Program Synthesis: The Book

Loris D'Antoni, Nadia Polikarpova

September 29, 2022

An Awesome Publisher

Contents

List of Figures

List of Tables

1 What is Program Synthesis?

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.

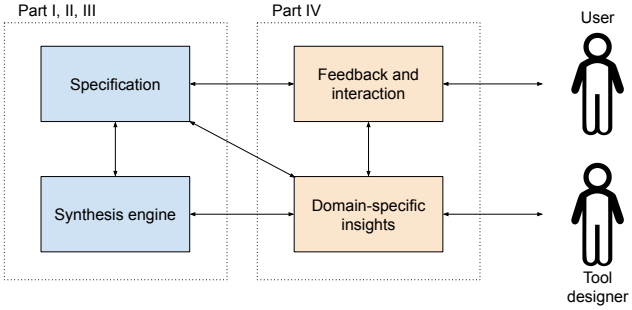


Figure 1.1: Outline of synthesis

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

Synthesizing Expressions: Two Simple Domains

2

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 $\mathsf{rk}_\Sigma : \Sigma \to \mathbb{N}$ 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 $A_0 \to \sigma^{(i)}(A_1, \ldots, A_i)$, 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 $r = A \to \beta$ 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 $\tau \to_G \tau'$ (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 $\tau \to_G^* \tau'$; 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.* $S \to_G^* t$.

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`:

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

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 $[\![\texttt{x} = \texttt{0} + \texttt{1}]\!](\epsilon_1) = \text{true}$, 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 $E^{ID}_{sy} = (\mathsf{T}_{ID}, G_{CI})$ 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 $\mathsf{E}_{ID}(t)$ 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^{ID}_{sy}. Therefore, the problem E^{ID}_{sy} 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 $E^{double}_{sy} = (\mathsf{T}_{double}, G_{CI})$ 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^{double}_{sy} 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 $\mathscr{E}$. If S finds a solution that is correct on all elements of $\mathscr{E}$, 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 $\mathscr{E}$ and repeats the following steps:

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

In the following, we use the notation $[\![t]\!](\mathscr{E})$ and $\mathsf{M}(\mathscr{E})$ to denote the vector of outputs corresponding to the inputs in $\mathscr{E}$ (for this purpose we treat $\mathscr{E}$ 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* $E_{sy}^{\mathscr{E}} = ((\mathsf{M}, \mathsf{E}_{\mathscr{E}}), G)$.

An example-based expression synthesis problem is **realizable** *iff there exists an expression* $t \in L(G)$ *such that* $\mathsf{E}_{\mathscr{E}}(t) \equiv ([\![t]\!](\mathscr{E}) = \mathsf{M}(\mathscr{E}))$*—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 $\mathscr{E}$, we denote with $E_{sy}^{\mathscr{E}} = ((\mathsf{M}, \mathsf{E}_{\mathscr{E}}), G)$ the corresponding example-based expression synthesis problem that only requires the expression t to be correct on the examples in $\mathscr{E}$.

Lemma 2.1.1 (Unrealizability over examples) *If* $E_{sy}^{\mathscr{E}}$ *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 $E_{sy}^{\mathscr{E}}$ for every possible set $\mathscr{E}$—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* $\mathscr{E}$ *the problem* $E_{sy}^{\mathscr{E}}$ *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 total order $\leq$ over expressions has *no infinite descending chains* if for every expression τ, there exists only finitely many terms that are smaller than it—i.e., the cardinality of the set $\{\tau' \mid \tau' \leq \tau\}$ is finite for every term τ.

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 $\leq$ with no infinite descending chains over expressions to solve the generated example-based expression synthesis problems, CEGIS terminates and finds the minimal solution with respect to $\leq$ over expressions.*

Essentially, all we want is a synthesizer that eventually enumerates all the expressions and does not miss any. 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:

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

Programs produced by the nonterminal S evaluate to strings. The variable `x` is the input string, `"␣"` is the space character, $s_1 +\!\!+ s_2$ 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 $\textbf{\texttt{find}}(s_1, s_2)$ 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 $S +\!\!+ S$ instead of $\texttt{concat}(S, S)$ and $S[N \,..\, N]$ instead of $\texttt{substr}(S, N, N)$.

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

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

The following Stringy program performs the desired string transformation:

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

Semantics Now that we have informally described the semantics of a Stringy programs, we can do so more formally. We will use two functions, $[\![\cdot]\!]_S : \texttt{String} \to \texttt{String}$ and $[\![\cdot]\!]_N : \texttt{String} \to \mathbb{N}$ 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:

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

$$
\begin{aligned}
[\![\texttt{find}(s_1, s_2)]\!]_N(str) &= [\![s_1]\!]_S(str).\texttt{indexOf}([\![s_2]\!]_S(str)) \\
[\![\texttt{0}]\!]_N(str) &= 0 \\
[\![\texttt{\$}]\!]_N(str) &= str.\texttt{length} \\
[\![n+\texttt{1}]\!]_N(str) &= [\![n]\!]_N(str) + 1 \\
[\![n-\texttt{1}]\!]_N(str) &= [\![n]\!]_N(str) - 1
\end{aligned}
$$

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 $\mathcal{IO} = \{(i_1, o_1), \ldots, (i_j, o_j)\}$, 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, $\mathsf{E}(t^*) = \mathsf{false}$ 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:

```
10010101 ⟶00000010
01101111 ⟶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.

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

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 `x` = `10000000` and `x` = `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 Brrvy 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 $\psi(x, y) \wedge [\![t^*]\!](x) \neq y$ is unsatisfiable. Otherwise, $\mathsf{E}(t^*) = \mathsf{false}$ and it returns as counterexample an i such that $\psi(i, o) \wedge [\![t^*]\!](i) \neq o$ is a satisfying assignment (for some value o).

3 Enumerative Synthesis

In this chapter we introduce our first family of search algorithms for program synthesis, referred to in the literature as *enumerative* or *exhaustive* search. The general idea in these algorithms is to exhaustively enumerate all programs in the search space from smaller to larger, until we find one that satisfies the specification. Although this general idea is very simple, it turns out to be remarkably effective, when combined with clever strategies for *pruning* the search space (*i.e.* discarding irrelevant parts of the search space without enumerating their programs explicitly).

In the following, we will first discuss the structure of a search space defined by a regular tree grammar; then we will move on to concrete algorithms that enumerate this space in a specific order; finally, we will discuss pruning techniques that make these algorithms practical. Throughout the chapter, we target example-based expression synthesis problems $E^{\mathscr{E}}_{sy} = (\mathsf{M}, G, \mathscr{E})$, defined by the membership oracle M, a grammar G, and a set of inputs $\mathscr{E}$. In other words, we assume that the behavioral specification is given by input-output examples, and the structural specification is given by a regular tree grammar. As our target grammar, we will use (a subsets of) the Stringy DSL from Chapter 2.

3.1 The Search Space Defined by a Grammar

Recall from Chapter 2, that we are going to use regular tree grammars (or RTGs) to define the space of the programs that the synthesizer has to search (in other words, to define the *structural specification* of a synthesis problem). But what exactly is the search space that corresponds to an RTG? For a given grammar $G = (N, \Sigma, S, a, \delta)$, we have defined the language of this grammar, $L(G)$, to be the set of all terms that can be derived from S using the productions in δ. This definition, however, is not very practical: first, it does not immediately suggest an algorithm for how to systematically enumerate terms from $L(G)$, and second, for most grammars G their language is infinite, so we certainly cannot enumerate all trees from $L(G)$.

To get us closer to an algorithm, we define the notion of *(height-)bounded language of* G, denoted $L_{\leq n}(G)$, which, informally, is a subset of $L(G)$ where the height of all trees is limited by n. In order to define this set, we need an auxiliary notion of a *bounded language of a specific non-terminal* $A \in N$, denoted $L_{\leq n}(A)$, which is the set of all trees of height up to n derivable from A.

Definition 3.1.1 (Height-Bounded Language) *The n-bounded language of a non-terminal A, denoted $L_{\leq n}(A)$, is defined inductively using two inference*

Inductive Definitions.

rules:

$$\text{L-Base}\ \frac{A \to \sigma^{(0)} \in \delta}{\sigma^{(0)} \in L_{\leq 0}(A)} \qquad \text{L-Step}\ \frac{A \to \sigma^{(k)}(A_1, \cdots, A_k) \in \delta \quad \forall i \in 1 \cdots k \,.\, t_i \in L_{\leq n-1}(A_i)}{\sigma^{(k)}(t_1, \cdots, t_k) \in L_{\leq n}(A)}$$

The n-bounded language of a grammar G is the bounded language of its starting non-terminal S: $L_{\leq n}(G) = L_{\leq n}(S)$.

The base case rule, L-Base, says that for height zero, the only trees in $L_{\leq 0}(A)$ are zero-arity terminals derivable from A. The inductive case rule, L-Step, says that a tree of size up to n can be constructed from sub-trees $t_1, \cdots, t_k$, which are drawn from languages of the appropriate non-terminals of height up to $n - 1$.

In addition to the bounded language, $L_{\leq n}(A)$, we also define the *fixed-height language* $L_n(A)$, the set of terms derivable from A of height *exactly* n. This set can be defined concisely in terms of $L_{\leq n}(A)$:

$$\begin{aligned} L_0(A) &= L_{\leq 0}(A) \\ L_{n+1}(A) &= L_{\leq n+1}(A) \setminus L_{\leq n}(A) \end{aligned}$$

Example 3.1.1 Consider the following simplified grammar G_s for the String language:

$$\begin{aligned} S &::= \texttt{x} \mid \texttt{"␣"} \mid S \mathbin{+\!\!+} S \mid S[N \,..\, N] \\ N &::= \texttt{0} \mid \textbf{find}(S, S) \end{aligned}$$

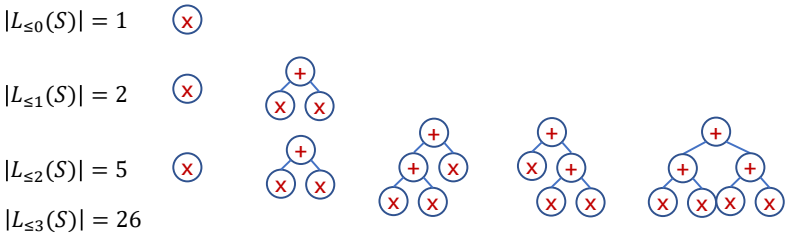
Here are the first few fixed-height languages for the two non-terminals of this grammar:

$$\begin{aligned} L_0(S) &= \{\texttt{x}, \texttt{"␣"}\} \\ L_0(N) &= \{\texttt{0}\} \\ L_1(S) &= \{\texttt{x} \mathbin{+\!\!+} \texttt{x}, \texttt{x} \mathbin{+\!\!+} \texttt{"␣"}, \texttt{"␣"} \mathbin{+\!\!+} \texttt{x}, \texttt{"␣"} \mathbin{+\!\!+} \texttt{"␣"}, \texttt{x[0 .. 0]}, \texttt{"␣"[0 .. 0]}\} \\ L_1(N) &= \{\textbf{find}(\texttt{x}, \texttt{x}), \textbf{find}(\texttt{x}, \texttt{"␣"}), \textbf{find}(\texttt{"␣"}, \texttt{x}), \textbf{find}(\texttt{"␣"}, \texttt{"␣"})\} \\ L_2(S) &= \{\texttt{x} \mathbin{+\!\!+} (\texttt{x} \mathbin{+\!\!+} \texttt{x}), \texttt{x} \mathbin{+\!\!+} (\texttt{x} \mathbin{+\!\!+} \texttt{"␣"}), \ldots, (\texttt{x} \mathbin{+\!\!+} \texttt{x}) \mathbin{+\!\!+} \texttt{x}, \ldots, \\ &\qquad \texttt{x[0 .. } \textbf{find}(\texttt{x}, \texttt{"␣"})\texttt{]}, \ldots\} \end{aligned}$$

It is easy to show by induction on n that:

1. All terms in a bounded language indeed have bounded height: $\forall t \in L_{\leq n}(A) : \mathsf{height}(t) \leq n$
2. Fixed-height language contains exactly those terms from a bounded language of height n: $\{t \in L_{\leq n}(A) \mid \mathsf{height}(t) = n\} = L_n(A)$

Although Definition 3.1.1 is not quite an algorithm, it is not hard to translate it into a recursive function that computes $L_{\leq n}(A)$. This function would iterate over the productions in δ to find all productions of the form $A \to \sigma^{(k)}(A_1, \cdots, A_k)$ (*i.e.* whose left-hand side is A), then recursively



$|L_{\leq 3}(S)| = 26$
$|L_{\leq 4}(S)| = 677$
$|L_{\leq 5}(S)| = 458330$
$|L_{\leq 6}(S)| = 210066388901$
$|L_{\leq 7}(S)| = 44127887745906175987802$
$|L_{\leq 8}(S)| = 1947270476915296449559703445493848930452791205$
$|L_{\leq 9}(S)| = 379186231026592608286823502802789327737023315224738 85$
$\ldots 847617341507177682544103411753253520 26$

Figure 3.1: (Sizes of) the first ten bounded languages for the grammar in Equation 3.1.

compute all sets $L_{\leq n-1}(A_i)$ and use all combinations of subterms from those sets to construct larger terms. Later in this chapter, we will make our enumeration algorithm more precise.

How big is the search space? Before we dive into the details of algorithms though, let's first get a feel for how big this searh space really it, that is, how fast $L_{\leq n}(G)$ grows with n. Deriving an analytical solution for our Stringy grammar is a little tricky, so let's do it for an even simpler language: one that has a single nullary and a single binary terminal:

$$S ::= \mathsf{x} \mid S + S \tag{3.1}$$

Bounded languages for this grammar for heights up to two are depicted in Figure 3.1.

In general, for an arbitrary n, $L_{\leq n}(S)$ always has a single term with root x (that is, the term x), and the rest of the terms have root $+$. Of the terms with root $+$, each has two subterms, both drawn from the set $L_{\leq n-1}(S)$. Hence, the size of the bounded language satisfies the following recurrence relation:

$$|L_{\leq n}(S)| = 1 + |L_{\leq n-1}(S)|^2$$

In closed form, this function is doubly exponential: $|L_{\leq n}(S)| \sim c^{2^n}$, where $c > 1$. The first ten values of this function are given in Figure 3.1. You can see that the size of the search space grows extremely quickly with the height, even for this absolutely trivial grammar: exhaustively enumerating all terms of height up to four is no problem at all, but doing so for height up to six is completely hopeless. The search space, of course, grows even faster for larger grammars. For example, for the Stringy grammar from Example 3.1.1:

$$|L_{\leq 2}(S)| = 266$$
$$|L_{\leq 3}(S)| = 1,194,608$$
$$|L_{\leq 4}(S)| = 5,982,295,413,033,458$$

so in this case, even for heights up to four, exhaustive enumeration is off the table.

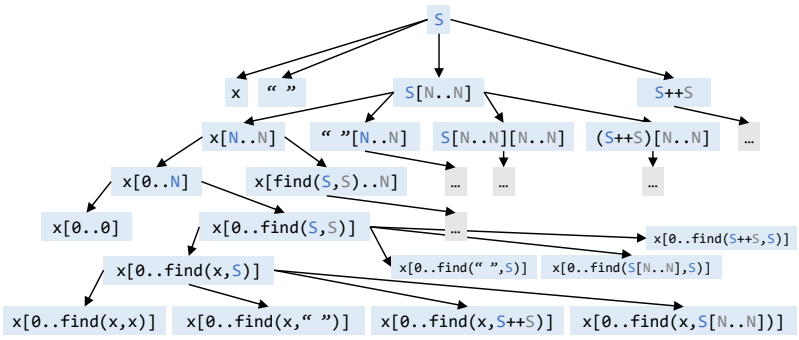


Figure 3.2: A search-tree representation of the StringY language from Example 3.1.1.

3.2 Top-Down Enumeration

All enumerative synthesis algorithms build upon one of the two core enumeration mechanisms: *top-down* or *bottom-up* enumeration. These names are derived from the order in which the algorithm constructs the nodes of the program's AST: top-down enumeration constructs a program starting from its root and bottom-up enumeration starts from the leaves.

We will begin in this section with top-down enumeration, because it is the closest to the informal recursive enumeration procedure we outlined in the previous section. At a high-level, top-down enumeration explores an (infinite) *search tree*. The nodes of this tree are incomplete programs $\tau \in \mathcal{T}_{\Sigma \cup N}$; the root node is the starting non-terminal, and the children of each node τ are all programs τ' that can be derived from τ in one-step, *i.e.* such that $\tau \longrightarrow \tau'$.[1] For example, Figure 3.2 shows a part of the search tree for the StringY grammar G_s from Example 3.1.1. You can see, for example, that the node $S[N \,..\, N]$ has four children: one for each production for its leftmost nonterminal S: $S \longrightarrow \texttt{x}$, $S \longrightarrow \texttt{"␣"}$, $S \longrightarrow S +\!\!+ S$, and $S \longrightarrow S[N \,..\, N]$. Each path from the root in the search tree leads to a unique incomplete program. It is sometimes useful to think of the search tree as consisting of *levels*, where each level i contains all programs τ whose derivation contains exactly i productions.

1: Recall from Chapter 2 that this means applying a production to the *left-most* nonterminal in τ.

The top-down enumeration algorithm works by traversing the search tree in some order: typically either depth-first (for a fixed depth bound) or breadth-first.[2] The pseudocode is given in Algorithm 1. This algorithm maintains a worklist *wl* of term incomplete programs (*i.e.* nodes of the search tree), which is initialized with a single incomplete term S (the starting non-terminal). In each iteration, the algorithm dequeues a term τ from the front of the worklist; if τ is a complete term and behaves correctly on all examples, it is returned as the solution. Otherwise, the auxiliary procedure New-Terms computes the set of all programs τ' that can be reached from τ by unrolling its left-most non-terminal; all these programs are added to the worklist.

2: In section Section 3.5 we will also discuss the possibility of traversing this tree in *best-first* order, according to some cost function on incomplete programs.

The order in which the search tree is explored can be configured by varying the behavior of the enqueue function: if new nodes are added to the front of the worklist, this results in a depth-first traversal; if they are added to the back, this results in a breadth-first traversal. It is often useful to impose a bound on the depth or size of the programs that the algorithm has to explore (especially when using a depth-first traversal,

Algorithm 1 Top-down search algorithm

Input: Example-based expression syntheis problem $E^{\mathscr{E}}_{sy} = (\mathsf{M}, G, \mathscr{E})$
Output: Expression t^* such that $[\![t^*]\!](\mathscr{E}) = \mathsf{M}(\mathscr{E})$

