Verifying Solutions to Semantics-Guided Synthesis Problems

CHARLIE MURPHY, University of Wisconsin-Madison, USA*
KEITH J.C. JOHNSON, University of Wisconsin-Madison, USA
THOMAS REPS, University of Wisconsin-Madison, USA
LORIS D'ANTONI, University of California at San Diego, USA

Semantics-Guided Synthesis (SemGuS) provides a framework to specify synthesis problems in a solver-agnostic and domain-agnostic way, by allowing a user to provide both the syntax and semantics of the language in which the desired program should be synthesized. Because synthesis and verification are closely intertwined, the SemGuS framework raises the following question: *how does one verify that a user-given program satisfies a given specification when interpreted according to a user-given semantics?*

In this paper, we prove that this form of *language-agnostic* verification (specifically that verifying whether a program is a valid solution to a SemGuS problem) can be reduced to proving validity of a query in the μCLP calculus, a fixed-point logic that is capable of expressing alternating least and greatest fixed-points. Our encoding into μCLP allows us to further classify the SemGuS verification problems into ones that are reducible to satisfiability of (*i*) first-order-logic formulas, (*ii*) Constrained Horn Clauses, and (*iii*) μCLP queries. Furthermore, our encoding shines light on some limitations of the SemGuS framework, such as its inability to model nondeterminism and reactive synthesis. We thus propose a modification to SemGuS that makes it more expressive, and for which verifying solutions is exactly equivalent to proving validity of a query in the μCLP calculus. Our implementation of SemGuS verifiers based on the above encoding can verify instances that were not even encodable in previous work. Furthermore, we use our SemGuS verifiers within an enumeration-based SemGuS solver to correctly synthesize solutions to SemGuS problems that no previous SemGuS synthesizer could solve.

CCS Concepts: • **Theory of computation → Operational semantics**; **Automated reasoning**; **Logic and verification**; *Constraint and logic programming*; • **Software and its engineering → Semantics**.

Additional Key Words and Phrases: SemGuS, Semantics, SMT, CHC, Program Verification, Program Synthesis

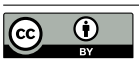
ACM Reference Format:
Charlie Murphy, Keith J.C. Johnson, Thomas Reps, and Loris D'Antoni. 2025. Verifying Solutions to Semantics-Guided Synthesis Problems. *Proc. ACM Program. Lang.* 9, PLDI, Article 217 (June 2025), 25 pages. https://doi.org/10.1145/3729320

1 Introduction

In program synthesis, the goal is to find a program in a given search space that meets a given specification. Synthesis has found great successes in specific domains, e.g., spreadsheet transformations [35] and bit-vector manipulations [19], where the search space is fixed and its properties can be exploited to design powerful domain-specific synthesis solvers. However, for synthesis to become a general-purpose technology that can help users with a variety of tasks, one should be

*This work submitted prior to joining Amazon Web Services.

Authors' Contact Information: Charlie Murphy, University of Wisconsin-Madison, USA , tcmurphy4@wisc.edu; Keith J.C. Johnson, University of Wisconsin-Madison, USA, keithj@cs.wisc.edu; Thomas Reps, University of Wisconsin-Madison, USA, reps@cs.wisc.edu; Loris D'Antoni, University of California at San Diego, USA, ldantoni@ucsd.edu.



ACM 2475-1421/2025/6-ART217
https://doi.org/10.1145/3729320

able to customize the search space and specifications of a synthesis problem in a programmable way that is agnostic of a specific domain or synthesis solver.

To address the problem of making synthesis "programmable", Kim et al. [26] proposed the SemGuS framework, which enables one to specify synthesis problems in a solver-agnostic and domain-agnostic way. The key differentiating aspect of the SemGuS framework is that a user provides a set of Constrained Horn Clauses (CHCs) whose *least solution* defines the semantics of the programming language over which one is interested in performing synthesis. (A detailed example of a SemGuS problem is given in Figure 1 and discussed in §2.1.) While this formalism enables a great deal of flexibility when describing a synthesis problem—e.g., one can naturally define the operational semantics of an imperative programming language—this generality comes at a cost: building solvers for general SemGuS problems can be difficult [15].

Due to this complexity, most of the work on SemGuS has limited its focus to problems whose specifications are given as a finite set of examples [26]. To go beyond specifications with examples and consider specifications with quantified variables, we require a technique capable of verifying whether a candidate solution is correct. Verification is not only needed to check that the final solution meets the specification, it is also often used to implement synthesis algorithms that use enumeration and constraint-solving to efficiently explore the search space of possible programs—i.e., *verifying whether a candidate solution is correct is a crucial missing component that is needed for solving SemGuS problems involving complex specifications, and for building effective SemGuS solvers.*

One of the key challenges raised by SemGuS (and the focus of this paper) is *how does one build a language-agnostic verifier that can check if a program matches a specification?*—i.e., when the verification user provides the program, specification, *and* the semantics of the language in which the program should be interpreted. Specifically, in SemGuS the user provides the semantics of the programming language as a set of CHCs [8]—i.e., a set of Horn clauses augmented with first-order theories that give meaning to a set of relations defining the semantics of programs.

From the standpoint of the SemGuS framework, the language for which we are performing verification is *not* fixed—and can in fact be arbitrary—and the verification technique needs to be able to reason about the CHCs that define a language's semantics. In particular, because the semantics of the input language is provided logically, there is no easy way to relate it to known verification approaches that are tailored to specific programming constructs. This last aspect makes existing verification approaches that are tied to specific programming languages [20, 23, 29, 34, 42] not suitable for verifying solutions to SemGuS problems. In particular, these verification approaches take advantage of a fixed programming language and its fixed semantics to use specialized techniques such as loop-invariants and Hoare-style reasoning for imperative programs [29] and type-based reasoning for functional programs [34]. The recently released SemGuS Toolkit [25] contains a baseline verifier as part of their KS2 SemGuS synthesizer. While the baseline verifier is in theory capable of handling specifications beyond examples, it does so via a naive encoding of the specification and CHC semantic relations as an SMT formula with recursively defined functions, a theory that SMT solvers can rarely reason about successfully.

In this paper, we present a comprehensive study of the problem of verifying solutions to SemGuS problems. The first contribution of this paper is the following: given a program p, a semantics defined using Constrained Horn Clauses *Sem*, and a specification φ (which is allowed to mention the semantic relations defined by *Sem*), we show that the problem of verifying whether p—when evaluated according to *Sem*—satisfies φ can be expressed as a validity check in the μCLP calculus [39]. μCLP is a fixed-point logic that generalizes CHCs and co-CHCs by combining both least and greatest fixed-points with interpreted first-order theories. While SemGuS uses only least fixed-points to define the semantics of programs (i.e., as the least solution to a set of CHCs), the fact that the semantic relations can appear in both positive and negative positions in a user-supplied

specification means that reasoning needs to be carried out in a proper fixed-point logic, such as μCLP. Because of the complexity of building solvers for checking validity of μCLP queries, the second contribution of the paper is to identify fragments of SemGuS verification problems that can be reduced to checking satisfiability of first-order-logic formulas (Satisfiability Modulo Theories) and CHCs, for which more scalable solvers exist [10, 21]. Furthermore, because we show an equivalence between SemGuS verification and μCLP satisfiability (Theorem 4.8), these reductions are directly applicable to checking satisfiability of μCLP formulas themselves.

Our study highlights a strong connection between SemGuS and μCLP, and raises the question of whether there exist synthesis problems for which verification is expressible using μCLP, but cannot be expressed in SemGuS. We answer this question affirmatively by showing that SemGuS cannot reason about programs involving nondeterminism and games, both of which are commonly found in reactive-synthesis problems [3]. To close the loop between μCLP and SemGuS, we define a minimal extension of SemGuS—i.e., we allow relations to appear in a negated form in the semantic definitions and require an ordering over semantic relations—which results in a new framework that aligns exactly with what is verifiable using μCLP. Finally, we incorporate our verification technique into a synthesizer for SemGuS problems that is capable of solving SemGuS problems with complex logical specifications.

Contributions. Our work makes the following contributions.

- We identify how the problem of verifying programs in SemGuS is tightly related to proving validity in fixed-point logics (§2).
- We propose an extension of the SemGuS framework that can capture, e.g., reactive synthesis problems (§3).
- We analyze when solutions to SemGuS problems can be verified using various logical fragments (SMT, CHC, and μCLP) and show that our extension of SemGuS aligns exactly with what is verifiable using μCLP (§4) .
- We implemented our approach in a tool, Muse, together with several optimizations (§5), and used Muse to verify (or disprove correctness for) solutions to SemGuS problems (§6). The problems comprise a variety of functional and reactive-synthesis problems, encoding such domains as imperative programs, regular expressions, Büchi games, and robot path planning.
- We incorporated Muse within the SemGuS synthesizer Ks2 to enable Ks2 to solve SemGuS problems that could not be solved by any previous approach (§6). In particular, by incorporating Muse into Ks2, we enabled Ks2 to solve 135 SemGuS problems involving specifications with quantified variables (74 of which could not be solved by Ks2 using the baseline verifier).

§7 discusses related work. §8 concludes.

2 Overview

This section illustrates our approaches for verifying a solution to a SemGuS problem using three problems of increasing complexity. Our technique reduces the verification task to checking validity of queries in various logical fragments that can be dispatched to existing solvers. The examples should provide enough details to understand the SemGuS framework.

2.1 Max2: Quantified SMT

Consider the problem of synthesizing a loop-free imperative program with two variables `x` and `y` that computes the maximum of two values. Figure 1 gives all the components necessary to define this synthesis problem in the SemGuS framework:

- A grammar G_{max2} defining the syntax of the language under consideration (Figure 1a).

$$S := \texttt{x = } E \mid \texttt{y = } E \mid S;S \mid \texttt{Ite } B\ S\ S$$
$$E := \texttt{0} \mid \texttt{1} \mid \texttt{x} \mid \texttt{y} \mid E\texttt{+}E$$
$$B := E \texttt{ < } E$$

(a) Grammar G_{max2}

$$\forall x, y, x'.\ \left(\begin{array}{c} \exists y'.Sem_S(max2, x, y, x', y') \\ \Updownarrow \\ (x' = x \vee x' = y) \wedge x \leq x' \wedge y \leq x' \end{array} \right)$$

(b) Specification φ_{max2}

```
Ite
  (x < y)
  (x = y)
  (x = x)
```

(c) Solution s_{max2}

$$\frac{Sem_E(\texttt{e}, x, y, x') \wedge y = y'}{Sem_S(\texttt{x = e}, x, y, x', y')} \qquad \frac{Sem_E(\texttt{e}, x, y, y') \wedge x = x'}{Sem_S(\texttt{y = e}, x, y, x', y')} \qquad \frac{Sem_B(\texttt{b}, x, y, \top) \quad Sem_S(\texttt{t}, x, y, x', y')}{Sem_S(\texttt{Ite b t e}, x, y, x', y')}$$

$$\frac{Sem_S(\texttt{s}, x, y, x'', y'') \quad Sem_S(\texttt{t}, x'', y'', x', y')}{Sem_S(\texttt{s; t}, x, y, x', y')} \qquad \frac{Sem_B(\texttt{b}, x, y, \bot) \quad Sem_S(\texttt{e}, x, y, x', y')}{Sem_S(\texttt{Ite b t e}, x, y, x', y')}$$

$$\frac{}{Sem_E(\texttt{0}, x, y, 0)} \qquad \frac{}{Sem_E(\texttt{1}, x, y, 1)} \qquad \frac{Sem_E(\texttt{s}, x, y, r_s) \quad Sem_E(\texttt{t}, x, y, r_t) \quad b \Leftrightarrow r_s < r_t}{Sem_B(\texttt{s < t}, x, y, b)}$$

$$\frac{Sem_E(\texttt{s}, x, y, r_s) \quad Sem_E(\texttt{t}, x, y, r_t) \quad r = r_s + r_t}{Sem_B(\texttt{s + t}, x, y, r)} \qquad \frac{}{Sem_E(\texttt{x}, x, y, x)} \qquad \frac{}{Sem_E(\texttt{y}, x, y, y)}$$

(d) Semantics Sem_{max2}

Fig. 1. SemGuS definition for the problem of synthesizing an *imperative program max*2 that computes the maximum of two input values x and y. Figure 1a contains a regular tree grammar defining the syntax of the language we can use to build programs (i.e., imperative programs with if-then-else, comparisons, and linear assignments). The semantics of the language is inductively defined using constrained Horn clauses (Figure 1d)—e.g., the semantics of programs derivable from nonterminal S is given via the inductively defined relation $Sem_S(\texttt{s}, x, y, x', y')$ where, for example, $Sem_S(\texttt{x=1, 3, 3, 1, 3})$ denotes that running the program x=1 with initial values of 3 for both x and y results in a state where x is 1 and y is 3. Figure 1b specifies when the solution is correct: on an input state x, y, the program $max2$ should output a state x', y' such that x' is greater or equal than the values of x and y and is equal to one of them. The program in Figure 1c (parenthesis are added for readability) is a possible solution to this SemGuS problem—this program is in the grammar G_{max2} and when evaluated on any possible input state according to the semantics Sem_{max2}, it satisfies the specification φ_{max2}.

