**THE UNIVERSITY
OF BIRMINGHAM**

Learning Method Outlines in Proof Planning

*Mateja Jamnik, Manfred Kerber and
Christoph Benzmüller*

CSRP-01-8
February 2001

**School of Computer Science
The University of Birmingham
Birmingham B15 2TT
United Kingdom**

*Cognitive Science
Research Papers*

Learning Method Outlines 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/

February 16, 2001

Abstract

In this paper we present a framework for automated learning within mathematical reasoning systems. In particular, this framework enables proof planning systems to automatically learn new proof methods from well chosen examples of proofs which use a similar reasoning strategy to prove related theorems. Our framework consists of a representation formalism for methods and a machine learning technique which can learn methods using this representation formalism. Our aim is to emulate some of the human informal mathematical reasoning, in particular the human learning capability on machines. This work bridges two areas of research, namely it applies machine learning techniques to advance the capability of automated reasoning systems.

1 Introduction

Proof planning [4] is an approach to theorem proving which uses proof methods rather than low level logical inference rules to find a proof of a theorem at hand. A proof method specifies and encodes a general reasoning strategy that can be used in a proof, and typically 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 successful 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. Indeed, the communication of proofs amongst mathematicians can be viewed to be on the level of proof plans.

One of the ways to extend the power of a proof planning system is to enlarge the set of available proof methods. This is particularly beneficial when a class 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 proof 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. In this work, 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 (e.g., in the form of classroom example proofs) 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 present an approach to automatic learning of proof methods within a proof planning framework. This work bridges at least two well established AI areas: automated reasoning and machine learning. From the automated reasoning point of view our work is interesting, because it aims to improve the reasoning systems by using machine learning techniques. From the machine learning point of view, our work can be seen as an interesting application of machine learning techniques in automated reasoning systems.

Our approach is ambitious and presents several contributions to the field. First, proof methods have complex structures, and are hence very hard to learn by the existing machine learning techniques. Our framework approaches this problem by abstracting as much information from the proof method representation as needed, so that the machine learning techniques can tackle it. Later, after the reasoning pattern is learnt, the abstracted information is restored as much as possible using a technique called precondition analysis.

Second, unlike in some of the existing related work [6, 12], we are not aiming to improve ways of directing proof search within a fixed set of primitives, typically inference steps. Rather, we aim to learn the primitives themselves, and hence improve the framework and reduce the search within the proof planning environment. Namely, instead of searching amongst numerous low level inference steps, a proof planner can now search only for a newly learnt proof method which encapsulates these low level inference steps.

Figure 1 gives a structure of our approach to learning proof methods, and hence an outline of the rest of this paper. Here we report only the first half of the cycle in Figure 1,

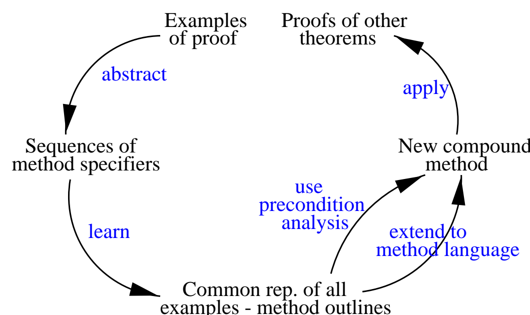

Figure 1: An approach to learning proof methods.

namely, the representation and the learning part. However, we do outline the entire framework to give a general context of the learning approach. 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 that renders the learning process as easy as possible is discussed. We continue in Section 4 to present our learning algorithm for learning proof methods from examples of proofs. An example of a running algorithm is also presented. 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. In Section 6 we discuss the implementation of the system within the proof planner of ΩMEGA [2] and give some examples of the running system. In Section 7 we relate our work to that of others. Finally, in Section 8, we conclude with some future directions and final remarks.

2 Motivating Example

A proof method in proof planning basically consists of a triple – preconditions, postconditions and a tactic. A tactic is a program which given that the preconditions are satisfied executes a number of inference steps which transform an expression representing a subgoal in a way that the postconditions are satisfied by the transformed subgoal. If no method on an appropriate level 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 present how 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.

