Language to Specify Syntax-Guided Synthesis Problems

Mukund Raghothaman        Abhishek Udupa

Friday 23rd May, 2014

Abstract

We present a language to specify syntax guided synthesis (SyGuS) problems. Syntax guidance is a prominent theme in contemporary program synthesis approaches, and SyGuS was first described in [1]. This paper describes concretely the input format of a SyGuS solver.

1 Introduction

We present a language to specify syntax guided synthesis (SyGuS) problems. Syntax guidance is a prominent theme in contemporary program synthesis approaches, and SyGuS was first described in [1]. An instance of a SyGuS problem has four parts:

1. A base vocabulary and theory, specifying the basic types, primitive operations over the types, and their properties,
2. a finite set of typed “synthesis” functions \( f_1, f_2, \ldots \), whose bodies are to be synthesized,
3. syntactic constraints: for each synthesis function \( f_i \), a grammar \( G_i \) describing the syntactic structure of the potential solutions, and
4. semantic constraints: a formula \( \varphi \), with some universally quantified variables \( v_1, v_2, \ldots \), which constrains the values of the synthesis functions.

The problem is to find expression bodies for each synthesis function \( f_i \) from the grammar \( G_i \) so that the constraint is universally satisfied:

\[
\forall v_1, v_2, \ldots, \varphi (f_1, f_2, \ldots, v_1, v_2, \ldots) .
\]

The constraint formula \( \varphi \) is quantifier-free, and the logical symbols and their interpretation in \( \varphi \) and the grammar are restricted to a background theory.

For example, over the theory of linear integer arithmetic, the functions computing the maximum \( max_2 \) and minimum \( min_2 \) of a pair of integers may be specified as

\[
\forall x, y : \mathbb{Z}, \quad max_2 (x, y) \geq x \land max_2 (x, y) \geq y \\
\land (max_2 (x, y) = x \lor max_2 (x, y) = y) \\
\land (max_2 (x, y) + min_2 (x, y) = x + y) .
\]

We are interested in piecewise linear functions, so the grammar \( G \) for both functions would be

\[
\begin{align*}
\text{Expr} & ::= 0 \mid 1 \mid x \mid y \\
& \mid \text{Expr} + \text{Expr} \\
& \mid \text{Expr} - \text{Expr} \\
& \mid (\text{ite BoolExpr Expr Expr}) \\
\text{BoolExpr} & ::= \text{BoolExpr} \land \text{BoolExpr} \\
& \mid \neg \text{BoolExpr} \\
& \mid \text{Expr} \leq \text{Expr}
\end{align*}
\]
2 Example SyGuS Specification

Before formally describing the language, we present a concrete example of a SyGuS specification.

We continue the example of $\text{max}_2$ and $\text{min}_2$ from the previous section, and present the corresponding SyGuS code in figure 1. The first command (set-logic LIA) informs the synthesizer to load symbols corresponding to linear integer arithmetic. Next, we describe the functions to be synthesized: the command (synth-fun max2 ...) command first specifies that $\text{max}_2$ is a function of two integer arguments $x$ and $y$, and returns an integer value. The rest of the command describes the grammar for $\text{max}_2$. Start and StartBool are integer-valued and boolean-valued non-terminal symbols respectively. Start is the special starting non-terminal of the grammar. The description of $\text{min}_2$ is identical to that of $\text{max}_2$, except for the function name, and some useful shorthands (Constant Int) and (Variable Int) which respectively expand to any integer constant and integer-valued variable currently in scope. Finally, the code lists the constraints that these functions satisfy. Pick a pair of integers $x$ and $y$. The first constraint requires that $\text{max}_2(x, y) \geq x$. The final synthesis constraint $\varphi$ is the conjunction of the constraints imposed by the individual constraint commands.

