Thesis Proposal

Learning to Discover Proofs and Theorems Without Supervision

Jonathan Laurent

August 2022

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis committee:
André Platzer
Marijn Heule
Zico Kolter
Armando Solar-Lezama

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Abstract

Interactive theorem provers have been successfully used to formally verify safety-critical software. However, doing so requires a significant level of resources and expertise. A promising approach to scaling-up theorem proving and formal verification is to augment tactic-based interactive theorem provers with machine-learning for automation. The dominant approach in this area is to train a neural network to imitate human experts. However, this approach is limited by the acute scarcity of representative proofs.

An alternative approach inspired by the success of AlphaZero is to use reinforcement learning instead and train an agent to interact with a theorem prover via trial and error. Unfortunately, existing tactic-based interfaces offer unbounded action spaces that are hardly amenable to random exploration. Such interfaces are optimized for formalizing human insights in a concise way but often fail to define a tractable search space for deriving those insights in the first place. Moreover, although reinforcement learning alleviates the need for human proofs, the problem remains of providing theorem proving tasks of suitable relevance, diversity and difficulty for the learner.

In this thesis, we propose a novel framework for theorem proving where a teacher agent is trained to generate interesting and relevant tasks while a solver agent is co-trained to solve them. Both agents can leverage domain-specific expert strategies in the form of nondeterministic programs. Choice points in those programs are resolved by neural network oracles that are trained via reinforcement learning in a self-supervised fashion. This allows leveraging minimal amounts of domain knowledge to tackle problems for which training data is entirely unavailable and hard to synthesize.

This work aims to establish conceptual and engineering foundations for such a framework. We introduce novel curriculum learning algorithms and build a new theorem prover based on neural-guided nondeterministic programming. We introduce standard abstractions and design principles for writing teacher and solver strategies. Finally, we plan to evaluate our theorem prover on a collection of tasks such as loop invariant synthesis, deductive program synthesis, arithmetic inequality proving and safe robot planning.

Contents

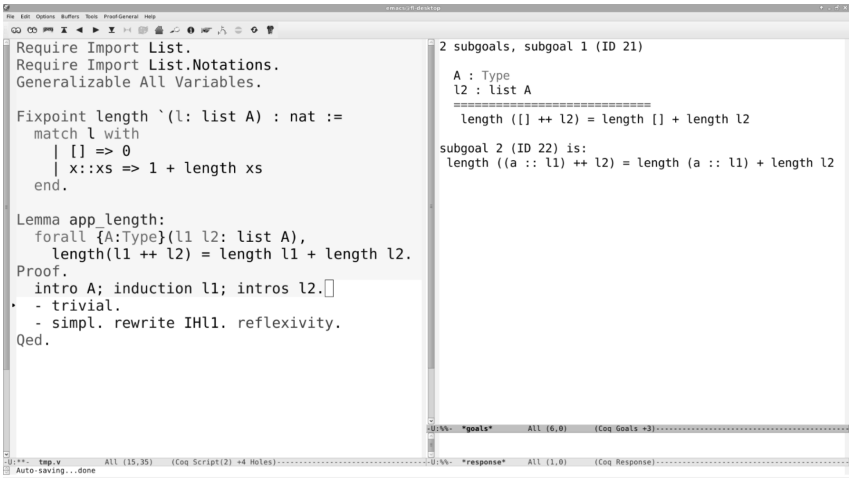


Figure 1: Using the Coq interactive theorem prover to prove a simple property about lists. The property shown here states that the `length` function (as defined above recursively) distributes over list concatenation. Users can prove this property interactively by issuing a sequence of proof commands called *tactics* (shown here between the `Proof` and `Qed` delimiters). Each call to a tactic transforms the current proof goal into a new collection of goals indicating what remains to be proven. In this screenshot, the right pane displays the proof state after executing the first line of the proof script. The dominant approach for integrating machine-learning with interactive theorem proving is to train a neural network to map such proof states to tactic suggestions using imitation learning.

1 Introduction

Interactive theorem provers (Figure 1, [1, 2, 3]) have been successfully used to formally verify large-scale software [4, 5]. However, doing so requires a significant level of resources and expertise. A promising approach to scaling-up theorem proving and formal verification is to augment tactic-based interactive theorem provers with machine-learning for automation. Doing so successfully would not only enable automating most of the low-level details of conducting a formal proof, but it would also provide developers with precious high-level feedback in the form of ranked suggestions.

The dominant approach in this area consists in using imitation learning on large corpora of formal proofs [6, 7, 8, 9, 10]. The idea is to train a large neural network to map proof goals to probability distributions over relevant tactics by mimicking human-produced proof scripts. The output of such a neural network can then be used as a form of recommendation system for human developers or plugged as a heuristic in a tree-search procedure for full automation.

However, despite impressive progress involving large language models [11] and self-supervised pre-training [10], this approach is still limited by the amount of available human-produced training data [12]. Indeed, not only is the total amount of available formal proofs already really small on the scale at which deep learning operates, but the amount of available proofs that are most relevant to each specific area (e.g. proving the correctness of a compiler transformation, conducting geometric reasoning for robotic planning...) is even more limited.

An alternative approach inspired by the success of AlphaZero [13] is to use reinforcement learning and let an agent self-train to interact with a theorem prover via trial and error [14, 15]. However, previous attempts of doing so have been hampered by two fundamental issues:

- First, existing tactic-based theorem provers offer infinite action spaces that are hardly amenable to random exploration. Also, they are optimized for formalizing the *outcome* of human insights concisely but often fail to define a tractable search space for deriving those insights in the first place [16]. For example, using tactics effectively often requires providing insights in advance that are more easily found later in the search process (e.g. constructing an object to instantiate an existential quantifier [17]). More generally, most of the human process of discovering proofs, which involves trial and error, sketching and abductive reasoning is not captured by standard prover tactics and so we can hardly

```
assume x >= 1;
y = 0;
while (y < 1000) {
  x = x + y;
  y = y + 1;
}
assert x >= y;
```

A desirable *invariant* is a formula $I[x, y]$ such that:

$$\forall x, y \begin{cases} x \geq 1 \wedge y = 0 \rightarrow I[x, y] & \text{(holds initially)} \\ y < 1000 \wedge I[x, y] \rightarrow I[x + y, y + 1] & \text{(preserved)} \\ y \geq 1000 \wedge I[x, y] \rightarrow x \geq y & \text{(implies post)} \end{cases}$$

Such an invariant is $I[x, y] := (x \geq y \wedge x \geq 1 \wedge y \geq 0)$

Figure 2: An example program and an associated loop invariant

expect a reinforcement learning agent to learn this process through sheer interaction with a tactic-based prover.

- Second, although reinforcement learning alleviates the need for human proofs during training, the agent must still be provided learning tasks of suitable relevance, diversity and difficulty. Existing work uses human written theorem statements for this purpose but doing so only shifts the problem one level [14, 18, 12]. This is in contrast with applications of AlphaZero in board games, where the symmetric nature of games such as Chess or Go enables leveraging symmetric self-play as an infinite source of training data.

In this proposal, we suggest a novel approach to learning theorem proving that does not rely on human-provided proofs and theorems. Instead, a teacher agent is trained to generate interesting and relevant tasks while a solver agent is co-trained to solve them. Both agents can leverage domain-specific expert strategies in the form of nondeterministic programs for building proofs and theorems. Choice points in those programs are resolved by neural network oracles that are trained via reinforcement learning in a purely self-supervised fashion. This allows leveraging minimal amounts of domain knowledge to tackle problems for which training data is unavailable and hard to synthesize.

2 Completed Work: An Invariant Synthesis Use-Case

In this section, we introduce and demonstrate our framework in a well-contained yet challenging setting, namely the verification of single-loop imperative programs. However, none of our core contributions are specific to this scenario.

2.1 Background: Verifying Imperative Programs with Loops

Suppose we are given a program such as the one in Figure 2 and we want to prove that the final assertion always holds. The way to proceed is to find a predicate called a *loop invariant* with the following properties: *i)* it is true before the loop, *ii)* it is preserved by the loop body when the loop guard holds and *iii)* it implies the final assertion when the loop guard does not hold. By "preserved", we mean that if I is true before executing the loop body then it is also true afterwards. Finding loop invariants is the most crucial aspect of program verification [19, 20] and still resists automation [21].

Despite the difficulty of automatically synthesizing loop invariants, humans do so routinely using well-understood search strategies. For example, in the case of the example in Figure 2, one would first try to prove the postcondition $x \geq y$ itself as an invariant and then note that it is not preserved by the loop body as the following implication does not hold: $y < 1000 \wedge x \geq y \not\rightarrow x + y \geq y + 1$. However, simplifying the right-hand side, we see that the proof works if we can show $x \geq 1$ to be an invariant itself. Using a similar form of abductive reasoning, we find that this in turn requires $y \geq 0$ to be an invariant and we end up proposing $x \geq y \wedge x \geq 1 \wedge y \geq 0$ as a loop invariant.

2.2 Approach: Self-Supervised Refining of Proof Search Strategies

2.2.1 Expressing Search Strategies as Nondeterministic Programs

The search strategy we followed for the example above can be formalized as a nondeterministic program, which is shown in Figure 3. As a nondeterministic program, it features a `choose` operator

```
1  def solver(
2        init: Formula, guard: Formula,
3        body: Program, post: Formula) -> Formula:
4
5     def prove_inv(inv: Formula) -> List[Formula]:
6        assert valid(Implies(init, inv))
7        inductive = Implies(And(guard, inv), wlp(body, inv))
8        match abduct(inductive):
9           case Valid:
10             return [inv]
11          case [*suggestions]:
12             aux = choose(suggestions)
13             return [inv] + prove_inv(aux)
14
15    inv_cand = choose(abduct(Implies(Not(guard), post)))
16    inv_conjuncts = prove_inv(inv_cand)
17    reward(max(-1, -0.2 * len(inv_conjuncts))
18    return And(*inv_conjuncts)
```

Figure 3: A simple strategy for finding loop invariants, described in Python-like pseudocode syntax. In this strategy, an initial invariant candidate is selected nondeterministically that implies the postcondition (line 15). One ensures that this candidate holds initially (line 6), without which the strategy fails immediately. Then, one attempts to prove that it is preserved by the loop body (lines 7 and 8, where `wlp` denotes Hoare's weakest liberal precondition operator [22]). If the candidate is not preserved, the `abduct` function suggests a list of assumptions that make it so. One then uses the `choose` operator to nondeterministically select one of them, which we try and prove invariant recursively (line 13). Because the number of abduction candidates to choose from can be large, successfully using such a strategy depends on having an effective oracle to guide search. To provide sufficient context to such an oracle, calls to `choose` must provide some extra information that is omitted here for brevity. In this case, we would pass a special token to indicate the call site, the program being analyzed and the value of `inv` in the case of line 12. See Appendix A.1 for more details and for a full listing of the invariant synthesis strategy that we use in our experiments.

that takes a list of objects as an input and selects one of them nondeterministically. A nondeterministic program can be refined into a deterministic one by providing an external oracle to implement the `choose` operator. It also defines a search tree that can be explored with or without a guiding heuristic. In this work, we use neural networks to implement such oracles and guiding heuristics.

In addition, a key aspect of this strategy is to infer missing assumptions under which a currently failing proof obligation would hold. This form of reasoning is called *abductive reasoning* and it is fundamental in the way humans search for proofs. It is implemented in a separate `abduct` procedure that takes a formula as an input and returns either `Valid` or a list of abduction candidates. For example, the `abduct` procedure fails to prove the implication $x \geq 0 \rightarrow x + y \geq 1$ but may suggest $x + y \geq 1$, $x < 0$ and $y \geq 1$ as possible missing assumptions. The use of abductive reasoning for theorem proving and loop invariant synthesis specifically has been proposed in the past [23, 24]. However, abduction procedures are hard to implement [25] and typically only available for specific decidable theories. Also, using them in proof search tends to scale poorly in the absence of good heuristics to rank and filter abduction candidates. Our proposed framework makes abductive reasoning practical by addressing both issues: it allows leveraging self-learned guidance to select abduction candidates and it allows implementing abduction procedures as nondeterministic programs that are amenable to learning themselves.

Another notable feature of the strategy in Figure 3 is the use of the `reward` operator on line 17 to incentivize finding short invariants (short proofs are desirable in general as they tend to be easier to check, interpret and generalize). The `reward` operator can be used multiple times and at any point of the strategy execution. An implicit reward of 1 or -1 is emitted when the strategy successfully returns or fails respectively. An optimal execution of a nondeterministic strategy is one that maximizes the cumulative amount of collected rewards. In this example, we bound the maximal invariant size penalty in such a way to ensure that finding a proof is always rewarded more than failing at doing so.

As argued by Selsam [16], defining search strategies as nondeterministic programs provides a natural and flexible way for experts to leverage domain-specific knowledge while outsourcing difficult proof decisions to search algorithms and learned heuristics. This paradigm differs from the standard paradigm of tactic-based theorem provers in which an external entity must orchestrate stateless tactics that do not call for any form of user interaction internally. In contrast, the nondeterministic programming approach *inverts* control and has strategies call external oracles rather than the other way around, which allows for a much tighter control of the resulting search space.

2.2.2 Refining Nondeterministic Strategies with Reinforcement Learning

A nondeterministic program induces a (deterministic) Markov Decision Process (MDP) where intermediate states are choice points and final states are either success states (the program returns successfully) or failure states (an assertion is violated or `choose` is called on an empty list of choices). Standard search algorithms can be used to navigate this MDP but doing so in an efficient and scalable way requires strong heuristics for guiding search.

In this work, we propose to learn such heuristics in a self-supervised fashion using the AlphaZero algorithm [13, 26]. In AlphaZero, a neural network is trained to map any state to a probability distribution over available actions along with a value estimate (i.e. an estimate of the expected sum of future rewards). The neural network alone can be used as a policy for navigating the MDP. However, a stronger policy results from combining the network with a tree search algorithm such as Monte-Carlo Tree Search (MCTS) [27]. The key insight underlying AlphaZero is that the network can be trained via an iterative improvement process where it is successively *i)* used as an MCTS heuristic to try and solve problems and then *ii)* updated using gradient descent so as to better predict the outcome of each attempt along with the action selected by MCTS on all states encountered along the way. More details on AlphaZero can be found in the literature [13].

Using AlphaZero, we can refine nondeterministic proof search strategies *without* external proof examples to learn from. However, AlphaZero must still be provided with a set of training problems to be solved. Training problems should be diverse, relevant and numerous enough to allow proper generalization. They should also be of varied difficulty to allow for learning to bootstrap.

2.2.3 Generating Training Problems with Conditional Generative Strategies

In theorem proving, high-quality training problems produced by humans are often not available in quantities even remotely matching the needs of reinforcement learning to properly generalize across problem instances. A possible solution is to use procedural generation techniques to assemble large datasets of synthetic problems. This works well in some specific areas and particularly for problems that can be framed as inverse problems such as symbolic integration [28]. In other areas, generating interesting problems is as hard as solving existing ones, possibly harder.

This is the case in particular with the problem of loop invariant synthesis. Here, a natural idea for generating problem instances would be to use a probabilistic grammar for repeatedly sampling triples consisting of a program, a loop invariant and an assertion and then reject all triples in which the properties defining valid invariants do not hold. Unfortunately, not only would doing so naively lead to a very high rejection rate, but the resulting dataset would be heavily biased towards trivial samples that are hardest to reject (e.g. programs where the final assertion is the negation of the loop guard and where the invariant does not even matter). In contrast, many classes of interesting problems from standard human-written benchmarks would only be sampled with an infinitesimal probability.

