



# Synthesizing Formal Semantics from Executable Interpreters

JIANGYI LIU, University of Wisconsin – Madison, USA
CHARLIE MURPHY, University of Wisconsin – Madison, USA
ANVAY GROVER, University of Wisconsin – Madison, USA
KEITH J.C. JOHNSON, University of Wisconsin – Madison, USA
THOMAS REPS, University of Wisconsin – Madison, USA
LORIS D'ANTONI, University of California, San Diego, USA

Program verification and synthesis frameworks that allow one to customize the language in which one is interested typically require the user to provide a formally defined semantics for the language. Because writing a formal semantics can be a daunting and error-prone task, this requirement stands in the way of such frameworks being adopted by non-expert users. We present an algorithm that can automatically synthesize inductively defined syntax-directed semantics when given (*i*) a grammar describing the syntax of a language and (*ii*) an executable (closed-box) interpreter for computing the semantics of programs in the language of the grammar. Our algorithm synthesizes the semantics in the form of Constrained-Horn Clauses (CHCs), a natural, extensible, and formal logical framework for specifying inductively defined relations that has recently received widespread adoption in program verification and synthesis. The key innovation of our synthesis algorithm is a Counterexample-Guided Synthesis (CEGIS) approach that breaks the *hard* problem of synthesizing a set of constrained Horn clauses into small, tractable expression-synthesis problems that can be dispatched to existing SyGuS synthesizers. Our tool Synantic synthesized inductively-defined formal semantics from 14 interpreters for languages used in program-synthesis applications. When synthesizing formal semantics for one of our benchmarks, Synantic unveiled an inconsistency in the semantics computed by the interpreter for a language of regular expressions; fixing the inconsistency resulted in a more efficient semantics and, for some cases, in a 1.2x speedup for a synthesizer solving synthesis problems over such a language.

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

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

**ACM Reference Format:**
Jiangyi Liu, Charlie Murphy, Anvay Grover, Keith J.C. Johnson, Thomas Reps, and Loris D'Antoni. 2024. Synthesizing Formal Semantics from Executable Interpreters. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 284 (October 2024), 27 pages. https://doi.org/10.1145/3689724

## 1 Introduction

Recent work on frameworks for program verification and program synthesis has created tools that are parametric in the language that is supported [5, 13, 15]. A user of such a framework must define the language of interest by giving both a syntactic specification and a formal semantic specification

Authors' Contact Information: Jiangyi Liu, University of Wisconsin – Madison, Madison, USA, jiangyi.liu@wisc.edu; Charlie Murphy, University of Wisconsin – Madison, Madison, USA, tcmurphy4@wisc.edu; Anvay Grover, University of Wisconsin – Madison, Madison, USA, anvayg@cs.wisc.edu; Keith J.C. Johnson, University of Wisconsin – Madison, Madison, USA, keith.johnson@wisc.edu; Thomas Reps, University of Wisconsin – Madison, Madison, USA, reps@cs.wisc.edu; Loris D'Antoni, University of California, San Diego, La Jolla, USA, ldantoni@ucsd.edu.



ACM 2475-1421/2024/10-ART284
https://doi.org/10.1145/3689724

of the language. The semantic specification assigns a meaning to each program in the language. However, for most programming languages, and even for simple ones used in program-synthesis applications, it is usually a demanding task to create a *formal* semantics that defines the behaviors of the programs in the language. Obstacles include: (*i*) the language's semantics might only be documented in natural language, and thus may be ambiguous (or worse, inconsistent), and (*ii*) the sheer level of detail that is involved in writing such a semantics.

*Synthesizing Formal Semantics from Interpreters.* In this paper, we propose an alternative approach—based on synthesis—that is applicable to any programming language for which a compiler or interpreter exists. Such infrastructure serves as an operational semantics for the language, albeit one for which anything other than closed-box access would be difficult. Assuming existence of a working compiler or interpreter is not hard — usually a language (typically not an "academic" language) already has an interpreter already implemented, and the language users, if they want to access techniques like verification and synthesis, need a formal semantics. Thus, we take closed-box access as a given, and ask the following question:

> *Is it possible to use an existing compiler or interpreter for a language L to create a formal semantics for L automatically?*

In this paper, we assume that the given compiler or interpreter is capable of executing any program or subprogram in language $L$.

This question is natural, but answering it formally requires one to address two key challenges.

First, in what formalism should the formal semantics be expressed? The right formalism should be expressive enough to capture common semantics, yet structured enough to allow synthesis to be possible. Furthermore, the formalism should not be tied to any specific programming language—i.e., it should be language-agnostic.

Second, how can the synthesis problem be broken down into simple enough small problems for which one can design a practical approach? The representation of the semantics of most programming languages is usually very large, and a monolithic synthesis approach that does not take advantage of the compositionality of semantics definitions is bound to fail.

*Our Approach.* In this paper, we address both of these challenges and present an algorithm that can automatically synthesize an inductively defined syntax-directed semantics when given (*i*) a grammar describing the syntax of the language, and (*ii*) an executable (closed-box) interpreter for computing the semantics of programs in the language on given inputs.

To address the first of the aforementioned challenges, we choose to synthesize the formal semantics in the form of Constrained Horn Clauses (CHCs), a well-studied fragment of first-order logic that already provides the foundation of SemGuS [6, 13], a domain- and solver-agnostic framework for defining arbitrary synthesis problems. CHCs can naturally express a big-step operational semantics, structured as an inductive definition over a language's abstract syntax, which makes them appropriate for compositional reasoning.

For example, the operational semantics for an assignment to a variable `x` in an imperative programming language can be written as the following CHC:

$$\frac{[\![e]\!](s_1) = r_1 \qquad s_0 = s_1 \wedge r_0 = s_0[x \mapsto r_1]}{[\![\texttt{x} := e]\!](s_0) = r_0}$$

The CHC is defined inductively in terms of the semantics of the child term $e$.

To address the second aforementioned challenge, we take advantage of the inductive structure of CHCs and design a synthesis algorithm that inductively synthesizes the semantics of programs

in the grammar, starting from simple base constructs and moving up to more complex inductively-defined constructs. For each construct in the language, our algorithm uses a counter-example-guided inductive synthesis (CEGIS) loop to synthesize the semantic rule—i.e., the CHC—for that construct. For each construct, we use input-output valuations obtained by calling the closed-box interpreter to approximate the behavior of its child terms. Such an approximation allows us to synthesize the semantics construct-by-construct, rather than all at once, which converts the problem of synthesizing semantics into many smaller problems that only have to synthesize part of the overall semantics.

To evaluate our approach, we implemented it in a tool called Synantic. Our evaluation of Synantic involved synthesizing the semantics for languages with a wide variety of features, including assignments, conditionals, while loops, bit-vector operations, and regular expressions. The evaluation revealed that our approach not only can help synthesize semantics of non-trivial languages but can also help debug existing semantics.

*Goals and No-goals.* Our tool Synantic mainly targets users who want to use verification and synthesis techniques on an existing language. Once Synantic creates the semantics in SemGuS format, a wide range of tools based on SemGuS can be instantly applied [10]. For example, such a way enables the user to get a synthesizer for the existing language *for free*, because the creation of SemGuS files requires minimal manual labor. Also the original goal is helping SemGuS users, our techniques are general and we envision they could potentially be applied to other semantic-specification frameworks (e.g., to help formalize semantics for use with a theorem prover). Synthesizing semantics of purely academic languages is a no-goal for our tool, because most of them already have a formal semantics available before the interpreter is implemented, thus creating the SemGuS specifications would be trivial.

*Contributions.* Our work makes the following contributions:

- We introduce a new kind of synthesis problem: the *semantics-synthesis problem* (Section 3).
- We devise an algorithm for solving semantics-synthesis problems (Section 4). In this algorithm, we harness an example-based program synthesizer (specifically a SyGuS solver) to synthesize the constraint in each CHC.
- We implement our algorithm in a tool, called Synantic, which also supports an optimization for multi-output productions, i.e., productions whose semantic constraints include multiple output variables (Section 5).
- We evaluate Synantic on a range of different language benchmarks from the program-synthesis literature. For one benchmark, the Synantic-generated semantics revealed an inconsistency in the way the original semantics had been formalized. Fixing the inconsistency in the semantics resulted in a more efficient semantics and a speedup (in some case 1.2x) for a synthesizer solving synthesis problems over such a language (Section 6)

Section 2 illustrates how our algorithm synthesizes the semantics of an imperative while-loop language. Section 7 discusses related work. Section 8 concludes.

References of the form Appendix A.1 refer to appendices that are available in the arXiv version of this paper [17].

## 2 Illustrative Example

As discussed in Section 1, our technique synthesizes a semantic specification that is compatible with the Semantics-Guided Synthesis (SemGuS) format [13]. SemGuS is a domain- and solver-agnostic framework for specifying program synthesis and verification problems [6]. A SemGuS problem consists of three components that the user must provide: (*i*) a grammar specifying the syntax

of programs; (*ii*) a semantics for every program in the language of the grammar, provided as a set of Constrained Horn Clauses (CHCs) assigned to the productions of the grammar; and (*iii*) a specification of the desired program that makes use of the semantic predicates. Crucially, SemGuS enables the development of general tools for program synthesis and verification, thus reducing the burden of creating such tools for custom languages [10]. However, the stumbling block is that the end user must be able to provide a semantics of the language they are interested in working with, a task that can be burdensome and error-prone to perform by hand. In this section, we illustrate how our technique (implemented in Synantic) automatically synthesizes such a semantics for an imperative language Imp (cf. Example 2.1)—a simple but illustrative example of Synantic's abilities.

*Example 2.1 (Syntactic Definition of Imp).* Consider the grammar $G_{\text{Imp}_n}$ that defines the syntax of Imp for programs with $n$ variables $\texttt{x}_1, \ldots, \texttt{x}_n$:

$$
\begin{aligned}
S &::= \texttt{x}_1 \coloneqq E \mid \cdots \mid \texttt{x}_n \coloneqq E \mid S\ ;\ S \mid \texttt{ite}\ B\ S\ S \mid \texttt{while}\ B\ \texttt{do}\ S \\
&\quad \mid \texttt{do}\ S\ \texttt{while}\ B \mid \texttt{repeat}\ S\ \texttt{until}\ B \\
B &\coloneqq \texttt{false} \mid \texttt{true} \mid \neg\, B \mid B \wedge B \mid B \vee B \mid E < E \\
E &\coloneqq \texttt{0} \mid \texttt{1} \mid \texttt{x}_1 \mid \cdots \mid \texttt{x}_n \mid E + E \mid E - E
\end{aligned}
$$

The Imp language consists of arithmetic and Boolean expressions, statements for assignment to the variables $\texttt{x}_1$ through $\texttt{x}_n$, sequential composition, if-then-else, and various looping constructs. Imp also comes equipped with an executable interpreter $\mathcal{I}_{\text{Imp}}$ that assigns to each term $t \in \mathcal{L}(G)$ its standard (denotational) semantics (e.g., arithmetic and Boolean expressions are evaluated as in linear integer arithmetic, $\texttt{x}_i \coloneqq e$ takes as input a state, and outputs the input state with $x_i$'s value updated by the result of evaluating $e$, etc.).

Suppose that we did not know the semantics of Imp *a priori*; that is, suppose that we only have access to the interpreter $\mathcal{I}_{\text{Imp}}$. How can we synthesize a formal semantics for each program in $G_{\text{Imp}}$ using the interpreter? A naïve approach would randomly generate a large set of terms and inputs, and try to learn a function mapping inputs to outputs for each term. However, this approach would only provide a semantics for the enumerated terms, and fails to generalize to the entire language. A less naïve approach might attempt to form a monolithic synthesis problem to synthesize a semantic function for each production of the grammar that satisfies a set of generated example terms and input-output pairs. However, it is known that synthesizers scale exceptionally poorly in the size of the desired output [3], even for $\text{Imp}_1$, which has only 17 productions, this approach would be practically impossible.

*Nullary productions.* One of the key innovations of our approach is that we synthesize the semantics on a per-production basis, i.e., working one production at a time. We start by synthesizing a semantics for nullary (leaf) productions. For $\text{Imp}_1$, this means we synthesize a semantics for the productions $\texttt{0}$, $\texttt{1}$, $\texttt{x}_1$, $\texttt{false}$, and $\texttt{true}$ before we synthesize the semantics of any other productions. For a nullary production $\texttt{p}$, we synthesize a semantics of the form:

$$
\frac{x_0^{out} = f(x_0^{in})}{Sem(\texttt{p}, x_0^{in}, x_0^{out})}
$$

which states that, because the term $\texttt{p}$ has no sub-terms, the output is only a function of the input $x_0^{in}$. In our approach, we use a Counter-Example-Guided Synthesis (CEGIS) approach to synthesize a function $f$ that captures the behavior of $\mathcal{I}_{\text{Imp}}$ on production $p$. Within the CEGIS loop, we synthesize a candidate function $f$, then verify if it is consistent with $\mathcal{I}_{\text{Imp}}$ (e.g., on a larger number of inputs

$x_0^{in}$). If $f$ is consistent, then we have successfully learned the semantics of $p$; otherwise, the verifier generates a counter-example and a new candidate semantic function $f$.

