The SyGuS Language Standard Version 2.1

Saswat Padhi Elizabeth Polgreen Mukund Raghothaman
Andrew Reynolds Abhishek Udupa

Monday 19th July, 2021

1 Introduction

We present a language to specify instances of the syntax-guided synthesis (SyGuS) problem. An instance of a SyGuS problem specifies:

1. The vocabularies, theories and the base types that are used to specify (a) semantic constraints on the function to be synthesized, and (b) the definition of the function to be synthesized itself. We refer to these as the *input* and *output* logics respectively.
2. A finite set of typed functions $f_1, \ldots, f_n$ that are to be synthesized.
3. The syntactic constraints on each function $f_i, i \in [1, n]$ to be synthesized. The syntactic constraints are specified using context-free grammars G_i which describe the syntactic structure of candidate solution for each of these functions. The grammar may only involve function symbols and sorts from the specified output logic.
4. The semantic constraints and assumptions that describe the behavior of the functions to be synthesized. Constraints are given as quantifier-free formulas φ and assumptions are given as quantifier-free formulas α in the input logic, and may refer to the functions-to-synthesize as well as universally quantified variables $v_1, \ldots, v_m$.

The objective then is to find definitions e_i (in the form of expression bodies) for each function f_i such that (a) the expression body belongs to the grammar G_i that is used to syntactically constrain solutions for f_i, and (b) the constraint $\forall v_1, \ldots, v_m.\, \alpha \implies \varphi$ is valid in the background theory under the assumption that functions $f_1, \ldots, f_n$ are interpreted as functions returning $e_1, \ldots, e_n$. Note that each e_i is an expression containing a fixed set of free variables which represent the argument list of the function f_i.

Overview of This Document This document defines the SyGuS format version 2.0, which is intended to be used as the standard input and output language for solvers targeting the syntax-guided synthesis problem. The language borrows many concepts and language constructs from the standard format for Satisfiability Modulo Theories (SMT) solvers, the SMT-LIB standard (version 2.6) [3].

Outline In the remainder of this section, we cover differences of the SyGuS format described in this document with previous revisions [5, 1, 2] and cover the necessary preliminaries. Then, Section 2 gives the concrete syntax for commands in the SyGuS input language. Section 3 documents the well-formedness and the semantics of input commands. Section 4 documents the expected output of synthesis solvers in response to these commands. Section 5 describes formally the notion of a *SyGuS logic* and how it restricts the set of commands that are allowed in an input. Section 6 formalizes what constitutes a correct response to a SyGuS input. Finally, Section 9 provides examples of possible inputs in the SyGuS language and solvers responses to them.

1.1 Differences from Previous Versions

In this section, we cover the differences in this format with respect to the one described in the previous version of the SyGuS format [5], and its extensions [1, 2].

1.1.1 Changes from SyGuS 1.0 to SyGuS 2.0

1. The syntax for providing grammars inside the `synth-fun` command now requires that non-terminal symbols are declared upfront in a *predeclaration*, see Section 3.4 for details.
2. The keyword `Start`, which denoted the starting (non-terminal) symbol of grammars in the previous standard, has been removed. Instead, the first symbol listed in the grammar is assumed to be the starting symbol.
3. Terms that occur as the right hand side of production rules in SyGuS grammars are now required to be binder-free. In particular, this means that let-terms are now disallowed within grammars. Accordingly, the keywords `InputVariable` and `LocalVariable`, which were used to specify input and local variables in grammars respectively, have been removed since the former is equivalent to `Variable` and the latter has no affect on the grammar.
4. The datatype keyword `Enum` and related syntactic features have been removed. The standard SMT-LIB 2.6 commands for declaring datatypes are now adopted, see Section 3.5 for details.
5. The `set-options` command has been renamed `set-option` to correlate to the existing SMT-LIB version 2.6 command.
6. The syntax for terms and sorts now coincides with the corresponding syntax for terms and sorts from SMT-LIB versions 2.0 and later. There are three notable changes with respect to the previous SyGuS format that come as a result of this change. First, negative integer and real constants must be written with unary negation, that is, the integer constant negative one must be written `(- 1)`, whereas previously it could be written `-1`. Second, the syntax for bit-vectors sorts `(BitVec n)` is now written `(_ BitVec n)`. Third, all let-bindings do *not* annotate the type of the variable being bound. Previously, a let-term was written `(let ((x T t)) ...)` where `T` indicates the type of `t`. Now, it must be written in the SMT-LIB compliant way `(let ((x t)) ...)`.
7. The signature, syntax and semantics for the theory of strings is now the one given in an initial proposal [7] to SMT-LIB. Thus, certain new symbols are now present in the signature, and some existing ones have changed names. The semantics of all operators, which was not specified previously, is now consistent with that provided in the proposal [7].
8. The command `declare-primed-var`, which was a syntactic sugar for two `declare-var` commands in the previous language standard, has been removed for the sake of simplicity. This command did not provide any benefit to invariant synthesis problems, since constraints declared via `inv-constraint` do not accept global variables.
9. The command `declare-fun`, which declared a universal variable of function type in the previous language standard, has been removed.
10. A formal notion of *input logics*, *output logics*, and *features* have been defined as part of the SyGuS background logic, and the command `set-feature` has been added to the language, which is used for further refining the constraints, grammars and commands that may appear in an input.
11. The expected output for fail responses from a solver has changed. In particular, the response `(fail)` from the previous language standard should now be provided as `fail`, that is, without parentheses. Additionally, a solver may answer `infeasible`, indicating that it has determined there are no solutions to the given conjecture.

1.1.2 Changes from SyGuS 2.0 to SyGuS 2.1

1. We introduce the command `assume`, which introduces a list of assumptions α, and the synthesis conjecture is now $\exists f_1, \ldots, f_n. \forall v_1, \ldots, v_m. \alpha \implies \varphi$.
2. Oracle constraints are introduced. Oracle constraints are constraints that are generated when an external oracle is called with a concrete set of input values. The constraints are conjoined to the list of constraints φ in the synthesis conjecture.

3. Oracle assumptions are introduced. Oracle assumptions are assumptions that are generated when an external oracle is called with a concrete set of input values. The assumptions are conjoined to the list of assumptions α in the synthesis conjecture.
4. Oracle Functions are introduced. These are functional symbols whose interpretation is associated to an external oracle. The functions are treated as universally quantified uninterpreted functions, and calls to that external oracle place with concrete input values generate assumptions over the behaviour of the uninterpreted function. These assumptions are conjoined to the list of assumptions α in the synthesis conjecture.
5. A set of new commands are introduced to enable easy specification of specific types of oracles, for instance input-output oracles. These oracles are based on the formalization by Jha and Seshia [4].
6. A new theory of tables is referenced as a possible underlying theory in Section 5.1.
7. A new SyGuS logic, called `CHC_`X, has been introduced for specifying synthesis problems using constrained Horn clause systems over an SMT-LIB base logic X.
8. A new command, `chc-constraint`, has been introduced for specifying constraints in the form of constrained Horn clauses.
9. The `synth-inv` command, used within invariant synthesis (`Inv_`X) logic earlier, has been removed. Instead, all functions (including predicates) to be synthesized are now specified using `synth-fun`.
10. The expected successful response for SyGuS solvers to a `check-synth` command is now enclosed in parentheses.
11. A new command `declare-weight`, has been introduced for introducing user-defined weight attributes.
12. A new feature `weights` has been added, which allows weight annotations on terms, as well as the use of weight symbols within terms, as described in Section 5.2.1.
13. A new command `optimize-synth` has been added for specifying optimization queries.
14. The signature, syntax and semantics for the theory of strings is now the one given in the official version 2.6 release of the theory of Unicode strings [3].

1.2 Preliminaries

In this document, we assume basic standard notions of multi-sorted first-order logic. We assume the reader is familiar with sorts, well-sorted terms, (quantified) formulas and free variables[1] If e is a term or formula, then we write $e[x]$ to denote that x occurs free in e, and $e[t]$ to denote the result of replacing all occurrences of x by t. We write $\lambda \vec{x}.\, t[\vec{x}]$ to denote a *lambda term*, that is, an anonymous function whose argument list is $\vec{x}$ that returns the value of $t[\vec{s}]$ for all inputs $\vec{x} = \vec{s}$. Given an application of a lambda term to a concrete argument list $\vec{s}$, i.e. the term $(\lambda \vec{x}.\, t[\vec{x}](\vec{s})$, then its *beta-reduction* is the term $t[\vec{s}]$. We use $\approx$ to denote the binary (infix) equality predicate.

2 Syntax

In this section, we describe the concrete syntax for the SyGuS 2.0 input language. Many constructs in this syntax coincide with those in SMT-LIB 2.6 standard [3]. In the following description, italic text within angle-brackets represents EBNF non-terminals, and text in typewriter font represents terminal symbols.

A SyGuS input $\langle SyGuS \rangle$ is thus a sequence of zero or more commands.

$$\langle SyGuS \rangle \quad ::= \quad \langle Cmd \rangle^*$$

We first introduce the necessary preliminary definitions, and then provide the syntax for commands $\langle Cmd \rangle$ at the end of this section.

[1] The definition of each of these coincides with the definition given in the SMT-LIB 2.6 standard [3].

2.1 Comments

Comments in SyGuS specifications are indicated by a semicolon ;. After encountering a semicolon, the rest of the line is ignored.

2.2 Literals

A *literal* ⟨*Literal*⟩ is a special sequence of characters, typically used to denote values or constant terms. The SyGuS format includes syntax for several kinds of literals, which are listed below. This treatment of most of these literals coincides with those in SMT-LIB version 2.6. For full details, see Section 3.1 of the SMT-LIB 2.6 standard [3].

$$\begin{array}{rcccccc} \langle Literal \rangle & ::= & \langle Numeral \rangle & | & \langle Decimal \rangle & | & \langle BoolConst \rangle & | \\ & & \langle HexConst \rangle & | & \langle BinConst \rangle & | & \langle StringConst \rangle & \end{array}$$

Numerals (⟨*Numeral*⟩) Numerals are either the digit 0, or a non-empty sequence of digits $[\texttt{0} - \texttt{9}]$ that does not begin with 0.

Decimals (⟨*Decimal*⟩) The syntax for decimal numbers is ⟨*Numeral*⟩ $.\texttt{0}^*$ ⟨*Numeral*⟩.

Booleans (⟨*BoolConst*⟩) Symbols `true` and `false` denote the Booleans true and false.

Hexidecimals (⟨*HexConst*⟩) Hexadecimals are written with #`x` followed by a non-empty sequence of (case-insensitive) digits and letters taken from the ranges $[\texttt{A} - \texttt{F}]$ and $[\texttt{0} - \texttt{9}]$.

Binaries (⟨*BinConst*⟩) Binaries are written with #`b` followed by a non-empty sequence of bits $[\texttt{0} - \texttt{1}]$.

Strings (⟨*StringConst*⟩) A string literal ⟨*StringConst*⟩ is any sequence of printable characters delimited by double quotes `" "`. The characters within these delimiters are interpreted as denoting characters of the string in a one-to-one correspondence, with one exception: two consecutive double quotes within a string denote a single double quotes character. In other words, `"a""b"` denotes the string whose characters in order are `a`, `"` and `b`. Strings such as `"\n"` whose characters are commonly interpreted as escape sequences are not handled specially, meaning this string is interpreted as the one consisting of two characters, `\` followed by `n`.
Literals are commonly used for denoting 0-ary symbols of a theory. For example, the theory of integer arithmetic uses numerals to denote non-negative integer values. The theory of bit-vectors uses both hexadecimal and binary constants in the above syntax to denote bit-vector values.

2.3 Symbols

Symbols are denoted with the non-terminal ⟨*Symbol*⟩. A symbol is any non-empty sequence of upper- and lower-case alphabets, digits, and certain special characters (listed below), with the restriction that it may not begin with a digit and is not a reserved word (see Appendix A for a full list of reserved words). A special character is any of the following:

```
_ + - * & | !  ~ < > = / % ?  .  $ ^
```

Note this definition coincides with simple symbols in Section 3.1 of SMT-LIB version 2.6, apart from differences in their reserved words.

Keywords (⟨*Keyword*⟩) Following SMT-LIB version 2.6, a keyword ⟨*Keyword*⟩ is a symbol whose first character is `:`.

2.4 Identifiers

An identifier $\langle Identifier \rangle$ is a syntactic extension of symbols that includes symbols that are indexed by integer constants or other symbols.

$$\begin{array}{rcl} \langle Identifier \rangle & ::= & \langle Symbol \rangle \mid \texttt{(\_} \ \langle Symbol \rangle \ \langle Index \rangle^{+} \texttt{)} \\ \langle Index \rangle & ::= & \langle Numeral \rangle \mid \langle Symbol \rangle \end{array}$$

Note this definition coincides with identifiers in Section 3.1 of SMT-LIB version 2.6.

2.5 Attributes

An attribute $\langle Attribute \rangle$ is a keyword and an (optional) associated value.

$$\langle Attribute \rangle \ ::= \ \langle Keyword \rangle \mid \langle Keyword \rangle \, \langle AttributeValue \rangle$$

The permitted values of attribute values $\langle AttributeValue \rangle$ depend on the attribute they are associated with. Possible values of attributes include symbols, as well as (lists of) sorts and terms. The above definition of attribute coincides with attributes in Section 3.4 of SMT-LIB version 2.6. All attributes standardized in this document are listed in Section 8.

2.5.1 Weight Attributes

Beyond the SMT-LIB standard, we standardize one class of attributes, namely that of *weights*. Some keywords we classify as *weight keywords*. In particular, we assume that `:weight` is the (builtin) weight keyword. Additional weight keywords can be user-defined via the command `declare-weight`.

A weight attribute consists of a weight keyword and an attribute value that is a integer numeral. For example, `:weight` 5 denotes the attribute associating the builtin weight keyword to 5. Terms can be annotated with weight attributes (see Section 2.7.1), which will have a special semantics which we will describe in detail in Section 5.2.1.

