TacticToe: Learning to reason with HOL4 Tactics

Thibault Gauthier[1], Cezary Kaliszyk[1], and Josef Urban[2]

[1] University of Innsbruck
{thibault.gauthier,cezary.kaliszyk}@uibk.ac.at
[2] Czech Technical University, Prague.
josef.urban@gmail.com

Abstract

Techniques combining machine learning with translation to automated reasoning have recently become an important component of formal proof assistants. Such "hammer" techniques complement traditional proof assistant automation as implemented by tactics and decision procedures. In this paper we present a unified proof assistant automation approach which attempts to automate the selection of appropriate tactics and tactic-sequences combined with an optimized small-scale hammering approach. We implement the technique as a tactic-level automation for HOL4: TacticToe. It implements a modified A*-algorithm directly in HOL4 that explores different tactic-level proof paths, guiding their selection by learning from a large number of previous tactic-level proofs. Unlike the existing hammer methods, TacticToe avoids translation to FOL, working directly on the HOL level. By combining tactic prediction and premise selection, TacticToe is able to re-prove 39% of 7902 HOL4 theorems in 5 seconds whereas the best single HOL(y)Hammer strategy solves 32% in the same amount of time.

1 Introduction

Example 1. Proof automatically generated by TacticToe for the user given goal

```
Goal: ‘‘∀l. FOLDL (λxs x. SNOC x xs) [] l = l‘‘
Proof:
  SNOC_INDUCT_TAC THENL
  [ REWRITE_TAC [APPEND_NIL, FOLDL],
    ASM_REWRITE_TAC [APPEND_SNOC, FOLDL_SNOC]
      THEN CONV_TAC (DEPTH_CONV BETA_CONV)
      THEN ASM_REWRITE_TAC [APPEND_SNOC, FOLDL_SNOC] ]
```

Many of the state-of-the-art interactive theorem provers (ITPs) such as HOL4 [27], HOL Light [12], Isabelle [32] and Coq [2] provide high-level parameterizable tactics for constructing the proofs. Such tactics typically analyze the current goal state and assumptions, apply nontrivial proof transformations, which get expanded into possibly many basic kernel-level inferences or significant parts of the proof term. In this work we develop a tactic-level automation procedure for the HOL4 ITP which guides selection of the tactics by learning from previous proofs. Instead of relying on translation to first-order automated theorem provers (ATPs) as done by the hammer systems [3, 9], the technique directly searches for sequences of tactic applications that lead to the ITP proof, thus avoiding the translation and proof-reconstruction phases needed by the hammers.

To do this, we *extract and record* tactic invocations from the ITP proofs (Section 2) and *build efficient machine learning classifiers* based on such training examples (Section 3). The learned data serves as a guidance for our *modified A*-algorithm* that explores the different proof paths (Section 4). The result, if successful, is a certified human-level proof composed of HOL4 tactics. The system is evaluated on a large set of theorems originating from HOL4 (Section 5), and we

show that the performance of the single best TacticToe strategy exceeds the performance of a hammer system used with a single strategy and a single efficient external prover.

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.

For 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 module.

The system ML4PG [22, 14] groups related proofs thanks to its clustering algorithms. It allows Coq users to inspire themselves from similar proofs and notice duplicated proofs. Our predictions comes from a much more detailed description of the open goal. However, we simply create a single label for each tactic call whereas each of its arguments is treated independently in ML4PG. Our choice is motivated by the k-NN algorithm already used in HOL(y)Hammer for the selection of theorems.

SEPIA [11] is a powerful system able to generate proof scripts from previous Coq proof examples. Its strength lies in its 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 following 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 an A*-algorithm which allows for heuristic guidance in the exploration of the search tree. Finally, SEPIA was evaluated on three chosen parts (totaling 2382 theorems) of the Coq library demonstrating that it globally outperforms individual Coq tactics. In contrast, we demonstrate the competitiveness of our system against the successful general-purpose hammers on the HOL4 standard library (7902 theorems).

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 [28], or attempted fully automatically, where such lemma selection is handed over to the ATP component of a *hammer* system [3, 9, 18, 4, 21].

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 [31] and FEMaLeCoP [19]. 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 [8] 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.

2 Recording Tactic Calls

Existing proof recording for HOL4 [33, 23] relies on manual modification of all primitive inference rules in the kernel. Adapting this approach to record tactics would require the manual modification of the 750 declared HOL4 tactics. Instead, we developed an automatic transformation on the actual proofs. Our process singles out tactic invocations and introduces calls to general purpose recording in the proofs. The main benefit of our approach is an easy access to the string representation of the tactic and its arguments which is essential to automatically construct a human-level proof script. As in the LCF-style theorem prover users may introduce new tactics or arguments with the `let` construction inline, special care needs to be taken so that the tactics can be called in any other context. The precision of the recorded information will influence the quality of the selected tactics in later searches. The actual implementation details of the recording are explained in the Appendix.

3 Predicting Tactics

The learning-based selection of relevant lemmas significantly improves the automation for hammers [4]. Therefore we propose to adapt one of the strongest hammer lemma selection methods to predict tactics in our setting: the modified distance-weighted k *nearest-neighbour* (k-NN) classifier [17, 7]. Premise selection usually only prunes the initial set of formulas given to the ATPs, which then try to solve the pruned problems on their own. Here we will use the prediction of relevant tactics to actively guide the proof search algorithm (described in Section 4).