*Inductively synthesizing semantics.* Next, our approach synthesizes the semantics for other arithmetic and Boolean expressions. In this step, we inductively synthesize the semantics of productions by reusing the semantics of previously learned productions to learn the semantics of new productions. At this point, we may assume that we know the semantics of all nullary productions. For instance, suppose that we wish to next learn the semantics of `+`. At first, our algorithm generates examples favoring terms like `1 + 1`, `x + 1`, etc. that contains sub-terms whose semantics have already been learned. For $\texttt{t}_1$ `+` $\texttt{t}_2$, our algorithm generates a semantics that can rely on the semantics of its sub-terms $\texttt{t}_1$ and $\texttt{t}_2$. Specifically, the semantics of $\texttt{t}_1$ `+` $\texttt{t}_2$ takes the following form:

$$\frac{\begin{array}{ccc} sem(\texttt{t}_1, x_1^{in}, x_1^{out}) & & sem(\texttt{t}_2, x_2^{in}, x_2^{out}) \\ x_1^{in} = f_1(x_0^{in}) & x_2^{in} = f_2(x_0^{in}, x_1^{out}) & x_0^{out} = f_0(x_0^{in}, x_1^{out}, x_2^{out}) \end{array}}{sem(\texttt{t}_1 \texttt{ + } \texttt{t}_2, x_0^{in}, x_1^{out})}$$

which states that the semantics of $\texttt{t}_1$ `+` $\texttt{t}_2$ is inductively defined in terms of the semantics of $\texttt{t}_1$ and the semantics of $\texttt{t}_2$. The semantics enforces a left-to-right evaluation order:[1] the rule expresses that the input to $\texttt{t}_1$, $x_1^{in}$, is a function of $\texttt{t}_1$ `+` $\texttt{t}_2$'s input, $x_0^{in}$, and similarly that $\texttt{t}_2$'s input, $x_2^{in}$, is a function of $\texttt{t}_1$ `+` $\texttt{t}_2$'s input, $x_0^{in}$, and $\texttt{t}_1$'s output, $x_1^{out}$. Finally, it also expresses that the $\texttt{t}_1$ `+` $\texttt{t}_2$'s output, $x_0^{out}$, is a function of its input, $x_0^{in}$, and the outputs of $\texttt{t}_1$ ($x_1^{out}$) and $\texttt{t}_2$ ($x_2^{out}$).

When the semantics of a sub-term $\texttt{t}_i$ is known (e.g., for nullary productions), we substitute its learned semantics for $sem(\texttt{t}_i, x_i^{in}, x_i^{out})$; otherwise, we approximate its semantics using examples. Again, we use a CEGIS loop to generate examples for the entire term $\texttt{t}_1$ `+` $\texttt{t}_2$, as well as any sub-terms whose exact semantics have not yet been synthesized (e.g., for a sub-term that uses `+` or `−`). The process proceeds analogously for most other productions in Imp.

*Semantically recursive productions.* The final interesting case is for `while` loops, for which the semantics is recursive on the term itself. For semantically recursive productions, we assume that the semantics can make a recursive call (i.e., effectively acting as if the term itself is a sub-term). We additionally synthesize a predicate determining if the recursive call should be made or not. For `while b do s`, we synthesize two semantic rules, one in which the recursive call is made, and one in which it is not.

$$\frac{\begin{array}{ccc} sem(\texttt{b}, x_1^{in}, x_1^{out}) & sem(\texttt{s}, x_2^{in}, x_2^{out}) & \neg Pred_{rec}(x_0^{in}, x_1^{out}, x_2^{out}) \\ x_1^{in} = f_1(x_0^{in}) & x_2^{in} = f_2(x_0^{in}, x_1^{out}) & x_0^{out} = f_0(x_0^{in}, x_1^{out}, x_2^{out}) \end{array}}{sem(\texttt{while b do s}, x_0^{in}, x_1^{out})}$$

$$\frac{\begin{array}{cccc} sem(\texttt{b}, x_1^{in}, x_1^{out}) & sem(\texttt{s}, x_2^{in}, x_2^{out}) & sem(\texttt{while b do s}, x_3^{in}, x_3^{out}) & Pred_{rec}(x_0^{in}, x_1^{out}, x_2^{out}) \\ x_1^{in} = f_1(x_0^{in}) & x_2^{in} = f_2(x_0^{in}, x_1^{out}) & x_3^{in} = f_2(x_0^{in}, x_1^{out}, x_2^{out}) & x_0^{out} = f_0(x_0^{in}, x_1^{out}, x_2^{out}, x_3^{out}) \end{array}}{sem(\texttt{while b do s}, x_0^{in}, x_1^{out})}$$

As with the previous productions, our algorithm uses a CEGIS loop to synthesize a candidate semantics of the above form, verify its correctness, and generate a counter-example if the candidate semantics is incorrect. While we may employ learned semantics for sub-terms, recursive calls to a sub-term must be approximated using examples because we are still in the process of learning its semantics. We formally define the semantics-synthesis problem that we solve in Section 3 and explain how our synthesis algorithm works in Section 4.

*Multi-output productions.* In the above `while`-loop example, we saw that the function $f_0$ had four inputs that must be considered when synthesizing a term to instantiate $f_0$. As the number of input

[1] We show how to overcome this restriction in Section 5.1.

variables and the size of the desired result grows, synthesis scales poorly. In the above examples, the notation is not showing the full picture. For $\textsc{Imp}_n$ all input and (most) output variables are an $n$-tuple of variables representing a state of an $\textsc{Imp}_n$ program. Even for just $\textsc{Imp}_2$, $f_0$ has twice as many inputs.

To address this problem, we allow synthesizing the semantics of each output of a production independently. For example, consider the production $\texttt{x}_\emptyset \coloneqq \texttt{t}$ (for $\textsc{Imp}_2$). We generate a semantics using two constraints $F$ and $G$, independently. The constraint $F$ (resp. $G$) represents the pair of functions $f_0$ and $f_1$ (resp. $g_0$ and $g_1$).

$$\frac{sem(\texttt{t}, x_1^{in}, x_1^{in}) \qquad x_1^{in} = f_1(x_0^{in}) \qquad x_0^{out} = f_0(x_0^{in}, x_1^{out})}{sem(\texttt{x}_\emptyset \coloneqq \texttt{t}, x_0^{in}, x_0^{out})}\ F$$

$$\frac{sem(\texttt{t}, x_1^{in}, x_1^{in}) \qquad x_1^{in} = g_1(x_0^{in}) \qquad x_0^{out} = g_0(x_0^{in}, x_1^{out})}{sem(\texttt{x}_\emptyset \coloneqq \texttt{t}, x_0^{in}, x_0^{out})}\ G$$

By independently synthesizing $F$ and $G$, we reduce the burden on the underlying synthesizer; however, now the synthesizer is allowed to return an $F$ and $G$ for which $f_1 \neq g_1$. Thus, $F$ and $G$ have inconsistent inputs being provided to the child-term $t$. We use an SMT solver to determine if $f_1$ and $g_1$ are consistent for each of the example inputs to the term $\texttt{x}_\emptyset \coloneqq \texttt{t}$. If so, we will return either $f_0, g_0, f_1$ (or $f_0, g_0, g_1$ because $f_1$ and $g_1$ are consistent on all examples—i.e., when evaluated on the same example they return equal outputs—otherwise, we discover that $f_1$ and $g_1$ are inconsistent on some input and add a new constraint to ensure that the same pair of functions $f_1$ and $g_1$ cannot be synthesized again. This optimization is further discussed in Section 5.3.

## 3 Problem Definition

In this paper, we consider the problem of synthesizing a formal logical semantics for a deterministic language from an executable interpreter. While there are many possible ways to logically define a semantics, we are interested in an approach that is language-agnostic and inductive. The SemGuS synthesis framework has proposed using Constrained Horn Clauses as a way of defining program semantics that meets both of our desiderata. Concretely, SemGuS already supports synthesis for a large number of languages (which we consider in our experimental evaluation) by allowing a user to provide a user-defined semantics. As mentioned above, in SemGuS, semantics are defined inductively on the structure of the grammar (i.e., per production/language construct) using logical relations represented as Constrained Horn Clauses (CHCs) [13]. In this paper, we follow suit and address the problem of learning a semantics of this form from an executable interpreter for the given language. This section formalizes the semantics-synthesis problem that we consider. We begin by detailing our representation of syntax (Section 3.1), interpreters (Section 3.2), semantics (Section 3.3), and semantic-equivalence oracles (Section 3.4). Finally, we formalize the semantics-synthesis problem in Section 3.4.

### 3.1 Syntax

We consider languages represented as regular tree grammars (RTGs). A **ranked alphabet** is a tuple $\langle \Sigma, rk_\Sigma \rangle$ that consists of a finite set of symbols $\Sigma$ and a function $rk_\Sigma : \Sigma \to \mathbb{N}$ that associates every symbol with a rank (or arity). For any $n \geq 0$, $\Sigma^n \subseteq \Sigma$ denotes the set of symbols of rank $n$. The set of all *(ranked) Trees* over $\Sigma$ is denoted by $T_\Sigma$. Specifically, $T_\Sigma$ is the least set such that $\Sigma^0 \subseteq T_\Sigma$ and if $\sigma^k \in \Sigma^k$ and $t_1, \ldots, t_k \in T_\Sigma$, then $\sigma^k(t_1, \ldots, t_k) \in T_\Sigma$. In the remainder of the paper, we assume a fixed ranked alphabet $\langle \Sigma, rk_\Sigma \rangle$.

A **typed regular tree grammar** (RTG) is a tuple $G = \langle N, \Sigma, \delta, \mathrm{T}, \theta, \tau \rangle$, where $N$ is a finite set of non-terminal symbols of rank 0, $\Sigma$ is a ranked alphabet, $\delta$ is a set of productions over a set of types T, and for each non-terminal $A \in N$, and $\theta_A$ (resp. $\tau_A$) assigns $A$ an input-type (resp. output-type) from T. Each production in $\delta$ takes the form:

$$A_0 \rightarrow \sigma\left(A_1, A_2, \ldots, A_{rk_\Sigma(\sigma)}\right)$$

where $A_i \in N$ and $\sigma \in \Sigma$. We use $\mathcal{L}(A)$ to denote the language of non-terminal $A$ and $\delta(A)$ the set of all productions associated with $A$ (i.e., all productions where $A_0$ is $A$). In the remainder, we assume a fixed grammar $G = \langle N, \Sigma, \delta, \mathrm{T}, \theta, \tau \rangle$.

*Example 3.1 ($G_{\text{IMP}}$ as a Regular Tree Grammar).* Consider the IMP language detailed in Section 2, $G_{\text{IMP}}$ is a regular tree grammar that has been stylized to ease readability. For example, the non-terminals consist of the rank-0 symbols $E$, $B$, and $S$. The productions include $S \rightarrow$ `x1 :=`$(E)$, $S \rightarrow$ `;`$(S, S)$, and $S \rightarrow$ `while`$(B, S)$. For IMP$_2$ (IMP with two variables $x_1$ and $x_2$), $\theta_E$ is the type $\mathbb{Z} \times \mathbb{Z}$, representing the state of the two variables, and $\tau_E$ is $\mathbb{Z}$, representing the return type of arithmetic expressions.

### 3.2 Interpreters

We consider a class of deterministic executable interpreters—i.e., a program evaluator for which we may only observe input-output behavior.

*Definition 3.2 (Interpreter).* Formally, an *interpreter* for $G$ maps each non-terminal $A \in N$ to a partial function $\mathcal{I}_A : (\mathcal{L}(A) \times \theta_A) \rightarrow \tau_A$—with the interpretation that the interpreter maps a program $t \in \mathcal{L}(A)$ and input value $in \in \theta_A$ to some output $out \in \tau_A$ if and only if $t$ starting with the input value $in$ terminates with the output value $out$.

*Example 3.3 (Interpreters for IMP$_1$).* Recall the IMP language defined in Section 2. The interpreter $\mathcal{I}$ for IMP consists of three base interpreters $\mathcal{I}_E$, $\mathcal{I}_B$, and $\mathcal{I}_S$, which are used to evaluate arithmetic expressions, Boolean expressions, and statements, respectively. Throughout this paper, we assume the interpreters for IMP$_1$ (and all IMP variants) evaluate according to the standard denotational semantics (e.g., `0` is the expression that always returns 0 regardless of input state; `+` is mathematical +; `while b s` evaluates $b$, executes the loop body, and recurses if `b` evaluates to *true* and otherwise immediately terminates; etc.).

### 3.3 Semantics

We represent the big-step semantics of a language (defined by some grammar $G$) using a set of Constrained Horn Clauses (CHCs) within some background theory $\mathcal{T}$ per production. While CHCs (at first glance) seem limiting, this formulation of semantics has been employed by the SemGuS framework to represent user-defined semantics for many languages [6, 13], including many variations of IMP, regular expressions, SyGuS expressions within the theory of bit vectors, algebraic data types, linear integer arithmetic.

*Definition 3.4 (Constrained Horn Clause).* A CHC (in theory $\mathcal{T}$) is a first-order formula of the form:

$$\forall \bar{x}_1, \ldots, \bar{x}_n, \bar{x}.\ \phi \wedge R_1(\bar{x}_1) \wedge \cdots \wedge R_n(\bar{x}_n) \Rightarrow H(\bar{x})$$

where $R_1, \ldots, R_n$ and $H$ are uninterpreted relations, $\bar{x}_1, \ldots, \bar{x}_n$ and $\bar{x}$ are variables, and $\phi$ is a quantifier-free $\mathcal{T}$-constraint over the variables.

