Automating Pruning in Top-Down Enumeration for Program Synthesis Problems with Monotonic Semantics

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

In top-down enumeration for program synthesis, abstraction-based pruning uses an abstract domain to approximate the set of possible values that a partial program, when completed, can output on a given input. If the set does not contain the desired output, the partial program and all its possible completions can be pruned. In its general form, abstraction-based pruning requires manually designed, domain-specific abstract domains and semantics, and thus has only been used in domain-specific synthesizers.

This paper provides sufficient conditions under which a form of abstraction-based pruning can be automated for arbitrary synthesis problems in the general-purpose Semantics-Guided Synthesis (SemGuS) framework without requiring manually-defined abstract domains. We show that if the semantics of the language for which we are synthesizing programs exhibits some monotonicity properties, one can obtain an abstract interval-based semantics for free from the concrete semantics of the programming language, and use such semantics to effectively prune the search space. We also identify a condition that ensures such abstract semantics can be used to compute a precise abstraction of the set of values that a program derivable from a given hole in a partial program can produce. These precise abstractions make abstraction-based pruning more effective.

We implement our approach in a tool, Moito, which can tackle synthesis problems defined in the SemGuS framework. Moito can automate interval-based pruning without any a-priori knowledge of the problem domain, and solve synthesis problems that previously required domain-specific, abstraction-based synthesizers—e.g., synthesis of regular expressions, CSV file schema, and imperative programs from examples.

CCS Concepts: • **Theory of computation → Program semantics**; **Abstraction**; • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: program synthesis, abstract interpretation, grammar flow analysis

ACM Reference Format:
Keith J.C. Johnson, Rahul Krishnan, Thomas Reps, and Loris D'Antoni. 2024. Automating Pruning in Top-Down Enumeration for Program Synthesis Problems with Monotonic Semantics. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 304 (October 2024), 27 pages. https://doi.org/10.1145/3689744

1 Introduction

The goal of program synthesis is to find a program that satisfies a given specification, typically given as a logical formula or a set of input-output examples. A simple synthesis technique is

*Both authors contributed equally to the paper.

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

ACM 2475-1421/2024/10-ART304
https://doi.org/10.1145/3689744

top-down enumeration, where one enumerates programs derivable from a regular-tree grammar—i.e., the search space of the synthesis problem—by iteratively expanding *partial programs*—i.e., syntax trees that contain one or more *hole* symbols to be filled by as-yet undetermined sub-trees—using productions in the grammar. Top-down enumeration is effective in practice when one can aggressively prune the search space of enumerated programs. Pruning is done by discarding partial programs that cannot be completed to yield a program that satisfies the given specification.

Domain-Specific Pruning Strategies. Clever pruning strategies have resulted in top-down enumeration approaches that can quickly synthesize imperative programs [So and Oh 2017], regular expressions [Lee et al. 2016], SQL queries [Wang et al. 2017a], and Datalog programs [Si et al. 2018]. However, coming up with the domain-specific insights to allow enumeration to scale is a challenging task, to the extent that such insights are the main contribution of many papers.

Consider the problem of synthesizing a regular expression that accepts the (positive) example string 11. During enumeration, a partial regular expression $0 \cdot \square_R$ can be pruned because no matter what regular expression we replace $\square_R$ with, the string 11 will never be accepted. This simple technique was the main contribution that allowed the tool AlphaRegex [Lee et al. 2016] to synthesize regular expressions of non-trivial size.

Similarly, consider the problem of synthesizing an imperative program that is consistent with a set of input-output examples, and suppose that we have the partial program `x := 0; y := `$\square_E$, where $\square_E$ can be replaced with an expression over the variables x, y and integer constants. This program can be discarded if the synthesis goal requires mapping the input $x = 5$ to output $x = 7$: any instantiation of the assignment to y still results in the input $x = 5$ being mapped to $x = 0$.

Despite the time and care that went into developing these techniques, they cannot be automatically tailored to potentially similar tasks outside of their rigid predefined domains.

Automating Pruning. In this paper, we show that the aforementioned pruning strategies can be "rationally reconstructed" as instances of a common **automated domain-agnostic** framework for pruning partial programs. Given a partial program with holes P, our framework uses abstract interpretation to abstract the semantics of all the possible program completions on a specific input, and obtain, in the form of an interval $[l, u]$, a superset of the possible values that a completed program might compute. When the value we want the synthesized program to compute is not in the interval $[l, u]$, the program P can be pruned. For example, the set of possible regular expressions that $0 \cdot \square_R$ can take on can be overapproximated by the interval $[\emptyset, 0 \cdot (0 \mid 1)^*]$, where a regular expression r is inside this interval if and only if $\emptyset \subseteq L(r) \subseteq L(0 \cdot (0 \mid 1)^*)$. However, the positive example 11 is not accepted by any regular expression in this interval, because $11 \notin L(0 \cdot (0 \mid 1)^*)$, meaning we can prune out $0 \cdot \square_R$. Similarly, for the imperative-program pruning example, the output value of x produced by `x := 0; y := `$\square_E$ on input $x = 5$ can be expressed as (and hence "overapproximated by") the interval $[0, 0]$. Because the desired output 7 is not in this interval, we can prune this partial program.

Our work also identifies sufficient conditions on the semantics of the programs appearing in the search space that allow one to generate the operations needed to compute such abstractions **automatically**. If the semantics of the programs in the search space satisfies monotonicity conditions (that often can be automatically checked), our framework provides for free a *precise* interval-based abstraction. The key insight is that given a monotonically increasing function f and an interval of possible inputs $[l, u]$, the tightest interval that encloses the result of evaluating f on values in $[l, u]$ is $[f(l), f(u)]$, which can be computed automatically by simply evaluating f.

For the regular-expression pruning strategy presented earlier, because concatenation is monotonic (in a sense described later in the paper) given the partial program $0 \cdot \square_R$, and intervals $[0, 0]$ and $[\emptyset, (0 \mid 1)^*]$, representing sets of possible completions of the (partial) regular expressions 0

and $\square_R$, the set of completions of $0 \cdot \square_R$ can be exactly captured by the interval $[0 \cdot \emptyset, 0 \cdot (0 \mid 1)^*]$ $= [\emptyset, 0 \cdot (0 \mid 1)^*]$—i.e., what AlphaRegex would compute using a specialized algorithm.

A similar argument explains the pruning technique for $\mathtt{x} := \emptyset$; $\mathtt{y} := \square_E$—i.e., the semantics of assignments is also monotonic with respect to its argument (in a sense described later).

Phrasing the pruning approaches as interval-based abstract interpretation unlocks a new opportunity for pruning that had not been identified in prior work. Existing approaches assume that holes can produce arbitrary programs that yield arbitrary values in the interval $\top$—i.e., the interval that describes every possible value. Instead, we show how the order under which the semantics is monotonic can be used to automatically compute tighter bounds on the values of such intervals. For example, if a hole can only be completed with numbers in the grammar $N \rightarrow 0 \mid 1 + N$, our approach can prove that any term derived from nonterminal N must be non-negative.

The SemGuS Framework. In practice, it is challenging to build a domain-agnostic synthesizer that can accommodate the diversity of synthesis tasks described above. To achieve such generality, we target problems in the SemGuS format [Kim et al. 2021], a unifying domain-agnostic and solver-agnostic framework for specifying arbitrary synthesis problems. SemGuS is expressive enough to specify synthesis problems involving regular expressions, CSV schemas, bitvectors, and imperative programs. However, to the best of our knowledge, prior SemGuS solvers cannot tackle such synthesis problems with reasonable efficiency because of the challenges of generalizing domain-agnostic synthesis techniques beyond naïve enumeration.

Contributions. Our work makes the following contributions.

- We unify a number of domain-specific pruning approaches into a general *framework for interval-based pruning* (Sections 2 and 3).
- We define a theory of *semantic monotonicity* that provides sufficient conditions under which it is possible to automate precise interval-based pruning (Section 4).
- We combine our abstraction framework with a technique called grammar-flow analysis [Möncke and Wilhelm 1991] to obtain *precise hole abstractions* that can be used to further prune the explored search space (Section 5).
- We provide a technique for automatically *synthesizing orders* under which a semantics is monotonic, thus enabling interval-based pruning (Section 6).
- We implemented our framework in the tool Moito, which supports synthesis problems expressed in SemGuS, a domain-agnostic framework for specifying synthesis problems. Moito can automatically discover interval-based pruning approaches that were hard-coded in existing domain-specific solvers, and use them to outperform a vanilla enumeration on several SemGuS benchmarks (Section 7).

2 Overview

This section illustrates how our framework unifies the pruning techniques used in AlphaRegex [Lee et al. 2016] (for regular expressions) and SIMPL [So and Oh 2017] (for imperative programs). We will further discuss in Section 8 how our framework also unifies some prior approaches for synthesizing SQL queries [Wang et al. 2017a], Datalog programs [Si et al. 2018], and data-processing tasks [Mell et al. 2024].

The tool AlphaRegex synthesizes regular expressions in the following fixed grammar G_α:

$$R ::= \ \mathtt{c} \mid \epsilon \mid \emptyset \mid (R \mid R) \mid (R \cdot R) \mid R^*$$

Examples are given as pairs of strings and Boolean values, denoting whether a string should be accepted or rejected by the regular expression to be synthesized. For example, given the examples

$\mathcal{E}_1^R = \{(1, \mathit{true}), (10, \mathit{true}), (111, \mathit{true}), (0, \mathit{false}), (00, \mathit{false}), (100, \mathit{false})\}$, AlphaRegex might synthesize the regular expression $(1 \cdot (0 \mid 1))^* \cdot 1$, which accepts all non-empty bitstrings where every odd position contains the digit 1.

The tool SIMPL synthesizes imperative programs. For illustrative purposes, we do not consider programs containing loops, and assume that SIMPL targets programs in the following fixed grammar G_I where the only two variables are x and y:

$$S ::= x := E \mid y := E \mid S; S \qquad\qquad E ::= 0 \mid 1 \mid x \mid y \mid E + E \mid E - E$$

Examples are given as pairs of input and output states, where a state is an assignment of values to x and y. For example, given the examples $\mathcal{E}_1^I = \{(\langle x = 4, y = 2\rangle, \langle x = 2, y = 4\rangle), (\langle x = 3, y = 3\rangle, \langle x = 3, y = 3\rangle)\}$, SIMPL might synthesize the imperative program `x := x-y; y := x+y; x := y-x;`, which swaps the values of variables `x` and `y`.

2.1 Top-Down Enumeration and Pruning in AlphaRegex and Simpl

Given a synthesis problem, *top-down enumeration* exhaustively searches over the space of programs in the grammar G, expanding *partial programs* according to the productions of the grammar. A partial program is a tree that can contain *hole* symbols—i.e., unexpanded nonterminals—to be filled by as-yet-undetermined sub-trees. For example, $0 \cdot \square_R$ and $(\square_R \mid \square_R)^*$ are partial programs (regular expressions) that could be further expanded using grammar G_α, and `x := x+`$\square_E$ and `x := `$\square_E$`;` `y := `$\square_E$ are partial (imperative) programs that could be further expanded using grammar G_I.

Blindly enumerating all possible programs is impractical, but through clever pruning strategies, one can mitigate the combinatorial nature of exhaustive enumeration and reach more complex programs deeper in the search space. Given a partial program P, if one can prove that there exists *no* way to turn P into a complete program (i.e., that contains no holes) that is consistent with the given examples $\mathcal{E}$, the partial program P can be pruned. When a partial program P is pruned, none of the potentially infinitely many completions of P will ever be considered by the enumeration!

Pruning in AlphaRegex. Given the examples $\mathcal{E}_1^R = \{(1, \mathit{true}), (10, \mathit{true}), (111, \mathit{true}), (0, \mathit{false}), (00, \mathit{false}), (100, \mathit{false})\}$, AlphaRegex eventually enumerates the partial program $0 \cdot \square_R$. The key observation presented in the AlphaRegex paper is that no matter what program is used to fill $\square_R$, the resulting overall program will only accept strings that start with 0, and thus will never accept the string 1—in particular, the program will be inconsistent with the example $(1, \mathit{true})$. AlphaRegex automates this check by observing that the semantics of regular expressions is such that when we replace $\square_R$ with the regular expression $(0 \mid 1)^*$, we obtain the "most permissive" possible regular expression—i.e., if $0 \cdot (0 \mid 1)^*$ does not accept all the positive examples, no completion of $0 \cdot \square_R$ will. A similar check is performed for negative examples—i.e., if $0 \cdot \emptyset$ does not reject all the negative examples, no completion of $0 \cdot \square_R$ will. Therefore, when either of these two checks fails, the corresponding candidate partial program can be pruned away.

Pruning in SIMPL. Given the examples $\mathcal{E}_1^I = \{(\langle x = 4, y = 2\rangle, \langle x = 2, y = 4\rangle), (\langle x = 3, y = 3\rangle, \langle x = 3, y = 3\rangle)\}$, SIMPL eventually enumerates the partial program `x := 0; y :=` $\square_E$. The key idea in SIMPL is that no matter what sub-tree is used to replace $\square_E$, the resulting program is incorrect because it must produce a final state in which $x = 0$. SIMPL automates this check using interval-based abstract interpretation to overapproximate the set of values any program constructed from $\square_E$ could return. Intuitively, by applying appropriate interval semantics to the program `x := 0; y :=` $[-\infty, \infty]$, we know that the output state can be overapproximated by the abstract state $(x \in [0, 0], y \in [-\infty, \infty]) = [0, 0] \times [-\infty, \infty]$. Because none of the desired output states—e.g., $\langle x = 2, y = 4\rangle$—fall in this set, this partial program can be pruned. For the interval-based

static analysis used in SIMPL, a *manually-defined* interval abstract transformer was created for every construct in G_I.

