# Neurosymbolic Program Synthesis

Swarat CHAUDHURI
*The University of Texas at Austin*

**Abstract.** We survey *neurosymbolic program synthesis*, an emerging research area at the interface of deep learning and symbolic artificial intelligence. As in classical machine learning, the goal in neurosymbolic program synthesis is to learn functions from data. However, these functions are represented as *programs* that use symbolic primitives, often in conjunction with neural network components, and must, in some cases, satisfy certain additional behavioral constraints. The programs are induced using a combination of symbolic search and gradient-based optimization. In this survey, we categorize the main ways in which symbolic and neural learning techniques come together in this area. We also showcase the key advantages of the approach — specifically, greater reliability, interpretability, verifiability, and compositionality — over end-to-end deep learning.

**Keywords.** Neurosymbolic AI, Program Synthesis, Machine Learning

## 1. Introduction

Research on artificial intelligence (AI) has, for a long time, been split between the symbolists and the connectionists. The older symbolic agenda sought to describe the world using logic-based rules. In contrast, the machine learning (ML) perspective posited that patterns can be learned purely from data. In recent years, the ML perspective has tended to win out — in particular, thanks to breathtaking progress in the field of deep learning.

At the same time, concerns remain about the use of deep learning in real-world problems [1]. As opaque black boxes, deep neural networks are hard to trust and interpret. They cannot be easily taught to conform to commonsense knowledge that humans have. They cannot reuse components in the way traditional software can. Their training can be unreliable and fail in data-poor environments.

*Neurosymbolic* machine learning [1,2,3] is an increasingly prominent response to these concerns. The idea here is to inject human-held symbolic knowledge into a deep learning process. This knowledge can constrain the learned functions to use certain algorithmic abstractions, and can also serve as a *regularizer*, allowing more reliable learning.

In particular, this paper focuses on an emerging form of neurosymbolic learning called *neurosymbolic program synthesis* (NSP). NSP generalizes classical program synthesis [4], where the goal is to automatically discover programs that are written in a domain-specific language (DSL) and satisfy a logical specification. These programs are obtained through the composition of a set of symbolic modules with well-specified interpretations, using combinators that enforce natural requirements at the interface of different components. The program synthesis problem is solved using symbolic methods, e.g., discrete search and logic-based pruning.

NSP broadens both the problem statement and the solution space of program synthesis. The programs in NSP can be built entirely from symbolic primitives. Often, however, they use a mix of symbolic code and neural modules. The synthesis problem in NSP is to induce a program — including the program's discrete structure, or *architecture*, as well as the parameters of its neural components, if any — that minimizes a cost function. In some cases, the synthesized program is also required to satisfy a set of additional behavioral constraints like in classical program synthesis. This synthesis problem is solved using a variety of strategies that bring together deep learning (including the use of large language models [5,6]) and symbolic search and reasoning techniques.

## 2. Motivating Goals

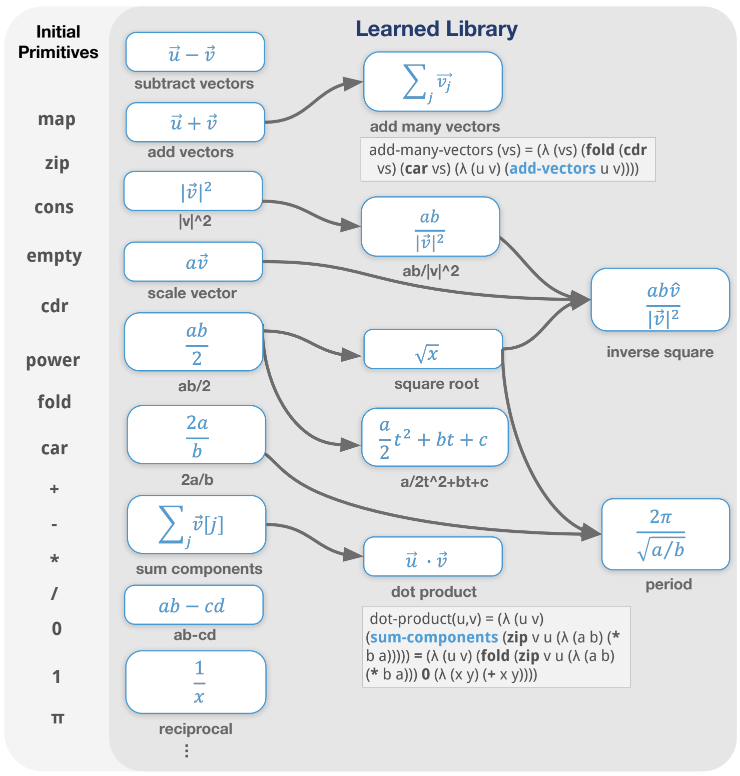
Now we elaborate on the main technical goals of NSP.

*Generalization.* Generalization is a foundational concept in machine learning. A learner *generalizes* if it can predict well on new test cases outside of its training set, ideally after training with a relatively small number of training samples. A key benefit of NSP is that neurosymbolic models can potentially generalize better than fully neural models.

For example, one way to improve the learner's ability to generalize is through *regularization*, which is a way of supplying an *inductive bias* — a preference order among hypotheses. Over the years, many regularization schemes for deep learning have been proposed. NSP introduces a new form of regularization by constraining the syntax and semantics of hypotheses using specifications (DSLs and behavioral constraints). This kind of *programmatic regularization* facilitates the use of domain knowledge. That is, a human user can encode insights about true solutions to the task in the language specification, biasing the learner towards hypotheses that are more likely to generalize.

Programmatic regularization is especially beneficial for procedural tasks, in which an agent performs sequences of actions realizing complex objectives, and for learning over structured data types like sequences and trees. In these settings, neural networks often fail to generalize from a modest number of examples. In contrast, one can often represent solutions to these tasks as programs that use high-level data and control abstractions. Such a task representation inductively generalizes *by construction*, for example, using looping primitives that operate over data of unbounded length or branching primitives that can break up a task into subtasks.