To specify the big-step semantics of a non-terminal $A \in N$ (for which the interpreter has type $\mathcal{I}_A : (\mathcal{L}(A) \times \theta_A) \rightarrow \tau_A$), we introduce the semantic relation $Sem_A(t_A, x_A^{in}, x_A^{out})$, where $t_A$ is a

variable representing elements of $\mathcal{L}(A)$, $x_A^{in}$ is a variable of type $\theta_A$, and $x_A^{out}$ is a variable of type $\tau_A$. Throughout this paper, we may also use $[\![t_A]\!]_{Sem}(x_A^{in}) = x_A^{out}$ to denote that $Sem_A(t_A, x_A^{in}, x_A^{out})$ holds.

*Example 3.5 (Semantic relations).* Consider the $\text{IMP}_1$ language introduced in Section 2; a semantics for $\text{IMP}_1$ uses the semantic relations:

$$Sem_E : \mathcal{L}(E)\times\mathbb{Z}\times\mathbb{Z} \to \texttt{bool} \qquad Sem_B : \mathcal{L}(B)\times\mathbb{Z}\times\texttt{bool} \to \texttt{bool} \qquad Sem_S : \mathcal{L}(S)\times\mathbb{Z}\times\mathbb{Z} \to \texttt{bool}$$

While CHCs are quite general and capable of defining both deterministic and non-deterministic semantics, we limit our scope to CHCs that represent deterministic semantics. Furthermore, for a grammar $G$, we assume that each production $A_0 \to \sigma(A_1, \ldots, A_n) \in G$ evaluates sub-terms in a fixed order from left to right (i.e., for a term $p(t_1, \ldots, t_n)$ sub-term $t_1$ is evaluated before $t_2$, etc.). While this imposed order may seem too restrictive, we later show how this restriction can be lifted by considering all permutations of sub-terms.

*Definition 3.6 (Semantic Rule, Semantic Constraint).* Given a production $A_0 \to p(A_1, \ldots, A_n)$ a **semantic rule** for $p$ is a CHC of the form:

$$\frac{Sem_{A_1}(t_1, x_1^{in}, x_1^{out}) \quad \ldots \quad Sem_{A_n}(t_n, x_n^{in}, x_n^{out}) \quad F(x_0^{in}, \ldots, x_n^{in}, x_0^{out}, \ldots, x_n^{out})}{Sem_{A_n}(p(t_1, \ldots, t_n), x_0^{in}, x_0^{out})} \tag{1}$$

where $F$ is constraint over theory $\mathcal{T}$, which we call a **semantic constraint**, that takes the form:

$$x_1^{in} = f_1(x_0^{in}) \wedge \cdots \wedge x_n^{in} = f_n(x_1^{out}, \ldots, x_{n-1}^{out}, x_0^{in}) \wedge x_0^{out} = f_0(x_1^{out}, \ldots, x_n^{out}, x_0^{in}) \wedge P(x_0^{in}, x_0^{out}, \ldots, x_n^{out}) \tag{2}$$

where each $f_i$ is a function that returns a term of type $\theta_{A_i}$ for $i > 0$ and $\tau_{A_0}$ for $i = 0$. The semantic constraint also includes predicate $P(x_{A_0}^{in}, x_{A_1}^{out}, \ldots, x_{A_n}^{out})$ that determines when the semantic rule is valid (e.g., for conditionals and loops).

*Example 3.7 (Semantics of* `do_while`*).* We give the semantics of the `do_while` IMP statement below:

$$\frac{[\![b]\!](x_2) = r_b \qquad [\![\texttt{do } s \texttt{ while } b]\!](x_3) = x_3' \qquad \begin{array}{c}[\![s]\!](x_1) = x_1' \\ r_b\end{array} \qquad x_1 = x_0 \qquad x_2 = x_1' \qquad x_3 = x_1' \qquad x_0' = x_3'}{[\![\texttt{do } s \texttt{ while } b]\!](x_0) = x_0'}$$

$$\frac{[\![s]\!](x_1) = x_1' \qquad [\![b]\!](x_2) = r_b \qquad \neg r_b \qquad x_1 = x_0 \qquad x_2 = x_1' \qquad x_0' = x_1'}{[\![\texttt{do } s \texttt{ while } b]\!](x_0) = x_0'}$$

The first rule executes the statement $s$ and then, if the guard $b$ is true recursively executes the whole loop and returns the resulting value. The second rule executes the statement $s$ and then, if the guard $b$ is false returns the output produced when executing the statement $s$.

### 3.4 Equivalence Oracle and Semantics Synthesis Problem

For a grammar $G$, a semantics $Sem$ for $G$, and an interpreter $\mathcal{I}$ for $G$, an *equivalence oracle* is used to determine whether $Sem$ is equivalent to the semantics defined by the interpreter $\mathcal{I}$.

*Definition 3.8 (Equivalent, Equivalence Oracle).* Given an interpreter $\mathcal{I}$ for a language $G$, a subgrammar $G' \subseteq G$, and a semantics $Sem$ for $G'$, we say that $\mathcal{I}$ and $Sem$ are **equivalent** on $G'$ if and only if for every term $t \in \mathcal{L}(G')$, input $in \in \theta_A$, and output $out \in \tau$, we have:

$$I(t, in) = out \Leftrightarrow [\![t]\!]_{Sem}(in) = out$$

An **equivalence oracle** $\mathcal{E}$ for $\mathcal{I}$ is a function that takes as input a semantics $Sem$ for $G'$ and determines if $Sem$ is equivalent to $\mathcal{I}$ on $G'$. If $Sem$ is not equivalent to $\mathcal{I}$, then $\mathcal{E}$ returns an

example $\langle in, t, out \rangle$ for which $\mathcal{I}$ and *Sem* disagree—i.e., there is some term $t$ and input $in$ such that $[\![t]\!]_{Sem}(in) \neq [\![t]\!]_{\mathcal{I}}(in)$—and otherwise returns *None* when *Sem* and $\mathcal{I}$ are equivalent.

Given a language (a grammar and accompanying interpreter), the semantics synthesis problem is to find some semantics of the language that is equivalent to the interpreter. We formalize the semantics synthesis problem as follows:

*Definition 3.9 (Semantics-Synthesis Problem, Solution).* A **semantics-synthesis problem** is a tuple $\mathcal{P} \triangleq \langle G, \mathcal{I}, \mathcal{E} \rangle$, where $G$ is a grammar, $\mathcal{I}$ is an interpreter for $G$, and $\mathcal{E}$ is an equivalence oracle for $\mathcal{I}$. A **solution** to the semantics-synthesis problem $\mathcal{P}$ is a semantics *Sem* for $G$ that is equivalent to $\mathcal{I}$ as determined by $\mathcal{E}$.

## 4 Semantics Synthesis

This section presents an algorithm SemSynth (Algorithm 1) to synthesize a semantics for a language from an executable interpreter. The input to SemSynth is a semantics-synthesis problem consisting of (*i*) a grammar $G$, (*ii*) an executable interpreter $\mathcal{I}$ for $G$, and (*iii*) an equivalence oracle $\mathcal{E}$ for $\mathcal{I}$. Upon termination, SemSynth returns a semantics *Sem* for $G$ that is equivalent to the executable interpreter $\mathcal{I}$ as determined by the equivalence oracle $\mathcal{E}$.

Synthesizing a semantics for arbitrary languages comes with several challenges. In general, semantics are defined as complex recursively defined functions that provide an interpretation to every program within the language. Trying to directly synthesize such a semantics is already impractical for relatively small languages, such as the Imp language defined in Example 2.1.

As described in Section 3.3, we consider semantics represented using logical relations defined by a set of Constrained Horn Clauses per production of $G$ (cf. Definition 3.6). By formulating the desired semantics as CHCs per production, SemSynth can synthesize the semantics of $G$ one production at a time. In fact, because SemSynth uses examples to approximate the semantics of all sub-terms during synthesis (cf. Section 4.1), SemSynth can synthesize the semantics of each production independently. Finally, by fixing the shape of the semantics (i.e., as a set of CHCs per production), SemSynth reduces the monolithic synthesis problem to a series of first-order synthesis problems—specifically, by using a SyGuS or sketch-based synthesizer to synthesize the constraint of each semantic rule (CHC) defining the semantics of a production.

The remainder of this section is structured as follows: Section 4.1 provides a high-level overview of how SemSynth solves semantic-synthesis problems, Sections 4.2 and 4.3 provide specifications for SynthSemanticConstraint and Verify, which synthesize semantic constraints from examples and verify candidate semantic constraints against the interpreter, respectively. Finally, Section 4.4 explains how SemSynth handles semantically recursive productions.

### 4.1 Overview of SemSynth

SemSynth (Algorithm 1) uses the counter-example-guided synthesis (CEGIS) paradigm to synthesize a semantics for $G$ that is equivalent to $\mathcal{I}$ according to the equivalence oracle $\mathcal{E}$. Throughout this section, we will use the Imp language from Example 2.1 to illustrate how SemSynth operates.

*Synthesizing a Candidate Semantics.* After initialization, SemSynth synthesizes the semantics of each production. SemSynth employs a CEGIS loop to synthesize the semantics of each production. During each iteration, SemSynth first synthesizes a candidate semantic constraint (cf. Definition 3.6) for production $p$ using SynthSemanticConstraint. The procedure SynthSemanticConstraint returns some semantic constraint for $p$ that satisfies the set of examples $E$. Section 4.2 provides a formal specification of SynthSemanticConstraint's operation.

**Algorithm 1:** Semantics-Synthesis Algorithm

```
1  Procedure SemSynth (G, ℐ, ℰ)
2      foreach production p of G do
3          E ← ∅ ;                                   // Example Set for Production p
4          do
5              Sem[p] ← SynthSemanticConstraint(p, E) ;   // Get candidate semantics
6              CEX ← Verify(Sem[p], p, ℐ, ℰ) ;            // Check candidate semantics
7              if CEX ≠ ∅ then
8                  E ← E ∪ CEX ;                          // Update example set
9          while CEX ≠ ∅;
10     return Sem;
```

SemSynth then uses the procedure Verify to determine if the semantics synthesized for production $p$ is consistent with the interpreter $\mathcal{I}$ as determined by the equivalence oracle $\mathcal{E}$. A formal specification of Verify is provided in Section 4.3. If Verify determines that the candidate semantics of $p$ is correct, then Verify returns an empty set of examples and SemSynth proceeds to synthesize the semantics of the next production. Otherwise, if Verify determines that the candidate semantics of $p$ is not equivalent to the interpreter $\mathcal{I}$, Verify returns a set of examples. The new examples are added to the example set $E$, and the CEGIS loop repeats and synthesizes a new candidate semantics for $p$.

### 4.2 Specification of SynthSemanticConstraint

Before formally specifying SynthSemanticConstraint (Section 4.2.3), we first define example sets (Section 4.2.1) and when a semantic constraint is consistent with an example set (Section 4.2.2).

*4.2.1 Example Sets.* For an interpreter $\mathcal{I}$, an example set $E$ is a set of examples consistent with $\mathcal{I}$.

*Definition 4.1 (Example set for interpreter $\mathcal{I}$).* Given an interpreter $\mathcal{I}$ for grammar $G$, an example set $E$ for interpreter $\mathcal{I}$ is a finite set of examples of the form $\langle in, t, out\rangle$, where $t \in L(G)$ and $\mathcal{I}(t, in) = out$.

*Example 4.2 (Example set for Imp$_1$).* Recall the interpreter $\mathcal{I}_{\text{Imp}_1}$ described in Example 3.3 for language Imp$_1$. An example set $E$ for $\mathcal{I}_{\text{Imp}_1}$ might include the examples $\langle 0, \texttt{0}, 0\rangle$, $\langle 1, \texttt{0}, 0\rangle$, $\langle 1, \texttt{x} \coloneqq \texttt{0;x} \coloneqq \texttt{x} + \texttt{4}, 4\rangle$, and $\langle 10, \texttt{while 0} < \texttt{x do x} \coloneqq \texttt{x} - \texttt{1}, 0\rangle$; however, an example set for $\mathcal{I}_{\text{Imp}}$ could not include any example of the form $\langle n, \texttt{while 0} < \texttt{x do x} \coloneqq \texttt{x} + \texttt{1}, n'\rangle$ where $n$ (the initial value of x) is some positive number. Since, $\texttt{while 0} < \texttt{x do x} \coloneqq \texttt{x} + \texttt{1}$ would not terminate on the input $n$. The example $\langle n, \texttt{while 0} < \texttt{x do x} \coloneqq \texttt{x} + \texttt{1}, n'\rangle$ would violate the assumption that $E$ only contains examples consistent with the interpreter $\mathcal{I}_{\text{Imp}}$.

*4.2.2 Example Consistency.* In SemSynth, we use the example set $E$ to ensure that the semantic constraint returned by SynthSemanticConstraint is consistent with $\mathcal{I}$ for at least the examples appearing in $E$.

*Definition 4.3 (Consistency with Example Set).* Given a production $A_0 \to p(A_1, \ldots, A_n)$, a semantic rule $R$ with semantic constraint $F$ of the form defined in Definition 3.6, and example set $E$, we say $R$ is **consistent** with $E$ if and only if the semantic constraint $F$ is consistent with $E$. Furthermore, the semantic constraint $F$ is **consistent** with the example set $E$ if for every example

$\left\langle in_{A_0}, p(t_1, \ldots, t_n), out_{A_0} \right\rangle \in E$ the following condition holds:

$$\forall x_0^{in}, \ldots, x_n^{in}, x_0^{out}, \ldots, x_n^{out}. \left( \begin{array}{l} x_0^{in} = in_0 \\ \wedge\ Summary(t_1) \\ \ldots \\ \wedge\ Summary(t_n) \\ \wedge\ F \end{array} \right) \Rightarrow x_0^{out} = out_0 \tag{3}$$

where $Summary(t_i) = \bigvee \{ x_i^{in} = in_i \wedge x_i^{out} = out_i : \langle in_i, t_i, out_i \rangle \in E \}$ summarizes the semantics of $t_i$ according to the examples found in $E$.

