Optimal Program Synthesis via Abstract Interpretation

STEPHEN MELL, University of Pennsylvania, USA
STEVE ZDANCEWIC, University of Pennsylvania, USA
OSBERT BASTANI, University of Pennsylvania, USA

We consider the problem of synthesizing programs with numerical constants that optimize a quantitative objective, such as accuracy, over a set of input-output examples. We propose a general framework for optimal synthesis of such programs in a given DSL, with provable optimality guarantees. Our framework enumerates programs in a general search graph, where nodes represent subsets of concrete programs. To improve scalability, it uses A^* search in conjunction with a search heuristic based on abstract interpretation; intuitively, this heuristic establishes upper bounds on the value of subtrees in the search graph, enabling the synthesizer to identify and prune subtrees that are provably suboptimal. In addition, we propose a natural strategy for constructing abstract transformers for monotonic semantics, which is a common property for components in neurosymbolic DSLs. Finally, we implement our approach in the context of two existing DSLs for data classification, demonstrating that our algorithm is more scalable than existing optimal synthesizers.

CCS Concepts: • **Theory of computation → Abstraction**; **Program analysis**; • **General and reference → General conference proceedings**; **Design**; • **Computing methodologies → Supervised learning**.

ACM Reference Format:
Stephen Mell, Steve Zdancewic, and Osbert Bastani. 2023. Optimal Program Synthesis via Abstract Interpretation. 1, 1 (November 2023), 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

There has been a great deal of recent interest in synthesizing programs to solve data processing and querying tasks, which often operate over semi-structured and unstructured data such as images and natural language text. Examples include *neurosymbolic programs* that incorporate deep neural network (DNN) components to extract semantic information from raw data [Chen et al. 2021; Shah et al. 2020], as well as *fuzzy matching programs* that use predicates with quantitative semantics to approximately match real-valued data [Mell et al. 2023]. For instance, Shah et al. [2020] synthesizes programs that label sequence data, Mell et al. [2023] synthesizes queries over trajectories output by an object tracker, and Chen et al. [2021] synthesizes web question answering programs. Most work focuses on programming by example (PBE), where the user provides a set of input-output (IO) examples, and the goal is to synthesize a program that generates the correct output for each input. There are two key properties that distinguish synthesis of such programs from traditional PBE:

- **Quantitative (real-valued) objectives:** The goal in neurosymbolic synthesis is typically to optimize a quantitative objective such as accuracy or F_1 score rather than to identify a program that is correct on all examples (which may be impossible).

Authors' addresses: Stephen Mell, sm1@cis.upenn.edu, University of Pennsylvania, Philadelphia, Pennsylvania, USA; Steve Zdancewic, stevez@cis.upenn.edu, University of Pennsylvania, Philadelphia, Pennsylvania, USA; Osbert Bastani, obastani@seas.upenn.edu, University of Pennsylvania, Philadelphia, Pennsylvania, USA.

XXXX-XXXX/2023/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- **Numerical constants:** Programs operating on fuzzy real-world data or the outputs of DNN components typically include real-valued constants that serve as thresholds; for example, when querying video trajectories, one constant may be a threshold on the maximum velocity of the object.

While these properties occasionally arise in traditional PBE settings (e.g., minimizing resource consumption), they are fundamental issues in neurosymbolic synthesis. Furthermore, in the neurosymbolic setting, there is often additional structure pertaining to the real-values that can be exploited to improve synthesis performance—for instance, some of the numerical components might be monotone in their inputs.

Most existing systems focus on synthesizing examples in a particular domain-specific language (DSL). In these settings, prior work has leveraged monotonicity of the semantics to prune the search space [Mell et al. 2023]. One general framework is Shah et al. [2020], which uses *neural relaxations* to guide search over a general DSL. At a high level, they use A^* search to enumerate partial programs in the DSL, which are represented as a directed acyclic graph (DAG). In general, A^* search prioritizes the order in which to enumerate partial programs based on a score function (called a *heuristic*) that maps each partial program to a real-valued score. When the heuristic is *admissible*—i.e., its output is an upper bound on the objective value for *any* completion of that partial program (assuming the goal is to maximize the objective)—then A^* search is guaranteed to find the optimal program (assuming it terminates).[1]

Then, Shah et al. [2020] proposes the following heuristic: fill each hole in the partial program with an untrained DNN, and then maximize the quantitative objective as a function of the DNN parameters. However, this score function is only guaranteed to be admissible under assumptions that typically do not hold in practice: (i) the neural relaxations are sufficiently expressive to represent any program in the DSL, which requires very large DNNs, and (ii) maximization of the DNN parameters converges to the global optimum, which does not hold for typical strategies such as stochastic gradient descent (SGD). Furthermore, SGD cannot handle non-differentiable objectives, which include common objectives such as accuracy and F_1 score.

Thus, a natural question is whether we can construct practical heuristics that are always guaranteed to be admissible. In this work, we take inspiration from *deduction-guided synthesis*, which uses automated reasoning techniques such as SMT solvers [Bornholt et al. 2016] or abstract interpretation [Cousot and Cousot 1977] to prune partial programs from the search space—i.e., prove that no completion of a given partial program can satisfy the given IO examples. In particular, we propose using abstract interpretation to construct heuristics for synthesis for quantitative objectives. Traditionally, abstract interpretation can be used to prune partial programs by replacing each hole with an abstract value overapproximating all possible concrete values that can be taken by that hole in the context of a given input. Then, if the abstract output does not include the corresponding concrete output, that partial program cannot possibly be completed into a program that satisfies that IO example, so it can be pruned.

Our key insight is that abstract interpretation can similarly be used to construct an admissible heuristic for a quantitative objective. Essentially, we can use abstract interpretation to overapproximate the possible objective values obtained by any completion of a given partial program; then, the supremum of concrete values represented by the abstract output serves as an upper bound of the objective, so it can be used as an admissible heuristic. Thus, given abstract transformers for the DSL components and for the quantitative objective, our framework can synthesize optimal programs.

[1]We mean optimal on the given IO examples. This notion ignores suboptimality due to generalization error, which can be handled using standard techniques from learning theory [Kearns and Vazirani 1994].

In addition, we propose general strategies for constructing abstract domains and transformers for common DSLs and objectives. As discussed above, many DSLs have monotone components. In these settings, a natural choice of abstract domain is to use intervals for the real-valued constants; then, a natural abstract transformer is to evaluate the concrete semantics on the upper and lower bounds of the intervals. This strategy can straightforwardly be shown to correctly overapproximate the concrete semantics.

We have implemented our approach in the context of two DSLs—namely, the NEAR [Shah et al. 2020] DSL for the CRIM13 [Burgos-Artizzu et al. 2012] dataset, and the Quivr [Mell et al. 2023] DSL and benchmark. In our experiments, we demonstrate that our approach significantly outperforms an adaptation of Metasketches [Bornholt et al. 2016]—an existing optimal synthesis framework based on SMT solving—to our setting, as well as an ablation that uses breadth first search instead of A^* search. Our approach significantly outperforms both of these baselines in terms of running time. In summary, our contributions are:

- We propose a novel algorithm for optimal synthesis which performs enumerative search over a space of *generalized partial programs*. To prioritize search, it uses the A^* algorithm with a search heuristic based on abstract interpretation. If it returns a program, then that program is guaranteed to be optimal (Section 3).
- In practice, many DSLs have components with monotone semantics—i.e., the concrete values have a partial order that is preserved by the concrete semantics. For these semantics, we propose to use intervals as the abstract domains, in which case a natural choice of abstract transformer is to simply apply the concrete semantics to the lower and upper bound of the interval (Section 4).
- We implement our framework in the context of two existing DSLs (Section 5) and show in our experiments that it outperforms Metasketches [Bornholt et al. 2016], a state-of-the-art optimal synthesis technique based on SMT solvers, and a baseline that uses breadth-first search instead of our search heuristic (Section 6).

2 MOTIVATING EXAMPLE

We consider a task where the goal is to synthesize a program for predicting the behavior of mice based on a video of them interacting [Shah et al. 2020]. This task is motivated by a data analysis problem in biology. In particular, biologists use mice as model animals to investigate both basic biological processes and to develop new therapeutic interventions, which sometimes requires determining the effect of an intervention on mouse behavioral patterns, including the nature and duration of interactions with other mice. For example, Figure 1 depicts two mice in an enclosure, engaging in the "sniff" behavior.

Doing this behavior analysis typically involves researchers viewing and manually annotating these behaviors in hours of video, which is very labor intensive. As a result, program synthesis has been applied to automating this task [Shah et al. 2020]. Their approach first uses an object tracker to track each mouse across frames in the video, producing trajectories represented as a sequence of 2D positions for each mouse. Then, they featurize each step in the trajectory—for instance, if there are two mice in the video, then one feature might be the distance between them in each frame. Based on this sequence of features, the goal is to predict a label for the behavior (if any) that the mice are engaged in during each frame (producing a sequence of labels, one for each frame). Shah et al. [2020] solves this problem by synthesizing a neurosymbolic program in a functional DSL for processing trajectories. In summary, the goal is to synthesize a program that takes as input a sequence of feature vectors, and outputs a sequence of labels. We consider the programming by

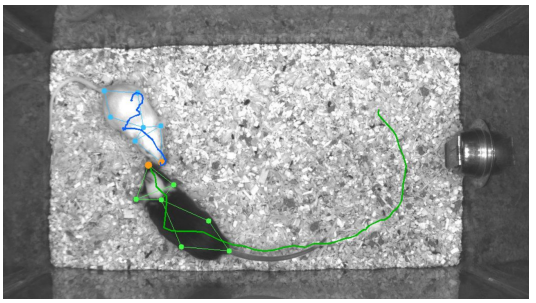


Fig. 1. A frame from a video of two mice interacting [Sun et al. 2021]; the mice are very close together, and are exhibiting the "sniff" behavior. The video has been processed using deep neural networks to produce certain keypoints, which are shown.

example (PBE) setting, where we are given a number of human annotated examples, and the goal is to synthesize a program that maximizes some objective, such as accuracy or F_1 score.

For example, consider synthesizing a program that, given a featurized trajectory representing two mice in a video, outputs the behavior of the mice at each time step. In particular, the input is a trajectory $x \in \mathbb{R}^*$ (where T^* denotes lists of Ts), where the feature $x[t] \in \mathbb{R}$ on time step t encodes the distance between the two mice at that time step, and the output is $y \in \{\mathbf{f}, \mathbf{t}\}^*$, where $y[t]$ encodes whether the mice are engaging in the "sniff" behavior ($y = \mathbf{t}$ if so, and $y = \mathbf{f}$ if not) at time step t.

We consider synthesizing such a program based on a single training example $x_1 = (100, 65)$ and $y_1 = (\mathbf{f}, \mathbf{t})$ (i.e., the first frame has mouse distance 100 and is labeled "not sniff" and the second frame has mouse distance 65 and is labeled "sniff"). Our goal is to find some program that classifies a dataset of videos well—i.e., if we evaluate the program on the videos to get predicted labels, and then compute a classification metric (e.g. accuracy) between the predicted and true labels, the discrepancy should be small. Given a program p, its accuracy is 1 if $[\![p]\!](x_1) = y_1$ and 1/2 if $[\![p]\!](x_1)[0] = y_1[0]$ but $[\![p]\!](x_1)[1] \neq y_1[1]$ (or vice versa), and 0 otherwise (where $z[i]$ is the i-th item in a sequence z).

Consider the candidate program "map$(d \leq 50)$". In traditional syntax, this program is map $(\lambda d\,.\,d \leq 50)\ x$, but the input is specified separately and the λ omitted (in combinator-style, similar to the NEAR DSL). This program performs a map over the sequence of frames, and for each one, it would output whether the mouse distance in that frame is less than or equal to 50. For example, when evaluated on x_1, it outputs

$$[\![\text{map}(d \leq 50)]\!](x_1) = (\mathbf{f}, \mathbf{f}).$$

Thus, its accuracy is 1/2, since it correctly labels the first frame but not the second.