2.6 Sorts

We work in a multi-sorted logic where terms are associated with sorts $\langle Sort \rangle$. Sorts are constructed via the following syntax.

$$\langle Sort \rangle \ ::= \ \langle Identifier \rangle \mid \texttt{(} \ \langle Identifier \rangle \ \langle Sort \rangle^{+} \texttt{)}$$

The *arity* of the sort is the number of (sort) arguments it takes. A *parametric* sort is one whose arity is greater than zero. Theories associate identifiers with sorts and sort constructors that have an intended semantics. Sorts may be defined by theories (see examples in Section 5.1) or may be user-defined (see Section 3.5).

2.7 Terms

We use terms $\langle Term \rangle$ to specify grammars and constraints, which are constructed by the following syntax.

$$
\begin{array}{rcl}
\langle Term \rangle & ::= & \langle Identifier \rangle \\
& | & \langle Literal \rangle \\
& | & \texttt{(} \langle Identifier \rangle \ \langle Term \rangle^{+} \texttt{)} \\
& | & \texttt{(!} \ \langle Term \rangle \ \langle Attribute \rangle^{+} \texttt{)} \\
& | & \texttt{(exists (} \langle SortedVar \rangle^{+} \texttt{)} \ \langle Term \rangle \texttt{)} \\
& | & \texttt{(forall (} \langle SortedVar \rangle^{+} \texttt{)} \ \langle Term \rangle \texttt{)} \\
& | & \texttt{(let (} \langle VarBinding \rangle^{+} \texttt{)} \ \langle Term \rangle \texttt{)} \\
\\
\langle BfTerm \rangle & ::= & \langle Identifier \rangle \\
& | & \langle Literal \rangle \\
& | & \texttt{(} \langle Identifier \rangle \ \langle BfTerm \rangle^{+} \texttt{)} \\
& | & \texttt{(!} \ \langle BfTerm \rangle \ \langle Attribute \rangle^{+} \texttt{)} \\
\\
\langle SortedVar \rangle & ::= & \texttt{(} \langle Symbol \rangle \ \langle Sort \rangle \texttt{)} \\
\langle VarBinding \rangle & ::= & \texttt{(} \langle Symbol \rangle \ \langle Term \rangle \texttt{)}
\end{array}
$$

Above, we distinguish a subclass of *binder-free* terms $\langle BfTerm \rangle$ in the syntax above, which do not contain bound (local) variables. Like sorts, the identifiers that comprise terms can either be defined by the user or by background theories.

2.7.1 Term Annotations

In SMT-LIB, terms t may be annotated with *attributes*. The purpose of an attribute is to mark a term with a set of special properties, which may influence the expected result of later commands. Attributes are specified using the syntax `(!` t A_1 $\ldots$ A_n`)` where t is a term and $A_1, \ldots, A_n$ are attributes. An attribute can be any The term above is semantically equivalent to t itself. Several attributes are standardized by the SMT-LIB standard, while others may be user-defined.

2.8 Features

A feature $\langle Feature \rangle$ is a keyword denoting a restriction or extension on the kinds of SyGuS commands that are allowed in an input. It is an enumeration in the following syntax.

$$
\langle Feature \rangle \quad ::= \quad \texttt{:grammars} \mid \texttt{:fwd-decls} \mid \texttt{:recursion} \mid \texttt{:oracles} \mid \texttt{:weights}
$$

More details on features are given in Section 5.2.

2.9 Commands

A command $\langle Cmd \rangle$ is given by the following syntax.

$$
\begin{array}{rcl}
\langle Cmd \rangle & ::= & \texttt{(assume} \ \langle Term \rangle \texttt{)} \\
& | & \texttt{(check-synth)} \\
& | & \texttt{(chc-constraint (} \langle SortedVar \rangle^{*} \texttt{)} \ \langle Term \rangle \ \langle Term \rangle \texttt{)} \\
& | & \texttt{(constraint} \ \langle Term \rangle \texttt{)} \\
& | & \texttt{(declare-var} \ \langle Symbol \rangle \ \langle Sort \rangle \texttt{)} \\
& | & \texttt{(declare-weight} \ \langle Symbol \rangle \ \langle Attribute \rangle^{*} \texttt{)} \\
& | & \texttt{(inv-constraint} \ \langle Symbol \rangle \ \langle Symbol \rangle \ \langle Symbol \rangle \ \langle Symbol \rangle \texttt{)} \\
& | & \texttt{(optimize-synth (} \langle Term \rangle^{*} \texttt{)} \ \langle Attribute \rangle^{*} \texttt{)} \\
& | & \texttt{(set-feature} \ \langle Feature \rangle \ \langle BoolConst \rangle \texttt{)} \\
& | & \texttt{(synth-fun} \ \langle Symbol \rangle \ \texttt{(} \langle SortedVar \rangle^{*} \texttt{)} \ \langle Sort \rangle \ \langle GrammarDef \rangle^{?} \texttt{)} \\
& | & \langle OracleCmd \rangle \\
& | & \langle SmtCmd \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\langle OracleCmd \rangle & ::= & \texttt{(oracle-assume (}\langle SortedVar \rangle^*\texttt{) (}\langle SortedVar \rangle^*\texttt{)} \ \langle Term \rangle \ \langle Symbol \rangle \ \texttt{)} \\
& | & \texttt{(oracle-constraint (}\langle SortedVar \rangle^*\texttt{) (}\langle SortedVar \rangle^*\texttt{)} \ \langle Term \rangle \ \langle Symbol \rangle \ \texttt{)} \\
& | & \texttt{(declare-oracle-fun} \ \langle Symbol \rangle \ \texttt{(}\langle Sort \rangle^*\texttt{)} \ \langle Sort \rangle \ \langle Symbol \rangle \ \texttt{)} \\
& | & \texttt{(oracle-constraint-io} \ \langle Symbol \rangle \ \langle Symbol \rangle \ \texttt{)} \\
& | & \texttt{(oracle-constraint-cex} \ \langle Symbol \rangle \ \langle Symbol \rangle \ \texttt{)} \\
& | & \texttt{(oracle-constraint-membership} \ \langle Symbol \rangle \ \langle Symbol \rangle \ \texttt{)} \\
& | & \texttt{(oracle-constraint-poswitness} \ \langle Symbol \rangle \ \langle Symbol \rangle \ \texttt{)} \\
& | & \texttt{(oracle-constraint-negwitness} \ \langle Symbol \rangle \ \langle Symbol \rangle \ \texttt{)} \\
& | & \texttt{(declare-correctness-oracle} \ \langle Symbol \rangle \ \langle Symbol \rangle \ \texttt{)} \\
& | & \texttt{(declare-correctness-cex-oracle} \ \langle Symbol \rangle \ \langle Symbol \rangle \ \texttt{)} \\
\\
\langle SmtCmd \rangle & ::= & \texttt{(declare-datatype} \ \langle Symbol \rangle \ \langle DTDecl \rangle\texttt{)} \\
& | & \texttt{(declare-datatypes (}\langle SortDecl \rangle^{n+1}\texttt{) (}\langle DTDecl \rangle^{n+1}\texttt{))} \\
& | & \texttt{(declare-sort} \ \langle Symbol \rangle \ \langle Numeral \rangle\texttt{)} \\
& | & \texttt{(define-fun} \ \langle Symbol \rangle \ \texttt{(}\langle SortedVar \rangle^*\texttt{)} \ \langle Sort \rangle \ \langle Term \rangle\texttt{)} \\
& | & \texttt{(define-sort} \ \langle Symbol \rangle \ \langle Sort \rangle\texttt{)} \\
& | & \texttt{(set-info} \ \langle Keyword \rangle \ \langle Literal \rangle\texttt{)} \\
& | & \texttt{(set-logic} \ \langle Symbol \rangle\texttt{)} \\
& | & \texttt{(set-option} \ \langle Keyword \rangle \ \langle Literal \rangle\texttt{)} \\
\\
\langle SortDecl \rangle & ::= & \texttt{(}\langle Symbol \rangle \ \langle Numeral \rangle\texttt{)} \\
\langle DTDecl \rangle & ::= & \texttt{(}\langle DTConsDecl \rangle^+\texttt{)} \\
\langle DTConsDecl \rangle & ::= & \texttt{(}\langle Symbol \rangle \ \langle SortedVar \rangle^*\texttt{)} \\
\\
\langle GrammarDef \rangle & ::= & \texttt{(}\langle SortedVar \rangle^{n+1}\texttt{) (}\langle GroupedRuleList \rangle^{n+1}\texttt{)} \\
\langle GroupedRuleList \rangle & ::= & \texttt{(}\langle Symbol \rangle \ \langle Sort \rangle \ \texttt{(}\langle GTerm \rangle^+\texttt{))} \\
\\
\langle GTerm \rangle & ::= & \texttt{(Constant} \ \langle Sort \rangle\texttt{)} \mid \texttt{(Variable} \ \langle Sort \rangle\texttt{)} \mid \langle BfTerm \rangle
\end{array}
$$

For convenience, we distinguish between three kinds of commands above. The commands listed under $\langle Cmd \rangle$ and $\langle OracleCmd \rangle$ are specific to the SyGuS format, with the latter pertaining to oracles. The remaining commands listed under $\langle SmtCmd \rangle$ are borrowed from SMT-LIB 2.6. The semantics of these commands are detailed in Section 3.

3 Semantics of Commands

A SyGuS input file is a sequence of commands, which at a high level are used for defining a (single) synthesis conjecture, and invoking a solver for this conjecture. This conjecture is a closed formula of the form:

$$\exists f_1, \ldots, f_n. \forall v_1, \ldots, v_m. (\alpha_1 \wedge \ldots \wedge \alpha_r) \implies (\varphi_1 \wedge \ldots \wedge \varphi_q)$$

We wil use Ψ to refer to the formula $\forall v_1, \ldots, v_m. (\alpha_1 \wedge \ldots \wedge \alpha_r) \implies (\varphi_1 \wedge \ldots \wedge \varphi_q)$, and thus the synthesis conjecture can be written $\exists f_1, \ldots, f_n. \Psi$. In this section, we define how this conjecture is is established via SyGuS commands. Given a sequence of commands, the current state consists of the following information:

- A list $f_1, \ldots, f_n$, which we refer to as the current list of *functions to synthesize*,
- A list $v_1, \ldots, v_m$ of variables, which we refer to as the current list of *universal variables*,
- A list of formulas $\varphi = \varphi_1, \ldots, \varphi_q$, which we refer to as the current list of *constraints*,
- A list of formulas $\alpha = \alpha_1, \ldots, \alpha_r$, which we refer to as the current list of *assumptions*,
- A *signature* denoting the set of defined symbols in the current scope. A signature is a mapping from symbols to expressions (either sorts or terms). Each of these symbols may have a predefined semantics either given by the theory, or defined by the user (e.g. symbols that are defined as macros fit the latter category).

- A *SyGuS logic* denoting the terms and sorts that may appear in constraints and grammars.

In the initial state of a SyGuS input, the lists of functions-to-synthesize, universal variables, constraints, assumptions, and the signature are empty, and the SyGuS logic is the default one (see Section 5 for details).

In the following, we first describe restrictions on the order in which commands can be specified in SyGuS inputs. We then describe how each command $\langle Cmd \rangle$ updates the state of the sets above and the current signature.

3.1 Command Ordering

A SyGuS input is not well-formed if it specifies a list of commands that do not meet the restrictions given in this section regarding their order. The order is specified by the following regular pattern:

$$(\{\textit{set logic command}\})?\ (\{\textit{setter commands}\})^*\ (\{\textit{other commands}\})^*$$

where the set $\{\textit{set logic command}\}$ consists of the set of all `set-logic` commands, set $\{\textit{setter commands}\}$ includes the `set-feature` and `set-option` commands, the set $\{\textit{other commands}\}$ include all the SyGuS commands except the commands in the aforementioned two sets.

In other words, a SyGuS input is well formed if it begins with at most one `set-logic` command, followed by a block of zero or more `set-feature` and `set-option` commands in any order, followed by zero or more instances of the other SyGuS commands.

3.2 Setting the Logic

The logic of a SyGuS specification consists of three parts — an *input logic*, an *output logic* and a *feature set*. Roughly, the input logic determines what terms can appear in constraints, and the output logic determines what terms can appear in grammars and solutions. The feature set places additional restrictions or extensions on the constraints, grammars as well as the commands that are allowed in an input. These are described in detail in Section 5.

- `(set-logic` L`)`

  This sets the SyGuS background logic to the one that L refers to. The logic string L can be a standard one defined in the SMT-LIB 2.6 standard [3] or may be solver-specific. If L is an SMT-LIB standard logic, then it must contain quantifiers or this command is not well-formed, that is, logics with the prefix `QF_` are not allowed.[2] If this command is well-formed and L is an SMT-LIB standard logic, then this command sets the SyGuS logic to the one whose input and output logics are `QF_`L and whose feature set is the default one defined in this document (Section 5.2). In other words, when this command has L as an argument and L is an SMT-LIB standard logic, this indicates that that terms in the logic of L are allowed in constraints, grammars and solutions, but they are are restricted to be quantifier-free. As a consequence, the overall synthesis conjectures allowed by default when L is a standard SMT-LIB logic have at most two levels of quantifier alternation.

- `(set-feature :`F b`)`

  This enables the feature specified by F in the feature set component of the SyGuS background logic if b is `true`, or disables it if b is `false`. All features standardized in SyGuS 2.0 are given in Section 5.2.

3.3 Declaring Universal Variables

- `(declare-var` S σ`)`

  This command appends S to the current list of universal variables and adds the symbol S of sort σ to the current signature. This command should be rejected if S already exists in the current signature.

[2] By convention quantifiers are always included in the logic. This is because the overall synthesis conjecture specified by the state may involve universal quantification.

3.4 Declaring Functions-to-Synthesize

- `(synth-fun` S `(`$(x_1\ \sigma_1)\ldots(x_n\ \sigma_n)$`)` σ $G^?$`)`

  This command adds S to the current list of functions to synthesize, and adds the symbol S of sort $\sigma_1 \times \ldots \times \sigma_n \to \sigma$ to the current signature. This command should be rejected if S is already a symbol in the current signature. We describe restrictions and well-formedness requirements for this command in the following.

