Program Synthesis using Conflict-Driven Learning

Yu Feng
University of Texas at Austin
U.S.A
yufeng@cs.utexas.edu

Ruben Martins*
Carnegie Mellon University
U.S.A
rubenm@cs.cmu.edu

Osbert Bastani
Massachusetts Institute of Technology
U.S.A
obastani@csail.mit.edu

Isil Dillig
University of Texas at Austin
U.S.A
isil@cs.utexas.edu

Abstract

We propose a new *conflict-driven* program synthesis technique that is capable of learning from past mistakes. Given a spurious program that violates the desired specification, our synthesis algorithm identifies the *root cause* of the conflict and learns new lemmas that can prevent similar mistakes in the future. Specifically, we introduce the notion of *equivalence modulo conflict* and show how this idea can be used to learn useful lemmas that allow the synthesizer to prune large parts of the search space. We have implemented a general-purpose CDCL-style program synthesizer called Neo and evaluate it in two different application domains, namely data wrangling in R and functional programming over lists. Our experiments demonstrate the substantial benefits of conflict-driven learning and show that Neo outperforms two state-of-the-art synthesis tools, Morpheus and DeepCoder, that target these respective domains.

CCS Concepts • **Software and its engineering → Programming by example**; **Automatic programming**;

Keywords program synthesis, conflict-driven learning, automated reasoning

ACM Reference Format:
Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis using Conflict-Driven Learning. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3192366.3192382

*Yu Feng and Ruben Martins contributed equally to this work.

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192382

1 Introduction

In recent years, there has been significant interest in program synthesis, in which the goal is to automatically generate programs from high-level specifications. Automated program synthesis has proven to be useful to both end-users and programmers: For instance, programming-by-example (PBE) has been used to automate tedious tasks that arise in everyday life, such as string and format manipulations in spreadsheets [13, 27] or data wrangling tasks on tabular and hierarchical data [8, 36, 39]. Program synthesis has also been used for improving programmer productivity by automatically completing parts of a program [25, 31, 32] or helping programmers use complex APIs [9, 15, 17].

While there are several different approaches to program synthesis, one common method is to perform enumerative search over the space of programs that can be generated using the context-free grammar of some DSL [1, 10, 35, 39]. In a nutshell, these techniques enumerate DSL programs according to some cost metric and return the first program that satisfies the user-provided specification. Several recent methods also combine enumerative search with deduction with the goal of ruling out *partial programs* [8, 10, 23].

Despite recent advances in program synthesis, a common shortcoming of existing techniques is that they are not able to *learn from past mistakes*. To understand what we mean by this, consider the input-output specification $[1, 2, 3] \mapsto [1, 2]$ and a candidate program of the form $\lambda x.\ \texttt{map}(x, \ldots)$. Here, it is easy to see that no program of this shape can satisfy the given specification, since the output list is shorter than the input list, but the `map` combinator yields an output list whose length is the same as the input list. In fact, we can take this generalization one step further and deduce that no program of the form $\lambda x.\texttt{f}(x, \ldots)$ can satisfy the specification *as long as* `f` yields a list whose length is greater than or equal to that of the input list. This kind of reasoning allows the synthesizer to learn from past mistakes (in this case, the spurious program $\lambda x.\ \texttt{map}(x, \ldots)$) and rule out many other erroneous programs (e.g., $\lambda x.\texttt{reverse}(x)$, $\lambda x.\ \texttt{sort}(x)$) that are guaranteed *not* to satisfy the desired specification.

In this paper, we present a new *conflict-driven* synthesis algorithm that is capable of learning from its past mistakes.

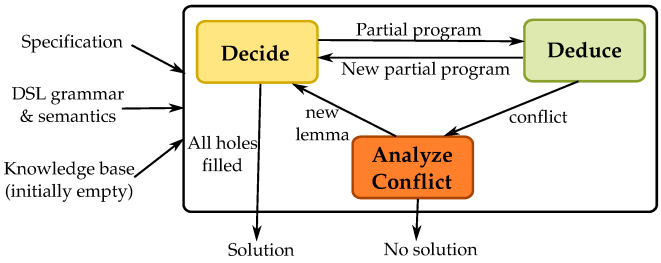


Figure 1. High-level architecture of our synthesis algorithm

Our method is inspired by the success of *conflict-driven learning* in automated theorem provers and analyzes conflicts to learn useful lemmas that guide the search. Furthermore, our method can synthesize programs over any arbitrary DSL and is not restricted to any particular application domain.

At a high level, the general structure of our synthesis algorithm resembles the architecture of SAT and SMT solvers based on *conflict-driven clause learning (CDCL)*. As shown in Figure 1, our synthesis algorithm consists of three key components, namely *Decide*, *Deduce*, and *AnalyzeConflict*:

- ***Decide:*** Given a partial program P with holes (representing unknown program fragments), the *Decide* component selects *which* hole to fill and determines *how* to fill it using the constructs in the DSL.
- ***Deduce:*** Given the current partial program, the *Deduce* component makes new inferences based on the syntax and semantics of the DSL as well as a *knowledge base*, which keeps track of useful "lemmas" learned during the execution of the algorithm.
- ***Analyze Conflict:*** When the *Deduce* component detects a conflict (meaning that the partial program is infeasible), the goal of *AnalyzeConflict* is to identify the *root cause* of failure and learn new lemmas that should be added to the knowledge base. Because the decisions made by the *Decide* component need to be consistent with the knowledge base, these lemmas prevent the algorithm from making similar bad decisions in the future.

Based on this discussion, the main technical contributions of this paper are two-fold: First, we introduce the paradigm of *synthesis using conflict driven learning* and propose a CDCL-style architecture for building program synthesizers. Second, we propose a new technique for analyzing conflicts and automatically learning useful lemmas that should be added to the knowledge base. Our learning algorithm is based on the novel notion of *equivalence modulo conflict*. In particular, given a spurious partial program P that uses DSL construct ("component") c, our conflict analysis procedure automatically infers other components $c_1, \ldots, c_n$ such that replacing c with any of these c_i's yields a spurious program P' with the same *root cause* of failure as P. We refer to such components as being *equivalent modulo conflict (EMC)* and our conflict analysis procedure infers a maximal set of EMC components from an infeasible partial program. Our learning algorithm then uses these equivalence classes to identify other infeasible partial programs and adds them as lemmas to the knowledge base. Because the assignments made by *Decide* must be consistent with the knowledge base, the lemmas learnt using *AnalyzeConflict* allow the synthesizer to prune a large number of programs from the search space.

We have implemented the proposed synthesis technique in a tool called Neo and evaluate it in two different application domains that have been explored in prior work: First, we use Neo to perform data wrangling tasks in R and compare Neo with Morpheus, a state-of-the-art synthesizer targeting this domain [8]. Second, we evaluate Neo in the domain of list manipulation programs and compare it against a re-implementation of DeepCoder, a state-of-the-art synthesizer based on deep learning [3]. Our experiments clearly demonstrate the benefits of learning from conflicts and show that our general-purpose synthesis algorithm outperforms state-of-the-art synthesizers that target these domains.

Contributions. To summarize, this paper makes the following key contributions:

- We propose a new CDCL-style framework for conflict-driven program synthesis.
- We introduce the notion of *equivalence modulo conflict* and show how it can be used to learn useful lemmas from conflicts.
- We implement the proposed ideas in a general-purpose synthesis tool called Neo and evaluate it in two different application domains.

2 Motivating Example

Suppose that we are given a list containing the scores of different teams in a soccer league, and our goal is to write a program to compute the total scores of the best k teams. For instance, if the input list is [49, 62, 82, 54, 76] and k is specified as 2, then the program should return 158 (i.e., 82 + 76). It is easy to see that the `computeKSum` procedure below (written in Haskell-like syntax) implements the desired functionality:

```
computeKSum :: List -> Int -> Int
computeKSum x1 x2 =
  -- Sort x1 in ascending order
  L1 <- sort x1
  --L2 is x1 in descending order
  L2 <- reverse L1
  -- Take L2's first x2 entries
  L3 <- take L2 x2
  -- Compute sum of all elements in L3
  sum L3
```

We now explain our key ideas using this simple example and the small DSL shown in Figure 2. In this section (and throughout the paper), we represent programs using their abstract syntax tree (AST) representation. For instance, the

$$
\begin{aligned}
N &\rightarrow \texttt{0} \mid \ldots \mid \texttt{10} \mid x_i \mid \texttt{last}(L) \mid \texttt{head}(L) \mid \texttt{sum}(L) \\
&\quad \mid \texttt{maximum}(L) \mid \texttt{minimum}(L) \\
L &\rightarrow \texttt{take}(L, N) \mid \texttt{filter}(L, T) \mid \texttt{sort}(L) \mid \texttt{reverse}(L) \mid x_i \\
T &\rightarrow \texttt{geqz} \mid \texttt{leqz} \mid \texttt{eqz}
\end{aligned}
$$

Figure 2. The grammar of a simple DSL for manipulating lists of integers; in this grammar, N is the start symbol.

Component	Specification
`head`	$x_1.size > 1 \wedge y.size = 1 \wedge y.max \leq x_1.max$
`take`	$y.size < x_1.size \wedge y.max \leq x_1.max \wedge$ $x_2 > 0 \wedge x_1.size > x_2$
`filter`	$y.size < x_1.size \wedge y.max \leq x_1.max$

Table 1. Examples of component specifications. Here, y denotes the output, and x_i denotes the i'th input.

