Program Synthesis: The Book

Loris D'Antoni, Nadia Polikarpova

September 6, 2022

An Awesome Publisher

Contents

List of Figures

List of Tables

What is Program Synthesis? 1

Program synthesis is the task of automatically finding a program within a given search space (i.e., a set of programs provided) that meets a given specification (i.e., a user intent specified typically by a logical formula or a set of input-output examples). For example, we can imagine that one provides a search space consisting of all recursive functional programs, and a set of input-output examples where the input is potentially an unsorted list of integers, and the output is the input sorted. The goal of the synthesizer is to come up with a recursive function that sorts such a list. What makes synthesis particularly hard is that there is no direct mechanical syntax-directed way to "translate" the user-intent into the desired program. This aspect is what differentiates synthesis from traditional compilation techniques where one can map a program in high-level language to a low-level language by recursively modifying the input program one piece at a time.

The first program synthesizers emerged in the logic-programming literature [1] in 1969, where constraint solving techniques were modified to yield programs for certain types of specification. Shortly after, several papers started tackling various program synthesis techniques at a theoretical level: inverting programs [2], and generating reactive controllers that satisfy temporal constraints [3]. After these early attempts, researchers essentially got scared by how hard of a problem synthesis was, and programs synthesis somewhat disappeared from the research landscape and got reduced to mostly a theoretical topic.

The hiatus of research in program synthesis ended in the early 2000s when a number of independent pieces of work showed that synthesis could be made practical by taking advantage of the giant leaps in automatic constraint solving techniques [4] and by carefully restricting the synthesis problems to enable new clever search techniques [5]. What was particularly exciting about this work was that they tackled industrial applications—for e.g., automating spreadsheet manipulations in Microsoft Excel [5]. Once the door was cracked open, the excitement quickly ramped up and many new ideas started emerging, which in turn led to synthesis making an impact in a variety of domains, such as Computer Networks [6], Visualization [7], and Databases [8], etc. This textbook is an attempt to summarize what has happened in program synthesis research since 2000 while making the topic accessible to a wider audience. The book is also accompanied by code that shows how different techniques can be implemented (`https://github.com/nadia-polikarpova/synthesis-book-code`).

1.1 Dimensions in Program Synthesis

We are now ready to informally define what a synthesis problem is.

Definition 1.1.1 (The synthesis problem) *Given a specification φ and* *a search space S, find a program p in the search space S that meets the specification φ.*

Throughout this book, we will explore different variations of the synthesis problem and many techniques for solving it. Our hope is that the reader can learn to see the common themes and relationships between these techniques. To help study the techniques more systematically, we will classify them along *three dimensions*, originally introduced by Gulwani [9]:

1. *Behavioral specification:* How do we explain to the synthesizer what the target program is supposed to do? Popular kinds of behavioral specifications are input-output examples, assertions, types, or even natural language descriptions. Formally, behavioral specification defines the language of specifications φ in the synthesis problem.
2. *Structural specification:* What is the space of programs that the synthesizer considers? This space can be either built into the synthesis algorithm, or may be defined by the user, for example, in the form of a grammar or a library of functions to use.
3. *Search strategy:* The algorithm used to explore the space of programs. Popular search strategies include exhaustive enuemration, stochastic search, and delegating the search to constraint solvers.

Example 1.1.1 An example of (potentially very hard to solve) synthesis problem is one where: 1) The behavioral specification is a logical formula φ stating that the program should take as input a list of input l and return a sorted version of l. 2) The search space—i.e., the structural specification—is the set of all valid C programs.

A possible search strategy is one that enumerates all strings in lexicographic order and checks whether any of them is a valid C program that satisfies the specification.

While this simplistic problem is somewhat unrealistic, it already highlights many problems we will have to deal with in this book, such as constraining our search space to be more tractable to search over and perhaps co-designing with a search strategy that can take advantage of the structural specification.

At this point the reader might be wondering what sets program synthesis apart from techniques used in machine learning (such as training neural networks), which also can be thought to generate functions according to some constraints or objectives. The main differentiating factor is that the search space in program synthesis is more complex and customizable. With a structural specification, a user (or an algorithm designer) can easily embed their domain knowledge into the synthesizer, which in turn enables the synthesizer to learn from very little data. The downside, of course, is that exploring such complex spaces requires combinatorial search (e.g., enumerating programs), and cannot be done using the efficient gradient descent techniques that only work over differentiable search spaces (e.g., neural networks) with differentiable objectives (e.g., norms).

Parts I, II, and III of this book are entirely dedicated at the pure algorithmic parts of program synthesis, that is, how do we build effective tools for solving synthesis problems.

Synthesis and Verification Before we dive deep into the synthesis world, some readers well-versed in formal methods might notice that the problem we are setting ourselves to solve is a hard one. In particular, even program verification, that is, checking that a given program meets a given specification is an undecidable problem—i.e., we cannot build a tool that always succeeds at solving this problem. Throughout the book we will often cover verification topics as part of our journey. In fact, while synthesis is strictly harder than verification, these two problems are very much related and many synthesis techniques build upon verification ones and vice-versa. For example, we will show how type systems (a common verification technique) can inform synthesizers in how to search complex search spaces. On the other hand, we will look at how synthesis can be used to generate loop invariants, the key ingredient for successful program verification.

