The SemGuS Toolkit

Keith J.C. Johnson[1], Andrew Reynolds[2], Thomas Reps[1], and Loris D'Antoni[1]

[1] University of Wisconsin–Madison
[2] University of Iowa

Abstract. Semantics-Guided Synthesis (SemGuS) is a programmable framework for defining synthesis problems in a domain- and solver-agnostic way. This paper presents the standardized SemGuS format, together with an open-source toolkit that provides a parser, a verifier, and enumerative SemGuS solvers. The paper also describes an initial set of SemGuS benchmarks, which form the basis for comparing SemGuS solvers, and presents an evaluation of the baseline enumerative solvers.

1 Introduction

The field of program synthesis aims to create tools that can automatically create a program from a specification of desired behavior. Synthesis holds the promise of easing the burden on programmers (e.g., by finding solutions to tricky special cases automatically), and allowing non-programmers to create programs merely by indicating the outcome that they want the program to produce.

While program synthesis has seen successes in many industrial applications [23,9], these successes have typically been achieved using domain-specific synthesizers that take advantage of the structure of the specific domain.

To apply synthesis beyond specific domains, synthesis frameworks and tools should allow one to customize the search space and specifications of a synthesis problem in a programmable way that is agnostic of a specific domain or synthesis solver. To address the problem of making synthesis "programmable", Kim et al. [15] proposed the SemGuS framework, which enables one to specify synthesis problems in a solver-agnostic and domain-agnostic way [7].

The SemGuS framework allows one to specify an arbitrary synthesis problem by defining a programming language via (i) a grammar (the syntax), and (ii) a set of Constrained Horn Clauses (CHCs) (the semantics). Once one has described the language, one can define synthesis problems over that language by providing a specification as a formula. Solving the synthesis problem means finding a program in the language that satisfies the specification. Building solvers for general SemGuS problems can be difficult due to the framework's flexibility [7].

This paper presents the SemGuS toolkit, which provides an open-source implementation of the components needed for researchers to get started building SemGuS solvers. The toolkit consists of the following components.

SemGuS Format 1.0: The first standardized format for SemGuS, which is built on top of the SMT-LIB and SyGuS formats [3,21], thus making it expressible,

extensible, modular, and easy to integrate with existing constraint solvers (e.g., to build SemGuS verifiers). We provide an open-source parser (Section 2).

Baseline Verifier and Solvers: The flexibility of SemGuS makes verifying whether a term is a solution to a SemGuS problem undecidable. Furthermore, because the semantics of the user-provided programming language is expressed declaratively using CHCs, it is even challenging to efficiently execute programs in the language. Our implementation provides a compiler that, given a term t in the user-specified language, can extract efficiently executable semantics for t from the declarative one provided by the user, as well as an incomplete SMT-based verifier that can construct constraints for checking whether t matches a specification φ. We also provide implementations of top-down and bottom-up example-based enumerative solvers that are integrated with these verifiers and can thus produce solutions to SemGuS problems (Section 3).

Benchmarks: We provide 431 SemGuS benchmarks from different domains. Our solvers can only solve 161/431 benchmarks, and we hope this toolkit will energize the community to build solvers for the remaining challenging problems and to provide additional benchmarks (Section 4).

2 The SemGuS Format 1.0

We refer the reader to the original SemGuS paper [15] for a more formal definition of the SemGuS framework, but in this section we show how each component is expressed in our proposed standard format. The SemGuS parser (https://github.com/semgus-git/Semgus-Parser) can translate the textual SemGuS format into two intermediate representations: a JSON format and a declarative S-expression format, which is then used by solvers and other tools.

Figures 1 and 2 give an example specification of a SemGuS problem, which we describe in detail in this section. In this example, the goal is to synthesize an imperative program (with loops) that multiplies two numbers through iterative addition. We choose this example because it illustrates how SemGuS can describe synthesis problems involving complex programming constructs and is thus strictly more expressive than limited synthesis frameworks, such as SyGuS [1].

