REPRESENTING PARTIAL PROGRAMS WITH BLENDED ABSTRACT SEMANTICS

Anonymous authors
Paper under double-blind review

ABSTRACT

Synthesizing programs from examples requires searching over a vast, combinatorial space of possible programs. In this search process, a key challenge is representing the behavior of a partially written program before it can be executed, to judge if it is on the right track and predict where to search next. We introduce a general technique for representing partially written programs in a program synthesis engine. We take inspiration from the technique of abstract interpretation, in which an approximate execution model is used to determine if an unfinished program will eventually satisfy a goal specification. Here we *learn* an approximate execution model implemented as a modular neural network. By constructing compositional program representations that implicitly encode the interpretation semantics of the underlying programming language, we can represent partial programs using a flexible combination of concrete execution state and learned neural representations, using the learned approximate semantics when concrete semantics are not known (in unfinished parts of the program). We show that these hybrid neuro-symbolic representations enable execution-guided synthesizers to use more powerful language constructs, such as loops and higher-order functions, and can be used to synthesize programs more accurately for a given search budget than pure neural approaches in several domains.

1 INTRODUCTION

Inductive program synthesis – the problem of inferring programs from examples – offers the promise of building machine learning systems which are interpretable, generalize quickly, and allow us to solve structured tasks such as planning and interacting with computer systems. In recent years, neurally-guided program synthesis, which use deep learning to guide search over the space of possible programs, has emerged as a promising approach (Balog et al., 2016; Devlin et al., 2017). In this framework, partially-constructed programs are judged to determine if they are on the right track and to predict where to search next (see Figure 1). A key challenge in neural program synthesis is *representing* the behavior of partially written programs, in order to make these judgments. In this work, we present a novel method for representing the semantic content of partially written code, which can be used to guide search to solve program synthesis tasks.

Consider a tower building domain where a hand drops blocks, Tetris-style, onto a vertical 2D scene. Given an image $\mathcal{X}$ of a scene (Figure 1), our task is to write a program which builds a tower matching the image $\mathcal{X}$. To do this, a model can perform search in the space of programs, iteratively adding code until the program is complete. While attempting to synthesize a program, imagine that arriving at a partially-constructed program s (short for *sketch*), where `HOLE` signifies unfinished code:

$s =$ `loop(4, [buildColumn(1), moveHand(<HOLE>)])`

Note that this partial program can't reach the goal state, because the target image has columns of height 2, but this program can only build columns of height 1. For an algorithm to determine if it should expand s or explore another part of the search space, it needs to answer the following question: Is s on the right track to satisfy the goal? Answering this question requires an effective *representation* of partial programs.

Neural program synthesis techniques differ in how they represent programs. Some represent programs by their syntax (Devlin et al., 2017; Allamanis et al., 2018), encoding the structure of the

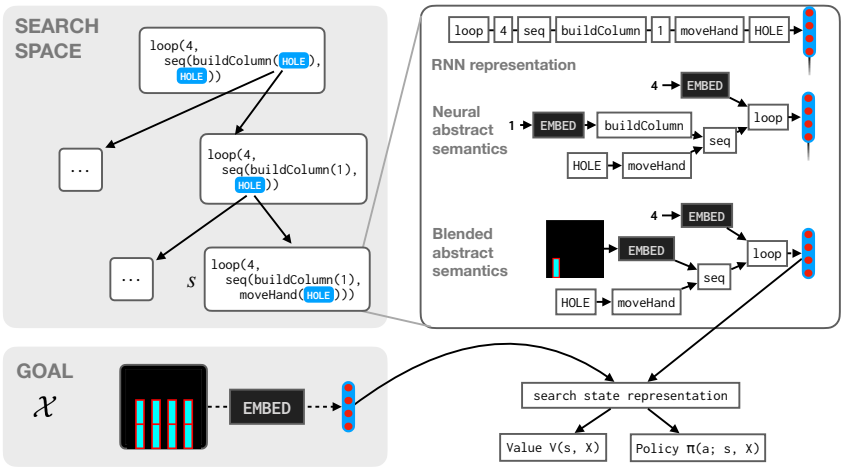


Figure 1: Schematic overview of the search procedure and representational scheme. We characterize program synthesis as a goal-conditioned search through the space of partial programs (left), and propose a novel representational scheme (blended abstract semantics) to facilitate this search process. Left: a particular trajectory through the space of partial programs, where the goal is to find a program satisfying the target image. Right: three encoding schemes for partial programs, which can each be used as the basis of a code-writing search policy and code-assessing value function.

programs using sequence and graph neural networks. Recently, approaches which instead represent partial programs via their *semantic* state have been shown to be particularly effective. In these **execution-guided neural synthesis** approaches (Chen et al., 2018; Ellis et al., 2019; Zohar & Wolf, 2018), partial programs are *executed* and represented with their return values. (To see why this is helpful, consider two distinct syntactic expressions $2 + 1$ and $6/2$; a syntax-based model needs to represent them separately, whereas a model using a semantic representation will represent both as 3.) However, execution is not always possible for a partial program. In our running example, before the `HOLE` is filled with an integer value, we cannot meaningfully execute the partially-written loop in s. This is a common problem for languages containing higher-order functions and control flow, where execution of partially written code is often ill-defined. Thus, a key question is: How might we represent the semantics of unfinished code?

A classic method for representing program state, known as abstract interpretation, can be used to reason about the set of states that a partial program could reach, given the possible instantiations of the unfinished parts of the program. Using abstract interpretation, an approximate execution model can determine if an unfinished program will eventually satisfy a goal specification. For example, in our tower-building domain, an abstract interpreter could be designed to track, for every horizontal location, the minimum tower height which all continuations are guaranteed to exceeded. However, this technique is often low-precision; hand-designed abstract execution models greatly overapproximate the set of possible execution states, and contain no notion of what code is likely to be written.

