Syntax-Guided Synthesis with Quantitative Syntactic Objectives

Qinheping Hu(✉) and Loris D'Antoni

University of Wisconsin-Madison, Madison, USA
{qhu28,loris}@cs.wisc.edu

Abstract. Automatic program synthesis promises to increase the productivity of programmers and end-users of computing devices by automating tedious and error-prone tasks. Despite the practical successes of program synthesis, we still do not have systematic frameworks to synthesize programs that are "good" according to certain metrics—e.g., produce programs of reasonable sizes or with good runtime—and to understand when synthesis can result in such good programs. In this paper, we propose QSyGuS, a unifying framework for describing syntax-guided synthesis problems with quantitative objectives over the syntax of the synthesized programs. QSyGuS builds on weighted (tree) grammars, a clean and foundational formalism that provides flexible support for different quantitative objectives, useful closure properties, and practical decision procedures. We then present an algorithm for solving QSyGuS. Our algorithm leverages closure properties of weighted grammars to generate intermediate problems that can be solved using non-quantitative SyGuS solvers. Finally, we implement our algorithm in a tool, Quasi, and evaluate it on 26 quantitative extensions of existing SyGuS benchmarks. Quasi can synthesize optimal solutions in 15/26 benchmarks with times comparable to those needed to find an arbitrary solution.

1 Introduction

The goal of program synthesis is to find a program in some search space that meets a specification—e.g., a set of examples or a logical formula. Recently, a large family of synthesis problems has been unified into a framework called syntax-guided synthesis (SyGuS). A SyGuS problem is specified by a context-free grammar describing the search space of programs, and a logical formula describing the specification. Many synthesizers now support this format [2] and annually compete in synthesis competitions [4]. Thanks to these competitions, these solvers are now quite mature and are finding wide application [14].

While the logical specification mechanism provided by SyGuS is powerful, it can only capture the functional requirements of the synthesis problem—e.g., the program should perform correctly on a given set of input/output examples. When multiple possible programs can satisfy the specification, SyGuS *does not* provide a way to prefer one to the other—e.g., one cannot ask a solver to return the program with the fewest if-statements. As a consequence, existing synthesis

H. Chockler and G. Weissenbacher (Eds.): CAV 2018, LNCS 10981, pp. 386–403, 2018.
https://doi.org/10.1007/978-3-319-96145-3_21

tools do not provide guarantees about what solution is returned if multiple ones exist. While a few synthesizers have attempted to include some form of specification to express this kind of quantitative intents [7,15,16,19], these approaches are domain-specific, do not apply to SyGuS problems, and do not provide a simple and flexible specification mechanism. The lack of a formal treatment of quantitative requirements stands in the way of designing synthesizers that can take advantage of quantitative objectives to perform more efficient forms of synthesis.

In this paper, we propose QSyGuS, a unifying framework for describing syntax-guided synthesis problems with quantitative objectives over the syntax of the synthesized programs—e.g., find the most likely program with respect to a given probability distribution—and present an algorithm for solving synthesis problems expressed in this framework. We focus on syntactic objectives because they are the most common ones in practical applications of program synthesis. For example, in programming by examples it is desirable to produce small programs with fewer constants because these programs are more likely to generalize to examples outside of the specification [13]. QSyGuS extends SyGuS in two ways. First, in QSyGuS the search space is represented using weighted grammars, which augment context-free grammars with the ability to assign weights to programs. Second, QSyGuS allows the user to specify constraints over the weight of the program, including optimization objectives—e.g., find the program with the fewest if-statements and with the lowest depth.

QSyGuS is a natural, general, and flexible formalism and is grounded in the well-studied theory of weighted grammars. We leverage this theory and design an algorithm for solving QSyGuS problems using closure properties of weighted grammars. Given a QSyGuS problem, our algorithm generates a SyGuS problem that can be delegated to existing SyGuS solvers. The algorithm then iteratively refines the solution returned by the SyGuS solver to find an optimal one by further generating new SyGuS instances using weighted grammar operations. We implement our algorithm in a tool, QuaSi, and evaluate it on 26 quantitative extensions of existing SyGuS benchmarks. QuaSi can synthesize optimal solutions in 15/26 benchmarks with times comparable to those needed to find a solution that does not need to satisfy any quantitative objective.

