Noname manuscript No.
(will be inserted by the editor)

TacticToe: Learning to Prove with Tactics

Thibault Gauthier · Cezary Kaliszyk · Josef Urban · Ramana Kumar · Michael Norrish

the date of receipt and acceptance should be inserted later

Abstract We implement a automated tactical prover TacticToe on top of the HOL4 interactive theorem prover. TacticToe learns from human proofs which mathematical technique is suitable in each proof situation. This knowledge is then used in a Monte Carlo tree search algorithm to explore promising tactic-level proof paths. On a single CPU, with a time limit of 60 seconds, TacticToe proves 66.4% of the 7164 theorems in HOL4's standard library, whereas E prover with auto-schedule solves 34.5%. The success rate rises to 69.0% by combining the results of TacticToe and E prover.

1 Introduction

Many of the state-of-the-art interactive theorem provers (ITPs) such as HOL4 [30], HOL Light [14], Isabelle [34] and Coq [2] provide high-level parameterizable tactics for constructing proofs. Tactics analyze the current proof state (goal and assumptions) and apply non-trivial proof transformations. Formalized proofs take advantage of different levels of automation which are in increasing order of generality: specialized rules, theory-based strategies and general purpose strategies. Thanks to progress in proof automation, developers can delegate more and more complicated proof obligations to general purpose strategies. Those are implemented by automated theorem provers (ATPs) such as E prover [28]. Communication between an ITP and ATPs is made possible by a "hammer" system [3,10]. It acts as an interface by performing premise selection, translation and proof reconstruction. Yet, ATPs are not flawless and more precise user-guidance, achieved by applying a particular sequence of specialized rules, is almost always necessary to develop a mathematical theory.

Thibault Gauthier and Cezary Kaliszyk
Department of Computer Science, University of Innsbruck, Innsbruck, Austria
{thibault.gauthier,cezary.kaliszyk}@uibk.ac.at

Josef Urban
Czech Technical University, Prague
josef.urban@gmail.com

Ramana Kumar and Michael Norrish
Data61

In this work, we develop in the ITP HOL4 a procedure to select suitable tactics depending on the current proof state by learning from previous proofs. Backed by this machine-learned guidance, our prover TacticToe executes a *Monte Carlo tree search* [6] algorithm to find sequences of tactic applications leading to an ITP proof.

1.1 The problem

An ITP is a development environment for the construction of formal proofs in which it is possible to write a proof as a sequence of applications of primitive inference rules. This approach is not preferred by developers because proving a valuable theorem requires often many thousands of primitive inferences. The preferred approach is to use and devise high-level tools and automation that abstract over useful ways of producing proof pieces. A specific, prevalent instance of this approach is the use of *tactics* for *backward* or goal-directed proof. Here, the ITP user operates on a proof state, initially the desired conjecture, and applies tactics that transform the proof state until it is solved. Each tactic performs a sound transformation of the proof state: essentially, it promises that if the resulting proof state can be solved, then so can the initial one. This gives rise to a machine learning problem: can we learn a mapping from a given proof state to the next tactic (or sequence of tactics) that will productively advance the proof towards a solution?

Goals and theorems. A goal (or proof state) is represented as a *sequent*. A sequent is composed of a set of assumptions and a conclusion, all of which are higher-order logic formulas [12]. When a goal is proven, it becomes a theorem. And when the developer has given the theorem a name, we refer to it as a *top-level* theorem.

Theories and script files. Formal developments in HOL4 are organized into named theories, which are implemented by *script* files defining modules in the functional programming language SML. Each script file corresponds to a single theory, and contains definitions of types and constants as well as statements of theorems together with their proofs. In practice, tactic-based proofs are written in an informally specified *embedded domain-specific language*: the language consisting of pre-defined tactics and *tacticals* (functions that operate on tactics). However, the full power of SML is always available for users to define their own tactics or combinations thereof.

Tactics in HOL4. A *tactic* is a function that takes a goal (or proof obligation) and returns a list of goals (subgoals of the original goal) together with a *validation function*. Calling the validation function on a list of *theorems*, corresponding to the list of subgoals, results in a theorem corresponding to the original goal. For example, if the list of subgoals is empty, then calling the validation function on the empty list yields the original goal sequent as a theorem. In this way, tactics implement the plumbing of goal-directed proof construction, transforming the conjecture by reasoning in a backward manner. Since validation functions are only important to check the final proof, we omit them during the description of the proof search algorithm. We denote by $t(g)$ the list of goals produced by the application of a tactic t to a goal g.

1.2 Contributions

This paper extends work described previously [11], in which we proposed the idea of a tactic-based prover based on supervised learning guidance. We achieved a 39% success rate on theorems of the HOL4 standard library by running TacticToe with a 5 second timeout. The contributions of this paper and their effect on our system are:

- Monte Carlo tree search (MCTS) replaces A* as our search algorithm (Section 4). The MCTS algorithm gives more balanced feedback on the success of each proof step by comparing subtrees of the search tree instead of leaves. The policy and value function are learned through supervised learning.
- Proof guidance required by MCTS is given by prediction algorithms through supervised learning (Section 3). These predictors are implemented for three kinds of objects: tactics, theorems, and lists of goals.
- We introduce an orthogonalization process that eliminates redundant tactics (Section 6).
- We introduce a tactic abstraction mechanism (Section 7), which enables us to create more general and flexible tactics by dynamically predicting tactic arguments.
- The internal ATP Metis is complemented by asynchronous calls to E prover (Section 4.3) to help TacticToe during proof search.
- Proof recording at the tactic level is made more precise (Section 8). We now support pattern matching constructions and opening SML modules.
- Evaluation of TacticToe with a 60 seconds timeout achieves a 66% success rate on the standard library. Comparisons between TacticToe and E prover on different type of problems are reported in Section 9.
- Minimization and embellishment of the discovered proof facilitates user interaction (Section 10).

1.3 Plan

The different components depicted in Figure 1 give a high-level overview of TacticToe and our approach to generating tactic-level proofs by learning from recorded knowledge. On one side, we have the learning aspect of TacticToe whose purpose is to produce a high quality function to predict a tactic given a goal state. Tactic prediction uses a knowledge base that is gleaned, in the first place, from human-written proofs. On the other side, we have the proving aspect of TacticToe, which uses tactic prediction to guide a proof search algorithm in the context of a given conjecture.

This paper is organized firstly around the proving aspect: We define the proof search tree (Section 2), explain the essence of our approach to learning to predict tactics (Section 3), and then present the prediction-guided proof search algorithm (Section 4). We also describe our preselection method (Section 5) for speeding up prediction during search. Afterwards, we delve into some details of the learning aspect of TacticToe: We describe approaches to improving the knowledge base that supports prediction (orthogonalization in Section 6 and abstraction in Section 7), and describe how we build the knowledge base in the first place by extracting and recording tactic invocations from a library of human-written proofs (Section 8).

In Section 9, we present an evaluation of TacticToe on a large set of theorems originating from HOL4. Finally, we present a feature of TacticToe that makes it

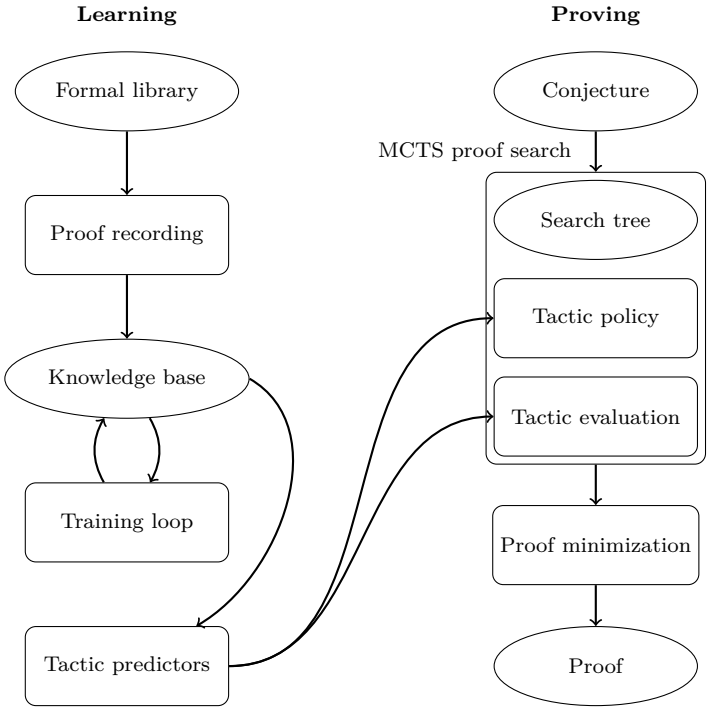

Fig. 1 Relation between modules of the learning and proving toolchains. Functions are represented by rectangles and objects by ellipses.

more suitable for use by human provers, namely, minimization and prettification (Section 10) and compare TacticToe's generated proofs to human proofs (Section 11).

2 Search Tree

The application of a tactic to an initial conjecture produces a list of goals. Each of the created goals can in turn be the input to other tactics. Moreover, it is possible to try multiple tactics on the same goal. In order to keep track of progress made from the initial conjecture, we organize goals and tactics into a graph with lists of goals as nodes and tactics as edges (See Definition 1).

In this section, we only give a description of the graph at a given moment of the search, after some number of tactic applications. Construction of the tree is done by the MCTS algorithm in Section 4. The MCTS algorithm is guided by prediction algorithms presented in Section 3.

Definition 1 (search tree)
A search tree $\mathfrak{T}$ is a sextuple $(\mathbb{T}, \mathbb{G}, \mathbb{A}, T, G, A)$ that respects the following conditions:

– $\mathbb{T}$ is a set of tactics, $\mathbb{G}$ is a set of goals and $\mathbb{A}$ is a set of nodes representing lists of goals. All objects are tagged with their position in the tree (see Figure 2).