We demonstrate our ideas with an example of a theorem in group theory. Its proof consists of simplifying an expression using a number of primitive simplification methods such as both (left and right) axioms of identity, both axioms of inverse and the axioms of associativity (where e is the identity element, i is the inverse function, and LHS $\Rightarrow$ RHS

stands for rewriting LHS to RHS):

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

Here are two examples of proof steps which simplify given expressions:

Example 1	**Example 2**
$a \circ ((a^i \circ c) \circ b)$	$a^i \circ (a \circ b)$
$\Downarrow$ (A-l)	$\Downarrow$ (A-l)
$(a \circ (a^i \circ c)) \circ b$	$(a^i \circ a) \circ b$
$\Downarrow$ (A-l)	$\Downarrow$ (Inv-l)
$((a \circ a^i) \circ c) \circ b$	$e \circ b$
$\Downarrow$ (Inv-r)	$\Downarrow$ (Id-l)
$(e \circ c) \circ b$	b
$\Downarrow$ (Id-l)	
$c \circ b$	

The first example can be summarised in the following string of method identifiers: $\{$A-l, A-l, Inv-r, Id-l$\}$. The second example can be summarised in the following string of method identifiers: $\{$A-l, Inv-l, Id-l$\}$. It is clear that the two examples have a similar structure which could be captured in a new simplification method. In pseudo-code, one application of such a simplification method could be described as follows:[1]

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-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-l).

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

[1]Note that this is not the most general simplification method, because it does not use methods such as (A-r) and (Id-r), but is the one that can be learnt from the given examples above.

3 Method Outline Representation

The representation of a problem is of crucial importance for solving it – a good representation of a problem often renders the search for its solution easy [11]. 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. 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 as much as possible so that the proof planner can use the newly learnt method.[2]

The methods we aim to learn are complex and are beyond the complexity that can typically be tackled in the field of machine learning. Therefore, we first simplify the problem and aim 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 (see 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):

- for any $p \in P$, let $p \in L$,
- for any $l_1, l_2 \in L$, let $[l_1, l_2] \in L$,
- for any $l_1, l_2 \in L$, let $[l_1|l_2] \in L$,
- for any $l \in L$, let $l^* \in L$,
- for any $l \in L$ and $n \in \mathbb{N}$, let $l^n \in L$.

"[" and "]" are auxiliary symbols used to separate subexpressions, "|" denotes a *disjunction*, "," denotes a *sequence*, "$*$" denotes a *repetition* of a subexpression any number of times (including 0), and n a fixed number of times.[3] Let the set of primitives P be $\{\text{A-l}, \text{A-r}, \text{Inv-l}, \text{Inv-r}, \text{Id-l}, \text{Id-r}\}$. Using this language, and given the appropriate pre- and postconditions, the tactic of our simplification method described by the two examples above could be expressed as:

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

We refer to expressions in language L which describe compound methods as *method outlines*. *simplify* is a typical method outline that we aim our formalism to learn automatically.

[2]Some information may be irrecoverably lost. In this case, some extra search in the application of the newly learnt methods may be necessary.

[3]The expressiveness of our language L still needs to be established. For example, proofs which are not sequences and branch into a tree may not be adequately expressible using this language. However, these may be encoded in implicit sequential representation.

4 Learning Technique

Method outlines are abstract methods which have a simple representation that is amenable to learning. We now present an algorithm which can learn method outlines from a set of well chosen examples. This learning technique may be of interest to the machine learning community, since it enables one to extract a pattern from several strings and expresses it using the language defined above. The main contribution is that the algorithm can find an adequate minimal generalisation within the given language restrictions.

Our learning algorithm is based on the generalisation of the simultaneous compression of well-chosen examples. As with compression algorithms in general, we have to compromise the expressive power of the language used for compression with the time and space efficiency of the compression process. Optimal compression – in the sense of Kolmogorov complexity – can be achieved by using a Turing-complete programming language, however, is not computable in general, i.e., there is no algorithm which finds the shortest program to represent any particular string. As a compromise we have selected regular expressions with explicit exponents, which seem to offer a framework which is on the one hand general enough for our purpose, and on the other hand – augmented with appropriate heuristics – sufficiently efficient.

