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 μCLP calculus, a fixed-point logic that generalizes Constrained Horn Clauses and co-Constrained Horn Clauses. Our encoding into μCLP 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) μCLP queries. Furthermore, our encoding shines light on some limitations of the SemGuS framework, such as its inability to model nondeterminism and reactive synthesis. We thus propose a modification to SemGuS that makes it more expressive, and for which verifying solutions is exactly equivalent to proving validity of a query in the μCLP calculus. Our implementation of SemGuS verifiers based on the above encoding can verify instances that were not even encodable in previous work. Furthermore, we use our SemGuS verifiers within an enumeration based SemGuS solver to correctly synthesize solutions to SemGuS problems that no previous SemGuS synthesizer could solve.

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/nnnnnnn.nnnnnnn

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 *Sem*, and a specification φ (which is allowed to mention the semantic relations defined by *Sem*), we show that the problem of verifying whether p—when evaluated according to *Sem*—satisfies φ can be expressed as a validity check in the μCLP calculus. μCLP is a fixed-point logic that generalizes CHCs and co-CHCs by combining both least and greatest fixed points with interpreted first-order theories. While SemGuS uses only least fixed points to define the semantics of programs (i.e., 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 μCLP. Because of the complexity of building solvers for checking validity of μCLP queries, the second contribution of the paper is to identify fragments of SemGuS verification problems that can be reduced to checking satisfiability of first-order-logic formulas, CHCs, and co-CHCs, for which more scalable solvers exists.

Our study highlights a strong connection between SemGuS and μCLP, and raises the question of whether there exist programming languages for which verification is expressible using μCLP, 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 μCLP and SemGuS, we define a minimal extension of SemGuS—i.e., we allow relations to appear in a negated form in the semantic definitions—which results in a new framework that aligns exactly with what is verifiable using μCLP. Finally, we incorporate our verification technique into a synthesizer for SemGuS problems that is capable of solving SemGuS problems with complex logical specifications.

Contributions. Our work makes the following contributions.

- We identify how the problem of verifying programs in SemGuS is tightly related to proving validity in fixed-point logics (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 μCLP) and show that our extension of SemGuS aligns exactly with what is verifiable using μCLP (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_{max2} defining the syntax of the language under consideration (Figure 1a).
- A set of constrained Horn clauses Sem_{max2} that inductively define the semantics (as a least fixed point) of all programs in the language (Figure 1d).
- A specification φ_{max2} that describes how the synthesized program should behave when evaluated according to Sem_{max2} (Figure 1b).

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

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

Beyond 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 φ_{max2} is valid when interpreted using the least solution of the semantic relations.

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

While the semantic relation Sem_{max2} is defined via a least fixed point over a set of constrained Horn clauses, the positive occurrence of Sem_S within φ_{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 $Sem^{LFP}_{max2} \models \psi$ would be to check that the formula $Sem^{LFP}_{max2} \wedge \neg\psi$ is unsatisfiable, which requires every occurrence of a semantic relation in ψ 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 *max*2, where the semantic definitions are recursively defined with respect to the term's proper subterms, one can

$$S ::= \texttt{x = } E \mid \texttt{y = } E \mid S\texttt{;}S \mid \texttt{Ite}\ B\ S\ S$$

$$E ::= \texttt{0} \mid \texttt{1} \mid \texttt{x} \mid \texttt{y} \mid E\texttt{+}E$$

$$B ::= E \texttt{ < } E$$

(a) Grammar G_{max2}

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

(b) Specification φ_{max2}

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

(c) Solution s_{max2}

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

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

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

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

(d) Semantics Sem_{max2}

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

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

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

Because the tree is finite, the relation $Sem_{s_{max2}}$ can be defined 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 $\varphi_{Sem_{s_{max2}}}$, which exactly characterizes $Sem_{s_{max2}}$:

$$\exists r_b.(\exists r_s, r_t.r_s = x \wedge r_t = y \wedge r_b \Leftrightarrow r_s < r_t) \wedge ((r_b \wedge x' = y \wedge y' = y) \vee (\neg r_b \wedge x' = x \wedge y' = y)) \tag{3}$$

We can now replace the term $Sem_S(s_{max2}, x, y, x', y)$ in the formula in Equation (1) with the freshly computed term $\varphi_{Sem_{s_{max2}}}$ and obtain the following formula, which is entirely describable in first-order logic without requiring any fixed-point reasoning.

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

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_{max2} is a correct solution to this SemGuS problem. Given the incorrect candidate program `Ite (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_{loop} is restricted so that assignments can only increment and decrement variables (Figure 2a)—i.e., a correct program for the task must contain a loop. The semantics Sem_{loop} of this language (Figure 2d) is defined similarly to the one in our previous example. The key distinction is how the second CHC, which defines the (big-step) semantics of loops, is not structurally decreasing—i.e., the loop l appears again in a semantic relation in the premise of the CHC. The specification φ_{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_{loop}) terminates with the output y', then y' is twice x (Figure 1b).

Similar to the previous example, verifying whether the program s_{loop} given in Figure 2b is correct requires proving that the query $Q_{loop} \triangleq Sem^{LFP}_{loop} \models \varphi_{loop}[f_{loop} \mapsto s_{loop}]$ is valid—i.e., that the candidate solution s_{loop} satisfies the specification φ_{loop} when interpreted using the least solution of the semantic rules Sem_{loop}. Unlike the previous example, the specification φ_{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_{loop} in 0.2 seconds, thereby proving that s_{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_{loop} because the specification φ_{loop} did not contain positive occurrences of the semantic relations in Sem_{loop}. In the next example, we consider the same grammar G_{loop} and semantics Sem_{loop} as in Figure 2, but introduce a modified specification φ^{tot}_{loop} that requires a form of total correctness:

$$\varphi^{tot}_{loop} = \forall x, y'.(0 \leq x \wedge 2x = y') \Rightarrow Sem_L(f_{loop}, x, 0, 0, y') \tag{4}$$

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 $Q^{tot}_{loop} = Sem^{LFP}_{loop} \models \varphi^{tot}_{loop}[f_{loop} \mapsto s_{loop}]$ cannot be solved using a least-fixed-point logic because the specification has a positive occurrence of the semantic relation Sem_L. Instead, we show that one *can* construct a logically equivalent query

$$\begin{aligned} L &:= \texttt{while}\ B\ \texttt{do}\ \langle S\rangle \\ S &:= \texttt{x++} \mid \texttt{x--} \mid \texttt{y++} \\ &\quad \mid \texttt{y--} \mid S;S \\ B &:= \texttt{0 < x} \mid \texttt{0 < y} \end{aligned}$$

(a) Grammar G_{loop}

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

(b) Solution s_{loop}

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

(c) Specification φ_{loop}

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

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

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

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

(d) Semantics Sem_{loop}

Fig. 2. Computing $2x$ in the language Imp_{loop} that allows increments and loops.