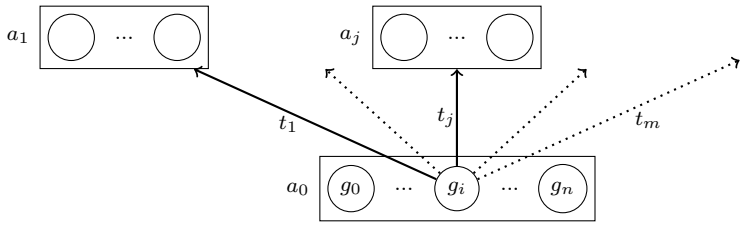

Fig. 2 A node of a search tree a_0, the list of goals $g_0 \dots g_n$ it contains and the nodes $a_1 \dots a_m$ derived from the application of the tactics $t_1 \dots t_m$ to g_i.

- T is a function from $\mathbb{G}$ to $\mathcal{P}(\mathbb{T})$. It takes a goal and return the set of tactics already applied to this goal.
- G is a function from $\mathbb{A}$ to $\mathcal{P}(\mathbb{G})$. It takes a node and returns the list of goals of this node.
- A is a function that takes a pair $(g,t) \in \mathbb{G} \times \mathbb{T}$ such as $t \in T(g)$ and returns a node $A(g,t)$ such that $t(g) = G(A(g,t))$. In other words, the output $t(g)$ is exactly the list of goals contained in $A(g,t)$.
- It is acyclic, i.e., a node cannot be a strict descendant (see Definition 3) of itself.
- There is exactly one node with no parents. This root node contains exactly one goal which is the initial goal (conjecture).

If no explicit order is given, we assume that the sets $\mathbb{T}, \mathbb{G}, \mathbb{A}$ are equipped with an arbitrary *total order*. Figure 2 depicts part of a search tree. Goals, nodes and tactics are represented respectively by circles, rectangles and arrows.

In the following, properties are defined in the context of a search tree $\mathfrak{T}$. In Definition 2, we specify different states for a goal: open, pending or solved. There, we define what a solved goal and solved node are by mutual recursion on the number of steps it takes to solve a goal and a node.

Definition 2 (solved goal, solved nodes, open goal, pending goal)
The set of *solved nodes* $\mathbb{A}^*$ and the set of *solved goals* $\mathbb{G}^*$ are defined inductively by:

$$\begin{aligned}
\mathbb{A}^0 &=_{\text{def}} \{a \in \mathbb{A} \mid G(a) = \emptyset\} \\
\mathbb{G}^0 &=_{\text{def}} \{g \in \mathbb{G} \mid \exists t \in T(g).\ A(g,t) \in \mathbb{A}^0\} \\
\mathbb{A}^{n+1} &=_{\text{def}} \{a \in \mathbb{A} \mid \forall g \in G(a).\ g \in \mathbb{G}^n\} \\
\mathbb{G}^{n+1} &=_{\text{def}} \{g \in \mathbb{G} \mid \exists t \in T(g).\ A(g,t) \in \mathbb{A}^{n+1}\} \\
\mathbb{A}^* &=_{\text{def}} \bigcup_{n \in \mathbb{N}} \mathbb{A}^n \qquad \mathbb{G}^* =_{\text{def}} \bigcup_{n \in \mathbb{N}} \mathbb{G}^n
\end{aligned}$$

We call a goal *unsolved* if it is not in the set of solved goals. Similarly, we call a node *unsolved* if it does not belong to the set of solved nodes. The *open goal* of a node is the first unsolved goal of this node according to some preset order. The other unsolved nodes are called *pending nodes*. If a node is unsolved, then it contains a unique open goal. During MCTS exploration, we do not explore pending goals of an unsolved node before its open goal is solved which justifies our terminology. In

other words, our search tree has the following property: if g is a pending node then $T(g)$ is empty.

In Definition 3, we define relations between goals and nodes in terms of parenthood.

Definition 3 (children, descendant, ancestors)
The *children of a goal* g is a set of nodes defined by:

$$\mathit{Children}(g) = \{A(g,t) \in \mathbb{A} \mid t \in T(g)\}$$

By extension, we define the *children of a node* a by:

$$\mathit{Children}(a) = \bigcup_{g \in G(a)} \mathit{Children}(g)$$

The *descendants of a node* a by:

$$\begin{aligned} \mathit{Descendants}^0(a) &=_{\text{def}} \{a\} \\ \mathit{Descendants}^{n+1}(a) &=_{\text{def}} \bigcup_{a' \in \mathit{Descendants}^n(a)} \mathit{Children}(a') \\ \mathit{Descendants}(a) &=_{\text{def}} \bigcup_{n \in \mathbb{N}} \mathit{Descendants}^n(a) \end{aligned}$$

And the *ancestors of a node* a by:

$$\mathit{Ancestors}(a) =_{\text{def}} \{b \in \mathbb{A} \mid a \in \mathit{Descendants}(b)\}$$

In Definition 4, we decide if a tactic is productive based on its contribution to the search tree.

Definition 4 (productive tactic) The application of a tactic t on an open goal g is called *productive* if and only if all the following conditions are satisfied:

- It does not fail or timeout. To prevent tactics from looping, we interrupt tactics after a short amount of time (0.05 seconds).
- It does not loop, i.e., its output does not contain any goals that appear in the ancestor of the node of the current goal.
- It is not a redundant step, i.e., there does not exist t' in $T(g)$ such as $t'(g) \subseteq t(g)$.

The third point of the definition is a partial attempt at preventing confluent branches. The general case where two branches join after n steps is handled by a tactic cache which memorizes tactic effects on goals. This cache allows faster re-exploration of confluent branches which remain separated in the search tree.

3 Prediction

The learning-based selection of relevant lemmas significantly improves the automation in hammer systems [4]. In TacticToe we use the distance-weighted k *nearest-neighbour* classifier [8] adapted for lemma selection [18]. It allows for hundreds of predictions per second, while maintaining very high prediction quality [20]. We rely on this supervised machine learning algorithm to predict objects of three kinds: tactics, theorems and lists of goals. The number of predicted objects k is called the *prediction radius*. For a goal g, we denote by $Predictions_k^{tactic}(g)$ (respectively $Predictions_k^{theorem}(g)$ and $Predictions_k^{goal\_list}(g)$) the k tactics (respectively theorems and lists of goals) selected by our prediction algorithm.

We discuss below the specifics associated with the prediction of each kind of object. In particular, we present the dataset from which objects are selected and the purpose of the selection process. The similarity measure backing the predictions is described in Section 3.1 and three methods for improving the speed and quality of the predictions are presented in Sections 5, 6 and 7.

Tactics To start with, we build a database of tactics consisting of goal-tactic pairs recorded from successful tactic applications in human proofs (recording will be discussed in Section 8). During proof search, the recorded goals will be reordered according to their similarity with a target goal g (usually the open goal of a node). The recorded pairs and their similarity to g induce an order on tactics. Intuitively, tactics which have been successful on goals similar to g should be tried first. Indeed, they are more likely to lead to a proof. This predicted tactic order is then transformed into a prior policy of our MCTS algorithm (see Section 4.2). Tactic selection is also used to improve the quality of the database of tactics during orthogonalization (see Section 6).

Theorems as Arguments of Tactics We first collect a set of theorems for our predictor to select from. It includes the HOL4 theorem database and theorems from the local namespace. Predicted tactics during proof search and orthogonalization may include tactics where arguments (lists of theorems) have been erased (see Section 7). We instantiate those tactics by theorems that are syntactically the closest to the goal as they are more likely to help. The same algorithm selects suitable premises for ATPs integrated with TacticToe (see Section 4.3).

Lists of Goals as Output of Tactics We compile a dataset of tactic outputs during orthogonalization (see Section 6). Some elements of this set will be considered positive examples (see Section 4.2). During proof search, given a list of goals l created by a tactic, we select a set of lists of goals that are most similar to l. The ratio of positive examples in the selection gives us a prior evaluation for MCTS.

3.1 Feature Extraction and Similarity Measures

We predict tactics through their associated goals. So, the features for each kind of object can be extracted from different representations of mathematical formulas. We start by describing features for HOL4 terms. From there, we extend the extraction mechanism to goals, theorems and lists of goals. Duplicated features are always

removed so that each object has an associated set of features. Here are the kinds of features we extract from terms:

- names of constants, including the logical operators,
- type constructors present in the types of constants and variables,
- subterms with all variables replaced by a single placeholder V,
- names of the variables.

Goals (respectively theorems) are represented by pairs consisting of a list of terms (assumptions) and a term (conclusion). We distinguish between features of the assumptions and features of the conclusion by adding a different tag to elements of each set. The feature of a goal (respectively theorem) are the union of all these tagged features. From that, we can construct features for a list of goals by computing the union of the features of each goal in the list.

We estimate the similarity between two objects o_1 and o_2 through their respective feature sets f_1 and f_2. The estimation is based on the frequency of the features in the intersection of f_1 and f_2. A good rule of thumb is that the rarer the shared features are, the more similar two goals should be. The relative rarity of each feature can be estimated by calculating its TF-IDF weight [17]. We additionally raise these weights to the sixth power giving even more importance to rare features [4].

The first similarity measure sim_1 sums the weight of all shared features to compute the total score. In a second similarity measure sim_2, we additionally take the total number of features into account, to reduce the seemingly unfair advantage of big feature sets in the first scoring function.

$$sim_1(o_1, o_2) = \sum\nolimits_{f \in f_0 \cap f_1} \mathrm{tf\_idf}(f)^6$$

$$sim_2(o_1, o_2) = \frac{\sum_{f \in f_0 \cap f_1} \mathrm{tf\_idf}(f)^6}{ln(e + |f_0| + |f_1|)}$$

In our setting, making predictions for an object o consists of sorting a set of objects by their similarity to o. We use sim_1 for tactics and theorems and sim_2 for lists of goals. The reason why sim_1 is used for predicting tactics is that tactics effective on a large goal g are often also suitable for a goal made of sub-formulas of g. The same monotonicity heuristic can justify the use of sim_1 for theorems. Indeed, in practice a large theorem is often a conjunction of theorems from the same domain. And if a conjunction contains a formula related to a problem, then the other conjuncts from the same domain may also help to solve it.

