SemGuS-Lib Format 1.0

Jinwoo Kim March 28, 2022

This document defines the current version of the SemGuS format, which is intended to be the standard input and output format for solvers aiming to solve SemGuS problems. The current version of the SemGuS format is designed such that existing solvers supporting the SyGuS format [1] or the SMT-LIB standard will be able to parse SemGuS problems with minimal overhead. As such, the SemGuS format borrows many concepts and language constructs from SyGuS and SMT-LIB [2]. In particular, where the syntax and semantics of constructs are identical, we will refrain from copying the whole definition and instead refer readers to the corresponding sections of the SyGuS and SMT-LIB standard documents, partly in order to keep in sync with updates to the other formats.

This document is currently incomplete; specifically, it is missing a section that details the semantics of each command that we will describe in this document. This will be added in a future version, along with example SemGuS problems.

An instance of a SemGuS problem consists of the following:

1. A base background theory T,
2. A finite set of *term types* that dictate a universe of terms to be used in the synthesis problem, specified as a regular tree grammar (RTG) R
3. *Semantics* for each constructor within the term-type declaration, specified using Constrained Horn Clauses $c_1, c_2, \cdots, c_i$
4. A set of functions $f_1, f_2, \cdots, f_j$ to synthesize,
5. Syntactic constraints for $f_1, f_2, \cdots, f_j$, also specified as a RTG G that defines a *subset* of terms inside the universe of terms defined by R
6. A semantic constraint ϕ that act as the behavioral specification of the set of functions to synthesize, which may be specified over the CHCs $c_1, c_2, \cdots, c_i$, a set of (implicitly) universally quantified variables $x_1, x_2, \cdots, x_k$, and the functions $f_1, f_2, \cdots, f_j$ themselves.

The goal of a SemGuS problem is to find a term $e \in G$ such that $\phi[e/f]$ is true.

1 Syntax

We now introduce the syntax for specifying SemGuS problems. A SemGuS problem $\langle Semgus \rangle$ is defined as a list of commands $\langle Cmd \rangle$s:

$$\langle Semgus \rangle ::= \langle Cmd \rangle^*$$

We now introduce the possible cases that a command $\langle Cmd \rangle$ may take first; then we introduce any required sub-expressions.

1.1 Comments

Comments in SemGuS, like SyGuS and SMT-LIB, are indicated by a leading semicolon. Upon encountering a semicolon, the rest of the input upto the next newline character should be ignored.

1.2 Metadata

SemGuS supports recording metadata related to SemGuS problems (such as problem authors, creation date, expected answers, etc.) directly as part of its syntax. Metadata commands are specified via the following syntax:

$$\langle Semgus \rangle ::= (metadata : \langle Attribute \rangle)$$

An $\langle Attribute \rangle$ is either a single keyword or a keyword paired with a value appropriate for the attribute (see Section 1.8).

The mechanism for specifying metadata coincides with the definition for annotations and term attributes in the SMT-LIB Standard, except for the fact that a metadata command should not have a leading ! symbol (see Section 3.4, 3.5 of the SMT-LIB Standard, Version 2.6).

1.3 Term Type Declarations

Term-type declarations define a universe of possible terms for use in the synthesis problem (that may be further restricted by a separate grammar later on). For those coming from SyGuS, it is convenient to think of the term-type declaration a declaring a *background theory of terms* that one may later choose to further restrict when solving a particular synthesis problem, similar to how SyGuS problems allow one to restrict the considered terms within a background theory such as LIA.

Term-types are declared via the following syntax:

$$(declare\text{-}term\text{-}types\ (\langle SortDecl \rangle^{n+1})\ (\langle DTDecl \rangle^{n+1}))$$

This syntax (including the definitions for subterms $\langle SortDecl \rangle$ and $\langle DTDecl \rangle$) is identical to the datatype declaration command *declare-datatypes* within SMT-LIB and SyGuS (see Section 2.9 of the SyGuS language standard, Version 2.1, and Section 4.2.3 of the SMT-LIB Standard) except for the fact that the heading command is named *declare-term-types*.