One strategy for computing the optimal program (i.e., the one that maximizes the objective) is to enumerate partial programs (i.e., programs with *holes* representing pieces that need to filled to obtain a concrete program) in the DSL, evaluate the objective on every concrete program, and then choose the best one. There are several challenges to this approach:

- Unlike traditional synthesis, where we can stop enumerating when we reach a concrete program that satisfies the given specification, for optimal synthesis, we need to enumerate all programs or risk returning a suboptimal program.
- Traditional synthesizers use a variety of techniques to prune the search space to improve scalability. For instance, they might use deduction to prove that no completion of a partial

program can satisfy the specification—i.e., no matter how the holes in the partial program are filled, the specification will not hold. However, these techniques are not directly applicable to optimal synthesis.

- Synthesizing real-valued constants poses a problem: one approach is to discretize the these constants, enumerate all of them, and choose the best, but this approach can be prohibitively slow. For example, suppose we are enumerating completions of the partial program

$$\text{map}(d \leq ??),$$

where ?? is a *hole* that needs to be filled with a real value $\theta \in [0, 100]$ to obtain a concrete program. If we discretize $\theta \in \{0, 1, ..., 100\}$, then we would enumerate $\text{map}(d \leq 0)$, ..., $\text{map}(d \leq 100)$, evaluate each of these on (x_1, y_1) to measure its accuracy, and choose the program with the highest accuracy.

Our framework uses two key innovations to address these challenges:

- **Generalized partial programs:** Our framework takes the traditional notion of partial programs, representing sets of concrete programs as completions of syntax with holes, and extends it to more general sets of programs, equipped with a directed acyclic graph (DAG) structure.
- **A* search:** Rather than enumerate programs in an arbitrary order (e.g., breadth first search), our framework uses A^* search to enumerate programs.

We describe each of these techniques in more detail below.

Generalized partial programs. Traditionally, the search space over partial programs is a DAG, where the nodes are partial programs, and there is an edge $\hat{p} \longrightarrow \hat{p}'$ if $\hat{p}'$ can be obtained by filling a hole in $\hat{p}$ using some production in the DSL. For instance, there is an edge

$$\text{map}(d \leq ??) \longrightarrow \text{map}(d \leq 50)$$

in this DAG, since we have filled the hole with the value 50. However, even if we discretize the search space, there are 101 ways to fill this hole. As a consequence, if even a single completion of $\text{map}(d \leq ??)$ is valid, then we cannot prune it from the search space (ignoring for now the fact that we want to synthesize the optimal program instead of any valid program).

Instead, in our framework, we allow search DAGs beyond just programs with holes, so long as each node represents a set of concrete programs and the children of a node should collectively represent the same set. As a practical instantiation of the general framework, we consider partial programs where holes for real-valued constants may be annotated with constraints on the value that can be used to fill them. For instance, the generalized partial program

$$\text{map}(d \leq ??_{[50,100]})$$

represents the set of concrete programs

$$\{\text{map}(d \leq \theta) \mid \theta \in [50, 100]\}$$

Then, the children of this generalized partial program in the search DAG should split the constraint in a way that covers the search space—e.g.,

$$\text{children}(\text{map}(d \leq ??_{[50,100]})) = \{\text{map}(d \leq ??_{[50,75]}), \text{map}(d \leq ??_{[75,100]})\}.$$

This strategy presents more opportunities for pruning the search space. For instance, even if we cannot prune the program $\text{map}(d \leq ??_{[50,100]})$, we may be able to prune the program $\text{map}(d \leq ??_{[50,75]})$. Then, rather than needing to enumerate 51 programs (i.e., one for each $\theta \in \{50, ..., 100\}$), we would only need to prune $\text{map}(d \leq ??_{[50,75]})$ and evaluate 26 programs (i.e., one for each $\theta \in \{75, ..., 100\}$). Of course, we can further subdivide the search space to further reduce enumeration.

A^* search. Next, we describe our strategy for achieving the equivalent of pruning a partial program in the optimal synthesis setting. In particular, this task is accomplished by using A^* search. At a high level, A^* search enumerates nodes in a search graph according to a *heuristic*; for our purposes, a heuristic is function that maps a partial program $\hat{p}$ to a real value μ, and a heuristic is said to be *admissible* if it is an upper bound on the best possible objective value for any completion of $\hat{p}$—i.e.,

$$p \in \text{completions}(\hat{p}) \Rightarrow \mu \geq \text{objective}(p).$$

The heuristic adapts deductive reasoning to optimal synthesis. Whereas deductive reasoning guarantees that no completion of $\hat{p}$ can satisfy the given specification, the heuristic guarantees that no completion of $\hat{p}$ can achieve objective value greater than μ—e.g., if we find a concrete program with objective value $\geq \mu$, we can safely prune completions of $\hat{p}$ from the search DAG.

While A^* search has previously been used in synthesis for quantitative objectives [Shah et al. 2020], the heuristics used are not admissible and so do not provide theoretical guarantees. Our key contribution is showing that abstract interpretation can naturally be adapted to design admissible heuristics. Abstract interpretation can be used for traditional deductive reasoning as follows: fill each hole in the current partial program with an abstract value $\top$, evaluate the partial program using abstract semantics, and check if the abstract output is consistent with the specification.

In our example, a natural choice of abstract domain is the interval domain. In addition, rather than fill each hole with $\top$, if a hole has a constraint $??_{[a,b]}$, we can instead fill it with the interval $[a, b]$. For instance, for our example program $\text{map}(d \leq ??_{[50,75]})$, we can fill the hole with the interval $[50, 75]$ to obtain $\text{map}(d \leq [50, 75])$. Then, the abstract semantics $[\![\cdot]\!]^\#$ evaluate as follows:

$$[\![\text{map}(d \leq [50, 75])]\!]^\#(x_1) = ([\![100 \leq [50, 75]]\!]^\#, [\![65 \leq [50, 70]]\!]^\#) = (\mathbf{f}, \top). \tag{1}$$

In other words, the first element is $\mathbf{f}$ since $x_1 = (100, 65)$, and we know $100 \not\leq \theta$ for any $\theta \in [50, 75]$, and the second element is $\top$ since the relationship between 65 and $\theta \in [50, 75]$ can be either $\mathbf{f}$ or $\mathbf{t}$. Importantly, here the abstract values are over holes in the program rather than over the input to it.

Traditionally, we would then check whether this abstract output is consistent with the specification. For optimal synthesis, we observe that we can define an abstract semantics for the objective function. In our example, we can compute an "abstract accuracy" as follows (where $\mathbb{1}$ is a indicator function, and $\leq$, $\mathbb{1}$, $+$, $\cdot$, and $=$ are abstracted in the obvious way):

$$\begin{aligned}
\text{accuracy}^\#(\hat{p}) &= \frac{1}{2}\sum_{t=1}^{2} \mathbb{1}\left[[\![\text{map}(d \leq ??_{[50,75]})]\!]^\#(x_1)[t] = y_1[t]\right] \\
&= \frac{1}{2}\left(\mathbb{1}[\mathbf{f} = \mathbf{f}] + \mathbb{1}[(\top = \mathbf{t})\right) \\
&= \frac{1}{2}([1, 1] + [0, 1]) \\
&= [1/2, 1].
\end{aligned} \tag{2}$$

In other words, for the first frame, the concrete programs represented by $\text{map}(d \leq ??_{[50,75]})$ all predict $\mathbf{f}$, which equals $y_1[1]$, so this frame is always correctly classified. In contrast, for the second frame, concrete programs programs may output either $\mathbf{f}$ or $\mathbf{t}$, so we are uncertain whether this frame is correctly classified. Thus, the true accuracy is in the interval $[1/2, 1]$. Since abstract interpretation is guaranteed to overapproximate the semantics, we can use the upper bound of the abstract objective value as our heuristic—e.g., for $\text{map}(d \leq [50, 75])$, this heuristic computes $\mu = 1$.

Finally, we summarize the full search procedure starting from the node $\text{map}(d \leq ??_{[0,100]})$ in our search DAG, which has an abstract objective value of $[1/2, 1]$. First, we split it into

$$\hat{p}_1 = \text{map}(d \leq [0, 50]) \qquad \text{and} \qquad \hat{p}_2 = \text{map}(d \leq [50, 100]).$$

The abstract accuracies of $\hat{p}_1$ and $\hat{p}_2$ are $[1/2, 1/2]$ and $[1/2, 1]$, so their heuristic values are $\mu_1 = 1/2$ and $\mu_2 = 1$, respectively. Since $\mu_2 > \mu_1$, our algorithm explores $\hat{p}_2$ next, splitting it into

$$\hat{p}_3 = \text{map}(d \leq [50, 75]) \qquad \text{and} \qquad \hat{p}_4 = \text{map}(d \leq [75, 100]),$$

which have abstract accuracies of $[1/2, 1]$ and $[1, 1]$, respectively, and heuristic values of $\mu_3 = 1$ and $\mu_4 = 1$, respectively. In this example, the lower bound on the accuracy of $\hat{p}_4$ is also 1, so we know that any choice $\theta \in [75, 100]$ for filling this hole is guaranteed to achieve an accuracy of 1; thus, any concrete program $\text{map}(d \leq \theta)$ such that $\theta \in [75, 100]$ is optimal, and our algorithm can terminate without ever considering $\hat{p}_1$ or $\hat{p}_3$.

In general, our algorithm terminates once the range of possible optimal values is sufficiently small. For each node on the search frontier, we have an upper and lower bound on the objective value of all the programs it represents. The greatest of these lower bounds provides a lower bound on the best possible objective value, and the greatest of these upper bounds provides an upper bound on the best possible objective value. As a result, once the difference between the bounds is $\leq \epsilon$, we know that we have a program within ϵ of being optimal.

3 OPTIMAL SYNTHESIS VIA ABSTRACT INTERPRETATION

In this section, we consider the program synthesis problem where: (i) programs in the domain-specific language may have real-valued constants, and (ii) the synthesis objective is real-valued, where the goal is to return the optimal program (Section 3.1). Then, we describe our synthesis algorithm for solving this problem, which uses A^* search in conjunction with a search heuristic based on abstract interpretation (Section 3.2).

3.1 Problem Formulation

Domain-specific language. For concreteness, consider a DSL whose syntax is given by a context-free grammar $\mathcal{G} = (V, \Sigma, R, P)$, where V is the set of nonterminals, Σ is the set of terminals, $P \in V$ is the start symbol, and R is the set of productions

$$P ::= X \mid c \mid f(P, ..., P) \qquad (c \in \mathcal{C}, f \in \mathcal{F})$$

where X is a symbol representing the input, $c \in \mathcal{C}$ is a constant (including real-valued constants, i.e., $\mathbb{R} \subseteq \mathcal{C}$), and $f \in \mathcal{F}$ is a DSL component (i.e., function), but our framework extends straightforwardly to more general grammars. We let $p \in \mathcal{P} = \mathcal{L}(\mathcal{G}) \subseteq \Sigma^*$ denote the concrete programs in our DSL. Furthermore, we assume the DSL has denotational semantics $[\![\cdot]\!]$, where $[\![p]\!] : \mathcal{X} \to \mathcal{Y}$ maps inputs $x \in \mathcal{X}$ to outputs $y \in \mathcal{Y}$ according to the following rules:

$$[\![X]\!](x) = x, \quad [\![c]\!](x) = c, \quad [\![f(p_1, ..., p_k)]\!](x) = f([\![p_1]\!](x), ..., [\![p_k]\!](x)),$$

where we assume the functions $f : \mathcal{X}_1 \times ... \times \mathcal{X}_k \to \mathcal{Y}$ are given.

In Section 2, we considered programs like $\text{map}(d \leq 50)$, which we will use as a running example in this section. They are simplified versions of programs from the NEAR DSL, and are generated by the grammar:

$$E ::= d \mid c \mid \text{map}(E) \mid E \leq E \qquad (c \in \mathbb{R})$$

Task specification. We consider programming by example (PBE), where each task is specified by a set $Z \subseteq \mathcal{Z} = \mathcal{X} \times \mathcal{Y}$ of input-output (IO) examples, and the goal is to compute a program p^* that maximizes a given quantitative objective $\phi : \mathcal{P} \times \mathcal{Z} \to \mathbb{R}$

$$p^* \in \arg\max_{p \in \mathcal{P}} \phi(p, Z).$$

Often, ϕ is a function of the semantics applied to the examples $(x, y) \in Z$—i.e., there is a function $\phi_0 : (\mathcal{Y} \times \mathcal{Y})^*$ such that

$$\phi(p, Z) = \phi_0\left(\{(\llbracket p \rrbracket(x), y)\}_{(x,y) \in Z}\right), \tag{3}$$

In our running example, we choose ϕ_0 as follows, so $\phi(p, Z)$ is the accuracy of P on Z:

$$\phi_0(W) = \frac{1}{|W|} \sum_{(y', y) \in W} \mathbb{1}[y' = y],$$

where $\mathbb{1}[\cdot]$ is the indicator function.

