# 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 a.bundy@ed.ac.uk

**Abstract.** Formal verification of computer programs 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 learning attempts have had limited success, which we believe is a result of missing key goal properties in the strategies. Capturing such properties requires predicate invention, and the only state-of-the-art ILP technique which supports this is *meta-interpretive learning* (MIL). We show that MIL is applicable to this problem, but that without type information it offers limited improvements in quality over previous work. We then extend MIL with *types* and give preliminary results indicating that this extension learns better-quality strategies with suitable goal properties. We also show that the quality of the learned strategies can be further enhanced through the use of dependent learning.

## 1 Introduction

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

An observed phenomenon is that proofs often group into families, such that, once the expert user has discharged one proof, she can mentally extract a proof strategy which she uses to complete the rest [3]. The remaining proofs have to be manually guided as well, although the proof strategy is clear in her head.

If one could learn and reapply proof strategies from a few examples then this could significantly increase automation, making the overall approach more cost-effective – a key bottleneck for industrial application – and provide support for more elegant automated proofs that a user can understand.

* 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.

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

Previous work to learn proof strategies [12,6,8] has only attempted to extract general strategies from a large corpus of proofs, and has not addressed the desirable extraction of "local" strategies from small families. It has simplified the problem to composition of tactics, with no explanations of *why* or *when* a strategy should be applied. Such *explanation* is crucial so the user can understand the learnt strategy. It also reduces the search space and ensures termination without resorting to hard-coded heuristics which may rule out some proofs.

This deficiency was part of our motivation in developing the *PSGraph* language [9], which describes such "why"s and "when"s by including information about the tactics and sub-goals. This is achieved by representing proof strategies as graphs, where proof tactics are represented by boxes and goal information by predicates which label the wires, which we call *wire predicates*.

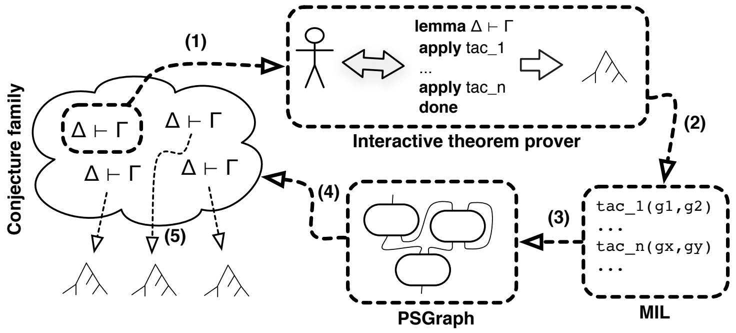


**Fig. 1.** Overview of our approach

Meta-Interpretive Learning (MIL) [19] supports predicate invention and the learning of recursive definitions of the kind represented by definitions of the wire predicates. Our vision is based around using MIL to extract proof strategies which currently reside in the ITP user's head. Our approach is illustrated in Fig. 1, where (1) a user selects one (or a few) conjectures from a "conjecture family" and guides the proof as normal (using a proof script) in an ITP system. This will generate a proof tree. (2) Metagol is then used to learn a proof strategy from the proof tree, (3) which can then be translated into PSGraph and (4) applied to the other conjectures in the family, which will generate new proof trees (5).

In this paper we take the first steps towards this. We show (C1) that MIL can learn proof strategies for the PSGraph language. As we are working with rich data, we show that these strategies have a high branching factor leading to 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 impractically large. We extend MIL with *types* (C2) and demonstrate (C3) that *"typed MIL learns more deterministic proof strategies than untyped MIL"*. We further show that introducing dependent learning reduces the time taken to learn a strategy (C4).

## 2 Related work

The development of proof-based methods – such as formal methods for software development, or mechanised mathematics – can be divided into 3 phases: (1) specification of the formal system; (2) conjecturing of the properties to be proven; (3) proving the conjectures. (1) includes discovery of invariants required for verification and (2) includes discovery of required intermediate lemmas. There may be mutual dependency between these phases: e.g. a failure from (3) may be used to highlight mistakes in the specification or a need for intermediate lemmas.

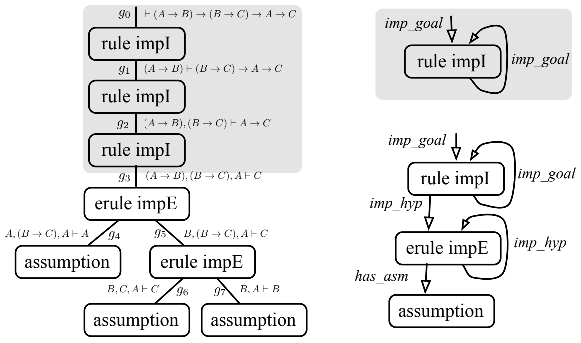
The main focus for (1) has been the discovery of intermediate assertions such as system invariants. This work typically involves simulating a specification and generating a set of traces. These become the examples used in machine learning. Daikon is one of the most well-known tools for invariant discovery and is based on inductive techniques [7]. In [11] Progol is used to discover invariants for a case study. They note the need for predicate invention, which is our main reason for using MIL. In [17] some domain-specific heuristics are developed for HR (a declarative machine learning tool) in order to discover invariants for a formal method called Event-B. [1] uses several several ILP systems to repair faulty specifications based on counter-examples and witnesses provided as examples generated by a model checker.

The main focus for (2) has been lemma discovery. Given a source lemma, [10] uses statistical machine learning to find analogous target lemmas and a "mutation" algorithm is developed to mutate the required intermediate lemmas from the target theory into the source theory. There are (at least) two recent projects building on this work (see [21]).