3 Specification Language

The SyGuS specification language is closely modeled on SMT-LIB2 [2]. A SyGuS input file is a sequence of commands; in subsections 3.2-3.11, we describe the syntax of each command. In the following description, italicized text within angle-brackets represents EBNF non-terminals, and text in typewriter font represents terminal symbols.

\[
\langle \text{SyGuS} \rangle ::= \langle \text{SetLogicCmd} \rangle \langle \text{Cmd} \rangle^+ \\
\langle \text{Cmd} \rangle ::= \langle \text{SortDefCmd} \rangle \\
\langle \text{VarDeclCmd} \rangle \\
\langle \text{FunDefCmd} \rangle \\
\langle \text{SynthFunCmd} \rangle \\
\langle \text{ConstraintCmd} \rangle \\
\langle \text{CheckSynthCmd} \rangle \\
\langle \text{SetOptsCmd} \rangle
\]

3.1 Language trivia

3.1.1 Reserved words

The following keywords are reserved, and may not be used as identifiers in any context: set-logic, define-sort, declare-var, declare-fun, define-fun, synth-fun, constraint, check-synth, set-options, BitVec, Array, Int, Bool, Enum, Real, Constant, Variable, InputVariable, LocalVariable, let, true, false.

3.1.2 Comments

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

3.1.3 Identifiers

Identifiers are denoted with the non-terminal \(\langle \text{Symbol} \rangle\). An identifier is any non-empty sequence of upper- and lower-case alphabets, digits, and certain special characters, with the restriction that it may not begin
(set-logic LIA)

(synth-fun max2 ((x Int) (y Int)) Int
  ((Start Int (0 1 x y
    (+ Start Start)
    (- Start Start)
    (ite StartBool Start Start)))

  (StartBool Bool ((and StartBool StartBool)
    (not StartBool)
    (<= Start Start))))

(synth-fun min2 ((x Int) (y Int)) Int
  ((Start Int ((Constant Int) (Variable Int)
    (+ Start Start)
    (- Start Start)
    (ite StartBool Start Start)))

  (StartBool Bool ((and StartBool StartBool)
    (not StartBool)
    (<= Start Start))))

(declare-var x Int)
(declare-var y Int)

(constraint (>= (max2 x y) x))
(constraint (>= (max2 x y) y))

(constraint (or (= x (max2 x y))
  (or (= y (max2 x y)))))

(constraint (= (+ (max2 x y) (min2 x y))
  (+ x y)))

(check-synth)

Figure 1: SyGuS specification for functions computing the maximum and minimum of two integers.
with a digit.

\[
\langle \text{SpecialChar} \rangle = \{\_, +, -, *, &, !, \!, <, >, =, /, \%, \?, \$, *\}
\]

\[
\langle \text{Symbol} \rangle ::= ( [a-z] \mid [A-Z] \mid \langle \text{SpecialChar} \rangle )
\]

\[
( [a-z] \mid [A-Z] \mid [0-9] \mid \langle \text{SpecialChar} \rangle )^*
\]

A quoted literal, \( \langle \text{QuotedLiteral} \rangle \) is a non-empty sequence of alphabets, digits and the period (.) enclosed within double-quotes.

\[
\langle \text{QuotedLiteral} \rangle ::= "( [a-z] \mid [A-Z] \mid [0-9] \mid . )^+"
\]

### 3.1.4 Literals

\[
\langle \text{Literal} \rangle ::= \langle \text{IntConst} \rangle \mid \langle \text{RealConst} \rangle \mid \langle \text{BoolConst} \rangle
\]

\[
\langle \text{IntConst} \rangle ::= [0-9]^+ \mid -[0-9]^+
\]

\[
\langle \text{RealConst} \rangle ::= [0-9]^+. [0-9]^+ \mid -[0-9]^+. [0-9]^+
\]

\[
\langle \text{BoolConst} \rangle ::= \text{true} \mid \text{false}
\]

\[
\langle \text{BVConst} \rangle ::= \#b[0-1]^+ \mid \#x( [0-9] \mid [a-f] \mid [A-F] )^+
\]

\[
\langle \text{EnumConst} \rangle ::= \langle \text{Symbol} \rangle :: \langle \text{Symbol} \rangle
\]

Integer constants are written as usual, in decimal, with an optional minus at the beginning to denote a negative number. Real numbers are written using their decimal expansion: at least one decimal digit before and after a mandatory period, and an optional minus sign at the beginning. \text{true} and \text{false} are the predefined boolean constants. Bit-vector constants may be written using either their traditional binary or hexadecimal representations. Enumerated constants are written in two parts: the first identifier names the sort the constant belongs to, and the second identifier names the constructor. The definition of enumerated sorts is described in subsection 3.3.

