J. Automated Reasoning manuscript No.
(will be inserted by the editor)

Machine Learning Guidance for Connection Tableaux

Michael Färber · Cezary Kaliszyk · Josef Urban

Abstract Connection calculi allow for very compact implementations of goal-directed proof search. We give an overview of our work related to connection tableaux calculi: First, we show optimised functional implementations of connection tableaux proof search, including a consistent Skolemisation procedure for machine learning. Then, we show two guidance methods based on machine learning, namely reordering of proof steps with Naive Bayesian probabilities, and expansion of a proof search tree with Monte Carlo Tree Search.

1 Introduction

Connection calculi enable goal-directed proof search in a variety of logics. Connections were considered among others for classical first-order logic (Letz et al., 1992), for higher-order logic (Andrews, 1989) and for linear logic (Galmiche, 2000).

An important family of connection provers for first-order logic is derived from leanCoP (Otten and Bibel, 2003; Otten, 2008). leanCoP was inspired by leanTAP (Beckert and Posegga, 1995), 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 (Otten, 2005)), modal (MleanCoP (Otten, 2014)), and nonclausal first-order logic (nanoCoP (Otten, 2016)). The intuitionistic version of leanCoP (Otten, 2005) became the state-of-art prover for first-order problems in intuitionistic logic (Raths et al.,

M. Färber
University of Innsbruck, Austria
E-mail: michael.faerber@gedenkt.at

C. Kaliszyk
University of Innsbruck, Austria
E-mail: cezary.kaliszyk@uibk.ac.at

J. Urban
Czech Technical University in Prague, Czech Republic
E-mail: josef.urban@gmail.com

2007). A variant of leanCoP with interpreted linear arithmetic (leanCoP-Ω) won the TFA division of CASC-J5 (Sutcliffe, 2011). Various implementation modifications can be performed very elegantly, such as search strategies, scheduling, restricted backtracing (Otten, 2010), randomization of the order of proof search steps (Raths and Otten, 2008), and internal guidance (Urban et al., 2011; Kaliszyk and Urban, 2015a).

We have used connection provers from the leanCoP family as a basis for experiments with *machine learning* (see section 5) and *proof certification* (Kaliszyk et al., 2015a). 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 (Harrison, 2009), HOL4 (Slind and Norrish, 2008), Isabelle (Wenzel et al., 2008), Coq (Bertot, 2008), and Agda (Bove et al., 2009) are written in functional programming languages, lending themselves well to integration of functional proof search tactics. Second, several machine learning algorithms such as Naive Bayes and k-NN have been implemented efficiently for ITPs in functional languages (Kaliszyk and Urban, 2015b; Blanchette et al., 2016b). Third, we achieve better performance with functional-style implementations, which is important to compensate for the performance penalty incurred by machine learning.

In this paper we develop an integration of internal guidance based on machine learning and Monte Carlo methods in connection-style proof search. The contributions described in this paper are:

- We implement proof search based on clausal and nonclausal connection tableaux calculi in functional programming languages, improving performance upon previous Prolog-based implementations, see section 3.
- We show a method to order proof search steps by using a Naive Bayes classifier based on previous proofs, see section 4.
- We use Monte Carlo Tree Search to guide connection proof search, see section 5. To this end, we propose and evaluate several proof state evaluation heuristics, including two that learn from previous proofs.

The paper combines, compares, and extends our works presented at LPAR 2015 (Kaliszyk and Urban, 2015a) and CADE 2017 (Färber et al., 2017). The techniques added over the conference versions include: consistent Skolemisation applicable also for nonclausal proof search and efficient functional-style implementation of proof search in clausal and nonclausal connection calculi.[1]

2 Connection Calculi

Connection calculi provide a goal-oriented way to search for proofs in classical and nonclassical logics (Otten, 2008). 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 (Hähnle, 2001). Among these calculi are the connection method (Bibel, 1983, 1987), the connection tableau calculus (Letz and Stenz, 2001), and model elimination (Loveland, 1968).

[1] The source code of all implementations in this article is available at http://cl-informatik.uibk.ac.at/users/mfaerber/cop.html.

Axiom $\dfrac{}{\{\}, M, Path}\,\mathrm{A}$

Start $\dfrac{C_2, M, \{\}}{\varepsilon, M, \varepsilon}\,\mathrm{S}$ where C_2 is a copy of $C_1 \in M$

Reduction $\dfrac{C, M, Path \cup \{L'\}}{C \cup \{L\}, M, Path \cup \{L'\}}\,\mathrm{R}$ where $\sigma(L) = \sigma(\overline{L'})$

Extension $\dfrac{C_2 \setminus \{L'\}, M, Path \cup \{L\} \quad C, M, Path}{C \cup \{L\}, M, Path}\,\mathrm{E}$

where C_2 is a copy of $C_1 \in M$ and $L' \in C_2$ with $\sigma(L) = \sigma(\overline{L'})$

Fig. 1: Clausal connection calculus rules.

In this section, we introduce the *clausal connection calculus* that we will use throughout the paper. As this calculus has a small set of rules, it lends itself very well to machine learning. For a description of the rules of the *nonclausal connection calculus*, we refer to (Otten, 2011).

The connection calculi in this paper operate on *matrices*, where a matrix is a set of clauses. In the clausal connection calculus, a clause is a set of literals. 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 now give a definition of the common parts of the clausal and nonclausal connection calculi.

Definition 1 (Connection Calculus, Connection Proof) The words of a *connection calculus* are tuples $\langle C, M, Path\rangle$, where C is a clause, M is a matrix, and $Path$ is a set of literals called the *active path*. C and $Path$ can 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 $\langle C, M, Path\rangle$ is a derivation in a connection calculus for $\langle C, M, Path\rangle$ in which all leaves are axioms. A connection proof for M is a connection proof for $\langle \varepsilon, M, \varepsilon\rangle$.

To complete the definition of the clausal connection calculus, we present the calculus rules in Figure 1.

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 1 Consider the following formula F and its prenex conjunctive normal form F'. We will show that F' implies $\bot$:

$$F = Q \wedge P(a) \wedge \forall x.(\neg P(x) \vee (\neg P(s^2x) \wedge (P(sx) \vee \neg Q)))$$
$$F' = \forall x.(Q \wedge P(a) \wedge (\neg P(x) \vee \neg P(s^2x)) \wedge (\neg P(x) \vee P(sx) \vee \neg Q))$$

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

Fig. 2: Clausal connection proof.

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

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

A formal proof for M' in the clausal connection calculus is given in Figure 2.

Soundness and completeness have been proved both for the clausal (Letz and Stenz, 2001) and for the nonclausal connection calculus (Otten, 2011). We will discuss practical functional-style implementations of proof search for both calculi in section 3.

3 Functional-style Connection Prover

In this section, 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 in the remainder of the paper.

The connection prover performs the following tasks. Given a classical first-order logic problem, it creates a matrix for the problem, see subsection 3.1. The matrix is then used to build an index that provides an efficient way to find connections during proof search, see subsection 3.3. Finally, proof search with iterative deepening is performed, see subsection 3.4.

3.1 Problem Preprocessing

In this section, we show how the prover transforms problems into 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 (Sutcliffe, 2009b) 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 (Otten, 2008, 2016).

1. The conjecture C is combined with the axioms $\{A_1,\ldots,A_n\}$ to form the new problem $(A_1\wedge\cdots\wedge A_n)\rightarrow C$ (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 paper 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

$$\forall x.x = x \tag{refl$_=$}$$
$$\forall xy.x = y \to y = x \tag{sym$_=$}$$
$$\forall xyz.x = y \wedge y = z \to x = z \tag{trans$_=$}$$

   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 $(P \wedge \#) \to (C \wedge \#)$. $\#$ is a marker that can be understood to be equivalent to $\top$. It allows proof search to recognise clauses stemming from the conjecture (Otten, 2008, section 2.1).
5. Implications and equivalences are expanded, e.g. $A \to B$ becomes $\neg A \vee B$.
6. Quantifiers are pushed inside so 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 so that smaller clauses are processed earlier. In nanoCoP, the size of a formula is

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

   and for any subformula $t_1 \wedge t_2$ or $t_1 \vee t_2$, 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 subsection 3.2 instead of outer Skolemisation as performed in the original Prolog version.