In this work, we propose to generate problems using the same methodology we use to solve them: by leveraging reinforcement learning to refine nondeterministic search strategies.

Specifically, experts can define *teacher strategies* in the form of stochastic and nondeterministic programs, each run of which either fails or successfully returns with a problem instance. Reinforcement learning can then be used to refine such programs with the two objectives of maximizing the diversity and interestingness of generated problems while avoiding rejection. However, these objectives are naturally in tension since an agent may be locally incentivized to avoid generating particular types of problems that are easier to reject. We introduce a design template for a class of strategies that avoid this obstacle. We call such strategies *conditional generative strategies*.

A conditional generative strategy generates a problem in three steps: *i)* it samples a set of constraints that define desired features for the generated problem, *ii)* it nondeterministically generates a problem

```
1  def teacher(rng: RandGen) -> Prog:
2     cs = sample_constrs(rng)
3     p = generate_prog(cs)
4     return transform(p, rng)
5
6  def generate_prog(cs: Constrs):
7     p = Prog("
8        assume init;
9        while (guard) {
10         invariant inv_lin;
11         invariant inv_aux;
12         invariant inv_main;
13         body;
14       }
15       assert post;")
16    p = refine_guard(p, cs)
17    p = refine_inv(p, cs)
18    p = refine_body(p, cs)
19    assert valid(inv_preserved(p))
20    p = refine_post(p, cs)
21    assert valid(inv_post(p))
22    p = refine_init(p, cs)
23    assert valid(inv_init(p))
24    nv = num_violations(p, cs)
25    reward(max(-1.5, -0.5 * nv))
26    return p
27
28 def transform(p: Prog, rng: RandGen):
29    p = shuffle_formulas(p, rng)
30    p = add_useless_init(p, rng)
31    p = add_useless_post(p, rng)
32    ...
33    p = hide_invariants(p, rng)
34    return p
```

Figure 4: Simplified teacher strategy. Problems are generated by nondeterministically refining the template in lines 8-15 in a way to optimize random constraints (see examples in Table 1). In this template, the invariant is expressed as a conjunction of at most three sub-invariants: `inv_lin` is a linear equality or inequality, `inv_main` is a disjunction of atomic formulas (i.e. comparisons) and `inv_aux` is a conjunction of atomic formulas that can be used in proving `inv_main` but not `post`. The `num_violations` function also penalizes the presence of useless or redundant problem components. For example, a penalty is applied if `inv_main` features a disjunct that can be removed without invalidating the problem. Appendix A.2 provides more details on how the `refine_*` functions can be implemented. The simplest way to do so is to have a grammar and use the `choose` operator to select rules recursively. However, fancier strategies can easily be implemented in the nondeterministic programming framework. In particular, our implementation uses the `abduct` procedure introduced in Figure 3 to suggest values for constants and subformulas.

Name	**Type**	**Description**
`num-inv-main-disjuncts`	`none|1|2`	If an integer n, then `inv_main` is refined with a disjunction of n atomic formulas.
`has-conditional`	`bool`	Whether `body` must include a conditional statement.
`loop-guard-useful-for-post`	`bool`	Whether assuming the negation of the loop guard is useful in proving the postcondition `post`.
`body-implies-main-inv`	`bool`	If `true`, then `inv_main` always holds after executing `body` regardless of whether or not it holds before.
`eq-only-for-init`	`bool`	Whether or not to use equalities only in `init`.

Table 1: Examples of teacher constraints. Some descriptions refer to names introduced in Figure 4. The full list of constraints we used in our experiments is available in Appendix A.2.

that respects as many constraints as possible and gets rewarded for doing so and *iii)* it applies a sequence of random and validity-preserving problem transformations as a way to further increase diversity. We provide a high-level description of a such a teacher strategy for invariant synthesis in Figure 4. The interest of such an architecture is that it enables decoupling the two conflicting objectives of generating valid and diverse problems: diversity is guaranteed by the first and third steps while learning can focus on the second step. This also allows using the exact same learning algorithm for training both teacher and solver agents (AlphaZero in this work).

2.2.4 Predicting Events Rather Than Rewards

In the standard AlphaZero setting, the network is trained to predict the value of encountered states, that is the expected sum of future rewards. However, such a number conflates a lot of valuable information. For example, suppose a teacher state is assigned a value of 0. Does 0 mean that the teacher is predicted to either fail or succeed with equal probability or that it will certainly succeed in generating a problem that violates two constraints? And if so, which two?

Although this information is not relevant to MCTS, there are several reasons for having the network predict it anyway. First, doing so can result in an increased sample efficiency by providing the network with more detailed feedback on its mistakes. Such an effect has been previously demonstrated in the KataGo project [18]. Second, it is interesting from an interpretability point of view and makes it easier to diagnose problems and weaknesses in a given network. We found a third reason that is more subtle, which has to do with getting the combined benefits of issuing intermediate and final rewards.

Indeed, in the teacher strategy showed in Figure 4, we only penalize the agent for violating a constraint *at the very end*. Intuitively, there is a lost opportunity here since many violations could be detected much earlier while the problem is still being refined. Emitting an intermediate reward at this point would certainly improve learning efficiency. However, the same violation can be potentially detected at several different points in time and we must ensure that a reward is only issued the first time. This in turns makes the job of the value prediction network harder since it needs to keep track of whether or not each violation has been penalized already. Encoding this information alone can be costly, and especially so for attention-based network architectures such as transformers with a quadratic inference cost. Having the network predict constraint violations separately offers an elegant way out: all rewards are issued at the end but from the moment a constraint violation is detected, the network's predicted probability for this violation is overridden to 1 whenever a value estimate is computed.

More formally, we introduce the concept of an *event*. Strategies can declare an arbitrary number of events e and each one is associated with a reward r_e along with a maximal number of occurrences m_e ($m_e = 1$ for constraint violation events) that can be counted towards the final reward. The `reward` operator introduced in Figure 3 is replaced by an `event` operator. When a strategy terminates, a reward is issued implicitly with a value of -1 in case of a failure and

$$\max\left\{1 + \sum_e r_e \min(n_e, m_e),\ r_{\texttt{min}}\right\}$$

in case of a success. In this expression, n_e denotes the number of calls made to $\texttt{event}(e)$ during the whole episode and setting $r_{\texttt{min}} > -1$ guarantees that successes are always rewarded more than failures. Note that it is important not to issue event penalties after a failure. Otherwise, faced with a likely failure, an agent may be incentivized to simply give up and fail immediately to avoid further penalties in search of success. Also, there are two reasons for bounding the maximal number of occurrences of event e that are counted towards the final reward by a constant quantity m_e. First, this allows having a network with a finite softmax head predict the number of occurrences of e (as we see below). Second, the $m_e = 1$ case is particularly useful to model events such as constraint violations that can be detected multiple times but should only be penalized once.

In this framework, the value head of the network is tasked with predicting the following probabilities: *i)* the probability p_0 of a failure, *ii)* the probability p_1 of succeeding with minimal reward $r_{\texttt{min}}$, *iii)* the probability $p_2 = 1 - p_0 - p_1$ of succeeding with a greater reward and *iv)* the probabilities p_e^i that $\min(n_e, m_e) = i$ for all events e and $0 \leq i \leq m_e$. A value estimate derives from these probabilities:

$$-p_0 + p_1 \cdot r_{\texttt{min}} + p_2 \cdot \sum_e \sum_i \hat{p}_e^i \cdot i \cdot r_e$$

where $\hat{p}_e^i \propto \mathbf{1}\{n_e \leq i\} \cdot p_e^i$ is a corrected probability estimate accounting for the number of times n_e that $\texttt{event}(e)$ was raised already. In our invariant synthesis experiments, we use events in both the teacher and solver strategies to represent constraint violations ($m_e = 1$) and penalize proof steps respectively ($m_e > 1$).

2.3 Implementation and Engineering Contributions

This proposal advocates for a hybrid approach to automated theorem proving where human experts from a variety of areas can formalize their knowledge succinctly in the form of nondeterministic

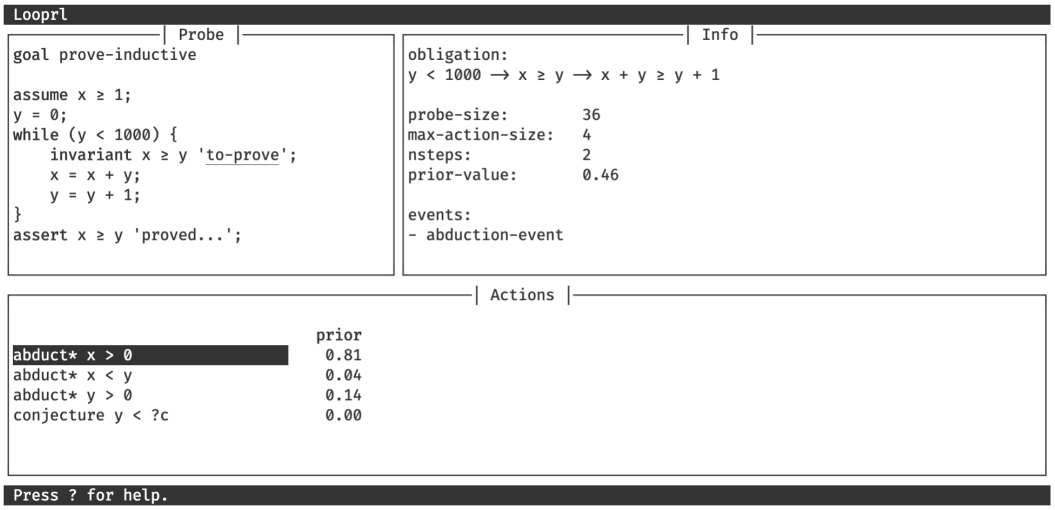


Figure 5: Using the Looprl UI to inspect the solver agent on the problem from Figure 2. Additional commented screenshots are available in Appendix A.5.

proving and teaching strategies. Reinforcement learning is leveraged to fill in the blanks in all inevitable cases where the human intuition escapes formalization.

However, for this vision to be practical, writing strategies and iterating on them must be as frictionless as possible. New tools and abstractions are needed to automate the process of converting nondeterministic programs into reinforcement learning environments, ease debugging and foster code reuse. In this work, we accomplish a crucial step in this direction by introducing the Looprl theorem prover. Looprl consists of the following collection of features:

- A domain specific language for writing strategies embedded in OCaml, with first-class support for neural-guided nondeterministic programming. Strategies are automatically compiled into reinforcement learning environments that can be explored in Python.
- A graphical interface that can be used to interact with strategies manually while inspecting the neural network's predictions and the behavior of MCTS (see Figure 5).
- A library of utility functions for writing program verification strategies. This library includes an implementation of the `abduct` function introduced in Figure 3 for integer linear arithmetic (see details in Appendix A.6).
- A parallel, high-performance implementation of the (Gumbel) AlphaZero algorithm [13, 26] for refining search strategies. Our implementation uses Pytorch [29] and Ray [30]. It provides support for events (see section 2.2.4) and unbounded, variable-size action spaces.

The first point on developing a domain-specific language for writing strategies is particularly important and deserves more discussion. In our experience, having such a language is not just a mere matter of convenience but one of feasibility. In fact, we did try to implement an initial version of a teacher strategy for loop invariant synthesis by writing pseudocode on paper and manually compiling it into an MDP implemented in Python. However, doing so resulted in a complexity explosion where any single-line change to the pseudocode could take days of work to implement.

The reason compiling a nondeterministic program into an MDP manually is so tedious is that the whole program state must be made explicit, which includes the program stack. A tempting alternative would be to write nondeterministic code as normal Python code parametrized by an arbitrary `choose` function. However, this does not allow using tree search on the resulting program since doing so would require cloning the whole execution state of a Python program. Fortunately, there exists a solution to this problem in the programming languages community, which involves a construct called a *search monad* [31, 16] and which essentially enables writing arbitrary nondeterministic code and then reifying it into a search tree that can be manipulated explicitly.

In Looprl, we implement a domain specific language embedded in OCaml for expressing nondeterministic strategies. OCaml is a natural choice because it is fast (about two orders of magnitude faster than Python for symbolic code such as proof inferences) and it has good support for monads and Python interoperability. Our language provides built-in support for the `choose` and `event` operators. Also, it defines an intermediate graph-based format for representing data to be sent to neural networks in an architecture-agnostic way. Any piece of information that is passed to the `choose` operator must be convertible to this format and feature suitable metadata ensuring a seamless integration with the Looprl user interface and debugging tools. Looprl consists of about 15K lines of code written in OCaml and Python. It is available on Github.

2.4 Experiments

We tested Looprl on the problem of synthesizing loop invariants for single-loop imperative programs and evaluated it on the standard Code2Inv [32] benchmark. This benchmark contains a set of 132 programs written in C. All programs feature a single loop along with a final assertion to be proven. They feature linear integer arithmetic and sometimes involve conditionals, assumptions and nondeterministic tests. We provide examples of Code2Inv problems in Appendix A.3. Most existing invariant generation tools can only solve a subset of these problems [33]. To the best of our knowledge, only one existing tool can solve them all [34], but only by using specific optimization techniques that are not generalizable beyond the setting of purely numerical programs.

We used Looprl to implement an invariant generation strategy, which we describe in Appendix A.1. In a nutshell, it generalizes the simple strategy described in Figure 3 by adding features such as the ability to abduct disjunctive invariants or to conjecture invariant templates with placeholder constants to be refined later using abduction. To our surprise and although this generic strategy can be described in only one page of pseudocode, it is sufficient to simply solve *every* Code2Inv benchmark problem in a few seconds when combined with the vanilla MCTS algorithm [27] (with no learned heuristic).

We therefore evaluate a self-trained solver agent on its ability to solve as many Code2Inv problems as possible *without* resorting to any search. At every decision point, it can only take the highest ranked action according to the network's policy head and no backtracking is allowed. We argue that such a metric is the most skeptical measure for the quality of the resulting learning. Indeed, finding an invariant for a single loop is of limited interest of its own. Rather, doing so is typically useful as a subtask in a hierarchy of increasingly complex and relevant problems. The next level of this hierarchy may be to prove a piece of code with nested loops, and then to synthesize a function respecting a specification, which is still several levels away from writing full-fledged software systems. Complex problems cannot be solved if backtracking search is needed at every level of this deep hierarchy.

2.4.1 Training Protocol

We train a teacher agent and a solver agent in sequence using the (Gumbel) AlphaZero algorithm. Both agents use a similar 1.6M parameters Dynamic Graph Transformer network [35] (see architecture details in Appendix A.8). The teacher agent is trained first for 20 AlphaZero iterations. During each iteration, it goes through 8000 problem generation episodes, using 64 MCTS simulations per step. Then, the network is updated using samples from the k previous iterations with k evolving from 1 to 6 during training. Each iteration simulates 800 additional episodes to generate *validation* samples that are not used to train the network but to control overfitting via early stopping. After the teacher agent is trained, a dataset of 50K problems is gathered for training the solver, which includes a mix of 40K problems generated during the last five training iterations and 10K new ones that are generated without Gumbel exploration noise [26]. The solver agent is also trained for 20 iterations. During each iteration, it attempts to solve 20K randomly selected problems using 32 MCTS simulations per step. In addition, 5K additional problems are solved to generate validation samples. See Appendix A.7 for a complete list and explanation of all training hyperparameters.

A complete training run takes about 16h on a single machine with a 10-core Intel i9-10900KF processor, 64GB of RAM and a NVIDIA GeForce RTX 3080 GPU.

