The Dissertation Committee for Yu Feng
certifies that this is the approved version of the following dissertation:

Program Synthesis using Statistical Models and Logical Reasoning

Committee:

Isil Dillig, Supervisor

Raymond Mooney

Philipp Krähenbühl

Alex Aiken

Program Synthesis using Statistical Models and Logical Reasoning

by

Yu Feng

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2018

Dedicated to my mother, my wife *Lijuan*,

and my newborn baby, Byron.

Acknowledgments

Pursuing a Ph.D. is the best decision in my life.

First, I sincerely thank my advisor, Isil Dillig, for her unconditional support during the past five years. She is the best advisor I could ever imagine and she completely reshaped my life through her endless guidance. Because of her, I had chances to work with the most talented researchers in this world, receive the best education, become a computer scientist, and continue my future research as a professor. Isil not only taught me how to perform top-notch research, but also repeatedly illustrated how to form research ideas, write good papers, give impressive talks, communicate with people, and advise students. I wish I could be as good as her on advising my future students.

Second, I would like to thank Saswat Anand, who was my first mentor on program analysis, as well as Alex Aiken, for his unconditional support during my Ph.D., job search, and his attitude on doing high impact research. I would also like to thank Thomas Dillig, for his invaluable feedback during my Ph.D. and world-class barbecue. Thanks to Swarat Chaudhuri, Bor-Yuh Evan Chang, Vivek Sarkar, and Keshav Pingali for their feedback on my job talk.

Pursuing a Ph.D. alone can be miserable and I feel so lucky to work with my reliable and talented labmates, Ruben Martins, Osbert Bastani, Jia Chen, Yuepeng Wang, Kostas Ferles, Xinyu Wang, Navid Yaghmazadeh, Marijn

Heule, Valentin Wüstholz, Arati Kaushik, Jiayi Wei, Jacob Van Geffen, and all other members in the Utopia group. Thank you all for bearing with both my sensible and non-sensible ideas, and thanks to Ruben Martins for cheering me up after paper rejections. Thanks to Justine Sherry and Kostas Ferles for their feedback on my research statement.

Furthermore, I would also like to thank Lazaro Clapp, Manolis Papadakis, Stefan Heule, and all other members in the STAMP group, for giving me feedback during my visit at Stanford. Meanwhile, I want to express my sincere gratitude for all the valuable comments from my other committee members, Raymond Mooney and and Philipp Krähenbühl.

Finally, a special thanks to my family: my lovely wife, Lijuan Cheng, our baby boy, Byron, my mother, Xiaoling Huang, and my sibling, Jing Feng, for their unlimited love and sacrifices.

Program Synthesis using Statistical Models and Logical Reasoning

Publication No. _____

Yu Feng, Ph.D.
The University of Texas at Austin, 2018

Supervisor: Isil Dillig

Complex APIs in new frameworks (Spark, R, TensorFlow, etc) have imposed steep learning curves on everyone, especially for people with limited programming backgrounds. For instance, due to the messy nature of data in different application domains, data scientists spend close to 80% of their time in data wrangling tasks, which are considered to be the "janitor work" of data science. Similarly, software engineers spend hours or even days learning how to use APIs through official documentation or examples from online forums.

Program synthesis has the potential to automate complex tasks that involve API usage by providing powerful search algorithms to look for executable programs that satisfy a given specification (input-output examples, partial programs, formal specs, etc). However, the biggest barrier to a practical synthesizer is the size of search space, which increases strikingly fast with the complexity of the programs and the size of the targeted APIs.

To address the above issue, this dissertation focuses on developing algorithms that push the frontiers of program synthesis. First, we propose a type-directed graph reachability algorithm in SyPet, a synthesizer for assembling programs from complex APIs. Second, we show how to combine enumerative search with lightweight constraint-based deduction in Morpheus, a synthesizer for automating real-world data wrangling tasks from input-output examples. Finally, we generalize the previous approaches to develop a novel conflict-driven synthesis algorithm that can learn from past mistakes.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Complex APIs in new frameworks (Spark, R, TensorFlow, etc) have imposed steep learning curves on everyone, especially for people with limited programming backgrounds. For instance, due to the messy nature of data in different application domains, data scientists spend close to 80% of their time in data wrangling tasks, which are considered to be the "janitor work" of data science. Similarly, software engineers spend hours or even days learning how to use APIs through official documentation or examples from online forums.

Program synthesis has the potential to automate complex tasks that involve API usage by providing powerful search algorithms to look for executable programs that satisfy a given specification (input-output examples, partial programs, formal specs, etc). However, the biggest barrier to a practical synthesizer is the size of search space, which increases strikingly fast with the complexity of the programs and the size of the targeted APIs.

To overcome the space explosion issue, researchers from both the programming languages and machine learning communities have started to develop practical synthesis algorithms that can address real-world challenges. Those approaches can be categorized into two main classes, namely, those based on

statistical models and *logical reasoning*.

Statistical models. There are several ongoing synthesis efforts in the ML community, and one representative approach is to incorporate statistical knowledge to guide a symbolic program synthesizer. These approaches train statistical models to predict the most promising program to explore next. For instance, the statistical model can be a log-linear model to predict the most likely DSL operator based on features of the input-output example [74], a deep neural network to learn features that can be used to make such predictions [13], or 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 [17].

Logical reasoning. Although statistical models are very successful in providing the most likely programs with respect to the distribution in the training set, they alone are not sufficient to solve complex tasks because the search space is still very large. To prune the search space, there are many prior techniques [42, 58, 37, 113] that leverage logical specifications to aid synthesis. The logical specifications can be the types [80, 84], or first-order logic formulas that capture the formal semantics of a given task [58].

While synthesis algorithms based on logical reasoning can incorporate domain specific knowledge to significantly prune the search space, they alone are still difficult to scale to large programs as it lacks of prior knowledge of the problems. As a result, their search strategies are typically based on some

heuristics, such as length of the candidate programs, which may not be the best strategies in practice. In this dissertation, to overcome the limitations of previous mentioned approaches, we show how to build scalable program synthesizers for real-world synthesis tasks by combining the power of logical reasoning and statistical models, and empirically demonstrate their practicality and effectiveness in the context of automating data wrangling tasks and list manipulation in functional programming. Specifically, to handle programming tasks that require using an API with thousands of methods, we present a compact Petri-net representation to model relationships between methods in an API, as well as a novel type-directed algorithm for component-based synthesis in Chapter 2. Given a target method signature $\mathcal{S}$, our approach performs reachability analysis on the underlying Petri-net model to identify sequences of method calls that could be used to synthesize an implementation of $\mathcal{S}$. The programs synthesized by our algorithm are guaranteed to type check and pass all test cases provided by the user. We have implemented this approach in a tool called SyPet, and used it to successfully synthesize real-world programming tasks extracted from on-line forums and existing code repositories. We also compare SyPet with two state-of-the-art synthesis tools, namely InSynth and CodeHint, and demonstrate that SyPet can synthesize more programs in less time.

Since SyPet uses primitive types as the coarse-grained specifications to prune infeasible candidates that are not well-typed, it will degrade to naive enumerative search in some domains which typically consume and produce

the same types, such as string manipulation and data wrangling. To address the disadvantage of SyPet, in Chapter 3, we present a novel component-based synthesis algorithm that marries the power of type-directed search with lightweight SMT-based deduction and partial evaluation. Given a set of components together with their *over-approximate* first-order specifications, our method first generates a *program sketch* over a subset of the components and checks its feasibility using an SMT solver. Since a program sketch typically represents *many* concrete programs, the use of SMT-based deduction greatly increases the scalability of the algorithm. Once a feasible program sketch is found, our algorithm completes the sketch in a bottom-up fashion, using partial evaluation to further increase the power of deduction for rejecting partially-filled program sketches. We apply the proposed synthesis methodology for automating a large class of data preparation tasks that commonly arise in data science. We have evaluated our synthesis algorithm on dozens of data wrangling and consolidation tasks obtained from on-line forums, and we show that our approach can automatically solve a large class of problems encountered by R users.

While both SyPet and Morpheus incorporate statistical models to speed up enumerative search and use logical reasoning to prune search space, none of them can learn from past mistakes. To address this limitation as well as provide a unified framework that combines statistical models and logical reasoning in a natural way, in Chapter 4 we propose a new *conflict-driven* program synthesis framework 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.

In summary, we show in this dissertation that by combining the power of logical reasoning and statistical models, we could build scalable synthesizers that can automate a large class of complex and tedious tasks facing by majority of the end-users.

Chapter 2

SyPet [1]

Component-based approaches to program synthesis assemble programs from a database of existing components, such as methods provided by an API. In practice, one challenge is to model relationships between methods which can have multiple arguments or side effects, as well as an efficient algorithm to enumerate viable candidates. To address this problem, in this chapter, we use of a compact Petri-net representation to model relationships between methods in an API, as well as a novel type-directed algorithm for component-based synthesis. Specifically, given a target method signature $\mathcal{S}$, our approach performs reachability analysis on the underlying Petri-net model to identify sequences of method calls that could be used to synthesize an implementation of $\mathcal{S}$. The programs synthesized by our algorithm are guaranteed to type check and pass all test cases provided by the user.

We have implemented this approach in a tool called SyPet, and used it to successfully synthesize real-world programming tasks extracted from online forums and existing code repositories. We also compare SyPet with two state-of-the-art synthesis tools, namely InSynth and CodeHint, and

[1] Parts of this chapter have appeared in [27].

demonstrate that SyPet can synthesize more programs in less time. Finally, we compare our approach with an alternative solution based on hypergraphs and demonstrate its advantages.

The rest of this chapter is organized as follows: First, we start by presenting an example to motivate our approach (Section 2.2) and provide some necessary background on Petri nets (Section 2.3). After presenting an outline of the main synthesis algorithm in Section 2.4, we then elaborate on the core technical pieces in Section 2.5, Section 2.6 and Section 2.7. In Section 2.8 and Section 2.9, we describe implementation details and present our main experimental results.

2.1 Overview

The goal of *component-based synthesis* is to automatically generate loop-free programs from a collection of base components, such as methods provided by an API [43, 57]. Considering the explosion of software libraries over the last few decades, component-based synthesis promises to simplify programming by automatically composing the building blocks needed to achieve some implementation task. Hence, instead of spending precious time in learning how to use existing libraries, programmers can focus on challenging algorithmic tasks.

Despite significant advances in component-based synthesis over the last several years [57, 44, 43, 83], existing algorithms have two key shortcomings: First, they can only handle a small number of components, typically in the range

of 5-20 methods; but real-world APIs typically involve thousands of procedures. Second, most existing tools require logical specifications for the underlying components; however, few APIs contain methods that are formally specified. As a result, the applicability of component-based synthesis remains limited to domain-specific applications, such as bit-vector, string, or data-structure manipulations [57, 29, 97].

In this chapter, we propose a new algorithm for component-based synthesis that overcomes both of these difficulties. Similar to recent work on type-directed API-completion [68, 49, 80, 47], our algorithm uses types as a coarse proxy for logical specifications and can handle APIs with thousands of procedures. However, unlike API completion tools, our algorithm does not require a partial implementation, and can synthesize complete programs from method signatures and test cases. The programs synthesized by our approach are always guaranteed to type-check and pass all user-provided tests. Furthermore, our approach is oblivious to the underlying components, and can be used to synthesize Java code using any combination of APIs.

The workflow of our synthesis algorithm is illustrated in Figure 3.1. At a technical level, a key idea underlying our approach is to represent relationships between API components using a certain kind of *Petri net* where places (nodes) correspond to types, transitions represent methods, and tokens denote the number of program variables of a given type. For example, Figure 2.6 shows a Petri net that describes the relationships between a subset of the functions in the `java.awt.geometry` API. Given such a Petri net $\mathcal{N}$ and a

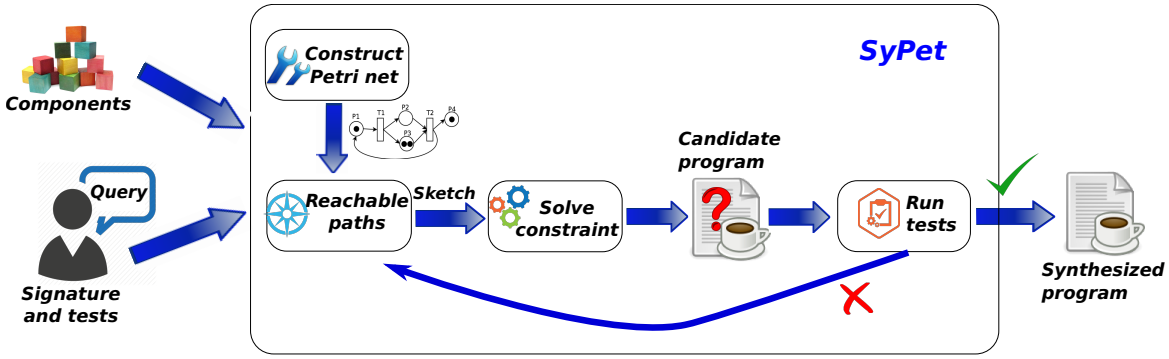


Figure 2.1. Workflow of the SYPET tool

target configuration defined by the method signature, our algorithm performs reachability analysis on $\mathcal{N}$ to identify a sequence of transitions (i.e., method calls) that "produce" the output type by "consuming" the input types.

In our approach, a reachable path in the Petri-net model corresponds to a program sketch rather than a complete executable program. In particular, to keep the underlying Petri net representation compact, our algorithm deliberately decomposes the synthesis task into two separate *sketch-generation* and *sketch-completion* phases. Hence, after we perform reachability analysis on the Petri net, we must still complete the sketch by determining what arguments to provide for each procedure. Toward this goal, our algorithm generates constraints that encode various syntactic and semantic requirements on the synthesized program, and uses a SAT solver to find a model. The satisfying assignment produced by the solver is then used to generate a candidate implementation that can be tested. If the synthesized program fails any test case, our algorithm backtracks and generates a different implementation, either by finding another model of

```
public void test1() {
    Area a1 = new Area(new Rectangle(0, 0, 10, 2));
    Area a2 = new Area(new Rectangle(-2, 0, 2, 10));
    Point2D p = new Point2D.Double(0, 0);
    assertTrue(a2.equals(rotate(a1, p, Math.PI/2)));
}
```

Figure 2.2. Example test case for the `rotate` method

the SAT formula or by exploring a different reachable path in the Petri net.

At a very high level, our algorithm can be viewed as a generalization of techniques that use graph-reachability analysis for API completion. For example, standard graph reachability has been used to synthesize *jungloids*, which are sequences of *single argument* methods [68]. However, because our goal is to develop a *general solver* for component-based synthesis, we require a more expressive graph representation that can faithfully model relationships between multi-argument functions. In this work, we choose to use Petri nets as the underlying formalism because they have several advantages compared to other generalized graph representations, such as hypergraphs. As we show later in Section 2.9, Petri nets allow us to synthesize a larger class of imperative programs, including those that call the same procedure multiple times or where components can have side effects.

2.2 Motivating Example

Consider a programmer, Bob, who wants to implement functionality for rotating a 2-dimensional geometric object. Specifically, Bob has the following

signature in mind:

```
Area rotate(Area obj, Point2D pt, double angle)
```

Here, the `rotate` method should take a 2-dimensional object called `obj` and return a new object that is the same as `obj` except that it has been rotated by the specified `angle` around the specified point `pt`. The types `Area` and `Point2D` are defined in the `java.awt.geom` library. Bob thinks that there is probably a way of implementing this functionality using the `java.awt.geom` package, but he cannot figure out how.

SyPet can help a programmer like Bob by automatically synthesizing the desired `rotate` method. To use SyPet, Bob only needs to provide (a) the method signature above, and (b) write one or more test cases. In this case, suppose Bob has written the unit test shown in Figure 2.2. This test creates a rectangle `a1` and its variant `a2` that has been rotated by 90°; it then asserts that invoking `rotate` on `a1` yields an object that is identical to `a2`.

Given this test case and method signature, SyPet automatically synthesizes the implementation of `rotate` shown in Figure 2.3 in 2.01 seconds. Observe that writing this code is non-trivial for a programmer like Bob for several reasons: First, Bob must know about the existence of a class called `AffineTransform` in the `java.awt.geom` library. Second, he must know about (and correctly use) the `setToRotation` method, which sets up a matrix representing the desired transformation. Finally, the call to `createTransformedArea` creates a new `Area` object that contains the same geometry as `obj`, but transformed by

```
Area rotate(Area obj, Point2D pt, double angle) {
    AffineTransform at = new AffineTransform();
    double x = pt.getX();
    double y = pt.getY();
    at.setToRotation(angle, x, y);
    Area obj2 = obj.createTransformedArea(at);
    return obj2;
}
```

Figure 2.3. Implementation synthesized by SyPet

the specified transformation `at`. Hence, from the user's perspective, SyPet can significantly boost programmer productivity by automatically finding the relevant API methods and invoking them in the right manner.

From the synthesizer's perspective, automatically generating an implementation of `rotate` offers several challenges: First, the `java.awt.geom` library, which we use to synthesize this code, contains 725 methods. Hence, even though the implementation consists of just 6 lines of code, the number of components is quite large. Second, even when we restrict ourselves to code snippets of length 3 (measured in terms of the number of API calls), there are already over 3.1 million implementations of `rotate` that type check. Because the search space is so large, finding the right implementation of `rotate` is akin to finding a needle in the proverbial hay stack.

2.3 Primer on Petri Nets

Because the remainder of this paper relies on basic knowledge about Petri nets, we first provide some background on this topic.

2.3.1 Petri Net Definition

A Petri net is a bipartite graph with two types of nodes: *places*, which are drawn as circles, and *transitions*, represented as solid bars (see Figure 2.4). Each place in a Petri net can contain a number of *tokens*, which are drawn as dots and typically represent resources. A *marking* (or *configuration*) of a Petri net is a mapping from each place p to the number of tokens at p. Transitions in the Petri net correspond to events that change the marking. In particular, incoming edges of a transition t represent necessary conditions for t to fire, and outgoing edges represent the outcome. For example, consider transition T_1 from Figure 2.4. A necessary condition for T_1 to fire is that there must be at least one token present at P_1, because the incoming edge to T_1 has weight 1. Because the precondition of this transition is met, we say that T_1 is *enabled*. If we fire transition T_1, we consume one token from place P_1 and produce one token at place P_2, because the outgoing edge of T_1 is also labeled with 1. Figure 2.5 shows the result of firing T_1 at the configuration shown in Figure 2.4. Observe that transition T_2 is *disabled* in both Figure 2.4 and Figure 2.5 because there are fewer than two tokens at place P_2.

Definition 2.1. *(***Petri net***) A Petri net* $\mathcal{N}$ *is a 5-tuple* (P, T, E, W, M_0) *where* P *is a set of places,* T *is a set of transitions, and* $E \subseteq (P \times T) \cup (T \times P)$ *is the set of edges (arcs). Finally,* W *is a mapping from each edge* $e \in E$ *to a weight, and* M_0 *is the initial marking of* $\mathcal{N}$.

Example 2.1. *Consider the Petri net shown in Figure 2.4. Here, we have* $P = \{P_1, P_2, P_3\}$ *and* $T = \{T_1, T_2, T_3\}$. *Let* e^* *be the edge* $P_2 \to T_2$. *We have*

$W(e^*) = 2$, *and* $W(e) = 1$ *for all other edges* e *in* E *(e.g.,* $P_1 \to T_1$*). The initial marking* M_0 *assigns* P_1 *to 2, and all other places to* 0.

A *run* (or *trace*) of a Petri net $\mathcal{N}$ is a sequence of transitions that are fired. For instance, some feasible runs of the Petri net shown in Figure 2.4 include T_1, T_1, T_2 and T_1, T_1, T_2, T_3. However, T_1, T_2 and T_1, T_2, T_3 are not feasible.

2.3.2 Reachability and k-safety in Petri Nets

A key decision problem about Petri nets is *reachability*: Given Petri net $\mathcal{N}$ with initial marking M_0 and *target marking* M^*, is it possible to reach M^* by starting at M_0 and firing a sequence of transitions? For instance, consider Figure 2.4 and target marking $M^* = [P_1 \mapsto 0, P_2 \mapsto 0, P_3 \mapsto 1]$. This marking is reachable because we can get to marking M^* by firing the sequence of transitions T_1, T_1, T_2. The *reachable state space* of a Petri net $\mathcal{N}$, denoted $\mathcal{R}(\mathcal{N})$, is the set of all markings that are reachable from the initial state. Given Petri net $\mathcal{N}$ and target marking M^*, a run of $\mathcal{N}$ is *accepting* if it ends in M^*.

Another important concept about Petri nets is *k-safety*: A Petri net $\mathcal{N}$ is said to be *k-safe* if no place contains more than k tokens for any marking in $\mathcal{R}(\mathcal{N})$. For example, the Petri net of Figure 2.4 is 2-safe, because no place can contain more than 2 tokens in any configuration. However, if we modify this Petri net by adding a back edge from T_1 to P_1 (with an arc weight of 1), then the resulting Petri net is not k-safe for any k. As we will see later, the notion of k-safety plays an important role in the reachability analysis of Petri nets because the reachable state space $\mathcal{R}(\mathcal{N})$ is bounded iff $\mathcal{N}$ is k-safe.