4 Proof Search

Our prediction algorithms are not always accurate. Therefore, a selected tactic may fail, proceed in the wrong direction or even loop. For that reason, predictions need to be accompanied by a proof search mechanism that allows for backtracking and can choose which proof tree to extend next and in which direction. Our search algorithm is a Monte Carlo tree search [6] algorithm that relies on a prior evaluation function and a prior policy function. Those priors are learned through direct supervised learning for the policy and via the data accumulated during orthogonalization for the evaluation. Therefore, this MCTS algorithm does not need to rely on roll outs for

evaluation. The policy and evaluation are estimated by the simple supervised k-NN algorithm. A characteristic of the MCTS algorithm is that it offers a good trade-off between exploitation and exploration. This means that the algorithm searches deeper the more promising branches and leaves enough time for the exploration of less likely alternatives.

Remark 1 Neural networks trained through reinforcement learning can be very effective for approximating the policy and evaluation as demonstrated in .e.g. AlphaGo Zero [29]. But training neural networks is computationally expensive and has not yet been proven effective in our context. That is why we chose a simpler machine learning model for our project.

We first describe the proof search algorithm and explain later how to compute the prior policy `PriorPolicy` and prior evaluation `PriorEvaluation` functions.

4.1 Proof exploration

The proof search starts by creating a search tree during initialization. The search tree then evolves by the repetitive applications of MCTS steps. A step in the main loop of MCTS consists of three parts: node selection, node extension, and backpropagation. And the decision to stop the loop is taken by the resolution module.

Initialization The input of the algorithm is a goal g (also called conjecture) that we want to prove. Therefore the search tree starts with only one node containing the list of goals $[g]$.

Node selection Through node extension steps the search tree grows and the number of paths to explore increases. To decide which node to extend next, the MCTS algorithm computes for each node a value (see Definition 5) that changes after each MCTS step. The algorithm that performs node selection starts from the root of the search tree. From the current node, the function `CompareChildren` chooses the child node with the highest value among the children of its open goal.

If the highest children value is higher than the *widening policy* (see Section 4.2), then the selected child becomes the current node, otherwise the final result of node selection is the current node. The following pseudo-code illustrates our node selection algorithm:

```
CurrentNode = Root(Tree);
while true do
  if Children(CurrentNode) = ∅ then break;
  (BestChild, BestValue) = CompareChildren (CurrentNode);
  if WideningPolicy (ChosenNode) ≥ BestValue then break;
  CurrentNode = BestChild
end;
return CurrentNode;
```

Definition 5 (Value)
The value of the i^{th} child a_i of the open goal of a parent node p is determined by:

$$Value(a_i) = CurEvaluation(a_i) + c_{exploration} * Exploration(a_i)$$

The current evaluation $CurEvaluation(a_i)$ is the average evaluation of all descendants of a_i including node extension failures.

$$CurEvaluation(a_i) = \sum_{a' \in Descendants(a_i)} \frac{PriorEvaluation(a')}{|Descendants(a_i)| + Failure(a_i)}$$

where the number $Failure(a_i)$ is the number of failures that occurred during node extension from descendants of a_i .

The exploration term is determined by the prior policy and the current policy. The current policy is calculated from the number of times a node x has been traversed during node selection, denoted $Visit(x)$.

$$Exploration(a_i) = \frac{PriorPolicy(a_i)}{CurPolicy(a_i)}$$

$$CurPolicy(a_i) = \frac{1 + Visit(a_i)}{\sqrt{Visit(p)}}$$

The policy can be seen as a skewed percentage of the number of times a node was visited. The square root favors exploration of nodes with few visits. The coefficient $c_{exploration}$ is experimentally determined and adjusts the trade-off between exploitation and exploration.

Node extension Let a be the result of node selection. If a is a solved node or the descendant of a solved node, then extending a would not be productive and the algorithm reports a failure for this MCTS step. If a is not solved, it applies the best untested tactic t on the open goal g of this node according to the prediction order for g. If no such tactic exists or if t is not productive, the algorithm returns a failure. If node extension succeeds, a new node containing $t(g)$ is added to the children of a.

Backpropagation During backpropagation we update the statistics of all the nodes traversed or created during this MCTS step:

- Their visit number is incremented.
- If node extension failed, their failure count is incremented.
- If node extension succeeded, they inherit the evaluation of the created child.

These changes update the current evaluation and the current policy of the traversed nodes. After completing backpropagation, the process loops back to node selection.

Resolution The search ends when the algorithm reaches one of these 3 conditions:

- It finds a proof, i.e., the root node is solved. In this case, the search returns a minimized and prettified tactic proof (see Section 10).
- It saturates, i.e., there are no tactics to be applied to any open goal. This occurs less than 0.1 percent of the time in the full-scale experiment. And this happens only at the beginning of the HOL4 library, where the training data has very few tactics.
- It exceeds a given time limit (60 seconds by default).

4.2 Supervised learning guidance

Our MCTS algorithm relies on the guidance of two supervised learning methods. The prior policy estimates the probability of success of a tactic t on a goal g and associate it with the potential children that would be created by the application of t to g. The prior evaluation judges if a branch is fruitful by analyzing tactic outputs. Both priors influence the rates at which branches of the search tree are explored.

Prior policy Similarity between goals produced by the tactic predictor are hard to translate into a prior policy, so we rely solely on the prediction order to estimate the probability with which each tactic should be tried. Let $\mathfrak{T}$ be a search tree after a number of MCTS steps. Let p be a selected node in $\mathfrak{T}$ and g its open goal. We order the list of n productive tactics already applied to g by their similarity score. We note the resulting list $t_0, \ldots, t_{n-1}$. Let c_{policy} be a constant heuristic optimized during our experiments. We calculate the prior policy of a child a_i produced by the tactic t_i by:

$$PriorPolicy(a_i) = (1 - c_{policy})^i * c_{policy}$$

In order to include the possibly of trying more tactics on g we define the widening policy on the parent p for its open goal g to be:

$$WideningPolicy(p) = (1 - c_{policy})^n * c_{policy}$$

Prior evaluation We now concentrate on the definition of a reasonable evaluation function for the output of tactics. Intuitively, a list of goals is worth exploring further if we believe there exists a short proof for it. We could estimate the likelihood of finding such a proof from previous proof searches. However, extracting lists of goals from all proof attempts creates too much data which slows down the prediction algorithm. So, we only collect outputs of tactics tested during orthogonalization (see Section 6). We declare a list of goals l to be *positive* if it has been produced by the winner of an orthogonalization competition. The set of positive examples in $Predictions_k^{goal\_list}(l)$ is denoted $Positive_k^{goal\_list}(l)$. We chose $k = 10$ as a default evaluation radius in our experiments. And we evaluate a node a through the list of goals $G(a)$ it contains using the prior evaluation function:

$$PriorEvaluation_k(a) = \frac{|Positive_k^{goal\_list}(G(a))|}{k}$$

4.3 ATP Integration

General-purpose proof automation mechanisms which combine proof translation to ATPs with machine learning ("hammers") have become quite successful in enhancing the automation level in proof assistants [3]. Since external automated reasoning techniques sometimes outperform the combined power of tactics, we combine the TacticToe search with general purpose provers such as the ones found in HOL(y)Hammer for HOL4 [10].

HOL4 already integrates a superposition-based prover Metis [16] for this purpose. It is already recognized by the tactic selection mechanism in TacticToe and thanks to abstraction, its premises can be predicted dynamically. Nevertheless, we think that the performance of TacticToe can be boosted by giving the ATP Metis a special status among tactics. This arrangement consists of always predicting premises for Metis, giving it a slightly higher timeout and trying it first on each open goal. Since Metis does not create new goals, these modifications only induce an overhead that increases linearly with the number of nodes.

In the following, we present the implementation of asynchronous calls to E prover [28] during TacticToe's proof search. Other external ATPs can be integrated with TacticToe in a similar manner.

First, we develop a tactic which expects a list of premises as an argument and calls E prover on the goal together with the premises translated to a first-order problem. If E prover succeeds on the problem, the premises used in the external proof are used to reconstruct the proof inside HOL4 with Metis. Giving a special status to E prover is essential as E prover calls do not appear in human-written tactic proofs. Here, we use an even higher timeout (5 seconds) and a larger number of predicted premises (128). We also try E prover first every time a new goal is created. Since calls to E prover are computationally expensive, they are run in parallel and asynchronously. To update the MCTS values, the result of each thread is back-propagated in the tree after completion. This avoids slowing down TacticToe's search loop. The number of asynchronous calls to E prover that can be executed at the same time is limited by the number of cores available to the user.

5 Preselection

In order to speed up the predictions during the search for a proof of a conjecture c, we preselect 500 goal-tactic pairs and 500 theorems and a larger number of lists of goals induced by the selection of goal-tactic pairs. Preselected objects are the only objects available to TacticToe's search algorithm for proving c.

The first idea is to select goal-tactic pairs and theorems by their similarity with c (i.e. $Predictions_{500}^{tactic}(c)$ and $Predictions_{500}^{theorem}(c)$). However, this selection may not be adapted at later stages of the proof where goals may have drifted significantly from c. In order to anticipate the production of diverse goals, our prediction during preselection takes dependencies between objects of the same dataset into account. This dependency relation is asymmetric. Through the relation, each object has multiple children and at most one parent. Once a dependency relation is established we can calculate a dependency score for each object, which is the maximum of its similarity score and the similarity score of its parent. Finally, the 500 entries with highest dependency score are preselected in each dataset.