To prove a conjecture (3), we separate between *automated* and *interactive theorem provers* (ATPs and ITPs), where the former is fully automatic and the latter is more expressive but requires user guidance. Machine learning has been successful in improving automation of ATPs by selecting relevant hypotheses (see e.g. [13]). We aim to support a common methodology within ITP systems where a strategy is extracted from few examples and used to automate the proof of similar conjectures. This is orthogonal to work in ATP systems. [14] uses neural nets to provide hints for the user, but does not generalise proofs into strategies. A similar approach was used in [15] to classify proofs, again without learning strategies. Other approaches [12,6,8] have simplified the problem to only address tactic composition. The *Learn$\Omega$mega* system [12] implemented an algorithm which computes a least general generalisation of such traces captured as a regular expression. In her PhD thesis [6], Duncan used a combination of genetic algorithms and statistical methods to generate a similar regular expression for the Isabelle prover. Sepia [8] infers a state machine, which is searched over using breadth-first search for the Coq prover, allowing richer tactics. All these examples assume a large corpus of proofs to learn from and do not provide any guidance for the proof strategies, e.g. none address terminating loops. This is partially overcome by hard-coded heuristics [6] or search strategies with sub-optimal memory consumption [8]. In comparison we are addressing the more challenging problem of learning strategies which provide this guidance in terms of wire predicates. As these are arbitrary and unpredictable recursive functions on the term structure of a (higher-order) goal, predicate invention will be required. As far as we are aware, we are the first to apply ILP to such rich problems, with MIL the only machine learning technique supporting predicate invention.

## 3 Interactive Theorem Proving & PSGraph

```
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
```



$g_0$ $\vdash (A \to B) \to (B \to C) \to A \to C$
rule impI
$g_1$ $(A \to B) \vdash (B \to C) \to A \to C$
rule impI
$g_2$ $(A \to B), (B \to C) \vdash A \to C$
rule impI
$g_3$ $(A \to B), (B \to C), A \vdash C$
erule impE
$A, (B \to C), A \vdash A$ $g_4$
$g_5$ $B, (B \to C), A \vdash C$
assumption
erule impE
$B, C, A \vdash C$ $g_6$
$g_7$ $B, A \vdash B$
assumption
assumption
imp_goal
rule impI
imp_goal
imp_goal
rule impI
imp_goal
imp_hyp
erule impE
imp_hyp
has_asm
assumption

**Fig. 2.** Left to right: Isabelle proof script; proof tree; and strategy as PSGraph.

*Interactive theorem provers* (ITPs) enable users to interact with a proof system and guide a proof. They also support automation in terms of user-provided tactics. In this paper we will use the higher-order logic (Isabelle/HOL) embedding of the Isabelle theorem prover [20].

To develop theories and proofs, a user works with a *proof script*. The proof script shown on the left of Fig. 2 illustrates a proof of a conjecture in propositional logic. The first line states the conjecture to be proved, and is followed by a sequence of **apply** commands before the proof is closed by **done**. Note that each command is applied to the first sub-goal. The **apply** commands are used to apply so-called *proof tactics*, which are programs which reduce a goal to a list of (normally) smaller sub-goals. rule applies backwards resolution to a goal, erule applies an elimation rule to a hypothesis, while assumption proves the goal by applying one of the hypothesis to the goal[5]. Fig. 2 (middle) illustrates a proof tree of this script, where we have labelled each goal[6]. Note that the proof script shows a single branch of search space: each tactic may produce multiple branches and backtracking may be required.

The overall proof strategy is to first remove all $\longrightarrow$ in the conclusion (rule impI), then all $\longrightarrow$ in the hypothesis (erule impE). At the end, all sub-goals can be proven by the assumption tactic. The same strategy can be applied to prove $(A \longrightarrow B \longrightarrow C) \longrightarrow (A \longrightarrow B) \longrightarrow A \longrightarrow C$. However, the proof script will vary slightly as additional erule impE and assumption applications are required.

While there are richer tactics to prove these two conjectures automatically, they nevertheless illustrate a common phenomenon within ITP systems: by proving a single conjecture a user will develop a proof strategy that she can reapply

[5] Stricly speaking rule is the tactic and impI is its argument.
[6] This can be extracted from Isabelle using the *ProofProcess* framework [22].

across a family of similar conjectures (as seen in Fig. 1). Such families are common, either as separate conjectures or as sub-goals within a single conjecture. The strategies are normally in the head of a user, although an expert may manually encode them as tactics. – our goal is to help automate the extraction and application of such strategies.

A proof strategy needs to include procedural information about which tactics to apply. However, they should also contain declarative information about the type of goal and progress made to show how a goal should evolve. The lack of such information is evident in [12,6,8], and to support such strategies we developed the *PSGraph* language [9]. Here, a directed labelled graph is used to capture proof strategies: the boxes of the graph contain the tactics, while the wires are labelled by *wire predicates* – predicates to describe *why* a sub-goal should be on a given wire. A graph is evaluated as a flow graph, where a goal flows between tactics on a directed edge if the predicate on the edge holds for that particular goal. Fig. 2 (right) shows the proof strategy described above as a PSGraph. Here, *imp_goal* is a predicate that holds if the conclusion is an implication; *imp_hyp* is a predicate that holds if the hypothesis is an implication and the conclusion is not; while *has_assm* holds if the conclusion is present in the hypotheses. The remainder of the paper addresses learning of PSGraphs from a small set of examples. The shaded parts of Fig. 2 highlight a sub-proof (middle), which we can learn a sub-strategy from (right). We will return to this below.

