A Details on the Solver Strategy

In Figure 7, we provide some detailed pseudocode for the solver strategy that we implemented for the purpose of the experiments in Section 4. This strategy is similar to the simple one introduced in Figure 2 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 7 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 2.5) 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 minimum total reward delivered in case of a success is $r_{\texttt{min}} = 0$.

Hints on how to use our proposed solver strategy to solve Code2Inv benchmark problems are available in Appendix C.

B Details on the Teacher Strategy

The teacher strategy we use for loop invariant generation follows the structure introduced in Figure 3. We provide additional details below:

- **Sampling constraints:** The full list of available constraints is available in Table 3. 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 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.

```
1  def solver(
2     init: Formula,
3     guard: Formula,
4     body: Program,
5     post: Formula) -> List[Formula]:
6
7     invs_proved: List[Formula] = []
8     constrs: List[Formula] = []
9     pending: List[Pending] = []
10
11    def prove_post():
12       pending[-1].status = TO_PROVE
13       to_prove = Implies(
14          constrs + invs_proved + [Not(guard)],
15          post)
16       match abduct(to_prove):
17          case Valid:
18             pending[-1].status = PROVED
19          case [*suggs]:
20             assum = suggest_missing(suggs)
21             closing = implies(assum, to_prove)
22             pending[-1].status = \
23                PROVED_COND if closing
24                else TO_PROVE_NEXT
25             prove_missing(assum, as_inv=True)
26             prove_post()
27
28    def suggest_missing(suggs: List[Formula]):
29       suggs += conjectures(guard, body)
30       num_disjs = choose([1, 2, 3])
31       disjs = []
32       for i in range(num_disjs):
33          d = choose(suggs)
34          disjs.append(d)
35          if is_conjecture(d):
36             event(CONJECTURING_EVENT)
37          else:
38             event(ABDUCTION_EVENT)
39       return Or(*disjs)
40
41    def prove_missing(f: Formula, as_inv: bool):
42       if meta_only(f):
43          constrs.append(f)
44          assert sat(constrs)
45       else:
46          assert as_inv
47          inv, fresh = strengthen(f)
48          for c, _ in fresh:
49             constrs.append(Ge(Metavar(c), 0))
50          pending.append(
51             Pending(INV, inv, TO_PROVE))
52          prove_init(inv)
53          prove_preserved(inv)
54          invs_proved.append(inv)
55          pending.pop()
56          for c, btype in fresh:
57             cval = abduct_refinement(
58                c, btype, constrs)
59             subst(invs_proved, c, cval)
60             subst(constrs, c, cval)
61
62    def prove_init(inv: Formula):
63       pending[-1].status = TO_PROVE
64       to_prove = Implies(
65          constrs + proved_invs + [init],
66          inv)
67       match abduct(to_prove):
68          case Valid: return
69          case [*suggs]:
70             assum = choose(suggs)
71             prove_missing(assum, False)
72
73    def prove_preserved(inv: Formula):
74       pending[-1].status = TO_PROVE
75       to_prove = Implies(
76          constrs +
77          proved_invs + [guard, inv],
78          loop_body.wlp(inv))
79       match abduct(to_prove):
80          case Valid: return
81          case [*suggs]:
82             suggs = suggest_missing(suggs)
83             closing = implies(assum, to_prove)
84             pending[-1].status = \
85                PROVED_COND if closing
86                else TO_PROVE_NEXT
87             perform(action)
88             prove_inv_inductive(inv)
89
90    pending.append(
91       Pending(POST, post, TO_PROVE))
92    prove_post()
93    pending.pop()
94    return invs_proved
```

Figure 7: 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 all variables that are 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.

- **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 4. 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 3 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 3).
- **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 5.

C Examples of Code2Inv Problems

We show examples of Code2Inv benchmark problems in Table 6, along with some hints on how they can be solved using the strategy detailed in Appendix A.

D Examples of Generated Teacher Problems

We show examples of challenge problems generated by our trained teacher agent in Table 7.