The PROPEL [7] approach to reinforcement learning (RL) is an example of this kind of regularization. The goal in RL is to learn a *policy* — a function mapping environment states to actions — for an agent exploring an unknown environment. In contrast to standard deep RL, a policy in PROPEL is a neurosymbolic program that additively combines a neural network and a symbolic program. The symbolic program is a composition of traditional control primitives (e.g., PID controllers) and thus captures expert knowledge. The neural net gives this program data-derived flexibility. PROPEL's learning algorithm simultaneously induces the neural and symbolic parts of the policy from data gathered via exploration. Compared to using a purely neural policy, this approach has provably lower variance (and is thus more sample-efficient) [7]. Empirically, policies learned through PROPEL tend to memorize fewer low-level details of the training environment (compared to neural policies), and thus generalize better to changes in the environment.



```
add-many-vectors (vs) = (λ (vs) (fold (cdr
vs) (car vs) (λ (u v) (add-vectors u v))))
```

```
dot-product(u,v) = (λ (u v)
(sum-components (zip v u (λ (a b) (*
b a))))) = (λ (u v) (fold (zip v u (λ (a b)
(* b a))) 0 (λ (x y) (+ x y))))
```

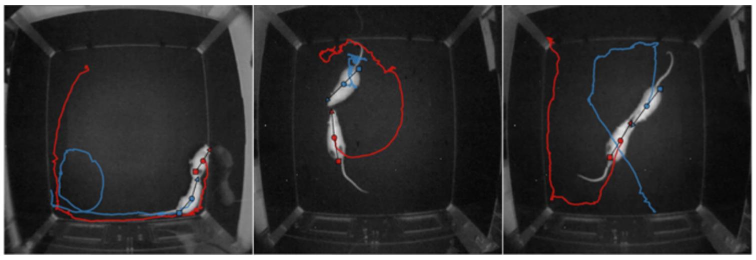
**Figure 1.** A sample output of DREAMCODER [8]. The system is tasked with synthesizing programs, written in a basic functional language, that fit random data simulated from physics equations. In synthesizing these programs it assembles a library of reusable procedures, including basic vector algebra and common patterns found in physics equations, which can be *transferred* to future synthesis tasks.

*Modular Learning.* Modern high-level programming is inherently modular. Programmers commonly break down complex coding tasks into subtasks that are solved using pre-existing library modules. An important promise of NSP is that it enables such modularity in learning using high-level programming primitives.

Specifically, NSP can use symbolic primitives to decompose a learning task into a composition of simpler tasks. These simpler tasks can be represented by either symbolic or neural modules. The modules can be learned afresh on a new task, or drawn from a "library" learned on previous tasks.

HOUDINI [9] and DREAMCODER [8] are two well-known frameworks for modular NSP. Both approaches exploit modularity to *transfer* knowledge across tasks. Specifically, HOUDINI is trained on a curriculum of tasks and iteratively grows a library of neural modules during this training. The synthesis algorithm in HOUDINI searches over programs that compose library modules, functional combinators such as **map** and **fold**, and fresh neural modules, and finds a program that solves the task. After outputting this program, the newly trained modules are added to the library for use in future tasks. On a set of tasks involving algorithmic computation over images, this approach outperforms neural approaches to transfer learning.

**(a)**



**(b)**

$$\textbf{map}\ (\textbf{lambda}\ x_t\,.\textbf{if}\ \mathit{DistAffine}_{[.0217];-.2785}(x_t)$$
$$\textbf{then}\ \mathit{AccAffine}_{[-.0007,.0055,.0051,-.0025];3.7426}(x_t)$$
$$\textbf{else}\ \mathit{DistAffine}_{[-.2143];1.822)}(x_t))\ x$$

**Figure 2.** (a) Stills from the CRIM13 behavior-tracking dataset for mice. (b) Programmatic classifier for a "sniff" action, learned using NEAR [10]. *DistAffine* and *AccAffine* are functions that first select the parts of the input that represent distance and acceleration measurements, respectively (the subscripts represent numerical parameters). The program has an accuracy of 0.87 (vs. 0.89 for a recurrent neural network baseline) and can be interpreted as: if the distance between two mice is small, they are doing a "sniff". Otherwise, they are doing a "sniff" if the difference between their accelerations is small.

DREAMCODER [8] (Figure 1) exploits modularity for transfer learning using symbolic modules. Starting with a set of first- and higher-order symbolic primitives and a corpus of training problems, DREAMCODER iteratively synthesizes code satisfying each training specification and mines this code for symbolic "templates" that can be reused in future tasks. The method is shown to solve a range of problems, including classic inductive programming, symbolic regression, and the design of complex scenes, that are beyond the scope of purely neural approaches.

*Interpretability.* *Interpretability* is an important consideration in many machine learning applications. However, interpretability is often at odds with learning performance. Deep neural networks are performant but not interpretable. On the other hand, linear models and shallow decision trees are interpretable but lack performance on complex tasks.

The representation of machine learning models as programs is an alternative path to interpretability. Interpreting a model's functionality now amounts to reading its code step by step and understanding the semantics of the underlying language primitives. Symbolic program analysis techniques can also now be used to algorithmically discover the model's properties. At the same time, assuming a suitably sophisticated DSL, even short programs can express complex computations and perform tasks beyond the reach of shallow learning models.

For example, the NEAR [10] approach, which we will describe in more detail in Section 5, was used to discover interpretable models of animal behavior. Figure 2-(a) shows stills from a video of such behavior for laboratory mice. Figure 2-(b) illustrates a sample program that describes when two mice are doing a "sniff" action.

*Safety Verification.* Safety is increasingly a central concern in deep learning research. It is well-known that deep networks can fail badly on adversarial or unexpected inputs. At the same time, it is impossible to reason about their correctness using traditional manual techniques. This makes the deployment of deep networks in safety-critical applications a fraught endeavor.

*Automatic verification* of neural networks, using formal methods such as abstract interpretation and SMT-solving, is an increasingly popular response to this challenge. Unfortunately, formal methods tend to only be efficient for programs that are small or have an architecture that enables compositional verification of subproblems. Neural networks do not satisfy either property. As a result, verification of neural networks has so far only scaled to fairly small models.

In contrast, approaches based on NSP can use high-level abstractions to facilitate verification. So far, these methods have been primarily used to learn programmatic policies for RL agents that provably satisfy invariants such as smoothness [11,7] and stability [12], and in some cases, more comprehensive functional correctness requirements [12].

Some of these methods, such as PIRL [11] and Viper [12], learn fully symbolic programs that can be verified using classical formal methods. However, even when learned programs have neural modules, they can sometimes be soundly approximated by symbolic programs. For example, Zhu et al. [13] and Anderson et al. [14] consider neurosymbolic policies of the form "**if** $(\psi(x, f(x))$ **then** $f(x)$ **else** $g(x)$", where $g$ is a neural network and $f$ and $\psi$ are short symbolic expressions. To verify such programs, they abstract the neural component $g$ with a nondeterministic choice, then automatically verify that the resulting symbolic program is safe under all instantiations of this choice. If it is, then the original neurosymbolic program is safe as well.

## 3. Applications

The technical goals described in Section 2 cut across many real-world applications. Now we present an incomplete list of some of these applications.

*Scientific Discovery.* Building learning algorithms that discover new scientific hypotheses and guide experiments is a grand challenge in AI. Such algorithms must respect constraints known to hold in the world and produce outputs that scientists can interpret. This makes NSP a natural fit to this space.

