FDS Seminar

Interactive Theorem Prover

Interactive Theorem Prover

Interactive Theorem Prover (ITP) becomes a more common way of establishing the correctness of software and mathematical proofs :

- Compilers
- Operating Systems
- Kepler Conjecture
- Feit-Thompson Theorem

1. Kepler Conjecture : stack of sphres in the plan has a density of 74%.
2. Feit-Thompson Theorem : every finite group of odd order is solvable.

Ability to learn and re-apply proof strategies would significantly increase the productivity and effectiveness.

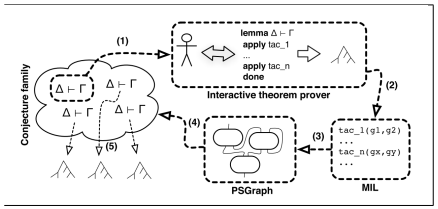
```
lemma K: A -> B -> A
proof
  assume A
  show B -> A
  proof
    show A by fact
  qed
qed
```

This is a small example of a proof for the K combinator with Isabelle. There are multiple proof and sub-proof which could be reuse.

Previous work to learn proof strategies has only attempted to extract general strategies from a large corpus of proofs.

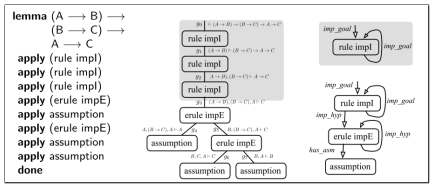
They are not addressing the desirable extraction of local strategies. Which was the motivation for developing the PSGraph language.

Previous work to learn proof strategies has only attempted to extract general strategies from a large corpus of proofs.



Overview of the approach.

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 used to learn a proof strategy from the proof tree.
3. The proof tree is translated to PSGraph.
4. We apply the proof strategy to the other conjectures of the familly.
5. Which will generate new proof trees.



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

1. To develop theories and proofs, a user works with a proof script. The proof shown 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.
2. For this specific proof, the idea is to remove all the implication in the conclusion by the deductiom lemma. Then remove all implication in the hypothesis by applying the assumption. At the end, everything can be proven by the assumption tactics.
3. The apply (tactics) take a lemma and a proof methods.

By proving a single conjecture a user will develop a proof strategy that he can reapply across a family of similar conjectures.

The strategies are normally in the head of a user, although an expert may manually encode them as tactics, the goal is to help automate the extraction and application of such strategies.

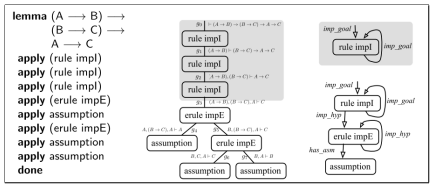
1. Such families are common, either as separate conjectures or as sub-goals within a single conjecture.

1. That's why PSGraph has been developed. The other system are lacking the last two points.
2. Wire predicates : predicates to describe why a sub-goal should be on a given wire.

A proof strategy needs to include procedural information about which tactics to apply, the type of goal and progress made to show how a goal should evolve.

A directed labelled graph is used to capture proof strategies : the boxes of the graph contain the tactics, wires are labelled by wire predicates.

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.



Left to right: Isabelle proof script; proof tree; and strategy as 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.

The framework is built on top of MIL by using Metagol.
Metagol[1] is an inductive logic programming (ILP) system based on meta-interpretive learning.

1. 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.
2. The basic idea is to process the unification on the predicate and not on the argumments.

Metagol : output

```
% clauses: 1
% clauses: 2
% clauses: 3
grandparent(A,B):-grandparent_1(A,C),grandparent_1(C,B).
grandparent_1(A,B):-mother(A,B).
grandparent_1(A,B):-father(A,B).
```

1. The predicate grandparent/2 is invented and corresponds to the parent relation.

Metarules

Metagol requires higher-order metarules to define the form of clauses permitted in a hypothesis.

```
metarule([P,Q,R],([P,A,B]:-[[Q,A,C],[R,C,B]])).
```

Users need to supply Metarules. Work is in progress[1] for automatically identifying the necessary metarules.

[1] at the time of this paper

1. The list of symbols in the first argument denote the existentially quantified variables which Metagol will attempt to find substitutions for during the learning.