## 4 Typed Meta-Interpretive Learning

Our framework is built on top of MIL [18,19], which is a form of ILP based on an adapted Prolog meta-interpreter. Whereas a standard Prolog meta-interpreter attempts to prove a goal by repeatedly fetching first-order clauses whose heads unify with a given goal, a MIL learner attempts to prove a set of goals by repeatedly fetching higher-order metarules (e.g. $P(X,Y) \leftarrow Q(Y,X)$) whose heads unify with a given goal. The resulting meta-substitutions are saved in an abduction store and can be reused in later proofs. Following the proof of a set of goals, a hypothesis is formed by projecting the meta-substitutions onto their corresponding metarules, allowing for a form of ILP which supports predicate invention and the learning of recursive theories.

To demonstrate this technique, suppose the background knowledge consists of the ground atom *parent(alice,bob)* and our goal is the ground atom *child(bob,alice)*. Let $P(X,Y) \leftarrow Q(Y,X)$ be a metarule. To prove this goal, a MIL learner fetches the metarule and applies the meta-substitution $\theta = \{P/child, Q/parent\}$ to unify the head of the goal with the metarule. The ground atom *inverse(child,parent)*, representing the meta-substitution $\theta$, is saved in an abduction store, and the learner continues the proof by attempting to recursively prove the body of the metarule. Once a proof is complete, the ground atom *inverse(child,parent)*, which is saved in the abduction store, is projected onto the corresponding metarule to obtain the clause $child(X,Y) \leftarrow parent(Y,X)$.

A novel aspect of MIL is its use of predicate invention for problem decomposition. In [16], the authors used MIL to induce string transformation programs. In this approach, solutions to simple problems, including their constituent predicates, are incorporated into the background knowledge and can be reused to learn solutions to more difficult problems. This *dependent learning* approach resulted in more compact programs through predicate reuse, and also led to reduced learning times.

```
prove([], Prog, Prog).
prove([Atom|As], Prog1, Prog2) :-
  metarule(Name, MetaSub, (Atom :- Body), Order),
  Order,
  abduce(metasub(Name, MetaSub), Prog1, Prog3),
  prove(Body, Prog3, Prog4),
  prove(As, Prog4, Prog2).
```

**Fig. 3.** Prolog code for generalised meta-interpreter

Our work uses the Metagol$_{DF}$ implementation [16] of MIL, displayed in Fig. 3. We extend this framework with simple types:

**Definition 1 (Typed Meta-Interpretive Learning).** Typed MIL *extends* MIL *by labelling each predicate $P$ with a constant $t$ representing its type. This is written $P : t$, such that $P(X,Y)$ becomes $P : t(X,Y)$. To reuse the algorithm of Fig. 3, the type is treated as an extra constant argument, thus $P : t(X,Y)$ is internally represented as $P(t,X,Y)$. For readability we use the former.*

## 5 Typed MIL for Proof Strategies

Learning PSGraphs from example proofs can be reduced to two mutually dependent learning problems: (1) learning a graph's structure; and (2) learning suitable wire predicates. Previous learning attempts [12,6,8] have simplified the problem to just (1), with the learned strategies lacking explanation resulting in, as our experiments show, a higher branching factor and thus slower search.

| **Name** | **Metarule** | **Order** |
|---|---|---|
| Lift | $P : psgraph(x,y) \leftarrow W : wpred(x), R : tactic(x,y)$ | $P \succ R$ |
| Chain | $P : psgraph(x,y) \leftarrow Q : psgraph(x,z), R : psgraph(z,y)$ | $P \succ Q, P \succ R$ |
| Loop | $P : psgraph(x,y) \leftarrow Q : psgraph(x,z), P : psgraph(z,y)$ | $P \succ Q, x \succ z \succ y$ |
| WChain | $P : wpred(x) \leftarrow Q : gdata(x,z), R : gdata(x,z)$ | $P \succ Q, P \succ R, Q \succ R$ |
| WBin | $P : wpred(x) \leftarrow Q : gdata(x,z)$ | $P \succ Q$ |

**Fig. 4.** Typed metarules for learning PSGraphs and wire predicates, with associated ordering constraints. $\succ$ is a pre-defined ordering over symbols in the signature. The letters $P$, $Q$, $R$ and $W$ denote existentially quantified higher-order variables; $x$, $y$, and $z$ denote universally quantified first-order variables.

These two problems have different features: (1) requires learning clauses which can be translated into a graph with wire predicates, while (2) needs a large search space in order to learn unknown predicates. Working with higher-order logic this includes arbitrary

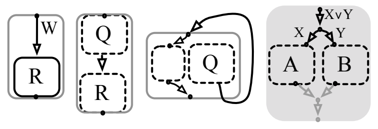


**Fig. 5.** Metarules in PSGraph

recursive functions. In typed MIL we can control the structure of what is learned with metarules, and use *types* to separate the learning problems.

Fig. 4 shows the metarules used, with a set of rules to learn proof strategies (with type *psgraph*) and rules to learn wire predicates (type *wpred*). For general problems the latter set may be larger. A graphical view is given in Fig. 5, with the outer grey lines indicate the learnt PSGraph in each case and stippled boxes indicating that the boxes are of type *psgraph*, meaning they may represent a sub-graph. The dots indicate where input and output wires are plugged.