  If provided, the syntax for the grammar G consists of two parts: a *predeclaration* `(`$(y_1\ \tau_1)\ldots(y_n\ \tau_n)$`)`, followed by a *grouped rule listing* `(`$(y_1\ \tau_1\ (g_{11}\ldots g_{1m_1}))\ldots(y_n\ \tau_n\ (g_{n1}\ldots g_{nm_n}))$`)` where $y_1,\ldots,y_n$ are the *non-terminal* symbols of the grammar. Note that the same variable symbols $y_1,\ldots,y_n$ and their sorts $\tau_1,\ldots,\tau_n$ appear both in the predeclaration and as heads of each of the rules. If this is not the case, then this command is not well-formed. For all i,j, recall that grammar term g_{ij} is either a term, or a class of terms denoted by `(Constant` σ_c`)` and `(Variable` σ_v`)` denoting respectively the set of constants whose sort is σ_c, and the set of all variables from $x_1,\ldots,x_n$ whose sort is σ_v. If g_{ij} is an ordinary term, then its free variables may contain $y_1,\ldots,y_n$, as well as S itself. If the grammar contains S itself, then it is possible that the definition given for S in a solution is recursive, however, this feature is disallowed in the default logic (see Section 5).

  This command is not well-formed if τ_1 (the type of the start symbol) is not σ. It is also not well-formed if G generates a term t from y_i that does not have type τ_i for some i.

  If provided, the grammar G must also be one that is allowed by the output logic of the current SyGuS logic. For more details on the restrictions imposed on grammars by the logic, see Section 5. If G does not meet the restrictions of the background logic, it should be rejected.

  If no grammar is provided, then any term of the appropriate sort in the output logic may be given in the body of a solution for S.

  More details on grammars and the terms they generate, as well as what denotes a term that meets the syntactic restrictions of a function-to-synthesize is discussed in detail in Section 6.1.

3.5 Declaring Sorts

In certain logics, it is possible for the user to declare user-defined sorts. For example, `declare-datatypes` commands may be given when the theory of datatypes is enabled in the background logic, `declare-sort` commands may be given when uninterpreted sorts are enabled in the background logic.

- `(declare-datatype` S D`)`

  This is syntax sugar for `(declare-datatypes ((`S `0)) (`D`))`.

- `(declare-datatypes (`$(S_1\ a_1)\ldots(S_n\ a_n)$`) (`$D_1\ldots D_n$`))`

  This command adds symbols corresponding to the datatype definitions $D_1,\ldots,D_n$ for $S_1,\ldots,S_n$ to the current signature. For each $i=1,\ldots,n$, integer constant a_i denotes the arity of datatype S_i. The syntax of D_i is a *constructor listing* of the form

  $$((c_1\ (s_{11}\ \sigma_{11})\ \ldots\ (s_{1m_1}\ \sigma_{1m_1}))\ \ldots\ (c_k\ (s_{k1}\ \sigma_{k1})\ \ldots\ (s_{km_k}\ \sigma_{km_k})))$$

  For each i, the following symbols are added to the signature:

  1. Symbol S_i is added to the current signature, defined it as a datatype sort whose definition is given by D_i,
  2. Symbols $c_1,\ldots,c_k$ are added to the signature, where for each $j=1,\ldots,k$, symbol c_j is defined as a *constructor* of sort $\sigma_{j1}\times\ldots\times\sigma_{jm_j}\to D_i$,
  3. For each $j=1,\ldots,k$, $\ell=1,\ldots m_j$, symbol $s_{j\ell}$ is added to the signature, defined as a *selector* of sort $D_i\to\sigma_{j\ell}$.

  This command should be rejected if any of the above symbols this command adds to the signature are already a symbol in the current signature. We provide examples of datatype definitions in Section 9. For full details on well-formed datatype declarations, refer to Section 4.2.3 of the SMT-LIB 2.6 standard [3].

- `(declare-sort` S n`)`

  This command adds the symbol S to the current signature and associates it with an uninterpreted sort of arity n. This command should be rejected if S is already a symbol in the current signature.

3.6 Declaring Weight Keywords

- `(declare-weight` S $A_1 \ldots A_n$`)`

  This command declares the symbol : S as a weight keyword. The attributes $A_1 \ldots A_n$ can be used to indicate properties of the weight, such as its default value. For details, see Section 8.1.

  Terms can be subsequently annotated with this keyword, e.g. `(!` t `:` S `1)`, which are then given a special semantics as described in Section 5.2.1.

This command is only allowed when the `weights` feature is enabled.

3.7 Defining Macros

- `(define-fun` S `((`x_1 σ_1`) ... (`x_n σ_n`))` σ t`)`

  This adds to the current signature the symbol S of sort σ if $n = 0$ or $\sigma_1 \times \ldots \sigma_n \to \sigma$ if $n > 0$. The variables $x_1, \ldots, x_n$ may occur freely in t. It defines S as a term whose semantics are given by the function $\lambda x_1, \ldots, x_n . t$. Notice that t may not contain any free occurrences of S, that is, the definition above is not recursive. This command is not well-formed if t is not a well-sorted term of sort σ. This command should be rejected if S is already a symbol in the current signature.

- `(define-sort` S `(`$u_1 \ldots u_n$`)` σ`)`

  This adds the symbol S to the current signature. It defines S as the sort σ. The sort variables $u_1, \ldots, u_n$ may occur free in σ, while S may not occur free in σ. This command is not well-formed if σ is not a well-formed sort. This command should be rejected if S is already a symbol in the current signature.

3.8 Asserting Synthesis Constraints and Assumptions

- `(constraint` t`)`

  This adds t to the set of constraints. This command is well formed if t is a well-sorted formula, that is, a well-sorted term of sort Bool. Furthermore, the term t should be allowed based on the restrictions of the current logic, see Section 5 for more details.

- `(assume` t`)`

  This adds t to the set of assumptions. Like constraints, this command is well formed if t is a well-sorted term of sort Bool. and is allowed based on the restrictions of the current logic.

- `(inv-constraint` S S_{pre} S_{trans} S_{post}`)`

  This command adds a set of constraints to the current state that correspond to an invariant synthesis problem for function-to-synthesize S, where S_{pre} denotes a pre-condition, S_{post} denotes a post-condition and S_{trans} denotes a transition relation.

  A constraint of this form is well-formed if:

  1. S is the function-to-synthesize of sort $\sigma_1 \times \ldots \times \sigma_n \to$ Bool,
  2. S_{pre} is a defined symbol whose definition is of the form $\lambda x_1, \ldots, x_n . \varphi_{pre}$,
  3. S_{trans} is a defined symbol whose definition is of the form $\lambda x_1, \ldots, x_n, y_1, \ldots, y_n . \varphi_{trans}$, and
  4. S_{post} is a defined symbol whose definition is of the form $\lambda x_1, \ldots, x_n . \varphi_{post}$.

  where $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_n)$ are tuples of variables of sort $(\sigma_1, \ldots, \sigma_n)$ and φ_{pre}, φ_{trans} and φ_{post} are formulas.

  When this command is well-formed, given the above definitions, this command is syntax sugar for:

```
(declare-var v_1 σ_1)
(declare-var v'_1 σ_1)
...
(declare-var v_n σ_n)
(declare-var v'_n σ_n)
(constraint (=> (S_pre v_1 ... v_n) (S v_1 ... v_n)))
(constraint (=> (and (S v_1 ... v_n) (S_trans v_1 ... v_n v'_1 ... v'_n)) (S v'_1 ... v'_n)))
(constraint (=> (S v_1 ... v_n) (S_post v_1 ... v_n)))
```

where $v_1, v'_1, \ldots, v_n, v'_n$ are fresh symbols.

- `(chc-constraint ((`x_1 σ_1`) ... (`x_m σ_m`)) `t_{body} t_{head}`)`

  This command adds a universally quantified implication constraint to the current state in the form of a constrained Horn clause (CHC). Concretely, it adds m fresh variables $(v_1, \ldots, v_m)$ of sorts $(\sigma_1, \ldots, \sigma_m)$ respectively to the list of universal variables, and a constraint

  $$((\lambda x_1, \ldots, x_m. t_{body})\ v_1\ \ldots\ v_m) \implies ((\lambda x_1, \ldots, x_m. t_{head})\ v_1\ \ldots\ v_m)$$

  to the current state. This is a generalization of the `inv-constraint` command, that allows for specifying problems with multiple, possibly interdependent, invariant predicates to be synthesized.

  A constraint of this form is well-formed if the *body* of the CHC (i.e., t_{body}) and the *head* of the CHC (i.e., t_{head}) are both well-sorted terms of Bool sort.

  When this command is well-formed, given the above definitions, this command is syntax sugar for:

```
(declare-var v_1 σ_1)
...
(declare-var v_m σ_m)
(define-fun F_body ((x_1 σ_1) ... (x_m σ_m)) Bool t_body)
(define-fun F_head ((x_1 σ_1) ... (x_m σ_m)) Bool t_head)
(constraint (=> (F_body v_1 ... v_m) (F_head v_1 ... v_m)))
```

  where $v_1, \ldots, v_m$, F_{body}, and F_{head} are fresh symbols.

3.9 Asserting Oracle Constraints and Assumptions

- `(oracle-constraint ((`x_1 σ_1`) ... (`x_n σ_n`)) ((`y_1 τ_1`) ... (`y_m τ_m`)) `t N`)`

  This informs the solver of the existance of an external binary with name N which can be used as means of adding new constraints to the problem. This command is well-formed only if N *implements* a function of sort $\sigma_1 \times \ldots \times \sigma_n \to \tau_1 \times \ldots \times \tau_m$ (for a definition of the expected implementation of an external binary, see Section 7), and t is a well-sorted term of sort Bool whose free symbols may include those in the current signature, as well as any symbol in $x_1 \ldots x_n$ and $y_1 \ldots y_m$, and moreover is allowed based on the restrictions of the current logic.

  Assuming this command is well-formed, the interaction between the synthesis solver and the binary can be understood as the solver passing values $c_1, \ldots, c_n$ for $x_1, \ldots x_n$ as input to the binary, and the binary generating a list of values $d_1, \ldots, d_m$ corresponding to the output $y_1, \ldots y_m$. The expected implementation for passing constant values as input and output is described in Section 7. For each such call, the formula $t[c_1 \ldots c_n d_1 \ldots d_m]$, i.e., t with all occurences of $x_1, \ldots, x_n, y_1, \ldots, y_m$ replaced with $c_1, \ldots, c_n, d_1, \ldots d_m$, is added to the current list of constraints φ in the conjecture.

  Note that the synthesis solver may choose to call the binary N any time during solving, and as many times as it chooses. Note that it is possible that the synthesis solver may call the binary with the same input more than once.

- `(oracle-assume ((`x_1 σ_1`) ... (`x_n σ_n`)) ((`y_1 τ_1`) ... (`y_m τ_m`)) `t N`)`

  This command is identical to `oracle-constraint`, but the term $t[c_1 \ldots c_n d_1 \ldots d_m]$ obtained from a call to the external binary is added to the set of assumptions instead of the set of constraints.

3.10 Declaring Oracle Functional Symbols

- `(declare-oracle-fun` S `(`$\sigma_1 \ldots \sigma_n$`)` σ N`)`

  This adds to the current signature a symbol S of function sort $\sigma_1 \times \ldots \times \sigma_n \to \sigma$ whose interpretation is given by an external oracle also with name N. For every call to the oracle, an assumption about the behaviour of S is added to the list of assumptions α in the conjecture. Note that this command is syntactic sugar for:

  ```
  (declare-var S (-> σ1 ... σn σ))
  (oracle-assume ((x1 σ1) ... (xn σn)) ((x σ)) (= (S x1 ... xn) x) N)
  ```

  where $x_1, \ldots, x_n$ and x are fresh variables.

3.11 Pre-defined Oracle Types

Given a synthesis function symbol F in the current signature with sort $\sigma_1, \ldots \sigma_n \to \sigma$, each of the following commands is used to declare an external oracle with name N, which generate specific types of oracle constraints and assumptions over the behaviour of F.

- `(oracle-constraint-io` F N`)`

  This declares an input-output oracle for function F. It is syntactic sugar for:

  `(oracle-constraint ((`x_1 σ_1`) ... (`x_n σ_n`)) ((`x σ`))(=` `(`F $x_1 \ldots x_n$`)` x`)` N`)`

- `(oracle-constraint-poswitness` F N`)`

  This declares a positive witness oracle for synthesis function F and is syntactic sugar for:

  `(oracle-constraint ((`x_1 σ_1`) ... (`x_n σ_n`)(`x σ`)) (=` `(`F $x_1 \ldots x_n$`)` x`)` N`)`

- `(oracle-constraint-negwitness` F N`)`

  This declares a negative witness oracle for synthesis function F and is syntactic sugar for:

  `(oracle-constraint ((`x_1 σ_1`) ... (`x_n σ_n`)(`x σ`)) (not (=` `(`F $x_1 \ldots x_n$`)` x`))` N`)`

- `(oracle-constraint-membership` F N`)`

  This declares a membership-query oracle for synthesis function F and is syntactic sugar for:

  `(oracle-constraint ((`x_1 σ_1`) ... (`x_n σ_n`)(`x σ`)) ((`R Bool`)) (= (=` `(`F $x_1 \ldots x_n$`)` x`)` R`)` N`)`

- `(oracle-constraint-cex` F N`)`

  This declares a counter-example oracle for synthesis function F and is syntactic sugar for:

  `(oracle-constraint ((`F_c `(->` σ_1 ... σ_n σ`)))((`R Bool`)(`x_1 σ_1`) ... (`x_n σ_n`))`
  `(=>` R `(not(=` `(`F $x_1 \ldots x_n$`)` `(`F_c $x_1 \ldots x_n$`))))` N`)`

  where F_c is a candidate implementation for F, and R is a boolean that indicates that the oracle was able to find a counterexample.

- `(declare-correctness-oracle` F N`)`

  This declares a correctness oracle that determines whether a candidate implementation of a synthesis function F, where F is a synthesis function symbol in the current signature with sort $\sigma_1, \ldots \sigma_n \to \sigma$, is correct.

  This oracle is mandatory to call in order to determine correctness of the synthesis function and so it is syntactic sugar for:

  `(declare-oracle-fun` s `((->` $\sigma_1 \ldots \sigma_n$ σ`))` Bool N`)`
  `(constraint (`s F`))`

  where s is a fresh symbol.

