Program Synthesis

Program Synthesis

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

Oleksandr Polozov
University of Washington
polozov@cs.washington.edu

Rishabh Singh
Microsoft Research
risin@microsoft.com

now
the essence of knowledge
Boston — Delft

Foundations and Trends® in Programming Languages

Published, sold and distributed by:
now Publishers Inc.
PO Box 1024
Hanover, MA 02339
United States
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:
now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

The preferred citation for this publication is

S. Gulwani, O. Polozov and R. Singh. *Program Synthesis*. Foundations and Trends® in Programming Languages, vol. 4, no. 1-2, pp. 1–119, 2017.

This Foundations and Trends® issue was typeset in LaTeX using a class file designed by Neal Parikh. Printed on acid-free paper.

ISBN: 978-1-68083-292-1

Foundations and Trends® in Programming Languages

Volume 4, Issue 1-2, 2017

Editorial Board

Editorial Scope

Topics

Foundations and Trends® in Programming Languages publishes survey and tutorial articles in the following topics:

- Abstract interpretation
- Compilation and interpretation techniques
- Domain specific languages
- Formal semantics, including lambda calculi, process calculi, and process algebra
- Language paradigms
- Mechanical proof checking
- Memory management
- Partial evaluation
- Program logic
- Programming language implementation
- Programming language security
- Programming languages for concurrency
- Programming languages for parallelism
- Program synthesis
- Program transformations and optimizations
- Program verification
- Runtime techniques for programming languages
- Software model checking
- Static and dynamic program analysis
- Type theory and type systems

Information for Librarians

Foundations and Trends® in Programming Languages, 2017, Volume 4, 4 issues. ISSN paper version 2325-1107. ISSN online version 2325-1131. Also available as a combined paper and online subscription.

Foundations and Trends® in Programming Languages
Vol. 4, No. 1-2 (2017) 1–119

DOI: 10.1561/2500000010

now
the essence of knowledge

Program Synthesis

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

Oleksandr Polozov
University of Washington
polozov@cs.washington.edu

Rishabh Singh
Microsoft Research
risin@microsoft.com

Contents

Abstract

Program synthesis is the task of automatically finding a program in the underlying programming language that satisfies the user intent expressed in the form of some specification. Since the inception of AI in the 1950s, this problem has been considered the holy grail of Computer Science. Despite inherent challenges in the problem such as ambiguity of user intent and a typically enormous search space of programs, the field of program synthesis has developed many different techniques that enable program synthesis in different real-life application domains. It is now used successfully in software engineering, biological discovery, computer-aided education, end-user programming, and data cleaning. In the last decade, several applications of synthesis in the field of programming by examples have been deployed in mass-market industrial products.

This survey is a general overview of the state-of-the-art approaches to program synthesis, its applications, and subfields. We discuss the general principles common to all modern synthesis approaches such as syntactic bias, oracle-guided inductive search, and optimization techniques. We then present a literature review covering the four most common state-of-the-art techniques in program synthesis: enumerative search, constraint solving, stochastic search, and deduction-based programming by examples. We conclude with a brief list of future horizons for the field.

S. Gulwani, O. Polozov and R. Singh. *Program Synthesis*. Foundations and Trends® in Programming Languages, vol. 4, no. 1-2, pp. 1–119, 2017.

DOI: 10.1561/2500000010.

1

Introduction

1.1 Program Synthesis

Program Synthesis is the task of automatically finding programs from the underlying programming language that satisfy user intent expressed in some form of constraints. Unlike typical compilers that translate a fully specified high-level code to low-level machine representation using a syntax-directed translation, program synthesizers typically perform some form of search over the space of programs to generate a program that is consistent with a variety of constraints (e.g. input-output examples, demonstrations, natural language, partial programs, and assertions).

The problem of program synthesis has long been considered the holy grail of Computer Science. Pnueli considered program synthesis to be one of the most central problems in the theory of programming [110]. There has been a lot of progress made in this field in many different communities including programming languages, machine learning, and artificial intelligence. The idea of constructing interpretable solutions (algorithms) with proofs by composing solutions of smaller sub-problems was considered as early as in 1932 in the early work on constructive Mathematics [70]. After the development of first automated theorem

provers, there was a lot of pioneering work on deductive synthesis approaches [41, 85, 144]. The main idea behind these approaches was to use the theorem provers to first construct a proof of the user-provided specification, and then use the proof to extract the corresponding logical program. Another approach that became popular shortly afterwards was that of transformation-based synthesis [86], where a high-level complete specification was transformed repeatedly until achieving the desired low-level program.

The deductive synthesis approaches assumed a complete formal specification of the desired user intent was provided, which in many cases proved to be as complicated as writing the program itself. This lead to new inductive synthesis approaches that were based on inductive specifications such as input-output examples, demonstrations etc. Shaw et al. [125] developed a framework for learning restricted Lisp programs from a single input-output example. Summers [137] and Biermann [13] developed techniques to learn a rich class of LISP programs from multiple input-output examples. Pygmalion [131] was one of the first successful programming by demonstration systems that inferred recursive programs from a set of concrete executions of a program. There has also been a lot of pioneering work on using genetic programming approaches to automatically evolve programs that are consistent with a specification [73]. These approaches are inspired from Darwin's theory of evolution, and evolve a random population of programs continuously into new generations until generating the desired programs.

The more recent program synthesis approaches allow a user to additionally provide a skeleton (grammar) of the space of possible programs in addition to the specification [3]. This results in two benefits. First, the grammar provides structure to the hypothesis space, which can result in a more efficient search procedure. Second, the learnt programs are also more interpretable since they are derived from the grammar. The Sketch [132] system pioneered this idea to allow programmers to write partial program sketches (programs with holes), which are then automatically completed given some specification. FlashFill [43, 49] is perhaps one of the most visible Programming By Examples system that is shipping in Microsoft Excel. FlashFill defines the hypothesis space of

programs using a domain-specific language of regular expression based string transformations, and uses version-space algebra based synthesis techniques to efficiently synthesize string transformation programs from few input-output examples.

Many modern program synthesis applications are built on top of some meta-synthesis framework. Such frameworks allow a user to separately define a program space (a grammar or a program skeleton) and describe some insights for the synthesis algorithm (e.g. encoding of the synthesis problem into SAT/SMT constraints or inverse semantics of the program's operators). The framework then automatically converts these definitions into an efficient synthesizer for the given application domain. Most popular synthesis frameworks include the aforementioned SKETCH system [132], the PROSE framework for FlashFill-like programming by examples [113], and the ROSETTE virtual machine for solver-aided programming [139].

1.2 Challenges

Program synthesis is a notoriously challenging problem. Its inherent challenge lies in two main components of the problem: intractability of the program space and diversity of user intent.

Program Space In its most general formulation (for a Turing-complete programming language and an arbitrary constraint) program synthesis is undecidable, thus almost all successful synthesis approaches perform some kind of search over the program space. This search itself is a hard combinatorial problem. The number of programs in any non-trivial programming language quickly grows exponentially with program size, and this vast number of possible candidates for a long time has rendered the task intractable.

Early approaches to program synthesis focused on deductive and transformational methods [85, 86]. Such methods are based on a exponentially growing tree of theorem-proving deductive inferences or correctness-preserving code rewrite rules, respectively. Both approaches guarantee that the produced program satisfies the provided constraint

by construction but the non-deterministic nature of a theorem-proving or code-rewriting loop cannot guarantee efficiency or even termination of the synthesis process. Modern successful applications of similar techniques employ clever domain-specific heuristics for cutting down the derivation tree (see, for example, [63, 104]).

The last two decades brought a resurgence of program synthesis research with a number of technological and algorithmic breakthroughs. First, Moore's law and advances in constraint solving allowed exploring larger program spaces in reasonable time. This led to many successful constraint-based synthesis applications tracing their roots back to SKETCH and the invention of counterexample-guided inductive synthesis [132]. Second, novel approaches to program space enumeration such as stochastic techniques [105, 123] and deductive top-down search [43, 113] enabled synthesis applications in new domains that were difficult to formalize through theorems and rewrite rules.

However, even though modern-day synthesis techniques produce sizable real-life code snippets, they are still rarely applicable to industrial-size projects. For instance, at the time of this writing, the state-of-the-art superoptimization technique (i.e., synthesizer of shorter implementations of a given function; see §2.6) by Phothilimthana et al. [109] is able to explore a program space of size 10^{79}. In contrast, discovering an expert implementation of the MD5 hash function requires exploring a space of 10^{5943} programs![1] New algorithmic advances and clever exploitation of domain-specific knowledge to facilitate large program space exploration is an active research area in program synthesis.

User Intent Even armed with an efficient search technique, program synthesizers may not immediately reach the dream of automatic programming. The second challenge in synthesis is accurately expressing and interpreting user intent—the specification on the desired program.

Different methods for expressing user intent range from formal logical specifications to informal natural-language descriptions or input-output examples. Specifications on the formal end of this spectrum

[1] See Rastislav Bodik's ICFP-2015 keynote talk "Program Synthesis: Opportunities for the Next Decade" for a detailed comparison: `https://youtu.be/PI99A08Y83E`.

(traditionally required by deductive synthesis techniques) often appear to the user as complex as writing the program itself. Specifications on the informal end, on the other hand, are highly ambiguous. For instance, for a given input-output example ("`John Smith`" → "`Smith, J.`") the program space of FlashFill [43] may contains millions of programs consistent with it. Most of these programs simply overfit the example and do not satisfy the spirit of user intent. However, FlashFill has no way to discover this without additional communication from the user.

Many real-life application domains for program synthesis are too complex to be described completely with formal or informal specifications. First, such a description would likely contain so many implementation details and special cases that it would be comparable in size to the produced program. Second, and most importantly, the users themselves often do not imagine the full scope of their intent until they begin an interaction with a programmer or a program synthesis system. Both of these observations imply that applying program synthesis to larger industrial applications is much a human-computer interaction (HCI) problem as it is an algorithmic one. This survey mostly focuses on algorithmic approaches to program synthesis but we also briefly discuss some HCI-related research in §3.2, 3.3 and 7.4.

1.3 Dimensions in Program Synthesis

A synthesizer is typically characterized by three key dimensions: the kind of constraints that it accepts as expression of user intent, the space of programs over which it searches, and the search technique it employs [42]. The synthesized program may be explicitly presented to the user for debugging, re-use, or for being incorporated as part of a larger workflow. However, in some cases, the synthesized program may be implicit and is simply used to automate the intended one-off task for the user, as in case of spreadsheet string transformations [43].

1.3.1 User Intent

The user intent can be expressed in various forms including logical specification, examples [44], traces, natural language [28, 46, 79], partial

programs [132], or even related programs. A particular choice may be more suited in a given scenario depending on the underlying task as well as on the technical background of the user.

A logical specification is a logical relation between inputs and outputs of a program. It can act as a precise and succinct form of functional specification of the desired program. However, complete logical specifications are often quite tricky to write.

End users, who are not programming experts, may find providing examples as more approachable and natural. Example-based specifications more generally include asserting properties of the output (as opposed to specifying the full output) on a given input state [113]. A key challenge in this environment is that of resolving ambiguity that is inherent in the example-based specification. Such an ambiguity is often resolved in an interactive loop with the user, where the user may iteratively provide more examples dependant on the behavior of the program synthesized in the last step.

A trace is a detailed step-by-step description of how the program should behave on a given input. A trace is a more detailed description than an input-output example since it also illustrates how a specific input should be transformed into the corresponding output as opposed to just describing what the output should be. Traces are an appropriate model for programming by demonstration systems [25], where the intermediate states resulting from the user's successive actions on a user interface constitute a valid trace. From the perspective of the synthesizer, traces are preferable to input-output examples since the former contains more information. From the user's perspective, providing demonstrations in may be more taxing in general than providing input-output examples.

In some cases, a program itself might act as the best means of specifying the intent. This happens trivially for certain applications such as superoptimization [9, 109], deobfuscation [59] and synthesis of program inverses [134], where the program to be optimized, deobfuscated, or inverted respectively forms the specification. However, even for applications such as discovery of new algorithms [47], users might find it easier to write the specification as an inefficient program rather than a logical relation.

1.3.2 Search Space

The search space should strike a good balance between expressiveness and efficiency. On one hand, the space should be large/expressive enough to include a large class of programs for the underlying domain. While on the other hand, the space of the programs should be restrictive enough so that it is amenable to efficient search, and it should be over a domain of programs that are amenable to efficient reasoning.

The search space can be over imperative or functional programs (with possible restrictions on the control structure or the operator set), The program space can be restricted to a subset of an existing programming language (general purpose or domain-specific) or to a specifically designed domain-specific language. The space of programs can be qualified by at least two attributes: (i) the operators used in the program, and (ii) the control structure of the program. The control structure of the program may be restricted to a user-provided looping template [135], a partial program with holes [132], straight-line programs [8, 47, 63, 87, 109], or a guarded statement set with control flow at the very top [43].

The search space can even be over restricted models of computations such as regular or context-free grammars/transducers. Regular expression synthesis can be used for constructing text editing programs [100]. Context-free grammar synthesis is useful for paser construction [81]. Succinct logical representations may also serve as a good choice for the search space. For instance, class of first order logic together with fixed point equals the class of PTIME algorithms over ordered structures such as graphs, trees, and strings. Hence, this class and also some of its useful subclasses (such as those with a fixed quantifier depth) can serve as good target languages for synthesizing efficient graph or tree algorithms [57].

1.3.3 Search Technique

The search technique can be based on enumerative search, deduction, constraint solving, statistical techniques, or some combination of these.

Enumerative An enumerative search technique enumerates programs in the underlying search space in some order and for each program checks whether or not it satisfies the synthesis constraints. While this might appear simple, it is often a very effective strategy. A naïve implementation of enumerative search often does not scale. Many practical systems that leverage enumerative search innovate by developing various optimizations for pruning the search space or by ordering it.

Deductive The deductive top-down search [113] follows the standard divide-and-conquer technique, where the key idea is to recursively reduce the problem of synthesizing a program expression e of a certain kind and that satisfies a certain specification ϕ to simpler sub-problems (where the search is either over sub-expressions of e or over sub-specifications of ϕ), followed by appropriately combining those results. The reduction logic for reducing a synthesis problem to simpler synthesis problems depends on the nature of the involved expression e and the inductive specification ϕ. In particular, if e is of the form $F(e_1, e_2)$, the reduction logic leverages the inverse semantics of F to push constraints on e down through the grammar into constraints on e_1 and e_2.

While enumerative search is bottom-up (i.e., it enumerates smaller sub-expressions before enumerating larger expressions), the deductive search is top-down (i.e., it fixes the top-part of an expression and then searches for its sub-expressions). Enumerative search can be seen as finding a programmatic path (within an underlying grammar that connects inputs and outputs) starting from the inputs to outputs. Deduction does the same, but it searches for the programmatic path in a backward direction starting from the outputs leveraging the operator inverses. If the underlying grammar allows for a rich set of constants, the bottom-up enumerative search can get lost in simply guessing the right constants. On the other hand, the top-down deductive technique can deduce constants based on the accumulated constraints as the last step in the search process.

Constraint Solving The constraint solving based techniques [132, 135] involve two main steps: constraint generation, and constraint resolution.

Constraint generation refers to the process of generating a logical constraint whose solution will yield the intended program. Generating such a logical constraint typically requires making some assumption about the control flow of the unknown program and encoding that control flow in some manner. Three different kinds of methods have been used in the past for constraint generation: invariant-based, path-based, and input-based. On one extreme, we have invariant-based methods that generate constraints that faithfully assert that the program satisfies the given specification [133].

Such methods also end up synthesizing an inductive proof of correctness in addition to the program itself. A disadvantage of such methods is that the generated constraints may be very sophisticated since the inductive invariants are often much more complicated and over a richer logic than the program itself. On the other extreme, we have input-based methods that generate constraints that assert that the program satisfies the given specification on a certain collection of inputs [132]. Such constraints are usually much simpler in nature than the ones generated by the invariant-bases method. Unless paired with a sound counterexample guided inductive synthesis strategy (CEGIS), described in §3.2, this method trades off soundness for efficiency. A middle ground is achieved by path-based methods that generate constraints that assert that the program satisfies the given specification on all inputs that execute a certain set of paths [134]. Compared to input-based methods, these methods may achieve a faster convergence, if paired up with an outer CEGIS loop.

Constraint solving involves solving the constraints outputted by the constraint generation phase. These constraints often involve second-order unknowns and universal quantifiers. A general strategy is to first reduce the second-order unknowns to first-order unknowns and then eliminate universal quantifiers, and then solve the resulting first-order quantifier-free constraints using an off-the-shelf SAT/SMT solver. The second-order unknowns are reduced to first-order unknowns by use of templates. The universal quantifiers can be eliminated using a variety of strategies including Farkas lemma, cover algorithms, and sampling.

Statistical Various kinds of statistical techniques have been proposed including machine learning of probabilistic grammars, genetic programming, MCMC sampling, and probabilistic inference.

Machine learning techniques can be used to augment other search methodologies based on enumerative search or deduction by providing likelihood of various choices at any choice point. One such choice point is selection of a production for a non-terminal in a grammar that specifies the underlying program space. The likelihood probabilities can be function of certain cues found in the input-ouput examples provided by the user or the additionally available inputs [89]. These functions are learned in an offline phase from training data.

Genetic programming is a program synthesis method inspired by biological evolution [72]. It involves maintaining a population of individual programs, and using that to produce program variants by leveraging computational analogs of biological mutation and crossover. Mutation introduces random changes, while crossover facilitates sharing of useful pieces of code between programs being evolved. Each variant's suitability is evaluated using a user-defined fitness function, and successful variants are selected for continued evolution. The success of a genetic programming based system crucially depends on the fitness function. Genetic programming has been used to discover mutual exclusion algorithms [68] and to fix bugs in imperative programs [146]

MCMC sampling has been used to search for a desired program starting from a given candidate. The success crucially depends on defining a smooth cost metric for Boolean constraints. STOKE [124], a superoptimization tool, uses Hamming distance to measure closeness of generated bit-values to the target on a representative test input set, and rewards generation of (almost) correct values in incorrect locations.

Probabilistic inference has been used to evolve a given program by making local changes, one at a time. This relies on modeling a program as a graph of instructions and states, connected by constraint nodes. Each constraint node establishes the semantics of some instruction by relating the instruction with the state immediately before the instruction and the state immediately after the instruction [45]. Belief propagation

has been used to synthesize imperative program fragments that execute polynomial computations and list manipulations [62].

1.4 Roadmap

This survey is organized as follows. We start out by discussing some prominent applications of program synthesis in Chapter 2. We then discuss some general principles used across many synthesis techniques in Chapter 3. We then describe the four key search techniques: enumerative (Chapter 4), constraint-solving based (Chapter 5), stochastic (Chapter 6), and deduction-based programming by examples (Chapter 7). Chapter 8 concludes with some discussion on future work.

2

Applications

2.1 Data Wrangling

While the digital revolution resulted in massive digitization of human generated data, the past few years have seen an explosive growth in machine generated data because of cloud computing and IoT (Internet of Things). Data is the new oil. It is the new currency of the digital world that enables business decisions, advertising, and recommendations.

Data Wrangling refers to the process of cleaning, transforming, and preparing data from its raw semi-structured format to a more structured format that is amenable for analysis and presentation. It is estimated that data engineers/scientists spend 80% of their time in data wrangling to bring the data into a form where they can apply machine learning techniques to draw appropriate insights from. A typical data wrangling pipeline involves various kinds of activities including extraction, transformation, and formatting, which we discuss below. PBE can enable easier and faster data wrangling [44]. In case of one-off scenarios that deal with small-sized data that is easy to eye-ball, the user interaction with the system can happen simply at the level of examples and the synthesized program may be hidden from the user. However, when the user intends to execute the synthesized program multiple

times or over big data, the user may want to inspect the program to validate its correctness on unseen inputs.

2.1.1 Transformations

One of the most useful applications of program synthesis has been in the context of string or other datatype transformations. This is a task that is routinely performed by many people, and most of those people do not have the programming expertise to automate it.

Languages like Perl, Awk, Python came into existence to support efficient string/text processing, while mainstream languages like Java/C# already provide a rich support for string processing. Even spreadsheet systems like Microsoft Excel allow users to write macros using a rich inbuilt library of string and numerical functions. There are around one billion users of such spreadsheet systems. Unfortunately, 99% of those users are not proficient in programming; they find it too difficult to write desired macros or scripts. A case study of spreadsheet help forums identified that string processing is one of the most common class of programming problems that end users struggle with [43]. These users described the specification of an intended program to the experts on the other side of the help forums using examples. Since examples may lead to underspecification, the interaction between the user and the expert often involved a few rounds of communication (over multiple days). PBE is the ideal technology to automate such interactions.

Syntactic String Transformations Consider, for instance, the task of converting an email address of the form "Firstname.Lastname@domain" to "firstname lastname" as illustrated in Figure 2.1. Such syntactic transformations are useful for converting strings from one format to another, normalizing strings into a unified format, cleaning incorrectly formatted strings. The FlashFill PBE technology [43] can generate an intended program for such syntactic string transformations once the user provides a representative set of examples.

Semantic String Transformations Some string transformation tasks also require making use of some background knowledge that is encoded

	A	B
1	Email	Column 2
2	Nancy.FreeHafer@fourthcoffee.com	nancy freehafer
3	Andrew.Cencici@northwindtraders.com	andrew cencici
4	Jan.Kotas@litwareinc.com	jan kotas
5	Mariya.Sergienko@gradicdesigninstitute.com	mariya sergienko
6	Steven.Thorpe@northwindtraders.com	steven thorpe
7	Michael.Neipper@northwindtraders.com	michael neipper
8	Robert.Zare@northwindtraders.com	robert zare
9	Laura.Giussani@adventure-works.com	laura giussani
10	Anne.HL@northwindtraders.com	anne hl
11	Alexander.David@contoso.com	alexander david
12	Kim.Shane@northwindtraders.com	kim shane
13	Manish.Chopra@northwindtraders.com	manish chopra
14	Gerwald.Oberleitner@northwindtraders.com	gerwald oberleitner
15	Amr.Zaki@northwindtraders.com	amr zaki
16	Yvonne.McKay@northwindtraders.com	yvonne mckay
17	Amanda.Pinto@northwindtraders.com	amanda pinto

Figure 2.1: The FlashFill PBE technology, released in Excel 2013, can automate syntactic string transformations. Once the user provides one instance of the transformation (row 2, col. B) and proceeds to transforming another instance (row 3, col. B), FlashFill synthesizes an intended program and applies it to the remaining rows to populate col. B.

in the form of relational tables. Figure 2.2 illustrates such as task. The task requires performing a join of two (bottom) tables, then indexing the resultant table with the input columns (from the top table). The task also requires performing syntactic transformations before indexing and on the values read after indexing.

Number and Date Transformations PBE can also be useful for number transformations [126] and date transformations [129]. Consider the task of formatting numbers to two decimal places in Figure 2.3(a). A task even as simple as this would require a programmer to discover/recall the format descriptor in the underlying programming language as shown in Figure 2.3(b). In contrast, examples act as a natural means

Input v_1	**Input** v_2	**Output**
Stroller	10/12/2010	$145.67+0.30*145.67
Bib	23/12/2010	$3.56+0.45*3.56
Diapers	21/1/2011	$21.45+0.35*21.45
Wipes	2/4/2009	$5.12+0.40*5.12
Aspirator	23/2/2010	$2.56+0.30*2.56

MarkupRec

Id	**Name**	**Markup**
S30	Stroller	30%
B56	Bib	45%
D32	Diapers	35%
W98	Wipes	40%
A46	Aspirator	30%
...	...	...

CostRec

Id	**Date**	**Price**
S30	12/2010	$145.67
S30	11/2010	$142.38
B56	12/2010	$3.56
D32	1/2011	$21.45
W98	4/2009	$5.12
A46	2/2010	$2.56
...	...	...

Figure 2.2: A semantic string transformation that requires performing syntactic manipulations on multiple lookup results. The goal is to compute the `selling price` of an item (Output) from its `name` (Input v_1) and `selling date` (Input v_2) using the `MarkupRec` and `CostRec` tables. The selling price of an item is computed by adding its purchase price (for the corresponding month) to its markup charges, which in turn is calculated by multiplying the markup percentage by the purchase price. Such transformations can be inferred by examples [127].

of describing intent in such cases, and PBE can be used to automate such tasks. The PBE paradigm also naturally extends to enabling more sophisticated transformations of the kind shown in Figure 2.3(c) and Figure 2.3(d).

