Typed meta-interpretive learning for proof strategies

Colin Farquhar[1], Gudmund Grov[1], Andrew Cropper[2], Stephen Muggleton[2], and Alan Bundy[3]

[1] Heriot-Watt University, Edinburgh, UK, {cif30,G.Grov}@hw.ac.uk
[2] Imperial College, London, UK {a.cropper13,s.muggleton}@imperial.ac.uk
[3] University of Edinburgh, UK bundy@staffmail.ed.ac.uk

Abstract. Formal verification is increasingly used in industry. A popular technique is *interactive theorem proving*, used for instance by Intel in HOL light. The ability to learn and re-apply proof strategies from a small set of proofs would significantly increase the productivity of these systems, and make them more cost-effective to use. Previous attempts have had limited success, which we believe is a result of missing key goal properties in the strategies. Capturing such properties will require predicate invention, and the only technique we are familiar which supports this is *meta-interpretive learning* (MIL). We show that MIL is applicable to this problem, but that it offers limited improvements over previous work. We then extend MIL with *types* and give preliminary results indicating that this extension learns better strategies with suitable goal properties.

1 Introduction

The expressiveness of (higher order) *interactive theorem provers* have made them a popular choice for formalised mathematics and software verification[4]. However, the expressiveness comes at the expense of proof automation: users often have to manually provide step-by-step guidance of the proofs, where each step applies a *proof tactic* that splits a goal into smaller and simpler sub-goals.

A commonly observed phenomenon is that proofs often group into families, such that once the expert user has discharged one proof the same pattern can be applied to the rest [2]. For a common user, the remaining proofs have to be manually guided as well.If one could learn and reapply proof strategies from one (or a few) example(s) then this could significantly increase proof automation, making the overall approach more cost-effective - a key bottleneck for industrial application of software verification - and provide support for more elegant proofs.

Previous attempts to learn proof strategies [4,6] focused on tactic composition, which does not capture *when* a sub-strategy should be applied. This will either result in a large, possibly non-terminating, search space, or a hardcoding of heuristics which may rule out some proofs. This deficiency was part of our motivation in developing the *PSGraph* language [5], which incorporates information on the tactics and (sub-)goals. This is achieved by representing proof strategies as graphs, where proof tactics live on the nodes and goal information is represented as predicates which label the wires.

[4] See e.g. the *AFP* [afp.sourceforge.net] and L4.verified [sel4.systems].

Meta-Interpretive Learning (MIL) [12] was designed to learn from such small sets of examples. It supports predicate invention, which is required to learn details of the goals due to their rich and recursive nature. In this paper, we first show that MIL is capable of learning proof strategies for the PSGraph language (C1). As we are working with very rich data, we show that the proof strategies MIL extracts have a high branching factor giving a large search space. We therefore say they are *highly non-deterministic*, where a deterministic strategy has a single branch. Non-determinism is undesirable as the search space becomes large and the strategies hard to maintain. We claim that nondeterminism can be overcome by introducing typing into MIL (C2) and validate the claim C3 that *"typed MIL learns more deterministic proof strategies than (untyped) MIL."*

2 Related work

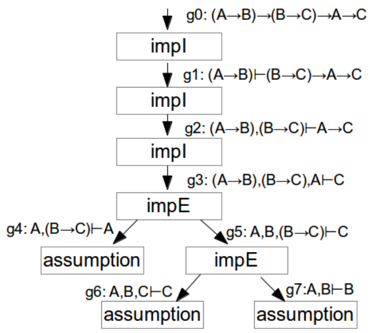
Machine learning has been very successful within *automated theorem proving* where it has been used to select relevant hypothesis and has considerably increased proof automation (see e.g. [7]). This problem is orthogonal to ours, as our aim is to automate proofs that require additional *proof guidance* which, as far as we know, have not been tackled there. [8] uses machine learning to provide hints for the user, but does not generalise proofs into strategies.

The most relevant works that attempt to learn *proof strategies* are the LearnOmega system [6] and Duncan's PhD thesis [4]. In both cases, a regular expression language is used to represent the proof strategies. This language enables generalisations through repetition (Kleene star) and choice. Duncan uses a combination of genetic algorithms and statistical methods while the LearnOmega system develops its own machine learning algorithm. The drawback of these approaches are that they are not able to learn which branch to choose and when to stop repeating. [6] supports, but does not learn, conditions on the goals.

