The SyGuS Language Standard Version 2.0

Mukund Raghothaman Andrew Reynolds Abhishek Udupa

Friday 15th March, 2019

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 $f_1, f_2, \ldots, f_n$ 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 that describe the behavior of the functions to the synthesized. This is described as a quantifier-free formula φ in the input logic, and may refer to the functions-to-synthesize as well as universally quantified variables $v_1, v_2, \ldots, v_m$.

The objective then is to find interpretations 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 $\forall v_1, v_2, \ldots, v_m.\ \varphi$ 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 whose free variables may contain 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 input language version 2.0, which is intended to be used as the standard input and output language for solvers targeting the syntax-guided synthesis problem and users of these solvers. The language borrows many concepts and language constructs from the standard format for Satisfiability Modulo Theories (SMT) solvers, the SMT-LIB language (version 2.6) [3].

Outline In the remainder of this section, we cover differences of the language described in this document with previous revisions of the SyGuS input language [4] 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 7 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 SyGuS standard document [4], and the extensions described in [1, 2].

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 version 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 to SMT-LIB [5]. Thus, certain new symbols are now present in the signature, and some existing ones have changed names. For example, the conversion functions between strings and their integer representations `str.to.int` and `int.to.str` have been renamed to `str.to-int` and `str.from-int`. The semantics of all operators, which was not specified previously, is now the one given by [5].

8. The command `declare-primed-var`, which was a syntactic sugar for two `declare-var` commands in the previous 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 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 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.

1.2 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 $t[\vec{s}]$.

2 Syntax

In this section, we describe the concrete syntax of SyGuS version 2.0 inputs. Many constructs in this syntax coincide with those in the SMT-LIB version 2.6 format [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.

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

The syntax for commands $\langle Cmd \rangle$ is given at the end of this section. We first introduce the necessary preliminary definitions.

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* $\langle Literal \rangle$ 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 the SMT-LIB version 2.6 format. For full details, see Section 3.1 of [3].

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

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

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

Booleans ($\langle BoolConst \rangle$) Symbols `true` and `false` denote the Booleans true and false.

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

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

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:

$$\_ \; + \; - \; * \; \& \; | \; ! \; \sim \; < \; > \; = \; / \; \% \; ? \; . \; \$ \; \hat{} \;$$

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

2.4 Identifiers

An identifier ⟨*Identifier*⟩ is a syntactic extension of symbols that includes symbols that are indexed by integer constants or other symbols.

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

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

2.5 Sorts

We work in a multi-sorted logic where terms are associated with sorts ⟨*Sort*⟩. Sorts are constructed via the following syntax.

$$\langle Sort \rangle \;\; ::= \;\; \langle Identifier \rangle \;\; | \;\; (\; \langle Identifier \rangle \;\; \langle Sort \rangle^{+} \;)$$

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).

2.6 Terms

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

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

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 Features

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

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

More details on features are given in Section 5.2.

