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: 4999

Abstract

Program Synthesizers are programs that write programs, usually with a formal logical specification. In this project, we 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 that 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. We use a combination of enumerative and Monte Carlo 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 Minimum Description Length of different sequences in (total) programming languages. We discuss the results of some such experiments, and discuss potential future extensions.

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 underlying 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. Secondly, because it will be helpful in eliminating redundant programs, 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 interesting programs that we can search. This equates to narrowing the search as much as possible and expanding the portion of the search space we can feasibly examine as much as possible. This means we should both prune our search wherever possible and emphasize performance in our code, both of which require 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 program enumeration algorithm.
3. The incorporation of semantic analysis into our search.
4. Different kinds of encoding schemes.
5. The use of the Metropolis-Hastings algorithm to expand the scope of our search space.
6. The 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 complicated the implementation, but allowed me to make optimizations which would not have been possible in a higher level language (such as Haskell, which probably would have made building a prototype simpler).

[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 few errors. 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.

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 casts, clones, trait derivations, unreachable code branches, etc…), meaning code snippets as they appear in this document may not be strictly correct.[3]

[3]In particular, it may include Rust-specific errors, such as ownership/borrow checker violations.

2 Program Evaluation

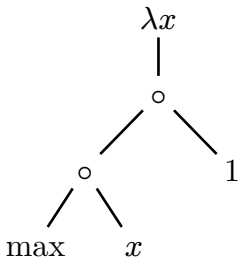
Throughout this project, we will only consider simply typed (total) functional languages without any advanced features (such as pattern matching, type constructors, etc…).[4] These languages are easy to define, implement, and analyze, and the fact that they have a common grammar allows us to enumerate programs easily. What this means is that a language's behaviour should defined solely by the builtin primitives it provides (See Section 2.2 for an example).

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,
                // (useful for implementing shared reduction)
}
```

The `Value` type stores a pointer to a value, whose type has been erased (similar to how Haskell erases all type information at runtime). 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.

In order to simplify our code, define a `term!` macro which parses terms at compile-time. This allows us to write in a more familiar Haskell-style syntax:

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

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

```
// 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`.

```
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 the environment to run it 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:

```
// ifcte c t e = if (c) { t } { e }
// Lazy in `t' and `e'
let ifcte = 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 λ-abstraction or a primitive function applied to too few

arguments. Our reduction strategy is based on *spine reduction*. We traverse the term until we reach its *head* (the first subterm which is not an application), if this is an application of a λ-abstraction to an argument, we perform *template instantiation*, substituting a reference to the argument in place of the parameter wherever 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 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 with the result of the invocation. We continue until we perform no more reductions. A simplified version of the interpreter's core 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:

$$\lambda x_1 \cdots x_n.vt_1...t_m$$

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 $(\lambda xy.(+)yx)$), or which are η-equivalent. Both of these concerns can be addressed using semantic analysis.

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

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

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 is fairly straightforward, it is difficult 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 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.

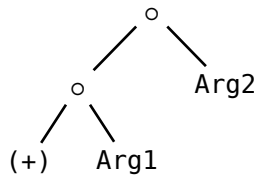
```
// A search query
type Query = (Type, usize);

// 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 store in a single cache entry.
pub const CACHE_SIZE_LIMIT: usize = 16;

// The result of a search 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 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 search queries yield few (if any) results, even if the search would be very dificult.

3.2.2 Argument Pruning

When we use the (App) 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, even with a simple function with only 2 arguments, `Arg1` and `Arg2` may have any of the $k-4$ combinations of sizes adding up to $k-3$. 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 unsatisfiable size partition.

Here, with some modifications, 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 (App) which includes a search with an `Unknown` result for a certain parameter, which might turn out to be `Empty`. When this happens, we have to be careful to end 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

While the enumerator may be sufficient to find simple programs in small program spaces, we can massively shrink the program space by performing some semantic analysis. This feature is optional and language-specific. We can do this by building up a 'Canonical' representation of the semantics of each subexpression in any given program. This way, we can enumerate only programs with distinct semantics. This also speeds up the enumeration, since we can also avoid search paths which repeat the same semantics more than once. For example, we might want to detect that the programs `(+) x y` and `(+) y x` are the same, and therefore our enumerator should output one of these, but not both.

4.1 Semantic Analysis Interface

Since these semantics are language-specific, we define a `Language` trait (or interface) with methods defining the language's `Context` and semantics:

```
pub trait Language: Sized + Clone + Debug {
  // The type of the semantic representation of programs of this
language
  type Semantics: Semantics + Sized;

  fn context(&self) -> Context; // The primitives this Language provides

  // Semantics of a variable (with type annotations)
  fn svar(&self, var: Identifier, ty: &Type) -> Analysis<Self>;
  fn slam( // Semantics of a lambda abstraction
      &self,
      ident: Identifier, // Variable being abstracted over
      body: Analysis<Self>, // Semantics of function body
      ty: &Type, // The type of the lambda abstraction
  ) -> Analysis<Self>;
  fn sapp( // Semantics of an application
      &self,
      fun: Analysis<Self>, // The function being applied
      arg: Analysis<Self>, // The argument to the function
      ty: &Type, // The type of the application
  ) -> Analysis<Self>;
}

pub enum Analysis<L: Language>
where
    L::Semantics: Semantics,
{
```

```
    Malformed, // Reject Term entirely (i.e, unnecessarily complex)
    Unique,    // Allow, but do not construct canonical form
    Canonical(L::Semantics), // Group into equivalence class by
canonical form
}
```

As we build up the program, we also build up its semantics using the semantics of its subterms.[6] Since the semantics of each subterm depend only on the semantics of its own subterms, we can ensure that any two terms with the same semantics (and the same type) are interchangeable, and we don't lose any expressivity by not repeating terms with the same semantics.[7] Since we shouldn't expect to be able to able to perfectly analyze every program, we include the `Unique` variant, which allows us to indicate that a certain term should be treated as the sole term of its equivalence class. We also include a `Malformed` variant, which allows us to indicate that a term should not be included in our search at all.

4.2 Semantics of Polynomials

As a simple example, we revisit the `Polynomials` language. Since the programs expressible here are the functions mapping to polynomials with coefficients in $\mathbb{N}^+$, we can convert to the following form:

$$\lambda x_1 \cdots x_n.(a_0 + a_1(v_{11}v_{12}\cdots) + a_2(v_{21}v_{22}\cdots) + \cdots)$$

This leads to the `PolySem` data structure:

```
// Semantics of a term (a term taking zero or more arguments and
// returning a polynomial)
pub struct PolySem {
  arguments: Vec<Identifier>,
  polynomial: Sum
};

// A sum of products (i.e., a polynomial), with a constant shift
pub struct Sum(i32, Vec<Product>);

// A product of terms, with a constant scaling factor
pub struct Product(i32, Vec<Identifier>);
```

We can translate any expression into this form by expanding polynomials, though we must be careful regarding variable collisions (especially with primitives). In order

[6] We are defining a *fold* (or catamorphism) from programs of a certain language (with type annotations) to their canonical semantic representations.

[7] This is based on the idea of *Contextual Equivalence*: If two terms S, T, have the same semantics, then for any context $C[X]$, $C[S]$ should have the same semantics as $C[T]$. Essentially, we are forbidding introspection.

to ensure that this representation is unique, we can sort our variables & monomials lexicographically. As you might expect, this greatly shrinks the space of programs we are considering.

Term Size (`N => N`)	Number of terms (no analysis)	Number of terms (analysis)
2	3	3
6	18	4
10	29	8
50	677	249

Polynomials are simple enough to be analyzed easily and exactly (i.e., we have semantic equality between two polynomial terms iff their analysis yields α-equivalent `PolySem`s). We cannot hope to do this generally, but even in more complex languages, we can still greatly reduce our search space by informing the enumerator of simple equivalences.

5 Encoding Schemes

The tools we have developed for narrowing our search space thus far may seem primitive, but with some care, we can exert greater control over the search space.

5.1 Query Selection

The most obvious way to restrict the enumeration is by modifying the type of the program we are enumerating, and the context we feed into it. For example, considering the `Polynomials` language and an integer sequence $\{a_n\}$, we could choose to synthesize:

- A mapping `f : N => N` from $n \mapsto a_n$.
- A function `p : N => N => N` which should be iterated: $a_{n+1} = p(a_n)$, using the given value of a_0 (For example, if $a_n = \sum_{k=0}^{n} k$).
- Arguments `z` and `f` to the natural fold (or catamorphism) on the (Peano) naturals, defined as:

```
foldNat : N => N => (N => N => N) => N
foldNat Zero     z _ = z
foldNat (Succ n) z f = f (Succ n) (foldNat n z f)
```

- A function `p : (N => N) => N => N` whose fixed point is $n \mapsto a_n$. In this case, our program `p` essentially acts as a recursive definition, akin to how recursive functions are defined in denotational semantics: $a_n = f^{\infty}(n)$ where $f^n(n) = p(f^{n-1}, n)$.
- Many other possibilities.

Additionally, there are theoretical results about the expressiveness of programs of different types. For example, if we were concerned with sequences of objects of a finite type `A`, we could provide an `equal? : A => A > Bool` primitive and restrict ourselves to the Regular Languages by constructing terms of type `(A => Q => Q) => Q => Q` (where `Q` is a type encoding the states of a DFA recognizing the language). Alternatively, we could restrict ourselves to PTIME, PSPACE, k-EXPTIME, or k-EXPSPACE (for any k).[8]

5.2 Grammars Through Types

As mentioned previously, our interpreter erases all type information at runtime. What this means is that the type system we provide only serves to indicate to the enumerator where each function is syntactically valid. This means that we can take advtantage of this to specify a precise grammar for our language, beyond the basic restriction of it being a functional language.

[8] See [6] or [7] for more.

For example, we might want to search the space of *conjunctive queries* from database theory, which form a powerful language with simple semantics:

$$\lambda x_1 \cdots x_n . \exists v_1 \cdots v_m \bigwedge_i P_i$$

where P_i are atomic formulae. These could be defined with the following simple grammar:

$$\begin{array}{lll} Q \to \lambda v Q & B \to \exists v B & C \to P \wedge C \\ Q \to B & B \to C & C \to P \end{array}$$

For any such rules, we can construct primitives with corresponding types, so that program enumeration corresponds to searching the grammar:[9]

```
// Helper functions to unwrap terms
let int = |t: &Term| t.get::<u32>();
let bln = |t: &Term| t.get::<bool>();

// B -> C
let boolean = builtin! {
  Conjunction => Boolean
  |c| => c.clone()
};

// B -> ∃v.B
let exists = builtin! {
  // `exists l p` means `There exists an n in {1,...,l} such that p(n)`
  (Variable => Boolean) => Variable => Boolean
  context |input, pred| => Term::val(
    (1..=input) // Search range {1,...,n}
      .any(|n|  // Evaluate predicate at each n
        bln(&context.evaluate(&term!([program] [:n])))
      )
  )
};

// C -> P
let conjunction = builtin! {
  Predicate => Conjunction
  |p| => p.clone()
}

// C -> P && C
```

[9]Here, our primitive must capture the `Context` in order to recursively evaluate its argument.

```
let and = builtin! {
  Predicate => Conjunction => Conjunction
  |p, b| => Term::val(
    bln(&p) && bln(&p)
  )
};
```

Notice that there is a direct correspondance between the grammar and the terms of our language: every type corresponds to a non-terminal, and every function corresponds to a production rule, with parameters corresponding to the non-terminals in the production rule, and the return type corresponding to the produced non-terminal.[10]

5.3 Semantic Pruning

Semantics can be useful for narrowing our search space beyond just detecting duplicate terms. For example, say we wanted to modify our `Polynomials` language to include a construct to define variables, similar to Haskell's `let` expressions, so that we could, for example, write something akin to `let y = x + 1 in y*y*y` instead of having to write `x*x*x+(1+1+1)*x*x+(1+1+1)*x+1`. This could be very useful to our synthesizer, since it allows complex (but interesting) programs to have smaller representations. Unfortunately, our simple functional languages based on the λ-calculus cannot support this construct. We might try to get around this restriction by converting let constructs to β-redexes:

$$\texttt{(let v = d in e)} \longmapsto \texttt{(λv.e)d}$$

Since our enumerator only generates terms in β-normal form, programs of this form will never be generated. We can get around this by intruducing a new primitive:

```
let abstract = builtin! {
  (N => N) => N => N
  |e, d| => term!([e] [d])
};
```

This is because while `(λv.e)d` is not in β-normal form, `(abstract e d)` is (as long as `e` and `d` are β-normal). However, if we do this, we will generate well-typed terms such as `abstract (plus one) zero` instead of `plus one zero`, which many be undesirable. If we have semantic analysis, the enumerator will detect these duplicates once they are generated, but in more complex situations, we could avoid this by marking any

[10]Note that the only non-obvious correspondance is between b and `Variable => Boolean`. this type is chosen because we want b to be parametrized by a new variable. this breaks the correspondance because the grammar does not capture this behavior. instead, it is usually implied in our variable terminals. Q does not have a corresponding function because λ-abstraction is a basic feature of functional languages.

term of the form `abstract e` as `Malformed` if `e` is not a λ-abstraction which uses its argument.

6 Stochastic Search

While enumeration can be used to search for small programs, it relies on completely exhausting the search space, which is usually infeasible. To attempt to mitigate this, we show how to use a stochastic method to generate larger terms.

6.1 Metropolis-Hastings

The Metropolis-Hastings algorithm is an algorithm to sample from a complex probability distribution. In our case, we will define a distribution that assigns the highest likelihoods to programs that perform best, and then sample from it in the hopes that, since correct are the programs with the highest probability, we will pick a program which performs well.

This is a Markov Chain Monte Carlo method, meaning it works by beginning with some initial candidate and repeatedly transitioning to new ones. In order to implement the Metropolis-Hastings algorithm, we will need to define:

1. A space X over which our distribution acts.
2. A distribution $\mathbb{P} : X \Rightarrow \mathbb{R}$.
3. A proposal distribution $g(x|y)$, which we can efficiently sample from. This gives the probability of proposing x given that the previous proposal was y. It should be possible to compute $\frac{g(x|y)}{g(y|x)}$ efficiently, and we must ensure that $g(x|y)$ is nonzero iff $g(y|x)$ is.

Once we have these, the algorithm consists of:

1. Choosing an initial candidate x_1.
2. Selecting a proposal p_n according to g.
3. Set $x_{n+1} = p_n$ with probability $\min\left(1, \frac{\mathbb{P}(x_n)g(p_n|x_n)}{\mathbb{P}(p_n)g(x_n|p_n)}\right)$, and $x_{n+1} = x_n$ otherwise.
4. Repeating the previous 2 steps many times, then selecting the candidate with the highest score.

6.2 Application to a Functional Setting

The method described here was inspired by [8], in which the Metropolis-Hastings algorithm is used to generate "Superoptimized" assembly code (meaning, code which is not just efficient, but the most efficient version way of accomplishing its purpose). Converting to a functional setting brings a few challenges. For example, it's unclear:

1. How to evaluate the correctness of a program. In the assembly setting, programs were evaluated based on the Hamming distance between CPU registers after

executing the original and synthesized programs. However, this is probably not effective in a functional setting.
2. How to mutate programs. In the assembly setting, there were several different kinds of modifications that were possible which do not translate to a functional setting (such as swapping the order of two instructions or replacing one with a NOOP).

For the first issue, make a simple choice: The correctness of a program is just how many examples it evaluates correctly. Then, we can define $\mathbb{P}(x) = e^{C*N(x)}$, where C is a parameter which we can tune, and $N(x)$ is the number of correct answers.

For the second issue, we introduce two new properties to `Language`s:
1. `SMALL_SIZE: usize`: The largest size we can efficiently enumerate entirely.
2. `LARGE_SIZE: usize`: A larger size, which will be more (but not prohibitively) expensive to compute.

Now, we can define three types of mutations:
1. Variable Swaps, in which we replace one variable in a term with another of the same type.
2. Small Subterm Swaps, in which we replace a small subterm with another of the same size (and type).
3. Large Subterm Swaps, in which we replace a large subterm with another of the same type, (though possibly not of the same size). This is important because it allows our candidate program to change size.

We must be careful to ensure that the terms we generate are always in β-normal form. If we do not, then we violate the requirement that $g(x|y)$ is nonnegative iff $g(y|x)$ is, since our enumerator never produces terms which are not in β-normal form.

We can select replacement terms uniformly at random by enumerating terms (using reservoir sampling to avoid having to store the entire program space in memory). Whenever we make a large swap, in order to compute $\frac{g(x|y)}{g(y|x)}$, we may have to enumerate not only the possible replacements, but also the terms with the size and type of the replaced node, even though we don't use any of them. Again, we can use caching to mitigate this issue.[11]

Another issue is that the candidate's size tends to grow without bound. This is because, when we perform a large swap, we are usually replacing a small subterm with a large one (because there are many more small subterms than large ones).

[11] We could also try to use a more clever counting algorithm which does not generate terms as it counts them, but this would not work for languages with nontrivial semantics.

We can compensate by biasing our sampling algorithm to try to replace terms with others of roughly the same size, but this is difficult since it may not always be possible to construct terms of the right type of a specific size. To mitigate this issue, we can include a correction term in $\mathbb{P}(x)$ which punishes terms for being too large. Again, for brevity, we omit the implementation.

7 Results

The two languages we consider in this section are `Polynomials`, and `NumLogic`, a slight modification/extension of the conjunctive query language shown in Section 5.2, with the primitives below (along with a brief description):

