LearnΩmatic: System Description

Mateja Jamnik[1,2], Manfred Kerber[2], and Martin Pollet[3,2]

[1]University of Cambridge Computer Laboratory
J.J. Thomson Avenue, Cambridge, CB3 0FD, England, UK
`http://www.cl.cam.ac.uk/~mj201`
[2]School of Computer Science, The University of Birmingham
Birmingham B15 2TT, England, UK
`http://www.cs.bham.ac.uk/~mmk`
[3]Fachbereich Informatik, Universität des Saarlandes, 66041 Saarbrücken, Germany
`http://www.ags.uni-sb.de/~pollet`

1 Introduction

We devised a framework within which a proof planning [1] system can learn frequently occurring patterns of reasoning automatically from a number of typical examples, and then use them in proving new theorems [2]. The availability of such patterns, captured as proof methods in a proof planning system, reduces search and proof length. We implemented this learning framework for the proof planner ΩMEGA [3], and present it in this paper – we call our system LEARNΩMATIC. The entire process of learning and using new proof methods in LEARNΩMATIC consists of the following steps:

1. The user chooses informative examples and gives them to ΩMEGA to be automatically proved. Traces of these proofs are stored.
2. Proof traces of typical examples are given to the learning mechanism which automatically learns so-called *method outlines*.
3. Method outlines are automatically enriched by adding to them additional information and performing search for information that cannot be reconstructed in order to get fully fleshed proof methods that ΩMEGA can use in proofs of new theorems.

2 Learning and Using Learnt Methods

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 simplify the problem and aim to learn *method outlines*, which are expressed in the following language L, where P is a set of known identifiers of primitive 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 $l^n \in L$,
– for any *list* such that all $l_i \in$ *list* are also $l_i \in L$, let $T(\mathit{list}) \in L$.

"[" and "]" are auxiliary symbols used to separate subexpressions, "," denotes a *sequence*, "|" denotes a *disjunction*, "$*$" denotes a *repetition* of a subexpression any number of times (including 0), n a fixed number of times, and T is a constructor for a branching point (*list* is a list of branches), i.e., for proofs which are not sequences but branch into a tree. For more information on the choice of this language, the reader is referred to [2].

Here is an example from group theory of a *simplify* method outline which applies the associativity left method, and then reduces the theorem by applying appropriate inverse and identity methods: $[\textit{assoc-l}^*, [\textit{inv-r} \mid \textit{inv-l}], \textit{id-l}]$.

Learning Technique Our learning technique considers some typically small number of positive examples which are represented in terms of sequences of identifiers for primitive methods (e.g., *assoc-l, inv-r*), and generalises them so that the learnt pattern is in language L (e.g., *simplify* given above). The pattern is of *smallest size* with respect to a defined heuristic measure of *size* [2], which essentially counts the number of primitives in an expression. The pattern is also *most specific* (or equivalently, least general) with respect to the definition of specificity *spec*. *spec* is measured in terms of the number of nestings for each part of the generalisation [2]. Again, this is a heuristic measure. We take both, the size (first) and the specificity (second), in account when selecting the appropriate generalisation. If the generalisations considered have the same rating according to the two measures, then we return all of them.

The algorithm is based on the generalisation of the simultaneous compression of well-chosen examples. Here is just an abstract description of the learning algorithm, but the detailed steps with examples of how they are applied can be found in [2]:

1. Split every example trace into sublists of all possible lengths.
2. If there is any branching in the examples, then recursively repeat this algorithm on every element of the list of branches.
3. For each sublist in each example find consecutive repetitions, i.e. patterns, and compress them using exponent representation.
4. Find compressed patterns that match in all examples.
5. If there are no matches in the previous step, then generalise the examples by joining them disjunctively.
6. For every match, generalise different exponents to a Kleene star, and the same exponents to a constant.
7. For every matching pattern in all examples, repeat the algorithm on both sides of the pattern.
8. Choose the generalisations with the smallest size and largest specificity.

