Verifying Solutions to Semantics-Guided Synthesis Problems

ANONYMOUS AUTHOR(S)

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 problem of how to verify programs in a solver and domain-agnostic way.

We prove that the problem of verifying whether a program is a valid solution to a SemGuS problem can be reduced to proving validity of a query in the $\mu$CLP calculus, a fixed-point logic that generalizes Constrained Horn Clauses and co-Constrained Horn Clauses. Our encoding into $\mu$CLP allows us to further classify the SemGuS verification problems into ones that are reducible to validity of (i) first-order-logic formulas, (ii) Constrained Horn Clauses, (iii) co-Constrained Horn Clauses, and (iv) $\mu$CLP 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 $\mu$CLP 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.

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 [Polozov and Gulwani 2015] and bit-vector manipulations [Gulwani 2012], 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 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. [2021] 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 can use Constrained Horn Clauses (a least-fixed-point logic) to define the semantics of the programming language over which one is interested in performing synthesis. (A detailed example of SemGuS problem is given in Figure 1 and discussed in Section 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 [D’Antoni et al. 2021].

Solving a synthesis problem requires, at the very least, to be able to verify whether a synthesized program satisfies the desired specification. Because of the added complexity introduced by Constrained Horn Clauses, Kim et al. [2021] have so far only proposed ways to verify programs in cases where the specification is given through a finite set of examples—i.e., there is currently no way to verify the solution to a SemGuS problem for a general specification—e.g., those involving quantified variables. 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. The problem of 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.

2024. XXXX-XXXX/2024/5-ART $15.00
https://doi.org/10.1145/nnnnnn.nnnnnnnn

, Vol. 1, No. 1, Article . Publication date: May 2024.
The key challenge in verifying solutions to SemGuS problem lies in the fact that the semantics of the programming language for which we are performing verification is an input parameter, given as a set of Constrained Horn Clauses (CHCs)—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 [Gupta and Rybalchenko 2009; Henzinger et al. 2008; Leino 2010; Pereira and Ravara 2021; Zheng et al. 2017] 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 [Leino 2010] and type-based reasoning for functional programs [Pereira and Ravara 2021].

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 \( \text{Sem} \), and a specification \( \varphi \) (which is allowed to mention the semantic relations defined by \( \text{Sem} \)), we show that the problem of verifying whether \( p \)—when evaluated according to \( \text{Sem} \)—satisfies \( \varphi \) can be expressed as a validity check in the \( \mu \text{CLP} \) calculus. \( \mu \text{CLP} \) 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., via CHCs), the fact that the semantic relations can appear in both positive and negative positions in a user-supplied specification makes a least-fixed-point calculus not expressive enough to verify whether a program meets the specification, resulting in the need for a more expressive calculus, such as \( \mu \text{CLP} \). Because of the complexity of building solvers for checking validity of \( \mu \text{CLP} \) 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, CHCs, and co-CHCs, for which more scalable solvers exist.

Our study highlights a strong connection between SemGuS and \( \mu \text{CLP} \), and raises the question of whether there exist programming languages for which verification is expressible using \( \mu \text{CLP} \), but for which SemGuS cannot define the semantics. 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 [Alur et al. 2018]. To close the loop between \( \mu \text{CLP} \) and SemGuS, we define a minimal extension of SemGuS—i.e., we allow relations to appear in a negated form in the semantic definitions—which results in a new framework that aligns exactly with what is verifiable using \( \mu \text{CLP} \). 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 (Section 2).
- We propose an extension of the SemGuS framework that can capture, e.g., reactive synthesis problems (Section 3).
- We analyze when solutions to SemGuS problems can be verified using various logical fragments (SMT, CHC, co-CHC, and \( \mu \text{CLP} \)) and show that our extension of SemGuS aligns exactly with what is verifiable using \( \mu \text{CLP} \) (Section 4).
We implemented our approach in a tool, Muse, together with several optimizations (Section 5), and used Muse to verify (or disprove correctness for) solutions to SemGuS problems that could not be solved by any prior approach (Section 6).

We incorporated Muse within the SemGuS synthesizer Ks2 to enable Ks2 to solve SemGuS problems that cannot be solved by any previous approach (Section 6). In particular, Ks2 + Muse is the first SemGuS synthesizer that is able to solve SemGuS problems that involve a general specification—i.e., involving quantified variables.

Section 7 discusses related work. Section 8 concludes.

2 OVERVIEW

This section illustrates our approaches for verifying a solution to a SemGuS problem using four 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 \(G_{\text{max2}}\) defining the syntax of the language under consideration (Figure 1a).
- A set of constrained Horn clauses \(Sem_{\text{max2}}\) that inductively define the semantics (as a least fixed point) of all programs in the language (Figure 1d).
- A specification \(\varphi_{\text{max2}}\) that describes how the synthesized program should behave when evaluated according to \(Sem_{\text{max2}}\) (Figure 1b).

The specification \(\varphi_{\text{max2}}\) 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.

Beyond least fixed points. Because the semantics is already defined using a least-fixed-point logic, namely CHCs, it is natural to be able to solve the problem in terms of validity of CHCs. That is, at first blush, it seems plausible to check the validity query in Equation (1), which states that \(\varphi_{\text{max2}}\) is valid when interpreted using the least solution of the semantic relations.

\[
Sem_{\text{LFP}}_{\text{max2}} \models \forall x, y, x'. (\exists y'. Sem_{\varphi_{\text{max2}}} (max2, x, y, x', y') \iff (x' = x \lor x' = y) \land x \leq x' \land y \leq x')
\]  

While the semantic relation \(Sem_{\text{max2}}\) is defined via a least fixed point over a set of constrained Horn clauses, the positive occurrence of \(Sem_{\varphi_{\text{max2}}}\) results in a query that cannot be reasoned about within a least-fixed-point logic (namely CHCs). Note that the typical approach to prove the validity of a query of the form \(Sem_{\text{LFP}}_{\text{max2}} \models \psi\) would be to check that the formula \(Sem_{\text{LFP}}_{\text{max2}} \land \neg \psi\) is unsatisfiable, which requires every occurrence of a semantic relation in \(\psi\) to be positive. Otherwise, the query falls outside of the CHC fragment of first-order logic.

Finite derivation trees can be desugared. Our first insight is that for problems like \(\text{max2}\), where the semantic definitions are recursively defined with respect to the term’s proper subterms, one can
Semantics exactly characterizes \(\varphi\) in the antecedent. Through this process, we can extract the following formula:

\[
\exists r_b. (\exists r_s, r_t, r_s = x \wedge r_t = y \wedge r_b \Leftrightarrow r_s < r_t) \wedge ((r_b \wedge x' = y \wedge y' = y) \vee (\neg r_b \wedge x' = x \wedge y' = y))
\]

always build a finite derivation tree that describes the semantics of a given program. For example, the derivation tree for \(s_{\text{max2}}\) is as follows:

\[
\begin{align*}
S & := x = E \mid y = E \\
| S \; S | & \text{Ite } B \; S \\
E & := 0 \mid 1 \mid x \mid y \mid E + E \\
B & := E < E \\
\end{align*}
\]

Because the tree is finite, the relation \(\text{Sem}_{s_{\text{max2}}}\) can be defined equivalently with a logic that does not require fixed points. 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. Through this process, we can extract the following formula \(\varphi_{\text{Sem}_{s_{\text{max2}}}}\), which exactly characterizes \(\text{Sem}_{s_{\text{max2}}}\):

\[
\exists r_b. (\exists r_s, r_t, r_s = x \wedge r_t = y \wedge r_b \Leftrightarrow r_s < r_t) \wedge ((r_b \wedge x' = y \wedge y' = y) \vee (\neg r_b \wedge x' = x \wedge y' = y))
\]
We can now replace the term $\text{Sem}_S(s_{\text{max}}, x, y, x', y)$ in the formula in Equation (1) with the freshly computed term $\varphi_{\text{Sem}_{\text{max}}}^S$, and obtain the following formula, which is entirely describable in first-order logic without requiring any fixed-point reasoning.

\[ \forall x, y, x'. (\exists y'. y = y' \land ((x < y \land x' = y) \lor (x \geq y \land x' = x)) \Leftrightarrow (x' = x \land x' = y) \land x \leq x' \land y \leq x' \]

The resulting formula is logically equivalent to Equation (1) and is thus valid if and only if the candidate program satisfies the synthesis problem. In our tool Muse, the quantified satisfiability modulo theories (SMT) solver Z3 [Bjørner and Janota 2015] proves this formula valid in 0.06 seconds, thereby proving that $s_{\text{max}}$ is a correct solution to this SemGuS problem. Given the incorrect candidate program $t_\text{e} (y < x) (x = y) (x = x)$, our tool Muse performs the same process and proves in 0.06 seconds that this program is not a solution to this SemGuS problem.

### 2.2 DoubleViaLoop Partial: CHCs

The technique presented in Section 2.1 relies on the fact that for every program in the language, one can build a finite derivation tree that describes its semantics. Our second example considers a more complex synthesis task where such a property does not hold (Figure 2).

The task is to synthesize an imperative program (this time potentially containing a loop) given initial values $x$ and $y$ for the variables $x$ and $y$, respectively; if the program terminates, it must set the value $y'$ of variable $y$ to 2x. The grammar $G_{\text{loop}}$ 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 $\text{Sem}_{\text{loop}}$ 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. The specification $\varphi_{\text{loop}}$ requires that any correct solution to the synthesis problem must be partially correct: if the input value $x$ is non-negative and the solution (symbolically represented as $f_{\text{loop}}$) terminates with the output $y'$, then $y'$ is twice $x$ (Figure 1b).

Similar to the previous example, verifying whether the program $s_{\text{loop}}$ given in Figure 2b is correct requires proving that the query $Q_{\text{loop}} \triangleq \text{Sem}_{\text{loop}}^\text{LFP} \models \varphi_{\text{loop}}[f_{\text{loop}} \mapsto s_{\text{loop}}]$ is valid—i.e., that the candidate solution $s_{\text{loop}}$ satisfies the specification $\varphi_{\text{loop}}$ when interpreted using the least solution of the semantic rules $\text{Sem}_{\text{loop}}$. Unlike the previous example, the specification $\varphi_{\text{loop}}$ (Figure 1b) contains only negative occurrences of the semantic relations (because it does not require the program to terminate), thus enabling the use of a least-fixed-point logic (namely CHCs) to reason about the query. In Muse, we use the CHC solver Spacer [Komuravelli et al. 2013] to solve $Q_{\text{loop}}$ in 0.2 seconds, thereby proving that $s_{\text{loop}}$ is a valid solution.

