (c * h) (c ## t)
poly_neg:|- neg = (##) (-- &1)
poly_mul_conjunct0:|- [] ** l2 = []
poly_mul_conjunct1:|- CONS h t ** l2 =
   (if t = [] then h ## l2 else h ## l2 ++ CONS (&0) (t ** l2))
poly_exp_conjunct0:|- poly_exp p 0 = [&1]
poly_exp_conjunct1:|- poly_exp p (SUC n) = p ** poly_exp p n
poly_diff_aux_conjunct0:|- poly_diff_aux n [] = []
poly_diff_aux_conjunct1:|- poly_diff_aux n (CONS h t) =
                        CONS (&n * h) (poly_diff_aux (SUC n) t)
poly_diff:|- !l. poly_diff l =
               (if l = [] then [] else poly_diff_aux 1 (TL l))
LENGTH_POLY_DIFF_AUX:|- !l n. LENGTH (poly_diff_aux n l) = LENGTH l
LENGTH_POLY_DIFF:|- !l. LENGTH (poly_diff l) = PRE (LENGTH l)
POLY_ADD_CLAUSES_conjunct0:|- [] ++ p2 = p2
POLY_ADD_CLAUSES_conjunct1:|- p1 ++ [] = p1
POLY_ADD_CLAUSES_conjunct2:|-
          CONS h1 t1 ++ CONS h2 t2 = CONS (h1 + h2) (t1 ++ t2)
POLY_NEG_CLAUSES_conjunct0:|- neg [] = []
POLY_NEG_CLAUSES_conjunct1:|- neg (CONS h t) = CONS (--h) (neg t)
POLY_MUL_CLAUSES_conjunct0:|- [] ** p2 = []
POLY_MUL_CLAUSES_conjunct1:|- [h1] ** p2 = h1 ## p2
POLY_MUL_CLAUSES_conjunct2:|- CONS h1 (CONS k1 t1) ** p2 =
                       h1 ## p2 ++ CONS (&0) (CONS k1 t1 ** p2)
POLY_DIFF_CLAUSES_conjunct0:|- poly_diff [] = []
POLY_DIFF_CLAUSES_conjunct1:|- poly_diff [c] = []
POLY_DIFF_CLAUSES_conjunct2:|- poly_diff (CONS h t) = poly_diff_aux 1 t
POLY_ADD:|- !p1 p2 x. poly (p1 ++ p2) x = poly p1 x + poly p2 x
POLY_CMUL:|- !p c x. poly (c ## p) x = c * poly p x
POLY_NEG:|- !p x. poly (neg p) x = --poly p x
```

```
POLY_MUL:|- !x p1 p2. poly (p1 ** p2) x = poly p1 x * poly p2 x
POLY_EXP:|- !p n x. poly (poly_exp p n) x = poly p x pow n
POLY_DIFF_LEMMA:|- !l n x.
       ((\x. x pow SUC n * poly l x) diffl
        x pow n * poly (poly_diff_aux (SUC n) l) x)
       x
POLY_DIFF:|- !l x. ((\x. poly l x) diffl poly (poly_diff l) x) x
POLY_DIFFERENTIABLE:|- !l x. (\x. poly l x) differentiable x
POLY_CONT:|- !l x. (\x. poly l x) contl x
POLY_IVT_POS:|- !p a b.
       a < b /\ poly p a < &0 /\ poly p b > &0
       ==> (?x. a < x /\ x < b /\ poly p x = &0)
POLY_IVT_NEG:|- !p a b.
       a < b /\ poly p a > &0 /\ poly p b < &0
       ==> (?x. a < x /\ x < b /\ poly p x = &0)
POLY_MVT:|- !p a b.
       a < b
       ==> (?x. a < x /\
                x < b /\
                poly p b - poly p a = (b - a) * poly (poly_diff p) x)
POLY_MVT_ADD:|- !p a x.
       ?y. abs y <= abs x /\
           poly p (a + x) = poly p a + x * poly (poly_diff p) (a + y)
POLY_ADD_RZERO:|- !p. poly (p ++ []) = poly p
POLY_MUL_ASSOC:|- !p q r. poly (p ** q ** r) = poly ((p ** q) ** r)
POLY_EXP_ADD:|- !d n p. poly (poly_exp p (n + d)) =
                        poly (poly_exp p n ** poly_exp p d)
POLY_DIFF_AUX_ADD:|- !p1 p2 n.
       poly (poly_diff_aux n (p1 ++ p2)) =
       poly (poly_diff_aux n p1 ++ poly_diff_aux n p2)
POLY_DIFF_AUX_CMUL:|- !p c n. poly (poly_diff_aux n (c ## p)) =
                              poly (c ## poly_diff_aux n p)
POLY_DIFF_AUX_NEG:|- !p n. poly (poly_diff_aux n (neg p)) =
                           poly (neg (poly_diff_aux n p))
POLY_DIFF_AUX_MUL_LEMMA:|- !p n. poly (poly_diff_aux (SUC n) p) =
```