Contributions. In summary, our contributions are:

- QSyGuS, a formal framework grounded in the theory of weighted grammars that can describe syntax-guided synthesis problems with quantitative objectives over the syntax of the synthesized programs (Sect. 3).
- An algorithm for solving QSyGuS problems that leverages closure properties of weighted grammars and existing SyGuS solvers (Sect. 4).
- QuaSi, a tool for specifying and solving QSyGuS problems that interfaces with existing SyGuS solvers and a comprehensive evaluation of QuaSi, which shows that QuaSi can efficiently solve QSyGuS problems over different types of weights, including additive weights, probabilities, and combinations of multiple weights (Sect. 5).

$$\begin{array}{rcl} \text{Start} ::= & \text{Start} + \text{Start}/(\mathbf{0},\mathbf{1}) \\ & \mid \text{if(BExpr) then Start else Start}/(\mathbf{1},\mathbf{0}) \\ & \mid x \mid y \mid 0 \mid 1 \end{array} \qquad \begin{array}{rcl} \text{BExpr} ::= & \text{Start} > \text{Start} \\ & \mid \neg\text{BExpr} \\ & \mid \text{BExpr} \wedge \text{BExpr} \end{array}$$

Fig. 1. Weighted grammar that assigns weight $(w_1, w_2) \in \mathsf{Nat} \times \mathsf{Nat}$ to a program where w_1 is the number of if-statements and w_2 is the number of plus-statements.

2 Illustrative Example

In this section, we illustrate the main components of our framework using an example. We start with a Syntax-Guided Synthesis (SyGuS) problem in which no quantitative objective is provided. We recall that the goal of a SyGuS problem is to synthesize a function f of a given type that is accepted by a context-free grammar G, and such that $\forall x.\phi(f, x)$ holds (for a given Boolean constraint ϕ).

The following SyGuS problem asks to synthesize a function that is accepted by the following grammar and that computes the max of two numbers.

$$\begin{array}{rl} \text{Start} ::= & \text{Start} + \text{Start} \mid \text{if(BExpr) then Start else Start} \mid x \mid y \mid 0 \mid 1 \\ \text{BExpr} ::= & \text{Start} > \text{Start} \mid \neg\text{BExpr} \mid \text{BExpr} \wedge \text{BExpr} \end{array}$$

The semantic constraint is given by the following formula.

$$\psi(f) \stackrel{\text{def}}{=} \forall x, y. f(x, y) \geq x \wedge f(x, y) \geq y \wedge (f(x, y) = x \vee f(x, y) = y)$$

The following three programs are semantically equivalent, but syntactically different solutions.

$$\begin{array}{l} max_1(x, y) = \text{if}(x > y) \text{ then } x \text{ else } y \\ max_2(x, y) = \text{if}(x > y) \text{ then } (x + 0) \text{ else } (y + 0) \\ max_3(x, y) = \text{if}(x > y) \text{ then } x \text{ else } (\text{if}(y > x) \text{ then } y \text{ else } x) \end{array}$$

All solutions are correct, but the user might, for example, prefer the smallest one. However, SyGuS does not provide ways to specify this quantitative intent.

Adding Weights. In our formalism, QSyGuS, we augment context-free grammars to assign weights to programs in the search space. Concretely, we adopt weighted grammars [10], a well-studied formalism with many desirable properties. In a weighted grammar, each production is assigned a weight. For example, the weighted grammar shown in Fig. 1 extends the one from the previous SyGuS example to assign to each program p a pair of weights (w_1, w_2) where w_1 is the number of if-statements and w_2 is the number of plus operators in p. In this case, the weights are pairs of integers and the weight of a grammar derivation is the pairwise sum of all the weights of the productions involved in the derivation—e.g., the sum of (w_1, w_2) and (w_1', w_2') is $(w_1 + w_1', w_2 + w_2')$. In the figure, we write $/(w_1, w_2)$ to assign weight (w_1', w_2') to a production. We omit the weight for productions with cost $(0, 0)$. The functions max_1, max_2 and max_3 have weights $(1, 0)$, $(1, 2)$, and $(2, 0)$ respectively.

Adding and Solving Quantitative Objectives. Once we have a way to assign weights to programs, QSyGuS allows the user to specify quantitative objectives over the weights of the productions—e.g., only allow solutions with fewer than 4 if-statements. In our example, we could require the solution to be minimal with respect to the number of if-statements, i.e., minimize the first component of the paired weight. With these constraints both max_1 and max_2 would be considered optimal solutions because there exists no solution with 0 if-statements. If we require the solution to also be minimal with respect to the second component of the paired weight, max_1 will be *a possible* optimal solution.

Our tool QuaSi can automatically discover solutions in both these cases. Let's consider the last minimization objective. In this case, QuaSi first uses existing SyGuS solvers to synthesize an initial solution using the non-weighted version of the grammar. Let's say that the returned solution is, for example, max_3 of weight $(2, 0)$. QuaSi uses this solution to build a new SyGuS instance that only accepts programs with at most one if-statement. Solving this SyGuS problem can, for example, result in the program max_2 of weight $(1, 2)$, which will require our solver to build yet another SyGuS instance. This approach is repeated and if it terminates, an optimal program is found.

3 SyGuS with Quantitative Objectives

In this section, we introduce our framework for defining syntax-guided synthesis problems with quantitative objectives over the syntax of the synthesized programs. We first provide preliminary definitions for notions such as semirings (Sect. 3.1) and weighted tree grammars (Sect. 3.2), and then use these notions to augment SyGuS problems with quantitative objectives (Sect. 3.3).