1.4 Semantic Declarations

SemGuS requires one to explicitly state the semantics for terms within the considered universe defined in §1.3; combined with the term-type declarations, one may consider these as the (custom-defined) background theory to operate over.

Semantics for terms defined in §1.3 is done using the *define-funs-rec* command, as following:

$$(\textit{define-funs-rec}\ (\langle \textit{Function\_Dec} \rangle^{n+1})\ (\langle \textit{Term} \rangle^{n+1}))$$

This syntax (including the definitions for subterms $\langle \textit{Function\_Dec} \rangle$ and $\langle \textit{Term} \rangle$) is identical to the command for declaring mutually recursive functions in the SMT-LIB standard (see Section 4.2.3 of the SMT-LIB standard, Version 2.6).

Although the syntax for specifying semantics in SemGuS files is permissive, SemGuS requires that semantics are defined using semantic relations and CHCs. In addition, SemGuS requires that these semantics be defined on an production-by-production basis—thus in practice, the following restrictions should be enforced by a SemGuS solver:

1. The return types of all functions declared in *declare-funs-rec* should i) contain an argument t of a term-type defined in §1.3, and ii) return a value of type *Bool*.
2. The body part $\langle \textit{Term} \rangle$ should be a *match* statement that matches t over the possible constructs defined in §1.3.

In addition, SemGuS allows multiple $\langle \textit{Term} \rangle$s to be associated with a single $\langle \textit{Pattern} \rangle$ in pattern matching. This allows one to equip multiple CHCs (for cases like while loops, or nondeterminstic programs) to a single constructor. Thus the syntax for $\langle \textit{match-case} \rangle$ (taken from the SMT-LIB standard, used for defining match statements) is changed to:

$$\langle \textit{match-case} \rangle ::= (\langle \textit{Pattern} \rangle\ \langle \textit{Term} \rangle^{+})$$

All other subexpressions are identical to the SMT-LIB standard.

1.5 Synth-Fun

The *synth-fun* command declares a single actual function to synthesize. The syntax is identical to the synth-fun command in the SyGuS standard; refer to Section 2.9 of the Sygus format, Version 2.1 for details.

To synthesize multiple functions, one should have multiple *synth-fun* commands in the file, one for each function to synthesize.

1.6 Constraints

The *constraint* command declares the behavioral specification for the functions to synthesize defined using *synth-fun*. The syntax once again borrows from the SyGuS standard; refer to Section 2.9 of the Sygus standard, Version 2.1 for details.

1.7 Other Commands

SemGuS accepts other standard SMT-LIB and SyGuS commands such as variable or sort declarations. All accepted other commands coincide with the syntax in SyGuS and SMT-LIB; a full list of other accepted commands can be found in §1.9.

To keep in sync with updates to SyGuS, we omit a hardcoded syntax definition for most constructs in this part, and instead refer the reader to the corresponding sections within the SyGuS format specification.

1.8 Subexpressions

Most subexpression definitions used in SemGuS (such as $\langle Term \rangle$s, $\langle Attribute \rangle$s, etc.) are identical to their definitions in SyGuS and SMT-LIB. We give a quick overview here.

1.8.1 Literals

Literals are special sequences of characters that are mostly used to denote values and 0-ary symbols of a background theory. The definition for literals in SemGuS is identical to that in SyGuS; for further information, consult Section 2.2 of the SyGuS standard, Version 2.1, or Section 3.1 of the SMT-LIB standard, Version 2.6.

1.8.2 Symbols and Identifiers

A symbol is a non-empty sequence of alphabets, digits, and certain special characters, that may not begin with a digit and is not a reserved word. An identifier is an extension of symbols to symbols that are indexed by integer constants or other symbols. The syntax of symbols and identifiers in SemGuS is identical to SyGuS and SMT-LIB; we refer the reader to Section 2.3 and 2.4 of the SyGuS standard, Version 2.1, for further information.