For instance, the three sequences of method outlines $[\textit{assoc-l,assoc-l,inv-r,id-l}]$, $[\textit{assoc-l,inv-l,id-l}]$, and $[\textit{assoc-l,assoc-l,assoc-l,inv-r,id-l}]$ will be generalised to the *simplify* method $[\textit{assoc-l}^*, [\textit{inv-r} \mid \textit{inv-l}], \textit{id-l}]$.

The learning algorithm is implemented in SML of NJ v.110. Its inputs are the sequences of methods extracted from proofs that were 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. Properties of our learning algorithm are discussed in [2].

There are some disadvantages to our technique, mostly related to the run time of the algorithm relative to the length of the examples considered for learning. The algorithm can deal with relatively small examples, which we encounter in our application domain, in an optimal way. The complexity of the algorithm is exponential in the worst case. Hence, we use some heuristics for large and badly behaved examples [2].

Using learnt methods From a learnt outline a learnt method can automatically be generated. The learnt method is applicable if some instantiation of the method outline, i.e., a sequence of methods, is applicable. Since methods are planning operators with pre- and postconditions, these conditions must be checked for the methods of the method outline. The complex structure of methods does not allow the precondition of a subsequent method of the learnt outline to be tested, without the instantiated postconditions of the previous methods. That is, the methods of an outline have to be applied to the current proof situation.

The applicability test performs a depth first search on the learnt outline. Besides the choice points from the operators of the outline language, i.e., disjunctions and number of repetitions for the Kleene operator, there can be more than one goal where a method of the learnt outline can be applied. Additionally, for methods containing parameters, an instantiation has to be chosen. The parameters of a method are instantiated by control rules that guide the proof search. Every control rule that gives an instantiation of parameters for the current method is evaluated and the resulting possibilities for parameters are added to the search space.

The application test is performed as the precondition of the learnt method. The application of a learnt method for which the test was successful will introduce the open nodes and hypotheses generated during the applicability test as postcondition of the learnt method to the current proof.

3 Examples and Evaluation

In order to evaluate our approach, we carried out an empirical study in different problem domains on a number of theorems. This test set includes the theorems from which new methods were learnt, but most of them are new and more complex. They are from the domains of residue classes (e.g, *commutativity* of the operation $\lambda x, y.x \bar{+} y$ on the residue class set of integers $\mathbb{Z}_2$), set theory (e.g., $\forall x, y, z.((x \cup y) \cap z) = (x \cap z) \cup (y \cap z))$, and group theory (e.g., $group(G, \circ, e, i) \Rightarrow \forall a, b, c, d, f \in G.a \circ (((a^{-1} \circ b) \circ (c \circ d)) \circ f) = (b \circ (c \circ d)) \circ f)$. The learnt methods were added to the search space in a way that their applicability is checked first, before the existing standard methods are tried.

Table 1 compares the values of *matchings* and *proof length* for the three problem domains. It compares these measures when the planner searches for the proof with the standard set of available methods (column marked with S), and when in addition to these, there are also our newly learnt methods available to the planner (column marked with L). "—" means that the planner ran out of resources (four hours of CPU time) and could not find a proof plan. The counter *matchings* counts the successful and unsuccessful application tests of methods in the process of finding a complete successful proof plan. It also contains the method matchings performed by the search engine for learnt methods. *Matchings* provides an important measure, since on the one hand it indicates how directed was the performed search for a proof. On the other hand, checking the candidate methods that may be applied in the proof is by far the most expensive part of the proof search. Hence, *matchings* is a good measure to approximate the *time* needed by the two approaches (i.e., with and without learnt methods) while it is also independent of the concrete implementation inefficiencies.

We tested the system with (and without) the use of the following learnt methods: for residue classes we used two learnt methods, *tryanderror* and *choose*, for set theory we used one learnt method, and for group theory we learnt five new methods, but only used two, since these two are recursive applications of the others. As is evident from Table 1, the number of candidate methods that the planner has to check if they can be applied in the proof (i.e., *matchings*) is reduced in all domains where our newly learnt methods are available. In general, the more complicated the theorem, the better is the improvement made by the availability of the learnt methods. In the case of group theory, some complex theorems can be proved *only* within the resource limits when our learnt methods are available to the planner. Hence, the *coverage* of the system that uses learnt methods is increased. Furthermore, we noticed that for some very simple theorems of group theory, a larger number of *matchings* is required if the learnt methods are available in the search space. However, for more complex examples, this is no longer the case, and an improvement is noticed. The reason for this behaviour is that additional methods increase the search space, and the application test for learnt methods is expensive, especially when a learnt method is not applicable, but still all possible interpretations of the learnt method outline have to be checked by the search engine.