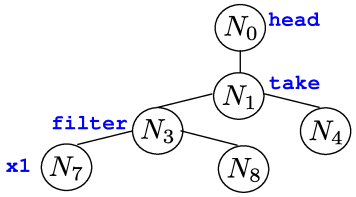


Figure 3. An example partial program

AST shown in Figure 3 corresponds to the *partial* program `head(take(filter(x1,?),?))`, where each question mark is a *hole* (i.e., unknown program) yet to be determined. We think of partial programs as assignments from each AST node to a specific component. For instance, the AST from Figure 3 corresponds to the following *partial assignment*:

$$\{N_0 \mapsto \texttt{head},\ N_1 \mapsto \texttt{take}, N_3 \mapsto \texttt{filter}, N_7 \mapsto \texttt{x1}\}$$

Decide. Given an AST representing a partial program P (initially, a single unassigned root node), our synthesis algorithm determines how to fill one of the holes in P. In other words, thinking of partial programs as partial assignments from AST nodes to DSL constructs, the goal of the *Decide* component is to choose an unassigned node N in the AST and determine which DSL construct to assign to N.

Our technique requires the assignments made by the *Decide* component to obey any lemmas that have been added to the knowledge base Ω. In particular, Ω consists of a set of propositional formulas over variables $c_{i,\chi}$ whose truth value indicates whether AST node with unique identifier i is assigned to component χ. Thus, making an assignment consistent with the knowledge base requires checking the satisfiability of a propositional formula. However, since there are typically many different decisions that are consistent with the knowledge base, the *Decide* component additionally consults a statistical model to predict the most "promising" assignment.

Deduce. After every assignment made by the *Decide* component, Neo performs deduction to check whether the current partial program is feasible. [1] Our deduction engine utilizes the semantics of the DSL, provided as first-order specifications of each component. For instance, Table 1 shows the specifications for three DSL constructs, namely `take`, `head` and `filter`. Here, the specification for `take` says that the maximum element and size of the output list are no larger than those of the input list. The specification for `head` is similar and states that the maximum element of the output is no larger than that of the input. Finally, the specification for `filter` says that the size (resp. maximum) of the output is smaller (resp. not larger) than the size of the input list.

$$
\begin{aligned}
\Phi_P &= \phi_{N_0} \wedge \phi_{N_1} \wedge \phi_{N_3} \wedge \phi_{N_7} \\
\phi_{N_0} &= \underline{y \leq v_1.max} \ \wedge\ v_1.size > 1 \ \wedge\ y.size = 1 \\
\phi_{N_1} &= \underline{v_1.max \leq v_3.max} \ \wedge\ v_1.size < v_3.size \wedge \\
&\quad\ v_4 > 0 \ \wedge\ v_3.size > v_4 \\
\phi_{N_3} &= v_3.size < v_7.size \ \wedge\ \underline{v_3.max \leq v_7.max} \\
\phi_{N_7} &= \underline{x_1 = v_7}
\end{aligned}
$$

Figure 4. Specification of partial program from Figure 3, where v_i represents the intermediate value at node N_i and variables x_1 and y denote the input and output, respectively.

Neo uses the specifications of each component to infer a specification of the current partial program. In particular, Figure 4 shows the inferred specification Φ_P for partial program P from Figure 3. Observe that Φ_P uses the specifications of `head`, `take`, and `filter` to infer a specification for each node in the AST. Our method determines the feasibility of partial program P by checking the satisfiability of the SMT formula $\Phi_P \wedge \Phi$ where Φ represents the user-provided specification. In this case, the input-output example corresponds to the specification $x_1 = [49, 62, 82, 54, 76] \wedge y = 158$. Observe that the formula $\Phi_P \wedge \Phi$ is unsatisfiable: Since $x_1.max = 82$, Φ_P implies $y \leq 82$, contradicting the fact that $y = 158$.

Analyzing conflicts. The key novelty of our technique is its ability to analyze conflicts and learn useful lemmas that prevent similar bad decisions in the future. In particular, Neo learns new lemmas by identifying components that are *equivalent modulo conflict*. That is, given an infeasible partial program P containing component χ, our method identifies other components $\chi_1, \ldots, \chi_n$ such that replacing χ with any χ_i results in another infeasible program. By computing these equivalence classes, we can generalize from the current conflict and learn many other partial programs that are guaranteed to be infeasible. This information is then encoded as a SAT formula and added to the knowledge base to avoid similar conflicts in the future.

We now illustrate how Neo learns new lemmas after it detects the infeasibility of partial program P from Figure 3. To identify the *root cause* of the conflict, Neo starts by computing a *minimal unsatisfiable core (MUC)* of the formula

[1] Note that SMT-based deduction is not a contribution of this paper; however, it is a prerequisite for learning from conflicts.

$\Phi_P \wedge \Phi$. For this example, the MUC includes all underlined predicates in Figure 4. We represent the MUC as a set of triples (φ_i, N_i, χ_i) where χ_i is a component labeling node N_i and each formula φ_i corresponds to a part of χ_i's specification. For our running example, the MUC corresponds to the following set κ:

$$\left\{ \begin{array}{c} (y \leq x_1.max, N_0, \texttt{head}),\ (y.max \leq x_1.max, N_1, \texttt{take}) \\ (y.max \leq x_1.max, N_3, \texttt{filter}),\ (y = x_1, N_7, x_1) \end{array} \right\}$$

Our learning algorithm infers components that are equivalent modulo conflict by analyzing the MUC. In particular, let (φ, N, χ) be an element of the MUC, and let χ' be another component with specification γ. Now, if γ logically implies φ, we can be sure that replacing the annotation of node N with χ' in partial program P will result in an infeasible program with the same MUC as P. Thus, we can conclude that χ and χ' are equivalent modulo conflict at node N.

Going back to our example, the partial program from Figure 3 contains the `take` component, and the relevant part of its specification that appears in the MUC is the predicate $y.max \leq x_1.max$. Since the specification of `sort` (see Table 1) logically implies $y.max \leq x_1.max$, we can conclude that changing the annotation of node N_1 to `sort` will still result in a spurious program. Using this strategy, we can learn that the following components all belong to the same equivalence class with respect to the current conflict:

$$\texttt{take} \equiv_{N_1} \texttt{reverse} \equiv_{N_1} \texttt{sort} \equiv_{N_1} \texttt{filter}$$

In other words, changing the assignment of node N_1 to `sort`, `reverse`, or `filter` is guaranteed to result in another infeasible program. Using the same kind of reasoning for other nodes, we can learn a lemma (whose form is described in Section 5) that allows us to rule out 63 other partial programs that would have otherwise been explored by the synthesis algorithm. Thus, learning from conflicts allows us to prune large parts of the search space.

3 Preliminaries

Before describing our algorithm in detail, we first provide some background that will be used throughout the paper.

3.1 Domain-Specific Language & Semantics

Our synthesis algorithm searches the space of programs described by a given domain-specific language (DSL), which consists of a context-free grammar $\mathcal{G}$ together with the semantics of DSL operators (i.e., "components").

Syntax. The syntax of the DSL is described by a context-free grammar $\mathcal{G}$. In particular, $\mathcal{G}$ is a tuple (V, Σ, R, S), where V is the set of nonterminals, Σ represents terminals, R is the set of productions, and S is the start symbol. The terminals $\chi \in \Sigma$ correspond to built-in DSL operators (e.g., `+`, `concat`, `map` etc), constants, and variables. We assume that $\mathcal{G}$ always includes special terminal symbols $x_1, ..., x_k \in \Sigma$ denoting the k program inputs. The productions $p \in R$ have the form $p = (A \rightarrow \chi(A_1 ... A_k))$, where $\chi \in \Sigma$ is a DSL operator and $A, A_1, ..., A_k \in V$ are nonterminals. We use the notation Σ_k to denote DSL operators of arity k, and we write $\Sigma_{A, A_1 ... A_k}$ to denote DSL operators χ such that R contains a production $A \rightarrow \chi(A_1 ... A_k)$.

Semantics. As mentioned in Section 1, our synthesis algorithm uses the semantics of DSL constructs to make useful deductions and analyze conflicts. We assume that the DSL semantics are provided as a mapping Ψ from each DSL operator $\chi \in \Sigma_n$ to a first-order formula over variables $y, x_1, \ldots, x_n$ where x_i represents the i'th argument of χ and y represents its return value. For instance, consider a unary function `inc` that returns its argument incremented by 1. Then, we have $\Psi(\texttt{inc}) = (y = x_1 + 1)$.

3.2 Partial Programs

Since the key data structure maintained by our synthesis algorithm is a *partial program*, we now introduce some terminology related to this concept.

Definition 3.1. (Partial program) Given a DSL defined by context-free grammar $\mathcal{G} = (V, \Sigma, R, S)$, a *partial program* P in this DSL is a string $P \in (\Sigma \cup V)^*$ such that $S \overset{*}{\Rightarrow} P$.

In contrast to a *concrete program* which only contains symbols from Σ, a partial program may contain non-terminals. We say that concrete program P' (or just "program" for short) is a completion of P if $P \overset{*}{\Rightarrow} P'$.

We represent partial programs as *abstract syntax trees* (AST). Given partial program P, we use the notation $\mathsf{Nodes}(P)$, $\mathsf{Internal}(P)$, and $\mathsf{Leaves}(P)$ to denote the set of all nodes, internal nodes, and leaves in P, respectively. We also write $\mathsf{Children}(N)$ to denote the children of internal node N.

