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

Full text available at: http://dx.doi.org/10.1561/2500000010
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.
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

1 Introduction ................................................. 2
   1.1 Program Synthesis ..................................... 2
   1.2 Challenges ............................................. 4
   1.3 Dimensions in Program Synthesis ....................... 6
   1.4 Roadmap ................................................ 12

2 Applications ................................................. 13
   2.1 Data Wrangling ......................................... 13
   2.2 Graphics ................................................ 21
   2.3 Code Repair ............................................. 24
   2.4 Code Suggestions ....................................... 28
   2.5 Modeling ................................................ 29
   2.6 Superoptimization ...................................... 30
   2.7 Concurrent Programming .................. 32

3 General Principles ........................................... 35
   3.1 Second-Order Problem Reduction ...................... 35
   3.2 Oracle-Guided Synthesis ............................... 37
   3.3 Syntactic Bias .......................................... 43
   3.4 Optimization ........................................... 51

Full text available at: http://dx.doi.org/10.1561/2500000010
4 Enumerative Search 54
  4.1 Enumerative Search ................................. 54
  4.2 Bidirectional Enumerative Search ................. 59
  4.3 Offline Exhaustive Enumeration and Composition 60

5 Constraint Solving 62
  5.1 Component-Based Synthesis .......................... 64
  5.2 Solver-Aided Programming .......................... 68
  5.3 Inductive Logic Programming ....................... 72

6 Stochastic Search 74
  6.1 Metropolis-Hastings Algorithm for Sampling Expressions 74
  6.2 Genetic Programming ............................... 77
  6.3 Machine Learning .................................. 81
  6.4 Neural Program Synthesis ........................... 82

7 Programming by Examples 87
  7.1 Problem Definition ................................. 87
  7.2 Version Space Algebra ............................. 89
  7.3 Deduction-Based Techniques ....................... 91
  7.4 Ambiguity Resolution .............................. 96

8 Future Work 102

Acknowledgements 105

References 106
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.

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
1.1. Program Synthesis

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 an 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 tech-
niques 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 success-
ful constraint-based synthesis applications tracing their roots back to
SKETCH and the invention of counterexample-guided inductive synthe-
sis [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! 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 pro-
gramming. 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

---

1See Rastislav Bodik’s ICFP-2015 keynote talk “Program Synthesis: Opportunities
for the Next Decade” for a detailed comparison: [https://youtu.be/PI99A08Y83E](https://youtu.be/PI99A08Y83E)
Introduction

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 may contain 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. 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.

1.3.1 User Intent

The user intent can be expressed in various forms including logical specification, examples, traces, natural language, 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 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 parser 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.
1.3. Dimensions in Program Synthesis

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 $\phi$ to simpler sub-problems (where the search is either over sub-expressions of $e$ or over sub-specifications of $\phi$), 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 $\phi$. 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.
1.3. Dimensions in Program Synthesis

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-output 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
Introduction

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.
References


References


References


References

[40] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing machines. 


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 

[44] Sumit Gulwani. Programming by examples - and its applications in data 
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, 

[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, 

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.

2011.


[50] Sumit Gulwani, Mikaël Mayer, Filip Niksic, and Ruzica Piskac. StriSynth: 
References


References


References


[98] Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural program-
mer: Inducing latent programs with gradient descent. arXiv preprint

[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

[101] Aditya V. Nori, Sherjil Ozair, Sriram K. Rajamani, and Deepak Vijaya-
keerthy. Efficient synthesis of probabilistic programs. In ACM SIGPLAN

[102] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed
program synthesis. In Proceedings of the 36th ACM SIGPLAN Confer-
ence on Programming Language Design and Implementation (PLDI),

[103] Pavel Panchekha and Emina Torlak. Automated reasoning for web
Conference on Object-Oriented Programming, Systems, Languages, and

[104] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary
Tatlock. Automatically improving accuracy for floating point expressions.

[105] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li,
CoRR, abs/1611.01855, 2016.

Type-directed completion of partial expressions. In ACM SIGPLAN

[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,

[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,


References


References


References


