CPSC 548: Directed Studies Report
Program Synthesis
Academic Term: 2022W2
Name: Jifeng Wu
Student Number: 25952896
April 28, 2023

1 Introduction

Program synthesis is the task of automatically finding programs that satisfy user intent. User intent can be expressed in forms such as formal specifications [14, 28, 29, 49], input-output examples [6, 15, 17, 19, 40, 41, 42, 46], demonstrations [44], partial programs [45], and program skeletons [1]. Program synthesis has long been regarded as the pinnacle of computer science: Pnueli ranked it among the most important issues in programming theory [37].

Unfortunately, due to the variety in user intent and, particularly, the intractability of the program space, program synthesis is a famously tricky subject. To tackle this problem, new methods of program space enumeration have been developed, including stochastic techniques [35, 39], deductive top-down search [15, 38], and clever domain-specific heuristics for cutting down the program space [23, 34]. This enables modern program synthesis techniques to produce sizable code snippets for many restricted scenarios: number and date transformations [17, 41, 42], constructing graphical objects [8, 9, 16, 18, 20], code repair [12, 22, 24, 33, 43], and code superoptimization [4, 5, 23, 30, 36, 39]).

In this directed study, I aim to acquire comprehensive literacy in program synthesis and develop a preliminary understanding of its problems and future research directions. Specifically, I will study relevant concepts underlying program synthesis and understand the operating principles of several classic program synthesis algorithms before implementing them and evaluating them over a set of benchmarks. Through such an evaluation, I can not only gain a detailed comprehension of the workings of the implemented algorithms and assess their pros and cons through comparison but also derive high-level insights concerning program synthesis algorithms in general.

The rest of this report is organized as follows. Section 2 explains the concepts common to all program synthesis algorithms. Section 3 introduces each classical program synthesis algorithm we studied. Section 4 covers the implementation and comparison work that we have done, discussing implementation details, our selected benchmarks, our evaluation procedure, the results of our evaluation, and insights gained in the process. Finally, Section 5 concludes.

2 Prelimiary Concepts

2.1 Syntactic Bias

To achieve program synthesis, one typically has to search through many possible programs. This is a challenging combinatorial problem as the number of programs that can be generated grows exponentially with the program's size. Therefore, restricting the space of possible programs, or introducing syntactic bias, is crucial for scaling up program synthesis.

One popular approach, Syntax-Guided Synthesis [1], limits the set of possible programs by supplementing the logical specification with a user-provided syntactic template in a context-free grammar, with a synthesized program being a valid derivation of such a grammar.

For example, to synthesize a `max2` function that computes the maximum value of two given values, `x` and `y`, we can define a context-free grammar that incorporates linear expressions with additions and subtractions over the terminal values `x`, `y`, `0`, and `1` and if-then-else expressions, with a derivation of $Start$ being a synthesized program:

```
Start ::= x
        | y
        | 0
        | 1
        | Start + Start
        | Start - Start
        | ite(StartBool, Start, Start)
StartBool :: StartBool and StartBool
            | StartBool or StartBool
            | not StartBool
            | Start <= Start
            | Start == Start
            | Start >= Start
```

Under this context-free grammar, a synthesized program that satisfies the specification is $ite(x >= y, x, y)$.

Being able to model many domain-specific languages and even general-purpose programming languages, context-free grammars are some of the most common syntactic biases used by program synthesis algorithms. However, there are also widely used syntactic biases that enforce restrictions on arbitrary context-free grammar.

For example, programs that contain if-then-else expressions or other expressions that select a value based on a boolean expression (such as the `max2` example presented above) can effectively be seen as decision trees whose leaf nodes (nodes that directly return a value) and internal nodes (nodes that evaluates a boolean expression to select a value to return from its children) are derivations of at least two non-terminals within a CFG, a term non-terminal and a separate predicate non-terminal. This is the syntactic bias employed by the winner of the Syntax-Guided Synthesis Competition 2016, EUsolver (enumerative-unification solver) [2, 3], which enumerates terms from the term non-terminal and synthesizes larger expressions by generating predicates from the predicate non-terminal and learning a decision tree.

On the other hand, there are syntactic biases where there is only one non-terminal in the corresponding context-free grammar. One of them is components, which allow terminal values and domain-specific functions that can accept terminal values or values returned by any other function as arguments. Thus, there are no restrictions on their composition. This can be viewed as context-free grammar with only one non-terminal. Such a syntactic bias was proposed in research on component-based program synthesis [21], while it is also widely used in generic programming-based program synthesis algorithms [25, 27], as this ensures that programs remained well-formed after potentially arbitrary mutations and crossover during the evolution of the programs.

Moreover, another syntactic bias that can be seen as a subset of context-free grammars is reversible state-modifying instructions, which involves synthesizing a series of instructions that modify an explicitly defined program state (such as the values of a finite set of registers), with each instruction chosen from a pre-defined set of instructions and being reversible (given a post-state, there is a tractable number of possible pre-states). This is the syntactic bias employed in bidirectional enumerative search, a technique applied to synthesizing geometry construction programs [16] and super-optimizing assembly code [36].

Furthermore, inductive logic programming [10] is a niche form of program synthesis, which generates logical programs (rules) given specific observations (examples). A logical program consists of a set of universally-quantified Horn clauses in implication form resembling $\forall A.lego\_builder(A) \wedge enjoys\_lego(A) \rightarrow happy(A)$ with the predicates that may appear in each side of the implications forming disjoint sets and the logical variables are chosen from a predefined set. Given such constraints, logical programs can also be seen as a derivation of context-free grammar.

In summary, the relationships among these syntactic biases can be represented using the Euler diagram in Figure 1, with CFG, CFGWSPN, COMP, RSMI, and LP representing context-free grammar, context-free grammar with separate predicate non-terminal, components, reversible state-modifying instructions, and logical programs, respectively.

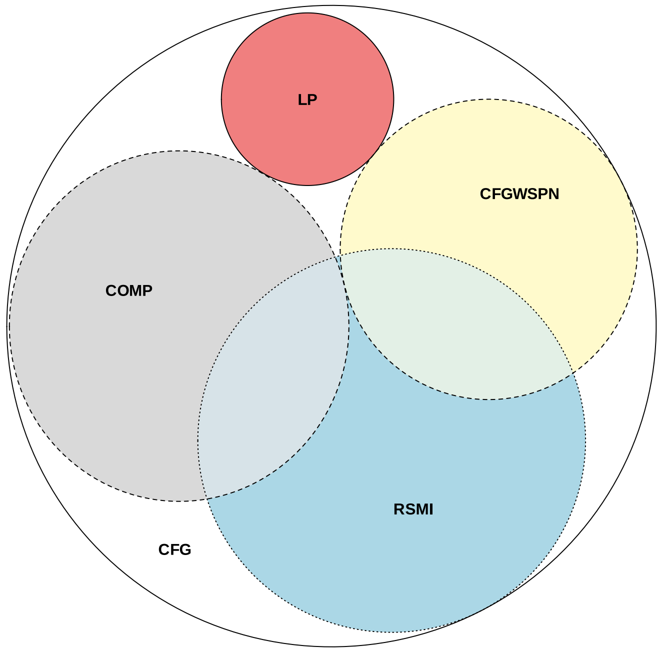

Figure 1: Relationships Among the Syntactic Biases

2.2 Background Theories and Verifying a Candidate Program

In program synthesis, both the constraints and the candidate programs are frequently formulas in first-order logic with variables and constants of different types, including booleans, integers, rational numbers, bit vectors, and arrays, as well as operations on them. The formula is said to be under a specific background theory depending on the types of constants and variables within the formula and the operations on them. Some of the most commonly encountered background theories in program synthesis include Linear Integer Arithmetic, the theory of bit vectors, and the theory of arrays.

2.2.1 Linear Integer Arithmetic

Linear Integer Arithmetic is one of the simplest and most widely used background theories. It allows boolean and integer variables, boolean, integer, and rational constants, the arithmetic operators $+, \times$, the logical operators $\wedge, \vee, \neg$, and the relational operators $=, \leq$ in the following recursively defined formulae [7]:

- A boolean variable or constant is a Linear Integer Arithmetic formula.

- $\sum_i a_i x_i \bowtie k$, where $\bowtie \in \{=, \leq\}$, x_i is an integer variable, and a_i and k are integer or rational constants, is a Linear Integer Arithmetic formula.
- If α and β are Linear Integer Arithmetic formulae, so are $\alpha \wedge \beta$, $\alpha \vee \beta$, $\neg\alpha$, and $\neg\beta$.

For example, $(p_1 \vee (x_1 + 2x_2 \leq 2)) \wedge (p_2 \vee (3x_3 + 4x_4 + 5x_5 = 2) \vee (-x_2 - x_3 \leq 3)$, where p_1, p_2 are boolean variables, and x_1, x_2, x_3, x_4, x_5 are integer variables, is a Linear Integer Arithmetic formula.

Linear Integer Arithmetic is used widely in software verification and automated reasoning as many programs utilize integer variables and perform arithmetic operations on them [31].