Partial programs. A common strategy for PBE is to enumerate *partial programs*, which are programs in the DSL that have holes, to try and find one that satisfies the given IO examples. Intuitively, partial programs are partial derivations in the grammar $\mathcal{G}$. To formalize this notion, given two sequences $\hat{p}, \hat{p}' \in (\Sigma \cup V)^*$, we write $\hat{p} \to \hat{p}'$ if $\hat{p}'$ can be obtained by replacing a nonterminal symbol $N_i \in V$ in $\hat{p}$ with the right-hand side of a production $r = N_i \to M_1 ... M_k \in R$ for that nonterminal—i.e., $\hat{p} = N_1 ... N_i ... N_h$ and $\hat{p}' = N_1 ... M_1 ... M_k ... N_h$. We denote this relationship by $\hat{p}' = \text{fill}(\hat{p}, i, r)$—i.e., we obtain $\hat{p}'$ by filling the ith hole N_i in $\hat{p}$ using production r. Next, we write $\hat{p} \xrightarrow{*} \hat{p}'$ if there exists a sequence $\hat{p} = \hat{p}_1 \to ... \to \hat{p}_n = \hat{p}'$, and say $\hat{p}'$ can be *derived* from $\hat{p}$.

Note that concrete programs $p \in \mathcal{P}$ are sequences $p \in \Sigma^*$ that can be derived from the start symbol P (i.e., $P \xrightarrow{*} p$). Similarly, a partial program is a sequence $\hat{p} \in \hat{\mathcal{P}} \subseteq (\Sigma \cup V)^*$ that can be derived from P—i.e., $P \xrightarrow{*} \hat{p}$. The only difference is that $\hat{p}$ may contain nonterminals, which are called *holes*. The space of partial programs naturally forms a directed acyclic graph (DAG) via the relation $\hat{p} \to \hat{p}'$; note that concrete programs are leaf nodes in this DAG. Thus, we can perform synthesis by enumerating partial programs according to the structure of this DAG. Furthermore, given a partial program $\hat{p} \in \hat{\mathcal{P}}$ and a concrete program $p \in \mathcal{P}$, we say p is a *completion* of $\hat{p}$ if $\hat{p} \xrightarrow{*} p$ (i.e., p can be obtained from $\hat{p}$ by iteratively filling the holes of $\hat{p}$).

In our running example, traditional partial programs can be viewed as complete derivations in the following grammar, where ?? has been added as a terminal symbol:

$$\bar{E} ::= ?? \mid d \mid c \mid \text{map}(\bar{E}) \mid \bar{E} \geq \bar{E} \qquad (c \in \mathbb{R}).$$

Generalized partial programs. A key challenge is searching over real-valued constants $c \in \mathbb{R}$. Our grammar in theory contains infinitely many productions of the form $P \to \theta$ for $\theta \in \mathbb{R}$, and even if we discretize this search space, the number of productions is still large in practice.

To address this challenge, we propose a strategy where we enumerate *generalized partial programs* $\hat{p} \in \hat{\mathcal{P}}$, which generalize (i) the fact that partial programs correspond to sets of concrete programs (i.e., the set of their completions), and (ii) the DAG structure of partial programs.

Definition 3.1. A space of *generalized partial programs* is a set $\hat{\mathcal{P}}$ together with a *concretization function* $\gamma : \hat{\mathcal{P}} \to 2^{\mathcal{P}}$ and a *DAG structure* $\text{children} : \hat{\mathcal{P}} \to 2^{\hat{\mathcal{P}}}$, such that

$$\gamma(\hat{p}) = \bigcup_{\hat{p}' \in \text{children}(\hat{p})} \gamma(\hat{p}'). \tag{4}$$

Intuitively, $\gamma(\hat{p}) \subseteq \mathcal{P}$ is the set of concrete programs represented by the abstract program $\hat{p}$. In addition, children encodes a DAG structure on $\hat{\mathcal{P}}$ that is compatible with γ—i.e., the children $\hat{p}'$ of $\hat{p}$ must collectively contain all the concrete programs in $\hat{p}$.

For example, to capture traditional partial programs, we let

$$\gamma(\hat{p}) = \{p \in \mathcal{P} \mid \hat{p} \xrightarrow{*} p\},$$

i.e., p is a completion of $\hat{p}$, and

$$\text{children}(\hat{p}) = \{\hat{p}' \in \hat{\mathcal{P}} \mid \hat{p} \to \hat{p}'\}.$$

In Section 4, we will propose generalized partial programs that can include constraints on real-valued holes—e.g., $??_{[0,1]}$ is a partial program that can only be filled by a real value $\theta \in [0, 1]$.

Finally, a simple way to satisfy Definition 3.1 is to define γ based on the children function—i.e., we can define $p \in \gamma(\hat{p})$ if there exists a sequence $\hat{p} = \hat{p}_1, ..., \hat{p}_n = p$ of generalized partial programs such that $\hat{p}_{j+1} \in \text{children}(\hat{p}_j)$ for all $1 < j < n$. In other words, we can reach the concrete program p from the generalized partial program $\hat{p}$ in the search DAG. This strategy straightforwardly guarantees (4), since by definition, every $p \in \gamma(\hat{p})$ must be the descendant of some child of $\hat{p}$.

In our running example, the generalized partial programs correspond to the grammar

$$\hat{E} ::= ?? \mid X \mid ??_{[a,b]} \mid \text{map}(\hat{E}) \mid \hat{E} \geq \hat{E} \qquad (a, b \in \mathbb{R}),$$

while the concretization function satisfies

$$\gamma(\text{map}(d \leq ??_{[50,100]})) = \{\text{map}(d \leq c) \mid a \leq c \leq b\}$$

and the children function satisfies

$$\text{children}(\text{map}(d \leq ??_{[50,100]})) = \{\text{map}(d \leq ??_{[50,75]}), \text{map}(d \leq ??_{[75,100]})\}.$$

Abstract objective. For now, we consider an *abstract objective* $[\![\cdot]\!]^{\#}_{\phi}$ that directly maps generalized partial programs to abstract real values; it is typically constructed compositionally by providing abstract transformers for each component $f \in \mathcal{F}$ as well as for the objective function ϕ_0, and then composing them together. In particular, the abstract objective has type $[\![\hat{p}]\!]^{\#}_{\phi} : \mathcal{Z} \to \hat{\mathbb{R}}$, where $\hat{\mathbb{R}}$ is an abstract domain for the reals representing the potential objective values $\phi(p, Z)$ (e.g., the interval domain). Rather than require a concretization function for $\hat{\mathbb{R}}$, we only need an upper bound for this abstract domain—i.e., a function $\mu : \hat{\mathbb{R}} \to \mathbb{R}$, which encodes the intuition that "$\mu(\hat{r})$ is larger than any real number r contained in $\hat{r}$".

Definition 3.2. Given objective ϕ and generalized partial programs $(\hat{\mathcal{P}}, \gamma, \text{children})$, an abstract objective $([\![\cdot]\!]^{\#}_{\phi}, \mu)$ is *valid* if

$$\big(p \in \gamma(\hat{p})\big) \Rightarrow \big(\mu([\![\hat{p}]\!]^{\#}_{\phi}(Z)) \geq \phi(p, Z)\big) \qquad (\forall p \in \mathcal{P}), \tag{5}$$

and

$$\mu([\![p]\!]^{\#}_{\phi}(Z)) = \phi(p, Z) \qquad (\forall p \in \mathcal{P}). \tag{6}$$

Intuitively, (5) says that $\mu([\![\hat{p}]\!]^{\#}_{\phi}(Z))$ is an upper bound on the objective value $\phi(p, Z)$ for concrete programs p is contained in the abstract program $\hat{p}$. In addition, (6) says that for concrete programs, the abstract objective and concrete objective coincide.

Finally, a typical choice of $\hat{\mathbb{R}}$ is the space of intervals $\hat{\mathbb{R}} = \mathbb{R} \times \mathbb{R}$, where $(r, r') \in \hat{\mathbb{R}}$ represents the set of real numbers $\{r'' \in \mathbb{R} \mid r \leq r'' \leq r'\}$. Then, the upper bound is given by $\mu((r, r')) = r'$. The abstract objective $[\![\cdot]\!]^{\#}_{\phi}$ depends on the DSL; we describe a general construction in Section 4.

Algorithm 1 Our algorithm takes as input a task specification Z, along with a DSL $\mathcal{G}$, a space of generalized programs $(\hat{\mathcal{P}}, \gamma)$, abstract objective $(\llbracket\cdot\rrbracket^{\#}_{\phi}, \mu)$, objective lower bound ν, objective error tolerance ε, and returns the optimal program p^* for task Z. To do so, it uses the abstract objective $\llbracket\cdot\rrbracket^{\#}_{\phi}$ as a heuristic in A^* search, starting from the initial generalized partial program P.DEL: description of heap operations since they are out-of-date and they should be self-explanatory if someone knows what a "heap" is in the first place (I didn't) CHANGED ALGORITHM

procedure A^*-SYNTHESIS($Z; \mathcal{G}, \hat{\mathcal{P}}, \gamma, \llbracket\cdot\rrbracket^{\#}_{\phi}, \mu$)
  $h \leftarrow$ heap(sort_by = $\lambda\hat{p}.\mu(\llbracket\hat{p}\rrbracket^{\#}_{\phi}(Z))$)
  h.push(P)
  while true do
    $\hat{p} \leftarrow h$.pop()
    if $\max\{\mu(\hat{p}') \mid \hat{p}' \in h)\} - \max\{\nu(\hat{p}') \mid \hat{p}' \in h)\} \leq \varepsilon$ **then**
      return $\arg\max\{\nu(\hat{p}') \mid \hat{p}' \in h\}$
    end if
    for $\hat{p}' \in$ children($\hat{p}$) **do**
      h.push($\hat{p}'$)
    end for
  end while
end procedure

In our running example, the objective ϕ decomposes according to Equation 3 into a semantics $\llbracket\cdot\rrbracket$ and accuracy ϕ_0. We thus define the abstract objective $\llbracket\cdot\rrbracket^{\#}_{\phi}$ in terms of an abstract semantics $\llbracket\cdot\rrbracket^{\#} : \hat{\mathcal{P}} \to \hat{\mathbb{R}}$ (Equation 1), and abstract accuracy (Equation 2).

3.2 A^* Synthesis via Abstract Interpretation

Given a set of IO examples, Algorithm 1 uses A^* search over generalized partial programs $\hat{p} \in \hat{\mathcal{P}}$ in conjunction with the heuristic $\hat{p} \mapsto \mu(\llbracket\hat{p}\rrbracket^{\#}_{\phi}(Z))$ to compute the optimal program. In particular, it uses a heap h to keep track of the generalized partial program $\hat{p}$ in the frontier of the search DAG; at each iteration, it pops the current best node $\hat{p}$, and then enumerates the children $\hat{p}'$ of $\hat{p}$ and adds them to the heap according to the heuristic. Termination occurs when the objective lower bound $\nu : \hat{\mathbb{R}} \to \mathbb{R}$ (analagous to μ, but lower-bounding rather than upper-bounding the abstract objective) is within a tolerance $\varepsilon \geq 0$ of the upper bound μ. When when the objective abstraction $\hat{\mathbb{R}}$ is real intervals, $\nu((r, r')) = r$ is a natural choice.

