Learning Proof Search in Proof Assistants

dissertation

by

Michael Färber

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of PhD

advisor: Ass.-Prof. Dr. Cezary Kaliszyk

Innsbruck, 31 July 2018

universität
innsbruck

dissertation

Learning Proof Search in Proof Assistants

Michael Färber (1119628)

31 July 2018

advisor: Ass.-Prof. Dr. Cezary Kaliszyk

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

________________________ ________________________

Datum Unterschrift

Abstract

Proof assistants are programs that verify the correctness of formal proofs. They can increase the confidence in results from domains such as mathematics, informatics, physics, and philosophy. However, it requires extensive labour and expertise to write proofs accepted by proof assistants. In this thesis, we improve proof automation in proof assistants. *Automated theorem provers* are programs that search for proofs automatically. Our goal is to find proofs in proof assistants using automated theorem provers. However, this is not directly possible when the logic of an automated theorem prover and that of a proof assistant differ. In this case, the integration of the automated theorem prover into the proof assistant requires the translation of statements to the logic of the automated theorem prover and the translation of proofs to the logic of the proof assistant. To restrict the search space of the automated theorem prover, only a selection of facts relevant to the current conjecture is translated. The success rate of the automatic proof search in proof assistants depends on the various translations, the selection of relevant facts as well as on the automated theorem prover itself. We improve the integration of automated theorem provers into proof assistants. Among others, we *learn* from previous proofs to select relevant facts as well as to guide automated theorem provers to make good decisions. Furthermore, we create automated proof translations for several automated theorem provers for which such a translation was not previously available. Our work increases the success ratio of proof search in proof assistants.

Acknowledgments

I would like to thank all members of the Computational Logic group, in particular my colleagues Burak Ekici, Thibault Gauthier, Sebastiaan J.C. Joosten, Alexander Maringele, Yutaka Nagashima, Julian Parsert, Thomas Powell, Thomas Prokosch, Jonas Schöpf, and Qingxiang (Shawn) Wang. You shared the office with me or dropped by more or less frequently for having a chat or going to lunch. You helped me in various ways, such as by proofreading my articles, by discussing research, or simply by creating an atmosphere in which it was a pleasure to work in. I would like to especially thank Michael Schaper for having been my Haskell mentor during the first year of my PhD, as well as Aart Middeldorp and René Thiemann for proofreading my thesis and improving it by their valuable suggestions.

Free software is crucial for my work. For the tools I use nearly every day, I would like to thank

- Richard Stallman for initiating the GNU project and for all its tools, especially GNU `make`, which was indispensable for creating reproducible experiments;
- Linus Torvalds for the Linux kernel and the version control system Git;
- John MacFarlane for the document converter Pandoc that I extensively used to write my articles and this thesis;
- all the teams around these persons.

During my PhD, I was fortunate enough to correspond with many experts in automated and interactive theorem proving that took their time to help me. I would like to thank

- John R. Harrison for his proof assistant HOL Light and for integrating my work into it;
- Joe Leslie-Hurd for advice on the reconstruction of proofs in the calculus of his theorem prover Metis;
- Jens Otten for providing *platinum support* for his theorem provers, for taking so much time to explain to me the intricacies of nonclausal connection proof search, and for creating programs that demonstrate that logic can be a piece of art;
- Chad E. Brown for his great help with his theorem prover, for challenging me with logical problems, for the political discussions and for being a model by pursuing his ideas with unparalleled perseverance.

I would like to thank the reviewers of CADE, CPP, FroCoS, IJCAR, and LPAR for their valuable comments.

I would like to thank all organisations that financed my studies. This work has been supported by the Austrian Science Fund (FWF) grant P26201 and a doctoral scholarship of the University of Innsbruck. My research visits have been supported by the European Research Council (ERC) grant *AI4REASON* and an Erasmus scholarship.

I would like to thank Josef Urban for inviting me two times to research in Prague

for a total of five months. During these visits, Josef was a very attentionate host, who organised hikes, the Prague Inter-reasoning Workshops (PIWo), movie nights, etc. He even lent me a bike so I could learn to appreciate the cobbled streets of Prague. We discussed nearly every day, and he contributed several ideas that greatly influenced my work – without him, I would not be where I am today.

I would like to thank my main supervisor Cezary Kaliszyk for having supported me since we met at the first session of the Master Seminar of Computational Logic in 2011. Little did I know at that time that I would be still working with the same guy seven years later. Cezary always motivated me to be productive, which in hindsight turned out to be very useful given my innate perfectionism. His personal mantra might well be “Get stuff done!”, and I tried to apply it. He gave me time to explore my interests and accepted my choices, even when he did not agree with them. (My choice of lazy functional programming languages springs to my mind.) I consider him to be not only an outstanding scientist, but also an amazingly capable programmer, and I learnt a lot from him in both regards. He always had an ear for my problems, which we discussed while walking around the campus, thus following in the footsteps of the Peripatetic school.

Finally, I would like to thank my family, my friends, and especially Mathilde for supporting me during my studies.

I am deeply indebted to you all.

Contents

1 Introduction **1**
1.1 Introduction . . . 1
1.2 Interactive Theorem Provers . . . 2
1.3 Automated Theorem Provers . . . 4
1.4 Hammers . . . 7
1.5 Contributions . . . 8
1.6 Outline . . . 9

2 Premise Selection **11**
2.1 Introduction . . . 11
2.2 k-nearest neighbours . . . 13
2.3 Naive Bayes . . . 14
2.4 Decision Trees . . . 16
2.4.1 Feature Selection . . . 17
2.4.2 Incremental Learning . . . 19
2.4.3 Querying . . . 20
2.5 Random Forests . . . 22
2.5.1 Sample Selection . . . 22
2.5.2 Incremental Learning . . . 22
2.5.3 Querying . . . 23
2.6 Evaluation . . . 23
2.7 Related Work . . . 27
2.8 Conclusion . . . 27

3 Connection Proof Search **29**
3.1 Introduction . . . 29
3.2 Connection Calculi . . . 30
3.3 Problem Preprocessing . . . 33
3.4 Consistent Skolemisation . . . 36
3.5 Connection Search . . . 39
3.6 Proof Search . . . 40
3.6.1 Prolog . . . 41
3.6.2 Lazy Lists and Streams . . . 41
3.6.3 Continuations . . . 42
3.6.4 Stacks . . . 43
3.7 Clause Processing Order . . . 43
3.8 Extension Clause Anomaly . . . 44

3.9 Evaluation . . . 45
3.10 Reproducible Experiments . . . 47
3.11 Conclusion . . . 49

4 Internal Guidance **51**
4.1 Introduction . . . 51
4.2 Naive Bayes with Monoid Occurrences . . . 52
4.3 Features . . . 53
4.4 Training Data Recording . . . 54
4.5 Clause Processing . . . 54
4.6 Clause Ranking . . . 55
4.7 Parameter Tuning . . . 55
4.7.1 Offline Tuning . . . 55
4.7.2 Particle Swarm Optimisation . . . 56
4.8 Implementation . . . 56
4.8.1 Machine Learning . . . 57
4.8.2 Strategies . . . 57
4.9 Evaluation . . . 57
4.10 Related Work . . . 60
4.11 Conclusion . . . 61

5 Monte Carlo Proof Search **63**
5.1 Introduction . . . 63
5.2 Monte Carlo Tree Search . . . 64
5.3 Child Selection Policy . . . 65
5.4 Child Probability . . . 66
5.5 Reward . . . 66
5.6 Expansion Policy . . . 68
5.7 Implementation . . . 68
5.8 Evaluation . . . 69
5.9 Conclusion . . . 70

6 Proof Reconstruction **73**
6.1 Introduction . . . 73
6.2 Translation to First-Order Logic . . . 73
6.3 Metis . . . 75
6.4 Connection Proofs . . . 77
6.4.1 Connection Calculi for Proof Translation . . . 78
6.4.2 Connection Proof Translation . . . 79
6.4.3 Clausal Proof Translation . . . 80
6.4.4 Nonclausal Proof Translation . . . 81
6.5 Evaluation . . . 84
6.6 Related Work . . . 84
6.7 Conclusion . . . 85

7 Conclusion **87**
7.1 Future Work 88
7.1.1 Machine Learning 88
7.1.2 Automated Theorem Proving 89

Bibliography **91**

Chapter 1

Introduction

1.1 Introduction

An *interactive theorem prover* (ITP), also called *proof assistant*, is a computer program that allows users to unambiguously formulate statements and their proofs, verifying the correctness of the proofs. To motivate the usage of ITPs, consider the Kepler conjecture from 1611.

Conjecture 1.1 (Kepler Conjecture)**.** No packing of congruent balls in three-dimensional Euclidean space has density greater than that of the face-centered cubic packing [Kep11].

The first proof of the Kepler conjecture was published only in 2006 by Thomas C. Hales. However, its reviewers were not able to certify its correctness with absolute confidence [HAB+17]. This led to the creation of the Flyspeck project with the goal to mechanically verify Hales's proof in an ITP. The project was carried out as an international collaboration with the participation of more than 20 people. It was finished in 2014 and certified that Hales's proof of the Kepler conjecture was indeed correct. The Flyspeck project shows that ITPs can increase confidence in mathematical results. However, the Flyspeck project also demonstrates that the effort required to create mechanically verifiable proofs can be considerable.

A way to help ITP users is to provide them with methods that find proofs automatically. While many conjectures in ITPs can be efficiently proven by decision procedures, there remains a large amount of problems intractable by decision procedures. For this purpose, *automated theorem provers* (ATPs) were integrated into ITPs: Where interactive theorem provers verify the correctness of given proofs, automated theorem provers search for new proofs. A milestone was the proof of the previously unproven Robbins conjecture, found by William McCune in 1996 using an ATP [McC97].

Conjecture 1.2 (Robbins Conjecture)**.** Assuming the associativity and commutativity of addition (+), the Huntington equation $n(n(x) + y) + n(n(x) + n(y)) = x$ is logically equivalent to the Robbins equation $n(n(x + y) + n(x + n(y))) = x$ [McC97].

Initially, to use ATPs in ITPs, ITP users had to provide ATPs with a selection of facts thought relevant to solve their current problem. This selection was necessary because passing all facts available in an ITP to an ATP risks exploding the search space of the ATP, thus reducing the chance of the ATP finding a proof. However, selecting relevant facts can be difficult for users. To relieve users, so-called *hammer* systems have been

developed. Hammer systems combine ATPs and decision procedures with techniques to find relevant facts to search for proofs in ITPs without human guidance. With a hammer, more than 40% of the proofs in the formalisation of the proof of Kepler's conjecture can be found automatically [KU14]. This suggests that such tools can significantly reduce the work required in future formalisations.

The goal of this thesis is to create and evolve techniques and tools for using automated approaches in interactive theorem provers in order to allow the automatic construction of proofs that can be mechanically verified. To evaluate our progress, we use a standard ATP metric, namely the number of (logical) problems that a method solves in a fixed amount of time. There exist several canonical sets of test problems on which ATPs are usually evaluated. As we wish to improve automated reasoning in ITPs, we focus on problems stemming from ITPs.

The introduction chapter is structured as follows: In Sections 1.2, 1.3 and 1.4, we introduce ITPs, ATPs, and their integration, respectively. In Section 1.5, we give an overview of the contributions in this thesis and outline its structure in Section 1.6.

1.2 Interactive Theorem Provers

An interactive theorem prover (ITP) is a program that allows users to state conjectures and their proofs, which are then mechanically checked. If a user trusts the soundness of an ITP, the user can be certain that conjectures proven inside the ITP indeed hold. The ITP provides the user with feedback, for example showing which parts of a conjecture are left to prove, rejecting invalid inference steps and so on. We will now give a historical overview of ITPs, showing some of nowadays most frequently used ITPs and their predecessors. This overview is far from comprehensive; for example, we omit systems such as *Metamath* [Meg07], *Agda* [BDN09], and the Boyer-Moore family of provers [BM98].

In 1968, Nicolaas Govert de Bruijn introduced *Automath* as a formal language to express mathematical proofs that could be mechanically checked by a computer [Wie02]. Automath is based on the typed lambda calculus with dependent types. Dependent types allow the encoding of mathematical propositions as types. This relationship is known as Curry-Howard isomorphism. It was generalised to predicate logic by Per Martin-Löf [ML84]. A proposition is proven by giving a term of the proposition's type. Proof checking then corresponds to verifying whether the type of a given proof term corresponds to its stated type (the proposition). The distinction between proofs and programs blurs. The largest project realised in Automath is the formalisation of Edmund Landau's "Grundlagen der Analysis" [Lan30, vBJ77].

In 1972, Robin Milner proposed the *LCF* system [Gor00] as a proof checker for the "Logic for Computable Functions" by Dana Scott. The unique approach taken by the LCF system is that proofs are written as programs in the strongly-typed meta-language (ML), which ensures that values of type proof can only be constructed via a precisely defined set of primitive inference rules. As long as the primitive rules are sound, ML's type system ensures the soundness of the whole system. Furthermore, as ML is a general-purpose programming language, it allows more complex inference rules to be defined in terms of

the primitive inference rules. Tactics allow proving goals by breaking them into smaller subgoals, which enables so-called "goal-directed proof search".

Around 1973, Andrzej Trybulec started to work on *Mizar*, a formal language for mathematics [MR05, NK09]. The language should be close to conventional mathematical language, but with clear semantics in order to allow automatic processing, including proof verification. Mizar proofs are written in declarative style, where intermediate statements are explicitly written out and linked together to show the main statement. Mizar is based on first-order logic; however, to allow e.g. for induction, free second-order variables can be instantiated. Mizar is used to develop a large library of formalised mathematical knowledge, namely the Mizar Mathematical Library (MML), consisting of more than 40,000 theorems. While Mizar allows in principle multiple foundations to formalise mathematics, the predominant part of the MML is based on Tarski-Grothendieck set theory. Despite being far away from formalising all of contemporary mathematics, it can be seen as a step towards the goal of the QED project, namely the formalisation and verification of all mathematical knowledge [QED94].

In 1982, Gérard Huet started the Projet Formel, having as goal the creation of a proof system based on LCF [BC04, Ber08]. The result was a functional programming language based on the *Calculus of Constructions*, which is a variant of typed lambda calculus with dependent types and polymorphism. Similarly to Automath, a proposition is proven by giving a term of the proposition's type. Proofs can be written either by composition of functions, or via a set of tactics, allowing goal-directed proof search in the LCF tradition. Later, the calculus was extended to the *Calculus of Inductive Constructions*, to allow the formalisation of algorithms operating on inductive data types. Starting from 1989, the proof system was released as *Coq*. The development of the system also yielded the language *Caml* in 1986, which evolved into *Objective Caml* (OCaml) in 1996. By default, Coq is constructive, which allows code extraction, at the expense of not assuming the law of excluded middle and extensionality (two functions are equal if they return the same output for the same input). Two of the largest projects done in Coq are the verified C compiler CompCert [Ler06] and the verification of the Feit-Thompson theorem [GAA+13].

In the mid-1980s, Mike Gordon replaced the logic of the LCF system by higher-order logic, resulting in the *HOL* system [Gor00]. The first stable version of HOL was released around 1988 as HOL88, whose logic has remained unchanged for all subsequent versions. Later, Konrad Slind ported HOL to Standard ML [SN08], resulting in HOL90. After HOL98, the latest version was released as HOL4. One of the largest formalisations done in HOL4 is CakeML, a verified implementation of ML [KMNO14].

In 1986, Larry Paulson initiated the *Isabelle* system, after having worked on Cambridge LCF in the mid-1980s [Pau88]. The speciality of Isabelle is its separation in meta-logic and object-logics: Isabelle's meta-logic *Pure* is a fragment of higher-order logic that allows the specification of syntax and inference rules of various object-logics, such as HOL [NPW02] and ZF (Zermelo-Fraenkel set theory). Proofs can be written via the application of tactics à la LCF or declaratively with the *Isar* (Intelligible semi-automated reasoning) language inspired by Mizar [WPN08]. Proofs in the object-logics are verified by a common logical core, following the LCF tradition. By far the most

frequently used Isabelle object-logic is *Isabelle/HOL*: One of its most powerful proof search methods is *Sledgehammer*, which we introduce in Section 1.4. In addition to proof search, Isabelle/HOL also supports counterexample search: *Quickcheck* instantiates the free variables of executable formulas with random instances to find contradictions, and *Refute* translates formulas to propositional logic and searches for finite countermodels with a SAT solver. Code can be generated from constructive Isabelle/HOL functions to functional languages, including Standard ML, OCaml, and Haskell. One of the largest Isabelle/HOL projects is the verification of an operating system kernel [KAE+10]. The Archive of Formal Proofs (AFP) is a collection of proof libraries that can be mechanically verified and reused in Isabelle.

In 1995, John Harrison created *HOL Light* based on HOL90 in collaboration with Konrad Slind, with the goal of having a kernel that is small and easy to adapt [Har09]. Initially being implemented in Caml, HOL Light later switched to OCaml. It was used at Intel for verification of floating-point arithmetic, as well as for the Flyspeck project [HAB+17].

All HOL systems mentioned here (HOL4, Isabelle/HOL, HOL Light) are based on Church's simple type theory with polymorphic type variables, without subtyping or dependent types [PS07]. Yet, there are differences between the logical foundations of the systems; for example, unlike HOL4 and HOL Light, Isabelle/HOL uses axiomatic type classes. The HOL systems provide similar reasoning tools, such as tableaux provers [Har96, Pau99], the resolution prover Metis [Hur03, PS07], decision procedures (e.g. for linear arithmetic), and simplifiers for higher-order rewriting.

1.3 Automated Theorem Provers

An automated theorem prover (ATP) is a program that attempts to prove logical conjectures automatically. ATPs exist for logics of different order, such as first-order logic and higher-order logic, as well as for classical and non-classical logics, such as intuitionistic and modal logic. The yearly CADE Automated Systems Competition (CASC) evaluates the performance of ATPs on different sets of logical problems [Sut16b]. We give an overview of the ATPs that are considered in this thesis. All ATPs presented below are performing proof by contradiction; that is, to show that a conjecture follows from a set of axioms, the ATPs show that the negation of the conjecture and the axioms imply $\bot$.

MESON (Model Elimination Subgoal OrieNted) is a proof search method introduced by Donald W. Loveland and Mark E. Stickel [LS73, Lov16] based on Loveland's model elimination calculus [Lov68]. Stickel later introduced a "Prolog Technology Theorem Prover" (*PTTP*), i.e. a complete proof search method in Prolog using sound unification, the reduction rule of the model elimination calculus, and inference-bounded iterative deepening [Sti88]. PTTP was the basis of Harrison's implementation of the MESON method [Har96]. Based on the inference-bounded search strategy of Stickel, Harrison introduced a divide-and-conquer approach that tries different orders of goals with reduced inference bounds. Furthermore, Harrison implemented depth-bounded and best-first

strategies. The resulting prover uses "positive refinement" [Pla90] to restrict the application of the reduction rule. Furthermore, input clauses are reordered such that smaller ones are tried first, and literals are reordered such that those with few free variables are tried first. The "caching" technique [AS92] is used to cancel proof attempts when it is clear from previous proof attempts that they will fail. Harrison integrated MESON into HOL Light as a proof search tactic: The tactic translates the current goal to first-order logic, where MESON attempts to find a proof that is then reconstructed in HOL. We use Harrison's MESON tactic to generate logical problems from HOL Light as well as to benchmark our own proof search tactics, see Chapter 6.

leanCoP [OB03, Ott08] and *nanoCoP* [Ott16] are ATPs for first-order logic developed by Jens Otten. They are based on the clausal and nonclausal connection tableaux calculus [Ott11], respectively. These enable very compact implementations of goal-directed proof search. We describe the calculi and the search procedure in Chapter 3, where we also introduce several functional implementations. Furthermore, we show the expansion of a proof search tree with Monte Carlo Tree Search in Chapter 5. Finally, we give a translation of connection tableaux proofs to Gentzen's sequent calculus LK [Gen35] in Section 6.4, enabling proof certification and automatic proof search in ITPs.

Vampire [KV13] is an ATP for first-order logic with equality developed by a team around Andrei Voronkov, Kryštof Hoder, and Laura Kovács. It is based on the superposition calculus [BG94]. The AVATAR framework [Vor14] allows the integration of SAT (such as MiniSat [ES03]) and SMT solvers (such as Z3 [dMB08]) to efficiently handle clauses that can be split into subsets with disjoint variables. To show satisfiability of problems, Vampire uses the instantiation generation method [Kor13] as well as finite model building using SAT/SMT solvers. During preprocessing, relevance filtering can be performed with SInE [HV11]. Superposition provers such as Vampire use saturation algorithms to structure proof search. All saturation algorithms in Vampire belong to the family of given-clause algorithms [McC03]. We introduce a simple given-clause algorithm: Given an initial set of clauses to refute, the set of *unprocessed* clauses is the initial set of clauses, and the set of *processed* clauses is the empty set. At every iteration of the algorithm, a *given clause* is selected from the unprocessed clauses and moved to the processed clauses. Clauses that can be inferred from the given clause and the processed clauses are added to the unprocessed clauses. The algorithm terminates as soon as either the set of unprocessed clauses is empty (the problem is satisfiable) or the empty clause was generated (the problem is unsatisfiable). An important distinction between given-clause algorithms is the question which clauses can contribute to simplifying inferences. In the DISCOUNT [DKS97] loop, simplification is done only with processed clauses, whereas in the Otter loop [McC03], simplification is done both with processed *and* unprocessed clauses. In addition to the DISCOUNT and Otter loops, Vampire has a "Limited Resource Strategy" based on the Otter loop that discards clauses that seem unlikely to be processed within the given time limit.

E [Sch13] is an ATP for clausal first-order logic with equality developed by Stephan Schulz. It is based on a variant of the superposition calculus [BG94] with literal selection. Relevance filtering can be performed with SInE [HV11]. Proof search is performed with the DISCOUNT loop; see the paragraph on Vampire. E uses shared terms, i.e. every

distinct term is stored exactly once in a term database. Unconditional rewriting steps are cached, that is, rewrite steps that are performed on a term are stored in the term database, such that equivalent terms can be rewritten in future simplification steps. Furthermore, E uses efficient clause indexing to find inference partners. E provides a flexible system for clause evaluation: Users can specify arbitrary priority queues and a weighted round-robin scheme to influence which unprocessed clauses are picked. Some evaluation functions prefer clauses that seem related to the conjecture, thus making proof search goal-directed. Clause evaluation functions can be learned from previous proofs [Sch00]. Strategies or strategy schedules can be automatically chosen for the current problem by taking the strategy that performed best on similar problems.

Metis [Hur03] is an ATP with an LCF-style kernel for clausal first-order logic developed by Joe Leslie-Hurd. The LCF principle enables the combination and cooperation of different proof procedures, e.g. by sharing unit clauses.[1] Furthermore, the LCF approach is useful for proof recording, which simplifies the reconstruction of proofs found by Metis in ITPs. Originally, Metis implemented three proof procedures: (i) the subgoal-oriented variant MESON [LS73] of Loveland's model elimination procedure [Lov68], (ii) Robinson's resolution procedure [Rob65] with the given-clause algorithm, using term nets for fast unification and subsumption checking, and (iii) the Delta procedure based on Schumann's Delta preprocessor [Sch94]. Metis 2.0 [Sut09a] dropped the model elimination and Delta procedures from the prover, leaving ordered resolution and ordered paramodulation. Originally integrated into HOL4, Metis has been integrated since into Isabelle/HOL [PS07], where it serves to automate proofs as well as reconstruct proofs found by automated provers. As shown in [KUV15], paramodulation-based provers such as Metis perform better than tableaux-based provers such as MESON and leanCoP on many problems involving equality. We describe our integration of Metis into HOL Light in Section 6.3.

Satallax [Bro12] is an ATP for simply-typed higher-order logic developed by Chad E. Brown. It is based on a tableaux calculus with extensionality and choice [BB11]. Similarly to the given-clause algorithm, the search procedure of Satallax [Bro13] keeps processed/unprocessed formulas and processed/unprocessed terms (for instantiation). In every iteration of the search procedure, one of three actions is performed: (i) an unprocessed formula is moved to the processed formulas, (ii) an unprocessed term is moved to the processed terms, or (iii) an unprocessed term is generated. These actions are picked from a priority queue. When an unprocessed formula/term u is moved to the processed formulas/terms, formulas inferable from u and processed formulas/terms are generated. These formulas are moved to the unprocessed formulas and commands for processing them are put on the priority queue. The problem is unsatisfiable iff at some point the set of processed formulas becomes unsatisfiable. Satallax keeps a satisfiability-preserving translation of formulas to propositional and predicate logic, to check for unsatisfiability with MiniSat and E, respectively [Sut16a].

[1] Using a small logical kernel in the spirit of LCF to combine proof procedures has also been implemented since e.g. in the Psyche system [Gra13].

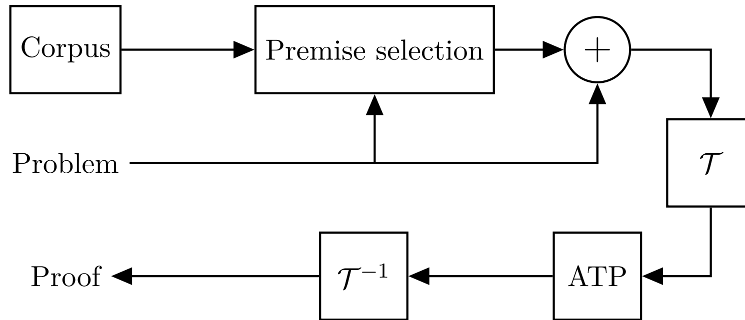


Figure 1.1: A typical hammer workflow.

1.4 Hammers

The widespread usage of ITPs in mathematics and related domains, such as software verification, has so far been hindered by the high level of knowledge and experience required to operate ITPs. In particular, users struggle to find facts related to their current problems, and once they have found such facts, how to combine them to solve their problems.

A *hammer* system attempts to ease reasoning with ITPs by automatically proving conjectures, thus reducing the entry barrier for users to employ ITPs. Many frameworks use ATPs, which in contrast to ITPs attempt to solve problems without any user interaction, but whose logic might be different from an ITP's logic.

The function of such a system is illustrated in Figure 1.1: The input consists of a problem together with a corpus of axioms and theorems. First, *premise selection* chooses a subset of the corpus thought to be relevant to the solution of the problem.[2] Then, the selected subset and the problem are translated with a method $\mathcal{T}$ to the logic of an ATP. If the ATP finds a proof, the proof is reconstructed with a method $\mathcal{T}^{-1}$ in the ITP logic, either by extracting a precise small set of facts present in the ATP proof, or by translating the ATP proof to recreate a skeleton of an ITP proof. In some cases, the reconstruction of ATP proofs fails, because the ATP does not provide sufficient information to translate a proof. Still, it is often possible to reconstruct the ATP proof by reproving the conjecture with a different ATP, using the facts the first ATP used in its proof.

Sledgehammer [Pau10] is a hammer system for Isabelle/HOL. It was first released in 2007 with the vision to provide "one-click invocation", meaning that users could invoke it for any conjecture without providing additional information to guide the system, such as conjecture-relevant facts. To find such facts, Sledgehammer uses a combination of two machine learning techniques, namely the Meng-Paulson relevance filter [MP09] and a variant of Naive Bayes [BGK$^+$16]. An Isabelle/HOL-specific challenge is the translation of problems to first-order logic: While Leslie-Hurd [Hur03] observed that omitting type information in the translation to first-order logic increases the success rate, the widespread usage of axiomatic type classes in Isabelle/HOL prevents this. Therefore,

[2] As we do not distinguish between axioms and lemmas in premise selection, we denote their union as *theorems*. Furthermore, we denote the theorems used in a proof attempt as *premises*.

Sledgehammer translates only enough type information to allow for type-class reasoning, omitting all other type information. Higher-order constructs are treated locally, such that a single higher-order lemma does not necessarily change the translation of all other lemmas. Lambda abstractions are eliminated via combinatory logic. Once the problem is translated to first-order logic, Sledgehammer searches for proofs with untrusted tools such as E, Vampire, and SPASS. A proof found by one of these systems can either be reconstructed with a single call to Metis using the facts present in the proof, or translated line-by-line to an Isar proof script. As part of proof reconstruction, proofs are minimised, i.e. the ATP is called with subsets of the facts it used in its proof, until a minimal set of facts necessary to prove the problem is found. An alternative to reducing problems to first-order logic is to integrate higher-order logic ATPs, which was done with Satallax and LEO-II [SBP13]. Furthermore, SMT solvers were integrated [BBP11].

Systems similar to Sledgehammer are *HOL(y)Hammer* [KU14, KU15b] for HOL4 and HOL Light, and *MizAR* [KU15d] for Mizar.

In this thesis, we improve three parts of the hammer infrastructure, namely premise selection, automated theorem provers, and proof reconstruction. Machine learning played a crucial part in the first two parts.

1.5 Contributions

We make the following contributions in this thesis:

- We investigate online and offline random forests for premise selection: We improve an offline random forest algorithm with incremental learning and add multi-path querying with depth weighting to integrate secondary classifiers.
- We implement efficient proof search based on clausal and nonclausal connection tableaux calculi in functional programming languages. Furthermore, we introduce a consistent Skolemisation method and an alternative clause processing order for nonclausal proof search.
- We introduce a method to guide given-clause ATPs based on positive and negative examples. We apply our technique to an ATP operating on higher-order logic, thus realising the first internal guidance for higher-order ATPs.
- We use Monte Carlo Tree Search to guide ATPs. To this end, we propose and evaluate several proof state evaluation heuristics, including two that learn from previous proofs.
- We translate problems in higher-order logic to first-order logic. We also certify first-order logic proofs by translation to an ITP: We consider Metis as well as clausal and nonclausal connection tableaux proofs. For connection tableaux calculi, we give a unified formulation adapted for proof translation.

This thesis includes content from several published articles [FK15a, FK15b, FB16, FKU17] and an article submitted to JAR [FKU18]. For each of the aforementioned articles, more than 90% of the content was written by the author of this thesis. Furthermore, the author of this thesis has implemented and evaluated all programs and algorithms described in these articles.

This thesis improves on the articles by providing uniform notation and a structure that groups related topics from different articles, thus reducing redundancy. To improve the presentation of the existing content, several examples were added. Furthermore, the thesis describes some topics not present in the articles, such as k-nearest neighbours for premise selection (Section 2.2), an anomaly occurring in nonclausal proof search (Section 3.8), an approach to conduct reproducible experiments (Section 3.10), and an improved reconstruction of Metis proofs (Section 6.3).

1.6 Outline

This thesis is structured as follows:

- In Chapter 2, we explain how to select premises using the machine learning algorithms k-nearest neighbours, Naive Bayes, and random forests. This chapter is based on [FK15b] published in Frontiers of Combining Systems (FroCoS).
- In Chapter 3, we explain how to implement connection proof search in functional programming languages to improve the speed of proof search and to integrate machine learning. This chapter is based on unpublished work submitted to Journal of Automated Reasoning (JAR) [FKU18].
- In Chapter 4, we explain internal guidance of given-clause provers with positive and negative examples and apply it to higher-order proof search. This chapter is based on [FB16] published in International Joint Conference on Automated Reasoning (IJCAR).
- In Chapter 5, we explain how to expand proof search trees using Monte Carlo Tree Search and present heuristics adapted to proof search. This chapter is based on [FKU17] published in International Conference on Automated Deduction (CADE-26).
- In Chapter 6, we explain how to translate higher-order problems to first-order problems and to translate first-order proofs to higher-order proofs. This chapter is based on [FK15a] published in Global Conference on Artificial Intelligence (GCAI) and unpublished work submitted to JAR [FKU18].

Chapter 2

Premise Selection

2.1 Introduction

The large number of facts present in mathematical corpora can pose a challenge to automated theorem provers. To increase the likelihood that an ATP quickly finds a proof, it can help to supply the ATP with a smaller set of facts that seems sufficient to make the ATP prove the conjecture. The selection of such facts is called *premise selection* (or *relevance filtering* or *fact selection*).

Definition 2.1 (Premise Selection)**.** Given a set of theorems T (i.e. a theorem corpus) and a conjecture c, *premise selection* returns a set of premises $P \subseteq T$ such that an ATP is likely to find a proof of $P \vdash c$ [AHK+14, BGK+16].

Premise selection is an important processing step before the translation of a problem to a different logic, as the complexity of the translation often depends on the lemmas to be translated. Therefore a good estimation of the facts that are useful for a proof can considerably increase the chances that an ATP finds a proof within a given time limit.

To find relevant premises, one can use information from previous proofs which premises were used to prove conjectures. We found that the following informal assumptions can be used to build fairly accurate premise selectors, when theorems are suitably characterised by features:

- Theorems sharing many features or rare features are similar.
- Theorems are likely to have similar theorems as premises.
- Similar theorems are likely to have similar premises.
- The fewer premises a theorem has, the more important they are.