Given a goal g, the classifier selects a set of previously solved goals similar to g, and considers the tactics that were used to solve these goals as relevant for g. As the predictor bases its relevance estimation on frequent similar goals, it is crucial to estimate the distance between the goals in a mathematically relevant way. We will next discuss the extraction of the features from the goals and the actual prediction. Both have been integrated in the SML proof search.

3.1 Features

We start by extracting the syntactic features that have been successfully used in premise selection from the goal:

- names of constants, including the logical operators,
- type constructors present in the types of constants and variables,
- first-order subterms (fully applied) with all variables replaced by a single place holder V.

We additionally extract the following features:

- names of the variables,
- the top-level logical structure with atoms substituted by a single place holder A and all its substructures.

We found that the names of variables present in the goal are particularly important for tactics such as case splitting on a variable (`Cases_on var`) or instantiation of a variable (`SPEC_TAC var term`). Determining the presence of top-level logical operators (i.e implication) is essential to assess if a "logical" tactic should be applied. For example, the presence of an implication may lead to the application of the tactic `DISCH_TAC` that moves the precondition to the assumptions. Top-level logical structure gives a more detailed view of the relationship between those logical

components. Finally, we also experiment with some general features because they are natural in higher-order logic:

- (higher-order) subterms with all variables unified, including partial function applications.

3.2 Scoring

In all proofs, we record each tactic invocation and link (associate) the currently open goal with the tactic's name in our database. Given a new open goal g, the *score of a tactic T wrt. g* is defined to be the score (similarity) of T's associated goal which is most similar to g. The idea is that tactics with high scores will be more likely to solve the open goal g, since they were able to solve similar goals before.

We estimate the similarity (or co-distance) between an open goal g_o and a previously recorded goal g_p using their respective feature sets f_o and f_p. The co-distance $tactic\_score_1$ computed by the k-NN algorithm is analogous to the one used in the premise selection task [17]. The main idea is to find the features shared by the two goals and estimate the rarity of those features calculated via the TF-IDF [16] heuristics. In a second co-distance $tactic\_score_2$, we additionally take into account the total number of features to reduce the seemingly unfair advantage of big feature sets in the first scoring function.

$$tactic\_score_1(f_o, f_p) = \sum\nolimits_{f \in f_o \cap f_p} \text{tfidf}(f)^{\tau_1}$$

$$tactic\_score_2(f_o, f_p) = \frac{tactic\_score_1(f_o, f_p)}{(1 + ln\ (1 + card\ f_o))}$$

Moreover, we would like to compare the distance of a recorded goal to different goals opened at different moment of the search. That is why we normalize the scores by dividing them by the similarity of the open goal with itself. As a result, every score will lie in the interval $[0, 1]$ where 1 is the highest degree of similarity (i.e. the shortest distance). We respectively refer to those normalized scores later as $tactic\_norm\_score_1$ and $tactic\_norm\_score_2$.

3.3 Preselection

Since the efficiency of the predictions will be crucial during the proof search, we preselect 500 tactics before the search. A sensible approach here is to preselect a tactic based on the distance between the statement of the conjecture to be proven and the statement(s) for which the tactic is part of the proof. During the proof, when an open goal is created, only the scores of the 500 preselected tactics will be recalculated and the tactics will be reordered according to these scores.

3.4 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 do not directly assign a goal to the associated tactic, but organize a competition on the closest feature vectors (tactic string together with the features of an associated goal). The winner is the tactic appearing in the most feature vectors provided that it has the same effect as the original tactic. We associate this tactic with the features of the targeted goal instead of the original in our feature database. As a result, already successful tactics are preferred, and new tactics are considered only if they provide a different contribution.

3.5 Self-learning

If the search algorithm finds a proof, we record both the human and computer-generated proof in the feature database. Since recording and re-proving are intertwined, the additional data is available for the next proof search. The hope is that it will be easier for TacticToe to learn from its own discovered proofs than from the human proof scripts [29].

4 Proof Search Algorithm

Despite the best efforts of the prediction algorithms, the selected tactic may not solve the current goal, proceed in the wrong direction or even loop. For that reason, the prediction needs 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 takes inspiration from the A*-algorithm [13] which uses a cost function and heuristics to estimate the total distance to the destination and choose the shortest route. The first necessary adaptation of the algorithm stems from the fact that a proof is in general not a path but a tree. This means that our search space has two branching factors: the choice of a tactic, and the number of goals produced by tactics. The proof is not finished when the current tactic solves its goal because it often leaves new pending open goals along the current path.

Algorithm Description In the following, we assume that we already know the distance function (it will be defined in 4.1) and describe how the A*-algorithm is transformed into a proof search algorithm. In order to help visualizing the proof steps, references to a proof search example depicted in Figure 1 will be made throughout the description of the algorithm. To minimize the width of the trees in our example the branching factor is limited to two tactics $tactic_1$ and $tactic_2$ but a typical search relies on 500 preselected tactics.

Our search algorithm starts by creating a root node containing the conjecture as an open goal. A list of 500 potential tactics is attached to this node. A score for each of those tactics is given by the tactic selection algorithm. The tactic with the best score ($tactic_2$ in our example) is applied to the conjecture. If no error occurs, it produces a new node containing a list of goals to be solved. The first of these goals ($goal_1$) is the open goal for the node, other goals ($goal_2$) are pending goals waiting for the first goal to be proved. From now, we have more than one node that can be extended, and the selection process has two steps: First, we select the best unused tactic for each open goal ($tactic_1$ for $goal_1$, $tactic_1$ for the $conjecture$). Next, we chose the node ($goal_1$) with the highest co-distance (see next paragraph) which is supposed to be the closest to finish the proof. The algorithm goes on creating new nodes with new open goals ($goal_3$) until a tactic ($tactic_2$) proves a goal ($goal_1$). This is the case when a tactic returns an empty list of goals or if all the goals directly produced by the tactic have already been proven. At this point, all branches originating from the node of the solved goal are deleted and the tactic that led to the proof is saved for a later reconstruction (see Section 4.2).