### 2.3 DoubleViaLoop Total: CHCs and co-CHCs

In Section 2.2, we were able to use a CHC solver to reason about the query $Q_{\text{loop}}$ because the specification $\varphi_{\text{loop}}$ did not contain positive occurrences of the semantic relations in $\text{Sem}_{\text{loop}}$. In the next example, we consider the same grammar $G_{\text{loop}}$ and semantics $\text{Sem}_{\text{loop}}$ as in Figure 2, but introduce a modified specification $\varphi_{\text{loop}}^{\text{tot}}$ that requires a form of total correctness:

\[ \varphi_{\text{loop}}^{\text{tot}} = \forall x, y'. (0 \leq x \land 2x = y') \Rightarrow \text{Sem}_{\text{L}}(f_{\text{loop}}, x, 0, 0, y') \]  \hspace{1cm} (4)

The above specification states that when the program to be synthesised starts in a state where variable $x$ takes a non-negative value $x$ and variable $y$ is 0, then it will terminate in a state where $x$ is 0 and $y$ is twice $x$'s initial value $x$. However, the resulting query $Q_{\text{loop}}^{\text{tot}} = \text{Sem}_{\text{loop}}^\text{LFP} \models \varphi_{\text{loop}}^{\text{tot}}[f_{\text{loop}} \mapsto s_{\text{loop}}]$ cannot be solved using a least-fixed-point logic because the specification has a positive occurrence of the semantic relation $\text{Sem}_{\text{L}}$. Instead, we show that one can construct a logically equivalent query...
where $\neg Sem_{\text{loop}}^L(s_{\text{loop}}, x, 0, 0, y')$ is the greatest fixed point of the coCHCs that define the dual semantics of $Sem_{\text{loop}}$. Intuitively, the query $\tilde{Q}_{\text{loop}}^\text{tot}$ asks if there is a positive value for $x$ for which the candidate program does not compute $2x$—either because the candidate program does not terminate on the input or because it terminates in a state where $y$ is not $2x$.

MUSE takes as input $s_{\text{loop}}$, $Sem_{\text{loop}}$, and $\phi_{\text{loop}}^\text{tot}$ and produces the query $\tilde{Q}_{\text{loop}}^\text{tot}$. The process dualizes every semantic relation and each semantic rule. For example, our encoding produces the following rule to define $Sem_{\text{loop}}^\neg(s; t, x, y, x', y')$.

\[
Sem_{\text{loop}}^\neg(s; t, x, y, x', y') \Rightarrow \forall x', y''. Sem_{\text{loop}}(s, x, y, x', y'') \lor Sem_{\text{loop}}^\neg(t, x'', y'', x', y')
\]

After the encoding, the produced query $\tilde{Q}_{\text{loop}}^\text{tot}$ is logically equivalent to the original query $Q_{\text{loop}}^\text{tot}$ and formulated entirely within a fragment of first-order logic that can be solved using only greatest fixed points (namely coCHCs). Thus, we can use a coCHC solver to prove the validity of this query to determine that $s_{\text{loop}}$ is correct with respect to the specification $\phi_{\text{loop}}^\text{tot}$. Due to the lack of coCHC solvers, MUSE uses the $\mu$CLP solver MuVal [Unno et al. 2023] to solve this instance in 1.6 seconds.
**CHC and co-CHC.** The solutions to the two examples discussed above can be verified using either CHCs or co-CHCs alone because the semantic relations appear in the specifications either only positively (in which case we use the dual semantics) or only negatively (in which case we use the original semantics).

Expressing total correctness requires specifications in which the semantic relations appear both positively and negatively, as in the following example:

\[ \varphi_{loop}^{both} = \forall x, y'. 0 \leq x \Rightarrow (\text{Sem}_L(x, 0, 0, y') \Leftrightarrow 2x = y') \]

(6)

Although this specification does not allow one to directly use any of the techniques we presented, in this case, the specification can be split into two separate specifications in which the semantic relations appear only positively in one, and only negatively in the other. In fact, this split results in the two specifications \( \varphi_{loop} \) and \( \varphi_{loop}^{tot} \). To verify that a candidate program satisfies the specification in \( \varphi_{loop}^{both} \), it is sufficient to check that the candidate program satisfies both \( \varphi_{loop} \) and \( \varphi_{loop}^{tot} \) by checking validity of the queries \( Q_{loop} \) and \( \tilde{Q}_{loop}^{tot} \).

### 2.4 Hyperproperties: \( \mu \)-CLP

Section 2.3 presented a technique for verifying a solution for cases when the specification can be split into finitely many formulas in which the semantic relations appear only positively or only negatively. However, the splitting approach is not always possible!

For example, consider the following specification that requires the synthesized function to be commutative in its arguments. (Such properties are sometimes called hyperproperties because their falsification requires one to consider two different executions of the program, starting from different input states.)

\[ \varphi_{comm} \triangleq \forall x, y, x', y'. \text{Sem}_L(s, x, y, x', y') \Rightarrow \text{Sem}_L(s, y, x, x', y') \]

(7)

The specification \( \varphi_{comm} \) could arise when trying to synthesize a program like

\[ s_{plus} = \text{while } 0 < x \text{ do } x--; y++ \]

(8)

which, when it terminates, sets the value of variable \( y \) to the sum of the inputs \( x \) and \( y \). This program is in the language defined by the grammar \( G_{loop} \) in Figure 2a, and we assume it operates over the semantics \( \text{Sem}_{loop} \) in Figure 2d.

To prove that the program \( s_{plus} \) satisfies the specification \( \varphi_{comm} \), one must reason simultaneously about the relation \( \text{Sem}_L(\text{sum}, x, y, x', y') \) and its complement \( \text{Sem}_L(\text{sum}, y, x, x', y') \). Even if we define the dual semantics of the language, we still need to reason about both such relations simultaneously. We show that the problem of verifying whether a program meets a specification like \( \varphi_{comm} \) — and in fact every specification expressible in \( \text{SemGuS} \) — can be reduced to checking validity in the \( \mu \)-CLP calculus, a logic that combines least- and greatest-fixed-point reasoning [Unno et al. 2023].

Muse reduces this verification problem to the following \( \mu \)-CLP query that combines both the positive semantics \( \text{Sem}_{loop} \) and negative semantics \( \text{Sem}_L^{\neg CLP} \):

\[ Q_{plus} \triangleq \text{Sem}_L^{\text{CLP}} \land \text{Sem}_L^{\neg \text{CLP}} \models \forall x, y, x', y'. \text{Sem}_L(s_{plus}, x, y, x', y') \lor \text{Sem}_L(s_{plus}, y, x, x', y'). \]

(9)

which follows from Equation (7) by (i) instantiating \( s \) as \( s_{plus} \), (ii) replacing “\( \text{Sem}_L(s_{plus}, x, y, x', y') \Rightarrow \ldots \)” with “\( \neg \text{Sem}_L(s_{plus}, x, y, x', y') \lor \ldots \)” and (iii) replacing “\( \neg \text{Sem}_L(s_{plus}, x, y, x', y') \)” with “\( \text{Sem}_L(s_{plus}, x, y, x', y') \)”.

In Muse, we use the \( \mu \)-CLP solver MuVal [Unno et al. 2023] to solve \( Q_{comm} \) in 6.2 seconds, thereby proving that \( s_{plus} \) is commutative.
We call the semantic relation $\text{Buchi}^\neg$ winning the game in Figure 3, we generate the following verification query:

\begin{align*}
S := & \text{repeat } S \mid \text{stay} \mid L \mid R \\
& \mid L; S \mid R; S \\
\text{repeat } & R; R; R; R; R; \\
& L; L; L; L; L \\
\neg & \text{Buchi}^\neg(\text{strat}, 0, 0)
\end{align*}

(a) Robot Strategy \hspace{1cm} (b) Solution strat \hspace{1cm} (c) Specification

\begin{align*}
\text{Buchi}^\neg(\text{strat}, x, y) & \leftarrow x \neq y \lor (\exists y'. 0 \leq y' \leq 5 \land \neg \text{Reach}(\text{strat}, x, y')) \\
\text{Reach}(\text{strat}, x, y) & \leftarrow \neg \text{Buchi}^\neg(\text{strat}, x, y) \lor (\exists x', \text{strat}'. \text{Move}(\text{strat}, x, x', \text{strat}') \land \text{reach}(\text{strat}', x', y)) \\
\text{Move}(\text{repeat } s, x, x', \text{strat}') & \leftarrow \text{Move}(s; \text{repeat } s, x, x', \text{strat}') \\
\text{Move}(L; s, x, x', \text{strat}') & \leftarrow \text{strat}' = s \land x' = x - 1 \\
\text{Move}(R; s, x, x', \text{strat}') & \leftarrow \text{strat}' s \land x' = x + 1
\end{align*}

(d) Semantics $\text{Sem}_{\text{Buchi}}$

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.

2.5 Beyond SemGuS

Section 2.4 showed that for every SemGuS problem, one can verify the correctness of a candidate solution using a $\mu$CLP solver. This connection raises a natural question in the opposite direction: Are there programming languages for which verification is expressible using $\mu$CLP, but whose semantics cannot be expressed using the SemGuS framework? 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 4.8, SemGuS$^\mu$ captures exactly every programming language for which solutions can be verified using $\mu$CLP.

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:

$$\text{Sem}_{\text{Buchi}}^{\text{FP}} \models \neg \text{Buchi}^\neg(\text{strat}, 0, 0) \tag{10}$$

We call the semantic relation $\text{Buchi}^\neg$ 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 ($\text{Sem}_{\text{Buchi}}^{\text{FP}}$) defines the rules of the game. Intuitively, Reach encodes that the robot must eventually satisfy the Büchi condition (i.e., denoted by $\neg \text{Buchi}^\neg(\text{strat}, x, y)$). The
Büchi condition is represented implicitly as the negation of its dual \( \text{Buchi}^\sim \). 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.

To solve the verification query in Equation (10), we must compute the fixed point of \( \text{Sem}_{\text{Buchi}} \). Unlike the previous examples, the semantics \( \text{Sem}_{\text{Buchi}} \) is not defined using CHCs—most notably due to the negative occurrences of \( \text{Buchi}^\sim \) and \( \text{Reach} \) within the premise of the semantic rules. In fact, \( \text{Sem}_{\text{Buchi}} \) does not even define a least fixed-point. Because \( \text{Reach} \) occurs negatively in the premise of \( \text{Buchi}^\sim \)'s definition, the least fixed-point of \( \text{Buchi}^\sim \) requires computing the greatest fixed-point of \( \text{Reach} \) (the dual of \( \text{Reach} \)). Similarly, because \( \text{Buchi}^\sim \) appears negatively within the premise of the rule defining \( \text{Reach} \), the least fixed-point of \( \text{Reach} \) requires computing the greatest fixed-point of \( \text{Buchi} \) (the dual of \( \text{Buchi}^\sim \)). Ultimately, because \( \text{Buchi}^\sim \) appears negatively within the verification query (Equation (10)), \( \text{Sem}_{\text{Buchi}}^{\text{FP}} \) computes the greatest fixed-point of \( \text{Buchi} \) and the least fixed-points of \( \text{Reach} \) and \( \text{Move} \).

