Design and Applications of a Program Synthesizer

Candidate Number: 1076461

University of Oxford

Project Report

Honour School of Computer Science, Part B

Trinity 2025

Abstract

Program Synthesis is the field of programs which write programs, usually with a formal logical specification. In this project, I focus on designing and building a PBE (Programming-by-Example) Synthesizer, with the primary goal of being able to deduce from a piece of data (such as, a set or sequence of integers), a program which might have generated it, or which can explain most of its content. Synthesizers of this sort are useful for pattern recognition in settings where datasets are small and approximate answers are not desired. I use a combination of enumerative and Montecarlo techniques, and discuss the practical and theoretical implications of different possible design choices. The resulting tool is very general, and can be used to evaluate and compare the expressivity of programming languages, or to try to determine the Kolmogorov Complexity (or Minimum Description Length), of different sequences in any (total) programming language. I discuss the results of some such experiments, and give outlines of several other similar potential future experiments.

Contents

1 Introduction 4
1.1 Goals & Motivation 5
1.2 Report Structure 5
1.3 Choice of Technology 6
2 Program Evaluation 7
2.1 Functional Terms 7
2.2 Functional Languages 8
2.3 Term Reduction 10
3 Enumeration 12
Bibliography 13

1 Introduction

Program synthesis is the problem of generating a program from a certain specification, often expressed as a logical constraint. The problem we consider is one of Programming-by-Example synthesis, where instead of a formal specification, we attempt to generate a program using input-output pairs. This is useful in situations where we are interested in discovering exact, possibly complex patterns in a piece of data. This method has strong limitations, as you might expect, but has the advantages that it can work even when datasets are far too small to use statistical methods, that we make very few assumptions about the data (beyond the fact that it has some computable pattern), and that once we generate a program, we can examine and completely understand its behaviour.

Some notable applications of this kind of PBE synthesis include:

- "Flash Fill" [1]: This is the technology behind Microsoft Excel's autocomplete feature. It is what allows the software to detect and extend a user's actions using very few examples (often even just one).
- "IntelliCode" [2]: This is a feature of Visual Studio Code which monitors a user's actions and attempts to discover the patterns of user's edits, and then suggests further refactors (usually only requiring 2 examples).
- Query Synthesis [4]: PBE has been used to develop tools which construct SQL queries based on small numbers of tuple examples of rows which should be fetched.[1]

PBE problems are, by their nature, underspecified, as there will usually be many (even infinitely many) programs satisfying any set of input-output constraints, but it does allow us to generate reasonable conjectures about the underying structure of any given piece of data. It has the very strong advantage that we do not have to write any kind of formal specification for our program, which is often not much easier than writing the program itself.

[1] This is not the same as Query-by-Example, or QBE, which is a feature of many databases which provides a more user-friendly querying interface, and which has existed since the 1970s.

1.1 Goals & Motivation

Although the tools developed throughout this project are very general and could be applied to many different areas, we focus especially on synthesizing programs generating integer sequences. One motivating application of this could be in mathematical research, where we might wish to guess patterns in structured integer sequences. This is exactly the concern that the Online Encyclopedia of Integer Sequences (OEIS) was invented to address, and we will use its database to evaluate our program. However, we also discuss how these tools could be applied elsewhere.

We also focus on synthesizing small solution programs. Firstly, because this allows us to avoid overfitting (for example, programs which simply match their inputs to the given input-output pairs, without giving any insight into the larger pattern). Secondly, because it will be helpful in eliminating redundant programs (i.e., programs which are syntactically distinct but semantically identical), vastly reducing our search space. And thirdly, because the minimum description length of a mathematical object or piece of data (i.e., the length of the shortest program generating it) is known as its Kolmogorov Complexity, and though it is incomputable, it is of theoretical interest[2].

Of course, we cannot hope to produce a perfect synthesizer, so a great deal of effort has gone into expanding the space of programs we can search. This equates to (1) trying to shrink the search space as much as possible, and (2) expanding the portion of the search space we can feasibly examine as much as possible. This means both pruning our search wherever possible and emphasizing performance in our code, both of which come with the cost of some added complexity in our synthesizer.

1.2 Report Structure

Each of the main chapters of this report covers a different aspect of this program synthesizer, in varying levels of detail. We discuss the reasons behind each major design decision, as well as their strengths and limits compared to other possible decisions.

The main chapters cover:

- The interpreter used to evaluate programs.