For example, Cranmer et al. [15] proposed a method for *symbolic regression* — the automatic discovery of symbolic equations from data — and apply it to a task in cosmology (dark matter prediction). Also, some of the NSP efforts mentioned in this survey were applied in the behavior analysis of laboratory animals. Specifically, NEAR [10], described in Section 5, was used to classify sequential animal behaviors. The NSP-based representation learning method [16] we describe in Section 4 as an exemplar of "symbolically guided deep learning" was used for interpretable clustering of such behaviors. FunSearch [5], an NSP method that uses large language models to direct evolutionary searches over programs, was utilized to discover witnesses to mathematical statements. LaSR [6], which augments a FunSearch-like method with a form of LLM-directed concept learning, was used to solve symbolic regression tasks, and in particular, discovered a new scaling law for LLMs.

*Programming Systems.* Over the last few years, code generation has emerged as one of the most successful applications of deep learning. Most approaches here generate programs token by token using a neural language model. While this strategy has worked remarkably well for high-level, API-manipulating code that follows idioms well-represented in publicly available code. However, generating code with novel structure, interpreting and satisfying sophisticated requirements, and maintaining consistency within large amounts of generated code remain challenges. Given that symbolic methods have long been used successfully to rigorously analyze programs and requirements and verify programs, NSP is a natural way to overcome some of these issues.

Recent work on code generation systems has begun to capitalize on some of this promise. Specifically, the emerging body of work on language-model *agents* for programming [17,18] and mathematical reasoning [19] do not just generate programs using a language model, but execute and analyze the resulting code and use the feedback from these processes to influence generation. As such, they can be seen as an instance of NSP.

*Dialog Systems.* Task-oriented dialog systems assist users with specific goals through a natural language interface. As digital assistants, they facilitate travel booking, database question answering, scheduling, and much more. The key challenge of task-oriented dialog is *state tracking* – identifying the user's intent and parameters in each dialog act, and using them to drive the system's actions. Fundamentally, dialog state is an intermediate symbolic representation that depends on complex, high-dimensional semantic context, namely dialog history and the underlying knowledge base or API. Thereby, NSP is a natural choice for modeling dialog state, successfully applied in many domains. For example, Andreas et al. [20] design a calendar assistant in which scheduling actions, dialog corrections, and exceptions are represented as compositional programs, synthesized by neurosymbolic models in context.

*Process Automation.* The field of *robotic process automation (RPA)* aims to automate procedural GUI workflows to facilitate business digitization and software testing. RPA agents interact with Web browsers, GUI applications, and APIs to accomplish the user's parameterized tasks. They are typically pretrained for each task using natural language commands, UI-grounded demonstrations, task completion rewards, or some combination thereof.

In RPA, the agent's state and action spaces are enormous – the current screen or Web page defines the state and the action space includes all possible interactions with its elements. Learning a robust and interpretable RPA agent is challenging even from grounded demonstrations as supervision. Instead, recent approaches leverage NSP ideas and model the agent as a neurosymbolic task program. For example, Srivastava et al. [21] combine neural language modeling with inductive program synthesis [4] to learn a generative model of programs that both guarantees consistency with the demonstrations and optimizes natural language alignment.

*Robotics and Control.* When designing policies or controllers for autonomous embodied systems, factors such as safety and data efficiency become paramount. For both low-level control and high-level planning problems, the standard practice has been to leverage symbolic domain knowledge (e.g., the governing equations of motion for the system, or an automaton representation of the high-level states) to design structured models that have certifiable guarantees, good generalization, or both (*e.g.*, Verma et al. [7]). An emerging research direction is to automatically learn or discover the structure of the symbolic knowledge (*e.g.*, Xu et al. [22]), which can be viewed as an instance of NSP.

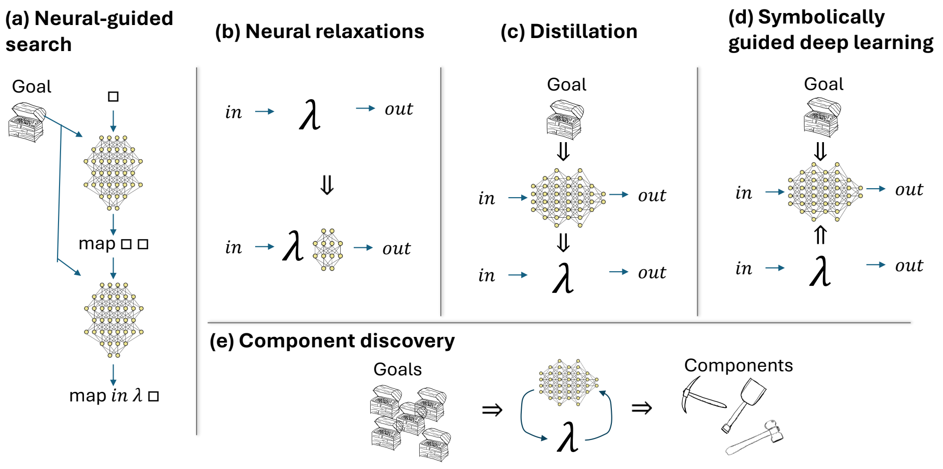


**Figure 3.** A taxonomy of learning algorithms in NSP. (a) shows neural-guided search algorithms, in which a neural network, trained using a corpus of programming tasks, is used to guide a search over programs. (b) illustrates *neural relaxation* methods, which approximate programs by differentiable neural functions. (c) depicts the dual distillation paradigm, in which neural functions are approximated by programs. (d) shows symbolically-guided deep learning, where a deep learning process incorporates synthesized symbolic information. (e) shows techniques for discovering new DSL components from synthesized programs.

## 4. Learning Algorithms

Now we sketch the main categories of learning algorithms used to synthesize programs in NSP. A high-level overview of these approaches is shown in Figure 3.

### *4.1. Starting Point: Search and Train*

Algorithms in NSP build on a substantial prior literature on search-based program synthesis [4]. Common algorithmic strategies in that literature include enumeration, sampling, version-space techniques, and the use of solvers. In *top-down* enumerative search [23,24], one repeatedly applies the rules of a DSL's grammar, starting with the initial nonterminal, until complete programs are generated. On the way, automatic deduction is used to recursively decompose the task specification into subgoals, as well as to rule out choices that provably fail to guarantee the specification. In *genetic programming* approaches [25], one maintains a pool of candidate programs and progressively modifies the pool using crossover and mutation operations. Bottom-up enumeration techniques [26] also maintain pools of candidates, updating the pool by composing simple programs into more complex ones.