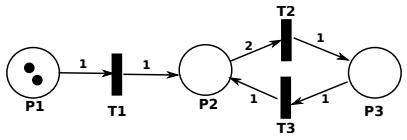


Figure 2.4. A simple Petri net

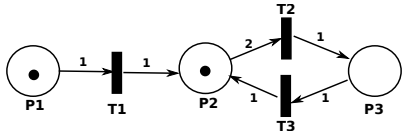


Figure 2.5. Result of firing T_1 in Figure 2.4

2.4 Algorithm Overview

We now give an overview of SyPet's synthesis algorithm and illustrate how it works on the example from Section 2.2. As shown in Algorithm 3.1, the Synthesize procedure takes a method signature $\mathcal{S}$, a set of components Λ, and test cases $\mathcal{E}$. Its output is either $\perp$, meaning that the specification cannot be synthesized using components Λ, or a well-typed program that passes all test cases $\mathcal{E}$.

Petri-net construction. The first step of our synthesis algorithm is to construct a Petri net using *signatures* of components in Λ. In particular, the procedure ConstructPetri in Algorithm 3.1 constructs a Petri net $\mathcal{N}$ where each transition is a component $f \in \Lambda$ and each place correspond to a type. If there is an edge in the Petri net from τ to f with weight w, component f takes w arguments of type τ. Similarly, an edge from f to τ' indicates that f's

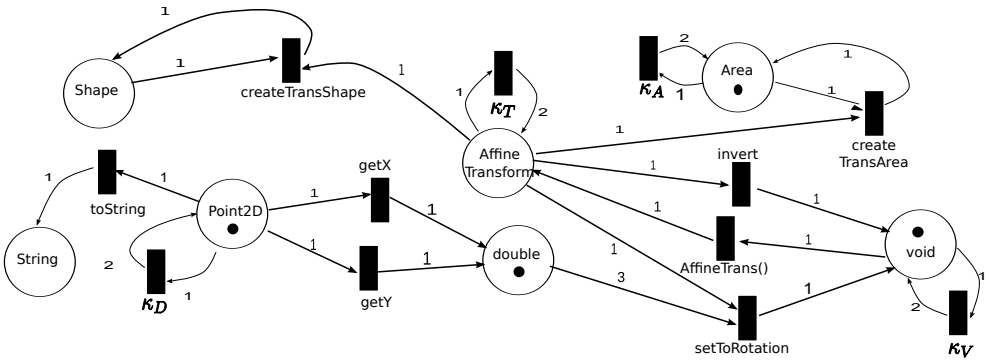


Figure 2.6. Petri net for motivating example

return value has type τ'.

Example 2.2. *Figure 2.6 shows (a small part of) the Petri net generated by* ConstructPetri *for the example from Section 2.2. The transition labeled* `getX` *has one incoming edge of weight 1 from* `Point2D` *because it takes a single argument of this type. There is also an edge from* `getX` *to* `double` *because* `getX`*'s return value is* `double`*. As another example, the weight of the edge from* `double` *to* `setToRotation` *is 3 because this method requires three arguments of type* `double`*. Note that Figure 2.6 also contains special* ***clone transitions*** *labeled κ: Intuitively, these κ transitions allow us to duplicate tokens. As we will see in Section 2.5, the clone transitions allow us to reuse program variables in the synthesis context.*

The initial and final markings on the Petri net are determined by the signature $\mathcal{S}$ provided by the user. For instance, the tokens on the Petri net $\mathcal{N}$

Algorithm 2.1 Synthesis Algorithm

```
1:  procedure SYNTHESIZE(𝒮, Λ, ℰ)
2:      Input: Signature 𝒮 of method to synthesize,
3:             components Λ, and tests ℰ
4:      Output: Synthesized program or ⊥ for failure
5:      (𝒩, M*) := CONSTRUCTPETRI(𝒮, Λ)
6:      while true do
7:          π := GETNEXTPATH(𝒩, M*)
8:          (Σ, φ) := SKETCHGEN(π)
9:          for all σ ∈ MODELS(φ) do
10:             if RUNTESTS(Σ[σ], ℰ) then
11:                 return Σ[σ]
12:     return ⊥
```

from Figure 2.6 indicate the initial marking M_0 of $\mathcal{N}$. In particular, because the desired `rotate` method takes arguments of type `Area`, `Point2D`, and `double`, the initial marking assigns one token to each of these types. In addition, M_0 also assigns a single token to the special type `void`. In contrast, $M_0[\texttt{Shape}] = 0$ because `rotate` does not take any arguments of type `Shape`.

The target marking M^* of the Petri net is determined by the return type of $\mathcal{S}$. In our example, $M^*[\texttt{Area}] = 1$ because the return value of `rotate` is of type `Area`. However, for all other types τ (except for `void`), we require $M^*[\tau]$ to be 0, because this value effectively enforces that the synthesized implementation should not generate unused values. For instance, the target marking for the `rotate` example assigns `Point2D` to 0, thereby enforcing that the implementation uses argument `pt` and does not generate any other unused variables of type `Point2D`.

Reachability analysis. After constructing a Petri net $\mathcal{N}$ that models the relationships between components in Λ, we next perform reachability analysis to lazily find $\mathcal{N}$'s accepting runs (line 7 in Algorithm 3.1). For instance, an accepting run r for Figure 2.6 consists of the following sequence of transitions:

κ_D, `getX`, `getY`, `new AffineTransform`,
κ_T, `setToRotation`, `createTransformedArea`

Another accepting run r' can be obtained by replacing the transition `createTransformedArea` by `invert`. Observe that κ_D, `getX`, `getY` is *not* an accepting run because the marking obtained after this run assigns 3 tokens to `double`.

Sketch generation. Each accepting run of the Petri net $\mathcal{N}$ corresponds to a possible sequence of method calls with unknown arguments. Hence, the SketchGen procedure used in line 8 of Algorithm 3.1 converts each reachable path π to a program sketch Σ which is then used to resolve unknown arguments. For example, consider the accepting run r of $\mathcal{N}$ that we considered earlier. This run r corresponds to the following code sketch:

```
x = #1.getX(); y = #2.getY();
t = new AffineTransform();
#3.setToRotation(#4, #5, #6);
a = #7.createTransformedArea(#8);
return #9;
```

In other words, we can convert an accepting run r to a program sketch Σ by ignoring the κ transitions and passing unknown arguments (denoted as

#i) to each component. Furthermore, our construction guarantees that it is always possible to complete sketch Σ in a way that type-checks and satisfies certain well-formedness requirements. However, there may be multiple ways to instantiate the holes in Σ. For instance, we must assign #1 and #2 to `pt`, but we can assign #4 to either `angle`, `x`, or `y`, because the only requirement is that #4 is of type `double`.

Sketch completion. Similar to other sketching-based techniques (e.g., [101]), our technique uses a SAT solver to find possible completions of the generated program sketch. For this purpose, the SketchGen procedure generates a propositional formula ϕ that encodes various semantic requirements on the generated program, including being well-typed, not containing unused variables, and having all holes filled. Specifically, our encoding introduces Boolean variables of the form $h_v^{\#i}$, which encode that hole #i is filled with program variable v. For example, for hole #4, our encoding generates the following constraint:

$$h_{angle}^{\#4} + h_x^{\#4} + h_y^{\#4} = 1.$$

This formula stipulates that hole #4 must be filled with exactly one of `angle`, `x`, or `y` because those are the only program variables of type `double`. In addition, our encoding stipulates that each program variable must be used *at least once*. For instance, for variable `angle`, we generate the following constraint:

$$h_{angle}^{\#4} + h_{angle}^{\#5} + h_{angle}^{\#6} \geq 1.$$

This formula expresses that at least one of the holes #4, #5 and #6 must be instantiated with `angle`, because those are the only holes of type `double`.

After generating such a pseudo-boolean formula, we transform these constraints to CNF and use a SAT solver to find an assignment to each variable. For our running example, the following assignment σ is a model:

$$h_{pt}^{\#1} \wedge h_{pt}^{\#2} \wedge h_{t}^{\#3} \wedge h_{angle}^{\#4} \wedge \neg h_{x}^{\#4} \wedge \neg h_{y}^{\#4} \wedge \neg h_{angle}^{\#5} \wedge h_{x}^{\#5} \wedge \neg h_{y}^{\#5} \wedge \neg h_{angle}^{\#6} \wedge \neg h_{x}^{\#6} \wedge h_{y}^{\#6} \wedge h_{obj}^{\#7} \wedge h_{t}^{\#8} \wedge \neg h_{obj}^{\#9} \wedge h_{a}^{\#9}$$

Observe that σ corresponds to instantiating holes $\#1 - \#9$ in our code sketch with variables `pt`, `pt`, `t`, `angle`, `x`, `y`, `obj`, `t`, and `a`, respectively.

Validation and backtracking. Once we generate a complete program P, we then compile it and run P on the test cases provided by the user (line 10 in Algorithm 3.1). If all tests pass, we return P as a solution to the synthesis problem. If at least one test case fails, our algorithm backtracks and finds another satisfying assignment σ' to ϕ (if one exists) and generates a different completion of sketch Σ. If we have already considered all possible ways to fill the holes in Σ, our algorithm backtracks by finding a different accepting run of the Petri net $\mathcal{N}$ and generating a different sketch.

Discussion of design choices. A key design decision underlying our algorithm is to decompose the synthesis algorithm into two phases, namely *sketch generation* and *sketch completion*. In particular, an accepting run of the Petri net corresponds to a sequence of method calls, but there are, in general, multiple possible ways of choosing which variables to pass as arguments. We believe

this decomposition between sketch generation and completion is beneficial because it allows us to perform reachability analysis on a more compact graph representation. We have considered an alternative Petri-net representation in which nodes represent parameters and return values instead of types. Under this representation, an accepting run of the Petri net can be directly translated into a code snippet rather than a sketch. However, because the corresponding Petri net is much larger, we found that the reachability problem becomes much harder, thereby making the algorithm less scalable.

2.5 Petri-Net Construction

We now explain in more detail how our algorithm constructs a Petri net $\mathcal{N}$ from type signatures of components. In the remainder of this paper, we assume a first-order language of type signatures with classes and built-in primitive types (`string`, `int`, etc.).[2] Given library components Λ and a desired method signature $\mathcal{S}$, the algorithm constructs $\mathcal{N} = (P, T, E, W, M_0)$ and a target marking M^* as follows:

- Places P correspond to types used in Λ.
- Transitions T represent methods in Λ. In addition, for every type $\tau \in P$, there is a special transition called κ_τ.

[2] As described in Section 2.8, our approach also handles polymorphism, but using monomorphic instantiation.

- Arc (τ, f) is in E and $W[(\tau, f)] = k$ if component $f \in \Lambda$ takes k inputs of type τ.
- Arc (f, τ) is in E and $W[(f, \tau)] = 1$ if f's return type is τ for some component $f \in \Lambda$.
- Arcs (τ, κ_τ) and (κ_τ, τ) are both in E. Furthermore, $W[(\tau, \kappa_\tau)] = 1$ and $W[(\kappa_\tau, \tau)] = 2$.
- $M_0[\texttt{void}] = 1$ and $M_0[\tau] = k$ if S has k inputs of type τ.
- If the return type of S is τ, then $M^*[\tau] = 1$, $M^*[\texttt{void}] \geq 0$ and $M^*[\tau'] = 0$ for all other types τ'.[3]

At a high level, the Petri-net construction outlined above views types as resources. In particular, a transition associated with component $f \in \Lambda$ "consumes" its input types and produces a token at its output type. Hence, if the desired signature $\mathcal{S}$ has type $(\tau_1 \times \ldots \times \tau_n) \to \tau$, our goal is to produce a token at place τ by consuming the incoming tokens at places $\tau_1, \ldots, \tau_n$.

While this resource analogy fits very well with linear types, conventional types do not exactly behave as resources: In particular, invoking a component $f \in \Lambda$ on input x does not actually "consume" x; indeed, in a Java program, x can be used again. For this reason, the Petri-net construction outlined above introduces special transitions κ_τ (called *clone transitions*) that effectively allow

[3]If the return type of S is `void`, then $M^*[\texttt{void}] \geq 0$.

us to "duplicate" objects of type τ. Intuitively, the number of clone transitions taken in a given run indicates the total number of times variables will be reused in the synthesized program.[4]

To illustrate the necessity of clone transitions, consider our motivating example from Section 2.2. Here, to synthesize the implementation of `rotate`, we must retrieve the x and y coordinates of point `pt`. However, because we initially only have one token at `Point2D`, we can only call `getX` or `getY`, but not both. By invoking the clone transition κ_D, we can generate two resources of type `Point2D`, allowing us to invoke both `getX` and `getY` on parameter `pt`.

Another interesting aspect of our construction is the choice of target marking M^*. First, observe that M^* assigns 0 tokens to all places other than `void` and the return type of $\mathcal{S}$. Intuitively, this requirement dictates that the synthesized method should use all of its inputs as well as any intermediate values that are produced. This property is desirable because a method implementation that takes x as an input but does not use x is unlikely to be correct. Furthermore, a method that produces unused variables necessarily performs redundant work and can be replaced by a simpler implementation.[5]

[4]Our use of clone transitions is somewhat related to the use of read arcs in the Petri-net literature [118]. A read arc is a transition that does not consume tokens when fired. An alternative to having clone transitions is to use read arcs; however, this design choice would require us to use a different target marking that does not enforce the property that all inputs must be used.

[5]There are some methods, such as the `add` method of collections, that return a Boolean value that is often ignored. For such functions, we also consider a variant of the method that returns `void`.

2.6 Sketch Synthesis via Petri-Net Reachability

Given a Petri net $\mathcal{N}$ with target marking M^*, we need to answer the following questions to generate a suitable code sketch:

(1) Is $M^* \in \mathcal{R}(\mathcal{N})$? If the answer to this question is negative, we know that it is not possible to synthesize well-typed code using the components we have available.

(2) If $M^* \in \mathcal{R}(\mathcal{N})$, to synthesize candidate program sketches, we must identify exactly those runs of $\mathcal{N}$ that end in M^*.

To answer these questions, we must overcome two difficulties: First, because our Petri nets are not k-safe, the state space $\mathcal{R}(\mathcal{N})$ is unbounded. While there are existing methods for answering question (1) for unsafe Petri nets [62, 32], they cannot be used for answering question (2). Second, because the number of available components may be very large, we must develop effective heuristics for pruning the search space. In the rest of this section, we describe a practical algorithm for finding reachable paths for the class of Petri nets described in Section 2.5.

At a high level, there are three key insights underlying our reachability algorithm. The first insight is that we can bound the search space without losing completeness in our context. That is, even though $\mathcal{R}(\mathcal{N})$ is unbounded, exploring a subset $\mathcal{R}^*(\mathcal{N})$ of $\mathcal{R}(\mathcal{N})$ is sufficient for identifying *all* accepting runs of $\mathcal{N}$ (see Section 2.6.2). The second key insight is to use an over-approximation

$\alpha(\mathcal{N})$ of $\mathcal{N}$ to avoid exploring states that are irrelevant for reaching the target configuration M^* (see Section 2.6.3). Finally, rather than explicitly constructing $\mathcal{R}^*(\mathcal{N})$, we encode it symbolically and lazily enumerate the "most-promising" accepting runs of $\mathcal{N}$ by solving an optimization problem (see Section 2.6.4).

2.6.1 Basic Reachability Algorithm

Our algorithm for constructing the reachability graph $\mathcal{R}^*(\mathcal{N})$ is presented as pseudo-code in Algorithm 2.2. We first consider a basic version of the algorithm without lines 12–15, which is roughly equivalent to the standard algorithm for constructing $\mathcal{R}(\mathcal{N})$. The additional lines 12–15 correspond to our customization, and allow us to construct $\mathcal{R}^*(\mathcal{N})$ instead of $\mathcal{R}(\mathcal{N})$.

The procedure ReachGraph shown in Algorithm 2.2 takes as input a Petri net $\mathcal{N}$ with initial marking M_0 and the return type τ of the method we would like to synthesize, and returns a *reachability graph* $\mathcal{R}^*$. The nodes of $\mathcal{R}^*$ correspond to markings of $\mathcal{N}$, and a (directed) edge $\langle M, T, M' \rangle$ indicates that we can reach marking M' from M by firing transition T of $\mathcal{N}$. We denote nodes of $\mathcal{R}^*$ using labels of the form $\langle k_1, \ldots, k_n \rangle$, which indicates that there are k_i tokens at place P_i. For example, the marking of the Petri net from Figure 2.4 corresponds to the node label $\langle 2, 0, 0 \rangle$, whereas the marking from Figure 2.5 is given by $\langle 1, 1, 0 \rangle$.

The loop in lines 7–19 of Algorithm 2.2 iteratively constructs $\mathcal{R}^*$ starting from initial marking M_0. In particular, the worklist Φ contains all reachable markings that have not yet been processed. Initially, the only reachable marking

Algorithm 2.2 Algorithm to construct reachability graph

```
 1: procedure ReachGraph(𝒩, τ)
 2:     Input: Petri net 𝒩, desired output type τ
 3:     Output: Reachability graph ℛ*
 4:     assume 𝒩 = (P, T, E, W, M_0)
 5:     ℛ* := ({M_0}, ∅, M_0)                          ▷ Initialize
 6:     Φ := {M_0}                                     ▷ Initialize worklist Φ
 7:     while Φ ≠ ∅ do
 8:         choose M ∈ Φ                               ▷ Process next in Φ
 9:         Φ := Φ − {M}
10:         for all T ∈ enabled(M) do
11:             (M', p) := fire(M, T)                  ▷ Add successors
12:             if ∀e ∈ out(p). M'[p] > W[e] + 1 then
13:                 continue
14:             if ¬ PathExists(p, τ, α(𝒩)) then
15:                 continue
16:             if M' ∉ Nodes(ℛ*) then
17:                 Nodes(ℛ*).insert(M')
18:                 Φ := Φ ∪ {M'}
19:             Edges(ℛ*).insert(⟨M, T, M'⟩)
20:     return ℛ*
```

is M_0; hence we initialize Φ to the singleton set $\{M_0\}$ at line 6. In each iteration of the loop, we compute the successor states of some marking M in Φ by firing its enabled transitions. Specifically, the procedure fire used at line 11 takes a marking M and a transition T and returns the resulting marking M', as well as the output place p of transition T.[6] Now, ignoring lines 12–15, we add the edge $\langle M, T, M' \rangle$ to our reachability graph $\mathcal{R}^*$ and insert M' into the worklist if

[6] In our context, each transition has exactly one outgoing edge because every component has exactly one return type.

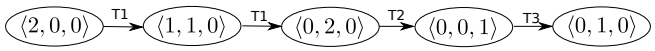


Figure 2.7. Reachability graph for Petri net from Figure 2.4

it has not already been processed.

Example 2.3. *Figure 2.7 shows the reachability graph for the Petri net from Figure 2.4. Observe that feasible runs of* $\mathcal{N}$ *correspond to paths starting with* M_0 *in the reachability graph. Hence, using the reachability graph, we immediately see that* T_1, T_1, T_2 *is a feasible run, but* T_1, T_2, T_3 *is not.*

2.6.2 Ensuring Termination

As mentioned earlier, the construction outlined in Section 2.5 results in Petri nets that are not k-safe for any k. In particular, while the clone transitions κ_τ are necessary for synthesizing code that reuses the same variable multiple times, they also cause us to accumulate arbitrarily many tokens at a given place. For example, we can obtain an unbounded number of tokens at place `Point2D` of Figure 2.6 by taking the clone transition κ_D as many times as we want. As a result, the size of the reachability graph is unbounded, meaning that the basic reachability algorithm from Section 2.6.1 will not terminate.

Fortunately, it turns out that we can bound the size of the reachable state space without losing completeness. In particular, when constructing the reachability graph for Petri net $\mathcal{N}$, we can safely ignore markings that assign more than $k + 1$ tokens to a place p, where k denotes the maximum weight

of any outgoing edge of p.[7] To see why we can ignore such markings, observe that no transition in $\mathcal{N}$ can be disabled due to p as long as we have at least k tokens at p. Furthermore, no matter what transition we take from the current marking, p will have at least 1 remaining token. Because our Petri nets contain clone transitions for every place, we can always produce k tokens at p by taking the clone transition sufficiently many times, as long as we have at least 1 token at p.

To formalize this intuition, let "$paths_{[M_0,M^*]}(G)$" denote the set of transition sequences in some reachability graph G that start at initial marking M_0, end at target M^*, and ignore all clone transitions. We can now state the following theorem:[8]

Theorem 1. *Let $\mathcal{R}(\mathcal{N})$ be the reachability graph constructed by the basic algorithm of Section 2.6.1, and let $\mathcal{R}^*(\mathcal{N})$ be the reachability graph constructed by employing lines 12–15 of Algorithm 2.2. If $p \in paths_{[M_0,M^*]}(\mathcal{R}(\mathcal{N}))$, then $p \in paths_{[M_0,M^*]}(\mathcal{R}^*(\mathcal{N}))$.*

Effectively, this theorem states we do not "lose" any valid code sketches by considering the paths of $\mathcal{R}^*(\mathcal{N})$ instead of $\mathcal{R}(\mathcal{N})$. Furthermore, because the size of $\mathcal{R}^*(\mathcal{N})$ is bounded by n^{k+1} where n is the number of places and k is the maximum edge weight in $\mathcal{N}$, Algorithm 2.2 is guaranteed to terminate.

[7] For simplicity, we assume that the number of initial tokens at place p is less than or equal to $k+1$. If this assumption is violated, the upper bound is given by the maximum of $k+1$ and the number of initial tokens.

[8] Proofs of all theorems are given in the extended version of the paper [28].

However, because places in $\mathcal{N}$ correspond to classes defined by a library, the reachability graph can still be very large. In the next subsection, we describe a pruning strategy to further reduce the size of the reachability graph.

2.6.3 Pruning using Graph Reachability

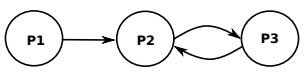
Another key idea of our algorithm is to use standard graph reachability to overapproximate Petri-net reachability. In particular, consider a place τ' in the Petri net that is not backwards reachable from our target type τ. Because there is no path from τ' to τ in $\mathcal{N}$, it is unnecessary to consider markings where τ' contains a non-zero number of tokens. Line 14 in Algorithm 2.2 exploits this observation to prune redundant nodes of $\mathcal{R}(\mathcal{N})$.