- A set of constrained Horn clauses Sem_{max2} that inductively define (as the least solution of Sem_{max2}) the semantics of all programs in the language (Figure 1d).
- A specification φ_{max2} that describes how the synthesized program should behave when evaluated according to Sem_{max2} (Figure 1b).

The specification φ_{max2} states that the synthesized program (represented symbolically by the variable *max2*) must terminate in a state in which x' (i.e., the final value of the variable x) is the maximum of the initial-state values assigned to variables x and y—i.e., x and y. Solving this SemGuS problem means providing a program in the grammar that satisfies this specification when evaluated according to the semantics.

Figure 1c presents a *candidate* solution s_{max2} to this SemGuS problem. Rather than determining how to synthesize s_{max2}, this paper tackles the following question: how do we show that when the program s_{max2} is "evaluated" according to the semantics Sem_{max2}, it satisfies the specification φ_{max2}.

Beyond CHCs. Because the semantics is already defined as the least solution to a set of CHCs, it is natural to expect to be able to solve the problem in terms of CHC satisfiability. That is, at first blush, it seems plausible to check the query in Equation (1), which states that φ_{max2} is valid when interpreted using the least solution of the semantic relations (Sem^{LFP}_{max2}).

$$Sem^{LFP}_{max2} \models \forall x, y, x'.(\exists y'.Sem_S(max2, x, y, x', y') \Leftrightarrow (x' = x \vee x' = y) \wedge x \leq x' \wedge y \leq x') \qquad (1)$$

While the semantics is defined as the least solution to a set of constrained Horn clauses Sem_{max2}, the positive occurrence of Sem_S within φ_{max2} results in a query that cannot be reasoned about using the typical decision procedure for CHCs (i.e., CHC satisfiability). Given a set of CHCs C and a query ψ, the typical approach to prove ψ is valid under the least solution of C is to instead check if $C \wedge \neg\psi$ is unsatisfiable. This translation is sound provided that ψ contains only negative occurrences of the uninterpreted relations defined by C. Because the specification φ_{max2} contains a positive occurrence of Sem_S, we must turn to a different approach to solving the validity query.

Finite derivation trees can be desugared. Our first insight is that for problems like $max2$, where the semantic definitions are recursively defined with respect to the term's proper subterms, one can always build a finite derivation tree that describes the semantics of a given program. For example, the derivation tree for s_{max2} is as follows:

$$\dfrac{\dfrac{\dfrac{r_s = x}{Sem_E(\texttt{x}, x, y, r_s)} \quad \dfrac{r_t = x}{Sem_E(\texttt{y}, x, y, r_t)} \quad r_s < r_t}{Sem_B(\texttt{x < y}, x, y, b)} \quad b \wedge \dfrac{\dfrac{y = x'}{Sem_E(\texttt{y}, x, y, x')} \quad y = y'}{Sem_S(\texttt{x = y}, x, y, x', y')} \quad \vee \neg b \wedge \dfrac{\dfrac{x = x'}{Sem_E(\texttt{x}, x, y, x')}}{Sem_S(\texttt{x = x}, x, y, x', y')}}{Sem_{s_{max2}} = Sem_S(\texttt{Ite (x < y) (x = y) (x = x)}, x, y, x', y')} \tag{2}$$

Because the tree is finite, the relation $Sem_{s_{max2}}$ can be defined by a recursion-free formula. In particular, we can "symbolically execute" the tree in Equation (2) starting from the leaves and working toward the root. At each step, a semantic relation in the succedent of an inference-rule instance is replaced by its definition and simplified using the properties available in the antecedent. Via this process, we can extract a formula that exactly characterizes $Sem_{s_{max2}}$, which we then substitute for $Sem_{s_{max2}}$ in Equation (1). We thus obtain the following equivalent formula ($\varphi_{SEM_{s_{max2}}}$), which is stated entirely in first-order logic, without any fixed-point operators:

$$\forall x, y, x'.\ (\exists y'.y = y' \wedge ((x < y \wedge x' = y) \vee (x \geq y \wedge x' = x))) \Leftrightarrow (x' = x \vee x' = y) \wedge x \leq x' \wedge y \leq x'$$

In our tool Muse, the quantified satisfiability modulo theories (SMT) solver Z3 [10] proves this formula valid in 0.06 seconds, proving that s_{max2} is a correct solution to this SemGuS problem.

2.2 DoubleViaLoop: Partial (CHCs) and Total Correctness (μCLP)

While CHCs were insufficient for reasoning about the example presented in §2.1, a large class of interesting SemGuS verification problems can be solved as a CHC satisfiability problem—i.e., SemGuS problems whose specifications encode partial correctness. Consider the SemGuS problem given in Figure 2, which requires synthesizing an imperative program (this time potentially containing a loop) that is partially correct. In particular, if the variables x and y start with the values x and y, respectively, and the program terminates, then it must set the value y' of variable y to $2x$. The grammar G_{loop} is restricted so that assignments can only increment and decrement variables (Figure 2a)—i.e., a correct program for the task must contain a loop. The semantics Sem_{loop} of this language (Figure 2d) is defined similarly to the one in our previous example. The key distinction is how the second CHC, which defines the (big-step) semantics of loops, is not structurally decreasing—i.e., the loop l appears again in a semantic relation in the premise of the CHC. Similar to the previous example, we can prove s_{loop} correct by proving validity of the query $Q_{loop} \triangleq Sem^{LFP}_{loop} \models \varphi_{loop}[f_{loop} \mapsto s_{loop}]$. Because φ_{loop} contains only negative occurrences of the semantic relations, one can solve the query using the typical approach for CHCs. In our tool Muse, we use the CHC solver Spacer [28] to solve the query in 0.2s, proving that s_{loop} is a valid solution.

Total Correctness. Consider the following alternative specification, φ^{tot}_{loop}, which states a form of total correctness—if x is positive and y' is twice x, then starting in the state where x is x and y is 0

$L := \texttt{while}\ B\ \texttt{do}\ \langle S \rangle$

$S := \texttt{x++} \mid \texttt{x--} \mid \texttt{y++} \mid \texttt{y--} \mid S;S$

$B := \texttt{0 < x} \mid \texttt{0 < y}$

(a) Grammar G_{loop}

```
while 0 < x do
  x--;
  y++;
  y++
```

(b) Solution s_{loop}

$$\forall x, y'. \begin{pmatrix} 0 \le x \land Sem_L(f_{loop}, x, 0, 0, y') \\ \Downarrow \\ 2x = y' \end{pmatrix}$$

(c) Specification φ_{loop}

$$\frac{Sem_B(\texttt{b}, x, y, \top) \qquad Sem_S(\texttt{s}, x, y, x'', y'') \qquad Sem_L(\texttt{while b do s}, x'', y'', x', y')}{Sem_L(\texttt{while b do s}, x, y, x', y')}$$

$$\frac{Sem_B(\texttt{b}, x, y, \bot) \qquad x = x' \qquad y = y'}{Sem_L(\texttt{while b do s}, x, y, x', y')} \qquad \frac{Sem_S(\texttt{s}, x, y, x'', y'') \qquad Sem_S(\texttt{t}, x'', y'', x', y')}{Sem_S(\texttt{s; t}, x, y, x', y')}$$

$$\frac{x' = x + 1 \qquad y' = y}{Sem_S(\texttt{x++}, x, y, x', y')} \qquad \frac{x' = x - 1 \qquad y' = y}{Sem_S(\texttt{x--}, x, y, x', y')} \qquad \frac{b \Leftrightarrow 0 < x}{Sem_B(\texttt{0 < x}, x, y, b)}$$

$$\frac{x' = x \qquad y' = y + 1}{Sem_S(\texttt{y++}, x, y, x', y')} \qquad \frac{x' = x \qquad y' = y - 1}{Sem_S(\texttt{y--}, x, y, x', y')} \qquad \frac{b \Leftrightarrow 0 < y}{Sem_B(\texttt{0 < y}, x, y, b)}$$

(d) Semantics Sem_{loop}

Fig. 2. The four components of the specification of "multiply by 2" in the language G_{loop}.

the candidate program *must terminate* in a state where x takes value 0 and y takes value y':

$$\forall x, y'.\ 0 \le x \land 2x = y' \implies Sem_L(f_{loop}, x, 0, 0, y')$$

The above formula ensures termination by having Sem_L appear positively in the specification—i.e., for a program t to satisfy the specification, if $0 \le x \land 2x = y'$, then $Sem_L(t, x, 0, 0, y')$ must be inhabited (i.e., because Sem_L is the least solution to the set of semantic rules in Figure 2d, the proof-tree of $Sem_L(t, x, 0, 0, y')$ must have finite height for all values x and y' can take). The reason why this specification ensures total correctness is because the semantic rules defining Sem_L are deterministic. If the semantics of Sem_L were non-deterministic, this specification would only ensure that there is *at least one* terminating execution, not that *all* executions are terminating.

Similar to the example in §2.1, the specification contains a positive occurrence of a semantic relation and thus cannot be solved via reduction to CHC satisfiability. However, unlike the example in §2.1, the program s_{loop}'s semantics does not have a finite derivation tree (i.e., the semantics of loops are necessarily recursive). To prove the query $Q^{tot}_{loop} \triangleq Sem^{LFP}_{loop} \models \varphi^{tot}_{loop}[f_{loop} \to s_{loop}]$, we turn to a different approach. Rather than CHCs, we consider a more expressive logic, μCLP [39].

The μCLP calculus is a fixed-point logic that generalizes CHCs. Specifically, we can translate the CHCs in Sem_{loop} into *least fixed-point* equations within the μCLP calculus (whose one and only solution is exactly Sem^{LFP}_{loop}). The process to translate from CHCs to fixed-point equations in μCLP is straightforward. A CHC of the form $\forall \overline{x}_0, \ldots, \overline{x}_n.\ H(\overline{x}_0) \leftarrow R_1(\overline{x}_1) \land R_n(\overline{x}_n) \land \varphi$ is translated to the least fixed-point equation $H(\overline{x}_0) =_\mu \exists \overline{x}_1, \ldots, \overline{x}_n.\ R_1(\overline{x}_1) \land R_n(\overline{x}_n) \land \varphi$. For a set of CHCs, the translation is applied to each CHC, then equations with identical left-hand sides are combined disjunctively. Thanks to the first-class support for least fixed-points in the μCLP calculus, we can exactly translate Q^{tot}_{loop}—and all SemGuS verification queries—into an equivalent μCLP query. In our tool Muse, we perform this translation and solve the produced μCLP query using MuVal ([39]) in 1.6s, proving that s_{loop} is totally correct.

S := repeat S | stay | L | R
| L; S | R; S

(a) Robot Strategy

repeat
R; R; R; R; R;
L; L; L; L; L

(b) Solution *strat*

$\neg\overline{Buchi}(strat, 0, 0)$

(c) Specification

$$\overline{Buchi}(strat, x, y) \leftarrow x \neq y \vee (\exists y'.\ 0 \leq y' \leq 5 \wedge \neg Reach(strat, x, y'))$$
$$Reach(strat, x, y) \leftarrow \neg\overline{Buchi}(strat, x, y) \vee (\exists x', strat'.\ Move(strat, x, x', strat') \wedge reach(strat', x', y))$$
$$Move(\texttt{repeat}\ s, x, x', strat') \leftarrow Move(s; \texttt{repeat}\ s, x, x', strat')$$
$$Move(\texttt{L}; s, x, x', strat') \leftarrow strat' = s \wedge x' = x - 1$$
$$Move(\texttt{R}; s, x, x', strat') \leftarrow strat's \wedge x' = x + 1$$

(d) Semantics Sem_{Buchi}.