```
                              poly (poly_diff_aux n p ++ p)
POLY_DIFF_ADD:|- !p1 p2. poly (poly_diff (p1 ++ p2)) =
                         poly (poly_diff p1 ++ poly_diff p2)
POLY_DIFF_CMUL:|- !p c. poly (poly_diff (c ## p)) = poly (c ## poly_diff p)
POLY_DIFF_NEG:|- !p. poly (poly_diff (neg p)) = poly (neg (poly_diff p))
POLY_DIFF_MUL_LEMMA:|- !t h. poly (poly_diff (CONS h t)) =
                             poly (CONS (&0) (poly_diff t) ++ t)
POLY_DIFF_MUL:|- !p1 p2.
       poly (poly_diff (p1 ** p2)) =
       poly (p1 ** poly_diff p2 ++ poly_diff p1 ** p2)
POLY_DIFF_EXP:|- !p n.
       poly (poly_diff (poly_exp p (SUC n))) =
       poly ((&(SUC n) ## poly_exp p n) ** poly_diff p)
POLY_DIFF_EXP_PRIME:|- !n a.
       poly (poly_diff (poly_exp [--a; &1] (SUC n))) =
       poly (&(SUC n) ## poly_exp [--a; &1] n)
POLY_LINEAR_REM:|- !t h. ?q r. CONS h t = [r] ++ [--a; &1] ** q
POLY_LINEAR_DIVIDES:|- !a p. poly p a = &0 <=>
                             p = [] \/ (?q. p = [--a; &1] ** q)
POLY_LENGTH_MUL:|- !q. LENGTH ([--a; &1] ** q) = SUC (LENGTH q)
POLY_ROOTS_INDEX_LEMMA:|- !n p.
       ~(poly p = poly []) /\ LENGTH p = n
       ==> (?i. !x. poly p x = &0 ==> (?m. m <= n /\ x = i m))
POLY_ROOTS_INDEX_LENGTH:|- !p. ~(poly p = poly [])
       ==> (?i. !x. poly p x = &0 ==> (?n. n <= LENGTH p /\ x = i n))
POLY_ROOTS_FINITE_LEMMA:|- !p. ~(poly p = poly [])
       ==> (?N i. !x. poly p x = &0 ==> (?n. n < N /\ x = i n))
FINITE_LEMMA:|- !i N P. (!x. P x ==> (?n. n < N /\ x = i n))
                           ==> (?a. !x. P x ==> x < a)
POLY_ROOTS_FINITE:|- !p. ~(poly p = poly []) <=>
       (?N i. !x. poly p x = &0 ==> (?n. n < N /\ x = i n))
POLY_ENTIRE_LEMMA:|- !p q.
       ~(poly p = poly []) /\ ~(poly q = poly [])
       ==> ~(poly (p ** q) = poly [])
POLY_ENTIRE:|- !p q. poly (p ** q) = poly [] <=>
```

```
              poly p = poly [] \/ poly q = poly []
POLY_MUL_LCANCEL:|- !p q r.
       poly (p ** q) = poly (p ** r) <=> poly p = poly [] \/ poly q = poly r
POLY_EXP_EQ_0:|- !p n. poly (poly_exp p n) = poly [] <=> poly p = poly [] /\ ~(n = 0)
POLY_PRIME_EQ_0:|- !a. ~(poly [a; &1] = poly [])
POLY_EXP_PRIME_EQ_0:|- !a n. ~(poly (poly_exp [a; &1] n) = poly [])
POLY_ZERO_LEMMA:|- !h t. poly (CONS h t) = poly [] ==> h = &0 /\ poly t = poly []
POLY_ZERO:|- !p. poly p = poly [] <=> ALL (\c. c = &0) p
POLY_DIFF_AUX_ISZERO:|- !p n.
                ALL (\c. c = &0) (poly_diff_aux (SUC n) p) <=> ALL (\c. c = &0) p
POLY_DIFF_ISZERO:|- !p. poly (poly_diff p) = poly [] ==> (?h. poly p = poly [h])
POLY_DIFF_ZERO:|- !p. poly p = poly [] ==> poly (poly_diff p) = poly []
POLY_DIFF_WELLDEF:|- !p q.
                       poly p = poly q ==> poly (poly_diff p) = poly (poly_diff q)
divides:|- !p2 p1. poly_divides p1 p2 <=> (?q. poly p2 = poly (p1 ** q))
POLY_PRIMES:|- !a p q.
       poly_divides [a; &1] (p ** q) <=>
       poly_divides [a; &1] p \/ poly_divides [a; &1] q
POLY_DIVIDES_REFL:|- !p. poly_divides p p
POLY_DIVIDES_TRANS:|- !p q r.
                        poly_divides p q /\ poly_divides q r ==> poly_divides p r
POLY_DIVIDES_EXP:|- !p m n. m <= n ==> poly_divides (poly_exp p m) (poly_exp p n)
POLY_EXP_DIVIDES:|- !p q m n.
       poly_divides (poly_exp p n) q /\ m <= n
       ==> poly_divides (poly_exp p m) q
POLY_DIVIDES_ADD:|- !p q r. poly_divides p q /\ poly_divides p r
                                    ==> poly_divides p (q ++ r)
POLY_DIVIDES_SUB:|- !p q r. poly_divides p q /\ poly_divides p (q ++ r)
                                    ==> poly_divides p r
POLY_DIVIDES_SUB2:|- !p q r. poly_divides p r /\ poly_divides p (q ++ r)
                                    ==> poly_divides p q
POLY_DIVIDES_ZERO:|- !p q. poly p = poly [] ==> poly_divides q p
POLY_ORDER_EXISTS:|- !a d p.
       LENGTH p = d /\ ~(poly p = poly [])
       ==> (?n. poly_divides (poly_exp [--a; &1] n) p /\
```

