Testing typed functional programs and re-synthesizing them

Katherine Ye, advised by Prof. David Walker[1]

[1] *Princeton University*

Building on Godefroid (2005)'s work on symbolic execution for testing of imperative programs, we develop a novel algorithm to test typed functional programs. This algorithm generates inputs that traverse every branch of a program, solving equations along the way. These inputs can serve as input examples for an example-driven synthesis system, which will then attempt to re-synthesize the program. This process creates a connection between functional testing and program synthesis.

We implement these algorithms in the context of Myth, a type- and example-directed synthesis system (Osera and Zdancewic (2015)), and we name our implementation Pyth. Examining the differences between these input-output examples and human-written examples, as well as their respective synthesized and re-synthesized programs, sheds light on the additional information needed for Myth to synthesize a program.

I. INTRODUCTION

The two problems of verifying programs and synthesizing them can be viewed as sides of the same coin. Thirty years ago, Manna and Waldinger (1980) presented an approach that regarded program synthesis as a theorem-proving problem. Recently, Srivastava and Gulwani (2010) interpreted program synthesis as generalized program verification. They encoded a synthesis problem as a verification problem such that their verification tool would infer both program invariants and the program itself.

Here, we present a connection between *testing* programs and synthesizing them.

A. Symbolic execution and testing programs

When it comes to automatic program testing, there are two main philosophies: *blackbox* fuzzing and *whitebox* testing. A test here is some input to a program, paired with an expected output. Blackbox fuzzing treats the program as a black box. It tests that a program behaves the way it should, without inspecting the program's code or structure. It does this by providing random or malignant inputs, or systematically mutating well-formed inputs, to try and cause the program to crash.

On the other hand, whitebox testing treats the program as transparent. It tries to ensure that every path through the program is well-formed and doesn't result in crashes. One way of reaching what testers call "100% code coverage" is to systematically pick each path through a program and determine which inputs traverse that path. This can be accomplished by symbolic execution.

Note that both testing methods generate tests for the program. Blackbox fuzzers are easier to write, but whitebox tests tend to give more code coverage and account for more corner cases.

B. Program synthesis and Myth

Program synthesis aims to turn human insight into computer code. There are three main axes of synthesis:

1. Does it take existing code and fill holes in it, or does it take a skeleton of examples and synthesize functions?
2. Does it synthesize imperative code or functional code?
3. To synthesize code, does it use SAT solvers or does it use proof theory?

The Sketch system, by Solar-Lezama (2008), is a well-known system that gives the former answer to these three question. This paper considers the system Myth, by Osera and Zdancewic (2015), which gives the latter answer to these three questions. Myth is a program synthesis system that synthesizes recursive functions that process algebraic datatypes, and it operates on a typed functional ML-like language that is a subset of OCaml. It takes as input a list of concrete examples of inputs and output that a function would produce. This is a way of partially specifying a function, and given these, Myth attempts to produce a full specification of a function—that is, the function itself. Osera and Zdancewic (2015) introduce Myth with the following example:

```
(* Type signature for natural numbers and lists *)
type nat =
| O
| S of nat

 type list =
 | Nil
 | 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 =
match l1 with
| Nil -> l1
| Cons(n1, l2) -> Cons(n1, Cons(n1, f1 l2))
```

The user gives Myth an environment of types, as well their understanding of the *stutter* function in the form of small examples that duplicate each element of the input list, and Myth synthesizes the full function.

This paper will use Myth as an environment in which to perform program de-synthesis and re-synthesis. That is, our benchmarks are the most complex of their benchmarks, our language is a subset of theirs, and our implementation is embedded as additional modes in the Myth implementation.

C. Contributions

This paper makes the following contributions to the study of typed functional programs.

First, we make a connection between testing such programs and synthesizing them.

We summarize Godefroid's work in Section II, then we extend Godefroid (2005)'s work on whitebox-fuzzing programs in C-like languages to whitebox-fuzzing programs in ML-like languages, for which there is little prior work. In Sections III and IV, we develop a novel algorithm to solve equations involving algebraic datatypes, as well as a novel algorithm for backwards symbolic execution. These algorithms together can generate thorough test case inputs that traverse every execution path of a function. At the same time, they can detect dead or unreachable code.

To extend testing to synthesis, in Section IV, we develop a novel algorithm to augment the test case inputs into input-output examples for Myth. We then give detailed analyses of several end-to-end de-syntheses and re-syntheses of complex Myth benchmark functions in Sections IV and V. To our knowledge, this is the first attempt to systematically de-synthesize functions into input-output examples (which are a common type of input for program synthesis systems). This raises the interesting question of using these examples to characterize functions in an information-theoretic manner. We discuss our conclusions and future work in Section VI, followed by appendices on the algorithm implementations in OCaml.

We give the name Pyth to the end-to-end system consisting of the equation-solving algorithm, the backwards symbolic execution algorithm, and the input-output example augmentation algorithm.

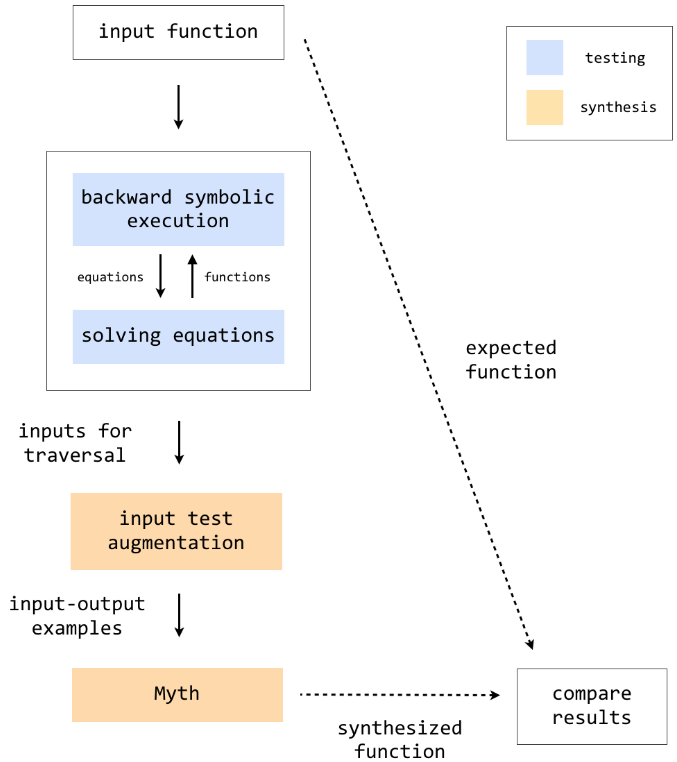

Figure 1. The Pyth system.

II. TESTING IMPERATIVE PROGRAMS

A. Motivation and DART

First, we discuss prior work in the area of whitebox testing. DART is a tool for automatically testing software by combining blackbox fuzzing with whitebox testing, yielding a technique called "whitebox fuzzing" (Godefroid (2005)). It turns out that in practice, this technique is very effective at finding bugs. A similar tool called SAGE, by the same authors, is currently used to test large codebases at Microsoft. SAGE was able to find a bug that caused a critical security vulnerability without any knowledge of the file formats involved. Moreover, according to Godefroid (2012), "extensive blackbox fuzzing of this code failed to uncover the bug and that existing static-analysis tools were not capable of finding the bug without excessive false positives," showcasing the effectiveness of whitebox fuzzing compared to current testing techniques.

Blackbox fuzzing often fails precisely because it treats the program as a black box, so it fails to find corner cases in code such as the following:

```
if (x == 1)
  thisLeadsToFailure();
else
  (...)
```

(Let x be an integer.) Code like this is quite common.

To test it, blackbox fuzzing will guess random values for x, but since there are 2^{32} possibilities (the size of an integer), the probability of it triggering the error is infinitesimally small. Whitebox testing, however, would easily be able to generate $x = 1$ to trigger the failure. DART and SAGE build on this insight.

B. How DART and SAGE ensure code coverage

Here is the core of DART and SAGE's symbolic execution algorithm. Given this piece of code, reproduced from Godefroid (2008):

```
void top(char input[4]) {
    int cnt=0;
    if (input[0] == b) cnt++;
    if (input[1] == a) cnt++;
    if (input[2] == d) cnt++;
    if (input[3] == !) cnt++;
    if (cnt >= 3) abort(); // error
}
```

Like in blackbox fuzzing, this algorithm takes an initial well-formed output. We pick the array of letters "*good*". Like in whitebox testing, it inspects the path this input takes through the function, and collects constraints along the way. These constraints arise whenever the program branches on a conditional. For the "*good*" path, the path constraints are the conjunction of $i[0] \neq b, i[1] \neq a, i[2] \neq d, i[3] \neq !$. Any inputs that satisfy these constraints lies in the same equivalence class of inputs as "*good*" does; for example, "*abcd*" is in the same equivalence class.

To create an input in a different equivalence class—that is, an input that traverses a different path than the "*good*" path—the algorithm systematically negates the collected constraints in some way, and arrives at an input by passing this new set of constraints to a SAT solver. Here, we choose to negate the last constraint, which yields the set $i[0] \neq b, i[1] \neq a, i[2] \neq d, i[3] = !$, which yields the input "*goo*!". (Conceivably, here, a SAT solver could have returned "*abc*!"; but for clarity we use previous inputs.) This "*goo*!" traverses a new path of the program—for "*good*", *cnt* was 0, but now it is 1.

Note how here we are mutating well-formed inputs in a manner reminiscent of blackbox fuzzing, but traversing the function in a whitebox manner.

There are $2^4 = 16$ possible program paths. If the negations are performed in depth-first order, then the search space is explored from left to right as shown in the tree in Figure 2. Each fork partitions the inputs into two equivalence classes. The first fork, for example, partitions the inputs into ones that start with b and ones that don't.

The error is finally reached with the input *badd*, which causes *cnt* to be 3 and the program to abort. The depth-first search concludes with the last input *bad*!, which also causes the program to abort.

There are two problems with this traversal algorithm. In practice, codebases are large and interlinked, so the number of paths explodes exponentially. Additionally,

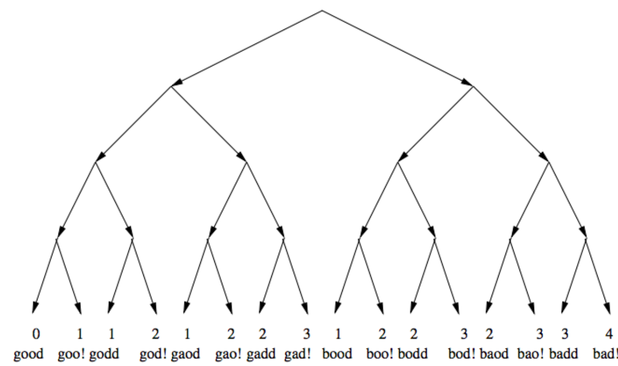

Figure 2. Search space for the code above, with the value of the variable *cnt* at the end of each run and the corresponding input string. *Diagram and caption reproduced from Godefroid (2008).*

symbolic execution may be imprecise or impossible due to factors such as floating-point arithmetic and calls to opaque functions. Godefroid (2008) improves on this traversal algorithm to present an algorithm called *generational search*. Generational search is designed to maximize code coverage while efficiently generating new tests, and it is resilient to path divergence. We do not summarize it here because we study the behavior of Pyth algorithms on single short functions; see Godefroid (2008) for the details.

III. TESTING FUNCTIONAL PROGRAMS: PROBLEM STATEMENT

A. Motivation

There has been plenty of work done on code path traversal, symbolic execution, fuzzing, and whitebox testing of imperative programs. See Godefroid (2005), whose DART system tests C programs as summarized in Section II. However, there has been little work done on testing of typed functional programs in ML-like languages such as OCaml. Therefore, one main appeal of this work is its novelty.

This work also possesses utility. Functional programs are much less likely to crash for C-like reasons such as dereferencing null pointers. However, they may still throw exceptions instead of safely capturing possible failure in a type signature. It is easy to unwittingly perform an unsafe operation such as taking the head of an empty list. (Indeed, the *head* function in the Haskell standard library throws an exception if the input list is empty, instead of returning a value of type *listoption*, which would capture errors at compile-time).

We name our system Pyth.

B. Novelty

OCaml possesses many language features that C does not. (From now on, we will use "OCaml" as a synonym for "typed functional ML-like language" and "C" as a synonym for "imperative C-like language.") The most notable of these features are algebraic data types, pattern-matching on values, and inductively defined types such as natural numbers.