Our tool \textsc{MUSE} dispatches this query to the \( \mu \text{CLP} \) validity solver MuVal [Unno et al. 2023], which proves the above query valid in 21s, thereby proving that the strategy \textit{strat} is a valid solution to the \textsc{SemGuS} synthesis problem in Figure 3.

3 SEMGUS AND SEMGUS\(^\mu\)

This section reviews the \textsc{SemGuS} framework [Kim et al. 2021] and describes the more expressive framework \textsc{SemGuS}\(^\mu\) we propose. A \textsc{SemGuS} synthesis problem is defined in three parts: a grammar defining the syntax of the language over which programs are to be synthesized (Section 3.1), a set of logical formulas defining the semantics of programs in the language (Section 3.2), and a specification defining the properties the synthesized program should exhibit (Section 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 \textit{ranked alphabet} is a tuple \( \langle \Sigma, rk_\Sigma \rangle \) consisting of a finite set of symbols (\( \Sigma \)) 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 \( \Sigma \) is denoted by \( T_\Sigma \). Specifically, \( T_\Sigma \) 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 \textit{typed Regular Tree Grammar} (RTG) is a tuple \( G = \langle N, \Sigma, S, T, a, \delta \rangle \), where \( N \) is a finite set of non-terminal symbols of rank 0; \( \Sigma \) is a ranked alphabet; \( S \in N \) is the starting non-terminal; \( T = \{ \tau_0, \ldots, \tau_k \} \) 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 \( \delta \) a finite set of productions of the form \( A_0 \rightarrow \sigma^i(A_1, \ldots, A_l) \) 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 \( r = A \rightarrow \beta \in \delta \) to \( t \) to produce a tree \( t' \) by replacing the leftmost occurrence of \( A \) in \( t \) with \( \beta \). A tree \( t \in T_\Sigma \) is generated by the grammar \( G(t \in L(G)) \) when \( t \) is the result of applying some sequence of production rules \( r_0, \ldots, r_n \in \delta^n \) to the initial non-terminal \( S \).

**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 \( \text{while}^2, x++^0, x--^0, y++^0, y--^0, \text{seq}^2, 0 < x^0, \) and \( 0 < y^0 \), and productions \( L \rightarrow \text{while}(B, S), S \rightarrow x++, S \rightarrow x--,
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 $\Sigma$, the syntax of formulas and terms are given by the following grammar:

$$
\phi ::= X(t_1, \ldots, t_{rkz}(x)) \mid p(t_1, \ldots, t_{rkz}(p)) \mid \neg \phi_1 \mid \phi_1 \land \phi_2 \mid \forall x : s. \phi_1
$$

$$
t ::= x \mid f(t_1, \ldots, t_{rkz}(f))
$$

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

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

$$
\forall x_0, \ldots, x_n. X_0(x_0) \leftarrow X_1(x_1) \land \cdots \land X_n(x_n) \land F(x_0, \ldots, x_n), \tag{11}
$$

where each $x_i$ 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. [2021], the semantics of programs in the language defined by the regular tree grammar is provided by defining a logical relation and using a least-fixed-point logic, namely 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 Section 2.5, in our work, we use a logic that is more expressive than CHCs to define the elements of the relation—in particular, relations can appear both positively and negatively in the premises of a rule.

**Definition 3.3 (SemGuS$^\delta$ semantics).** Given a first-order theory $\mathcal{T}$ and regular tree grammar $G = \langle N, \Sigma, S, T, a, \delta \rangle$ a semantics for $G$ is a pair $\langle \text{SEM}, \llbracket \cdot \rrbracket \rangle$ where $\text{SEM}$ maps each non-terminal $A \in N$ to a non-empty finite set of uninterpreted relations $\text{SEM}_{\mathcal{A}} = \{ \text{SEM}^1_{\mathcal{A}}, \ldots, \text{SEM}^n_{\mathcal{A}} \}$ and $\llbracket \cdot \rrbracket$ maps each production rule $A_0 \rightarrow \sigma^I(A_1, \ldots, A_i)$ of type $(\tau_0, \ldots, \tau_i)$ and semantic relation $\text{SEM}^j_{A_0} \in \text{SEM}_{\mathcal{A}_0}$ to a formula of the form $\text{SEM}^j_{A_0}(t_{A_0}, \Gamma^0, \gamma^0) \leftarrow \phi$ such that:

- $\phi$ is a (possibly quantified) $T$ formula,
- $t_{A_0}$ is a variable representing elements of $L(A_0)$, $\gamma^0$ is a variable of type $\tau_0$, and $\Gamma^0$ is a variables representing state,
- $\phi$'s free variables belong to $\Gamma^0$, $\gamma^0$, or $\{ t_{A_0} \}$, and
- For each $\text{SEM}^j_{A_k}(t_{A_k}, \Gamma^{k,l}, \gamma^k)$ appearing in $\phi$:
  - $0 \leq k \leq i$ and $\text{SEM}^j_{A_k} \in \text{SEM}_{A_k}$ and
  - $t_{A_k}, \gamma^k$, and $\Gamma^{k,l}$ are defined analogously to $t_{A_0}, \gamma^0, \Gamma^0$.

1The 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 $\mu$CLP), we implicitly assume that one can use values in the algebraic data type to express tree-valued constants.
Example 3.4. Consider the semantics $Sem_{\text{loop}} = \langle SEM, \cdot \rangle$ in Figure 2d. Each non-terminal is mapped to a single semantic relation (i.e., $SEM_L = \{Sem_l\}$, $SEM_S = \{Sem_s\}$, and $SEM_B = \{Sem_B\}$). The semantic function $\cdot$ maps each semantic relation $Sem_A$ and production rule $A \rightarrow \sigma(A_1, \ldots, A_l)$ to the semantic relation whose head is of the form $Sem_A(\sigma(t_1, \ldots, t_n), \Gamma, \Upsilon)$. For example, $[\emptyset < x]_{Sem_B}$ is the rule $Sem_B(\emptyset < x, x, y, b) \leftarrow 0 < x$.

Our semantics generalizes the semantic rules considered by Kim et al. [2021] in two 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.

If we restrict our semantic definition to have a single semantic relation per non-terminal and to rules of the form $Sem_A(t, \Gamma, \Upsilon) \leftarrow \varphi$, where $\varphi$ contains only existential quantification and positive occurrences of semantic relations, then our definition is equivalent to the semantics considered in SemGuS [Kim et al. 2021]. 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. 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 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 $\mathcal{P} = (G = \langle N, \Sigma, T, a, \delta \rangle, \langle SEM, \cdot \rangle, F, \varphi)$, where

- $G$ is a regular tree grammar.
- $\langle SEM, \cdot \rangle$ is a semantics for $G$.
- $F$ is a finite set of functions we want to synthesize—pairs of the form $(f, A)$ 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 $t \in L(A)$.
- $\varphi$ a specification in the theory $T$ such that
  - The free variables of $\varphi$ must be functions to synthesize, $FV(\varphi) \subseteq \{f : (f, A) \in F\}$ and
  - For any $(f, A) \in F$, $f$ appears only in atoms of the form $Sem_A^l(f, \Gamma, \Upsilon)$ where $Sem_A^l \in SEM_A$.

For a semantics $\langle SEM, \cdot \rangle$, an interpretation $\rho$ is a function that maps each semantic relation $Sem_A^l(t, \Gamma, \Upsilon) \in SEM$ to a formula whose free variables are $\Gamma \cup \Upsilon \cup \{t\}$. The interpretation $SEM_{LFP}$ is the interpretation that maps each semantic relation to its least fixed point.

A solution to the SemGuS$^\mu$ problem $\mathcal{P}$ is a function $S$ that maps each $(f, A) \in F$ 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 $\varphi$ is in an atom of the form $Sem_A^l(f, \Gamma, \Upsilon)$ where $Sem_A^l \in SEM_A$. 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,
\(SEM^{\text{LFP}}\) 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 \(\mathcal{P}\).

For example, consider the semantic rules \(\text{Sem}_A(x) \leftarrow \text{Sem}_A(x) \lor \neg\text{Sem}_B(x)\) and \(\text{Sem}_B(x) \leftarrow \text{Sem}_A(x)\)—recall Figure 3d, whose semantics follows a similar pattern. Computing the least fixed point of \(\text{Sem}_A\) requires first computing the greatest fixed point of \(\text{Sem}_B\), and similarly the least fixed point of \(\text{Sem}_B\) requires first computing the greatest fixed point of \(\text{Sem}_A\). That is, the least fixed-point of \(\text{Sem}_A\) requires computing the fixed-point of the following fixed point equations
\[
\text{Sem}_A(x) =_\mu \text{Sem}_A(x) \lor \text{Sem}_B(x) \land \text{Sem}_A(x),
\]
and similarly the least fixed-point of \(\text{Sem}_B\) can be computed using the fixed point equations
\[
\text{Sem}_B(x) =_\mu \text{Sem}_B(x) \lor \text{Sem}_A(x) \land \text{Sem}_A(x).
\]

Example 3.6. Consider the SemGuS problem \(\langle G_{\text{max2}}, \text{Sem}_{\text{max2}}, \text{max2}, \varphi\rangle\) and candidate solution \(s_{\text{max2}}\) in Figure 1. The interpretation \(SEM^{\text{LFP}}\) maps \(\text{Sem}_S(s_{\text{max2}}, x, y, x', y')\) to its least fixed point, which is the formula \(\varphi_{\text{SEM}_{\text{max2}}}\) we computed in Equation (3) to capture the semantics of \(s_{\text{max2}}\).

As such, we conclude that \(SEM^{\text{LFP}} \models \varphi_{\text{max2}}\) for \(\text{max2} \mapsto s_{\text{max2}}\).

4 VERIFYING CANDIDATE PROGRAMS

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

4.1 Encoding Nonrecursive SemGuS\textsuperscript{\(\mu\)} Verification Problems with Quantified SMT

In Section 2.1, we were able to produce a first-order-logic formula that is free of any semantic relations and is satisfiable exactly when \(\varphi_{\text{max2}}\) 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 \(\varphi_{\text{max2}}\) 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 \(A \rightarrow \sigma^i(A_1, \ldots, A_i)\), semantic relation \(\text{Sem}_A \in \text{SEM}_{A}\) of \(A\), and \(\text{[A} \rightarrow \sigma^i(A_1, \ldots, A_i)]\text{Sem}_{A} = \text{Sem}_{A}(t, \Gamma, \Upsilon) \leftarrow \varphi\).

