LEMMA: Bootstrapping High-Level Mathematical Reasoning with Learned Symbolic Abstractions

Zhening Li[1,*] **Gabriel Poesia**[2,*]

Omar Costilla-Reyes[1] **Noah Goodman**[2] **Armando Solar-Lezama**[1]

[1] Computer Science and Artificial Intelligence Lab, MIT
[2] Stanford University
[1]{zli11010,costilla,asolar}@csail.mit.edu, [2]{poesia,ngoodman}@stanford.edu

Abstract

Humans tame the complexity of mathematical reasoning by developing hierarchies of *abstractions*. With proper abstractions, solutions to hard problems can be expressed concisely, thus making them more likely to be found. In this paper, we propose Learning Mathematical Abstractions (LEMMA): an algorithm that implements this idea for reinforcement learning agents in mathematical domains. LEMMA augments Expert Iteration with an abstraction step, where solutions found so far are revisited and rewritten in terms of new higher-level actions, which then become available to solve new problems. We evaluate LEMMA on two mathematical reasoning tasks—equation solving and fraction simplification—in a step-by-step fashion. In these two domains, LEMMA improves the ability of an existing agent, both solving more problems and generalizing more effectively to harder problems than those seen during training.

1 Introduction

Mathematical reasoning has been a key ability behind many achievements of human intelligence. By formalizing ideas and working with symbolic representations (e.g., number systems and geometry), we can ultimately develop technology and make predictions that would have been utterly impossible from perception and intuition alone (e.g., sending robots to space). As a result, AI systems capable of performing complex symbolic reasoning could amplify this impact by helping advance research as well as education.

Reasoning problems can be posed as searching for a sequence of steps arriving at a solution, such as a complete mathematical proof. Systematic search for sequences of steps can dramatically benefit from learning: patterns from previous problems can help guide future searches towards more promising sequences. This idea of combining search and learning has been widely applied, with notable domains including neural theorem proving [3, 5, 6, 11, 16], program synthesis [2, 4, 15] and game playing [1, 13].

Learning improves search, but only up to a point. The space of sequences of steps still grows exponentially as solution length increases. In addition to using intuition from past problems, humans keep search manageable by developing *mathematical abstractions*: useful higher-level actions that capture reusable reasoning patterns, allowing increasingly complex problems to be solvable within a small number of steps. Indeed, with the right level of abstraction, even non-trivial statements

[*]Equal contribution

36th Conference on Neural Information Processing Systems (NeurIPS 2022) Workshop on Math-AI.

(e.g., that $x^{19} - x^5 + 3 = 0$ has a solution) can have short solutions (e.g., "all polynomials have a root by the fundamental theorem of algebra"). Similarly, "simple" problems (e.g., synthesizing a program to sort a list) can become arbitrarily hard depending on the action space (e.g., in an Assembly language). Therefore, agents that aim to solve increasingly complex problems should also benefit from learning abstractions. Ultimately, problem difficulty is not an absolute measure: it always depends on the available actions.

The idea of learning abstractions as a way to bootstrap towards harder problems has been successfully leveraged in program synthesis. Notably, DreamCoder [4] has demonstrated this idea by learning a library of functions from its own solutions so far—programs that solved earlier program synthesis tasks. Equipped with those new functions, new tasks come to reach, as their solutions can now be expressed by short programs. We take that inspiration to propose Learning Mathematical Abstractions (LEMMA): a method that combines *learning to search* with abstraction learning for mathematical reasoning. LEMMA naturally applies to methods similar in style to Expert Iteration (ExIt) [1], where agents alternate between solving mathematical problems and training a model to guide future searches. After a large enough batch of problems has been solved, LEMMA mines useful abstractions from its solutions and adds them to the agent's action space. In experiments in two domains from the Common Core mathematical environments—equation solving and fraction simplification—we observe that LEMMA improves the success rate of the base learning method and allows it to generalize better to harder problems in a zero-shot fashion.

2 Related Work

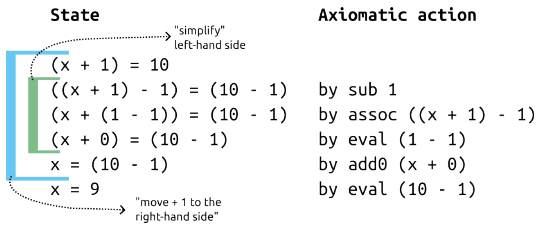
Our work proposes to augment methods that alternate between learning and search to solve mathematical reasoning problems. Several methods of this flavor have been introduced recently, such as Expert Iteration [1] and AlphaZero [13] for game playing, HTPS [6] and GPT-f [11] for neural theorem proving, and ConPoLe [10] for mathematical problems from the Common Core environments.