Example 2 Consider the axioms

$$\forall xAB.x \in A \cup B \leftrightarrow (x \in A \vee x \in B) \tag{def$_\cup$}$$
$$\forall AB.(\forall x.x \in A \leftrightarrow x \in B) \to A = B \tag{def$_=$}$$

that we want to use to prove

$$\forall ABC.A \cup (B \cup C) = (A \cup B) \cup C \tag{C}$$

The problem is preprocessed as follows:

1. The axioms $A \equiv \text{def}_\cup \wedge \text{def}_=$ and the conjecture are combined, resulting in $A \to C$.

2. Constants and variables are mapped to integers, e.g. $\{\text{“}\in\text{”} \mapsto 0, \text{“}\cup\text{”} \mapsto 1, \text{“}=\text{”} \mapsto 2\}$ and $\{\text{“}x\text{”} \mapsto 0, \text{“}A\text{”} \mapsto 1, \text{“}B\text{”} \mapsto 2\}$. We will continue the presentation of this example with the original representation.
3. Congruence axioms are generated for all constants, i.e. “$\in$” and “$\cup$”:

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

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

   The combination of all equality axioms is

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

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

$$\forall x AB.(x \notin A \cup B \vee (x \in A \vee x \in B)) \wedge (\neg(x \in A \vee x \in B) \vee x \in A \cup B) \quad (\text{def}_\cup)$$

$$\forall AB.((\neg\forall x.((x \notin A \vee x \in B) \wedge (x \notin B \vee x \in A))) \vee A = B) \qquad (\text{def}_=)$$

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

$$\begin{array}{c}(\forall x AB.(x \notin A \cup B \vee (x \in A \vee x \in B)))\wedge \\ (\forall x AB.(\neg(x \in A \vee x \in B) \vee x \in A \cup B))\end{array} \qquad (\text{def}_\cup)$$

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

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

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

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

$$\begin{array}{c}(\forall x AB.((x \notin A \wedge x \notin B) \vee x \in A \cup B))\wedge \\ (\forall x AB.(x \notin A \cup B \vee (x \in A \vee x \in B)))\end{array} \qquad (\text{def}_\cup)$$

   and the resulting formula is $(\neg\# \vee C_\neg) \wedge (\# \wedge ((\text{def}_= \wedge \text{def}_\cup) \wedge E))$. 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

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

   where s_A, s_B, and s_C are nullary Skolem functions. We explain Skolemisation in more detail in subsection 3.2.

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 $A \wedge (B \vee C) \equiv (A \vee B) \wedge (A \vee C)$ 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 (Tseitin, 1983; Plaisted and Greenbaum, 1986; Otten, 2010). 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.

3.2 Consistent Skolemisation

The Skolemisation of a formula Δ replaces existentially quantified variables occurring in Δ by newly introduced function symbols called Skolem functions, yielding a formula equisatisfiable to Δ without existential quantifiers. Skolemisation may introduce distinct Skolem functions when a single Skolem function would have sufficed. For example, a subformula $\exists x.P(x)$ of two formulas Δ_1 and Δ_2 may be Skolemised to $P(s_1)$ in Δ_1 and to $P(s_2)$ in Δ_2, such that $s_1 \neq s_2$. This makes it difficult to spot in hindsight that s_1 and s_2 were produced from equivalent subformulas. For machine learning, however, when we learn something about a formula containing a Skolem function, such as $P(s_1)$, we wish to transfer this knowledge to a different formula where the same formula was Skolemised to $P(s_2)$. To solve this problem, we present a new Skolemisation method that introduces Skolem functions "consistently". In general, a consistent Skolemisation method can instantiate different existentially quantified variables with the same Skolem functions under certain conditions. For example, a consistent Skolemisation method could ensure for the example above that $s_1 = s_2$.

Consistent Skolemisation methods have been studied in the context of the δ-rule in tableaux methods (Beckert et al., 1993). The Skolem terms introduced by such methods may lead to rather large formulas unless techniques such as structure sharing are used (Giese and Ahrendt, 1999). However, in our setting, such techniques would complicate the parallel execution of several prover instances and require the adaption of both the prover and the machine learning methods. We propose a new consistent Skolemisation method which produces reasonably small Skolem functions without relying on structure sharing.

We assume that the formulas in this section are in negation normal form. A position p is a sequence $p_1 \dots p_n$, where every p_i is either 0 or 1. The empty sequence ε denotes the root position, and pq is the concatenation of two positions p and q. The subformula of F at the position p, denoted as $F|_p$, is defined as follows: $F|_\varepsilon = F$, and if $F = \exists x.G$ or $F = \forall x.G$, then $F|_{0p} = G|_p$, and if $F = G_1 \wedge G_2$ or $F = G_1 \vee G_2$, then $F|_{0p} = G_1|_p$ and $F|_{1p} = G_2|_p$.

The sequence of free variables of F is denoted by $\mathcal{FV}\mathrm{ar}(F)$. The order of the sequence $\mathcal{FV}\mathrm{ar}(F)$ must not depend on the names of the variables; i.e. for any bijective substitution σ, if $\sigma F = G$, then $\sigma\,\mathcal{FV}\mathrm{ar}(F) = \mathcal{FV}\mathrm{ar}(G)$. The sequence of existentially/universally quantified variables of a formula F along p is $\mathcal{V}\mathrm{ar}_Q(F, p) = \bigcup_{i<|p|} \{x \mid \Delta|_{p_1 \dots p_i} = Qx.G\}$, where the sequence is ordered by ascending i and $Q \in$

$\{\forall, \exists\}$. For example, if $\Delta = \forall x.\exists yz.P(x, y, z)$ and $p = 00$, then $\Delta|_p = \exists z.P(x, y, z)$, $\mathcal{V}\mathrm{ar}_\forall(\Delta, p) = [x]$, and $\mathcal{V}\mathrm{ar}_\exists(\Delta, p) = [y]$.

To describe multiple Skolemisation methods, we introduce a Skolemisation operator $S_f(\sigma, \Delta, p)$. This operator is parametrised by a Skolemisation function $f(x, \sigma, \Delta, p) = \sigma'$, which yields for a formula $\sigma(\Delta|_p) = \exists x.F$ an equisatisfiable formula $\sigma'(\Delta|_{p0})$.[2] The operator $S_f(\sigma, \Delta, p)$ returns the f-Skolemisation of $\sigma(\Delta|_p)$. It follows that the f-Skolemisation of Δ is $S_f(\emptyset, \Delta, \varepsilon)$. The purpose of a Skolemisation function is to eliminate a single existential quantifier, whereas the Skolemisation operator uses the Skolemisation function to eliminate all existential quantifiers of a formula.

$$S_f(\sigma, \Delta, p) = \begin{cases} S_f(f(x, \sigma, \Delta, p), \Delta, p0) & \text{if } \Delta|_p = \exists x.F \\ \forall x.S_f(\sigma, \Delta, p0) & \text{if } \Delta|_p = \forall x.F \\ S_f(\sigma, \Delta, p0) \wedge S_f(\sigma, \Delta, p1) & \text{if } \Delta|_p = F_1 \wedge F_2 \\ S_f(\sigma, \Delta, p0) \vee S_f(\sigma, \Delta, p1) & \text{if } \Delta|_p = F_1 \vee F_2 \\ \sigma\Delta|_p & \text{if } \Delta|_p = A \text{ or } \Delta|_p = \neg A \end{cases}$$

The Skolemisation function for inner Skolemisation is $IS(x, \sigma, \Delta, p) = \sigma \cup \{x \mapsto s(\vec{y})\}$ with s denoting a fresh Skolem function symbol and $\vec{y} = \mathcal{FV}\mathrm{ar}(\sigma(\Delta|_p))$. This performs inner Skolemisation from left to right, to avoid nesting of Skolem functions (Nonnengart, 1996). While producing small formulas, this is not a consistent Skolemisation method, as all different existentially quantified variables are mapped to different Skolem functions.

An alternative Skolemisation method uses epsilon notation (Hilbert and Bernays, 1939). The critical axiom of the epsilon calculus is $P(t) \rightarrow P(\epsilon x.P(x))$ from which one can derive $\exists x.P(x) \leftrightarrow P(\epsilon x.P(x))$. The Skolemisation function for epsilon-Skolemisation is then $\epsilon S(x, \sigma, \Delta, p) = \sigma \cup \{x \mapsto \epsilon x.\sigma(\Delta|_{p0})\}$. Epsilon-Skolemisation maps the subformulas $\exists x.P(x)$ from the introductory example to the same term, namely $P(\epsilon x.P(x))$, therefore it is a consistent Skolemisation method. However, epsilon-Skolemisation requires an extension of first-order logic. Furthermore, like previous consistent Skolemisation methods, it can yield exponentially large formulas unless structural sharing is used.

The key to obtaining a consistent Skolemisation method for our setting is to combine inner Skolemisation with epsilon-Skolemisation. We can express any Skolem term $s(\vec{y})$ introduced by inner Skolemisation by some epsilon term $\epsilon x.F$ introduced by epsilon-Skolemisation. We will show a consistent Skolemisation method that uses this correspondence, by introducing the same Skolem terms whenever their underlying epsilon terms are alpha-equivalent.

We now show how to consistently Skolemise a subformula $\exists x.F$ of Δ at position p. We will refer to variables that are existentially quantified in Δ as existential variables and to variables that are universally quantified in Δ as universal variables. Our consistent Skolemisation proceeds in three steps: First, we obtain the smallest subformula $F_{\min}$ of Δ that contains $\Delta|_p$ and contains no free existential variables. The minimality of $F_{\min}$ serves to maximise the number of equivalent existential variables

[2] The substitution σ maps existentially quantified variables to their respective Skolem terms. The new substitution σ' is a strict superset of σ because it preserves all previously established mappings to Skolem terms as well as adds a new mapping from the variable x to some Skolem term.