Progol [10] uses mode declarations to indicate the *types* of variables allowed within atoms in hypothesised clauses. *Dependent MIL* [9] takes a layered approach to learning, where each layer learns predicates used at the higher layers. One could see this a special instance of *typed predicate* learning in which each layer learns predicates of a given type. However, this will not work when there are mutual dependencies between predicates of different types.

3 Framework

MIL [11] is an ILP technique aimed at supporting learning of recursive definitions. A powerful and novel aspect of MIL is that when learning a predicate definition it automatically introduces sub-definitions, allowing decomposition into a hierarchy of reuseable parts. MIL is based on an adapted version of a Prolog meta-interpreter. Normally such a meta-interpreter derives a proof by repeatedly fetching first-order Prolog clauses whose heads unify with a given goal. By contrast, a meta-interpretive learner additionally fetches higher-order metarules whose heads unify with the goal, and saves the resulting meta-substitutions to form a program. Our work uses the Metagol$_{DF}$ implementation [9] of MIL. First, we extend this framework with simple types:




```
lemma (A ⟶ B) ⟶ (B ⟶ C) ⟶ A ⟶ C
 apply (rule impI)
 apply (rule impI)
 apply (rule impI)
 apply (erule impE)
 apply assumption
 apply (erule impE)
 apply assumption
 apply assumption
 done
```

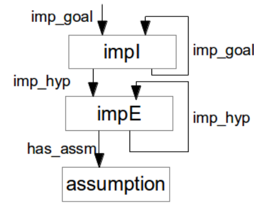


Fig. 1. Left to right: Proof tree, Isabelle proof script; and strategy as PSGraph.

Definition 1 (Typed Meta-Interpretive Learning). *In* typed MIL*, each predicate and argument in the background, examples and meta-rules are tagged with a constant t_i denoting its type. To illustrate, typing $P(X, Y)$ becomes:*

$$P : t_1(X : t_2, Y : t_3).$$

To unify two predicates their types must also unify. Types for predicates, e.g. $P(X : t_2, Y : t_3)$, or arguments, e.g. $P : t_1(X, Y)$, may be omitted if they have a single type. We call these argument typed MIL *and* predicate typed MIL*.*

Our work will use *predicate typed MIL*. Note that in order to work the Metagol$_{DF}$ framework, the predicate type is represented as an additional argument: e.g. $P : t_1(X, Y)$ is internally represented as $P(t_1, X, Y)$.

In our experiments we apply (typed) MIL to proofs from the state-of-the-art Isabelle theorem prover [13]. Figure 1 (left) illustrates a proof tree acting as an example to learn from. This tree has been generated from the *proof script* (middle) using the *ProofProcess* framework [14]. In the proof script, each tactic is preceded by the **apply** keyword and works by splitting a single goal into a list of new sub-goals. Note that each tactic may introduce branching, and the script/tree only shows one of the branches – in other cases the user needs to backtrack. Our goal is to generalise this proof into a proof strategy in PSGraph that can be applied to "similar proofs", as illustrated on the right of the figure. Here, repeated application has been generalised to a loop, where the *'wire predicates'* capture the cases where it should loop and where the loop should terminate. Also note that a wire can hold multiple goals, and that the graph is *open*: a wire without a source represents an input, while a wire without a destination represents an output. A proof is created by sending a goal down an input edge. This will apply the tactic at the destination to it, and send the new sub-goals to its output wires. Crucially, the wire predicate has to succeed for a given goal, and if multiple output wires succeeds then this will introduce a branching in the search space. For more details see [5].