For our running example, consider the following generalized partial programs and their abstract objective values:

$$\hat{p}_0 \coloneqq \text{map}(d \leq ??_{[0,100]}) \qquad \llbracket\hat{p}_0\rrbracket^{\#}_{\phi} = [1/2, 1]$$
$$\hat{p}_1 \coloneqq \text{map}(d \leq ??_{[0,50]}) \qquad \llbracket\hat{p}_1\rrbracket^{\#}_{\phi} = [1/2, 1/2]$$
$$\hat{p}_2 \coloneqq \text{map}(d \leq ??_{[50,100]}) \qquad \llbracket\hat{p}_2\rrbracket^{\#}_{\phi} = [1/2, 1]$$
$$\hat{p}_3 \coloneqq \text{map}(d \leq ??_{[50,75]}) \qquad \llbracket\hat{p}_3\rrbracket^{\#}_{\phi} = [1/2, 1]$$
$$\hat{p}_4 \coloneqq \text{map}(d \leq ??_{[75,100]}) \qquad \llbracket\hat{p}_4\rrbracket^{\#}_{\phi} = [1, 1]$$

The search process starts with $h_0 = \{(\hat{p}_0 : 1)\}$. In the first iteration, $\hat{p}_0$ is popped, since it (trivially) has the highest heuristic value. Its children, $\hat{p}_1$ and $\hat{p}_2$ are pushed, resulting in $h_1 = \{\hat{p}_1 : 1, \hat{p}_2 : 1/2\}$.

Here we can see the "soft pruning" of A^* at work: we know descendents of $\hat{p}_2$ cannot achieve objective values $> 1/2$. We can't prune $\hat{p}_2$ *per se*, since the optimal objective value may be $\leq 1/2$, but we can defer considering descendents of $\hat{p}_2$ until we know that the optimal objective value is $\leq 1/2$.

Next, we pop $\hat{p}_1$, since it has the highest heuristic value. Its children, $\hat{p}_3$ and $\hat{p}_4$ are pushed, resulting in $h_2 = \{\hat{p}_3, : 1, \hat{p}_4 : 1, \hat{p}_2 : 1/2\}$. Finally, we observe that the distance between the greatest lower bound (1) and greatest upper bound (1) of the abstract objective intervals is 0, and so search terminates, returning an arbitrary concrete descendent of $\hat{p}_4$.

We have the following optimality guarantee:

Theorem 3.3. *If Algorithm 1 returns a program p, then $p \in \arg\max_{p' \in \mathcal{P}} \phi(p', Z)$ is optimal.*

Proof. By (4), Algorithm 1 preserves the invariant that every program $p \in \mathcal{P}$ is contained in some generalized partial program $\hat{p} \in h$—i.e.,

$$\bigcup_{\hat{p} \in h} \gamma(\hat{p}) = \mathcal{P}.$$

This property can be checked by an easy induction argument on the while loop iteration.

Next, suppose Algorithm 1 terminates and returns $p \in \mathcal{P}$. For any concrete program $p' \in \mathcal{P}$ such that $p' \neq p$, let $\hat{p}' \in h$ be such that $p' \in \gamma(\hat{p}')$; such a $\hat{p}'$ exists by the above claim. Then, we have

$$\phi(p, Z) = \mu([\![p]\!]^{\#}_{\phi}(Z)) \geq \mu([\![\hat{p}']\!]^{\#}_{\phi}(Z)) \geq \phi(p', Z)$$

where the equality follows by (6), the first inequality follows by the heap property, and the second inequality follows by (5). The claim follows. □

We can ensure termination straightforwardly by using a finite DAG as the search space; for instance, we can do so by discretizing the real-valued constants. Even without discretization, we can terminate when the error interval becomes smaller than some error tolerance $\epsilon \in \mathbb{R}_{\geq 0}$. In general, we can guarantee convergence when the abstract losses of every infinite chain converge. For example, this property holds when the objective is Lipschitz continuous in the real-valued program parameters. (The gap between the upper and lower bounds of the objective value is bounded by the Lipschitz constant times the diameter of the box, which goes to zero as search proceeds and the boxes become smaller.) However, Lipschitz continuity often does not hold; we leave exploration of alternative ways to ensure convergence to future work.

4 INSTANTIATION FOR INTERVAL DOMAINS

Section 3 described a general framework for optimal synthesis when given an abstract semantics and search DAG for the target DSL. In this section, we describe a natural strategy for constructing abstract semantics when the concrete semantics are monotone. We begin by showing how to construct abstract semantics for individual components with monotone semantics (Section 4.1) and construct abstract transformers for monotone objectives (Section 4.2). Next, we describe a space of partial programs where holes corresponding to real-valued constants are optionally annotated with interval constraints (Section 4.3). Finally, we show how to combine the abstract transformers constructed in Sections 4.1 and 4.2 to perform abstract interpretation for our interval-constrained partial programs (Section 4.4).

4.1 Interval Transformers for Monotone Functions

Many DSLs have the property that parts of their concrete semantics are *monotone*—i.e., for an appropriate partial ordering on the inputs and outputs of a program statement, the concrete semantics preserve the partial order relationship.

Definition 4.1. Consider a function $f : \mathcal{X}_1 \times ... \times \mathcal{X}_k \to \mathcal{Y}$, and where $\mathcal{X}_1, ..., \mathcal{X}_k$ and $\mathcal{Y}$ are partially ordered sets. We say f is *monotone* if

$$\bigwedge_{i=1}^{k} x_i \leq x_i' \Rightarrow f(x_1, ..., x_k) \leq f(x_1', ..., x_k').$$

For example, the + operator is monotonically increasing in both of its inputs. We assume f is monotonically increasing without loss of generality; we can capture monotonically decreasing functions by considering the output space with the opposite partial order relation.

We describe a natural strategy for constructing abstract semantics for a monotone function f—i.e., abstract domains $\hat{\mathcal{X}}_i$ for each $\mathcal{X}_i$ and $\hat{\mathcal{Y}}$ for $\mathcal{Y}$, along with an abstract transformer $\hat{f} : \hat{\mathcal{X}}_1 \times ... \times \hat{\mathcal{X}}_k \to \hat{\mathcal{Y}}$. For our purposes, we only require each abstract domain to be a partial order together with an abstract transformer that overapproximates the concrete semantics.

Definition 4.2. Given a partially ordered set $\mathcal{Z}$, let $\bar{\mathcal{Z}} = \mathcal{Z} \cup \{-\infty, +\infty\}$, where $-\infty \leq z \leq +\infty$ for all $z \in \mathcal{Z}$. Then, the *interval domain* is the set $\hat{\mathcal{Z}} = \{(a, b) \in \bar{\mathcal{Z}}^2 \mid a \leq b\} \cup \{\bot\}$, together with an abstraction function $\alpha : \mathcal{Z} \to \hat{\mathcal{Z}}$ defined by $\alpha(z) = (z, z)$, and a concretization function $\gamma : \hat{\mathcal{Z}} \to 2^{\mathcal{Z}}$ defined by

$$\gamma((z_0, z_1)) = \{z \in \mathcal{Z} \mid z_0 \leq z \leq z_1\}.$$

In other words, γ maps (z_0, z_1) to the interval $[z_0, z_1] = \{z \in \mathcal{Z} \mid z_0 \leq z \leq z_1\}$. It is clear that $z \in \gamma(\alpha(z))$. Next, we define our notion of abstract transformer for the interval domain.

Definition 4.3. Given a monotone function $f : \mathcal{X}_1 \times ... \times \mathcal{X}_k \to \mathcal{Y}$, let $\hat{\mathcal{X}}_i$ be the interval domain for each $\mathcal{X}_i$ and $\hat{\mathcal{Y}}$ be the interval domain for $\mathcal{Y}$. The *interval transformer* for f is the function $\hat{f} : \hat{\mathcal{X}}_1 \times ... \times \hat{\mathcal{X}}_k \to \hat{\mathcal{Y}}$ defined by

$$\hat{f}((x_{1,0}, x_{1,1}), ..., (x_{k,0}, x_{k,1})) = (f(x_{1,0}, ..., x_{k,0}), f(x_{k,1}, ..., x_{k,1})).$$

If $x_{i,0} = -\infty$ for any $i \in [k]$, then we let the lower bound be $f(x_{1,0}, ..., x_{k,0}) = -\infty$, and if $x_{i,1} = +\infty$ for any $i \in [k]$, then we let the upper bound be $f(x_{k,1}, ..., x_{k,1}) = +\infty$.

For example, since $+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is monotone, then $\hat{+} : \hat{\mathbb{R}} \times \hat{\mathbb{R}} \to \hat{\mathbb{R}}$ is given by

$$(a, b)\hat{+}(c, d) = (a + c, b + d)$$

The following key result shows that $\hat{f}$ overapproximates the concrete semantics:

Lemma 4.4. *Let $f : \mathcal{X}_1 \times ... \times \mathcal{X}_k \to \mathcal{Y}$ be monotone, and let $\hat{f}$ be its interval transformer. For any $x_i \in \mathcal{X}_i$ and $\hat{x}_i = (x_{i,0}, x_{i,1}) \in \hat{\mathcal{X}}_i$ such that $x_i \in \gamma(\hat{x}_i)$ for all $i \in [k]$, we have*

$$f(x_1, ..., x_k) \in \gamma\left(\hat{f}(\hat{x}_1, ..., \hat{x}_k)\right).$$

Proof. For all $i \in [k]$, by our assumption that $x_i \in \gamma(\hat{x}_i)$, we have $x_{i,0} \leq x_i \leq x_{i,1}$. Thus, by monotonicity of f, we have

$$f(x_{1,0}, ..., x_{k,0}) \leq f(x_1, ..., x_k) \leq f(x_{1,1}, ..., x_{k,1}). \tag{7}$$

By definition of $\hat{f}$, we have $\hat{f}(\hat{x}_1, ..., \hat{x}_k) = (f(x_{1,0}, ..., x_{k,0}), f(x_{1,1}, ..., x_{k,1}))$, so by (7) and definition of γ, we have $f(x_1, ..., x_k) \in \gamma(\hat{f}(\hat{x}_1, ..., \hat{x}_k))$, as claimed. □

In other words, if x_i is contained in the interval $\hat{x}_i$ for each i, then $f(x_1, ..., x_k)$ is contained in the interval $\hat{f}(\hat{x}_1, ..., \hat{x}_k)$. Thus, $\hat{f}$ overapproximates the concrete semantics of f.

4.2 Interval Transfomers for Monotone Objectives

Similarly, if ϕ_0 is monotone—i.e., for $W_0 = \{(y'_{0,1}, y_{0,1}), ..., (y'_{0,k}, y_{0,k})\}$ and $W_1 = \{(y'_{1,1}, y_{1,1}), ..., (y'_{1,k}, y_{1,k})\}$, if $y'_{0,i} \leq y'_{1,i}$ for all $i \in [k]$, then we have

$$\phi_0(W_0) \leq \phi_0(W_1).$$

Note that we only require monotonicity in the labels $y'_{j,i}$ output by a candidate program p, not the ground truth labels $y_{j,i}$, since the latter are always concrete values. Then, we can construct the abstract transformer $\hat{\phi}_0 : (\hat{\mathcal{Y}} \times \mathcal{Y})^* \to \hat{\mathbb{R}}$ by

$$\hat{\phi}_0\left(\{((y'_{0,i}, y'_{1,i}), y_i)\}_{i=1}^k\right) = \left(\phi_0\left(\{(y'_{0,i}, y_i)\}_{i=1}^k\right), \phi_0\left(\{(y'_{1,i}, y_i)\}_{i=1}^k\right)\right).$$

4.3 Partial Programs with Interval Constraints

Next, we describe a space of generalized partial programs which extend partial programs with hole annotations that constrain the values that can be used to fill those holes.

Definition 4.5. Assume that the space of constants C is partially ordered, and let $\hat{C}$ be its interval domain. Then, an *interval-constrained partial program* $\tilde{p} = (\hat{p}, \kappa)$ is a partial program $\hat{p}$ together with a mapping $\kappa : \text{holes}(\hat{p}) \to \hat{C} \cup \{\varnothing\}$, where $\text{holes}(\hat{p}) \subseteq \mathbb{N}$ are the indices of the holes in $\hat{p}$.

For example, suppose that $\hat{p} = N_1 ... N_i ... N_h$ is a partial program, where N_i is a nonterminal (and therefore a hole), so $i \in \text{holes}(\hat{p})$. Intuitively, an annotation $\kappa(i) = \hat{c}$ imposes the constraint that the value used to fill N_i must be a constant $c \in C$, and that c must satisfy $c \in \gamma(\hat{c})$ (i.e., c is contained in the interval $[c_0, c_1]$). Alternatively, if $\kappa(i) = \varnothing$, then no such constraint is imposed—i.e., N_i may be filled with any constant or a different production in the grammar. (Note that $\varnothing$ is not the same as providing the interval $[-\infty, +\infty]$ because that constraint would require this hole to be filled by a *constant*, whereas the optimal program might need a non-constant expression.) We let $\tilde{\mathcal{P}}$ denote the space of interval-constrained partial programs.