2.4.2 Experimental Results

We show training curves for the solver and teacher agents in Figures 6 and 7 respectively. These curves record the evolution of the average reward collected by MCTS combined with the latest

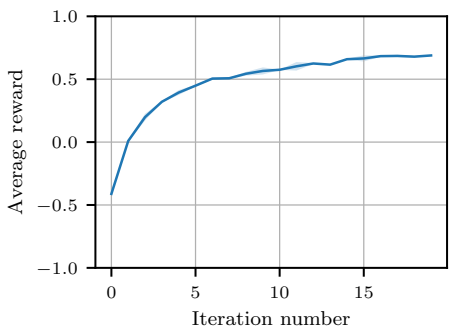


Figure 6: Teacher training curve

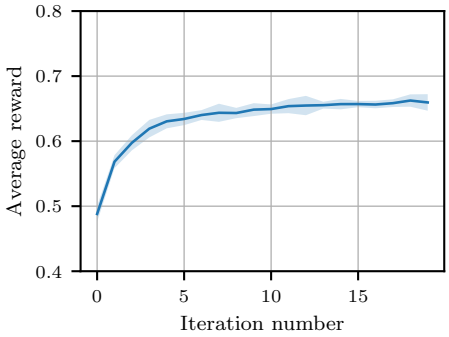


Figure 7: Solver training curve

Policy	% Problems solved
Random	18.4 ± 0.0
Network (untrained teacher)	39.7 ± 1.6
Network (trained teacher)	$\mathbf{61.5 \pm 0.4}$

Table 2: Experimental results on the Code2Inv benchmark (using no backtracking search). The score for the Random heuristic is computed as an average across 10K attempts. Standard deviations are computed based on using two different random seeds for the full training experiment.

network during each iteration. Error intervals are estimated based on two random seeds. Note that the maximum theoretical reward obtainable by the teacher agent is <1 since problem generation constraints may be sampled that are mutually incompatible, in which case the agent simply does its best to minimize the number of violated constraints. The same holds for the solver agent since all generated invariants are penalized proportionally to their size. The solver agent goes through a more modest reward gain during training (the y-axis is rescaled in Figure 7 to emphasize the trend). This is reflective of the fact that most problems generated by the teacher can be solved by MCTS alone and so most of the training concentrates on learning to solve about 10% of hard problems.

We show the results of our evaluation on the Code2Inv benchmark in Table 2. Three different agents are compared on the average ratio of benchmark problems for which they can successfully generate an invariant of minimal size *without* search or backtracking. The first agent is a baseline that simply selects proof actions at random. The second agent is a solver network that was trained using problems generated by an untrained teacher (i.e. using MCTS but no network heuristic). Finally, the third agent is a solver network trained using the full protocol described in section 2.4.1. Unsurprisingly, an inferior teacher leads to an inferior solver with decreased generalization capabilities.

2.5 Summary of Contributions

Our completed work demonstrates our framework on the problem of loop invariant synthesis. Our agent solves all Code2Inv challenges. More importantly, it learns to solve a majority of problems with no search at all despite never seeing these problems during training. None of our core contributions, however, are specific to invariant synthesis. These include: *i)* our key insight that *nondeterministic programming* and *reinforcement learning* can be similarly combined to implement solvers and teachers, *ii) conditional generative strategies* as a general template to write teachers, *iii) abductive reasoning* as a design principle that is made scalable by self-learned guidance, *iv) strategy events* for easier reward engineering and better sample-efficiency and *v)* an implementation that establishes engineering foundations for making the whole approach practical.

3 Proposed Work

This proposal pursues a vision where general theorem provers allow users to write teacher and solver strategies for various domains of mathematics and programming in a modular and distributed fashion. A large pretrained neural network can then be fined-tuned to serve as an oracle for all these strategies, allowing better generalization and sample efficiency. However, many challenges lie ahead. This section identifies some of these challenges and proposes extensions to our current framework that, we argue, have the potential to make it practical on increasingly complex and diverse problems. We organize these extensions according to three separate goals: implementing more effective teachers that foster the discovery of diverse and interesting problems intrinsically (Section 3.1.1), providing a richer language for writing nondeterministic search strategies (Section 3.1.2) and integrating our framework with large pretrained language models (Section 3.1.3). We also propose to evaluate our framework and implementation on a series of domains (Section 3.2) such as program verification, deductive program synthesis, real inequality proving and robot planning. Finally, we propose to build a high-quality implementation of our framework to be used as a basis for further research (Section 3.3)

3.1 Extension Proposals

3.1.1 Toward Teachers with Intrinsic Diversity and Relevance Mechanisms

Writing conditional generative strategies (Section 2.2.3) is engineering-intensive since diversity is enforced extrinsically via manually-defined constraints. Indeed, the teacher agent has no incentive to generate different problems for any given combination of constraints. Ensuring diversity therefore requires a large number of constraints, even in the simple setting of our preliminary use-case (Section 2). This not only creates an engineering burden but also increases the amount of contextual information that the network has to process and reason about. Moreover, many constraint combinations are not going to result in problems that are suitably interesting and difficult for a given solver agent, further reducing learning efficiency. We propose two distinct research ideas to address these issues, which we develop in the rest of this section.

3.1.1.1 Using Solver Feedback as a Source of Intrinsic Rewards

In addition to the teacher being rewarded for generating problems that match sampled constraints, we propose having the solver reward the teacher directly for producing interesting problems. There are several ways to define *"interesting"* in this context. For example, we can call a problem interesting if it surprises the current solver in the sense that the MCTS policy significantly differs from the network policy on this problem. Alternatively, we can call a problem interesting if the current solver can solve it but only at the price of a significant amount of search, guaranteeing that the problem is neither too easy nor too hard [36]. A diagram of our proposed teacher/solver architecture is shown in Figure 8.

One advantage of this architecture is that it creates an implicit curriculum for the solver by encouraging problems of suitable difficulty to be generated at every step. Moreover, the generated problems can be kept in a buffer to avoid forgetting and such a buffer will naturally end up with a diversity of interesting problems, an interesting problem being defined as a problem that was once a useful learning challenge for the solver. On the other hand, training such an architecture may require extra careful tuning since it creates an additional interdependency between the teacher and the solver. Moreover, it requires updating the teacher and solver networks at a higher frequency than previously needed, which might raise the kind of sensitivity and instability issues that are common in reinforcement learning when neural networks are made to fit fast-moving targets [37].

3.1.1.2 Evolving a Population of Problems

The teacher agents we have been considering so far generate problems independently from each other: each run of a teacher strategy attempts to generate a single problem from scratch. As an alternative, we can imagine a teacher agent maintaining a population of problems that is progressively expanded via successive mutations.

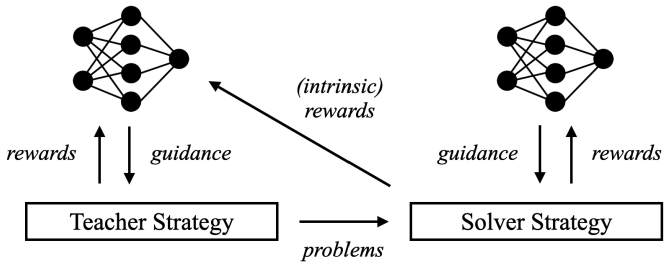


Figure 8: A teacher/solver architecture where the solver provides an intrinsic reward signal to the teacher. Note that this does not exclude the teacher strategy from defining an additional extrinsic reward signal itself. Also, the teacher and solver networks may share some or all their weights.

For example, at each iteration, one may pick a problem from the population and sample a set of random constraints. The teacher is then tasked to perform a minimal amount of edits to the problem in such a way as to produce a variation that respects as many constraints as possible. The associated proof must also be repaired or regenerated to ensure validity. When a valid problem is generated, it is added to the population where it can be used for training the solver and selected for further mutation.

In our invariant synthesis use-case, mutation constraints may say something like "generate a variation of the selected program where $n > 0$ is not assumed anymore; the only allowed editing operations are to add assignments to x and y in the loop body and change the loop invariant". As an addition or replacement to mutation constraints, problems can also be scored and rewarded by the solver as suggested in Section 3.1.1.1. The resulting interestingness scores can further be used as a selection mechanism to filter problems to be added to the population and bias their sampling. The population can be either initialized by a handful of seed problems provided by experts or using a vanilla conditional generative strategy (Section 2.2.3).

This evolutionary approach to problem generation has several potential advantages. First, it adds an intrinsic source of diversity by having the teacher strategy run from a growing variety of initial states. Second, experts are given a particularly intuitive proxy for biasing the teacher process by providing a small number of seed problems. Finally, assuming interestingness scores are easily computable, favoring the mutation of an interesting problem provides a more reliable way to ensure that similar problems will be generated in the future than simply giving the teacher a one-time reward.

3.1.2 Toward a Richer Language for Designing Strategies

A key aspect of our framework is to provide domain experts with an expressive language for writing teacher and solver strategies in the form of self-refining nondterministic programs. In our completed work, we introduced such a language that is built on the `choose` and `event` operators. In this section, we propose additional language constructs that we expect to be useful for scaling up our framework. Careful thought must be put into how these features interact with search, learning and with each other.

3.1.2.1 Leveraging Alternate Forms of Neural Guidance

The `choose` operator provides a simple and powerful abstraction but it is not best suited to implement all forms of neural proof guidance. For example, a common pattern for integrating machine-learning with automated theorem proving is to have an oracle estimate the relevance of different assumptions for proving the current proof goal [38, 39]. Doing so with `choose` is possible in theory but raises several practical issues. First, providing relevance hints for n assumptions requires either n calls to `choose` or a single call with 2^n arguments, which is wasteful in terms of network inference. Second, one may not necessarily wish tree search to branch on all 2^n possibilities since getting every hint right is usually not a requirement for successfully finding a proof. Finally, making unambiguous references to elements of a proof goal in separate self-contained expressions requires using explicit labels or indexes that are inconvenient to manipulate for both humans and neural networks [40, 41].

A more efficient way to obtain relevance hints is to have the neural network annotate the proof goal directly by attaching probabilistic tags to selected tokens of its own input (Figure 9). Standard

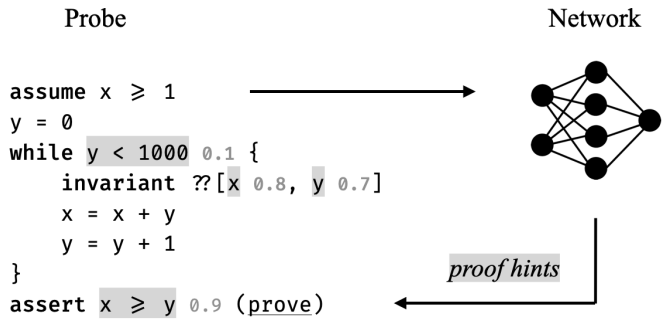


Figure 9: An example of having a neural network provide probabilistic relevance hints by annotating the current proof goal. Here, the network indicates that the loop guard is likely irrelevant in proving the postcondition when assuming the invariant (with probability 0.9) and that the invariant must probably involve both variables x and y. Other types of annotations could be considered. For example, the network may also annotate variable x in the invariant placeholder to indicate that this variable should not be upper bounded in the invariant with probability 0.8.

network architectures such as Transformers [42] or Graph Neural Networks [43] can do so naturally since a probability distribution over possible annotations can be extracted for each input token from its final encoder state.

We propose augmenting our strategy language with an `annotate` operator that takes a structured, tokenizable object as an input and attaches probabilistic tags to a pre-specified subset of its parts. Probabilistic tags can be used directly (Figure 10) or turned into non-probabilistic tags via branching (Figure 11). Explicit branching is allowed by a `branch` operator that acts like `choose` except that it does not query an external oracle and receives an explicit probability distribution over choices instead. To enable learning, strategies can specify hindsight ground truth values for annotations using a special `feedback` method (Figures 10 and 11).

Although our examples have focused on relevance hints, annotations can be used to provide more fine-grained guidance on where and how different parts of a proof goal must be used. Strategy writers can specify custom annotation types by declaring a list of eligible values along with a flag indicating whether tag probabilities must be normalized across values, locations or both. Normalizing across locations is particularly useful for implementing *pointers* [40, 44]. More generally, the existence of interactive theorem provers with point-and-click interfaces [41] suggests that the `annotate` operator provides a general and flexible mechanism for proof guidance.

Another construct that we consider adding to our strategy language is a `guess` operator that allows generating an arbitrary parsable object of a given type using an autoregressive model. Indeed, generating complex expressions using `choose` can be engineering intensive and most recent work in applying machine-learning to interactive theorem proving successfully uses Transformers [42] to output tactics token by token. Of course, an obvious tradeoff is that such models are harder to use with reinforcement learning since the space of all strings is not amenable to exploration. Still, Figure 12 provides a use-case for using such an operator within our framework without relying on external supervised data.

3.1.2.2 Leveraging Modularity in Proofs

An important way in which humans solve complex problems is by recursively decomposing them into simpler, more manageable problems to be solved (mostly) independently [45]. This modularity is reflected in the tree structure of typical formal proofs and also in the basic paradigm of most interactive tactic-based provers where applying a tactic on a proof goal generates a sequence of subgoals (Figure 1). Yet, our current framework has limited support for such modularity. While it is possible to have a strategy identify a sub-problem and call a distinct sub-procedure to solve it, this procedure does not have an associated value function and all decision points within it must compute a value estimate that reflects the probability for the whole problem to be solved instead of the particular sub-problem at hand. This complicates the task of estimating values and requires passing contextual

```
1  RELEVANCE = declare_annot_type(values=[True, False], once=False)
2
3  def prove(fml: Formula):
4     fml = fml.copy().map_atoms(lambda a: Annotated([RELEVANCE], a))
5     annotate(fml)
6     annots = list(fml.annots(RELEVANCE))
7     annots.sort(key=lambda a: a.prob(True), reverse=True)
8     for a in annots: a.set(False)
9     for a in annots:
10       a.set(True)
11       smaller = fml.copy()
12       smaller.weaken_annotated({RELEVANCE: False})
13       if is_valid(smaller, timeout=1):
14          for a in annots: a.feedback()
15          return
16    assert False
```

Figure 10: An example of leveraging relevance annotations without branching. To prove a given formula, this function makes a sequence of increasingly large calls to an `is_valid` decision procedure by discarding all assumptions from the original problem and then bringing them back by decreasing order of estimated relevance. Doing so is useful since many decision procedures such as those based on quantifier elimination [41] are super-exponential with respect to input size. Line 1 declares a type for relevance annotations. The `once=False` argument indicates that relevance probabilities should not be normalized across locations, meaning that several assumptions can be deemed relevant with probability 1. Line 4 wraps all parts of the formula to be annotated in special `Annotated` objects. Executing line 5 attaches each of these wrappers one `Annotation` object per annotation type. An `Annotation` object features a distribution over tag values that is accessible via the `prob` method. It can also be assigned a deterministic value via the `set` method. The call on line 12 discards all assumptions that are deemed irrelevant via `set`. In case of success, all annotations are assigned a ground truth for learning on line 14 based on their current deterministic value.

```
1  def prove(fml: Formula):
2     fml.map_atoms(lambda a: Annotated([RELEVANCE], a))
3     branch_annotate(fml)
4     reward(sum(-0.1 for a in fml.annots(RELEVANCE) if a.value is True))
5     smaller = fml.copy().weaken_annotated({RELEVANCE: False})
6     assert is_valid(smaller)
7
8  def branch_annotate(obj, filter=(lambda c: True)):
9     annotate(obj)
10    values = [[(a, v) for v in a.type.values()] for a in obj.annots()]
11    combinations = [c for c in itertools.product(*values) if filter(c)]
12    likelihoods = [product(a.prob(v) for a, v in c) for c in combinations]
13    combination = branch(combinations, weights=likelihoods)
14    for a, v in combination:
15       a.set(v)
16       a.feedback()
```

Figure 11: An example of leveraging relevance annotations via branching. The `prove` function is a variation of the one defined in Figure 10. It relies on a standard `branch_annotate` function that uses the `annotate` and `branch` operators internally to assign deterministic tag values nondeterministically. Line 4 emits some rewards to encourage the finding of minimal sets of relevant assumptions (events could be used instead). Although this example generates an exponential number of branches, this behavior can be changed by providing a `filter` argument to `branch_annotate` in such a way to restrict the set of considered tag combinations. In this case, a possibility is to only consider the set of all combinations where only the k top-rated assumptions are deemed relevant (for all values of k).

```
def solve(x):
   sols, feedback = guess(parser=FORMULA, probe=..., beam=8)
   if (sol := choose(sols + [None])) is None:
      reward(-GUESS_FAILURE_PENALTY)
      sol = guided_solve(x)
   assert is_solution(sol)
   feedback.set(sol)
   return sol
```

Figure 12: An example use-case for the `guess` operator. When attempting to solve a problem or a sub-problem, the idea is to use the `guess` operator to first try and guess a few solutions directly with beam search. In case of a failure, the option remains to use a fallback procedure to generate a solution in a more guided fashion. In both cases, a valid solution can be turned into a data point for training the `guess` oracle. Once trained, such an oracle allows taking considerable shortcuts in the proof search process, making it easier to scale search to increasingly hard problems. Moreover, when using the same neural network for implementing both `guess` and `choose`, this use of the `guess` operator may improve sample efficiency in the same way that `event` typically does, by providing the network a richer training signal. Finally, providing an alternative guided resolution procedure may not even be needed in cases where some amount of supervised data is already available to initialize the `guess` oracle well enough for the combination of beam search and AlphaZero to bootstrap [12].

information to our sub-procedure that would normally not be needed. Finally, when a problem is decomposed into sub-problems, tree search currently does not have the opportunity to take advantage of this structure and explore these in parallel.