Term Universe. SemGuS problems define a universe of terms with a modified SMT datatype declaration using the command `declare-term-types` (lines 1–8 of Figure 1). This command defines the syntax of the programming language over which one can specify synthesis problems. The term universe L is intentionally separated from the sub-universe (defined by a grammar) from which the answer is to be synthesized, and from the constraints on the answer (Figure 2). The user defines L and its semantics once and for all, and can reuse those definitions for different synthesis problems. This separation enables both *(i)* building specialized SemGuS solvers for important languages (e.g., L = SQL), and *(ii)* instantiating more restricted synthesis problems by confining the search space to just the terms generated by a grammar.

```
1  ;; Nonterminals
2  (declare-term-types ((F 0) (S 0) (E 0) (B 0))
3  ;; Term Universe (i.e., language syntax)
4  ((($function S E))                                  ;; F
5   (($x<- E) ($y<- E) ($r<- E)
6    ($noop) ($seq S S) ($while B S))                  ;; S
7   (($r) ($0) ($1) ($x) ($y) ($+ E E) ($- E E))      ;; E
8   (($< E E))))                                      ;; B
9  ;; Constrained Horn Clauses (i.e., language semantics)
10 (define-funs-rec
11  ;; Types of semantic relations
12  ((F.Sem ((t F) (x Int) (y Int) (ret Int)) Bool)
13   (S.Sem ((t S) (xi Int) (yi Int) (ri Int) (xo Int)
14           (yo Int) (ro Int)) Bool)
15   (E.Sem ((t E) (xi Int) (yi Int) (ri Int) (out Int)) Bool)
16   (B.Sem ((t B) (xi Int) (yi Int) (ri Int) (out Bool)) Bool))

17  ;; CHCs defining semantic relations
18  (;; Semantics of functions
19   (! (match t (...)) :input (x y) :output (ret))
20   ;; Semantics of statements
21   (! (match t
22       ...more S productions...
23       (($noop)          ;; Noop statement
24        (and (= xi xo) (= yi yo) (= ri ro)))
25       (($seq t1 t2)     ;; Sequential composition
26        (exists ((x1 Int) (y1 Int) (r1 Int))
27          (and (S.Sem t1 xi yi ri x1 y1 r1)
28               (S.Sem t2 x1 y1 r1 xo yo ro))))
29       (($while tb ts)  ;; While statement
30        (exists ((b Bool) (x1 Int) (y1 Int) (r1 Int))
31          (and (B.Sem tb xi yi ri b) ;; While-true
32               (= b true)
33               (S.Sem ts xi yi ri x1 y1 r1)
34               (S.Sem t x1 y1 r1 xo yo ro)))
35        (exists ((b Bool))
36          (and (B.Sem tb xi yi ri b) ;; While-false
37               (= b false)
38               (= xo xi) (= yo yi) (= ro ri))))))
39   :input (xi yi ri) :output (xo yo ro))
40   ;; Semantics of integer expressions
41   (! (match t (...)) :input (xi yi ri) :output (out))
42   ;; Semantics of Boolean expressions
43   (! (match t (...)) :input (xi yi ri) :output (out))))
```

Fig. 1. Definition of a programming language (i.e., a set of programs) in the SemGuS format. The syntax of terms is given in lines 1–8, and their semantics is given in lines 9–43. The gray text denotes parts that have been omitted for brevity.

Semantics as CHCs. The semantics of our term language is given by the SMT-LIB command `define-funs-rec` (lines 9–43). In a nutshell, this command defines a set of Constrained Horn Clauses (CHCs) inductively over terms in the universe. A CHC is a first-order formula of the form:

$$\forall \bar{x}_1, \ldots, \bar{x}_n, \bar{x}.\ \phi \wedge R_1(\bar{x}_1) \wedge \cdots \wedge R_n(\bar{x}_n) \Rightarrow H(\bar{x})$$

where $R_1, \ldots, R_n$ and H are uninterpreted relations, $\bar{x}_1, \ldots, \bar{x}_n$ and $\bar{x}$ are (vectors/tuples of) variables, and ϕ is a quantifier-free constraint over the variables within some first-order theory. In the specification, one provides the names and types of the semantic relations used in the semantic definitions (lines 11–16), and then CHCs that define such relations (lines 17–43). To better align with the fact that CHCs are used to define the semantics of programs inductively (i.e., as an interpreter), in the SemGuS format we encode CHCs as a set of mutually-recursive SMT functions, taking the term to be evaluated, input variables, and output variables as arguments, and returning a Boolean. A function is provided for every non-terminal, and `match` statements are used to dispatch on the term constructors for which one is defining the semantics. The match statement must match on all productions for the given term type (i.e., the corresponding non-terminal). The match statement can also be annotated with which variables are inputs and outputs in the specific semantics (note that some semantics, e.g., a term-rewriting system, do not necessarily have inputs and outputs). Each match on a production starts with an optional `exists` block, which specifies auxiliary variables, followed by the CHC body as a conjunction. Some productions, such as the *while* production in Figure 1 (lines 29–38), have two associated CHCs. For example, the `While-false` CHC can be logically written as