The whole process can stop in three different ways. The conjecture is proven if all goals created by a tactic applied to the conjecture are closed. The search saturates if no predicted tactics are applicable to any open goals. The process times out if it runs longer than the fixed time limit (5 seconds in our experiments).

Optimizations A number of constraints are used to speed up the proof search algorithm. We forbid the creation of nodes that contain a parent goal in order to avoid loops. We minimize

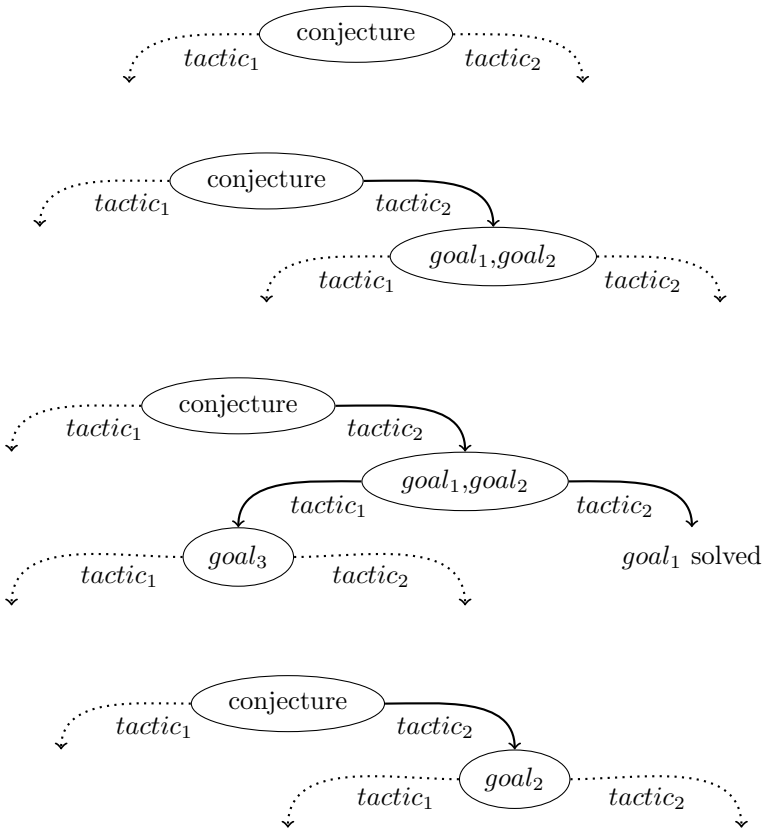

Figure 1: 4 successive snapshots of a proof attempt showing the essential steps of the algorithm: node creation, node extension and node deletion.

parallel search by imposing that two sibling nodes must not contain the same set of goals. We cache the tactic applications and the predictions so that they can be replayed quickly if the same open goals reappears anywhere in the search. Tactics are restricted to a very small time limit, the default being 0.02 seconds in our experiment. Indeed, a tactic may loop or take a large amount of time, which would hinder the whole search process. Finally, we reorder the goals in each node so that the hardest goals according to the selection heuristic are considered first.

4.1 Heuristics for Node Extension

It is crucial to define a good distance function for the (modified) A*-algorithm. This distance (or co-distance) should estimate for the edges of each node, how close it is to complete the proof. For the heuristic part, we rely on the score of the best tactic not yet applied to the node's first goal. Effectively, the prediction scores evaluate a co-distance to a provable goal, with which we approximate the co-distance to a theorem. A more precise distance estimation could be obtained by recording the provable subgoals that have already been tried [20], however

this is too costly in our setting. We design the cost function, which represents the length of the path already taken as a coefficient applied to the heuristics. By changing the parameters, we create and experiment with 5 possible co-distance functions:

$$\begin{aligned} codist_1 &= tactic\_norm\_score_1 \\ codist_2 &= tactic\_norm\_score_2 \\ codist_3(k_1) &= k_1^d * tactic\_norm\_score_1 \\ codist_4(k_1, k_2) &= k_1^d * k_2^w * tactic\_norm\_score_1 \\ codist_5(k_1, k_2) &= k_1^d * k_2^w \end{aligned}$$

where d is the depth of the considered node, w is the number of tactics previously applied to the same goal and k_1, k_2 are coefficients in $]0, 1[$.

Remark 1. If $k_1 = k_2$, the fifth co-distance has the same effect as the distance $d + w$.

Admissibility of the Heuristic and Completeness of the Algorithm An important property of the A*-algorithm is the admissibility of its heuristic. A heuristic is admissible if it does not overestimate the distance to the goal. The fifth co-distance has no heuristic, so it is admissible. As a consequence, proof searches based on this co-distance will find optimal solutions relative to its cost function. For the third and fourth co-distances, we can only guarantee a weak form of completeness. If there exists a proof involving the 500 preselected tactics, the algorithm will find one in a finite amount of time. It is sufficient to prove that eventually the search will find all proofs at depth $\leq k$. Indeed, there exists a natural number n, such that proofs of depth greater than n have a cost coefficient smaller than the smallest co-distance at depth $\leq k$. Searches based on the first two co-distances are only guided by their heuristic and therefore incomplete. This allows them to explore the suggested branches much deeper.

