Synthesizing Formal Semantics from Executable Interpreters

ANONYMOUS AUTHOR(S)*

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.

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, 11, 13]. A user of such a framework must define the language of interest by giving both a syntactic specification and a formal semantic specification 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. 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.

2475-1421/2018/1-ART1 $15.00
https://doi.org/

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, 11], 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:

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

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.

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.

2 ILLUSTRATIVE EXAMPLE

Suppose we have an imperative language IMP (cf. Example 2.1), and we want to synthesize an IMP program to automate some programming tasks. However, we find that there is no synthesizer for IMP. Next, we remember that the SemGuS framework can synthesize programs for user-defined languages [11]. However, when we start trying to write our synthesis problem using SemGuS, we quickly realize IMP's semantics is defined using an interpreter and we do not have a formal-logic-based semantics (i.e., a set of Constrained Horn Clauses), which SemGuS requires. At this point we are stuck, and cannot synthesize any IMP program unless we manually write the IMP semantics as a set of CHCs, a tedious and error-prone task.

In this paper, we consider the problem of synthesizing a formal (logical) semantics for a language from an executable interpreter. We use the IMP language described in Example 2.1 as a running example in this section to illustrate our algorithm (and return to it in Section 4).

Example 2.1 (Syntactic Definition of IMP). Consider the grammar G_{IMP_n} that defines the syntax of IMP for programs with n variables $\mathtt{x_1}, \dots, \mathtt{x_n}$:

$$
\begin{aligned}
S &::= \mathtt{x_1} \coloneqq E \mid \cdots \mid \mathtt{x_n} \coloneqq E \mid S \,\mathtt{;}\, S \mid \mathtt{ite}\ B\ S\ S \mid \mathtt{while}\ B\ \mathtt{do}\ S \\
&\quad \mid \mathtt{do}\ S\ \mathtt{while}\ B \mid \mathtt{repeat}\ S\ \mathtt{until}\ B \\
B &\coloneqq \mathtt{false} \mid \mathtt{true} \mid \neg B \mid B \wedge B \mid B \vee B \mid E < E \\
E &\coloneqq \mathtt{0} \mid \mathtt{1} \mid \mathtt{x_1} \mid \cdots \mid \mathtt{x_n} \mid E + E \mid E - E
\end{aligned}
$$

The IMP language consists of arithmetic and Boolean expressions, statements for assignment to the variables $\mathtt{x_1}$ through $\mathtt{x_n}$, 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, $\mathtt{x_i} \coloneqq e$ 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_{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 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 $\textsc{Imp}_1$, this means we synthesize a semantics for the productions `0`, `1`, $\mathtt{x}_1$, `false`, and `true` before we synthesize the semantics of any other productions. For a nullary production `p`, we synthesize a semantics of the form:

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

which states that, because the term `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 $\mathcal{I}_{\textsc{Imp}}$ on production p. Within the CEGIS loop, we synthesize a candidate function f, then verify if it is consistent with $\mathcal{I}_{\textsc{Imp}}$ (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 $\mathtt{t}_1 + \mathtt{t}_2$, our algorithm generates a semantics that can rely on the semantics of its sub-terms $\mathtt{t}_1$ and $\mathtt{t}_2$. Specifically, the semantics of $\mathtt{t}_1 + \mathtt{t}_2$ takes the following form:

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

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

When the semantics of a sub-term $\mathtt{t}_\mathtt{i}$ is known (e.g., for nullary productions), we substitute its learned semantics for $sem(\mathtt{t}_\mathtt{i}, x_i^{in}, x_i^{out})$; otherwise, we approximate its semantics using examples. Again, we use a CEGIS loop to generate examples for the entire term $\mathtt{t}_1 + \mathtt{t}_2$, 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 $\textsc{Imp}$.

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

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

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 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 Imp$_n$ all input and (most) output variables are an n-tuple of variables representing a state of an Imp$_n$ program. Even for just 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 $\texttt{x}_0 := \texttt{t}$ (for 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).

$$\frac{sem(\texttt{t}, x_1^{in}, x_1^{in}) \qquad x_1^{in} = f_1(x_0^{in}) \qquad x_0^{out} = f_0(x_0^{in}, x_1^{out})}{sem(\texttt{x}_0 := \texttt{t}, x_0^{in}, x_0^{out})}\ F$$

$$\frac{sem(\texttt{t}, x_1^{in}, x_1^{in}) \qquad x_1^{in} = g_1(x_0^{in}) \qquad x_0^{out} = g_0(x_0^{in}, x_1^{out})}{sem(\texttt{x}_0 := \texttt{t}, x_0^{in}, x_0^{out})}\ G$$

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 $\texttt{x}_0 := \texttt{t}$. 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) [11]. 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 Σ 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 Σ is denoted by T_Σ. Specifically, T_Σ is the least set such that $\Sigma^0 \subseteq T_\Sigma$ and if $\sigma^k \in \Sigma^k$ and $t_1, \ldots, t_k \in T_\Sigma$, then $\sigma^k(t_1, \ldots, t_k) \in T_\Sigma$. In the remainder 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, Σ is a ranked alphabet, δ is a set of productions over a set of types T, and for each non-terminal $A \in N$, and θ_A (resp. τ_A) assigns A an input-type (resp. output-type) from T. Each production in δ takes the form:

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

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_{IMP} as a Regular Tree Grammar). Consider the Imp language detailed in Section 2, G_{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 $S \to \mathtt{x_1} \mathrel{:=}(E)$, $S \to\ ;(S, S)$, and $S \to \mathtt{while}(B, S)$. For Imp_2 (Imp with two variables x_1 and x_2), θ_E is the type $\mathbb{Z} \times \mathbb{Z}$, representing the state of the two variables, and τ_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 $\mathcal{I}_A : (\mathcal{L}(A) \times \theta_A) \to \tau_A$—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, 11], 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 ϕ 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 θ_A, and x_A^{out} is a variable of type τ_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 Imp_1 language introduced in Section 2; a semantics for Imp_1 uses the semantic relations:

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

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 $A_0 \rightarrow \sigma(A_1, \ldots, A_n) \in G$ 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 $A_0 \rightarrow p(A_1, \ldots, A_n)$ a **semantic rule** for p is a CHC of the form:

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

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 θ_{A_i} for $i > 0$ and τ_{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:

$$\frac{\llbracket s \rrbracket(x_1) = x_1' \quad \llbracket b \rrbracket(x_2) = r_b \quad \llbracket \texttt{do}\ s\ \texttt{while}\ b \rrbracket(x_3) = x_3' \quad r_b \quad x_1 = x_0 \quad x_2 = x_1' \quad x_3 = x_1' \quad x_0' = x_3'}{\llbracket \texttt{do}\ s\ \texttt{while}\ b \rrbracket(x_0) = x_0'}$$

$$\frac{\llbracket s \rrbracket(x_1) = x_1' \quad \llbracket b \rrbracket(x_2) = r_b \quad \neg r_b \quad x_1 = x_0 \quad x_2 = x_1' \quad x_0' = x_1'}{\llbracket \texttt{do}\ s\ \texttt{while}\ b \rrbracket(x_0) = x_0'}$$

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, we define when *Sem* is equivalent to the semantics defined by interpreter $\mathcal{I}$ via an *equivalence oracle.*

Definition 3.8 (Equivalent, Equivalence Oracle). Given an interpreter $\mathcal{I}$ for a language G, a sub-grammar $G' \subseteq G$, 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:

$$I(t, in) = out \Leftrightarrow \llbracket t \rrbracket_{Sem}(in) = out$$

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 $\llbracket t \rrbracket_{Sem}(in) \neq \llbracket t \rrbracket_{\mathcal{I}}(in)$—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 inductively synthesize the semantics of G

Algorithm 1: Semantics-Synthesis Algorithm

Procedure SemSynth $(G, \mathcal{I}, \mathcal{E})$
- $\mathit{Order} \leftarrow$ Some total ordering of productions of G;
- $N \leftarrow |G|$; `// N the number of productions of G.`
- `// Assume, each production is indexed 1 to N according to Order`
- $\mathit{Sem} \leftarrow \lambda p.\ \bot$; `// Maps each production to a semantics.`
- $E \leftarrow \lambda p.\ \emptyset$; `// Maps each production to a set of examples.`
- $i \leftarrow 1$; `// Start from production indexed by 1.`
- **while** $i \leq N$ **do** `// Synthesize semantics one production at a time according to Order.`
  - $\mathit{Sem}[p_i] \leftarrow$ SynthSemanticConstraint(Sem, p_i, E);
  - $\mathit{cex}, p_j \leftarrow$ Verify$(\mathit{Sem}, p_i, \mathcal{I}, \mathcal{E})$;
  - **if** $\mathit{cex} \neq \mathit{None}$ **then** `// Counter-example found for production p_j.`
    - $E[p_j] \leftarrow E[p_j] \cup \{\mathit{cex}\}$; `// Update examples for production p_j.`
    - $i \leftarrow j$; `// Backtrack and resynthesize production p_j's semantics.`
  - **else**
    - $i \leftarrow i + 1$; `// Proceed to synthesize next production's semantics.`
- **return** Sem;

one production at a time. Furthermore, SemSynth uses this flexibility to synthesize the semantics of simpler productions and fragments of the G before synthesizing the semantics of more complex productions/operators. 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 4.4 provide specifications for SynthSemanticConstraint and Verify, which synthesize semantic constraints from examples and verify candidate semantic constraints against the interpreter, respectively. Section 4.3 details how SemSynth synthesizes the semantics of productions from examples. Finally, Section 4.5 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.

Choosing an Order. To begin, SemSynth determines an order to iterate over the productions of G. While any ordering is sound, we assume that SemSynth picks an order *Order* such that the following property holds. Suppose the grammar G is treated as a graph whose nodes are the productions and non-terminals of G and each production $A_0 \to p(A_1, \ldots, A_n)$ induces an edge from A_0 to p and from p to each A_1 through A_n. If there is a path from p_i to p_j, but no path from p_j to p_i then p_i is ordered before p_j by *Order*.

For example, for the Imp language, *Order* will order productions for arithmetic expressions, like `0`, `1`, and `x`, before the production `+`, and all productions for arithmetic expressions before the assignment-statement production `x :=`. Mutually recursive productions (e.g., sequencing (`;`), `ite`, and `while`), may be explored in any order. For simplicity, we index each production by a natural number from 1 to N (the number of productions in G). Next, SemSynth initializes the synthesized semantics *Sem* (marking every production as initially undefined), and E to the example set (cf. definition 4.1) that maps each production to a set of examples.

Synthesizing a Candidate Semantics. After initialization, SemSynth iteratively synthesizes the semantics of productions in the order defined by *Order*. 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_i using SynthSemanticConstraint. The procedure SynthSemanticConstraint returns some semantic constraint for p_i that satisfies the set of examples E. Section 4.2 provides a formal specification of SynthSemanticConstraint's operation.

SemSynth then uses the procedure Verify to determine if the semantics synthesized thus far is consistent with the interpreter $\mathcal{I}$ as determined by the equivalence oracle $\mathcal{E}$. A formal specification of Verify is provided in Section 4.4. If Verify determines that *Sem* is correct, then it returns *None*, and SemSynth advances to attempt to synthesize the semantics of production p_{i+1}. If Verify determines that *Sem* does not match the semantics of $\mathcal{I}$, then it returns some example *cex* for which *Sem* and $\mathcal{I}$ disagree. It additionally places blame on some production p_j's synthesized semantics. SemSynth then updates the set of examples E and backtracks to resynthesize the semantics of p_j.

