Syntax-Guided Synthesis

Rajeev ALUR [a], Rastislav BODIK [b], Eric DALLAL [c], Dana FISMAN [a],
Pranav GARG [d], Garvit JUNIWAL [b], Hadas KRESS-GAZIT [e], P. MADHUSUDAN [d],
Milo M. K. MARTIN [a], Mukund RAGHOTHAMAN [a], Shamwaditya SAHA [d],
Sanjit A. SESHIA [b], Rishabh SINGH [f,g], Armando SOLAR-LEZAMA [g],
Emina TORLAK [h] and Abhishek UDUPA [a,1]

[a] *University of Pennsylvania*
[b] *University of California, Berkeley*
[c] *University of Michigan*
[d] *University of Illinois at Urbana-Champaign*
[e] *Cornell University*
[f] *Microsoft Research, Redmond*
[g] *Massachusetts Institute of Technology*
[h] *University of Washington*

Abstract. The classical formulation of the program-synthesis problem is to find a program that meets a correctness specification given as a logical formula. Recent work on program synthesis and program optimization illustrates many potential benefits of allowing the user to supplement the logical specification with a syntactic template that constrains the space of allowed implementations. Our goal is to identify the core computational problem common to these proposals in a logical framework. The input to the *syntax-guided synthesis* problem (SyGuS) consists of a background theory, a semantic correctness specification for the desired program given by a logical formula, and a syntactic set of candidate implementations given by a grammar. The computational problem then is to find an implementation from the set of candidate expressions so that it satisfies the specification in the given theory. We describe alternative solution strategies that combine learning, counterexample analysis and constraint solving. We report on prototype implementations, and present experimental results on the set of benchmarks collected as part of the first SyGuS-Comp competition held in July 2014.

Keywords. Program Synthesis, Constraint Solving, Counterexamples, Machine Learning.

1. Introduction

The goal of *program synthesis* is to automatically synthesize a program implementation that satisfies a given correctness specification. Being able to do this reliably even at a small scale could have a significant impact on software development by simultaneously making software more reliable and reducing the effort required to produce it. Classically, program synthesis was viewed as a problem in deductive theorem proving: a program is

[1] This paper is an extended version of [1]. Compared to [1] it contains additional solution strategies and reports on more extensive evaluation on a larger set of benchmarks.

derived from the constructive proof of the theorem that states that for all inputs, there exists an output, such that the desired correctness specification holds (see [6]). Recent work, however, has demonstrated an alternative approach to synthesis where the programmer, in addition to the correctness specification, provides a syntactic template for the desired program. For instance, in the programming approach advocated by the SKETCH system, a programmer writes a partial program with incomplete details, and the synthesizer fills in the missing details using user-specified assertions as the correctness specification [7].

The use of a syntactic constraint in addition to the correctness specification has many potential benefits. First, the synthesis problem becomes more tractable because the syntactic template limits the search space of potential implementations. Second, this approach gives the programmer the flexibility to describe the desired artifact using a combination of syntactic and semantic constraints. Such forms of *multi-modal* specifications have the potential to make programming more intuitive. Third, the syntactic template can be used to constrain the space of implementations for the purpose of performance optimizations. For example, to optimize the computation of the product of two matrices, we can syntactically limit the search space to implementations that use only 7 recursive multiplications on smaller sub-matrices to force the synthesizer to discover Strassen's algorithm. Fourth, because the synthesis problem boils down to finding a correct expression from the syntactic space of expressions, this search problem lends itself to machine learning and inductive inference as discussed in Section 3.

The use of syntactic restrictions has been embraced by a number of recent efforts such as synthesis of loop-free programs [8], synthesis of Excel macros from examples [9], program de-obfuscation [10], synthesis of protocols from the skeleton and example behaviors [11], synthesis of loop-bodies from pre/post conditions [12], integration of constraint solvers in programming environments for program completion [13], and super-optimization by finding equivalent shorter loop bodies [14]. Also related are techniques for automatic generation of invariants using templates and by learning [15,16,17], and recent work on solving quantified Horn clauses [18]. Despite their common approach to synthesis based on a combination of syntactic and semantic constraints, however, each of these tools uses a different representation of the synthesis problem and different algorithms to solve it. This makes it difficult to systematically explore the tradeoffs between many of the algorithms used by these systems or to transfer synthesis technologies from one system to another.

The main contribution of this paper is to formalize the *syntax guided synthesis problem* (SyGuS) and to present a standard format to describe such problems. The goal of the SyGuS formalism is to capture the syntactic guidance that is essential to recent synthesis efforts under a general formal framework based on logics and grammars—as opposed to a specific programming language. In our formalization, the correctness specification of the function f to be synthesized is given as a logical formula φ that uses symbols from a background theory T. The syntactic space of possible implementations for f is described as a set L of expressions built from the theory T, and this set is specified using a grammar. The syntax-guided synthesis problem then is to find an implementation expression $e \in L$ such that the formula $\varphi[f/e]$ is valid in the theory T. It is important to note that this problem cannot be translated to determining the truth of a formula in the theory T, even with additional quantifiers. To illustrate an application of the SyGuS-problem, suppose we want to find a completion of a partial program with holes so as to satisfy given assertions. A typical SyGuS-encoding of this task will translate the concrete parts of the partial program and the assertions into the specification formula φ, while the holes will be represented with the unknown functions to be synthesized, and the space of expressions that can substitute the holes will be captured by the grammar.

The precedent for this kind of standardization effort can be found in the development of solvers for Satisfiability Modulo Theories (SMT) [4]. Modern SMT solvers can determine the truth of a complex logical formula with thousands of variables, despite the computational intractability of the problem (see [4,5]). A key driving force behind the sustained innovations that have made this possible has been the standardization of a common interchange format for benchmarks called SMT-LIB (see smt-lib.org) and the associated annual competition (see smtcomp.org). These efforts have proved to be instrumental in creating a virtuous feedback loop between the developers and users of SMT solvers: with the availability of open-source and highly optimized solvers, researchers from verification and other application domains find it beneficial to translate their problems into the common format instead of attempting to develop their own tools from scratch, and the limitations of the current SMT tools are constantly exposed by the ever growing repository of different kinds of benchmarks. Many of the synthesis systems mentioned earlier actually rely on SMT solvers internally, but the existing formalization behind SMT-LIB has no suitable abstraction for capturing the syntactic guidance. By creating a common format for synthesis problems that is able to capture these syntactic restrictions, we expect synthesis to benefit from a similar feedback loop.

The rest of the paper is organized in the following manner. In Section 2, we formalize the core problem of syntax-guided synthesis with examples. In Section 3, we discuss a generic architecture for solving the proposed problem using the iterative *counter-example guided inductive synthesis* strategy [19] that combines a learning algorithm with a verification oracle. For the learning algorithm, we show alternative solution strategies, as follows. The *enumerative* technique generates the candidate expressions of increasing size relying on the input-output examples for pruning. The *constraint-based* techniques encode the grammar production rules in a symbolic fashion, and generate a syntactic constraint stipulating the existence of an expression adhering the grammar rules of upto certain size or depth; then they call an SMT solver to find an expression consistent with all the examples encountered so far. The *stochastic* search uniformly samples the set L of expressions as a starting point, and then executes (probabilistic) traversal of the graph where two expressions are neighbors if one can be obtained from the other by a single edit operation on the parse tree. Last, the *geometry-based* solver is targeting generation of guarded linear expressions only; it is agnostic to the SyGuS constraints and encodes the examples as pairs $(\vec{x}, val)$ where $\vec{x}$ are perceived as vectors over k variables. It works in two phases: in the first it uses computational geometry techniques to generate a set of linear expressions such that at least one of them works for any given example; and in the second phase it uses decision tree classifiers to generate the guards. In Section 4 we report on the performance of prototype implementations for the mentioned solution strategies, over a set of 250 benchmarks collected as part of the first competition for solvers for SyGuS — SYGUS-COMP 2014 that took place in July 2014.[2] We conclude in Section 5.

[2] SYGUS-COMP 2014 was a satellite event of SYNT/CAV 2014, and a part of FLoC Olympic Games 2014 at the Vienna Summer of Logic (VSL).

2. Problem Formulation

At a high level, the functional synthesis problem consists of finding a function f such that some logical formula φ capturing the correctness of f is valid. In syntax-guided synthesis, the synthesis problem is constrained in three ways: (1) the logical symbols and their interpretation are restricted to a *background theory*, (2) the *specification* φ is limited to a first order formula in the background theory with all its variables universally quantified, and (3) the universe of possible functions f is restricted to syntactic expressions described by a *grammar*. We now elaborate on each of these points.