$$\frac{B.Sem(tb, xi, yi, ri, b) \quad b = false \wedge xo = xi \wedge yo = yi \wedge ro = ri}{S.Sem((\texttt{\$while}\ tb\ ts), xi, yi, ri, xo, yo, ro)}\ \texttt{While-false}$$

The signature at the bottom of the CHC—i.e., the particular variables names used in this relation instance—is the one defined in Figure 1 (line 13).

As discussed in the original SemGuS paper [15], many synthesizers have achieved scalability by exploiting alternative semantics that either underapproximate the actual semantics of the programming language (to speed up evaluation and enable constraint solving) or overapproximate it (which sometimes makes it possible to prune the search space of programs). While such previous work "hardcodes" and takes advantage of such semantics in the solver itself, SemGuS allows one to write such semantics directly in the SemGuS file. In fact, there is no limit on how many semantic relations one can define in a SemGuS file. For example, one might define a semantic relation that associates costs to programs (but does not evaluate programs) and a semantic relation that captures program evaluation. The specification can then require finding a program that (i) performs a computation correctly, and (ii) has a cost that is less than a specific constant. The ability of SemGuS to describe multiple semantics enables reusable solving techniques and interoperability between solvers.

It should be noted that when one defines multiple semantics, the burden of showing that they are properly related (e.g., that an abstract semantics is related to the concrete one by a Galois connection [5]) is in the hands of the user. Doing so automatically is a research direction enabled by the SemGuS format.

`synth-fun` Command. SemGuS uses the same syntax as in SyGuS to declare what type of term we are interested in synthesizing (Figure 2). Unlike SyGuS, a solution to a SemGuS problem is a term in the provided syntax, as opposed to a function in an SMT theory. For instance, the command `(synth-fun mul () F)` in Figure 2 (line 2) asks for a term named `mul`, rooted at the non-terminal `F`. This command can optionally take a grammar (the second argument) to further restrict the search space, using the same format for grammars as in the SyGuS format [21]. For example, to synthesize programs that have the fewest number of while-loops [12], one might first solve the SemGuS problem discussed in this section and obtain a program with one loop, and then create a new SemGuS problem where the grammar is restricted to disallow loops. The two problems will share the same language definition despite having different grammars.

Specification. Specification constraints for SemGuS problems are stated using SMT expressions involving the root CHC for the term to be synthesized. The typical form for input/output examples is shown in Figure 2 (lines 4–9). Note that in SemGuS, constraints are specified as relations and not functions (as in SyGuS). Relations allow modeling nondeterminism or nonterminating semantics—e.g., one can state that, for the specific input pair (5,3), the answer is a positive value if the program terminates: `(constraint (forall ((x Int)) (=> (F.Sem mul 5 3 x) (>= x 0))))`.

```
1  ;; Function to synthesize
2  (synth-fun mul () F)
3  ;; Constraints for examples
4  (constraint (F.Sem mul 0 0 0))
5  (constraint (F.Sem mul 1 1 1))
6  (constraint (F.Sem mul 2 2 4))
7  (constraint (F.Sem mul 3 3 9))
8  (constraint (F.Sem mul 5 3 15))
9  (constraint (F.Sem mul 3 4 12))
10 ;; Perform synthesis
11 (check-synth)
```

Fig. 2. Constraints for a few example input/output pairs, used to synthesize a function `mul` that behaves like multiplication.