There are some disadvantages to our technique, mostly related to the run time speed of the algorithm relative to the length of the examples considered for learning. The algorithm can deal with relatively small examples in an optimal way (see Section 4.2).

Our learning technique considers some number of positive examples which are represented in terms of sequences of method identifiers, and generalises them so that the learnt pattern is in language L. The pattern is most specific according to the given examples, and is of smallest size with respect to this defined measure of size (where $l_1, l_2, p \in L$, $p \in P$ and $n \in \mathbb{N}$ for some finite n):

$$\begin{aligned} size([l_1, l_2]) &= size(l_1) + size(l_2) \\ size([l_1 | l_2]) &= size(l_1) + size(l_2) \\ size(l_1^*) &= size(l_1) \\ size(l_1^n) &= size(l_1) \\ size(p) &= 1 \end{aligned}$$

Here is the learning algorithm. Given some x number of examples:

1. For every example e_i, split it into substrings of all possible lengths plus the rest of the string[4] (e.g., if $e = \{a, b, c, d\}$ then substrings of all possible fixed lengths (i.e., length 1, 2 and 3, where 4 is excluded as it cannot be compressed) are $\{\{[a], [b], [c], [d]\}, \{[a, b], [c, d]\}, \{[a], [b, c], [d]\}, \{[a, b, c], [d]\}, \{[a], [b, c, d]\}\}$).[5]

[4] Notice that there are n mod m ways of splitting an example of length n into different substrings of length m. Namely, the substrings of length m can start in positions $1, 2, \ldots n$ mod m.

[5] We introduce a terminology whereby a set of substrings consists of pattern lists pl_i, and each pattern list consists of patterns p_i.

2. For every example e_i and for every pattern list pl_i find sequential repetitions of the same patterns p_i in the same example. Using an exponent denoting the number of repetitions, compress them into p_i^c and hence pl_i^c (e.g., if $pl_i = \{[a], [a], [a], [c], [c], [d]\}$ then $pl_i^c = \{[a]^3, [c]^2, [d]\}$).

3. For every compressed pattern $p_i^c \in pl_i^c$ of every example e_i, compare it with p_j^c in all other examples e_j, and find matching p^c that have the same constituent pattern, but may have different numbers of occurrences of this pattern (e.g., if for e_1 we have $pl_1^{c1} = \{[a]^2, [b]^2\}$ and $pl_1^{c2} = \{[a, a], [b, b]\}$, and for e_2 we have $pl_2^{c1} = \{[a]^4, [c]^2\}$ and $pl_2^{c2} = \{[a, a]^2, [c, c]\}$, then the possible matches are: $m_1 = (pl_1^{c1}, pl_2^{c1})$ because $[a]^2$ and $[a]^4$ have the same constituent $[a]$, and $m_2 = (pl_1^{c2}, pl_2^{c2})$ because $[a, a]$ and $[a, a]^2$ have the same constituent $[a, a]$).

4. If there are no matches in the previous step, then generalise the examples by joining them disjunctively using the "|" constructor.

5. For every matching p_i^c, generalise different exponents to a "$*$" constructor, and the same exponents n to a constant n. (e.g., if $p_1^c = [a]^2$, $p_2^c = [a]^3$ and $p_3^c = [a]^6$, then in every example this part of the pattern list can be generalised to $p_g = [a]^*$).[6]

6. For every match p_g, transform the rest of the pattern list on the left and on the right of p_g back to the example sequence, and recursively repeat the algorithm on them. (e.g., if we have $e_1 = [[b], [a], [b], [a], [c]^2, [a]^2, [d]^4]$, $e_2 = [[b], [a], [b], [a], [c]^4, [a]^3, [d]^7]$ and $e_3 = [[b], [a], [b], [a], [c]^3, [a]^6, [d]^3]$ where the $[a]^*$ is matched and generalised in all three examples to $p_g = [a]^*$, then the algorithm is recursively repeated on $\{b, a, b, a, c, c\}$, $\{b, a, b, a, c, c, c, c\}$ and $\{b, a, b, a, c, c, c\}$ on the left of p_g and on $\{d, d, d, d\}$, $\{d, d, d, d, d, d, d\}$ and $\{d, d, d\}$ on the right of p_g).