- `(declare-correctness-cex-oracle` F N`)` This declares a correctness oracle that determines whether a candidate implementation of a synthesis function F, where F is a synthesis function symbol in the current signature with sort $\sigma_1, \ldots \sigma_n \to \sigma$, is correct and returns a counterexample if not.

  This oracle is mandatory to call to determine correctness of a synthesis function, and it adds a constraint to the oracle constraints, so it is syntactic sugar for the following commands:

  `(oracle-constraint ((`F_c `(->` σ_1 ... σ_n σ`)))((`R Bool`)(`x_1 σ_1`)` ... `(`x_n σ_n`))`
  `(=>`R `(not(=` $(F\ x_1 \ldots x_n)$ $(F_c\ x_1 \ldots x_n)$`))))` N`)`
  `(declare-oracle-fun` s `((->` $\sigma_1 \ldots \sigma_n$ σ`))` Bool N`)`
  `(constraint (`s F`))`

  where s is a fresh symbol.

3.12 Initiating Synthesis Solver

- `(check-synth)`

  This asks the synthesis solver to find a solution for the synthesis conjecture corresponding to the current list of functions-to-synthesize, universal variables and constraints.

- `(optimize-synth (`$s_1 \ldots s_j$`)` $A_1 \ldots A_k$`)`

  Like the command above, this asks the synthesis solver to find a solution for the synthesis conjecture corresponding to the current list of functions-to-synthesize, universal variables and constraints. Moreover, that solution should be optimal with respect to the objective specified by the list of terms $s_1, \ldots, s_j$ and attributes $A_1, \ldots, A_k$. Objectives can be specified both using annotations on terms $s_1, \ldots, s_j$ as well as the use of attributes $A_1, \ldots, A_k$ to specify orderings on tuples. Details on objectives in given in Section 8.2.

  To be well formed, $s_1, \ldots, s_j$ cannot contain any occurrence of universal variables in the current state, and must be permitted by the background logic.

The expected output from the synthesis solver for these commands is covered in Section 4.

3.13 Setting Benchmark Information

- `(set-info :`S L`)`

  This sets meta-information specified by the symbol S to the (literal) value L, whose syntax is given in Section 2.2. This has no impact on the state, and is used to annotate the benchmark with relevant information. For the purposes of this document, we define the meaning of the concrete symbol, `sygus-version`, which when passed to this command is used to indicate the version of the SyGuS format used in the benchmark. A benchmark whose header contains the line

  `(set-info :sygus-version 2.1)`

  will use the version 2.1 syntax as specified by this document, whereas e.g. a value of 1.0 indicates that the benchmark will use the syntax in the previous SyGuS format [5, 1, 2].

3.14 Setting Solver Options

- `(set-option :`S L`)`

  This sets the solver-specific option specified by the symbol S to the (literal) value L, whose syntax is given in Section 2.2. We do not give concrete examples of such options in this document. It is recommended that synthesis solvers ignore unrecognized options, and choose reasonable defaults when the options are left unspecified.

4 Synthesis Solver Output

This section covers the expected output from a synthesis solver, which currently is limited to responses to `check-synth` only.

- A response to `check-synth` is one of the following:

  1. A list of commands, enclosed in parentheses, of the form:

     $$\begin{array}{l}(\\ (\langle FunDefCmdKwd\rangle\ \ f_1\ X_1\ \sigma_1\ t_1)\\ \ldots\\ (\langle FunDefCmdKwd\rangle\ \ f_n\ X_n\ \sigma_n\ t_n)\\ )\end{array}$$

     where functions $f_1, \ldots, f_n$ are the functions-to-synthesize in the current state, $X_1, \ldots, X_n$ are sorted variable lists, $\sigma_1, \ldots, \sigma_n$ are types, and $t_1, \ldots, t_n$ are terms. The syntax $\langle FunDefCmdKwd\rangle$ can be either `define-fun` or `define-fun-rec`. The latter must be used for f_i if it occurs free in t_i, that is, when the definition of f_i is recursive. It is required that $f_1, \ldots, f_n$ be provided in the order in which they were declared.[3] To be a well-formed response, for each $j = 1, \ldots, n$, it must be the case that t_j is a term of σ_j, and X_j is identical to the sorted variable list used when introducing the function-to-synthesize f_j.
  2. The output `infeasible`, indicating that the conjecture has no solutions.
  3. The output `fail`, indicating that the solver failed to find a solution to the conjecture.

  A response to `check-synth` of the first kind is a correct solution if it satisfies both the semantic and syntactic restrictions given by the current state. We describe this in more detail in Section 6.

  We do not define the correctness of an `infeasible` response in this document. We remark that a response of this form should be given by a solver only if it is certain that a solution to the current conjecture does not exist. A conjecture may be infeasible based on the current semantic restrictions, or may be infeasible due to a combination of semantic restrictions and the syntactic ones imposed on functions-to-synthesize. The response `fail` indicates that the solver was unable to find a solution, which does not necessarily imply that the conjecture is infeasible.

- A (successful) well-formed response to `(optimize-synth (`$s_1 \ldots s_j$`)` $A_1 \ldots A_k$`)` is a tuple of terms followed list of commands, enclosed in parentheses, of the form:

  $$\begin{array}{l}(\\ (c_1 \ldots c_j)\\ (\langle FunDefCmdKwd\rangle\ \ f_1\ X_1\ \sigma_1\ t_1)\\ \ldots\\ (\langle FunDefCmdKwd\rangle\ \ f_n\ X_n\ \sigma_n\ t_n)\\ )\end{array}$$

  To be a well-formed response, $c_1, \ldots, c_j$ must be values of the same type as $s_1, \ldots, s_j$, and the remaining commands must be a well-formed response based on the criteria for `check-synth` above. A response of the above form is correct if the specified functions satisfy the syntactic restrictions given by the current state. Additionally, the solution must satisfy the semantic specification for optimization queries, which implies that both the semantic specification is satisfied and that the values $c_1, \ldots, c_j$ are consistent valuations of terms $s_1, \ldots, s_j$ under the current solution for functions to synthesize $f_1, \ldots, f_n$. For details, see Section 6.

  Additionally, analogous to `check-synth`, a solver may respond with `infeasible` or `fail` to indicate the conjecture has no solutions, or to indicate a failure.

  The role of $A_1 \ldots A_k$ is to provide an ordering on solutions generated in response to this command, where a solver is expected to generate a solution that is optimal with respect to this ordering. For details, see Section 8.2. These attributes do not impact what consistutes a correct solution.

5 SyGuS Background Logics

In this section, we describe how the background logic restricts the constraints and grammars that are allowed as inputs. A SyGuS background logic consists of three parts:

[3] This is to ensure that the definition of f_i, which may depend on a definition of f_j for $j < i$, is given in the correct order.

1. An *input logic*, which can be set as part of a `set-logic` command. This corresponds to a SMT-LIB standard logic or may be solver specific. The input logic determines the set of terms that are allowed in constraints.
2. An *output logic*, which can be set as part of a `set-logic` command. Like input logics, this can correspond to an SMT-LIB standard logic or may be solver specific. The output logic determines the set of terms that are allowed in grammars and solutions.
3. A *feature set*, which restricts or extends the set of commands that are allowed in a SyGuS input, and may further refine the constraints, grammars and solutions allowed by the logic. Generally, feature sets are not expressible in an input or output logic and are independent of them.

We refer to input and output logics as *base* logics. Base logics have the same scope as SMT-LIB logics, in that their purpose is to define a set of terms and formulas. Further extensions and restrictions of the SyGuS language are recommended to be expressed as new base logics whenever it is possible to do so. On the other hand, the feature set is allowed to restrict or extend the *commands* that are allowed in the input or specific relationships between how terms appear in commands, which is not expressible in a base logic.

The default SyGuS logic is the one whose input and output logics include only the core theory of Booleans, and whose feature set enables grammars and the core commands of the language. We describe logics and feature sets in more detail in Sections 5.1 and 5.2. A formal definition of how these restrict the set of terms that may appear in constraints and grammars is then given in Sections 5.3 and 5.4.

5.1 Input and Output Logics

SMT-LIB provides a catalog of standard logics, available at `www.smt-lib.org`. SyGuS logics may use any of these logics either as input logics or as output logics. For many applications, the input and output logic is expected to be the same, although no relationship between the two is required.

At a high level, a *logic* includes a set of theories and defines a subset of terms constructible in the signature of those theories that belong to it. If a logic includes a theory, then its symbols are added to the current signature when a `set-logic` command is issued. Further details on the formal definition of theory and logic declarations can be found in Sections 3.7 and 3.8 of the SMT-LIB 2.6 standard [3]. We briefly review some of the important logics and theories in the following. More concrete details on standard SMT-LIB logics can be found in the reference grammars in Appendix B.

The input and output logic components of the default SyGuS logic include only the *core theory*. The signature of the core theory has the Boolean sort `Bool`, the Boolean constants `true` and `false`, the usual logical Boolean connectives `not`, `and`, `or`, implication `=>`, `xor`, and the parametric symbols = and `ite` denoting equality and if-then-else terms for all sorts in the signature. Logics that include the theories mentioned in this section supplement the signature of the current state with additional sorts and symbols of that theory.

Arithmetic The theory of integers is enabled in logics like linear integer arithmetic `LIA` or non-linear integer arithmetic `NIA`. The signature of this theory includes the integer sort `Int` and typical function symbols of arithmetic, including addition + and multiplication ∗. Unary negation and subtraction are specified by −. Constants of the theory are integer constants. Positive integers and zero are specified by the syntax ⟨*Numeral*⟩ from Section 2.2, whereas negative integers are specified as the unary negation of positive integer, that is, (− `2`) denotes negative two. Analogously, the theory of reals is enabled in logics like linear real arithmetic `LRA` or non-linear real arithmetic `NRA`. Its signature includes the real sort `Real`. Some of the function symbols of arithmetic are syntactically identical to those from the theory of integers, including +, − and ∗. The signature of the theory of reals additionally includes real division /. Positive reals and zero in this theory can either be specified as decimals using the syntax ⟨*Decimal*⟩ or as rationals of the form (/ `m n`), where `m` and `n` are numerals. Negative reals are specified as the unary negation of a decimal or as a negative rational (/ (− `m`) `n`).

Bit-Vectors The theory of fixed-width bit-vectors is included in logics specified by symbols that include the substring `BV`. The signature of this theory includes a family of indexed sorts (_ `BitVec n`) denoting bit-vectors of width n. The functions in this signature include various operations on bit-vectors, including bit-wise, arithmetic, and shifting operations.

Strings The theory of (unbounded) Unicode strings and regular expressions is included in some logics specified with `S` as a prefix, such as `S` (strings) or `SLIA` (strings with linear integer arithmetic). The string of this theory includes the string sort `String`, interpreted as the set of all unicode strings. Functions in this signature include string concatenation `str.++`, string length `str.len` as well as many extended functions such as string containment `str.contains`, string search `str.indexof` which returns the index of a string in another, and so on. A full description of this theory is given in a proposal to SMT-LIB [7].

Arrays The theory of arrays is included in logics specified with `A` as a prefix, such as `ABV` or `ALIA`. The signature of this theory includes a parametric sort `Array` of arity two, whose sort parameters indicate the index type and the element type of the array. It has two function symbols, `select` and `store`, interpreted as array select and array store.

Datatypes The theory of datatypes is included in logics specified by symbols that include the substring `DT`. Logics that include datatypes are such that `declare-datatypes` commands are permitted in inputs, whereas all others do not. The signature of the theory of datatypes is largely determined by the concrete datatypes definitions provided by the user. As mentioned in Section 3.5, these commands append datatype sorts, constructors and selectors to the current signature. Constructor symbols are used for constructing values (e.g. `cons` constructs a list from an element and another list), and selectors access subfields (e.g. `tail` returns its second argument). Notice that the value of *wrongly applied* selectors, e.g. `tail` applied to the `nil` list, is underspecified and hence freely interpreted in models of this theory. The only fixed symbol in the theory of datatypes is the unary indexed *discriminator* predicate (`_ is C`), which holds if and only if its argument is an application of constructor C. For example, assuming the standard definition of a list datatype with constructors `cons` and `nil`, we have that ((`_ is nil`) `x`) holds if and only if x is the `nil` list.

Uninterpreted Functions In SMT-LIB, uninterpreted functions and sorts may be declared in logics that include the substring `UF`, whose interpretations are not fixed. Declarations for functions and sorts are made via SMT-LIB commands `declare-fun` and `declare-sort` respectively. In the SyGuS language, we do not permit the declaration of functions with `declare-fun` command. Instead, the language includes only the latter command. Thus, the only effect that specifying `UF` in the logic string has is that user-defined sorts may be declared via `declare-sort`, where variables and functions-to-synthesize may involve these sorts in the usual way. We remark here that encoding synthesis problems that involve (existentially quantified) uninterpreted functions can be represented by declaring those functions using `synth-fun` commands where no grammar is provided. Synthesis problems that involve universally quantified variables of function sort are planned to be addressed in a future revision of this document that includes concrete syntax for function sorts.

5.1.1 Other Theories

Many other theories are possible beyond those supported in the SMT-LIB standard. In this section, we mention theories that are of interest to synthesis applications that are not included in the SMT-LIB standard.

Bags and Tables An SMT-LIB compliant theory of tables is proposed in [6]. We give the salient details of this theory here.

The theory of tables is implemented as an extension of a theory of bags (multisets). The signature of this theory includes all sorts of the form `(Bag T)` for all sorts `T`. This sort denotes bags (i.e. multisets) of elements of sort `T`. The theory of bags includes a multitude of operators for e.g. taking unions, intersections and differences of bags, as well as higher-order operators like `map`, `fold` and `partition`. A *table* is a bag whose element sort T is a tuple, where a tuple is a parametric sort taking n types corresponding to the types of the elements that comprise the tuple. For example, `(Tuple Int Int)` denotes tuples of arity two. The sort `(Table Int Int)` is syntax sugar for `(Bag (Tuple Int Int))` and is used to denote tables with two integer columns. A table value is a multiset union of tuple values. The number of rows in a table value is equal to its cardinality. Notice that ordering of rows is not semantically captured, and thus not modelled in this theory. For further details on the complete signature and semantics of this theory is available in [6].