One central contribution of this paper is the development of a novel algorithm to handle these language constructs in symbolic execution. Solving this problem has revealed interesting correspondences between Pyth's algorithm and DART's algorithm. A typical function that DART considers, taken directly from Godefroid (2005), looks like this:

```
int f(int x) { return 2 * x; }
int h(int x, int y) {
  if (x != y)
    if (f(x) == x + 10)
      abort(); /* error */
  return 0;
}
```

A typical function that Pyth considers, taken directly from Osera and Zdancewic (2015)'s benchmarks for Myth, looks like this:

```
let rec list_compress (l1 : list) : list =
    match l1 with
      | Nil -> Nil
      | Cons (n1, l2) ->
      (match list_compress l2 with
        | Nil -> l1
        | Cons (n2, l3) ->
           (match compare n2 n1 with
            | LT -> Cons (n1, Cons (n2, l3))
            | EQ -> failwith "runtime error"
            | GT -> Cons (n1, Cons (n2, l3))))
```

Primitive types in the DART system correspond to algebraic data types in Pyth and Myth, and if statements (which test primitive types) correspond to match statements (which deconstruct algebraic data types vis pattern-matching). Both if-statements and match statements may both be nested arbitrarily deeply, contain arbitrarily many branches that partition the inputs, and contain arbitrary expressions such as function calls in the guard. We will call the "guard" of an if-statement its predicate, and the "guard" of a match statement the expression it matches on.

In both functions, it is a nontrivial task to figure out inputs that bypass a guard to land in a particular branch. To provoke failure, Godefroid (2005) must figure out x, y such that $x \neq y$ and $f(x) == x + 10$. Similarly, to provoke failure, Pyth must find an input $l1$ such that three guards, listed below, are bypassed. Godefroid (2008) solves the task of finding inputs by collecting constraints and SAT solvers to solve their conjunction, each negated selectively to choose which branch to land in. Analogously, to find inputs to bypass match guards, Pyth must solve functional equations such as the following.

```
find n2, n1 : compare n2 n1 = EQ /\
find l2 : list_compress l2 = Cons(n2, l3) /\
find l1 : l1 = Cons (n1, l2)
```

The last of these equations is trivial, but the first two are not. The first one involves a call to a recursive function with multiple holes to be filled (the parameters $n1$ and $n2$) and the second one involves a recursive call to $list\_compress$ itself. In addition, note that the second equation involves a variable $l3$ whose value doesn't matter.

The rest of this section is devoted to clearly defining the problems Pyth aims to solve, and the next section will describe the solutions.

C. Language definition

The language of Myth is a typed, purely functional subset of OCaml. It includes user-defined algebraic data types (and no primitive data types, such as integers) and recursive functions. The language of Pyth is a minimal subset of Myth's.

Pyth's language includes only three types. First, it includes booleans, as a non-recursive type with nullary constructors, which are constructors that take no arguments. It also includes inductively defined natural numbers, as a recursive type with unary constructors. Next, it includes functions, but not first-class functions, since we do not wish to have to synthesize functions ourselves. The difficulty of including lists will be illustrated in Section V, where we give a detailed trace involving $list\_compress$. (Lists are not formally included in this spec because they are not included in the algorithm implementation.) Arbitrary algebraic datatypes are even more difficult.

Here is the formal definition of Pyth's language:

```
types ::= bool | nat | t1 -> t2

bool ::= True | False
nat ::= O | S of nat

e ::=
  | x
  | f e
  | (match e with | True -> e1 | False -> e2)
  | (match e with | Z -> e1 | S n -> e2) (modulo
     naming of n)

f ::=
  | fix (x : t1) : t2 = e
  | fix (x : t1) : t2 = em
```

Note that functions with multiple arguments can be represented, as usual, by currying the function, or translating it into a sequence of functions that each takes one argument. Note that match statements can indeed match on other match expressions, though that is not common practice. Lastly, note that we include recursive functions.

In addition to arbitrary data types, Pyth's language has the following limitations:

- It excludes some language features of Myth, such as tuples.
- It excludes higher-order functions, because that may give rise to the problem of solving a constraint for a function, which an entire paper in itself.
- It excludes if-statements, because the problem of reaching each branch in an if-statement like this one:

```
if (x > 0 || y + z < 5 + x) {
  branch 1
} else if (x - y <= -1) {
  branch 2
} else {
  branch 3
}
```

  has already been solved by DART on imperative programs using SAT solvers, as covered in Section II.
- It excludes more general kinds of match statements such as ones that match on more than one constructor, or wildcards that are not inside a constructor:

```
match expr with
  | O -> OK
  | S m -> OK
  | S (S O) -> not allowed
  | S (S m) -> not allowed
  | _ -> not allowed
```

D. Problem statement: input-output example generation

Throughout the rest of this section we will take *list_compress* as a goal function to desynthesize, which will illustrate the problems that arise in generating input-output examples, as well as many subproblems.

```
let rec list_compress (l1 : list) : list =
    match l1 with
      | Nil -> Nil
      | Cons (n1, l2) ->
      (match f1 l2 with
        | Nil -> l1
        | Cons (n2, l3) ->
           (match compare n2 n1 with
            | LT -> Cons (n1, Cons (n2, l3))
            | EQ -> Cons (n1, l3) (*)
            | GT -> Cons (n1, Cons (n2, l3))))
```

Given any valid function f synthesized by Myth that contains only constructs in the Pyth language described above, try to "desynthesize" the function. That is, try to generate a set of input-output examples such that Myth can resynthesize the "same" function f. This is the top-level goal of the paper, and it comes with a built-in way to measure success; namely, we strive to reproduce the human-written input-output examples included in the Myth benchmarks for each function. *list_compress* was synthesized with 15 examples:

```
let list_compress : list -> list |>
{ [] => []
| [0] => [0]
| [3] => [3]
| [2] => [2]
| [1] => [1]
| [1;0] => [1;0]
| [3;2] => [3;2]
| [3;3;2] => [3;2]
| [0;0] => [0]
| [3;3] => [3]
| [3;3;3] => [3]
| [0;1] => [0;1]
| [0;0;1] => [0;1]
| [2;3] => [2;3]
| [2;2;3] => [2;3]
} = ?
```

Let's informally examine how these examples relate to the function and to each other. First, it seems that each "branch" of *list_compress* is populated by at least one example. The branch marked by $(*)$ is populated by $[3;3;3] => [3]$, among others, and the reader may verify that this is true for the rest. This observation is borne out by Myth's algorithm, which explicitly generates a match on an expression only if each of the branches would be populated by an example. This is a hint that symbolic execution is necessary, if possibly not sufficient, for de-synthesizing Myth functions—it is essentially running part of the Myth algorithm in reverse.

Another observation is that every non-trivial example in the set possesses a corresponding sub-example, such as the following subset:

```
[2;2;3] => [2;3]
[2;3] => [2;3]
[3] => [3]
[] => []
```

This is because *list_compress* is recursive, and requires recursive sub-examples of each example that provokes a recursive call. Again, a close look at the Myth algorithm will verify that this "recursive backpatching" is happening. The reader may verify the existence of recursive sub-examples for the rest of the set.

Lastly, there seem to exist structurally similar examples that traverse the same branches of the function, but contain different constants. $[2;2;3]$ (and its recursive subexamples) and $[0;0;1]$ (and its subexamples) may form such a pair. We hypothesize that the constants must be varied to avoid Myth over-fitting to particular constants such as $[2;2;3]$. In fact, if $[0;0;1]$ and its subexamples are removed, Myth overfits on the *compare* outputs:

```
match compare n2 n1 with
  | LT -> l1
  | EQ -> Cons (n1, l3)
```

```
| GT -> [2; 3]
```

Combining these three insights, we choose to accomplish the goal of desynthesis by performing symbolic execution on f to find inputs (by the first observation), then augmenting the resulting inputs to include recursive subexamples and structurally similar examples.

An interesting question is whether these human-written examples constitute some sort of minimal set needed to synthesize a function. It may be minimal in size, quality, or both. However, this is out of the scope of the project, and we do not guarantee that the input-output examples generated constitute the minimal set of examples needed to synthesize a function. We also do not guarantee that Myth necessarily needs all the examples generated by Pyth; some may be superfluous.

We also ignore the problem of checking that the resynthesized f is equivalent to the original f. This is a non-trivial problem, but we take the approach of "whatever you get, you get." We also do not iteratively search for examples based on how "close" the resynthesized f is to the original f, though this is one possible optimization.

E. Problem statement: symbolic execution

Recall that Myth will only synthesize a match statement if each of its "branches" is populated by at least one example. Here we state precisely how we model the structure of these statements, then define the symbolic execution problem.

Pyth's language includes match statements S of the following form:

```
match expr with
| Constructor1 args1 -> leafExpr1
| ...
| ConstructorN argsN -> leafExprN
```

where any leaf expression may also be a match statement, or some other expression such as a function call. Also, as mentioned earlier, Pyth's language specification excludes more general kinds of match statements such as ones that match on more than one constructor, or wildcards that are not inside a constructor:

```
match expr with
  | O -> OK
  | S m -> OK
  | S (S O) -> not allowed
  | S (S m) -> not allowed
  | _ -> not allowed
```

We model such statements S using an arboreal metaphor. The statement itself is a tree graph, since it contains no cycles. Within S we call $expr$ a guard and | $Constructor\_i\ args\_i \rightarrow leafExpr\_i$ a branch. Within that branch, we call $Constructor\_i\ args\_i$ a pattern and $leafExpr\_i$ a leaf.

Here is a picture of $list\_compress$ as a tree.

Note the nested match statements. (The body of the $compare$ function is omitted for brevity.) In this case, we call the leaves with no children the *terminal* leaves. The terminal leaves are a subset of *general* leaves, which can be match statements. Here, the terminal leaves would be Nil, $l1$, $Cons(n1, Cons(n2, l3))$, $Cons(n1, l3)$, and $Cons(n1, Cons(n2, l3))$.

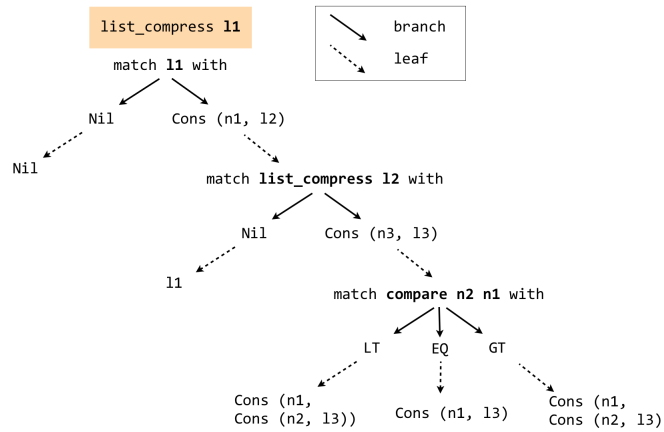

Figure 3. list compress's tree graph.

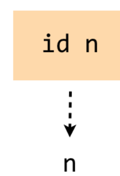

Figure 4. id's tree graph.

For comparison, here is a picture of id as a tree. Note that it contains no match statement, therefore no branches. It is only a leaf.

Now that we have defined terminology, we state the problem: **Given any valid function that contains only the constructs in the Pyth language, as well as an environment of types and functions, generate a set of inputs that traverse the function to reach every possible terminal leaf of the function.** (That is, given a particular terminal leaf, find some inputs that reach it.) If any terminal leaf cannot be reached, the algorithm should fail, either by returning failure (which is possible in many cases) or by looping forever.

In this statement, we allow advanced features such as functions with multiple arguments, functions calling other functions in the environment (which will then be traversed themselves), and recursive functions. However, we do not guarantee being able to find *all* inputs that reach a particular terminal leaf, just at least one solution, or some type of failure if no solution.

This symbolic execution is useful for program synthesis because the inputs must also traverse every branch of a program on their way to the terminal leaves, and as we figured out, Myth requires that every branch must be populated with at least one input. This is also useful for testing functional programs thoroughly for three reasons: to find inputs that cause the program to crash, to generate thorough test coverage, and to point out areas of dead or unreachable code.

