Program synthesis

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

Jeremy Yallop

jeremy.yallop@cl.cam.ac.uk

The program 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)
```

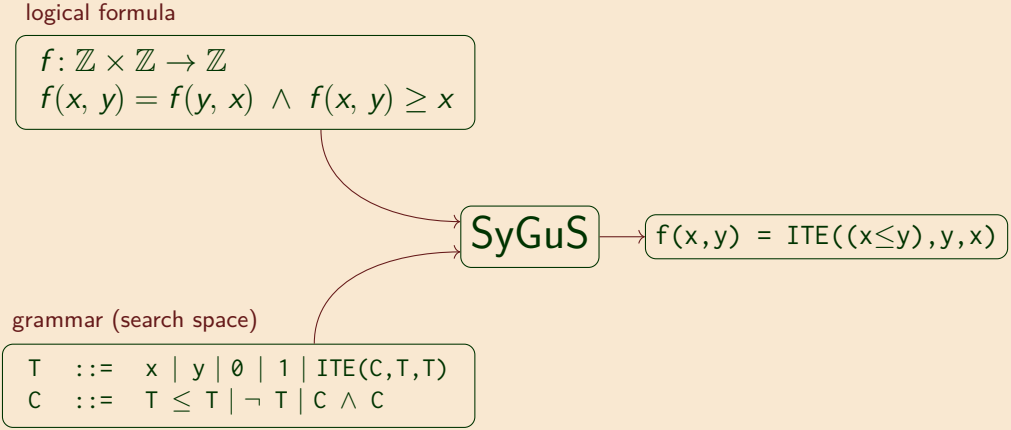
Input-output examples

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

Natural language

"The larger of x and y"

One approach: Syntax-Guided Synthesis (SyGuS)



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

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

SMT solving

Type checking

$\Gamma \vdash e : \tau$

Human inspection

Testing

✓

Success in limited domains

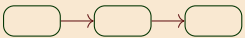
Spreadsheet formulas

Regular expressions

```
a(b|c)*d
```

Trigonometric functions

Loop-free programs



SQL queries

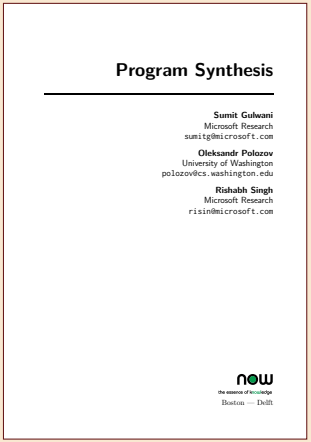
```
from t select *
where
```

Bit twiddling