As expected, the *proof length* is much reduced by using learnt methods, since they encapsulate patterns in which several other methods are used in the proof.

On average, the *time* it took to prove theorems of residue classes and conjectures of set theory was up to 50% and 15% shorter, respectively, than without

Domain	Theorems	Matchings		Length	
		S	L	S	L
Residue Class (using *tryanderror* method)	assoc-z3z-times	651	113	63	2
	assoc-z6z-times	4431	680	441	2
	average of all	1362.0	219.5	134.0	2.0
Residue Class (using *choose* method)	closed-z3z-plusplus	681	551	49	34
	closed-z6z-plusplus	3465	2048	235	115
	average of all	1438.8	918.3	101.0	57.3
Set theory	average of all	33.5	12.5	13.0	2.0
Group theory	average of all (simple)	94.2	79.0	15.5	8.3
Group theory	average of all (complex)	—	189.6	—	9.8

Table 1. Evaluation results.

such methods. The search in group theory took approximately 100% longer than without the learnt methods. The time results reflect in principle the behaviour of the proof search measured by method *matchings*, but also contain the overhead due to the current implementation for the reuse of the learnt methods. For example, the current proof situation is copied for the applicability test of the learnt method, and the new open goals and hypotheses resulting from a successful application are copied back into the original proof.

The reason for the improvements described above is due to the fact that our learnt methods provide a structure according to which the existing methods can be applied, and hence they direct search. This structure also gives better explanation why certain methods are best applied in particular combinations. For example, the simplification method for group theory examples indicates how the methods for associativity, inverse and identity should be combined together, rather than be applied blindly in any possible combination.

4 Future Work and Availability

There are several limitations of our approach that could be improved in the future. Namely, the learning algorithm may overgeneralise, so we need to examine what are good heuristics for our generalisation and how suboptimal solutions can be improved. In order to reduce unnecessary steps, the preconditions of the learnt methods would ideally be stronger. Currently, we use an applicability test to search if the preconditions of the method outline are satisfied. In the future, preconditions should be learnt as well. Finally, in order to model the human learning capability in theorem proving more adequately it would be necessary to model how humans introduce new vocabulary for new (emerging) concepts.

A demonstration of LearnΩmatic implementation can be found on the following web page: http://www.cs.bham.ac.uk/~mmk/demos/LearnOmatic/. Further information, also with links to papers with more comprehensive references can be found on http://www.cs.bham.ac.uk/~mmk/projects/MethodFormation/.

Acknowledgements We would like to thank Alan Bundy, Predrag Janičić, Achim Jung, and Stephen Muggleton for their helpful advice on our work, and Christoph Benzmüller, Andreas Meier, and Volker Sorge for their help with some of the implementation in Ωmega. This work was supported by EPSRC grant GR/M22031 and European Commission IHP Calculemus Project grant HPRN-CT-2000-00102.

References

1. Bundy, A.: The use of explicit plans to guide inductive proofs. In 9th Conference on Automated Deduction. LNCS 310, Springer (1988), 111–120.
2. Jamnik, M., Kerber, M., Pollet, M., Benzmüller, C.: Automatic learning of proof methods in proof planning. Technical Report CSRP-02-05, School of Computer Science, The University of Birmingham, Birmingham, England, UK, (2002). ftp://ftp.cs.bham.ac.uk/pub/tech-reports/2002/CSRP-02-05.ps.gz
3. Benzmüller, C., *et al.*: Ωmega: Towards a mathematical assistant. In 14th Conference on Automated Deduction. LNAI 1249, Springer (1997), 252-255.