Using Synthesis in Practice While the problem of synthesizing a program feels entirely algorithmic, every programming language needs a human input to work. Interacting with a user is a challenge—the tool should be easy to use—but also an opportunity. In particular, expert users can provide hints and insights that can guide the search space of a synthesizer, whereas end users—i.e., those trying to come up with a program—can provide hints and examples to help the synthesizer quickly solve one specific synthesis problem.

Part IV of the book is dedicated to the human aspects of program synthesis.

Figure 1.1 summarizes our view of program synthesis.

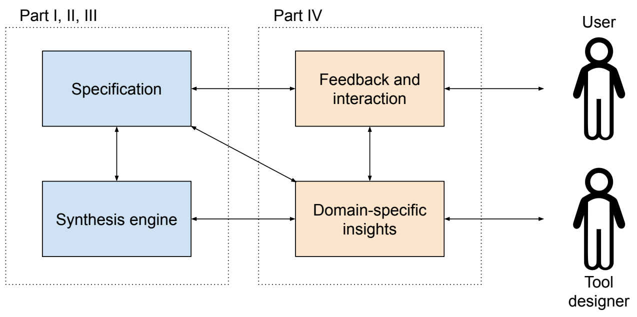

Figure 1.1: Outline of synthesis

1.2 Applications of Program Synthesis

Throughout the book, we will present our ideas using, among others, the following well-known applications of program synthesis.

Data wrangling The goal is to automatically generate programs that can format data based on input-output examples. For example, extracting the initials from a string containing someone's names or normalizing some date formats. This was the first industrial application of program synthesis, and synthesizers for this domain can nowadays be found in Microsoft Excel [9] among other tools.

Superoptimization Given an inefficient function over bit-vectors (or some other type), the goal is to automatically generate a fast bit-vector function that is equivalent to the provided one. This technique can be used by a compiler to optimize simple pieces of code.

1.2.1 Organization

This book is a work-in-progress and in constant change and some parts will be added once completed. Our intended outline is to have four parts. Part I covers the problem of synthesizing expressions (i.e., loop-free programs). Part II covers the problem of synthesizing complex programs (i.e., imperative and functional programs). Part III covers complex specifications involving quantitative semantic objectives (e.g., complexity metrics and probabilistic objectives). Part IV covers how synthesis can be improved by allowing a user to interact with a synthesizer.

Part I

Synthesizing expressions

2 Synthesizing Expressions: Two Simple Domains

In this chapter, we introduce two domains where synthesis has been successful: data-wrangling and bit-vector transformations. These domains are a good start for our journey because they contain programs that do not manipulate states and that do not use control-flow constructs—i.e., they only use expressions.

2.1 The Expression-Synthesis Problem

An expression is a program for which the semantics can be computed using a syntactic derivation that has the same size as the expression. Most importantly, each sub-expression can be evaluated independently from other sub-expressions and the semantics of each expression only depends on the overall input to the program.

When defining a synthesis problem, one has to provide two inputs: a search space (a set of programs we can choose from) and a behavioral specification (what the synthesized program should do).

2.1.1 Search Spaces as Grammars

Although the search space can be described in a number of ways, an elegant formalism that encompasses many approaches is to define a simple language of programs using a grammar. Because we are interested in defining programs at an expression level (and not at a string level), instead of using the more common formalism of context-free grammars (which operate over strings), we present regular tree grammars, which directly describe the syntax trees of the expressions.

A *ranked alphabet* is a tuple $(\Sigma, \mathsf{rk}_\Sigma)$ where Σ is a finite set of symbols and $\mathsf{rk}_\Sigma : \Sigma \rightarrow \mathbb{N}$ associates a rank to each symbol. For every $m \geq 0$, the set of all symbols in Σ with rank m is denoted by $\Sigma^{(m)}$. In our examples, a ranked alphabet is specified by showing the set Σ and attaching the respective rank to every symbol as a superscript—e.g., $\Sigma = \{+^{(2)}, c^{(0)}\}$. (For brevity, the superscript is sometimes omitted). We use $\mathcal{T}_\Sigma$ to denote the set of all (ranked) trees over Σ—i.e., $\mathcal{T}_\Sigma$ is the smallest set such that (*i*) $\Sigma^{(0)} \subseteq \mathcal{T}_\Sigma$, (*ii*) if $\sigma^{(k)} \in \Sigma^{(k)}$ and $t_1, \ldots, t_k \in \mathcal{T}_\Sigma$, then $\sigma^{(k)}(t_1, \cdots, t_k) \in \mathcal{T}_\Sigma$. In the context of program synthesis, we refer to trees from $\mathcal{T}_\Sigma$ as *terms*, *expressions*, or *programs*. In what follows, we assume a fixed ranked alphabet $(\Sigma, \mathsf{rk}_\Sigma)$.