Splitting Transformations Splitting involves extracting the various sub-fields from a long string. For instance, consider the input column and the desired output table in Figure 2.4. Performing the desired splitting is challenging since there are multiple delimiters and not all occurrences of a delimiter string are actual delimiters. Again, examples can serve as a natural means for expressing intent in such cases.

While the above-mentioned transformations convert a tuple of strings into another string, the splitting transformation converts a string into

Input	Output
123.4567	123.46
123.4	123.40
78.234	78.23

(a)

Language	Format descriptor for rounding to two decimal places
Excel, C#	#.00
Python, C	.2f
Java	#.##

(b)

Input	Output
0d 5h 26m	5:00
0d 4h 57m	4:30
0d 4h 27m	4:00
0d 3h 57m	3:30

(c)

Input	Output
08/21/2010	08/21/2010
07/24/2010	07/24/2010
20.08.2010	08/20/2010
23.08.2010	08/23/2010
2010-06-07	07/06/2010
2010-24-08	08/24/2010

(d)

Figure 2.3: Sample number and date transformations that can be automated using PBE: (a) Rounding to two decimal places, (c) Nearest lower half hour, (d) Formatting dates to a consistent format. (b) shows the format descriptors in different programming languages required to perform the rounding transformation in (a).

a tuple of strings. Though splitting can be seen as multiple instances of a string transformation, this view does not leverage the specific cues about the splitting intent. Leveraging that domain knowledge, in fact, enables a predictive synthesis technique for splitting [116], wherein the intended splitting can be performed without any user specification but from just the input data.

2.1.2 Extraction

Data is locked up into documents of various types such as text/log files, semi-structured spreadsheets, webpages, JSON, XML, and pdf documents. These documents offer great flexibility for storing and

```
191.128.19.55 - - [09/Jun/2016:18:05:33 -0800] "GET /checks.txt
174.13.04.3 - - [09/Jun/2016:19:43:23 -0800] "GET /images/pic.png
192.16.201.109 - - [10/Jun/2016:06:10:03 -0800] "GET /pdf/document.pdf
11.0.4.50 - - [10/Jun/2016:16:10:02 -0800] "GET /index.html
191.169.12.13 - - [11/Jun/2016:11:10:02 -0800] "GET /index.html
172.18.0.102 - - [11/Jun/2016:16:12:34 -0800] "GET /logs/access.log
192.19.2.100 - - [11/Jun/2016:17:32:36 -0800] "GET /index.html
10.129.2.78 - - [11/Jun/2016:17:45:38 -0800] "GET /data/2/4
171.19.3.12 - - [11/Jun/2016:22:12:01 -0800] "GET /data/
191.168.125.112 - - [11/Jun/2016:23:12:52 -0800] "GET /pictures/pic2.png
174.26.0.223 - - [12/Jun/2016:16:13:04 -0800] "GET /images/pic4.gif
175.16.0.24 - - [12/Jun/2016:17:29:33 -0800] "GET /index.html
196.168.29.105 - - [12/Jun/2016:18:33:11 -0800] "GET /styles.css
101.22.54.38 - - [13/Jun/2016:20:32:43 -0800] "GET /js/scripts.js
```

(a)

191.128.19.55	-	-	[09/Jun/2016	: 18:05:33	- 0800	] "	GET	/	checks.txt
174.13.04.3	-	-	[09/Jun/2016	: 19:43:23	- 0800	] "	GET	/	images/pic.png
192.16.201.109	-	-	[10/Jun/2016	: 06:10:03	- 0800	] "	GET	/	pdf/document.pdf
11.0.4.50	-	-	[10/Jun/2016	: 16:10:02	- 0800	] "	GET	/	index.html
191.169.12.13	-	-	[11/Jun/2016	: 11:10:02	- 0800	] "	GET	/	index.html
172.18.0.102	-	-	[11/Jun/2016	: 16:12:34	- 0800	] "	GET	/	logs/access.log
192.19.2.100	-	-	[11/Jun/2016	: 17:32:36	- 0800	] "	GET	/	index.html
10.129.2.78	-	-	[11/Jun/2016	: 17:45:38	- 0800	] "	GET	/	data/2/4
171.19.3.12	-	-	[11/Jun/2016	: 22:12:01	- 0800	] "	GET	/	data/
191.168.125.112	-	-	[11/Jun/2016	: 23:12:52	- 0800	] "	GET	/	pictures/pic2.png
174.26.0.223	-	-	[12/Jun/2016	: 16:13:04	- 0800	] "	GET	/	images/pic4.gif
175.16.0.24	-	-	[12/Jun/2016	: 17:29:33	- 0800	] "	GET	/	index.html
196.168.29.105	-	-	[12/Jun/2016	: 18:33:11	- 0800	] "	GET	/	styles.css
101.22.54.38	-	-	[13/Jun/2016	: 20:32:43	- 0800	] "	GET	/	js/scripts.js

(b)

Figure 2.4: PBE can be used for column splitting: (a) input data column, (b) output columns.

counter example guided ??
e.g. user provides cex to help synthesis

	A	B	C	D	E
1		value	year	value	year
2	Albania	1000	1950	930	1981
3	Austria	3139	1951	3177	1955
4	Belgium	541	1947	601	1950
5	Bulgaria	2964	1947	1959	1958
6	Czech ...	2416	1950	2503	1960

...

(a)

Country	Harvest	Date
Albania	1000	1950
Albania	930	1981
...		
Austria	3139	1951
Austria	3177	1955
...		
Belgium	541	1947
Belgium	601	1950

...

(b)

Figure 2.5: FlashRelate can transform the semi-structured table (a) into the output structured table (b) once the user provides a couple of examples of tuples in the output table, for instance, the ones highlighted in orange and green, respectively.

organizing hierarchical data by combining presentation/formatting with the underlying data. However, this flexibility makes it hard to perform tasks such as querying or transforming data in content or to another storage format. The ability to extract data out of such documents into a structured tabular form can facilitate such desired processing using appropriate data-processing tools.

The FlashRelate PBE technology allows extracting tabular/relational data from semi-structured spreadsheets. The two-dimensional grid structure of a spreadsheet leads to creative solutions for storing higher-dimensional data through use of spatial layouts involving headers, whitespace, and relative positioning. While this leads to compact and intuitive visual representations of data well suited for human understanding, such encodings complicate the use of powerful data-manipulation tools (e.g., relational query engines) that expect data in a certain form. We call these spreadsheets *semi-structured* because their data is in a regular format that is nonetheless inaccessible to data-processing tools. It is conjectured that around 50% of spreadsheets contain data in this not-easy-to-use format. FlashRelate allows a user to provide examples of tuples in the output table—it then synthesizes a program to extract more such tuples from the input semi-structured spreadsheet. Figure 2.5 illustrates the capability of FlashRelate.

The FlashExtract PBE technology allows extracting structured (tabular or hierarchical) data from semi-structured text/log files and webpages. For each field in the output data schema, the user provides positive/negative instances of that field and FlashExtract generates a program to extract all instances of that field. Figure 2.6 illustrates the capability of FlashExtract. The FlashExtract technology has been shipped in two Microsoft products: (i) ConvertFrom-String cmdlet in Powershell in Windows 10, wherein the user provides examples of the strings to be extracted by inserting tags around them in text. Several Microsoft MVPs (Most Valued Professionals) have built UI experiences on top of this cmdlet. (ii) Custom Fields feature in Microsoft Operations Management Suite—a SAAS service for IT professionals to collect machine data from any cloud and get operational insights. This feature allows extracting custom fields from log files by providing examples.

2.1.3 Layout

Users sometimes want to perform layout transformations of data organized in tables or tree-shaped structures like XML/JSON. Such transformations do not change the textual content of any data element, but they rearrange the layout of the data, i.e. the manner in which the various data elements are spatially/logically arranged or grouped.

For instance, consider the directory structure of music files shown in Figure 2.8(a). The user may want to categorize her music based on file type while also maintaining the original organization based on genre as shown in Figure 2.8(b). PBE techniques can help automate such transformations, thereby obviating the need to write bash scripts or XSLT expressions for transforming such hierarchically structured data [149].

Another interesting class of layout transformations is that of reformatting semi-structured tables in spreadsheets. This is challenging because of their special layout or formatting attributes such as subheaders, footers, and filter cells (blank cells or cells with some special characters to aid visual readability of table content). For instance, consider the reformatting task illustrated in Figure 2.7. Examples can again

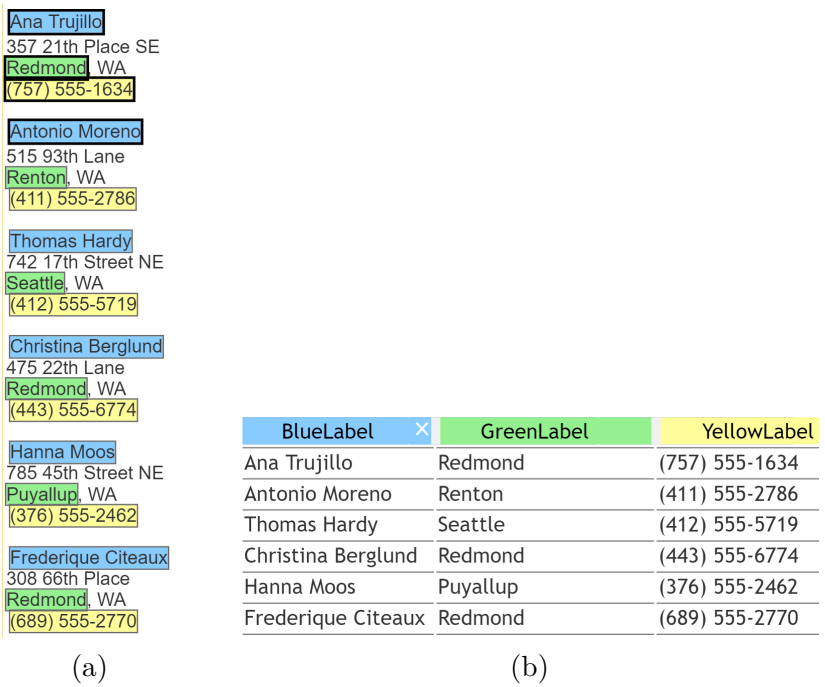


Figure 2.6: FlashExtract enables tabular data extraction from text/log files and web pages using examples. Once the user highlights one or two examples of each field in a different color (in the text file on the left side), FlashExtract extracts more such instances and arranges them in a structured data format (table on the right side).

act as a natural means of describing intent in such cases and PBE techniques can help automate such transformations [52].

2.2 Graphics

Synthesizing programs for constructing graphical objects has many applications.

Constructive programmatic descriptions of graphical objects can enable dynamic geometry that can lead to quick recomputation of coordinates of various dependant points after a new value is assigned to a free variable. This can enable interactive editing experiences and efficient animations. Program synthesis techniques have been applied to

PROJ	CAT	SPONSOR	DEPT	ELTS	DUE
SPEC	OOH	Infiniti	Design	elt 1	11/10
SPEC	OOH	Infiniti	Desing	elt 2	
SPEC	Print		Design	elt 3	11/30
SPEC	Print		Design	elt 4	11/30
SPEC	Print	Infiniti	Design	elt 5	11/30

(a)

		SPEC	
		OOH	
Infiniti	Design	elt 1	11/10
Infiniti	Desing	elt 2	
		Print	
	Design	elt 3	11/30
	Design	elt 4	11/30
Infiniti	Design	elt 5	11/30

(b)

Figure 2.7: PBE can help automate reformatting transformations over semi-structured spreadsheet tables: (a) Example input table, (b) Example output table.

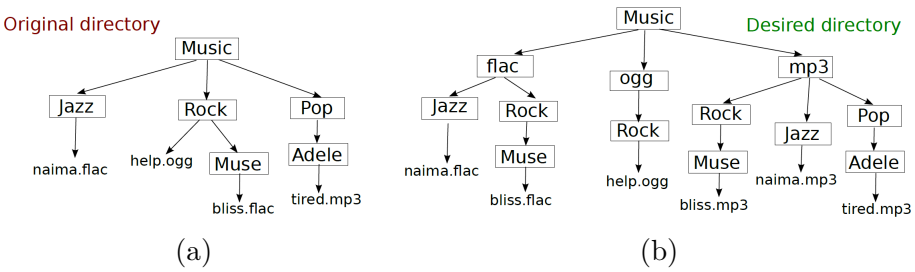


Figure 2.8: PBE can help automate transformations on tree-structured data like directory structure [149].

Figure 2.9: PBE can be used for constructing structured repetitive drawings from a few examples of the repetitive drawing elements: (a) Lines moving on a circle pattern, (b) Linear pattern, (c) Polylines moving on circle pattern.

generate solutions to geometry construction problems of the kind found in high-school curriculum [48].

Images and drawings sometimes contain structured repetition as in brick patterns, tiling patterns, and architectural drawings. Figure 2.9 illustrates some such drawings. Constructing such drawings requires the user to write a script using CAD APIs or perform tedious copy-paste operations with some underlying mathematical logic. PBE can be to used to construct such structured drawings, wherein the user provides a few examples of repetitive drawing elements, and the underlying system predicts the next elements in the sequence [21].

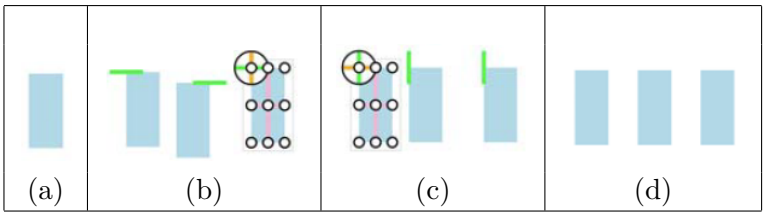
PBE can bring together the complementary strengths of direct manipulation and programmatic manipulation. While direct manipulation enables easy (but simple) manipulation of a concrete object, programmatic manipulation allows for much more freedom and reusability (but requires skill). Gulwani et.al. have proposed an approach that enables construction of complex shell scripts in steps, where the user provides examples for each step [50]. Chugh et.al. have proposed an approach called *prodirect manipulation* that enables creation and modification of programs using GUI-based manipulation for the domain of scalable vector graphics [23, 53]. The user draws shapes, relates their attributes, groups and edits them using the GUI, and the drawing is kept synchronized with an underlying program. The various GUI-based actions translate to constraints over example drawing. Constraint solvers are used to generate candidate modifications to the underlying program so that the resultant program execution generates a drawing satisfying

those constraints. Smart heuristics are used to select an intended modification from among the many solutions. A skilled user can edit the resulting program during any step to refine the automatically generated modification or to implement some new functionality.

Example 2.1. Consider the task of drawing three equi-distant rectangles as shown in Figure 2.10(d) using the SKETCH-N-SKETCH tool [53] that implements prodirect manipulation. The user starts out by drawing a single rectangle using the GUI (Figure 2.10(a)), which results in the code shown in Figure 2.10(e). The user then makes two copies of the rectangle and drags them, roughly aligning them in the vertical direction and roughly spacing them evenly in the horizontal direction. The user then invokes the Merge selection over the three rectangles to indicate a relationship and the tool abstracts the underlying representation to the code shown in Figure 2.10(f). The user then performs the MakeEqual selection over the y-positions of the boxes (Figure 2.10(b)), followed by DigHole selection over the x-positions of the boxes (Figure 2.10(c)), resulting in the refactored code shown in Figure 2.10(g). The user then manually text-edits the hole to redefine `rect2x` to bind the result of the expression `(+rect1x (/ (-rect3x rect1x) 2))` to encode the desired spacing (since there isn't a built-in GUI selection for "distributing" the shapes), resulting in the intended drawing in Figure 2.10(d).

2.3 Code Repair

There have been many synthesis techniques recently developed for the code repair problem [26, 60, 99, 130]. Given a buggy program P and a specification ϕ, the code repair problem requires to compute modifications to the buggy program to obtain a new program P' such that P' now satisfies the specification ϕ. The general idea of these techniques is to first insert alternate choices for the expressions present in the buggy program and then use synthesis techniques to find replacements or modifications of the expressions from the inserted choices such that the updated program satisfies the given specification. Some example code repairs generated by the SemFix [99] and AutoProf [130] systems are shown in Figure 2.11 and Figure 2.12 respectively.




```
(def rect1 (let [x y w h color] [50 60 40 90 'blue']
     (rect color x y w h)))
(def main (svg [rect1]))
```

(e)

```
(def customRect (λ (x y) (let [w h color] [40 90 'blue']
              (rect color x y w h))))
(def main (svg[ (customRect 50 60) (customRect 110 73)
                (customRect 200 55) ]))
```

(f)

```
(def y 60); Variable to equate values
; Variables and Hole to relate Values
(def [rect1x rect2x rect3x] [50 110 200])
(def main (svg[ (customRect rect1x y) (customRect rect2x y)
                (customRect rect3x y) ]))
```

(g)

Figure 2.10: Prodirect Manipulation using SKETCH-N-SKETCH [53]. The user arrives at the diagram in (d) by going through the intermediate stages in (a), (b), and (c). Correspondingly, the underlying code evolves as shown in (e), (f), and, (g).

Test set

Input			Output	
inb	usep	dsep	expected	actual
1	0	100	0	0
1	11	110	1	0
0	100	50	1	1
1	-20	60	1	0
0	0	10	0	0

```
1  int buggy(int inb, int usep, int dsep) {
2    int bias;
3    if (inhb)
4      bias = dsep; //fix: bias = usep+100
5    else
6      bias = usep;
7    if (bias > dsep)
8      return 1;
9    else
10     return 0;
11 }
```

Figure 2.11: An example code repair synthesized by the SemFix [99] on a Tcas benchmark using the set of passing and failing test cases.

Given a buggy program P and a test suite T consisting of both passing and failing test cases, SemFix first uses statistical fault localization techniques to generate a list of program statements ranked by their suspiciousness of being the cause of the bug. It then iteratively tries to repair statements in the order of their suspiciousness until a successful repair is generated. For a given statement, it replaces the statement with an unknown function over the live variables, and uses component-based synthesis to compute the unknown function such that it satisfies the constraints obtained from the testcases.

AutoProf uses constraint-based synthesis to repair student submissions. Unlike SemFix, AutoProf uses the reference implementation as the specification. It first uses an error model (corresponding to common mistakes that students typically make on a given assignment) to

```
1  def computeDeriv(pol):
2    deriv = []
3    zero = 0
4    if (len(pol)==1):
5      return deriv
6    for e in range(0, len(pol)):
7      if (pol[e]==0):
8        zero += 1
9      else:
10       deriv.append(pol[e]*e)
11   return deriv
```

Generated Feedback (Repair)

The program requires **3** changes:

- In the return statement **return deriv** in **line 5**, replace **deriv** by [**0**].
- In the comparison expression **(pol[e] == 0)** in **line 7**, change **(pol[e] == 0)** to **False**.
- In the expression **range(0, len(pol))** in **line 6**, change **0** to **1**.

Figure 2.12: An example repair generated by the AutoProf [130] system on a student submission to an introductory programming course on the edX platform.

define the space of modifications to a student program. It then uses the SKETCH solver to find minimum number of changes (from the space of modifications) to the student submission such that it becomes functionally equivalent to the reference implementation. Similar synthesis techniques have also been developed for auto-grading assignments for an Embedded systems laboratory course [65] and was recently deployed for a MOOC course. Recently, a technique has been proposed to also take into account semantic distance in addition to syntactic distance to compute better repairs for student submissions [26].

There are also some game-based repair techniques [60]. The game consists of the product of a modified version of the program and an automaton representing the LTL specification of the program. The program game is an LTL game that captures the possible repairs of the program by making some values in the program unknown. The game is played between the environment that provides the inputs and the system that provides the correct value for the unknown expression. The game is won if for any input, the system can provide a value for the unknowns such that the specification is satisfied. The repair technique computes a memoryless winning strategy for the game, which corresponds to a repair for the buggy program.

2.4 Code Suggestions

Finding code suggestions or completing code snippets is a direct application of program synthesis to software engineering, in the spirit of the original vision of the field. Most modern code editors and IDEs include an "autocompletion" capability, which predicts the next likely token in the program based on the previously typed ones (*e.g.*, IntelliSense in Microsoft Visual Studio or Content Assist in Eclipse). Program synthesis has the potential to enhance this capability by automatically completing whole snippets of code instead of single tokens. In addition, it can take into account optional annotations about the desired snippet properties from the programmer such as expression types or keywords in the identifiers.

Automatic code completion is particularly handy in the presence of unfamiliar APIs. Modern software libraries/framework include tens of thousands of members, which are difficult to navigate for an outsider. A programmer often needs to write a multi-line snippet composing multiple unconnected API calls in order to accomplish a task. Researchers in program synthesis have proposed several approaches to resolve this issue, most notable being *statistical models* [114], *type-directed completion* [106], and tools like InSynth [51] and Bing Developer Assistant [151].

Statistical techniques, employed by Raychev et al. [114] and Zhang et al. [151], use probabilistic models to solve the synthesis problem in whole or in part. They leverage the vast dataset of Web code snippets from GitHub, StackOverflow, and other programmer networks (colloquially known as "Big Code") to build a model of API call sequences [114] or to find a snippet matching a given natural language query [151]. In [114], API call sequences are treated similarly to textual sentences, and modelled using N-gram or RNN-based language models. In [151], a user-provided query like *"how to save an image"* is submitted to a search engine, and then the discovered code snippets are transformed to fit the user's code context (e.g. input/output variable names). Both projects share some common important characteristics:

- They require a logic-based component to analyze data flow and control flow of the synthesized snippets, since treating programs purely as text is likely to produce nonsensical output.
- They rely on some high-quality ranking function to present a list of suggestions to the user. This ranking function commonly involves a mix of statistically induced similarity scores based on the user's query and dataset, and some heuristically chosen weights for precise tailoring.

Both approaches are highly successful. Raychev et al. predict the desired completion among top 3 suggestions in more than 90% of cases. Bing Developer Assistant was released as a Visual Studio extension with more than 600K downloads and mostly positive reviews. According to Zhang et al., it saved 28% of the time for the programmers in their experimental study.

Gvero et al. [51] and Perelman et al. [106] use *typing information* as a primary driver behind their completion approaches. Both projects solve the problem of synthesizing a snippet of API calls from a partial program (optionally annotated with keywords or type information). They solve it using enumerative search (forward in [106] and backward in [51]), augmented with a *ranking function* and *space abstractions*.

The key part of both algorithms is some form of space abstraction. It leverages type information to reformulate the synthesis problem in a more concise and abstract program space. The particular abstraction techniques vary: Gvero et al. introduce an abstract lambda calculus based on *succinct types* (i.e. program types grouped by equivalence modulo products and currying), and Perelman et al. infer *abstract types* such as "path" or "font family name" based on usage patterns. This abstraction makes the enumerative search tractable, bringing its runtime to a fraction of a second for a typical query.

2.5 Modeling

Probabilistic modeling is a technique for analysis of complex dependent systems based on empirical evidence. The most widespread tools for probabilistic modeling are various forms of machine learning: Markov

models, neural networks, Bayesian networks, etc. Typically, a probabilistic model involves two components: a *structure* of the system (i.e. a description of dependencies between system components) and its *parameters* (i.e. particular statistical distributions for all component). Most forms of machine learning focus on inferring parameters of the model, assuming a given structure such as linear regression. Program synthesis has been applied to learning the model structure in a form of a *probabilistic program* [101].

Figure 2.13 shows an example of a *sketch* for a probabilistic program for modeling player skills in an online game, together with associated empirical evidence: some sample outcomes of such games. In this sketch, the assumed distributions for the system components (that is, player skills and their performances in individual games) are replaced with *holes*. The programmer also annotates holes with hypothesized dependencies between the components. The PSketch system [101] takes this sketch and a dataset as an input, and completes the sketch by synthesizing probabilistic expressions for the holes that match the empirical evidence. The synthesis problem includes both the model structure (that is, which distributions should be chosen for the components) and its parameters.

2.6 Superoptimization

Superoptimization is the task of synthesizing an optimal sequence of instructions (over some target architecture) that is functionally equivalent to a given piece of code [87].