Solving symbolic mathematical problems have been a challenge since early Computer Algebra Systems [7], and they have gained significant attention in the Reinforcement Learning (RL) community in recent years. In particular, ConPoLe has been shown to learn how to solve problems in the educational mathematical domains of the Common Core environments [10], without having access to human solutions. The Common Core environments are a simplified setting compared to industrial theorem proving languages, like Lean [8] or Isabelle/HOL [9]. A range of significant work has been developed in RL agents to find proofs in these languages [11, 6, 3], typically using a language model fine-tuned on human proofs as the action generator.

The idea of learning abstractions has been successfully explored in program synthesis, with recent notable examples including DreamCoder [4] and LAPS [15]. In these settings, abstractions are functions induced from synthesized programs and added to a library that is available for future problems. We note the similarity between such functions and what is known in theorem-proving languages as *tactics*: procedures that perform common operations on proof objects. This analogy, which is made precise by the Curry-Howard correspondence [14], is a central motivation behind our work. While the abstractions we induce are rather simple compared to tactics in general theorem proving languages, we hope that our work will inspire the investigation of *tactic induction* in more complex settings, which would be a significant step towards developing autonomous agents for mathematical reasoning.

3 Method

Consider the example of a formal solution to an equation in Figure 1. Here, the underlying formal system allows one to apply low-level properties of equations and of operations on real numbers, such as that 0 is the identity element of addition (`add0`), and that equality is preserved if we subtract equal terms from both sides (`sub`). These axioms can be combined to solve arbitrary linear equations, but even simple-looking equations might require dozens of steps [10]. Looking at the solution, one can identify intuitive high-level operations that are being performed, such as "moving the $+1$ to the other side" (blue segment), which itself involves first subtracting 1 from both sides, then "simplifying the left-hand side" (green segment).



```
State                         Axiomatic action
(x + 1) = 10
((x + 1) - 1) = (10 - 1)      by sub 1
(x + (1 - 1)) = (10 - 1)      by assoc ((x + 1) - 1)
(x + 0) = (10 - 1)            by eval (1 - 1)
x = (10 - 1)                  by add0 (x + 0)
x = 9                         by eval (10 - 1)
```

Figure 1: Example of an axiomatic formal solution to a simple linear equation. We would like to automatically discover abstractions, or high-level actions, such as "simplify an expression" or "move a term to the other side", which can be expressed in terms of the base axioms.

LEMMA will attempt to synthesize these latent high-level actions by finding similarities in solutions it has found so far. Since mathematical actions typically take parameters, exactly identical actions are rather rare. Instead, we leverage a *projection function* π_A over actions which will keep some information about the action, but discard details so that LEMMA finds matches. We propose two action projection functions in LEMMA. The first function, which we call `SeqAbs`, only keeps the axiom name, ignoring its arguments. In this case, an abstraction simply corresponds to a sequence of axioms that are to be applied. We also consider a more specific projection function that takes the (relative) position of the arguments in the original expression into account. This projection allows us to capture an abstraction such as "evaluate the left child of a binary operation and then the operation itself", instead of simply "evaluate twice" which `SeqAbs` can represent. This more specific abstraction could lead from the equation $x+(1+2) = (3+4)+5$ directly into $x+(1+2) = 12$. The abstraction "evaluate twice" can also produce this successor, but any of the two "evaluate" actions might apply to $(1+2)$, leading to a larger number of successors in general. We call the method with parameterized abstractions `RelAbs`, since it uses sequences of axioms as well as a relative indexing of where they are applied to.

