# Towards Learning New Methods in Proof Planning

Mateja Jamnik, Manfred Kerber, and Christoph Benzmüller
School of Computer Science, The University of Birmingham
Birmingham, B15 2TT, England, UK
{M.Jamnik|M.Kerber|C.E.Benzmuller}@cs.bham.ac.uk
http://www.cs.bham.ac.uk/~{mxj|mmk|ceb}

**Abstract.** *In this paper we propose how proof planning systems can be extended by an automated learning capability. The idea is that a proof planner would be capable of learning new proof methods from well chosen examples of proofs which use a similar reasoning strategy to prove related theorems, and this strategy could be characterised as a proof method. We propose a representation framework for methods, and a machine learning technique which can learn methods using this representation framework. The technique can be applied (amongst other) to learn whether and when to call external systems such as theorem provers or computer algebra systems.*

## 1 Introduction

Proof planning [2] is an approach to theorem proving which uses proof methods rather than low-level logical inference rules to prove a theorem at hand. A proof method specifies and encodes a general reasoning strategy that can be used in a proof, and hence represents a number of individual inference rules. For example, an induction strategy can be encoded as a proof method. Proof planners search for a proof plan of a theorem which consists of applications of several methods. An object-level logical proof can be generated from a proof plan. Proof planning is a powerful technique because it often dramatically reduces the search space, allows reuse of proof methods, and moreover generates proofs where the reasoning strategies of proofs are transparent, so they may have an intuitive appeal to a human mathematician.

One of the ways to extend the power of a proof planning system is to enlarge the set of available proof methods. Often a number of theorems can

be proved in a similar way, hence a new proof method can encapsulate the general structure, i.e., the reasoning strategy of a proof for such theorems. A difficulty in applying a strategy to many domains is that in the current proof planning systems new methods have to be implemented and added by the developer of a system. Our aim is to explore how a system can learn new methods automatically given a number of well chosen examples of related proofs of theorems. This would be a significant improvement, since examples exist typically in abundance, while the extraction of methods from these examples can be considered as a major bottleneck of the proof planning methodology. In this paper we therefore propose an approach to automatic learning of proof methods within a proof planning framework.

There is a twofold, albeit loose, relation to the *Calculemus idea*. The first relation is with respect to Kowalski's equation *algorithm* = *logic* + *control* [9] – our work aims at exploring general reasoning patterns and the control knowledge hidden in the sets of well chosen proof examples. The extracted knowledge, which is in Kowalski's sense a form of algorithmic knowledge, is represented in such a way that it can be reused for tackling new problems. The exploration of algorithmic knowledge is especially desirable in cases when this knowledge is not a priori available to a reasoning system (e.g., in form of a built-in or connected computer algebra system). The second relation is that our approach, which is described and applied here solely in the context of simplification proof examples in group theory, is not restricted only to this domain. It can analogously be applied to learn control knowledge from proof examples that already contain calls to computer algebra systems. Therefore, note that computer algebra systems may be employed in a subordinated way in proof planning by calling them within proof methods to perform particular computations [8]. Given a set of examples each of which contains probably several such calls to computer algebra systems, our approach would enable a system to learn the overall pattern of these calls (if there is one). In this sense the learnt methods also encode knowledge of the controlled usage of computer algebra systems. More generally, this argument also applies to other external reasoning system which are subordinately employed within proof methods, and is not restricted to computer algebra systems only.

Figure 1 gives a structure of our approach to learning proof methods, and hence an outline of the rest of this paper. First, we give some background and motivation for our work. In Section 2 we examine what needs to be learnt, choose our problem domain and give some examples of proofs that use a similar reasoning strategy. Then, in Section 3, the representation of methods (in the form of method specifiers and method outlines) that renders the learning process as easy as possible is discussed. We continue

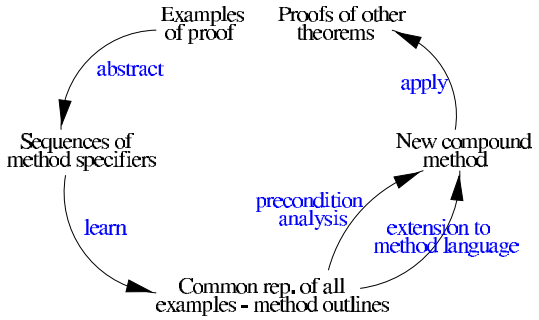


**Figure 1.** An approach to learning proof methods.

in Section 4 to present one possible learning algorithm for learning proof methods from examples of proofs. Some alternative learning techniques are also discussed. Next, in Section 5, we revisit our method representation and enrich it so that the newly learnt methods can be used in a proof planner for proofs of other theorems. We use precondition analysis to acquire the information for extending the method representation. Finally, in Section 6, we relate our work to that of others, and conclude with some future directions and final remarks.