2.8 Commands

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

$$
\begin{array}{rcl}
\langle Cmd \rangle & ::= & (\texttt{check-synth}) \\
& | & (\texttt{constraint} \; \langle Term \rangle \,) \\
& | & (\texttt{declare-var} \; \langle Symbol \rangle \;\; \langle Sort \rangle \,) \\
& | & (\texttt{inv-constraint} \; \langle Symbol \rangle \;\; \langle Symbol \rangle \;\; \langle Symbol \rangle \;\; \langle Symbol \rangle \,) \\
& | & (\texttt{set-feature} \; : \langle Feature \rangle \;\; \langle BoolConst \rangle \,) \\
& | & (\texttt{synth-fun} \; \langle Symbol \rangle \;\; (\, \langle SortedVar \rangle^{*} \,) \;\; \langle Sort \rangle \;\; \langle GrammarDef \rangle^{?} \,) \\
& | & (\texttt{synth-inv} \; \langle Symbol \rangle \;\; (\, \langle SortedVar \rangle^{*} \,) \;\; \langle GrammarDef \rangle^{?} \,) \\
& | & \langle SmtCmd \rangle \\
\\
\langle SmtCmd \rangle & ::= & (\texttt{declare-datatype} \; \langle Symbol \rangle \;\; \langle DTDec \rangle \,) \\
& | & (\texttt{declare-datatypes} \; (\, \langle SortDecl \rangle^{n+1} \,) \; (\, \langle DTDec \rangle^{n+1} \,)) \\
& | & (\texttt{declare-sort} \; \langle Symbol \rangle \;\; \langle Numeral \rangle \,) \\
& | & (\texttt{define-fun} \; \langle Symbol \rangle \;\; (\, \langle SortedVar \rangle^{*} \,) \;\; \langle Sort \rangle \;\; \langle Term \rangle \,) \\
& | & (\texttt{define-sort} \; \langle Symbol \rangle \;\; \langle Sort \rangle \,) \\
& | & (\texttt{set-logic} \; \langle Symbol \rangle \,) \\
& | & (\texttt{set-option} \; : \langle Symbol \rangle \;\; \langle Literal \rangle \,) \\
\\
\langle SortDecl \rangle & ::= & (\, \langle Symbol \rangle \;\; \langle Numeral \rangle \,) \\
\langle DTDec \rangle & ::= & (\, \langle DTConsDec \rangle^{+} \,) \\
\langle DTConsDec \rangle & ::= & (\, \langle Symbol \rangle \;\; \langle SortedVar \rangle^{*} \,) \\
\langle GrammarDef \rangle & ::= & (\, \langle SortedVar \rangle^{n+1} \,) \; (\, \langle GroupedRuleList \rangle^{n+1} \,) \\
\langle GroupedRuleList \rangle & ::= & (\, \langle Symbol \rangle \;\; \langle Sort \rangle \;\; (\, \langle GTerm \rangle^{+} \,)) \\
\\
\langle GTerm \rangle & ::= & (\texttt{Constant} \; \langle Sort \rangle \,) \mid (\texttt{Variable} \; \langle Sort \rangle \,) \mid \langle BfTerm \rangle
\end{array}
$$

For convenience, we distinguish between two kinds of commands above. The commands listed under $\langle Cmd \rangle$ are specific to the SyGuS version 2.0 standard. The remaining commands listed under $\langle SmtCmd \rangle$ are borrowed from the SMT-LIB version 2.6 standard. Details on the semantics of these commands are given 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 of the form:

$$\exists f_1, \ldots, f_n. \forall x_1, \ldots, x_m. (\varphi_1 \wedge \ldots \wedge \varphi_p)[f_1, \ldots, f_n, x_1, \ldots, x_m]$$

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 $x_1, \ldots, x_m$ of variables, which we refer to the current list of *universal variables*,
- A list of formulas $\varphi_1, \ldots, \varphi_p$, which we refer to as the current list of *constraints*,
- 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, and the signature are empty, and the SyGuS logic is the default one (for details on the default SyGuS logic, see Section 5).

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 SyGuS 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` S)

  This sets the SyGuS background logic to the one that S refers to. The logic string S can be a standard one defined in SMT-LIB [3] or may be solver-specific. If S 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 S is an SMT-LIB standard logic, then this command sets the SyGuS logic to the one whose input and output logics are `QF_`S and whose feature set is the default one defined in this document (Section 5.2). In other words, when this command has S as an argument and S is an SMT-LIB standard logic, this indicates that that terms in the logic of S 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 S 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 by this document 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 is already a symbol in the current signature.

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.

[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.

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

  This is syntax sugar for (`synth-fun` S $((x_1\ \sigma_1)\ldots(x_n\ \sigma_n))$ `Bool` $G^?$).

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 7. For full details on well-formed datatype declarations, refer to Section 4.2.3 of the SMT-LIB version 2.6 [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 Defining Macros

- (`define-fun` S $((x_1\ \sigma_1)\ldots(x_n\ \sigma_n))$ 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.

3.7 Asserting Synthesis Constraints

- (`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.

- (`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 $\sigma_1 \times \ldots \times \sigma_n \to \mathsf{Bool}$,
  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. Given that 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.

3.8 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. The expected output from the synthesis solver is covered in Section 4.

3.9 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 well-formed response to `check-synth` is one of the following:

1. A list of commands of the form:

$$(\langle FunDefCmdKwd\rangle\ \ f_1\ X_1\ \sigma_1\ t_1)$$
$$\ldots$$
$$(\langle FunDefCmdKwd\rangle\ \ f_n\ X_n\ \sigma_n\ t_n)$$

   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 output, 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.

