Electronic Notes in Theoretical Computer Science 58 No. 2 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume58.html 14 pages

A Pragmatic Approach to Reuse in Tactical Theorem Proving

Axel Schairer

German Research Center for Artificial Intelligence (DFKI GmbH)
66111 Saarbrücken, Germany

Serge Autexier

FR 6.2 Informatik
Saarland University
66041 Saarbrücken, Germany

Dieter Hutter

German Research Center for Artificial Intelligence (DFKI GmbH)
66111 Saarbrücken, Germany

Abstract

In interactive theorem proving, tactics and tacticals have been introduced to automate proof search. In this scenario, user interaction traditionally is restricted to the mode in which the user decides which tactic to apply on the top-level, without being able to interact with the tactic once it has begun running.

We propose a technique to allow the implementation of derivational analogy in tactical theorem proving. Instead of replaying tactics including backtracked dead ends our framework makes choice points in tactics explicit and thus avoids dead ends when reusing tactics. Additionally users can override choices a tactic has made or add additional steps to a derivation without terminating the tactic. The technique depends on an efficient replay of tactic executions without repeating search that the original computation may have involved.

1 Introduction

In interactive theorem proving, tactics and tacticals have been introduced to automate proof search. In its simplest case, a tactic is a sequence of rules to be applied in order. Virtually all tactic languages provide additional control structures like iteration, conditional branching or indeterministic choice over tactics. These control structures are called tacticals. Tacticals provide a means to write more powerful tactics. Invoking a tactic on a goal produces a representation of the (partial) proof that the tactic constructs while simplifying the original goal.

Tactics decompose a goal into subgoals and call other tactics or themselves recursively to solve these subgoals. In this sense tactics are a way to structure strategic knowledge about proof search. This describes the static call graph of a set of related tactics. E.g. an induction tactic sets up the induction, applies a difference reduction tactic to the induction conclusion and hypothesis, applies the induction hypothesis, and finally simplifies the result.

Executing such a tactic gives rise to another dynamic structure, namely the tree of tactic calls that were actually carried out while solving a problem. This structure is closely related to the call graph but depends on the goal the tactic is run on. It is a summary of the call stack over the time of the tactic execution and describes the computation that was carried out to apply the tactic to a concrete goal. Because tactics involve search (usually implemented by indeterministic choice and explicit failure with backtracking) whole subtrees of this structure may correspond to branches of the search space that turned out to be dead ends. I.e. they describe computations that do not manifest themselves in the final proof representation. As an example, depending on the goal, the difference reduction tactic might call itself twice on subproblems and a simplification tactic inside one of the recursive calls. Also it might have to undo the second call to itself because the call did not produce the desired result and try again in a different way after some additional rewriting.

If we have a closer look at this structure we see two ways in which a user can interact with a tactical theorem prover. First, the user can decide which tactics to apply at the top-level and let the tactics fill in the details. I.e. the user decides on the overall structure of the proof while the machine decides on the details. This technique is applied in many theorem provers like Isabelle, PVS, Coq, KIV, Inka.

Another way the user should be able to interact with the theorem prover is to let a tactic decide on the overall proof. If the prover gets stuck the user decides on the details by overriding decisions of the tactic, inserting intermediate steps or suggesting a different way to solve a particular subproblem without otherwise terminating the tactic. I.e. after the intervention by the user the tactic carries on with what was left for it to do when the user intervened. In current theorem provers this way of interaction is not possible without changing the code of tactics and rerunning them. This has two severe disadvantages: the code of the tactics needs to be changed by the user and the tactics have to be rerun, repeating all the search that the tactic carried out before the intervention. In current theorem provers, when a tactic gives up the prover either does backtracking and leaves the user with the original proof state the tactic started out with, or it leaves a proof state corresponding to the partial success the tactic had. In any case the tactic call stack is gone and the user is back at the top-level. There is no way to intervene and then resume the original tactic at the place it gave up. That is, once a tactic has started it either produces a result fully automatically, or it does not produce a result at all. An integration of automation and interaction is not possible once a tactic has been started.