*Lift* takes a single node in a proof tree with input $g_x$ and output $g_y$ and constructs a PSGraph consisting of a single node representing the corresponding tactic and a labelled input edge. Metagol finds the appropriate tactic clause in the background information, labelled with type *tactic*, and tries to find a clause of type *wpred* to define the wire predicate on the input edge. If a suitable *wpred* clause can be found in the background information it will be inserted, otherwise Metagol will use further metarules (such as *WChain* and *WBin* in Fig. 4) to attempt to find a suitable definition from the available information. Thus we find that the node in the proof tree has been "lifted" into the PSGraph:

$$PSGraph(psgraph, g_x, g_y) \leftarrow predicate(wpred, g_x), tactic(tactic, g_x, g_y).$$

*Chain* sequentially composes two such *psgraphs* to find a larger strategy, which are themselves found using the *Lift* rule. Starting at an edge represnting some goal $g_x$ on a proof tree and terminating at $g_y$ via some intermediate goal $g_z$, the resulting PSGraph would be:

$$\begin{aligned} PSGraph(psgraph, g_x, g_y) \leftarrow\ & subgraph\_1(psgraph, g_x, g_z), \\ & subgraph\_2(psgraph, g_z, g_y). \end{aligned}$$

*Loop* introduces iteration, which is represented in PSGraph as an additional output edge from a node looping back round to act as an additional input to that node. Strategies learned using *Loop* will always have two clauses: a base case in which the tactic is applied once and a recursive case where it is applied repeatedly:

$$\begin{aligned} PSGraph\_base(psgraph, g_x, g_y) \leftarrow\ & predicate(wpred, g_x), tactic(tactic, g_x, g_y). \\ PSGraph\_rec(psgraph, g_x, g_y) \leftarrow\ & PSGraph\_base(psgraph, g_x, g_z), \\ & PSGraph\_rec(psgraph, g_z, g_y). \end{aligned}$$

**Theorem 1.** *A PSGraph constructed using* Lift*,* Chain *and* Loop *will have a single typed input, no outputs and all wires will be labelled with appropriate predicates.*

*Proof.* The proof follows by rule induction over *Lift*, *Chain* and *Loop*. The base case *Lift* is trivial: there will be single input wire labelled with a learnt wire predicate to a node containing the tactic. By the induction hypothesis (IH), two components sequentially composed using *Chain* are guaranteed to have one input. To connect them a wire is added from the first to the second with the same input wire predicate as the second. The input of the composition is that of the first. For *Loop*, we know by the IH that the base case has one input with a wire predicate. The same predicate will label the feedback wire from the node's output, as illustrated in Fig. 5. When using a tail-recursive metarule, in order to ensure termination MIL must abduce a base case where the the order constraints associated with the metarules ensure convergence. For *Loop*, the learnt PSGraph is thus guaranteed to have a non-recursive component (see Fig. 5). If the output is "chained" to another component then the output from the non-recursive component is used. Finally, there may be multiple clauses describing a learnt PSGraph and without loss of generality we assume there are two clauses, $A$ and $B$, as in Fig. 5 (shaded). By the IH we know that both $A$ and $B$ have a single input with predicates $X$ and $Y$ respectively. Here, $A$ and $B$ are put side by side and an identity box (which does nothing) is added with one input. This has predicate $X \vee Y$. The outputs are then sent to $A$ and $B$ using their respective types. They will not have output wires. If this composed box is chained to another component $R$, which has input wire with predicate $Z$, then the output of all non-recursive components are combined to an idenity box which is plugged to $R$. Any wires introduced will have predicate $R$.

**Theorem 2.** *The translation of* Lift*,* Chain *and* Loop *into a PSGraph of Fig. 5 will generate an unique PSGraph.*

Note that this assumes that Metagol has successfully learned suitable wire predicates for each edge, either by predicate invention or by finding them in teh background information. We omit the proof of this but note that the language to express these in PSGraph is very close to Prolog so the translation is straightforward.

The metarules in Fig. 4 are used to learn proof strategies from an encoding of a proof tree. As a running example we will use the shaded sub-proof tree of Fig. 2 (middle) to illustrate this encoding, and how MIL learns the shaded PSGraph on the right of this figure.

We must define a sub-tree in the context of proof trees, which will be more restrictive than the standard definition, in order to support learning from them. This is because for a given tactic in a proof tree with more than one output, a sub-tree must capture every output. If one is left out then the generalisation may not be valid as not all (sub-)goals have been handled. A sub-tree is defined in terms of its *boundary* within a *proof tree*:

**Definition 2 (Proof tree).** *A proof tree is a tree with a dummy root node, each edge labelled by a unique named goal, and all other nodes tactic applications.*

**Definition 3 (Proof sub-tree boundary).** *Let $E$ be the set of edges in a proof tree $P$. A* boundary *of $P$ is a pair $(g, gs)$ where: $g \in E$ and $gs$ is a non-empty list where $\forall g' \in gs$, $g' \in E$; $\forall g' \in gs$ $g'$ is reachable from $g$; the path length from $g$ is equal for all $g' \in gs$, unless $g'$ is terminal (see below); if $g'$ is a goal in $gs$ such that no other goal in $gs$ has a longer path to $g$, then $\nexists g'' \notin gs$ such that $g''$*

*is after $g$ and has equal path length as $g'$ to $g$ or $g''$ is terminal with a shorter path length.*

To illustrate, $(g_0, [g_3])$ and $(g_2, [g_4, g_5])$ are valid subtree boundaries while $(g_2, [g_4, g_7])$ is not as $g_4$ and $g_7$ are not equidistant from $g_2$ (and there are other nodes which are).