```
                ~poly_divides (poly_exp [--a; &1] (SUC n)) p)
POLY_ORDER:|- !p a.
       ~(poly p = poly [])
       ==> (?!n. poly_divides (poly_exp [--a; &1] n) p /\
                 ~poly_divides (poly_exp [--a; &1] (SUC n)) p)
order:|- !a p.
       order a p =
       (@n. poly_divides (poly_exp [--a; &1] n) p /\
            ~poly_divides (poly_exp [--a; &1] (SUC n)) p)
ORDER:|- !p a n.
       poly_divides (poly_exp [--a; &1] n) p /\
       ~poly_divides (poly_exp [--a; &1] (SUC n)) p <=>
       n = order a p /\ ~(poly p = poly [])
ORDER_THM:|- !p a.
       ~(poly p = poly [])
       ==> poly_divides (poly_exp [--a; &1] (order a p)) p /\
           ~poly_divides (poly_exp [--a; &1] (SUC (order a p))) p
ORDER_UNIQUE:|- !p a n.
       ~(poly p = poly []) /\
       poly_divides (poly_exp [--a; &1] n) p /\
       ~poly_divides (poly_exp [--a; &1] (SUC n)) p
       ==> n = order a p
ORDER_POLY:|- !p q a. poly p = poly q ==> order a p = order a q
ORDER_ROOT:|- !p a. poly p a = &0 <=> poly p = poly [] \/ ~(order a p = 0)
ORDER_DIVIDES:|- !p a n.
       poly_divides (poly_exp [--a; &1] n) p <=>
       poly p = poly [] \/ n <= order a p
ORDER_DECOMP:|- !p a.
       ~(poly p = poly [])
       ==> (?q. poly p = poly (poly_exp [--a; &1] (order a p) ** q) /\
                ~poly_divides [--a; &1] q)
ORDER_MUL:|- !a p q.
       ~(poly (p ** q) = poly [])
       ==> order a (p ** q) = order a p + order a q
ORDER_DIFF:|- !p a.
```

```
        ~(poly (poly_diff p) = poly []) /\ ~(order a p = 0)
        ==> order a p = SUC (order a (poly_diff p))
POLY_SQUAREFREE_DECOMP_ORDER:|- !p q d e r s.
        ~(poly (poly_diff p) = poly []) /\
        poly p = poly (q ** d) /\
        poly (poly_diff p) = poly (e ** d) /\
        poly d = poly (r ** p ++ s ** poly_diff p)
        ==> (!a. order a q = (if order a p = 0 then 0 else 1))
rsquarefree:|- !p. rsquarefree p <=>
        ~(poly p = poly []) /\ (!a. order a p = 0 \/ order a p = 1)
RSQUAREFREE_ROOTS:|- !p. rsquarefree p <=>
                (!a. ~(poly p a = &0 /\ poly (poly_diff p) a = &0))
RSQUAREFREE_DECOMP:|- !p a.
        rsquarefree p /\ poly p a = &0
        ==> (?q. poly p = poly ([--a; &1] ** q) /\ ~(poly q a = &0))
POLY_SQUAREFREE_DECOMP:|- !p q d e r s.
        ~(poly (poly_diff p) = poly []) /\
        poly p = poly (q ** d) /\
        poly (poly_diff p) = poly (e ** d) /\
        poly d = poly (r ** p ++ s ** poly_diff p)
        ==> rsquarefree q /\ (!a. poly q a = &0 <=> poly p a = &0)
normalize_conjunct0:|- normalize [] = []
normalize_conjunct1:|- normalize (CONS h t) =
    (if normalize t = []
     then if h = &0 then [] else [h]
     else CONS h (normalize t))
POLY_NORMALIZE:|- !p. poly (normalize p) = poly p
degree:|- !p. degree p = PRE (LENGTH (normalize p))
DEGREE_ZERO:|- !p. poly p = poly [] ==> degree p = 0
POLY_ROOTS_FINITE_SET:|- !p.
                    ~(poly p = poly []) ==> FINITE {x | poly p x = &0}
POLY_MONO:|- !x k p. abs x <= k ==> abs (poly p x) <= poly (MAP abs p) k
NOT_POLY_CMUL_NIL:|- !h p. ~(p = []) ==> ~(h ## p = [])
NOT_POLY_MUL_NIL:|- !p1 p2. ~(p1 = []) /\ ~(p2 = []) ==> ~(p1 ** p2 = [])
NOT_POLY_EXP_NIL:|- !n p. ~(p = []) ==> ~(poly_exp p n = [])
```

```
NOT_POLY_EXP_X_NIL:|- !n. ~(poly_exp [&0; &1] n = [])
POLY_CMUL_LID:|- !p. &1 ## p = p
POLY_MUL_LID:|- !p. [&1] ** p = p
POLY_MUL_RID:|- !p. p ** [&1] = p
POLY_ADD_SYM:|- !x y. x ++ y = y ++ x
POLY_ADD_ASSOC:|- !x y z. x ++ y ++ z = (x ++ y) ++ z
TL_POLY_MUL_X:|- !p. TL ([&0; &1] ** p) = p
HD_POLY_MUL_X:|- !p. HD ([&0; &1] ** p) = &0
TL_POLY_EXP_X_SUC:|- !n. TL (poly_exp [&0; &1] (SUC n)) =
                              poly_exp [&0; &1] n
HD_POLY_EXP_X_SUC:|- !n. HD (poly_exp [&0; &1] (SUC n)) = &0
HD_POLY_ADD:|- !p1 p2. ~(p1 = []) /\ ~(p2 = []) ==>
                         HD (p1 ++ p2) = HD p1 + HD p2
HD_POLY_CMUL:|- !x p. ~(p = []) ==> HD (x ## p) = x * HD p
TL_POLY_CMUL:|- !x p. ~(p = []) ==> TL (x ## p) = x ## TL p
HD_POLY_MUL:|- !p1 p2. ~(p1 = []) /\ ~(p2 = []) ==> HD (p1 ** p2) =
                              HD p1 * HD p2
HD_POLY_EXP:|- !n p. ~(p = []) ==> HD (poly_exp p n) = HD p pow n
POLY_ADD_IDENT:|- neutral (++) = []
POLY_ADD_NEUTRAL:|- !x. neutral (++) ++ x = x
MONOIDAL_POLY_ADD:|- monoidal (++)
POLY_DIFF_AUX_ADD_LEMMA:|- !t1 t2 n.
       poly_diff_aux n (t1 ++ t2) = poly_diff_aux n t1 ++ poly_diff_aux n t2
POLYDIFF_ADD:|- !p1 p2. poly_diff (p1 ++ p2) = poly_diff p1 ++ poly_diff p2
POLY_DIFF_AUX_POLY_CMUL:|- !p c n. poly_diff_aux n (c ## p) =
                                    c ## poly_diff_aux n p
POLY_CMUL_POLY_DIFF:|- !p c. poly_diff (c ## p) = c ## poly_diff p
POLY_CMUL_LENGTH:|- !c p. LENGTH (c ## p) = LENGTH p
POLY_ADD_LENGTH:|- !p q. LENGTH (p ++ q) = MAX (LENGTH p) (LENGTH q)
POLY_MUL_LENGTH:|- !p h t. LENGTH (p ** CONS h t) >= LENGTH p
POLY_EXP_X_REC:|- !n. poly_exp [&0; &1] (SUC n) =
                      CONS (&0) (poly_exp [&0; &1] n)
POLY_MUL_LENGTH2:|- !q p. ~(q = []) ==> LENGTH (p ** q) >= LENGTH p
POLY_EXP_X_LENGTH:|- !n. LENGTH (poly_exp [&0; &1] n) = SUC n
POLY_SUM_EQUIV:|- !p x.
```