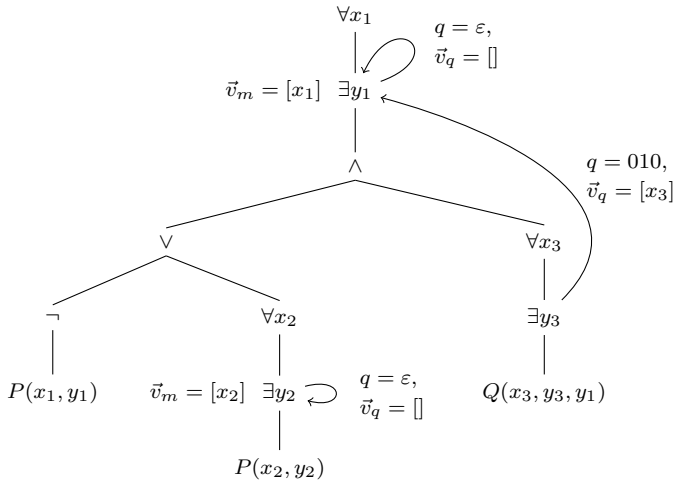


Fig. 3: Illustration of consistent Skolemisation example.

being mapped to the same Skolem function, which is important for learning. Next, we obtain the position q of $\Delta|_p$ in $F_{\min}$. The idea is that encoding the combination of $F_{\min}$ and q in the name of the Skolem function suffices to characterise $\Delta|_p$, allowing to reconstruct the epsilon term $\epsilon x.F$. Finally, we obtain the arguments of the Skolem function.

1. We show how to obtain $F_{\min}$. For this, we determine the path $p_1 \dots p_m$ that is the longest prefix of p such that $\Delta|_{p_1 \dots p_m}$ does not contain free existential variables. We obtain this by $m = \max\{i \mid \mathcal{V}\text{ar}_\exists(\Delta, p_1 \dots p_i) \cap \mathcal{FV}\text{ar}(\Delta|_{p_1 \dots p_i}) = \emptyset\}$, from which we can obtain $F_{\min} = \Delta|_{p_1 \dots p_m}$. Because $F_{\min}$ may contain free universal variables $\vec{v}_m = \mathcal{FV}\text{ar}(F_{\min})$, we abstract over $\vec{v}_m$ to obtain a closed term $F_\lambda = \lambda \vec{v}_m.F_{\min}$ that can be alpha-normalised. We call F_α the alpha-normalisation of F_λ.
2. We obtain the position q of $\Delta|_p$ in $F_{\min}$ by the equation $p = p_1 \dots p_m q$. Let q_1, $\dots$, q_k be all prefixes of q, such that $F_{min}|_{q_i}$ is an existentially quantified formula. Note, $q_1 = \epsilon$ and $q_k = q$.
3. For each i in $[1, k]$, let v_i be the sequence of universal variables that freely occur in $F_{min}|_{q_i}$ and do not belong to any of v_j where $j < i$. In other words, v_i is the sequence of universal variables freely occurring in $F_{min}|_{q_i}$ and bound under the previous existential quantifier in F_{min}, if any.
4. We determine the arguments of the Skolem term to be $\vec{y} = \vec{v}_m v_1 \dots v_k$.

Putting everything together, the Skolemisation function for our consistent Skolemisation is $CS(x, \sigma, \Delta, p) = \sigma \cup \left\{x \mapsto s_q^{F_\alpha}(\vec{y})\right\}$, where s_p^F denotes a first-order function symbol that carries in its name the formula F as well as the position p.

Example 3 Our consistent Skolemisation of the introductory example $\exists x.P(x)$ is $P(s_\varepsilon^{\exists x.P(x)})$.

Example 4 Let

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

and let Δ^n denote the subformula $\exists y_n \ldots$ in Δ. The formula Δ is illustrated in Figure 3, where an arrow from Δ^i to a formula F signifies that F is the $F_{\min}$ corresponding to Δ^i, and the arrow labels q and $\vec{v}_q$ are q and $\vec{v}_q$ corresponding to Δ^i. We then see that $F_{\min}$ for Δ^1 and Δ^3 is Δ^1, and $F_{\min}$ for Δ^2 is Δ^2. Furthermore, let Δ^n_λ and Δ^n_α be F_λ and F_α for Δ^n, respectively. Then $\Delta^1_\lambda = \lambda x_1.\exists y_1.(\neg P(x_1, y_1) \vee (\forall x_2 \exists y_2.P(x_2, y_2)) \wedge (\forall x_3 \exists y_3.Q(x_3, y_3, y_1)))$ and $\Delta^2_\lambda = \lambda x_2.\exists y_2.P(x_2, y_2)$. The position q of Δ^3 in Δ^1 is 010. Therefore, our consistent Skolemisation of Δ is $\sigma\Delta'$, where

$$\sigma = \left\{ y_1 \to s_\varepsilon^{\Delta^1_\alpha}(x_1),\ y_2 \to s_\varepsilon^{\Delta^2_\alpha}(x_2),\ y_3 \to s_{010}^{\Delta^1_\alpha}(x_1, x_3) \right\}$$

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

The combination of F_α and q in the newly introduced Skolem function name allows the reconstruction of the epsilon term which epsilon-Skolemisation would have introduced for x. We can expand $s_q^{F_\alpha}(\vec{x})$ with $F_\alpha = \lambda \vec{v}_m.F_{\min}$ to its corresponding Skolem epsilon term by performing epsilon-Skolemisation of $F_{\min}$, obtaining the epsilon term t that was introduced for the variable that is existentially quantified at position q in $F_{\min}$, and instantiating the free variables of t by beta-reducing $(\lambda \vec{v}_m.\lambda \vec{v}_q.t)\vec{x}$, where $\vec{v}_q = \mathcal{V}\mathrm{ar}_\forall(F_{\min}, q) \cap \mathcal{FV}\mathrm{ar}(F_{\min}|_q)$.

Encoding F_α and q in the name of a first-order function symbol avoids dedicated procedures during proof search to decide the equivalence of Skolem functions and does not require adaptations of the machine learning methods to account for Skolem functions. The resulting Skolem function names are linear in the size of Δ.

3.3 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*.

Definition 2 (Clausal Contrapositive) Given a clausal matrix M with $C \in M$ and $L \in C$, the formula $\overline{L} \to C \setminus \{L\}$ 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 5 Consider the matrix M' from Example 1 on page 3. A contrapositive of M' is $Q \to (\neg P(x) \vee P(sx))$. This contrapositive was used to find the connection $\{Q, \neg Q\}$ and to generate the corresponding extension clause $\{\neg P(x'), P(sx')\}$ in Figure 2.

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 (Ramakrishnan et al., 2001). However, the overall effect on the performance of storing contrapositives in a discrimination tree (Greenbaum, 1986) 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, and replace their free variables with fresh ones. 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.

3.4 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 connection provers leanCoP and nanoCoP use a number of optimisation techniques, such as regularity, lemmas, and restricted backtracking (Otten, 2010). 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 anymore. 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 low speed (Urban et al., 2011). Later, (Kaliszyk and Urban, 2015a; Kaliszyk et al., 2015a) showed that implementations of leanCoP in a functional programming language allow for fast machine learning. 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 can be adapted to use restricted backtracking: The simplified version of leanCoP shown in subsubsection 3.4.1 is the smallest, but also the slowest implementation. For the sake of performance comparison, we take care that all subsequent implementations perform the proof search in precisely the same order as the original Prolog implementation. We then introduce purely functional implementations in subsubsection 3.4.2 using lazy lists and streams. This version slightly increases code size compared to the Prolog version, but greatly improves performance, as shown in the evaluation in

Listing 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).
```

subsection 3.5. 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 paper uses an explicit stack and exceptions for backtracking. However, as it proves in our evaluation just as many problems as the continuation-based version, we will only briefly discuss it.

3.4.1 Prolog

A simplified version of the original leanCoP in Prolog is given in Listing 1. We explain and relate it to the clausal connection calculus introduced in section 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 subsection 3.3 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.

3.4.2 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

Listing 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)
```

a common interface for lazy lists and streams, we solely present the lazy list version here.

Listing 2 shows a functional leanCoP implementation using lazy lists.[3] 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 $[\sigma_1, \ldots, \sigma_n]$, 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.[4] 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.[5] As we use lazy lists / streams, a substitution σ_i is only calculated if proof search failed for all σ_j with $j < i$.

3.4.3 Continuations

Continuation passing style (CPS) allows the implementation of algorithms with complicated control flow in functional languages (Plotkin, 1975). Listing 3 shows a

[3] Several of the algorithms shown in this paper rely on lazy evaluation. Therefore, we show Haskell versions of our algorithms, which are shorter than those in our actual implementation language OCaml.

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

[5] The shown program could be easily improved, for example by moving the check `lim <= 0` from line 7 to line 5. However, the actual implementation performs at this place a more complex check which cannot be moved this way. Therefore we leave the check here as it is.

Listing 3: CPS implementation of clausal proof search.

```
1  prove [] path lim sub alt rem = rem sub alt
2  prove (lit : cla) path lim sub alt rem = reduce path where
3    reduce (plit : path) =
4      let alt1 = reduce path
5      in case unify sub (negate lit) plit of
6        Nothing -> alt1
7        Just sub1 -> prove cla path lim sub1 alt1 rem
8    reduce [] = extend (unifyDB sub (negate lit))
9
10   extend ((sub1, cla1) : contras) =
11     let alt1 = extend contras
12     in if lim <= 0 then alt1
13        else
14          let rem1 sub alt = prove cla path lim sub alt rem
15          in prove cla1 (lit : path) (lim - 1) sub1 alt1 rem1
16   extend [] = alt
```

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 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). The `extend` function 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).