Existing work has studied adapting MCTS [46] and AlphaZero [12] to leverage the tree structure of proof scripts in tactic-based interactive provers. The main idea is to perform search on bipartite trees that feature both *or*-nodes (one child must be proved) and *and*-nodes (all children must be proved). Implementing this solution for the subset of our strategy language that can define the tactic automata [16] used in standard provers is easy. However, as this section is about to illustrate, tactic automata tend to define poor search spaces by forcing important proof insights to be provided well in advance of when they are naturally discovered. An interesting research question is whether the idea of modular subgoals can be reconciled with the full expressivity of our framework.

To investigate this question, let us consider augmenting our strategy language with a `subgoals` operator that takes as an argument a sequence of sub-procedure calls and turns them into subgoals with separate value functions to be pursued independently:

$$y_1, \ldots, y_n = \texttt{subgoals}(f_1(x_1), \ldots, f_n(x_n)).$$

After such an operator is added, strategies cannot be reified into standard search trees anymore. Instead, they are reified into tree structures with special nodes that we call *multi-nodes*. The instruction above is compiled into a multi-node that is defined by n children trees along with a delimited continuation that maps $(y_i)_i$ return values to a new search tree associated with the surrounding goal. Subgoals in this proposal are more general than those considered in previous work [46, 12] in the sense that *i)* they are allowed to return values, *ii)* they can issue rewards or events and *iii)* they can be associated with nontrivial delimited continuations. We call a continuation *nontrivial* if it can fail or issue rewards. All three aspects listed above raise new challenges. Restricting the use of the `subgoals` operator to rule them out gives us back the standard notion used in existing work.

For concreteness, we provide a simple example in Figure 13 that illustrates the potential usefulness of our generalized `subgoals` operator. This example proposes three strategies for proving a conjunction of the form $\exists x\,(P_1(x) \wedge P_2(x))$ with x a real variable: (a) shows the standard solution used by interactive theorem provers, (b) is a simple but misguided attempt at making it more search-friendly using generalized subgoals and (c) is our attempt to reconcile late quantifier instantiation with modularity.

Although Figure 13c provides an intuitive case for generalized subgoals, it remains to be defined how these should behave with respect to learning and search. A key question is how to estimate the value of multi-nodes with nontrivial continuations. One possibility here is to require the oracle to produce a value estimate $V_{\texttt{m}}$ for the proof state before subgoals are spawn and then assign the resulting multi-node a value of $\min(V_{\texttt{m}}, V_1 \times \cdots \times V_n)$ where the V_i are the subgoal values. Doing

```
def prove_ex_conj(P1, P2, x):
   v = guess_witness_somehow(P1, P2, x)
   subgoals(prove(subst(P1, x, v)), prove(subst(P2, x, v)))
```

(a) The standard solution of tactic-based provers. In order to prove the existential statement, a witness is provided to instantiate the quantifier and both conjuncts are proved independently afterwards. Equating values with success probabilities for simplicity, the value of the multi-node is equal to the product of the value of its subgoals. The weakness of this strategy is that the witness must be guessed *before* any proof of a conjunct is attempted, which an oracle can hardly do without figuring out the whole proof itself. The astute reader might point out that some interactive provers allow instantiating x with a metavariable to be refined later (e.g. via unification). However, doing so ties both subgoals together and makes it impossible to assign them separate value functions.

```
def prove_ex_conj(P1, P2, x):
   C1, C2 = subgoals(abduct(P1, x), abduct(P2, x))
   assert sat(And(C1, C2))
```

(b) A naive attempt at a modular search strategy. This strategy uses a nondeterministic `abduct` procedure that is different from the one used in section 2. The `abduct` procedure takes a formula P and a variable x as an input and returns a nontrivial constraint C on x that implies P. The multi-node in this example is associated with a nontrivial continuation. Writing V its value and V_1, V_2 the values of its subgoals, the $V = V_1V_2$ identity does not hold anymore and we only have $V \leq V_1V_2$ since the multi-node can fail even when its subgoals succeed. Putting aside this technical issue, this strategy is intuitively unsatisfactory since the oracle has no way to guarantee the compatibility of abducted constraints and no recourse in case of a failure.

```
def abduct(P, var, blacklist=[]):
   C = ...
   assert valid(Implies(C, P))
   for B in blacklist:
      assert not valid(Implies(B, C))
   return C

def prove_ex_conj(P1, P2, x, prev1=[], prev2=[]):
   C1, C2 = subgoals(abduct(P1, x, prev1), abduct(P2, x, prev2))
   if not satisfiable(And(C1, C2)):
      match choose(['keep1', 'keep2', 'drop']):
         case 'keep1':
            C2 = abduct(And(P2, C1), x, prev2)
         case 'keep2':
            C1 = abduct(And(P1, C2), x, prev1)
         case 'drop':
            return prove_ex_conj(P1, P2, x, prev1+[C1], prev2+[C2])
```

(c) A tentative modular strategy. This strategy improves the previous attempt by providing the oracle with some recourse in cases where incompatible constraints are generated. It uses an updated version of `abduct` that takes an extra `blacklist` argument for ruling out particular solutions. The main strategy starts as usual by calling `abduct` on P_1 and P_2 separately. In the event where the resulting constraints are not compatible, the strategy either chooses to keep one constraint and reattempt solving the other goal with this constraint added or to blacklist both constraints and retry from scratch.

Figure 13: Three strategies for proving $\exists x\ (P_1(x) \wedge P_2(x))$.

so requires passing an additional `probe` argument to the `subgoals` operator to estimate $V_{\mathtt{m}}$. Handling rewards and events raises other questions. For example, when a subgoal is successfully solved, should we pass its returned value to the continuation straight away or should we perform more search in case another solution can be found that is associated with a higher reward? Finally, the strategy in Figure 13c works best when the constraints returned by `abduct` are tight: it is suboptimal to return the $x > 1$ constraint when $x > 0$ would also do. However, encouraging such behavior via rewards is tricky since it is generally unclear whether or not a solution is suboptimal until a better one is found. Some ideas from the field of preference-based reinforcement learning [47] may be relevant here.

3.1.3 Toward an Integration with Large Language Models

In the last year, Transformer models [42] with up to hundred billions parameters have proved surprisingly successful on program synthesis [48, 49, 50] and reasoning tasks [51]. For example, the state-of-the-art on the MATH dataset [52] went from about 6% to more than 50% over this period [53]. The key of this success has been the pretraining of increasingly large networks on massive self-supervised corpora of text and code (e.g. the whole Github data along with all text that can be found on the internet), which has been qualified as a paradigm shift by many researchers [54].

We see our work as complementary to these advances since large pretrained language models can play a natural role in our framework as universal oracles that are fine-tuned via reinforcement learning. Moreover, despite impressive recent results, current techniques are still bottlenecked by the availability of human-produced data [50]. Our framework allows trading compute for data, which we expect to become increasingly relevant as the amount of available compute power continues to grow faster than the amount of available human data.

In this section, we discuss some proposals for exploring the use of large pretrained language models in our framework that do not require an unrealistic amount of compute.

3.1.3.1 Fine-Tuning Large Pretrained Language Models

Our completed work uses 1.6M parameter neural networks that are trained from scratch without pretraining. Through careful engineering, we managed to meaningfully train such models via pure reinforcement learning on commodity hardware (using a 10 cores CPU and an Nvidia RTX 3080 GPU). Realistically, we can get routine access to an order of magnitude more compute power and probably two for episodic experiments by applying to programs such as HAICORE [55].

We propose experimenting with the following networks and pretraining schemes:

- **Using the PLUR Framework.** The PLUR framework [56] features a collection of graph-based architectures for program learning, understanding and repair. These include the GREAT architecture [44] that we use in our completed work (Appendix A.8). The PLUR framework compiles supervised training data for 16 recently published tasks that we can use for pretraining. Typical networks trained with PLUR have about 20M parameters, which is still small enough for plugging such networks as-is in our framework. Also, their graph-based architecture offers a good inductive bias for the kind of symbolic tasks that we are interested in. However, the benefits of pretraining have been shown to strongly increase with scale [57] and so it is unclear whether the PLUR models are large enough and the associated datasets substantial and diverse enough for pretraining to be fully effective.
- **Using the PolyCoder Model.** PolyCoder [58] is a GPT-like Transformer model that has been trained on data from GitHub. It is available in three different sizes of 160M, 405M and 2.7G parameters. Although we expect the impact of pretraining to be much larger at this scale, integrating even the smallest PolyCoder model in our current framework would require an unrealistic amount of compute power given our resources. The next section suggests an idea for integrating such large models at a smaller computational price.

3.1.3.2 Accelerating Inference via Hierarchichal Oracles

The computational cost of training an AlphaZero agent is typically dominated by the cost of performing network inference when running MCTS to generate training data. For example, the original AlphaGo Zero uses 5000 TPUs for data generation but only 64 TPUs for network training. The main computational obstacle to using large language models with our framework is the cost of inference.

To alleviate this issue, we propose a hierarchical oracle architecture (Figure 14) where a large pretrained model such as PolyCoder can serve as a *global oracle* that runs once per problem instance to generate a latent context tensor that is passed to a smaller *planning oracle* used by MCTS. Intuitively, the global oracle can leverage its deeper semantic understanding of programs and formulas to generate global proof insights that are shared with the lower-level planning network. The planning network can then focus on the more mundane task of translating these insights into concrete proof actions.

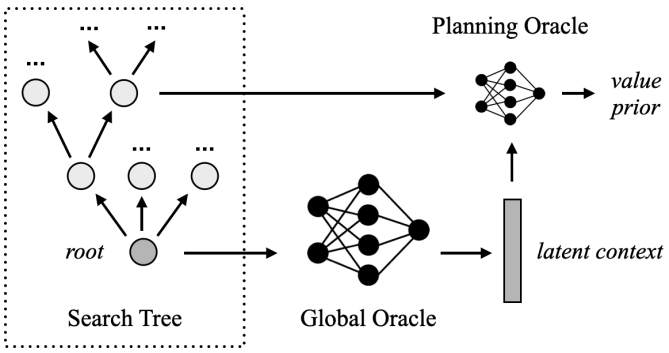


Figure 14: An overview of our proposed hierarchical oracle architecture.

As an extension, it may be desirable to rerun the global oracle on the independent subgoals (section 3.1.2.2) that are pursued within a proof. However, doing so on all subgoals that are considered during search would be too expensive. A possible solution is to only call the global oracle on subgoals that one is committed to (AlphaZero simulates an episode by alternating between planning and committing to an action). Before such a commitment happens, descendants of the subgoal are passed a zero context vector. As a consequence, the planning oracle must be trained to make decisions *with* and *without* the help of the global oracle.

Finally, although our proposed architecture makes inference massively cheaper, training both oracles in an end-to-end fashion still requires paying the cost of propagating gradients through the global oracle. Doing so may become the new computational bottleneck, albeit a much more manageable one. A possible optimization may be to freeze the weights of the global oracle for subsets of gradient updates. This may work especially well since the largest pretrained models tend to require little to no fine-tuning for learning new tasks [57].

3.2 Evaluating our Framework for Depth and Breadth

3.2.1 Evaluation Strategy

Our preliminary work evaluated our framework on the problem of single-loop invariant synthesis with linear arithmetic. However, making a truly compelling experimental case requires further evaluation in both terms of *depth* and *breadth*:

- Regarding depth, our goal is to demonstrate a clear improvement over the existing state-of-the-art on at least one application domain. Our success must be measured along four different dimensions: *i)* how many benchmark problems can be solved by the trained agent, *ii)* how much search is needed to do so, *iii)* how many resources are required for training and *iv)* how much expert knowledge had to be provided (measured in lines of strategy code).
- A specific evaluation is also needed to assess the broad applicability of our approach and its potential for general theorem proving. An important aspect of our thesis is about devising powerful abstractions for designing strategies and such contributions can hardly be evaluated on a single task. Our second goal is therefore to demonstrate our framework on at least three different application domains.

Choosing the right application domains and benchmarks for evaluating our framework is delicate. Indeed, existing benchmarks tend to be shaped by current solver technology and problems that would best illustrate its strengths are not necessarily found in them. In addition, an ideal evaluation task for our framework must have the following properties:

- *The task is well-contained:* although we believe that our framework is ultimately relevant to general theorem proving, realizing this vision requires a large-scale, distributed strategy writing effort that goes beyond the scope of this thesis.
- *Problem instances are concise yet challenging:* because our framework uses Transformer-like neural networks where inference cost is quadratic in the input size, some problems of

inconsequential size for traditional solving approaches would require a significant amount of compute for network processing alone that we would rather spend in other places.

3.2.2 Application Domains

We propose the following tentative application domains to conduct our evaluation.

Invariant synthesis Our preliminary work is evaluated on the Code2Inv [33] benchmark suite. These benchmarks can still be used to measure further progress until every problem can be solved without search. However, they are not fully satisfactory for the purpose of our depth evaluation since they can all be solved by existing solvers via search alone. The Ilinva benchmark suite [24] features more advanced problems with nested loops and advanced data types such as lists. Some of them are unsolved and many currently take minutes to hours to solve. Interestingly, the Ilinva paper proposes an approach for invariant synthesis that is directly expressible as a nondeterministic strategy. Moreover, the proposed benchmark problems are written for the Why3 platform [59]. Why3 is written in OCaml, which makes it easy for our strategy language to interact with its API directly. Additional invariant synthesis tasks can be found in the SV-COMP benchmark suite [60].