There is also a body of methods that use stochastic methods, for example, Metropolis-Hastings sampling [27] and Sequential Monte Carlo, to effectively sample desirable programs from a space defined by a DSL. Version space methods use a specialized data structure to represent exponentially sized sets of programs. *Solver-driven* search [28] translates the space of programs and the specification into a joint constraint that is solved using a satisfiability solver.

The simplest learning methods in NSP use these search methods to generate program architectures and then train the resulting architectures using gradient descent. For such *search-and-train* methods to work, the programs must be *differentiable* in their parameters. This assumption does not hold in general but can be realized either by restricting the DSL or through the continuous relaxation techniques discussed in Section 4.3.

For example, the HOUDINI system [9] uses a search-and-train approach. HOUDINI uses top-down enumerative search to iterate over strongly-typed, differentiable program architectures, using a type system to significantly prune the search space on the way. For each architecture that is fully generated, the parameters of these modules are learned using gradient descent.

## 4.2. Neural-Guided Search

The main challenge with search-and-train approaches is scalability. Symbolic methods can rule out untenable programs using logical techniques. However, they do not give a way to find optimal architectures and are of limited value when the synthesis problem is not tightly constrained by logical requirements. As a result, recent research has studied ways to accelerate the search over programs using neural techniques. Collectively, these algorithmic techniques form the core of the area of NSP.

The most well-established of these methods are naturally described as *neural-guided search*. Here, one learns, from data, neural *policies* that generate probability distributions over the space of program architectures. These distributions can then be used to guide a combinatorial search over programs.

ROBUSTFILL [29], a neural-guided method for programming by example, was an early exemplar of such a method. ROBUSTFILL uses an encoder-decoder model with an attention mechanism to learn a distribution over programs conditioned on input-outputs. Next, it uses a beam search to generate programs from the learned distribution.

The neural models used in ROBUSTFILL were relatively small. Over the last few years, much larger transformer models trained on large code corpora have been used to guide program generation. There are many such methods; here we mention one, Alphacode [30]. This method solves programming problems using an encoder-decoder neural network that is pretrained on open-source code corpora available on the internet, then finetuned on programming competition problems. In addition to the basic neural-guided program generation strategy, the method uses additional innovations, such as the use of a neural network to generate candidate inputs for synthesized programs. Programs that exhibit the same behavior on these generated inputs are clustered into equivalence classes; from each equivalence class, a single member is selected for final use.

Later efforts have used more sophisticated stochastic search methods — in particular, reinforcement learning (RL) — to discover policies for program synthesis [31,32]. The challenge here is that sampling over programs is difficult, as "bad" programs vastly outnumber the "good" ones — which means stochastic methods may need massive numbers of samples to learn to find good programs. To overcome this problem, Bunel et al. [31] first train a supervised model for program generation. Then they use the REINFORCE algorithm to fine-tune the model by sampling programs from the distribution and giving a positive reward for a program that produces the desired output. Chen et al. [32] use automated deduction to use automated deduction to prune out subspaces of undesirable programs as the search progresses, and also to give the learner a signal regarding the kind of programs to avoid exploring.

Such RL methods have been recently scaled in efforts such as AlphaTensor [33] and AlphaDev [34]. In particular, AlphaDev studies the problem of discovering, given an assembly-language program, a *low-latency* equivalent program. This problem is framed as an RL task in which the reward depends on both correctness and latency. Innovations include the use of separate encoders for program and memory, and a "two-headed" RL value function in which one head is used for latency and the other for correctness.

Recent approaches to program synthesis and mathematical reasoning that combine language modeling, tool use, and search can also be seen to follow the general paradigm of neural-guided search. For example, SWE-agent [17] repeatedly invokes a large language model (LLM) that is allowed to use symbolic software tools, such as those for creating and editing code files, navigating software repositories, and executing tests and other programs. It is shown that this approach substantially exceeds the performance achieved via state-of-the-art non-interactive language models. Thakur et al. [19] use an LLM-guided search for formal theorem-proving in frameworks like Lean [35]. The FunSearch [5] and LaSR [6] approaches use LLMs to direct an evolutionary search over programs in the application context of mathematical and scientific discovery.

### *4.3. Neural relaxations*

A different category of methods *relaxes* the program learning problem into a smooth optimization problem, then solves using gradient descent. Such a strategy can be valuable for two reasons. First, the DSL may permit architectures that use non-differentiable constructs (*e.g.*, branching or sampling operations). Relaxations allow the parameters of such programs to be learned using gradient-based methods. Second, in some cases, one can use smooth relaxations of the discrete landscape of program architectures into a continuous landscape that allows easier optimization.

Smooth interpretation [36,37] was an early example of program relaxations in parameter learning. This method defines a *smooth operational semantics* of programs that amounts to stochastically perturbing the program's input and taking the expectation of the (probabilistic) output on this input. This semantics is now used to produce a signal for a continuous optimizer.

NEAR [10] points to a different way of using neural relaxations. The goal here is to find a program that optimally fits a labeled dataset. To meet this goal, NEAR derives programs top-down. Incomplete programs encountered during this derivation process are approximated by differentiable neurosymbolic functions that are trained end to end, and the training loss is used as a *heuristic function* that can guide informed searches (such as $A^*$) over programs. We describe this method in more depth in Section 5.

Cui et al. [38] have extended NEAR into an end-to-end differentiable method for program synthesis. Building on prior work on differentiable architecture search, this methods models the entire DSL as a differentiable program. That is, at each node in the search graph, the categorical choice between different edges out of the node is relaxed into a trainable softmax. The gradient updates for these softmax weights are guided by a lower-level signal from neural relaxations.

### *4.4. Distillation*

In the related *distillation* paradigm [11,7,12,14,13], one starts by relaxing the set of all programs in the DSL into a differentiable, overparameterized space of functions represented by a neural network. Because the neural network can be assumed to be more

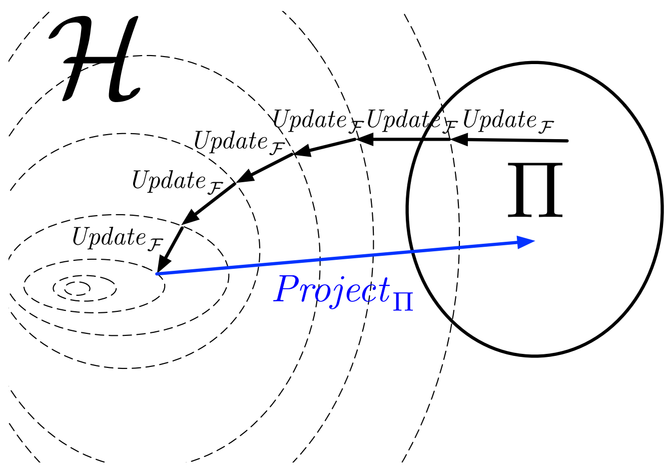


**Figure 4.** The PROPEL algorithm [7]. $\mathscr{H}$ is the relaxed space; $\Pi$ is the space of DSL programs. *Update*$_{\mathscr{F}}$ represents gradient updates; *Project*$_{\Pi}$ is the distillation operation.

expressive than the DSL, every program in the DSL has a representation as a parameterization of the network. At the same time, the neural representation can be optimized using gradient-based methods. The learning algorithm uses this fact to learn an optimal function $g$ in the relaxed space. This function may not have a symbolic representation in the DSL, hence it is *distilled* into a DSL program whose behavior is as close as possible to $g$.