Background Theory

The syntax for writing specifications is the same as classical typed first-order logic, but the formulas are evaluated with respect to a specified background theory T. The theory gives the vocabulary used for constructing formulas, the set of values for each type, and the interpretation for each of the function and relation (predicate) symbols in the vocabulary. We are mainly interested in theories T for which well-understood decision procedures are available for determining satisfaction modulo T (see [4] for a survey). A typical example is the theory of *linear integer arithmetic* (LIA) where each variable is either a boolean or an integer, and the vocabulary consists of boolean and integer constants, standard boolean connectives, addition ($+$), comparison ($\leq$), and conditionals (*ITE*). Note that the background theory can be a combination of logical theories, for instance, LIA and the theory of uninterpreted functions with equality.

Correctness Specification

For the function f to be synthesized, we are given the type of f and a formula φ as its correctness specification. The formula φ is a Boolean combination of predicates from the background theory, involving universally quantified free variables, symbols from the background theory, and the function symbol f, all used in a type-consistent manner.

Example 1 *Assuming the background theory is LIA, consider the specification of a function f of type $int \times int \mapsto int$:*

$$\varphi_1 : \; f(x,y) = f(y,x) \;\wedge\; f(x,y) \geq x.$$

The free variables in the specification are assumed to be universally quantified: a given function f satisfies the above specification if the quantified formula $\forall x,y.\, \varphi_1$ holds, or equivalently, if the formula φ_1 is valid.

Set of Candidate Expressions

In order to make the synthesis problem tractable, the "syntax-guided" version allows the user to impose structural (syntactic) constraints on the set of possible functions f. The structural constraints are imposed by restricting f to the set L of functions defined by a given context-free grammar G_L. Each expression in L has the same type as that of the function f, and uses the symbols in the background theory T along with the variables corresponding to the formal parameters of f.

Example 2 *Suppose the background theory is LIA, and the type of the function f is $int \times int \mapsto int$. We can restrict the set of expressions $f(x,y)$ to be linear expressions of the inputs by restricting the body of the function to expressions in the set L_1 described by the grammar below:*

$$LinExp \ := \ x \mid y \mid Const \mid LinExp + LinExp$$

Alternatively, we can restrict $f(x,y)$ to conditional expressions with no addition by restricting the body terms from the set L_2 described by:

$$Term := x \mid y \mid Const \mid ITE(Cond, Term, Term)$$

$$Cond := Term \leq Term \mid Cond \wedge Cond \mid \neg Cond \mid (Cond)$$

Grammars can be conveniently used to express a wide range of constraints, and in particular, to bound the depth and/or the size of the desired expression.

SyGuS Problem Definition

Informally, given the correctness specification φ and the set L of candidates, we want to find an expression $e \in L$ such that if we use e as an implementation of the function f, the specification φ is valid. Let us denote the result of replacing each occurrence of the function symbol f in φ with the expression e by $\varphi[f/e]$. Note that we need to take care of binding of input values during such a substitution: if f has two inputs that the expressions in L refer to by the variable names x and y, then the occurrence $f(e_1,e_2)$ in the formula φ must be replaced with the expression $e[x/e_1, y/e_2]$ obtained by replacing x and y in e by the expressions e_1 and e_2, respectively. Now we can define the *syntax-guided synthesis* problem, SyGuS for short, precisely:

> Given a background theory T, a typed function symbol f, a formula φ over the vocabulary of T along with f, and a set L of expressions over the vocabulary of T and of the same type as f, find an expression $e \in L$ such that the formula $\varphi[f/e]$ is valid modulo T.

Example 3 *For the specification φ_1 presented earlier, if the set of allowed implementations is L_1 as shown before, there is no solution to the synthesis problem. On the other hand, if the set of allowed implementations is L_2, a possible solution is the conditional if-then-else expression $ITE(x \geq y, x, y)$.*

In some special cases, it is possible to reduce the decision problem for syntax guided synthesis to the problem of deciding formulas in the background theory using additional quantification. For example, every expression in the set L_1 is equivalent to $ax+by+c$, for integer constants a,b,c. If φ is the correctness specification, then deciding whether there exists an implementation for f in the set L_1 corresponds to checking whether the formula $\exists a,b,c. \forall X. \, \varphi[f/ax+by+c]$ holds, where X is the set of all free variables in φ. This reduction was possible for L_1 because the set of all expressions in L_1 can be represented by a single parameterized expression in the original theory. However, the grammar may permit expressions of arbitrary depth which may not be representable in this way, as in the case of L_2.

Synthesis of Multiple Functions

A general synthesis problem can involve more than one unknown function. In principle, adding support for problems with more than one unknown function is merely a matter of syntactic sugar. For example, suppose we want to synthesize functions $f_1(x_1)$ and $f_2(x_2)$, with corresponding candidate expressions given by grammars G_1 and G_2, with start non-terminals S_1 and S_2, respectively. Both functions can be encoded with a single function $f_{12}(id, x_1, x_2)$. The set of candidate expressions is described by the grammar that contains the rules of G_1 and G_2 along with a new production $S := ITE(id = 0, S_1, S_2)$, with the new start non-terminal S. Then, every occurrence of $f_1(x_1)$ in the specification can be replaced with $f_{12}(0, x_1, *)$ and every call to $f_2(x_2)$ can be replaced with $f_{12}(1, *, x_2)$. Although adding support for multiple functions does not fundamentally increase the expressiveness of the notation, it does offer significant convenience in encoding real-world synthesis problems.

Let Expressions in Grammar Productions

The SMT-LIB interchange format for specifying constraints allows the use of let expressions as part of the formulas, and this is supported by our language also: $(let \ \ [var = e_1] \ \ e_2)$. While *let*-expressions in a specification can be desugared, the same does not hold when they are used in a grammar. As an example, consider the grammar below for the set of candidate expressions for the function $f(x,y)$:

$$T := (let\ ((z\ Int\ U))\ z + z)$$

$$U := x \mid y \mid Const \mid U + U \mid U * U \mid T$$

The top-level expression specified by this grammar is the sum of two identical subexpressions built using arithmetic operators, and such a structure cannot be specified using a standard context-free grammar. In the example above, every *let* introduced by the grammar uses the same variable name. If the application of *let*-expressions are nested in the derivation tree, the standard rules for shadowing of variable definitions determine which definition corresponds to which use of the variable.

SyGuS-LIB Input Format

To specify the input to the SyGuS problem, we have developed an interchange format, called SyGuS-LIB[3]. It is based on the syntax of SMT-LIB2—the input format accepted by the SMT solvers (see smt-lib.org). The input for the SyGuS problem to synthesize the function f with the specification φ_1 in the theory LIA, with the grammar for the languages L_1 is encoded in SyGuS-LIB as:

```
(set-logic LIA)
(SyGuS-fun f ((x Int) (y Int)) Int
  ((Start Int (x y
               (Constant Int)
               (+ Start Start)))))
(declare-var a Int)
(declare-var b Int)
```

[3] The full description of SyGuS-LIB is available at [2].

```
(constraint (= (f a b) (f b a)))
(constraint (>= (f a b) a))
(check-synth)
```

Optimality Criterion

The answer to our synthesis problem need not be unique: there may be two expressions e_1 and e_2 in the set L of allowed expressions such that both implementations satisfy the correctness specification φ. Ideally, we would like to associate a cost with each expression, and consider the problem of *optimal synthesis* which requires the synthesis tool to return the expression with the least cost among the correct ones. A natural cost metric is the size of the expression. In the presence of *let*-expressions, the size directly corresponds to the number of instructions in the corresponding straight-line code, and thus such a metric can be used effectively for applications such as super-optimization.

3. Inductive Synthesis

Algorithmic approaches to program synthesis range over a wide spectrum, from *deductive synthesis* to *inductive synthesis*. In deductive program synthesis (e.g., [6]), a program is synthesized by constructively proving a theorem, employing logical inference and constraint solving. On the other hand, inductive synthesis [20,21,22] seeks to find a program matching a set of input-output examples. It is thus an instance of learning from examples, also termed as *inductive inference* or *machine learning* [23,24]. Many current approaches to synthesis blend induction and deduction [25]; syntax guidance is usually a key ingredient in these approaches.

Inductive synthesizers generalize from examples by searching a restricted space of programs. In machine learning, this restricted space is called the *concept class*, and each element of that space is often called a candidate *concept*. The concept class is usually specified syntactically. Inductive learning is thus a natural fit for the syntax-guided synthesis problem introduced in this paper: the concept class is simply the set L of permissible expressions.