Superoptimization is used to generate optimized machine code. The traditional application is to optimize straight-line code fragments [9, 109]. However, it has also been applied to optimize loop fragments in the context of auto-vectorization [12]. Superoptimization has also been used to synthesize bitvector programs, which combine arithmetic and bitwise operations. Bitvector programs can be quite unintuitive and extremely difficult for average, or sometimes even expert, programmers to discover methodically. Consider, for example, the problem of computing the average of two numbers x and y without using conditionals. Note that $(x + y)/2$ is not correct since the computation can overflow. Instead

Sketch:

```
1   struct game {
2       int id;
3       int p1;
4       int p2;
5       int result;
6   }

8   double[] TSSketch(struct game[] games, int count) {
9       double[] skills;
10      int[] r;

12      for i=0 to count-1
13          skills[i] := ??;
14      foreach g in games
15          r[g.id] = ??(skills[g.p1], skills[g.p2]);
16      foreach g in games
17          observe(g.result == r[g.id]);
18      return skills;
19  }
```

Data:

m.p1	m.p2	result
0	1	1
1	2	1
0	2	1

id	skill
0	105
1	95
2	90

Figure 2.13: A sketch of a probabilistic program for a TrueSkill model [55], and associated evidence data (re-used from [101]). The holes `??` should be replaced with synthesized probabilistic expressions so that the entire program best matches the data. For this example, the first hole is filled in with `Gaussian(100, 10)` and the second one with `Gaussian(skills[g.p1], 15) > Gaussian(skills[g.p2], 15)`.

this can be achieved using the following composition of bitwise and arithmetic operators: $(x|y) - ((x \oplus y) >> 1)$.

One approach to superoptimization has been to simply enumerate sequences of increasing length or cost, testing each for equality with the target specification [87]. Another approach has been to constrain the search space to a set of equality-preserving transformations expressed by the system designer [63] and then select the one with the lowest cost. Yet another approach has been to reduce the code sequence search problem to that of SAT/SMT constraint solving, thereby allowing use of powerful off-the-shelf constraint solvers [47].

Sometimes superoptimization can be too expensive to be performed during a compilation phase. One scalability paradigm has been to generate general purpose peephole optimizers in an offline phase—by automatically discovering replacement rules mapping original sequences to their optimized counterparts and organizing them into a lookup table [8]. Another paradigm has been to develop new optimization techniques. The LENS algorithm [109] does this elegantly in the context of enumerative search methods—it leverages an elegant memoization strategy, wherein it computes programs of bounded size that satisfy a given set of examples and incrementally refines these programs with more examples in the next iteration. The algorithm also leverages a powerful meet-in-the-middle pruning technique based on bidirectional search, where the candidate programs are enumerated forward from input states and also backward from output states.

2.7 Concurrent Programming

Writing concurrent programs with efficient synchronization is a challenging and error-prone task even for proficient programmers. There have been several synthesis techniques designed to help programmers write such complex code, e.g. by automatically synthesizing placement of minimal synchronization constructs in a concurrent program [17, 143] or by inferring placement of memory fences in concurrent programs running on relaxed memory models [75]. The Abstraction-Guided Synthesis (AGS) [143] technique models the problem of efficient synchronization

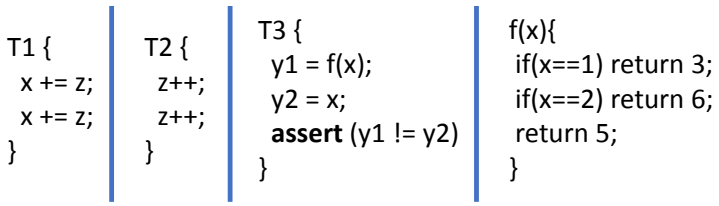


Figure 2.14: A simple example program executing three processes T1, T2, and T3 in parallel taken from [143].

inference in a given concurrent program as one in which both the abstraction of the program (using abstract interpretation) and the program itself are allowed to be modified until the abstraction becomes precise enough to verify the program.

Consider a simple example of a program executing three processes $T_1|T_2|T_3$ in parallel in Figure 2.14 taken from [143]. The variables $y1$ and $y2$ in T_3 can take different values depending upon the different interleavings of the statements in the processes. For example, the interleaving `z++; x+=z; x+=z; y1=f(x); y2=x; z++; assert` results in the values `{y1=6, y2=2}`, whereas the interleaving `x+=z; x+=z; y1=f(x); y2=x; z++; z++; assert` results in `{y1=5, y2=0}`. The interleaving `z++; x+=z; y1=f(x); z++; x+=z; y2=x;assert` results in the assertion violation with `{y1=3, y2=3}`. The goal of the synthesis process is to place minimal synchronization such that the safety requirements are met, i.e. the assert statement in T_3 is satisfied.

The AGS algorithm first performs an abstract interpretation of the program using an abstract domain and checks if there is any abstract interleaving that results in violation of the safety constraint. If none, it returns the current program. Otherwise, there exists a counter-example interleaving that violates the assertion and AGS has two choices: either refine the abstraction (parity to interval, interval to octagon etc.) or modify the program by adding synchronization such that this interleaving is disallowed. The algorithm non-deterministically chooses one (may need to backtrack sometimes) and continues this process until finding a program that can be verified given the abstraction. For the simple example, a sample run of the algorithm is shown in Figure 2.15.

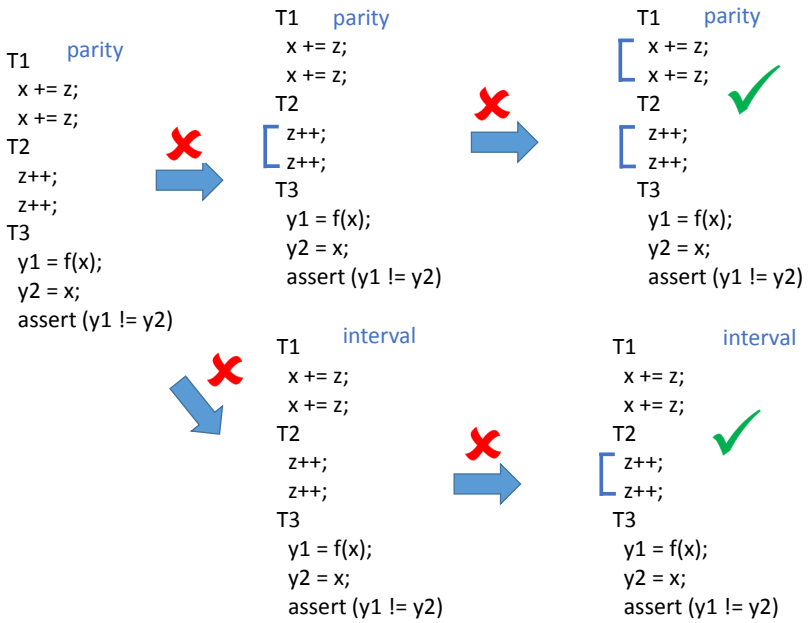


Figure 2.15: A sample run of the AGS algorithm on the example problem in Figure 2.14.

3

General Principles

3.1 Second-Order Problem Reduction

Program synthesis is a second-order search problem, where the goal is to discover a function that satisfies a given specification. Clearly, this is an undecidable problem in general. If we can reduce this second-order search problem to a first-order search problem, we can leverage off-the-shelf constraint solvers that can deal with first-order constraints. One approach to doing that is via use of templates, which are hints about the syntactic structure of the artifact to be discovered [135]. The synthesis task thus reduces to searching for the missing components in the template. The templates act as an expressive yet accessible mechanism for the programmer to express their insight since the programmer's task is now limited to simply specifying the high-level structure without worrying about the low level details.

As an example, consider the task of drawing a pixelated line, approximating a real line, between coordinates points $(0, 0)$ and (X, Y), say in the top-right quadrant. The desired program is expected to generate (x, y) for the pixelated line such that the output values should not deviate more than half a pixel away from the real value, i.e., we need $|y - \frac{Y}{X}x| \leq \frac{1}{2}$ for any (x, y). This property is captured formally using

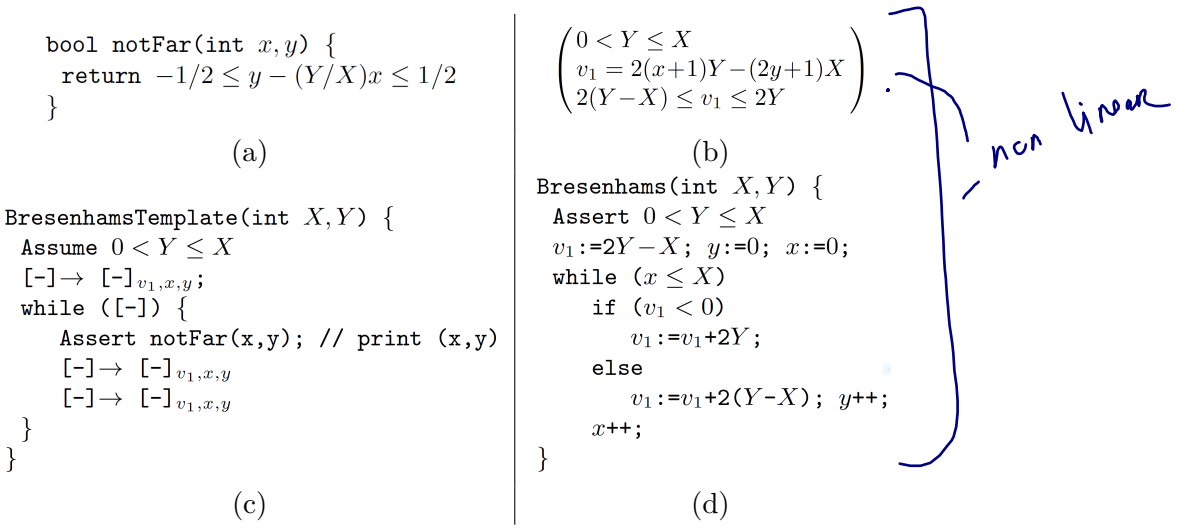


Figure 3.1: (b) shows the inductive loop invariant to establish correctness of Bresenhams code in (d). Both (b) and (d) can be synthesized from the templatized Bresenhams code in (c) and the specification in (a).

the `notFar` predicate in Figure 3.1(b). The programmer may specify the program template shown in Figure 3.1(c) the involves a loop with an unknown loop body and some unknown initialization.

While it is obvious that the statements will involve updates to x and y, the insight for using an auxiliary variable v_1 may not immediately occur to the programmer. The programmer may add that in a template refinment step after the synthesizer fails to return a solution. It turns out that this extra variable v_1 is used by the program as a measure of the error variable between the discrete line and the real line as shown in the synthesized program in Figure 3.1(d).

The template-based program synthesis approach builds on top of the template-based program verification approach [133]. The latter attempts to find a inductive proof of correctness, given a program and the pre/postcondition pair. For instance, such techniques can automatically synthesize the inductive invariant in Figure 3.1(b), given the program in Figure 3.1(d) and an appropriate template for the inductive invariant. It turns out that this verification methodology extends to the more general

synthesis setting where the program statements are also abstracted away via templates as shown in Figure 3.1(c).

Templates are quite similar to sketches in the Sketch system [132], discussed in §3.3.1. However, there are two key differences. Sketch synthesizes integers that are bounded for each hole, while program templates contain holes that are arbitrary expressions and predicates, and therefore the synthesizer needs to generate values that are potentially unbounded. Secondly, Sketch only has the notion of partial programs and ensures only bounded verification, while templates fit well with an underlying template-based full verification methodology that allows specification of partial invariants as well.

3.2 Oracle-Guided Synthesis

Templates work well for reducing complexity of program synthesis, when they are straightforward to compose. In most cases, however, relevant domain-specific templates are unavailable. For such situations, the community has adopted an alternative simple approach to program synthesis. It is based on the following observation: while *synthesizing* a program that satisfies a given property is a second-order problem that may be infeasible, *verifying* if a given program satisfies that property is a first-order problem, and typically more straightforward [132].

Consider the task of finding a maximum number m in a list ℓ. Formally, synthesizing a program $P_{\max}$ that solves this task can be described by the following specification:

$$\exists P_{\max}\, \forall \ell, m\colon P_{\max}(\ell) = m \implies (m \in \ell) \wedge (\forall x \in \ell\colon m \geq x) \tag{3.1}$$

This is a second-order formula. Suppose now that we have a candidate program $P_{\max}$. Validating that $P_{\max}$ does, in fact, solve the task is equivalent to validating the following formula:

$$\forall \ell, m\colon P_{\max}(\ell) = m \implies (m \in \ell) \wedge (\forall x \in \ell\colon m \geq x) \tag{3.2}$$

Instead of proving Equation 3.2 we can disprove its negation:

$$\exists\, \ell, m\colon (P_{\max}(\ell) = m) \wedge (m \notin \ell \vee \exists x \in \ell\colon m < x) \tag{3.3}$$

Equation 3.3 is a first-order formula.[1] If it is unsatisfiable, then $P_{\max}$ is a valid solution to the original problem. If it is satisfiable, then a specific binding of ℓ and m constitutes a *counterexample*. If we take this counterexample (ℓ, m) into account during our subsequent search, then our new candidate programs must not return m on an input ℓ – thus we have strengthened the specification, eliminating the previous incorrect candidate $P_{\max}$.

Furthermore, we can reformulate our verification procedure to produce a *constructive* counterexample – an input ℓ together with the corresponding correct output m^*.

$$\exists\, \ell, m^* \colon (P_{\max}(\ell) \neq m^*) \wedge (m^* \in \ell) \wedge (\forall x \in \ell \colon m^* \geq x) \tag{3.4}$$

If Equation 3.4 is satisfiable, it produces a input-output pair (ℓ, m^*) that our desired program *must* satisfy. In contrast, Equation 3.3 produces an input-output pair (ℓ, m) that the desired program *must not* satisfy, which is a much weaker constraint. Note that such reformulation is only possible when only one correct output can satisfy the specification.

Observation above leads us to the architecture called *counterexample-guided inductive synthesis (CEGIS)*, shown in Figure 3.2. In CEGIS, we split the synthesis problem into *search* and *verification*. The first component (often called the *solver*) starts with a simple first-order specification, which is a simplified version of the original second-order requirement on the desired program. It produces a *program candidate* that satisfies this simplified specification. The second component (often called the *verifier*) validates that this program candidate satisfies the original specification. If it does not, the verifier produces a *counterexample*, which is returned back to the solver. The solver adds it to the specification and repeats its search, looking for a new program candidate that, in addition, satisfies this counterexample. This process repeats in iterations until either **(a)** the verifier accepts some program candidate – that is, we have found a program that satisfies the entire second-order specification, or **(b)** the solver cannot find a candidate that is consistent with its current specification – that is, the original problem is unsolvable.

[1] Apparently nested existential quantifiers can in fact be flattened into one because ℓ is a finite list of numbers.

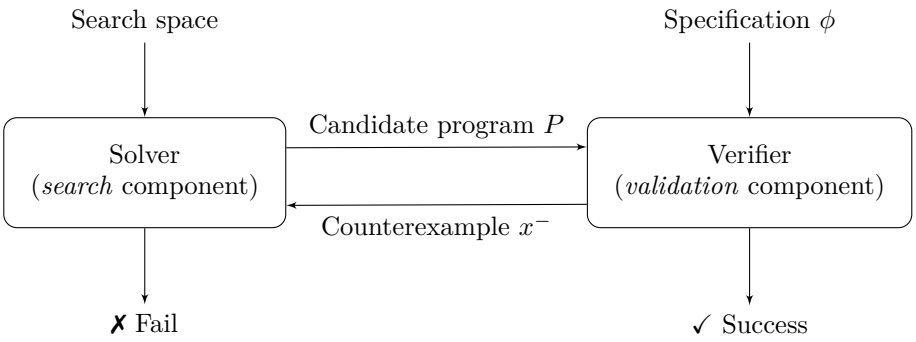


Figure 3.2: Counterexample-guided inductive synthesis.

CEGIS was introduced by Solar-Lezama in his dissertation for SKETCH [132] (covered in more detail in §3.3.1). Later, Jha and Seshia generalized the principle to *oracle-guided inductive synthesis (OGIS)* [58]. In OGIS, the solver first searches for a program candidate using a simplified specification (as described above), and then consults an *oracle* regarding validity of this candidate. Different flavors of OGIS use different kinds of oracles. CEGIS, the most popular variant of OGIS, requires a *correctness oracle*: for a given candidate program P, the oracle may return "YES" if P satisfies the required specification ϕ, or ("NO", x^-) if P does not satisfy ϕ and x^- is a counterexample – an input to P on which it violates ϕ. Other useful oracle queries have been studied in the literature; the most interesting one, the *distinguishing input oracle*, is covered in more detail in §3.2.1.

3.2.1 Distinguishing Inputs

The idea of distinguishing inputs has been first presented by Jha et al. in 2010 [59]. In this work, the goal is to automatically synthesize *loop-free programs* from a library of *basic components* that satisfy a given specification. Such a procedure is useful in multiple domains, including bitvector manipulation and function deobfuscation, studied in the paper.

The problem definition assumes the existence of a complete but expensive *validation oracle*. It takes as input a candidate program P and verifies whether it satisfies the specification. Depending on the domain

and the specification, writing such an oracle may be a challenging task. For instance, if the specification is given by an alternative inefficient implementation P_ϕ, we can run a program equivalence tool on P and P_ϕ to detect whether they behave identically on all possible inputs. However, program equivalence in a general case is undecidable. In general, the best validation oracle for most domains and specifications tends be the user, who can inspect the candidate program P manually. Obviously, this approach prevents us from querying the validation oracle automatically during synthesis often and in a scalable way.

To avoid querying the validation oracle, Jha et al. reformulate the problem in terms of *program distinguishability on concrete inputs*. Instead of a validation oracle, they require an *I/O oracle* that can produce a valid program output on an arbitrary given input. They then reduce the problem using these key observations:

1. If a program P is consistent with a specification ϕ, it must be consistent with any input/output pairs $(i_1, o_1), \cdots, (i_n, o_n)$ produced by the I/O oracle.

2. Let P be an *incorrect* program that is consistent with n input/output pairs $(i_1, o_1), \cdots, (i_n, o_n)$ produced by the I/O oracle. If there exists a correct program P^*, then P and P^* should produce the same outputs on all inputs $i_1, \cdots, i_n$ but they should produce different outputs on some other input i^*.

Observation Item 1 tells us that a program that is consistent with a finite number of valid input/output pairs is a good candidate for a program that may be consistent with the entire specification ϕ. Observation Item 2 tells us that an incorrect candidate that happens to be consistent with a chosen set of inputs can be detected by extending the set with a new *distinguishing input*. That input yields different behavior for the current candidate program P and some other possible candidate P^* (which is still consistent with the previous inputs). If such input does not exist, then all programs that are consistent with the chosen set of inputs are *semantically equivalent* – they produce the same outputs on *all inputs*. Thus, either any such candidate is correct (which can be verified with a single call to the validation oracle), or a correct program

```
function CBSynthesis(I/O oracle 𝓘, validation oracle 𝓥, components L)
    Set of examples E ← {i_0, 𝓘(i_0)}  // Initialize E with an arbitrary input
    loop
        I/O constraint ψ_E ← BuildIOConstraint(E, L)
        Candidate program P ← SAT(ψ_E)
        if P = ⊥ then
            return "Components L are insufficient to satisfy the spec."
        Distinguishing constraint ψ_P ← BuildDistConstraint(E, P)
        Distinguishing input i* ← SAT(ψ_P)
        if i* = ⊥ then
            if 𝓥(P) then return P
            else return "Components L are insufficient to satisfy the spec."
        E ← E ∪ {i*, 𝓘(i*)}
```

Figure 3.3: The learning procedure of Jha et al. [59]. The I/O constraint ψ_E constructs a program that is consistent with the current set of input/output pairs E. The distinguishing constraint ψ_P constructs an input i^* that yields different behavior for the current candidate program P and some other program P^* that is consistent with E. For finite domains, CBSynthesis either terminates with a semantically unique solution or reports that components L are insufficient to build one.

cannot be constructed from given components. The final procedure is summarized at a high level in Figure 3.3.

It turns out that for the bitvector domain, even with a random choice of distinguishing inputs, this procedure converges to a semantically unique program in at most 25 steps [59]. More generally, the number of iterations depends on the chosen distinguishing inputs. They split the search space for the programs more or less aggressively, depending on the number of collisions that different operators yield on these inputs. This property surfaces in many settings; for example, when replacing an SMT solver with random sampling to discover distinguishing inputs faster. In the bitvector domain, most common operators are more influenced by rightmost bits of the input [145, Chapter 2]. This suggests that sampling inputs with a bias for more variety in rightmost bits should discover distinguishing inputs faster, which was confirmed by Jha et al. experimentally.

Godefroid and Taly took this idea further in their automated synthesis of symbolic instruction encodings [38]. They determined that for a given subset of candidate functions, there exists a subset of *universal*

distinguishing inputs—a set that is *a priori* guaranteed to distinguish any two functions from the subset. Moreover, they have designed the templates (function subsets) and corresponding universal distinguishing inputs that cover 534 instructions of Intel x86 ALU. The domain here is the same bitvector operations as in the work of Jha et al., but without a verification oracle.

End-user confidence Distinguishing inputs are useful not only from a technical perspective as a tool for speeding up synthesis. They also play an important role as a medium for communication with the user. Consider a program synthesis system that is exposed to a user, who iteratively introduces constraints on the synthesis task. At each round, the system presents the user with a candidate program, which satisfies the constraints accumulated so far. Ultimately, the user here acts as a verification oracle: she needs to determine when the current program satisfies her overall intent and does not require adding more constraints to synthesize a better program. However, in some cases such a verification may be cumbersome or even impossible for the user. In such cases, the system may help her with interactive feedback, suggesting ways to validate that the current candidate program is unambiguous. Distinguishing inputs is one such possible feedback.

Mayer et al. introduced this idea in their FlashProg system, and called it *conversational clarification*. FlashProg is a system for *programming by examples* – a subfield of program synthesis where specification is provided by the means of inputs-ouput examples (Chapter 7). Programming by examples has been applied to many different domains; in FlashProg, the chosen one is *data wrangling* (§2.1). In this domain, the user is trying to extract some information from a semi-structured text file by the means of providing examples of extracted data, and the system is synthesizing an extraction script in the underlying DSL. In such a setting, verifying the validity of a candidate extraction script is tedious: it requires verifying the entire extracted dataset and matching it against the input file.

To alleviate the verification workload, FlashProg automatically detects when the current candidate program is ambiguous. It examines

alternative programs in the DSL that are consistent with the current examples, runs them on the entire input document, and detects any discrepancies in their output. A discrepancy between the outputs of a candidate program and its alternative suggests a location in the input document that serves as a distinguishing input. FlashProg then proactively asks the user a question: "Should the extracted output in this location be X, Y, or something else?" Such proactive clarification points the user to sources of ambiguity, allowing her to quickly find errors in the output and increasing her confidence in the final result.

3.3 Syntactic Bias

As outlined in §1.3, one of the key ideas to scaling up modern program synthesis is syntactically restricting the space of possible programs. This *syntactic bias* can be expressed by various means such as a partial program sketch, a grammar, or a domain-specific language (DSL). In this subsection, we introduce these common means, and discuss practical design considerations for a tractable and usable DSL.

3.3.1 Sketching

SKETCH [132] is a synthesis system that allows programmers to provide insights through a partial program called *sketch*, which expresses the high-level structure of the intended implementation but leaves holes for low-level implementation details. The SKETCH synthesizer fills up these holes from a finite set of choices such that the completed program satisfies the provided specification using the CEGIS algorithm. The main idea in using sketches is to make synthesis accessible to programmers as they can provide insights easily using the programming model formalism they already know without having to learn other formalism such as specification languages or theorem proving.

The syntax for the SKETCH language is quite similar to a language like C with only one additional feature – a symbol `??` that represent an unknown constant integer value. A simple example sketch is shown in Figure 3.4. The sketch represents a partial program with an unknown integer `h` and a simple assertion. The `harness` keyword indicates to the

```
1 harness void tripleSketch(int x) {
2     int h = ??; // hole for unknown constant
3     assert h * x == x + x + x;
4 }
```

Figure 3.4: A simple sketch for computing three times an input value x.

```
1 int tripleSketch(int x) implements tripleRef {
2     int h = ??; // hole for unknown constant
3     return h * x;
4 }

6 int tripleRef(int y) {
7     return y + y + y;
8 }
```

Figure 3.5: A simple sketch with reference implementation sketch.

synthesizer that it should compute a value for `h` such that the assertion is satisfied for all input values `x`. For this example, the SKETCH synthesizer computes the value $\mathtt{h} = 3$ as expected.