*Example 4.4 (Example Consistency).* Consider the production for the operator +, and the (correct) semantic constraint $F \triangleq x_1^{in} = x_0^{in} \wedge x_2^{in} = x_0^{in} \wedge x_0^{out} = x_1^{out} + x_2^{out}$; $F$ is consistent with the examples $\langle 0, \mathtt{x_0} + 1, 1 \rangle$, $\langle 0, \mathtt{x_0}, 0 \rangle$, and $\langle 0, 1, 1 \rangle$. Specifically, the following formula is valid:

$$\forall x_0^{in}, x_1^{in}, x_2^{in}, x_0^{out}, x_1^{out}, x_2^{out}. (x_0^{in} = 0 \wedge (x_1^{in} = 0 \wedge x_1^{out} = 0) \wedge (x_1^{in} = 0 \wedge x_1^{out} = 1) \wedge F) \Rightarrow x_0^{out} = 1.$$

*4.2.3 Formal Specification of SynthSemanticConstraint.* The procedure SynthSemanticConstraint takes as input the production $p$ whose semantics is to be synthesized and the current example set $E$; it returns a constraint $F$—of the form defined in Definition 3.6—defining a semantics for production $p$ that is consistent with the example set $E$.

*Example 4.5 (Synthesizing semantics of $\mathtt{x} \coloneqq$ consistent with examples).* Recall that for the language Imp, the semantics of the production $\mathtt{x} \coloneqq$ is represented as (a set of) CHC rule(s) of the form:

$$\frac{Sem_E(e, x_1^{in}, x_0^{out}) \wedge x_1^{in} = f(x_0^{in}) \wedge x_0^{out} = g(x_0^{in}, x_1^{out})}{Sem_S(\mathtt{x} \coloneqq e, x_0^{in}, x_0^{out})}$$

for some functions $f$ and $g$ (in the theory of linear integer arithmetic). The procedure call SynthSemanticConstraint($\mathtt{x} \coloneqq, E$) synthesizes the formulas $f(x_0^{in}) = t_f$ and $g(x_0^{in}, x_1^{out}) = t_g$, and returns the constraint $F \triangleq x_1^{in} = t_f \wedge x_0^{out} = t_g$ so that $F$ is consistent with $E$.

We note that for functions expressible in a decidable first-order theory, this problem can be exactly encoded as a Syntax-Guided Synthesis (SyGuS) problem [2] and solved by a SyGuS solver (e.g., cvc5 [4]).

### 4.3 Specification of Verify

The procedure Verify takes as input the production $p$, a candidate semantics of $p$, the interpreter $\mathcal{I}$, and the equivalence oracle $\mathcal{E}$; it determines if *Sem* is equivalent to the interpreter $\mathcal{I}$ for all terms of the form $p(t_1, ..., t_k) \in L(G)$. If Verify determines that the candidate semantics of $p$ is not equivalent to $\mathcal{I}$, Verify returns a set of counter-examples *CEX* such that (*i*) *CEX* is consistent with $\mathcal{I}$, (*ii*) *CEX* is not consistent with the candidate semantics of production $p$, and (*iii*) for the input production $p$, there is exactly one example of the form $\langle i, p(t_1, \ldots, t_k), o \rangle$ appearing in *CEX* (for any other production $p' \neq p$, there can be many examples of the form $\langle i', p'(t_1, \ldots, t_k), o' \rangle$ in *CEX*). Otherwise, Verify returns an empty-set to signify that the semantics of $p$ is equivalent to $\mathcal{I}$ for all terms of the form $p(t_1, \ldots, t_k) \in L(G)$.

*Example 4.6 (Synthesizing Semantics of $\mathtt{0}$ for $G_{Imp}$).* Recall the Imp language in Example 2.1. On some iterations, SemSynth will consider the production $\mathtt{0}$ (a leaf/nullary production). During the first iteration of the CEGIS loop for $\mathtt{0}$, the example set $E$ will be empty and SynthSemanticConstraint may return any constraint $F$ of the form $x_0^{out} = f(x_0^{in})$. Assume that SynthSemanticConstraint returns the constraint $x_0^{out} = 1$. Verify returns the counter-example $\langle 0, \mathtt{0}, 0 \rangle$, and the example set $E$ is updated.

In the next iteration, the CEGIS loop must return a constraint satisfying the updated example set. For example, suppose that SynthSemanticConstraint returns the constraint $x_0^{out} = x_0^{in}$. Again, Verify determines that $x_0^{out} = x_0^{in}$ is incorrect and returns the new counter-example $\langle 1, \texttt{0}, 0\rangle$. The example set $E$ is updated with the returned counter-example.

A new iteration of the loop is run. On this iteration, SynthSemanticConstraint must return a constraint that satisfies both of the previously returned examples. This time SynthSemanticConstraint returns the constraint $x_0^{out} = 0$, Verify determines that $x_0^{out} = 0$ is correct, and SemSynth proceeds to synthesize the semantics of the next production (e.g., 1).

In Example 4.6, we see how SemSynth handles nullary (leaf) productions. SemSynth works nearly identically for most production rules (excluding semantically recursive productions like `while` loops). We demonstrate in Example 4.7 how SemSynth synthesizes a semantics for non-nullary productions.

*Example 4.7 (Synthesizing Semantics of Sequencing for Imp.).* Continuing from Example 4.6, SemSynth proceeds and comes to the sequencing operator (i.e., for production $S \rightarrow\ ;(S, S)$). After several attempts at synthesizing the semantics of sequencing, $E$ contains the examples $\langle 0, \texttt{x} \coloneqq \texttt{1}; \texttt{x} \coloneqq \texttt{0}, 0\rangle$, $\langle 0, \texttt{x} \coloneqq \texttt{0}; \texttt{x} \coloneqq \texttt{x} + \texttt{1}, 1\rangle$, and $\langle 1, \texttt{x} \coloneqq \texttt{0}; (\texttt{x} \coloneqq \texttt{1}; \texttt{x} \coloneqq \texttt{x} + \texttt{1}), 2\rangle$.

In addition to these examples, we summarize the semantics of each example's sub-term with further examples in the example set $E$. These summarized examples of sub-terms are generated by data-flow propagation through the term $p(t_1, \ldots, t_k)$ using the input $i$. Because the execution output of a certain sub-term $t_j$ can be used as input for any following term $t_l$ where $l > j$, we repeatedly enumerate all possible inputs for each sub-term (and add them into $E$) until we reach a fix-point, i.e., no new examples for sub-terms are found. SemSynth then generates the formula specifying that the desired semantic constraint is consistent with the example set $E$ using the generated summaries, and produces a new semantic constraint using SynthSemanticConstraint. On this iteration, SynthSemanticConstraint returns the correct semantic constraint, Verify determines that it is correct, and SemSynth proceeds to synthesize a semantics for the next production.

### 4.4 Synthesizing Semantics for Semantically Recursive Productions

So far, we have seen how SemSynth handles nullary productions and structurally recursive productions (e.g., `ite` and sequencing). However, we have not yet seen how to handle productions that are *semantically* recursive (e.g., `while` loops). To handle semantically recursive productions, we augment the form of the desired constraint to be synthesized: SynthSemanticConstraint must synthesize a predicate $P_{rec}$ and two base constraints $F_{nonrec}$ and $F_{rec}$ such that for every example $\langle in, p(t_1, \ldots, t_n), out\rangle$, the following conditions hold:

$$\frac{\ldots \quad Sem_{A_n}(t_n, x_{A_n}^{in}, x_{A_n}^{out}) \quad \neg P_{rec}(x_{A_0}^{in}, x_{A_1}^{out}, \ldots, x_{A_n}^{out}) \quad Sem_{A_1}(t_1, x_{A_1}^{in}, x_{A_1}^{out}) \quad F_{non-rec}(x_{A_0}^{in}, x_{A_1}^{out}, \ldots, x_{A_n}^{out}) \quad x_{A_0}^{in} = in}{x_{A_0}^{out} = out}\ \text{NON-REC}$$

$$\frac{Sem_{A_0}(p(t_1, \ldots, t_n), x_{A_0}^{in\prime}, x_{A_0}^{out\prime}) \quad P_{rec}(x_{A_0}^{in}, x_{A_1}^{out}, \ldots, x_{A_n}^{out}) \quad Sem_{A_1}(t_1, x_{A_1}^{in}, x_{A_1}^{out}) \quad \ldots \quad Sem_{A_n}(t_n, x_{A_n}^{in}, x_{A_n}^{out}) \quad F_{rec}(x_{A_0}^{in}, x_{A_1}^{out}, \ldots, x_{A_n}^{out}) \quad x_{A_0}^{in} = in}{x_{A_0}^{out} = out}\ \text{REC}$$

where $P_{rec}$ determines if the non-rec or rec condition should hold. The non-recursive case is similar to the conditions for non-semantically recursive statements (with the addition of asserting that $P_{rec}$ is false). The recursive case, however additionally allows the semantics to make use of a recursive call to the program term. Other than the change in the shape of the desired semantics, SemSynth remains unchanged.

*Example 4.8 (Synthesizing semantics of while loops for Imp.).* Continuing from Example 4.7, SemSynth eventually considers the `while` production. We assume that the grammar $G$ additionally annotates whether each production is semantically recursive.

After several iterations of the CEGIS loop, the example set $E$ contains the examples $\langle 0, t, 0\rangle$, $\langle 1, t, 0\rangle$, and $\langle 2, t, 0\rangle$, where $t$ is the term `while 0 < x do x := x − 1`. In this iteration, SynthSemanticConstraint gets called with a recursive summary of $t$ containing the three examples, and examples for `x := x − 1` and `0 < x`.

In this iteration, SynthSemanticConstraint finds the correct $P_{rec}$, $F_{non-rec}$ and $F_{rec}$. Verify determines that the result is indeed correct and the main loop of SemSynth continues to the next production. If `while` is the last production of the considered grammar $G$, then SemSynth terminates and returns the synthesized semantics for each production.

Now that we have defined how SemSynth handles semantically recursive productions, SemSynth is fully specified. Theorem 4.9 states that SemSynth is sound.

Theorem 4.9 (SemSynth is sound). *For any semantics-synthesis problem $\mathcal{P} = \langle G, \mathcal{I}, \mathcal{E}\rangle$, if SemSynth$(G, \mathcal{I}, \mathcal{E})$ returns a semantics Sem, then Sem is a solution to $\mathcal{P}$.*

Proof. SemSynth iterates over the productions in some order, say $p_0, \ldots, p_{k-1}$. For all iterations $0 \leq i \leq k$, SemSynth maintains the invariant that the synthesized semantics *Sem* is correct with respect to the oracle $\mathcal{E}$ for all previously considered productions $p_0$ through $p_{i-1}$. This condition trivially holds on the first iteration. To proceed to iteration $i + 1$, the CEGIS loop for production $p_i$ must terminate. For the CEGIS loop to terminate, Verify must return an empty set of counter-examples, which implies that *Sem* is correct for the production $p_i$ (and that the semantics for productions $p_1, \ldots, p_{i-1}$ were left unmodified)—and thus the invariant is maintained. The algorithm only terminates after exploring all productions. Consequently, upon termination, *Sem* must be correct for all productions of $G$—i.e., *Sem* satisfies the given semantics-synthesis problem $\mathcal{P}$. □

While Theorem 4.9 states the soundness of SemSynth, it fails to show that SemSynth will eventually synthesize a correct semantics. Theorem 4.10 states that SemSynth makes progress. Intuitively, it states that once a semantic rule for production $p$ is explored during some iteration of the CEGIS loop, it is never explored in any future iteration of the CEGIS loop for production $p$.

Theorem 4.10 (SemSynth makes progress). *For any semantics-synthesis problem $\mathcal{P} = \langle G, \mathcal{I}, \mathcal{E}\rangle$, if SemSynth$(G, \mathcal{I}, \mathcal{E})$ is synthesizing the semantics of production $p$ and on the $k^{th}$ iteration of the CEGIS loop for production $p$, SynthSemanticConstraint produces the semantic relation $R_k$, then for all future iterations $j > k$, SynthSemanticConstraint will return some relation $R_j \neq R_k$.*

Proof. Assume that the negation holds, i.e., "$\exists j > k.R_j = R_k$". By the assumption $j > k$, it must be that Verify $(R_k, p, \mathcal{I}, \mathcal{E})$ returned a non-empty set of examples *CEX*. Otherwise, the CEGIS loop for production $p$ would have immediately terminated and not continued to iteration $j$. By definition, $R_k$ is inconsistent with the set of counter-examples *CEX*. The returned counter-examples *CEX* are then added to the example set $E$ for all future iterations. By assumption, $R_j$ must be consistent with the example set $E$, and thus $R_j$ must not be $R_k$, a contradiction. □

## 5 Implementation

This section gives details of Synantic, which implements our approach to synthesizing semantics via the algorithm SemSynth. Synantic is developed in Scala (version 2.13), and uses cvc5 (version 1.0.3) to solve SyGuS problems—which are used within our implementation of SynthSemantic-Constraint to generate candidate semantic constraints. The remainder of this section is structured

as follows: Section 5.1 details how we implement SynthSemanticConstraint. Section 5.2 summarizes the implementation of Verify, and explains how we approximate an equivalence oracle for an interpreter. Section 5.3 presents an optimization of SynthSemanticConstraint for productions with multiple outputs (i.e., where the output type of a production is a tuple).