2.2 A Unifying Framework for Interval-Based Pruning

The pruning approaches adopted by AlphaRegex and SIMPL are both extremely effective, but require domain-specific insights or manually-designed static analyses to compute precise abstractions. While the methods for pruning in these two examples appear to be very different, once we describe their semantics in a logical format (in our case, as Constrained Horn Clauses in the SemGuS framework), it becomes possible to handle these pruning approaches in a unified way. Specifically, both pruning approaches can be explained and generalized as instances of *interval-based abstract interpretation*. Most importantly, we demonstrate that the semantics ascribed to the two grammars G_α and G_I enjoy special properties that allow the appropriate abstract interval-transformers to be created automatically from the user-provided logical semantics.

Interval-Based Pruning in AlphaRegex. Given the partial program $0 \cdot \square_R$, our first observation is that one can think of AlphaRegex as computing a precise interval-based abstraction of the partial program $0 \cdot \square_R$ by interpreting $\square_R$ as the interval $[\emptyset, (0 \mid 1)^*]$—i.e., with the range of all possible strings that a regular expression that fills $\square_R$ could produce. To reformulate this computation in abstract terms: AlphaRegex is computing an abstract semantics as $[\![0 \cdot \square_R]\!]^\sharp(\square_R = [\emptyset, (0 \mid 1)^*]) = [0,0] \cdot^\sharp [\emptyset, (0 \mid 1)^*]$, where $\cdot^\sharp$ is the interval abstract transformer for concatenation. The second key observation is that the abstract value $[0,0] \cdot^\sharp [\emptyset, (0 \mid 1)^*]$ can be computed as $[0 \cdot \emptyset, 0 \cdot (0 \mid 1)^*]$—i.e., the abstract transformer $\cdot^\sharp$ over an interval of regular expressions can be computed by applying the concrete operation $\cdot$ to the left and right bounds of its interval arguments. This last step is actually true for all regular-expression operators! Our final key observation is that this trivial computation of an interval-abstract transformer can always be performed as long as the semantics of the operator under consideration is *monotonic* with respect to the order over which intervals are defined. In this case, because intervals are ordered by language inclusion, we have that if $L(r_1) \subseteq L(r_1')$ and $L(r_2) \subseteq L(r_2')$ then $L(r_1 \cdot r_2) \subseteq L(r_1' \cdot r_2')$—i.e., the semantics of concatenation is monotonically increasing in its arguments. Consequently, the abstract transformer for $\cdot^\sharp$ is $[r_l, r_u] \cdot^\sharp [r_l', r_u'] = [r_l \cdot r_l', r_u \cdot r_u']$. The same argument applies to other regular-expression operators.

The above argument gives us a systematic way to explain how AlphaRegex merely computes an interval abstraction of all possible strings that can be produced by some completion of a partial regular expression. Thus, if we can determine that the operations in the user-defined semantics in the SemGuS format exhibit the monotonicity property above, we automatically get abstract interval transformers for these operations for free.

Interval-Based Pruning for Imperative Programs. It is now easy to see how AlphaRegex and SIMPL are similar. In SIMPL, the abstract semantics of $\square_E$ for a specific input example (let's say $\langle x = 4, y = 2 \rangle$) is expressed as the interval $[-\infty, \infty]$, which intuitively states that on the given input, if we were to run any of the programs with which one can replace $\square_E$ on the input example, we would get an output in the interval $[-\infty, \infty]$. The key point is that although SIMPL provided manually crafted abstract transformers to evaluate the semantics of the partial program for this interval domain, in many cases, such a semantics can be computed automatically, like we did above! In particular, all the operators appearing in the partial program $x := 0; y := \square_E$ are monotonic (in a sense that we will formalize later in the paper), and therefore $[\![x := 0; y := \square_E]\!]^\sharp(x = 4, y = 2, \square_E = [-\infty, \infty])$ can be computed as $[[\![x := 0; y := \square_E]\!](x = 4, y = 2, \square_E = -\infty), [\![x := 0; y := \square_E]\!](x = 4, y = 2, \square_E = \infty)]$.

2.3 Computing Precise Hole Abstractions via Grammar Flow Analysis

We now illustrate how the interval-based framework unlocks another pruning opportunity. Consider the task of synthesizing a regular expression in the following grammar G_{CSV}, which defines regular expressions for describing the format of rows in a CSV file—i.e., comma-separated entries that (for the sake of this example) can contain either alphabetic or numerical characters.

$$\begin{aligned} \mathit{Row} &::= \mathit{Alpha} \mid \mathit{Num} \mid \mathit{Alpha} \cdot , \cdot \mathit{Row} \mid \mathit{Num} \cdot , \cdot \mathit{Row} \\ \mathit{Alpha} &::= a \mid \cdots \mid z \mid (\mathit{Alpha} \cdot \mathit{Alpha}) \mid (\mathit{Alpha} \mid \mathit{Alpha}) \mid (\mathit{Alpha})^* \\ \mathit{Num} &::= 0 \mid \cdots \mid 9 \mid (\mathit{Num} \cdot \mathit{Num}) \mid (\mathit{Num} \mid \mathit{Num}) \mid (\mathit{Num})^* \end{aligned}$$

Suppose we are given the set of examples $\mathcal{E}_{CSV} = \{(\texttt{"303, name"}, \mathit{true})\}$ and eventually enumerate the partial program $\square_{\mathit{Alpha}} \cdot , \cdot \square_{\mathit{Row}}$. If one tries to prune this partial program by replacing $\square_{\mathit{Alpha}}$ and $\square_{\mathit{Row}}$ with the "default" overapproximation $[\emptyset, .^*]$ (where . denotes any character), we would get the following interval to represent the set of possible solutions: $[\emptyset, (.^* \cdot , \cdot .^*)]$, and conclude that the partial program $\square_{\mathit{Alpha}} \cdot , \cdot \square_{\mathit{Row}}$ cannot be pruned. A careful analysis of the nonterminal *Alpha* shows that any program derived from $\square_{\mathit{Alpha}}$ can only match alphabetic strings. Thus, the interval $[\emptyset, (a \mid \cdots \mid z)^*]$ would provide a better abstraction of the semantics of all programs derivable from *Alpha* and allow us to prune the partial program $\square_{\mathit{Alpha}} \cdot , \cdot \square_{\mathit{Row}}$ (because no completion can start with "`303`.")

Our interval-based framework opens up a simple way to compute these more precise abstractions automatically. In particular, once we have a concrete representation of the interval ordering relation, we can use a technique called Grammar Flow Analysis (GFA) [Möncke and Wilhelm 1991].

Specifically, for every nonterminal N we can construct a system of constraints based on the provided grammar and computed abstract semantics, for which the least solution is the tightest interval that overapproximates the semantics of the programs derivable from N. For instance, for the nonterminal *Alpha*, the constraints contain as free variables the interval bounds $[l, u]$ we are looking for (in this case, denoted by regular expressions themselves) and take the following form:

$$\forall v. \big(a \preceq v \preceq a \vee \cdots \vee z \preceq v \preceq z \vee (l \cdot l) \preceq v \preceq (u \cdot u) \vee (l \mid l) \preceq v \preceq (u \mid u) \vee (l)^* \preceq v \preceq (u)^* \Rightarrow l \preceq v \preceq u\big) \quad (1)$$

Assuming that $\preceq$ denotes language inclusion, the assignment $l = \emptyset$ and $u = (a \mid \cdots \mid z)^*$ is a valid solution to Equation (1). We will show in Section 5 how we solve such equations.

3 Top-Down Enumeration and Abstraction-Based Pruning

In this section, we formulate the synthesis problems that we tackle (Section 3.1), formalize a top-down enumeration algorithm (Section 3.2), and show how interval-based abstractions can be used to prune the set of programs enumerated via top-down enumeration (Section 3.3). We illustrate all our techniques using a simple imperative programming language (Figure 1). The extended version of the paper [Johnson et al. 2024a] contains another example for a language of regular expressions.

3.1 Synthesis Problem

In this section, we introduce the terminology used throughout the rest of the paper, and define the scope of our example-based synthesis problem.

Definition 3.1 (Regular tree grammar). A *regular tree grammar* (RTG) is a tuple $G = (\mathcal{N}, \Sigma, S, \delta)$, where $\mathcal{N}$ denotes a finite set of non-terminal symbols; Σ is a ranked alphabet; $S \in \mathcal{N}$ is the start nonterminal; and δ is a finite set of productions $N_0 \to \rho(N_1, \ldots, N_n)$.

For each nonterminal N, we use $\square_N$ to denote a node variable—i.e., a hole—associated with nonterminal N. We define a partial program P to be a tree that may contain holes. Given a partial

program P with a leftmost hole occurrence $\square_N$, we say that a program P' can *be derived in one step* from P if there exists a production $N \to \rho(N_1, \ldots, N_n)$ such that replacing the leftmost hole occurrence $\square_N$ with $\rho(\square_{N_1}, \ldots, \square_{N_n})$ results in P'—denoted by $P \mapsto P'$. We say that a partial program P' is *derived* from another partial program P in zero or more steps, denoted by $P \mapsto^* P'$, if it is in the reflexive and transitive closure of the one-step derivation relation.

Given an RTG G, we say that a partial program P is *generated by a nonterminal* N if it can be generated from $\square_N$—i.e., $\square_N \mapsto^* P'$. Finally, program P is *complete* if it does not contain holes. We use $\mathcal{L}(N)$ to denote the set of complete programs that can be generated by a nonterminal N. We use $\mathcal{L}(G) = \mathcal{L}(S)$ to denote the set of complete programs accepted by the grammar G.

Example 3.2 (Imperative Grammar). Consider the imperative language defined by grammar G_I in Figure 1a. From the partial program `x := 0; y := ` $\square_E$ we can derive in one step the complete program `x := 0; y := x` and the partial program `x := 0; y := (` $\square_E$ `+` $\square_E$`)`.

The following definition shows how to associate a semantics to programs defined by a grammar.

Definition 3.3 (Semantics). Let $G = (\mathcal{N}, \Sigma, S, \delta)$ be a regular-tree grammar with set of nonterminals $\mathcal{N} = \{N_1, \ldots N_k\}$. A *semantics* for G is a pair $(\{[\![\cdot]\!]_{N_1}, \ldots, [\![\cdot]\!]_{N_k}\}, \sigma)$, such that each $[\![t]\!]_{N_i} : t \in \mathcal{L}(N_i) \to I_{N_i} \to O_{N_i}$ is a function symbol for which the interpretation is given by σ, a function that maps each production $M_0 \to \rho(M_1, \ldots, M_n)$ to a set of well-typed first-order implication formulas—i.e., the rules that define the function $[\![t]\!]_{M_0}$—of the following form:

$$\frac{[\![t_1]\!]_{M_1}(x_1) = y_1 \quad \cdots \quad [\![t_n]\!]_{M_n}(x_n) = y_n \qquad \varphi_{=}(x, x_1, \ldots, x_n, y_1, \ldots, y_n) \wedge y = f(x, y_1, \ldots, y_n)}{[\![\rho(t_1, \ldots, t_n)]\!](x) = y} \tag{2}$$

In the formulas, all variables t_i, x_i, y_i are universally quantified. $\varphi_{=}$ is a conjunction of equalities over the variables x, x_i, and y_i; and f is a function.

The interpretation of the semantic function symbols $\{[\![\cdot]\!]_{N_1}, \ldots, [\![\cdot]\!]_{N_k}\}$ is then the least fixed-point solution of the set of first-order formulas $\bigcup_{p \in \delta} \sigma(p)$.

These first-order formulas used to define the semantics are a restricted form of *Constrained Horn Clauses* (CHCs) [Komuravelli et al. 2014] and we will refer to formulas of the above form as CHCs. The above definition of semantics is a restricted fragment of the SemGuS framework, which instead supports CHCs with arbitrary predicates. Our restricted format cannot capture nondeterminism, but allows us to model executable semantics for deterministic programs, which are the focus of our work. In the rest of the paper, to avoid clutter we drop the nonterminal subscripts (unless noted otherwise) and simply write $[\![\cdot]\!]$ to denote all semantic relations. When writing $[\![\cdot]\!]$, we also implicitly assume the corresponding mapping σ is given to us, so we drop it from most definitions.

Example 3.4 (Imperative Semantics). Figure 1d presents an (operational) semantics for programs in the grammar G_I (Figure 1a). The semantics associated with nonterminal E, i.e., $[\![\cdot]\!]_E$, has type $Int \times Int \to Int$, i.e., the programs derived from E map pairs of integer variable values to integers. The semantics $[\![\cdot]\!]_S$ has type $Int \times Int \to Int \times Int$, i.e., a program derived from S maps a pair of values—one for each variable—to a pair of values. In each rule, instead of explicitly presenting the formula $\varphi_{=}$ (as in Eqn. (2)), we introduce unique names to denote all variables that are equal according to $\varphi_{=}$. In the rule Sub, the function appearing in the premise of the CHC would be $f_{Minus}((x, y), v_1, v_2) = v_1 - v_2$.