To make this discussion more precise, let us define $\alpha(\mathcal{N})$ to be the graph *induced by* Petri net $\mathcal{N}$ as follows:

Definition 2.2. (Induced graph) *Let $\mathcal{N} = (P, T, E, W, M_0)$ be a Petri net. The graph induced by $\mathcal{N}$, denoted $\alpha(\mathcal{N})$, is a directed graph (V, E') where $V = P$ and $(P, P') \in E'$ iff there is a transition $f \in T$ such that $(P, f) \in E$ and $(f, P') \in E$.*

In other words, $\alpha(\mathcal{N})$ includes an edge between two places P, P' if it is possible to reach P' from P by firing a single transition.

Example 2.4. *The graph induced by the Petri net of Figure 2.4 is shown below:*



Theorem 2. *Let $\mathcal{N}$ be a Petri net with no path from τ' to τ in $\alpha(\mathcal{N})$. Let M^* be the target marking that assigns one token to target type τ, and let M be a marking such that $M(\tau') > 0$. Then, there is no path from M to M^* in $\mathcal{R}(\mathcal{N})$.*

According to this theorem, if a marking M assigns a non-zero value to any place τ' that is not backwards-reachable from τ in $\alpha(\mathcal{N})$, then there is no path from M to M^* in $\mathcal{R}(\mathcal{N})$. Hence, we can prune such a marking M without affecting completeness. Line 14 in Algorithm 2.2 takes advantage of this fact by only adding M' to $\mathcal{R}^*(\mathcal{N})$ if p is backwards reachable from τ.

2.6.4 Symbolic Encoding using ILP

So far, our algorithm explicitly constructs $\mathcal{R}^*(\mathcal{N})$ and enumerates all paths of $\mathcal{R}^*(\mathcal{N})$. However, because $\mathcal{R}^*(\mathcal{N})$ can have many accepting paths, this strategy is suboptimal. Instead, a better alternative is to encode this problem symbolically and *lazily* generate accepting runs of $\mathcal{N}$ in order of increasing cost. Toward this goal, we formulate the problem of finding an accepting run of $\mathcal{N}$ as a 0-1 Integer Linear Programming (ILP) problem and obtain the "most-promising" path by minimizing a heuristic objective function.

Our lazy symbolic path-enumeration algorithm is presented in Algorithm 2.3. We consider accepting runs of $\mathcal{N}$ in increasing order of length, starting from the minimum bound k (line 6). In particular, if τ_i is one of the input types and τ is the desired output type, then any accepting run of $\mathcal{N}$ must contain at least as many transitions as the shortest path between τ_i and τ in

Algorithm 2.3 Lazy symbolic path enumeration

```
1:  procedure LazyPathGen(N, τ_1, ..., τ_n, τ)
2:      Input: Petri net N, input types τ_1, ..., τ_n,
3:             output type τ
4:      Output: An accepting run t of N if one exists
5:      π_i := ShortestPath(α(N), τ_i, τ)                    ▷ Lower bound
6:      k := max(length(π_1), ..., length(π_n))
7:      while true do
8:          φ := Encode(N, k)                                ▷ Unfolding of length k
9:          ψ := true
10:         while true do
11:             σ := Minimize(Σ_i c_i x_i, φ ∧ ψ)
12:             if σ = ⊥ then
13:                 break
14:             if Check(σ) then
15:                 return Trace(σ)
16:             ψ := ψ∧ Block(σ)
17:         k := k + 1
18:     return ⊥
```

$\alpha(\mathcal{N})$; hence, we do not need to look for accepting runs below this threshold.

Now, given a target length k, we symbolically encode the k-reachability problem of $\mathcal{N}$ as a propositional formula ϕ. In particular, formula ϕ from line 8 is satisfiable if and only if there exists an accepting run of $\mathcal{N}$ of length k. Our symbolic encoding is similar to previous SAT-based encodings of Petri nets [79, 73, 52], but we make use of the observations from Sections 2.6.2 and 2.6.3. While a full discussion of our symbolic encoding is beyond the scope of this paper, we refer the interested reader to the extended version of the paper [28].

The inner loop in lines 10–16 of Algorithm 2.3 lazily enumerates paths of length k in order of increasing cost, where the cost is determined by some heuristic evaluation function. To generate the "most-promising" path, we solve an ILP problem with objective function $\Sigma_i c_i x_i$ (line 11). Here, x_i is a variable that is assigned to 1 by our encoding if and only if component T_i is used in the accepting run and to 0 otherwise. The costs c_i used in the objective function reflect the likelihood of component T_i being used in the synthesized code—i.e., the smaller the c_i, the more likely it is that component T_i is useful. While there are many possible heuristics for assigning costs to components, our current implementation uses a similarity metric between the name of the desired method and the documentation and name of each library component.[9] Going back to our running example from Section 2.2, this methodology assigns a lower cost to a component called `setToRotate` compared to another component called `invert` because the former component is likely to be more "similar" to the desired `rotate` method.

Once we obtain a satisfying assignment σ of ϕ that minimizes our heuristic objective function, we ask an "oracle" to confirm or refute it (lines 14–15). In this context, the oracle completes the code sketch given by σ (see Section 2.7) and runs the test cases. If σ does not correspond to a satisfactory code sketch, we need to "block" this assignment in future iterations by adding a blocking clause ψ. In the simplest case, a blocking clause can be obtained

[9]We refer the interested reader to the extended version of the paper [28] for a more detailed discussion of our similarity metrics.

as the negation of σ; however, our algorithm generates a stronger blocking clause by performing a particular form of partial-order reduction [10, 81] on the current path p. In particular, if p contains two consecutive calls to methods f and g that cannot be called with the same arguments, then our algorithm also blocks variants of this path where calls to f and g have been re-ordered.

2.7 Code Synthesis from Paths

Given an accepting run r of the Petri net described in Section 2.5 and Section 2.6, to synthesize a suitable program from r, we still need to perform the following tasks:

(a) Use the transitions in r to create a code sketch Σ

(b) Fill the holes in Σ with program variables

Each transition in r corresponds to either an invocation of a method `foo` from an API or a special κ transition. When synthesizing code, we ignore clone transitions and only consider API calls. In particular, if some API method `foo` used in r has n input parameters, the code sketch for `foo`'s invocation looks like the following:

```
// if m is a virtual method
T_o out = #1.foo(#2, #3, #4, ..., #n+1)

// if m is a static method or constructor
T_o out = foo(#1, #2, #3, ..., #n)
```

In general, if trace r is of length l and contains k clone transitions, the corresponding synthesized program contains $l - k + 1$ lines, where the first

$l - k$ lines correspond to API calls and the last line is a return statement of the form `return#m` (when the program does not return `void`).

Now, given sketch Σ, we need to instantiate each hole with a program variable. To achieve this goal, we generate a propositional formula ϕ that encodes well-formedness requirements. In particular, our encoding introduces Boolean variables $h_v^{\#i}$ that are true when program variable v is used to fill hole `#i`. To ensure type compatibility, we only introduce Boolean variable $h_v^{\#i}$ if the type of program variable v matches the type of hole `#i`. Furthermore, because a program variable cannot be used before it is defined, we only introduce $h_v^{\#i}$ if v is a parameter or the result of an invocation that appears before hole `#i`.

While our construction of the Boolean variables guarantees that the holes will be filled in a type-compatible way, we still have to ensure that no hole remains empty and that all variables are used. Let V be the set of all program variables and H the set of all holes in Σ. Let $getV$ be a function that receives V and a hole h and returns $V' \subseteq V$, where V' corresponds to all program variables that can be placed in hole h. Similarly, let $getH$ be a function that receives H and a variable $v \in V$ and returns $H' \subseteq H$, where H' corresponds to all holes where v can be placed. Using these definitions, we generate a formula ϕ as follows:

(1) Each hole is filled with one program variable:

$$\forall_{\#i \in H} \forall_{v \in getV(V,\#i)} \sum h_v^{\#i} = 1$$

(2) Each program variable is used at least once:

$$\forall_{v \in V} \forall_{\#i \in getH(H,v)} \sum h_v^{\#i} \geq 1$$

Example 2.5. *Consider the code sketch in Section 2.4. From requirement (1), we generate the following constraints:*

$$h_{pt}^{\#1} = 1 \ ; \ h_{pt}^{\#2} = 1 \ ; \ h_t^{\#3} = 1 \ ; \ h_{angle}^{\#4} + h_x^{\#4} + h_y^{\#4} = 1$$
$$h_{angle}^{\#5} + h_x^{\#5} + h_y^{\#5} = 1 \ ; \ h_{angle}^{\#6} + h_x^{\#6} + h_y^{\#6} = 1$$
$$h_{obj}^{\#7} = 1; h_t^{\#8} = 1 \ ; \ h_{obj}^{\#9} + h_a^{\#9} = 1$$

Similarly, from requirement (2), we generate the constraints:

$$h_{pt}^{\#1} \geq 1 \ ; \ h_{pt}^{\#2} \geq 1 \ ; \ h_t^{\#3} \geq 1$$
$$h_{angle}^{\#4} + h_{angle}^{\#5} + h_{angle}^{\#6} \geq 1 \ ; \ h_x^{\#4} + h_x^{\#5} + h_x^{\#6} \geq 1$$
$$h_y^{\#4} + h_y^{\#5} + h_y^{\#6} \geq 1 \ ; \ h_{obj}^{\#7} + h_{obj}^{\#9} \geq 1 \ ; \ h_a^{\#9} \geq 1$$

Because each satisfying assignment σ to ϕ corresponds to a well-typed completion of sketch Σ, we can now run the user-provided test cases on $\Sigma[\sigma]$. If any test fails, we then obtain a different instantiation of the sketch by obtaining a model of $\phi \wedge \neg\sigma$ in the next iteration.

2.8 Implementation

We have implemented our synthesis algorithm as a new tool called SYPET, which consists of approximately 10,000 lines of Java code. SYPET uses the Sat4j [15] tool for solving SAT problem, and can be instantiated with any Java API (or combinations of APIs) to synthesize straight-line Java code.

Lib	ID	Description	Synthesis Time (s)	#Paths	#Progs	#Tests	#Comps	#Holes
apache math	1	Compute the pseudo-inverse of a matrix	6.78	255	509	1	3	4
	2	Compute the inner product between two vectors	0.25	1	1	1	3	5
	3	Determine the roots of a polynomial equation	0.64	7	13	1	3	5
	4	Compute the singular value decomposition of a matrix	0.16	1	1	1	3	4
	5	Invert a square matrix	0.63	16	31	1	3	4
	6	Solve a system of linear equations	28.25	790	1,605	1	6	8
	7	Compute the outer product between two vectors	2.12	14	48	1	4	6
	8	Predict a value from a sample by linear regression	2.56	25	51	2	5	5
	9	Compute the i^{th} eigenvalue of a matrix	164.60	3,197	7,636	2	6	8
geometry	10	Scale a rectangle by a given ratio	1.37	78	271	1	4	7
	11	Shear a rectangle and get its tight rectangular bounds	1.76	79	280	1	4	7
	12	Rotate a rectangle about the origin by the specified number of quadrants	0.32	9	21	1	4	6
	13	Rotate two dimensional geometry object by the specified angle about a point	2.01	67	226	2	5	8
	14	Perform a translation on a given rectangle	0.72	41	150	1	4	7
	15	Compute the intersection of a rectangle and the rectangular bounds of an ellipse	0.08	1	1	1	3	5
joda	16	Compute number of days since the specified date	4.55	78	156	2	3	4
	17	Compute the number of days between two dates considering timezone	174.16	774	4,736	3	4	6
	18	Determine if a given year is a leap year	35.32	306	613	3	4	5
	19	Return the day of a date string	0.74	1	1	2	3	5
	20	Find the number of days of a month in a date string	35.23	175	531	2	4	6
	21	Find the day of the week of a date string	47.27	126	376	2	4	6
	22	Compute age given date of birth	7.90	142	288	3	3	4
jsoup, dom, text	23	Compute the offset for a specified line in a document	0.31	3	5	1	3	5
	24	Get a paragraph element given its offset in the a document	1.14	33	65	1	4	6
	25	Obtain the title of a webpage specified by a URL	10.29	277	553	1	3	4
	26	Return doctype of XML document generated by string	0.87	9	17	1	6	7
	27	Generate an XML element from a string	0.89	26	51	1	6	7
	28	Read XML document from a file	0.11	1	1	1	3	4
	29	Generate an XML from file and query it using XPath	16.33	20	44	1	7	10
	30	Read XML document from a file and get the value of root attribute specified by a string	0.29	3	5	1	5	7

Figure 2.8. Summary of experimental results

Soot [115] is used to parse the .jar files of the libraries and extract the signatures of classes and methods, which will be converted to places and transitions in the Petri-net, respectively.

Because many Java libraries use parametric polymorphism, our implementation also supports generic types. Our handling of polymorphism is similar to template instantiation in C++. For instance, given a polymorphic type of the form `Foo` <?`extendsA` > and subclasses `B`, `C` of `A`, we generate three different copies of type `Foo`, namely `FooA`, `FooB`, and `FooC`, each of which corresponds to a different place in the Petri net. We also handle polymorphic methods in a similar way and create different transitions for each instantiation of a polymorphic API component.

As mentioned in Section 2.6, SyPet uses a symbolic encoding of the Petri-net-reachability problem, but our implementation differs from Algorithm 2.3 in one small way. Given a Petri net $\mathcal{N}$, recall that Algorithm 2.3 explores all reachable paths of length k before moving on to paths of length $k+1$. While this approach simplifies our presentation, it is not a very good implementation strategy: Because there can be *many* paths of length k , we have found that a better strategy is to explore different path lengths in a round-robin fashion. In particular, our search strategy is parametrized by two integers n, m: Given a starting path length k, we first explore m paths of size k, and then move on to paths of length $k+1$. After exploring m paths each of length $k, \ldots, k+n$, we go back to exploring paths of length k. In our current implementation, we use the values 2 and 100 for n and m, respectively.

2.9 Evaluation

To evaluate SyPet, we performed experiments that were designed to answer the following questions:

1. How well does SyPet perform on component-based synthesis tasks that involve Java APIs?

2. How many test cases does the user typically need to supply for SyPet to succeed?

3. How complex are the programs synthesized by SyPet?

4. How does SyPet's success rate compare with other tools for component-based synthesis?

To answer these questions, we collected six widely-used Java APIs: a math library (`apache.commons.math`), a geometry library (`java.awt.geom`), a time/date library (`joda − time`), and text and XML-related libraries (`jsoup`, `w3c.dom` and `javax.xml`). In addition to being widely used, these libraries are reasonably large, containing 50–1215 classes and 751–9578 methods. The average number of classes and components in each library is 528 and 4721, respectively.

For each of these APIs, we collected a set of programming tasks that require non-trivial interaction between different classes. Our programming tasks come from two sources—namely, online forums like *stackoverflow* and

existing Github repositories. For the former category, we manually curated common questions that programmers typically ask about the relevant API. For the latter category, we wrote a script to crawl over Github projects and filter straight-line methods that use one of the aforementioned APIs. A brief summary of each programming task is provided under the "Description" column in Figure 2.8.

2.9.1 SyPet Performance

Setup. To evaluate SyPet on these programming tasks, we provided a signature of the desired method as well as one or more test cases. We also specify which libraries are used for each programming task, e.g., `joda.time`, `apache.commons.math`, etc. However, it is easy to configure the tool to use any set of libraries. For the benchmarks taken from Github, we used the existing method signature (and test cases if available). For most *stackoverflow* benchmarks, method signature and test cases were not available in the forum discussion, so we wrote them ourselves. For all benchmarks, we initially provided a *single* test case and used SyPet to synthesize an implementation that works on that test case. We then manually inspected the synthesized code and provided an additional test case if the synthesized code did not perform the desired functionality. We then repeated this process until the code produced by SyPet met our expectations.

The results of our evaluation are summarized in Figure 2.8 (For more detailed results, please refer to the extended version of the paper [28]). All

experiments are conducted using Oracle HotSpot JVM 1.7.0_75 on an Intel Xeon(R) computer with an E5-2640 v3 CPU and 32G of memory, running Ubuntu 14.04.

Performance and statistics. As shown in the "Synthesis Time" column of Figure 2.8, SyPet can successfully synthesize all benchmarks in an average of 2.33 seconds.[10] Note that the synthesis time neither includes compilation time nor the overhead of parsing the .jar files with Soot. Compilation has an average overhead of 53% on the running time and Soot takes an average of 7.00 seconds to parse the Java libraries. The "#Paths" column indicates the total number of code sketches generated by our tool. Note that this number is equivalent to the number of explored paths (accepting runs) of the Petri net. On average, SyPet explores 29 different code sketches before it identifies the correct sequence of method calls. Furthermore, each iteration of the tool is quite fast; SyPet finds an accepting run of the Petri net in 0.08 seconds on average. The column labeled "#Progs" indicates the total number of programs generated by SyPet before finding the correct program. On average, SyPet explores 61 programs before generating an implementation that performs the desired functionality.

While SyPet synthesizes 73% of the benchmarks in < 10 seconds and 93% in < 60 seconds, a few benchmarks (e.g., 9 and 17) take longer. We have

[10]If there are multiple rounds of user interaction to create additional test cases, we report statistics for the last one. We calculate averages using geometric mean.

manually inspected these outliers and found that the user-provided signatures for these examples match the signature of many API components. Hence, SyPet ends up exploring hundreds of code sketches before it synthesizes the intended one.

Usability. In addition to successfully synthesizing the desired code in a reasonable amount of time, we also see that SyPet does not require many test cases from the user. In particular, as shown under the "#Tests" column in Figure 2.8, SyPet requires 1 test case on average, with the maximum number of test cases being 3.

Synthesized programs. The "#Comps" and "#Holes" columns in Figure 2.8 provide information about the synthesized programs. In particular, "#Comps" reports the number of components in the code sketch (in terms of the length of the accepting run), and "#Holes" indicates the number of holes. The average synthesized program contains 4 components and 6 holes. These statistics reinforce our earlier claim that SyPet combines the practicality of API completion tools with the power of synthesis tools: While programs synthesized by SyPet are moderately sized, straight-line code fragments, SyPet can handle two orders of magnitude more components than previous synthesis tools [57, 44, 43, 83]. On the other hand, while API-completion tools [68, 47, 49, 34] can handle thousands of components, they can typically

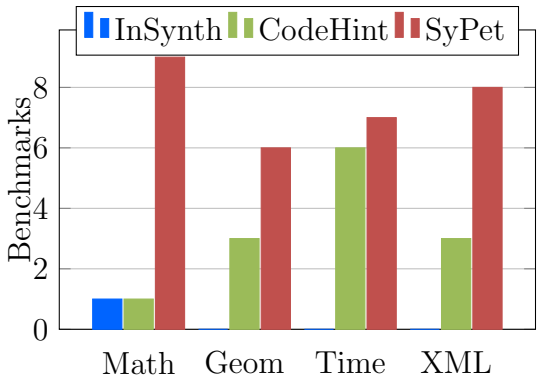


Figure 2.9. Comparison with other tools

only suggest very small (single-line) code snippets.[11]

2.9.2 Comparison with Other Tools

To validate our claim that SyPet compares favorably with existing synthesis tools that do not require logical specifications, we also compare SyPet with CodeHint and InSynth. CodeHint is a state-of-the-art type-based synthesis tool, and, similar to SyPet, it takes as input a method signature and test case. In contrast, InSynth is a type-directed API-completion tool that can synthesize expressions of a given type.

The results of our comparison are provided in Figure 8, which shows how many benchmarks were synthesized by each tool within a 30-minute time limit. For both CodeHint and InSynth, we consider the synthesis task to be successful if the correct implementation is among any of the suggested code snippets. While SyPet is able to synthesize all 30 benchmarks, CodeHint

[11]For instance, 94% of the benchmarks used in evaluating InSynth [49, 47] (a state-of-the-art completion tool) involve a single API call.

synthesizes 13 benchmarks and InSynth can synthesize just one of them.

Because InSynth is mainly intended to be used as a single-line code-completion tool, we also performed a second (simpler) experiment using InSynth. Specifically, given the full implementation of each benchmark except a single line of code, we tried to use InSynth to complete the right-hand-side of each assignment one at a time. We considered InSynth to be successful if it was able to complete the right-hand-side of all assignments used in the implementation. However, even for this easier task, InSynth was only able to solve 14 out of the 30 benchmarks.

2.10 Summary

In this chapter, we have proposed a new type-directed approach to component-based program synthesis. Our approach constructs a Petri net from the signatures of API components and generates a code sketch by identifying accepting runs of the resulting Petri net. The code sketches are then completed using SAT-based reasoning and tested on the user-provided examples.

We evaluated SyPet on a collection of programming tasks involving six widely-used APIs. Our evaluation shows that SyPet can synthesize the desired program in a practical manner using few test cases. Our tool is publicly available [5] and can be easily used by programmers to synthesize complex APIs from test cases.

Chapter 3

Morpheus [1]

Since SyPet uses primitive types as the coarse-grained specifications to prune ill-typed candidates, it will degrade to naive enumerative search in some domains which typically consume and produce the same primitive types, such as string manipulation and data wrangling. To address this disadvantage in SyPet, this chapter presents a novel component-based synthesis algorithm that marries the power of type-directed search with lightweight SMT-based deduction and partial evaluation. Given a set of components together with their *over-approximate* first-order specifications, our method first generates a *program sketch* over a subset of the components and checks its feasibility using an SMT solver. Since a program sketch typically represents *many* concrete programs, the use of SMT-based deduction greatly increases the scalability of the algorithm. Once a feasible program sketch is found, our algorithm completes the sketch in a bottom-up fashion, using partial evaluation to further increase the power of deduction for rejecting partially-filled program sketches. We apply the proposed synthesis methodology for automating a large class of data preparation tasks that commonly arise in data science. We have evaluated

[1] Parts of this chapter have appeared in [26].