[2]Notably, there are theoretical results showing that if we could compute Kolmogorov complexity, we could approximate Algorithmic Probability, which in turn allows us to compute the source of an infinite sequence correctly and with relatively low error. This formalizes our intuition that, when looking at a piece of data from an unknown source, a "simple" program is more likely to have generated it than a "complex" one. See [5] for more information.

- The enumerator which allows us to iterate over certain program spaces.
- The incorporation of semantic analysis into our search.
- The use of the Metropolis-Hastings algorithm to expand the scope of our search space.
- The results and effectiveness of the synthesizer.

1.3 Choice of Technology

Because of the need for efficiency, I chose to use Rust to implement this synthesizer. This did complicate its implementation, it allowed me to make optimizations which would not have been possible in a higher level language (such as Haskell, which is probably the language which would have made a simple version of this implementation the simplest).

In order to be clear and precise, I try wheverever possible to show the relevant code, but for the sake of brevity and readability, I omit parts of the code which add complexity without offering any insight (type cases, clones, trait derivations, unreachable code branches, etc...), meaning code snippets as they appear in this document may not be strictly correct[3]. Wherever library types are used, I explain their purpose, without delving into their meaning or implementation. However, if you would like more information, see `https://doc.rust-lang.org/std`.

[3]If you are unfamiliar with Rust, the code as shown should be readily comprehensible. If you are familiar with Rust, please ignore any ownership or borrow checker violations.

2 Program Evaluation

The first major design decision we must make is what kinds of programming languages we will consider. Throughout this project, we will only consider simply typed functional languages without any advanced features (such as pattern matching, exceptions, type constructors, etc...). This is because their simplicity makes them much easier to define, implement, analyze, and because they share a common grammar, which allows us to enumerate programs much more easily. What this means is that a language's behaviour should defined solely by the builtin primitive constants it provides. As we will see later on, this limitation will actually turn out to be a powerful tool in defining our search space more precisely.

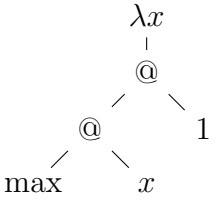
2.1 Functional Terms

We define expressions in such a language as follows:

```
pub type Thunk = Rc<RefCell<Term>>; // A pointer to a mutable term
pub type Value = Rc<dyn TermValue>; // A pointer to a value

pub enum Term {
    Val(Value), // A primitive value
    Var(Identifier), // A variable identifier
    Lam(Identifier, Rc<Term>), // A lambda abstraction
    App(Thunk, Thunk), // An application of one term on another
    Ref(Thunk), // Transparent indirection to another term
}
```

The `Value` type stores a pointer to a value, whose type has been erased (similar to how Haskell erases all type information at runtime). The `Ref` variant is simply a transparent pointer to another term, which will be useful during term reduction. As an example, the term $(\lambda x.\max(x)(1))$ would be represented as follows (omitting pointers and `Ref`s, and representing application by @):



In order to simplify this syntax in our code, define a `term!` macro which parses these terms at compile-time. It allows us to write in a more familiar Haskell-style syntax, and insert variables and constants into terms:

```
// A pure lambda-term
let apply = term!(f x -> f x x);

// Inserting a term stored in a variable into a template.
let square = term!([apply] multiply);

// Literals are parsed as values
let two = term!((a b -> a) 2 "x");

// Brackets starting with a colon indicate a variables
// should be parsed as a value.
let two = 2;
let four = term!([square] [:two]);
```

2.2 Functional Languages

As stated earlier, in their most basic form, our languages are determined by the primitives they offer, each of which will be annotated with a `Type`. Again, we will only consider simply-typed programs, meaning we do not allow any kind of polymorphism.

```
pub enum Type {
    Var(Identifier), // A base type
    Fun(Rc<Type>, Rc<Type>), // A function type
}
```

In order to evaluate a term, we will have to define an environment in which to run in. We do this by defining a `Language` trait (an interface, in other languages) with a method to construct the `Context` terms will be evaluated in. We also provide a `Builtin` type and a `builtin!` macro to simplify the definition of primitives, both of whose definitions we omit from this report. A simple example we will revisit several times is the language of polynomials with positive integer coefficients:

```
// Polynomials is a data structure with no fields
pub struct Polynomials;

impl Language for Polynomials {
  fn context(&self) -> Context {
    let plus = builtin!(
        N => N => N
        |x, y| => Term::val(x.get::<i32>() + y.get::<i32>())
    );
```

```
    let mult = builtin!(
        N => N => N
        |x, y| => Term::val(x.get::<i32>() * y.get::<i32>())
    );

    // Constants, which don't take any arguments
    let one = builtin!(
        N
        | | => Term::val(1i32)
    );

    let zero = builtin!(
        N
        | | => Term::val(0i32)
    );

    //Mapping from Identifiers to builtins
    Context::new(&[
      ("plus", plus),
      ("mult", mult),
      ("one",  one ),
      ("zero", zero),
    ])
  }
}
```