2.2.2 The Theory of Bit Vectors

Another standard theory is the theory of bit vectors, which is similar to Linear Integer Arithmetic, but with the following differences[1]:

- Variables and constants must be either booleans or bit vectors (unsigned or signed two-complement integers of a fixed length used by CPUs and mainstream programming languages).
- Aside from arithmetic operators, bitwise operators on bit vectors such as bitwise and, bitwise or, bitwise not, left shift, and right shift are also supported.
- Comparison operators are divided into signed comparison operators and unsigned comparison operators.

For example, the formula $ULE(x\&y, x >> 0x0000001F) \wedge (x\&(x - 0x00000001) \neq 0x0000001F)$, where x and y are 32-bit integer variables, $0x0000001F$ and $0x00000001$ are 32-bit integer constants, ULE is the unsigned less than or equal to comparison operator, $\&$ is the bitwise and operator, and $>>$ is the right shift operator, is a formula under the theory of bit vectors.

The theory of bit vectors allows modeling the precise semantics of unsigned and signed two-complements arithmetic, which is helpful in modeling and verifying hardware designs and "bit twiddling hacks" that can be used as alternatives to potentially more expensive operations[2].

2.2.3 The Theory of Arrays

The theory of arrays extends Linear Integer Arithmetic by introducing the array type, which supports the following two operators:

- $read(a, i)$, which reads the value from the integer index i of the array a.
- $write(a, i, v)$, which writes the value v to the integer index i of the array a.

The theory of arrays is commonly used in modeling and verifying computer programs that use arrays and is also fundamental to modeling higher-level data types, such as sets and finite maps [11].

[1]https://microsoft.github.io/z3guide/docs/theories/Bitvectors/
[2]https://graphics.stanford.edu/ seander/bithacks.html

2.3 The Satisfiability Modulo Theory Problem, SMT Solvers, and Verifying a Candidate Program

Given a background theory and a formula under the background theory, the problem of determining whether the formula is satisfiable, and if so, finding possible values to assign to each variable, is known as the satisfiability modulo theory (SMT) problem.

SMT solvers, such as z3[3] and cvc5[4], can solve this problem for a practical subset of background theories and have been used as a building block for a wide range of applications across computer science, including software testing and verification, proving the correctness of programs, and program synthesis.

SMT solvers play a decisive role in program synthesis by allowing us to verify a candidate program against the constraints. Specifically, suppose the constraints and candidate programs are under a background theory supported by an SMT solver. In that case, we can use the SMT solver to verify a candidate program by plugging it into the constraints and checking whether the negation of the resulting formula is satisfiable. If it is satisfiable, there exist inputs for which the candidate program would produce an output inconsistent with the specification, and the candidate program is rejected. Otherwise, the candidate program is accepted.

However, an SMT solver may be slow or ineffective on some formulas. As an alternative, it is also possible to verify a candidate program by randomly sampling inputs from a predefined input space, and evaluating, on each sampled input, the expression that results from the negation of the candidate program plugged into the constraints. If no input is found such that the expression evaluates to true, we can consider the candidate program accepted. Otherwise, we reject the candidate program.

2.4 Counterexample Guided Inductive Synthesis

As mentioned in 2.2, we verify a candidate program using a verification oracle that internally uses an SMT solver or randomly samples inputs from a predefined input space. However, calling a verification oracle could be expensive, which is aggravated by the fact that many candidate programs are being searched. To mediate this problem, Armando Solar-Lezama proposed a technique known as Counterexample Guided Inductive Synthesis [45], in which each time we verify a candidate program, we not only check whether the negation of plugging the candidate program into the constraints is satisfiable or not but also ask the verification oracle to find possible values to assign to each variable if it is satisfiable. This is possible regardless of whether the verification oracle uses an SMT solver or randomly samples inputs from a predefined input space.

These values form what is known as a counterexample, or inputs on which at least some candidate programs violate the specifications. By storing counterexamples derived from the verification oracle within the program synthesis algorithm, we can check that a candidate program does not violate the specifications on all counterexamples at hand before sending the candidate program to the verification oracle. Candidate programs that fail on at least one counterexample are pruned. This reduces the number of times we call the verification oracle and boosts program synthesis efficiency.

In the aforementioned `max2` example, we can specify the constraints as $\forall x, y \in \mathbb{Z}. max2(x, y) \geq x \wedge max2(x, y) \geq y \wedge (max2(x, y) = x \vee max2(x, y) = y)$. Suppose we have synthesized the candidate program x, which plugged into the formula and negated results in the formula $\exists x, y \in \mathbb{Z}. x < x \vee x < y \vee x \neq x \wedge x = y$, for which a verification oracle proves is satisfiable and provides the counterexample

[3] https://microsoft.github.io/z3guide/

[4] https://cvc5.github.io/

$x = 0, y = 1$. With this counterexample, the program synthesis algorithm can reject the candidate program $2x$, as $2x$ plugged into the constraints on $x = 0, y = 1$ results in $0 \geq 1 \wedge 0 \geq 1 \wedge (0 = 0 \vee 0 = 1)$, which is evaluated to false.

3 Introduction to Each Studied Program Synthesis Technique

We have studied a number of classic program synthesis techniques. In this section, we will introduce each technique according to its syntactic bias.

3.1 Program Synthesis Techniques With a Context-free Grammar Syntactic Bias

3.1.1 Top-down Tree Search

Given context-free grammar, one of the simplest program synthesis algorithms is top-down tree search, which searches for candidate programs by finding derivations in the grammar in a top-down manner. Specifically, it maintains a priority queue of partial derivations to visit P and a set of visited partial derivations P_v (both initialized to the start symbol). In each iteration of the algorithm, a partial derivation is dequeued. If it contains no non-terminals, it is a valid candidate program, and it is sent for verification. Elsewise, each non-terminal in the partial derivation is expanded using each production rule for that non-terminal, resulting in a potentially large number of new partial derivations. For each such partial derivation, if it does not correspond to a visited partial derivation, it is enqueued, and the set of visited partial derivations is updated.

In the `max2` example, we have used the following context-free grammar as the syntactic bias:

```
Start ::= x
        | y
        | 0
        | 1
        | Start + Start
        | Start - Start
        | ite(StartBool, Start, Start)
StartBool :: StartBool and StartBool
           | StartBool or StartBool
           | not StartBool
           | Start <= Start
           | Start == Start
           | Start >= Start
```

Under the top-down tree search algorithm, P and P_v are initialized to $\{Start\}$. In the first iteration of the algorithm, $Start$ is dequeued from P, and containing non-terminals, it is expanded in every possible way, resulting in $x, y, 0, 1, Start + Start, Start - Start, ite(StartBool, Start, Start)$, which are enqueued into P and used to update P_v. In the following iterations, $x, y, 0, 1$ are dequeued and are sent to verification (as they contain no non-terminals), while each non-terminal in $Start + Start, Start - Start, ite(StartBool, Start, Start)$ is expanded (resulting in $x + Start, y + Start, 0 + Start, 1 + Start, Start + Start + Start$ etc.), with the new partial derivations further updating P and P_v.

3.1.2 Bottom-up Tree Search (Enumerative Learning)

Although conceptually simple, top-down tree search is an inefficient program synthesis technique as a lot of repeated statements are enumerated in the process, even when a set of visited partial derivations P_v is employed. For example, in the example presented above, starting from $x + S + S$, $x + S + S \Rightarrow$

$x+y+S \Rightarrow x+y+x$ and $x+S+S \Rightarrow x+x+S \Rightarrow x+x+y$ all result in the same candidate program, with $y+x$ and $x+y$ being equivalent derivations of $S+S$, yet there is no overlap between the two partial derivations in the process of expanding the non-terminals before the candidate programs are yielded.

To overcome this problem, instead of gradually expanding non-terminals in a partial derivation, we can directly generate all possible derivations of the non-terminals of a given size by using their production rules and their known possible derivations of a smaller given size in what is known as the bottom-up tree search algorithm or the enumerative learning algorithm [48]. For example, in the example above, we already know that the non-terminal $Start$ has four derivations of maximum size 1: $x, y, 0, 1$. With this, it is possible to derive all derivations of $Start$ with maximum size 2 through the $Start \rightarrow Start + Start$ and $Start \rightarrow Start - Start$ production rules: $x+y, x+0, x+1, y+x, y+0, y+1, 0+x, 0+y, 0+0, 0+1, 1+x, 1+y, 1+0, 1+1, x-y, x-0, x-1, y-x, y-0, y-1, 0-x, 0-y, 0-0, 0-1, 1-x, 1-y, 1-0, 1-1$, which, after simplification and eliminating seen derivations, results in $x+y, x+1, y+1, 2, x-y, x-1, y-x, y-1, -x, -y, -1, 1-x, 1-y$. Similarly, $true, false, x \leq y, x \leq 0, x \leq 1, y \leq x, y \leq 0, y \leq 1, 0 \leq x, 0 \leq y, 1 \leq x, 1 \leq y, x = y, x = 0, x = 1, y = 0, y = 1$ can be generated fro the $StartBool$ non-terminal. Such a set of derivations with a maximum size of 2 will be further used when generating derivations of $Start$ and $StartBool$ with larger maximum sizes. This significantly reduces the number of expressions considered in searching for candidate programs.