5.2 Features and Feature Sets

A feature set is a set of values, called *features*, which for the purposes of this document can be seen as an enumeration type. Their syntax is given in Section 2.8. The meaning of all features standardized by this document are listed below.

- `grammars`: if enabled, then grammars may be provided for functions-to-synthesize in `synth-fun` commands.
- `fwd-decls`: if enabled, grammars of `synth-fun` may refer to previously declared synthesis functions, called *forward declarations*.
- `recursion`: if enabled, grammars of `synth-fun` can generate terms that correspond to recursive definitions.
- `oracles`: if enabled, commands from $\langle OracleCmd \rangle$ are permitted.
- `weights`: if enabled, terms of the form `(_ w f)` are permitted, as described below.

Formal definition of these features are given within the Sections 5.3 and 5.4. Other features and their meanings may be solver specific, which are not covered here.

The feature set component of the default SyGuS logic is the set {`grammars`}. In other words, grammars may be provided, but those involving forward declarations and recursion are not permitted by default, nor are oracles or weight constraints.

5.2.1 Weights

For each function-to-synthesize f and weight keyword W as mentioned in Section 2.5.1, the *weight symbol for f with respect to W* is a (nullary) integer symbol $\omega_{f,W}$ whose concrete syntax is `(_ w f)` where w is the suffix of W after its first character (:). For example, `(_ weight f)` denotes a symbolic integer constant corresponding to the weight of f with respect to the builtin weight keyword. We will use annotations on terms that appear in grammars to denote weighted production rules. The class of nullary integer symbols of the above form are interpreted as the sum of weights of production rules used for generating the body of f.

Formally, let G be a grammar provided for function-to-synthesize f, and let W be a weight keyword. Let y be a non-terminal symbol of G and let t be a term appearing in the grouped rule listing for y, as described in the syntax for grammar definitions in Section 3.4. We consider the *base term* of t, defined such that the base term of unannotated term is itself, and the base term of t is t_0 if t is of the form `(!` t_0 A_1 $\ldots$ A_n`)`. Notice that nested annotations on t_0 are not considered in this defintion. Assume that the base term of t is t_0 and that t has been annotated with the (possibly empty) list of attributes $A_1, \ldots, A_n$. In this case, we say that G contains the production rule $y \mapsto t_0$ whose *weight with respect to W* is:

$$\begin{cases} k & \text{if exactly one } A_i \text{ is of the form } \texttt{W}\ k \text{ for some numeral } k \\ k_{def} & \text{otherwise, where } k_{def} \text{ is the default weight value for } \texttt{W} \end{cases}$$

The default weight value for all weight keywords is 0. This value can be overridden via an attribute as described in Section 8.1.

Let $y \mapsto_{W,k} t_0$ denote a production rule whose weight with respect to W is k. We write $G \mapsto^*_{W,k} r$ if it is possible to construct a sequence of terms $s_1, \ldots, s_n$ with s_1 is the starting symbol of G and $s_n = r$ where for each $1 \leq i \leq n$, term s_i is obtained from s_{i-1} by replacing an occurrence of some y_i by t_i where $y_i \mapsto_{W,k_i} t_i$ is a rule in G, and $k_1 + \ldots k_n = k$. When the body of function-to-synthesize f is interpreted as term r, the symbol $\omega_{f,W}$ can be interpreted as k if and only if $G \mapsto^*_{W,k} r$.

Notice that there may be *multiple* sequences of terms that generate the same term r above. Thus, weight symbols may have multiple valid interpretations for a given interpretation for the body of a function-to-synthesize. The solver is free to pick any such interpretation.

5.3 SyGuS Logic Restrictions on Constraints

Let $\mathcal{L}$ be a SyGuS logic whose input logic is one from the SMT-LIB standard, or an externally defined logic. A term t is not allowed by $\mathcal{L}$ to be an argument to a `constraint` command if it is not allowed by the *input* logic of $\mathcal{L}$, according to the definition of that logic.

5.4 SyGuS Logic Restrictions on Grammars

Let $\mathcal{L}$ be a SyGuS logic whose output logic is one from the SMT-LIB standard. A grammar G is not allowed by $\mathcal{L}$ if it generates some term t with no free occurrences of non-terminal symbols that is not allowed by the *output* logic of $\mathcal{L}$, according to the definition of that logic.

Notice that it may be the case that a grammar G contains a rule whose conclusion is a term that does not itself meet the restrictions of the output logic. For example, consider the logic of *linear* integer arithmetic and a grammar G containing a non-terminal symbol y_c of integer type such that G generates only constants from y_c. Grammar G may be allowed in the logic of linear integer arithmetic even if it has a rule whose conclusion is $y_c * t$, noting that no non-linear terms can be generated from this rule, provided that no non-linear terms can be generated from t. An example demonstrating this case is given in Section 9.

The feature set component of the SyGuS logic imposes additional restrictions on the terms that are generated by grammars. Note the following definition. The *expanded form* of a term t is the (unique) term obtained by replacing all functions f in t that are defined as macros with their corresponding definition until a fixed point is reached. A grammar G for function-to-synthesize f is not allowed by a SyGuS logic $\mathcal{L}$ if G contains a rule whose conclusion is a term t whose expanded form contains applications of functions-to-synthesize unless the feature `fwd-decls` is enabled in the feature set of $\mathcal{L}$; it is not allowed if t contains f itself unless the feature `recursion` is enabled in the feature set of $\mathcal{L}$; it is not allowed regardless of the terms it generates unless `grammars` is enabled in the feature set of $\mathcal{L}$.

5.5 Additional SyGuS Logics

Here, we cover additional SyGuS logics that are standardized by this document that are *not* defined by the SMT-LIB standard.

Programming-by-examples (PBE) Given an SMT-LIB standard logic X that does not contain the prefix `QF_`, the base logic `PBE_`X denotes the logic where constraints are limited to (conjunctions of) equalities whose left hand side is a term $f(\vec{c})$ and whose right hand side is d, where f is a function and $\vec{c}$, d are constants. We refer to an equality of this form as a *PBE equality*. Such equalities denote a relationship between the inputs and output of function f for a single point. Notice that formulas allowed by logic `PBE_`X are a subset of those allowed by `QF_`X.

We use the logic string `PBE_`X to refer to a SyGuS logic as well. Given an SMT-LIB standard logic X that does not contain the prefix `QF_`, the SyGuS logic `PBE_`X is the one whose input logic is `PBE_`X, whose output logic `QF_`X, and whose feature set is the default one. In other words, the constraints allowed by this logic are limited to conjunctions of PBE equalities, but gives no special restrictions on the solutions or grammars that can be provided.

By construction of the overall synthesis conjecture, a SyGuS command sequence meets the requirements of the input logic `PBE_`X if and only if each `constraint` command takes as argument a PBE equality.

Single Invariant-to-Synthesize (Inv) Given an SMT-LIB standard logic X, the base logic `Inv_`X denotes the logic where formulas are limited to the invariant synthesis problem for a single invariant-to-synthesize. Concretely, this means that formulas are limited to those that are (syntactically) a conjunction of three implications, for a single predicate symbol I to be synthesized:

1. The first is an implication whose antecedent (the pre-condition) is an arbitrary formula in the logic X and and whose conclusion is an application $I(\vec{x})$ where $\vec{x}$ is a tuple of unique variables,
2. The second is an implication whose antecedent is $I(\vec{x}) \wedge \varphi$ where $\vec{x}$ is a tuple of unique variables and φ (the transition relation) is an arbitrary formula in the logic X, and whose conclusion is $I(\vec{y})$ where $\vec{y}$ is a tuple of unique variables disjoint from $\vec{x}$,
3. The third is an implication whose antecedent is an application $I(\vec{x})$ where $\vec{x}$ is a tuple of unique variables, and whose conclusion (the post-condition) is an arbitrary formula in the logic X.

The variables $\vec{x}$ in each of these three formulas are not required to be the same.

Like the previous section, we use the logic string `Inv_`X to refer to SyGuS logics as well. Given an SMT-LIB standard logic X that does not contain the prefix `QF_`, logic `Inv_`X denotes the SyGuS logic whose input logic is `Inv_QF_`X, whose output logic is `QF_`X, and whose feature set is the default one.

Note that if a SyGuS command sequence is such that it contains (1) exactly one `synth-fun` command with `Bool` return type, (2) exactly one `inv-constraint` command whose arguments are predicates with definition within logic X, and (3) no other commands that introduce constraints, then the command sequence is guaranteed to meet the requirements of the input logic `Inv_`X.

Constrained Horn Clause Systems (CHC) Given an SMT-LIB standard logic X, the base logic `CHC_`X denotes the logic where formulas are restricted to a valid system of constrained Horn clauses (CHCs), i.e., the formulas (syntactically) are a conjunction of CHCs over a, possibly empty, set of predicate symbols to be synthesized. To define a valid system of CHCs, we first define the notion of an *atomic term*.

Definition 1 (Atomic Term)**.** Given set $\mathbf{S}$ of symbols, a term t is said to be an $\mathbf{S}$-atomic term if either: (a) t does not contain any symbol that belongs to $\mathbf{S}$, or (b) t is of the form $(S\ u_1\ \ldots\ u_k)$ for some predicate symbol $S \in \mathbf{S}$ and $u_1, \ldots, u_k$ are variables. □

Given this definition, a set $\mathbf{C}$ of CHC constraints over a set $\mathbf{S}$ of symbols to be synthesized is considered a valid system of CHCs for logic `CHC_`X if $\mathbf{C}$ satisfies the following:

1. Each constraint $C \in \mathbf{C}$ satisfies the following:
   (a) C must be an implication of the form $B_C(\vec{x}) \implies H_C(\vec{x})$, where B_C (called the *body* of C) and H_C (called the *head* of C) are defined predicates, and $\vec{x}$ is a tuple of unique variables.
   (b) The definition of H_C must be an $\mathbf{S}$-atomic term.
   (c) The definition of B_C must either be an $\mathbf{S}$-atomic term, or a conjunction of $\mathbf{S}$-atomic terms.

   Henceforth, we will call such a constraint an $\mathbf{S}$-valid CHC, or simply a valid CHC when the set $\mathbf{S}$ is obvious from the context.
2. Exactly one *query*, i.e., a constraint $C \in \mathbf{C}$ with H_C defined as `false`.

The variables $\vec{x}$ in each of the constraints are not required to be the same.

As in the previous section, we use the logic string `CHC_`X to refer to SyGuS logics as well. Given an SMT-LIB standard logic X that does not contain the prefix `QF_`, logic `CHC_`X denotes the SyGuS logic whose input logic is `CHC_QF_`X, whose output logic is `QF_`X, and whose feature set is the default one.

A well-sorted SyGuS command sequence meets the requirements of the input logic `CHC_`X if it contains

1. `synth-fun` commands with `Bool` return type only, that declare a set $\mathbf{S}$ of predicates to be synthesized,
2. one or more `chc-constraint` command each of which is such that
   (a) its t_{head} is an $\mathbf{S}$-atomic predicate with definition within logic X,
   (b) its t_{body} is either an $\mathbf{S}$-atomic predicate, or a conjunction of $\mathbf{S}$-atomic predicates, each with definitions within logic X,
3. exactly one `chc-constraint` command with `false` as t_{head}, and
4. no other commands that introduce constraints.

6 Formal Semantics

Here we give the formal semantics for what constitutes a correct solution for a synthesis conjecture.

6.1 Satisfying Syntactic Specifications

In this section, we formalize the notion of satisfying the *syntactic specification* of the synthesis conjecture.

As described in Section 3.4, a grammar G is specified as a predeclaration and a *grouped rule listing* of the form

$$((y_1\ \tau_1\ (g_{11} \ldots g_{1m_1})) \ldots (y_n\ \tau_n\ (g_{n1} \ldots g_{nm_n})))$$

where $y_1, \ldots, y_n$ are variables and $g_{11} \ldots g_{1m_1}, \ldots, g_{n1} \ldots g_{nm_n}$ are grammar terms. We associate each grammar with a sorted variable list X, namely the argument list of the function-to-synthesize. We refer to y_1 as the *start symbol* of G.

We interpret G as a (possibly infinite) set of rules of the form $y \mapsto t$ where t is an (ordinary) term based on the following definition. For each y_i, g_{ik} in the grouped rule list, if g_{ik} is `(Constant` σ_c`)`, then G contains the rule $y_i \mapsto c$ for all constants of sort σ_c. If g_{ik} is `(Variable` σ_v`)`, then G contains the rule $y_i \mapsto x$ for all variables $x \in X$ of sort σ_v. Otherwise, if g_{ik} is an ordinary term, then G contains the rule $y_i \mapsto g_{ik}$.

We say that G *generates* term r from s if it is possible to construct a sequence of terms $s_1, \ldots, s_n$ with $s_1 = s$ and $s_n = r$ where for each $1 \leq i \leq n$, term s_i is obtained from s_{i-1} by replacing an occurrence of some y by t where $y \mapsto t$ is a rule in G.

Let f be a function to synthesize. A term $\lambda X.\, t$ satisfies the syntactic specification for f if one of the following hold:

1. A grammar G is provided for f, t contains no free occurrences of non-terminal symbols, and G generates t starting from y_1, where y_1 is the start symbol of G.
2. No grammar is provided for f, and t is any term allowed in the output logic whose sort is the same as the return sort of f.

Furthermore, A tuple of functions $(\lambda X_1.\, t_1, \ldots, \lambda X_n.\, t_n)$ satisfies the syntactic restrictions for functions-to-synthesize $(f_1, \ldots, f_n)$ in conjecture $\exists f_1, \ldots, f_n.\, \Psi$ if $\lambda X_i.\, t_i$ satisfies the syntactic specification for f_i for each $i = 1, \ldots, n$.

6.2 Satisfying Semantic Specifications

In this section, we formalize the notion of satisfying the *semantic specification* of the synthesis conjecture for background theories T from the SMT-LIB standard. The notion of satisfying semantic restrictions for background theories that are not standardized in the SMT-LIB standard are not covered by this document.