We focus on *typed* regular tree grammars, in which each nonterminal and each symbol is associated with a type. There is a finite set of types $\{T_1, \ldots, T_k\}$. Associated with each symbol $\sigma^{(i)} \in \Sigma^{(i)}$, there is a type assignment $a_{\sigma^{(i)}} = (T_0, T_1, \ldots, T_i)$, where T_0 is called the *left-hand-side type* and $T_1, \ldots, T_i$ are called the *right-hand-side types*. Tree grammars are similar to word grammars, but generate trees over a ranked alphabet instead of words.

Definition 2.1.1 (Regular Tree Grammar) *A* **typed regular tree grammar** *(RTG) is a tuple $G = (N, \Sigma, S, a, \delta)$, where N is a finite set of nonterminal symbols of arity 0; Σ is a ranked alphabet; $S \in N$ is an initial nonterminal; a is a type assignment that gives types for members of $\Sigma \cup N$; and δ is a finite set of productions of the form $A_0 \longrightarrow \sigma^{(i)}(A_1, \ldots, A_i)$, where for $1 \leq j \leq i$, each $A_j \in N$ is a non-terminal such that if $a(\sigma^{(i)}) = (T_0, T_1, \ldots, T_i)$ then $a(A_j) = T_j$.*

We often refer to tree grammars simply as grammars. Trees that can contain both alphabet and nonterminal symbols, $\tau \in \mathcal{T}_{\Sigma \cup N}$, are called *incomplete terms*. Given an incomplete term $\tau \in \mathcal{T}_{\Sigma \cup N}$, applying a production $r = A \longrightarrow \beta$ to τ—denoted $\tau[r]$—produces another incomplete term τ' resulting from replacing the left-most occurrence of A in τ' with the production's right-hand side β. In this case we also say that τ' can be *derived in one step* from τ using r, denoted $\tau \longrightarrow_G \tau'$ (where the grammar G can be omitted when it is clear from the context). The reflexive and transitive closure of the one-step derivation relation is denoted $\tau \longrightarrow_G^* \tau'$; in other words, a term τ' can be *derived* from τ iff τ' can be obtained by applying a sequence of productions $r_1 \cdots r_n$ to τ. We say that a complete term $t \in \mathcal{T}_\Sigma$ (i.e., a term without nonterminals) is *generated* by the grammar G—denoted by $t \in L(G)$—iff it can be derived from the initial non-terminal S, *i.e.* $S \longrightarrow_G^* t$.

In general, it can be cumbersome to use the notation $\sigma^{(i)}(A_1, \ldots, A_i)$ for the productions of a grammar. In the rest of the book, we often relax this notation by either avoiding explicitly presenting parenthesis or by allowing infix operations as shown in the following example.

Example 2.1.1 The following grammar G_{CI} describes conditional integer expressions that may contain one variable x:

$$\begin{aligned} S &::= \texttt{x} \mid \texttt{0} \mid S + \texttt{1} \mid \texttt{ite}(B, S, S) \\ N &::= \texttt{x} = S \end{aligned}$$

In the grammar, `ite` is the if-then-else operator. For example, this grammar can produce the tree: $\texttt{ite}(\texttt{x} = \texttt{0} + \texttt{1}, \texttt{0}, \texttt{x})$.

Semantics of Terms While we have shown how to describe sets of trees (i.e., the programs in our language), we also need to give these trees a meaning (i.e., what the programs output when fed an input). In general, an input ϵ to an expression is a map $[\overline{x \mapsto v}]$, *i.e.* an environment/store that maps variables to values. We write $\epsilon(x)$ to denote the value of the variable x in the map. Then, given an expression t, its semantics $[\![t]\!]$ is function from environments to values.

Example 2.1.2 We define the semantics of the expressions produced by the grammar G_{CI} in example 2.1.1. For this language, an input ϵ is simply a map $[x \mapsto v_x]$ that assigns a value to the variable x. We define the semantics inductively one constructor at a time:

- $[\![\texttt{x}]\!](\epsilon) = \epsilon(x)$, *i.e.* the semantics of an expression x is the value of x in the environment;

- $[\![\texttt{0}]\!](\epsilon) = 0$;
- $[\![t + \texttt{1}]\!](\epsilon) = [\![t]\!](\epsilon) + 1$, *i.e.* the semantics of $t + \texttt{1}$ is the semantics of t on the same input plus 1;
- $[\![\texttt{ite}(t_b, t_1, t_2)]\!](\epsilon) = v$ where if $[\![t_b]\!](\epsilon)$ is true then $v = [\![t_1]\!](\epsilon)$, otherwise $v = [\![t_2]\!](\epsilon)$.
- $[\![\texttt{x} = t]\!](\epsilon)$ is true if $\epsilon(x)$ is equal to $[\![t]\!](\epsilon)$ and false otherwise.