```
1:  procedure Top-Down-Search(M, G = (N, Σ, S, a, δ), E)
2:      wl ← [S]                                   ▷ Initialize the worklist
3:      while wl ≠ [] do
4:          τ ← wl.dequeue                         ▷ Dequeue the first element
5:          if τ ∈ T_Σ then                        ▷ τ is complete
6:              if [[τ]](E) = M(E) then            ▷ Behaves correctly on E
7:                  return τ                       ▷ Solution found
8:          else
9:              for τ' ∈ New-Terms(τ, G) do
10:                 wl.enqueue(τ')                 ▷ Add to worklist
11: procedure New-Terms(e, G)
12:     A ← left-most non-terminal in τ
13:     for A → β ∈ δ do                           ▷ For all productions from A
14:         τ' ← τ[A → β]
15:         if τ' is within bounds then
16:             yield τ'
```

which would otherwise go down a "rabbit hole"). This is accomplished in line 15.

Example 3.2.1 Consider the synthesis problem where the task is to extract first names from the following two strings in Stringy:

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

More formally, we are given an example-based expression synthesis problem $(\mathsf{M}, G'_s, \mathscr{E})$, where

$$\mathscr{E} = \{[x \mapsto \texttt{"Nadia␣Polikarpova"}], [x \mapsto \texttt{"Loris␣D'Antoni"}]\}$$

and the membership oracle M is such that

$$\mathsf{M}(\texttt{"Nadia␣Polikarpova"}) = \texttt{"Nadia"}$$
$$\mathsf{M}(\texttt{"Loris␣D'Antoni"}) = \texttt{"Loris"}$$

For illustration purposes, we take the grammar G'_s to be:

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

This is the same grammar as G_s from Example 3.1.1, but the order of the concatenation and substring productions is swapped (the role of this change is discussed below). Let us examine how this problem can be solved using Algorithm 1.

We start with the initial worklist $[S]$. In the first iteration, we dequeue S, which is incomplete. NewTerms(S, G) returns four programs, once for each S-production in G, which are added back to the worklist:

$$wl = [\mathtt{x}, \texttt{"␣"}, S[N \mathinner{..} N], S \mathbin{+\!\!+} S]$$

In iterations two and three we dequeue the first two of these programs, which are complete; these programs are tested using the membership oracle, but neither of them is the solution, so we continue with the worklist

$$wl = [S[N \mathinner{..} N], S \mathbin{+\!\!+} S]$$

In iteration four, we dequeue $S[N \mathinner{..} N]$ and expand its leftmost nonterminal, S, in four different ways once again; we add the resulting four programs to the front of the worklist (assuming we are interested in depth-first traversal):

$$wl = [\texttt{x}[N \mathinner{..} N], \texttt{"␣"}[N \mathinner{..} N], \\ (S[N \mathinner{..} N])[N \mathinner{..} N], (S \mathbin{+\!\!+} S)[N \mathinner{..} N], S \mathbin{+\!\!+} S]$$

In iteration five, we expand the leftmost N non-terminal of $\texttt{x}[N \mathinner{..} N]$ according to the two N productions, resulting in:

$$wl = [\texttt{x}[\texttt{0} \mathinner{..} N], \texttt{x}[\texttt{find}(S, S) \mathinner{..} N], \ldots \text{(4 old terms)}]$$

A similar thing happens in the next iteration, expanding the sole N non-terminal of $\texttt{x}[\texttt{0} \mathinner{..} N]$:

$$wl = [\texttt{x}[\texttt{0} \mathinner{..} \texttt{0}], \texttt{x}[\texttt{0} \mathinner{..} \texttt{find}(S, S)], \ldots \text{(5 old terms)}]$$

The front of this worklist is a term, but once again does not match the membership oracle (this term is equivalent to the empty string).

In four more iterations, the term $\texttt{x}[\texttt{0} \mathinner{..} \texttt{find}(\texttt{x}, \texttt{"␣"})]$ will finally appear at the front of the worklist; this term matches the membership oracle, and hence will be returned as the solution.

Depth-First vs Breadth-First As you can see, the content of the worklist in the example above corresponds to the blue terms in Figure 3.2; in other words, in this example, we were exploring the search tree from Figure 3.2 in depth-first order. In this case we were lucky and found the solution relatively quickly, after constructing only 21 terms. Consider what would happen, however, if we switched back to the original grammar G_s, where the concatenation production comes before substring. In this case, the term $S \mathbin{+\!\!+} S$ would appear in the worklist before $S[N \mathinner{..} N]$, and would be expanded first; the search would then go down a rabbit hole, and never even get to explore terms with substrings unless we impose a bound on the term size or depth. Even with such a bound in place, the run time of depth-first search is extremely sensitive to the order of productions: for example, if we impose the depth bound of 3, depth-first search would explore $\sim 70K$ terms of the form $S \mathbin{+\!\!+} S$ before it exhausts the bound and switches to exploring terms of the form $S[N \mathinner{..} N]$.

Breadth-first search is much less brittle in that sense: in explores the search tree *level-by-level*, and hence the order in which terms end up at the front of the worklist varies much less with the order of productions in the grammar. But it has a complementary downside: it *always* enumerates many irrelevant terms; for example, exploring the search tree in Figure 3.2 in a breadth-first manner would expand all the nodes hidden behind the gray ellipses all the way down to level six, before we get to test our

solution, `x[0 .. find(x, "␣")]`. In Section 3.5 we will discuss how we can do better than naive depth- or breadth-first search in case we have some information about what kind of programs are more likely to solve the synthesis task.

Correctness How do we argue that Algorithm 1 is "correct"? The notion of correctness for a synthesis algorithm typically encompasses three desirable properties:

1. *Soundness*: if the algorithm returns a program, it is guaranteed to be a solution (to satisfy the specification).
2. *Completeness*: if there exists a solution, the algorithm is guaranteed to eventually return one.
3. *Termination*: the algorithm cannot run forever.

It is rare for a synthesis algorithm to have all three of these properties, because in most cases the space of programs is infinite, so it is hard to come up with an algorithm that is both complete and always terminates.

Which of the three properties does Algorithm 1 possess? Soundness is trivial to show in this case, because line 5 explicitly checks that e's behavior matches the oracle. This is typically the case for all synthesis algorithms that solve the example-based expression synthesis problem, because checking program correctness in this case amounts to simply evaluating a candidate expression on the example inputs.

To show completeness, we need to argue that any program in the language $L(G)$ will eventually find its way to the front of the worklist. As the first step, we can show that the infinite search tree the algorithm explores indeed contains all $t \in L(G)$; this is true because for any such t there exists a derivation from the initial non-terminal S of the grammar G, and our search tree clearly contains all derivations from S.

Now we need to check whether the algorithm eventually visits every node of the search tree. Let us first consider *unbounded* search (*i.e.* the check in line 15 always passes). In this case, breadth-first search is complete, because it explores the tree level-by-level, and each level contains finitely many nodes (since there are finitely many productions in G, and each level i must be generated by a sequence of i productions). Unbounded depth-first search, on the other hand, is not complete, because it can get stuck in one infinitely-deep branch of the search tree and never get to explore another. Neither of the two search strategies terminate when the synthesis problem is unrealizable, unless $L(G)$ is finite. Now if we *bound* the search by either depth or size, both strategies become terminating and incomplete (since the solution might lie beyond the bound).

Avoiding Repeated Work You might have noticed that our naïve top-down enumeration algorithm does a lot of repeated work, unrolling the same non-terminals many times in different contexts. For example, consider once again solving the synthesis task of Example 3.2.1 via depth-first search with the original grammar G_s and the depth bound of 3. We already mentioned that the search has to explore all terms of the form $S + S$ of depths up to 3 before it can switch to the fruitful branch $S[N \,..\, N]$; what is worse, however, is that as part of this exploration it essentially computes $L_{\leq 2}(S)$ *twice*, once for the S on the left and once for

the S on the right. In other words, the program synthesis problem has *overlapping subproblems* [13].

A common way to speed up search in the presence of overlapping subproblems is to use some form of *dynamic programming* to cache or memoize the solutions to subproblems, so that they can be later reused. This is exactly the approach taken by the bottom-up enumeration algorithm, which we will discuss next.

3.3 Bottom-Up Enumeration

At a high level, the idea of bottom-up search is to construct every fixed-height language $L_n(A)$ one by one in the order of increasing height n, storing all the enumerating terms in order to reuse them when building larger terms. This algorithm is depicted in Algorithm 2. In order to avoid re-computing the same subterms multiple times, the algorithm maintains a map $\mathsf{B}_i(A)$, which we call an *expression bank*. The bank stores all enumerated expressions, indexed by their height and the non-terminal from which they were derived. For example, the term `x ++ x` of height one from Example 3.1.1 would be stored in $\mathsf{B}_1(S)$, while **`find`**`(x ++ x, "␣")` would be stored in $\mathsf{B}_2(N)$.

Each iteration of the top-level loop (line 3) constructs all expressions of height exactly n and adds them to the bank (or returns if the solution is found in line 5). To that end, it invokes the auxiliary procedure New-Terms, which computes the set of all terms from grammar G of height exactly n, assuming that all terms of smaller heights are already stored in the bank B. Note that New-Terms is implemented as an iterator, *i.e.* it lazily yields terms one at a time. Together with each term t, the procedure also yield the non-terminal A it is derived from; this information is needed in line 5 to check whether t is a whole program (*i.e.* is derived from the starting non-terminal), and also in line 7 to add t to the appropriate component of the bank.

Inside, New-Terms iterates over all productions in the grammar and all combinations $(t_1, \ldots, t_k)$ of expressions in the bank that can be used as sub-terms to construct a new term of height n (for this to hold, one of the expressions t_i has to be of height $n-1$, and the rest can be of any height less than n). Note that when either the height n or the arity k are zero, the loop in line 13 produces an empty set of terms (because there are no subterms to combine); this is almost always what we want, except when *both* height and arity are zero: this is the base case of our algorithm, and it should return the set of all zero-arity terminals, which do not require any subterms. This case is handled in line 10.

Example 3.3.1 Let us come back to the problem of extracting first names, introduced in Example 3.2.1, and examine how it can be solved using Algorithm 2.

We start with an empty bank and height $n = 0$. In the first iteration, New-Terms only yields zero-arity terminals `x`, `"␣"`, and `0` (via line 10). None of these expressions satisfy the specification, so they are all added to the bank. After this iteration there are two non-empty entries

Algorithm 2 Bottom-up search algorithm

Input: Example-based expression syntheis problem $E^{\mathscr{E}}_{sy} = (\mathsf{M}, G, \mathscr{E})$
Output: Expression t^* such that $[\![t^*]\!](\mathscr{E}) = \mathsf{M}(\mathscr{E})$

