Learning Complex Actions from Proofs in Theorem Proving

Zsolt Zombori[1] and Josef Urban[2]

[1] Alfréd Rényi Institute of Mathematics, Budapest
[2] Czech Technical University in Prague

Introduction We propose to develop procedures that extend simple theorem provers with complex actions that are the result of learning. Learning is based on traces of successful proof attempts: we will use inductive logic programming (ILP) [5] to learn Prolog programs that can reproduce (some repetitive parts of) the proofs found. Such programs are then incorporated into the next iteration of the prover as new actions, resulting in a hierarchy of more advanced provers. This approach has several motivations:

1. Long, repetitive parts of the proofs can be delegated to algorithms that are not burdened with search, resulting in shorter proof search. This is a well established approach used in current proof assistants and SMT solvers that implement a number of *manually* designed algorithms, tactics and decision procedures combined in various ways with the proof search.
2. We believe such algorithms are gradually learned by humans who generalize over the proofs found so far. Our goal is to emulate this process, starting from simple actions and generating more and more complex ones, in a similar way as when human mathematicians develop more and more advanced proving methods. The generated programs are typically much shorter and easier to understand than the proof traces, hence they foster human understanding.
3. With more and more recent work related to theorem proving using learned guidance, we argue that theorem provers should start moving focus from simple statistical learning and guidance of the primitive actions to setups that also learn symbolically new (more complicated) actions (executable symbolic programs) that are added to the portfolio of statistically guided proof actions. Compared to standalone statistical learning, symbolic programs enjoy a number of interesting properties - among others the possibility to formally prove their correctness for certain classes of inputs.

Learning Setup We use the rlCoP [4] system, based on Monte Carlo Tree Search (MCTS) guiding the leanCoP [6] connection tableau prover. We have back-ported the RL (reinforcement learning) extensions to the Prolog language in which leanCoP was originally implemented. We call this system plCoP. Prolog was chosen because both prover actions and ILP-learned programs are easy to represent as Prolog clauses and it is easy to incorporate a learned program into the set of valid inference steps. rlCoP learning consists of iterations of proof search using MCTS (data collection) and model fitting (training XGBoost [1] policy and value regressors), following the approach in [8].

One can very naturally interweave this process with occasional ILP-style program generation. Once the prover has generated some proof traces, we try to generate a program that reproduces the proofs. This program is incorporated into the prover as a new action: selecting this action corresponds to executing the program on the current goal. The program is very similar to ITP style tactics or SMT style decision procedures, with the important novelty that it was learned from proof traces.

Program Generation using Simple Inductive Logic Programming A proof trace in leanCoP is a sequence of goal-action pairs. A goal is a literal and an action is an ordered formula in clausal normal form, called *contrapositive*. For each step there is some substitution σ that unifies goal G and the negation of the first literal of contrapositive $A = A_1 \vee A_2 \vee \ldots \vee A_n$, i.e. $G\sigma = \neg A_1\sigma$. Such a pair can be easily turned into the following Prolog clause:

$A_1\sigma$:- $\neg A_2\sigma, \neg A_3\sigma \ldots \neg A_n\sigma$.

This rule is an instance of A specialized to goal G and it is typically too specific to be used in the final generated program. However, if we consider several instantiations of the contrapositive A (coming from one or more proofs), then we can compute the Least General Generalization (lgg) [7] of the instances. For each contrapositive that occurs in the proof traces, we create a new clause using the lgg operator. Next, the clauses are ordered (we illustrate program generation in Appendix A):

1. If the head of clause C_1 is more specific than that of clause C_2, then C_1 comes before C_2.
2. If two clauses are incomparable with respect to head specificity, then the one that occurs more frequently in the proof traces comes first.

Experiments Our experiments are preliminary. We ran plCoP on simple arithmetic equalities of the form $N_1\{\cdot,+\}N_2 = N_3$ with N_i nonnegative integers, using the axioms of Robinson Arithmetic, described in Appendix B. Proofs of these problems have a strong shared structure, however, they can get very long as numbers increase. The training set consisted of all 200 problems with $N_1, N_2 < 10$. We ran two experiments: one using standard leanCoP and another using leanCoP extended with paramodulation. Adding paramodulation to leanCoP makes it a more powerful prover when it comes to equational reasoning. However, the presence of new valid actions makes it harder to navigate in the search space. The addition of paramodulation is so far manual, but it is also an example of a more complicated action that we can (given enough background knowledge) try to learn automatically from many proof traces that use equality and congruence axioms.