### 3.2 Declaring the problem logic

\( \langle \text{SetLogicCmd} \rangle \)

On encountering the optional \( \langle \text{SetLogicCmd} \rangle \), the synthesizer loads appropriate pre-defined function symbols and constants. Current theories include

1. \text{LIA}: Linear integer arithmetic, for functions such as \(+\) and \(-\),
2. \text{BV}: Theory of bit-vectors, for functions such as \text{bvadd} and \text{bvlshr},
3. \text{Reals}: Theory of real numbers, and
4. \text{Arrays}: Theory of arrays.

\[
\langle \text{SetLogicCmd} \rangle ::= \langle \text{set-logic} \langle \text{Symbol} \rangle \rangle
\]

### 3.3 Defining new sorts

\( \langle \text{SortDefCmd} \rangle, \langle \text{SortExpr} \rangle \)

\( \text{SyGuS} \) expects that the sorts of functions, variables, and grammar symbols be explicitly specified. The syntactic construct \( \langle \text{SortExpr} \rangle \) is used for this, and the sort definition command \( \langle \text{SortDefCmd} \rangle \) permits defining useful shorthands.
The sorts `Int`, `Bool`, and `Real` refer to integers, booleans and real numbers respectively. For each positive integer \( n \), \( \text{BitVec} \ n \) refers to the sort of bit-vectors \( n \) bits long. Given a set of constructor symbols \( S_1, S_2, \ldots \), the sort \( \text{Enum} \ (S_1 \ S_2 \ldots) \) refers to the enumerated type having those elements. Since the only way to represent an enumerated constant (subsection 3.1.4) is by also specifying the sort-name, the constructors \( S_1, S_2 \) etc. may have the same names as previously defined variables, functions, or sorts. The sort \( \text{Array} \ S_1 S_2 \) represents arrays that map elements of sort \( S_1 \) to elements of sort \( S_2 \).

Once a sort \( S \) has been defined using the command \( \text{define-sort} \ S \langle \text{SortExpr} \rangle \), it may subsequently be referred to simply as \( S \) rather than the full expression \( \langle \text{SortExpr} \rangle \). The identifier \( \langle \text{Symbol} \rangle \) used to name a sort should not have been previously used as a sort name. Every \( \langle \text{SortExpr} \rangle \) in a SyGuS specification must be well-formed. We say that a \( \langle \text{SortExpr} \rangle \) is well-formed if

1. it is an instance of `Int`, `Bool`, `Real`, `BitVec` or `Enum`, or
2. it is an instance of `Array` and both domain and range of the array sort are well-formed, or
3. it is a \( \langle \text{Symbol} \rangle \), and \( \langle \text{Symbol} \rangle \) has been previously defined using a \( \langle \text{SortDefCmd} \rangle \).

### 3.4 Universally quantified variables

Universally quantified variables may be declared with \( \langle \text{VarDeclCmd} \rangle \).

\[
\langle \text{VarDeclCmd} \rangle ::= \text{declare-var} \ \langle \text{Symbol} \rangle \ \langle \text{SortExpr} \rangle
\]

The variable name \( \langle \text{Symbol} \rangle \) must not clash with the following:

1. any previously declared universally quantified variable \( \langle \text{VarDeclCmd} \rangle \),
2. any previously declared 0-arity uninterpreted function \( \langle \text{FunDeclCmd} \rangle \),
3. any previously defined 0-arity function macro \( \langle \text{FunDefCmd} \rangle \), and
4. any previously declared 0-arity synthesis function \( \langle \text{SynthFunCmd} \rangle \).