$\bar{Q}^{tot}_{loop}$ that can be represented using *co-Constrained Horn Clauses* (co-CHCs). Whereas CHCs are able to define least fixed points, co-CHCs are able to define greatest fixed points. This capability allows us to define a new relation—the *complement relation* $Sem^{\neg}_L(f_{loop}, x, 0, 0, y')$—as a coCHC, which allows us to reason about negative occurrences of the form $\neg Sem_L(\cdot,\cdot,\cdot,\cdot,\cdot)$. This approach allows us to solve a new query in which all relations (*i*) are defined as greatest fixed points, and (*ii*) appear negatively within the specification:

$$\bar{Q}^{tot}_{loop} = Sem^{\neg\ GFP}_{loop} \models \forall x, y'.(0 \le x \land 2x = y') \Rightarrow \neg Sem^{\neg}_L(s_{loop}, x, 0, 0, y'), \tag{5}$$

where $Sem^{\neg\ GFP}_{loop}$ is the greatest fixed point of the coCHCs that define the dual semantics of Sem_{loop}. Intuitively, the query $\bar{Q}^{tot}_{loop}$ 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_{loop}, Sem_{loop}, and φ^{tot}_{loop} and produces the query $\bar{Q}^{tot}_{loop}$. The process dualizes every semantic relation and each semantic rule. For example, our encoding produces the following rule to define $Sem^{\neg}_S(\texttt{s; t}, x, y, x', y')$.

$$Sem^{\neg}_S(\texttt{s; t}, x, y, x', y') \Rightarrow \forall x'', y''. Sem^{\neg}_S(\texttt{s}, x, y, x'', y'') \lor Sem^{\neg}_S(\texttt{t}, x'', y'', x', y')$$

After the encoding, the produced query $\bar{Q}^{tot}_{loop}$ is logically equivalent to the original query Q^{tot}_{loop} 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 co-CHC solver to prove the validity of this query to determine that s_{loop} is correct with respect to the specification φ^{tot}_{loop}. Due to the lack of coCHC solvers, Muse uses the μCLP 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:

$$\varphi^{both}_{loop} = \forall x, y'. 0 \leq x \Rightarrow (Sem_L(x, 0, 0, y') \Leftrightarrow 2x = y') \tag{6}$$

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 φ_{loop} and φ^{tot}_{loop}. To verify that a candidate program satisfies the specification in φ^{both}_{loop}, it is sufficient to check that the candidate program satisfies both φ_{loop} and φ^{tot}_{loop} by checking validity of the queries Q_{loop} and $\bar{Q}^{tot}_{loop}$.

2.4 Hyperproperties: μCLP

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.)

$$\varphi_{comm} \triangleq \forall x, y, x', y'. Sem_L(s, x, y, x', y') \Rightarrow Sem_L(s, y, x, x', y') \tag{7}$$

The specification φ_{comm} could arise when trying to synthesize a program like

$$s_{plus} = \texttt{while 0 < x do x--; y++} \tag{8}$$

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 Sem_{loop} in Figure 2d.

To prove that the program s_{plus} satisfies the specification φ_{comm}, one must reason simultaneously about the relation $Sem_L(sum, x, y, x', y')$ and its complement $Sem^{\neg}_L(sum, x, y, x', y')$. 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 φ_{comm}—and in fact every specification expressible in SemGuS—can be reduced to checking validity in the μCLP calculus, a logic that combines least- and greatest-fixed-point reasoning [Unno et al. 2023].

Muse reduces this verification problem to the following μCLP query that combines both the positive semantics Sem_{loop} and negative semantics $Sem^{\neg}_{loop}$:

$$Q_{plus} \triangleq Sem^{LFP}_{loop} \wedge Sem^{\neg\ GFP}_{loop} \models \forall x, y, x', y'. Sem^{\neg}_L(s_{plus}, x, y, x', y') \vee Sem_L(s_{plus}, y, x, x', y'), \tag{9}$$

which follows from Equation (7) by (i) instantiating s as s_{plus}, (ii) replacing "$Sem_L(s_{plus}, x, y, x', y') \Rightarrow \ldots$" with "$\neg Sem_L(s_{plus}, x, y, x', y') \vee \ldots$," and ($iii$) replacing "$\neg Sem_L(s_{plus}, x, y, x', y')$" with "$Sem^{\neg}_L(s_{plus}, x, y, x', y')$."

In Muse, we use the μCLP solver MuVal [Unno et al. 2023] to solve Q_{comm} in 6.2 seconds, thereby proving that s_{plus} is commutative.

$$S ::= \texttt{repeat}\ S \mid \texttt{stay} \mid L \mid R \mid \texttt{L};S \mid \texttt{R};S$$

(a) Robot Strategy

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

(b) Solution *strat*

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

(c) Specification

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

(d) Semantics Sem_{Buchi}.

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

2.5 Beyond SemGuS

Section 2.4 showed that for every SemGuS problem, one can verify the correctness of a candidate solution using a μCLP solver. This connection raises a natural question in the opposite direction: *Are there programming languages for which verification is expressible using μCLP, 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 μCLP.

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

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

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

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

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

Our tool Muse dispatches this query to the μCLP validity solver MuVal [Unno et al. 2023], which proves the above query valid in 21s, thereby proving that the strategy $strat$ is a valid solution to the SemGuS$^{\mu}$ synthesis problem in Figure 3.