In this chapter, we discuss machine learning methods to build premise selectors that implement the above assumptions. For that, we give an encoding of theorem corpora as machine learning input.

Definition 2.2 (Samples, Labels, Features)**.** Any proven theorem $l \in T$ of a given theorem corpus T gives rise to a training *sample* $(l, \vec{f}) \in S$, where l is a *label* and $\vec{f}$ is the set of *features* that characterises l. For every sample $s = (l, \vec{f})$, the function premises(s) returns the samples corresponding to the premises of l. Inversely, premises$^{-1}(s)$ returns all samples that have s as premise.

Example 2.3. We consider the HOL Light theorem `ADD_SYM`, which is defined in Listing 2.1 and states $\vdash \forall m\, n.\, m + n = n + m$. We can choose to characterise theorems by the

Listing 2.1: HOL Light proof of symmetry of addition.

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

constants and types present in their statements. This way, the sample s corresponding to `ADD_SYM` is $(\texttt{ADD_SYM}, \vec{f})$ with $\vec{f} = \{+, =, \forall, \texttt{num}, \texttt{bool}\}$. To record which theorems were used in a proof, we use a modified version of HOL Light. This yields that in the proof of `ADD_SYM`, `INDUCT_TAC` uses the premise `num_INDUCTION`, and `ASM_REWRITE_TAC` rewrites the goal using the premises `ADD_CLAUSES`, `REFL_CLAUSE`, and `FORALL_SIMP`. As `ADD_CLAUSES` is a conjunction of existing theorems and rewriting uses two of them, namely `ADD` and `ADD_SUC`, those theorems are also recorded as premises. We therefore encode the premises of s as $\text{premises}(s) = \{(\texttt{ADD_CLAUSES}, \vec{f_1}), (\texttt{ADD}, \vec{f_2}), (\texttt{ADD_SUC}, \vec{f_3}), (\texttt{REFL_CLAUSE}, \vec{f_4}), (\texttt{FORALL_SIMP}, \vec{f_5}), (\texttt{num_INDUCTION}, \vec{f_6})\}$, where $\left\{\vec{f_1}, \dots, \vec{f_6}\right\}$ are the features corresponding to the premises.[1]

This encoding allows us to view premise selection as a multi-label classification problem [TK07].

Definition 2.4 (Multi-Label Classifier)**.** Given a set of samples S, a *multi-label classifier* trained on S is a function r that takes a set of features $\vec{f}$ and returns a set of labels $\{l_1, \dots, l_n\}$.

Using multi-label classification, we can obtain suitable premises from a set of theorems S for a conjecture c as follows:

1. Obtain a multi-label classifier r for S.
2. Compute $\vec{f}$, the features of the conjecture c.
3. Return $r(\vec{f})$, the set of labels predicted by the classifier.

For premise selection, we demand an order on the set of labels returned by the multi-label classifier. We thus obtain a list of labels $[l_1, \dots, l_n]$. This additional information allows us to communicate to ATPs which premises are considered more useful to solve the given problem. By convention, the list of labels is sorted by decreasing estimated usefulness.

To evaluate a multi-label classification method, we proceed as follows: A dataset consists of a sequence of samples, some of which are *evaluation samples*. A multi-label classifier learns the samples of the dataset sequentially. Whenever the classifier encounters an evaluation sample $e = (l, \vec{f})$, the classifier predicts labels for $\vec{f}$ before learning e. The predicted labels are evaluated e.g. by comparing them with $\text{premises}(e)$. This procedure is explained in more detail in Section 2.6. We use datasets derived from following interactive theorem provers, with statistics given in Table 2.1:

- Mizar MPTP2078 [AHK+14] updated to Mizar 8.1.02 [KU15d],
- HOL Light SVN version 193 core library [KU15b], and
- Isabelle 2014 theory HOL/Probability together with its dependencies [KBKU13].

[1] It is possible to omit the features in $\text{premises}(s)$, because $\text{premises}(s) \subset S$ for all $s \in S$. However, we choose this encoding of premises for simpler presentation of the machine learning algorithms.

Table 2.1: Premise selection datasets. Averages are given per sample.

Dataset	Samples	Eval. smpl.	Features	Avg. feats.	Avg. premises
Mizar	3221	2050	3773	14.2	8.8
HOL Light	2548	2247	4331	13.4	2.6
Isabelle	23442	1656	31308	23.1	4.2

2.2 k-nearest neighbours

The *k-nearest neighbours* (k-NN) algorithm measures the distance of the learnt samples to the current features and returns the k closest ones [CH67]. For premise selection, we propose several adaptations, namely weighted proximity, premises, and adaptive k. We first introduce some functions.

Definition 2.5 (Indicator Function)**.** The *indicator function* or *characteristic function* $\mathbf{1}_A(x)$ returns 1 if $x \in A$ and 0 otherwise.

Definition 2.6 (f-rank)**.** The *f-rank* of x with respect to the set A is the number of elements in A for which the output of f is larger than $f(x)$, i.e. $\text{rank}_f(x, A) = |\{y \mid y \in A, f(x) < f(y)\}|$.

We now recall the classical k-NN algorithm. We assume the existence of a norm $\|\cdot\|$ and a subtraction operator $(-)$ on the feature space. Given a set of features $\vec{f}$, k-NN returns those k samples S_k whose features are closest to $\vec{f}$.

Definition 2.7 (Proximity)**.** The *proximity* of a sample $(l_i, \vec{f_i})$ is the inverse distance of the sample from the feature vector $\vec{f}$, i.e. $\text{prox}(l_i, \vec{f_i}) = 1/\|\vec{f} - \vec{f_i}\|$.

Definition 2.8 (k-nearest Neighbours)**.** The *k-nearest neighbours* are those samples with a proximity rank smaller than k, i.e. $S_k = \{s \in S \mid \text{rank}_{\text{prox}}(s, S) < k\}$. The labels of S_k are the output of the classifier.

We are now going to show the adaptations for k-NN to accommodate for premise selection.

We use *inverse document frequency* (IDF) to weigh the relevance of features [Jon73]. The idea is that the more features two feature sets share and the less frequent the shared features occur among all samples, the closer we consider the two feature sets to be. To this end, we replace the proximity in Definition 2.7 by the proximity in Definition 2.10.

Definition 2.9 (Inverse Document Frequency)**.** The *inverse document frequency* of a feature f is

$$\text{idf}(f) = \frac{|S|}{\left|\left\{(l_i, \vec{f_i}) \in S \mid f \in \vec{f_i}\right\}\right|}$$

Definition 2.10 (IDF-weighted Proximity)**.** The *IDF-weighted proximity* of a sample $(l_i, \vec{f_i})$ is $\sum_{f \in \vec{f} \cap \vec{f_i}} \text{idf}(f)$.

When a sample s is a k-nearest neighbour, then we also want to propose its premises, i.e. those samples that were used to prove s. We derive the relevance of a sample from the relevance of the k-nearest neighbours that have the sample as a premise. For this, let us define a new proximity function $\text{prox}'(s) = \text{prox}(s)/\log(\text{rank}_{\text{prox}}(s, S) + 1)$. Then the final relevance of a sample s is

$$r(s) = \mathbf{1}_{S_k}(s)\,\text{prox}'(s) + \sigma \sum_{p \in \text{premises}^{-1}(s) \cap S_k} \frac{\text{prox}'(s)}{|\text{premises}(p)|}$$

where σ is a constant that controls the relevance of premises.

In traditional k-NN, k is fixed, and only those labels with a rank smaller than k are predicted. For premise selection, using a bad value for k may cause the neighbours to be very similar and to have similar premises. Therefore, we use an adaptive k: We choose k such that the number of predicted labels (and their premises) is as small as possible, but fulfils the requested number of premises.

Example 2.11. We want to propose at least three potential premises for a conjecture with features $\vec{f}$. Let $S = \{s_1, s_2, s_3, s_4\}$ with $\text{prox}(s_1) > \text{prox}(s_2) > \text{prox}(s_3) > \text{prox}(s_4)$ and $\text{premises}(s_1) = \{s_3\}$, $\text{premises}(s_2) = \{s_1, s_3\}$, and $\text{premises}(s_3) = \{s_4\}$. This is illustrated in Figure 2.1. The 1-nearest neighbour of $\vec{f}$ is s_1. It has a single premise, namely s_3. Therefore for $k = 1$, we would predict s_1 and s_3. However, as we want to propose at least three premises, we consider $k = 2$. The 2-nearest neighbours of $\vec{f}$ are s_1 and s_2. Their premises are s_2 and s_3, so we end up with precisely three samples. The order in which they will be suggested is given by their relevances:

$$\begin{aligned}
r(s_1) &= \text{prox}'(s_1) \quad && + \sigma \frac{\text{prox}'(s_2)}{|\text{premises}(s_2)|} \\
r(s_2) &= \text{prox}'(s_2) && \\
r(s_3) &= && \sigma \left(\frac{\text{prox}'(s_1)}{|\text{premises}(s_1)|} + \frac{\text{prox}'(s_2)}{|\text{premises}(s_2)|} \right)
\end{aligned}$$

The proximity rank of s_1 is 0, and the proximity rank of s_2 is 1. Therefore $\text{prox}'(s_1) = \text{prox}(s_1)/\log(1)$ and $\text{prox}'(s_2) = \text{prox}(s_2)/\log(2)$.

2.3 Naive Bayes

The Naive Bayes classifier orders labels by the probability that they are useful in the presence of a set of features $\vec{f}$. The statistical independence of features is (naively) assumed and Bayes' theorem is applied.

We show a formula NB to rank labels based on Naive Bayesian probability that can be used for premise selection [KU15d] as well as for internal guidance, which we discuss in Chapter 4. The specialities of internal guidance with respect to premise selection are that premises of samples are not defined and labels can appear multiple times with different features. For this, we introduce a function $F(l)$, which returns the multiset of sets of

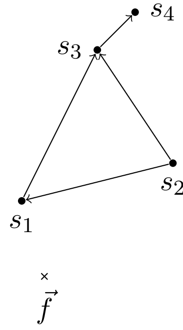


Figure 2.1: Samples in the feature space. If $a \to b$, then b is a premise of a.

features that co-occurred with l, i.e. $F(l) = \{\vec{f} \mid (l, \vec{f}) \in S\}$. The total number of times that l occurred is $|F(l)|$.

Example 2.12. $F(l_1) = \{\{f_1, f_2\}, \{f_2, f_3\}\}$ means that the label l_1 was used twice previously; once in a state characterised by the features f_1 and f_2, and once when features f_2 and f_3 were present.

Let $P(l_i, \vec{f})$ denote the probability that a label l_i from a set $\vec{l}$ of potential labels is useful in a state characterised by features $\vec{f}$. Using Bayes' theorem together with the (naive) assumption that features are statistically independent, we derive

$$P(l_i \mid \vec{f}) = \frac{P(l_i)P(\vec{f} \mid l_i)}{P(\vec{f})} = \frac{P(l_i)}{P(\vec{f})} \prod_{f_j \in \vec{f}} P(f_j \mid l_i)$$

To increase numerical stability, we calculate the logarithm of the probability

$$\ln P(l_i \mid \vec{f}) = \ln P(l_i) - \ln P(\vec{f}) + \sum_{f_j \in \vec{f}} \ln P(f_j \mid l_i)$$

In the final formula $\textsc{nb}(l_i, \vec{f})$ to rank labels, we modify $\ln P(l_i \mid \vec{f})$ as follows:

- We add a term to disadvantage features not present in $\vec{f}$ that occurred in previous situations with the label l_i.
- We weigh the probability of any feature f by its inverse document frequency $\operatorname{idf}(f)$ to give more weight to rare features.
- We drop the term $\ln P(\vec{f})$, as we compare only values for fixed features $\vec{f}$.
- We weigh the individual parts of the sum with constants σ_1, σ_2 and σ_3.

The resulting formula is

$$\begin{aligned}\textsc{nb}(l_i, \vec{f}) = {} & \sigma_1 \ln P(l_i) \\ & + \sigma_2 \sum_{f_j \in \vec{f}} \operatorname{idf}(f_j) \ln P(f_j \mid l_i) \\ & + \sigma_3 \sum_{f_j \in \bigcup F(l_i) \setminus \vec{f}} \operatorname{idf}(f_j) \ln(1 - P(f_j \mid l_i))\end{aligned}$$

The unconditional label probability $P(l_i)$ is calculated as follows:

$$P(l_i) = \frac{|F(l_i)|}{\sum_{l_j \in \vec{l}} |F(l_j)|}$$

In practice, as the denominator of the fraction is the same for all l_i, we drop it, similarly to $P(\vec{f})$ above.

To obtain the conditional feature probability $P(f_j \mid l_i)$, we distinguish whether a feature f_j already appeared in conjunction with a label l_i. If so, then its probability is the ratio of times f_j appeared when l_i was used and all times that l_i was used. Otherwise, the probability is estimated to be a minimal constant probability μ:

$$P(f_j \mid l_i) = \begin{cases} \sum_{\vec{f'} \in F(l_i)} \mathbf{1}_{\vec{f'}}(f_j) / |F(l_i)| & \text{if } \exists \vec{f'} \in F(l_i). f_j \in \vec{f'} \\ \mu & \text{otherwise} \end{cases}$$

2.4 Decision Trees

Decision trees are used in machine learning for classification and regression. They are also the underlying classification method for *random forests*, which we will discuss in Section 2.5.

Definition 2.13 (Binary Decision Tree, Splitting Feature)**.** A *binary decision tree* is either a branch (l, P, r) with a predicate P and two subtrees l and r, or a leaf. We say that a branch (l, P, r) has a *splitting feature* f_i iff $P(\vec{f}) \leftrightarrow f_i \in \vec{f}$.

Definition 2.14 (Samples and Features)**.** The features of a set of samples S are $\vec{f}(S) = \bigcup_{(l_i, \vec{f_i}) \in S} \vec{f_i}$. The samples of S that have and that do not have a certain feature f are

$$\begin{aligned} S_f &= \{s \in S \mid f \in \vec{f}(s)\}, \\ S_{\neg f} &= S \setminus S_f. \end{aligned}$$

Constructing a tree from a set of samples S involves either creating a leaf containing S, or creating a branch (l, P, r) with splitting feature $f \in \vec{f}(S)$ and trees l and r constructed from S_f and $S_{\neg f}$, respectively. Querying a branch (l, P, r) with a value x involves querying l if $P(x)$ is fulfilled, otherwise querying r. Querying a leaf returns the leaf. We explain building and querying of decision trees in more detail in the following sections.

A part of an example tree used for premise selection is shown in Figure 2.2. Leaf nodes have unique identifiers `t[yn]*`, which encode their position in the tree. The branch predicates verify the presence of certain symbols in a theorem, such as `plus`, and the data in the leaves are theorems that are relevant if the tree path to them corresponds to the symbols of the conjecture we seek to prove. For example, the branch node with feature `even(plus)` has a positive leaf node with four theorems, namely `even_sum`, `odd_plus_odd` (two times), and `odd_plus_even` – all having features `plus`

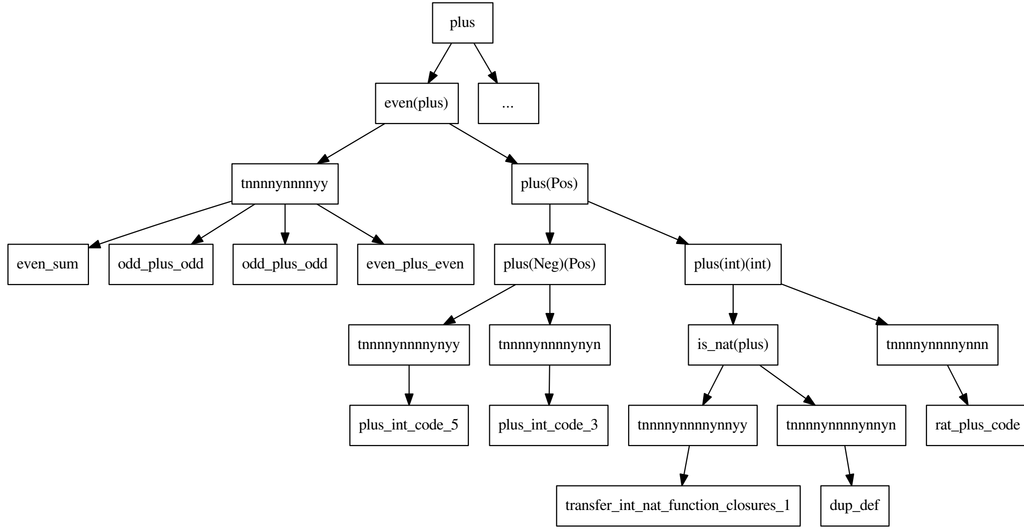


Figure 2.2: Excerpt from a decision tree trained on the Isabelle dataset.

and `even(plus)`. The theorem `plus_int_code_3` has features `plus` and `plus(Pos)`, but neither `even(plus)` nor `plus(Neg)(Pos)`.

To deal with the specifics of premise selection, we propose a number of extensions to decision trees, such as incremental learning, multi-path querying, depth weighting, and integration of secondary classifiers in the tree leaves.

2.4.1 Feature Selection

We determine a splitting feature for a set of samples S in two steps: First, one selects a set of candidate splitting features $\vec{f_\sigma} \subseteq \vec{f}(S)$. Then, one evaluates each of the features in $\vec{f_\sigma}$ to obtain a suitable splitting feature.

In [AGPV13], the candidate splitting features are obtained by randomly drawing with replacement (an element is drawn a set, then placed back in the set) a set of features $\vec{f_R}$ from $\vec{f}(S)$, where $n_R = \left|\vec{f_R}\right|$ is a user-defined constant. When we applied the method in the context of premise selection, we frequently obtained trees of small height with many labels at each leaf. This is because many features occur relatively rarely in our datasets. For example in the Mizar dataset there are 2026 features which occur only a single time among all samples, and only 34 that occur ten times, see Figure 2.3. Taking larger subsets of random features alleviates this problem, but it also makes the evaluation of the features slower. To increase performance, we determine for each feature in $\vec{f_R}$ how

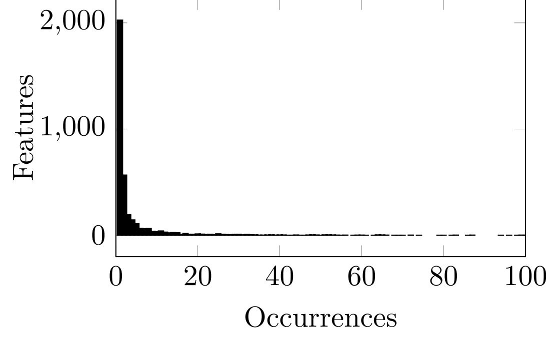


Figure 2.3: Feature histogram for the Mizar MPTP2078 dataset.

evenly it divides the set of samples in two, by evaluating

$$\sigma(S, f) = \frac{||S_f| - |S_{\neg f}||}{|S|}$$

The best output of $\sigma(S, f)$ for a feature is 0, which is the case when a feature splits the sample set S in two sets of exactly the same size, and the worst output is 1, when the feature appears either in all samples or in none. The final candidate splitting features $\vec{f_\sigma}$ then are those n_σ features of $\vec{f_R}$ that yield the best values for $\sigma(S, f)$. The motivation behind this is to preselect features which are more likely candidates to become splitting features, thus saving time in the evaluation phase.

Now we show how to select a splitting feature from the candidate splitting features $\vec{f_\sigma}$. To obtain a tree that is not too high, it is desirable for a splitting feature to split S evenly, such that S_f and $S_{\neg f}$ have roughly the same number of labels. Furthermore, the best splitting feature for a set of samples S should be a feature f which makes the samples in S_f and $S_{\neg f}$ more homogeneous compared to S [AGPV13]. Common measures to determine splitting features are information gain and Gini impurity [RS04, AGPV13]. We adapt Gini impurity to premise selection: Gini impurity measures the frequency of each premise among a set of samples, and gives premises with very high or very low frequency a low value. That means that the more similar the samples are (meaning they possess similar premises), the lower the Gini impurity.

Definition 2.15 (Gini Impurity)**.** The *Gini impurity* $g(S)$ of a set of samples S is

$$g(S) = \sum_{p \in \bigcup\{\text{premises}(s) | s \in S\}} P(p)(1 - P(p))$$

$$P(p) = \sum_{s \in S} P(p|s)P(s) \quad P(p|s) = \frac{\mathbf{1}_{\text{premises}(s)}(p)}{|\text{premises}(s)|} \quad P(s) = \frac{|\text{premises}(s)|}{\sum_{s' \in S} |\text{premises}(s')|}$$

In general, we look for a function $s(S, f)$, which determines the quality of f being a splitting feature for S. The best splitting feature can then be obtained by $\arg\min_{f \in \vec{f_\sigma}} s(S, f)$.

We evaluate two definitions for $s(S, f)$ in Section 2.6:

$$s_\sigma(S, f) = \sigma(S, f)$$
$$s_g(S, f) = \frac{1}{|S|}(|S_f|g(S_f) + |S_{\neg f}|g(S_{\neg f}))$$

While s_σ optimally divides S into two evenly sized sets S_f and $S_{\neg f}$, it does not take into account their homogeneity, unlike s_g, which considers their Gini impurity and their size.

2.4.2 Incremental Learning

Whenever a user proves a conjecture in an interactive theorem prover, premise selection methods should learn about the conjecture in order to be able to propose it in future proof attempts. Updating classifiers with new knowledge is called *online* or *incremental learning*, in contrast to *offline* learning, where all knowledge is learnt in one pass. We show three methods to learn decision trees. The first two are existing online and offline methods. We explain why they are not suitable for premise selection. Then, we present a third method, which is our incremental version of the second method.

Saffari et al. [SLS+09] present an online algorithm to learn decision trees. In this algorithm, all trees are initially leafs. Adding a sample to a leaf consists of adding the sample to the samples in the leaf. As soon as the number of samples in a leaf exceeds a certain threshold or a sufficiently good splitting feature for the sample set is found, the leaf splits into a branch with two leafs. When adding a sample to a branch, the sample gets added to the left or to the right child of the branch, according to whether or not the sample has the splitting feature of the branch. The method of Saffari et al. introduces a bias such that features which appear in early learned samples will be at the tree roots. Saffari et al. solve this problem by removing trees with a high prediction error (OOBE, out-of-bag error). However, this introduces a bias towards the latest learned samples, which is useful for computer graphics applications such as object tracking, but undesirable for premise selection, as the advice asked from a predictor will frequently not correspond to the last learned theorems.

Agrawal et al. [AGPV13] show an offline algorithm to learn decision trees. To learn a decision tree for a set of samples S, the algorithm first determines a splitting feature (explained in Subsection 2.4.1) for S. If $|S_f| < \mu$ or $|S_{\neg f}| < \mu$, where μ is the minimal number of samples which a leaf has to contain, the algorithm returns a leaf node containing S, otherwise the algorithm recursively calculates trees for S_f and $S_{\neg f}$ and combines them into a branch node with the splitting feature f. The approach of Agrawal et al. has several disadvantages when used for premise selection: While we need to learn data quickly and query only a few times after each learning phase, the algorithm of [AGPV13] is optimised to answer queries in logarithmic time, whereas its learning phase is relatively slow. Furthermore, because the algorithm is an offline algorithm, it rebuilds all trees to learn new samples.

We show an improved version of the offline algorithm given in [AGPV13] which updates decision trees with new samples. Given a tree t and a set of new samples S to learn, the algorithm calculates S', which is the union of S with all the samples in the leaf nodes of

t, and a splitting feature f for S'. If t is a branch (l, P, r) with f as a splitting feature, we recursively update l with S_f and r with $S_{\neg f}$. Otherwise, we construct a new tree for S': If $|S'_f| < \mu$ or $|S'_{\neg f}| < \mu$, we return a leaf node with S', otherwise we construct trees l' for S'_f and r' for $S'_{\neg f}$, returning a branch node (l', P', r') with f as splitting feature. This algorithm returns the same trees as the original algorithm, but can be significantly faster in case of updates. For example, predicting advice for the whole Mizar dataset takes 1287s with this optimisation and 3442s without.

We evaluated three functions to calculate μ, which depend on the samples of the whole tree, namely $\mu_{\log}(S) = \log |S|$, $\mu_{\text{sqrt}}(S) = \sqrt{|S|}$, and $\mu_{\text{const}}(S) = 1$. Agrawal et al. [AGPV13] use only $\mu_{\log}$. We show the results for each of these functions in Section 2.6.

2.4.3 Querying

To query a decision tree with features $\vec{f}$, a common approach is to recursively go to the left tree l of a branch node (l, P, r) if $P(\vec{f})$ and to the right tree r if not, until encountering a leaf with samples S, upon which S is returned. This approach frequently misses samples with interesting features when these do not completely correspond to the query features. This is why we consider a different kind of tree query, which we call *multi-path querying* (MPQ). In contrast to *single-path querying* (SPQ), MPQ considers not only the path with 100% matching features, but also all other paths in the tree. At each branch node where the taken path differs from the path foreseen by the splitting feature of the node, we store the depth d of the node.

Example 2.16. In Figure 2.4, we show an excerpt of a decision tree generated from the Isabelle/HOL Probability dataset. We assume that the query features (marked in green) are {`tSet.set`, `Set.member`}. The numbers next to the branches indicate the depth of wrongly taken decisions. These decisions are accumulated and shown below the samples at the bottom.

The output of a multi-path query for a tree t and features $\vec{f}$ is $mq(t, 0, \emptyset)$, where

$$mq(t, d, E) = \begin{cases} \{(S, d, E)\} & \text{if } t \text{ is leaf with samples } S \\ mq(l, d+1, E) \cup mq(r, d+1, E \cup \{d\}) & \text{if } t \text{ is branch } (l, P, r) \text{ and } P(\vec{f}) \\ mq(r, d+1, E) \cup mq(l, d+1, E \cup \{d\}) & \text{if } t \text{ is branch } (l, P, r) \text{ and } \neg P(\vec{f}) \end{cases}$$

The output of $mq(t, 0, \emptyset)$ is a set of triples (S, d, E), where S is a set of samples in some leaf node n of t, d is the depth of n in t, and E are the depths of the nodes on the path to n where the query features $\vec{f}$ do not correspond to the path.

We want to assign to each tree leaf a weight, which indicates how well the features $\vec{f}$ correspond to the features along the path from the root of the tree to the leaf. To do this, we consider the depths of the branch nodes where we took a different path than foreseen by $\vec{f}$, and calculate for each of the depths a weight, which we later combine to form a branch or sample weight.

For each $e \in E$, where $(S, d, E) \in mq(t, 0, \emptyset)$, we calculate a depth weight. We tried

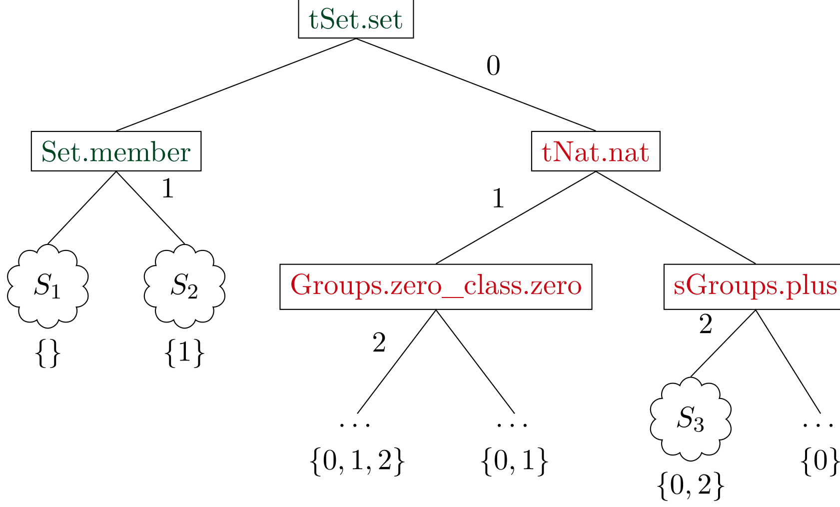


Figure 2.4: Multi-path query example.

different depth weight functions, where the constant μ represents the minimal weight:

$$\begin{aligned}
e_{\text{ascending}}(d, e) &= \mu + (1 - \mu)\left(\frac{e}{d}\right) \\
e_{\text{descending}}(d, e) &= 1 - (1 - \mu)\left(\frac{e}{d}\right) \\
e_{\text{inverse}}(d, e) &= 1 - \frac{1 - \mu}{e + 1} \\
e_{\text{const}}(d, e) &= \mu
\end{aligned}$$

Let us fix the used depth weight function to be e_i. Using the depth weights, we calculate a weight for each sample:

$$w_t(s) = \sum_{\substack{(S,d,E) \in mq(t,0,\emptyset) \\ s \in S}} \prod_{e \in E} e_i(d, e)$$

Regular decision trees with single-path querying return all the labels of the chosen branch. To order the results from multiple branches in a tree, which is necessary with multi-path querying, we run a secondary classifier on all the leaf samples of the tree. The secondary classifier is modified to take into account the weight of each branch. In our experiments, the secondary classifier is a k-NN algorithm adapted for premise selection (see Section 2.2), which we modified to accept sample weights: k-NN will give premises that appear in samples with higher weights precedence over those from samples with lower weights. In default k-NN, all samples would have weight 1, while in our secondary classifier, the weight of a sample s is given by $w_t(s)$, which stems from the path to s in the decision tree.

2.5 Random Forests

Random forests [Bre01] are a family of bagging algorithms [Bre96], meaning that they choose random subsets of data to build independent classifiers and combine their predictions to form a final prediction. In the case of random forests, the independent classifiers are decision trees. Random forests are known for high speed and quality in many domains [CN06]. Many different versions of random forests have been proposed [AGPV13, Bre96, LRT14, SLS$^+$09]. Random forests are used in applications where large amounts of data needs to be classified in a short time, such as automated proposal of advertisement keywords for web pages [AGPV13] or prediction of object positions in real-time computer graphics [SLS$^+$09].

To deal with the specifics of premise selection, we propose a number of extensions to random forests, such as different sample selection heuristics and incremental learning. Combining these extensions with the decision trees shown in Section 2.4, we improve upon the k-nearest neighbour algorithm both in terms of prediction quality and ATP performance. We will show this in Section 2.6.

2.5.1 Sample Selection

When learning new samples S, random forests determine which trees learn which samples. In [AGPV13], each tree in a forest randomly draws n samples from S. This approach may introduce a bias, namely that some samples are drawn more often than others, while some samples might not be drawn (and learned) at all. Therefore, instead of each tree drawing a fixed number of samples to learn, in our approach, each sample draws a fixed number of *trees* by which it will be learned. We call this fixed number *sample frequency* and denote it by f_s. This approach has the advantage that by definition, every sample is guaranteed to be learned as often as all other samples.

2.5.2 Incremental Learning

The method of Saffari et al. [SLS$^+$09] incrementally updates random forests: When learning a new sample, it is added to all trees with a probability determined by a Poisson distribution with $\lambda = 1$ [OR01]. We adopt this use of probability distributions to create incrementally learning versions of offline bagging algorithms as follows.

Given a bagging algorithm (such as random forests) whose individual predictors (in our scenario the decision trees of the forest) learn a random subset of samples offline, we show a method to decrease the runtime of incrementally learning new data. The method is based on the following observation: When learning only a small number of new samples (compared to the number of samples already learned), most predictors will not include any of those new samples, thus they do not need to be updated. To model this, let r be a binomially distributed random variable $r \sim B(s, P)$, where s is the number of samples in each predictor and $P = \frac{n_{\text{new}}}{n_{\text{new}} + n_{\text{old}}}$ is the probability of drawing a new sample from the common pool of new and old samples. The number of new samples drawn by a predictor is then modelled by r. Each predictor evaluates the random variable r, and if its value

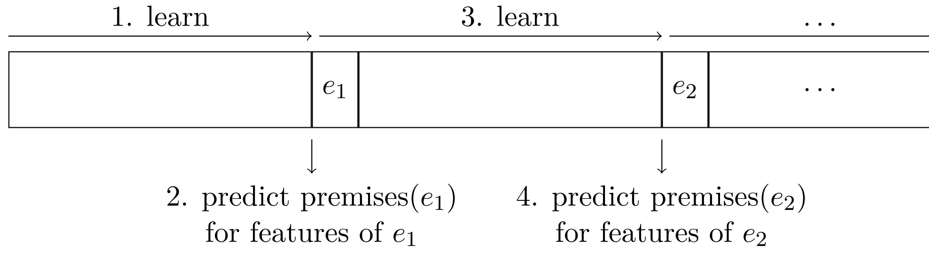


Figure 2.5: In an evaluation, a set of samples is learned until an evaluation sample e_i is encountered, for which premises(e_i) are predicted.

r_p is 0, the predictor remains unchanged. Only if r_p is greater than 0, the predictor is retrained with r_p samples from the set of new samples and $s - r_p$ samples from the set of old samples. This method gives a performance increase over always rebuilding all predictors.