Synthesis Command. The `check-synth` command instructs the solver to solve the problem and produce an SMT term. The following term is a solution to the example presented in this section.

```
1 ((define-fun mul () F ($function
2         ($while ($< $0 $y)                ;; while (0<y)
3             ($seq ($y<- ($- $y $1))       ;;      y <- y-1
4                   ($r<- ($+ $r $x))))     ;;      r <- r+x
5         $r)))                             ;; return r
```

Relationship between SemGuS and SyGuS. Every SyGuS problem can be automatically converted to an equivalent SemGuS problem, and our parser implements this transformation. The only technical detail of interest is that SyGuS synthesizes *function SMT terms*, whereas SemGuS synthesizes terms in

a term universe that is interpreted using a *relational semantics*. For example, if the predicate φ of the SyGuS specification contains invocations of the function g to be synthesized, e.g., $\varphi(g(i_1), \ldots, g(i_n))$, we can create the new SemGuS specification as $\exists o_1, \ldots, o_n.\ \varphi(o_1, \ldots, o_n) \wedge \text{Sem}_G(g, i_1, o_1) \wedge \ldots \wedge \text{Sem}_G(g, i_n, o_n)$.

Because SemGuS is more expressive than SyGuS, not every SemGuS problem can be converted to an equivalent SyGuS problem. In general, it is undecidable to check when such a translation is possible, because SemGuS is Turing complete. We have implemented a sound (but incomplete) translation of a limited fragment of statically detectable SemGuS problems into SyGuS. The fragment essentially captures when the SyGuS-to-SemGuS translation can be inverted.

3 A Baseline SemGuS Solver

In this section, we present KS2, a toolkit for researchers to build SemGuS solvers. KS2 implements techniques for (efficiently) verifying whether a candidate solution meets the specification (Section 3.1). KS2 also contains implementations of bottom-up and top-down enumerative synthesizers (Section 3.2). KS2 is written in Common Lisp, which makes it easy to compile code generated at synthesis-time for speeding up evaluation of candidate solutions. In addition, KS2 is implemented modularly, so new solvers and features can be easily added as plugins.

3.1 Verifying Candidate Solutions

When building synthesizers, one wants two types of verifiers: one that can quickly tell if a candidate solution is correct on a finite set of input examples E, and one that can (less quickly) tell if a solution is correct on all inputs, and thus satisfies a logical specification. When the latter verifier finds a violation of the specification, it will typically produce a new input example e that can be added to the set E to restart synthesis with a fresh set of examples. These two verifiers together form the basis of the counterexample-guided synthesis algorithm. For SemGuS, building either of these verifiers is generally undecidable as one may have to deal with an arbitrarily powerful programming language.

In this section, we present two sound (but incomplete) implementations of such verifiers. These implementations are not the only verifier implementations that can be built for SemGuS, but just two that were successful in meeting our needs. Building other verifier implementations based on other technologies, such as bounded model checkers, symbolic execution, and logic programming, is an interesting future research direction.

Building Executable Semantics from CHCs. To tell quickly whether a candidate program is correct on a given input, one needs to "run" the program on the input according to the semantics. To do so efficiently is nontrivial because the semantics of a candidate program is expressed declaratively using CHCs. KS2 first "operationalizes" the semantics given by CHCs into executable blocks, which are then compiled. In general, not all CHCs can be transformed into executable code (for example, non-deterministic CHCs that can map one input

to different outputs); therefore, KS2 supports only a fragment of CHCs that is practically useful (all benchmarks discussed in Section 4 fall into this fragment).

We illustrate the compilation to native code using the following (recursive) CHC corresponding to the `While-true` case in Figure 1:

$$\text{Sem}_S(t(t_b, t_s), i, o) \impliedby \text{Sem}_B(t_b, i, b) \wedge b \wedge \text{Sem}_S(t_s, i, o') \wedge \text{Sem}_S(t, o', o)$$

KS2 requires each position in each relation to be annotated as an input or output variable.[3] In the example, the first position of Sem_S and Sem_B is the term being executed, which is always assumed to be an input; the second position is the input on which the term should be executed, and the last position is the output. To operationalize the CHC, KS2 performs the following steps to identify an evaluation order: it analyzes each relation instance in the body of the CHC, and performs a dataflow analysis to determine an order in which the blocks can be executed. This step is done by building a dataflow graph and then performing a topological sort to identify an order in which each relation can be computed. In the given example, one possible order is to first evaluate $\text{Sem}_B(t_b, i, b)$ because i is readily available, then determine whether b is true, then evaluate $\text{Sem}_S(t_s, i, o')$, and finally evaluate $\text{Sem}_S(t, o', o)$ (o' depends on one of the previous relations). Our implementation of this transformation has some basic requirements. First, no two relations can output the same variable—otherwise one cannot resolve which instance to use. Second, every input variable i' to a relation is either the output of another relation or appears in the first-order formula of the CHC in the form $i' = f(\cdot)$—i.e., the value of i' can be computed without having to call an SMT solver.