In the remainder of the paper, we consider synthesis problems in which the program is only required to be correct on a finite set of examples. We are now ready to define the synthesis problems we consider in this paper. Our definition is an instance of the Semantics-Guided Synthesis (SemGuS)

$S ::= x := E \mid y := E \mid S; S$

$E ::= 0 \mid 1 \mid x \mid y \mid E + E \mid E - E$

(a) Grammar G_I

$\{(\langle x = 4, y = 2\rangle, \langle x = 2, y = 4\rangle),$
$(\langle x = 3, y = 3\rangle, \langle x = 3, y = 3\rangle)\}$

(b) Examples $\mathcal{E}_1^I$

```
x := x-y;  y := x+y;  x := y-x;
```

(c) Solution to $\mathcal{E}_1^I$

$$\frac{[\![e]\!](x,y) = o_e \qquad (o_x, o_y) = (o_e, y)}{[\![x := e]\!](x,y) = (o_x, o_y)}\ \text{Assn-}x$$

$$\frac{[\![e]\!](x,y) = o_e \qquad (o_x, o_y) = (x, o_e)}{[\![y := e]\!](x,y) = (o_x, o_y)}\ \text{Assn-}y$$

$$\frac{[\![s_1]\!](x,y) = (x_1, y_1) \quad [\![s_2]\!](x_1,y_1) = (x_2,y_2) \qquad (o_x,o_y) = (x_2,y_2)}{[\![s_1; s_2]\!](x,y) = (o_x,o_y)}\ \text{Seq}$$

$$\frac{v = 0}{[\![0]\!](x,y) = v}\ \text{Zero} \qquad \frac{v = 1}{[\![1]\!](x,y) = v}\ \text{One}$$

$$\frac{v = x}{[\![x]\!](x,y) = v}\ x \qquad \frac{v = y}{[\![y]\!](x,y) = v}\ y$$

$$\frac{[\![e_1]\!](x,y) = v_1 \quad [\![e_2]\!](x,y) = v_2 \quad v = v_1 + v_2}{[\![e_1 + e_2]\!](x,y) = v}\ \text{Add}$$

$$\frac{[\![e_1]\!](x,y) = v_1 \quad [\![e_2]\!](x,y) = v_2 \quad v = v_1 - v_2}{[\![e_1 - e_2]\!](x,y) = v}\ \text{Sub}$$

(d) Semantics

$$\frac{[L,U] = [(-\infty,-\infty),(\infty,\infty)]}{[\![\square_S]\!]^\sharp([L_0,U_0]]) = [L,U]}\ \text{Hole}_S^\sharp \qquad \frac{[l,u] = [-\infty,\infty]}{[\![\square_E]\!]^\sharp([L_0,U_0]) = [l,u]}\ \text{Hole}_E^\sharp$$

$$\frac{[\![e]\!]^\sharp([L_0,U_0]) = [l_e,u_e] \qquad [L,U] = [(l_e, L_0(y)), (u_e, U_0(y))]}{[\![x := e]\!]^\sharp([L_0,U_0]) = [L,U]}\ \text{Assn-x}^\sharp$$

$$\frac{[\![e]\!]^\sharp([L_0,U_0]) = [l_e,u_e] \qquad [L,U] = [(L_0(x), l_e), (U_0(x), u_e)]}{[\![y := e]\!]^\sharp([L_0,U_0]) = [L,U]}\ \text{Assn-y}^\sharp$$

$$\frac{[\![s_1]\!]^\sharp([L_0,U_0]) = [L_1,U_1] \quad [\![s_2]\!]^\sharp([L_1,U_1]) = [L_2,U_2] \qquad [L,U] = [L_2,U_2]}{[\![s_1; s_2]\!]^\sharp([L_0,U_0]) = [L,U]}\ \text{Seq}^\sharp$$

$$\frac{[l,u] = [0,0]}{[\![0]\!]^\sharp([L,U]) = [l,u]}\ \text{Zero}^\sharp \qquad \frac{[l,u] = [1,1]}{[\![1]\!]^\sharp([L,U]) = [l,u]}\ \text{One}^\sharp$$

$$\frac{[l,u] = [l_x,u_x]}{[\![x]\!]^\sharp([(l_x,l_y),(u_x,u_y)]) = [l,u]}\ x^\sharp \qquad \frac{[l,u] = [l_y,u_y]}{[\![y]\!]^\sharp([(l_x,l_y),(u_x,u_y)]) = [l,u]}\ y^\sharp$$

$$\frac{[\![e_1]\!]^\sharp([L,U]) = [l_1,u_1] \quad [\![e_2]\!]^\sharp([L,U]) = [l_2,u_2] \quad [l,u] = [l_1 + l_2, u_1 + u_2]}{[\![e_1 + e_2]\!]^\sharp([L,U] = [l,u]}\ \text{Add}^\sharp$$

$$\frac{[\![e_1]\!]^\sharp([L,U]) = [l_1,u_1] \quad [\![e_2]\!]^\sharp([L,U]) = [l_2,u_2] \quad [l,u] = [l_1 - u_2, u_1 - l_2]}{[\![e_1 - e_2]\!]^\sharp([L,U] = [l,u]}\ \text{Sub}^\sharp$$

(e) Interval abstract semantics

Fig. 1. An example-based SemGuS problem for imperative programs (Figures 1a, 1b and 1d), and a sound abstract semantics for the grammar G_I (Figure 1e). We use lowercase l, u variables to denote integers and uppercase variables L, U to denote pairs of integers, where $L(x)$ and $U(x)$ correspond to the first element of the pair, and $L(y), U(y)$ the second element.

framework for describing synthesis problem, where semantics are provided using CHCs; thus, we use the same name.

Definition 3.5 (Example-based SemGuS Problem). An *example-based SemGuS problem* is a triple $(G, [\![\cdot]\!], \mathcal{E})$ where G is a regular-tree grammar specifying the search space, $[\![\cdot]\!]$ is a semantics that uses CHCs to ascribe meaning to the trees/programs generated by the grammar, and $\mathcal{E} = \{(i_1, o_1), \ldots, (i_n, o_n)\}$ is a set of input-output examples.

A program P in the language $\mathcal{L}(G)$ described by the grammar G is a *solution* to the synthesis problem if it is consistent with all the examples, i.e., for every $1 \leq j \leq n$, we have $[\![P]\!](i_j) = o_j$, which we denote by $P \vdash \mathcal{E}$.

The grammar G_I, its semantics, and the set of examples $\mathcal{E}^I$ shown in Figure 1 form an example-based SemGuS problem.

3.2 Top-Down Enumeration

In this section, we describe the standard top-down enumeration approach to synthesis that we will apply to example-based SemGuS problems. To find a solution to an example-based synthesis

problem $(G, [\![\cdot]\!], \mathcal{E})$, top-down enumeration enumerates trees $t \in \mathcal{L}(G)$ to find one that is consistent with all the examples, i.e., $t \vdash \mathcal{E}$.

Given a synthesis problem, Algorithm 1 enumerates partial programs by exploring their one-step derivations until a complete program is found that is a solution. The algorithm starts with the smallest partial program, a single hole $\square_S$, corresponding to start nonterminal S (Line 2), and iterates over all partial and total programs in the grammar using a priority queue Q (Line 3). The priority for the queue controls the order in which programs are enumerated (e.g., by size, by depth, etc.).

Algorithm 1 Top-down enumeration w. pruning

```
1:  function Search(G, [[·]], E)
2:      Q ← {□_S}
3:      while Q ≠ ∅ do
4:          P ← Dequeue(Q)
5:          if IsComplete(P) ∧ P ⊢ E then return P
6:          else
7:              for all P' s.t. P ↦ P' do
8:                  if ¬Prune(P', E) then Enqueue(Q, P')
9:  function Prune(P', E)
10:     for all (i_k, o_k) ∈ E do
11:         if o_k ∉ [[P']]^♯(i_k) then return True
12:     return False
```

If the program P considered at a certain iteration is complete—i.e., it has no holes—the program is evaluated against the synthesis constraint, and returned as the answer if it satisfies all input-output examples in $\mathcal{E}$ (Line 5). If the considered program P is partial—i.e., contains a sequence of holes—the leftmost hole is successively replaced with each possible production to obtain all the programs P' derivable in one step from P (Line 7).

The algorithm then queries a Prune function (whose implementation will be explained in Section 3.3) that determines if we can soundly discard partial program P', which would prune away the potentially-infinite set of concrete programs derivable from P'. The following definition states that a function Prune is sound if it only prunes partial programs that cannot derive valid complete solutions to the synthesis problem.

Definition 3.6. We say that the function Prune is *sound* if for every partial program P', if $\textsc{Prune}(P', \mathcal{E}) = \textsc{True}$ then there exists no complete program P'' such that (*i*) P'' is derivable from P', and (*ii*) P'' is consistent with the examples—i.e., $\neg\exists P''.P' \mapsto^* P'' \wedge P'' \vdash \mathcal{E}$.

If the pruning function is sound and the priority queue returns the smallest program with respect to a well-founded order over programs, the algorithm (under reasonable conditions) finds a solution if one exists [Solar-Lezama 2013].

3.3 Pruning using Abstractions

While there are many ways to design pruning functions, in this paper we are only interested in pruning by considering *abstract representations* of certain sets of values. For a candidate partial program P', we wish to capture the sets of possible outputs of the set of possible completions of P' when these complete programs are run on input i_k from an input-output example (i_k, o_k) in $\mathcal{E}$.

Later, we denote such a value-set by $[\![P']\!]^\sharp(i_k)$, which represents an overapproximation of the set of possible states that any program P derived from candidate P' can produce, given input i_k. To define $[\![P']\!]^\sharp$ formally, we need to introduce some terminology from abstract interpretation. In this paper, we focus on interval-based abstractions—i.e., a set of elements from a set Y is abstracted using a pair/interval $[y_1, y_2]$ of elements to denote the boundaries of the set.

Definition 3.7 (Interval abstract domain). An *interval abstract domain* for a set Y is a tuple $(Y \times Y, \preceq, \top, \bot)$, where $\preceq$ is a partial order over the elements of Y. A concrete element $y \in Y$ is in abstract interval $[y_l, y_u]$ if $y_l \preceq y \preceq y_u$. We also define the order $\sqsubseteq$ on $Y \times Y$ as $[y_1, y_2] \sqsubseteq [y_1', y_2']$ iff $y_1' \preceq y_1$ and $y_2 \preceq y_2'$. By definition, $\bot \sqsubseteq [y_1, y_2] \sqsubseteq \top$ for any $[y_1, y_2]$.

To define the value computed by $[\![P']\!]^\sharp$, we introduce a notion of abstract semantics that assigns meanings to both total and partial programs, and lifts the semantics $[\![\cdot]\!]$ to operate over intervals.

We use $G_{int} = (\mathcal{N}, \Sigma \cup \{\square_{N_1}, \ldots, \square_{N_k}\}, S, \delta \cup \{N_i \leftarrow \square_{N_i} \mid 1 \leq i \leq k\})$ to denote the same grammar as G but where each nonterminal can derive a hole, with the intention of defining the semantics over an interval domain. Note that $\mathcal{L}(G_{int})$ contains all the partial programs derivable from G. (We will show in Section 4.2 how such domains can be provided.)

Definition 3.8 (Interval Abstract Semantics). Let $G = (\mathcal{N}, \Sigma, S, a, \delta)$ be a regular-tree grammar with a set of non-terminals $\mathcal{N} = \{N_1, \ldots N_k\}$, and let $(\{[\![\cdot]\!]_{N_1}, \ldots, [\![\cdot]\!]_{N_k}\}, \sigma)$ be a semantics as defined in Definition 3.3.

An *interval semantics* $(\{[\![\cdot]\!]^\sharp_{N_1}, \ldots, [\![\cdot]\!]^\sharp_{N_k}\}, \sigma^\sharp)$ for G_{int}—i.e., the grammar that augments G with holes, representing the language of partial programs—is a semantics defined over interval types. Each type of $[\![\cdot]\!]^\sharp_{N_i}$ is lifted to the corresponding interval type—i.e., if $[\![\cdot]\!]_{N_i} : I_{N_i} \to O_{N_i}$, then $[\![\cdot]\!]^\sharp_{N_i} : I_{N_i} \times I_{N_i} \to O_{N_i} \times O_{N_i}$. We assume that each type T has an associated order $\preceq_T$.

The interval semantics for G_{int} is a *sound interval abstract semantics* for G if every (potentially partial) program $P \in \mathcal{L}(G_{int})$ maps an input interval to an output interval that overapproximates the set of possible outputs that any program P' derivable from P would produce on the same input:

$$\forall P \in \mathcal{L}(G_{int}), \forall [l, u], \forall l \preceq x \preceq u, \forall P' \in \mathcal{L}(G), P \mapsto^* P' \Rightarrow [\![P']\!]_S(x) \in [\![P]\!]^\sharp_S([l, u])$$

We will discuss in Section 4 how an abstract semantics can be defined inductively in practice. Again, to avoid notational clutter we typically write $[\![\cdot]\!]^\sharp$ to denote an abstract semantics.