Given the expression $\texttt{ite}(\texttt{x} = \texttt{0} + \texttt{1}, \texttt{0}, \texttt{x})$ and an example $\epsilon_1 = [x \mapsto 1]$, we have that $[\![\texttt{ite}(\texttt{x} = \texttt{0} + \texttt{1}, \texttt{0}, \texttt{x})]\!](\epsilon_1) = 0$ because $[\![\texttt{0} + \texttt{1}]\!](\epsilon_1) = 1$, which implies that $[\![\texttt{x} = \texttt{0} + \texttt{1}]\!](\epsilon_1) = $ true, which implies that $[\![\texttt{ite}(\texttt{x} = \texttt{0} + \texttt{1}, \texttt{0}, \texttt{x})]\!](\epsilon_1) = [\![\texttt{0}]\!](\epsilon_1) = 0$.

2.1.2 Behavioral Function Specification

In an expression-synthesis problem, the goal is to synthesize a function f that not only belongs to the given grammar, but also satisfies a behavioral specification. In general, the behavioral specification can be given in a number of ways—e.g., input-output examples or logical formula. In this section, we provide a simple theoretical framework to reason about behavioral specifications that will be useful in the rest of the book.

First, we make a simplifying assumption that we will relax later in the book: *All our specifications are functional*[1]—i.e., for every input i there exists at most one output o allowed by the specification. For example, consider the formulas $\phi_{ID} = \forall x.f(x) = x$ and $\phi_{GE} = \forall x.f(x) > x$. The formula ϕ_{ID} describes a functional specification for the function f (i.e., the identity) where every input is mapped to exactly one output, while the formula ϕ_{GE} is not functional as any input can be mapped to any output greater than the input.

1: Assumption: All our specifications are functional.

While we have restricted ourselves to functional specifications, we still haven't given a way for a user to provide one. To avoid tying ourselves to a particular formalism, for now we define the specification through an interface between the synthesizer and a 'teacher' (also called an oracle [10]). The teacher knows what program we are trying to synthesize and can answer some queries for us through two interfaces:

Membership queries can be used to ask the teacher *What's the output of the target function f on an input i?*

Equivalence queries can be used to ask the teacher whether a conjectured function f^* is correct—*Is f^* the correct target function?* To this query the teacher can answer true or false. In the latter case, the teacher also provides a counterexample input i^*—i.e., an input for which $f^*(i^*)$ is returning an incorrect output.

A teacher that can answer these types of query is often called a minimally adequate teacher (MAT) [11]. While these queries are fairly abstract, we will show in the rest of the section how they can be implemented in practical settings using two examples. In the following example, we use M and E to denote membership and equivalence queries for a teacher T.

2.1.3 Problem Definition

We are now ready to define the expression-synthesis problem. Informally, a synthesis problem is parametric in a teacher T and to solve the problem one has to find a function in the search space (the grammar) that is equivalent to the one the teacher T has in mind.

Definition 2.1.2 (Expression-synthesis problem) *An expression-synthesis problem is a pair $E_{sy} = (\mathsf{T}, G)$ where T is a teacher able to answer membership (M) and equivalence (E) queries and G is a regular tree grammar that only contains expressions.*

An expression-synthesis problem is **realizable** *iff there exists an expression $t \in L(G)$ such that $\mathsf{E}(t) = \mathsf{true}$—we say that t is a solution to the synthesis problem. Otherwise we say that the problem is* **unrealizable**.

Example 2.1.3 Consider the problem of synthesizing an identity expression t_{ID}, *i.e.* $\forall[x \mapsto v].[\![t]\!]([x \mapsto v]) = v$, using only expressions in the grammar G_{CI} from Example 2.1.1.

This problem can be defined as an expression-synthesis problem $E_{sy}^{ID} = (\mathsf{T}_{ID}, G_{CI})$ where the teacher $\mathsf{T}_{ID} = (\mathsf{M}_{ID}, \mathsf{E}_{ID})$ answers queries in the obvious way, *i.e.* $\mathsf{M}_{ID}([x \mapsto v]) = v$ for every value v, and $\mathsf{E}_{ID}(e)$ returns true only when t is an expression computing the identity and false together with an example $[x \mapsto v_c]$ such that $[\![t]\!]([x \mapsto v_c]) \neq v_c$ otherwise.

For example, $\mathsf{E}_{ID}(\mathtt{x})$ returns true as $\mathtt{x}$ is a solution to the synthesis problem E_{sy}^{ID}. Therefore, the problem E_{sy}^{ID} is realizable. However, $\mathsf{E}_{ID}(\mathtt{ite}(\mathtt{x} = \mathtt{0} + \mathtt{1}, \mathtt{0}, \mathtt{x}))$ returns false and the counterexample $[x \mapsto 1]$ because the given expression is incorrect (only) on the input where x is equal to 1.

Finally, consider a slightly different problem $E_{sy}^{double} = (\mathsf{T}_{double}, G_{CI})$ where the goal is synthesize an expression t_{double} such that $\forall[x \mapsto v].[\![t]\!]([x \mapsto v]) = 2v$. The expression-synthesis problem E_{sy}^{double} is unrealizable because no expression t in the grammar G_{CI} can express the desired behavior.