## 2 Motivating Example

A proof method in proof planning basically consists of a triple – precondition, postcondition and a tactic. A tactic is a program which given that the preconditions are satisfied executes a number of inference steps in order to transform an expression representing a subgoal so that the postconditions are satisfied by the transformed subgoal. If no appropriate method is available in a given planning state, then the user (in case of interactive systems) or the planner (in case of automated systems) has to explicitly apply a number of lower-level methods (with inference rules as the lowest-level methods) in order to prove a given theorem. It often happens that such a pattern of lower-level methods is applied time and time again in proofs of different problems. In this case it is sensible and useful to encapsulate this inference pattern in a new proof method. Such a higher-level proof method based on lower-level methods can either be implemented and added to the system by the user or the developer of the system. Alternatively, we propose that these methods could be learnt by the system automatically.

The idea is that the system starts with learning simple proof methods. As the database of available proof methods grows, the system can learn

more complex proof methods. Initially, the user constructs simple proofs which consist only of basic inference rules rather than proof methods. A learning mechanism built into the proof planner then spots the proof pattern occurring in a number of proofs and extracts it in a new proof method. Hence, there is a hierarchy of proof steps from inference rules to complex methods. Inference rules can be treated as methods by assigning to them pre- and postconditions. Thus, from the learning perspective we can have a unified view of inference rules and methods as given sequences of elements from which the system is learning a pattern.[1]

To demonstrate our ideas with an example we need to first determine our problem domain – we choose theorems of abstract algebra, and in particular theorems of group theory. An example of a proof method is a simplification method which simplifies an expression using a number of simplification inference rules (which are in our unified view just basic-level proof methods).[2] In the case of group theorems, the simplification method may consist of applying both (left and right) axioms of identity, both axioms of inverse and the axioms of associativity. Note that $e$ is the identity element, $i$ is the inverse function, and LHS $\Rightarrow$ RHS stands for rewriting LHS to RHS:

$$
\begin{array}{rcll}
(X \circ Y) \circ Z & \Rightarrow & X \circ (Y \circ Z) & \text{(A-r)} \\
X \circ (Y \circ Z) & \Rightarrow & (X \circ Y) \circ Z & \text{(A-l)} \\
e \circ X & \Rightarrow & X & \text{(Id-l)} \\
X \circ e & \Rightarrow & X & \text{(Id-r)} \\
X \circ X^{i} & \Rightarrow & e & \text{(Inv-r)} \\
X^{i} \circ X & \Rightarrow & e & \text{(Inv-l)}
\end{array}
$$

We now give two examples of proof steps which simplify given expressions and which are concrete applications of a simplification method that we want a system to learn.

---

[1] Note that as a consequence of the hierarchic character of the method language – with methods corresponding to calculus-level rules at the lowest-level – the approach is in principle general enough to learn methods on every level of abstraction. While some heuristic information for the compound method can be computed from the component methods, learning more precise heuristic information for the compound method will be necessary. We do not address this problem in this paper.

[2] One may assume that the simplification inference rules are already learnt as basic proof methods from rewriting style of proofs employing a single rewriting rule and appropriately instantiated group axioms.

**Example 1**

$$
\begin{array}{rcl}
a \circ ((a^i & \circ & c) \circ b) \\
 & \Downarrow & \text{(A-l)} \\
(a \circ (a^i & \circ & c)) \circ b \\
 & \Downarrow & \text{(A-l)} \\
((a \circ a^i) & \circ & c) \circ b \\
 & \Downarrow & \text{(Inv-r)} \\
(e \circ c) & \circ & b \\
 & \Downarrow & \text{(Id-l)} \\
c & \circ & b
\end{array}
$$

**Example 2**

$$
\begin{array}{rcl}
a & \circ & (a^i \circ b) \\
 & \Downarrow & \text{(A-l)} \\
(a & \circ & a^i) \circ b \\
 & \Downarrow & \text{(Inv-r)} \\
e & \circ & b \\
 & \Downarrow & \text{(Id-l)} \\
 & b &
\end{array}
$$

In pseudocode, one application of simplification could be described as follows (notice that a repeated application of simplification can be learnt separately):

**Precondition:**

*there are subterms in the initial term that are inverses of each other, and that are not separated by other subterms, but only by brackets.*

**Tactic:**

1. *apply associativity* (A-r) *and/or* (A-l) *for as many times as necessary (including* 0 *times) to bring the subterms which are inverses of each other together, and then*
2. *apply inverse inference rule* (Inv-r) *or* (Inv-l) *to reduce the expression, and then*
3. *apply the identity inference rule* (Id-r) *or* (Id-l).