3.1. Synthesis via Active Learning

A common approach to inductive synthesis is to formulate the overall synthesis problem as one of *active learning* using a *query-based* model. Active learning is a special case of machine learning in which the learning algorithm can control the selection of examples that it generalizes from and can query one or more oracles to obtain both examples as well as labels for those examples. In our setting, we can consider the labels to be binary: positive or negative. A positive example is simply an interpretation to f in the background theory T that is consistent with the specification φ; i.e., it is a valuation to the arguments of the function symbol f along with the corresponding valuation of f that satisfies φ. A negative example is any interpretation of f that is not consistent with φ. We refer the reader to a paper by Angluin [26] for an overview of various models for query-based active learning.

In program synthesis via active learning, the query oracles are often implemented using deductive procedures such as model checkers or satisfiability solvers. Thus, the

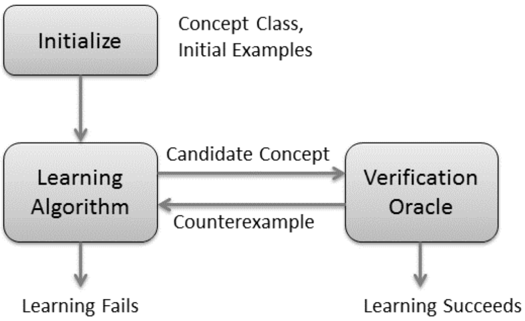


Figure 1. Counterexample-Guided Inductive Synthesis (CEGIS)

overall synthesis algorithm usually comprises a top-level inductive learning algorithm that invokes deductive procedures (query oracles); e.g., in our problem setting, it is intuitive, although not required, to implement an oracle using an SMT solver for the theory T. Even though this approach combines induction and deduction, it is usually referred to in the literature simply as "inductive synthesis." We will continue to use this terminology in the present paper.

Consider the syntax-guided synthesis problem of Sec. 2. Given the tuple (T, f, φ, L), there are two important choices one must make to fix an inductive synthesis algorithm: (1) *search strategy:* How should one search the concept class L? and (2) *example selection strategy:* Which examples do we learn from?

3.2. Counterexample-Guided Inductive Synthesis

Counterexample-guided inductive synthesis (CEGIS) [19,29] shown in Figure 1 is perhaps the most popular approach to inductive synthesis today. CEGIS has close connections to algorithmic debugging using counterexamples [22] and counterexample-guided abstraction refinement (CEGAR) [30]. This connection is no surprise, because both debugging and abstraction-refinement involve synthesis steps: synthesizing a repair in the former case, and synthesizing an abstraction function in the latter (see [25] for a more detailed discussion).

The defining aspect of CEGIS is its example selection strategy: *learning from counterexamples provided by a verification oracle*. The learning algorithm, which is initialized with a particular choice of concept class L and possibly with an initial set of (positive) examples, proceeds by searching the space of candidate concepts for one that is consistent with the examples seen so far. There may be several such consistent concepts, and the search strategy determines the chosen candidate, an expression e. The concept e is then presented to the verification oracle $\mathcal{O}_V$, which checks the candidate against the correctness specification. $\mathcal{O}_V$ can be implemented as an SMT solver that checks whether $\varphi[f/e]$ is valid modulo the theory T. If the candidate is correct, the synthesizer terminates and outputs this candidate. Otherwise, the verification oracle generates a counterexample, an interpretation to the symbols and free variables in $\varphi[f/e]$ that falsifies it. This counterexample is returned to the learning algorithm, which adds the counterexample to its set of examples and repeats its search; note that the precise encoding of a counterexample and its use can vary depending on the details of the learning algorithm

Expression to Verifier	Learned Test Input
x	$\langle x=0, y=1\rangle$
y	$\langle x=1, y=0\rangle$
1	$\langle x=0, y=0\rangle$
$x+y$	$\langle x=1, y=1\rangle$
$ITE(x \leq y, y, x)$	–

Table 1. A run of the enumerative algorithm

employed. It is possible that, after some number of iterations of this loop, the learning algorithm may be unable to find a candidate concept consistent with its current set of (positive/negative) examples, in which case the learning step, and hence the overall CEGIS procedure, fails.

Several search strategies are possible for learning a candidate expression in L, each with its pros and cons. In the following sections, we describe three different search strategies and illustrate the main ideas in each using a small example.

3.3. Illustrative Example

Consider the problem of synthesizing a program which returns the maximum of two integer inputs. The specification of the desired program max is given by:

$$\max(x,y) \geq x \ \wedge \ \max(x,y) \geq y \wedge$$
$$(\max(x,y) = x \vee \max(x,y) = y)$$

The search space is suitably defined by an expression grammar which includes addition, subtraction, comparison, conditional operators and the integer constants 0 and 1.

3.4. Enumerative Learning

The enumerative learning algorithm [11] adopts a dynamic programming based search strategy that systematically enumerates concepts (expressions) in increasing order of complexity. Various complexity metrics can be assigned to concepts, the simplest being the expression size. The algorithm needs to store all enumerated expressions, because expressions of a given size are composed to form larger expressions in the spirit of dynamic programming. The algorithm maintains a set of concrete test cases, obtained from the counterexamples returned by the verification oracle. These concrete test cases are used to reduce the number of expressions stored at each step by the dynamic programming algorithm.

We demonstrate the working of the algorithm on the illustrative example. Table 1 shows the expressions submitted to the verification oracle (an SMT solver) during the execution of the algorithm and the values for which the expression produces incorrect results. Initially, the algorithm submits the expression x to the verifier. The verifier returns a counterexample $\langle x=0, y=1\rangle$, corresponding to the case where the expression x violates the specification. The expression enumeration is started from scratch every time a counterexample is added. All enumerated expressions are checked for conformance with the accumulated (counter)examples before making a potentially expensive query to the

Production	Component	
$E \to ITE(B,E,E)$	Inputs:	$(i_1 : B)(i_2, i_3 : E)$
	Output:	$(o : E)$
	Spec:	$o = ITE(i_1, i_2, i_3)$
$B \to E \leq E$	Inputs:	$(i_1, i_2 : E)$
	Output:	$(o : B)$
	Spec:	$o = i_1 \leq i_2$

Table 2. Components from Productions

verifier. In addition, if the algorithm enumerates two expressions e and e' that evaluate to the same value on the examples obtained so far, then only one of e or e' needs to be considered for the purpose of constructing larger expressions.

Proceeding with the illustrative example, the algorithm then submits the expression y and the constant 1 to the verifier. The verifier returns the values $\langle x = 1, y = 0\rangle$ and $\langle x = 0, y = 0\rangle$, respectively, as counterexamples to these expressions. The algorithm then submits the expression $x + y$ to the verifier. The verifier returns the values $\langle x = 1, y = 1\rangle$ as a counterexample. The algorithm then submits the expression shown in the last row of Table 1 to the verifier. The verifier certifies it to be correct and the algorithm terminates.

The optimization of pruning based on concrete counterexamples helps in two ways. First, it reduces the number of invocations of the verification oracle. In the example we have described, the correct expression was examined after only four calls to the SMT solver, although about 200 expressions were enumerated by the algorithm. Second, it reduces the search space for candidate expressions significantly (see [11] for details). For instance, in the run of the algorithm on the example, although the algorithm enumerated about 200 expressions, only about 80 expressions were stored.

3.5. Constraint-based Learning

The symbolic CEGIS approach uses a constraint solver both for searching for a candidate expression that works for a set of concrete input examples (concept learning) and verification of validity of an expression for all possible inputs. We use component based synthesis of loop-free programs as described by Jha et al. [10,8]. Each production in the grammar corresponds to a component in a library. A loop-free program comprising these components corresponds to an expression from the grammar. Some sample components for the illustrative example are shown in Table 2 along with their corresponding productions.

The input/output ports of these components are typed and only well-typed programs correspond to well-formed expressions from the grammar. To ensure this, Jha et al.'s encoding [8] is extended with typing constraints. We illustrate the working of this algorithm on the maximum of two integers example. The library of allowed components is instantiated to contain one instance each of *ITE* and all comparison operators($\leq, \geq, =$) and the concrete example set is initialized with $\langle x = 0, y = 0\rangle$. The first candidate loop-free program synthesized corresponds to the expression x. This candidate is submitted to the verification oracle which returns with $\langle x = -1, y = 0\rangle$ as a counterexample. This counterexample is added to the concrete example set and the learning algorithm is queried again. The SMT formula for learning a candidate expression is solved in an incremental fashion; i.e., the constraint for every new example is added to the list of constraints from

Iteration	Loop-free program	Learned counter-example
1	$o_1 := x$	$\langle x = -1, y = 0\rangle$
2	$o_1 := x \leq x$ $o_2 := ITE(o_1,\ y,\ x)$	$\langle x = 0, y = -1\rangle$
3	$o_1 := y \geq x$ $o_2 := ITE(o_1,\ y,\ x)$	–

Table 3. A run of the constraint learning algorithm

the previous examples. The steps of the algorithm on the illustrative example are shown in Table 3.