Before stating the formal definition for satisfying semantic specifications, we require the following definitions. Consider a synthesis conjecture $\exists f_1, \ldots, f_n.\, \Psi$ in background theory T, and let $\alpha = (\lambda X_1.\, t_1, \ldots, \lambda X_n.\, t_n)$ be an assignment for functions-to-synthesize $(f_1, \ldots, f_n)$ whose grammars are $G_1, \ldots, G_n$. Let Ψ_α be the formula:

$$\bigwedge_{i=1}^{n} \forall X_i.\, f(X_i) \approx t_i$$

In other words, Ψ_α contains quantified formulas that constrain the behavior of the functions to the synthesize based on their definition in α. To reason about weight symbols in the definition below, we say that substitution σ is *consistent with* α if it is of the form:

$$\{\omega_{f_i,W} \mapsto k \mid G_i \mapsto^*_{W,k} t_i\}$$

In other words, σ is consistent with α if it replaces weight symbols for functions to synthesize f_i with only valid interpretations based on their bodies t_i in α.

We now state the formal definition for satisfying semantic specifications based on the above definitions. We say that α satisfies the semantic restrictions for conjecture $\exists f_1, \ldots, f_n.\, \Psi$ if $(\Psi_\alpha \wedge \Psi) \cdot \sigma$ is T-valid formula for some substitution σ that is consistent with α. The formal definition for T-valid here corresponds to the definition given by SMT-LIB, for details see Section 5 of [3]. This definition covers cases where $f_1, \ldots, f_n$ have recursive definitions or references to forward declarations, which is why their definitions are explicitly given in the definition of Ψ_α.

6.2.1 Satisfying Semantic Specifications for Optimization Queries

We extend the notion of semantic correctness to responses to optimization queries of the form:

`(optimize-synth (`$s_1 \ldots s_j$`)` $A_1 \ldots A_k$`)`

Recall from Section 4 that a successful response to this command contains:

1. A tuple of terms $(c_1, \ldots, c_j)$ corresponding to the valuation of terms $(s_1, \ldots, s_j)$, and
2. A tuple of functions $(\lambda X_1.\, t_1, \ldots, \lambda X_n.\, t_n)$ corresponding to a solution for functions-to-synthesize $f_1, \ldots, f_n$.

Let $\exists f_1, \ldots, f_n.\, \Psi$ be the synthesis conjecture for the current state, and let Ψ_o be the formula:

$$\bigwedge_{i=1}^{j} s_j \approx c_j$$

A solution of the above form for an optimization query satisfies semantic specifications if only if $(\lambda X_1.\, t_1, \ldots, \lambda X_n.\, t_n)$ satisfies the semantic specifications for the synthesis conjecture $\exists f_1, \ldots, f_n.\Psi \wedge \Psi_o$.

Notice that the attributes $A_1, \ldots, A_k$ specify an ordering on the solution, but do not impact its correctness.

7 Calling Oracles

In the following, we formalize the interface between the synthesis solver and an external binary, which we call an *oracle*. The synthesis solver can query an oracle using text files with the extension *.oracle*. The expected interface is given by the following definition.

Definition 2. An oracle *implements an interface* with input $(\sigma_1 \times \ldots \times \sigma_n)$ and output $(\tau_1 \times \ldots \times \tau_m)$ if it has the following behavior. Let *textfile.oracle* be a file containing the text

$$(v_1 \ldots v_n)$$

where $v_1, \ldots, v_n$ are values[4] of sorts $\sigma_1, \ldots, \sigma_n$ written using the syntax for values described in this document. Let `oracle` be the name of the binary corresponding to the oracle. When the oracle is executed with the command `oracletextfile.oracle`, it returns the text on the standard output channel

$$(w_1 \ldots w_m)$$

where $w_1, \ldots, w_n$ are values of sorts $\tau_1, \ldots, \tau_m$ written using the syntax for values described in this document. □

In other words, the synthesis solver and the external binary communicate with one another using a text interface, where the input text contains a tuple of values corresponding to input to the binary, and the output from the binary is a tuple of terms indicating the output of the binary, which is printed on the standard output channel. It is expected that the synthesis solver parses these values to ascertain the result of the query.

The oracle accepts the name of the text file (`textfile.oracle` in the definition above) as its only command line argument. It is important to note that this text file is expected to take a tuple of values only. In some cases, the syntax for those values may involve symbols that are user-defined. As an example, consider an oracle implementing an interface whose input takes a user-defined datatype. The definition of that datatype is *not* provided as a preamble to the given tuple of values. This means that an external binary is responsible for parsing the given text file, even when it uses symbols whose definitions are not self-contained in that text file. It is assumed that the user has ensured the syncronization of the oracle specification and the input format accepted by the binary.

It is the responsibility of the user to specify oracles in SyGuS commands that implement interfaces of the appropriate sorts. If this is not the case, the synthesis solver may choose to ignore the output of the oracle or terminate with an error.

For example, consider an oracle that implements the interface declared with the following command:

`(oracle-constraint ((`$x_1\ \sigma_1$`) ... (`$x_n\ \sigma_n$`)) ((`$y_1\ \tau_1$`) ... (`$y_m\ \tau_m$`)) `t` `N`)`

The text file must contain a list of values $(v_1 \ldots v_n)$ for the input parameters $x_1 \ldots x_n$, and v_i must have the same sort as σ_i. The oracle will return a list of values $(w_1 \ldots w_m)$, corresponding to the output parameters $y_1 \ldots y_m$, and w_i must have the same sort as τ_i. The solver calls the oracle with the command `N textfile.oracle`, where `N` is the name of the oracle implememtation as specified in the oracle constraint declaration.

[4] A *value* of type T coincides with a term generated by the SyGuS grammar term `(Constant` T`)`. For example, the values of type `Int` have syntax `N` or `(-N)` where `N` is a numeral. The values for function types include all closed lambda terms.

8 Attributes

In this section, we provide standardize attributes that are specific to the features introduced in this document.

8.1 Weights

1. The attribute `:default` expects an integer attribute value n. It specifies that the default weight value of declared weight keyword is n.

Recall from Section 5.2.1 that the default value for weight keywords is 0. On the other hand, the command:

```
(declare-weight numOps :default 1)
```

declares a weight keyword `numOps` whose default weight value is 1.

8.2 Objectives

In this section, we standardize attributes that pertain to specifying objectives of the `optimize-synth` command. We classify two kinds of attributes. The first pertain to the orderings on values, which are supplied in the first argument of `optimize-synth` command. The second pertain to orderings on the tuple itself. Below, we use $\succ$ to denote orderings on values or tuples of values, such that when $e_1 \succ e_2$, we have that e_1 is a more preferred solution.

Value Orderings

1. The attribute `:max` specifies that the ordering on values $\succ$ for a given term t should be maximized according to the default ordering of the type of t. The specification is well-defined only if the type of t has a default ordering. We assume the default ordering $>$ is used for reals and integers, such that $v_1 \succ v_2$ if $v_1 > v_2$. Orderings on other types are not standardized by this document. Note that this attribute expects no provided attribute value.

2. The attribute `:min` is dual to `:max`, such that that ordering $\succ$ should minimize the value based on the default ordering of its type. Like `:max`, this attribute expects no provided attribute value.

Tuple Orderings

1. The attribute `:lexico` specifies that the ordering on tuples is lexicographic. In particular, assuming orderings $\succ_1, \ldots, \succ_n$ on the components of n-tuples, $(c_1, \ldots, c_n) \succ (d_1, \ldots, d_n)$ holds if there exists a $0 \leq j \leq n$ such that $c_i = d_i$ for $i < j$ and $c_j \succ_j d_j$. This attribute expects no provided attribute value.

For example, consider the command:

```
(optimize-synth ((! x :min) (! y :max)) :lexico)
```

This indicates that the objective is to first minimize x, and then maximize y. Hence, in this example, solutions in response to this command are ordered such that $(0, 3) \succ (1, 3) \succ (1, 0)$.

9 Examples

Example 1 (Linear Arithmetic with Constant Coefficients)**.** Consider the following example:

```
(set-logic LIA)
(synth-fun f ((x Int) (y Int)) Int
  ((I Int) (Ic Int))
  ((I Int (0 1 x y
           (+ I I)
           (* Ic I)))
   (Ic Int (0 1 2 (- 1) (- 2)))))
(declare-var x Int)
```

```
(declare-var y Int)
(constraint (= (f x y) (* 2 (+ x y))))
(check-synth)
```

In this example, the logic is set to linear integer arithmetic. The grammar of the function-to-synthesize f has two non-terminals, `I` and `Ic`. What is notable in this example is that the grammar for f includes a rule for `I` whose right hand side is the term $(*$ `Ic I`$)$. If a term of this form were to appear in a constraint, then it would not be allowed since it is the multiplication of two non-constant terms and thus is not allowed by the input logic `LIA`. However, by the definition of our restrictions on grammars in Section 5.4, this grammar *is allowed*, since all closed terms that the grammar generates are allowed by linear arithmetic. A possible correct response for this example from a synthesis solver is:

```
(
  (define-fun f ((x Int) (y Int)) Int (+ (* 2 x) (* 2 y)))
)
```

Example 2 (Datatypes with Linear Arithmetic)**.** Consider the following example:

```
(set-logic DTLIA)
(declare-datatype List
  ((nil)
   (cons (head Int) (tail List))))
(synth-fun f ((x List)) Int
  ((I Int) (L List) (B Bool))
  ((I Int (0 1
           (head L)
           (+ I I)
           (ite B I I)))
   (L List (nil
            x
            (cons I L)
            (tail L)))
   (B Bool (((_ is nil) L)
            ((_ is cons) L)
            (= I I)
            (>= I I)))))
(constraint (= (f (cons 4 nil)) 5))
(constraint (= (f (cons 0 nil)) 1))
(constraint (= (f nil) 0))
(check-synth)
```

In this example, the logic is set to datatypes with linear integer arithmetic `DTLIA`. A datatype `List` is then declared, which encodes lists of integers with two constructors `nil` and `cons`. The input contains a single-function to synthesize f that takes as input a list and returns an integer. Its grammar contains non-terminal symbols for integers, lists and Booleans, and includes applications of constructors `nil` and `cons`, selectors `head` and `tail`, and discriminators (_ `is nil`) and (_ `is cons`). A possible correct response for this example from a synthesis solver is:

```
(
  (define-fun f ((x List)) Int (ite ((_ is nil) x) 0 (+ 1 (head x))))
)
```

In other words, a possible solution for f returns zero whenever its argument is the empty list `nil`, and returns one plus the head of its argument otherwise.

Example 3 (Bit-Vectors with Concatenation and Extraction)**.** Consider the following example:

```
(set-logic BV)
(synth-fun f ((x (_ BitVec 32))) (_ BitVec 32)
  ((BV32 (_ BitVec 32)) (BV16 (_ BitVec 16)))
  ((BV32 (_ BitVec 32) (#x00000000 #x00000001 #xFFFFFFFF
                        x
                        (bvand BV32 BV32)
```

```
                          (bvor BV32 BV32)
                          (bvnot BV32)
                          (concat BV16 BV16)
                          ))
    (BV16 (_ BitVec 16) (#x0000 #x0001 #xFFFF
                          (bvand BV16 BV16)
                          (bvor BV16 BV16)
                          (bvnot BV16)
                          ((_ extract 31 16) BV32)
                          ((_ extract 15 0) BV32)))))
(constraint (= (f #x0782ECAD) #xECAD0000))
(constraint (= (f #xFFFF008E) #x008E0000))
(constraint (= (f #x00000000) #x00000000))
(check-synth)
```

The this example, the logic is set to bit-vectors `BV`. A single function-to-synthesize f is given that takes a bit-vector of bit-width 32 as input and returns a bit-vector of the same width as output. Its grammar involves non-terminals whose sorts are bit-vectors of bit-width 32 and 16. The semantics of the operators in this example are defined in the SMT-LIB standard. In particular, the operator `concat` concatenates its two arguments, and the indexed operator (_ `extract` `n` `m`) returns a bit-vector containing bits n through m of its argument, where $n \geq m$. A possible correct response for this example from a synthesis solver is:

```
(
  (define-fun f ((x (_ BitVec 32))) (_ BitVec 32)
     (concat ((_ extract 15 0) x) #x0000))
)
```

In other words, a possible solution for f returns the concatenation of bits 15 to 0 of its argument with the bit-vector #`x0000`.

Example 4 (Grammars with Defined Symbols, Forward Declarations and Recursion)**.** Consider the following example:

```
(set-logic LIA)
(set-feature :fwd-decls true)
(set-feature :recursion true)
(define-fun x_plus_one ((x Int)) Int (+ x 1))
(synth-fun f ((x Int)) Int
  ((I Int))
  ((I Int (0 1 x (x_plus_one I)))))
(define-fun fx_plus_one ((x Int)) Int (+ (f x) 1))
(synth-fun g ((x Int)) Int
  ((I Int))
  ((I Int (0 1 x (fx_plus_one I)))))
(synth-fun h ((x Int)) Int
  ((I Int) (B Bool))
  ((I Int (0 1 x (- I 1) (+ I I) (h I) (ite B I I)))
   (B Bool ((= I I) (> I I)))))
(declare-var y Int)
(constraint (= (h y) (- (g y) (f y))))
(check-synth)
```

This example contains three well-formed `synth-fun` commands. The first one for function f contains an application of a macro `x_plus_one`, which abbreviates adding one to its argument. This grammar is allowed in the default SyGuS logic and in this example. The grammar for g contains an application of a macro `fx_plus_one`, whose expanded form contains a previously declared function-to-synthesize f. This grammar is allowed since the feature `fwd-decls` is enabled in this example. The grammar for h contains a rule that contains an application of h itself. This grammar is allowed since `recursion` is enabled in this example. A possible correct response for this example from a synthesis solver is:

```
(
  (define-fun f ((x Int)) Int x)
```

```
  (define-fun g ((x Int)) Int (fx_plus_one x))
  (define-fun h ((x Int)) Int 1)
)
```

Notice the above definitions for f, g, and h are given in the order in which they were introduced via `synth-fun` commands in the input.