Example 3.9. The semantics defined in Figure 1e is a sound interval abstract semantics for G_I. The abstract semantics is defined over intervals of (x, y)-pairs: $[(x_l, y_l), (x_u, y_u)]$. The abstract semantics for nonterminal E has type $[\![\cdot]\!]^\sharp_E : (Int \times Int) \times (Int \times Int) \to Int \times Int$. The abstract semantics for $[\![\cdot]\!]^\sharp_S$ has type $(Int \times Int) \times (Int \times Int) \to (Int \times Int) \times (Int \times Int)$. The partial order on (x, y)-pairs is the pairwise order $(x_1, y_1) \preceq (x_2, y_2)$ iff $x_1 \leq x_2$ and $y_1 \leq y_2$. An interval's type is a pair of the types of the interval's components. The abstract semantics of a hole is the widest possible: $[\![\square_S]\!]^\sharp([L, U]) = [(-\infty, -\infty), (\infty, \infty)]$, and $[\![\square_E]\!]^\sharp([L, U]) = [-\infty, \infty]$. We assume this instantiation of the abstract semantics of holes for now, but we revisit it in Section 5.

To summarize, for a partial program P, if for any of the examples (i_k, o_k) in $\mathcal{E}$, the output o_k is not in $[\![P]\!]^\sharp(i_k) = [l_o, u_o]$—i.e., it is not the case that $l_o \preceq_O o_k \preceq_O u_o$—the definition of a sound abstract semantics tells us that P can be *pruned* from further consideration: $[\![P]\!]^\sharp(i_k)$ overapproximates the outputs of all possible completions of P, and there only needs to be one failure among all the input-output examples to determine that no completion of P could be a correct answer.

Theorem 3.10 (Sound Abstract Semantics and Pruning). *If $[\![\cdot]\!]^\sharp$ is a* sound interval abstract semantics *for G, the function Prune described in Algorithm 1 (lines 9-12) is sound.*

The following example illustrates how the algorithm can prune using interval abstract semantics.

Example 3.11 (Interval-Based Pruning). We now show how to compute the interval abstraction for $[\![\texttt{x := 0; y := } \square_E]\!]^\sharp([(4, 2), (4, 2)])$ for the input-output example $(\langle x = 4, y = 2\rangle, \langle x = 2, y = 4\rangle)$ from our abstract semantics in Figure 1e. We have $[\![0]\!]^\sharp([(4, 2), (4, 2)]) = [0, 0]$ and $[\![\texttt{x := 0}]\!]^\sharp([(4, 2)]) = [(0, 2), (0, 2)]$, as defined in the Zero$^\sharp$ and Ass-x$^\sharp$ rules, respectively. The hole $\square_E$ has abstract semantics $[\![\square_E]\!]^\sharp([(0, 2)]) = [-\infty, \infty]$ (rule Hole$^\sharp_E$), which allows us to define $[\![\texttt{y := } \square_E]\!]^\sharp([(0, 2)]) = [(0, -\infty), (0, \infty)]$ (rule Ass-y$^\sharp$). If we combine these results using the Seq$^\sharp$

rule, we get $[\![\texttt{x := 0; y := }\square_E]\!]^\sharp([(4,2)]) = [(0,\infty),(0,-\infty)]$. Because the expected output $(2,4)$ is not in this interval, we can prune this candidate partial program.

4 Automated Construction of Precise Interval Abstractions

To perform pruning in practice, we need to have an abstract semantics in hand to run the abstraction-based synthesis procedure. Typically, designing precise abstract semantics (even for interval abstract domains) is the topic of complex research papers, and bugs are often found in tools that use interval abstract interpretation [Kalita et al. 2022]. In this section, we show our main result: if the concrete semantics is monotonic (in a sense that we define formally), a corresponding sound (and precise) interval abstract semantics can be generated automatically from the concrete semantics. For simplicity, in the rest of this section we assume that every production has only one associated CHC, and describe the points in which modifications need to be made to accommodate multiple CHCs.

4.1 Abstract Semantics of Monotonic Functions

In our running example in Figure 1, the concrete and abstract semantics looked very similar to each other. Consider, for example, the concrete and abstract semantics for Union in Figure 5 in the extended version of the paper [Johnson et al. 2024a]. On the input s, the concrete semantics computes the two matrices $A_1 = [\![r_1]\!](s)$ and $A_2 = [\![r_2]\!](s)$ for the two subexpressions r_1 and r_2, and outputs the matrix $A = A_1 + A_2$. Similarly, the abstract semantics computes two intervals of matrices $[L_1, U_1]$ and $[L_2, U_2]$ for the two subexpressions r_1 and r_2, and outputs the interval of matrices $[L, U]$ computed as $[L_1 + L_2, U_1 + U_2]$.

In this example, the two elements of the interval of matrices can be computed by taking the pairwise operation that was applied to the concrete semantics—i.e., the sum of the semantics of the subexpressions. In other words, the left and right bounds for the interval representing the abstract semantics for this particular production are precisely the concrete semantics applied to the left and right bounds of the input intervals, respectively. This pattern applies to all the rules except $\text{Sub}^\sharp$ in Figure 1, where one can still do something analogous. For the subtraction operator, given an interval of matrices $[l_i, u_i]$ for the abstract semantics of subexpression e_i, the abstract semantics of $e_1 - e_2$ can be defined as $[l_1 - u_2, u_1 - l_2]$

However, this recipe may not always result in a sound interval abstract semantics. Consider a language involving a function $[\![f]\!](x) = x^2$ that outputs the square of its input: if we wrote $[\![f]\!]^\sharp([l,u]) = [l^2, u^2]$, then we would have $[\![f]\!]^\sharp([-2,2]) = [4,4]$, which is not a sound abstract semantics—i.e., 0 is in the interval $[-2,2]$, but the corresponding output $f(0) = 0$ is not in the interval $[4,4]$.

Why does the endpoint recipe work for some functions, but not for others? The relevant property is the *monotonicity* of the function f appearing in the CHC that defines the concrete semantics (with respect to some order). In this section, we define what it means for the semantics to exhibit monotonicity, and demonstrate how to automatically check monotonicity of the semantics.

Definition 4.1 (Monotonicity). Consider a function $f : Y_0 \times Y_1 \times \ldots \times Y_n \to Y$, where $Y_0, Y_1, \cdots, Y_n$, and Y are sets that are partially ordered under $\preceq_0, \preceq_1, \cdots, \preceq_n$, and $\preceq$, respectively. For each argument i, we define the *freezing function* $f_i[\vec{c}] : Y_i \to Y$ produced by fixing a vector of constants $\vec{c}$ for the arguments aside from y_i—i.e., $f_i[\vec{c}](y_i) = f(c_0, c_1, \ldots c_{i-1}, y_i, c_{i+1}, \ldots c_n)$.

We say that f is *monotonically increasing* ($\uparrow$) in its i^{th} argument if for every $\vec{c}$, the output of the freezing function f_i *increases* when the i^{th} input *increases*: $\forall y_1, y_2 \in Y_i, y_1 \preceq_i y_2 \implies f_i[\vec{c}](y_1) \preceq f_i[\vec{c}](y_2)$.

Likewise, f is *monotonically decreasing* ($\downarrow$) in its i^{th} argument if for every $\vec{c}$, the output of the freezing function f_i *decreases* when the i^{th} input *increases*: $\forall y_1, y_2 \in Y_i, y_1 \preceq_i y_2 \implies f_i[\vec{c}](y_2) \preceq f_i[\vec{c}](y_1)$.

If for every argument i, f is monotonically increasing or decreasing in the i^{th} argument, we say that f is *monotone*.

Before continuing, we observe that if all the order relations $\preceq_i$ and functions in the semantics are expressed in a decidable theory, the problem of checking whether a function is monotone can be solved using a constraint solver—i.e., by checking if the constraints in Definition 4.1 hold.

Example 4.2 (Monotonic Functions). Recall the semantics defined in Figure 1d, and the fact that CHCs as defined in Definition 3.3 contain a premise of the form $y = f(x, y_1, \ldots, y_n)$. In this example, we assume that all integer variables are ordered with respect to the numerical order $\leq$, and pairs of integers are ordered with respect to their pairwise integer order—i.e., $(x, y) \preceq (x', y')$ iff $x \leq x'$ and $y \leq y'$. We now discuss some of the rules from Figure 1d, and whether they have a premise that uses a monotonic function.

The function $f_{\text{Add}}((x, y), v_1, v_2) = v_1 + v_2$, which defines the semantics of the Add rule in Figure 1d, is monotonically increasing in the last two arguments, whereas the function $f_{Sub}((x, y), v_1, v_2) = v_1 - v_2$ is monotonically increasing in its second argument, and monotonically decreasing in its third argument. Both of these functions are monotonically increasing *and* decreasing in their first input argument, (x, y), which only happens when the function output does not depend on that argument.

The function $f_{ITE}((x, y), b, s_1, s_2) = (\text{if } b \text{ then } s_1 \text{ else } s_2)$ that defines the semantics of an if-then-else operator is monotonically increasing with respect to its first, third, and fourth arguments, but not monotonic with respect to its second argument b (assuming the order *false* $\preceq$ *true*).

Monotonicity is the key property that enables our interval-based pruning approach, as formulated in the following theorem. If the function appearing in the CHC defining the semantics of a production is monotonic with respect to an appropriate set of orders, the "endpoint recipe" provides a sound interval abstract semantics for intervals defined over the same orders under which the function is monotonic. The following theorem tells us how to automatically abstract the semantics of a monotone function regardless of the direction of the monotonicity.

Theorem 4.3 (Abstraction of Monotonic Functions). *Let $f : Y_0 \times Y_1 \times \ldots Y_n \to Y$ be a monotone function where $\preceq_i, \preceq$ are orders associated with Y_i, Y, and $m \in \{\uparrow, \downarrow\}^{n+1}$ is a vector such that $m_k = \uparrow$ (resp. $m_k = \downarrow$) if f is monotonically increasing (resp. decreasing) in its k^{th} argument.*

We denote the endpoint extension *of f to be a function $\hat{f} : (Y_0 \times Y_0) \times (Y_1 \times Y_1) \times \ldots (Y_n \times Y_n) \to Y \times Y$ defined as follows: $\hat{f}(\ldots, \emptyset, \ldots) = \emptyset$; if $L_\uparrow(l, u) = U_\downarrow(l, u) = l$ and $L_\downarrow(l, u) = U_\uparrow(l, u) = u$ then:*

$$\hat{f}([l_0, u_0], \ldots, [l_n, u_n]) = [f(L_{m_0}(l_0, u_0), \ldots, L_{m_n}(l_n, u_n)), f(U_{m_0}(l_0, u_0), \ldots, U_{m_n}(l_n, u_n))]$$

Then $\hat{f}$ is a sound interval abstraction of f in the following sense:

$$\forall [l_i, u_i], l_i \preceq_i x_i \preceq_i u_i, \qquad \hat{f}([l_0, u_0], \ldots, [l_n, u_n]) = [l, u] \Rightarrow l \preceq f(x_0, \ldots, x_n) \preceq u$$

Furthermore, $\hat{f}$ is the most precise abstraction for f in the following sense: if $\hat{f}([l_0, u_0], \ldots, [l_n, u_n]) = [l, u]$, there exist $x_0^l, \ldots, x_n^l$ and $x_0^u, \ldots, x_n^u$, such that for every i, $l_i \preceq_i x_i^l \preceq_i u_i$ and $l_i \preceq_i x_i^u \preceq_i u_i$, and $f(x_0^l, \ldots, x_n^l) = l$ and $f(x_0^u, \ldots, x_n^u) = u$.

Example 4.4 (Endpoint Extension). Recall the functions from Example 4.2. The endpoint extension of $f_{Plus}((x, y), v_1, v_2) = v_1 + v_2$ is the function $\hat{f}_{Plus}([L, U], [l_1, u_1], [l_2, u_2]) =$

$[f_{Plus}(L, l_1, l_2), f_{Plus}(L, u_1, u_2)] = [l_1 + l_2, u_1 + u_2]$ (because the function is monotonically increasing and decreasing in (x, y), we can choose either L or U as the first argument). The endpoint extension of $f_{Minus}((x, y), v_1, v_2) = v_1 - v_2$ is the function $\hat{f}_{Minus}([L, U], [l_1, u_1], [l_2, u_2]) = [f_{Minus}(L, l_1, u_2), f_{Minus}(L, u_1, l_2)] = [l_1 - u_2, u_1 - l_2]$ because f is monotonically increasing in its second argument and monotonically decreasing in the third argument. For example, for any $[L, U]$, we have $\hat{f}_{Minus}([L, U], [6, 7], [1, 2]) = [6 - 2, 7 - 1] = [4, 6]$, which is the most-precise interval.

4.2 Automatically Generating Abstract Semantics

Now that we have connected monotone functions to interval abstractions, we have a way to generate abstract semantics for all programs in a grammar for free, as long as the functions involved are monotone! For all CHCs defining the concrete semantics of constructs in our grammar, if the function f appearing in the CHC can be proven monotone, we can automatically generate an associated precise interval abstract semantics by defining a new CHC where the function in the premise is simply $\hat{f}$. When the functions are not monotone, we can conservatively define the semantics of a production to return $\top$.[1]

This observation reduces the problem of constructing the abstract semantics to that of determining the monotonicity of semantic constructs. If we can automatically prove the monotonicity of these constructs, then we can obtain these abstract semantics in an automated fashion for free! The following definition tells us how to extract abstract semantics from the original program semantics.