These constraints are imposed by the structure of the search DAG, which is a extended version of the search DAG over partial programs. Below, we first describe the children function for interval-constrained partial programs; the concretization function is constructed from the children function.

Children function. Next, we describe the children of an interval constrained partial program $\text{children}(\tilde{p}) \subseteq \tilde{\mathcal{P}}$. Intuitively, if a hole N in a partial program $\hat{p}$ can be filled with a constant value $c \in C$ (i.e., there is a production $N \to c$) to obtain $\hat{p}'$, then $\hat{p}' \in \text{children}(\hat{p})$. In contrast, for an interval-constrained partial program $\tilde{p}$, we include a child annotating N with $[-\infty, \infty]$. Then, subsequent children can further split interval constraints to obtain finer-grained interval constraints (the concrete constant value c can be represented by the constraint $(c, c) \in \hat{C}$). To formalize this notion, we first separate out productions for constants.

Definition 4.6. A production $r \in R$ is *constant* if it has the form $r = N \to c$ for some $c \in C$.

Now, we can partition R into the set R_C of constant productions and its complement $\hat{R} = R \setminus R_C$, which we call *non-constant productions*. Next, given an interval-constrained partial program $\tilde{p} = (\hat{p}, \kappa)$, its holes are simply the holes of the underlying partial program $\hat{p}$: $\text{holes}(\tilde{p}) = \text{holes}(\hat{p})$. Then, we partition these holes into *unannotated holes* and *annotated holes*—i.e.,

$$\text{holes}_\varnothing((\hat{p}, \kappa)) = \{i \in \text{holes}(\hat{p}) \mid \kappa(i) = \varnothing\}$$
$$\text{holes}_C((\hat{p}, \kappa)) = \{i \in \text{holes}(\hat{p}) \mid \kappa(i) \neq \varnothing\},$$

respectively. Finally, we define three kinds of children for an interval-constrained partial program $\tilde{p}$. First, we include children obtained by filling an unannotated hole with a non-constant production:

$$\text{children}_{\varnothing}(\tilde{p}) = \{(\hat{p}', \kappa') \in \tilde{\mathcal{P}} \mid \exists i \in \text{holes}_{\varnothing}(\tilde{p}), r \in \hat{R} \,.\, \hat{p} = \text{fill}(\hat{p}, i, r) \wedge \kappa' = \text{repair}(\kappa; \hat{p}, i, r)\}.$$

These children are the same as the children constructed in the original search DAG over partial programs. Here, $\text{repair}(\kappa; \hat{p}, i, r)$ "repairs" κ by accounting for how the indices in $\hat{p}$ change after applying production r to fill hole i in $\hat{p}$. In particular, this operation changes the indices of nonterminals in $\hat{p}$; κ' accounts for these changes without modifying the annotations themselves. Formally, if $\hat{p}' = \text{fill}(\hat{p}, N_i \to M_1 ... M_h, i)$, with $\hat{p} = N_1 ... N_i ... N_k$ and $\hat{p}' = N_1 ... M_1 ... M_h ... N_k$, then we have

$$\kappa'(j) = \begin{cases} \kappa(j) & \text{if } j < i \\ \varnothing & \text{if } i \le j \le h \\ \kappa(j - h + 1) & \text{if } j > h. \end{cases}$$

In particular, κ' includes the same annotations as κ.

Second, we consider children obtained by filling an unannotated hole with the interval $[-\infty, \infty]$:

$$\text{children}_{\infty}(\tilde{p}) = \{(\hat{p}', \kappa') \in \tilde{\mathcal{P}} \mid \exists i \in \text{holes}_{\varnothing}(\tilde{p}) \,.\, \hat{p}' = \hat{p} \wedge \kappa'(i) = [-\infty, \infty]\}.$$

In other words, the partial program $\hat{p}$ remains unchanged, but we introduce an annotation onto one of the previously unannotated holes of $\tilde{p}$.

Third, we consider children obtained by replacing an annotation with a tighter annotation:

$$\text{children}_{C}(\tilde{p}) = \{(\hat{p}', \kappa') \in \tilde{\mathcal{P}} \mid \exists i \in \text{holes}_{C}(\tilde{p}) \,.\, \hat{p}' = \hat{p} \wedge \text{subset}(\kappa'(i), \kappa(i))\}.$$

Here, $\text{subset}(\hat{c}, \hat{c}')$ checks whether $\hat{c} = (c_0, c_1)$ is a *strict* subset of $\hat{c}' = (c_0', c_1')$, that is, $c_0' \le c_0$ and $c_1 \le c_1'$, and one of these inequalities is strict; equivalently, $[c_0, c_1] \subsetneq [c_0', c_1']$.

Finally, our overall search DAG is defined by the union of these children:

$$\text{children}(\tilde{p}) = \text{children}_{\varnothing}(\tilde{p}) \cup \text{children}_{\infty}(\tilde{p}) \cup \text{children}_{C}(\tilde{p}). \tag{8}$$

Note that by defining children in this way, there may be infinitely many children; in addition, multiple children may cover the same concrete program, leading to redundancy in the search DAG. Practical implementations can restrict to a finite subset of these children as long as they satisfy (4)—i.e., the union of the concrete programs in the children of $\tilde{p}$ cover all concrete programs in $\tilde{p}$. In addition, these children are ideally chosen so the overlap is minimal.

Concretization function. Recall that the concretization function $\gamma(\tilde{p})$ contains concrete programs p represented by $\tilde{p}$. We take the approach where we define γ based on the children function—i.e., $p \in \gamma(\tilde{p})$ if there exists a sequence $\tilde{p}_1, ..., \tilde{p}_n = p$ such that $\tilde{p}_1 = \tilde{p}$, $\tilde{p}_n = p$, and $\tilde{p}_{j+1} \in \text{children}(\tilde{p}_j)$ for all $1 < j < n$.

4.4 Interval Transformers for Partial Programs with Interval Constraints

Next, we describe how to implement abstract interpretation for partial programs with interval constraints. While abstract interpretation is typically performed with respect to program inputs, in our case it is with respect to program constants. We assume all components $f \in \mathcal{F}$ have an abstract transformer $\hat{f}$, and the objective ϕ_0 has an abstract transformer $\hat{\phi}_0$ (if they are monotone, their abstract transformers can be defined as in Section 4.1). First, we extend the grammar of programs so that the constants C include abstract values $(c_0, c_1) \in \hat{C}$—i.e., $C \leftarrow C \cup \hat{C}$. While the concrete semantics cannot be applied to these programs, we will define abstract semantics for them.

Now, given a generalized partial program $\tilde{p} = (\hat{p}, \kappa)$, for each unannotated hole $i \in \text{holes}_{\varnothing}(\tilde{p})$, we replace the corresponding nonterminal N_i in $\hat{p}$ with the abstract value $(-\infty, \infty)$, and for each annotated hole $i \in \text{holes}_{C}(\tilde{p})$, we replace the corresponding nonterminal N_i with the annotation

$\kappa(i)$. Finally, we replace any constant c in $\hat{p}$ with the abstract value (c, c). Once we have performed this transformation, we can define the following abstract semantics for $\hat{p}$:

$$[\![X]\!]^\#(x) = \alpha(x), \quad [\![\hat{c}]\!]^\#(x) = \hat{c}, \quad [\![f(p_1, ..., p_k)]\!]^\#(x) = \hat{f}([\![p_1]\!]^\#(x), ..., [\![p_k]\!]^\#(x)),$$

Now, we can combine $[\![\cdot]\!]^\#$ with $\hat{\phi}_0$ to obtain the abstract objective $[\![\cdot]\!]^\#_{\hat{\phi}}$:

$$[\![(\hat{p}, \kappa)]\!]^\#_{\hat{\phi}}(Z) = \hat{\phi}_0\left(\{([\![\hat{p}]\!]^\#(x), y)\}_{(x,y)\in Z}\right).$$

In other words, we apply the abstract semantics $[\![\hat{p}]\!]^\#$ to each input x, obtain the corresponding abstract output $\hat{y}'$, and apply $\hat{\phi}_0$ to the resulting set $\{(\hat{y}', y)\}$.

5 IMPLEMENTATION

We instantiate our framework for two different domain specific languages (DSLs):

- **NEAR DSL:** The NEAR language for the CRIM dataset, described in Section 2 (Section 5.1).
- **Quivr DSL:** A query language over video trajectories, which uses constructs similar to regular expressions to match trajectories (Section 5.2).

Although both of these DSLs process sequence data, their computation models are quite different: NEAR focuses on folding combinators over the inputs, whereas Quivr's primary operations reduce to matrix multiplication.

For both DSLs, we refine the definition of children compared to Equation 8. In particular, in Section 4.3, when defining the search space over programs with interval constraints on holes, children_C is defined such that the children of $[a, b]$ are *all* of its strict subintervals. As discussed there, this means that each node has (potentially) infinitely many children, which is impractical for an implementation. Instead, our implementation splits intervals into just two children—i.e., the children of $[a, b]$ are $[a, (a+b)/2]$ and $[(a+b)/2, b]$. These intervals partition $[a, b]$, so all concrete programs are still contained in the search space, retaining soundness. This splits an interval into child intervals of equal length, but with additional domain-knowledge, other choices could be made (e.g. with a prior distribution over parameters, splitting into intervals of equal probability mass).

Finally, we also describe how we construct the abstract transformer for the F_1 score, which is commonly used as the objective function in binary classification problems (Section 5.3).

5.1 NEAR DSL for Trajectory Labeling

In this DSL, the program input is a featurized trajectory, which is a sequence of feature vectors $z \in (\mathbb{R}^n)^*$, where $\cdot^*$ denotes sequences of any length and n is the dimension of the feature vector for each frame. The output is a sequence of labels $y \in \{\mathbf{t}, \mathbf{f}\}^*$ of the same length as the input, where $y[t] = \mathbf{t}$ if a frame exhibits the given behavior and $y[t] = \mathbf{f}$ otherwise (i.e., the task is binary classification at the frame level).

Syntax. This DSL has three kinds of expressions, encoded by their corresponding nonterminal in $\mathcal{G}$: (i) vv represents functions mapping feature vectors to real values (e.g., used inside a fold operator), (ii) ℓv represents functions mapping lists to real values (e.g., fold), and (iii) $\ell\ell$ represents functions mapping lists to lists (e.g., map). In particular, this DSL has the following productions:

$$\begin{aligned} vv &::= z_i \mid z_f \mid c \in \mathbb{R} \mid vv + vv \mid vv \cdot vv \mid \text{ite}(vv, vv, vv) \qquad (i \in [n]) \\ \ell v &::= \text{fold}(vv) \mid \text{ite}(\ell v, \ell v, \ell v) \\ \ell\ell &::= \text{map}(vv) \mid \text{mapprefix}(\ell v) \mid \text{ite}(\ell v, \ell\ell, \ell\ell), \end{aligned}$$

The DSL syntax is in the combinatory style, so λs are omitted. In particular, the expressions encoded by vv are combinators designed to be used within a higher-order function such as map or fold. These

combinators are applied to individual elements of the input list, where z_i is a variable representing the ith element of the current feature vector $z = x[t]$, and z_f is a special symbol used inside fold to represent its accumulated running state.

The start symbol is $\ell\ell$. We impose a type constraint that a program in this DSL maps a list of feature vectors x to a list of real values r of the same length. Each real-valued output $r[t]$ implicitly encodes the label

$$y[t] = \begin{cases} \mathbf{t} & \text{if } r[t] \geq 0 \\ \mathbf{f} & \text{otherwise.} \end{cases}$$

The running example involved programs in a toy DSL of the form $\text{map}(d \leq c)$. In the NEAR DSL, this would be represented as $\text{map}(-1 \cdot z_4 + c)$, as "distance between mice" is the 4th feature, and the label will be obtained by comparing the program output to 0.