The programs generated from plCoP proof traces can be found in Appendix C (leanCoP only) and Appendix D (leanCoP plus paramodulation). In both cases, the generated programs can fall into infinite loop by repeatedly applying some unproductive rules. However, such loops can be avoided by only allowing regular proofs, i.e., the same goal cannot occur twice on the same path. This is a well known optimization that does not affect completeness. We can ensure regularity by writing a simple Prolog interpreter that keeps track of all literal on the current branch. We provide the interpreter code in Appendix E. Once regularity is ensured, the programs in both experiments can generate proofs for all problems, irrespective of the numbers inside. In this example, the learned program solves a very specific class of problems, so its benefit is limited, however, plCoP can be extended with it as an extra action and the system can learn when it is worth using it.

Conclusion and Future Work Our project aims to extend theorem provers with complex actions that are learned from proof traces using ILP. We believe that this approach is suitable to delegate large parts of the proof task to deterministic algorithms, allowing proof search to focus on parts that truly require search. So far, we have back-ported rlCoP to Prolog and experimented with simple ILP for learning arithmetic from the proof traces. The next steps include addition of more advanced ILP learning over richer domains with larger background knowledge and better statistical guidance of the ILP search for suitable Prolog programs analogous to our existing efficient statistical guidance of ATPs [4, 2, 3].

References

[1] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM.

[2] Karel Chvalovský, Jan Jakubuv, Martin Suda, and Josef Urban. ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 197–215. Springer, 2019.

[3] Jan Jakubuv and Josef Urban. Hammering mizar by learning clause guidance. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPIcs*, pages 34:1–34:8. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[4] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olsák. Reinforcement learning of theorem proving. In *NeurIPS*, pages 8836–8847, 2018.

[5] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *J. Log. Program.*, 19/20:629–679, 1994.

[6] Jens Otten and Wolfgang Bibel. leanCoP: lean connection-based theorem proving. *J. Symb. Comput.*, 36:139–161, 2003.

[7] Gordon D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5:153–163, 1970.

[8] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.

Appendix A Example Program Generation

Suppose our proof traces use the following contrapositives:

- $X + s(Y) \neq s(X + Y)$ used on goals $0 + s(0) = X$ and $s(0) + s(0) = X$.
- $X = Y \vee X \neq Z \vee Z \neq Y)$ used on goals $X + s(0) = s(s(0))$ and $s(0) + X = s(0)$.

The corresponding rule instances and their least general generalizations are

Rule1: $0 + s(0) = s(0 + 0)$.
Rule2: $s(0) + s(0) = s(s(0) + 0)$.
Lgg: $X + s(0) = s(X + 0)$.

Rule1: $X + s(0) = s(s(0))$ `:-` $X + s(0) = Z, Z = s(s(0))$.
Rule2: $s(0) + X = s(0)$ `:-` $s(0) + X = Z, Z = s(0)$.
Lgg: $X + Y = s(V)$ `:-` $X + Y = Z, Z = s(V)$.

We obtained two clauses with heads $X + s(0)$ and $X + Y$. The former is more specific, so that comes first. The resulting program is:

```
X+s(0) = s(X + 0).
X+Y = s(V):-
    X + Y = Z, Z = s(V)
```

Appendix B Axioms of Robinson Arithmetic

Our experiments with Robinson Arithmetic use the following axioms:

- $\forall X : \neg(o = s(X))$
- $\forall X, Y : (s(X) = s(Y)) \Rightarrow (X = Y)$
- $\forall X : plus(X, o) = X$
- $\forall X, Y : plus(X, s(Y)) = s(plus(X, Y))$
- $\forall X : mul(X, o) = o$
- $\forall X, Y : mul(X, s(Y)) = plus(mul(X, Y), X)$

The axioms are automatically extended in leanCoP with rules for handling equality:

- $\forall X : X = X$
- $\forall X, Y : (X = Y) \Rightarrow (Y = X)$
- $\forall X, Y, Z : (X = Y) \wedge (Y = Z) \Rightarrow (X = Z)$
- $\forall X, Y : (X = Y) \Rightarrow (s(X) = s(Y))$
- $\forall X_1, X_2, Y_1, Y_2 : (X_1 = X_2) \wedge (Y_1 = Y_2) \Rightarrow plus(X_1, Y_1) = plus(X_2, Y_2)$
- $\forall X_1, X_2, Y_1, Y_2 : (X_1 = X_2) \wedge (Y_1 = Y_2) \Rightarrow mul(X_1, Y_1) = mul(X_2, Y_2)$