In this paper we describe a technique that enables the second mode of interaction. The basic idea is the following: while executing a tactic on a goal the prover collects a trace of the computation. Each element of this trace corresponds to a point of time in the execution of the tactic. If the user wants to suggest an intermediate step to the tactic, the prover can reexecute the initial part of the computation along the trace that it had collected earlier. When it reaches the point where the user wants to intervene it can execute a tactic that was given by the user. We call such a tactic *callback*. Since the prover has reexecuted the initial part of the computation, the callback can be executed in exactly the same situation in which the theorem prover was when the original, supposedly unsuccessful subtactic was executed. In particular all information that was collected by the execution of the tactic up to this point in time in the original execution is again present (including information entered by the user in the original execution). Our technique takes care that the reexecution of the trace is carried out in synchronization with the tactic that generated the trace in the first place. This means that not only can the callback be executed in the situation in which the original tactic would have been carried on, but also can the original tactic be resumed at exactly the same place in its execution in which it was intercepted.

The user can also override a decision by the tactic. Assume that the tactic has decided to apply a particular lemma at some point in the proof search, but the user has decided that it would be better to use another lemma. This is achieved by reexecuting the initial part of the computation up to the point at which the wrong lemma was chosen by the tactic. This decision is overriden by the lemma determined by the user and the tactic resumes with the new lemma instead of the old one.

In order to allow this mode of interaction, we assume a standard tactic language that offers facilities most tactic languages in use provide. In particular the language supports explicit choice from a list of alternatives and explicit failure to initiate backtracking. The language also supports calls to functions defined in the underlying programming language which does not support the kind of backtracking needed for implementing choice and failure. These operations interface the tactics to the proof representation and objects of the logic, and they also serve to use the full power of an efficient programming language to compute heuristic information.

Since the technique depends on the reexecution of tactics involving search it is crucial that dead branches of the search space can be cut off when repeating a computation.

The rest of this paper is organised as follows. In Sect. 2 we give a motivating example of a situation in which we can use our technique. Sect. 3

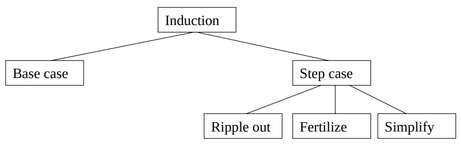

Fig. 1. Structure of example induction tactic

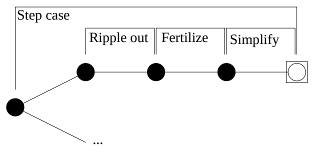

Fig. 2. Proof representation for failed proof

informally describes the technique. Sect. 4 revisits the earlier example and shows how the technique works in this case. Sect. 5 discusses how the technique can be used to reuse proofs analogically. Finally, we describe related work in Sect. 6 and conclude.

2 An Example

Suppose we want to prove the theorem $\forall x.\ \exists y.\ \mathrm{half}(y) = x$ by induction on x, given natural numbers generated by 0 and $s(\_)$ and the rewrite rules $\mathrm{half}(0) = 0$ and $\mathrm{half}(s(s(X))) = s(\mathrm{half}(X))$. We will use an induction strategy which is implemented as a tactic with the structure given in Fig. 1.

The figure shows the call graph of the tactic: ‘step case’, e.g., calls ‘ripple out’, ‘fertilize’, and ‘simplify’ in this order.

‘Induction’ sets up the base and step case. For brevity, we will ignore the base case in the rest of the example. The formula to prove in the step case is

$$\forall x.\ [\exists y.\ \mathrm{half}(y) = x] \Longrightarrow [\exists y.\ \mathrm{half}(y) = s(x)]$$

from which the quantifiers are removed by skolemization. ‘Ripple out’ uses rippling to make the induction hypothesis applicable to the induction conclusion. Rippling [3,9] is a reasoning technique exploiting the semantic knowledge that the induction hypothesis must be applied in the step case, and additionally assumes that the induction conclusion contains syntactically the induction hypothesis. The differences between induction conclusion and induction hypothesis are called *wave-fronts* and are denoted by boxes, while the common parts compose the *skeleton* and are denoted by underlining them. E.g., in our example the differences between induction conclusion and induction hypothesis are represented by

$$\mathrm{half}(y) = x \Longrightarrow \mathrm{half}(Y) = \boxed{s(\underline{x})}$$