Semantics. We let $V = \mathbb{R}^n$ (where n is the dimension of the feature space) be the space of feature vectors, $L = V^*$ is the space of trajectories, and $K = \mathbb{R}^*$ is the space of output sequences. Then, the nonterminal vv in this DSL has semantics $[\![vv]\!] : V \times \mathbb{R} \to \mathbb{R}$, where

$$\begin{aligned}
[\![z_i]\!](v, s) &:= v_i \\
[\![z_f]\!](v, s) &:= s \\
[\![c]\!](v, s) &:= c \\
[\![vv_1 + vv_2]\!](v, s) &:= [\![vv_1]\!](v, s) + [\![vv_2]\!](v, s) \\
[\![vv_1 \cdot vv_2]\!](v, s) &:= [\![vv_1]\!](v, s) \cdot [\![vv_2]\!](v, s) \\
[\![\text{ite}(vv_1, vv_2, vv_3)]\!](v, s) &:= \text{if } ([\![vv_1]\!](v, s) \geq 0) \text{ then } [\![vv_2]\!](v, s) \text{ else } [\![vv_3]\!](v, s).
\end{aligned}$$

Next, the nonterminal ℓv in this DSL has semantics $[\![\ell v]\!] : V^* \to \mathbb{R}$, where

$$\begin{aligned}
[\![\text{fold}(vv)]\!](\ell) &:= \text{fold } (\lambda v \lambda s.\ [\![vv]\!](v, s))\ \ell\ 0 \\
[\![\text{ite}(\ell v_1, \ell v_2, \ell v_3)]\!](\ell) &:= \text{if } ([\![\ell v_1]\!](\ell) \geq 0) \text{ then } [\![\ell v_2]\!](\ell) \text{ else } [\![\ell v_3]\!](\ell).
\end{aligned}$$

Here, $\text{fold} : (V \to \mathbb{R} \to \mathbb{R}) \to V^* \to \mathbb{R} \to \mathbb{R}$ is standard, and passes the intermediate value as the s argument to vv.

The nonterminal $\ell\ell$ in this DSL has semantics $[\![\ell\ell]\!] : L \to K$, where

$$\begin{aligned}
[\![\text{map}(vv)]\!](\ell) &:= \text{map } (\lambda v.\ [\![vv]\!](v, 0))\ \ell \\
[\![\text{mapprefixes}(\ell v)]\!](\ell) &:= \text{map } (\lambda \ell'.\ [\![\ell v]\!](\ell'))\ \text{prefixes}(\ell) \\
[\![\text{ite}(\ell v, \ell\ell_1, \ell\ell_2)]\!](\ell) &:= \text{if } ([\![\ell v]\!](\ell) \geq 0) \text{ then } [\![\ell\ell_1]\!](\ell) \text{ else } [\![\ell\ell_2]\!](\ell).
\end{aligned}$$

Here, map is standard[2]

$$\text{prefixes} : (x_1, \ldots, x_n) \mapsto ((x_1), (x_1, x_2), (x_1, x_2, x_3), \ldots (x_1, \ldots, x_n)).$$

Finally, as described above, the labels y are obtained by thresholding $[\![\ell\ell]\!](\ell)$. That is, we let $[\![\ell\ell]\!]^b : L \to \{\mathbf{t}, \mathbf{f}\}^*$ denote the label

$$([\![\ell\ell]\!]^b(\ell))[t] = \begin{cases} \mathbf{t} & \text{if } ([\![\ell\ell]\!](\ell))[t] \geq 0 \\ \mathbf{f} & \text{otherwise.} \end{cases}$$

[2]But note that z_f is treated as 0 if it appears inside an expression not within fold

Abstract semantics. We abstract $\mathbb{R}$ with the usual real intervals, $\hat{\mathbb{R}}$. We abstract K with $(\hat{\mathbb{R}})^*$, products of real intervals. Addition is monotone, and multiplication $[a_1, b_1] \cdot [a_2, b_2]$ can be shown to be tightly abstracted by $[\min(a_1b_1, a_1b_2, a_2b_1, a_2b_2), \max(a_1b_1, a_1b_2, a_2b_1, a_2b_2)]$. For $\mathbb{R}$-valued conditionals, we define $\widehat{\text{ite}}(c, a, b) = \mathbb{1}[c \geq 0] \cdot a + \mathbb{1}[c < 0] \cdot b$, and thus can abstract vv-level comparison using addition, multiplication, and the abstraction sending $\mathbb{1}[[\mathbf{f}, \mathbf{f}]] = [0, 0]$, $\mathbb{1}[[\mathbf{f}, \mathbf{t}]] = [0, 1]$, and $\mathbb{1}[[\mathbf{t}, \mathbf{t}]] = [1, 1]$.

Search space. The search space over vv has a redundancy due to commutativity, associativity, and distributativity, which unduly hinders search. Instead, we use a *normalized* version, where vv expressions are constrainted to be sums-of-products (fully distributed), and where we ignore commutativity and associative in the sums of products. Essentially, we only consider polynomials, where the variables are the features z_i, the fold variable z_f, and indicator variables $\mathbb{1}[vv \geq 0]$ for each vv (to maintain the expressiveness of ite). Further, we consider only constants in $[-1, 1]$ and we also normalize the dataset so x_i in $[-1, 1]$.

We define a notion of size for programs, so that we can bound the search space, where ite, map and mapprefix have size 1, fold has size 0 (since mapprefix must contain a fold and already has size 1). Each monomial in the polynomial size cost 1, and each polynomial variable in a monomial has size 1. In a given polynomial, the size of indicators is amortized: each nested vv has size 1 and produces a new polynomial variable.

5.2 Quivr DSL For Trajectory Queries

In the Quivr DSL [Mell et al. 2023], the program input is a featurized trajectory, which is a sequence $z \in (\mathbb{R}^n)^*$ as before. However, the output is now a single label $y \in \{\mathbf{t}, \mathbf{f}\}$ for the entire trajectory. This DSL is designed to allow the user to select trajectories that satisfy certain properties—e.g., the user may want to identify all cars that make a right turn at a certain intersection in a traffic video.

Syntax. This DSL is based on the Kleene algebra with tests [Kozen 1997], which, intuitively, are regular expressions where the "characters" are actually predicates. Its syntax is

$$\begin{aligned} Q &::= f \mid g(C) \mid Q\,;\,Q \mid Q \wedge Q \qquad (f \in \mathcal{F}_\varnothing, g \in \mathcal{F}_C) \\ C &::= c \qquad (c \in \mathbb{R}) \end{aligned}$$

where $\mathcal{F}_\varnothing$ and $\mathcal{F}_C$ are given sets of domain-specific predicates, the latter of which have constants $c \in \mathbb{R}$ that need to be chosen by the synthesizer.

Semantics. Expressions in this DSL denote functions mapping sequences of feature vectors $z \in (\mathbb{R}^n)^*$ to whole-sequence labels ($\{\mathbf{t}, \mathbf{f}\}$), defined as

$$\begin{aligned} [\![f]\!](z) &:= f(z) \\ [\![f(c)]\!](z) &:= \mathbb{1}[f(z) \geq c] \\ [\![Q_1 \wedge Q_2]\!](z) &:= [\![Q_1]\!](z) \wedge [\![Q_2]\!](z) \\ [\![Q_1\,;\,Q_2]\!](z) &:= \bigvee_{k=0}^{n} [\![Q_1]\!](z_{0:k}) \wedge [\![Q_2]\!](z_{k:n}) \end{aligned}$$

Each predicate $f \in \mathcal{F}_\varnothing$ has type $f : (\mathbb{R}^n)^* \to \{\mathbf{t}, \mathbf{f}\}$, and indicates whether z matches f, and each predicate $f \in \mathcal{F}_C$ has type $f : (\mathbb{R}^n)^* \to \mathbb{R}$, and produces a real-valued score thresholded at a given constant c to indicate whether z matches f.

Abstract semantics. Note that under the standard orders for reals $\mathbb{R}$ and Booleans $\mathbb{B} = \{\mathbf{t}, \mathbf{f}\}$ (i.e., $\mathbf{f} < \mathbf{t}$), the semantics $[\![f(c)]\!](z)$ is monotone decreasing with respect to c. Furthermore, both

conjunction and disjunction are monotone increasing in their inputs. Thus, we can use our interval transformers for monotone functions from Section 4.4. In particular, recall that $\hat{\mathbb{R}}$ is the abstract domain of real intervals, and let $\hat{\mathbb{B}}$ be the abstract domain of Boolean intervals

$$\hat{\mathbb{B}} = \{(\mathbf{f},\mathbf{f}), (\mathbf{f},\mathbf{t}), (\mathbf{t},\mathbf{t})\}.$$

Then, we use the abstract transformers

$$\mathbb{1}[f(z) \geq (c_0, c_1)] = \begin{cases} (\mathbf{f},\mathbf{f}) & \text{if } f(z) < c_0 \\ (\mathbf{f},\mathbf{t}) & \text{if } c_0 \leq f(z) < c_1 \\ (\mathbf{t},\mathbf{t}) & \text{if } c_1 \leq f(x) \end{cases}$$

$$\hat{x}_1 \hat{\wedge} \hat{x}_2 = \begin{cases} (\mathbf{t},\mathbf{t}) & \text{if } \hat{x}_1 = (\mathbf{t},\mathbf{t}) \wedge \hat{x}_2 = (\mathbf{t},\mathbf{t}) \\ (\mathbf{f},\mathbf{f}) & \text{if } \hat{x}_1 = (\mathbf{f},\mathbf{f}) \vee \hat{x}_2 = (\mathbf{f},\mathbf{f}) \\ (\mathbf{f},\mathbf{t}) & \text{otherwise} \end{cases}$$

$$\hat{x}_1 \hat{\vee} \hat{x}_2 = \begin{cases} (\mathbf{t},\mathbf{t}) & \text{if } \hat{x}_1 = (\mathbf{t},\mathbf{t}) \vee \hat{x}_2 = (\mathbf{t},\mathbf{t}) \\ (\mathbf{f},\mathbf{f}) & \text{if } \hat{x}_1 = (\mathbf{f},\mathbf{f}) \wedge \hat{x}_2 = (\mathbf{f},\mathbf{f}) \\ (\mathbf{f},\mathbf{t}) & \text{otherwise,} \end{cases}$$

Since sequencing is defined in terms of conjunction and disjunction, these transformers suffice to define the abstract semantics.

5.3 Abstract F_1 Score

In Section 5, we described how to construct abstract transformers for monotone functions, which covers many of the components in these DSLs. However, many objectives commonly used in practice, such as the F_1 score, are non-monotone. For non-monotone objectives, we need to provide a custom abstract transformer that overapproximates their concrete semantics for the interval domain. We describe how to do so for the F_1 score. Let W be the multiset of outcomes of the form (y', y) where y' is the prediction and y is the ground truth. Then the F_1 score is given by:

$$TP(W) \coloneqq \sum_{(y',y)\in W^+} \mathbb{1}[y = \mathbf{t} \wedge y' = \mathbf{t}]$$

$$FP(W) \coloneqq \sum_{(y',y)\in W^-} \mathbb{1}[y = \mathbf{f} \wedge y' = \mathbf{t}]$$

$$F_1(W) \coloneqq 2 \cdot \frac{TP(W)}{TP(W) + FP(W) + |W^+|},$$

where $|W^+| = |\{(y', y) \in W \mid y = \mathbf{t}\}|$, TP is the number of true positives, and FP is the number of false positives. Note that TP and FP are monotone in y', so we can use the corresponding interval transformers:

$$TP^\#(\hat{W}) \coloneqq \sum_{(\hat{y}',y)\in \hat{W}} \begin{cases} [1,1] & \text{if } \mathbb{1}[y = \mathbf{t} \wedge \hat{y}' = (\mathbf{t},\mathbf{t})] \\ [0,1] & \text{if } \mathbb{1}[y = \mathbf{t} \wedge \hat{y}' = (\mathbf{f},\mathbf{t})] \\ [0,0] & \text{if } \mathbb{1}[y = \mathbf{t} \wedge \hat{y}' = (\mathbf{f},\mathbf{f})] \end{cases}$$

$$FP^\#(\hat{W}) \coloneqq \sum_{(\hat{y}',y)\in \hat{W}} \begin{cases} [1,1] & \text{if } \mathbb{1}[y = \mathbf{f} \wedge \hat{y}' = (\mathbf{t},\mathbf{t})] \\ [0,1] & \text{if } \mathbb{1}[y = \mathbf{f} \wedge \hat{y}' = (\mathbf{f},\mathbf{t})] \\ [0,0] & \text{if } \mathbb{1}[y = \mathbf{f} \wedge \hat{y}' = (\mathbf{f},\mathbf{f})]. \end{cases}$$

