Language to Specify Syntax-Guided Synthesis Problems

Mukund Raghothaman Abhishek Udupa

Friday 8th November, 2013

1 Introduction

We present a language to specify syntax guided synthesis (SyGuS) problems. An instance of a SyGuS problem has three parts:

1. A typed function f, whose body is to be synthesized,
2. a grammar G describing the syntactic structure of the potential solutions, and
3. a constraint φ, with some universally quantified variables v_1, v_2, $\ldots$

The problem is to find an expression bodies for f from the grammar G so that the constraint is universally satisfied, i.e. to answer the question

$$\exists f, \forall v_1, v_2, \ldots, \varphi\left(f, v_1, v_2, \ldots\right)?$$

The logical symbols and their interpretations in f are restricted to a background theory, and the specification constraint φ is a quantifier-free formula.

For example, over the theory of linear integer arithmetic, the function computing the maximum of a pair of integers, max_2 may be specified as

$$\begin{aligned}\exists max_2 : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}, \forall x, y : \mathbb{Z},\ & max_2\left(x, y\right) \geq x \wedge max_2\left(x, y\right) \geq y \\ & \wedge \left(max_2\left(x, y\right) = x \vee max_2\left(x, y\right) = y\right)?\end{aligned}$$

We are interested in expression bodies for $max_2\left(x, y\right)$ drawn from piecewise linear functions, so the grammar G would be

$$\begin{aligned}\texttt{Expr} ::= {} & 0 | 1 | x | y \\ & | \texttt{Expr} + \texttt{Expr} \\ & | \texttt{Expr} - \texttt{Expr} \\ & | \texttt{if BoolExpr},\ \texttt{then Expr},\ \texttt{else Expr} \\ \texttt{BoolExpr} ::= {} & \texttt{BoolExpr} \wedge \texttt{BoolExpr} \\ & | \neg \texttt{BoolExpr} \\ & | \texttt{BoolExpr} \leq \texttt{BoolExpr}\end{aligned}$$

2 Example SyGuS Programs

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

Continuing with the example of max_2 from the previous section, figure 1 contains the corresponding SyGuS code. The first command informs the synthesizer to load symbols corresponding to linear integer

```
(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)))))

(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)))))

(check-synth)
```

Figure 1: SyGuS specification for a function computing the maximum of two integers.

arithmetic. The `synth-fun` command describes the function to be synthesized: `max2` is a function of 2 integer arguments, `x` and `y`, and returns an integer value. The remainder of the command specifies the grammar for f. The grammar is specified as a sequence of typed non-terminal symbols. In this case, there are two of these: `Start` and `StartBool`, of sorts `Int` and `Bool` respectively. The non-terminal `Start` is special in that it specifies the starting non-terminal for the grammar. We then declare the universally quantified variables `x` and `y`. The final specification constraint is the conjunction of the individual constraints.

3 Specification Language

The SyGuS specification language is closely modeled on SMT-Lib2. 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.

$$
\begin{aligned}
\langle \mathit{SyGuS} \rangle ::= {} & \langle \mathit{SetLogicCmd} \rangle \; \langle \mathit{Cmd} \rangle^{+} \\
& \mid \langle \mathit{Cmd} \rangle^{+} \\
\langle \mathit{Cmd} \rangle ::= {} & \langle \mathit{SortDefCmd} \rangle \\
& \mid \langle \mathit{VarDeclCmd} \rangle \\
& \mid \langle \mathit{FunDeclCmd} \rangle \\
& \mid \langle \mathit{FunDefCmd} \rangle \\
& \mid \langle \mathit{SynthFunCmd} \rangle \\
& \mid \langle \mathit{ConstraintCmd} \rangle \\
& \mid \langle \mathit{CheckSynthCmd} \rangle \\
& \mid \langle \mathit{SetOptsCmd} \rangle
\end{aligned}
$$