7. If there is more than one generalisation remaining at the end of the recursive steps, then pick the one with the smallest size according to the definition above. Else, the examples cannot be generalised.

4.1 Example of Running Algorithm

Let us consider the following example sequences: $e_1 = \{a, a, a, a, b, c\}$ and $e_2 = \{a, a, a, b, c\}$. We follow the algorithm above:

1. Substrings:
   for e_1: $\{\{[a], [a], [a], [a], [b], [c]\}, \{[a, a], [a, a], [b, c]\}, \{[a], [a, a], [a, b], [c]\}, \{[a, a, a], [a, b, c]\}, \{[a], [a, a, a], [b, c]\}, \ldots\}$
   for e_2: $\{\{[a], [a], [a], [b], [c]\}, \{[a, a], [a, b], [c]\}, \{[a], [a, a], [b, c]\}, \{[a, a, a], [b, c]\}, \{[a], [a, a, b], [c]\}, \ldots\}$.

[6]Notice that here is a point where our generalisation technique can over-generalise (see Section 4.2).

2. Compressed substrings:
   $pl_1^c = \{\{[a]^4, [b], [c]\}, \{[[a, a]^2], [b, c]\}, \dots\}$
   $pl_2^c = \{\{[a]^3, [b], [c]\}, \{[a], [a, a], [b, c]\}, \dots\}$

3. Matches found (amongst other):
   $m_1 = (pl_1^{c1}, pl_2^{c1})$ due to $[a]^4$ and $[a]^3$
   $m_2 = (pl_1^{c2}, pl_2^{c2})$, due to $[b, c]$ and $[b, c]$, etc.

4. Not applicable.

5. Generalise matches:
   for m_1: $[a]^4$ and $[a]^3$ are generalised to $[a]^*$;
   for m_2: $[b, c]$ and $[b, c]$ are generalised to $[b, c]$.

6. Repeat on LHS and RHS:
   for m_1 in e_1: LHS= $\{\}$, $p_g = [a]^*$, repeat on RHS=$\{b, c\}$
   for m_1 in e_2: LHS= $\{\}$, $p_g = [a]^*$, repeat on RHS=$\{b, c\}$
   for m_2 in $e_1 1$: repeat on LHS= $\{a, a, a, a\}$, $p_g = [b, c]$, RHS= $\{\}$
   for m_2 in e_2: repeat on LHS= $\{a, a, a\}$, $p_g = [b, c]$, RHS= $\{\}$.

7. Select smallest generalisation: not applicable yet as we only have a generalisation of a part of our examples.

After the algorithm is repeated on the rest of our examples, it is easy to see that the learnt method outline will be:

$$[[a]^*, [b, c]]$$

4.2 Properties

The algorithm just given is sound, where the soundness property can be defined as: "given a number of examples, the algorithm learns a generalisation which covers all examples". This is clearly true, as the generalisation is made in step five of the algorithm and it is made on a match over all examples.

The minimality property can be defined as follows: "the algorithm learns a generalisation g which is minimal if and only if for any other generalisation h that can be learned, h is more general than g". This is not true for our algorithm yet, since it does not account for nested generalisations such as $[[a]^2]^*$.[7] This is a task for the future – an idea is to take generalised expressions of language L as primitives and run the algorithm over them again.

A desired property of the algorithm is optimality, which can be defined as follows: "the learning algorithm will always find the smallest generalisation from a certain number of given examples according to the definition of size and within the restrictions of our language

[7] For instance, if we have examples $\{a, a\}$, $\{a, a, a, a\}$ and $\{a, a, a, a, a, a\}$ then it would be reasonable to assume that the generalisation is $[[a]^2]^*$ and not $[a]^*$.