2.5.3 Querying

We query a forest with a set of features $\vec{f}$ by querying each tree in the forest with $\vec{f}$, combining the prediction sequences L of all trees. For each label l, we calculate its rank in a prediction sequence $\vec{l} = [l_1, \dots, l_n]$ as:

$$\varrho(l, \vec{l}) = \begin{cases} i & \text{if } l = l_i \text{ and } l_i \in \vec{l} \\ m & \text{otherwise} \end{cases}$$

Here, m is a maximal rank attributed to labels that do not appear in a prediction sequence. Then, for each label, we calculate its ranks $R(l) = \biguplus_{\vec{l} \in L} \varrho(l, \vec{l})$ for all prediction sequences. We sort the labels by the arithmetic, quadratic, geometric, or the harmonic mean of $R(l)$ in descending order to obtain the final prediction sequence.

2.6 Evaluation

We explain how to evaluate predictor performance on a sequence of samples. For this, we define a subset of the samples as *evaluation samples*, for which the classifier will predict premises by iterating over all samples in order and predicting premises for each evaluation sample e before learning e, as illustrated in Figure 2.5. We can evaluate the quality of the predictions in two ways: First, the predictions can be translated to an ATP problem and given to an automated theorem prover. Second, they can be compared to the actual labels of the evaluation samples. For the second method, we will show two quality measures, namely n-Precision and AUC [ZZ14]. We first introduce some notation: Given a sequence of distinct elements $\vec{x} = [x_1, \dots, x_n]$, we denote $\vec{x}_i^e = [x_i, x_{i+1}, \dots, x_e]$. Furthermore, when it is clear from the context, we treat sequences as sets, where the set elements are the elements of the sequence.

The first measure, n-Precision, is similar to Precision: Precision computes the percentage of premises from the training sample appearing among the predicted premises. The n-Precision considers only the first n predictions, which corresponds to our passing only a fixed maximal number of premises to ATPs. If not stated otherwise, we use 100-Precision in our evaluations.

Definition 2.17 (n-Precision)**.** The *n-Precision* for a sequence of predicted premises $\vec{p}$ and a set of real premises $\vec{r}$ is

$$\mathrm{Prec}_n(\vec{p}, \vec{r}) = \frac{|\vec{p}_1^n \cap \vec{r}|}{|\vec{r}|}$$

The second measure, AUC, models the probability that for a randomly drawn real premise $r_i \in \vec{r}$ and a randomly drawn non-real predicted premise $p_j \in \vec{p} \setminus \vec{r}$, r_i appears in the predictions before p_j.

Definition 2.18 (AUC)**.** The *AUC* for a sequence of predicted premises $\vec{p}$ and a set of real premises $\vec{r}$ is

$$\mathrm{AUC}(\vec{p}, \vec{r}) = \begin{cases} \dfrac{\sum_{n=1}^{|\vec{p}|} |\vec{p}_1^n \cap \vec{r}|}{|\vec{r}| \times |\vec{p} \setminus \vec{r}|} & \text{if } |\vec{r}| \times |\vec{p} \setminus \vec{r}| > 0 \\ 1 & \text{if } |\vec{r}| \times |\vec{p} \setminus \vec{r}| = 0 \end{cases}$$

We implemented the presented premise selection methods in Haskell.[2] Our initial implementation of random forests following [AGPV13] was several magnitudes slower than k-NN even for small datasets, rendering it impracticable for incremental learning. Furthermore, the prediction quality was lower than expected: For the first 200 evaluation samples of the Mizar dataset, a random forest with 4 trees and 16 random features evaluated at every tree branch achieved an AUC of 82.96% in 82s, whereas k-nearest neighbours achieved an AUC of 95.84% in 0.36s.

Our default random forest configuration (RF) uses 4 trees with a sample frequency of 16, the sample selection method "samples draw trees", the minimal sample function $\mu_{\log}$, no Gini impurity, the depth weight function e_{Inverse} with $\mu = 0.8$. The final prediction is obtained by running k-NN with IDF over the weighted leaf samples of each tree, combining results with the harmonic mean.

The experimental results of our improved random forests for premise selection are given in Table 2.2: Random forests achieve the best results when combined with multi-path querying and path-weighted k-NN+IDF classifier in the leaves. Both considering Gini impurity and taking random subsets of features decreases the prediction quality and has a very negative impact on runtime. Different sample selection methods (samples draw trees vs. trees draw samples) have a large impact when using small sample frequencies, but when using higher sample frequencies, the difference is negligible. In this evaluation, we simulated single-path querying (SPQ) by a constant depth weight with $\mu = 0$ (meaning that all non-perfect tree branches receive the minimal score 0). Running this method

[2] The source code can be obtained at http://cl-informatik.uibk.ac.at/users/mfaerber/predict.html.

Table 2.2: Results for Mizar dataset. $\sum t$ is the prediction time for the whole dataset, and $\bar{t}$ is the average prediction time per evaluation sample.

Configuration	100-Prec [%]	AUC [%]	$\sum t$ [min]	$\bar{t}$ [s]
k-NN + IDF	87.5	95.39	0.5	**0.02**
RF (IDF)	88.0	95.68	32	0.93
RF (no IDF)	77.8	91.40	25	0.75
RF (single-path query)	53.7	60.86	37	1.07
RF ($f_s = 2$, trees draw s.)	65.6	72.76	2	0.05
RF ($f_s = 2$, samples draw t.)	88.0	95.59	4	0.10
RF (random features $n_R = 32$)	88.0	95.65	151	4.41
RF (Gini impurity, $n_\sigma = 2$)	88.0	95.65	97	2.84
RF (Gini impurity, $n_\sigma = 16$)	88.0	95.62	220	6.44
RF ($e_{\text{ascending}}$)	88.0	95.72	36	1.07
RF ($e_{\text{descending}}$)	88.1	95.66	39	1.15
RF (e_{inverse})	88.0	95.68	38	1.12
RF (e_{const})	88.1	95.81	37	1.08
RF (arithmetic mean)	87.5	95.49	33	0.98
RF (geometric mean)	88.0	95.67	35	1.01
RF (quadratic mean)	87.4	95.34	33	0.97
RF (100 trees, $f_s = 50$)	88.5	95.85	137	4.01
RF (24 trees, $f_s = 12$)	88.5	95.83	31	0.90
RF (24 trees, $f_s = 12$, e_{const})	**88.6**	**95.91**	22	0.66

takes longer than real SPQ, but gives a good upper bound on SPQ's prediction quality. Random forests have a longer runtime than k-NN, but still, the average prediction time for our test set is below one second, which is sufficient for premise selection in interactive theorem provers. We are going to show that despite the seemingly small improvement in 100-Prec and AUC, using random forests instead of k-NN can considerably decrease the overall time needed to solve the same number of problems.

To produce the number of proven theorems in Table 2.3, we predict premises (at most 128 for Mizar and 1024 for HOL Light) for each conjecture, translate the chosen facts together with the conjecture to TPTP first-order formulas [Sut09b] and run E 1.8 [Sch13] with automatic strategy scheduling and 30s timeout.

Alama et al. [AHK$^+$14] have reported 548 proven theorems with Vampire (10s timeout) without external premise selection, which their best premise selection method (MOR-40/100) increases to 824 theorems (+50.4%). On our data, E (10s timeout) without premise selection proves only 414 theorems, increasing with 30s timeout to 653 theorems (+57.7%) and with 10s timeout and RF premise selection to 962 (+132.3%).

In Table 2.4, we compare ATP runtime required to prove the same number of theorems using k-NN and RF predictions. While RF classification requires more runtime than k-NN, the ATP timeout can be decreased by more than 25%, resulting in overall runtime

Table 2.3: Results of k-NN and random forest predictions on two datasets. For random forests, we used the best configuration from Table 2.2, i.e. 24 trees, sample frequency 12, and constant depth weight.

Data / Predictor	100-Prec [%]	AUC [%]	Proved
Mizar / k-NN	87.5	95.39	931
Mizar / RF	88.6	95.91	959 (+3.0%)
HOL Light / k-NN	91.9	95.65	789
HOL Light / RF	92.9	96.29	823 (+4.3%)

Table 2.4: Comparison of runtime necessary to achieve the same number of proven theorems (969) for the Mizar dataset.

Classifier	Classifier runtime	E timeout	E runtime	Total runtime
k-NN	0.5min	15s	341min	341min
RF	22min	10s	252min	272min

reduction of about 20%.

In Figure 2.6, we show how the prediction quality develops for the Mizar dataset as more data is learned: For this purpose, we calculated statistics for the predictions of just our first evaluation sample, then for the first two, etc. When comparing the output of our random forest predictor (24 trees, sample frequency 12, constant depth weight) with k-NN, we see that it consistently performs better.

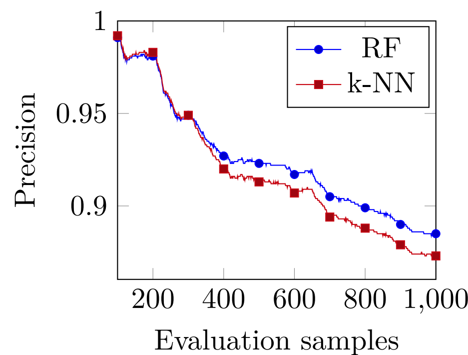


Figure 2.6: Comparison of k-NN with random forests by number of evaluation samples on Mizar dataset.

2.7 Related Work

The Meng-Paulson relevance filter (MePo) integrated into Isabelle/HOL as part of Sledgehammer was one of the first premise selectors for ITPs [MP09]. It is an iterative algorithm which counts function symbols in clauses and compares them to the function symbols in the conjecture to prove. In contrast to many other premise selectors, MePo does not consider the premises used to prove similar theorems.

Later approaches applied established machine learning methods for multi-label classification [ZZ05] to premise selection, such as kernel methods [AHK+14], PageRank [KU15c], k-nearest neighbours, and Naive Bayes [BGK+16].

Naive Bayes was the first machine learning algorithm used in an automated reasoning loop, and thanks to premises, the prediction quality improved upon syntactic tools [Urb04]. Simple perceptron networks have also been evaluated for HOL(y)Hammer predictions [KU14], and while their results are weak, they are complementary to other methods.

Machine learning algorithms such as k-nearest neighbours [ZZ05] and Naive Bayes were integrated into Sledgehammer as part of MaSh (Machine learning for Sledgehammer) [BGK+16], significantly improving ATP performance on the translated problems. The single most powerful method used for premise selection in HOL(y)Hammer, MizAR, and Sledgehammer/MaSh is a customized implementation of k-NN [KU14]. Stronger machine-learning methods that use kernel-based multi-output ranking (MOR [AHK+14] and MOR-CG [Kü14]) or deep neural networks [ISA+16, WTWD17] were found to perform better, at the cost of longer training or prediction times.

Premise selection is also used in ATPs to reduce original problems before the actual proof search. For example, SInE (Sumo Inference Engine) [HV11] improves the performance of the Vampire theorem prover [KV13] when working with large theories. SInE has also been implemented as a part of E [Sch13]. Premise selection has become especially important in the "large theory bench" division added to CASC in 2008 [Sut09a], with systems such as MaLARea [USPV08] and E.T. [KSUV15] achieving notable results.

2.8 Conclusion

Premise selection benefits from the usage of machine learning. We showed versions of k-nearest neighbours and Naive Bayes adapted to premise selection. Furthermore, we investigated random forests as premise selection method, in particular versions of it that are able to deal with the large amounts of learning data present in mathematical corpora. We evaluated several random forest approaches for ATP premise selection: Without modifications, the algorithms return worse predictions than the current state-of-the-art premise selectors included in HOL(y)Hammer, MizAR, and Sledgehammer/MaSh, and the time needed to select facts from a larger database is significant. We then proposed a number of extensions to the random forest algorithms designed for premise selection, such as incremental learning, multi-path querying, and various heuristics for the choice of samples, features and size of the trees. We combined random forests with a k-NN

predictor at the tree leaves of the forest, which increases the number of theorems from the HOL Light dataset that E can successfully reprove over the previous state-of-art classifier k-NN by 4.3%. We showed that to attain the same increase with k-NN, it is necessary to run E for 50% longer.

In scenarios where the number of queries is large in comparison with the number of learning phases, the random forest approach is an effective way of improving prediction quality while keeping runtime acceptable. This is the case for usage in systems such as HOL(y)Hammer and MizAR, but not for Sledgehammer, where data is relearned more frequently. The performance of random forests could still be improved by recalculating the best splitting feature only after having seen a certain minimal number of new samples since the last calculation of the best feature. This would improve learning speed while not greatly altering prediction results, because it is relatively unlikely that adding few samples to a big tree changes the tree's best splitting feature. Further runtime improvements could be made by parallelising random forests.

Chapter 3

Connection Proof Search

3.1 Introduction

The connection calculus [Bib91] was introduced as a variant of tableaux [LS01]. Connection calculi enable goal-directed proof search in a variety of logics. Connections were considered among others for classical first-order logic [LSBB92], for higher-order logic [And89] and for linear logic [Gal00].

An important family of connection provers for first-order logic is derived from leanCoP [OB03, Ott08]. leanCoP was inspired by leanTAP [BP95], which is a prover based on free-variable semantic tableaux. leanTAP popularised *lean theorem proving*, which uses Prolog to maximise efficiency while minimising code. The compact Prolog implementation of *lean theorem provers* made them attractive for experiments both with the calculus and with the implementation. For example, leanCoP has been adapted for intuitionistic (ileanCoP [Ott05]), modal (MleanCoP [Ott14]), and nonclausal first-order logic (nanoCoP [Ott16]). The intuitionistic version of leanCoP [Ott05] became the state-of-art prover for first-order problems in intuitionistic logic [ROK07]. A variant of leanCoP with interpreted linear arithmetic (leanCoP-Ω) won the TFA division of CASC-J5 [Sut11]. Various implementation modifications can be performed very elegantly, such as search strategies, scheduling, restricted backtracing [Ott10], randomization of the order of proof search steps [RO08], and internal guidance [UVv11, KU15a].

We have used connection provers from the leanCoP family as a basis for experiments with *machine learning* (see Chapter 5) and *proof certification* (see Section 6.4). For these applications, we implemented connection provers in functional instead of logic programming languages. There are several reasons: First, a large number of interactive theorem provers (ITPs), such as HOL Light, HOL4, Isabelle, Coq, and Agda are written in functional programming languages, lending themselves well to integration of functional proof search tactics. Second, several machine learning algorithms have been recently implemented efficiently for these ITPs in functional languages. Third, we achieve better performance with functional-style implementations, which is important to compensate for the performance penalty incurred by machine learning.

We first describe connection tableaux calculi in Section 3.2. We then discuss different aspects of implementing automated theorem provers for the given calculi, namely problem preprocessing in Section 3.3, consistent Skolemisation in Section 3.4, connection search in Section 3.5 and functional-style proof search in Section 3.6. We describe our findings related to nonclausal proof search in Section 3.7 and Section 3.8. Finally, we evaluate

our implementations in Section 3.9 and introduce our approach to create reproducible experiments in Section 3.10.

3.2 Connection Calculi

Connection calculi provide a goal-oriented way to search for proofs in classical and nonclassical logics [Ott08]. Common to these calculi is the concept of connections $\{P, \neg P\}$ between literals P and $\neg P$, which correspond to closing a branch in the tableaux calculus [Häh01].

Connection tableaux calculi, such as [Bib83], are members of the family of connection calculi. As the calculi considered in this thesis have a very small set of rules, they lend themselves very well to proof translation and machine learning.

In this section, we introduce the *clausal* and the *nonclausal connection calculus* that we will use throughout this thesis.

The connection calculi in this thesis operate on *matrices*, where a matrix is a set of clauses. In the nonclausal calculus, clauses do not only contain literals, but also matrices, giving rise to a nested structure. We use the symbols M for a matrix, C for a clause, L for a literal, x for a variable, and $\vec{x}$ for a sequence of variables, as in $\forall \vec{x}. P(\vec{x})$. A substitution σ is a mapping from variables to terms. The *complement* $\overline{L}$ is A if L has the shape $\neg A$, otherwise $\overline{L}$ is $\neg A$. A *σ-complementary connection* $\{L, L'\}$ exists if $\sigma\overline{L} = \sigma L'$. Given a relation R, its transitive closure is denoted by R^+ and its transitive reflexive closure by R^*.

We will focus on two variants of the calculus: clausal and nonclausal. As the two alternatives will differ only in the clauses and rules, we first give a definition of the common parts of the clausal and nonclausal connection calculi, omitting the calculus rules.

Definition 3.1 (Connection Calculus, Connection Proof)**.** The words of a *connection calculus* are tuples $\langle C, M, \mathit{Path} \rangle$, where C is a clause, M is a matrix, and Path is a set of literals called the *active path*. C and Path can also be empty, denoted ε. In the calculus rules, σ is a term substitution and $\{L, L'\}$ is a σ-complementary connection. The substitution σ is global (or *rigid*), i.e. it is applied to the whole derivation. A *connection proof* for $\langle C, M, \mathit{Path} \rangle$ is a derivation in a connection calculus for $\langle C, M, \mathit{Path} \rangle$ in which all leaves are axioms. A connection proof for M is a connection proof for $\langle \varepsilon, M, \varepsilon \rangle$.

To complete the definitions of the variants of the connection calculus, we need to specify the types of clauses and the rules. In the clausal connection calculus, a clause is a set of literals. The calculus rules are presented in Figure 3.1.

In the nonclausal connection calculus, a clause is a set of literals and matrices. The following definitions of the concepts used in the extension rule follow Otten [Ott11, Ott16].

Definition 3.2 (Clause Predicates)**.** A clause C *contains* L iff $L \in^+ C$. A clause $C \in^+ M$ is *α-related* to a literal L iff there is some $M' \in^* M$ with $\{C_L, C_C\} \subseteq M'$ such that $C_L \neq C_C$, $L \in^+ C_L$, and $C \in^* C_C$. A clause C' is a *parent clause* of C iff $M' \in C'$ and $C \in M'$ for some matrix M'.

$$
\begin{array}{ll}
\text{Axiom} & \dfrac{}{\{\}, M, Path}\,\text{A} \\[2ex]
\text{Start} & \dfrac{C_2, M, \{\}}{\varepsilon, M, \varepsilon}\,\text{S} \quad \text{where } C_2 \text{ is copy of } C_1 \in M \\[2ex]
\text{Reduction} & \dfrac{C, M, Path \cup \{L'\}}{C \cup \{L\}, M, Path \cup \{L'\}}\,\text{R} \quad \text{where } \sigma(L) = \sigma(\overline{L'}) \\[2ex]
\text{Extension} & \dfrac{C_2 \setminus \{L'\}, M, Path \cup \{L\} \quad C, M, Path}{C \cup \{L\}, M, Path}\,\text{E} \\
& \text{where } C_2 \text{ is copy of } C_1 \in M \text{ and } L' \in C_2 \text{ with } \sigma(L) = \sigma(\overline{L'})
\end{array}
$$

Figure 3.1: Clausal connection calculus rules.

Definition 3.3 (Clause Functions)**.** A *copy of the clause* C in the matrix M is created by replacing all free variables in C with fresh variables. $M[C_1 \backslash C_2]$ denotes the matrix M in which the clause C_1 is replaced by the clause C_2.

Definition 3.4 (Extension Clause, β-clause)**.** C is an *extension clause* (*e-clause*) of the matrix M with respect to a set of literals $Path$ iff either (a) C contains a literal of $Path$, or (b) C is α-related to all literals of $Path$ occurring in M and if C has a parent clause, that parent clause contains a literal of $Path$. The *β-clause* of C_2 with respect to L_2 is C_2 with L_2 and all clauses that are α-related to L_2 removed.

The rules of the nonclausal calculus are shown in Figure 3.2. The difference in the calculus rules to the clausal variant is the addition of a decomposition rule, and the adaptation of the extension rule to the nonclausal setting.

Given an order $<$, we can write sets as ordered sequences $[X_1, \ldots, X_n]$, where for all $i < n$, $X_i < X_{i+1}$. Clauses and matrices can thus be shown as horizontal and vertical sequences, respectively.

Example 3.5. Consider the following formula F and its prenex conjunctive normal form F'. We will show that they imply $\bot$:

$$
\begin{aligned}
F &= Q \land P(a) \land \forall x.(\neg P(x) \lor (\neg P(s^2x) \land (P(sx) \lor \neg Q))) \\
F' &= \forall x.(Q \land P(a) \land (\neg P(x) \lor \neg P(s^2x)) \land (\neg P(x) \lor P(sx) \lor \neg Q))
\end{aligned}
$$

For brevity, we write sx for $s(x)$ and s^2x for $s(s(x))$. The nonclausal matrix M corresponds to F and the clausal matrix M' to F':

$$
\begin{aligned}
M &= \left[[Q][P(a)] \left[\begin{matrix} \neg P(x) \\ \left[[\neg P(s^2x)] \begin{bmatrix} P(sx) \\ \neg Q \end{bmatrix} \right] \end{matrix} \right] \right] \\
M' &= \left[[Q][P(a)] \begin{bmatrix} \neg P(x) \\ \neg P(s^2x) \end{bmatrix} \begin{bmatrix} \neg P(x) \\ P(sx) \\ \neg Q \end{bmatrix} \right]
\end{aligned}
$$

Axiom $$\frac{}{\{\}, M, Path}\,\text{A}$$

Start $$\frac{C_2, M, \{\}}{\varepsilon, M, \varepsilon}\,\text{S} \quad \text{where } C_2 \text{ is copy of } C_1 \in M$$

Reduction $$\frac{C, M, Path \cup \{L'\}}{C \cup \{L\}, M, Path \cup \{L'\}}\,\text{R} \quad \text{where } \sigma(L) = \sigma(\overline{L'})$$

Extension $$\frac{C_3, M[C_1 \backslash C_2], Path \cup \{L\} \quad C, M, Path}{C \cup \{L\}, M, Path}\,\text{E}$$

where C_3 is the β-clause of C_2 with respect to L', C_2 is copy of C_1, C_1 is e-clause of M with respect to $Path \cup \{L\}$, C_2 contains L' with $\sigma(L) = \sigma(\overline{L'})$

Decomposition $$\frac{C \cup C', M, Path}{C \cup \{M'\}, M, Path}\,\text{D} \quad \text{where } C' \in M'$$

Figure 3.2: Nonclausal connection calculus rules.

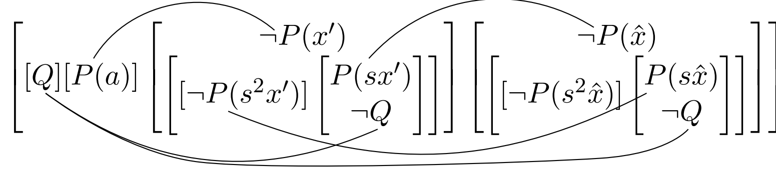


Figure 3.3: Nonclausal graphical connection proof.

Nonclausal and clausal graphical proofs for the problem are given in Figures 3.3 and 3.4: There, lines represent connections, and the substitution used is $\sigma = \{x' \mapsto a, \hat{x} \mapsto sx', \bar{x} \mapsto x'\}$. A formal proof for M' in the clausal connection calculus is given in Figure 3.5. A shorter proof for M' as well as a formal proof for M will be given using slightly modified versions of the calculi in Subsection 6.4.1.

Soundness and completeness have been proved both for the clausal [LS01] and for the nonclausal calculus [Ott11].

In the next sections, we develop an efficient implementation of a connection prover for classical first-order logic in a functional programming language. The resulting implementation will be the basis for all experiments with connection provers in the

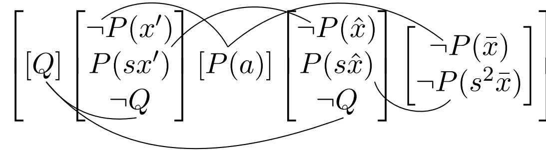


Figure 3.4: Clausal graphical connection proof.

$$
\dfrac{
  \dfrac{}{\{\}, M, \{Q, \neg P(x')\}}\,\mathrm{A}
  \qquad
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{
          \dfrac{}{\{\}, M, \ldots}\,\mathrm{A}
          \quad
          \dfrac{}{\{\}, M, \ldots}\,\mathrm{A}
        }{\{\neg P(\bar{x})\}, M, \{Q, P(sx'), P(s\hat{x})\}}\,\mathrm{E}
        \quad
        \dfrac{
          \dfrac{}{\{\}, M, \{Q, P(sx')\}}\,\mathrm{A}
        }{\{\neg Q\}, M, \{Q, P(sx')\}}\,\mathrm{R}
      }{\{P(s\hat{x}), \neg Q\}, M, \{Q, P(sx')\}}\,\mathrm{E}
      \quad
      \dfrac{}{\{\}, M, \{Q\}}\,\mathrm{A}
    }{\{P(sx')\}, M, \{Q\}}\,\mathrm{E}
  }{\{\neg P(x'), P(sx')\}, M, \{Q\}}\,\mathrm{E}
  \quad
  \dfrac{}{\{\}, M, \{\}}\,\mathrm{A}
}{
  \dfrac{\{Q\}, M, \{\}}{\epsilon, M, \epsilon}\,\mathrm{S}
}\,\mathrm{E}
$$

Figure 3.5: Clausal connection proof.

remainder of this thesis. The connection prover performs the following tasks. Given a classical first-order logic problem, it creates a matrix for the problem, see Section 3.3. The matrix is then used to build an index which provides an efficient way to find connections during proof search, see Section 3.5. Finally, proof search with iterative deepening is performed, see Section 3.6.

3.3 Problem Preprocessing

In this section, we show how the prover transforms problems to formulas and processes them to yield a matrix. We focus on first-order logic problems represented as a set of axioms $\{A_1, \ldots, A_n\}$ together with a conjecture C, where all axioms and the conjecture are closed formulas. The goal is to show that the axioms imply the conjecture. For convenience, in the actual implementation we use the TPTP format [Sut09b] as input. Each parsed input problem is transformed according to the following procedure. Only the steps 2 and 6 differ in comparison with the original Prolog implementations of leanCoP and nanoCoP [Ott08, Ott16].

1. The conjecture C is combined with the axioms $\{A_1, \ldots, A_n\}$ to form the new problem $(A_1 \land \cdots \land A_n) \to C$ (just C if no axioms are present).
2. Constants and variables are mapped to integers, to enable more efficient lookup and comparison during the proof search, as needed e.g. for fast unification.
3. As the connection tableaux calculi considered in this thesis do not have special rules for equality, equality axioms are added to the problem if equality appears in the original problem. The axioms are symmetry, reflexivity, and transitivity

$$\forall x.x = x \qquad (\text{symm}_=)$$
$$\forall xy.x = y \to y = x \qquad (\text{comm}_=)$$
$$\forall xyz.x = y \land y = z \to x = z \qquad (\text{trans}_=)$$

   as well as congruence:
   - For every n-ary function f, the formula $x_1 = y_1 \to \ldots \to x_n = y_n \to f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)$ is introduced.

- For every n-ary predicate P, the formula $x_1 = y_1 \to \ldots \to x_n = y_n \to P(x_1, \ldots, x_n) \to P(y_1, \ldots, y_n)$ is introduced.

4. If the formula has the shape $P \to C$, then it is transformed to the equivalent $(P \land \#) \to (C \land \#)$. $\#$ is a marker that can be understood to be equivalent to $\top$. It allows proof search to recognise clauses stemming from the conjecture [Ott08, section 2.1].
5. Implications and equivalences are expanded, e.g. $A \to B$ becomes $\neg A \lor B$.
6. Quantifiers are pushed inside such that their scope becomes minimal.
7. The formula is negated (to perform a proof by refutation) and converted to negation normal form.
8. The formula is reordered such that smaller clauses are processed earlier. In nanoCoP, the size of a formula is

$$\text{paths}(t) = \begin{cases} \text{paths}(t_1) \times \text{paths}(t_2) & \text{if } t = t_1 \land t_2 \\ \text{paths}(t_1) + \text{paths}(t_2) & \text{if } t = t_1 \lor t_2 \\ \text{paths}(t_1) & \text{if } t = \forall x.t_1 \text{ or } t = \exists x.t_1 \\ 1 & \text{if } t \text{ is a literal} \end{cases}$$

   and for any subformula $t_1 \land t_2$ or $t_1 \lor t_2$, if $\text{paths}(t_1) > \text{paths}(t_2)$, then t_1 and t_2 are exchanged.
9. The formula is Skolemised. For machine learning, we use consistent Skolemisation as discussed in Section 3.4 instead of outer Skolemisation as performed in the original Prolog version.

Example 3.6. Consider the axioms

$$\forall xAB.x \in A \cup B \leftrightarrow (x \in A \lor x \in B) \qquad (\text{def}_\cup)$$

$$\forall AB.(\forall x.x \in A \leftrightarrow x \in B) \to A = B \qquad (\text{def}_=)$$

that we want to use to prove

$$\forall ABC.A \cup (B \cup C) = (A \cup B) \cup C \qquad (C)$$

The problem is preprocessed as follows:

1. The axioms $A \equiv \text{def}_\cup \land \text{def}_=$ and the conjecture are combined, resulting in $A \to C$.
2. Constants and variables are mapped to integers, e.g. $\{"\in" \mapsto 0, "\cup" \mapsto 1, "=" \mapsto 2\}$ and $\{"x" \mapsto 0, "A" \mapsto 1, "B" \mapsto 2\}$. We will continue the presentation of this example with the original representation.
3. Congruence axioms are generated for all constants, i.e. "$\in$" and "$\cup$":

$$\forall x_1 y_1 x_2 y_2.(x_1 = x_2 \land y_1 = y_2) \land x_1 \in y_1 \to x_2 \in y_2 \qquad (\text{cong}_\in)$$

$$\forall x_1 y_1 x_2 y_2.(x_1 = x_2 \land y_1 = y_2) \to x_1 \cup y_1 = x_2 \cup y_2 \qquad (\text{cong}_\cup)$$

   The combination of all equality axioms is

$$((\text{sym}_= \land (\text{comm}_= \land \text{trans}_=)) \land \text{cong}_\in) \land \text{cong}_\cup \qquad (E)$$

   and the resulting formula is $E \land A \to C$.

4. The conjecture is marked, resulting in $((E \land A) \land \#) \rightarrow (\# \land C)$.
5. Implications and equivalences are unfolded. Among others, this transforms

$$\forall xAB.(x \notin A \cup B \lor (x \in A \lor x \in B)) \land (\neg(x \in A \lor x \in B) \lor x \in A \cup B) \quad (\text{def}_\cup)$$

$$\forall AB.((\neg \forall x.((x \notin A \lor x \in B) \land (x \notin B \lor x \in A))) \lor A = B) \quad (\text{def}_=)$$

   The resulting formula is $\neg((E \land A) \land \#) \lor (\# \land C)$.
6. Pushing quantifiers inside transforms for example

$$\begin{aligned}&(\forall xAB.(x \notin A \cup B \lor (x \in A \lor x \in B)))\land\\ &(\forall xAB.(\neg(x \in A \lor x \in B) \lor x \in A \cup B))\end{aligned} \quad (\text{def}_\cup)$$

7. The whole formula is negated and converted to negation normal form. In particular, the negation of the conjecture is

$$\exists ABC.A \cup (B \cup C) \neq (A \cup B) \cup C \quad (C_\neg)$$

   and the resulting formula is $((E \land A) \land \#) \land (\neg\# \lor C_\neg)$.
8. Reordering of the formula yields among others

$$\text{cong}_\cup \land (\text{cong}_\in \land (\text{sym}_= \land (\text{comm}_= \land \text{trans}_=))) \quad (E)$$

$$\begin{aligned}&(\forall xAB.((x \notin A \land x \notin B) \lor x \in A \cup B))\land\\ &(\forall xAB.(x \notin A \cup B \lor (x \in A \lor x \in B)))\end{aligned} \quad (\text{def}_\cup)$$

   and the resulting formula is $(\neg\# \lor C_\neg) \land (\# \land ((\text{def}_= \land \text{def}_\cup) \land E))$. Note that the equality axioms move to the end of the formula, so they are being processed last.
9. Skolemisation replaces existentially quantified variables by Skolem terms and removes existential quantifiers. For example, the Skolemised negated conjecture is

$$s_A \cup (s_B \cup s_C) \neq (s_A \cup s_B) \cup s_C \quad (C_\neg)$$

   where s_A, s_B, and s_C are nullary Skolem functions. We explain Skolemisation in more detail in Section 3.4.

The matrix is built from the resulting formula. For the clausal connection prover, this involves a transformation of the formula into clausal normal form. The *standard transformation* applies distributivity rules of the shape $A \land (B \lor C) \equiv (A \lor B) \land (A \lor C)$ to the formula until it is in conjunctive normal form. In the worst case, this transformation makes the formula grow exponentially. To avoid this, the *definitional transformation* introduces new symbols [Tse83, PG86, Ott10]. Similarly to Skolemisation, the introduced symbols should be consistent across different problems, which is achieved by using a normalised string representation of the clause literals as new symbol names. For the nonclausal connection prover, no clausification is required, as the formula can be directly transformed into the nonclausal matrix. For both clausal and nonclausal matrices, the polarity of literals is encoded by the sign of the integer representing the predicate symbol.