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:

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.

[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.

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 version 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. The logics mentioned in this section supplement the signature of the current state with additional sorts and symbols.

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 $\langle Numeral \rangle$ 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 $\langle Decimal \rangle$ 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 index-of `str.indexof` which returns the index of a string in another, and so on. A full description of this theory is given by the proposal in [5].

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

of 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 standard 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.

5.2 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 2.7. 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.

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.

5.3 SyGuS Logic Restrictions on Constraints

Let $\mathcal{L}$ be a SyGuS logic whose input logic is one from the SMT-LIB standard. 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 in the SMT-LIB standard.

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 in the SMT-LIB standard.

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 7.

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:

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 (1) it contains one `synth-inv` command, (2) it contains one `inv-constraint` command whose arguments are predicates whose definitions belong to logic X, and (3) it contains no other commands that introduce constraints, then its constraints are guaranteed to meet the requirements of the input logic `INV_`X.

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 *grouped rule listing* of the form

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

where $y_1, \dots, y_n$ are variables and $g_{11} \dots g_{1m_1}, \dots, g_{n1} \dots g_{nm_n}$ 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 $s_1, \dots, s_n$ 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 $(\lambda X_1.\, t_1, \dots, \lambda X_n.\, t_n)$ satisfies the syntactic restrictions for functions-to-synthesize $(f_1, \dots, f_n)$ in conjecture $\exists f_1, \dots, f_n.\, \varphi$ if $\lambda X_i.\, t_i$ satisfies the syntactic specification for f_i for each $i = 1, \dots, n$.

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.

A tuple of functions $(\lambda X_1.\, t_1, \dots, \lambda X_n.\, t_n)$ satisfies the semantic restrictions for functions-to-synthesize $(f_1, \dots, f_n)$ in conjecture $\exists f_1, \dots, f_n.\, \varphi$ in background theory T if φ is T-valid formula when $f_1, \dots, f_n$ are defined to be terms whose semantics are given by the functions $\lambda X_1.\, t_1, \dots, \lambda X_n.\, t_n$. The formal definition for T-valid here corresponds to the definition given by SMT-LIB, for details see Section 5 of [3].

The above definition covers cases where $f_1, \dots, f_n$ have recursive definitions or references to forward declarations. The formal definition of T-valid formulas that contain these kinds of function definitions assumes the semantics for SMT-LIB input formulas that involve (defined) macros and recursive functions.

7 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 List))
   ((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 solution to this synthesis conjecture 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-datatypes ((List 0)) ((nil) (cons (head Int) (tail (List Int)))))
(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 solution to this synthesis conjecture 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 solution for f 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 from the synthesis solver in this example 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 (Strings Programming by Examples (PBE))**.** Consider the following example:

```
(set-logic PBE_SLIA)
(synth-fun f ((fname String) (lname String)) String
    ((ntString String) (ntInt Int))
    ((ntString String (fname lname " "
                       (str.++ ntString ntString)
                       (str.replace ntString ntString ntString)
                       (str.at ntString ntInt)
                       (int.to.str ntString)
                       (str.substr ntString ntInt ntInt)))
     (ntInt Int (0 1 2
                 (+ ntInt ntInt)
                 (- ntInt ntInt)
                 (str.len ntString)
                 (str.to.int ntString)
                 (str.indexof ntString ntString ntInt)))))
(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 from the synthesis solver in this example is:

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

Example 6 (Invariant Synthesis (INV))**.** Consider the following example:

```
(set-logic INV_LIA)
(synth-inv inv-f ((x Int) (y Int)))
(define-fun pre-f ((x Int) (y Int)) Bool
  (and (>= x 5) (<= x 9) (>= y 1) (<= 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 $\texttt{inv} - \texttt{f}$, and only introduces constraints via a single `inv-constraint` command, it meets the restrictions of the logic. A possible correct response from the synthesis solver in this example is:

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

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

[5] 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`, `constraint`, `declare-datatype`, `declare-datatypes`, `declare-sort`, `declare-var`, `define-fun`, `define-sort`, `exists`, `forall`, `inv-constraint`, `let`, `set-feature`, `set-logic`, `set-option`, `synth-fun`, `synth-inv`, 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 that the grammar contains.