Deductive program synthesis We propose using our framework to solve deductive synthesis tasks where programs or fragments thereof are generated from logical specifications. The same invariant synthesis benchmarks that we mentioned earlier can be repurposed for invariant synthesis by hiding fragments of benchmark problems and tasking our solver to build them back. This provides us with a source of hard and interesting synthesis challenges while creating an opportunity to share large amounts of strategy code. Alternatively, many standard benchmarks are available for synthesizing programs from input/output examples [61, 62, 63]. However, as explained in Section 4, these benchmarks may not be as interesting for evaluating our framework since synthetic training data can be generated for those easily, thereby alleviating the need for a teacher agent.

Real inequality proving Real arithmetic inequalities provide challenging yet concise and well-contained challenges for evaluating our framework (e.g. prove that $a/(b+c) + b/(c+a) + c/(a+b) \geq 3/2$ for all $a, b, c > 0$). They already constitute a common target for theorem proving research [64, 12]. Procedural generation methods have been proposed to generate synthetic inequalities [65] but the resulting samples are generally unimpressive and a teacher agent from our framework is likely to generate outputs of greater quality. In terms of concrete benchmarks, the Polya suite [66] provides a collection of inequalities involving special functions that are ill-supported by traditional SMT solvers. More ambitiously, the MiniF2F benchmarks [67] features a collection of Olympiad-level inequalities.

Safe robot planning Differential dynamic logic [68] provides a logical framework for specifying and verifying the safety of cyber-physical systems with continuous dynamics and discrete control. It allows leveraging interactive theorem proving to reason about complex systems that are intractable for model-checkers [41]. As a more open-ended stretch goal, we propose to use our framework to solve robot planning puzzles by learning how to co-generate symbolic controllers and safety proofs using differential dynamic logic. Doing so, we take an unusual angle on the problem of designing provably safe autonomous systems [69]: while the dominant approach is to use formal methods to prove the safety of neural-network controllers [70], our proposal is to have a neural network learn to use formal methods to generate symbolic controllers with safety proofs instead.

3.3 Tool Proposal

As we mentioned before, our long-term vision for this work is dependent on a distributed effort where experts from a variety of areas contribute teacher and solver strategies for their particular domain. For this to happen, writing strategies must be frictionless. Our goal is to release a tool that enables this.

We discussed some desirable features for such a tool in Section 2.3. These include a rich language for writing strategies, a graphical interface for debugging them and a standard library that promotes effective design principles. In addition, our tool must support distributed development by allowing strategies to be shared as independent libraries. Good documentation and tutorials are essential. Finally, a key feature that we are unsure how to best implement yet is the ability to train oracles incrementally. Indeed, retraining an oracle from scratch after each strategy change is incompatible

with fast iteration cycles. A better way must be found, probably by selectively caching, generating and deprecating training data.

At a minimum, we intend to have our tool reviewed by submitting a dedicated tool paper. However, the true criterion for success is to have other students or researchers use it successfully.

4 Related Work

Leveraging nondeterministic programming for proof search The idea of combining nondeterministic expert strategies with neural oracles was proposed by Selsam [16, 71] as a possible angle for tackling the IMO Grand Challenge [72]. However, how to train such oracles remains an open problem. One proposal [35] is to use supervised learning for training a universal oracle that is conditioned on the description of arbitrary search problems and can therefore use training data from a large number of heterogeneous sources. In contrast, we propose using reinforcement learning with the insight that nondeterministic search strategies can be used to *generate* problems in addition to solving them.

Learning to generate synthetic theorems Procedural generation techniques have been used for producing synthetic theorems [65, 73, 74, 28, 11]. These usually proceed in a backwards fashion by generating random proof trees from a given set of axioms and rules. Wang et al. proposed to use supervised learning to learn how to guide this process towards generating more interesting theorems that are similar to those of a reference dataset [75]. In contrast, our proposed approach does not rely on such a dataset and leverages a combination of extrinsic and intrinsic rewards to guide the discovery of novel problems.

Using reinforcement learning for loop invariant synthesis Reinforcement learning has already been applied to the problem of loop invariant synthesis [32, 33], albeit in a very different way. In the aforementioned work (which introduces the Code2Inv benchmark), a separate reinforcement learning agent is trained from scratch on every benchmark problem, using counterexamples from an SMT solver as a training signal. This process takes minutes to hours for each problem to be solved. In contrast, we train a single agent to generalize across problem instances so that new problems can be solved in a matter of milliseconds.

Using reinforcement learning for theorem proving The HOList Zero [14] and TacticZero [15] systems use reinforcement learning to learn how to interact with tactic-based theorem provers without relying on human proofs. However, they still rely on large human-produced corpora of formalized mathematical statements to be used as training tasks and are not yet competitive with approaches based on imitation learning. The use of reinforcement learning has also been explored in the context of saturation-based or tableau-based provers for first-order logic [76, 77]. However, learned heuristics in such settings must operate at a very low-level and are subject to a speed-accuracy tradeoff that is unfavorable to deep learning.

State-of-the-art in machine-learning for general theorem proving The current state-of-the-art [12, 64] for applying machine-learning to general interactive theorem proving involves a three-step training process where *i)* a large language model is pretrained on a big unsupervised corpus of text and code (e.g. GitHub, Math StackExchange and Common Crawl...), *ii)* it is fine-tuned using supervised learning on a corpus of formalized proofs and *iii)* it is improved iteratively via reinforcement learning by generating novel proofs for the theorems in this corpus. Unlike other applications of Transformers [57], scaling up model size beyond 700M parameters hasn't provided meaningful gains [64]. This suggests that the availability of human-written formal statements is a bottleneck.

Leveraging machine-learning for program synthesis A large body of work [61, 62, 63, 78, 79, 80, 81, 82] investigates the use of machine-learning in program synthesis. However, these works concentrate on the problem of synthesizing programs from input/output examples. In this setting, generating training tasks is typically not an issue since problems can be generated by sampling programs from a grammar and evaluating them on random inputs. In contrast, our framework focuses on synthesizing programs from formal specifications. Finally, large language models have been used with remarkable success to generate programs from natural language descriptions [48, 49]. Closer to our setting is the recent work of Haluptzok et al. [50] that demonstrates large language models on

	Safe Goals	Target Goals	Reach Goals
Extensions	Evolving problems	Solver feedback One language extension	All language extensions PolyCoder + Hierarchical
Depth	Code2Inv (no search) Ilinva (large teacher)	Ilinva (small teacher)	Robot planning
Breadth	Real ineqs (Polya) Prog synth (Code2Inv)	Real ineqs (MiniF2F) Prog synth (Ilinva) Robot planning (toy)	$\geq$ 6 application domains
Tool	Strategy libraries UI and documentation	Incremental training More than one user	Formal user study

Table 3: Summary of our thesis objectives.

programming puzzle benchmarks where each puzzle is specified by a single testing function written in Python. In particular, the aforementioned work proposes to generate new training problems by prompting large language models to autocomplete lists of previous interesting benchmarks.

5 Conclusion

This thesis proposal takes the bold challenge of self-learning theorem proving without relying on example theorems and proofs. It advocates a hybrid approach to automated theorem proving where experts are provided a flexible language to formalize their domain-specific knowledge in the form of nondeterministic teacher and solver strategies, leaving blanks to be filled by learning.

Our plans are ambitious. In particular, they feature a massive engineering and tool-building component. However, our completed work demonstrates the feasibility of the proposed challenges. Indeed, we built a highly popular open source implementation [83] of AlphaZero (>1K stars on Github) and demonstrated a meaningful theorem proving use-case for this notoriously compute-intensive algorithm on commodity hardware (Section 2.4). This leaves us able to realistically leverage one to two orders of magnitude more compute for conducting further experiments.

5.1 Timeline and Objectives

We propose a flexible 1 to 2 year timeline for completing our thesis. Our objectives are summarized in Table 3. We organize those into three categories. Safe goals present little to no risk or uncertainty while providing enough material for several research papers. We expect to meet all of them within six months. Target goals constitute together what we argue is a strong and complete case in favor of our proposed framework. They present moderate levels of risk and uncertainty and we believe that they can be met within a year. Finally, we propose a set of reach goals to further strengthen our thesis. Our plan is to dedicate up to a year to these depending on how early our target goals are met and how promising early results are.

Safe Goals

1. Improve our invariant synthesis results on the Code2Inv benchmarks in such a way that all problems can be solved without search (unequaled in the literature) while keeping the amount of strategy code below 1000 LOCs. On the basis of our current untuned agent's performance, achieving this should only require minor improvements to the teacher strategy and to the training pipeline (e.g. have the solver resample problems that it struggles with more frequently). [**1 month**]
2. Write a simple strategy for deductive program synthesis and evaluate it on its ability to reconstruct Code2Inv benchmark programs. Most infrastructure and teacher code can be reused from the invariant synthesis use-case. [**0.5 months**]

3. Write a simple strategy for real inequality proving and evaluate it on the Polya benchmark suite. Doing so should not be conceptually hard since (imperfect) procedures are already known to generate arithmetic inequalities. [**1 month**]
4. Implement a solver strategy for the Ilinva benchmarks suite. Doing so is engineering intensive but conceptually simple since the Ilinva paper already defines what amounts to such a strategy. Implementing a small but effective teacher is hard but we can start with a larger teacher that hardcodes many patterns found in the benchmark suite. [**1.5 month**]
5. Implement the evolutionary problem generation scheme defined in Section 3.1.1.2, which probably presents the best benefits/complexity ratio among all our proposed extensions. Measure its impact on simplifying our Code2Inv teacher. [**1 month**]
6. Release a first version of our tool with support for strategy libraries, some decent UI and documentation. [**1 month**]
7. **Minimal Publication Strategy:** Based on initial feedback from reviewers, our completed work alone may already get accepted to the NeurIPS conference. In case of rejection, implementing the top 2 or 3 goals above should make for a fool-proof resubmission. Another AI paper can then be submitted with the evolutionary problem generation scheme and the early Ilinva results. Achieving the last goal from the list above would also provide enough material for a tool paper. Finally, a publication in a programming language venue (e.g. POPL or PLDI) is warranted since our work has an important language-design component. Such a submission can be made significantly stronger by implementing additional language extensions amd improving our breadth evaluation. Although we classified these as target goals (to be defined next), they can be secured without risk if we decide to solely focus on them as a fallback strategy.

Target Goals

1. Improve our invariant synthesis solver and teacher strategies. The goal is to solve more problems than Ilinva and do so with significantly less search while getting the amount of required strategy code under 500 lines. Doing so will likely require implementing additional framework extensions. [**3 months**]
2. Implement the solver feedback extension (Section 3.1.1.1) and measure its impact in improving and simplifying our teacher strategy. Implement one language extension (Section 3.1.2) that is best motivated by the Ilinva use-case. [**1 month**]
3. Add support for incremental training to our tool. Have another student or researcher use it to tackle a novel problem or make a substantial improvement to an existing strategy. [**0.5 months**]
4. Scale up our program synthesis and real inequality use-cases to tackle the (repurposed) Ilinva and MiniF2F benchmarks respectively. Get initial results on proving small formulas in differential dynamic logic in preparation for the robot planning use-case. [**1.5 months**]

Reach Goals

1. Integrate a pretrained language model (PLUR or PolyCoder if enough compute resources can be secured) into our framework and measure its impact in terms of generalization and learning efficiency. [**2-6 months**]
2. Conduct and publish a large case study on using our framework for safe robot planning. Ideally, do so in collaboration with another researcher. [**1-6 months**]
3. Write simple strategies that can be described in less than a page of pseudocode for as many application domains as possible (at least 6). Doing so would enable to push our tool forward and make a stronger case for the broad applicability of our framework. [**1-3 months**]
4. Run a proper user study for evaluating our tooling and strategy language. [**1-2 months**]

References

[1] The Coq Development Team. *The Coq Reference Manual, version 8.14*, August 2021. Available electronically at `https://coq.inria.fr/documentation`.

[2] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[3] 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*, pages 378–388. Springer, 2015.

[4] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[5] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[6] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. Gamepad: A learning environment for theorem proving. *arXiv preprint arXiv:1806.00608*, 2018.

[7] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning*, pages 6984–6994. PMLR, 2019.

[8] 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*, pages 454–463. PMLR, 2019.

[9] Wenda Li, Lei Yu, Yuhuai Wu, and Lawrence Paulson. Isarstep: a benchmark for high-level mathematical reasoning. *ArXiv*, 2021.

[10] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models. *arXiv preprint arXiv:2102.06203*, 2021.

[11] Stanislas Polu and Ilya Sutskever. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393*, 2020.

[12] 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.

[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] Kshitij Bansal, Christian Szegedy, Markus N Rabe, Sarah M Loos, and Viktor Toman. Learning to reason in large theories without imitation. *arXiv preprint arXiv:1905.10501*, 2019.

[15] Minchao Wu, Michael Norrish, Christian Walder, and Amir Dezfouli. Tacticzero: Learning to prove theorems from scratch with deep reinforcement learning. *arXiv preprint arXiv:2102.09756*, 2021.

[16] Daniel Selsam. Beyond the tactic-state automaton. *Mathematical Reasoning in General Artificial Intelligence Workshop, ICLR*, 2021.

[17] Reiner Hähnle and Peter H. Schmitt. The liberalized δ-rule in free variable semantic tableaux. *J. Autom. Reasoning*, 13(2):211–221, 1994.

[18] David J Wu. Accelerating self-play learning in go. *arXiv preprint arXiv:1902.10565*, 2019.

[19] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1):70–90, 1978.

[20] David Harel. *First-Order Dynamic Logic*. Springer, New York, 1979.

[21] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. `https://hal.inria.fr/hal-00790310`.

[22] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[23] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive invariant generation via abductive inference. *Acm Sigplan Notices*, 48(10):443–456, 2013.

[24] Mnacho Echenim, Nicolas Peltier, and Yanis Sellami. Ilinva: Using abduction to generate loop invariants. In *International Symposium on Frontiers of Combining Systems*, pages 77–93. Springer, 2019.

[25] Thomas Eiter and Georg Gottlob. The complexity of logic-based abduction. *J. ACM*, 42(1):3–42, 1995.

[26] Ivo Danihelka, Arthur Guez, Julian Schrittwieser, and David Silver. Policy improvement by planning with gumbel. In *International Conference on Learning Representations*, 2021.

[27] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[28] Guillaume Lample and François Charton. Deep learning for symbolic mathematics. *arXiv preprint arXiv:1912.01412*, 2019.

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[30] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, 2018.

[31] Jules Hedges. Monad transformers for backtracking search. *arXiv preprint arXiv:1406.2058*, 2014.

[32] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. Learning loop invariants for program verification. In *Neural Information Processing Systems*, 2018.

[33] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. Code2inv: a deep learning framework for program verification. In *International Conference on Computer Aided Verification*, pages 151–164. Springer, 2020.

[34] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. Cln2inv: learning loop invariants with continuous logic networks. *arXiv preprint arXiv:1909.11542*, 2019.

[35] Daniel Selsam, Jesse Michael Han, Leonardo de Moura, and Patrice Godefroid. Universal policies for software-defined mdps. *arXiv preprint arXiv:2012.11401*, 2020.

[36] Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel. Reverse curriculum generation for reinforcement learning. In *Conference on robot learning*, pages 482–495. PMLR, 2017.

[37] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[38] Sarah Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. *arXiv preprint arXiv:1701.06972*, 2017.

[39] Jan Jakubuv and Josef Urban. Enigma: efficient learning-based inference guiding machine. In *International Conference on Intelligent Computer Mathematics*, pages 292–302. Springer, 2017.

[40] Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. Neural program repair by jointly learning to localize and repair. *arXiv preprint arXiv:1904.01720*, 2019.