2.1.4 Counterexample-Guided Inductive Synthesis

Now that we have defined what an expression-synthesis problem is, we need a way to solve such problems! In general, synthesis is hard and there is no one-size fits all technique for solving synthesis problems. However, the setting we have just defined, in which a teacher can answer membership and equivalence queries, already allows us to perform some simplifications. In particular, we show that instead of solving the whole synthesis problem in one shot, we can just synthesize an expression that is correct on a given set of input/output examples.

The Counterexample-Guided Inductive Synthesis (CEGIS) algorithm is a popular approach to solving synthesis problems. Instead of directly looking for an expression that satisfies the specification φ on *all* possible inputs, the CEGIS algorithm uses a synthesizer S that can find expressions that are correct on a *finite* set of examples $\mathcal{E}$. If S finds a solution that is correct on all elements of $\mathcal{E}$, CEGIS uses calls to the equivalence oracle E to check if the solution is indeed correct. If not, a counterexample obtained from E is added to the set of examples, and the process repeats. More formally, CEGIS starts with an empty set of examples $\mathcal{E}$ and repeats the following steps:

1. Call the synthesizer S to find an expression t^* such that $\forall \epsilon \in \mathcal{E}. [\![t^*]\!](\epsilon) = \mathsf{M}(\epsilon)$ holds and go to step 2; return *unrealizable* if no expression exists.
2. Call the equivalence oracle $\mathsf{E}(e^*)$ and add a counterexample ϵ_c to $\mathcal{E}$ if the answer is false; return t^* as a valid solution if $\mathsf{E}(t^*) = \mathsf{true}$.

In the following, we use the notation $[\![t]\!](\mathcal{E})$ and $\mathsf{M}(\mathcal{E})$ to denote the vector of outputs corresponding to the inputs in $\mathcal{E}$ (for this purpose we treat $\mathcal{E}$ as an ordered vector rather than an unordered set).

The CEGIS algorithm is sound but incomplete for both synthesis and unrealizability.

Before continuing, we need to define synthesis problems where the specification is given by a finite set of examples.

Definition 2.1.3 (Example-based expression synthesis problem) *An* example-based expression synthesis problem *is a triple* $E_{sy}^{\mathcal{E}} = ((\mathsf{M}, \mathsf{E}_{\mathcal{E}}), G)$.

An example-based expression synthesis problem is **realizable** *iff there exists an expression* $t \in L(G)$ *such that* $\mathsf{E}_{\mathcal{E}}(t) \equiv ([\![t]\!](\mathcal{E}) = \mathsf{M}(\mathcal{E}))$*—we say that* t *is a solution to the synthesis problem. Otherwise we say that the problem is* **unrealizable**.

Given an expression-synthesis problem $E_{sy} = ((\mathsf{M}, \mathsf{E}), G)$ and a finite set of inputs $\mathcal{E}$, we denote with $E_{sy}^{\mathcal{E}} = ((\mathsf{M}, \mathsf{E}_{\mathcal{E}}), G)$ the corresponding example-based expression synthesis problem that only requires the expression t to be correct on the examples in $\mathcal{E}$.

Lemma 2.1.1 (Unrealizability over examples) *If* $E_{sy}^{\mathcal{E}}$ *is unrealizable, then* E_{sy} *is unrealizable.*

Lemma 2.1.2 (Soundness) *If CEGIS terminates, it returns a correct answer.*

Even when given a perfect synthesizer S—i.e., one that can solve a problem $E_{sy}^{\mathcal{E}}$ for every possible set $\mathcal{E}$—there are SyGuS problems for which the CEGIS algorithm is not powerful enough to prove unrealizability.

Lemma 2.1.3 (Incompleteness for Unrealizability) *There exists an unrealizable SyGuS problem* E_{sy} *such that for every finite set of examples* $\mathcal{E}$ *the problem* $E_{sy}^{\mathcal{E}}$ *is realizable.*

However, under reasonable assumptions, the CEGIS algorithm can always find a solution to the synthesis problem if one exists. We say that a synthesizer is *fair* if it returns the smallest (with respect to some total order $\leq$ over expressions) expression that is a solution to a synthesis problem when a solution exists.

Lemma 2.1.4 (Completeness for Synthesis) *If E_{sy} is realizable and CEGIS uses a fair synthesizer with respect to some total order $\leq$ over expressions to solve the generated example-based expression synthesis problems, CEGIS terminates and finds the minimal solution with respect to $\leq$ over expressions.*

A synthesizer that simply enumerates all expressions in some lexicographic order and checks whether they pass the equivalence oracle will be fair.

In the following two sections, we introduce two concrete domains and show how to build a teacher for synthesis problems in such domains. In the rest of the sections, we will show how we can then build synthesizers for solving synthesis problems with examples, which coupled with the teachers and CEGIS can be used to solve problems for such domains.