```
1:  procedure Bottom-Up-Search(M, G = (N, Σ, S, a, δ), E)
2:      B_i(A) ← ∅ for all i ∈ ℕ, A ∈ N                ▷ Initialize the bank
3:      for n ∈ [0 .. ∞] do                            ▷ For all term heights
4:          for (A, t) ∈ New-Terms(n, G, B) do
5:              if A = S ∧ [[t]](E) = M(E) then        ▷ Whole and correct
6:                  return e                           ▷ Solution found
7:              B_n(A) += e                            ▷ Add to bank
8:  procedure New-Terms(n, G, B)
9:      for A → σ(A_1, ..., A_k) ∈ δ do                ▷ For all grammar productions
10:         if n = 0 ∧ k = 0 then                      ▷ Base case: arity and height 0
11:             yield (A, σ)
12:         else                                       ▷ Build using sub-terms from the bank
13:             for (n_1 ..., n_k) ∈ [0 .. n − 1]^k s.t. ∃i.n_i = n − 1 do
14:                 for (t_1, ..., t_k) ∈ B_{n_1}(A_1) × ... × B_{n_k}(A_k) do
15:                     yield (A, σ(t_1, ..., t_k))    ▷ Construct new term
```

in the bank:

$$\mathsf{B}_0(S) = \{\texttt{x}, \texttt{"␣"}\} \qquad \mathsf{B}_0(N) = \{\texttt{0}\}$$

(note that this matches the fixed-height languages $L_0(S)$ and $L_0(N)$, which we computed in Example 3.1.1).

Moving on to the second iteration ($n = 1$), assume that New-Terms chooses the production $S \longrightarrow S \mathbin{+\!\!+} S$ with arity 2. In this case, there is a single option for subterm heights in line 13: $(n_1, n_2) \in \{(0, 0)\}$; hence line 14 iterates over all pairs of terms from $\mathsf{B}_0(S)$, yielding new terms $\texttt{x} \mathbin{+\!\!+} \texttt{x}$, $\texttt{x} \mathbin{+\!\!+} \texttt{"␣"}$, $\texttt{"␣"} \mathbin{+\!\!+} \texttt{x}$, and $\texttt{"␣"} \mathbin{+\!\!+} \texttt{"␣"}$. By the end of the second iteration, B_0 remains the same, and the state of B_1 is:

$$\mathsf{B}_1(S) = \{\texttt{x} \mathbin{+\!\!+} \texttt{x}, \texttt{x} \mathbin{+\!\!+} \texttt{"␣"}, \texttt{"␣"} \mathbin{+\!\!+} \texttt{x}, \texttt{"␣"} \mathbin{+\!\!+} \texttt{"␣"}, \texttt{x[0 .. 0]}, \texttt{"␣"[0 .. 0]}\}$$
$$\mathsf{B}_1(N) = \{\texttt{find(x, x)}, \texttt{find(x, "␣")}, \texttt{find("␣", x)}, \texttt{find("␣", "␣")}\}$$

In the third iteration ($n = 2$) there are more choices for subterm heights in line 13. For example, with the same production $S \longrightarrow S \mathbin{+\!\!+} S$, those heights now range over $(n_1, n_2) \in \{(0, 1), (1, 0), (1, 1)\}$, generating terms such as $\texttt{x} \mathbin{+\!\!+} (\texttt{x} \mathbin{+\!\!+} \texttt{x})$, $(\texttt{x} \mathbin{+\!\!+} \texttt{x}) \mathbin{+\!\!+} \texttt{x}$, and $(\texttt{x} \mathbin{+\!\!+} \texttt{x}) \mathbin{+\!\!+} (\texttt{x} \mathbin{+\!\!+} \texttt{x})$, among others. $(n_1, n_2) = (0, 0)$ is explicitly not considered here, because this would generate a term of height one instead of two.

Luckily, in the course of this iteration we stumble upon the term $\texttt{x[0 .. find(x, "␣")]}$, generated from the production $S \longrightarrow S[N \mathinner{..} N]$ with subterm heights $(0, 0, 1)$. Evaluating this term on the example inputs yields the expected outputs `"Nadia"` and `"Loris"`, so this term is returned as the solution. Note that this likely happens before we have enumerated all 318 terms of height two (but how much before depends on the order of productions in line 9). This might not seem like a big deal, but early termination would make a huge difference if our solution were of height three, since there are *over a million* terms of this height, and we should certainly try to avoid computing them all.

Correctness Recall that in the previous section we discussed soundness, completeness, and termination of top-down search. Which of these

properties hold of our bottom-up search algorithm? Like in the previous case, soundness holds trivially, since line 5 performs an explicit check against the membership oracle; on the other hand, termination trivially does not hold, and can only be enforced by changing the algorithm to impose a bound on program height.

The proof of completeness for Algorithm 2, although not particularly tricky, requires the use of mathematical *induction* (in this case, on the number of iterations of the algorithm or, equivalently, on the height of the constructed terms). More precisely, we want to show that after an arbitrary iteration n, the bank contains all terms in L_G of height up to n. To be able to prove this by induction, we must strengthen this property to the following:

$$\forall i \in [0 \,..\, n], \forall A \in N . \mathsf{B}_i(A) = L_i(A) \tag{3.2}$$

In the base case ($n = 0$), this holds because $L_0(A) = L_{\leq 0}(A)$ which, by the rule L-Base of Definition 3.1.1 contains all $\sigma^{(0)}$ such that $A \longrightarrow \sigma^{(0)} \in \delta$, which are exactly the terms returned in line 11 of Algorithm 2.

For the recursive case, let us assume that (3.2) holds after some iteration $n - 1$, and prove that it still hold after the next iteration n. The reason why this is not entirely trivial is subtle: recall that $L_n(A)$ is defined declaratively as the subset of $L_{\leq n}(A)$ of height n; $L_{\leq n}(A)$ is in turn defined by taking all combinations of subterms from $L_{\leq n-1}(A)$. On the other hand, for efficiency reasons, $\mathsf{B}_n(A)$ is computed in lines 13–15 by combining *only* the sub-terms of appropriate heights from smaller banks (not by combining *all* subterms from smaller banks and then filtering out those that aren't tall enough). So essentially we need to show that lines 13–15 do not miss any terms of height n. This holds because the height of a compound term t is one greater than the maximum height of its subterms; hence for t to have hight n it is *necessary* that one of its subterms has height $n - 1$—the property enforced by line 13.

Caching Evaluation Results Enumerative synthesis relies on churning through a huge number of terms quickly, so it makes sense to optimize the construction and testing of new terms as much as possible. One simple yet important optimization opportunity hides in line 5 of our algorithm, were we evaluate the term t on the inputs $\mathscr{E}$. Since evaluating t involves evaluating all its subexpressions, we end up evaluating the same expressions on the same inputs over and over again. For instance, in Example 3.3.1, we end up evaluating $\mathsf{x} +\!\!+ \mathsf{x}$ once at height one (as a top-level term), and then many more times at height two (while evaluating terms like $\mathsf{x} +\!\!+ (\mathsf{x} +\!\!+ \mathsf{x})$ and $(\mathsf{x} +\!\!+ \mathsf{x}) +\!\!+ \mathsf{x}$). To avoid this repeated computation, an efficient implementation of bottom-up search stores the outputs of each expression t on $\mathscr{E}$ in the bank alongside the expression, and uses these cached values when evaluating larger expressions.

Size-Based Enumeration Algorithm 2 enumerates expressions in the order of increasing *height*. Although this is very natural and easy to implement, it turns out that in practice it is often much more efficient to enumerate expressions in the order of increasing *size*, *i.e.* the total number of nodes in their abstract syntax tree. This might seem a little counter-intuitive at first, but let's look at some examples.

In our first name extraction task (Example 3.2.1), the smallest solution, `x[0 .. find(x, "␣")]`, has height 2 and size 6. The language $L(G)$ contains 266 programs of height up to 2 and only 52 programs of size up to 6, so with the worst-case choice of the order of grammar productions we would have to explore about five time more programs by-height than by-size before we find this solution.

Of course for such small search spaces, the difference in synthesis time would be unnoticeable, but consider a slightly more complex problem:

Example 3.3.2 Consider a variation of the synthesis problem from Example 3.2.1 where we want to repeat the first name twice, like so:

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

In this case, the smallest solution from the grammar G_s is the program