In our representation of partial programs, every node N is labeled with a corresponding grammar symbol $A_N \in V$ such that N may be expanded using any production whose left-hand side is A_N. Every node N is also optionally labeled with a symbol $\chi_N \in \Sigma$ indicating that N has been expanded using the production $A_N \rightarrow \chi_N(\ldots)$. Observe that internal nodes in the AST *must* have these χ_N annotations, but leaf nodes do not. In particular, any leaf node N without a χ_N annotation represents an unknown program fragment; thus, we refer to such nodes as *holes*. Given partial program P, we write $\mathsf{Holes}(P)$ to represent the set of all holes in P.

Example 3.2. Consider the partial program from Figure 3. Here, we have the following annotations for each node:

$$\begin{array}{lll} A_{N_0} = N & A_{N_1} = L & A_{N_3} = L \\ A_{N_4} = N & A_{N_7} = L & A_{N_8} = T \\ \chi_{N_0} = \texttt{head} & \chi_{N_1} = \texttt{take} & \chi_{N_3} = \texttt{filter} \\ \chi_{N_7} = \texttt{x1} & & \end{array}$$

Observe that leaf nodes N_4 and N_8 correspond to holes in this partial program.

Algorithm 1 Given DSL with syntax $\mathcal{G}$ and semantics Ψ as well as a specification Φ, Synthesize either returns a DSL program P such that $P \models \Phi$ or $\perp$ if no such program exists.

```
1:  procedure Synthesize(G, Ψ, Φ)
2:      P ← Root(S)
3:      Ω ← ∅
4:      while true do
5:          (H, p) ← Decide(P, G, Φ, Ω)
6:          P ← Propagate(P, G, (H, p), Ω)
7:          κ ← CheckConflict(P, Ψ, Φ)
8:          if κ ≠ ∅ then
9:              Ω ← Ω ∪ AnalyzeConflict(P, G, Ψ, κ)
10:             P ← Backtrack(P, Ω)
11:         if UNSAT(⋀_{φ∈Ω} φ) then
12:             return ⊥
13:         else if IsConcrete(P) then
14:             return P
```

In the remainder of this paper, we assume that there is a unique index s_N associated with each node N in the AST. In particular, if N is the i'th node at depth d of the AST, we assign N the id $k^{d-1} + i - 1$, where k is the maximum arity of any DSL operator. [2] Observe that this numbering scheme ensures that nodes at the same position in two different ASTs are always assigned the same identifier. As we will see in the next section, this assumption is important for learning reusable lemmas from conflicts.

4 Synthesis Algorithm

In this section, we describe the architecture of our conflict-driven synthesis algorithm and explain each of its components in detail. However, because conflict analysis is one of the main contributions of this paper, we defer a detailed discussion of *AnalyzeConflict* to Section 5.

4.1 Overview

Algorithm 1 shows the high-level structure of our synthesis algorithm, which takes as input a specification Φ that must be satisfied by the synthesized program as well as a domain-specific language with syntax $\mathcal{G}$ and semantics Ψ. We assume that specification Φ is an SMT formula over variables $\vec{x}, y$, which represent the inputs and output of the program respectively. The output of the Synthesize procedure is either a concrete program P in the DSL or $\perp$, meaning that there is no DSL program that satisfies Φ. As we will prove later, our synthesis algorithm is both *sound* and *complete* with respect to the provided DSL semantics. In particular, the program P returned by Synthesize is guaranteed to satisfy Φ with respect to Ψ, and Synthesize returns $\perp$ only if there is indeed no DSL program that satisfies Φ.

[2] We assume that the root node is at depth 1.

Internally, our synthesis algorithm maintains two data structures, namely a partial program P and a *knowledge base* Ω. The knowledge base Ω is a set of *learnt lemmas* derived from the input specification Φ with respect to Ψ, where each lemma is represented as a propositional (SAT) formula. The Synthesize procedure initializes P to contain a single root node labeled with the start symbol S (line 3); thus, P initially represents any syntactically legal DSL program. The knowledge base Ω is initialized to the empty set (line 4), but will be updated by the algorithm as it learns new lemmas from each conflict.

The key part of the synthesis procedure is the conflict-driven learning loop in lines 4–14. Given a partial program P containing holes, the Decide procedure selects a hole H in P as well as a candidate production p with which to fill H. The decision (H, p) returned by Decide should be consistent with the knowledge base in order to prevent the algorithm from making wrong choices as early as possible. In other words, filling hole H according to production p should yield a partial program P' that does not violate the lemmas in Ω. The Decide procedure is further described in Section 4.3.

After choosing a hole H to be filled using production p, the synthesis algorithm performs two kinds of deduction, represented by the calls to Propagate and CheckConflict in lines 6 and 7 respectively. In particular, Propagate is analogous to *Boolean Constraint Propagation (BCP)* in SAT solvers [3] and infers new assignments that are implied by the knowledge base as a result of filling hole H with production p. In contrast, the CheckConflict procedure uses the semantics of the DSL constructs to determine if there exists a completion of P that can satisfy Φ. If P cannot be completed in a way that satisfies Φ, we have detected a *conflict* (i.e., P is *spurious*), and CheckConflict returns the *root cause* of the conflict. As explained in Section 2, we represent the root cause of each conflict as a *minimal unsatisfiable core (MUC)* κ of the SMT formula representing the specification of P. If $\kappa = \emptyset$, this means that CheckConflict did not find any conflicts, so the algorithm goes back to making new decisions if there are any remaining holes in P. The Propagate and CheckConflict procedures are further described in Sections 4.4 and 4.5, respectively.

As mentioned earlier, the key innovation underlying our synthesis algorithm is its ability to make generalizations from conflicts. Given a non-empty MUC returned by CheckConflict, the AnalyzeConflict procedure (line 9) analyzes the unsatisfiable core κ to identify other spurious partial programs that have the same root cause of failure as P. Thus, the lemmas returned by AnalyzeConflict prevent the Decide component from generating partial programs that will *eventually* result in a similar conflict as P. The algorithm adds these new lemmas to the knowledge base and *backtracks* by

[3] Recall that BCP in SAT solvers exhaustively applies unit propagation by finding all literals that are implied by the current assignment.

undoing the assignments made by Decide and Propagate during the last iteration.

Algorithm 1 has two possible termination conditions that are checked after each iteration: If the conjunction of lemmas in the knowledge base Ω has become unsatisfiable, this means that there is no DSL program that can satisfy Φ; thus, the algorithm returns $\perp$. On the other hand, if P is a concrete program without holes, it must satisfy Φ with respect to the provided semantics Ψ; thus, the algorithm returns P as a possible solution.

Discussion. The soundness of the Synthesize procedure from Algorithm 1 is with respect to the provided semantics Ψ of the DSL. Thus, if Ψ defines a complete semantics of each DSL construct, then the synthesized program is indeed guaranteed to satisfy Φ. However, if Ψ *over-approximates* (i.e., *under-specifies*) the true semantics of the DSL, then the synthesized program is not guaranteed to satisfy Φ. For programming-by-example applications where the specification Φ represents concrete input-output examples, we believe that a sensible design choice is to use *over-approximate* specifications of the DSL constructs and then check whether P actually satisfies Φ by executing P on these examples. The benefit of *over-approximate* specifications is two-fold: First, for some operators, it may be infeasible to precisely encode their functionality using a first-order theory supported by SMT solvers. Second, the use of over-approximate specifications allows us to control the tradeoff between effectiveness of deduction/learning and overhead of SMT solving.

4.2 Knowledge Base and SAT Encoding of Programs

Before we can explain each of the subroutines used in Algorithm 1, we first describe the *knowledge base* Ω maintained by the synthesis algorithm. As mentioned earlier, Ω is a set of learnt lemmas, where each lemma ϕ is a SAT formula over *encoding variables* $c_{s_N,p}$. Here, s_N corresponds to the *unique* index associated with an AST node N and p is a production in the grammar. Thus, the encoding variable $c_{s_N,p}$ indicates whether N is labeled with (i.e., assigned to) production p.

In order to ensure that the choices made by the algorithm are consistent with the knowledge base, it is convenient to represent partial programs in terms of a SAT formula over encoding variables. Towards this goal, we introduce the following SAT encoding of partial programs:

Definition 4.1. (SAT encoding of program) Let P be the AST representation of a partial program, as explained in Section 3.2. We use the notation π_P to denote the following SAT encoding of P:

$$\pi_P = \bigwedge_{N \in \text{Nodes}(P)} c_{s_N, \chi_N}$$

where s_N denotes the unique index of node N and χ_N is the component labeling node N.

Algorithm 2 Outline of Decide

```
procedure Decide(P, 𝒢, Φ, Ω)
    V ← {(H, p) | H ∈ Holes(P), p = A_H → χ(A_1...A_k)}
    V' ← {(H, p) ∈ V | Fill(P, H, p) ~ Ω}
    return arg max_{(H,p)∈V'} L_θ(Fill(P, H, p) | Φ)
```

Example 4.2. The partial program in Figure 3 can be denoted using the following SAT encoding π_P:

$$c_{0,\texttt{head}} \wedge c_{1,\texttt{take}} \wedge c_{3,\texttt{filter}} \wedge c_{7,\texttt{x1}}.$$

Definition 4.3. (Consistency with KB) We say that a partial program P is *consistent* with knowledge base Ω, denoted $P \sim \Omega$, if the formula $\pi_P \wedge \bigwedge_{\phi \in \Omega} \phi$ is satisfiable.

Definition 4.4. (Consistency with spec) We say that a partial program P is *consistent with* specification Φ, denoted $P \sim \Phi$, if there exists some completion of P that satisfies Φ.

