Design and Applications of a Program Synthesizer

Candidate Number: 1076461

University of Oxford

Project Report

Honour School of Computer Science, Part B

Trinity 2025

Word Count: X

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

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 (PBE) 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:

1. "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).
2. "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).
3. Query Synthesis [3]: 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 powerful property 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.

1.1 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

[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.

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 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:

1. The interpreter used to evaluate programs.
2. The enumerator which allows us to iterate over certain program spaces.
3. The incorporation of semantic analysis into our search.
4. The use of the Metropolis-Hastings algorithm to expand the scope of our search space.
5. 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

[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 [4] for more information.

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

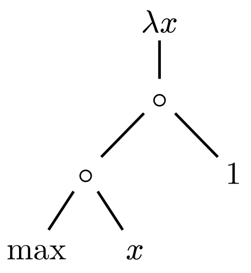
The first major design decision we must make is what kinds of programming languages we will consider. Throughout this project, we will only consider simply typed functional languages without any advanced features (such as pattern matching, exceptions, type constructors, etc…).[4] This is because their 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 λ-term $\lambda x.\max(x)(1)$ would be represented as follows (omitting pointers and `Ref`s, and denoting application by $\circ$):



We define the **size** of a term as the number of nodes in this tree. For example, the term in the figure above has size 6.

[4] Anyone unfamiliar with the basics of Lambda Calculus should see Section A.1 for some important definitions.

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* [5]. 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 Program Enumeration

The core of our synthesizer is its enumeration algorithm, which takes as input a language, a size and a type, and enumerates the β-normal terms of that size with that type in that language. The restriction to β-normal terms is useful because:

1. Every normalizable λ-term has a unique normal form, so we never generate two β-equivalent terms.[5]
2. Every typable λ-term is normalizable, so we don't reduce the expressibility of our language by only considering β-normal terms.
3. It vastly reduces our search space.

The term enumerator is the most performance-critical part of the code, and so has been rewritten several times with increasing complexity to reach its current level of performance. Because of this, we include very little code in this section, and focus on the high level approach.

3.1 Basic Algorithm

Our enumeration method is based on the observation that every λ-term in β-normal form has the following structure:

$$\lambda x_1 \cdots x_n . v t_1 ... t_m$$

where v is a variable and $\{t_i\}$ are also in β-normal form.

This suggests a recursive algorithm, where, given some type T, we:

1. Enumerate over the variables v which can appear as the head of a term of type T (i.e, variables with types of the form $A_1 \Rightarrow \cdots \Rightarrow A_k \Rightarrow T$, where $k \geq 0$). We then enumerate all β-normal terms of types $\{A_i\}$, and apply v to each combination of them.
2. If $T \equiv A \Rightarrow B$, then we also enumerate the terms $\lambda x.M_i$ of type B, where $\{M_i\}$ are β-normal terms which may include some fresh variable x of type A.

This corresponds to the following set of rules, which types exactly the β-normal terms in any context Γ.

[5] However, we may still generate terms which are equivalent in a particular language (for example $(\lambda xy.(+)xy)$ is equivalent to $(\lambda xy.(+)yx)$) if $(+)$ has its usual definition. We might also generate several η-equivalent terms. Both of these concerns can be addressed using the semantic analysis tools discussed in the next chapter.

$$\frac{\{\Gamma; x : A\} \vdash b : B}{\Gamma \vdash (\lambda x.b) : A \Rightarrow B}\text{(Abs)}$$

$$\frac{n \geq 0 \quad v : (A_1 \Rightarrow \cdots \Rightarrow A_n \Rightarrow T) \in \Gamma \quad \Gamma \vdash a_i : T_i}{\Gamma \vdash v a_1 \cdots a_n : T}\text{(App)}$$

Since we only want to generate terms of a fixed size, at every point in the enumeration, we must make sure to keep track of how large the term is so far, so that we can backtrack whenever it gets too large.

3.2 Caching

While the above technique could be implemented quite simply, it is not easy to attain high performance. The most important optimizations we can make are those which prune the search space as early as possible. Even costly optimizations of this sort will usually save a lot of time. There are many possible techniques of this sort which could be used, but we discuss only the two most simplest and most important optimizations:

3.2.1 Query Pruning

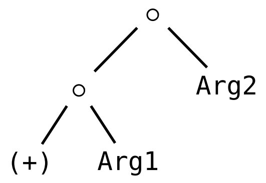
A search (or enumeration) query in a particular language is defined by the type and size of the enumerated terms. We can maintain a cache with the results of previously made queries.

```
// A search query
type Query = (Type, usize);

// A map from queries to results
type PathCache<L: Language> = HashMap<Query, SearchResult<L>>;

// Since some queries have large results, we place a limit on how many terms we
can
// can store in a single cache entry.
pub const CACHE_SIZE_LIMIT: usize = 16;

// The result of a query
pub enum SearchResult<L: Language> {
    Unknown, // If the search is still in process
    Inhabited {
        // The first few Terms output by this query
        cache: Vec<Term>,
        // The number of terms that have been found
        // (may be more than those that have been cached)
        count: usize,
        // The state of the search after the cached terms have been enumerated.
        state: Option<Box<SearchNode<L>>>,
    },
    Empty, // If the search does not yield any terms
}
```

Whenever we begin a new search, we consult the cache to see if this query has been made before. If so, we either skip the search (if it's `Empty`), or use the cached values (before picking up the search at the point where the previous search ran out of space in the cache). This is extremely useful, since many queries yield empty results, or otherwise have few results, even if the search would otherwise be very large.