```
x & 0xBEEF << y
```

Reading

Background reading: Program Synthesis



"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.
Foundations and Trends in Programming Languages,
vol. 4, no. 1-2, pp. 1–119, 2017.

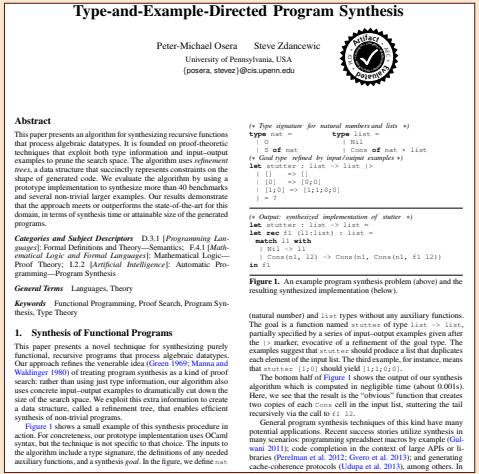
Online:
https://microsoft.com/en-us/research/wp-content/uploads/2017/10/program_synthesis_now.pdf

Paper 1: types and examples (2015)

The problem

Challenges

Reading



Type-and-Example-Directed Program Synthesis

Peter-Michael Osera Steve Zdancewic
University of Pennsylvania, USA
{posera, stevez}@cis.upenn.edu

Abstract

This paper presents an algorithm for synthesizing recursive functions that process algebraic datatypes. It is founded on proof-theoretic techniques that exploit both type information and input–output examples to prune the search space. The algorithm uses *refinement trees*, a data structure that succinctly represents constraints on the shape of generated code. We evaluate the algorithm by using a prototype implementation to synthesize more than 40 benchmarks and several non-trivial larger examples. Our results demonstrate that the approach meets or outperforms the state-of-the-art for this domain, in terms of synthesis time or attainable size of the generated programs.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Proof Theory; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program Synthesis

General Terms Languages, Theory

Keywords Functional Programming, Proof Search, Program Synthesis, Type Theory

1. Synthesis of Functional Programs

This paper presents a novel technique for synthesizing purely functional, recursive programs that process algebraic datatypes. Our approach refines the venerable idea (Green 1969; Manna and Waldinger 1980) of treating program synthesis as a kind of proof search: rather than using just type information, our algorithm also uses concrete input–output examples to dramatically cut down the size of the search space. We exploit this extra information to create a data structure, called a refinement tree, that enables efficient synthesis of non-trivial programs.

Figure 1 shows a small example of this synthesis procedure in action. For concreteness, our prototype implementation uses OCaml syntax, but the technique is not specific to that choice. The inputs to the algorithm include a type signature, the definitions of any needed auxiliary functions, and a synthesis *goal*. In the figure, we define `nat`

```
(* Type signature for natural numbers and lists *)
type nat =                    type list =
  | O                           | Nil
  | S of nat                    | Cons of nat * list
(* Goal type refined by input/output examples *)
let stutter : list -> list |>
  { []    => []
  | [0]   => [0;0]
  | [1;0] => [1;1;0;0]
  } = ?