Definition 4.5. (Correctness of KB) The knowledge base Ω is *correct* with respect to specification Φ if, for any partial program P, $P \sim \Phi$ implies $P \sim \Omega$.

Thus, given a correct knowledge base Ω, the synthesis algorithm can safely prune any partial program P that is inconsistent with Ω. In particular, if P is inconsistent with Ω, the correctness of the knowledge base guarantees that there is no completion of P that satisfies specification Φ.

4.3 The Decide Subroutine

We will now explain each of the auxiliary procedures used in Algorithm 1, starting with the Decide component. The high-level idea underlying our Decide procedure is to fill one of the holes in P such that (a) the resulting partial program P' is consistent with the knowledge base, and (b) P' is the most likely completion of P with respect to a probabilistic model. Thus, our Decide procedure combines logical constraints with statistical information in a unified framework.

Algorithm 2 shows the high-level structure of our Decide component and makes use of a procedure $\text{Fill}(P, H, p)$ which fills hole H in partial program P with a production $p = (A_H \rightarrow \chi(A_1 ... A_k))$. Specifically, Fill generates a new partial program P' that is the same as P except that node H is now labeled with DSL operator χ and has k new children labeled $A_1, \ldots, A_k$. Thus, if $P' = \text{Fill}(P, H, p)$ and we think of P and P' as strings in $(\Sigma \cup V)^*$, then we have $P \Rightarrow P'$.

Algorithm 2 proceeds in three steps: First, it constructs the set V of all pairs (H, p), where H is a hole in the partial program P, and $p = (A_H \rightarrow \chi(A_1 ... A_k))$ is a grammar production that can be used to fill H. Second, it restricts this set to pairs (H, p) such that the program $P' = \text{Fill}(P, H, p)$ satisfies the knowledge base Ω. Finally, it assumes access to a probabilistic model L_θ (parametrized by $\theta \in \mathbb{R}^d$) that can be used to score partial programs conditioned on the

Algorithm 3 Outline of Propagate

```
procedure Propagate(P, G, (H, p), Ω)
    P ← Fill(P, H, p)
    S ← Holes(P) × Productions(G)
    S' ← {(N, p) | (N, p) ∈ S ∧ p = (A_N → χ(...))}
    for all (H_i, p_i) ∈ S' do
        R ← {p | p ∈ Productions(G) ∧ p = (A_{H_i} → ...)}
        if (⋀_{φ∈Ω} φ ∧ ⋁_{p_j∈R} c_{s_{H_i},p_j} ∧ π_P) ⇒ c_{s_{H_i},p_i} then
            P ← Propagate(P, G, (H_i, p_i), Ω)
    return P
```

specification Φ, i.e.,

$$L_\theta(P' \mid \Phi, P) = \Pr[P' \mid \Phi, P, \theta].$$

Based on this model, Decide returns the pair (H, p) resulting in the most likely partial program $P' = \mathsf{Fill}(P, H, p)$ according to this model. [4]

4.4 The Propagate Subroutine

After each invocation of Decide, the synthesis algorithm infers additional assignments that are implied by the current decision. Such inferences are made by the Propagate procedure summarized in Algorithm 3.

Given a decision (H, p), Propagate first fills hole H with production p; it then checks whether this decision implies additional assignments. It does so by querying whether the following implication is valid:

$$\Big(\bigwedge_{\phi\in\Omega} \phi \wedge \bigvee_{p_j\in R} c_{s_{H_i},p_j} \wedge \pi_P\Big) \Rightarrow c_{s_{H_i},p_i}$$

where R is the set of all productions that can be used to fill hole H_i. Intuitively, we check if Ω and π_P imply that the only feasible choice for hole H_i is to fill it using production p_i. If this is the case, the Propagate procedure recursively calls itself to further propagate this assignment. [5]

Remark. The Propagate procedure is a necessary ingredient of our algorithm rather than a mere optimization because it enforces the consistency of the current assignment with the constraints stored in Ω. In particular, if Propagate was not invoked after each decision, then the Decide procedure could continously choose the same decision (H, p).

Example 4.6. Suppose that the current partial program consists of a single hole (i.e., AST with only node N_0) and the knowledge base Ω contains the following two lemmas:

$$\Big\{\neg c_{0,\texttt{filter}} \vee \neg c_{2,\texttt{eqz}},\ \ \neg c_{0,\texttt{filter}} \vee \neg c_{2,\texttt{leqz}}\Big\}$$

[4] We do not fix a particular statistical model because different models may be suitable for different applications. Section 6 describes two different statistical models used in our implementation.

[5] In general, Propagate can discover conflicts if the decision (H, p) will lead to a hole H_i that cannot be filled with any production p_j. In this case, the algorithm will backtrack and pick another decision. For simplicity, we omit this case from the Propagate subroutine and the main synthesis algorithm.

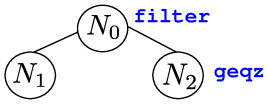
Algorithm 4 Outline of CheckConflict

```
1: procedure CheckConflict(P, Ψ, Φ)
2:     Φ_P ← InferSpec(P)
3:     ψ ← SMTSolve(Φ_P ∧ Φ)
4:     κ ← {(φ, N, χ_N) | φ ∈ ψ ∧ N = Node(φ)}
5:     κ' ← {(φ', N, χ_N) | φ' = Rename(φ) ∧ (φ, N, χ_N) ∈ κ}
6:     return κ'
```

$$\begin{aligned}
\Phi_P &= \Phi_R[y/v_{s_R}] \ \text{ where } R = \mathsf{Root}(P)\\
\Phi_N &= \Psi_N \wedge \textstyle\bigwedge_{N_i\in\mathsf{Children}(N)} \Phi_{N_i}\\
\Psi_N &= \begin{cases} \mathit{true} & \text{if } N \in \mathsf{Holes}(P)\\ \Psi(\chi_N)[v_{s_N}/y] & \text{else if } N \in \mathsf{Leaves}(P)\\ \Psi(\chi_N)[C(N)/\vec{x}, v_{s_N}/y] & \text{otherwise}\end{cases}\\
C(N) &= [v_{s_{N_1}}, \ldots, v_{s_{N_k}}] \text{ where } [N_1, \ldots, N_k] = \mathsf{Children}(N)
\end{aligned}$$

Figure 5. Rules defining InferSpec(P)

If *Decide* makes the assignment $N_0 \mapsto$ `filter`, *Propagate* will result in the following partial program:



In particular, observe that $\mathsf{Fill}(P, N_0, \texttt{filter})$ results in two new nodes N_1, N_2 where $A_{N_1} = L$ and $A_{N_2} = T$. Furthermore, since the knowledge base Ω and the assignment $N_0 \mapsto$ `filter` together imply that N_2 cannot be assigned to `leqz` or `eqz`, *Propagate* infers that N_2 must be assigned to `geqz`.

4.5 The CheckConflict Subroutine

In addition to identifying assignments implied by the current decision, our synthesis procedure performs a different form of deduction to prune partial programs that do not satisfy the specification. This form of deduction is performed by the CheckConflict procedure outlined in Algorithm 4.

The core part of CheckConflict is the InferSpec procedure, described as inference rules in Figure 5. Given a partial program P, InferSpec generates an SMT formula Φ_P that serves as a specification of P. This formula Φ_P is constructed recursively by traversing the AST bottom-up and uses a variable v_{s_N} to denote the return value of the sub-program rooted at node N. The specification Φ_N of node N is obtained by conjoining the specifications of the children of N with the (suitable renamed) specification Ψ_N of the component labeling N. Observe that the final SMT formula Φ_P is over variables $\vec{x}, y$ describing P's inputs and outputs respectively as well as auxiliary variables $\vec{v}$ denoting intermediate values of P's sub-expressions.

Example 4.7. Figure 4 shows the result of calling InferSpec on the partial program from Figure 3.

The following theorem states the correctness of the InferSpec procedure presented in Figure 5:

Theorem 4.8. Assuming Ψ provides a sound semantics of the DSL and $P \sim \Phi$, then $\Phi_P \wedge \Phi$ is satisfiable.

According to this theorem, if $\Phi_P \wedge \Phi$ is unsatisfiable, then there is in fact no completion of P that can satisfy Φ, meaning that P is infeasible. Thus, CheckConflict invokes the SMT-Solve procedure to check the satisfiability of $\Phi_P \wedge \Phi$ (line 3 of Algorithm 4). If this formula is unsatisfiable, Theorem 4.8 allows us to prune partial program P from the search space.

Since our learning algorithm makes use of a *minimal unsatisfiable core (MUC)*, we represent the return value of the SMTSolve procedure as a set of clauses ψ representing the MUC. [6] In particular, the MUC ψ is a set of SMT formulas $\{\phi_1, \ldots, \phi_n\}$ such that:

1. $\bigwedge_i \phi_i \models \mathit{false}$,
2. Each ϕ_i either corresponds to a clause (conjunct) of Φ or a clause of $\Psi(\chi)$ for some component χ (modulo renaming)
3. ψ is minimal, i.e., for any $\phi_i \in \psi$, $\bigwedge_{\phi_j \in \psi \setminus \{\phi_i\}} \phi_j \not\models \mathit{false}$

Because our AnalyzeConflict procedure requires the MUC to be represented in a special form, lines 4–6 of CheckConflict post-process ψ to generate an MUC consisting of triples (ϕ_i, N, χ_i) where χ_i is a component labeling node N and ϕ_i is a clause in χ_i's specification. In particular, line 4 identifies, for each $\phi_i \in \psi$, the AST node $N = \text{Node}(\phi_i)$ that is associated with ϕ_i and attaches to ϕ_i the component χ_N labeling N. Finally, since each $\phi_i \in \psi$ refers to auxiliary variables associated with nodes in the AST, lines 5 converts each ϕ_i to a "normal form" over variables $\vec{x}, y$ rather than variables $\vec{v}$ used in Φ_P. Observe that clauses of the MUC that come from Φ are dropped during this post-processing step.