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 Identifiers

Identifiers are denoted with the non-terminal $\langle \mathit{Symbol} \rangle$:

$$
\begin{aligned}
\langle \mathit{SpecialChar} \rangle \; = {} & \{\_, +, -, *, \&, |, !, \sim, <, >, =, /, \%, ?, ., \$, \hat{}\,\} \\
\langle \mathit{Symbol} \rangle ::= {} & (\texttt{[a-z]} | \texttt{[A-Z]} | \langle \mathit{SpecialChar} \rangle)\; (\texttt{[a-z]} | \texttt{[A-Z]} | \texttt{[0-9]} | \langle \mathit{SpecialChar} \rangle)^{*}
\end{aligned}
$$

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

$$
\langle \mathit{QuotedLiteral} \rangle ::= \texttt{"}\; (\texttt{[a-z]} | \texttt{[A-Z]} | \texttt{[0-9]} | .)^{+} \; \texttt{"}
$$

3.1.3 Literals

$$
\begin{aligned}
\langle \mathit{Literal} \rangle ::= {} & \langle \mathit{IntConst} \rangle | \langle \mathit{RealConst} \rangle | \langle \mathit{BoolConst} \rangle | \langle \mathit{BVConst} \rangle | \langle \mathit{EnumConst} \rangle \\
\langle \mathit{IntConst} \rangle ::= {} & \texttt{[0-9]}^{+} \Big| - \texttt{[0-9]}^{+} \\
\langle \mathit{RealConst} \rangle ::= {} & \texttt{[0-9]}^{+} \,.\, \texttt{[0-9]}^{+} \Big| - \texttt{[0-9]}^{+} \,.\, \texttt{[0-9]}^{+} \\
\langle \mathit{BoolConst} \rangle ::= {} & \texttt{true} | \texttt{false} \\
\langle \mathit{BVConst} \rangle ::= {} & \#\texttt{b}\, \texttt{[0-1]}^{+} \Big| \#\texttt{x}\, (\texttt{[0-9]} | \texttt{[a-f]} | \texttt{[A-F]})^{+} \\
\langle \mathit{EnumConst} \rangle ::= {} & \langle \mathit{Symbol} \rangle :: \langle \mathit{Symbol} \rangle
\end{aligned}
$$

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. Boolean constants may be either `true` or `false`. 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 will be covered in subsection 3.3.

3.2 Declaring the problem logic $\langle \mathit{SetLogicCmd} \rangle$

On encountering the optional $\langle \mathit{SetLogicCmd} \rangle$, the synthesizer loads appropriate pre-defined function symbols and constants. Current theories include linear integer arithmetic (`LIA`), the theory of bit-vectors (`BV`), and the theories of reals (`Reals`) and arrays (`Arrays`).

$$\langle \mathit{SetLogicCmd} \rangle ::= (\texttt{set-logic}\ \langle \mathit{Symbol} \rangle\ )$$

3.3 Defining new sorts $\langle \mathit{SortDefCmd} \rangle$, $\langle \mathit{SortExpr} \rangle$

SyGuS expects that the sorts of functions, variables, and grammar symbols be explicitly specified. The syntactic construct $\langle \mathit{SortExpr} \rangle$ is used for this, and the sort definition command $\langle \mathit{SortDefCmd} \rangle$ permits defining useful shorthands.

$$\begin{aligned}
\langle \mathit{SortExpr} \rangle ::= &\ \texttt{Int} | \texttt{Bool} | \texttt{Real} \\
&| (\texttt{BitVec}\ \langle \mathit{PositiveInteger} \rangle\ ) \\
&| (\texttt{Enum}\ (\ \langle \mathit{Symbol} \rangle^{+}\ )) \\
&| (\texttt{Array}\ \langle \mathit{SortExpr} \rangle\ \ \langle \mathit{SortExpr} \rangle\ ) \\
&|\ \langle \mathit{Symbol} \rangle \\
\langle \mathit{SortDefCmd} \rangle ::= &(\texttt{define-sort}\ \langle \mathit{Symbol} \rangle\ \ \langle \mathit{SortExpr} \rangle\ )
\end{aligned}$$