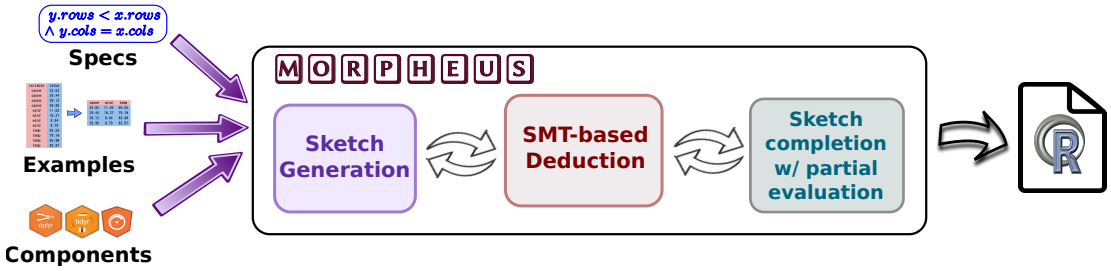


Figure 3.1. Overview of our approach

our synthesis algorithm on dozens of data wrangling and consolidation tasks obtained from on-line forums, and we show that our approach can automatically solve a large class of problems encountered by R users.

3.1 Overview

A particularly interesting version of automating programming concerns the synthesis of programs that manipulate *tabular data*. Such programs are especially important in an era where data analytics has gained enormous popularity across a wide range of disciplines, ranging from biology to business to the social sciences. Since raw data is rarely in a form that is immediately amenable to an analytics or visualization task, data scientists typically spend over 80% of their time performing tedious data preparation tasks [20]. Such tasks include consolidating multiple data sources into a single table, reshaping data from one format into another, or adding new rows or columns to an existing table.

While data preparation tasks would seem to be natural targets for synthesis, many such tasks are too complex to be handled by existing techniques.

If written in a low-level language, programs implementing these tasks would be simply too large to be discovered by combinatorial search. One way around this difficulty is to describe the relevant computations using a set of predefined library functions, or *components*, and then synthesize programs that use these high-level primitives. Another advantage of such a *component-based synthesis* approach is its flexibility: Since the reasoning of the synthesizer is not hard-wired to a fixed set of DSL constructs, the underlying algorithm can generate more complex programs as new libraries emerge or as more components are added to its knowledge base.

Unfortunately, a key challenge in developing such a general component-based synthesis algorithm for automating data preparation tasks is scalability: Since many languages (e.g., R) provide a large number of components that are typically used in data preparation, the size of the search space that must be explored by the underlying synthesis algorithm can be very large. Due to this difficulty, prior techniques for automating table transformations (e.g., [51, 128]) focus on narrowly-defined DSLs, such as subsets of the Excel macro language [51] or fragments of SQL [128]. Unfortunately, many common data preparation tasks (e.g., those that involve reshaping tables or require performing nested table joins) fall outside the scope of these previous approaches.

In this chapter, we propose a general component-based synthesis algorithm for automating a large class of data preparation tasks. Specifically, our synthesis algorithm is parametrized over a set of components, which can include both higher-order and first-order combinators. The set of components

used by the synthesizer can be customized by the user or extended over time as new libraries emerge.

In order to address the scalability challenges that arise from our more general formulation of the problem, we propose a new synthesis algorithm that combines type-directed enumerative search with *lightweight SMT-based deduction* and *partial evaluation*. In our formulation of the synthesis problem, each component $\mathcal{C}$ is equipped with a logical, incomplete specification that over-approximates $\mathcal{C}$'s behavior. These specifications are utilized by the synthesizer to perform lightweight SMT-based reasoning, with the goal of rejecting infeasible partial programs. Furthermore, specifications are provided *per component*, so they can be re-used across arbitrarily many synthesis tasks. Since our technique does not depend on hard-coded component-specific reasoning, our approach significantly generalizes prior uses of deduction in example-guided synthesis (e.g., [30]).

Figure 3.1 shows a schematic illustration of our synthesis algorithm, implemented in a tool called Morpheus. To facilitate effective use of SMT-based deduction, our algorithm decomposes the synthesis task into two separate *sketch generation* and *sketch completion* phases. In particular, a sketch specifies the top-level combinators used in the program, but not their corresponding arguments. Our algorithm uses type-directed enumerative search to lazily explore the space of all possible program sketches and infers a specification of each candidate sketch using the specifications of the underlying components. Hence, once we have a candidate sketch $\mathcal{S}$, we can use an SMT solver to test

whether $\mathcal{S}$ is consistent with the provided input-output examples. Because a program sketch typically represents *many* concrete programs, the rejection of program sketches using SMT-based reasoning dramatically improves the scalability of the synthesis algorithm.

Once our algorithm finds a feasible program sketch, it then tries to complete it in a bottom-up, type-directed way. In particular, the synthesizer *evaluates* sub-terms of the partial program $\mathcal{P}$ to infer a more precise specification for $\mathcal{P}$ and again uses SMT-based reasoning with the goal of refuting the partially-completed sketch. Hence, the use of partial evaluation further improves the scalability of the synthesis algorithm by allowing us to refute partial programs obtained during sketch completion.

While the core ideas underlying our algorithm are generally applicable to any component-based synthesizer, we have used these ideas to automate table consolidation and transformation tasks that commonly arise in data science. Specifically, our implementation, Morpheus, takes as input a set of source data frames in R, as well as the target data frame that should be generated using the synthesized program. Additionally, the user can also provide a set of components (i.e., library methods), optionally with their corresponding first-order specifications. However, since our implementation already comes with a built-in set of components that are commonly used in data preparation, the user does not need to provide any additional components but can do so if she so desires. Using the ideas outlined above, Morpheus then automatically synthesizes an R program that can now be applied to other data frames.

To evaluate our techniques, we have collected a suite of data preparation tasks for the R programming language, drawn from discussions among R users in on-line forums such as Stackoverflow. The "components" in our evaluation are methods provided by two popular R libraries, namely `tidyr` and `dplyr`, for data tidying and manipulation. Our experiments show that MORPHEUS can successfully synthesize a diverse class of real-world data preparation programs. We also evaluate the performance of MORPHEUS using component specifications of different granularities and demonstrate that SMT-based deduction and partial evaluation are crucial for the scalability of our approach.

3.2 Motivating Examples

In this section, we illustrate the diversity of data preparation tasks using a few examples collected from Stackoverflow.

Example 3.1. *An R user has the data frame in Figure 3.2(a), but wants to transform it to the following format [2]:*

id	*A_2007*	*B_2007*	*A_2009*	*B_2009*
1	*5*	*10*	*5*	*17*
2	*3*	*50*	*6*	*17*

Even though the user is quite familiar with R libraries for data preparation, she is still not able to perform the desired task. Given this example, MORPHEUS *can automatically synthesize the following R program:*

```
df1=gather(input,var,val,id,A,B)
df2=unite(df1,yearvar,var,year)
df3=spread(df2,yearvar,val)
```

Observe that this example requires both reshaping the table and appending contents of some cells to column names.

Example 3.2. *Another R user has the data frame from Figure 3.2(b) and wants to compute, for each source location L, the number and percentage of flights that go to Seattle (SEA) from L [3]. In particular, the output should be as follows:*

origin	*n*	*prop*
EWR	*2*	*0.6666667*
JFK	*1*	*0.3333333*

MORPHEUS *can automatically synthesize the following R program to extract the desired information:*

```
df1=filter(input, dest == "SEA")
df2=summarize(group_by(df1, origin), n = n())
df3=mutate(df2, prop = n / sum(n))
```

Observe that this example involves selecting a subset of the data and performing some computation on that subset.

Example 3.3. *A data analyst has the following raw data about the position of vehicles for a driving simulator [4]:*

Table 1:

frame	*X1*	*X2*	*X3*
1	*0*	*0*	*0*
2	*10*	*15*	*0*
3	*15*	*10*	*0*

Table 2:

frame	*X1*	*X2*	*X3*
1	*0*	*0*	*0*
2	*14.53*	*12.57*	*0*
3	*13.90*	*14.65*	*0*

Here, Table 1 contains the unique identification number for each vehicle (e.g., 10, 15), with 0 indicating the absence of a vehicle. The column labeled "frame" in Table 1 measures the time step, and the columns "X1", "X2", "X3" track which vehicle is closer to the driver. For example, at frame 3, the vehicle with ID 15 is the closest to the driver. Table 2 has a similar structure as Table 1 but contains the speeds of the vehicles instead of their identification number. For example, at frame 3, the speed of the vehicle with ID 15 is 13.90 m/s. The data analyst wants to consolidate these two data frames into a new table with the following shape:

frame	*pos*	*carid*	*speed*
2	*X1*	*10*	*14.53*
3	*X2*	*10*	*14.65*
2	*X2*	*15*	*12.57*
3	*X1*	*15*	*13.90*

Despite looking into R libraries for data preparation, the analyst still cannot figure out how to perform this task and asks for help on Stackoverflow. Morpheus *can synthesize the following R program to automate this complex task:*

```
df1=gather(table1,pos,carid,X1,X2,X3)
df2=gather(table2,pos,speed,X1,X2,X3)
df3=inner_join(df1,df2)
df4=filter(df3,carid != 0)
df5=arrange(df4,carid,frame)
```

id	year	A	B
1	2007	5	10
2	2009	3	50
1	2007	5	17
2	2009	6	17

(a)

flight	origin	dest
11	EWR	SEA
725	JFK	BQN
495	JFK	SEA
461	LGA	ATL
1696	EWR	ORD
1670	EWR	SEA

(b)

Figure 3.2. (a) Data frame for Example 3.1; (b) for Example 3.2.

3.3 Problem Formulation

In order to precisely describe our synthesis problem, we first present some definitions that we use throughout this chapter.

Definition 3.1. *(***Table***)* *A* table T *is a tuple* (r, c, τ, ς) *where:*

- r, c *denote number of rows and columns respectively*
- $\tau : \{l_1 : \tau_1, \ldots, l_c : \tau_c\}$ *denotes the type of* T*. In particular, each* l_i *is the name of a column in* T *and* τ_i *denotes the type of the value stored in* T*. We assume that each* τ_i *is either* `num` *or* `string`*.*
- ς *is a mapping from each cell* $(i, j) \in ([0, r) \times [0, c))$ *to a value* v *stored in that cell*

Given a table $\mathsf{T} = (r, c, \tau, \varsigma)$, we write $\mathsf{T}.\mathsf{row}$ and $\mathsf{T}.\mathsf{col}$ to denote r and c respectively. We also write $\mathsf{T}_{i,j}$ as shorthand for $\varsigma(i, j)$ and $\mathit{type}(\mathsf{T})$ to represent τ. We refer to all record types $\{l_1 : \tau_1, \ldots, l_c : \tau_c\}$ as type `tbl`. In addition, tables with only one row are referred to as being of type `row`.

Definition 3.2. *(***Component***) A component* $\mathcal{X}$ *is a triple* (f, τ, ϕ) *where* f *is a string denoting* $\mathcal{X}$*'s name,* τ *is the type signature (see Figure 3.3), and* ϕ *is a first-order formula that specifies* $\mathcal{X}$*'s input-output behavior.*

Given a component $\mathcal{X} = (f, \tau, \phi)$, the specification ϕ is over the vocabulary $x_1, \ldots, x_n, y$, where x_i denotes $\mathcal{X}$'s i'th argument and y denotes $\mathcal{X}$'s return value. Note that specification ϕ does not need to *precisely* capture $\mathcal{X}$'s input-output behavior; it only needs to be an *over-approximation*. Thus, *true* is always a valid specification for any component.

With slight abuse of notation, we sometimes write $\mathcal{X}(\ldots)$ to mean $f(\ldots)$ whenever $\mathcal{X} = (f, \tau, \phi)$. Also, given a component $\mathcal{X}$ and arguments $c_1, \ldots, c_n$, we write $[\![\mathcal{X}(c_1, \ldots, c_n)]\!]$ to denote the result of evaluating $\mathcal{X}$ on arguments $c_1, \ldots, c_n$.

Definition 3.3. *(***Problem specification***)* *The* specification *for a synthesis problem is a pair* $(\mathcal{E}, \Lambda)$ *where:*

- $\mathcal{E}$ *is an input-output example* $(\vec{\mathsf{T}}_{\text{in}}, \mathsf{T}_{\text{out}})$ *such that* $\vec{\mathsf{T}}_{\text{in}}$ *denotes a list of input tables, and* T_{out} *is the output table,*
- $\Lambda = (\Lambda_{\mathsf{T}} \cup \Lambda_v)$ *is a set of components, where* $\Lambda_{\mathsf{T}}, \Lambda_v$ *denote* table transformers *and* value transformers *respectively. We assume that* Λ_{T} *includes higher-order functions, but* Λ_v *consists of first-order operators.*

Given an input-output example $\mathcal{E} = (\vec{\mathsf{T}}_{in}, \mathsf{T}_{out})$, we write $\mathcal{E}_{in}$, $\mathcal{E}_{out}$ to denote $\vec{\mathsf{T}}_{in}$, T_{out} respectively. Also, we classify components Λ into two disjoint

Cell type γ	:=	`num` | `string`
Primitive type β	:=	γ | `bool` | `cols`
Table type `tbl`	:=	$\{l_1 : \gamma_1, ..., l_n : \gamma_n\}$ (`row <:` `tbl`)
Type τ	:=	β | `tbl` | $\tau_1 \to \tau_2$ | $\tau_1 \times \tau_2$

Figure 3.3. Types used in components; `cols` represents a list of strings where each string is a column name in some table.

classes Λ_{T} and Λ_v, where Λ_{T} denotes *table transformer* components that take at least one table as an argument and return a table. Components of all other types are *value transformers* Λ_v. While table transformers can be higher-order combinators, value transformers are always first-order. In the rest of the chapter, we assume that table transformers only take tables and first-order functions (constructed using constants and components in Λ_v) as arguments.

Example 3.4. *Consider the selection operator σ from relational algebra, which takes a table and a predicate and returns a table. In our terminology, such a component is a* table transformer. *In contrast, an aggregate function such as* sum *that takes a list of values and returns their sum is a* value transformer. *Similarly, the boolean operator $\geq$ is also a value transformer.*

Definition 3.4. *(***Synthesis problem)** *Given specification $(\mathcal{E}, \Lambda)$ where $\mathcal{E} = (\vec{T}_{\text{in}}, T_{\text{out}})$, the synthesis problem is to infer a program $\lambda\vec{x}.e$ such that (a) e is a well-typed expression over components in Λ, and (b) $(\lambda\vec{x}.e)\,\vec{T}_{\text{in}} = T_{\text{out}}$.*

3.4 Hypotheses as Refinement Trees

Before we can describe our synthesis algorithm, we first introduce *hypotheses* that represent partial programs with unknown expressions (i.e.,

$$[\![(?_i : \tau)]\!]_\partial = ?_i \qquad [\![(?_i : \tau)@(x, \mathsf{T})]\!]_\partial = \mathsf{T} \qquad [\![(?_i : \tau)@t]\!]_\partial = t$$

$$[\![?_i^{\mathcal{X}}(\mathcal{H}_1, \ldots, \mathcal{H}_n)]\!]_\partial = \begin{cases} \mathcal{X}([\![\mathcal{H}_1]\!]_\partial, \ldots, [\![\mathcal{H}_n]\!]_\partial) & \text{if } \exists i \in [1, n].\ \textsc{Partial}([\![\mathcal{H}_i]\!]_\partial) \\ [\![\mathcal{X}([\![\mathcal{H}_1]\!]_\partial, \ldots, [\![\mathcal{H}_n]\!]_\partial)]\!] & \text{otherwise} \end{cases}$$

Figure 3.4. Partial evaluation of hypothesis. We write $\textsc{Partial}([\![\mathcal{H}]\!]_\partial)$ if $[\![\mathcal{H}]\!]_\partial$ contains at least one question mark.

$$\begin{array}{lcl} \text{Term } t & := & \text{const} \mid y_i \mid \mathcal{X}(t_1, ..., t_n)\ (\mathcal{X} \in \Lambda_v) \\ \text{Qualifier } \mathcal{Q} & := & (x, \mathsf{T}) \mid \lambda y_1, \ldots y_n.\ t \\ \text{Hypothesis } \mathcal{H} & := & (?_i : \tau) \mid (?_i : \tau)@\mathcal{Q} \\ & & \mid ?_i^{\mathcal{X}}(\mathcal{H}_1, ..., \mathcal{H}_n)\ \ (\mathcal{X} \in \Lambda_{\mathsf{T}}) \end{array}$$

Figure 3.5. Context-free grammar for hypotheses

holes). More formally, hypotheses $\mathcal{H}$ are defined by the grammar presented in Figure 3.5. In the simplest form, a hypothesis $(?_i : \tau)$ represents an unknown expression of type τ. More complicated hypotheses are constructed using table transformation components $\mathcal{X} \in \Lambda_{\mathsf{T}}$. In particular, if $\mathcal{X} = (f, \tau, \phi) \in \Lambda_{\mathsf{T}}$, a hypothesis of the form $?_i^{\mathcal{X}}(\mathcal{H}_1, \ldots, \mathcal{H}_n)$ represents an expression $f(e_1, \ldots, e_n)$.

During the course of our synthesis algorithm, we will progressively fill the holes in the hypothesis with concrete expressions. For this reason, we also allow hypotheses of the form $(?_i : \tau)@\mathcal{Q}$ where *qualifier* $\mathcal{Q}$ specifies the term that is used to fill hole $?_i$. Specifically, if $?_i$ is of type `tbl`, then its corresponding qualifier has the form (x, T), which means that $?_i$ is instantiated with input variable x, which is in turn bound to table T in the input-output example provided by the user. On the other hand, if $?_i$ is of type $(\tau_1 \times \ldots \times \tau_n) \to \tau$, then the qualifier must be a first-order function $\lambda y_1, \ldots y_n.t$ constructed using

components Λ_v. [2]

Our synthesis algorithm starts with the most general hypothesis and progressively makes it more specific. Therefore, we now define what it means to *refine* a hypothesis:

Definition 3.5. *(***Hypothesis refinement***) Given two hypotheses $\mathcal{H}, \mathcal{H}'$, we say that $\mathcal{H}'$ is a refinement of $\mathcal{H}$ if it can be obtained by replacing some subterm $?_i : \tau$ of $\mathcal{H}$ by $?_i^{\mathcal{X}}(\mathcal{H}_1, \ldots, \mathcal{H}_n)$ where $\mathcal{X} = (f, \tau' \to \tau, \phi) \in \Lambda_{\mathcal{T}}$.*

In other words, a hypothesis $\mathcal{H}'$ refines another hypothesis $\mathcal{H}$ if it makes it more constrained.

Example 3.5. *The hypothesis $\mathcal{H}_1 = ?_0^{\sigma}(?_1 : \texttt{tbl}, ?_2 : \texttt{row} \to \texttt{bool})$ is a refinement of $\mathcal{H}_0 = ?_0 : \texttt{tbl}$ because $\mathcal{H}_1$ is more specific than $\mathcal{H}_0$. In particular, $\mathcal{H}_0$ represents any arbitrary expression of type* `tbl`*, whereas $\mathcal{H}_1$ represents expressions whose top-level construct is a selection.*

Since our synthesis algorithm starts with the hypothesis $?_0 : \texttt{tbl}$ and iteratively refines it, we will represent hypotheses using *refinement trees* [80]. Effectively, a refinement tree corresponds to the *abstract syntax tree (AST)* for the hypotheses from Figure 3.5. In particular, note that internal nodes labeled $?_i^{\chi}$ of a refinement tree represent hypotheses whose top-level construct is χ. If an internal node $?_i^{\chi}$ has children labeled with unknowns $?_j, \ldots, ?_{j+n}$, this means that hypothesis $?_i$ was refined to $\chi(?_j, \ldots, ?_{j+n})$. Intuitively, a refinement tree

[2] We view constants as a special case of first-order functions.

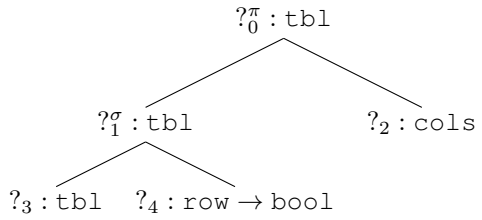


Figure 3.6. Representing hypotheses as refinement trees

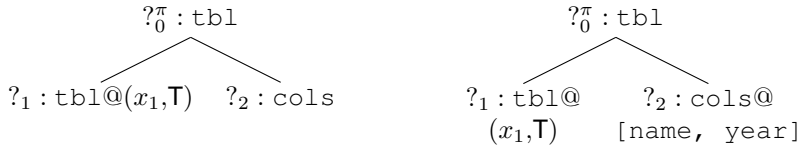


Figure 3.7. A sketch (left) and a complete program (right)

captures the *history* of refinements that occur as we search for the desired program.

Example 3.6. *Consider the refinement tree from Figure 3.6, and suppose that* π, σ *denote the standard projection and selection operators in relational algebra. This refinement tree represents the partial program* $\pi(\sigma(?, ?), ?)$*. The refinement tree also captures the search history in our synthesis algorithm. Specifically, it shows that our initial hypothesis was* $?_0$*, which then got refined to* $\pi(?_1, ?_2)$*, which in turn was refined to* $\pi(\sigma(?_3, ?_4), ?_2)$*.*

As mentioned in Section 3.1, our approach decomposes the synthesis task into two separate *sketch generation* and *sketch completion* phases. We define a *sketch* to be a special kind of hypothesis where there are no unknowns of type `tbl`.