### 5.1 Implementation of SynthSemanticConstraint

In Section 4, SemSynth is parameterized on the procedure SynthSemanticConstraint. On line 5 of Algorithm 1, we assume that SynthSemanticConstraint produces a semantic constraint $F$ for production $p_i$ that satisfies the example set $E$. To accomplish this task, we construct a SyGuS problem consisting of a grammar of allowable semantic constraints and a set of conditions to enforce that the semantic constraint is consistent with the example set. To handle productions whose semantics does not evaluate its child terms from left to right, we run in parallel a version of SynthSemanticConstraint for each permutation of the child terms and immediately return upon any permutation's success. In practice, for all of our benchmarks, all the productions evaluate their children from left to right.

We defer discussion of the SyGuS grammars we use to Section 6.1 when we discuss each benchmark. The specification of the semantic constraint is exactly the condition specified in Equation (3).

### 5.2 Implementation of Verify

In Section 4, Algorithm 1 is parameterized on the procedure Verify (line 6), which uses the equivalence oracle $\mathcal{E}$ to determine if the learned semantics *Sem* is consistent with the interpreter for all terms of the form $p(t_1, \ldots, t_k) \in L(G)$ for some production $p$. In Synantic, we approximate an equivalence oracle using fuzzing. Specifically, we randomly generate terms and inputs and use the interpreter $\mathcal{I}$ to generate an output. We then use the learned constraint for $p_i$ to generate inputs to each sub-term (from left to right), and compute outputs for each using interpreter $\mathcal{I}$. In effect, we are computing a new example set $E'$, and testing the semantic constraints learned so far. If any example disagrees with the learned semantics of production $p$, we return the example (and necessary child-term summaries) as a counter-example.

When Verify fuzzes the semantics, it uses the interpreter to generate examples (i.e., terms with corresponding input-output examples). During example generation, we set a recursion limit of 1,000 recursive calls. We discard an example—i.e., we assume the program does not terminate—if its run exceeds the recursion depth. We then evaluate the candidate semantic constraint from left-to-right to ensure that the semantic constraint is consistent with each of the generated examples. We return the first example (and the child-term summaries for the example) that is inconsistent with the candidate constraint.

### 5.3 Optimized SynthSemanticConstraint for Multi-Output Productions

In Section 5.1, we described how SynthSemanticConstraint produces and solves (using cvc5) a SyGuS problem to synthesize a semantic constraint that is consistent with the current example set. However, it is well known that SyGuS solvers scale poorly as a function of the size of the desired grammar/result. This issue is especially problematic when learning a semantic constraint for a language in which productions have multiple outputs (e.g., statements for Imp with more than one variable) and thus the grammar and resulting constraint grow with the number of outputs.

For some languages, it is possible to augment the semantics of the language to use a suitable theory to encode multiple outputs as a single output—e.g., using the theory of arrays to support multi-variable states in the ImpArr language (cf. Section 6.1). However, for other languages this methodology may require the use of theories that are not well suited for existing SyGuS and

**Algorithm 2:** Verifier implementation using (approximate) fuzzing based oracle.

**Procedure** *Verify* $(R_p, p, \mathcal{I}, \cdot)$
  $E \leftarrow$ random set of examples of the form $\langle in, p(t_1, \ldots, t_k), out\rangle$ consistent with $\mathcal{I}$ ;
  $(\bigwedge_i x_i^{in} = f_i) \wedge x_i^{out} = f_0 \leftarrow R_p$ ; // Destruct semantic constraint to recover each $f_i$
  **for** $\langle in, p(t_1, \ldots, t_k), out\rangle \in E$ **do**
    $E' \leftarrow \langle in, p(t_1, \ldots, t_k), out\rangle$;
    $M \leftarrow \{x_0^{in} \mapsto in\}$ ; // Build up model to evaluate sub-terms's semantics
    **for** $i \leftarrow 1$ *to* $k$ **do**
      $in_i \leftarrow [\![f_i]\!]_M$ ; // Get input to term $t_i$.
      $out_i \leftarrow \mathcal{I}(t_i, in_i)$ ; // Evaluate term $t_i$
      $M \leftarrow M[x_i^{out} \mapsto out_i]$ ; // Update model. Ensures next term's input is defined.
      $E' \leftarrow E' \cup \{\langle in_i, t_i, out_i\rangle\}$ ; // Add sub-term's summary to set of examples
    **if** $[\![f_0]\!]_M \neq out$ **then**
      **return** $E'$ ; // The output computed by evaluating $R_p$ is inconsistent with $E'$
  **return** $\emptyset$

SMT solvers (e.g., RegEx($k$) in Section 6.1 would require the theory of strings). Instead, for such instances we developed a variant of SynthSemanticConstraint that synthesizes a constraint for each output independently. However, this process may lead to constraints that do not agree on the internal data flow of the constraints (i.e., the functions determining the input to each child term). To remedy this issue, our implementation of SynthSemanticConstraint uses an additional CEGIS loop that resynthesizes the constraint for each output until all agree on the inputs to each child term.

We detail SynthSemanticConstraint for $N$ outputs in Algorithm 3. For simplicity, we explain how Algorithm 3 works for a production that has two outputs (i.e., $N = 2$). Consider the case for $A_0 \rightarrow p(A_1, \ldots, A_n)$ where $\tau_{A_0} \triangleq \tau_1 \times \tau_2$. In this scenario, our goal is to synthesize two constraints $F$ and $G$ (i.e., $F = F_1$ and $G = F_2$),

$$F \triangleq x_1 = f_1(x_0) \wedge \cdots \wedge x_n = f_n(x_0, x_1', \ldots, x_{n-1}') \wedge x_{0_0}' = f_0(x_0, x_1', \ldots, x_n') \tag{4}$$

$$G \triangleq x_1 = g_1(x_0) \wedge \cdots \wedge x_n = g_n(x_0, x_1', \ldots, x_{n-1}') \wedge x_{0_1}' = g_0(x_0, x_1', \ldots, x_n') \tag{5}$$

To determine if $F$ and $G$ agree on each child term's input for example set $E'$, we generate the formula $\phi$ shown below, for each example $\langle in, p(t_1, \ldots, t_n), out\rangle \in E'$:

$$\begin{aligned} x_0^F = x_0^G = in \wedge Summary(t_1)(x_1^F, x_1^{'F}) \wedge \cdots \wedge Summary(t_n)(x_n^F, x_n^{'F}) \\ \wedge Summary(t_1)(x_1^G, x_1^{'G}) \wedge \cdots \wedge Summary(t_n)(x_n^G, x_n^{'G}) \\ \wedge F \wedge G \wedge (x_1^F \neq x_1^G \vee \cdots \vee x_n^F \neq x_n^G) \wedge \left\langle x_{0_0}^{'F}, x_{0_1}^{'G} \right\rangle = out \end{aligned} \tag{6}$$

which asks if $F$ and $G$ agree on the input to each child term for the given example. To make this concept concrete, consider the following example.

*Example 5.1 (Synthesizing Semantic Constraint for Multi-Output Production.).* Consider the task of synthesizing a semantics for $\texttt{x}_0 \coloneqq$ in the language Imp$_2$, using the examples: $\langle\langle 0, 1\rangle, \texttt{x}_0 \coloneqq \texttt{x}_1, \langle 1, 1\rangle\rangle$, $\langle\langle 0, 1\rangle, \texttt{x}_1, 1\rangle, \langle\langle 1, 1\rangle, \texttt{x}_1, 1\rangle$.

For the above examples, SynthSemanticConstraint might generate $F \triangleq x_{1,0}^{in} = x_{0,0}^{in} \wedge x_{1,1}^{in} = x_{0,1}^{in} \wedge x_{0,0}^{out} = x_{1,1}^{out}$ and $G \triangleq x_{1,0}^{in} = x_{0,1}^{in} \wedge x_{1,1}^{in} = x_{0,1}^{in} \wedge x_{0,0}^{out} = x_{1,1}^{out}$, where $x_{i,j}^{in}$ is the $j^{th}$ projection of $x_i^{in}$. While both $F$ and $G$ are consistent with the examples, the data-flow of $F$ is not consistent

**Algorithm 3:** SynthSemanticConstraint for multi-output productions.

**Procedure** *SynthSemanticConstraint*$(p, E)$
  $A_0 \to \sigma(A_1, \ldots, A_n) \leftarrow p$;
  $\tau_1 \times \cdots \times \tau_N \leftarrow \tau_{A_0}$ ; // Determine number of outputs for production $p$.
  $D \leftarrow true$ ; // Data flow constraints.
  **while** *true* **do**
    $\Gamma \leftarrow D$;
    **for** $i \leftarrow 1$ *to* $N$ **do**
      // Construct per-output conditions (c.f., Equation (4))
      $F_i \leftarrow x_1 = f_1^{F_i}(x_0) \wedge \cdots \wedge x_n = f_n^{F_i}(x_0, x_1', \ldots, x_{n-1}') \wedge x_0{}_i' = f_0^{F_i}(x_0, x_1', \ldots, x_n')$;
      $\Gamma \leftarrow \Gamma \wedge F_i$;
    // Generate SyGuS conditions (c.f., lines 1-2 of Equation (6))
    $\phi \leftarrow \left(\bigwedge_{i,j} Summary(t_i)(x_i^{F_j}, x_i^{F_j\prime})\right) \wedge \langle x_0{}_0^{F_0\prime}, \ldots, x_0{}_n^{F_n\prime}\rangle = out$;
    $\Gamma \leftarrow \phi \wedge \left(\bigwedge_i x_0^{F_i} = in\right)$;
    $m \leftarrow$ SolveSygus$(\Gamma)$;
    $M \leftarrow$ CheckSat$(\phi)$;
    // Check if inconsistency is found (line 3 of Equation (6))
    **if** $M.sat \wedge \exists i, j, k : M(x_i^{F_j}) \neq M(x_i^{F_k})$ **then**
      // Inconsistency is caused by inaccurate summary of a child term
      **if** $\exists t_i : \forall \langle in, t_i, out\rangle \in E : \forall j : in \neq M(x_i^{F_j})$ **then** $E \leftarrow E \cup \{\langle in, t, \mathcal{I}(t_i, in)\rangle\}$ ;
      // Real data flow inconsistency
      **else** $D \leftarrow D \wedge \left(\bigvee_{i,j} x_i^{F_j} \neq M(x_i^{F_j})\right)$;
    **else**
      // No inconsistency
      merge $f_i^{F_j} \in m$ to form the solution;
      **return** *solution*;

with the data-flow of $G$ (i.e., in $F$, $x_{1,0}^{in}$ is assigned $x_{0,0}^{in}$, while in $G$, $x_{1,0}^{in}$ is assigned $x_{0,1}^{in}$). We can construct the formula in Equation (6) for $F$ and $G$, and find out that in $F$, the variable $x_{1,0}^{in\ F}$ takes the value 0, and in $G$, the variable $x_{1,0}^{in\ G}$ takes value 1. Thus, $F$ and $G$ are not consistent on data-flows to children for the provided example. We generate a new condition for the next iteration of SynthSemanticConstraint that asserts $x_{0,0}^{in\ F} \neq 0 \vee x_{0,0}^{in\ G} \neq 1$.

In practice, we create a copy of each variable indexed by $F$ and $G$, respectively, to avoid clashing variable names when encoding the constraints $F$ and $G$ within a single formula. To check the consistency of $F$ and $G$'s data flows, we use cvc5 to check the satisfiability of the formula $\phi$ in Equation (6). If $\phi$ is unsatisfiable, then $F$ and $G$ must agree on the inputs of all child terms for the given examples. If so, then we may return either $F \wedge x_{02} = g_0(\ldots)$ or $G \wedge x_{01} = f_0(\ldots)$ (i.e., because $F$ and $G$ agree on all child term inputs, we may use either to constrain the data-flow to child terms).

If $\phi$ is satisfiable, then $F$ and $G$ do not agree on the input to all child terms. In this case, we find a model that satisfies $\phi$. If there is some subterm $t_i$ such that there is no example $\langle in, t_i, out\rangle \in E$ such that $in = M(x_i^F)$ or $in = M(x_i^G)$, then we add the example $\langle in, t, \mathcal{I}(t_i, in)\rangle$ to the set of examples, and resynthesize the constraints $F$ and $G$. Otherwise, we know that the sub-term summaries are

sufficient to fully specify both $F$ and $G$ for all examples in $E$. Thus, we must add a new constraint that ensures the pair of constraints $F$ and $G$ are never synthesized again. To do this, we add a new constraint $x_0^F \neq M(x_0^F) \vee x_0^G \neq M(x_0^G) \vee \cdots \vee x_n^F \neq M(x_n^F) \vee x_n^G \neq M(x_n^G)$, which ensures that the input of at least one of the child terms for either $F$ or $G$ must change. A new candidate $F$ and $G$ are then synthesized. The CEGIS loop continues until it finds a valid pair of $F$ and $G$ for the set of examples.

## 6 Evaluation

The goal of our evaluation is to answer the following questions:

**RQ1** Can Synantic synthesize the semantics of non-trivial languages?
**RQ2** Where is time spent during synthesis?
**RQ3** Is the multi-output optimization from Section 5.3 effective?
**RQ4** How do synthesized semantics compare to manually written ones?

All experiments were run on a machine with an Intel(R) i9-13900K CPU and 32 GB of memory, running NixOS 23.10 and Scala 2.13.13. All experiments were allotted 2 hours, 4 cores of CPU, and 24 GB of memory. Cvc5 version 1.0.3 is used for SMT solving and SyGuS function synthesis. For the total running time of each experiment, we report the median of 7 runs using different random seeds. For every language, we record whether Synantic terminates within the given time limit of 2 hours, and when it does, we also record the set of synthesized semantic rules. A language that does not terminate within the time limit on more than half of the seeds is reported as a timeout.