2.2 Example 1: Data Wrangling

While program synthesis has been studied for many decades, the first industrial success of program synthesis came in 2013, when Sumit Gulwani [5] built FlashFill, a tool for automatically synthesizing string transformations in Excel spreadsheets.

Example 2.2.1 Consider the following examples of input and output strings for a synthesis problem where the goal is to synthesize a string transformation that extracts the initials of someone's first and last names

```
  "Loris␣D'Antoni" ⟶"LD"
"Nadia␣Polikarpova" ⟶"NP"
```

Given these examples, FlashFill outputs a program that correctly extracts the initials. The users of the tool can then run the program on other examples to see if the program is correct.

This domain is interesting because examples are the right specification mechanism—i.e., it is very natural to describe what one wants the program to do using a handful of examples.

Let's try do define more formally what a synthesis problem will look like in this domain. We will need to define an appropriate grammar G_{ff} of what programs we are interested in and then what a teacher T would look like for problems in this domain.

Let's assume for a second that the grammar G_{ff} is given. In this domain, the teacher can be naturally implemented by a human who interacts with the tool. The human can answer membership queries by providing outputs and equivalence queries by checking the synthesized program on some other inputs or by manually inspecting the program.

In the rest of the section, we formally define such components.

2.2.1 Stringy: A Language for String Transformations

We start by presenting the syntax and semantics Stringy, a simple language for describing string transformations.[2]

2: Stringy is different from the language used inside the tool FlashFill: it is simple enough that makes describing synthesis algorithms easier and complex enough to describe complex string transformations.

Syntax Given a string $s = a_0 \cdots a_{j-1}$ with j characters, and two numbers n_1 and n_2, such that $0 \leq n_1 \leq n_2 < j$, we use $s[n_1]$ to denote the character a_{n_1}, and $s[n_1 .. n_2]$ to denote the substring $a_{n_1} \cdots a_{n_2}$.

The following grammar describes the space of Stringy programs:

$$S ::= \mathtt{x} \mid \texttt{"␣"} \mid S \mathbin{+\!\!+} S \mid S[N .. N]$$
$$N ::= \mathtt{0} \mid \mathtt{\$} \mid N+1 \mid N-1 \mid \mathbf{find}(S, S)$$

Programs produced by the nonterminal S evaluate to strings. The variable x is the input string, "␣" is the space character, $s_1 \mathbin{+\!\!+} s_2$ is the concatenation of two strings, and $s[n_1 .. n_2]$ denote the substring of s between indices n_1 and n_2. Programs produced by N define numbers (i.e., string positions). 0 is the index zero, \$ is the length of the string under consideration, $n+1$ and $n-1$ are the next and previous position with respect to n, repspectively, and $\mathbf{find}(s_1, s_2)$ is the starting position of the first occurrence of the substring s_1 in s_2 (-1 if s_1 is not a substring of s_2). For readability, we use the notation $S \mathbin{+\!\!+} S$ instead of $\mathtt{concat}(S, S)$ and $S[N .. N]$ instead of $\mathtt{substr}(S, N, N)$.

Example 2.2.2 Consider again the synthesis problem where the task is to find a program that maps

$$\texttt{"Loris␣D'Antoni"} \longrightarrow \texttt{"LD"}$$
$$\texttt{"Nadia␣Polikarpova"} \longrightarrow \texttt{"NP"}$$

The following Stringy program performs the desired string transformation:

$$\mathtt{x}[\mathtt{0} .. \mathtt{0}+1] \mathbin{+\!\!+} \mathtt{x}[\mathbf{find}(\mathtt{x}, \texttt{"␣"})+1 .. \mathbf{find}(\mathtt{x}, \texttt{"␣"})+1+1]$$

Semantics Now that we have informally described the semantics of a Stringy programs, we can do so more formally. We will use two functions, $[\![\cdot]\!]_S : \mathtt{String} \to \mathtt{String}$ and $[\![\cdot]\!]_N : \mathtt{String} \to \mathbb{N}$ to denote the meaning of programs generated by S and N, respectively. We use *str* to denote the value of our input string. For the Stringy language, we provide our semantics using Scala-like code. For every string s, s_1, s_2 and numbers n, n_1, n_2:

$$\begin{aligned}
[\![\mathtt{x}]\!]_S(str) &= str \\
[\![\texttt{"␣"}]\!]_S(str) &= \texttt{"␣"} \\
[\![s_1 \mathbin{+\!\!+} s_2]\!]_S(str) &= [\![s_1]\!]_S(x){+}{+}[\![s_1]\!]_S(x) \\
[\![s[n_1 .. n_2]]\!]_S(str) &= [\![s]\!]_S(x).\mathtt{substring}([\![n_1]\!]_N([\![s]\!]_S(x)), [\![n_1]\!]_N([\![s]\!]_S(x))) \\
[\![\mathbf{find}(s_1, s_2)]\!]_S(str) &= [\![s_1]\!]_S(str).\mathtt{indexOf}([\![s_2]\!]_S(str))
\end{aligned}$$

$$\begin{aligned}
[\![\texttt{0}]\!]_N(str) &= \quad 0\\
[\![\texttt{\$}]\!]_N(str) &= \quad str\texttt{.length}\\
[\![n+\texttt{1}]\!]_N(str) &= \quad [\![n]\!]_N(str)+1\\
[\![n-\texttt{1}]\!]_N(str) &= \quad [\![n]\!]_N(str)-1
\end{aligned}$$