Learning PSGraphs from example proofs can be reduced to two mutually de- pendent learning problems :

1. learning a graph's structure.
2. learning suitable wire predicates.

Previous learning attempts have simplified the problem to just the first point.

1. with the learned strategies lacking explanation resulting in, as our experiments show, a higher branching factor and thus slower search.

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

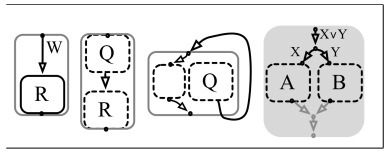
1. Lift : Metagol finds the appropriate tactic clause in the background information, labelled with type tactic, and tries to find a clause of type wpred to define the wire predicate on the input edge. If a suitable *wpred* clause can be found in the background information it will be inserted, otherwise 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.

$$PSGraph(psgraph, g_x, g_y) \leftarrow subgraph_1(psgraph, g_x, g_z), subgraph_2(psgraph, g_z, g_y).$$

1. Chain : 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 gx on a proof tree and terminating at gy via some intermediate goal gz, the resulting PSGraph would be.

$$PSGraph\_base(psgraph, gx, gy) \leftarrow predicate(wpred, gx), tactic(tactic, gx, gy).$$
$$PSGraph\_rec(psgraph, gx, gy) \leftarrow PSGraph\_base(psgraph, gx, gz), PSGraph\_rec(psgraph, gz, gy).$$

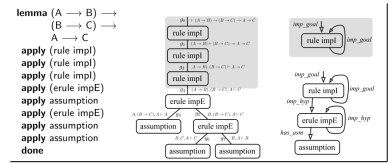
1. Loop : Loop introduces iteration, which is represented in PSGraph as an additional output edge from a node looping back round to act as an additional input to that node. Strategies learned using Loop will always have two clauses: a base case in which the tactic is applied once and a recursive case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



Metarules in PSGraph.

1. 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.
2. 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.

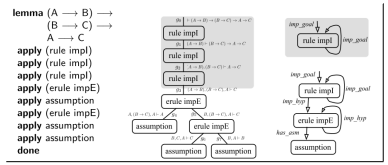
Proof tree encoding



Previous example.

$rule\_impl : tactic(g0, g1).rule\_impl : tactic(g1, g2).rule\_impl : tactic(g2, g3).$

1. To illustrate, (g0, [g3]) and (g2, [g4, g5]) are valid subtree boundaries while (g2, [g4, g7]) is not as g4 and g7 are not equidistant from g2 (and there are other nodes which are).
2. From the Lift rule, we see that a tactic R will have the form R : tactic(X, Y). For example, the tactic rule impl will be encoded as rule impl : tactic. Our running proof sub-tree is encoded as : formula



Previous example.

erule_impE : *tactic*($g3, g4$).*erule_impE* : *tactic*($g3, g5$).

1. If a tactic produces two sub-goals then two predicates are created, e.g. the step that turns g3 into g4 and g5 is represented by the clauses.
2. 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.

g_2 is $A \to B, B \to C \vdash A \to C$

These terms are projected from the goals by $hyp : gdata$ and $concl : gdata$.

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

1. All other goals contain a set of hypotheses and a conclusion, which we provide projections of.

1. ML techniques used in related work.

Advantages over machine learning techniques

An advantage that ILP techniques have over machine learning techniques is that we can enrich the background clauses and use this to guide and simplify learning.

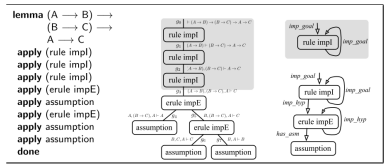
For example, definitions to extract the top level symbol in a conclusion or hypothesis are provided :

$$topsymbol : gdata(G, X) \leftarrow concl : gdata(G, app(app(X, A), B)).$$
$$hypsymbol : gdata(G, X) \leftarrow hyp : gdata(G, app(app(X, A), B)).$$

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 increasily larger sub-strategies.

1. 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 increasily larger sub-strategies.