```
x[0 .. find(x, "␣")] ++ "␣" ++ x[0 .. find(x, "␣")]
```
(3.3)

which has size 15 and height 4. There are almost 6×10^{15} programs of height up to 4, but only about $\approx 660K$ programs of size up to 15, which is a giant difference.

Is it a coincidence that the two programs we happen to look at appear in search space much earlier in the order of size than in the order of height? Not really. Generally, the disparity between the height- and size-based search arises because the grammar productions have different arity; if we build a program using predominantly high-arity productions—like substrings in Stringy—the resulting program will be quite a bit larger than most programs of the same height. Empirically, such programs are not very likely to be the solution we are looking for: simply because they are overusing a single production. In other words: useful programs are likely to be *tall and skinny* rather than *short and bushy*. As a result, height-based enumeration explores a huge number of unlikely, bushy programs before it can proceed to the skinny programs of a larger height.

Now that we are convinced that size-based enumeration is a good idea, how do we change Algorithm 2 to enumerate in the order of size? Surprisingly, this is a simple, one-line change, which we leave as an exercise to the reader.

Exercise 3.3.1 Modify Algorithm 2 so that it enumerates expressions in the order of size instead of height. Hint: you can simply reinterpret the loop variable n as size instead of height, and similarly reinterpret the bank as being indexed by size.

3.4 Search Space Pruning

The enumeration algorithms we considered so far have a couple of tricks up their sleeve, but their scalability is fundamentally limited, because they are guaranteed to enumerate *all* terms from $L(G)$ in the order of

height (or size), and we have observed that the size of this space grows extremely quickly. To tackle this scalability issue, existing enumerative synthesizers rely on two classes of techniques:

1. *pruning, i.e.* discarding parts of the search space that cannot possibly contain the solution, and
2. *prioritization, i.e.* exploring candidate programs in a different order, where the solution is more likely to occur earlier.

We discuss pruning in the rest of this section, and defer prioritization to Section 3.5.

How can we determine that a part of the search space is not worth exploring? There is no universal answer to this question, and in fact clever domain-specific pruning techniques often serve as the core insight behind research papers in program synthesis. That said, at a high level, pruning techniques use two main reasons for discarding (partial) programs:

1. *redundant* programs are discarded because they are equivalent to some other program considered by the search, from the point of view of satisfying the specification.
2. *infeasible* programs are discarded because all programs that can be derived from them are definitely incorrect.

In the rest of this section, we discuss the two classes of pruning techniques in turn.

3.4.1 Pruning Redundant Programs

Historically, the techniques for pruning redundant programs—commonly referred to as *equivalence reduction*—have been applied mainly to bottom-up enumeration. Hence, in this section we discuss equivalence reduction in the context of Algorithm 2 (but also mention variations that apply to top-down enumeration, whenever appropriate).

Recall Example 3.3.1, which illustrated bottom-up enumeration using the problem of extracting the first name from a full name. In that example, the first two levels of the expression bank were:

$$\begin{aligned}
\mathsf{B}_0(S) &= \{\texttt{x}, \texttt{"␣"}\} \\
\mathsf{B}_0(N) &= \{\texttt{0}\} \\
\mathsf{B}_1(S) &= \{\texttt{x} +\!\!+ \texttt{x}, \texttt{x} +\!\!+ \texttt{"␣"}, \texttt{"␣"} +\!\!+ \texttt{x}, \texttt{"␣"} +\!\!+ \texttt{"␣"}, \texttt{x}[\texttt{0} \mathinner{..} \texttt{0}], \texttt{"␣"}[\texttt{0} \mathinner{..} \texttt{0}]\} \\
\mathsf{B}_1(N) &= \{\textbf{\texttt{find}}(\texttt{x}, \texttt{x}), \textbf{\texttt{find}}(\texttt{x}, \texttt{"␣"}), \textbf{\texttt{find}}(\texttt{"␣"}, \texttt{x}), \textbf{\texttt{find}}(\texttt{"␣"}, \texttt{"␣"})\}
\end{aligned}$$

You might have noticed that some expressions in this bank are semantically equivalent to each other: for example, both `x[0 .. 0]` and `"␣"[0 .. 0]` always evaluate to the empty string, so there is no need to keep both of them in the bank; similarly, all three expressions `0`, **`find`**`(x, x)`, and **`find`**`("␣", "␣")` always evaluate to zero, hence it would make sense to only keep one of them, called the *representative* of the equivalence class, and discard the other two as *redundant*. Usually the smallest of the equivalent expressions is chosen as the representative.

Why is it beneficial to discard redundant expressions from the bank? Recall that bottom-up search constructs larger expressions by taking all combinations of smaller expressions; thanks to this combinatorial

explosion, discarding even a small number of terms from the lower levels of the bank leads to a significant reduction in the number of terms constructed at higher levels, and hence a significant speedup in the search. For example, discarding the three redundant expressions discussed above *more than halves* the number of terms constructed for $\mathsf{B}_2(S)$—from 258 to 106—and this reduction, in turn, will have an even more significant effect on the bank at depth 3.

On the other hand, if we only ever discard expressions that behave equivalently to the representative, this does not compromise the *completeness* of the synthesis algorithm: in our example, *any* expression synthesis problem that has a solution that uses `find(x, x)` as a subterm, also has a solution that instead uses `0` (and the latter solution is *smaller*, which is usually what we want).

The remaining question is: how do we determine that a program is redundant? In general, program equivalence is undecidable. In the special case where our programs are expressions understood by an SMT solver,[3] we can ask the solver whether two expressions are equivalent. This is, however, quite an expensive operation, and doing something expensive for each enumerated expression defeats the purpose of search space pruning.

3: Review Chapter 2 for an informal introduction to SMT solvers and Chapter 5 for more details.

Observational Equivalence In the context of an example-based synthesis problem $E^{\mathscr{E}}_{sy} = (\mathsf{M}, G, \mathscr{E})$ there is a simple way to define redundant programs: we can consider two programs equivalent if they produce the same output on the inputs $\mathscr{E}$ *from the specification* (as opposed to *on all inputs*, as full semantic equivalence would require). This notions of equivalence is called *observational equivalence* (OE).The insight here is that our behavioral specification cannot distinguish between two programs t_1 and t_2 if they produce the same output on $\mathscr{E}$ (even if they are not actually semantically equivalent), and hence for our purposes of solving $E^{\mathscr{E}}_{sy}$, t_1 and t_2 are completely interchangeable.

Recall again our running example, where

$$\mathscr{E} = \{[x \mapsto \texttt{"Nadia␣Polikarpova"}], [x \mapsto \texttt{"Loris␣D'Antoni"}]\}$$

The three semantically equivalent programs we mentioned above—`0`, `find(x, x)`, and `find("␣", "␣")`—are, of course, also observationally equivalent on these two inputs: all three programs evaluate to zero on both inputs. On the other hand, there are observationally equivalent programs that are not semantically equivalent: for example, `find("␣", x)` and `find("␣" ++ "␣", x)`. These programs return -1 on both inputs from $\mathscr{E}$, because neither input is a substring of either `"␣"` or `"␣" ++ "␣"`; the input `"␣␣"` would distinguish between these two programs—yielding the outputs -1 and 0, respectively—but that input is not part of our specification.

As you can see, observational equivalence has two major advantages over true semantic equivalence for our purposes: (1) it is *efficient* to test, and (2) it *over-approximates* true equivalence, that is, even more programs are considered redundant and therefore pruned from the search space. To give you a sense of how much OE reduction prunes the search space, let us revisit Example 3.3.2, where the task was to extract

the first name and repeat it twice. Our implementation of size-based bottom-up search with no pruning enumerates $\approx 500K$ programs before it encounters the solution (Equation 3.3), while the same search algorithm with OE reduction enumerates *fewer than a thousand* programs. Thanks to its simplicity and surprising effectiveness, OE reduction is extremely popular, and used practically in every bottom-up synthesis algorithm.[4]

4: The idea to use OE to prune redundant programs during bottom-up search appeared concurrently in two synthesizers: Transit [14] and Escher [15].

How do we modify Algorithm 2 to use OE reduction? This is a surprisingly simple, one-line change, which we leave an exercise for the reader. Note that OE reduction can be implemented more efficiently if the bank caches the evaluation results of all its programs—something we have already suggested implementing in the previous section anyway, because it helps speed up the evaluation of new terms.

Exercise 3.4.1 Modify Algorithm 2 so that it only stores a single representative for each set of observationally equivalent programs in the bank.

Equational Theory An alternative approach to identifying redundant programs is to ask the user to supply an *equational theory* for the target DSL (in addition to its grammar and interpreter). An equational theory is a set of equations that specify which patterns of terms are equivalent. For example, we could supply the following equational theory for our restricted Stringy DSL:

$$(S_1 \mathbin{+\!\!+} S_2) \mathbin{+\!\!+} S_3 \equiv S_1 \mathbin{+\!\!+} (S_2 \mathbin{+\!\!+} S_3)$$
$$\texttt{find}(S_1, S_1) \equiv \texttt{0}$$

The first equation says that string concatenation is associative, while the second one says that you can always find a string inside itself at index 0. Here each S_i is a *pattern variable*, which stands for an arbitrary term derived from the variable's associated non-terminal; we assume that equations are defined in such a way that any instance of their left- and right-hand side is a syntactically correct term in the grammar.

Intuitively, we consider two terms t_1 and t_2 equivalent under this theory ($t_1 \equiv t_2$) if we can rewrite t_1 into t_2 by substituting subterms matching one side of an equation with their counterparts matching the other side. For example, using the two equations above we can show that:

$$\texttt{x}[\texttt{find}(\texttt{x},\texttt{x}) \mathinner{..} \texttt{find}(\texttt{x},\texttt{"␣"})] \equiv \texttt{x}[\texttt{0} \mathinner{..} \texttt{find}(\texttt{x},\texttt{"␣"})]$$

Note that while observational equivalence necessarily *over-approximates* true semantic equivalence, a sound equational theory must *under-approximate* true equivalence, and most often it is a strict under-approximation. For example, using the two equations above, we cannot show that

$$\texttt{x}[\texttt{0} \mathinner{..} \texttt{0}] \equiv \texttt{"␣"}[\texttt{0} \mathinner{..} \texttt{0}]$$

Although we can add another rule to make this equivalence hold,[5] it is often cumbersome or impossible to add enough rules to capture all the equivalences in a DSL.

5: Can you come up with it? Recall that both sides must be in the grammar!

How should we use an equational theory to prune redundant programs during bottom-up search? A naïve approach is to attempt to prove equivalence between every newly constructed term and every term in the bank. There is, however, a much more efficient way to do this:[6] the main idea is to reduce the equational theory to a *confluent and terminating rewriting system*, *i.e.* a set of unidirectional rewrite rules, such that applying these rule to any term will eventually converge to a unique *normal form*, where no more rules apply. For example, we can turn the above equational theory into a confluent and terminating rewriting system by "orienting its equations" left-to-right:

6: This approach is due to [16]

$$(S_1 +\!\!+ S_2) +\!\!+ S_3 \longrightarrow S_1 +\!\!+ (S_2 +\!\!+ S_3) \tag{3.4}$$

$$\texttt{find}(S_1, S_1) \longrightarrow \texttt{0} \tag{3.5}$$

that is, allowing to rewrite the left-hand side into the right-hand side, but not vice versa.

At this point, the key insight is that we need not consider pairs of terms and try to prove their equivalence; instead we can simply discard any term *that is not in normal form*, *i.e.* whose subterms match the left-hand side of any rule. This does not compromise completeness because for any such term there exists an equivalent term in normal form, which belongs to the same grammar, and hence will eventually also be enumerated. For example, as soon as the search constructs the term $\texttt{find}(\texttt{x}, \texttt{x})$ at height 1, we detect that it matches Equation 3.5, and hence can be discarded; indeed, the normal form of this term—$\texttt{0}$—has already been enumerated at height 0.

Built-in Equivalences If your synthesizer works with a fixed DSL, as opposed to accepting a grammar from the user, you can manually eliminate certain types of redundant terms from the search space by a process called *grammar stratification*.[7] For example, recall the productions for the S nonterminal in our simplified Stringy grammar G_s:

7: One of the papers [17] on the Leon synthesizer contains an elegant formalization of this approach using *attribute grammars*.

$$S ::= \texttt{x} \mid \texttt{"␣"} \mid S +\!\!+ S \mid S[N \,..\, N]$$

This grammar allows generating both terms of the form $(S +\!\!+ S) +\!\!+ S$ and of the form $S +\!\!+ (S +\!\!+ S)$, which is redundant because string concatenation is associative. Can we modify the grammar so that one of these forms (say, the former) is not generated in the first place?

This can be accomplished by *stratifying* the S nonterminal into two levels: S_a for string terms without top-level concatenation and S for all string terms. The two nonterminals can be defined using the following production rules:

$$S_a ::= \texttt{x} \mid \texttt{"␣"} \mid S[N \,..\, N]$$
$$S ::= S_a +\!\!+ S$$

You can readily convince yourself that a term like $(\texttt{x} +\!\!+ \texttt{x}) +\!\!+ \texttt{x}$ does not belong to this grammar, because the left-hand side of a concatenation must be an S_a, and S_a cannot be a concatenation term.

Equivalence Reduction: Comparison We have now seen three different approaches to equivalence reduction: observational equivalence, equational theory, and grammar stratification with built-in equivalences. What are their comparative advantages and disadvantages?

Observational equivalence reduction is probably the most popular of the tree and for a good reason: it requires no additional input from the user (beyond the interpreter, which we already have), and prunes more terms that the other two approaches. On the other hand, OE can get costly to check if the number of examples or the size of the outputs is large. More importantly, because OE is parametrized by input examples, it can only be used with example-based synthesis. Moreover, when new examples are added—as routinely happens in the context of CEGIS—OE equivalence classes become more fine grained; hence the search cannot be resumed from the existing expression bank, since this bank might be missing some representatives for the new equivalence classes; instead the search must be restarted from scratch.

Instead, both equational theory and grammar stratification are entirely independent of the behavioral specification, and hence can be used for non example-based synthesis and do not require restarts in the context of CEGIS. They are also very efficient to check (with grammar factoring not requiring any extra checks during synthesis whatsoever). The downside, of course, is that they require additional domain knowledge about which DSL expressions are equivalent.[8]

8: One option for mitigating this limitation is to infer an equational theory automatically using tools like Ruler [18] or QuickSpec [19].

How specific are these equivalence reduction techniques to the search algorithm? Observational equivalence works for bottom-up enumeration because it needs to evaluate the expressions in order to determine their equivalence; in top-down search, most of the expressions the search is considering are incomplete, and hence cannot be evaluated. There is, however, a way to apply partial OE reduction to incomplete programs.[9] Consider two incomplete programs, `x[0 .. N]` and `x[find(x,x) .. N]`; we can discard the second one as redundant because: (1) these two terms only differ in complete sub-terms `0` vs `find(x,x)`, and (2) these complete sub-terms are observationally equivalent.

9: As far as we know, this technique was first used in Euphony [20].

On the other end of the spectrum, grammar factoring happens before the search even begins, and can be combined with any search technique whatsoever (even stochastic or constrained-based search discussed in in the following chapters). Normality checking with equational theories can technically be combined with any enumerative search, bottom-up or top-down; however, bottom-up search enables an important optimization: because we already know that all terms stored in the bank are in normal form, there is no need to check if a rewrite rule matches any *subterms* of the newly constructed term (it doesn't), we only need to match the root of the new term against the left-hand side of the rule. This check, referred to as *root normality*, is much more efficient than full normality checking.

3.4.2 Pruning Infeasible Programs

The second kind of search space pruning—pruning *infeasible programs*—is typically applied during top-down enumeration. Recall from Algorithm 1 that basic top-down search enumerates incomplete programs, but only

Algorithm 3 Top-down search with infeasibility pruning

Input: Example-based expression syntheis problem $E^{\mathscr{E}}_{sy} = (\mathsf{M}, G, \mathscr{E})$
Output: Expression t^* such that $[\![t^*]\!](\mathscr{E}) = \mathsf{M}(\mathscr{E})$