Discovering abstractions Consider a data set $\mathcal{D}$ of sequences of π_A-projected actions from an action space $\mathcal{A}$, corresponding to solutions found so far. The set of all contiguous subsequences of actions from $\mathcal{D}$ form a set $\mathcal{L}$ of candidate abstractions. From those, we select a subset $L \subseteq \mathcal{L}$ by optimizing the Bayesian criterion: $P(L \mid \mathcal{D}) \propto P(\mathcal{D} \mid L)P(L)$, where $P(L)$ is a uniform prior over candidate abstractions extracted from $\mathcal{D}$.[2]

To assign a meaningful probabilistic interpretation to $P(\mathcal{D} \mid L)$, imagine a random agent that takes actions chosen uniformly from $\mathcal{A} \cup L$, the set of available actions generated by both axioms $\mathcal{A}$ and abstractions L. We take $P(\mathcal{D} \mid L)$ to be the probability that this agent would generate exactly the sequences observed in $\mathcal{D}$. This yields the following negative log-likelihood objective:

$$\min_{L \subseteq \mathcal{L}} \quad J_{\mathcal{D}}(L) \equiv -\log P(\mathcal{D} \mid L) = \sum_{a \in \mathcal{A} \cup L} f_{\mathcal{D}/L}(a) \log |\mathcal{A} \cup L|,$$

where $f_{\mathcal{D}/L}(a)$ denotes the frequency of action a within $\mathcal{D}/L$. Here, $\mathcal{D}/L$ refers to $\mathcal{D}$ after abstraction with L where all subsequences of actions in $\mathcal{D}$ corresponding to an abstraction in L have been replaced by that abstraction. Thus, our objective is to find abstractions that make it most likely to generate the observed solutions when acting randomly using primitive actions and abstract actions from L. We solve this minimization problem approximately with a greedy algorithm. Starting with $L = \emptyset$, we greedily pick the abstraction a in $\mathcal{L}$ that decreases the objective the most and rewrite $\mathcal{D}$ with this new abstraction; repeat until the objective can no longer be decreased (Algorithm 1).

Applying abstractions Carrying out the axiomatic actions described by an abstraction in the environment is performed by a simple depth first search where next states are limited by the specification of the abstraction. For `SeqAbs`, the next states are obtained by applying the next axiom in all possible ways on the current state. For `RelAbs`, we only include the subset of these actions that are applied to the correct position in the expression tree relative to the previous action.

Bootstrapping learning with abstractions The full learning procedure consists of alternating episodes of reinforcement learning with a few rounds of abstraction (Algorithm 2). During each round of learning, we store in $\mathcal{D}$ successful action sequences taken by the agent during search. After

[2]One could use other priors, for example by positing that shorter abstractions are more likely.

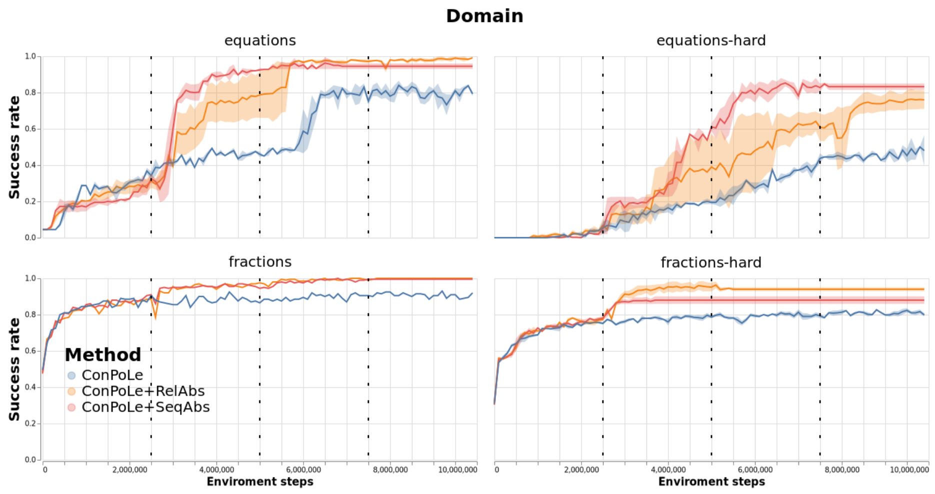


Figure 2: Success rate of agents on held-out problems on the four domains we test on, with standard errors across 3 random seeds. Vertical lines separate the 4 rounds of reinforcement learning.

learning, we apply LEMMA on $\mathcal{D}$ to obtain abstractions L and the abstracted data set $\mathcal{D}/L$. During the next round of learning, we augment the agent's action space to include these new abstractions. In addition, before continuing reinforcement learning, we perform a brief session of imitation learning using $\mathcal{D}/L$ as the expert, so that the agent quickly learns to use the new available actions.