We now give a mathematical definition of the dependency relation for each kind of object.

Definition 6 (Dependencies of a goal-tactic pair)
Let $\mathfrak{F}$ be the set of recorded goal-tactic pairs. The dependencies D^* of a goal-tactic pair (t_0, g_0) are inductively defined by:

$$
\begin{aligned}
D_0 &=_{\text{def}} \{(t_0, g_0)\} \\
D_{n+1} &=_{\text{def}} D_n \cup \{(t, g) \in \mathfrak{F} \mid \exists (t', g') \in D_n.\ g \in t'(g')\} \\
D^* &=_{\text{def}} \bigcup_{i \in \mathbb{N}} D_i
\end{aligned}
$$

Definition 7 (Dependencies of a theorem)
Let $\mathfrak{R}$ be the set of recorded theorems. The dependencies of a theorem t are the set of top-level theorems in $\mathfrak{R}$ appearing in the proof of t. These dependencies are recorded by tracking the use of top-level theorems through the inference kernel [10].

We do preselection also for lists of goals. To preselect a list of goals l from our database of lists of goals, we consider the tactic input g that was at the origin of the production of l during orthogonalization. The list l is preselected if and only if g appears in the 500 preselected goal-tactic pairs.

6 Orthogonalization

Different tactics may transform a single goal in the same way. Exploring such equivalent paths is undesirable, as it leads to inefficiency in automated proof search. To solve this problem, we modify the construction of the tactics database. Each time a new goal-tactic pair (t, g) is extracted from a tactic proof and about to be recorded, we consider if there does not already exist a better tactic for g in our database. To this end, we organize a competition between the k closest goal-tactic pairs $\mathit{Predictions}_k^{tactic}(g)$. In our experiments, the default orthogonalization radius is $k = 20$. The winner is the tactic that subsumes (see Definition 11) the original tactic on g and that appears in the largest number of goal-tactic pairs in the database. The winning tactic w is then associated with g, producing the pair (w, g) and is stored in the database instead of the original pair (t, g). As a result, already successful tactics with a large coverage are preferred, and new tactics are considered only if they provide a different contribution. We now give a formal definition of the concepts of subsumption and coverage that are required for expressing the orthogonalization algorithm.

Definition 8 (Coverage)
Let $\mathfrak{T}$ be the database of goal-tactic pairs. We define the coverage $\mathit{Coverage}(t)$ of a tactic t by the number of times this tactic appears in $\mathfrak{T}$. Expressing this in a formula, we get:

$$\mathit{Coverage}(t) =_{\text{def}} |\{g \mid (t, g) \in \mathfrak{T}\}|$$

Intuitively, this notion estimates how general a tactic is by counting the number of different goals it is useful for.

Definition 9 (Goal subsumption)
A goal subsumption $\leq$ is a partial relation on goals. Assuming we have a way to estimate the number of steps required to solve a goal, a general and useful subsumption definition is for example:

$g_1 \leq g_2 \Leftrightarrow_{\text{def}}$ g_1 has a proof with a length shorter or equal to the proof of g_2

By default, we however choose for efficiency reasons a minimal subsumption defined by:

$$g_1 \leq g_2 \Leftrightarrow_{\text{def}} g_1 \text{ is } \alpha\text{-equivalent to } g_2$$

We can naturally extend any goal subsumption to a subsumption on lists of goals.

Definition 10 (Goal list subsumption)
A goal list subsumption is a partial relation on lists of goals. Given two lists of goals l_1 and l_2, we define it from the goal subsumption $\leq$ by:

$$l_1 \leq l_2 \Leftrightarrow_{\text{def}} \forall g_1 \in l_1.\ \exists g_2 \in l_2.\ g_1 \leq g_2$$

This allows us to define subsumption for tactics.

Definition 11 (Tactic subsumption)
Given two non-failing tactics t_1 and t_2 on g, a tactic t_1 subsumes a tactic t_2 on a goal g, denoted $\leq_g$, when:

$$t_1 \leq_g t_2 \Leftrightarrow_{\text{def}} t_1(g) \leq t_2(g)$$

If one of the tactics is failing then t_1 and t_2 are not comparable through this relation.

Finally, the winning tactic of the orthogonalization competition can be expressed by the formula:

$$\mathbb{O}(t, g) = \underset{x \in \mathit{Predictions}_k^{\mathit{tactic}}(g) \cup \{t\}}{\operatorname{argmax}} \{\mathit{Coverage}(x) \mid x \leq_g t\}$$

A database built with the orthogonalization process thus contains $(\mathbb{O}(t, g), g)$ instead of (t, g).

7 Abstraction

One of the major weaknesses of the previous version of TacticToe was that it could not create its own tactics. Indeed, sometimes no previous tactic is adapted for a certain goal and creating a new tactic is necessary. In this section we present a way to create tactics with different arguments by abstracting them and re-instantiating them using a predictor for that kind of argument. In the spirit of the orthogonalization method, we will try to create tactics that are more general but have the same effect as the original one. The generalized tactics are typically slower that their original variants, but the added flexibility is worthwhile in practice. Moreover, since we impose a timeout on each tactic (0.05 seconds by default), very slow tactics fail and thus are not selected.

Abstraction of Tactic Arguments The first step is to abstract arguments of tactics by replacing them by a placeholder, creating a tactic with an unspecified argument. Suppose we have a recorded tactic t. Since t is a SML code tree, we can try to abstract any of the SML subterms. Let u be a SML subterm of t, and h a variable (playing the role of a placeholder), we can replace u by h to create an abstracted tactic. We denote this intermediate tactic $t[h/u]$. By repeating the process, multiple arguments may be abstracted in the same tactic. Ultimately, the more abstractions are performed the more general a tactic is, as many tactics become instances of the same abstracted tactic. As a consequence, it becomes harder and harder to predict suitable arguments. In our experiments, we create one abstraction $\hat{t}$ for each tactic t by abstracting all subterms of type theorem list in t.

Instantiation of an Abstracted Tactic An abstracted tactic is not immediately applicable to a goal, since it contains unspecified arguments. To apply an abstracted tactic $\hat{t}$, we first need to instantiate the placeholders h inside $\hat{t}$. Because it is difficult to build a general predictor effective on each type of argument, we manually design different argument predictors for each type. Those predictors are given a goal g as input and return the best predicted argument for this goal.

Our default algorithm relies on argument predictions for theorems with a radius of 16. It uses the produced list $Predictions_{16}^{tactic}(g)$ to replace all the placeholders in $\hat{t}$. The type of theorem lists is a very common type in HOL4 proofs and theorems contain enough information to be predicted accurately.

Selection of Abstracted Tactics As abstracted tactics do not appear in human proofs, we need to find a way to predict them so that they can contribute to the TacticToe proof search. A straightforward idea is to try $\hat{t}$ before t during the proof search. However, this risks doing unnecessary work as the two may perform similar steps. Therefore, we would like to decide beforehand if one is better than the other. In fact, we can re-use the orthogonalization module to do this task for us. We add $\hat{t}$ to the competition, initially giving it the coverage of t. If $\hat{t}$ wins, it is associated with g and is included in the database of tactic features and thus can be predicted during proof search. After many orthogonalization competitions, the coverage of $\hat{t}$ may exceed the coverage of t. At this point, the coverage of $\hat{t}$ is estimated on its own and not inherited from t anymore.

8 Proof Recording

Recording proofs in an LCF-style proof assistant can be done at different levels. In HOL4 all existing approaches relied on modifying the kernel. This was used either to export the primitive inference steps [35, 23] or to record dependencies of theorems [10]. This was not suitable for our purpose of learning proving strategies at the intermediate tactic level. We therefore discuss recording proofs in an LCF-style proof assistant, with the focus on HOL4 in this section. Rather than relying on the underlying programming language, we parse the original script file containing tactic proofs. This enables us to extract the code of each tactic. Working with the string representation of a tactic is better than working with its value:

– Equality between two functions is easy to compute from their string representation, which allows us to avoid predicting the same tactic repeatedly.

- The results of TacticToe can be returned in a readable form to the user. In this way, the user can learn from the feedback, analyze the proof and possibly improve on it. Furthermore TacticToe does not need to be installed to run tactic proofs generated by TacticToe. In this way, the further development of TacticToe does not affect the robustness of HOL4.
- It is difficult (probably impossible) to transfer SML values between theories, since they are not run in the same HOL4 session. In contrast, SML code can be exported and imported.
- To produce a tactic value from a code is easy in SML, which can be achieved by using a reference to a tactic and updating it with the function `use`.

In order to transfer our tactic knowledge between theories, we want to be able to re-use the code of a tactic recorded in one theory in another. We also would like to make sure that the code is interpretable and that its interpretation does not change in the other theory.

Even when developing one theory, the context is continually changing: modules are opened and local identifiers are defined. Therefore, it is unlikely that code extracted from a tactic proof is interpretable in any other part of HOL4 without any post-processing. To solve this issue, we recursively replace each local identifier by its definition until we are able to write any expression with global identifiers only. Similarly, we prefix each global identifier by its module. We call this process *globalization*. The result is a standalone SML code executable in any HOL4 theory. In the absence of side effects, it is interpreted in the same way across theories. Therefore, we can guarantee that the behavior of recorded stateless tactics does not change. Even with some stateful tactics, the prediction algorithm is effective because most updates on the state of HOL4 increase the strength of stateful tactics.

8.1 Implementation

We describe in more detail our implementation of the recording algorithm. It consists of 4 phases: tactic proof extraction, tactic proof globalization, tactic unit wrapping, and creation of goal-tactic pairs.

Because of the large number of SML constructions, we only describe the effect of these steps on a running example that contains selected parts of the theory of lists.