In addition to assertions, another mechanism to write desired specification in SKETCH is to provide a reference implementation. The simple example sketch from Figure 3.4 can be equivalently rewritten with a reference implementation specification as shown in Figure 3.5. The implements keyword specifies the synthesizer to compute the values for holes such that completed sketch program has the same functional behavior as the reference implementation.

These unknown integers can be used to specify a hypothesis space over a richer class of expressions in the sketches. For example, the sketch in Figure 3.6 uses an integer hole to define a space of five different binary operator expressions over two integer values `lhs` and `rhs`. The SKETCH language also provides a succinct language construct to specify such expression choices: `lhs {| + | - | * | / | % |} rhs`.

The SKETCH language also supports a generator function construct to specify a hypothesis space of possible code fragments that can be used to complete sketches. The generator functions can be used in

```
1  int chooseBinOp(int lhs, int rhs) {
2      int h = ??;
3      assert h < 5;

5      if (h == 0) return lhs + rhs;
6      if (h == 1) return lhs - rhs;
7      if (h == 2) return lhs * rhs;
8      if (h == 3) return lhs / rhs;
9      if (h == 4) return lhs % rhs;
10 }
```

Figure 3.6: Using holes to encode the space of richer expressions.

```
1  generator int linexp(int x, int y) {
2      return ?? * x + ?? * y;
3  }

5  harness void main(int x, int y) {
6      assert linexp(x, y) == x + x + y;
7      assert linexp(x, y) == x + y + y;
8  }
```

Figure 3.7: A sketch using a generator function.

the same way as a function, but the key difference between them and functions is that every call to a generator will be replaced by a concrete piece of code defined by the generator and the generator function can produce different code fragments for different calls. For example, consider the following sketch in Figure 3.7 with a generator function for linear expressions (`linexp`). The result of running the synthesizer on this sketch is shown in Figure 3.8.

The main expressive power of generator functions comes from their ability to recursively define a large space of expressions succinctly. For example, Figure 3.9 shows a recursive generator function defining a context-free grammar of possible expressions over two variables.

The SKETCH language supports several other advanced features such as function closures, higher order functions, and algebraic data

```
1  void main(int x, int y) {
2      assert (((2 * x) + y) == ((x + x) + y));
3      assert ((x + (2 * y)) == ((x + y) + y));
4  }
```

Figure 3.8: The result of running the SKETCH synthesizer on the sketch in Figure 3.7.

```
1  generator int recGen(int x, int y) {
2      int h = ??;
3      if (h == 0) return x;
4      if (h == 1) return y;
5      int a = recGen(x, y);
6      int b = recGen(x, y);
7      if (h == 2) return a + b;
8      if (h == 3) return a - b;
9      if (h == 4) return a * b;
10     if (h == 5) return a / b;
11 }
12 void main(int x, int y) {
13     assert recGen(x, y) == (x + y) * (y - x);
14 }
```

Figure 3.9: The result of running the SKETCH synthesizer on the sketch in Figure 3.7.

types. A more detailed description about the language and different features can be found in the SKETCH language manual.[2]

3.3.2 Syntax-Guided Synthesis

Syntax-Guided Synthesis [3] is a recent community effort towards formalizing the problem of program synthesis where the logical specification is supplemented with a user-provided syntactic template to constrain the hypothesis space of possible programs. The input to the syntax-guided synthesis problem (SyGuS) consists of a background theory T that defines the vocabulary and interpretation of function and relation symbols, a semantic correctness specification for the desired program given by a logical formula ϕ, and a syntactic set of candidate implementations

[2] https://people.csail.mit.edu/asolar/manual.pdf

```
1  ; set the background theory to LIA
2  (set-logic LIA)

4  ; grammar for max2 candidate implementations
5  (synth-fun max2 ((x Int) (y Int)) Int
6     ((Start Int (x y 0 1
7                 (+ Start Start)
8                 (- Start Start)
9                 (ite StartBool Start Start)))
10     (StartBool Bool ((and StartBool StartBool)
11                      (or  StartBool StartBool)
12                      (not StartBool)
13                      (<=  Start Start)
14                      (=   Start Start)
15                      (>=  Start Start)))))

17 ; universally quantified input variables x and y
18 (declare-var x Int)
19 (declare-var y Int)

21 ; correctness constraints on the max2 function
22 (constraint (>= (max2 x y) x))
23 (constraint (>= (max2 x y) y))
24 (constraint (or (= x (max2 x y)) (= y (max2 x y))))

26 ; synthesize command
27 (check-synth)
```

Figure 3.10: The SyGuS formulation for the `max` function over two variables.

given by a context-free grammar G. The computation problem then is to find an implementation from the set of candidate implementations (i.e. a derivation in the grammar G) such that the implementation satisfies the specification ϕ in the given theory T.

As an example, consider the problem of synthesizing a `max2` function that computes the maximum value of two given values x and y. The SyGuS formulation of the problem is shown in Figure 3.10. The encoding first defines the background theory of Linear Integer Arithmetic (LIA), where each variable is either a Boolean or an in-

teger, and the vocabulary consists of Boolean and integer constants, addition, comparisons, and conditionals. The grammar for the unknown function `max2` defines the type signature of the function and the space of possible candidate implementations. The grammar in the example corresponds to if-then-else expressions (ite) and linear expressions with additions and subtractions over the terminal values `x`, `y`, `0`, and `1`. The correctness specification defines the following logical constraint $\phi_1 \wedge \phi_2$, where $\phi_1 \equiv \texttt{max2(x,y)} \geq x \wedge \texttt{max2(x,y)} \geq y$, and $\phi_2 \equiv \texttt{max2(x,y)} = x \vee \texttt{max2(x,y)} = y$. Given this formulation, one candidate implementation from the grammar that satisfies the specification is `(ite (>= x y) x y)`.

The SyGuS effort has resulted in a community effort in collecting benchmarks from various synthesis domains into a common format. It has also lead to SyGuS-Comp, an annual synthesis competition, that allows solvers to compete on a collection of benchmarks. In the 2015 competition, 8 different solvers competed in three tracks corresponding to LIA (Conditional Linear Integer arithmetic track), INV (invariant generation track), and General (General SyGuS track) [5]. The SyGuS solvers employed various solving techniques including enumerative search, stochastic search, constraint-based search, machine learning, etc.

3.3.3 DSL Design

The choice of a domain-specific language is inspired by several factors.

Balanced Expressivity: On one hand, the DSL should be expressive enough to represent a wide variety of tasks in the underlying task domain. On the other hand, it should be restricted enough to allow efficient search.

Choice of Operators: The DSL should be made up of operators that allow efficient reasoning. For instance, the operators should have small inverses to enable a top-down deductive search strategy (§7.3) to be efficient.

Naturalness: The programs in the DSL should involve natural computational patterns that can be easily understood by the users.

This can increase user's confidence in the system. In fact, these computational patterns should be similar to how programmers might have written the code themselves. These programs might be read by users, who might then select between these programs, edit them, and even use them as part of larger workflows.

Efficiency: The operators in the DSL should have efficient implementations. This is important if the synthesized program is expected to be run in a tight loop over large number of inputs.

The synthesis designer can select an existing DSL or its subset that meets the desired constraints. For instance, Cheung et.al. [22] leveraged a subset of the SQL DSL when synthesizing optimized database-backed applications. Recently, Panchekha and Torlak leveraged a subset of the CSS language [103] for synthesizing non-trivial spreadsheet styles from layout constraints. However, in other cases, new DSLs might need to be designed. An approach to designing an appropriate DSL is to first manually write down programs for a variety of tasks in the underlying task domain and then identify common patterns/templates for inclusion in the DSL. Designing a good DSL is often the first non-trivial idea in developing a good program synthesizer. We next discuss such ideas behind some non-trivial DSLs from the data wrangling domain.

Flash Fill DSL The Flash Fill DSL for syntactic string transformations [43] contains programs that take an n-ary tuple of strings as input and return a string as output. These programs involve computing substrings of the strings in the input tuple, and then concatenating them appropriately along with some constant strings. There is also support for restricted forms of loops that concatenate a sequence of substrings from the input string (with the same delimited constant string) to facilitate more sophisticated string transformations as in abbreviation computation or string reversal. The Flash Fill programs can contain conditionals at the very top level to allow for different transformations for different kinds of data formats—this feature is useful for normalizing strings that may come in multiple formats. Consider, for instance, the task shown in Figure 2.1. The synthesized program extracts the

first two words from the input string, converts them to lowercase, and concatenates them separated by a space character.

The most interesting aspect of the Flash Fill DSL is the design of its substring construct that takes as input a string s and two position expressions in that string, and evaluates to the substring between those positions. A position expression can either be a constant offset (from the start or end of the string s) or is a $\texttt{Pos}(s, r_1, r_2, k)$ construct denoting the k^{th} occurrence of a position p such that (some suffix of the part of the string s on) the left side of p matches with r_1 and (some prefix of the part of the string s on) the right side of p matches with r_2. Consider the expression $e = \texttt{Substring}(s, p, p')$, where $p = \texttt{Pos}(s, r_1, r_2, k)$ and $p' = \texttt{Pos}(x, r_1', r_2', k')$. When $r_1 = r_2' = \epsilon$, then e describes a constraint over the substring to be extracted. When $r_2 = r_1' = \epsilon$, then e describes a constraint over the context around the substring to be extracted. The general case is even more expressive allowing constraints over both the substring to be extracted and its context with use of simple (and hence efficiently learnable) regular expressions. For instance, if $p = \texttt{Pos}(s, \epsilon, Word, 2)$ and $p_2 = \texttt{Pos}(s, Word, \epsilon, 2)$, then e describes the second word in string s. If $p_1 = \texttt{Pos}(s, '[', \epsilon, 1)$ and $p_2 = \texttt{Pos}(s, \epsilon, ']', 1)$, then e describes the content within the (first occurrence of opening and closing) brackets.

The Flash Fill DSL has been extended to allow semantic string transformations that involve lookup operations into some background tables [127]. This is done in a tightly integrated manner with the rest of the language by taking substrings of the input strings, using them to index into the key columns of a table and read a value from some other column, and then further treating that value as another input string for any subsequent computation.

FlashExtract DSL The FlashExtract DSL [78] contains programs that take a large string (representing a semi-structured text/log file) and a data structure definition (composed of nested structure and sequence constructs) and returns an instance of that data structure. Each program is composed of multiple sub-programs whose data-flow dependence follows the data structure definition. There are two kinds of

sub-programs: One that generates a substring of a given input string, and the other that generates a list of substrings of a given input string. The latter involves splitting the input string into a list of lines, filtering the result, and then mapping it to another list using a substring operator.

FlashRelate DSL The FlashRelate DSL contains programs that take a two-dimensional array (representing a semi-structured spreadsheet) as input and return a list of n-ary tuples (representing a relational table) as output [10]. These programs extract one of the output columns via a filter operation over cells of the two-dimensional input array, while other output columns are extracted as map operations over cells of an already extracted column. The filter and map operations employ *spatial constraints* over cells of the two-dimensional array in addition to regular expression based constraints over values in those cells. Consider, for instance, the task in Figure 2.5. The synthesized program computes the `Harvest` column via a filter operation that identifies all non-empty cells that are located *vertically below* any cell with string content "value". The `Date` column is computed via a map operation (over cells of the input array whose content appears in the `Harvest` column) that picks the cell on the *immediate right*. The `Country` column is computed via a map operation that picks the first *cell on the left side* containing alphabetic content.

3.4 Optimization

A basic formulation of the program synthesis problem asks to find *any* program P in an underlying space $\mathcal{L}$ (e.g. a DSL) that satisfies a spec ϕ. However, in many settings we are interested not just in *any* program, but in the *best* program according to some *cost function*. This is known as *ranking* or *optimization* in synthesis. Some interesting cost functions that arise in practice are:

Program speed. This is the default choice in *superoptimization* (§2.6).

Robustness. This is the default setting in *programming by examples* (Chapter 7) for *data wrangling* (§2.1), where a program is learned

on a small selection of input data, but it should be applicable on the rest of yet-unseen user data. In other words, learning should not overfit to provided input-output examples.

"Naturalness" or "readability". This is a measure of a program being acceptable to a developer. For instance, this cost function may take into account common idioms in an underlying language. The notion of idioms may be exploited explicitly, either by mining them from a corpus of source code [2], or by delegating ranking to a search engine [151].

Program AST size is often used as an approximation to all aforementioned cost functions. Arguably, a smaller program is more readable, a smaller program may be faster, and a smaller program may be more generally applicable. However, it is not always the case. For instance, a constant program has size 1 but it is not robust: it outputs the same value on all inputs. As another example, on certain Intel x86 microarchitectures the `inc` instruction may be slower than `add 1`, despite being shorter and more specific.

Ranking in program synthesis has been approached with many different techniques, including Markov Chain Monte Carlo (MCMC) methods (explained in detail in §6.1 and 6.2), version space algebras (§7.2), machine learning (§6.3 and 7.4.1), and metasketches [15].

Markov Chain Monte Carlo MCMC techniques in program synthesis employ a variation of genetic programming, where Metropolis-Hastings algorithm is used to explore a search space of possible programs. In MCMC sampling, we aim to draw programs from the space with probability proportional to their cost. In the limit, the most frequently sampled program will be the global optimum on the cost function; in practice, running an MCMC sampling process for a reasonable amount of time usually discovers a "sufficiently good" local optimum. In a sense, Metropolis-Hastings algorithm can be viewed as a "smart" hill climbing algorithm that is resilient to getting stuck at local optima and only limited by its time budget [123].

Version Space Algebras Version Space Algebras (VSAs) offer a succinct representation of the program space that is amenable to *structure-based ranking* [113]. Assuming a monotonic cost function $h(P)$, [3] the problem of finding the topmost program in a VSA is solvable with a *beam search* through the VSA structure, which constructs the desired program from its subexpressions, bottom-up.

Machine Learning Machine Learning methods have been used extensively for program synthesis and optimization. Their applications include automatically learning cost functions [7, 128], dealing with noise in the specification [115], and even end-to-end program synthesis [40, 64, 76, 98, 105, 118]. We describe some of these applications in §6.3 and 7.4.1.

Metasketches A *metasketch* [15] is a generalization of a sketch (§3.3.1). Instead of a single partial program, it gives an ordered family of such partial programs, along with a cost function and a *gradient*, which suggests which sketches in the family may contain a program with a lower cost. In other words, it describes a search space as a family of finite sub-spaces that can be explored independently, and provides guiding functions to navigate these sub-spaces in a cost-effective way. Using this formulation, Bornholt et al. present an optimal synthesis algorithm, which explores multiple local sub-spaces independently in parallel using CEGIS, and coordinates their findings using a separate global component to guide the search toward more optimal programs.

[3] A cost function $h(P)$ is called *monotonic* over the program structure, if better-valued subexpressions w.r.t. h lead to better-valued expressions w.r.t. h:

$$h(P_1) \geq h(P_2) \;\Rightarrow\; h(F[P_1]) \geq h(F[P_2])$$

where F is any DSL operator (straightforwardly generalizable to arbitrary arity).

4

Enumerative Search

Enumerative search based synthesis techniques have proven to be one of the most effective techniques for synthesizing small programs in rich complex hypothesis spaces. Smart pruning techniques in enumerative search have resulted in synthesizers that won the recent ICFP program synthesis competition [1] and the SyGuS competition [6]. The key idea in enumerative techniques is to first structure the hypothesis space using some program metrics such as program size, complexity etc., and then enumerate programs in the space with pruning to efficiently search for a program that satisfies the specification. The enumerative algorithms are typically designed to be semi-decidable for the infinite hypothesis spaces, but they are generally applicable for almost all kinds of hypothesis spaces and specification constraints.

4.1 Enumerative Search

We now describe a simple enumerative synthesis algorithm for hypothesis spaces that are defined using a CFG (similar to SyGuS [3]), but the algorithm can be easily extended to other forms of hypothesis spaces such as partial programs, context-sensitive languages etc. Let the hypothesis

```
S → x | y | (S + S) | (S - S) | if(B,S,S)
B → (S ≤ S) | (S = S) | (S ≥ S)
```

Figure 4.1: A simple context-free grammar for defining a space of conditional linear integer arithmetic expressions.

space be defined by a context-free grammar $G = (V, \Sigma, R, S)$, where V denotes the set of non-terminals in the grammar, Σ denotes a finite set of terminals including constants and program variables, R denotes a finite relation from V to $(V \cup \Sigma)^*$ corresponding to the grammar production rules, and S denotes the start symbol. For example, a simple grammar for defining the hypothesis space of a set of conditional linear arithmetic expressions is shown in Figure 4.1, where $V = \{S, B\}$, $\Sigma = \{x, y, +, -, if, (,), \leq, =, \geq\}$, $S = \{S\}$, and the relation R corresponds to the 8 relations shown in the figure.

The derivations (programs) in the grammar G can be enumerated either in a top-down or a bottom-up fashion to find a derivation that is consistent with the given specification.

4.1.1 Top-down Tree search

A simple top-down enumerative algorithm for searching over programs that satisfy a specification ϕ in a hypothesis space defined by a grammar G is shown in Figure 4.2. The algorithm enumerates the derivations in the grammar and maintains an ordered list of partial derivations $\widetilde{P}$ in a top-down manner starting from the start symbol S. In each iteration, it selects the first derivation p from the set $\widetilde{P}$ using the RemoveFirst() function and checks whether p satisfies the given specification ϕ. If yes, the algorithm returns p as the desired program. Otherwise, the algorithm computes the set of all non-terminal nodes $\widetilde{\alpha}$ in the partial derivation that can be expanded and ranks them using a pre-defined RankNonTerminal function. It then considers each non-terminal α in an ordered fashion and computes the set of possible production rules $\widetilde{\beta}$ for expanding α. The algorithm searches over the set of production rules again in a ranked order (using RankProductionRule function) to compute expanded derivations p' where the non-terminal node α is

```
function EnumTopDownSearch(grammar G, spec φ)
    P̃ ← [S] // An ordered list of partial derivations in G
    P̃_v ← {S} // A set of programs
    while P̃ ≠ ∅ do
        p ← RemoveFirst(P̃)
        if φ(p) then // Specification φ is satisfied
            return p
        α̃ ← NonTerminals(p)
        foreach α ∈ RankNonTerminals(α̃, φ) do
            β̃ ← {β|(α, β) ∈ R}
            foreach β ∈ RankProductionRule(β̃, φ) do
                p′ ← p[α → β]
                if ¬Subsumed(p′, P̃_v, φ) then
                    P̃.Insert(p′)
                    P̃_v ← P̃_v ∪ p′
```

Figure 4.2: A simple top-down enumeration algorithm to search for a derivation p in a hypothesis space defined by a CFG G that satisfies a given specification ϕ.

replaced by β. Finally, the algorithm prunes the search space by avoiding adding newly expanded programs p' if p' is already subsumed by some program in $\widetilde{P_V}$ that has previously been considered by the algorithm. The algorithm iteratively continues until it finds a program p satisfying the specification. This algorithm is also typically provided with an additional bound for the maximum size of the program derivations, which is used to terminate the search.

Consider the grammar for conditional linear integer arithmetic in Figure 4.1 (without the subtraction operator for brevity), and let the specification ϕ be provided as the following input-output examples : $\phi \equiv \{(x = 0, y = 1, f(x, y) = 1), (x = 1, y = 2, f(x, y) = 3)\}$, where f denotes the unknown function to be synthesized. A possible function f that satisfies the specification is $f(x, y) = x + y$. An example run of the top-down enumerative algorithm is shown in Figure 4.3, with a simple ranking function that prefers expressions with fewer number of non-terminals. Note that the subsumption check can help prune the search by disallowing programs such as $S_1 + x$ to be added to $\widetilde{P}$ if $x + S_2$ is already present in $\widetilde{P_v}$.

$\widetilde{P}$	p	$\widetilde{\alpha}$	$\widetilde{\beta}$
$[S]$	S	$[S]$	$[x,\ y,\ S_1 + S_2,$ $\mathsf{if}(B,\ S_1,\ S_2)]$
$[x,\ y,\ S_1 + S_2,\ \mathsf{if}(B,\ S_1,\ S_2)]$	x	$\bot$	$\bot$
$[y,\ S_1 + S_2,\ \mathsf{if}(B,\ S_1,\ S_2)]$	y	$\bot$	$\bot$
$[S_1 + S_2,\ \mathsf{if}(B,\ S_1,\ S_2)]$	$S_1 + S_2$	$[S_1,\ S_2]$	$[x + S_2,\ y + S_2,$ $S_3 + S_4 + S_2,$ $\mathsf{if}(B_2,\ S_3,\ S_4) + S_2,$ $S_1 + x,\ S_1 + y,$ $S_1 + S_3 + S_4,$ $S_1 + \mathsf{if}(B_2,\ S_3,\ S_4)]$
$[x + S_2,\ y + S_2,\ \cdots]$	$x + S_2$	S_2	$[x + x,\ x + y,$ $x + S_5 + S_6,\ \ldots]$
$[x + x,\ x + y,\ \cdots]$	$x + x$	$\bot$	$\bot$
$[x + y,\ \cdots]$	$\mathbf{x + y}$		

Figure 4.3: An example run of the top-down enumerative search algorithm on the addition of two numbers x and y.

4.1.2 Bottom-up Tree-search

The bottom-up tree search approach caches the results of intermediate programs constructed during the search to prune a redundant family of programs. Specifically, if two derivations (programs) p and p' are equivalent with respect to the specification ϕ, then only one of them needs to be considered during the search. For the top-down algorithm shown in Figure 4.2, this check is performed inside the SUBSUMED function to avoid adding a derivation p' to the set of derivations $\widetilde{P}$. This optimization does not greatly reduce the search space in the top-down enumerative strategy because of a limitation that only full derivations can be evaluated for equivalence. However, it can have a dramatic effect on a bottom-down search technique [4, 141], which we describe below.

A simple bottom-up enumerative algorithm is shown in Figure 4.4. The algorithm first starts building a set of leaf expressions in the grammar G in the order of their size `progSize`. It then incrementally builds the set of candidate expressions $\widetilde{E}$ using the smaller expressions to find one that satisfies the specification. The key idea in this algorithm is to maintain a semantically unique set of expressions $\widetilde{E}$, i.e. no two

```
function EnumBottomUpSearch(grammar G, spec φ)
    Ẽ ← {Φ} // Set of expressions in G
    progSize ← 1
    while True do
        C̃ ← EnumerateExprs(G, Ẽ, progSize)
        foreach c ∈ C̃ do
            if φ(c) then // Specification φ is satisfied
                return c
            if ¬∃ e ∈ Ẽ : Equiv(e, c, φ) then
                Ẽ.Insert(c)
        progSize ← progSize + 1
```

Figure 4.4: A simple bottom-down enumeration algorithm to search for a derivation e in a hypothesis space defined by a CFG G that satisfies a given specification ϕ.

expressions e and e' in $\widetilde{E}$ are functionally equivalent with respect to the specification ϕ. This pruning allows the bottom-up enumeration algorithm to significantly decrease the space of expressions that need to be considered before finding the desired expression.

Consider the grammar for conditional linear integer arithmetic in Figure 4.1 with an additional leaf denoting constant 1 and the specification of the unknown function f to be synthesized to be that of the maximum function such that $\phi : f(x,y) \geq x \wedge f(x,y) \geq y \wedge (f(x,y) = x \vee f(x,y) = y)$. The bottom-up algorithm uses the CEGIS algorithm to incrementally build the input-output examples for the specification. Let's assume the enumeration algorithm first starts with a random example of $(x = 0, y = 1, f(x,y) = 1)$. It can construct the expression y as a satisfying expression. The verifier then returns a counter-example $(x = 1, y = 0, f(x,y) = 1)$. Given these two examples, the algorithm constructs the conforming expression 1 and the verifier again returns the counter-example $(x = 0, y = 0, f(x,y) = 0)$. The search algorithm can now return $x + y$ as a conforming expression. Note that during this search since both $x+y$ and $y+x$ are functionally equivalent, only one of these expressions are added to the candidate set for constructing larger expressions. This iteration between the bottom-up search algorithm and the verifier continues until the search algorithm finds the desired

Candidate Expressions	Counter-example from Verifier
x	$(x = 0, y = 1, f(x, y) = 1)$
y	$(x = 1, y = 1, f(x, y) = 1)$
1	$(x = 0, y = 0, f(x, y) = 0)$
$x + y$	$(x = 1, y = 1, f(x, y) = 1)$
$\texttt{if}(x \leq y, y, x)$	ϕ

Figure 4.5: An example run of the bottom-up enumerative search algorithm on the maximum of two numbers x and y [3].

expression $\texttt{if}(x \geq y, x, y)$. An example run of an enumerative solver is shown in Figure 4.5.