We say that a semantic relation \(\text{Sem}'_A \in \text{SEM}_{A}\) is a \(t\)-ancestor of \(\text{Sem}_A\) if and only if (i) \(\text{Sem}'_A(t, \Gamma, \Upsilon)\) appears in the antecedent \(\varphi\) for some values of \(\Gamma\) and \(\Upsilon\), or (ii) there is some symbol \(\text{Sem}'_A(t, \Gamma, \Upsilon)\) that appears in \(\varphi\) and \(\text{Sem}'_A\) is a \(t\)-ancestor of \(\text{Sem}'_A\).

We say that \(\text{Sem}_A\) is non-recursive on \(t\) if (i) \(\text{Sem}_A\) is not a \(t\)-ancestor of itself, and (ii) for each \(\text{Sem}_{A_1}(t_j, \Gamma, \Upsilon)\) appearing in \(\varphi\), \(\text{Sem}_{A_1}\) is non-recursive on \(t_j\). If \(\text{Sem}_A\) is non-recursive on \(t\) then for any \(\Gamma\) and \(\Upsilon\), the derivation tree of \(\text{Sem}_A(t, \Gamma, \Upsilon)\) has finite height.

Example 4.2 (Non-recursive on \(t\)). Consider the \text{max2} example from Section 2.1. The semantics \(\text{Sem}_S\) is non-recursive on the candidate solution \(s_{\text{max2}} = \text{It}\text{e} (x < y) (x = y) (x = x)\) (as well as every other tree derivable from grammar \(G_{\text{max2}}\)). As shown in Figure 1d, every occurrence of a semantic relation within the premise of a semantic rule is applied to a structurally smaller term of the language \(G_{\text{max2}}\).
Definition 4.3 defines the formula of atoms of the form $Sem_A(t, \Gamma, \Upsilon)$ by replacing the atom with the premise of the rule defining it.

Definition 4.3 (Formula of). Assume that we are given a non-terminal $A \in N$, a semantic relation $Sem_A \in SEM_A$, and a production $A \rightarrow \sigma'(A_1, \ldots, A_n) \in \delta$. If $[A \rightarrow \sigma'(A_1, \ldots, A_n)]_{Sem_A}$ is of the form $Sem_A(t, \Gamma, \Upsilon) \leftarrow \varphi$, then the formula of $Sem_A(t', \Gamma', \Upsilon')$ (denoted by $\varphi$-of ($Sem_A(t', \Gamma', \Upsilon')$)) is $\varphi[t \mapsto t', \Gamma \mapsto \Gamma', \Upsilon \mapsto \Upsilon']$, which replaces the formal arguments of $Sem_A$ with the actual argument of the application.

We now turn to defining the procedure $smt$-formula-of that encodes that a solution $P$ is valid for a SemGuS 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.

1 Procedure $smt$-formula-of($P = \langle G, \langle SEM, \[\] \rangle , F, \varphi \rangle , S$)

   2 rules $\leftarrow \top$  // empty set of rules to begin with
   3 $\varphi$ $\leftarrow$ $\varphi$[$\langle f, A \rangle \in F.f \mapsto S(f)$]  // substitute solution into specification
   4 foreach $Sem_A(t, \Gamma, \Upsilon)$ appearing in $\varphi$ do
   5    $\psi$ $\leftarrow$ $\varphi$-of ($Sem_A(t, \Gamma, \Upsilon)$)  // repeatedly replace semantic relations with their def.
   6    while $Sem_A(t', \Gamma', \Upsilon')$ appears in $\psi$ do
   7        $\psi$ $\leftarrow$ $\psi$[$Sem_A(t', \Gamma', \Upsilon')$ $\mapsto$ $\varphi$-of ($Sem_A(t', \Gamma', \Upsilon')$)]
   8    rules $\leftarrow$ rules $\land$ ($Sem_A(t, \Gamma, \Upsilon)$ $\Rightarrow$ $\psi$)  // update rules to add definition for $Sem_A$
   9 return $\langle\text{rules}, \varphi\rangle$  // rules\textsuperscript{LFP}$ \models \varphi$ if and only if $S$ is valid solution to $P$

Applying $smt$-formula-of to the verification problem in Section 2.1 yields the formula in Equation (1). The following theorem states under which conditions $smt$-formula-of($P, P$) returns a formula that is satisfiable if and only if $P$ is a valid solution to $P$.

Theorem 4.4 ($smt$-formula-of is sound). For any SemGuS problem $P = \langle G = \langle N, X, S, T, a, \delta \rangle , \langle SEM, \[\] \rangle , F, \varphi \rangle$ and solution $P$ of $P$, if $Sem_A$ is non-recursive on $P(f)$ for each occurrence of $Sem_A(f, \Gamma, \Upsilon)$ within the specification $\varphi$, then $smt$-formula-of($P, P$) is valid if and only if $P$ is a valid solution of $P$.

4.2 Encoding CHC-like SemGuS Verification Problems with CHCs and Co-CHCs

In Sections 2.2 and 2.3, we saw how to encode the SemGuS verification problem from Figure 2 into the CHC and co-CHC fragments of first-order logic when using, respectively, the specifications $\varphi_{\text{loop}}$ from Figure 2c and $\varphi_{\text{loop}}^{\text{tot}}$ from Equation (4). In this section, we formalize when and how a SemGuS verification problem may be encoded with CHCs or coCHCs.

To encode the verification problem into either a set of CHCs or coCHCs, we require the semantics of the solution to be equivalent to a set of CHCs (i.e., formulas of the form described in Equation (11)). In this section, we also consider the co-CHC fragment of first-order logic. A co-CHC is a formula of the form:

$$\forall \bar{x}_0, \ldots, \bar{x}_n. R_0(\bar{x}) \Rightarrow R_1(\bar{x}_1) \lor \cdots \lor R_n(\bar{x}_n) \lor F(\bar{x}_0, \ldots, \bar{x}_n),$$

where each component is as described when defining CHCs (cf. Equation (11)). Note that the definitions presented here are logically equivalent to the typical definition used in constraint logic programming [Unno et al. 2023]. For CHCs (respectively coCHCs) the decision problem of interest is “given a set of CHCs (respectively coCHCs) and a query formula of the form $\forall \bar{x}. R(\bar{x}) \Rightarrow \varphi$ (respectively $\forall \bar{x}. \varphi \Rightarrow R(\bar{x})$), determine if the query is derivable from the set of CHCs (resp.
coCHCs).” It is known that this decision problem is equivalent to determining if some interpretation of the uninterpreted relations satisfies each rule and the query formula [Björner et al. 2013]. Furthermore, this decision problem is also equivalent to determining if the least (respectively greatest) interpretation (fixed-point) that satisfies all rules also satisfies the given query [Hetzl and Kloibhofer 2021]. We use this final notion to formulate our verification procedures chc-of and co-chc-of.

**Definition 4.5 (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 \( A \rightarrow \sigma^i(A_1, \ldots, A_i) \), \( \operatorname{Sem}(A) \in \operatorname{SEM}(A) \) a semantic relation of \( A \), and \( [A \rightarrow \sigma^i(A_1, \ldots, A_i)]_{\operatorname{Sem}(A)} = \operatorname{Sem}(t, \Gamma, \Upsilon) \). We say the rules defining \( \operatorname{Sem}(A) \) are CHC-like for \( t \) if and only if (i) \( \varphi \) has no negative occurrences of a semantic relation, (ii) \( \varphi \) contains no universal quantifiers, and (iii) for every \( \operatorname{Sem}(A_j)(t_j, \Gamma_j, \Upsilon_j) \) appearing in \( \varphi \), the rules defining \( \operatorname{Sem}(A_j) \) are CHC-like for \( t_j \).

For example, the rules defining both \( \operatorname{Sem}_{\max} \) and \( \operatorname{Sem}_{\log} \) in Figures 1d and 2d are CHC-like (for any program within their respective grammars), while the rules for \( \operatorname{Sem}_{\text{Buchi}} \) in Figure 3d are not. We now define the procedure chc-of, which encodes as a set of CHCs the property that a solution \( P \) is valid for a SemGuS’s problem (where the semantics is CHC-like for \( P \)). We first define an auxiliary function rules-of that, given a semantic relation \( \operatorname{Sem}(A) \) and tree \( t \in L(A) \), returns the disjunctive normal form (dnf) of the rules defining \( \operatorname{Sem}(A) \) for the root production of \( t \) if \( \operatorname{Sem}(A) \) is CHC-like then each disjunct of the dnf of the rules defining it a CHC. That is, rules-of (\( \operatorname{Sem}(A), t \)) = \( \{\operatorname{Sem}(A,t, \Gamma, \Upsilon) \rightarrow \varphi : \varphi \in \text{dnf}(\varphi)\} \), where \( \operatorname{Sem}(A,t, \Gamma, \Upsilon) \rightarrow \varphi = [A \rightarrow \sigma^i(A_1, \ldots, A_i)]_{\operatorname{Sem}(A)} \) and \( A \rightarrow \sigma^i(A_1, \ldots, A_i) \) is the root production of \( t \). The chc-of procedure produces a set of CHCs by effectively performing a breadth-first search to find each of the rules needed to define the semantics of each of the candidate programs.

```
1 Procedure chc-of((\( G = (N, \Sigma, S, T, a, \delta) \), \( (\operatorname{SEM}, [\cdot]) \), \( F, \varphi \), \( S \))
2     rules ← T ;
3     Q ← \{\( \langle \operatorname{Sem}(A), t \rangle : \operatorname{Sem}(A,t, \Gamma, \Upsilon) \) appears in \( \varphi \}\} ; // Queue of relations that need defining
4     while \( Q \neq \emptyset \) do
5         \( \langle \operatorname{Sem}(A), t' \rangle \) ← pick \( Q \) ;
6         rules' ← rules-of (\( \operatorname{Sem}(A), t' \)) ; // Definition of \( \operatorname{Sem}(A) \) for \( t' \) as a set of CHCs
7         Q ← Q ∪ \{\( \langle \operatorname{Sem}(A_j), t_j \rangle : \operatorname{Sem}(A_j)(t_j, \Gamma_j, \Upsilon_j) \) appears negatively in \( \text{rule}[t \mapsto t'] \) for some rule in \( \text{rules}' \)\} ;
8         rules ← rules ∧ \( \bigwedge \) \( \text{rules}' \) ; // Add rules defining \( \operatorname{Sem}(A) \) for \( t' \) to \( \text{rules} \)
9         \( \psi \leftarrow \varphi[\{f, A \} \in F. f \mapsto S(f)] \); // \( \text{rules}^{\text{LFP}} \models \varphi \) if and only if \( S \) is a valid solution to \( P \)
10    return \( \text{rules}, \psi \) ;
```

The procedure co-chc-of is nearly identical to chc-of. The procedure uses the auxiliary function \( \text{dual}(\varphi) = \neg \varphi[\operatorname{Sem}(t, \Gamma, \Upsilon) \mapsto \neg \operatorname{Sem}(t, \Gamma, \Upsilon)] \) that computes the dual of the input formula (i.e., if \( \varphi \) is a CHC then \( \text{dual}(\varphi) \) is a co-CHC). The co-chc-of procedure changes lines 2 and 9. Line 2 becomes \( \varphi \leftarrow \text{dual}(\varphi)[\{f, A \} \in F. f \mapsto P(f)] \) and line 9 becomes \( \varphi \leftarrow \varphi \land \{\text{dual}(\text{rule}) : \text{rule} \in \text{rules}'\} \).