Remark 2. The completeness result holds only if the co-distance is positive, which happens when top-level logical structures are considered.

In the future, we consider implementing the UCT-method [6] commonly used as a selection strategy in Monte-Carlo tree search. This method would most likely find a better balance between completeness and exploration.

4.2 Reconstruction

When a proof search succeeds (there are no more pending goals at the root) we need to reconstruct a HOL4 human-style proof. The saved nodes consist of a set of trees where each edge is a tactic and the proof tree is the one starting at the root. In order to obtain a single HOL4 proof, we need to combine the tactics gathered in the trees using tacticals. By the design of the search, a single tactic combinator, `THENL`, is sufficient. It combines a tactic with a list of subsequent ones, in such a way that after the parent tactic is called, for each created goal a respective tactic from the list is called. The proof tree is transformed into a final single proof script by the following recursive function P taking a tree node t and returning a string:

$$P(t) = \begin{cases} P(c) & \text{if } t \text{ is a root,} \\ tac & \text{if } t \text{ is a leaf,} \\ tac \texttt{ THENL } [P(c_0), \ldots, P(c_n)] & \text{otherwise.} \end{cases}$$

where *tac* is the tactic that produced the node, c is the only successful child of the root and $c_0, \ldots, c_n$ are the children of the node produced by the successful tactic.

The readability of the created proof scripts is improved, by replacing replacing `THENL` by `THEN` when the list has length 1. Further post-processing such as removing unnecessary tactics and theorems has yet to be developed but would improve the user experience greatly [1].

4.3 Small "hammer" Approach

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]. As external automated reasoning techniques often outperform the combined power of the tactics, we would like to combine the TacticToe search with HOL(y)Hammer for HOL4 [9]. Moreover our approach can only use previously called tactics, so if a theorem is essential for the current proof but has never been used as an argument of a tactic, the current approach would fail.

Unfortunately external calls to HOL(y)Hammer at the proof search nodes are too computationally expensive. We therefore create a "small hammer" comprised of a faster premise selection algorithm combined with a short call to the internal prover Metis [15]. First, before the proof search, we preselect 500 theorems for the whole proof search tree using the usual premise selection algorithm with the dependencies. At each node a simpler selection process will select a small subset of the 500 to be given to Metis using a fast similarity heuristic (8 or 16 in our experiment). The preselection relies on the theorem dependencies, which usually benefits hammers, however for the final selection we only compute the syntactic feature distance works better.

During the proof search, when a new goal is created or a fresh pending goal is considered, the "small hammer" is always called first. Its call is associated with a tactic string for a flawless integration in the final proof script.

5 Experimental Evaluation

The results of all experiments are available at:

http://cl-informatik.uibk.ac.at/users/tgauthier/tactictoe/

5.1 Methodology and Fairness

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, they are recorded and included in the training examples. For each theorem we first attempt to run the TacticToe search with a time limit of 5 seconds, before processing the original proof script. In this way, the fairness of the experiments is guaranteed by construction. Only previously declared SML variables (essentially tactics, theorems and simpsets) are accessible. And for each theorem to be re-proven TacticToe is only trained on previous proofs.

Although the training process in each strategy on its own is fair, the selection of the best strategy in Section 5.2 should also be considered as a learning process. To ensure the global fairness, the final experiments in Section 5.3 runs the best strategy on the full dataset which is about 10 times larger. The performance is minimally better on this validation set.

ID	Learning parameter	Solved	$U(D_1)$
D_0	$codist_1$ (length penalty)	172 (20.0%)	5
D_1	$codist_0$ (default)	179 (20.8%)	0
D_2	no top features	175 (20.3%)	18
D_3	no higher-order features	178 (20.7%)	8

Table 1: Success rate of strategies with different learning parameters on the training set.

5.2 Choice of the Parameters

In order to efficiently determine the contribution of each parameter, we design a series of small-scale experiments where each evaluated strategy is run on every tenth goal in each theory. A smaller dataset (training set) of 860 theorems allows testing the combinations of various parameters. To compare them, we propose three successive experiments that attempt to optimize their respective parameters. To facilitate this process further, every strategy will differ from a default one by a single parameter. The results will show in addition to the success rate, the number of goals solved by a strategy not solved by another strategy X. This number is called $U(X)$.

The first experiment concerns the choice of the right kind of features and feature scoring mechanism. The results are presented in Table 1. We observe that the higher-order features and the feature of the top logical structure increase minimally the number of problems solved. It is worth noting that using only first-order features leads to 18 proofs not found by relying on additional higher-order features. The attempted length penalty on the total number of features is actually slightly harmful.

In the next experiment shown in Table 2, we focus our attention on the search parameters. To ease comparison, we reuse the strategy D_1 from Table 1 as the default strategy. We first try to change the tactic timeout, as certain tactics may require more time to complete. It seems that the initial choice of 0.02 seconds per tactic inspired by hammer experiments [10] involving Metis is a good compromise. Indeed, increasing the timeout leaves less time for other tactics to be tried, whereas decreasing it too much may prevent a tactic from succeeding. Until now, we trusted the distance heuristics completely not only to order the tactics but also to choose the next extension step in our search. We will add coefficients that reduce the scores of nodes deep in the search. From D_6 to D_8, we steadily increase the strength of the coefficients, giving the cost function of the A*-algorithm more and more influence on the search. The success rate increases accordingly, which means that using the current heuristics is a poor selection method for extending nodes. A possible solution may be to try to learn node selection independently from tactic selection. So it is not surprising that the strategy D_9 only relying on the cost function performs the best. As a minor consolation, the last column shows that the heuristic-based proof D_1 can prove 10 theorems that D_9 cannot prove. Nevertheless, we believe that the possibility of using a heuristic as a guide for the proof search is nice asset of TacticToe.