### 6.1 Benchmarks

We collected 15 benchmarks from the two sources discussed below. For every language discussed in this section, we manually translated the semantics to a simple equivalent interpreter written in Scala; our goal was then to synthesize an appropriate CHC-based semantics from the interpreter. The one non-standard feature of our setup is that the interpreter must be capable of interpreting the programs derived from *any* nonterminal in the grammar.

*SemGuS benchmarks.* Our first source of benchmarks is the SemGuS benchmark repository [13]. This dataset contains SemGuS synthesis problems where each problem consists of a grammar of terms, a set of CHCs inductively defining the semantics of terms in the grammar, and a specification that the synthesized program should meet. For our purposes, we ignored the specification and collected the grammar plus semantics for 11 distinct languages that appear in the repository. We do not consider languages that contain abstract data types (e.g., stacks) or require a large range of inputs (e.g., ASCII characters) due to their poor support by the SyGuS solver. These languages gave us 11 benchmarks.

Some of the languages used in the SemGuS benchmark set are parametric (denoted by a parameter $k$), meaning that the semantics is slightly different based on a given parameter (e.g., number of program variables for IMP and length of the input string for regular expressions). For these benchmarks, we ran Synantic on an increasing sequence of parameter values and reported the largest parameter value for which Synantic succeeds.

RegEx($k$) is a language for matching regular expressions on strings of length $k$; Given a regular expression $r$ and string $s$ of length $k$ (index starts from 0), the semantic functions produce a Boolean matrix $M \in \texttt{Bool}^{(k+1)\times(k+1)}$ such that $M_{i,j} = \mathit{true}$ iff the substring $s_{i\ldots j-1}$ matches regular expression $r$—here $s_{i\ldots i}$ denotes the empty string, and by definition, $M_{i,j} = \mathit{false}$ for $i \geq j$.

Cnf($k$), Dnf($k$) and Cube($k$) are languages of Boolean formulas (of the syntactic kind indicated by their names, i.e., conjunctive normal form, disjunctive normal form, and cubes) involving up to $k$ variables.

Imp is an imperative language that contains common control flow structures, such as conditionals and while loops, for programs with $k$ integer variables. Note that Imp includes operators such as `while` and `do_while` for which the semantics involves semantically recursive productions (Section 4.4). The complete semantics of Imp can be found in the supplementary material. Two versions of Imp are used in our benchmarks. The first version is called Imp($k$), where we explicitly record the states of $k$ variables as $k$ arguments of semantic functions. Synantic could synthesize its semantics up to $k = 2$. We also present another version of this language called ImpArr where an arbitrary number of variables can be used. In ImpArr, variables are named $var_0, var_1, \ldots$ where the subscript is any natural number. We use the theory of arrays to store the variable states into an array, passing the array as an argument to the semantic function. The array is indexed by variable id. When we present results later in the section, the results for both languages (i.e., Imp(2) and ImpArr) are shown for comparison. (For Imp(2), the goal is to synthesize a semantics that works on states with exactly 2 variables; for ImpArr, the goal is to synthesize a semantics that works for states with any number of variables.)

IntArith is a benchmark about basic integer calculations, like addition, multiplication, and conditional selection. It also includes three constants whose value can be specified in the input to the semantic relations.

BvSimple($k$) describes bit-vector operations involving $k$ bit-vector constants. BvSimpleImp($m, n$) is essentially a variant of BvSimple($k$) that augments the language with let-expressions. Parameters $m$ and $n$ mean that the language can use up to $m$ bit-vector constants and $n$ bit-vector variables. BvSaturate($k$) and BvSaturateImp($k$) use the same syntaxes as BvSimple($k$) and BvSimpleImp($k$), respectively, but operations use a saturating semantics that never overflows or underflows.

*Attribute-grammar synthesis [12].* Our second source of benchmarks is from the Panini tool for synthesizing attribute grammars [12]. An attribute grammar (AG) associates each nonterminal of an underlying context-free grammar with some number of *attributes*. Each production has a set of attribute-definition rules (sometimes called *semantic actions*) that specify how the value of one attribute of the production is set as a function of the values of other attributes of the production. In a given derivation tree of the AG, each node has an associated set of *attribute instances*. The attribute-definition rules are used to obtain a consistent assignment of values to the tree's attribute instances: each attribute instance has a value equal to its defining function applied to the appropriate (neighboring) attribute instances of the tree. Effectively, AGs assign a semantics to programs via attributes, and the underlying attribute-definition rules can be captured via CHCs. While there are AG extensions to handle circular AGs [11, 18]—i.e., AGs in which some derivation trees have attribute instances that are defined in terms of themselves—the work of Kalita et al. concerns non-circular AGs.

Kalita et al. [12] present 12 benchmarks. We ignored 4 benchmarks that are either (*i*) not publicly accessible, or (*ii*) use semantic functions that cannot be expressed in SMT-LIB and are thus beyond what can be synthesized using a SyGuS solver—e.g., complex data structures, or (*iii*) identical to existing benchmarks from other sources. We did not run their tool on our benchmarks because our problem is more general than theirs, supporting a wider range of language semantics: the scope of our work includes recursive semantics, which can be handled only indirectly in a system such as theirs (which supports only non-circular AGs)—i.e., by introducing powerful hard-to-synthesize recursive functions that effectively capture an entire construct's semantics. The running time is also not directly comparable, because Kalita et al.'s approach uses user-provided sketches (i.e., partial solutions to each semantic action), which simplifies the synthesis problem. In contrast, in

our work we do not assume that a sketch is provided for the semantic constraints and instead consider general SyGuS grammars.

The remaining 8 benchmarks of Kalita et al. are consolidated as 4 languages (i.e., giving us four benchmarks). IteExpr is a language of basic integer operations, comparison expressions, and ternary if-then-else expressions (not statements). Our IteExpr benchmark subsumes benchmarks B3, B4, and B5 of Kalita et al. because their only differences stem from whether the expression is written in prefix, postfix, or infix notation. For Synantic, such surface-syntax differences are unimportant because Synantic uses regular tree grammars to express a language's abstract syntax, and the underlying abstract syntax of prefix, postfix, and infix expressions is the same. BinOp is a language of binary strings (combined from benchmarks B1 and B2 of Kalita et al.), along with built-in functions for popcount (counting the number of ones) and binary-to-decimal conversion. Currency is a language for currency exchange and calculation. Diff is a language for computing finite differences. Because the original benchmark from Kalita et al. involves differentiation and real numbers (which are not supported by existing SyGuS solvers), we modified the benchmark to perform the related operation of finite differencing over integer-valued functions. Specifically, for a function $f$, its finite difference is defined as $\Delta f = f(x{+}1) - f(x)$. Starting from here, finite differences for sums and products can be obtained compositionally, e.g., $\Delta\,(u \cdot v) = u(x)\Delta v(x) + v(x+1)\Delta u(x)$.

*SyGuS grammars.* For each semantic function, we also provided a grammar for the SyGuS solver, which contains the operators of the underlying logical theory and any specific functions that must appear in the target semantics.

For instance, for all benchmarks using the logic fragment `NIA`, we allow the use of basic integer operations and integer constants, along with language-specific operations like conditional operators (if-then-else).

For the languages Diff and Currency we did not include conditional operators, because they do not appear in the semantics.

For BVSaturated and BVIMPSaturated we provided operators for detecting overflow and underflow.

Lastly, for languages known to be free of side effects, we modified the SyGuS grammars to forbid data flow between siblings, and only allow parent-to-child and child-to-parent assignments.

### 6.2 RQ1: Can Synantic Synthesize the Semantics of Non-trivial Languages?

Table 1 presents a highlight of the results of running Synantic on each benchmark (column 1) for each production rule (column 2). For the parametric languages, we ran each benchmark up to the largest parameter $k$ for which the solver timed out and reported the running time and other metrics for the largest such $k$ (more details below). The third column provides the median number of CEGIS iterations taken to synthesize each production, and the fourth column provides the median number of $\langle in, term, out\rangle$ counterexamples found for one production rule. We take the median of total execution time on one production rule and list it in column 7. Columns 5–6 are breakdowns of the total time into time for SyGuS solving and time for SMT solving. To summarize, Synantic could synthesize complete semantics for $12/15 \approx 80\%$ of benchmark languages (two languages exist for the Imp benchmark, see below).

For RegEx($k$) ($k = 2, \ldots, 8$) , Synantic could synthesize a semantics for up to $k = 2$. For Cnf($k$) ($k = 4, \ldots, 8$), Dnf($k$) ($k = 4, \ldots, 8$), and Cube($k$) ($k = 4, \ldots, 11$), Synantic could synthesize semantics for all parameters included in the SemGuS benchmarks. For the bit vector benchmarks, Synantic could synthesize a semantics for BVSimple($k$) up to $k = 3$, and a semantics for BVIMPSimple($m, n$) ($(m, n) \in \{(1, 2), (3, 3)\}$) up to $m = 1$ and $n = 2$.

Table 1. Detailed results for selected benchmarks. See supplementary material for the full list of results.

| Lang. | Rule | # Iter. | # Ex | SyGuS (s) | SMT (s) | Total (s) |
|---|---|---|---|---|---|---|
| ImpArr | $E \to \texttt{0}$ | 1 | 1 | 0.01 | 0.01 | 0.04 |
| | $E \to \texttt{1}$ | 1 | 1 | 0.01 | 0.01 | 0.04 |
| | $B \to \texttt{f}$ | 1 | 1 | 0.01 | 0.01 | 0.05 |
| | $B \to \texttt{t}$ | 1 | 1 | 0.01 | 0.01 | 0.10 |
| | $S \to \texttt{dec\_var}_i$ | 3 | 2 | 4.29 | 0.56 | 5.36 |
| | $S \to \texttt{inc\_var}_i$ | 3 | 2 | 3.56 | 0.54 | 4.51 |
| | $B \to \neg B$ | 3 | 2 | 0.01 | 1.42 | 5.53 |
| | $E \to \texttt{var}_i$ | 3 | 2 | 0.01 | 0.28 | 0.66 |
| | $E \to E + E$ | 3 | 2 | 0.02 | 6.51 | 12.38 |
| | $E \to E - E$ | 3 | 2 | 0.01 | 6.43 | 12.13 |
| | $B \to E < E$ | 4 | 3 | 0.01 | 3.38 | 10.33 |
| | $B \to B \wedge B$ | 4 | 3 | 0.06 | 2.36 | 6.23 |
| | $S \to \texttt{var}_i \coloneqq E$ | 2 | 1 | 0.03 | 8.11 | 11.60 |
| | $B \to B \vee B$ | 5 | 4 | 0.03 | 2.42 | 6.14 |
| | $S \to S\ ;\ S$ | 3 | 1 | 0.02 | 13.88 | 25.91 |
| | $S \to \texttt{do\_while}\ S\ B$ | 5 | 2 | 0.22 | 342.25 | 499.11 |
| | $S \to \texttt{while}\ B\ S$ | 4 | 2 | 0.10 | 218.66 | 321.11 |
| | $S \to \texttt{ite}\ B\ S\ S$ | 4 | 2 | 0.03 | 7.08 | 27.82 |
| IMP(2) | $E \to \texttt{0}$ | 1 | 1 | 0.01 | 0.01 | 0.05 |
| | $E \to \texttt{1}$ | 1 | 1 | 0.01 | 0.01 | 0.04 |
| | $S \to \texttt{x}--$ | 2 | 2 | 0.06 | 0.02 | 0.11 |
| | $S \to \texttt{y}--$ | 2 | 2 | 0.11 | 0.03 | 0.17 |
| | $B \to \texttt{f}$ | 1 | 1 | 0.01 | 0.01 | 0.06 |
| | $S \to \texttt{x}++$ | 2 | 2 | 0.04 | 0.03 | 0.11 |
| | $S \to \texttt{y}++$ | 2 | 2 | 0.12 | 0.02 | 0.16 |
| | $B \to \texttt{t}$ | 1 | 1 | 0.01 | 0.02 | 0.13 |
| | $E \to \texttt{x}$ | 2 | 2 | 0.01 | 0.01 | 0.04 |
| | $E \to \texttt{y}$ | 1 | 1 | 0.01 | 0.01 | 0.04 |
| | $S \to \texttt{x} \coloneqq E$ | 2 | 2 | 0.10 | 3.23 | 6.17 |
| | $S \to \texttt{y} \coloneqq E$ | 2 | 2 | 0.04 | 3.22 | 6.19 |
| | $B \to \neg B$ | 3 | 3 | 0.02 | 2.49 | 5.26 |
| | $E \to E + E$ | 4 | 3 | 0.05 | 8.52 | 14.83 |
| | $E \to E - E$ | 5 | 2 | 0.13 | 8.03 | 13.83 |
| | $B \to E < E$ | 8 | 5 | 0.08 | 7.50 | 13.66 |
| | $B \to B \wedge B$ | 4 | 4 | 0.03 | 5.33 | 11.71 |
| | $B \to B \vee B$ | 4 | 4 | 0.05 | 4.61 | 8.99 |
| | $S \to S\ ;\ S$ | 5 | 3 | 4.55 | 15.00 | 72.53 |
| | $S \to \texttt{do\_while}\ S\ B$ | 27 | 35 | 858.50 | 257.33 | 1374.13 |
| | $S \to \texttt{while}\ B\ S$ | 9 | 7 | 16.88 | 122.41 | 266.80 |
| | $S \to \texttt{ite}\ B\ S\ S$ | 11 | 5 | 525.28 | 33.88 | 628.71 |
| BinOp | $B \to \texttt{0}$ | 1 | 1 | 0.01 | 0.01 | 0.07 |
| | $B \to \texttt{1}$ | 1 | 1 | 0.01 | 0.01 | 0.22 |
| | $B \to \texttt{x}$ | 2 | 2 | 0.01 | 0.01 | 0.08 |
| | $N \to \texttt{atom}\ B$ | 2 | 2 | 0.09 | 0.04 | 0.30 |
| | $M \to \texttt{atom}'\ B$ | 3 | 3 | 0.07 | 0.05 | 0.26 |
| | $S \to \texttt{bin2dec}\ M$ | 2 | 2 | 0.02 | 0.09 | 0.30 |
| | $S \to \texttt{count}\ N$ | 2 | 2 | 0.04 | 0.05 | 0.24 |
| | $N \to \texttt{concat}\ N\ B$ | 5 | 5 | 8.61 | 0.22 | 10.31 |
| | $M \to \texttt{concat}'\ M\ B$ | 5 | 5 | 288.81 | 0.23 | 308.50 |
| RegEx(2) | $Start \to \texttt{eval}\ R$ | 3 | 3 | 0.02 | 4.43 | 13.40 |
| | $R \to \texttt{?}$ | 3 | 3 | 3.84 | 0.07 | 4.07 |
| | $R \to \texttt{a}$ | 4 | 4 | 11.10 | 0.07 | 11.53 |
| | $R \to \texttt{b}$ | 5 | 5 | 11.63 | 0.06 | 12.01 |
| | $R \to \epsilon$ | 1 | 1 | 0.07 | 0.07 | 2.38 |
| | $R \to \emptyset$ | 1 | 1 | 0.19 | 0.07 | 0.46 |
| | $R \to !R$ | 5 | 5 | 2.85 | 15.77 | 77.36 |
| | $R \to R^*$ | 6 | 6 | 0.99 | 13.06 | 31.91 |
| | $R \to R \cdot R$ | 24 | 24 | 333.71 | 72.58 | 495.45 |
| | $R \to R \mid R$ | 10 | 10 | 10.96 | 59.54 | 140.82 |

