SemGuS-Lib Format 1.0

Jinwoo Kim July 21, 2021

This document details a preliminary format for how problems should be specified in the SemGuS format. This document assumes SMT-LIB and SyGuS-lib as backgrounds for defining the syntax of SemGuS. This document is intended to be a preliminary version for development.

1 The Grammar for a SemGuS Problem

A SemGuS problem is defined using the following formal grammar, where everything inside brackets <> is further expanded:

```
List[<Metadata> | <Declarations>]
(synth-term <Term-Name> <Type-Name>
  (List[(<NTDeclaration> | <VarDeclaration>)]
   List[<LHS-Production-Set>]
))
List[<Metadata> | < Declarations> | <Constraint>]
```

Each subproduction is explained in detail below.

```
<Metadata> ::= (metadata :<Tag> <S-Expression>)
```

`Metadata` serves as metadata for a semgus problem, e.g., authors, benchmark names, expected results, etc. We choose to introduce this feature because this sort of data is typically much more well tracked when written as an actual part of a Semgus problem (as opposed to annotated with comments, etc.).

One should think of each metadata statement as a simple key-value pair. `<Tag>`, which is a simple string, acts as the key; the following `<S-Expression>` is an arbitrary s-expression defining the value for the key.

Solvers, in general, will choose to ignore metadata information; it is also possible for them to collect some information (such as benchmark names) when running benchmark suites for easier catalogging.

```
<Declarations> ::= SMT-LIB2 Declaration | <TermDeclaration>
<VarDeclaration> ::= SMT-LIB2 Variable Declaration


<TermDeclaration> ::= (declare-term-type <Type-Name>)
```

`<Declaration>` refers to an arbitrary SMT-LIB2 declarations (e.g., variable declarations, sort declarations, etc), or `<TermDeclaration>`s. `<VarDeclarations>` refers specifically to variable declarations (`(declare-var ...)`). Semgus requires that all names used in declarations are unique—that is, one cannot have a sort named `X` and a variable named `X`, or two variables named `X`, and so on.

A `<TermDeclaration>` is a feature specific to SemGuS, which can be intuitively though of as declaring a 'type' for syntactic representations of synthesized terms (term-types). The intuition behind term-types is that terms generated from different nonterminals from a grammar may still have the same type—for example, in the two-production grammar $E \to A \mid E + 1; A \to 0 \mid x$, both E and A can only generate terms that are integers. Term-types capture this idea to structure the syntactic representation of synthesized terms. `<Type-Name>` is a simple `String` that serves as a name for the term-type. Note that one should (and can) not use pre-defined types such as `Int` for `<Type-Name>`: `<Term-Type>` intends to capture the fact that the generated terms are syntactic representations of *expressions* of type `Int`; thus one should use a type like `IExp` instead.

Notice that `<VarDeclarations>` are allowed within the `synth-term` block (which specifies the synthesis problem in SemGuS). Variables declared within the `synth-term` block will be local to the block, that is, one cannot use the variable `X` in places like `<Constraint>` if `X` has been declared within `synth-term`. In addition, to use a variable within a CHC premise or conclusion inside the `synth-term` block, it must be declared within the `synth-term` block. This restriction is because variables used inside a CHC have effectively 'local' behavior in the sense that they are captured by the universal quantifiers of their resident CHCs; the restriction that CHC variables must be declared within the `synth-term` block highlights this restriction.

Naming restrictions still apply to variables declared within the block. In addition, SemGuS requires that variable names only start with alphabetical characters (this restriction is not present in SMT-LIB2).

In addition to standard SMT-LIB2 declarations, Semgus also allows one to define multiple variables of the same sort at once using the following syntax:

```
(declare-var (x y ... z) Int)
```

This is syntactically expanded to `(declare-var x Int) (declare-var y Int) ... (declare-var z Int)`.

```
(synth-term <Term-Name> (...))
```

The `synth-term` block defines a grammar and its semantics (defined within the `...` part) upon which to define a synthesis problem to synthesize the term `<Term-Name>`. `<Term-Name>` is a simple string that defines the name of the term to be synthesized. The `<Type-Name>` at the start of the `synth-term` specifies the term-type of the term to be synthesized. For example, when one is trying to synthesize an expression, `<Type-Name>` might be `IExp`; if one is trying to synthesize a statement, it might be something like `Stmt` (both of which would have to be defined previously via `<TermDeclaration>`).

