Semantics-Guided Synthesis

JINWOO KIM, University of Wisconsin-Madison, USA
QINHEPING HU, University of Wisconsin-Madison, USA
LORIS D’ANTONI, University of Wisconsin-Madison, USA
THOMAS REPS, University of Wisconsin-Madison, USA

This paper develops a new framework for program synthesis, called semantics-guided synthesis (SemGuS), that allows a user to provide both the syntax and the semantics for the constructs in the language. SemGuS accepts a recursively defined big-step semantics, which allows it, for example, to be used to specify and solve synthesis problems over an imperative programming language that may contain loops with unbounded behavior. The customizable nature of SemGuS also allows synthesis problems to be defined over a non-standard semantics, such as an abstract semantics. In addition to the SemGuS framework, we develop an algorithm for solving SemGuS problems that is capable of both synthesizing programs and proving unrealizability, by encoding a SemGuS problem as a proof search over Constrained Horn Clauses: in particular, our approach is the first that we are aware of that can prove unrealizability for synthesis problems that involve imperative programs with unbounded loops, over an infinite syntactic search space. We implemented the technique in a tool called MÆSY, and applied it to both SyGuS problems (i.e., over expressions) and synthesis problems over an imperative programming language.

1 INTRODUCTION

Program synthesis refers to the task of finding a program within a given search space that meets a given behavioral specification (where the specification could be a logical formula or a set of input-output examples). Program synthesis has been studied from a variety of perspectives, which have lead to great practical advances in specific domains [9, 10, 18].

The proliferation of domain-specific synthesis tools has led to numerous attempts to build frameworks that allow one to define and solve synthesis problems in a general fashion. Tools such as Sketch [22] and Rosette [24] have introduced the notion of a solver-aided language, which allows one to define a synthesis problem using a specialized language and then solve the specified problem using a constraint solver. To retain the ability to solve practical problems, these tools have restricted their languages in ways that enable the use of constraint-based synthesis methods—e.g., Sketch and Rosette do not allow arbitrary search spaces involving programs of unbounded size.

While solver-aided languages made synthesis more “programmable”, their mutual incompatibility and language restrictions led to a natural questions: Can we define synthesis problems in a language-agnostic way? This question was partly answered by the framework of of syntax-guided synthesis (SyGuS) [1], which provides a logical framework for defining synthesis problems. In a SyGuS problem, the search space is described using a context-free grammar of terms from a given theory, and the behavioral specification is expressed using a formula in that same logical theory. The unified logical format offered by SyGuS spurred researchers to design synthesizers that could solve problems defined in the SyGuS format [3, 4], and these solvers compete annually in SyGuS competitions [2]. However, SyGuS introduced its own limitation: namely, that the semantics of SyGuS problems are limited to those from a fixed theory, such as linear integer arithmetic (LIA) or bitvectors. This limitation has created a gap between the two approaches: solver-aided languages are unable to express SyGuS problems with infinite search spaces, while SyGuS cannot express...
problems with semantics outside of a supported theory, such as imperative programs containing
loops (which could be modeled using tools like Sketch and Rosette).

**SemGuS.** In this paper, we bridge this gap and present a new synthesis framework, called
semantics-guided synthesis (SemGuS), that attempts to encompass and generalize the two approaches.
Like SyGuS, the goal of SemGuS is to provide a general, logical framework that expresses the
core computational problem of program synthesis [1], without being tied to a specific solution or
implementation. However, in addition to a syntactic search space and a behavioral specification,
SemGuS also allows the user to define the semantics of constructs in the grammar in terms of a set
of inference rules—hence the name “semantics-guided synthesis”.

SemGuS formalizes the concept of semantics through Constrained Horn Clauses, which are
a class of logical formulas that are expressive enough to define a recursive big-step semantics.
The flexibility of the semantic formalism that SemGuS supports allows us to introduce imperative
statements, such as assignments or while loops: in §2, we show how the semantics of these common
constructs can be defined in SemGuS. The customizable aspect of the semantics also provides
a natural way of defining synthesis problems over an alternative semantics (see §4). In essence,
SemGuS extends the “logical framework” of SyGuS towards semantics, resulting in a framework
that is capable of defining SyGuS synthesis problems as well as problems that currently require a
solver-aided language.

**Solving SemGuS Problems.** Following the definition of the SemGuS framework, this paper develops
a method for solving general SemGuS problems capable of producing two-sided answers to a problem:
either synthesizing a solution, or proving that the problem is unrealizable, i.e., has no solution.
Proving the unrealizability of synthesis problems has applications in synthesizing programs that
are optimized with respect to some metric [13], and can be employed in tandem with general
synthesis algorithms as well. However, existing program synthesizers are generally unable to prove
 unrealizability, and focus only on synthesizing terms.

Although SemGuS can be used for much more than imperative program synthesis, solving
SemGuS problems over an imperative programming language illustrates many of the challenges in
computing solutions to general SemGuS problems:

**Reasoning while lacking a direct background theory.** Unlike SyGuS, in which problems are
defined over decidable theories, such as LIA, SemGuS over an imperative programming
language must deal with factors such as state, and there is typically no decidable theory of
the programming language involved.

**Loops.** Loops provide a double challenge in the context of program synthesis: each loop could have
(i) an infinite number of syntactic elaborations (of the condition and the loop-body), each of
which may execute for (ii) an arbitrary number of iterations. Thus, a synthesis algorithm
must reason about sets of loop-body elaborations instead of individual ones—otherwise, the
search space becomes intractable. Existing constraint-based methods often deal with loops
by setting an unrolling bound, which is a factor that limits the kinds of synthesis problems
they can define or solve: SemGuS explicitly tries to avoid this approach.

SemGuS and our solving algorithm address these issues by exploiting the fact that semantics can
be formalized as CHCs, which can be solved using several existing tools.\footnote{In general, the problem of finding a solution to a set of CHCs is uncomputable.}

In §2 and §5, we show that an entire SemGuS problem—syntax, semantics, and behavioral
specification—can be encoded using CHCs, effectively reducing program synthesis into a proof-
search problem that can be solved with off-the-shelf CHC solvers, such as Z3 [8]. If a proof for the
specification exists within the CHC-encoded syntax and semantic rules, the SemGuS problem is
realizable, and the proof identifies a specific term that satisfies the specification. If, on the other hand, the solver can prove that the specification is unsatisfiable using the given rules, then the problem is unrealizable. SemGuS is semantics-guided not only in the sense that it accepts a semantics, but in this proof-search step as well: among the lemmas established during the proof search (by the external solver), some may involve the semantics supplied to SemGuS by the user.

**Contributions.** This paper makes the following contributions:

- The SemGuS framework, which allows the user to supply inference rules that specify the syntax and semantics of the target language. In particular, the SemGuS framework can be used to specify synthesis problems over an imperative programming language (§4).
- A constraint-based approach for solving SemGuS problems using CHCs (§2 and §5), capable of producing both a synthesized program for realizable problems, and a proof of unrealizability for unrealizable ones.
- Multiple instantiations of the framework—with different kinds of semantics—that express variant SemGuS problems whose solutions can sometimes be obtained more efficiently (§6).
- An implementation of a SemGuS solver using Z3 [8, 14], called SEMSY (§7, §8). We instantiate SEMSY to come with a variety of semantics out-of-the-box, allowing users to easily define and solve SemGuS problems. Moreover, SEMSY is the first tool that is capable of both (i) solving synthesis problems, and (ii) proving unrealizability for imperative-language problems that involve a search space with an infinite number of programs.

§3 provides background material. §9 discusses related work. §10 concludes.

## 2 MOTIVATING EXAMPLE

Consider the problem of synthesizing an imperative program that stores the bitwise-xor of two variables \( x \) and \( y \) in the variable \( x \), using only bitwise-and and bitwise-or operations and no auxiliary variables. We show how one can define this problem in SemGuS and prove it unrealizable.

### 2.1 Defining a SemGuS Problem

The first contribution of this paper is the SemGuS framework (§4). A SemGuS problem is defined using three components: (i) a search space given by a regular tree grammar \( G \), (ii) a semantics for the grammar \( G \), and (iii) a specification of the desired behavior of the program.

**Supplying SemGuS with a grammar.** In this example, the grammar \( G_{ex} \) in Figure 1 describes a language of single-loop programs that can contain an arbitrary number of assignments to \( x \) and \( y \), but involve only bitwise-and and bitwise-or operations. In the figure, the numbers in the black circles are used as unique identifiers for each production. Note that SyGuS cannot describe the language \( L(G_{ex}) \) due to the presence of assignments and loops.

**Supplying SemGuS with a semantics.** The next component of a SemGuS problem is a semantics for terms in the language \( L(G_{ex}) \). There are many possible ways to define the formal semantics of an imperative language. For example, if we let \( \Gamma, \Gamma_1, \) and \( \Gamma_2 \) denote valuations of the variables \( x \) and \( y \), then the program "\( x := \) \( x \) xor \( y \)" can be defined by the following rules:

\[
\begin{align*}
\text{Start} & ::= \text{while } B \text{ do } S \\
B & ::= E < E \\
S & ::= S; S \mid x := E \mid y := E \\
E & ::= x \mid y \mid E \&\& E \mid E || E
\end{align*}
\]

Fig. 1. Example grammar \( G_{ex} \).

2 The name M32SY stands for Semantic (SEM written backwards) Synthesizer (SY).
Given a logical specification, one can always generate a set of examples and add more examples as needed through a technique known as counterexample-guided inductive synthesis (CEGIS), which is applied in many synthesizers.

3 The ability to define multiple semantic rules for a production is useful when defining semantics for productions such as Start → while b do s, which is commonly equipped with two rules that describe looping and loop termination.

4 For clarity, we sometimes use the format from Equation (1) when introducing semantics for productions such as Start → while b do s. In §6, we show how our algorithm (implemented in MARSY) can synthesize a valid solution on a subset of Eex, namely, [(6, 9)] with output [15], where bitwise-xor is equivalent to bitwise-or, i.e., this sub-problem is realizable. We then describe how our algorithm proves that no program in the language of L(Gex) can compute the bitwise-xor for all the examples in Eex, i.e., that the problem is unrealizable.

5 Given a logical specification, one can always generate a set of examples and add more examples as needed through a technique known as counterexample-guided inductive synthesis (CEGIS), which is applied in many synthesizers.
2.2 Solving SEMGuS Problems

The second contribution of this paper is a procedure for solving SEMGuS problems (§5). To solve a SEMGuS problem, this paper utilizes two key ideas: (i) both the syntax and the semantics of a synthesis problem can be described using Constrained Horn Clauses, and, (ii) one can phrase the synthesis problem as a proof search over CHCs.

Syntax and Semantic Rules. Describing a grammar using CHCs is a straightforward process: taking the production \( \text{Start} \rightarrow \text{while } B \text{ do } S \) as an example, the production states that one can obtain a valid term for the nonterminal \( \text{Start} \) using valid terms for nonterminals \( B \) and \( S \). Equation (3) encodes this property as a CHC.

\[
\frac{\text{syn}_B(b) \quad \text{syn}_S(s)}{\text{syn}_\text{Start}(\text{while } b \text{ do } s)} \quad \text{syntax}_{\text{Start} \rightarrow \text{while } B \text{ do } S}
\]

The logical relations \( \text{syn}_B, \text{syn}_S, \) and \( \text{syn}_\text{Start} \) in Equation (3) model whether the supplied arguments are valid terms that may be derived from the corresponding nonterminals \( B, S, \) and \( \text{Start} \). We refer to relations such as \( \text{syn}_S \) as syntax relations, and rules such as Equation (3) as syntax rules.

§2.1 illustrated how the programming-language semantics can be expressed using CHCs; in tandem with the syntax rules, they represent the semantics of all possible programs in the language.

Specification Query. The final step to solving a SEMGuS problem is to create a query that encodes the behavioral specification, asking whether any of the programs generated by the grammar is consistent with the specification on the set of input examples \( E \). This question is posed via the Query rule below, which checks for the existence of a term \( t \) that satisfies the syntax rules and the semantic rules, each instantiated with input \( e_i \in E \) and corresponding output value \( o_i \).

\[
\frac{\text{syn}_\text{Start}(t) \quad \land_{e_i \in E} \text{sem}_\text{Start}((e_i, t), o_i)}{\text{Realizable}} \quad \text{Query}
\]

Generally, one could choose to use symbolic variables for \( o_i \) instead of concrete output examples, by adding an additional premise \( \land_{e_i \in E} \psi(e_i, o_i) \) to ensure that the input-output pair \( e_i, o_i \) meets the specification \( \psi \). In this section, we consider concrete output examples for ease of presentation.

Expressing the entire SEMGuS problem as a set of inference rules and a query effectively reduces solving the SEMGuS problem to a proof search to establish that \( \text{Realizable} \) holds using the given inference rules. If one can prove that the premises of Equation (4) hold, then the SEMGuS problem is realizable, and the term \( t \) is a concrete answer to the problem. If there exists no proof for \( \text{Realizable} \) using the inference rules, then the SEMGuS problem is unrealizable.

Synthesizing Programs. To see how a valid program is synthesized based on our construction, consider our problem of synthesizing a program that computes bitwise-xor, specified using the singleton example set \[(x, y) = [(6, 9)]\]. In this case, the CHC solver is responsible for finding a term \( t \) that satisfies the conjunction of the relations (and, as stated above, also corresponds to proving):

\[
\text{syn}_\text{Start}(t) \quad \text{sem}_\text{Start}(((6, 9), t), (15))
\]

For this input/output pair, bitwise-xor is indistinguishable from bitwise-or, making the problem realizable. In this case, the term “while \( x < y \) do \( x := x \parallel y \)” corresponds to a solution—and our tool \( \text{MS2SY} \) (which is based on Z3 [8] and its CHC solver \( \text{Spacer} \) [14]) succeeds in finding the term.

---

6 In practice, we encode terms into an alternate representations because it is difficult to express terms directly in SMT solvers. This encoding is presented in §5.
Proving Unrealizability. To see how a SemGuS problem is proved unrealizable, recall our full example set $E_{ex}$, for which the solver must find some term $t$ that satisfies the relations:

\[
\begin{align*}
\text{syn}_{\text{Start}}(t) & \quad \text{sem}_{\text{Start}}(((6, 9), t), 15) \\
\text{sem}_{\text{Start}}(((44, 247), t), 219) & \quad \text{sem}_{\text{Start}}(((14, 15), t), 1)
\end{align*}
\]

Put another way, the solver must establish that there exists no term $t$ that satisfies all four relations at once—i.e., that Realizable is unsatisfiable—to prove the problem unrealizable.