Example 4.9. Consider the formula $\Phi_P \wedge x_1.max = 82 \wedge y = 158$ where Φ_P is the formula from Figure 4. This formula is unsatisfiable and its MUC ψ consists of the following clauses:

$$\left\{ \begin{array}{c} x_1.max = 82,\ y = 158,\ y \leq v_1.max, \\ v_1.max \leq v_3.max, v_3.max \leq v_7.max,\ x_1 = v_7 \end{array} \right\}$$

Since the first two clauses come from the specification Φ, they are dropped during post-processing. The clause $y \leq v_1.max$ is generated from the root node N_0 from Figure 3; thus, the set representation κ of the MUC contains $(y \leq x_1.max, N_0, \texttt{head})$. The clause $v_1.max \leq v_3.max$ is generated from node N_1 with annotation `take`; thus, κ also contains $(y.max \leq x_1.max, N_1, \texttt{take})$. Using similar reasoning for the other clauses in ψ, we obtain the following set representation κ of the MUC:

$$\left\{ \begin{array}{c} (y \leq x_1.max, N_0, \texttt{head}),\ (y.max \leq x_1.max, N_1, \texttt{take}) \\ (y.max \leq x_1.max, N_3, \texttt{filter}),\ (y = x_1, N_7, x_1) \end{array} \right\}$$

[6] If $\Phi_P \wedge \Phi$ is satisfiable, then ψ is simply the empty set.

5 Analyzing Conflicts

In this section, we turn our attention to the conflict analysis procedure for learning new lemmas to add to the knowledge base. To the best of our knowledge, our approach is the first synthesis technique that can rule out unrelated partial programs by analyzing the root cause of the conflict.

The key idea underlying our learning algorithm is to identify DSL operators that are *equivalent modulo conflict*:

Definition 5.1. Let P be a partial program that is inconsistent with specification Φ, and let χ be a DSL operator labeling node N of P. We say that components χ, χ' are *equivalent modulo conflict* at node N, denoted $\chi \equiv_N \chi'$ if replacing label χ of node N with χ' results in a program P' that is also inconsistent with Φ.[7]

To see why the notion of equivalence modulo conflict (EMC) is useful for synthesis, let P be a spurious partial program containing n assigned nodes, and suppose that, for each node N, we have m different components that are equivalent modulo conflict to χ_N. Using this information, we can learn m^n other partial programs that are all infeasible with respect to specification Φ. By encoding these partial programs as lemmas in our knowledge base, we can potentially prune a large number of programs from the search space.

As illustrated by this discussion, we would like to find *as many components as possible* that are equivalent to each component used in P. Specifically, the bigger the size of each equivalence class (i.e., m), the more programs we can prune.

The most straightforward way to identify components that are equivalent modulo conflict is to check whether their specifications are logically equivalent. While this approach would clearly be sound, it would not work well in practice because different DSL constructs rarely share the same specification. Thus, the size of each equivalence class would be very small, meaning that the synthesizer cannot rule out many programs as a result of a conflict.

The core idea underlying our learning algorithm is to infer equivalence classes by analyzing the root cause of the infeasibility of a given partial program P. In particular, the idea is to extract the root cause of P's infeasibility by obtaining a minimal unsatisfiable core of the formula $\Phi_P \wedge \Phi$, where Φ_P represents the specification of P. Now, because each clause in the MUC refers to a small subset of the clauses in component specifications, we can identify maximal equivalence classes by utilizing precisely those clauses that appear in the MUC. The following theorem makes this discussion more precise.

Theorem 5.2. *Let P a partial program inconsistent with specification Φ, and let κ be the MUC returned by Algorithm 4. We have $\chi \equiv_N \chi'$ if $\Psi(\chi') \Rightarrow \phi$, where $(\phi, N, \chi) \in \kappa$.*

Intuitively, if $\Psi(\chi')$ logically implies ϕ, then the specification $\Psi(\chi')$ for χ' is more restrictive than the specification

[7] Note that equivalence modulo conflict also depends on the partial program P; we omit this information to simplify our notation.

Algorithm 5 Algorithm for learning lemmas

```
1: procedure AnalyzeConflict(P, 𝒢, Ψ, κ)
2:     φ ← false
3:     for (ϕ, N, χ_N) ∈ κ do
4:         (A_1, ..., A_k) ← (A_{N_i} | N_i ∈ Children(N))
5:         Σ_N ← {χ | χ ∈ Σ_{A_N,A_1,...,A_k} ∧ Ψ(χ) ⇒ ϕ}
6:         φ ← φ ∨ ⋀_{χ∈Σ_N} ¬c_{s_N,χ}
7:     return φ
```

$\Psi(\chi)$ for χ, considering only the subformula ϕ of $\Psi(\chi)$ contained in the MUC. Thus, changing the annotation of node N from χ to χ' in P is guaranteed to result in an another infeasible partial program, meaning that χ and χ' are equivalent modulo conflict at node N.

Example 5.3. Consider the element $(y.max \leq x_1.max, N_1, \texttt{take})$ in the MUC from Example 4.9. Also, recall from Table 1 that the specification of `sort` is $y.size = x_1.size \wedge y.max = x_1.max$. Since the formula

$$(y.size = x_1.size \wedge y.max = x_1.max) \Rightarrow y.max \leq x_1.max$$

is logically valid, we have $\texttt{sort} \equiv_{N_1} \texttt{take}$.

We now discuss how the AnalyzeConflict procedure from Algorithm 5 leverages Theorem 5.2 to learn new lemmas to add to the knowledge base. As shown in Algorithm 5, the AnalyzeConflict procedure takes as input the partial program P, the syntax $\mathcal{G}$ and semantics Ψ of the DSL, as well as the MUC κ representing the root cause of infeasibility of P. The output of the algorithm is a lemma φ that can be added to the knowledge base.

The AnalyzeConflict procedure iterates over all elements (ϕ, N, χ_N) in the MUC κ and uses Theorem 5.2 to compute a set Σ_N such that Σ_N contains all components χ' that are equivalent to χ modulo conflict at node N. It then generates the following lemma to add to the knowledge base:

$$\bigvee_{N \in \kappa} \bigwedge_{\chi \in \Sigma_N} \neg c_{s_N, \chi}$$

Here, the outer disjunct states that we must change the assignment for at least one of the nodes N that appear in the proof of infeasibility of the current partial program P. The inner conjunct says that node N cannot be assigned to any of the χ's that appear in Σ_N because Theorem 5.2 guarantees that changing the assignment of N to χ must result in another infeasible program.

The following theorem states the correctness of the lemmas returned by AnalyzeConflict:

Theorem 5.4. *Let φ be a lemma returned by AnalyzeConflict. If $P \sim \Phi$, then the formula $\pi_P \wedge \varphi$ is satisfiable.*

Since π_P is the SAT encoding of P, this theorem says that the learnt lemma φ must be consistent with π_P for it to be the case that $P \sim \Phi$. Thus, we have:

Corollary 1. *The knowledge base Ω maintained by Algorithm 1 is correct with respect to specification Φ.*

Finally, the soundness and completeness of our algorithm follow from Theorem 4.8 and Corollary 1:

Theorem 5.5. (Soundness) *If Algorithm 1 returns P as a solution to the synthesis problem defined by $\mathcal{G}, \Psi, \Phi$, then P satisfies specification Φ with respect to DSL semantics Ψ.*

Theorem 5.6. (Completeness) *If Algorithm 1 returns $\perp$ as a solution to the synthesis problem defined by $\mathcal{G}, \Psi, \Phi$, then there is no DSL program that satisfies Φ with respect to DSL semantics Ψ.*

6 Implementation

We have implemented our conflict-driven synthesis framework in a tool called Neo, written in Java. Neo uses the SAT4J [4] SAT solver to implement Decide and Propagate and employs the Z3 [6] SMT solver to check for conflicts.

Decide. As explained in Section 4.3, Neo uses a combination of logical and statistical reasoning to identify which hole to fill and how to fill it. However, our implementation of Decide differs from Algorithm 2 in that we do not issue a full SAT query to determine whether the decision is consistent with the knowledge base. Since checking satisfiability for each combination of holes and components is potentially very expensive, our implementation of Decide over-approximates satisfiability through unit propagation. [8] In particular, we consider an assignment to be feasible if applying unit propagation to the corresponding SAT formula does not result in a contradiction. Note that replacing a full SAT query with unit propagation does not affect the soundness or completeness of our approach. In particular, the algorithm may end up detecting conflicts later than if it were using a full SAT query, but it also reduces overhead without affecting any soundness and completeness guarantees.

Since there are many possible assignments that do not contradict the knowledge base, Neo uses a statistical model to identify the "most promising" one. Our current implementation supports two different statistical models, namely a 2-gram model (as used in Morpheus [8]) as well as a deep neural network model (as described in DeepCoder [3]). While the 2-gram model only considers the current partial program to make predictions, the deep neural network model considers both the specification and the current partial program.

Propagate. As described in Section 4.4, the goal of propagation is to identify additional assignments implied by the knowledge base. We identify such assignments by performing unit propagation on the corresponding SAT formula.