### 3.5 Uninterpreted functions

Uninterpreted functions are declared using \( \langle \text{FunDeclCmd} \rangle \).

#### 3.5.1 Syntax

\[
\langle \text{FunDeclCmd} \rangle ::= \text{declare-fun} \ \langle \text{Symbol} \rangle \ (\langle \text{SortExpr} \rangle^\ast) \ \langle \text{SortExpr} \rangle
\]

The \( \langle \text{Symbol} \rangle \) names the uninterpreted function being declared, the first list of \( \langle \text{SortExpr} \rangle \) identifies the number and sorts of the input arguments, and the final \( \langle \text{SortExpr} \rangle \) identifies the sort of the function return value. The function name \( \langle \text{Symbol} \rangle \) must not clash with the following:

1. if the function is of 0-arity, then \( \langle \text{Symbol} \rangle \) should not clash with any previously declared universally quantified variable \( \langle \text{VarDeclCmd} \rangle \).
(set-logic LIA)

(declare-fun uf (Int) Int)

(synth-fun f ((x Int) (y Int)) Bool
 ((Start Bool (true false
 (<= IntExpr IntExpr)
 (= IntExpr IntExpr)
 (and Start Start)
 (or Start Start)
 (not Start)))
 (IntExpr Int (0 1 x y
 (+ IntExpr IntExpr)
 (- IntExpr IntExpr)))))

(declare-var x Int)

(constraint (f (uf x) (uf x)))

(check-synth)

Figure 2: Example SyGuS specification using uninterpreted functions.

(define-fun f ((x Int) (y Int)) Bool
 (= x y))

Figure 3: Sample valid answer for SyGuS specification of figure 2.

2. any previously declared uninterpreted function ((FunDeclCmd)) with the same input argument type signature,
3. any previously defined function macro ((FunDefCmd)) with the same input argument type signature, and
4. any previously declared synthesis function ((SynthFunCmd)) with the same input argument type signature.

3.5.2 Semantics

When uninterpreted functions are used in a SyGuS problem, the synthesized functions must satisfy the specification for all models of the uninterpreted functions. Uninterpreted functions may only be used in constraints (section 3.9), and not in function macros or grammars (sections 3.7 and 3.5).

For example, consider the specification in figure 2. Informally, this requires that for all functions uf : \( \mathbb{Z} \rightarrow \mathbb{Z} \) and integers \( x \in \mathbb{Z} \), \( f(uf(x), uf(x)) \) must hold. Therefore, the function in figure 3 satisfies the specification, but the function in figure 4 does not, even though it works for a specific instance of uf, viz. \( \forall x \in \mathbb{Z}, uf(x) = 5 \).
(define-fun f ((x Int) (y Int)) Bool
  (= x 5))

Figure 4: Example incorrect solution to the specification of figure 2. Note that even though this works for some instances of `uf`, it is incorrect because it does not work for all.

3.6 Terms and grammars

⟨Term⟩, ⟨GTerm⟩

⟨Term⟩ ::= (⟨Symbol⟩ ⟨Term⟩*)
| ⟨Literal⟩
| ⟨Symbol⟩
| ⟨LetTerm⟩

⟨LetTerm⟩ ::= (let (((⟨Symbol⟩ ⟨SortExpr⟩ ⟨Term⟩)+) ⟨Term⟩)

⟨GTerm⟩ ::= (⟨Symbol⟩ ⟨GTerm⟩*)
| ⟨Literal⟩
| ⟨Symbol⟩
| ⟨LetGTerm⟩
| ⟨Constant ⟨SortExpr⟩⟩
| ⟨Variable ⟨SortExpr⟩⟩
| ⟨InputVariable ⟨SortExpr⟩⟩
| ⟨LocalVariable ⟨SortExpr⟩⟩

⟨LetGTerm⟩ ::= (let (((⟨Symbol⟩ ⟨SortExpr⟩ ⟨GTerm⟩)+) ⟨GTerm⟩)

To describe function macros, grammars and constraints in SyGuS, one uses the ⟨Term⟩ and ⟨GTerm⟩ constructs. The difference between the two is the set of predefined macros (such as ⟨Constant ...⟩, etc.) that a ⟨GTerm⟩ may expand to. To allow synthesizers to perform common subexpression elimination to speed up their computation or reduce the size of their answers, let-expressions are allowed.