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 $\mathcal{I}$ maps each production $A_0 \to p(A_1, \ldots, A_n) \in G$ to a finite set of examples of the form $\langle in, p(t_1, \ldots, t_n), out\rangle$, where $t_i \in L(A_i)$ and $\mathcal{I}(p(t_1, \ldots, t_n), in) = out$.

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 $\langle 0, \texttt{0}, 0\rangle$, $\langle 1, \texttt{0}, 0\rangle$, $\langle 1, \texttt{x := 0; x := x + 4}, 4\rangle$, and $\langle 10, \texttt{while 0 < x do x := x - 1}, 0\rangle$; however, an example set for $\mathcal{I}_{\text{Imp}}$ could not include any example of the form $\langle n, \texttt{while 0 < x do x := x + 1}, n'\rangle$ where n (the initial value of `x`) is some non-negative number. Since, `while 0 < x do x := x + 1` would not terminate on the input n. The example $\langle n, \texttt{whie 0 < x do x := x + 1}, n'\rangle$ 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:

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

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 $\langle 0, \texttt{x}_0 + \texttt{1}, 1\rangle$, $\langle 0, \texttt{x}_0, 0\rangle$, and $\langle 0, \texttt{1}, 1\rangle$. 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 current semantics Sem, the production p_i 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_i that is consistent with the example set E.

Example 4.5 (Synthesizing semantics of `x :=` consistent with examples). Recall that for the language Imp, the semantics of the production `x :=` is represented as (a set of) CHC rule(s) of the form:

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

for some functions f and g (in the theory of linear integer arithmetic). The procedure call SynthSemanticConstraint(Sem, `x :=`, 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 Synthesizing from Examples

To synthesize the semantics of a production, SemSynth uses the counter-example guided inductive synthesis (CEGIS) paradigm. At a high level, SemSynth synthesizes a candidate semantics—for productions p_1 through p_i, one production at a time—from a set of examples. If the candidate semantics is incorrect, a counter-example is produced (by Verify), which is added to the set of examples, and a new candidate semantics is synthesized. We illustrate how SemSynth uses and generates examples to synthesize the semantics of a nullary production in Example 4.6.

4.4 Specification of Verify

The procedure Verify takes as input the currently synthesized semantics Sem, the production p_i whose semantics was just synthesized, the interpreter $\mathcal{I}$, and equivalence oracle $\mathcal{E}$; it determines if Sem is equivalent to the interpreter $\mathcal{I}$ for all terms in the sub-grammar G' that consists of only the productions p_1 through p_i. If Verify determines that Sem is not equivalent to $\mathcal{I}$, it returns a counter-example (i, t, o) and production p_j (with $1 \leq j \leq i$) such that t's root production is p_j, $[\![t]\!]_{\mathcal{I}}(i) = o$, and $[\![t]\!]_{Sem}(i) \neq o$. Otherwise, Verify returns *None* to signify that Sem is equivalent to $\mathcal{I}$ for all terms within the sub-grammar G'.

Example 4.6 (Synthesizing Semantics of `0` for G_{IMP}.). Recall the Imp language in Example 2.1.

SemSynth first synthesizes the semantics of the leaves of G_{Imp}. Assume that *Order* assigns the production `0` index 1 (i.e., it is the first production explored by SemSynth). During the first iteration of SemSynth, the example set is 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 $\langle 0, \texttt{0}, 0\rangle$, and the example set E is updated.

In the next iteration, SynthSemanticConstraint must return a constraint satisfying the updated example set. For example, suppose that SynthSemanticConstraint returns the constraint $x_0^{out} = x_1^{in}$. Again, Verify determines that $x_0^{out} = x_1^{in}$ is incorrect and returns the new counter-example $\langle 1, 0, 0\rangle$. The example set E is updated with the returned counter-example.

A new iteration of the loop is run. On this loop, SynthSemanticConstraint must return a constraint that satisfies both of the previously returned examples. This time SynthSemanticConstraint returns the constraint $x_0^{out} = 0$. This time, 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 $S \longrightarrow ;(S, S)$). After several attempts at synthesizing the semantics of sequencing, $E(;)$ contains the examples $\langle 0, \texttt{x := 1; x := 0}, 0\rangle$, $\langle 0, \texttt{x := 0; x := x + 1}, 1\rangle$, and $\langle 1, \texttt{x := 0; (x := 1; x := x + 1)}, 2\rangle$.

As in the nullary case, we summarise the semantics of each example sub-term using the example set E. SemSynth then generates the formula specifying that the desired semantic constraint satisfies 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 whether it is correct, and SemSynth proceeds to synthesize a semantics for the next production.

4.5 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 $\langle in, p(t_1, \dots, t_n), out\rangle$, the following conditions hold:

$$\frac{\begin{array}{c} Sem_{A_1}(t_1, x_{A_1}^{in}, x_{A_1}^{out}) \\ \dots \quad Sem_{A_n}(t_n, x_{A_n}^{in}, x_{A_n}^{out}) \quad \neg P_{rec}(x^{in}A_0, x_{A_1}^{out}, \dots, x_{A_n}^{out}) \quad F_{non-rec}(x^{in}A_0, x_{A_1}^{out}, \dots, x_{A_n}^{out}) \quad x_{A_0}^{in} = in \end{array}}{x_{A_0}^{out} = out}\ \text{non-rec}$$

$$\frac{\begin{array}{c} Sem_{A_1}(t_1, x_{A_1}^{in}, x_{A_1}^{out}) \quad \dots \quad Sem_{A_n}(t_n, x_{A_n}^{in}, x_{A_n}^{out}) \\ Sem_{A_0}(p(t_1, \dots, t_n), x_{A_0}^{in\,\prime}, x_{A_0}^{out\,\prime}) \quad P_{rec}(x^{in}A_0, x_{A_1}^{out}, \dots, x_{A_n}^{out}) \quad F_{rec}(x^{in}A_0, x_{A_1}^{out}, \dots, x_{A_n}^{out}) \quad x_{A_0}^{in} = in \end{array}}{x_{A_0}^{out} = out}\ \text{rec}$$

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 reaches the `while` production. We assume that the grammar G additionally annotates whether each production is semantically recursive.

After several more iterations, the set $E(\texttt{while})$ contains the examples $\langle 0, t, 0\rangle$, $\langle 1, t, 0\rangle$, and $\langle 1, t, 0\rangle$, where t is the term $\texttt{while } 0 < \texttt{x} \texttt{ do } \texttt{x} := \texttt{x} - 1$. In this iteration, SynthSemanticConstraint gets called with a recursive summary of t containing the three examples, and the exact synthesized semantics for $\texttt{x} := \texttt{x} - 1$ and $0 < \texttt{x}$.

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 terminates (because `while` is the last production in the grammar). Finally, SemSynth 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. In SemSynth, we maintain the loop invariant that the synthesized semantics *Sem* is correct with respect to the oracle $\mathcal{E}$ for productions p_1 through p_{i-1}. This condition trivially holds on the first iteration. To proceed to iteration $i + 1$, Verify must return *None*, which implies that *Sem* is correct for productions p_1 through p_i. Thus the invariant is maintained. Upon back-tracking, the invariant is trivially true (because it held for some greater iteration). Thus, upon termination with $i = N + 1$, *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 ensures that once a semantics is explored during an iteration of SemSynth, it is never explored in any future iterations of SemSynth. However, the formal statement relaxes this condition because back-tracking may cause (a finite number of) future iterations to explore the same semantics.

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 on iteration k of the main loop with current synthesized semantics Sem_k, then for some iteration $k_0 > k$, for all iterations $k' \geq k_0$, Sem will never take the value Sem_k again (i.e., $Sem_{k'} \neq Sem_k$).*

Proof. Assume the negation holds, i.e., "$\forall k_0 > k, \exists k' \geq k_0, Sem_{k'} = Sem_k$". Suppose that at iteration k, the example set is E_k, and at iteration k_0, the example set is E_0. Given that $Sem_{k'} = Sem_k$, those two semantics are defined for the same set of productions $\{p_1, \ldots, p_c\}$. Thus, line 10–12 of Algorithm 1 must be executed at least once between the iterations k and k_0 (both inclusive). If we consider the first execution of line 11 after iteration k (and let k_1 denote the current iteration at that moment), we have that for production p_j, where $(1 \leq j \leq c)$, $E_k[p_j] \cup \{cex\} \subset E_0[p_j]$, where $cex = (in, t, out)$ satisfies $[\![t]\!]_{Sem_k}(in) = [\![t]\!]_{Sem_{k'}}(in) \neq out$. This situation will never happen, because after cex is added to E in iteration k_1, on line 8, the constraints in Section 4.2 do not allow $[\![t]\!]_{Sem_{k'}}(in) \neq out$. Thus we reach 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 SynthSemanticConstraint 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 8 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 also parameterized on the procedure Verify (line 9), which uses the equivalence oracle $\mathcal{E}$ to determine if the learned semantics *Sem* is consistent with the interpreter for all terms that only uses the productions p_1 through p_i. 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_j (for $1 \leq j \leq i$), the example and production p_j are returned 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 input—i.e., we assume the program does not terminate—if its run exceeds the recursion depth.

5.3 Optimized SynthSemanticConstraint for Multi-Output Productions

In Section 5.1, we described how SynthSemanticConstraint produces and uses cvc5 to solve 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.

To address this issue, we modified SynthSemanticConstraint to synthesize 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. For simplicity, we explain how this optimization works for a production that has two outputs. 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,

$$
\begin{aligned}
F &\triangleq x_1 = f_1(x_0) \wedge \cdots \wedge x_n = f_n(x_0, x_1', \ldots, x_{n-1}') \wedge {x_0}_0' = f_0(x_0, x_1', \ldots, x_n') \\
G &\triangleq x_1 = g_1(x_0) \wedge \cdots \wedge x_n = g_n(x_0, x_1', \ldots, x_{n-1}') \wedge {x_0}_1' = g_0(x_0, x_1', \ldots, x_n')
\end{aligned}
$$

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

$$
\begin{aligned}
x_0^F = x_0^G = in \wedge Summary(t_1)(x_1^F, x_1'^F) \wedge \cdots \wedge Summary(t_1)(x_n^F, x_n'^F) && (4)\\
\wedge\, Summary(t_1)(x_1^G, x_1'^G) \wedge \cdots \wedge Summary(t_1)(x_n^G, x_n'^G) && (5)\\
\wedge\, F \wedge G \wedge (x_1^F \neq x_1^G \vee \cdots \vee x_n^F \neq x_n^G) \wedge \left\langle {x_0}_0'^F, {x_0}_1'^G \right\rangle = out && (6)
\end{aligned}
$$

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 $\texttt{x}_0$:= in the language Imp$_2$, using the examples: $\langle\langle 0, 1\rangle, \texttt{x}_0 := \texttt{x}_1, \langle 1, 1\rangle\rangle$, $\langle\langle 0, 1\rangle, \texttt{x}_1, 1\rangle$, $\langle\langle 1, 1\rangle, \texttt{x}_1, 1\rangle$.
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 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 (4) 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 ϕ in Equation (4). If ϕ 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 $F \wedge {x_0}_2 = g_0(\ldots)$ or $G \wedge {x_0}_1 = f_0(\ldots)$ (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 ϕ 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 ϕ. 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 14 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 [11]. 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 10 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 10 languages gave us 10 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(k) 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 4.5). The complete semantics of Imp(k) can be found in the supplementary material.