3.4 Consistent Skolemisation

First-order Skolemisation introduces new function symbols. For machine learning algorithms, it is beneficial to introduce names *consistently* across problems, meaning that Skolem terms originating from the same axiom in two different problems should be syntactically equivalent. Consistent Skolemisation methods have been studied in the context of the δ-rule in tableaux methods, e.g. [BHS93]. [GA99] pointed out that the Skolem terms introduced may lead to rather large formulae, which can be solved by structure sharing. However, in our setting, structure sharing across different problems is not possible, which makes it necessary to find different approaches. In previous work [KU15a], consistent Skolemisation was part of the clausification procedure. The implementation of nonclausal proof search motivated a more general consistent Skolemisation method. To recognise the same Skolem term across different problems, it is necessary to capture by the Skolem term only the part of the formula that defines the existential variable. For this, we propose a consistent Skolemisation method based on ϵ-terms.

Let us first introduce the setting for this section: Let Δ be a formula in negation normal form that is to be Skolemised. Without loss of generality, we assume Δ to be rectified, i.e. any two distinct quantifiers in Δ bind variables with different names. Furthermore, let the size of a formula F be the length of the string representation of F, and denote it by $|F|$.

Definition 3.7 (Skolemisation)**.** The *Skolemisation* of a formula Δ yields a formula equisatisfiable to Δ, not containing any existential quantifiers. To this end, Skolemisation replaces any subterm in Δ of the shape $\exists x.F$ by $F[t/x]$, where t is called the *Skolem term* for x.

There exist different methods to construct valid Skolem terms. We show such a method.

Example 3.8 (Inner Skolemisation)**.** *Inner Skolemisation* replaces any subterm in Δ of the shape $\exists x.F$ by $F[t/x]$, where the Skolem term t is $f(x_1, \dots, x_n)$ such that f is a fresh function symbol and $x_1, \dots, x_n$ are the universally bound variables free in F.

Inner Skolemisation as defined above is not a consistent Skolemisation method, as the freshness condition imposes all existentially quantified variables to be replaced by Skolem terms with different function symbols. We are now going to discuss consistent Skolemisation methods.

Replacing existentially quantified variables with ϵ-terms using the defining property of ϵ-terms

$$\exists x.P(x) \leftrightarrow P(\epsilon x.P(x))$$

we obtain a formula equivalent to the original one [HB39]. However, recursively replacing existential quantifiers by ϵ-terms can lead to an exponential size of the Skolemised formula. To show this, we are going to define two kinds of ϵ-Skolemisation and show that they both produce exponentially large output. In particular, the blowup is caused by the introduction of new Skolem names that contain other Skolem names.

Definition 3.9. Let F a subformula of Δ such that F is of the shape $\exists x.G$. A *naive ϵ-Skolemisation step* replaces F in Δ by $G[(\epsilon x.G)/x]$.

Naive ϵ-Skolemisation (NϵS) of a formula Δ repeatedly applies naive ϵ-Skolemisation steps until the formula does not contain any more existential quantifiers. To fix the order of Skolemisation steps, let outside-in NϵS replace subformulas only if they are not subformulas of an existential quantification, and inside-out NϵS replace subformulas only if they do not have subformulas containing existential quantifiers. We now give an example for which both outside-in and inside-out NϵS produce exponentially large Skolemised formulas.[1]

Example 3.10. Let ϕ_n be a formula recursively defined by $\phi_0 = P(x_0, x_0)$ and $\phi_{n+1} = \exists x_n.(P(x_{n+1}, x_{n+1}) \to \phi_n)$.

Lemma 3.11. *Inside-out NϵS of ϕ_n produces a formula exponential in n.*

Proof. Denote the inside-out NϵS of ϕ_n as $sk(\phi_n)$. Then $sk(\phi_0) = P(x_0, x_0)$ and $sk(\phi_{n+1}) = P(x_{n+1}, x_{n+1}) \to sk(\phi_n)[t_n/x_n]$, where $t_n = \epsilon x_n.(P(x_{n+1}, x_{n+1}) \to sk(\phi_n))$ is the Skolem term corresponding to x_n. For every n, $sk(\phi_n)$ contains at least two occurrences of x_n, and the Skolem term t_n corresponding to x_n is larger than $|sk(\phi_n)|$. Therefore, for every n, $|sk(\phi_{n+1})| > 2|sk(\phi_n)|$. □

Lemma 3.12. *Outside-in NϵS of ϕ_n produces a formula exponential in n.*

Proof. Let $\Delta = \forall x_m.\phi_m$ be the formula to be Skolemised. Then the Skolem terms corresponding to x_n can be given by

$$s_n = \begin{cases} x_m & \text{if } n = m \\ \epsilon x_n.P(s_{n+1}, s_{n+1}) \to \phi_n & \text{otherwise} \end{cases}$$

For any $n < m$, because every Skolem term s_n contains two occurrences of s_{n+1}, we have that $|s_n| > 2|s_{n+1}|$. As the base case s_m is greater than zero, we have that $|s_0|$ is exponential in m. □

The example above motivates a new consistent Skolemisation method that produces quadratic output and is also applicable to nonclausal search. For this, let us define some notation first: $\mathcal{FV}\text{ar}(F)$ denotes the free variables of F, and $\mathcal{FV}\text{ar}_\forall(F)$ and $\mathcal{FV}\text{ar}_\exists(F)$ denote the free variables in a subformula F of Δ that are universally and existentially bound in Δ, respectively. Δ^x is the subformula of Δ that binds the variable x, i.e. if Δ has a subformula $\exists x.F$, then Δ^x is $\exists x.F$. Furthermore, the transitive universal and existential free variables of F are denoted by

$$\mathcal{FV}\text{ar}^*_Q(F) = \mathcal{FV}\text{ar}_Q(F) \cup \bigcup_{x \in \mathcal{FV}\text{ar}_\exists(F)} \mathcal{FV}\text{ar}^*_Q(\Delta^x)$$

where $Q \in \{\forall, \exists\}$.

[1]Structure sharing would avoid the exponential blowup.

Definition 3.13 (Consistent ϵ-Skolemisation)**.** Let x be an existentially quantified variable in Δ. The *ϵ-defining formula* of Δ^x is $\max\{\Delta^y \mid y \in \mathcal{FV}\mathrm{ar}^*_\exists(\Delta^x)\}$. The maximum is computed with respect to the size of the formula, that is, it returns the largest formula in the set. The consistent ϵ-Skolem term for x is $\epsilon x.F$, where F is the ϵ-defining formula of Δ^x with all quantifiers for $\mathcal{FV}\mathrm{ar}^*_\forall(\Delta^x)$ and x removed. *Consistent ϵ-Skolemisation* of a formula Δ replaces any subformula of Δ of the shape $\exists x.F$ by $F[t/x]$, where t is the consistent ϵ-Skolem term for x.

Example 3.14. Let

$$\Delta = \forall x_1 \exists y_1.(P(x_1, y_1) \to (\forall x_2 \exists y_2.P(x_2, y_2)) \wedge (\forall x_3 \exists y_3.Q(x_3, y_3, y_1)))$$

The ϵ-defining formula of Δ^{y_1} and Δ^{y_3} is Δ^{y_1}, and of Δ^{y_2}, it is Δ^{y_2}. The consistent ϵ-Skolem term for y_n is s_n, where

$$\begin{aligned}
s_1 &= \epsilon y_1.P(x_1, y_1) \to (\forall x_2 \exists y_2.P(x_2, y_2)) \wedge (\forall x_3 \exists y_3.Q(x_3, y_3, y_1)) && \mathcal{FV}\mathrm{ar}(s_1) = \{x_1\} \\
s_2 &= \epsilon y_2.P(x_2, y_2) && \mathcal{FV}\mathrm{ar}(s_2) = \{x_2\} \\
s_3 &= \epsilon y_3.\exists y_1.P(x_1, y_1) \to (\forall x_2 \exists y_2.P(x_2, y_2)) \wedge Q(x_3, y_3, y_1) && \mathcal{FV}\mathrm{ar}(s_3) = \{x_1, x_3\}
\end{aligned}$$

Theorem 3.15. *Consistent ϵ-Skolemisation of a formula Δ yields a formula equivalent to Δ.*

Proof. Let us consider an arbitrary existentially quantified variable x in Δ. Let D be the ϵ-defining formula of Δ^x. We can move all universal quantifiers corresponding to $\mathcal{FV}\mathrm{ar}^*_\forall(\Delta^x)$ in front of D, yielding $\forall \vec{y}_1.D_1$. Note that $\mathcal{FV}\mathrm{ar}(D_1) = \mathcal{FV}\mathrm{ar}^*_\forall(\Delta^x)$ and $\vec{y}_1 = \mathcal{FV}\mathrm{ar}^*_\forall(\Delta^x) \setminus \mathcal{FV}\mathrm{ar}_\forall(D)$. Furthermore, we can move the existential quantifier for x in front of D_1, resulting in $\forall \vec{y}_1 \exists x.D_2$. Now, we can use the defining property of ϵ-terms to obtain $\forall \vec{y}_1.D_2[t/x]$, where $t = \epsilon x.D_2$. Finally, we can move back the quantifiers from $\vec{y}_1$ to their original places to yield D_3. (D_3 could have been equally obtained by removing the quantifier $\exists x$ from D and replacing x by $\epsilon x.D_2$.) As all free variables of every such ϵ-term t are universally quantified, we can execute the operations above to replace in arbitrary order every existentially quantified variable with its corresponding ϵ-term. As all the operations preserve equivalence of the formula, this yields a formula equivalent to Δ, and we can see that it is exactly the outcome of consistent ϵ-Skolemisation. □

Lemma 3.16. *Consistent ϵ-Skolemisation of Δ yields a formula of size smaller than $|\Delta|^2$.*

Proof. The maximal size of a consistent ϵ-Skolem term is $|\Delta|$, and as there are less than $|\Delta|$ occurrences of existentially quantified variables in Δ, replacing them yields a formula of size smaller than $|\Delta|^2$. □

To obtain a first-order formula from consistent ϵ-Skolemisation, we replace every consistent ϵ-Skolem term of the shape $\epsilon x.F$ with $f_{[\epsilon x.F]}(\vec{y})$, where $\vec{y}$ is $\mathcal{FV}\mathrm{ar}(\epsilon x.F)$ (and thus also $\mathcal{FV}\mathrm{ar}^*_\forall(\Delta^x)$). Here, $[t]$ denotes a normalisation of t such that for any t_1 and t_2, if t_1 is α-equivalent to t_2, then $[t_1] = [t_2]$. We use this normalisation to recognise equivalent Skolem terms even if their variables are differently named. This yields a formula equisatisfiable to Δ.

The following corollary states under which conditions variables that are existential bound at different locations will be consistently Skolemised to the same function symbol. Note that this result holds across different formulas and problems.

Corollary 3.17 (Consistency)**.** *Two existential variables are mapped to the same first-order function symbol iff their corresponding ϵ-Skolem terms are α-equivalent.*

Proof. Follows from Definition 3.13. □

3.5 Connection Search

We explain how the prover efficiently searches for connections that correspond to extension steps. For this, let us introduce the concept of a *contrapositive*. We will first formulate its general nonclausal definition, then show the specific case for clausal matrices.

Definition 3.18 (Contrapositive)**.** Given a matrix M with $C \in M$ and $L \in^+ C$, the formula $\overline{L} \rightarrow C_\beta$ is a *contrapositive* of M, where C_β is the β-clause of C with respect to L.

Definition 3.19 (Clausal Contrapositive)**.** Given a clausal matrix M with $C \in M$ and $L \in C$, the formula $\overline{L} \rightarrow C \setminus \{L\}$ is a *contrapositive* of M.

To find a connection with a literal L, it suffices to find a contrapositive of M with an antecedent L' such that L and L' can be unified. The consequent of the contrapositive can then be used to generate extension clauses.

Example 3.20. Consider the matrix M' from Example 3.5 on page 31. A contrapositive of M' is $Q \rightarrow (\neg P(x) \vee P(sx))$. This contrapositive was used to find the connection $\{Q, \neg Q\}$ in Figure 3.4 and to generate the corresponding extension clause $\{\neg P(x'), P(sx')\}$ in Figure 3.5.

The original versions of leanCoP and nanoCoP rely on Prolog's internal literal indexing to keep a contrapositive database. We considered storing contrapositives in first-order term indexing structures [RSV01]. However, the overall effect on performance of storing contrapositives in a discrimination tree [Gre86] on the considered datasets is minor, as unification with array substitutions (see below) is relatively fast. In our implementations, we store all contrapositives in a hash table indexed by the polarity and the predicate symbol of the antecedents. To find connections with a literal L, we perform two steps: First, we retrieve from the hash table all contrapositives whose antecedents have the same polarity and predicate symbol as L. Second, we return those contrapositives obtained in the first step whose antecedents can be unified with L.

Unification is one of the most time consuming parts of proof search. Therefore it is crucial to represent data, including substitutions, in a way that allows efficient unification. The simplest approach to represent substitutions is to use association lists from variables to terms. This is done e.g. in the HOL Light implementation of MESON. However, as variable lookup is linear in the number of bound variables, this approach does not scale

well. An improvement over this is to use tree-based maps, used for example by Metis. Both solutions however incur a significant overhead in tableaux proof search, where a single large substitution is needed. In functional languages with efficient support for arrays (e.g. the ML language family, used in many proof systems), it is more efficient to store the substitution in a single global mutable array. As variables can be represented by positive integers, the nth array element contains the term bound to the variable n. By keeping a stack of variables bound in each prover state, it is also possible to backtrack efficiently: variables removed from the top of the stack are removed from the global array. This way, backtracking can be done as if the substitution was contained in a purely functional data structure, however allowing for more efficient unification.

3.6 Proof Search

Proof search in connection tableaux calculi is analytic, i.e. the proof tree is constructed bottom-up. As the proof search is not confluent, i.e. making a wrong choice can lead to a dead end, backtracking is necessary for completeness. The proof tree is constructed with a depth-first strategy, which results in an incomplete proof search. To remedy this, iterative deepening is used, where the maximal path length is increased in every iteration.

The principal implementations of connection tableaux calculi, leanCoP and nanoCoP, use a number of optimisation techniques, such as regularity, lemmas, and restricted backtracking [Ott10]. When backtracking is restricted, as soon as the proof search finds some proof tree to close a branch, no other potential proof trees for that branch are considered any more. While restricted backtracking loses completeness, it significantly increases the number of problems solved for various first-order problem classes.

Prolog allows for a very elegant and succinct implementation of proof search. First attempts to directly integrate machine learning into Prolog leanCoP have suffered from slow speed [UVv11]. Later, [KU15a, KUV15] showed that implementations of leanCoP in a functional programming language allow for fast machine learning as well as for efficient proof reconstruction in interactive theorem provers. However, implementing proof search with restricted backtracking in a functional language is not straightforward.

In this section, we discuss several implementations of a clausal prover loop that permits restricted backtracking: The simplified version of leanCoP shown in Subsection 3.6.1 is the smallest, but also the slowest implementation. Care is taken that all subsequent implementations perform the proof search in precisely the same order as the original Prolog implementation. We then introduce a purely functional implementation in Subsection 3.6.2 using lazy lists or streams. This version slightly increases code size compared to the Prolog version, but greatly improves performance, as shown in the evaluation in Section 3.9. We also discuss an approach based on continuations, still purely functional, but more complicated than the stream version. In exchange, this version has slightly better performance than the stream one, likely due to not having to allocate memory for (stream) constructors. The fastest, but also most complicated implementation considered in this thesis uses an explicit stack and exceptions for backtracking. However, as it proves just as many problems as the continuation-based version, we will only briefly

Listing 3.1: Clausal proof search in Prolog.

```
1  prove([],_,_).
2  prove([Lit|Cla],Path,PathLim) :-
3      (-NegLit=Lit;-Lit=NegLit) ->
4         ( member(NegL,Path), unify_with_occurs_check(NegL,NegLit)
5           ;
6           lit(NegLit,Cla1),
7           ( length(Path,K), K<PathLim -> true ; fail ),
8           prove(Cla1,[Lit|Path],PathLim)
9         ),
10        prove(Cla,Path,PathLim).
```

discuss it.

3.6.1 Prolog

A simplified version of the original leanCoP in Prolog is given in Listing 3.1. We explain and relate it to the clausal connection calculus introduced in Section 3.2.

The main predicate `prove(`C`, `$Path$`, PathLim)` succeeds iff there exists a closed proof tree for $\langle C, M, Path\rangle$ with a maximal path length of `PathLim`. For this, `prove` attempts to close the proof tree for the first literal `Lit` of C in lines 4–9, and if successful, it continues with the remaining clause `Cla` of C in line 10.

Let us detail the proof search for the current literal `Lit`: Line 4 corresponds to the *reduction* rule: The branch is closed if the negation of `Lit` can be unified with a literal on the $Path$. Lines 6–8 correspond to the *extension* rule: The contrapositive database as explained in Section 3.5 is implemented by the predicate `lit(`L`, `C`)`, which succeeds iff the matrix contains some clause that can be unified with $\{L\} \cup C$. This is used to obtain some contrapositive `Cla1` for the negation of `Lit`. If the path does not exceed the length limit (line 7), new branches are opened for `Cla1` in line 8.

Backtracking is handled by the Prolog semantics: For example, if choosing the first matching contrapositive for `Lit` leads to the proof search getting stuck, the next contrapositive will be tried by Prolog.

3.6.2 Lazy Lists and Streams

Proof search in a functional language can be elegantly implemented as a function from a branch to a *lazy list* of proofs, where a lazy list is an arbitrarily long list built on demand. However, as the proof search considers every list element at most once, the memoization done for lazy lists creates an unnecessary overhead. For that reason, *streams* can be used instead of lazy lists, where a stream is a special case of a lazy list that restricts list elements to be traversed at most once. As our application uses a common interface for lazy lists and streams, we solely present the lazy list version here.

Listing 3.2: Lazy list implementation of clausal proof search.

```
1  prove [] path lim sub = [sub]
2  prove (lit : cla) path lim sub =
3    let
4      reductions = mapMaybe (unify sub (negate lit)) path
5      extensions = unifyDB sub lit & concatMap
6        (\ (sub1, cla1) ->
7          if lim <= 0 then []
8          else prove cla1 (lit : path) (lim - 1) sub1)
9    in concatMap (prove cla path lim) (reductions ++ extensions)
```

Listing 3.2 shows a functional leanCoP implementation using lazy lists. Let us first introduce the semantics of the used constructs:

- `x & f` denotes `f x`.
- `\ x -> y` stands for a lambda term $\lambda x.y$.
- `unify sub lit1 lit2` unifies two literals `lit1` and `lit2` under a substitution `sub`, returning a new substitution if successful.
- `unifyDB sub lit` finds all contrapositives in the database which could match the literal `lit` under the substitution `sub`. It returns a list of substitution-contrapositive pairs. It corresponds to the `lit` predicate in the Prolog version.
- `mapMaybe f l` returns the results of `f` for the elements of `l` on which `f` succeeded.
- `concatMap f l` maps `f` over all elements of `l` and concatenates the resulting list of lists to form a flat list.
- `x ++ y` is the concatenation of two lists `x` and `y`.

The main function `prove` C $Path$ `lim` σ returns a list of substitutions $[\sigma_1, \dots, \sigma_n]$, where every substitution σ_i corresponds to a closed proof tree for $\langle C, M, Path\rangle$ with a maximal path length smaller than `lim`, where the global initial substitution is σ and the final substitution is σ_i.[2] Similarly to the Prolog version, `prove` attempts to close the proof tree for the first literal `lit` of C in lines 4–8, and the resulting substitutions are used to close the proof trees for the remaining clause `cla` of C in line 9. Line 4 corresponds to the reduction rule, and lines 5–8 correspond to the extension rule. As we use lazy lists / streams, a substitution σ_i is only calculated if proof search failed for all σ_j with $j < i$.

3.6.3 Continuations

Continuation passing style (CPS) allows the implementation of algorithms with complicated control flow in functional languages [Plo75]. Listing 3.3 shows a leanCoP implementation using CPS. The main function `prove` C $Path$ `lim` σ `alt rem` searches for a closed proof tree for $\langle C, M, Path\rangle$ with a maximal path length smaller than `lim` under the substitution σ. If `prove` finds such a proof tree, it calls the `rem` continuation to treat remaining proof obligations (line 1). Otherwise, `prove` calls the `alt` continuation

[2] In this simplified implementation, the actual proof tree is not recorded, in contrast to our actual implementation. The same holds for the Prolog version.

Listing 3.3: CPS implementation of clausal proof search.

```
prove [] path lim sub alt rem = rem sub alt
prove (lit : cla) path lim sub alt rem = reduce path where
  reduce (plit : path) =
    let alt1 () = reduce path
    in case unify sub (negate lit) plit of
      Nothing -> alt1 ()
      Just sub1 -> prove cla path lim sub1 alt1 rem
  reduce [] = extend (unifyDB sub (negate lit))

  extend ((sub1, cla1) : contras) =
    let alt1 () = extend contras
    in if lim <= 0 then alt1 ()
       else
         let rem1 sub alt = prove cla path lim sub alt rem
         in prove cla1 (lit : path) (lim - 1) sub1 alt1 rem1
  extend [] = alt ()
```

to backtrack to an alternative (line 16). The `reduce` function in lines 3–7 corresponds to the reduction rule, and the `extend` function in lines 10–15 corresponds to the extension rule. If no more reductions can be performed, extensions are tried (line 8), and if no more extensions can be performed, we backtrack (line 16). Both `reduce` and `extend` define a continuation `alt1` (line 4 and 11) to provide a way to backtrack to the current state, and pass it to `prove` (line 7 and 15). `extend` additionally defines a continuation `rem1` (line 14), which serves to continue proof search for the clause `cla` once a proof for the contrapositive clause `cla1` was found (line 15).

3.6.4 Stacks

We considered an implementation based on stacks. There, the main prove function has the same arguments as the `prove` function of the stream-based implementation, plus a stack. This stack contains tuples with information about clauses that still have to be processed, together with the depth at which the clauses have been put onto the stack. Once the current clause has been completely refuted, the next tuple is popped from the stack and the clause in the tuple is processed.

3.7 Clause Processing Order

Proof search processes clauses and matrices in a certain order $<$, such that for any elements a, b of the same clause or matrix, a is processed before b iff $a < b$. The order $<$ is usually derived from the structure of the formula obtained in Section 3.3. For nonclausal proof search, we have evaluated different ways to order β-clauses: The original nanoCoP processes the β-clause of a clause C with respect to a literal L (see Definition 3.4) using

the order $<_L$, where $a <_L b$ iff a contains L or $a < b$. The reconstruction of proofs created in this order requires some postprocessing; this motivated our usage of the regular $<$ for β-clauses.

Example 3.21. Let M contain a clause

$$C = \left[\begin{matrix} M_1 \\ \left[\begin{matrix} C_1 & \begin{bmatrix} L \\ M_2 \\ M_3 \end{bmatrix} \end{matrix} \right] \end{matrix} \right]$$

Then the β-clauses of C with respect to L ordered by $<$ and $<_L$ are $\beta_<$ and $\beta_{<_L}$:

$$\beta_< = \left[\begin{matrix} M_1 \\ \left[\begin{matrix} C_1 & \begin{bmatrix} M_2 \\ M_3 \end{bmatrix} \end{matrix} \right] \end{matrix} \right] \qquad \beta_{<_L} = \left[\begin{matrix} \left[\begin{matrix} \begin{bmatrix} M_2 \\ M_3 \end{bmatrix} & C_1 \end{matrix} \right] \\ M_1 \end{matrix} \right]$$

One can see that when using $\beta_{<_L}$, the neighbours M_2 and M_3 of L are processed first, unlike when using $\beta_<$.

3.8 Extension Clause Anomaly

The implementation of nonclausal proof search in functional style exposed in the original Prolog implementation an anomaly related to extension clauses. We are going to describe this anomaly, how we found it, and how we avoided it using principles found in functional programming.

Example 3.22 (Extension Clause Anomaly)**.** Consider the extension rule shown in Figure 3.2. Let

$$M = \left[\begin{matrix} \left[\begin{matrix} \left[\begin{matrix} \begin{bmatrix} L' \\ [C_c] \\ M_b \end{bmatrix} & C_b \end{matrix} \right] \\ M_a \end{matrix} \right] & C_a \end{matrix} \right]$$

Assume that M_b contains more than one clause. Furthermore, let C and $Path$ be arbitrary. Then the set of possible values for C_3 in the extension rule are copies of

$$\left\{ \begin{bmatrix} [C_c] \\ M_b \end{bmatrix}, \left[\begin{matrix} \left[\begin{matrix} \begin{bmatrix} [C_c] \\ M_b \end{bmatrix} & C_b \end{matrix} \right] \\ M_a \end{matrix} \right] \right\}$$

However, the Prolog implementation admits yet another value for C_3, namely a copy of C_c.

We found that behaviour as in Example 3.22 occurs under the following circumstances: A clause $C_{L'}$ with $L' \in C_{L'}$ (in Example 3.22, $C_{L'} = \{L', \{C_c\}, M_b\}$) contains a matrix

M' consisting of a single clause (in Example 3.22, $M' = \{C_c\}$), and M' is the smallest clause in $C_{L'} \setminus \{L'\}$ with respect to the clause processing order (introduced in Section 3.7).

This behaviour is due to the way in which contrapositives are stored and extension clauses are generated in the Prolog implementation. While we originally implemented extension clause generation like in the original Prolog implementation (thus inheriting the anomaly), we later used algebraic data types to enforce stronger constraints. As a side effect, we avoided the anomaly described above, which we discovered by comparing proof search traces between the two implementations. Consequently, we fixed the anomaly in the Prolog implementation.

While we did not encounter an example showing that the original version of nanoCoP is unsound, the usage of functional programming principles was helpful in discovering an anomaly that might otherwise have remained undiscovered.

3.9 Evaluation

We evaluate our work on several first-order problem datasets, with statistics given in Table 3.1:

- TPTP [Sut09b] is a large benchmark for automated theorem provers. It is used in CASC [Sut16b]. The contained problems are based on different logics and come from various domains. In our evaluation we use the nonclausal first-order problems of TPTP 6.3.0.
- MPTP2078 [AHK+14] contains 2078 problems exported from the Mizar Mathematical Library. This dataset is particularly suited for symbolic machine learning since symbols are shared between problems. It comes in the two flavours “bushy” and “chainy”: In the “chainy” dataset, every problem contains all facts stated before the problem, whereas in the “bushy” dataset, every problem contains only the Mizar premises required to prove the problem.
- Miz40 contains the problems from the Mizar library for which at least one ATP proof has been found using one of the 14 combinations of provers and premise selection methods considered in [KU15d]. The problems are translated to untyped first-order logic using the MPTP infrastructure [Urb04]. Symbol names are also used consistently in this dataset, and the problems are minimised using ATP-based pseudo-minimisation, i.e., re-running the ATP only with the set of proof-needed axioms until this set no longer becomes smaller. This typically leads to even better axiom pruning and ATP-easier problems than in the Mizar-based pruning used for the “bushy” version above.
- HOL Light: We translate theorems proven in HOL Light to first-order logic, following a similar procedure as [KU14]. We export top-level theorems (“top”) as well as theorems proven by the MESON tactic (“msn”).[3] We consider the theorems proven in the core of HOL Light (“HL”) as well as those proven by the Flyspeck

[3] As part of exporting theorems solved by MESON, we perform some of the original MESON preprocessing, such as propositional simplification, Skolemisation, and currying, see Section 6.2. This preprocessing may solve the problem, in which case we do not export the problem at hand.

Table 3.1: Evaluation datasets and number of contained first-order problems.

Dataset	TPTP	MPTP	Miz40	HL-top	HL-msn	FS-top	FS-msn
Problems	7492	2078	32524	2498	1108	27111	39979

project (“FS”), which finished in 2014 a formal proof of the Kepler conjecture [HAB+17].

We use a 48-core server with AMD Opteron 6174 2.2GHz CPUs, 320 GB RAM, and 0.5 MB L2 cache per CPU. Each problem is always assigned one CPU. We run all provers with a timeout of 10 seconds per problem.

We evaluate several prover configurations in Table 3.2. As state of the art, we use the ATPs Vampire 4.0 [KV13] and E 2.0 [Sch13], which performed best in the first-order category of CASC-J8 [Sut16a]. Vampire and E are written in C++ and C, respectively, implement the superposition calculus, and perform premise selection with SInE [HV11]. Furthermore, Vampire integrates several SAT solvers [BDKV14], and E automatically determines proof search settings for a given problem. We ran E with `--auto --auto-schedule` and Vampire with `--mode casc`. In addition, we evaluated the ATP Metis [Hur03]: It implements the ordered paramodulation calculus (having inference rules for equality just like the superposition calculus), but is considerably smaller than Vampire and E and is implemented in a functional language, making it more comparable to our work.

We implemented functional-style versions of leanCoP 2.1 and nanoCoP 1.0 in the functional programming language OCaml.[4] Our implementations use the techniques introduced such as hash-based indexing and array-based substitutions (Section 3.5), efficient control flow (Section 3.6), alternative clause processing orders (Section 3.7), and consistent Skolemisation (Section 3.4). Our functional OCaml implementations are fleanCoP and fnanoCoP, whereas the original Prolog versions are pleanCoP and pnanoCoP. The Prolog versions were run with ECLiPSe 5.10. A prover configuration containing “+x” or “−x” means that feature x was enabled or disabled, respectively. “cut” denotes restricted backtracking, “conj” stands for conjecture-directed search, and $\beta_{<_L}$ refers to the default β-clause ordering shown in Section 3.7. leanCoP was evaluated without definitional clausification, see Section 3.3. The OCaml implementations use streams to control backtracking (see Subsection 3.6.2) and arrays as substitutions. As strategy scheduling is not a focus of this work, we evaluate our provers with disabled strategy scheduling.

The results are shown in Table 3.2: The OCaml versions outperform the Prolog versions in almost all cases. The most impressive result is achieved by fleanCoP+cut+conj on the chainy dataset: The OCaml version proves 58.8% more problems than its Prolog counterpart, thus even passing E. Furthermore, on four out of six datasets, our strongest configuration proves more problems than Metis. $\beta_{<_L}$ seems to have an effect mostly when cut is enabled. However, the result depends greatly on the dataset: On the chainy

[4]The source code is available at http://cl-informatik.uibk.ac.at/users/mfaerber/cop.html.

Table 3.2: Comparison of provers without machine learning.

Prover	TPTP	Bushy	Chainy	Miz40	FS-top	FS-msn
Vampire	4404	1253	656	30341	6358	39760
E	3664	1167	287	26003	7382	39740
Metis	1376	500	75	18519	3537	38625
fleanCoP+cut+conj	1859	670	289	12204	3980	35738
fleanCoP+cut−conj	1782	598	244	11796	3520	30668
fleanCoP−cut+conj	1617	499	192	7826	3849	35204
fleanCoP−cut−conj	1534	514	164	11115	3492	36334
pleanCoP+cut+conj	1673	606	182	11243	3664	35234
pleanCoP+cut−conj	1621	548	153	11227	3305	30416
pleanCoP−cut+conj	1428	453	143	7287	3671	34437
pleanCoP−cut−conj	1374	460	123	10442	3415	35499
fnanoCoP+cut	1724	511	192	12332	3178	30327
fnanoCoP+cut−$\beta_{<L}$	1776	547	233	11197	3182	30216
fnanoCoP−cut	1567	542	151	13316	1993	37938
fnanoCoP−cut−$\beta_{<L}$	1559	541	152	13173	1991	37923
pnanoCoP+cut	1585	480	112	11921	2970	30272
pnanoCoP−cut	1485	510	126	12943	1986	38015

dataset, disabling $\beta_{<L}$ solves 21.3% more problems, but on the Miz40 dataset, it solves 8.8% less.

We evaluate different proof search implementation styles in Tables 3.3 and 3.4. Here, inferences denote the number of successful unifications performed by some prover on all problems within 10 seconds timeout. This metric is not available for the Prolog versions, as these do not print the number of inferences performed when prematurely terminated.

To measure the impact of the substitution structure, we evaluated the best-performing implementation, i.e. the stack-based one, using a list-based substitution instead of an array-based substitution, see Table 3.3. This decreased the number of inferences by 50%, showing that the performance of the substitution structure is crucial for fast proof search.

Unless noted otherwise, we will use the stream-based implementation with array-based substitution in the remainder of this thesis.