2.2.2 Specifying Data Extraction problems

As we mentioned earlier, it is natural in this domain to specify synthesis problems using a *finite* set of input-output examples. An input-output example is a pair (i, o) where i is the input—e.g., `"Nadia␣Polikarpova"`—and o is the corresponding output string—e.g., `"NP"`. If a user is interested in synthesizing a program t that is correct on all examples in a set $\mathcal{IO} = \{(i_1, o_1), \dots, (i_j, o_j)\}$, the corresponding STRINGY expression-synthesis problem will have a teacher $\mathsf{T} = (\mathsf{M}, \mathsf{E})$ such that

- For every $(i, o) \in \mathcal{IO}$, we have that $\mathsf{M}(i) = o$. For every i' such that there is no o' such that $(i', o') \in \mathcal{IO}$, $\mathsf{M}(i)$ can be any value.[3]
- Given an expression t^*, we have that $\mathsf{E}(t^*) = \mathsf{true}$ iff $\forall(i, o) \in \mathcal{IO}. [\![t^*]\!](i) = o$. Otherwise, $\mathsf{E}(e^*) = \mathsf{false}$ and it returns as counterexample an i such that there exists a $(i, o) \in \mathcal{IO}$ such that $[\![t^*]\!](i) \neq o$.

3: In practice, our synthesis algorithms will never ask the synthesizer for values outside of the set $\mathcal{IO}$.

Note that it is very easy to implement these oracles programmatically. Once we are given a set of examples, the oracles are simple programs that evaluate programs on such examples.

2.3 Example 2: Hacker's Delight

There are small challenging programming tasks that even the most seasoned programmers may struggle with. An example of such a task is that of writing efficient transformations of bit-vectors. This domain is a good one for synthesis because the programs are typically small and therefore easy to identify for a synthesizer. For example, the synthesizer CVC5 [12] can efficiently synthesize complex bit-vector transformations.

Example 2.3.1 Consider the following examples of input and output bit-vectors for a synthesis problem where the goal is to synthesize a program (using only bit-vector operations) that "isolates" the rightmost 0 bit:

$$\texttt{10010101} \longrightarrow \texttt{00000010}$$

$$\texttt{01101111} \longrightarrow \texttt{00010000}$$

While it is relatively easy to write an imperative program that performs this transformation by traversing the bit-vector, it is fairly challenging to come up with a program that achieves the same result using only bit-level operations.

2.3.1 Bitvy: A Language of Bit-Vector Transformations

Syntax For simplicity, We consider fixed length bit-vectors (8 bits).

The following grammar describes the space of Bitvy programs:

$$\begin{aligned} B ::= {} & \texttt{x} \mid \texttt{0} \mid \texttt{1} \mid \texttt{bvadd}(B, B) \mid \texttt{bvsub}(B, B) \\ & \mid \texttt{bvnot}(B) \mid \texttt{bvor}(B, B) \mid \texttt{bvand}(B, B) \mid \texttt{bvxor}(B, B) \end{aligned} \tag{2.1}$$

Programs produced by the non-terminal B evaluate to bit-vectors. The variable `x` is the input bit-vector,[4] while `0` and `1` are the constant bit-vecotrs `00000000` and `00000001`, respectively. The remaining operations are the natural operations over bit-vectors, addition, subtraction, negation, or, and, and xor.

4: In general, we could have more than one variables, but we consider the single-variable case here for simplicity.

Example 2.3.2 Consider again the synthesis problem from example 2.3.1. The following Bitvy program performs the desired bit-vector transformation:

$$\texttt{bvand}(\texttt{bvnot}(\texttt{x}), \texttt{bvadd}(\texttt{x}, \texttt{1}))$$

Semantics We define the semantics of Bitvy programs inductively in terms of bit-vector operations.

$$\begin{aligned} [\![\texttt{x}]\!]_S(x) = {} & x \\ [\![\texttt{0}]\!]_S(x) = {} & \texttt{00000000} \\ [\![\texttt{1}]\!]_S(x) = {} & \texttt{00000001} \\ [\![\texttt{bvadd}(b_1, b_2)]\!]_S(x) = {} & [\![b_1]\!]_S(x) + [\![b_2]\!]_S(x) \\ [\![\texttt{bvsub}(b_1, b_2)]\!]_S(x) = {} & [\![b_1]\!]_S(x) - [\![b_2]\!]_S(x) \\ [\![\texttt{bvnot}(b_1)]\!]_S(x) = {} & \sim [\![b_1]\!]_S(x) \\ [\![\texttt{bvor}(b_1, b_2)]\!]_S(x) = {} & [\![b_1]\!]_S(x) \texttt{|} [\![b_2]\!]_S(x) \\ [\![\texttt{bvand}(b_1, b_2)]\!]_S(x) = {} & [\![b_1]\!]_S(x) \& [\![b_2]\!]_S(x) \\ [\![\texttt{bvxor}(b_1, b_2)]\!]_S(x) = {} & [\![b_1]\!]_S(x) \texttt{␣} [\![b_2]\!]_S(x) \end{aligned}$$

2.3.2 Specifying bit-vector problems

Verification is possible but synthesis can be hard if we try on all inputs.