Fig. 3. An example of a SemGuS$^{\mu}$ problem encoding a Büchi game (a kind of reactive synthesis problem). The Büchi game requires the player (a robot) to follow a given strategy to forever reach a sequence of moving targets. The set of allowable strategies is displayed in (a). The robot can move left or right (possibly forever using repeat). In (b) a solution satisfying the Büchi game is displayed. Following *strat* the robot will repeatedly patrol right and left five paces. The specification in (c), requires the robot to reach the moving target forever, when starting at the origin. In (d) we express the rules of the Büchi game as well as the semantics of the productions used to define the solution.

2.3 Beyond SemGuS

§2.2 showed that for every SemGuS problem, one can verify the correctness of a candidate solution using a μCLP solver. This connection raises a natural question in the opposite direction: *Is there an extension of SemGuS that can express a class of interesting synthesis problems whose semantics and properties of interest cannot currently be encoded in SemGuS, but whose verification conditions are expressible within μCLP?* In this paper, we answer the question affirmatively and propose SemGuS$^{\mu}$, a relatively minor extension of SemGuS such that, in a sense for which we provide a formal proof in Theorem 4.7, SemGuS$^{\mu}$ *captures exactly* every synthesis problem for which solutions can be verified using μCLP.

We illustrate this extension with the SemGuS$^{\mu}$ synthesis problem shown in Figure 3, which requires synthesizing a strategy for a robot to reach a series of targets infinitely often. These types of synthesis problems are often referred to as reactive synthesis problems. For simplicity, we consider a world in which the robot and target's positions are represented by integers with the targets appearing within a bounded region (e.g., between 0 and 5). Once the robot reaches a target, an adversary picks the location of the next target and the game continues.

In Figure 3b, we depict a strategy, *strat*, for the robot. Intuitively, the strategy represents the robot patrolling left and right within a bounded region (i.e., *strat* instructs the robot to move five units right, then five units left, and repeat). To verify that the strategy *strat* results in the robot winning the game in Figure 3, we generate the following verification query:

$$Sem^{FP}_{Buchi} \models \neg\overline{Buchi}(strat, 0, 0) \tag{3}$$

We call the semantic relation $\overline{Buchi}$ because its dual (along with *Reach*) defines a Büchi game. In general, Büchi games are played between two players—the first player tries to reach a goal infinitely often, while the second player tries to thwart the first player. Intuitively, the right-hand side of the verification query encodes when the robot (using the strategy *strat*) wins the Büchi game; the left-hand side of the query (Sem^{FP}_{Buchi}) defines the rules of the game. Intuitively, *Reach* encodes that the robot must eventually satisfy the Büchi condition (i.e., denoted by $\neg\overline{Buchi}(strat, x, y)$). The Büchi condition is represented implicitly as the negation of its dual $\overline{Buchi}$. The Büchi condition

states that the robot must have reached the target, and then an adversary gets to chose a new target and the game repeats, forever.

To solve the verification query in Equation (3), we must compute the fixed-point of Sem_{Buchi}. Unlike the previous examples, the semantics Sem_{Buchi} is not defined using CHCs—most notably due to the negative occurrences of $\overline{Buchi}$ and *Reach* within the premise of the semantic rules. In fact, Sem_{Buchi} does not even define a least fixed-point (due to the negative occurrences). However, such alternating least and greatest fixed-points can be defined in μCLP. Our tool Muse dispatches this query to the μCLP validity solver MuVal [39], which proves the above query valid in 21s, thereby proving that the strategy *strat* is a valid solution to the SemGuS$^\mu$ synthesis problem in Figure 3.

3 SemGuS and SemGuS$^\mu$

This section reviews the SemGuS framework [26] and describes the more expressive framework SemGuS$^\mu$ we propose. A SemGuS synthesis problem is defined in three parts: a grammar defining the syntax of the language over which programs are to be synthesized (§3.1), a set of logical formulas defining the semantics of programs in the language (§3.2), and a specification defining the properties the synthesized program should exhibit (§3.3).

3.1 Syntax as Regular Tree Grammars

The syntax of a programming language is defined as a typed regular tree grammar (RTG). A *ranked alphabet* is a tuple $\langle \Sigma, rk_\Sigma \rangle$ consisting of a finite set of symbols (Σ) and a function $rk_\Sigma : \Sigma \to \mathbb{N}$ that associates every symbol with a rank. For any $n \geq 0$, $\Sigma^n \subseteq \Sigma$ denotes the set of symbols of rank n. The set of all *(ranked) Trees* over Σ is denoted by T_Σ. Specifically, T_Σ is the least set such that $\Sigma^0 \subseteq T_\Sigma$ and 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 the remainder, we assume a fixed ranked alphabet $\langle \Sigma, rk_\Sigma \rangle$.

Definition 3.1 (Regular Tree Grammar). A *typed Regular Tree Grammar* (RTG) is a tuple $G = \langle N, \Sigma, T, a, \delta \rangle$, where N is a finite set of non-terminal symbols of rank 0; Σ is a ranked alphabet; $T = \{\tau_0, \ldots, \tau_k\}$ is a finite set of types; a is a type assignment assigning each non-terminal to a type and each symbol of rank i to a tuple of of types $\langle \tau_0, \ldots, \tau_i \rangle \in T^{i+1}$; and δ a finite set of productions of the form $A_0 \to \sigma^i(A_1, \ldots, A_i)$ such that for all $0 \leq j \leq i$, $A_j \in N$ is a non-terminal and if $a_{\sigma^i} = \langle \tau_0, \ldots, \tau_i \rangle$ then $a_{A_j} = \tau_j$.

Given a tree $t \in T_{\Sigma \cup N}$, one may apply the production rule $r = A \to \beta \in \delta$ to t to produce a tree t' by replacing the leftmost occurrence of A in t with β. A tree $t \in T_\Sigma$ is generated by the grammar G ($t \in L(A)$) when t is the result of applying some sequence of production rules $r_0, \ldots, r_n \in \delta^n$ to a non-terminal $A \in N$.

Example 3.2 (RTG). For example, the syntax of programs considered in Figure 2a represents a regular tree grammar. It consists of the nonterminals L, S, and B; ranked symbols $\texttt{while}^2$, $\texttt{x++}^0$, $\texttt{x--}^0$, $\texttt{y++}^0$, $\texttt{y--}^0$, $\texttt{seq}^2$, $\texttt{0 < x}^0$, and $\texttt{0 < y}^0$, and productions $L \to \texttt{while}(B, S)$, $S \to \texttt{x++}$, $S \to \texttt{x--}$, $S \to \texttt{y++}$, $S \to \texttt{y--}$, $S \to \texttt{seq}(S, S)$, $B \to \texttt{0 < x}$, and $B \to \texttt{0 < y}$. In the examples in the paper, we often drop the ranks of symbols and use infix notation to enhance readability.[1]

3.2 Semantics via Logical Relations and Fixed-Point Logics

We begin by reviewing some necessary details of the fragments of first-order logic we use in this paper. Given a (possibly multi-sorted) first-order theory $\mathcal{T}$ over a signature Σ, the syntax of

[1]The grammars used in the paper are referred to at various places as "grammars" or "regular-tree grammars" (Defn. 3.1). The trees/terms in the language of a grammar would be represented using algebraic data types. In the logics used in the paper (CHCs, co-CHCs, and μCLP), we implicitly assume that one can use values in the algebraic data type to express tree-valued constants.

formulas and terms are given by the following grammar:

$$\varphi ::= X(t_1, \ldots, t_{rk_\Sigma(X)}) \mid p(t_1, \ldots, t_{rk_\Sigma(p)}) \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \forall x : s.\varphi_1$$
$$t ::= x \mid f(t_1, \ldots, t_{rk_\Sigma(f)})$$

where x and X are term and predicate variables, respectively; f and p are function and predicate symbols of Σ; and s is a sort of Σ. Disjunction, implication, existential quantification, etc. are omitted from the syntax and may be defined as expected (e.g., $\varphi \vee \psi \triangleq \neg(\neg\varphi \wedge \neg\psi)$). We will use φ and ψ to refer to possibly quantified formulas, and F and G to refer to quantifier-free formulas. We use $FV(\varphi)$ and $FV(t)$ to denote the free variables of a formula and term, respectively. Given a formula φ, variable x, and term t, we use $\varphi[x \mapsto t]$ to denote φ with every free occurrence of x replaced with t. Additionally, for a set of variables V, we use $\varphi[V \mapsto c_x]$ to represent replacing every free occurrence of each $x \in V$ with a constant c_x.

A constrained Horn clause (CHC) is a formula over some background theory of the form:

$$\forall \bar{x}_0, \ldots, \bar{x}_n.X_0(\bar{x}_0) \leftarrow X_1(\bar{x}_1) \wedge \cdots \wedge X_n(\bar{x}_n) \wedge F(\bar{x}_0, \ldots, \bar{x}_n), \tag{4}$$

where each $\bar{x}_i$ is a sequence of term variables, X_i is a predicate variable, and F is a constraint over the variables in each predicate. In the remainder of the paper, we abuse notation and allow arbitrary first-order terms to appear as arguments to each X_i.

In the SemGuS framework originally defined by Kim et al. [26], the semantics of programs in the language defined by the regular tree grammar is provided by defining a logical relation and using CHCs over some theory, to define the elements of the relation by giving rules for each of the productions of the grammar. As discussed in §2.3, in our work, we represent the input semantics in a logic more expressive than CHCs—in particular, relations are ordered, and their definitions can include quantified variables and conjunctive and disjunctive combinations of constraints and positive and negative occurrences of semantic relations.

Definition 3.3 (SemGuS$^\mu$ semantics). Given a first-order theory $\mathcal{T}$ and regular tree grammar $G = \langle N, \Sigma, T, a, \delta\rangle$ a semantics for G is a tuple $\left\langle SEM, <_{SEM}, [\![\cdot]\!]\right\rangle$ where SEM maps each non-terminal $A \in N$ to a non-empty finite set of uninterpreted relations ($SEM_A = \{Sem_A^1, \ldots, Sem_A^n\}$), $<_{SEM}$ is a total ordering over all semantic relations, and $[\![\cdot]\!]$ maps each production rule $A_0 \to \sigma^i(A_1, \ldots, A_i)$ of type $(\tau_0, \ldots, \tau_i)$ and semantic relation $Sem_{A_0}^j \in SEM_{A_0}$ to a formula of the form $Sem_{A_0}^j(t_{A_0}, \Gamma^{0,j}, \Upsilon^0) \leftarrow \varphi$ such that:

- φ is a (possibly quantified) $\mathcal{T}$ formula,
- t_{A_0} is a variable representing elements of $L(A_0)$, Υ^0 is a variable of type τ_0, and $\Gamma^{0,j}$ are variables representing state,
- φ's free variables belong to $\Gamma^{0,j}$, Υ^0, or $\{t_{A_0}\}$,
- $Sem_{A_0}^j$ does not appear negatively within φ (either directly within φ or indirectly within the definition of the semantic relations appearing in φ), and
- For each $Sem_{A_k}^l(t_{A_k}, \Gamma^{k,l}, \Upsilon^k)$ appearing in φ:
  - $0 \le k \le i$ and $Sem_{A_k}^l \in SEM_{A_k}$ and
  - t_{A_k}, Υ^k, and $\Gamma^{k,l}$ are defined analogously to t_{A_0}, Υ^0, $\Gamma^{0,j}$.

Example 3.4. Consider the semantics $Sem_{loop} = \left\langle SEM, <_{SEM}, [\![\cdot]\!]\right\rangle$ in Figure 2d. Each non-terminal is mapped to a single semantic relation (i.e., $SEM_L = \{Sem_L\}$, $SEM_S = \{Sem_S\}$, and $SEM_B = \{Sem_B\}$). In this instance, order does not matter, and we can assume an arbitrary order, such as $Sem_L <_{SEM} Sem_S <_{SEM} Sem_B$. The semantic function $[\![\cdot]\!]$ maps each semantic relation Sem_A and production rule $A \to \sigma^i(A_1, \ldots, A_i)$ to the semantic relation whose head is of the form $Sem_A(\sigma^i(t_1, \ldots, t_n), \Gamma, \Upsilon)$. For example, $[\![\texttt{0 < x}]\!]_{Sem_B}$ is the rule $Sem_B(\texttt{0 < x}, x, y, b) \leftarrow 0 < x$.