From the *Lift* rule, we see that a tactic $R$ will have the form $R : \mathit{tactic}(X, Y)$. For example, the tactic rule impI will be encoded as $\mathit{rule\_impI} : \mathit{tactic}$. Our running proof sub-tree is encoded as:

$$\mathit{rule\_impI} : \mathit{tactic}(g_0, g_1). \quad \mathit{rule\_impI} : \mathit{tactic}(g_1, g_2). \quad \mathit{rule\_impI} : \mathit{tactic}(g_2, g_3).$$

If a tactic produces two sub-goals then two predicates are created, e.g. the step that turns $g_3$ into $g_4$ and $g_5$ is represented by the clauses:

$$\mathit{erule\_impE} : \mathit{tactic}(g_3, g_4). \qquad \mathit{erule\_impE} : \mathit{tactic}(g_3, g_5).$$

For tactics that do not produce any sub-goals (e.g. assumption), a dummy goal is created with no goal information in order to preserve the syntax. We call such goals *terminal*. All other goals contain a set of *hypotheses* and a *conclusion*, which we provide projections of. For example, $g_2$ is $A \longrightarrow B, B \longrightarrow C, A \vdash A \longrightarrow C$. It has three hypothesis: $A \longrightarrow B$, $B \longrightarrow C$ and $A$, and the conclusion: $A \longrightarrow C$. These *terms* are projected from the goals by *hyp:gdata* and *concl:gdata*. To illustrate use of these, the wire predicate used by the *assumption* tactic requires the same term to be in the hypothesis and conclusion: :

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

Isabelle internally stores terms as typed lambda expressions [20], using De Bruijn indices [5] to abstract over names of bound variables. The following clauses are used to encode terms:

$$b(I) \quad c(S) \quad v(S) \quad \mathit{app}(T, U) \quad \mathit{lambda}(V, T) \quad \mathit{exists}(T) \quad \mathit{forall}(Y)$$

Here, $b$ is the De Bruijn index used for a bound variable, $c$ is a constant, $v$ is a variable, *app* application, and *lambda* is a binder. While *exists* and *forall* can be expressed by *lambda* we have simplified them as they appear frequently.

We are not trying to learn new terms, and so have not translated their underlying types. We do not type these predicates for simplicity[7]. To illustrate the encoding, the goal information for $g_2$ becomes:

$$\begin{array}{l} \mathit{hyp} : \mathit{gdata}(g_2, \mathit{app}(\mathit{app}(c(\longrightarrow), c(a)), c(b))). \\ \mathit{hyp} : \mathit{gdata}(g_2, \mathit{app}(\mathit{app}(c(\longrightarrow), c(b)), c(c))). \\ \mathit{concl} : \mathit{gdata}(g_2, \mathit{app}(\mathit{app}(c(\longrightarrow), c(a)), c(c))). \end{array}$$

Note that a type *gdata* is used to represent goal data information. By treating Prolog, which is first-order, as a *meta-language* in this way we can fully encode

[7] Strictly speaking we should have written e.g. $b : \mathit{term}(I)$, $c : \mathit{term}(S)$ and so on.

the higher-order logic of Isabelle by abstracting away from some of its features. This can be applied to any formula provided by Isabelle and written in its declarative tactic language.

An advantage ILP techniques have over machine learning techniques used in e.g. [12,6,8], is that we can enrich the background clauses and use this to guide and simplify learning. For example, we provide definitions to extract the top level symbol in a conclusion or hypothesis[8]:

$$\begin{aligned}&\mathit{topsymbol} : \mathit{gdata}(G, X) \leftarrow \mathit{concl} : \mathit{gdata}(G, \mathit{app}(\mathit{app}(X, A), B)).\\ &\mathit{hypsymbol} : \mathit{gdata}(G, X) \leftarrow \mathit{hyp} : \mathit{gdata}(G, \mathit{app}(\mathit{app}(X, A), B)).\end{aligned}$$

For our running example we have that

$$\mathit{topsymbol} : \mathit{gdata}(g_0, \longrightarrow)\;\; \mathit{topsymbol} : \mathit{gdata}(g_1, \longrightarrow)\;\; \mathit{topsymbol} : \mathit{gdata}(g_2, \longrightarrow).$$

With these definitions we define a proof-tree encoding as:

**Definition 4 (Proof-tree encoding).** *In a* proof-tree encoding *each step of the proof tree is encoded as a relation of type* tactic*, with associated encoding of the goal information in terms of their hypotheses and conclusion.*

In order to learn the wire predicates, properties of the terms have to be learned. To support this, we introduce a set of *atomic term operators*:

**Definition 5 (Atomic term operator).** *The following clauses are* atomic term operators*:* $\mathit{const} : \mathit{gdata}(c(X))$*,* $\mathit{var} : \mathit{gdata}(v(X))$*,* $\mathit{bound} : \mathit{gdata}(b(X))$*,* $\mathit{left} : \mathit{gdata}(\mathit{app}(X, Y), X)$*,* $\mathit{right} : \mathit{gdata}(\mathit{app}(X, Y), Y)$*,* $\mathit{into} : \mathit{gdata}(\mathit{forall}(X), X)$*,* $\mathit{into} : \mathit{gdata}(\mathit{exists}(X), X)$ *and* $\mathit{into} : \mathit{gdata}(\mathit{lambda}(V, X), X)$*.*

For example, the $X$ in $\mathit{app}(\mathit{app}(X, A), B)$ can be projected by two consecutive $\mathit{left} : \mathit{gdata}$ applications. We can now define our learning problem:

**Definition 6 (Typed MIL of PSGraph).** *In* typed MIL of PSGraph *a binary relation of type* psgraph *is learned where the background information at least contains encodings of one or more proof trees together with atomic term operators, and the given examples are one or more subtree boundaries of the encoded proof trees.*

When using sub-trees and not the full trees we can learn sub-strategies and, as discussed later, we can apply dependent learning to learn increaisly larger sub-strategies.