4 Experiments

We evaluate LEMMA on the two hardest tasks from the Common Core environments [10]: `equations` and `fractions`. Each of these environments defines a distribution over starting states (linear equations or expressions with fractions, respectively) and an action generator that applies a set of axioms to a state, listing all available axiom applications along with the successor states they lead to. We use ConPoLe [10] as our base learning method, which has been shown to perform well in these domains. Details about the environments, including axioms, are given in [10].

In addition, we created harder versions of each of these two domains. For equations, we took the set of equations used in Rafferty et al. [12] as templates, which are longer than the original templates used in the Common Core environments. For fractions, we increase the number of primes in the environment (4 to 5) as well as the maximum number of prime factors used when sampling constants (4 to 6). We denote these environments as `equations-hard` and `fractions-hard`, respectively. For each environment, we train vanilla ConPoLe with and without LEMMA for 10^7 steps. We use 4 rounds of abstraction learning, which happen every 2.5×10^6 steps. Every 10^5 steps, we evaluate all agents on a fixed set of 200 held-out problems, on which we report their success rates.

Figure 2 shows our main result: the success rate of all agents during the course of training. In all domains, agents obtained highest success rates with abstractions. `SeqAbs` performs best on `equations-hard`, while in all other domains, `RelAbs` achieves the best success rate. Abstractions improve results in all cases, though the best projection function might depend on the domain. [3] LEMMA successfully learned interpretable abstractions, examples of which are given in Appendix B.

Finally, to evaluate generalization, we took the best agents trained on `equations` and evaluated them on `equations-hard`. We found the gap between their performances under this distribution shift: the success rate of vanilla ConPoLe dropped from 92% to 45.5%, whereas `ConPoLe + RelAbs` went from 99.5% to 60% on the harder evaluation set *without further training*. This result suggests that abstractions can help an agent better generalize to longer solutions.

[3]This dependence is subtle: a coarser projection function makes abstractions easier to learn (since matches between common subsequences occur after fewer solutions), but at the same time make the action space in general larger (since they are more widely applicable).

5 Conclusion

We presented LEMMA, a method for augmenting mathematical reasoning agents with learned high-level actions, a symbolic analog to temporal abstractions in general reinforcement learning. Our method improved the success rate of an agent in two mathematical reasoning tasks, and was able to recover intuitive abstractions without additional supervision. We believe these insights are applicable to more expressive theorem proving languages. As we have argued, which problems are within reach of a solver heavily depends on what actions are available. Therefore, synthesizing actions of increasing abstraction may be a crucial step towards automated mathematical reasoning.

References

[1] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. *Advances in Neural Information Processing Systems*, 30, 2017.

[2] M Balog, AL Gaunt, M Brockschmidt, S Nowozin, and D Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations (ICLR 2017)*, 2017.

[3] Kshitij Bansal, Sarah Loos, Markus Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher order logic theorem proving. In *International Conference on Machine Learning*, pp. 454–463. PMLR, 2019.

[4] Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*, pp. 835–850, 2021.

[5] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Miroslav Olšák. Reinforcement learning of theorem proving. *Advances in Neural Information Processing Systems*, 31, 2018.

[6] Guillaume Lample, Marie-Anne Lachaux, Thibaut Lavril, Xavier Martinet, Amaury Hayat, Gabriel Ebner, Aurélien Rodriguez, and Timothée Lacroix. Hypertree proof search for neural theorem proving. *arXiv preprint arXiv:2205.11491*, 2022.

[7] William A Martin and Richard J Fateman. The macsyma system. In *Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, pp. 59–75, 1971.

[8] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The lean theorem prover (system description). In *International Conference on Automated Deduction*, pp. 378–388. Springer, 2015.

[9] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[10] Gabriel Poesia, WenXin Dong, and Noah Goodman. Contrastive reinforcement learning of symbolic reasoning domains. *Advances in Neural Information Processing Systems*, 34:15946–15956, 2021.

[11] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning. *arXiv preprint arXiv:2202.01344*, 2022.

[12] Anna N Rafferty, Rachel A Jansen, and Thomas L Griffiths. Assessing mathematics misunderstandings via bayesian inverse planning. *Cognitive science*, 44(10):e12900, 2020.

[13] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362 (6419):1140–1144, 2018.

[14] Philip Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, 2015.

[15] Catherine Wong, Kevin M Ellis, Joshua Tenenbaum, and Jacob Andreas. Leveraging language to learn program abstractions and search heuristics. In *International Conference on Machine Learning*, pp. 11193–11204. PMLR, 2021.