If synthesis fails for a component library, we add one instance of every operator to the library and restart the algorithm with the new library. We also tried a modification to the original algorithm [8], in which, instead of searching for a loop-free program that utilizes all components from the given library at once, we search for programs of increasing length such that every line can still select any component from the library. The program length is increased in an exponential fashion (1, 2, 4, 8, $\cdots$) for a good coverage. This approach provides better running times for most benchmarks in our set, but it can also be more expensive in certain cases.

Sketch-based Solver

An alternative constraint-based learning approach is implemented in the Sketch-based solver, which uses the symbolic CEGIS implementation in SKETCH [19,29]. This solver implementation lets us leverage and build upon numerous encoding optimizations in SKETCH for an efficient symbolic encoding of our constraints. A grammar for the unknown function is encoded as a typed SKETCH function. The productions for each grammar variable are encoded as a typed SKETCH generator, where holes are used to encode different return choices for the production rules. For the illustrative example, the unknown function f is translated to the following SKETCH function.

```
int f(int x, int y){
  return rulefunc2(x,y,BND);
}
```

The generator names for the grammar production rules such as `rulefunc2` are freshly generated for each grammar variable. Each generator call also takes an additional parameter `BND` that is used to bound the number of recursive calls of the generator. For instance, the *Term* production rules in the grammar of language L_2 are encoded as the following SKETCH generator.

```
generator int rulefunc2(int x, int y, int bnd){
  assert(bnd>0);
  if(??) return x;
  if(??) return y;
  if(??) return ??;
  int t1 = rulefunc2(x,y,bnd-1);
  int t2 = rulefunc2(x,y,bnd-1);
  bit t3 = rulefunc1(x,y,bnd-1);
  if(??) return ite(t3, t1, t2);
  assert false;
}
```

The generator `rulefunc2` first ensures that the recursion inline bound is not violated (`bnd` > 0). It uses the SKETCH hole construct (`??`) to encode different rule choices, where a hole can be assigned any constant integer value. An important optimization in this encoding is to factor out recursive generator calls into temporary variables such as `t1` and `t2` instead of inlining them into return expressions of the rule choices. This helps the solver perform symmetry breaking to identify common sub-expressions in recursive calls.

Since SKETCH uses a SAT solver for both concept learning as well as the verification step, we need to bound the search space for possible concepts. This is done by bounding the space of possible inputs and the inlining depth of recursive calls in grammar production rules. The Sketch-based solver first starts with small bounds for inputs and inline amount, and then iteratively increases the two bounds in a coordinated way.

3.6. Learning by Stochastic Search

The stochastic learning procedure is an adaptation of the algorithm recently used by Schufza et al. [14] for program super-optimization. The learning algorithm of the CEGIS loop uses the Metropolis-Hastings procedure to sample expressions. The probability of choosing an expression e is proportional to a value $\text{Score}(e)$, which indicates the extent to which e meets the specification φ. The Metropolis-Hastings algorithm guarantees that, in the limit, expressions e are sampled with probability proportional to $\text{Score}(e)$. To complete the description of the search procedure, we need to define $\text{Score}(e)$ and the Markov chain used for successor sampling. We define $\text{Score}(e)$ to be $\exp(-\beta C(e))$, where β is a smoothing constant (set by default to 0.5), and the cost function $C(e)$ is the number of concrete examples on which e does *not* satisfy φ.

We now describe the Markov chain underlying the search. Fix an expression size n, and consider all expressions in L with parse trees of size n. The initial candidate is chosen uniformly at random from this set [31]. Given a candidate e, we pick a node v in its parse tree uniformly at random. Let e_v be the subexpression rooted at this node. This subtree is replaced by another subtree (of the same type) of size equal to $|e_v|$ chosen uniformly at random. Given the original candidate e, and a mutation e' thus obtained, the probability of making e' the new candidate is given by the Metropolis-Hastings acceptance ratio $\alpha(e,e') = \min(1, \text{Score}(e')/\text{Score}(e))$.

The final step is to describe how the algorithm selects the expression size n. Although the solver comes with an option to specify n, the expression size is typically not known a priori given a specification φ. Intuitively, we run concurrent searches for a range of values for n. Starting with $n = 1$, with some probability p_m (set by default to 0.01), we switch at each step to one of the searches at size $n \pm 1$. If an answer e exists, then the search at size $n = |e|$ is guaranteed to converge.

Consider the earlier example for computing the maximum of two integers. There are 768 integer-valued expressions in the grammar of size six. Thus, the probability of choosing $e = ITE(x \leq 0, y, x)$ as the initial candidate is $1/768$. The subexpression to mutate is chosen uniformly at random, and so the probability of deciding to mutate the boolean condition $x \leq 0$ is $1/6$. Of the 48 boolean conditions in the grammar, $y \leq 0$ may be chosen with probability $1/48$. Thus, the mutation $e' = \mathit{ITE}(0 \leq y, y, x)$ is considered with probability $1/288$. Given a set of concrete examples $\{(-1,4),(-3,-1),(-1,-2),(1,2),(3,1),(6,2)\}$, $\text{Score}(e) = \exp(-2\beta)$, and $\text{Score}(e') = \exp(-3\beta)$, and so e' becomes the new candidate with probability $\exp(-\beta)$. If, on the other hand, $e' = \mathit{ITE}(x \leq y, y, x)$ had been the mutation considered, then $\text{Score}(e') = 1$, and e' would have become the new candidate with probability 1.

Our algorithm differs from that of Schufza et al. [14] in three ways: (1) we do not attempt to optimize the size of the expression while the super-optimizer does so; (2) we synthesize expression graphs rather than straight-line assembly code, and (3) since we do not know the expression size n, we run concurrent searches for different values of n, whereas the super-optimizer can use the size of the input program as an upper bound on program size.

3.7. Learning Functions using Classifiers and Geometry

Another CEGIS-based approach to synthesize expressions is to use machine learning techniques in a more “black-box” manner, learning purely from counterexamples. As in earlier sections, the synthesis engine consists of two parts, a learner and a teacher, who work in turns and communicate with each other. Importantly, in this approach the learner is completely agnostic of the constraints imposed by the SyGuS formula. This characteristic distinguishes the approach of this section from the previous three methods — those methods do not strictly require input-output pairs as counterexamples since they learn from the SyGuS constraints grounded by counterexamples. Additionally, as in the enumerative method, the learner has an inductive bias towards learning simpler expressions (say, expressions of smaller size and smaller constants).

Learning Guarded Linear Expressions

We now describe a learning algorithm that synthesizes guarded linear expressions. These are functions that can be expressed in the logic:

$$\begin{aligned} GLE &:= ite(LP, GLE, GLE) \mid LE \\ LP &:= LE < LE \mid LE \leq LE \mid LE = LE \\ LE &:= c \mid cx \mid LE + LE \end{aligned}$$

where x ranges over a fixed finite set of variables V with domain D (which can be reals, rationals, integers, or natural numbers), and where c ranges over rationals.

We would like the learner to build a guarded linear expression that satisfies a certain set of examples. The most natural notion of an example that can guide the learner in learning a function f with domain D^d and range D is a pair $(\vec{x}, val)$, where $\vec{x} \in D^d$ and $val \in D$, with the intended meaning that $f(\vec{x}) = val$. In other words, we assume the teacher can give counter-examples that precisely point to certain inputs and also give the values of the function on these inputs.

The learner hence learns from a set of examples $E = \{(\vec{x}_i, val_i), i = 1, \ldots n\}$. A guarded linear expression e satisfies such a set of examples E if the function f defined by the expression e maps each $\vec{x}_i$ to val_i.

Building a teacher who can examine a hypothesis function h given by the learner and check whether it satisfies a SyGuS specification is easy, as in all other CEGIS approaches. However, it turns out that giving concrete counterexamples of the form $(\vec{x}, val)$ is not easy for general SyGuS problems. We use certain heuristic techniques that use constraint solvers to build a teacher that can return such counterexamples for some SyGuS problems.

The learner we designed works in two phases, given a set of examples E:

- Phase 1: [Geometry] First, we synthesize a set of linear expressions $LE_1, \dots, LE_k$ such that at least one of them will work for any given example. In other words, for every $(\vec{x}, v) \in E$, there is some i such that LE_i when evaluated on $\vec{x}$ gives v. This problem is essentially to find a small set of planes that include all the given points in $(d+1)$-dimensional space (viewing the space describing the inputs and the output of the function synthesized), and we use a greedy algorithm that uses computational geometry techniques.
- Phase 2: [Classifier] Given the linear expressions from the first phase, which will form the leaves of the expression we synthesize, we then synthesize the guards. This is done using a classification algorithm, which decides a way to assign one of the linear expressions to each input point such that the examples are mapped correctly. We use a decision tree classifier for this, using a fast and scalable machine-learning algorithm for decision trees [36,24].