E Looprl UI Screenshots

We show examples of using the Looprl user interface to inspect the solver and teacher strategies in Figures 8 and 9 respectively.

F Implementing Abduction

Both the teacher and solver strategies we use as examples in this paper 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 [45].

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.

F.1 Case where $n = 1$

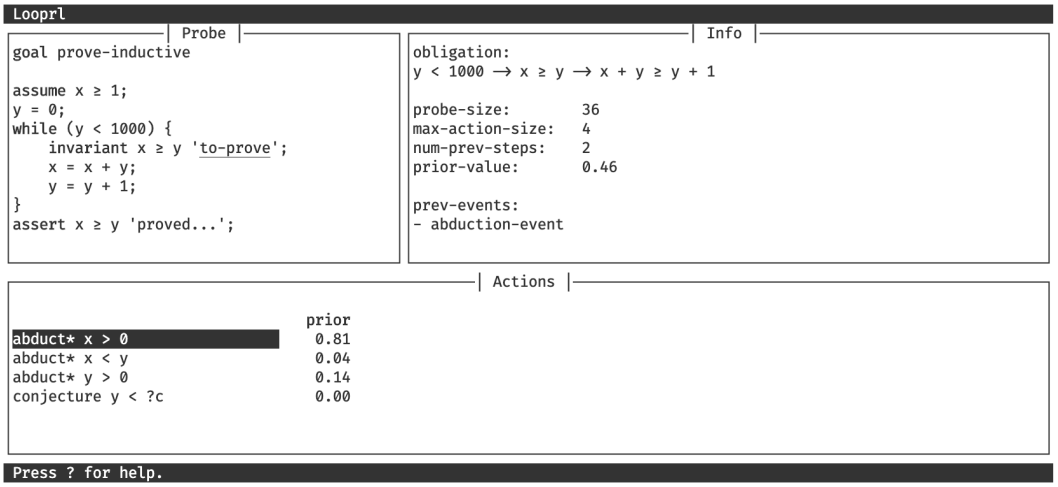
If F can be expressed as a disjunction of atomic formulas, we can compute abduction suggestions as follows: *i)* one considers the negation of F, obtaining a set of atomic assumptions and *ii)* one

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.

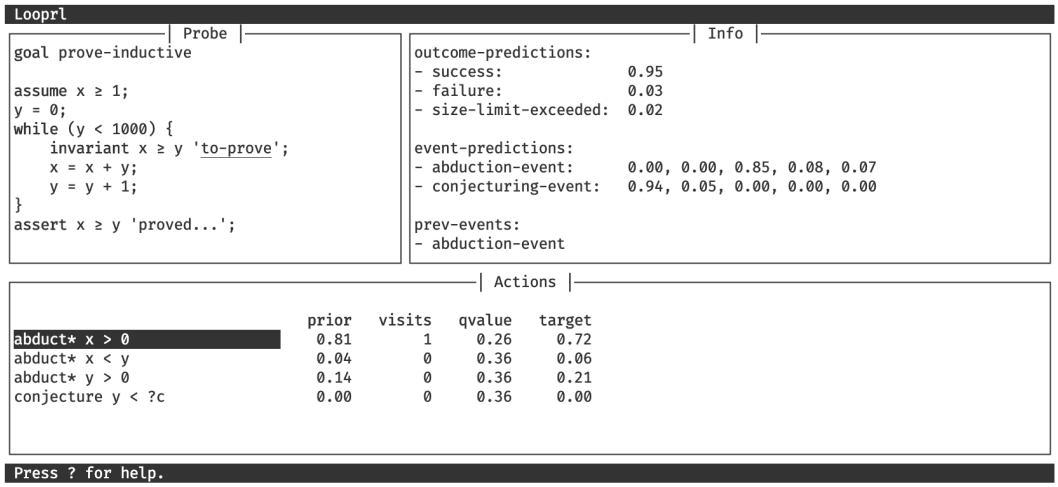
Table 3: Complete list of teacher constraints. Every constraint type is associated with a separate violation event. The associated reward is -0.5 for all constraint types except the last four ones where it is -0.2. A total reward of at least $r_{\texttt{min}} = -0.5$ is delivered in case of a success.