Note that our algorithm does not provide additional machinery to reason about loops. Instead, we rely on the CHC solver to discover lemmas about sets of loops—as opposed to single loops—to prune the search space. When proving that no program in $L(G_{ex})$ is consistent with the examples in $E_{ex}$, the CHC solver Spacer infers the lemma that states that for the third example, namely, $(14, 15) \rightarrow 1$, the third bit of whatever value is assigned to $x$ when the loop terminates must be set to true. This condition conflicts with the output 1 (in which the third bit is false), which shows that the third example can never be satisfied—which, in turn, implies that the synthesis problem is unrealizable!

Note that this lemma is an invariant of the nonterminal Start—i.e., an invariant of all programs derivable from Start—not just some specific program derivable from Start.

One might be tempted to give an operational reading of the Query rule as following the paradigm of generate and test: $\text{syn}_{\text{Start}}(t)$ generates $t$, which then must pass the tests $\text{sem}_{\text{Start}}((I_1, t), O_1) \ldots \text{sem}_{\text{Start}}((I_n, t), O_n)$. However, the ability of Spacer to prove lemmas of the sort discussed above means that M32SY is not merely enumerating and testing individual programs. On the contrary, the technique for solving SemGuS problems infers lemmas about the behavior of multiple programs in the language of the grammar, and uses them to prune the search space!

### 2.3 Instantiating SemGuS with Other Semantics

The procedure described in the previous section gives a general way to solve SemGuS problems, but also suffers from several limitations. For example, one might have to prove a large number of semantic relations from the premise of the Query rule if there are a large number of input-output examples; or, because solving CHCs is still difficult in general, the problem may simply be too difficult to solve. As a third contribution, we show how, thanks to its generality, SemGuS can be supplied with alternative semantics to address some of these challenges (§6). As an example, here we show how to supply SemGuS with an abstract semantics to prove unrealizability more efficiently.

Consider again the problem of proving that synthesizing a bitwise-xor program from the grammar $G_{ex}$ is unrealizable. As described in §2.2, the lemma used to prove this fact states that the third bit of $x$ under the example $(14, 15) \rightarrow 1$ is always set to true, conflicting with the output 1. While we proved this problem unrealizable using a precise semantics, it is also possible to prove unrealizability using an abstract domain. For example, consider the abstract domain $B_3$, which only tracks the value of the third bit of every variable, using the values true, false, and $\top$ (top), where $\top$ represents the scenario in which the third bit may be either true or false; i.e., the semantics may be imprecise. Then, one could supply an abstract semantics for a term $e_1 \&\& e_2$, created from the production $E \rightarrow E\&\&E$, as:

\[
[e_1]^*(I^*, v_1^*) \quad [e_2]^*(I^*, v_2^*) \quad \nu^* = (if \ (v_1^* = \top \lor v_2^* = \top) \ then \ \top \ else \ v_1^* \&\& v_2^*)
\]

The final premise in Equation (5) represents the abstract transformer of bitwise-and $B_3$, which sends the computation to $\top$ if any of $v_1^*$ or $v_2^*$, the abstract values for $v_1$ and $v_2$, are $\top$, or computes the exact value otherwise. $\top$ can be generated in $B_3$ by operators such as $+\$, which always loses precision because it does not track carry bit values from the second position.

Proc. ACM Program. Lang., Vol. 1, No. CONF, Article 1. Publication date: January 2018.
From a SemGuS point of view, an abstract semantics is merely a different semantics, which allows SemGuS problems with abstract semantics such as \( \mathbb{B}_3 \) to be solved using the same algorithm described in §2.2. Although \( \mathbb{B}_3 \) is more lightweight compared to the precise semantics discussed in §2.1, it is sufficient to prove the unrealizability of synthesizing bitwise-xor from \( \mathcal{G}_{ex} \)—therefore resulting in a more efficient solving procedure.

In §6, we show how other semantics, such as an underapproximating one, can be supplied to the SemGuS framework, each with their advantages. These semantics illustrate one of the benefits of allowing a user to supply their own semantics in SemGuS—in addition to a wider range of definable problems, one can also describe specific strategies to optimize the synthesis problem at hand!

## 3 PRELIMINARIES

In this section, we provide some background information on concepts that we build upon for the rest of the paper. §3.1 provides background on Horn Clauses, which are used in §5 to define our procedure for solving SemGuS problems. §3.2 is about trees, regular tree grammars, and program semantics, which are required for our definition of the SemGuS problem in §4.

### 3.1 Constrained Horn Clauses

**Constrained Horn Clauses (CHCs)** are a class of logical rules that we use to formalize the concept of semantics, as well as use in our algorithm for solving SemGuS problems.

**Definition 3.1 (Constrained Horn Clauses,).** A Constrained Horn Clause is a first-order formula of the form \( \forall \bar{x}, \bar{x}_1, \ldots, \bar{x}_n. (\phi \land R_1(\bar{x}_1) \land \cdots \land R_n(\bar{x}_n) \implies H(\bar{x})) \), where \( \phi \) is a constraint over some background theory that may contain variables from \( \bar{x}, \bar{x}_1, \ldots, \bar{x}_n \), and \( R_1, \ldots, R_n \) and \( H \) are uninterpreted relations.

**Example 3.2.** Equations (6) and (7) give an example of how the syntax and semantic rules from §2 can be interpreted as CHCs.

\[
\forall b, s. \text{syn}_B(b) \land \text{syn}_S(s) \Rightarrow \text{syn}_{\text{Start}}(\text{while } b \text{ do } s) \tag{6}
\]

\[
\forall \Gamma, \Gamma_1, \Gamma_2, b, s. (v_b = \text{true}) \land \text{sem}_B(\langle \Gamma, b, v_b \rangle) \land \text{sem}_S(\langle \Gamma_1, s, \Gamma_\text{I} \rangle) \land \text{sem}_{\text{Start}}(\langle \Gamma_1, \text{while } b \text{ do } s, \Gamma_2 \rangle) \Rightarrow \text{sem}_{\text{Start}}(\langle \Gamma, \text{while } b \text{ do } s, \Gamma_2 \rangle) \tag{7}
\]

Syntax and semantic relations such as \( \text{syn}_B \) or \( \text{sem}_{\text{Start}} \) are expressed as uninterpreted relations, while atomic semantic operations such as addition are represented using the constraint \( \phi \).

CHCs occur frequently in program verification, where verification conditions for constructs such as loop invariants are produced in the form of CHCs. As a consequence, many efficient algorithms for solving CHCs have been developed [5, 14, 16]. In this paper, we rely on off-the-shelf CHC solvers to produce an answer for the queries we generate.

### 3.2 Trees, Tree Grammars, and Semantics

A ranked alphabet is a tuple \( (\Sigma, \text{rk}_\Sigma) \), where \( \Sigma \) is a finite set of symbols, and \( \text{rk}_\Sigma : \Sigma \rightarrow \mathbb{N} \) associates a rank to each symbol. For every \( m \geq 0 \), the set of all symbols in \( \Sigma \) with rank \( m \) is denoted by \( \Sigma^{(m)} \). In our examples, a ranked alphabet is specified by showing the set \( \Sigma \) and attaching the respective rank to every symbol as a superscript—e.g., \( \Sigma = \{ s^{(2)}, c^{(0)} \} \). (For brevity, the superscript is often omitted.) We use \( T_\Sigma \) to denote the set of all (ranked) trees over \( \Sigma \)—i.e., \( T_\Sigma \) is the smallest set such that (i) \( \Sigma^{(0)} \subseteq T_\Sigma \), (ii) if \( \sigma^{(k)} \in \Sigma^{(k)} \) and \( t_1, \ldots, t_k \in T_\Sigma \), then \( \sigma^{(k)}(t_1, \ldots, t_k) \in T_\Sigma \). In what follows, we assume a fixed ranked alphabet \( (\Sigma, \text{rk}_\Sigma) \).
In this paper, we focus on typed regular tree grammars, in which each nonterminal and each symbol is associated with a type. There is a finite set of types \( \{ \tau_1, \ldots, \tau_l \} \). Associated with each symbol \( \sigma^{(i)} \in \Sigma^{(i)} \), there is a type assignment \( a_{\sigma^{(i)}} = (\tau_0, \tau_1, \ldots, \tau_l) \), where \( \tau_0 \) is called the left-hand-side type and \( \tau_1, \ldots, \tau_l \) are called the right-hand-side types. Tree grammars are similar to word grammars, but generate trees over a ranked alphabet instead of words.

**Definition 3.3 (Regular tree grammar).** A typed regular tree grammar (RTG) is a tuple \( G = (N, \Sigma, S, a, \delta) \), where \( N \) is a finite set of non-terminal symbols of arity 0; \( \Sigma \) is a ranked alphabet; \( S \in N \) is an initial nonterminal; \( a \) is a type assignment that gives types for members of \( \Sigma \cup N \); and \( \delta \) is a finite set of productions of the form \( A_0 \rightarrow \sigma^{(i)}(A_1, \ldots, A_l) \), where for \( 1 \leq j \leq i \), each \( A_j \in N \) is a nonterminal such that if \( a_{\sigma^{(i)}} = (\tau_0, \tau_1, \ldots, \tau_l) \), then \( a_{A_j} = \tau_j \).

Given a tree \( t \in T_{\Sigma \cup N} \), applying a production \( r = A \rightarrow \beta \) to \( t \) produces the tree \( t' \) resulting from replacing the leftmost occurrence of \( A \) in \( t \) with the right-hand side \( \beta \). A tree \( t \in T_{\Sigma} \) is generated by the grammar \( G \)—denoted by \( t \in L(G) \)—iff it can be obtained by applying a sequence of productions \( r_1 \cdots r_n \) to the tree whose root is the initial non-terminal \( S \).

Figure 1 from §2 shows an example of a typed regular tree grammar. For readability, the grammar does not contain explicit symbols—e.g., the production \( \text{Start} \rightarrow \text{while} B \text{ do } S \) should be more correctly stated as a production \( \text{Start} \rightarrow \text{while}(B,S) \), where while is a binary symbol. We will use the former notation for readability, and assume that all expressions are well-typed.

We note that terms can be represented using trees of productions, which makes it easier to distinguish terms created by different productions with identical operators: we use this representation in §5.2, where we show how listings can be used to represent terms and optimize our solving procedure.

**Example 3.4.** Recall the grammar \( G_{ex} \) from Figure 1, where each production is labeled with a unique identifier \( \circ \). The term “\( \text{while } x < x \text{ do } x := y' \)” can be represented using the tree \( \text{Tree}_3(\text{Tree}_2(\circ, \circ), \text{Tree}_1(\circ)) \). The first child tree \( \text{Tree}_2(\circ, \circ) \) represents the condition “\( x < x' \)”, while the second child tree \( \text{Tree}_1(\circ) \) represents the assignment “\( x := y' \)”. When defining a \( \text{SEMGuS} \) problem, one has to provide a semantics for the productions in the RTG. The semantic definitions are all to use terms from a theory \( T \) (e.g., linear integer arithmetic).

**Definition 3.5 (Production-based semantics).** Given an RTG \( G = (N, \Sigma, S, a, \delta) \) and a theory \( T \), a semantics for the grammar is a function \( \{ \} \) that maps every production \( A_0 \rightarrow \sigma^{(i)}(A_1, \ldots, A_l) \) of type \( a_{\sigma^{(i)}} = (\tau_0, \tau_1, \ldots, \tau_l) \) to a set of Constrained Horn Clauses of the form \( \phi \land \text{sem}_{A_0}(\Gamma_0, t_{A_0}, v_0) \land \cdots \text{sem}_{A_l}(\Gamma_l, t_{A_l}, v_l) \implies \text{sem}_{A_k}(\Gamma_k, t_{A_k}, v_k) \) where \( \text{sem}_{A_0}, \text{sem}_{A_1}, \ldots, \text{sem}_{A_l} \) are uninterpreted relations, \( \Gamma_0, \Gamma_1, \ldots, \Gamma_l \) are variables that represent state, \( t_{A_k} \) is a variable that represents a term \( t \in L(A_k) \), \( v_0, v_1, \ldots, v_l \) are variables of type \( \tau_0, \tau_1, \ldots, \tau_l \), and \( \phi \) is a constraint within the theory \( T \).

The function \( \{ \} \) can be lifted to trees as follows: for every subtree \( t' \) of \( t \), if \( t' = \sigma^{(i)}(t_1, \ldots, t_l) \), then \( [t'] = [\sigma^{(i)}([t_1], \ldots, [t_l])] \).

As is common in many semantic definitions, Def. 3.5 defines the semantics of terms in the grammar inductively. This ability to equip the grammar with customized semantics is the defining characteristic that distinguishes \( \text{SEMGuS} \) from \( \text{SyGuS} \). In \( \text{SyGuS} \), the underlying theory—e.g., \( \text{LIA} \)—is what corresponds to the specified semantics. In \( \text{SEMGuS} \), the semantics can be any Constrained Horn Clause defined over the relations \( \text{sem}_{A_0}, \text{sem}_{A_1}, \ldots, \text{sem}_{A_l} \).

**Example 3.6.** The big-step semantics of simple imperative languages can be expressed using rules like the one illustrated in Equation (2), which inductively defines the semantic of the production \( \text{Start} \rightarrow \) while \( B \) do \( S \) through the semantic relations for nonterminals \( B \) and \( S \).
4 SEMANTICS-GUIDED SYNTHESIS AND ITS PROPERTIES

We now provide a formal definition of the Semantics-Guided Synthesis problem:

Definition 4.1 (SemGuS). A SemGuS problem over a theory $T$ is a tuple $\text{sem} = (G, \forall x. \psi(x, f(x)))$, where $G$ is a regular tree grammar with a production-based semantics $\sem$, and $\forall x. \psi(x, f) (x)$ is a Boolean formula over the theory $T$ that specifies the desired behavior of $f$, where $f$ is a free second-order variable. A solution to the SemGuS problem $\text{sem}$ is a term $s \in L(G)$ such that $\forall x. \psi(x, [s] (x))$ holds. We say that $\text{sem}$ is realizable if a solution exists and unrealizable otherwise.

Example 4.2. The problem of synthesizing a program for bitwise-xor described in §2 can be written as a SemGuS problem $\text{sem} = (G_{ex}, \forall x, y. f(x, y) = x \oplus y)$ (with $\oplus$ denoting bitwise-xor), where $G_{ex}$ is equipped with a semantics that contains the rule given in Equation (2).

Example 4.2 gives an example of a SemGuS problem where the grammar is equipped with a semantics that one would normally expect for imperative programs. Definition 4.1, which defines SemGuS problems, shows that SemGuS can be instantiated with different kinds of semantics, as long as the semantics satisfies the definition of a production-based semantics (Definition 3.5). This feature allows SemGuS problems to be instantiated with a semantics that is approximate with respect to some original semantics. An approximate semantics can be used to efficiently compute one-sided answers to the original problem—either synthesis or unrealizability—depending on the relation between the approximating and the original semantics.