3.4.4 Stacks

The last considered implementation uses explicit 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.

3.5 Evaluation

We evaluate the functional connection provers on several first-order problem datasets, with statistics given in Table 1:

– TPTP (Sutcliffe, 2009b) is a large benchmark for automated theorem provers. It is used in CASC (Sutcliffe, 2016b). 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.

Table 1: Evaluation datasets and the 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

– MPTP2078 (Alama et al., 2014) 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 (Kaliszyk and Urban, 2015c). The problems are translated to untyped first-order logic using the MPTP infrastructure (Urban, 2004). Symbol names are also used consistently in this dataset, and the problems are minimised using ATP-based 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 (Kaliszyk and Urban, 2014). We export top-level theorems ("top") as well as theorems proven by the MESON tactic ("msn").[6] We consider the theorems proven in the core of HOL Light ("HL") as well as those proven by the Flyspeck project ("FS"), which finished in 2014 a formal proof of the Kepler conjecture (Hales et al., 2017).

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 2. As state of the art, we use the ATPs Vampire 4.0 (Kovács and Voronkov, 2013) and E 2.0 (Schulz, 2013), which performed best in the first-order category of CASC-J8 (Sutcliffe, 2016a). Vampire and E are written in C++ and C, respectively, implement the superposition calculus, and perform premise selection with SInE (Hoder and Voronkov, 2011). Furthermore, Vampire integrates several SAT solvers (Biere et al., 2014), and E automatically determines proof search settings for a given problem. We ran E with `--auto-schedule` and Vampire with `--mode casc`. In addition, we evaluated the ATP Metis (Hurd, 2003): 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.[7] Our implementations use the techniques

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

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

Table 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	1567	542	151	13316	1993	37938
pnanoCoP+cut	1585	480	112	11921	2970	30272
pnanoCoP−cut	1485	510	126	12943	1986	38015

introduced such as hash-based indexing and array-based substitutions (subsection 3.3), efficient control flow (subsection 3.4), and consistent Skolemisation (subsection 3.2), as well as all optimisation techniques of the Prolog implementations, such as regularity, lemmas, and restricted backtracking. We refer to our functional OCaml implementations as fleanCoP and fnanoCoP, whereas we refer to the original Prolog versions as 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 and "conj" stands for conjecture-directed search. leanCoP was evaluated without definitional clausification, see subsection 3.1. The OCaml implementations use streams to control backtracking (see subsubsection 3.4.2) 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 2: The OCaml versions outperform the Prolog versions in almost all cases (the exception being fnanoCoP−cut on the FS-msn dataset). 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.

nanoCoP solves more problems than leanCoP on the datasets Miz40 and FS-msn, in both cases without cut. However, for both datasets, nanoCoP proves fewer problems than any of the reference provers Vampire, E, and Metis. In conclusion, in scenarios where both Metis and leanCoP are available, the current version of nanoCoP cannot play its theoretical strength[8] in any of the datasets evaluated.

We evaluate different proof search implementation styles in Tables 3 and 4. Here, inferences denote the number of successful unifications performed by some prover on

[8] The nonclausal calculus underlying nanoCoP can linearly simulate the clausal calculus underlying leanCoP, but there exist nonclausal proofs of which no clausal equivalent of polynomial size exists (Otten, 2011).

Table 3: Impact of implementation on the efficiency of clausal proof search on the bushy MPTP2078 dataset with 10 seconds 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 4: Impact of implementation on efficiency of nonclausal proof search on the bushy MPTP2078 dataset with 10 seconds timeout and restricted backtracking (+cut).

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

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. This decreased the number of inferences by 50%, showing that the performance of the substitution structure is crucial for fast proof search.

4 Naive Bayesian Internal Guidance

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. Ranking heuristics that learn from previous proofs are an example of internal guidance. In this section, we propose an internal guidance method using Naive Bayesian probability to guide connection proof search based on its intermediate proof state and previous proofs.

The assumption underlying our approach is the following: An action that was useful in a past state is likely to be useful in similar future states. What do actions, usefulness, and states signify in our setting of guiding connection proof search? We consider as action the application of the extension rule with a given contrapositive (see subsection 3.3), because the order in which extension steps are tried has a significant effect on the performance of proof search. Furthermore, we consider an action to have

been useful if the extension step ends up in the final proof. Finally, the state in which an action is performed is the proof branch in which the extension step is applied.

This assumption implies that we can estimate the usefulness of an action in a present state from the usefulness of the action in similar past states. More specifically, to estimate the usefulness of a contrapositive in the current proof branch, we can consider the usefulness of the contrapositive in similar proof branches of previous proofs. When we have a choice between different contrapositives, we can process them in order of decreasing estimated usefulness, in order to find proofs faster.

To measure the similarity between proof branches, we characterise them by *features* (Kaliszyk et al., 2015b), which we explain in subsection 4.1. In subsection 4.2, we then calculate the utility of a contrapositive in the current branch, given knowledge about its utility in previous proofs. In subsection 4.3, we motivate the integration of machine learning methods in the prover and introduce the prover FEMaLeCoP, which we evaluate in subsection 4.4.

4.1 Tableau Branch Characterisation

The words of the connection tableaux calculus $\langle C, M, Path\rangle$ correspond to a set of tableau branches sharing the active $Path$. Therefore, to characterise a branch, we use as its *features* the set of symbols occurring in the active path. This does not include symbols in the substitution.[9] We weigh the symbols by the number of times they appeared in all problems, giving higher weight to rarer symbols via *inverse document frequency* (Jones, 1973), as well as by the distance between the current depth and the depth the symbols were put onto the path, giving higher weight to symbols more recently processed.

4.2 Naive Bayes

Given a set of contrapositives that are applicable in a tableau branch, we wish to obtain an ordering of the contrapositives such that trying the contrapositives in the given order minimises the time spent to find a proof. In this subsection, we show how to order the set of applicable contrapositives by a formula NB that is based on Naive Bayesian probability, as used for premise selection (Kaliszyk and Urban, 2015c).

Adopting machine learning jargon, we will refer to contrapositives as *labels*, and we say that a label l co-occurred with a set of features $\vec{f}$ if the contrapositive l was used in a proof branch characterised by features $\vec{f}$ and l contributed to the final proof, and we say that l occurred if it co-occurred with some set of features. For this, we introduce a function $F(l)$, which returns the multiset of sets of 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 6 $F(l_1) = \{\{f_1, f_2\}, \{f_2, f_3\}\}$ 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.

[9] It is possible either to consider or to disregard symbols that are introduced by previous unifications, i.e. symbols in the global substitution. In our experiments, disregarding such symbols as features turned out to be more beneficial.

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 discriminate against 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 $\mathrm{idf}(f)$ 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

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

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 the number of times f_j appeared when l_i was used to the number of times that l_i was used. Otherwise, the probability is estimated to be a minimal constant probability μ:

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

Here, $\mathbf{1}_A(x)$ denotes the indicator function that returns 1 if $x \in A$ and 0 otherwise.

4.3 Implementations

The *Machine Learning Connection Prover* (MaLeCoP) was the first leanCoP-based system to explore the feasibility of machine-learnt internal guidance (Urban et al., 2011). MaLeCoP relies on an external machine learning framework (using by default the SNoW system (Carlson et al., 1999)), 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 subsection 4.2 into the prover (Kaliszyk and Urban, 2015a). 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 subsection 3.3. Performance is further improved by updating branch features from the previous branch, instead of fully recalculating them in every new branch.

4.4 Evaluation

The evaluation of Naive Bayes guidance (as well as the comparative evaluation of other methods in the next section) involves collecting training data by running leanCoP on a training dataset followed by running guided FEMaLeCoP both on training data and on a testing set. Additionally, to maximize the amount of available training data, we will split the dataset in such a way that problems that unmodified leanCoP can solve will be in the training set and unsolved problems will be in the testing set. We run both leanCoP and FEMaLeCoP on the bushy MPTP2078 dataset with a timeout of 60 seconds, use nondefinitional clausification, conjecture-directed search and restricted backtracking. Both leanCoP and FEMaLeCoP considered in this evaluation are implemented in OCaml using continuation passing style and array-based substitutions, see subsubsections 3.4.3 and 3.3.

The original leanCoP orders the input formula so that more promising (e.g. smaller) clauses are tried earlier, see subsection 3.1. To evaluate the ability of FEMaLeCoP to learn useful clause orders itself, we evaluate versions of leanCoP and FEMaLeCoP that either order the clauses like the original leanCoP or reverse the original order of clauses in the matrix. The latter reduces the number of proven problems compared to the default clause order.

Our evaluation proceeds as follows: We first run leanCoP on all problems. This divides our problems into a training set, namely the problems that leanCoP solves, and a testing set, namely the problems that leanCoP does not solve. From the proofs

Table 5: FEMaLeCoP results, default clause order. Provers run with 60 seconds timeout and restricted backtracking (+cut).

