Program synthesis

$$\Gamma \vdash ? : \tau$$

Jeremy Yallop

jeremy.yallop@cl.cam.ac.uk

The program synthesis problem

What is the synthesis problem?

Program Synthesis (Gulwani et al, 2017):

> *...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**.*
>
> *(emphasis mine)*

That is, it's a search for a constructive proof of a quantified formula:

$$\exists f . \forall \; input . Specification$$

When is program synthesis useful?

Efficiency in programming

(low-level code from high-level specifications)

Effective compilation

(e.g.*superoptimization*)

Program repair

(updating buggy programs to fit a specification)

Deobfuscation

(restoring readability)

End-user programming

(e.g. interactive programming-by-examples)

Program transformation

(updating programs as specifications evolve)

What is a specification?

"...the user intent expressed in the form of some specification ..."

A logical specification

$f(x, y) \geq x \wedge f(x, y) \geq y$

A type

$$x : \mathbb{Z} \to y : \mathbb{Z} \to \{z : \mathbb{Z} \mid z = max(x, y)\}$$

An existing program

`slow_max(x,y)`

Input-output examples

$f(2, 4) = 4, f(5, 2) = 5, \ldots$

Natural language

"The larger of x and y"

One approach: Syntax-Guided Synthesis (SyGuS)

logical formula

$$f\colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$
$$f(x, y) = f(y, x) \ \wedge \ f(x, y) \geq x$$

grammar (search space)

```
T   ::=  x | y | 0 | 1 | ITE(C,T,T)
C   ::=  T ≤ T | ¬ T | C ∧ C
```

SyGuS

```
f(x,y) = ITE((x≤y),y,x)
```

Example from *Search-based Program Synthesis*, Alur et al (2018)

Why is program synthesis hard?

Challenge: big search space

Synthesis is often based on some form of **enumeration** of programs.

However, the search space is extremely large (exponential in program length).

Some form of **pruning** or **guidance** is necessary, e.g. by using

abstract interpretation   grammar refinement   syntactic templates

domain equations   component-based construction

stochastic search   constraint solving   precise types

Challenges 2: determining correctness

How can we tell when we've found a solution?

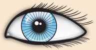
SMT solving

Z3

Type checking

$$\Gamma \vdash e : \tau$$

Human inspection



Testing

✓

Success in limited domains

Spreadsheet formulas

Regular expressions

a(b|c)*d

Trigonometric functions

Loop-free programs

SQL queries

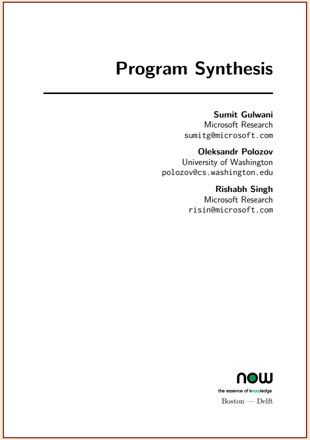
from `t` **select** `*`
where

Bit twiddling

`x & 0xBEEF << y`

Reading

Background reading: 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

"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."

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

Online:
`https://microsoft.com/en-us/research/wp-content/uploads/2017/10/program_synthesis_now.pdf`

Paper 1: oracle-guided component-based (2010)

The problem

Challenges

Reading

Oracle-Guided Component-Based Program Synthesis

Susmit Jha, UC Berkeley, jha@eecs.berkeley.edu
Sumit Gulwani, Microsoft Research, sumitg@microsoft.com
Sanjit A. Seshia, UC Berkeley, sseshia@eecs.berkeley.edu
Ashish Tiwari, SRI International, tiwari@csl.sri.com

ABSTRACT

We present a novel approach to automatic synthesis of loop-free programs. The approach is based on a combination of oracle-guided learning from examples, and constraint-based synthesis from components using satisfiability modulo theories (SMT) solvers. Our approach is suitable for many applications, including as an aid to program understanding tasks such as deobfuscating malware. We demonstrate the efficiency and effectiveness of our approach by synthesizing bit-manipulating programs and by deobfuscating programs.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; I.2.2 [**Artificial Intelligence**]: Program Synthesis; K.3.2 [**Learning**]: Concept Learning