For all these parametric cases that timeout, the number of input and output variables in semantic functions is large: 10 inputs and 10 outputs for RegEx(3).

Additionally, Synantic timed out for the benchmarks Diff, BVSaturated, and BVIMPSaturated.[2] For Diff, 4 of the 7 runs resulted in a timeout, so Diff is reported as a timeout (even though at least one run could synthesize the semantics of all the productions). For the 4 runs that

[2]Data for some languages are only listed in the supplementary material.

timed out, Synantic can solve the semantics of 5 of the 6 productions in the grammar. Synantic could synthesize the semantics of 9/18 productions for BVIMPSaturated, and 10/17 productions for BVSaturated in at least one run.

In benchmarks that timed out, the time-out happened during a call to the SyGuS solver—i.e., the functions to be synthesized were too complex (more details in Section 6.3).

*Finding:* To answer RQ1, Synantic can synthesize semantics for many non-trivial languages as long as the semantics does not involve very large functions (more than 20 terms).

### 6.3 RQ2: Where is Time Spent during Synthesis?

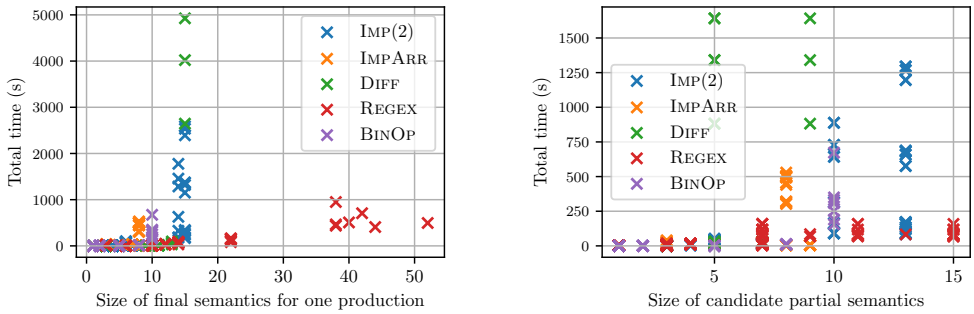
*SyGuS vs SMT Time.* Appendix B also presents the breakdown of how much time the solver spends solving SyGuS problems (to find candidate functions) and calling SMT solvers (to compute complete summaries). Among all the benchmarks, a median of 16.24% of the total solving time is spent on SyGuS problems, and a median of 19.90% of the time is spent solving SMT queries. However, for the slowest 10% production rules (>32.17 s), the median of SyGuS solving time grows to 64.91%, which indicates that SyGuS contributes to most of the execution time on slow-running cases.

Among all benchmarks, 90% of the per-production semantics are solved within 32.17 s. The 12 rules that take longer than 32.17 s to be synthesized are all non-leaf rules and their partial semantic constraints fall into the following three categories: (*i*) 5 of them contain large integers or complex SMT primitives (e.g., 32-bit integer division, theory of arrays); (*ii*) 3 of them involve large logical formulas with sizes ranging between 8 and 24 subterms, e.g., formulas representing $3 \times 3$ matrix multiplication or other matrix operations; (*iii*) 4 of them contain multiple input and output parameters of semantic functions that correspond to variable states, e.g., `while` and `do_while`. In particular, Synantic takes 1374.13 s to synthesize the CHC for `do_while` in Imp(2) because there can be many possible ways to modify the data flow between the production's child terms; this aspect occurs in many CEGIS iterations. In all of the above cases, as expected from known limitations of CVC5, the SyGuS and SMT solvers account for most of the execution time—45.63% and 27.98% of the total running time is spent calling the SyGuS and SMT solvers, respectively.

*Relation to CEGIS Iterations and Size of Solutions.* Table 1 hints that the cost of synthesizing a semantics may be proportional to the number of CEGIS iterations, which in general is a good indicator of the complexity of a formula (and of how expressive the underlying SyGuS grammar is). Additionally, the cost should also be proportional to the size of synthesized parts in the SyGuS problems, which directly indicates formula complexity. We plotted Figure 1 to better understand those relations by using the data from some slowest benchmarks.

Figure 1a shows the relationship between the time for synthesizing a per-production rule semantics and the size of the final semantics. For the same language, the time grows exponentially with the increase in the size of the final solution. Figure 1b shows that the time also grows exponentially with the increase in solution size for per-output partial semantic constraint.

Because the performance varies greatly across different benchmarks, to better understand the impact of CEGIS iterations, we focus our attention on one difficult benchmark, Imp(2). Specifically, we analyze the time taken to synthesize the semantic rule for `do_while`, which was one of the hardest productions in our benchmark set (2,500s). Figure 4 provides a stack plot detailing the running time for all 16 CEGIS iterations needed to synthesize `do_while`. As expected, as more examples are accumulated by CEGIS iterations, the SyGuS solver requires more time. The execution time for different parts is plotted by the areas of different colors. We can conclude that for the rule of `do_while`, SyGuS solver takes 64.3% of the execution time.




(a) Time vs. semantic constraint size

(b) Time vs. partial semantic constraint size

Fig. 1. Plots relating the time to synthesize the semantics of one production rule vs final semantic constraint solution size (a) and partial semantic constraint solution size (b). We only included selected slowest benchmarks due to graph size limit.

$$[\![\cdot]\!]^{\textsc{Imp}(2)}_{Sem.S} : \mathcal{L}(S) \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$$
$$[\![\cdot]\!]^{\textsc{ImpArr}}_{Sem.S} : \mathcal{L}(S) \times \mathcal{A}^{\mathbb{Z}}_{\mathbb{Z}} \times \mathcal{A}^{\mathbb{Z}}_{\mathbb{Z}}$$

Fig. 2. Selected differences of semantic signatures between Imp(2) and ImpArr. $\mathcal{A}^{\mathbb{Z}}_{\mathbb{Z}}$ stands for an SMT array mapping integers to integers.

$$\frac{[\![b]\!](\sigma) = v \qquad [\![s_1]\!](\sigma) = \sigma' \qquad [\![s_2]\!](\sigma) = \sigma''}{[\![\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2]\!](\sigma) = v \ ? \ \sigma' : \sigma''} \textsc{Ite}$$

$$\frac{[\![b]\!](\sigma) = \mathit{true} \qquad [\![s]\!](\sigma) = \sigma' \qquad [\![\texttt{while } b \texttt{ do } s]\!](\sigma') = \sigma''}{[\![\texttt{while } b \texttt{ do } s]\!](\sigma) = \sigma''} \textsc{WhileLoop}$$

$$\frac{[\![b]\!](\sigma) = \mathit{false}}{[\![\texttt{while } b \texttt{ do } s]\!](\sigma) = (\sigma)} \textsc{WhileEnd}$$

Fig. 3. Selected semantic rules for ImpArr. $\sigma, \sigma', \sigma''$ are arrays.

*Simplifying synthesis with appropriate SMT theory.* To our surprise, the language ImpArr, which uses the theory of array to model an arbitrary number of variables, takes 1041.2s on average, being nearly twice as fast as Imp(2). To understand why this is the case, note the difference in their semantic signatures (Figure 2): the signature of Imp(2)'s semantics contains 3 input arguments and 2 output arguments. However, the signature of ImpArr's semantics contains only 2 input arguments and 1 output argument, packing program states into a single array rather than $k$ arguments (see Figure 3 for some examples). By choosing an appropriate theory, the signature of the semantics can be simplified, thus shrinking the solution space for synthesis.

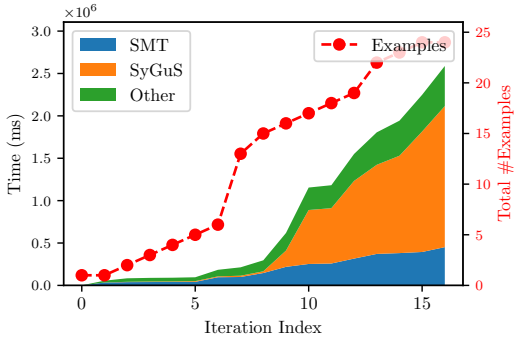



Fig. 4. Execution time per iteration for `do_while` in Imp(2)

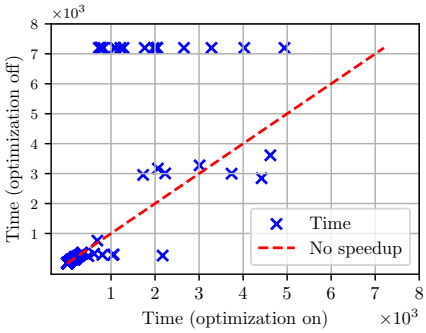



Fig. 5. Speedup provided by optimization

*Finding:* To answer RQ2, Synantic spends most of the time (71.78%) solving *SyGuS problems*, and the time is affected by the size of the candidate semantic function.

### 6.4 RQ3: Is the Multi-output Optimization from Section 5.3 Effective?

Figure 5 compares the running time of Synantic with and without the multi-output optimization (Section 5.3) on all the runs of our tools for the 7 different random seeds.

With the optimization turned off, Synantic timed out on 10 more runs (specifically all the 7 runs for RegEx and 3 more runs for Diff). All the benchmarks for which disabling the optimization caused a timeout have 3 or more output variables. Comparing Figure 1a and Figure 1b shows how the semantic functions used in the RegEx benchmarks are very large (up to size 50), but thanks to the optimization, our algorithm only has to solve SyGuS problems on formulas of size at most 15.

On the runs that terminated both with and without the optimization, the non-optimized algorithm is on average 8% faster—i.e., the two versions of the algorithm have comparable performance. However, for 15/98 runs the optimization results in a 20% or more slowdown. When inspecting these instances, we observed that the multi-output optimization spent many iterations synchronizing the many possible data flows for productions where the final term was actually small but many variables were involved—e.g., sequential composition in Imp(2).

*Finding:* The multi-output optimization from Section 5.3 is effective for languages with 3 or more output variables in their semantics.

### 6.5 RQ4: How do Synthesized Semantics Compare to Manually Written Ones?

The synthesized semantics for almost all of our benchmarks are either identical to the original manually constructed one, or each CHC in the synthesized semantics is logically equivalent to the CHC of the original semantics.