The first phase, based on geometry, works as follows. Assume that $|E| = n$ where n is large, and assume that we want to find k planes, where k is a small constant, that cover all the examples, viewing the examples as points in $(d+1)$-dimensional space. Given $d+2$ points in $d+1$ dimensions, $(\vec{x}_i, val_i)$, where each $\vec{x}_i = \langle x_i^1, \dots x_i^{d+1} \rangle$, we can *check* if these points are co-planar, by checking whether the following holds:

$$\begin{vmatrix} x_1^1 & x_1^2 & \dots x_1^{d+1} & 1 \\ x_2^1 & x_2^2 & \dots x_2^{d+1} & 1 \\ \vdots & \vdots & \vdots & \vdots \\ x_{d+2}^1 & x_{d+2}^2 & \dots x_{d+2}^{d+1} & 1 \end{vmatrix} = 0$$

This is a standard result in computational geometry [38].

We now build a greedy algorithm for finding a small number of planes that cover all the examples. Note that if k planes are sufficient to cover *all* points, then there must be a plane that covers $\lceil \frac{n}{k} \rceil$ of the points. Furthermore $(d+1)$ points are sufficient to determine a plane. Consequently, we can *randomly* choose $(d+2)$ points that are close to each other, and check for their co-planarity. If they are co-planar, and cover more than a threshold of $\lceil \frac{n}{k} \rceil$ points, we select the plane determined by these points and recurse to find the remaining $(k-1)$ planes from the non-covered points. Otherwise, we select a different set of $(d+2)$ points and iterate.

In the second phase, we label each input in the examples with a plane (from the ones identified in the first phase) that correctly maps that input to its output. We then use a decision-tree classification algorithm [36,24] to classify all inputs according to label planes such that the examples are mapped to the correct planes. Decision-trees can classify points according to a finite set of *numerical attributes*. We choose numerical attributes that are linear combinations of the variables, with bounded integer coefficients. The decision-tree learner then constructs a classification that uses Boolean combinations of formulas of the form $a \leq t$, where a is a numerical attribute and t is a threshold (constant) which it synthesizes. Note that the linear coefficients for the guards are hence enumerated by our tool— the decision tree learner just picks appropriate numerical attributes and synthesizes the thresholds.

The decision-tree learner that we use is a standard state-of-the-art decision tree algorithm, called C5.0 [37,36], and is extremely scalable and has an inductive bias to learn smaller trees. It constructs trees using an algorithm that does no back-tracking, but chooses the best attributes heuristically using *information gain*, calculated using Shannon's entropy measure. We modify these algorithms so that some of the steps (such as *pruning*) which are traditionally performed to reduce overfitting, are removed, since we want a classifier that works precisely on the given sample and cannot tolerate errors.

Building a teacher who counters the learner's hypotheses with concrete counterexamples is hard. First, if the SyGuS specification does not describe a function *exactly* (i.e., if there are many functions that could satisfy the formula), then finding a precise input and output will be hard, or even impossible. (For instance, if the SyGuS specification just demanded $f(x) > x$, then there is no concrete counterexample we can give.) However, typically, there are at least certain inputs where the function is determined by the specification, and the teacher tries to extract such points when the hypothesis fails to satisfy the specification.

The learner and the teacher work together to synthesize the expression; the expression is found when the teacher finds that the current hypothesis of the learner satisfies the SyGuS specification. We implemented this solver, called ALCHEMIST, and its evaluation is discussed in the section on experimental results.

4. Experimental Results

We now present the experimental evaluation of the previously described prototype synthesizers on a benchmark suite of synthesis problems. The current set of benchmarks is limited to synthesis of loop-free functions on integers and bitvectors with no optimality criterion; nevertheless, the benchmarks provide an initial demonstration of the expressiveness of the base formalism and of the relative merits of the individual solution strategies presented earlier. Specifically, in this section we explore three key questions about the benchmarks and the prototype synthesizers.

- **Complexity of the benchmarks.** Our suite includes a range of benchmarks from simple toy problems to non-trivial functions that are difficult to derive by hand. Some of the benchmarks can be solved in a few hundredths of a second, whereas others could not be solved by any of our prototype implementations. In all cases, however, the complexity of the problems derives from the size of the space of possible functions and not from the complexity of checking whether a candidate solution is correct.
- **Relative merits of different solvers.** The use of a standard format allows us to perform the first side-to-side comparison of different approaches to synthesis.

None of the implementations were engineered with high-performance in mind, so the exact solution times are not necessarily representative of the best that can be achieved by a particular approach. However, the order of magnitude of the solution times and the relative complexity of the different approaches on different benchmarks can give us an idea of the relative merits of each of the approaches described earlier.

- **Effect of problem encoding.** For many problems, there are different natural ways to encode the space of desired functions into a grammar, so we are interested in observing the effect of these differences in encoding for the different solvers.

4.1. *Experimental Setup*

The solvers were run on the StarExec platform [33] with a dedicated cluster of 9 nodes, were each node consisted of two 4-core 2.4GHz Intel processors with 256GB RAM and a 1TB hard drive. For the competition, we limited the memory usage of each solver run to be 20GB.

To account for variability and for the constant factors introduced by the prototype nature of the implementations in the first competition, we report the running times on a roughly logarithmic scale. To be precise, we classified the solution times into buckets according to the following solution time values: 1, 3, 10, 30, 100, 300, 1000, 3600 seconds, such that the first bucket is for running times less than 1 second, the second for running times between 1 to 3 seconds, the third for running times between 3 to 10 seconds and so on. The last bucket is for runs that timed out after an hour.

4.2. *Overview of the Results*

We collected 254 synthesis benchmarks from various sources and grouped them into ten main categories: Hacker's Delight [32] problems, integer benchmarks, arrays, Boolean and bit-vector problems, invariant generation problems (with and without unbounded integers), benchmarks using the *let* construct, benchmarks with multiple unknown functions to be synthesized, and benchmarks from the area of vehicle collision control. We now briefly describe the benchmarks problems in each of the category.

The results for the number of benchmarks solved by each solver is summarized in Fig. 2. The top figure shows, for each category, the total number of benchmarks in that category, and the number of benchmarks solved in that category by at least one of the participating solvers. We can see that none of the ICFP benchmarks were solved, and that the arrays and bitvectors categories have lower percentage of solved benchmarks. For the Hacker's Delight category, the two invariant generation categories, the vehicle control category, and the multiple functions category, the percentage of solved instances is relatively higher.

The bottom-left figure shows the number of benchmarks solved per solver per category. We can observe that the enumerative solver solved more benchmarks than the other solvers (a total of 132), and that the stochastic and Sketch-based solvers are not far behind (with totals of 93 and 88 respectively). The Symbolic solver solved a total of 43 benchmarks, whereas the Alchemist solver solved a total of 8 benchmarks.

The bottom-right figure shows the number of benchmarks solved uniquely per solver per category (where a benchmark is said to be *solved uniquely* by a solver if that is the

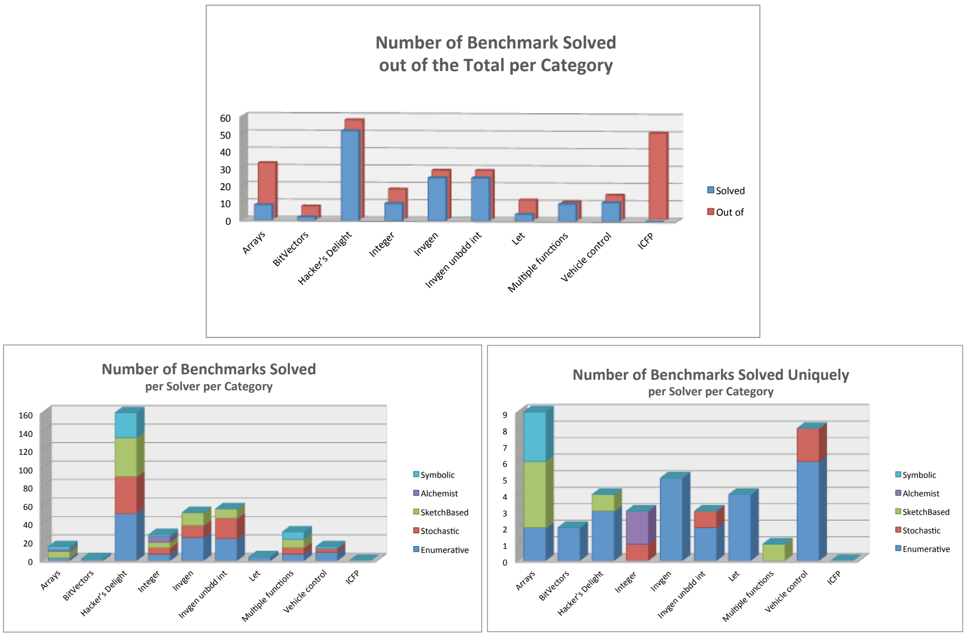


Figure 2. Results on the number of solved benchmarks.

only solver among the five that solved that benchmark). It is notable that each of the solvers were able to solve some benchmarks that none of the other solvers managed to solve.