We hypothesize that, by mimicking the compositional structure of abstract interpretation, learned components can be used to effectively represent ambiguous program state. In this work, we make two contributions: we introduce **neural abstract semantics**, in which a compositional, approximate execution model is used to represent partially written code. We further introduce **blended abstract semantics**, which aims to represent the state of unfinished programs as faithfully as possible by concretely executing program components whenever possible, and otherwise, approximating program state with a learned abstract execution model.

Consider again the partial program s and the blended abstract semantics encoding in Figure 1. The sub-expression `buildColumn(1)` is fully concrete, and can thus be concretely executed to render an image. On the other hand, for functions whose arguments are not fully defined, such as `moveHand`, we instead employ abstract neural modules to represent the execution state. For this example, blended neural execution makes it easy to recognize that s is not a suitable partial program, because no integer argument to `moveHand`—which controls the spacing between the columns—would make the state in s match the goal $\mathcal{X}$.

This combination of learned execution and concrete execution allows robust representation of partial programs, which can be used for downstream synthesis tasks. We show that our model can effectively learn to represent partial program states for languages where previous execution-guided synthesis techniques are not applicable. In summary,

- We introduce blended neural semantics, a novel method for representing the semantic state of partially written programs inspired by abstract interpretation.
- We describe how to integrate our program representations into existing approaches for learning search policies and search heuristics.
- We validate our new approach with program synthesis experiments in three domains: tower construction, list processing, and string editing. We show that our approach outperforms neural synthesis baselines, solving at least 5% more programs in each domain.

2 Related work

Synthesizing programs from examples is a classic AI problem which has seen advances from the Programming Languages community (Solar-Lezama, 2008). Recently, much progress has been made using neural methods to aid search. Such techniques include enumerative approaches (Balog et al., 2016), translation-based techniques (Devlin et al., 2017), hybrid approaches using sketches such as Murali et al. (2017), Nye et al. (2019), and Dong & Lapata (2018), among others. The recent work of Odena & Sutton (2019) also studies program representation, proposing property signatures. Work has also been done using graph neural networks to encode the syntax of programs (Allamanis et al., 2018; Brockschmidt et al., 2018) for bug fixing, variable naming, and synthesis.

Recent work has introduced the notion of "execution-guided neural program synthesis" (Ellis et al., 2019; Chen et al., 2018; Zohar & Wolf, 2018). In this framework, the neural representations used for search are conditioned on the executed program state instead of the program syntax. These techniques have been shown to solve difficult search problems outside the scope of enumerate or syntax-based neural synthesis alone. However, such execution-guided approaches have several limitations. We aim to generalize execution guided synthesis, so that it can be applicable to a wider range of domains, search techniques, and programming language constructs.

Our work is directly inspired by two techniques: abstract interpretation-based synthesis (Wang et al., 2017; Hu et al., 2019) and neural module networks (Andreas et al., 2016; Johnson et al., 2017). We employ module networks to implement blended neural execution, which aims to provide a learned execution scheme directly inspired by abstract interpretation. This approach is also related to other tree-structured encoders (Socher et al., 2011; Dyer et al., 2016).

3 Blended Abstract Semantics

Consider the problem of synthesizing arithmetic expressions from input–output pairs. Suppose we have the following context-free grammar for expressions:

$$\mathcal{G} = \texttt{E} \rightarrow \texttt{E * E} \mid \texttt{E + E} \mid \texttt{x} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4}$$

and a specification $\mathcal{X}$ consisting of the input–output pairs $\{(x = 3, y = 7), (x = 5, y = 11)\}$. Suppose further that we have a candidate program `(2 * x) + 1` $\in \mathcal{G}$. To check that this program is consistent with the specification, we can evaluate it on the inputs x in the specification according to the **concrete semantics** of the language: to evaluate `(2 * x) + 1` on the example $(x = 3, y = 9)$, we observe that the expression `(2 * x)` evaluates to the integer 6, and the expression `1` evaluates to the integer 1; thus the whole expression evaluates to 7, as desired. Repeating this process with $x = 5$ returns the value 11.

Formally: Let $\mathcal{C}$ denote a context (e.g. $\{x = 3\}$)). The **concrete value** of an expression E in a context $\mathcal{C}$ is:[1]

$$[\![E]\!]_{\mathcal{C}} = \begin{cases} [\![k]\!]_{\mathcal{C}} = k & \text{a constant } k \\ [\![x]\!]_{\mathcal{C} \models x=v} = v & \text{a variable } x \text{ is evaluated to its value in the context} \\ [\![f(E_1 \cdots)]\!]_{\mathcal{C}} = \text{run}[f, \mathcal{C}]([\![E_1]\!]_{\mathcal{C}} \cdots) & \text{recursively evaluate the arguments, then run } f \end{cases}$$

[1]Domains with lambdas have slightly more complicated semantics. See Appendix A for details.

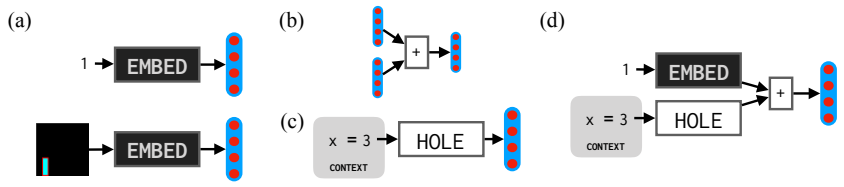


Figure 2: (a) Example applications of the EMBED function. (b) Neural abstract module for +. (c) Neural placeholder module encoding a HOLE with the context $\{x = 3\}$. (d) Neural abstract semantic encoding of the partial program `1 + HOLE` with the context $\{x = 3\}$.