```
        ~(p = [])
        ==> poly p x = sum (0..PRE (LENGTH p)) (\i. EL i p * x pow i)
ITERATE_RADD_POLYADD:|- !n x f.
        iterate (+) (0..n) (\i. poly (f i) x) = poly (iterate (++) (0..n) f) x
```

A.2.4 Isaplanner benchmark

Definitions (N means natural numbers, different constant names are used for natural numbers and lists, to avoid conflict with the original types in HOL Light

```
N_INDUCT:!P. P Z /\ (!a. P a ==> P (Suc a)) ==> (!x. P x)
N_RECURSION:!f0 f1. ?fn. fn Z = f0 /\ (!a. fn (Suc a) = f1 a (fn a))
List_INDUCT:!P. P Nil /\ (!a0 a1. P a1 ==> P (Cons a0 a1)) ==> (!x. P x)
List_RECURSION:!f0 f1. ?fn. fn Nil = f0 /\
                            (!a0 a1. fn (Cons a0 a1) = f1 a0 a1 (fn a1))
Tree_INDUCT:!P. P Leaf /\
           (!a0 a1 a2. P a0 /\ P a2 ==> P (Node a0 a1 a2)) ==> (!x. P x)
Tree_RECURSION:!f0 f1.
    ?fn. fn Leaf = f0 /\
         (!a0 a1 a2. fn (Node a0 a1 a2) = f1 a1 a0 a2 (fn a0) (fn a2))
add_0:Z ++ y = y /\ Suc x ++ y = Suc (x ++ y)
mult_0:x ** Z = Z /\ x ** Suc y = x ++ x ** y
append:(!ys. append Nil ys = ys) /\
(!x ys xs. append (Cons x xs) ys = Cons x (append xs ys))
lastl:last (Cons x xs) = (if xs = Nil then x else last xs)
butlastl:butlast Nil = Nil /\
butlast (Cons x xs) = (if xs = Nil then Nil else Cons x (butlast xs))
mapl:(!f. map f Nil = Nil) /\
                 (!f x xs. map f (Cons x xs) = Cons (f x) (map f xs))
revl:rev Nil = Nil /\ rev (Cons x xs) = append (rev xs) (Cons x Nil)
filterl:filter P Nil = Nil /\
filter P (Cons x xs) = (if P x then Cons x (filter P xs) else filter P xs)
takeWhile:takeWhile P Nil = Nil /\
takeWhile P (Cons x xs) = (if P x then Cons x (takeWhile P xs) else Nil)
```

```
dropWhile:dropWhile P Nil = Nil /\
dropWhile P (Cons x xs) = (if P x then dropWhile P xs else Cons x xs)
member:(!x. mem x Nil <=> F) /\ (!x. mem x (Cons y ys) <=> x = y \/ mem x ys)
delete:(!x. delete x Nil = Nil) /\
(!x y ys.
     delete x (Cons y ys) =
     (if x = y then delete x ys else Cons y (delete x ys)))
len:len Nil = Z /\ (!h t. len (Cons h t) = Suc (len t))
ins:(!n. ins n Nil = Cons n Nil) /\
(!n h t.
     ins n (Cons h t) =
     (if n less h then Cons n (Cons h t) else Cons h (ins n t)))
ins_1:(!n. ins_1 n Nil = Cons n Nil) /\
(!n h t.
     ins_1 n (Cons h t) = (if n = h then Cons n t else Cons h (ins_1 n t)))
count:(!x. count x Nil = Z) /\
(!x y ys.
     count x (Cons y ys) = (if x = y then Suc (count x ys) else count x ys))
insort:insort x Nil = Cons x Nil /\
insort x (Cons y ys) =
(if x leq y then Cons x (Cons y ys) else Cons y (Cons x ys))
sortl:sort Nil = Nil /\ sort (Cons x xs) = insort x (sort xs)
mirror:mirror Leaf = Leaf /\
mirror (Node l data r) = Node (mirror r) data (mirror l)
nodes:nodes Leaf = Z /\ nodes (Node l data r) = Suc Z ++ nodes l ++ nodes r
height:height Leaf = Z /\ height (Node l data r) =
                              Suc (maxn (height l) (height r))
sorted:(sorted Nil <=> T) /\
(sorted (Cons x Nil) <=> T) /\
(sorted (Cons x (Cons y ys)) <=> x leq y /\ sorted (Cons y ys))
minus:Z minus m = Z /\ Suc m minus Z = Suc m /\ Suc m minus Suc n = m minus n
less:(x less Z <=> F) /\ (Z less Suc y <=> T) /\
                                (Suc z less Suc y <=> z less y)
leq:(Z leq y <=> T) /\ (Suc x leq Z <=> F) /\ (Suc x leq Suc z <=> x leq z)
maxn:maxn Z y = y /\ maxn (Suc x) Z = Suc x /\
```