3.2.2 Argument Pruning

When we use the Application rule, we have to consider every combination of argument sizes. For example, if we are trying to generate a term of type `N` and size k by applying arguments to the function `(+) : N => N => N`, then we will reach a point in the search where we have the following search tree:



Here, `Arg1` and `Arg2` may have any of the $k-4$ combinations of sizes adding up to $k-3$. If we instead used a function taking n arguments, there would be $O(k^{n-1})$ possible size partitions. We would benefit from pruning this search space, so that we don't begin to enumerate all the possible values of one argument before realizing we have chosen an empty size partition.

It is a bit difficult to express this algorithm in (pseudo-)code without going into excessive detail about the implementation of the enumeration algorithm, but we can again take advantage of our previous cache, and prune any search paths which correspond only to partitions which do not yield any output terms. This allows us to search only partitions where the result of the search for each argument is either `Unknown` or `Inhabited`.

3.2.3 Caching Considerations

There are a few more considerations we have to take into account to get our implementation correct. For example:

1. When we abstract over a variable, we may invalidate previous cache entries. For example, a certain language may not have any variables of type T, so if we run the query $(T, 1)$, we will mark it empty in the cache, but if we later make the

query $(T \Rightarrow T, 2)$, then can find the term $\lambda x.x$, which has a subterm (that is, x) of type T and size 1, which contradicts our cache entry.

2. When we begin a search for the first time, we mark it as `Unknown` in the cache. Later, when we complete it, we mark it `Inhabited` or `Empty`. We may select a partition for the Application which includes a search with an `Unknown` result for a certain paramter, which might turn out to be `Empty`. When this happens, we have to be careful to close ongoing searches properly, which requires care. Similarly, we must be careful when entering a search using a state from an `Inhabited` search result.

4 Semantic Analysis

A Appendix

A.1 Lambda Calculus Primer

1. A **$\boldsymbol{\lambda}$-term** is either:
   - a variable,
   - an abstraction over another λ-term (i.e., $\lambda x.M$ where M is a λ-term)
   - an application of two λ-terms (i.e., (MN) where M, N are λ-terms)
2. The **subterms** of a λ-term are all the λ-terms which appear within it (including itself).
3. A **redex** (from reducible expression) of a λ-term is a subterm of a λ-term of the form $(\lambda x.M)N$.
4. An occurence of a variable v in a λ-term t is **bound** if it appears in a subterm $t = \lambda v.u$. Otherwise, it is **free**. We say a variable is **fresh** if it does not appear in any of the λ-terms under discussion.
5. Terms are **$\boldsymbol{\alpha}$-equivalent** if they are equal up to the renaming of bound variables.
6. **$\boldsymbol{\beta}$-reduction** is the operation mapping a redex $(\lambda x.M)N$ to $M[N/x]$ (that is, which substitutes N for x in the usual way, accounting for variable collisions). When we say a λ-term **$\boldsymbol{\beta}$-reduces** to another, this may require more than several reduction steps.
7. Two λ-terms are **$\boldsymbol{\beta}$-equivalent** if they β-reduce to a common term (up to α-equivalence).
8. **$\boldsymbol{\eta}$-reduction** is the operation mapping a term $(\lambda x.Mx)$ to M. η-equivalent terms are not necessarily β-equivalent.
9. A λ-term is in **$\boldsymbol{\beta}$-normal form** if it does not contain any redexes. When it exists, the β-normal form is unique and the same for all β-equivalent λ-terms.
10. All λ-terms have the form $\lambda x_1 \cdots x_k.t_1 \cdots t_m$ where $\{x_i\}$ are variables, $\{t_i\}$ are λ-terms, $k \geq 0$, and $m \geq 1$. We say t_1 is the **head** of the λ-term.
11. When the head of a λ-term is a variable, it is in **head normal form**.

Bibliography

[1] S. Gulwani, O. Polozov, R. Singh, and others, “Program synthesis,” *Foundations and Trends® in Programming Languages*, vol. 4, no. 1–2, pp. 1–119, 2017.

[2] A. Miltner *et al.*, “On the fly synthesis of edit suggestions,” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019, doi: 10.1145/3360569.

[3] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik, “Discovering queries based on example tuples,” *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, 2014, [Online]. Available: https://api.semanticscholar.org/CorpusID:6440703

[4] R. J. Solomonoff, “Algorithmic probability: Theory and applications,” *Information theory and statistical learning*, pp. 1–23, 2009.

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