3 SEMGUS AND SEMGUS$^{\mu}$

This section reviews the SemGuS framework [Kim et al. 2021] and describes the more expressive framework SemGuS$^{\mu}$ we propose. A SemGuS synthesis problem is defined in three parts: a grammar defining the syntax of the language over which programs are to be synthesized (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 *ranked alphabet* is a tuple $\langle \Sigma, rk_\Sigma \rangle$ consisting of a finite set of symbols (Σ) and a function $rk_\Sigma : \Sigma \to \mathbb{N}$ that associates every symbol with a rank. For any $n \geq 0$, $\Sigma^n \subseteq \Sigma$ denotes the set of symbols of rank n. The set of all *(ranked) Trees* over Σ is denoted by T_Σ. Specifically, T_Σ is the least set such that $\Sigma^0 \subseteq T_\Sigma$ and if $\sigma^k \in \Sigma^k$ and $t_1, \ldots, t_k \in T_\Sigma$, then $\sigma^k(t_1, \ldots, t_k) \in T_\Sigma$. In the remainder, we assume a fixed ranked alphabet $\langle \Sigma, rk_\Sigma \rangle$.

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

Given a tree $t \in T_{\Sigma \cup N}$, one may apply the production rule $r = A \to \beta \in \delta$ to t to produce a tree t' by replacing the leftmost occurrence of A in t with β. A tree $t \in T_\Sigma$ is generated by the grammar G ($t \in L(G)$) 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 $\texttt{while}^2$, $\texttt{x++}^0$, $\texttt{x--}^0$, $\texttt{y++}^0$, $\texttt{y--}^0$, $\texttt{seq}^2$, $\texttt{0 < x}^0$, and $\texttt{0 < y}^0$, and productions $L \to \texttt{while}(B, S)$, $S \to \texttt{x++}$, $S \to \texttt{x--}$,

$S \to$ `y++`, $S \to$ `y--`, $S \to$ `seq`(S, S), $B \to$ `0 < x`, and $B \to$ `0 < y`. In the examples in the paper, we often drop the ranks of symbols and use infix notation to enhance readability.[1]

3.2 Semantics via Logical Relations and Fixed-point Logics

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

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

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

$$\forall \bar{x}_0, \ldots, \bar{x}_n.X_0(\bar{x}_0) \longleftarrow X_1(\bar{x}_1) \wedge \cdots \wedge X_n(\bar{x}_n) \wedge F(\bar{x}_0, \ldots, \bar{x}_n), \tag{11}$$

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

In the SemGuS framework originally defined by Kim et al. [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$^\mu$ 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 $\langle SEM, [\![\cdot]\!]\rangle$ where SEM maps each non-terminal $A \in N$ to a non-empty finite set of uninterpreted relations ($SEM_A = \{Sem^1_A, \ldots, Sem^n_A\}$) and $[\![\cdot]\!]$ maps each production rule $A_0 \longrightarrow \sigma^i(A_1, \ldots, A_i)$ of type $(\tau_0, \ldots, \tau_i)$ and semantic relation $Sem^j_{A_0} \in SEM_{A_0}$ to a formula of the form $Sem^j_{A_0}(t_{A_0}, \Gamma^{0,j}, \Upsilon^0) \longleftarrow \varphi$ such that:

- φ is a (possibly quantified) $\mathcal{T}$ formula,
- t_{A_0} is a variable representing elements of $L(A_0)$, Υ^0 is a variable of type τ_0, and $\Gamma^{0,j}$ are variables representing state,
- φ's free variables belong to $\Gamma^{0,j}$, Υ^0, or $\{t_{A_0}\}$, and
- For each $Sem^l_{A_k}(t_{A_k}, \Gamma^{k,l}, \Upsilon^k)$ appearing in φ:
  - $0 \le k \le i$ and $Sem^l_{A_k} \in SEM_{A_k}$ and
  - t_{A_k}, Υ^k, and $\Gamma^{k,l}$ are defined analogously to t_{A_0}, Υ^0, $\Gamma^{0,j}$.

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

Example 3.4. Consider the semantics $Sem_{loop} = \langle SEM, [\![\cdot]\!] \rangle$ 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 $A \to \sigma^i(A_1, \ldots, A_i)$ to the semantic relation whose head is of the form $Sem_A(\sigma^i(t_1, \ldots, t_n), \Gamma, \Upsilon)$. For example, $[\![\texttt{0 < x}]\!]_{Sem_B}$ is the rule $Sem_B(\texttt{0 < x}, x, y, b) \leftarrow 0 < x$.

Our semantics generalizes the semantic rules considered by Kim et al. [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 $Sem_A(t_A, \Gamma, \Upsilon) \leftarrow \varphi$, where φ contains only existential quantification and positive occurrences of semantic relations, then our definition is equivalent to the semantics considered in SemGuS [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 $\mathcal{P} = \langle G = \langle N, \Sigma, T, a, \delta \rangle, \langle SEM, [\![\cdot]\!] \rangle, F, \varphi \rangle$, 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 $\langle f, A \rangle$ where f is a variable representing a procedure that we want to synthesize, and $A \in N$ is the root nonterminal from which f is to be derived—i.e., the solution for f must be a tree $t \in L(A)$.
- φ a specification in the theory $\mathcal{T}$ such that
  - The free variables of φ must be functions to synthesize, $FV(\varphi) \subseteq \{f : \langle f, A \rangle \in F\}$ and
  - For any $\langle f, A \rangle \in F$, f appears only in atoms of the form $Sem^i_A(f, \Gamma, \Upsilon)$ where $Sem^i_A \in SEM_A$.

For a semantics $\langle SEM, [\![\cdot]\!] \rangle$, an interpretation ρ is a function that maps each semantic relation $Sem^l_A(t, \Gamma, \Upsilon) \in SEM$ 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 $\langle f, A \rangle \in F$ to a tree $t \in L(A)$. The solution S is *valid* when $SEM^{LFP} \models \varphi[\langle f, A \rangle \in F.f \mapsto S(f)]$. Note that the values $S(f)$ being substituted into the formula are program-valued constants represented as terms in the algebraic data type for G. Moreover, by the last case of Definition 3.5, each occurrence of f in φ is in an atom of the form $Sem^i_A(f, \Gamma, \Upsilon)$ where $Sem^i_A \in SEM_A$. Consequently, in the resulting formula, each such program-valued constant will be interpreted according to the least fixed point of a semantic relation of an appropriate kind. Note that, while SEM^{LFP} appears to use only least fixed points, because we allow semantic relations to appear negated within the premise of a semantic rule,

SEM^{LFP} represents arbitrarily nested greatest and least fixed points whose top-level fixed point is a least fixed point. We say that $\mathcal{P}$ is *realizable* if there exists a valid solution to $\mathcal{P}$.