The Rippling technique tries to apply so-called wave rules, like the definition of half

$$\mathrm{half}(\boxed{s(s(\underline{U}))}) = \boxed{s(\underline{\mathrm{half}(U)})},$$

that preserve the skeleton while moving the differences to the outside, in order to eventually end up in a situation where the induction hypothesis is a proper subterm of the induction conclusion. In the running example there is no wave rule to move the only wave front occurring in

$$\mathrm{half}(y) = x \Longrightarrow \mathrm{half}(Y) = \boxed{s(\underline{x})}$$

and the induction hypothesis is directly applicable anyway for $Y \leftarrow y$. So 'ripple out' returns and 'fertilize' applies the induction hypothesis:

$$\mathrm{half}(y) = x \Longrightarrow x = \boxed{s(\underline{x})}$$

The subsequent simplification then fails because s is a free constructor. So, the system comes back to the user saying that there is an open goal it could not solve. The resulting proof state (proof representation) that has been built and which is displayed to the user (cf. Fig. 2) includes the open goal, and links – representing reasoning steps – which are annotated by the part of the tactic that has produced them. Browsing the sequence of reasoning steps that lead to the open goal, the user will notice Y would have to be instantiated to a term different from y. And indeed, we can repair the proof by introducing an additional wave front by instantiating Y with $s(s(\_))$ before we ripple out (we will come back to this example in Sect. 4).

In current systems, what we have to do in this case is either to change the tactic code and try again from scratch, or to remove the parts of the proof representation that was built after 'normalize' and carry out the introduction of wave rules and the rest of the induction tactic manually. Since our tactics are procedural (rather than declarative), we cannot use the induction tactic in the latter case because we would have to restart it right in the middle. In Sect. 3 we describe a technique that enables us to interact with the theorem prover in such a way that we can intercept the induction tactic before 'ripple out' is called, introduce the wave front manually, and then resume the tactic as if the wave front had been introduced by the tactic itself.

3 Tactic Language

In this section we explain our technique in an informal way. We describe the semantics of tactics in continuation passing style which is a standard technique (cf. [1,8,17,15,6,5]). The idea is to describe the evaluation of an expression e with respect to (wrt.) a continuation S which represents the future of the computation after e has been evaluated. A continuation S is a function of one argument. In order to evaluate e wrt. S, the continuation S is applied to the value that e evaluates to. Evaluation of a literal l wrt. S is, therefore, described by $S(l)$. Evaluation of complex expressions is decomposed in such a way that a part of the expression is evaluated wrt. a new continuation which combines the rest of the evaluation of the complex expression and also considers the original continuation. As an example, if e is a list of expressions $e_1, e_2, \ldots, e_n$, its evaluation relative to S can be described by saying that e_1 should be evaluated (assume the result is h) wrt. the continuation that evaluates $e_2, \ldots, e_n$ (assume the result is t) and then applies the original continuation S to the list composed of the head h and the tail t. Evaluation of complex expressions can be decomposed until the expression to evaluate is a literal or variable and the rest of the evaluation is encoded in the success continuation. This defines a non-standard interpreter for the tactic programming language in continuation passing style. So we can also view the definition operationally as the way in which a tactic expression is executed. The general idea is that evaluation is flattened (or sequentialised) into a sequence of basic evalutation steps. This is in contrast to the standard evaluation strategy, which evaluates an expression by evaluating its subexpressions along its syntactic structure, its abstract syntax tree, and does not explicitly sequentialise the evaluation.

Tactics can be defined and named, e.g. as in "tactic $f(x) = e$", where the formal parameter x of the named tactic f may be free in the body e. Tactics may call other named tactics to solve subgoals of the current goal. Depending on the proof representation this necessitates setting up the proof state to focus on the subproblem before evaluating the body of the called tactic. It may also necessitate cleaning up after evaluating the body. This is dependent on the concrete proof representation, and the semantics has to take care that setting and cleaning up are done appropriately, i.e. before and after a tactic is called. Note that the interpreter knows which named tactic is being called and the actual values it is called with when it sets up the proof state. So it is possible to make the proof state dependent on the tactic that is called.