Finally, if the specification \( \varphi \) contains both positive and negative occurrences of semantic relations, but can be split into two specification \( \varphi^+ \) and \( \varphi^- \) that, respectively, contain only positive and only negative occurrences of semantic relations, then \( \varphi \) can be encoded into two separate programs using the above two encodings. In Theorem 4.6, we state under which conditions chc-of and co-chc-of are sound.

, Vol. 1, No. 1, Article . Publication date: May 2024.
**Theorem 4.6 (chc-of and co-chc-of are sound).** For any SemGuS \( \mu \) problem \( \mathcal{P} = \langle G = \langle N, X, S, T, a, d \rangle, \langle \text{SEM}, [\cdot] \rangle, F, \varphi \rangle \) and solution \( P \) of \( \mathcal{P} \), if for each occurrence of \( \text{Sem}_A(f, \Gamma, \Upsilon) \) within the specification \( \varphi \), it appears negatively (respectively positively) and the rules defining \( \text{Sem}_A \) are CHC-like for \( P(f) \), then the query returned by chc-of(\( \mathcal{P}, P \)) (respectively co-chc-of(\( \mathcal{P}, P \))) is valid if and only if \( P \) is a valid solution of \( \mathcal{P} \).

Conversely, in Theorem 4.7, we prove that, in general, a more expressive fragment of first-order logic is required to verify solutions of an arbitrary SemGuS problems. Specifically, a fragment of the \( \mu \text{CLP} \) calculus that combines both the CHC and coCHC fragments of first-order logic.

**Theorem 4.7 (Verification of SemguS is not reducible to (co-)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 nor coConstrained Horn Clauses.

### 4.3 Encoding all SemGuS\( \mu \) Verification Problems with \( \mu \text{CLP} \)

In Sections 2.4 and 2.5, we examined two SemGuS verification problems for which there is no possible encoding to fixed-point-free formulas, CHCs, or coCHCs. Instead, these problems were encoded into a fragment of first-order logic, \( \mu \text{CLP} \), that allows defining both greatest and least fixed-points. Unlike the previous encodings, for any SemGuS\( \mu \) problem \( \mathcal{P} \) one can always use \( \mu \text{CLP} \) to encode that \( P \) is a valid solution to \( \mathcal{P} \).

A \( \mu \text{CLP} \) formula is a sequence of formulas of the form:

\[
X_0(x_0) \equiv_{\text{fix}_0} \varphi_0 \quad \ldots \quad X_n(x_n) \equiv_{\text{fix}_n} \varphi_n,
\]

where each \( X_i \) is an uninterpreted relation, \( x_i \) is a sequence of term variables, and the \( \varphi_i \) are formulas within some background theory whose free variables are \( x_i \) and which may include positive occurrences of the uninterpreted relations \( X_0, \ldots, X_n \). Each \( \text{fix}_i \) is either \( \mu \) or \( \nu \) referring to whether or not the equation \( X_i(x_i) \equiv_{\text{fix}_i} \varphi_i \) should represent a least or greatest fixed point, respectively. We refer the reader to Unno et al. [2023] for a detailed formalization of \( \mu \text{CLP} \).

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 \( \mu \text{uCLP}-\text{of} \), which encodes as a \( \mu \text{CLP} \) query the property that a solution \( P \) is valid for a SemGuS\( \mu \) problem \( \mathcal{P} \). The procedure is again similar to chc-of and co-chc-of, in that it performs a breadth-first search over the semantic relations to produce the resulting \( \mu \text{CLP} \) query. For a formula \( \varphi \), we use \( \text{Norm}(\varphi) \) to denote the formula \( \varphi \) wherever a negative occurrence of \( \text{Sem}_A(t, \Gamma, \Upsilon) \) is replaced by \( \neg \text{Sem}_A^\mu(t, \Gamma, \Upsilon) \).

Theorem 4.8 states that \( \mu \text{uCLP}-\text{of} \) soundly encodes any SemGuS and SemGuS\( \mu \) problem into a validity query within the \( \mu \text{CLP} \) calculus.

**Theorem 4.8 (\( \mu \text{uCLP}-\text{of} \) is sound).** For any SemGuS\( \mu \) problem \( \mathcal{P} = \langle G, \langle \text{SEM}, [\cdot] \rangle, F, \varphi \rangle \) and solution \( P \) of \( \mathcal{P} \), the query returned by \( \mu \text{uCLP}-\text{of}(\mathcal{P}, P) \) is valid if and only if \( P \) is a valid solution of \( \mathcal{P} \).

Theorem 4.9, states that the SemGuS\( \mu \) semantics can express any \( \mu \text{CLP} \) query—i.e., that any \( \mu \text{CLP} \) 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 \( \mu \text{CLP} \) calculus.

**Theorem 4.9 (SemGuS\( \mu \) semantics and \( \mu \text{CLP} \) are equally expressive).** For every \( \mu \text{CLP} \) query \( \langle \varphi, \text{preds} \rangle \), there is some SemGuS problem \( \mathcal{P} \) and solution \( P \in L(G_P) \) such that \( \langle \varphi, \text{preds} \rangle \) is valid if and only if \( P \) is a valid solution to \( \mathcal{P} \).

Conversely, in Theorem 4.10, we state that verification for SemGuS problems does not require the full generality of \( \mu \text{CLP} \). Specifically, SemGuS verification problems do require a fragment
Table 1. Summary of conditions when each encoding may be used to soundly verify that a program satisfies a SemGuS\(\mu\) problem.

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Condition</th>
<th>FOL Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>smt-formula-of</td>
<td>The semantics of the program is non-recursive (i.e., has a finite derivation tree).</td>
<td>SMT</td>
</tr>
<tr>
<td>chc-of</td>
<td>The semantics of the program is CHC-like and semantic relations appear only negatively within the specification.</td>
<td>CHC</td>
</tr>
<tr>
<td>co-chc-of</td>
<td>The semantics of the program is CHC-like and semantic relations appear only positively within the specification.</td>
<td>coCHC</td>
</tr>
<tr>
<td>muclp-of</td>
<td>Always.</td>
<td>(\mu)CLP</td>
</tr>
</tbody>
</table>

of first-order logic beyond both CHCs and coCHCs, but do not require arbitrary alternations of greatest and least fixed points. As a corollary of Theorems 4.9 and 4.10, SemGuS\(\mu\) is more expressive than SemGuS.

**Theorem 4.10 (SemGuS and \(\mu\)CLP are not equally expressive).** Verifying solutions to SemGuS problems can be encoded within a fragment of \(\mu\)CLP 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 four classes of problems discussed in Section 4.

*Muse* is implemented in OCaml, and uses the following solvers: Z3 for SMT formulas [Bjørner and Janota 2015], Spacer for CHCs [Komuravelli et al. 2013], and MuVal for co-CHCs and \(\mu\)CLP.
queries [Unno et al. 2023]. 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 Section 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 \textsc{reify} is to eliminate program terms from the semantic relations, thus
removing the burden of forcing 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 \textsc{Imp}_\textsc{loop} described in Figure 2.
Below we depict the AST of \( t \) and show the reified semantics of \( \textsc{Sem}_\textsc{loop} \) specialized to \( t \). Because
the program is known \textit{a priori}, the semantics can be reified to remove the explicit AST-valued
argument in the different \textsc{Sem} relations by introducing a new semantic relation for each node of
the AST.

\[
\begin{align*}
\text{seq} & \quad \text{Sem}_S^{x--; y++}(x, y, x', y') \leftarrow \text{Sem}_S^{x--;}(x, y, x', y') \land \text{Sem}_S^{y++}(x', y', x', y') \\
\sqrt{\text{x--}} & \quad \text{Sem}_S^{x--}(x, y, x', y') \leftarrow x' = x - 1 \land y = y' \\
\text{AST of x--; y++} & \quad \text{Sem}_S^{y++}(x, y, x', y') \leftarrow x' = x \land y' = y + 1
\end{align*}
\]

More formally, the reified semantics introduces a new semantic relation for every sub-tree of
the program’s AST. Each occurrence of \( \text{Sem}_{A_j}(t_j, \Gamma_j, \Upsilon_j) \) is then replaced by \( \text{Sem}_{A_j}^{t_j}(\Gamma, \Upsilon) \).

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

1. \( \text{Sem}_{A_j}^{t_j} \) is a reified semantic relation (\( \text{Sem}_{A_j}^{t_j} \in \text{SEM}^{\text{reify}} \)),
2. if \( \text{Sem}_{A_j}^{t_j} \in \text{SEM}^{\text{reify}} \) is a reified semantic relation, \( t \) is of the form \( \sigma^t(t_1, \ldots, t_l) \), and there
   is a rule of the form \( \text{Sem}_{A}(t, \Gamma, \Upsilon) \leftarrow \varphi \in \text{rules} \), then \( \text{Sem}_{A_j}^{t_j}(\Gamma, \Upsilon) \leftarrow \varphi[\text{Sem}_{A_j}(t_j, \Gamma_j, \Upsilon_j) \leftarrow \text{Sem}_{A_j}^{t_j}(\Gamma_j, \Upsilon_j)] \in \text{rules}^{\text{reify}} \) is a reified semantic rule, and
3. if \( \text{Sem}_{A}(\Gamma, \Upsilon) \leftarrow \varphi \in \text{rules}^{\text{reify}} \) is a reified semantic rule and \( \text{Sem}_{A_j}^{t_j} \) appears in \( \varphi \), then
   \( \text{Sem}_{A_j}^{t_j} \in \text{SEM}^{\text{reify}} \) is a reified semantic relation.

**Theorem 5.3 (Reification is Sound).** Given a formula \( \varphi \) and a set, rules, of semantic rules,
let \( \psi \) be the formula in which every occurrence of \( \text{Sem}_{A}(t, \Gamma, \Upsilon) \) is replaced by the reified semantic
relation \( \text{Sem}_{A}^{t_j}(\Gamma, \Upsilon) \), and \( \text{rules}^{\text{reify}} \) is the conjunction of the reified semantic rules produced by
\( \text{reify}(\text{rules}, \text{Sem}_A, t) \) for each \( \text{Sem}_{A}(t, \Gamma, \Upsilon) \) appearing in \( \varphi \). The constraint \( \varphi \) is valid under the original
semantic rules rules if and only if \( \psi \) is valid under the reified semantic rules: \( \text{rules} \models \varphi \iff \text{rules}^{\text{reify}} \models \psi \).