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

Example 3.6. Consider the SemGuS problem $\langle G_{max2}, Sem_{max2}, max2, \varphi_{max2}\rangle$ and candidate solution s_{max2} in Figure 1. The interpretation SEM^{LFP} maps $Sem_S(s_{max2}, x, y, x', y')$ to its least fixed point, which is the formula $\varphi_{SEM_{s_{max2}}}$ we computed in Equation (3) to capture the semantics of s_{max2}. As such, we conclude that $SEM^{LFP} \models \varphi_{max2}[max2 \mapsto s_{max2}]$.

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$^\mu$ framework described in Section 3 can be used to define verification problems that require the full capabilities of μCLP. We now describe each encoding in turn. In the remainder of this section, we consider a fixed SemGuS$^\mu$ problem $\mathcal{P} = \left\langle G = \langle N, \Sigma, S, T, a, \delta\rangle, \left\langle SEM, [\![\cdot]\!]\right\rangle, F, \varphi\right\rangle$ and candidate solution P.

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

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

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

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

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

Example 4.2 (Non-recursive on t). Consider the *max2* example from Section 2.1. The semantics Sem_S is non-recursive on the candidate solution s_{max2} = `Ite (x < y) (x = y) (x = x)` (as well as every other tree derivable from grammar G_{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_{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 $A \to \sigma^i(A_1, \ldots, A_n) \in \delta$. If $[\![A \to \sigma^i(A_1, \ldots, A_n)]\!]_{Sem_A}$ is of the form $Sem_A(t, \Gamma, \Upsilon) \leftarrow \varphi$, then the formula of $Sem_A(t', \Gamma', \Upsilon')$ (denoted by $\varphi\text{-}of(Sem_A(t', \Gamma', \Upsilon'))$) 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$^\mu$ problem (where the semantics is non-recursive on P) into first-order logic without fixed points (i.e., quantified SMT formulas). SMT-FORMULA-OF repeatedly replaces every occurrence of a semantic relation with the premise of the rule that defines it.

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

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($\mathcal{P}, P$) returns a formula that is satisfiable if and only if P is a valid solution to $\mathcal{P}$.

THEOREM 4.4 (SMT-FORMULA-OF IS SOUND). *For any SemGuS$^\mu$ problem $\mathcal{P} = \langle G = \langle N, \Sigma, S, T, a, \delta \rangle, \langle SEM, [\![\cdot]\!] \rangle, F, \varphi \rangle$ and solution P of $\mathcal{P}$, if Sem_A is non-recursive on $P(f)$ for each occurrence of $Sem_A(f, \Gamma, \Upsilon)$ within the specification φ, then* SMT-FORMULA-OF($\mathcal{P}, P$) *is valid if and only if P is a valid solution of $\mathcal{P}$.*

4.2 Encoding CHC-like SemGuS$^\mu$ Verification Problems with CHCs 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 φ_{loop} from Figure 2c and φ^{tot}_{loop} 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:

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

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 $\forall \bar{x}. R(x) \Rightarrow \varphi$ (respectively $\forall \bar{x}.\ \varphi \Rightarrow R(x)$), 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 [Bjørner 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 $A \to \sigma^i(A_1, \ldots, A_i)$, $Sem_A \in SEM_A$ a semantic relation of A, and $[\![A \to \sigma^i(A_1, \ldots, A_i)]\!]_{Sem_A} = Sem_A(t, \Gamma, \Upsilon) \leftarrow \varphi$.

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

For example, the rules defining both Sem_{max2} and Sem_{loop} in Figures 1d and 2d are CHC-like (for any program within their respective grammars), while the rules for Sem_{Buchi} in Figure 3d are not. We now define the procedure CHC-OF, which encodes as a set of CHCs the property that a solution P is valid for a SemGuS$^\mu$ problem (where the semantics is CHC-like for P). We first define an auxillary function *rules-of* that, given a semantic relation Sem_A and tree $t \in L(A)$, returns the disjunctive normal form (dnf) of the rules defining Sem_A for the root production of t—if Sem_A is CHC-like then each disjunct of the dnf of the rules defining it a CHC. That is, $\textit{rules-of}(Sem_A, t) = \{Sem_A(t, \Gamma, \Upsilon) \leftarrow \psi : \psi \in \textit{dnf}(\varphi)\}$, where $Sem_A(t, \Gamma, \Upsilon) \leftarrow \varphi = [\![A \to \sigma^i(A_1, \ldots, A_i)]\!]_{Sem_A}$ and $A \to \sigma^i(A_1, \ldots, A_i)$ 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$(\langle G = \langle N, \Sigma, S, T, a, \delta\rangle, \langle SEM, [\![\cdot]\!]\rangle, F, \varphi\rangle, S)$
2 $\quad$ $\textit{rules} \leftarrow \top$;
3 $\quad$ $Q \leftarrow \{\langle Sem_A, t\rangle : Sem_A(t, \Gamma, \Upsilon)$ appears in $\varphi\}$; // Queue of relations that need defining
4 $\quad$ **while** $Q \neq \emptyset$ **do**
5 $\quad\quad$ $\langle Sem_A, t'\rangle \leftarrow$ **pick** Q ;
6 $\quad\quad$ $\textit{rules}' \leftarrow \textit{rules-of}(Sem_A, t')$; // Definition of Sem_A for t' as a set of CHCs
7 $\quad\quad$ $Q \leftarrow Q \cup \{\langle Sem_{A_j}, t_j\rangle : Sem_{A_j}(t_j, \Gamma, \Upsilon)$ appears negatively in $\textit{rule}[t \mapsto t']$
8 $\quad\quad\quad$ for some rule in $\textit{rules}'\}$;
9 $\quad\quad$ $\textit{rules} \leftarrow \textit{rules} \wedge \bigwedge \textit{rules}'$; // Add rules defining Sem_A for t' to *rules*
10 $\quad$ $\psi \leftarrow \varphi[\langle f, A\rangle \in F.f \mapsto S(f)]$;
11 $\quad$ **return** $\langle \textit{rules}, \psi\rangle$; // $\textit{rules}^{LFP} \models \psi$ if and only if S is a valid solution to $\mathcal{P}$

The procedure CO-CHC-OF is nearly identical to CHC-OF. The procedure uses the auxiliary function $\textit{dual}(\varphi) = \neg\varphi[Sem_A(t, \Gamma, \Upsilon) \mapsto \neg Sem_A^{\neg}(t, \Gamma, \Upsilon)]$ that computes the dual of the input formula (i.e., if φ is a CHC then $\textit{dual}(\varphi)$ is a co-CHC). The CO-CHC-OF procedure changes lines 2 and 9. Line 2 becomes $\varphi \leftarrow \textit{dual}(\varphi[\langle f, A\rangle \in F.f \mapsto P(f)])$ and line 9 becomes $\varphi \leftarrow \varphi \wedge \{\textit{dual}(\textit{rule}) : \textit{rule} \in \textit{rules}'\}$.