4.1 Unrealizability of SemGuS Problems with Overapproximating Abstract Semantics

In this section, we see how an overapproximating semantics can be used to prove unrealizability. An overapproximating semantics overapproximates the set of reachable states with respect to an original program semantics; in essence, they are an abstract semantics [7], and we use the latter term for the rest of the paper. More specifically, we show that if a SemGuS problem $\text{sem} = (G, \psi(x, f(x)))$ is unrealizable when $G$ is equipped with an abstract semantics, then $\text{sem}$ is unrealizable when equipped with the original semantics as well.

Definition 4.3. For a grammar $G$ equipped with a semantics $\sem$, we say $\sem$ is an abstract semantics for $G$ with respect to $\sem$ if there exists an abstraction function $\alpha$ and a concretization function $\gamma$, such that for all $t \in L(G)$, if $[t] (\Gamma, v)$ holds, then $[[t]]^* (\alpha(\Gamma), \alpha(v))$ holds, and $\Gamma \in \gamma (\alpha(\Gamma)), v \in \gamma (\alpha(v))$, i.e., $\alpha$ and $\gamma$ form a Galois connection.

In SemGuS, an abstract semantics $\sem$ overapproximates the set of values that are obtainable by synthesizing a term from the grammar, again with respect to the original semantics $\sem$. Because the set of values is overapproximated, a term synthesized using the abstract semantics may not satisfy the specification when executed with the standard semantics. However, by showing the desired output is absent from the set of obtainable values, one can prove unrealizability in a sound manner!

Theorem 4.4 (Soundness of Abstract Semantics for Unrealizability). For a SemGuS problem $\text{sem} = (G, \forall x. \psi(x, f(x)))$, if $\text{sem}$ is unrealizable when $G$ is equipped with an abstract semantics $\sem$, then $\text{sem}$ is also unrealizable when $G$ is equipped with $\sem$.

Proof. If $\text{sem} = (G, \forall x. \psi(x, f(x)))$ with the semantics $\sem$ is unrealizable, then for every $t \in L(G)$, there exists some input-output pair $(i, o)$ such that $\psi(i, o)$ holds, but $[t]^*(\alpha(i), \alpha(o))$ does not hold. By contraposition, $[t] (i, o)$ also does not hold, and thus $\text{sem}$ equipped with $\sem$ is unrealizable. □

Equipping a SemGuS problem with an abstract semantics still results in a SemGuS problem, which can be solved using the procedure described in §5. Much like how abstract semantics are used for efficient program verification, an abstract semantics can sometimes be used to prove the unrealizability of a SemGuS problem with the original semantics in a much more efficient manner.
4.2 Solving Realizable SemGuS Problems with Underapproximating Semantics

In this section, we show that an underapproximating semantics, can be used to synthesize solutions to realizable SemGuS problems.

Definition 4.5. For a grammar $G$ equipped with a semantics $[\cdot]$, we say $[\cdot]^b$ underapproximates $[\cdot]$ on $G$, or that $[\cdot]^b$ is an underapproximating semantics for $G$ with respect to $[\cdot]$, if for every term $t \in L(G)$, every state $\Gamma$, and every value $v$ on which $[\cdot]^b$ is defined, $[t]^b(\Gamma, v) = [t](\Gamma, v)$.

Intuitively, an underapproximating semantics is defined as a subset of the original semantics. Outside of the subset upon which it is defined, an underapproximating semantics is undefined, which does not mean that a term can evaluate to any value, but rather that a term cannot evaluate to any value. More precisely, one cannot prove any theorems about the relation $[\cdot]^b$ is undefined on $t$, $\Gamma$, and $v$. Instead, an underapproximate semantics is precise on the subset upon which it is defined, i.e., $[t]^b(\Gamma, v) = [t](\Gamma, v)$ if $[\cdot]^b$ is defined on $t$, $\Gamma$, and $v$.

In SemGuS, an underapproximating semantics corresponds to a problem where synthesized terms only have meaning if their semantics is defined on the input-output examples. For the subset of terms for which the semantics is defined, the semantics is exact, which allows underapproximating semantics to be used for program synthesis. Because there may be an answer to the problem outside the defined subset, an underapproximating semantics cannot be used for unrealizability.

Theorem 4.6 (Soundness of Underapproximating Semantics for Synthesis). For a SemGuS problem $sem = (G, \forall x. \psi(x, f(x)))$, if $sem$ is realizable with solution $t$ when $G$ is equipped with an underapproximating semantics $[\cdot]^b$, then $t$ is also a solution for $sem$ when $G$ is equipped with $[\cdot]$.

Proof. That $sem = (G, \forall x. \psi(x, f(x)))$ equipped with $[\cdot]^b$ is realizable means that $\exists t \in L(G)$ such that $\forall x. \psi(x, v)$ and $[t]^b(x, v)$; thus $[t](x, v)$ as well, and $sem$ is realizable with $[\cdot]$. □

An underapproximate semantics indirectly restricts the search space for program synthesis. This restriction is not necessarily related to the grammar supplied to a SemGuS problem, but may have a semantic meaning—for example, a bound on the number of possible loop iterations.

As is the case with an abstract semantics, SemGuS can be supplied with an underapproximate semantics to yield a relatively more efficient procedure for program synthesis, as illustrated in §6.3.

5 SOLVING SEMANTICS-GUIDED SYNTHESIS PROBLEMS VIA CONSTRAINED HORN CLAUSES

This section presents a general procedure for encoding SemGuS problems so that they can be solved by answering a query over Constrained Horn Clauses, which in turn can be solved by an off-the-shelf CHC solver.

§5.1 describes how general SemGuS problems can be solved by solving SemGuS-with-examples problems in tandem with counterexample-guided inductive synthesis; it also states the correctness of our solving procedure. §5.2 presents a method for using flattened representations of terms as opposed to trees, to avoid the use of algebraic datatypes in SMT solvers.

5.1 Solving SemGuS Problems with Counterexample-Guided Inductive Synthesis

Counterexample-guided inductive synthesis (CEGIS) is a widely implemented algorithm in program synthesizers. The core idea of CEGIS is that instead of searching for a term that satisfies the specification for the entire input space, the synthesizer searches for a solution that satisfies the specification on a finite set of examples $E$. A verifier then attempts to prove that the solution is also correct on the universally quantified specification; if not, a counterexample is added to the set

Proc. ACM Program. Lang., Vol. 1, No. CONF, Article 1. Publication date: January 2018.
of examples. The algorithm then repeats. The main advantage of CEGIS is that it eliminates the universal quantifier over the space of program inputs, yielding a simpler problem.

The algorithm sketched in §2, as well as the one presented in §5.2, is designed to solve \textsc{SemGuS}-with-examples problems, which are \textsc{SemGuS} problems where the specification is given in terms of a set of examples \( E \), and has the form \( \land_{x \in E} \psi(x, [f](x)) \). To solve general \textsc{SemGuS} problems, the \textsc{SemGuS}-with-examples algorithm can then be embedded within a CEGIS loop, where the specification is given in terms of the set of counterexamples accumulated by CEGIS.

The general idea of using CHCs to describe the syntax and semantics of a \textsc{SemGuS}-with-examples problem \( sem = (G, \forall x \in E. \psi(x, f(x))) \) has already been described in §2: Equation (3) and Equation (2) show how the syntax and the semantics of a production \( \text{Start} \rightarrow \) while \( B \) do \( S \) can be written as CHCs, and it is straightforward to describe other productions in this manner as well.

The final query that describes the specification can be formally written as the following rule.

\[
\text{Realizable} \quad \frac{\text{syn}_{\text{Start}}(t) \land_{e_i \in E} \text{sem}_{\text{Start}}((e_i, t), a_i) \land_{e_i \in E} \psi(e_i, o_i)}{\text{Realizable}} \quad \text{Query (8)}
\]

\text{Realizable} is the final theorem that shows whether the given \textsc{SemGuS}-with-examples problem is realizable or not. If the CHC solver finds a proof for \text{Realizable}, then the problem is \text{realizable} and the program \( t \) is a solution. If the solver can establish that \text{Realizable} is \text{unsatisfiable}, then the problem is \text{unrealizable}. The correctness of our algorithm can be stated as the following theorem:

\textbf{Theorem 5.1 (Soundness and Completeness).} Consider a \textsc{SemGuS}-with-examples problem \( sem = (G, \forall x \in E. \psi(x, f(x))) \), equipped with semantic rules \( R_{sem} \), a specification set \( E \), and the Query rule (Equation (8)). Let the CHC form of \( G \) be \( R_{syn} \). Then, \text{Realizable} is a theorem over \( R_{sem} \) and \( R_{syn} \) if and only if the \textsc{SemGuS}-with-examples problem \( sem \) is realizable. Moreover, if \text{Realizable} is a theorem, then the value of \( t \) in the Query rule satisfies \( t \in L(G) \) and \( \forall x \in E. \psi(x, [f](x)) \).

Theorem 5.1 can be proved by proving the correctness of the syntax rules (see Appendix B).

As shown in prior work [22], the CEGIS algorithm is often powerful enough for program synthesis, where a term synthesized for the given examples generalizes to the entire space of possible inputs. Prior work on unrealizability [11, 12] also shows that CEGIS is often powerful enough to prove that a synthesis problem is unrealizable—i.e., the problem does not admit a solution even when only a finite number of examples are considered.

\textbf{Example 5.2.} The problem of synthesizing a program for bitwise-xor described in §2 can be written as a \textsc{SemGuS}-with-examples problem \( sem = (G_{ex}, \forall x, y \in E_{ex}. f(x, y) = x \oplus y) \), where \( E_{ex} = [(6, 9), (44, 247), (14, 15)] \). As seen in §2, \( E_{ex} \) is sufficient to prove that \( sem \) is unrealizable.

In particular, for a \textsc{SemGuS} problem \( sem \), the CEGIS algorithm is sound but incomplete for unrealizability [11]. As discussed in §8, CEGIS is still able to synthesize solutions to, or prove unrealizability of, many \textsc{SemGuS} problems. However, this procedure is incomplete.

\textbf{Theorem 5.3 (CEGIS for Unrealizability [11]).} Let \( sem_{E} \) be a \textsc{SemGuS}-with-examples problem identical to \( sem \), but where the specification is given over the input examples \( E \). If \( sem_{E} \) is unrealizable, then \( sem \) is unrealizable as well. However, there exists an unrealizable \textsc{SemGuS} problem \( sem \) for which \( sem_{E} \) is realizable for any finite set of examples \( E \).

\section{5.2 Using Flattened Representations of Terms to Solve \textsc{SemGuS} Problems}

While it is possible to solve \textsc{SemGuS}-with-examples problems using terms encoded as trees using the scheme given in §3.2, current solvers will sometimes fail to return an answer depending on
how well they can handle trees encoded as algebraic datatypes. In this section, we show how to alleviate this problem by using a flattened representation of terms, which we refer to as a listing.\footnote{Listings may be implemented as lists or arrays in an SMT solver.}

The key idea is that a term $t$ can be encoded using a pre-order listing $L_t$ of the productions applied to derive $t$.

Example 5.4. Consider once more the term $t = \text{while } (x < x) \text{ do } x := x$ from Example 3.4, constructed from the grammar $G_{ex}$ in Figure 1. The pre-order listing of productions applied to derive $t$ is $[1, 2, 6, 6, 6]$, where $\text{Start} \rightarrow \text{while } B \rightarrow S$ is the first production applied to the nonterminal $\text{Start}$, the next production $B \rightarrow E < E$ is applied next, and the remaining productions are applied in left-to-right order as well.

Following the list representation of terms, the next step is to modify the syntax relations and rules to operate over lists. Equation (9) describes the syntax rule generated using a flattened representation of terms for the production $A_0 \rightarrow \sigma(A_1, \ldots, A_i)$.

$$
\frac{\text{syn}_{A_1}(L_{in}, L_i) \quad \text{syn}_{A_{i-1}}(L_i, L_{i-1}) \cdots \text{syn}_{A_1}(L_2, L_1)}{\text{syn}_{A_0}(L_{in}, \langle \cdot : \cdot \rangle) \quad \text{syntax}_{A_0 \rightarrow \sigma(A_1, \cdots, A_i)}} \tag{9}
$$

There are several things to notice about Equation (9). First, the syntax relation $\text{syn}_{x}$ now ranges over two listings (term representations) as opposed to a single term, where the first listing may be interpreted as an incoming listing and the second an outgoing listing. Here, the relations should evaluate to true if and only if the outgoing listing is equivalent to the pre-order representation of the term concatenated to the incoming listing.

Second, the outgoing listing of a nonterminal is passed as the incoming listing of the next nonterminal in right-to-left order, followed by prepending the number of the production to the head of the listing. This algorithm effectively creates a pre-order representation of a term by performing a post-order traversal, appending each production encountered to the head of the listing.

Example 5.5. Consider Equation (3) from §2, which describes the syntax rule for the production $\text{Start} \rightarrow \text{while } B \text{ do } S$. Using a list representation of terms, the rule would be modified to:

$$
\frac{\text{syn}_{\text{Start}}(L_{in}, \langle \cdot : \cdot \rangle) \quad \text{syn}_{\text{B}}(L_2, L_1) \quad \text{syn}_{\text{A}}(L_2, L_1)}{\text{syntax}_{\text{Start} \rightarrow \text{while } B \text{ do } S}} \tag{10}
$$

Equation (10) traverses the nonterminals $B, S$ in right-to-left order, then prepends the identifier $1$ to the head of the list $L_1$.

Having encoded a pre-order representation of a term, the semantic rules must interpret this representation accordingly as well. The semantic relations now also range over 4 elements: an incoming listing $L_{in}$ and an incoming state $\Gamma$, and a resulting value $v$. They should evaluate to true if and only if for the list $L_t$ such that $L_{in} = L_t + L_{out}$, $[t](\Gamma, v)$ also evaluates to true for the corresponding term $t$.

Keeping that in mind, a semantic rule that uses a flattened representation of terms for the production $A_0 \rightarrow \sigma(A_1, \ldots, A_i)$, equipped with the semantics $\phi \land \text{sem}_{A_i}(\langle \Gamma_1, t_1 \rangle, v_1), \ldots, \text{sem}_{A_i}(\langle \Gamma_i, t_i \rangle, v_i)$ is described in Equation (11).

$$
\phi \land \text{sem}_{A_i}(\langle \Gamma_1, L_1 \rangle, \langle v_1, L_2 \rangle) \cdots \text{sem}_{A_i}(\langle \Gamma_i, L_i \rangle, \langle v_i, L_{out} \rangle) \implies \text{sem}_{A_0}(\langle \Gamma, t \rangle, v_0) \implies \text{sem}_{A_0}(\langle \Gamma, t \rangle, v_0, L_{out}) \tag{11}
$$
We now proceed to showcase the capabilities of the SemGuS framework by instantiating it with a variety of semantics to solve imperative program synthesis problems. In this section, we are concerned with various different semantics for the imperative programming language $G_{impv}$, from Figure 2. $G_{impv}$ ranges over integers, bitvectors, Boolean values, and arrays; which contains most common imperative structures, such as assignments, branches and loops. Imperative grammars that use the same operators but different productions can be viewed as being derived from $G_{impv}$, which means that the techniques introduced in this section are applicable to any imperative grammar as long as they use a subset of the operators in $G_{impv}$.