Our semantics generalizes the semantic rules considered by Kim et al. [26] in three ways:

(1) It allows each nonterminal to be associated with *multiple* semantic relations—e.g., to describe the multiple relations appearing in the example from Figure 3.
(2) The rules defining the semantic relations are expressed in a fragment of first-order logic that goes beyond CHCs—e.g., to describe the rules that define $Reach_T$ used in Figure 3.
(3) Each semantic relation is ordered to ensure that the semantic rules correspond to a unique fixed-point describing the semantics of the language.

If we restrict our semantic definition to have a single semantic relation per non-terminal and to rules of the form $Sem_A(t_A, \Gamma, \Upsilon) \leftarrow \varphi$, where φ contains only existential quantification and positive occurrences of semantic relations, then our definition is equivalent to the semantics considered in SemGuS [26]. Note that, while we allow only one rule per production per semantic relation, we do allow for the disjunction of semantic relations within the premise of a rule, thereby recovering equivalent expressiveness to allowing multiple rules per production rule. In the remainder of this paper, to enhance readability we omit variable types from semantic relations when appropriate (i.e., we write $Sem_A(t, \overline{x})$ instead of $Sem_A(t, \Gamma, \Upsilon)$). The robot-reachability synthesis problem considered in Figure 3 cannot be encoded in SemGuS, but can be encoded in SemGuS$^\mu$.

3.3 Specifications and SemGuS$^\mu$ Problems

Now that we have a way to define the syntax and semantics of the programming language over which we are trying to synthesize programs, all that is missing to define a SemGuS$^\mu$ problem is the specification we want the synthesized program to satisfy.

Definition 3.5 (SemGuS$^\mu$ problem, solution, validity, realizable). A SemGuS$^\mu$ problem is a tuple $\mathcal{P} = \left\langle G = \langle N, \Sigma, T, a, \delta\rangle, \left\langle SEM, <_{SEM}, [\![\cdot]\!]\right\rangle, F, \varphi\right\rangle$, where

- G is a regular tree grammar.
- $\left\langle SEM, <_{SEM}, [\![\cdot]\!]\right\rangle$ is a semantics for G.
- F is a finite set of functions we want to synthesize—pairs of the form $\langle f, A\rangle$ where f is a variable representing a procedure that we want to synthesize, and $A \in N$ is the root nonterminal from which f is to be derived—i.e., the solution for f must be a tree $t \in L(A)$.
- φ a specification in the theory $\mathcal{T}$ such that
  - The free variables of φ must be functions to synthesize, $FV(\varphi) \subseteq \{f : \langle f, A\rangle \in F\}$ and
  - For any $\langle f, A\rangle \in F$, f appears only in atoms of the form $Sem^i_A(f, \overline{x})$ where $Sem^i_A \in SEM_A$.

For a semantics $\left\langle SEM, <_{SEM}, [\![\cdot]\!]\right\rangle$, an interpretation ρ is a function that maps each semantic relation $Sem^l_A(t, \overline{x}) \in SEM$ to a formula whose free variables are $\overline{x} \cup \{t\}$. The interpretation SEM^{LFP} is the interpretation that maps each semantic relation to its least solution.

A *solution* to the SemGuS$^\mu$ problem $\mathcal{P}$ is a function S that maps each $\langle f, A\rangle \in F$ to a tree $t \in L(A)$. The solution S is *valid* when $SEM^{LFP} \models \varphi[\langle f, A\rangle \in F.f \mapsto S(f)]$. Note that the values $S(f)$ being substituted into the formula are program-valued constants represented as terms in the algebraic data type for G. Moreover, by the last case of Definition 3.5, each occurrence of f in φ is in an atom of the form $Sem^i_A(f, \overline{x})$ where $Sem^i_A \in SEM_A$. Consequently, in the resulting formula, each such program-valued constant will be interpreted according to the least fixed-point of a semantic relation of an appropriate kind. Note that, while SEM^{LFP} appears to use only least fixed-points, because we allow semantic relations to appear negated within the premise of a semantic rule, SEM^{LFP} represents arbitrarily nested greatest and least fixed-points whose top-level fixed-point is a least fixed-point. We say that $\mathcal{P}$ is *realizable* if there exists a valid solution to $\mathcal{P}$.

In SemGuS$^\mu$ we allow multiple (mutually recursive) semantic relations to afford a SemGuS$^\mu$ user flexibility when defining a semantics. For example, consider Figure 3, which uses this flexibility to

define the semantics of a Büchi game. Without multiple semantic relations, we could not define both $\overline{Sem}_{Büchi}$ and Sem_{Reach}. Furthermore, because we allow semantic relations to appear negatively within the body of semantic rules, the order of semantic rules can affect the interpretation of the rules! For example, consider the mutually recursively defined semantic relations Sem_A and Sem_B, whose semantics follows a similar pattern to the semantic relations appearing in Figure 3d:

$$Sem_A(x) \leftarrow Sem_A(x) \vee \neg Sem_B(x)$$
$$Sem_B(x) \leftarrow Sem_B(x) \vee \neg Sem_A(x)$$

The least fixed-point of Sem_A requires first computing the greatest fixed-point of $\overline{Sem}_B$, and similarly the least fixed-point of Sem_B requires first computing the greatest fixed-point of $\overline{Sem}_A$. To ensure that the semantic rules correspond to a unique fixed-point, we fix the order of evaluation of the semantic relations. For complete details we refer the reader to [39]. If $Sem_A <_{SEM} Sem_B$, then Sem_A is defined by the fixed-point $Sem_A(x) =_\mu Sem_A(x) \vee \overline{Sem}_B(x); \overline{Sem}_B(x) =_\nu \overline{Sem}_B(x) \wedge Sem_A(x)$ (which evaluates Sem_A to $\emptyset$). Otherwise, Sem_A is defined by $\overline{Sem}_B(x) =_\nu \overline{Sem}_B(x) \wedge Sem_A(x); Sem_A(x) =_\mu Sem_A(x) \vee \overline{Sem}_B(x)$ (which evaluates Sem_A to $\mathbb{Z}$).

4 Verifying Candidate Programs

This section formalizes the three methods used in §2 to verify that a program is a valid solution to a SemGuS problem. Each technique encodes when the program is a valid solution to the SemGuS problem in a fragment of first-order logic (with fixed-points). We describe each of the three encodings, and characterize the kinds of SemGuS verification problems on which they can be applied (§§ 4.1 to 4.3). Additionally, we prove that the SemGuS$^\mu$ framework described in §3 can be used to define verification problems that require the full capabilities of μCLP. We now describe each encoding in turn. In the remainder of this section, we consider a fixed SemGuS$^\mu$ problem $\mathcal{P} = \left\langle G = \langle N, \Sigma, S, T, a, \delta\rangle, \left\langle SEM, [\![\cdot]\!]\right\rangle, F, \varphi\right\rangle$ and candidate solution P.

4.1 Encoding Nonrecursive SemGuS$^\mu$ Verification Problems with Quantified SMT

In §2.1, we were able to produce a first-order-logic formula that is free of any semantic relations and is satisfiable exactly when φ_{max2} is a valid solution to the SemGuS problem displayed in Figure 1. We could obtain such a formula because the derivation tree of the semantics of φ_{max2} is finite. To formalize this intuition, we define two auxiliary notions: when a semantic relation is non-recursive on tree t, and when a semantic relation is a t-ancestor of another semantic relation—i.e., when the semantic relations are not recursive on the program term.

Definition 4.1 (t-ancestor, non-recursive on t). Let $A \in N$ be any non-terminal, $t \in L(A)$ be a tree of the form $t = \sigma^i(t_1, \ldots, t_i)$ for some production rule $A \to \sigma^i(A_1, \ldots, A_i)$, semantic relation $Sem_A \in SEM_A$ of A, and $[\![A \to \sigma^i(A_1, \ldots, A_i)]\!]_{Sem_A} = Sem_A(t, \overline{x}) \leftarrow \varphi$.

We say that a semantic relation $Sem'_A \in SEM_A$ is a *t-ancestor* of Sem_A if and only if (i) $Sem'_A(t, \overline{x})$ appears in the antecedent φ for some $\overline{x}$, or (ii) there is some symbol $Sem''_A(t, \overline{x})$ that appears in φ and Sem'_A is a t-ancestor of Sem''_A.

We say that Sem_A is *non-recursive on t* if (i) Sem_A is not a t-ancestor of itself, and (ii) for each $Sem_{A_j}(t_j, \overline{x})$ appearing in φ, Sem_{A_j} is non-recursive on t_j. If Sem_A is non-recursive on t then for any arguments $\overline{x}$, the derivation tree of $Sem_A(t, \overline{x})$ has finite height.

Definition 4.2 (Formula of). Given a non-terminal $A \in N$, a semantic relation $Sem_A \in SEM_A$, and a production $A \to \sigma^i(A_1, \ldots, A_n) \in \delta$, if $[\![A \to \sigma^i(A_1, \ldots, A_n)]\!]_{Sem_A}$ is of the form $Sem_A(t, \overline{x}) \leftarrow \varphi$, then the formula of $Sem_A(t', \overline{x}')$ (denoted by φ-$of\,(Sem_A(t', \overline{x}'))$) is $\varphi[t \mapsto t', \overline{x} \mapsto \overline{x}']$, which replaces the formal arguments of Sem_A with the actual arguments of the application.

Procedure SMT-FORMULA-OF($\mathcal{P} = \langle G, \langle SEM, [\![\cdot]\!] \rangle, F, \varphi \rangle, S$)
  $rules \leftarrow \top$ // empty set of rules to begin with
  $\varphi \leftarrow \varphi[\langle f, A \rangle \in F.f \mapsto S(f)]$ // substitute solution into specification
  foreach $Sem_A(t, \overline{x})$ *appearing in* φ **do**
    $\psi \leftarrow \varphi\text{-}of(Sem_A(t, \overline{x}))$ // repeatedly replace semantic relations with their def.
    while $Sem_{A'}(t', \overline{x}')$ *appears in* ψ **do**
      $\psi \leftarrow \psi[Sem_{A'}(t', \overline{x}') \mapsto \varphi\text{-}of(Sem_{A'}(t', \overline{x}'))]$
    $rules \leftarrow rules \wedge (Sem_A(t, \overline{x}) \Leftrightarrow \psi)$ // update rules to add definition for Sem_A
  return $\langle rules, \varphi \rangle$ // $rules^{LFP} \models \varphi$ if and only if S is valid solution to $\mathcal{P}$

We now turn to defining the procedure SMT-FORMULA-OF that encodes that a solution P is valid for a SemGuS$^{\mu}$ problem (where the semantics is non-recursive on P) into first-order logic without fixed-points (i.e., quantified SMT formulas). SMT-FORMULA-OF repeatedly replaces every occurrence of a semantic relation with the premise of the rule that defines it.

Applying SMT-FORMULA-OF to the verification problem in §2.1 yields the formula in Equation (1). The following theorem states under which conditions SMT-FORMULA-OF$(\mathcal{P}, P)$ returns a formula that is satisfiable if and only if P is a valid solution to $\mathcal{P}$.

THEOREM 4.3 (SMT-FORMULA-OF IS SOUND). *For any SemGuS$^{\mu}$ problem* $\mathcal{P} = \langle G = \langle N, \Sigma, S, T, a, \delta \rangle, \langle SEM, [\![\cdot]\!] \rangle, F, \varphi \rangle$ *and solution* P *of* $\mathcal{P}$*, if* Sem_A *is non-recursive on* $P(f)$ *for each occurrence of* $Sem_A(f, \overline{x})$ *within the specification* φ*, then* SMT-FORMULA-OF$(\mathcal{P}, P)$ *is valid if and only if* P *is a valid solution of* $\mathcal{P}$*.*

4.2 Encoding CHC-like SemGuS$^{\mu}$ Verification Problems with CHCs

In §2.2, we saw how to encode the SemGuS verification problem from Figure 2 into the CHC fragment of first-order logic when using the specification φ_{loop} from Figure 2c. In this section, we formalize when and how a SemGuS verification problem may be encoded with CHCs.

To encode the verification problem a CHC decision problem, we require the semantics of the solution to be equivalent to a set of CHCs (i.e., formulas of the form described in Equation (4)). For CHCs the decision problem of interest is "given a set of CHCs and a query formula of the form $\forall \bar{x}.R(x) \Rightarrow \varphi$, determine if the query is derivable from the set of CHCs"—or, equivalently, "determine if some interpretation of the uninterpreted relations satisfies all relations and the given query formula" [9]. Furthermore, for the given class of queries, the problem is also equivalent to determining if the least solution to the set of CHCs satisfies the desired query [24]. We use this final notion to formulate our verification procedure CHC-OF.