At this point, we generate code for each block. Child CHCs turn into function calls, guards into conditional statements, and value productions into assignments. This generated code is then compiled and turned into an executable function that implements the CHC's semantics. To execute a program on an input/output example, the top-level semantic function is called with the input state and the child's semantic functions, and the program returns the output state. This output state can be checked against the output example.

This implementation of an efficiently executable semantics is one of the main contributions in KS2. Identifying additional ways to compile the logical semantics into an efficiently-executable one is an interesting research direction that can benefit from techniques in compiler design and logic programming.

A Simple Incomplete Verifier for Logical Specifications. The declarative nature of SemGuS enables a simple way of building a verifier that can check a program against a specification and return a counterexample. As we argued, verification for SemGuS is undecidable, but the declarative nature of SemGuS allows us to build a simple, but incomplete, procedure for verifying some candidates in SemGuS solutions. Given a concrete term t (i.e., the program we are

[3] Automatically inferring such annotations ("mode inference") is a classical analysis problem in logic programming [6, §10.2.2]. Automatically supplying annotations is an interesting research direction for SemGuS.

trying to verify), our verifier performs a pre-order traversal of t and emits a potentially recursive SMT function for each node that corresponds to the CHC (or CHCs) for that node. Because t is a concrete term, each child term in the CHC can also be replaced by the concrete function implementing it. For example, the program `($+ $x $1)` would be verified by emitting three SMT functions for `$+`, `$x`, and `$1`. The function for `$+` will call the ones for `$x` and `$1` to perform the evaluation. Operators like `$while` require recursive function calls. The specification can then be used to define constraints over the root node and verified with an SMT solver. In the case of recursive semantics, the SMT functions will potentially be mutually recursive, thus relying on undecidable theories for which current SMT solvers struggle in practice. This verifier, while incomplete, is "good enough" for many of our current benchmarks, and extending SMT solvers or our verifiers to better handle such cases is a challenging research question.

3.2 Baseline Enumerative solvers

KS2 implements standard basic top-down and bottom-up enumeration algorithms as described in the literature [11]. Because we have a logical verifier that enables counterexample-guided inductive synthesis (CEGIS), our enumeration algorithms only check correctness on a set of examples.

For top-down enumeration, candidate programs are enumerated using a priority queue of potentially partial programs (i.e., with holes). At each iteration a program is extracted from the queue: if it has no holes, it is verified against the examples; otherwise, all the programs that can be obtained by expanding the leftmost hole with all possible child productions are added to the queue. The standard optimization for top-down enumerators is to check partial programs against the specification and prune them if possible. Existing optimizations are domain-specific and identifying ways to extend them to SemGuS problems is an open research question, which we hope this toolkit will help researchers work on.

For bottom-up enumeration, subterms of increasing size (or height) are enumerated and added to a program bank where they are grouped by size (or height). Enumeration of programs of a certain size or height happens lazily; they are verified and pushed into the bank of programs one at a time. A typical optimization used in a bottom-up enumeration is to use some form of equivalence-checking to deduplicate enumerated programs with the same behavior. One popular technique, observational equivalence, executes each enumerated program on the input example states and prunes a program if a previously enumerated program returns the same output state. However, because SemGuS supports imperative semantics, the possible input states for a sub-program are not necessarily the same as the top-level-program's input states (i.e., variable values change throughout the program execution), and thus there is not an easy way to perform an observational-equivalence check. The development of an appropriate pruning technique for a bottom-up SemGuS enumerator is an open research question, which we hope this toolkit will help researchers work on, for example by building on approaches such as equality saturation [25] and lifting interpretation to sets of programs [17].