In §6.1, we discuss how to instantiate an imperative SemGuS problem with an alternative exact semantics. This semantics, called a vectorized semantics, sidesteps the problem of having to consider multiple examples separately. In §6.2, we show how SemGuS can be instantiated with an abstract semantics to prove the unrealizablility of a synthesis problem, and in §6.3, how an underapproximating semantics can be used to more efficiently compute solutions for a realizable problem.

6.1 Instantiating SemGuS with an Alternate Exact Semantics
A straightforward way of instantiating a SemGuS problem is to supply SemGuS with a standard semantics, as discussed in §2 and §5. For example, the three rules in Figure 3a are standard semantic
rules that define the semantics of the terms "\( x := e \)" and "while \( b \) do \( s \)". These semantics operate over a single state, and compute exact values for all terms in the program.

However, this straightforward approach induces a substantial drawback in the Query rule in Equation (8). Each premise of the Query rule, the solver must re-derive proof trees for each example, even though they are all structurally similar due to sharing the same term representation.

To mitigate this inefficiency, we develop a different exact semantics, called the vectorized semantics, and show that \( \text{SemGuS} \) can be instantiated with this semantics as well. The vectorized semantics modifies the semantics of standard imperative programs to accomodate and execute multiple examples simultaneously in the form of vectors. This idea allows us to merge the examples, as well as the semantic premises \( \text{sem}_{\mathcal{N}}(\langle \Gamma, e_1 \rangle, \sigma_1), \ldots, \text{sem}_{\mathcal{N}}(\langle \Gamma, e_n \rangle, \sigma_n) \) of the Query rule, into a single semantic premise \( \text{sem}_{\mathcal{N}}(\langle \Gamma, \bar{e} \rangle, \bar{\sigma}) \), where \( \bar{e} \) and \( \bar{\sigma} \) represent the vectorized input-output examples.

The main challenge in defining a vectorized semantics is that, in the presence of loops and conditionals, different examples can cause a given loop to run a different number of times, and can take different branches of an if-statement. Here, we note that \( \text{SemGuS} \) is not the cause of these challenges, nor does it require the vectorized semantics; rather, \( \text{SemGuS} \) is what provides us with the possibility of defining different semantics that are better suited to solving the task at hand.

The three rules in Figure 3b present the big-step semantics for the terms \( x := e \) and while \( b \) do \( s \), the terms that are most relevant to overcoming these challenges. The most interesting rule here is WTrue. This rule states that as long as one of the examples in the vector makes the guard \( b \) true, the body of the loop should be entered. However, only the variable valuations that make the
guard true are updated in the loop-body $s$ (the $\text{proj}(\vec{r}, \vec{b})$ operator sets all valuations for which the guard is false to the special value $\bot$). The whole process is repeated (using the projected vector of valuations) until all entries of $\vec{b}$ are $\bot$, as stated in $W_{\text{False}}$. Finally, the vector of valuations in the bottom of the rule contains the $\text{merge}$ of valuations for which the guard was false, and valuations $\vec{r}_2$ that resulted from running the loop on the valuations $\text{proj}(\vec{r}, \vec{b})$ for which the guard was true.

The correctness of the vectorized semantics with respect to the standard semantics is stated and proved in Appendix B, Theorem B.2.

When supplying vectorized semantics to a $\text{SemGuS}$-with-examples problem, one should supply a single vectorized example that contains all the examples from the original example set. Aside from this difference in how examples should be supplied, the vectorized semantics can be treated just like any other semantics, meaning that the CHC-based solving procedure from §2 and §5 still holds. Moreover, as stated at the start of this section, the vectorized semantics illustrated above can be generated automatically for all subgrammars of $G_{\text{impv}}$, which allows it to be used as a general optimization for solving imperative $\text{SemGuS}$ problems (as our tool $M3SY$ does).

### 6.2 Using Abstract Semantics in $\text{SemGuS}$ to Prove Unrealizability

In this section, we show how the grammar $G_{\text{impv}}$ can be instantiated with an abstract semantics to prove the unrealizability of $\text{SemGuS}$ problems, following the idea introduced in §4.1.

There are many abstract semantics with which one can equip a language. Here, we use the abstract domain $B_i$ presented in §2.3 as an example, which tracks only the $i$-th bit of a variable using three values: true, false and $\top$ (the join of true and false). That $B_i$ is indeed an abstract domain is proved in Appendix B, Lemma B.3.

**Example 6.1.** Recall Equation (5), which represents the abstract semantics for a term $e_1 \& \& e_2$ from $G_{ex}$ of §2, using the abstract domain $B_3$. The right-hand side of the final premise describes the abstract semantic function $[\&\&]^a$ for the operator $\&\&$, which sends the computation to $\top$ if any of $v_1^a$ or $v_2^a$ are $\top$, and computes the exact value otherwise. Note how the semantic relations, as well as the structure of the semantic rule, remain unchanged—from the viewpoint of $\text{SemGuS}$, an abstract semantics expressed using CHCs is merely a different semantics supplied to $\text{SemGuS}$, for which one can apply the same solving procedure as given in §2 and §5.

Different abstract domains have different degrees of efficiency and precision in $\text{SemGuS}$. To see why, consider how one would deal with branches using the abstract domain described above. This particular abstract domain cannot handle comparisons well because it only tracks a single bit, and thus it is almost always the case that one does not know which branch to take in an if-statement. There are two possible approaches in this situation—one may just choose to assign $\top$ to the result of the branch, or one may try and execute both branches and assign their join to the result. This problem arises for both if-then-else statements and loops. As an example, two different rules for loop iteration are described in Example 6.2.

**Example 6.2.** Equations (13) and (14) each present different possible abstract semantics for the term while $b$ do $s$ from $G_{ex}$ of §2, using the abstract domain $B_3$, which tracks only the third bit of each variable.

\[
\begin{align*}
[b]^a(\Gamma^a, \nu_b^a) & \quad [s]^a(\Gamma^a, \Gamma_1^a) & \quad \text{while } b \text{ do } s]^a(\Gamma^a, \Gamma_2^a) & \quad \Gamma_r^a = \top & \quad W_{\text{True}}^a_{\text{Havoc}} \\
& \quad \text{[while } b \text{ do } s]^a(\Gamma^a, \Gamma_1^a) & \quad \text{[while } b \text{ do } s]^a(\Gamma^a, \Gamma_2^a) & \quad \Gamma_r^a = \text{JOIN}(\Gamma^a, \Gamma_1^a) & \quad W_{\text{True}}^a_{\text{Join}}
\end{align*}
\]

(13) (14)
In both scenarios, the value of $v_b$ will be $\top$ because knowing only the third bit does not give us enough information to resolve a condition of the term “$e \prec e$” from $G_{ex}$. In this situation, the rule $WTrue_{Havoc}$ simply gives up and assigns $\top$ to the resulting value $\Gamma_r$. On the other hand, the rule $WTrue_{Join}$ attempts to preserve some precision by assigning the join of when the condition evaluates to true ($\Gamma_2$, as the loop iterates in this case) and when the condition evaluates to false ($\Gamma'$, as the loop body does not execute). If both $\Gamma$ and $\Gamma_2$ contain $x^b = \text{true}$, then $WTrue_{Join}$ is capable of inferring that the result of while $b$ do $s$ also has $x^b = \text{true}$, while $WTrue_{Havoc}$ cannot.

The semantics expressed by $WTrue_{Join}$ is more precise and more expensive than the first option. For the example in §2, an abstract semantics using $WTrue_{Havoc}$ will fail to prove unrealizability of synthesizing bitwise-xor, because it cannot resolve the branch of the loop. On the other hand, the added precision from $WTrue_{Join}$ succeeds in proving unrealizability, showing how different abstract domains can solve different SemGuS problems.

### 6.3 Using Underapproximating Semantics in SemGuS for Program Synthesis

In this section, we demonstrate how SemGuS can be equipped with an underapproximating semantics to perform program synthesis, following the idea from §4.2. Example 6.3 shows an underapproximating semantics that sets a bound on the number of times each loop may be executed, as in bounded model checking [6]. The change to the semantics is simple—one simply adds a bound to the state and decreases the bound by one each time a loop iteration is performed.

**Example 6.3.** Equations (15) and (16) present an underapproximating semantics for the term while $b$ do $s$, where the number of loop iterations is bounded by a fresh variable $i$.

\[
\begin{align*}
[b]^b((\Gamma, i), \text{true}) & \quad i > 0 & [s]^b((\Gamma, i), (\Gamma', i)) & \quad \text{while } b \text{ do } s^b((\Gamma', i-1), (\Gamma_r, i-1)) & \quad WTrue^b \\
[\text{while } b \text{ do } s^b((\Gamma, i), (\Gamma_r, i))] & \quad WFalse^b \\
\end{align*}
\]

Equations (15) and (16) present an underapproximating semantics for the term while $b$ do $s$, where the number of loop iterations is bounded by a fresh variable $i$.

One can see how these rules are underapproximating by considering why one is unable to build a proof tree for a loop that must execute more iterations than the unrolling bound. For example, let the unrolling bound be $i = 1$. To prove that $[\text{while } b \text{ do } s]^b((\Gamma', 1), (\Gamma_r, 1))$, i.e., the conclusion with $i = 1$, one would also require a proof for the final premise in the rule, namely $[\text{while } b \text{ do } s]^b((\Gamma', 0), (\Gamma_r, 0))$. However, a proof of $[\text{while } b \text{ do } s]^b((\Gamma', 0), (\Gamma_r, 0))$ requires that $0 > 0$ due to the third premise $i > 0$, which is unsatisfiable. Thus, nothing can be proved about $[\text{while } b \text{ do } s]^b((\Gamma', 0), (\Gamma_r, 0))$—corresponding to the fact that $[\text{while } b \text{ do } s]^b((\Gamma', 0), (\Gamma_r, 0))$, and any relations that rely on this premise, are undefined.

In contrast, the semantics described by Equation (15) match exactly the standard semantics of a while loop for a loop that executes fewer iterations than the unrolling bound.

That the semantics presented in Example 6.3 is an underapproximating semantics is proved in Appendix B, Lemma B.4. The constraints that make a semantics underapproximating—for example, $i > 0$ in Example 6.3—can be encoded in the constraint element $\phi$ of a CHC.

### 7 IMPLEMENTATION AND OPTIMIZATIONS

In this section, we describe our implementation of M3SY, a solver for SemGuS problems, as well as some optimizations that were applied in M3SY.
7.1 Implementing M₃₂SY

At a high level, M₃₂SY accepts SEMGuS problems and encodes them as CHCs using the encoding in §5. It then passes the CHCs to Z₃ [8], which performs the actual proof search and produces an answer. The output from Z₃ is either UNSAT, which means the problem is unrealizable, or a proof for Realizable from Equation (8) using the inference rules from the SEMGuS problem: in this case, M₃₂SY can extract a solution to the SEMGuS problem from the proof.⁸ We note that the capability of M₃₂SY to synthesize programs also allows it to perform CEGIS for both program synthesis and unrealizability, which is unsupported in previous work on proving unrealizability [11, 12].

We report here that Z₃ itself varied in performance depending on whether particular internal flags were enabled.⁹ While enabling these flags are the default setting for Z₃ and result in better performance, they also made it difficult to recover the term representation from the output of Z₃ (which is required to synthesize a term). Thus, during our evaluation in §8, we disabled the flags; M₃₂SY can also be configured to run with the flags enabled.

In §5, we looked at different ways of translating SEMGuS problems into CHCs depending on whether trees or listings are used to represent terms. M₃₂SY supports three configurations for representing terms—a configuration that uses algebraic datatypes to model trees, and two configurations that respectively use lists and arrays to encode listings. In addition, M₃₂SY also implements a SEMGuS-specific optimization called the fused semantics, described in the next section.

7.2 Optimizing Imperative SEMGuS Problems with Fused Semantics

M₃₂SY offers an optimization that utilizes a slightly different method of encoding syntax and semantic rules: instead of building a term using the syntax rules and propagating it through the semantic rules separately, one can also think of a scheme where the semantics of a term is executed on-the-fly while the term is being constructed. We refer to this kind of encoding as the fused semantics. Fused semantics are different from supplying SEMGuS with a different semantics, because they are derived from an original semantics that SEMGuS is supplied with. Instead, one may think of them as an optimization for SEMGuS problems over subgrammars of G_{impv}.

The key idea for fused semantics is to modify the syntax relations so that they can check semantics as well as the syntactic structure, and modify the syntax rules accordingly as well. Thus, a syntax relation is now defined over three inputs—a term t, an input state Γ, and an output value v. The relation should evaluate to true if and only if t is a valid term, and [Γ] t v is also true. Generally, the syntax rule for a production A₀ → σ(A₁, · · · , Aᵢ) ∈ A, again equipped with the semantics ϕ ∧ semAᵢ(⟨Γ₁, t₁⟩, v₁), · · · , semAᵢ(⟨Γᵢ, tᵢ⟩, vᵢ) ⇒ semA₀(⟨Γ, t⟩, v₀), can be generated in the form of Equation (17): the structure of Equation (17) matches exactly the structure of the supplied semantics.

\[
\frac{\phi \land \text{syn}^{\text{fused}}_{A_i}(⟨Γ_1, t_1⟩, v_1) \cdots \text{syn}^{\text{fused}}_{A_i}(⟨Γ_i, t_i⟩, v_i)}{\text{syn}^{\text{fused}}_{A_0}(⟨Γ, σ(t_1, \cdots, t_i)⟩, v)} \quad \text{syntax}^{\text{fused}}_{A_0→σ(A_1, \cdots, A_i)} \in A
\]  

(17)

The new encoding presented in Equation (17) is enough to allow only the syntax rules to describe both the syntax and semantics of terms within a SEMGuS problem, provided that the grammar does not contain productions with while loops. However, productions that contain loops, such as N → while B do S ∈ A, require a separate procedure because there must be a guarantee that the same loop body is synthesized for each iteration. To ensure that the same loop body is synthesized, one can either impose an additional constraint that states that each synthesized loop body must be

---

⁸ Z₃ may also time out, or produce an error for various reasons, for example when dealing with algebraic datatypes.