```
1:  procedure Top-Down-Search-Pruning(M, G = (N, Σ, S, a, δ), E)
2:      wl ← [S]                                   ▷ Initialize the worklist
3:      while wl ≠ [] do
4:          τ ← wl.dequeue                         ▷ Dequeue the first element
5:          if F(E^E_sy, τ) then                   ▷ τ is feasible
6:              if τ ∈ T_Σ then                    ▷ τ is complete
7:                  return e                       ▷ Solution found
8:              else
9:                  for τ' ∈ New-Terms(τ, G) do
10:                     wl.enqueue(τ')             ▷ Add to worklist
```

discards programs when they are complete and can be executed. It would be great if instead we could somehow determine that an incomplete program is infeasible: that is, no complete program derived from it can possibly satisfy the specification. In this case, the infeasible program could be discarded from the worklist, preventing the synthesizer from exploring all of its descendants in the search tree.

More formally, let $E^{\mathscr{E}}_{sy} = (\mathsf{M}, G, \mathscr{E})$ be an example-based synthesis problem and $\tau \in \mathscr{T}_{\Sigma \cup N}$ be an incomplete program. We can formalize the feasibility check as a boolean oracle $\mathsf{F}(E^{\mathscr{E}}_{sy}, \tau)$, which must satisfy the following properties:

1. If $\mathsf{F}(E^{\mathscr{E}}_{sy}, \tau) = \mathsf{false}$, then τ is truly infeasible; that is, no complete programs t derived from τ satisfies the specification: $\forall t \in \mathscr{T}_{\Sigma}.\tau \rightarrow^* t \Rightarrow [\![t]\!](\mathscr{E}) \neq \mathsf{M}(\mathscr{E})$.
2. If $\mathsf{F}(E^{\mathscr{E}}_{sy}, \tau) = \mathsf{true}$, then τ might still be infeasible, *i.e.* the feasibility check is over-approximate; however when τ happens to be a complete program, then $\mathsf{F}(E^{\mathscr{E}}_{sy}, \tau)$ must return true only if τ is actually correct (it is easy to do so simply by implementing F as $[\![\tau]\!](\mathscr{E}) = \mathsf{M}(\mathscr{E})$ when τ is complete).

With such an over-approximate feasibility oracle at hand, we can modify the top-down search algorithm to prune infeasible programs as shown in Algorithm 3. The only difference from Algorithm 1 is in line 5 where we check if the current candidate τ is feasible, and otherwise it simply gets discarded. (The auxiliary procedure New-Terms remains the same as in Algorithm 1.) Note that a trivial oracle, which always returns true unless the program is complete, is valid for any DSL but reduces the above algorithm to plain top-down search.

For an example of a feasibility oracle in action, recall Example 3.3.2, where the task was to extract first names and repeat them twice:

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

Imagine that we are trying to solve this problem using top-down enumeration, and at some point during the search the incomplete program at the front of the worklist is $\tau = \mathsf{x}[N \,..\, N]$. An over-approximate feasibility oracle can soundly reject τ based on the consideration that at least in one of the input-output examples (and in fact in both), the output is not a substring of the input; hence no instantiation of the N non-terminals

in τ could possibly lead to a solution. Discarding $\mathsf{x}[N \,..\, N]$ allows Algorithm 3 to avoid exploring the entire branch of the search tree below this term (this branch was partly expanded in Figure 3.1).

The only remaining question is: how do we obtain a non-trivial feasibility oracle F? Ultimately, the oracle relies on some properties of the DSL semantics, and hence requires additional domain knowledge from the DSL designer. There are two common approaches to specifying such domain knowledge: *inverse semantics* and *abstract semantics*. Let us consider both of them in turn.

Inverse Semantics With this approach, the DSL designer specifies how the input-output specification can be propagated from a top-level expression to its sub-expressions.[10] More concretely, recall that the DSL *semantics* $[\![t]\!](\epsilon)$ maps a complete term t and an input environment ϵ to a value. Inverse semantics $[\![\tau]\!]^{-1}(\epsilon; v)$ (also called a *witness function*), maps an incomplete program τ, an input environment ϵ, and an output value v to the set of *all possible output values* for the leftmost non-terminal inside τ. For example, $[\![\mathsf{x}[N \,..\, N]]\!]^{-1}([\mathsf{x} \mapsto \texttt{"Nadia␣Polikarpova"}]; \texttt{"Nadia"}) = \{0\}$; this is because the output value `"Nadia"` occurs exactly once in the evaluation of x, and the starting position of this occurrence is 0.

10: This approach to pruning is the cornerstone of synthesis using *version-space algebras* [5, 21], which we will discuss in more detail in Section 3.6. However, it has also been used in purely enumerative synthesizers, such as λ^2 [22].

Inverse semantics can be defined as a recursive function over the structure of incomplete terms. Figure 3.3 shows a partial definition of inverse semantics for the Stringy DSL. For the sake of brevity, instead of defining a function that returns a set, in the figure we define a relation $v' \in [\![\tau]\!]^{-1}(\epsilon; v)$; it is straightforward to turn this definition into a function by returning all such v' that satisfy the relation.

The first two rules are simply the base cases of the recursion, where the required output value v is returned directly. To account for the situation when certain DSL operators are not efficiently invertible, we allow the inverse semantics to return a special value $\top$ to indicate that the inverse is the set of all values of the appropriate type. This is the case in the rule Substr-0, which allows an arbitrary string s' in $[\![\tau[N \,..\, N]]\!]^{-1}$, to reflect the fact that the substring operator is not efficiently invertible in its first argument. Technically, we could return the set of all strings that have the output s as a substring, but this set is not efficiently computable or finitely representable. On the other hand, the rule Substr-1 efficiently inverts the substring operator *wrt.* its second argument (the starting index), once the first argument (the string) is fixed. An analogous rule Substr-2 does the same for the third argument (the end index).

The Concat-0 rule is an example where inverse semantics returns a set that is neither a singleton nor $\top$: this rule inverts string concatenation *wrt.* its first argument and the output s by returning all non-empty prefixes of s. Finally, Concat-1 applies when the first argument to concatenation is fixed and evaluates to s_0; if s_0 is a non-empty prefix of s, it returns a singleton set (the rest of s), and otherwise it returns the empty set.

Once we have defined the inverse semantics, it is easy to define the feasibility oracle for τ by checking that $[\![\tau]\!]^{-1}$ is non-empty for all examples:

$$\mathsf{F}((\mathsf{M}, G, \mathscr{E}), \tau) = \bigwedge_{\epsilon \in \mathscr{E}} [\![\tau]\!]^{-1}(\mathsf{M}(\epsilon)) \neq \emptyset$$

$$\textsc{S-Base}\ \frac{}{s \in [\![S]\!]^{-1}(\epsilon; s)} \qquad \textsc{N-Base}\ \frac{}{n \in [\![N]\!]^{-1}(\epsilon; n)}$$

$$\textsc{Substr-0}\ \frac{\tau \notin \mathcal{T}_\Sigma}{s' \in [\![\tau[N \mathinner{..} N]]\!]^{-1}(\epsilon; s)}$$

$$\textsc{Substr-1}\ \frac{[\![t]\!](\epsilon) = s_0 \quad \tau \notin \mathcal{T}_\Sigma \quad s_0[i \mathinner{..} j] = s \quad v \in [\![\tau]\!]^{-1}(\epsilon, i)}{v \in [\![t[\tau \mathinner{..} N]]\!]^{-1}(\epsilon; s)}$$

$$\textsc{Substr-2}\ \frac{[\![t_0]\!](\epsilon) = s_0 \quad [\![t_1]\!](\epsilon) = i \quad s_0[i \mathinner{..} j] = s \quad v \in [\![\tau]\!]^{-1}(\epsilon, j)}{v \in [\![t_0[t_1 \mathinner{..} \tau]]\!]^{-1}(\epsilon; s)}$$

$$\textsc{Concat-0}\ \frac{\tau \notin \mathcal{T}_\Sigma \quad 0 < i < \mathsf{len}(s) \quad v \in [\![e]\!]^{-1}(\epsilon, s[0 \mathinner{..} i])}{v \in [\![\tau \mathbin{+\!\!+} S]\!]^{-1}(\epsilon; s)}$$

$$\textsc{Concat-1}\ \frac{[\![t]\!](\epsilon) = s[0 \mathinner{..} i] \quad 0 < i < \mathsf{len}(s) \quad v \in [\![\tau]\!]^{-1}(\epsilon, s[i \mathinner{..} \mathsf{len}(s)])}{v \in [\![t \mathbin{+\!\!+} \tau]\!]^{-1}(\epsilon; s)}$$

Figure 3.3: Inverse semantics of the substring and concatenation operators in Stringy.

Returning to task from the beginning of this section, we can compute that

$$[\![\mathsf{x}[N \mathinner{..} N]]\!]^{-1}([\mathsf{x} \mapsto \texttt{"Nadia Polikarpova"}]; \texttt{"Nadia Nadia"}) = \emptyset$$

because none of the rules apply here: Substr-1 is the only rule that satisfies the structure of the term, but its third premise, requiring that s be a substring of s_0, does not hold. Consequently, the feasibility oracle based on the inverse semantics returns false for this partial term: $\mathsf{F}(E^{\mathscr{E}}_{sy}, \mathsf{x}[N \mathinner{..} N]) = \mathsf{false}$.

You might have noticed that the two Concat rules do not quite satisfy our requirements for an over-approximate feasibility oracle: for example $\mathsf{F}(E^{\mathscr{E}}_{sy}, \mathsf{x}[0 \mathinner{..} 0] \mathbin{+\!\!+} S)$ would return false, even though the term is actually feasible, because Concat-1 does not allow the left-hand side to be an empty string. This does not cause incompleteness, however, since any term of the form $\mathsf{x}[0 \mathinner{..} 0] \mathbin{+\!\!+} t$ is functionally equivalent to just t. This is an example of how inverse semantics can be used to also prune some redundant terms, in addition to detecting infeasible ones.

Abstract Semantics If the inverse semantics approach executes an incomplete program backwards starting from the outputs, the *abstract semantics* approach executes it forward, simply assuming that the missing sub-terms can have any value whatsoever. This approach has its roots in an influential static analysis technique called *abstract interpretation*. In short, the main idea of abstract interpretation is as follows:

1. define an *abstract domain* $\mathscr{A}$, whose elements are abstract values that compactly represent sets of concrete values;
2. for each DSL operator, define its *abstract semantics* $[\![\cdot]\!]^{\#}$, which maps abstract values to abstract values and over-approximates the concrete semantics (we will explain what it means on an example shortly).

Traditionally, the abstract semantics is then used to "evaluate" a program on unknown inputs, in order to prove some properties about its behavior on all inputs. In our case, however, we will use the abstract semantics

$$\textsc{S-Base}\ \frac{}{[\![S]\!]^{\#}(\epsilon) = (0, \infty)} \qquad \textsc{S-Complete}\ \frac{n = \mathsf{len}([\![t]\!](\epsilon))}{[\![t]\!]^{\#}(\epsilon) = (n, n)}$$

$$\textsc{Substr}\ \frac{[\![\tau_0]\!]^{\#} = (l_0, u_0) \quad [\![\tau_1]\!]^{\#} = (l_1, u_1) \quad [\![\tau_2]\!]^{\#} = (l_2, u_2)}{[\![\tau_0[\tau_1 \mathinner{..} \tau_2]]\!]^{\#}(\epsilon) = (\max(0, l_2 - u_1), \min(u_0, u_2 - l_1))}$$

$$\textsc{Concat}\ \frac{[\![\tau_0]\!]^{\#} = (l_0, u_0) \quad [\![\tau_1]\!]^{\#} = (l_1, u_1)}{[\![\tau_0 +\!\!+ \tau_1]\!]^{\#}(\epsilon) = (l_0 + l_1, u_0 + u_1)}$$

Figure 3.4: One possible abstract semantics of a subset of Stringy; here the abstract domain is the range of string lengths.

to evaluate an incomplete program, which is similar in the sense that non-terminals in a program can be treated as additional inputs, whose values are unknown.

Let us apply this approach to Stringy. One popular choice for the abstract domain is so-called *interval domain*: let the abstract value for an string-typed term τ_S be a pair (l, h) where l and h are integers representing the minimum and maximum possible *length* of the string that τ_S could evaluate to; for an integer-typed term τ_N, its abstract value is simply the range of values τ_N could evaluate to. In this case, we can define the abstract semantics of the substring and concatenation operators as shown in Figure 3.4. The rule S-Base says that the abstract value of completely unknown string term S can be any natural number. On the other end of the spectrum, the rule S-Complete says that for a complete term t, the abstract value is a singleton interval, which contains just the length of the concrete string that t evaluates to. The other two rules deal with partially defined string terms. For example, the Substr rule says that the length of $\tau_0[\tau_1 \mathinner{..} \tau_2]$ is between the minimal and maximal difference between the values of τ_1 and τ_2, except that it also cannot be negative or longer than the original string.[11]

11: For simplicity we assume here that neither of the index bounds is negative, and that neither of the index intervals is empty.

It is easy to show that this abstract semantics over-approximates the concrete semantics, namely: for any incomplete term τ, any complete term t derived from it ($\tau \to^* t$), and any input ϵ, $[\![t]\!](\epsilon) \in [\![\tau]\!]^{\#}(\epsilon)$ (where $[\![\cdot]\!]^{\#}$ is interpreted as an inclusive interval). This property makes it possible to use the abstract semantics to define a feasibility oracle. Given an incomplete program τ and an input example ϵ, we can simply check whether the corresponding desired output is allowed by the abstract semantics, that is:

$$\mathsf{F}((\mathsf{M}, G, \mathscr{E}), \tau) = \bigwedge_{\epsilon \in \mathscr{E}} \mathsf{M}(\epsilon) \in [\![\tau]\!]^{\#}(\epsilon)$$

If this check fails on some input ϵ, then we conclude that no complete term t derived from τ can possibly evaluate to $\mathsf{M}(\epsilon)$, because its set of possible outputs does not include $\mathsf{M}(\epsilon)$; and hence τ is infeasible.[12]

12: This approach to pruning infeasible programs was pioneered by Morpheus [23] for table-manipulating programs, where a natural abstract domain is (intervals over) the table dimensions.

Let us know consider a Stringy synthesis problem with the following input-output example:

`"Nadia"` $\longrightarrow$ `"Nadia␣Nadia"`

Consider now the incomplete program $\mathsf{x}[N \mathinner{..} N]$. Its abstract value on

the example is:

$$[\![\mathsf{x}[N \,..\, N]]\!]^{\#}([\mathsf{x} \mapsto \texttt{"Nadia"}]) = (\max(0, -\infty), \min(\mathsf{len}(\texttt{"Nadia"}), \infty)) = (0, 5)$$

At the same time, the length of the desired output is 11, which is outside of the range of possible output lengths, hence the partial program is infeasible.[13]

13: Note that this particular abstract domain is not precise enough to prove infeasibility of the same incomplete program on the original example `"Nadia␣Polikarpova"` → `"Nadia␣Nadia"`.

Finally, we would like to note that although the infeasibility pruning approaches based on inverse and abstract semantics historically appeared as alternatives, in reality they represent two orthogonal dimensions. Along one dimension, the semantics might propagate values forward (bottom-up) or backward (top-down). Along the other dimension, it might use different abstract domains to represent sets of possible concrete values. In fact, the inverse semantics approach we presented above can also be thought of as using an abstract domain, albeit a very simple one: a set of concrete values is either represented explicitly as a list (when it is small enough), or becomes the default value $\top$ (when it is too large).

Infeasibility in Bottom-Up Search So far we have only considered the notion of infeasibility for incomplete programs, in the context of top-down enumeration. Does infeasibility pruning even make sense for bottom-up search? Yes, for bottom-up search, a term t is infeasible if combining it with arbitrary other terms cannot possibly lead to a solution. In the context of example-based synthesis, the simplest example would be a term t whose evaluation fails (for example, if we changed the semantics of substring operation in Stringy to fail whenever any index is out of bounds). In this case, combining t with any other term would still fail, and hence cannot possibly satisfy the spec (assuming the DSL is not lazy, and the spec does not ask the program to fail).

If the spec goes beyond input-output examples, more interesting cases of bottom-up infeasibility can be exploited: for example, if the spec limits the amount of resources a program can use, we can through out a subterm once it exceeds the resource bounds.

3.5 Weighted Enumeration

This section is still under construction.

3.6 Representing Large Spaces of Programs

This section is still under construction. VSAs, tree automata.

Stochastic Synthesis 4

In Chapter 3 we discussed a family of deterministic enumeration algorithms that can systematically explore the search space of the program synthesis problem. However, in certain scenarios these approaches are doomed to fail as the smallest correct programs is too large to be obtained via size- or height-based enumeration. In this chapter, we describe a stochastic approach to enumeration that can sometimes solve synthesis problem that systematic enumeration will fail on. Stochastic approaches are sound—i.e., they only output correct solutions—but incomplete—i.e., they might miss parts of the search space.

Like we did in earlier chapters, we target example-based expression synthesis problems $E^{\mathscr{E}}_{sy} = (\mathsf{M}, G, \mathscr{E})$, defined by the membership oracle M, a grammar G, and a set of inputs $\mathscr{E}$. In other words, we assume that the behavioral specifcaiton is given by input-output examples, and the structural specificaiton is given by a regular tree grammar.

4.1 Montecarlo Markov Chain Synthesis

While there are many ways to introduce stochasticity in the synthesis process, one of the principled approaches that has found adoption in practice is to use the theory of Montecarlo Markov Chains (MCMC) [24], a random-search formalism that provides desirable theoretical guarantees. For our purposes a Markov Chain is an infinite state machine (a graph) where the set of nodes is the set of all programs in the search space.

The key idea of a MCMC synthesis algorithm is to "explore" this infinite graph randomly and hopefully quickly find a program that solves the synthesis problem. In particular, the mechanism we present next is called Metropolis Hasting. Intuitively, an MCMC synthesis algorithm explores a program e at each iteration starting from an initial program. At each iteration, the algorithm randomly chooses a program (we assume a bound k on the depth of the possible programs) to move to in the next iteration and the choice is made considering two factors: 1) the edges between nodes/programs in the graph are labeled with probabilities W that bias the choice of the next program to consider, and 2) a quality function Q that tells us how close a program is to a correct one. More formally, given a program e a possible program e' is chosen as a program candidate with probability $W(e, e')$. If the quality $Q(e')$ of e' is higher than $Q(e)$, the algorithm sets e' to the program to explore in the next iteration, otherwise, e' is choosen probability $Q(e')W(e', e)/Q(e)W(e, e')$. In practice, it is common that $W(e', e) = W(e, e')$ and therefore one can simply choose the next program with probability $Q(e')/Q(e)$. Choosing programs that decrease the quality with non-zero probability is crucial to avoid getting stuck in local minima.

This algorithm is based on property of Markov Chains known as the Fundamental Theorem of Markov Chains, which states that if a Markov chain satisfies a few technical requirements, then randomly moving through the Markov Chain by following the transition probabilities will

Algorithm 4 MCMC search algorithm

Input: Example-based expression synthesis problem $E^{\mathscr{E}}_{sy} = (\mathsf{M}, G, \mathscr{E})$
Output: Expression e^* such that $[\![e^*]\!](\mathscr{E}) = \mathsf{M}(\mathscr{E})$