L". The intuition for the proof here is that if the algorithm is sound and minimal, and it searches amongst all possible generalisations the one with the smallest size, then it is optimal.

In terms of computational complexity, the algorithm is quadratic in step one[8] and is estimated to be exponential in step six, since we try every possible combination here. Further investigation is required to establish its exact complexity. All other steps are linear. The complexity of step six could be improved by using the initially computed information about all substrings of an example string, rather than recomputing it in every recursive step.

Since step six seems to be exponential, our learning algorithm does not run very efficiently for very large examples.[9] Hence, some optimisation may be required, which is discussed next.

4.3 Optimisation

In order to make the learning algorithm more efficient for very large examples, we can optimise its possibly exponential step six, which for all matches recursively calls the algorithm again on the left and on the right of the generalised part of the examples. This can be done by reducing the number of cases that the recursive call to the algorithms has to be carried out. Namely, rather than repeating the algorithm for all pattern matches, we use heuristics to prune this choice and pick just one pattern to consider further.

What might some good heuristics be? The general motivation for a good heuristic is that it will select one possible generalisation which makes a big impact on the final result. So far, we identified three possible heuristics (e.g., let the matches of patterns made for some examples be $m_1 = ([a,b,c]^3, [a,b,c]^3)$, $m_2 = ([a]^4, [a]^2)$ and $m_3 = ([a,b]^5, [a,b]^4)$):

- Pick a pattern match containing the pattern with the highest exponent, (e.g., this heuristic picks $[a,b]^5$ in m_3 and continues running the algorithm on m_3).
- Pick a pattern match with the smallest size (e.g., this heuristic picks $[a]^4$ in m_2).
- Pick a pattern match whose pattern of smallest length forms a maximal substring of the original example (e.g., the patterns of smallest length for each match are $[a,b,c]^3$ in m_1 whose length is 9, of $[a]^2$ in m_2 whose length is 2, and $[a,b]^4$ in m_3 whose length is 8; hence this heuristic picks $[a,b,c]^3$ in m_1 and continues running the algorithm on m_1).

[8] An example string of length n is split into all together n^2 different substrings: the are n substrings of length 1, $n-1$ substrings of length 2, $n-2$ of length 3, $n-k+1$ of length k and so on, and 1 substring of length n. Hence, in total, there are n^2 substrings of different fixed lengths. Notice that there exist algorithms, e.g., suffix trees, which run this step in linear time.

[9] However, we argue that proof methods that are being learnt typically do not consist of a very large number of low level methods. Indeed our algorithm runs very efficiently on all the tested examples (see Section 6).

We use all three listed heuristics and pick the best generalisation resulting from each. This enables the algorithm to deal with very large examples (e.g., lists of length 2000) which are way beyond the length of examples that we expect for learning our method outlines. Clearly, using such heuristic learning may miss the best (according to the measure defined above) generalisation.

5 From Method Outlines to Usable Methods

So far, we have come half way through the cycle (see Figure 1) in our approach to learning proof methods – this forms the main thrust of this paper. Namely, we have examples of proofs using low-level methods which we abstract into sequences of identifiers. We use these sequences as an input to our learning algorithm, which then learns a general pattern from them and represents it in a method outline. However, method outlines do not contain all the information which is needed for the proof planner to use them. Hence, we need to restore the missing information. In order to present the context for learning and to complete the picture we briefly mention the rest of the cycle which restores as much of the missing information as possible.

A typical method that a proof planner uses does not account for repeated applications of methods (i.e., methodicals), for disjunctive applications of methods, and for termination condition for repeated applications. Hence, we build our work on the extension of the method language to provide for these constructs, as described in [8].