We have fully automated the translation of the Isabelle proof scripts into Prolog clauses in order to apply Metagol to it. This encoding introduces four distinct types: *psgraph* for the PSGraph we are trying to learn; *tactic* for the underlying tactics of the theorem prover; *wpred* for the wire predicates and *gdata* for data associated with the goals. The proof tree structure is handled by a binary predicate for each tactic application, with one predicate for each branching. E.g. the step that turns $g3$ into $g4$ and $g5$ is represented by the two clauses:

$$erule\_impE : tactic(g3, g4). \qquad erule\_impE : tactic(g3, g5).$$

A (sub-)goal contains a set of *hypotheses* and a *goal*, e.g. $g4$ is $A, (B \longrightarrow C) \vdash A$. These *terms* are projected from the goals by *hyp:gdata* and *concl:gdata*. To illustrate, the edge predicate used by the *assumption* tactic requires the same term to be in the hypothesis and conclusion:

$$has\_asm : wpred(G) \leftarrow hyp{:}gdata(G, T), concl{:}gdata(G, T).$$

Isabelle internally stores terms as typed lambda expressions [13]. We have simplified their encoding by omitting type information and lambda expressions (as this is so far not used in our examples): a term is thus either a constant *const* , encoded as $c(const)$, or an application of two terms t_1 and t_2, written $app(t_1, t_2)$. The goal information for $g4$ thus becomes:

$$concl : gdata(g4, c(A)).$$
$$hyp : gdata(g4, c(A)). \quad hyp : gdata(g4, app(app(c(\longrightarrow), c(B)), c(C))).$$

In addition, we provide operators *left:gdata* and *right:gdata* to project the right and left sub-terms of an application and *const:data* to check if a term is a constant. These are provided to the learning, in addition to some additional information discussed in the next section. To learn a PSGraph we use the following *typed metarules*:

$$P : psgraph(X, Y) \leftarrow Q : wpred(X), R : tactic(X, Y). \tag{1}$$
$$P : psgraph(X, Y) \leftarrow Q : psgraph(X, Z), R : psgraph(Z, Y). \tag{2}$$
$$P : psgraph(X, Y) \leftarrow Q : psgraph(X, Z), P : psgraph(Z, Y). \tag{3}$$

(1) lifts a tactic to PSGraph with a single node (R) and an input wire with a predicate (Q); (2) sequentially composes two PSGraphs with an edge (Z) between them[5]; (3) is used to handle recursion (feedback loops in the graphs). Note that in our case, the metarules can be seen as giving the semantics for the *psgraph* type and to ensure that a valid PSGraph is learnt, as the type forces the learner to only use the above metarules since the positive examples will be of the same type. For the example of Figure 1, the positive examples are each branch (as this is an AND tree), i.e. to learn S, we give $S : psgraph(g0, g4)$, $S : psgraph(g0, g6)$ and $S : psgraph(g0, g7)$. Finally, note that by using types, we can have an arbitrary rich set of metarules for *wpred* to learn e.g. $Q : wpred(X)$.

4 Experiments

We have experimented with untyped and typed MIL to learn proof strategies from a collection of 15 proofs in propositional logic[6]. In addition to the metarules

[5] This wire is labelled by the label of the input wire of R.

[6] The examples are taken from: isabelle.in.tum.de/exercises

and examples discussed in §3, tactic definitions are provided as background information. The experiments were run using YAP on Ubuntu using a 3.10 GHz Intel i5-2400 CPU with 4GB RAM. A cut-off point of 5 minutes has been used, with the following key results [7]:

	success	mean nodes	mean clauses	mean br	mean evals
Untyped	13	4	3	9	1
Typed	2	5	3	1	2

As the table shows, untyped MIL managed to learn a strategy from 13 of the examples (86%), which dropped to 2 (13%) for typed MIL. The table also illustrates that the mean branching factor is considerably lower for typed MIL, while the mean number of nodes and clauses are similar. A more realistic comparison is made when we compare data for the 2 examples where typed MIL succeeded:

	nds (tree)	nds (UT)	nds (T)	cls (UT)	cls (T)	brs (UT)	brs (T)	evals (UT)	evals (T)	fails (UT)	fails (T)
a	3	2	6	2	3	3	1	1	0	13	14
b	2	2	4	1	3	1	1	0	4	14	10

This table shows the number of nodes (nds), clauses (cls), branches (brs) and successful and failed evaluations of the remaining 14 proofs, both untyped (UT) and typed (T). The table confirms that types reduce branching, but need more clauses. The extra clauses required to capture the wire predicates are the reason typed MIL fails in most examples, and will be addressed in future work.