```
1:  procedure Stochastic-Search(M, G = (N, Σ, S, a, δ), E, k)
2:      e ← initial_program                          ▷ Choose initial program
3:      while Q(e) < ∞ do                  ▷ Choose a random neighbour of e
4:          e' ← random_neighbour(e)                 ▷ Compute acceptance rate
5:          if Q(e') > Q(e) then   ▷ Quality improved, set e' as th next program
6:              e ← e'
7:          else
8:              A(e, e') ← Q(e')W(e', e)/Q(e)W(e, e').
9:              e ← e' with probability A(e, e')     ▷ Choose next program
10:     return e                                     ▷ Solution found
```

lead us to any state (in the limit) with probability proportional to its quality. We refer the reader to a survey on this topic at the moment.

4.2 Instantiating MCMC for Expression Synthesis

The quality function measures how good a solution is and here we assume that it reaches infinity when a correct program is found. For example, we can consider the following function that measures the quality of a program by considering how many examples the program is correct on.

$$Q_{\mathscr{E}}(e) = \frac{1}{|\{\epsilon \mid \epsilon \in \mathscr{E}.[\![e]\!](\epsilon) \neq \mathsf{M}(\epsilon)\}|}$$

It is possible to design other quality functions that consider different distances—e.g., by considering how close one is to making an example correct for numerical values.

Given a program e consider its derivation tree where each node n is labeled with corresponding nonterminal $NT(n)$ and the corresponding depth $depth(n)$. One possible strategy to get a new neighbour out of e is to pick a node n in e uniformly at random and then replacing n with a new subtree from $L_{\leq k'}(L)$ where $k' = k - depth(n)$ and $L = NT(n)$.

This choice of $random\_neighbour(e)$ satisfies the properties needed by MCMC to work in theory, i.e., $W(e, e') > 0$ iff $W(e', e) > 0$, and it will guarantee that better programs are visited more often.

5 Constraint-based Synthesis

In this chapter, we show how to solve expression synthesis problems using constraint solvers. We first describe how constraint solvers work and discuss some simplifications needed for enabling constraint-based encodings of synthesis problems—i.e., bounding the depth or size of the synthesized program. Then we present three different encodings for the expression synthesis problem: one that represents terms as trees, and two that represent terms as sequences of statements (i.e., in single static assignment form). The last two encodings differ in how they represent the terms.

5.1 Logic and Solvers

Constraint solvers, most commonly known now as Satisfiability Modulo Theory (SMT) solvers, have been one of the greatest achievements in formal methods. A constraint solver takes a set of logical constraints and produces a *satisfying assignment* for the variables appearing in the constraints when one exists, otherwise it returns *unsatisfiable*.

Example 5.1.1 Recall Example 2.3.1 where the goal was to synthesize a program (using only bit-vector operations) that "isolates" the rightmost 0 bit:

$$\texttt{10010101} \longrightarrow \texttt{00000010}$$
$$\texttt{01101111} \longrightarrow \texttt{00010000}$$

In this example, we assume for simplicity that our goal is to find a program that correctly behaves just on these two example. Furthermore, we assume that the programmer knows that the correct program is of the following form

$$P(?) = \texttt{bvand}(\texttt{bvnot}(\texttt{x}), \texttt{bvadd}(\texttt{x}, ?))$$

and they are trying to look for an integer value to substitute in the variable `?` that makes the program correct on the two examples. Formally the goal is to find a value v such that the following formula is true:

$$[\![P(v)]\!](\texttt{10010101}) = \texttt{00000010} \wedge [\![P(v)]\!](\texttt{01101111}) = \texttt{00010000}$$

If we unroll the semantics of $P(v)$ we get the following formula:

$$\begin{array}{rl} & \sim \texttt{10010101} \,\&\, (\texttt{10010101} + v) = \texttt{00000010} \\ \wedge & \sim \texttt{01101111} \,\&\, (\texttt{01101111} + v) = \texttt{00010000} \end{array}$$

which a constraint solver can solver for us! In particular, SMT solvers support all the bit-vector operations appearing in such a constraint. A possible solution an SMT solver will return for this constraint is the

value v = `00000001`, which corresponds to the intended program:

$$P(\texttt{00000001}) = \texttt{bvand}(\texttt{bvnot}(\texttt{x}), \texttt{bvadd}(\texttt{x}, \texttt{00000001}))$$

A modern SMT solver like Z3 or CVC5 can solve this constraint in milliseconds.

Our previous example showed how an SMT solver can help us find values for one (or more) constants in a program, so that the resulting program is correct on some examples. In general, this approach can be taken whenever the semantics of a program only involves operations that are supported by existing SMT solvers and the only unknowns are scalar values (often called first-order variables) and not actual programs (often called second-order variables). Modern SMT solvers support a large variaty of operations and first-order variable types, some of which we list here:

BitVectors arbitrary bit-vector operations over arbitrary bit-widths as shown in our previous example.

Linear Integer Arithmetic linear arithmetic over integer variables (i.e., variables can only be summed and not multiplied).

Arrays operations involving reading and writing from indexed arrays.

On top of these basic data-types, constraints (even over different types) can be combined using logical operations such as conjunction and negation. The reader can find more informations about how SMT solvers in the following survey [25].

Bounding size terms, grammar In the previous example, we used the SMT solver to fill in an unknown constant in a program with a *fixed* structure. But in general, we would also like the solver to help us determine the structure of the program. Recall that solving example-based synthesis problem $E^{\mathcal{E}}_{sy}$ with set of examples $\mathcal{E} = \{\epsilon_1, \ldots, \epsilon_n\}$ and grammar $G = (N, \Sigma, S, a, \delta)$ requires finding a program in the language of G that behaves correctly on all the examples (according to the membership oracle M). In this chapter, we only focus on grammars for which $[\![e]\!]$ can be expressed as an SMT formula for every e in the language of G. In particular, we will use Brrvy language for our examples, since the theory of bit-vectors is well supported by SMT solvers whereas the theory of strings is just now making its way into modern constraint solving. Formally, we are looking for program e^* that is a valid solution to the formula $\phi(e^*) \equiv \bigwedge_{\epsilon \in \mathcal{E}} . [\![e^*]\!](\epsilon) = \mathsf{M}(\epsilon)$. Even though the semantics of e^* is definable using SMT constraints, e^* is not a first-order variable and is the unknown of our formula!

Modeling a second order variable of unknown size—i.e., the function we are trying to synthesize—using only first order variables—i.e., variables of type int, bit-vector, etc.—is theoretically impossible. However, we have already found a similar issue when discussing enumeration algorithms in Chapter 3 where we enumerated terms up to a certain size/height, and our algorithms were complete in the sense that if there is a solution within the set bound, a solution will be found.

In the next sections, we will show three encodings that provide a similar approach and can generate sets of constraints that can find any solution to the synthesis problem if a solution exists within a certain bound.

5.2 Tree-based Encoding

As we discussed earlier, we know a solver can find unknown first-order constants. However, how can we use this ability to find a term (of a bounded size) from a grammar? The main idea is to imagine a big tree with unlabeled nodes and then simply ask the solver to label its nodes with grammar productions (possibly leaving some nodes in the lower levels of the tree unused).[1] As long as we bound the height of the trees, we will have a bound on how many nodes the tree can have. The tree in Figure 5.1 is an example of a tree of height 2 where the nodes are unlabeled. Note that the tree in Figure 5.1 is binary and can only accommodate productions that have at most 2 children.

Example 5.2.1 Consider the problem of synthesizing a term of height at most 2 in the Bitvy grammar:

$$B ::= \texttt{x} \mid \texttt{0} \mid \texttt{1} \mid \texttt{bvnot}(B) \mid \texttt{bvadd}(B,B) \mid \texttt{bvsub}(B,B) \mid \texttt{bvor}(B,B) \mid \texttt{bvand}(B,B) \mid \texttt{bvxor}(B,B)$$

Because, the height is at most two, the nodes at heights 0 and 1 can have arbitrary arity (i.e., they can have children), but the nodes at height 2 can only be leaves (i.e., each of them can only be `x`, `0`, or `1`). Our encoding will include *structural constraints* that enforce this structure and that guarantees that the children of each production are nodes of the appropriate non-terminals.

Because the maximum arity of an operation is 2, we can assume that all trees can be somehow fitted into the tree shown in Figure 5.1 by appropriately "ignoring" certain nodes when necessary. For example, if we set p^0 to be the production $\texttt{bvnot}(B)$, then p^1_1 will represent the child of `bvnot` whereas all the subtree with root p^1_2 will be ignored—i.e., when mapping the tree to an actual term we will ignore what productions have been mapped to this tree. This idea is illustrated in Figure 5.2.

While we have presented the intuition behind the constraints that capture our structural specification, we still have to ensure that the generated tree complies with the behavioral specification. Consider again our input-output example $\epsilon = \texttt{10010101}$ such that:

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

We will impose *behavioral constraints* that map the synthesized term (tree) to a value. To achieve this goal, we can simply state constraints of the following form:

- if p^1_2 is set to the production `x`, then the result $\mathrm{v}^\epsilon_1(2)$ of evaluating the tree with root p^1_2 on input `10010101` is `10010101` itself.
- if p^1_2 is set to the production `0`, then the result $\mathrm{v}^\epsilon_1(2)$ of evaluating the tree with root p^1_2 on input `10010101` is `00000000`.
- ...
- if p^0 is set to the production $\texttt{bvnot}(B)$, then the result v^ϵ_0 of evaluating the tree with root p^0 on input `10010101` is the negation of the result $\mathrm{v}^\epsilon_1(1)$ of evaluating the tree with root p^1_1 on `10010101`.

1: An encoding similar to the one illustrated in this section was first introduced in [26].

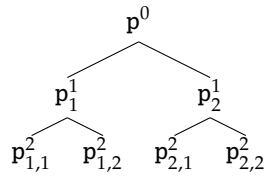


Figure 5.1: A tree with unknown productions. For each node, the variable p^i_π denotes that the node is at height i in the tree and can we reached by following the path π from the root (e.g., $\pi = 1, 2$ indicates that this node is the second child of the first child of the root node).

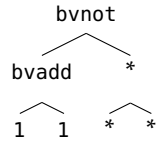


Figure 5.2: Tree corresponding to the expression `bvnot(bvadd(1,1))`. Because the production corresponding to p^0 is `bvnot` and has arity 1, the values of the productions in the right subtree are irrelevant (hence the $*$).

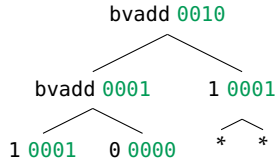


Figure 5.3: Tree corresponding to the expression `bvadd(bvadd(1,0),1)`. Because the production corresponding to p^1_2 is `1`, the values of the productions for $\mathrm{p}^2_{2,1}$ and $\mathrm{p}^2_{2,2}$ are irrelevant (hence the $*$). Last four digits of the values for v^ϵ_i (where $\epsilon = \texttt{10010101}$) at each node are shown in green.

- if $\mathtt{p}^0$ is set to the production $\mathtt{bvadd}(B, B)$, then the result $\mathtt{v}_0^\epsilon$ of evaluating the tree with root $\mathtt{p}^0$ on input `10010101` is the result $\mathtt{v}_1^\epsilon(1)$ of evaluating the tree $\mathtt{p}_1^1$ on `10010101` plus the result $\mathtt{v}_1^\epsilon(2)$ of evaluating the tree $\mathtt{p}_2^1$ on `10010101`.
- ...