4.2 Bidirectional Enumerative Search

The previous enumerative algorithms search for programs that when run on the inputs produce the corresponding outputs, which can also be explained as performing a *forward* search starting from the inputs (input states) and transforming the states through a series of intermediate states until obtaining the desired output states. This search space of programs grows exponentially with the size (depth) of the programs. In some cases, the search can be performed more efficiently with a bidirectional search, where the forward search from the input states is combined with a backward search from the desired output states.

A simple bidirectional enumerative algorithm is shown in Figure 4.6. Given a specification $\phi \equiv (\phi_{\texttt{pre}}, \phi_{\texttt{post}})$, where $\phi_{\texttt{pre}}$ specifies the set of input states and $\phi_{\texttt{post}}$ specifies the set of output states, the algorithm maintains two sets of expressions $\widetilde{F}$ and $\widetilde{B}$. The set $\widetilde{F}$ comprises of expressions derived by performing a forward enumerative search starting from the input states $\phi_{\texttt{pre}}$, whereas the set $\widetilde{B}$ comprises of expressions derived by performing a backward search starting from the output states $\phi_{\texttt{post}}$. It iteratively builds the two sets in increasing order of program sizes until finding an expression $f \in \widetilde{F}$ and $b \in \widetilde{B}$ such that the states corresponding to f and b can be matched. The resulting program is obtained by combining the two expressions.

function BIDIRECTIONALSEARCH(grammar G, spec $\phi \equiv (\phi_{\texttt{pre}}, \phi_{\texttt{post}})$)
  $\widetilde{F} \leftarrow \phi$ // Set of expressions from Forward search
  $\widetilde{B} \leftarrow \phi$ // Set of expressions from Backward search
  progSize $\leftarrow 1$
  while $\neg\exists f \in \widetilde{F}, b \in \widetilde{B}$: MATCHSTATE(f, b) **do**
    $\widetilde{F} \leftarrow$ ENUMFORWARDEXPRS$(G, \widetilde{F}, \phi_{\texttt{pre}}, \texttt{progSize})$
    $\widetilde{B} \leftarrow$ ENUMBACKWARDEXPRS$(G, \widetilde{B}, \phi_{\texttt{post}}, \texttt{progSize})$
    progSize $\leftarrow$ progSize $+ 1$
  $p \leftarrow f \oplus b$, where $\exists f \in \widetilde{F}, b \in \widetilde{B}$: MATCHSTATE(f, b)
  return p

Figure 4.6: A bidirectional enumeration algorithm to search for a derivation p in a hypothesis space defined by a CFG G that satisfies a given specification ϕ with inputs satisfying $\phi_{\texttt{pre}}$ and outputs satisfying $\phi_{\texttt{post}}$.

The bidirectional enumerative search has been used for many synthesis domains including learning programs to solve geometry constructions [48] and scaling up superoptimization of assembly code [109].

4.3 Offline Exhaustive Enumeration and Composition

Another interesting, but resource intensive, enumeration technique is to perform an offline exhaustive enumeration of all programs upto a given size. The programs are then evaluated on a large set of pre-defined inputs to obtain a corresponding mapping function from programs to the input-output pairs. Finally, given a set of input-output examples, the programs are retrieved using the mapping function. This strategy has been successfully implemented in systems such as MAGICHASKELLER [67] and Unagi [1].

The Unagi system won the 2013 ICFP program synthesis competition beating several state-of-the-art program synthesis techniques [1]. Given a DSL of bit-vector programs used in the competition, the Unagi system first perform an offline exhaustive enumeration of all expressions in the DSL upto a given size (15 in their case). During enumeration, several pruning techniques were employed to merge equivalent programs. For example, the expression `(not (if x (not y) z))` was normalized to `(if x y (not z))`. After performing the enumeration, each program

was evaluated on 256 pre-defined input bitvectors to create a mapping from programs to the input-output examples. During the competition, for smaller programs upto size 15, the Unagi system used the direct mapping to quickly find the corresponding program.

For enumerating larger programs, Unagi used an incomplete but effective pruning strategy based on input-output equivalence. It prunes expressions that results in the same output on 256 inputs to construct larger programs. In addition, using the training problems, Unagi also learnt several syntactic priors over the space of large programs, e.g. it learnt that the expression trees tend to be unbalanced in the training data and the initial accumulator value for fold problems was always zero. It used these syntactic priors to further guide the exhaustive enumeration.

Finally, to learn even larger programs of size greater than 50, it used a unification strategy to decompose the input-output examples to different sets. Each set of input-output examples could be solved by a smaller program and then the sets are glued together using `if` conditions. The Unagi system used 32 threads running independent search strategies and used 3000 hours of compute time on Amazon's EC2 cloud for the competition. Notably, this technique of unification of smaller expressions to learn larger programs was also used by the enumerative synthesis system that won the 2016 SyGuS competition [6].

5

Constraint Solving

Many successful applications of program synthesis rely on some *constraint solving* technique. In general, constraint solving refers to finding an instantiation of free variables in a formula (a *model*) that would make the formula true [37]. The key idea in applying constraint solving to program synthesis is encoding both the specification and the syntactic program restrictions in a single formula so that any true model corresponds to a correct program. The encoding should map variables in the formula to choices of subexpressions in the desired program.

We illustrate this idea on a simple example. Consider a small DSL of bitwise operations upon a 8-bit input variable x:

$$\begin{aligned} \text{program } P &::= \mathsf{plus}(E,E) \mid \mathsf{mul}(E,E) \mid \mathsf{shl}(E,C) \mid \mathsf{shr}(E,C) \\ \text{expression } E &::= x \mid C \\ \text{8-bit constant } C &::= \overline{\texttt{00000000}}_2 \mid \overline{\texttt{00000001}}_2 \mid \overline{\texttt{00000010}}_2 \mid \ldots \mid \overline{\texttt{11111111}}_2 \end{aligned}$$

Every program in this DSL contains a single top-most binary operation, with each operand being either the input or some 8-bit constant.

In the *theory of bitvectors* $\mathcal{T}_{\mathsf{BV}}$ [11], the following formula $\Phi(h_P, h_{E1}, h_{E2}, c_1, c_2)$ encodes any program P in this DSL that sat-

isfies the spec ϕ: $\forall x\ P(x) \geq 0$ (as a signed integer).[1] In other words, any valid assignment of free variables h_P, h_{E_1}, h_{E2}, c_1, c_2 corresponds to a valid program in the DSL.

```
(define-sort Bit8 () (_ BitVec 8)) ; A "8-bit vector" type
; Free variables of Φ
(declare-const hP Int)
(declare-const hE0 Bool)
(declare-const hE1 Bool)
(declare-const c0 Bit8)
(declare-const c1 Bit8)
(assert (and (>= hP 0) (< hP 4)))

; Definition of P(x)
(define-fun prog ((x Bit8)) Bit8
  (let ((left (ite hE0 c0 x))
        (right (ite hE1 c1 x)))
     (ite (= hP 0) (bvadd left right)
     (ite (= hP 1) (bvmul left right)
     (ite (= hP 2) (bvshl left c1)
                   (bvlshr left c1))))))
; Spec φ: ∀x P(x) ≥ 0
(assert (forall ((x Bit8)) (bvsge (prog x) #x00)))
```

The variable h_P encodes a non-deterministic choice between alternatives for the nonterminal P. It ranges in $\{0, 1, 2, 3\}$, which correspond to plus, mul, shl, and shr, respectively. h_{E1} and h_{E2} are Boolean variables that encode a similar choice for the nonterminal E used as the first or the second operand, respectively. Here `true` corresponds to C and `false` to x. Finally, c_0 and c_1 encode non-deterministic choices for the constants supplied for the left and right operand. If a program uses x instead of a constant for some operand $i \in \{0, 1\}$ (that is, $h_{Ei} =$ `false` for that model of Φ), the value of c_i in the model is irrelevant.

One possible satisfying model for Φ is $\{h_P = 3,\ h_{E0} = h_{E1} =$ `false`, $c_1 = \overline{00000001}_2$, c_0 is irrelevant$\}$. It corresponds to the program $P =$ shr$(x, \overline{00000001}_2)$, which obviously satisfies ϕ for any input.

[1] For clarity, we display most SMT formulas in this section in SMT-LIB syntax [11].

5.1 Component-Based Synthesis

5.1.1 End-to-end SMT Encoding

The first constraint-based approaches to program synthesis encoded the language and the spec into SAT/SMT constraints directly, similarly to our example above. A classical example of such an encoding is *component-based synthesis* by Jha et al. [59]. We introduced this work in §3.2.1 to present the idea of distinguishing inputs. Now, we will discuss their overall encoding of the synthesis problem.

In component-based synthesis, our syntactic bias is a library of allowed *components*. Each component is a domain-specific function that may be used in the desired program. Usually, there are no syntactic restrictions on their composition (*e.g.,* no grammar): any type-safe well-formed combination of components constitutes a valid program in the language. Without loss of generality, we assume that every program uses all provided components, each exactly once (multiple uses of a component can be achieved by including multiple copies in the library).

Jha et al. first apply their encoding to the domain of *bit manipulation programs*. In it, there is a single type (a fixed-length bitvector), thus all component usages are a priori type-safe. To solve a synthesis problem, they need to encode, at the minimum, that the desired program

(a) is well-formed,
(b) uses all components without violating their respective specs, and
(c) is consistent with the provided spec (*e.g.,* input-output examples).

In addition, to apply the distinguishing inputs technique (see §3.2.1), they encode in a separate query that the program

(d) disagrees with some other consistent program on a new input.

We now describe the *component-based synthesis* constraints (a)-(c).

Well-formedness A loop-free program is *well-formed* if it forms a directed acyclic graph (DAG) of components and their connections (that is, usages of outputs of one component as inputs in the others). In addition, it must use all components from the library exactly once.

An SMT encoding of well-formedness associates each *variable* in the program with a single *line number* where it is defined and assigned.

For component output variables O_i, this is also the line where the corresponding component f_i is used. For component input variables I_{ij}, this is the line that defines the output variable that is passed as a j^{th} argument to the component f_i.[2]

With this notation, encoding well-formedness reduces to:

(a) Ensuring that all line numbers range in their respective domain bounds (that is, program inputs are assigned in the first lines of the program, and all library components are invoked in the following lines).

(b) Ensuring consistency: all output variables (and their corresponding line numbers) should be different.

(c) Ensuring acyclicity: components should only take as input variables that were assigned earlier.

Component usage In the framework of [59], every component f_i is annotated with a *specification* $\phi_i(\vec{I_i}, O_i)$, which describes its semantics and relates its inputs $\vec{I_i}$ with its output O_i. All these specifications are appended to the synthesis constraint, with their inputs and outputs mapped to the corresponding variables in the program. To further ensure that the program's data flow is consistent, we add another constraint that ensures that an output variable retains the same value when it is used as an input to another component.

Specification The problem specification $\phi(\vec{I}, P(\vec{I}))$ relates its output (implicitly taken from the last line of the program) with its inputs. Jha et al. used input-output examples as a specification, which are easy to encode in additional clauses appended to the synthesis constraint. In general, any specification form would suffice as long as it can be encoded in the underlying SMT logic (or already represented as such). However, complex logical specifications may make the synthesis constraint more difficult to resolve, and thus a combination of input-output examples and an external validation oracle, employed by Jha et al., is preferable for efficient program synthesis.

[2] For simplicity, we represent global program arguments as additional 0-ary components in the library. They are "invoked" in the first lines of the program.

Example 5.1. Suppose we want to synthesize a program that turns off the rightmost 1 bit in a given 5-bit vector. Our desired program has a single input I and a single output O. Our library consists of 2 components, with the following specifications:

Component	Semantics
$f_1(x_1, x_2) \coloneqq x_1 \,\&\, x_2$	$\phi_1(x_1, x_2, y)\colon \;\; [y = \mathsf{bvand}(x_1, x_2)]$
$f_2(x) \coloneqq x - 1$	$\phi_2(x, y)\colon \;\; [y = \mathsf{bvsub}(x, \overline{00001}_2)]$

Finally, our spec ϕ is a single input-output example: $(\overline{01010}_2, \overline{01000}_2)$.

Figure 5.1 shows the SMT constraint that encodes this problem and includes all constraints (a)-(c), discussed above. Note that in this case ϕ contains a single input-output example, thus we need to encode only one data flow through the program (*i.e.* variables i_{jk} and o_j). In general, every input-output example requires its own set of data flow variables. The location variables ℓi_{jk} and ℓo_j, however, are only defined once: they represent structure of the desired program, which does not change across different input-output examples.

A valid model for this constraint contains the following assignments for the location variables:

$$\{\ell o_1 = 2,\, \ell o_2 = 1,\, \ell i_{11} = 1,\, \ell i_{12} = 0,\, \ell i_{21} = 0,\, \ell i_0 = 0,\, \ell o = 2\}$$

They correspond to the following program $P(x)$, which indeed turns off the rightmost 1 bit of x:

```
def P(x) { t1 := x - 1; t2 := t1 & x; return t2; }
```

5.1.2 Sketch generation and completion

The synthesis approach of Jha et al. produced a complete SMT encoding of the entire synthesis problem, given a library of components and the desired spec. While powerful, such an encoding is difficult to design and implement. An alternative solution is to split the synthesis process into *sketch generation* and *sketch completion*. The sketching technique, as introduced in §3.3.1, assumes the presence of a *sketch* – a partial program with holes to be filled with subexpressions by the solver. Typically, these

```
1  (define-sort Bit5 () (_ BitVec 5))
2  (define-fun input-count () Int 1)
3  (define-fun component-count () Int 2)
4  (define-fun line-count () Int (+ input-count component-count))

6  ; Component semantics
7  (define-fun comp1 ((x1 Bit5) (x2 Bit5)) Bit5 (bvand x1 x2))
8  (define-fun comp2 ((x Bit5)) Bit5 (bvsub x #b00001))

10 ; i_jk – k^th input of component f_j; o_j – its output;
11 (declare-const i11 Bit5) (declare-const i12 Bit5) (declare-const i21
       Bit5) (declare-const o1 Bit5) (declare-const o2 Bit5)
12 ; li_jk and lo_j are the corresponding line numbers
13 (declare-const li11 Int) (declare-const li12 Int) (declare-const li21
       Int) (declare-const lo1 Int) (declare-const lo2 Int)
14 ; Variables and line numbers for the program input i_0 and output o
15 (declare-const i0 Bit5) (declare-const li0 Int)
16    (declare-const o Bit5) (declare-const lo Int)
17 (assert (and (= li0 0) (= lo (- line-count 1))))

19 ; Well-formedness constraints
20 (assert (and (>= li11 0) (< li11 line-count)
21              (>= li12 0) (< li12 line-count)
22              (>= li21 0) (< li21 line-count)
23              (>= lo1 input-count) (< lo1 line-count)
24              (>= lo2 input-count) (< lo2 line-count)))
25 (assert (not (= lo1 lo2)))
26 (assert (and (< li11 lo1) (< li12 lo1) (< li21 lo2))) ; Acyclicity
27 (assert (and (= o1 (comp1 i11 i12)) (= o2 (comp2 i21)))) ; Components

29 ; Data flow constraints: equal locations should hold equal variables
30 (assert (and (=> (= lo2 lo) (= o2 o))
31   (=> (= li0 li11) (= i0 i11)) (=> (= li0 li12) (= i0 i12))
32   (=> (= li0 li21) (= i0 i21)) (=> (= li0 lo1) (= i0 o1))
33   (=> (= li0 lo2) (= i0 o2))  (=> (= li0 lo) (= i0 o))
34   (=> (= li11 li12) (= i11 i12)) (=> (= li11 li21) (= i11 i21))
35   (=> (= li11 lo1) (= i11 o1)) (=> (= li11 lo2) (= i11 o2))
36   (=> (= li11 lo) (= i11 o))    (=> (= li12 li21) (= i12 i21))
37   (=> (= li12 lo1) (= i12 o1)) (=> (= li12 lo2) (= i12 o2))
38   (=> (= li12 lo) (= i12 o))    (=> (= li21 lo1) (= i21 o1))
39   (=> (= li21 lo2) (= i21 o2)) (=> (= li21 lo) (= i21 o))
40   (=> (= lo1 lo2) (= o1 o2))  (=> (= lo1 lo) (= o1 o))))

42 (assert (and (= i0 #b01010) (= o #b01000))) ; Input-output example
```

Figure 5.1: A program synthesis constraint that encodes the problem in Example 5.1.

sketches are written by a human, but it is also possible to generate them automatically.

Recently, Feng et al. developed SyPet, a technique for component-based synthesis that **(a)** does not require the component library to be annotated with specifications, and **(b)** deliberately decomposes the synthesis process into sketch generation and sketch completion to keep the analysis tractable [31]. In this approach, the first phase performs reachability analysis on the component APIs (represented as a *Petri net* [119]). Every found reachable path corresponds to a viable combination of components in the library as a sketch. The second phase then completes this sketch with program parameters and variables using an SMT encoding.

The separation of component-based synthesis into two phases that are resolved by different techniques is a key idea that makes SyPet scale. While it is possible to construct a Petri net that represents the entire synthesis problem, that net is much larger, and analyzing its reachable paths quickly becomes intractable. We also note that the choice of Petri nets as a technology for sketch generation is specific to SyPet; a different application might warrant a different approach.

5.2 Solver-Aided Programming

Developing a problem-specific SAT/SMT encoding and generating the corresponding formula can be cumbersome and time-consuming. For this reason, the community built several *frameworks* for expressing synthesis problems (and some related ones) that translate a representation in a high-level programming language into the corresponding formula behind the scenes. More generally, such frameworks implement the idea of *solver-aided programming*: they augment high-level programming languages with new constructs that require constraint solving to be materialized [139]. Such constructs enable embedding second-level subproblems into regular programs in the *host language*.

In §3.3.1, we described sketching—a form of syntactic bias that describes the program space for synthesis as a program in a high-level C-like programming language augmented with *symbolic holes* and *gener-*

ators. SKETCH is the first framework for solver-aided program synthesis: its compiler translates the sketches into equivalent SAT formulas such that their satisfying models can be mapped to the corresponding choices for the holes and generator traces [132]. Thus, a sketch can be seen as a program in a high-level programming language whose compiler relies on a SAT solver to materialize some language constructs.

In 2013, Torlak and Bodik [139] and, independently, Uhler and Dave [142] extended the ideas of SKETCH to more general forms of solver-aided programming. Torlak and Bodik developed ROSETTE: a symbolic framework that integrates into the virtual machine of Racket[3] and extends the language with new solver-aided constructs. SMTEN implements a similar idea in the Haskell host language.

Apart from program synthesis, solver-aided constructs enable embedding *verification*, *fault localization*, and *angelic execution* queries into the programs. For instance, ROSETTE extends Racket with the following solver-aided constructs:

- `(`**`define-symbolic`** ⟨id⟩$_1$ ... ⟨id⟩$_k$ ⟨type⟩`)`
  Defines k *symbolic holes* that may be in Racket computations as regular variables. The ROSETTE compiler interprets an expression with such a hole *symbolically*, representing it as an AST. The parts without any holes are evaluated by the Racket VM as usual.
- `(`**`verify`** ⟨expr⟩`)`
  A *verification* query. Attempts to find any binding of the symbolic variables in the expression that violates the assertions in scope.
- `(`**`solve`** ⟨expr⟩`)`
  An *angelic execution* query. Attempts to find a binding of the symbolic variables in the expression that satisfies the assertions in scope.
- `(`**`synthesize #:forall`** ⟨inputs⟩ **`#:guarantee`** ⟨expr⟩`)`
  A *synthesis query*. Finds a binding of the symbolic variables in the expression that are not universally quantified in the `inputs`, such that they satisfy the assertions in scope. In addition to explicitly defined holes, these variables are also generated by

[3] `http://racket-lang.org`

grammars, which are defined similarly to generators in SKETCH. Applying `generate-forms` to the binding produces a syntactic representation of the synthesized expression in that grammar.

- `(`**`debug`** `[`⟨`filter`⟩`]` ⟨`failing-expr`⟩`)`
  A *debugging* or *fault localization* query. Produces a *minimal unsatisfiable core* of the expressions specified by the filter—that is, some subset of expressions that is guaranteed to be relevant to the given assertion failure. In other words, the assertion cannot be satisfied without changing at least one expression from the core.

Example 5.2 (Re-used from [140])**.** Consider a domain of finite state automata (FSAs). As shown in [74], it is possible to use Racket's metaprogramming facilities (i.e., macros) to define a fast and concise FSA DSL with executable semantics. In this DSL, an example definition of an FSA that accepts the language $c(ad)^*r$ looks as follows:

```
(define m (automaton init
           [init : (c → more)]
           [more : (a → more) (d → more) (r → end)]
           [end  : ]))
```

where `automaton` is a Racket macro that accepts a list of states and transitions, and returns an executable FSA.

Using ROSETTE's symbolic evaluation facilities, we can use solver-aided queries to verify FSA behavior, synthesize FSAs, or angelically execute them in search for an accepted input. We first define some utility functions to create and manipulate symbolic words:

```
; Draws a word of length k from the given alphabet.
(define (word k alphabet)
   (for/list ([i k])
      (define-symbolic* idx integer?)
      (list-ref alphabet idx)))

; Draws a word of length 0 ≤ n ≤ k from the given alphabet.
(define (word* k alphabet)
   (define-symbolic* n integer?)
   (take (word k alphabet) n))

; Returns a string encoding of the given list of symbols w.
```

```
13 ; For example, (word->str '(c a r)) returns "car".
14 (define (word->str w) (apply str-append (map symbol->str w)))

16 ; Returns true iff regex matches the string encoding of w.
17 (define (spec regex w) (regexp-match? regex (word->str w)))
```

and then construct the desired queries:

```
1  ; Angelic execution: find a word of length ≤ 4 accepted by m.
2  (define w (word* 4 '(c a d r)))
3  (define model (solve (assert (m w))))
4  (evaluate w model) ; '()
5  ; Indicates that the empty word '() is accepted by m.

7  ; Verification: find a counterexample to a given regex that m accepts.
8  (verify (assert (eq? (spec #px"^c[ad]*r$" w) (m w))))
9   verify: no counterexample found

11 ; Synthesis: find an FSA that implements the regex ^c[ad]+r$ for all
       words of length ≤ 4 using an FSA sketch M [140, Figure 3].
12 (define model
13    (synthesize #:forall w
14       (assert (eq? (spec #px"^c[ad]+r$" w) (M w)))))
15 (generate-forms model)
16 ; (define M
17 ;   (automaton init
18 ;     [init : (c → s1)]
19 ;     [s1 : (a → s2) (d → s2) (r → reject)]
20 ;     [s2 : (a → s2) (d → s2) (r → end)]
21 ;     [end : ]))
```

Implementation Internally, Rosette implements these constructs by embedding a *symbolic evaluator* into the Racket's VM. It follows the standard evaluation strategy, tracking the symbolic expressions as ASTs and evaluating the concrete ones using Racket semantics. Most importantly, Rosette's VM implements a clever technique for merging symbolic expressions produced by different conditional branches in the program's control flow [140]. Without such merging, the number of symbolic subexpressions and the resulting formula would grow exponentially, overlooking many opportunities for concrete evaluation.

The result of symbolic evaluation for each solver-aided construct is an SMT query to the underlying solver. Its solution (either a model or an unsatisfiable core) is translated by the VM back into the host language (Racket or Haskell) in terms of the original program.

In addition to simplifying design of problem encodings, solver-aided frameworks also enable *domain separation*: an application designer can focus on their DSL design without intertwining it with the encoding details of solver-aided queries. As a result, these frameworks are widely successful. Rosette, for instance, is used in numerous research projects on program synthesis and verification (*e.g.* [14, 107, 108, 147]).