The `Term::val` function converts its argument into a `Term` by converting it into a `Value`. The `Term::get` method casts a `Term::Val` into a given type (which can never fail if our program is well-typed). It's worth noting that these primitives are strict in all their arguments. We can get around this by reducing to a projection term instead of taking extra arguments:

```
// ifpos c t e = if (c) { t } { e }
// Lazy in 't' and 'e'
let ifpos = builtin!(
  Bool => N => N => N
  |c| => if c.get::<bool>() {
    term!(t e -> t)
  } else {
    term!(t e -> e)
  }
)
```

2.3 Term Reduction

The implementation we use is essentially the graph reduction technique described in *The Implementation of Functional Programming Lanugages* [3]. This allows for laziness and shared reduction and is performant enough for our purposes, but more sophisticated (even optimal) algorithms exist.

To evaluate a term, we reduce it until we reach a weak head normal form (WHNF). That is, either a lambda abstraction or a primitive function applied to too few arguments. Our reduction strategy is based on *spine reduction*. We traverse the term's leftmost nodes top-down until we reach the *head* of the term (the first subterm which is not an application). If the head is an application of a λ-abstraction to an argument, we perform a *template instantiation* operation, substituting a reference to the argument in place of the parameter everwhere it appears in the body of the lambda term (this is where the `Ref` variant is useful). If the head is a variable, we look it up in our context, and (if it exists), check if it is applied to enough arguments to invoke its definition. If so, we evaluate all of its arguments (hence the strictness of primitives) and replace the subnode at the appropriate level with the result. We continue until we perform no more reductions. A simplified version of the interpreter's main code is shown below.

```
enum CollapsedSpine {
    // If spine is in weak head normal form
    Whnf,
    // A built-in function & a stack of arguments
    Exec(BuiltIn, Vec<Thunk>),
}

impl Context {
  // Caller function ignores output of collapse_spine
  pub fn evaluate(&self, term: &mut Term) {
    self.collapse_spine(&mut term, 0);
  }

  //The depth is the number of arguments along the spine, so far
  pub fn collapse_spine(
    &self,
    term: &mut Term,
    depth: usize
  ) -> CollapsedSpine {
    match term {
      Ref(r) => self.collapse_spine(r, depth),
      Val(_) | Lam(_, _) => Whnf,
```

```
      Var(v) => match self.lookup(v) {
        // If head is a variable takes no arguments, once again,
        // replace it and continue,
        Some(builtin) if builtin.n_args == 0 => {
          *term = builtin.func(&[]);
          self.collapse_spine(term, depth)
        },
        // If we have enough arguments to apply this function,
        // we start building a stack of arguments.
        Some(builtin) if builtin.n_args <= depth {
          Exec(builtin, vec![])
        }
        // If we do not have enough arguments, we are in WHNF
        _ => Whnf,
      }
      App(l, r) => match self.collapse_spine(l, depth + 1) {
        Exec(builtin, mut args) => {
          args.push(r);
          //If we have enough arguments, apply the function
          if args.len() == builtin.n_args {
            // The args will be in reverse order
            args.reverse();
            for arg in &mut args {
              self.evaluate(arg);
            }
            // Call function & continue
            *term = builtin.func(&args);
            return self.collapse_spine(term, depth);
          }
          // If we do not have enough arguments, keep pushing
          // onto the stack of parameters.
          Exec(builtin, args)
        }
        // Template instantiation
        Whnf => if let Lam(arg, body) = l {
          *term = body.instantiate(arg, r);
          self.collapse_spine(term, depth)
        } else {
          Whnf
        }
      }
    }
  }
}
```

3 Enumeration

Bibliography

[1] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. Program synthesis. *Foundations and Trends® in Programming Languages*, 4(1-2):1–119, 2017.

[2] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. On the fly synthesis of edit suggestions. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[3] Simon L Peyton Jones. *The implementation of functional programming languages (prentice-hall international series in computer science)*. Prentice-Hall, Inc., 1987.

[4] Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. Discovering queries based on example tuples. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014.

[5] Ray J Solomonoff. Algorithmic probability: Theory and applications. *Information theory and statistical learning*, pages 1–23, 2009.