Finally, if the specification φ contains both positive and negative occurrences of semantic relations, but can be split into two specification φ^+ and φ^- that, respectively, contain only positive and only negative occurrences of semantic relations, then φ can be encoded into two separate problems using the above two encodings. In Theorem 4.6, we state under which conditions CHC-OF and CO-CHC-OF are sound.

Theorem 4.6 (chc-of and co-chc-of are sound.). *For any SemGuS$^\mu$ problem $\mathcal{P} = \left\langle G = \langle N, \Sigma, S, T, a, \delta\rangle, \left\langle SEM, [\![\cdot]\!]\right\rangle, F, \varphi\right\rangle$ and solution P of $\mathcal{P}$, if for each occurrence of $Sem_A(f, \Gamma, \Upsilon)$ within the specification φ, it appears negatively (respectively positively) and the rules defining 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 μCLP 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 μCLP

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, μCLP, that allows defining both greatest and least fixed-points. Unlike the previous encodings, for any SemGuS$^\mu$ problem $\mathcal{P}$ one can *always* use μCLP to encode that P is a valid solution to $\mathcal{P}$.

A μCLP formula is a sequence of formulas of the form:

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

where each X_i is an uninterpreted relation, $\bar{x_i}$ is a sequence of term variables, and the φ_i are formulas within some background theory whose free variables are $\bar{x_i}$ and which may include positive occurrences of the uninterpreted relations $X_0, \dots, X_n$. Each fix_i is either μ or ν referring to whether or not the equation $X_i(\bar{x_i}) =_{fix_i} \varphi_i$ should represent a least or greatest fixed point, respectively. We refer the reader to Unno et al. [2023] for a detailed formalization of μCLP.

In the SemGuS$^\mu$ semantics (Definition 3.5), every semantic relation's definition is oriented as a least fixed point. However, our semantics does allow one to introduce greatest fixed-points by taking the negation of a semantic relation. We now turn to defining muclp-of, which encodes as a μCLP query the property that a solution P is valid for a SemGuS$^\mu$ problem $\mathcal{P}$. The procedure is 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 μCLP query. For a formula φ, we use $Norm(\varphi)$ to denote the formula φ wherever a negative occurrence of $Sem_A(t, \Gamma, \Upsilon)$ is replaced by $\neg Sem_A^{\neg}(t, \Gamma, \Upsilon)$.

Theorem 4.8 states that muclp-of(s)oundly encodes any SemGuS and SemGuS$^\mu$ problem into a validity query within the μCLP calculus.

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

Theorem 4.9, states that the SemGuS$^\mu$ semantics can express any μCLP query—i.e., that any μCLP validity query can be equivalently reduced to a SemGuS$^\mu$ verification problem. Thus, SemGuS$^\mu$ can encode any problem that can be encoded within the μCLP calculus.

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

Conversely, in Theorem 4.10, we state that verification for SemGuS problems does not require the full generality of μCLP. Specifically, SemGuS verification problems do require a fragment

Procedure MUCLP-OF($\mathcal{P} = \langle G, \langle SEM, [\![\cdot]\!] \rangle, F, \varphi \rangle, S$)
  $rules \leftarrow \top$;
  $Q \leftarrow \{\langle Sem_A, t, \mu \rangle : Sem_A(t, \Gamma, \Upsilon)$ appears in $\varphi\} \cup$
    $\{\langle Sem_A, t, \nu \rangle : Sem_A^{\neg}(t, \Gamma, \Upsilon)$ appears in $\varphi\}$;
  while $Q \neq \emptyset$ **do**
    $\langle Sem_A, t', fix \rangle \leftarrow$ **pick** Q ;
    $rule \leftarrow rules\text{-}of(Sem_A, t')$;
    if $fix = \mu$ **then**
      $rule \leftarrow Norm(rule)$; `// Compute the rule as a least fixed point`
    else
      $rule \leftarrow Norm(dual(rule))$; `// Compute the rule as a greatest fixed point`
    $Q \leftarrow Q \cup \{\langle Sem_{A_j}, t_j, \mu \rangle : Sem_{A_j}(t_j, \Gamma, \Upsilon)$ appears in $rule[t \mapsto t']\}$;
    $Q \leftarrow Q \cup \{\langle Sem_{A_j}, t_j, \nu \rangle : Sem_{A_j}^{\neg}(t_j, \Gamma, \Upsilon)$ appears in $rule[t \mapsto t']\}$;
    $rules \leftarrow rules \land rule$;
  $\psi \leftarrow Norm(\varphi[\langle f, A \rangle \in F. f \mapsto S(f)])$;
  return $\langle rules, \psi \rangle$; // $rules^{FP} \models \psi$ if and only if S is a valid solution to $\mathcal{P}$

Table 1. Summary of conditions when each encoding may be used to soundly verify that a program satisfies a SemGuS$^{\mu}$ problem.