⁹ The particular flags are fp.xform.slice, fp.xform.inline_linear, and fp.xform.inline_eager.
identical, or more simply, one can apply the semantic relations described from §5 instead.

\[
\begin{align*}
\text{syn}^\text{fused}_B((\Gamma, b), \text{true}) & \quad \text{syn}^\text{fused}_N((\Gamma, s), \Gamma_1) & \quad \text{sem}_N((\Gamma_1, \text{while } b \text{ do } s), \Gamma_2) \\
\text{syn}^\text{fused}_N((\Gamma, \text{while } b \text{ do } s), \Gamma_2) & \quad \text{syn}^\text{fused}_N \rightarrow \text{while } B \text{ do } S
\end{align*}
\]

(18)

Consider the rule given in Equation (18). Note that the first two relations from the premise are syntax relations that both synthesize a term and execute its semantics. In contrast, the third relation is a semantic relation, which is defined identically to the semantic relations in §2. The semantic relations do not suffer from the problem of having to synthesize the same loop body over multiple iterations. The idea here is that the syntax relations synthesize the loop body on the first iteration, then pass the representation to the semantic relations for subsequent iterations.

Finally, multiple \( \text{sem}_N \) premises in the Query rule must be rewritten as \( \text{syn}^\text{fused}_N \) as well; this raises the same problem of potentially synthesizing different solutions for each example. Again, one must impose a constraint that all generated representations are identical. Instead of imposing this constraint directly, M32SY employs the fused semantics as an optimization to vectorized semantics only, which does not suffer from this problem because there is only a single vector of examples.

The soundness and completeness for our algorithm using the fused-semantics optimization is proved in Appendix B, Theorem B.5.

8 EVALUATION

In this section, we evaluate the feasibility of our algorithm to solve SemGuS problems through our implementation M32SY. Specifically, we investigate the following four issues:

Q1: We evaluate the effectiveness of M32SY on SyGuS benchmarks.
Q2: We evaluate the effectiveness of M32SY on imperative program-synthesis problems.
Q3: We evaluate the effectiveness of the optimizations discussed in §5 and §7.
Q4: We evaluate the effectiveness of approximate semantics supplied to M32SY.

Overall, our evaluation is tilted towards proving unrealizability compared to synthesizing programs. This is because because there already exist many program synthesizers that incorporate multiple years of engineering effort [4, 22]; it is beyond the scope of this paper and M32SY to directly compete with these synthesizers.

8.1 Benchmarks

We performed our evaluation using two sets of benchmarks.

The first set consists of 132 unrealizable variants of the 60 LIA (Linear Integer Arithmetic) benchmarks from the LIA SyGuS competition track. These benchmarks were generated by Hu et al. [11], and have been used as benchmarks for unrealizability in previous work [11, 12]. In each of the benchmarks, the grammar that specifies the search space is recursive, and hence generates infinitely many LIA terms. These benchmarks are unrealizable because they contain grammars that restrict how many times a certain operator (e.g., plus or if-then-else) can appear in the solution. To see how effective M32SY is as a synthesizer, we also test M32SY on the 60 original LIA SyGuS benchmarks in §8.2. These benchmarks have a completely unrestricted grammar, as opposed to the 132 unrealizable variants generated from them.

The second set consists of 289 imperative SemGuS problems defined over various fragments of the imperative grammar \( G_{impv} \). Out of these, 36 benchmarks were created by hand from common imperative programming questions, such as synthesizing a Fibonacci function or swapping variables using bitwise-xor. The remaining 253 benchmarks were derived by using the 30 benchmarks employed in a previous paper on synthesizing imperative programs via enumeration [21] as a
Table 1. Number of solved benchmarks for various configurations of M32SY, alongside results for Nay, ESolver, CVC4, and SIMPL. ✓ indicates cases where the tool is non-applicable. SIMPL could only be evaluated on 23 realizable imperative benchmarks, because SIMPL cannot accept a grammar.

| Solver        | SyGuS Realizable | SyGuS Unrealizable | Imperative Realizable | Imperative Unrealizable |
|---------------|------------------|--------------------|-----------------------|-------------------------|
| Nay           | ✓                | 70                 | ✓                     | ✓                       |
| ESolver       | 6                | ✓                  | ✓                     | ✓                       |
| CVC4          | 59               | ✓                  | ✓                     | ✓                       |
| SIMPL         | ✓                | ✓                  | ✓                     | ✓                       |
| Total         | 4                | 66                 | 8                     | 112                     |
| Fused Trees   | ✓                | 66                 | ✓                     | ✓                       |
| Fused Lists   | 4                | 66                 | 5                     | 31                      |
| Fused Arrays  | 2                | 64                 | 6                     | 91                      |
| Non-Fused Arrays | 2           | 66                 | 5                     | 56                      |
| Individual    | 0                | 56                 | 3                     | 10                      |
| Abstract      | ✓                | ✓                  | ✓                     | ✓                       |
| Underapproximate | ✓                | ✓                  | ✓                     | ✓                       |
| Total benchmarks | 60            | 132                | 67                    | 222                     |

template. Out of the 30 templates, we ignored 7 that contained division, on which Z3 would return an error, and derived 11 benchmarks from each of the 23 remaining templates for a total of 253 benchmarks. The 23 base templates consist largely of two categories: those that compute a function over a range of numbers 1 to n using a loop (such as factorials or sums), and those that compute a function over an array, again using a loop to iterate (such as finding the maximum element of an array, or adding two arrays together). To derive our benchmarks, we first instantiated SemGuS with the problem specification and the unbounded grammar $G_{imp}$ with a restriction on the number of loops: the grammar in this case replicates the templates used to specify the search space from [21]. Then, various restrictions were imposed on the grammar, such as limiting the number of statements allowed, or limiting the kinds of expressions that can occur as the loop condition. Out of the 11 benchmarks generated from each template, 2 were designed to be realizable, and 9 to be unrealizable. We developed our own set of benchmarks this way because the unrealizability of imperative programs is a previously unstudied field.

Each benchmark was given 10 minutes to complete on a machine with a 2.6GHz Intel Xeon processor with 32GB of RAM, with version 4.8.9 of Z3 as the external CHC solver. We note that the front-end processing step, to encode a SemGuS problem into CHCs, took less than 3 minutes for all of our benchmarks and configurations combined; the 10-minute timeout was separate from the front-end processing step, and devoted entirely to CHC solving.

Table 1 summarizes the numbers of solved benchmarks for various configurations of M32SY we tested, as well as comparisons for Nay [12], ESolver [3], CVC4 [4], and SIMPL [21].

8.2 Evaluating M32SY on SyGuS Benchmarks

In this section, we evaluate the effectiveness of M32SY on SyGuS benchmarks by comparing it against Nay, the state-of-the-art tool for proving unrealizability for SyGuS problems, and against the SyGuS synthesizers ESolver and CVC4.

Like M32SY, Nay checks the (un)realizability of a SyGuS problem when the specification is given as a set of examples; unlike M32SY, however, Nay cannot synthesize a solution for realizable problems. We used the set of 132 unrealizable SyGuS benchmarks described in §8.1 for evaluation. We
report that M32SY or NAY solves a problem if it can solve a problem using any of its configurations—
for M32SY, this encompasses the three different term representations, as well as the different
semantics that SemGuS can be instantiated with.

We implemented a CEGIS algorithm for M32SY, and compared it against the CEGIS algorithm
of NAY. Because NAY is incapable of synthesizing an answer to realizable a SyGuS problem, the
CEGIS loop of NAY relies on an external synthesizer ESolver to produce a term. Because M32SY
and NAY rely on different methods to produce counterexamples, their CEGIS iterations may differ.

With the standard CEGIS algorithm, M32SY can prove 61/132 benchmarks unrealizable, while
NAY can do so for 65/132. NAY also provides a modified “random” variant of CEGIS that allows
random examples to be added to the set of counterexamples throughout the CEGIS loop. Using this
technique, NAY can prove unrealizability for an additional 5 benchmarks. We ran M32SY on the
same set of examples produced using this technique; it was able to prove unrealizability for the
same 5 benchmarks as well, for a total of 66/132 benchmarks. There are 67 benchmarks where both
solvers timed out—stuck at some iteration of the CEGIS loop. On 17 of them, M32SY can complete
more iterations of the CEGIS loop than NAY (avg. 6.1 for M32SY vs. 4.8 for NAY). On 13 of them,
NAY can progress further (avg. 2.2 for M32SY vs. 3.1 for NAY). The two solvers were stuck at the
same iteration on the rest of the benchmarks. Table 2 in Appendix A presents a detailed comparison
of the runtimes for each tool, on benchmarks from the LIMITEDIF and LIMITEDPLUS categories.

Next, we compared the abilities of M32SY as a SyGuS synthesizer on the 60 original LIA bench-
marks upon which the unrealizable benchmarks were derived. M32SY succeeded in solving 4/60
benchmarks. This is comparable to the initial version of ESolver, which solved 6/60 benchmarks and
was the winner of the first SyGuS competition in 2014. Moreover, M32SY solved one benchmark
that ESolver could not solve. While M32SY is not competitive with current SyGuS solvers, such as
CVC4 [4], which solved 59/60 benchmarks, the fact that its performance is already comparable
with an early version of a SyGuS solver is encouraging, and one might hope that more efficient
algorithms for synthesizing solutions to SemGuS problems are possible in the future.10

M32SY was efficient at proving unrealizability: 56 out of the 66 benchmarks solved were solved
in under 10 seconds, and Table 2 shows that M32SY also has comparable runtimes with NAY. For
synthesizing programs, ESolver solved all solvable benchmarks in under two minutes each, while
M32SY required 6 minutes for two of the solved benchmarks.

To answer Q1: M32SY is quite effective on unrealizable SyGuS problems, to a degree that is
comparable with NAY, and can also synthesize solutions for realizable SyGuS problems: in particular,
M32SY is more general than previous tools as it can solve non-SyGuS problems, and can produce
two-sided answers to synthesis problem.

8.3 Q2: Evaluating M32SY on Imperative Synthesis Benchmarks
In this section, we evaluate the effectiveness of M32SY by seeing how well it can deal with
imperative synthesis benchmarks. We consider SemGuS-with-examples problems, as opposed to
ordinary SemGuS problems, due to the challenges of checking whether an imperative program
satisfies a specification or not (which makes it difficult to implement a CEGIS loop). In principle,
one could implement a CEGIS loop using an external verifier.

Out of our 289 imperative benchmarks, a total of 67 were designed to be realizable, while the
remaining 222 were designed to be unrealizable. As shown in Table 1, M32SY solved 8/67 realizable
benchmarks, and 112/222 unrealizable benchmarks, for a total of 120 benchmarks solved.

10 We also note that the LIA SyGuS benchmarks have an entirely free grammar and are single-invocation, which allows
CVC4 to use a specialized method involving quantifier elimination to synthesize programs.
Out of the 120 solved benchmarks, 10 benchmarks were those with infinite syntactic search spaces and also contained the possibility of an infinite loop. M₃₂SY also solved 15 benchmarks that did not contain loops, but nevertheless had infinite syntactic search spaces.

Overall, M₃₂SY had more success with proving unrealizability than synthesizing programs for both SyGuS and imperative benchmarks. This difference is in part due to how the generated CHCs are dealt with internally in Z3—as described in §2, Z3 proves unrealizability by discovering a lemma that conflicts with the specification. For realizable problems, however, Z3 in the worst case must conduct a search over all possible concrete terms from a possibly infinite search space, in a process similar to generate-and-test. The authors are unsure of whether Z3 is capable of discovering lemmas that can be used to prune the search space for realizable benchmarks; regardless of the answer, the results suggest that the added overhead from expressing semantics as CHCs in an SMT solver is large enough to make synthesis relatively more difficult compared to proving unrealizability.

In contrast, SIMPL [21], whose benchmarks we use in our evaluation of M₃₂SY, employs a strategy of performing static analysis in tandem with enumeration; SIMPL also employs heuristics that prefer smaller programs, and directly executes candidates to see if the specification is met. This enumerative approach makes SIMPL perform better as a synthesizer: SIMPL solves the full set of 23 realizable benchmarks upon which our benchmarks are based, while M₃₂SY can solve none. However, SIMPL is incapable of proving unrealizability, because it is based on enumeration. One also cannot express a syntactic search space in SIMPL outside of simple templates, which prevented us from running the rest of our realizable benchmarks on SIMPL.

M₃₂SY took less than 10 seconds to solve 82 of the 120, and the other 12 benchmarks required more than a minute to complete. Interestingly, whether a benchmark contained an unbounded loop or an infinite search space seemed to have little correlation with the runtimes: there were finite-search space benchmarks that took over a minute to complete, and benchmarks with both unbounded loops and infinite search spaces that took less than a second to complete. This phenomenon suggests the importance of discovered lemmas in solving SemGuS problems: given a powerful lemma, a SemGuS problem can be solved quickly, even in the presence of infinite loops and search spaces. On the other hand, without such a lemma, the problem can take a long time to solve even if the search space is finite.

To answer Q2: M₃₂SY is capable of solving SemGuS problems with infinite search spaces and imperative semantics, especially if the given problem is unrealizable. Notably, M₃₂SY is the first tool that can prove unrealizability for imperative synthesis problems.

8.4 Q3: Evaluating Optimized Methods for Solving SemGuS Problems

In this section, we compare the effectiveness of the various optimizations we described in §5 and §7 for solving SemGuS problems. Specifically, we investigate the following two issues:

1. We assess the effectiveness of the flattened term representation from §5.2, by comparing the performance of M₃₂SY configured to use trees, lists, and arrays as the term representation.
2. We assess the effectiveness of vectorized and fused semantics, by comparing the performance of M₃₂SY on (i) individual semantics, (ii) vectorized but non-fused semantics, and (iii) vectorized and fused semantics.

Effectiveness of Flattened Term Representations. To evaluate the effectiveness of the three term representations, we supplied SemGuS with vectorized semantics and enabled the fused-semantic-optimization, the configuration that yielded the overall best results in our evaluation. In this section, we say that a particular term representation “solved” a benchmark if Z3 was able to solve the CHCs produced by encoding the SemGuS problem using the given term representation.
Fig. 4. Runtime comparisons for various configurations of M32SY. Bar graphs on the outer axes show the distribution of the data points. 600 seconds indicates a timeout.

The first three rows of Table 1 summarize the results for the different term representations: for SyGuS benchmarks, all three representations were similar. For imperative benchmarks, the array representation is clearly better compared to the list and tree representations: in particular, the tree representation could only solve one benchmark that the array representation could not solve, while the list representation solved a strict subset of the benchmarks solved by the array representation.