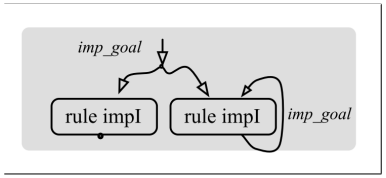


Previous example.

1. Returning to the previous example, the shaded part. Metagol will use a metarule of the *psgraph* type as this is the type given for *rule_imp*.
2. In our case, Loop will trigger learning of the "loop body" $Q : psgraph(x, z)$. Here, Lift is applied using the background information $ruleimpl : tactic(g0, g1)$ to generate rule $impl : tactic(A, B)$. Metagol must also find a wire predicate, of type wpred for the input.
3. Since no wpred clauses are given in the background information, Metagol must invent one. *WBin* instantiates B in topsymbol : $gdata(A, B)$ to $c(imp)$, which is the top symbol of the conclusion of $g0$. This invented predicate is called $impgoal : wpred$ and the invented graph component is called simpl: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 $simpl : psgraph$.

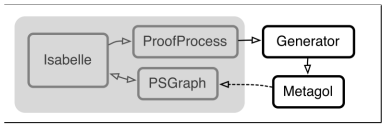
$rimpl : psgraph(A, B) \leftarrow simpl : psgraph(A, C), rimpl : psgraph(C, B).$
$rimpl : psgraph(A, B) \leftarrow impgoal : wpred(A), ruleimpl : tactic(A, B).$
$simpl : psgraph(A, B) \leftarrow impgoal : wpred(A), ruleimpl : tactic(A, B).$
$impgoal : wpred(A) \leftarrow topsymbol : gdata(A, c(imp)).$

1. The learnt program then becomes.



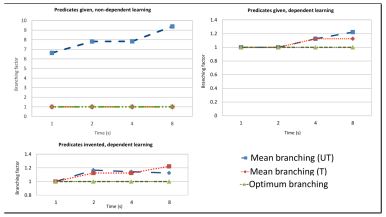
PSGraph encoding for the example.

1. 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.
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 sub- goals generated by each iteration of the tactic(s) on the recursive node, and define termination in terms of an ordering $>$ 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. or $\rightarrow$) for a goal : $G1 > -G2 \leftarrow \mathit{symbcount}(G1) > \mathit{symbcount}(G2)$.



Tool architecture.

1. 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, 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.



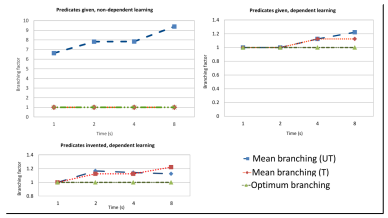
Mean branching factor ρ for untyped (UT) and typed (T) MIL. Experiments are running on 15 proofs in CL.

1. Our first experiment considered determinism of typed MIL in comparison to untyped MIL. For each example we consider the branching factor (ρ) 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).

These differences are due to how Metagol constructs its' solutions.
Consider a strategy consisting of repeated applications of a single tactic.

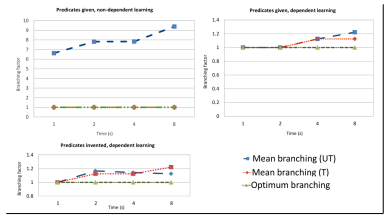
- Typed MIL forms this using the Loop rule.
- With untyped MIL there is no requirement to use psgraph clauses.

1. where both base and step cases must include psgraph clauses. These are formed using lifted tactic clauses with a wpred clause.
2. In untyped MIL there is no requirement to use psgraph clauses, and so a solution is found using tactic clauses with no wire predicates.
3. Consequently when a goal passes thTrough 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 ρ.



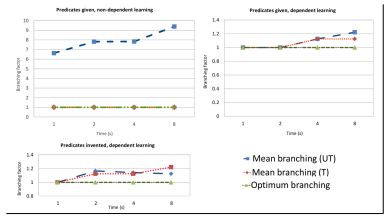
Mean branching factor ρ for untyped (UT) and typed (T) MIL.

1. 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



(Upper right) Smaller branching factor for UT.

1. By using dependent learning we reduce the time taken to learn more com- plex 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.



(Upper right) Smaller branching factor for UT.

1. 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.