F. Problem statement: solving equations

Let's try symbolic execution on $\mathit{list\_compress}$. As before, when we try to reach the terminal leaf marked $(*)$, we must bypass three guards in match statements along the way.

```
let rec list_compress (l1 : list) : list =
    match l1 with
      | Nil -> Nil
      | Cons (n1, l2) ->
      (match f1 l2 with
        | Nil -> l1
        | Cons (n2, l3) ->
           (match compare n2 n1 with
            | LT -> Cons (n1, Cons (n2, l3))
            | EQ -> Cons (n1, l3) (*)
            | GT -> Cons (n1, Cons (n2, l3))))
```

The guards require us to solve three equations before we may pass.

```
find n2, n1 : compare n2 n1 = EQ /\
find l2 : list_compress l2 = Cons(n2, l3) /\
find l1 : l1 = Cons (n1, l2)
```

In general, we are given a match statement S of this form,

```
match expr with
| Constructor1 args1 -> leafExpr1 (* branch 1 *)
| ...
| ConstructorN argsN -> leafExprN (* branch N *)
```

where we want to reach branch i, and expr has free variables v_i. Define "concrete value" to mean "value constructed via constructors; that is, something that is fully evaluated, not a function, and contains no free variables."

We want to find concrete values for the free variables such that when we substitute them in and evaluate the expression, the resulting expression matches the pattern corresponding to the branch we want. That is, we want to find $v_i = c_i$ such that $\mathit{expr}[v_i/c_i]$ matches the pattern $\mathit{Constructor\_i}\ \mathit{args}_i$. Note that args_i are not really part of the problem; they are wildcard arguments, and what we really care about is the head constructor being present.

We change the problem slightly such that the right side must be any concrete value, instead of just a head constructor. (As we will see later, the algorithm for the concrete value problem will work for the head constructor problem.) We give free variables the nickname of *holes*. Now, the problem looks like this: find concrete values c_i such that $e[v_i/c_i] = g$, where g is a concrete value that is the goal value. Also, we require that all functions provided must terminate on all inputs.

The problem includes the following features:

- Multiple holes on the left side of the equation ($\mathit{find}\ (n1\ n2 : \mathit{nat}) : \mathit{plus}\ n1\ n2 = 3$)
- Functions calling other functions
- Recursive functions
- Finding *all* values (or just more values) such that the equation is satisfied, which may be needed for deeper symbolic execution

The algorithm can usually handle the following, but a detailed study of this problem is omitted for brevity.

- Arbitrary expressions on the left hand side of the equation, so functions may be applied to input

  $\mathit{find}\ (n1\ n2 : \mathit{nat}) : \mathit{plus}\ (S\ n)\ (S\ n) = 3$

- Expressions on the left hand side of the equation where some arguments to functions have been supplied

  $\mathit{find}\ (n1\ n2 : \mathit{nat}) : \mathit{plus}\ (S\ n)\ (3) = 3$

We do not handle the following extensions:

- Guaranteed termination on equations with solutions involving only terminating functions. That is, the algorithm may fail nicely when there are no solutions, or it may diverge (infinite loop).
- Holes in the conclusion, such as here:

  $\mathit{find}\ (n : \mathit{nat}) : \mathit{plus}\ n\ 3 = \mathit{mult}\ n\ 3$

- Systems of constraints requiring SAT/SMT solver use such as the ones DART handles, such as

  $x > 5 \wedge y + z \leq 3 \vee z < 5$.

IV. TESTING FUNCTIONAL PROGRAMS: ALGORITHMS

A. Motivating example

Here is a motivating example for the capabilities of the Pyth implementation, as well as some of the questions it raises.

Given the following problem, we want to generate input-output examples for the max function, which finds the larger of two natural numbers. Given the input-output examples, we will feed them back into Myth and see what it synthesizes.

```
type nat =
  | O
  | S of nat

type cmp =
  | LT
  | EQ
  | GT
```

```
let rec compare (n1:nat) (n2:nat) : cmp =
  match n1 with
  | O ->
    (match n2 with
    | O -> EQ
    | S m -> LT)
  | S m1 ->
    (match n2 with
    | O -> GT
    | S m2 -> (compare m1 m2))

let max (n1 : nat) (n2 : nat) : nat =
  match compare n1 n2 with
  | LT -> n2
  | EQ -> n1                    (* arbitrary *)
  | GT -> n1

let toTraverse : nat -> nat -> nat =
  max ;;
```

Note some features that make this problem difficult:

1. Multiple types (*cmp*, *nat*), one of which is recursive (*nat*)
2. Recursive functions
3. Calls to another function
4. Matching on a call to another function
5. Two functions that take multiple arguments
6. Nested matches in one of the functions
7. Multiple solutions to inner contraint problems

Here is a high-level trace for how Pyth generates input-output examples:

1. Symbolic execution

   To reach the leaf of branch 1 (*LT*), it must solve the following constraint:

   ```
   find n1, n2 : compare n1 n2 = LT
   ```

   To reach the leaf of branch 2 (*EQ*), it must solve the following constraint:

   ```
   find n1, n2 : compare n1 n2 = EQ
   ```

   To reach the leaf of branch 2 (*GT*), it must solve the following constraint:

   ```
   find n1, n2 : compare n1 n2 = GT
   ```

2. Constraint solving

   For each branch, the algorithm inspects leaves to see if it returns the correct result. Then, it sees if can indeed reach that leaf, via a recursive call to itself. It returns the following results:

   ```
   branch LT: (n1 = O) => (n2 = S m)
   branch EQ: (n1 = O) => (n2 = O)
   branch GT: (n1 = S m1) => (n2 = O)
   ```

3. Postprocessing into concrete input-output examples

   The postprocessing algorithm is given a set of input rows as above. It finds unconstrained arguments (for example, if a function never matches on an arguments) and wildcards (such as *m* in *S m* above) and fills them in with arbitrary values. This process is described in Section IV.

   After this process, the final set of input rows is a concrete set of input arguments where no two rows are the same. See Section IV for the proof of this in the description of the symbolic execution algorithm. Then the postprocessing algorithm runs the function on each row of inputs to produce an output for that row.

   ```
   branch LT: (n1 = O) => (n2 = S O)
   branch EQ: (n1 = O) => (n2 = O)
   branch GT: (n1 = S O) => (n2 = O)

   branch LT: (n1 = O) => (n2 = S O) => S O
   branch EQ: (n1 = O) => (n2 = O) => O
   branch GT: (n1 = S O) => (n2 = O) => S O
   ```

   Finally, we have an input-output table, where each row contains *n* inputs and one output. Myth does not accept this table as an input-output format; it requires that all inputs be given in the order specified by the function's arguments, and that rows starting with the same inputs must be grouped together in a nested, partial function fashion. The postprocessing algorithm handles this and returns the following result.

   ```
   { O => { S O => S O,
            O   => O },
     S O => S O}
   ```

Pyth can then feed this collection of input-output examples straight into Myth. This is what Myth synthesizes:

```
let nat_max (n1 : nat) (n2 : nat) =
  match n1 with
  | O -> n2
  | S n3 -> 1
```

Note the two main differences between this and the *max* function we expect. Myth has over-fitted to the output "1" and returns that constant. In addition, it does not recurse.

Why doesn't it re-synthesize the *max* function we expect? To figure this out, let's examine the set of human-written input-output examples, which will shed light on the additional examples that Myth needs to fully re-synthesize a function.

Here is the corresponding Myth benchmark included in the PLDI benchmarks. Note the human-written input-output examples, and notice how they differ from the base input-output examples provided by symbolic exection and postprocessing. In particular, the base input-output examples are a strict subset of these. *nat_max* here is a *partial function* composed of input-output examples, and Myth will try to synthesize the full function.

```
type nat =
  | O
  | S of nat

type cmp =
  | LT
  | EQ
  | GT

let rec compare (n1:nat) (n2:nat) : cmp =
  match n1 with
  | O ->
    (match n2 with
     | O -> EQ
     | S m -> LT)
  | S m1 ->
    (match n2 with
     | O -> GT
     | S m2 -> (compare m1 m2))
;;

let nat_max : nat -> nat -> nat |>
{
  0 => ( 0 => 0
       | 1 => 1
       | 2 => 2 )
| 1 => ( 0 => 1
       | 1 => 1
       | 2 => 2 )
| 2 => ( 0 => 2
       | 1 => 2
       | 2 => 2 )
} = ?
```

Here is the function that Myth synthesizes:

```
let rec nat_max (n1 : nat) (n2 : nat) =
      match n1 with
        | O -> n2
        | S n3 -> (match n2 with
                     | O -> n1
                     | S n4 -> S (f1 n3 n4))
```

Interestingly, Myth synthesizes a more direct, though less understandable, function than the first human-written max function. It does this because the algorithm hesitates to grow the match *scrutinee* size (that is, the size of the match guard), opting to match on arguments instead of a function call. When the initial scrutinee size is increased in the Myth settings, Myth does indeed synthesize the expected *nat_max* function using *compare*.

For the rest of the section, we describe the three algorithms, which are the equation-solving algorithm, the backwards symbolic execution algorithm, and the input-output augmentation algorithm. The algorithm for solving equations is the trickiest of the three, and is the main technical contribution of the paper. Since the other two rely on it, we give the general algorithm first, run through some examples, and consider some generalizations.

B. Solving equations

As covered in the previous section, the equation-solving problem looks like this: given an expression e with free variables v_i, find concrete values c_i such that $e[v_i/c_i] = g$, where g is a concrete value that is the goal value. If no such c_i exist, fail or diverge. If multiple or infinite such $\{c_i\}$ solution sets exist, we guarantee finding at least one such $\{c_i\}$, which will often be the simplest one, but we do not guarantee finding all of them.

Solve the problem by induction on the structure of e. For convenience, here is the Pyth language definition.

```
types ::= bool | nat | t1 -> t2

bool ::= True | False
nat ::= O | S of nat

e ::=
  | x
  | f e
  | (match e with | True -> e1 | False -> e2)
  | (match e with | O -> e1 | S n -> e2) (modulo
      naming of n)

f ::=
  | fix (x : t1) : t2 = e
  | fix (x : t1) : t2 = em
```

Consider each case below, ranked by order of difficulty.

1. **e is a function: e = f.**

   This is not allowed. This is because we do not allow functions to have free variables like y in the following: $f\ x = y$. Thus, with no free variables (holes) to solve for, equations to solve will have one of two forms.

   They may be testing function equality ("find nothing such that $f = g$") which is not a common use case in symbolic execution. Or, they may require dealing with higher-order functions ("find x such that $f\ x = g$" (where $(e = f\ x)$ is a function), which is out of the scope of this project.

2. **e is a free variable: e = x.**

   Intuitively, this case is sort of an axiom, or base case. Free variables can always be assigned values, so it does not require any recursive solving of subproblems. The equation will always look like "find x such that $x = goal$." We consider two cases. Either the $goal$ contains free variables, or it does not.

If the goal doesn't contain free variables, it is composed of nulllary constructors, such as in "find x such that $x = \mathit{True}$." In this case, we simply return the equation $x = \mathit{True}$. This assigns a concrete value to x, which is what we want.

Or it can look like "find x such that $x = Sx0$" (or some other non-nullary constructor applied to an expression containing free variables). In this case, we return the equation $x = Sx0$, where the caller will deal with substituting the value of $x0$ if it is found in a later call.

Cases where x appears on both the left and the right side of the equation are dealt with by the unification algorithm, which described in the following subsection. Essentially, cases that reduce to "find $x : x = x$" return OK, meaning that all holes may be assigned any value, whereas cases like "find x : $x = S\ x$" return $\mathit{Failure}$, meaning that the equation is not solvable.

3. **e is a constructor applied to some expression: e = C e'.**

   First, we assume that the problem is sound; that is, the left and right sides are of the same type.

   The general problem is that, given an expression e, find concrete values c_i such that $e[v_i/c_i] = g$, where g is a concrete value that is the goal value. Now, it looks like "find concrete values c_i such that $C\ e'[v_i/c_i] = g$."

   There are three cases here.

   If e itself is a concrete value, then e contains no free variables. This might look like "find $(x : \mathit{bool}) : \mathit{True} = \mathit{True}$". If $e = g$, then return OK, meaning that it is safe to assign any value to the holes in the problem. (Here, x can be any boolean.) If $e \neq g$, then return $\mathit{Failure}$, meaning that there are no solutions, since the equation is not solvable.

   If not, check the head constructor of g. If it is not C, the equation is not solvable, so return $\mathit{Failure}$.

   If the head constructor of g is indeed C, then $g = C\ g'$. We return the solution of recursively solving this subproblem: "find concrete values c_i such that $e'[v_i/c_i] = g'$."

4. **e is a function application: e = f e'.**

   The problem looks like "find $c_i : (f\ e')[v_i/c_i] = g$." Let the function be defined as $fx = b$.