The third experiment, presented in Table 3, evaluates the effect of integrating the "small hammer" in the TacticToe search. At a first glance, the increased success rate is significant for all tested parameters. Further analysis reveals that increasing the number of premises from 8 to 16 with a timeout of 0.02 seconds is detrimental. The D_{19} experiment demonstrates that 0.1 seconds is a better time limit for reasoning with 16 premises. And the D_{18} experiment reveals the disadvantage of unnecessarily increasing the timeout of Metis. This reduces the time available for the rest of the proof search, which makes the success rate drop.

ID	Searching parameter	Solved	$U(D_9)$
D_1	$codist_0$ (default)	179 (20.8%)	10
D_4	tactic timeout 0.004 sec	175 (20.3%)	9
D_5	tactic timeout 0.1 sec	178 (20.7%)	10
D_6	$codist_3(0.8)$	192 (22.3%)	10
D_7	$codist_4(0.8, 0.8)$	199 (23.1%)	7
D_8	$codist_4(0.4, 0.4)$	205 (23.8%)	3
D_9	$codist_5(0.8, 0.8)$	211 (24.5%)	0

Table 2: Success rate of strategies with different search parameters on the training set.

ID	"small hammer" parameter	Solved	$U(D_{19})$
D_9	$codist_5(0.8, 0.8)$ (default: no small hammer)	211 (24.5%)	19
D_{16}	8 premises + timeout 0.02 sec	281 (32.7%)	21
D_{17}	16 premises + timeout 0.02 sec	270 (31.4%)	18
D_{18}	8 premises + timeout 0.1 sec	280 (32.6%)	11
D_{19}	16 premises + timeout 0.1 sec	289 (33.6%)	0

Table 3: Success rate of strategies with different parameters of "small hammer" on the training set.

The best strategy which does not rely on the "small hammer" approach D_9 will be called TacticToe (NH) (for no "small hammer") in the remaining part of the paper, and the best strategy relying on the approach, D_{19}, will be referred to as TacticToe (SH) ("small hammer").

5.3 Full-scale Experiments

We evaluate the two best TacticToe strategies on a bigger data set. Essentially, we try to reprove every theorem for which a tactic proof script was provided. The majority of theorems in the HOL4 standard library (7954 out of 10229) have been proved this way. The other theorems were created by forward rules and almost all of those proofs are bookkeeping operations such as instantiating a theorem or splitting a conjunction.

In addition we evaluate the two proposed more advanced strategies: self-learning and orthogonalization.

We will also compare the performance of TacticToe with the HOL(y)Hammer system HOL4 [9], which has so far provided the most successful general purpose proof automation. Although HOL(y)Hammer has already been thoroughly evaluated, we reevaluate its best single strategy to match the conditions of the TacticToe experiments. Therefore, this experiment is run on the current version of HOL4 with a time limit of 5 seconds. The current best strategy for HOL(y)Hammer in HOL4 is using E-prover [25, 26] with the *new_mzt_small* strategy discovered by `BliStr` [30]. To provide a baseline the less powerful *auto* strategy for E-prover was also tested.

The evaluation of TacticToe is performed as part of the HOL4 build process, whereas HOL(y)Hammer is evaluated after the complete build because of its export process. The con-

ID	Parameter	Solved	U(TacticToe(SH))
TacticToe (NH)	default	2349 (29.73%)	173
TacticToe (SH)	"small hammer"	3115 (39.42%)	$U(blistr)$: 1335
TacticToe (E_2)	self-learn	2343 (29.66%)	187
TacticToe (E_3)	self-learn + ortho	2411 (30.51%)	227
HOL(y)Hammer (auto)	E knn 128 auto	1965 (24.87%)	525
HOL(y)Hammer (blistr)	E knn 128 blistr	2556 (32.35%)	776

Table 4: Full-scale experiments and comparison of the different strategies on a common dataset of 7902 theorems

sequence is that overwritten theorems are not accessible to HOL(y)Hammer. Conversely, each theorem which was proved directly through forward rules was not considered by TacticToe. To estimate the relative strength of the two provers in a fair manner we decided to compute all subsequent statistics on the common part of the dataset. This common part consists of 7902 theorems from 134 theories.

Table 4 gathers the results of 4 TacticToe strategies and 2 HOL(y)Hammer strategies. Combining the advantages of tactic selection done by TacticToe with premise selection gives best results. Indeed, the combined method TacticToe (SH) solves 39.42% on the common dataset whereas the best HOL(y)Hammer single-strategy only solves 32.35% of the goals. Surprisingly, the effect of self learning was a little negative. This may be caused by the fact that recording both the human proof and the computer-generated script may cause duplication of the feature vectors which happens when the proofs are similar. The effect of this duplication is mitigated by the orthogonalization method which proves 62 more theorems than the default strategy. We believe that testing even stronger learning schemes is one of the most crucial steps in improving proof automation nowadays.