The goal of synthesis is to find an expression $E : [\![E]\!]_x = y$ under concrete semantics.

Iterative construction of partial programs Where did the expression `(2 * x) + 1` come from? Neurally-guided synthesis techniques generally employ search. In this work, we use top-down search: starting with the top-level (incomplete) expression HOLE, we consider all possible expansions, (HOLE $\rightarrow$ HOLE + HOLE, HOLE $\rightarrow$ 1, HOLE $\rightarrow$ 2, ...) and select the one we believe is most likely to succeed (Figure 1 left). Concrete semantics cannot be used for this selection, because expressions such as `x + HOLE` are incomplete and cannot be executed. Thus, we need a different mechanism to guide search. The more effectively we can filter the set of incomplete candidate programs, the faster our synthesis algorithm.

Conventional **abstract interpretation** solves this problem by defining an alternative semantics for which even incomplete expressions can be evaluated. Consider the candidate expression `HOLE * 2`. No matter how the HOLE is filled, the expression returns an even number, so it cannot be consistent with the specification above. In many problems, we can define a space of "abstract values" (like *even integer*) and abstract semantics so that the abstract value of a partial program can be determined. This allows us to rule out partial programs on the basis of the abstraction alone (Wang et al., 2017). However, constructing appropriate abstractions is difficult and requires domain-specific engineering; an ideal procedure would automatically discover an effective space of abstract interpretations.

Neural abstract semantics $[\![\cdot]\!]^{nn}$ As a first step, we implement the abstract interpretation procedure with a neural network. (This is a natural choice: neural networks excel at representation learning, and the goal of abstract interpretation is to encode an informative representation of the set of values that could be returned by a partial program.) For the program `1 + HOLE`, we can encode the expression `1` to a learned representation (Figure 2a, top), likewise encode HOLE (Figure 2c), and finally employ a learned abstract implementation of the + operation (Figure 2b).

For concrete leaf nodes, such as constants or variables bound to constants, neural semantics are given using a **state embedding function** EMBED$(\cdot)$, which maps any concrete state in the programming language into a vector representation: EMBED : $(State \mid \mathbb{R}^d) \rightarrow \mathbb{R}^d$. If the input to EMBED is already vector-valued, EMBED performs the identity operation. **Neural placeholders** provide a method for computing a vector representation of unwritten code, denoted by the HOLE token. To compute the representation for HOLE, we define a neural embedding function h which takes a context $\mathcal{C}$ and outputs a vector. For each built-in function f, the neural abstract semantics of a function f are given by a separate **neural module** (a learned vector-valued function as in Andreas et al. (2016)) $[\![f]\!]^{nn}$ with the same arity as f. Therefore, computing the neural semantics means applying the neural function $[\![f]\!]^{nn}$ to its arguments, which returns a vector. Since the neural semantics mirrors the concrete semantics, its implementation does not require changes to the underlying programming language. Formally, neural semantics involve a slightly larger set of cases than concrete semantics:

$$[\![E]\!]^{nn}_{\mathcal{C}} = \begin{cases} [\![k]\!]^{nn}_{\mathcal{C}} = \textsc{Embed}(k) & \text{a constant } k \text{ is embedded} \\ [\![x]\!]^{nn}_{\mathcal{C} \models x=v} = \textsc{Embed}(v) & \text{embed the value } v \text{ of the variable } x \\ [\![\texttt{HOLE}]\!]^{nn}_{\mathcal{C}} = h(\mathcal{C}) & \text{a neural placeholder based on context} \\ [\![f(E_1 \cdots)]\!]^{nn}_{\mathcal{C}} = [\![f]\!]^{nn}([\![E_1]\!]^{nn}_{\mathcal{C}} \cdots) & \text{using neural module } [\![f]\!]^{nn} \end{cases}$$

Blended abstract semantics $[\![\cdot]\!]^{blend}$ Notice that for an expression such as `(2 * x) + HOLE`, the concrete value of the sub-expression `(2 * x)` is known, since it contains no holes. The neural semantics above don't make use of this knowledge. To improve upon this, we extend neural semantics and introduce blended semantics, which alternates between neural and concrete interpretation as appropriate for a given expression:

- If the expression is a constant or a variable, use the concrete semantics.
- If the expression is a `HOLE`, use the neural semantics.
- If the expression is a function call, recursively evaluate the expressions that are the arguments to the function. If all arguments evaluate to concrete values, execute the function concretely. If any argument evaluates to a vector representation, transform all concrete values to vectors using EMBED and apply the neural semantics of the function.

Formally, we can write:

$$[\![E]\!]^{blend}_{\mathcal{C}} = \begin{cases} [\![k]\!]^{blend}_{\mathcal{C}} = [\![k]\!]_{\mathcal{C}} = k & \text{a constant } k \\ [\![x]\!]^{blend}_{\mathcal{C}\models x=v} = [\![x]\!]_{\mathcal{C}\models x=v} = v & \text{variable } x \text{ in context} \\ [\![\texttt{HOLE}]\!]^{blend}_{\mathcal{C}} = [\![\texttt{HOLE}]\!]^{nn}_{\mathcal{C}} = h(\mathcal{C}) & \text{a neural placeholder based on context} \\ [\![f(E_1 \cdots)]\!]^{blend}_{\mathcal{C}} = [\![f]\!]([\![E_1]\!]^{blend}_{\mathcal{C}} \cdots) & \text{if all arguments are concrete} \\ [\![f(E_1 \cdots)]\!]^{blend}_{\mathcal{C}} = [\![f]\!]^{nn}(\textsc{Embed}([\![E_1]\!]^{blend}_{\mathcal{C}}) \cdots) & \text{if any arguments are vectors} \end{cases}$$

Because blended abstract semantics replaces concrete sub-components with their concrete values, we expect blended semantics to result in more robust representations, especially for long or complex programs where large portions can be concretely executed.