- Show that MIL can learn proof strategies for the PSGraph language.
- Extend MIL with types.
- Demonstrate that "typed MIL learns more deterministic proof strategies than untyped MIL".
- Show that introducing dependent learning reduces the time taken to learn a strategy.

1. Able to learn proof strategies from 86% of the examples and thus supports that MIL is capable of learning proof strategies in our framework.
2. We have introduced types in the MIL framework by adding an additional constant argument to the predicate (C2).
3. 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.
4. 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).

Focus on HOL Light for two reasons :

- follows the LCF approach.
- implements higher-order logic as its foundation.

1. This means that complicated inferences are reduced to the most primitive ones and the data extrac- tion related modifications can be restricted the primitive inferences and it is relatively easy to extract proof steps at an arbitrary selected level of granularity.
2. Second, HOL Light implements higher-order logic (Church, 1940) as its foundation, which on the one hand is powerful enough to encode most of today's formal proofs, and on the other hand allows for an easy integration of many powerful automation mechanisms (Baader Nipkow, 1998; Paulson, 1999).

Training and testing examples are grouped by proof, for each proof we have

- the conjecture.
- the dependencies of the theorem.
- list of used and not used intermediate statements.

1. This means that the conjectures used in the training and testing sets are normally disjoint.
2. In the dataset, for every proof we provide the same number of useful and non-useful steps. As such, the proof step classification problem is a balanced two-class classification problem, where a random baseline would yield an accuracy of 0.5.

For each statements (conjecture, proof dependency, or intermediate statement) :

- human-like printout (with parenthesis).
- predefined tokenization.

1. The tokenization that we propose attempts to reduce the number of parentheses. To do this we compute the maximum number of arguments that each symbol needs to be applied to, and only mark partial application. This means that fully applied functions (more than 90% of the applications) do not require neither application operators nor parentheses. Top-level universal quantifications are eliminated, bound variables are represented by their de Bruijn indices (the distance from the corre- sponding abstraction in the parse tree of the term) and free variables are renamed canonically. Since the Hindley-Milner type inference Hindley (1969) mechanisms will be sufficient to reconstruct the most-general types of the expressions well enough for automated-reasoning techniques Kaliszyk et al. (2015) we erase all type information.

Is a statement useful ?

Focus on the first task, which can be specialized into two different tasks :

- Unconditioned classification of proof steps.
- Conditioned classification of proof steps.

1. Unconditioned classification of proof steps: determining how likely a given proof is to be useful for the proof it occurred in, based solely on the content of statement (i.e. by only providing the model with the step statement itself, absent any context).
2. Conditioned classification of proof steps: determining how likely a given proof is to be useful for the proof it occurred in, with "conditioning" on the conjecture statement that the proof was aiming to attain, i.e. by providing the model with both the step statement and the conjecture statement).

Tasks that require most human time :

- the search for good intermediate steps.
- the search for automation techniques able to justify the individual steps.
- searching theorem proving libraries for the necessary simpler facts.

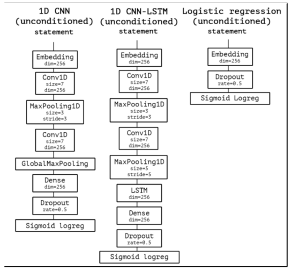
Corresponds to the machine learning tasks proposed previously.

1. Being able to predict the usefulness of a statement will significantly improve many automation techniques. The generation of good intermediate lemmas or intermediate steps can improve level of granularity of the proof steps. Understanding the correspondence between statements and their names can allow users to search for statements in the libraries more efficiently (Aspinall and Kaliszyk, 2016). Premise selection and filtering are already used in many theorem proving systems, and generation of succeeding steps corresponds to conjecturing and theory exploration.

For each tasks (conditioned and unconditioned classification) : three different deep learning architectures. Models are implemented in

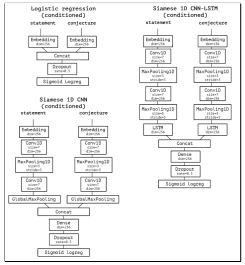
Tensorflow using the Keras framework.

1. cover a range of architecture features (from convolutional networks to recurrent networks), aiming at probing what characteristics of the data are the most helpful for usefulness classification



Unconditioned classification model architectures.