**Postcondition:**

*the initial term is reduced, i.e., it consists of fewer subterms.*

Note that this is a general simplification method, and the two examples given above would not be sufficient to learn it (e.g., the examples do not use (A-r), (Inv-l), (Id-r), hence additional examples that use these inference rules would have to be provided). Furthermore, our simplification method needs to be able to do loop applications of methods, where the number of loops is determined by the theorem the method is used to prove. In a sense, this type of control construct is similar to the notion of tacticals, hence we refer to them as methodicals. We are realising our ideas on learning methods in the proof planner of ΩMEGA [1] which does not explicitly represent loops. Loops in ΩMEGA are simulated by the use of control rules (see [4]).

Therefore, we are currently extending the existing representations used in ΩMEGA to provide explicit representation of loop applications of methods.

An alternative to this approach is to re-represent our problems, for instance, so as to omit the brackets in the presence of the associativity rules. However, this would be a much harder learning problem. In this paper we do not take this approach.

## 3 Method Outline Representation

The representation of a problem is of crucial importance for solving it – a good representation of a problem renders the search for its solution easy. This is a well known piece of advice from Pólya [14]. The difficulty is in finding a good representation. Our problem is to devise a mechanism for learning methods. Hence, the representation of a method is important and should make the learning process as easy as possible. Furthermore, it should be possible to represent loop applications of inference rules in methods. Here we present a simple representation formalism for methods, which abstracts away as much information as possible for the learning process, and then restores the necessary information so that the proof planner can use the newly learnt method. At the same time it caters for loop applications of inference rules in a method.

A major problem we are faced with when we want to learn compound methods from lower-level ones is the intrinsic complexity of methods, which goes beyond the complexity that can typically be tackled in the field of machine learning. For this reason we first simplify the problem by trying to learn the so-called *method outlines* (which is discussed next), and second, we reconstruct the full information by extending outlines to methods using precondition analysis (which will be discussed in Section 5).

Let us assume the following language $L$, where $P$ is a set of primitives (which are the known identifiers of methods used in a method that is being learnt). In essence, this language defines regular expressions over method identifiers. The weight $w$ defines the complexity of an expression:

- for any $p \in P$, let $p \in L$ and $w(p) = 0$,
- for any $l_1, l_2 \in L$, let $[l_1, l_2] \in L$ and $w([l_1, l_2]) = w(l_1) + w(l_2) + 1$,
- for any $l_1, l_2 \in L$, let $[l_1|l_2] \in L$ and $w([l_1|l_2]) = w(l_1) + w(l_2) + 1$,
- for any $l \in L$, let $l^*$ and $w(l^*) = w(l) + 1$.

"[" and "]" are auxiliary symbols used to separate subexpressions, "|" denotes an *exclusive-or-disjunction*, "," denotes a *sequence*, and "$*$" denotes a *repetition* of a subexpression any number of times (including 0). Let

the set of primitives $P$ be {A-l, A-r, Inv-l, Inv-r, Id-l, Id-r}. Using this language, and given the appropriate pre- and postconditions, the tactic of our simplification method described above could be expressed as:

$$\textit{simplify} \equiv \Big[\big[[\text{A-l} \mid \text{A-r}]^*,\ [\text{Inv-l} \mid \text{Inv-r}]\big],\ [\text{Id-l} \mid \text{Id-r}]\Big]$$

with $w(\textit{simplify}) = 6$. We refer to expressions in language $L$ which describe compound methods (as in the example above) as *method outlines*. *simplify* is a typical method outline that we would like our system to be able to learn automatically. The representation is simple enough that a mechanism can be devised to learn methods using this representation. We propose such a mechanism next.

## 4 How to Learn?

As explained in the previous section, we use our language $L$ for an abstract representation of methods, i.e., method outlines, in order to simplify the representation and render the learning process easier. Here we address how such method outlines can be learnt given a number of well chosen examples of proofs. Typically, there are many possible method outlines which describe a method that the system is learning. One possible approach is to learn the simplest (following Occam's razor principle) and optimal method outline, and we measure simplicity (in our first approach) in terms of weight as defined before in Section 3. Notice that our decision to use simplicity to select a possible method outline is a heuristic choice which in some cases may be inappropriate. That is, sometimes we do not prefer the simplest method outline, but perhaps the most general one, or even the least general one. As part of our future work we may have to refine our notion of simplicity.