4 PROGRAM SYNTHESIS WITH BLENDED ABSTRACT SEMANTICS

To perform synthesis, we experiment with methods to guide search introduced in Ellis et al. (2019).[2] In this work, the search over partial programs is formulated as an MDP, in which each state is a pair $(s, \mathcal{X})$ consisting of a partial-program and a specification, and actions $a \in \mathcal{G}$ are expansions of `HOLE`s under rules under the grammar. We assume a reward of 1 for programs which satisfy $\mathcal{X}$. In this framework, we **learn to search** by training a policy $\pi(a|s, \mathcal{X})$ that proposes expansions to s, and optionally a value function $V(s, \mathcal{X})$ that predicts the probability that $\mathcal{X}$ is solvable via any expansion of s.

Let $\mathcal{X} = \{(x_i, y_i)\}$, where (x_i, y_i) are input-output pairs. Let $[\![s]\!]^{blend}_{x_i}$ denote the blended abstract semantic representation of s with input x_i. The representation of a state $\texttt{rep}(s, \mathcal{X})$ is:

$$\texttt{rep}(s, \mathcal{X}) = \frac{1}{n} \sum_{x_i, y_i \in \mathcal{X}} \texttt{ReLU}(W([[\![s]\!]^{blend}_{x_i}; \textsc{Embed}(y_i)]))$$

Here, W is a learnable weight matrix, and the representation is averaged across all input-output pairs of $\mathcal{X}$. Given this state representation, the policy and value function are:

$$\pi(a \mid s, \mathcal{X}) = \texttt{softmax}(\texttt{MLP_a}(\texttt{rep}(s, \mathcal{X})))$$
$$V(s, \mathcal{X}) = \sigma(\texttt{MLP_V}(\texttt{rep}(s, \mathcal{X})))$$

Here, `MLP` is a multi-layer perceptron. Note the value function outputs a value between 0 and 1; this allows for a probabilistic interpretation (see below).

End-to-end training We train our policy π using imitation learning. Starting from the empty partial program $s_0 = \texttt{HOLE}$, we generate a sequence of partial programs $s_1, s_2, \cdots$ by sampling a sequence of expansions $a_0, a_1, \cdots$ [3] from the grammar $\mathcal{G}$. Let $p = s_T$ be the completed program. We obtain specifications $\mathcal{X} = \{(x_i, y_i)\}$ by sampling a set of inputs $x_1 \cdots x_n$ and obtaining outputs using concrete semantics $y_i = [\![p]\!]^{c}_{x_i}$. Thus, from a sequence of expansions $a_0, a_1, \cdots$, we can

[2] We review here, but see Ellis et al. (2019) for more details.

[3] Often a PCFG is used to sample these expansions. See Appendix B for details.

collect a set of triplets $\{(\mathcal{X}, s_i, a_i)\}$ as training data. This process is repeated to generate the training set $\mathcal{D}$. We can then perform supervised training, maximizing the log likelihood of the following:

$$\mathcal{L}(\pi) = \mathbb{E}_{(\mathcal{X},s,a)\sim\mathcal{D}} \left[\log \pi(a \mid s, \mathcal{X})\right]$$

We train the value function by sampling rollouts of partial programs $s_0 \cdots s_T$ from a fully-trained π, minimizing the error in a Monte-Carlo estimate of the expected reward R (i.e., the probability of success under the policy).

$$\mathcal{L}^{\text{RL}}(V) = \mathbb{E}_{(R,s_0\ldots s_T)\sim\pi(\cdot|s_0,\mathcal{X})}\left[\sum_{t\leq T} \text{err}(V(s_t, \mathcal{X}), R)\right]$$

For our error function, we use a logistic loss rather than the more common MSE.

Search We explore a variety of code-writing search algorithms. Using only a policy, we can employ sample-based search and best-first search (where the log probability of generating s under π is used as the scoring function). With the addition of a learned value function, we can perform A*-based search with $-\log V(s, \mathcal{X})$ as a heuristic (see Ellis et al. (2019) for details).

5 EXPERIMENTS

We evaluate our model in two domains containing language constructs not handled by concrete execution-guided synthesis approaches: a tower-building domain with looping constructs, and a list-processing domain with higher order functions. We additionally test on a string-editing domain for which execution-guided synthesis is possible, but requires extensive DSL modification; there, we examine how our approach fares without these modifications.

5.1 LOOPING CONSTRUCTS: TOWER CONSTRUCTION

We begin by investigating how our model performs in generative programming domains with higher-level control flow such as loops. Looping programs are an essential part of sophisticated programming languages, and aren't naturally handled by previous execution-guided synthesis approaches. Our experiments in the tower-building domain employ a DSL similar to the language depicted in the introduction to construct towers in a 2D world. As above, the goal is to construct a program which successfully renders to a target image (examples in Figure 3). Language details can be found in Appendix B. We compared against two baselines (see Figure 1): (1) Neural abstract semantics (defined in Section 3), which does not apply concrete execution to concrete subtrees, and (2) RNN encoding, which encodes partial programs using a GRU: $\pi(a \mid \texttt{GRU_enc}(s), \mathcal{X})$.