```
                                maxn (Suc x) (Suc y) = Suc (maxn x y)
minn:minn Z y = Z /\ minn (Suc x) Z = Z /\ minn (Suc x) (Suc y) = Suc (minn x y)
drop:drop n Nil = Nil /\
drop Z (Cons x xs) = Cons x xs /\
drop (Suc m) (Cons x xs) = drop m xs
take:take n Nil = Nil /\
take Z (Cons x xs) = Nil /\
take (Suc m) (Cons x xs) = Cons x (take m xs)
zipl:(!xs. zip xs Nil = Nil) /\
(!y ys. zip Nil (Cons y ys) = Nil) /\
(!z y zs ys. zip (Cons z zs) (Cons y ys) = Cons (z,y) (zip zs ys))
```

Test goals:

```
append (take n xs) (drop n xs) = xs
count n l ++ count n m = count n (append l m)
count n l leq count n (append l (Cons m Nil))
Suc Z ++ count n l = count n (Cons n l)
n = x ==> Suc Z ++ count n l = count n (Cons x l)
n minus (n ++ m) = Z
(n ++ m) minus n = m
(k ++ m) minus (k ++ n) = m minus n
(i minus j) minus k = i minus (j ++ k)
m minus m = Z
drop Z xs = xs
drop n (map f xs) = map f (drop n xs)
drop (Suc n) (Cons x xs) = drop n xs
filter P (append xs ys) = append (filter P xs) (filter P ys)
len (ins x l) = Suc (len l)
xs = Nil ==> last (Cons x xs) = x
n leq Z <=> n = Z
i less Suc (i ++ m)
len (drop n xs) = len xs minus n
len (sort l) = len l
n leq (n ++ m)
maxn (maxn a b) c = maxn a (maxn b c)
```

```
maxn a b = maxn b a
maxn a b = a <=> b leq a
maxn a b = b <=> a leq b
mem x l ==> mem x (append l t)
mem x t ==> mem x (append l t)
mem x (append l (Cons x Nil))
mem x (ins_1 x l)
mem x (ins x l)
minn (minn a b) c = minn a (minn b c)
minn a b = minn b a
minn a b = a <=> a leq b
minn a b = b <=> b leq a
dropWhile (\x. F) xs = xs
takeWhile (\x. T) xs = xs
~mem x (delete x l)
count n (append x (Cons n Nil)) = Suc (count n x)
count n (Cons h Nil) ++ count n t = count n (Cons h t)
take Z xs = Nil
take n (map f xs) = map f (take n xs)
take (Suc n) (Cons x xs) = Cons x (take n xs)
append (takeWhile P xs) (dropWhile P xs) = xs
zip (Cons x xs) ys =
(if ys = Nil then Nil else Cons (x,(@y. Cons y t = ys)) (zip xs ys))
zip (Cons x xs) (Cons y ys) = Cons (x,y) (zip xs ys)
zip Nil ys = Nil
~(xs = Nil) ==> butlast (append xs (Cons (last xs) Nil)) = xs
butlast (append xs ys) =
(if ys = Nil then butlast xs else append xs (butlast ys))
butlast xs = take (len xs minus Suc Z) xs
butlast (append xs (Cons x Nil)) = xs
count n l = count n (rev l)
count x l = count x (sort l)
(m ++ n) minus n = m
(i minus j) minus k = i minus (k ++ j)
drop n (append xs ys) = append (drop n xs) (drop (n minus len xs) ys)
```

```
drop n (drop m xs) = drop (n ++ m) xs
drop n (take m xs) = take (m minus n) (drop n xs)
drop n (zip xs ys) = zip (drop n xs) (drop n ys)
ys = Nil ==> last (append xs ys) = last xs
~(ys = Nil) ==> last (append xs ys) = last ys
last (append xs ys) = (if ys = Nil then last xs else last ys)
~(xs = Nil) ==> last (Cons x xs) = last xs
n less len xs ==> last (drop n xs) = last xs
last (append xs (Cons x Nil)) = x
i less Suc (m ++ i)
len (filter P xs) leq len xs
len (butlast xs) = len xs minus Suc Z
len (delete x l) leq len l
n leq (m ++ n)
m leq n ==> m leq Suc n
x less y ==> (mem x (ins y l) <=> mem x l)
~(x = y) ==> (mem x (ins y l) <=> mem x l)
rev (drop i xs) = take (len xs minus i) (rev xs)
rev (filter P xs) = filter P (rev xs)
rev (take i xs) = drop (len xs minus i) (rev xs)
~(n = h) ==> count n (append x (Cons h Nil)) = count n x
count n t ++ count n (Cons h Nil) = count n (Cons h t)
sorted l ==> sorted (insort x l)
sorted (sort l)
(Suc m minus n) minus Suc k = (m minus n) minus k
take n (append xs ys) = append (take n xs) (take (n minus len xs) ys)
take n (drop m xs) = drop m (take (n ++ m) xs)
take n (zip xs ys) = zip (take n xs) (take n ys)
zip (append xs ys) zs =
append (zip xs (take (len xs) zs)) (zip ys (drop (len xs) zs))
zip xs (append ys zs) =
append (zip (take (len ys) xs) ys) (zip (drop (len ys) xs) zs)
len xs = len ys ==> zip (rev xs) (rev ys) = rev (zip xs ys)
height (mirror a) = height a
```

A.2.5 Hoare logic

There are 39 theorems rather than 38 (see Tabel 4.4), because `sorted_take` and `sorted_drop` are proven together in one theorem in Isabelle and we ported them in the same way.