3.10 Reproducible Experiments

We have taken special care to design experiments to be reproducible. We are going to show how we use GNU `make` [SMS16] to define experiments.[5]

Example 3.23. To create Table 3.2, we have created `Makefile`s that perform the following tasks:

[5]The source code is available at http://cl-informatik.uibk.ac.at/users/mfaerber/cop.html.

Table 3.3: Impact of implementation on efficiency of clausal proof search on the bushy MPTP2078 dataset with 10s timeout, restricted backtracking (+cut), no definitional CNF, and conjecture-directed search (+conj).

Implementation	Solved	Inferences
Prolog	606	-
Lazy list	639	878199349
Stack (list substitution)	648	1253862954
Stream	670	1702827032
Continuation	681	2200272406
Stack	681	2490100879

Table 3.4: Impact of implementation on efficiency of nonclausal proof search on the bushy MPTP2078 dataset with 10s timeout and restricted backtracking (+cut).

Implementation	Solved	Inferences
Prolog	480	-
Lazy list	504	374849495
Streams	511	495368962

- Download[6] ATPs and build them.
- Download problem sets.
- Run ATPs on all problems for every problem set, in parallel.
- Accumulate results and generate a table.

For our own ATPs, we use the version control system *Git* [CS14]. This allows us to refer to particular development versions of our ATPs when defining experiments in `Makefiles`. When running an experiment, `make` automatically builds the right ATP version and executes it with the specified parameters.

Example 3.24. Let us assume we made some changes in our implementation of the prover *nanoCoP* and we want to evaluate its performance on a set of problems in the `bushy` directory. For this, we first tag our development version of nanoCoP with a unique identifier, say `git tag cop-170829`. We then proceed to define a new experiment as in Listing 3.4:

- Line 1 collects the problems in the `bushy` directory.
- Lines 3 and 4 reduce the whole set of problems to its individual problems.
- Line 5 specifies the dependency of the experiment on the prover version.
- Lines 6 and 7 define how any single problem should be evaluated.

Note that this depends on previously defined rules given in Listing 3.5 that tell `make` how to obtain (lines 1 and 2) and build (lines 4 and 5) any version of nanoCoP. To perform

[6] The `Makefiles` contain instructions to verify the checksum of downloaded files, aborting the build in case of mismatch.

Listing 3.4: Experiment rules.

```
BUSHY = $(shell find bushy/ -type f | sort -R)

o/bushy/10s/nanocop-170829: $(BUSHY:bushy/%=\
o/bushy/10s/nanocop-170829/%)
o/bushy/10s/nanocop-170829/%: cop-170829/nanocop.native bushy/%
	@mkdir -p `dirname $@`
	-timeout 10 $^ > $@
```

Listing 3.5: Rules to obtain a prover version and to build it.

```
cop-%: ../.git/refs/tags/cop-%
	git clone .. $@ && cd $@ && git checkout $@

%/nanocop.native: %
	make -C $< nanocop.native
```

the experiment, we run `make o/bushy/10s/nanocop-170829 -j$(CORES)`.

Using `Makefile`s to encode experiments has the following merits:

- The `Makefile`s serve as natural experiment logs, documenting all experiments performed.
- It ensures the exact reproducibility of experiments performed during the development of an ATP.
- It depends only on established tools such as GNU `make` and `git`.
- It trivially allows parallelisation and resumption of experiments.

While it took some time to develop this system, our reproducible experiment style has been very useful throughout our research.

3.11 Conclusion

We translated connection provers to functional programming languages, exploring possibilities to increase the speed of proof search while keeping the implementation as simple as possible. It turned out that unification and control flow can be efficiently implemented such that the prover performance increases, while the main proof loop remains small and easy to modify. We showed that the number of solved problems can be increased by up to 58.8%, on one dataset beating even E in automatic mode. However, we showed that on most datasets, current implementations of nonclausal proof search prove fewer problems than their clausal counterparts, despite having very favourable theoretical properties. It appears that this is due to the lower number of inferences, as can be seen in Tables 3.3 and 3.4. Thus it remains as future work to improve the performance of nonclausal implementations.

Chapter 4

Internal Guidance

4.1 Introduction

Internal guidance methods learn making decisions arising during proof search. Such methods do not influence decisions before proof search, such as which preprocessing options or which global strategies are used. The guided decisions have a large impact on the time required to find proofs, and in case of incomplete search strategies they determine whether a proof will be found at all. An example of internal guidance are ranking heuristics that learn from previous proofs.

Internal guidance methods aim to estimate the utility of actions according to the system's knowledge of the world and previous experiences. An experience can be characterised by a prover state $\vec{f}$ and an action l that was performed in that state, together with the information whether the action l was useful. Similarly to premise selection (see Chapter 2), this makes it possible to treat the choice of actions as a classification problem.

Example 4.1. Assume that the state of the theorem prover is modelled by the set of constants appearing in the previously processed propositions or in the conjecture. Let our conjecture be $\forall xy. x + y = y + x$ and let our premises include

$$\forall P. [P(0) \to (\forall x. P(x) \to P(s(x))) \to \forall x. P(x)] \qquad (\texttt{num_INDUCTION})$$

$$\forall x. x + 0 = x \qquad (\texttt{ADD_0})$$

Assume that we first process `num_INDUCTION`. The resulting prover state is then characterised by $\vec{f} = \{+, s, 0\}$. Furthermore, assume that we then process `ADD_0` and continue proof search. If we eventually find a proof, we analyse for every action taken whether it contributed to the final proof. For example, if processing `ADD_0` in the state characterised by $\vec{f}$ contributed to the proof, we register `ADD_0` with $\vec{f}$ as a *positive example*, otherwise as a *negative example*.

Intuitively, it seems desirable to perform an action in situations similar to those where the action was useful, and avoid performing an action in situations similar to those where the action was useless.

We propose a new internal guidance method for automated theorem provers using

(variations of) the given-clause algorithm, such as Vampire or E.[1] Our method influences the choice of given clauses, using positive and negative examples from previous proofs. To this end, we present an efficient scheme for Naive Bayes classification by generalising label occurrences to types with monoid structure, see Section 4.2. This makes it possible to extend existing fast classifiers, which consider only positive examples, with negative ones.

The integration of our internal guidance method into an ATP using the given-clause algorithm involves two tasks: The recording of training data (Section 4.4), and the ranking of unprocessed clauses (Section 4.6), which influences the choice of the given clause. To reduce the amount of data an ATP has to load for internal guidance, we process training data (Section 4.5) and transform it into classification data outside of the ATP. Furthermore, we show two methods to find good parameters for our method in Section 4.7. In Section 4.8, we show how to implement the method in the higher-order logic prover Satallax, where we modify the delay with which propositions are processed. In Section 4.9, we evaluate the performance of our method.

4.2 Naive Bayes with Monoid Occurrences

We found the combination of positive and negative training examples to be crucial for internal guidance. However, the classification methods presented in Chapter 2 only consider positive examples. For this reason, we generalise samples, i.e. the input of classifiers, to contain explicit information about their *occurrence*: Let us recall that the samples in Chapter 2 were label-feature pairs $(l, \vec{f})$. We now generalise samples to be triples $(l, \vec{f}, o) \in S$, where o encodes how often the combination of l and $\vec{f}$ occurred positively and negatively.

It is possible to extend the type of occurrences to not only encode the number of positive and negative occurrences, but also neutral ones etc. This leads us to the following question: Which properties should types of occurrences in general abide by? Let us consider some requirements for types of occurrences:

1. When we learn about the same label-feature pair with different occurrences o_1 and o_2, we want to be able to combine their occurrences, say $o_1 + o_2$.
2. It should not play a role in which order we learn about a label-feature pair. That is, $o_1 + o_2$ should be the same as $o_2 + o_1$.
3. There should be a neutral occurrence 0 that we can assume all label-feature pairs not yet encountered to have.

It turns out that these requirements for types of occurrences are well met by *commutative monoids*.

Definition 4.2 (Monoid)**.** A pair $(M, +)$ is a *monoid* if there exists a neutral element $0 \in M$ such that for all $x, y, z \in M$, $(x + y) + z = x + (y + z)$ and $x + 0 = 0 + x = x$. If

[1] Technically, our reference prover Satallax does not implement a given-clause algorithm, as Satallax treats terms instead of clauses, and it interleaves the choice of unprocessed terms with other commands. However, for the sake of internal guidance, we can consider Satallax to implement a variant of the given-clause algorithm. We describe the differences in more detail in Section 4.8.

furthermore $x + y = y + x$, then the monoid is *commutative*.

Example 4.3. Let us consider a classifier for positive and negative examples. For this, we can use the monoid $(\mathbb{N} \times \mathbb{N}, +_2)$, where the first and second elements of the pair represent the number of positive and negative occurrences, respectively. The $+_2$ operation is pairwise addition, and the neutral element is $0_2 = (0, 0)$. A triple learnt by this classifier could be $(l, \vec{f}, (1, 2))$, meaning that l occurs with $\vec{f}$ once positively and twice negatively.

We generalised the Naive Bayes classifier introduced in Section 2.3 to use the monoid given in Example 4.3 for occurrences. To this end, we show an adaption of the classifier's underlying probabilities $P(l_i)$ and $P(f_j \mid l_i)$ to positive and negative examples.

We first introduce some preliminaries. To obtain values between 0 and 1, we use the standard logistic sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The positive and negative occurrences are[2]

$$(p_i, n_i) = \sum \left\{ o \mid (l_i, \vec{f}, o) \in S \right\}$$
$$(p_{i,j}, n_{i,j}) = \sum \left\{ o \mid (l_i, \vec{f}, o) \in S, f_j \in \vec{f} \right\}$$

Finally, the probabilities adapted to positive and negative examples are:

$$P(l_i) = \sigma\left(\frac{p_i - n_i}{p_i + n_i}\right)$$
$$P(f_j \mid l_i) = \sigma\left(\frac{p_{i,j} - n_{i,j}}{p_i + n_i}\right)$$

4.3 Features

To characterise the prover state, we tried different kinds of features:

- Symbols of processed clauses: We collect the symbols of all processed clauses at the time a clause is inserted into the unprocessed clauses and call these symbols the features of the clause. However, this experimentally turned out to be a bad choice, because the set of features for each clause grows rapidly.
- Axioms of the problem: We associate every clause processed in a proof search with all the axioms of the problem. In contrast to the method above, this associates the same features to all propositions processed during the proof search for a problem, and is thus more a characterisation of the problem (similar to TPTP characteristics [SB10]) than of the prover state.

Calculating the influence of these features without them actually influencing the ranking makes Satallax prove fewer problems within a fixed timeout, due to the additional calculation time. Furthermore, the positive impact of the features on the proof search

[2]The occurrence o is a pair. Therefore, the sum Σ over occurrences uses $+_2$ and also returns a pair.

does not compensate for the slower performance. Therefore, we currently do not use features at all and associate the empty set of features with all labels, i.e. $\mathcal{F}(c) = \emptyset$. However, it turns out that even without features, learning from previous proofs can be quite effective, as shown in Section 4.9.

4.4 Training Data Recording

Recording training data can be done in different fashions:

- **In situ**: Information about clause usage is recorded every time an unprocessed clause gets processed. This method allows for more expressive prover state characterisation. On the other hand we found it to decrease the proof success rate, as the recording of proof data slows down proof search.
- **Post mortem**: Only when a proof was found, information about clause usage is reconstructed. As this method does not place any overhead on the proof search, we resorted to post-mortem recording, which is still sufficiently expressive for our purposes.

For every proof we consider the clauses processed during proof search: A processed clause l occurring in a state characterised by $\vec{f}$ yields a sample $(l, \vec{f}, o)$, where $o = (1, 0)$ if l contributed to the final proof and $o = (0, 1)$ if not. We use the monoid $(\mathbb{N} \times \mathbb{N}, +_2)$ with the neutral element $(0, 0)$ introduced in Section 4.2 to store positive and negative examples. We call the set of samples for a single proof a *training datum*. We ignore unprocessed clauses, as we cannot easily estimate whether they might have contributed to a proof. The samples of all training data are the (multiset) union of the samples of the individual training data.

4.5 Clause Processing

In our experiments, we frequently encounter clauses that differ only by Skolem constants. To this end, we process the set of processed clauses before creating samples from it. We tried different techniques to handle Skolem constants, as well as other processing methods:

- **Skolem filtering**: We discard clauses containing any Skolem constants.
- **Inference filtering**: We discard all clauses generated during proof search that are *not* part of the initial clauses.
- **Consistent Skolemisation**: We normalise Skolem constants inside all clauses, similarly to [UVv11]. That is, a clause $P(x, y, x)$, where x and y are Skolem constants, becomes $P(c_1, c_2, c_1)$.
- **Consistent normalisation**: Similarly to consistent Skolemisation, we normalise *all* symbols of a clause. That is, $P(x, y, x)$ as above becomes $c_1(c_2, c_3, c_2)$. This allows the ATP to discover similar groups of clauses, for example $a + b = b + a$ and $a \times b = b \times a$ both map to $c_1(c_2, c_3) = c_1(c_3, c_2)$, but on the other hand, this also maps different clauses such as $P(x)$ and $Q(z)$ to the same clause. Still, this method

is suitable in problem collections which do not share a common set of function constants, such as TPTP.

We denote the consistent Skolemisation/normalisation of a clause c described above as $\mathcal{N}(c)$.

4.6 Clause Ranking

This section describes how our internal guidance method influences the choice of unprocessed clauses using a previously constructed classifier.

At the beginning of proof search, the ATP loads the classifier. Some learning ATPs, such as E/TSM [Sch00], select and prepare knowledge relevant to the current problem before the proof search. However, as we store classifier data in a hash table, filtering irrelevant knowledge to the problem at hand would require a relatively slow traversal of the whole table, whereas lookup of knowledge is fast even in the presence of a large number of irrelevant facts. For this reason we do not filter the classification data per problem.

Every time the ATP chooses a clause from the unprocessed clauses C, it picks a clause $c \in C$ that maximises the relevance

$$R(c, \vec{f}) = r_{\text{ATP}}(c) + \text{NB}(\mathcal{N}(c), \vec{f})$$

where:

- $r_{\text{ATP}}(c)$ is an ATP function that calculates the relevance of a clause with traditional means (such as weight, age, ...),
- $\text{NB}(l, \vec{f})$ is the Naive Bayesian relevance function from Section 2.3 adapted to negative examples in Section 4.2,
- $\vec{f}$ is the current prover state as shown in Section 4.3, and
- $\mathcal{N}(c)$ is the normalisation function as introduced in Section 4.5.

4.7 Parameter Tuning

To automatically find good parameters for proof search, we employ *offline tuning* and *Particle Swarm Optimisation*. Offline tuning is fast, but it only estimates proof search performance and tunes only guidance-related parameters, such as σ_1, σ_2, and μ from Section 2.3. Particle Swarm Optimisation is slower, but precisely measures proof search performance and can be used to optimise any continuous-valued parameter.

4.7.1 Offline Tuning

Offline tuning analyses existing training data and attempts to find parameters that give proof-relevant clauses from the training data a high rank, while giving proof-irrelevant clauses a low rank. We calculate a score for every training datum S with the following formula, which adds for every proof-relevant clause the number of proof-irrelevant clauses

that were ranked higher:

$$\sum_{s \in S^+} \text{rank}_R(s, S^-)$$

where
- $S^+ = \{(l, \vec{f}) \mid (l, \vec{f}, o) \in S, o = (1, 0)\}$ is the set of positive examples in S,
- $S^- = S \setminus S^+$ is the set of negative examples in S,
- rank is defined in Definition 2.6, and
- R is the relevance formula from Section 4.6.

We sum up the results of the formula above for all training data and take the guidance parameters which minimise the value of the sum.

4.7.2 Particle Swarm Optimisation

Particle Swarm Optimisation (PSO) is a standard optimisation algorithm that can be applied to minimise the output of a function $f(\vec{x})$, where $\vec{x}$ is a vector of continuous values [PKB07]. A *particle* is defined by a location $\vec{x}$ (a candidate solution for the optimisation problem) and a velocity $\vec{v}$. Initially, p particles are created with random locations and velocities. Then, at every iteration of the algorithm, a new velocity is calculated for every particle and the particle is moved by that amount. The new velocity of a particle is:

$$\vec{v}(t+1) = \omega \times \vec{v}(t) + \phi_p \times \vec{r}_p \times (\vec{b}_p(t) - \vec{x}(t)) + \phi_g \times \vec{r}_g \times (\vec{b}_g(t) - \vec{x}(t))$$

where:
- $\vec{v}(t)$ is the old velocity of the particle,
- $\vec{b}_p(t)$ is the location of the best previously found solution among all particles,
- $\vec{b}_g(t)$ is the location of the best previously found solution of the particle,
- $\vec{r}_p$ and $\vec{r}_g$ are vectors of random numbers uniformly distributed in $[0, 1]$ that are randomly generated at each evaluation of the formula, and
- $\omega = 0.4$, $\phi_p = 0.4$, and $\phi_g = 3.6$ are constants.

We apply PSO to optimise the performance of an ATP on a problem set S. For this, we define $f(\vec{x})$ to be the number of problems in S the ATP can solve with a set of parameters being set to $\vec{x}$ and with timeout t. We then run PSO and take the best global solution obtained after n iterations. We fixed $t = 1s$, $p = 300$, and $|S| = 1000$. The algorithm has worst-case execution time $t \times p \times n \times |S|$.

4.8 Implementation

We implemented our internal guidance in Satallax version 2.8.[3] Satallax places several kinds of proof search commands on a priority queue: Among the 11 different commands in Satallax 2.8 are proposition processing, mating, and confrontation. Proof search processes the commands on the priority queue by descending priority, until a proof is found or a timeout is reached.

[3] The source code can be obtained at http://cl-informatik.uibk.ac.at/users/mfaerber/satallax.html.

4.8.1 Machine Learning

An analysis of several proof searches yielded that on average, more than 90% of commands put onto the priority queue of Satallax are proposition processing commands, which correspond to processing a clause from the set of unprocessed clauses in given-clause provers. For that reason, we decided to influence the priority of proposition processing commands, giving those propositions likely to be useful a higher priority. The procedure follows the one described in Section 4.6, but the ranking of a proposition is performed when the proposition processing command is put onto the priority queue. The Naive Bayes rank is added to the priority that Satallax without internal guidance would have assigned to the command. We pay attention to influence the priority of term processing commands only in moderation (by choosing appropriately small guidance parameters), to avoid discrimination of other types of commands in the priority queue.

To record training data, we use the terms from the proof search that contributed to the final proof. For this, Satallax uses `picomus` [Bie08] to construct a minimal unsatisfiable core.

4.8.2 Strategies

The priorities assigned to proof search commands are determined by *flags*, which are the settings Satallax uses for proof search. A set of flag settings is called a *mode* (in other ATPs frequently called *strategies*) and can be chosen by the user upon the start of Satallax.

Similar to other modern ATPs such as Vampire or E, Satallax supports timeslicing via *strategies* (in other ATPs frequently called *schedules*), which define a set of modes together with time amounts Satallax calls each mode with. Formally, a strategy is a sequence $[(m_1, t_1), \dots, (m_n, t_n)]$, where m_i is a mode and t_i the time to run the mode with. The total time of the strategy is the sum of times, i.e. $t_\Sigma(S) = \sum_{(m,t) \in S} t$. When running Satallax with a strategy S and a timeout $t_{\max}$, then all the times of the strategy are multiplied by $\frac{t_{\max}}{t_\Sigma(S)}$ if $t_{\max} > t_\Sigma(S)$, to divide the time between modes appropriately when running Satallax for longer than what the strategy S specifies. Then, every mode m_i in the strategy is run sequentially for time t_i until a proof is found or the timeout $t_{\max}$ is exceeded.

We extended Satallax to allow loading user-defined strategies, which was previously not possible as strategies were hard-coded. Furthermore, we implemented modifying flags via the command line, which is useful e.g. to change a flag among all modes of a strategy. We used this extensively in the automatic evaluation of flag settings via PSO, as shown in Subsection 4.7.2.

4.9 Evaluation

To evaluate the performance of our internal guidance method in Satallax, we use a THF0 [SB10] version (simply-typed higher-order logic) of the top-level theorems of the Flyspeck project, as generated by Kaliszyk and Urban [KU14]. The test set consists of 14185

Table 4.1: Overview of internal guidance results.

Type	Timeout	Unguided	Guided	Loss	Gain	Gain%
Online	1s × 1	2717	3374	59	716	+26%
Offline	1s × 2	2717	3428	75	786	+29%
Offline	30s × 2	3097	4028	106	1037	+33%

problems from topology, geometry, integration, and other fields. The premises of each problem are the actual premises that were used in the Flyspeck proofs, amounting to an average of 84.3 premises per problem.

For experiments with timeouts of 1s and 2s, we use an Intel Core i3-5010U CPU (2.1 GHz Dual Core, 3 MB Cache) and run at most one instance of Satallax at a time. For experiments with 30s timeout, we use an 48-core server with 2.2GHz AMD Opteron CPUs and 320GB RAM, running 10 instances of Satallax in parallel.

We use the Satallax 2.5 strategy (abbreviated as "S2.5"), because the newest strategy in Satallax 2.8 cannot always retrieve the terms that were used in the final proof, which is important to obtain training data. The "S2.5" strategy solves 2717 problems with a timeout of 1s. Doubling the timeout to 2s increases the number of solved problems to 3394. This increase is mostly due to the fact that Satallax tries up to 11 modes in 2s compared to only up to 7 modes in 1s. To measure the gain in solved problems more fairly, we create a strategy "S2.5_1s" which contains only those 7 modes that were already used during the 1s run. This strategy proves 2845 problems given a timeout of 2s. We use the "S2.5_1s" strategy throughout the remaining evaluation.

We propose two different scenarios to generate training data and to use it in subsequent proof searches, see Figure 4.1:

- **Online learning**: We run the ATP on every problem with internal guidance. For every problem the ATP solves, we update the classifier with the training data from the ATP proof.
- **Offline learning**: We first run the ATP on all problems without internal guidance, saving training data for every problem solved. We then create classification data from the training data and rerun the ATP with internal guidance on all problems.

We summarise our internal guidance results in Table 4.1 and explain them in more detail in the following paragraphs. We call a problem "lost" if Satallax with guidance can not solve it and Satallax without guidance can. Vice versa for "gained".

To evaluate online learning, we run Satallax on all Flyspeck problems by ascending order, accumulating training data and using it for all subsequent proof searches. We filter away terms in the training data that contain Skolem variables. As result, Satallax with online learning and a timeout of 1s solves 3374 problems (59 lost, 716 gained).

To evaluate offline learning, we first run Satallax without internal guidance on all problems, generating training data whenever Satallax finds a proof. Next, we create a

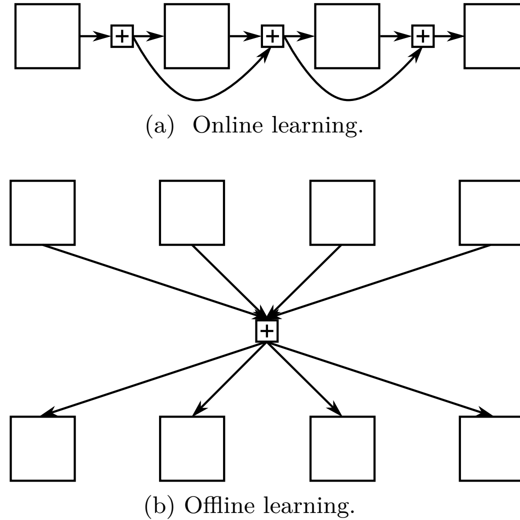


Figure 4.1: Comparison of online and offline learning. The large boxes symbolise an ATP proof search, which takes classifier data and returns training data (empty if no proof found). The small "+" boxes combine classifiers and training data, returning new classifier data.

classifier from the training data.[4] Finally, we run Satallax with internal guidance on all problems, using the same timeout as in the first run. We evaluated offline learning with timeouts of 1s and 30s. In the 1s case, unguided Satallax solves 2717 problems, whereas guided Satallax using inference filtering solves 3428 problems (75 lost, 786 gained). In the 30s case, unguided Satallax solves 3097 problems, whereas guided Satallax using Skolem filtering solves 4028 problems (106 lost, 1037 gained). The evolution of the number of solved problems is shown in Figure 4.2. The "jumps" in the data stem from changes of modes.

Offline learning can be parallelised, which makes it take less wall-clock time than online learning. However, offline learning evaluates every problem twice, taking potentially more CPU time. To know whether the increased effort pays off, we compare the performance of offline learning to running the ATP with doubled timeout. Offline learning with a timeout of 1s for both guided and unguided Satallax solves a total of 3503 problems. This means that offline learning solves 23.1% more problems than the unguided "S2.5_1s" strategy (using the same modes) and 3.2% more problems than the unguided "S2.5" strategy (using more modes), both running with a timeout of 2s. We conclude that using offline learning can be more effective than running the prover with a longer timeout and more modes.

We now compare the different clause processing options introduced in Section 4.5. For this, we create a classifier from the training data gathered during the 1s run. We then run Satallax with internal guidance in offline learning mode with 1s timeout. We perform this

[4]The effort required to create the classifier is insignificant. Creating a classifier with Skolem filtering from 3097 proofs takes 3s and results in a 1.8MB file.

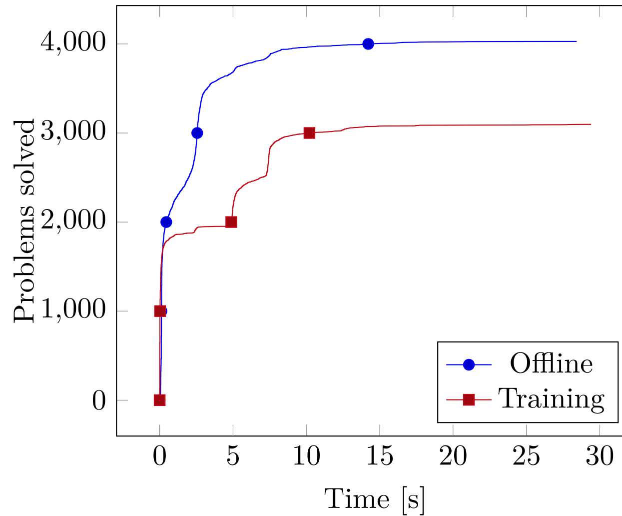


Figure 4.2: Problems solved in a certain time.

Table 4.2: Comparison of clause processing options.

Clause processing	Solved	Loss	Gain
Consistent normalisation	1911	920	114
Consistent Skolemisation	1939	885	107
None	2166	688	137
Skolem filtering	3395	98	776
Inference filtering	3428	75	786

procedure for each clause processing option. The results are given in Table 4.2. Internal guidance performs best when influencing only the priority of axioms (inference filtering), solving 786 problems that could not be solved by Satallax in 1s without internal guidance.

4.10 Related Work

A number of early learning and data based approaches to guide automated theorem provers has been surveyed in [DFGS99]. The Prover9 *hints* method [Ver96] was among the earliest attempts to directly influence proof search by learning from previous proofs. Hints allow the user to specify a set of clauses to treat in a special way. A similarly working *watch list* has been later integrated into E, along with other learning mechanisms [Sch01]. Further methods for guiding the actual proof search of ATPs using machine learning have been considered in the integration of Naive Bayes classifiers to select next proof actions in Enigma [JU17], where the clause selection in E uses a tree-based n-gram approach to approximate similarity to the learned proofs using a support vector machine classifier. Holophrasm [Wha16] introduces a theorem prover architecture using GRU

neural networks to guide the proof search of a tableaux style proof process of Metamath. TensorFlow neural network guidance was integrated into E [LISK17], showing that with batching and hybrid heuristics, it can solve a number of problems other strategies cannot solve. Finally, various reasons as to why the connection calculus is well suited for machine learning techniques, especially deep learning, are considered by Bibel [Bib17].

The *Machine Learning Connection Prover* (MaLeCoP) was the first leanCoP-based system to explore the feasibility of machine-learnt internal guidance [UVv11]. MaLeCoP relies on an external machine learning framework (using by default the SNoW system [CCRR99]), providing machine learning algorithms such as Naive Bayes and shallow neural networks based on perceptrons or winnow cells. During proof search, MaLeCoP sends features of its current branch to the framework, which orders the proof steps applicable in the current branch by their expected utility. The usage of a general framework eases experiments with different methods, but the prediction speed of MaLeCoP's underlying advisor system together with the communication overhead is several orders of magnitude lower than the raw inference speed of leanCoP. This was to some extent countered by fast query caching mechanisms and a number of strategies trading the machine-learnt advice for raw speed, yet the real-time performance of the system remains relatively low.

This motivated the creation of the *Fairly Efficient Machine Learning Connection Prover* (FEMaLeCoP), which improved speed by integrating a fast and optimised Naive Bayes classifier as shown in Section 2.3 into the prover [KU15a]. Naive Bayes was chosen because learning data can be easily filtered for the current problem, making the calculation of Naive Bayesian probabilities for a given branch efficient for each applicable contrapositive. FEMaLeCoP efficiently calculates the Bayesian probabilities of a given set of contrapositives by saving statistics directly in the contrapositive database, see Section 3.5. Performance is further improved by updating branch features from the previous branch, instead of fully recalculating them in every new branch.

Machine learning can also be applied to create strategies and strategy schedules for ATPs. Creating strategies (i.e. sets of parameters) for ATPs automatically has been researched in the Blind Strategymaker (BliStr) project [Urb15]. Given a set of ATP strategies that are known to perform well on different problem sets, new problems can benefit from custom strategy schedules that determine which strategies to apply for which time. The creation of such schedules was treated by Machine Learning of Strategies (MaLeS) [Kü14].

4.11 Conclusion

We have shown how to integrate internal guidance into ATPs based on the given-clause algorithm, using positive as well as negative examples. We have demonstrated the usefulness of this method experimentally, showing that on a simply-typed higher-order logic version of the Flyspeck problems, Satallax with internal guidance solves 23% more problems than Satallax without it.

ATPs with internal guidance could be integrated into hammer systems such as Sledgehammer (which can reconstruct Satallax proofs [SBP13]) or HOL(y)Hammer [KU15b],

continually improving their success rate with minimal overhead. It could also be interesting to learn internal guidance for ATPs from subgoals given by the user in previous proofs. Currently, we only learn from problems we could find a proof for, but in the future we could benefit from also considering proof searches that did not yield proofs. Furthermore, it would be interesting to see the effect of negative examples on existing ATPs with internal guidance, such as FEMaLeCoP [JU17]. Finding better characterisations of prover states is important to further improve the learning results.

Chapter 5

Monte Carlo Proof Search

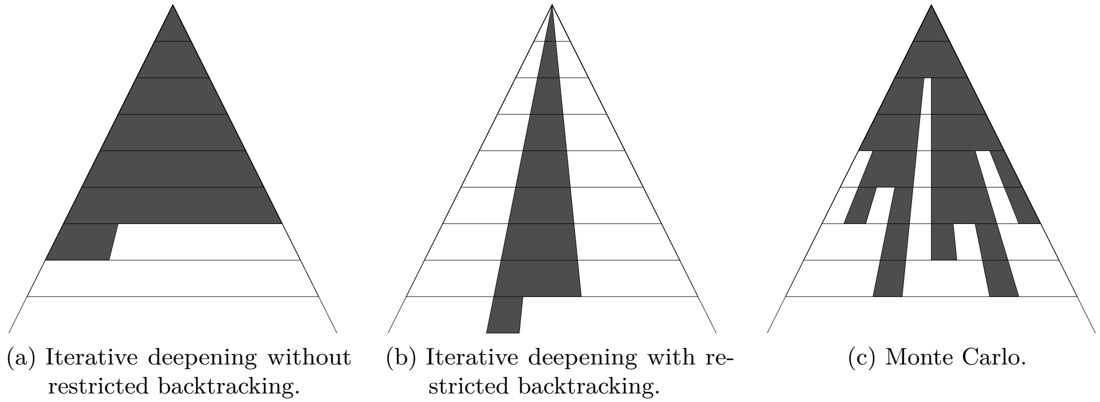
5.1 Introduction

Current automated theorem provers are still weak at finding more complicated proofs, especially over large formal developments [UHV10]. The search typically blows up after several seconds, making the chance of finding proofs in longer times exponentially decreasing [AKU12]. This behaviour is reminiscent of poorly guided search in games such as chess and Go. The number of all possible variants there typically also grows exponentially, and intelligent guiding methods are needed to focus on exploring the most promising moves and positions.

The guiding method that has recently very significantly improved automatic game play is Monte Carlo Tree Search (MCTS), i.e. expanding the search tree based on its (variously guided) random sampling [BPW+12]. MCTS has been found to produce state-of-the-art players for several games, most notably for the two-player game Go [SHM+16], but also for single-player games such as SameGame [SWTU12] and the NP-hard Morpion Solitaire [Ros11].