3.1 Backtracking

We assume the tactic language includes a special operator `choose` which implements search. It evaluates its argument, which should be a list of alternatives, and then chooses the first alternative for which the remaining part of the evaluation is successful, i.e. does not fail. If there is no such alternative, `choose` fails itself. As usual this is realised by backtracking and can be done straightforwardly in the setting of continuations by using a second continuation F that encodes the future of the evaluation if the evaluation of an expression fails. Again this is a standard technique covered in the literature cf. [1]. Tactics can explicitly call `fail` which is an abbreviation for `choose`([]) and fails straightaway. In order for this to work properly, side effects of tactics need to be restricted such that they can be backtracked over.

In particular, because there is an interaction between the evaluation of tactic expressions and the proof representation, the semantics has to ensure that the proof representation is adjusted when backtracking occurs. This is important in particular when updating the proof representation is done by destructively updating proof data structures. In this case, since evaluation of a tactic expression can modify the proof state between the time a choice point is set up and the time the subsequent computation fails, the proof state has to be remembered before a choice is tried and has to be rewound to the remembered state before a new alternative is tried. This has been built into the semantics of `choose` in such a way that only one computation for each choice point, corresponding to trying the first successful choice from the list of alternatives, has left modifications in the proof tree.

The net effect is that the proof representation is always well-formed and only contains parts that correspond to successful evaluation paths. This is, of course, relative to the concrete definition of the proof representation and its interface to the tactic evaluation. The technical details are out of the scope of this paper.

3.2 Replay

Remember that our aim is to be able to intervene a tactic in the middle of its execution and later resume it in the same context. The way we realise this is by reexecuting the tactic up to the point in its execution at which we want to intervene. When we reexecute a tactic, we do not want to repeat the search that was caried out when the tactic was originally executed. The idea is to remember the successful alternatives when executing a tactic and use them as an oracle for choosing the successful alternative without search on reexecution.

Therefore, we defined evaluation of a tactic expression such that it collects a trace of the sequentialised computation. A trace is a sequence the elements of which are either tactic call markers, written $\texttt{call\_to}(f)$, or choices, written $\texttt{choice}(v)$. When we start evaluating an expression e from the top-level we start with an empty trace $\langle\rangle$. We pass around the trace which we have collected so far from one step of the evaluation to the next. Whenever we reach a choice point and choose a value from the list of alternatives, say h, we add the choice we have made to the trace T we have collected so far, written $\langle T.\texttt{choice}(h)\rangle$, and then carry on collecting the trace of the rest of the computation starting with $\langle T.\texttt{choice}(h)\rangle$. Whenever we have to backtrack to this choicepoint later and choose another alternative we start with T again and add the new choice and the trace of the remaining computation to the end of T. In a similar manner we collect function call markers. Whenever we have evaluated the arguments of a named tactic f and make the next step, i.e. we evaluate the body, we extend the current trace by the tactic call marker $\texttt{call\_to}(f)$. When we backtrack over a call to a named tactic, the tactic call marker is removed from the collected trace as explained for choicepoints above. We define the trace for a tactic expression that succeeds, written $\mathit{trace}(e)$, to be the trace collected for the evaluation of e.

There is also a possibility to extract the trace of a computation that is not successful: we have defined an operation `abort` that tactics can call to give up completely, i.e. abort to the top-level. However, the trace collected so far is passed back. Tactics will call `abort` on timeout or on manual interruption. We will ignore this detail for the rest of the paper, however.

The computation trace that we have collected can be used to replay a computation without repeating the search that is involved in finding the appropriate alternative in each choice point. We use the successful choices represented in the computation trace as an oracle to choose the successful alternative without search on reexecution.

3.2.1 Simple Replay.

Evaluation of a tactic expression is now defined wrt. continuations and a computation trace t that we have collected as described above. We simply pass through t except for evaluation of `choose` and calls to named tactics. For `choose`, instead of setting up a choicepoint and searching for a successful alternative, we simply take the first event in the trace t, say `choice`(v), and use v as the value of the `choose` form without setting up a choicepoint. For calls to a named tactic f we simply ignore the first event in the computation trace, which is `call_to`(f).

This has the consequences that if $\mathit{trace}(e) = t$ then replaying the trace, i.e. evaluating e wrt. t, yields the same return value and has the same effect as the original evaluation, except that the former is more efficient: it does not carry out the search to determine successful choices because it can use the trace as an oracle to determine the correct guess.