   First we substitute e' for the argument x in the function's body b, yielding $b' = b[x/e']$. (Note that this process will happen separate times, recursively, for multi-argument functions, since they are curried.) Also note that the implementation requires capture-avoiding substitution.

   Then we return the solution of recursively solving this subproblem after substitution: "find concrete values c_i such that $b'[v_i/c_i] = g$."

5. **e is a match statement.** In our limited language, discounting lists, e can have one of two forms:

   ```
   match e with
     | True -> e1
     | False -> e2

   match e with
     | Z -> e1
     | S n -> e2 (modulo naming of n)
   ```

   Dealing with match statements is the most complicated of the cases. We consider the match statement on natural numbers, since the booleans are a special case of it.

   Consider each branch b_i. First, we need to figure out how to fill the holes so b_i's leaf, l_i, can possibly return the goal value g. Then, if it might be able to, we need to figure out how to fill the holes to l_i in the first place. Then, we need to see if the two hole assignments are compatible with each other.

   Formally, consider a match statement m with n branches. Branch solutions cannot interfere with each other, so consider each branch separately. Consider branch b_i.

   We must first solve the first equation for it concerning its leaf. Let v_i be the free variables of l_i. Then we must solve "find $c_i : l_i[v_i/c_i] = g$." Note that the goal has not changed here. Let the result of trying to solve this be r_1.

   Next if r_1 is not $\mathit{Failure}$, we must solve the second equation for it concerning its guard. Let v_i be the free variables of e, and $g = S\ *$. (The $*$ denotes a wildcard, meaning that the goal only needs to start with the head constructor.) Then we must solve "find $c_i : e[v_i/c_i] = S\ *$." (Note that e may be a call to another function!) Let the result of trying to solve this be r_2.

   Now we have two sets of results for the branch, which are r_1 and r_2. They can be $\mathit{Failure}$, OK, concrete values, or constraints; for example, $r_1 = \{n = S\ n', n' = O, y = \mathit{True}\}$. We unify r_1 and r_2 using the unification algorithm described after this, and return the resulting set, r_i.

   This was the process for an individual branch. To finish, given equation sets r_i for all the branches, just pick an arbitrary r_i and return it. The algorithm can also return the disjunction of the equation sets; that is, any one will work.

1. *Examples*

To solve the following very simple problem:

```
let id x = x
find x : id x = 0
```

The algorithm substitutes x for x in the body of id, yielding the problem $find\ x : x = O$. It uses the assignment rule and simply returns $x = O$, which unifies to the same thing.

To solve the following problem:

```
let rec plus n1 n2 =
  match n1 with
  |  O -> n2
  | S n3 -> S (plus n3 n2)

find n : plus n (S O) = O
```

The algorithm substitutes n for $n1$ and $S\ O$ for $n2$ in $plus$:

```
find n :
(match n with
|  O -> S O
| S n3 -> S (plus n3 (S O))) = O
```

Inspecting the first terminal leaf, $S\ O$ can't be equal to the goal of O, so it discards that branch. Inspecting the second terminal leaf, something that starts with S can't be equal to the goal of O, so it discards that branch. Both branches failed, so the algorithm returns failure.

See the next section for a detailed trace of $list\_compress$.

2. Unification

Often the algorithm will not return concrete solutions like $x = 5$, but sets of equations involving variables and constructors on both sides. Unification is simply the process of merging all of the equations into a consistent, general solution that assigns some symbolic value to every variable present in the equations. Martelli et al. (1982) give an algorithm for first-order syntactic unification that either computes the most general unifier or reports that there is no solution. See Martelli et al. (1982) for a detailed explanation of the general unification problem and algorithm.

In practice we usually only encounter simple sets to unify like these:

```
l2 = Cons (n2, l3)
l4 = Nil
l2 = Cons (n2, l4)
```

We unify the bindings by rewriting with $l4$ in the last equation, then merging $l3$ with Nil. This is consistent, so we arrive at the single answer: $l2 = Cons\ (n2, l3)$.

Most problems we encounter are solvable using rewrites, merging values, failure on unequal head constructors, and failure when the *occurs check* fails. (The occurs check simply checks if a variable appears in the conclusion in an equation that does not reduce to $x = x$, such as $x = S\ x$, which is unsolvable.)

See the trace of $list\_compress$ for examples of unification in action.

3. More than one solution

There may be 0, 1, many but finite, or infinite solutions to a given problem. For brevity, we will omit a detailed study of this problem. One way to deal with many solutions is to simply save all the solutions found so far and pass them to the equation-solving algorithm (plus some maximum depth setting), and tell it to skip these solutions and find the next one, if possible.

One way to solve the problem of infinite solutions for a problem like $find\ x, y, z : O = O$, where having infinite solutions is detectable, is to simply return OK, meaning that any assignment of values to the free variables is okay. Another way to solve the problem of infinite, but constrained, solutions, is to return a symbolic solution such as $x = Sz$, where the solution contains free variables.

4. No solutions

When there are no solutions to a problem, the algorithm may fail nicely as in the $double$ problem. Or, it may not know when to stop searching for solutions, as in the following problem: $find\ x : sum\ x\ Z = S(SZ)$. There are finite solutions to the problem, and the algorithm can find all of them, but doesn't know when to stop searching. The trace is left as an exercise for the reader.

One way to stop divergence is to identify when subproblems to be solved are identical to the original problem up to substitution. If solving $find\ x : f\ x = 0$ requires eventually solving $find\ y : f\ y = 0$, then there is no solution. Otherwise, we have not deeply studied the divergence behavior of the algorithm.

5. Recursive functions

The algorithm can correctly solve $find\ m : double\ m = 4$, where $double$ is defined as such:

```
let rec double (n : nat) : nat =
  match n with
  | O -> O
  | S n0 -> S (S (double n0))
```

The key idea is to use capture-avoiding substitution when substituting an argument into a function body. That is, the problem given above becomes $find\ n0 : double\ n0 = 2$, and when substituted into the body, the problem becomes:

```
find n0 :
  (match n0 with
  | O -> O
  | S n1 -> S (S (double n1))) = 2
```

Note how the name in the pattern binding has changed to $n1$.

In addition, the algorithm fails correctly, without divergence, on $\mathit{find}\ m : \mathit{double}\ m = 5$, because after whittling down the result after some recursive calls, the problem reduces to $\mathit{find}\ m : \mathit{double}\ m = 1 = S\ O$. The algorithm fails gracefully here because the first branch returns O and the second branch returns something that starts with $S\ (S\ O)$, and $S\ O$ cannot equal either of them.

C. Symbolic execution

As defined earlier, this is the problem to solve: **Given any valid function f that contains only the constructs in the Pyth language, as well as an environment of types and functions, generate a set of inputs that traverse the function to reach every possible terminal leaf of the function.** (That is, given a particular terminal leaf, find some inputs that reach it.) If any terminal leaf cannot be reached, the algorithm should fail, either by returning failure (which is possible in many cases) or by looping forever.

Recall that arbitrarily nested matches are represented as trees. The core is this algorithm is simple tree traversal, where any node may have an arbitrary number of branches. Abstracting out the details, here is the pseudocode.

```
A tree is either a terminal leaf or a node with a
    list of subtrees.

The traverse function takes a tree and returns a
    list of every possible path to a terminal leaf.
     Proceed by casework on the structure of the
    tree.

Traversing a terminal leaf l returns a list
    containing a path which is the singleton list
    of l.

To traverse a tree that is a node n with a list of
    subtrees l, first recursively traverse each
    subtree in the list. This will return a list
    with one unnecessary level of nesting, so
    concatenate the list. This will yield a list of
     every possible path to a terminal leaf,
    excluding the current node, so simply add the
    current node to the head of each list. Return
    the resulting list of paths.
```

After we have the list of paths, the algorithm performs postprocessing on it, mainly to fill in wildcards and unused arguments.

We describe the algorithm by doing a trace of its process on a complicated sample input function f. One difficulty is that f may ignore some of its arguments; for example, it might not match on one, but return it as a leaf.

Recall that f may either match on inputs or on function calls. We first walk through the algorithm on an f that does not match on a function call. Then we extend it to an f that does match on a function call.

First is f that matches on arguments, not on a function call. Let $(a = c_1) => (b = c_2)$ denote the fact that f matched on a first, then b, so that vector of assignments describes a unique path through the function to some leaf (which may not be a terminal leaf).

```
let f (x : nat) (y : bool) (z : nat) =
   match z with
     | O ->
        (match x with
             | O -> leaf1
             | S x0 -> leaf2)
      | S z0 ->
        (match y with
             | True -> leaf3
             | False ->
               (match x with
                 | O -> leaf4
                 | S x1 -> leaf5))
```

1. First, traverse the tree to reach all terminal leaves, as described above. We also record all bindings of match guards to patterns.

```
(z = O) => (x = O)
(z = O) => (x = S x0)
(z = S z0) => (y = True)
(z = S z0) => (y = False) => (x = O)
(z = S z0) => (y = False) => (x = S x1)
```

2. Note that in all paths starting in the first match statement, f did not match on y. In general, f may ignore any number of its arguments. So, for each path, find the unused variables and assign them the value of OK, which denotes that they may be anything. Now every path should contain all variables in some order, so sort them into the input order (here, x then y then z). All paths should be of the same length.

```
(x = O) => (y = OK) => (z = O)
(x = S x0) => (y = OK) => (z = O)
(x = OK) => (y = True) => (z = S z0)
(x = O) => (y = False) => (z = S z0)
(x = S x1) => (y = False) => (z = S z0)
```

3. This is the first step of filling in wildcards. The OKs may be any value, and we know the type of the variable and how to construct something of that type, so fill in all the OKs with an arbitrary value. (One extension is to fill in OKs not just with one value, but with many, using all the type's constructors, so $y = OK$ would become both $y = \mathit{True}$ and $y = \mathit{False}$, replicating the rest of the assignments in its path.)

```
(x = O) => (y = True) => (z = O)
(x = S x0) => (y = True) => (z = O)
(x = O) => (y = True) => (z = S z0)
(x = O) => (y = False) => (z = S z0)
(x = S x1) => (y = False) => (z = S z0)
```

4. This is the second step of filling in wildcards. Any free variable in a value in this scenario is a wildcard, e.g. $x1$ in $x = S\ x1$. Do the same as in the previous step. It can also be extended in the same way as the previous step.

```
(x = O) => (y = True) => (z = O)
(x = S O) => (y = True) => (z = O)
(x = O) => (y = True) => (z = S O)
(x = O) => (y = False) => (z = S O)
(x = S O) => (y = False) => (z = S O)
```

5. Group the first argument x by head constructor, then recursively repeat on the rest of the arguments.

```
(x = O) => (y = True) => (z = O)
(x = O) => (y = True) => (z = S O)
(x = O) => (y = False) => (z = S O)
(x = S O) => (y = True) => (z = O)
(x = S O) => (y = False) => (z = S O)
```

At this stage, all paths are the same length and contain all the arguments in the input order.

Lemma. All paths are unique.

Proof sketch. All paths are unique before *OK* instantiation, since match patterns must partition the guards into different head constructors, e.g. *True* vs. *False*.

After *OK* instantiation, all paths remain unique, since paths that were different before cannot become the same, and the same path will only have its *OK*s instantiated with different constructors (see the extension mentioned earlier), so it cannot create two of the same path. The same argument holds for free variable instantiation. □

6. All inputs should have concrete values, all paths should be unique, and all paths should have arguments in input order, so everything is ready for us to evaluate the function on each path. These paths serve as generated tests that ensure complete code coverage, and the user may note which paths cause the function to throw an exception or otherwise fail.

```
(x = O) => (y = True) => (z = O) => leaf1
(x = O) => (y = True) => (z = S O) => leaf3
(x = O) => (y = False) => (z = S O) => leaf4
(x = S O) => (y = True) => (z = O) => leaf2
(x = S O) => (y = False) => (z = S O) => leaf5
```

Lastly, recall that the very motivation of developing the equation-solving algorithm was to enable us to perform symbolic execution on *list_compress*:

```
let rec list_compress (l1 : list) : list =
    match l1 with
      | Nil -> Nil
      | Cons (n1, l2) ->
      (match f1 l2 with
        | Nil -> l1
          | Cons (n2, l3) ->
            (match compare n2 n1 with
             | LT -> Cons (n1, Cons (n2, l3)) (*)
             | EQ -> Cons (n1, l3)
             | GT -> Cons (n1, Cons (n2, l3))))
```

because it matched on a function call, not a variable. Bare-bones symbolic execution cannot figure out how to reach the terminal leaf marked (∗), for example. To do that, we must solve this equation:

```
find n2, n1 : compare n2 n1 = LT
```

And solving this equation will require us to perform symbolic execution on the *compare* function itself, to figure out how to reach the terminal leaf marked (∗∗).

```
let rec compare (n1:nat) (n2:nat) : cmp =
  match n1 with
  | O ->
    (match n2 with
     | O -> EQ
     | S m -> LT (**))
  | S m1 ->
    (match n2 with
     | O -> GT
     | S m2 -> (compare m1 m2))
```

Again, the two algorithms are mutually recursive. For a detailed trace of how symbolic execution and equation-solving interact, see the trace of *list_compress* in Section V.