We identified three possible approaches to learning [7]: a complete generation of method outlines, a guided generation of method outlines, and a technique similar to least general generalisation.

Exhaustive generation of all method outlines is our first attempt at a learning technique. The technique generates all possible method outlines in language $L$ of a certain weight. We then prune this set by eliminating the method outlines which could not be instantiated to the representation of the examples. The size of the set of possible method outlines increases hyper-exponentially which poses a severe problem. We can improve upon this performance by generating only the relevant method outlines – this is our second attempt at a learning technique. For instance, given any example, we could avoid generating all those method outlines that cannot be part of

the method outline to be learnt (*e.g.*, if the sequence Inv-r, Inv-r does not occur in any of the training examples then no method outline containing this sequence should be generated). This would prune the possibilities dramatically, however, it should be further examined to see if this is enough.

Although these two learning mechanisms are rather inefficient, they form a standard which can be used to measure the quality of a solution found by a procedure which is computationally more efficient.

**Generalisation** A third possible approach for a learning technique is similar to Plotkin's least general generalisation [12, 13]. Let us assume a few good examples of method outlines and the sequence of inference rules that are applied on them in order to simplify them:

1. $a \circ ((a^i \circ c) \circ b)$ which is simplified by the following sequence of application of rules [A-l, A-l, Inv-r, Id-l].
2. $a \circ (a^i \circ c)$ which is simplified by the application of [A-l, Inv-r, Id-l]. Using a sort of heuristically guided generalisation similar to least general generalisation we would get the following method outline from the first two examples: [A-l*, Inv-r, Id-l].
3. $a^i \circ (a \circ c)$ which is simplified by the application of [A-l, Inv-l, Id-l]. Adding this example, the method outline is generalised to [A-l*, [Inv-r|Inv-l], Id-l].
4. $b \circ (a \circ a^i)$ which is simplified by the application of [Inv-r, Id-r]. The method outline is now generalised to [A-l*, [Inv-r|Inv-l], [Id-l|Id-r]]. Notice that the first generalisation of A-l to A-l* is still okay here.
5. $(b \circ (c \circ a)) \circ a^i$ which is simplified by the application of [A-r, A-r, Inv-r, Id-r]. The method outline is now generalised to [[A-l*|[A-r, A-r]], [Inv-r|Inv-l], [Id-l|Id-r]].
6. $(b \circ (c \circ (d \circ a))) \circ (a^i \circ f)$ which is simplified by the application of [A-r, A-r, A-r, A-l, Inv-r, Id-r]. The method outline is finally generalised to [[A-l|A-r]*, [Inv-r|Inv-l], [Id-l|Id-r]].

Such generalisation is not guaranteed to produce an optimal solution or even any solution at all, however it *is* a computationally *feasible* technique. Devising an effective and efficient algorithm for heuristically guided generalisation will form part of our future work. Here are some preliminary ideas. There are two general principles that can be employed in the generalisation:

- whenever a method $M$ is applied a varied number of times in different examples (e.g., A-r is sometimes applied once, sometimes it is applied

twice or not at all), then the application of that method is generalised into a starred method outline $M^*$,

- whenever the methods used are different, then they are generalised into a disjunction of these methods (e.g., if in one example we use Id-l and in another at the same point in the proof we use Id-r, then these are generalised into $[\text{Id-l}|\text{Id-r}]$).

The difficult part is to distinguish between the different stages of the application of methods in the proof. For instance, using the generalisation of the first two examples above, $[\text{A-l}^*, \text{Inv-r}, \text{Id-l}]$, and the third example, $[\text{A-l}, \text{Inv-l}, \text{Id-l}]$, there are more possible generalisations than just the following: $[\text{A-l}^*, [\text{Inv-r}|\text{Inv-l}], \text{Id-l}]$. For instance, this is also possible: $[\text{A-l}^*, \text{Inv-r}^*, \text{Inv-l}^*, \text{Id-l}]$. The reason that the first generalisation was chosen is that we know that the method can be split into three parts: the first applies associativity rules, the second applies the inverse rules and the third applies the identity rules.[3] But how can we find such a partition of a method that we want to learn in general and automatically? The precondition analysis [15, 5] may help us here – we discuss this next.

## 5 Method Representation Revisited

Methods expressed using the language introduced in Section 3 do not specify what their preconditions and postconditions are. They also do not specify how the number of loop applications of inference rules is instantiated when used to prove a theorem. Hence, the method representation needs to be enriched to account for these factors. We propose to use the ideas from precondition analysis developed by Silver [15] and later extended by Desimone [5] in order to enrich our method representation.