[41] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp, and André Platzer. Keymaera x: An axiomatic tactical theorem prover for hybrid systems. In *International Conference on Automated Deduction*, pages 527–538. Springer, 2015.

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[43] Aditya Paliwal, Sarah Loos, Markus Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 2967–2974, 2020.

[44] Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *International conference on learning representations*, 2019.

[45] Timothy Gowers. How can it be feasible to find proofs? `https://gowers.wordpress.com/2022/04/28/announcing-an-automatic-theorem-proving-project/`, 2022.

[46] Daniel Whalen. Holophrasm: a neural automated theorem prover for higher-order logic. *arXiv preprint arXiv:1608.02644*, 2016.

[47] Christian Wirth, Riad Akrour, Gerhard Neumann, Johannes Fürnkranz, et al. A survey of preference-based reinforcement learning methods. *Journal of Machine Learning Research*, 18(136):1–46, 2017.

[48] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[49] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

[50] Patrick Haluptzok, Matthew Bowers, and Adam Tauman Kalai. Language models can teach themselves to program better. *arXiv preprint arXiv:2207.14502*, 2022.

[51] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *arXiv preprint arXiv:2206.14858*, 2022.

[52] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

[53] Jacob Steinhardt. Forecasting ml benchmarks in 2023. `https://bounded-regret.ghost.io/forecasting-math-and-mmlu-in-2023/`, 2020.

[54] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[55] Helmholtz AI. Haicore: computing resources for the helmholtz ai community. `https://www.helmholtz.ai/themenmenue/you-helmholtz-ai/computing-resources/index.html`, 2022.

[56] Zimin Chen, Vincent J Hellendoorn, Pascal Lamblin, Petros Maniatis, Pierre-Antoine Manzagol, Daniel Tarlow, and Subhodeep Moitra. Plur: A unifying, graph-based view of program learning, understanding, and repair. *Advances in Neural Information Processing Systems*, 34:23089–23101, 2021.

[57] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[58] Frank F Xu, Uri Alon, Graham Neubig, and Vincent J Hellendoorn. A systematic evaluation of large language models of code. *arXiv preprint arXiv:2202.13169*, 2022.

[59] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – where programs meet provers. In *European symposium on programming*, pages 125–128. Springer, 2013.

[60] Dirk Beyer. Software verification: 10th comparative evaluation (sv-comp 2021). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 401–422. Springer, 2021.

[61] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[62] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International conference on machine learning*, pages 990–998. PMLR, 2017.

[63] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

[64] 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.

[65] Yuhuai Wu, Albert Jiang, Jimmy Ba, and Roger Grosse. Int: An inequality benchmark for evaluating generalization in theorem proving. *arXiv preprint arXiv:2007.02924*, 2020.

[66] Jeremy Avigad, Robert Y Lewis, and Cody Roux. A heuristic prover for real inequalities. In *International Conference on Interactive Theorem Proving*, pages 61–76. Springer, 2014.

[67] Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. Minif2f: a cross-system benchmark for formal olympiad-level mathematics. *arXiv preprint arXiv:2109.00110*, 2021.

[68] André Platzer. Differential dynamic logic for hybrid systems. *Journal of Automated Reasoning*, 41(2):143–189, 2008.

[69] Sanjit A Seshia, Dorsa Sadigh, and S Shankar Sastry. Toward verified artificial intelligence. *Communications of the ACM*, 65(7):46–55, 2022.

[70] Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher Strong, Clark Barrett, Mykel J Kochenderfer, et al. Algorithms for verifying deep neural networks. *Foundations and Trends® in Optimization*, 4(3-4):244–404, 2021.

[71] Daniel Selsam. The IMO grand challenge. `http://aitp-conference.org/2020/slides/DS.pdf`, 2020. Talk at AITP 2020.

[72] Daniel Selsam. The IMO grand challenge: A battle of ideas. `https://dselsam.github.io/IMO-GC-battle-of-ideas/`, 2020.

[73] Eser Aygün, Zafarali Ahmed, Ankit Anand, Vlad Firoiu, Xavier Glorot, Laurent Orseau, Doina Precup, and Shibl Mourad. Learning to prove from synthetic theorems. *arXiv preprint arXiv:2006.11259*, 2020.

[74] Yury Puzis, Yi Gao, and Geoff Sutcliffe. Automated generation of interesting theorems. In *FLAIRS Conference*, pages 49–54, 2006.

[75] Mingzhe Wang and Jia Deng. Learning to prove theorems by learning to generate theorems. *arXiv preprint arXiv:2002.07019*, 2020.

[76] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Mirek Olšák. Reinforcement learning of theorem proving. *arXiv preprint arXiv:1805.07563*, 2018.

[77] Maxwell Crouse, Ibrahim Abdelaziz, Bassem Makni, Spencer Whitehead, Cristina Cornelio, Pavan Kapanipathi, Kavitha Srinivas, Veronika Thost, Michael Witbrock, and Achille Fokoue. A deep reinforcement learning approach to first-order logic theorem proving. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 6279–6287, 2021.

[78] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, 2018.

[79] Xinyun Chen, Dawn Song, and Yuandong Tian. Latent execution for neural program synthesis beyond domain-specific languages. *Advances in Neural Information Processing Systems*, 34:22196–22208, 2021.

[80] Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. *Advances in Neural Information Processing Systems*, 32, 2019.

[81] Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. In *International Conference on Machine Learning*, pages 4861–4870. PMLR, 2019.

[82] Maxwell Nye, Yewen Pu, Matthew Bowers, Jacob Andreas, Joshua B Tenenbaum, and Armando Solar-Lezama. Representing partial programs with blended abstract semantics. *arXiv preprint arXiv:2012.12964*, 2020.

[83] Jonathan Laurent. Alphazero.jl: A generic, simple and fast AlphaZero implementation. `https://github.com/jonathan-laurent/AlphaZero.jl`, 2021.

[84] George B Dantzig. Fourier-motzkin elimination and its dual. Technical report, STANFORD UNIV CA DEPT OF OPERATIONS RESEARCH, 1972.

[85] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[86] Vighnesh Shiv and Chris Quirk. Novel positional encodings to enable tree-based transformers. *Advances in Neural Information Processing Systems*, 32:12081–12091, 2019.

[87] Christopher Hahn, Frederik Schmitt, Jens U Kreber, Markus N Rabe, and Bernd Finkbeiner. Teaching temporal logics to neural networks. *arXiv preprint arXiv:2003.04218*, 2020.

A Details on the Invariant Synthesis Demonstrating Use-Case

A.1 The Solver Strategy

In Figure 15, we provide some detailed pseudocode for the solver strategy that we implemented for the purpose of the experiments in section 2.4. This strategy is similar to the simple one introduced in Figure 3 but it comes with the following extensions:

- **Ability to abduct disjunctive invariants:** Imagine a proof obligation fails and the `abduct` function returns $a_1, \ldots, a_n$ as possible missing assumptions. In some cases, none of those can be proved invariant but the disjunction of a subset of them can. This is why the `suggest_missing` function defined on line 28 is used to build an invariant candidate as a disjunction of at most 3 (atomic) abduction candidates.
- **Ability to strengthen abducted invariants:** An abducted invariant candidate may have to be strengthened before it can be proved invariant. For example, the abduction engine may suggest $x \neq 0$ as a missing assumption but $x \neq 0$ may not be a valid invariant whereas $x > 0$ (or $x < 0$) is. The call to `strengthen` on line 47 is used to optionally and nondeterministically strengthen an invariant candidate. Our current strategy supports two kinds of strengthening: replacing a formula of the form $A \neq B$ by either $A > B$ or $A < B$ or weakening an inequality of the form $A \leq B$ into $A \leq B + c_?$ where $c_?$ is a nonnegative constant whose exact value is to be determined later (see next point).
- **Ability to instantiate constants lazily via abduction:** Invariant candidates can feature metavariables that denote unknown constants. The `constrs` variable defined in line 8 collects global constraints about these metavariables. If a missing assumption is suggested that only features metavariables, then it is added as a constraint rather than as a new invariant candidate (line 42). After an invariant is proved, the metavariables it contains are instantiated using concrete values in a way to satisfy all global constraints (see call to `abduct_refinement` on line 57). A metavariable that appears in the invariant as an upper bound is instantiated with a value that is as low as possible, whereas a metavariable that appears as a lower bound is instantiated with a value that is as high as possible (this information is contained in the `btype` variable).
- **Ability to conjecture invariant templates:** In some cases, abduction alone is not enough to discover a missing invariant. For example, if two disjunctive invariants I_1 and I_2 are needed for proving the postcondition, one may have to conjecture the first one and then use abduction to find the other. The strategy in Figure 15 allows conjecturing invariant templates with metavariables to be instantiated via abduction (see previous point). The `conjectures` function called on line 29 returns three kinds of conjecture candidates: *i)* conjectures of the form $t \odot c_?$ where $\odot \in \{\geq, =\}$ and $t = \sum_i a_i x_i$ is a linear combination of variables that is preserved by the loop body, *ii)* relaxations of the loop guard (e.g. $x \leq 10 + c_?$ with $c_? > 0$ if the loop guard is $x \leq 10$) and *iii)* initial assumptions that only contain variables that are not modified by the loop body.

The solver strategy also emits two types of events (see section 2.2.4) at lines 38 and 36 respectively. Conjecturing events are associated with a reward of -0.3 and abduction events are associated with a reward of -0.2. Both kinds of events are counted at most four times ($m_e = 4$) and the minimal total reward delivered in case of a success is $r_{\texttt{min}} = 0$.

Hints on how to use our proposed solver strategy to solve problems from the Code2Inv benchmark are available in Appendix A.3.

A.2 The Teacher Strategy

The teacher strategy we use for loop invariant generation follows the structure introduced in Figure 4. We provide additional details below:

- **Sampling constraints:** The full list of available constraints is available in Table 4. Values are sampled *mostly* independently for each constraint types. In our implementation, we hardcode a small number of correlations (e.g. we are more likely to sample a disjunctive

```
def solver(
   init: Formula,
   guard: Formula,
   body: Program,
   post: Formula) -> List[Formula]:

   invs_proved: List[Formula] = []
   constrs: List[Formula] = []
   pending: List[Pending] = []

   def prove_post():
      pending[-1].status = TO_PROVE
      to_prove = Implies(
         constrs + invs_proved + [Not(guard)],
         post)
      match abduct(to_prove):
         case Valid:
            pending[-1].status = PROVED
         case [*suggs]:
            assum = suggest_missing(suggs)
            closing = implies(assum, to_prove)
            pending[-1].status = \
               PROVED_COND if closing
               else TO_PROVE_NEXT
            prove_missing(assum, as_inv=True)
            prove_post()

   def suggest_missing(suggs: List[Formula]):
      suggs += conjectures(guard, body)
      num_disjs = choose([1, 2, 3])
      disjs = []
      for i in range(num_disjs):
         d = choose(suggs)
         disjs.append(d)
         if is_conjecture(d):
            event(CONJECTURING_EVENT)
         else:
            event(ABDUCTION_EVENT)
      return Or(*disjs)

   def prove_missing(f: Formula, as_inv: bool):
      if meta_only(f):
         constrs.append(f)
         assert sat(constrs)
      else:
         assert as_inv
         inv, fresh = strengthen(f)
         for c, _ in fresh:
            constrs.append(Ge(Metavar(c), 0))
         pending.append(
            Pending(INV, inv, TO_PROVE))
         prove_init(inv)
         prove_preserved(inv)
         invs_proved.append(inv)
         pending.pop()
         for c, btype in fresh:
            cval = abduct_refinement(
               c, btype, constrs)
            subst(invs_proved, c, cval)
            subst(constrs, c, cval)

   def prove_init(inv: Formula):
      pending[-1].status = TO_PROVE
      to_prove = Implies(
         constrs + proved_invs + [init],
         inv)
      match abduct(to_prove):
         case Valid: return
         case [*suggs]:
            assum = choose(suggs)
            prove_missing(assum, False)

   def prove_preserved(inv: Formula):
      pending[-1].status = TO_PROVE
      to_prove = Implies(
         constrs +
         proved_invs + [guard, inv],
         loop_body.wlp(inv))
      match abduct(to_prove):
         case Valid: return
         case [*suggs]:
            suggs = suggest_missing(suggs)
            closing = implies(assum, to_prove)
            pending[-1].status = \
               PROVED_COND if closing
               else TO_PROVE_NEXT
            perform(action)
            prove_inv_inductive(inv)

   pending.append(
      Pending(POST, post, TO_PROVE))
   prove_post()
   pending.pop()
   return invs_proved
```

Figure 15: Strategy for the solver agent. The code in gray is only useful for providing the network with contextual information and can be disregarded on first reading. Every call to `choose` is implicitly passed the program counter along with the value of all global parameters and variables defined in lines 2 to 9. In particular, the `pending` variable summarizes all information from the program stack that is relevant to the neural network.

formula for `inv_main` if `inv_lin` is not used in order to keep things interesting). We also reject a number of constraint combinations that are clearly uninteresting or unsatisfiable.

- **Fixed constraints:** In addition to penalizing the violation of sampled constraints, the teacher implements fixed hard constraints that are always enforced. A list of all such constraints is available in Table 5. Violating one of these constraints leads to an immediate failure along with a reward of -1.
- **Refining formulas and using abduction:** Different parts of the problem template shown in Figure 4 are nondeterministically refined in turn. To refine an atomic formula, a template is first selected of the form $x_? \odot c_?$ or $x_? \odot y_?$ where $x_?$ and $y_?$ are variable placeholders, $c_?$ is a constant placeholder and $\odot \in \{<, \leq, >, \geq, =, \neq\}$. Each variable placeholder is then nondeterministically substituted by an existing or a fresh variable. Constant placeholders are either instantiated with concrete constants (a set of 6 available numerical constants is sampled at the start of the teacher strategy along with constraints) or parameters (special variables that cannot be modified by the program). Constant placeholders can also be left as-is and refined later using abduction (e.g. before invariant preservation is checked, abduction is used to suggest values for the remaining constant placeholders). Abduction

is also used to suggest required parameter assumptions that are added to `init` (e.g. $n > 0$ where n is a variable not modified in the program).

- **Refining programs:** Subprograms are refined by selecting a sequence of assignment templates of the form: $x_? := c_?$, $x_? := y_?$, $x_? := x_? + d_?$, $x_? := x_? - d_?$, $x_? := x_? + y_?$ and $x_? := c_? - y_?$ ($d_?$ is a placeholder for a strictly positive constant). A special `skip` template can be selected to stop adding assignments. Variable and constant placeholders are handled in the same way they are handled in formulas. Which templates are available is determined by the `assignment-templates` constraint (see Figure 4).
- **Extra refinement suggestions:** Extra suggestions are added to the standard templates when refining some formulas. When adding a disjunct to the main invariant, the loop guard itself is added as a suggestion along with a relaxed version of it (using a placeholder constant). When refining the last disjunct of the postcondition, abduction is used to suggest candidates that are consequences of the current assumptions in the associated proof obligation. Abduction is also used to suggest conjuncts for `init`.
- **Detecting constraint violations early:** Constraint violations are detected as early as possible to allow early feedback during search. For example, whether or not the invariant is satisfiable is checked right after the invariant has been refined and before the loop body is refined in turn. Most constraints must be checked again whenever a new parameter assumption is added. For example, an invariant $0 \leq x \wedge x < n$ may be initially judged as satisfiable. However, abduction may later on suggest $n < 0$ as a parameter assumption, making it unsatisfiable.
- **Applying random transformations to generated problems:** For increased diversity, a sequence of random transformations is applied to any problem generated by the teacher before it is returned. We provide a list of all such transformations in Table 6.