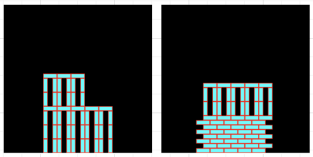
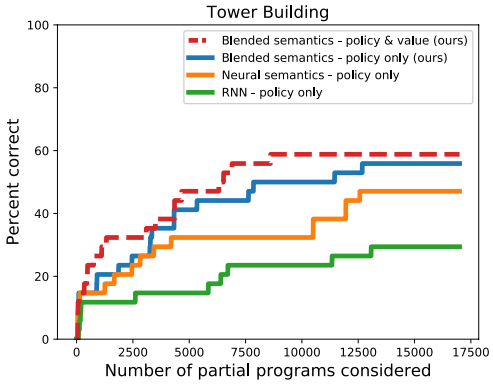


Figure 3: Example constructions in the tower-building domain

To evaluate our model, we constructed a test set of tower-building problems involving combinations of tower-building motifs seen during training. We evaluate our models by performing best-first search from the learned policy. We also test using a value function, where we are doing A* search with the policy as the prior cost and the value function as the heuristic future cost estimate.

Synthesis results Figure 4 left shows our overall synthesis results in the tower-building domain, measuring the percentage of test problems solved as a function of the number of search nodes (partial programs) considered. The sequence encoding performs poorly and is unable to solve a majority of test problems. The neural abstract semantics model achieves better performance, solving about half of the test problems within the allotted search budget. Blended execution outperforms both baselines. We additionally observe that adding a value function as a search heuristic further increases performance of our blended model, which is consistent with the findings in Ellis et al. (2019).

Comparison to abstract interpretation How does the learned value function compare to hand-coded abstract interpretation? Classic abstract interpretation is conservative; it can be thought of as a classifier with perfect recall, but poor precision, only rejecting the partial programs it knows



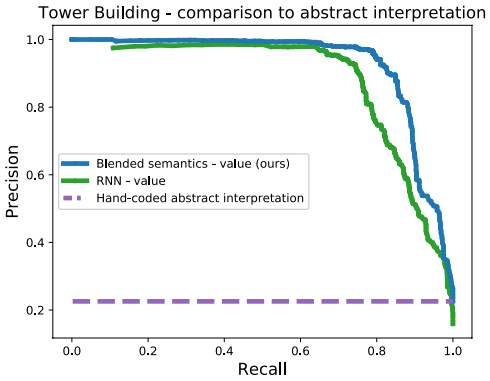


Figure 4: Left: Overall synthesis results in the tower-building domain. Right: Comparing the value function to a hand-coded abstract interpretation. Blended abstract semantics outperforms baselines in synthesis tasks, and obtains higher classification precision than hand-coded abstract interpretation.

for sure to be unsuitable. Can our value function also detect these clearly bad partial programs, but ascribe low value to less obviously bad candidates? To test this, we conditioned the model on tasks from our test corpus, and sampled 15 search trajectories from our blended semantics policy for each task. For each partial program encountered during search, we compute the model's value judgment, and recorded whether each rollout was successful. Treating rollout success as a noisy label of partial program quality, and using the value function as a classifier, we plot precision vs recall of the value judgements as we vary the classification threshold. Figure 4 right shows our results for this experiment. As the classification threshold is varied, our learned value maintains comparable recall compared to the hand-coded abstraction, while achieving better precision. For high classification thresholds, our model achieves performance comparable to the hand-coded abstract interpretation, and additional precision is gained by lowering the classification threshold. The RNN value performs worse on this test, achieving lower precision and recall.

5.2 Higher-order functions: functional list processing

List Processing

Examples	Program
`[-4,9,4,6] → [18,12]` `[15,3,3,-14] → [30,6,6]`	`map (λx.x*2) (filter` `(λx.(x>0 & x%3==0)) input)`
`[1,2,3,4] → [5,5,5,5]` `[1,2,4,-1] → [0,6,6,0]`	`zipwith (λx,y.x+y)` `(input) (reverse input)`

String Editing

Examples	Program
`+106 769-858-438 → (769)` `+63 099-824-351 → (099)`	`Const('(') | GetToken(Number, 1)` `| Const(')')`

`Mariya Sergienko → Dr. Mariya`
`Andrew Cencici → Dr. Andrew`
	`Const('D') | Const('r')`
	`| Const('.') | Const(' ')`
	`| GetToken(Word, 0)`

Figure 5: Example programs from the list processing (left) and string editing (right) domains.

In our second experimental domain, we seek to answer two questions: How well does our model perform on input-output synthesis? How effectively can it synthesize programs containing higher-order functions? Although previous work (Zohar & Wolf, 2018) has successfully applied execution-guided approaches to list processing (using the DeepCoder language), the use of higher-order functions was severely limited: only a small, predefined set of "lambdas," (such as `(*2)`, `is_even`, `(>0)`) were used as arguments for higher-order functions. For example, synthesizing a program which "filters all elements divisible by 3 from a list" is not possible with this DSL. However, in real programming languages, higher-order functions must be able to accept a combinatorially large set of possible lambda functions as input. This presents a challenge for execution-guided synthesis approaches such as Zohar & Wolf (2018), for which the assumption of a small set of lambda functions is key. To this end, we modified the Deepcoder DSL to allow a richer set of possible programs. We replaced the predefined set of lambda functions with a grammar allowing for the combinatorial combination of grammar elements (examples in Figure 5 left). The modified grammar is given in Appendix B.