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 [10]. Our second source of benchmarks is from the Panini tool for synthesizing attribute grammars [10]. 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 [9, 14]—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. [10] 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 $11/14 \approx 79\%$ of benchmark languages.

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 Imp(k) $(k = 1, 2)$, Synantic could synthesize a semantics up to $k = 2$. 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 $n = 2$.

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 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. Table 2 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 15.09% of the total solving time is spent on SyGuS problems, and a median of 20.67% of the time is spent solving SMT queries. However, for the slowest 10% production rules (>31.92 s), the median of SyGuS solving time grows to 65.99%, which indicates that SyGuS contributes to most of the execution time on slow-running cases.

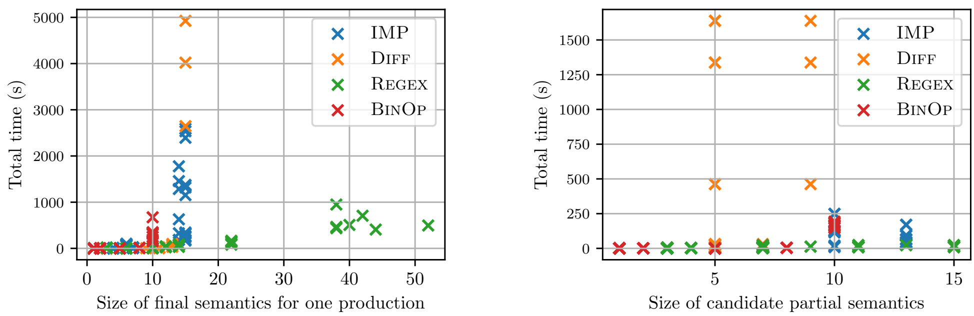
[2]Data for some languages are only listed in the supplementary material.

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)
Imp(2)	$E \rightarrow \texttt{0}$	1	1	0.01	0.01	0.05
	$E \rightarrow \texttt{1}$	1	1	0.01	0.01	0.04
	$S \rightarrow \texttt{x}\,--$	2	2	0.06	0.02	0.11
	$S \rightarrow \texttt{y}\,--$	2	2	0.11	0.03	0.17
	$B \rightarrow \texttt{f}$	1	1	0.01	0.01	0.06
	$S \rightarrow \texttt{x}\,++$	2	2	0.04	0.03	0.11
	$S \rightarrow \texttt{y}\,++$	2	2	0.12	0.02	0.16
	$B \rightarrow \texttt{t}$	1	1	0.01	0.02	0.13
	$E \rightarrow \texttt{x}$	2	2	0.01	0.01	0.04
	$E \rightarrow \texttt{y}$	1	1	0.01	0.01	0.04
	$S \rightarrow \texttt{x} := E$	2	2	0.10	3.23	6.17
	$S \rightarrow \texttt{y} := E$	2	2	0.04	3.22	6.19
	$B \rightarrow \neg B$	3	3	0.02	2.49	5.26
	$E \rightarrow E + E$	4	3	0.05	8.52	14.83
	$E \rightarrow E - E$	5	2	0.13	8.03	13.83
	$B \rightarrow E < E$	8	5	0.08	7.50	13.66
	$B \rightarrow B \wedge B$	4	4	0.03	5.33	11.71
	$B \rightarrow B \vee B$	4	4	0.05	4.61	8.99
	$S \rightarrow S \,;\, S$	5	3	4.55	15.00	72.53
	$S \rightarrow \texttt{do\_while}\ S\ B$	27	35	858.50	257.33	1374.13
	$S \rightarrow \texttt{while}\ B\ S$	9	7	16.88	122.41	266.80
	$S \rightarrow \texttt{ite}\ B\ S\ S$	11	5	525.28	33.88	628.71
BinOp	$B \rightarrow \texttt{0}$	1	1	0.01	0.01	0.07
	$B \rightarrow \texttt{1}$	1	1	0.01	0.01	0.22
	$B \rightarrow \texttt{x}$	2	2	0.01	0.01	0.08
	$N \rightarrow \texttt{atom}\ B$	2	2	0.09	0.04	0.30
	$M \rightarrow \texttt{atom}'\ B$	3	3	0.07	0.05	0.26
	$S \rightarrow \texttt{bin2dec}\ M$	2	2	0.02	0.09	0.30
	$S \rightarrow \texttt{count}\ N$	2	2	0.04	0.05	0.24
	$N \rightarrow \texttt{concat}\ N\ B$	5	5	8.61	0.22	10.31
	$M \rightarrow \texttt{concat}'\ M\ B$	5	5	288.81	0.23	308.50
RegEx(2)	$Start \rightarrow \texttt{eval}\ R$	3	3	0.02	4.43	13.40
	$R \rightarrow ?$	3	3	3.84	0.07	4.07
	$R \rightarrow \texttt{a}$	4	4	11.10	0.07	11.53
	$R \rightarrow \texttt{b}$	5	5	11.63	0.06	12.01
	$R \rightarrow \epsilon$	1	1	0.07	0.07	2.38
	$R \rightarrow \emptyset$	1	1	0.19	0.07	0.46
	$R \rightarrow\, !R$	5	5	2.85	15.77	77.36
	$R \rightarrow R^*$	6	6	0.99	13.06	31.91
	$R \rightarrow R \cdot R$	24	24	333.71	72.58	495.45
	$R \rightarrow R \mid R$	10	10	10.96	59.54	140.82

90% of the per-production semantics are solved within 31.92 s. The 10 rules that take longer than 31.92 s to be synthesized are all non-leaf rules and their partial semantic constraint fall into the following three categories: (*i*) 3 of them contain large integers or complex SMT primitives (e.g., 32-bit integer division); (*ii*) 3 of them involve large logical formulas with sizes ranging between 8 and 24 subterms, e.g., formulas representing 3×3 matrix multiplication or other matrix operations; (*iii*) 4 of them contain two or more input and output variables, e.g., `while` and `do_while`. In particular, Synantic takes 1374.13 s to synthesize the CHC for `do_while` because there can be many possible ways to modify the data flow between its child terms, and this aspect will incur in many CEGIS iterations. In all of the above cases, as expected from known limitations of CVC5, the SyGuS solver accounts for most of the execution time—74.51% of the total running time is spent calling the SyGuS solver and the last call to SyGuS solver takes on average 27.45% of the total running time.

Relation to CEGIS Iterations and Size of Solutions. Table 2 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



(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.

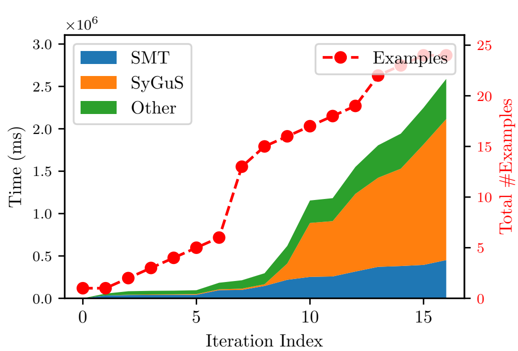


Fig. 2. Execution time per iteration for `do_while`

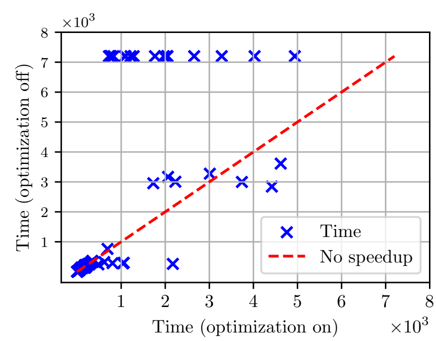


Fig. 3. Speedup provided by optimization

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.

Since the performance varies heavily across different benchmarks, to better understand the impact of CEGIS iterations, we focus our attention on one hard benchmark, IMP(k) where $k = 2$. 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 2 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.

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

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

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 3 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 4a, whereas Synantic synthesizes the CHC shown in Figure 4b. The two CHCs are not logically equivalent. For example, if the children evaluate to the matrices $M = \begin{pmatrix} \mathit{true} & \mathit{false} & \mathit{false} \\ & \mathit{false} & \mathit{false} \\ & & \mathit{true} \end{pmatrix}$ and $M' = \begin{pmatrix} \mathit{true} & \mathit{false} & \mathit{false} \\ & \mathit{false} & \mathit{false} \\ & & \mathit{true} \end{pmatrix}$, the outputs computed by the manually

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

Fig. 5. 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 3×3 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 2×2 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_ϵ 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 5 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

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, 12, 15]. 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. [15] 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 [12] synthesize recursive procedures from examples by first synthesizing blocks of straight-line code. This approach is similar in fashion to how we synthesize semantics by synthesizing semantic constraints. Unlike SemSynth, Lee and Cho do not use CEGIS to perform synthesis. Instead, they use a finite number of input examples to discriminate between recursive programs within the desired search space.

Datalog Synthesis Albarghouthi et al. [1] synthesize Datalog programs (i.e., Horn clauses) with SMT solvers, whereas Si et al. [16] 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), CHC also contains a function in a theory $\mathcal{T}$ (such as LIA or BV), which we have to synthesize.

Synthesizing attribute grammars Kalita et al. [10] 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.

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] 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

[10] 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.

[11] 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.

[12] 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.

[13] 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

[14] 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

[15] 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.

[16] 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.

A SEMANTICS FOR LANGUAGES USED IN BENCHMARK

In this section, we present the semantics synthesized by our tool SYNANTIC for any languages referenced within the main text. Appendix A.1 provides the synthesized semantics of SemGuS benchmarks and Appendix A.2 presents the synthesized semantics of the attribute grammar benchmarks.

A.1 SemGuS Benchmarks

The SemGuS suite of benchmarks consists of a total of 11 languages. We present the synthesized semantics of each as follows:

(1) CNF(k) is depicted in Figure 6.
(2) DNF(k) is depicted in Figure 7.
(3) CUBE(k) is depicted in Figure 8.
(4) INTARITH is depicted in Figure 9.
(5) REGEX(2) is depicted in Figure 10.
(6) REGEXSIMP() is depicted in Figure 11.
(7) IMP is depicted in Figures 12 and 13.
(8) BVSIMPLE(k) is depicted in Figure 14.
(9) BVSATURATED(k) is depicted in Figure 15.
(10) BVIMPSIMPLE(m, n) is depicted in Figure 16.
(11) BVIMPSATURATED(m, n) is depicted in Figure 17.

A.2 Attribute-Grammar Synthesis

The suite of attribute-grammar benchmarks from [Kalita et al. 2022] consists of four languages which we present as follows:

(1) BINOP is presented in Figure 18.
(2) CURRENCY is presented in Figure 19.
(3) DIFF is presented in Figure 20.
(4) ITEEXPR is presented in Figure 21.