SemGuS currently only allows the definition of *one* `synth-term` block per file; that is, one cannot write a problem that requires synthesizing more than one function at once.

SemGuS accepts regular tree grammars (RTGs), which are grammars that contain a single nonterminal on the LHS and either a leaf node or an operator with children (which are each leaf nodes or nonterminals) on the RHS.

```
<NTDeclaration> ::= (declare-nt <NT-Name> <Type-Name> <NT-RelationDec>)
<NT-RelationDec> ::= (<Rel-Name> List[Sort])

<NT-Name>: Nonterminal name, String
<Type-Name>: Term-type for terms generated by <NT-Name>
<NT-RelationDec>: The semantic relation declaration for the nonterminal <NT-Name>
```

`<NTDeclaration>` defines each nonterminal to be used in the target RTG, and pairs it with a term-type and a semantic relation for the nonterminal. It is assumed that every term $t \in L(\langle NT\text{-}Name \rangle)$ has the term-type `<Type-Name>` (which must be previously defined as a term-type above using `<TermDeclaration>`).

`<NT-RelationDec>` declares the signature for the semantic relation of the nonterminal `<NT-Name>`. The declaration syntax itself is identical to an SMT-LIB2 relation declaration declaring a relation with the name `<Rel-Name>`, sans the leading `declare-rel`. These relations are globally visible, from anywhere in the file. At the bare minimum, a semantic relation should accept a term of term-type `<Type-Name>` (in order to keep check whether a term is syntactically well-formed); in addition, it should take any number of arguments required to encode the semantics of terms in `<NT-Name>`.

An RTG is expressed within the `synth-term` block as a list of `<LHS-Production-Set>`s, which group all the productions in the grammar according to which LHS they have.

```
<Production> ::= (<LHS> List[<RHS>])
```

A single `<LHS-Production-Set>` contains a single `<LHS>` and a list of `<RHS>`es: each `<RHS>` in the list creates a production `<LHS> -> <RHS>` for the grammar.

In SemGuS, semantics of operators and grammars are given as Constrained Horn Clauses (CHCs) that capture the semantics of each production. To capture this information, the `<LHS>`es and `<RHS>`es of a SemGuS problem are defined in a richer syntax then that compared to one of a SyGuS problem.

```
<LHS> ::= (<NT-Name> <NT-Term-Var>) <NT-Relation>

<NT-Name>: String, the name of the LHS nonterminal
<NT-Term-Var>: A variable of the term-type for <NT-Name>
<NT-Relation>: An application of the relation declared in <NT-RelationDec>
```

An `<LHS>` is composed of four components: `<NT-Name>`, `<NT-Term-Var>`, `<NT-RelationDec>`, and `<NT-Relation>`

- `<NT-Name>` represents the name of the nonterminal on the LHS, and is a String (e.g., 'Start'). Again, `<NT-Name>` must previously be defined as a nonterminal using `<NTDeclaration>`.
- `<NT-Term-Var>` states the variable that represents the term *being created* by the production. Must be of sort `Term`, where `Term` is a reserved and pre-defined sort in SemGuS to represent syntactic terms.

- `NT-Relation` is an SMT-LIB2 relation application, that is to be used as the *conclusion* of the CHC associated with productions that have `<NT-Name>` as their LHS nonterminal.

A single `<RHS>` is composed of similar elements: the RHS in question, in addition to the premises to use for the CHC.

```
<RHS> ::= (<RHS-Exp> List[<Premise>])
<RHS-Exp> ::= (<OP-Name> List[<RHS-Atom>]) | <RHS-Atom>
<RHS-Atom> ::= (<NT-Name> <NT-Term-Var>) | <Leaf-Name>

<Premise> ::= A Boolean SMT-LIB2 formula.
<OP-Name> ::= String, the name of the operator on the RHS
<Leaf-Name> ::= String, the name of the leaf on the RHS
```