The sorts `Int`, `Bool`, and `Real` refer to integers, booleans and real numbers respectively. For each positive integer n, `BitVec` n refers to the sort of bit-vectors n bits long. Given a set of constructor symbols S_1, S_2, …, the sort (`Enum` (S_1 S_2 …)) refers to the enumerated type having those elements. Since the only way to represent an enumerated constant (subsection 3.1.3) 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 (`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 (`define-sort` S $\langle \mathit{SortExpr} \rangle$), it may subsequently be referred to simply as S rather than the full expression $\langle \mathit{SortExpr} \rangle$. The identifier $\langle \mathit{Symbol} \rangle$ used to name a sort may not have been previously used as a sort name. Furthermore, the $\langle \mathit{SortExpr} \rangle$ used in the body of a $\langle \mathit{SortDefCmd} \rangle$ must be well-formed. We say that a $\langle \mathit{SortExpr} \rangle$ is well-formed if

1. it is an instance of `Int`, `Bool`, `Real`, or a bit-vector or enumerated sort defined using `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 an instance of $\langle \mathit{Symbol} \rangle$, and the appropriate symbol has already been defined using a $\langle \mathit{SortDefCmd} \rangle$.

3.4 Universally quantified variables $\langle \mathit{VarDeclCmd} \rangle$

Universally quantified variables may be declared with $\langle \mathit{VarDeclCmd} \rangle$.

$$\langle \mathit{VarDeclCmd} \rangle ::= (\texttt{declare-var}\ \langle \mathit{Symbol} \rangle\ \ \langle \mathit{SortExpr} \rangle\ )$$

The variable name $\langle \mathit{Symbol} \rangle$ should not have been used previously to declare any other universally quantified variable, nor should it have been used to declare a 0-arity function using $\langle \mathit{FunDeclCmd} \rangle$, $\langle \mathit{FunDefCmd} \rangle$ or $\langle \mathit{SynthFunCmd} \rangle$. The variable sort $\langle \mathit{SortExpr} \rangle$ should be well-formed.

3.5 Uninterpreted functions $\langle FunDeclCmd\rangle$

$$\langle FunDeclCmd\rangle ::= (\texttt{declare-fun}\ \langle Symbol\rangle\ \ (\ \langle SortExpr\rangle^*\ )\ \ \langle SortExpr\rangle\ )$$

Uninterpreted functions are declared using $\langle FunDeclCmd\rangle$. The $\langle Symbol\rangle$ names the uninterpreted function being declared, the first list of $\langle SortExpr\rangle$ identifies the number and sorts of the input arguments, and the final $\langle SortExpr\rangle$ identifies the sort of the function return value. All instances of $\langle SortExpr\rangle$ must be well-formed. Its name should not clash with that of any previously declared uninterpreted function, macro, or synthesis function with the same input argument type signature. If the function is of 0-arity, then its name should also not clash with that of any previously declared universally quantified variable.

When uninterpreted functions are used in a SyGuS problem, the synthesized functions must satisfy the specification for all models of the uninterpreted functions. Furthermore, uninterpreted functions may be used in constraints, macros and in grammars, as described in subsections 3.6-3.9.

3.6 Terms and grammars $\langle Term\rangle$, $\langle GTerm\rangle$

$$\begin{aligned}
\langle Term\rangle &::= (\ \langle Symbol\rangle\ \ \langle Term\rangle^*\ )\\
&\quad\mid \langle Literal\rangle\\
&\quad\mid \langle Symbol\rangle\\
&\quad\mid \langle LetTerm\rangle\\
\langle LetTerm\rangle &::= (\texttt{let}\ ((\ \langle Symbol\rangle\ \ \langle Term\rangle\ )^+)\ \ \langle Term\rangle\ )
\end{aligned}$$

The sequence of symbol-term pairs in the first part of a $\langle LetTerm\rangle$ identify and define the bound variables, and the final $\langle Term\rangle$ is evaluated in a context with these additional variables being defined (or shadowing their original values if they were already in scope).