Example 1 (Running example)

```
open boolLib Tactic Prim_rec Rewrite
...
val LIST_INDUCT_TAC = INDUCT_THEN list_INDUCT ASSUME_TAC
...
val MAP_APPEND = store_thm ("MAP_APPEND",
  ``∀(f:α→β) l1 l2. MAP f (APPEND l1 l2) = APPEND (MAP f l1) (MAP f l2)``,
  STRIP_TAC THEN LIST_INDUCT_TAC THEN ASM_REWRITE_TAC [MAP, APPEND])
```

The first line of this script file opens modules (called structures in SML). Each of the modules contains a list of global identifiers which become directly accessible in the rest of the script. A local identifier `LIST_INDUCT_TAC` is declared next, which is a tactic that performs induction on lists. Below that, the theorem `MAP_APPEND` is proven. The global tactic `STRIP_TAC` first removes universal quantifiers. Then, the goal is split into a base case and an inductive case. Finally, both of these cases are

solved by `ASM_REWRITE_TAC [MAP, APPEND]`, which rewrites assumptions with the help of theorems previously declared in this theory.

We first parse the script file to extract tactic proofs. Each of them is found in the third argument of a call to `store_thm`. The result of tactic proof extraction for the running example in presented in Example 2.

Example 2 (Tactic proof extraction)

```
STRIP_TAC THEN LIST_INDUCT_TAC THEN ASM_REWRITE_TAC [MAP, APPEND]
```

In the next phase, we globalize identifiers of the tactic proofs. Infix operators such as `THEN` need to be processed in a special way so that they keep their infixity status after globalization. For simplicity, the globalization of infix operators is omitted in Example 3. In this example, the three main cases that can happen during the globalization are depicted. The first one is the globalization of identifiers declared in modules. The global identifiers `STRIP_TAC` and `ASM_REWRITE_TAC` are prefixed by their module `Tactic`. In this way, they will still be interpretable whether `Tactic` was open or not. The local identifier `LIST_INDUCT_TAC` is replaced by its definition which happens to contain two global identifiers. The previous paragraph describes the globalization for all identifiers except local theorems. We do not replace a local theorem by its tactic proof (which its SML definition) because we want to avoid unfolding proofs inside other proofs. If the theorem is stored in the HOL4 database available across HOL4 developments, we can obtain the theorem value by calling `DB.fetch`. Otherwise the globalization process fails and the local theorem identifier is kept unchanged. A recorded tactic with an unchanged local theorem as argument is only interpretable inside the current theory.

Example 3 (Globalization)

```
Tactic.STRIP_TAC THEN
Prim_rec.INDUCT_THEN (DB.fetch "list" "list_INDUCT") Tactic.ASSUME_TAC THEN
Rewrite.ASM_REWRITE_TAC [DB.fetch "list" "MAP", DB.fetch "list" "APPEND"]
```

Running the globalized version of a tactic proof will have the exact same effect as the original. But since we want to extract information from this proof in the form of tactics and their input goals, we modify it further. In particular, we need to define at which level we should record the tactics in the tactic proof. The simplest idea would be to record all SML subexpressions of type `tactic`. However, it will damage the quality of our data by dramatically increasing the number of tactics associated with a single goal. Imagine a tactic proof of the form `A THEN B THEN C`; then the tactics `A`, `A THEN B` and `A THEN B THEN C` would be valid advice for something close to their common input goal. The tactic `A THEN B THEN C` is likely to be too specific. In contrast, we can consider the tactic `REPEAT A` that is repetitively calling the tactic `A` until `A` has no effect. For such calls, it is often preferable to record a single call to `REPEAT A` rather than multiple calls to `A`. Otherwise branches after each call to `A` would be necessary as part of proof search. To sum up, we would like to record only the most general tactics which make the most progress on a goal. As a trade-off between these two objectives, we split proofs into tactic units.

Definition 12 (Tactic unit) A tactic unit is an SML expression of type `tactic` that does not contain an infix operator at its root.

Because such tactic units are constructed from visual information present in the tactic proof, they often represent what a human user considers to be a single step of the proof.

To produce the final recording tactic proof, we encapsulate each tactic unit in a recording function `R` (see Example 4). In order to record tactics in all HOL4 theories, we replace the original proof by the recording proof in each theory and rebuild the HOL4 library.

Example 4 (Tactic unit wrapping)

```
R "Tactic.STRIP_TAC" THEN
R "Prim_rec.INDUCT_THEN (DB.fetch \"list\" \"INDUCT\") Tactic.ASSUME_TAC" THEN
R "Rewrite.ASM_REWRITE_TAC
  [DB.fetch \"list\" \"MAP\", DB.fetch \"list\" \"APPEND\"]"
```

At run time the function `R` is designed to observe what input goals a tactic receives without changing its output. The implementation of `R` is presented in Example 5.

Example 5 (Code of the recording function)

```
fun R stac goal = (save (stac,goal); tactic_of_sml stac goal)
```

The function `save` writes the goal-tactic pair to disk increasing the number of entries in our database of tactics. The function `tactic_of_sml` interprets the SML code `stac`. The tactic is then applied to the input goal to replicate the original behavior. After all modified theories are rebuilt, each call to a wrapped tactic in a tactic proof is recorded as a pair containing its globalized code and its input goal.

9 Experimental Evaluation

The execution of TacticToe's main loop in each re-proving experiment is performed on a single CPU. An additional CPU is needed for experiments relying on asynchronous E prover calls.

9.1 Methodology

The evaluation imitates the construction of the library: For each theorem only the previous human proofs are known. These are used as the learning base for the predictions. To achieve this scenario we re-prove all theorems during a modified build of HOL4. As theorems are proved, their tactical proofs and their statements are recorded and included in the training examples. For each theorem we first attempt to run the TacticToe search with a time limit of 60 seconds before processing the original tactic proof. In this way, the fairness of the experiments is guaranteed by construction: only previously declared SML values (essentially tactics, theorems and simpsets) are accessible to TacticToe.

Datasets: optimization and validation All top-level theorems from the standard library are considered with the exception of 440 hard problems (containing a `let` construction in their proof) and 1175 easy problems (build from `save_thm` calls). Therefore, during the full-scale experiments, we evaluate 7164 theorems. We use every tenth theorem of the first third of the standard library for parameter optimization, which amounts to 273 theorems.

Although the evaluation of each set of parameters on its own is fair, the selection of the best strategy in Section 9.2 should also be considered as a learning process. To ensure the global fairness, the final experiment in Section 9.3 runs the best strategy on the full dataset which is about 30 times larger.

9.2 Tuning TacticToe

We optimize TacticToe by tuning parameters of six different techniques: the timeout of tactics, orthogonalization, abstraction, MCTS policy, MCTS evaluation and ATP integration. During training, optional techniques such as orthogonalization may also be turned off completely. TacticToe also includes other important techniques which are run with their default parameters: feature extraction, prediction algorithms, number of premises for ATPs and MCTS evaluation radius (set by default to 10). By choosing a set of parameters, we create a strategy for TacticToe that is evaluated on our training set of 273 theorems. And the strategy with the highest number of re-proven theorems is selected for a full scale evaluation.

In Table 1, the success rate of the different TacticToe strategy is presented. The first four techniques are tested relative to the same baseline indicated in the table by the tag "default". This default strategy relies on a tactic time out of 0.05s, an orthogonalization radius of 20 and a policy coefficient of 0.5. The two last techniques are tested relative to a baseline consisting of the best set of parameters discovered so far and are marked with the improvement over the "default" strategy. In each subtable, each experiment differs from the current baseline by exactly one parameter.

Thanks to the larger search time available in this experiment, the timeout for each tactic can be increased from the 0.02 seconds used in TacticToe's initial experiments [11] to 0.05 seconds. Removing tactics with similar effect as performed by the orthogonalization process is only beneficial when running TacticToe for short period of time. It seems that the allotted time allows a strategy running without orthogonalization to catch up. Yet, side experiments show that orthogonalization becomes relevant again when we add additional tactics through abstraction. It is best to predict a list of 16 theorems to instantiate arguments of type theorem list in tactics. At any rate, argument prediction for abstracted tactics is the technique that have the highest impact on the success of TacticToe. Integration of ATPs being a specialization of this technique contributes significantly as well. Experiments involving E prover, run the ATP using a separate process asynchronously with a timeout of 5 seconds and 128 premises. The fact that Metis outperforms E prover in this setting is due to the fact that Metis is run with a very short timeout, allowing to close different part of search tree quickly. Because Metis is weaker that TacticToe as an ATP, it is given less predictions. But even if an essential lemma for the proof of a goal is not predicted, a modification on the goal performed by a tactic can change its prediction to include the necessary lemma. This effect minimizes the drawback of relying on a small number of predictions.

Technique	Parameters	Solved (1s)	Solved (60s)
Tactic time out	0.02s		154
	0.05s (default)		156
	0.1s		154
Orthogonalization	none	105	156
	$radius = 10$	121	156
	$radius = 20$ (default)	121	156
	$radius = 40$	124	156
Abstraction	none (default)		156
	$theorems = 8$		195
	$theorems = 16$		199
	$theorems = 32$		195
MCTS policy	$c_{policy} = 0.4$		149
	$c_{policy} = 0.5$ (default)		156
	$c_{policy} = 0.6$		153
MCTS evaluation	none (default + best abstraction)		199
	$c_{exploration} = 1$		201
	$c_{exploration} = 2$		203
	$c_{exploration} = 4$		198
ATP integration	none (default + best abstraction + best MCTS evaluation)		203
	Metis 0.1s		216
	Metis 0.2s		212
	Metis 0.4s		212
	E prover		213
	Metis 0.1s + E prover		218

Table 1 Number of problem solved with different set of parameters for TacticToe on a training set of 273 theorems.

