Language to Specify Syntax-Guided Synthesis Problems

Mukund Raghothaman Abhishek Udupa

Sunday 23rd October, 2016

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, ..., 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 φ, with some universally quantified variables v_1, v_2, ..., 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:

$$\forall v_1, v_2, \ldots, \varphi\left(f_1, f_2, \ldots, v_1, v_2, \ldots\right).$$

The constraint formula φ is quantifier-free, and the logical symbols and their interpretation in φ 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

$$\begin{aligned}\forall x, y : \mathbb{Z}, \quad & max_2\left(x, y\right) \geq x \wedge max_2\left(x, y\right) \geq y \\ & \wedge \left(max_2\left(x, y\right) = x \vee max_2\left(x, y\right) = y\right) \\ & \wedge \left(max_2\left(x, y\right) + min_2\left(x, y\right) = x + y\right).\end{aligned}$$

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

$$\begin{aligned}\texttt{Expr} &::= 0 \mid 1 \mid x \mid y \\ &\mid \texttt{Expr} + \texttt{Expr} \\ &\mid \texttt{Expr} - \texttt{Expr} \\ &\mid (\texttt{ite BoolExpr Expr Expr}) \\ \texttt{BoolExpr} &::= \texttt{BoolExpr} \wedge \texttt{BoolExpr} \\ &\mid \neg\texttt{BoolExpr} \\ &\mid \texttt{Expr} \leq \texttt{Expr}\end{aligned}$$

```
(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.

2 Example SyGuS Specification

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

We continue the example of max_2 and 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 `max2` is a function of two integer arguments `x` and `y`, and returns an integer value. The rest of the command describes the grammar for `max2`. `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 `min2` is identical to that of `max2`, 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 $\texttt{max2}\,(\texttt{x}, \texttt{y}) \geq \texttt{x}$. The final synthesis contraint φ 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. 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.

$$
\begin{array}{rcl}
\langle SyGuS \rangle & ::= & \langle SetLogicCmd \rangle \, \langle Cmd \rangle^{+} \\
& | & \langle Cmd \rangle^{+} \\
\langle Cmd \rangle & ::= & \langle SortDefCmd \rangle \\
& | & \langle VarDeclCmd \rangle \\
& | & \langle FunDeclCmd \rangle \\
& | & \langle FunDefCmd \rangle \\
& | & \langle SynthFunCmd \rangle \\
& | & \langle ConstraintCmd \rangle \\
& | & \langle CheckSynthCmd \rangle \\
& | & \langle SetOptsCmd \rangle
\end{array}
$$

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 $\langle Symbol \rangle$. 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

with a digit.

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

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

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

3.1.4 Literals

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

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. `true` and `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 $\langle SetLogicCmd \rangle$

On encountering the optional $\langle SetLogicCmd \rangle$, the synthesizer loads appropriate pre-defined function symbols and constants. Current theories include

1. `LIA`: Linear integer arithmetic, for functions such as + and −,
2. `BV`: Theory of bit-vectors, for functions such as `bvadd` and `bvlshr`,
3. `Reals`: Theory of real numbers, and
4. `Arrays`: Theory of arrays.

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

3.3 Defining new sorts $\langle SortDefCmd \rangle$, $\langle SortExpr \rangle$

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

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

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 3.1.4) 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 (`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 $\langle SortExpr \rangle$), it may subsequently be referred to simply as S rather than the full expression $\langle SortExpr \rangle$. The identifier $\langle Symbol \rangle$ used to name a sort should not have been previously used as a sort name. Every $\langle SortExpr \rangle$ in a SyGuS specification must be well-formed. We say that a $\langle SortExpr \rangle$ 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 $\langle Symbol \rangle$, and $\langle Symbol \rangle$ has been previously defined using a $\langle SortDefCmd \rangle$.

3.4 Universally quantified variables $\langle VarDeclCmd \rangle$

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

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

The variable name $\langle Symbol \rangle$ must not clash with the following:

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

3.5 Uninterpreted functions $\langle FunDeclCmd \rangle$

Uninterpreted functions are declared using $\langle FunDeclCmd \rangle$.

3.5.1 Syntax

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

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. The function name $\langle Symbol \rangle$ must not clash with the following:

1. if the funtion is of 0-arity, then $\langle Symbol \rangle$ should not clash with any previously declared universally quantified variable ($\langle VarDeclCmd \rangle$),

```
(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 ($\langle FunDeclCmd \rangle$) with the same input argument type signature,
3. any previously defined function macro ($\langle FunDefCmd \rangle$) with the same input argument type signature, and
4. any previously declared synthesis function ($\langle SynthFunCmd \rangle$) 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 3.8).