Figures 4a and 4b compare the performances of the list versus tree representations, and the list versus array representations, on imperative benchmarks solved by at least one of the representations. As mentioned in §5.2, we suspect the differences between the different term representations is due to the fact that support for algebraic datatypes in Z3 still remains relatively limited. When the tree or list representations were used, Z3 terminated with the error “stuck on a lemma” far more often than when the array representation was used.

To answer part (1) of Q3: Flattened term representations are indeed effective, especially when using arrays to avoid the use of algebraic datatypes altogether.

Effectiveness of the Vectorized and Fused Semantics. To evaluate the effectiveness of the vectorized and fused semantics optimizations, we compared them against each other, and against individual semantics that are not vectorized nor fused (corresponding to the “standard” form of semantics mentioned in §6). We note that while the vectorized semantics is actually a different semantics that SEMGU$\S$ can be supplied with, M32SY can automatically vectorize the semantics for any subgrammar of $G_{impv}$ as an optimization; thus we treat it as an optimization for our evaluation.

Individual semantics vs. Non-fused vectorized semantics. We first make a quick comparison between the individual and non-fused, vectorized semantics. Using a list representation of terms, the individual semantics was able to solve 56 unrealizable SyGuS benchmarks, a strict subset of the 66 solved by the vectorized benchmarks. For imperative benchmarks, the individual semantics could only solve 3 realizable and 10 unrealizable benchmarks, compared to 5 realizable and 62 unrealizable when using the vectorized semantics.

In our correspondence of the authors of Z3 and Spacer, they mentioned that using inductive datatypes with the Horn Clause solver was highly experimental.
Non-fused vectorized semantics vs. Fused vectorized semantics. Next, we compare the performance of the non-fused vectorized and fused vectorized semantics. The "Fused Arrays" and "Non-Fused Arrays" rows of Table 1 describe the number of solved benchmarks: again, performance for the SyGuS benchmarks was similar. For imperative benchmarks, the non-fused vectorized array semantics solved 5 realizable and 56 unrealizable benchmarks, compared to 6 realizable and 91 unrealizable for the fused vectorized array semantics.

Figure 4c compares the results of the non-fused vectorized versus the fused vectorized semantics using array representations: while the fused semantics solve more benchmarks, the graph suggests that the fused semantics are not strictly better than the non-fused semantics. In particular, there are 11 unrealizable benchmarks that only the non-fused semantics can solve.

When using a list representation, the difference becomes less pronounced—the non-fused vectorized semantics can solve 4 realizable and 58 unrealizable benchmarks, compared to 5 realizable and 62 unrealizable for fused vectorized list semantics. We think the reason is again the limited support in Z3 for algebraic datatypes, which remains the main bottleneck when using list representations.

To answer part (2) of Q3: The fused vectorized semantics is effective as an optimization, especially for imperative benchmarks, but there exist some benchmarks for which the non-fused vectorized semantics performs better. Both are consistently better than the individual semantics.

8.5 Q4: Evaluating SemGuS and M32SY with Approximate Semantics

Finally, we evaluated how well M32SY performs when it is instantiated using an approximate semantics to produce one-sided answers to either synthesis or unrealizability. In this section, we focus on identifying the number of new benchmarks that an approximate semantics can solve compared to the standard version of the semantics. This approach is motivated by the fact that the nature of an approximate semantics can sometimes change the kind of answer that can be obtained—for example, using an abstract semantics for an unrealizable problem might make it realizable—and thus a direct performance comparison makes little sense.

Both an abstract and underapproximating semantics were implemented using arrays as the term representation, with the fused-semantics and vectorizing optimizations, which displayed the best overall performance in §8.4. The new benchmarks solved are ones that the exact, fused and vectorized array semantics was unable to solve.

Abstract semantics for unrealizability. To test the capabilities of abstract semantics, we implemented five variants of the abstract domain $\mathbb{B}_3$ from §6.2, where each domain tracked the first, second, third, fourth, and fifth bit of variables, respectively. This choice was driven by the fact that most of the input examples for our imperative synthesis benchmarks were small, between 0 to 31.

The "Abstract" row of Table 1 describes the number of benchmarks solved using the abstract semantics. The abstract semantics did not make a difference for the SyGuS benchmarks; all benchmarks that were solved by the abstract semantics were also solvable by the exact semantics. However, for the imperative benchmarks, the abstract semantics was able to solve 17 unrealizable benchmarks that the exact semantics could not solve.

Using abstract semantics also yielded faster runtimes: the abstract semantics timed out for less than 15 benchmarks from the entire suite, compared to over 200 for the exact semantics (although realizability results in the abstract semantics have no meaning).

One reason that the abstract semantics failed to make a difference on the SyGuS benchmarks could be that the SyGuS benchmarks themselves used inputs with very small values, often between 0 and 7: the abstract semantics were able to prove some SyGuS benchmarks as unrealizable using the lower bits, but nothing new. In addition, the abstract domains that we used did not work well
in the presence of addition, because carry bits often render a result to be \( \top \). All of our SyGuS benchmarks contain addition, while some imperative benchmarks do not.

Underapproximating semantics for program synthesis. For the underapproximating semantics, we implemented the technique of bounding the number of loop-unrollings from §6.3, and experimented with loop bounds of 10, 50, and 100. We only compare the imperative benchmarks here, because the SyGuS benchmarks do not contain loops.

The “Underapproximate” row of Table 1 describes the number of benchmarks solved using this semantics. The bound semantics was able to synthesize one more program compared to the non-bound semantics. Interestingly, \( \mathcal{M} \mathcal{Z} \mathcal{S} \mathcal{Y} \) succeeded in synthesizing the program (to compute the factorial function using a while loop) when the bound was set to 100, but not when the bound was 10 or 50. The small difference in performance may be due to the fact that \( \mathcal{M} \mathcal{Z} \mathcal{S} \mathcal{Y} \) generally performs worse as a synthesizer than a tool for proving unrealizability. The results also tell us that synthesizing imperative programs is difficult, even without the presence of infinite loops: it could be because unrolled loops still pose a significant burden when trying to compute the semantics of an imperative program, especially because our approach must ultimately prove that a candidate term satisfies the specification using semantics encoded as CHCs (which is likely to be slower compared to direct execution).

To answer Q4: Abstract semantics allows \( \mathcal{M} \mathcal{Z} \mathcal{S} \mathcal{Y} \) to solve many more unrealizable \( \mathcal{S} \mathcal{E} \mathcal{M} \mathcal{G} \mathcal{U} \mathcal{S} \) problems compared to using only exact semantics. The bound underapproximating semantics did allow \( \mathcal{M} \mathcal{Z} \mathcal{S} \mathcal{Y} \) to solve more realizable \( \mathcal{S} \mathcal{E} \mathcal{M} \mathcal{G} \mathcal{U} \mathcal{S} \) problems, but the improvement was small.

9 RELATED WORK

General Synthesis Frameworks. Sketch [22] and Rosette [24] are both solver-aided languages, where one specifies a synthesis problem using a domain-specific language, which is translated into an SMT problem. FlashMeta [19] is a synthesis framework that allows one to specify the semantics of operators in the language using witness functions, which roughly correspond to the “inverse” semantics of operators. In these tools, the way synthesis problems are defined is directly tied with how they are solved: one needs to develop non-standard inverse semantics for FlashMeta, or phrase the synthesis problem within the language of Sketch or Rosette, which are requirements imposed by their particular synthesis algorithms. Due to these reasons, these tools disallow defining (and therefore solving) synthesis problems involving infinite search spaces.

The first attempt to unify these frameworks into a logical one was provided by SyGuS [1]. However, SyGuS is not general enough as it cannot express synthesis problems over arbitrary syntactic constructs that do not lie inside a decidable SMT theory. \( \mathcal{S} \mathcal{E} \mathcal{M} \mathcal{G} \mathcal{U} \mathcal{S} \), on the other hand, provides a logical way to define synthesis problems with custom semantics. Moreover, the solving procedure for \( \mathcal{S} \mathcal{E} \mathcal{M} \mathcal{G} \mathcal{U} \mathcal{S} \) is motivated by the definition, not the other way around.\(^{12}\)

Customizing Semantics in SyGuS. SyGuS allows one to provide semantics for user-defined terms, but the support is limited to functions/operators that can be used in the grammar. Concretely, if we consider our formalization of semantics (Definition 3.5), the degree of customization available in SyGuS is limited to customizing the constraint \( \phi \) to a formula expressible in the background theory.\(^{13}\) This limitation prevents SyGuS from expressing imperative program-synthesis problems;

---

\(^{12}\) One may argue that semantics expressed as a CHC is a restriction, but as stated in §1, they are more of a formalization. One may also assume a different surface syntax, such as the format in Equation (1); a translation to CHCs is straightforward.

\(^{13}\) To be precise, this formula is further limited to a formula of the form \( v_0 = f(v_1, \ldots, v_i) \), where \( f \) is a non-recursive function expressible in the background theory. Notably, this prevents expressing relations between \( v_1, \ldots, v_i \), the results of nonterminals in the RHS.
SemGuS eases this restriction by allowing one to replace $\phi$ with any first-order formula, as well as introduce new relations or arguments for the relations.

Synthesis for Imperative Programs. There have been attempts at designing synthesizers specifically for imperative programs. Existing tools require the user to provide templates that specify most of the program [22, 23]; the tools then resort to various constraint-solving techniques to complete missing parts of the template, which often do not contain loops [23].

SIMPL [21] can synthesize imperative programs from input-output examples and a template that specifies most of the program. SIMPL employs a simple enumeration-based strategy, and uses abstract interpretation to rule out templates that will not result in a solution. Because SIMPL is based on enumeration, it performs well as a synthesizer. However, in contrast to $M\exists SY$, SIMPL cannot restrict the terms allowed in a program and it cannot establish that a problem is unrealizable.

Unrealizability. Nope [11] and Nay [12] are, to the best of our knowledge, the only two tools that can prove unrealizability for SyGuS benchmarks in which the grammar can generate infinitely many terms. Because Nay consistently outperforms Nope, we only compare against Nay in our evaluation. $M\exists SY$ can solve synthesis problems over any specified language, including imperative languages, whereas both Nope and Nay can only solve SyGuS problems. One variant of Nay also uses Constrained HornClauses, which are used to encode the problem of solving a set of equations that describes the sets of possible outputs of the program. In $M\exists SY$, the constraints are used for describing both the syntax and the semantics of the programs in the search space. Because of the syntactic constraints, $M\exists SY$ can extract the synthesized program when the problem is realizable, which Nay is unable to do.

There exist other tools that are capable of proving unrealizability in limited situations, such as CVC4 [4] or DryadSynth [2]. However, CVC4 can only prove unrealizability when the grammar is completely unrestricted, and DryadSynth does not accept a grammar as part of its specification; $M\exists SY$ is the only tool that can perform synthesis and unrealizability for general SemGuS problems.

The Use of Semantics in Program Synthesis. Synthesis using abstraction refinement (SYNGAR) [25] uses predicate abstraction to prune the search space of a synthesis-from-examples problem. SYNGAR builds a tree automaton representing all trees in the search space that are correct with respect to an abstract semantics expressed using predicate abstraction. SYNGAR can be viewed as a special case of SemGuS in which predicate abstraction is used to overapproximate the semantics of terms in the programming language. SYNGAR’s approach is tied to the use of an abstract semantics that can be expressed using a finite abstract domain, whereas our approach extends to infinite domains. In particular, with our approach, one can express the concrete semantics of a programming language.

FlashMeta [19], is also a way of using semantics in program synthesis.

The Use of Horn Clauses in Program Synthesis. In Inductive Logic Programming (ILP) [15, 17, 20], given background knowledge, typically in the form of Horn Clauses, the goal of ILP is to learn the defining formula for a logical relation that agrees with a given classification of input examples.

Both ILP and our framework use HornClauses to specify background knowledge—which for our algorithm consists of the syntax and semantics of the target programming language. However, the respective goals for the output answer are different: (i) In ILP, the goal is to create a Horn-Clause program as the answer. (ii) In our algorithm for SemGuS, the goal is to create a program in the language that has been specified via the background knowledge. Whether ILP techniques can be adapted to SemGuS is left for future work.

10 CONCLUSION

This paper develops a new framework SemGuS for program synthesis that allows one to specify both the syntax and the semantics of a synthesis problem. In particular, SemGuS can be used for
specifying synthesis problems over an imperative programming language; it also allows one to work with a variety of different semantics that may be better suited to solve a synthesis problem efficiently. The paper also presents a general procedure for solving SemGuS problems that is capable of both program synthesis and proving unrealizability, and an implementation MÃżSY to solve SemGuS problems.

SemGuS opens many future directions of work. For example, how can we explicitly prune search spaces for synthesis problems with lemmas? As mentioned in §2, our procedure for solving SemGuS problems relies on an external CHC solver to infer lemmas over sets of programs in the syntactic search space, using the semantics of terms. While we have relied on an external solver (Z3) to perform this inference for us, it is also unclear to what degree CHC solvers are capable of discovering lemmas to prune a syntactic search space. An algorithm to explicitly infer lemmas and prune parts of the search space would be especially useful in enhancing our solving algorithm as a synthesizer, allowing MÃżSY to compete with state-of-the-art synthesizers as well.

REFERENCES

[1] Alur, R., Bodik, R., Juniwal, G., Martin, M. M., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. Syntax-guided synthesis. In Formal Methods in Computer-Aided Design (FMCAD), 2013 (2013), IEEE, pp. 1–8.

[2] Alur, R., Pismans, D., Singh, R., and Solar-Lezama, A. Sygus-comp 2017: Results and analysis. arXiv preprint arXiv:1711.11438 (2017).

[3] Alur, R., Radhakrishna, A., and Udupa, A. Scaling enumerative program synthesis via divide and conquer. In International Conference on Tools and Algorithms for the Construction and Analysis of Systems (2017), Springer, pp. 319–336.

[4] Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., and Tinelli, C. Cvc4. In International Conference on Computer Aided Verification (2011), Springer, pp. 171–177.

[5] Blanc, R., Gupta, A., Kovács, L., and Kragl, B. Tree interpolation in vampire. In International Conference on Logic for Programming Artificial Intelligence and Reasoning (2013), Springer, pp. 173–181.

[6] Clarke, E., Kroening, D., and Yorav, K. Behavioral consistency of c and verilog programs using bounded model checking. In Proceedings 2003. Design Automation Conference (IEEE Cat. No. 03CH37451) (2003), IEEE, pp. 368–371.

[7] Cousot, P., and Cousot, R. Abstract interpretation frameworks. Journal of logic and computation 2, 4 (1992), 511–547.

[8] De Moura, L., and Bjørner, N. Z3: An efficient SMT solver. In Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Berlin, Heidelberg, 2008), TACAS’08/ETAPS’08, Springer-Verlag, pp. 337–340.