Returning to our running example of Fig. 2 (where our goal is to learn a strategy *rimp:psgraph* describing the shaded tree), Metagol will use a metarule of the *psgraph* type as this is the type given for *rimp*. In our case, *Loop* will trigger learning of the "loop body" $Q : \mathit{psgraph}(x, z)$. Here, *Lift* is applied using the background information $\mathit{rule\_impI} : \mathit{tactic}(g_0, g_1)$ to generate $\mathit{rule\_impI} : \mathit{tactic}(A, B)$. Metagol must also find a wire predicate, of type *wpred* for the input. Since no *wpred* clauses are given in the background information, Metagol

[8] With correct metarules, Metagol may be able to find better definitions.

must invent one. *WBin* instantiates $B$ in $topsymbol : gdata(A, B)$ to $c(imp)$, which is the top symbol of the conclusion of $g_0$. This invented predicate is called $imp\_goal : wpred$[9] and the invented graph component is called *simpI:psgraph*. In order to reach $g_3$, and end the loop (base case), *Lift* is again applied to find a clause with body identical to *simpI:psgraph*. The learnt program then becomes:

$$\begin{array}{lll}
rimpI : psgraph(A,B) & \leftarrow simpI : psgraph(A,C), rimpI : psgraph(C,B). & (Loop) \\
rimpI : psgraph(A,B) & \leftarrow imp\_goal : wpred(A), rule\_impI : tactic(A,B). & (Lift) \\
simpI : psgraph(A,B) & \leftarrow imp\_goal : wpred(A), rule\_impI : tactic(A,B). & (Lift) \\
imp\_goal : wpred(A) & \leftarrow topsymbol : gdata(A, c(imp)). & (WBin)
\end{array}$$

A PSGraph encoding of this, following our described translation, is given in Fig 6. Our metarules have taken a more "functional view" of iteration, meaning the solution deviates from the example strategy of Fig 2 as there is a separate base and step case, which are identical here. We discuss an alternative future approach in §7, which would discover a strategy closer to the one in Fig 2.

When considered as a sequence of tactics, we note that proof strategies do not always terminate in ITP. If the *Loop* metarule is naively applied, an infinite sequence of tactics may be introduced. We address this by evaluating the subgoals generated by each iteration of the tactic(s) on the recursive node, and define termination in terms of an ordering $\succ$ over the goals. Note that we cannot derive a general ordering for all possible conjectures. With the exception of proof by contradiction, we can use the total number of symbols for propositional and predicate logic addressed in this paper. Let *symbcount* be the total number of symbols (e.g. $\forall$ or $\rightarrow$) for a goal (including conclusion hypotheses). We define our termination measure as

$$G_1 \succ G_2 \leftarrow symbcount(G_1) > symbcount(G_2)$$

**Theorem 3.** *If all given tactics terminate, then a learnt PSGraph will either fail or terminate for any input.*

*Proof.* The proof follows by rule induction over the metarules that generate a PSGraph. A lifted tactic will either fail or generate output. If it does not have an outgoing edge or the wire predicates do not fit then it will fail. If not, it will succeed with the goals on the output wire. For chained components we can assume that each component terminates (or fails) from the IH. The chained component will therefore trivially terminate (or fail). For *Loop*, the nested component is guaranteed to terminate/fail from the IH. By $\succ$ each iteration is guaranteed to move towards a lower bound, thus the loop will terminate.

*Tool support* Fig. 6 shows our tool architecture. All components, except the stippled line, have been implemented. The shaded areas are external parts and not contributions of this paper. The tool process is as follows: *Isabelle* proofs are captured by the *ProofProcess* tool [22]. This produces a proof tree in XML form,

[9] We have renamed the invented names for readability.

which our *Generator* parses and translates into a Prolog file as described above. The generator is implemented in Standard ML on top of Isabelle, supported by Isabelle libraries. *Metagol* is applied to this file and will, if successful, produce a proof strategy represented as a *psgraph* typed predicate. This can then be translated to PSGraph, also implemented using Standard ML on top of Isabelle, and used to automate other proofs. Implementation of this is future work.

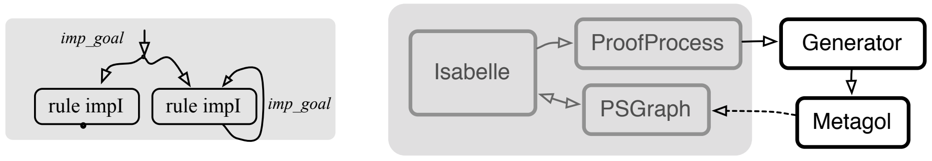


**Fig. 6.** Learnt PSGraph for running example (left) and tool architecture (right)

In the experiments discussed next translation is handled manually. The example proofs are converted into Prolog, and the learnt strategy is evaluated to see if these proofs could be expressed using it. This does not require translation into a PSGraph.

## 6 Experiments

We have experimented with untyped and typed MIL to learn proof strategies from a collection of 15 proofs in propositional logic[10]. In each example we provide tactic definitions, goal information and the metarule set given in fig. 4 as background information. The experiments were run using YAP on Ubuntu using a 3.10 GHz Intel i5-2400 CPU with 4GB RAM[11].

Our first experiment considered determinism of typed MIL in comparison to untyped MIL. For each example we consider the branching factor ($\sigma$) of the learned strategy, indicating the number of possible proof trees (including partial trees and failures) which could be constructed by applying the strategy to a goal. Branch points are found by manual inspection of the learned strategies. These occur when a goal could follow more than one edge in the graph, either through over-general wire predictes or edges with the same label. Automated extraction is not currently supported by PSGraph and is future work.