Prover	Training	Testing	$\sum$	$\bigcup$
leanCoP−def	643	0	643	701 (**+9.0%**)
FEMaLeCoP−def	607 (-5.6%)	58	665 (+3.4%)	
leanCoP+def	577	0	577	627 (+8.7%)
FEMaLeCoP+def	542 (-6.1%)	50	592 (+2.6%)	

Table 6: FEMaLeCoP results, reversed clause order. Provers run with 60 seconds timeout and restricted backtracking (+cut).

Prover	Training	Testing	$\sum$	$\bigcup$
leanCoP−def	574	0	574	664 (**+15.7%**)
FEMaLeCoP−def	550 (-4.2%)	90	640 (**+11.5%**)	
leanCoP+def	568	0	568	623 (+9.7%)
FEMaLeCoP+def	540 (-4.9%)	55	595 (+4.8%)	

for the problems in the training set, we extract the information which contrapositive contributed in which tableau branch. We combine this information for all proofs in a format that allows efficient retrieval of learning data for given contrapositives. With the training data generated from the leanCoP proofs, we run FEMaLeCoP on both training and testing set.

The results of the evaluation with default and reversed clause order are shown in Table 5 and Table 6, respectively. The $\sum$ column shows for every prover how many problems it solved in total (i.e. the sum of training and testing problems solved by the prover). The $\bigcup$ column shows how many problems were solved by either the underlying prover gathering data or the machine learning guided prover (i.e. the sum of training problems solved by the unguided prover and testing problems solved by the guided prover).

We detail the results of leanCoP and FEMaLeCoP without definitional clausification and with the reversed clause order. In this setting, leanCoP proves 574 problems. Running FEMaleCoP on this training set proves 550 problems, which is a loss of 4.2% compared to leanCoP. However, on the testing set, FEMaLeCoP proves 90 problems that were unsolved by leanCoP. Combining the problems from the training and testing set, FEMaLeCoP proves 640 problems, which is 11.5% more problems than solved by leanCoP, despite the fact that the inference rate of FEMaLeCoP is about 40% below leanCoP. The union of leanCoP and FEMaLeCoP proves 664 problems, adding 90 problems (15.7%) to the problems solved by leanCoP.

In comparison, when using versions of leanCoP and FEMaLeCoP that use the default clause order or definitional clausification (+def), the gain of proven problems for FEMaLeCoP is lower.

In the next section, we will show another machine learning method and compare its performance with the performance of Naive Bayesian guidance.

5 Monte Carlo Proof Search

Current automated theorem provers are still weak at finding more complicated proofs, especially over large formal developments (Urban et al., 2010). The search typically blows up after several seconds, making the chance of finding proofs in longer times exponentially decreasing (Alama et al., 2012). 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 (Browne et al., 2012). MCTS has been found to produce state-of-the-art players for several games, most notably for the two-player game Go (Silver et al., 2016), but also for single-player games such as SameGame (Schadd et al., 2012) and the NP-hard Morpion Solitaire (Rosin, 2011).

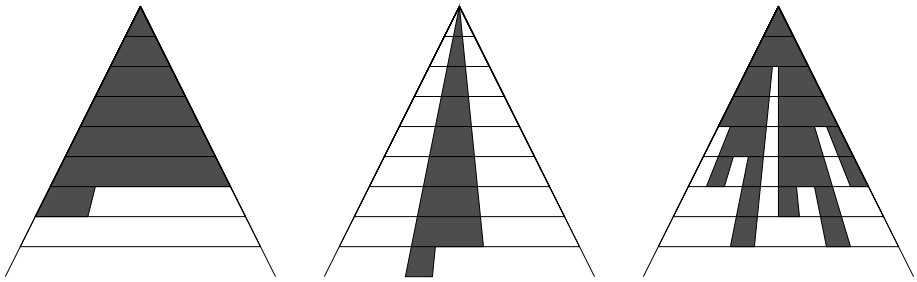
Theorem proving can be seen as a game. For instance, it has been modelled as a two-player game in the framework of game-theoretical semantics (Hintikka, 1982), but it can also be seen as a combinatorial single-player game. As shown for example in the AlphaGo system (Silver et al., 2016), machine learning 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 (Hoder et al., 2016), 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 section, 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 subsection 3.3. For an intuition of the relationship between different proof search strategies, see Figure 4: 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, searching different regions of the 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 subsection 5.1 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 subsection 5.6 and evaluate it in subsection 5.8.

5.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a method to search potentially infinite trees by sampling random tree paths (called *simulations*) (Browne et al., 2012). The outcome of simulations is used to estimate the quality of tree nodes, and MCTS steers search towards nodes with higher quality estimates.



(a) Iterative deepening without restricted backtracking.

(b) Iterative deepening with restricted backtracking.

(c) Monte Carlo.

Fig. 4: The two main leanCoP strategies compared with Monte Carlo proof search.

Definition 3 (Tree) A *tree* is a tuple $(N, n_0, \rightarrow)$, where N is a set of tree nodes, $n_0 \in N$ is the root node, and $\rightarrow \in N \times N$ is a cycle-free relation, i.e. there is no $n \in N$ such that $n \rightarrow^+ n$. We write that n' is a child of n iff $n \rightarrow n'$, and we write that n' is a descendant of n iff $n \rightarrow^+ n'$. Every $n \in N$ is the child of at most one node in N.

We consider connection proof search as traversal of a tree that we define as follows.

Definition 4 (Connection Proof Search Tree) A *connection proof search tree* for a word $\langle C, M, Path\rangle$ is a tree $(N, n_0, \rightarrow)$, where N is the set of derivations, n_0 is a derivation consisting of the word $\langle C, M, Path\rangle$, and $n \rightarrow n'$ iff n' can be obtained from n by a single application of a calculus rule. If $n \rightarrow n'$ by an application of the extension rule using the contrapositive c, then we write $n \xrightarrow{\text{ext}(c)} n'$.

The search for a proof of the word $\langle C, M, Path\rangle$ then succeeds if we find a node $n \in N$ with $n_0 \rightarrow^* n$ such that n is a closed derivation, where $(N, n_0, \rightarrow)$ is the connection proof search tree for $\langle C, M, Path\rangle$.

Let $\rho \in N \rightarrow \mathbb{R}$ 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 (Monte Carlo Tree) A *Monte Carlo tree* T for a tree $(N, n_0, \rightarrow)$ is a tuple $(N_T, \rightarrow_T, \rho_T)$, where $N_T \subseteq N$, $\rightarrow_T \subseteq \rightarrow^+$, and $\rho_T \in N \rightarrow \mathbb{R}$ is a mapping. We write that n' is a T-child of n iff $n \rightarrow_T n'$. The *initial Monte Carlo tree* T_0 is $(N_{T_0}, \rightarrow_{T_0}, \rho_{T_0})$ with $N_{T_0} = \{n_0\}$, $\rightarrow_{T_0} = \emptyset$ 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:[10]

1. **Selection**: A node $n \in N_T$ with $n_0 \rightarrow_T^* n$ is chosen with a *child selection policy*, see subsection 5.2.

[10] 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.

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 subsection 5.3. 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 subsection 5.5. The node n_e is added as a child to n with reward $\rho(n_s)$ (see subsection 5.4) 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 propose heuristics for the child selection policy, child probability, reward, and expansion policy.

5.2 Child Selection Policy

UCT (Upper Confidence Bounds for Trees) is a frequently used child selection policy for Monte Carlo Tree Search (Kocsis and Szepesvári, 2006). 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:

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

5.3 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, $P(n' \mid n) \propto 1/\left(1 + |b_o(n')|\right)$, 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 subsection 4.2 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:

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

  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:

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

5.4 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 leaves, 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

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

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

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

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

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

5.5 Expansion Policy

The expansion policy determines which node n_e of a simulation $n_1 \to \cdots \to n_s$ 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.

Listing 4: Monte Carlo Proof Search as advisor.

```
1  prove [] path lim sub = [sub]
2  prove (lit : cla) path lim sub =
3    let
4      mc = initTree lit path sub & mcps & take (1 + maxIterations)
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)
```

5.6 Implementation

We implemented Monte Carlo proof search (MCPS) on top of the functional implementation of leanCoP using lazy lists, see subsubsection 3.4.2.[11] 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 4, assuming for a simpler presentation that the default expansion policy from subsection 5.5 is used: In line 4, `initTree` L $Path$ σ creates an initial Monte Carlo tree for a connection proof search tree for the word $\langle\{L\}, M, Path\rangle$ under the substitution σ. Starting from this initial Monte Carlo tree T, `mcps` T constructs a (potentially infinite) lazy list of Monte Carlo iterations, with T as its head, where an iteration consists of a Monte Carlo tree and possibly a proof discovered during the simulation performed in the iteration. Of this list, we consider T and the following `maxIterations` elements: When `maxIterations` is set to 0, only T is considered and thus proof search behaves like leanCoP. When `maxIterations` is set to ∞, the whole proof search is performed in the MCPS part. As MCPS is performed lazily, MCPS may be performed for less than `maxIterations` iterations when it discovers some proof contributing to the final closed derivation. Here, the lazy list characterisation introduced in subsubsection 3.4.2 turns out to 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 subsection 3.3 requires that the proof search backtracks only to states whose substitution is a subset of the current state's substitution. However, because this requirement is not fulfilled for MCPS, we use association lists for substitutions.