```
-- Basic primitives:
mul :: Atom => Atom => Atom
mul a b = a * b

pow :: Atom => Atom => Num
pow b k = b^k

prime :: Var => Pred
prime = {true if n is prime}

and :: Pred => Conj => Conj
and p c = p && c

eq :: Atom => Atom => Pred
eq a b = a == b

less :: Atom => Atom => Pred
less a b = a < b

divisor :: Atom => Atom => Pred
divisor p q = p > 1 && q % p == 0

-- Reductions over ranges:
-- Check if a predicate is satisfied
exist :: Var => (Var => Bool) => Bool
exist v p = ∃n∈[1,..,v] (p n)

-- Count values with a property
count :: Var => (Var => Bool) => Num
count v p = #n∈[1,..,v] (p n)

-- Sum up values
sigma :: Var => (Var => Num ) => Num
sigma v f = Σn∈[1,..,v] (f n)

-- Type casts:
atom :: Var  => Atom
num  :: Atom => Num
conj :: Pred => Conj
bln  :: Conj => Bool
```

For `NumLogic`, we also include some pretty-printing functionality, which we will use instead of displaying the full programs. For example, we write `∃n<=k [Prime(n)]` instead of `exist k (n -> bln (conj (prime(n))))`.

7.1 Enumerative Search

The enumeration is significantly more powerful than might be expected, especially using caching and semantic analysis. For example, see the times needed to enumerate `NumLogic` programs of type `Var => Bool` of different sizes in Section A.2. This technique was able to discover formulas for many entries in the OEIS, sometimes in interesting ways (see A008585, for example). All of the sequences below can be generated in less than 20 seconds (They have size < 40):

1. A000961: Prime powers (`f -> ∃k<=f [∃m<=f [Prime(m) && (f)=(m^k)]]`)
2. A002808: Composite numbers (`f -> ∃k<=f [(k)|(f) && (k)<(f)]`)
3. A000430: Primes and squares of primes (`f -> ∃k<=f [Prime(k) && (f)|(k*k)]`)

If we instead synthesize terms of the type `Var => Num`:

1. A230980: The number of primes below n (`f -> #k<=f [Prime(k)]`)
2. A168014: The sum of i up to n of the number of divisors of i (`f -> Σk<=f [#m<=f [(m)|(f)]]`)
3. A000290: The squares (`f -> Σk<=f [(f)]`)
4. A000590: Sums of 5th powers (`f -> Σk<=f [Σm<=k [(k*k*k*k)]]`)
5. A325459: Sums of nontrivial divisors (`f -> Σk<=f [#m<=k [(m)|(k) && (m)<(k)]]`)
6. A010051: The characteristic function of the primes (`f -> #k<=f [Prime(f^k)]`)
7. A128913: $n\pi(n)$ (`f -> Σk<=f [#m<=f [Prime(m)]]`)
8. A008585: Multiples of 3 (`f -> Σk<=f [#m<=k [(f^m)|(f*f*f))]`)

7.2 Metropolis-Hastings Search

In cases where the solution program is small, enumeration is always faster than stochastic search. For simple languages, the Metropolis-Hastings synthesizer allows us to search for larger programs than would be possible with enumeration alone, though quite unreliably. For example, in `Polynomials`, consider the sequence $a_n = 6n^4 + 6n^2$. The shortest program generating this has size 30, and (on my machine) takes 19 seconds to generate through enumerative synthesis. A stochastic search (starting with the smallest program of type `N => N`, so as to avoid a biased starting point) on the other hand, is usually able to find it very quickly (usually around 1-2s, but sometimes as fast as 0.3s). However, when it does not find a solution quickly, it often does not find one at all.