Semantic Relation Inlining. In general, the MuVal solver that we use to solve \( \mu\text{CLP} \) queries scales
poorly in the number of relations used to define the semantics of a program. The goal of the
optimization \textsc{inline} is to eliminate semantic relations by inlining their meanings into a quantified
first-order formula.

The SMT encoding \textsc{smt-formula-of} in Section 2.1 can be seen as an application of \textsc{inline} that
eliminates semantic relations by inlining their meaning into a quantified first-order formula (i.e.
\( \phi_{\text{Sem}} \) in Section 2.1). The optimization \texttt{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 \texttt{INLINE}, is implemented nearly identically to the \texttt{smt-formula-of} encoding defined in Section 4.1. The only difference is the condition of the while loop on line 7 of \texttt{smt-formula-of}. When \texttt{smt-formula-of} is applied to a semantic relation that is recursive over the program of interest, it will fail to terminate and generate an ever-growing formula. The optimization \texttt{INLINE} instead uses the condition “there is some \( \text{Sem}_{A'}(t', \Gamma', \Upsilon') \)” in \( \psi \) such that there is no \( \text{Sem}_{A'}(t', \Gamma'', \Upsilon'') \) appearing in \( \phi \)-of \( \text{Sem}_{A'}(t', \Gamma', \Upsilon') \). In essence, \( \text{Sem}_{A'} \) does not recurse directly on itself for the program \( t' \). This condition ensures that \texttt{INLINE} will always terminate, and can be used even when the semantic relation of interest is recursive on the program of interest.

Example 5.4. Continuing Example 5.1, the semantic-inlining optimization would yield a single semantic relation \( \text{Sem}_{S}^{x'''}(x, y, x', y') \leftarrow \exists x''. x'' = x - 1 \land y'' = y \land x' = x'' \land y' = y'' + 1 \), which inlines the definitions of both \( \text{Sem}_{S}^{x''} \) and \( \text{Sem}_{S}^{y''} \) into the definition of \( \text{Sem}_{S}^{x'''} \).

Quantifier Elimination. In the above example, we see that inlining definitions can leave superfluous quantifiers (e.g. \( \exists x'', y'' \) in \( \text{Sem}_{S}^{x''} \)). 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. The quantifier-elimination optimization applies quantifier elimination to the semantics on a rule-by-rule basis. Continuing the above example, applying quantifier elimination yields a quantifier-free formula defining \( \text{Sem}_{S}^{x''} \): \( \forall (x, y, x', y') \leftarrow x' = x - 1 \land y' = y + 1 \).

6 EXPERIMENTS

We evaluated \texttt{Muse} with respect to the following research questions:

\textbf{Q1}: How effective is \texttt{Muse} at verifying solutions to SemGuS problems?

\textbf{Q2}: How effective are the optimizations from Section 5 at improving \texttt{Muse}’s performance?

\textbf{Q3}: Does \texttt{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. Each experiment was repeated three times with a timeout threshold of 5 minutes, with 6GB of allotted memory. We report the average time for the three runs. For each benchmark, we ran the \texttt{SMT}\texttt{-FORMULA-OF}, \texttt{CHC-OF}, and \texttt{MuCLP-OF} based solvers—we do not report \texttt{CO-CHC-OF} because it also uses the MuVal \( \mu \text{CLP} \) solver as its back-end and is therefore equivalent to \texttt{MuCLP-OF}. (We are not aware of any specialized \texttt{CO-CHC} solvers that we could have used.)

6.1 Benchmarks

We collected 12 SemGuS problems—whose semantics and logical specification were expressed within linear integer arithmetic—from the official SemGuS benchmarks (https://github.com/SemGuS-git/Semgus-Benchmarks), along with 27 new SemGuS problems and 107 problems from SyGuS. For each problem, we designed one correct solution and one incorrect solution to assess our solver’s ability to both prove and disprove whether a program is a correct solution. The total number of problems is 146, and thus the total number of benchmarks is 292. We split our problems into four suites (cf. Table 2).

The first suite SyGuS consists of 80 SemGuS versions of SyGuS problems in the CLIA track. The majority of benchmarks in the CLIA track come from three families of benchmarks: \texttt{ARRAYSEARCH},
Verifying Solutions to Semantics-Guided Synthesis Problems

Table 2. 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.

<table>
<thead>
<tr>
<th>Suite</th>
<th>Total</th>
<th>SMT-FORMULA-OF</th>
<th>CHC-OF</th>
<th>MUCLP-OF</th>
<th>Best</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>isSMT # Solved</td>
<td>isCHC # Solved</td>
<td># Solved</td>
<td># Solved</td>
</tr>
<tr>
<td>SyGuS</td>
<td>160</td>
<td>160 104 0.42s</td>
<td>160 152 0.18s</td>
<td>111 15.89s</td>
<td>152 0.12s</td>
</tr>
<tr>
<td>SyGuS-Imp</td>
<td>54</td>
<td>27 27 0.07s</td>
<td>54 54 2.27s</td>
<td>28 1.32s</td>
<td>54 2.04s</td>
</tr>
<tr>
<td>FuncImp</td>
<td>38</td>
<td>12 12 0.07s</td>
<td>11 11 0.13s</td>
<td>35 4.26s</td>
<td>35 1.02s</td>
</tr>
<tr>
<td>Reach</td>
<td>40</td>
<td>0 – –</td>
<td>0 – –</td>
<td>40 0.66s</td>
<td>40 0.66s</td>
</tr>
<tr>
<td>Total</td>
<td>292</td>
<td>199 143 0.30s</td>
<td>225 216 0.70s</td>
<td>213 8.09s</td>
<td>281 0.68s</td>
</tr>
</tbody>
</table>

ArraySum, and ArrayMax. Each of these problems asks for an LIA term that computes a function over a fixed-sized array.

The second suite SyGuS-Imp consists of 27 imperative versions of some of the ArraySearch, ArraySum, and ArrayMax family of SyGuS problems. They are a family of SemGuS benchmarks—for an imperative language using while loops, assignment, and conditionals—to perform the same tasks in an imperative setting (similar to the one in Figure 1a). There are 27 benchmarks (for arrays of size two to ten for each family of benchmark).

The third suite FuncImp consists of 19 simple imperative and functional program-synthesis tasks. This suite of benchmarks consists of 19 SemGuS problems: two using the language from Figure 1; ten using the loop and increment language from Figure 2; and seven using the simple deterministic functional language modeling possible strategies for the reachability game from Figure 3.

The final suite Reach consists of 10 variations of reachability and Büchi games similar to the one displayed in Figure 3 using different values for the parameters of the game (e.g., bounded vs. unbounded regions, and whether or not the target was stationary). Büchi games are interesting for Muse because the specifications require using both a least and greatest fixed point (in the form of a negated relation).

6.2 Q1: Effectiveness of Muse

Table 2 presents the results of our experiments, summarized per benchmark suite, and provides information about 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 instances fall within the fragment of first-order logic to which they can be applied. We note that the CHC-OF variant solved the most instances, solving 216 instances taking on average under one second. The MUCLP-OF variant comes in a close second, solving 213 instances; however, it averaged eight seconds per instance. Upon further investigation, we found that CHC-OF solved 26 instances not solved by the other two variants, while MUCLP-OF solved 56 instances that were not and could not be solved by the other variants. Overall, 281 of the 292 benchmarks were solved by at least one of the three solvers. For the benchmarks that all solvers could handle, CHC-OF was on average 20X faster than SMT-FORMULA-OF and 40X faster than MUCLP-OF. SMT-FORMULA-OF was on average 2X faster than MUCLP-OF. 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. 10X slower for CHC-OF, 3.6X slower for MUCLP-OF, 0.74X slower for SMT-FORMULA-OF).
To answer Q1, the verification techniques implemented in MUSE are effective and can verify 281/292 of our problem instances. Proving correctness of a solution is generally harder than proving that a solution is incorrect.

6.3 Q2: Effectiveness of the Optimizations from Section 5

Figure 4 illustrates how the optimizations described in Section 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 Section 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 4X 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.

To answer Q2, the optimizations presented in Section 5 are very effective, with reification being the most effective.

6.4 Q3: Integration with an Enumeration-Based SemGuS Synthesizer

To determine the effect that MUSE has on synthesis of solutions to SemGuS problems, we incorporated MUSE in the enumeration-based SemGuS solver Ks2 from (https://github.com/kjcjohnson/ks2-mono). We evaluate the modified synthesizer on 112 SemGuS problems (i.e., each of the SemGuS problems described in Section 6.1, excluding the “reach” category). For each SemGuS problem, we provide 5 input-output examples alongside the logical specification. Ks2 is a top-down enumerative solver that iteratively generates terms and checks the term against a set of input-output examples. We modify Ks2 to incorporate our verifier MUSE to check terms against the logical specification, and to resume searching if the provided solution is incorrect. For each verification problem encountered, we had Ks2 invoke the simplest solver (i.e., smt-formula-of for non-recursive solutions, chc-of...
Verifying Solutions to Semantics-Guided Synthesis Problems

Table 3. Results of running modified Ks2 on SemGuS benchmarks.

<table>
<thead>
<tr>
<th>Suite</th>
<th>SyGuS</th>
<th>SyGuS-Imp</th>
<th>FuncImp</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solved</td>
<td>10</td>
<td>25</td>
<td>10</td>
<td>45</td>
</tr>
<tr>
<td>Timed Out</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>Memed Out</td>
<td>54</td>
<td>0</td>
<td>0</td>
<td>54</td>
</tr>
<tr>
<td>Verification Calls</td>
<td>1210</td>
<td>367</td>
<td>53</td>
<td>1630</td>
</tr>
<tr>
<td>Time (s)</td>
<td>476.46</td>
<td>3319.77</td>
<td>48.14</td>
<td>3835.37</td>
</tr>
<tr>
<td>Verification Time (s)</td>
<td>453.33</td>
<td>3074.16</td>
<td>24.02</td>
<td>3545.51</td>
</tr>
<tr>
<td>Verif. Time / Call (s)</td>
<td>0.37</td>
<td>8.38</td>
<td>0.45</td>
<td>2.18</td>
</tr>
</tbody>
</table>

for CHC-like problems, and \( \mu \text{clp-of} \) for all others) with reification and predicate inlining (the configuration that performed the best in Q2).