Theorem proving can be seen as a game. For instance, it has been modelled as a two-player game in the framework of game-theoretical semantics [Hin82], but it can also be seen as a combinatorial single-player game. As shown for example in the AlphaGo system [SHM+16], machine learning can be used to train good position evaluation heuristics even in very complicated domains that were previously thought to be solely in the realm of "human intuition". While "finishing the randomly sampled game" – as used in the most straightforward MCTS for games – is not always possible in ATP (it would mean finishing the proof), there is a chance of learning good *proof state evaluation heuristics* that will guide MCTS for ATPs in a similar way as e.g. in AlphaGo. One-step lookahead can help Vampire proof search [HRSV16], suggesting that MCTS, whose simulation phase can be seen as multi-step lookahead, can effectively guide proof search. It therefore seems reasonable to apply MCTS to the game of theorem proving.

In this chapter, we study MCTS methods that can guide the search in automated theorem provers. We focus on connection tableaux calculi and the leanCoP prover as introduced in Section 3.5, building on previous machine learning extensions of leanCoP [UVv11, KU15a]. For an intuition of the relationship between different proof search strategies, see Figure 5.1: Iterative deepening considers all potential proof trees of a certain depth before considering trees of higher depth. Restricted backtracking uniformly discards a set of potential proof trees. MCTS allows for a more fine-grained proof search,



(a) Iterative deepening without restricted backtracking.
(b) Iterative deepening with restricted backtracking.
(c) Monte Carlo.

Figure 5.1: The two main leanCoP strategies compared with Monte Carlo proof search.

searching different regions of the search space more profoundly than others, based on heuristics. To our knowledge, our approach is the first to apply MCTS to theorem proving.

We introduce MCTS in Section 5.2 and then propose a set of heuristics adapted to proof search to expand of a proof search tree using MCTS. We show an implementation in Section 5.7 and evaluate it in Section 5.8.

5.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a method to search potentially infinite trees by sampling random tree paths (called *simulations*) [BPW+12]. The outcome of simulations is used to estimate the quality of tree nodes, and MCTS steers search towards nodes with higher quality estimates.

Definition 5.1 (Tree)**.** A *tree* is a tuple $(N, n_0, \to)$, where N is a set of tree nodes, $n_0 \in N$ is the root node, and $\to \in N \times N$ is a cycle-free relation, i.e. there is no $n \in N$ such that $n \to^+ n$. We write that n' is a child of n iff $n \to n'$ and n' is a descendant of n iff $n \to^+ n'$. Every $n \in N$ is the child of at most one node in N.

We consider proof search as traversal of a (usually infinite) tree $(N, n_0, \to)$, such that N is the set of derivations (tableaux), n_0 is a derivation that consists of some word $\langle C, M, Path\rangle$ corresponding to the matrix M of a given problem, and $n \to n'$ iff n' can be obtained from n by a single application of a calculus rule. If $n \to n'$ by an application of the extension rule using the contrapositive c, then we write $n \xrightarrow{\text{ext}(c)} n'$. Proof search succeeds when we find a leaf node of N that is a proof.

Let $\rho \in N \to \mathbb{R}$ be a *reward function* that estimates the distance of an unclosed derivation in the proof search tree from a closed derivation. Then we can use Monte Carlo Tree Search to traverse the proof search tree, giving preference to regions that yield higher rewards. For this, we first define Monte Carlo trees:

Definition 5.2 (Monte Carlo Tree)**.** A *Monte Carlo tree* T for a tree $(N, n_0, \to)$ is a tuple $(N_T, \to_T, \rho_T)$, where $\to_T \subseteq \to^+$ and $\rho_T \in N \to \mathbb{R}$ is a mapping. We write that n' is a T-child of n iff $n \to_T n'$. The *initial Monte Carlo tree* T_0 is $(N_{T_0}, \to_{T_0}, \rho_{T_0})$ with $N_{T_0} = \{n_0\}$, $\to_{T_0} = \emptyset$ and $\rho_{T_0}(n) = 0$ for all n.

A single iteration of Monte Carlo Tree Search takes a Monte Carlo tree T and returns a new tree T' as follows:[1]

1. **Selection**: A node $n \in N_T$ with $n_0 \to_T^* n$ is chosen with a *child selection policy*, see Section 5.3.
2. **Simulation**: A child n_1 of n is randomly chosen with child probability $P(n_1 \mid n)$ to be the *simulation root*, see Section 5.4. Every tree node is chosen at most once to be a simulation root, to guarantee the exploration of the tree. From n_1, a sequence of random transitions $n_1 \to \cdots \to n_s$ is performed, where for every $i < s$, n_{i+1} is randomly selected with child probability $P(n_{i+1} \mid n_i)$.
3. **Expansion**: A node n_e from $n_1 \to \cdots \to n_s$ is selected with the *expansion policy*, see Section 5.6. The node n_e is added as a child to n with reward $\rho(n_s)$ (see Section 5.5) to yield the new tree T':

$$N_{T'} = N_T \cup \{n_e\} \qquad \to_{T'} = \to_T \cup \{(n, n_e)\} \qquad \rho_{T'} = \rho_T\{n_e \mapsto \rho(n_s)\}$$

In the next sections, we show heuristics for the child selection policy, child probability, reward, and expansion policy.

5.3 Child Selection Policy

UCT (Upper Confidence Bounds for Trees) is a frequently used child selection policy for Monte Carlo Tree Search [KS06]. It uses $\text{visits}_T(n)$, which is the number of T-descendants of n, and $\overline{\rho}_T(n)$, which is the average T-descendant reward of n.

$$\text{visits}_T(n) = |\{n' \mid n \to_T^+ n'\}| \qquad \overline{\rho}_T(n) = \frac{\sum\{\rho_T(n') \mid n \to_T^* n'\}}{\text{visits}_T(n)}$$

Given a node n, UCT ranks every T-child n' of n with

$$\text{uct}(n, n') = \overline{\rho}_T(n') + C_p \sqrt{\frac{\ln \text{visits}_T(n)}{\text{visits}_T(n')}}$$

Here, C_p is called the *exploration constant*, where small values of C_p prefer nodes with higher average descendant reward and large values of C_p prefer nodes with fewer visits. In the UCT formula, division by zero is expected to yield ∞, so if a node n has unvisited children, one of them will be selected by UCT.

The UCT child selection policy $cs_T(n)$ recursively traverses the Monte Carlo tree T starting from the root n_0. $cs_T(n)$ chooses the T-child of n with maximal UCT value and recurses, unless n has no T-child, in which case n is returned:

[1] Frequently, MCTS is described to have a *backpropagation* step that adds rewards to the ancestors of the newly added nodes. We omit this step, adapting the child selection policy instead.

$$cs_T(n) = \begin{cases} cs_T\left(\underset{n' \in \{n' \mid n \to_T n'\}}{\arg\max} \operatorname{uct}(n, n')\right) & \text{if } \exists n'.\ n \to_T n' \\ n & \text{otherwise} \end{cases}$$

5.4 Child Probability

The child probability $P(n' \mid n)$ determines the likelihood of choosing a child node n' of n in a simulation. We show three different methods to calculate the child probability.

- The *baseline probability* assigns equal probability to all children, i.e. $P(n' \mid n) \propto 1$.
- The *open branches probability* steers proof search towards derivations with fewer open branches, by assigning to n' a probability inversely proportional to the number of open branches in n'. Therefore, $P(n' \mid n) \propto 1/(1 + |b_o(n')|)$, where $b_o(n)$ returns the open branches in n.
- The *Naive Bayes probability* attributes to n' a probability depending on the calculus rule applied to obtain n' from n: In case the extension rule was not used, the node obtains a constant probability. If the extension rule was used, the formula NB introduced in Section 2.3 is used, requiring contrapositive statistics from previous proofs. However, as NB does not return probabilities, we use it to rank contrapositives by the number of contrapositives with larger values of NB:

$$R_{\text{NB}}(n, c) = \operatorname{rank}_{\text{NB}}\left((c, \vec{f}(n)), \left\{(c', \vec{f}(n)) \mid n \xrightarrow{\operatorname{ext}(c')} n'\right\}\right)$$

where $\vec{f}(n)$ denotes the features of the derivation n. Then, we assign to nodes as probability the inverse of the Naive Bayes rank:

$$P(n' \mid n) \propto \begin{cases} 1/R_{\text{NB}}(n, c) & \text{if } n \xrightarrow{\operatorname{ext}(c)} n' \\ 1 & \text{otherwise} \end{cases}$$

5.5 Reward

The reward heuristic estimates the likelihood of a given derivation to be closable. In contrast, most prover heuristics (such as child probability) only compare the quality of children of the same node. We use our reward heuristics to evaluate the last node n of a simulation.

Several heuristics in this section require a normalisation function, for which we use a strictly increasing function $\operatorname{norm} \in [0, \infty) \to [0, 1)$ that fulfils $\lim_{x \to \infty} \operatorname{norm}(x) = 1$ and $\operatorname{norm}(0) = 0$. For example, $\operatorname{norm}(x) = 1 - (x + 1)^{-1}$.

- The *branch ratio reward* determines the reward to be the ratio of the number of closed branches and the total number of branches, i.e. $\rho(n) = |b_c(n)|/|b(n)|$.
- The *branch weight reward* is based on the idea that many open branches with large literals are indicators of a bad proof attempt. Here, the size $|l|$ of a literal is measured by the number of symbol occurrences in l. Furthermore, the closer to

the derivation root a literal appears, the more characteristic we consider it to be for the derivation. Therefore, the reward is the average of the inverse size of the branch leafs, where every leaf is weighted with the normalised depth of its branch.

$$\rho(n) = \frac{1}{|b_o(n)|} \sum_{b \in b_o(n)} \frac{\operatorname{norm}(\operatorname{depth}(b))}{|\operatorname{leaf}(b)|}$$

- The *machine-learnt closability reward* assumes that the success ratio of closing a branch in previous derivations can be used to estimate the probability that a branch can be closed in the current derivation. This needs the information about attempted branches in previous derivations, and which of these attempts were successful. We say that a literal l stemming from a clause c is attempted to be closed during proof search when l lies on some branch. The attempt is successful iff proof search manages to close all branches going through l. Given such data from previous proof searches, let $p(l)$ and $n(l)$ denote the number of attempts to close l that were successful and unsuccessful, respectively. We define the *unclosability* of a literal l as $\frac{n(l)}{p(l)+n(l)}$. However, the less data we have about a literal, the less meaningful our statistics will be. To account for this, we introduce *weighted unclosability*: We assume that a literal that never appeared in previous proof searches is most likely closable, i.e. its weighted unclosability is 0. The more often a literal was attempted to be closed, the more its weighted unclosability should converge towards its (basic) unclosability. Therefore, we model the probability of l to be closable as

$$P(l \text{ closable}) = 1 - \operatorname{norm}(p(l) + n(l)) \frac{n(l)}{p(l) + n(l)}$$

  Finally, the closability of a derivation is the mean closability of all leafs of open branches of the derivation, i.e. the final reward formula is

$$\rho(n) = \sum_{b \in b_o(n)} \frac{P(\operatorname{leaf}(b) \text{ closable})}{|b_o(n)|}$$

To measure the efficiency of a reward heuristic, we introduce *discrimination*: Assume that an MCTS iteration of the Monte Carlo tree T starts a simulation from the node n_p and finds a proof. Then the discrimination of T is the ratio of the average reward on the Monte Carlo tree branch from the root node n_0 to n_p and the average reward of all Monte Carlo tree nodes. Formally, let the average reward of a set of nodes N be

$$\overline{\rho}_T(N) = \frac{\sum \{\rho_T(n) \mid n \in N\}}{|N|}$$

Then the discrimination of T is

$$\frac{\overline{\rho}_T(\{n \mid n_0 \to_T^* n, n \to_T^* n_p\})}{\overline{\rho}_T(\{n \mid n_0 \to_T^* n\})}$$

5.6 Expansion Policy

The expansion policy determines which node n_e of a simulation $n_1 \to \dots \to n_s$ is added to the Monte Carlo tree. We implement two different expansion policies:

- The *default expansion policy* adds n_1, i.e. the simulation root, to the MC tree.
- The *minimal expansion policy* picks n_e to be the smallest of the simulation nodes with respect to a given norm $|\cdot|$, such that for all i, $|n_e| \leq |n_i|$. If multiple n_e are admissible, the one with the smallest index e is picked. We consider two norms on nodes:
  1. The first norm measures the number of open branches.
  2. The second norm measures the sum of depths of open branches.

The minimal expansion policy is similar to restricted backtracking in the sense that it restricts proof search to be resumed only from certain states, thus resulting in an incomplete search.

5.7 Implementation

We implemented Monte Carlo proof search (MCPS) based on leanCoP.[2] In our implementation, leanCoP provides the search tree and MCTS chooses which regions of the tree to search. Unlike for the traditional leanCoP, the depth of the search tree is not limited. To guarantee nonetheless that simulations terminate, simulations are stopped after a fixed number of simulation steps $s_{\max}$.

While it is possible to run MCPS from the root node until a proof is found, we found it to perform better when it serves as *advisor* for leanCoP. We show this in Listing 5.1, assuming for a simpler presentation that the default expansion policy from Section 5.6 is used: MCPS as performed by `mcps` L $Path$ σ attempts to find a connection proof for $\langle \{L\}, M, Path \rangle$. The result is a lazy list of Monte Carlo iterations, where an iteration consists of a Monte Carlo tree and possibly a proof discovered during the simulation performed in the iteration. The first `maxIterations` are considered in line 4: When `maxIterations` is set to 0, proof search behaves like leanCoP, and in case it is set to ∞, the whole proof search is performed in the MCPS part. As MCPS is performed lazily, MCPS is performed for less than `maxIterations` iterations when it discovers some proof contributing to the final closed derivation. Here, the lazy list characterisation introduced in Subsection 3.6.2 turns out to be permit a very concise implementation as well as an easy integration of techniques such as restricted backtracking. As soon as all proofs discovered during MCPS were considered (line 5), the tree T of the final Monte Carlo iteration `last mc` is obtained and the children of the root of T are sorted by decreasing average T-descendant reward $\overline{\rho}_T$ (line 6). Finally, the last applied proof step of each child is processed like in the lazy list implementation (lines 7–11).

The array substitution technique from Section 3.5 requires that the proof search backtracks only to states whose substitution is a subset of the current state's substitution.

[2]The source code is available at http://cl-informatik.uibk.ac.at/users/mfaerber/cop.html.

Listing 5.1: Monte Carlo Proof Search as advisor.

```
1  prove [] path lim sub = [sub]
2  prove (lit : cla) path lim sub =
3    let
4      mc = take maxIterations (mcps lit path sub)
5      proofs1 = mapMaybe getProof mc
6      proofs2 = last mc & root & children & sortOn avgReward & concatMap
7        (\ child -> case lastStep child of
8          Reduction sub1 -> [sub1]
9          Extension (sub1, cla1) ->
10           if lim <= 0 then []
11           else prove cla1 (lit : path) (lim - 1) sub1)
12   in concatMap (prove cla path lim) (proofs1 ++ proofs2)
```

However, because this requirement is not fulfilled for MCPS, we use association lists for substitutions.

5.8 Evaluation

We evaluate the presented heuristics on the bushy MPTP2078 problems, with definitional clausification and a timeout of 10s for each problem. Before evaluation, we collect training data for the machine learning heuristics by running leanCoP on all bushy problems with a timeout of 60s.

The base configuration of monteCoP uses the open branches probability (see Section 5.4), the branch ratio reward (see Section 5.5), and the minimal expansion policy 1 (see Section 5.6), where the maximal simulation depth $s_{\max} = 50$, the exploration constant $C_p = 1$, and the maximal number of MCTS iterations `maxIterations` $= \infty$. For any heuristic h not used in the base configuration, we replace the default heuristic with h and evaluate the resulting configuration. The results are shown in Table 5.1: The heuristics that most improve the base configuration are the machine-learnt closability reward and the minimal expansion policy 2.

We explore a range of values for several numeric parameters, for which we show results in Figure 5.2: The maximal number of MCTS iterations `maxIterations` performs best between 20 and 40, see Figure 5.2a: Below 20, MCTS cannot provide any meaningful quality estimates, and above 40, the quality estimates do not significantly improve any more, while costing computational resources. The exploration constant $C_p \approx 0.75$ gives best results, where the machine-learnt closability reward achieves a local optimum, see Figure 5.2b: At such an optimum, exploration and exploitation combine each other best, therefore the existence of such an optimum is a sanity check for reward heuristics (which the branch ratio reward does not pass). The maximal simulation depth $s_{\max} \approx 20$ seems to perform best, see Figure 5.2c. Above this value, the number of solved problems decreases, since the number of actually performed simulation steps decreases, as shown

Table 5.1: Comparison of Monte Carlo heuristics. Iterations, simulation steps and discrimination ratio are averages on the 196 problems solved by all configurations.

Configuration	Iterations	Sim. steps	Discr.	Solved
Base	116.46	1389.82	1.37	332
Uniform probability	949.62	17539.59	1.31	237
NB probability	528.39	8014.03	1.35	248
Random reward	104.88	1167.98	1.19	364
Branch weight reward	108.13	1268.88	1.12	334
ML closability reward	108.52	1151.61	**2.30**	**367**
Default exp. pol.	371.81	4793.58	1.38	328
Minimal exp. pol. 2	224.72	2769.12	1.40	348

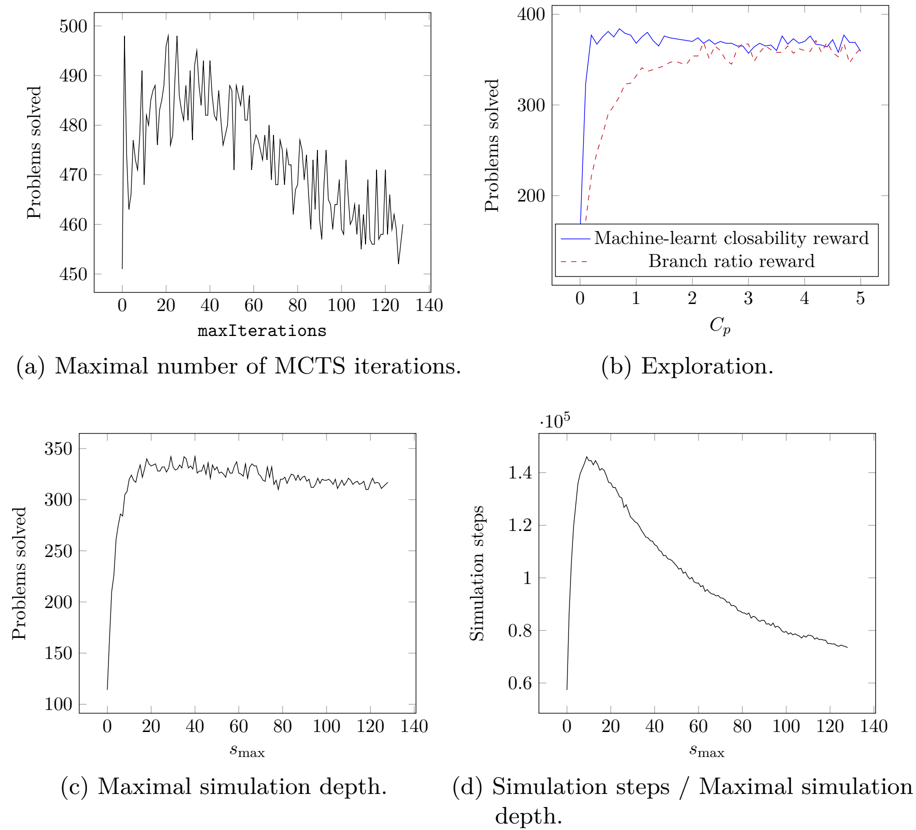
in Figure 5.2d. This might be explained by the fact that at higher simulation depths, the computational effort to calculate the set of possible steps increases, for example because the substitution contains more and larger elements.

We adapt the base configuration to use the best heuristics from Table 5.1 and the best values for parameters discussed in Figure 5.2, yielding $s_{\max} = 20$, $C_p = 0.75$, and `maxIterations` $= 27$. This improved configuration solves 538 problems, compared to 509 solved by the best single leanCoP strategy.

5.9 Conclusion

We have proposed a combination of Monte Carlo Tree Search and automated theorem proving, including a number of proof-state evaluation heuristics, some of which are learnt from previous proofs. This is the first time Monte Carlo Tree Search has been used to guide an automated theorem prover.

MCTS provides a theoretically founded fine-grained mechanism to control the search space of tableaux-based theorem provers based on random sampling and state evaluation heuristics, which might eventually even replace iterative deepening. We have shown that a fast rollout policy combined with a machine-learnt state evaluation heuristic and a custom expansion policy produces the best results. The strength of the current system has turned out to be its function as advisor for existing provers, demonstrated by our integration into leanCoP. This opens a wide space of future work, profiting from the ongoing research in MCTS; examples include self-updating reward heuristics, adaptive tuning of rollout policies during search [Ros11], combination of online and offline knowledge with the All-Moves-As-First (AMAF) heuristic [GS07], and different characterisations of tableaux search. Furthermore, identifying controversial choices in the base prover would allow using the Monte Carlo prover as advisor more efficiently. To increase efficiency, arrays should be used instead of association lists as substitution data structure, requiring precautions when changing states. The performed machine



(a) Maximal number of MCTS iterations.

(b) Exploration.

(c) Maximal simulation depth.

(d) Simulation steps / Maximal simulation depth.

Figure 5.2: Parameter influence.

learning experiments are promising enough to justify the enhancement of Monte Carlo Proof Search with stronger heuristics, such as neural networks. While we applied Monte Carlo Tree Search to theorem proving as a single-player game, it could also be used to treat theorem proving as two-player game.

Chapter 6

Proof Reconstruction

6.1 Introduction

The output of automated theorem provers is usually not trusted, as for performance reasons, few ATPs implement small, trusted kernels à la LCF. One way to certify the correctness of ATP proofs is to translate them to interactive theorem provers. Certification of ATP proofs is also important for the integration of ATPs into ITPs, providing automation in the form of proof tactics or automated justification for smaller steps. However, the proofs returned by powerful ATPs such as E and Vampire can be difficult to verify and reconstruct in ITPs. This is due to the ATP's amount and complexity of inference rules. One solution to this problem is to extract only the used premises from the ATP proofs and use a simpler ATP to find a new proof that is easier to reconstruct.

Most ATPs convert their input problems to clausal normal form as preprocessing step. To reconstruct the resulting clausal proofs in an ITP, it is necessary to verify in the ITP the conversion to clausal normal form. The ATP nanoCoP has demonstrated that a prover not requiring clausification can be effectively implemented. The reconstruction of nonclausal proofs eliminates the necessity of proving the correctness of the clausification, but on the other hand, translating the proofs is more involved.

In this chapter, we describe the reconstruction of proofs in HOL Light: After reducing problems to first-order logic in Section 6.2, we translate clausal proofs from Metis (Section 6.3) and leanCoP (Subsection 6.4.3) as well as nonclausal proofs from nanoCoP (Subsection 6.4.4). To this end, we propose variants of the connection calculi adapted for proof reconstruction in Subsection 6.4.1. The resulting proof search methods can be used for example as part of a hammer system such as HOL(y)Hammer. We evaluate the performance of our implementations on HOL Light problem sets in Section 6.5.

6.2 Translation to First-Order Logic

Many ATPs treat problems in first-order logic. To convert a HOL problem to a first-order problem, there exists a multitude of translation methods [Har96, Hur03, MP08, Bla12], differing in their treatment of types, lambda terms, Hilbert's ϵ operator and many more aspects. HOL Light includes a translation method to first-order logic as part of its integrated model elimination prover MESON. To translate HOL goals to first-order logic,

we reuse large parts of the MESON infrastructure in HOL Light. We now explain the details of this translation.

The translation is realised as a tactic that transforms a goal C together with some premises $\{P_1, \dots, P_n\}$ to a format that can be passed to a first-order ATP. The translation tactic proceeds as follows: First, the tactic refutes the goal, yielding the goal $\neg C \to \bot$. Next, the tactic adds the premises to the goal via `POLY_ASSUME_TAC`, which instantiates premises containing polymorphic constants with the types of the objects they are used with. This yields a goal of the shape $P_1 \to \dots \to P_n \to \neg C \to \bot$. Furthermore, the translation tactic performs the following steps:

- Elimination of Hilbert's ϵ operator: $\epsilon x.Px$ denotes an object x for which Px holds. HOL Light's `SELECT_ELIM_TAC` replaces occurrences of the form $\epsilon x.Px$ by a fresh variable v, adding the assumption $\forall x.Px \to Pv$. As this step does not preserve logical equivalence, it can make a provable goal unprovable.
- Fixing of function arities via currying: If the same function symbol appears multiple times in the goal with a different number of arguments, e.g. fx and fxy, it cannot be directly translated to a first-order function, as first-order functions have a fixed arity. The translation thus replaces applications of such functions with an application operator I, turning fx into $I(f, x)$ and fxy into $I(I(f, x), y)$.
- Elimination of lambda abstractions (via lambda lifting), β-conversion
- Conversion to negation normal form
- Instantiation of universally quantified variables with fresh variables
- Skolemisation

For clausal ATPs, the tactic additionally performs the following steps:

- Conversion to conjunctive normal form (CNF)
- Splitting: Performs case distinction for disjunctions, producing new subgoals. For example, this might break the goal $a \lor b \lor c \to g$ into subgoals $a \to g$, $b \to g$, and $c \to g$. The splitting limit specifies the maximum number of times case distinction is performed on a single disjunction.

For ATPs that do not support equality in their calculi (such as MESON), the tactic adds equality axioms to the goal. The antecedents $\{C_1, \dots, C_n\}$ of the final goal $C_1 \to \dots \to C_n \to \bot$ are the clauses that are passed to the ATP.

Example 6.1. Consider the Flyspeck lemma `length_eq_imp_length_tl_eq` as an example of a HOL goal to translate:

$$|s_1| = |s_2| \to |\operatorname{tl} s_1| = |\operatorname{tl} s_2|$$

Here, $|x|$ stands for the length of a list x and $\operatorname{tl} x$ denotes the tail of a list x. We wish to prove the goal using the following HOL Light theorems:

$$\forall l. l \neq [] \to |\operatorname{tl} l| = |l| - 1 \qquad \text{(LENGTH_TL)}$$

$$\forall l. |l| = 0 \leftrightarrow l = [] \qquad \text{(LENGTH_EQ_NIL)}$$

We first refute the goal, yielding:

$$\neg(|s_1| = |s_2| \to |\operatorname{tl} s_1| = |\operatorname{tl} s_2|) \to \bot$$

Then, we add the premises with POLY_ASSUME_TAC, yielding:

$$\texttt{LENGTH_TL} \to \texttt{LENGTH_EQ_NIL} \to \neg(|s_1| = |s_2| \to |\operatorname{tl} s_1| = |\operatorname{tl} s_2|) \to \bot$$

Here, POLY_ASSUME_TAC instantiates the polymorphic constant tl to match the types present in the conjecture. The translation produces the following (untyped) first-order problem, where v_1 and v_2 are the only variables:

$$\begin{aligned}
& v_1 = [] \vee |\operatorname{tl} v_1| = |v_1| - 1 && (\texttt{LENGTH_TL}') \\
\to\, & |v_2| \neq 0 \vee v_2 = [] && (\texttt{LENGTH_EQ_NIL}_{\to}) \\
\to\, & |v_2| = 0 \vee v_2 \neq [] && (\texttt{LENGTH_EQ_NIL}_{\Longleftarrow}) \\
\to\, & |s_1| = |s_2| \to |\operatorname{tl} s_1| \neq |\operatorname{tl} s_2| && (\text{negated conjecture}) \\
\to\, & \bot
\end{aligned}$$

This procedure produces untyped HOL problems, corresponding to Leslie-Hurd's *uHOL* problem set [Hur03]. As Blanchette showed in his PhD thesis [Bla12], encoding types as in Isabelle's Sledgehammer [MP08] can increase the number of reconstructible problems. We expect similar gains for HOL Light when using different translations. Due to the similarity of translations to first-order logic in MESON, leanCoP, nanoCoP, and Metis, we believe that all these methods could profit from such an improvement with relatively little adaptation.

6.3 Metis

The paramodulation-based prover Metis was designed with a small certified proof core to simplify its integration with interactive theorem provers, such as HOL4 and Isabelle/HOL [MP08]. To translate a proof found by Metis to HOL Light, we translate all Metis inferences to equivalent HOL Light inferences. As Metis proofs are untyped, this involves the reconstruction of the types of the variables used in the proof.

The calculus used in Metis as well as the HOL Light translation of each calculus rule is shown in Figure 6.1. We use the symbols C and D for clauses, L for a literal, t for a term, and p for a path (denoting the position of a subterm). The complement $\overline{L}$ is defined as in Section 3.2. The term of a literal L at position p is denoted by $L[p]$, and the replacement of a subterm of L at position p by a term t is denoted by $L[p \mapsto t]$.

In our previous work on reconstructing Metis proofs [FK15a], the reconstruction of a proof yielded a *most specific* HOL theorem: For example, when translating the reflexivity rule for a variable x, the most specific HOL translation attempted to infer the type of x by inspecting instances of the substitution rule: When a substitution σ maps the variable x to a constant c, one can infer that x will have the type of c. This can be seen in Figure 6.2: When reconstructing the HOL proof of $g(y) = f(x)$, the reconstruction can infer the types of the variables y and x because the substitution rule was used in a lower part of the proof, instantiating y with a and x with b.

Metis	HOL Light
$\dfrac{}{C}\ \text{axiom}\ C$	$\dfrac{\Gamma \vdash C}{\Gamma \vdash \sigma_{tm} C}\ \texttt{INST}$
$\dfrac{}{\{L, \neg L\}}\ \text{assume}\ L$	$\dfrac{\dfrac{}{\vdash \forall t. t \lor \neg t}\ \texttt{EXCLUDED_MIDDLE}}{\vdash L \lor \neg L}\ \texttt{SPEC}\ L$
$\dfrac{}{\{t = t\}}\ \text{refl}\ t$	$\dfrac{}{\vdash t = t}\ \texttt{REFL}$
$\dfrac{C}{\sigma C}\ \text{subst}\ \sigma$	$\dfrac{\Gamma \vdash C}{\Gamma \vdash \sigma_{ty} \sigma_{tm} C}\ \texttt{INSTANTIATE}$
$\dfrac{}{\{\overline{L}, L[p] \neq t, L[p \mapsto t]\}}\ \text{eq}\ L\ p\ t$	$\dfrac{\dfrac{\dfrac{\dfrac{}{L \vdash L}\ \texttt{ASSUME} \quad \dfrac{}{L[p] = t \vdash L[p] = t}\ \texttt{ASSUME}}{L, L[p] = t \vdash L[p \mapsto t]}\ \texttt{CONV_RULE}}{L \vdash L[p] \neq t \lor L[p \mapsto t]}\ \texttt{DISCH_DISJ}}{\vdash \overline{L} \lor L[p] \neq t \lor L[p \mapsto t]}\ \texttt{DISCH_DISJ}$
$\dfrac{\{L\} \cup C \quad \{\neg L\} \cup D}{C \cup D}\ \text{resolve}\ L$	$\dfrac{\dfrac{\dfrac{\Gamma \vdash C'}{\Gamma \vdash L \lor C}\ \texttt{FRONT}}{\Gamma \vdash \sigma_{ty}(L \lor C)}\ \texttt{ITY} \quad \dfrac{\dfrac{\Delta \vdash D'}{\Delta \vdash \neg L' \lor D}\ \texttt{FRONT}}{\Delta \vdash \sigma_{ty}(\neg L' \lor D)}\ \texttt{ITY}}{\Gamma \cup \Delta \vdash \sigma_{ty}(C \lor D)}\ \texttt{RESOLVE}$

Figure 6.1: Reconstruction of the Metis proof rules in HOL Light.

$$\dfrac{\dfrac{}{g(a) \neq f(b)}\ \text{axiom} \quad \dfrac{\dfrac{\dfrac{}{f(x) = g(y)}\ \text{axiom} \quad \dfrac{\dfrac{\dfrac{}{x_0 = x_0}\ \text{refl} \quad \dfrac{}{x_0 \neq x_0 \lor x_0 \neq y_0 \lor y_0 = x_0}\ \text{eq}}{x_0 \neq y_0 \lor y_0 = x_0}\ \text{resolve}}{f(x) \neq g(y) \lor g(y) = f(x)}\ \text{subst}}{g(y) = f(x)}\ \text{resolve}}{g(a) = f(b)}\ \text{subst}}{\square}\ \text{resolve}$$

Figure 6.2: Metis proof of $f(x) = g(y) \to g(a) \neq f(b) \to \bot$.

The *most specific* translation fails to reconstruct some proofs because not all variables in a proof are necessarily substituted. For that reason, we implemented a reconstruction that translates Metis proofs to the *most general* HOL theorem. In this translation, we do not need to keep a map from variables to types, as a variable will always have the most general type. When the variable is used in an application, its type is refined via type checking. For this, we convert HOL terms to HOL preterms, for which type checking is implemented in HOL Light.