1.8.3 Attributes

An attribute is either a keyword $\langle Keyword \rangle$ or a keyword with an associated value. Attributes are used to annotate terms, as well as provide metadata in the metadata command. The syntax for keywords and attributes is identical to that in SyGuS; we refer the reader to Section 2.5 of the SyGuS standard for further information. An $\langle AttributeValue \rangle$ depends on the $\langle Keyword \rangle$ it is associated with. It is up to the

solver to support combinations of keywords and attribute values; the SemGuS standard does not provide a pre-defined list of keywords that must be supported.

1.8.4 Terms

Terms $\langle Term \rangle$ are used to specify grammars, constraints, semantics, and many other things. They use the same syntax as in SyGuS; we refer the reader to Section 2.7 of the SyGuS standard for further information.

Note that any term may be annotated with an attribute via the ! $\langle Term \rangle$ $\langle Attribute \rangle^+$ syntax.

1.9 Command Syntax

We now give an incomplete list of commands that make up the SemGuS format, focusing on a basic list of commands that we expect to be central to specifying a SemGuS problem. In addition to these commands, SemGuS supports all commands that SyGuS supports with the provision that semantics of some of these commands may be different; in essence, SemGuS diverges only from the SyGuS format through the addition of the *declare-term-types* command, with some additional semantic restrictions.

Some of the productions listed here contain nonterminals (e.g., $\langle Function\_Dec \rangle$) that are not defined in this document; for the concrete definition of these productions, we refer the reader to Section 2.9 of the Sygus specification, Version 2.1.

$$
\begin{array}{lcl}
\langle Cmd \rangle & ::= & (\textit{check-synth}) \\
& | & (\textit{constraint } \langle Term \rangle) \\
& | & (\textit{declare-term-types } (\langle SortDecl \rangle^{n+1}) \ (\langle DTDecl \rangle^{n+1})) \\
& | & (\textit{synth-fun } \langle Symbol \rangle \ (\langle SortedVar \rangle^*) \ \langle Sort \rangle \ \langle GrammarDef \rangle^?) \\
& | & \langle SmtCmd \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle SmtCmd \rangle & ::= & (\textit{declare-var } \langle Symbol \rangle \ \langle Sort \rangle) \\
& | & (\textit{declare-datatype } \langle Symbol \rangle \ \langle Sort \rangle) \\
& | & (\textit{declare-datatypes } (\langle SortDecl \rangle^{n+1}) \ (\langle DTDecl \rangle^{n+1})) \\
& | & (\textit{declare-sort } \langle Symbol \rangle \ \langle Numeral \rangle) \\
& | & (\textit{define-fun } \langle Symbol \rangle \ (\langle SortedVar \rangle^*) \ \langle Sort \rangle \ \langle Term \rangle) \\
& | & (\textit{define-sort } \langle Symbol \rangle \ \langle Sort \rangle) \\
& | & (\textit{define-funs-rec } (\langle Function\_Dec \rangle^{n+1}) \ (\langle Term \rangle^{n+1})) \\
& | & (\textit{set-info } \langle Keyword \rangle \ \langle Literal \rangle) \\
& | & (\textit{set-logic } \langle Symbol \rangle) \\
& | & (\textit{set-option } \langle Keyword \rangle \ \langle Literal \rangle)
\end{array}
$$

2 Semantics

A SemGuS problem is ultimately a series of commands that describes the components of a SemGuS problem: a grammar, a semantics that complements the grammar, and a specification.