Definition 4.4 (CHC-like). Let $A \in N$ be any non-terminal, $t \in L(A)$ be a tree of the form $t = \sigma^i(t_1, \ldots, t_i)$ for some production rule $A \to \sigma^i(A_1, \ldots, A_i)$, $Sem_A \in SEM_A$ a semantic relation of A, and $[\![A \to \sigma^i(A_1, \ldots, A_i)]\!]_{Sem_A} = Sem_A(t, \overline{x}) \leftarrow \varphi$.

We say the rules defining Sem_A are *CHC-like for* t if and only if (*i*) φ has no negative occurrences of a semantic relation, (*ii*) φ contains no universal quantifiers, and (*iii*) for every $Sem_{A_j}(t_j, \overline{x}_j)$ appearing in φ, the rules defining Sem_{A_j} are CHC-like for t_j.

For example, the rules defining both Sem_{max2} and Sem_{loop} in Figures 1d and 2d are CHC-like (for any program within their respective grammars), while the rules for Sem_{Buchi} in Figure 3d are not. We now define the procedure CHC-OF, which encodes as a CHC satisfiability problem the property that a solution P is valid for a SemGuS$^{\mu}$ problem (where the semantics is CHC-like for P). We first define an auxiliary function *rules-of* that, given a semantic relation Sem_A and tree $t \in L(A)$, returns a set of CHCs logically equivalent to the semantic rule defining Sem_A for the

Procedure CHC-OF($\langle G = \langle N, \Sigma, S, T, a, \delta\rangle, \langle SEM, [\![\cdot]\!]\rangle, F, \varphi\rangle, S$)
  $rules \leftarrow \top$;
  $\psi \leftarrow \varphi[\langle f, A\rangle \in F.f \mapsto S(f)]$;
  $Q \leftarrow \{\langle Sem_A, t\rangle : Sem_A(t, \overline{x})$ appears in $\psi\}$; // Queue of relations that need defining
  while $Q \neq \emptyset$ **do**
    $\langle Sem_A, t'\rangle \leftarrow$ **pick** Q ;
    $rules' \leftarrow rules\text{-}of\,(Sem_A, t')$; // Definition of Sem_A for t' as a set of CHCs
    $Q \leftarrow Q \cup \{\langle Sem_{A_j}, t_j\rangle : Sem_{A_j}(t_j, \overline{x}_j)$ appears negatively in $rule[t \mapsto t']$
      for some rule in $rules'\}$;
    $rules \leftarrow rules \wedge \bigwedge rules'$; // Add rules defining Sem_A for t' to $rules$
  return $\langle rules, \psi\rangle$; // $rules^{LFP} \models \psi$ if and only if S is a valid solution to $\mathcal{P}$

root production of t. Let $Sem_A(t, \overline{x}) \leftarrow \varphi = [\![A \rightarrow \sigma^i(A_1, \ldots, A_i)]\!]_{Sem_A}$ and let $A \rightarrow \sigma^i(A_1, \ldots, A_i)$ denote the root production of t. Under our assumptions (that Sem_A is CHC-like on t), we may assume that φ is a formula that may contain existential quantification and arbitrary disjunction and conjunction of positive occurrences of semantic relations. Without loss of generality, we assume that any existentially bound variable in φ is uniquely named and does not appear in $\overline{x}$ (otherwise, it can be renamed). Let F be the quantifier-free formula obtained by erasing all existential quantifiers from φ. Then $dnf(\varphi)$ is the set of disjuncts of the dnf of F. Finally, define $rules\text{-}of\,(SemA, t)$ to be the set $\{Sem_A(t, \overline{x}) \leftarrow \psi : \psi \in dnf(\varphi)\}$. The CHC-OF procedure produces a set of CHCs that captures the semantics of each candidate program t by effectively performing a breadth-first search on each of the semantic relations that define the semantics of each sub-program of t. In Theorem 4.5, we state conditions under which CHC-OF is sound.

While one might expect the CHC-encodable fragment of SemGuS$^{\mu}$ to subsume the SMT encodable fragment of SemGuS$^{\mu}$—e.g., by encoding the output of SMT-FORMULA-OF into an equi-satisfiable set of CHCs—the two fragments are incomparable. Rather than performing complex encodings, both SMT-FORMULA-OF and CHC-OF consider fragments of SemGuS$^{\mu}$ that can be naturally encoded into SMT and CHCs, respectively. Specifically, SMT-FORMULA-OF only considers programs whose semantics recurses on structurally smaller programs (even if semantic relations appear positively or negatively in the specification or the semantic definitions). In contrast, CHC-OF only considers semantics that can be automatically translated into an equivalent set of CHCs (i.e., by restricting semantic rules to only include existential quantifiers and positive occurrences of other semantic relations, even those that recurse on non-structurally decreasing terms). For example, SMT-FORMULA-OF can encode the SemGuS problem found in fig. 1, because it involves a non-recursive program but could not encode the SemGuS problem found in fig. 2 because it requires reasoning about a recursive program. In contrast, CHC-OF cannot encode the example in fig. 1, because the specification includes both positive and negative occurrences of a semantic relation, whereas CHC-OF can encode the example in fig. 2 because the semantics is represented as CHCs and the specification only contains a negative occurrence of a semantic relation.

THEOREM 4.5 (CHC-OF IS SOUND.). *For any SemGuS$^{\mu}$ problem $\mathcal{P} = \langle G = \langle N, \Sigma, T, a, \delta\rangle, \langle SEM, <_{SEM}, [\![\cdot]\!]\rangle, F, \varphi\rangle$ and solution P of $\mathcal{P}$, if for each occurrence of $Sem_A(f, \overline{x})$ within the specification φ, it appears negatively and the rules defining Sem_A are CHC-like for $P(f)$, then the query returned by CHC-OF$(\mathcal{P}, P)$ is valid if and only if P is a valid solution of $\mathcal{P}$.*

Conversely, in Theorem 4.6, we prove that, in general, a more expressive fragment of first-order logic with fixed-points is required to verify solutions of an arbitrary SemGuS problems. In particular, we need a fragment of the μCLP calculus that only uses least fixed-point equations.

Procedure MUCLP-OF($\mathcal{P} = \langle G, \langle SEM, <_{SEM}, [\![\cdot]\!] \rangle, F, \varphi \rangle, S$)
  $rules \leftarrow \top$;
  $\psi \leftarrow Norm(\varphi[\langle f, A \rangle \in F.f \mapsto S(f)])$;
  $Q \leftarrow \{\langle Sem_A, t, \mu \rangle : Sem_A(t, \overline{x})$ appears in $\psi\} \cup$
    $\{\langle Sem_A, t, \nu \rangle : \overline{Sem_A}(t, \overline{x})$ appears in $\psi\}$;
  while $Q \neq \emptyset$ **do**
    $\langle Sem_A, t', fix \rangle \leftarrow$ **pick** Q ;
    $rule \leftarrow head\text{-}of(Sem_A, t') =_\mu Norm(body\text{-}of(Sem_A, t'))$; // Compute as LFP
    if $fix = \nu$ **then**
      $rule \leftarrow dual(rule)$; // Dualize to GFP
    $Q \leftarrow Q \cup \{\langle Sem_{A_j}, t_j, \mu \rangle : Sem_{A_j}(t_j, \overline{x}_j)$ appears in body of $rule[t \mapsto t']\}$;
    $Q \leftarrow Q \cup \{\langle Sem_{A_j}, t_j, \nu \rangle : \overline{Sem_{A_j}}(t_j, \overline{x}_j)$ appears in body of $rule[t \mapsto t']\}$;
    $rules \leftarrow rules \cup \{rule\}$;
  $rules \leftarrow rules$ sorted by $<_{SEM}$; // According to the head predicate of each rule.
  return $\langle rules, \psi \rangle$; // $rules^{FP} \models \psi$ if and only if S is a valid solution to $\mathcal{P}$

Theorem 4.6 (Verification of Semgus is not reducible to CHC Satisfiability). *There exists a program t and SemGuS problem Sy such that verifying t satisfies Sy cannot be reduced to satisfiability of Constrained Horn Clauses.*

4.3 Encoding all SemGuS$^\mu$ Verification Problems with μCLP

In §2.2 and §2.3, we examined two SemGuS$^\mu$ verification problems for which there is no possible encoding as fixed-point-free formulas (i.e., SMT or CHCs). Instead, these problems were encoded into a first-order fixed-point logic, μCLP, that allows defining both greatest and least fixed-points. Unlike the previous encodings, for any SemGuS$^\mu$ problem $\mathcal{P}$ one can *always* use μCLP to encode that P is a valid solution to $\mathcal{P}$. A μCLP formula is a sequence of formulas of the form:

$$X_0(\bar{x_0}) =_{fix_0} \varphi_0 \quad \ldots \quad X_n(\bar{x_n}) =_{fix_n} \varphi_n,$$

where each X_i is an uninterpreted relation, $\bar{x_i}$ is a sequence of term variables, and the φ_i are formulas within some background theory whose free variables are $\bar{x_i}$ and which may include positive occurrences of the uninterpreted relations $X_0, \ldots, X_n$. Each fix_i is either μ or ν referring to whether or not the equation $X_i(\bar{x_i}) =_{fix_i} \varphi_i$ should represent a least or greatest fixed-point, respectively. We refer the reader to Unno et al. [39] for a detailed formalization of μCLP.

In the SemGuS$^\mu$ semantics (Definition 3.5), every semantic relation's definition is oriented as a least fixed-point. However, our semantics does allow one to introduce greatest fixed-points by taking the negation of a semantic relation. We now turn to defining MUCLP-OF, which encodes as a μCLP query the property that a solution P is valid for a SemGuS$^\mu$ problem $\mathcal{P}$. The procedure is similar to CHC-OF, in that it performs a breadth-first search over the semantic relations to produce the resulting μCLP query. For a formula φ, we use $Norm(\varphi)$ to denote the formula φ wherever a negative occurrence of $Sem_A(t, \overline{x})$ is replaced by $\neg\overline{Sem_A}(t, \overline{x})$ (i.e., so that $\overline{Sem_A}(t, \overline{x})$ appears positively in $Norm(\varphi)$). Next, we define $dual(Sem_A(t, \overline{x}) =_\alpha \varphi)$ to be the dual fixed-point equation $\overline{Sem_A}(t, \overline{x}) =_{\overline{\alpha}} Norm(\neg\varphi)$ where $\overline{\mu} = \nu$ and $\overline{\nu} = \mu$. For all semantic relations Sem_A and programs $t \in L(A)$, let $head\text{-}of(Sem_A, t) \triangleq Sem_A(t, \overline{x})$ and $body\text{-}of(Sem_A, t) \triangleq \varphi$, where $Sem_A(t, \overline{x}) \leftarrow \varphi$ is the semantic relation that defines the semantics of the root production of t.

Theorem 4.7 states that MUCLP-OF soundly encodes any SemGuS and SemGuS$^\mu$ problem into a validity query within the μCLP calculus.

Theorem 4.7 (muclp-of is sound). *For any SemGuS$^\mu$ problem $\mathcal{P} = \langle G, \langle SEM, <_{SEM}, [\![\cdot]\!]\rangle, F, \varphi\rangle$ and solution P of $\mathcal{P}$, the query returned by* muclp-of$(\mathcal{P}, P)$ *is valid iff P is a valid solution of $\mathcal{P}$.*

Theorem 4.8, states that the SemGuS$^\mu$ semantics can express any μCLP query—i.e., that any μCLP validity query can be equivalently reduced to a SemGuS$^\mu$ verification problem. Thus, SemGuS$^\mu$ can encode any problem that can be encoded within the μCLP calculus.

Theorem 4.8 (SemGuS$^\mu$ semantics and μCLP are equally expressive). *For every μCLP query $\langle \varphi, preds\rangle$, there is some SemGuS problem $\mathcal{P}$ and solution $P \in L(G_{\mathcal{P}})$ such that $\langle \varphi, preds\rangle$ is valid if and only if P is a valid solution to $\mathcal{P}$.*

Conversely, in Theorem 4.9, we state that verification for SemGuS problems does not require the full generality of μCLP. Specifically, SemGuS verification problems do require a fragment of first-order logic (with fixed-points) beyond both CHCs and coCHCs, but do not require arbitrary alternations of greatest and least fixed-points. As a corollary of Theorems 4.8 and 4.9, SemGuS$^\mu$ is more expressive than SemGuS.