Keywords

Program synthesis, Oracle-based learning, SMT, SAT

1. INTRODUCTION

Automatic synthesis of programs has long been one of the holy grails of software engineering. It has found many practical applications: generating optimal code sequences [20, 11], optimizing performance-critical inner loops, generating general-purpose peephole optimizers [2, 3], automating repetitive programming tasks [15], and filling in low-level details after the higher-level intent has been expressed [24]. Two applications of synthesis are of particular interest in this paper. The first is that of automating the discovery of non-intuitive algorithms (e.g., [8]). The second application, as we show in this paper, is *program understanding*, and more specifically, *program deobfuscation*. The need for deobfuscation techniques has arisen in recent years, especially due to an increase in the amount of malicious, and mostly obfuscated, code (malware) [28]. Currently, human experts use decompilers and manually deobfuscate the resulting code (see, e.g., [22]). Clearly, this is a tedious task that could benefit from automated tool support.

A traditional view of program synthesis is that of synthesis from complete specifications. One approach is to give a specification as a formula in a suitable logic [19, 26, 10, 8]. Another is to write the specification as a simpler, but possibly far less efficient program [20, 11, 24]. While these approaches have the advantage of completeness of specification, such specifications are often unavailable, difficult to write, or expensive to check against using automated verification techniques. In this paper, we propose a novel *oracle-guided* approach to program synthesis, where an *I/O oracle* that maps a given program input to the desired output is used as an alternative to having a complete specification. The key idea of our algorithm is to query the I/O oracle on an input that can distinguish between non-equivalent programs that are consistent with the past interaction with the I/O oracle. The process is repeated until a semantically unique program is obtained. Our experimental results show that only few rounds of interaction are needed.

We apply the oracle-guided approach to automated synthesis of *loop-free programs*, those that compute functions of their input and terminate. Such programs arise in a variety of application contexts, such as low-level bit-manipulating code, scientific computing kernels, parts of control software in graphical languages such as LabVIEW, and even applications in high-level scripting languages such as Javascript and Ruby that are formed by chaining multiple high-level operators. A key characteristic of our method is that it is *component-based*, meaning that we synthesize a program by performing a circuit-style, loop-free composition of components drawn from a given component library. We can also address the challenge of identifying whether the given set of components is insufficient to synthesize the desired program. For this purpose, we additionally require making only one query to a more expensive *validation oracle* that checks whether the program is correct or not.

Our synthesis algorithm is based on a novel *constraint-based* approach that reduces the synthesis problem to that of solving two kinds of constraints: the *I/O-behavioral constraint* whose solution yields a candidate program consistent with the interaction with the I/O oracle, and the *distinguishing constraint* whose solution provides the input that distinguishes between non-equivalent candidate programs. These constraints can be solved using off-the-shelf SMT (Satisfiability Modulo Theory) solvers. Traditional synthesis algorithms perform a expensive combinatorial search over the space of all possible programs. In contrast, our technique leaves the inherent exponential nature of the problem to the underlying SMT solver, whose engineering advances over the years allow them to effectively deal with problem instances that arise in practice, which are usually not hard, and hence end up not requiring exponential reasoning.

Contributions and Organization.

"We present a novel approach to automatic synthesis of loop-free programs. The approach is based on a combination of oracle-guided learning from examples, and constraint-based synthesis from components using satisfiability modulo theories (SMT) solvers.[...]

"We demonstrate the efficiency and effectiveness of our approach by synthesizing bit-manipulating programs and by deobfuscating programs."

Paper 2: refinement types (2016)

Program Synthesis from Polymorphic Refinement Types

Nadia Polikarpova Ivan Kuraj Armando Solar-Lezama
MIT CSAIL, USA
{polikarn,ivanko,asolar}@csail.mit.edu

Abstract

We present a method for synthesizing recursive functions that provably satisfy a given specification in the form of a polymorphic refinement type. We observe that such specifications are particularly suitable for program synthesis for two reasons. First, they offer a unique combination of expressive power and decidability, which enables automatic verification—and hence synthesis—of nontrivial programs. Second, a type-based specification for a program can often be effectively decomposed into independent specifications for its components, causing the synthesizer to consider fewer component combinations and leading to a combinatorial reduction in the size of the search space. At the core of our synthesis procedure is a new algorithm for refinement type checking, which supports specification decomposition.