Results Figure 6 shows the results of synthesis using best-first search from a policy on test problems sampled the same distribution as the training problems. Our blended model finds the highest

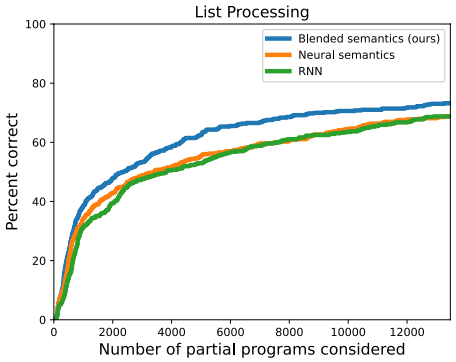


Figure 6: Synthesis results for list processing. Models were trained and tested on programs each containing 2-3 higher order functions with lambdas of depth 3. Blended semantics achieves the highest performance.

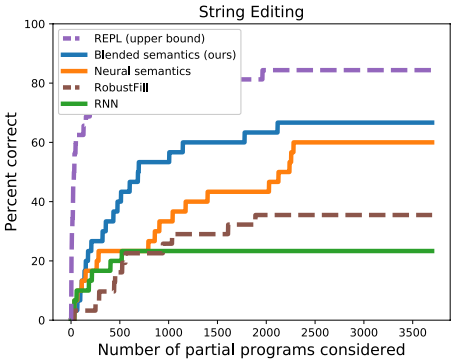


Figure 7: Synthesis results for string editing. Blended semantics outperforms all baselines, except for the execution-guided REPL model, which relies on domain-specific language modifications.

overall number of correct programs, achieving 5-10% higher accuracy given the same search budget compared to the neural semantics and RNN encoding schemes. The blended model also yielded superior results on numerous variations of these tasks (increasing number of higher order functions, varying integer ranges, varying search method, etc). See Appendix B for details.

5.3 IO PROGRAMMING: STRING EDITING

In our final experiment, we examine how our model performs on domains for which execution-guided synthesis is possible, but requires extensive changes to the underlying DSL.

For example, in the RobustFill DSL, a function `getSubStr(i,j)` slices a string from index i to index j. This function is not executable until both i and j are known. In order to perform execution-guided synthesis, Ellis et al. (2019) needed to replace `getSubString` with two separate functions: `getSubStrStart_i` and `getSubStrEnd_j`, where each half can be executed in the REPL. This process must be performed manually for every language construct which takes multiple arguments.

Here we seek to answer the question: can our model be used to successfully synthesize programs using the language as-is? To this end, we implement the code-writing policy using the DSL presented in Devlin et al. (2017) without modification (example programs in Figure 5 left). We do not expect that our approach would outperform the REPL system in Ellis et al. (2019), but we hope that it could achieve much of the gains. We additionally compare against another relevant baseline: RobustFill (Devlin et al., 2017). In contrast to the original paper, we train the RobustFill model using the same "unmodified" version of the DSL as our model, whose syntax has not been modified to aid with prediction. At test time, we used a sample-based search procedure, because the branching factor is prohibitively large for breadth-first search procedures explored above. While the blended encoding does not achieve the accuracy of the execution-guided REPL system, it outperforms the other baselines, including the RobustFill model, neural abstract semantics and the RNN baseline.

6 CONCLUSION

We introduced blended abstract semantics, a method for representing partially written code based on concrete execution and learned approximate semantics. We demonstrated how our approach, which combines abstract interpretation with representation learning, can be trained end-to-end as the basis for search policies and search heuristics. In program synthesis tasks, models equipped with blended abstract semantics outperformed neural baselines in several domains. Immediate future directions include exploring the use of blended abstract semantics for other synthesis tasks, including programming from language instruction and bug fixing. More generally, we hope that approaches which integrate learning and symbolic methods can be used to build systems which can write code more effectively and robustly.

REFERENCES

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Sygus-comp 2016: Results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.

Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 39–48, 2016.

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

Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490*, 2018.

Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. *ICLR*, 2018.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. *arXiv preprint arXiv:1703.07469*, 2017.

Li Dong and Mirella Lapata. Coarse-to-fine decoding for neural semantic parsing. *arXiv preprint arXiv:1805.04793*, 2018.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A Smith. Recurrent neural network grammars. *arXiv preprint arXiv:1602.07776*, 2016.

Kevin Ellis, Maxwell Nye, Yewen Pu, Felix Sosa, Josh Tenenbaum, and Armando Solar-Lezama. Write, execute, assess: Program synthesis with a repl. In *Advances in Neural Information Processing Systems*, pp. 9165–9174, 2019.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Luc Cary, Lucas Morales, Luke Hewitt, Armando Solar-Lezama, and Joshua B Tenenbaum. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep bayesian program learning. *arXiv preprint arXiv:2006.08381*, 2020.

Qinheping Hu, Jason Breck, John Cyphert, Loris D'Antoni, and Thomas Reps. Proving unrealizability for syntax-guided synthesis. In *International Conference on Computer Aided Verification*, pp. 335–352. Springer, 2019.

Justin Johnson, Bharath Hariharan, Laurens Van Der Maaten, Judy Hoffman, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Inferring and executing programs for visual reasoning. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2989–2998, 2017.

Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. Neural sketch learning for conditional program generation. *arXiv preprint arXiv:1703.05698*, 2017.

Maxwell Nye, Luke Hewitt, Joshua Tenenbaum, and Armando Solar-Lezama. Learning to infer program sketches. *arXiv preprint arXiv:1902.06349*, 2019.