Definition 4.5 (CHC Interval Abstraction). Consider a CHC Op as in Definition 3.3:

$$\frac{[\![t_1]\!]_{M_1}(x_1) = y_1 \ \cdots \ [\![t_n]\!]_{M_n}(x_n) = y_n \quad \varphi_{=}(x, x_1, \ldots, x_n, y_1, \ldots, y_n) \wedge y = f(x, y_1, \ldots, y_n)}{[\![\rho(t_1, \ldots, t_n)]\!](x) = y} \ \text{Op}$$

If f is monotone, let $f^\sharp$ be defined as $\hat{f}$ (as in Theorem 4.3), otherwise define it as $f^\sharp(y_1, \ldots, y_n) = \top$. We define the *interval abstract semantics* CHC $Op^\sharp$ as follows (all types are lifted to intervals):

$$\frac{[\![t_1]\!]^\sharp_{M_1}(x_1) = y_1 \ \cdots \ [\![t_n]\!]^\sharp_{M_n}(x_n) = y_n \quad \varphi_{=}(x, x_1, \ldots, x_n, y_1, \ldots, y_n) \wedge y = f^\sharp(x, y_1, \ldots, y_n)}{[\![\rho(t_1, \ldots, t_n)]\!]^\sharp(x) = y} \ Op^\sharp$$

To use our pruning algorithm, our abstract semantics also needs to assign semantics to holes. The following condition connects the abstract semantics of individual holes to the conditions required for an abstract semantics to be sound (Definition 3.8).

Definition 4.6 (Sound Abstract Semantics for a Hole). Consider the CHC $Hole_N$ assigning an abstract semantics to a hole corresponding to a nonterminal N of some grammar G:

$$\frac{[l, u] = f^\sharp([l_1, u_1])}{[\![\square_N]\!]^\sharp_N([l_1, u_1]) = [l, u]} \ Hole_N$$

We say that $Hole_N$ is a *sound hole abstract semantics* for $\square_N$ if the following holds:

$$\forall [l_1, u_1], \forall l_1 \preceq_I x \preceq_I u_1, \forall P \in \mathcal{L}(N), [\![P]\!]_N(x) \in [\![\square_N]\!]^\sharp_N([l_1, u_1])$$

In the above definition, setting $[l, u]$ to $\top$ is always a safe way to define a sound hole abstract semantics (as done in our abstract semantics in Figure 1d). This choice is also taken by the tools that our framework subsumes and that we described in Section 2). In Section 5, we will show an algorithm for computing more precise abstractions for holes than the $\top$ ones.

[1] There exist tools that can synthesize abstract semantics for arbitrary operators but require human input [Kalita et al. 2022].

We are now ready to define an interval abstract semantics that is guaranteed to be sound and therefore safe for pruning.[2]

Theorem 4.7 (Soundness of Endpoint Interval Semantics). *Let $G = (\mathcal{N}, \Sigma, S, \delta)$ be a regular-tree grammar with set of non-terminals $\mathcal{N} = \{N_1, \ldots N_k\}$ and $(\{[\![\cdot]\!]_{N_1}, \ldots, [\![\cdot]\!]_{N_k}\}, \sigma)$ a semantics for G.*

Let $(\{[\![\cdot]\!]^\sharp_{N_1}, \ldots, [\![\cdot]\!]^\sharp_{N_k}\}, \sigma^\sharp)$ be the semantics defined as follows:

- *for every production $p \in \delta$, then $\sigma^\sharp(p) = \{C^\sharp \mid C \in \sigma(p)\}$ (Definition 4.5);*
- *for every nonterminal $N \in \mathcal{N}$, then $\sigma^\sharp(N \leftarrow \square_N) = \{Hole_N\}$ where $Hole_N$ is a sound hole abstract semantics (Definition 4.6).*

Then the semantics $(\{[\![\cdot]\!]^\sharp_{N_1}, \ldots, [\![\cdot]\!]^\sharp_{N_k}\}, \sigma^\sharp)$ is a sound interval abstract semantics *for G.*

4.3 Extending to Nearly-Monotonic Semantics

Recall that in Example 4.2, the function $f_{ITE}((x, y), b, s_1, s_2) = (\text{if } b \text{ then } s_1 \text{ else } s_2)$ was monotonic with respect to all its arguments, except the conditional guard b. This section shows how one can easily modify our framework to generate an abstract semantics that is more precise than $\top$ in situations where a function is not monotonic in some of its arguments.

Partial Hole Filling. When enumerating children of an if-then-else production, one will have three holes to fill: a Boolean guard b, and the two branches. If the synthesis algorithm deterministically filled holes left-to-right (as done in our implementation and many other synthesizers), then the Boolean guard would be filled first. When the conditional guard is filled concretely with a value b_c, the function f_{ITE} can be partially evaluated to obtain a new function $g_{ITE}((x, y), s_1, s_2) = (\text{if } b_c \text{ then } s_1 \text{ else } s_2)$ that is now monotone in all its arguments! So, even without modifying our analysis, at this point our algorithm can compute a precise abstract transformer to decide whether the partial program (where the guard has been filled) can be pruned.

Enumerating Finite Domains. In the example above, because the variable b can only have two values, true and false, we can actually attempt to compute a non-trivial abstract semantics even when the Boolean guard has not yet been filled. Specifically, we can compute an interval overapproximation of $f_{ITE}((x, y), b, s_1, s_2)$ by taking the join of the abstractions of the two partially evaluated functions $f_{ITE}((x, y), \text{true}, s_1, s_2)$ and $f_{ITE}((x, y), \text{false}, s_1, s_2)$. More formally, $\hat{f}_{ITE}([L, U], [\text{false}, \text{true}], [l_{s_1}, u_{s_1}], [l_{s_2}, u_{s_2}]) = [f_{ITE}(L, \text{false}, l_{s_1}, l_{s_2}), f_{ITE}(U, \text{false}, u_{s_1}, u_{s_2})] \sqcup [f_{ITE}(L, \text{true}, l_{s_1}, l_{s_2}), f_{ITE}(U, \text{true}, u_{s_1}, u_{s_2})]$. This idea can be generalized to any setting in which a variable can only assume finitely many values by simply taking the join over all finite instantiations of that variable. We present a complete formalization in the extended version of the paper [Johnson et al. 2024a].

5 Computing Precise Hole Abstractions via Grammar Flow Analysis

Theorem 4.7 gives us a recipe for automatically generating interval abstract semantics for productions in the original grammar. In this section, we assume access to such an interval abstract semantics. However, the technique as presented so far does not give us a direct way to compute precise sound hole abstractions even when the semantics is monotonic (Definition 4.6). Assigning

[2] As mentioned early, we assume that every production is associated with exactly one CHC. When multiple CHCs are associated with a production, the abstract semantics has to take the join of the intervals computed by all the CHCs to take into account all possible semantics.

the interval $\top$ to the semantics of a hole always gives us a sound hole abstraction. However, formulating the problem in the SemGuS framework allows us to define a more precise abstraction that can be often automatically determined.

Example 5.1 (A Precise Hole Abstraction). Consider a setting in which someone wants to synthesize a program without subtraction, and provides a simplified version G_I^+ of the grammar G_I where the productions for nonterminal E are as follows (i.e., the production $E - E$ has been removed):

$$E ::= 0 \mid 1 \mid x \mid y \mid E + E$$

We assume the semantics is identical to the one in Figure 1d. This grammar can only produce terms that, when evaluated on non-negative inputs for x and y, produce numbers that are greater than or equal to one of x, y and 0. The following hole abstraction captures this idea and is more precise than the one outputting $\top = [-\infty, \infty]$:

$$\frac{[l,u] = [(\text{if } l_x, l_y \geq 0 \text{ then } 0 \text{ else } -\infty), \infty]}{[\![\square_E]\!]_E^\sharp([(l_x,l_y),(u_x,u_y)]) = [l,u]}\ \mathit{Hole}_E$$

For example, for the input $([1,3],[2,5])$ this abstract semantics will produce the interval $[0,\infty]$, which is much more precise than $[-\infty,\infty]$.

Solving for One Value at a Time. While the previous example showed a very precise abstract semantics for the hole (in fact, the most precise), it is challenging to come up with complex expressions automatically, like "if $l_x, l_y \geq 0$ then 0 else $-\infty$," and to guarantee that they are indeed precise abstractions, for all possible values in $(l_x, l_y), (u_x, u_y)$—i.e., the problem is an expression-synthesis problem [Alur et al. 2013]. However, if a specific input value $c = [(1,2),(3,5)]$ is provided, the constraints posed by Definition 4.6, become simpler.

$$\forall (x,y) \in c, \forall P \in \mathcal{L}(E), [\![P]\!]_E(x,y) \in \gamma([\![\square_E]\!]_E^\sharp(c)) \tag{3}$$

Intuitively, we want $[\![\square_E]\!]_E^\sharp(c)$ to overapproximate the set of outputs any program $P \in \mathcal{L}(E)$ can produce on *the specific input* of c. The problem of finding a solution $[l,u]$ to Equation (3) can be phrased in an abstract-interpretation framework called Grammar-Flow Analysis (GFA) [Möncke and Wilhelm 1991]. In GFA, this constraint can be stated using the following equations ($\sqcup$ denotes the join/union of two intervals).

$$[\![\square_E]\!]^\sharp(c) \supseteq [\![0]\!]^\sharp(c) \sqcup [\![1]\!]^\sharp(c) \sqcup [\![x]\!]^\sharp(c) \sqcup [\![y]\!]^\sharp(c) \sqcup [\![\square_E + \square_E]\!]^\sharp(c) \tag{4}$$

Intuitively, the constraints state that the abstract semantics of all the possible programs derivable from E should be included in the semantics of $\square_E$.

Note how the constraints are defined over the abstract semantics of the productions of that nonterminal, so we can directly apply our precise intervals based on our analysis from Section 4.2. Because we are looking for a solution to $\gamma([\![\square_N]\!]_N^\sharp(c))$ in the form of an interval $[l,u]$, we can rewrite Equation (4) as follows (where all abstract semantics have been rewritten according to their semantic rules):

$$[l,u] \supseteq [0,0] \sqcup [1,1] \sqcup [1,3] \sqcup [2,5] \sqcup [l+l, u+u] \tag{5}$$

By unrolling the definitions, the constraint in Equation (5) can be rewritten as follows:

$$\forall v, (0 \leq v \leq 0) \vee (1 \leq v \leq 1) \vee (1 \leq v \leq 3) \vee (2 \leq v \leq 5) \vee (l+l \leq v \leq u+u) \Rightarrow (l \leq v \leq u) \tag{6}$$

As a constraint on l and u, Equation (6) holds for $l \leq 0$ and $u = \infty$. The *tightest* solution is $l = 0$ and $u = \infty$, and thus the answer we seek is the interval $[0,\infty]$.

This procedure for computing precise hole abstractions can be lifted to any synthesis problem in the SemGuS format. Specifically, for every nonterminal N in the grammar G, we want to construct

a function $[\![\square_N]\!]^\sharp_N$ that, on an input x, returns a precise hole abstraction in the form of an interval $[l_N, u_N]$. Because a semantics defined using SemGuS is represented inductively, we are able to reason about each production locally, which results in the following system of constraints:

Definition 5.2 (Interval GFA). Given a grammar G with interval abstract semantics $[\![\cdot]\!]^\sharp$, as in Definition 4.5, *interval grammar flow analysis* is the problem of computing $[\![\square_N]\!]^\sharp_N(x)$ for each nonterminal $N \in G$, which is defined by the following system of constraints:

$$[\![\square_N]\!]^\sharp_N(x) \sqsupseteq \bigsqcup \{[\![p]\!]^\sharp(x) \mid (N \to p) \in G\} \qquad \text{for all } N \in G. \tag{7}$$

Our goal is to solve the GFA constraints to obtain the tightest possible intervals $[\![\square_N]\!]^\sharp_N(x) = [l_N, u_N]$. To avoid dealing with SMT maximization objectives (note that the formulas generated by GFA have quantifiers), our algorithm rewrites Equation (7) as a first-order logical constraint for each nonterminal N, where the bounds l_N, u_N are free variables to be solved for:

$$\forall v. \bigvee_{\{p \mid (N \to p) \in G\}} \left(v \in [\![p]\!]^\sharp(x)\right) \Rightarrow v \in [l_N, u_N] \tag{8}$$

If we then unroll the interval definitions as we did for Equation (4) (i.e., $x \in [l, u]$ can be rewritten as $l \preceq x \wedge x \preceq u$), the optimal solution can be computed through a linear search minimization (or maximization) procedure—e.g., by iteratively asking an SMT solver to find a tighter solution to our interval bounds l_N, u_N than the one computed in a previous step. The following theorem establishes when this iterative procedure is guaranteed to terminate:

Theorem 5.3 (Termination of Iterative GFA). *Suppose that the intervals $[l_N, u_N]$ from Equation (8) belong to an interval domain $\mathcal{D}$ equipped with a partial order $\preceq$. If $\mathcal{D}$ contains no infinite descending chains (i.e., $\prec$ is well-founded), then any algorithm that iteratively solves for l_N, u_N such that $[l_N, u_N] \sqsubsetneq [\![\square_N]\!]^\sharp_N$ will terminate in a finite number of steps.*

Because Theorem 5.3 proves that there are only finitely many possible iterations when the intervals are members of a domain with no infinite descending chains, the following approach is guaranteed to terminate under this condition: we first construct the formula in Equation (8), starting with the largest possible interval, and iteratively query tighter bounds on each of the interval endpoints (e.g., the free variables l_N, u_N). We repeat this process by updating the bounds of our abstract semantics $[\![\square_N]\!]^\sharp_N$ with our previously solved bounds l_N, u_N until the constraint solver returns unsat. One advantage of this process is that at termination, when the constraint solver returns unsat, the interval-hole abstract semantics is guaranteed to be the most precise solution. Another advantage is that, even if this process is terminated with an intermediate solution (i.e., the constraint solver has not yet returned unsat, but, e.g., iterative GFA had exceeded some timeout threshold), this tightened interval may not be most-precise, but is still a sound hole abstraction [Barrett and King 2010]. Note that solving for hole abstractions individually is not optimal, because a tighter hole abstraction for one nonterminal can lead to tighter solutions for other hole abstractions. Thus, we define our constraints over all hole abstractions simultaneously, where we aim to tighten at least one of the intervals on each iteration. Our solver takes this approach.[3]

The following theorem establishes that our computed precise hole abstractions are sound according to Definition 4.6:

[3]Abstract interpretation over intervals often involves using a widening operator. With the method described above, no widening is needed because it starts from $\top$—an over-approximation—and makes a sequence of calls to a logic solver, rather than starting from $\bot$—an under-approximation—and performing successive approximation. Yet another approach would be to use algorithms for computing precise least solutions to systems of (certain classes of) interval equations [Gawlitza et al. 2009; Su and Wagner 2005].

