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:

\[ \exists f. \forall \text{input. Specification} \]
When is program synthesis useful?

The problem

Efficiency in programming
(low-level code from high-level specifications)

Effective compilation
(e.g. superoptimization)

Program repair
(updating buggy programs to fit a specification)

Challenges

Deobfuscation
(restoring readability)

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

Program transformation
(updating programs as specifications evolve)

Reading
What is a specification?

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

<table>
<thead>
<tr>
<th>A logical specification</th>
<th>A type</th>
<th>An existing program</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x, y) \geq x \land f(x, y) \geq y$</td>
<td>$x : \mathbb{Z} \rightarrow y : \mathbb{Z} \rightarrow {z : \mathbb{Z}</td>
<td>z = \text{max}(x, y)}$</td>
</tr>
</tbody>
</table>

**Input-output examples**

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

**Natural language**

- “The larger of x and y”
The problem

Challenges

Reading

One approach: Syntax-Guided Synthesis (SyGuS)

logical formula

\[ f : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \]
\[ f(x, y) = f(y, x) \land f(x, y) \geq x \]

grammar (search space)

\[ T ::= x \mid y \mid 0 \mid 1 \mid \text{ITE}(C, T, T) \]
\[ C ::= T \leq T \mid \neg T \mid C \land C \]

Why is program synthesis hard?
The problem

**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

Type checking

Human inspection

Testing

\[ \Gamma \vdash e : \tau \]
Success in limited domains

Challenges

- Spreadsheet formulas
- Regular expressions
- Trigonometric functions
- Loop-free programs
- SQL queries
- Bit twiddling

Examples:
- X
- a(b | c) * d
- from t select *
  where
- x & 0xBEEF << y
Reading
“This survey is a general overview of the state-of-the-art approaches to program synthesis, its applications, and sub-fields. 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.

Online:
Oracle-Guided Component-Based Program Synthesis
Susmit Jha
UC Berkeley
jha@eecs.berkeley.edu
Sumit Goharnejad
Microsoft Research
sumitg@microsoft.com
Sanjits shahia
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 programs 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 heuristics [2, 9], automating repetitive programming tasks [33], 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-intrusive algorithms (e.g., [30]). 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 malware, and also due to the need for automated verification of programs. Current deobfuscation tools use heuristics and manually deobfuscate the resulting code [e.g., 28]. 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 [10, 11, 24]. With these approaches, one has the advantage of completeness: if such specifications are available, one can write a program through an interactive process, where 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.

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.”
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 Reasoning about Programs; 1.2.2 [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, thereby 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:

\[ \text{replicate} :: \text{Nat} \to \text{List} \to \text{List} \text{Nat} | \text{List} \to \text{Nat} \]
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 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.

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.”


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.

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

Challenges

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?