5.3 Inductive Logic Programming

Inductive logic programming (ILP) is the field of automatic inference of logical programs from examples [92]. In contrast to other synthesis approaches in this survey, ILP is focused on synthesizing first-order rules and relations instead of functional expressions or programs. However, it is similar to constraint-based synthesis in its methodology—applying logical inference to derive the desired relation. Since the main specification kind in ILP is a set of examples, ILP is also closely related to *programming by examples*, discussed in Chapter 7.

A problem in ILP is given by *examples* (a set of facts) and, optionally, *background knowledge* (a set of rules that describe the domain). For instance, given the following examples:

parent(jack, mary).	father(jack, mary).	male(jack). male(bob).
parent(mary, bob).	mother(mary, bob).	female(mary).

an ILP system is able to infer first-order rules and constraints such as:

$$\leftarrow \mathsf{male}(X), \mathsf{female}(X).$$
$$\mathsf{male}(X) \leftarrow \mathsf{father}(X, Y).$$
$$\mathsf{father}(X, Y), \mathsf{mother}(X, Y) \leftarrow \mathsf{parent}(X, Y).$$

A specification in ILP is a target predicate used in the examples whose definition is to be inferred. It is also not uncommon to omit it and let the system infer any interesting facts and rules from the provided

data. In both capacities ILP has been successfully applied to numerous domains, including biological discovery [69, 95, 122, 136], ecology [138], and natural language processing [83, 150].

Modern ILP algorithms perform various forms of *hypothesis search*, employing a combination of inductive and deductive logical inference. Specific search techniques such as *inverse resolution* [94] and *inverse entailment* [93] are beyond the scope of this survey. We refer the reader to ILP surveys [27, 92] for a comprehensive introduction.

The most notable recent development in ILP is *meta-interpretive learning* (MIL) [96, 97]. It extends classic generalization techniques with an efficient implementation of *predicate invention*: automatic introduction of useful sub-predicates in the derivation. This allows MIL to learn complex logical programs including recursive predicate definitions and clauses that are hierarchically constructed from re-usable blocks.

6

Stochastic Search

The stochastic synthesis approaches learn a distribution over the space of programs in the hypothesis space that is conditioned on the specification, and then sample programs from the distribution to learn a consistent program. The goal of these techniques is to use the learnt distribution to guide the search of programs that are more likely to lead to programs that conform with the specification. Some common approaches to learn program distributions include Metropolis-Hastings algorithm, genetic programming, and machine learning. We now briefly describe the main ideas of these approaches.

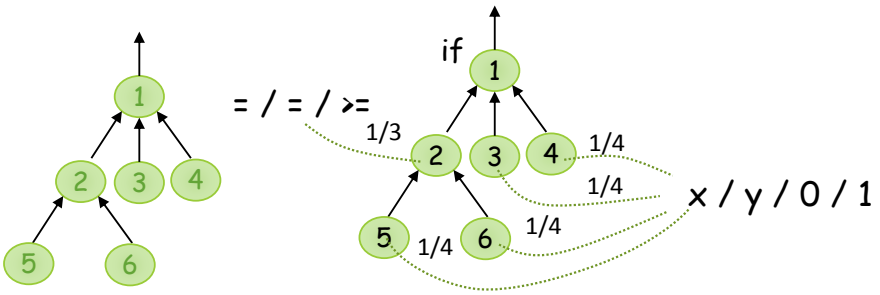
6.1 Metropolis-Hastings Algorithm for Sampling Expressions

The stochastic SyGuS solver [3] uses Metropolis-Hastings algorithm to sample expressions from a given grammar to learn an expression that is consistent with a specification. This solver was inspired from the work on using Markov Chain Monte Carlo (MCMC) sampling based techniques for super-optimizing the loop-free binary assembly code [124]. The key idea of the search algorithm is to first define a `Score` function that assign a cost to every program in the hypothesis space of all

possible programs, which defines a probability distribution over the domain of programs. Since the cost function aims to model a highly irregular and high-dimensional search space, it is typically complex and non-continuous, which makes it difficult for direct sampling techniques to sample programs from this distribution. Therefore, the Metropolis-Hastings algorithm is used to sample desired programs.

The space of all possible programs (of some fixed length) in the hypothesis space can be represented using a large dense graph, where the nodes represent different partial expressions and an edge from node n_1 (corresponding to expression e_1) to node n_2 (corresponding to expression e_2) denote the single-edit transformation to obtain expression e_2 from e_1. The Metropolis-Hastings algorithm performs a probabilistic walk on the graph starting from a random node to reach the desired node that satisfies the specification. Let `Score(e)` denote the cost function associated with a node n denoting the expression e, which captures the notion of the extent to which e meets the specification ϕ. Having the probability distribution P assign probability $P(e) \propto \texttt{Score}(e)$, we increase the chances of the algorithm to reaching the desired expression with the best score. The algorithm performs the probabilistic walk in the direction of increasing expression scores. However, since the cost function is non-continuous, the algorithm can also move to expressions in the graph with a lower score, with some low probability.

The stochastic SyGuS solver defines the score of an expression e as $\texttt{Score}(e) = e^{-\beta C(e)}$, where β is a constant set to 0.5 and $C(e)$ denotes the number of examples for which e is incorrect. The value of $\texttt{Score}(e)$ is large when $C(e)$ is small denoting that the expression e is correct on most of the specification examples. When $C(e) = 0$, the expression e is the desired expression. Consider the size of the programs is fixed to a constant k and the graph consists of all programs of size k in the hypothesis space. The solver picks the first expression e randomly in the graph. It then randomly selects a node ν in the parse tree of e. Let e_ν denote the sub-expression that is rooted at this node ν. The algorithm then uniformly chooses another expression e'_ν of size equal to that of e_ν from its neighborhood. Let e' denote the new expression obtained by replacing the subexpression e_ν in e



(a) (b)

Figure 6.1: (a) The shape of all expressions of size 6 in the grammar shown in Figure 4.1, (b) The likelihood of assigning different terminal symbols to each node in the parse trees of size 6.

with e'_ν. Given the original expression e and the mutated expression e', the Metropolis-Hastings algorithm chooses to make the mutated expression the new candidate expression using the following acceptance ratio: $\alpha(e, e') = \texttt{min}(1, \texttt{Score}(e)/\texttt{Score}(e'))$. Intuitively, the algorithm accepts the mutation if the new expression results in higher score value (i.e. $\alpha(e, e') = 1$). Otherwise, it chooses to accept the new expression with probability equal to $\alpha(e, e') < 1$.

Give a specification ϕ, the size of the desired expression is typically not known apriori. The stochastic solver starts with the program size $k = 1$ and using some probability p_m searches for programs of size $k + 1$ after each iteration of the Metropolis-Hastings algorithm.

Consider the conditional linear integer arithmetic grammar in Figure 4.1 (with additional leaf terminals denoting constants 0 and 1) and the specification of the unknown function f to be synthesized to be that of the maximum function such that $\phi : f(x, y) \geq x \wedge f(x, y) \geq y \wedge (f(x, y) = x \vee f(x, y) = y)$. Let's assume the stochastic solver is currently searching for expressions of size $k = 6$. The shape of the parse trees of all expressions of size 6 in the grammar is shown in Figure 6.1(a). There are in total 768 such expressions. The possible values of terminals for different nodes in the parse tree is shown in Figure 6.1(b).

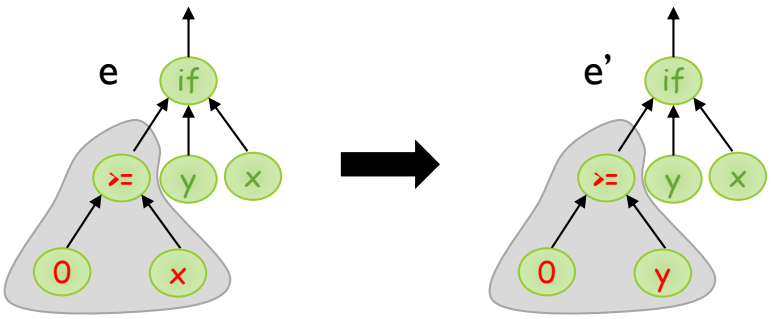


Figure 6.2: An example mutation considered by the stochastic solver.

Let us assume the solver chose the expression $e = if(x \leq 0, y, x)$ as the current candidate expression (whose likelihood is $1/768$), and the solver selects the Boolean predicate $x \leq 0$ to mutate. The example mutation is shown in Figure 6.2. Since there are 48 different predicate choices, the probability to mutate the predicate $x \leq 0$ to $y \leq 0$ is $1/6 \times 1/48 = 1/288$ and $e' = if(y \leq 0, y, x)$. For a given set of input-output examples $\{(-1, 4), (-3, -1), (-1, -2), (1, 2), (3, 1), (6, 2)\}$, the scores of the expressions are $\texttt{Score}(e) = e^{-2\beta}$ and $\texttt{Score}(e') = e^{-3\beta}$. The solver then choose to mutate expression e to e' with probability equal to $e^{-\beta}$. Note that if the algorithm had considered the mutation to obtain the expression $e'' = if(x \leq y, y, x)$, then the score would have been $\texttt{Score}(e'') = 1$, and e'' would be the new candidate expression chosen with probability 1 by the algorithm.

6.2 Genetic Programming

Genetic programming [73] can be considered as an extension of genetic algorithms [56] to the domain of computer programs, where a population of programs are continuously evolved and transformed into new

generations of programs using principles inspired from biological evolution such as Darwin's theory of natural selection and natural genetic operations including crossover, mutation, duplication, and deletion [71]. Compared to the traditional genetic algorithms, the evolving structures in genetic programming are hierarchical computer programs of dynamically varying shape and size, where a set of functions and terminal symbols define the complete hypothesis space of possible programs. The genetic programming algorithms maintain a population of individual programs and use the mutation and crossover operations to generate program variants. The mutation operation performs random changes to programs, whereas the crossover operation allows for reusing useful subprograms across different programs. The fitness function corresponds to the notion of how well a program satisfies the specification and is used to suitability of different program variants. A program that meets the specification during the evolution is then returned as the desired one.

The Genetic programming algorithm starts with a set of terminals and functions that are appropriate for a given domain. This set defines the complete search space of programs considered by the algorithm. There are two key requirements of the hypothesis space: i) the desired program to be synthesized exists in the search space, and ii) each function should be able to accept as its argument a value returned by any other function or a terminal value. The first requirement ensures existence of the desired program, whereas the second requirement ensures the well-formedness of the programs obtained by mutations and crossover during the evolution of the programs.

There are four main steps in designing the genetic programming algorithm. First, a set of terminal and function symbols need to be defined. Second, a fitness measure needs to be defined that corresponds to the viability of a program, i.e. how well the program satisfies the specification. Third, we need to define a number of search parameters such as the size of population, number of expressions in a program, probability of crossover, mutation, reproduction, deletion, etc. These parameters guide the evolutionary process. Finally, we need to determine the termination criterion for ending the evolutionary process and returning the resulting program.

The genetic algorithm first creates a random population of programs, where each individual program in the population are generated randomly using the terminal and function symbols. Each program in the population is syntactically correct because of the requirement on the set of terminal and function symbols. The fitness of each program in the population is then measured. The fitness measures are typically domain-specific and problem dependent. Some examples of fitness measures include the number of input-output examples being correct, the deviation between the program output and desired outputs, properties based on program execution (such as time, energy etc.), or a multi-objective cost function combining multiple criterion. The initial generation consisting of random programs generally has very poor fitness, but some individuals have a higher fitness score than others. This difference between fitness scores is used for performing crossover and mutation operations.

The crossover operation randomly selects two programs from the population (called parents) based on their fitness scores. It then randomly selects a node (in the corresponding parse trees) independently from each of the two programs and swaps them. We now obtain two offspring programs. This operation is repeated multiple times to obtain a set of new offspring programs. An example crossover operation is shown in Figure 6.3. The mutation operation randomly selects a program based on the fitness score and selects a node. It then deletes the sub-tree rooted at the selected node and grows a new sub-tree randomly using the terminal and function symbols. An example mutation operation is shown in Figure 6.4. The reproduction operation randomly selects a program based on the fitness scores and copies it to the new population.

The genetic programming algorithm iteratively uses these operations to evolve population of programs until the termination criterion is satisfied. After termination, the single best program produced during the evolution based on the fitness score is harvested and returned as the result. If the returned program corresponds to the desired program, the run of the genetic algorithm is designated as a successful run.

The success of genetic programming based systems crucially depends on the design of a good fitness function, which require non-trivial

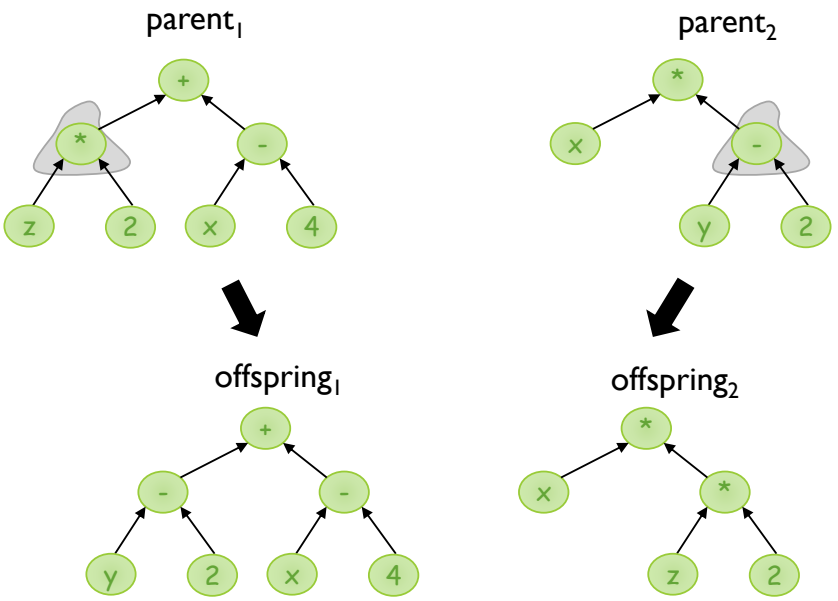


Figure 6.3: An example crossover operation. The algorithm randomly selects two nodes in two parent programs and swaps them to generate two new offspring programs.

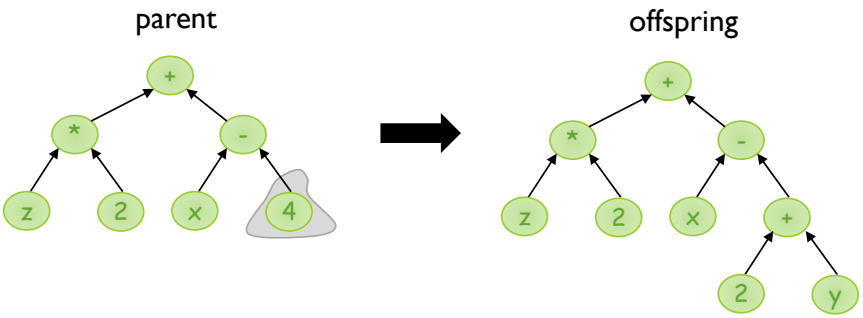


Figure 6.4: An example mutation operation. The algorithm randomly selects a node in a parent program and replaces the sub-tree rooted at it with a new randomly generated expression.

insights and creativity. Genetic programming has been used to discover new mutual exclusion algorithms [68] and to fix bugs in imperative programs [80]. There have also been some interesting recent work in combining genetic programming techniques with logical reasoning techniques [61, 68].

6.3 Machine Learning

There have been some machine learning based techniques developed to guide the search process for learning a correct program from the hypothesis space [82, 89]. The main idea of these techniques is to learn a probability distribution over the space of programs that is conditioned by the specification (such as input-output examples). This probability distribution is then used to sample programs that are more likely to be consistent with the given specification.

Menon et al. [89] propose a machine learning framework for Programming By Example, where the hypothesis space is defined by a context-free grammar (CFG), and given a set of input-output examples, the system learns weights for different rules (called *clues*) in the grammar to obtain a corresponding probabilistic context-free grammar (PCFG). The PCFG is then used to guide the enumerative search to find the desired program. The weights of different grammar rules conditioned on the input-output examples are trained using a training corpus of input-output examples and the corresponding programs.

The framework is instantiated on a text processing language, which adds text functions on top of the FlashFill DSL [43]. The domain expert then manually defines certain features that can help the learning algorithm identify correct programs given the input-output examples. Some features include `dedup_cue` that checks whether any string in the set of output strings is duplicated, `sort_cue` that checks if the output strings are sorted etc. The learning process exploits the fact that the chance of a rule being used for a desired program for a set of input-output pairs $(\bar{x}, \bar{y})$ greatly depends on certain characteristics in the structure of $\bar{x}$ and $\bar{x}$, namely the features. This approach results in an order of magnitude better performance over an enumerative baseline.

A drawback of this framework is that designing useful clues (features) manually is a time consuming and difficult task, and requires a lot of domain expertise. Moreover, since the learnt weight results in a PCFG, the framework can not use contextual information about the partial trees generated during the search to more efficiently guide the exploration.

6.4 Neural Program Synthesis

The problem of program synthesis and program induction has seen a lot of recent interest from the deep learning community. The proposed approaches can be divided into two categories: i) program induction, and ii) program synthesis. The neural architectures that perform program induction learn a network that is capable of replicating the behavior of the desired program, whereas the neural systems that perform program synthesis return an interpretable program that matches the desired specification behavior.

The program induction techniques develop new neural architectures that can learn the behavior of a program that is consistent with a given set of input-output examples. Many of these approaches are inspired from computation modules such as CPU [40, 118], GPU [66], and data structures such as stacks [64]. The key idea in these approaches is to develop a continuous representation of the atomic operations of the network, and then use either end-to-end training of a neural controller or reinforcement learning to learn the program behavior. Even though this work on program induction has led to several promising neural architectures, there are also some shortcomings. One shortcoming is that these techniques do not generate an interpretable model of the learnt program, and typically require large computational resources and several thousands of input-output examples per synthesis task.

Some recently proposed neural systems do learn interpretable programs. The Neural FlashFill [105] system develops two neural architectures: 1) the cross correlation I/O encoder that produces a continuous representation of the set of input-output examples, and ii) the Recursive-Reverse-Recursive Neural Network (R3NN) that incrementally synthesizes a program in the DSL given the continuous representation of the

examples. Neural-RAM [76] learns a circuit composed of a given set of modules that is consistent with a set of input-output examples. It first develops a continuous representation of all modules, and then learns a controller that defines how the different modules are connected with each other. DeepCoder [7] learns embeddings of integer input-output examples to learn a distribution over the possible space of functions that are likely to be useful for the desired transformation on the examples. It then uses the learnt distribution over functions to guide a depth-first based top-down enumerative algorithm similar to the one shown in §4.1.1. Finally, there are also some recent probabilistic programming languages such as Terpret [36] and Forth [120] that enable programmers to write partial programs and use gradient descent based techniques to complete them such that the completed programs satisfy the input-output examples.

Given the FlashFill DSL, the Neural FlashFill [105] system learns a generative model of programs in the DSL that is conditioned on the input-output examples. The generative model is trained end-to-end using a large training set of programs in the DSL together with their corresponding input-output examples. The training set is generate automatically by sampling programs from the DSL and then using a rule-based approach to generate conforming input-output examples. The system develops two key neural architectures: i) The I/O encoder and ii) the R3NN network. Starting from the start symbol of the grammar, the system uses the R3NN to incrementally construct the desired program. The R3NN takes a partial program tree (derivation in the grammar), and decides which non-terminal node in the tree to expand and with which expansion rule. The generative process is conditioned by the I/O encoding vector.

The I/O encoder generates a continuous representation of the set of input-output examples. The cross correlation encoder first runs bidirectional LSTMs over the input and output strings, and then computes the cross correlation between the two output tensors obtained from the LSTMs. The key idea insight is to learn which parts of output strings come from which parts of the input strings.

The R3NN learns a generative model over the trees in the DSL. The model has the following 4 parameters: i) a vector representation for each symbol in the grammar, ii) a vector representation for each rule in the grammar, iii) a deep neural network that takes as input the set of RHS symbols of a rule and generates a representation of the corresponding LHS symbol, and iv) a deep neural network that as input the representation of an LHS symbol of a rule and generates a representation for each of the corresponding RHS symbols. The R3NN first assigns a vector representation to each leaf node of a partial tree, and then performs a recursive pass going up in the tree to assign a global representation to the root. It then performs a reverse-recursive pass from the root to assign a global representation to each node in the tree. Intuitively, the idea is to assign a representation to each node in the tree such that the node knows about every other node in the tree. The R3NN encoding for an example partial tree in the grammar with the recursive and reverse-recursive passes is shown in Figure 6.5 and Figure 6.6 respectively.

The Neural FlashFill system after being trained on synthetic programs of size upto 13 is able to successfully synthesize both programs (upto size 13) that it has seen during the training but with different input-output examples, and programs that it has not seen during the training. The 1-best strategy (choosing best expansion from the distribution at each expansion) yields an accuracy of about 60% whereas it increases to 97% with 300-samples. Moreover, it is also able to learn desired programs for 91 (38%) of 238 real-world FlashFill benchmarks. A majority of the unsolved FlashFill benchmarks belong to the category of large programs. One shortcoming of the R3NN network is that it is currently challenging to train for programs of larger size.

There has also been some work on synthesizing programs using quantitative objectives [17, 18, 19]. The idea of program smoothing [19] is to reduce the synthesis problem to a sequence of numerical optimization problems, where the program is approximated in some continuous space. The approximate objective becomes closer to the original quantitative objective as the sequence progresses. The idea of combining

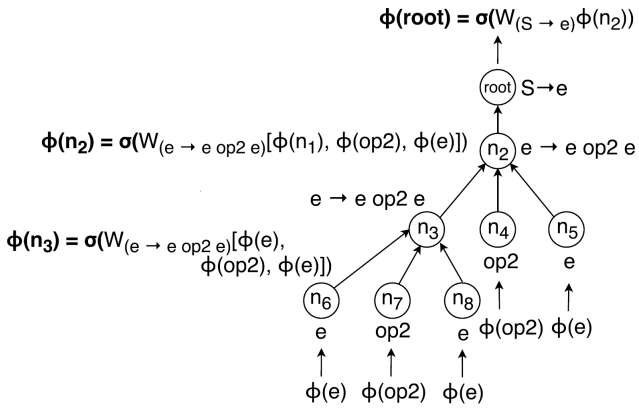


Figure 6.5: The initial recursive pass of the R3NN.

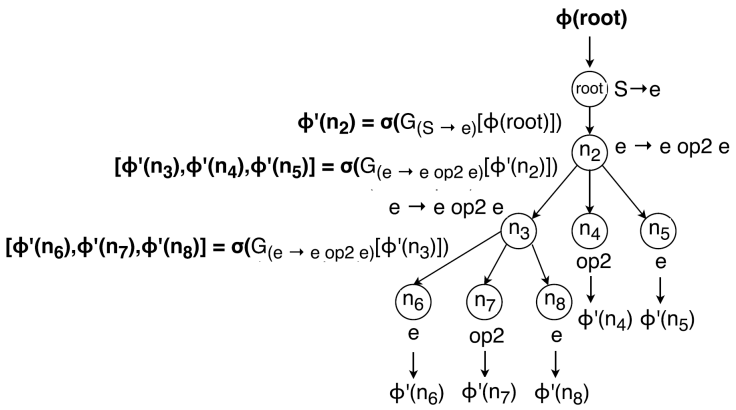


Figure 6.6: The reverse-recursive pass of the R3NN.

program smoothing with neural program synthesis techniques can be quite beneficial for the synthesis tasks with quantitative objectives.

7

Programming by Examples

7.1 Problem Definition

As described in §1.3, the problem of program synthesis is largely defined by the choice of its three main components: intent specification, program space, and search algorithm. When dealing with a novel problem, we can usually control the choice of the algorithm and, to some extent, the program space. However, the particular specification kind is typically a fundamental part of the real-life problem. Since the available specification kind naturally limits the available choice for the other two problem components, it is not surprising that limits in intent specification inspire the most innovations in subfields of synthesis. The subfield of *programming by example* (PBE) is one such instance.