Theorem 5.4 (Precise Hole Abstractions). *For every nonterminal N in a grammar G, the following rule is a sound hole abstraction if $GFASol(\llbracket \square_N \rrbracket^\sharp_N(x), G)$ is a valid solution to interval grammar flow analysis for the value of x:*

$$\frac{[l,u] = GFASol(\llbracket \square_N \rrbracket^\sharp_N(x), G)}{\llbracket \square_N \rrbracket^\sharp_N(x) = [l,u]}\ Hole_N$$

We are able to solve for precise hole abstractions in problems beyond integer arithmetic. For instance, we can derive precise hole abstractions for the CSV-schema example from Section 2.3.

In practice, implementing an abstract semantics requires one to solve $GFASol(\llbracket \square_N \rrbracket^\sharp_N(x), G)$ for every possible input for which the semantics of $\llbracket \square_N \rrbracket^\sharp_N$ needs to be evaluated during our enumeration algorithm. We will discuss this aspect in Section 7.

6 Synthesis of Order Relations

We have shown that abstract semantics can be easily extracted from a concrete semantics that is monotonic with respect to some (partial) order relations over the inputs and outputs. So far, we have assumed that these order relations are given to us. In this section, we show how to "choose" a best set of orders from a set of possible orderings. Between two sets of orders, we prefer the one for which the most productions in the grammar exhibit a monotone semantics.

Definition 6.1 (Comparison of Orders). Given a grammar G, a semantics $(\llbracket \cdot \rrbracket, \sigma)$ describing variables with types in the set $\{T_1, \ldots, T_N\}$, and a set of orders $\omega = \{\preceq_1, \ldots, \preceq_n\}$, we define $Mon_G(\omega)$ as the set of productions in G for which the semantics are monotone with respect to ω.

We say that a set of orders ω_1 is *better* than a set of orders ω_2 if $|Mon_G(\omega_1)| \geq |Mon_G(\omega_2)|$.

The following example illustrates this definition.

Example 6.2 (Bitvectors). Consider the following grammar G_{bv} for expressions over bitvectors.

$$B ::= x \mid \texttt{bvand}\ B\ B \mid \texttt{bvor}\ B\ B \mid \texttt{bvadd}\ B\ B$$

Assume that all variables are associated with one type: unsigned bitvectors of size 8. Consider a saturating semantics of `bvadd` in which addition saturates at $2^8 - 1$, and the following two possible orders over bitvectors:

- $v_1 \preceq_{bw} v_2$ if every bit in v_1 is less than the corresponding bit in v_2; in this case $Mon_G(\{\preceq_{bw}\})$ contains the productions for `bvand` and `bvor`;
- $v_1 \preceq_{bvleq} v_2$ if the integer value of v_1 is smaller or equal than the integer value of v_2; in this case $Mon_G(\{\preceq_{bvleq}\})$ only contains the production for `bvadd`.

The set of orders $\{\preceq_{bw}\}$ is better than the set of orders $\{\preceq_{bvleq}\}$ because it causes the semantics of two productions to be monotonic instead of just one.

While our framework can accommodate multiple possible orders and prune the search space using multiple abstract semantics all evaluated in parallel, this process can become expensive. Moreover, if too few productions are monotonic, the resulting abstract semantics will often just yield $\top$ (Definition 4.5) and not be able to prune any programs. The following definition captures the problem of synthesizing a best set of orders that maximizes monotonicity.

Definition 6.3. Given a grammar G, a semantics $(\llbracket \cdot \rrbracket, \sigma)$ describing variables with types in the set $\{T_1, \ldots, T_N\}$, and a search space of orders Ω, the *order-synthesis problem* is to find a best set of orders $\omega = \{\preceq_1, \ldots, \preceq_n\}$ such that every order $\preceq_i \in \Omega$. Here, a *best set of orders* is a set of orders ω such that ω is better than any set of orders in Ω, i.e. $\forall \omega' \in \Omega. |Mon_G(\omega)| \geq |Mon_G(\omega')|$. (Note that there can be multiple best orders.)

Our implementation only considers sets of orders Ω that are finite (and typically small), and solves the order-synthesis problem by enumerating all orders in Ω, computing how many productions are monotonic for each order, and returning a best one. We focus on enumerating orders that are conjunctions of smaller "atomic" orders from a pre-defined base set that can easily be augmented as needed. These atomic orders are required to match the type of each argument (e.g., $\leq$ for integers; $\rightarrow$ for Boolean; $\preceq_{bw}$ and $\preceq_{bvleq}$ comparison for bitvectors, etc.). This enumeration allow us to consider orders over complex semantic objects with mixed types. We found that initializing Ω with conjunctions over these atomic orders, while simple, performs well in practice and captures many reasonable orders for our monotonicity analysis.

A note on order optimality: There exist more advanced order-synthesis techniques that could improve the overall synthesis procedure. Currently, we locally maximize the number of productions at each non-terminal, but do not reason about global optimizations. It is possible that there is a stronger relation between the grammar and order, where it is more important to maximize productions closer to the starting nonterminal of the grammar, rather than the total number of productions. Additionally, expanding the types of orders considered in the order-synthesis problem (e.g., beyond conjunctive orders) could also lead to improvements in generating the abstract semantics. However, continuing to perform naive enumeration after nontrivially expanding the set of orders could lead to an intractable order-synthesis problem. Designing new algorithms for efficiently synthesizing best orders is an interesting direction beyond the scope of this paper.

7 Evaluation

We implemented our techniques in a tool called Moito, which takes problems in the SemGuS format as input [Kim et al. 2021] and consists of four components: (*i*) A monotonicity checker, built over SMT solvers Z3 [De Moura and Bjørner 2008] and CVC5 [Barbosa et al. 2022], that can determine what productions are monotonic with respect to a given partial order. (*ii*) An order synthesizer, which enumerates a set of partial orders and applies the monotonicity checker to find a solution to an order-synthesis problem. These orders are conjunctions over smaller predefined atomic orders from a defined base set. (*iii*) A solver for grammar-flow analysis, which automatically constructs the system of equations from Section 5, and performs a fixed-point computation by generating an SMT formula and querying Z3 to iteratively tighten interval bounds. (*iv*) A top-down enumerative synthesizer for SemGuS problems that takes the output from the previous steps, and generates the corresponding interval semantics, and uses it to prune partial programs during program synthesis.

Components (i), (ii), and (iii) are independent of (iv), in the sense that their output is reusable by other synthesis tools. The output of running (i), (ii), and (iii) is emitted as a JSON artifact, which describes what productions are monotonic and in what directions, as well as the computed hole abstractions on any provided input-output examples. The artifact can be used by *any* top-down enumerative solver to enable abstraction-based synthesis of SemGuS problems. In cases where we know a semantics is monotonic, but proving it is beyond the capabilities of current SMT solvers, the JSON format allows us to provide to a solver the information necessary to enable abstraction-based pruning. Our synthesizer (component (iv)) currently supports example-based synthesis problems, and invokes component (iii) to compute the precise hole abstractions for all nonterminals and examples *before* starting the enumeration (instead of dynamically calling them when evaluating the semantics and caching solutions, which can be be prohibitively expensive).

7.1 Research Questions and Benchmarks

In our evaluation, we consider as baseline a naive enumeration algorithm that does not perform any pruning—this baseline is the same one used to evaluate the SemGuS unrealizability prover

Messy [Kim et al. 2021]. We do not compare against Messy because Messy can check whether a SemGuS problem can be solved, but it cannot compute a solution. Our limited set of baselines is due to the fact that Moito is a domain-agnostic synthesizer, and we are unaware of any other synthesizers that support programs in the SemGuS format or that can run all of our benchmarks.

Our experimental evaluation is designed to answer the following questions:

RQ1 How effective is monotonicity-based pruning when compared to naive enumeration?
RQ2 How effective is our construction of precise hole abstractions from Section 5?
RQ3 How effective is Moito at identifying interval-abstract semantics?

All experiments were run on a cluster [Center for High Throughput Computing 2006], with each node having an AMD EPYC 7763 64-Core Processor, of which we requested two cores and 12 GiB of RAM. We set a timeout value of 2000s and memory limit of 8 GiB.[4] We ran each experiment 5 times, and report the median of these runs.

Benchmarks. We conducted our evaluation on 430 SemGuS benchmarks from multiple domains, building on top of the SemGuS Toolkit benchmark set [Johnson et al. 2024c].

The first benchmark category includes 82 regular-expression synthesis problems encoded in SemGuS (many of these benchmarks are part of the public SemGuS benchmark set). There are two ways to encode the semantics of regular expressions that each have pros and cons: (*i*) a *shallow* semantics that maps a program to the corresponding term in the SMT theory of regular expressions, (*ii*) a deep semantics that maps a program r and a string s to a Boolean *matrix* that tells us what substrings of s the expression r accepts (see Appendix A in the extended version of the paper [Johnson et al. 2024a]). The benchmark problems include the AlphaRegex benchmarks [Lee et al. 2016] (25 shallow, 25 matrix), regular expressions with complex operators such as negation and character classes (1 shallow, 14 matrix), and CSV-format synthesis problems like in Section 2.3 (15 shallow, 2 matrix) from the CSV schema language [Retter et al. 2016]. The shallow semantics is fast to execute using corresponding regex libraries, but it has limited support in constraint solvers (e.g., no solver for the theory for of quantified formulas). For this reason, for this semantics we must provide the JSON artifact of what productions are monotonic manually (note that effectively we only need to do so once, because all benchmarks share the same operators). The GFA intervals for these shallow semantics can be computed by considering only intervals of the form $[\emptyset, S^*]$, where S is the set of characters appearing in each grammar production. The matrix semantics is slower to execute, but enables constraint solving.

The second benchmark category consists of 10 problems over imperative programming languages with semantics over integer variables. These problems are simple imperative problems that were taken from the SemGuS imperative benchmark dataset. These imperative benchmarks mostly perform loop-free operations (e.g., swap) with three benchmarks involving loops.

The third benchmark category consists of 100 problems involving loop-free programs over bitvector variables. The 100 bitvector benchmarks are actually 25 concrete synthesis problems by Gulwani et al. [2011] expressed with different semantics (note that the ability to modify the program semantics is one of the key features of SemGuS). The four semantics are: (*i*) the traditional bitvector semantics encoded in SMTLib, (*ii*) a *saturated semantics*, where in the case of possible underflowing or overflowing, the result will instead take on the minimum or maximum value, respectively [Sharma and Reps 2017], (*iii*) a semantics that introduces intermediate variables for all sub-expressions being synthesized (similar to three-address code in compilers), and (*iv*) a combination of the second and third semantics. Unlike the other categories, the synthesis constraints are specified as logical

[4]Due to the nature of the computing cluster, there can be large variance in run time between trials; however, this variance was not enough to change whether or not a problem was solvable under the time and memory constraints.

Table 1. Benchmark performance of Moito over baseline, broken out by benchmark category

Domain		Moito only	>15% faster	+/-15%	>15% slower	baseline only
Regex	Matrix	1	5	11	12	-
	Shallow	2	7	6	5	-
	CSV	7	5	3	-	-
Imperative		2	4	4	-	-
Bitvector		6	6	28	4	-
Boolean		6	31	60	27	-
Total		24	58	112	48	-

formulas, requiring our enumeration algorithm to perform a Counterexample Guided Inductive Synthesis (CEGIS) loop that restarts the algorithm with new examples at each iteration. Because of this dynamic aspect of the algorithm, we do not run the precise hole abstraction from Section 5 on this set of benchmarks; it would require solving SMT constraints while performing enumeration.