The results are summarized in Table 3. We found that Muse enabled Ks2 to solve 45 SemGuS problems. For SemGuS problems that Ks2 did solve, Ks2 required on average 30.2 verification calls, taking on average 2.18 seconds each. By incorporating Muse in Ks2, we enabled Ks2 to solve 35 SemGuS problems that no previous SemGuS solver could solve (we exclude the SyGuS benchmarks because a SyGuS solver could solve such problems). We inspected the 67 SemGuS problems that the modified Ks2 could not solve. The 55 SyGuS problems failed to enumerate the correct solution within the time limit. One of the FuncImp benchmarks similarly timed out during enumeration. The other 8 FuncImp and the 2 SyGuS-Imp problems that timed out during verification after enumerating the correct solution. For the 2 SyGuS-Imp solutions and 5 of the FuncImp solutions, it was possible to verify that they were correct using a different verifier configuration (i.e., additionally using quantifier elimination), while the remaining 3 FuncImp could not be solved using any verifier configuration within the half-hour time limit.

To answer Q3, by modifying Ks2 to incorporate Muse we are able to synthesize 45 verifiably correct solutions, including 35 SemGuS problems that no previous SemGuS solver could solve. Of the 67 problems that remained unsolved, only 11 timed out due to verification (of which 8 more could be solved using the virtual best solver, which runs each verifier configuration in parallel and returns once one variant 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 [Gupta and Rybalchenko 2009; Henzinger et al. 2008; Leino 2010; McMillan 2006]. For functional languages, the typical approach uses type-based reasoning [Pereira and Ravara 2021; Unno et al. 2018; Xi 2002]. Unlike Muse, these approaches are not parameterized to support reasoning for arbitrary 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. Stețănescu et al. [2016] describe how to create an automated program verifier for arbitrary languages automatically, from an operational semantics written in the K framework; the produced verifier is limited to defining reachability properties (i.e., program safety) and is not general enough to reason about termination and liveness properties. In a
similar vein is work on creating and using an intermediate language for verification, such as Boogie [Barnett et al. 2006] or Why3 [Filliâtre and Paskevich 2013]. A key difference between what is involved when working with those tools versus working with 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 of a program, and constructs an appropriate Boogie/Why3 program whose meaning captures the semantics of the program. In contrast, with Muse, the semantics is specified declaratively, using logical relations in SemGuS, thus allowing one to model many diverse scenarios (e.g., our robot example). Many systems have used Boogie as their intermediate language, including Dafny [Leino 2010] and VCC [Cohen et al. 2009]. Other similar systems include Cameleer [Pereira and Ravara 2021] (on top of Why3) and various C analyzers built on top of the FRAMA-C platform [Kirchner et al. 2015]. 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 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 to verify programs. A broad class of work considers programs represented as transition formulas [Alberti et al. 2004; Arnold 1993; Baresi et al. 2012; Farzan and Kincaid 2017]. 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—used in Muse. Another logic-based formalism uses answer-set programming to formulate verification questions [Bromberger et al. 2021; Calvanese et al. 2013; Flederer et al. 2017]. A verifier based on answer-set programming represents the program and its verification conditions in a declarative language (e.g., Prolog or Datalog).

Another line of work translates verification queries into validity (or satisfiability) queries in fragments of first-order logic [Gurfinkel et al. 2015; Unno et al. 2021]. SeaHorn [Gurfinkel et al. 2015] compiles annotated C programs into a system of CHCs to discharge the generated verification conditions. The work of Unno et al. [2021] 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 to perform verification for whatever language is specified in the SemGuS input.

8 CONCLUSION

The SemGuS framework [Kim et al. 2021] is becoming a standard for program synthesis, as happened with the less expressive SyGuS framework. This paper presents the first domain-agnostic and solver-agnostic 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 that generalizes CHCs and co-CHCs), 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 (existing tools could only support input/output examples) and
verify whether these solutions are correct! While our tools can currently handle relatively small programs, improvement to our framework will lead to improvements in any SemGuS solver.

Additionally, while functional and reactive synthesis have historically been considered as two very separate problems, our newly proposed framework SemGuS⁹ and our verification approaches bring these forms of synthesis under the same umbrella, thus opening an exciting avenue of opportunities for building synthesizers that handle both of these settings.

REFERENCES


, Vol. 1, No. 1, Article . Publication date: May 2024.
A \ PROOFS OF THEOREMS

\textbf{Theorem 4.4 (smt-formula-of is sound).} For any $\text{SemGuS}^t$ problem $\mathcal{P} = \langle G = \langle N, \Sigma, S, T, a, \delta \rangle, \langle \text{SEM}, \langle \cdot \rangle \rangle, F, \varphi \rangle$ and solution $P$ of $\mathcal{P}$, if $\text{Sem}_{A}$ is non-recursive on $P(f)$ for each occurrence of $\text{Sem}_{A}(f, \Gamma, \Upsilon)$ within the specification $\varphi$, then $\text{smt-formula-of}(\mathcal{P}, P)$ is valid if and only if $P$ is a valid solution of $\mathcal{P}$.

\textbf{Proof.} The $\text{smt-formula-of}$ procedure replaces the definition of each semantic relation appearing in the specification $\varphi$. The process works by performing a fixedpoint computation, recursively replacing each occurrence of a semantic relation within a definition with the definition of the semantic relation. That is it replaces each occurrence of $\text{Sem}_{A}(t, \Gamma, \Upsilon)$ with it’s definition $\varphi$ of $\text{Sem}_{A}(t, \Gamma, \Upsilon)$. Clearly, each updates maintains the semantics of the semantic relation of interest. That is, we are simply replacing occurrences of semantic relations with their definitions. Finally, we must show that if $\text{Sem}_{A}$ is non-recursive on $t$, then $\text{smt-formula-of}$ terminates. Assume not, then there must be some infinite sequences of substitutions that replace a semantic rule with its definition. However, the semantics can only be applied to sub-terms of the program of interest $t$ (or to $t$ itself). If $t$ is strictly decreasing then clearly, $\text{smt-formula-of}$ must terminate. If instead, there is an infinite sequence of reductions using the same term $t$, then this violates the assumption that $\text{Sem}_{A}$ is non-recursive on $t$. Thus we may conclude $\text{smt-formula-of}$ terminates. \qed

\textbf{Theorem 4.6 (chc-of and co-chc-of are sound.).} For any $\text{SemGuS}^t$ problem $\mathcal{P} = \langle G = \langle N, \Sigma, S, T, a, \delta \rangle, \langle \text{SEM}, \langle \cdot \rangle \rangle, F, \varphi \rangle$ and solution $P$ of $\mathcal{P}$, if for each occurrence of $\text{Sem}_{A}(f, \Gamma, \Upsilon)$ within the specification $\varphi$, it appears negatively (respectively positively) and the rules defining $\text{Sem}_{A}$ are CHC-like for $P(f)$, then the query returned by $\text{chc-of}(\mathcal{P}, P)$ (respectively $\text{co-chc-of}(\mathcal{P}, P)$) is valid if and only if $P$ is a valid solution of $\mathcal{P}$.

\textbf{Proof.} Let rules $\vdash \psi$ be the query returned by $\text{chc-of}$. We must prove that rules$^{\text{LFP}} \vdash \psi$ if and only if $\text{Sem}^{\text{LFP}} \vdash \varphi$. The chc-of procedure proceeds by performing a search over the semantics and tree $t$ jointly to find compute a set of CHCs capturing the semantics of $t$. At each iteration chc-of picks a semantic relation $\text{Sem}_{A}$ and subtree $t'$ and computes a set of CHCs capturing logically equivalent to the rule defining $\text{Sem}_{A}(t, \Gamma, \Upsilon)$. Then each semantic relation sub-tree pair appearing any the rule is added to the queue so that the semantics for the full tree and not just the root production rule. At each iteration, the CHC rules computed by chc-of are computed from the cubes of the disjunctive normal form of the single rule defining the semantic relation $\text{Sem}_{A}$ for the root production rule of the tree $t$. Clearly, these rules are logically equivalent to the single rule they were derived for (i.e. it is easy to show $(a \leftarrow b) \land (a \leftarrow c) \iff (a \leftarrow (b \land c))$. Thus once the algorithm terminates, we are guaranteed that the resulting rules are logically equivalent to the original semantic relations. The proof of coCHCs proceeds similarly. \qed

\textbf{Theorem 4.7 (Verification of SemGuS is not reducible to (co-)CHC Satisfiability).} There exists a program $t$ and SemGuS problem $\text{Sy}$ such that verifying $t$ satisfies $\text{Sy}$ cannot be reduced to satisfiability of Constrained Horn Clauses nor coConstrained Horn Clauses.

\textbf{Proof.} To prove that some solution $T$ satisfies a SemGuS problem $\text{Sy}$, one must prove that the least solution to the semantic rules $\text{SEM}_{\text{Sy}}$ of $\text{Sy}$ satisfies the specification $\varphi_{\text{Sy}}$ when the solution $T$ is substituted for each occurrence of a synthesis function (i.e. $\text{SEM}_{\text{Sy}}^{\text{LFP}} \vdash \varphi_{\text{Sy}}[f \mapsto T(f): f \in \text{dom}(F_{\text{Sy}})]$). By assumption, we know that the semantic rules $\text{SEM}_{\text{Sy}}$ are represented as a set of constrained horn clauses, and the specification $\varphi_{\text{Sy}}$ as an arbitrary first-order formula with no free variables that may include any of the semantic relations defined by the semantic rules. Specifically, the specification $\varphi_{\text{Sy}}$ may contain both negative and positive occurrences of semantic relations.
Thus in general (and specifically for $\phi_{comm}$ in eq. (7)), the specification is not a valid query formula for checking satisfiability using constrained horn clauses.

Recall that satisfiability problem for CHCs is typically formulated as reachability (i.e., given a set of CHCs and a formula, can the formula be derived from the set of CHC rules?). This formulation is known to be logically equivalent to determining if some interpretation (namely the least interpretation [Hetzl and Kloibhofer 2021]) of the uninterpreted relations satisfies every CHC rule and the given formula when the provided query formula is of the form [Björner et al. 2013]:

$$\forall x. R(x) \Rightarrow \psi$$

where $R$ is an uninterpreted relation defined by the CHC rules, and $\psi$ is a constraint over the variables $\bar{x}$. Note, that there is no way to transform arbitrary first-order formulas (i.e. SemGuS specification) to this form. Thus we may conclude that there are some SemGuS verification problems not reducible to CHC satisfiability.

**Theorem 4.8 (muclp-of is sound).** For any SemGuS$^\mu$ problem $P = \langle G, \langle SEM, \{\} \rangle, F, \varphi \rangle$ and solution $P$ of $P$, the query returned by muclp-of($P, P$) is valid if and only if $P$ is a valid solution of $P$.