[16] Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. Tacticzero: Learning to prove theorems from scratch with deep reinforcement learning. *Advances in Neural Information Processing Systems*, 34:9330–9342, 2021.

A Algorithms

Algorithm 1 The LEMMA abstraction algorithm

```
Input: Axioms $\mathcal{A}$, data set $\mathcal{D}$ of example solutions, abstraction type $abs\_type$
Output: Library $L$ of learned abstractions, abstracted solutions $\mathcal{D}_{\text{abs}} = \mathcal{D}/L$
  $L \leftarrow \emptyset$
  $\mathcal{D}_{\text{abs}} \leftarrow \mathcal{D}$
  $\mathcal{L} \leftarrow$ get_candidate_abstractions($\mathcal{D}, abs\_type$)
  repeat
      $a_{\text{opt}} \leftarrow \arg\max_{a \in \mathcal{L}} S_{\mathcal{D}_{\text{abs}}}(a)$        ▷ $S_{\mathcal{D}_{\text{abs}}}(a) := J_{\mathcal{D}_{\text{abs}}}(L) - J_{\mathcal{D}_{\text{abs}}}(L \cup \{a\})$
      $s_{\text{opt}} \leftarrow S_{\mathcal{D}_{\text{abs}}}(a_{\text{opt}})$
      if $S_{\mathcal{D}_{\text{abs}}}(a_{\text{opt}}) \geq 0$ then
          $L \leftarrow L \cup \{a_{\text{opt}}\}$
          $\mathcal{D}_{\text{abs}} \leftarrow \mathcal{D}_{\text{abs}}/\{a_{\text{opt}}\}$
      end if
  until $s_{\text{opt}} < 0$
  return $L, \mathcal{D}_{\text{abs}}$
```

Algorithm 2 Reinforcement learning with LEMMA

```
Input:
    • Environment $E(\mathcal{A})$ with axioms $\mathcal{A}$
    • Abstraction type $abs\_type$
    • Number of learning rounds $k$
Output: Learned policy parameters $\theta^*$, library $L_{\text{all}}$ of all learned abstractions
  $\theta \leftarrow$ init_parameters()
  for $i \leftarrow 1$ to $k$ do
      if $i > 1$ then                                  ▷ Imitation learning with abstracted solutions
          $\theta \leftarrow$ LearnFromImitation($E(\mathcal{A}), \mathcal{D}_{\text{abs}}, \theta$)
      end if
      $\mathcal{D}, B \leftarrow \emptyset$
      for $episode \leftarrow 1$ to $N$ do             ▷ ExIt-style reinforcement learning, e.g., ConPole
          $problem \leftarrow E(\mathcal{A})$.sample_problem()
          $solution, visited\_states \leftarrow$ beam_search($E(\mathcal{A}), problem, \theta$)
          $\mathcal{D}$.add($solution$)
          $B$.add($visited\_states$)
          $\theta \leftarrow$ BatchGD($B, \theta$)
      end for
      if $i < k$ then                                  ▷ Abstraction with LEMMA
          $L_i, \mathcal{D}_{\text{abs}} \leftarrow$ LEMMA($\mathcal{A}, \mathcal{D}, abs\_type$)
          $\mathcal{A} \leftarrow \mathcal{A} \cup L_i$
      end if
  end for
  return $\theta^* = \theta, L_{\text{all}} = \bigcup_{i=1}^{k-1} L_i$
```

B Examples of learned abstractions

Here, we present example abstractions that LEMMA discovered during the last round of abstraction in main experiment on the `equations-hard` and `fractions` domains, with projection function `RelAbs`.

Abstractions will be written in the format

$$a_1, a_2, \ldots, a_k : (p_1, q_2), (p_2, q_3), \ldots, (p_{k-1}, q_k)$$

where the a_i's are axioms, and each pair (p_i, q_{i+1}) specifies the relative position of application between a_i and a_{i+1}. Suppose we represent the absolute position of application of a_i with a string

$P_i \in \{L, R\}^*$ (L = left child, R = right child) representing the node in the equation's expression tree to which a_i is applied. (For example, the root node is the empty string ε, and the left child of the right child of the root node is RL.) Then p_i, q_{i+1} are obtained by removing the maximal common prefix from P_i and P_{i+1}. For high-level abstractions, each a_i can be an abstraction itself, written inside curly braces. In this case, the relative position (p_i, q_{i+1}) between a_i and a_{i+1} is understood as the relative position between the last axiom in a_i and the first axiom in a_{i+1}.