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_term Int) (y_cons Int) (y_pred Bool))
   ((y_term Int  (y_cons
                  (Variable Int)
                  (- y_term)
                  (+ y_term y_term)
                  (- y_term y_term)
                  (* y_cons y_term)
                  (* y_term y_cons)
                  (div y_term y_cons)
                  (mod y_term y_cons)
                  (abs y_term)
                  (ite y_pred y_term y_term)))
    (y_cons Int  ((Constant Int)))
    (y_pred Bool ((> y_term y_term)
                  (>= y_term y_term)
                  (< y_term y_term
                  (<= y_term y_term))))))
```

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_term Int) (y_pred Bool))
   ((y_term Int  ((Constant Real)
                  (Variable Int)
                  (- y_term)
                  (+ y_term y_term)
                  (- y_term y_term)
                  (* y_term y_term)
                  (div y_term y_term)
                  (mod y_term y_term)
                  (abs y_term)
                  (ite y_pred y_term y_term)))
    (y_pred Bool ((> y_term y_term)
                  (>= y_term y_term)
                  (< y_term y_term
                  (<= y_term y_term))))))
```

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_term Real) (y_cons Real) (y_pred Bool))
   ((y_term Real (y_cons
                  (Variable Real)
                  (- y_term)
                  (+ y_term y_term)
                  (- y_term y_term)
                  (* y_cons y_term)
                  (* y_term y_cons)
                  (ite y_pred y_term y_term)))
    (y_cons Real ((Constant Real)))
    (y_pred Bool ((> y_term y_term)
                  (>= y_term y_term)
                  (< y_term y_term
                  (<= y_term y_term))))))
```

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 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_term Real) (y_cons Real) (y_pred Bool))
   ((y_term Real ((Constant Real)
                  (Variable Real)
                  (- y_term)
                  (+ y_term y_term)
                  (- y_term y_term)
```

```
                (* y_term y_term)
                (/ y_term y_term)
                (ite y_pred y_term y_term)))
    (y_pred Bool ((= y_term y_term)
                (> y_term y_term)
                (>= y_term y_term)
                (< y_term y_term
                (<= y_term y_term))))))
```

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 n m`), 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`, xor `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 7.

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

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

B.4 Strings

Notice that there is no officially adopted SMT-LIB standard for strings at the time this document was written. Thus, we adopt (a subset of) the proposed standard [5] as of the time of the 2019 edition of the SyGuS competition. Here, we omit discussion of regular expressions and characters.

```
(set-logic S)
(synth-fun f ((xs String) (xi Int)) String
   ((y_term_str String) (y_term_int String) (y_pred Bool))
   ((y_term_str String ((Constant String)
                        (Variable String)
                        (str.++ y_term_str y_term_str)
                        (str.at y_term_str y_term_str)
                        (str.substr y_term_str y_term_int y_term_int)
                        (str.indexof y_term_str y_term_str y_term_int)
                        (str.replace y_term_str y_term_str y_term_str)
```

```
                    (str.from-int y_term_int)
                    (ite y_pred y_term_str y_term_str)))
  ((y_term_int String ((Variable Int)
                    (str.len y_term_str)
                    (str.to-int y_term_str)
                    (ite y_pred y_term_int y_term_int)))
   (y_pred Bool ((str.contains y_term_str y_term_str)
                 (str.prefixof y_term_str y_term_str)
                 (str.suffixof y_term_str y_term_str)
                 (str.< y_term_str y_term_str)
                 (str.<= y_term_str y_term_str))))))
```

String constants may be specified by text delimited by double quotes based on the syntax $\langle StringConst \rangle$ described in Section 2.2.

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.[4] An example, which computes the length of a (non-parametric) list is given below.

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

```
(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.

Attribute 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 $(!\ t\ A^*)$ where t is a term and A is an attribute. The syntax for attributes (not given here) is similar to the one for specifying options and features in this document. The term above is semantically equivalent to t itself. Several attributes are standardized by the SMT-LIB standard, while others may be user-defined.

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`.