In these first experiments we provided an explicitly-defined wire predicate clause for each goal in the background information, with the type label omitted in the untyped experiments. The experiments were repeated with different time limits (1, 2, 4 and 8 seconds).

The graph in Fig. 7 (upper left) shows the average $\sigma$ for both untyped and typed strategies compared to an optimum value. This optimum represents a learned strategy from each example with one branch and thus one proof tree can be formed for each. Using untyped MIL $\sigma > 1$ initially, indicating more than one path on average, and $\sigma$ increases with time as larger solutions are found. With typed MIL $\sigma = 1$ initially, remaining constant over time. The results show that

[10] The examples are taken from: isabelle.in.tum.de/exercises
[11] Code for all experiments available at: https://sites.google.com/site/cifarquhar/

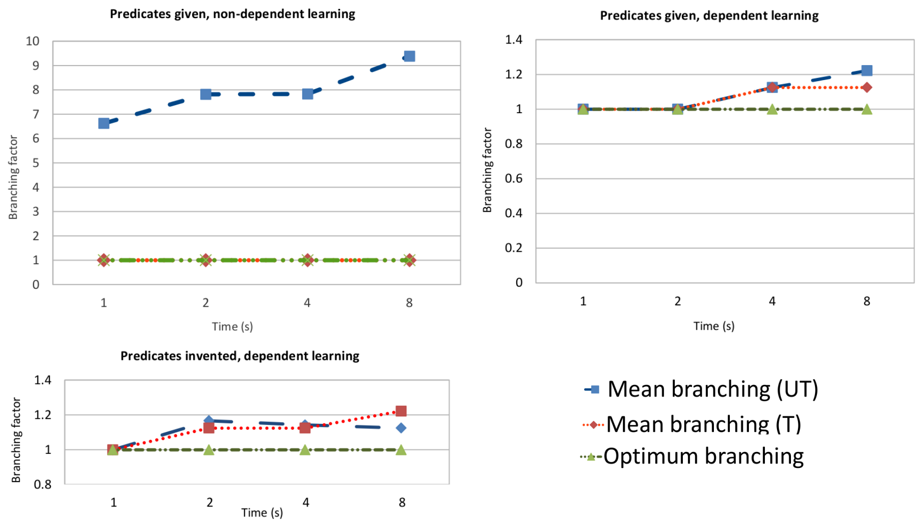


**Fig. 7.** Mean branching factor $\sigma$ for untyped (UT) and typed (T) MIL.

as time increases untyped MIL learns less efficient strategies. Conversely, typed MIL produces strategies which are optimally efficient.

These differences are due to how Metagol constructs its' solutions. Consider a strategy consisting of repeated applications of a single tactic. Typed MIL forms this using the *Loop* rule, where both base and step cases must include *psgraph* clauses. These are formed using lifted *tactic* clauses with a *wpred* clause. In untyped MIL there is no requirement to use *psgraph* clauses, and so a solution is found using *tactic* clauses with no wire predicates. Consequently when a goal passes through the node there is no wire predicate to direct it either towards the next tactic or around the loop, and so both must be tried. There is a similar issue whenever there are multiple outputs from a node, and it is this lack of pre-conditions which results in the higher $\sigma$.

At first neither untyped nor typed MIL was able to learn from every example within the given time limit. This was due to the size of the solution required; the time needed grows exponentially with the number of clauses in the solution. In the untyped case this is less pronounced as the solutions are less detailed. In typed MIL a solution contains one clause for every lifted tactic plus a number of clauses describing how they link together, resulting in larger definitions for the same strategies. We address this by using dependent learning to learn smaller sub-strategies as described in §4.

By using dependent learning we reduce the time taken to learn more complex strategies, however this has the trade-off of taking longer to find simpler strategies. We have still not achieved 100% success within the given time frame, nor are the additional strategies learned fully deterministic. However, there is a significantly smaller branching factor in the untyped case (Fig. 7, upper right). This is again due to Metagol's construction of solutions: Metagol will always use the "simplest" (usually smallest) solution. When using dependent learning to learn sub-strategies describing sub-trees this means that the best metarule to use will generally be *Chain*, as each sub-strategy can then be extended one node at a time. Consequently there are fewer potential branch points for untyped MIL, while typed MIL is largely unaffected.

We now address the problem of inventing pre-conditions. In previous experiments we have provided explicit pre-conditions in the background information, now we look for Metagol to invent definitions based on the goal data provided. Results of these tests are shown in Fig. 7 (lower right).

The metarules restrict possible solutions, reflected in the weakening of the strategies seen here. Definitions are limited to using *top_symbol*, *hyp_symbol*, *hyp* and *concl*, which rules out other predicates being found. Since Metagol is forced to find a *wpred* clause in the typed case, and a simple monadic predicate in the untyped case, it must produce a definition which is too generalised and which would permit goals to follow an incorrect edge in the corresponding graph. Future work will include experiments using a minimal set of metarules from which others can be inferred, allowing Metagol to find better definitions for pre-conditions, including using the $left(\_,\_)/right(\_,\_)$ notation introduced in Definition 5.

We note from Fig. 7 that $\sigma$ is now reduced for untyped MIL, and has increased for typed MIL. We also observe in Metagol's output limited reuse of invented predicates, mainly those defining pre-conditions. Although limited in scale, this illustrates how previously learned (sub-)strategy definitions can be reused in later examples. Future work will include scaling this up with larger examples and working with larger sub-strategies.