Table 5 compares the success rates of re-proving for different HOL4 theories. TacticToe (SH) outperforms TacticToe (NH) on every considered theory. Even if a lot weaker due to the missing premise selection component, TacticToe (NH) is hugely better than HOL(y)Hammer (blistr) in 4 theories: `measure`, `list`, `sorting` and `finite_map`. The main reason is that those theories are equipped with specialized tactics, performing complex transformation such as structural induction and TacticToe (NH) can reuse them. Conversely, HOL(y)Hammer (blistr) 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.

5.4 Reconstruction

96% of the HOL(y)Hammer (auto) (E-prover with auto strategy) proofs can be reconstructed by Metis in 2 seconds using only the dependencies returned by the prover. In comparison, all the TacticToe successful proofs could be reconstructed and resulted in proof scripts that were readable by HOL4 and solved their goals.

Furthermore, the generated proof returned by TacticToe is often more readable and informative than the single Metis call returned by HOL(y)Hammer. Since each tactic calls had a time limit of 0.02 seconds, the reconstructed proof is guaranteed to prove the goal in a very short amount of time. Those considerations indicate that often TacticToe generated scripts can contribute to the development of formal libraries in a smoother manner.

5.5 Time and Space Complexity

Here, we try to gain some insight by measuring different proof search variables. We will keep track of the total number of nodes in the proof state, the time it took to get a successful proof and the size of the final proof script. In Table 6, the number of nodes is computed over failing searches whereas the average time and the proof size is evaluated over successful searches. We estimate the total search space explored by the number of nodes. It is 4 times larger in the TacticToe (NH) version because of the lack of time consuming Metis calls. The average proof size (number of tactics in the final script) is around 3 even without explicit Metis invocations. A detailed analysis of the proofs by their size in Fig. 2 confirms the fact that most proofs are short. One of the proofs happens to be 39 steps long, but it is not a common case. This indicates the need to focus on high-quality predictions. Since the currently recorded tactics may not cover enough space, a way to generate new tactic calls may be necessary. The depiction of the numbers of problems solved in a certain amount of time in Fig. 3 shows that it is increasingly harder to solve new goals. Nevertheless, it seems that our strongest strategy TacticToe (SH) can benefit most from an increased time limit.

The total time of a search is split into 5 parts: predictions, tactic application, node creation, node selection, and node deletion. Usually in a failing search, the total prediction time takes less than a second, the tactic applications consume one to two seconds and the rest is used by the node processing parts. A simple improvement would be to reduce the bookkeeping part in a future version of TacticToe.

5.6 Case Study

Investigating further the different qualities of TacticToe, we study its generated proof scripts on an example in `list` theory (see Example 2). The theorem to be proven states the equivalence between the fact that a number n is greater than the length of a list ls with the fact that dropping n elements from this list returns an empty list.

The human proof proceeds by induction on n followed by solving both goals using rewrite steps combined with an arithmetic decision procedure. Both TacticToe proofs (NH and SH) follow the general idea of reasoning by induction but solve the base case and the inductive case in a different way. The base case only needs rewriting using the global simpset in the TacticToe (NH) proof, which is simulated by a call to Metis in the (SH) proof. The inductive case should require an arithmetic decision procedure as hinted by the human proof. This is achieved by

	arith	real	compl	meas
TacticToe (NH)	37.3	19.7	42.6	19.6
TacticToe (SH)	60.1	46.1	63.7	22.1
HOL(y)Hammer (blistr)	51.9	66.8	72.3	13.1
	proba	list	sort	f_map
TacticToe (NH)	25.3	48.1	32.7	53.4
TacticToe (SH)	25.3	51.9	34.7	55.5
HOL(y)Hammer (blistr)	25.3	23.3	16.4	18.1

Table 5: Percentage (%) of re-proved theorems in the theories `arithmetic`, `real`, `complex`, `measure`, `probability`, `list`, `sorting` and `finite_map`.

ID	nodes		proof size		time
	average	max	average	max	average
TacticToe (NH)	94.66	421	3.34	39	0.66
TacticToe (SH)	25.27	407	2.34	34	0.83

Table 6: search statistics

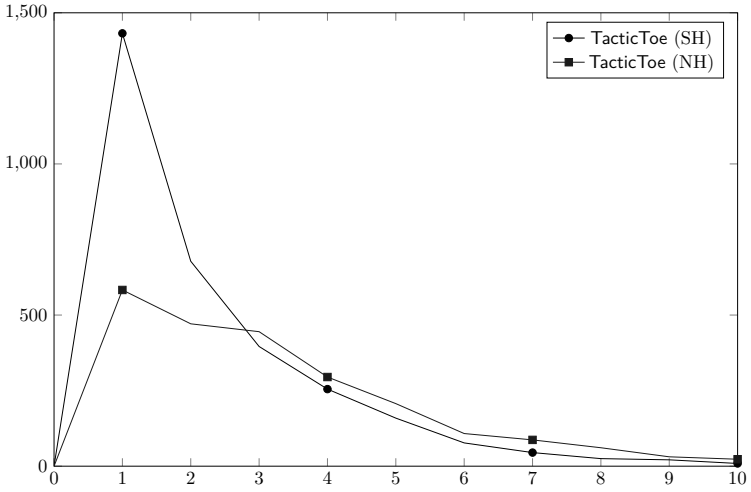

Figure 2: Number of searches (y axis) that result in a proof of size exactly x (x axis).

rewriting using an arithmetic simpset in the second proof. In the first proof however, a rewriting step and case splitting step were used to arrive at a point where Metis calls succeed. The tactic proof produced by TacticToe (NH) often looks better than the one discovered by TacticToe (SH) in that it does not involve Metis calls with a large numbers of premises.