3.3 Extensibility

KS2 can be extended by instantiating various interfaces with modules. For example, one might want to add a module that implements a technique for pruning enumerated programs with the bottom-up enumeration. To add this technique, the module would implement the `add-to-bank` interface, which is responsible for adding freshly enumerated programs to the bank of enumerated programs, and simply decline to add programs that the module can prune. In code, this implementation might look like:

```
1 (defmethod add-to-bank :around ((ext prune) bank prog metric)
2   "Adds the program PROG to BANK unless it should be pruned"
3   (unless (%should-prune prog) (call-next-method)))
```

where `prune` is the module class and `%should-prune` implements the predicate for whether or not a program should be pruned. At this time, among others, we have interfaces for adding solvers, adding verifiers, and inspecting and updating the SemGuS problem. We will continue to add more interfaces as the need arises; the most up-to-date documentation is available with KS2 and its supporting libraries.

Outside of KS2, the SemGuS Parser is available as a standalone tool for parsing SemGuS problems into JSON, as well as a .NET library for direct integration into solvers. We expect these parsing tools to lower the barrier to entry for building new SemGuS tooling.

4 Benchmarks and Performance of Baseline Solvers

We present an initial set of SemGuS benchmarks and evaluate the performance of our baseline solvers on such benchmarks.

Benchmarks. The ability of SemGuS to represent synthesis problems from disparate domains in the same solver-agnostic format is one of its key distinguishing features. We have created 431 SemGuS benchmarks, consisting of synthesis problems from a variety of domains.

Sample domains: 17 benchmarks of easy synthesis problems (10 for imperative programs with loops, 3 for SMT datatypes, and 4 integer-arithmetic benchmarks). These benchmarks are designed to help researchers build SemGuS solvers and are basic test of a solver's support of various features of the SemGuS format. They contain between 1 and 6 input/output examples each.

Regular expressions: 72 benchmarks for synthesizing regular expressions, which include problems from the original SemGuS paper [15], from the tool AlphaRegex [16], and CSV formatting problems. These benchmarks have between 2 and 244 input/output examples each. Benchmarks in this category may use two different semantics of regular expressions: one based on Boolean matrices and one based on SMT terms for the theory of regular expressions.

Boolean formulas: 88 benchmarks for synthesizing Boolean formulas, including DNF (32), CNF (33), and cube (23) formulas. Each benchmark has between 4 and 128 input/output examples.

Table 1. Solved benchmarks by category.

Domain	Total	TopDown	BottomUp(H)	BottomUp(S)	Virtual Best
Sample Domains	17	14	11	13	15
Regular Expressions	72	52	8	45	54
Boolean	88	45	47	46	49
Bitvectors	100	38	27	36	43
Messy	154	0	0	0	0
Total	431	149	93	140	161

Bitvectors: We provide 100 benchmarks over imperative loop-free bitvector programs [10]. In our adaptation of the existing benchmarks, we consider different bitvector semantics (e.g., one where bitvectors restart at 0 on overflow, and one where the values remain at INT_MIN or INT_MAX). The ability to customize programs semantics is a key feature of SemGuS. These benchmarks use logical specifications instead of input-output examples.

Messy: 154 benchmarks (15 bitvector, 18 imperative, 121 unrealizable SyGuS and imperative) from the original SemGuS paper.

We expect this set to be extended. New benchmarks may be submitted to the Semgus-Benchmarks GitHub repository via pull requests. All submissions are automatically checked for proper syntax and manually reviewed by maintainers for appropriateness before being included.

Performance of Baseline Solvers. Benchmark results for our top-down and bottom-up enumerators by height (H) and size (S) are shown Table 1 as a summary solved instances and Figure 3 as a cactus plot illustrating the time taken to solve the benchmarks. All experiments are run on a cluster [4], with each node having an AMD EPYC 7763 64-Core Processor, of which we requested two cores and 12 GiB of RAM. We set a timeout value of 2000s and memory limit of 8 GiB. We run each experiment 5 times and report the median of these runs.

For Sample Domains, the solvers performed similarly and cumulatively solved 15/17 benchmarks (Virtual Best). Top-down enumeration is a clear winner for Regular Expressions, with height-based bottom-up enumerator performing poorly because the solutions are typically narrow-but-tall. All solvers performed about equivalently on the Boolean benchmarks, although each solver solves a slightly different subset of the problems. For Bitvectors, the bottom-up height-based solver underperformed because these benchmarks have grammars with many productions per non-terminal, thus producing many programs at each