The PIRL [11] and VIPER [12] methods for "programmatic" reinforcement learning (RL) were the first distillation approaches to NSP. The goal in both methods is to learn a policy, expressed in a symbolic DSL, that maximizes an RL agent's long-term reward. Solving this problem requires program synthesis as well as computationally complex planning over a long time horizon. PIRL and VIPER first eliminate the program synthesis dimension of the problem by learning a relaxed neural policy using a standard deep RL algorithm. Distilling this policy into a symbolic one is now framed as *imitation learning* problem, which does not require planning and can be reduced to a supervised learning task. This supervised learning problem is solved using an extension of classic program synthesis techniques.

PROPEL [7] (Figure 4) repeatedly performs distillation and relaxation during training. The method considers two policy classes: the class of policies representable as programs in the DSL, and the relaxed class of policies of the form $f(x)+g(x)$, where $f(x)$ is a neural network and $g(x)$ is a program in the DSL. The discovery of an optimal programmatic policy is framed as a form of mirror descent. In every learning step, the algorithm first applies a series of gradient updates to the neural component of the current neurosymbolic policy. Next, using distillation techniques like in PIRL, the algorithm distills this neurosymbolic policy into a program from the DSL. This program is then lifted back into the relaxed policy representation by adding a neural term.

Recent work has extended these ideas to the setting of formally verified learning. Specifically, Zhu et al. [13] give a method similar to PIRL but uses a distillation step that is guaranteed to produce a formally verified program. Anderson et al. [14] extend PROPEL so that distillation, relaxation, and gradient updates in the relaxed policy space all preserve the invariant of provable safety. While these methods learn using neurosymbolic representations, they only verify the distilled purely symbolic programs.

## *4.5. Symbolically Guided Deep Learning*

A fourth category of methods uses synthesized programs to inject human-held knowledge into a deep learning process. In particular, such programs can be used to: (i) augment latent vector representations learned through neural methods, and (ii) to automatically construct loss functions used to train deep networks.

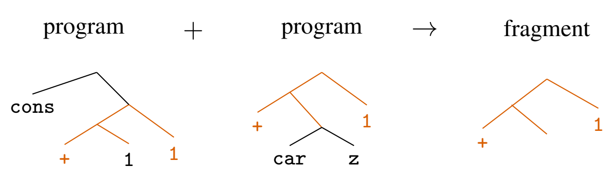
The *neurosymbolic encoders* presented by Zhan et al. [16] exemplify the former strategy. The approach modifies the variational autoencoder (VAE) framework, where a model comprises an encoder that compresses inputs into latent vectors and a decoder that decompresses these vectors back. It replaces the neural encoder with a "parallel composition" of a neural encoder and a symbolic program — i.e., the latent encoding now also includes some "bits" computed using the program. The neural and symbolic encoders and trained jointly with the neural decoder in an alternating optimization scheme that builds on NEAR. The method is shown to build representations that are significantly more well-factorized than those from standard VAEs, and demonstrated to lead to practical gains on downstream analysis tasks, such as for behavior classification.

AUTOSWAP, by Tseng et al. [39], is an instance of the second strategy. This method falls in the paradigm of weakly supervised learning, where *labeling functions* are used to automatically annotate data. While labeling functions are traditionally hand-written, AUTOSWAP uses the NEAR method to automatically discover task-level labeling functions from a small labeled dataset. A key technical idea is the use of a novel structural diversity cost that allows for the synthesis of *diverse* sets of LFs. The paper shows that AUTOSWAP outperforms existing approaches to behavior analysis using only a fraction of the data.

## *4.6. Component Discovery*

So far, we have discussed methods for learning programs for a predefined language. Now we proceed to neurosymbolic methods for learning DSLs. In general, a DSL consists of some universal combinators (*e.g.*, constructs for looping, branching, or processing basic data structures) and a "library" of reusable modules. These methods induce these reusable modules — either neural, symbolic, or both — and reuse them in new program learning tasks.

*Learning neural modules.* Methods for transferring neural modules start with a "pre-training" phase in which the module library is constructed, followed by a "deployment" or test-time phase when this library is used to solve new problems. During the "pre-training" phase, an inventory of small neural networks are composed to solve training synthesis tasks in tandem with training the weights of each network. The size of this bank of neural modules can either be fixed ahead of time — such as in modular meta-learning [40] and memoised wake-sleep [41] — or grow in size after each training experience, such as in HOUDINI [9]. At the test time or "deployment" phase, these neural modules can be used wholesale, with frozen weights [9]. Alternatively, their pretrained weights can be adapted to to each new task [40]. In either case, they do not need massive amounts of data to learn at test time.



| Example synthesized programs | Proposed library expression |
|---|---|
| `(λ (s) (map (λ (x)` `(if (= x '.') '-' x))) s)` `(λ (s) (map (λ (x)` `(if (= x '-') ',' x))) s)` | `(λ (s) (map (λ (x)` `(if (= x α) β x))) s)` |

**Figure 5. Top:** syntax trees of two programs sharing common structure, highlighted in orange, from which DreamCoder [8] can extract a reused fragment (bottom). The programs may be constructed through the *refactoring* of known programs using a set of rewrite rules. **Bottom:** two actual programs, and the generalized library abstraction created from their shared fragment.

*Learning Symbolic Modules.* There are also methods for inducing a library of reusable symbolic code. The most popular strategy is to compress out reused syntax trees across a training corpus of programs. DreamCoder [42] is a canonical example of such an approach. As outlined before, it is initialized with a set of basic primitives and a corpus of training tasks. It tackles these training tasks by jointly learning a neural network that can guide the synthesis of programs that perform these tasks, while also building a library of reused symbolic functions by mining synthesized programs for templates. The latter task is performed via a deductive method that uses a set of semantics-preserving rewrite rules refactor a set of existing programs and identify shared structure. Figure 5 illustrates a special case.

The subsequent Stitch approach [43] takes on the same problem as Dream-Coder, but discovers abstractions via a top-down search rather than a deductive refactoring-based method. It is shown to be several orders of magnitude faster than DreamCoder.

