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 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

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.

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.