Example 2. (In theory `list`)

```
Goal: ‘‘∀ls n. (DROP n ls = []) ⇔ n ≥ LENGTH ls‘‘

Human proof: LIST_INDUCT_TAC THEN SRW_TAC [] [] THEN DECIDE_TAC

TacticToe (NH) proof: LIST_INDUCT_TAC THENL [SRW_TAC [] [], SRW_TAC [ARITH_ss] []]

TacticToe (SH) proof:
  LIST_INDUCT_TAC THENL
  [ METIS_TAC [...],
    NTAC 2 GEN_TAC THEN SIMP_TAC (srw_ss ()) [] THEN
    Cases_on ‘n‘ THENL [METIS_TAC [...], METIS_TAC [...]] ]
```

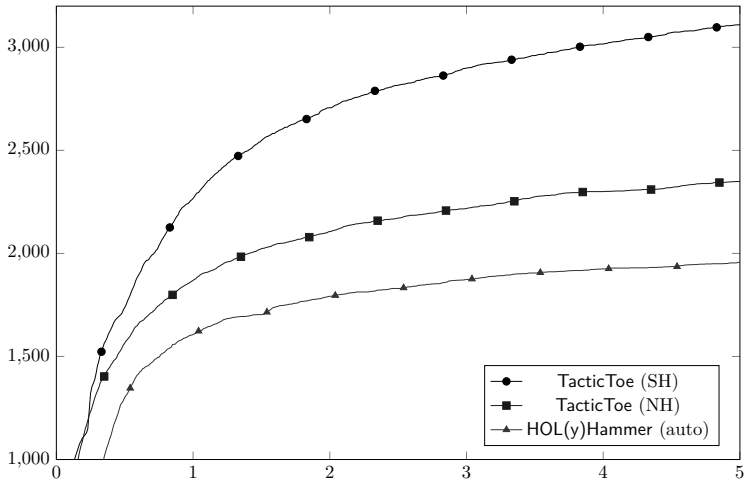

Figure 3: Number of problems solved in less than t seconds.

6 Conclusion

We proposed a new proof assistant automation technique which combines tactic-based proof search, with machine learning tactic prediction and a "small hammer" approach. Its implementation, TacticToe, achieves an overall performance of 39% theorems on the HOL4 standard library surpassing HOL(y)Hammer best single-strategy and proving 1335 additional theorems. 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 proof scripts usually 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.

There is plenty of future work in the directions opened here. To improve the quality of the predicted tactics, we would like to predict their arguments independently. To be even more precise, the relation between the tactic arguments and their respective goals could be used. Additionally, we could aim for a tighter combination with the ATP-based hammer systems. This would perhaps make TacticToe slower, but it might allow finding proofs that are so far both beyond the ATPs and TacticToe's powers. The idea of reusing high-level blocks of reasoning and then learning their selection could also be explored further in various contexts. Larger frequent blocks of (instantiated) tactics in ITPs as well as blocks of inference patterns in ATPs could be detected automatically, their usefulness in particular proof situations learned from the large corpora of ITP and ATP proofs, and reused in high-level proof search.

Acknowledgments This work has been supported by the ERC Consolidator grant no. 649043 *AI4REASON* and ERC starting grant no. 714034 *SMART*.

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] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.

[7] Sahibsingh A. Dudani. The distance-weighted k-nearest-neighbor rule. *Systems, Man and Cybernetics, IEEE Transactions on*, SMC-6(4):325–327, 1976.

[8] 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.

[9] 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.

[10] Thibault Gauthier, Cezary Kaliszyk, Chantal Keller, and Michael Norrish. Beagle as a HOL4 external ATP method. In Stephan Schulz, Leonardo De Moura, and Boris Konev, editors, *4th Workshop on Practical Aspects of Automated Reasoning (PAAR-2014)*, volume 31 of *EasyChair Proceedings in Computing*, pages 50–59. EasyChair, 2015.

[11] 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.

[12] 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.

[13] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107, 1968.

[14] Jónathan Heras and Ekaterina Komendantskaya. Recycling proof patterns in Coq: Case studies. *Mathematics in Computer Science*, 8(1):99–116, 2014.

[15] 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.

[16] Karen Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.

[17] 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.

[18] Cezary Kaliszyk and Josef Urban. Learning-assisted automated reasoning with Flyspeck. *Journal of Automated Reasoning*, 53(2):173–213, 2014.

[19] 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.

[20] Cezary Kaliszyk and Josef Urban. Learning-assisted theorem proving with millions of lemmas. *Journal of Symbolic Computation*, 69:109–128, 2015.

[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] 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.

[25] Stephan Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2-3):111–126, 2002.

[26] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.

[27] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Mohamed et al. [24], pages 28–32.

[28] Josef Urban. MPTP - Motivation, Implementation, First Experiments. *J. Autom. Reasoning*, 33(3-4):319–339, 2004.

[29] 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.

[30] Josef Urban. BliStr: The Blind Strategymaker. In Georg Gottlob, Geoff Sutcliffe, and Andrei Voronkov, editors, *Global Conference on Artificial Intelligence, GCAI 2015, Tbilisi, Georgia, October 16-19, 2015*, volume 36 of *EPiC Series in Computing*, pages 312–319. EasyChair, 2015.

[31] 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.