4.3. Benchmarks and Evaluation per Category

We now present a more detailed description of the benchmark categories and the performance of different solvers on the constituent benchmark problems in each category.

Hacker's Delight Benchmarks

This category includes 57 different benchmarks derived from 20 different bit-manipulation problems from the book *Hacker's Delight* [32]. These bit-vector problems were among the first to be successfully tackled by synthesis technology [19,10,8] and remain an active area of research. For these benchmarks, the goal is to discover clever implementations of bit-vector transformations (colloquially known as bit-twiddling). For most problems, there are three different levels of grammars numbered $d0$, $d1$ and $d5$; level $d0$ involves only the instructions necessary for the implementation, so the synthesizer only needs to discover how to connect them together. Level $d5$, at the other extreme, involves a highly unconstrained grammar, so the synthesizer must discover which operators to use in addition to how to connect them together.

Fig. 3 shows the performance of the solvers on these benchmarks. For the Hacker's Delight benchmarks (hd) we see that the enumerative solver dominates, followed by the stochastic and Sketch-based solvers. The levels $d0$-$d5$ are indicated as part of the name of the benchmark. It is worth mentioning, however, that none of the grammars for these problems required the synthesizer to discover the bit-vector constants involved in the

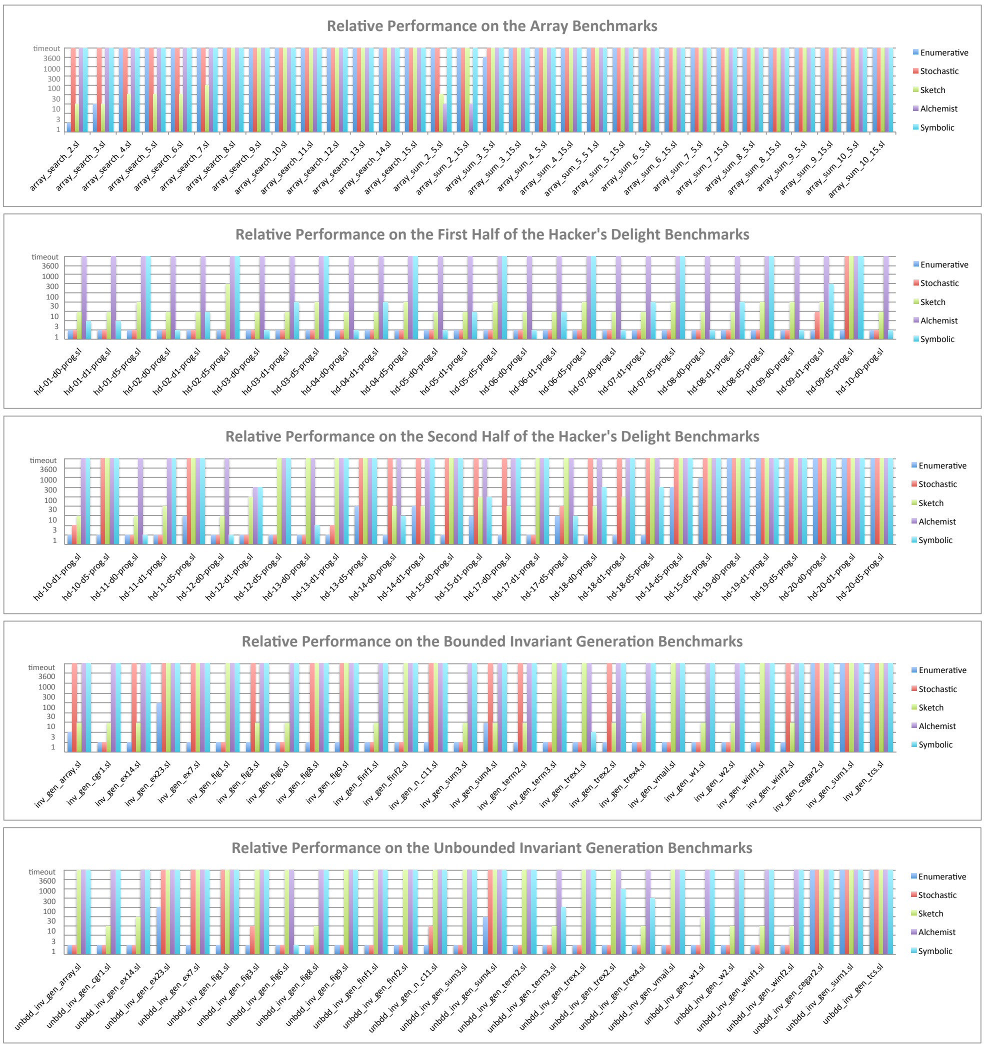


Figure 3. Results on number of benchmarks solved per solver per categories: arrays, Hacker's Delight, invariant generation with bounded integers, and invariant generation with unbounded integers.

efficient implementations. We have some evidence to suggest that the symbolic solver can discover such constants from the full space of 2^{32} possible constants with relatively little additional effort. On the other hand, for many of these problems the magic constants come from a handful of values such as 1, 0, or `0xffffffff`, so it is unnecessary for the enumerative solver to search the space of 2^{32} possible bit-vectors.

Finally, because these benchmarks have different grammars for the same problem, we can observe the effect of using less restrictive grammars as part of the problem description. We can see in the data that all solvers were affected by the encoding of the problem for at least some benchmark; although in some cases, the pruning strategies used by the solvers were able to mitigate the impact of the larger search space.

Array Benchmarks

This benchmark category comprises synthesis problems that involve synthesizing functions over an integer array of a bounded size. Since we currently only allow for integers, Boolean, and bitvector primitive types, the integer arrays in these benchmarks are represented using an ordered sequence of integer variables. One class of the benchmarks is `array-search`, which synthesizes a loop-free function that finds the index of an element in a sorted tuple of size n, for n ranging from 2 to 16. These benchmarks proved to be quite complex, as no solver was able to synthesize this function for $n > 7$. Another class of benchmarks is the `array-sum_m_n` problem, which synthesizes a loop-free function that finds the first two elements of an integer array of size n such that their sum is greater than m.

Fig. 3 shows the relative performance of the five solvers on these benchmarks. For the `array-search` problem, we can see that the Sketch-based solver performs best and is able to scale up to array problems of size $n = 7$, whereas the enumerative and stochastic solvers only scale up to arrays of size $n = 3$ and $n = 2$, respectively. We conjecture that the Sketch-based solver is able to learn the structural constraints of the unknown function using efficient symbolic encoding. For the `array-sum` problem, each of the enumerative, stochastic, and Sketch-based solver solves only a handful of the benchmarks and timed out on most of them.

Invariant generation w/o unbounded integers

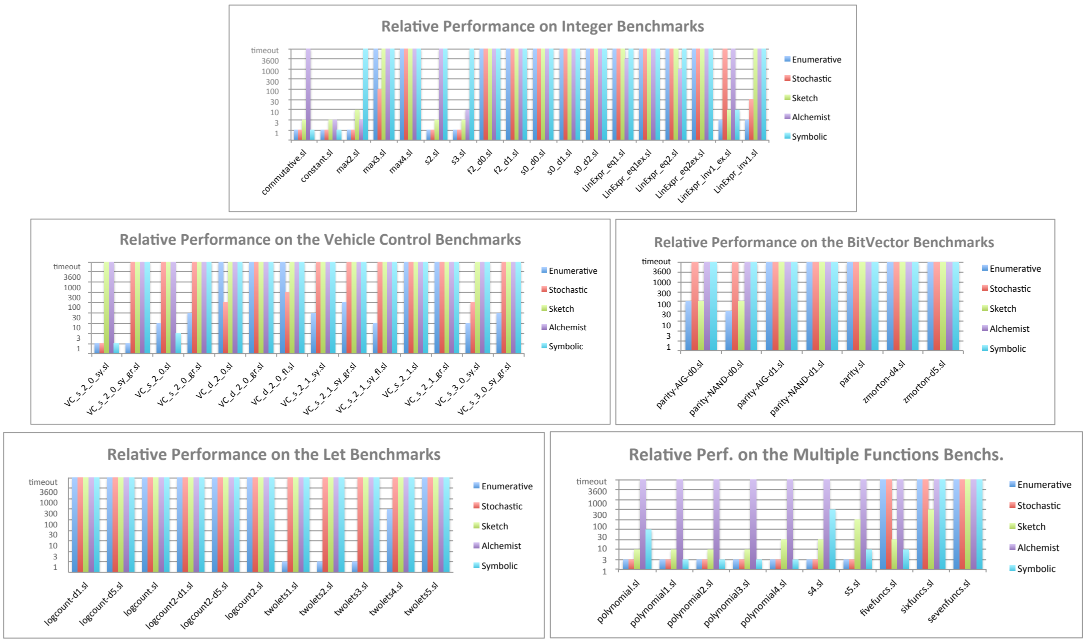
The benchmarks in these two categories are examples from loop invariant synthesis in program verification. They were studied in [27]; some of them are taken from the benchmarks of the competition of software verification (SV-COMP) [28] and others from the literature on invariant synthesis. The first category uses unbounded integers, and the second bounds the integers that can be used. The latter is a modification of the former aiming for synthesis of linear expressions guarded by Boolean combinations of linear guards.