Example 5 (Programming by Examples (PBE) with Strings)**.** Consider the following example:

```
(set-logic PBE_SLIA)
(synth-fun f ((fname String) (lname String)) String
  ((y_str String) (y_int Int))
  ((y_str String (" " fname lname
                  (str.++ y_str y_str)
                  (str.replace y_str y_str y_str)
                  (str.at y_str y_int)
                  (str.from_int y_int)
                  (str.substr y_str y_int y_int)))
     (y_int Int (0 1 2
                 (+ y_int y_int)
                 (- y_int y_int)
                 (str.len y_str)
                 (str.to_int y_str)
                 (str.indexof y_str y_str y_int)))))
(constraint (= (f "Nancy" "FreeHafer") "Nancy FreeHafer"))
(constraint (= (f "Andrew" "Cencici") "Andrew Cencici"))
(constraint (= (f "Jan" "Kotas") "Jan Kotas"))
(constraint (= (f "Mariya" "Sergienko") "Mariya Sergienko"))
(check-synth)
```

In this example, the logic is set to the SyGuS-specific logic `PBE_SLIA`, indicating strings with linear integer arithmetic where constraints are limited to a conjunction of PBE equalities. In this example, four constraints are given, each of which meet the restriction of being a PBE equality. A possible correct response for this example from a synthesis solver is:

```
(
  (define-fun f ((fname String) (lname String)) String
    (str.++ fname (str.++ " " lname)))
)
```

Example 6 (Weights)**.** Consider the following example:

```
(set-logic NIA)
(set-feature :weights true)
(declare-weight numX)
(synth-fun f ((x Int)) Int
  ((I Int))
  ((I Int (0 1
           (! x :numX 1)
           (+ I I)
           (! (* x x) :numX 2)))))
(constraint (= (_ numX f) 3))
(check-synth)
```

In this example, a weight keyword `numX` has been declared. A function-to-synthesize f has been declared, whose production rules include annotations. These annotations count the number occurrences of x, where recall that the default value for unannotated terms with respect to the given weight is 0. A constraint is given that states that the weight of f with respect to the weight keyword `numX` must be 3. A possible correct response for this example from a synthesis solver is:

```
(
  (define-fun f ((x Int)) Int (+ x (* x x)))
)
```

Example 7 (Weight with Multiple Interpretations)**.** Consider the following example:

```
(set-logic LIA)
(set-feature :weights true)
(declare-weight numI)
(synth-fun f ((x Int)) Int
  ((I Int))
  ((I Int (0 1 x
            (+ x 1)
            (! (- I) :numI 1)
            (! (+ I I) :numI 2)))))
(define-fun numRulesForF () Int (+ (_ numI f) 1))
(declare-var x Int)
(constraint (and (> (f x) x) (< numRulesForF 3)))
(check-synth)
```

In this example, a weight keyword `numI` has been declared, which counts the number of occurrences of the non-terminal `I` that were used in applications of the production rules that generated the body of f. The function `numRulesForF` is defined to be the weight of f with respect to this keyword plus one, or in other words, the number of rules used for generating the body of f. A possible correct response for this example from a synthesis solver is:

```
(
  (define-fun f ((x Int)) Int (+ x 1))
)
```

In this example, the body `(+ x 1)` given for f can be generated in multiple ways from the given grammar. In particular, it could be generated directly from the start symbol, or e.g. via the sequence `I`, `(+ I I)`, `(+ x I)`, `(+ x 1)`. Thus, when the body of f is interpreted as `(+ x 1)`, the interpretation of `(+ (_ numI f) 1)` may either be 1 or 3. The constraint `(< numRulesForF 3)` is satisfied in the former interpretation, and hence the solution for f satisfies all input constraints.

Example 8 (Optimization with Weights)**.** Consider the following example:

```
(set-logic LIA)
(set-feature :weights true)
(declare-weight branches)
(synth-fun f ((x Int)) Int
   ((I Int) (B Bool))
   ((I Int (0 1 x (+ I I) (! (ite B I I) :branches 1)))
    (B Bool ((>= I I) (= I I)))))
(declare-var x Int)
(constraint (= (f x) (ite (= x 0) 0 x)))
(optimize-synth ((! (_ branches f) :min)))
```

In this example, a weight keyword `branches` has been declared. The grammar of function `f` has been annotated such that `branches` corresponds to the number of `ite` terms in its body. An optimization query is made, where the objective is to minimize the number of branches in `f`. This is specified by a term annotation on `(_ branches f)`. A possible correct response for this example is:

```
(
(1)
(define-fun f ((x Int)) Int (ite (= x 0) 0 x))
)
```

Another possible response for this example is:

```
(
(0)
(define-fun f ((x Int)) Int x)
)
```

In both cases, the value of term `(_ branches f)` was given in the solution that was consistent with the number of `ite` terms in the body of `f`. Hence, both responses are correct. The latter solution gives a smaller value for this term, and hence it is preferred based on the specified objective.

Notice that in this example, no attribute is provided for the `optimize-synth` command, meaning that no ordering is specified on the tuple passes as argument to this command. For details on the default semantics for objectives, see Section 8.2. In the next example, we show how ordering on tuples can be specified.

Example 9 (Lexicographic Optimization)**.** Consider the following example:

```
(set-logic LIA)
(synth-fun f ((x Int)) Int
   ((I Int) (B Bool))
   ((I Int (0 1 x (+ I I) (ite B I I)))))
(declare-var x Int)
(constraint (or (= (f x) 1) (= (f x) x)))
(optimize-synth ((! (f 0) :max) (! (f 100) :max)) :lexico)
```

This example also contains an optimization query. The objective of the query is to optimize the values of terms `(f 0)` and `(f 100)`. Moreover, the ordering on tuples of these terms is lexicographic with respect to the orderings specified for each term position. In particular, this means that the best solution for this example is one that first maximizes `(f 0)` and then maximizes `(f 100)`. A possible correct response for this example is:

```
(
(1 1)
(define-fun f ((x Int)) Int 1)
)
```

Another possible response for this example is:

```
(
(0 100)
(define-fun f ((x Int)) Int x)
)
```

The first example is preferred, since the first value in the first solution is greater than the first value in the second, since the command specified that solutions are ordered lexicographically.

Example 10 (Single Invariant Synthesis (Inv) over Linear Integer Arithmetic)**.** Consider the following imperative program in a C-like language:

```
int x, y;
assume (5 <= x && x <= 9);
assume (1 <= y && y <= 3);
while (*) {
  x += 2;
  y += 1;
}
assert (y < x);
```

The problem of synthesizing an invariant to verify this program, can be expressed in SyGuS as follows:

```
(set-logic Inv_LIA)
(synth-fun inv-f ((x Int) (y Int)) Bool)
(define-fun pre-f ((x Int) (y Int)) Bool
  (and (<= 5 x) (<= x 9) (<= 1 y) (<= y 3)))
(define-fun trans-f ((x Int) (y Int) (xp Int) (yp Int)) Bool
  (and (= xp (+ x 2)) (= yp (+ y 1))))
(define-fun post-f ((x Int) (y Int)) Bool (< y x))
(inv-constraint inv-f pre-f trans-f post-f)
(check-synth)
```

In this example, the logic is set to the SyGuS-specific logic `Inv_LIA`, indicating linear integer arithmetic where constraints are limited to the invariant synthesis problem for a single invariant-to-synthesize. Since this example contains only linear integer arithmetic terms, a single predicate-to-synthesize `inv-f`, and only introduces constraints via a single `inv-constraint` command, it meets the restrictions of the logic. A possible correct response for this example from a synthesis solver is:

```
(
  (define-fun inv-f ((x Int) (y Int)) Bool (> x y))
)
```

Example 11 (CHCs over Linear Integer Arithmetic with a Single Predicate to Synthesize)**.** The verification problem from the previous example can also be expressed as a system of CHCs, which can be encoded in SyGuS as follows:

```
(set-logic CHC_LIA)
(synth-fun inv-f ((x Int) (y Int)) Bool)
(chc-constraint ((x Int) (y Int))
  (and (<= 5 x) (<= x 9) (<= 1 y) (<= y 3))
  (inv-f x y))
(chc-constraint ((x Int) (y Int) (xp Int) (yp Int))
  (and (inv-f x y) (= xp (+ x 2)) (= yp (+ y 1)))
  (inv-f xp yp))
(chc-constraint ((x Int) (y Int))
  (and (inv-f x y) (not (< y x)))
  false)
(check-synth)
```

In this example, the logic is set to the SyGuS-specific logic `CHC_LIA`, indicating linear integer arithmetic where constraints are limited to constrained Horn clauses. Since this example contains only linear integer arithmetic terms, only predicate symbols to be synthesized (i.e., `inv-f`, only introduces valid CHCs via `chc-constraint` command, and only a single CHC query, it meets the restrictions of the logic. As for the previous example, a possible correct response for this example from a synthesis solver is:

```
(
  (define-fun inv-f ((x Int) (y Int)) Bool (> x y))
)
```

Example 12 (CHCs over Linear Integer Arithmetic with Multiple Predicates to Synthesize)**.** Consider the following imperative program in a C-like language:

```
int x;
int y = x, n = 0;
while (*) {
  x += 1;
  n += 1;
}
x *= 2;
while (n != 0) {
  x -= 2;
  n -= 1;
}
assert (x == 2 * y);
```

The verification problem for this program is more naturally expressed as a system of CHCs using the `CHC` logic in SyGuS instead of the `Inv` logic:

```
(set-logic CHC_LIA)
(synth-fun inv1 ((x Int) (y Int) (n Int)) Bool)
(synth-fun inv2 ((x Int) (y Int) (n Int)) Bool)
(chc-constraint ((x Int) (y Int) (n Int))
  (and (= y x) (= n 0))
  (inv1 x y n))
(chc-constraint ((x Int) (y Int) (n Int) (xp Int) (np Int))
  (and (inv1 x y n) (= xp (+ x 1)) (= np (+ n 1)))
  (inv1 xp y np))
(chc-constraint ((x Int) (y Int) (n Int) (xp Int))
  (and (inv1 x y n) (= xp (* 2 x)))
  (inv2 xp y n))
```

```
(chc-constraint ((x Int) (y Int) (n Int) (xp Int) (np Int))
  (and (inv2 x y n) (not (= n 0)) (= xp (- x 2)) (= np (- n 1)))
  (inv2 xp y np))
(chc-constraint ((x Int) (y Int) (n Int))
  (and (= n 0) (inv2 x y n) (not (= x (* 2 y))))
  false)
(check-synth)
```

In this example, the logic is set to the SyGuS-specific logic `CHC_LIA`, indicating linear integer arithmetic where constraints are limited to constrained Horn clauses. Since this example contains only linear integer arithmetic terms, only predicate symbols to be synthesized (i.e., `inv1` and `inv2`), only introduces valid CHCs via `chc-constraint` command, and only a single CHC query, it meets the restrictions of the logic. A possible correct response for this example from a synthesis solver is:

```
(
  (define-fun inv1 ((x Int) (y Int) (n Int)) Bool (= x (+ y n)))
  (define-fun inv2 ((x Int) (y Int) (n Int)) Bool (= x (* 2 (+ y n))))
)
```

Example 13 (PBE with Oracle Function Symbols)**.** Oracle function symbols are used to represent black-box functions within specifications. Consider a PBE problem where the user has an external oracle and wishes to synthesize a summary of the external oracle for a specific set of inputs. This can be encoded using an oracle function symbol (note this is syntactic sugar for an oracle assumption and an uninterpreted function):

```
(set-logic BV)
(set-feature :oracles true)
(synth-fun f ((x (_ BitVec 64))) (_ BitVec 64))
(declare-oracle-fun target ((_ BitVec 64)) (_ BitVec 64) binaryname)
(constraint (= (f #x28085a970e13e12c) (target #x28085a970e13e12d)))
(constraint (= (f #xbe5341bebd2a0749) (target #xbe5341bebd2a0749)))
(constraint (= (f #xe239460eed2cc34e) (target #xe239460eed2cc34f)))
(constraint (= (f #xac5b1b5e9b236b10) (target #xac5b1b5e9b236b11)))
(constraint (= (f #x4069a4c7173e1786) (target #x4069a4c7173e1786)))
(constraint (= (f #x39419062091119a6) (target #x39419062091119a6)))
(constraint (= (f #x49aeeca628644ee0) (target #x49aeeca628644ee0)))
(constraint (= (f #x75e5bc2a07c77c97) (target #x75e5bc2a07c77c97)))
(constraint (= (f #x4c5ee4be98c5ee7d) (target #x4c5ee4be98c5ee7d)))
(constraint (= (f #xcd67bd5beaac575e) (target #xcd67bd5beaac575e)))
(check-synth)
```

In this way, a PBE problem can be encoded without the user knowing the correct behaviour of the target function explicitly. Instead, the target function is implemented by the oracle `binaryname`, and the solver can call `binaryname` with an input given in a constraint and `binaryname` returns the result of applying `target` to the input.

Example 14 (Synthesis with Oracle Assumptions)**.** In addition to their use in oracle function symbols, oracle assumptions can be used to relax a synthesis specification in other ways. Consider the following synthesis problem. The oracle, when called, generates an assumption that relaxes the values of x that the specification must hold for, i.e., when $x < z$ the specification need not hold. This could be used, for example, in invariant synthesis if an oracle can determine that certain parts of the state-space are unreachable from the initial states and backward unreachable from the violation of the post-condition. This could be particularly useful if the specified transition relation is over-approximate.

```
(set-logic LIA)
(set-feature :oracles true)
(synth-fun inv-f ((x Int)) Bool)
(declare-var x Int)
(declare-var x! Int)
(define-fun pre-f ((x Int)) Bool (= (mod x 2) 0))
(define-fun trans-f ((x Int)(x! Int)) Bool (or (= x! x) (= x! (* x (x-1)))))
```

```
(define-fun post-f ((x Int)) Bool (= (mod x 4) 0)
(constraint (=> (pre-f x) (inv-f x)))
(constraint (=> (and (inv-f x) (trans-f x x!)) (inv-f x!)))
(constraint (=> (inv-f x) (post-f x)))
(oracle-assume ((inv-candidate (-> Int Bool))) ((z Int)) binaryname
  (>= x z))
(check-synth)
```

Example 15 (Synthesis with Oracle Constraints and Oracle Function Symbols)**.** As well as PBE style problems, oracle function symbols can be used to incorporate other verification engines via correctness oracles. Here, an oracle function symbol must return "true" before the synthesised function is valid. However, a black-box correctness oracle along does not provide much information to the synthesiser. This oracle can be supplemented with an oracle constraint. Here the oracle constraint provides a fresh positive witness whenever it is called, constraining the space of possible candidate functions.

```
(set-logic LIA)
(set-feature :oracles true)
(synth-fun f ((x Int)) Bool)
(declare-oracle-fun isCorrect ((-> Int Bool)) Bool binaryname)
(constraint (isCorrect f))
(oracle-constraint () ((x Int)) ((z Bool)) binaryname2
  (=> (f x) z))
(check-synth)
```

References

[1] Rajeev Alur, Dana Fisman, P. Madhusudan, Rishabh Singh, and Armando Solar-Lezama. SyGuS Syntax for SyGuS-COMP 15. http://sygus.org, 2015.