The final goal is to find a solution to the constraints that makes the value of $\mathtt{v}_0^\epsilon$ (i.e., the result of evaluating the tree rooted at $\mathtt{p}^0$) to `00000010`.

In this example, $\mathtt{bvadd}(\mathtt{bvadd}(\mathtt{1}, \mathtt{0}), \mathtt{1})$ is a solution and the corresponding tree is shown in Figure 5.3 together with the corresponding last four digits of the values for $\mathtt{v}_i^\epsilon$ at each node (in green).

In the rest of the section, we formalize the encoding illustrated by the above example. First, we are going to fix D to be the maximum depth of the tree we are building and K be the maximum arity of a production appearing in the grammar. In Example 5.2.1, D was 2 and K was 2.

The tricky part of the encoding is that we need a naming convention for accessing the nodes in our trees so that we can talk about what nodes are the children of what nodes. As illustrated in Figure 5.1, a node at depth d can be identified by a path, i.e. a sequence of child choices $[c_1, \ldots, c_d]$, where each c_i is a number in $[0..K-1]$. In Example 5.2.1 the root was labeled as $\mathtt{p}^0$ (i.e., it's not the child of any node), whereas the right child of the root was $\mathtt{p}_2^1$ (i.e., it was the second child of the root and it's a depth 1). The left child of $\mathtt{p}_2^1$ is the node $\mathtt{p}_{2,1}^2$. If the max arity were 3, the third child of $\mathtt{p}_2^1$ would be the node $\mathtt{p}_{2,3}^2$.

Uninterpreted functions Before we continue with our encoding, we describe a theory that is supported by modern SMT solvers and allows us to efficiently model our encoding: uninterpreted functions.

An uninterpreted function is a function signature with no pre-assigned meaning. When we set constraints over the values of such a function, the goal of the SMT solver becomes to find an interpretation of the function that makes the constraints true. As long as the domain of the function is finite, the SMT solver can always build an interpretation for the finitely many values that are relevant to the constraints. In our example, an uninterpreted function $\mathtt{p}^2 : [0..K-1] \times [0..K-1] \mapsto [0..|\delta - 1|]$ (where $|\delta|$ is the number of productions in the grammar), can be used to assign production identifiers to all nodes at depth 2. For readability, in the rest of the section we often write $\mathtt{p}^2(1, 2)$ instead of $\mathtt{p}_{1,2}^2$ to illustrate that $\mathtt{p}^2$ is indeed a function. Furthermore, we often write $\mathtt{p}^2(1, 2) = op(N_1, \ldots, N_j)$ to denote a specific production rather than the corresponding numerical value in $[0..|\delta - 1|]$ associated with it.

We will also use uninterpreted functions to store the results of evaluating the subtrees on the input examples, and to record what nonterminal generates the production of each node.

Variables We are now ready to state what the variables appearing in our encoding are:

- D uninterpreted function variables $\mathtt{n}^0, \ldots, \mathtt{n}^{D-1}$, where each $\mathtt{n}^i$ encodes all nonterminals at level i in the tree. Each $\mathtt{n}^i : [0..K-1]^i \mapsto$

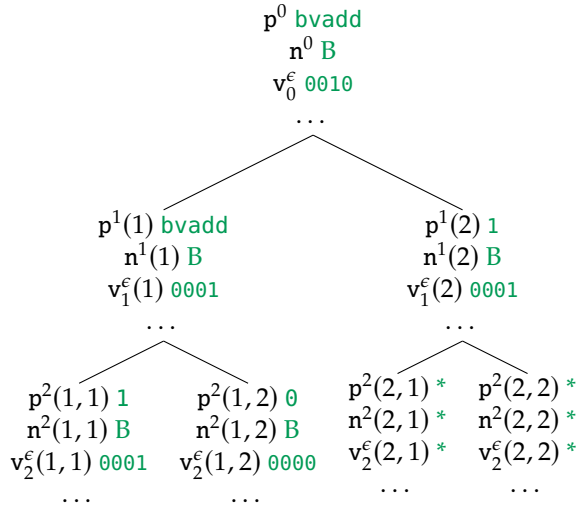


Figure 5.4: Illustration of what variables are mapped to each node in the tree. We also augment all the variables with values in green to show what values they would have for the tree `bvadd(bvadd(1, 0), 1)` from Example 5.2.1 and with ϵ = `10010101`.

$[0..|N|-1]$ is an uninterpreted function from paths of length i to nonterminals identifiers

- D variables $\mathrm{p}^0, \ldots \mathrm{p}^{D-1}$, which are like n^i, but map paths to productions identifiers
- For each example ϵ, there are D variables $\mathrm{v}_0^\epsilon, \ldots, \mathrm{v}_{D-1}^\epsilon$, which are uninterpreted functions that map each node to the value it evaluates to in the example ϵ.

The role of each variable is illustrated in Figure 5.4. Intuitively, each node is assigned a production, a nonterminal, and the values computed by evaluating the tree rooted at that node on the given examples.

Constraints Now that all variables are in place, we can state the constraints needed to ensure that a valid tree that correctly produces valid outputs is built.

Structural constraints: ensure that the generated tree belongs to the grammar.

- The root node should be the initial non-terminal of the grammar

$$\mathrm{n}^0 = S \tag{5.1}$$

- For each path, if a node at that path is assigned a certain production, its children are appropriate non-terminals. Because we are interested in the children of a node, we should state this constraint only for non-leaf nodes. Thus the following constraint is added for any depth $d < D$:

$$\begin{aligned}&\forall i_1, \ldots, i_d < K.\\ &\quad \mathrm{p}^d(i_1, \ldots, i_d) = op(N_0, \ldots, N_j) \Rightarrow\\ &\qquad \mathrm{n}^{d+1}(i_1, \ldots, i_d, 0) = N_0 \wedge \ldots \wedge \mathrm{n}^{d+1}(i_1, \ldots, i_d, j) = N_j\end{aligned}$$

- If a node is a certain non-terminal, it can only be one of the corresponding productions (at the last level, only 0-arity productions are allowed). We write $r \in \delta(N)$ to denote a production r with left nonterminal N and $r \in \delta^0(N)$ to denote a 0-arity production r

with left nonterminal N. We start with the first case and state the following constraint for all $d < D$:

$$\forall i_1, \ldots, i_d < K.$$
$$\mathfrak{n}^d(i_1, \ldots, i_d) = N \Rightarrow \bigvee_{r \in \delta(N)} \mathfrak{p}^d(i_1, \ldots, i_d) = r$$

The constraint for the last level D is very similar but only allows 0-arity productions.

$$\forall i_1, \ldots, i_D < K.$$
$$\mathfrak{n}^D(i_1, \ldots, i_D) = N \Rightarrow \bigvee_{r \in \delta^0(N)} \mathfrak{p}^D(i_1, \ldots, i_D) = r$$

Behavioral constraints: ensure that the generated tree is correct on all examples.

- for each IO example the value of the root node on that example should be the correct output

$$\forall \epsilon \in \mathcal{E}.\ \mathfrak{v}_0^\epsilon = \mathsf{M}(\epsilon) \tag{5.2}$$

- for each example $\epsilon \in \mathcal{E}$, the $\mathfrak{v}_d^\epsilon$ is computed consistently with $\mathfrak{p}^d$ and the values of the children. The following constraint is stated for each example $\epsilon \in \mathcal{E}$ and for each depth $d < D$:

$$\forall i_1, \ldots, i_d < K.$$
$$\bigwedge_{op(N_1,\ldots,N_j) \in \delta} \mathfrak{p}^d(i_1, \ldots, i_d) = op(N_1, \ldots, N_j) \Rightarrow$$
$$\mathfrak{v}_d^\epsilon(i_1, \ldots, i_d) = [\![op]\!](\mathfrak{v}_{d+1}^\epsilon(i_1, \ldots, i_d, 0), \ldots, \mathfrak{v}_{d+1}^\epsilon(i_1, \ldots, i_d, j))$$

- Again a little bit of care has to go for the last depth in making sure we only use 0-arity productions:

$$\forall i_1, \ldots, i_D < K.$$
$$\bigwedge_{op() \in \delta^0} \mathfrak{p}^D(i_1, \ldots, i_D) = op() \Rightarrow \mathfrak{v}_D^\epsilon(i_1, \ldots, i_D) = [\![op]\!]()$$

It is pretty easy to see that the above encoding is sound as it simply "executes" the program it generates.

Once the solver finds a satisfying assignment to all variables, we can use the values of $\mathfrak{p}^d$ to reconstruct the program (and ignore out all the other variables that are only needed to ensure the generated tree satisfies the specification). If the solver returns *unsatisfiable*, it means that there is no term within the given depth bound that satisfies all the examples.

5.3 Linear Encoding

While representing a program as a tree is very natural, sometimes it can be efficient. Consider the program `bvsub(bvsub(bvsub(bvsub(x,1),1),1),1))`. This program has depth 5 and size 9. However, a complete binary tree of depth 5 has $2^6 - 1 = 63$ nodes, which means that to find a program

of size 9, the tree encoding might have to generate a very large tree and ignore most of the nodes.[2]

Our next encoding addresses this issue by generating terms in a "linearized" form as a sequence of statements. This approach allows us bound the *size* of a term, rather than its *depth*.

Example 5.3.1 Consider Example 5.2.1 where we were building a tree like the one illustrated in Figure 5.4.

Consider, the expression `bvadd(bvadd(1,0),1)`, which in our example was the solution to the synthesis problem. Intuitively, the tree corresponding to this expression can be represented using the sequence of imperative assignments shown in Fig. 5.5. While the representation of a term as a tree is unique, the linear version has a bit more leeway in a how the term is represented. For example, in Figure 5.4, we could assign `0` to t_1 and `1` to t_0 and replace every other t_0 and t_1 with t_1 and t_0, respectively, and we would still produce the same tree.

One could also allow statements that do not contribute to the output tree similarly to how we had subtrees that were ignored in our tree example (Figure 5.6).

The above example showed the intuition behind a linear encoding of a tree. While for trees we bounded the depth of the tree, for a linear encoding we will bound the number of nodes in the encoding—i.e., intuitively the size of the tree.[3] The constraint solver maps lines to productions and the children of each line to another line appearing before.

In the rest of the section, we formalize the encoding illustrated by the above example. First, we are going to fix L to be the maximum size of the tree we are building through our linear encoding and again K be the maximum arity of a production appearing in the grammar. In Example 5.4.1, L was 3 and K was 2.

The main idea is that there are L statements that can be identified by an identifier in $[0..L-1]$.

Variables We are now ready to state what the variables appearing in our encoding are:

- one uninterpreted function $\mathtt{p}^l$, which is like $\mathtt{n}^l$, but maps lines to productions identifiers
- one uninterpreted function variable $\mathtt{n}^l : [0..D-1] \mapsto [0..|N|-1]$ that maps statement lines to nonterminal identifiers.
- one uninterpreted function $\mathtt{c} : [0..L-1] \times [0..K-1] \mapsto [0..L-1]$, which maps each line l and child id to the corresponding line.
- For each example ϵ, there is a variable $\mathtt{v}_l^\epsilon : [0..L-1] \mapsto Val$, which maps the value to which the line evaluates in the example ϵ.

The role of each variable is illustrated in Figure 5.7.

2: In this case, the program can be represented by an equivalent one of smaller depth `bvsub(bvsub(x,1),bvadd(bvadd(1,1),1))`, which has depth 4, but recall that in a synthesis problem the grammar can impose further restrictions on the syntactic structure and perhaps prevent this program. It is thus still a potential problem that the desirable solution in the program search space is high and thin rather than shallow and wide.

$$
\begin{aligned}
t_0 &= \mathtt{0} \\
t_1 &= \mathtt{1} \\
t_2 &= \mathtt{bvadd}(t_1, t_0) \\
t_3 &= \mathtt{bvadd}(t_2, t_1)
\end{aligned}
$$

Figure 5.5: One possible linear representation of the term `bvadd(bvadd(1,0),1)`. The intermediate variables t_i are used to store intermediate trees.

$$
\begin{aligned}
t_0 &= \mathtt{0} \\
t_1 &= \mathtt{1} \\
t_2 &= \mathtt{bvadd}(t_1, t_1)
\end{aligned}
$$

Figure 5.6: Representation of the tree `bvadd(1,1)` ignores the tree assigned to t_0.

3: Because nodes can be shared, as shown in the previous example, we are really bounding the number of "unique" nodes in the tree. The reason why this allows "compression" is that the same line can be reused by multiple children—i.e., the tree is really represented as a direct acyclic graph where nodes can be reused.

$line$	$\mathtt{p}^l(line)$	$\mathtt{c}(line,0)$	$\mathtt{c}(line,1)$	$\mathtt{n}^l(line)$	$\mathtt{v}^\epsilon_l(line)$	...
0	`0`	*	*	B	0000	
1	`1`	*	*	B	0001	
2	`bvadd`(B,B)	1	0	B	0001	
3	`bvadd`(B,B)	2	1	B	0010	

Figure 5.7: Illustration of what variables are mapped to each line/statement in the linear encoding. We also augment all the variables with values in green to show what values they would have for the linear encoding of the expression `bvadd(bvadd(1,0),1)` from Figure 5.5 and with ϵ = `10010101`. The values denoted with a * are when building the final tree (0-arity nodes don't have children) and can therefore be any values that satisfies the constraints.