## 5. A Deeper Dive: Near

Now we do a deeper dive into a specific Nsp approach, Near [10]. Here, we spell out the problem that the method targets as well as its algorithmic approach. Figure 2 illustrates a sample problem solved using Near. See the original paper for more detailed experimental results.

*Problem Statement.* Near takes on a supervised learning task: given a dataset of labeled examples and a cost function, it seeks to find a program in a given DSL that minimizes the cost function.

Let us view a program abstractly as a pair $(\alpha, \theta)$, where $\alpha$ is a discrete *program structure* (or *architecture*) and $\theta$ is a vector of real-valued parameters. It is assumed that the architecture $\alpha$ is generated using a *context-free grammar* [44]. The grammar starts with an initial nonterminal, then iteratively applies the rules to produce a series of *partial architectures*: sentences made from one or more nonterminals and zero or more terminals. The process continues until there are no nonterminals left, i.e., we have a complete architecture.

The *semantics* of the architecture $\alpha$ is given by a function $[\![\alpha]\!](x,\theta)$, defined by rules that are fixed for the DSL. This function is required to be differentiable in $\theta$. Also, a *structural cost* is defined for architectures. Let each rule $r$ in the DSL grammar have a non-negative real cost $s(r)$. The structural cost of $\alpha$ is defined us $s(\alpha) = \sum_{r \in \mathscr{R}(\alpha)} s(r)$, where $\mathscr{R}(\alpha)$ is the multiset of rules used to create $\alpha$. Intuitively, architectures with lower structural costs are simpler are more human-interpretable.

Now we formalize the learning problem in NEAR. Let us assume an unknown distribution $D(x,y)$ over inputs $x$ and labels $y$, and consider the prediction error function $\zeta(\alpha,\theta) = \mathbf{E}_{(x,y)\sim D}[\mathbf{1}([\![\alpha]\!](x,\theta) \neq y)]$, where $\mathbf{1}$ is the indicator function. The learning problem, then, is to find an architecturally simple program with low prediction error, i.e., to solve the optimization problem:

$$(\alpha^*, \theta^*) = \underset{(\alpha,\theta)}{\arg\min}(s(\alpha) + \zeta(\alpha,\theta)). \qquad (1)$$

*Program synthesis as Graph Search.* NEAR piggybacks an informed search on a *program derivation graph* that models the top-down construction of program architectures. The nodes in this graph are incomplete architectures; at a node $u$, the heuristic $h(u)$ approximates the *cost to go*, i.e., the additional cost incurred by the best sequence of program derivation actions that can be taken from this point on.

More precisely, NEAR considers a graph $\mathscr{G}$ in which:

- The node set consists of all partial and complete architectures permissible in the DSL.
- The *source node* $u_0$ is the empty architecture. Each complete architecture $\alpha$ is a *goal node*.
- Edges are directed and capture single-step applications of rules of the DSL. Edges can be divided into: (i) *internal edges* $(u,u')$ between partial architectures $u$ and $u'$, and (ii) *goal edges* $(u,\alpha)$ between partial architecture $u$ and complete architecture $\alpha$. An internal edge $(u,u')$ exists if one can obtain $u'$ by substituting a nonterminal in $u$ following a rule of the DSL. A goal edge $(u,\alpha)$ exists if we can complete $u$ into $\alpha$ by applying a rule of the DSL.
- The cost of an internal edge $(u,u')$ is given by the structural cost $s(r)$, where $r$ is the rule used to construct $u'$ from $u$. The cost of a goal edge $(u,\alpha)$ is $s(r) + \zeta(\alpha,\theta^*)$, where $\theta^* = \arg\min_\theta \zeta(\alpha,\theta)$ and $r$ is the rule used to construct $\alpha$ from $u$.

An example of such a graph appears in Figure 6.

Let a *completion* of a partial architecture $u$ be a (complete) architecture $u[\alpha_1,\ldots,\alpha_k]$ obtained by replacing the nonterminals in $u$ by suitably typed architectures $\alpha_i$. Let $\theta_u$ be the parameters of $u$ and $\theta$ be parameters of the $\alpha_i$-s. The *cost-to-go* at $u$ is now given by:

$$J(u) = \min_{\alpha_1,\ldots,\alpha_k,\theta_u,\theta}((s(u[\alpha_1,\ldots,\alpha_k] - s(u)) + \zeta(u[\alpha_1,\ldots,\alpha_k],(\theta_u,\theta)) \qquad (2)$$

where the structural cost $s(u)$ is the sum of the costs of the grammatical rules used to construct $u$.

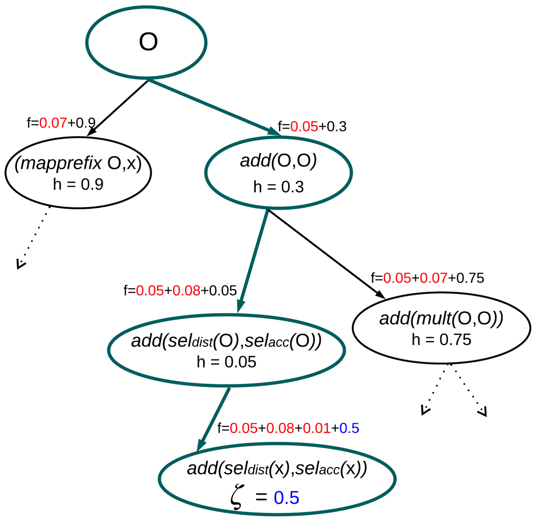


**Figure 6.** An example of program synthesis formulated as graph search [10]. Structural costs are in red, heuristic values in black, prediction errors $\zeta$ in blue, O refers to a nonterminal in a partial architecture, and the path to a goal node returned by A* search with NEAR heuristics is in teal.

*Neural Heuristics.* To compute a heuristic cost $h(u)$ for a partial architecture $u$ encountered during search, NEAR substitutes the nonterminals in $u$ with neural networks parameterized by $\omega$. These networks are *type-correct* — for example, if a nonterminal is supposed to generate subexpressions whose inputs are sequences, then the neural network used in its place is recurrent.

Let us view the neurosymbolic programs resulting from this substitution as tuples $(u, (\theta_u, \omega))$. One can define a semantics for such programs by extending our DSL's semantics, and lift the function $\zeta$ to assign costs $\zeta(u, (\theta_u, \omega))$ to such programs. The heuristic cost for $u$ is now given by:

$$h(u) = \min_{w,\theta} \zeta(u, (\theta_u, \omega)). \tag{3}$$

As $\zeta(u, (\theta_u, \omega))$ is differentiable in $\omega$ and $\theta_u$, $h(u)$ can be computed using gradient descent.