```
take_Suc_conv_app_nth:!xs i. i < LENGTH xs
    ==> TAKE (SUC i) xs = APPEND (TAKE i xs) [EL i xs]
id_take_nth_drop:!xs i.
    i < LENGTH xs
    ==> xs = APPEND (TAKE i xs) (CONS (EL i xs) (DROP (SUC i) xs))
drop_take:!xs m n. DROP n (TAKE m xs) = TAKE (m - n) (DROP n xs)
take_take:!l a b. TAKE a (TAKE b l) = TAKE (MIN a b) l
ASG_conjunct0:ASG [] i n = []
ASG_conjunct1:ASG (CONS h t) 0 n = CONS n t
ASG_conjunct2:ASG (CONS h t) (SUC i) n = CONS h (ASG t i n)
length_list_update:!l i n. LENGTH (ASG l i n) = LENGTH l
nth_list_update:!xs i j x.
    i < LENGTH xs ==> EL j (ASG xs i x) = (if i = j then x else EL j xs)
nth_list_update_eq:!xs i x. i < LENGTH xs ==> EL i (ASG xs i x) = x
nth_list_update_neq:!xs i j x. ~(i = j) ==> EL j (ASG xs i x) = EL j xs
list_update_id:!l i. ASG l i (EL i l) = l
list_update_beyond:!l i n. LENGTH l <= i ==> ASG l i n = l
list_update_nonempty:!k x xs. ASG xs k x = [] <=> xs = []
list_update_same_conv:!xs i x. i < LENGTH xs
                     ==> (ASG xs i x = xs <=> EL i xs = x)
list_update_append1:!xs i. i < LENGTH xs ==>
                   ASG (APPEND xs ys) i x = APPEND (ASG xs i x) ys
list_update_append:!xs n ys.
    ASG (APPEND xs ys) n x =
    (if n < LENGTH xs
     then APPEND (ASG xs n x) ys
     else APPEND xs (ASG ys (n - LENGTH xs) x))
list_update_length:!xs ys. ASG (APPEND xs (CONS x ys)) (LENGTH xs) y
                                       = APPEND xs (CONS y ys)
map_update:!xs k. MAP f (ASG xs k y) = ASG (MAP f xs) k (f y)
```