D. Input-output example generation

Here we turn from functional testing to functional synthesis. To transform the tests to input-output examples, we simply nest examples recursively by the same value, which is the format that requires. Since we have already recursively sorted the paths by value, this is easy.

```
(x = O) => (y = True) => (z = O) => leaf1
(x = O) => (y = True) => (z = S O) => leaf3
(x = O) => (y = False) => (z = S O) => leaf4
(x = S O) => (y = True) => (z = O) => leaf2
(x = S O) => (y = False) => (z = S O) => leaf5

  O => {  True => {  O => leaf1,
                    S O => leaf3 } ,
         False => { S O => leaf4 }},
S O => {   True => {  O => leaf2 }}
       { False => { S O => leaf5 }}
```

1. Augmentation

As discussed in the problem statement, there are three main ways to augment the input-output examples derived from symbolic execution, because they are sometimes not enough to re-synthesize the function.

First, we generate recursive subexamples. That is, for a list [3; 2; 1], we automatically add [2; 1], [1], and [] to our list of inputs. This is because Myth is often not able to deal with functions that recurse on smaller lists without knowing what to do on the smaller lists themselves. We do this for any recursive type where the function recurses on a smaller value of the type.

First, the more we enumerate any free variables left in the input examples, the more inputs we have for Myth. Filling in these wildcards gives examples that fall in the same equivalence class; that is, they traverse the same path, as discussed in the DART/SAGE section.

Indeed, in general, the idea of generating another "structurally similar" example that traverses exactly the same path as an existing example is important. This is because these examples together signal to Myth to find a deeper structure and not overfit to constants. This can be done in several ways; for example, one can ask the equation-solving algorithm to find several solutions if they exist, and not just one.

The augmentation process is described concretely at the end of the next section on *list_compress*.

V. RE-SYNTHESIZING *list_compress*

Here we give a detailed trace of the algorithm and its successes and failures on the most complicated Myth benchmark, which is *list_compress*. After that, we briefly summarize the capabilities of the various implementations in terms of benchmarks.

First, we describe the process of backwards symbolic execution on *list_compress*. Then we postprocess the generated inputs, which yields tests for the function. Then we augment the input-output examples and examine the process of re-synthesizing *list_compress*.

A. Backwards symbolic execution

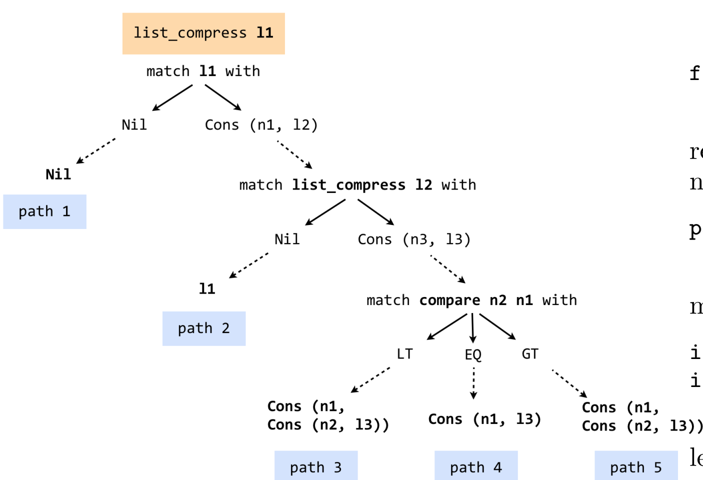

Figure 5. Paths of *list_compress*.

We will examine each of the five paths, which end at each of the five terminal leaves, as labeled in the diagram. On each path, the algorithm collects a set of equations which must be satisfied in order to reach that terminal leaf. Each equation is solved individually, yielding a set of bindings for each.

An equation may be found at the location at any solid arrow on the diagram. It looks like *match guard with* → *pattern* and yields the equation "find free variables of the guard such that the evaluated guard equals the pattern." Equations can be broadly classified as "easy," meaning they only involve variable and constructors, or "hard," meaning they involve calls to other functions, and those functions may have match statements themselves.

Finally, after solving each equation for a set of constraints, unification is used on that set to yield a final set of constraints or a concrete value for each input that will allow that terminal leaf to be reached. This process will become clearer as it is described concretely for each path.

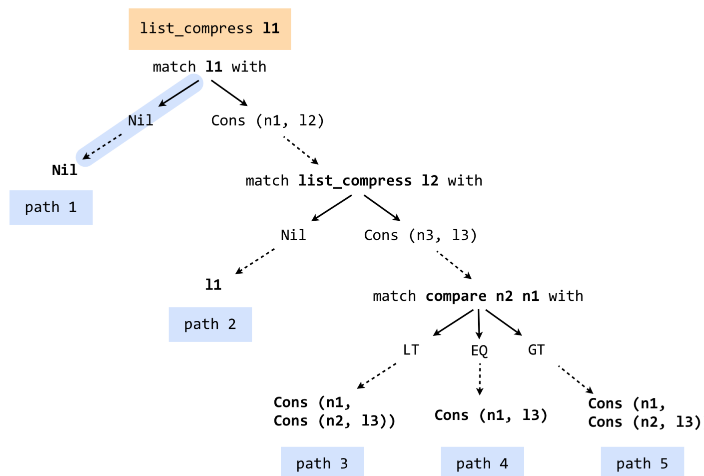

Figure 6. Path 1.

For path 1, the only equation collected is

```
find l1 : l1 = Nil.
```

This is passed to the equation-solving algorithm, which returns the only solution, which is a concrete value with no free variables:

```
path 1: (l1 = Nil).
```

For path 2, the equations collected (in order from terminal leaf to root) are:

```
i. find l2 : list_compress l2 = Nil
ii. find l1 : l1 = Cons (n1, l2)
```

Intuitively, they are doing "can I get to this terminal leaf?" followed by "can I get to the leaf (match statement) that contains the terminal leaf?"

For *i*, the equation-solving algorithm substitutes *l*1 into the body of *list_compress* and looks for a leaf that

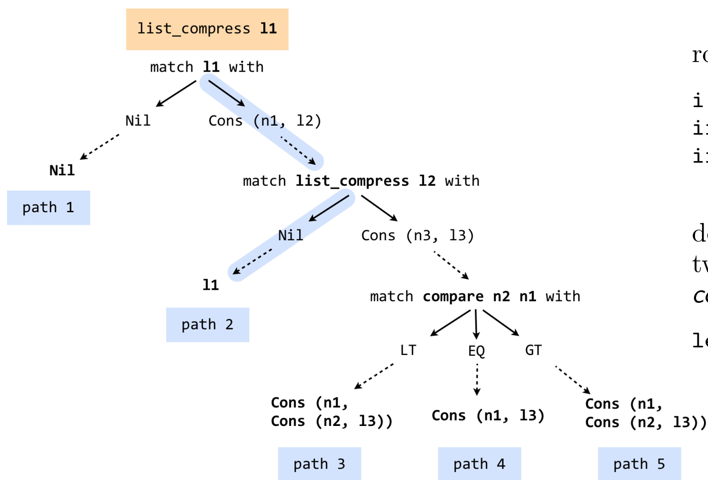

Figure 7. Path 2.

might return *Nil*. It finds that leaf, which returns *Nil* if the equation $\mathit{find}\, l2 : l2 = \mathit{Nil}$ is solved. It returns $l2 = \mathit{Nil}$.

For *ii*, the equation-solving algorithm simply assigns the value to $l1$ and returns $l1 = \mathit{Cons}(n1, l2)$.

The solutions to *i* and *ii* result in two bindings:

```
l2 = Nil
l1 = Cons (n1, l2)
```

on which we run the unification algorithm, resulting in the solution $l1 = \mathit{Cons}(n1, \mathit{Nil})$, where $n1$ is a free variable, and so may be anything. So, a list like [5] would traverse path 2 in *list_compress*.

The two easy cases done, we now consider path 4, followed by paths 3 and 5 together.

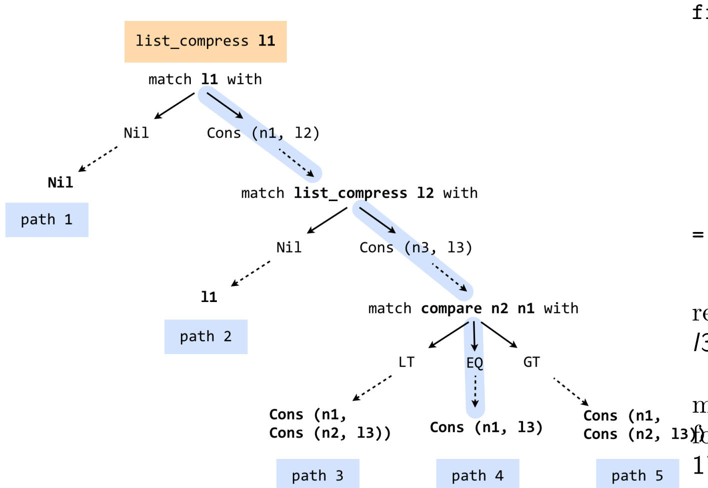

Figure 8. Path 4.

Thinking about path 4 intuitively, it is the path that actually compresses the list by only *Cons*ing one of two duplicate elements.

The equations collected here, again listed from tip to root order, are:

```
i. find n1, n2 : compare n1 n2 = EQ
ii. find l2 : list_compress l2 = Cons (n2, l3)
iii. find l1 : l1 = Cons (n1, l2)
```

We summarize what the equation-solving algorithm does on each. On equation *i*, we basically need to find two equal nats. The algorithm inspects the leaves of the *compare* function:

```
let rec compare (n1:nat) (n2:nat) : cmp =
  match n1 with
  | O ->
    (match n2 with
     | O -> EQ
     | S m -> LT)
  | S m1 ->
    (match n2 with
     | O -> GT
     | S m2 -> (compare m1 m2))
```

and sees that the function returns *EQ* when $n2 = O$ and $n1 = O$. It returns the first solution it finds, so it returns this. Note that there are infinite solutions if we search the recursive call $\mathit{compare}\ m1\ m2$; for example, the algorithm could find $n2 = SO$ and $n1 = SO$.

Equation *ii* is tricky because it requires us to search *list_compress* twice. This is the problem:

```
ii. find l2 : list_compress l2 = Cons (n2, l3)
```

The equation-solving algorithm substitutes the argument into the function body in a manner that avoids capturing free variables in the function body or in the equation itself. Note that $n2$ and $l3$ are free in the equation.

After substitution, the new problem is:

```
find l2 :
    (match l2 with
     | Nil -> []
     | Cons (n1, l4) (* fresh name *) ->
         (match list_compress l4 with
         | Nil -> l2
         | Cons (n3, l5) -> (* fresh names *)
               ...omitted... ))
= Cons (n2, l3)
```

Note that $l2$ in the original function body has been renamed to a fresh variable $l4$, and the same for renaming $l3$ to $l5$ and $n2$ to $n3$.

Now, the equation-solving algorithm inspects the terminal leaves of the match statement in figures 7 and 8 for the goal expression $\mathit{Cons}\ (n2, l3)$. Looking at path 1's terminal leaf here fails because returning *Nil* cannot possibly return $\mathit{Cons}\ (n2, l3)$.

Now the equation-algorithm tries the terminal leaf of path 2 here. It sees that the terminal leaf is $l2$ and thinks, OK, $l2$ is a variable. If I assign $l2 = \mathit{Cons}\ (n2, l3)$ (the goal value), then I will succeed. Let's see if I can indeed do that, and what I will need to do in order to reach this terminal leaf in the first place.