[8] Unit propagation (also known as Boolean constraint propagation) applies unit resolution to a fixed point. In particular, unit resolution derives the clause $\{x_1, \ldots, x_n\}$ from the unit clause $\{l\}$ and another clause $\{\neg l, x_1, \ldots, x_n\}$.

CheckConflict. Our implementation of CheckConflict follows Algorithm 4 and uses Z3 to query the satisfiability of the corresponding SMT formula [6]. Given an unsatisfiable formula ϕ, we also use Z3 to obtain an unsatisfiable core and post-process it as described in Section 4.5. The unsatisfiable core returned by Z3 is not guaranteed to be minimal. We do not minimize the unsatisfiable core since this procedure can be time consuming, but in practice we have observed that the unsatisfiable cores returned by Z3 are often minimal.

Our implementation of CheckConflict performs an additional optimization over Algorithm 4: Since different partial programs may share the same SMT specification, Algorithm 4 ends up querying the satisfiability of the same SMT formula multiple times. Thus, our implementation memoizes the result of each SMT call to avoid redundant Z3 queries.

AnalyzeConflict. Our implementation of AnalyzeConflict performs two additional optimizations over the algorithm presented in Section 5. First, our implementation does not keep all learnt lemmas in the knowledge base. In particular, since the efficiency of Decide and Propagate is sensitive to the size of the knowledge base, our implementation uses heuristics to identify likely-not-useful lemmas and periodically removes them from the knowledge base. Second, our implementation performs an optimization to facilitate the computation of components that are equivalent modulo conflict. Specifically, we maintain a mapping from each subformula φ occurring in a component specification to all components $\chi_1, \ldots, \chi_n$ such that $\Psi(\chi_i) \Rightarrow \varphi$. This off-line computation allows us to replace an SMT query with a map lookup in most cases.

Backtracking. Similar to CDCL-based SAT solvers, Neo can perform *non-chronological backtracking* by analyzing the lemma obtained from AnalyzeConflict. Specifically, suppose that the learnt lemma refers to components $\chi_1, \ldots, \chi_n$, where each χ_i was chosen at decision level d_i. Our implementation adopts the standard SAT solver heuristic of backtracking to the second highest decision level among all d_i's. This strategy often results in non-chronological backtracking and causes the algorithm to undo multiple assignments at the same time.

Instantiating Neo in new domains. As a general synthesis framework, Neo can be instantiated in new domains by providing a suitable DSL and the corresponding specifications of each DSL construct. As mentioned previously, these specifications need not be precise and typically underspecify the constructs' functionality to achieve a good tradeoff between performance overhead and pruning of the search space.

We have currently implemented two instantiations of Neo, one of which targets data wrangling tasks in R and the other of which targets list manipulations in a functional paradigm. For both domains, our specifications are expressed in quantifier-free Presburger arithmetic. More specifically, for the data wrangling domain, we use the same DSL and the same specifications considered in prior work [8]. For the list manipulation domain, we use the same DSL as in prior work [3] but write our own specification since they are not available in the DeepCoder setting [3]. In particular, our specifications capture the size of the list, the values of its first and last elements, and the minimum and maximum elements of the list.

Our experience indicates that it does not require much manual effort to instantiate Neo in new domains. For example, it took us less than one day to instantiate Neo in each of the two domains mentioned earlier.

7 Evaluation

We evaluated Neo by conducting three experiments that are designed to answer the following questions:

- **Q1.** How does Neo compare against state-of-the-art synthesis tools?
- **Q2.** How significant is the benefit of conflict-driven learning in program synthesis?

To answer these questions, we instantiated Neo on two different domains explored in prior work, namely (i) data wrangling in R and (ii) functional programming over lists. Specifically, to compare Neo against existing tools, we adopted the DSL used in Morpheus [8] for domain (i) and the language used in DeepCoder [3] for (ii). All of the experiments discussed in this section are conducted on an Intel Xeon(R) computer with an E5-2640 v3 CPU and 32G of memory, running the Ubuntu 16.04 operating system and using a timeout of 5 minutes.

7.1 Comparison against Morpheus

In our first experiment, we compare Neo against Morpheus [8], a state-of-the-art synthesis tool that automates data wrangling tasks in R. While Neo is similar to Morpheus in that both techniques use deduction to prune the search space, Morpheus uses several domain-specific heuristics that specifically target table transformations (e.g., *"table-driven type inhabitation"*). In contrast, Neo does not use any such domain-specific heuristics and directly applies the general-purpose synthesis algorithm presented in Section 4. To allow a fair comparison between the tools, we instantiate Neo with the same set of R library methods used by Morpheus as well as the same component specifications. Furthermore, since Morpheus uses a 2-gram model to prioritize its search, we also use the same statistical model in Neo's Decide component. As in Morpheus, we train the 2-gram model on 15,000 code snippets collected from Stackoverflow.

Benchmark selection. We compare Neo against Morpheus on a data set consisting of 50 challenging data wrangling tasks. Out of these 50 benchmarks, 30 correspond to the most difficult benchmarks used for evaluating Morpheus,

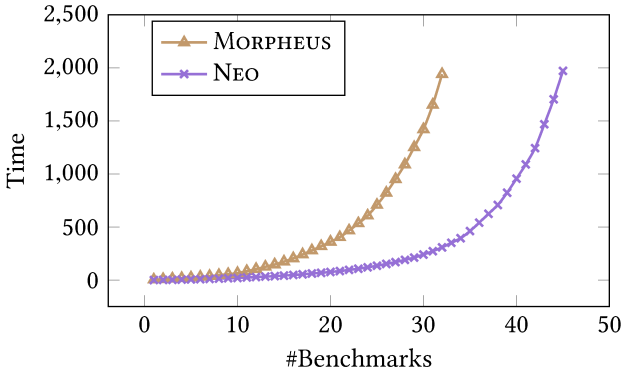


Figure 6. Comparison between Neo and Morpheus

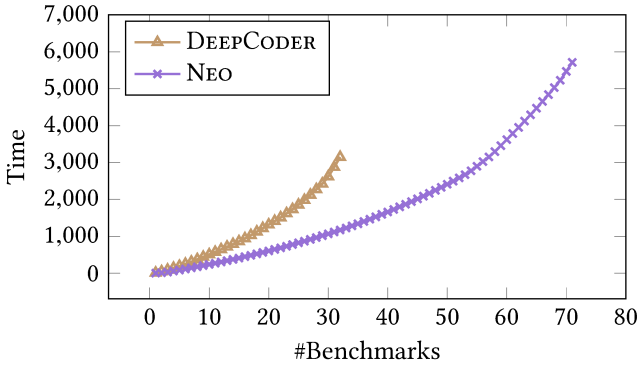


Figure 8. Comparison between Neo and DeepCoder

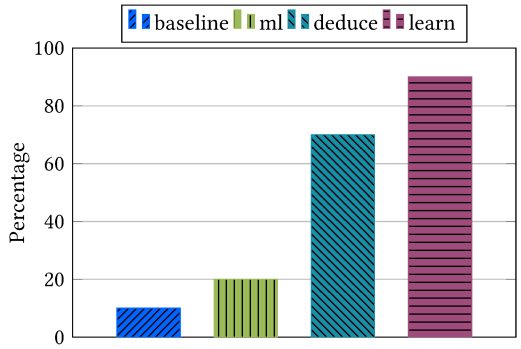


Figure 7. Impact of each component for data wrangling

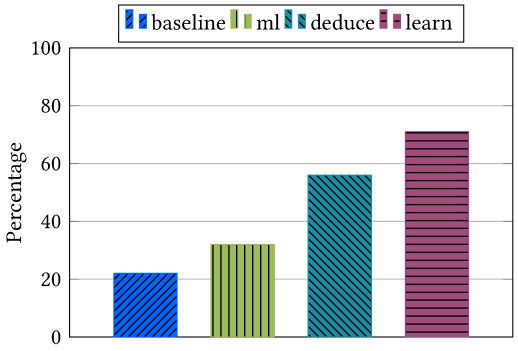


Figure 9. Impact of components for list manipulation

where difficulty is measured in terms of synthesis time. [9] We also include 20 additional benchmarks collected from Stackoverflow posts. To ensure that these benchmarks are sufficiently challenging, we consider only those posts where (a) the desired program is included in an answer, and (b) this program contains more than 12 AST nodes and at least four higher-order components.

Results. The results of our first experiment are summarized in Figure 6, which plots cumulative synthesis time (the y-axis) against the number of benchmarks solved (the x-axis). As we can see from this figure, Neo significantly outperforms Morpheus both in terms of synthesis time as well as the number of benchmarks solved within the 5 minute time limit. In particular, Neo can solve 90% of these benchmarks with an average running time of 19 seconds, whereas Morpheus solves 64% with an average running time of 68 seconds. These results indicate that our proposed synthesis methodology is able to outperform a *domain-specific* synthesis tool that specifically targets data wrangling tasks.

[9] The performance of Neo and Morpheus is very similar on the 20 easy benchmarks from the Morpheus data set. Specifically, both tools can synthesize all of these benchmarks in under 4 seconds.

Impact of different components. Figure 7 evaluates the impact of different components in Neo on the data wrangling benchmarks. In particular, we compare the number of benchmarks solved by Neo with four variants: "baseline", "ml", "deduce" and "learn". The "baseline" variant uses a depth-first search with a random decider and only solves 10% of the benchmarks. The "ml" variant uses as decider the 2-gram model from Morpheus, which further improves the number of solved benchmarks to 20%. The "deduce" variant significantly improves the number of solved benchmarks from 20% to 70% by combining statistical reasoning and deduction. Finally, Neo achieves its best performance and can solve 90% of the benchmarks by combining all of these ingredients.