The MCTS evaluation, despite relying on the largest amount of collected data, does not provide a significant improvement over a strategy relying on no evaluation. The main reason is that our predictor learning abilities is limited and we are using an estimate of the provability of lists of goals for evaluation. A more accurate evaluation could be based on an estimation of the length of the proof required to close a list of goals.

9.3 Full-scale experiment

Based on the results of parameter tuning, we now evaluate a version of TacticToe with its best parameters on an evaluation set of 7164 theorems from the HOL4 standard library. We compare it with the performance of E prover. For reasons of fairness, E prover asynchronous calls are not included in the best TacticToe strategy. The ATP E prover is run in auto-schedule mode with 128 premises. The settings for

TacticToe are the following: 0.05 seconds tactic timeout, an orthogonalization radius of 20, theorem list abstraction with 16 predicted theorems for instantiation, a prior policy coefficient of 0.5, an evaluation radius of 10, an exploration coefficient of 2 and priority calls to Metis with a timeout of 0.1 seconds.

	Solved (60s)
E prover	2472 (34.5%)
TacticToe	4760 (66.4%)
Total	4946 (69.0%)

Table 2 Evaluation on 7164 top-level theorems of the HOL4 standard library

The results shows that TacticToe is able to prove almost twice as many theorems as E prover. Combining the results of TacticToe and E prover we get a 69.0% success rate which is significantly above the 50% success rates of hammers on this type of problems [10]. Moreover, TacticToe is running a single set of parameters (strategy), where as hammers and ATPs have been optimized and rely on a wide range of strategies.

Reconstruction Tactic proofs produced by TacticToe during this experiment are all verifiable in HOL4. By the design of the proof search, reconstruction of TacticToe proof succeeds, unless one of the tactics modifies the state of HOL4 in a way that changes the behavior of a tactic used in the final proof. This has not occurred a single time in our experiments. And a tactical proof generated by TacticToe during the final experiment takes on average 0.37 seconds to replay. More details on how a proof is extracted from the search tree is given in Section 10. Comparatively, we achieve a reconstruction rate of 95 percent for E prover, by calling Metis for 2.0 seconds with the set of theorems used in E prover's proof as argument.

Table 3 compares the re-proving success rates for different HOL4 theories. TacticToe outperforms E prover on every considered theory. E prover is more suited to deal with dense theories such as `real` or `complex` where a lot of related theorems are available and most proofs are usually completed by rewriting tactics. Thanks to its ability to re-use custom-built tactics, TacticToe largely surpasses E prover on theories relying on inductive terms and simplification sets such as `arithmetic`, `list` and `f_map`. Indeed, TacticToe is able to recognize where and when to apply induction, a task at which ATPs are known to struggle with.

	arith	real	compl	meas
TacticToe	81.2	74.0	79.6	31.3
E prover	59.9	72.0	67.1	12.8
	proba	list	sort	f_map
TacticToe	45.8	79.5	65.3	82.0
E prover	24.1	26.5	15.8	24.7

Table 3 Percentage (%) of re-proved theorems in the theories `arithmetic`, `real`, `complex`, `measure`, `probability`, `list`, `sorting` and `finite_map`.

Reinforcement learning All our feature vectors have been learned form human proofs. We now can now also add goal-tactic pairs that appears in the last proof of TacticToe. To prevent duplication of effort, orthogonalization of those tactics is essential to have a beneficial effect. Since recording and re-proving are intertwined during evaluation, the additional data is available for the next proof search. The hope is that the algorithm will improve faster by learning from its own discovered proofs than from the human-written proofs [32]. Side experiments show that this one shot reinforcement learning method increases TacticToe's success rate by less than a percent.

9.4 Complexity of the proof search

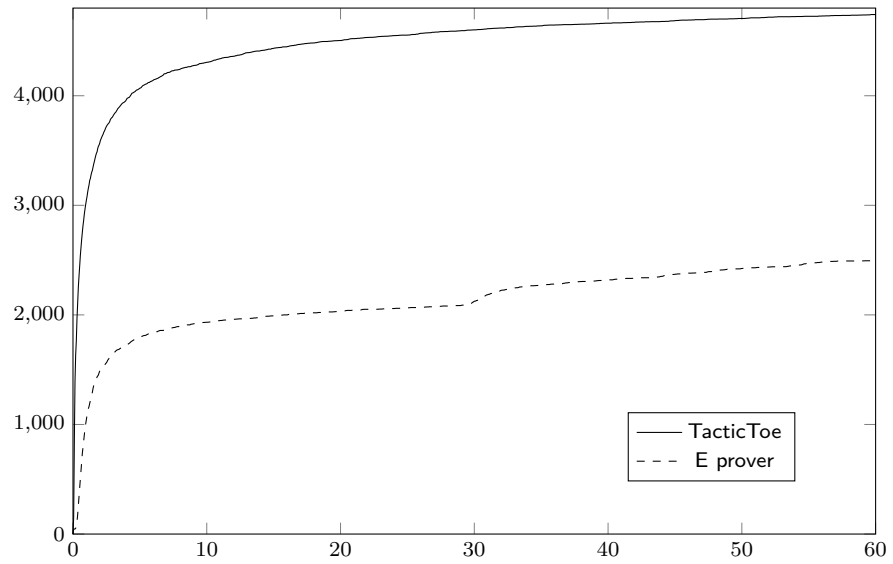

Fig. 3 Number of problems solved in less than x seconds.

We first investigate how TacticToe and E prover scale as the search time grows in Figure 3. In 10 seconds, TacticToe solves 90 percent of the problems it can solve in 60 seconds. The analysis is a bit different for E prover. Indeed, we can clearly deduce from the bump at 30 seconds that E prover is using at least two strategies and probably more. Strategies are a useful method to fight the exponential decline in the number of newly proven theorems over time. Therefore, integrating strategy scheduling in TacticToe could be something to be experimented with.

In order to appreciate the difficulty of the evaluation set from a human perspective, the length distribution of human proofs in the validation set is shown in Figure 4. It is clear from the graph that most of the human proofs are short. The proofs found by TacticToe follow a similar distribution. If TacticToe finds a proof, there is about a 50 percent chance that it will be shorter than what a human would come up with.

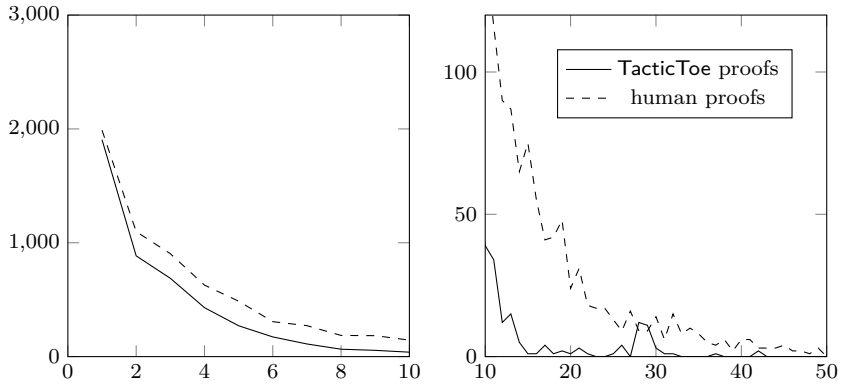

Fig. 4 Number of tactic proofs with x tactic units.

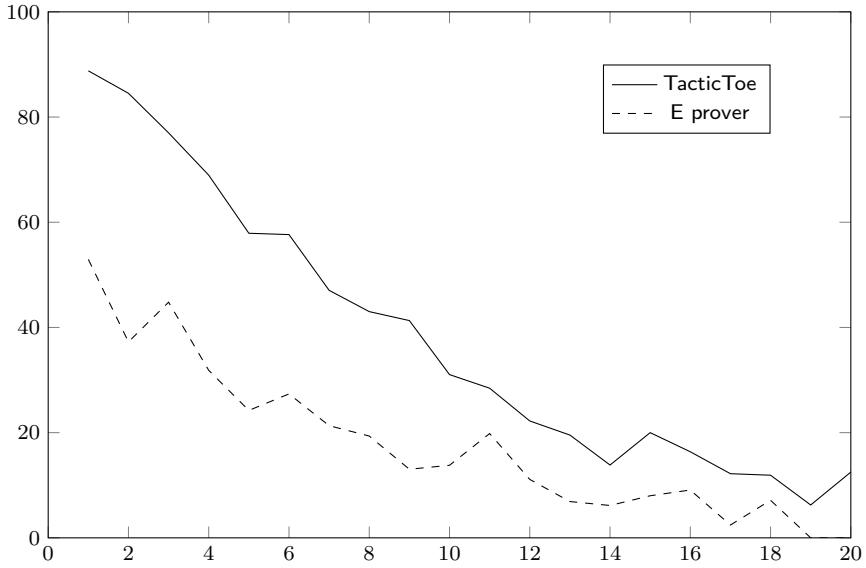

Fig. 5 Percentage of problem solved with respect to the length of the original proof until length 20.

In Figure 5, we regroup problems by the length of their human proof to measure how well TacticToe and E prover cope with increasing levels of difficulty. As expected, the longer the original proof is, the harder it is for TacticToe and E prover to re-prove the theorem on their own. The performance of TacticToe is compelling with more than half of the theorems that required a proof of length six being re-proven. It is also consistently better than E prover for any proof length.

10 Minimization and Embellishment

When TacticToe finds a proof of an initial goal, it returns a search tree $\mathfrak{T}$ where the root node containing the initial goal is solved. In order to transform this search tree into a tactic proof, we need to extract the tactics that contributed to the proof and combine them using tactic combinators.

By the design of the search, a single tactic combinator, `THENL`, is sufficient. It combines a tactic t with a list of subsequent ones, in such a way that after t is called, for each created goal a respective tactic from the list is called.