### 5.1 Precondition Analysis

The idea of precondition analysis is to examine the reasons for applying each inference at each step of the proof. This is achieved by providing explanations for each step in the proof which are usually extracted from the information of preconditions and postconditions of a step. The preconditions of each rule used in a method are paired with additional information, namely the methods that generated these preconditions. Similarly, the postconditions of each rule used in a method are paired with the methods that use these postconditions. We extend Desimone's *method schema* representation with the *effects* that an inference rule has in the proof. Effects

[3] Notice also that the chosen method outline is of smaller weight. However, weight as defined before, (i.e., complexity) may not always be the best heuristic choice.

are used to express a change in the proof planning state which is not explicitly planned for, because, for instance, the underlying language may not be rich enough to express these changes. We demonstrate a representation of a method schema with an example.

Let there be a proof of a theorem $T$ which consists of three steps, $M_1, M_2$ and $M_3$. These methods consist of preconditions, postconditions, effects and tactics as demonstrated in the table below:

| | Preconditions | Postconditions | Effects | Tactic |
|---|---|---|---|---|
| $M_1$ | $x \wedge y \wedge p(d)$ | $x \wedge y \wedge z$ | $w = f(d)$ | $t_1$ |
| $M_2$ | $y \wedge p(w)$ | $m$ | | $t_2$ |
| $M_3$ | $z \wedge m$ | $n$ | | $t_3$ |

Notice that $p(d)$ denotes some property $p$ of $d$, where this property is a precondition of $M_1$. $f$ denotes a function which under the application of $M_1$ changes a term $d$ occurring in a precondition $p(d)$ to a term $w$. The initial state of the proof of the theorem can be described in terms of $x \wedge y \wedge p(d)$ which holds for theorem $T$. The preconditions of each method $M_i$ used in a proof of an example are analysed to determine the explanations for using these particular steps in the proof. The explanations are generated in a bottom-up fashion starting with the application of the final method. $M_3$ was obviously applied in order to reach a solution denoted by $n$ and is not an interesting case. Now consider reasons for applying method $M_2$. This is done by the analysis of the preconditions of $M_3$ which may be provided as the postconditions of $M_2$. The preconditions for method $M_3$ are $z \wedge m$. The precondition $z$ was already satisfied before the application of $M_2$,[4] but the precondition $m$ was generated as a postcondition of the application of method $M_2$. Hence, one possible explanation is that method $M_2$ was applied in order to generate precondition $m$ for method $M_3$. The same can be said for method $M_1$ – it is applied in order to generate an effect $w$ whose property $p(w)$ is a precondition for $M_2$, and a precondition $z$ for $M_3$. Together, an explanation can be that methods $M_1$ and $M_2$ are applied in order to generate the preconditions $z \wedge m$ for $M_3$.[5] Desimone captures

[4]Note that all changes to the state of the proof caused by the application of a method have to be explicitly stated in the specification of the method (i.e., in its pre-, postconditions or effects). For example, nothing is mentioned about $z$ in the specification of $M_2$, hence the application of $M_2$ preserves $z$.

[5]Using the precondition analysis as explained in [15], we get the explanations for applying all of the inference rules used in the examples. In our example, the associativity inference rules are applied to generate the preconditions for the inverse inference rules. Furthermore, the inverse inference rules are used to generate the preconditions for the identity rules. Hence, this gives the partition of the rules that we need in the generalisation learning process, as discussed in Section 4. The details of how the partitioning is done still need to be worked out.

these explanations in a method schema.

Using the language for method outlines described in Section 3 we are able to re-represent methods in the way suggested by the precondition analysis. In our example above, the method schema, say $M_4$ which is $[M_1, M_2, M_3]$ in our language $L$, can be re-represented as a method as given in Figure 2, where "*pre*" is a predicate with two arguments: the

| Pre | $x \wedge y$ | | |
|---|---|---|---|
| Tactic | | Preconditions | Postconditions |
| | $M_1$ | $pre(x, \_), pre(y, \_)$ | $post(x, \_), post(y, M_2),$ |
| | | $pre(p(d), \_)$ | $post(z, M_3)$ |
| | | | Effects: $w = f(d)$ |
| | $M_2$ | $pre(y, M_1), pre(p(w), M_1)$ | $post(m, M_3)$ |
| | $M_3$ | $pre(z, M_1), pre(m, M_2)$ | $post(n, \_)$ |
| Post | $n$ | | |

**Figure 2.** Compound method.

precondition and the method which created this precondition as a postcondition; "*post*" is a predicate with two arguments: the postcondition and the method which uses this postcondition as a precondition; "_" stands for no method, that is, the precondition is true before the application of the new method or the postcondition is not used as a precondition for any other method used in the tactic.