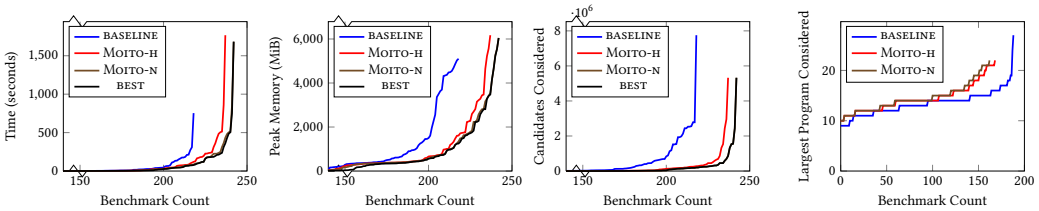
The fourth category of benchmarks contains 238 problems where the goal is to synthesize Boolean formulas with restricted syntaxes—e.g., cubes (84 benchmarks), CNF (77 benchmarks), and DNF (77 benchmarks). Because a general-purpose solver cannot compete with specialized state-of-the-art synthesizers for Boolean formulas, we created a simple dataset of Boolean-function synthesis problems by randomly generating formulas of the different syntax styles with varying lengths (3-11 for cube, 2-12 for CNF/DNF) and number of variables (4-15 for cube, 4-10 for CNF/DNF).

The fifth category of benchmarks contains 5 problems adapted from Mell et al. [2024], a new approach for synthesizing data-classification programs over a quantitative objective function—e.g., accuracy or F_1 score—that manually exploits monotonicity. These benchmarks were created using the two DSLs presented by Mell et al. that use folding/map operators and Kleene algebra with tests, both to synthesize video trajectories. One of the benchmarks is an encoding of their simple motivating example. The semantics of each DSL is encoded twice to create four other benchmarks with differing input sizes (because SemGuS requires one to define the number of inputs). In contrast to the other categories, we exclusively use these benchmarks to evaluate RQ3 (i.e., whether our tool can synthesize abstract semantics), because traditional top-down enumeration is not sufficient to solve synthesis problems with quantitative objectives. These five benchmarks are not included in our total of 430.

7.2 Effectiveness of Monotonicity-Based Pruning

We evaluated the effectiveness of Moito on our benchmarks when an abstract semantics is provided; we also measured the time taken to compute the abstract semantics (see Section 7.4). We present data for baseline top-down enumeration (baseline), top-down enumeration with interval-based pruning without precise hole abstractions (Moito-h), and top-down enumeration with interval-based pruning with precise hole abstractions (Moito-n). In what follows, we use Moito to denote the virtual best version of our tool that runs Moito-h and Moito-n in parallel and reports the result of the first terminating instance, and best to denote the virtual best version solver that runs Moito-h, Moito-n, and baseline in parallel.

Table 1 provides an overview comparing the performance between Moito and baseline. Moito (242/430 solved) can solve 24 more benchmarks than baseline (218/430 solved). The benchmarks that Moito can solve but baseline cannot, fall into the following categories: regular-expressions/CSV (10), imperative (2), bit vectors (6), and Boolean (6). Moito also significantly outperforms baseline



(a) Running time (b) Memory (c) Enumerated programs (d) Program size

Fig. 2. The first three plots compare MOITO-H, MOITO-N, and BASELINE across time, memory, and number of enumerated programs (we start the x-axis at 150 to better illustrate the interesting behavior—i.e., we do not show the behavior on the 150 easiest benchmarks). The last plot shows the maximum size reached before timing out on problems that could not be solved. Note that a line that is lower and to the right represents better performance in the first three plots, and higher and to the left in the final plot.

on 58 additional benchmarks that both could solve. These 82 benchmarks where MOITO markedly outperforms BASELINE are typically the larger and more complex benchmarks that were solvable. On the other hand, there are 48 benchmarks where MOITO is considerably slower than BASELINE. There were no benchmarks that only BASELINE could solve that MOITO could not. The fact that MOITO is not always faster is due to a known issue in program synthesis: computing an abstract semantics and checking whether *every* partial program can be pruned can be more expensive than simply exploring the search space, especially if few programs are pruned [Guria et al. 2023].

Figure 2a shows a cactus plot of the cumulative number of benchmarks solved by each configuration after a certain amount of time. MOITO (and both MOITO-H and MOITO-N) can solve more benchmarks than BASELINE in cumulatively less time. Similar trends are observed for memory usage in Figure 2b. Figure 2c shows the number of concrete programs enumerated for each benchmark before returning a solution, which reveals how many programs MOITO can prune: for 75% of the benchmarks, MOITO explores fewer than 50% of the programs explored by BASELINE. The trends shown in Figure 2c hint that this difference would grow if we were to consider longer timeouts.

The benchmarks that none of the tools can solve fall into the following categories: regular-expressions/CSV (19), bit vectors (56), and Boolean (114). All 10 imperative benchmarks could be solved. The vast majority of unsolved benchmarks require finding large solutions that are far beyond the search space explored in the given time: for instance, the Boolean benchmarks contain up to twelve clauses or 15 variables, and the bitvector benchmarks contain up to 15 imperative variables or a solution AST of size up to 59. Moreover, because there are four instances of each bitvector problem with different semantics, and the four instances are of similar difficulty, MOITO is more likely to solve all four or none of the instances for a particular problem. Figure 2d shows how deep in the search space (i.e., what program sizes) each tool got for the benchmarks that each tool could run but not solve.[5] In general, MOITO enumerated much larger programs than BASELINE (e.g., size 22 vs. 15) before timing out, thus showing its ability to reach deeper into the search space.

To answer RQ1: The monotonicity-based pruning approach allows MOITO to solve more benchmarks than the enumeration BASELINE (242 vs. 218), with MOITO-H solving 237 and MOITO-N all 242. For the benchmarks that both MOITO and BASELINE can solve, MOITO shows modest improvements in the time taken by MOITO to solve some of the benchmarks, as well as less memory being required. Even when the benchmarks do not finish within the time limit or memory limit, MOITO reaches larger program sizes in the search space than those explored by BASELINE.

[5]Note that the x-axis of Figure 2d represents all benchmarks that timed out for some tool, and thus is a different set than the x-axes of Figures 2a, 2b, and 2c.

A note on comparisons to state-of-the-art solvers: Moito can solve 19/25 AlphaRegex [Lee et al. 2016] benchmarks within our timeout of 2000s, (when using the shallow semantics), whereas AlphaRegex reports solving 25/25 in less than a minute each. AlphaRegex implements many other domain-specific optimizations (e.g., a simple form of equality saturation) that our tool cannot automate because the input problem is given as an arbitrary SemGuS file. For Simpl [So and Oh 2017], a direct comparison is more difficult, because their benchmark set focuses on loops. Moito cannot prove loops monotonic, and so our technique would not produce pruning benefits on these benchmarks. On the other hand, many of our non-Boolean benchmarks (97/192) are beyond the reach of existing customized synthesizers. Specifically, 5/39 matrix regular expressions are not expressible in AlphaRegex (due to negation operators). None of the CSV benchmarks (17/17) are supported by AlphaRegex (due to character classes beyond "0"/"1" and CSV-format-specific grammars). 75/100 of the bitvector benchmarks are not solvable using Brahma [Gulwani et al. 2011], because the benchmarks use an alternative semantics (imperative and/or saturating semantics). This generality is an advantage of a parameterized framework like SemGuS, where users can apply a solver that supports all instantiations of the framework, rather than a dozen domain-specific tools with their own restrictive DSLs.

7.3 Effectiveness of Precise Hole Abstractions

Figure 3 compares Moito-n and Moito-h using a scatter plot. Both variants seem to be beneficial in different settings, although on average, Moito-n is 6% faster than Moito-h (geomean, variance 1.06), and Moito-h explores twice as many programs as Moito-n. Additionally, there were 5 CSV benchmarks that were only solved by Moito-n. We conjecture that Moito-n is sometimes slower than Moito-h because while Moito-n can compute more precise hole abstractions, propagating the semantics of intervals different than $\top$ is generally more expensive. Therefore, in cases where the increased precision does not prune more programs, Moito-n is slower.

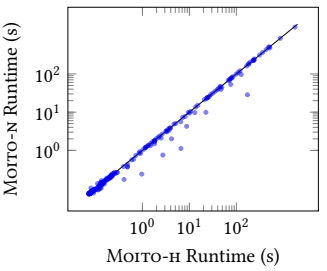


Fig. 3. Moito-n vs Moito-h (log,log)

As expected, Moito-n performs much better than Moito-h on benchmarks where it can prune many more programs, and about the same where it cannot. This bimodality is evident in Figure 3; most benchmarks are on the 1x (no improvement) diagonal line, but there is a subset of benchmarks below the 1x diagonal line, showing substantial improvement. A frequency analysis shows the largest cluster of benchmarks within 0.95x - 1.05x improvement (172 of 237), with a second cluster of 18 benchmarks above 1.20x improvement (max of 6x improvement). Similar trends are seen for the number of concrete candidate programs considered for each benchmark: most benchmarks (154 of 237) do not check any fewer programs with Moito-n, with a long tail of benchmarks showing improvement (10 benchmarks check over 100x fewer programs). For example, for the 10/17 benchmarks in the CSV category solved by both tools, Moito-n explores on average only 0.1% of the programs explored by Moito-h, and is on average 2.6x faster.

Benchmarks in other categories also took advantage of Moito-n. For example, for the imperative benchmark "max3-impv," (which computes the maximum of 3 integers), Moito-n (*i*) proved that there were non-terminals that returned only values in the interval $[0, \infty]$, (*ii*) ran 24% faster, and (*iii*) checked 47% fewer concrete programs. For other cases where Moito-n could not compute better intervals than $\top$, both Moito-n and Moito-h checked the same number of concrete programs and had similar running times, as expected.

To answer RQ2: Moito-n is very beneficial for instances when the structure of the grammar restricts the possible values of certain nonterminals, and therefore the more precise hole abstractions can prune many programs.

7.4 Effectiveness of Computing Abstract Semantics

Moito can compute the interval abstract semantics for 373/430 of the original benchmarks. Moito can also prove monotonicity on all five benchmarks adapted from Mell et al. [2024], where Moito was able to automatically generate the abstract semantics that were manually defined by Mell et al..

Computing the abstract semantics timed out on 12 of the imperative benchmarks over bitvectors and 4 regular-expression with matrix-semantics benchmarks. These benchmarks involved semantics in which functions take 10-100 variables as input, thus causing the order-synthesis algorithm to consider many possible order combinations. As we mentioned, there is currently no solver that supports the quantified theory of regular expressions, which is necessary for computing an abstract semantics of the 41 regular-expression benchmarks with shallow semantics.

The time to compute the abstract semantics varied across domains. All the Boolean benchmarks could terminate in less than a second (avg 0.57s) and the imperative benchmarks took around 0.8 to 5 seconds each (avg 2.1s). The variance was larger for regular-expressions and bitvector benchmarks, and we show a detailed analysis of these categories in Figure 4. On average, it took 150s to compute the semantics for regular-expression benchmarks, and the time scales with the size of the matrices used in the semantics (i.e., the length of the input examples). Similarly, computing the semantics of bitvector benchmarks took on average 11s with the time scaling exponentially with the number of variables in the considered programming language, because the size of the order search space grows exponentially.

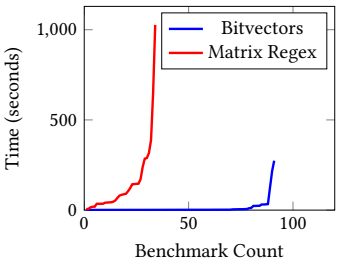


Fig. 4. Time to compute interval abstract semantics

We note that in practice, an abstract semantics does not need to be recomputed every time the specification of the input problem changes, as long as the language remains the same.

While Moito always outputs an abstract semantics, the individual CHCs are only precise abstractions if the original semantics was monotonic. Moito can identify orders under which the semantics is monotonic for all the productions in the grammars for regular expressions, Boolean, and imperative programs. In the case of bitvectors, Moito can find at least one order for either the traditional or saturating semantics for ~87% of productions. We observed variability in these benchmarks; in one case only 6 out of 11 productions could be proven monotonic. An example showing why a bitvector semantics is not monotonic for all productions was illustrated in Section 6.

Computing precise hole abstractions takes on average 0.73s per input example, although the time can vary across different applications (regular expressions 1.3s, imperative 9.9s, Boolean 0.02s). Although this step can be costly, we note that Moito-n can sometimes provide large performance gains on certain benchmarks that Moito-h cannot (Section 7.3). As previously mentioned, we omit the bitvector benchmarks from this analysis because they use logical specifications.

To answer RQ3: Moito can discover precise abstract semantics (and precise hole abstractions) for most benchmarks. This result confirms that our framework can *automatically* discover many of the domain-specific techniques used in existing tools, and generalizes them to new domains (e.g., Boolean formulas, bitvector programs, and DSLs for video trajectories).