Let T_{sol} be a partial function that from a goal g returns a tactic t for which $A(g,t)$ is solved in $\mathfrak{T}$. The proof extraction mechanism is defined by mutual recursion on goals and nodes of $\mathfrak{T}$ by the respective function P_{goal} and P_{node}:

$$P_{goal}(g) =_{\text{def}} T_{sol}(g)\ \texttt{THENL}\ P_{node}(A(g, T_{sol}(g)))$$
$$P_{node}(a) =_{\text{def}} [P_{goal}(g_1), \ldots, P_{goal}(g_n)] \quad \text{with } G(a) = g_1, \ldots, g_n$$

The extracted tactic proof of $\mathfrak{T}$ is $P_{goal}(g_{root})$. We minimally improve it by substituting `THENL` by `THEN` when the list of goals is a singleton and removing `THENL []` during the extraction phase. Further post-processing such as eliminating unnecessary tactics and theorems has been developed and improves the user experience greatly [1].

Proof Length Minimization A fast and simple minimization is applied when processing the final proof. If the tactic `A THEN B` appears in the proof and has the same effect as `B` then we can replace `A THEN B` by `B` in the proof. Optionally, stronger minimization can be obtained by rerunning TacticToe with a low prior policy coefficient, no prior evaluation and a database of tactics that contains tactics from the discovered proof only.

Tactic Arguments Minimization Let t be a tactic applied to a goal g containing a list l (of theorems) as argument. And let t' be the tactic t where one element e of l as be removed. If t and t' have the same effect on g then t' can replace t in the final proof. This process is repeated for each element of l. This is a generalization of the simplest method used for minimizing a list of theorems in "hammers" [3].

Embellishment Without embellishment, the returned tactic is barely readable as it contains information to guarantees that each SML subterm is interpreted in the way in any context. Since we return the proof at a point where the SML interpreter is in a specific state, we can strip unnecessary information, such as module prefixes. If possible, we group all local declarations in the proof under a single `let` binding at the start of the proof. We also replace extended terms by their quoted version. All in all, if a prettified tactic t_p has the same effect its original t_o, we replace t_p by t_o in the proof.

The total effect of these minimization and embellishment techniques on the readability of a tactic proof is depicted in Example 6.

Example 6 (Theorem EVERY_MAP)
Before minimization and embellishment:

```
boolLib.REWRITE_TAC [DB.fetch "list" "EVERY_CONJ", DB.fetch "list" "EVERY_MEM",
 DB.fetch "list" "EVERY_EL", ... , combinTheory.o_DEF] THEN
BasicProvers.Induct_on [HolKernel.QUOTE " (*#loc 1 11380*)l"] THENL
 [BasicProvers.SRW_TAC [] [],
  simpLib.ASM_SIMP_TAC (BasicProvers.srw_ss ()) [boolLib.DISJ_IMP_THM,
   DB.fetch "list" "MAP", DB.fetch "list" "CONS_11", boolLib.FORALL_AND_THM]]
```

After minimization and embellishment:

```
Induct_on `l` THENL
 [SRW_TAC [] [], ASM_SIMP_TAC (srw_ss ()) [DISJ_IMP_THM, FORALL_AND_THM]]
```

11 Case Study

Since the proofs generated by TacticToe are meant to be parts of a HOL4 development, it interesting to compare their quality with the original human proofs. The quality of a proof in HOL4 can be measured in terms of length, readability, maintainability and verification speed. We study these properties in three examples taken from our full-scale experiment. We list the time needed by HOL4 to check a proof in parentheses.

We start with Example 7, which proves that greatest common divisors are unique. The human formalizer recognized that it follows from two theorems. The relevance filtering of TacticToe is not as powerful as that used in hammers and is therefore not able to find the DIVIDES_ANTISYM property. The proof proceeds instead by rewriting the goal with the definitions then splitting the goal into multiple cases until the goal obligations are similar to DIVIDES_ANTISYM. As expected, TacticToe's proof also takes much longer to check.

Example 7 IS_GCD_UNIQUE in theory gcd

```
∀a b c d. is_gcd a b c ∧ is_gcd a b d ⇒ (c = d)
```

Human proof (5 milliseconds)

```
PROVE_TAC[IS_GCD, DIVIDES_ANTISYM]
```

TacticToe proof (80 milliseconds)

```
STRIP_TAC THEN
REWRITE_TAC [fetch "gcd" "is_gcd_def"] THEN
REPEAT Cases THENL
 [METIS_TAC [],
  REWRITE_TAC [SUC_NOT, ALL_DIVIDES_0, compute_divides] THEN
   METIS_TAC [NOT_SUC],
  METIS_TAC [NOT_SUC, DIVIDES_ANTISYM],
  METIS_TAC [LESS_EQUAL_ANTISYM, DIVIDES_LE, LESS_0],
  METIS_TAC [],
  RW_TAC numLib.arith_ss [divides_def],
  METIS_TAC [DIVIDES_ANTISYM],
  METIS_TAC [LESS_EQUAL_ANTISYM, DIVIDES_LE, LESS_0]]
```

In Example 8, we try to prove that for any surjective function f from s to t, there exists an injective function g from t to s such that $f\ o\ g$ is the identity function. The human proof is quite complicated. In contrast, TacticToe finds a much smaller proof that expands the definition of injectivity and surjectivity and calls Metis. However, the TacticToe's proof takes much longer to check due to the proof search happening inside Metis.

Example 8 SURJ_INJ_INV in theory pred_set

```
∀f s t. SURJ f s t ⇒ ?g. INJ g t s ∧ ∀y. y IN t ⇒ (f (g y) = y)
```

Human proof (2 milliseconds)

```
REWRITE_TAC [IMAGE_SURJ] THEN
DISCH_TAC THEN Q.EXISTS_TAC `THE o LINV_OPT f s` THEN
BasicProvers.VAR_EQ_TAC THEN REPEAT STRIP_TAC THENL
 [irule INJ_COMPOSE THEN Q.EXISTS_TAC `IMAGE SOME s` THEN
   REWRITE_TAC [INJ_LINV_OPT_IMAGE] THEN REWRITE_TAC [INJ_DEF, IN_IMAGE] THEN
   REPEAT STRIP_TAC THEN REPEAT BasicProvers.VAR_EQ_TAC THEN
   FULL_SIMP_TAC std_ss [THE_DEF],
  ASM_REWRITE_TAC [LINV_OPT_def, o_THM, THE_DEF] THEN
   RULE_ASSUM_TAC (Ho_Rewrite.REWRITE_RULE
    [IN_IMAGE', GSYM SELECT_THM, BETA_THM]) THEN ASM_REWRITE_TAC []]
```

TacticToe proof (50 milliseconds)

```
SRW_TAC [] [SURJ_DEF, INJ_DEF] THEN METIS_TAC []
```

In Example 9, we prove a theorem about lists, a domain where TacticToe excels compared to ATPs. Given two list l_1 and l_2 where the length of l_1 (denoted p) is less than a natural number n, the theorem states updating the n^{th} element of the concatenation of l_1 and l_2 is the same as updating the m^{th} element of l_2 where $m = n - p$. Again, the TacticToe's proof is much more readable. It starts by applying induction on l. It solves the base case of the induction by rewriting and proceeds by cases on $n = 0$ or $n > 0$ in the induction hypothesis. It finalizes the proof with a short call to Metis using the definition of LUPDATE. Here, TacticToe's proof is arguably smaller, faster and easier to understand and maintain. Such proofs after an expert review could replace the their respective original human proofs in the HOL4 repository.

Example 9 LUPDATE_APPEND2 in theory rich_list

```
∀l1 l2 n x. LENGTH l1 ≤ n ⇒
 (LUPDATE x n (l1 ++ l2) = l1 ++ (LUPDATE x (n − LENGTH l1) l2))
```

Human proof (63 milliseconds)

```
 rw[] THEN simp[LIST_EQ_REWRITE] THEN Q.X_GEN_TAC `z` THEN
 simp[EL_LUPDATE] THEN rw[] THEN simp[EL_APPEND2,EL_LUPDATE] THEN
 fs[] THEN Cases_on `z < LENGTH l1` THEN
 fs[] THEN simp[EL_APPEND1,EL_APPEND2,EL_LUPDATE]
```

TacticToe proof (17 milliseconds)

```
Induct_on `l1` THENL [SRW_TAC [] [],
 Cases_on `n` THENL [SRW_TAC [] [],
  FULL_SIMP_TAC (srw_ss ()) [] THEN METIS_TAC [LUPDATE_def]]]
```

In Example 10, we experiment with TacticToe on a goal that does not originate from the HOL4 library. The conjecture is that the set of numbers $\{0, ..., n+m-1\} \setminus \{0, ..., n-1\}$ is the same as the set obtained by adding n to everything in $\{0, ..., m-1\}$. In this example, TacticToe uses the simplification set `ARITH_ss` to reduce arithmetic formulas. This exemplifies another advantage that TacticToe has over ATPs, namely its ability to take advantage of user-defined simplification sets.

Example 10
```
count (n+m) DIFF count n = IMAGE ((+) n) (count m)
SRW_TAC [ARITH_ss] [EXTENSION, EQ_IMP_THM] THEN
Q.EXISTS_TAC `x − n` THEN
SRW_TAC [ARITH_ss] []
```

12 Related Work

There are several essential components of our work that are comparable to previous approaches: tactic-level proof recording, tactic selection through machine learning techniques and automatic tactic-based proof search. Our work is also related to previous approaches that use machine learning to select premises for the ATP systems and guide ATP proof search internally.

In HOL Light, the Tactician tool [1] can transform a packed tactical proof into a series of interactive tactic calls. Its principal application was so far refactoring the library and teaching common proof techniques to new ITP users. In our work, the splitting of a proof into a sequence of tactics is essential for the tactic recording procedure, used to train our tactic prediction mechanism.