Method outlines which are expressed using the language L defined in Section 3 do not specify what the preconditions and postconditions of the methods are. They also do not specify how the number of loop applications of inference rules are instantiated when used to prove a theorem. Hence, the method outlines need to be enriched to account for these factors. We use the ideas from precondition analysis developed by Silver [13] and later extended by Desimone [5] in order to enrich our method representation. Precondition analysis provides explanations for proof steps. 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 required vocabulary.[10]

6 Implementation

The framework for learning proof methods presented in this paper is implemented in the proof planner of ΩMEGA [2]. The method representation has been extended so that ΩMEGA can now apply repeated applications and disjunctions of methods. Additional vocabulary has also been implemented. There is a prototype implementation, which can generate ΩMEGA proof methods for certain outlines. There are some open problems with the translation of method outlines to usable methods. Namely, all of the parameter information for each step of the newly learnt method is not restored by the precondition analysis (e.g., the termination function

[10] To learn automatically this new vocabulary is an open problem.

for loop application of methods, the direction of reasoning, the positions in the expression that the methods are applied to). These open issues will be addressed in the future.

The main thrust of this paper, namely the learning algorithm was implemented in Standard ML of New Jersey. Its input are the sequences of proofs constructed in ΩMEGA. Its output are method outlines which are passed back to ΩMEGA. The algorithm was tested on several examples of proofs and it successfully produced the required method outlines.

Consider again the group theory examples. Example 1 ($a \circ ((a^i \circ c) \circ b)$) and Example 2 ($a^i \circ (a \circ b)$) given in Section 2 are interactively constructed in ΩMEGA, and the traces of these examples are passed on to the learning algorithm in the form of sequences of method identifiers: $\{$A-l, A-l, Inv-r, Id-l$\}$ and $\{$A-l, Inv-l, Id-l$\}$. The learning algorithm then automatically learns a method outline: $\big[$A-l*, [Inv-r|Inv-l], Id-l$\big]$, which is passed back to ΩMEGA. Modulo some of the parameter information mentioned above, which is now provided by hand, all information is available to transform the method outline into a fully fleshed usable method called *simplify*. ΩMEGA can then use this new method in the proof of a new problem, e.g., to simplify the following expression: $a^i \circ (((a \circ c) \circ b) \circ d)$.

Other examples on which we are currently testing our framework include set equations such as: $\forall x, y, z. (x \cup y) \cap z) = (x \cap z) \cup (y \cap z)$ and $\forall x, y, z. (x \cup y) \cap z) = (x \cup z) \cap (y \cup z)$. Their respective proofs (the former is valid, whereas the latter is not) are constructed automatically by ΩMEGA, and consist of repeatedly applying universal quantification introduction (in backward reasoning) ($\forall_i$), followed by set extensionality (*set-ext*), another ($\forall_i$), and definition expansion (*defni**) which produces expressions without any variables. This expression is then in the first example verified by the propositional prover (*atp-otter*), and in the second case refuted by the model generator (*counterex-satchmo*). Hence, the sequences of method identifiers look as follows: $\{\forall_i, \forall_i, \forall_i,$ *set-ext*, $\forall_i$, *defni**, *atp-otter*$\}$ and $\{\forall_i, \forall_i, \forall_i,$ *set-ext*,$\forall_i$, *defni**, *counterex-satchmo*$\}$, respectively. Our learning mechanism then automatically learns the following method outline:

$$\big[[\forall_i]^3, \textit{set-ext}, \forall_i, \textit{defni*}, [\textit{atp-otter}|\textit{counterex-satchmo}]\big].$$

Due to certain technicalities about the use of external systems such as OTTER and SATCHMO, the automatic conversion of a method outline to a fully usable method is not completed yet.

Examples of other domains on which we are currently testing our framework include residue classes theorems [9], and theorems of topology.

7 Related Work

Some work has been done in the past on applying machine learning techniques to theorem proving, in particular on improving the proof search [6, 12]. However, not much work has concentrated on high level learning of structures of proofs and extending the reasoning primitives within an automated reasoning system.