3.1.3 Uniform Random Sampling

An alternative to enumerating derivations in a context-free grammar and checking each derivation in the order that they were enumerated is to generate all derivations of the context-free grammar of a given size, iterate over them in a random order, and verify each randomly sampled derivation.

We can use two recursive functions to generate all derivations of the context-free grammar of a given size. The first function returns all derivations of a given size generated by a token (a terminal or a non-terminal in context-free grammar). In contrast, the second function returns all derivations of a given size generated by a production rule (a sequence of tokens in context-free grammar).

In the first function, if the token is a terminal, it is returned. If the token is a non-terminal, we return all possible derivations generated by each of its production rules.

As for the second function, we split a production rule into its head (first token) and its tail (the remaining tokens, which could be considered another production rule) and split our specified size between the head and the tail. If we were to generate derivations of size s production rule, and the head generates derivations of size $s'(s' < s)$, then the tail should generate derivations of size $s - s'$. For each such split, we recursively return all derivations generated by the head and the tail and concatenate them to obtain all derivations generated by the production rule.

For example, in the `max2` example, to generate all derivations of $Start$ of size three, we try to generate all derivations of the production rules $x, y, 0, 1, Start+Start, Start-Start, ite(StartBool, Start, Start)$ of size 3. No derivations of size three can be generated for $x, y, 0, 1, ite(StartBool, Start, Start)$, while for $Start + Start$, we attempt to generate derivations of the first $Start$ of size one and derivations of the remaining $+Start$ of size two, as well as derivations of the first $Start$ of size two and derivations of the remaining $+Start$ of size one. No derivations of the first $Start$ of size two can be generated, while derivations of the first $Start$ of size one result in $x, y, 0, 1$. As for derivations of $+Start$ of size two, we generate derivations of the first + of size one and derivations of the remaining $Start$ of size one. The former results in + itself, while the latter results in $x, y, 0, 1$ again.

As we can see, these two functions will be repeatedly invoked on the same inputs (token or production rule and a given size). Thus, calls to them should be memoized. Memoization also allows for the efficient sampling of derivations generated by a token of size $s+1, s+2, \dots$ after sampling those of size s, as the previously sampled derivations and the intermediate results in their calculation can effectively be reused.

An issue with this approach is that it may be problematic when there is left-recursion in the context-free grammar. However, this can be mediated by setting a maximum limit to the number of useful recursions of $m(1+s)$, where m is the number of non-terminals in the context-free grammar, and s is our desired size, as proposed by Frost et. al. [13]

From an implementation perspective, uniformly sampling context-free grammar derivations resembles bottom-up tree search as both involve generating and caching derivations of context-free grammar. However, they are conceptually very different, as bottom-up tree search enumerates and verifies all derivations of the context-free grammar in order, pruning any redundant derivation in the process, while the sampling technique only samples derivations of a context-free grammar of a specific given size at a time. During the sampling process, we make no effort to remove redundant derivations (as this leads to a non-uniform sampling).

3.1.4 Metropolis-Hastings Sampling

Another sampling-based technique [1] adapts the Metropolis-Hastings algorithm originally used for sampling from high-dimensional probability distributions to sampling context-free grammar derivations.

The Metropolis-Hastings algorithm represents an input space using an undirected graph in its original form, as depicted in Figure 2. Each node in the graph represents a particular input, and an edge connecting two nodes represents that the corresponding inputs are "adjacent" in some sense. To sample from the high-dimensional probability distribution, the Metropolis-Hastings algorithm does a random walk on the undirected graph starting from a randomly chosen node, with the random walk sequence being the sampled sequence. Each iteration calculates the probability density function f on the current node j and a randomly selected neighbor k. It then calculates an acceptance ratio $P(j,k) = min(1, \frac{f(k)}{f(j)})$, and walks from j to k with probability equal to $P(j,k)$. Otherwise, the next node in the random walk is still j itself.

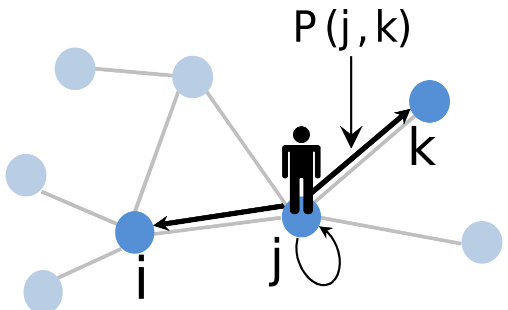

Figure 2: The Metropolis-Hastings Algorithm

To adapt the Metropolis-Hastings algorithm to sampling expressions, several tricks are used.

First, each node represents a derivation of the context-free grammar of a given size. Its neighbors are all derivations that have replaced a subexpression rooted at a node with an equally-sized one corresponding to the same non-terminal, as depicted in Figure 3.

Second, given a node e, its probability density function f is defined as $f(e) = e^{-\beta C(e)}$, in which $\beta = 0.5$ and $C(e)$ is the number of examples on which e is incorrect when plugged into the specifications.

Moreover, the Metropolis-Hastings algorithm is run on derivations of the context-free grammar with size

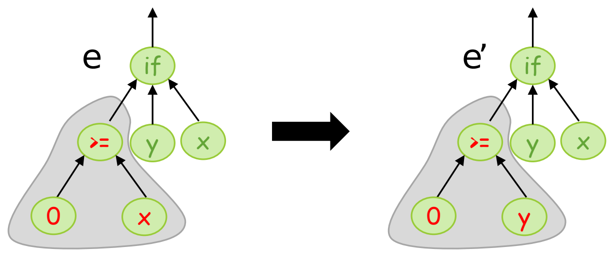

Figure 3: Replacing a Subexpression Rooted at a Node With an Equally-sized One Corresponding to the Same Non-terminal

k before being run on derivations of the context-free grammar with size $k + 1$. Thus, the Metropolis-Hastings algorithm can be implemented by intertwining using the memoized uniform random sampling algorithm to efficiently generate derivations of increasing size and sampling on derivations of a given size.

3.1.5 Stochastic Sampling From a Context-Free Grammar Using Features and Clues

Menon et. al. [32] proposed an approach targeting string transformation problems in Microsoft Excel, which are domain-specific programming-by-example (i.e., the specifications are input-output pairs) problems where a wealth of training data (of input-output pairs as well as correct programs for them) is available a priori (i.e., independent of the specifications at hand).

In this approach, a set of predefined predicates that can be evaluated on input-output examples, known as features, and a set of predefined functions which select grammar rules based on features that are evaluated to be true, known as clues, need to be manually specified. Moreover, each clue has a weight associated with it. The weights of the clues are trained on the training data in advance, with the input-output pairs as the inputs and the correct programs for them as the ground truth outputs.

After the training process, given an input-output pair, it is possible to evaluate the predicates on the input-output pair and evaluate the clues on the predicates before constructing a probabilistic context-free grammar with each production rule augmented with weights based on the weights of the relevant clues that suggested that rule. Afterward, such a probabilistic context-free grammar is used to guide a biased sampling to generate derivations that are more likely valid programs for the given input-output pair. This process is depicted in Figure 4.

Although such a technique appears promising, it requires a large amount of pre-collected training data and domain expertise spent designing the features and the clues, both limiting its applicability to certain domain-specific tasks.

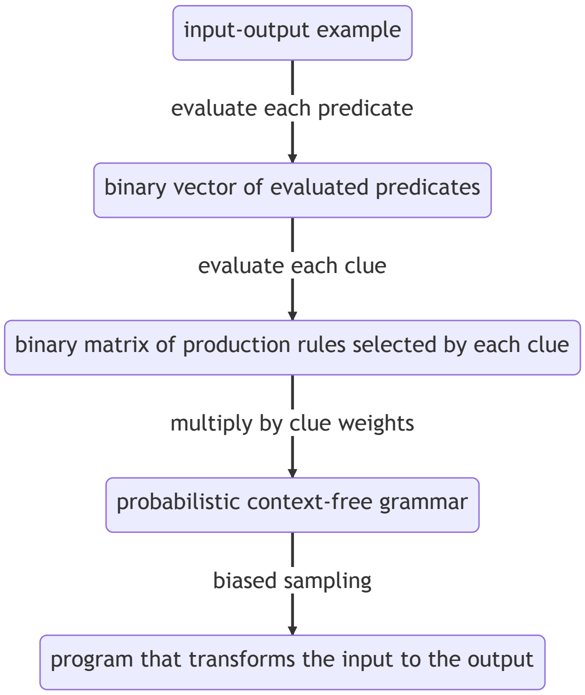
input-output example
evaluate each predicate
binary vector of evaluated predicates
evaluate each clue
binary matrix of production rules selected by each clue
multiply by clue weights
probabilistic context-free grammar
biased sampling
program that transforms the input to the output

Figure 4: Stochastic Sampling From a Context-Free Grammar Using Features and Clues

3.2 Program Synthesis Techniques With a Context-free Grammar With Separate Predicate Non-terminal Syntactic Bias

3.2.1 EUsolver

EUsolver (enumerative-unification solver) [3] is an approach that specifically targets synthesizing programs that contain expressions that select an expression based on a boolean expression, such as the solution to the `max2` problem, $ite(x >= y, x, y)$. In these programs, there are at least two non-terminal nodes: one for deriving expressions selected by boolean expressions such as x and y, known as a term non-terminal, and one for deriving boolean expressions such as $x \geq y$, known as a predicate non-terminal.