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

Table 7: 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

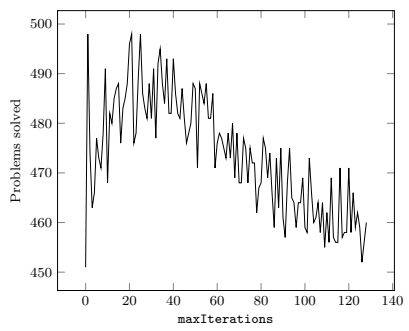
5.7 Parameter Tuning

To obtain suitable parameters for our heuristics, we evaluate them on the bushy MPTP2078 problems, with definitional clausification and a timeout of 10 seconds for each problem. Before evaluation, we collect training data for machine learning heuristics by running leanCoP with a strategy schedule on all bushy problems with a timeout of 60 seconds. This solves 600 problems.

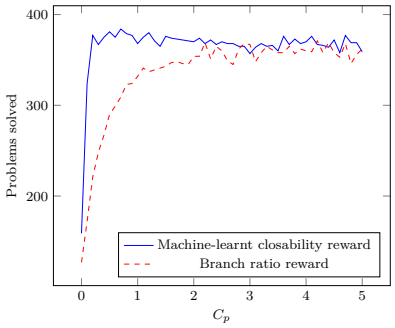
The base configuration of monteCoP uses the open branches probability (see subsection 5.3), the branch ratio reward (see subsection 5.4), and the minimal expansion policy 1 (see subsection 5.5), where the maximal simulation depth $s_{\max} = 50$, the exploration constant $C_p = 1$, and the maximal number 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 7: 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: The maximal number of MCTS iterations `maxIterations` performs best between 20 and 40, see Figure 5a: 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 5b: 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 5c. Above this value, the number of solved problems decreases, since the number of actually performed simulation steps decreases, as shown in Figure 5d. 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.

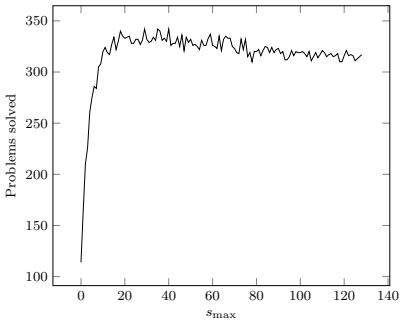
We adapt the base configuration to use the best heuristics from Table 7 and the best values for parameters discussed in Figure 5, yielding $s_{\max} = 20$, $C_p = 0.75$, and `maxIterations` $= 27$. We use this improved configuration as basis for the following evaluation.



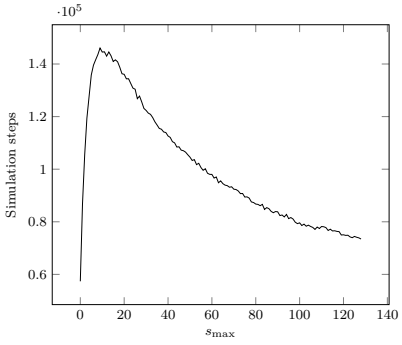
(a) Maximal number of MCTS iterations.



(b) Exploration.



(c) Maximal simulation depth.



(d) Simulation steps / Maximal simulation depth.

Fig. 5: Parameter influence.

5.8 Evaluation

We now compare the performance of FEMaLeCoP with default clause order and the improved configuration of monteCoP obtained in subsection 5.7. In the following, leanCoP/m and leanCoP/F refer to the leanCoP versions used to generate training data for monteCoP and FEMaLeCoP, respectively. We use the same evaluation methodology as in subsection 4.4: First, we run leanCoP/m and leanCoP/F on all problems for 60 seconds each, collecting training data. Next, to maximize the amount of available training data, we split the dataset for both leanCoP/m and leanCoP/F in such a way that problems that the respective prover can solve will be in its training set and unsolved problems will be in its testing set. Then, we run FEMaLeCoP and monteCoP on the testing sets corresponding to their respective underlying leanCoP versions, again for 60 seconds each.

With a timeout of 60 seconds, monteCoP solves 601 problems, compared to 563 solved by the best single leanCoP/m strategy, see Table 8. In comparison, FEMaLeCoP solves 592 problems, compared to 577 solved by the best single leanCoP/F strategy, see Table 5.

Figure 6 shows for any moment in time the total number of both training and testing problems solved up to that point. To this end, the testing data graph offsets

Table 8: Final evaluation results. Provers run with 60 seconds timeout, definitional clausification (+def), and restricted backtracking (+cut). The $\sum$ and $\bigcup$ columns are explained in subsection 4.4.

Prover	Training	Testing	$\sum$	$\bigcup$
leanCoP/m	563	0	563	653 (**+16.0%**)
monteCoP	511 (-9.2%)	90	601 (+6.7%)	
leanCoP/F	577	0	577	627 (+8.7%)
FEMaLeCoP+def	542 (-6.1%)	50	592 (+2.6%)	

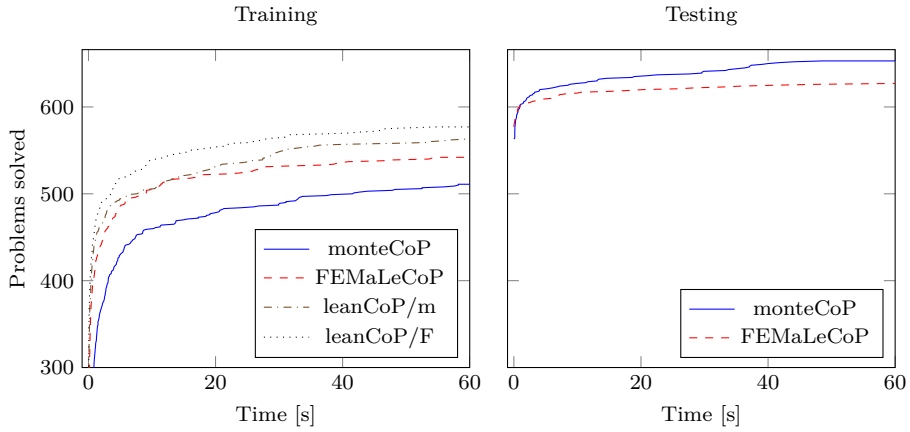


Fig. 6: Comparison of monteCoP and FEMaLeCoP. Provers run with definitional clausification (+def) and restricted backtracking (+cut)

the curves for monteCoP and FEMaLeCoP by the number of problems solved during training by leanCoP/m and leanCoP/F, respectively. Furthermore, we show on the training graph the number of training problems solved by monteCoP and FEMaLeCoP using training data from leanCoP/m and leanCoP/F.

On Figure 6, we can see that despite monteCoP's poor performance on the training set and despite the lower amount of problems solved by leanCoP/m compared to leanCoP/F, monteCoP is quickly taking the lead on the testing set compared to FEMaLeCoP. In total, the combination of leanCoP/F and FEMaLeCoP proves 627 problems, whereas leanCoP/m and monteCoP prove 653 problems. That means that in the evaluated scenario, the combination of leanCoP/m and monteCoP is more effective than leanCoP/F and FEMaLeCoP.

6 Related Work

A number of related works has already been discussed in previous sections. In particular, in section 2, we introduced the connection calculus (Bibel, 1991) as a variant of tableaux (Letz and Stenz, 2001), we discussed its implementation in the leanCoP theorem prover (Otten and Bibel, 2003), a number of improvements

introduced in the second version of leanCoP (Otten, 2008) including restricted backtracking (Otten, 2010), and the nonclausal variant of the connection calculus (Otten, 2011) together with its implementation (Otten, 2016).

The compact Prolog implementation of theorem provers following the *lean* architecture made it attractive for many experiments both with the calculus and with the implementation. The intuitionistic version of leanCoP (Otten, 2005) became the state-of-art prover for first-order problems in intuitionistic logic (Raths et al., 2007). Connections have also been considered for first-order modal logic in mleanCoP (Otten, 2014), for higher-order logic (Andrews, 1989) and for linear logic (Galmiche, 2000). Various implementation modifications can be performed very elegantly, such as search strategies, scheduling, randomization of the order of proof search steps (Raths and Otten, 2008), and internal guidance (Urban et al., 2011; Kaliszyk and Urban, 2015a).

A number of early learning and data based approaches to guide automated theorem provers has been surveyed in (Denzinger et al., 1999). The Prover9 hints method (Veroff, 1996) allows the user to specify (an often large set of) clauses to treat in a special way. A similarly working *watch list* has been later integrated in E, along with other learning mechanisms (Schulz, 2001). Using machine learning for internal guidance is historically motivated by the success of the *external guidance* methods used mainly for premise selection outside of the core ATP systems (Blanchette et al., 2016a; Urban et al., 2008; Kaliszyk and Urban, 2014). Guiding the actual proof search of ATPs using machine learning has been considered in the integration of a Naive Bayesian classifier to select next proof actions in Satallax (Färber and Brown, 2016), as well as in Enigma (Jakubův and Urban, 2017) 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 (Whalen, 2016) 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 in E (Loos et al., 2017), 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 in (Bibel, 2017).