1. Logistic regression on top of learned token embeddings. This minimal model aims to determine to which extent simple differences between token distribution between useful and non-useful statements can be used to distinguish them. It provides an absolute floor on the performance achievable on this task.
2. 2-layer 1D convolutional neural network (CNN) with global maxpooling for sequence re- duction. This model aims to determine the importance of local patterns of tokens.
3. 2-layer 1D CNN with LSTM sequence reduction. This model aims to determine the importance of order in the features sequences.



Conditioned classification model architectures.

1. For this task, we use versions of the above models that have two siamese branches (identical branches with shared weights), with one branch processing the proof step statement being considered, and the other branch processing the conjecture. Each branch outputs an embedding; these two embeddings (step embedding and conjecture embedding) are then concatenated and the classified by a fully-connected network. See figure 2 for a layer-by-layer description of these models.

Two types of encoding :

- Character-level encoding of the human-readable versions of the statements.
- Token-level encoding of the versions of the statements.

1. It should be noted that all of our models start with an Embedding layer, mapping tokens or characters in the statements to dense vectors in a low-dimensional space. We consider two possible encodings for presenting the input statements (proof steps and conjectures) to the Embedding layers of our models:
2. Character-level encoding of the human-readable versions of the statements, where each character (out of a set of 86 unique characters) in the pretty-printed statements is mapped to a 256-dimensional dense vector. This encoding yields longer statements (training state- ments are 308 character long on average).
3. Token-level encoding of the versions of the statements rendered with our proposed high- level tokenization scheme. This encoding yields shorter statements (training statements are 60 token long on average), while considerably increasing the size of set of unique tokens (1993 total tokens in the training set).

Table 2: HolStep proof step classification accuracy without conditioning

	Logistic regression	1D CNN	1D CNN-LSTM
Accuracy with char input	0.71	0.82	**0.83**
Accuracy with token input	0.71	**0.83**	0.77

Table 3: HolStep proof step classification accuracy with conditioning

	Logistic regression	Siamese 1D CNN	Siamese 1D CNN-LSTM
Accuracy with char input	0.71	0.81	**0.83**
Accuracy with token input	0.71	0.82	0.77

1. unconditioned 1D CNN model yields an accuracy of 82acter encoding and token encoding (tables 2 and 3). This demonstrates that patterns of characters or patterns of tokens are considerably more informative than single tokens for the purpose of usefulness classification
2. Finally, our unconditioned convolutional-recurrent model does not improve upon the results of the 1D CNN, which indicates that our models are not able to meaningfully leverage order in the feature sequences into which the statements are encoded.

Table 2: HolStep proof step classification accuracy without conditioning

	Logistic regression	1D CNN	1D CNN-LSTM
Accuracy with char input	0.71	0.82	**0.83**
Accuracy with token input	0.71	**0.83**	0.77

Table 3: HolStep proof step classification accuracy with conditioning

	Logistic regression	Siamese 1D CNN	Siamese 1D CNN-LSTM
Accuracy with char input	0.71	0.81	**0.83**
Accuracy with token input	0.71	0.82	0.77

1. For the logistic regression model and the 2-layer 1D CNN model, the choice of input encoding seems to have little impact. For the convolutional-recurrent model, the use of the high-level tokenization seems to cause a large decrease in model performance (figs. 4 and 6). This may be due to the fact that token encoding yields shorter sequences, making the use of a LSTM less relevant.

Table 2: HolStep proof step classification accuracy without conditioning

	Logistic regression	1D CNN	1D CNN-LSTM
Accuracy with char input	0.71	0.82	**0.83**
Accuracy with token input	0.71	**0.83**	0.77

Table 3: HolStep proof step classification accuracy with conditioning

	Logistic regression	Siamese 1D CNN	Siamese 1D CNN-LSTM
Accuracy with char input	0.71	0.81	**0.83**
Accuracy with token input	0.71	0.82	0.77

1. None of our conditioned models appear to be able to improve upon the unconditioned models, which indicates that our architectures are not able to leverage the information provided by the conjecture. The presence of the conditioning does however impact the training profile of our models, in partic- ular by making the 1D CNN model converge faster and overfit significantly quicker (figs. 5 and 6).