The one exception is the semantics synthesized for the language of RegEx(2), for which the individual CHCs for Or, Concat, Neg, and Star are not logically equivalent to the manually-written ones. For instance, consider the Concat rule for the semantics of concatenation. For this construct, the manually written CHC is shown in Figure 6a, whereas Synantic synthesizes the CHC shown in Figure 6b. The two CHCs are not logically equivalent. For example, if the children evaluate to the matrices $M = \begin{pmatrix} true & false & false \\ false & false & \\ & & true \end{pmatrix}$ and $M' = \begin{pmatrix} true & false & false \\ false & false & \\ & & true \end{pmatrix}$, the outputs computed by the manually

$$\frac{[\![e_1]\!](s) = \begin{pmatrix} M_{0,0} & M_{0,1} \\ & M_{1,1} \end{pmatrix} \qquad [\![e_2]\!](s) = \begin{pmatrix} M'_{0,0} & M'_{0,1} \\ & M'_{1,1} \end{pmatrix}}{[\![e_1 \cdot e_2]\!](s) = \begin{pmatrix} (M_{0,0} \wedge M'_{0,0}) & (M_{0,0} \wedge M'_{0,1}) \vee (M_{0,1} \wedge M'_{1,1}) & (M_{0,0} \wedge M'_{0,2}) \vee (M_{0,1} \wedge M'_{1,2}) \vee (M_{0,2} \wedge M'_{2,2}) \\ & (M_{1,1} \wedge M'_{1,1}) & (M_{1,1} \wedge M'_{1,2}) \vee (M_{1,2} \wedge M'_{2,2}) \\ & & (M_{2,2} \wedge M'_{2,2}) \end{pmatrix}} \textsc{ConcatM}$$

(a) Manually written semantics

$$\frac{[\![e_1]\!](s) = \begin{pmatrix} M_{0,0} & M_{0,1} \\ & M_{1,1} \end{pmatrix} \qquad [\![e_2]\!](s) = \begin{pmatrix} M'_{0,0} & M'_{0,1} \\ & M'_{1,1} \end{pmatrix}}{[\![e_1 \cdot e_2]\!](s) = \begin{pmatrix} M_{2,2} \wedge M'_{1,1} & (M_{0,0} \wedge M'_{0,1}) \vee (M'_{0,0} \wedge M_{0,1}) & (M_{2,2} \wedge M'_{0,2}) \vee (M_{0,2} \wedge M'_{0,0}) \vee (M_{0,1} \wedge M'_{1,2}) \\ & (M_{0,0} \vee M_{2,2}) \wedge M'_{2,2} & (M'_{1,1} \wedge M_{1,2}) \vee (M_{1,1} \wedge M'_{1,2}) \\ & & (M_{1,1} \wedge M'_{0,0}) \end{pmatrix}} \textsc{ConcatS}$$

(b) Synthesized Semantics

Fig. 6. Manually-written and synthesized semantics for Concat in RegEx(2)

$$\frac{[\![e_1]\!](s) = \left(M_\epsilon, \begin{pmatrix} M_{0,0} & M_{0,1} \\ & M_{1,1} \end{pmatrix}\right) \qquad [\![e_2]\!](s) = \left(M'_\epsilon, \begin{pmatrix} M'_{0,0} & M'_{0,1} \\ & M'_{1,1} \end{pmatrix}\right)}{[\![e_1 \cdot e_2]\!](s) = \left(M_\epsilon \wedge M'_\epsilon, \begin{pmatrix} (M_\epsilon \wedge M'_{0,0}) \vee (M_{0,0} \wedge M'_\epsilon) & (M_\epsilon \wedge M'_{0,1}) \vee (M_{0,0} \wedge M'_{1,1}) \vee (M_{0,1} \wedge M'_\epsilon) \\ & (M_\epsilon \wedge M'_{1,1}) \vee (M_{1,1} \wedge M'_\epsilon) \end{pmatrix}\right)} \textsc{Concat}$$

Fig. 7. New semantics for Concat in RegExSimp.

written CHC and the synthesized CHC are $M_{man} = \begin{pmatrix} true & false & false \\ & false & false \\ & & true \end{pmatrix}$, and $M_{syn} = \begin{pmatrix} false & false & false \\ & true & false \\ & & false \end{pmatrix}$, respectively, which have different values on the diagonal.

When inspecting the two rules, we realized that the example matrices $M$ and $M'$ shown above cannot actually be produced by the semantic rules for regular expressions. In particular, the examples require different Boolean values to appear on the diagonal of one $3 \times 3$ matrix. However, all the elements on the diagonal represent the semantics of the regular expression on the empty string, so they must all have the same value! We note that this inconsistency in the semantics can also be observed without a reference semantics to compare against because different runs of the algorithm could return logically inequivalent CHCs—in fact, such inequivalence was how we initially discovered the inconsistency.

Synantic helped us discover an inefficiency in the semantics that was being used in the standard regular expressions benchmarks in the SemGuS repository. We thus modified the interpreter so that for the example above it only produces a $2 \times 2$ matrix $M = \begin{pmatrix} M_{0,1} & M_{0,2} \\ & M_{1,2} \end{pmatrix}$ (corresponding to the non-empty substrings of the input string) and a *single* variable $M_\epsilon$ to denote whether the regular expression should accept the empty string (instead of the previous multiple copies of logically equivalent variables). This semantics reduces the total number of variables in the semantic domain from 6 to 4 in this example.

We call this new semantics RegExSimp (see Figure 7 for an example). After modifying the interpreter to produce this new semantics, Synantic synthesized the corresponding CHCs in a median of 1968.00 s.

To check whether the semantics RegExSimp is indeed more efficient than the original semantics RegEx, we modified all the 28 regular-expression synthesis benchmarks appearing in the SemGuS

benchmark set. Each of these benchmarks requires one to find a regular expression that accepts some examples and rejects others.

We then used the Ks2 enumeration-based synthesizer to try to solve all the benchmarks with either of the two semantics. Because Ks2 enumerates programs of increasing size and uses the semantics to execute them and discard invalid program candidates, we conjectured that executing programs faster allows Ks2 to explore the search space faster.

Ks2 was faster at solving synthesis problems with the RegExSimp semantics than with the RegEx ones (although both solved the same set of benchmarks). Although the speedup over all benchmarks is only 1.1x, the new semantics RegExSimp *was particularly beneficial* for the harder synthesis problems. When considering the 13 benchmarks for which synthesis using the RegEx took longer than one second, the speedup increased to 1.18x.

*Finding:* Synantic synthesized semantics that were identical to the manually written ones for 13/14 benchmarks. When Synantic found a logically inequivalent semantics, it unveiled a performance bug.

## 7 Related Work

*Semantics-based Synthesis vs. Library-based Synthesis.*

As discussed throughout the paper, our framework is intended for extracting a formal SemGuS semantics that can be used to then take advantage of existing SemGuS synthesizers. One can compare our two-step approach (i.e., first synthesizing the semantics in SemGuS format, then using it to synthesize programs) to one-step approaches that only use the given program interpreter in a closed-box fashion to evaluate input-output examples and use them to perform example-based synthesis. (Such an approach is also used when synthesizing programs that contain calls to closed-box library functions [9].) On one hand, the closed-box example-based approach is flexible because it can be used for a library/language of any complexity. On the other hand, our approach allows one to use any program synthesizer, even constraint-based ones [13], and to synthesize programs that meet logical (and not just example-based) specifications (e.g., as in [20]) our approach provides an explicit logical representation of the program semantics, whereas example-based approaches are limited to generate-and-test synthesis techniques, such as program enumeration [10] and example-based specifications.

*Synthesis of Recursive Programs.* At a high level, the semantics-synthesis problem we consider is similar to a number of works on synthesizing recursively defined programs [7, 8, 14, 19]. In effect, a semantics for a recursively defined grammar is a recursive program assigning meaning to programs within the language. Both Farzan et al. [7], Farzan and Nicolet [8] use recursion skeletons to reduce their task from synthesizing a recursive program to synthesizing a non-recursive program. Our use of semantic constraints plays a similar role. While both of their techniques assume programs are only structurally recursive (i.e., no recursion on the program term itself), and our framework explicitly allows for program terms that are self-recursive (e.g., `while` loops in Imp).

Similar to the approach used by Miltner et al. [19] to synthesize simple recursive programs, SemSynth employs a bottom-up approach to synthesis (i.e., we first synthesize semantics for nullary productions before moving on to other productions). However, unlike Miltner et al., SemSynth is well-defined for any ordering of production rules and targets a more complex setting—i.e., synthesizing program semantics.

Finally, Lee and Cho [14] synthesize recursive procedures from examples by first finding likely sub-expressions that can be used to build a complex recursive program and then guessing the recursive structure of the program. The key difference is that our approach targets a more restricted program, synthesizing program semantics, and therefore already has the recursive structure in hand

(thanks to the presence of the grammar in the problems Synantic is solving). Because we have limited the synthesis target to an inductively defined program semantics, Synantic can directly focus its effort on synthesizing the semantic functions of each CHC using well-known synthesis techniques.

*Datalog Synthesis.* Albarghouthi et al. [1] synthesize Datalog programs (i.e., Horn clauses) with SMT solvers, whereas Si et al. [21] use a syntax-guided approach. In our work, we use constrained Horn clauses, which are strictly more expressive than Datalog programs, to denote semantics. Aside from the fact that the Datalog-synthesis problem considers different inputs (i.e., the data), a CHC also contains a function in a theory $\mathcal{T}$ (such as LIA or BV), which Synantic has to synthesize.

*Synthesizing Attribute Grammars.* Kalita et al. [12] proposed a sketch-based method for synthesizing attribute grammars. When provided with a context-free grammar, their tool can automatically create appropriate *semantic actions* from sketches of attribute grammars. Instead of semantic actions, in our work we use CHCs to express program semantics. Our approach can model recursive semantics whereas the technique by Kalita et al. is limited to non-circular attribute grammars. Additionally, while their method requires providing a distinct program sketch (i.e., a partial program) for each production, our approach only requires providing a (fairly general) SyGuS grammar for each nonterminal in the language.

## 8 Conclusion

Writing logical semantics for a language can be a difficult task and our work supplies a method to automatically synthesize a language's semantics from an executable interpreter that is treated as a closed-box. By generating example terms and input-output pairs from the interpreter, we use a SyGuS solver to synthesize semantic rules. Our evaluation shows that the approach applies to a wide range of language features, e.g., recursive semantic functions with multiple outputs.

As discussed in Section 2, one motivation for this work is to be able to generate automatically the kind of semantics that is needed to create a program synthesizer using the SemGuS framework. In our algorithm, we harness a SyGuS solver to synthesize the constraint in each CHC—i.e., we harness SyGuS in service to SemGuS—which limits us to synthesizing constraints that are written in theories that SyGuS supports. Going forward, we would like to make use of "higher-level" theories, supporting such abstractions as stores or algebraic data types. As SemGuS-based synthesizers and verifiers improve, we might be able to satisfy this wish by using SemGuS in service to SemGuS! That is, we could extend Synantic to use SemGuS solvers to synthesize semantic constraints.

## Data-Availability Statement

The artifact that contains Synantic and all benchmark data is available on Zenodo [16].

## Acknowledgments

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

[1] Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. 2017. Constraint-based synthesis of datalog programs. In *Principles and Practice of Constraint Programming: 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28–September 1, 2017, Proceedings 23*. Springer, 689–706.

[2] Rajeev Alur, Rastislav Bodík, 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 *Formal Methods*

*in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 1–8. https://ieeexplore.ieee.org/document/6679385/

[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 the construction and analysis of systems*. Springer, 319–336.

[4] Haniel Barbosa, Clark 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*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 415–442.

[5] Xiaohong Chen and Grigore Rosu. 2019. A Semantic Framework for Programming Languages and Formal Analysis. In *Engineering Trustworthy Software Systems - 5th International School, SETSS 2019, Chongqing, China, April 21-27, 2019, Tutorial Lectures (Lecture Notes in Computer Science, Vol. 12154)*, Jonathan P. Bowen, Zhiming Liu, and Zili Zhang (Eds.). Springer, 122–158. https://doi.org/10.1007/978-3-030-55089-9_4

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

[7] Azadeh Farzan, Danya Lette, and Victor Nicolet. 2022. Recursion synthesis with unrealizability witnesses. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 244–259.

[8] Azadeh Farzan and Victor Nicolet. 2021. Counterexample-Guided Partial Bounding for Recursive Function Synthesis. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33*. Springer, 832–855.

[9] Kangjing Huang and Xiaokang Qiu. 2022. Bootstrapping Library-Based Synthesis. In *International Static Analysis Symposium*. Springer, 272–298.

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

[11] Larry G. Jones. 1990. Efficient Evaluation of Circular Attribute Grammars. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 429–462. https://doi.org/10.1145/78969.78971

[12] Pankaj Kumar Kalita, Miriyala Jeevan Kumar, and Subhajit Roy. 2022. Synthesis of semantic actions in attribute grammars. In *2022 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 304–314.

[13] 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.

[14] Woosuk Lee and Hangyeol Cho. 2023. Inductive synthesis of structurally recursive functional programs from non-recursive expressions. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 2048–2078.

[15] Junghee Lim and Thomas W. Reps. 2013. TSL: A System for Generating Abstract Interpreters and its Application to Machine-Code Analysis. *ACM Trans. Program. Lang. Syst.* 35, 1 (2013), 4:1–4:59. https://doi.org/10.1145/2450136.2450139

[16] Jiangyi Liu, Charlie Murphy, Anvay Grover, Keith Johnson, Thomas Reps, and Loris D'Antoni. 2024. *Artifact of paper "Synthesizing Formal Semantics from Executable Interpreters"*. https://doi.org/10.5281/zenodo.13368062

[17] Jiangyi Liu, Charlie Murphy, Anvay Grover, Keith J. C. Johnson, Thomas Reps, and Loris D'Antoni. 2024. Synthesizing Formal Semantics from Executable Interpreters. arXiv:2408.14668 [cs.PL] https://arxiv.org/abs/2408.14668

[18] Eva Magnusson and Görel Hedin. 2003. Circular Reference Attributed Grammars - Their Evaluation and Applications. In *Workshop on Language Descriptions, Tools and Applications, LDTA@ETAPS 2003, Warsaw, Poland, April 12-13, 2003 (Electronic Notes in Theoretical Computer Science, Vol. 82)*, Barrett R. Bryant and João Saraiva (Eds.). Elsevier, 532–554. https://doi.org/10.1016/S1571-0661(05)82627-1

[19] Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up synthesis of recursive functional programs using angelic execution. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–29.

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

[21] 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*. 515–527.

Received 2024-04-06; accepted 2024-08-18