In grammars, a grammar expansion (⟨Constant ⟨SortExpr⟩⟩) expands to any literal of type ⟨SortExpr⟩. (⟨Variable ⟨SortExpr⟩⟩) expands to any variable currently in score of appropriate type, (⟨InputVariable ⟨SortExpr⟩⟩) and (⟨LocalVariable ⟨SortExpr⟩⟩) expand to any formal argument of the synthesis function, and any variable bound locally within a let-expression respectively.

The interpretation of the various syntactic constructs is as usual. In a let-construct, the first set of bindings (((⟨Symbol⟩ ⟨Term⟩)+) (resp. ⟨GTerm⟩)) refers to the parallel assignment of each ⟨Term⟩ (resp. ⟨GTerm⟩) to the corresponding ⟨Symbol⟩, as is the case in SMT-Lib2. If the ⟨Symbol⟩ bound by a let-expression is already bound, then its value is shadowed while evaluating the nested ⟨Term⟩.

(⟨Term⟩) and (⟨GTerm⟩) constructs are type-checked in the intuitive manner. The important restriction is that let-bound variables can shadow previously declared variables only if they are of the same sort.

3.7 Defining macros

⟨FunDefCmd⟩

⟨FunDefCmd⟩ ::= (define-fun ⟨Symbol⟩ (((⟨Symbol⟩ ⟨SortExpr⟩)*) ⟨SortExpr⟩ ⟨Term⟩)

⟨FunDefCmd⟩ command defines a function macro.

1. The function name ⟨Symbol⟩ may not clash with the following:

   (a) if the function is of 0-arity, then ⟨Symbol⟩ should not clash with any previously declared universally quantified variable ⟨VarDeclCmd⟩,
(b) any previously declared uninterpreted function (⟨FunDeclCmd⟩) with the same input argument type signature,
(c) any previously defined function macro (⟨FunDefCmd⟩) with the same input argument type signature, and
(d) any previously declared synthesis function (⟨SynthFunCmd⟩) with the same input argument type signature.

2. All arguments must have distinct names.

3. No nested let-bound variable in ⟨Term⟩ may shadow an input argument to the function.

4. ⟨Term⟩ is interpreted in the scope containing all previously defined function macros and formal arguments.

5. The sort of ⟨Term⟩ must match the return sort mentioned in ⟨SortExpr⟩.

3.8 Defining synthesis functions

⟨SynthFunCmd⟩

⟨SynthFunCmd⟩ ::= ⟨synth-fun ⟨Symbol⟩ (⟨(Symbol) ⟨SortExpr⟩)* ⟨SortExpr⟩ (⟨NTDef⟩+)⟩

⟨NTDef⟩ ::= ⟨ ⟨Symbol⟩ ⟨SortExpr⟩ ⟨GTerm⟩+ ⟩

A ⟨SynthFunCmd⟩ describes the sort and syntax of a function to be synthesized. The ⟨SynthFunCmd⟩ specifies the function name, input parameters, output sort, and grammar production rules respectively. The production rules corresponding to each non-terminal are described by an ⟨NTDef⟩, which specifies, in order, the non-terminal name, the sort of the resulting productions, and a non-empty sequence of production rules. Each ⟨GTerm⟩ corresponds to a production rule.