Definition 1. A SemGuS problem over a background theory $\mathcal{T}$ consists of a tuple $(G_{[\![\cdot]\!]}, \forall x.\psi(x, f(x)))$, where G is a regular tree grammar equipped the CHC-based semantics $[\![\cdot]\!]$, and $\forall x.\psi(x, f(x))$ is a Boolean formula over $\mathcal{T}$ that specifies the desired behavior of the second-order variable f. When synthesizing j functions at once, this is extended such that there are j pairs of RTGs and specifications $(G_{1[\![\cdot]\!]_1}, \forall x.\psi_1(x, f_1(x)),$ $\cdots$ $(G_{j[\![\cdot]\!]_j}, \forall x.\psi_j(x, f_n(x)))$.

A SemGuS problem is *realizable* if $\exists s \in L(G)$, such that $\forall x.\psi(x, [\![s]\!](x))$ holds. If such $s \in L(G)$ does not exist, then the problem is *unrealizable*.

We briefly discuss how the commands described in §2 establish the components of a SemGuS problem.

2.1 Defining the Syntax

The syntax of a SemGuS problem is defined in two steps—the *declare-term-types* command first declares a background universe of terms we are willing to consider within a SemGuS problem, which may be further syntactically restricted for each function by a grammar within a *synth-fun* block. We mainly consider defining the background universe of terms through *declare-term-types* here, and discuss further restricting the grammar for individual functions in §2.3.

declare-term-types is syntactically identical to the *declare-datatypes* command in SMT-LIB and SyGuS, and is defined using the following syntax:

$$(\texttt{declare-term-types}\ ((\delta_1\ k_1)\cdots(\delta_n\ k_n))\ (d_1\cdots d_n))$$

In a SemGuS file, this command declares an regular tree grammar where each symbol $\delta_1 \cdots \delta_n$ represent the nonterminals of the RTG. The arity of the nonterminals k_i is fixed to 0 for each δ_i.[1] Each d_i in $(d_1 \cdots d_n)$ is a list of constructors $(c_1 \cdots c_m)$, where each c_j is, intuitively, a constructor that acts the right-hand side of a production. The RTG that *declare-term-types* defines consists of the nonterminals $\delta_1, \cdots, \delta_n$, and the set of productions constructed as following: for each $1 \geqslant i \geqslant n$, given the nonterminal δ_i, with its corresponding list of productions $d_i = (c_1, \cdots c_m)$, for each $1 \geqslant j \geqslant m$, the production $\delta_i \to c_j$ is added to the set of productions being considered. This RTG designates the background grammar considered for a particular SemGuS problem.

If there are multiple *declare-term-types* within the constructor, the background universe is considered to be defined by the RTG obtained by taking the union of each RTG within *declare-term-types*. If, for whatever reason, *declare-term-types* fails to declare a valid RTG, then the SemGuS file is ill-formed.

[1] Retaining the k_is in the syntax, even though they are fixed to 0 and can be easily inferred, is a design choice to maintain compatibility with existing SMT-LIB and SyGuS parsers.

Example 1. Consider a simple RTG of the form $E \to 1 \mid x \mid E + E$. This RTG can be declared in a SemGuS file using the following command:

```
1 (declare-term-types
2   ((E 0))
3   (
4     ($1)
5     ($x)
6     ($+ (e1 E) (e2 E))
7   )
8 )
```

On line 2, (`E0`) declares that we are declaring a single nonterminal E. On lines 4 to 6, the constructors (RHSes) of the nonterminal E are declared: in particular, note line 6 where the RHS uses two subexpressions $e1$ and $e2$ of the nonterminal E. The preceding \$ symbols on the operators serve to distinguish between operators in the RTG, and symbols reserved in SyGuS and SMT-LIB (other symbol names can be used as well as long as there is no overlap between SMT-LIB).

The reason for the two-step specification, where we define a background grammar first, is because we expect many synthesis problems to share a universe of terms, but differ in their specific syntactic restrictions: for examples, all LIA SyGuS problems can be understood as sharing the background LIA grammar, which can be further restricted syntactically through another grammar (defined within the *synth-fun* block) depending on the problem. Keeping a separation between the universe (i.e., the "background theory") and syntactic restrictions (i.e., the "grammar") facilitates reuse, especially as semantics for operators are defined at the background-level as opposed to separately for each grammar.