Theorem 4.9 (SemGuS and μCLP are not equally expressive). *Verifying solutions to SemGuS problems can be encoded within a fragment of μCLP that uses at most one alternation between greatest and least fixed-points.*

5 Implementation

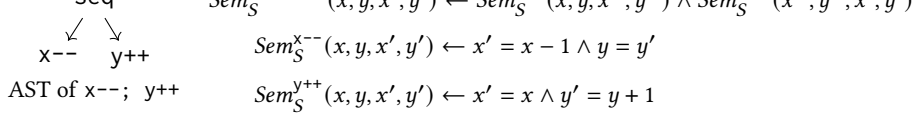
We implement our algorithms in a tool, called Muse, which extends SemGuS to SemGuS$^\mu$. Muse supports all of the encoding schemes for the three classes of problems discussed in §4.

Muse is implemented in OCaml, and uses Z3 for SMT formulas [10], Spacer for CHCs [28], and MuVal for μCLP queries [39]. As part of implementing Muse, we extended the implementation of MuVal to support algebraic data types, which we use to represent programs in first-order logic.

The remainder of this section describes three optimizations that one may apply to transform the encodings described in §4. The goal of these optimizations is to use knowledge of the SemGuS verification problem to make the resulting optimized queries simpler.

Reification of Terms in Semantic Relations. In our verification problems, we have a specific concrete program (or programs) whose semantics we wish to capture using the semantic relations. The goal of the first optimization reify is to eliminate program terms from the semantic relations, thus removing the burden of the solver to reason about terms using the theory of algebraic data types.

Example 5.1. Consider the program $t \equiv$ `x--; y++` from the language Imp_{loop} described in Figure 2. Below we depict the AST of t and show the reified semantics of Sem_{loop} specialized to t. Because the program is known *a priori*, the semantics can be reified to remove the AST-valued argument in the different *Sem* relations by introducing a new semantic relation for each node of the AST.



More formally, the reified semantics introduces a new semantic relation for every sub-tree of the program's AST. Each occurrence of $Sem_{A_j}(t_j, \overline{x}_j)$ is then replaced by $Sem_{A_j}^{t_j}(\overline{x}_j)$.

Definition 5.2 (Reified Semantics). Given a non-terminal A, a program $t \in L(A)$, a semantic relation Sem_A, and a set of semantic rules *rules* defining the semantics of Sem_A, the *semantics of Sem_A reified to t* is a pair $\text{reify}(rules, Sem_A, t) = \langle SEM^{reify}, rules^{reify}\rangle$ such that SEM^{reify} and $rules^{reify}$ are the least solution to the following rules:

(1) Sem^t_A is a reified semantic relation ($Sem^t_A \in SEM^{reify}$),
(2) if $Sem^{t'}_A \in SEM^{reify}$ is a reified semantic relation, t' is of the form $\sigma^i(t_1, \ldots, t_i)$, and there is a rule of the form $Sem_A(t', \overline{x}') \leftarrow \varphi \in rules$, then $Sem^{t'}_A(\overline{x}') \leftarrow \varphi[Sem_{A_j}(t_j, \overline{x}_j) \mapsto Sem^{t_j}_{A_j}(\overline{x}_j)] \in rules^{reify}$ is a reified semantic rule, and
(3) if $Sem^{t'}_A(\overline{x}') \leftarrow \varphi \in rules^{reify}$ is a reified semantic rule and $Sem^{t_j}_{A_j}$ appears in φ, then $Sem^{t_j}_{A_j} \in SEM^{reify}$ is a reified semantic relation.

Theorem 5.3 (Reification is sound). *Given a formula φ and a set, rules, of semantic rules, let ψ be the formula in which every occurrence of $Sem_A(t, \overline{x})$ is replaced by the reified semantic relation $Sem^t_A(\overline{x})$, and $rules^{reify}$ is the conjunction of the reified semantic rules produced by* reify*$(rules, Sem_A, t)$ for each $Sem_A(t, \overline{x})$ appearing in φ. The constraint φ is valid under the original semantic rules rules if and only if ψ is valid under the reified semantic rules: $rules \models \varphi \Leftrightarrow rules^{reify} \models \psi$.*

Semantic Relation Inlining. In general, the MuVal solver that we use to solve μCLP queries scales poorly in the number of relations used to define the semantics of a program. The goal of the optimization inline is to eliminate semantic relations by inlining their definitions.

The SMT encoding smt-formula-of in §2.1 can be seen as an application of inline that eliminates semantic relations by inlining their meaning into a quantified first-order formula (i.e. $\varphi_{Sem_{max2}}$ in §2.1). The optimization inline applies a similar insight to as many semantic relations as possible to reduce the number of semantic relations that the final solver has to deal with.

The inlining optimization inline is implemented nearly identically to the smt-formula-of encoding defined in §4.1, except the condition on line 7 of smt-formula-of is changed to ensure termination by stopping iteration if it comes to a semantic rule that is self-recursive on t'.

Example 5.4. Continuing Example 5.1, semantic-inlining inlines Sem^{x--}_S and $Sem^{x--;y++}_S$ to yield $Sem^{x--;\ y++}_S(x, y, x', y') \leftarrow \exists x'', y''. x'' = x - 1 \wedge y'' = y \wedge x' = x'' \wedge y' = y'' + 1$.

Quantifier Elimination. In the above example, we see that inlining definitions can leave superfluous quantifiers (e.g. $\exists x'', y''.$ in $Sem^{x--;\ y++}_S$). Eliminating these unnecessary quantifiers using simple quantifier-elimination methods can yield formulas that are easier for existing solvers to handle, which often exhibit performance that degrades exponentially in the number of quantifier alternations. Continuing the above example, in Muse, we apply Z3's quantifier-elimination tactic rule-by-rule to yield the quantifier-free formula $Sem^{x--;\ y++}_S(x, y, x', y') \leftarrow x' = x - 1 \wedge y' = y + 1$.

6 Experiments

We evaluated Muse with respect to the following research questions:

Q1: How effective is Muse at verifying solutions to SemGuS$^\mu$ problems?
Q2: How effective are the optimizations from §5 at improving Muse's performance?
Q3: Does Muse enable SemGuS synthesizers to handle problems with logical specifications?

All experiments were conducted on a desktop running Ubuntu 18.04 LTS, equipped with a 4-core Intel(R) Xeon(R) processor running at 3.2GHz with 12GB of memory. For each experiment we allotted a maximum of 6GB of memory; for each verification query we set a timeout of 5 minutes; and for each synthesis query we set a timeout of 30 minutes. We repeated each experiment three times and report the median result.

6.1 Benchmarks

We collected 141 SemGuS problems—whose semantics and logical specifications were expressed within linear integer arithmetic—from the official SemGuS benchmarks (https://github.com/

Table 1. All benchmarks, broken down by benchmark suite. For each solver, we list the number of benchmarks to which it can be applied, the number of benchmarks solved, and the average time per solved instance. The results given in the table are for the best configuration of each solver. The rows for "Partial Correctness" and "Total Correctness" summarize the results for all SemGuS benchmarks that require proving/refuting partial and total correctness, respectively, of imperative programs containing a loop (across all benchmark suites).

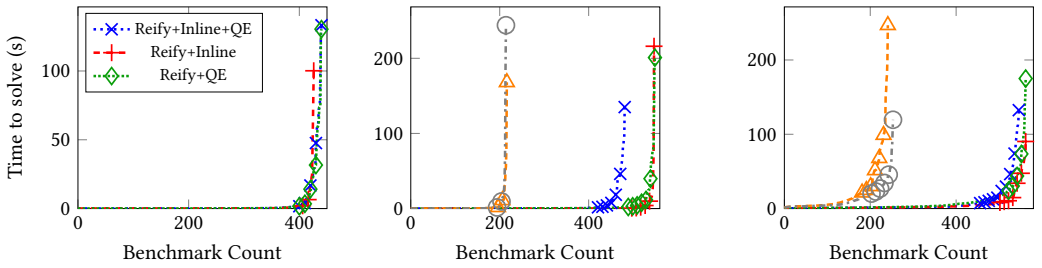
Suite	Total	SMT-FORMULA-OF			CHC-OF			MUCLP-OF		Best	
		isSMT	# Verified	Time	isCHC	# Verified	Time	# Verified	Time	# Verified	Time
SyGuS	160	160	104	0.42s	160	152	0.18s	111	15.89s	152	0.12s
SyGuS-Imp	54	27	27	0.07s	54	54	2.27s	28	1.32s	54	2.04s
FuncImp	38	12	12	0.07s	11	11	0.13s	35	4.26s	35	1.02s
Boolean	176	176	176	0.08s	176	176	0.09s	169	2.44s	176	0.08s
Regex	102	92	77	4.09s	102	96	4.61s	46	4.83s	97	3.28s
ScoreCards	60	60	40	0.10s	60	60	0.43s	51	3.69s	60	0.10s
Partial Correctness	70	27	27	0.07s	70	63	1.96s	40	2.20s	70	1.76s
Total Correctness	18	0	–	–	0	–	–	15	5.34s	15	5.34s
Controllers	80	0	–	–	0	–	–	75	5.50s	75	5.50s
PDDL	20	0	–	–	0	–	–	20	2.37s	20	2.37s
Games	40	0	–	–	0	–	–	40	0.66s	40	0.66s
Total	730	527	436	0.86s	563	549	1.15s	575	5.67s	709	1.40s

SemGuS-git/Semgus-Benchmarks)—which we augmented with quantified logical specifications. Because of the limitations of MuVal [39] (the solver we use for reasoning about μCLP queries), we restricted our focus to examples whose semantics can be expressed in LIA. We additionally created 80 SemGuS problems encoding SyGuS problems from SyGuS-comp (https://github.com/SyGuS-Org/benchmarks). We created an additional 74 SemGuS problems and 70 SemGuS$^{\mu}$ problems. Our suite of benchmarks consists of 295 functional-synthesis problems and 70 reactive-synthesis problems. It consists of SemGuS problems for various domains, including imperative programs, functional programs, score cards, SyGuS problems, Boolean formulas, regular expressions. Our SemGuS$^{\mu}$ problems consider relatively small (but infinite-state) reactive-synthesis problems, including reactive controllers, LTL formulas, reachability and Büchi games, and robotic path-planning problems. We split our problems into 9 suites (cf. Table 1). A complete description of the benchmarks can be found in Appendix B of the extended report [33], including a description of each suite of benchmarks, the sizes of benchmarks, and the process by which we generated each specification.

6.2 Experimental Setup

We evaluate Muse as follows: (*i*) we ablate each optimization described in §5 to evaluate Muse's performance on hand-crafted verification problems, and (*ii*) we incorporate Muse into an existing SemGuS synthesizer, Ks2 [25], to determine if Muse enables solving SemGuS problems with logical specifications. For both studies, we evaluate on the suite of SemGuS problems described in §6.1.

Ablation Study. For each SemGuS problem described in §6.1, we handcrafted two programs: one that satisfies the SemGuS problem, and one that does not. To generate the hand-crafted programs, we created the smallest program that satisfied the SemGuS problem. We then made minimal modifications to the satisfying program so that it no longer satisfied the logical specification. For each benchmark, we ran the SMT-FORMULA-OF, CHC-OF, and MUCLP-OF based solvers. We ran each solver in five configurations: (*i*) without any optimizations, (*ii*) with all three optimizations, (*iii*) with reification and predicate inlining, (*iv*) with reification and quantifier elimination, and (*v*) with predicate inlining and quantifier elimination. Figure 4 compares each configuration for each solver, and Table 1 details the result of the best configuration for each solver.



(a) SMT-FORMULA-OF (b) CHC-OF (c) MUCLP-OF

Fig. 4. Three cactus plots, one per encoding, with one line per optimization configuration used. If a point (x, y) appears along the line labeled by *config*, then *config* solved x instances in under y seconds. A line lower and further to the right is better.

SemGuS Synthesizer Study. Prior to Muse, most SemGuS synthesizers have limited their scope to solving problems whose specifications are represented as a finite set of input-output examples. The SemGuS toolkit [25] introduced Ks2, which was the first SemGuS synthesizer that was in theory capable of handling SemGuS problems with logical specifications. However, the baseline verifier provided with Ks2 relies on a naive SMT encoding that uses recursive axioms, a theory that is not well supported by SMT solvers. As such, one of our goals in developing a general-purpose verifier for SemGuS is to enable Ks2 (and other SemGuS synthesizers) to solve SemGuS problems with logical specifications. We incorporated Muse into Ks2 to replace the baseline solver to determine if Muse would allow Ks2 to solve more SemGuS problems with logical specifications.