Analogously, a method outline with a disjunction such as $[M_1|M_2]$ can be represented as a method schema, the precondition of which is a disjunction of preconditions for $M_1$ and $M_2$. Similarly, the postcondition of this method schema is a disjunction of postconditions of $M_1$ and $M_2$. The planner has to be able to handle disjunctions of pre- and postconditions, which is a non-trivial open problem in proof planning. Extending a planner to deal with such disjunctions needs to be addressed in detail in the future. The second argument of pre- and postcondition pairs is determined as explained above. We further extend Desimone's method schema representation with a disjunction of explanations for method applications. Namely, if there is more than one method which generates/uses a pre-/postcondition, then these are combined disjunctively. This allows us to encode the fact that a postcondition of a particular rule is also a precondition of the same rule, hence, the rule can be applied several times.

Depending on the example that the method is used to prove, the pre- and postcondition pairs are instantiated differently. This determines if and how many repeated applications of a rule are needed. Hence, a method outline with a "*" such as $[M_1^*, M_2]$ has $pre(x, \_ \vee M_1), pre(y, \_ \vee M_1)$ as

preconditions of $M_1$, and $post(x, \_ \vee M_1), post(y, M_1 \vee M_2), post(z, \_)$ as postconditions of $M_1$ (the rest is as expected). Hence, if after the application of $M_1$ all the postconditions of $M_1$ satisfy its preconditions, and furthermore no other method in the tactic is applicable then $M_1$ is applied again. If its postconditions satisfy the preconditions of another method, say $M_2$, then $M_1$ is no longer applied in the proof. This is of course a heuristic decision and further research will have to examine whether it is appropriate.

## 5.2 From Outlines to Methods

Now we can represent a method outline

$$\Big[[\text{A-l}|\text{A-r}]^*, [\text{Inv-r}|\text{Inv-l}], [\text{Id-l}|\text{Id-r}]\Big]$$

as a simplification method using pairs of pre- and postconditions with methods that generate or use them, and the effects that the rules have. To save space, but still convey the main points, we only consider a simplified version of simplification, namely a method outline $[\text{A-l}^*, [\text{Inv-r}|\text{Inv-l}], \text{Id-l}]$. Notice that in order to be able to attach explanations to the inference rules in the style of precondition analysis, the method language needs to be extended. We extend it with the following vocabulary: $subt(X, Y)$ for "$X$ is a subterm of $Y$", $nb(X, Y)$ for "$X$ is a neighbour of $Y$" (two subterms of a term are neighbours if they are listed one after another when a tree representing the term is traversed in post-order), $\delta(X, Y, Z)$ for "a distance between subterms $X$ and $Y$ in a term $Z$" (the distance between two subterms is the number of nodes between the two subterms when a tree representing a term is traversed in post-order decreased by 1), and $red(E_1, E_2)$ for the fact that an expression $E_1$ is reduced to $E_2$ (an expression is reduced when it consists of fewer subterms than originally).[6] Figure 3 shows the relevant inference rules augmented with appropriate explanations. $E'$ is the term generated from $E$ by applying a corresponding method.

Figure 4 gives a method schema representation for a method outline $[\text{A-l}^*, [\text{Inv-r}|\text{Inv-l}], \text{Id-l}]$.[7] Note that $applicable(x)$ means that all the preconditions for $x$ are satisfied. Additional information that all methods assume is a position parameter which specifies a subterm on which a method is applied. This information is used in the expansion of the method to the lower-layer methods. That is, should the user want an object-level proof from a proof plan, then ultimately all the methods need to be expanded to

[6] The choice of this vocabulary is not important for this paper, and needs to be discussed elsewhere.