*$\varepsilon$-Admissibility.* The neural networks used to guide the search may only form an approximate relaxation of the space of programs; also, the training of these networks may not reach global optima. To account for these errors, NEAR considers an approximate notion of admissibility [45]. For a fixed constant $\varepsilon > 0$, let an *$\varepsilon$-admissible heuristic* be a function $h^*(u)$ over architectures such that $h^*(u) \leq J(u) + \varepsilon$ for all $u$. Now consider any completion $u[\alpha_1, \ldots, \alpha_k]$ of an architecture $u$. As neural networks with adequate capacity are universal function approximators, there exist parameters $\omega^*$ for the neurosymbolic program such that for all $u, \alpha_1, \ldots, \alpha_k$, $\theta_u$, and $\theta$:

$$\zeta(u, (\theta_u, \omega^*)) \leq \zeta(u[\alpha_1, \ldots, \alpha_k], (\theta_u, \theta)) + \varepsilon. \tag{4}$$

Because edges in the search graph have non-negative costs, $s(u) \leq s(u[\alpha_1, \ldots, \alpha_k])$, implying:

$$\begin{aligned} h(u) &\leq \min_{\alpha_1,\ldots,\alpha_k,\theta_u,\theta} \zeta(u[\alpha_1,\ldots,\alpha_k],(\theta_u,\theta)) + \varepsilon \\ &\leq \min_{\alpha_1,\ldots,\alpha_k,\theta_u,\theta} \zeta(u[\alpha_1,\ldots,\alpha_k],(\theta_u,\theta)) + (s(u[\alpha_1,\ldots,\alpha_k]) - s(u)) + \varepsilon \\ &= J(u) + \varepsilon. \end{aligned} \tag{5}$$

In other words, $h(u)$ is $\varepsilon$-admissible.

## 6. Conclusion

We have given an overview of NSP, an emerging field at the intersection of deep learning and program synthesis. We have sketched the main technical goals of the area and listed some of its applications. We have described a taxonomy of algorithmic approaches in the area and described NEAR as an exemplar of such approaches.

*Limitations and Challenges.* NSP-based approaches face two broad challenges. The first challenge is *scalability*. Searching over program architectures is a combinatorially hard problem. When the architectures follow broadly known idioms, neural language models trained on large code corpora can effectively guide this search. However, the generation of entirely novel program architectures remains a hard open problem.

The second challenge is *specification*. NSP assumes that humans can specify domain knowledge effectively via DSLs and behavioral constraints. While these specifications can help with generalization, constructing the "right" specification for a domain is nontrivial, and unnecessarily restrictive specifications can impede learning performance.

LLMs form a powerful response to both challenges. In the best case, an LLM can generate entire program architectures step by step, eliminating the need for explicit search. However, many applications of neurosymbolic programming require the creation of *novel* programs, rather than programs well-represented in the pretraining corpus. However, methods such as Funsearch [5] and LaSR [6] illustrate that the priors and abstraction capabilities of LLMs can prove useful even when novelty is desirable. As for specification, LLMs have been successfully used in the *autoformalization* of informal mathematical statements [46]. One can imagine such autoformalization techniques being more generally used to construct specifications for NSP.

Collectively, these two limitations imply that there are many tasks — for example, answering complex, non-synthetic questions over open-domain text, or the discovery of programmatic policies for real-world, learning-enabled robots — in which NSP is applicable in principle but not yet in practice. For NSP to reach its full potential, there needs to be progress on both fronts. Specifically, algorithmic innovations that enable the generation of more complex program structures are required. Also, more research must be done on systematizing the inductive bias that different kinds of specifications introduce, precisely evaluating their benefits with respect to goals such as interpretability and safety verification, and simplifying or automating the design of such specifications.

*Future Horizons.* Overall, we are excited about the trajectory of NSP. Data-efficiency, safety, and interpretability are increasingly salient issues in ML, and these issues open up a window of opportunity for fresh learning approaches like NSP. As the field further develops methods to address the aforementioned limitations, another opportunity is creating software frameworks — equivalents of Pytorch [47] or Keras [48] for NSP— that can accelerate the adoption of these methods. Standardized benchmarks and competitions that showcase the value of neurosymbolic methods and are also of interest to the broader AI community would also be of great value. ARC [49], the AI Math Olympiad, and Natural Plan [50] are three examples of such challenges. Finally, NSP researchers should prioritize the development of algorithms and infrastructures that enable their methods to scale. Of particular interest are new methods to parallelize search over program architectures and parameters and new interfaces for interacting between neural models (in particular, large language models) and symbolic tools.

## References

[1] Marcus G, Davis E. Rebooting AI: Building artificial intelligence we can trust. Pantheon; 2019.

[2] Garcez Ad, Lamb LC. Neurosymbolic AI: The 3rd Wave. arXiv preprint arXiv:201205876. 2020.

[3] Chaudhuri S, Ellis K, Polozov O, Singh R, Solar-Lezama A, Yue Y, et al. Neurosymbolic programming. Foundations and Trends® in Programming Languages. 2021;7(3):158-243.

[4] Gulwani S, Polozov O, Singh R. Program synthesis. Foundations and Trends in Programming Languages. 2017;4(1-2):1-119.

[5] Romera-Paredes B, Barekatain M, Novikov A, Balog M, Kumar MP, Dupont E, et al. Mathematical discoveries from program search with large language models. Nature. 2024;625(7995):468-75.

[6] Grayeli A, Sehgal A, Costilla-Reyes O, Cranmer M, Chaudhuri S. Symbolic regression with a learned concept library. In: Neural Information Processing Systems; 2024. .

[7] Verma A, Le HM, Yue Y, Chaudhuri S. Imitation-Projected Programmatic Reinforcement Learning. In: Neural Information Processing Systems (NeurIPS); 2019. .

[8] Ellis K, Wong C, Nye MI, Sablé-Meyer M, Morales L, Hewitt LB, et al. DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In: PLDI '21. ACM; 2021. p. 835-50.

[9] Valkov L, Chaudhari D, Srivastava A, Sutton C, Chaudhuri S. HOUDINI: Lifelong Learning as Program Synthesis. In: Neural Information Processing Systems (NeurIPS); 2018. p. 8701-12.

[10] Shah A, Zhan E, Sun JJ, Verma A, Yue Y, Chaudhuri S. Learning Differentiable Programs with Admissible Neural Heuristics. In: Advances in Neural Information Processing Systems; 2020. .

[11] Verma A, Murali V, Singh R, Kohli P, Chaudhuri S. Programmatically Interpretable Reinforcement Learning. In: ICML; 2018. p. 5052-61.

[12] Bastani O, Pu Y, Solar-Lezama A. Verifiable Reinforcement Learning via Policy Extraction. In: Neural Information Processing Systems (NeurIPS); 2018. p. 2499-509.