[2] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. SyGuS Syntax for SyGuS-COMP 16. http://sygus.org, 2016.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

[4] Susmit Jha and Sanjit A. Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica*, 54(7):693–726, 2017.

[5] Mukund Raghothaman and Abhishek Udupa. Language to specify syntax-guided synthesis problems. 2014.

[6] Andrew Reynolds and Chenlong Wang. Proposal for a theory of tables. 2021.

[7] Cesare Tinelli, Clark Barrett, and Pascal Fontaine. Unicode Strings (Draft 1.0). http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml, 2018.

A Reserved Words

A *reserved word* is any of the literals from Section 2.2, or any of the following keywords:
`!`, `_`, `check-synth`, `Constant`, `chc-constraint`, `constraint`, `declare-correctness-cex-oracle`, `declare-correctnes`
`declare-datatype`, `declare-datatypes`, `declare-oracle-fun`, `declare-sort`, `declare-var`, `declare-weight`,
`define-fun`, `define-sort`, `exists`, `forall`, `inv-constraint`, `let`, `optimize-synth`, `oracle-assume`,
`oracle-constraint`, `oracle-constraint-cex`, `oracle-constraint-io`, `oracle-constraint-membership`,
`oracle-constraint-negwitness`, `oracle-constraint-poswitness`, `set-feature`, `set-info`, `set-logic`,
`set-option`, `synth-fun`, and `Variable`.

B Reference Grammars

In this section, for convenience, we provide the concrete syntax for grammars that generate exactly the set of terms that belong to SMT-LIB logics of interest for a fixed set of free variables. In particular, this means that each of the following `synth-fun` commands are equivalent to those in which no grammar is provided. Note this is not intended to be a complete list of logics. In particular, we focus on logics that include a single background theory whose sorts are not parametric. Each of these grammars are derived based on the definition of logics and theories described in the theory and logic declaration documents available at www.smt-lib.org.

For each grammar, we omit the predicate symbols that are shared by all logics according to the SMT-LIB standard, that is, those included in the core theory described in Section 5.1, which includes Boolean connectives and equality. We provide the grammar for a single function over one of the sorts in the logic, and assume it has one variable in its argument list for each non-Boolean sort in the grammar.

B.1 Integer Arithmetic

The following grammar for f generates exactly the integer-typed terms in the logic of linear integer arithmetic (LIA) with one free integer variable `x`.

```
(set-logic LIA)
(synth-fun f ((x Int)) Int
  ((y_int Int) (y_const_int Int) (y_bool Bool))
  ((y_int Int (y_const_int
               (Variable Int)
               (- y_int)
               (+ y_int y_int)
               (- y_int y_int)
               (* y_const_int y_int)
               (* y_int y_const_int)
               (div y_int y_const_int)
               (mod y_int y_const_int)
               (abs y_int)
               (ite y_bool y_int y_int)))
   (y_const_int Int ((Constant Int)))
   (y_bool Bool ((= y_int y_int)
                 (> y_int y_int)
                 (>= y_int y_int)
                 (< y_int y_int)
                 (<= y_int y_int)))))
```

Above, `div` denotes integer division, `mod` denotes integer modulus and `abs` denotes the absolute value function. Positive integer constants and zero are written using the syntax for numerals $\langle Numeral \rangle$. Negative integer constants are written as the unary negation of a positive integer constant.

The following grammar for g generates exactly the integer-typed terms in the logic of non-linear integer arithmetic (NIA) with one free integer variable x.

```
(set-logic NIA)
(synth-fun g ((x Int)) Int
  ((y_int Int) (y_bool Bool))
```

```
  ((y_int Int ((Constant Int)
               (Variable Int)
               (- y_int)
               (+ y_int y_int)
               (- y_int y_int)
               (* y_int y_int)
               (div y_int y_int)
               (mod y_int y_int)
               (abs y_int)
               (ite y_bool y_int y_int)))
   (y_bool Bool ((= y_int y_int)
                 (> y_int y_int)
                 (>= y_int y_int)
                 (< y_int y_int)
                 (<= y_int y_int)))))
```

B.2 Real Arithmetic

The following grammar for f generates exactly the real-typed terms in the logic of linear real arithmetic (LRA) with one free real variable x.

```
(set-logic LRA)
(synth-fun f ((x Real)) Real
  ((y_real Real) (y_const_real Real) (y_bool Bool))
  ((y_real Real (y_const_real
                 (Variable Real)
                 (- y_real)
                 (+ y_real y_real)
                 (- y_real y_real)
                 (* y_const_real y_real)
                 (* y_real y_const_real)
                 (/ y_real y_const_real)
                 (ite y_bool y_real y_real)))
   (y_const_real Real ((Constant Real)))
   (y_bool Bool ((= y_real y_real)
                 (> y_real y_real)
                 (>= y_real y_real)
                 (< y_real y_real)
                 (<= y_real y_real)))))
```

Notice that positive real constants and zero can either be written as decimal values using the syntax $\langle Decimal \rangle$ or as rationals, e.g. the division of two numerals `(/ m n)` where n is not zero. Negative reals are written either as the unary negation of decimal value or a rational of the form `(/ (- m) n)` for numerals m and n where n is not zero.

The following grammar for g generates exactly the real-typed terms in the logic of non-linear real arithmetic (NRA) with one free real variable x.

```
(set-logic NRA)
(synth-fun g ((x Real)) Real
  ((y_real Real) (y_bool Bool))
  ((y_real Real ((Constant Real)
                 (Variable Real)
                 (- y_real)
                 (+ y_real y_real)
                 (- y_real y_real)
                 (* y_real y_real)
                 (/ y_real y_real)
                 (ite y_bool y_real y_real)))
   (y_bool Bool ((= y_real y_real)
                 (> y_real y_real)
                 (>= y_real y_real)
```

```
                    (< y_real y_real)
                    (<= y_real y_real)))))
```

B.3 Fixed-Width Bit-Vectors

The signature of bit-vectors includes an indexed sort `BitVec`, which is indexed by an integer constant that denotes its bit-width. We show a grammar below for one particular choice of this bit-width, 32. We omit indexed bit-vector operators such as extraction function `(_ extract m n)`, the concatenation function `concat`, since these operators are polymorphic. For brevity, we also omit the *extended* operators of this theory as denoted by SMT-LIB, which includes functions like bit-vector subtraction `bvsub`, exclusive or `bvxor`, signed division `bvsdiv`, and predicates like unsigned-greater-than-or-equal `bvuge` and signed-less-than `bvslt`. These extended operators can be seen as syntactic sugar for the operators in the grammar below. Example inputs that involve some of the omitted operators are given in Section 9.

```
(set-logic BV)
(synth-fun f ((x (_ BitVec 32))) (_ BitVec 32)
  ((y_bv (_ BitVec 32)) (y_bool Bool))
  ((y_bv (_ BitVec 32) ((Constant (_ BitVec 32))
                        (Variable (_ BitVec 32))
                        (bvnot y_bv)
                        (bvand y_bv y_bv)
                        (bvor y_bv y_bv)
                        (bvneg y_bv)
                        (bvadd y_bv y_bv)
                        (bvmul y_bv y_bv)
                        (bvudiv y_bv y_bv)
                        (bvurem y_bv y_bv)
                        (bvshl y_bv y_bv)
                        (bvlshr y_bv y_bv)
                        (ite y_bool y_bv y_bv)))
   (y_bool Bool ((bvult y_bv y_bv)))))
```

Bit-vector constants may be specified using either the hexadecimal format $\langle HexConst \rangle$ or the binary format $\langle BinConst \rangle$.

B.4 Strings

The syntax below reflects the official version 2.6 release of the theory of Unicode strings and regular expressions [3]. Some operators (e.g. indexed regular expression operators) have been omitted from the grammar below for the sake of brevity.

```
(set-logic S)
(synth-fun f ((xs String) (xr RegLan) (xi Int)) String
  ((y_str String) (y_rl RegLan) (y_int Int) (y_bool Bool))
  ((y_str String ((Constant String)
                  (Variable String)
                  (str.++ y_str y_str)
                  (str.at y_str y_str)
                  (str.substr y_str y_int y_int)
                  (str.indexof y_str y_str y_int)
                  (str.replace y_str y_str y_str)
                  (str.from_int y_int)
                  (str.from_code y_int)
                  (ite y_bool y_str y_str)))
    (y_rl RegLan ((Constant RegLan)
                  (Variable RegLan)
                  re.none
                  re.all
                  re.allchar
                  (str.to_re y_str)
```

```
                    (re.++ y_rl y_rl)
                    (re.union y_rl y_rl)
                    (re.inter y_rl y_rl)
                    (re.* y_rl)
                    (re.+ y_rl)
                    (re.opt y_rl)
                    (re.range y_str y_str)
    (y_int Int ((Constant Int)
                (Variable Int)
                (str.len y_str)
                (str.to_int y_str)
                (str.to_code y_str)
                (ite y_bool y_int y_int)))
    (y_bool Bool ((Constant Bool)
                  (Variable Bool)
                  (str.in_re y_str y_rl)
                  (str.contains y_str y_str)
                  (str.prefixof y_str y_str)
                  (str.suffixof y_str y_str)
                  (str.< y_str y_str)
                  (str.<= y_str y_str)
                  (str.is_digit y_str)))))
```

String constants may be specified by text delimited by double quotes based on the syntax ⟨*StringConst*⟩ described in Section 2.2. The sort `RegLan` denotes the regular expression sort. Its values are all ground (i.e. variable free) regular expressions.

Notice that since the logic specified above is `S`. The signature of this logic includes some functions involving the integer sort like `str.len`. However, the above logic does not permit inputs containing integer constants or the standard symbols of arithmetic like `+`, `-`, `>=` and so on, since the logic `S` does not include the theory of integer arithmetic. Thus in pratice, the string theory is frequently combined with the theory of integers in the logic `SLIA`, i.e. strings with linear integer arithmetic.

C Language Features of SMT-LIB not Covered

For the purpose of self-containment, many of the essential language features of SMT-LIB version 2.6 are redefined in this document. However, other less essential ones are omitted. We briefly mention other language features not mentioned in this document. We do not require solvers to support these features. However, we recommend that if solvers support any of the features below, they use SMT-LIB compliant syntax, as described briefly below.

Parametric Datatype Definitions We do not cover the concrete syntax or semantics of parametric datatypes (that is, datatypes whose arity is non-zero) in this document. An example of a datatype definition for a parametric list is given below.

```
(declare-datatypes ((List 1)) (
  (par (T) ((nil) (cons (head T) (tail (List T)))))))
```

Above, the datatype `List` is given in the predeclaration with the numeral `1`, indicating its arity is one. The datatype definition that follows includes quantification on a type parameter `T`, where this quantification is specified using the `par` keyword. Within the body of this quantification, a usual constructor listing is given, where the type parameter `T` may occur free. The constructors of this datatype are `nil` and `cons`.

Qualified Identifiers In SMT-LIB version 2.6, identifiers that comprise terms may be *qualified* with a type-cast, using the keyword `as`. Type casts are required for symbols whose type is ambiguous, such as parametric datatype constructors, e.g. the nil constructor for a parametric list For the parametric datatype above, (`as nil (List Int)`) and (`as nil (List Real)`) denote the type constructors for the empty list of integers and reals respectively.

Match Terms The SMT-LIB version 2.6 includes a `match` term that case splits on the constructors of datatype terms. For example, the match term:

```
(match x ((nil (- 1)) ((cons h t) h)))
```

returns negative one if the list x is empty, or the first element of x (its head) if it is non-empty.

Recursive Functions The SMT-LIB version 2.6 includes a command `define-fun-rec` for defining symbols that involve recursion.[5] An example, which computes the length of a (non-parametric) list is given below.

```
(define-fun-rec len ((x List)) Int
  (match x ((nil 0) ((cons h t) (+ 1 (len t))))))
```

Functions may be defined to be mutually recursive by declaring them in a single block using the `define-funs-rec` command. An example, which defines two predicates `isEven` and `isOdd` for determining whether a positive integer is even and odd respectively, is given in the following.

```
(define-funs-rec (
  (isEven ((x Int)) Bool)
  (isOdd ((x Int)) Bool)
)(
  (ite (= x 0) true (isOdd (- x 1)))
  (ite (= x 0) false (isEven (- x 1)))
))
```

Notice that recursive function definitions are not required to be terminating (the above functions do not terminate for negative integers). They are not even required to correspond to consistent definitions, for instance:

```
(define-fun-rec inconsistent ((x List)) Int
  (+ (inconsistent x) 1))
```

The semantics of recursive functions is given in the SMT-LIB version 2.6 standard [3]. Each recursive function can be seen as a universally quantified constraint that asserts that each set of values in the domain of the function is equal to its body.

Additional Logics and Theories Several standard logics and theories are omitted from discussion in this document. This includes the (mixed) theory of integers and reals, the theory of floating points, the integer and real difference logic, and their combinations. More details on the catalog of logics and theories in the SMT-LIB standard is available at www.smt-lib.org.

[5] Recall that recursive definitions are prohibited from macro definitions in the command `define-fun`.