Ks2 is a top-down enumeration-based SemGuS synthesizer that is available as part of the SemGuS toolkit (a collection of tools made available to enable development of SemGuS synthesizers) as a baseline synthesizer [25].[2] At a high-level, Ks2 operates by generating terms, checking each term against a set of input-output examples, then verifying if the term is correct with respect to the logical specification (if provided) and returns the first term that satisfies the specification. We modified Ks2 to use our verifier Muse instead of the baseline verifier when checking if enumerated terms meet a logical specification. We denote the augmented synthesizer as Ks2+Muse. For each verification problem encountered, we had Ks2+Muse invoke the simplest solver (i.e., SMT-FORMULA-OF for non-recursive solutions, CHC-OF for CHC-like problems, and MUCLP-OF for all others) with reification and predicate inlining (the configuration that performed the best in the ablation study).

We evaluated Ks2+Muse against baseline Ks2 on the 295 SemGuS problem described in §6.1. We excluded the reactive-synthesis problems because Ks2 does not support SemGuS$^{\mu}$ problems (i.e., problems whose semantics are not represented as CHCs). For each SemGuS problem, we provided 5 input-output examples alongside the logical specification to improve the search performed by Ks2.

Table 1 details the results of the Muse ablation study. Table 2 details the results of the synthesis study. While both Table 1 and Table 2 are based on the same set of benchmarks, *Table 1 and Table 2 consider a different number of experiments*—i.e., Table 1 considers 730 verification experiments (one satisfying and one falsifying program for each of the 295 SemGuS and 70 SemGuS$^{\mu}$ benchmarks), while Table 1 summarizes the results of running Ks2+Muse on each of the 295 SemGuS benchmarks.

6.3 Q1: Effectiveness of Muse

Table 1 presents the results of our experiments, summarized per benchmark suite, and summarizes how the three encodings supported by Muse compare. Each column labeled by a variant of Muse indicates the number of instances solved within the allotted time limit, together with the average solving time. The first columns of the SMT-FORMULA-OF and CHC-OF blocks also specify how many

[2] Ks2 is available as part of the SemGuS toolkit from the official SemGuS website: https://www.semgus.org.

instances fall within the fragment of first-order logic to which they can be applied. We note that the MUCLP-OF variant solved the most instances, solving 575 instances taking on average 5.6 seconds. The CHC-OF variant came in a close second solving 549 instances, averaging 1.1 seconds each. Upon further investigation, we found that CHC-OF solved 55 instances not solved by the other two variants, while MUCLP-OF solved 145 instances that were not and *could* not be solved by the other variants. Overall, 709 of the 730 benchmarks were solved by at least one of the three solvers. For the benchmarks that all solvers could handle, CHC-OF and SMT-FORMULA-OF performed similarly and were 12X faster than MUCLP-OF on average (geomean). For each problem for which a solver terminated for both the questions of verifying a correct solution and refuting an incorrect solution, proving that a solution was correct was in general slower than proving that a solution was incorrect (avg. 1.2X slower for CHC-OF, 2.8X slower for MUCLP-OF, 1.1X slower for SMT-FORMULA-OF).

MUSE solved 709/730 verification instances (in at least one variant). We inspected the instances where MUSE failed: across our 295 SemGuS problems (590 verification queries), 88 verification queries required reasoning about a program containing a loop, 70 of which required proving/refuting partial correctness and 18 of which required proving/refuting total correctness. Our benchmarks skew more towards partial correctness because the SyGuS-IMP benchmark includes 54 such verification problems that all involve partial-correctness specifications. While MUSE could solve all of the partial-correctness queries, it had a harder time proving/refuting total correctness, failing to refute one program and failing to prove totally correct 2/9 programs. Of the remaining 13 unsolved SemGuS verification queries (from the Regex and SemGuS-encoding-of-SyGuS problems), the verification query required proving correctness for programs whose semantics were relatively large (over 1000 AST nodes, some of which had over 500,000 AST nodes).

For SemGuS$^{\mu}$ verification queries, MUSE solved 135/140 of them. We note that due to the added complexity of encoding reactive synthesis problems within SemGuS, many of our reactive synthesis benchmarks consider relatively small (but infinite state) reactive synthesis problems, for which we expect existing techniques specialized for reactive synthesizers would excel.

To answer **Q1**, the verification techniques implemented in MUSE are effective and can verify 709/730 of our problem instances. We found that overall, proving correctness of a solution is generally harder than proving that a solution is incorrect. We note that in its current state MUSE is limited in how it can be incorporated into synthesis algorithms (e.g., those based on CEGIS [1]) because there is not yet a principled way to extract a meaningful counter-example when a solution fails to satisfy a SemGuS$^{\mu}$ problem. The root cause of this limitation is that MuVal does not provide a counter-model/proof when it refutes a formula.

6.4 Q2: Effectiveness of the Optimizations from Section 5

Figure 4 illustrates how the optimizations described in §5 affect the performance of each encoding. Each of the three graphs in Figure 4 shows the results of an ablation study—i.e., we compared the effectiveness of the optimizations by considering five configurations: no optimizations, all optimizations, and three configurations in which a single optimization was disabled. The three cactus plots show the performance for the three encodings in §4; the lines in each cactus plot show the performance of the five considered optimization configurations. One solver is better than another if its line is lower and to the right of the other solver's line (i.e., it can solve more problems in less time). For example, the SMT-FORMULA-OF variant performed best using reification and quantifier elimination. For all three encodings, the configurations using reification performed the best, solving on average 2.5X the number of verification problems solved without reification. In-lining and quantifier elimination do help somewhat; however, closer inspection of the results revealed that attempting quantifier elimination on the larger formulas generated during the execution of SMT-FORMULA-OF can lead to poor results.

Table 2. Results of running Ks2+Muse on SemGuS benchmarks.

	Suite	SyGuS	SyGuS-Imp	FuncImp	Boolean	Regex	ScoreCards	Total
Ks2+Muse	Solved	**10**	**25**	**10**	**38**	**33**	**19**	**135**
	Timed Out	1	2	10	5	10	1	29
	Memed Out	54	0	0	45	8	10	117
	Verification Calls	1210	367	53	38	108	19	1795
	Time (s)	476.46	3319.77	48.14	984.53	1240.85	58.19	6118.94
	Verification Time (s)	453.33	3074.16	24.02	16.60	197.16	8.62	3767.89
	Verif. Time / Call (s)	**0.37**	**8.38**	**0.45**	**0.43**	**1.82**	**0.45**	**2.10**
Ks2	Solved	3	0	4	35	0	**19**	61
	Timed Out	62	27	16	53	51	11	234
	Memed Out	0	0	0	0	0	0	0
	Verification Calls	81	–	19	35	–	19	154
	Time (s)	30.74	–	105.49	9749.43	–	250.29	10135.95
	Verification Time (s)	29.81	–	81.50	9554.98	–	190.17	9856.46
	Verif. Time / Call (s)	**0.37**	–	4.24	273.00	–	10.00	64.00

To answer **Q2**, the optimizations are very effective, with reification being the most effective.

6.5 Q3: Integration with an Enumeration-Based SemGuS Synthesizer

Table 2 summarizes the results of the synthesis study. We found that by incorporating Muse in Ks2, we enabled Ks2+Muse to solve 135 SemGuS problems with logical specifications—*74 of which could not be solved by Ks2 with the baseline verifier, nor any other previous SemGuS synthesizer*. For SemGuS problems that Ks2+Muse did solve, Ks2+Muse required on average 13.3 verification calls, taking on average 2.1 seconds each. Instead, Ks2 with the baseline verifier solved only 61 SemGuS problems, taking on average 2.5 verification calls, each taking on average 64 seconds. We found that neither Ks2+Muse nor baseline Ks2 solved many of the SemGuS-encoding-of-SyGuS problems; however, upon closer inspection we found that Ks2+Muse typically ran out of memory while enumerating terms whereas the baseline Ks2 timed out during a verification call. We note that all 65 of the SemGuS-encoding-of-SyGuS problems (when expressed as a SyGuS problem) could be solved by the CVC5 SyGuS solver [5] in under a second. As such, if one knows that a synthesis problem can be expressed in SyGuS, then using a SyGuS solver is obviously better.

For both synthesizers, we inspected the SemGuS problems that could not be solved within the time limit. Among the 146 SemGuS problems that Ks2+Muse could not solve, the 55 SemGuS-encoding-of-SyGuS problems failed to enumerate the correct solution within the time limit. Similarly, 1 FuncImp, 50 Boolean, 10 Regex, and 11 Scorecard problems failed to enumerate a correct solution within the time limit. The remaining 8 FuncImp, 2 SyGuS-Imp, and 8 Regex benchmarks timed out during verification after enumerating the correct solution. For the majority of the 234 SemGuS problems not solved by the baseline Ks2, Ks2 spent the majority of its time in the baseline verifier.

To answer **Q3**, by incorporating Muse into Ks2, we enabled Ks2 to synthesize 135 verifiably correct solutions, 74 of which could not be solved by any other SemGuS synthesizer. Of the 146 problems that remained unsolved, only 18 timed out due to verification (of which 11 could be solved by the virtual best solver, which runs each solver in parallel and returns once any terminates).

7 Related Work

Fixed-Semantics Program Verification. There is a large volume of work on automated verification for programs within a fixed language semantics. The typical approach often depends on the form of the language considered. For example, a popular verification methodology for imperative programs is to generate verification conditions automatically, and use invariant-generation techniques to satisfy those conditions [20, 23, 29, 32]. For functional languages, the typical approach uses type-based reasoning [34, 38, 41]. While there are a plethora of techniques for verification of imperative, declarative, and functional languages, these approaches (unlike Muse) do not support a user-defined semantics (e.g., via SemGuS$^{\mu}$), but can use domain-specific techniques to obtain better performance.

Verification Frameworks. More closely related to our work is the line of work on verification frameworks and intermediate verification languages. Stefănescu et al. [37] describe how to create an automated program verifier for arbitrary languages automatically, from an *operational* semantics written in the K framework. In a conversation with the K team, we confirmed that while matching logic—an expressive first-order fixed-point logic similar to μCLP—forms the basis of the K framework, the produced verifier is limited to answering reachability queries on inductively defined languages [30]. For example, while one can write a matching-logic formula that encodes the Büchi game in Figure 3, the K framework does not support defining a language whose semantics is a Büchi game (i.e., one cannot automatically generate a verifier for the language). Similarly, the verifier produced by K for an encoding of the IMP language in Figure 2 would not be able to verify that s_{loop} is totally correct (cf. §2.2). Finally, while the verifier automatically produced by the K framework cannot answer these verification problems, matching logic is general enough to express formulas with both least and greatest fixed-points. A reduction to matching logic might serve as the basis for yet another verifier for SemGuS.

In a similar vein is work on creating and using an intermediate language for verification, such as Boogie [7] or Why3 [17]. A key difference between those tools and Muse is that with Boogie and Why3, a language's semantics is specified via a *translational semantics*, whereas with Muse a language's semantics is specified *declaratively*. With a translational semantics, one has to define a function that walks over the abstract-syntax tree t of a program, and constructs an appropriate Boogie/Why3 program whose meaning captures the semantics of t. In contrast, with Muse, the semantics is specified declaratively, using logical relations in SemGuS$^{\mu}$, thus allowing one to model many diverse scenarios (e.g., our robot example). Many systems have used Boogie as their intermediate language, including Dafny [29] and VCC [14]. Other similar systems include Cameleer [34] (on top of Why3) and various C analyzers built on top of the FRAMA-C platform [27]. While intermediate verification languages allow the reuse of verifiers for multiple languages, they generally support a single *language paradigm* (e.g, object-oriented, functional, etc.) and a single verification strategy (e.g., pre/post conditions and loop invariants) and thus may be difficult to use for a language based on a different paradigm. In contrast, the SemGuS$^{\mu}$ framework uses a logic-based approach to specifying semantics, which allows Muse to be applied to a wide variety of problems.

Logic-Based Verification. There is also a substantial body of work that uses fragments of first-order logic (with fixed-points) to verify programs. A broad class of work considers programs represented as transition formulas [2, 4, 6, 16]. That is, the verification task takes as input a formula modeling the transitions of the program. While these techniques and ours all take as input a logical formalism describing the program of interest, transition formula are monolithic formulas defined on a program-by-program basis, and differ from the modular semantic relations—supplied on a per-language basis via SemGuS$^{\mu}$—used in Muse.