7.2 Comparison against DeepCoder

In our second experiment, we compare Neo against a re-implementation of DeepCoder, which is a state-of-the-art synthesis tool that uses deep learning to guide search [3]. [10] Because DeepCoder specializes in functional programs that manipulate lists, we instantiated Neo on the same domain, using the same DSL constructs as DeepCoder. However,

[10] We implemented our own version of DeepCoder since the tool is not publicly available. Our re-implementation is faithful to the description in [3] as well as e-mail communications with the developers of DeepCoder.

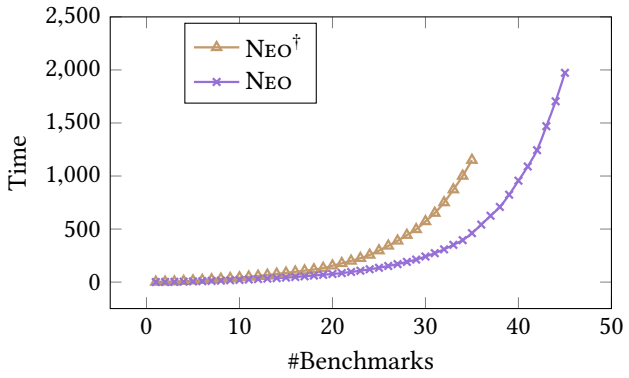


Figure 10. Impact of learning in data wrangling domain

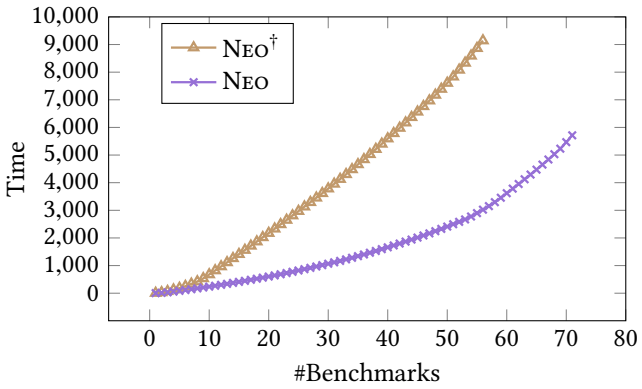


Figure 11. Impact of learning for the list domain

since DeepCoder does not utilize component specifications to prune the search space, we additionally wrote first-order specifications for each DSL construct. To allow a fair comparison between the tools, we also use the same deep neural network model used in DeepCoder. In particular, DeepCoder predicts the likelihood that χ is the right DSL operator based on the given input-output example. As in [3], we trained our deep neural network model on 1,000,000 randomly generated programs and their corresponding input-output examples.

Benchmark selection. Since the benchmarks used for evaluating DeepCoder are also not publicly available, we generate 100 benchmarks following the same methodology described in [3]. Specifically, we enumerate DSL programs with at least 5 components and randomly generate inputs and the corresponding output. This procedure is repeated for a fixed number of times until we either obtain 5 valid input-output examples or no examples have been found within the iteration limit. In the latter case, we restart this process and randomly search for a different program.

Results. The results of this experiment are summarized in Figure 8, which also plots running time against the number of solved benchmarks. As we can see from Figure 8, Neo outperforms DeepCoder in terms of running time and the number of benchmarks solved within the 5 minute time limit. In particular, Neo can solve 71% of these benchmarks with an average running time of 99 seconds. In contrast, DeepCoder solves 32% of the benchmarks with an average running time of 205 seconds.

Impact of different components. Figure 9 compares the percentage of benchmarks solved by Neo with four variants ("baseline", "ml", "deduce" and "learn"). The "baseline" variant which uses a depth-first search enumeration can only solve 22% of the benchmarks. The "ml" variant uses a neural network decider and increases the percentage of solved benchmarks to 32%. Combining statistical model and deduction ("deduce") further improves the performance of Neo to solve 56% of the benchmarks. Finally, Neo can solve 71% of the benchmarks when we combine all of these ingredients.

7.3 Benefit of Conflict-driven Learning

In our third experiment, we further evaluate the benefit of conflict-driven learning by comparing Neo against Neo†, which is a version of Neo that does not perform conflict analysis. In other words, Neo† is the same as Neo except that it does not invoke the AnalyzeConflict procedure and does not add lemmas to the knowledge base (beyond blocking the current assignment). To ensure that Neo† does not incur unnecessary overhead, we also modify the CheckConflict procedure to give a yes/no answer rather than producing a minimal unsatisfiable core.

The results of this experiment are summarized in Figures 10 and 11. Specifically, Figure 10 compares Neo against Neo† on the data wrangling benchmarks from Section 7.1, whereas Figure 11 shows the same comparison for the list manipulation benchmarks used in Section 7.2. As we can see from these figures, learning has a very significant positive impact on the overall performance of Neo. Specifically, in the data wrangling domain, Neo† times out on 30% of the benchmarks and its average running time is 38 seconds. On the other hand, Neo only times out on 10% of the benchmarks while maintaining an average running time of 19 seconds. As shown in Figure 11, the effect of learning is even more substantial in the list manipulation domain. Specifically, Neo† times out on 44% of the benchmarks and its average running time is 199 seconds. In contrast, Neo only times out on 29% of the benchmarks and has an average running time of 99 seconds.

Discussion. To evaluate the impact of conflict-driven learning on scalability, we classify our data wrangling tasks into three categories (i.e., "easy", "moderate", and "hard") depending on the complexity of the target program. While Neo is 2.7x faster than Neo† on "easy" benchmarks, Neo outperforms Neo† by 5.7x on the medium category. For benchmarks

Benchmark	Neo Speedup		
	Min	Avg	Max
Data Wrangling	2.8	5.5	17.6
Lists	1.6	4.0	14.8

Table 2. Impact of learning

in the "hard" category, Neo is 19.8x faster than Neo† on average. [11] Also, to further evaluate the impact of learning on hard benchmarks, we conduct an additional experiment on exactly those problems that are solved by Neo but not by Neo† within the 5 minute time-limit. For these 25 benchmarks, we re-run Neo† with a much longer time limit of one hour. Table 2 shows the impact of learning on these harder benchmarks. Specifically, Neo has an average speedup of 5.5x and 4.0x on the Morpheus and DeepCoder benchmarks respectively. The maximum speedup is 17.6x for the data wrangling domain and 14.8x for the list manipulation programs. For example, Neo† takes around 45 minutes to solve a benchmark that can be solved by Neo in less than 3 minutes.

8 Limitations

In this section, we discuss some of the limitations of the proposed approach. First, because our method does not reason about termination, Neo does not currently support synthesizing recursive programs. Second, even though Neo is a generic framework that can be instantiated in different domains, it is likely to be more effective when synthesizing functional programs that can be expressed as a composition of library methods. Third, the effectiveness of our technique depends on the quality of the specifications. For example, if the specifications are too detailed, they might significantly increase SMT solving overhead without providing additional benefit. On the other extreme, if the specifications are very coarse-grained, Neo may not be able to make useful deductions and learn from conflicts. Overall, we believe that Neo is likely to be more effective in domains where the DSL includes many related components and the specifications of these components expose their shared functionality.

9 Related Work

Program synthesis is an active research topic that has found many applications, including string processing [13, 26] bit-vector manipulations [16], data wrangling [8, 37], query synthesis [36, 40, 42], API completion [9, 15, 17], functional programming [10, 22, 23], and data processing [28, 39]. In what follows, we discuss prior work that is most closely related to our proposed approach.

Logical reasoning in program synthesis. The technique proposed in this paper leverages logical specifications of DSL constructs to enable conflict-driven learning. While we are not aware of prior work that learns useful lemmas from conflicts, there are many prior techniques that leverage logical specifications to aid synthesis. In particular, synthesis algorithms that use specifications can be grouped along two axes: (a) whether they require exact vs. approximate specifications, and (b) whether they use specifications to guide search or completely reduce synthesis to constraint solving.

There are several techniques that formulate synthesis as a constraint solving problem [12, 14, 16, 34]. For example, Brahma [16] uses component specifications to generate an $\exists\forall$ formula such that any satisfying assignment to this formula is a solution to the synthesis problem. More recent work such as Synudic [12] also reduces synthesis to constraint solving, but uses the abstract semantics of components to simplify the resulting constraint-solving problem.

An alternative approach is to formulate synthesis as a search –rather than constraint-solving– problem and use logical specifications to prune the search space [8, 10, 23]. For example, Synquid uses liquid type specifications to avoid the exploration of some program terms. Other tools, such as λ^2 [10] and Morpheus [8], similarly use logical reasoning to prune the search space but allow these specifications to be over-approximate. Neo differs from all prior techniques in that it leverages logical specifications to infer useful lemmas that prevent the exploration of spurious programs that are semantically similar to previously encountered ones.

Another work that is closely related to Neo is Blaze [38], which performs program synthesis using *counterexample-guided abstraction refinement*. Similar to our approach, Blaze prunes its search space also by using a form of deductive reasoning. However, a key difference is that Blaze uses abstract interpretation to enumerate only those programs that satisfy the specification with respect to a given abstract semantics. In contrast, Neo uses automated theorem proving (i.e., SAT and SMT) to prune partial programs that have no feasible completion. Furthermore, while both approaches perform some form of learning, Blaze learns a new abstract domain during refinement, whereas Neo directly learns infeasible partial programs. We believe that Neo has two main advantages over Blaze: First, Neo does not require a domain expert to provide an abstract domain in the form of predicate templates. Second, Neo can handle higher-order constructs more naturally and efficiently compared to Blaze.