In more complex settings, such as in the `NumLogic` language, I was not able to make stochastic search effective. In both cases, the ineffectiveness is likely due to the synthesizer's tendency to search for increasingly large terms, but also is probably because of poorly tuned parameters and a scoring function which is not particularly useful when the proposal drifts too far from the solution. There are a lot of potential improvements to be made, and this would be interesting to explore further.

7.3 Reflections

This project was a fascinating learning opportunity. It was especially interesting to be able to apply several concepts from my recent modules in unexpected ways, most notably Lambda Calculus & Types and Principles of Programming Languages. While the final codebase sits at only around 5,000 lines, large portions had to be repeatedly rewritten from scratch to acheive the final result. This was the most difficult project I've undertaken, and even steps that seemed simple at the start (such as implementing the enumerator efficiently) ended up being months of careful effort. There are still many directions this work could continue in, for example, using information about the provided input-output examples in the enumeration process, or experimenting with different approaches to the stochastic search.

A Appendix

A.1 Lambda Calculus Basics

1. a **λ-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. Terms are **α-equivalent** if they are equal up to the renaming of bound variables.
5. **β-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 **β-reduces** to another, this may require more than several reduction steps.
6. Two λ-terms are **β-equivalent** if they β-reduce to a common term (up to α-equivalence).
7. **η-reduction** is the operation mapping a term $(\lambda x.Mx)$ to M. η-equivalent terms are not necessarily β-equivalent.
8. A λ-term is in **β-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.
9. 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.
10. When the head of a λ-term is a variable, it is in **head normal form**.
11. The **order** of a λ-term is the order of its type, defined as:

$$\text{order}(T) \coloneqq \begin{cases} 0 & \text{if } T \text{ is a variable} \\ \max(1 + \text{order}(A), \text{order}(B)) & \text{if } T \equiv A \Rightarrow B \end{cases}$$

A.2 Enumeration Time Table

Size	# of terms	Enumeration time (s)
8	1	0.00020
20	8	0.00295
25	52	0.01268
26	291	0.03499
27	220	0.03098
28	454	0.07579
29	344	0.08978
30	373	0.18885
31	390	0.22263
32	2231	0.51554
33	1080	0.57150
34	5138	1.3557
35	2558	1.4925
36	5929	3.1751
37	3788	4.2012
38	15703	8.3403
39	6516	15.090
40	56226	24.417
41	15572	40.995
42	106827	77.327
43	29220	102.68
44	171195	183.61
45	45822	256.10
46	496258	483.86
47	89844	746.35
48	1310846	1282.98
49	184442	2363.75
50	2313868	3836.52

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.

[6] G. G. Hillebrand, P. C. Kanellakis, and H. G. Mairson, "Database Query Languages Embedded in the Typed Lambda Calculus," *Information and Computation*, vol. 127, no. 2, pp. 117–144, 1996, doi: https://doi.org/10.1006/inco.1996.0055.

[7] G. Hillebrand and P. Kanellakis, "On the expressive power of simply typed and let-polymorphic lambda calculi," in *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, 1996, pp. 253–263. doi: 10.1109/LICS.1996.561337.

[8] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, in ASPLOS '13. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 305–316. doi: 10.1145/2451116.2451150.