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 4: 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 8: 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 8b. 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 5: 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 $y > 0$.

F.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.

G Training Hyperparameters

We provide an exhaustive list of all hyperparameter values used in our experiments in Table 8.

`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 $i - s = c_?$, then using abduction to refine $c_?$ and finally abducting the disjunctive invariant $i \leq n + 1 \vee s = 0$ while trying to prove the postcondition.

Table 6: 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 network has been 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 7: 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.

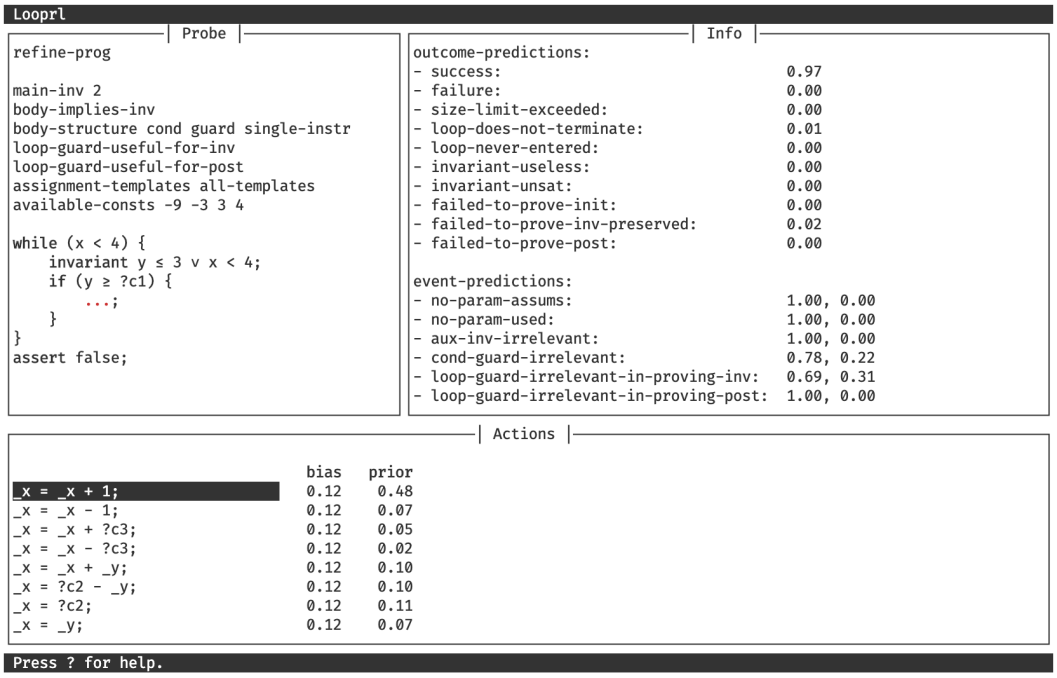


Figure 9: Visualizing the teacher strategy with the Looprl UI. This screenshot captures a choice point where the network is tasked with adding an assignment to the true branch of the conditional within the loop body. Several templates are proposed, which are going to be refined in turn. The upper left 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 relevant for 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 8: Training hyperparameters for the experiments described in Section 4. Hyperparameters are organized according to a hierarchichal structure that is represented using indentation.

H Network Architecture and Choice Points Encoding

H.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 2.5). In turn, a choice point is represented as *i)* a *probe* [10] that encodes all relevant state information provided to `choose` (see Figure 7 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 [32]. DGT networks are similar to Transformers [46] 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 [47].

H.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 [48, 49]:

- 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 9: 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 [49]. Doing so has been demonstrated to result in better generalization capabilities [50].
- 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 9).
- Following [48], we also add semantic edges to the encoding of programs and formulas that reflect the way information flows within them. Details can be consulted in Table 9.