EUsolver uses the bottom-up tree search (enumerative learning) algorithm on the term non-terminal and the predicate non-terminal to generate expressions selected by boolean expressions and boolean expressions themselves. Specifically, it first generates a set of expressions that cover all inputs (i.e., for each input, there is at least one expression, when inserted into the constraints and evaluated on the input, evaluates to true). Afterward, it generates a set of predicates and uses the expressions and predicates to construct a decision tree.

The decision tree construction process first searches for expressions that could cover all inputs. If such expressions are found, they are directly returned as candidate programs. Elsewise, we find a predicate that can best split the inputs. The left and right subtrees are constructed on the inputs for which the predicate evaluates to true and false, respectively, using the remaining predicates. Afterward, we can build decision trees using the predicate that can best split the inputs and the left and right subtrees. If, for any reason, a failure is encountered (e.g., there are no expressions that can cover all inputs or no predicates remaining when constructing the left or right subtrees), more expressions or more predicates are enumerated accordingly before the decision tree construction process starts again.

In the decision tree construction process, the predicate that can best split the inputs is found by calculating the Information Gain Heuristic. This is done in the following procedure:

Given a set of inputs $X = \{x_1, \dots, x_m\}$ and a set of expressions $E = \{e_1, \dots, e_n\}$, we first store the *relative probabilities* of each input covered by each expression in a two-dimensional array. The element at row i column j should correspond to the relative probability of x_i being covered by e_j, and each row should sum up to one, i.e., the sum of the relative probabilities of x_i being covered by $e_1, \dots, e_n$ is 1.

In the original EUsolver paper, if *multiple expressions* $e_\alpha, e_\beta, \dots$ cover an input x_i, the relative probabilities of that input x covered by an expression e_α is proportional to the total number of inputs that expression covers within the set of inputs X, i.e., proportional to the size of the set $\{x | x \in X, e_\alpha \text{ covers } x\}$. For instance, given three inputs $X = \{x_1, x_2, x_3\}$ and three expressions $E = \{e_1, e_2, e_3\}$, if e_1 covers x_1, x_2, e_2 covers x_2, x_3, e_3 covers x_1, x_3, then the *relative probabilities* of x_1 being covered by e_1, e_2, e_3 are $\frac{2}{4}, \frac{0}{4}, \frac{2}{4}$, or $\frac{3}{7}, \frac{1}{7}, \frac{3}{7}$ with *add-one smoothing*. Similarly, the *relative probabilities* of x_2 being covered by e_1, e_2, e_3 with *add-one smoothing* are $\frac{3}{7}, \frac{3}{7}, \frac{1}{7}$, and the *relative probabilities* of x_3 being covered by e_1, e_2, e_3 with *add-one smoothing* are $\frac{1}{7}, \frac{3}{7}, \frac{3}{7}$.

Next, we average the rows of this two-dimensional array to obtain the *relative probabilities* of *any input* covered by each expression. The jth element should be the relative probability of any input being covered by the jth expression, and all elements should sum to 1. In the example above, the result of averaging the rows is $\frac{1}{3}((\frac{3}{7}, \frac{1}{7}, \frac{3}{7}) + (\frac{3}{7}, \frac{3}{7}, \frac{1}{7}) + (\frac{1}{7}, \frac{3}{7}, \frac{3}{7})) = (\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$, which suggests that an arbitrarily selected input $x \in X$ has an equal chance of being covered by each of e_1, e_2, e_3.

Such an array can also be interpreted as the probabilities of *the entire set of inputs X* associated with each expression. Thus, it is possible to use such probabilities to calculate the *entropy H* of the set of inputs, which is $-(\frac{1}{3} \log \frac{1}{3} + \frac{1}{3} \log \frac{1}{3} + \frac{1}{3} \log \frac{1}{3}) = 1.10$ in the example above. In this case, a high entropy corresponds to the set of inputs *potentially requiring many expressions within E to fully cover it*, while a low entropy means the opposite.

In the decision tree construction process, to decide the predicate that can best split the input set X, for each predicate p, we calculate the entropies $H(X_1), H(X_2)$ of the two sets of inputs for which the predicate evaluates to true and false, X_1 and X_2. Then, we calculate the information gain of the predicate, $G(p) = \frac{|X_1|}{|X|} H(X_1) + \frac{|X_2|}{|X|} H(X_2)$. This can be viewed as *a weighted average of the entropies of the two sets*. In this sense, as we want the decision tree to be *as simple as possible*, we select the predicate with the *lowest* information gain, such that recursively constructing the left and right subtrees tend to both lead to *finding a term that can cover all inputs* instead of further recursively constructing decision trees with subtrees.

3.3 Program Synthesis Techniques With a Components Syntactic Bias

3.3.1 Genetic Programming

The genetic programming algorithm is an adaptation of genetic algorithms to program synthesis. In this adaptation, we start with a population consisting of randomly generated programs and use the crossover, mutation, and reproduction operations in each iteration to generate new programs and duplicate existing ones.

To accomplish this goal, the genetic programming algorithm requires specifying the size of the randomly generated population of programs, the probabilities of doing each of crossover, mutation, and reproduction in an iteration, and a function that quantizes the quality of a program (e.g., the number of inputs on which the program is incorrect when plugged into the specifications).

Given these hyperparameters, the crossover operation randomly selects two programs from the population based on their fitness scores before randomly selecting a node independently from each program and swapping the subexpressions rooted at the nodes, as depicted in Figure 5. The mutation operator randomly selects a program based on its fitness score and randomly selects a node from the program. It then replaces the subexpression rooted at the node with a new, randomly generated expression, as depicted in Figure 6. Finally, the mutation operator randomly selects a program based on the fitness score and copies it to the new population.

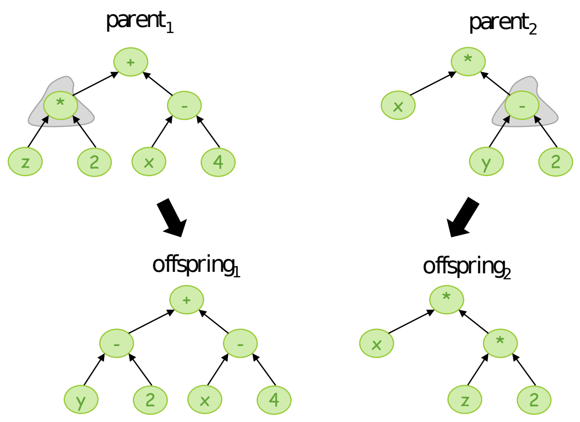

Figure 5: The Crossover Operation

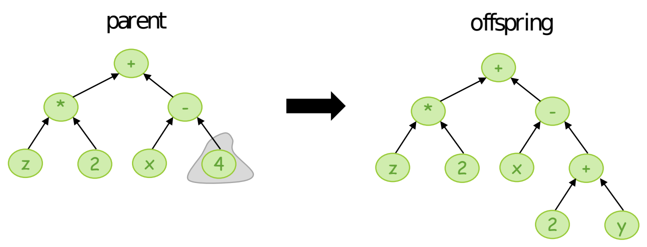

Figure 6: The Mutation Operation

Since random nodes are selected and arbitrarily modified in the crossover and mutation operations, the generic programming algorithm should use the components syntactic bias, in which any two subexpressions are freely interchangeable without breaking the well-formedness of a program.

3.4 Program Synthesis Techniques With a Reversible State-modifying Instructions Syntactic Bias

3.4.1 Bidirectional Enumerative Search

Bidirectional enumerative search is a technique that synthesizes sequences of instructions that modify an explicit program state, such as the values of a finite set of registers for assembly code or the geometric objects currently constructed for geometry construction programs, given input-output pairs in the form of prestate-poststate pairs.

Specifically, given a prestate-poststate pair, bidirectional enumerative search considers program states to be nodes within an implicit, unexplored, directed graph, with the edges being instructions. The algorithm does forward and backward breadth-first search simultaneously from the nodes corresponding to the prestate and the poststate. Forward breadth-first search explores all possible successors of a node by exploring all its out-edges (valid instructions that can be executed on the node) and executing them. Conversely, backward breadth-first search explores all possible predecessors of a node by exploring all its in-edges (valid instructions that could have been executed to generate the current node) and executing their reverses.

During this process, all edges explored are recorded. This process continues until there is an intersection between the nodes visited by the forward breadth-first search and those visited by the backward breadth-first search. In this case, a path from the prestate to the poststate, corresponding to a valid instruction sequence that can be executed to transform the prestate into the poststate, is found, as depicted in Figure 7.