Note that there are several possibilities to guess the successful choice. We can either, as was described above, remember and reuse the value that was chosen in the computation. Or we can remember the position of the successful choice in the list of alternatives and reuse the element at that position. Since for this paper we assume that nothing has changed with the axiomatisation or the theorem these two possibilities are equivalent. If we allowed changes to the theorem or the axiomatisation we have to transfer the decisions made for each individual choicepoint to its analogous counterpart. We will discuss this point in more detail in Sect. 5.

would have to use an entirely different notion of what “reusing the old choice” means. This is not in the scope of this paper, however.

As described in Sect. 2 the main point of the replay in our scenario is to be able to replay an initial part of a computation to recover the context for the rest of the computation. Obviously, the initial part of a computation is associated with a prefix of a computation trace. Therefore, we define evaluation of a tactic expression also wrt. a trace t which is a prefix of the trace s collected when the expression was evaluated earlier. Evaluation is now defined as for simple replay if t is non-empty, and is defined as for evaluation from scratch when t is empty. In other words, we replay the computation corresponding to the prefix t and switch back to evaluation from scratch when we run out of events in the trace. Again, if *trace*$(e) = s$ and t is a prefix of s, then replaying the trace t yields the same result as the original computation. The replay is more efficient, however, because it does not repeat the search for the initial part of the computation described by the trace t.

3.2.2 Replay with Intervention.

With the evaluation strategy given so far we can replay an intial part of the computation corresponding to the evaluation of a tactic expression, and then revert to evalution from scratch in the proper context. In order to be able to intervene the tactic at this point in the evaluation instead of continuing, we allow the last event of a trace prefix to be changed before the prefix is replayed. If the last event is `choice`(v), we allow this event to be replaced by another event `choice`(v'). Instead of choosing the alternative v as in the original computation, on reexecution v' is chosen and the tactic is resumed with v' instead of v. This allows overriding choices that the tactic has made without interrupting the tactic. As an example, a lemma that a tactic may have chosen to apply at at specific point may simply be the wrong one. By changing the event the tactic can be forced to use another lemma and carry on as before.

Another way to interact with tactics is to replace a call to a named tactic by the evaluation of another tactic expression, called a callback. Consider the example in Sect. 2 where the step case tactic applied ripple out without introducing the additional wave front. In this case, instead of calling the tactic ripple out, step case should let the user carry out a reasoning step interactively and then call ripple out and carry on with the tactic. To achieve this, if the last event of an initial trace is `call_to`(f), we allow this event to be replaced by `callback`(e). On reexecution, now instead of calling the tactic f we evaluate the expression e in the context in which f was called when t was collected. Since the context in which e is evaluated includes the lexical bindings of program variables, e may be an expression in the programming language that includes free occurrences of the program variables that were visible at the place in the original tactic definition at which f was called.

In any case, the replay collects a trace that includes the choice made to the trace before reexecution was started, and the trace for the subsequent computation. This means that the resulting trace can be replayed, in effect replaying the computation with the overridden decision without any further user intervention in the replay. This is an important property of our technique. The trace of any computation can be replayed, yielding the same collected trace. I.e. if $s =$ *trace*(e) when replaying t, then *trace*$(e) = s$ when replaying s. Also, for such an s, replaying s is equivalent to the original computation that produced s, i.e. it builds up the same proof representation.

4 The Example Revisited

We now go back to the example of Sect. 2 and show how the technique described in the previous section is applied to repair the failed proof attempt. We noted in passing in Sect. 2, that the user spotted a possible problem with the instantiation of Y to y in

$$\text{half}(y) = x \Longrightarrow \text{half}(Y) = s(x) \tag{1}$$

where Y should probably have the form $s(s(\_))$. The user will, therefore, want to retry the proof attempt with Y instantiated with $s(s(Y'))$, where Y' is a new metavariable. So, before 'ripple out' is called, the formula is changed manually to

$$\text{half}(y) = x \Longrightarrow \text{half}(s(s(Y'))) = s(x)$$

and then the induction tactic is resumed. 'Ripple out' marks the additional wave front,

$$\text{half}(y) = x \Longrightarrow \text{half}(\boxed{s(s(\underline{Y'}))}) = \boxed{s(\underline{x})}$$

applies the definition (2) of 'half' to push the left wave front outside the 'half', removes the s on both sides of the equation to make the induction hypothesis applicable.

$$\text{half}(y) = x \Longrightarrow \boxed{s(\underline{\text{half}(Y')})} = \boxed{s(\underline{x})}$$

$$\text{half}(y) = x \Longrightarrow \text{half}(Y') = x$$

At this point, 'fertilize' succeeds and without further simplification the proof is complete.