SEPIA [13] can generate proof scripts from previous Coq proof examples. Its strength lies in the ability to produce likely sequences of tactics for solving domain specific goals. It operates by creating a model for common sequences of tactics for a specific library. This means that in order to propose the next tactic, only the previously called tactics are considered. Our algorithm, on the other hand, relies mainly on the characteristics of the current goal to decide which tactics to apply next. In this way, our learning mechanism has to rediscover why each tactic was applied for the current subgoals. It may lack some useful bias for common sequences of tactics, but is more reactive to subtle changes. Indeed, it can be trained on a large library and only tactics relevant to the current subgoal will be selected. Concerning the proof search, SEPIA's breadth-first search is replaced by MCTS which allows for supervised learning guidance in the exploration of the search tree. Finally, SEPIA was evaluated on three chosen parts of the Coq library demonstrating that it globally outperforms individual Coq tactics. Here, we demonstrate the competitiveness of our system against E prover on the HOL4 standard library.

ML4PG [22,15] groups related proofs using clustering algorithms. It allows Coq users to inspire themselves from similar proofs and notice duplicated proofs. Comparatively, our predictions come from a much more detailed description of the target goal. TacticToe can also organize the predictions to produce verifiable proofs and is not restricted to user advice.

Proof patching [27] is an approach that attempts to learn the proof changes necessary for a modification of a prover library or system. The approach is hard to apply for new proofs. Following this idea, TacticToe could also try to learn from different versions of the prover library.

It is important to consider the strength of the language used for building proofs and proof procedures when designing a system such as TacticToe. Indeed, such system will be re-using those procedures. In Isabelle, the language Eisbach [24] was created on top of the standard tactic language by giving access to constructors of the tactic language to the end-user. The language PSL [26] was also developed. It can combine regular tactics and tools such as Sledgehammer. In Coq, the Ltac [7] meta language was designed to enrich the tactic combinators of the Coq language. It adds functionalities such as recursors and matching operators for terms and proof contexts. In HOL4, the strength of SML as a proof strategy language has been demonstrated by the implementation of complex proof procedures such as Metis.

Machine learning has also been used to advise the best library lemmas for new ITP goals. This can be done either in an interactive way, when the user completes the proof based on the recommended lemmas, as in the Mizar proof advisor MizAR [31, 21], or attempted fully automatically, where such lemma selection is handed over to the ATP component of a *hammer* system, as in the hammers for HOL4 [10], HOL Light [19] and Isabelle [4].

Internal learning-based selection of tactical steps inside an ITP is analogous to internal learning-based selection of clausal steps inside ATPs such as MaLeCoP [33] and FEMaLeCoP [20]. These systems use the naive Bayes classifier to select clauses for the extension steps in tableaux proof search based on many previous proofs. Satallax [5] can guide its search internally [9] using a command classifier, which can estimate the priority of the 11 kinds of commands in the priority queue based on positive and negative examples.

13 Conclusion

We proposed a new proof assistant automation technique which combines tactic-based proof search with machine learning prediction. Its implementation, TacticToe, achieves an overall success rate of 66.4% on 7164 theorems of the HOL4 standard library, surpassing E prover with auto-schedule. Its effectiveness is especially visible on theories which use inductive data structures, specialized decision procedures, and custom built simplification sets. Thanks to the learning abilities of TacticToe, the generated tactic proofs often reveal the high-level structure of the proof. We therefore believe that predicting ITP tactics based on the current goal features is a very reasonable approach to automatically guiding proof search, and that accurate predictions can be obtained by learning from the knowledge available in today's large formal proof corpora.

To improve the quality of the predicted tactics, we would like to predict other type of arguments independently, as it was done for theorems. In this direction, the most interesting arguments to predict next are terms as they are ubiquitous in tactics. Further along the way, new tactics could be created by programming them with any construction available in SML. The proof search guidance can also be improved, for example by considering stronger machine learning algorithms such as deep neural networks as a model for the policy and evaluation in MCTS. The quality of such models could be enhanced by training and testing existing and synthesized tactics. Conjecturing suitable intermediate steps could also allow TacticToe to solve problems which require long proofs by decomposing it into multiple easier steps.

Acknowledgments We would like to thank Lasse Blaauwbroek and Yutaka Nagashima for their insightful comments which contributed to improve the quality of this paper. This work has been supported by the ERC Consolidator grant no. 649043 *AI4REASON*, the ERC starting grant 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

1. Mark Adams. Refactoring proofs with Tactician. In Domenico Bianculli, Radu Calinescu, and Bernhard Rumpe, editors, *Human-Oriented Formal Methods (HOFM)*, volume 9509 of *LNCS*, pages 53–67. Springer, 2015.
2. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.
3. Jasmin Blanchette, Cezary Kaliszyk, Lawrence Paulson, and Josef Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9(1):101–148, 2016.
4. Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. A learning-based fact selector for Isabelle/HOL. *J. Autom. Reasoning*, 57(3):219–244, 2016.
5. Chad E. Brown. Reducing higher-order theorem proving to a sequence of SAT problems. *Journal of Automated Reasoning*, 51(1):57–77, Mar 2013.
6. Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):1–43, 2012.
7. David Delahaye. A tactic language for the system Coq. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning*, pages 85–95, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.
8. Sahibsingh A. Dudani. The distance-weighted k-nearest-neighbor rule. *Systems, Man and Cybernetics, IEEE Transactions on*, SMC-6(4):325–327, 1976.
9. Michael Färber and Chad E. Brown. Internal guidance for Satallax. In Nicola Olivetti and Ashish Tiwari, editors, *8th International Joint Conference on Automated Reasoning (IJCAR 2016)*, volume 9706 of *LNCS*, pages 349–361. Springer, 2016.
10. Thibault Gauthier and Cezary Kaliszyk. Premise selection and external provers for HOL4. In Xavier Leroy and Alwen Tiu, editors, *Proc. of the 4th Conference on Certified Programs and Proofs (CPP'15)*, pages 49–57. ACM, 2015.
11. Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. TacticToe: Learning to reason with HOL4 tactics. In Thomas Eiter and David Sands, editors, *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pages 125–143. EasyChair, 2017.
12. Mike Gordon. Hol - a machine oriented formulation of higher order logic, 2001.
13. Thomas Gransden, Neil Walkinshaw, and Rajeev Raman. SEPIA: search for proofs using inferred automata. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 246–255, 2015.
14. John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer, 2009.
15. Jónathan Heras and Ekaterina Komendantskaya. Recycling proof patterns in Coq: Case studies. *Mathematics in Computer Science*, 8(1):99–116, 2014.
16. Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, number NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, September 2003.
17. Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.
18. Cezary Kaliszyk and Josef Urban. Stronger automation for Flyspeck by feature weighting and strategy evolution. In Jasmin Christian Blanchette and Josef Urban, editors, *PxTP 2013*, volume 14 of *EPiC Series*, pages 87–95. EasyChair, 2013.

19. Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with Flyspeck. *Journal of Automated Reasoning*, 53(2):173–213, 2014.
20. Cezary Kaliszyk and Josef Urban. FEMaLeCoP: Fairly Efficient Machine Learning Connection Prover. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2015)*, pages 88–96, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
21. Cezary Kaliszyk and Josef Urban. MizAR 40 for Mizar 40. *J. Autom. Reasoning*, 55(3):245–256, 2015.
22. Ekaterina Komendantskaya, Jónathan Heras, and Gudmund Grov. Machine learning in Proof General: Interfacing interfaces. In *Proceedings 10th International Workshop On User Interfaces for Theorem Provers, UITP 2012, Bremen, Germany, July 11th, 2012.*, pages 15–41, 2012.
23. Ramana Kumar and Joe Hurd. Standalone tactics using OpenTheory. In Lennart Beringer and Amy P. Felty, editors, *Interactive Theorem Proving (ITP)*, volume 7406 of *Lecture Notes in Computer Science*, pages 405–411. Springer, 2012.
24. Daniel Matichuk, Toby Murray, and Makarius Wenzel. Eisbach: A proof method language for Isabelle. *Journal of Automated Reasoning*, 56(3):261–282, March 2016.
25. Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors. *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*. Springer, 2008.
26. Yutaka Nagashima and Ramana Kumar. A proof strategy language and proof script generation for Isabelle/HOL. In Leonardo de Moura, editor, *26th International Conference on Automated Deduction (CADE 2017)*, volume 10395 of *LNCS*, pages 528–545. Springer, 2017.
27. Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Adapting proof automation to adapt proofs. In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018*, pages 115–129. ACM, 2018.
28. Stephan Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2-3):111–126, 2002.
29. David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.
30. Konrad Slind and Michael Norrish. A brief overview of HOL4. In Mohamed et al. [25], pages 28–32.
31. Josef Urban. MPTP - Motivation, Implementation, First Experiments. *J. Autom. Reasoning*, 33(3-4):319–339, 2004.
32. Josef Urban. Malarea: a metasystem for automated reasoning in large theories. In Geoff Sutcliffe, Josef Urban, and Stephan Schulz, editors, *Empirically Successful Automated Reasoning in Large Theories (ESLART)*, volume 257 of *CEUR*. CEUR-WS.org, 2007.
33. Josef Urban, Jiří Vyskočil, and Petr Štěpánek. MaLeCoP Machine Learning Connection Prover. In Kai Brünnler and George Metcalfe, editors, *Automated Reasoning with Analytic Tableaux and Related Methods: 20th International Conference, TABLEAUX 2011, Bern, Switzerland, July 4-8, 2011. Proceedings*, pages 263–277, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
34. Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle framework. In Mohamed et al. [25], pages 33–38.
35. Wai Wong. Recording and checking HOL proofs. In *Higher Order Logic Theorem Proving and Its Applications. 8th International Workshop, volume 971 of LNCS*, pages 353–368. Springer-Verlag, 1995.