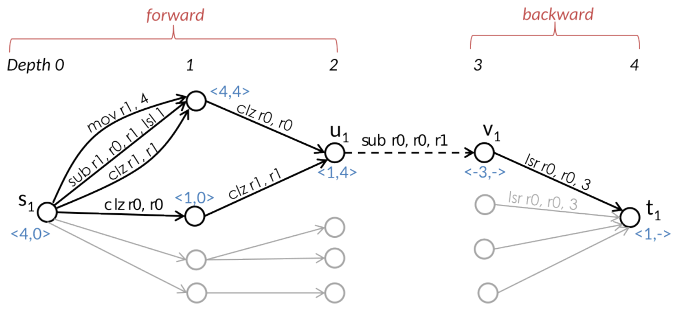

Figure 7: A Path From the Prestate to the Poststate Is Found

However, the path found for one prestate-poststate pair may not apply to all prestate-poststate pairs. Even though the underlying implicit directed graph may be different for the next prestate-poststate pair, the valid sequences of edges that were explored for the previous prestate-poststate pair can be saved, checked, and used as starting points for finding a path for the next prestate-poststate pair (some sequences of edges could still be valid for the next prestate-poststate pair, while some have become invalid). It is worth noticing that we do not have to explore sequences of edges that were not explored for the previous prestate-poststate pair. This is because those sequences, by not being explored for the previous prestate-poststate pair, cannot be expanded to valid paths for the previous prestate-poststate pair. Even if they could be expanded to valid paths for the next prestate-poststate pair, they cannot be expanded to valid paths for all input-output pairs, and thus it is safe not to explore them. This is how bidirectional enumerative search safely prunes the search space.

4 Experimental Evaluation

Of the algorithms we introduced, we have implemented the top-down tree search, bottom-up tree search, uniformly sampling context-free grammar derivations, and Metropolis-Hastings algorithms targeting a context-free grammar syntactic bias and the EUsolver algorithm targeting a context-free grammar with separate predicate non-terminal syntactic bias, and evaluate them on a set of benchmarks. We haven't implemented stochastic sampling from a context-free grammar using features and clues, genetic programming, or the bidirectional enumerative search algorithm. This is because these algorithms require a separate training dataset, domain-specific expertise, tuning a large hyperparameter search space, and providing a wealth of input-output examples impractical on our evaluation benchmarks.

4.1 Implementation Details

We have implemented our program synthesis algorithms and verification oracle using Python and the `z3-solver` package. Specifically,

- We use `z3` variables to represent non-terminals and terminals in context-free grammar and `z3` expressions to represent candidate programs and partial derivations. This allows tight integration with the verification oracle, implemented using `z3` package. However, `z3` expressions are *immutable*. At the same time, we frequently need to mutate candidate programs and partial derivations in both the program synthesis algorithms and the verification oracle for tasks such as generating derivations, plugging a candidate program into the constraints, and checking the validity of the negated results or evaluating it on a counterexample. Thus, we implement a collection of functions to convert `z3` expressions to and from tuples of `z3` variables and operators that encode the originally written production rule in Reverse Polish Notation. For example, the expression $ite(x >= y, x, y)$ is represented as $(x, y, x, y, >=, ite)$. This makes it easy for us to iterate over symbols within and arbitrarily manipulate them, which is impossible with the immutable `z3` expressions. For the same reasons, we also record production rules of context-free grammars in this format, e.g., the production rule $Start + Start$ in the grammar for `max2` is represented as $(Start, Start, +)$.

- We implement each program synthesis algorithm as Python coroutines[5], which yield candidate programs that pass checks on available counterexamples for verification and accept new counterexamples. Such an implementation allows us to iteratively generate candidate programs on demand and connect the program synthesis algorithm with the verification oracle with loose coupling, all under clean and succinct code.

- The verification oracle is also a coroutine that accepts candidate programs and yields counterexamples (if one could be found) or `None`.

All code, benchmarks, and replication instructions are available on GitHub[6].

4.2 Benchmarks

We assess the implemented program synthesis techniques on a subset of benchmarks originally curated for the 1st Syntax-Guided Synthesis Competition at CAV 2014[7]. The benchmarks are grouped into integer benchmarks, boolean and bit-vector problems, and hacker's delight problems.

[5]https://book.pythontips.com/en/latest/coroutines.html

[6]https://github.com/abbaswu/implementation-of-syntax-guided-program-synthesis-and-verification-techniques

[7]https://github.com/rishabhs/sygus-comp14

4.2.1 Integer Benchmarks

These benchmarks under the Linear Integer Arithmetic background theory and context-free grammar with separate predicate non-terminal syntactic bias provide a rough representation of synthesis problems that deal with programs having branching structures. Among these are the `array-search` benchmarks, which aim to synthesize a program that locates the index of an element within 2 to 16 sorted parameters without using loops. Also present are the `max` benchmarks, which are similar but find the maximal element among n parameters.

4.2.2 Boolean and Bit-vector Problems

These benchmarks are all under the Bit Vector background theory. The `parity` benchmarks, with context-free grammar or component syntactic biases, synthesize parts of parity-checking circuits. With component syntactic biases, the `zmorton` benchmarks synthesize the matrix indexing function for the ZMORTON layout.

4.2.3 Hacker's Delight Problems

These benchmarks are all under the Bit Vector background theory and have component syntactic biases. They originate from 20 bit-manipulation problems from the book Hacker's Delight [50], which aims to find efficient ways to implement bit vector transformations.

Since all algorithms struggle on the boolean and bit-vector problems and all algorithms except for EUsolver fail on the `array-search` and `max` benchmarks for $n \geq 2$ within our time limit of 900 seconds, we have omitted them from our evaluation.

4.3 Experimental Procedure

Our implemented algorithms span the context-free grammar and the context-free grammar with separate predicate non-terminal hypothesis spaces. Given that the context-free grammar with separate predicate non-terminal hypothesis space is a subset of the context-free grammar hypothesis space, we first compare all algorithms except EUsolver on the integer and hacker's delight benchmarks, before comparing all algorithms, including EUsolver on the integer benchmarks.

We ran the algorithms on the benchmarks using GNU Parallel [47] on a Linux machine with 20 CPU cores and 64GB of RAM with a time limit of 900 seconds, or 15 minutes. Since the outcomes of randomized algorithms such as uniform random sampling and Metropolis-Hastings sampling vary from run to run, and to account for other fluctuations, we evaluate each algorithm ten times over each benchmark.

We first present the percentage of runs of each algorithm on each benchmark that can synthesize a correct program within our time limit, before presenting a more detailed description comparing the running times of different solvers on the constituent benchmark problems.

Moreover, we present the running times using a logarithmic scale to mitigate constant factors arising from the implementations' preliminary nature. Specifically, we categorized the solution times into various buckets, namely 1, 3, 10, 30, 100, 300, and 900 seconds, such that the initial buckets correspond to running times under 1 second, the second buckets for running times between 1 to 3 seconds, the third buckets for running times between 3 to 10 seconds, and so forth. The second-to-last bucket is for running times between 300 to 900 seconds, while the last range is for instances that exceed our time limit of 900 seconds.

4.4 Experimental Results

4.4.1 Comparing All Algorithms Except EUsolver on the Integer and Hacker's Delight Benchmarks

The percentage of finished runs and the time to finish for each algorithm on each benchmark are presented in Figures 8 and 9, respectively.