[7] Notice in Figure 4 that in the application of (A-l), $A_1$ matches with $A$ and $A_2$ matches with $A^i$, or $A_1$ matches with $A^i$ and $A_2$ matches with $A$.

| Rule | Preconditions | Postconditions |
|---|---|---|
| (A-r) | $subt((X \circ Y) \circ Z, E) \wedge$<br>$subt(A, Y) \wedge subt(B, Z) \wedge$<br>$nb(A, B, E) \wedge \delta(A, B, E) > 0$ | $subt(X \circ (Y \circ Z), E') \wedge$<br>$subt(A, Y) \wedge$<br>$subt(B, Z) \wedge nb(A, B, E')$<br>Effects: $\delta(A, B, E') =$<br>$\delta(A, B, E) - 1$ |
| (Id-l) | $subt(e, E)$ | $red(E, E')$ |
| (Inv-r) | $subt(X \circ X^i, E) \wedge subt(X, E) \wedge$<br>$subt(X^i, E) \wedge$<br>$nb(X, X^i) \wedge \delta(X, X^i, E) = 0$ | $subt(e, E')$<br>$red(E, E')$ |
| (Inv-l) | $subt(X^i \circ X, E) \wedge subt(X, E) \wedge$<br>$subt(X^i, E) \wedge$<br>$nb(X, X^i) \wedge \delta(X, X^i, E) = 0$ | $subt(e, E')$<br>$red(E, E')$ |

**Figure 3.** Methods with explanations for their application.

| Pre | $nb(A, A^i, E) \wedge (applicable(\text{A-l}) \vee applicable(\text{Inv-r}) \vee applicable(\text{Inv-l}))$ | | |
|---|---|---|---|
| Tactic | Tact | Preconditions | Postconditions |
| | (A-l) | $pre(subt(K \circ (L \circ M), E_1),$ _ $\vee$ (A-l))<br>$pre(subt(A_1, K),$ _ $\vee$ (A-l))<br>$pre(subt(A_2, L),$ _ $\vee$ (A-l))<br>$pre(nb(A, A^i, E_1),$ _ $\vee$ (A-l))<br>$pre(\delta(A, A^i, E_1) > 0,$ _ $\vee$ (A-l)) | $post(subt((K \circ L) \circ M, E_2),$ _)<br>$post(subt(A_1, K),$<br>(A-l) $\vee$ (Inv-r) $\vee$ (Inv-l))<br>$post(subt(A_2, L),$<br>(A-l) $\vee$ (Inv-r) $\vee$ (Inv-l))<br>$post(nb(A, A^i, E_2),$<br>(A-l) $\vee$ (Inv-r) $\vee$ (Inv-l))<br>Effects: $\delta(A, A^i, E_2) =$<br>$\delta(A, A^i, E_1) - 1$ |
| | (Inv-r) | $pre(subt(A \circ A^i, E_3),$ _ $\vee$ (A-l))<br>$pre(subt(A, E_3),$ _ $\vee$ (A-l))<br>$pre(subt(A^i, E_3),$ _ $\vee$ (A-l))<br>$pre(nb(A, A^i),$ _ $\vee$ (A-l))<br>$pre(\delta(A, A^i, E_3) = 0,$ _ $\vee$ (A-l)) | $post(subt(e, E_4),$ (Id-l))<br>$post(red(E_3, E_4),$ _) |
| | (Inv-l) | $pre(subt(A^i \circ A, E_5),$ _ $\vee$ (A-l))<br>$pre(subt(A, E_5),$ _ $\vee$ (A-l))<br>$pre(subt(A^i, E_5),$ _ $\vee$ (A-l))<br>$pre(nb(A, A^i),$ _ $\vee$ (A-l))<br>$pre(\delta(A, A^i, E_5) = 0,$ _ $\vee$ (A-l)) | $post(subt(e, E_6),$ (Id-l))<br>$post(red(E_5, E_6),$ _) |
| | (Id-l) | $pre(subt(e, E_7),$ (Inv-r) $\vee$ (Inv-l)) | $post(red(E_7, E_8),$ _) |
| Post | $red(E, E_i)$ | | |

**Figure 4.** Newly learnt compound method *simplify*.

the inference rule level. Therefore, our new *simplify* method also requires the position parameter information. Details about the extraction of a position parameter still need to be resolved, hence we do not discuss them here.

## 6 Further work and Conclusion

In this paper we introduced a language for method outlines which can be used for describing compound proof methods in proof planning on an abstract level. These methods can carry out loop applications of less complex methods and can apply them disjunctively, depending on the theorem for which they are used to prove. We also introduced a technique for learning method outlines from a number of examples of method applications. The heuristics for this generalisation technique need to be worked out in the future.

The method outlines of the introduced language can be encoded as proof method schemas in the style of Silver and Desimone in their work on precondition analysis. We demonstrated how a method outline learnt from a number of examples of the simplification method can be represented as a method schema.

Our approach is restricted to learning new higher-level proof methods on the basis of the already given ones. We cannot learn language extensions such as a coloured term language which would be a prerequisite for learning any kind of methods similar to rippling [3]. In this paper we do not address the question how such a vocabulary can be learnt by a machine. Work by Furse on MU learner [6] may be relevant for this task.