B BENCHMARK DATA

We present the full detailed evaluation results for all languages and productions rules in Table 2. For each production we present the number of CEGIS iterations, number of generated examples, and execution time (*i*) to solve SyGuS problems, (*ii*) to solve SMT queries, and (*iii*) overall. For each column we repor the number for the median run based on the total run time. See Section 6 for full description of experimental setup.

$$\frac{i = 0, 1, \ldots, (k-1)}{[\![\texttt{v}_i]\!](x_0, \ldots, x_{k-1}) = x_i}\ \textsc{VarAtom} \qquad \frac{[\![v]\!](x_0, \ldots, x_{k-1}) = r}{[\![\texttt{var}\ v]\!](x_0, \ldots, x_{k-1}) = r}\ \textsc{Var}$$

$$\frac{[\![v]\!](x_0, \ldots, x_{k-1}) = r}{[\![\texttt{nvar}\ v]\!](x_0, \ldots, x_{k-1}) = \neg r}\ \textsc{NotVar} \qquad \frac{[\![c]\!](x_0, \ldots, x_{k-1}) = r}{[\![\texttt{clause}\ c]\!](x_0, \ldots, x_{k-1}) = r}\ \textsc{Clause}$$

$$\frac{[\![c]\!](x_0, \ldots, x_{k-1}) = r_1 \qquad [\![b]\!](x_0, \ldots, x_{k-1}) = r_2}{[\![c \land b]\!](x_0, \ldots, x_{k-1}) = r_1 \land r_2}\ \textsc{And}$$

$$\frac{[\![v]\!](x_0, \ldots, x_{k-1}) = r_1 \qquad [\![c]\!](x_0, \ldots, x_{k-1}) = r_2}{[\![v \lor c]\!](x_0, \ldots, x_{k-1}) = r_1 \lor r_2}\ \textsc{Or}$$

Fig. 6. Semantics of Cnf(k)

$$\frac{i = 0, 1, \ldots, (k-1)}{[\![\texttt{v}_i]\!](x_0, \ldots, x_{k-1}) = x_i}\ \textsc{VarAtom} \qquad \frac{[\![v]\!](x_0, \ldots, x_{k-1}) = r}{[\![\texttt{var}\ v]\!](x_0, \ldots, x_{k-1}) = r}\ \textsc{Var}$$

$$\frac{[\![v]\!](x_0, \ldots, x_{k-1}) = r}{[\![\texttt{nvar}\ v]\!](x_0, \ldots, x_{k-1}) = \neg r}\ \textsc{NotVar} \qquad \frac{[\![c]\!](x_0, \ldots, x_{k-1}) = r}{[\![\texttt{conj}\ c]\!](x_0, \ldots, x_{k-1}) = r}\ \textsc{Conjunction}$$

$$\frac{[\![c]\!](x_0, \ldots, x_{k-1}) = r_1 \qquad [\![b]\!](x_0, \ldots, x_{k-1}) = r_2}{[\![c \lor b]\!](x_0, \ldots, x_{k-1}) = r_1 \lor r_2}\ \textsc{Or}$$

$$\frac{[\![v]\!](x_0, \ldots, x_{k-1}) = r_1 \qquad [\![c]\!](x_0, \ldots, x_{k-1}) = r_2}{[\![v \land c]\!](x_0, \ldots, x_{k-1}) = r_1 \land r_2}\ \textsc{And}$$

Fig. 7. Semantics of Dnf(k)

$$\frac{i = 0, 1, \ldots, (k-1)}{[\![\texttt{v}_i]\!](x_0, \ldots, x_{k-1}) = x_i}\ \textsc{VarAtom} \qquad \frac{[\![v]\!](x_0, \ldots, x_{k-1}) = r}{[\![\texttt{var}\ v]\!](x_0, \ldots, x_{k-1}) = r}\ \textsc{Var}$$

$$\frac{[\![b]\!](x_0, \ldots, x_{k-1}) = r_1 \qquad [\![b]\!](x_0, \ldots, x_{k-1}) = r_2}{[\![b \land b]\!](x_0, \ldots, x_{k-1}) = r_1 \land r_2}\ \textsc{And}$$

Fig. 8. Semantics of Cube(k)

$$\frac{i = 0, 1, \ldots, 3}{\llbracket i \rrbracket(v_0) = i}\ \textsc{IntLiteral} \qquad \frac{}{\llbracket \texttt{t} \rrbracket(v_0) = \mathit{true}}\ \textsc{True} \qquad \frac{}{\llbracket \texttt{f} \rrbracket(v_0) = \mathit{false}}\ \textsc{False}$$

$$\frac{}{\llbracket \texttt{x} \rrbracket(v_0) = v_0}\ \textsc{VarX} \qquad \frac{}{\llbracket \texttt{y} \rrbracket(v_0) = v_0}\ \textsc{VarY} \qquad \frac{}{\llbracket \texttt{z} \rrbracket(v_0) = v_0}\ \textsc{VarZ}$$

$$\frac{\llbracket b \rrbracket(v_0) = r_0 \qquad \llbracket e_1 \rrbracket(v_0) = r_1 \qquad \llbracket e_2 \rrbracket(v_0) = r_2 \qquad r_0}{\llbracket \texttt{ite}\ b\ e_1\ e_2 \rrbracket(v_0) = r_1}\ \textsc{Ite1}$$

$$\frac{\llbracket b \rrbracket(v_0) = r_0 \qquad \llbracket e_1 \rrbracket(v_0) = r_1 \qquad \llbracket e_2 \rrbracket(v_0) = r_2 \qquad \neg r_0}{\llbracket \texttt{ite}\ b\ e_1\ e_2 \rrbracket(v_0) = r_2}\ \textsc{Ite2}$$

$$\frac{\llbracket e_1 \rrbracket(v_0) = r_1 \qquad \llbracket e_2 \rrbracket(v_0) = r_2}{\llbracket e_1 + e_2 \rrbracket(v_0) = r_1 + r_2}\ \textsc{Plus} \qquad \frac{\llbracket e_1 \rrbracket(v_0) = r_1 \qquad \llbracket e_2 \rrbracket(v_0) = r_2}{\llbracket e_1 \times e_2 \rrbracket(v_0) = r_1 \cdot r_2}\ \textsc{Multiply}$$

$$\frac{\llbracket e_1 \rrbracket(v_0) = r_1 \qquad \llbracket e_2 \rrbracket(v_0) = r_2}{\llbracket e_1 < e_2 \rrbracket(v_0) = (r_1 < r_2)}\ \textsc{LessThan} \qquad \frac{\llbracket b_1 \rrbracket(v_0) = r_1 \qquad \llbracket b_2 \rrbracket(v_0) = r_2}{\llbracket \texttt{and}\ b_1\ b_2 \rrbracket(v_0) = (r_1 \wedge r_2)}\ \textsc{And}$$

$$\frac{\llbracket b_1 \rrbracket(v_0) = r_1 \qquad \llbracket b_2 \rrbracket(v_0) = r_2}{\llbracket \texttt{or}\ b_1\ b_2 \rrbracket(v_0) = (r_1 \vee r_2)}\ \textsc{Or} \qquad \frac{\llbracket b \rrbracket(v_0) = r}{\llbracket \texttt{not}\ b \rrbracket(v_0) = (\neg r)}\ \textsc{Not}$$

Fig. 9. Semantics of IntArith

$$M := \begin{pmatrix} M_{0,0} & M_{0,1} & M_{0,2} \\ & M_{1,1} & M_{1,2} \\ & & M_{2,2} \end{pmatrix} \qquad M' := \begin{pmatrix} M'_{0,0} & M'_{0,1} & M'_{0,2} \\ & M'_{1,1} & M'_{1,2} \\ & & M'_{2,2} \end{pmatrix} \qquad \frac{[\![e]\!](s) = M}{[\![\texttt{eval}\ e]\!](s) = M_{0,2}}\ \textsc{Eval}$$

$$\frac{}{[\![\epsilon]\!](s) = \begin{pmatrix} \mathit{true} & \mathit{false} & \mathit{false} \\ & \mathit{true} & \mathit{false} \\ & & \mathit{true} \end{pmatrix}}\ \textsc{Eps} \qquad \frac{}{[\![\phi]\!](s) = \begin{pmatrix} \mathit{false} & \mathit{false} & \mathit{false} \\ & \mathit{false} & \mathit{false} \\ & & \mathit{false} \end{pmatrix}}\ \textsc{Phi}$$

$$\frac{}{[\![\texttt{a}]\!](s) = \begin{pmatrix} \mathit{false} & (s_0{=}\texttt{a}) & \mathit{false} \\ & \mathit{false} & (s_1{=}\texttt{a}) \\ & & \mathit{false} \end{pmatrix}}\ \textsc{CharA} \qquad \frac{}{[\![\texttt{b}]\!](s) = \begin{pmatrix} \mathit{false} & (s_0{=}\texttt{b}) & \mathit{false} \\ & \mathit{false} & (s_1{=}\texttt{b}) \\ & & \mathit{false} \end{pmatrix}}\ \textsc{CharB}$$

$$\frac{}{[\![?]\!](s) = \begin{pmatrix} \mathit{false} & (s_0{\neq}\epsilon) & \mathit{false} \\ & \mathit{false} & (s_1{\neq}\epsilon) \\ & & \mathit{false} \end{pmatrix}}\ \textsc{Any} \qquad \frac{[\![e_1]\!](s) = M \qquad [\![e_2]\!](s) = M'}{[\![e_1 + e_2]\!](s) = \begin{pmatrix} (M_{0,0}{\vee}M'_{0,0}) & (M_{0,1}{\vee}M'_{0,1}) & (M_{0,2}{\vee}M'_{0,2}) \\ & (M_{1,1}{\vee}M'_{1,1}) & (M_{1,2}{\vee}M'_{1,2}) \\ & & (M_{2,2}{\vee}M'_{2,2}) \end{pmatrix}}\ \textsc{Or}$$

$$\frac{[\![e_1]\!](s) = M \qquad [\![e_2]\!](s) = M'}{[\![e_1 \cdot e_2]\!](s) = \begin{pmatrix} (M_{0,0}{\wedge}M'_{0,0}) & \bigvee_{i=0\ldots1}(M_{0,i}{\wedge}M'_{i,1}) & \bigvee_{i=0\ldots2}(M_{0,i}{\wedge}M'_{i,2}) \\ & (M_{1,1}{\wedge}M'_{1,1}) & \bigvee_{i=1\ldots2}(M_{1,i}{\wedge}M'_{i,2}) \\ & & (M_{2,2}{\wedge}M'_{2,2}) \end{pmatrix}}\ \textsc{Concat}$$

$$\frac{[\![e]\!](s) = M}{[\![e^*]\!](s) = \begin{pmatrix} \mathit{true} & M_{0,1} & M_{0,2}{\vee}(M_{0,1}{\wedge}M_{1,2}) \\ & \mathit{true} & M_{1,2} \\ & & \mathit{true} \end{pmatrix}}\ \textsc{Star} \qquad \frac{[\![e]\!](s) = M}{[\![e^*]\!](s) = \begin{pmatrix} \neg M_{0,0} & \neg M_{0,1} & \neg M_{0,2} \\ & \neg M_{1,1} & \neg M_{1,2} \\ & & \neg M_{2,2} \end{pmatrix}}\ \textsc{Neg}$$