Augustus Odena and Charles Sutton. Learning to represent programs with property signatures. In *International Conference on Learning Representations*, 2019.

Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. *ICLR*, 2018.

Richard Socher, Cliff Chiung-Yu Lin, Andrew Y Ng, and Christopher D Manning. Parsing natural scenes and natural language with recursive neural networks. In *ICML*, 2011.

Armando Solar-Lezama. *Program synthesis by sketching*. PhD thesis, University of California, Berkeley, 2008.

Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–30, 2017.

Amit Zohar and Lior Wolf. Automatic program synthesis of long programs with a learned garbage collector. In *Advances in Neural Information Processing Systems*, pp. 2098–2107, 2018.

A SEMANTICS

Here we fully define the semantics for concrete, neural, and blended semantics, covering details that were omitted in the main paper, namely, the semantics of lambda expressions.

Concrete semantics In this work, we consider domains where the underlying programming language is *functional*. Let $\lambda x.E$ be a lambda expression of one argument, and let $x = v$ be an assignment of the variable x to the value v. Lambda application is defined as follows:

$$(\lambda x.E)(v) = [\![E]\!]_{\{x=v\}}$$

That is, we evaluate the function body E, replacing all instances of the function variable x with the value v. For example, $[\![(\lambda x.x + x)(5)]\!]_{\{\}} = [\![x + x]\!]_{\{x=5\}} = 5\hat{+}5 = 10$. Where $\hat{+}$ denotes the execution semantics of the built-in function $+$. The concrete semantics $[\![\cdot]\!]$ is defined:

$$[\![E]\!]_{\mathcal{C}} = \begin{cases} [\![k]\!]_{\mathcal{C}} = k & \text{a constant } k \\ [\![x]\!]_{\mathcal{C}\models x=v} = v & x = v \\ [\![f(E_1 \cdots)]\!]_{\mathcal{C}} = \hat{f}([\![E_1]\!]_{\mathcal{C}} \cdots) & \text{executing built-in } f \\ [\![(\lambda x_1 \cdots x_n.E)(E_1, \cdots, E_n)]\!]_{\mathcal{C}} = [\![E]\!]_{\mathcal{C}\cup\{x_1=[\![E_1]\!]_{\mathcal{C}},\cdots,x_n=[\![E_n]\!]_{\mathcal{C}}\}} & \text{lambda application} \end{cases}$$

Neural semantics For built-in functions f, we use f^{nn}, a neural module function of k vector inputs, where k is the arity of f.

In our domains, lambda expressions are only used as arguments to higher-order functions. Therefore, since we will never apply a lambda expression directly under neural semantics, we only require vector representation of lambdas. However, we still require a mechanism to represent arbitrary lambda expressions built combinatorially from primitive functions. This representation is constructed in a modular fashion by encoding the body E of the lambda expression.

$$[\![E]\!]^{nn}_{\mathcal{C}} = \begin{cases} [\![k]\!]^{nn}_{\mathcal{C}} = \textsc{Embed}(k) & \text{a constant } k \text{ is embedded} \\ [\![x]\!]^{nn}_{\mathcal{C}\models x=v} = \textsc{Embed}(v) & \text{embed the value } v \text{ of the variable } x \\ [\![\texttt{HOLE}]\!]^{nn}_{\mathcal{C}} = h(\mathcal{C}) & \text{a neural placeholder for a hole based on context} \\ [\![f(E_1 \cdots)]\!]^{nn}_{\mathcal{C}} = f^{nn}([\![E_1]\!]^{nn}_{\mathcal{C}} \cdots) & \text{using neural module} \\ [\![\lambda x.E]\!]^{nn}_{\mathcal{C}} = [\![E]\!]^{nn}_{\mathcal{C}\cup\{x=\text{null}\}} & \text{encode the body of a lambda} \end{cases}$$

Note that in representing a lambda expression, we used the context with the assignment $x = \text{null}$. This is used to account for the fact that, in the time of the lambda function's definition (as an argument to a higher-order function), its argument is still unknown under neural execution.

The definition for blended semantics proceeds in an analogous fashion, with concrete subtrees executed concretely.

B EXPERIMENTAL DETAILS

All models are trained with the AMSGrad (Reddi et al., 2018) variant of the Adam optimizer with a learning rate of 0.001. All RNNs are 1-layer and bidirectional GRUs, where the final hidden state is used as the output representation. Unless otherwise stated, holes are encoded by applying the EMBED function to the context, and then applying a type-specific neural module to the resulting vector.

B.1 TOWERS

We employ the tower-building domain and DSL introduced in Ellis et al. (2020), which consists of the basic commands: `PlaceHorizontalBlock`, `PlaceVerticalBlock`, `MoveHand(n)`, `ReverseHand()`, `Embed`, `Loop` and integers `n` from 1 to 8. (The higher-order `Embed` function takes an expression as input, executes it, and then returns the hand to its initial location.) In order to test the compatibility of our approach with library-learning techniques, we additionally use library functions learned by the DreamCoder system by combining the above functions. Following Ellis

et al. (2020), the grammar is implemented in continuation-passing style. Our training data consisted of tower programs randomly sampled from a PCFG generative model (Ellis et al., 2020).

We trained policy networks on 480000 programs. We trained value functions on 240000 rollouts from the policy. We perform search for up to 300 seconds per problem.

All neural modules consist of a single linear layer (input dimension $512 * n_{args}$ and output dimension 512) followed by ReLU activation. Tower images are embedded with a simple CNN-ReLU-MaxPool architecture, as in Ellis et al. (2020).