Appendix C Program Generation from Proof Traces in leanCoP

After running plCoP on the arithmetic equalities described in Section , we used the successful proof traces to generate the following program:

```
eq(mul(A,o),o).
eq(plus(A,s(B)),s(plus(A,B))).
eq(s(A),s(B)):-
    eq(A,B).
eq(plus(A,o),A).
eq(plus(s(s(s(A))),s(s(B))),plus(s(s(s(A))),s(s(B)))):-
    eq(s(s(s(A))),s(s(s(A)))),eq(s(s(B)),s(s(B))).
eq(A,A).
eq(mul(A,s(B)),plus(mul(A,B),A)).
eq(A,B):-
    eq(A,C),eq(C,B).
eq(A,B):-
    eq(B,A).
eq(A,B):-
    eq(s(A),s(B)).
```

This program – when only regular proofs are allowed – can prove any of the arithmetic problems, irrespective of the numbers inside.

The program contains some unnecessary rules, which can be iteratively removed, making sure that the coverage does not change. After pruning, we obtain the following program:

```
eq(mul(A,o),o).
```

```
eq(plus(A,s(B)),s(plus(A,B))).
eq(s(A),s(B)):-
    eq(A,B).
eq(plus(A,o),A).
eq(mul(A,s(B)),plus(mul(A,B),A)).
eq(A,B):-
    eq(A,C),eq(C,B).
```

Appendix D Program Generation from Proof Traces in leanCoP Extended with Paramodulation

After running plCoP using paramodulation on the arithmetic equalities described in Section , we used the successful proof traces to generate the following program:

```
eq(plus(A,s(B)),s(plus(A,B))).
eq(plus(s(A),s(B)),plus(s(A),C)):-
    eq(s(A),s(A)),eq(s(B),C).
eq(plus(A,s(B)),C):-
    eq(s(plus(A,B)),C),true.
eq(s(A),plus(B,s(C))):-
    eq(s(A),s(plus(B,C))),true.
eq(plus(A,o),A).
eq(plus(A,o),B):-
    eq(A,B),true.
eq(A,plus(B,o)):-
    eq(A,B),true.
eq(A,A).
eq(mul(s(s(s(A))),s(s(s(B)))),plus(mul(s(s(s(A))),s(s(B))),s(s(s(A))))).
eq(mul(s(o),s(s(o))),mul(s(o),plus(s(s(o)),o))):-
    eq(s(o),s(o)),eq(s(s(o)),plus(s(s(o)),o)).
eq(mul(A,s(B)),C):-
    eq(plus(mul(A,B),A),C),true.
eq(A,mul(s(B),s(C))):-
    eq(A,plus(mul(s(B),C),s(B))),true.
eq(mul(A,o),o).
eq(s(plus(A,B)),C):-
    eq(plus(A,s(B)),C),true.
eq(s(A),s(B)):-
    eq(A,B).
eq(A,B):-
    eq(B,A).
eq(A,B):-
    eq(s(A),s(B)).
eq(A,B):-
    eq(A,C),eq(C,B).
```

This program – when only regular proofs are allowed – can prove any of the arithmetic problems,

irrespective of the numbers inside.

The program contains some unnecessary rules, which can be iteratively removed, making sure that the coverage does not change. After pruning, we obtain the following program:

```
eq(plus(A,s(B)),C):-
    eq(s(plus(A,B)),C),true.
eq(s(A),plus(B,s(C))):-
    eq(s(A),s(plus(B,C))),true.
eq(plus(A,o),B):-
    eq(A,B),true.
eq(A,plus(B,o)):-
    eq(A,B),true.
eq(A,A).
eq(mul(A,s(B)),C):-
    eq(plus(mul(A,B),A),C),true.
eq(A,mul(s(B),s(C))):-
    eq(A,plus(mul(s(B),C),s(B))),true.
eq(mul(A,o),o).
eq(s(A),s(B)):-
    eq(A,B).
```

Appendix E Prolog Interpreter

We provide a small Prolog interpreter (written in Prolog) that extends regular Prolog by adding reduction step and and filtering irregular proofs.

```
execute(true, _, []):- !.
execute((G1, G2), Path, Proof):- !,
    execute(G1, Path, Proof0),
    execute(G2, Path, Proof1),
    append(Proof0, Proof1, Proof).
execute(Goal, Path, Proof):-
    ( has_loop(Goal, Path) -> fail
    ; reduction(Goal, Path, Proof)
    ; extension(Goal, Path, Proof)
    ).

negate(neg(Goal), Goal):- !.
negate(Goal, neg(Goal)).

has_loop(Goal, Path):-
    member(G, Path), G == Goal, !.

reduction(Goal, Path, [red(Clause)]):-
    negate(Goal, NegGoal),
    member(NegGoal, Path),
    copy_term(Goal-NegGoal, Clause).
```