In PBE, the synthesis problem is given by *input-output examples* (or, more generally, input-output constraints). They specify the behavior of the desired program on a subset of its valid inputs. In the simplest PBE scenario, the specification defines the program's outputs; in more complex cases, it specifies some *properties* or *constraints* on the outputs instead of their precise values. Examples are generally preferred thanks to their ease of use for a user specifying the problem, but sufficiently simple constraints on the outputs may be used for the same purpose.

Example 7.1. The following set of input-output examples specifies the behavior of the program $\sqrt{x} : \mathbb{R}^+ \to \mathbb{R}$:

$$\{1 \mapsto 1,\ 9 \mapsto 3,\ -3 \mapsto \bot\}$$

Example 7.2. The following set of input-output examples specifies the behavior of a Flash Fill program (see §2.1) that extracts the first word from a given input string:

Input	**Output**
`Alice Smith`	`Alice`
`Benjamin H. Baker`	`Benjamin`

Example 7.3. The following set of input-output constraints specify the behavior of a program that filters valid numbers out of a given list of strings. Each constraint specifies an input list and a *subset* of the corresponding output list. Formally, an output constraint of kind "**?** $\sqsupseteq [e_1, \ldots, e_n]$" requires that the program's output ("**?**") must contain $[e_1, \ldots, e_n]$ as a sublist. Such a constraint is usually preferable to full input-output examples when the correct output is too large to specify.

Input	**Output**
["`Alice`", "`50`", "`North America`", "`-34`", "`0`", ...]	**?** $\sqsupseteq$ ["`50`", "`0`"]
["`Benjamin`", ...]	**?** $\sqsupseteq$ []

Note how the first input-output constraint omits "`-34`" even though it should also be present in the desired output (because the desired program must filter all valid numbers from the given input list).

Input-output examples exhibit several unique properties, which set the PBE subfield apart from the rest of program synthesis.

Ease of use: Examples are very easy to provide, explain, and verify (as opposed to other widespread spec kinds such as logical formulas and assertions). This makes them an ideal specification kind for non-programming application domains such as data wrangling in Excel or direct graphical manipulation.

Ambiguity: Examples are an *under-specification*: most of the time, there exists more than one program that is consistent with the given set of examples. Moreover, in typical real-life languages, the space of consistent programs is either infinite or extremely large (e.g. up to 10^{20} in wrangling domains [113]). This inherent ambiguity in problem specification constitutes an additional challenge in PBE: we need to find not just *some* program that is consistent with the spec but the *intended* one (or semantically equivalent).

In the rest of this section, we focus on the most popular methods for dealing with challenges of PBE: (a) finding consistent programs, (b) handling an enormous space of consistent program candidates, and (c) disambiguating user intent in order to pick a single program candidate.

7.2 Version Space Algebra

If a synthesis algorithm aims to satisfy an *underlying* user intent in an example-based spec (as opposed to finding any single program that is consistent with it), it must somehow deal with enormous spaces of ambiguous program candidates. One popular tool to resolve that is using a data structure called *version space algebra* (VSA) [77, 91, 113]. It allows to represent potentially exponential program sets in polynomial space and perform various operations on these sets in polynomial time.

Definition 1 (Version space algebra)**.** Let N be a symbol in a DSL $\mathcal{L}$. A *version space algebra* is a representation for a set $\widetilde{N}$ of programs rooted at N. The grammar of VSAs is:

$$\widetilde{N} \;:=\; \{P_1, \ldots, P_k\} \mid \mathbf{U}(\widetilde{N}_1, \ldots, \widetilde{N}_k) \mid F_{\bowtie}(\widetilde{N}_1, \ldots, \widetilde{N}_k)$$

where F is any k-ary operator in $\mathcal{L}$, and P_j are some programs in $\mathcal{L}$. The semantics of VSA as a set of programs is given as follows:

$$\begin{aligned}
&P \in \{P_1, \ldots P_k\} && \text{if } \exists\, j\colon P = P_j \\
&P \in \mathbf{U}(\widetilde{N}_1, \ldots, \widetilde{N}_k) && \text{if } \exists\, j\colon P \in \widetilde{N}_j \\
&P \in F_{\bowtie}(\widetilde{N}_1, \ldots, \widetilde{N}_k) && \text{if } P = F(P_1, \ldots, P_k) \wedge \forall j\colon P_j \in \widetilde{N}_j
\end{aligned}$$

Intuitively, a VSA is a DAG where each node represents a set of programs. *Leaf nodes* contain explicit enumerations of programs; they

are composed into larger sets by two possible VSA constructor nodes. *Union nodes* $\mathbf{U}$ represent a set union of their constituent VSAs. *Join nodes* $F_{\bowtie}$ represent a cross-product of their constituent VSAs, with an associated operator F applied to all combinations of parameter programs from the cross-product.

One can also interpret VSA as a sub-language of its original DSL $\mathcal{L}$. If we associate each VSA leaf node with a fresh terminal symbol, and each constructor node with a fresh nonterminal, we obtain a context-free grammar $\mathcal{L}' \subset \mathcal{L}$ where join nodes correspond to operator calls and union nodes correspond to alternations:

Example 7.4. Consider the following DSL $\mathcal{L}$, which models a subset of Flash Fill:

$$\begin{aligned}
\texttt{string}\ N &\coloneqq \mathsf{ConstantString}(s) \mid \mathsf{Substring}(P,\ P);\\
\texttt{int}\ P &\coloneqq \mathsf{AbsolutePosition}(k);\\
\texttt{int}\ k;&\quad\ \texttt{string}\ s;
\end{aligned}$$

It contains two nonterminals N and P, two terminals k and s, and references three operators ConstantString, Substring, and AbsolutePosition. Every program rooted at N in this DSL outputs a string–either a constant string s or a substring of a given input defined by two positions P. Each position, in turn, is defined by an integer k, which represents an absolute position in a string, counted from the left if $k \geq 0$ and from the right if $k < 0$.

The following VSA $\widetilde{N}$ represents $2 + 2 \times (1 + 2) = 8$ programs in $\mathcal{L}$:

$$\begin{aligned}
\widetilde{N} = \mathbf{U}\,(&\mathsf{ConstantString}_{\bowtie}(\{\text{“}\texttt{foo}\text{”}, \text{“}\texttt{bar}\text{”}\}),\\
&\mathsf{Substring}_{\bowtie}(\mathsf{AbsolutePosition}_{\bowtie}(\{0, 1\},\\
&\quad\mathbf{U}\,(\mathsf{AbsolutePosition}_{\bowtie}(\{10\}),\ \mathsf{AbsolutePosition}_{\bowtie}(\{-1, -5\})))))
\end{aligned}$$

It is equivalent to the following context-free grammar $\mathcal{L}' \subset \mathcal{L}$:

$$\begin{aligned}
\texttt{string}\ N' &\coloneqq \mathsf{ConstantString}(s') \mid \mathsf{Substring}(P'_1,\ P'_2);\\
\texttt{int}\ P'_1 &\coloneqq \mathsf{AbsolutePosition}(k'_1);\\
\texttt{int}\ P'_2 &\coloneqq Q'_1 \mid Q'_2;\\
\texttt{int}\ Q'_1 &\coloneqq \mathsf{AbsolutePosition}(k'_2);
\end{aligned}$$

$$\begin{aligned} \texttt{int } Q'_2 &\coloneqq \mathsf{AbsolutePosition}(k'_3); \\ \texttt{string } s' &\in \{\text{“}\texttt{foo}\text{”}, \text{“}\texttt{bar}\text{”}\}; \\ \texttt{int } k'_1 &\in \{0,1\}, \quad k'_2 \in \{10\}, \quad k'_3 \in \{-1,-5\} \end{aligned}$$

The key property of VSAs is their ability to encode exponential sets of programs in polynomial space. They achieve that by providing two kinds of sharing among program subexpressions. One is provided by the join nodes, which encode a cross-product of their subexpression sets. The other is provided by the DAG structure of a VSA, which allows subexpression sharing among program sets that reference the same VSA node through different paths in the DAG.

Another powerful property of VSAs is ability to quickly perform various set-theoretic operations on them. These operations are usually defined inductively over the VSA structure. Thus, their running time is proportional to the *number of nodes* in the VSA, as opposed to the *number of programs* it represents (which is usually exponentially greater). We refer the reader to [113] for the implementation details of all such operations; in this overview, we simply present a list of them:

- VSA *intersection* $\widetilde{N}_1 \cap \widetilde{N}_2$ constructs a VSA that represents all programs present in both $\widetilde{N}_1$ and $\widetilde{N}_2$.
- VSA *clustering* $\widetilde{N}/_{\sigma}$ partitions a VSA $\widetilde{N}$ into non-intersecting sub-VSAs based on the output of their contained programs on a given input σ.
- VSA *ranking* $\mathsf{Top}_h(\widetilde{N}, k)$ finds the topmost k programs in a VSA $\widetilde{N}$ with respect to a ranking function $h\colon \widetilde{N} \to \mathbb{R}$.
- VSA *projection*, or *filtering* $\widetilde{N}\restriction_{\phi}$ eliminates all programs in $\widetilde{N}$ that do not satisfy a given spec ϕ.

The synthesis algorithms in §7.3.1 make use of these operations.

7.3 Deduction-Based Techniques

Although any of the previously discussed synthesis algorithms can be applied to a PBE problem, certain specialized algorithms usually perform

much better. They are based on the idea of *deductive search*: pushing the given examples through the grammar top-down. [1] Intuitively, it applies the principle of divide-and-conquer to reduce the given synthesis problem to smaller subproblems of the same kind. The particular choice of subproblems and composition of their results depends on the algorithm.

Two families of deduction-based techniques have been developed in the last decade. One, popularized by Flash Fill [43] and the following PROSE framework [113], uses *inverse semantics* of the DSL operators. The other, popularized by the MYTH [102] and SYNQUID [111] systems, uses a *type-theoretic interpretation* of the PBE problem.

7.3.1 Inverse Semantics

In this approach, examples ϕ are propagated top-down through the grammar from expressions to their subexpressions. In other words, given a synthesis problem to find an expression of kind $F(E_1, E_2)$ that satisfies a spec ϕ (denoted as $F(E_1, E_2) \vDash \phi$), it reduces this problem to several simpler subproblems of kind $E_1 \vDash \phi_1$ and $E_2 \vDash \phi_2$. Here ϕ_1 and ϕ_2 are fresh specs on the subexpressions E_1 and E_2 of the desired program. They are constructed in a way that ensures that any program $F(E_1, E_2)$ with such subexpressions would satisfy ϕ.

Construction of subexpression specs depends on the requested top-level operator F in the desired program. Essentially, we need to *invert* F: if a program $F(E_1, E_2)$ outputs y on an input σ, what should E_1 and E_2 output on the same input? Answering this question provides us with *necessary* specs on E_1 and E_2 that ensure the entire program satisfies ϕ.

In practice, using a complete inverse semantics for each operator in the grammar may be overly computationally expensive. Most real-life operators do not have a simple closed-form inverse. To find the necessary subexpression specs, we may need to search over the entire codomain of the corresponding subexpressions, which is often either infinite or too

[1] Also known in the literature as *divide-and-conquer search*, *top-down search*, and *backpropagation-based search* (not to be confused with the backpropagation algorithm for training neural networks [20], although this name was inspired by similarities between the two techniques).

large. For example, consider an inverse semantics to a list-processing operator $\mathsf{Filter}(\lambda x \to \Pi, L)$ with respect to a given output list ℓ. It must find all lists L and predicates Π such that filtering of L with Π yields ℓ. For any non-trivial grammar underneath the symbols L and Π the number of possible such inputs is infinite.

To mitigate this, the approach uses three key ideas:

Witness function: A generalization of inverse semantics. It returns a *subset* of the possible inputs that produce the outputs in ϕ by employing some heuristics to pick only *likely* inputs. For instance, in the Filter scenario above, it may ignore large input lists L.

Constraints: An under-specification that generalizes examples. It describes some *property* of the output of the desired program, as opposed to providing the entire output value (like in Example 7.3). In the Filter scenario, one necessary spec on L is "its output should contain ℓ as a sub-list". Of course, the witness functions for the operators underneath L must be able to handle such a spec.

Case split: Per-parameter decomposition of inverse semantics. Instead of constructing all subexpression specs (i.e., necessary inputs) at the same time, we construct them one at a time. Each parameter E_i of each operator has a corresponding witness function, which constructs a necessary spec ϕ_i for that parameter subexpression. We then recursively learn a set of programs $\widetilde{E_i} \vDash \phi_i$ that are consistent with this spec. This set is *split* into subsets based on the outputs of its programs on the inputs in ϕ_i.[2] After that, the search considers each branch of the case split independently, under the assumption that the desired subexpression E_i evaluates to the corresponding output. All subsequent witness functions for the other parameters take this assumption into account, constructing smaller and simpler necessary specs.

In the Filter scenario, we can decompose the inverse semantics by first synthesizing possible input lists L, and then splitting on

[2] If ϕ_i is an example-based spec, the split will contain only one subset because all programs in $\widetilde{E_i}$ must output the same value. In a more general case where ϕ_i is a constraint there may be more than one subset.

them to synthesize possible predicates Π. Suppose we learned some finite set of programs $\widetilde{L}$ that are consistent with the constraint "its output should contain ℓ as a sub-list". For each concrete input list ℓ' produced by some program in $\widetilde{L}$ we can invoke a *conditional witness function* for the parameter Π. This witness function constructs a necessary spec on Π, answering the following question: what predicate filters a list ℓ' into ℓ? One natural implementation of this witness function constructs a constraint "Π must return `true` on all values in ℓ, and `false` on all values in $\ell \setminus \ell'$."

The overall algorithm relies on the designer-provided witness functions to backpropagate the examples through DSL operators. Traditionally, it also uses VSAs (§7.2) as its underlying representation for sets of learnt programs. It does not have to be a VSA but this data structure is particularly suitable as it supports efficient implementations of two important operations:

Intersection of program sets: to construct a set of programs consistent with multiple examples $\phi_1 \wedge \cdots \wedge \phi_n$ from n sets of programs that are consistent with individual examples.

Clustering of program sets: to perform a *case split* after learning a program set for a single parameter of a DSL operator.

(Optional) Ranking: to learn the best program with respect to some ranking function h, picking the best parameter programs from the learnt program sets on the fly.

7.3.2 Type-Based Perspective

A *type-based* perspective on deductive top-down PBE interprets a synthesis problem as a *type inhabitance* problem [33, 102, 111]. This interpretation is most powerful in a setting where the underlying DSL is loosely-constrained, that is, permits arbitrary type-safe combinations of subexpressions. In particular, any ML-like calculus with algebraic data types can serve as a core language for type-driven synthesis.

Type-theoretic interpretation of PBE relies on two key ideas. First, it views examples, specs, and constraints on the desired program as *type*

refinements. The theory of refinement types [32, 34, 121] studies types decorated with predicates from a decidable logic. For example, the type $\{\nu\colon \mathsf{List}\langle\alpha\rangle \mid \mathsf{length}\ \nu = n\}$ describes all lists of a fixed length n. Under this interpretation, an input-output example $(i\colon \tau_1,\ o\colon \tau_2)$ is simply a function type $\{\nu\colon \tau_1 \to \tau_2 \mid \nu\ i = o\}$, or, more succinctly, $\{i\} \to \{o\}$ where $\{x\}$ denotes a singleton type which contains only the value x.

Second, the theory of refinement types has developed procedures for solving the *type inhabitance* problem—finding a term that has a given type (which can be viewed isomorphically as *refinement type checking*). This is a well-studied problem in the literature, with natural deduction [29] being the most common approach. More generally, all such approaches to type checking/inhabitance exhibit properties similar to the top-down search: they push the refinements on an expression down to its subexpressions, and use the results of type checking for the subexpressions to infer new refinements for the sibling terms. Since ill-typed terms vastly outnumber well-typed ones, and local inference procedures eliminate the ill-typed candidates, checking type inhabitance is very efficient for sufficiently expressive refinements.

The most notable works that employ type-theoretic interpretation of program synthesis are Myth [33, 102] and Synquid [111]. The Myth framework is a type-theoretic interpretation of PBE, whereas Synquid supports arbitrary decidable constraints as refinements. Both works leverage and extend prior work on refinement type checking, described above. Specifically, Frankle et al. develop a sequent calculus with an efficient top-down theorem proving mechanism, and Polikarpova et al. extend the calculus of *liquid types* [121] with a similar top-down type inference procedure.

In addition to being an elegant interpretation of the synthesis problem, type-driven approaches also work extremely efficiently in scenarios when the underlying language permits efficient reasoning by a type inhabitance algorithm. A notable recent example of such a scenario is the application of Synquid to enforcement of information flow policies [112]. In this work, a Synquid-based component repairs an invalid access policy by synthesizing the weakest missing guard that would make the policy safe. Since the language of access policies is encoded

using liquid types, SYNQUID repairs complex policies in at most 100 sec, with the median time below 1 sec.

7.4 Ambiguity Resolution

Examples are inherently an under-specification, i.e. there might be multiple possible programs in a rich hypothesis space that are consistent with a given set of examples. Given enough examples, one can refine the specification such that only desired programs are consistent. But in many real-world settings, expecting users to provide large number of examples is not realistic, and the usability of the PBE system is often characterized by the fewest number of examples needed to understand the user intent. The number or representative input-output examples required to learn a desired program is a function of the underlying DSL and has also been previously referred to as the *teaching dimension* of the DSL [39]. As the expressiveness of the DSL increases (allowing users to perform richer variety of tasks), the number of representative examples needed to discriminate the programs in the DSL also increases. We now discuss two techniques to enable learning programs in rich DSLs using very few input-output examples (often only 1 input-output example).

7.4.1 Ranking

The main idea of ranking is to assign a likelihood score to each program in the set of programs induced from a small set of input-output examples such that the programs with the highest scores correspond to the desired user-intended programs. There have been many previous approaches in program synthesis where such ranking functions have been defined manually [51, 84, 90, 106]. The design of a good ranking function requires a lot of domain expertise and insights, and is typically a time-consuming and error-prone process. Moreover, such ranking functions are inherently not robust to changes in the DSL or benchmark problems as the space of possible ranking functions is quite large and it is difficult to analyze each possible function manually. Singh and Gulwani [128] proposed a machine learning based technique to automatically learn a ranking function (over syntactic program features) for PBE systems from

training data. Ellis and Gulwani [30] proposed another machine learning based technique to learn a ranking function that rely upon features that are independent of the program structure, instead relying upon a learned bias over program behaviors, and more generally over program execution traces. There are three key challenges in automatically learning such ranking functions. First, a large set of labeled training data is needed. Second, an appropriate machine learning technique is needed with the corresponding cost function to optimize. Finally, the ranking function should be such that it allows for efficient identification of the top-ranked program without having to enumerate every single program in the set of induced programs.

The machine learning based ranking technique was applied to the FlashFill PBE system [128] and resulted in significant improvements over previous manually designed ranking functions. FlashFill constructs the set of programs consistent with a given set of input-output examples using a VSA consisting of union nodes, join nodes, and explicit enumeration nodes. The ranking function is correspondingly defined in a hierarchical fashion in terms of individual ranking function for each kind of node in the VSA. The ranking function first recursively computes the top expression for each individual node in the DAG, and then composes them using an algorithm similar to Dijsktra's shortest path algorithm to find the top ranked top-level expression in the DSL. The hierarchical ranking function allows efficient identification of top-ranked program without enumeration (i.e. without breaking up the VSA-based sharing).

The idea to rank programs is inspired from the work on *learning to rank* [16, 24, 35, 54] techniques used in information retrieval. These techniques typically aim at ranking all relevant documents above all non-relevant documents or ranking the most relevant document as highest. However, in program synthesis, it is sufficient to rank *any* correct program higher than *all* incorrect programs. This is the key insight that is used to design the cost function to be optimized.

Let's assume the training data consists of a set of tasks $T = \{t_1, \cdots, t_n\}$. A task t_i consists of a set of input-output examples $E^i = \{e^i_1, \cdots, e^i_{n(t_i)}\}$, where an example $e^i_j = (\mathtt{in}^i_j, \mathtt{out}^i_j)$ denotes a pair of input string ($\mathtt{in}_i$) and output string ($\mathtt{out}_i$). During training, it is

assumed that enough input-output examples are provided such that only desired programs are consistent with the set of examples. The complete set of input-output examples for all tasks can be obtained by taking the union of the set of examples for each task $E = \{e_1, \cdots, e_{n(e)}\} = \cup_t E^t$. Let p_i denote the set of synthesized programs that are consistent with example e_i such that $p_i = \{p_i^1, \cdots, p_i^{n(i)}\}$, where $n(i)$ denotes the number of programs in the set p_i. The programs are labeled as positive and negative programs using the following definition.

Definition 7.1 (Positive and Negative Programs)**.** A program $p \in p_j$ is labeled as a positive program if it belongs to the set intersection of the set of programs for all examples of task t_i, i.e. $p \in p_1 \cap p_2 \cap \cdots \cap p_{n(t_i)}$. Otherwise, the program $p \in p_j$ is labeled a negative (or incorrect) program i.e. $p \notin p_1 \cap p_2 \cap \cdots \cap p_{n(t_i)}$.

In other words, if a program is consistent with all the input-output examples then it is labeled as a positive program. Otherwise, it is labeled as a negative program.

The training data can be automatically generated given a set of FlashFill benchmarks. Consider a training task t_i consisting of the input-output examples $E^i = \{(e_1, \cdots, e_{n(t_i)}\}$ and let p_j be the set of programs synthesized by the synthesis algorithm that are consistent with the input-output example e_j. For a task t_i, the set of all positive programs can be constructed as $p_1 \cap p_2 \cap \cdots \cap p_{n(t_i)}$, whereas the set of all negative programs can be computed as the set: $\{p_k \setminus (p_1 \cap p_2 \cap \cdots \cap p_{n(t_i)}) \mid 1 \leq k \leq n(t_i)\}$. The VSA based represented allows a polynomial time construction of these labeled program sets. The set of programs $p_i = \{p_i^1, \cdots, p_i^{n(i)}\}$ for an example e_i are assigned a corresponding set of labels $y_i = \{y_i^1, \cdots, y_i^{n(i)}\}$, where label y_i^j denotes the label for program p_i^j. The labels y_i^j take binary values such that the value $y_i^j = 1$ denotes that the program p_i^j is a positive program for the task, whereas the label value 0 denotes that program p_i^j is a negative program for the task.

Given the labeled training data, task is to learn a ranking function that can assign a score to programs to prefer positive programs over negative programs. A feature vector $x_i^j = \phi(e_i, p_i^j)$ can be computed for each example-program pair $(e_i, p_i^j), e_i \in E, p_i^j \in p_i$. For each example e_i, a training instance (x_i, y_i) is added to the train-

ing set, where $x_i = \{x_i^1, \cdots, x_i^{n(i)}\}$ denotes the list of feature vectors and $y_i = \{y_i^1, \cdots, y_i^{n(i)}\}$ denotes their corresponding labels. The goal is to learn a ranking function f that computes the ranking score $z_i = (f(x_i^1), \cdots, f(x_i^{n(i)}))$ for each example such that a positive program is ranked as highest.

This problem formulation is similar to the problem formulation of listwise approaches for learning-to-rank [16, 148]. The main difference comes from the fact that while previous listwise approaches aim to rank most documents in accordance with their training scores or rank the most relevant document as highest, in program synthesis, it is sufficient to rank any one positive program higher than all negative programs. Therefore, the loss function counts the number of examples where a negative program is ranked higher than all positive programs, as shown in Equation 7.1. For each example, the loss function compares the maximum rank of a negative program with the maximum rank of a positive program, and adds 1 to the loss function if a negative program is ranked highest (and subtracts 1 otherwise).

$$\begin{aligned} L(E) &= \sum_{i=1}^{n(e)} L(y_i, z_i) \\ &= \sum_{i=1}^{n(e)} \text{sign}\Big(\max\{f(x_i^j) \mid y_i^j = 0\} - \max\{f(x_i^k) \mid y_i^k = 1\}\Big) \end{aligned} \tag{7.1}$$

The non-continuity of the sign and max functions loss function makes it unsuitable for gradient descent based optimization as the gradient of the function can not be computed. Therefore, smooth approximations of the sign and max functions using the hyperbolic tanh function and softmax function respectively (with scaling constants c_1 and c_2) are used to obtain a continuous and differentiable loss function in Equation 7.2.

$$\begin{aligned} L(y_i, z_i) = \tanh\Big(c_1 \times \Big(&\frac{1}{c_2} \times \log \sum_{y_i^j=0} e^{c_2 \times f(x_i^j)} \\ &- \frac{1}{c_2} \times \log \sum_{y_i^k=1} e^{c_2 \times f(x_i^k)}\Big)\Big) \end{aligned} \tag{7.2}$$

The desired ranking function $f(x_i^j) = \vec{w} \cdot x_i^j$ is assumed to be a linear function over the features. The features are defined over programs and input-output example strings. Let there be m features in the feature vector $x_i^j = \{g_1, \cdots, g_m\}$ such that $f(x_i^j) = w_0 + w_1 g_1 + \cdots + w_m g_m$. The weights for the function can be learnt using the gradient descent algorithm that minimizes the loss function from Equation 7.2. The algorithm needs to be restarted multiple times to avoid local minimas.