Definition 3.6. *(***Sketch)** *A* sketch *is a special form of hypothesis where all leaf nodes of type* `tbl` *have a corresponding qualifier of the form* (x, T)*.*

T_1

id	name	age	GPA
1	Alice	8	4.0
2	Bob	18	3.2
3	Tom	12	3.0

T_2

id	name	age	GPA
2	Bob	18	3.2
3	Tom	12	3.0

Figure 3.8. Tables for Example 3.8

In other words, a sketch completely specifies the table transformers used in the target program, but the first-order functions supplied as arguments to the table transformers are yet to be determined.

Example 3.7. *Consider the refinement tree from Figure 3.6. This hypothesis is not a sketch because there is a leaf node (namely* $?_3$*) of type* `tbl` *that does not have a corresponding qualifier. On the other hand, the refinement tree shown in Figure 3.7 (left) is a sketch and corresponds to the partial program* $\pi(x_1, ?)$ *where* $?$ *is a list of column names. Furthermore, this sketch states that variable* x_1 *corresponds to table* T *from the input-output example.*

Definition 3.7. *(***Complete program***)* *A* complete program *is a hypothesis where all leaf nodes are of the form* $(?_i : \tau)@\mathfrak{Q}$.

In other words, a complete program fully specifies the expression represented by each ? in the hypothesis. For instance, a hypothesis that represents a complete program is shown in Figure 3.7 (right) and represents the relational algebra term $\lambda x_1.\pi_{name,\ year}(x_1)$.

As mentioned in Section 3.1, our synthesis procedure relies on performing partial evaluation. Hence, we define a function $[\![\mathcal{H}]\!]_\partial$, shown in Figure 3.4,

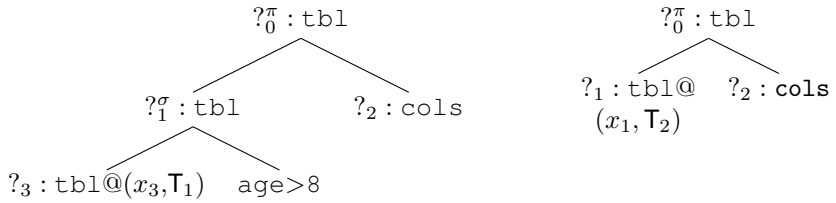


Figure 3.9. Partial evaluation on hypothesis from Figure 3.6; `age>8` stands for $?_4$: `row` $\rightarrow$ `bool`@$\lambda x.\ (x.\texttt{age} > 8)$.

for partially evaluating hypothesis $\mathcal{H}$. Observe that, if $\mathcal{H}$ is a complete program, then $[\![\mathcal{H}]\!]_\partial$ evaluates to a concrete table. Otherwise, $[\![\mathcal{H}]\!]_\partial$ returns a partially evaluated hypothesis. We write PARTIAL($[\![\mathcal{H}]\!]_\partial$) if $[\![\mathcal{H}]\!]_\partial$ does not evaluate to a concrete term (i.e., contains question marks).

Example 3.8. *Consider hypothesis $\mathcal{H}$ on the left-hand side of Figure 3.9, where T_1 is Table 1 from Figure 3.8. The refinement tree on the right-hand-side of Figure 3.9 shows the result of partially evaluating $\mathcal{H}$, where T_2 is Table 2 from Figure 3.8.*

3.5 Synthesis Algorithm

In this section, we describe the high-level structure of our synthesis algorithm, leaving the discussion of SMT-based deduction and sketch completion to the next two sections.

As illustrated schematically in Figure 3.10, our synthesis algorithm maintains a priority queue of hypotheses, which are either converted into a sketch or refined to a more specific hypothesis during each iteration. Specifically, the synthesis procedure picks the most promising hypothesis $\mathcal{H}$ according to

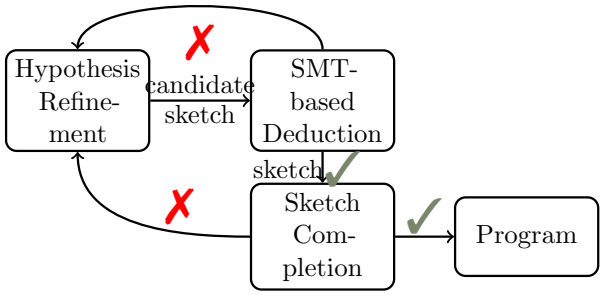


Figure 3.10. Illustration of the top-level synthesis algorithm

some heuristic cost metric (explained in Section 3.8) and asks the deduction engine if $\mathcal{H}$ can be successfully converted into a sketch. If the deduction engine refutes this conjecture, we then discard $\mathcal{H}$ but add all possible (one-level) refinements of $\mathcal{H}$ into the worklist. Otherwise, we convert hypothesis $\mathcal{H}$ into a sketch $\mathcal{S}$ and try to complete it using the *sketch completion engine*.

Algorithm 3.1 describes our top-level synthesis algorithm in more detail. Given an example $\mathcal{E}$ and a set of components Λ, SYNTHESIZE either returns a complete program that satisfies $\mathcal{E}$ or yields $\bot$, meaning that no such program exists.

Internally, the SYNTHESIZE procedure maintains a priority queue W of all hypotheses. Initially, the only hypothesis in W is $?_0$, which represents any possible program. In each iteration of the while loop (lines 5–18), we pick a hypothesis $\mathcal{H}$ from W and invoke the DEDUCE procedure (explained later) to check if $\mathcal{H}$ can be directly converted into a sketch by filling holes of type `tbl` with the input variables. Note that our deduction procedure is sound but, in general, not complete: Since component specifications are over-approximate, the deduction procedure can return $\top$ (i.e., true) even though no

valid completion of the sketch exists. However, DEDUCE returns $\perp$ only when the current hypothesis requires further refinement. Hence, the use of deduction does not lead to a loss of completeness in our overall synthesis approach.

If DEDUCE does not find a conflict, we then convert the current hypothesis $\mathcal{H}$ into a set of possible sketches (line 11). The function SKETCHES used at line 11 is presented using inference rules in Figure 3.11. Effectively, we convert hypothesis $\mathcal{H}$ into a sketch by replacing each hole of type `tbl` with one of the input variables x_j, which corresponds to table T_j in the input-output example.

Now, given a candidate sketch $\mathcal{S}$, we try to complete it using the call to FILLSKETCH at line 12 (explained in Section 3.7). FILLSKETCH returns a *set* of complete programs $\mathcal{P}$ such that each $p \in \mathcal{P}$ is valid with respect to our deduction procedure. However, as our deduction procedure is incomplete, p may not satisfy the input-output examples. Hence, we only return p as a solution if p satisfies $\mathcal{E}$ (line 14).

$$\frac{\begin{array}{c}\mathsf{T}_j \in \mathsf{T}_{in} \\ \mathcal{H} = (?_i : \texttt{tbl})\end{array}}{\mathcal{H}@(x_j, \mathsf{T}_j) \in \mathit{Sketches}(\mathcal{H}, \vec{\mathsf{T}}_{in})} \quad (1)$$

$$\frac{\begin{array}{c}\mathcal{H} = ?_i : \tau_i \\ \tau_i \neq \texttt{tbl}\end{array}}{\mathcal{H} \in \mathit{Sketches}(\mathcal{H}, \vec{\mathsf{T}}_{in})} \quad (2)$$

$$\frac{\begin{array}{c}\mathcal{H} = ?_i^{\chi}(\mathcal{H}_1, ..., \mathcal{H}_n) \\ \mathcal{H}'_i \in \mathit{Sketches}(\mathcal{H}_i, \vec{\mathsf{T}}_{in})\end{array}}{?_i^{\chi}(\mathcal{H}'_1, ..., \mathcal{H}'_n) \in \mathit{Sketches}(\mathcal{H}, \vec{\mathsf{T}}_{in})} \quad (3)$$

Figure 3.11. Converting a hypothesis into a sketch.

Algorithm 3.1 Synthesis Algorithm

```
1:  procedure SYNTHESIZE(E, Λ)
2:      input: Input-output example E and components Λ
3:      output: Synthesized program or ⊥ if failure
4:      W := {?0:tbl}                                   ▷ Init worklist
5:      while W ≠ ∅ do
6:          choose H ∈ W;
7:          W := W\{H}
8:          if DEDUCE(H, E) = ⊥ then                    ▷ Contradiction
9:              goto refine;
10:                                                     ▷ No contradiction
11:         for S ∈ SKETCHES(H, E_in) do
12:             P := FILLSKETCH(S, E)
13:             for p ∈ P do
14:                 if CHECK(p, E) then return p
15:         refine:                                     ▷Hypothesis refinement
16:         for X ∈ Λ_T, (?_i: tbl) ∈ LEAVES(H) do
17:             H' := H[?_j^X(?_j : τ⃗)/?_i]
18:             W := W ∪ H'
19:     return ⊥
```

Lines 16-18 of Algorithm 3.1 perform *hypothesis refinement*. The idea behind hypothesis refinement is to replace one of the holes of type `tbl` in $\mathcal{H}$ with a component from Λ_{T}, thereby obtaining a more specific hypothesis. Each of the refined hypotheses is added to the worklist and possibly converted into a sketch in future iterations.

Lib	Component	Description	Specification
tidyr	spread	Spread a key-value pair across multiple columns.	$\mathsf{T}_{out}.\mathsf{row} \leq \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{col} \geq \mathsf{T}_{in}.\mathsf{col}$
	gather	Takes multiple columns and collapses into key-value pairs, duplicating all other columns as needed.	$\mathsf{T}_{out}.\mathsf{row} \geq \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{col} \leq \mathsf{T}_{in}.\mathsf{col}$
dplyr	select	Project a subset of columns in a data frame.	$\mathsf{T}_{out}.\mathsf{row} = \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{col} < \mathsf{T}_{in}.\mathsf{col}$
	filter	Select a subset of rows in a data frame.	$\mathsf{T}_{out}.\mathsf{row} < \mathsf{T}_{in}.\mathsf{row}$ $\mathsf{T}_{out}.\mathsf{col} = \mathsf{T}_{in}.\mathsf{col}$

Table 3.1. Sample specifications of a few components

3.6 SMT-based Deduction

In the previous section, we described the structure of the synthesis algorithm, but did not yet explain the underlying deductive reasoning engine. The key idea here is to generate an SMT formula that corresponds to the specification of the current sketch and to check whether the input-output example satisfies this specification.

Component specifications. We use the specifications of individual components to derive the overall specification for a given hypothesis. As mentioned earlier, these specifications need not be precise and can, in general, over-approximate the behavior of the components. For instance, Table 3.1 shows sample specifications for a subset of methods from two popular R libraries. Note that these sample specifications do not fully capture the behavior of each component and only describe the relationship between the number of rows and

$$
\begin{array}{rl}
\Phi(\mathcal{H}_i) & = \alpha([\![\mathcal{H}_i]\!]_\partial)[?_i/x] \text{ if } \neg\textsc{Partial}([\![\mathcal{H}_i]\!]_\partial) \\
\Phi(\mathcal{H}_i) & = \top \qquad \text{else if } \textsc{Isleaf}(\mathcal{H}_i) \\
\Phi(?_0^{\mathcal{X}}(\mathcal{H}_1, ..., \mathcal{H}_n)) & = \bigwedge_{1 \leq i \leq n} \Phi(\mathcal{H}_i) \wedge \phi_\chi[?_0/y, \vec{?_i}/\vec{x_i}]
\end{array}
$$

Figure 3.12. Constraint generation for hypotheses. $?_i$ denotes the root variable of $\mathcal{H}_i$ and the specification of $\mathcal{X}$ is $\phi_\mathcal{X}$. Function α generates an SMT formula describing its input table.

columns in the input and output tables. [3] For example, consider the `filter` function from the `dplyr` library for selecting a subset of the rows that satisfy a given predicate in the data frame. The specification of `filter`, which is effectively the selection operator σ from relational algebra, is given by:

$$\mathsf{T}_{out}.\mathsf{row} < \mathsf{T}_{in}.\mathsf{row} \wedge \mathsf{T}_{out}.\mathsf{col} = \mathsf{T}_{in}.\mathsf{col}$$

In other words, this specification expresses that the table obtained after applying the `filter` function contains fewer rows but the same number of columns as the input table. [4]

Generating specification for hypothesis. Given a hypothesis $\mathcal{H}$, we need to generate the specification for $\mathcal{H}$ using the specifications of the individual components used in $\mathcal{H}$. Towards this goal, the function $\Phi(\mathcal{H})$ defined in Figure 3.12 returns the specification of hypothesis $\mathcal{H}$.

In the simplest case, $\mathcal{H}_i$ corresponds to a complete program (line 1 of

[3] The actual specifications used in our implementation are slightly more involved. In Section 2.9, we compare the performance of Morpheus using two different specifications.

[4] In principle, the number of rows may be unchanged if the predicate does not match any row. However, we need not consider this case since there is a simpler program without `filter` that satisfies the example.

Figure 3.12) [5]. In this case, we evaluate the hypothesis to a table T and obtain $\Phi(\mathcal{H}_i)$ as the "abstraction" of T. In particular, the *abstraction function* α used in Figure 3.12 takes as input a concrete table T and returns a constraint describing that table. In general, the definition of the abstraction function α depends on the granularity of the component specifications. For instance, if our component specifications only refer to the number of rows and columns, then a suitable abstraction function for an $m \times n$ table would yield $x.\mathsf{row} = m \wedge x.\mathsf{col} = n$. In general, we assume variable x is used to describe the input table of α.

Let us now consider the second case in Figure 3.12 where $\mathcal{H}_i$ is a leaf, but not a complete program. In this case, since we do not have any information about what $\mathcal{H}_i$ represents, we return $\top$ (i.e., *true*) as the specification.

Finally, let us consider the case where the hypothesis is of the form $?_0^{\mathcal{X}}(\mathcal{H}_1, \ldots, \mathcal{H}_n)$. In this case, we first recursively infer the specifications of sub-hypotheses $\mathcal{H}_1, \ldots, \mathcal{H}_n$. Now suppose that the specification of $\mathcal{X}$ is given by $\phi_{\mathcal{X}}(\vec{x}, y)$, where $\vec{x}$ and y denote $\mathcal{X}$'s inputs and output respectively. If the root variable of each hypothesis $\mathcal{H}_i$ is given by $?_i$, then the specification for the overall hypothesis is obtained as:

$$\bigwedge_{1 \leq i \leq n} \Phi(\mathcal{H}_i) \wedge \phi_{\chi}[?_0/y, \vec{?_i}/\vec{x_i}]$$

Example 3.9. *Consider hypothesis $\mathcal{H}$ from Figure 3.6, and suppose that the*

[5] Recall that the DEDUCE procedure will also be used during sketch completion. While $\mathcal{H}$ can never be a complete program when called from line 8 of the SYNTHESIZE procedure (Algorithm 3.1), it can be a complete program when DEDUCE is invoked through the sketch completion engine.

specifications for relational algebra operators π and σ are the same as `select` *and* `filter` *from Table 3.1 respectively. Then,* $\Phi(\mathcal{H})$ *corresponds to the following Presburger arithmetic formula:*

$$?_1.\mathit{row} < ?_3.\mathit{row} \wedge ?_1.\mathit{col} = ?_3.\mathit{col} \wedge$$
$$?_0.\mathit{row} = ?_1.\mathit{row} \wedge ?_0.\mathit{col} < ?_1.\mathit{col}$$

Here, $?_3, ?_0$ *denote the input and output tables respectively, and* $?_1$ *is the intermediate table obtained after selection.*

Deduction using SMT. Algorithm 3.2 presents our deduction algorithm using the constraint generation function Φ defined in Figure 3.12. Given a hypothesis $\mathcal{H}$ and input-output example $\mathcal{E}$, Deduce returns $\perp$ if $\mathcal{H}$ does not correspond to a valid sketch. In other words, $\textsc{Deduce}(\mathcal{H}, \mathcal{E}) = \perp$ means that we cannot obtain a program that satisfies the input-output examples by replacing holes with inputs.

As shown in Algorithm 3.2, the Deduce procedure generates a constraint ψ and checks its satisfiability using an SMT solver. If ψ is unsatisfiable, hypothesis $\mathcal{H}$ cannot be unified with the input-output example and can therefore be rejected.

Let us now consider the construction of SMT formula ψ in Algorithm 3.2. First, given a hypothesis $\mathcal{H}$, the corresponding sketch must map each of the unknowns of type `tbl` to one of the arguments. Hence, the constraint φ_{in} generated at line 5 indicates that each leaf with label $?_j$ corresponds to some argument x_i. Similarly, φ_{out} expresses that the root variable of hypothesis $\mathcal{H}$

Algorithm 3.2 SMT-based Deduction Algorithm

1: **procedure** DEDUCE($\mathcal{H}, \mathcal{E}$)
2: **input:** Hypothesis $\mathcal{H}$, input-output example $\mathcal{E}$
3: **output:** $\bot$ if cannot be unified with $\mathcal{E}$; $\top$ otherwise
4: $\mathcal{S} := \{?_j \mid ?_j : \texttt{tbl} \in \text{LEAVES}(\mathcal{H})\}$
5: $\varphi_{in} := \bigwedge_{?_j \in \mathcal{S}} \bigvee_{1 \leq i \leq |\mathcal{E}_{in}|} (?_j = x_i)$
6: $\varphi_{out} := (y = \text{ROOTVAR}(\mathcal{H}))$
7: $\psi := \left(\begin{array}{c} \Phi(\mathcal{H}) \wedge \varphi_{in} \wedge \varphi_{out} \wedge \\ \bigwedge_{\mathsf{T}_i \in \mathcal{E}_{in}} (\alpha(\mathsf{T}_i)[x_i/x]) \wedge \alpha(\mathsf{T}_{out})[y/x] \end{array} \right)$
8: **return** SAT(ψ)

must correspond to the return value y of the synthesized program. Hence, the constraint $\Phi(\mathcal{H}) \wedge \varphi_{in} \wedge \varphi_{out}$ expresses the specification of the sketch in terms of variables $x_1, \ldots, x_n, y$.

Now, to check if $\mathcal{H}$ is unifiable with example $\mathcal{E}$, we must also generate constraints that describe each table T^i_{in} in terms of x_i and T_{out} in terms of y. Recall from earlier that the abstraction function $\alpha(\mathsf{T})$ generates an SMT formula describing T in terms of variable x. Hence, the constraint

$$\bigwedge_{\mathsf{T}_i \in \mathcal{E}_{in}} (\alpha(\mathsf{T}_i)[x_i/x]) \wedge \alpha(\mathsf{T}_{out})[y/x]$$

expresses that each T^i_{in} must correspond to x_i and T_{out} must correspond to variable y. Thus, the unsatisfiability of formula ψ at line 7 indicates that hypothesis $\mathcal{H}$ can be rejected.

Example 3.10. *Consider the hypothesis from Figure 3.6, and suppose that the input and output tables are T_1 and T_2 from Figure 3.8 respectively. The*

DEDUCE *procedure from Algorithm 3.2 generates the following constraint* ψ*:*

$$?_1.\mathit{row} < ?_3.\mathit{row} \wedge ?_1.\mathit{col} = ?_3.\mathit{col} \wedge ?_0.\mathit{row} = ?_1.\mathit{row}$$
$$\wedge\ ?_0.\mathit{col} < ?_1.\mathit{col} \wedge x_1 = ?_3 \wedge y = ?_0 \wedge$$
$$x_1.\mathit{row} = 3 \wedge x_1.\mathit{col} = 4 \wedge y.\mathit{row} = 2 \wedge y.\mathit{col} = 4$$

Observe that $\Phi(\mathcal{H}) \wedge \varphi_{in} \wedge \varphi_{out}$ *implies* $y.\mathit{col} < x_1.\mathit{col}$*, indicating that the output table should have fewer columns than the input table. Since we have* $x_1.\mathit{col} = y.\mathit{col}$*, constraint* ψ *is unsatisfiable, allowing us to reject the hypothesis.*

3.7 Sketch Completion

The goal of sketch completion is to fill the remaining holes in the hypothesis with first-order functions constructed using components in Λ_v. For instance, consider the sketch $\pi(\sigma(x, ?_1), ?_2)$ where π, σ are the familiar projection and selection operators from relational algebra. Now, in order to fill hole $?_1$, we need to know the columns in table x. Similarly, in order to fill hole $?_2$, we need to know the columns in the intermediate table obtained using selection.