- `<RHS-Exp>` defines the actual syntactic structure of an RHS: either an operator `<OP-Name>` followed by some children atoms, or a leaf node on its own.
- `<OP-Name>` is the name of the operator used on the RHS (e.g., 'Plus').
- `<RHS-Atom>` is an atomic RHS element, which is either a nonterminal or a leaf node.
- For nonterminals, the syntax is `(<NT-Name> <NT-Term-Var>)`, where `<NT-Name>` and `<NT-Term-Var>` again respectively represent the nonterminal name and the term from the nonterminal. In this case, the term variables are combined with the operator *to create* the term variable specified on the LHS side. For example, if `t` was specified as the term variable for the LHS for an RHS `(Plus (E t1) (E t2))`, one would have that `t = Plus(t1, t2)`.
- For leaves, `<Leaf-Name>` encodes the name of the leaf node to use (e.g. 'Zero').
- `<Premise>`: A boolean SMT-LIB2 formula that serves as the premise of the CHC associated with the production `<LHS> -> <RHS>`. One should use this and `<NT-Relation>` from the LHS side to construct a CHC that captures the *semantics* of the production.[1]

  Multiple premises may be associated with a single production, which is why the grammar expects a list of `<Premise>`s. This functionality is useful when expressing constructs such as loops or nondeterministic semantics.

All names declared in the productions—that is, `<NT-Name>`, `<Rel-Name>`, `<OP-Name>`, and `<Leaf-Name>`—are subject to the same rule that their names may not overlap with anything else declared in the file. For example, one may not name the constant leaf node `1` directly as `1` in `<LEAF-NAME>`, because it overlaps with the constant 1—one must rename this to something like `One` or `1.Syn` (the syntactic representation of `1`).

[1] One need not encode the syntactic constraint that, for example, `t = Plus(t1, t2)`.

This is required both at the SemGuS file level in order to eliminate ambiguities, and also implements the design principle of SemGuS that a syntax should be kept strictly separate from its semantics—by enforcing that one cannot use the same symbol for the syntactic and semantic manifestations of terms, this separation is is kept explicitly in-view.

In addition, `<OP-Name>` and `<Leaf-Name>` are allowed to start with leading zeros (or other numbers) in SemGuS. This is a difference from SMT-LIB2, which prohibits the use of leading zeros in most of its constructs.

Finally, the `<Constraint>` section defines the constraints that the synthesized term `<Term-Name>` should satisfy. It is defined like a standard SyGuS constraint:

```
<Constraint> ::= (constraint <Boolean SMT-LIB2 Formula>)
```

In SemGuS, one uses the semantic relations defined in the `synth-term` block directly, alongside term name `<Term-Name>`, to define the semantics that a term should satisfy: for example, `(constraint (E.Sem first x y x))` states that the term `first` (which should be defined as the name for a `synth-term` block) must satisfy the constraint `(E.Sem first x y x)`.

If there are multiple `<Constraint>`s, the conjunction of their contents is taken as the entire constraint to satisfy.

2 Semantics of LHS and RHS constructs

Consider as an example, a production $E1 \to \mathit{Plus}(E2, E2)$ and its related term-type and semantic relations defined in the following manner:

```
...
(declare-term-type Eexp)
(synth-term ...
(declare-nt E1 Eexp (E1.Sem Eexp Int Int Int))
(declare-nt E2 Eexp (E2.Sem Eexp Int Int Int))
((E1 t) (E1.Sem t x y r)
  ((Plus (E2 t1) (E2 t2))
   (and (E2.Sem t1 x y r1) (E2.Sem t2 x y r2) (= r (+ r1 r2)))
  )
)
```

This creates the following CHC:

$$E2.Sem(t1, x, y, r1) \wedge E2.Sem(t2, x, y, r2) \wedge t = \mathit{Plus}(t1, t2) \wedge r = r1 + r2 \implies E1.Sem(t, x, y, r)$$

That is, `<NT-Relation>` is directly used as the conclusion of the CHC; each `<Premise>` for an RHS is

augmented with a syntactic constraint for terms (in this case, $t = Plus(t1, t2)$) then used as the premise for the CHC.

The syntactic constraint is inferred automatically from the structure of the nonterminal. Given a production $L \rightarrow op(R_1, R_2, \cdots R_n)$, labeled with the term variables t (for the LHS) and $t_1, t_2, \cdots t_n$ (for the RHS), the constraint $t = op(t_1, t_2, \cdots, t_n)$, where op is automatically inferred to be a valid constructor that produces terms of the term-type for L, is added in the form of a conjunction to `<PREMISE>`. In terms of SMT-LIB2, one can understand term-types as recursive datatypes; and that each production creates a new constructor `op` for the recursive datatype `<Type-Name>` that takes as argument the term-types for the nonterminals $R_1, R_2, \cdots R_n$. If `op` is previously defined in another production, their signatures must match.