As a final experiment we attempt to learn a strategy in predicate logic, using the example $\forall A\ B.\ A \wedge B \longrightarrow B \wedge A$. Using the notation given in Definition 5, we define the initial goals as:

$$\begin{gathered}
concl(gdata, a_0, forall(forall(app(app(c(imp), app(app(c(conj), b(1)), b(0))),\\
app(app(c(conj), b(0)), b(1))))))).\\
concl(gdata, a_1, forall(app(app(c(imp), app(app(c(conj), v(a)), b(0))),\\
app(app(c(conj), b(0)), v(a))))).
\end{gathered}$$

As quantifiers are removed from the goal as the proof is evaluated (firstly with the *rule_allI* tactic) we replace the corresponding bound variables. This experiment learned a strategy similar to the running example shown in §4, including invented predicates.

## 7 Conclusion and further work

Our first experiment was able to learn proof strategies from 86% of the examples and thus supports (C1) - that MIL is capable of learning proof strategies in our framework. However, in terms of branching, untyped MIL seems to offer no improvements over previous work [6,12] as it does not learn wire predicates to provide explanation for a strategy. We have introduced types in the MIL framework by adding an additional constant argument to the predicate (C2). The results show that typed MIL learns wire predicates and reduces branching, although we were able to learn strategies from fewer examples. Our assertion (C3) that typed MIL reduces non-determinism is distinct from success rate, however, and so is validated. The introduction of types means a larger number of clauses to represent strategies, which increases the run time for Metagol and is the reason for failure in most cases. This is addressed through the use of dependent learning, and the slight increase in successful learning validates (C4).

We will investigate a way of improving our learned strategies by using a combinator-based approach to learning. This will involve the development of more complex metarules in order to capture branching within a single clause representing one node, rather than the multiple clauses currently required. We will experiment with making our definitions functional and introducing combinators (such as *OR* and *LOOP*) to handle multiple outputs. As discussed in §5, we aim to learn strategies closer to the one in fig. 2.

We will move on from examples focused on propositional logic and begin to look at more complex proofs. As shown in §5, we have already begun implementing a representation for higher-order logic, allowing us to look at predicate logic. We will also look at examples from group theory, including those used by [12], some geometry and we will attempt to learn the *rippling* proof strategy [2], which will require inventing very complex wire predicates.

As one of the key motivations for this work is the reasoning behind applying a given strategy, we will also investigate learning wire predicates from existing complex tactics with the aim of improving their efficiency, eg. Isabelle's *auto* tactic. This will be one instance which requires learning from multiple positive examples, which we have had some success with already. We will make further use of dependent learning to enable us to reuse parts of strategies which can be applied to multiple proofs in order to help us with this.

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 [4] with argument types. A current student project is investigating the use of MIL to learn safety case patterns. Longer term, we plan to study 'type invention' and support for 'higher order types'.

## References

1. D. Alrajeh, J. Kramer, A. Russo, and S. Uchitel. Elaborating requirements using model checking and inductive learning. *Software Engineering, IEEE Transactions on*, 39(3):361–383, 2013.
2. A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning.* Cambridge University Press, 2005.
3. 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.
4. A. Cropper and S. Muggleton. Learning efficient logical robot strategies involving composable objects. In *IJCAI*, 2015. To appear.
5. N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae*, volume 75, pages 381–392. Elsevier, 1972.
6. H. Duncan. *The use of Data-Mining for the Automatic Formation of Tactics.* PhD thesis, University of Edinburgh, 2002.
7. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

8. T. Gransden, N. Walkinshaw, and R. Raman. Sepia: Search for proofs using inferred automata. In A. P. Felty and A. Middeldorp, editors, *CADE-25*, volume 9195 of *LNCS*, pages 246–255. Springer, 2015.
9. G. Grov, A. Kissinger, and Y. Lin. A graphical language for proof strategies. In *LPAR*, volume 8312 of *LNCS*, pages 324–339. Springer, 2013.
10. J. Heras, E. Komendantskaya, M. Johansson, and E. Maclean. Proof-pattern recognition and lemma discovery in acl2. In *LPAR*, pages 389–406. Springer, 2013.
11. D. T. Ho, M. Zhang, and K. Ogata. A case study on extracting the characteristics of the reachable states of a state machine formalizing a communication protocol with inductive logic programing. In PreProceedings of ILP 2015.
12. 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.
13. C. Kaliszyk and J. Urban. Learning-assisted automated reasoning with flyspeck. *JAR*, 53(2):173–213, 2014.
14. E. Komendantskaya, J. Heras, and G. Grov. Machine learning in proof general: Interfacing interfaces. In *UITP 2012*, pages 15–41, 2013.
15. E. Komendantskaya. Machine learning coalgebraic proofs. *Short Post-proceedings of ILP*, 11, 2011.
16. D. Lin, E. Dechter, K. Ellis, J. Tenenbaum, and S. Muggleton. Bias reformulation for one-shot function induction. In *Proceedings of the 23rd European Conference on Artificial Intelligence (ECAI 2014)*, pages 525–530, Amsterdam, 2014. IOS Press.
17. M. T. Llano, A. Ireland, and A. Pease. Discovery of invariants through automated theory formation. *Formal Aspects of Computing*, 26(2):203–249, 2014.
18. S. Muggleton, D. Lin, N. Pahlavi, and A. Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Machine Learning*, 94:25–49, 2014.
19. 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.
20. L. C. Paulson. The foundation of a generic theorem prover. *JAR*, 5(3):363–397, 1989.
21. A. Velykis, G. Grov, and L. Freitas. Contributions to AI4FM 2015. Available from `http://www.ai4fm.org/papers/ai4fm-2015-proceedings.pdf`.
22. A. Velykis. *Capturing Proof Process*. PhD thesis, Newcastle University, 2015.