As this example illustrates, the vocabulary of first-order functions that can be supplied as arguments to table transformers often depends on the shapes (i.e., schemas) of the other arguments of type `tbl`. For this reason, our sketch completion algorithm synthesizes the program *bottom-up*, evaluating terms of type `tbl` before synthesizing the other arguments. Furthermore, as discussed in Section 3.1, the completion of program sketches in a bottom-up manner allows us to perform partial evaluation, which in turn increases the effectiveness of the deductive reasoning engine.

$$\frac{\mathit{type}(\mathsf{T}) = \{l_1 : \tau_1, ..., l_n : \tau_n\} \quad c = [l_i \mid i \in C_i] \text{ for } C_i \in \mathcal{P}([1, n])}{\Gamma \vdash c \in \Omega(\texttt{cols}, \mathsf{T})} \quad \text{(Cols)}$$

$$\frac{c \in \mathsf{T}, \quad \mathit{type}(c) = \tau \quad \tau \in \{\texttt{num}, \texttt{ string}\}}{\Gamma \vdash c \in \Omega(\tau, \mathsf{T})} \quad \text{(Const)}$$

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash x \in \Omega(\tau, \mathsf{T})} \quad \text{(Var)}$$

$$\frac{\Gamma \vdash t_1 \in \Omega(\tau_1, \mathsf{T}) \quad \Gamma \vdash t_2 \in \Omega(\tau_2, \mathsf{T})}{\Gamma \vdash (t_1, t_2) \in \Omega(\tau_1 \times \tau_2, \mathsf{T})} \quad \text{(Tuple)}$$

$$\frac{(f, \tau' \to \tau, \phi) \in \Lambda_v \quad \Gamma \vdash t \in \Omega(\tau', \mathsf{T})}{\Gamma \vdash f(t) \in \Omega(\tau, \mathsf{T})} \quad \text{(App)}$$

$$\frac{\tau = (\tau_1 \times \ldots \times \tau_n \to \tau') \quad \Gamma' = \Gamma \cup \{x_1 : \tau_1, \ldots x_n : \tau_n\} \quad \Gamma' \vdash t \in \Omega(\tau', \mathsf{T})}{\Gamma \vdash (\lambda x_1, \ldots, x_n.\ t) \in \Omega(\tau, \mathsf{T})} \quad \text{(Lambda)}$$

Figure 3.13. Table-driven type inhabitation rules.

Table-driven type inhabitation. At a high level, our sketch completion procedure is type-directed and synthesizes an argument of type τ by enumerating all inhabitants of τ. However, as argued earlier, the valid inhabitants of type τ are determined by a particular table. Hence, we consider the table-driven variant of the standard type inhabitation problem: That is, given a type τ and a concrete table T, what are all valid inhabitants of τ with respect to the universe of constants used in T?

We formalize this variant of the type inhabitation problem using the inference rules shown in Figure 3.13. Specifically, these rules derive judgments of the form $\Gamma \vdash t \in \Omega(\tau, \mathsf{T})$ where Γ is a type environment mapping variables to types. The meaning of this judgment is that, under type environment Γ, term t is a valid inhabitant of type τ with respect to table T. Observe that we need the type environment Γ due to the presence of function types: That is, given a function type $\tau_1 \to \tau_2$, we need Γ to enumerate valid inhabitants of τ_2. Since the typing rules from Figure 3.13 resemble those for the simply-typed lambda calculus, we do not explain them in detail. The main difference is that constants of type `cols` are drawn from lists of column names from the table schema, and constants of type `num` and `string` are drawn from values in the table.

Example 3.11. *Consider table* T_1 *from Figure 3.8 and the type environment* $\Gamma : \{x \mapsto \texttt{string}\}$. *Assuming* $\texttt{eq} : \texttt{string} \times \texttt{string} \to \texttt{bool}$ *is a component in* Λ_v, *we have* $\texttt{eq}(x, \texttt{"Alice"}) \in \Omega(\texttt{bool}, \mathsf{T}_1)$ *using the* App, Const, Var *rules. Similarly,* $\lambda x.\texttt{eq}(x, \texttt{"Bob"})$ *is also a valid inhabitant of* $\texttt{string} \to \texttt{bool}$ *with respect to* T_1.

Sketch completion algorithm. Our sketch completion procedure is described using the inference rules shown in Figure 3.14. As mentioned previously, the algorithm is bottom-up and first synthesizes all arguments of type `tbl` before synthesizing other arguments. Given sketch $\mathcal{S}$ and example $\mathcal{E}$,

$$\frac{\begin{array}{c}\mathcal{S} = (?_i : \tau_i) \\ t \in \Omega(\tau_i, \mathsf{T}, \emptyset) \\ \textsc{Deduce}(\mathcal{S}_f[\mathcal{S}@t/\mathcal{S}], \mathcal{E}) \neq \bot\end{array}}{\mathcal{S}@t \in \mathcal{C}_v(\mathcal{S}, \mathcal{S}_f, \mathcal{E}, \mathsf{T})} \quad (1)$$

$$\frac{\mathcal{S} = (?_i, \texttt{tbl})@(x, \mathsf{T})}{(\mathcal{S}, \mathsf{T}) \in \mathcal{C}_\mathsf{T}(\mathcal{S}, \mathcal{S}_f, \mathcal{E})} \quad (2)$$

$$\frac{\begin{array}{c}\mathcal{S} = ?_i^x(\vec{\mathcal{H}} : \texttt{tbl}, \vec{\mathcal{H}'} : \tau) \quad (\tau \neq \texttt{tbl}) \\ (\mathcal{P}_j, \mathsf{T}_j) \in \mathcal{C}_\mathsf{T}(\mathcal{H}_j, \mathcal{S}_f, \mathcal{E}) \\ \mathcal{P}'_j \in \mathcal{C}_v(\mathcal{H}'_j, \mathcal{S}_f[\vec{\mathcal{P}}/\vec{\mathcal{H}}], \mathcal{E}, \mathsf{T}_1 \times \ldots \times \mathsf{T}_n) \\ \textsc{Deduce}(\mathcal{S}_f[\vec{\mathcal{P}}/\vec{\mathcal{H}}, \vec{\mathcal{P}'}/\vec{\mathcal{H}'}], \mathcal{E}) \neq \bot \\ \mathcal{P}^* = \mathcal{S}[\vec{\mathcal{P}}/\vec{\mathcal{H}}, \vec{\mathcal{P}'}/\vec{\mathcal{H}'}]\end{array}}{(\mathcal{P}^*, [\![\mathcal{P}^*]\!]_\partial) \in \mathcal{C}_\mathsf{T}(\mathcal{S}, \mathcal{S}_f, \mathcal{E})} \quad (3)$$

$$\frac{(\mathcal{P}, \mathsf{T}) \in \mathcal{C}_\mathsf{T}(\mathcal{S}, \mathcal{S}, \mathcal{E})}{\mathcal{P} \in \textsc{FillSketch}(\mathcal{S}, \mathcal{E})} \quad (4)$$

Figure 3.14. Sketch completion rules.

$\textsc{FillSketch}(\mathcal{S}, \mathcal{E})$ returns a set of hypotheses representing complete well-typed programs that are valid with respect to our deduction system.

The first rule in Figure 3.14 corresponds to a base case of the FillSketch procedure and is used for completing hypotheses that are *not* of type `tbl`. Here, $\mathcal{S}$ represents a subpart of the sketch that we want to complete, T is the table that should be used in completing $\mathcal{S}$, and $\mathcal{S}_f$ is the full sketch. Since $\mathcal{S}$ represents an unknown expression of type τ_i, we use the type inhabitation rules from Figure 3.13 to find a well-typed instantiation t of τ_i with respect to table T. Given completion t of $?_i$, the full sketch now becomes $\mathcal{S}_f[\mathcal{S}@t/\mathcal{S}]$, and we use the deduction system to check whether the new hypothesis is valid.

Since our deduction procedure uses partial evaluation, we may now be able to obtain a concrete table for some part of the sketch, thereby enhancing the power of deductive reasoning.

The second rule from Figure 3.14 is also a base case of the FillSketch procedure. Since any leaf $?_i$ of type `tbl` is already bound to some input variable x in the sketch, there is nothing to complete; hence, we just return $\mathcal{S}$ itself.

Rule (3) corresponds to the recursive step of the FillSketch procure and is used to complete a sketch with top-most component χ. Specifically, consider a sketch of the form $?_i^{\chi}(\vec{\mathcal{H}}, \vec{\mathcal{H}'})$ where $\vec{\mathcal{H}}$ denotes arguments of type `tbl` and $\vec{\mathcal{H}'}$ represents first-order functions. Since the vocabulary of $\vec{\mathcal{H}'}$ depends on the completion of $\vec{\mathcal{H}}$ (as explained earlier), we first recursively synthesize $\vec{\mathcal{H}}$ and obtain a set of complete programs $\vec{\mathcal{P}}$, together with their partial evaluation $\mathsf{T}_1, \ldots, \mathsf{T}_n$. Now, observe that each $\mathcal{H}'_j \in \vec{\mathcal{H}'}$ can refer to any of the columns in $\mathsf{T}_1 \times ... \times \mathsf{T}_n$; hence we recursively synthesize the remaining arguments $\vec{\mathcal{H}'}$ using table $\mathsf{T}_1 \times ... \times \mathsf{T}_n$. Now, suppose that the hypotheses $\vec{\mathcal{H}}$ and $\vec{\mathcal{H}'}$ are completed using terms $\vec{\mathcal{P}}$ and $\vec{\mathcal{P}'}$ respectively, and the new (partially filled) sketch is now $\mathcal{S}_f[\vec{\mathcal{P}}/\vec{\mathcal{H}}, \vec{\mathcal{P}'}/\vec{\mathcal{H}'}]$. Since there is an opportunity for rejecting this partially filled sketch, we again check whether $\mathcal{S}_f[\vec{\mathcal{P}}/\vec{\mathcal{H}}, \vec{\mathcal{P}'}/\vec{\mathcal{H}'}]$ is consistent with the input-output examples using deduction.

Example 3.12. *Consider hypothesis $\mathcal{H}$ from Figure 3.6, the input table T_1 from Figure 3.8, and the output table T_3 from Figure 3.15. We can successfully convert this hypothesis into the sketch $\lambda x.?_0^{\pi}(?_1^{\sigma}(?_3@(x, \mathsf{T}_1), ?_4), ?_2)$. Since*

T_3

id	name	age
2	Bob	18
3	Tom	12

T_4

id	name	age	GPA
2	Bob	18	3.2

Figure 3.15. Tables for Example 3.12

FillSketch *is bottom-up, it first tries to fill hole* $?_4$*. In this case, suppose that we try to instantiate hole* $?_4$ *with the predicate* `age > 12` *using rule (1) from Figure 3.14. However, when we call* Deduce *on the partially-completed sketch* $\lambda x.?_0^{\pi}(?_1^{\sigma}(?_3@(x, T_1), \mathsf{age} > 12), ?_2)$*,* $?_1$ *is refined as* T_4 *in Figure 3.15 and we obtain the following constraint:*

$$?_1.row < ?_3.row \wedge ?_1.col = ?_3.col \wedge ?_0.row = ?_1.row \wedge$$
$$?_0.col < ?_1.col \wedge x_1 = ?_3 \wedge x_1.row = 3 \wedge x_1.col = 4 \wedge$$
$$y = ?_0 \wedge y.row = 2 \wedge y.col = 3 \wedge \underline{\mathbf{?_1.col = 4 \wedge ?_1.row = 1}}$$

Note that the last two conjuncts (underlined) are obtained using partial evaluation. Since this formula is unsatisfiable, we can reject this hypothesis without having to fill hole $?_2$*.*

3.8 Implementation

We have implemented our synthesis algorithm in a tool called Morpheus, written in C++. Morpheus uses the Z3 SMT solver [21] with the theory of Linear Integer Arithmetic for checking the satisfiability of constraints generated by our deduction engine.

Recall from Section 3.5 that Morpheus uses a cost model for picking the "best" hypothesis from the worklist. Inspired by previous work on code completion [88], we use a cost model based on a statistical analysis of existing code.

Specifically, MORPHEUS analyzes existing code snippets that use components from Λ_T and represents each snippet as a 'sentence' where 'words' correspond to components in Λ_T. Given this representation, MORPHEUS uses the 2-gram model in SRILM [110] to assign a score to each hypothesis. Specifically, we train our language model by collecting approximately 15,000 code snippets from Stackoverflow using the search keywords `tidyr` and `dplyr`. For each code snippet, we ignore its control flow and represent it using a "sentence" where each "word" corresponds to an API call. Based on this training data, the hypotheses in the worklist W from Algorithm 3.1 are then ordered using the scores obtained from the n-gram model.

Following the *Occam's razor* principle, MORPHEUS explores hypotheses in increasing order of size. However, if the size of the correct hypothesis is a large number k, MORPHEUS may end up exploring many programs before reaching length k. In practice, we have found that a better strategy is to exploit the inherent parallelism of our algorithm. Specifically, MORPHEUS uses multiple threads to search for solutions of different sizes and terminates as soon as any thread finds a correct solution.

3.9 Evaluation

To evaluate our method, we collected 80 data preparation tasks, all of which are drawn from discussions among R users on Stackoverflow. The MORPHEUS project webpage [1] contains (i) the Stackoverflow post for each benchmark, (ii) an input-output example, and (iii) the solution synthesized by

Category	Description	#	No deduction		Spec 1		Spec 2	
			#Solved	Time	#Solved	Time	#Solved	Time
C1	*Reshaping* dataframes from either "long" to "wide" or "wide" to "long"	4	2	198.14	4	15.48	4	6.70
C2	*Arithmetic computations* that produce values not present in the input tables	7	6	5.32	7	1.95	7	0.59
C3	Combination of *reshaping* and *string manipulation* of cell contents	34	28	51.01	31	6.53	34	1.63
C4	*Reshaping* and *arithmetic computations*	14	9	162.02	10	90.33	12	15.35
C5	Combination of *arithmetic computations* and *consolidation* of information from multiple tables into a single table	11	7	8.72	10	3.16	11	3.17
C6	*Arithmetic computations* and *string manipulation* tasks	2	1	280.61	2	49.33	2	3.03
C7	*Reshaping* and *consolidation* tasks	1	0	✗	1	135.32	1	130.92
C8	Combination of *reshaping*, *arithmetic computations* and *string manipulation*	6	1	✗	3	198.42	6	38.42
C9	Combination of *reshaping*, *arithmetic computations* and *consolidation*	1	0	✗	0	✗	1	97.3
Total		80	54 (67.5%)	95.53	68 (85.0%)	8.57	78 (97.5%)	3.59

Figure 3.16. Summary of experimental results. All times are median in seconds and ✗ indicates a timeout (> 5 minutes).

Morpheus.

Our evaluation aims to answer the following questions:

Q1. Can Morpheus successfully automate real-world data preparation tasks and what is its running time?

Q2. How big are the benefits of SMT-based deduction and partial evaluation in the performance of Morpheus?

Q3. How complex are the data preparation tasks that can be successfully automated using Morpheus?

Q4. Are there existing synthesis tools that can also automate the data preparation tasks supported by Morpheus?

To answer these questions, we performed a series of experiments on the 80 data preparation benchmarks, using the input-output examples provided by the authors of the Stackoverflow posts. In these experiments, we use the table transformation components provided by two popular table manipulation libraries, namely `tidyr` and `dplyr`. The value transformers we use in our evaluation include standard comparison operators such as < , > as well as aggregate functions like `MEAN` and `SUM`. In total, our experiments make use of a total of 20 different components. All experiments 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.

Summary of results. The results of our evaluation are summarized in Figure 3.16. Here, the *"Description"* column provides a brief English description of each category, and the column "#" shows the number of benchmarks in each category. The *"No deduction"* column indicates the running time of a version of MORPHEUS that uses purely enumerative search without deduction. (This basic version still uses the statistical analysis described in Section 3.8 to choose the "best" hypothesis.) The columns labeled *"Spec 1"* and *"Spec 2"* show variants of MORPHEUS using two different component specifications. Specifically, *Spec 1* is less precise and only constrains the relationship between the number of rows and columns, as shown in Table 3.1. On the other hand, *Spec 2* is strictly more precise than *Spec 1* and also uses other information, such as cardinality and number of groups.

Performance. As shown in Figure 3.16, the full-fledged version of Morpheus (using the more precise component specifications) can successfully synthesize 78 out of the 80 benchmarks and times out on only 2 problems. Hence, overall, Morpheus achieves a success rate of 97.5% within a 5-minute time limit. Morpheus's median running time on these benchmarks is 3.59 seconds, and 86.3% of the benchmarks can be synthesized within 60 seconds. However, it is worth noting that running time is actually dominated by the R interpreter: Morpheus spends roughly 68% of the time in the R interpreter, while using only 15% of its running time to perform deduction (i.e., solve SMT formulas). Since the overhead of the R interpreter can be significantly reduced with sufficient engineering effort, we believe there is considerable room for improving Morpheus's running time. However, even in its current form, these results show that Morpheus is practical enough to automate a diverse class of data preparation tasks within a reasonable time limit.

Impact of deduction. As Figure 3.16 shows, deduction has a huge positive impact on the algorithm. The basic version of Morpheus that does not perform deduction times out on 32.5% of the benchmarks and achieves a median running time of 95.53 seconds. On the other hand, if we use the coarse specifications given by *Spec 1*, we already observe a significant improvement. Specifically, using *Spec 1*, Morpheus can successfully solve 68 out of the 80 benchmarks, with a median running time of 8.57 seconds. These results show that even coarse and easy-to-write specifications can have a significant positive

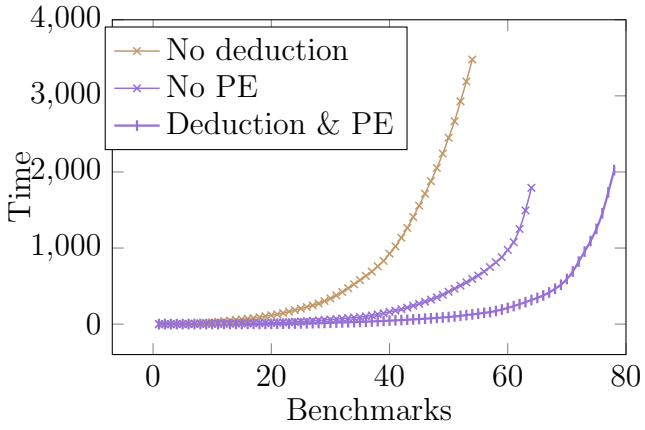


Figure 3.17. Cumulative running time of MORPHEUS

impact on synthesis.

Impact of partial evaluation. Figure 3.17 shows the cumulative running time of MORPHEUS with and without partial evaluation. Partial evaluation significantly improves the performance of MORPHEUS, both in terms of running time and the number of benchmarks solved. In particular, without partial evaluation, MORPHEUS can only solve 62 benchmarks with median running time of 34.75 seconds using *Spec 1* and 64 benchmarks with median running time of 17.07 seconds using *Spec 2*. When using partial evaluation, MORPHEUS can prune 72% of the partial programs without having to fill all holes in the sketch, thereby resulting in significant performance improvement.

Impact of language model. As described in Section 3.8, MORPHEUS uses a statistical language model (namely 2-grams) for choosing the most promising hypothesis in its worklist. Even though the idea of using statistical language

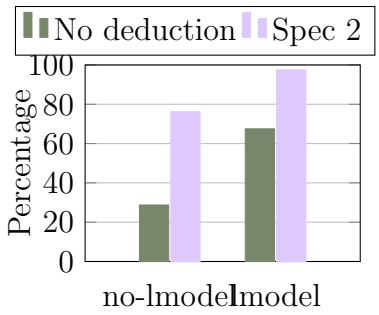


Figure 3.18. Impact of language model

models is not a contribution of this paper and is inspired by the prior work of Raychev et al. [88], we nevertheless evaluate its impact on our benchmark set consisting of various data preparation tasks. Specifically, Figure 3.18 shows the percentage of benchmarks solved by MORPHEUS with and without a language model for ordering the hypotheses. As shown in Figure 3.18, the use of the language model has a significant positive impact on the performance of MORPHEUS. Specifically, while MORPHEUS can solve 97.5% of the benchmarks using the statistical language model, it is only able to solve 76.25% of the benchmarks without the 2-gram model. However, it is worth noting that the statistical language model alone is not sufficient for solving many of our benchmarks. In particular, if we disable the deductive reasoning capabilities of MORPHEUS, we can only solve 67.5% of the benchmarks. Furthermore, MORPHEUS can only solve 28.75% of the benchmarks if we disable both deduction as well as the statistical language model.

Complexity of benchmarks. To evaluate the complexity of tasks that MORPHEUS can handle, we conducted a small user study involving 9 partici-

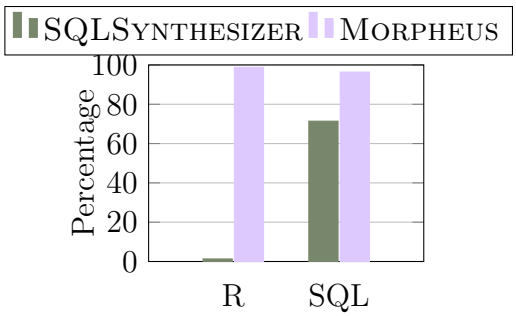


Figure 3.19. Comparison with SQLSynthesizer

pants. Of the participants, four are senior software engineers at a leading data analytics company and do data preparation "for a living". The remaining 5 participants are proficient R programmers at a university and specialize in statistics, business analytics, and machine learning. We chose 5 representative examples from our 80 benchmarks and asked the participants to solve as many of them as possible within one hour. These benchmarks belong to four categories (C2, C3, C4, C7) and take between 0.22 and 204.83 seconds to be solved by Morpheus.

In our user study, the average participant completed 3 tasks within the one-hour time limit; however, only 2 of these tasks were solved *correctly* on average. These results suggest that our benchmarks are challenging even for proficient R programmers and expert data analysts.

Comparison with λ^2. To demonstrate the advantages of our proposed approach over previous component-based synthesis techniques, we compared Morpheus with λ^2 [30], which is a general-purpose tool for synthesizing higher-order functional programs over data structures.

Since λ^2 does not have built-in support for tables, we evaluated λ^2 on the benchmarks from Figure 3.16 by representing each table as a list of lists. Even though we confirmed that λ^2 can synthesize very simple table transformations involve projection and selection, it was not able to successfully synthesize *any* of the benchmarks used in our evaluation. Upon further inspection, we believe that λ^2 fails to synthesize many of our benchmarks for two reasons: First, hypotheses in λ^2 are restricted to be of the form $\lambda x.\ F\ e\ x$, where F is a higher-order combinator, e is an expression and x is the input. However, many of our benchmarks require more general hypotheses of the form $\lambda x.F\ e_1\ e_2$ where e_1, e_2 are arbitrary expressions. Furthermore, λ^2 can only perform deduction for a built-in set of higher-order combinators for which it is possible to infer concrete input-output examples for the sub-components. However, many of the benchmarks used in our evaluation are difficult to express concisely using the set of combinators supported by λ^2.