We have evaluated our prototype implementation on a large set of synthesis problems and found that it exceeds the state of the art in terms of both scalability and usability. The tool was able to synthesize more complex programs than those reported in prior work (several sorting algorithms and operations on balanced search trees), as well as most of the benchmarks tackled by existing synthesizers, often starting from a more concise and intuitive user input.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; I.2.2 [*Automatic Programming*]: Program Synthesis

1. Introduction

The key to scalable program synthesis is modular verification. Modularity enables the synthesizer to prune candidates for different subprograms independently, whereby combinatorially reducing the size of the search space it has to consider. This explains the recent success of *type-directed* approaches to synthesis of functional programs [12, 14, 15, 27]: not only do ill-typed programs vastly outnumber well-typed ones, but more importantly, a type error can be detected long before the whole program is put together.

Simple, coarse-grained types alone are, however, rarely sufficient to precisely describe a synthesis goal. Therefore, existing approaches supplement type information with other kinds of specifications, such as input-output examples [1, 12, 27], or pre- and post-conditions [20, 21]. Alas, the corresponding verification procedures rarely enjoy the same level of modularity as type checking, thus fundamentally limiting the scalability of these techniques.

In this work we present a novel system that pushes the idea of type-directed synthesis one step further by taking advantage of *refinement types* [13, 33]: types decorated with predicates from a decidable logic. For example, imagine that a user intends to synthesize the function `replicate`, which, given a natural number n and a value x, produces a list that contains n copies of x. In our system, the user can express this intent by providing the following signature:

$$\texttt{replicate} :: n{:}\texttt{Nat} \to x{:}\alpha \to \{\nu{:}\ \texttt{List}\ \alpha \mid \texttt{len}\ \nu = n\}$$

"We present a method for synthesizing recursive functions that provably satisfy a given specification in the form of a polymorphic refinement type.

"a unique combination of expressive power and decidability [...] a type-based specification for a program can often be effectively decomposed into independent specifications for its components [...] leading to a combinatorial reduction in the size of the search space.

"The tool was able to synthesize more complex programs than those reported in prior work (several sorting algorithms and operations on balanced search trees) [...] often starting from a more concise and intuitive user input."

Paper 3: abstract interpretation (2023)

Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation

YONGHO YOON, Seoul National University, Korea
WOOSUK LEE*, Hanyang University, Korea
KWANGKEUN YI, Seoul National University, Korea

A key challenge in example-based program synthesis is the gigantic search space of programs. To address this challenge, various work proposed to use abstract interpretation to prune the search space. However, most of existing approaches have focused only on forward abstract interpretation, and thus cannot fully exploit the power of abstract interpretation. In this paper, we propose a novel approach to inductive program synthesis via iterative forward-backward abstract interpretation. The forward abstract interpretation computes possible outputs of a program given inputs, while the backward abstract interpretation computes possible inputs of a program given outputs. By iteratively performing the two abstract interpretations in an alternating fashion, we can effectively determine if any completion of each partial program as a candidate can satisfy the input-output examples. We apply our approach to a standard formulation, syntax-guided synthesis (SyGuS), thereby supporting a wide range of inductive synthesis tasks. We have implemented our approach and evaluated it on a set of benchmarks from the prior work. The experimental results show that our approach significantly outperforms the state-of-the-art approaches thanks to the sophisticated abstract interpretation techniques.

CCS Concepts: • **Software and its engineering** → **Programming by example**; **Automatic programming**; • **Theory of computation** → **Abstraction**; **Program analysis**.

Additional Key Words and Phrases: Program Synthesis, Programming by Example, Abstract Interpretation

ACM Reference Format:
Yongho Yoon, Woosuk Lee, and Kwangkeun Yi. 2023. Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. *Proc. ACM Program. Lang.* 7, PLDI, Article 174 (June 2023), 25 pages. https://doi.org/10.1145/3591288

1 PROBLEM AND OUR APPROACH

Inductive program synthesis aims to synthesize a program that satisfies a given set of input-output examples. The popular top-down search strategy is to enumerate *partial programs* with missing parts and then complete them to a full program.