In the process of repairing this proof the user is never aware of the computation traces and their replay. Pointing to the node in the proof representation where the manual intervention should take place presents the user with a choice of possible actions, one of them being "Intervene and retry". In the example, the relevant node is the one which contains the formula (1) on which the 'ripple out' has been applied.

Internally, each node of the proof representation is annotated with the computation trace that was collected when the node was created in the failed proof attempt and the tactic that was called on the relevant proof node. Therefore, the system can execute the rippling tactic from scratch by replaying the initial trace t up to (but excluding) the call to 'ripple out' (the computation trace stored with the node). Before replaying the tactic, the computation trace t that is to be replayed is extended by the callback event which calls a tactic implementing user interaction and, when the interaction tactic returns, calls 'ripple out'. The evaluation rules of the tactic language ensure that after 'ripple out' returns, the original tactic is resumed as if the user interaction had not happened and the wave front had been inserted by the tactic itself. This only necessitates the user to make one interaction instead of simulating the rest of the induction tactic manually.

The callback and the user interaction are both stored in the trace that is collected when the successful proof is constructed. This means that when the computation needs to be replayed again later the user interaction is replayed by the system as well and no user interaction is necessary.

5 From Replay to Reuse by Analogy

As tactics decompose a goal into subgoals by calling other tactics, an implicit proof plan is encoded into the call graph of the used tactics. In our running example we decided to prove the theorem by induction. In more detail we tackled the step case by rippling out, fertilization and simplification tactics. In even more detail, we had to instantiate Y to enable the use of the definition of half inside the rippling tactic. Thus, replaying the same tactics on a different problem corresponds to reuse by analogy. In contrast to a simple replay, we have to translate the choicepoints of the orginal problem to corresponding choicepoints of the target problem and use the choice that is analogous to the choice made in the source proof. In [10] we illustrated how annotations can be used to maintain the proof history of individual symbol occurrences during the proof. Relating symbol occurrences of source and target proof if they share the same annotations (i.e. proof history), we can implement a sophisticated approach for transformational analogy. Given this mapping of symbol occurrences of source and target problem (which is automatically extended to derived formulas with the help of the annotated calculus [10]), we are able to map the choices made inside the source problem to corresponding choices to be made inside the target problem.

6 Related Work

Theorem proving by analogy has been investigated since the early days of automated theorem proving. For a long time, computational accounts of this analogy have been dominated by the idea of mapping symbols and single proof steps of a source proof to a target theorem (see e.g. [16]). This turned out to be rather insufficient and is far to limited to capture the intuitive notion of a proof by analogy. Inspired by explanation based learning techniques [14], Kolbe and Walther [12,11] developed a method for reusing proofs which is based on generalization of proofs and conjectures by replacing occurrences of function symbols by second order functions. Felty's and Howe's [7] approach on proof reuse is also based on proof generalization introducing meta-variables in proofs to find a minimal proof using given rule schemata.

Melis [13] applied the paradigm of derivational analogy to theorem proving at the more abstract planning level. The notion of derivational analogy was invented by Carbonell [4,19] for planning and general problem solving where the "lines of reasoning"(i.e. the sequence of decisions and their justifications) are replayed to solve a target problem. Melis' approach is based on the paradigm of proof planning and thus on an explicit representation of control knowledge in terms of methods and proof plans. Our approach can be easily integrated into standard tactic languages thus allowing to implement derivational analogy in a procedural (instead of declarative) control language.

Richardson and Smaill [18] use a technique similar to ours to be able to intervene with and then resume declarative methods. They store an explicit form of the continuation with the proof representation and do not collect or replay computation traces.