Not much work has been done in the past on applying machine learning techniques to theorem proving, in particular proof planning. We already mentioned work by Silver [15] and Desimone [5] who used precondition analysis to learn new method schemas – we explained how we use their ideas in our work. Of interest is work on generalisation [13], and other machine learning techniques such as inductive logic programming [11] and explanation based generalisation [10].

Finally, there are many open questions that remain to be answered. Here are some of them:

- Are the descriptions of preconditions, postconditions and effects as given in the example in this paper adequate to describe methods? Will a proof planner be able to use such method schemas in order to instantiate them into methods, and hence prove theorems? What type of extensions of a proof planner are needed to accommodate the use of method schemas?

- Does the representation of the method schema given in the example in this paper adequately describe the method outline represented using our language $L$? Can the method schema representation be simplified?

- So far we considered proofs which are constructed in a purely sequential rewriting style without any case splits; i.e., we considered proof chains and not general proof trees. Does our approach fully apply also to styles of proofs other than rewriting?
- Our recent experiments showed that our primitive proof methods are assumed to focus on particular subterm occurrences. These are specified by additional position parameters provided by the methods. Furthermore, the learnt *simplify* method has to come with an additional position parameter which indicates where in the expression of a theorem the method is applied. How can the parameters required for a new method be inferred from the parameters given for the primitive methods? How can the learnt method generally guide its expansion to an object-level proof by providing appropriate parameter information to the primitive methods?
- How can we most efficiently learn a general method outline in language $L$ describing a method schema? Which one of the two proposed approaches, namely guided generation of all method outlines describing the example and then pruning them, and a technique similar to Plotkin's least general generalisation, is best to use? Are there examples for which the first technique is better than the second, and vice versa? How can we determine when a technique is or is not appropriate? Is there another, more appropriate technique that we could use in order to learn new methods automatically?

Some of the answers to these questions have the potential to significantly contribute to the strength of the proof planning approach to mechanised reasoning.

## Acknowledgements

We would like to thank Alan Bundy for his continuing interest in and advice on our work, and in particular for pointing us to the work of Silver and Desimone. Furthermore we would like to thank Andreas Meier and Volker Sorge for their invaluable help in starting to implement our ideas in ΩMEGA. This work was supported by EPSRC grants GR/M22031 and GR/M99644.

## References

[1] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, and V. Sorge. Ω: Towards a mathematical assistant. In

W. McCune, editor, *14th Conference on Automated Deduction*, pages 252–255, 1997.

[2] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[3] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[4] L. Cheikhrouhou. *Forthcoming.* PhD thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany, 2000.

[5] R.V. Desimone. Learning control knowledge within an explanation-based learning framework. In I. Bratko and N. Lavrač, editors, *Progress in Machine Learning – Proceedings of 2nd European Working Session on Learning, EWSL-87*, Bled, Yugoslavia, 1987. Sigma Press. Also available from Edinburgh as DAI Research Paper 321.

[6] E. Furse. The mathematics understander. In J.H. Johnson, S. McKee, and A. Vella, editors, *Artificial Intelligence in Mathematics*. Clarendon Press, Oxford, 1994.

[7] M. Jamnik, M. Kerber, and C. Benzmüller. Towards Learning New Methods in Proof Planning. Technical Report, School of Computer Science, The University of Birmingham, CSRP-00-9, 2000.

[8] M. Kerber, M. Kohlhase, and V. Sorge. Integrating Computer Algebra Into Proof Planning. *Journal of Automated Reasoning*, 21(3):327–355, 1998.

[9] R. Kowalski. Algorithm = Logic + Control. *Communications of the Association for Computing Machinery*, 22:424–436, 1979.

[10] T.M. Mitchell, R.M. Keller, and S.T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986. Also available as Tech. Report ML-TR-2, SUNJ Rutgers, 1985.

[11] S.H. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19, 20:629–679, 1994.

[12] G. Plotkin. A note on inductive generalization. In D. Michie and B. Meltzer, editors, *Machine Intelligence 5*, pages 153–164. Edinburgh University Press, 1969.

[13] G. Plotkin. A further note on inductive generalization. In D. Michie and B. Meltzer, editors, *Machine Intelligence 6*, pages 101–126. Edinburgh University Press, 1971.

[14] G. Pólya. *How to solve it.* Princeton University Press, 1945.

[15] B. Silver. Precondition analysis: Learning control information. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning 2*. Tioga Publishing Company, 1984.