The reconstruction of a Metis proof is done recursively top-down, starting at the proof of the empty clause ($\Box$). We take care not to instantiate variables appearing in the context Γ, as this would change the conjecture we are about to prove. We now detail the reconstruction of inferences shown in Figure 6.1.

- Axiom C: We match the first-order clause C with all available higher-order disjunctions, to obtain a term substitution σ_{tm} and a higher-order theorem $\Gamma \vdash C$. We instantiate C with σ_{tm}.
- Assume L and Reflexivity t: The translation of both rules is trivial, by translating L and t to their most general HOL counterparts.
- Substitution σ: As the substitution σ can lead to the specialisation of the types of variables in C, we have to do type checking. For this, we create (untyped) HOL preterms of the shape $s = t$ for every element $s \mapsto t$ in σ. We then create the conjunction of these preterms and typecheck it, yielding a HOL term substitution σ_{tm}. As the types in σ_{tm} may contain fresh variables, we unify the types of the variables in C with the types of the variables in σ_{tm}, yielding a HOL type substitution σ_{ty}. Finally, we instantiate C with σ_{ty} and σ_{tm}.
- Equality $L\ p\ t$: We translate L, p, and t to HOL, unifying the types of $L[p]$ and t. We then rewrite the term $L[p]$ to t with `CONV_RULE` and `PATH_CONV` p, and move the assumptions from the context with `DISCH_DISJ` to create a disjunction. Here, `DISCH_DISJ` is similar to the `DISCH` rule, but creates a disjunction instead of an implication: It takes a theorem $\Gamma \vdash C$ and a literal L, creating $\Gamma \setminus \{L\} \vdash \overline{L} \vee C$.
- Resolve L: The literals L and $\neg L$ are not necessarily the first disjuncts of the given HOL conclusions C' and D'. Therefore we have to identify L and $\neg L'$ in C' and D' such that L and L' correspond to the first-order L and the types of L and L' can be unified. This yields a type substitution σ_{ty}, which we orient to ensure that no type variable appearing in Γ is substituted. We then move L and $\neg L'$ to the `FRONT` of the disjunctions, instantiate the disjunctions with σ_{ty} (`ITY` stands for `INST_TYPE`), and resolve them.

6.4 Connection Proofs

In this section, we show how to certify connection tableaux proofs by reconstructing them in HOL Light. To abstract from the technical details, we give translations for both clausal and nonclausal versions of connection proofs to Gentzen's sequent calculus LK [Gen35].

$$\text{Start} \quad \frac{\bigwedge_i \langle X_i, M, \{\}\rangle}{\varepsilon, M, \varepsilon}\,\text{S} \quad \text{where } \{X_1, \ldots, X_n\} \text{ is copy of } C \in M$$

$$\text{Reduction} \quad \frac{}{L, M, \mathit{Path} \cup \{L'\}}\,\text{R} \quad \text{where } \sigma(L) = \sigma(\overline{L'})$$

$$\text{Extension} \quad \frac{\bigwedge_i \langle L_i, M, \mathit{Path} \cup \{L\}\rangle}{L, M, \mathit{Path}}\,\text{E}$$

where $\{L_1, \ldots, L_n\} \cup \{L'\}$ is copy of $C \in M$ and $\sigma(L) = \sigma(\overline{L'})$

Figure 6.3: Clausal connection calculus for translation.

$$\text{Start} \quad \frac{\bigwedge_i \langle X_i, M, \{\}\rangle}{\varepsilon, M, \varepsilon}\,\text{S} \quad \text{where } \{X_1, \ldots, X_n\} \text{ is copy of } C \in M$$

$$\text{Reduction} \quad \frac{}{L, M, \mathit{Path} \cup \{L'\}}\,\text{R} \quad \text{where } \sigma(L) = \sigma(\overline{L'})$$

$$\text{Extension} \quad \frac{\bigwedge_i \langle X_i, M[C_1 \backslash C_2], \mathit{Path} \cup \{L\}\rangle}{L, M, \mathit{Path}}\,\text{E}$$

where $\{X_1, \ldots, X_n\}$ is the β-clause of C_2 with respect to L', C_2 is copy of C_1, C_1 is e-clause of M with respect to $\mathit{Path} \cup \{L\}$, C_2 contains L' with $\sigma(L) = \sigma(\overline{L'})$

$$\text{Decomposition} \quad \frac{\bigwedge_i \langle X_i, M, \mathit{Path}\rangle}{M', M, \mathit{Path}}\,\text{D} \quad \text{where } \{X_1, \ldots, X_n\} \in M'$$

Figure 6.4: Nonclausal connection calculus for translation.

6.4.1 Connection Calculi for Proof Translation

In the presentation of the connection calculi in [Ott11], all proof rules have a fixed number of premises. To ease the translation of proofs, we present slightly reformulated versions of the calculi. We introduce the following notation for rules with an arbitrary number of premises:

$$\frac{\bigwedge_i P_i}{C} \equiv \frac{P_1 \quad \ldots \quad P_n}{C}$$

The reformulated calculi for translation are shown in Figures 6.3 and 6.4. The words of the original calculi were $\langle C, M, \mathit{Path}\rangle$. In the reformulated calculi, the words are $\langle X, M, \mathit{Path}\rangle$, where X denotes an arbitrary clause element, i.e. a matrix or a literal. In the new calculi, the axiom rule becomes obsolete. Proofs can be trivially translated between the connection calculi in this chapter and those shown in Section 3.2.

$$\cfrac{\cfrac{Q, M, \{\}}{\epsilon, M, \epsilon}\,\text{S}}{} \qquad \cfrac{\cfrac{}{\neg P(x'), M', \{Q\}}\,\text{E} \quad \cfrac{\cfrac{\cfrac{}{P(s\hat{x}), \hat{M}, \{Q, P(sx')\}}\,\text{E} \quad \cfrac{}{\neg Q, \hat{M}, \{Q, P(sx')\}}\,\text{R}}{\left[[\neg P(s^2\hat{x})] \begin{bmatrix} P(s\hat{x}) \\ \neg Q \end{bmatrix}\right], \hat{M}, \{Q, P(sx')\}}\,\text{D}}{P(sx'), M', \{Q\}}\,\text{E}}{Q, M, \{\}}\,\text{E}$$

Figure 6.5: Proof in the nonclausal connection calculus for translation.

$$\cfrac{\cfrac{}{\neg P(x'), M, \{Q\}}\,\text{E} \quad \cfrac{\cfrac{\cfrac{}{\neg P(\bar{x}), M, \{Q, P(sx'), P(s\hat{x})\}}\,\text{E}}{P(s\hat{x}), M, \{Q, P(sx')\}}\,\text{E} \quad \cfrac{}{\neg Q, M, \{Q, P(sx')\}}\,\text{R}}{P(sx'), M, \{Q\}}\,\text{E}}{\cfrac{Q, M, \{\}}{\epsilon, M, \epsilon}\,\text{S}}\,\text{E}$$

Figure 6.6: Proof in the clausal connection calculus for translation.

Example 6.2. Consider Example 3.5 on page 31. For the matrices M and M', proofs in the connection calculi for translation are given in Figures 6.5 and 6.6.

6.4.2 Connection Proof Translation

A connection proof for a first-order formula M consists of a connection proof tree and a global substitution σ. Given this information, we want to construct a proof of $M \vdash \bot$, which is written in LK as $M \vdash$. We now show such a translation both from clausal and nonclausal proofs to LK.

We translate connection proof trees recursively by distinguishing the different rules of the calculus. We denote by $[\Gamma \vdash]$ the LK translation of the connection proof for Γ. In the LK translation, matrices and clauses represent formulas, where a matrix is a conjunction of clauses and a clause is a (potentially universally quantified) disjunction. We write that C is in M iff $M = C_1 \wedge \dots \wedge C_n$ with $C = C_i$ for some i with $1 \leq i \leq n$.

We use an LK rule $\wedge$L which extracts a conjunct from a conjunction, however keeping the conjunction in the context:

$$\cfrac{\Gamma, C_i, C_1 \wedge \dots \wedge C_n \vdash \Delta}{\Gamma, C_1 \wedge \dots \wedge C_n \vdash \Delta}\,\wedge\text{L}$$

Furthermore, we use an LK rule $\bot$L which derives $\bot$ from two complementary literals in the context:

$$\cfrac{}{\Gamma, A, \overline{A} \vdash}\,\bot\text{L}$$

Connection Calculus	LK
$\dfrac{\bigwedge_i \langle X_i, M, \{\}\rangle}{\varepsilon, M, \varepsilon}\,\mathrm{S}$ where $\{X_1, \dots, X_n\}$ is copy of $C \in M$	$\dfrac{\dfrac{\dfrac{[X_1, M, \{\} \vdash] \quad \dots \quad [X_n, M, \{\} \vdash]}{X_1 \lor \dots \lor X_n, M \vdash}\,\lor\mathrm{L}}{\forall \vec{x}.(X_1 \lor \dots \lor X_n), M \vdash}\,\forall\mathrm{L}}{M \vdash}\,\land\mathrm{L}$ where $\forall \vec{x}.(X_1 \lor \dots \lor X_n)$ in M
$\dfrac{}{L, M, Path \cup \{L'\}}\,\mathrm{R}$ where $\sigma(L) = \sigma(\overline{L'})$	$\dfrac{}{L, M, Path \cup \{L'\} \vdash}\,\bot\mathrm{L}$ where $L = \overline{L'}$

Figure 6.7: LK translation of common connection calculus rules.

$$\dfrac{\dfrac{\dfrac{\dfrac{[L_1, M, P \vdash] \quad \dots \quad \dfrac{}{\overline{L}, M, P \vdash}\,\bot\mathrm{L} \quad \dots \quad [L_n, M, P \vdash]}{L_1 \lor \dots \lor \overline{L} \lor \dots \lor L_n, M, P \vdash}\,\lor\mathrm{L}}{\forall \vec{x}.(L_1 \lor \dots \lor L_n), M, P \vdash}\,\forall\mathrm{L}}{M, P \vdash}\,\land\mathrm{L}}{L, M, Path \vdash}$$

where $\forall \vec{x}.(L_1 \lor \dots \lor L_n)$ in M and $P = Path \cup \{L\}$

Figure 6.8: LK translation of the clausal extension rule.

Let us start with the two rules that are translated the same way for clausal and nonclausal proofs, namely the start and the reduction rule. The translation of these rules is shown in Figure 6.7.

For the start rule, the translation obtains the formula corresponding to the clause C with the $\land$L rule, and instantiates it with the $\forall$L rule. The substitution σ is used to determine the instantiations, where fresh names are invented when a variable is unbound in the substitution. Then, the sequent is split into several subsequents $[X_i, M, \{\} \vdash]$, which represent the translations of the connection proofs for $\langle X_i, M, \{\}\rangle$.[1]

6.4.3 Clausal Proof Translation

The translation of the clausal extension rule (shown in Figure 6.3) is given in Figure 6.8. First, $L, M, Path \vdash$ is transformed to the equivalent $M, P \vdash$, where $P = Path \cup \{L\}$. Structurally, the remaining translation resembles that of the start rule, with the exception that it additionally closes a proof branch containing the negated literal $\overline{L}$.

[1] In the clausal setting, X_i could be written as L_i, but as the same rule is used in the nonclausal setting, where X_i can represent either a literal or a matrix, we write X_i for the common rules.

Connection Calculus	LK
$\dfrac{\bigwedge_i \langle X_i, M, Path\rangle}{M', M, Path}\,\mathrm{D}$ where $\{X_1, \dots, X_n\} \in M'$	$\dfrac{\dfrac{\dfrac{\left[X_1, \vec{M}', Path \vdash\right] \quad \dots \quad \left[X_n, \vec{M}', Path \vdash\right]}{X_1 \vee \dots \vee X_n, \vec{M}', Path \vdash}\,\vee\mathrm{L}}{\forall \vec{x}.(X_1 \vee \dots \vee X_n), \vec{M}', Path \vdash}\,\forall\mathrm{L}}{M', \vec{M}, Path \vdash}\,\wedge\mathrm{L}$ where $\forall \vec{x}.(X_1 \vee \dots \vee X_n)$ in M' and $\vec{M}' = \{M'\} \cup \vec{M}$

Figure 6.9: LK translation of the decomposition rule.

6.4.4 Nonclausal Proof Translation

We now proceed with the translation of nonclausal connection proofs, using the calculus introduced in Figure 6.4. The LK context in the translation of nonclausal proofs now has the shape $X, \vec{M}, Path$, where $\vec{M}$ is a set of matrices instead of a single matrix M as in the clausal case. During translation, $\vec{M}$ is extended such that for each word $\langle L, M, Path\rangle$ in the connection calculus and its corresponding sequent $L, \vec{M}, Path \vdash$ in LK, the e-clauses of M with respect to $Path \cup \{L\}$ are the clauses C for which C in M' and $M' \in \vec{M}$. We will see this in detail in the explanation for the extension rule.

The LK translation of nonclausal proofs reuses the translations of the start and the reduction rules given in Figure 6.7. However, occurrences of M in the LK translation are replaced by $\vec{M}$. The start rule uses $\vec{M} = \{M\}$, i.e. $\vec{M}$ contains only the initial problem matrix M.

The decomposition rule of the nonclausal calculus can be seen as a generalisation of the start rule. We give its translation to LK in Figure 6.9.

Let us now consider a nonclausal extension step applied to $\langle L, M, Path\rangle$. Let C_1 denote the e-clause of M with respect to $Path \cup \{L\}$ that was used for the extension step. By construction of $\vec{M}$ mentioned above, C_1 is some clause in $M_1 \in \vec{M}$. Furthermore, let β_1 be the β-clause of C_1 with respect to $\overline{L}$. Then we can find some m such that M_1, C_1 and β_1 can be written as in Figure 6.10.

The translation of the nonclausal extension rule is shown in Figure 6.11. We first transform $L, \vec{M}, Path \vdash$ to $\vec{M}^0, P \vdash$ which is equivalent due to $\vec{M}^0 = \vec{M}$ and $P = Path \cup \{L\}$. We then determine $M_1 \in \vec{M}$ and put it into the context by contraction (CL).

Now we recursively prove the sequent $M_i, \vec{M}^{i-1}, P \vdash$ as follows: If M_i is the literal $\overline{L}$, we prove the sequent $\overline{L}, \vec{M}^m, P \vdash$ with the $\perp$L rule. Otherwise, we proceed in the following way: First, we put the appropriate clause C_i of M_i that corresponds to β_i into the context with the $\wedge$L rule. In the same step, we merge M_i with $\vec{M}^{i-1}$, yielding $\vec{M}^i$. After the instantiation of C_i with the $\forall$L rule, the clause elements $X_{i,1}$ to X_{i,n_i} give rise to several proof branches where all but one are closed by translation of the proof branches of the connection proof. The one remaining clause element M_{i+1} gives rise to a

$$M_i = \begin{cases} \left[\cdots \overbrace{\begin{bmatrix} X_{i,1} \\ \vdots \\ M_{i+1} \\ \vdots \\ X_{i,n_i} \end{bmatrix}}^{C_i} \cdots \right] & \text{if } i \leq m \\ \overline{L} & \text{otherwise} \end{cases} \qquad \beta_i = \begin{cases} \begin{bmatrix} X_{i,1} \\ \vdots \\ [\beta_{i+1}] \\ \vdots \\ X_{i,n_i} \end{bmatrix} & \text{if } i \leq m \\ [] & \text{otherwise} \end{cases}$$

Figure 6.10: Definition of matrix M_i, clause C_i, and β-clause β_i.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\left[X_{m,1}, \vec{M}^m, P \vdash\right] \quad \cdots \quad \dfrac{}{\overline{L}, \vec{M}^m, P \vdash}\,\bot\mathrm{L} \quad \cdots \quad \left[X_{m,n_m}, \vec{M}^m, P \vdash\right]}{\vdots}\,\vee\mathrm{L}}{} \qquad \left[X_{1,1}, \vec{M}^1, P \vdash\right] \quad \cdots \quad \dfrac{}{M_2, \vec{M}^1, P \vdash}\,\wedge\mathrm{L} \quad \cdots \quad \left[X_{1,n_1}, \vec{M}^1, P \vdash\right]}{X_{1,1} \vee \cdots \vee X_{1,n_1}, \vec{M}^1, P \vdash}\,\vee\mathrm{L}}{\forall \vec{x}.(X_{1,1} \vee \cdots \vee X_{1,n_1}), \vec{M}^1, P \vdash}\,\forall\mathrm{L}}{M_1, \vec{M}^0, P \vdash}\,\wedge\mathrm{L}}{\dfrac{\vec{M}^0, P \vdash}{L, \vec{M}, Path \vdash}}\,\mathrm{CL}$$

Figure 6.11: LK translation of the nonclausal extension rule, where $\vec{M}^j = \vec{M} \cup \{M_i \mid 1 \leq i \leq j\}$ and $P = Path \cup \{L\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\neg P(s^2a), \vec{M}_2, \{Q, P(sa), P(s^2a)\} \vdash}\bot\text{L}}{\neg P(s^2a) \wedge (P(sa) \vee \neg Q), \vec{M}_2, \{Q, P(sa), P(s^2a)\} \vdash}\wedge\text{L}}{\vec{M}_2, \{Q, P(sa), P(s^2a)\} \vdash}\text{CL}}{\boxed{P(s^2a), \vec{M}_2, \{Q, P(sa)\} \vdash}} \qquad \dfrac{}{\boxed{\neg Q, \vec{M}_2, \{Q, P(sa)\} \vdash}}\bot\text{L}$$

$$\dfrac{\dfrac{}{\neg P(sa), \vec{M}_1, \{Q, P(sa)\} \vdash}\bot\text{L} \qquad \dfrac{\dfrac{\cdots}{P(s^2a) \vee \neg Q, \vec{M}_2, \{Q, P(sa)\} \vdash}\vee\text{L}}{\boxed{\neg P(s^3a) \wedge (P(s^2a) \vee \neg Q), \vec{M}_1, \{Q, P(sa)\} \vdash}}\wedge\text{L}}{\neg P(sa) \vee (\neg P(s^3a) \wedge (P(s^2a) \vee \neg Q)), \vec{M}_1, \{Q, P(sa)\} \vdash}\vee\text{L}$$

$$\dfrac{\dfrac{\dfrac{\neg P(sa) \vee (\neg P(s^3a) \wedge (P(s^2a) \vee \neg Q)), \vec{M}_1, \{Q, P(sa)\} \vdash}{\forall x. \neg P(x) \vee (\neg P(s^2x) \wedge (P(sx) \vee \neg Q)), \vec{M}_1, \{Q, P(sa)\} \vdash}\forall\text{L}}{M, \vec{M}_1, \{Q, P(sa)\} \vdash}\wedge\text{L}}{\vec{M}_1, \{Q, P(sa)\} \vdash}\text{CL}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P(a), \vec{M}_0, \{Q, \neg P(a)\} \vdash}\bot\text{L}}{M, \vec{M}_0, \{Q, \neg P(a)\} \vdash}\wedge\text{L}}{\vec{M}_0, \{Q, \neg P(a)\} \vdash}\text{CL}}{\boxed{\neg P(a), \vec{M}_0, \{Q\} \vdash}} \qquad \dfrac{\dfrac{\dfrac{\vec{M}_1, \{Q, P(sa)\} \vdash}{\boxed{P(sa), \vec{M}_1, \{Q\} \vdash}} \qquad \dfrac{}{\neg Q, \vec{M}_1, \{Q\} \vdash}\bot\text{L}}{P(sa) \vee \neg Q, \vec{M}_1, \{Q\} \vdash}\vee\text{L}}{\neg P(s^2a) \wedge (P(sa) \vee \neg Q), \vec{M}_0, \{Q\} \vdash}\wedge\text{L}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\neg P(a) \vee (\neg P(s^2a) \wedge (P(sa) \vee \neg Q)), \vec{M}_0, \{Q\} \vdash}\vee\text{L}}{\forall x. \neg P(x) \vee (\neg P(s^2x) \wedge (P(sx) \vee \neg Q)), \vec{M}_0, \{Q\} \vdash}\forall\text{L}}{M, \vec{M}_0, \{Q\} \vdash}\wedge\text{L}}{\vec{M}_0, \{Q\} \vdash}\text{CL}}{\boxed{Q, \vec{M}_0, \{\} \vdash}}}{\boxed{\vec{M}_0 \vdash}}\wedge\text{L}$$

Figure 6.12: Translation of a nonclausal proof to LK.

sequent $M_{i+1}, \vec{M}^i, P \vdash$, which we translate by recursion. This concludes the translation of the extension rule.

Example 6.3. Consider the nonclausal proof given in Figure 6.5. We show its translation to LK in Figure 6.12, where boxed sequents indicate words of the original proof. We use F from Example 3.5 to define

$$\begin{aligned}
\vec{M}_0 &= \{F\} \\
\vec{M}_1 &= \vec{M}_0 \cup \{\neg P(s^2a) \wedge (P(sa) \vee \neg Q)\} \\
\vec{M}_2 &= \vec{M}_1 \cup \{\neg P(s^3a) \wedge (P(s^2a) \vee \neg Q)\}
\end{aligned}$$

Listing 6.1: Example evaluation of a proof reconstruction method.

```
let conjecture = "m <= n ==> m..n = m INSERT (SUC m..n)" in
let premises = [NUMSEG_LREC; ADD1] in
MESON premises (parse_term conjecture)
```

Table 6.1: HOL Light evaluation datasets and number of contained problems.

HL-top	HL-msn	FS-top	FS-msn
2499	1119	27112	44468

6.5 Evaluation

We implemented proof search tactics for HOL Light based on leanCoP, nanoCoP, and Metis.[2] For evaluation, we compare their performance with the MESON tactic integrated into HOL Light. Similarly to [KUV15], we disable splitting for MESON. We use datasets derived from toplevel ("top") and MESON ("msn") goals from core HOL Light ("HL") and Flyspeck ("FS"), as described in Section 3.9. Statistics for these datasets are shown in Table 6.1.[3] We use the Git version 08f4461 of HOL Light from March 2017, running every tactic with a timeout of 10 seconds on each problem. We use the same hardware as in Section 3.9.

A reconstruction problem consists of a conjecture and a set of premises that were used to prove the conjecture. We evaluate a problem by feeding the conjecture and the premises to a proof reconstruction method in HOL Light, as shown in Listing 6.1. If the method can find and reconstruct a proof within a given time limit, the problem counts as proven.

The results are shown in Table 6.2: Metis solves the largest number of problems among all considered datasets. This may surprise, given that in Table 3.2 on the "FS-top" dataset, our functional implementation of leanCoP proves more problems than Metis. Apart from different preprocessing, this can be explained by different array access performance: Array access is more than 30 times faster in native OCaml programs compared to programs compiled in OCaml's toplevel (as used in HOL Light). This heavily disadvantages our connection provers in HOL Light, as fast unification via arrays is critical for their performance, see Section 3.5.

6.6 Related Work

Coq includes a proof certifying version of the intuitionistic first-order automated theorem prover JProver [SLKN01] and Matita includes a proof certifying version of an ordered paramodulation prover [AT07]. A translation of connection tableaux proofs to expansion

[2] The source code can be retrieved at http://cl-informatik.uibk.ac.at/users/mfaerber/tactics.html.

[3] Note that Table 6.1 mentions a larger number of MESON goals than Table 3.1. This is because we consider for this evaluation also those problems that are solved by the first-order export.

Table 6.2: Number of problems solved by various HOL Light tactics.

Prover	HL-top	HL-msn	FS-top	FS-msn
Metis	807	1029	4626	42829
MESON	736	900	4221	39227
leanCoP+cut	724	948	3714	39922
leanCoP−cut	717	844	3800	38528
nanoCoP+cut	538	802	2743	34213
nanoCoP−cut	550	811	2351	34769

trees which can be used for proof certification was studied in [Rei15]. The GAPT framework provides translations for a multitude of calculi and automated theorem provers [EHR+16]. An alternative approach to proof certification is the usage of verified automated theorem provers [RM05].

6.7 Conclusion

We developed proof search tactics in HOL Light based on the ATPs Metis, leanCoP, and nanoCoP. For this, we showed how to translate Metis proofs to HOL Light as well as clausal and nonclausal connection proofs to LK. The tactics can be used directly by users as proof search methods or as part of hammer systems to reconstruct proofs found by stronger ATPs.

Further experiments could be conducted with different translations from higher-order to first-order logic, as done for Isabelle [Bla12]. To ease the reconstruction of proofs from different first-order provers, it would be worthwhile to define a common first-order calculus that provers could translate their proofs to. This could be similar to the calculus of Metis, but more abstract at the same time, as the calculus of Metis assumes clausality and is closely linked to resolution. By providing a reconstruction method for proofs in this common first-order calculus to proof assistants based on higher-order logic, reconstruction of proofs from different ATPs could be simplified.

Chapter 7

Conclusion

In this thesis, we improved automatic proof search in proof assistants, focusing on premise selection, automated theorem proving, and proof reconstruction. The greatest amount of work in this thesis went into the improvement of automated theorem provers, which we discussed in three chapters. We now summarise our work:

- **Premise Selection**: We gave new formulations of machine learning techniques that were previously used to select premises, namely k-nearest neighbours and Naive Bayes. We then researched the usability of decision trees and random forests for premise selection. We evaluated existing online and offline approaches, finding that lack of incremental learning and forgetting of previously learnt data, respectively, made them unsuitable for premise selection. We therefore came up with two methods to efficiently update random forest classifiers with new knowledge. Furthermore, we evaluated different approaches to select samples and splitting features for the construction of decision trees. Finally, we combined decision trees with modified k-nearest neighbours to account for specificities of premise selection. We showed that our random forests approach improves upon k-nearest neighbours, both in terms of classification metrics such as AUC and Precision as well of number of problems solved by an ATP using the selected premises.

- **Connection Proof Search**: We showed several techniques to efficiently implement connection proof search in functional programming languages, which is important for integration into interactive theorem provers. We generalised a consistent Skolemisation method to nonclausal proof search, opening the possibility to do machine learning in nonclausal provers. Furthermore, we experimented with clause processing orders and developed a system to perform reproducible experiments with ATPs. We showed that functional versions of leanCoP and nanoCoP preserve the conciseness of their original Prolog implementations, while also considerably improving upon their performance: On one dataset, our functional implementation proves more than 50% more problems.

- **Internal Guidance**: We devised a method to guide ATPs based on the given-clause algorithm using positive and negative examples. For this, we generalised the occurrence of training samples to monoids, and adapted a Naive Bayes classifier to work with positive and negative examples. We then came up with a method to influence the selection of given clauses using the modified Naive Bayes classifier, and introduced several processing methods to match clauses to previously encountered

ones. Finally, we implemented the method in the higher-order ATP Satallax, where we could show a significant increase of problems proven.

- **Monte Carlo Proof Search**: We researched the usage of Monte Carlo Tree Search (MCTS) to control which part of a proof search tree should be expanded. To guide MCTS, we came up with several child probability and reward heuristics, some of which learn from previous proofs. Furthermore, we explored using MCTS as an advisor for a base prover. We implemented Monte Carlo Proof Search on top of our functional implementation of leanCoP. It turned out that MCTS as an advisor with non-learning heuristics for child probability and learning heuristics for reward performed best. Our MCTS-guided base prover solved more problems in the same time than the unguided base prover.
- **Proof Reconstruction**: We have defined translations from higher-order to first-order statements as well as from various first-order to higher-order proofs. In particular, we gave proof translations for the calculus of Metis and for nonclausal as well as clausal connection tableaux calculi, for which we defined new versions of the calculi suited towards proof reconstruction. This way, we integrated the first-order ATPs Metis, leanCoP, and nanoCoP into the higher-order ITP HOL Light. We showed that Metis is the method that most improves upon MESON, the other most powerful ATP tactic available in HOL Light.

7.1 Future Work

The ongoing advances in machine learning and its successful application to automated theorem proving suggest that more progress can be made in that direction. We give an overview of future work to be done.

7.1.1 Machine Learning

There are several problems in automated theorem proving that could benefit from machine learning methods to learn arbitrary functions on terms $\mathcal{T}$. For example:

1. How likely is a prover to finish a proof from a given prover state? ($\mathcal{T} \to \mathbb{R}$)
2. What are the features of a given conjecture? ($\mathcal{T} \to \mathbb{R}^n$)
3. What are likely intermediate proof goals for a given conjecture? ($\mathcal{T} \to \mathcal{T}^n$)
4. What statements are likely true and useful given a set of definitions? ($2^{\mathcal{T}} \to \mathcal{T}^n$)

Point 1 corresponds to reward heuristics as shown in Section 5.5. Point 2 corresponds to term embeddings, which are a precondition for many premise selection methods shown in Chapter 2. Points 3 and 4 correspond to conjecturing, which is likely necessary if computers are ever to prove e.g. Kepler's conjecture only from definitions. Conjecturing was realised for example in the Hipster system [JRSC14].

Neural networks were successfully employed to learn term embeddings [WTWD17]. This suggests the extension of this approach to reward heuristics. However, the approach is not directly applicable to conjecturing, as conjecturing requires the generation of terms.

This suggests the research of more general machine learning techniques able to generate meaningful arbitrary-sized terms. Datasets such as HolStep provide a rich resource to evaluate such approaches [KCS17].

A substantial restriction of all methods presented in this thesis is their dependency on consistent symbol names. That is, knowledge learnt about a symbol "+" is not automatically transferred to a symbol "plus", even if the two symbols occurred in similar contexts. That means that knowledge learnt on one formalisation cannot be easily transferred to another one. To remedy this, it seems useful to think of methods to identify symbols by their context instead of their names, similarly to existing alignment techniques [GK19].

The formalisation of machine learning methods has received little attention so far, with the exception of [Ben16]. Premise selection methods, such as k-nearest neighbours [ZZ05] or Naive Bayes could be formalised, as [DGL97] shows interesting properties about these methods. A formalisation will increase the confidence in aforementioned methods as well as advance the infrastructure to formalise probability-based machine learning methods.

7.1.2 Automated Theorem Proving

In our opinion, nonclausal proof search offers exciting possibilities. Operating on a formula much closer to the original than its clausal form provides more information about the context in which terms appear. Using such information might allow internal guidance methods to more effectively steer proof search than in the clausal setting.

To further deepen our understanding of nonclausal proof search and to potentially also simplify and speed up the provers, mechanically checkable proofs of their properties would be useful. Such a formalisation could be carried out similarly to previous projects, such as the formalisation of a resolution prover [SBTW18].

If a nonclausal prover can be made to outperform its clausal counterpart, one might adapt other calculi and provers to the nonclausal setting, for example the superposition calculus underlying the provers Vampire and E. This would pave the road towards a nonclausal state-of-the-art theorem prover.

Much of our research in the intersection of machine learning and theorem proving has greatly benefited from connection provers such as leanCoP. Their compactness and flexibility allowed for various integrations of machine learning techniques into proof search, while preserving soundness. The functional versions of leanCoP and nanoCoP developed as part of this thesis provide a stable and performant basis for future experiments. We believe that connection provers are natural candidates for future research on learning proof search.

Bibliography

[ABD08] Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors. *IJCAR*, volume 5195 of *LNCS*. Springer, 2008. doi:10.1007/978-3-540-71070-7.

[AGPV13] Rahul Agrawal, Archit Gupta, Yashoteja Prabhu, and Manik Varma. Multi-label learning with millions of labels: recommending advertiser bid phrases for web pages. In Daniel Schwabe, Virgílio A. F. Almeida, Hartmut Glaser, Ricardo A. Baeza-Yates, and Sue B. Moon, editors, *WWW*, pages 13–24. ACM, 2013. doi:10.1145/2488388.2488391.

[AHK+14] Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. Premise selection for mathematics by corpus analysis and kernel methods. *J. Autom. Reasoning*, 52(2):191–213, 2014. doi:10.1007/s10817-013-9286-5.

[AKU12] Jesse Alama, Daniel Kühlwein, and Josef Urban. Automated and human proofs in general mathematics: An initial comparison. In Nikolaj Bjørner and Andrei Voronkov, editors, *LPAR-18*, volume 7180 of *LNCS*, pages 37–45. Springer, 2012. doi:10.1007/978-3-642-28717-6_6.

[And89] Peter B. Andrews. On connections and higher-order logic. *J. Autom. Reasoning*, 5(3):257–291, 1989. doi:10.1007/BF00248320.

[AS92] Owen L. Astrachan and Mark E. Stickel. Caching and lemmaizing in model elimination theorem provers. In Deepak Kapur, editor, *CADE-11*, volume 607 of *LNCS*, pages 224–238. Springer, 1992. doi:10.1007/3-540-55602-8_168.

[AT07] Andrea Asperti and Enrico Tassi. Higher order proof reconstruction from paramodulation-based refutations: The unit equality case. In Manuel Kauers, Manfred Kerber, Robert Miner, and Wolfgang Windsteiger, editors, *MKM*, volume 4573 of *LNCS*, pages 146–160. Springer, 2007. doi:10.1007/978-3-540-73086-6_14.