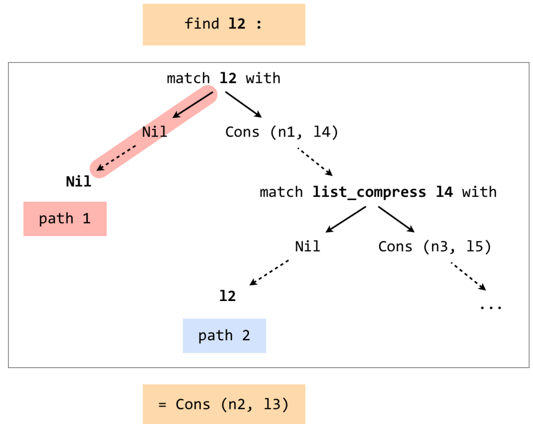

Figure 9. First attempt to solve subproblem fails.

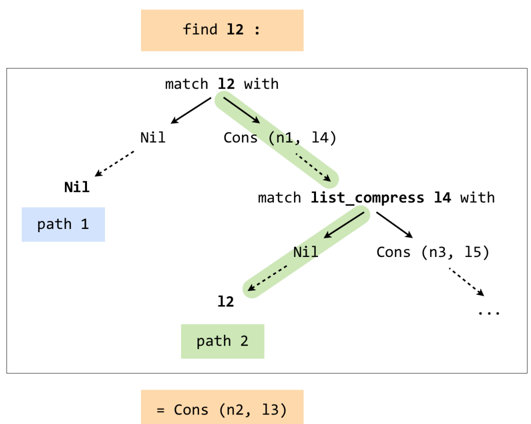

Figure 10. Second attempt to solve subproblem succeeds.

This is exactly a symbolic execution problem, since we need to figure out what inputs are needed to traverse path 2, so we call that algorithm in a mutually recursive fashion. It collects the following constraints:

```
ia. find l4 : list_compress l4 = Nil
iia. find l2 : l2 = Cons (n2, l4).
```

Equation *ia* was already discussed when we solved path 1 for *list_compress* earlier; $l4 = Nil$. The second equation is the simple assignment $l2 = Cons\ (n2, l4)$. Recall that earlier the equation-solving algorithm guessed that we could assign $l2 = Cons\ (n2, l3)$ (the goal value) to succeed, so it added it to the set of bindings. Now, the final set of bindings is:

```
l2 = Cons (n2, l3)
l4 = Nil
l2 = Cons (n2, l4)
```

The unification algorithm unifies the bindings by rewriting with $l4$, then seeing that $l3 = Nil$. This is consistent, so we arrive at the single answer: $l2 = Cons\ (n2, l3)$.

Finally, equation *ii* is done. It was the most difficult of the three, since it involved a function call.

Lastly, equation *iii* is easy to solve. We simply return $l1 = Cons\ (n1, l2)$.

Now, we go back up a level to examine the union of the binding set from all equations:

```
i. n1 = O, n2 = O
ii. l2 = Cons (n2, l4)
iii. l1 = Cons (n1, l2)
```

Using unification, one can rewrite $l2$ in *iii* to yield $Cons\ (n1, Cons\ (n2, l3))$, then rewrite $n1$ and $n2$ to yield the final answer of $l1 = Cons\ (O, Cons\ (O, l3))$. As a sanity check, yes, this answer makes sense! This list will clearly be compressed due to the duplicate elements at the head. So, it does indeed reach path 4.

Note two things about this solution. First, $l3$ is free, so it could be anything. Second, a structurally similar input—that is, an input that is different but would traverse exactly the same path—would be $l1 = Cons\ (n, Cons\ (n, l3))$ for any nat n, and with the same $l3$. These could be found by searching deeper into the recursive call in the *compare* function, as mentioned earlier.

Paths 3 and 5 are very similar to each other and to path 4. Thus, the trace for paths 3 and 5 is left as an exercise for the reader. Their solutions are listed below.

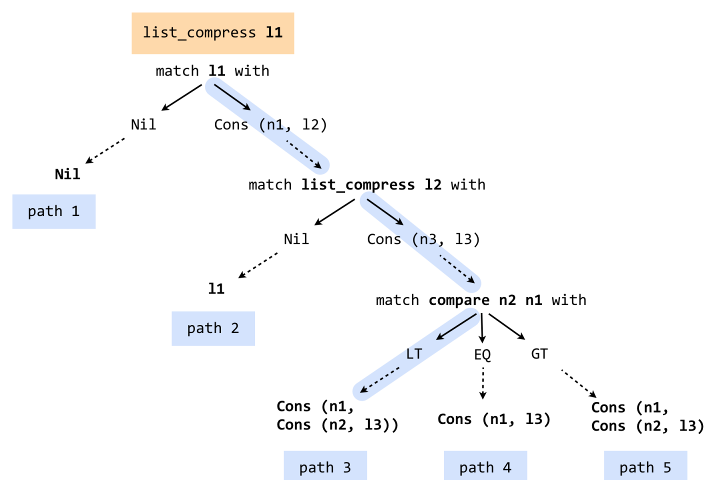

Figure 11. Path 3 (exercise for reader, along with path 5).

```
Path 3: l1 = Cons (O, Cons (S O, l3)
Path 5: l1 = Cons (S O, Cons (O, l3))
```

B. Augmenting inputs for synthesis

After solving the five paths, we have five general shapes that $l1$ needs to take (letting pn = path n):

```
p1: l1 = []
p2: l1 = Cons (n1, Nil)
```

```
p3: l1 = Cons (0, Cons (S m, l3))
p4: l1 = Cons (0, Cons (0, l3))
p5: l1 = Cons (S m, Cons (0, l3))
```

Note that these are not concrete inputs yet, since they contain free variables. First, we fill the free variables.

Let's try filling them with only one value, and the simplest one possible. (This can be done algorithmically, since we know the type of a variable, and how to construct a value of the type.) So all nats will get O and all lists will get Nil, with no additional augmentation.

```
p1: l1 = []
p2: l1 = Cons (0, Nil)
RSE: l1 = Cons (1, Nil)
p3: l1 = Cons (0, Cons (1, Nil))
p4: l1 = Cons (0, Cons (0, Nil))
p5: l1 = Cons (1, Cons (0, Nil))
```

We also add a recursive sub-example (RSE) which is a sub-example for $p4$, so Myth doesn't complain about not knowing what to do on the smaller list. We then evaluate the function on all inputs to get outputs, for use in input-output examples. Now try to guess what Myth will synthesize!

For reference, here is the original $list\_compress$ function:

```
let rec list_compress (l1 : list) : list =
    match l1 with
      | Nil -> Nil
      | Cons (n1, l2) ->
      (match f1 l2 with
        | Nil -> l1
        | Cons (n2, l3) ->
           (match compare n2 n1 with
            | LT -> Cons (n1, Cons (n2, l3))
            | EQ -> Cons (n1, l3) (*)
            | GT -> Cons (n1, Cons (n2, l3))))
```

And here is the bare-bones re-synthesized function:

```
let rec list_compress (l1 : list) : list =
    match l1 with
      | Nil -> []
      | Cons (n1, l2) ->
        (match l2 with
          | Nil -> l1
          | Cons (n2, l3) ->
             match compare n2 n1 with
               | LT -> [1; 0]
               | EQ -> [0]
              | GT -> [0; 1]))
```

This is a very interesting result. We immediately have most of the *structure* of the original function, including the call to compare. However, the input lists are not long enough for Myth to recurse, which is why it matches on $l2$ instead of $list\_compress l2$. Lastly, Myth has also overfitted and is returning constants instead of symbolic expressions.

Let's try to solve the constants problem by filling in wildcards (free variables) with two values instead of one. Lists are hard to deal with, so let's just do it for the nats, and deal with $l3$ later. In practice, this means replacing one symbolic example with two, one containing 0 and the other containing 1 in place of the variable. This approximates telling Myth, "hey, find a deeper structure than a constant!" Also, we will add the recursive sub-example $Cons\ (2, Nil)$. The two-nat-wildcard inputs are:

```
p1: l1 = []
p2: l1 = Cons (0, Nil)
p2: l1 = Cons (1, Nil)
RSE: l1 = Cons (2, Nil)
p3: l1 = Cons (0, Cons (1, Nil))
p3: l1 = Cons (0, Cons (2, Nil))
p4: l1 = Cons (0, Cons (0, Nil))
p5: l1 = Cons (1, Cons (0, Nil))
p5: l1 = Cons (2, Cons (0, Nil))
```

And the synthesized function is:

```
let rec list_compress (l1 : list) : list =
    match l1 with
      | Nil -> []
      | Cons (n1, l2) ->
        (match l2 with
          | Nil -> l1
          | Cons (n2, l3) ->
            (match compare n2 n1 with
             | LT -> l1
             | EQ -> [0]
             | GT -> l1))
```

Two of the constants are gone! This version is quite close to the expected version of $list\_compress$. We just need longer lists and a structurally similar example for $[0; 0]$, since it is the only example in the list right now that gets compressed. Recall that we mentioned this earlier—recursively searching $compare$ for path 4 can generate more structurally similar examples. We add the first one we find, which is $[1; 1]$.

```
p1: l1 = []
p2: l1 = Cons (0, Nil)
p2: l1 = Cons (1, Nil)
RSE: l1 = Cons (2, Nil)
p3: l1 = Cons (0, Cons (1, Nil))
p3: l1 = Cons (0, Cons (2, Nil))
p4: l1 = Cons (0, Cons (0, Nil))
p5: l1 = Cons (1, Cons (0, Nil))
p5: l1 = Cons (2, Cons (0, Nil))
compare: l1 = Cons (1, Cons (1, Nil))
```

This is the synthesized function:

```
let rec list_compress (l1 : list) : list =
    match l1 with
      | Nil -> []
      | Cons (n1, l2) ->
        (match l2 with
          | Nil -> l1
          | Cons (n2, l3) ->
            (match compare n2 n1 with
             | LT -> l1
             | EQ -> l2
             | GT -> l1))
```

Our guess was right—the constant has gone away and all outputs are symbolic! We're extremely close, but still missing the last thing. We need longer list inputs to force *list_compress* to match on a recursive call to deal with the rest of the list, instead of just matching on $l2$ and returning $l1$ or $l2$.

We can do this by augmenting the $l3$s that we left out earlier and only replaced with *Nil*s, now adding both $l3 = Cons\ (0, Nil)$ and $l3 = Cons\ (1, Nil)$ (again, to avoid constant overfitting) and any recursive subexamples needed.

We started with this general set, and we will augment the starred inputs, also including our previous augmentations.

```
p1: l1 = []
p2: l1 = Cons (n1, Nil)
p3: l1 = Cons (0, Cons (S m, l3)) *
p4: l1 = Cons (0, Cons (0, l3)) *
p5: l1 = Cons (S m, Cons (0, l3)) *


(* original set *)
p1: l1 = []
p2: l1 = Cons (0, Nil)
p2: l1 = Cons (1, Nil)
RSE: l1 = Cons (2, Nil)
p3: l1 = Cons (0, Cons (1, Nil))
p3: l1 = Cons (0, Cons (2, Nil))
p4: l1 = Cons (0, Cons (0, Nil))
p5: l1 = Cons (1, Cons (0, Nil))
p5: l1 = Cons (2, Cons (0, Nil))
comp: l1 = Cons (1, Cons (1, Nil))
RSE2: l1 = [2;1]

(* l3 = [0] *)
p3: l1 = Cons (0, Cons (1, Cons (0, Nil)))
p3: l1 = Cons (0, Cons (2, Cons (0, Nil)))
p4: l1 = Cons (0, Cons (0, Cons (0, Nil)))
p5: l1 = Cons (1, Cons (0, Cons (0, Nil)))
p5: l1 = Cons (2, Cons (0, Cons (0, Nil)))
comp: l1 = Cons (1, Cons (1, Cons (0, Nil)))

(* l3 = [1] *)
p3: l1 = Cons (0, Cons (1, Cons (1, Nil)))
p3: l1 = Cons (0, Cons (2, Cons (1, Nil))) (* needs
    new RSE *)
p4: l1 = Cons (0, Cons (0, Cons (1, Nil)))
p5: l1 = Cons (1, Cons (0, Cons (1, Nil)))
p5: l1 = Cons (2, Cons (0, Cons (1, Nil)))
comp: l1 = Cons (1, Cons (1, Cons (1, Nil)))
```

This is the result:

```
let rec list_compress (l1 : list) : list =
    match l1 with
      | Nil -> []
      | Cons (n1, l2) ->
        (match list_compress l2 with
          | Nil -> l1
          | Cons (n2, l3) ->
            (match compare n2 n1 with
             | LT -> Cons (n1, Cons (n2, l3))
             | EQ -> Cons (n1, l3)
             | GT -> Cons (0, Cons (n2, l3))))
             (* oops *)
```