[13] Zhu H, Xiong Z, Magill S, Jagannathan S. An Inductive Synthesis Framework for Verifiable Reinforcement Learning. In: PLDI; 2019. .

[14] Anderson G, Verma A, Dillig I, Chaudhuri S. Neurosymbolic Reinforcement Learning with Formally Verified Exploration. In: Neural Information Processesing Systems (NeurIPS); 2020. .

[15] Cranmer M, Sanchez-Gonzalez A, Battaglia P, Xu R, Cranmer K, Spergel D, et al. Discovering symbolic models from deep learning with inductive biases. In: Neural Information Processing Systems (NeurIPS); 2020. .

[16] Zhan E, Sun JJ, Kennedy A, Yue Y, Chaudhuri S. Unsupervised Learning of Neurosymbolic Encoders; 2021.

[17] Yang J, Jimenez CE, Wettig A, Lieret K, Yao S, Narasimhan K, et al. Swe-agent: Agent-computer interfaces enable automated software engineering. arXiv preprint arXiv:240515793. 2024.

[18] Zhang Y, Ruan H, Fan Z, Roychoudhury A. Autocoderover: Autonomous program improvement. arXiv preprint arXiv:240405427. 2024.

[19] Thakur A, Tsoukalas G, Wen Y, Xin J, Chaudhuri S. An In-Context Learning Agent for Formal Theorem-Proving; 2024. Available from: `https://arxiv.org/abs/2310.04353`.

[20] Andreas J, Bufe J, Burkett D, Chen C, Clausman J, Crawford J, et al. Task-Oriented Dialogue as Dataflow Synthesis. Transactions of the Association for Computational Linguistics. 2020 Dec;8:556–571.

[21] Srivastava S, Polozov O, Jojic N, Meek C. Learning Web-based Procedures by Reasoning over Explanations and Demonstrations in Context. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics; 2020. p. 7652-62.

[22] Xu D, Nair S, Zhu Y, Gao J, Garg A, Fei-Fei L, et al. Neural task programming: Learning to generalize across hierarchical tasks. In: 2018 IEEE International Conference on Robotics and Automation (ICRA). IEEE; 2018. p. 3795-802.

[23] Feser JK, Chaudhuri S, Dillig I. Synthesizing data structure transformations from input-output examples. In: ACM SIGPLAN Notices. vol. 50. ACM; 2015. p. 229-39.

[24] Albarghouthi A, Gulwani S, Kincaid Z. Recursive program synthesis. In: International conference on computer aided verification. Springer; 2013. p. 934-50.

[25] Langdon WB, Poli R. Foundations of genetic programming. Springer Science & Business Media; 2013.

[26] Udupa A, Raghavan A, Deshmukh JV, Mador-Haim S, Martin MM, Alur R. TRANSIT: specifying protocols with concolic snippets. ACM SIGPLAN Notices. 2013;48(6):287-96.

[27] Schkufza E, Sharma R, Aiken A. Stochastic superoptimization. In: ACM SIGARCH Computer Architecture News. vol. 41. ACM; 2013. p. 305-16.

[28] Solar-Lezama A, Jones C, Arnold G, Bodík R. Sketching Concurrent Datastructures. In: PLDI 08; 2008. .

[29] Devlin J, Uesato J, Bhupatiraju S, Singh R, Mohamed A, Kohli P. RobustFill: Neural Program Learning under Noisy I/O. In: ICML; 2017. .

[30] Li Y, Choi D, Chung J, Kushman N, Schrittwieser J, Leblond R, et al. Competition-level code generation with alphacode. Science. 2022;378(6624):1092-7.

[31] Bunel R, Hausknecht MJ, Devlin J, Singh R, Kohli P. Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net; 2018. .

[32] Chen Y, Wang C, Bastani O, Dillig I, Feng Y. Program Synthesis Using Deduction-Guided Reinforcement Learning. In: International Conference on Computer Aided Verification. Springer; 2020. p. 587-610.

[33] Fawzi A, Balog M, Huang A, Hubert T, Romera-Paredes B, Barekatain M, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. Nature. 2022;610(7930):47-53.

[34] Mankowitz DJ, Michi A, Zhernov A, Gelmi M, Selvi M, Paduraru C, et al. Faster sorting algorithms discovered using deep reinforcement learning. Nature. 2023;618(7964):257-63.

[35] Moura Ld, Ullrich S. The Lean 4 theorem prover and programming language. In: Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28. Springer; 2021. p. 625-35.

[36] Chaudhuri S, Solar-Lezama A. Smooth interpretation. In: PLDI; 2010. p. 279-91.

[37] Chaudhuri S, Solar-Lezama A. Smoothing a Program Soundly and Robustly. In: CAV; 2011. p. 277-92.

[38] Cui G, Zhu H. Differentiable Synthesis of Program Architectures. Advances in Neural Information Processing Systems. 2021;34.

[39] Tseng A, Sun JJ, Yue Y. Automatic synthesis of diverse weak supervision sources for behavior analysis. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition; 2022. p. 2211-20.

[40] Alet F, Lozano-Pérez T, Kaelbling LP. Modular meta-learning. Conference on Robot Learning (CoRL). 2018.

[41] Hewitt LB, Le TA, Tenenbaum JB. Learning to learn generative programs with Memoised Wake-Sleep. UAI. 2020.

[42] Ellis K, Wong C, Nye M, Sable-Meyer M, Cary L, Morales L, et al. Dreamcoder: Growing generalizable, interpretable knowledge with wake-sleep Bayesian program learning. arXiv preprint arXiv:200608381. 2020.

[43] Bowers M, Olausson TX, Wong L, Grand G, Tenenbaum JB, Ellis K, et al. Top-down synthesis for library learning. Proceedings of the ACM on Programming Languages. 2023;7(POPL):1182-213.

[44] Hopcroft JE, Motwani R, Ullman JD. Introduction to automata theory, languages, and computation, 3rd Edition. Pearson international edition. Addison-Wesley; 2007.

[45] Harris LR. The heuristic search under conditions of error. Artificial Intelligence. 1974;5(3):217-34.
[46] Wu Y, Jiang AQ, Li W, Rabe M, Staats C, Jamnik M, et al. Autoformalization with large language models. Advances in Neural Information Processing Systems. 2022;35:32353-68.
[47] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems. 2019;32.
[48] Chollet F. Deep learning with Python. Simon and Schuster; 2021.
[49] Chollet F. On the measure of intelligence. arXiv preprint arXiv:191101547. 2019.
[50] Zheng HS, Mishra S, Zhang H, Chen X, Chen M, Nova A, et al. NATURAL PLAN: Benchmarking LLMs on Natural Language Planning. arXiv preprint arXiv:240604520. 2024.