Encoding	Condition	FOL Fragment
SMT-FORMULA-OF	The semantics of the program is non-recursive (i.e., has a finite derivation tree).	SMT
CHC-OF	The semantics of the program is CHC-like and semantic relations appear only negatively within the specification.	CHC
CO-CHC-OF	The semantics of the program is CHC-like and semantic relations appear only positively within the specification.	coCHC
MUCLP-OF	Always.	μCLP

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

5 IMPLEMENTATION

We implement our algorithms in a tool, called Muse, which extends SemGuS to SemGuS$^{\mu}$. Muse supports all of the encoding schemes for the 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 μCLP 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 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 Imp_{loop} described in Figure 2. Below we depict the AST of t and show the reified semantics of Sem_{loop} specialized to t. Because the program is known *a priori*, the semantics can be reified to remove the explicit AST-valued argument in the different *Sem* relations by introducing a new semantic relation for each node of the AST.

$$\begin{array}{c} \texttt{seq} \\ \swarrow \quad \searrow \\ \texttt{x--} \quad \texttt{y++} \\ \text{AST of } \texttt{x--; y++} \end{array} \qquad \begin{array}{l} Sem_S^{\texttt{x--; y++}}(x, y, x', y') \leftarrow Sem_S^{\texttt{x--}}(x, y, x'', y'') \wedge Sem_S^{\texttt{y++}}(x'', y'', x', y') \\ Sem_S^{\texttt{x--}}(x, y, x', y') \leftarrow x' = x - 1 \wedge y = y' \\ Sem_S^{\texttt{y++}}(x, y, x', y') \leftarrow x' = x \wedge y' = y + 1 \end{array}$$

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

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

(1) Sem_A^t is a reified semantic relation ($Sem_A^t \in SEM^{reify}$),
(2) if $Sem_A^t \in SEM^{reify}$ is a reified semantic relation, t is of the form $\sigma^i(t_1, \ldots, t_i)$, and there is a rule of the form $Sem_A(t, \Gamma, \Upsilon) \leftarrow \varphi \in rules$, then $Sem_A^t(\Gamma, \Upsilon) \leftarrow \varphi[Sem_{A_j}(t_j, \Gamma_j, \Upsilon_j) \mapsto Sem_{A_j}^{t_j}(\Gamma_j, \Upsilon_j)] \in rules^{reify}$ is a reified semantic rule, and
(3) if $Sem_A^t(\Gamma, \Upsilon) \leftarrow \varphi \in rules^{reify}$ is a reified semantic rule and $Sem_{A_j}^{t_j}$ appears in φ, then $Sem_{A_j}^{t_j} \in SEM^{reify}$ is a reified semantic relation.

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

Semantic Relation Inlining. In general, the MuVal solver that we use to solve μCLP queries scales poorly in the number of relations used to define the semantics of a program. The goal of the optimization INLINE is to eliminate semantic relations by inlining their meanings into a quantified first-order formula.

The SMT encoding SMT-FORMULA-OF in Section 2.1 can be seen as an application of INLINE that eliminates semantic relations by inlining their meaning into a quantified first-order formula (i.e.

$\varphi_{Sem_{max2}}$ in Section 2.1). The optimization INLINE applies a similar insight to as many semantic relations as possible to reduce the number of semantic relations that the final solver has to deal with.

The inlining optimization INLINE, is implemented nearly identically to the SMT-FORMULA-OF encoding defined in Section 4.1. The only difference is the condition of the while loop on line 7 of SMT-FORMULA-OF. When 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 INLINE instead uses the condition "there is some $Sem_{A'}(t', \Gamma', \Upsilon')$ in ψ such that there is no $Sem_{A'}(t', \Gamma'', \Upsilon'')$ appearing in $\varphi\text{-}of\,(Sem_{A'}(t', \Gamma', \Upsilon'))$." In essence, $Sem_{A'}$ does not recurse directly on itself for the program t'. This condition ensures that 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 $Sem_S^{\texttt{x--};\ \texttt{y++}}(x, y, x', y') \leftarrow \exists x'', y''. x'' = x - 1 \wedge y'' = y \wedge x' = x'' \wedge y' = y'' + 1$, which inlines the definitions of both $Sem_S^{\texttt{x--}}$ and $Sem_S^{\texttt{y++}}$ into the definition of $Sem_S^{\texttt{x--;y++}}$.

Quantifier Elimination. In the above example, we see that inlining definitions can leave superfluous quantifiers (e.g. $\exists x'', y''.$ in $Sem_S^{\texttt{x--};\ \texttt{y++}}$). 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 $Sem_S^{\texttt{x--};\ \texttt{y++}}(x, y, x', y') \leftarrow x' = x - 1 \wedge y' = y + 1$.

6 EXPERIMENTS

We evaluated MUSE with respect to the following research questions:

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

All experiments were conducted on a desktop running Ubuntu 18.04 LTS, equipped with a 4-core Intel(R) Xeon(R) processor running at 3.2GHz with 12GB of memory. 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 SMT-FORMULA-OF, CHC-OF, and MUCLP-OF based solvers—we do not report CO-CHC-OF because it also uses the MuVal μCLP solver as its back-end and is therefore equivalent to MUCLP-OF. (We are not aware of any specialized 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$^\mu$ 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: ARRAYSEARCH,

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.

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

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

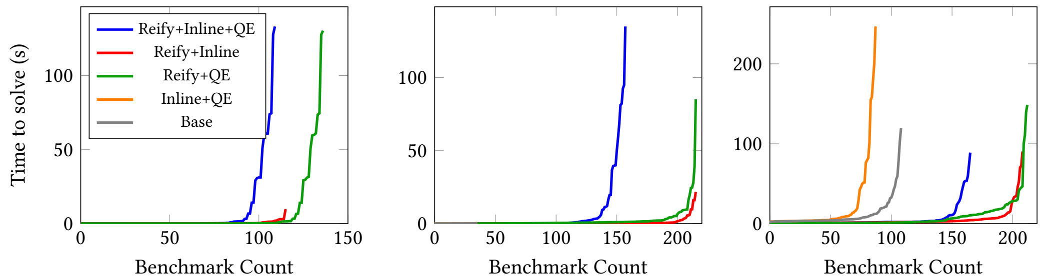
The second suite SyGuS-Imp consists of 27 imperative versions of some of the ArraySearch, ArraySum, and ArrayMax family of SyGuS problems. They are a family of SemGuS benchmarks—for an imperative language using while loops, assignment, and conditionals—to perform the same tasks in an imperative setting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



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

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

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

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

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

for CHC-like problems, and MUCLP-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. Stefănescu 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 t of a program, and constructs an appropriate Boogie/Why3 program whose meaning captures the semantics of t. In contrast, with Muse, the semantics is specified declaratively, using logical relations in SemGuS$^{\mu}$, thus allowing one to model many diverse scenarios (e.g., our robot example). Many systems have used Boogie as their intermediate language, including Dafny [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$^{\mu}$ framework uses a logic-based approach to specifying semantics, which allows Muse to be applied to a wide variety of problems.