We already mentioned work by Silver [13] and Desimone [5] who used precondition analysis to learn new method schemas. In terms of a learning mechanism, the work done

on inductive inference of languages by Angluin [1] is of interest. Angluin's work differs in that it looks for various patterns, i.e., fixed sequences of characters occurring in a string, but it does not attempt to generalise repeating patterns, which is required in our problem.

The work on learning regular expressions and grammar inference [3] is also related. Learning regular expressions is equivalent to learning finite state automata (FSA) [7], which are also recognisers for regular grammars. As required in our case, these techniques can learn from positive examples only. However, they typically require either a large number of examples in order to make a reliable generalisation, or supervision or oracle which confirms when new examples are representative of the inferred generalisation.

Another related piece of work on grammatical inference methods can be found in [10]. Muggleton learns FSA by using prefix trees (similar to suffix trees) first and then generalises them into k-contextual automata. Further investigation of how our approach compares to Muggleton's technique is needed.

8 Further Work and Conclusion

In this paper we introduced a framework for automatic learning of new proof methods within a proof planning system. This framework consists of a representation formalism for methods, and a learning technique. The representation formalism consists of abstracting some information from the examples so that the learning task is eased. After the method outline is learned, it then restores as much of the necessary information as possible so that the proof planning system can use the new method in proving new problems. The learning mechanism, which is the main focus of this paper, takes as input a number of sequences of method identifiers which are abstracted from the given well chosen examples, and learns a method outline which is the smallest generalisation of the given examples. Our work presents an approach to automated reasoning using machine learning techniques, and is an interesting application on machine learning techniques to automated reasoning. Hence it bridges two well established areas of AI.

Finally, there are several aspects of our learning framework which need to be addressed in the future. With respect to the representation formalism we have only considered sequential rewriting proofs. Other styles (different directions of reasoning) and structures (trees) should also be considered. The expressiveness of our language L needs to be assessed, and L may have to be extended to allow the representation of proofs as, e.g., trees. The technical problems with the reconstruction of full methods will be addressed next. We also need to consider how to either compute the missing parameters that are not restored by the precondition analysis or to extend planning accordingly.

Regarding the learning algorithm itself, we need to examine what are good heuristics for our generalisation and how can suboptimal solutions be improved. Furthermore, our measure of size for disjunctions needs to be refined, so that it is measured with respect to the size and number of examples. Finally, we want to investigate the relation between Muggleton's k-contextual language learning and our learning algorithm.

Acknowledgements

We would like to thank Alan Bundy, Achim Jung, Andreas Meier, Stephen Muggleton and Volker Sorge for their advice and help in our research. This work was supported by EPSRC grants GR/M22031 and GR/M99644.

References

[1] D. Angluin. Inductive inference: Theory and methods. *Computing Surveys*, 15(3):237–269, 1983.

[2] 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. ΩMEGA: Towards a mathematical assistant. In W. McCune, editor, *14th Conference on Automated Deduction*, pages 252–255, 1997.

[3] A.W. Biermann and J.A. Feldman. A survey of results in grammatical inference. In S. Watanabe, editor, *Frontiers of Pattern Recognition*, pages 31–54. Academic Press, 1972.

[4] 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.

[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] M. Fuchs and M. Fuchs. Feature-based learning of search-guiding heuristics for theorem proving. *AI Communications*, 11:175–189, 1998.

[7] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[8] M. Jamnik, M. Kerber, and C. Benzmüller. Towards learning new methods in proof planning. In *Proceedings of the Calculemus 2000: 8th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, 2000. Also appeared in "Proceedings of CADE-17 Workshop The Role of Automated Deduction in Mathematics".

[9] A. Meier and V. Sorge. Exploring the domain of residue classes. In S. Colton, U. Martin, and V. Sorge, editors, *Workshop on The Role of Automated Deduction in Mathematics at CADE-17*, pages 50–54, 2000.

[10] S. Muggleton. *Acquisition of Expert Knowledge.* Addision-Wesley, 1990.

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

[12] S. Schulz. *Learning Search Control Knowledge for Equational Deduction.* PhD thesis, Fakultät für Informatik, Technische Universität München, 2000. To be published by Infix.

[13] 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.