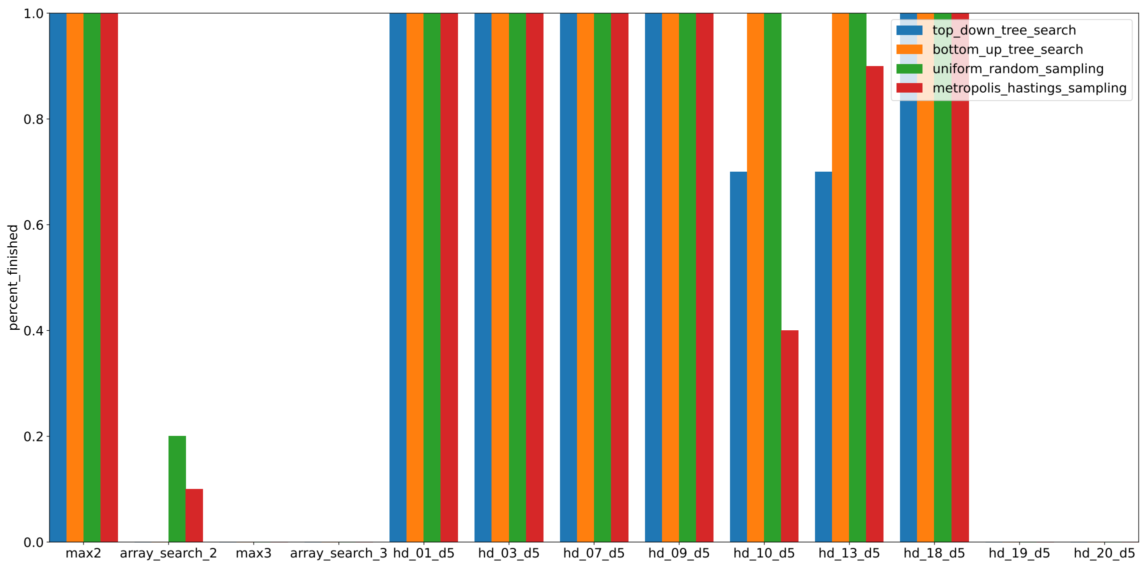

Figure 8: The Percentage of Finished Runs for Each Algorithm on Each Benchmark

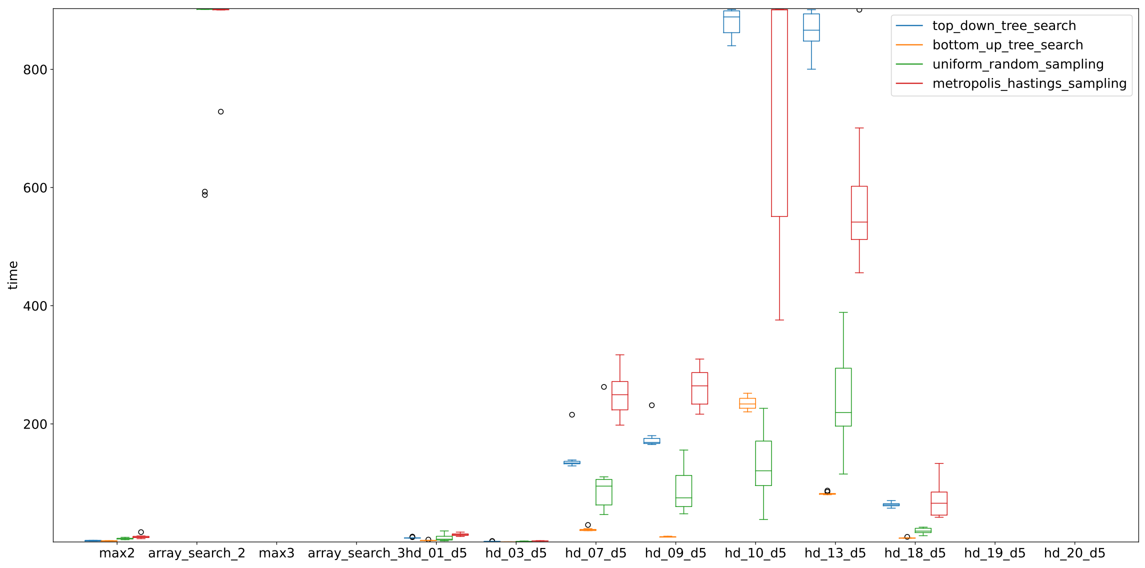

Figure 9: The Time to Finish for Each Algorithm on Each Benchmark

When comparing top-down tree search with bottom-up tree search, bottom-up tree search is the clear winner as it consistently finishes more often on each benchmark while spending less time to finish, with a drastic reduction of time seen in the `hd_10_d5` and `hd_13_d5` benchmarks. This proves the effectiveness of the pruning techniques introduced in bottom-up tree search on the algorithm's scalability.

However, when we compare bottom-up tree search against uniform random search, even though uniform random search tends to spend more time than bottom-up tree search, it achieves an equal success rate. It even makes some breakthroughs on the `array_search_2` benchmark where all runs of bottom-up tree search fail, as shown in Figure 8. This is because bottom-up tree search sequentially builds context-free grammar derivations by combining smaller existing derivations using production rules and searches through those derivations while enforcing that no newly built derivation is duplicated. In contrast, uniform random search randomly searches through context-free grammar's derivations of a given size, makes no effort to eliminate duplicates, and may "get lucky" (i.e., find any duplicate of the correct derivation) in some situations.

What is counterintuitive is that uniform random sampling achieves much better performance compared with Metropolis-Hastings sampling in terms of success rate and search time, even though Metropolis-Hastings sampling is a more sophisticated algorithm. In contrast, uniform random sampling simply randomly samples a context-free grammar's derivations of a given size. To uncover the reason behind this phenomenon, we investigated the logs generated by each algorithm. We found that Metropolis-Hastings sampling generated an order of magnitude candidate programs on each benchmark than the second-ranking algorithm, as shown in Figure 10.

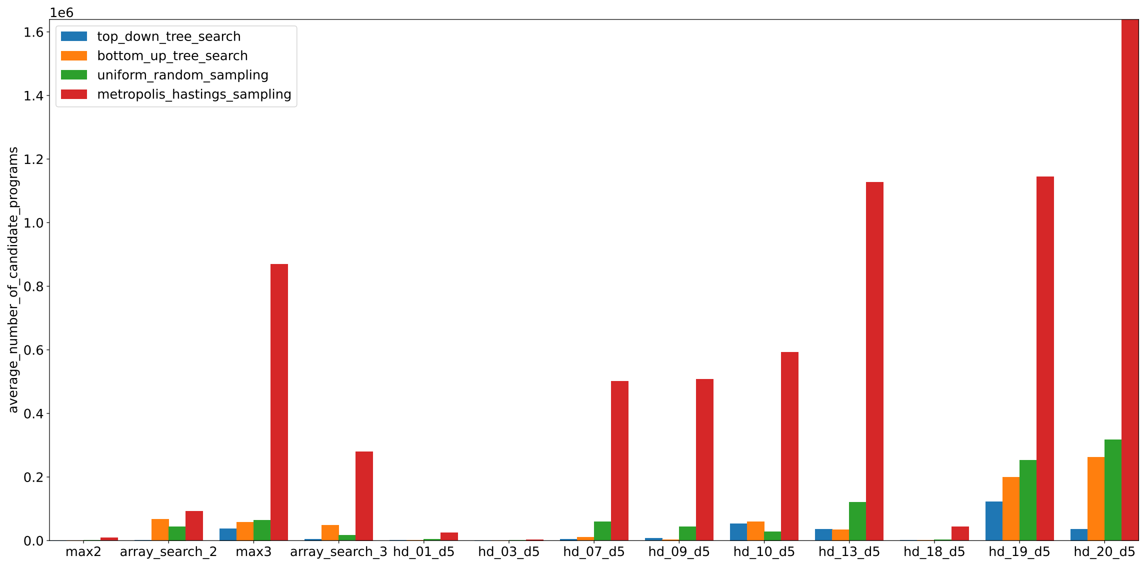

Figure 10: The Average Number of Candidate Programs for Each Algorithm on Each Benchmark

Although so many candidate programs were being explored, we also noticed that the percentages of unique candidate programs explored by Metropolis-Hastings sampling are far less in any other algorithm, as presented in Figure 11. Each candidate program explored by top-down and bottom-up tree search is unique, while even the candidate programs explored by random sampling are mostly (more than 80%) unique. In stark contrast, the percentages of unique candidate programs explored by Metropolis-Hastings sampling were consistently less than 10% on all benchmarks.

In each iteration of the Metropolis-Hastings algorithm, the current candidate program left unchanged or with a randomly-selected subexpression replaced is selected as the next candidate program based on their probability density function values, defined as the number of counterexamples on which they fail. Upon further investigation, we found that the current candidate program is selected instead of a randomly-mutated program in the vast majority of iterations due to the current candidate program having an identical or higher probability density function value. This is because the number of possible

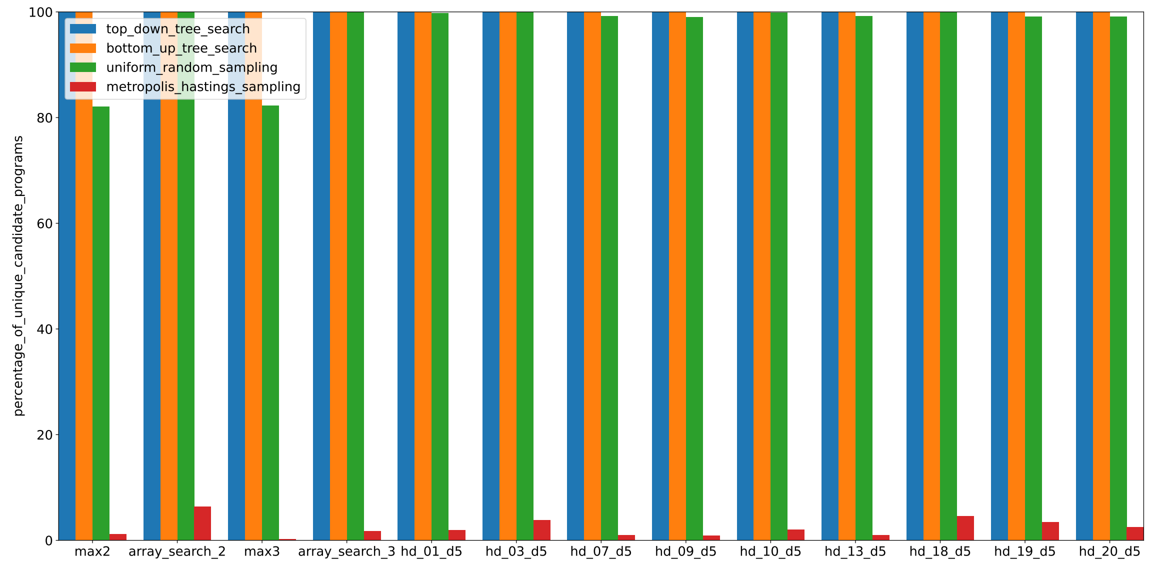

Figure 11: The Percentage of Unique Candidate Programs for Each Algorithm on Each Benchmark

mutations increases exponentially with program size, and most such mutations result in more "faulty" programs with lower probability density function values given the current set of counterexamples, which leads to the Metropolis-Hastings algorithm being stuck on generating the same program candidate for long periods. This problem is also aggravated because the average number of counterexamples is quite small, as shown in Figure 12. It may be difficult to assess the actual quality of a candidate program.