For example, consider the specification in figure 2. Informally, this requires that for all functions $\texttt{uf} : \mathbb{Z} \to \mathbb{Z}$ and integers $x \in \mathbb{Z}$, $\texttt{f}\left(\texttt{uf}\left(x\right), \texttt{uf}\left(x\right)\right)$ 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. $\forall x \in \mathbb{Z}, \texttt{uf}\left(x\right) = 5$.

```
(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 $\langle Term \rangle$, $\langle GTerm \rangle$

$$
\begin{array}{rcl}
\langle Term \rangle & ::= & (\ \langle Symbol \rangle\ \ \langle Term \rangle^{*}\ ) \\
& | & \langle Literal \rangle \\
& | & \langle Symbol \rangle \\
& | & \langle LetTerm \rangle \\
\langle LetTerm \rangle & ::= & (\texttt{let}\ ((\ \langle Symbol \rangle\ \ \langle SortExpr \rangle\ \ \langle Term \rangle\ )^{+})\ \ \langle Term \rangle\ )
\end{array}
$$

$$
\begin{array}{rcl}
\langle GTerm \rangle & ::= & (\ \langle Symbol \rangle\ \ \langle GTerm \rangle^{*}\ ) \\
& | & \langle Literal \rangle \\
& | & \langle Symbol \rangle \\
& | & \langle LetGTerm \rangle \\
& | & (\texttt{Constant}\ \langle SortExpr \rangle\ ) \\
& | & (\texttt{Variable}\ \langle SortExpr \rangle\ ) \\
& | & (\texttt{InputVariable}\ \langle SortExpr \rangle\ ) \\
& | & (\texttt{LocalVariable}\ \langle SortExpr \rangle\ ) \\
\langle LetGTerm \rangle & ::= & (\texttt{let}\ ((\ \langle Symbol \rangle\ \ \langle SortExpr \rangle\ \ \langle GTerm \rangle\ )^{+})\ \ \langle GTerm \rangle\ )
\end{array}
$$

To describe function macros, grammars and constraints in SyGuS, one uses the $\langle Term \rangle$ and $\langle GTerm \rangle$ constructs. The difference between the two is the set of predefined macros (such as (`Constant` ...), etc.) that a $\langle GTerm \rangle$ 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` $\langle SortExpr \rangle$) expands to any literal of type $\langle SortExpr \rangle$. (`Variable` $\langle SortExpr \rangle$) expands to any variable currently in score of appropriate type, (`InputVariable` $\langle SortExpr \rangle$) and (`LocalVariable` $\langle SortExpr \rangle$) 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 $((\ \langle Symbol \rangle\ \ \langle Term \rangle\ )^{+})$ (resp. $\langle GTerm \rangle$) refers to the parallel assignment of each $\langle Term \rangle$ (resp. $\langle GTerm \rangle$) to the corresponding $\langle Symbol \rangle$, as is the case in SMT-Lib2. If the $\langle Symbol \rangle$ bound by a `let`-expression is already bound, then its value is shadowed while evaluating the nested $\langle Term \rangle$.

$\langle Term \rangle$ and $\langle GTerm \rangle$ 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 $\langle FunDefCmd \rangle$

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

$\langle FunDefCmd \rangle$ command defines a function macro.

1. The function name $\langle Symbol \rangle$ may not clash with the following:

   (a) if the funtion is of 0-arity, then $\langle Symbol \rangle$ should not clash with any previously declared universally quantified variable ($\langle VarDeclCmd \rangle$),

(b) any previously declared uninterpreted function ($\langle FunDeclCmd \rangle$) with the same input argument type signature,

(c) any previously defined function macro ($\langle FunDefCmd \rangle$) with the same input argument type signature, and

(d) any previously declared synthesis function ($\langle SynthFunCmd \rangle$) with the same input argument type signature.

2. All arguments must have distinct names.

3. No nested `let`-bound variable in $\langle Term \rangle$ may shadow an input argument to the function.

4. $\langle Term \rangle$ is interpreted in the scope containing all previously defined function macros and formal arguments.

5. The sort of $\langle Term \rangle$ must match the return sort mentioned in $\langle SortExpr \rangle$.

3.8 Defining synthesis functions $\langle SynthFunCmd \rangle$

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

A $\langle SynthFunCmd \rangle$ describes the sort and syntax of a function to be synthesized. The $\langle SynthFunCmd \rangle$ 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 $\langle NTDef \rangle$, which specifies, in order, the non-terminal name, the sort of the resulting productions, and a non-empty sequence of production rules. Each $\langle GTerm \rangle$ corresponds to a production rule.

1. The function name $\langle Symbol \rangle$ may not clash with the following:

   (a) if the funtion is of 0-arity, then $\langle Symbol \rangle$ should not clash with any previously declared universally quantified variable ($\langle VarDeclCmd \rangle$),

   (b) any previously declared uninterpreted function ($\langle FunDeclCmd \rangle$) with the same input argument type signature,

   (c) any previously defined function macro ($\langle FunDefCmd \rangle$) with the same input argument type signature, and

   (d) any previously declared synthesis function ($\langle SynthFunCmd \rangle$) with the same input argument type signature.

2. All arguments must have distinct names.

3. No nested `let`-bound variable in any $\langle GTerm \rangle$ 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 ($\langle FunDefCmd \rangle$),

   (b) any formal argument to the function, and

   (c) any `let`-bound variable in any production rule.

5. All `let`-bound variables in all $\langle GTerm \rangle$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 $\langle SynthFunCmd \rangle$ 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` $\rightarrow$ `z` is well-formed in the grammar of figure 5.

7. The sort of each production rule $\langle GTerm \rangle$ 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 ouput sort of the $\langle SynthFunCmd \rangle$ being declared.

3.9 Describing synthesis constraints $\langle ConstraintCmd \rangle$

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

A $\langle ConstraintCmd \rangle$ adds the constraint that when the synthesized functions are substituted into $\langle Term \rangle$, for all values of the universally quantified variables, and all models of uniterpreted functions, $\langle Term \rangle$ evaluates to true. $\langle Term \rangle$ 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 $\langle CheckSynthCmd \rangle$

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

Synthesis is initiated with $\langle CheckSynthCmd \rangle$. 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 $\langle FunDefCmd \rangle$ 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 $\langle SetOptsCmd \rangle$

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

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

The behavior of a synthesizer on encountering a $\langle SetOptsCmd \rangle$ is implementation defined. It is recommended 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.