The wire predicates had to be provided as background information as initial experiments in inventing them had limited success. Untyped MIL ignores the predicate as it is not needed to find the simplest possible solution. This is overcome in typed MIL, as we can enforce wire predicates with the metarules. However, inventing a *wpred* in terms of *gdata* has so far failed. Finding a solution to this is ongoing work.

5 Conclusion and further work

We have been able to learn proof strategies from 86% of the examples and have therefore validated our claim (C1) that the MIL framework is capable of learning proof strategies. However, in terms of branching, it seems to offer no improvements over previous work [4,6] as it ignores learning the required goal properties. We have introduced types in the MIL framework by adding an additional constant argument to the predicate (claim C2). While the results show that typed MIL learns goal properties and reduces branching, we were only able to learn strategies from 13% of the examples. We have therefore only partly validated our main claim (C3) that typed MIL reduces non-determinism. Validating this claim requires further experiments. The introduction of types means a larger number of clauses to represent larger strategies, which drastically increases the execution time for Metagol and is the reason for failure in most cases. This problem has

[7] Code with all experiments available at: www.macs.hw.ac.uk/ cif30/ilp15.zip.

to be addressed first. Examples including *rewriting*, which most real-world examples do, is likely to provide more convincing arguments for types as they may not terminate without the goal properties. We have started with examples from group theory, including those used by [6][8], and to see if we can learn the *rippling* proof strategy [1], which will require inventing very complex wire predicates.

We further plan to compare the *generality* of the strategies: our experiments suggest that this will require learning from multiple examples. However Metagol timed out for all examples we attempted. We may also need negative examples, which can be automatically extracted from failed branches when executing strategies. We would also like to show the advantages of typed MIL for other domains: the approach we have taken should be applicable for most cases where labelled graphs are learnt, while we have started experimenting on extending previous work on learning robot strategies [3] with argument types. Longer term, we plan to study 'type invention' and support for 'higher order types'.

Acknowledgments. This work has been supported by EPSRC grant EP/J001058/1, and the first author is supported by a James Watt scholarship. The fourth author acknowledges support from his Royal Academy of Engineering/Syngenta Research Chair.

References

1. A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning.* Cambridge University Press, 2005.
2. A. Bundy, G. Grov, and C. B. Jones. Learning from experts to aid the automation of proof search. In *AVoCS'09*, CSR-2-2009, pages 229–232. Swansea Uni., 2009.
3. A. Cropper and S. Muggleton. Learning efficient logical robot strategies involving composable objects. In *IJCAI*, 2015. To appear.
4. H. Duncan. *The use of Data-Mining for the Automatic Formation of Tactics.* PhD thesis, University of Edinburgh, 2002.
5. G. Grov, A. Kissinger, and Y. Lin. A graphical language for proof strategies. In *LPAR*, volume 8312 of *LNCS*, pages 324–339. Springer, 2013.
6. M. Jamnik, M. Kerber, M. Pollet, and C. Benzmüller. Automatic learning of proof methods in proof planning. *Logic Journal of IGPL*, 11(6):647–673, 2003.
7. C. Kaliszyk and J. Urban. Learning-assisted automated reasoning with flyspeck. *JAR*, 53(2):173–213, 2014.
8. E. Komendantskaya, J. Heras, and G. Grov. Machine learning in proof general: Interfacing interfaces. In *UITP 2012*, pages 15–41, 2013.
9. D. Lin, E. Dechter, K. Ellis, J. Tenenbaum, and S. Muggleton. Bias reformulation for one-shot function induction. In *ECAI*, pages 525–530, 2014.
10. S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13:245–286, 1995.
11. S. Muggleton, D. Lin, N. Pahlavi, and A. Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Machine Learning*, 94:25–49, 2014.
12. S. Muggleton, D. Lin, and A. Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. *Machine Learning*, 2015. Published online: DOI 10.1007/s10994-014-5471-y.
13. L. C. Paulson. The foundation of a generic theorem prover. *JAR*, 5(3):363–397, 1989.
14. A. Velykis. *Capturing Proof Process.* PhD thesis, Newcastle University, 2015.

[8] Available from: [bit.ly/1IEWu3H]