`equations-hard`: In this domain, LEMMA discovered 15 abstractions after 3 rounds of abstraction. Here, we discuss 3 of the abstractions that have remarkably high interpretability.

1. $A_1 = \{\texttt{sub}, \texttt{eval}, \texttt{comm} : (\varepsilon, R), (R, LL)\}, \{\texttt{assoc}, \texttt{eval}, \texttt{add0} : (\varepsilon, R), (R, \varepsilon)\} : (L, \varepsilon)$

   A_1 reduces any equation of the form $(b + ax) = c$ to $ax = [c - b]$ in one step, which is what we usually mean when we say "subtract b from both sides."[4] This reduces the search depth by 5 for the class of equations reducible to $(b+ax) = c$, hence facilitating the agent's search for their solutions.

   The first 3 axioms of A_1 constitute a subabstraction that subtracts b from both sides of the equation and puts b and $-b$ next to each other on the left-hand side. This transforms $(b + ax) = c$ into $((ax + b) - b) = [c - b]$. The final 3 axioms constitute a subabstraction that simplifies an expression $(A + B) - B$ to A. Thus, it simplifies the left-hand side $((ax + b) - b)$ to ax.

   An example instance of the abstraction seen during training solves $(3 + x) = -4$ with a single application of A_1. To see how the individual axioms of the abstraction operate on this example, we present below the solution expanded into its individual axioms.

$$\begin{aligned}
(3 + x) &= (-4) && \\
((3 + x) - 3) &= ((-4) - 3) && \text{(by applying \texttt{sub} with parameter 3 to node } \varepsilon) \\
((3 + x) - 3) &= (-7) && \text{(by applying \texttt{eval} to node } R) \\
((x + 3) - 3) &= (-7) && \text{(by applying \texttt{comm} to node } LL) \\
(x + (3 - 3)) &= (-7) && \text{(by applying \texttt{assoc} to node } L) \\
(x + 0) &= (-7) && \text{(by applying \texttt{eval} to node } LR) \\
x &= (-7) && \text{(by applying \texttt{add0} to node } L)
\end{aligned}$$

2. $A_2 = \{\texttt{add}, \texttt{eval}, \texttt{comm}, \texttt{assoc}, \texttt{comm} : (\varepsilon, R), (R, L), (\varepsilon, \varepsilon), (\varepsilon, L)\},$
   $\{\texttt{assoc}, \texttt{eval}, \texttt{add0} : (\varepsilon, R), (R, \varepsilon)\} : (L, \varepsilon)$

   This abstraction simplifies any equation of the form $(ax - b) = c$ to $ax = [c + b]$ in one step, which expresses what we usually mean when we say "add b to both sides." Thus, for the class of equations reducible to this form, A_2 reduces the search depth by 7, significantly facilitating search for their solutions.

   Note that A_2's second subabstraction $\texttt{assoc}, \texttt{eval}, \texttt{add0} : (\varepsilon, R), (R, \varepsilon)$ is the same as the second subabstraction of A_1. This shows the utility of iteratively developing a hierarchy of abstractions: higher-level abstractions in later rounds of abstraction can reuse low-level abstractions learned during earlier rounds as subcomponents.

   The first 5 axioms of A_2 constitute the subabstraction that that adds b to both sides and rearranges the left-hand side to swap the positions of b and $-b$, resulting in $(ax + b) - b = [c - b]$. This is done since the `assoc` axiom in the ConPoLe environment cannot be directly applied to expressions of the form $(A - B) + C$. The second subabstraction, as already described, fully simplifies the left-hand side to ax.

   An example instance of the abstraction seen during training simplifies $(8x - 9) = 5$ to $8x = 14$ with a single application of A_2. We present below the expanded sequence of

[4] $[c - b]$ refers to the constant that results from evaluating $c - b$.

axiomatic steps to show how the individual axioms operate on this example.