The main use of machine learning in automated and interactive theorem provers today is to reduce original problems before the actual proof search. Machine learning based methods (Kühlwein et al., 2013; Blanchette et al., 2016b) improve on and complement the various ATP heuristics (Hoder and Voronkov, 2011) and ITP heuristics (Meng and Paulson, 2009). The problem of selecting the most useful lemmas for the given proof, referred to as "premise selection" or "relevance filtering" (Alama et al., 2014) nowadays uses syntactic similarity approaches, simple Naive Bayes and k-NN based classifiers, regression and kernel based methods (Kühlwein et al., 2012), as well as deep neural networks (Irving et al., 2016). This has become especially important in the "large theory bench" division added to the CADE Automated Systems Competition in 2008 (Sutcliffe, 2009a), with systems such as MaLARea (Urban et al., 2008) and ET (Kaliszyk et al., 2015c) achieving notable results.

7 Conclusion and Future work

We have presented our framework for integrating machine learning in connection tableaux. First, we presented translations to functional programming languages, exploring possibilities to increase the speed of proof search while keeping the implementation as simple as possible. 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. Then, we discussed machine learning integration in leanCoP via context-sensitive clause ordering and Monte Carlo Tree Search, showing that both these techniques can increase the number of solved problems, despite fewer inferences being performed.

The performed machine 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 a two-player game.

The combination of several tools that are small, simple and comprehensible can be more effective than a large, monolithic tool. While the resulting connection provers cannot yet outperform larger systems like Vampire (Kovács and Voronkov, 2013) and E (Schulz, 2013), we hope that the insight gained by experiments performed in connection provers might be used in their complex counterparts. Connection provers might be candidates for the core of future automated reasoning tools and artificial intelligence experiments.

Acknowledgements We thank the reviewers of CPP, LPAR, CADE, and JAR for their valuable comments. This work has been supported by a doctoral scholarship of the University of Innsbruck, the European Research Council (ERC) grants no. 649043 *AI4REASON* and no. 714034 *SMART*, the Czech project AI&Reasoning CZ.02.1.01/0.0/0.0/15_003/0000466, and the European Regional Development Fund.

References

Alama, Jesse, Daniel Kühlwein, and Josef Urban. 2012. Automated and human proofs in general mathematics: An initial comparison. In *LPAR-18*, eds. Nikolaj Bjørner and Andrei Voronkov. Vol. 7180 of *LNCS*, 37–45. Springer. doi:10.1007/978-3-642-28717-6_6. ISBN 978-3-642-28716-9.

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

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

Armando, Alessandro, Peter Baumgartner, and Gilles Dowek, eds. 2008. IJCAR. Vol. 5195 of *LNCS*. Springer. doi:10.1007/978-3-540-71070-7. ISBN 978-3-540-71069-1.

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

Beckert, Bernhard, Reiner Hähnle, and Peter H. Schmitt. 1993. The even more liberalized δ-rule in free variable semantic tableaux. In *Kurt gödel colloquium*, eds. Georg Gottlob, Alexander Leitsch, and Daniele Mundici. Vol. 713 of *LNCS*, 108–119. Springer. doi:10.1007/BFb0022559. ISBN 3-540-57184-1.

Berghofer, Stefan, Tobias Nipkow, Christian Urban, and Makarius Wenzel, eds. 2009. TPHOLs. Vol. 5674 of *LNCS*. Springer. doi:10.1007/978-3-642-03359-9. ISBN 978-3-642-03358-2.

Bertot, Yves. 2008. A short presentation of Coq. In *TPHOLs*, eds. Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar. Vol. 5170 of *LNCS*, 12–16. Springer. doi:10.1007/978-3-540-71067-7_3. ISBN 978-3-540-71065-3.

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

Bibel, Wolfgang. 1987. *Automated theorem proving*, 2nd edn. *Artificial intelligence*. Vieweg. http://www.worldcat.org/oclc/16641802.

Bibel, Wolfgang. 1991. Perspectives on automated deduction. In *Automated reasoning: Essays in honor of Woody Bledsoe*, ed. Robert S. Boyer. *Automated reasoning series*, 77–104. Kluwer Academic Publishers. ISBN 0-7923-1409-3.

Bibel, Wolfgang. 2017. A vision for automated deduction rooted in the connection method. In *TABLEAUX*, eds. Renate A. Schmidt and Cláudia Nalon. Vol. 10501 of *LNCS*, 3–21. Springer. doi:10.1007/978-3-319-66902-1_1. ISBN 978-3-319-66901-4.

Biere, Armin, Ioan Dragan, Laura Kovács, and Andrei Voronkov. 2014. Experimenting with SAT solvers in Vampire. In *MICAI 2014. part I*, eds. Alexander F. Gelbukh, Félix Castro q. Espinoza, and Sofía N. Galicia q. Haro. Vol. 8856 of *LNCS*, 431–442. Springer. doi:10.1007/978-3-319-13647-9_39. ISBN 978-3-319-13646-2.

Blanchette, Jasmin Christian, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. 2016a. Hammering towards QED. *J. Formalized Reasoning* 9 (1): 101–148. doi:10.6092/issn.1972-5787/4593.

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

Bove, Ana, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda - A functional language with dependent types. In *TPHOLs*, eds. Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Vol. 5674 of *LNCS*, 73–78. Springer. doi:10.1007/978-3-642-03359-9_6. ISBN 978-3-642-03358-2.

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

Brünnler, Kai, and George Metcalfe, eds. 2011. TABLEAUX. Vol. 6793 of *LNCS*. Springer. doi:10.1007/978-3-642-22119-4. ISBN 978-3-642-22118-7.

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

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

Färber, Michael, and Chad E. Brown. 2016. Internal guidance for Satallax. In *IJCAR*, eds. Nicola Olivetti and Ashish Tiwari. Vol. 9706 of *LNCS*, 349–361. Springer. doi:10.1007/978-3-319-40229-1_24. ISBN 978-3-319-40228-4.

Färber, Michael, Cezary Kaliszyk, and Josef Urban. 2017. Monte Carlo tableau proof search. In *CADE-26*, ed. Leonardo de Moura. Vol. 10395 of *LNCS*, 563–579. Springer. doi:10.1007/978-3-319-63046-5_34. ISBN 978-3-319-63045-8.

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

Giese, Martin, and Wolfgang Ahrendt. 1999. Hilbert's epsilon-terms in automated theorem proving. In *TABLEAUX*, ed. Neil V. Murray. Vol. 1617 of *LNCS*, 171–185. Springer. doi:10.1007/3-540-48754-9_17. ISBN 3-540-66086-0.

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

Hähnle, Reiner. 2001. Tableaux and related methods. In *Handbook of automated reasoning (in 2 volumes)*, eds. John Alan Robinson and Andrei Voronkov, 100–178. Elsevier and MIT Press. ISBN 0-444-50813-9.

Hales, Thomas C., 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. 2017. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi* 5. doi:10.1017/fmp.2017.1.

Harrison, John. 2009. HOL Light: An overview. In *TPHOLs*, eds. Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Vol. 5674 of *LNCS*, 60–66. Springer. doi:10.1007/978-3-642-03359-9_4. ISBN 978-3-642-03358-2.

Hilbert, David, and Paul Bernays. 1939. *Grundlagen der Mathematik. II.* Vol. 50 of *Die Grundlehren der mathematischen Wissenschaften*. Springer.

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

Hoder, Kryštof, and Andrei Voronkov. 2011. Sine qua non for large theory reasoning. In *CADE-23*, eds. Nikolaj Bjørner and Viorica Sofronie q. Stokkermans. Vol. 6803 of *LNCS*, 299–314. Springer. doi:10.1007/978-3-642-22438-6_23. ISBN 978-3-642-22437-9.

Hoder, Kryštof, Giles Reger, Martin Suda, and Andrei Voronkov. 2016. Selecting the selection. In *IJCAR*, eds. Nicola Olivetti and Ashish Tiwari. Vol. 9706 of *LNCS*, 313–329. Springer. doi:10.1007/978-3-319-40229-1_22. ISBN 978-3-319-40228-4.

Hurd, Joe. 2003. First-order proof tactics in higher-order logic theorem provers. In *Design and application of strategies/tactics in higher order logics (STRATA)*, eds. Myla Archer, Ben Di Vito, and César Muñoz. *NASA technical reports*, 56–68. http://www.gilith.com/research/papers.

Irving, Geoffrey, Christian Szegedy, Alexander A. Alemi, Niklas Eén, François Chollet, and Josef Urban. 2016. DeepMath - deep sequence models for premise selection. In *NIPS*, eds. Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, 2235–2243. http://papers.nips.cc/paper/6280-deepmath-deep-sequence-models-for-premise-selection.

Jakubův, Jan, and Josef Urban. 2017. ENIGMA: efficient learning-based inference guiding machine. In *CICM*, eds. Herman Geuvers, Matthew England, Osman Hasan, Florian Rabe, and Olaf Teschke. Vol. 10383 of *LNCS*, 292–302. Springer. doi:10.1007/978-3-319-62075-6_20. ISBN 978-3-319-62074-9.

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

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

Kaliszyk, Cezary, and Josef Urban. 2015a. FEMaLeCoP: Fairly efficient machine

learning connection prover. In *LPAR-20*, eds. Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov. Vol. 9450 of *LNCS*, 88–96. Springer. doi:10.1007/978-3-662-48899-7_7. ISBN 978-3-662-48898-0.