```
rev_update:!xs k.
    k < LENGTH xs
    ==> REVERSE (ASG xs k y) = ASG (REVERSE xs) (LENGTH xs - k - 1) y
sorted_conjunct0:SORTED [] <=> T
sorted_conjunct1:SORTED (CONS h t) <=> ALL ((<=) h) t /\ SORTED t
sorted_single:SORTED [x]
sorted_many:x <= y ==> SORTED (CONS y zs) ==> SORTED (CONS x (CONS y zs))
sorted_many_eq:SORTED (CONS x (CONS y zs)) <=> x <= y /\ SORTED (CONS y zs)
sorted_tl:!xs. SORTED xs ==> SORTED (TAIL xs)
sorted_append:!xs ys.
    SORTED (APPEND xs ys) <=>
    SORTED xs /\ SORTED ys /\ (!x. MEM x xs ==> ALL ((<=) x) ys)
sorted_nth_mono:!xs i j. SORTED xs ==> i <= j ==>
                            j < LENGTH xs ==> EL i xs <= EL j xs
sorted_nth_monoI:!xs. (!i j. i <= j /\ j < LENGTH xs ==>
                            EL i xs <= EL j xs) ==> SORTED xs
sorted_butlast:!xs. ~(xs = []) /\ SORTED xs ==> SORTED (BUTLAST xs)
sorted_take:SORTED xs ==> SORTED (TAKE n xs)
sorted_drop:SORTED xs ==> SORTED (DROP n xs)
list_update_overwrite:!l i x y. ASG (ASG l i x) i y = ASG l i y
list_update_swap:!xs i i'. ~(i = i') ==>
                    ASG (ASG xs i x) i' x' = ASG (ASG xs i' x') i x
take_update_cancel:!xs n m. n <= m ==> TAKE n (ASG xs m y) = TAKE n xs
drop_update_cancel:!xs n m. n < m ==> DROP m (ASG xs n x) = DROP m xs
upd_conv_take_nth_drop:i < LENGTH xs ==>
             ASG xs i a = APPEND (TAKE i xs) (CONS a (DROP (SUC i) xs))
take_update_swap:n < m ==> TAKE m (ASG xs n x) = ASG (TAKE m xs) n x
drop_update_swap:m <= n ==> DROP m (ASG xs n x) = ASG (DROP m xs) (n - m) x
```

Bibliography

Alama, J., Heskes, T., Kühlwein, D., Tsivtsivadze, E., and Urban, J. (2014). Premise selection for mathematics by corpus analysis and kernel methods. *Journal of Automated Reasoning*, 52(2):191–213.

Alama, J., Kühlwein, D., and Urban, J. (2012). Automated and human proofs in general mathematics: An initial comparison. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 37–45. Springer.

Arthur, D. and Vassilvitskii, S. (2007). k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

Blanchette, J. C., Kaliszyk, C., Paulson, L. C., and Urban, J. (2016). Hammering towards QED. *Journal of Formalized Reasoning*, 9(1):101–148.

Böhme, S. and Nipkow, T. (2010). Sledgehammer: Judgement day. In Giesl, J. and Hähnle, R., editors, *Automated Reasoning*, pages 107–121, Berlin, Heidelberg. Springer Berlin Heidelberg.

Boulton, R. J. (1992). Boyer-Moore automation for the HOL system. In Claesen, L. J. M. and Gordon, M. J. C., editors, *Higher Order Logic Theorem Proving and its Applications: Proceedings of the IFIP TC10/WG10.2 Workshop*, volume A-20 of *IFIP Transactions*, pages 133–142, Leuven, Belgium. North-Holland/Elsevier.

Boyer, R. and Moore, J. (1979). *A Computational Logic*. ACM monograph series. Academic Press.

Bridge, J. P. (2010). *Machine learning and automated theorem proving*. PhD thesis, University of Cambridge.

Brock, B., Kaufmann, M., and Moore, J. S. (1996). ACL2 theorems about commercial microprocessors. In Srivas, M. and Camilleri, A., editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 275–293. Springer-Verlag.

Brown, C. E. (2012). Satallax: An automatic higher-order prover. In *International Joint Conference on Automated Reasoning*, pages 111–117. Springer.

Bundy, A. (1983). *The computer modelling of mathematical reasoning*, volume 10. Academic Press London.

Bundy, A. (1988). The use of explicit plans to guide inductive proofs. In Lusk, R. and Overbeek, R., editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag.

Bundy, A. (1996). Proof Planning. In Drabble, B., editor, *Proceedings of the 3rd International Conference on AI Planning Systems, (AIPS) 1996*, pages 261–267. also available as DAI Research Report 886.

Bundy, A. (2001). The automation of proof by mathematical induction. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning, Volume 1*, chapter 13. Elsevier.

Bundy, A., Basin, D., Hutter, D., and Ireland, A. (2005). *Rippling: Meta-level Guidance for Mathematical Reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

Carlson, A., Cumby, C., Rosen, J., and Roth, D. (1999). The SNoW learning architecture. Technical report, Technical report UIUCDCS.

Chamarthi, H. R., Dillinger, P., Manolios, P., and Vroon, D. (2011). The ACL2 sedan theorem proving system. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 291–295. Springer.

Claessen, K., Johansson, M., Rosén, D., and Smallbone, N. (2012). HipSpec: Automating Inductive Proofs of Program Properties. In *ATx/WInG@ IJCAR*, pages 16–25.

Claessen, K., Johansson, M., Rosén, D., and Smallbone, N. (2013). Automating inductive proofs using theory exploration. In *International Conference on Automated Deduction*, pages 392–406. Springer.

Claessen, K., Johansson, M., Rosén, D., and Smallbone, N. (2015). Tip: tons of

inductive problems. In *Conferences on Intelligent Computer Mathematics*, pages 333–337. Springer.

Cover, T. and Hart, P. (1967). Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27.

Cruanes, S. (2017). Superposition with structural induction. In *International Symposium on Frontiers of Combining Systems*, pages 172–188. Springer.

De Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer.

Dixon, L. and Fleuriot, J. D. (2003). IsaPlanner: A prototype proof planner in Isabelle. In *Proceedings of CADE'03*, volume 2741 of *LNCS*, pages 279–283.

Dixon, L. and Fleuriot, J. D. (2004). Higher order rippling in IsaPlanner. In *Theorem Proving in Higher Order Logics'04*, volume 3223 of *LNCS*, pages 83–98. Springer.

Gentzen, G. (1969). Investigations into logical deduction. *The collected papers of Gerhard Gentzen*, pages 68–131.

Gordon, M. J., Milner, A. J., and Wadsworth, C. P. (1979). *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag.

Hales, T. C. (2006). Introduction to the Flyspeck project. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Harrison, J. (1996a). HOL light: a tutorial introduction. In Srivas, M. and Camilleri, A., editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 265–269.

Harrison, J. (1996b). Optimizing proof search in model elimination. In *International Conference on Automated Deduction*, pages 313–327. Springer.

Harrison, J. (2009). *Handbook of practical logic and automated reasoning*. Cambridge University Press.

Harrison, J. (2016). The HOL Light system reference. *See URL: https://www.cl.cam.ac.uk/ jrh13/hol-light/reference.pdf*.

Heras, J. and Komendantskaya, E. (2014). ACL2(ml): Machine-Learning for ACL2. *arXiv preprint arXiv:1404.3034*.