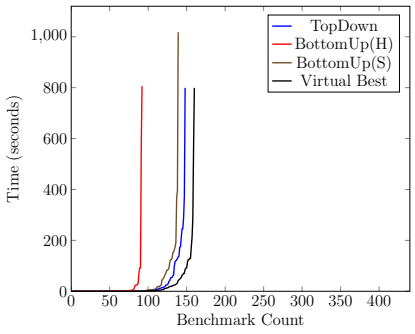

Fig. 3. Cactus plot of runtime. (Lower and to the right is better.)

height. However, size-based bottom-up enumeration could solve 5 problems that the top-down enumerator could not solve, and the top-down enumerator solved 7 that the bottom-up, size-based enumerator could not solve. Note that the Bitvector benchmarks have relational specifications and were solved with CEGIS, but for 19 benchmarks, the verifier failed to check a candidate program or generate counterexamples for at least one solver. For the 43 solved benchmarks, the verifier generated between 1 and 10 counterexamples (average 4.5), in less than 150ms each (average 30ms). The remaining 38 benchmarks generated up to 12 counterexamples before exceeding the timeout or memory limit. Our solver could not solve any Messy benchmarks: most are unrealizable (i.e., they have no solution) or use specifications that are hard to verify using KS2's SMT-based verifier. The Messy solver is particularly good at proving problems unrealizable, but it has not been ported to the SemGuS format and we cannot include it in our baseline.

In terms of enumeration throughput (enumerated programs per second), our solvers perform similarly, and they can enumerate up to 150,000 programs per second (average 33,000) for benchmarks for which verification is quick. The advantage of building and using executable semantics is obvious: if the logical verifier is instead called on each candidate, the throughput drops to at most 800 programs per second (average 175). On the benchmarks where solving with executable semantics and the logical verifier are both supported, the use of the executable semantics is on average 220 times faster than the logical verifier (geomean).

These results provide a baseline rate for future SemGuS solvers to be compared against; the advantage of simple enumerators is their raw speed.

5 Related Work

The syntax-guided-synthesis paradigm [1] has been successfully used in many applications, including invariant synthesis [18,8], and synthesis of rewrite rules and invertibility conditions [20,19]. Several efficient solvers are available for this format [24,2,13]. This effort has inspired several domain-specific extensions for domains that cannot be captured by standard SMT-LIB theories [21]. In contrast, this work develops a general framework for which these extensions can be expressed in a uniform way. Moreover, SemGuS allows one to define synthesis problems—e.g. for imperative programs—that cannot be captured in a natural way by an SMT theory. The syntax-guided-synthesis paradigm has been extended to signatures with oracles [14,22], or symbols whose semantics are given by user-provided binaries. In contrast, in SemGuS, the semantics of all symbols are fully expressed in the problem description.

Acknowledgements

The authors would like to thank Jinwoo Kim, for initial discussions about the SemGuS format; Wiley Corning, Rahul Krishnan, and Shaan Nagy, for code con-

tributions to the SemGuS parser; Evan Geng, Jiangyi Liu, and Charlie Murphy for finding and reporting bugs; and, in addition to everyone previously listed, Kanghee Park, Anvay Grover, and all future contributors for providing SemGuS benchmarks.

Supported, in part, by a Microsoft Faculty Fellowship; a gift from Rajiv and Ritu Batra; and NSF under grants CCF-{1750965, 1918211, 2023222, 2211968, 2212558}. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities.

References

1. Alur, R., Bodik, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: 2013 Formal Methods in Computer-Aided Design. pp. 1–8 (2013). https://doi.org/10.1109/FMCAD.2013.6679385
2. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: Legay, A., Margaria, T. (eds.) 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. Lecture Notes in Computer Science, vol. 10205, pp. 319–336 (2017). https://doi.org/10.1007/978-3-662-54577-5_18, https://doi.org/10.1007/978-3-662-54577-5_18
3. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at www.SMT-LIB.org
4. Center for High Throughput Computing: Center for high throughput computing (2006). https://doi.org/10.21231/GNT1-HW21, https://chtc.cs.wisc.edu/
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973, https://doi.org/10.1145/512950.512973
6. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. J. Log. Program. **13**(2&3), 103–179 (1992). https://doi.org/10.1016/0743-1066(92)90030-7, https://doi.org/10.1016/0743-1066(92)90030-7
7. D'Antoni, L., Hu, Q., Kim, J., Reps, T.: Programmable program synthesis. In: Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33. pp. 84–109. Springer (2021)
8. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Quantified invariants via syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11561, pp. 259–277. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_14, https://doi.org/10.1007/978-3-030-25540-4_14