A.3 Examples of Code2Inv Problems

We show examples of Code2Inv benchmark problems in Table 7, along with some hints on how they can be solved using the strategy detailed in Appendix A.1.

A.4 Examples of Generated Teacher Problems

We show examples of challenge problems generated by our trained teacher agent in Table 8.

A.5 Looprl UI Screenshots

We show examples of using the Looprl user interface to inspect the solver and teacher strategies in Figures 16 and 17 respectively.

A.6 Implementing Abduction

Both the teacher and solver strategies in this use-case rely on an `abduct` function that takes as an input a formula F and then either proves it valid or returns a (possibly empty) set of assumptions A such that $A \rightarrow F$ can be proved to be valid.

Implementing such a function is hard in the general case and so an implementation of `abduct` may leverage nondeterminism. However, because we are only dealing with arithmetic of limited complexity in this work, we implemented a fully-specified abduction procedure for linear integer arithmetic that relies on Fourier-Motzkin elimination [84].

When given a formula F, our abduction procedure first rewrites it in conjunctive normal form as $F = \bigwedge_{i=1}^{n} \bigvee_j F_{ij}$ where F_{ij} are atomic formulas of the form $\sum_k a_k x_k \odot c$ where $\odot \in \{\geq, =\}$ and c, a_k are integer constants.

A.6.1 Case where $n = 1$

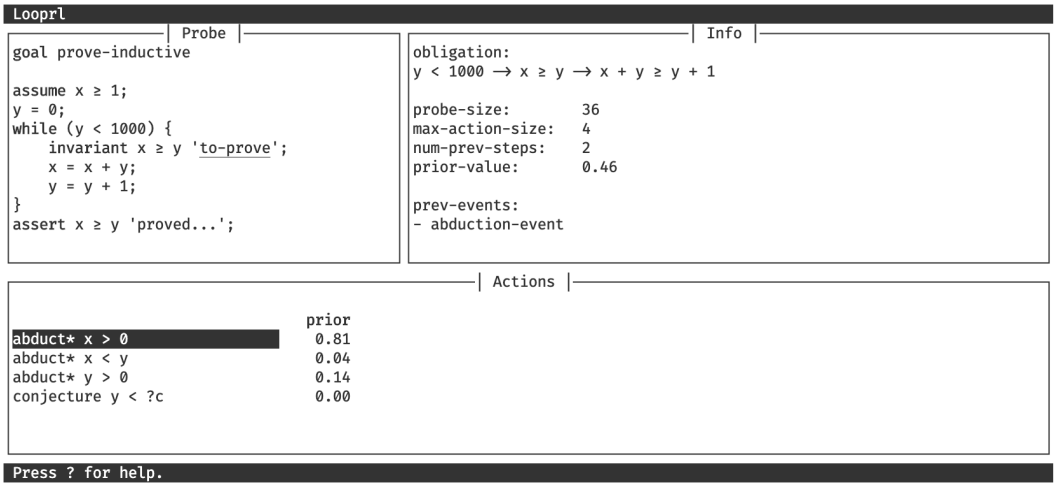
If F can be expressed as a disjunction of atomic formulas, we can compute abduction suggestions as follow: *i)* one considers the negation of F, obtaining a set of atomic assumptions and *ii)* one

Name	Type	Description
num-preserved-term-vars	none|2|3	If an integer n, then inv_lin is refined with an invariant of the form $\sum_{i=1}^{n} a_i x_i = c$.
num-inv-main-disjuncts	none|1|2	If an integer n, then inv_main is refined with a disjunction of n atomic formulas.
num-inv-aux-conjuncts	none|1|2	If an integer n, then inv_aux is refined with a conjunction of n atomic formulas.
num-post-disjuncts	1|2	Number of desired atomic disjuncts for post.
has-conditional	bool	Whether body must include a conditional statement.
has-else-branch	bool	Whether the conditional in body has an else branch.
has-cond-guard	bool	Whether the conditional in body has a guard. If not, the guard is refined with the nondeterministic expression *.
body-implies-main-inv	bool	If true, then inv_main always holds after executing body regardless of whether or not it holds before.
loop-guard-useful-for-inv	bool	Whether assuming the loop guard is useful in proving that inv_main is preserved.
loop-guard-useful-for-post	bool	Whether assuming the negation of the loop guard is useful in proving the postcondition post.
use-params	bool	Whether or not to use variables that have a constant value throughout the program.
eq-only-for-init	bool	Whether or not to use equalities only in init.
loop-guard-template	template	The template to be used to refine the loop guard (e.g. a constant upper bound on a variable).
assignment-templates	templates	Allowed assignment templates for the body (e.g. constant var increment, assigning a var to another one...)
allow-vcomp-in-inv-main	bool	Whether inv_main's first disjunct can feature a comparison between two variables modified by the program.

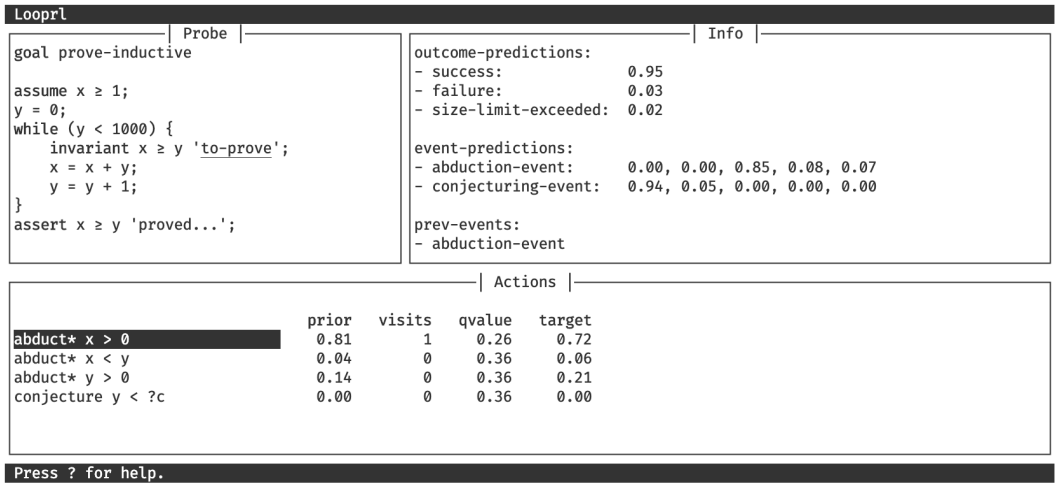
Table 4: Complete list of teacher constraints. Every constraint type is associated with a separate violation event. The associated reward is -0.5 for all constraint types except the last four ones where it is -0.2. A total reward of at least $r_{\texttt{min}} = -0.5$ is delivered in case of a success.

Name	Description
correctness	The problem is correct, meaning that the invariant (i.e. the conjunction of inv_lin, inv_main and inv_aux) respects the three properties defining a valid invariant (i.e holds initially, preserved by the loop body and implies the postcondition).
{inv_main,post,init}-not-valid-unsat-or-redundant	Disjunctive or conjuctive formulas such as inv_main, post and init must not be valid, unsatisfiable or redundant in the sense that they can be simplified (e.g. $x > 0 \vee x = 0$ is redundant because it simplifies to $c \geq 0$ and $x > 1 \wedge x > 2$ is redundant because it simplifies to $x > 2$).
inv-sat	The invariant must be satisfiable.
loop-terminates	The loop guard must not be preserved by the loop body when assuming the invariant, in which case the loop would never terminate once entered. (Note this constraint only rules out a subset of nontermination cases.)
loop-entered	The init formula does not imply the negation of the loop guard.

Table 5: Table of fixed teacher hard constraints. Violation of such a constraint leads to an immediate failure and to a reward of -1.



(a) Showing the proof obligation associated with the abduction call along with the neural network policy prior.



(b) Showing event predictions along with MCTS statistics.

Figure 16: Visualizing the solver strategy with the Looprl UI. In the screenshots above, the UI is used to examine a choice point in the solver strategy where the current invariant candidate $x \geq y$ cannot be proved to be inductive and the user must choose between proving one of several abduction candidates or making a conjecture (this roughly corresponds to the call to `choose` on line 33). Here, one can see the network assigning a high prior probability to the optimal proof action, which is to try and prove $x > 0$ as an invariant. The network also predicts a value of 0.46 for this state, which is close to the truth of 0.4 (proving this problem requires three abduction events with cost 0.2 each). The details of how the value is estimated can be consulted in screenshot 16b. Here, we can see that the network predicts a 0.95 probability of success along with a probability of 0.94 for not requiring any conjecture and a probability of 0.85 for needing two more abduction events.

Name	Description
`add-useless-loop-guard`	If the loop guard is irrelevant, assign a random formula in its place.
`add-useless-init`	Add a random conjunct to `init`.
`add-useless-post`	Add a random disjunct to `post`.
`add-useless-cond`	Replace `body` by `if (cond) body` where `cond` is a random formula.
`rearrange-commutative`	Shuffle the order of disjunctions and conjunctions.
`move-conditional`	Commute the conditional statement in `body` with other instructions.
`shuffle-instrs`	Shuffle the order of consecutive assignments.
`randomize-comparisons`	Randomize comparisons by changing variable order or converting strict inequalities to nonstrict inequalities and vice versa.
`move-param-assum`	Remove a parameter assumption and add its negation to `post`.
`make-post-assums`	Rewrite a disjunctive final assertion into a sequence of atomic assumptions with a final atomic assertion (e.g. rewrite "`assert x>0||y>0`" into "`assume x<=0; assert y>0`").
`make-init-instrs`	Replace the `init` conjunctive assumption by a sequence of variable assignments ad atomic assumptions.
`weaken-post`	Weaken the final (atomic) postcondition (e.g. replace "`assert x>0`" by "`assert x!=0`").

Table 6: Complete list of the final problem transformations implemented by the teacher. Some transformations may trigger or not based on a fixed probability. A transformation application is cancelled if it leads to violating a hard constraint or increasing the number of soft constraint violations.

derives as many consequences as possible from those assumptions using Fourier-Motzkin elimination (linearly combining inequalities so as to eliminate variables). If a contradiction is derived, then F is valid. If no contradiction is dervied given some timeout, then the negation of any derived consequence can be considered as an abduction candidate.

Example Suppose we want to compute abduction candidates for $F = x \geq 0 \ \rightarrow \ x + y \geq 1$. The conjunctive normal form of F is $(x < 0 \vee x+y \geq 1)$. Taking the negation yields $(x \geq 0 \wedge x+y < 1)$, which we normalize into the set of assumptions $\{x \geq 0, -x - y \geq 0\}$ (all variables are integers). Then, we can take a Fourier-Motzkin step by adding these two assumptions and derive the following consequence: $-y \geq 0$. After this, no other reasoning step is applicable end we end up with the following set of facts: $\{x \geq 0, -x - y \geq 0, -y \geq 0\}$. We therefore suggest the following abduction candidates: $x < 0$, $x + y > 0$ and $y > 0$.

A.6.2 Case where $n > 1$

If the conjunctive normal form of F has two conjuncts or more, we apply the procedure above on each conjunct separately. For example, suppose that $F = G \wedge H$, G admits a set of abduction candidates $\{A_i\}_i$ and H admits a set of abduction candidates $\{B_i\}_i$. One possibility would be to return all $\{A_i \wedge B_j\}_{ij}$ combinations as abduction candidates for F. However, doing so can quickly result in a combinatorial explosion. Therefore, our implementation does something different and returns the union of the $\{A_i\}_i$ and $\{B_i\}_i$ instead. Doing so, it cannot provide the guarantee that any resulting abduction candidate A is sufficient in implying F. Rather, abduction candidates are seen as suggestions to unblock one part of the proof (i.e. enable proving one conjunct of F) but not necessarily the whole proof.

A.7 Training Hyperparameters

We provide an exhaustive list of all hyperparameter values we used in our experiments in Table 9.

```
x = 0;
while (x < 5) {
  x = x + 1;
  if (y > z) {
    y = z;
  }
}
assert y <= z;
```

Problem 3

Invariant: $x < 5 \vee y \leq z$.

This problem can be solved using our proposed strategy by directly abducting the correct disjunctive invariant when attempting to prove the postcondition.

```
assume x <= 10;
assume y >= 0;
while (*) {
  x = x + 10;
  y = y + 10;
}
assume x == 20;
assert y != 0;
```

Problem 7

Invariant: $x - y \leq 10$.

The star ($*$) corresponds to a nondeterministic boolean value. In this case, the loop body can be executed an arbitrary number of times.

This problem can be solved by conjecturing an invariant of the form $x - y \leq c_?$ and then using abduction to refine $c_?$.

```
assume n >= 0;
i = 0;
x = 0;
y = 0;
while (i < n) {
  i = i + 1;
  if (*) {
    x = x + 1;
    y = y + 2;
  } else {
    x = x + 2;
    y = y + 1;
  }
}
assert 3*n == x + y;
```

Problem 93

Invariant: $3i = x + y \wedge i \leq n$.

This problem can be solved by conjecturing an invariant of the form $3i - x - y = c_?$, refining $c_?$ through abduction and then abducting $i \leq n$ as a missing invariant while trying to prove the postcondition.

```
s = 0;
i = 1;
while i <= n:
  i = i + 1
  s = s + 1
assume s != 0
assert s == n
```

Problem 110

Invariant: $i - s = 1 \wedge (i \leq n + 1 \vee s = 0)$.

This problem can be solved by first conjecturing an invariant of the form $i - s = c_?$, using abduction to refine $c_?$ and then abducting the disjunctive invariant $i \leq n + 1 \vee s = 0$ while trying to prove the postcondition.

Table 7: Some examples of Code2Inv problems.

`main-inv 1`
`body-structure no-cond`
`loop-guard-useful-for-post`
`available-consts -9 -6 4 8`
 	
`assume y == x;`
`while (x < 1) {`
`  invariant y == x;`
`  y = y + 1;`
`  x = x + 1;`
`}`
`assert y > 0;`
	In this example, the network has been tasked to generate an example of a program with a single atomic invariant and a loop guard that is useful to establish the postcondition but not the invariant itself.

`main-inv 1`
`use-aux-inv 2`
`body-structure no-cond`
`allow-vcomp-in-prim-inv`
`no-var-const-assign`
`available-consts -8 -7 2 6`
 	
`assume x < y;`
`assume x >= 5;`
`while (*) {`
`  invariant x < y;`
`  invariant y >= 6 && x >= -1;`
`  x = x + 6;`
`  y = y + x;`
`}`
`assert x < y;`
	In this example, the network has been tasked to generate an example that involves an auxilliary event with two conjuncts. The auxilliary event can be useful to prove the main invariant but not the postcondition. Assignments of the form $x = y$ or $x = c$ where x and y are variables and c is a constant are not allowed.

`preserved-term 2`
`disjunctive-post`
`body-structure no-cond`
`only-constr-incr`
`available-consts -55 -52 21 21`
 	
`assume x == 21;`
`assume y == -52;`
`while (*) {`
`  invariant -x - 3*y == 135;`
`  y = y - 21;`
`  x = x + 63;`
`}`
`assert y != 21 || x == -198;`
	In this example, the neural network is tasked to find an example of a problem involving a linear invariant with two variables, no loop guard and a disjunctive postcondition. Note that an irrelevant loop guard may be added in the final transformation stage of the teacher.

Table 8: Some examples of problems generated by the teacher. We show the associated invariants for clarity but those should of course be hidden before problems are sent to the solver agent. These invariants only provide one way to solve the associated problems and alternative invariants may exist. We disable the final random transformations applied by the teacher for clarity and to avoid clutter from useless formulas and instructions. Finally, some problem constraints are not listed for brevity unless their value is different from the default with highest probability.