[32] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle framework. In Mohamed et al. [24], pages 33–38.

[33] 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.

A Recording Tactic Calls

In the appendix we present the implementation details of recording tactic calls from the LCF-style proof scripts of HOL4. We first discuss parsing the proofs and identifying individual tactic calls. We next show how tactics calls are recorded together with their corresponding goals using a modified proof script. Finally, the recorded data is organized as feature vectors with tactics as labels and characteristics of their associated goals as features. These feature vectors constitute the training set for our selection algorithm.

A.1 Extracting Proofs

Our goal is to do a first-pass parsing algorithm to extract the proofs and give them to a prerecording function for further processing. For that purpose, we create a custom theory rebuilder that parses the string representation of the theory files and modifies them. The proofs are extracted and stored as a list of global SML declarations. The rebuilder then inserts a call to the prerecorder before each proof with the following arguments: the proof string, the string representation of each declaration occurring before the proof, the theorem to be proven and its name. After each theory file has been modified, a build of the HOL4 library is triggered and each call of the prerecording function will perform the following steps: identifying tactics, globalizing tactics and registering tactic calls. The effects of those steps will be depicted on a running example taken from a proof in the list theory.

Example 3. Running example (original call)

```
val MAP_APPEND = store_thm ("MAP_APPEND",
  --‘!(f:’a->’b).!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]);
```

Example 4. Running example (extracted proof)

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

A.2 Identifying Tactics in a Proof

Parsing proofs is a more complex task than extracting them due to the presence of infix operators with different precedences. For this reason, in this phase we rely on the Poly/ML interpreter to extract tactics instead of building a custom parser. In theory, recording nested tactics is possible, but we decided to restrict ourselves to the outermost tactics, excluding those constructed by a tactical (see list in Example 5). The choice of the recording level was made to reduce the complexity of predicting the right tactic and minimizing the number of branches in the proof search. In particular, we do not consider `REPEAT` to be a tactical and record `REPEAT X` instead of repeated calls to `X`.

Example 5. `THEN ORELSE THEN1 THENL REVERSE VALID by suffices_by`

Example 6. Running example (identified tactics)

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

A.3 Globalizing Tactics

The globalization process attempts to modify a tactic string so that it is understood by the compiler in the same way anywhere during the build of HOL4. In that manner, the TacticToe proof search will be able to reuse previously called tactics in future searches. A first reason why a tactic may become inaccessible is that the module where it was declared is not open in the current theory. Therefore, during the prerecording we call the Poly/ML compiler again to obtain the module name (signature in SML) of the tactic tokens. This prefixing also avoids conflicts, where different tactics with the name appears in different module. There are however some special cases where the module of a token is not declared in a previous module. If the token is a string, already prefixed, or a SML reserved token then we do not need to perform any modifications. If a value is declared in the current theory (which is also a module), we replace the current value (or function) by its previous declaration in the file. This is done recursively to globalize the values. Theorems are treated in a special manner. Thanks to the HOL(y)Hammer

tagging system [18], they can be in most cases fetched from the HOL4 database. Terms are reprinted with their types to avoid misinterpretation of overloaded constants.

Since certain values in HOL4 are stateful (mostly references), we cannot guarantee that the application of a tactic will have exactly the same effect in a different context. This is not a common issue, as a fully functional style is preferred, however there is one important stateful structure that we need to address: the simplification set is stored globally and the simplification procedures rely on the latest version available at the moment of the proof.

Example 7. Running example (globalized tactics)

The tactic `LIST_INDUCT_TAC` is not defined in the signature of the *list* theory. That is why, to be accessible in other theories its definition appears in its globalization.

```
"Tactic.STRIP_TAC"
"let val LIST_INDUCT_TAC = Prim_rec.INDUCT_THEN
( DB.fetch \"list\" \"list_INDUCT\" ) Tactic.ASSUME_TAC in LIST_INDUCT_TAC end"
"Rewrite.ASM_REWRITE_TAC
[( DB.fetch \"list\" \"MAP\" ) , ( DB.fetch \"list\" \"APPEND\" ) ]"
```

A.4 Registering Tactic Calls

To judge the effectiveness of a tactic on proposed goals, we record how it performed previously in similar situations. For that, we modify the proofs to record and associate the globalized tactic with the goal which the original tactic received. Each original tactic is automatically modified to perform this recording as a side effect. The code of the record function `R` is defined below in Example 8. The first line checks if the globalized tactic `gtac` produces the same open goals as the original tactic. In the second line we save the globalized tactic and the features of the goal to a file. Storing features instead of goals was preferred in order to avoid unnecessary recomputation. It is also more convenient since features can be stored as a list of strings. In the running example only constant features are presented (the complete set of extracted features was discussed in Section 3.1). Finally, the original tactic is called to continue the proof.

Example 8. Pseudo-code of the recording function

```
fun R (tac,gtac) goal =
  (test_same_effect gtac tac goal; save (gtac, features_of goal); tac goal)
```

Example 9. Running example (recording proof string) R is the recording function

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

The application of the recording function `R` and its subcalls will only take place during a second HOL4 build where the proofs have been replaced by their recording variants. This replacement will be performed by a modified version of the rebuilder that extracted the proofs. It will also create a call to our search algorithm before the recording proof and a call to a post-recorder after it. The post-recorder will create feature vectors consisting of the name of the current theorem, its features and every globalized tactic in the proof. This second set is used to preselect the tactics before trying to re-prove a theorem (see Section 3.3).