We only differ from *list_compress* by the constant 0 in the *GT* branch. This is probably happening because, when we solved

```
find n2, n1 : compare n2 n1 = GT
```

in path 3, we returned the first solution we found: $n2 = S\ m; n1 = 0$. This is causing the function to overfit on the constant 0. This behavior is similar to earlier, when we stopped searching *compare* too early on *EQ*. The next solution to this equation is $n2 = 1; n1 = 2$. If we add $[1; 2]$ to the long list of inputs, it does indeed force the full *list_compress* to be synthesized! (As a sanity check, adding $[1; 2]$ does not help, or change the function.)

```
let rec list_compress (l1 : list) : list =
    match l1 with
      | Nil -> []
      | Cons (n1, l2) ->
        (match list_compress l2 with
          | Nil -> l1
          | Cons (n2, l3) ->
            (match compare n2 n1 with
             | LT -> Cons (n1, Cons (n2, l3))
             | EQ -> Cons (n1, l3)
             | GT -> Cons (n1, Cons (n2, l3))))
```

C. Human-written examples

Here is our final list:

```
let list_compress : list -> list |>
{ [] => []
| [0] => [0]
| [1] => [1]
| [2] => [2]
| [0;1] => [0;1]
| [0;2] => [0;2]
| [1;0] => [1;0]
| [2;0] => [2;0]
| [0;0] => [0]
| [1;1] => [1]
| [0;1;0] => [0;1;0]
| [0;2;0] => [0;2;0]
| [1;0;0] => [1;0]
| [2;0;0] => [2;0]
| [0;0;0] => [0]
| [1;1;0] => [1;0]
| [2;1] => [2;1]
| [0;1;1] => [0;1]
| [0;2;1] => [0;2;1]
| [1;0;1] => [1;0;1]
| [2;0;1] => [2;0;1]
| [0;0;1] => [0;1]
| [1;1;1] => [1]
| [1;2] => [1;2]
} = ?
```

Are any of these *extraneous* examples? There must be, because here is a set of human-written examples that works:

```
let list_compress : list -> list |>
```

```
{ [] => []
| [0] => [0]
| [3] => [3]
| [2] => [2]
| [1] => [1]
| [1;0] => [1;0]
| [3;2] => [3;2]
| [3;3;2] => [3;2]
| [0;0] => [0]
| [3;3] => [3]
| [3;3;3] => [3]
| [0;1] => [0;1]
| [0;0;1] => [0;1]
| [2;3] => [2;3]
| [2;2;3] => [2;3]
} = ?
```

There are only 15 of them, but 23 of them in our algorithmically- and ad-hoc-generated set. The human-written set may or may not be minimal, but it is true that removing any one of them causes the synthesis to fail. The reader is invited to compare the two sets of examples and see which ones are isomorphic, and which ones differ.

D. Other benchmarks

The implementation for the equation-solving algorithm has 37 benchmarks, and they are available in the repository. The implementation is about 500 lines of OCaml code, and it uses the Myth language parser. The most advanced problems it can currently solve involve natural numbers and recursive functions. For example, it can correctly solve *find m* : *double m* = 8 in this input format:

```
let rec double (n : nat) : nat =
  match n with
  | O -> O
  | S n0 -> S (S (double n0))

let problem (m : nat) : nat = double m

let result : nat = 8
```

for which it returns $m := 4$. Additionally, it fails gracefully on *find m* : *double m* = 7 by returning "No solution found."

The implementation for the symbolic execution algorithm has 12 benchmarks, and they are available in the repository. The implementation is about 200 lines of OCaml code, also using the Myth language parser. The most advanced problem it can solve is along the lines of "traverse the *compare* function":

```
let rec compare (n1:nat) (n2:nat) : cmp =
  match n1 with
  | O ->
    (match n2 with
    | O -> EQ
    | S m -> LT)
  | S m1 ->
    (match n2 with
    | O -> GT
    | S m2 -> (compare m1 m2))

let toTraverse : nat -> nat -> cmp =
  compare
```

It successfully returns these paths:

```
* (n1, O) (n2, O)
* (n1, O) (n2, S m)
* (n1, S m1) (n2, O)
* (n1, S m1) (n2, S m2)
```

VI. CONCLUSION

We extend Godefroid et al.'s work on DART and SAGE from imperative languages to typed functional languages. By doing so, we draw parallels between C-like language features and ML-like language features, as well as the methods needed to deal with both. Godefroid et al. work with primitive types such as integers that appear in if-statements. They collect constraints from the if-statement guards and solve them with a SAT solver. On the other hand, we work with algebraic data types that are deconstructed by pattern-matching, and we collect equations from the pattern-matching guards and solve them with unification.

We introduced two new algorithms, an equation-solving algorithm and a backwards symbolic execution algorithm. In the tradition of whitebox testing, given a function, these algorithms together generate input tests with 100% code coverage for it. We also introduce a new example augmentation algorithm that, given these tests, can augment them into input-output examples that attempt to induce Myth to re-synthesize the original function.

We study the successes and failures of these algorithms on many test cases. In particular, when de-synthesizing a function into examples and trying to re-synthesize the function from these examples, we observe an interesting divergence between the original and the synthesized *nat_max*. Lastly, after much augmentation, we also succeed in re-synthesizing the complex Myth benchmark of *list_compress*.

A. Future work

There are directions for future work in each topic covered by this paper. Within the topic of solving functional equations, one could extend the equation-solving algorithm to deal with multiple holes, solve holes in the goal, detect divergence, find infinite solutions, and handle arbitrary user-defined algebraic data types. This is broadly related to relational programming, in particular as implemented in the programming language miniKanren, which can solve more general equations (Byrd (2009)). Combining the two approaches, the goal would be to solve complex equations such as the following:

```
find (n m : nat) : add n m = mult n m
find (t : tree) : reflectOverYAxis t = t
find (t : abstractSyntaxTree) : eval t = Const 5
```

Next, there is very little work done on symbolic execution for typed functional programs. Being able to whitebox-fuzz large functional systems to find crashes, as well as generate thorough tests for use as regression tests, are useful applications. One could apply the work in this paper to empirically test large functional systems for known bugs and compare the results to those of blackbox fuzzers. This would be similar to the work on SAGE, which was tested on large Microsoft codebases and found bugs that caused critical security vulnerabilites that had evaded both blackbox fuzzing and static analysis.

Lastly, one could extend the work here on finding input-output examples sufficient to synthesize a program. One interesting line of inquiry is to find an example set of minimal size and complexity. Or, one could find a *necessary* example set, where every example is necessary, but may not be sufficient. (The de-synthesis in this paper produces examples that are not always both necessary and sufficient.) Solving these problems could be put to good use in characterizing the size and complexity of a program in an information-theoretic way, parametrized by a program synthesis system. That is, given a system like Myth, characterize the amount of information a program contains in terms of the quality and quantity of examples needed to synthesize it.

ACKNOWLEDGMENTS

First, I owe a lot to Prof. David Walker, who was a great advisor. He stuck with me through many iterations of research ideas, taught me about judgements and typing rules, and demonstrated how to find simple algorithms and counterexamples.

Next, I'd like to thank Jonathan Frankle for helping me get started with Myth and explaining the synthesis rules. I also would like to thank Peter-Michael Osera and Prof. Steve Zdancewic of UPenn for graciously allowing me to join their project and for answering my questions on the synthtype list.

Lastly, thanks to Mark Jason Dominus for helping me notice that unification was exactly what I needed to complete the equation-solving algorithm. Thanks to Alex Clemmer for insightful comments on the problem of finding minimal example sets for re-synthesis. Omar Rizwan provided useful comments on the final draft of this paper, and T. Dickinson provided an abundance of miso soup.

---

[1] Byrd, William E. *Relational programming in minikanren: Techniques, applications, and implementations.* Diss. faculty of the University Graduate School in partial fulfillment of the requirements for the degree Doctor of Philosophy in the Department of Computer Science, Indiana University, 2009.

[2] Godefroid, Patrice, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." *ACM Sigplan Notices.* Vol. 40. No. 6. ACM, 2005.

[3] Godefroid, Patrice, Michael Y. Levin, and David A. Molnar. "Automated Whitebox Fuzz Testing." *NDSS.* Vol. 8. 2008.

[4] Godefroid, Patrice, Michael Y. Levin, and David Molnar. "SAGE: whitebox fuzzing for security testing." *Queue* 10.1 (2012): 20.

[5] Gulwani, Sumit. "Dimensions in program synthesis." *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming.* ACM, 2010.

[6] Manna, Zohar, and Richard Waldinger. "A deductive approach to program synthesis." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2.1 (1980): 90-121.

[7] Martelli, Alberto, and Ugo Montanari. "An efficient unification algorithm." *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.2 (1982): 258-282.

[8] Osera, Peter-Michael, and Steve Zdancewic. "Type-and-Example-Directed Program Synthesis." *Programming Language Design and Implementation (PLDI).* 2015.

[9] Solar-Lezama, Armando. Program synthesis by sketching. *ProQuest*, 2008.

[10] Srivastava, Saurabh, Sumit Gulwani, and Jeffrey S. Foster. "From program verification to program synthesis." *Principles of Programming Languages (POPL).* 2010.

Appendix A: Pyth

The code for the equation-solving and symbolic execution algorithms is available at `github.com/hypotext/synthesis`.

The rest of the appendices highlights important sections of the code, then gives two algorithm-generated benchmark traces.

Appendix B: Myth's grammar

```
type id = string

type typ =
  | TBase of id
  | TArr of typ * typ
  | TTuple of typ list (* Invariant: List must
      always have two members. *)
  | TUnit
```

```
type ctor = id * typ

type pattern =
  | PWildcard
  | PVar of id
  | PTuple of pattern list

type pat = id * (pattern option) (* (id of
    constructor, pattern). *)

type arg = id * typ

type env = (id * value) list

and decl =
  | DData of id * ctor list
  | DLet of id * bool * arg list * typ * exp

and exp =
  | EVar of id
  | EApp of exp * exp
  | EFun of arg * exp
  | ELet of id * bool * arg list * typ * exp * exp
  | ECtor of id * exp
  | EMatch of exp * branch list
  | EPFun of (exp * exp) list
  | EFix of id * arg * typ * exp
  | ETuple of exp list (* Invariant: List must
      always have two members. *)
  | EProj of int * exp (* int is the index of proj.
      of the tuple (1-indexed). *)
  | EUnit

and branch = pat * exp

and value =
  | VCtor of id * value
  | VFun  of id * exp * env ref (* ref? *)
  | VPFun of (value * value) list
  | VTuple of value list (* Invariant: List must
      always have two members. *)
  | VUnit
```

Appendix C: Types in the equation-solving code

These are the types for the current implementation, which uses the Myth types defined above. The implementation is limited in scope and does not reflect the full generality of the algorithm given in the paper. In particular, it does ad-hoc unification on one hole, not general unification involving many holes.

```
type var = string
type hole = var
type solution =
  | Concrete of (hole * exp) list | OK
type binding =
    { name     : exp;
      value    : exp; }

type constraint_problem =
    { holes    : hole list;
      types    : decl list;
      fns      : decl list;
      problem  : exp;
      goal     : exp;

      (* State *)
      depth    : int;
      prevSols : solution list;
      bindings : binding list;
      (* e.g. {l0 = Cons (x0, xs0), x0 = 1...}*)
    }
```

Appendix D: Types in the symbolic execution code

These are the types for the current implementation, which is limited in scope and does not reflect the full generality of the algorithm given in the paper.

```
type var = string
type inputVal =
  | Impossible
  | Any
  | Some of exp
type inputAssn = var * inputVal
type output = exp

(* e.g. [("x", O) => ("b", True)] (=> being a list
    semicolon) *)
type inputRow = inputAssn list
type inputTable = inputRow list
(* e.g. ( [("x", O) => ("b", True)] => S O) *)
type ioList = (inputAssn list * output) list

module M = Map.Make(String);;
type argsUsedMap = bool M.t;;

type traverse_problem =
    { name       : string;
      body       : exp;
      isRec      : bool;
      args       : arg list;
      fnType     : typ;
      fns        : decl list;
      types      : decl list;
      inputTable : inputTable;
      argsUsed   : argsUsedMap;
      depth      : int;
    }
```

Appendix E: Full list of benchmarks

Equation-solving benchmarks:

1. *bool_and_not_fail*
2. *bool_asgn*
3. *bool_asgn_fail*
4. *bool_id*
5. *bool_match_const_none*

6. *bool_match_const_ok*
7. *bool_match_id*
8. *bool_neg*
9. *bool_xor_one*
10. *bool_xor_two*
11. *list_compress*
12. *list_diverge*
13. *list_pairwise_swap.out*
14. *nat_anything_goes*
15. *nat_capture_subst*
16. *nat_cases_one_sol*
17. *nat_cases_ret_input_0*
18. *nat_cases_ret_input_fst*
19. *nat_cases_ret_input_snd*
20. *nat_cases_two_sol*
21. *nat_cases_use_patt*
22. *nat_id*
23. *nat_plus*
24. *nat_plus2*
25. *nat_plus2_fail*
26. *nat_plus_O*
27. *nat_plus_S*
28. *nat_plus_concrete_left*
29. *nat_plus_concrete_left*
30. *nat_plus_concrete_right*
31. *nat_plus_concrete_right*
32. *nat_plus_fail*
33. *nat_plus_fail_concrete_left*
34. *nat_plus_fail_concrete_right*
35. *nat_rec_double*
36. *nat_rec_double_fail*
37. *nat_rename*

Symbolic execution benchmarks:

1. *bool_b2_unused.ml*
2. *bool_id.ml*
3. *bool_neg.ml*
4. *bool_xor_one.ml*
5. *list_compress.ml*
6. *list_diverge.ml*
7. *nat_compare.ml*
8. *nat_max.ml*
9. *nat_max_match_call.ml*
10. *nat_minus1.ml*
11. *nat_plus.ml*
12. *nat_plus2.ml*

Their code is available in the repository linked at the end.