Comparison with SQLSynthesizer. Since Morpheus is a general tool that can be used to synthesize many kinds of table transformations, we also compare it against SQLSynthesizer, which is a specialized tool for synthesizing SQL queries from examples [128]. To compare Morpheus with SQLSynthesizer, we used two different sets of benchmarks. First, we evaluated SQLSynthesizer on the 80 data preparation benchmarks from Figure 3.16. Note that some of the data preparation tasks used in our evaluation cannot be expressed using SQL, and therefore fall beyond the scope of a tool like

SQLSynthesizer. Among our 80 benchmarks, SQLSynthesizer was only able to successfully solve *one*.

To understand how Morpheus compares with SQLSynthesizer on a narrower set of table transformation tasks, we also evaluated both tools on the 28 benchmarks used in evaluating SQLSynthesizer [128]. To solve these benchmarks using Morpheus, we used the same input-output tables as SQLSynthesizer and used a total of eight higher-order components that are relevant to SQL. As shown in Figure 3.19, Morpheus also outperforms SQLSynthesizer on these benchmarks. In particular, Morpheus can solve 96.4% of the SQL benchmarks with a median running time of 1 second whereas SQLSynthesizer can solve only 71.4% with a median running time of 11 seconds.

3.10 Summary

In this chapter, we have presented a new component-based synthesis algorithm that combines type-directed enumerative search with lightweight SMT-based deduction and partial evaluation. Given a set of components equipped with over-approximate logical specifications, our approach automatically infers logical specifications of partial programs and uses SMT-based reasoning to prune the search space. Our approach further increases the power of its deductive reasoning engine by employing partial evaluation. We have applied the proposed ideas to automate a large class of data preparation tasks that involve table consolidation and reshaping. As shown in our experimental

evaluation, our tool, MORPHEUS, can automate challenging data wrangling tasks that are difficult even for proficient R programmers. Our tool is publicly available [1] and will also be released as an RStudio plug-in.

Chapter 4

NEO [1]

While both SYPET and MORPHEUS incorporate statistical models to speed up enumerative search and use logical reasoning to prune search space, none of them can learn from past mistakes. To address this limitation as well as provide a unified framework that combines statistical models and logical reasoning in a natural way, in this chapter, 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

[1] Parts of this chapter will appear in [25].

these respective domains.

4.1 Overview

So far we have shown two program synthesizers that can automate a wide range of tasks. However, 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 $\lambda x.\ \texttt{map}(x, ...)$. 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 $\lambda x.\texttt{f}(x, ...)$ 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 $\lambda x.\ \texttt{map}(x, ...)$) 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 chapter, we present a new *conflict-driven* synthesis algorithm that is capable of learning from its past mistakes. 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.

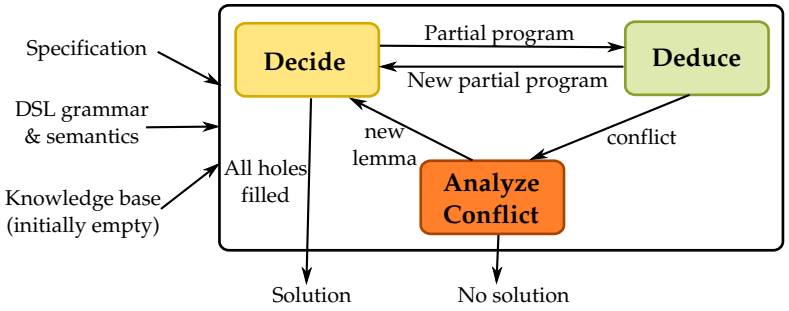


Figure 4.1. High-level architecture of our synthesis algorithm

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 4.1, 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 chapter 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 [26]. 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 [13]. 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.

4.2 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 4.2. In this section (and throughout this chapter), we

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

Figure 4.2. The grammar of a simple DSL for manipulating lists of integers; in this grammar, N is the start symbol.

represent programs using their abstract syntax tree (AST) representation. For instance, the AST shown in Figure 4.3 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 4.3 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

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

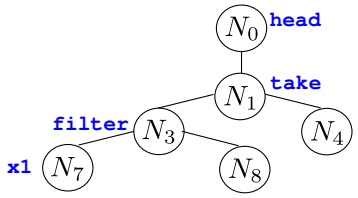


Figure 4.3. An example partial program

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. [2] Our deduction engine utilizes the semantics of the DSL, provided as first-order specifications of each component. For instance, Table 4.1 shows the specifica-

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

tions 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.

NEO uses the specifications of each component to infer a specification of the current partial program. In particular, Figure 4.4 shows the inferred specification Φ_P for partial program P from Figure 4.3. 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 $\chi_1, \ldots, \chi_n$

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

Figure 4.4. Specification of partial program from Figure 4.3, where v_i represents the intermediate value at node N_i and variables x_1 and y denote the input and output, respectively.

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 4.3. To identify the *root cause* of the conflict, NEO starts by computing a *minimal unsatisfiable core (MUC)* of the formula $\Phi_P \wedge \Phi$. For this example, the MUC includes all underlined predicates in Figure 4.4. 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 4.3 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 4.1) 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 **??**) 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.

4.3 Preliminaries

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

4.3.1 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 $p = (A \to \chi(A_1...A_k))$, 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 $A \to \chi(A_1...A_k)$.

Semantics. As mentioned in Section 4.1, 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 $\Psi(\texttt{inc}) = (y = x_1 + 1)$.

4.3.2 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 4.1. (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 $S \stackrel{*}{\Rightarrow} P$.

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 $P \stackrel{*}{\Rightarrow} P'$.

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 $A_N \to \chi_N(\ldots)$. 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 4.1. *Consider the partial program from Figure 4.3. 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.

4.4 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 4.5.

4.4.1 Overview

Algorithm 4.1 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 $\vec{x}, y$,

Algorithm 4.1 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
```

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.

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 4.4.3.

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*),

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

and CheckConflict returns the *root cause* of the conflict. As explained in Section 4.2, 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 4.4.4 and 4.4.5, 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 undoing the assignments made by Decide and Propagate during the last iteration.

Algorithm 4.1 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 4.1 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.

4.4.2 Knowledge Base and SAT Encoding of Programs

Before we can explain each of the subroutines used in Algorithm 4.1, 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 4.2. (SAT encoding of program) Let P be the AST representation of a partial program, as explained in Section 4.3.2. We use the notation π_P to denote the following SAT encoding of P:

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

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

Example 4.2. *The partial program in Figure 4.3 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 $P \sim \Omega$.

Algorithm 4.2 Outline of DECIDE

procedure DECIDE($P,\ \mathcal{G},\ \Phi,\ \Omega$)
  $V \leftarrow \{(H, p) \mid H \in \mathsf{Holes}(P),\ p = A_H \rightarrow \chi(A_1 ... A_k)\}$
  $V' \leftarrow \{(H, p) \in V \mid \mathsf{Fill}(P, H, p) \sim \Omega\}$
  return $\arg\max_{(H,p)\in V'} L_\theta(\mathsf{Fill}(P, H, p) \mid \Phi)$

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.

4.4.3 The Decide Subroutine

We will now explain each of the auxiliary procedures used in Algorithm 4.1, 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 4.2 shows the high-level structure of our DECIDE component and makes use of a procedure $\mathsf{Fill}(P, H, p)$ 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 $P' = \mathsf{Fill}(P, H, p)$ and we think of P and P' as strings in $(\Sigma \cup V)^*$, then we

have $P \Rightarrow P'$.

Algorithm 4.2 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 $P' = \mathsf{Fill}(P, H, p)$ 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 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]

4.4.4 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 4.3.

Given a decision (H, p), PROPAGATE first fills hole H with production p; it then checks whether this decision implies additional assignments. It does

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

Algorithm 4.3 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
```

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).

[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.

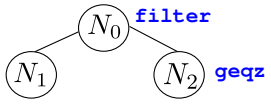
Algorithm 4.4 Outline of CheckConflict

```
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′
```

Example 4.3. *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:*

$$\left\{\neg c_{0,\texttt{filter}} \vee \neg c_{2,\texttt{eqz}},\quad \neg c_{0,\texttt{filter}} \vee \neg c_{2,\texttt{leqz}}\right\}$$

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



In particular, observe that 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`.

4.4.5 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

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

Figure 4.5. Rules defining InferSpec(P)

programs that do not satisfy the specification. This form of deduction is performed by the CheckConflict procedure outlined in Algorithm 4.4.

The core part of CheckConflict is the InferSpec procedure, described as inference rules in Figure 4.5. 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 4.4. *Figure 4.4 shows the result of calling InferSpec on the partial program from Figure 4.3.*

The following theorem states the correctness of the InferSpec procedure

presented in Figure 4.5:

Theorem 4.1. 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 SMTSolve procedure to check the satisfiability of $\Phi_P \wedge \Phi$ (line 3 of Algorithm 4.4). If this formula is unsatisfiable, Theorem 4.1 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

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

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 $N = \textsc{Node}(\phi_i)$ 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 4.5. *Consider the formula* $\Phi_P \wedge x_1.max = 82 \wedge y = 158$ *where* Φ_P *is the formula from Figure 4.4. 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 4.3; 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:*

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

4.5 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 4.6. *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.

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

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 4.2. *Let P a partial program inconsistent with specification Φ, and let κ be the MUC returned by Algorithm 4.4. 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

Algorithm 4.5 Algorithm for learning lemmas

1: **procedure** AnalyzeConflict($P,\ \mathcal{G},\ \Psi, \kappa$)
2: $\quad\varphi \leftarrow \mathit{false}$
3: $\quad$**for** $(\phi, N, \chi_N) \in \kappa$ **do**
4: $\quad\quad(A_1, \ldots, A_k) \leftarrow (A_{N_i} \mid N_i \in \mathsf{Children}(N))$
5: $\quad\quad\Sigma_N \leftarrow \{\chi \mid \chi \in \Sigma_{A_N, A_1, \ldots, A_k}\ \wedge\ \Psi(\chi) \Rightarrow \phi\}$
6: $\quad\quad\varphi \leftarrow \varphi \vee \bigwedge_{\chi \in \Sigma_N} \neg c_{s_N, \chi}$
7: $\quad$**return** φ

χ' is more restrictive than the specification $\Psi(\chi)$ 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 4.6. *Consider the element* $(y.max \leq x_1.max, N_1, \texttt{take})$ *in the MUC from Example 4.5. Also, recall from Table 4.1 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 4.5 leverages Theorem 4.2 to learn new lemmas to add to the knowledge base. As shown in Algorithm 4.5, 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 4.2 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 4.2 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 4.3. *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 4.1 is correct with respect to specification Φ.

Finally, the soundness and completeness of our algorithm follow from Theorem 4.1 and Corollary 1:

Theorem 4.4. (Soundness) *If Algorithm 4.1 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 4.5. (Completeness) *If Algorithm 4.1 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.*

4.6 Implementation

We have implemented our conflict-driven synthesis framework in a tool called NEO, written in Java. NEO uses the SAT4J [15] SAT solver to implement DECIDE and PROPAGATE and employs the Z3 [21] SMT solver to check for conflicts.

Decide. As explained in Section 4.4.3, 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 4.2 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,

[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 $\{\neg l, x_1, \ldots, x_n\}$.

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 [26]) as well as a deep neural network model (as described in DEEPCODER [13]). 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 4.4.4, 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.

CheckConflict. Our implementation of CHECKCONFLICT follows Algorithm 4.4 and uses Z3 to query the satisfiability of the corresponding SMT formula [21]. Given an unsatisfiable formula ϕ, we also use Z3 to obtain an unsatisfiable core and post-process it as described in Section 4.4.5. The un-

satisfiable 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 4.4: Since different partial programs may share the same SMT specification, Algorithm 4.4 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 4.5. 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 ANALYZE-CONFLICT. 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 under-specify the constructs' functionality to achieve a good trade-off 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 [26]. For the list manipulation domain, we use the same DSL as in prior work [13] but write our own specification since they are not available in the DeepCoder setting [13]. 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.

4.7 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 [26] for domain (i) and the language used in DEEPCODER [13] for (ii). [9] 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.

4.7.1 Comparison against Morpheus

In our first experiment, we compare NEO against MORPHEUS [26], a state-of-the-art synthesis tool that automates data wrangling tasks in R.

[9] The interested reader can find both DSLs and their specifications under supplementary materials.

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 4.4. 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, where difficulty is measured in terms of synthesis time. [10] 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.

[10]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.

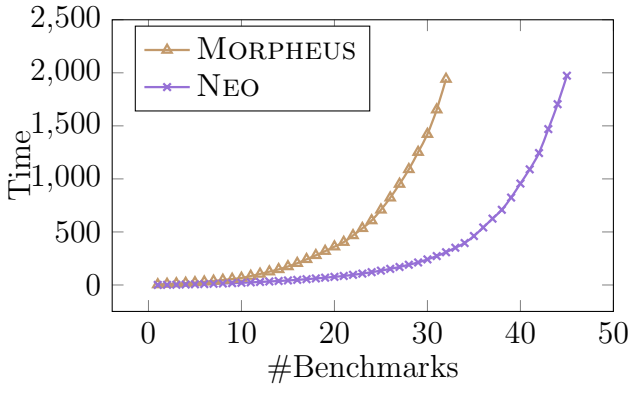


Figure 4.6. Comparison between Neo and Morpheus

Results. The results of our first experiment are summarized in Figure 4.6, 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.

Impact of different components. Figure 4.7 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

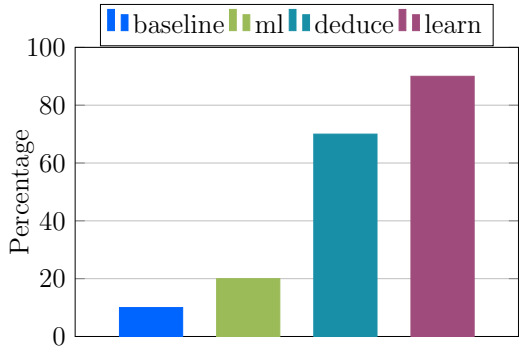


Figure 4.7. Impact of each component for data wrangling

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.

4.7.2 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 [13]. [11] Because DEEPCODER specializes in functional programs that manipulate lists, we instantiated NEO on the same domain, using the same

[11] We implemented our own version of DEEPCODER since the tool is not publicly available. Our re-implementation is faithful to the description in [13] as well as e-mail communications with the developers of DEEPCODER.

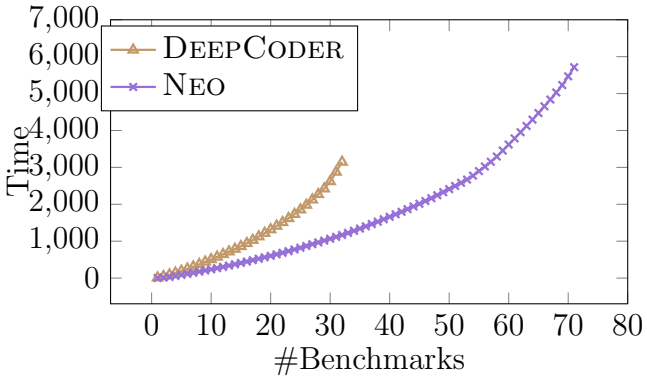


Figure 4.8. Comparison between Neo and DeepCoder

DSL constructs as DeepCoder. However, 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 [13], 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 [13]. 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

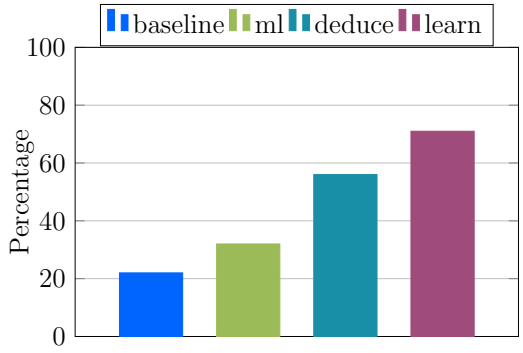


Figure 4.9. Impact of components for list manipulation

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 4.8, which also plots running time against the number of solved benchmarks. As we can see from Figure 4.8, 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 4.9 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

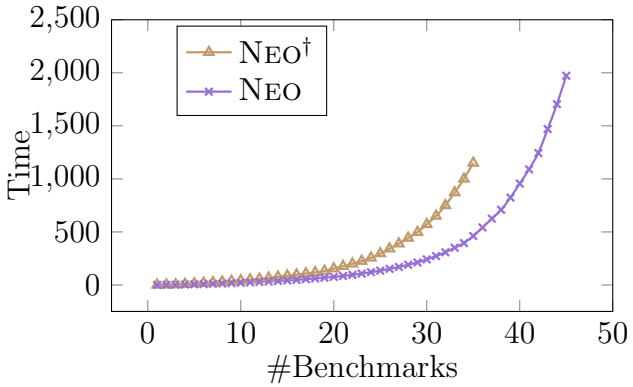


Figure 4.10. Impact of learning in data wrangling domain

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.

4.7.3 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 4.10 and 4.11. Specifically, Figure 4.10 compares NEO against NEO† on the data wrangling

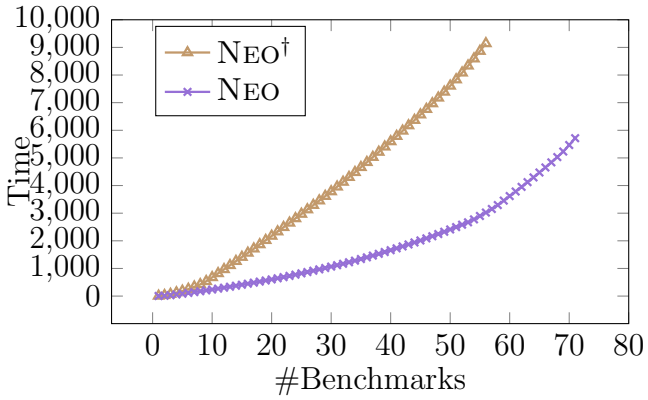


Figure 4.11. Impact of learning for the list domain

benchmarks from Section 4.7.1, whereas Figure 4.11 shows the same comparison for the list manipulation benchmarks used in Section 4.7.2. 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 4.11, 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.

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

Table 4.2. Impact of learning

While NEO is 2.7x faster than NEO† on "easy" benchmarks, NEO outperforms NEO† by 5.7x on the medium category. For benchmarks in the "hard" category, NEO is 19.8x faster than NEO† on average. [12] 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 4.2 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.

4.8 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

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

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.

4.9 Summary

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. 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. NEO is publicly available [24] and can be easily instantiated in

different synthesis tasks.

Chapter 5

Related Work

Program synthesis is an active research topic that has found many applications, including string processing [38, 96] bit-vector manipulations [58], data wrangling [26, 120], query synthesis [119, 125, 129], API completion [27, 69, 49], functional programming [84, 30, 80], and data processing [100, 123]. In what follows, we discuss prior work that is most closely related to this dissertation.

5.1 Component-based synthesis

Component-based synthesis refers to generating (straight-line) programs from a set of components, such as methods provided by an API [43, 57, 68, 59, 27, 100]. Some of these efforts [57, 43] use an SMT-solver to *search* for a composition of components. In contrast, our approaches in Neo and Morpheus use an SMT-solver as a *pruning tool* in enumerative search and does not require precise specifications of components. On the other hand, SyPet [27] searches for well-typed programs using a Petri net representation. Similar to Neo, SyPet can also work with any set of components and decomposes synthesis into two separate sketch generation and sketch completion phases. However, both the

application domains (Java APIs vs. table transformations) and the underlying techniques (Petri net reachability vs. SMT-based deduction) are very different. Finally, another related approach is Bigλ [100], which can synthesize non-trivial data-parallel programs using a set of pre-defined components. However, unlike NEO and MORPHEUS, Bigλ does not incorporate deductive reasoning to prune the search space.

5.2 Applications

In this section, we discuss applications that are related to this dissertation.

5.2.1 Table transformations

This dissertation is related to a line of work on programming-by-example (PBE) [39, 51, 14, 83, 63, 6, 30, 80, 67, 124, 85]. Of particular relevance are PBE techniques that focus on table transformations [51, 128, 67, 14]. Among these techniques, FLASHEXTRACT and FLASHRELATE address the specific problem of extracting structured data from spreadsheets and do not consider a general class of table transformations. More closely related are Harris and Gulwani's work on synthesis of spreadsheet transformations [51] and Zhang et al.'s work on synthesizing SQL queries [128]. Our approach is more general than these methods in that they use DSLs with a fixed set of primitive operations (components), whereas our approach takes a set of components as a *parameter*. For instance, Zhang et al. cannot synthesize programs that perform table

reshaping while Harris et al. supports data reshaping, but not computation or consolidation. Hence, these approaches cannot automate many of the data preparation tasks that we consider.

5.2.2 Data wrangling

Another term for data preparation is *"data wrangling"*, and prior work has considered methods to facilitate such tasks. For instance, Wrangler is an interactive visual system that aims to simplify data wrangling [61, 45]. OpenRefine is a general framework that helps users perform data transformations and clean messy data. Tools such as Wrangler and OpenRefine facilitate a larger class of data wrangling tasks than Morpheus, but they do not automatically synthesize table transformations from examples.

5.2.3 API completion

Code completion refers to the generation of small code snippets involving API calls [47, 49, 68, 112, 94, 54, 82, 91, 46, 126]. While the line between component-based synthesis and API completion is rather blurry, code-completion tools typically expect a partial program and provide a ranked list of (single-line) completions. Hence, code snippets generated by API completion tools are typically much simpler compared to synthesis tools.