(* Output: synthesized implementation of stutter *)
let stutter : list -> list =
let rec f1 (l1:list) : list =
  match l1 with
    | Nil -> l1
    | Cons(n1, l2) -> Cons(n1, Cons(n1, f1 l2))
in f1
```

Figure 1. An example program synthesis problem (above) and the resulting synthesized implementation (below).

(natural number) and `list` types without any auxiliary functions. The goal is a function named `stutter` of type `list -> list`, partially specified by a series of input–output examples given after the `|>` marker, evocative of a refinement of the goal type. The examples suggest that `stutter` should produce a list that duplicates each element of the input list. The third example, for instance, means that `stutter [1;0]` should yield `[1;1;0;0]`.

The bottom half of Figure 1 shows the output of our synthesis algorithm which is computed in negligible time (about 0.001s). Here, we see that the result is the "obvious" function that creates two copies of each `Cons` cell in the input list, stuttering the tail recursively via the call to `f1 l2`.

General program synthesis techniques of this kind have many potential applications. Recent success stories utilize synthesis in many scenarios: programming spreadsheet macros by example (Gulwani 2011); code completion in the context of large APIs or libraries (Perelman et al. 2012; Gvero et al. 2013); and generating cache-coherence protocols (Udupa et al. 2013), among others. In

"It is founded on proof-theoretic techniques that exploit both type information and input–output examples to prune the search space.

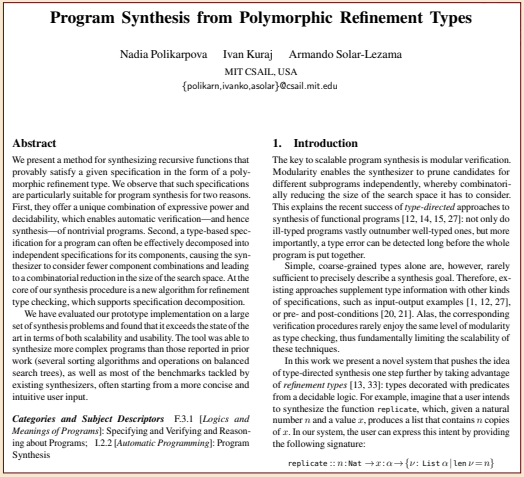
[...]

The goal is [...] partially specified by a series of input–output examples given after the $|>$ marker, evocative of a refinement of the goal type.

[...]

Our algorithm [...] uses the proof-theoretic idea of searching only for programs in β-normal, η-long form

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:

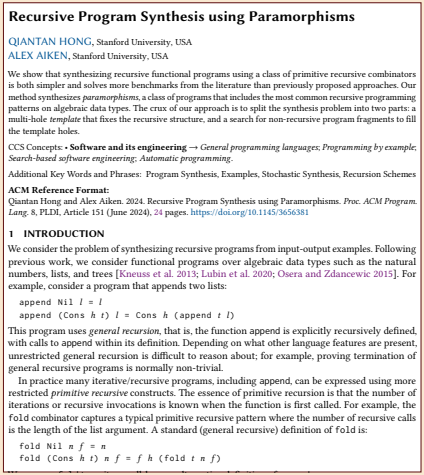
$$\texttt{replicate} :: n\!:\!\texttt{Nat} \to x\!:\!\alpha \to \{\nu\!:\texttt{List}\,\alpha \mid \texttt{len}\,\nu = n\}$$

"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: paramorphisms (2024)



Recursive Program Synthesis using Paramorphisms

QIANTAN HONG, Stanford University, USA
ALEX AIKEN, Stanford University, USA

We show that synthesizing recursive functional programs using a class of primitive recursive combinators is both simpler and solves more benchmarks from the literature than previously proposed approaches. Our method synthesizes *paramorphisms*, a class of programs that includes the most common recursive programming patterns on algebraic data types. The crux of our approach is to split the synthesis problem into two parts: a multi-hole *template* that fixes the recursive structure, and a search for non-recursive program fragments to fill the template holes.

CCS Concepts: • **Software and its engineering** → *General programming languages*; *Programming by example*; *Search-based software engineering*; *Automatic programming*.

Additional Key Words and Phrases: Program Synthesis, Examples, Stochastic Synthesis, Recursion Schemes

ACM Reference Format:
Qiantan Hong and Alex Aiken. 2024. Recursive Program Synthesis using Paramorphisms. *Proc. ACM Program. Lang.* 8, PLDI, Article 151 (June 2024), 24 pages. https://doi.org/10.1145/3656381

1 INTRODUCTION

We consider the problem of synthesizing recursive programs from input-output examples. Following previous work, we consider functional programs over algebraic data types such as the natural numbers, lists, and trees [Kneuss et al. 2013; Lubin et al. 2020; Osera and Zdancewic 2015]. For example, consider a program that appends two lists:

```
append Nil l = l
append (Cons h t) l = Cons h (append t l)
```

This program uses *general recursion*, that is, the function append is explicitly recursively defined, with calls to append within its definition. Depending on what other language features are present, unrestricted general recursion is difficult to reason about; for example, proving termination of general recursive programs is normally non-trivial.

In practice many iterative/recursive programs, including append, can be expressed using more restricted *primitive recursive* constructs. The essence of primitive recursion is that the number of iterations or recursive invocations is known when the function is first called. For example, the `fold` combinator captures a typical primitive recursive pattern where the number of recursive calls is the length of the list argument. A standard (general recursive) definition of `fold` is:

```
fold Nil n f = n
fold (Cons h t) n f = f h (fold t n f)
```

"Our method synthesizes *paramorphisms*, a class of programs that includes the most common recursive programming patterns on algebraic data types.

[...]

The paramorphism combinator on lists is:
para Nil g_{Nil} $g_{\text{Cons}} = g_{\text{Nil}}$
para (Cons h t) g_{Nil} $g_{\text{Cons}} = g_{\text{Cons}}$ h $(t,$ para t g_{Nil} $g_{\text{Cons}})$

[...]

We have shown by experiment that an implementation of our approach is able to synthesize all the problems handled by the current state of the art as well as substantially harder problems."

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?