Fig. 10. Semantics of RegEx

$$M := \begin{pmatrix} M_{0,0} & M_{0,1} & M_{0,2} \\ & M_{1,1} & M_{1,2} \\ & & M_{2,2} \end{pmatrix} \qquad M' := \begin{pmatrix} M'_{0,0} & M'_{0,1} & M'_{0,2} \\ & M'_{1,1} & M'_{1,2} \\ & & M'_{2,2} \end{pmatrix} \qquad \frac{[\![e]\!](s) = (M_\epsilon, M)}{[\![\texttt{eval}\ e]\!](s) = M_{0,1}}\ \textsc{Eval}$$

$$\frac{}{[\![\epsilon]\!](s) = \left(true, \begin{pmatrix} false & false \\ & false \end{pmatrix}\right)}\ \textsc{Eps} \qquad \frac{}{[\![\phi]\!](s) = \left(false, \begin{pmatrix} false & false \\ & false \end{pmatrix}\right)}\ \textsc{Phi}$$

$$\frac{}{[\![\texttt{a}]\!](s) = \left(false, \begin{pmatrix} s_0{=}\texttt{a} & false \\ & s_0{=}\texttt{a} \end{pmatrix}\right)}\ \textsc{CharA} \qquad \frac{}{[\![\texttt{b}]\!](s) = \left(false, \begin{pmatrix} s_0{=}\texttt{b} & false \\ & s_0{=}\texttt{b} \end{pmatrix}\right)}\ \textsc{CharB}$$

$$\frac{}{[\![?]\!](s) = \left(false, \begin{pmatrix} s_0{\neq}\epsilon & false \\ & s_0{\neq}\epsilon \end{pmatrix}\right)}\ \textsc{Any} \qquad \frac{[\![e_1]\!](s) = (M_\epsilon, M) \qquad [\![e_2]\!](s) = (M'_\epsilon, M')}{[\![e_1 + e_2]\!](s) = \left(M_\epsilon \vee M'_\epsilon, \begin{pmatrix} M_{0,0} \vee M'_{0,0} & M_{0,1} \vee M'_{0,1} \\ & M_{1,1} \vee M'_{1,1} \end{pmatrix}\right)}\ \textsc{Or}$$

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

$$\frac{[\![e]\!](s) = (M_\epsilon, M)}{[\![e^*]\!](s) = \left(true, \begin{pmatrix} M_{0,0} & M_{0,1} \vee (M_{0,0} \wedge M_{1,1}) \\ & M_{1,1} \end{pmatrix}\right)}\ \textsc{Star} \qquad \frac{[\![e]\!](s) = (M_\epsilon, M)}{[\![e^*]\!](s) = \left(\neg M_\epsilon, \begin{bmatrix} \neg M_{0,0} & \neg M_{0,1} \\ & \neg M_{1,1} \end{bmatrix}\right)}\ \textsc{Meg}$$

Fig. 11. Semantics of RegExSimp

$$\frac{}{[\![\texttt{0}]\!](v_0, v_1) = 0}\;\textsc{Const0} \qquad \frac{}{[\![\texttt{1}]\!](v_0, v_1) = 1}\;\textsc{Const1} \qquad \frac{}{[\![\texttt{t}]\!](v_0, v_1) = \mathit{true}}\;\textsc{ConstT}$$

$$\frac{}{[\![\texttt{f}]\!](v_0, v_1) = \mathit{false}}\;\textsc{ConstF} \qquad \frac{}{[\![\texttt{x}]\!](v_0, v_1) = v_0}\;\textsc{VarX} \qquad \frac{}{[\![\texttt{y}]\!](v_0, v_1) = v_1}\;\textsc{VarY}$$

$$\frac{[\![e_1]\!](v_0, v_1) = v_1 \qquad [\![e_2]\!](v_0, v_1) = v_2}{[\![e_1 \;\texttt{+}\; e_2]\!](v_0, v_1) = v_1 + v_2}\;\textsc{Plus} \qquad \frac{[\![e_1]\!](v_0, v_1) = v_1 \qquad [\![e_2]\!](v_0, v_1) = v_2}{[\![e_1 \;\texttt{-}\; e_2]\!](v_0, v_1) = v_1 - v_2}\;\textsc{Minus}$$

$$\frac{[\![e_1]\!](v_0, v_1) = v_1 \qquad [\![e_2]\!](v_0, v_1) = v_2 \qquad v_1 < v_2}{[\![e_1 \;\texttt{<}\; e_2]\!](v_0, v_1) = \mathit{true}}\;\textsc{LessThanTrue}$$

$$\frac{[\![e_1]\!](v_0, v_1) = v_1 \qquad [\![e_2]\!](v_0, v_1) = v_2 \qquad v_1 \geq v_2}{[\![e_1 \;\texttt{<}\; e_2]\!](v_0, v_1) = \mathit{false}}\;\textsc{LessThanFalse}$$

$$\frac{[\![b_1]\!](v_0, v_1) = v_1 \qquad [\![b_2]\!](v_0, v_1) = v_2}{[\![b_1 \;\texttt{and}\; b_2]\!](v_0, v_1) = v_1 \wedge v_2}\;\textsc{BoolAnd}$$

$$\frac{[\![b_1]\!](v_0, v_1) = v_1 \qquad [\![b_2]\!](v_0, v_1) = v_2}{[\![b_1 \;\texttt{or}\; b_2]\!](v_0, v_1) = v_1 \vee v_2}\;\textsc{BoolOr} \qquad \frac{[\![b]\!](v_0, v_1) = v}{[\![\texttt{not}\; b]\!](v_0, v_1) = \neg v}\;\textsc{BoolNot}$$

$$\frac{[\![e]\!](v_0, v_1) = v}{[\![\texttt{x} \mathrel{:=} e]\!](v_0, v_1) = v}\;\textsc{AssignX} \qquad \frac{[\![e]\!](v_0, v_1) = v}{[\![\texttt{y} \mathrel{:=} e]\!](v_0, v_1) = v}\;\textsc{AssignY}$$

$$\frac{}{[\![\texttt{x}++]\!](v_0, v_1) = v_0 + 1}\;\textsc{IncX} \qquad \frac{}{[\![\texttt{y}++\; e]\!](v_0, v_1) = v_1 + 1}\;\textsc{IncY}$$

$$\frac{}{[\![\texttt{x}--]\!](v_0, v_1) = v_0 - 1}\;\textsc{DecX} \qquad \frac{}{[\![\texttt{y}--]\!](v_0, v_1) = v_1 - 1}\;\textsc{DecY}$$

Fig. 12. Semantics of Imp(2), part 1

$$\frac{[\![s_1]\!](v_0, v_1) = v_0' \qquad [\![s_2]\!](v_0') = v_0''}{[\![s_1;s_2]\!](v_0, v_1) = v_0''}\ \textsc{Seq}$$

$$\frac{[\![b]\!](v_0, v_1) = v \qquad [\![s_1]\!](v_0, v_1) = v_1 \qquad [\![s_2]\!](v_0, v_1) = v_2}{[\![\texttt{if}\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!](v_0, v_1) = v\ ?\ v_1 : v_2}\ \textsc{Ite}$$

$$\frac{[\![b]\!](v_0, v_1) = \mathit{true} \qquad [\![s]\!](v_0, v_1) = v_1 \qquad [\![\texttt{while}\ b\ \texttt{do}\ s]\!](v_1) = v_2}{[\![\texttt{while}\ b\ \texttt{do}\ s]\!](v_0, v_1) = v_2}\ \textsc{WhileLoop}$$

$$\frac{[\![b]\!](v_0, v_1) = \mathit{false}}{[\![\texttt{while}\ b\ \texttt{do}\ s]\!](v_0, v_1) = v_0}\ \textsc{WhileEnd}$$

$$\frac{[\![s]\!](v_0, v_1) = v_1 \qquad [\![b]\!](v_1) = \mathit{true} \qquad [\![\texttt{do}\ s\ \texttt{while}\ b]\!](v_1) = v_2}{[\![\texttt{do}\ s\ \texttt{while}\ b]\!](v_0, v_1) = v_2}\ \textsc{DoWhileLoop}$$

$$\frac{[\![s]\!](v_0, v_1) = v_1 \qquad [\![b]\!](v_1) = \mathit{false}}{[\![\texttt{do}\ s\ \texttt{while}\ b]\!](v_0, v_1) = v_1}\ \textsc{DoWhileEnd}$$

Fig. 13. Semantics of Imp(2), part 2

$$\frac{i = 0, 1, \ldots, (k-1)}{[\![\mathtt{v}_i]\!](x_0, \ldots, x_{k-1}) = x_i}\ \textsc{VarAtom} \qquad \frac{}{[\![\mathtt{0}]\!](x_0, \ldots, x_{k-1}) = \mathtt{0x00000000}}\ \textsc{BvZero}$$

$$\frac{}{[\![\mathtt{1}]\!](x_0, \ldots, x_{k-1}) = \mathtt{0x00000001}}\ \textsc{BvOne}$$

$$\frac{[\![e_1]\!](x_0, \ldots, x_{k-1}) = r_1 \qquad [\![e_2]\!](x_0, \ldots, x_{k-1}) = r_2}{[\![e_1 < e_2]\!](x_0, \ldots, x_{k-1}) = r_1 <_{unsigned} r_2}\ \textsc{BvUlt}$$

$$\frac{[\![e_1]\!](x_0, \ldots, x_{k-1}) = r_1 \qquad [\![e_2]\!](x_0, \ldots, x_{k-1}) = r_2}{[\![e_1 \geq e_2]\!](x_0, \ldots, x_{k-1}) = \neg(r_1 <_{unsigned} r_2)}\ \textsc{BvUge}$$

$$\frac{[\![e_1]\!](x_0, \ldots, x_{k-1}) = r_1 \qquad [\![e_2]\!](x_0, \ldots, x_{k-1}) = r_2}{[\![e_1 \leq e_2]\!](x_0, \ldots, x_{k-1}) = \neg(r_2 <_{unsigned} r_1)}\ \textsc{BvUle}$$

$$\frac{[\![e_1]\!](x_0, \ldots, x_{k-1}) = r_1 \qquad [\![e_2]\!](x_0, \ldots, x_{k-1}) = r_2 \qquad \odot \in \{\&, |, \oplus, \ggg, \lll\}}{[\![e_1 \odot e_2]\!](x_0, \ldots, x_{k-1}) = (r_1 \odot r_2)}\ \textsc{BvBitwise}$$

$$\frac{[\![e]\!](x_0, \ldots, x_{k-1}) = r \qquad r \neq \mathtt{0x00000000}}{[\![\mathtt{any\_bit}\ e]\!](x_0, \ldots, x_{k-1}) = \mathtt{0x00000001}}\ \textsc{AnyBit1}$$

$$\frac{[\![e]\!](x_0, \ldots, x_{k-1}) = r \qquad r = \mathtt{0x00000000}}{[\![\mathtt{any\_bit}\ e]\!](x_0, \ldots, x_{k-1}) = \mathtt{0x00000000}}\ \textsc{AnyBit0} \qquad \frac{[\![e]\!](x_0, \ldots, x_{k-1}) = r}{[\![\sim e]\!](x_0, \ldots, x_{k-1}) = \sim e}\ \textsc{BvNot}$$

$$\frac{[\![e]\!](x_0, \ldots, x_{k-1}) = r}{[\![\neg\ e]\!](x_0, \ldots, x_{k-1}) = \neg e}\ \textsc{BvNeg}$$

$$\frac{[\![e_1]\!](x_0, \ldots, x_{k-1}) = r_1 \qquad [\![e_2]\!](x_0, \ldots, x_{k-1}) = r_2 \qquad \odot \in \{+, -, \times, \div\}}{[\![e_1 \odot e_2]\!](x_0, \ldots, x_{k-1}) = (r_1 \odot r_2)}\ \textsc{BvArith}$$