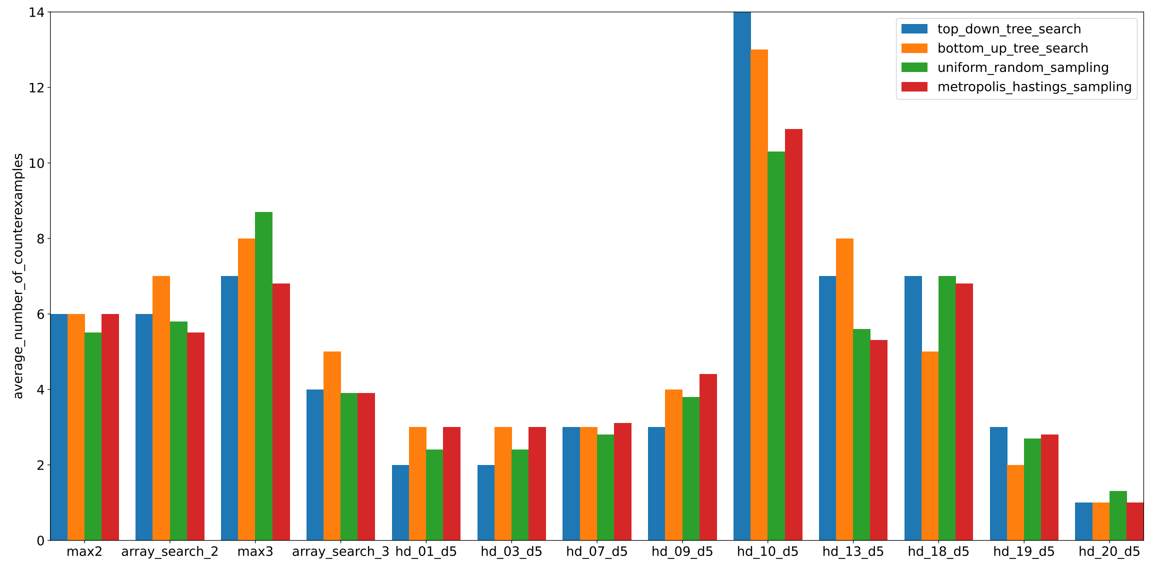

Figure 12: The Average Number of Counterexamples for Each Algorithm on Each Benchmark

Although we haven't implemented the genetic programming algorithm, we hypothesize that we will see similar phenomena when we evaluate it as its operations of mutation and crossover also involve randomly manipulating subexpressions both within a candidate program and between candidate programs, and that it also uses a fitness scores function to assess the quality of candidate programs.

Based on these observations, we suggest using the Metropolis-Hastings algorithm and the genetic programming algorithm only when it is relatively easy to randomly mutate a candidate program (e.g., the candidate program shouldn't be too large). Moreover, each candidate program should be assessed precisely, with different candidate programs having greatly varying qualities (e.g., many input-output examples are given). In other situations, we suggest using the uniform random sampling algorithm instead.

4.4.2 Comparing All Algorithms Including EUsolver on the Integer Benchmarks

We now compare all algorithms, including EUsolver, on the integer benchmarks. For each algorithm on each benchmark, the average time to finish and the average number of unique candidate programs explored are depicted in Figures 13 and 14, respectively.

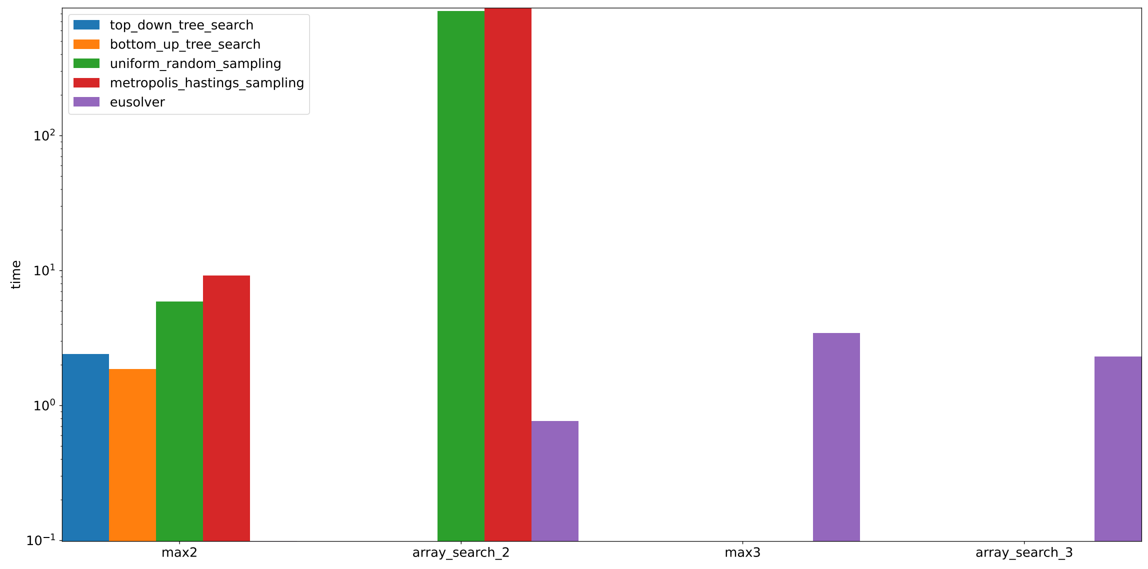

Figure 13: The Average Time to Finish for Each Algorithm on Each Benchmark (Log Scale)

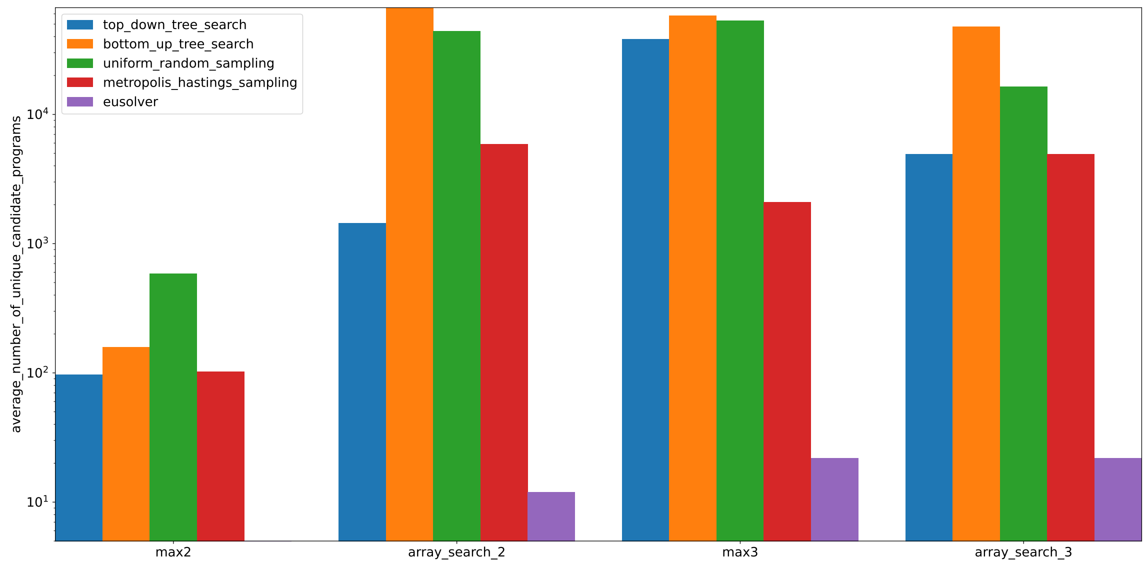

Figure 14: The Average Number of Candidate Programs for Each Algorithm on Each Benchmark (Log Scale)

EUsolver is the only algorithm to succeed on the `max3` and `array_search_3` benchmarks and finishes nearly three orders of magnitude faster than uniform random sampling and Metropolis-Hastings sampling on the `array_search_2` benchmark. Moreover, EUsolver also explores several orders of magnitude fewer unique candidate programs than other algorithms. This starkly contrasts bottom-up tree search and uniform random search exploring more unique candidate programs compared to top-down tree search and Metropolis-Hastings sampling.

The success of EUsolver largely lies in its design of decision tree construction, which is tightly coupled with its syntactic bias. In such a scheme, complex decision trees, which would require exploring a huge search space of context-free derivations to find, can be rapidly constructed out of a relatively small number of simple term and predicate expressions. Moreover, in the decision-tree construction process, either decision trees with different left and right subtrees guaranteed to satisfy all counterexamples are constructed, or the decision-tree construction process fails due to a lack of term or predicate expressions. This is why such few unique candidate programs are generated by EUsolver - all generated candidate programs are correct regarding known counterexamples and can be directly sent to the verification oracle. However, there are no such guarantees of correctness with regards to known counterexamples for the candidate programs explored with top-down tree search, bottom-up tree search, uniform random sampling, and Metropolis-Hastings sampling, and in this case, exploring as many unique programs as fast as possible and checking them on known counterexamples would be beneficial.