7 Conclusion

In this paper we have described a technique that allows user interaction in tactics without terminating the tactics. Users can override choices of the tactic and insert other proof steps to be carried out before the tactic resumes. The technique works by replaying initial parts of tactic executions and then carrying on with the computation in the same situation in which the tactic execution was intercepted. The tactic can then be resumed.

Our approach is simple and can be implemented easily in existing systems. We have implemented the semantics straightforwardly in Common Lisp using continuations. While this implementation works and is useful for experimenting with the technique, it is not efficient enough for a production strength theorem prover.

Therefore, we have implemented a much more efficient version by compiling the tactic language into common lisp code. This has several advantages: tactic expressions are analysed at compile-time, not run-time; careful compilation avoids the need to construct most of the lexical closures at run-time; program variables are translated into common lisp variables; and finally, the resulting code can be fed through the common lisp compiler, so all its optimisations apply to the tactics code for free, in particular most calls to named tactics and success continuations are tail calls and so do not consume stack space. The tactic compiler is integrated with the underlying programming language by a common lisp macro. We plan to integrate the compiled version into the INKA system [2].

The advantage of the technique is that both the user and the machine can determine the overall structure of a proof, and both can fill in details of a proof that the other is working on without interrupting it. This is not possible in existing theorem provers.

The technique is a particular simple starting point to implement derivational analogy in tactical theorem provers. More sophisticated forms of analogy can be implemented by refining the way in which choice points are transferred from the source to the target proof attempt. Also, ideas from transformational analogy can be used to patch traces before they are replayed. We will investigate these extensions in the future.

References

[1] H. Abelson and G. Sussman with J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1996.

[2] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. System description: INKA 5.0 – a logical voyager. In *Proc. 16th International Conference on Automated Deduction (CADE-16)*, 1999.

[3] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, August 1993.

[4] J.G. Carbonell. Derivational analogy: a theory of reconstructive problem solving and expertise acquisition. In R.S. Michalsky, J.G. Carbonell, and T.M.Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach.* M. Kaufmann, 1986.

[5] M. Carlsson. On implementing prolog in functional programming. *New Generation Computing*, 2(4), 1984.

[6] C. Elliott and F. Pfenning. A semi-functional implementation of a higher-order logic programming language. In Peter Lee, editor, *Topics in Advanced Language Implementation.* MIT Press, 1991.

[7] A. Felty and D. Howe. Generalization and reuse of tactic proofs. In *5th International Conference on Logic Programming and Automated Reasoning (LPAR-94)*, 1994.

[8] P. Graham. *On Lisp – Advanced Techniques for Common Lisp.* Prentice Hall, 1994.

[9] D. Hutter. Colouring terms to control equational reasoning. *Journal of Automated Reasoning*, 18(3):399–442, 1997. Kluwer-Publishers.

[10] D. Hutter. Annotated Reasoning. *Annals of Mathematics and Artificial Intelligence*, 29:183–222, 2000. Kluwer-Publishers.

[11] T. Kolbe. *Optimizing Proof Search by Machine Learning Techniques.* PhD thesis, Technische Hochschule Darmstadt, 1997.

[12] T. Kolbe and C. Walther. Reusing proofs. In *Proc. 11th European Conference on Artificial Intelligence (ECAI-94)*, 1994.

[13] E. Melis. A model of analogy-driven proof-plan construction. In *Proc. 14th International Joint Conferences on Artificial Intelligence (IJCAI-95)*, 1995.

[14] T.M. Mitchell, R.M. Keller, and S.T. Kedar-Cabelli. Explanation based generalization: a unifying view. *Machine Learning*, 1:47–80, 1986.

[15] P. Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* M. Kaufmann, 1992.

[16] S. Owen. *Analogy for Automated Reasoning.* Academic Press, 1990.

[17] J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6, 1993.

[18] J. Richardson and A. Smaill. Continuations of proof strategies. In *Proc. 4th International Workshop on Strategies in Automated Deduction (STRATEGIES 2001)*, 2001.

[19] M. Veloso and J.G. Carbonell. Towards scaling up machine learning: A case study with derivational analogy in prodigy. In S. Minton, editor, *Machine Learning Methods for Planning.* M. Kaufmann, 1993.