Fig. 14. Semantics of $\textsc{BvSimple}(k)$

$$\frac{i = 0, 1, \ldots, (k-1)}{[\![\texttt{v}_i]\!](x_0, \ldots, x_{k-1}) = x_i}\ \textsc{VarAtom} \qquad \frac{}{[\![\texttt{0}]\!](x_0, \ldots, x_{k-1}) = \texttt{0x00000000}}\ \textsc{BvZero}$$

$$\frac{}{[\![\texttt{1}]\!](x_0, \ldots, x_{k-1}) = \texttt{0x00000001}}\ \textsc{BvOne}$$

$$\frac{[\![e_1]\!](x_0, \ldots, x_{k-1}) = r_1 \qquad [\![e_2]\!](x_0, \ldots, x_{k-1}) = r_2}{[\![e_1 < e_2]\!](x_0, \ldots, x_{k-1}) = r_1 <_{unsigned} r_2}\ \textsc{BvUlt}$$

$$\frac{[\![e_1]\!](x_0, \ldots, x_{k-1}) = r_1 \qquad [\![e_2]\!](x_0, \ldots, x_{k-1}) = r_2}{[\![e_1 \geq e_2]\!](x_0, \ldots, x_{k-1}) = \neg(r_1 <_{unsigned} r_2)}\ \textsc{BvUge}$$

$$\frac{[\![e_1]\!](x_0, \ldots, x_{k-1}) = r_1 \qquad [\![e_2]\!](x_0, \ldots, x_{k-1}) = r_2}{[\![e_1 \leq e_2]\!](x_0, \ldots, x_{k-1}) = \neg(r_2 <_{unsigned} r_1)}\ \textsc{BvUle}$$

$$\frac{[\![e_1]\!](x_0, \ldots, x_{k-1}) = r_1 \qquad [\![e_2]\!](x_0, \ldots, x_{k-1}) = r_2 \qquad \odot \in \{\&, |, \oplus, \ggg, \lll\}}{[\![e_1 \odot e_2]\!](x_0, \ldots, x_{k-1}) = (r_1 \odot r_2)}\ \textsc{BvBitwise}$$

$$\frac{[\![e]\!](x_0, \ldots, x_{k-1}) = r \qquad r \neq \texttt{0x00000000}}{[\![\texttt{any\_bit}\ e]\!](x_0, \ldots, x_{k-1}) = \texttt{0x00000001}}\ \textsc{AnyBit1}$$

$$\frac{[\![e]\!](x_0, \ldots, x_{k-1}) = r \qquad r = \texttt{0x00000000}}{[\![\texttt{any\_bit}\ e]\!](x_0, \ldots, x_{k-1}) = \texttt{0x00000000}}\ \textsc{AnyBit0} \qquad \frac{[\![e]\!](x_0, \ldots, x_{k-1}) = r}{[\![\sim e]\!](x_0, \ldots, x_{k-1}) = \sim e}\ \textsc{BvNot}$$

$$\frac{[\![e]\!](x_0, \ldots, x_{k-1}) = r}{[\![\neg\ e]\!](x_0, \ldots, x_{k-1}) = \neg e}\ \textsc{BvNeg}$$

$$\frac{[\![e_1]\!](x_0, \ldots, x_{k-1}) = r_1 \qquad [\![e_2]\!](x_0, \ldots, x_{k-1}) = r_2 \qquad \odot \in \{+, -, \times, \div\}}{[\![e_1 \odot e_2]\!](x_0, \ldots, x_{k-1}) = (r_1 \odot_{sat} r_2)}\ \textsc{BvArith}$$

$$\begin{aligned} a \odot_{sat} b &= a \odot b && \text{(when no overflow or underflow)} \\ a \odot_{sat} b &= \texttt{0xffffffff} && \text{(when overflow happens)} \\ a \odot_{sat} b &= \texttt{0x00000000} && \text{(when underflow happens)} \end{aligned}$$

Fig. 15. Semantics of $\textsc{BvSaturated}(k)$

$$\frac{i = 0, 1, \ldots, (m-1)}{[\![\mathtt{v}_i]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = u_i}\ \textsc{ConstAtom}$$

$$\frac{i = 0, 1, \ldots, (n-1)}{[\![\mathtt{o}_i]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = x_i}\ \textsc{VarAtom}$$

$$\frac{[\![e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r}{[\![\mathtt{o}_i \mathrel{:=} e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = (u_0, \ldots, u_{m-1}, x_0, \ldots, x_{i-1}, r, x_{i+1}, \ldots, x_{n-1})}\ \textsc{Assign}$$

$$\frac{[\![s_1]\!](\vec{u}, \vec{x}) = (\vec{u}', \vec{x}') \qquad [\![s_1]\!](\vec{u}', \vec{x}') = (\vec{u}'', \vec{x}'')}{[\![s_1 \mathbin{;} s_2]\!](\vec{u}, \vec{x}) = (\vec{u}'', \vec{x}'')}\ \textsc{Seq} \qquad \vec{u} = (u_0, \ldots, u_{m-1}), \vec{x} = (x_0, \ldots, x_{n-1})$$

$$\frac{}{[\![\mathtt{0}]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = \mathtt{0x00000000}}\ \textsc{BvZero}$$

$$\frac{}{[\![\mathtt{1}]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = \mathtt{0x00000001}}\ \textsc{BvOne}$$

$$\frac{[\![e_1]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_1 \qquad [\![e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_2}{[\![e_1 < e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_1 <_{unsigned} r_2}\ \textsc{BvUlt}$$

$$\frac{[\![e_1]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_1 \qquad [\![e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_2}{[\![e_1 \geq e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = \neg(r_1 <_{unsigned} r_2)}\ \textsc{BvUge}$$

$$\frac{[\![e_1]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_1 \qquad [\![e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_2}{[\![e_1 \leq e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = \neg(r_2 <_{unsigned} r_1)}\ \textsc{BvUle}$$

$$\frac{\begin{array}{c}[\![e_1]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_1 \\ [\![e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_2 \qquad \odot \in \{\&, |, \oplus, \ggg, \lll\}\end{array}}{[\![e_1 \odot e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = (r_1 \odot r_2)}\ \textsc{BvBitwise}$$

$$\frac{[\![e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r \qquad r \neq \mathtt{0x00000000}}{[\![\mathtt{any\_bit}\ e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = \mathtt{0x00000001}}\ \textsc{AnyBit1}$$

$$\frac{[\![e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r \qquad r = \mathtt{0x00000000}}{[\![\mathtt{any\_bit}\ e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = \mathtt{0x00000000}}\ \textsc{AnyBit0}$$

$$\frac{[\![e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r}{[\![\sim e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) =\sim e}\ \textsc{BvNot} \qquad \frac{[\![e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r}{[\![\neg\ e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = \neg e}\ \textsc{BvNeg}$$

$$\frac{[\![e_1]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_1 \qquad [\![e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_2 \qquad \odot \in \{+, -, \times, \div\}}{[\![e_1 \odot e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = (r_1 \odot r_2)}\ \textsc{BvArith}$$

Fig. 16. Semantics of $\textsc{BvImpSimple}(k)$

$$\frac{i = 0, 1, \ldots, (m-1)}{[\![\texttt{v}_i]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = u_i}\ \textsc{ConstAtom}$$

$$\frac{i = 0, 1, \ldots, (n-1)}{[\![\texttt{o}_i]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = x_i}\ \textsc{VarAtom}$$

$$\frac{[\![e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r}{[\![\texttt{o}_i := e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = (u_0, \ldots, u_{m-1}, x_0, \ldots, x_{i-1}, r, x_{i+1}, \ldots, x_{n-1})}\ \textsc{Assign}$$

$$\frac{[\![s_1]\!](\vec{u}, \vec{x}) = (\vec{u}', \vec{x}') \qquad [\![s_1]\!](\vec{u}', \vec{x}') = (\vec{u}'', \vec{x}'')}{[\![s_1\ \texttt{;}\ s_2]\!](\vec{u}, \vec{x}) = (\vec{u}'', \vec{x}'')}\ \textsc{Seq} \qquad \vec{u} = (u_0, \ldots, u_{m-1}), \vec{x} = (x_0, \ldots, x_{n-1})$$

$$\frac{}{[\![\texttt{0}]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = \texttt{0x00000000}}\ \textsc{BvZero}$$

$$\frac{}{[\![\texttt{1}]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = \texttt{0x00000001}}\ \textsc{BvOne}$$

$$\frac{[\![e_1]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_1 \qquad [\![e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_2}{[\![e_1 < e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_1 <_{unsigned} r_2}\ \textsc{BvUlt}$$

$$\frac{[\![e_1]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_1 \qquad [\![e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_2}{[\![e_1 \geq e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = \neg(r_1 <_{unsigned} r_2)}\ \textsc{BvUge}$$

$$\frac{[\![e_1]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_1 \qquad [\![e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_2}{[\![e_1 \leq e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = \neg(r_2 <_{unsigned} r_1)}\ \textsc{BvUle}$$

$$\frac{\begin{array}{c}[\![e_1]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_1 \\ [\![e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r_2 \qquad \odot \in \{\&, |, \oplus, \ggg, \lll\}\end{array}}{[\![e_1 \odot e_2]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = (r_1 \odot r_2)}\ \textsc{BvBitwise}$$

$$\frac{[\![e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r \qquad r \neq \texttt{0x00000000}}{[\![\texttt{any\_bit}\ e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = \texttt{0x00000001}}\ \textsc{AnyBit1}$$

$$\frac{[\![e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r \qquad r = \texttt{0x00000000}}{[\![\texttt{any\_bit}\ e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = \texttt{0x00000000}}\ \textsc{AnyBit0}$$

$$\frac{[\![e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r}{[\![\sim e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) =\sim e}\ \textsc{BvNot} \qquad \frac{[\![e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = r}{[\![\neg\ e]\!](u_0, \ldots, u_{m-1}, x_0, \ldots, x_{n-1}) = \neg e}\ \textsc{BvNeg}$$

$$\frac{[\![e_1]\!](x_0, \ldots, x_{k-1}) = r_1 \qquad [\![e_2]\!](x_0, \ldots, x_{k-1}) = r_2 \qquad \odot \in \{+, -, \times, \div\}}{[\![e_1 \odot e_2]\!](x_0, \ldots, x_{k-1}) = (r_1 \odot_{sat} r_2)}\ \textsc{BvArith}$$