Logic-Based Verification. There is also a substantial body of work that uses fragments of first-order logic 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$^{\mu}$—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$^{\mu}$ 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 SemGuS$^\mu$ 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

Marco Alberti, Davide Daolio, Paolo Torroni, Marco Gavanelli, Evelina Lamma, and Paola Mello. 2004. Specification and verification of agent interaction protocols in a logic-based system. In *Proceedings of the 2004 ACM symposium on Applied computing*. 72–78.

Rajeev Alur, Salar Moarref, and Ufuk Topcu. 2018. Compositional and symbolic synthesis of reactive controllers for multi-agent systems. *Information and Computation* 261 (2018), 616–633.

André Arnold. 1993. Verification and comparison of transition systems. In *Colloquium on Trees in Algebra and Programming*. Springer, 121–135.

Luciano Baresi, Angelo Morzenti, Alfredo Motta, and Matteo Rossi. 2012. A logic-based semantics for the verification of multi-diagram UML models. *ACM SIGSOFT Software Engineering Notes* 37, 4 (2012), 1–8.

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

Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. 2013. On solving universally quantified horn clauses. In *Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings 20*. Springer, 105–125.

Nikolaj S Bjørner and Mikolás Janota. 2015. Playing with Quantified Satisfaction. *LPAR (short papers)* 35 (2015), 15–27.

Martin Bromberger, Irina Dragoste, Rasha Faqeh, Christof Fetzer, Markus Krötzsch, and Christoph Weidenbach. 2021. A datalog hammer for supervisor verification conditions modulo simple linear arithmetic. In *International Symposium on Frontiers of Combining Systems*. Springer, 3–24.

Diego Calvanese, Giuseppe De Giacomo, Marco Montali, and Fabio Patrizi. 2013. Verification and synthesis in description logic based dynamic systems. In *International Conference on Web Reasoning and Rule Systems*. Springer, 50–64.

Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*. Springer, 23–42.

Loris D'Antoni, Qinheping Hu, Jinwoo Kim, and Thomas Reps. 2021. Programmable program synthesis. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33*. Springer, 84–109.

Azadeh Farzan and Zachary Kincaid. 2017. Strategy synthesis for linear arithmetic games. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.

Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3—where programs meet provers. In *Programming Languages and Systems: 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 22*. Springer, 125–128.

Frank Flederer, Ludwig Ostermayer, Dietmar Seipel, and Sergio Montenegro. 2017. Source code verification for embedded systems using prolog. *arXiv preprint arXiv:1701.00630* (2017).

Sumit Gulwani. 2012. Synthesis from examples. In *WAMBSE (Workshop on Advances in Model-Based Software Engineering) Special Issue, Infosys Labs Briefings*, Vol. 10. Citeseer.

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

Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn verification framework. In *International Conference on Computer Aided Verification*. Springer, 343–361.

Thomas A Henzinger, Thibaud Hottelier, and Laura Kovács. 2008. Valigator: A verification tool with bound and invariant generation. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 333–342.

Stefan Hetzl and Johannes Kloibhofer. 2021. A fixed-point theorem for Horn formula equations. *arXiv preprint arXiv:2109.04633* (2021).

Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. 2021. Semantics-guided synthesis. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–32.

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2015. Frama-C: A software analysis perspective. *Formal aspects of computing* 27 (2015), 573–609.

Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M Clarke. 2013. Automatic abstraction in SMT-based unbounded software model checking. In *International Conference on Computer Aided Verification*. Springer, 846–862.

K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*. Springer, 348–370.

Kenneth L McMillan. 2006. Lazy abstraction with interpolants. In *Computer Aided Verification: 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings 18*. Springer, 123–136.

Mário Pereira and António Ravara. 2021. Cameleer: A Deductive Verification Tool for OCaml. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33*. Springer, 677–689.

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

Andrei Stefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. 2016. Semantics-based program verifiers for all languages. *ACM SIGPLAN Notices* 51, 10 (2016), 74–91.

Hiroshi Unno, Yuki Satake, and Tachio Terauchi. 2018. Relatively complete refinement type system for verification of higher-order non-deterministic programs. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 1–29.

Hiroshi Unno, Tachio Terauchi, Yu Gu, and Eric Koskinen. 2023. Modular Primal-Dual Fixpoint Logic Solving for Temporal Verification. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 2111–2140.

Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. 2021. Constraint-based relational verification. In *International Conference on Computer Aided Verification*. Springer, 742–766.

Hongwei Xi. 2002. Dependent types for program termination verification. *Higher-Order and Symbolic Computation* 15 (2002), 91–131.

Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Murphy Berzish, Julian Dolby, and Xiangyu Zhang. 2017. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods in System Design* 50 (2017), 249–288.

A PROOFS OF THEOREMS

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

Proof. The smt-formula-of procedure replaces the definition of each semantic relation appearing in the specification φ. 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 $Sem_A(t, \Gamma, \Upsilon)$ with it's definition φ-$of(Sem_A(t, \Gamma, \Upsilon))$. 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 Sem_A is non-recursive on t, then 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, 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 Sem_A is non-recursive on t. Thus we may conclude smt-formula-of terminates. □

Theorem 4.6 (chc-of and co-chc-of are sound.). *For any SemGuS$^\mu$ problem* $\mathcal{P} = \langle G = \langle N, \Sigma, S, T, a, \delta \rangle, \langle SEM, [\![\cdot]\!] \rangle, F, \varphi \rangle$ *and solution* P *of* $\mathcal{P}$*, if for each occurrence of* $Sem_A(f, \Gamma, \Upsilon)$ *within the specification* φ*, it appears negatively (respectively positively) and the rules defining* 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* $\mathcal{P}$.

Proof. Let $rules \models \psi$ be the query returned by chc-of. We must prove that $rules^{LFP} \models \psi$ if and only if $SEM^{LFP} \models \varphi$. 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 Sem_A and subtree t' and computes a set of CHCs capturing logically equivalent to the rule defining $Sem_A(t, \Gamma, \Upsilon)$. 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 nromal form of the single rule defining the semantic relation 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 $(a \leftarrow b) \wedge (a \leftarrow c)) \Leftrightarrow (a \leftarrow (b \wedge c))$. 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. □

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.*

Proof. To prove that some solution T satisfies a SemGuS problem Sy, one must prove that the least solution to the semantic rules SEM_{sy} of Sy satisfies the specification φ_{Sy} when the solution T is substituted for each occurence of a synthesis function (i.e. $SEM_{Sy}^{LFP} \models \varphi_{sy}[f \mapsto T(f) : f \in dom(F_{sy})]$). By assumption, we know that the semantic rules SEM_{Sy} are represented as a set of constrained horn clauses, and the specification φ_{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 φ_{Sy} may contain both negative *and* positive occurrences of semantic relations.