[9] Feser, J. K., Chaudhuri, S., and Dillig, I. Synthesizing data structure transformations from input-output examples. ACM SIGPLAN Notices 50, 6 (2015), 229–239.

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

[11] Hu, Q., Breck, J., Cyphert, J., DâĂŹAntoni, L., and Reps, T. Proving unrealizability for syntax-guided synthesis. In International Conference on Computer Aided Verification (2019), Springer, pp. 335–352.

[12] Hu, Q., Cyphert, J., DâĂŹAntoni, L., and Reps, T. Exact and approximate methods for proving unrealizability of syntax-guided synthesis problems. In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (2020), pp. 1128–1142.

[13] Hu, Q., and DâĂŹAntoni, L. Syntax-guided synthesis with quantitative syntactic objectives. In International Conference on Computer Aided Verification (2018), Springer, pp. 386–403.

[14] Komuravelli, A., Gurfinkel, A., and Chaki, S. Smt-based model checking for recursive programs. Formal Methods in System Design 48, 3 (2016), 175–205.

[15] Lavrač, N., and Džeroski, S. Inductive Logic Programming: Techniques and Applications. Ellis Horwood, 1994.

[16] McMillan, K. L., and Rybalchenko, A. Solving constrained horn clauses using interpolation. Tech. Rep. MSR-TR-2013-6 (2013).

[17] Muggleton, S. Inductive logic programming. New Generation Comp. 8, 4 (1991), 295–317.

[18] Phothilimthana, P. M., Elliott, A. S., Wang, A., Jagda, A., Hagedorn, B., Barthels, H., Kaufman, S. J., Grover, V., Torlak, E., and Bodik, R. Swizzle inventor: data movement synthesis for gpu kernels. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (2019), pp. 65–78.
A DETAILED COMPARISON OF M32SY AND NAY ON UNREALIZABLE SYGUS BENCHMARKS

Table 2 presents a detailed comparison of the runtimes for M32SY and Nay on the SyGuS benchmarks from the LIMITEDIf and LIMITEDPLUS categories. Both solvers could easily solve all the benchmarks in the easier LIMITEDCONST category with comparable running times (less than 1 second on most benchmarks).
Table 2. Performance of M Ś 2 SY and NAY on a selected set of benchmarks from the LimitedIf and LimitedPlus categories, which are the benchmarks that best highlight the differences between the tools (other benchmarks were quickly solved by both tools). CEGIS indicates the performance when using the CEGIS algorithm (with an external synthesizer for NAY) while Oracle Examples describes the performance of the two solvers on a predefined set of examples for which the problem is known to be unrealizable. The table shows the number of nonterminals (|N|), productions (|δ|), and variables (|V|) in the problem grammar; the number of examples produced in the CEGIS loop (|E|); and the total running time of M Ś 2 SY and NAY. ✓ denotes a timeout.

| Problem  | Grammar | CEGIS | Oracle Examples |
|---------|---------|-------|----------------|
|         | | NAY | M Ś 2 SY | NAY | M Ś 2 SY |
| | | | | | |
| guard1 | 7 24 3 | 2 | 1.56 | 2 | 5.37 | 0.41 | 0.2 |
| guard2 | 9 34 3 | 3 | 19.27 | 3 | 26.12 | 21.51 | 1.12 |
| guard3 | 11 41 3 | 1 | 1.3 | 1 | 4.46 | 0.07 | 2.88 |
| guard4 | 11 72 3 | 2 | ✓ | 2 | ✓ | 72.58 | 1.93 |
| plane1 | 2 5 2 | 1 | 1.28 | 2 | 6.65 | 0.12 | 0.02 |
| plane2 | 17 60 2 | 2 | ✓ | 5 | ✓ | 1.24 | 7.51 |
| plane3 | 29 122 2 | 1 | ✓ | 4 | ✓ | 25.42 | 95.17 |
| ite1 | 7 23 | 2 | 2.6 | 3 | 7.7 | 2.07 | 0.13 |
| ite2 | 9 34 3 | 2 | ✓ | 7 | ✓ | 29.54 | 45.67 |
| sum_2_5 | 11 40 2 | 2 | ✓ | 4 | ✓ | 20.47 | 15.65 |
| search_2 | 5 16 3 | 3 | 3.13 | 6.29 | 2 | 0.13 |
| search_3 | 7 25 4 | 4 | 4.07 | 8.17 | 4.81 | 0.36 |
| max2 | 1 5 2 | 4 | 1.42 | 4 | 1.48 | 0.18 | 0.18 |
| max3 | 3 15 3 | 9 | 16.57 | 9 | ✓ | 9.67 | ✓ |
| sum_2_5 | 1 5 2 | 3 | 1.49 | 3 | 0.69 | 0.26 | 0.69 |
| sum_2_15 | 1 5 2 | 3 | 1.42 | 3 | 0.87 | 0.26 | 0.87 |
| sum_3_5 | 3 15 3 | 8 | 34.84 | 8 | ✓ | 29.85 | ✓ |
| sum_3_15 | 3 15 3 | 9 | 41.87 | 6 | ✓ | 31.03 | ✓ |
| search_2 | 3 15 3 | 5 | 42.55 | 5 | ✓ | 29.92 | ✓ |
| example1 | 3 10 2 | 3 | 1.41 | 3 | 1.12 | 0.16 | 6.54 |
| guard1 | 1 6 2 | 4 | 1.38 | 4 | 0.43 | 0.14 | 0.46 |
| guard2 | 1 6 2 | 4 | 1.54 | 4 | 0.49 | 0.24 | 0.23 |
| guard3 | 1 6 2 | 4 | 1.47 | 4 | 0.46 | 0.25 | 0.85 |
| guard4 | 1 6 2 | 4 | 1.37 | 4 | 0.58 | 0.13 | 0.21 |
| ite1 | 3 15 3 | 8 | 3.59 | 8 | ✓ | 5.35 | ✓ |

B PROOFS

This section presents proofs for Theorem 5.1 and Theorem 5.7, as well as stating and proving the correctness of the vectorized semantics (Theorem B.2) from §6.1, properties of the abstract and underapproximating semantics (Lemma B.3 and Lemma B.4) from §6.2 and §6.3, and soundness and completeness of the fused-semantics optimization (Theorem B.5).

In the proofs, we denote rules using their implication forms: for example, the inference rule in Equation (2) can be written as

\[ \text{sem}^B \left( \langle \Gamma, b \rangle, \text{true} \right) \land \text{sem}^S \left( \langle \Gamma, s \rangle, \Gamma_1 \right) \land \text{sem}^\text{Start} \left( \langle \Gamma_1, \text{while } b \text{ do } s \rangle, \Gamma_2 \right) = \Rightarrow \text{sem}^\text{Start} \left( \langle \Gamma, \text{while } b \text{ do } s \rangle, \Gamma_2 \right). \]

**Theorem (Soundness and Completeness, 5.1).** Consider a SemGuS-with-examples problem sem = \( (G, \forall x \in E. \psi(x, f(x))) \), equipped with semantic rules \( R_{\text{sem}} \), a specification set E, and the Query rule (Equation (8)). Let the CHC form of G be \( R_{\text{syn}} \). Then, Realizable is a theorem over \( R_{\text{sem}} \) and \( R_{\text{syn}} \) if and only if the SemGuS-with-examples problem sem is realizable. Moreover, if Realizable is a theorem, then the value of t in the Query rule satisfies \( t \in L(G) \) and \( \forall x \in E. \psi(x, \lfloor t \rfloor(x)) \).

**Proof.** We prove the theorem by a lemma proving the correctness of the syntax rules \( R_{\text{syn}} \).
LEMMA B.1 (SOUNDNESS AND COMPLETENESS OF $R_{syn}$). Suppose $R_{syn}$ is the CHC representation of a grammar $G$, as described in §2 and §5. Then for any nonterminal $N \in G$, and some term $t$, $sy_N(t)$ is a theorem of $R_{syn}$ if and only if $t \in L(N)$.

**Proof.** By induction on the size of the term $t$ (the number of productions applied to produce $t$), denoted by $n$.

- Base case ($n = 1$): To see completeness, assume $t \in L(N)$. Then $t$ must be created by a production $P$ of the form $N \rightarrow t$, because $t$ is of size 1, thus $P$ cannot contain any nonterminals in the RHS. The CHC representation for $P$ is true $\implies sy_N(t)$, thus $sy_N(t)$ holds.

- Induction hypothesis ($n \leq k$): Assume the Lemma holds for all $n \leq k$.

- Inductive Step ($n = k + 1$): To see completeness, assume $t \in L(N)$; because $n = k + 1$, the first production $P$ in the leftmost derivation of $t$ must contain nonterminals in the RHS (otherwise, $t$ becomes size 1). Without loss of generality, let $P = N \rightarrow op(N_1, \ldots, N_i)$. The CHC representation of this rule is $sy_N(t_1) \land \cdots \land sy_N(t_i) \implies sy_N(t)$, where $t = op(t_1, \ldots, t_i)$. The induction hypothesis holds on terms $t_1, \ldots, t_i$, thus $sy_N(t_1) \land \cdots \land sy_N(t_i)$ holds and $sy_N(t)$ holds as well.

To prove soundness, assume $sy_N(t)$ is a theorem of $R_{syn}$. This implies that there exists a rule $sy_N(t_1) \land \cdots \land sy_N(t_i) \implies sy_N(t)$ in $R_{syn}$; the existence of this rule implies the existence of the production $N \rightarrow op(N_1, \ldots, N_i) \in G$. Because the induction hypothesis holds for $sy_N(t_1), \ldots, sy_N(t_i)$, it follows that $t_1 \in N_1, \ldots, t_i \in N_i$, and thus $t \in N$.

\[ \square \]

The semantic rules are supplied by the user, and we assume they correctly encode the semantics of the SemGuS problem.

As the final step, recall the Query rule from Equation (8). By Lemma B.1, $sy_{Start}(t)$ holds if and only if $t$ is a valid term in $L(G)$; the semantic relations $\land_{e_i \in E} sy_{Start}(t, e_i, o_i)$ hold if and only if $t$ executed on the example set $e_i$ results in the outputs $o_i$, which satisfy the specification. Thus the algorithm is both sound and (relatively) complete.

\[ \square \]

**THEOREM (CORRECTNESS OF LISTING SEMANTICS, 5.7).** Let $R_{list}^\text{Sem}$ be a set of semantic rules using a flattened representation of terms, created from the set of semantic rules $R_{sem}$. Then for any nonterminal $N$, $sem_N(\langle \Gamma, L_{in}, \langle v, L_{out} \rangle \rangle)$ is a theorem of $R_{list}^\text{Sem}$ if and only if $sem_N(\langle \Gamma, t, v \rangle)$ is a theorem of $R_{sem}$, and $L_{in} = L_{I} L_{out}$ (i.e., the concatenation of $L_{I}$ and $L_{out}$) where $L_{I}$ is the pre-order listing of a term $t \in L(N)$.

**Proof.** The correctness of the syntax rules using flattened representations of terms can be proved in an identical manner to Lemma B.1. For the proof of correctness of the semantic rules, we proceed by induction on the height of the derivation tree for $sem_N(\langle \Gamma, L_{in}, \langle v, L_{out} \rangle \rangle)$ (for completeness) and $sem_N(\langle \Gamma, t, v \rangle)$ (for soundness), denoted by $n$.

- Base case ($n = 1$): To see completeness when $n = 1$, notice $sem_N(\langle \Gamma, L_{in}, \langle v, L_{out} \rangle \rangle)$ must be proved by a single rule without premises, namely true $\implies sem_N(\langle \Gamma, L_{in}, \langle v, L_{out} \rangle \rangle)$; by Definition 3.5, the only productions that may create such a rule are of the form $N \rightarrow t \ 1$, where $t$ is a leaf node. In this case, $L_{in} = 1 :: L_{out}$ and the preorder listing of $t$ is $1$. Also note that $N \rightarrow t \ 1$ must be equipped with the semantics true $\implies sem_N(\langle \Gamma, t, v \rangle)$; thus $sem_N(\langle \Gamma, t, v \rangle)$ is a theorem of $R_{sem}$, and $t \in L(N)$.

Proc. ACM Program. Lang., Vol. 1, No. CONF, Article 1. Publication date: January 2018.
A similar proof works for soundness: that $\text{sem}_N((\Gamma, t), v)$ is a theorem of $\mathcal{R}_{\text{sem}}$ for $t \in L(N)$ implies the existence of a production $N \rightarrow t$ equipped with the semantics true \implies $\text{sem}_N((\Gamma, t) :: L_{\text{out}}), (v, L_{\text{out}})$; thus $\text{sem}_N((\Gamma, t) :: L_{\text{out}}), (v, L_{\text{out}})$ is a theorem of $\mathcal{R}_{\text{sem}}$.

- Induction hypothesis ($n \leq k$): We assume the theorem holds for all $n \leq k$.
- Inductive step ($n = k + 1$): Both completeness and soundness can be proved using a similar procedure to the base case.

To see completeness, assume $\text{sem}_N((\Gamma, L_{\text{in}}), (v, L_{\text{out}}))$ is a theorem of $\mathcal{R}_{\text{sem}}$ with a derivation tree of height $k + 1$. Since the height is greater than 1, the derivation tree must contain premises of the form $\phi \land \text{sem}_N((\Gamma, L_1), (v_1, L_2)) \land \cdots \land \text{sem}_N((\Gamma, L_i), (v_i, L_{\text{out}})) \implies \text{sem}_N((\Gamma, t), (v, L_{\text{out}}))$, as stated in Equation (11). Now notice the existence of such a rule in $\mathcal{R}_{\text{sem}}$ implies the existence of a production $A_0 \rightarrow \sigma(A_1, \ldots, A_i)$ equipped with the semantics $\phi \land \text{sem}_{A_i}((\Gamma_1, t_1), v_1), \ldots, \text{sem}_{A_i}((\Gamma_1, t_i), v_i) \implies \text{sem}_{A_i}((\Gamma, t), v_0)$; this in turn implies the rule $\phi \land \text{sem}_{A_i}((\Gamma_1, t_1), v_1), \ldots, \text{sem}_{A_i}((\Gamma_1, t_i), v_i) \implies \text{sem}_{A_0}((\Gamma, t), v_0)$ is in $\mathcal{R}_{\text{sem}}$.

Now note that, due to the induction hypothesis, the theorem holds for the premises $\text{sem}_N((\Gamma_1, L_1), (v_1, L_2)), \ldots, \text{sem}_N((\Gamma, L_i), (v_i, L_{\text{out}}))$, and thus $\text{sem}_{A_i}((\Gamma, t), v_0)$ is a theorem of $\mathcal{R}_{\text{sem}}$, $t \in L(N)$ and $L_{\text{in}} = L_t + + L_{\text{out}}$ (where $L_t$ is the pre-order listing of $t$).