[BB11] Julian Backes and Chad E. Brown. Analytic tableaux for higher-order logic with choice. *J. Autom. Reasoning*, 47(4):451–479, 2011. doi:10.1007/s10817-011-9233-2.

[BBP11] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT solvers. In Bjørner and Sofronie-Stokkermans [BS11], pages 116–130. doi:10.1007/978-3-642-22438-6_11.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN 978-3-642-05880-6. doi:10.1007/978-3-662-07964-5.

[BDKV14] Armin Biere, Ioan Dragan, Laura Kovács, and Andrei Voronkov. Experimenting with SAT solvers in Vampire. In Alexander F. Gelbukh, Félix Castro-Espinoza, and Sofía N. Galicia-Haro, editors, *MICAI 2014. Part I*, volume 8856 of *LNCS*, pages 431–442. Springer, 2014. doi:10.1007/978-3-319-13647-9_39.

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda - A functional language with dependent types. In Berghofer et al. [BNUW09], pages 73–78. doi:10.1007/978-3-642-03359-9_6.

[Ben16] Alexander Bentkamp. An Isabelle formalization of the expressiveness of deep learning. Master's thesis, Universität des Saarlandes, 2016. URL http://matryoshka.gforge.inria.fr/pubs/bentkamp_msc_thesis.pdf.

[Ber08] Yves Bertot. A short presentation of Coq. In Mohamed et al. [MMT08], pages 12–16. doi:10.1007/978-3-540-71067-7_3.

[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4(3):217–247, 1994. doi:10.1093/logcom/4.3.217.

[BGK+16] Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. A learning-based fact selector for Isabelle/HOL. *J. Autom. Reasoning*, 57(3):219–244, 2016. doi:10.1007/s10817-016-9362-8.

[BHS93] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. The even more liberalized δ-rule in free variable semantic tableaux. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Kurt Gödel Colloquium*, volume 713 of *LNCS*, pages 108–119. Springer, 1993. doi:10.1007/BFb0022559.

[Bib83] Wolfgang Bibel. Matings in matrices. *Commun. ACM*, 26(11):844–852, 1983. doi:10.1145/182.183.

[Bib91] Wolfgang Bibel. Perspectives on automated deduction. In Robert S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Automated Reasoning Series, pages 77–104. Kluwer Academic Publishers, 1991.

[Bib17] Wolfgang Bibel. A vision for automated deduction rooted in the connection method. In Renate A. Schmidt and Cláudia Nalon, editors, *TABLEAUX*, volume 10501 of *LNCS*, pages 3–21. Springer, 2017. doi:10.1007/978-3-319-66902-1_1.

[Bie08] Armin Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.

[Bla12] Jasmin Christian Blanchette. *Automatic proofs and refutations for higher-order logic*. PhD thesis, Technical University Munich, 2012. URL http://nbn-resolving.de/urn:nbn:de:bvb:91-diss-20120628-1097834-1-6.

[BM98] Robert S. Boyer and J Strother Moore. *A computational logic handbook*. Academic Press international series in formal methods. Academic Press, 2nd edition, 1998. ISBN 978-0-12-122955-9.

[BM11] Kai Brünnler and George Metcalfe, editors. *TABLEAUX*, volume 6793 of *LNCS*. Springer, 2011. doi:10.1007/978-3-642-22119-4.

[BNUW09] Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors. *TPHOLs*, volume 5674 of *LNCS*. Springer, 2009. doi:10.1007/978-3-642-03359-9.

[BP95] Bernhard Beckert and Joachim Posegga. leanTAP: Lean tableau-based deduction. *J. Autom. Reasoning*, 15(3):339–358, 1995. doi:10.1007/BF00881804.

[BPW+12] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):1–43, 2012. doi:10.1109/TCIAIG.2012.2186810.

[Bre96] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996. doi:10.1007/BF00058655.

[Bre01] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001. doi:10.1023/A:1010933404324.

[Bro12] Chad E. Brown. Satallax: An automatic higher-order prover. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR*, volume 7364 of *LNCS*, pages 111–117. Springer, 2012. doi:10.1007/978-3-642-31365-3_11.

[Bro13] Chad E. Brown. Reducing higher-order theorem proving to a sequence of SAT problems. *J. Autom. Reasoning*, 51(1):57–77, 2013. doi:10.1007/s10817-013-9283-8.

[BS11] Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors. *CADE-23*, volume 6803 of *LNCS*. Springer, 2011. doi:10.1007/978-3-642-22438-6.

[Bun94] Alan Bundy, editor. *CADE-12*, volume 814 of *LNCS*. Springer, 1994. doi:10.1007/3-540-58156-1.

[CCRR99] Andrew J. Carlson, Chad M. Cumby, Jeff L. Rosen, and Dan Roth. SNoW user guide. Technical Report UIUCDCS-R-99-2101, University of Illinois at Urbana-Champaign, May 1999. URL http://cogcomp.org/papers/CCRR99.pdf.

[CH67] Thomas M. Cover and Peter E. Hart. Nearest neighbor pattern classification. *IEEE Trans. Information Theory*, 13(1):21–27, 1967. doi:10.1109/TIT.1967.1053964.

[CN06] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In William W. Cohen and Andrew Moore, editors, *ICML*, volume 148 of *ACM*, pages 161–168. ACM, 2006. doi:10.1145/1143844.1143865.

[CS14] Scott Chacon and Ben Straub. *Pro Git*. Apress, 2nd edition, 2014. ISBN 978-1-4842-0077-3. URL https://git-scm.com/book/en/v2.

[DFGS99] Jörg Denzinger, Matthias Fuchs, Christoph Goller, and Stephan Schulz. Learning from Previous Proof Experience. Technical Report AR99-4, Institut für Informatik, Technische Universität München, 1999.

[DGL97] Luc Devroye, László Györfi, and Gábor Lugosi. *A Probabilistic Theory of Pattern Recognition*, volume 31 of *Applications of Mathematics*. Springer, corrected 2nd edition, 1997.

[DKS97] Jörg Denzinger, Martin Kronenburg, and Stephan Schulz. DISCOUNT - A distributed and learning equational prover. *J. Autom. Reasoning*, 18(2):189–198, 1997. doi:10.1023/A:1005879229581.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

[EHR+16] Gabriel Ebner, Stefan Hetzl, Giselle Reis, Martin Riener, Simon Wolfsteiner, and Sebastian Zivota. System description: GAPT 2.0. In Olivetti and Tiwari [OT16], pages 293–301. doi:10.1007/978-3-319-40229-1_20.

[ES03] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003. doi:10.1007/978-3-540-24605-3_37.

[FB16] Michael Färber and Chad E. Brown. Internal guidance for Satallax. In Olivetti and Tiwari [OT16], pages 349–361. doi:10.1007/978-3-319-40229-1_24.

[FK15a] Michael Färber and Cezary Kaliszyk. Metis-based paramodulation tactic for HOL Light. In Gottlob et al. [GSV15], pages 127–136. doi:10.29007/z9mz.

[FK15b] Michael Färber and Cezary Kaliszyk. Random forests for premise selection. In Carsten Lutz and Silvio Ranise, editors, *FroCoS*, volume 9322 of *LNCS*, pages 325–340. Springer, 2015. doi:10.1007/978-3-319-24246-0_20.

[FKU17] Michael Färber, Cezary Kaliszyk, and Josef Urban. Monte Carlo tableau proof search. In Leonardo de Moura, editor, *CADE-26*, volume 10395 of *LNCS*, pages 563–579. Springer, 2017. doi:10.1007/978-3-319-63046-5_34.

[FKU18] Michael Färber, Cezary Kaliszyk, and Josef Urban. Machine learning guidance and proof certification for connection tableaux. *CoRR*, abs/1805.03107, May 2018. URL http://arxiv.org/abs/1805.03107.

[GA99] Martin Giese and Wolfgang Ahrendt. Hilbert's epsilon-terms in automated theorem proving. In Neil V. Murray, editor, *TABLEAUX*, volume 1617 of *LNCS*, pages 171–185. Springer, 1999. doi:10.1007/3-540-48754-9_17.

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *ITP*, volume 7998 of *LNCS*, pages 163–179. Springer, 2013. doi:10.1007/978-3-642-39634-2_14.

[Gal00] Didier Galmiche. Connection methods in linear logic and proof nets construction. *Theor. Comput. Sci.*, 232(1-2):231–272, 2000. doi:10.1016/S0304-3975(99)00176-0.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. I. *Mathematische Zeitschrift*, 39(1):176–210, 1935. ISSN 1432-1823. doi:10.1007/BF01201353.

[GK19] Thibault Gauthier and Cezary Kaliszyk. Aligning concepts across proof assistant libraries. *J. Symb. Comput.*, 90:89–123, 2019. doi:10.1016/j.jsc.2018.04.005.

[Gor00] Mike Gordon. From LCF to HOL: a short history. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 169–186. The MIT Press, 2000.

[Gra13] Stéphane Graham-Lengrand. Psyche: A proof-search engine based on sequent calculus with an LCF-style architecture. In Didier Galmiche and Dominique Larchey-Wendling, editors, *TABLEAUX*, volume 8123 of *LNCS*, pages 149–156. Springer, 2013. doi:10.1007/978-3-642-40537-2_14.

[Gre86] Steven Greenbaum. *Input transformations and resolution implementation techniques for theorem-proving in first-order logic.* PhD thesis, University of Illinois at Urbana-Champaign, 1986.

[GS07] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In Zoubin Ghahramani, editor, *ICML*, volume 227 of *ACM*, pages 273–280. ACM, 2007. doi:10.1145/1273496.1273531.

[GSV15] Georg Gottlob, Geoff Sutcliffe, and Andrei Voronkov, editors. *GCAI*, volume 36 of *EPiC Series in Computing*. EasyChair, 2015.

[HAB+17] Thomas C. Hales, Mark Adams, Gertrud Bauer, Dat Tat Dang, John Harrison, Truong Le Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Thang Tat Nguyen, Truong Quang Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, An Hoai Thi Ta, Trung Nam Tran, Diep Thi Trieu, Josef Urban, Ky Khac Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5, 2017. doi:10.1017/fmp.2017.1.

[Häh01] Reiner Hähnle. Tableaux and related methods. In Robinson and Voronkov [RV01], pages 100–178. ISBN 0-444-50813-9.

[Har96] John Harrison. Optimizing proof search in model elimination. In Michael A. McRobbie and John K. Slaney, editors, *CADE-13*, volume 1104 of *LNCS*, pages 313–327. Springer, 1996. doi:10.1007/3-540-61511-3_97.

[Har09] John Harrison. HOL Light: An overview. In Berghofer et al. [BNUW09], pages 60–66. doi:10.1007/978-3-642-03359-9_4.

[HB39] David Hilbert and Paul Bernays. *Grundlagen der Mathematik. II*, volume 50 of *Die Grundlehren der mathematischen Wissenschaften*. Springer, 1939. ISBN 978-3-540-05110-7.

[Hin82] Jaakko Hintikka. Game-theoretical semantics: insights and prospects. *Notre Dame Journal of Formal Logic*, 23(2):219–241, 1982. doi:10.1305/ndjfl/1093883627.

[HRSV16] Kryštof Hoder, Giles Reger, Martin Suda, and Andrei Voronkov. Selecting the selection. In Olivetti and Tiwari [OT16], pages 313–329. doi:10.1007/978-3-319-40229-1_22.

[Hur03] Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA)*, number NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, September 2003. URL http://www.gilith.com/research/papers.

[HV11] Kryštof Hoder and Andrei Voronkov. Sine qua non for large theory reasoning. In Bjørner and Sofronie-Stokkermans [BS11], pages 299–314. doi:10.1007/978-3-642-22438-6_23.

[ISA+16] Geoffrey Irving, Christian Szegedy, Alexander A. Alemi, Niklas Eén, François Chollet, and Josef Urban. DeepMath - deep sequence models for premise selection. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors,

NIPS, pages 2235–2243, 2016. URL http://papers.nips.cc/paper/6280-deepmath-deep-sequence-models-for-premise-selection.

[Jon73] Karen Spärck Jones. Index term weighting. *Information Storage and Retrieval*, 9(11):619–633, 1973. doi:10.1016/0020-0271(73)90043-0.

[JRSC14] Moa Johansson, Dan Rosén, Nicholas Smallbone, and Koen Claessen. Hipster: Integrating theory exploration in a proof assistant. In Stephen M. Watt, James H. Davenport, Alan P. Sexton, Petr Sojka, and Josef Urban, editors, *CICM*, volume 8543 of *LNCS*, pages 108–122. Springer, 2014. doi:10.1007/978-3-319-08434-3_9.

[JU17] Jan Jakubův and Josef Urban. ENIGMA: efficient learning-based inference guiding machine. In Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke, editors, *CICM*, volume 10383 of *LNCS*, pages 292–302. Springer, 2017. doi:10.1007/978-3-319-62075-6_20.

[KAE+10] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010. doi:10.1145/1743546.1743574.

[KBKU13] Daniel Kühlwein, Jasmin Christian Blanchette, Cezary Kaliszyk, and Josef Urban. MaSh: Machine learning for Sledgehammer. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *ITP*, volume 7998 of *LNCS*, pages 35–50. Springer, 2013. doi:10.1007/978-3-642-39634-2_6.

[KCS17] Cezary Kaliszyk, François Chollet, and Christian Szegedy. HolStep: A machine learning dataset for higher-order logic theorem proving. *CoRR*, abs/1703.00426, 2017. URL http://arxiv.org/abs/1703.00426.

[Kep11] Johannes Kepler. *Strena seu de nive sexangula.* Godefridum Tambach, Francofurti ad Moenum, 1611. ISBN 978-1-58988-053-5.

[KMNO14] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *POPL*, pages 179–192. ACM, 2014. doi:10.1145/2535838.2535841.

[Kor13] Konstantin Korovin. Inst-Gen - A modular approach to instantiation-based automated reasoning. In Andrei Voronkov and Christoph Weidenbach, editors, *Programming Logics - Essays in Memory of Harald Ganzinger*, volume 7797 of *LNCS*, pages 239–270. Springer, 2013. doi:10.1007/978-3-642-37651-1_10.

[KS06] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *ECML*, volume 4212 of *LNCS*, pages 282–293. Springer, 2006. doi:10.1007/11871842_29.

[KSUV15] Cezary Kaliszyk, Stephan Schulz, Josef Urban, and Jiří Vyskočil. System description: E.T. 0.1. In Amy P. Felty and Aart Middeldorp, editors, *CADE-25*, volume 9195 of *LNCS*, pages 389–398. Springer, 2015. doi:10.1007/978-3-319-21401-6_27.

[KU14] Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with Flyspeck. *J. Autom. Reasoning*, 53(2):173–213, 2014. doi:10.1007/s10817-014-9303-3.

[KU15a] Cezary Kaliszyk and Josef Urban. FEMaLeCoP: Fairly efficient machine learning connection prover. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *LPAR-20*, volume 9450 of *LNCS*, pages 88–96. Springer, 2015. doi:10.1007/978-3-662-48899-7_7.

[KU15b] Cezary Kaliszyk and Josef Urban. HOL(y)Hammer: Online ATP service for HOL Light. *Mathematics in Computer Science*, 9(1):5–22, 2015. doi:10.1007/s11786-014-0182-0.

[KU15c] Cezary Kaliszyk and Josef Urban. Learning-assisted theorem proving with millions of lemmas. *J. Symb. Comput.*, 69:109–128, 2015. doi:10.1016/j.jsc.2014.09.032.

[KU15d] Cezary Kaliszyk and Josef Urban. MizAR 40 for Mizar 40. *J. Autom. Reasoning*, 55(3):245–256, 2015. doi:10.1007/s10817-015-9330-8.

[KUV15] Cezary Kaliszyk, Josef Urban, and Jiří Vyskočil. Certified connection tableaux proofs for HOL Light and TPTP. In Xavier Leroy and Alwen Tiu, editors, *CPP*, pages 59–66. ACM, 2015. doi:10.1145/2676724.2693176.

[KV13] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *CAV*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013. doi:10.1007/978-3-642-39799-8_1.

[Kü14] Daniel A. Kühlwein. *Machine Learning for Automated Reasoning*. PhD thesis, Radboud Universiteit Nijmegen, 2014.

[Lan30] Edmund Landau. *Grundlagen der Analysis*. Akademische Verlagsgesellschaft, Leipzig, 1930. URL http://www.cs.ru.nl/~freek/aut/grundlagen-1.0.tar.gz.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 42–54. ACM, 2006. doi:10.1145/1111037.1111042.

[LISK17] Sarah M. Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. In Thomas Eiter and David Sands, editors, *LPAR-21*, volume 46 of *EPiC Series in Computing*, pages 85–105. EasyChair, 2017. URL http://www.easychair.org/publications/paper/340345.

[Lov68] Donald W. Loveland. Mechanical theorem-proving by model elimination. *J. ACM*, 15(2):236–251, 1968. doi:10.1145/321450.321456.

[Lov16] Donald W. Loveland. Mark Stickel: His earliest work. *J. Autom. Reasoning*, 56(2):99–112, 2016. doi:10.1007/s10817-015-9342-4.

[LRT14] Balaji Lakshminarayanan, Daniel M. Roy, and Yee Whye Teh. Mondrian forests: Efficient online random forests. In Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger, editors, *NIPS*, pages 3140–3148, 2014. URL http://papers.nips.cc/paper/5234-mondrian-forests-efficient-online-random-forests.

[LS73] Donald W. Loveland and Mark E. Stickel. A hole in goal trees: Some guidance from resolution theory. In Nils J. Nilsson, editor, *IJCAI*, pages 153–161. William Kaufmann, 1973. URL http://ijcai.org/Proceedings/73/Papers/018.pdf.

[LS01] Reinhold Letz and Gernot Stenz. Model elimination and connection tableau procedures. In Robinson and Voronkov [RV01], pages 2015–2114. ISBN 0-444-50813-9.

[LSBB92] Reinhold Letz, Johann Schumann, Stefan Bayerl, and Wolfgang Bibel. SETHEO: A high-performance theorem prover. *J. Autom. Reasoning*, 8(2):183–212, 1992. doi:10.1007/BF00244282.

[McC97] William McCune. Solution of the Robbins problem. *J. Autom. Reasoning*, 19(3):263–276, 1997. doi:10.1023/A:1005843212881.

[McC03] William McCune. OTTER 3.3 reference manual. Technical Report ANL/MCS-TM-263, Argonne National Laboratory, October 2003.

[Meg07] Norman D. Megill. *Metamath: A Computer Language for Pure Mathematics*. Lulu Press, 2007. ISBN 978-1-4116-3724-5. URL http://us.metamath.org/downloads/metamath.pdf.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory*, volume 1 of *Studies in Proof Theory. Lecture Notes*. Bibliopolis, Naples, 1984. ISBN 88-7088-105-9. Notes by Giovanni Sambin.

[MMT08] Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors. *TPHOLs*, volume 5170 of *LNCS*. Springer, 2008. doi:10.1007/978-3-540-71067-7.

[MP08] Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *J. Autom. Reasoning*, 40(1):35–60, 2008. doi:10.1007/s10817-007-9085-y.

[MP09] Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *J. Applied Logic*, 7(1):41–57, 2009. doi:10.1016/j.jal.2007.07.004.

[MR05] Roman Matuszewski and Piotr Rudnicki. Mizar: the first 30 years. *Mechanized Mathematics and Its Applications*, 4(1), 2005. URL http://mizar.org/people/romat/MatRud2005.pdf.

[NK09] Adam Naumowicz and Artur Korniłowicz. A brief overview of Mizar. In Berghofer et al. [BNUW09], pages 67–72. doi:10.1007/978-3-642-03359-9_5.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. ISBN 3-540-43376-7. doi:10.1007/3-540-45949-9.

[OB03] Jens Otten and Wolfgang Bibel. leanCoP: lean connection-based theorem proving. *J. Symb. Comput.*, 36(1-2):139–161, 2003. doi:10.1016/S0747-7171(03)00037-3.

[OR01] Nikunj C. Oza and Stuart J. Russell. Online bagging and boosting. In Thomas S. Richardson and Tommi S. Jaakkola, editors, *AISTATS*. Society for Artificial Intelligence and Statistics, 2001. URL http://www.gatsby.ucl.ac.uk/aistats/aistats2001/files/oza149.ps.

[OT16] Nicola Olivetti and Ashish Tiwari, editors. *IJCAR*, volume 9706 of *LNCS*. Springer, 2016. doi:10.1007/978-3-319-40229-1.

[Ott05] Jens Otten. Clausal connection-based theorem proving in intuitionistic first-order logic. In Bernhard Beckert, editor, *TABLEAUX*, volume 3702 of *LNCS*, pages 245–261. Springer, 2005. doi:10.1007/11554554_19.

[Ott08] Jens Otten. leanCoP 2.0 and ileanCoP 1.2: High performance lean theorem proving in classical and intuitionistic logic (system descriptions). In Armando et al. [ABD08], pages 283–291. doi:10.1007/978-3-540-71070-7_23.

[Ott10] Jens Otten. Restricting backtracking in connection calculi. *AI Commun.*, 23(2-3):159–182, 2010. doi:10.3233/AIC-2010-0464.

[Ott11] Jens Otten. A non-clausal connection calculus. In Brünnler and Metcalfe [BM11], pages 226–241. doi:10.1007/978-3-642-22119-4_18.

[Ott14] Jens Otten. Mleancop: A connection prover for first-order modal logic. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *IJCAR*, volume 8562 of *LNCS*, pages 269–276. Springer, 2014. doi:10.1007/978-3-319-08587-6_20.

[Ott16] Jens Otten. nanoCoP: A non-clausal connection prover. In Olivetti and Tiwari [OT16], pages 302–312. doi:10.1007/978-3-319-40229-1_21.

[Pau88] Lawrence C. Paulson. Isabelle: The next seven hundred theorem provers. In Ewing L. Lusk and Ross A. Overbeek, editors, *CADE-9*, volume 310 of *LNCS*, pages 772–773. Springer, 1988. doi:10.1007/BFb0012891.

[Pau99] Lawrence C. Paulson. A generic tableau prover and its integration with Isabelle. *J. UCS*, 5(3):73–87, 1999. doi:10.3217/jucs-005-03-0073.

[Pau10] Lawrence C. Paulson. Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In Renate A. Schmidt, Stephan Schulz, and Boris Konev, editors, *PAAR*, volume 9 of *EPiC Series in Computing*, pages 1–10. EasyChair, 2010. URL http://www.easychair.org/publications/paper/52675.

[PG86] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986. doi:10.1016/S0747-7171(86)80028-1.

[PKB07] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm Intelligence*, 1(1):33–57, 2007. doi:10.1007/s11721-007-0002-0.

[Pla90] David A. Plaisted. A sequent-style model elimination strategy and a positive refinement. *J. Autom. Reasoning*, 6(4):389–402, 1990. doi:10.1007/BF00244355.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975. doi:10.1016/0304-3975(75)90017-1.

[PS07] Lawrence C. Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In Klaus Schneider and Jens Brandt, editors, *TPHOLs*, volume 4732 of *LNCS*, pages 232–245. Springer, 2007. doi:10.1007/978-3-540-74591-4_18.

[QED94] QED. The QED manifesto. In Bundy [Bun94], pages 238–251. doi:10.1007/3-540-58156-1_17.

[Rei15] Giselle Reis. Importing SMT and connection proofs as expansion trees. In Cezary Kaliszyk and Andrei Paskevich, editors, *PxTP*, volume 186 of *EPTCS*, pages 3–10, 2015. doi:10.4204/EPTCS.186.3.

[RM05] Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *LNCS*, pages 294–309. Springer, 2005. doi:10.1007/11541868_19.

[RO08] Thomas Raths and Jens Otten. randoCoP: Randomizing the proof search order in the connection calculus. In Boris Konev, Renate A. Schmidt, and Stephan Schulz, editors, *PAAR*, volume 373 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008. URL http://ceur-ws.org/Vol-373/paper-08.pdf.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965. doi:10.1145/321250.321253.

[ROK07] Thomas Raths, Jens Otten, and Christoph Kreitz. The ILTP problem library for intuitionistic logic. *J. Autom. Reasoning*, 38(1-3):261–271, 2007. doi:10.1007/s10817-006-9060-z.

[Ros11] Christopher D. Rosin. Nested rollout policy adaptation for Monte Carlo tree search. In Toby Walsh, editor, *IJCAI*, pages 649–654. IJCAI/AAAI, 2011. doi:10.5591/978-1-57735-516-8/IJCAI11-115.

[RS04] Laura Elena Raileanu and Kilian Stoffel. Theoretical comparison between the Gini index and information gain criteria. *Ann. Math. Artif. Intell.*, 41(1):77–93, 2004. doi:10.1023/B:AMAI.0000018580.96245.c6.

[RSV01] I. V. Ramakrishnan, R. C. Sekar, and Andrei Voronkov. Term indexing. In Robinson and Voronkov [RV01], pages 1853–1964. ISBN 0-444-50813-9.

[RV01] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001. ISBN 0-444-50813-9.

[SB10] Geoff Sutcliffe and Christoph Benzmüller. Automated reasoning in higher-order logic using the TPTP THF infrastructure. *J. Formalized Reasoning*, 3(1):1–27, 2010. doi:10.6092/issn.1972-5787/1710.

[SBP13] Nik Sultana, Jasmin Christian Blanchette, and Lawrence C. Paulson. LEO-II and Satallax on the Sledgehammer test bench. *J. Applied Logic*, 11(1):91–102, 2013. doi:10.1016/j.jal.2012.12.002.

[SBTW18] Anders Schlichtkrull, Jasmin Christian Blanchette, Dmitriy Traytel, and Uwe Waldmann. Formalization of Bachmair and Ganzinger's ordered resolution prover. *Archive of Formal Proofs*, 2018, 2018. URL https://www.isa-afp.org/entries/Ordered_Resolution_Prover.html.

[Sch94] Johann Schumann. DELTA - A bottom-up preprocessor for top-down theorem provers - system abstract. In Bundy [Bun94], pages 774–777. doi:10.1007/3-540-58156-1_58.

[Sch00] Stephan Schulz. *Learning search control knowledge for equational deduction*, volume 230 of *DISKI*. Infix Akademische Verlagsgesellschaft, 2000. ISBN 978-3-89838-230-4. URL http://d-nb.info/95899126X.

[Sch01] Stephan Schulz. Learning search control knowledge for equational theorem proving. In Franz Baader, Gerhard Brewka, and Thomas Eiter, editors, *KI*, volume 2174 of *LNCS*, pages 320–334. Springer, 2001. doi:10.1007/3-540-45422-5_23.

[Sch13] Stephan Schulz. System description: E 1.8. In Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *LPAR-19*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013. doi:10.1007/978-3-642-45221-5_49.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. doi:10.1038/nature16961.

[SLKN01] Stephan Schmitt, Lori Lorigo, Christoph Kreitz, and Aleksey Nogin. JProver: Integrating connection-based theorem proving into interactive proof assistants. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *IJCAR*, volume 2083 of *LNCS*, pages 421–426. Springer, 2001. doi:10.1007/3-540-45744-5_34.

[SLS+09] Amir Saffari, Christian Leistner, Jakob Santner, Martin Godec, and Horst Bischof. On-line random forests. In *ICCV Workshops*, pages 1393–1400, 2009. doi:10.1109/ICCVW.2009.5457447.

[SMS16] Richard M. Stallman, Roland McGrath, and Paul D. Smith. *GNU Make*. Free Software Foundation, 2016. URL https://www.gnu.org/software/make/manual/.

[SN08] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Mohamed et al. [MMT08], pages 28–32. doi:10.1007/978-3-540-71067-7_6.

[Sti88] Mark E. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *J. Autom. Reasoning*, 4(4):353–380, 1988. doi:10.1007/BF00297245.

[Sut09a] Geoff Sutcliffe. The 4th IJCAR automated theorem proving system competition - CASC-J4. *AI Commun.*, 22(1):59–72, 2009. doi:10.3233/AIC-2009-0441.

[Sut09b] Geoff Sutcliffe. The TPTP problem library and associated infrastructure. *J. Autom. Reasoning*, 43(4):337–362, 2009. doi:10.1007/s10817-009-9143-8.

[Sut11] Geoff Sutcliffe. The 5th IJCAR automated theorem proving system competition - CASC-J5. *AI Commun.*, 24(1):75–89, 2011. doi:10.3233/AIC-2010-0483.

[Sut16a] Geoff Sutcliffe. The 8th IJCAR automated theorem proving system competition - CASC-J8. *AI Commun.*, 29(5):607–619, 2016. doi:10.3233/AIC-160709.

[Sut16b] Geoff Sutcliffe. The CADE ATP system competition - CASC. *AI Magazine*, 37(2):99–101, 2016. URL http://www.aaai.org/ojs/index.php/aimagazine/article/view/2620.

[SWTU12] Maarten P. D. Schadd, Mark H. M. Winands, Mandy J. W. Tak, and Jos W. H. M. Uiterwijk. Single-player Monte-Carlo tree search for SameGame. *Knowl.-Based Syst.*, 34:3–11, 2012. doi:10.1016/j.knosys.2011.08.008.

[TK07] Grigorios Tsoumakas and Ioannis Katakis. Multi-label classification: An overview. *IJDWM*, 3(3):1–13, 2007. doi:10.4018/jdwm.2007070101.

[Tse83] Gregory S. Tseitin. On the complexity of derivation in propositional calculus. In Jörg H. Siekmann and Graham Wrightson, editors, *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 466–483. Springer, 1983. ISBN 978-3-642-81955-1. doi:10.1007/978-3-642-81955-1_28.

[UHV10] Josef Urban, Kryštof Hoder, and Andrei Voronkov. Evaluation of automated theorem proving on the Mizar Mathematical Library. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *ICMS*, volume 6327 of *LNCS*, pages 155–166. Springer, 2010. doi:10.1007/978-3-642-15582-6_30.

[Urb04] Josef Urban. MPTP - motivation, implementation, first experiments. *J. Autom. Reasoning*, 33(3-4):319–339, 2004. doi:10.1007/s10817-004-6245-1.

[Urb15] Josef Urban. BliStr: The blind strategymaker. In Gottlob et al. [GSV15], pages 312–319. doi:10.29007/8n7m.

[USPV08] Josef Urban, Geoff Sutcliffe, Petr Pudlák, and Jiří Vyskočil. MaLARea SG1- machine learner for automated reasoning with semantic guidance. In Armando et al. [ABD08], pages 441–456. doi:10.1007/978-3-540-71070-7_37.

[UVv11] Josef Urban, Jiří Vyskočil, and Petr Štěpánek. MaLeCoP machine learning connection prover. In Brünnler and Metcalfe [BM11], pages 263–277. doi:10.1007/978-3-642-22119-4_21.

[vBJ77] L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the Automath system.* PhD thesis, Technische Universiteit Eindhoven, 1977. doi:10.6100/IR23183.

[Ver96] Robert Veroff. Using hints to increase the effectiveness of an automated reasoning program: Case studies. *J. Autom. Reasoning*, 16(3):223–239, 1996. doi:10.1007/BF00252178.

[Vor14] Andrei Voronkov. AVATAR: the architecture for first-order theorem provers. In Armin Biere and Roderick Bloem, editors, *CAV*, volume 8559 of *LNCS*, pages 696–710. Springer, 2014. doi:10.1007/978-3-319-08867-9_46.

[Wha16] Daniel Whalen. Holophrasm: a neural automated theorem prover for higher-order logic. *CoRR*, abs/1608.02644, 2016. URL http://arxiv.org/abs/1608.02644.

[Wie02] Freek Wiedijk. A new implementation of automath. *J. Autom. Reasoning*, 29(3-4):365–387, 2002. doi:10.1023/A:1021983302516.

[WPN08] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle framework. In Mohamed et al. [MMT08], pages 33–38. doi:10.1007/978-3-540-71067-7_7.

[WTWD17] Mingzhe Wang, Yihe Tang, Jian Wang, and Jia Deng. Premise selection for theorem proving by deep graph embedding. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *NIPS*, pages 2783–2793, 2017. URL http://papers.nips.cc/paper/6871-premise-selection-for-theorem-proving-by-deep-graph-embedding.

[ZZ05] Min-Ling Zhang and Zhi-Hua Zhou. A k-nearest neighbor based algorithm for multi-label classification. In Xiaohua Hu, Qing Liu, Andrzej Skowron, Tsau Young Lin, Ronald R. Yager, and Bo Zhang, editors, *International Conference on Granular Computing*, pages 718–721. IEEE, 2005. doi:10.1109/GRC.2005.1547385.

[ZZ14] Min-Ling Zhang and Zhi-Hua Zhou. A review on multi-label learning algorithms. *IEEE Trans. Knowl. Data Eng.*, 26(8):1819–1837, 2014. doi:10.1109/TKDE.2013.39.