Letting $TP^{\#}(W) = [a_1, b_1]$ and $FP^{\#}(W) = [a_2, b_2]$, and noting that for positive numbers we can abstract division as $[a_1, b_1] \hat{/} [a_2, b_2] = \left[\frac{a_1}{b_2}, \frac{b_1}{a_2}\right]$, a naïve strategy of applying abstractions for $+$, $\cdot$, and $/$ leads to the abstract F_1 score

$$F_1^{\#}(W) \coloneqq 2 \cdot \frac{[a_1, b_1]}{[a_1, b_1] + [a_2, b_2] + |W^+|} = 2 \cdot \left[\frac{a_1}{b_1 + b_2 + |W^+|}, \frac{b_1}{a_1 + a_2 + |W^+|}\right],$$

where $|W^+|$ is independent of y', so we can treat it as a constant. However, this abstraction is very loose, and we found it not to be useful in practice—e.g., it can be as loose as $[0, 2]$ even though the F_1 score never exceeds 1. Instead, we can rewrite

$$F_1(W) = 2 \cdot \frac{\frac{TP(W)}{FP(W)+|W^+|}}{\frac{TP(W)}{FP(W)+|W^+|} + 1} = 2 \cdot L\left(\frac{TP(W)}{FP(W) + |W^+|}\right)$$

where $L(x) = \frac{x}{x+1}$ is monotone, which leads to the abstract F_1 score

$$F_1^{\#}(W) \coloneqq 2 \cdot L\left(\frac{[a_1, b_1]}{[a_2, b_2] + |W^+|}\right) = 2 \cdot L\left(\left[\frac{a_1}{b_2 + |W^+|}, \frac{b_1}{a_2 + |W^+|}\right]\right) = 2 \cdot \left[\frac{a_1}{a_1 + b_2 + |W^+|}, \frac{b_1}{b_1 + a_2 + |W^+|}\right].$$

6 EXPERIMENTS

We experimentally evaluate our approach in the context of the NEAR and Quivr DSLs described in Section 5. We demonstrate that our synthesizer outperforms two synthesis baselines in terms of scalability (specifically, running time): Metasketches [Bornholt et al. 2016], an optimal synthesizer based on SMT solvers (Section 6.2), as well as an ablation that uses breadth-first search instead of A^* search with our abstract interpretation based heuristic (Section 6.3).

6.1 Experimental Setup

Benchmarks. We consider two different neurosymbolic program synthesis benchmarks, based on the DSLs described in Section 5:

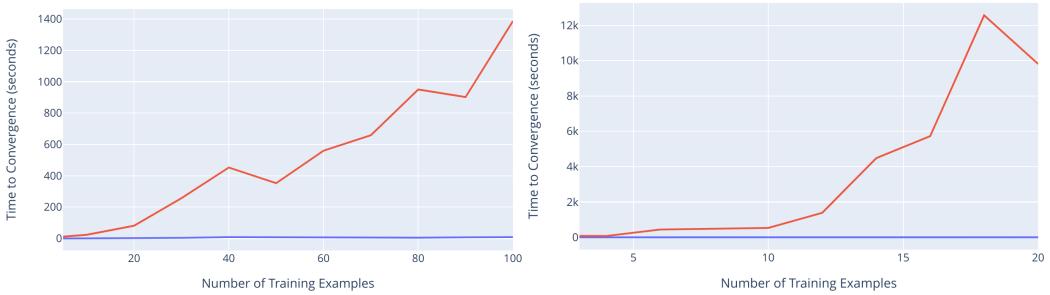
- **CRIM13:** The NEAR DSL [Shah et al. 2020] applied to the two CRIM13 datasets "sniff" (A) and "other" (B) [Burgos-Artizzu et al. 2012]. This dataset consists of featurized videos of two mice interacting in an enclosure, with 12,404 training examples with 100 frames each.
- **Quivr:** The Quivr [Mell et al. 2023] language, applied to the 17 tasks that they evaluate on. Of these, 6 tasks are on the MABe22 [Sun et al. 2022] dataset, a dataset of interactions between 3 mice, and 10 tasks are on the YTStreams [Bastani et al. 2020] dataset, a dataset of video from fixed-position traffic cameras.

Compute. We ran all experiments on a Intel Xeon Gold 6342 CPU (2.80GHz, 36 cores/72 threads). Our implementation uses PyTorch, a library which provides fast matrix operations.

6.2 Comparison to Metasketches

Metasketches performs optimal synthesis using an SMT solver by, in addition to the correctness specification, adding an SMT constraint that the program's score be greater than the best score seen so far. Thus if the SMT solver returns "SAT", a better program will have been found, and the process is repeated. Note that in our setting, there is no correctness specification, and so achieving a better score is the only SMT constraint.

A similar strategy, which we found to be more effective, is to do binary search on the objective score: supposing that the objective is in $[0, 1]$, we ask the SMT solver whether there is a program achieving score at least $1/2$; if it returns "SAT", we ask for $3/4$, and if "UNSAT" we ask for $1/4$,



(a) Quivr, task G (b) CRIM13, task A

Fig. 2. The time (seconds) to identify the optimal program and prove its optimality, for our approach (blue) and an SMT solver (red), as a function of the size of the training dataset, for two different tasks.

and so on. Our implementation used the Z3 [De Moura and Bjørner 2008] SMT solver. For a fairer comparison, in this experiment we restricted PyTorch to a single CPU core.

The two approaches were both run until they had converged to the exact optimal program. Figure 2 shows that the SMT solver scales very poorly as a function of the number of trajectories in the training dataset. While competitive for three or four trajectories, we would like to evaluate on datasets of hundreds or thousands of trajectories.

6.3 Comparison to Breadth-First Search

Next, to show the benefit of the search heuristic, we compare against an ablation which ignores the heuristic and does breadth-first search.

At any point in the search process, there is a heap of search nodes, each of which has a lower and upper bound of F_1 scores reachable from it. Rather than using the lower bound from the abstract objective value, we instead evaluate the concrete program whose parameters are the midpoint of the hyper-rectangle of abstract parameters, to get a concrete objective value; this is a better lower-bound, and it is cheap to compute. The greatest of these lower bounds provides a lower bound on the optimal F_1 score, and the greatest of these upper bounds provides an upper bound on the optimal F_1 score. Note that if the lower and upper bounds are equal, they equal the true optimal F_1 score, and search terminates.

To make search tractable, on the CRIM13 benchmarks we consider only expressions with structural cost at most 4. Note that this rules out "ite" expressions, but performs well in practice. On the Quivr benchmarks, we bound the search space in the same way that their paper does, limiting to programs with at most 3 predicates, at most 2 of which have parameters.

We use 100 trajectories from each dataset. For CRIM13, these are randomly sampled from the training set. For Quivr, to ensure that we have some positive examples, because positives are very sparse on some tasks, we use 2 positive and 10 negative trajectories specially designated in the dataset, and the remaining 88 are sampled randomly from the training set.

Table 1 shows, at different times during the search process, the best found F_1 score (the lower-bound of the interval), as well as the width of the interval of optimal F_1 scores. On most tasks, our approach (H) achieves higher F_1 scores more quickly than the ablation (B), as well as tighter intervals.

Table 1. Best F_1 score found and range between upper and lower bounds, at a particular time during search, for different tasks and different algorithms. "CA" and "CB" are the NEAR CRIM13 queries, and "QA" through "QQ" are the Quivr queries. "H" is A^* search and "B" is breadth-first search. For each task and each time, the best F_1 score is bolded and the smallest range is bolded. A range of 0 implies that search has converged to the optimal F_1 score.

Setting		10 s		30 s		1 m		2 m		5 m		10 m	
CA	H	0.21	(0.79)	0.21	(0.79)	0.21	(0.79)	**0.52**	**(0.14)**	**0.53**	**(0.03)**	**0.53**	**(0.00)**
	B	0.21	(0.79)	0.21	(0.79)	0.21	(0.79)	0.22	(0.78)	0.46	(0.54)	0.50	(0.50)
CB	H	0.75	(0.25)	0.75	(0.25)	0.75	(0.25)	**0.80**	**(0.05)**	**0.80**	**(0.04)**	**0.80**	**(0.01)**
	B	0.75	(0.25)	0.75	(0.25)	0.75	(0.25)	0.75	(0.25)	0.75	(0.25)	0.75	(0.25)
QA	H	0.12	(0.88)	**0.77**	**(0.23)**	**0.77**	**(0.23)**	**0.77**	**(0.23)**	**0.77**	**(0.23)**	**0.77**	**(0.23)**
	B	0.12	(0.88)	0.26	(0.74)	0.26	(0.74)	0.26	(0.74)	0.26	(0.74)	0.28	(0.72)
QB	H	0.27	(0.73)	0.31	(0.69)	**0.52**	**(0.48)**	**0.52**	**(0.48)**	**0.52**	**(0.48)**	**0.52**	**(0.48)**
	B	0.27	(0.73)	**0.40**	**(0.60)**	0.40	(0.60)	0.40	(0.60)	0.42	(0.58)	0.50	(0.50)
QC	H	0.06	(0.94)	0.14	(0.86)	**0.30**	**(0.70)**	**0.33**	**(0.67)**	**0.38**	**(0.62)**	**0.38**	**(0.62)**
	B	0.06	(0.94)	**0.25**	**(0.75)**	0.25	(0.75)	0.26	(0.74)	0.26	(0.74)	0.26	(0.74)
QD	H	0.04	(0.96)	**0.25**	**(0.75)**	**0.25**	**(0.75)**	**0.25**	**(0.75)**	**0.40**	**(0.60)**	**0.44**	**(0.56)**
	B	0.04	(0.96)	0.06	(0.94)	0.06	(0.94)	0.06	(0.94)	0.09	(0.91)	0.19	(0.81)
QE	H	0.04	(0.96)	**0.20**	**(0.80)**	**0.44**	**(0.56)**	**0.44**	**(0.56)**	**0.44**	**(0.56)**	**0.44**	**(0.56)**
	B	0.04	(0.96)	0.06	(0.94)	0.06	(0.94)	0.10	(0.90)	0.10	(0.90)	0.15	(0.85)
QF	H	0.38	(0.62)	**0.78**	**(0.22)**	**0.78**	**(0.22)**	**0.78**	**(0.22)**	**0.78**	**(0.22)**	**0.78**	**(0.22)**
	B	0.38	(0.62)	0.42	(0.58)	0.42	(0.58)	0.55	(0.45)	0.56	(0.44)	0.60	(0.40)
QG	H	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)
	B	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)
QH	H	0.74	(0.26)	0.74	(0.26)	0.74	(0.26)	0.74	(0.26)	0.74	(0.26)	0.74	(0.26)
	B	**0.78**	**(0.22)**	**0.78**	**(0.22)**	**0.78**	**(0.22)**	**0.78**	**(0.22)**	**0.78**	**(0.22)**	**0.78**	**(0.22)**
QI	H	0.75	(0.25)	**0.80**	**(0.20)**	**0.80**	**(0.20)**	**0.80**	**(0.20)**	**0.80**	**(0.20)**	**0.80**	**(0.20)**
	B	0.75	(0.25)	0.75	(0.25)	0.75	(0.25)	0.75	(0.25)	0.75	(0.25)	0.75	(0.25)
QJ	H	0.64	(0.36)	0.64	(0.36)	0.64	(0.36)	0.64	(0.36)	0.64	(0.36)	0.64	(0.36)
	B	0.64	(0.36)	**0.73**	**(0.27)**	**0.73**	**(0.27)**	**0.73**	**(0.27)**	**0.73**	**(0.27)**	**0.73**	**(0.27)**
QK	H	0.75	(0.25)	0.75	(0.25)	0.75	(0.25)	0.75	(0.25)	0.75	(0.25)	0.75	(0.25)
	B	0.75	(0.25)	0.75	(0.25)	0.75	(0.25)	0.75	(0.25)	0.75	(0.25)	0.75	(0.25)
QL	H	0.71	(0.29)	0.91	(0.09)	0.91	(0.09)	0.91	(0.09)	0.91	(0.09)	0.91	(0.09)
	B	0.71	(0.29)	0.91	(0.09)	0.91	(0.09)	0.91	(0.09)	0.91	(0.09)	0.91	(0.09)
QM	H	0.73	(0.27)	**0.92**	**(0.08)**	**0.92**	**(0.08)**	**0.92**	**(0.08)**	0.92	(0.08)	0.92	(0.08)
	B	0.73	(0.27)	0.80	(0.20)	0.80	(0.20)	0.80	(0.20)	0.92	(0.08)	0.92	(0.08)
QN	H	0.73	(0.27)	0.73	(0.27)	**0.89**	**(0.11)**	**0.89**	**(0.11)**	**0.89**	**(0.11)**	**0.89**	**(0.11)**
	B	0.73	(0.27)	0.73	(0.27)	0.80	(0.20)	0.80	(0.20)	0.80	(0.20)	0.80	(0.20)
QO	H	0.67	(0.33)	0.67	(0.33)	**0.86**	**(0.14)**	**0.86**	**(0.14)**	**0.86**	**(0.14)**	**0.86**	**(0.14)**
	B	0.67	(0.33)	0.67	(0.33)	0.80	(0.20)	0.80	(0.20)	0.80	(0.20)	0.80	(0.20)
QP	H	0.50	(0.50)	0.50	(0.50)	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)
	B	0.50	(0.50)	0.50	(0.50)	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)
QQ	H	0.80	(0.20)	0.80	(0.20)	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)
	B	0.80	(0.20)	0.80	(0.20)	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)	1.00	(0.00)