InSynth is a recent API-completion tool that uses theorem proving to compute type inhabitants [47, 49]. While InSynth handles higher-order functions and polymorphism quite elegantly, it cannot synthesize multi-statement

code snippets that involve impure functions. As discussed in Section 2.9, InSynth can only synthesize one example out of the 30 benchmarks used in our evaluation.

Another recent code-completion tool is SLANG [91] which predicts probabilities of API calls using statistical methods. Because SLANG is based on machine learning, it requires training data and is therefore only applicable when the target API has a significant number of clients. However, we believe that the SLANG approach is complementary to ours. In particular, we could use a SLANG-like approach to prioritize some reachable paths in the Petri net over others.

Our approach is also related to type-directed completion, in which users issue queries using partial expressions [82]. An example of such a partial expression is `?(img, size)`, which queries for API components that are likely to use variables `img` and `size`. While extremely useful in IDEs, this approach can only synthesize single-line code snippets rather than entire methods.

Another tool that is related to automated API completion is MatchMaker, which synthesizes "glue code" to allow framework classes to interact with each other [126]. Unlike SyPet where the query is a method signature, MatchMaker queries are of the form "How can I get type `A` and type `B` to interact with each other?" Because MatchMaker uses dynamic traces, the techniques underlying this tool are very different from SyPet.

5.3 Search strategies

In this section, we discuss common search strategies that are widely adopted by mainstream synthesizers.

5.3.1 Type-directed search

Our approach in SyPet also resembles prior work that has framed synthesis as type inhabitation [48, 80, 33, 84]. Of these approaches, InSynth [48] is type-directed rather than example-directed. Myth [80] and its successors [33] cast type- and example-directed synthesis as type inhabitation in a refinement type system. In contrast to these techniques, our approach only enumerates type inhabitants in the context of sketch completion and uses table contents to finitize the universe of constants.

Another work that is closely related to SyPet is Synquid [84], which takes advantage of recent advances in polymorphic refinement types [93, 116]. Similar to our approach, Synquid also adopts a type-directed SMT-based deduction system to prune its search space. However, unlike our system which can work with any incomplete (over-approximate) specification, Synquid requires precise specifications of the underlying components. In other words, Synquid fails to synthesize the desired program if the component specifications are over-approximate. Since it is difficult to write precise specifications of many library methods, we believe that SyPet's ability to perform lightweight deduction using incomplete specifications can be useful in many different contexts.

5.3.2 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 [77, 76]. 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) [111]. 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 [74], and DeepCoder uses a deep neural network to learn features that can be used to make such predictions [13]. 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 [89]. Later work by Raychev et al. extends this approach to the case of program synthesis with noisy input-output examples [86], and Feng et al. use a similar n-gram model for synthesizing table transformations [26]. 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.

5.4 Pruning techniques

In this section, we elaborate recent techniques that are used to prune search space.

5.4.1 Logical reasoning.

The technique proposed in NEO 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 [42, 58, 37, 113]. For example, Brahma [58] 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 [37] 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 [30, 84, 26]. For example, SYNQUID uses liquid type specifications to avoid the exploration of some program terms. Other tools, such as λ^2 [30] and MORPHEUS [26], 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 [121], 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.

5.4.2 Connections to Theorem Prover

5.4.2.1 CEGIS

Counterexample-guided inductive synthesis (CEGIS) is a popular framework for synthesizing programs that satisfy a given specification Φ [9, 103, 102]. 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 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 in NEO 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, ANALYZECONFLICT learns new lemmas that typically rule out many additional programs.

5.4.2.2 Conflict-driven learning

Our synthesis framework in NEO is directly inspired by the success of CDCL-style SAT and SMT solvers [72, 127, 18, 36]. 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, Neo 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.

Chapter 6

Conclusion and Future Work

Program synthesis is a well-known technique to automate complex APIs usage, and it provides powerful search algorithms to look for executable programs that satisfy a given specification (input-output examples, partial programs, formal specs, etc). However, the biggest barrier to a practical synthesizer is the size of search space, which increases strikingly fast with the complexity of the programs and the number of building blocks (i.e., methods of an API). In this dissertation, we address the space explosion by combining the power of statistical models and logical reasoning. Specifically, we first developed a type-directed graph reachability algorithm in SyPet, a synthesizer for assembling programs from complex APIs. We also introduced Morpheus, a lightweight constraint-based synthesizer for automating real-world data wrangling tasks from examples. Finally, we generalized previous approaches and developed a novel conflict-driven algorithm in a system called Neo, which can learn from past mistakes and seamless incorporate state-of-the-art statistical models.

We systematically evaluated our tools on non-trivial benchmarks collected from online forums such as *Stackoverflow* and other publicly available

data sources (e.g. Github, DeepCoder). Our results demonstrated that our systems can efficiently synthesize programs that automate real-world complex tasks.

Moving forward, we believe here are some future directions to improve the systems presented in this dissertation.

Overfitting. While input-output examples are easy to specify for users, the quality of the examples affect the performance and output of the synthesizer. Since this form of specification is very under-constrained, the synthesizer may generate a solution that is consistent with the examples but is not the desired solution. One naive approach is to keep asking the user for more examples until the system converges, but a more interesting direction would be to achieve consensus with minimal number of examples using Generative adversarial networks (GAN).

Specification inference. To prune the search space, existing synthesizers rely heavily on the granularity of the specifications for the underlying components (i.e., API methods). Typically, these specifications require a substantial amount of manual effort to write, and the difficulty only increases as the number and complexity of components grows. Another direction is to investigate techniques that can automatically discover specifications for complex APIs.

Scalability. There are many directions for further improving the scalability of existing synthesizers. One approach is to design synthesis algorithms that are friendly to parallelism. Inspired by the success of modular verification

in program analysis, devising modular synthesis algorithm that can break a complex task into multiple subtasks will also be a compelling direction.

Appendices

Appendix A

Proofs of Theorems

A.1 SyPet

A.1.1 Proof of Theorem 1.

Proof. Let M' be a configuration (marking) such that $M'(p) > k+1$ for some place p, and suppose that $t' = T_1, T_2, \ldots, T_n$ is a sequence of transitions that can be fired starting from M'. Now, let M be another configuration such that $M(p) = k+1$ and for all other p', $M'(p) = M(p)$. We will show that trace $t = T_1, \kappa^{c_1}, T_2, \ldots, \kappa^{c_n}, T_n$ can be fired from M, where each c_i is the number of p tokens consumed by T_i and κ denotes a generic clone transition. Because we consider these two paths t, t' to be equivalent, this property implies any trace that can be generated from M' can also be generated from M.

We will prove this claim using induction, using the following (strengthened) inductive hypothesis. If $T_1, \ldots, T_i$ is reachable from M' then we can conclude that: (i) $T_1, \kappa^{c_1}, \ldots, T_i, \kappa^{c_i}$ is reachable from M and (ii) $M_i(p) = k+1$ and $M_i(p') = M'_i(p')$ for all $p' \neq p$. (Here, we use M_i to denote the marking right before transition T_{i+1}.)

For the base case, we have $i = 1$. Because T_1 is reachable from M' in one step, we can fire T_1 in M'. Now let $p_1, \ldots, p_m$ be the predecessors of T_1

with edge weights $w_1, \ldots, w_m$. For any $p_i \neq p$, M has the same number of tokens as M'. Furthermore, if $p_i = p$, then $w_i \leq k$. Hence, T_1 is also enabled at M. Furthermore, we have at least 1 token left at p after taking transition T_1, so the clone transition remains enabled after T_1. Because the clone transition does not decrease the number of tokens, it remains enabled, so we can execute it as many times as we want. Hence if T_1 is reachable from M', then T_1, κ^{c_1} is reachable from M.

Now, we'll prove property (ii) for the base case. Suppose transition t consumed c_1 number of p tokens. Right before T_2, we still have $k+1$ tokens at p because we fired c_1 clone transitions. Furthermore, for all other places p', the number of tokens remains the same because they were the same in M, M' and we took the same transition T_1 in both traces.

For the inductive step, we show the property for $i+1$. Suppose we take transition T_{i+1} in t'. By the inductive hypothesis, we know:

1. $T_1, \kappa^{c_1}, \ldots, T_i, \kappa^{c_i}$ is a prefix of t and
2. $M'_i(p') = M_i(p')$ for $p' \neq p$ and $M_i(p) = k+1$

Observe that if T_{i+1} is enabled at M'_i, then it must also be enabled at M_i using (2) and the same reasoning as in the base case. Furthermore, we will have at least one p token left after executing T_{i+1}, so the clone transition is again enabled. Now, we execute as many clones as T_{i+1} consumed p tokens, so $M_{i+1}(p)$ will remain $k+1$. For all other places p', we still have $M_{i+1}(p') = M'_{i+1}(p')$

because they were initially the same, and T_{i+1} consumed an equal number of tokens. □

Proof of Theorem 2.

Proof. Let p be any path that starts at marking M and ends at M' in $\mathcal{R}(\mathcal{N})$. We will prove that $M'(\tau^*) > 0$ for some place $\tau^* \neq \texttt{void}$ that is reachable from τ' in $\alpha(\mathcal{R}(\mathcal{N}))$. Because τ^* is reachable from τ' in $\alpha(\mathcal{R}(\mathcal{N}))$, we have $\tau' \neq \tau$. Furthermore, because M^* must assign 0 to τ^*, this property implies that no path starting at M can end in M^*.

The proof is by induction on the length of path p. For the base case, we have $\mathit{length}(p) = 0$ (i.e., $M' = M$). Because $M(\tau') > 0$ and because τ' is reachable from itself, the property holds in the base case.

For the inductive step, let us consider a path p of length $k+1$ that ends in M'', and let p' be the prefix of p of length k. By the inductive hypothesis, p' ends in a marking such that $M'(\tau^*) > 0$ for some place τ^* reachable from τ' in $\alpha(\mathcal{R}(\mathcal{N}))$. There are two possibilities: We either fire a transition f that (i) has τ^* as its predecessor or (ii) does not have τ^* as its predecessor. In the latter case, $M''(\tau') > 0$ because we did not consume any tokens of τ^*, so the property holds. For case (i), f consumes at least one token of τ^* but produces at least one token at some other place τ'', so we have $M'(\tau'') > 0$. Because τ'' is reachable from τ^*, it is also reachable from τ' in $\alpha(\mathcal{R}(\mathcal{N}))$. Furthermore, τ'' cannot be `void`; otherwise, this would imply that τ is reachable from τ'

in $\alpha(\mathcal{N})$ because every type is reachable from `void`. Because we have shown that $M'(\tau'') > 0$ for some $\tau'' \neq$ `void`, the property also holds in the inductive step. $\square$

Bibliography

[1] Morpheus. https://utopia-group.github.io/morpheus/. Accessed 27-Mar-2017.

[2] Motivating Example 1. http://stackoverflow.com/questions/30399516/complex-data-reshaping-in-r. Accessed 27-Mar-2017.

[3] Motivating Example 2. http://stackoverflow.com/questions/33207263/finding-proportions-in-flights-dataset-in-r. Accessed 27-Mar-2017.

[4] Motivating Example 3. http://stackoverflow.com/questions/32875699/how-to-combine-two-data-frames-in-r-see-details. Accessed 27-Mar-2017.

[5] SyPet. http://fredfeng.github.io/sypet/.

[6] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *CAV*, pages 934–950. Springer-Verlag, 2013.

[7] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive

Program Synthesis. In *Proc. International Conference on Computer Aided Verification*, pages 934–950. Springer, 2013.

[8] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–8, 2013.

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

[10] Rajeev Alur, Robert K Brayton, Thomas A Henzinger, Shaz Qadeer, and Sriram K Rajamani. Partial-order reduction in symbolic state space exploration. In *CAV*, pages 340–351. Springer, 1997.

[11] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Inf. Process. Lett.*, 8(3):121–123, 1979.

[12] Brendan Avent, Anna Ritz, and T. Murali. halp: Hypergraph Algorithms Package. `http://murali-group.github.io/halp/`.

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

[14] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin G. Zorn. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. In *Proc. Conference on Programming Language Design and Implementation*, pages 218–228. ACM, 2015.

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

[16] Tewodros A. Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. A constraint-based approach to solving games on infinite graphs. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 221–234, 2014.

[17] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. PHOG: probabilistic model for code. In *Proc. International Conference on Machine Learning*, pages 2933–2942. ACM, 2016.

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

[19] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. Sniff: A search engine for java using free-form queries. In *FASE*, pages 385–400. Springer-Verlag, 2009.

[20] Tamraparni Dasu and Theodore Johnson. *Exploratory data mining and data cleaning*, volume 479. John Wiley & Sons, 2003.

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

[22] Javier Esparza and Keijo Heljanko. *Unfoldings - A Partial-Order Approach to Model Checking*. Springer, 2008.

[23] Patrick Fabiani and Yannick Meiller. Planning with tokens: an approach between satisfaction and optimisation. In *PuK*, pages 26–35, 2000.

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

[25] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. Program synthesis using conflict-driven learning. In *Proc. Conference on Programming Language Design and Implementation*, pages 420–435. ACM, 2018.

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

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

[28] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-Based Synthesis for Complex APIs. Technical report, University of Texas at Austin, 2016.

[29] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, pages 229–239. ACM, 2015.

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

[31] Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, pages 189–208, 1971.

[32] Alain Finkel. *The minimal coverability graph for Petri nets*. Springer, 1993.

[33] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *Proc. Symposium on Principles of Programming Languages*, pages 802–815. ACM, 2016.

[34] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and

Koushik Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *ICSE*, pages 653–663. ACM, 2014.

[35] Giorgio Gallo, Giustino Longo, and Stefano Pallottino. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42(2):177–201, 1993.

[36] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): fast decision procedures. In *Proc. International Conference on Computer Aided Verification*, pages 175–188. Springer, 2004.

[37] Adrià Gascón, Ashish Tiwari, Brent Carmer, and Umang Mathur. Look for the proof to find the program: Decorated-component-based program synthesis. In *Proc. International Conference on Computer Aided Verification*, pages 86–103. Springer, 2017.

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

[39] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330. ACM, 2011.

[40] Sumit Gulwani, William R Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[41] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Component based synthesis applied to bitvector circuits. Technical Report MSR-TR-2010-12, February 2010.

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

[43] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73. ACM, 2011.

[44] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In *PLDI*, pages 50–61. ACM, 2011.

[45] Philip J. Guo, Sean Kandel, Joseph M. Hellerstein, and Jeffrey Heer. Proactive Wrangling: Mixed-initiative End-user Programming of Data Transformation Scripts. In *Proc. Symposium on User Interface Software and Technology*, pages 65–74. ACM, 2011.

[46] Tihomir Gvero and Viktor Kuncak. Synthesizing Java expressions from free-form queries. In *OOPSLA*, pages 416–432, 2015.

[47] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *PLDI*, pages 27–38, 2013.

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

[49] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Interactive synthesis of code snippets. In *CAV*, pages 418–423, 2011.

[50] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. Interactive synthesis of code snippets. In *CAV*, pages 418–423, 2011.

[51] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *PLDI*, pages 317–328. ACM, 2011.

[52] Keijo Heljanko. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe petri nets. *Fundamenta Informaticae*, pages 247–268, 1999.

[53] Sarah L. Hickmott, Jussi Rintanen, Sylvie Thiébaux, and Langford B. White. Planning via petri net unfolding. In *IJCAI*, pages 1904–1911. AAAI Press, 2007.

[54] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *ICSE*, pages 117–125. ACM, 2005.

[55] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Strathcona example recommendation tool. In *ESEC/FSE*, pages 237–240. ACM, 2005.

[56] Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. Jsketch: Sketching for Java. In *ESEC/FSE*, pages 934–937. ACM, 2015.

[57] Susmit Jha, Sumit Gulwani, Sanjit Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pages 215–224. IEEE, 2010.

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

[59] Troy A. Johnson and Rudolf Eigenmann. Context-sensitive domain-independent algorithm composition and selection. In *Proc. Conference on Programming Language Design and Implementation*, pages 181–192. ACM, 2006.

[60] Rajeev Joshi, Greg Nelson, and Keith H. Randall. Denali: A goal-directed superoptimizer. In *PLDI*, pages 304–314. ACM, 2002.

[61] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proc. International Conference on Human Factors in Computing Systems*, pages 3363–3372. ACM, 2011.

[62] Richard M Karp and Raymond E Miller. Parallel program schemata. *Journal of Computer and System Sciences*, pages 147–195, 1969.

[63] Emanuel Kitzelmann. A combined analytical and search-based approach for the inductive synthesis of functional programs. *Künstliche Intelligenz*, 25(2):179–182, 2011.

[64] Donald E. Knuth. A generalization of dijkstra's algorithm. *Inf. Process. Lett.*, 6(1):1–5, 1977.

[65] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Complete functional synthesis. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 316–329, 2010.

[66] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. Programming by demonstration using version space algebra. *Mach. Learn.*, pages 111–156, 2003.

[67] Vu Le and Sumit Gulwani. FlashExtract: a framework for data extraction by examples. In *Proc. Conference on Programming Language Design and Implementation*, pages 542–553. ACM, 2014.

[68] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI*, pages 48–61. ACM, 2005.

[69] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proc. Conference on*

Programming Language Design and Implementation, pages 48–61. ACM, 2005.

[70] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 48–61, 2005.

[71] Ravi Mangal, Xin Zhang, Aditya V Nori, and Mayur Naik. A user-guided approach to program analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 462–473. ACM, 2015.

[72] Joao Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[73] Kenneth L McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *CAV*, pages 164–177. Springer, 1993.

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

[75] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

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

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

[78] Lars Relund Nielsen, Kim Allan Andersen, and Daniele Pretolani. Finding the K shortest hyperpaths. *Computers & OR*, 32:1477–1497, 2005.

[79] Shougo Ogata, Tatsuhiro Tsuchiya, and Tohru Kikuno. SAT-based verification of safe petri nets. In *ATVA*, pages 79–92. Springer, 2004.

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

[81] Doron Peled. Ten years of partial order reduction. In *CAV*, pages 17–28. Springer, 1998.

[82] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *PLDI*, pages 275–286. ACM, 2012.

[83] Daniel Perelman, Sumit Gulwani, Dan Grossman, and Peter Provost. Test-driven synthesis. In *PLDI*, page 43. ACM, 2014.

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

[85] Oleksandr Polozov and Sumit Gulwani. FlashMeta: A framework for inductive program synthesis. In *Proc. International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126. ACM, 2015.

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

[87] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from big code. In *POPL*, volume 50, pages 111–124. ACM, 2015.

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

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

[90] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *PLDI*, volume 49, pages 419–428. ACM, 2014.

[91] Veselin Raychev, Martin T. Vechev, and Eran Yahav. Code completion with statistical language models. In *PLDI*, page 44. ACM, 2014.

[92] Alex Reinking and Ruzica Piskac. A type-directed approach to program repair. In *CAV*, pages 511–517, 2015.

[93] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Proc. Conference on Programming Language Design and Implementation*, pages 159–169. ACM, 2008.

[94] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: Mining for sample code. In *OOPSLA*, pages 413–430. ACM, 2006.

[95] Gerard Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, pages 613–620, 1975.

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

[97] Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. *PVLDB*, pages 740–751, 2012.

[98] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, pages 634–651. ACM, 2012.

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

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

[101] Armando Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, 2008.

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

[103] Armando Solar-Lezama. The sketching approach to program synthesis. In *Proc. Asian Symposium on Programming Languages and Systems*, pages 4–13. Springer, 2009.

[104] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodík, Vijay A. Saraswat, and Sanjit A. Seshia. Sketching stencils. In *PLDI*, pages 167–178. ACM, 2007.

[105] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, pages 281–294. ACM, 2005.

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

[107] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415. ACM, 2006.

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

[109] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Template-based program verification and program synthesis. *STTT*, 15(5-6):497–518, 2013.

[110] Andreas Stolcke et al. Srilm-an extensible language modeling toolkit. In *Interspeech*, volume 2002, 2002.

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

[112] Suresh Thummalapenta and Tao Xie. Parseweb: A programmer assistant

for reusing open source code on the web. In *ASE*, pages 204–213. ACM, 2007.

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

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

[115] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, page 13, 1999.

[116] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for typescript. In *Proc. Conference on Programming Language Design and Implementation*, pages 310–325. ACM, 2016.

[117] Ruben Verborgh and Max De Wilde. *Using OpenRefine*. Packt Publishing Ltd, 2013.

[118] Walter Vogler. Efficiency of asynchronous systems and read arcs in petri nets. In *ICALP*, pages 538–548. Springer, 1997.

[119] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proc. Conference*

on Programming Language Design and Implementation, pages 452–466. ACM, 2017.

[120] Xinyu Wang, Isil Dillig, and Rishabh Singh. Synthesis of Data Completion Scripts using Finite Tree Automata. *Proc. International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2017.

[121] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *Proc. Symposium on Principles of Programming Languages*, 2018.

[122] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott A. Mahlke. The theory of deadlock avoidance via discrete control. In *POPL*, pages 252–263. ACM, 2009.

[123] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proc. Conference on Programming Language Design and Implementation*, pages 508–521. ACM, 2016.

[124] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proc. Conference on Programming Language Design and Implementation*, pages 508–521. ACM, 2016.

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

[126] Kuat Yessenov, Zhilei Xu, and Armando Solar-Lezama. Data-driven synthesis for object-oriented frameworks. In *OOPSLA*, pages 65–82. ACM, 2011.

[127] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *Proc. of International Conference on Computer-Aided Design*, pages 279–285. IEEE Computer Society, 2001.

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

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