Appendix F: Algorithm-generated trace for equation-solving:

Trace for double for equation-solving:

```
SOLVE MODE

prog
type nat =
  | O
  | S of nat

let rec double (n:nat) : nat =
  match n with
    | O -> O
    | S n0 -> S (S (double n0))
;;

let problem (m:nat) : nat =
  double m
;;

let result : nat =
  3
;;

let double : nat -> nat |> { 0 => 0, 1 => 2 } = ?
end solve

------------------------------------------------
Constraint problem:

> Holes:
m

> Problem:
double m

> Goal: = 3

> Depth: 0
```

```
> Previous solutions: currently unsupported

> Bindings:


Extracted function application info
Function name: double, input args: m
Function arg names: n
Function # free vars: 3
Function application: sub arg m into body first

Argument # free vars: 1
Top-level: Substitute (m) for n in (match n with
  | O -> 0
  | S n0 -> S (S (double n0))) with env (TODO)
   Free vars of e: m
   Free vars of body: n double n0

Renaming free vars of body to avoid clashes with fv
    of e
   Free vars of e: m
   Free vars of body: n double n0

No clash found

No clash found

No clash found
Sub-level exp: Substitute (m) for n in (match n with
  | O -> 0
  | S n0 -> S (S (double n0))) with env (TODO)
Sub-level exp: Substitute (m) for n in (n) with env
    (TODO)
Sub-level exp: Substitute (m) for n in (0) with env
    (TODO)
Sub-level exp: Substitute (m) for n in (()) with env
     (TODO)
Sub-level pat: Substitute (m) for n in (O) with env
    (TODO)
Sub-level exp: Substitute (m) for n in (S (S (double
     n0))) with env (TODO)
Sub-level exp: Substitute (m) for n in (S (double n0
    )) with env (TODO)
Sub-level exp: Substitute (m) for n in (double n0)
    with env (TODO)

TODO: substituting in function application
Sub-level exp: Substitute (m) for n in (double) with
     env (TODO)
Sub-level exp: Substitute (m) for n in (n0) with env
     (TODO)
Sub-level pat: Substitute (m) for n in (S n0) with
    env (TODO)
Substituted body with arg m: (match m with
  | O -> 0
  | S n0 -> S (S (double n0)))
Sub'd body with all args: (match m with
  | O -> 0
  | S n0 -> S (S (double n0)))
Function application: TODO replace function body for
     patterns

=> Solving application subproblem
```

```
------------------------------------------------
Constraint problem:

> Holes:
m

> Problem:
match m with
  | O -> 0
  | S n0 -> S (S (double n0))

> Goal: = 3

> Depth: 1

> Previous solutions: currently unsupported

> Bindings:


=> Solving branch

=> Solving branch equation 1 (leaf only)

------------------------------------------------
Constraint problem:

> Holes:


> Problem:
0

> Goal: = 3

> Depth: 2

> Previous solutions: currently unsupported

> Bindings:
(m, O)

Ctor comparison failed

=> Solving branch

=> Solving branch equation 1 (leaf only)

------------------------------------------------
Constraint problem:

> Holes:
n0

> Problem:
S (S (double n0))

> Goal: = 3

> Depth: 2

> Previous solutions: currently unsupported
```

```
> Bindings:
(m, S n0)

=> Solving constructor subproblem

------------------------------------------------
Constraint problem:

> Holes:
n0

> Problem:
S (double n0)

> Goal: = 2

> Depth: 3

> Previous solutions: currently unsupported

> Bindings:
(m, S n0)

=> Solving constructor subproblem

------------------------------------------------
Constraint problem:

> Holes:
n0

> Problem:
double n0

> Goal: = 1

> Depth: 4

> Previous solutions: currently unsupported

> Bindings:
(m, S n0)

Extracted function application info
Function name: double, input args: n0
Function arg names: n
Function # free vars: 3
Function application: sub arg n0 into body first

Argument # free vars: 1
Top-level: Substitute (n0) for n in (match n with
  | O -> 0
  | S n0 -> S (S (double n0))) with env (TODO)
   Free vars of e: n0
   Free vars of body: n double n0

Renaming free vars of body to avoid clashes with fv
    of e
   Free vars of e: n0
   Free vars of body: n double n0

No clash found

No clash found
Clash found for name n0
Sub-level exp: Substitute (n0') for n0 in (match n
    with
  | O -> 0
  | S n0 -> S (S (double n0))) with env (TODO)
Sub-level exp: Substitute (n0') for n0 in (n) with
    env (TODO)
Sub-level exp: Substitute (n0') for n0 in (0) with
    env (TODO)
Sub-level exp: Substitute (n0') for n0 in (()) with
    env (TODO)
Sub-level pat: Substitute (n0') for n0 in (O) with
    env (TODO)
Sub-level exp: Substitute (n0') for n0 in (S (S (
    double n0))) with env (TODO)
Sub-level exp: Substitute (n0') for n0 in (S (double
     n0)) with env (TODO)
Sub-level exp: Substitute (n0') for n0 in (double n0
    ) with env (TODO)

TODO: substituting in function application
Sub-level exp: Substitute (n0') for n0 in (double)
    with env (TODO)
Sub-level exp: Substitute (n0') for n0 in (n0) with
    env (TODO)
Sub-level pat: Substitute (n0') for n0 in (S n0)
    with env (TODO)
Sub-level exp: Substitute (n0) for n in (match n
    with
  | O -> 0
  | S n0' -> S (S (double n0'))) with env (TODO)
Sub-level exp: Substitute (n0) for n in (n) with env
     (TODO)
Sub-level exp: Substitute (n0) for n in (0) with env
     (TODO)
Sub-level exp: Substitute (n0) for n in (()) with
    env (TODO)
Sub-level pat: Substitute (n0) for n in (O) with env
     (TODO)
Sub-level exp: Substitute (n0) for n in (S (S (
    double n0'))) with env (TODO)
Sub-level exp: Substitute (n0) for n in (S (double
    n0')) with env (TODO)
Sub-level exp: Substitute (n0) for n in (double n0')
     with env (TODO)

TODO: substituting in function application
Sub-level exp: Substitute (n0) for n in (double)
    with env (TODO)
Sub-level exp: Substitute (n0) for n in (n0') with
    env (TODO)
Sub-level pat: Substitute (n0) for n in (S n0') with
     env (TODO)
Substituted body with arg n0: (match n0 with
  | O -> 0
  | S n0' -> S (S (double n0')))
Sub'd body with all args: (match n0 with
  | O -> 0
  | S n0' -> S (S (double n0')))
Function application: TODO replace function body for
     patterns

=> Solving application subproblem
```

```
-------------------------------------------------
Constraint problem:

> Holes:
n0

> Problem:
match n0 with
  | O -> 0
  | S n0' -> S (S (double n0'))

> Goal: = 1

> Depth: 5

> Previous solutions: currently unsupported

> Bindings:
(m, S n0)

=> Solving branch

=> Solving branch equation 1 (leaf only)

-------------------------------------------------
Constraint problem:

> Holes:


> Problem:
0

> Goal: = 1

> Depth: 6

> Previous solutions: currently unsupported

> Bindings:
(n0, 0) (m, S n0)

Ctor comparison failed

=> Solving branch

=> Solving branch equation 1 (leaf only)

-------------------------------------------------
Constraint problem:

> Holes:
n0'

> Problem:
S (S (double n0'))

> Goal: = 1

> Depth: 6

> Previous solutions: currently unsupported

> Bindings:
(n0, S n0') (m, S n0)

=> Solving constructor subproblem

-------------------------------------------------
Constraint problem:

> Holes:
n0'

> Problem:
S (double n0')

> Goal: = 0

> Depth: 7

> Previous solutions: currently unsupported

> Bindings:
(n0, S n0') (m, S n0)

Ctor comparison failed

-------------------------------------------

RESULT: no solution found
```

Appendix G: Algorithm-generated trace for symbolic execution

```
TRAVERSE MODE
Function to traverse:
(let rec compare (n1:nat) (n2:nat) : cmp =
  match n1 with
    | O -> (match n2 with
             | O -> EQ
             | S m -> LT)
    | S m1 -> (match n2 with
                | O -> GT
                | S m2 -> compare m1 m2)
;;)

> Symbolic execution

-------------------------------------------------
Traverse problem:

> Function name:
compare

> Function body:
match n1 with
  | O -> (match n2 with
           | O -> EQ
           | S m -> LT)
  | S m1 -> (match n2 with
              | O -> GT
              | S m2 -> compare m1 m2)

> Input table:
```

```
*

> Args used:
(n2, false) (n1, false)

> Depth: 0

------------------------------------------------

Processing branches with recursive calls
> Processing branch with var guard n1

Branch: | O -> match n2 with
  | O -> EQ
  | S m -> LT

First processing branch leaf

> Symbolic execution

------------------------------------------------
Traverse problem:

> Function name:
compare

> Function body:
match n2 with
  | O -> EQ
  | S m -> LT

> Input table:
*

> Args used:
(n2, false) (n1, true)

> Depth: 1

------------------------------------------------

Processing branches with recursive calls
> Processing branch with var guard n2

Branch: | O -> EQ

First processing branch leaf

> Symbolic execution

------------------------------------------------
Traverse problem:

> Function name:
compare

> Function body:
EQ

> Input table:
*

> Args used:
(n2, true) (n1, true)

> Depth: 2

------------------------------------------------
> Processing branch with var guard n2

Branch: | S m -> LT

First processing branch leaf

> Symbolic execution

------------------------------------------------
Traverse problem:

> Function name:
compare

> Function body:
LT

> Input table:
*

> Args used:
(n2, true) (n1, true)

> Depth: 2

------------------------------------------------
> Processing branch with var guard n1

Branch: | S m1 -> match n2 with
  | O -> GT
  | S m2 -> compare m1 m2

First processing branch leaf

> Symbolic execution

------------------------------------------------
Traverse problem:

> Function name:
compare

> Function body:
match n2 with
  | O -> GT
  | S m2 -> compare m1 m2

> Input table:
*

> Args used:
(n2, false) (n1, true)

> Depth: 1

------------------------------------------------

Processing branches with recursive calls
> Processing branch with var guard n2
```

```
Branch: | O -> GT

First processing branch leaf

> Symbolic execution

------------------------------------------------
Traverse problem:

> Function name:
compare

> Function body:
GT

> Input table:
*

> Args used:
(n2, true) (n1, true)

> Depth: 2

------------------------------------------------
> Processing branch with var guard n2

Branch: | S m2 -> compare m1 m2

First processing branch leaf

> Symbolic execution

------------------------------------------------
Traverse problem:

> Function name:
compare

> Function body:
compare m1 m2

> Input table:
*

> Args used:
(n2, true) (n1, true)

> Depth: 2

------------------------------------------------

> Results

Input table:

* (n1, O) (n2, O)
* (n1, O) (n2, S m)
* (n1, S m1) (n2, O)
* (n1, S m1) (n2, S m2)

Args used:

(n2, true) (n1, true)
```