The learnt ranking function performed significantly better than a ranking approach based on Occam's razor that smallest programs are the most general program. The learnt ranking function requires on average 1.44 examples per benchmark as compared to 4.18 required by the baseline approach. Moreover, it is able to learn the desired program from only 1 example for 74% of the benchmarks.

7.4.2 Active Learning

Another approach to reduce ambiguity is using the active learning approach to ask users for minimal number of additional input-output examples. Jha et.al. [59] proposed the idea of synthesizing *distinguishing inputs* for disambiguation in program synthesis, which we introduced in §3.2.1. The traditional synthesis approaches learn a program P that is consistent with a set of input-output examples $\{(i_k, o_k)\}_k$. The idea in active learning based approach is to synthesize two programs P_1 and P_2, and an input i, such that the two programs are consistent on the set of input-output examples, i.e. $P_1(i_k) = P_2(i_k) = o_k$, but they produce different output on the input i, i.e. $P_1(i) \neq P_2(i)$. The system can then ask the user to provide the desired output on input i to further guide the synthesis process.

Distinguishing inputs is one form of *active learning*—soliciting additional feedback from the user to initiate a new round of learning. Other user interaction models for PBE explored in the literature include:

- Displaying the program to the user. This may simplify finding a discrepancy by letting her analyze the program's behavior.
- Paraphrasing the program in natural language. This is a variation of the previous option, more suitable for non-programmers.

- Accepting *negative examples*, which indicate a discrepancy but do not provide the correct output for it. This simplifies interaction if manually computing the desired output is too cumbersome.

Mayer et al. used the FlashProg system [88], also described in §3.2.1, to compare these interaction models in the domain of text extraction by example. They found that disambiguating questions help achieve greater correctness rates and faster task completion times, as compared to other models. Comparison of different strategies for producing disambiguating question that would potentially alleviate unnecessary cognitive burden on the user is a topic of future research.

As of now, *minimizing* the number of clarifying questions during active learning is an open research problem. As mentioned before, the minimum number of questions required to learn a given concept is known as a *teaching dimension* of the DSL. Not only finding an optimal teaching sequence is NP-hard in general [39], it also has not yet been computed or bounded for most non-trivial languages used in program synthesis [58]. Despite the lack of optimal guarantees, active learning based approaches tend to converge quickly to the right solution, e.g. in 7 examples on average for SMT-guided learning of bitvector programs [59].

8

Future Work

We have seen tremendous progress recently in program synthesis techniques, and many PBE techniques are becoming mainstream inside industrial products. However, many challenges still remain. We conclude here with an overview of some immediate problems for the field.

Debuggability An important challenge going forward will be that of *debuggability*. The user would require active assistance to refine the specification. The user would like to be be confident that the synthesizer generated an intended program, especially when the synthesized program needs to be executed more than once, on sensitive data, or on large amounts of data where the results are not easy to verify manually.

Multi-modal input Another big opportunity is to define the next generation of programming experience that goes beyond the requirement to compose syntactically correct sequence of instructions to realize a particular task. This new paradigm shall facilitate interactive programming using multi-modal natural input from the user. While this article has focused on techniques for handling example-based specifications, it turns out that natural language is a better fit for certain class of tasks such as

spreadsheet queries [46] and smartphone scripts [79]. The new paradigm shall allow expressing intent using combination of various means [117] such as examples, demonstrations, natural language, keywords, and sketches.

Adaptivity Program synthesis techniques can potentially benefit by leveraging data from past invocations of the synthesizer by the same user or by other users in the cloud or enterprise. This past data can help guide the search more efficiently or help resolve the ambiguity in the user's under-specification more effectively.

Statistical techniques Program synthesis techniques have mostly leveraged use of logical methods for search. While these techniques are good at leveraging semantic knowledge or properties of the various operators, they fall into scalability challenges when having to handle many disjunctive choices during search. There have recently been very impressive advances in use of deep learning methods for predicting various kinds of intended artifacts after being trained on relevant data. An interesting line of work would be to synergistically combine logical techniques with statistical techniques like deep learning methods to develop the next generation of foundational search techniques. It would also facilitate reasoning and synthesis for larger pieces of code.

Scaling The current program synthesis techniques have been successful at learning small programs with complex logic. Scaling the size of programs that can be synthesized remains an active area of research.

Knowledge transfer Most modern synthesis approaches leverage some form of domain-specific knowledge to reach sufficient scaling levels. While developing fast universal search algorithms currently is out of reach for synthesis, it may be possible to automatically generalize and transfer the insights learned from one domain to another.

Industrialization An important milestone in the lifetime of a technology is making it accessible for the general engineering audience, as

opposed to professionals with a relevant background. Program synthesis is a relatively new field, and thus is not yet as widespread as program analysis. Further development of synthesis frameworks, solver-aided languages, and separation of domain-specific search components should help to further close this gap.

Acknowledgements

We thank Ravi Chugh for his assistance with the write-up on prodirect manipulation, and Emina Torlak for her feedback and assistance with the write-up on solver-aided programming.

References

[1] Takuya Akiba, Kentaro Imajo, Hiroaki Iwami, Yoichi Iwata, Toshiki Kataoka, Naohiro Takahashi, Michał Moskal, and Nikhil Swamy. Calibrating research in program synthesis using 72,000 hours of programmer time. *Microsoft Research, Redmond, WA, USA, Technical Report*, 2013.

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293. ACM, 2014.

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

[4] Rajeev Alur, Pavol Cerný, and Arjun Radhakrishna. Synthesis through unification. In *Proceedings of the 27th International Conference on Computer-Aided Verification (CAV)*, pages 163–179, 2015.

[5] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. Results and analysis of SyGuS-Comp'15. In *Proceedings Fourth Workshop on Synthesis, SYNT*, pages 3–26, 2015.

[6] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. SyGuS-Comp 2016: Results and analysis. In *Proceedings of the Fifth Workshop on Synthesis, SYNT@CAV 2016, Toronto, Canada, July 17-18, 2016.*, pages 178–202, 2016.

[7] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. *CoRR*, abs/1611.01989, 2016.

[8] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 394–403, 2006.

[9] Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 177–192, 2008.

[10] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin G. Zorn. FlashRelate: extracting relational data from semi-structured spreadsheets using examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 218–228, 2015.

[11] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard Version 2.6*, 2010.

[12] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, César Kunz, and Mark Marron. From relational verification to SIMD loop synthesis. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*, pages 123–134, 2013.

[13] Alan W Biermann. The inference of regular LISP programs from examples. *IEEE transactions on Systems, Man, and Cybernetics*, 8(8): 585–600, 1978.

[14] James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and litmus tests. In *Programming Languages Design and Implementation*, 2017.

[15] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. In *ACM SIGPLAN Notices*, volume 51, pages 775–788. ACM, 2016.

[16] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: from pairwise approach to listwise approach. In *ICML*, 2007.

[17] Pavol Cerný, Krishnendu Chatterjee, Thomas A. Henzinger, Arjun Radhakrishna, and Rohit Singh. Quantitative synthesis for concurrent programs. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 243–259, 2011.

[18] Krishnendu Chatterjee, Thomas A. Henzinger, Barbara Jobstmann, and Rohit Singh. Measuring and synthesizing systems in probabilistic environments. *J. ACM*, 62(1):9:1–9:34, March 2015. ISSN 0004-5411.

[19] Swarat Chaudhuri, Martin Clochard, and Armando Solar-Lezama. Bridging boolean and quantitative synthesis using smoothed proof search. *SIGPLAN Not.*, 49(1):207–220, January 2014. ISSN 0362-1340.

[20] Yves Chauvin and David E. Rumelhart. *Backpropagation: theory, architectures, and applications*. Psychology Press, 1995.

[21] Salman Cheema, Sarah Buchanan, Sumit Gulwani, and Joseph J. LaViola Jr. A practical framework for constructing structured drawings. In *IUI'14 19th International Conference on Intelligent User Interfaces, IUI'14, Haifa, Israel, February 24-27, 2014*, pages 311–316, 2014.

[22] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 3–14, 2013.

[23] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. Programmatic and direct manipulation, together at last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 341–354, 2016.

[24] David Cossock and Tong Zhang. Subset ranking using regression. *Learning Theory*, 4005:605–619, 2006.

[25] Allen Cypher, editor. *Watch What I Do – Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[26] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program Repair with Quantitative Objectives. In *Computer Aided Verification (CAV)*. Springer-Verlag, 2016.

[27] Luc De Raedt. *Logical and Relational Learning*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[28] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R., and Subhajit Roy. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 345–356, 2016.

[29] Joshua Dunfield. *A unified system of type refinements*. PhD thesis, Air Force Research Laboratory, 2007.

[30] Kevin Ellis and Sumit Gulwani. Learning to learn programs from examples: Going beyond program structure. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

[31] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017.

[32] Cormac Flanagan. Hybrid type checking. In *ACM Sigplan Notices*, volume 41, pages 245–256. ACM, 2006.

[33] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *ACM SIGPLAN Notices*, volume 51, pages 802–815. ACM, 2016.

[34] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM. .

[35] Yoav Freund, Raj Iyer, Robert E. Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *The Journal of machine learning research*, 4:933–969, 2003.

[36] Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. TerpreT: A probabilistic programming language for program induction. *CoRR*, abs/1608.04428, 2016.

[37] Khaled Ghédira. *Constraint Satisfaction Problems: CSP Formalisms and Techniques*. John Wiley & Sons, 2013.

[38] Patrice Godefroid and Ankur Taly. Automated synthesis of symbolic instruction encodings from I/O samples. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 441–452. ACM, 2012.

[39] Sally A. Goldman and Michael J. Kearns. On the complexity of teaching. *Journal of Computer and System Sciences*, 50:303–314, 1992.

[40] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[41] C. Cordell Green. Application of theorem proving to problem solving. In *IJCAI*, pages 219–240, 1969.

[42] Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, page 1, 2010.

[43] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330, 2011.

[44] Sumit Gulwani. Programming by examples - and its applications in data wrangling. In *Dependable Software Systems Engineering*, pages 137–158. 2016.

[45] Sumit Gulwani and Nebojsa Jojic. Program verification as probabilistic inference. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 277–289, 2007.

[46] Sumit Gulwani and Mark Marron. NLyze: interactive programming by natural language for spreadsheet data analysis and manipulation. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 803–814, 2014.

[47] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 62–73, 2011.

[48] Sumit Gulwani, Vijay Anand Korthikanti, and Ashish Tiwari. Synthesizing geometry constructions. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 50–61, 2011.

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

[50] Sumit Gulwani, Mikaël Mayer, Filip Niksic, and Ruzica Piskac. StriSynth: Synthesis for live programming. In *37th IEEE/ACM International Conference on Software Engineering, ICSE*, pages 701–704, 2015.

[51] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *ACM SIGPLAN Notices*, volume 48, pages 27–38. ACM, 2013.

[52] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 317–328, 2011.

[53] Brian Hempel and Ravi Chugh. Semi-automated SVG programming via direct manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology, UIST 2016, Tokyo, Japan, October 16-19, 2016*, pages 379–390, 2016.

[54] Ralf Herbrich, Thore Graepel, and Klaus Obermayer. Large margin rank boundaries for ordinal regression. *Advances in Neural Information Processing Systems*, pages 115–132, 1999.

[55] Ralf Herbrich, Tom Minka, and Thore Graepel. TrueSkill™: A Bayesian skill rating system. In *Advances in neural information processing systems*, pages 569–576, 2006.

[56] John H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[57] Shachar Itzhaky, Sumit Gulwani, Neil Immerman, and Mooly Sagiv. A simple inductive synthesis methodology and its applications. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 36–46, 2010.

[58] Susmit Jha and Sanjit A. Seshia. A Theory of Formal Synthesis via Inductive Learning. *ArXiv e-prints*, May 2015.

[59] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering (ICSE)*, volume 1, pages 215–224. IEEE, 2010.

[60] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. Program repair as a game. In *Computer Aided Verification (CAV)*, pages 226–238. Springer-Verlag, 2005.

[61] Colin G. Johnson. Genetic programming with fitness based on model checking. In *European Conference on Genetic Programming*, pages 114–124. Springer, 2007.

[62] Vladimir Jojic, Sumit Gulwani, and Nebojsa Jojic. Probabilistic inference of programs from input/output examples. Technical Report MSR-TR-2006-103, Microsoft Research, 2006.

[63] Rajeev Joshi, Greg Nelson, and Keith H. Randall. Denali: A goal-directed superoptimizer. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 304–314, 2002.

[64] Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *NIPS*, pages 190–198, 2015.

[65] Garvit Juniwal, Alexandre Donzé, Jeff C. Jensen, and Sanjit A. Seshia. CPSGrader: Synthesizing temporal logic testers for auto-grading an embedded systems laboratory. In *EMSOFT*, pages 24:1–24:10, 2014.

[66] Łukasz Kaiser and Ilya Sutskever. Neural GPUs learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

[67] Susumu Katayama. MagicHaskeller on the Web: Automated programming as a service. In *Haskell Symposium*, 2013.

[68] Gal Katz and Doron A. Peled. Genetic programming and model checking: Synthesizing new mutual exclusion algorithms. In *Automated Technology for Verification and Analysis, 6th International Symposium, ATVA 2008, Seoul, Korea, October 20-23, 2008. Proceedings*, pages 33–47, 2008.

[69] Ross D. King, Stephen Muggleton, Ashwin Srinivasan, and M.J. Sternberg. Structure-activity relationships derived by machine learning: The use of atoms and their bond connectivities to predict mutagenicity by inductive logic programming. *Proceedings of the National Academy of Sciences*, 93(1):438–442, 1996.

[70] A. N. Kolmogorov. Zur deutung der intuitionistischen logik. *Math. Zeitschr*, 35:58–365, 1932.

[71] John R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 768–774. Morgan Kaufmann Publishers Inc., 1989.

[72] John R. Koza. *Genetic programming - on the programming of computers by means of natural selection*. Complex adaptive systems. MIT Press, 1993.

[73] John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2):87–112, 1994.

[74] Shriram Krishnamurthi. Educational pearl: Automata via macros. *Journal of Functional Programming*, 16(03):253–267, 2006.

[75] Michael Kuperstein, Martin T. Vechev, and Eran Yahav. Automatic inference of memory fences. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 111–119, 2010.

[76] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.

[77] Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. Version space algebra and its application to programming by demonstration. In *Proceedings of the 17th International Conference on Machine Learning (ICML)*, pages 527–534, 2000.

[78] Vu Le and Sumit Gulwani. FlashExtract: a framework for data extraction by examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 55, 2014.

[79] Vu Le, Sumit Gulwani, and Zhendong Su. SmartSynth: synthesizing smartphone automation scripts from natural language. In *The 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'13, Taipei, Taiwan, June 25-28, 2013*, pages 193–206, 2013.

[80] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[81] Alan Leung, John Sarracino, and Sorin Lerner. Interactive parser synthesis by example. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 565–574, 2015.

[82] Percy Liang, Michael I. Jordan, and Dan Klein. Learning programs: A hierarchical Bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 639–646, 2010.

[83] Francesca A. Lisi. Building rules on top of ontologies for the Semantic Web with inductive logic programming. *Theory and Practice of Logic Programming*, 8(03):271–300, 2008.

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

[85] Zohar Manna and Richard J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165, 1971.

[86] Zohar Manna and Richard J. Waldinger. Knowledge and reasoning in program synthesis. *Artif. Intell.*, 6(2):175–208, 1975.

[87] Henry Massalin. Superoptimizer - A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), Palo Alto, California, USA, October 5-8, 1987.*, pages 122–126, 1987.

[88] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Ben Zorn, and Sumit Gulwani. User interaction models for disambiguation in programming by example. In *28th ACM User Interface Software and Technology Symposium*, pages 291–301. ACM, 2015.

[89] Aditya Krishna Menon, Omer Tamuz, Sumit Gulwani, Butler W. Lampson, and Adam Kalai. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 187–195, 2013.

[90] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *OOPSLA*, pages 997–1016, 2012.

[91] Tom M. Mitchell. Generalization as search. *Artificial intelligence*, 18(2): 203–226, 1982.

[92] Stephen Muggleton. Inductive logic programming. *New generation computing*, 8(4):295–318, 1991.

[93] Stephen Muggleton. Inverse entailment and Progol. *New generation computing*, 13(3-4):245–286, 1995.

[94] Stephen Muggleton and Wray Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the fifth international conference on machine learning*, pages 339–352, 1992.

[95] Stephen Muggleton, Ross D. King, and Michael J.E. Stenberg. Protein secondary structure prediction using logic-based machine learning. *Protein Engineering*, 5(7):647–657, 1992.

[96] Stephen Muggleton, Dianhuan Lin, Niels Pahlavi, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning: application to grammatical inference. *Machine Learning*, 94(1):25–49, 2014.

[97] Stephen Muggleton, Dianhuan Lin, and Alireza Tamaddoni-Nezhad. Meta-interpretive learning of higher-order dyadic Datalog: Predicate invention revisited. *Machine Learning*, 100(1):49–73, 2015.

[98] Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834*, 2015.

[99] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: program repair via semantic analysis. In *35th International Conference on Software Engineering, ICSE*, pages 772–781, 2013.

[100] Robert P. Nix. Editing by example. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, USA, January 1984*, pages 186–195, 1984.

[101] Aditya V. Nori, Sherjil Ozair, Sriram K. Rajamani, and Deepak Vijaykeerthy. Efficient synthesis of probabilistic programs. In *ACM SIGPLAN Notices*, volume 50, pages 208–217. ACM, 2015.

[102] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 619–630. ACM, 2015.

[103] Pavel Panchekha and Emina Torlak. Automated reasoning for web page layout. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016*, pages 181–194, 2016.

[104] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. *ACM SIGPLAN Notices*, 50(6):1–11, 2015.

[105] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. *CoRR*, abs/1611.01855, 2016.

[106] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *ACM SIGPLAN Notices*, volume 47, pages 275–286. ACM, 2012.

[107] Stuart Pernsteiner, Calvin Loncaric, Emina Torlak, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Jonathan Jacky. *Investigating Safety of a Radiotherapy Machine Using System Models with Pluggable Checkers*, pages 23–41. Springer International Publishing, Cham, 2016. .

[108] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *ACM SIGPLAN Notices*, volume 49, pages 396–407. ACM, 2014.

[109] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodík, and Dinakar Dhurjati. Scaling up superoptimization. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 297–310, 2016.

[110] Amir Pnueli and Roni Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190. ACM, 1989.

[111] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 522–538. ACM, 2016.

[112] Nadia Polikarpova, Jean Yang, Shachar Itzhaky, and Armando Solar-Lezama. Type-driven repair for information flow security. *arXiv preprint arXiv:1607.03445*, 2016.

[113] 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, OOPSLA 2015*, pages 107–126, 2015.

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

[115] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. In *ACM SIGPLAN Notices*, volume 51, pages 761–774. ACM, 2016.

[116] Mohammad Raza and Sumit Gulwani. Automated data extraction using predictive program synthesis. In *AAAI*, 2017.

[117] Mohammad Raza, Sumit Gulwani, and Natasa Milic-Frayling. Compositional program synthesis from natural language and examples. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 792–800, 2015.

[118] Scott Reed and Nando de Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

[119] Wolfgang Reisig. *Petri nets: an introduction*, volume 4. Springer Science & Business Media, 2012.

[120] Sebastian Riedel, Matko Bošnjak, and Tim Rocktäschel. Programming with a differentiable Forth interpreter. *arXiv preprint arXiv:1605.06640*, 2016.

[121] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *ACM SIGPLAN Notices*, volume 43, pages 159–169. ACM, 2008.

[122] Jose C.A. Santos, Houssam Nassif, David Page, Stephen Muggleton, and Michael J.E. Sternberg. Automated identification of protein-ligand interaction features using inductive logic programming: a hexose binding case study. *BMC bioinformatics*, 13(1):1, 2012.

[123] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 305–316. ACM, 2013.

[124] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13*, pages 305–316, 2013.

[125] David E. Shaw, William R. Swartout, and C. Cordell Green. Inferring LISP programs from examples. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1*, pages 260–267. Morgan Kaufmann Publishers Inc., 1975.

[126] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Computer Aided Verification - 24th International Conference, CAV 2012*, pages 634–651, 2012.

[127] Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. *Proceedings of the VLDB Endowment*, 5(8): 740–751, 2012.

[128] Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In *CAV*, pages 398–414, 2015.

[129] Rishabh Singh and Sumit Gulwani. Transforming spreadsheet data types using examples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 343–356, 2016.

[130] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automatic feedback generation for introductory programming assignments. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 15–26, 2013.

[131] David Canfield Smith. *Pygmalion: A Creative Programming Environment.* PhD thesis, Stanford University, Stanford, CA, USA, 1975.

[132] Armando Solar-Lezama. *Program synthesis by sketching.* ProQuest, 2008.

[133] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 313–326, 2010.

[134] Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. Path-based inductive synthesis for program inversion. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 492–503, 2011.

[135] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer (STTT)*, 15(5-6):497–518, 2013.

[136] Michael J.E. Sternberg, Alireza Tamaddoni-Nezhad, Victor I. Lesk, Emily Kay, Paul G. Hitchen, Adrian Cootes, Lieke B. van Alphen, Marc P. Lamoureux, Harold C. Jarrell, Christopher J. Rawlings, et al. Gene function hypotheses for the campylobacter jejuni glycome generated by a logic-based approach. *Journal of molecular biology*, 425(1):186–197, 2013.

[137] Phillip D. Summers. A methodology for LISP program construction from examples. *Journal of the ACM (JACM)*, 24(1):161–175, 1977.

[138] Alireza Tamaddoni-Nezhad, Ghazal Afroozi Milani, Alan Raybould, Stephen Muggleton, and David A. Bohan. Construction and validation of food webs using logic-based machine learning and text mining. *Advances in Ecological Research*, 49:225–289, 2013.

[139] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152. ACM, 2013.

[140] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Notices*, volume 49, pages 530–541. ACM, 2014.

[141] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. TRANSIT: Specifying protocols with concolic snippets. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Languages Design and Implementation (PLDI)*, pages 287–296, 2013.

[142] Richard Uhler and Nirav Dave. Smten: automatic translation of high-level symbolic computations into SMT queries. In *International Conference on Computer Aided Verification*, pages 678–683. Springer, 2013.

[143] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 327–338, 2010.

[144] Richard J. Waldinger and Richard C. T. Lee. PROW: A step toward automatic program writing. In *IJCAI*, pages 241–252, 1969.

[145] Henry S. Warren. *Hacker's delight*. Pearson Education, 2013.

[146] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 364–374, 2009.

[147] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Scalable verification of border gateway protocol configurations with an SMT solver. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 765–780. ACM, 2016.

[148] Fen Xia, Tie-Yan Liu, Jue Wang, Wensheng Zhang, and Hang Li. Listwise approach to learning to rank: theory and algorithm. In *ICML*, 2008.

[149] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. Synthesizing transformations on hierarchically structured data. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 508–521, 2016.

[150] Xiaofeng Yang, Jian Su, Jun Lang, Chew Lim Tan, Ting Liu, and Sheng Li. An entity-mention model for coreference resolution with inductive logic programming. In *ACL*, pages 843–851, 2008.

[151] Hongyu Zhang, Anuj Jain, Gaurav Khandelwal, Chandrashekhar Kaushik, Scott Ge, and Wenxiang Hu. Bing developer assistant: improving developer productivity by recommending sample code. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 956–961. ACM, 2016.