4.5 Insights

Several high-level insights can be acquired from the experimental results.

- A program synthesis algorithm that is *guaranteed to find candidate programs correct on all known counterexamples in a fail-fast manner* has the potential for vast gains in synthesis speed. This may require *limiting the hypothesis space* (i.e., only targeting a certain subset of derivations within the original context-free grammar) and *tightly integrating the program synthesis algorithm with the restricted hypothesis space*.
- If this is not the case, a program synthesis algorithm should *explore as many distinct candidate programs as fast as possible*. This is especially important when it is difficult to exploit and further improve currently found candidate programs (such as when the candidate programs are large and when it is difficult to discriminate between the slight performance differences of different candidate programs).

5 Conclusions

Throughout this directed study on program synthesis, I have gained a comprehensive understanding of fundamental concepts and underlying principles, studied, implemented, and compared several classic program synthesis algorithms, and acquired valuable insights into their strengths and weaknesses as well as challenges and future research directions, leading to an in-depth literacy of program synthesis as a whole.

A lot of time has been spent on understanding the details of the program synthesis algorithms, implementing high-level ideas using code, and debugging the implementations. A particular example is EUsolver, which I spent the most time on. It took me quite a while to understand that entropy should be *minimized* instead of maximized (possible typo in the paper [3]) so that the decision tree can be as simple as possible. Moreover, though theoretically possible, any zeroes within the probabilities when calculating entropy would lead to a faulty runtime result of `NaN`. I eventually overcame this problem by calculating the probabilities using *add-one smoothing*. Furthermore, the original paper only mentions that "if a decision tree does not exist, we generate additional terms and predicates and retry." However, we generate additional terms *or* predicates based on the lack of either terms or predicates. This is further complicated by the decision tree construction process being a recursive function called by the function that generates terms and predicates. Generating terms or predicates within such recursive calls would lead to messy code and data consistency problems. Ultimately, I defined two user-defined exceptions, `NotEnoughTermsException` and `NotEnoughPredicatesException`, that would be thrown if there is a lack of terms or predicates. These exceptions are caught and handled by the caller of the recursive decision tree construction function, and terms or predicates are generated in the exception-handling blocks. Such a novel approach allows rapidly exiting nested recursive calls. It is also consistent with Python's philosophy of using exceptions as mundane flow control structures instead of C++'s, Java's, and C#'s philosophy of only using them to handle abnormal, unpredictable, erroneous situations [26].

More importantly, this process has equipped me with a wealth of knowledge and skills that can be readily applied to future research endeavors. For example, background theories and SMT solvers are widely utilized in all alleys of software testing and verification. Furthermore, performing experiments in parallel, automating the processing of lengthy program logs to extract data and gain insights, and developing functions that abstract away the tedious process of drawing grouped bar charts and box plots, problems that I devoted time to solve to finish this directed study, are all common recurring requirements in various research work.

Lastly, I would like to express my deep gratitude to my advisor Caroline Lemieux for the invaluable insights and suggestions she has provided me throughout this project.

References

[1] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. *Syntax-guided synthesis*. IEEE, 2013.

[2] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama. Sygus-comp 2016: Results and analysis. *arXiv preprint arXiv:1611.07627*, 2016.

[3] R. Alur, A. Radhakrishna, and A. Udupa. Scaling enumerative program synthesis via divide and conquer. In *Tools and Algorithms for the Construction and Analysis of Systems: 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I 23*, pages 319–336. Springer, 2017.

[4] S. Bansal and A. Aiken. Binary translation using peephole superoptimizers. In *OSDI*, volume 8, pages 177–192, 2008.

[5] G. Barthe, J. M. Crespo, S. Gulwani, C. Kunz, and M. Marron. From relational verification to simd loop synthesis. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 123–134, 2013.

[6] A. W. Biermann. The inference of regular lisp programs from examples. *IEEE transactions on Systems, Man, and Cybernetics*, 8(8):585–600, 1978.

[7] S. Cai, B. Li, and X. Zhang. Local search for smt on linear integer arithmetic. In *Computer*

Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part II, pages 227–248. Springer, 2022.

[8] S. Cheema, S. Buchanan, S. Gulwani, and J. J. LaViola Jr. A practical framework for constructing structured drawings. In *Proceedings of the 19th international conference on Intelligent User Interfaces*, pages 311–316, 2014.

[9] R. Chugh, B. Hempel, M. Spradlin, and J. Albers. Programmatic and direct manipulation, together at last. *ACM SIGPLAN Notices*, 51(6):341–354, 2016.

[10] A. Cropper and S. Dumančić. Inductive logic programming at 30: a new introduction. *Journal of Artificial Intelligence Research*, 74:765–850, 2022.

[11] L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. Technical Report MSR-TR-2009-121, September 2009. URL `https://www.microsoft.com/en-us/research/publication/generalized-efficient-array-decision-procedures/`. A conference version of this report appears in the proceedings of FMCAD 2009.

[12] L. D'Antoni, R. Samanta, and R. Singh. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification*, pages 383–401. Springer, 2016.

[13] R. Frost, R. Hafiz, and P. Callaghan. Modular and efficient top-down parsing for ambiguous left-recursive grammars. In *Proceedings of the Tenth International Conference on Parsing Technologies*, pages 109–120, 2007.

[14] C. Green. Application of theorem proving to problem solving. In *Readings in Artificial Intelligence*, pages 202–222. Elsevier, 1981.

[15] S. Gulwani. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[16] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. *ACM SIGPLAN Notices*, 46(6):50–61, 2011.

[17] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[18] S. Gulwani, M. Mayer, F. Niksic, and R. Piskac. Strisynth: synthesis for live programming. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 701–704. IEEE, 2015.

[19] S. Gulwani, J. Esparza, and O. Grumberg. Programming by examples (and its applications in data wrangling)[c]. verification and synthesis of correct and secure systems, 2016.

[20] B. Hempel and R. Chugh. Semi-automated svg programming via direct manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 379–390, 2016.

[21] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 215–224, 2010.

[22] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *International conference on computer aided verification*, pages 226–238. Springer, 2005.

[23] R. Joshi, G. Nelson, and K. Randall. Denali: A goal-directed superoptimizer. *ACM SIGPLAN Notices*, 37(5):304–314, 2002.

[24] G. Juniwal, A. Donzé, J. C. Jensen, and S. A. Seshia. Cpsgrader: Synthesizing temporal logic testers for auto-grading an embedded systems laboratory. In *Proceedings of the 14th International Conference on Embedded Software*, pages 1–10, 2014.

[25] G. Katz and D. Peled. Genetic programming and model checking: Synthesizing new mutual exclusion algorithms. In *Automated Technology for Verification and Analysis: 6th International Symposium, ATVA 2008, Seoul, Korea, October 20-23, 2008. Proceedings 6*, pages 33–47. Springer, 2008.

[26] J. R. Kiniry. Exceptions in java and eiffel: Two extremes in exception design and application. *Advanced Topics in Exception Handling Techniques*, pages 288–300, 2006.

[27] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2011.

[28] Z. Manna and R. Waldinger. Knowledge and reasoning in program synthesis. *Artificial intelligence*, 6(2):175–208, 1975.

[29] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165, 1971.

[30] H. Massalin. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News*, 15(5):122–126, 1987.

[31] J. McCarthy. Towards a mathematical science of computation. *Program Verification: Fundamental Issues in Computer Science*, pages 35–56, 1993.

[32] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai. A machine learning framework for programming by example. In *International Conference on Machine Learning*, pages 187–195. PMLR, 2013.

[33] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE, 2013.

[34] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. *ACM SIGPLAN Notices*, 50(6):1–11, 2015.

[35] E. Parisotto, A.-r. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855*, 2016.

[36] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 297–310, 2016.

[37] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190, 1989.

[38] O. Polozov and S. Gulwani. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126, 2015.

[39] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News*, 41(1):305–316, 2013.

[40] D. E. Shaw, W. R. Swartout, and C. C. Green. Inferring lisp programs from examples. In *IJCAI*, volume 75, pages 260–267, 1975.

[41] R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *International Conference on Computer Aided Verification*, pages 634–651. Springer, 2012.

[42] R. Singh and S. Gulwani. Transforming spreadsheet data types using examples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 343–356, 2016.

[43] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26, 2013.

[44] D. C. Smith. *Pygmalion: a creative programming environment.* Stanford University, 1975.

[45] A. Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.

[46] P. D. Summers. A methodology for lisp program construction from examples. *Journal of the ACM (JACM)*, 24(1):161–175, 1977.

[47] O. Tange. *GNU parallel 2018.* Lulu. com, 2018.

[48] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur. Transit: specifying protocols with concolic snippets. *ACM SIGPLAN Notices*, 48(6):287–296, 2013.

[49] R. J. Waldinger and R. C. Lee. Prow: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*, pages 241–252, 1969.

[50] H. S. Warren. *Hacker's delight.* Pearson Education, 2013.