Fig. 17. Semantics of $\textsc{BvImpSaturated}(k)$

$$\frac{}{[\![1]\!](v_0) = 1}\ \textsc{One} \qquad \frac{}{[\![0]\!](v_0) = 0}\ \textsc{Zero} \qquad \frac{}{[\![x]\!](v_0) = v_0}\ \textsc{VarX} \qquad \frac{[\![n]\!](v_0) = r}{[\![\texttt{count}\ n]\!](v_0) = r}\ \textsc{CountBit}$$

$$\frac{[\![m]\!](v_0) = r}{[\![\texttt{bin2dec}\ m]\!](v_0) = r}\ \textsc{BinToDec} \qquad \frac{[\![n]\!](v_0) = r_1 \qquad [\![b]\!](v_0) = r_2}{[\![\texttt{concat}\ n\ b]\!](v_0) = r_1 + \texttt{int}(r_2)}\ \textsc{CountBitConcat}$$

$$\frac{[\![m]\!](v_0) = r_1 \qquad [\![b]\!](v_0) = r_2}{[\![\texttt{concat}'\ m\ b]\!](v_0) = 2 \cdot r_1 + \texttt{int}(r_2)}\ \textsc{BinToDecConcat} \qquad \frac{[\![b]\!](v_0) = r}{[\![\texttt{atom}\ b]\!](v_0) = r}\ \textsc{CountBitAtom}$$

$$\frac{[\![b]\!](v_0) = r}{[\![\texttt{atom}'\ b]\!](v_0) = r}\ \textsc{BinToDecAtom}$$

Fig. 18. Semantics of BinOp

$$\frac{}{[\![0]\!](v_0) = 0}\ \textsc{Zero} \qquad \frac{}{[\![1]\!](v_0) = 1}\ \textsc{One} \qquad \frac{}{[\![2]\!](v_0) = 2}\ \textsc{Two} \qquad \frac{}{[\![4]\!](v_0) = 4}\ \textsc{Four}$$

$$\frac{}{[\![8]\!](v_0) = 8}\ \textsc{Eight} \qquad \frac{}{[\![x]\!](v_0) = v_0}\ \textsc{VarX} \qquad \frac{[\![k_1]\!](v_0) = r_1 [\![k_2]\!](v_0) = r_2}{[\![k_1 +_k k_2]\!](v_0) = r_1 + r_2}\ \textsc{ScalarPlus}$$

$$\frac{[\![s_1]\!](v_0) = r_1 [\![s_2]\!](v_0) = r_2}{[\![s_1 + s_2]\!](v_0) = r_1 + r_2}\ \textsc{CurrencyPlus} \qquad \frac{[\![s_1]\!](v_0) = r_1 [\![s_2]\!](v_0) = r_2}{[\![s_1 - s_2]\!](v_0) = r_1 - r_2}\ \textsc{CurrencySubtract}$$

$$\frac{[\![s]\!](v_0) = r_1 [\![k]\!](v_0) = r_2}{[\![s \times k]\!](v_0) = r_1 \cdot r_2}\ \textsc{CurrencyTimesScalar} \qquad \frac{[\![k]\!](v_0) = r}{[\![\texttt{jpy}\ k]\!](v_0) = r}\ \textsc{CurrencyJpy}$$

$$\frac{[\![k]\!](v_0) = r}{[\![\texttt{cny}\ k]\!](v_0) = 21 \cdot r}\ \textsc{CurrencyCny} \qquad \frac{[\![k]\!](v_0) = r}{[\![\texttt{usd}\ k]\!](v_0) = 152 \cdot r}\ \textsc{CurrencyUsd}$$

Fig. 19. Semantics of Currency

$$\frac{}{[\![0]\!](v_0) = (0, 0, 0))}\ \textsc{Zero} \qquad \frac{}{[\![1]\!](v_0) = (1, 1, 0)}\ \textsc{One} \qquad \frac{}{[\![2]\!](v_0) = (2, 2, 0)}\ \textsc{Two}$$

$$\frac{}{[\![x]\!](v_0) = (v_0, v_0 + 1, 0)}\ \textsc{VarX} \qquad \frac{[\![e_1]\!](v_0) = (r_1, s_1, t_1) [\![e_2]\!](v_0) = (r_2, s_2, t_2)}{[\![e_1 + e_2]\!](v_0) = (r_1 + r_2, s_1 + s_2, t_1 + t_2)}\ \textsc{Plus}$$

$$\frac{[\![e_1]\!](v_0) = (r_1, s_1, t_1) [\![e_2]\!](v_0) = (r_2, s_2, t_2)}{[\![e_1 \times e_2]\!](v_0) = (r_1 \cdot r_2, s_1 \cdot s_2, r_1 \cdot t_2 + r_2 \cdot s_1)}\ \textsc{Multiply}$$

Fig. 20. Semantics of Diff

$$\frac{i = 0, 1, \ldots, 8}{[\![i]\!](v_0) = i}\ \textsc{IntLiteral} \qquad \frac{}{[\![\texttt{x}]\!](v_0) = v_0}\ \textsc{VarX} \qquad \frac{[\![e]\!](v_0) = r}{[\![\texttt{expr}\ e]\!](v_0) = r}\ \textsc{Expr}$$

$$\frac{[\![b]\!](v_0) = r_0 \qquad [\![e_1]\!](v_0) = r_1 \qquad [\![e_2]\!](v_0) = r_2 \qquad r_0}{[\![\texttt{ite}\ b\ e_1\ e_2]\!](v_0) = r_1}\ \textsc{Ite1}$$

$$\frac{[\![b]\!](v_0) = r_0 \qquad [\![e_1]\!](v_0) = r_1 \qquad [\![e_2]\!](v_0) = r_2 \qquad \neg r_0}{[\![\texttt{ite}\ b\ e_1\ e_2]\!](v_0) = r_2}\ \textsc{Ite2}$$

$$\frac{[\![e]\!](v_0) = r_1 \qquad [\![f]\!](v_0) = r_2}{[\![e + f]\!](v_0) = r_1 + r_2}\ \textsc{Plus} \qquad \frac{[\![e]\!](v_0) = r_1 \qquad [\![f]\!](v_0) = r_2}{[\![e - f]\!](v_0) = r_1 - r_2}\ \textsc{Minus}$$

$$\frac{[\![f]\!](v_0) = r}{[\![\texttt{atom}\ f]\!](v_0) = r}\ \textsc{Atom} \qquad \frac{[\![f]\!](v_0) = r_1 \qquad [\![g]\!](v_0) = r_2}{[\![f * g]\!](v_0) = r_1 \cdot r_2}\ \textsc{Multiply}$$

$$\frac{[\![f]\!](v_0) = r_1 \qquad [\![g]\!](v_0) = r_2}{[\![f \div g]\!](v_0) = r_1 \div r_2}\ \textsc{Divide} \qquad \frac{[\![g]\!](v_0) = r}{[\![\texttt{num}\ g]\!](v_0) = r}\ \textsc{Atom}$$

$$\frac{[\![e_1]\!](v_0) = r_1 \qquad [\![e_2]\!](v_0) = r_2 \qquad \ominus \in \{<, \leq, >, \geq, =, \neq\}}{[\![e_1 \ominus e_2]\!](v_0) = (r_1 \ominus r_2)}\ \textsc{Cmp}$$

Fig. 21. Semantics of IteExpr

Table 2. Evaluation results with optimization turned on.[3]

Lang.	Rule	# Iter.	# Ex	SyGuS (s)	SMT (s)	Total (s)
BVSimple(3)	$E \rightarrow \texttt{0}$	1	1	0.01	0.02	0.29
	$E \rightarrow \texttt{1}$	1	1	0.01	0.01	0.14
	$E \rightarrow \texttt{v}_0$	1	1	0.01	0.01	0.13
	$E \rightarrow \texttt{v}_1$	1	1	0.01	0.02	0.11
	$E \rightarrow \texttt{v}_2$	1	1	0.02	0.01	0.09
	$E \rightarrow -E$	2	2	0.09	1.09	1.92
	$E \rightarrow \sim E$	2	2	0.06	0.88	1.65
	$E \rightarrow \texttt{any\_bit}\ E$	4	4	1.93	0.93	3.64
	$E \rightarrow E + E$	9	2	1.98	0.89	4.12
	$E \rightarrow E \,\&\, E$	6	2	2.67	1.47	7.31
	$E \rightarrow E \div E$	6	2	1.39	37.08	62.33
	$E \rightarrow E = E$	19	6	69.51	4.16	77.67
	$E \rightarrow E \ggg E$	9	3	1.72	2.05	5.54
	$E \rightarrow E \times E$	9	2	2.31	0.69	3.98
	$E \rightarrow E \mid E$	10	3	2.55	1.19	4.98
	$E \rightarrow E \ll E$	9	3	1.60	3.13	6.71
	$E \rightarrow E - E$	9	2	1.28	1.61	4.61
	$S \rightarrow E \geq E$	14	6	4.73	2.34	10.39
	$S \rightarrow E \leq E$	13	5	2.55	1.84	6.75
	$S \rightarrow E < E$	6	5	0.46	1.54	3.62
	$E \rightarrow E \oplus E$	9	3	2.00	1.56	5.10
BVSaturated(2)(T7)	$E \rightarrow \texttt{0}$	1	1	0.01	0.01	0.15
	$E \rightarrow \texttt{1}$	1	1	0.01	0.01	0.06
	$E \rightarrow \texttt{v}_0$	1	1	0.01	0.01	0.05
	$E \rightarrow \texttt{v}_1$	1	1	0.01	0.01	0.05
	$E \rightarrow E \,\&\, E$	5	3	0.46	0.94	3.41
	$E \rightarrow E \ggg E$	6.0	3.0	1.07	2.67	5.65
	$E \rightarrow E \times E$	10.0	6.0	404.76	3.09	414.49
	$E \rightarrow E \mid E$	5.5	3.0	1.62	1.30	4.31
	$E \rightarrow E \ll E$	6.0	3.0	0.70	2.03	4.41
	$E \rightarrow E - E$	6.0	5.0	5.46	3.07	12.22
	$E \rightarrow E \oplus E$	5.5	2.5	0.64	1.03	2.94
BVIMPSimple(1, 2)	$E \rightarrow \texttt{0}$	1	1	0.01	0.02	0.36
	$E \rightarrow \texttt{1}$	1	1	0.01	0.03	0.17
	$E \rightarrow \texttt{o0}$	1	1	0.01	0.02	0.10
	$E \rightarrow \texttt{o1}$	1	1	0.01	0.01	0.07
	$E \rightarrow \texttt{v0}$	1	1	0.01	0.01	0.14
	$S \rightarrow \texttt{o0} := E$	1	1	0.35	0.49	1.27
	$S \rightarrow \texttt{o1} := E$	2	2	0.32	0.38	1.47
	$E \rightarrow -E$	2	2	0.07	0.79	1.64
	$E \rightarrow \sim E$	2	2	0.08	0.61	1.47
	$E \rightarrow \texttt{any\_bit}\ E$	4	4	1.30	0.79	2.72
	$E \rightarrow E + E$	5	2	1.19	0.92	3.48
	$E \rightarrow E \,\&\, E$	7	3	3.65	1.46	8.17
	$E \rightarrow E \div E$	7	3	2.58	35.00	60.76
	$E \rightarrow E = E$	20	6	83.47	19.99	108.10
	$E \rightarrow E \ggg E$	9	3	2.52	2.19	6.40
	$E \rightarrow E \times E$	9	3	2.48	0.86	4.39
	$E \rightarrow E \mid E$	9	3	2.00	1.10	4.30
	$E \rightarrow E \ll E$	10	3	1.83	2.59	6.72
	$E \rightarrow E - E$	7	2	1.71	1.52	4.99
	$B \rightarrow S \geq S$	26	8	18.24	2.87	25.36
	$B \rightarrow S \leq S$	23	6	13.59	1.06	16.75
	$B \rightarrow S < S$	6	5	0.33	1.02	2.67
	$E \rightarrow E \oplus E$	6	2	1.34	1.10	3.50
	$S \rightarrow S\ ;\ S$	13	3	535.19	15.61	590.21
Cube(11)	$V \rightarrow \texttt{v0}$	2	2	0.01	0.01	0.12
	$V \rightarrow \texttt{v1}$	2	2	0.01	0.01	0.06
	$V \rightarrow \texttt{v10}$	3	3	0.02	0.01	0.05
	$V \rightarrow \texttt{v2}$	2	2	0.01	0.01	0.05
	$V \rightarrow \texttt{v3}$	2	2	0.01	0.01	0.05
	$V \rightarrow \texttt{v4}$	3	3	0.01	0.01	0.03
	$V \rightarrow \texttt{v5}$	3	3	0.01	0.01	0.04
	$V \rightarrow \texttt{v6}$	3	3	0.01	0.01	0.06
	$V \rightarrow \texttt{v7}$	3	4	0.02	0.01	0.06
	$V \rightarrow \texttt{v8}$	3	3	0.02	0.01	0.05
	$V \rightarrow \texttt{v9}$	3	4	0.01	0.01	0.05
	$B \rightarrow \texttt{var}\ V$	4	4	0.06	0.51	1.07
	$B \rightarrow B \wedge B$	116	8	1810.40	4.43	1831.92