Constraints Now that all variables are in place, we can state the constraints needed to ensure that a valid tree that correctly produces valid outputs is built:

Structural constraints:

- Line $L-1$ should be assigned the initial non-terminal:

$$\mathtt{n}^l(L-1) = S \tag{5.3}$$

- For each line, if that line is assigned a certain production, its children are appropriate non-terminals and appear at lower numbered lines.[4]

$$\begin{aligned}\forall i < L.\, \mathtt{p}^l(i) = op(N_0,\ldots,N_j) \Rightarrow \\ \mathtt{n}^l(\mathtt{c}(i,0)) = N_0 \wedge \ldots \wedge \mathtt{n}^l(\mathtt{c}(i,j)) = N_j \\ \mathtt{c}(i,0) < i \wedge \ldots \wedge \mathtt{c}(i,j) < i\end{aligned}$$

- If a line is assigned a certain non-terminal, it can only contain one of the corresponding productions:

$$\forall i < L.\, \mathtt{n}^l(i) = N \Rightarrow \bigvee_{r \in \delta(N)} \mathtt{p}^l(i) = r$$

Behavioral constraints:

- For each example, the value at line $L-1$ should be the correct output on that example:

$$\forall \epsilon \in \mathscr{E}.\, \mathtt{v}^\epsilon_l(L-1) = \mathsf{M}(\epsilon) \tag{5.4}$$

- for each example $\epsilon \in \mathscr{E}$, the value of the line $\mathtt{v}^\epsilon_f$ is computed consistently with $\mathtt{p}^l$ and the values of the children. The following constraint is stated for each example $\epsilon \in \mathscr{E}$ and for each line $i < L$:

$$\begin{aligned}\forall i < L.\, \bigwedge_{op(N_1,\ldots,N_j)\in\delta} \mathtt{p}^l(i) = op(N_1,\ldots,N_j) \Rightarrow \\ \mathtt{v}^\epsilon_l(i) = [\![op]\!](\mathtt{v}^\epsilon_l(\mathtt{c}(i,0)),\ldots,\mathtt{v}^\epsilon_l(\mathtt{c}(i,j)))\end{aligned}$$

4: To avoid cycles.

5.4 Brahma Encoding

The encodings we presented so far assign productions to a template specified a priori (either as a tree or as a sequence of statements). Because the production assigned to a node/statement is not known a priori, such

encodings also require defining behavioral constraints that effectively execute the program based on what expressions are mapped to each node/statement.

In some settings, we might be able to address this limitation if we know a priori some information on what productions will appear in the final term. The key idea of this last encoding is that if one knows a priori (an overapproximation) of what productions will appear in the linear encoding (e.g., the final program contains at most two instances of `bvadd`, one `1`, and one `x`), one can perform the mapping we performed earlier backwards—i.e., instead of assigning productions to lines, we can assign lines to the set of known productions. As a result we do not need constraints like"if this line has this production, this is the semantics" (for each production); instead we encode the semantics of each production once and for all as a single behavioral constraint. This last encoding we present was introduced in a paper by [27] and implemented in a tool called Brahma (thus the name).

Example 5.4.1 Consider again Example 5.4.1 where we were building a sequence of statements like the one illustrated in Figure 5.7 to describe the expression $\texttt{bvadd}(\texttt{bvadd}(\texttt{1},\texttt{0}),\texttt{1})$. In the linear representation, each line is mapped to a corresponding production.

In the Brahma encoding, we assume that we are given a set of production instances and we are mapping such productions to their corresponding lines. For example, let's assume that we know for a fact that the final program contains exactly most two instances of `bvadd`, one `1` and one `0` (the 0-ary expressions can be shared by multiple statements like we did in the linear encoding). More formally, we are given four production instances $\texttt{o}^0 = \texttt{0}$, $\texttt{o}^1 = \texttt{1}$, $\texttt{o}^2 = \texttt{bvadd}(\texttt{i}_1^2, \texttt{i}_2^2)$, and $\texttt{o}^3 = \texttt{bvadd}(\texttt{i}_1^3, \texttt{i}_2^3)$ and the goal of the encoding is to map each production and production child to a line number—i.e., a value in the set [0..3].

To do so, we introduce one variable $\texttt{l}^v$ for every "variable" v appearing in the encoding—i.e., the set $\{\texttt{o}^0, \texttt{o}^1, \texttt{o}^2, \texttt{o}^3, \texttt{i}_1^2, \texttt{i}_2^2, \texttt{i}_1^3, \texttt{i}_2^3\}$. The encoding then maps each line variable to a line. For example, the same sequence of expressions shown in Figure 5.5 can be obtained using the assignment in Figure 5.8.

The key idea of the encoding is that if two variables are assigned the same line they must also have the same values. Intuitively, all our "components" (productions) are given and all we are doing is "wiring" their inputs and outputs. Because we don't know what production is the "root" of the program, we also need a special variable $\texttt{o}^f$ to denote what value is returned. In this example $\texttt{l}^{\texttt{o}^f} = 3$.

For each production p, $\eta(p)$ is the maximum number of times production p can appear in the program. We use M to denote the sum of all $\mu(p)$ (for all productions). We use $p_0, \ldots, p_{M-1}$ to denote all the productions that are available to us (note that the same production can appear multiple times and these are not variables but actual production choices).

Production instances

$\texttt{o}^0 = \texttt{0}$

$\texttt{o}^1 = \texttt{1}$

$\texttt{o}^2 = \texttt{bvadd}(\texttt{i}_1^2, \texttt{i}_2^2)$

$\texttt{o}^3 = \texttt{bvadd}(\texttt{i}_1^3, \texttt{i}_2^3)$

Assignments to line variables:

$\texttt{l}^{\texttt{o}^0} = 0$ $\quad$ $\texttt{l}^{\texttt{i}_1^2} = 1$

$\texttt{l}^{\texttt{o}^1} = 1$ $\quad$ $\texttt{l}^{\texttt{i}_2^2} = 0$

$\texttt{l}^{\texttt{o}^2} = 2$ $\quad$ $\texttt{l}^{\texttt{i}_1^3} = 2$

$\texttt{l}^{\texttt{o}^3} = 3$ $\quad$ $\texttt{l}^{\texttt{i}_2^3} = 1$

$\texttt{l}^{\texttt{o}^f} = 3$

Figure 5.8: Representation of the linear sequence in Fig. 5.5 using Brahma encoding. If two variables are assigned the same lines, they must have the same values. The assignment $\texttt{l}^{\texttt{o}^f} = 3$ denotes that the value at line 3 is what the program returns.

Variables We are now ready to state what the variables appearing in our encoding are:

- For each production instance $p_i = op_i(N_1, \ldots, N_k)$ of arity k, we have one variable $\mathtt{o}^i$ to denote the value of output of evaluating this production and j variables $\mathtt{i}^i_j$ to denote the values of the inputs.[5] We use $Ovar$ to denote the set of all output variables $\{\mathtt{o}^0, \ldots, \mathtt{o}^{M-1}\}$, and $Ivar$ to denote the set of all input variables $\{\mathtt{i}^0_1, \mathtt{i}^0_2, \ldots, \mathtt{i}^{M-1}_k\}$.
- One variable $\mathtt{o}^{fin}$ for the output value of the whole program. We use O to denote the singleton set $\{\mathtt{o}^{fin}\}$.
- For each variable in $x \in Ovar \cup Ivar \cup O$, we use $\mathtt{l}^x$ to denote the line assigned to that variable. For an output variables $\mathtt{o}^i \in Ovar$, we have that $\mathtt{l}^{\mathtt{o}^i}$ denotes the line the production p_i is mapped to. For an input variables $\mathtt{i}^i_j \in Ivar$, we have that $\mathtt{l}^{\mathtt{i}^i_j}$ denotes the line the$j$-th input of production p_i is drawn from. The line of $\mathtt{o}^{fin}$ is set to the last line $M - 1$.

5: Note that productions are not really evaluated until we provide an input to the whole encoding.

Constraints Now that all variables are in place, we can state the constraints needed to ensure that a valid tree that correctly produces valid outputs is built.

Structural constraints:

- All $\mathtt{l}^i$ values are distinct and less than M

$$\forall i < j \leq M.\, \mathtt{l}^{\mathtt{o}^i} \neq \mathtt{l}^{\mathtt{o}^j}$$
$$\forall i \leq M.\, 0 \leq \mathtt{l}^{\mathtt{o}^i} < M$$
$$\forall i < M.\, \forall 1 \leq j \leq arity(p_i).\, 0 \leq \mathtt{l}^{\mathtt{i}^i_j} < M$$

- All $\mathtt{cl}^i_j$ are strictly smaller than the corresponding $\mathtt{l}^i$ (we draw inputs from earlier lines)

$$\forall i < M.\, \forall 1 \leq j \leq arity(p_i).\, \mathtt{l}^{\mathtt{i}^i_j} < \mathtt{l}^{\mathtt{o}^i}$$

- The final output variables (it will be the value of the last line)

$$\mathtt{l}^{\mathtt{o}^{fin}} = M - 1$$

- Assert that "if two lines are the same, then the values in these lines are also the same"

$$\forall x, y \in Ovar \cup Ivar \cup O.\mathtt{l}^x = \mathtt{l}^y \Rightarrow x = y$$

Behavioral constraints:

- Each components is evaluated according to its semantics:

$$\forall i < M.\, \mathtt{o}^i = [\![p_i]\!](\mathtt{i}^i_j, \ldots, \mathtt{i}^i_{arity(p_i)})$$

- For each example ϵ, the output value is correct when the variable productions are set to the corresponding values from the example:

$$(\bigwedge_{p_i = \mathtt{x}()} \mathtt{o}^i = \epsilon(x)) \Rightarrow \mathtt{o}^{fin} = \mathtt{M}(\epsilon)$$

5.5 Conclusion

We have now seen multiple encodings of the space of expressions. The main lesson to learn is not these specific encodings, but the general technique of parametrizing some program space by a finite set of first-order variables. You might want to develop your own domain-specific encodings in the future and the encodings we presented will serve as a good starting point.

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, 33).

[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).

[13] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009 (cit. on p. 23).

[14] Abhishek Udupa et al. "TRANSIT: specifying protocols with concolic snippets". In: *ACM SIGPLAN Notices* 48.6 (2013), pp. 287–296 (cit. on p. 29).

[15] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. "Recursive program synthesis". In: *International Conference on Computer Aided Verification*. Springer. 2013, pp. 934–950 (cit. on p. 29).

[16] Calvin Smith and Aws Albarghouthi. "Program Synthesis with Equivalence Reduction". In: *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*. 2019, pp. 24–47. DOI: 10.1007/978-3-030-11245-5_2 (cit. on p. 30).

[17] Manos Koukoutos, Etienne Kneuss, and Viktor Kuncak. "An Update on Deductive Synthesis and Repair in the Leon Tool". In: *Electronic Proceedings in Theoretical Computer Science* 229 (2016), pp. 100–111. DOI: 10.4204/eptcs.229.9 (cit. on p. 30).

[18] Chandrakana Nandi et al. "Rewrite Rule Inference Using Equality Saturation". In: *Proc. ACM Program. Lang.* 5.OOPSLA (2021). DOI: 10.1145/3485496 (cit. on p. 31).

[19] NICHOLAS SMALLBONE et al. "Quick specifications for the busy programmer". In: *Journal of Functional Programming* 27 (2017), e18. DOI: 10.1017/S0956796817000090 (cit. on p. 31).

[20] Woosuk Lee et al. "Accelerating search-based program synthesis using learned probabilistic models". In: *ACM SIGPLAN Notices* 53.4 (2018), pp. 436–449 (cit. on p. 31).

[21] Oleksandr Polozov and Sumit Gulwani. "FlashMeta: a framework for inductive program synthesis". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2015, pp. 107–126 (cit. on p. 33).

[22] John K Feser, Swarat Chaudhuri, and Isil Dillig. "Synthesizing data structure transformations from input-output examples". In: *ACM SIGPLAN Notices*. Vol. 50. 6. ACM. 2015, pp. 229–239 (cit. on p. 33).

[23] Yu Feng et al. "Component-based synthesis of table consolidation and transformation tasks from examples". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 2017, pp. 422–436 (cit. on p. 35).

[24] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic Superoptimization". In: *SIGPLAN Not.* 48.4 (2013), pp. 305–316. DOI: 10.1145/2499368.2451150 (cit. on p. 37).

[25] Clark Barrett and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 305–343. DOI: 10.1007/978-3-319-10575-8_11 (cit. on p. 40).

[26] Sergey Mechtaev et al. "Symbolic Execution with Existential Second-Order Constraints". In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 389–399. DOI: 10.1145/3236024.3236049 (cit. on p. 41).

[27] Sumit Gulwani et al. "Synthesis of Loop-Free Programs". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 62–73. DOI: 10.1145/1993498.1993506 (cit. on p. 47).