Comparison to CEGIS. Counterexample-guided inductive synthesis (CEGIS) is a popular framework for synthesizing programs that satisfy a given specification Φ [2, 29, 30]. The key idea underlying CEGIS is to decompose the problem into separate *synthesis* and *verification* steps. Specifically, the synthesizer proposes a candidate program P that is consistent with a given set of examples, and the verifier checks whether P actually satisfies Φ. If this is not the case, then the verifier provides the synthesizer with a counterexample. Our baseline synthesis algorithm (i.e., without learning) can be

[11] We did not perform the same comparison for the list domain since all benchmarks have similar complexity.

roughly formulated in the CEGIS framework. At each step, the synthesizer (i.e., Decide) proposes a *partial program P*. Then, the verifier (i.e., Deduce) checks whether there is any completion of P that can satisfy Φ. If not, it provides counterexample (i.e., an UNSAT core τ). Thus, our work can be thought of as extending CEGIS to incorporate learning. In particular, the baseline algorithm does not learn any additional information from a counter-example τ reported by the verifier (other than the trivial fact that τ is unsatisfiable). In contrast, given a MUC reported by our verifier, Analyze-Conflict learns new lemmas that typically rule out many additional programs.

Machine learning for synthesis. The work combining machine learning with program synthesis has roughly followed two paths. The first approach uses neural networks (or "neural programmers") to directly generate programs [20, 21]. This line of work is inspired by sequence-to-sequence models in machine translation, where one neural network (the "encoder") encodes an input phrase in the source language (e.g., English) into a vector, and a second neural network (the "decoder") decodes this vector into a phrase in the target language (e.g., French) [33]. The neural programmer follows the same paradigm, where the "source language" is the specification (e.g., examples or natural language), and the "target language" is the programming language (e.g., SQL).

The second approach, which is the one we adopt, incorporates statistical knowledge to guide a symbolic program synthesizer. These approaches train a statistical model to predict the most promising program to explore next. For instance, Menon et al. use a log-linear model to predict the most likely DSL operator based on features of the input-output example [19], and DeepCoder uses a deep neural network to learn features that can be used to make such predictions [3]. Alternatively, Raychev et el. use an n-gram model, trained on a large database of code, to predict the most likely completion of a hole based on its ancestors in the AST [25]. Later work by Raychev et al. extends this approach to the case of program synthesis with noisy input-output examples [24], and Feng et al. use a similar n-gram model for synthesizing table transformations [8]. Similar to all of these techniques, Neo also uses a statistical model to predict the most likely completion of a hole during its Decide step but also takes into account the "hard constraints" encoded in its knowledge base.

Conflict-driven learning. Our proposed synthesis framework is directly inspired by the success of CDCL-style SAT and SMT solvers [5, 11, 18, 41]. Given a partial assignment that results in a conflict, the idea is to learn a so-called *conflict clause* that prevents similar conflicts in the future. While the architecture of our synthesis algorithm is (intentionally) very similar to CDCL-style SAT solvers, the mechanisms used for learning these conflict clauses are very different. In particular, SAT solvers typically learn a conflict clause by constructing an *implication graph* that describes which assignments lead to which other assignments as a result of unit propagation. Given such an implication graph, a conflict clause is inferred by performing resolution between clauses that contribute to the conflict. In contrast, our method learns "conflict clauses" (i.e., lemmas) by identifying DSL constructs that are equivalent modulo conflict. Our method also differs from CDCL-style constraint solvers in the way it makes decisions and performs deduction. In particular, we use a statistical model to make assignments and detect conflicts by issuing an SMT query.

10 Conclusion and Future Work

We have presented a new synthesis framework based on the idea of *conflict-driven learning*. Given a spurious partial program that violates the specification, the idea is to infer a lemma that can be used to prevent similar mistakes in the future. Our synthesis algorithm infers these lemmas by identifying DSL constructs that are *equivalent modulo conflict*, meaning that replacing one component with the other results in an another infeasible program.

We have implemented these ideas in a synthesis framework called Neo and instantiated Neo for two different application domains, namely data wrangling and list manipulation. Neo is publicly available [7] and can be instantiated in other application domains by providing a suitable DSL and its specification.

We have evaluated Neo on 130 benchmarks that pertain to data wrangling and list transformation tasks. Our evaluation shows that Neo outperforms state-of-the art synthesis tools, namely Morpheus and DeepCoder, that specialize in these two domains respectively. Our experiments also demonstrate that conflict-driven learning substantially improves the capabilities of the synthesizer.

In future work, we are interested in instantiating Neo in more application domains. Since conflict-driven learning is particularly helpful in domains that involve a large number of components, we plan to use Neo to synthesize programs that require the use of many different libraries.

Acknowledgments

We thank Thomas Dillig, Navid Yaghmazadeh, Xinyu Wang, Yuepeng Wang, Jiayi Wei, and Jia Chen for their insightful comments. We would also like to thank our shepherd Ben Zorn, and the anonymous reviewers for their helpful feedback.

This work was supported in part by NSF Award #1453386, #1162076, #1762299, AFRL Awards #8750-14-2-0270, and a Google Fellowship. The views, opinions, and findings contained in this paper are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

References

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Proc. International Conference on Computer Aided Verification*. Springer, 934–950.

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *Proc. Formal Methods in Computer-Aided Design*. IEEE, 1–8.

[3] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. Deepcoder: Learning to write programs. In *Proc. International Conference on Learning Representations*. OpenReview.

[4] Daniel Le Berre and Anne Parrain. 2010. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation* (2010), 59–6.

[5] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. 2009. Conflict-driven clause learning SAT solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications* (2009), 131–153.

[6] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proc. Tools and Algorithms for Construction and Analysis of Systems*. Springer, 337–340.

[7] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Neo. http://utopia-group.github.io/neo/.

[8] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 422–436.

[9] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas Reps. 2017. Component-Based Synthesis for Complex APIs. In *Proc. Symposium on Principles of Programming Languages*. ACM, 599–612.

[10] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 229–239.

[11] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2004. DPLL(T): Fast Decision Procedures. In *Proc. International Conference on Computer Aided Verification*. Springer, 175–188.

[12] Adrià Gascón, Ashish Tiwari, Brent Carmer, and Umang Mathur. 2017. Look for the Proof to Find the Program: Decorated-Component-Based Program Synthesis. In *Proc. International Conference on Computer Aided Verification*. Springer, 86–103.

[13] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proc. Symposium on Principles of Programming Languages*. ACM, 317–330.

[14] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 62–73.

[15] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 27–38.

[16] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proc. International Conference on Software Engineering*. ACM/IEEE, 215–224.

[17] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid mining: helping to navigate the API jungle. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 48–61.

[18] Joao Marques-Silva and Karem A. Sakallah. 1999. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Computers* 48, 5 (1999), 506–521.

[19] Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. 2013. A machine learning framework for programming by example. In *Proc. International Conference on Machine Learning*. Proceedings of Machine Learning Research, 187–195.

[20] Arvind Neelakantan, Quoc V Le, Martin Abadi, Andrew McCallum, and Dario Amodei. 2017. Learning a natural language interface with neural programmer. In *Proc. International Conference on Learning Representations*. OpenReview.

[21] Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. 2016. Neural programmer: Inducing latent programs with gradient descent. In *Proc. International Conference on Learning Representations*. OpenReview.

[22] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 619–630.

[23] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. *Proc. Conference on Programming Language Design and Implementation* (2016), 522–538.

[24] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. Learning programs from noisy data. In *Proc. Symposium on Principles of Programming Languages*. ACM, 761–774.

[25] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 419–428.

[26] Rishabh Singh. 2016. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment* 9, 10 (2016), 816–827.

[27] Rishabh Singh and Sumit Gulwani. 2016. Transforming spreadsheet data types using examples. In *Proc. Symposium on Principles of Programming Languages*. ACM, 343–356.

[28] Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 326–340.

[29] Armando Solar-Lezama. 2008. *Program synthesis by sketching*. University of California, Berkeley.

[30] Armando Solar-Lezama. 2009. The Sketching Approach to Program Synthesis.. In *Proc. Asian Symposium on Programming Languages and Systems*. Springer, 4–13.

[31] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. 2005. Programming by sketching for bit-streaming programs. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 281–294.

[32] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 404–415.

[33] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*. 3104–3112.

[34] Ashish Tiwari, Adria Gascón, and Bruno Dutertre. 2015. Program synthesis using dual interpretation. In *Proc. International Conference on Automated Deduction*. Springer, 482–497.

[35] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. *Proc. Conference on Programming Language Design and Implementation* (2013), 287–296.

[36] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 452–466.

[37] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of Data Completion Scripts using Finite Tree Automata. In *Proc. International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 62:1–62:26.

[38] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program Synthesis using Abstraction Refinement. In *Proc. Symposium on Principles of Programming Languages*. ACM, 63:1–63:30.

[39] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing transformations on hierarchically structured

data. In *Proc. Conference on Programming Language Design and Implementation*. ACM, 508–521.

[40] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. In *Proc. International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 63:1–63:26.

[41] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. 2001. Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In *Proc. of International Conference on Computer-Aided Design*. IEEE Computer Society, 279–285.

[42] Sai Zhang and Yuyin Sun. 2013. Automatically synthesizing sql queries from input-output examples. In *Proc. International Conference on Automated Software Engineering*. IEEE, 224–234.