2.2 Defining the Semantics

A SemGuS file must define a semantics for each production in the RTG defined by *declare-term-types* as a CHC, using semantic relations. For further details on how to define a big-step semantics using relations and CHCs, we refer readers to the SemGuS paper [3]. Here, we assume that the user is familiar with defining semantics in this way, and proceed to show how this information is expressed in a SemGuS file through the *define-funs-rec* command.

$$(\texttt{define-funs-rec}\ (d_1 \cdots d_n)\ (t_1 \cdots t_n))$$

Using the syntax above, *define-funs-rec* defines a set of n functions that may be mutually recursive, where the name and signature of the i-th function is defined in the i-th declaration d_i. The function body of the function declared in d_i is defined in t_i, which is an expression of the return type of the i-th function.

In SemGuS, the main role of this command is to define the semantic relation for each nonterminal. Because SemGuS requires that semantics be expressed as CHCs using semantic relations, the return type

for all functions should be Boolean. Additionally, the function body t_i must be a SMT-LIB *match* statement (`match` s $(c_1 \cdots c_m)$) where s is a variable with the sort of a nonterminal N declared in the universal syntax, and $c_1 \cdots c_m$ matches over the possible RHS productions for N.

Example 2. Following Example 1, let us see how the RTG $E \to 1 \mid x \mid E + E$ is equipped with a semantics using the following command:

```
1  (define-funs-rec (
2    (sem-E ((et E) (x Int) (r Int)) Bool)
3    (match et
4      ($1 (= r 1))
5      ($x (= r x))
6      (($+ et1 et2)
7       (exists ((r1 Int) (r2 Int))
8         (and
9           (sem-E et1 x r1)
10          (sem-E et2 x r2)
11          (= r (+ r1 r2))
12 )))))
```

In line 2, we declare the *semantic relation* for the nonterminal E as a function $sem_E : E \times Int \times Int \to Bool$. In lines 4-6, the `match` statement lists each of the possible productions of E, and for each production, details the condition for which the relation sem_E should evaluate to True.

Logically interpreted, the three case statements on lines 4 to 11 each act as the premise of a CHC that describe the semantics of the three productions of E. The function declaration on line 2 acts as the conclusion of the CHC. For example, this means that the production $E \to E + E$ is equipped with the following CHC as semantics:[2]

$$sem_E(et1, x, r1) \wedge sem_E(et2, x, r2) \wedge r = r1 + r2 \implies sem_E(+(et1, et2), x, r)$$

It is possible to define multiple sets of semantics for a single universe of terms; for example, one may wish to equip a grammar with both concrete and abstract semantics. These may all be defined in a single *define-funs-rec* block, or one may choose to use separate *define-funs-rec* blocks for each set of semantics (similarly, it is also possible to divide a *single* set of semantics into multiple *define-funs-rec* blocks if the semantic relations need not be mutually recursive). Regardless of the manner chosen, all functions defined within a *define-funs-rec* block will be available for use in other parts of the file (e.g., when defining the behavioral specification using *constraint*).

[2]The quantifier is left out as all variables in CHCs are implicitly universally quantified.

2.3 Declaring Functions to Synthesize

Each function to synthesize inside a SemGuS file is defined separately using its own *synth-fun* command, which follows the same syntax as SyGuS. As in SyGuS, a *synth-fun* block may contain an optional grammar block. In SemGuS, this grammar must express an RTG that is a subgrammar of the background grammar declared using *declare-term-types*; this serves as the actual syntactic restriction of the function to synthesize. Because the semantics for all terms in the background grammar have already been declared, there is no need to state a separate set of semantics within the *synth-fun* block.