Our work on Muse also has connections to verification based on answer-set programming and stratified logic programming [12, 13, 18]. Specifically, techniques based on answer-set programming and stratified logic programming automatically compile a program and specification into a logic program. Our technique could be viewed similarly, specifically because μCLP can be seen as a generalization of logic programs (with first-order constraints and fixed-point operators). While the SemGuS$^{\mu}$ framework expects a user to provide a declarative specification of a language's semantics, typical approaches based on answer-set programming or stratified logic programming expect the user to provide an encoding of both the specification and the program's semantics on a program-by-program basis.

Another line of work translates verification queries into validity (or satisfiability) queries in fragments of first-order logic [22, 40]. SeaHorn [22] compiles annotated C programs into a system

of CHCs to discharge the generated verification conditions. The work of Unno et al. [40] formulates a number of program-verification tasks in the pfwCSP fragment of first-order logic (a constraint language similar in expressiveness to μCLP). These techniques are similar to the ones used in Muse in that they answer verification queries by generating a logical query and using a solver to answer the query; however, the methods in these other tools are defined for a fixed language (e.g., a fragment of C), whereas Muse is parameterized by the language specified in the SemGuS$^{\mu}$ input.

SemGuS Synthesizers. As discussed in §6, we incorporated our SemGuS verifier, Muse, within an existing SemGuS synthesizer Ks2, which enabled Ks2+Muse to solve 135 SemGuS problems with logical specifications—74 of which could not be solved by any previous SemGuS synthesizer. In addition to describing Ks2, Johnson et al. [25] also present a format for describing SemGuS problems, which our tool Muse accepts. Besides Ks2, the only other existing SemGuS synthesizer is Messy, which only handles specifications in the form of input-outuput examples [26]. While Messy can prove if a SemGuS problem is realizable/unrealizable, when Messy proves that a SemGuS problem is realizable, it is unable to produce a concrete program.[3] For this reason, we could not combine our verifier Muse with Messy to enable it to solve SemGuS problems with logical specifications.

While Ks2 is the first SemGuS synthesizer capable of solving SemGuS problems with logical specifications, there are other domain-specific synthesizers that can (quickly) solve some of the synthesis problems used in our benchmarks when represented within the constraints of their domains. For example, CVC5 [5] can quickly solve all of the SyGuS problems that we encoded within the SemGuS framework to evaluate Ks2+Muse. However, to solve a SemGuS problem using a SyGuS solver, we would first need to recognize that the SemGuS problem encodes a SyGuS problem—i.e., by proving that the syntax and semantics of the SemGuS problem can be reformulated automatically as a SyGuS problem. To summarize, to support a general framework, we must develop general SemGuS solvers.

Reactive Synthesis. Our work on SemGuS$^{\mu}$ has ties to reactive synthesis [3, 11, 36]—i.e., SemGuS$^{\mu}$ is flexible enough to encode both functional and reactive-synthesis problems (e.g., Büchi and reachability games, controllers, and imperative programs that satisfy a temporal property). However, although SemGuS$^{\mu}$ is general enough to encode such problems, and thus offers a way to reason uniformly about functional and reactive-synthesis problems, we do not expect solvers based on Muse to be competitive compared to specialized reactive synthesizers (e.g., [11, 31, 36]).

8 Conclusion

The SemGuS framework [26] is becoming a standard for program synthesis, as happened with the less expressive SyGuS framework. This paper presents the methodology for verifying that a candidate solution is valid for a SemGuS problem. Our technique reduces verification questions to validity questions in μCLP (a fragment of first-order fixed-point logic), or validity questions in easier logics, when possible. Our work fills an important gap in the pipeline of techniques needed to build practical SemGuS synthesizers. One can now build solvers that synthesize solutions to SemGuS problems involving complex specifications and verify whether these solutions are correct! While our tools currently handle relatively small programs, improvement to our framework will lead to improvements in any SemGuS solver.

Acknowledgments

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

[3]This limitation is detailed in the official documentation of Messy: https://github.com/SemGuS-git/Semgus-Messy.

References

[1] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. 2018. Counterexample guided inductive synthesis modulo theories. In *International Conference on Computer Aided Verification*. Springer, 270–288. https://doi.org/10.1007/978-3-319-96145-3_15

[2] Marco Alberti, Davide Daolio, Paolo Torroni, Marco Gavanelli, Evelina Lamma, and Paola Mello. 2004. Specification and verification of agent interaction protocols in a logic-based system. In *Proceedings of the 2004 ACM symposium on Applied computing*. 72–78. https://doi.org/0.1145/967900.967918

[3] Rajeev Alur, Salar Moarref, and Ufuk Topcu. 2018. Compositional and symbolic synthesis of reactive controllers for multi-agent systems. *Information and Computation* 261 (2018), 616–633. https://doi.org/10.1016/j.ic.2018.02.021

[4] André Arnold. 1993. Verification and comparison of transition systems. In *Colloquium on Trees in Algebra and Programming*. Springer, 121–135. https://doi.org/10.1007/3-540-56610-4_60

[5] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

[6] Luciano Baresi, Angelo Morzenti, Alfredo Motta, and Matteo Rossi. 2012. A logic-based semantics for the verification of multi-diagram UML models. *ACM SIGSOFT Software Engineering Notes* 37, 4 (2012), 1–8. https://doi.org/10.1145/2237796.2237811

[7] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. 2006. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4*. Springer, 364–387. https://doi.org/10.1007/11804192_17

[8] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. Horn clause solvers for program verification. In *Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. Springer, 24–51. https://doi.org/doi.org/10.1007/978-3-319-23534-9_2

[9] Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. 2013. On solving universally quantified horn clauses. In *Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings 20*. Springer, 105–125. https://doi.org/10.1007/978-3-642-38856-9_8

[10] Nikolaj S Bjørner and Mikolás Janota. 2015. Playing with Quantified Satisfaction. *LPAR (short papers)* 35 (2015), 15–27. https://doi.org/10.29007/vv21

[11] Roderick Bloem, Krishnendu Chatterjee, and Barbara Jobstmann. 2018. Graph games and reactive synthesis. *Handbook of model checking* (2018), 921–962. https://doi.org/10.1007/978-3-319-10575-8_27

[12] Martin Bromberger, Irina Dragoste, Rasha Faqeh, Christof Fetzer, Markus Krötzsch, and Christoph Weidenbach. 2021. A datalog hammer for supervisor verification conditions modulo simple linear arithmetic. In *International Symposium on Frontiers of Combining Systems*. Springer, 3–24. https://doi.org/10.1007/978-3-030-86205-3_1

[13] Diego Calvanese, Giuseppe De Giacomo, Marco Montali, and Fabio Patrizi. 2013. Verification and synthesis in description logic based dynamic systems. In *International Conference on Web Reasoning and Rule Systems*. Springer, 50–64. https://doi.org/10.1007/978-3-642-39666-3_5

[14] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*. Springer, 23–42. https://doi.org/10.1007/978-3-642-03359-9_2

[15] Loris D'Antoni, Qinheping Hu, Jinwoo Kim, and Thomas Reps. 2021. Programmable program synthesis. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33*. Springer, 84–109. https://doi.org/10.1007/978-3-030-81685-8_4

[16] Azadeh Farzan and Zachary Kincaid. 2017. Strategy synthesis for linear arithmetic games. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30. https://doi.org/10.1145/3158149

[17] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3—where programs meet provers. In *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*. Springer, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

[18] Frank Flederer, Ludwig Ostermayer, Dietmar Seipel, and Sergio Montenegro. 2017. Source code verification for embedded systems using prolog. *arXiv preprint arXiv:1701.00630* (2017). https://doi.org/10.4204/EPTCS.234.7

[19] Sumit Gulwani. 2012. Synthesis from examples. In *WAMBSE (Workshop on Advances in Model-Based Software Engineering) Special Issue, Infosys Labs Briefings*, Vol. 10. Citeseer. https://doi.org/10.1109/SYNASC.2012.69

[20] Ashutosh Gupta and Andrey Rybalchenko. 2009. Invgen: An efficient invariant generator. In *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26-July 2, 2009. Proceedings 21*. Springer, 634–640.

https://doi.org/10.1007/978-3-642-02658-4_48

[21] Arie Gurfinkel and Nikolaj Bjørner. 2019. The science, art, and magic of constrained horn clauses. In *2019 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 6–10. https://doi.org/10.1109/SYNASC49474.2019.00010

[22] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn verification framework. In *International Conference on Computer Aided Verification*. Springer, 343–361. https://doi.org/10.1007/978-3-319-21690-4_20

[23] Thomas A Henzinger, Thibaud Hottelier, and Laura Kovács. 2008. Valigator: A verification tool with bound and invariant generation. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 333–342. https://doi.org/10.1007/978-3-540-89439-1_24

[24] Stefan Hetzl and Johannes Kloibhofer. 2021. A fixed-point theorem for Horn formula equations. *arXiv preprint arXiv:2109.04633* (2021). https://doi.org/10.4204/EPTCS.344.5

[25] Keith JC Johnson, Andrew Reynolds, Thomas Reps, and Loris D'Antoni. 2024. The SemGuS Toolkit. In *International Conference on Computer Aided Verification*. Springer, 27–40. https://doi.org/10.1007/978-3-031-65633-0_2

[26] Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. 2021. Semantics-guided synthesis. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–32. https://doi.org/10.1145/3434311

[27] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal aspects of computing* 27 (2015), 573–609. https://doi.org/10.1007/s00165-014-0326-7

[28] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M Clarke. 2013. Automatic abstraction in SMT-based unbounded software model checking. In *International Conference on Computer Aided Verification*. Springer, 846–862. https://doi.org/10.1007/978-3-642-39799-8_59

[29] K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*. Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[30] Zhengyao Lin, Xiaohong Chen, Minh-Thai Trinh, John Wang, and Grigore Roşu. 2023. Generating Proof Certificates for a Language-Agnostic Deductive Program Verifier. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 56–84. https://doi.org/10.1145/3586029

[31] Manuel Mazo Jr, Anna Davitian, and Paulo Tabuada. 2010. Pessoa: A tool for embedded controller synthesis. In *International conference on computer aided verification*. Springer, 566–569. https://doi.org/10.1007/978-3-642-14295-6_49

[32] Kenneth L McMillan. 2006. Lazy abstraction with interpolants. In *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 18*. Springer, 123–136. https://doi.org/10.1007/11817963_14

[33] Charlie Murphy, Keith Johnson, Thomas Reps, and Loris D'Antoni. 2024. Verifying Solutions to Semantics-Guided Synthesis Problems. https://doi.org/10.48550/arXiv.2408.15475 arXiv:2408.15475 [cs.PL]

[34] Mário Pereira and António Ravara. 2021. Cameleer: A Deductive Verification Tool for OCaml. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33*. Springer, 677–689. https://doi.org/10.1007/978-3-030-81688-9_31

[35] Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 107–126. https://doi.org/10.1145/2858965.2814310

[36] Vasumathi Raman, Alexandre Donzé, Dorsa Sadigh, Richard M Murray, and Sanjit A Seshia. 2015. Reactive synthesis from signal temporal logic specifications. In *Proceedings of the 18th international conference on hybrid systems: Computation and control*. 239–248. https://doi.org/10.1145/2728606.2728628

[37] Andrei Stefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. 2016. Semantics-based program verifiers for all languages. *ACM SIGPLAN Notices* 51, 10 (2016), 74–91. https://doi.org/10.1145/3022671.2984027

[38] Hiroshi Unno, Yuki Satake, and Tachio Terauchi. 2018. Relatively complete refinement type system for verification of higher-order non-deterministic programs. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 1–29. https://doi.org/10.1145/3158100

[39] Hiroshi Unno, Tachio Terauchi, Yu Gu, and Eric Koskinen. 2023. Modular Primal-Dual Fixpoint Logic Solving for Temporal Verification. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 2111–2140. https://doi.org/10.1145/3571265

[40] Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. 2021. Constraint-based relational verification. In *International Conference on Computer Aided Verification*. Springer, 742–766. https://doi.org/10.1007/978-3-030-81685-8_35

[41] Hongwei Xi. 2002. Dependent types for program termination verification. *Higher-Order and Symbolic Computation* 15 (2002), 91–131. https://doi.org/10.1109/LICS.2001.932500

[42] Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Murphy Berzish, Julian Dolby, and Xiangyu Zhang. 2017. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods in System*

Design 50 (2017), 249–288. https://doi.org/10.1007/s10703-016-0263-6

Received 2024-11-14; accepted 2025-03-06