Hand-coded abstract interpretation We implemented an abstract domain which tracked, a) the range of possible locations of the "hand" and b) for each horizontal location, the minimum height which must be achieved by the partially constructed tower. This representation allows us to eliminate invalid partial programs because once a block is dropped, it cannot be removed through any subsequent commands.

B.2 LIST PROCESSING

Data for this domain was generated by modifying the DeepCoder dataset (Balog et al., 2016). Specifically, DeepCoder training programs of size 2 (containing 2-3 higher order functions, such as `map f (filter g input)` or `zipwith f (map g input) (map h input)`) were modified by changing the lambdas in the program (`f`, `g`, and `h` in the above examples) from a small set of constant lambdas such as `(*2)` to depth-3 lambdas sampled from our modified grammar (see below). For example: `(λx.max(x+2,x/2))`. For each program, 5 example input lists were sampled, each with length 10 and values in the range [-64, 64]. The program was then executed to yield the corresponding outputs. Programs with output or intermediate values outside of the range [-64, 64] were discarded. Programs producing the identity function or constant functions were also discarded. We trained and tested only on functions of type `[int] → [int]`. At test time when running search, we similarly reject programs with intermediate values outside of the desired integer range.

All policy networks were trained on 500000 programs. We perform search for 180 seconds per problem.

All neural modules consist of a single linear layer (input dimension $64 * n_{args}$ and output dimension 64) followed by ReLU activation. Integers are encoded digit-wise via a GRU. Lists are encoded via a GRU encoding over the representations of the integers they contain.

Unbound variables within a lambda function are embedded via a learned representation parameterized by the variable name (one vector representing x and one representing y). When encoding holes within lambda functions, we ignore context, and instead embed holes only as a function of the hole type.

Modified lambda grammar Below is the grammar used for lambda functions:

```
L → (λx,y.S) | (λx.S)
S → I | B
I → I+I | I*I | I/I | min(I,I) | max(I,I) | A
B → I > I | or(B,B) | and(B,B) | I%I==0
A → x | y | N
N → -2 | -1| 0 | 1 | 2
```

Variations on training and testing conditions Many variations on the training and testing conditions achieve similar results to those shown in the main paper (i.e., blended semantics consistently achieves the highest performance). Several of these variations are shown in Figure 8.

B.3 STRING EDITING

For the string editing tasks, we use the DSL from Devlin et al. (2017). We train on randomly sampled programs, sampling I/O pairs and propagating constraints from programs to inputs to ensure that

input strings are relevant for the target program (see Devlin et al. (2017)). We condition on 4 I/O examples for each program. We used string editing problems from the SyGuS (Alur et al., 2016) program synthesis competition as our test corpus.

We trained all models on 2 million training programs. At test time, we sample programs from the model for a maximum timeout of 30 seconds.

Input and output strings are encoded by embedding each character via a 20-dimensional character embedding and concatenating the resulting vectors to form a representation for each string. Representations of "expressions" e (as defined in the RobustFill DSL) are concatenated together using an "append" module. Following Ellis et al. (2019), neural modules consist of a single dense block with 5 layers and a growth rate of 128 (input dimension $256 * n_{args}$ and output dimension 256).

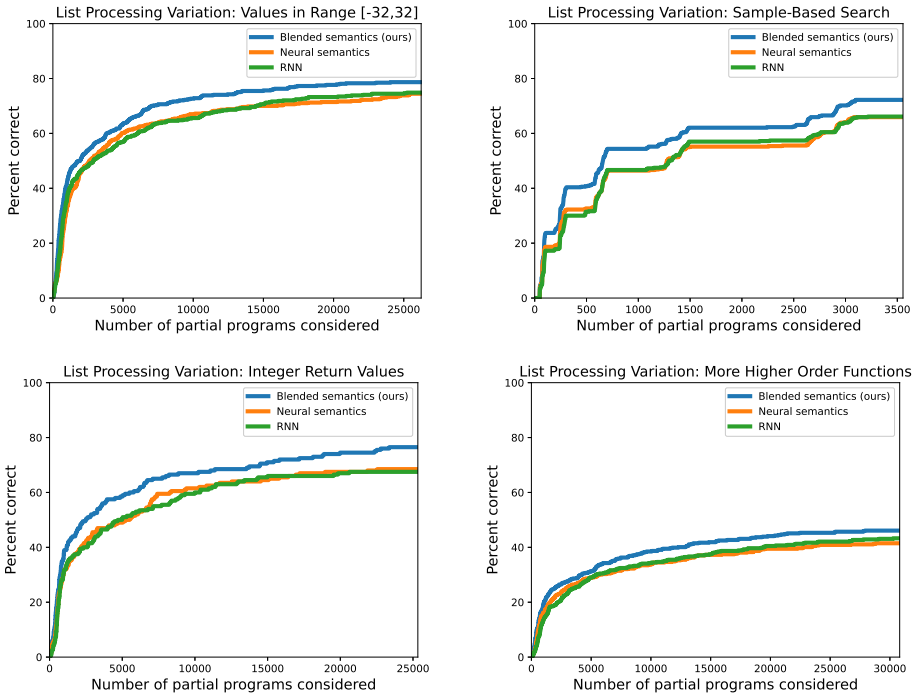


Figure 8: Variations on the list processing task. Top Left: using integer values in the range [-32, 32] instead of [-64, 64]. Top Right: Using sample-based search instead of best-first search. Bottom Left: extending the training and testing data to allow for `[int]` $\rightarrow$ `int` functions. Bottom Right: the original model tested on deeper DeepCoder programs with 3-6 higher-order functions. Notably, in all test conditions, the blended semantics consistently outperforms the RNN and neural semantics.