There are 28 benchmarks in each of these categories. Fig. 3 shows that the enumerative solver outperformed the others in both (solving 24 and 25 benchmarks, respectively). For the stochastic solver we can see a large difference between these two sets: in the category where integers were unbounded the stochastic solver solved 22 benchmarks compared to 13 in the bounded integers category. We conjecture that on harder instances of invariant synthesis (with unbounded integers) the stochastic solver would outperform the enumerative.

Integer Arithmetic Benchmarks

These benchmarks are meant to be loosely representative of synthesis problems involving functions with complex branching structures over linear integer arithmetic expressions. For instance, the `max` benchmarks compute the maximum of a tuple of a given size n. The `LinExpr` benchmarks return either the maximum or minimum of a set of variables based on a comparison of the values of two linear expressions over the variables.

Fig. 4 shows the relative performance of the five solvers on these benchmarks. It can be observed that some of these benchmarks were relatively easy for most of the solvers, while others were hard for all the solvers. On the `max` benchmarks, we see that all solvers but one solved `max2`, only the stochastic solver was able to solve `max3` and none was able to solve `max4` in less than an hour.



Relative Performance on Integer Benchmarks

Relative Performance on the Vehicle Control Benchmarks

Relative Performance on the BitVector Benchmarks

Relative Performance on the Let Benchmarks

Relative Perf. on the Multiple Functions Benchs.

Figure 4. Results on number of benchmarks solved per solver per categories: integers, vehcle-control, bit-vector, let and multiple functions.

The results on the `LinExpr` benmarks are quite divergent. Some instances were solved only by Alchemist, while others by all but Alchemist and Stochastic. The Alchemist tool was able to solve the `LinExpr` problems that involved specifying the correctness constraints in a symbolic form, whereas for benchmarks in which the specification was specified using concrete values for input variables, the Alchemist solver did not perform that well. The reason for this seems to be that Alchemist's geometric solver needs "enough" points to find a plane, so when there are too few points, it can fail.

Vehicle Collision Control Benchmarks

These benchmarks simulate a simplified version of a vehicle control problem. Specifically, we consider that there is some number of vehicles approaching an intersection and we would like to synthesize a set of functions (one per vehicle) each mapping the vector of vehicle positions to some positive velocity such that the vehicles avoid collisions when following the control velocities prescribed by these functions. If no such set of velocities is possible from some initial condition, then the functions are expected to return a velocity of zero. The simplified version assumes discrete positions and velocities. Collisions are defined by interpolated trajectories, i.e., taking into consideration vehicle positions at non-integer times as well. [4]

We consider two types of problem instances. In the first type of problem instance, all vehicles travel on the same road, and a crash occurs if two vehicles are within a distance of δ or less, for some integer parameter $\delta \geq 0$. For example, if there are two vehicles which are at positions 1 and 2 at time $t = 0$, $\delta = 0$, and velocities 3 and 1 are chosen,

[4] Occurrence of such collisions can be verified with calculations involving only integers.

then the vehicles collide at time $t = 0.5$, when both occupy position 2.5. The second type of problem instance considers vehicles on different roads. In this case, a collision occurs only when a pair of vehicles occupy the position 0 simultaneously (the position 0 represents the center of the intersection along each road).

The benchmarks' names are of the form $VC\_r\_i\_d\_sy\_fl\_gr$ indicating the parameters chosen as follows: $r \in \{s, d\}$ indicate whether vehicles use the same road or different roads, i is the number of vehicles, d is the δ distance, presence of *sy* indicates that problem symmetries were exploited to synthesize a single function rather than one per vehicle, the presence of the fl suffix indicates the problem was further simplified to provide part of the solution, leaving the solver to synthesize the missing parameters, and finally the presence of *gr* indicates that the grammar was richer than necessary.

The results are summarized in Fig. 4. We can observe that relaxing the grammar (by addition of just one or two operators) made the problem much harder, and the enumerative solver was the only one able to solve such instances. On the other hand, the stochastic solver was the only one able to cope with the problem when the cars were positioned on different roads.

BitVector Benchmarks

The `parity` benchmark computes the parity of a set of Boolean values. The different versions represent different grammars to describe the set of Boolean functions. As with other benchmarks, the enumerative solver was always faster, whereas the symbolic solver failed on every instance. These results show the impact that different encodings of the same space of functions can have on the solution time for both of the solution strategies that succeeded. Unlike the `hd` benchmarks where the different grammars for a given benchmark were strict subsets of each other, in this case the encodings `AIG` and `NAND` correspond to different representations of the same space of functions.

The `Morton` benchmarks, which involve the synthesis of a function to compute Morton numbers, are intended as challenge problems, and could not be solved by any of the solvers.

Let Benchmarks

The benchmarks in this category make use of the *let* construct, explained in Section 2. The let construct was supported only by the enumerative and stochastic solvers. The *logcount* benchmarks encode the problem of synthesizing a program that counts the number of ones in a bit-vector in $O(\log(n))$ steps and were not solved by any of the solvers. The *twolets* are toy examples using two lets, with varying degrees of flexibility in the grammars. The enumerative solver was able to tackle all but the most difficult of these. The stochastic solver could not solve any of them. The treatment of *let* in the stochastic solver involves unrolling of the expression to account for the scope of the let-variables that may be used. We conjecture this makes these problems harder for the stochastic solver to manage.

Multiple Functions Benchmarks

The benchmarks in this category require the solver to synthesize multiple functions. The constraints in this set relate a subset of the functions (to be synthesized) by comparing one function to the other or to the addition/subtraction thereof. Most of the benchmarks were easily solved by all solvers supporting multiple functions. On the harder instances, which included synthesizing five or more functions, the symbolic and Sketch-based solvers outperform the other solvers. We conjecture that this phenomenon might be because of the succinct encoding of the huge space of possible function choices in the symbolic approaches [34].

ICFP Benchmarks

The benchmarks in this category were taken from the 2013 ICFP Programming Competition [35], which was organized on the theme of program synthesis. The competition rules specified that the participants had to find secret programs in a language called λ_{BV}, a small language consisting of functions over 64-bit vectors. The language consisted of the following primitive bitwise operators: negation (`not`), shift left by one bit (`shl1`), shift right by one (`shr1`), four (`shr4`), or sixteen bits (`shr16`), addition (`plus`), conjunction (`and`), disjunction (`or`), and exclusive or (`xor`). The language also consisted of a conditional operator (`if0`) that would check if the first argument was 0 for branching to the corresponding expression, and a fold operator to loop over each byte of the bitvector from right to left. The constant bitvectors were only 0 and 1.

Since we can not provide the secret bitvector function in the specification, we created three categories of the problems where we generated 10, 100, and 1000 random input-output 64-bit bitvectors that approximated the functional behavior of the secret function. For the grammar of the unknown functions, we provided the complete λ_{BV} grammar except the `fold` operator. These benchmarks proved to be the most challenging for the solvers, and none of the solvers could solve any of the 50 benchmark problems. We hypothesize the size of secret functions (> 15) and the unconstrained grammar for the unknown function caused the search space to become inhibitively large. In the ICFP competition, some of these benchmarks were solved by the participants using enormously large compute clusters. We believe scaling the solvers to handle these benchmarks would lead to newer insights in program synthesis research.

4.4. *Expression Sizes*

On most of the benchmarks there was not much variation in the size the expressions produced by the different solvers. Since the enumerative solver by definition always produces the minimal result, we can conclude that most of the solved instances expressions were in the same order of magnitude as their optimal solution.