A term is well-typed in a scope if

1. it is an instance of $(\ \langle Symbol\rangle\ \ \langle Term\rangle^*\ )$, all sub-terms are well-typed, and there is a unique function symbol declared with input type signature equal to the type signature of the present arguments, or
2. it is an instance of $\langle Literal\rangle$, or
3. it is a $\langle Symbol\rangle$, and refers to some 0-arity symbol currently in scope, or
4. it is a $\langle LetTerm\rangle$ $(\texttt{let}\ ((\ \langle Symbol\rangle\ \ \langle Term\rangle\ )^+)\ \ \langle Term\rangle\ )$, where all variables are bound to well-typed terms, and the final term is well-typed in a context with these additional variables in scope (or shadowing their original values if they were already in scope).

3.7 Defining macros $\langle FunDefCmd\rangle$

$$\langle FunDefCmd\rangle ::= (\texttt{define-fun}\ \langle Symbol\rangle\ \ ((\ \langle Symbol\rangle\ \ \langle SortExpr\rangle\ )^*)\ \ \langle SortExpr\rangle\ \ \langle Term\rangle\ )$$

This command defines a function macro. The function may not have the same name as any previously defined uninterpreted function, macro or synthesis function with the same input argument type signature. In addition, if the function is of 0-arity, then its name should also not clash with that of any previously declared universally quantified variable. No two input arguments may have the same name, but may shadow previously defined functions of 0-arity. All internally occurring $\langle SortExpr\rangle$ must be well-formed. The function body $\langle Term\rangle$ must have the same sort as the immediately preceding $\langle SortExpr\rangle$, with all previously defined function macros and present function arguments in scope.

3.8 Defining synthesis functions $\langle SynthFunCmd \rangle$

$$\langle SynthFunCmd \rangle ::= (\texttt{synth-fun}\ \langle Symbol \rangle\ ((\ \langle Symbol \rangle\ \ \langle SortExpr \rangle\ )^{*})\ \langle SortExpr \rangle\ (\ \langle NTDef \rangle^{+}\ ))$$
$$\langle NTDef \rangle ::= (\ \langle Symbol \rangle\ \ \langle SortExpr \rangle\ \ \langle GTerm \rangle^{+}\ )$$

This command 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.

The function may not have the same name as any previously defined uninterpreted function, macro or synthesis function with the same input argument type signature. In addition, if the function is of 0-arity, then its name should also not clash with that of any previously declared universally quantified variable.

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

No two non-terminal descriptions ⟨*NTDef*⟩ which share the same name may have different sorts. The sort of each nested ⟨*GTerm*⟩, with all macros, function arguments and present non-terminals in scope must match the sort of the parent ⟨*NTDef*⟩.

3.9 Describing synthesis constraints $\langle ConstraintCmd \rangle$

$$\langle ConstraintCmd \rangle ::= (\texttt{constraint}\ \langle Term \rangle\ )$$

Adds the constraint that when the synthesized functions are substituted into ⟨*Term*⟩, for all values of the universally quantified variables, and all models of uniterpreted functions, the ⟨*Term*⟩ evaluates to true. ⟨*Term*⟩ must have boolean sort in the context with all universally quantified variables, uninterpreted functions, function macros and synthesis functions in scope.

3.10 Initiating synthesis and synthesizer output $\langle CheckSynthCmd \rangle$

$$\langle CheckSynthCmd \rangle ::= (\texttt{check-synth})$$

Synthesis is initiated with ⟨*SynthFunCmd*⟩. Exactly those synthesis functions declared before the occurrence of this command need to be synthesized. Exactly those constraints occurring before this command need to be satisfied. On successful completion of synthesis, the synthesizer prints, for each previously declared synthesis function, a ⟨*FunDefCmd*⟩ drawn from the appropriate syntax, so that all synthesized functions together satisfy the specification.

3.11 Solver-specific options $\langle SetOptsCmd \rangle$

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

$$\langle SetOptsCmd \rangle ::= (\texttt{set-options}\ ((\ \langle Symbol \rangle\ \ \langle QuotedLiteral \rangle\ )^{+}))$$

The behavior of a synthesizer on encountering a ⟨*SetOptsCmd*⟩ is implementation defined. It is recommended however, that synthesizers ignore unrecognized options.