8 Related Work

Top-Down Enumeration and Pruning. A number of papers have addressed the problem of program synthesis by applying top-down enumeration in specific domains, such as regular expressions [Lee et al. 2016], imperative programs [So and Oh 2017], SQL queries [Wang et al. 2017a], Datalog programs [Si et al. 2018], and functional programs [Polikarpova et al. 2016]. These tools differ from our work in two key ways. First, our approach applies to arbitrary synthesis problems defined in the SemGuS framework, whereas these tools each implement a solution to *one fixed* domain-specific synthesis problem. Because of this specificity, these tools outperform Moito on their respective tasks, but their implementations are monolithic and tailored to such tasks. Second, while these tools implement hard-coded pruning strategies, Moito automatically discovers pruning opportunities by extracting an abstract semantics for the given SemGuS problem. In summary, our work can automatically discover ways to prune in top-down enumeration for problems defined in the SemGuS framework, while these tools use manually-defined pruning strategies that target specific domains.

Besides subsuming several of the pruning strategies used by AlphaRegex [Lee et al. 2016] and SIMPL [So and Oh 2017], our interval-based framework also captures some of the pruning approaches used when synthesizing SQL queries [Wang et al. 2017a], Datalog programs [Si et al. 2018], and data-processing tasks [Mell et al. 2024]. Scythe [Wang et al. 2017a] (indirectly) uses an interval $[T_l, T_u]$ (where T_l and T_u are tables) to represent what possible output tables could be the result of evaluating the completion of a partial SQL query, and uses the fact that most queries are monotonic with respect to the predicates appearing in a where-clause—i.e., a more permissive clause yields a bigger table. Si et al. [2018] use a similar insight to construct an interval (akin to a version-space algebra) over the set of Datalog programs that are consistent with a set of input-output examples. We do not evaluate our approach on these applications because the SemGuS format currently lacks some features that are necessary to model these applications (e.g., the theory of bags for SQL and fixed-point logics for Datalog), and the sizes of the inputs used in these domains (e.g., tables) are only within the reach of domain-specific tools.

Mell et al. [2024] use interval abstractions when productions are monotonic to guide their search for optimal synthesis, and their specific monotonicity property can be viewed as a specific instantiation of ours. Their work focuses on two specific domains, that of numbers and Booleans under their standard orders. While Mell et al. had to *manually* prove monotonicity *a priori*, Moito was able to *automatically* infer that the semantics constructs used in their experiments are monotonic (Section 7). Although their work uses monotonicity in an additional way—i.e., to maximize an objective function—their tool could be another client of our automated monotonicity analysis.

Abstraction-Guided Synthesis. Complex forms of abstraction-based pruning have been applied in many other program-synthesis tools [So and Oh 2017; Vechev et al. 2010; Wang et al. 2017b]. However, these tools are also domain-specific and cannot tackle SemGuS problems.

Simba [Yoon et al. 2023] combines forward abstract interpretation (for soundly approximating the set of possible outputs obtainable from inputs of partial programs) with backward abstract interpretation (for approximating the set of possible inputs, starting from the outputs). Simba only supports SyGuS problems (i.e., expression-synthesis problems) and requires the user to manually provide highly-precise abstract semantics. Despite this limitation, an interesting research direction is whether Simba's approach can be automatically generalized to SemGuS in the same way that our framework automatically generalizes interval-based pruning—i.e., we hope our work will be the first of many in this spirit. Absynthe [Guria et al. 2023] is a general-purpose framework for synthesis with abstraction-based pruning that allows users to manually supply abstract semantics for the language over which synthesis is being performed. In contrast, Moito can automatically discover a precise abstract interval semantics from the user-provided concrete semantics.

Generating Abstract Semantics. Amurth [Kalita et al. 2022] synthesizes abstract semantics for arbitrary languages by asking a user to provide a grammar of possible abstract functions to choose from (i.e., our function $f^\sharp$ in Definition 4.5). While Amurth is very general, it is not fully automated and requires the user of the tool to provide a specialized grammar for each abstract domain—and in many cases for each abstract function. (These grammars often contain complex insights on what a particular abstract function should look like.) Instead, our work is based on monotonicity conditions under which abstract semantics can be generated *automatically*. Combining Amurth with our tool to generate abstract semantics beyond the automatically generated ones discussed in this paper is an interesting research direction, although it would require ways to identify what grammars one should provide to Amurth.

Atlas [Wang et al. 2018] also learns abstractions for pruning in synthesis. Atlas considers abstract domains consisting of linear equalities, which work better than intervals in certain settings—e.g., reasoning about string lengths—but are limited to numerical domains—e.g., they cannot capture the conjunctions of Booleans used in many of our benchmarks. Most importantly, Atlas requires a set of training problems to synthesize an abstract domain and the transformers (i.e., the tool needs to be trained for every new domain), while Moito is domain-agnostic, does not require a training phase, and it synthesizes an abstract semantics directly from the provided concrete semantics alone.

Other Forms of Enumeration. Bottom-up enumeration enumerates *subprograms* of increasing size derivable from each nonterminal in the grammar, and prunes the search space by only maintaining programs that are observationally inequivalent on the examples [Alur et al. 2017]. Observational equivalence can be automated for expression-synthesis problems, but not, for example, for synthesizing imperative programs. This limitation is due to the fact that in an imperative programming language, different subprograms are evaluated on different states (i.e., the programs are stateful). For the same reason, there is currently no way to implement bottom-up enumeration (with pruning) for SemGuS problems, hence our focus on top-down enumeration. Hybrid versions of top-down and bottom-up enumeration share the same limitation [Lee 2021].

Symbolic Solvers. Messy [Kim et al. 2021] is currently the only published SemGuS solver, and therefore the only solver that is designed to solve the same range of tasks as Moito. Messy employs a constraint-based approach for solving SemGuS problems using a Constrained Horn Clause solver with the dual goals of being able to synthesize a program or to prove that the synthesis problem is unrealizable. While Messy performs well for proving unrealizability of SemGuS problems, it has effectively no synthesis capabilities (i.e., it cannot produce an output program when a problem is realizable); therefore we do not compare Moito against it in our evaluation.

The baseline implementation of top-down enumeration used in our evaluation is the same as the baseline called MessyEnum used to evaluate Messy. To the best of our knowledge, Moito is the first enumeration technique for SemGuS problems that moves beyond naive enumeration.

9 Conclusion

This paper presents a unified framework for determining precise interval abstract semantics that can speed up program synthesis via enumeration for problems written in the SemGuS framework. Unlike existing works on top-down enumeration, our framework is domain-agnostic (i.e., it does not know *a priori* the semantics of programs appearing in the search space).

Recall that the solvers in the initial SyGuS competition were unable to solve a majority of the original SyGuS benchmarks. The difficulty of the benchmark set then spurred the development of second- and third-generation solvers that could solve most of the competition benchmarks [Alur et al. 2016]. SemGuS and the existing SemGuS solvers are currently in their infancy—i.e., they have limited scalability—but we hope that Moito is the first in a series of improved SemGuS solvers. In

particular, our work opens the door for generalizing and automating many other domain-specific synthesis techniques, so that they can be lifted to a general framework like SemGuS. For example, the Amurth [Kalita et al. 2022] tool for synthesizing abstract transformers or the Atlas [Wang et al. 2018] tool for constructing linear abstractions could be combined with our theory to specialize our work to more complex abstract domains. In the same way domain-specific insights have caused tremendous speedups in SyGuS solvers (which initially could only solve trivial problems), we are hopeful that future efforts similar to the one described in this paper will result in SemGuS solvers that—despite their generality—can solve complex problems.

Data-Availability Statement

The software that supports Section 7 is available on Zenodo [Johnson et al. 2024b].

Acknowledgments

The authors would like to thank Wiley Corning for early thoughts on this work. Supported, in part, by a Microsoft Faculty Fellowship; a gift from Rajiv and Ritu Batra; and NSF under grants CCF-1750965, CCF-1918211, CCF-2023222, CCF-2211968, and CCF-2212558. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities.

References

Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385

Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2016. Results and Analysis of SyGuS-Comp'15. *Electronic Proceedings in Theoretical Computer Science* 202 (Feb. 2016), 3–26. https://doi.org/10.4204/eptcs.202.3

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*. https://api.semanticscholar.org/CorpusID:9465894

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Edd Barrett and Andy King. 2010. Range and Set Abstraction using SAT. In *Proceeding of the Second International Workshop on Numerical and Symbolic Abstract Domains, NSAD@SAS 2010, Perpignan, France, September 13, 2010 (Electronic Notes in Theoretical Computer Science, Vol. 267)*, Antoine Miné and Enric Rodríguez-Carbonell (Eds.). Elsevier, 17–27. https://doi.org/10.1016/J.ENTCS.2010.09.003

Center for High Throughput Computing. 2006. Center for High Throughput Computing. https://doi.org/10.21231/GNT1-HW21

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

Thomas Gawlitza, Jérôme Leroux, Jan Reineke, Helmut Seidl, Grégoire Sutre, and Reinhard Wilhelm. 2009. Polynomial Precise Interval Analysis Revisited. In *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 5760)*, Susanne Albers, Helmut Alt, and Stefan Näher (Eds.). Springer, 422–437. https://doi.org/10.1007/978-3-642-03456-5_28

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-Free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 62–73. https://doi.org/10.1145/1993498.1993506

Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. 2023. Absynthe: Abstract Interpretation-Guided Synthesis. *Proc. ACM Program. Lang.* 7, PLDI, Article 171 (jun 2023), 24 pages. https://doi.org/10.1145/3591285

Keith J.C. Johnson, Rahul Krishnan, Thomas Reps, and Loris D'Antoni. 2024a. Automating Pruning in Top-Down Enumeration for Program Synthesis Problems with Monotonic Semantics. arXiv:2408.15822 [cs.PL] https://arxiv.org/abs/2408.15822

Keith J.C. Johnson, Rahul Krishnan, Thomas Reps, and Loris D'Antoni. 2024b. *Automating Pruning in Top-Down Enumeration for Program Synthesis Problems with Monotonic Semantics.* https://doi.org/10.5281/zenodo.12669773

Keith J.C. Johnson, Andrew Reynolds, Thomas Reps, and Loris D'Antoni. 2024c. The SemGuS Toolkit. In *Computer Aided Verification*, Arie Gurfinkel and Vijay Ganesh (Eds.). Springer Nature Switzerland, Cham, 27–40.

Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D'Antoni, Thomas Reps, and Subhajit Roy. 2022. Synthesizing Abstract Transformers. arXiv:2105.00493 [cs.PL]

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.

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-Based Model Checking for Recursive Programs. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 17–34.

Mina Lee, Sunbeom So, and Hakjoo Oh. 2016. Synthesizing Regular Expressions from Examples for Introductory Automata Assignments. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Amsterdam, Netherlands) *(GPCE 2016)*. Association for Computing Machinery, New York, NY, USA, 70–80. https://doi.org/10.1145/2993236.2993244

Woosuk Lee. 2021. Combining the Top-down Propagation and Bottom-up Enumeration for Inductive Program Synthesis. *Proc. ACM Program. Lang.* 5, POPL, Article 54 (jan 2021), 28 pages. https://doi.org/10.1145/3434335

Stephen Mell, Steve Zdancewic, and Osbert Bastani. 2024. Optimal Program Synthesis via Abstract Interpretation. *Proc. ACM Program. Lang.* 8, POPL, Article 16 (jan 2024), 25 pages. https://doi.org/10.1145/3632858

Ulrich Möncke and Reinhard Wilhelm. 1991. Grammar flow analysis. In *Attribute Grammars, Applications and Systems*, Henk Alblas and Bořivoj Melichar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–186.

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093

Adam Retter, David Underdown, and Rob Walpole. 2016. *CSV Schema Language 1.1.* https://digital-preservation.github.io/csv-schema/csv-schema-1.1.html

Tushar Sharma and Thomas Reps. 2017. Sound Bit-Precise Numerical Domains. 500–520. https://doi.org/10.1007/978-3-319-52234-0_27

Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. 2018. Syntax-Guided Synthesis of Datalog Programs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 515–527. https://doi.org/10.1145/3236024.3236034

Sunbeom So and Hakjoo Oh. 2017. Synthesizing Imperative Programs from Examples Guided by Static Analysis. arXiv:1702.06334 [cs.PL]

Armando Solar-Lezama. 2013. Program sketching. *Int. J. Softw. Tools Technol. Transf.* 15, 5-6 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

Zhendong Su and David A. Wagner. 2005. A class of polynomially solvable range constraints for interval analysis without widenings. *Theor. Comput. Sci.* 345, 1 (2005), 122–138. https://doi.org/10.1016/J.TCS.2005.07.035

Martin Vechev, Eran Yahav, and Greta Yorsh. 2010. Abstraction-Guided Synthesis of Synchronization. *SIGPLAN Not.* 45, 1 (jan 2010), 327–338. https://doi.org/10.1145/1707801.1706338

Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017a. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365

Xinyu Wang, Greg Anderson, Isil Dillig, and Kenneth L McMillan. 2018. Learning Abstractions for Program Synthesis. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I 30*. Springer, 407–426.

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017b. Program Synthesis using Abstraction Refinement. *CoRR* abs/1710.07740 (2017). arXiv:1710.07740 http://arxiv.org/abs/1710.07740

Yongho Yoon, Woosuk Lee, and Kwangkeun Yi. 2023. Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation. *Proc. ACM Program. Lang.* 7, PLDI, Article 174 (jun 2023), 25 pages. https://doi.org/10.1145/3591288