Most solved benchmarks were solved using expression sizes less than 40, where the size of an expression is determined by the number of nodes in its parse tree. The biggest expression, of size 1056, was produced for `LinExpr_eq1.sl` by Alchemist, who was the only solver which managed to tackle this benchmark. The second biggest expression, of size 425 was for `VC22_c2.sl` by Stochastic, which again was the only one able to solve this instance. Other benchmarks, with interesting divergence in size of expressions, are listed in Table 4.

4.5. *The First Syntax-Guided Synthesis Competition*

The first Syntax-Guided Synthesis Competition (SYGUS-COMP 2014) was organized as a satellite event of SYNT/CAV 2014, and as part of the FLoC Olympic Games 2014 at

Benchmark	Enum.	Stoch.	Sket.	Alch.	Symb.	Fastest	Time
array_sum_2_5.sl	-	-	31	139	-	Alchemist	9.012
hd-17-d5-prog.sl	8	73	-	-	9	Enumerative	5.653
LinExpr_eq1.sl	-	-	-	1056	-	Alchemist	1128.870
LinExpr_eq2.sl	-	-	-	180	-	Alchemist	303.780
max4.sl	-	39	-	-	-	Stochastic	95.683
VC22_f2.sl	14	118	-	-	-	Enumerative	5.475
VC22_c2.sl	-	425	-	-	-	Stochastic	128.635
VC22_a_g.sl	6	-	-	-	-	Enumerative	0.016
fivefuncs.sl	-	-	35	-	43	Symbolic	2.540
sixfuncs.sl	-	-	122	-	-	Sketch-based	233.447

Table 4. A sample of benchmarks with interesting variations in expression sizes, and the time it took for the fastest solver to complete (in seconds).

Vienna Summer of Logic (VSL). The competition was organized as part of the NSF Expeditions in Computing project ExCAPE by Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. The final ranking in SYGUS-COMP 2014 was performed in the following way. The solvers were ranked relative to each other according to the following parameters (in order of importance): correctness, solving time, and expression size. For each benchmark, the best solver was assigned a score of 1, the next best solver was assigned a score of 2 and so on. If there was a tie between two solvers, the solvers received the same score. For each benchmark category, these scores were averaged for each solver. This way we received a relative ranking of each solver for each benchmark category. The final ranking of the solvers was obtained by adding the relative ranks of the solvers for each category. With this ranking metric, the Enumerative solver came first with the score of 12.03, the stochastic solver came second with a score of 22.86, and the Sketch-based solver came third with a score of 25.18.

For the competition we took into account the absolute solving times rather than the buckets as presented here. Ranking according to the buckets would have result in the same set of overall ranking of the solvers. The chosen ranking metric favors solvers that support many language features, and solvers that can solve many of the simpler problems quickly (as opposed to solvers that can solve some of the harder problems but take more time to solve simpler problems). We might revisit the ranking metric in future competition to address some of these issues.

5. Conclusions

Aimed at formulating the core computational problem common to many recent tools for program synthesis in a canonical and logical manner, we have formalized the problem of syntax-guided synthesis. We implemented five prototype tools for the alternative approaches discussed in Section 3. Around 500 benchmarks were collected as part of the first SyGuS-Comp competition held in July 2014. The first competition restricted the background theories to that of quantifier-free bit-vector arithmetic and linear integer arithmetic. The relative performance of the prototype solvers on 250 benchmarks was reported and analyzed. We hope that this effort, namely the SyGuS-LIB format, the publicly available prototypes, and the initiation of an annual competition of solvers for SyGuS, will boost research and development of improved solution strategies and generic solvers for SyGuS.

Acknowledgements

This research is supported by the NSF Expeditions in Computing project ExCAPE (award CCF 1138996).

References

[1] R. Alur, R. Bodík, G. Juniwal, M. Martin, M. Raghothaman, S. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, “Syntax-Guided Synthesis,” in *FMCAD*, 2013, pp. 1–17.

[2] M. Raghothaman and A. Udupa. Language to Specify Syntax-Guided Synthesis Problems. In *http://arxiv.org/pdf/1405.5590.pdf*, May, 2014.

[3] SyGuS Org. http://www.sygus.org.

[4] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, “Satisfiability modulo theories,” in *Handbook of Satisfiability*, 2009, vol. 4, ch. 8.

[5] L. M. de Moura and N. Bjørner, “Satisfiability modulo theories: Introduction and applications,” *Commun. ACM*, vol. 54, no. 9, 2011.

[6] Z. Manna and R. Waldinger, “A deductive approach to program synthesis,” *ACM TOPLAS*, vol. 2, no. 1, pp. 90–121, 1980.

[7] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioglu, “Programming by sketching for bit-streaming programs,” in *PLDI*, 2005.

[8] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, “Synthesis of loop-free programs,” *SIGPLAN Not.*, vol. 46, pp. 62–73, June 2011.

[9] S. Gulwani, W. R. Harris, and R. Singh, “Spreadsheet data manipulation using examples,” *Commun. ACM*, vol. 55, no. 8, pp. 97–105, 2012.

[10] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, “Oracle-guided component-based program synthesis,” in *ICSE*, 2010, pp. 215–224.

[11] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, “TRANSIT: Specifying Protocols with Concolic Snippets,” in *PLDI*, 2013, pp. 287–296.

[12] S. Srivastava, S. Gulwani, and J. S. Foster, “From program verification to program synthesis,” in *POPL*, 2010, pp. 313–326.

[13] V. Kuncak, M. Mayer, R. Piskac, and P. Suter, “Software synthesis procedures,” *Commun. ACM*, vol. 55, no. 2, pp. 103–111, 2012.

[14] E. Schkufza, R. Sharma, and A. Aiken, “Stochastic superoptimization,” in *ASPLOS*, 2013, pp. 305–316.

[15] M. Colón, S. Sankaranarayanan, and H. Sipma, “Linear invariant generation using non-linear constraint solving,” in *CAV*, 2003, pp. 420–432.

[16] A. Rybalchenko, “Constraint solving for program verification: Theory and practice by example,” in *CAV*, 2010, pp. 57–71.

[17] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori, “A data driven approach for algebraic loop invariants,” in *ESOP*, 2013, pp. 574–592.

[18] N. Bjørner, K. L. McMillan, and A. Rybalchenko, “On solving universally quantified Horn clauses,” in *SAS*, 2013, pp. 105–125.

[19] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. Saraswat, “Combinatorial sketching for finite programs,” in *ASPLOS*, 2006.

[20] E. M. Gold, “Language identification in the limit,” *Information and Control*, vol. 10, no. 5, pp. 447–474, 1967.

[21] P. D. Summers, “A methodology for LISP program construction from examples,” *J. ACM*, vol. 24, no. 1, pp. 161–175, 1977.

[22] E. Y. Shapiro, *Algorithmic Program Debugging*. Cambridge, MA, USA: MIT Press, 1983.

[23] D. Angluin and C. H. Smith, "Inductive inference: Theory and methods," *ACM Computing Surveys*, vol. 15, pp. 237–269, Sep. 1983.

[24] T. M. Mitchell, *Machine Learning*. McGraw-Hill, 1997.

[25] S. A. Seshia, "Sciduction: Combining induction, deduction, and structure for verification and synthesis," in *DAC*, 2012, pp. 356–365.

[26] D. Angluin, "Queries and concept learning," *Machine Learning*, vol. 2, pp. 319–342, 1988.

[27] P. Garg, C. Løding, P. Madhusudan, and D. Neider, "ICE, A Robust Framework for Learning Invariants," in *CAV*, 2014 pp. 69–87.

[28] https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp13/loops/

[29] A. Solar-Lezama, "Program synthesis by sketching," Ph.D. dissertation, University of California, Berkeley, 2008.

[30] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[31] B. McKenzie, "Generating strings at random from a context free grammar," 1997.

[32] H. S. Warren, *Hacker's Delight*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[33] A. Stump,G. Sutcliffe,C. Tinelli, "StarExec: A Cross-Community Infrastructure for Logic Solving" in *IJCAR*, 2014 pp. 367–373.

[34] R. Singh, R. Singh, Z. Xu, R. Krosnick, A. Solar-Lezama, "Modular Synthesis of Sketches Using Models" in *VMCAI*, 2014 pp. 395–414.

[35] T. Akiba, K. Imajo, H. Iwami, Y. Iwata, T. Kataoka, N. Takahashi, M. Mmoskal, N. Swamy, "Calibrating Research in Program Synthesis Using 72,000 Hours of Programmer Time" in *MSR-TR*, December 2013.

[36] J. Ross Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, pp. 81–106, 1986.

[37] J. Ross Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[38] Eric W. Weisstein, "Coplanar," *http://mathworld.wolfram.com/Coplanar.html*, December 2014.