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

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

Kaliszyk, Cezary, Josef Urban, and Jiří Vyskočil. 2015a. Certified connection tableaux proofs for HOL Light and TPTP. In *CPP*, eds. Xavier Leroy and Alwen Tiu, 59–66. ACM. doi:10.1145/2676724.2693176. ISBN 978-1-4503-3296-5.

Kaliszyk, Cezary, Josef Urban, and Jiří Vyskočil. 2015b. Efficient semantic features for automated reasoning over large theories. In *IJCAI*, eds. Qiang Yang and Michael Wooldridge, 3084–3090. AAAI Press. ISBN 978-1-57735-738-4. http://ijcai.org/Abstract/15/435.

Kaliszyk, Cezary, Stephan Schulz, Josef Urban, and Jiří Vyskočil. 2015c. System description: E.T. 0.1. In *CADE-25*, eds. Amy P. Felty and Aart Middeldorp. Vol. 9195 of *LNCS*, 389–398. Springer. doi:10.1007/978-3-319-21401-6_27. ISBN 978-3-319-21400-9.

Kocsis, Levente, and Csaba Szepesvári. 2006. Bandit based Monte-Carlo planning. In *ECML*, eds. Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Vol. 4212 of *LNCS*, 282–293. Springer. doi:10.1007/11871842_29. ISBN 3-540-45375-X.

Kovács, Laura, and Andrei Voronkov. 2013. First-order theorem proving and Vampire. In *CAV*, eds. Natasha Sharygina and Helmut Veith. Vol. 8044 of *LNCS*, 1–35. Springer. doi:10.1007/978-3-642-39799-8_1. ISBN 978-3-642-39798-1.

Kühlwein, Daniel, Twan van Laarhoven, Evgeni Tsivtsivadze, Josef Urban, and Tom Heskes. 2012. Overview and evaluation of premise selection techniques for large theory mathematics. In *IJCAR*, eds. Bernhard Gramlich, Dale Miller, and Uli Sattler. Vol. 7364 of *LNCS*, 378–392. Springer. doi:10.1007/978-3-642-31365-3_30. ISBN 978-3-642-31364-6.

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

Letz, Reinhold, and Gernot Stenz. 2001. Model elimination and connection tableau procedures. In *Handbook of automated reasoning (in 2 volumes)*, eds. John Alan Robinson and Andrei Voronkov, 2015–2114. Elsevier and MIT Press. ISBN 0-444-50813-9.

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

Loos, Sarah M., Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. 2017. Deep network guided proof search. In *LPAR-21*, eds. Thomas Eiter and David Sands. Vol. 46 of *Epic series in computing*, 85–105. EasyChair. http://www.easychair.org/publications/paper/340345.

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

Meng, Jia, and Lawrence C. Paulson. 2009. Lightweight relevance filtering for machine-generated resolution problems. *J. Applied Logic* 7 (1): 41–57.

doi:10.1016/j.jal.2007.07.004.

Mohamed, Otmane Aït, César A. Muñoz, and Sofiène Tahar, eds. 2008. TPHOLs. Vol. 5170 of *LNCS*. Springer. doi:10.1007/978-3-540-71067-7. ISBN 978-3-540-71065-3.

Nonnengart, Andreas. 1996. Strong skolemization, Research Report MPI-I-96-2-010, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany.

Olivetti, Nicola, and Ashish Tiwari, eds. 2016. IJCAR. Vol. 9706 of *LNCS*. Springer. doi:10.1007/978-3-319-40229-1. ISBN 978-3-319-40228-4.

Otten, Jens. 2005. Clausal connection-based theorem proving in intuitionistic first-order logic. In *TABLEAUX*, ed. Bernhard Beckert. Vol. 3702 of *LNCS*, 245–261. Springer. doi:10.1007/11554554_19. ISBN 3-540-28931-3.

Otten, Jens. 2008. leanCoP 2.0 and ileanCoP 1.2: High performance lean theorem proving in classical and intuitionistic logic (system descriptions). In *IJCAR*, eds. Alessandro Armando, Peter Baumgartner, and Gilles Dowek. Vol. 5195 of *LNCS*, 283–291. Springer. doi:10.1007/978-3-540-71070-7_23. ISBN 978-3-540-71069-1.

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

Otten, Jens. 2011. A non-clausal connection calculus. In *TABLEAUX*, eds. Kai Brünnler and George Metcalfe. Vol. 6793 of *LNCS*, 226–241. Springer. doi:10.1007/978-3-642-22119-4_18. ISBN 978-3-642-22118-7.

Otten, Jens. 2014. Mleancop: A connection prover for first-order modal logic. In *IJCAR*, eds. Stéphane Demri, Deepak Kapur, and Christoph Weidenbach. Vol. 8562 of *LNCS*, 269–276. Springer. doi:10.1007/978-3-319-08587-6_20. ISBN 978-3-319-08586-9.

Otten, Jens. 2016. nanoCoP: A non-clausal connection prover. In *IJCAR*, eds. Nicola Olivetti and Ashish Tiwari. Vol. 9706 of *LNCS*, 302–312. Springer. doi:10.1007/978-3-319-40229-1_21. ISBN 978-3-319-40228-4.

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

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

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

Ramakrishnan, I. V., R. C. Sekar, and Andrei Voronkov. 2001. Term indexing. In *Handbook of automated reasoning (in 2 volumes)*, eds. John Alan Robinson and Andrei Voronkov, 1853–1964. Elsevier and MIT Press. ISBN 0-444-50813-9.

Raths, Thomas, and Jens Otten. 2008. randoCoP: Randomizing the proof search order in the connection calculus. In *PAAR*, eds. Boris Konev, Renate A. Schmidt, and Stephan Schulz. Vol. 373 of *CEUR workshop proceedings*. CEUR-WS.org. http://ceur-ws.org/Vol-373/paper-08.pdf.

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

Robinson, John Alan, and Andrei Voronkov, eds. 2001. *Handbook of automated reasoning (in 2 volumes)*. Elsevier and MIT Press.

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

Schadd, Maarten P. D., Mark H. M. Winands, Mandy J. W. Tak, and Jos W. H. M. Uiterwijk. 2012. Single-player Monte-Carlo tree search for SameGame. *Knowl.-*

Based Syst. 34: 3–11. doi:10.1016/j.knosys.2011.08.008.

Schulz, Stephan. 2001. Learning search control knowledge for equational theorem proving. In *KI*, eds. Franz Baader, Gerhard Brewka, and Thomas Eiter. Vol. 2174 of *LNCS*, 320–334. Springer. doi:10.1007/3-540-45422-5_23. ISBN 3-540-42612-4.

Schulz, Stephan. 2013. System description: E 1.8. In *LPAR-19*, eds. Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov. Vol. 8312 of *LNCS*, 735–743. Springer. doi:10.1007/978-3-642-45221-5_49. ISBN 978-3-642-45220-8.

Silver, David, 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. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529 (7587): 484–489. doi:10.1038/nature16961.

Slind, Konrad, and Michael Norrish. 2008. A brief overview of HOL4. In *TPHOLs*, eds. Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar. Vol. 5170 of *LNCS*, 28–32. Springer. doi:10.1007/978-3-540-71067-7_6. ISBN 978-3-540-71065-3.

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

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

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

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

Sutcliffe, Geoff. 2016b. The CADE ATP system competition - CASC. *AI Magazine* 37 (2): 99–101. doi:10.1609/aimag.v37i2.2620.

Tseitin, Gregory S. 1983. On the complexity of derivation in propositional calculus. In *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, eds. Jörg H. Siekmann and Graham Wrightson, 466–483. Springer. doi:10.1007/978-3-642-81955-1_28. ISBN 978-3-642-81955-1.

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

Urban, Josef, Kryštof Hoder, and Andrei Voronkov. 2010. Evaluation of automated theorem proving on the Mizar Mathematical Library. In *ICMS*, eds. Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama. Vol. 6327 of *LNCS*, 155–166. Springer. doi:10.1007/978-3-642-15582-6_30. ISBN 978-3-642-15581-9.

Urban, Josef, Jiří Vyskočil, and Petr Štěpánek. 2011. MaLeCoP machine learning connection prover. In *TABLEAUX*, eds. Kai Brünnler and George Metcalfe. Vol. 6793 of *LNCS*, 263–277. Springer. doi:10.1007/978-3-642-22119-4_21. ISBN 978-3-642-22118-7.

Urban, Josef, Geoff Sutcliffe, Petr Pudlák, and Jiří Vyskočil. 2008. MaLARea SG1-machine learner for automated reasoning with semantic guidance. In *IJCAR*, eds. Alessandro Armando, Peter Baumgartner, and Gilles Dowek. Vol. 5195 of *LNCS*, 441–456. Springer. doi:10.1007/978-3-540-71070-7_37. ISBN 978-3-540-71069-1.

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

Wenzel, Makarius, Lawrence C. Paulson, and Tobias Nipkow. 2008. The Isabelle framework. In *TPHOLs*, eds. Otmane Aït Mohamed, César A. Muñoz, and Sofiène

Tahar. Vol. 5170 of *LNCS*, 33–38. Springer. doi:10.1007/978-3-540-71067-7_7. ISBN 978-3-540-71065-3.

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