The use of the arguments in *synth-fun* is different in SemGuS compared to SyGuS, although the syntax is the same. In SemGuS, the second element (a list of ⟨*SortedVar*⟩s; i.e., the list of input arguments) must be an empty list[3], while the third element (a ⟨*Sort*⟩; i.e., the return type) must indicate the term-type of the term to be synthesized.[4]

Example 3. Following Examples 1 and 2, we now declare a function to synthesize in this background grammar. In particular, we would like to further restrict the grammar such that a maximum of one + operation is allowed; this can be expressed using the following command.

```
1  (synth-fun max2 () E
2    ((Start E) (A E)) (
3    (Start E (
4      A
5      ($+ A A)
6    ))
7    (A E (
8      $1
9      $x
10   ))
11 ))
```

In line 1, we declare that we wish to synthesize a term of the name *max2*, which is of the term-type E (that is, it should be a term in the language of the nonterminal E). Starting from line 2, we are declaring a new grammar that further restricts the search space allowed for *max2*.

Line 2 declares the nonterminals of this sub-grammar: a starting nonterminal *Start*, and another nonterminal for atoms A, both of which still produce terms in the language of the (background) nonterminal E (that is, $L(\mathit{Start}) \subseteq L(E)$ and $L(A) \subseteq L(E)$). Starting from line 3, the productions of the sub-grammar are listed—that $\mathit{Start} \to A \mid A + A$, and $A \to 1 \mid x$.

Given a *synth-fun* block defining a sub-grammar G_{sub} this way, logically, a SemGuS problem is defined

[3]Again, this is a design choice to maintain compatibility with existing SyGuS parsers.

[4]This is because in SemGuS, one is more synthesizing a term which can be interpreted via the semantic relations, as opposed to a function in a traditional sense.

using G_{sub} as the grammar, the background semantics $[\![\cdot]\!]$ defined in Example 2, and constraints that will be defined further down in the file. Because $L(Start) \subseteq L(E)$ and $L(A) \subseteq L(E)$, the semantics of all terms within $L(Start)$ and $L(A)$ may be interpreted using the (background) semantic relation sem_E defined using *define-funs-rec*—thus there is no need to separately declare semantics for subgrammars.

2.4 Declaring Universal Variables and Sorts

Like SyGuS, SemGuS allows one to define additional auxiliary variables and sorts through the *declare-var*, *declare-datatypes*, and *declare-sort* command. Like SyGuS, all variables defined in this manner will be universally quantified when appearing in constraints; we refer the reader to Section 3 of the SyGuS standard for further details.

2.5 Behavioral Specifications

The behavioral specification for each function f_i to synthesize is given by the series of *constraint* commands within the SemGuS file. In particular, most of the time one will directly use the functions defined using *define-funs-rec* to express that a synthesized term satisfies some constraint.

Example 4. Continuing with Examples 1, 2, and 3, we give some sample constraints one may use for `max2`. If the specification was given inductively via examples, one may have constraints such as following, using two examples:

```
1 (constraint sem-E max2 4 2 4)
2 (constraint sem-E max2 2 4 4)
```

On the other hand, a universal specification using variables could also be expressed:

```
1 (declare-var x Int)
2 (declare-var y Int)
3 (declare-var r Int)
4 (constraint (and (sem-E x y x) (= (ite (> x y) x y) r)))
```

Given multiple *constraint* commands, a conjunction of the constraints is took as the specification for the target functions. *constraint* only allows Boolean expressions; SemGuS files that contain *constraint* commands with non-Boolean expressions are ill-formed.

References

[1] Saswat Padhi, Elizabeth Polgreen, Mukund Raghothaman, Andrew Reynolds, and Abhishek Udupa. The SyGuS languages standard version 2.1. Technical report, 2021. Available at `sygus.org`.

[2] 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`.

[3] Jinwoo Kim, Qinheping Hu, Loris D'Antoni, and Thomas Reps. Semantics-guided synthesis. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–32, 2021.