1. The function name ⟨Symbol⟩ may not clash with the following:
   (a) if the function is of 0-arity, then ⟨Symbol⟩ should not clash with any previously declared universally quantified variable (⟨VarDeclCmd⟩),
   (b) any previously declared uninterpreted function (⟨FunDeclCmd⟩) with the same input argument type signature,
   (c) any previously defined function macro (⟨FunDefCmd⟩) with the same input argument type signature, and
   (d) any previously declared synthesis function (⟨SynthFunCmd⟩) with the same input argument type signature.

2. All arguments must have distinct names.

3. No nested let-bound variable in any ⟨GTerm⟩ may shadow an input argument to the function.

4. All non-terminals must have unique names. For each non-terminal, its name should not clash with any of the following:
   (a) any previously defined 0-arity function macro (⟨FunDefCmd⟩),
   (b) any formal argument to the function, and
   (c) any let-bound variable in any production rule.

5. All let-bound variables in all ⟨GTerm⟩s with the same name have the same type.

6. Each production rule is interpreted in the scope with the following in scope:
(synth-fun f ((x Int) (y Int)) Int
  ((Start Int (x y z
    (+ Start Start)
    (let ((z Int Start)) Start)))))

Figure 5: Example of a well-formed ⟨SynthFunCmd⟩ involving let-expressions.

(a) all previously defined function macros,
(b) all formal arguments to the function, and
(c) all let-bound variables in all production rules. For an example of why this is the case, consider
that the expansion Start → z is well-formed in the grammar of figure 5

7. The sort of each production rule ⟨GTerm⟩ must match the sort at the non-terminal declaration.

8. There must be a non-terminal named Start. The sort of this non-terminal must match the output sort
of the ⟨SynthFunCmd⟩ being declared.

3.9 Describing synthesis constraints

⟨ConstraintCmd⟩

⟨ConstraintCmd⟩ ::= (constraint ⟨Term⟩)

A ⟨ConstraintCmd⟩ adds the constraint that when the synthesized functions are substituted into ⟨Term⟩,
for all values of the universally quantified variables, and all models of uninterpreted functions, ⟨Term⟩ evaluates
to true. ⟨Term⟩ must have boolean sort in the context with the following in scope:

1. all previously declared universally quantified variables,
2. all previously declared uninterpreted functions,
3. all previously defined function macros and
4. all previously declared synthesis functions.

3.10 Initiating synthesis and synthesizer output

⟨CheckSynthCmd⟩

⟨CheckSynthCmd⟩ ::= (check-synth)

Synthesis is initiated with ⟨CheckSynthCmd⟩. Exactly those synthesis functions declared before the
occurrence of this command need to be synthesized. Exactly those constraints occurring before this command
should be satisfied. On successful completion of synthesis, the synthesizer prints, for each previously declared
synthesis function, a well-typed ⟨FunDefCmd⟩ drawn from the appropriate syntax, so that all synthesized
functions together satisfy the specification. Otherwise, the synthesizer prints (fail). We give an example of
the output produced by a valid synthesizer on successfully synthesizing the specification of figure 1 in figure 6.

3.11 Solver-specific options

⟨SetOptsCmd⟩

Synthesizer flags and parameters may be controlled with ⟨SetOptsCmd⟩ – examples include specifying the
search strategy, or search parameters such as expression size. The syntax is as follows:

⟨SetOptsCmd⟩ ::= (set-options ((⟨Symbol⟩ ⟨QuotedLiteral⟩) \+)

The behavior of a synthesizer on encountering a ⟨SetOptsCmd⟩ is implementation defined. It is recom-
mended however, that synthesizers ignore unrecognized options, and choose reasonable defaults when the
options are left unspecified.
(define-fun max2 ((x Int) (y Int)) Int
 (ite (<= x y) y x))

(define-fun min2 ((x Int) (y Int)) Int
 (ite (<= x y) x y))

Figure 6: An example of valid synthesizer output to the specification of figure 1.

References

[1] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In FMCAD, pages 1–17, 2013.

[2] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB standard: Version 2.0. In Proceedings of the 8th International Workshop on Satisfiability Modulo Theories, volume 13, page 14, 2010.