**Proof.** Let rules $\models \psi$ be the query returned by muclp-of. We must prove rules$^{FP} \models \psi$ if and only if SEM$^{LFP} \models \varphi[f \mapsto P(f)]$. We proceed to prove by induction on the number of iterations of the main loop of muclp-of, that rules are logically equivalent to to rules defining the semantic relations that have been explored previously. Trivially this holds true before entry to the loop. Let $SEMA, t', fix$ be the next elements to processed by the main loop of muclp-of. The first step is to retrieve the rule defining the $Sem_A$ for the root production rule of $t'$. Next, we optionally dualize the the rule if fix is $\nu$ (i.e. we are interested in computing the dual relation of $Sem_A$). Next, the rule is normalized. This process replaces very negative occurrence of a semantic relation with it’s dual relation (i.e. $Sem^n_A$). Each of these transformations preserves the semantic relations interpretation.

Then finally, every positive occurring semantic relation $Sem_{A_j}$ is added to the queue to compute its semantics, and every semantic relation appearing negatively is added to the queue to compute it’s dual semantics. Thus, in some future iteration, each semantic relation of interest will have it’s a rule defining it computed and added to the set of rules. Once, muclp-of, terminates it does so with a set of rules rules that are equivalent to the definition of the semantic relations appearing in the specification $\varphi$. Since $\psi = Norm(\varphi)$ it’s clear that rules$^{FP} \models \psi$ if and only if SEM$^{LFP} \models \varphi$.

**Theorem 4.9 (SemGuS$^\mu$ semantics and $\mu$CLP are equally expressible).** For every $\mu$CLP query $\langle \varphi, preds \rangle$, there is some SemGuS problem $P$ and solution $P \in L(GP)$ such that $\langle \varphi, preds \rangle$ is valid if and only if $P$ is a valid solution to $P$.

**Proof.** Let $X_0(x_0) = _{fix_0} \varphi_0; \ldots; X_n(x_n) = _{fix_n} \varphi_n$ be the sequence of predicates making up $pred$.

Consider the following grammar $A := \tau$, that consists of a single nonterminal $A$ with a single production rule $\tau$. The following is the language of $A$ consists of a single word $L(A) = \{ \tau \}$. First, define $Y_i$ to be $X_i$ if fix$_i$ is $\mu$ and $X_i^{-}$ otherwise, and similarly, $\psi_j$ to be $\varphi_j$ if $fix_j$ is $\mu$ and $¬\varphi_j$ otherwise.

Let $SEM_A = \{ Y_0, \ldots, Y_n \}$ be the set of newly introduced predicate relations. For each $Y_i$ define $[\tau]_{Y_i}$ to be the rule $Y_i(\tau, x_i) \mapsto \psi_i[m]$ where $m$ maps every occurrence of $X_i(x_i)$ in $\psi_i$ to $Y_i(t_A, x_i)$ if $fix_i$ is $\mu$ and $¬Y_i(t_A, x_i)$ otherwise ($t_A$ is a variable representing elements of $L(A)$). Similarly, define $\varphi_{spec} = \varphi[m]$ to be the constraint of the muclp query using the same substitution $m$. Finally, define $\mathcal{P}$ to be the SemGuS$^\mu$ problem defined by the grammar for $A$, semantics $\langle SEM_A, \{\} \rangle$, specification, and set of synthesis functions $F = \{ \langle t_A, A \rangle \}$.

In order to prove that $\text{pred}^{FP} \models \varphi$ if and only if $SEM_A^{LFP} \models \varphi_{spec}$, we prove that for each predicate $X_i$ that if $fix_i$ is $\mu$ then the fixpoint of $X_i$ is the least fixpoint of $Y_i (X_i^{FP} = Y_i^{LFP})$; otherwise, that the fixpoint of $X_i$ is the dual of the least fixpoint of $Y_i (X_i^{FP} = ¬Y_i^{LFP})$. We proceed by induction on $n$ the
Verifying Solutions to Semantics-Guided Synthesis Problems

We may then conclude that the reified rules and original rules are logically equivalent.

\[ \square \]

\[ (at \ most) \ contains \ equations \ describing \ the \ semantic \ and \ dual \ semantic \ relations \ with \ no \ interaction \]

\[ \exists X \] if the property holds, then it will continue to hold regardless of how the semantic rules are defined.

\[ \text{rules} \ with \ program \ terms \ are \ logically \ equivalent \ to \ their \ reified \ version. \ The \ argument \ holds \ that, \]

\[ \text{hypothesis} \ may \ be \ used \ to \ show \ that \ the \ substitution \ preserve \ logical \ equivalence \ of \ the \ two \ rules. \]

\[ \text{Sem} \ \text{computes} \ \text{the \ rule} \]

\[ \text{trivial. \ Next, \ assume \ for \ any \ subtree} \]

\[ \text{semantic \ rules} \]

\[ \text{rules} \]

\[ \text{fix} \]

\[ \text{of} \]

\[ \text{define} \]

\[ \text{for} \]

\[ \text{each} \]

\[ \text{and} \]

\[ \text{we \ can \ conclude \ that} \]

\[ \text{exactly} \]

\[ \text{the \ substitution \ substitutes \ equivalent} \]

\[ \text{terms. \ We \ note, \ that \ the \ rule \ defining} \]

\[ \text{current} \]

\[ \text{to \ prove \ the \ case \ for} \]

\[ \text{if} \]

\[ \text{then} \]

\[ \text{this} \]

\[ \text{case} \]

\[ \text{have \ proved} \]

\[ \text{if} \]

\[ \text{then} \]

\[ \text{the} \]

\[ \text{least \ fixpoint \ of} \]

\[ \text{□} \]

\[ \text{Theorem 4.10 (SemGuS and } \mu\text{CLP \ are \ not \ equally \ expressive). \ Verifying \ solutions \ to \ SemGuS \ problems \ can \ be \ encoded \ within \ a \ fragment \ of } \mu\text{CLP \ that \ uses \ at \ most \ one \ alternation \ between \ greatest} \]

\[ \text{least \ fixpoints.} \]

\[ \text{Proof. \ As \ stated \ in \ Theorem 4.9, \ every \ SemGuS} \]

\[ \text{query \ and \ vice \ versa. \ Thus, \ verification \ of \ solutions \ to \ SemGuS} \]

\[ \text{requires \ the \ full \ generality} \]

\[ \text{SemGuS} \]

\[ \text{problem} \]

\[ \text{clearly \ verification \ of \ SemGuS \ problems \ can \ be \ reduced \ to \ validity \ of \ a } \mu\text{CLP \ query.} \]

\[ \text{However, \ verification \ of \ solutions \ to \ SemGuS \ problems \ do \ not \ require \ the \ full \ generality} \]

\[ \text{of} \]

\[ \text{Specifically, \ the \ encoding \ described \ in} \]

\[ \text{will \ result in} \]

\[ \text{formula} \]

\[ \text{that} \]

\[ \text{(at \ most) \ contains \ equations \ describing \ the \ semantic \ and \ dual \ semantic \ relations \ with \ no \ interaction} \]

\[ \text{between \ the} \]

\[ \text{thus} \]

\[ \text{may} \]

\[ \text{not} \]

\[ \text{require \ any \ interaction \ between \ greatest} \]

\[ \text{points.} \]

\[ \text{Therefore, \ we \ may \ conclude} \]

\[ \text{that} \]

\[ \text{SemGuS} \]

\[ \text{problems} \]

\[ \text{require \ a \ more \ expressive \ logical \ encoding \ than \ the \ encoding \ of \ verification \ of \ solutions \ to \ SemGuS} \]

\[ \text{problems.} \]

\[ \square \]

\[ \text{Theorem 5.5 (Reification is sound). \ Given \ a \ formula} \]

\[ \text{let} \]

\[ \text{be the \ formula \ in \ which \ every \ occurrence} \]

\[ \text{Sem}_{A}(t, \Gamma, \Upsilon) \] is \ replaced \ by \ the \ reified \ semantic \ relation \ \text{Sem}^r_{A}(\Gamma, \Upsilon), \text{and} \] \text{rules}^\text{reify} \text{is} \ the \ conjunction \ of \ the \ reified \ semantic \ rules \ produced \ by \ \text{reify} \text{(rules, } \text{Sem}_{A}, t) \text{ for each } \text{Sem}_{A}(t, \Gamma, \Upsilon) \text{ appearing in } \varphi. \] \text{The \ constraint} \ \varphi \ \text{is \ valid \ under \ the \ original} \ \text{semantic \ rules \ if \ and \ only \ if} \] \text{only \ if} \ \psi \ \text{is \ valid \ under} \ \text{the \ reified \ semantic \ rules: \ } \models \varphi \iff \text{rules}^\text{reify} \models \psi.

\[ \text{Proof. \ We \ begin \ by \ induction \ on \ the \ height \ of} \]

\[ \text{the} \]

\[ \text{case \ when} \]

\[ \text{trivial. \ Next, \ assume \ for \ any \ subtree} \]

\[ \text{t} \]

\[ \text{that} \]

\[ \text{Sem}_{A}(t', \Gamma', \Upsilon') \] is \ logically \ equivalent \ to \ \text{Sem}^r_{A}(\Gamma', \Upsilon'). \text{ \ reify \ computes} \ the \ rule \ \text{Sem}^r_{A}(\Gamma, \Upsilon) \iff \varphi[m] \text{ where} \] \text{m} \text{ is \ a \ map \ substituting \ every \ occurrence \ of} \ \text{Sem}_{A,j}(t_j, \Gamma_j, \Upsilon_j) \text{ with} \ \text{Sem}^r_{A,j}(\Gamma_j, \Upsilon_j). \text{ Either} \ t_j \text{ is \ a \ subtree \ of} \ t \text{ or} \ t_j = t. \text{ In} \text{ the} \text{ first} \text{ case, \ the} \text{ inductive} \text{ hypothesis} \text{ may} \text{ be} \text{ used} \text{ to} \text{ show} \text{ that} \text{ the} \text{ substitution} \text{ preserve} \text{ logical} \text{ equivalence} \text{ of} \text{ the} \text{ two} \text{ rules.} \text{ Otherwise, \ if} \ t_j = t, \text{ we \ use} \text{ coinduction \ to} \text{ show} \text{ that} \text{ the} \text{ possibly} \text{ infinite} \text{ cycle} \text{ of} \text{ cycle} \text{ of} \text{ semantic} \text{ rules} \text{ with} \text{ program} \text{ terms} \text{ are} \text{ logically} \text{ equivalent} \text{ to} \text{ their} \text{ reified} \text{ version.} \text{ The} \text{ argument} \text{ holds} \text{ that,} \text{ if} \text{ the} \text{ property} \text{ holds,} \text{ then} \text{ it} \text{ will} \text{ continue} \text{ to} \text{ hold} \text{ regardless} \text{ of} \text{ how} \text{ the} \text{ semantic} \text{ rules} \text{ are} \text{ defined.} \text{ We \ may \ then \ conclude} \text{ that} \text{ the} \text{ reified} \text{ rules} \text{ and} \text{ original} \text{ rules} \text{ are} \text{ logically} \text{ equivalent.} \]