Soundness can be proved in an identical manner by reversing the flow of rules, and applying the induction hypothesis at the last step.

We note that this proof assumes that the height of the derivation tree is finite, which assumes that the given program terminates. This makes little difference, as our algorithm for program synthesis also requires that the produced derivation tree is finite. As shown in §8, this does not affect the practicality of the approach, as $\text{M32SY}$ succeeded in synthesizing programs with possibly infinite loops.

Theorem B.2 states the correctness of the vectorized semantics from §6.1.

**Theorem B.2 (Correctness of Vectorized Semantics).** Given a set of examples $E = [\Gamma_1, \ldots, \Gamma_n]$ and a term $t$, $[t]_E[([\Gamma_1, \ldots, \Gamma_n], [\Gamma_1', \ldots, \Gamma_n'])$ if and only if for every $1 \leq i \leq n$, $[t]_E(\Gamma_i, \Gamma_i')$.

**Proof.** We proceed by structural induction on $G_{\text{impv}}$, from Figure 2. Note $\bot$ is a special state that ignores all computations, i.e., $[t](\bot, \bot)$ for any term $t$.

- Base case: Expressions ($\text{BVExpr}, \text{IntExpr}, \text{BoolExpr}$): Take the expression $x$ as an example. It is clear that $[t]_E([\Gamma_1, \ldots, \Gamma_n], [v_1, \ldots, v_n])$ if and only if for every $1 \leq i \leq n$, $[t]_E(\Gamma_i, v_i)$, because each $v_i$ will be obtained using $\Gamma_i(x)$, remaining independent for all examples. Other base cases in the expressions category can be proved in a similar manner; then one may apply structural induction on the expressions to prove the theorem for expressions.
- Assignments and Updates ($x := E, x := C, \text{arr}[E] := E$): Assignments and updates form the base case for other statements in $G_{\text{impv}}$. Take $x := E$ as an example. Then the rule Assign in Figure 3b directly shows that $[t]_E([\Gamma_1, \ldots, \Gamma_n], [\Gamma_1', \ldots, \Gamma_n'])$ if and only if for every $1 \leq i \leq n$, $[t]_E(\Gamma_i, \Gamma_i')$, as each $\Gamma_i$ is updated separately (as shown on the second premise). A similar argument works for the other cases.
- Sequential Composition ($s_1; s_2$): The vectorized semantic rule for $s_1; s_2$ is as follows:

$$\frac{[s_1]_E(\vec{\Gamma}, \vec{T}_1)}{[s_1; s_2]_E(\vec{\Gamma}, \vec{T}_2)} \text{Seq}_{\text{E}}$$

The case holds due to the induction hypothesis.
• Branch Statements (if \(b\) then \(s_1\) else \(s_2\)): The vectorized semantic rule for \(b\) then \(s_1\) else \(s_2\) is as following:

\[
\begin{bmatrix}
[b]_E(\vec{T}, \vec{v}_b) \\
[s_1]_E(\text{PROJ}(\vec{T}, \vec{v}_b), \vec{T}_1)
\end{bmatrix}
\begin{bmatrix}
[s_2]_E(\text{PROJ}(\vec{T}, \neg \vec{v}_b), \vec{T}_2)
\end{bmatrix}
\stackrel{\text{SITE}_E}{\rightarrow}
\begin{bmatrix}
\text{if } b \text{ then } s_1 \text{ else } s_2
\end{bmatrix}(\vec{T}, \text{MERGE}(\vec{T}_1, \vec{T}_2))
\]

Where PROJ and MERGE are defined in Figure 3b. By the induction hypothesis, the entries of \(\vec{T}_1\) are correct on those states which have not been projected to \(\bot\), i.e. \(b\) evaluates to true; in other cases it is \(\bot\). The same holds for \(\vec{T}_2\) for states where \(b\) evaluates to false; in other cases it is \(\bot\). Then by the definition of MERGE, the theorem holds for if-then-else as well.

• Loops (while \(b\) do \(s\)): The rules WTrue_E and WFalse_E are the two semantic rules associated with the term while \(b\) do \(s\). To prove the case for while loops, we introduce an additional induction on the number of loop iterations, denoted by \(n\).

– Base case \((n = 0)\): In this case, there are 0 iterations and thus the tree consists of only one application of WFalse_E; the theorem holds as all conditions are false or \(\bot\) (which implies the incoming state is \(\bot\) as well), and the input states equal the output states for all examples, for both the vectorized and standard semantics.

– Induction hypothesis \((n \leq k)\): Assume the theorem holds for all \(n \leq k\).

– Inductive step \((n = k + 1)\): If \(n \geq 1\), note that there must be at least one application of the WTrue_E rule in the derivation tree. Consider the first of these applications, and note that the induction hypothesis holds for all three semantic premises: \([b]_E(\vec{T}, \vec{v}_b)\), \([s]_E(\text{PROJ}(\vec{T}, \vec{v}_b), \vec{T}_1)\) (structural induction), and \([\text{while } b \text{ do } s]_E(\vec{T}_1, \vec{T}_2)\) (height of the derivation tree). Again, by the semantics of the MERGE operator, the theorem holds for the inductive step as well: the PROJ and MERGE operators send individual examples on which the loop should not iterate (i.e., \(v_{b_i}\) evaluates to false for the \(i\)-th entry \(\vec{T}_i\)) to \(\bot\), and recovers them to their original values after iteration has finished for all examples in the vector.

\[\square\]

Lemma B.3 states that the abstract domain \(\mathbb{B}_i\) from §6.2 satisfies our definition of an abstract semantics.

**Lemma B.3.** The semantics \([\cdot]^*\) defined in §6.2 over the domain \(\mathbb{B}_i\) is an abstract semantics for \(G_{\text{imp}v}\), with respect to the standard semantics \([\cdot]\) partly given in Figure 3a, defined over the integers \(\mathbb{Z}\).

**Proof.** The abstraction function \(\alpha\) is given as \(\alpha(v) = \text{bit}_i(v)\), where \(\text{bit}_i(v)\) denotes the \(i\)-th bit of \(v\). The concretization function \(\gamma\) can be given as \(\gamma(\text{true}) = \{v \in \mathbb{Z} \mid \text{bit}_i(v) = \text{true}\}\), \(\gamma(\text{false}) = \{v \in \mathbb{Z} \mid \text{bit}_i(v) = \text{false}\}\), and \(\gamma(\top) = \mathbb{Z}\); \(\alpha\) and \(\gamma\) form a Galois connection, thus the semantics are abstract. The different abstract transformers presented in Equations (13) (WTrue\textsubscript{Havoc}) and (14) (WTrue\textsubscript{Join}) are both sound: clearly WTrue\textsubscript{Havoc} is sound as \(\Gamma^\top = \top\) and thus all values in the state are concretized to \(\mathbb{Z}\) through \(\gamma\). Similarly, WTrue\textsubscript{Join} is sound as well as \(\Gamma^\top = \top\), or, given \([\text{while } b \text{ do } s]_E(\vec{T}, \vec{v}_b)\) for the standard semantics, the elements of \(\vec{T}_\gamma\) are guaranteed to be within the concretization of \(\vec{T}^\gamma\) by the definition of JOIN.

\[\square\]

Lemma B.4 states that the underapproximating semantics from §6.3 and Example 6.3 satisfies our definition of an underapproximating semantics.

**Lemma B.4.** The semantics \([\cdot]^\prime\) described in §6.3 and Example 6.3 is an underapproximating semantics with respect to the standard semantics \([\cdot]\) partly given in Figure 3a.
Prove. \([.]^b\) is defined by adding the constraint \(i > 0\) to the standard semantics. Because the semantic rules have identical structure, one can apply a simple structural induction to show that a theorem \([t]^b(\Gamma, v)\) can only be proved if \([t](\Gamma, v)\).

Finally, Theorem 5.7 states the soundness and completeness of the fused-semantics optimizations from §7.2.

**Theorem B.5 (Soundness and Completeness of Fused Semantics).** Let \(\mathcal{R}_{\text{syn}}^{\text{fused}}\) denote a set of fused syntax rules created according to the fused-semantics optimization described in §7.2, from a grammar \(G\) equipped with a semantics \(\mathcal{R}_{\text{sem}}\). Then for any nonterminal \(N \in G\), \(\mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t, v)\) is a theorem over \(\mathcal{R}_{\text{syn}}^{\text{fused}}\) and \(\mathcal{R}_{\text{sem}}\) if and only if \(t \in L(N)\) and \(\mathcal{R}_{\text{sem}}(\Gamma, t, v)\) is a theorem over \(\mathcal{R}_{\text{sem}}\).

**Proof.** Similar to the way we proved correctness of the listing semantics, we proceed by induction on the height of the derivation tree for \(\mathcal{R}_{\text{sem}}(\Gamma, t, v)\) (for completeness) and \(\mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t, v)\) (for soundness), denoted by \(n\): the proof is essentially a merging of the proofs for Lemma B.1 and Theorem 5.7.

- **Base case** \((n = 1)\): To see completeness, assume that \(t \in L(N)\) and \(\mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t, v)\) is a theorem over \(\mathcal{R}_{\text{sem}}\). Because the \(n = 1\), \(t\) is a leaf of size 1; and thus there must exist a production \(N \rightarrow t\) equipped with the semantics true \(\Rightarrow \mathcal{R}_{\text{sem}}(\Gamma, t, v)\). Such a production and its semantics is encoded using the fused-semantics optimization as true \(\Rightarrow \mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t, v)\) in \(\mathcal{R}_{\text{syn}}^{\text{fused}}\), thus the theorem holds.

To see soundness, assume that \(\mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t, v)\) is a theorem of \(\mathcal{R}_{\text{syn}}^{\text{fused}}\). Because \(n = 1\), the derivation tree for \(\mathcal{R}_{\text{syn}}^{\text{fused}}\) cannot contain any premises and thus there must be a rule of the form true \(\Rightarrow \mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t, v)\) in \(\mathcal{R}_{\text{syn}}^{\text{fused}}\); this implies the existence of a production \(N \rightarrow t\), equipped with the semantics true \(\Rightarrow \mathcal{R}_{\text{sem}}(\Gamma, t, v)\), in \(G\). Thus \(t \in L(N)\) and \(\mathcal{R}_{\text{sem}}(\Gamma, t, v)\) is a theorem over \(\mathcal{R}_{\text{sem}}\); the theorem holds.

- **Inductive step** \((n = k + 1)\): To see completeness, again assume that \(t \in L(N)\) and \(\mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t, v)\) is a theorem over \(\mathcal{R}_{\text{sem}}\). As \(n > 1\), there must be premises in the first rule applied for proving \(\mathcal{R}_{\text{sem}}\); i.e., the rule must be of form \(\phi \wedge \mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t_1, v_1) \wedge \cdots \mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t_i, v_i)\). Because the \(i\)th rule in \(\mathcal{R}_{\text{sem}}\) implies the existence of a production \(N \rightarrow t_i\) equipped with the rule as a semantics (for example, \(N \rightarrow \) while \(S\) do \(S\) from Figure 5 below). This rule in \(G\) is encoded into the rule \(\phi \wedge \mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t_1, v_1) \wedge \cdots \mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t_i, v_i)\) \(\Rightarrow \mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t, v)\) in \(\mathcal{R}_{\text{syn}}^{\text{fused}}\) (following the existence of \(\mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t, v)\). This in turn implies the existence of a production \(N \rightarrow \) op\((N_1, \ldots, N_i)\) in \(G\), equipped with the semantics \(\phi \wedge \mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t_1, v_1) \wedge \cdots \mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t_i, v_i)\). Because the induction hypothesis, \(t \in L(N)\) and \(\mathcal{R}_{\text{sem}}(\Gamma, t, v)\) is a theorem of \(\mathcal{R}_{\text{sem}}\), the proof works for loops as well, given that the loop terminates.声学科学在一系列程序中：假设 \(\mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t, v)\) 是 \(\mathcal{R}_{\text{syn}}^{\text{fused}}\) 的一个定理，因为 \(n > 1\), 第一个规则在推导 \(\mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t, v)\) 必须是形式 

\(\phi \wedge \mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t_1, v_1) \wedge \cdots \mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t_i, v_i)\). 这时在 \(G\), 装备了它们的定义， \(\phi \wedge \mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t_1, v_1) \wedge \cdots \mathcal{R}_{\text{syn}}^{\text{fused}}(\Gamma, t_i, v_i)\). 由归纳假设， \(t \in L(N)\) 和 \(\mathcal{R}_{\text{sem}}(\Gamma, t, v)\) 是一个定理。
Figure 5 displays fused syntax rules for the productions $S \rightarrow x := E$, $S \rightarrow S; S$, $S \rightarrow$ if $B$ then $S$ else $S$, $S \rightarrow$ while $B$ do $S$ in $G_{impv}$ as an example, generated by applying the fused-semantics optimization on standard semantics.

$$
\text{syn}_{S \rightarrow x := E}^{\text{fused}}(⟨\Gamma, e⟩, v) \quad \text{syn}_{S}^{\text{fused}}((⟨\Gamma, x := e⟩, \Gamma_1))
$$

$$
\text{syn}_{S \rightarrow S; S}^{\text{fused}}(⟨\Gamma, s_1⟩, \Gamma_1) \quad \text{syn}_{S}^{\text{fused}}((⟨\Gamma, s_2⟩, \Gamma_2))
$$

$$
\text{syn}_{S \rightarrow \text{if } B \text{ then } S \text{ else } S, \text{true}}^{\text{fused}}(⟨\Gamma, b⟩, \text{true}) \quad \text{syn}_{S}^{\text{fused}}((⟨\Gamma, s_1⟩, \Gamma_1))
$$

$$
\text{syn}_{S \rightarrow \text{if } B \text{ then } S \text{ else } S, \text{false}}^{\text{fused}}(⟨\Gamma, b⟩, \text{false}) \quad \text{syn}_{S}^{\text{fused}}((⟨\Gamma, s_2⟩, \Gamma_1))
$$

$$
\text{syn}_{S \rightarrow \text{while } B \text{ do } S, \text{true}}^{\text{fused}}(⟨\Gamma, b⟩, \text{true}) \quad \text{sem}_{S}(⟨\Gamma_1, \text{while } b \text{ do } s⟩, \Gamma_2)
$$

$$
\text{syn}_{S \rightarrow \text{while } B \text{ do } S, \text{false}}^{\text{fused}}(⟨\Gamma, b⟩, \text{false})
$$

$$
\text{syn}_{S \rightarrow \text{while } B \text{ do } s, \Gamma}^{\text{fused}}(⟨\Gamma, \text{while } b \text{ do } s⟩, \Gamma)
$$

Fig. 5. Example syntax rules using the fused semantics.