The Theory of Bit-Vectors and SMT Solvers To define Bitvy synthesis problem we can take advantage of the fact that bit-vectors, when manipulated using only the operations defined in Bitvy, form a decidable theory—i.e., there exist constraint solvers that given a Boolean formula involving bit-vector variables and the Bitvy operators can tell us whether the formula is satisfiable. For example, given the formula $\texttt{bvadd}(x, x) = \texttt{bvsub}(y, y)$, an SMT solver can quickly find the following satisfying assignment $\texttt{x} = \texttt{10000000}$ and $\texttt{x} = \texttt{00000000}$

We can use this property to write equivalence oracles that check the correctness of a program against a given bit-vector formula $\psi(x)$—e.g., some formula describing the behavior of some function implementation we are trying to optimize in our synthesis problem. If a user is interested in synthesizing a program e that is equivalent to some formula $\psi(x, y)$ (where x is the input variable and y is the corresponding output value), the corresponding Bitvy expression-synthesis problem will have a teacher $\mathsf{T} = (\mathsf{M}, \mathsf{E})$ such that

- For every input i, we have that $\mathsf{M}(i) = \psi(x, y)$.
- Given an expression t^*, we have that the equivalence oracle $\mathsf{E}(t^*) = true$ iff the formula $\psi(x, y) \wedge [\![(]\!]x) \neq y$ is unsatisfiable. Otherwise, $\mathsf{E}(t^*) = \mathsf{false}$ and it returns as counterexample an i such that $\psi(i, o) \wedge [\![(]\!]i) \neq o$ is a satisfying assignment (for some value o).

Bibliography

Here are the references in citation order.

[1] Cordell Green. "Application of Theorem Proving to Problem Solving". In: *Proceedings of the 1st International Joint Conference on Artificial Intelligence*. IJCAI'69. Washington, DC: Morgan Kaufmann Publishers Inc., 1969, pp. 219–239 (cit. on p. 1).

[2] Edsger Dijkstra. "Program inversion". In: *Program Construction* (1979), pp. 54–57 (cit. on p. 1).

[3] A. Pnueli and R. Rosner. "On the Synthesis of a Reactive Module". In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 179–190. DOI: 10.1145/75277.75293 (cit. on p. 1).

[4] Armin Biere et al., eds. *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009 (cit. on p. 1).

[5] Sumit Gulwani. "Automating String Processing in Spreadsheets Using Input-Output Examples". In: *SIGPLAN Not.* 46.1 (2011), pp. 317–330. DOI: 10.1145/1925844.1926423 (cit. on pp. 1, 11).

[6] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. "Genesis: synthesizing forwarding tables in multi-tenant networks". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. Ed. by Giuseppe Castagna and Andrew D. Gordon. ACM, 2017, pp. 572–585. DOI: 10.1145/3009837.3009845 (cit. on p. 1).

[7] Chenglong Wang et al. "Falx: Synthesis-Powered Visualization Authoring". In: *CHI '21: CHI Conference on Human Factors in Computing Systems, Virtual Event / Yokohama, Japan, May 8-13, 2021*. Ed. by Yoshifumi Kitamura et al. ACM, 2021, 106:1–106:15. DOI: 10.1145/3411764.3445249 (cit. on p. 1).

[8] Xiangyu Zhou et al. "Synthesizing Analytical SQL Queries from Computation Demonstration". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 168–182. DOI: 10.1145/3519939.3523712 (cit. on p. 1).

[9] Sumit Gulwani. "Dimensions in Program Synthesis". In: *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. PPDP '10. Hagenberg, Austria: Association for Computing Machinery, 2010, pp. 13–24. DOI: 10.1145/1836089.1836091 (cit. on pp. 2, 3).

[10] Susmit Jha et al. "Oracle-Guided Component-Based Program Synthesis". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. Cape Town, South Africa: Association for Computing Machinery, 2010, pp. 215–224. DOI: 10.1145/1806799.1806833 (cit. on p. 8).

[11] Dana Angluin. “Learning regular sets from queries and counterexamples”. In: *Information and Computation* 75.2 (1987), pp. 87–106. DOI: `https://doi.org/10.1016/0890-5401(87)90052-6` (cit. on p. 8).

[12] Haniel Barbosa et al. “cvc5: A Versatile and Industrial-Strength SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Cham: Springer International Publishing, 2022, pp. 415–442 (cit. on p. 13).