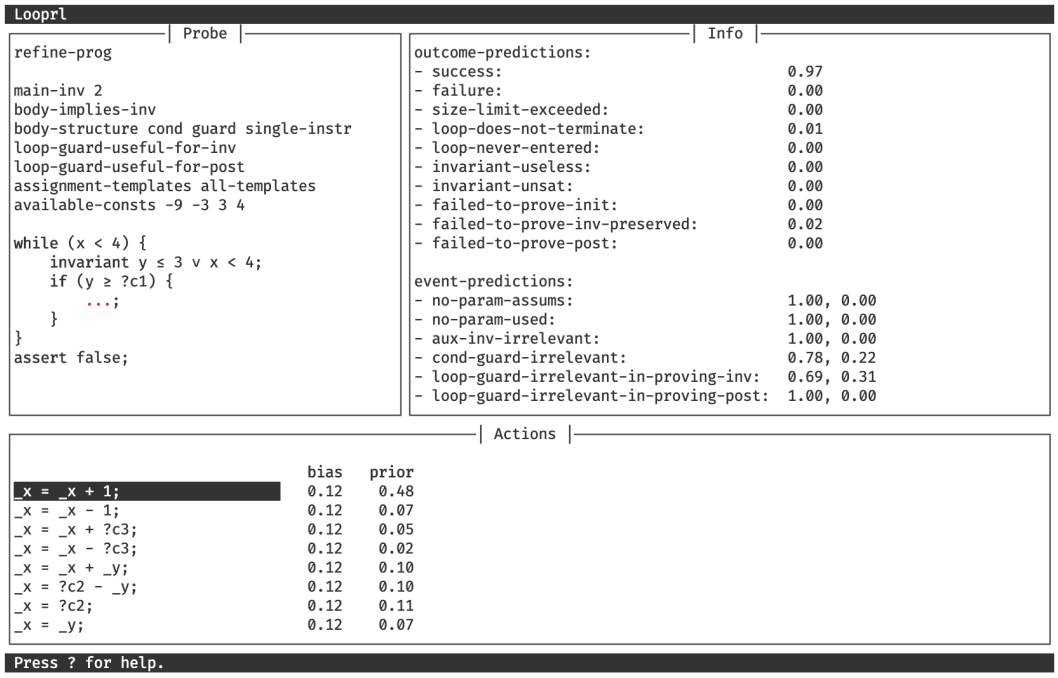


Figure 17: Visualizing the teacher strategy with the Looprl UI. This screenshot captures a choice point where the network is tasked with adding an assignment to the true branch of the conditional within the loop body. Several templates are proposed, which are going to be refined in turn. The upper left pane (i.e. the *probe* pane) features all contextual information that is sent to the network to help it make a choice. Among this information, we can see a list of all constraints that the generated problem should ideally satisfy. The *info* pane gives us some insights into the network's prediction about future events and outcome. For example, the network estimates with 0.97 probability that a valid problem will be generated (along with a 0.02 probability that a failure will be encountered due to the invariant not being preserved by the loop body). The network also estimates a 31% risk that the generated problem will violate the soft constraint according to which the loop guard should be irrelevant in proving the invariant.

Parameter	Value	Description
`params`		All hyperparameters.
`* teacher`		Hyperparameters for the teacher agent.
`* * agent`		Hyperparameters common to all AlphaZero agents.
`* * * num-iters`	20	Total number of training iterations.
`* * * num-problems-per-iter`	8000	Number of data generation episodes per iteration.
`* * * num-validation-problems`	800	Number of validation-data generation episodes per iteration.
`* * * num-workers`	600	Number of episodes simulated in parallel or asynchronously during data generation.
`* * * num-processes`	none	Number of distinct CPU processes spawned for data generation (by default, this value is set to the number of available physical CPU cores).
`* * * search`		Proof search limits.

* * * * `max-proof-length`	60	Maximum number of allowed environment steps before failing automatically.
* * * * `max-probe-size`	80	Maximum allowed size of state encodings. Bigger state encodings result in immediate failures.
* * * * `max-action-size`	12	Maximum allowed size of action encodings. Actions with bigger encodings are discarded.
* * * `encoding`		State and action encoding hyperparameters.
* * * * `d-model`	128	Embedding size for tokens, which is also equal to the neural network's state dimension.
* * * * `pos-enc-size`	32	Maximal size of the tree positional encoding added to each token embedding.
* * * * `uid-emb-size`	16	Maximum number of unique identifiers that can be encoded.
* * * * `const-emb-size`	0	Maximum number of bits used to encode the binary value of numerical constants. No encoding is done if a value of 0 is provided.
* * * * `enable-numerical-edges`	true	Add comparison edges between numerical constants.
* * * * `add-reverse-edges`	true	Add reverse edges for all edge types.
* * * `network`		Hyperparameters for the Dynamic Graph Transformer network.
* * * * `num-heads`	4	Number of transformer attention heads.
* * * * `probe-encoder-layers`	6	Number of transformer blocks used to encode states.
* * * * `action-encoder-layers`	3	Number of transformer blocks used to encode actions
* * * * `combiner-layers`	1	Number of transformer blocks used in the combiner network that combines state and action encodings.
* * * * `dropout-rate`	0.05	Dropout rate.
* * * * `num-head-layers`	2	Number of layers in the feed-forward networks used for the value and policy heads.
* * * * `head-dim`	256	Hidden dimension of layers in the feed-forward networks used for the value and policy heads.
* * * `mcts`		MCTS hyperparameters.
* * * * `num-simulations`	64	Number of MCTS simulations used for planning every environment step.
* * * * `num-considered-actions`	8	The number of top-ranked actions that are considered in the sequential halving Gumbel exploration process.
* * * * `value-scale`	0.1	The c_{scale} hyperparameter (see original Gumbel AlphaZero paper).
* * * * `max-visit-init`	50	The c_{visit} hyperparameter (see original Gumbel AlphaZero paper).
* * * * `fpu-red`	0.1	Value estimates for unvisited children are decreased by this value so as to encourage deeper search trees (see LC0 AlphaZero implementation).
* * * * `reset-tree`	true	Whether the MCTS tree is reset after an environment move is taken.
* * * * `max-tree-size`	256	Maximal size of the MCTS tree. This parameter is relevant to avoid out-of-memory errors when `reset-tree` if false.

* * * * dirichlet-alpha	10	Concentration parameter for the Dirichlet exploration noise (see original AlphaZero paper).
* * * * dirichlet-eps	0.25	Magnitude of the Dirichlet exploration noise. No Dirichlet exploration noise is normally used with Gumbel AlphaZero but we add some anyway in the teacher for increased diversity.
* * * training		Network training hyperparameters.
* * * * max-epochs	6	The maximum number of training epochs.
* * * * improvement-required	1	Gradient updates are stopped if the validation loss fails to decrease for more than this number of epochs.
* * * * batch-size	400	Training batch size.
* * * * lr-base	0.0005	Maximal learning rate used in a cosine learning rate schedule with warm-up.
* * * * warmup-epochs	0.2	Length of the warm-up part of the learning rate schedule (in epochs).
* * * * weight-decay	0.01	Weight decay parameter.
* * * * outcome-loss-coeff	0.7	Coefficient for the loss term evaluating outcome prediction accuracy (e.g. success or failure).
* * * * event-loss-coeff	3	Coefficient for the loss term evaluating event prediction accuracy.
* * * * policy-loss-coeff	1	Coefficient for the loss term evaluating the divergence from policy priors to their targets.
* * * training-window	1, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6, 7	The number of previous iterations from which samples are used to update the network at each iteration. If the provided list is shorter than the total number of iterations, the last number in this list is duplicated as many times as necessary.
* solver		Hyperparameters for the solver agent.
* * agent		Hyperparameters common to all AlphaZero agents.
* * * num-iters	20	Total number of training iterations.
* * * num-problems-per-iter	20000	Number of data generation episodes per iteration.
* * * num-validation-problems	5000	Number of validation-data generation episodes per iteration.
* * * num-workers	600	Number of episodes simulated in parallel or asynchronously during data generation.
* * * num-processes	none	Number of distinct CPU processes spawned for data generation (by default, this value is set to the number of available physical CPU cores).
* * * search		Proof search limits.
* * * * max-proof-length	12	Maximum number of allowed environment steps before failing automatically.
* * * * max-probe-size	80	Maximum allowed size of state encodings. Bigger state encodings result in immediate failures.
* * * * max-action-size	12	Maximum allowed size of action encodings. Actions with bigger encodings are discarded.
* * * encoding		State and action encoding hyperparameters.
* * * * d-model	128	Embedding size for tokens, which is also equal to the neural network's state dimension.

* * * * pos-enc-size	32	Maximal size of the tree positional encoding added to each token embedding.
* * * * uid-emb-size	16	Maximum number of unique identifiers that can be encoded.
* * * * const-emb-size	0	Maximum number of bits used to encode the binary value of numerical constants. No encoding is done if a value of 0 is provided.
* * * * enable-numerical-edges	true	Add comparison edges between numerical constants.
* * * * add-reverse-edges	true	Add reverse edges for all edge types.
* * * network		Hyperparameters for the Dynamic Graph Transformer network.
* * * * num-heads	4	Number of transformer attention heads.
* * * * probe-encoder-layers	6	Number of transformer blocks used to encode states.
* * * * action-encoder-layers	3	Number of transformer blocks used to encode actions
* * * * combiner-layers	1	Number of transformer blocks used in the combiner network that combines state and action encodings.
* * * * dropout-rate	0.1	Dropout rate.
* * * * num-head-layers	2	Number of layers in the feed-forward networks used for the value and policy heads.
* * * * head-dim	256	Hidden dimension of layers in the feed-forward networks used for the value and policy heads.
* * * mcts		MCTS hyperparameters.
* * * * num-simulations	32	Number of MCTS simulations used for planning every environment step.
* * * * num-considered-actions	8	The number of top-ranked actions that are considered in the sequential halving Gumbel exploration process.
* * * * value-scale	0.1	The c_{scale} hyperparameter (see original Gumbel AlphaZero paper).
* * * * max-visit-init	50	The c_{visit} hyperparameter (see original Gumbel AlphaZero paper).
* * * * fpu-red	0	Value estimates for unvisited children are decreased by this value so as to encourage deeper search trees (see LC0 AlphaZero implementation).
* * * * reset-tree	false	Whether the MCTS tree is reset after an environment move is taken.
* * * * max-tree-size	256	Maximal size of the MCTS tree. This parameter is relevant to avoid out-of-memory errors when `reset-tree` if false.
* * * * dirichlet-alpha	10	Concentration parameter for the Dirichlet exploration noise (see original AlphaZero paper).
* * * * dirichlet-eps	none	Magnitude of the Dirichlet exploration noise. No Dirichlet exploration noise is normally used with Gumbel AlphaZero but we add some anyway in the teacher for increased diversity.
* * * training		Network training hyperparameters.
* * * * max-epochs	1	The maximum number of training epochs.

* * * * improvement-required	1	Gradient updates are stopped if the validation loss fails to decrease for more than this number of epochs.
* * * * batch-size	300	Training batch size.
* * * * lr-base	0.0003	Maximal learning rate used in a cosine learning rate schedule with warm-up.
* * * * warmup-epochs	0.2	Length of the warm-up part of the learning rate schedule (in epochs).
* * * * weight-decay	0.01	Weight decay parameter.
* * * * outcome-loss-coeff	1	Coefficient for the loss term evaluating outcome prediction accuracy (e.g. success or failure).
* * * * event-loss-coeff	1	Coefficient for the loss term evaluating event prediction accuracy.
* * * * policy-loss-coeff	1	Coefficient for the loss term evaluating the divergence from policy priors to their targets.
* * * training-window	1, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6, 7	The number of previous iterations from which samples are used to update the network at each iteration. If the provided list is shorter than the total number of iterations, the last number in this list is duplicated as many times as necessary.
* num-teacher-iters-used-by-solver	5	The number of teacher training iterations from which solver training samples are collected.
* extra-teacher-problems	10000	Number of extra teacher problems that are generated once the teacher is trained, with no exploration noise.

Table 9: Training hyperparameters for the experiments described in section 2.4. Hyperparameters are organized according to a hierarchichal structure that is represented using indentation.

A.8 Network Architecture and Choice Points Encoding

A.8.1 Network Architecture

Neural networks that are used as oracles for nondeterministic strategies take as an input a *choice point* and return *i)* a probability distribution over all available choices, *ii)* some success and failure probability estimates and *iii)* some event occurence probability estimates (see section 2.2.4). In turn, a choice point is represented as *i)* a *probe* [16] that encodes all relevant state information provided to `choose` (see Figure 15 for examples) and *ii)* a list of possible choices that were passed as arguments to `choose`.

Our proposed network architecture works as follows. Given a choice point, the probe is encoded using a Dynamic Graph Transformer (DGT) neural network [35]. DGT networks are similar to Transformers [42] but they allow leveraging some additional graph structure over the source tokens by associating edge types to learned attention biases. Each choice is encoded separately using another DGT. It is then concatenated with the probe encoding and passed to a combiner network that outputs a score. Scores are normalized into a probability distribution using a softmax operation. The probe encoding is also passed to a value head that produces outcome and event predictions. Batches of choice points can be evaluated efficiently in parallel using scatter operations [85].

A.8.2 Encoding Programs and Formulas

All data that is to be passed to the neural network must be encodable into a sequence of tokens with an optional graph structure. This is the case of programs and formulas in particular. We use standard techniques to encode those [44, 86]:

- Abstract syntax trees (ASTs) are encoded into a sequence of tokens in polish notation order (e.g. $x + 3y$ is encoded as "`PLUS VAR(x) MUL CONST(3) VAR(y)`"). The edges in the original AST are preserved as graph edges to be passed to the DGT network.

Edge type	Description
PARENT	Connect any token to its parent in the syntax tree.
PREV_SIBLING	Connect any token to its previous sibling in the syntax tree.
PREV_LEXICAL_USE	Connect any token associated with name s to the previous occurence of s.
LAST_READ	Connect a variable occurence to places where it was possibly read last.
LAST_WRITE	Connect a variable occ. to places where it was possibly written last.
GUARDED_BY	Connect a variable occ. to assumptions made about it.
GUARDED_BY_NEG	Connect a variable occ. to negated assumptions made about it.
COMPUTED_FROM	Connect the lhs of any assignment to the variables in its rhs.
SAME_CONST	Connect two identical numerical constants.
SMALLER_CONST	Compare two different numerical constants.

Table 10: Edge types used to encode formulas and programs. We organize these edges into three groups: syntactic edges, semantic edges and numerical edges. Within a conditional statement, every variable in the `if` branch is connected to the guard using a `GUARDED_BY` edge whereas every variable in the `else` branch is connected to the guard using a `GUARDED_BY_NEG` edge.

- We use a different positional encoding scheme for tokens that leverages the tree structure of the underlying AST [86]. Doing so has been demonstrated to result in better generalization capabilities [87].
- Identifier names are randomly mapped into a finite number of unique ids for each choice point. Unique ids are encoded using a one-hot encoding scheme. If a choice point introduces more names than there are unique ids available (rare), the last occuring names are made to share a single uid that is flagged to indicate a naming conflict.
- Numerical constants with an absolute value greater than 3 are represented with generic `POS_CONST` and `NEG_CONST` tokens. A binary encoding of their value is also provided to the network. Moreover, special edges are added to compare all numerical constants involved in a program or formula (see Table 10).
- Following [44], we also add semantic edges to the encoding of programs and formulas that reflects the way information flows within them. Details can be consulted in Table 10.