Thus in general (and specifically for φ_{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 [Bjørner et al. 2013]:

$$\forall \bar{x}.\ R(x) \Rightarrow \psi$$

where R is an uninterpreted relation defined by the CHC rules, and ψ 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 $\mathcal{P} = \langle G, \langle SEM, [\![\cdot]\!] \rangle, F, \varphi \rangle$ and solution P of $\mathcal{P}$, the query returned by* muclp-of$(\mathcal{P}, P)$ *is valid if and only P is a valid solution of $\mathcal{P}$.*

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 ν (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. $Sem^{\neg}_{A'}$). 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 teh 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 φ. 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 μCLP are equally expressive). *For every μCLP query $\langle \varphi, preds \rangle$, there is some SemGuS problem $\mathcal{P}$ and solution $P \in L(G_{\mathcal{P}})$ such that $\langle \varphi, preds \rangle$ is valid if and only if P is a valid solution to $\mathcal{P}$.*

Proof. Let $X_0(\bar{x}_0) =_{fix_0} = \varphi_0; \ldots ; X_n(\bar{x}_n) =_{fix_n} \varphi_n$ be the sequence of predicates making up *pred*.

Consider the following grammar $A := \top$, that consists of a single nonterminal A with a single production rule $\top$. That is the the language of A consists of a single word $L(A) = \{\top\}$. First, define Y_i to be X_i if fix_i is μ and $X_i^{\neg}$ otherwise, and similarly, ψ_i to be φ_i if fix_i is μ and $\neg\varphi_i$ otherwise.

Let $SEM_A = \{Y_0, \ldots, Y_n\}$ be the set of newly introduced predicate relations. For each Y_i define $[\![\top]\!]_{Y_i}$ to be the rule $Y_i(\top, \bar{x}_i) \leftarrow \psi_i[m]$ where m maps every occurrence of $X_j(\bar{x}_j)$ in ψ_i to $Y_j(t_A, \bar{x}_j)$ if fiX_j is μ and $\neg Y_j(t_A, \bar{x}_j)$ 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 $pred^{FP} \models \varphi$ if and only if $SEM_A^{LFP} \models \varphi_{spec}$, we prove that for each predicate X_i that if fix_i is μ 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 ($X_i^{FP} = \neg Y_i^{LFP}$). We proceed by induction on n the number of rules defining the μCLP query. First, consider the case when there are no rules. Trivially, the claim holds. Now, assume that for each $i > 0$ that the claim holds for X_i and Y_i. We no proceed to prove the case for X_0 and Y_0. Consider the case when fix_0 is μ. By definition $X_i^{FP} = X_i^{LFP}$, $X_i = Y_i$ and $[\![\top]\!]_{y_i} = Y_i(\top, \bar{x}_i) \longleftarrow \varphi_i[m]$. Both X_i and Y_i are defined as least fixed points. Additionally $\varphi_i[m]$ is identical to φ_i except every occurrence of X_j has been replace with either Y_i if fix_j is μ or by $\neg Y_i$ if fix_j is ν. In either case, we may it is clear from the inductive hypothesis that the substitution preserves fixedpoints. Since, both rules are defined as least fixedpoints, with logically equivalent definitions, we can conclude that X_i^{FP} and Y_i^{LFP} are equivalent. Next, we consider the case when fix_0 is ν. Then by definition, $Y_0 = X_0^{\neg}$ and $[\![\top]\!]_{Y_0}$ is $Y_0(\top, \bar{x}_0) \longleftarrow \neg\varphi_0[m]$. Similarly, to the previous case, we may use the IH to assume that for each $X_j \neq X_0$ the substitution substitutes equivalent terms. We note, that the rule defining Y_0 is dual to the rule defining X_0. Thus, we can may conclude this case and have proved that if fix_i then the fixpoint of X_i is the least fixpoint of Y_i and otherwise X_i is the dual fixpoint of Y_i. □

Theorem 4.10 (SemGuS and μCLP are not equally expressive). *Verifying solutions to SemGuS problems can be encoded within a fragment of μCLP that uses at most one alternation between greatest and least fixed points.*

Proof. As stated in theorem 4.9, every SemGuS$^\mu$ verification problem is expressible as a μCLP query and vice versa. Thus, verification of solutions to SemGuS$^\mu$ problems requires the full generality of the μCLP calculus. Since every SemGuS problem is equivalently representable as a SemGuS$^\mu$ problem, clearly verification of SemGuS problems can be reduced to validity of a μCLP query. However, verification of solutions to SemGuS problems do not require the full generality of the μCLP calculus. Specifically, the encoding described in muclp-of will result in a μCLP formula that (at most) contains equations describing the semantic and dual semantic relations with no interaction between the two (i.e. does not require any interaction between greatest and least fixed-points). Thus we may conclude the fact that verification of solutions to SemGuS$^\mu$ problems in general require a more expressive logical encoding than the encoding of verification of solutions to SemGuS problems. □

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

Proof. We begin by induction on the height of the tree t. The case when t has height 0 is trivial. Next, assume for any subtree t' of t that $Sem_A(t', \Gamma', \Upsilon')$ is logically equivalent to $Sem_A^{t'}\Gamma', \Upsilon$. reify computes the rule $Sem_A^t(\Gamma, \Upsilon) \longleftarrow \varphi[m]$ where m is a map substituting every occurrence of $Sem_{A_j}(t_j, \Gamma_j, \Upsilon_j)$ with $Sem_{A_j}^{t_J}(\Gamma_j, \Upsilon_j)$. Either t_j is a subtree of t or $t_j = t$. In the first case, the inductive hypothesis may be used to show that the substitution preserve logical equivalence of the two rules. Otherwise, if $t_j = t$, we use coinduction to show that the possibly infinite cycle of cycle of semantic rules with program terms are logically equivalent to their reified version. The argument holds that, if the property holds, then it will continue to hold regardless of how the semantic rules are defined. We may then conclude that the reified rules and original rules are logically equivalent. □