7 RELATED WORK

Neurosymbolic synthesis. There has been a great deal of recent interest in neurosymbolic synthesis [Chaudhuri et al. 2021], including synthesis of functional programs [Gaunt et al. 2016; Shah et al. 2020; Valkov et al. 2018], reinforcement learning policies [Anderson et al. 2020; Inala et al. 2020],

programs for extracting data from unstructured text [Chen et al. 2023, 2021; Ye et al. 2021], and programs that extract data from video trajectories [Bastani et al. 2021; Mell et al. 2023; Shah et al. 2020]. Some of these approaches have proposed pruning strategies based on monotonicity [Chen et al. 2021; Mell et al. 2023], but for specific DSLs. NEAR is a general framework for neurosymbolic synthesis based on neural admissible heuristics [Shah et al. 2020]; however, their approach is not guaranteed to synthesize optimal programs due to nonconvexity of neural network learning. To the best of our knowledge, our work proposes the first general framework for optimal synthesis of neurosymbolic programs.

Optimal synthesis. More broadly, there has been recent interest in optimal synthesis [Bornholt et al. 2016; Smith and Albarghouthi 2016], typically focusing on optimizing performance properties of the program such as running time rather than accuracy; superoptimization is a particularly well studied application [Bansal and Aiken 2008; Massalin 1987; Mukherjee et al. 2020; Phothilimthana et al. 2016; Sasnauskas et al. 2017]. Our experiments demonstrate that our approach outperforms Bornholt et al. [2016], a general framework for optimal synthesis based on SMT solvers. There has also been work on synthesizing a program that maximizes an objective (expressed as a neural network scoring function) [Ye et al. 2021], but this work does not target programs with neural network components. Optimal synthesis has also been leveraged to synthesizing minimal guards for memory safety [Dillig et al. 2014], or synthesizing chemical reaction networks [Cardelli et al. 2017], and optimal layouts for quantum computing [Tan and Cong 2020].

Abstract interpretation for synthesis. There has been work on leveraging abstract interpretation for pruning portions of the search space in program synthesis [Guria et al. 2023; So and Oh 2017], as well as using abstraction refinement [Wang et al. 2017]; however, these approaches target traditional synthesis. Rather than evaluating an abstract semantics on partial programs, Wang et al. [2017] constructs a data structure compactly representing concrete programs whose abstract semantics are compatible with the input-output examples. However, it is not obvious how their data structure (which targets Boolean specifications) can be adapted to our quantitative synthesis setting.

Abstract interpretation for planning. One line of work, initiated by the FF Planner [Hoffmann and Nebel 2001], uses abstract semantics to perform reachability analysis to prune invalid plans [Gregory et al. 2012; Hoffmann 2003; Zhi-Xuan et al. 2022]. However, other than pruning invalid plans, the reachability analysis not used in the computation of the search heuristic, which is a traditional heuristic such as the $h_{\max}$ heuristic (in particular, $h_{\max}$ computes the shortest plan in a "relaxed model" that drops delete lists from the postconditions of abstract actions, and outputs the length of this plan). In contrast, in our work, an abstract transformer for the objective function provides a lower bound that is directly used as the search heuristic.

A second line of work [Marthi et al. 2008; Vega-Brown and Roy 2018] considers computing optimal plans by underapproximating the cost function. They assume that the total cost of a plan equals the sum of the costs of the individual actions in that plan. Then, given a lower bound on the cost of each action, they simply sum these lower bounds to obtain a lower bound on the cost of the overall plan. This strategy makes strong assumptions about the structure of the overall cost function, whereas our abstract interpretation based approach requires no such assumptions.

Finally, another key difference is that we are abstracting over real-valued parameters of partial programs, whereas the above approaches are abstracting over continuous states. Thus, our framework requires a way to iteratively refine the program space (specified by the "children" function), which is absent from their frameworks.

8 CONCLUSION

We have proposed a general framework for synthesizing programs with real-valued inputs and outputs, using A^* search in conjunction with a search heuristic based on abstract interpretation. Our framework searches over a space of generalized partial programs, which represent sets of concrete programs, and uses the search heuristic to establish upper bounds on the objective value of a given generalized partial program. In addition, we propose a natural strategy for constructing abstract transformers for components with monotone semantics. If our algorithm returns a program, then this program is guaranteed to be optimal. Our experimental evaluation demonstrates that our approach is more scalable than existing optimal synthesis techniques. Directions for future work include improving the scalability of our approach and applying it to additional synthesis tasks.

Acknowledgements. We thank the anonymous reviewers for their helpful feedback. This work was supported in part by NSF Award CCF-1910769, NSF Award CCF-1917852, ARO Award W911NF-20-1-0080, and Amazon/ASSET Gift for Research in Trustworthy AI.

REFERENCES

Greg Anderson, Abhinav Verma, Isil Dillig, and Swarat Chaudhuri. 2020. Neurosymbolic reinforcement learning with formally verified exploration. *Advances in neural information processing systems* 33 (2020), 6172–6183.

Sorav Bansal and Alex Aiken. 2008. Binary Translation Using Peephole Superoptimizers.. In *OSDI*, Vol. 8. 177–192.

Favyen Bastani, Songtao He, Arjun Balasingam, Karthik Gopalakrishnan, Mohammad Alizadeh, Hari Balakrishnan, Michael Cafarella, Tim Kraska, and Sam Madden. 2020. MIRIS: Fast Object Track Queries in Video. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1907–1921.

Favyen Bastani, Songtao He, Ziwen Jiang, Osbert Bastani, and Sam Madden. 2021. SkyQuery: an aerial drone video sensing platform. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 56–67.

James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing synthesis with metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 775–788.

Xavier P. Burgos-Artizzu, Piotr Dollár, Dayu Lin, David J. Anderson, and Pietro Perona. 2012. Social behavior recognition in continuous video. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*. 1322–1329. https://doi.org/10.1109/CVPR.2012.6247817

Luca Cardelli, Milan Češka, Martin Fränzle, Marta Kwiatkowska, Luca Laurenti, Nicola Paoletti, and Max Whitby. 2017. Syntax-guided optimal synthesis for chemical reaction networks. In *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II 30*. Springer, 375–395.

Swarat Chaudhuri, Kevin Ellis, Oleksandr Polozov, Rishabh Singh, Armando Solar-Lezama, Yisong Yue, et al. 2021. Neurosymbolic programming. *Foundations and Trends® in Programming Languages* 7, 3 (2021), 158–243.

Qiaochu Chen, Arko Banerjee, Çağatay Demiralp, Greg Durrett, and Isil Dillig. 2023. Data Extraction via Semantic Regular Expression Synthesis. In *OOPSLA*.

Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. 2021. Web question answering with neurosymbolic program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 328–343.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 238–252.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

Thomas Dillig, Isil Dillig, and Swarat Chaudhuri. 2014. Optimal guard synthesis for memory safety. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*. Springer, 491–507.

Alexander L Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. 2016. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428* (2016).

Peter Gregory, Derek Long, Maria Fox, and J. Christopher Beck. 2012. Planning Modulo Theories: Extending the Planning Paradigm. *Proceedings of the International Conference on Automated Planning and Scheduling* 22, 1 (May 2012), 65–73.

https://doi.org/10.1609/icaps.v22i1.13505
Sankha Narayan Guria, Jeffrey S Foster, and David Van Horn. 2023. Absynthe: Abstract Interpretation-Guided Synthesis. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1584–1607.
Jörg Hoffmann. 2003. The Metric-FF Planning System: Translating"Ignoring Delete Lists"to Numeric State Variables. *Journal of artificial intelligence research* 20 (2003), 291–341.
Jörg Hoffmann and Bernhard Nebel. 2001. The FF Planning System: Fast Plan Generation through Heuristic Search. *J. Artif. Int. Res.* 14, 1 (may 2001), 253–302.
Jeevana Priya Inala, Yichen Yang, James Paulos, Yewen Pu, Osbert Bastani, Vijay Kumar, Martin Rinard, and Armando Solar-Lezama. 2020. Neurosymbolic transformers for multi-agent communication. *Advances in Neural Information Processing Systems* 33 (2020), 13597–13608.
Michael J Kearns and Umesh Vazirani. 1994. *An introduction to computational learning theory*. MIT press.
Dexter Kozen. 1997. Kleene algebra with tests. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 3 (1997), 427–443.
Bhaskara Marthi, Stuart Russell, and Jason Andrew Wolfe. 2008. Angelic Hierarchical Planning: Optimal and Online Algorithms.. In *ICAPS*. Citeseer, 222–231.
Henry Massalin. 1987. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News* 15, 5 (1987), 122–126.
Stephen Mell, Favyen Bastani, Stephan Zdancewic, and Osbert Bastani. 2023. Synthesizing Video Trajectory Queries. In *CAV*.
Manasij Mukherjee, Pranav Kant, Zhengyang Liu, and John Regehr. 2020. Dataflow-based pruning for speeding up superoptimization. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–24.
Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. 2016. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 297–310.
Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. 2017. Souper: A synthesizing superoptimizer. *arXiv preprint arXiv:1711.04422* (2017).
Ameesh Shah, Eric Zhan, Jennifer Sun, Abhinav Verma, Yisong Yue, and Swarat Chaudhuri. 2020. Learning differentiable programs with admissible neural heuristics. *Advances in neural information processing systems* 33 (2020), 4940–4952.
Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. *Acm Sigplan Notices* 51, 6 (2016), 326–340.
Sunbeom So and Hakjoo Oh. 2017. Synthesizing Imperative Programs from Examples Guided by Static Analysis. In *Static Analysis*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 364–381.
Jennifer J. Sun, Brian Geuther, Vivek Kumar, Alice Robie, Catherine Schretter, Keith Sheppard, Dipam Chakraborty, Kristin Branson, and Ann Kennedy. 2022. The MABe22 Benchmarks for Representation Learning of Multi-Agent Behavior. https://doi.org/10.22002/D1.20186
Jennifer J Sun, Tomomi Karigo, Dipam Chakraborty, Sharada P Mohanty, Benjamin Wild, Quan Sun, Chen Chen, David J Anderson, Pietro Perona, Yisong Yue, and Ann Kennedy. 2021. The Multi-Agent Behavior Dataset: Mouse Dyadic Social Interactions. *arXiv preprint arXiv:2104.02710* (2021).
Bochen Tan and Jason Cong. 2020. Optimal layout synthesis for quantum computing. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–9.
Lazar Valkov, Dipak Chaudhari, Akash Srivastava, Charles Sutton, and Swarat Chaudhuri. 2018. Houdini: Lifelong learning as program synthesis. *Advances in neural information processing systems* 31 (2018).
William Vega-Brown and Nicholas Roy. 2018. Admissible Abstractions for Near-Optimal Task and Motion Planning. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence* (Stockholm, Sweden) *(IJCAI'18)*. AAAI Press, 4852–4859.
Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.
Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2021. Optimal Neural Program Synthesis from Multimodal Specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021*. 1691–1704.
Tan Zhi-Xuan, Joshua B Tenenbaum, and Vikash K Mansinghka. 2022. Abstract Interpretation for Generalized Heuristic Search in Model-Based Planning. *arXiv preprint arXiv:2208.02938* (2022).