[3] Note: The label (Ti) in the language name means the language timeouts under i runs.

DIFF(T4)	$E \to \texttt{0}$	1	1	0.01	0.01	0.11
	$E \to \texttt{1}$	1	1	0.01	0.01	0.06
	$E \to \texttt{2}$	1	1	0.02	0.01	0.06
	$E \to \texttt{x}$	2	2	0.35	0.01	0.41
	$E \to E \times E$	5	5	92.68	1.00	95.15
	$E \to E + E$	3	3	9.12	2.09	13.47
BVIMPSAT.(1, 2)(T7)	$E \to \texttt{0}$	1	1	0.01	0.02	0.37
	$E \to \texttt{1}$	1	1	0.01	0.01	0.14
	$E \to \texttt{o0}$	1	1	0.01	0.01	0.12
	$E \to \texttt{o1}$	1	1	0.01	0.01	0.09
	$E \to \texttt{v0}$	1	1	0.01	0.02	0.14
	$E \to E + E$	16	3	31.60	1.11	34.15
	$E \to E \mathbin{\&} E$	6	3	1.75	1.89	7.24
CNF(8)	$V \to \texttt{v0}$	2	2	0.01	0.01	0.12
	$V \to \texttt{v1}$	2	2	0.01	0.01	0.04
	$V \to \texttt{v2}$	2	2	0.01	0.01	0.05
	$V \to \texttt{v3}$	2	3	0.01	0.01	0.04
	$V \to \texttt{v4}$	2	2	0.01	0.01	0.03
	$V \to \texttt{v5}$	3	3	0.01	0.01	0.04
	$V \to \texttt{v6}$	3	3	0.01	0.01	0.04
	$V \to \texttt{v7}$	3	4	0.01	0.01	0.04
	$B \to \texttt{clause}\ C$	4	4	0.03	0.27	0.48
	$C \to \texttt{nvar}\ V$	5	5	0.05	0.32	0.70
	$C \to \texttt{var}\ V$	4	4	0.05	0.31	0.74
	$B \to C \wedge B$	39	6	30.52	0.56	31.83
	$C \to V \vee C$	41	8	37.03	0.69	38.62
DNF(8)	$V \to \texttt{v0}$	2	2	0.01	0.01	0.13
	$V \to \texttt{v1}$	2	2	0.01	0.01	0.04
	$V \to \texttt{v2}$	2	2	0.01	0.01	0.04
	$V \to \texttt{v3}$	2	3	0.01	0.01	0.04
	$V \to \texttt{v4}$	2	2	0.01	0.01	0.03
	$V \to \texttt{v5}$	3	3	0.01	0.01	0.04
	$V \to \texttt{v6}$	3	3	0.01	0.01	0.03
	$V \to \texttt{v7}$	3	4	0.01	0.01	0.05
	$B \to \texttt{conj}\ C$	4	4	0.05	0.30	0.57
	$C \to \texttt{nvar}\ V$	5	5	0.05	0.33	0.71
	$C \to \texttt{var}\ V$	4	4	0.05	0.32	0.76
	$C \to V \wedge C$	33	7	28.84	0.36	29.75
	$B \to C \vee B$	72	6	93.47	0.79	95.62
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
	$S \to \texttt{x} := E$	2	2	0.10	3.23	6.17
	$S \to \texttt{y} := E$	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

IntArith	$E \rightarrow \texttt{0}$	1	1	0.01	0.01	0.05
	$E \rightarrow \texttt{1}$	1	1	0.01	0.01	0.04
	$E \rightarrow \texttt{2}$	1	1	0.01	0.01	0.05
	$E \rightarrow \texttt{3}$	1	1	0.03	0.04	0.10
	$B \rightarrow \texttt{f}$	1	1	0.01	0.02	0.07
	$B \rightarrow \texttt{t}$	1	1	0.01	0.03	0.16
	$E \rightarrow \texttt{x}$	2	2	0.01	0.03	0.09
	$E \rightarrow \texttt{y}$	2	2	0.01	0.02	0.07
	$E \rightarrow \texttt{z}$	2	2	0.02	0.03	0.09
	$B \rightarrow \neg B$	4	4	0.06	5.09	15.22
	$E \rightarrow E \times E$	3	3	1.38	12.51	22.58
	$E \rightarrow E + E$	3	3	1.60	11.42	21.82
	$B \rightarrow E < E$	6	6	0.73	11.20	26.87
	$B \rightarrow B \wedge B$	5	5	0.08	8.08	14.95
	$B \rightarrow B \vee B$	4	4	0.05	7.70	14.19
	$E \rightarrow \texttt{ite}\ B\ E\ E$	4	4	0.78	13.54	31.00
IteExpr	$G \rightarrow \texttt{0}$	1	1	0.01	0.01	0.27
	$G \rightarrow \texttt{1}$	1	1	0.01	0.01	0.11
	$G \rightarrow \texttt{2}$	1	1	0.01	0.01	0.09
	$G \rightarrow \texttt{3}$	1	1	0.05	0.01	0.13
	$G \rightarrow \texttt{4}$	1	1	0.01	0.01	0.10
	$G \rightarrow \texttt{5}$	1	1	0.05	0.01	0.13
	$G \rightarrow \texttt{6}$	1	1	0.09	0.01	0.18
	$G \rightarrow \texttt{7}$	1	1	0.18	0.01	0.23
	$G \rightarrow \texttt{8}$	1	1	0.01	0.01	0.05
	$G \rightarrow \texttt{x}$	1	1	0.02	0.01	0.07
	$E \rightarrow \texttt{atom}\ F$	2	2	0.03	0.25	0.55
	$S \rightarrow \texttt{expr}\ E$	1	1	0.02	0.10	0.20
	$F \rightarrow \texttt{num}\ G$	1	1	0.03	0.30	0.69
	$F \rightarrow F \times G$	2	2	1.19	0.77	3.33
	$E \rightarrow E + F$	2	2	1.25	0.25	1.79
	$E \rightarrow E - F$	2	2	1.12	0.27	1.68
	$F \rightarrow F \div G$	4	3	1.92	0.94	3.88
	$B \rightarrow E = E$	5	4	0.09	0.24	0.71
	$B \rightarrow E \geq E$	5	5	1.79	0.33	2.60
	$B \rightarrow E > E$	5	5	0.18	0.26	0.79
	$B \rightarrow E \leq E$	6	6	0.24	0.56	1.48
	$B \rightarrow E < E$	5	5	0.12	0.30	0.85
	$B \rightarrow E \neq E$	6	6	5.35	0.26	6.11
	$S \rightarrow \texttt{ite}\ B\ E\ E$	3	3	0.29	0.29	0.92
RegEx(2)	$R \rightarrow \texttt{?}$	3	3	3.84	0.07	4.07
	$R \rightarrow \texttt{a}$	4	4	11.10	0.07	11.53
	$R \rightarrow \texttt{b}$	5	5	11.63	0.06	12.01
	$R \rightarrow \epsilon$	1	1	0.07	0.07	2.38
	$R \rightarrow \emptyset$	1	1	0.19	0.07	0.46
	$Start \rightarrow \texttt{eval}\ R$	3	3	0.02	4.43	13.40
	$R \rightarrow \texttt{!}R$	5	5	2.85	15.77	77.36
	$R \rightarrow R^{*}$	6	6	0.99	13.06	31.91
	$R \rightarrow R \cdot R$	24	24	333.71	72.58	495.45
	$R \rightarrow R \mid R$	10	10	10.96	59.54	140.82
BinOp	$B \rightarrow \texttt{0}$	1	1	0.01	0.01	0.07
	$B \rightarrow \texttt{1}$	1	1	0.01	0.01	0.22
	$B \rightarrow \texttt{x}$	2	2	0.01	0.01	0.08
	$N \rightarrow \texttt{atom}\ B$	2	2	0.09	0.04	0.30
	$M \rightarrow \texttt{atom}'\ B$	3	3	0.07	0.05	0.26
	$S \rightarrow \texttt{bin2dec}\ M$	2	2	0.02	0.09	0.30
	$S \rightarrow \texttt{count}\ N$	2	2	0.04	0.05	0.24
	$N \rightarrow \texttt{concat}\ N\ B$	5	5	8.61	0.22	10.31
	$M \rightarrow \texttt{concat}'\ M\ B$	5	5	288.81	0.23	308.50
Currency	$K \rightarrow \texttt{0}$	1	1	0.01	0.01	0.03
	$K \rightarrow \texttt{1}$	1	1	0.01	0.01	0.02
	$K \rightarrow \texttt{2}$	1	1	0.01	0.01	0.02
	$K \rightarrow \texttt{4}$	1	1	0.01	0.01	0.02
	$K \rightarrow \texttt{8}$	1	1	0.01	0.01	0.02
	$K \rightarrow \texttt{x}$	2	2	0.01	0.01	0.10
	$S \rightarrow \texttt{cny}\ K$	3	3	21.55	0.28	22.48
	$S \rightarrow \texttt{jpy}\ K$	1	1	0.01	0.10	0.22
	$S \rightarrow \texttt{usd}\ K$	2	2	19.65	0.21	20.27
	$S \rightarrow S \times K$	2	2	0.03	0.11	0.24
	$S \rightarrow S + S$	2	2	0.03	0.14	0.33
	$S \rightarrow S - S$	2	2	0.04	0.16	0.38
	$K \rightarrow K +_{\mathrm{K}} K$	3	3	0.28	0.35	1.19