$$\begin{aligned}
(8x-9) &= 5 && \\
((8x-9)+9) &= (5+9) && \text{(by applying } \texttt{add} \text{ with parameter } 9 \text{ to node } \varepsilon) \\
((8x-9)+9) &= 14 && \text{(by applying } \texttt{eval} \text{ to node } R) \\
(9+(8x-9)) &= 14 && \text{(by applying } \texttt{comm} \text{ to node } L) \\
((9+8x)-9) &= 14 && \text{(by applying } \texttt{assoc} \text{ to node } L) \\
((8x+9)-9) &= 14 && \text{(by applying } \texttt{comm} \text{ to node } LL) \\
(8x+(9-9)) &= 14 && \text{(by applying } \texttt{assoc} \text{ to node } L) \\
(8x+0) &= 14 && \text{(by applying } \texttt{eval} \text{ to node } LR) \\
8x &= 14 && \text{(by applying } \texttt{add0} \text{ to node } L)
\end{aligned}$$

3. $A_3 = \texttt{div}, \texttt{eval}, \texttt{comm}, \texttt{assoc}, \texttt{eval}, \texttt{mul1} : (\varepsilon, R), (R, LL), (L, \varepsilon), (\varepsilon, R), (R, \varepsilon)$

   This abstraction solves any equation of the form $ax = b$ in one step, or, more generally, simplifies an equation of the form $aE = b$ to $E = [b/a]$ in one step for any expression E. Thus, A_3 expresses our concept of "dividing both sides by a."

   In the example equation for A_2, the agent now solves $(8x - 9) = 5$ in just two steps: the first step simplifies it to $8x = 14$ by applying A_2, and the second step solves it to $x = [7/4]$ by applying A_3. This is identical to how most humans would solve the equation.

   As a specific example of how the individual axioms of A_3 operate, we present below how they convert $8x = 14$ into $x = [7/4]$.

$$\begin{aligned}
8x &= 14 && \\
(8x/8) &= 14/8 && \text{(by applying } \texttt{div} \text{ with parameter } 8 \text{ to node } \varepsilon) \\
(8x/8) &= [7/4] && \text{(by applying } \texttt{eval} \text{ to node } R) \\
((x*8)/8) &= [7/4] && \text{(by applying } \texttt{comm} \text{ to node } LL) \\
(x*(8/8)) &= [7/4] && \text{(by applying } \texttt{assoc} \text{ to node } L) \\
(x*1) &= [7/4] && \text{(by applying } \texttt{eval} \text{ to node } LR) \\
x &= [7/4] && \text{(by applying } \texttt{mul1} \text{ to node } L)
\end{aligned}$$

`fractions`: In this domain, LEMMA discovered 15 abstractions after 3 rounds of abstraction learning. Here, we present an example solution the agent produced that contains 2 highly-interpretable abstractions:

$$\begin{aligned}
&21 - [21]/[7] && \\
&21 - 3 && \text{(by applying } A_4 \text{ to nodes } RL, R) \\
&18 && \text{(by applying } A_5 \text{ to nodes } \varepsilon, \varepsilon, \varepsilon, L, \varepsilon)
\end{aligned}$$

where

$$\begin{aligned}
A_4 &= \texttt{factorize}, \texttt{cancel} : (L, \varepsilon) \\
A_5 &= \texttt{mfrac}, \{\texttt{mfrac}, \texttt{combine}, \texttt{eval}, \texttt{simpl1} : (\varepsilon, \varepsilon), (\varepsilon, L), (L, \varepsilon)\} : (\varepsilon, \varepsilon)
\end{aligned}$$

The `fractions` has a small set of 8 axioms, which do not include direct evaluation of division or evaluation of addition/subtraction of integers not in the numerator/denominator of a fraction. Despite this restriction, LEMMA successfully learned abstractions corresponding to these actions: A_4 performs division $[a]/[b]$ if the result is an integer, and A_5 performs addition/subtraction of integers $a \pm b$. This demonstrates another potential application of abstraction learning: it reduces the need for a "perfect" set of axioms that optimizes learning and search, since omitted axioms that would be useful could be discovered as abstractions during abstraction learning.

Again, to show the details of how the axioms of the abstraction play out, we expand the solution above into its individual axiomatic actions:

$21 - [21]/[7]$	
$21 - [(3 * 7)]/[7]$	(by applying `factorize` to node RL)
$21 - 3$	(by applying `cancel` to node R)
$[21]/[1] - 3$	(by applying `mfrac` to node ε*)
$[21]/[1] - [3]/[1]$	(by applying `mfrac` to node ε*)
$[(21 - 3)]/[1]$	(by applying `combine` to node ε)
$[18]/[1]$	(by applying `eval` to node L)
18	(by applying `simpl1` to node ε)

* The environment considers the position of application here to be the root node due to an artifact of how `mfrac` is defined in the domain.