9. Gulwani, S.: Synthesis from examples. In: WAMBSE (Workshop on Advances in Model-Based Software Engineering) Special Issue, Infosys Labs Briefings. vol. 10. Citeseer (2012)
10. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 62–73. PLDI '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1993498.1993506, https://doi.org/10.1145/1993498.1993506
11. Gulwani, S., Polozov, O., Singh, R.: Program synthesis. Foundations and Trends® in Programming Languages **4**(1-2), 1–119 (2017). https://doi.org/10.1561/2500000010, http://dx.doi.org/10.1561/2500000010
12. Hu, Q., D'Antoni, L.: Syntax-guided synthesis with quantitative syntactic objectives. In: International Conference on Computer Aided Verification. pp. 386–403. Springer (2018)
13. Huang, K., Qiu, X., Shen, P., Wang, Y.: Reconciling enumerative and deductive program synthesis. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 1159–1174. ACM (2020). https://doi.org/10.1145/3385412.3386027, https://doi.org/10.1145/3385412.3386027
14. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Kramer, J., Bishop, J., Devanbu, P.T., Uchitel, S. (eds.) Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010. pp. 215–224. ACM (2010). https://doi.org/10.1145/1806799.1806833, https://doi.org/10.1145/1806799.1806833
15. Kim, J., Hu, Q., D'Antoni, L., Reps, T.: Semantics-guided synthesis. Proceedings of the ACM on Programming Languages **5**(POPL), 1–32 (2021)
16. Lee, M., So, S., Oh, H.: Synthesizing regular expressions from examples for introductory automata assignments. SIGPLAN Not. **52**(3), 70–80 (oct 2016). https://doi.org/10.1145/3093335.2993244, https://doi.org/10.1145/3093335.2993244
17. Li, X., Zhou, X., Dong, R., Zhang, Y., Wang, X.: Efficient bottom-up synthesis for programs with local variables. Proc. ACM Program. Lang. **8**(POPL) (jan 2024). https://doi.org/10.1145/3632894, https://doi.org/10.1145/3632894
18. Miltner, A., Padhi, S., Millstein, T.D., Walker, D.: Data-driven inference of representation invariants. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 1–15. ACM (2020). https://doi.org/10.1145/3385412.3385967, https://doi.org/10.1145/3385412.3385967
19. Niemetz, A., Preiner, M., Reynolds, A., Barrett, C.W., Tinelli, C.: Solving quantified bit-vectors using invertibility conditions. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10982, pp. 236–255. Springer (2018). https://doi.org/10.1007/978-3-319-96142-2_16, https://doi.org/10.1007/978-3-319-96142-2_16
20. Nötzli, A., Reynolds, A., Barbosa, H., Niemetz, A., Preiner, M., Barrett, C.W., Tinelli, C.: Syntax-guided rewrite rule enumeration for SMT solvers. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing - SAT

2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11628, pp. 279–297. Springer (2019). https://doi.org/10.1007/978-3-030-24258-9_20, https://doi.org/10.1007/978-3-030-24258-9_20

21. Padhi, S., Polgreen, E., Raghothaman, M., Reynolds, A., Udupa, A.: The sygus language standard version 2.1. CoRR **abs/2312.06001** (2023). https://doi.org/10.48550/ARXIV.2312.06001, https://doi.org/10.48550/arXiv.2312.06001
22. Polgreen, E., Reynolds, A., Seshia, S.A.: Satisfiability and synthesis modulo oracles. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13182, pp. 263–284. Springer (2022). https://doi.org/10.1007/978-3-030-94583-1_13, https://doi.org/10.1007/978-3-030-94583-1_13
23. Polozov, O., Gulwani, S.: Flashmeta: A framework for inductive program synthesis. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 107–126 (2015)
24. Reynolds, A., Barbosa, H., Nötzli, A., Barrett, C.W., Tinelli, C.: cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11562, pp. 74–83. Springer (2019). https://doi.org/10.1007/978-3-030-25543-5_5, https://doi.org/10.1007/978-3-030-25543-5_5
25. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: Egg: Fast and extensible equality saturation. Proc. ACM Program. Lang. **5**(POPL) (jan 2021). https://doi.org/10.1145/3434304, https://doi.org/10.1145/3434304