Though such a strategy is effective for synthesizing small programs, it hardly scales to large programs without being able to rapidly reject spurious candidates due to the exponential size of the search space.

Therefore, various techniques have been proposed to prune the search space [Feng et al. 2017; Gulwani 2011; Lee 2021; Polikarpova et al. 2016; Wang et al. 2017a]. In particular, abstract interpretation [Cousot 2021; Rival and Yi 2020] has been widely used for pruning the search space

"A key challenge in example-based program synthesis is the gigantic search space of programs. To address this challenge, various work proposed to use abstract interpretation to prune the search space.[...]

"The forward abstract interpretation computes possible outputs of a program given inputs, while the backward abstract interpretation computes possible inputs of a program given outputs.[...]

"We apply our approach to a standard formulation, syntax-guided synthesis (SyGuS), thereby supporting a wide range of inductive synthesis tasks."

Paper 4: MegaLibm (2024)

Implementation and Synthesis of Math Library Functions

IAN BRIGGS, University of Utah, USA
YASH LAD, University of Utah, USA
PAVEL PANCHEKHA, University of Utah, USA

Achieving speed and accuracy for math library functions like exp, sin, and log is difficult. This is because low-level implementation languages like C do not help math library developers catch mathematical errors, build implementations incrementally, or separate high-level and low-level decision making. This ultimately puts development of such functions out of reach for all but the most experienced experts. To address this, we introduce MegaLibm, a domain-specific language for implementing, testing, and tuning math library implementations. MegaLibm is safe, modular, and tunable. Implementations in MegaLibm can automatically detect mathematical mistakes like sign flips via semantic wellformedness checks, and components like range reductions can be implemented in a modular, composable way, simplifying implementations. Once the high-level algorithm is done, tuning parameters like working precisions and evaluation schemes can be adjusted through orthogonal tuning parameters to achieve the desired speed and accuracy. MegaLibm also enables math library developers to work interactively, compiling, testing, and tuning their implementations and invoking tools like Sollya and type-directed synthesis to complete components and synthesize entire implementations. MegaLibm can express 8 state-of-the-art math library implementations with comparable speed and accuracy to the original C code, and can synthesize 5 variations and 3 from-scratch implementations with minimal guidance.

CCS Concepts: • **Mathematics of computing → Numerical analysis**.

Additional Key Words and Phrases: Function approximation, libm, DSL, type-directed synthesis, e-graphs

ACM Reference Format:
Ian Briggs, Yash Lad, and Pavel Panchekha. 2023. Implementation and Synthesis of Math Library Functions. 1, 1 (November 2023), 28 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Mathematical computations in tasks as diverse as aeronautics, banking, scientific simulations, and data analysis are typically implemented as operations on floating-point numbers. The basic operators—addition, subtraction, multiplication, and possibly division, square roots, and fused multiply-adds—are typically provided by the hardware, but higher-level mathematical functions such as trigonometric or exponential functions are implemented in software in libraries such as libm. The speed and accuracy of these software libraries can have a dramatic impact on applications such as 3D graphics [Briggs and Panchekha 2022].

To maximize performance, math libraries are written in low-level languages like C; Figure 1 shows one example. Ensuring correctness and accuracy is thus challenging. These implementation languages cannot prevent mathematical errors such as mixing up signs or using the wrong

"Achieving speed and accuracy for math library functions like exp, sin, and log is difficult [...]

"[W]e introduce MegaLibm, a domain-specific language for implementing, testing, and tuning math library implementations.

"MegaLibm can express 8 state-of-the-art math library implementations with comparable speed and accuracy to the original C code, and can synthesize 5 variations and 3 from-scratch implementations with minimal guidance.[...]

"Unfortunately, determining equality for arbitrary real-valued expressions is known to be hard — dependent on unproven mathematical conjectures, and possibly undecidable."

Writing suggestions

Decidability
How does the system determine when a solution is valid?

Scalability
How complex can specifications be?
How large can generated programs be?
What subset of the language is targeted?
How long does synthesis take?

Practicability
How easy is it for users to express specifications?

Applicability
What range of problems might the system apply to?