Heras, J., Komendantskaya, E., Johansson, M., and Maclean, E. (2013). Proof-pattern recognition and lemma discovery in ACL2. In McMillan, K. L., Middeldorp, A., and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, volume 8312 of *Lecture Notes in Computer Science*, pages 389–406. Springer.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580.

Hoder, K. and Voronkov, A. (2011). Sine Qua Non for large theory reasoning. In *Automated Deduction–CADE-23*, pages 299–314. Springer.

Hurd, J. (2003). First-order proof tactics in higher-order logic theorem provers. *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports*, pages 56–68.

Ireland, A. (1992). The Use of Planning Critics in Mechanizing Inductive Proofs. In Voronkov, A., editor, *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag. Also available from Edinburgh as DAI Research Paper 592.

Ireland, A. and Bundy, A. (1996). Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111.

Johansson, M., Dixon, L., and Bundy, A. (2010). Case-analysis for Rippling and inductive proof. In Kaufmann, M. and Paulson, L., editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 291–306. Springer.

Kaliszyk, C. and Urban, J. (2013). Stronger automation for Flyspeck by feature weighting and strategy evolution. In *Third International Workshop on Proof Exchange for Theorem Proving (PxTP 2013)*, page 87.

Kaliszyk, C. and Urban, J. (2014). Learning-assisted automated reasoning with Flyspeck. *Journal of Automated Reasoning*, pages 1–41.

Kaliszyk, C. and Urban, J. (2015). Hol (y) hammer: Online ATP service for HOL Light. *Mathematics in Computer Science*, 9(1):5–22.

Kaufmann, M., Manolios, P., and Moore, J. (2000). *Computer-Aided Reasoning: An Approach*. Advances in Formal Methods. Springer US.

Kaufmann, M. and Moore, J. (2004). How to prove theorems formally. *See URL: http://www. cs. utexas. edu/users/moore/-publications/how-to-prove-thms/main. ps*.

Kovács, L. and Voronkov, A. (2013). First-order theorem proving and Vampire. In *Computer Aided Verification*, pages 1–35. Springer.

Kreisel, G. (1965). Lectures on modern mathematics, volume 3, chapter mathematical logic.

Kühlwein, D., Blanchette, J. C., Kaliszyk, C., and Urban, J. (2013). Mash: Machine learning for Sledgehammer. In *Interactive Theorem Proving*, pages 35–50. Springer.

Kühlwein, D., Urban, J., Tsivtsivadze, E., Geuvers, H., and Heskes, T. (2011). Multi-output ranking for automated reasoning. In *KDIR: Proceedings of the International Conference on Knowledge Discovery and Information Retrieval*, pages 42–51. sn: SciTePress.

Kühlwein, D., van Laarhoven, T., Tsivtsivadze, E., Urban, J., and Heskes, T. (2012). Overview and evaluation of premise selection techniques for large theory mathematics. In *Automated Reasoning*, pages 378–392. Springer.

Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., and Vouillon, J. (2014). The OCaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique*.

Lindhé, V. and Logren, N. (2016). Proof output and machine learning for inductive theorem provers. Master's thesis, Chalmers University of technology.

Loveland, D. W. (1968). Mechanical theorem-proving by model elimination. *Journal of the ACM (JACM)*, 15(2):236–251.

Meng, J. and Paulson, L. C. (2009). Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, 7(1):41–57.

Moore, J. S. (2003). Proving theorems about Java and the JVM with ACL2. *NATO Science Series Sub Series III Computer and Systems Sciences*, 191:227–290.

Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. MIT press.

Nipkow, T. (2013). Programming and proving in Isabelle/HOL. *https://isabelle.in.tum.de/doc/prog-prove.pdf*.

Nipkow, T. and Klein, G. (2014). Concrete semantics. *A Proof Assistant Approach*.

Nipkow, T., Paulson, L. C., and Wenzel, M. (2002). *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer.

Papapanagiotou, P. (2007). On the automation of inductive proofs in HOL Light. Master's thesis, University of Edinburgh.

Papapanagiotou, P. and Fleuriot, J. D. (2018). The Boyer-Moore waterfall model revisited. *ArXiv*, abs/1808.03810.

Paulson, L. C. and Blanchette, J. C. (2010). Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. *IWIL-2010*, 1.

Reynolds, A. and Kuncak, V. (2015). Induction for SMT solvers. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 80–98. Springer.

Rifkin, R., Yeo, G., Poggio, T., et al. (2003). Regularized least-squares classification. *Nato Science Series Sub Series III Computer and Systems Sciences*, 190:131–154.

Robinson, G. and Wos, L. (1969). Paramodulation and theorem-proving in first-order theories with equality. In Michie, D., editor, *Machine Intelligence 4*, pages 103–33. Edinburgh University Press.

Roederer, A., Puzis, Y., and Sutcliffe, G. (2009). Divvy: An ATP meta-system based on axiom relevance ordering. In *Automated Deduction–CADE-22*, pages 157–162. Springer.

Schirmer, N. (2008). A sequential imperative programming language syntax, semantics, Hoare logics and verification environment. *Archive of Formal Proofs*. `http://isa-afp.org/entries/Simpl.html`, Formal proof development.

Schulz, S. (2013). System Description: E 1.8. In McMillan, K., Middeldorp, A., and Voronkov, A., editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer.

Scott, P. (2015). *Ordered geometry in Hilbert's Grundlagen der Geometrie*. PhD thesis, The University of Edinburgh.

Slind, K. and Norrish, M. (2008). A brief overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics*, pages 28–32. Springer.

Smaill, A. and Green, I. (1996). Higher-order annotated terms for proof search. In von Wright, J., Grundy, J., and Harrison, J., editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1275 of *Lecture Notes in Computer Science*, pages 399–414, Turku, Finland. Springer-Verlag. Also available as DAI Research Paper 799.

Sonnex, W., Drossopoulou, S., and Eisenbach, S. (2012). Zeno: An automated prover for properties of recursive data structures. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 407–421. Springer.

Sutcliffe, G. (2009). The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362.

Sutcliffe, G. (2016). The CADE ATP System Competition - CASC. *AI Magazine*, 37(2):99–101.

Sutcliffe, G. and Puzis, Y. (2007). Srass-a semantic relevance axiom selection system. In *Automated Deduction–CADE-21*, pages 295–310. Springer.

Urban, J. (2013). Blistr: The blind strategymaker. *arXiv preprint arXiv:1301.2683*.

Walker, S. H. and Duncan, D. B. (1967). Estimation of the probability of an event as a function of several independent variables. *Biometrika*, 54(1/2):167–179.

Zhang, H. (2004). The optimality of Naive Bayes. In Barr, V. and Markov, Z., editors, *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference, Miami Beach, Florida, USA*, pages 562–567. AAAI Press.