3.1 Weights over Semirings

We now define the universe of weights we will assign to programs. In general, weights are defined using monoids—i.e., sets equipped with an addition operator—but when a grammar is nondeterministic—i.e., it can produce the same term using multiple derivations—the same term might be assigned multiple weights. Hence, we choose to use semirings. Since we also care about optimization objectives, we assume all our semirings are equipped with a partial order.

Definition 1 (Semiring). *A (ordered)* semiring *is a pair* $(\boldsymbol{S}, \preceq)$ *where (*i*)* $\boldsymbol{S} = (S, \oplus, \otimes, 0, 1)$ *is an algebra consisting of a commutative monoid* $(S, \oplus, 0)$ *and a monoid* $(S, \otimes, 1)$ *such that* $\otimes$ *distributes over* $\oplus$*,* $0 \neq 1$*, and, for every* $x \in S$*,* $x \otimes 0 = 0$*, (*ii*)* $\preceq \subset S \times S$ *is a partial order over* S*.*

We often use the word semiring to refer to just the algebra $\mathbf{S}$.

Example 1. In this paper, we focus on semirings with the following algebras.

Boolean $\mathsf{Bool} = (\mathbb{B}, \vee, \wedge, 0, 1)$. This semiring only contains the values *true* and $false$ and is used to represent non-quantitative problems.

Tropical $\mathsf{Trop} = (\mathbb{Z} \cup \{\infty\}, \min, +, \infty, 0)$. This semiring is the most common one and is used to assign additive weights—e.g., term sizes and term depth. In this case, we typically consider the order $\preceq \stackrel{\text{def}}{=} \leq$.

Probabilistic $\mathsf{Prob} = ([0, 1], +, \cdot, 0, 1)$. This semiring is used to assign probabilities to terms in a grammar. □

In our framework, we allow synthesis problems to have multiple objectives. Hence, we define a product operation to compose semirings. Intuitively, the following operation composes algebras of semirings to create a pair and applies the operation of each algebra to the corresponding projections of the pair. Similarly, two orders can be composed to create an order over pairs of elements. We propose two such compositions, one which assigns equal weights to the two orders and one which prefers one order over the other (Sorted).

Definition 2 (Products). *Given two algebras* $\boldsymbol{S}_1 = (S_1, \oplus_1, \otimes_1, 0_1, 1_1)$ *and* $\boldsymbol{S}_2 = (S_2, \oplus_2, \otimes_2, 0_2, 1_2)$*, the* product algebra *is the tuple* $\boldsymbol{S}_1 \times_{\boldsymbol{S}} \boldsymbol{S}_2 = (S_1 \times S_2, \oplus, \otimes, (0_1, 0_2), (1_1, 1_2))$ *such that for every* $x_1, x_2 \in S_1$ *and* $y_1, y_2 \in S_2$*, we have* $(x_1, y_1) \oplus (x_2, y_2) \stackrel{\text{def}}{=} (x_1 \oplus_1 x_2, y_1 \oplus_2 y_2)$ *and* $(x_1, y_1) \otimes (x_2, y_2) \stackrel{\text{def}}{=} (x_1 \otimes_1 x_2, y_1 \otimes_2 y_2)$.

Given two partial orders $\preceq_1 \subset S_1 \times S_1$ *and* $\preceq_2 \subset S_2 \times S_2$*, the* Pareto product *of the two orders is defined as the partial order* $\preceq_p = \textsc{par}(\preceq_1, \preceq_2) \subseteq (S_1 \times S_2) \times (S_1 \times S_2)$ *such that, for every* $x_1, x_2 \in S_1$ *and* $y_1, y_2 \in S_2$*, we have* $(x_1, y_1) \preceq_p (x_2, y_2)$ *iff* $x_1 \preceq_1 x_2$ *and* $y_1 \preceq_2 y_2$.

Given two partial orders $\preceq_1 \subset S_1 \times S_1$ *and* $\preceq_2 \subset S_2 \times S_2$*, the* Sorted product *of the two orders is defined as the partial order* $\preceq_s = \textsc{sort}(\preceq_1, \preceq_2) \subseteq (S_1 \times S_2) \times (S_1 \times S_2)$ *such that, for every* $x_1, x_2 \in S_1$ *and* $y_1, y_2 \in S_2$*, we have* $(x_1, y_1) \preceq_s (x_2, y_2)$ *iff* $x_1 \preceq_1 x_2$ *or* $(x_1 = x_2$ *and* $y_1 \preceq_2 y_2)$.

Example 2. The weights in the grammar in Fig. 1 are from the product semiring $\mathsf{Trop} \times_{\mathbf{S}} \mathsf{Trop}$. When using the Pareto partial orders, we have, for example, $(1, 0) \preceq (2, 0)$ and $(1, 0) \preceq (1, 2)$, but $(1, 2)$ is incomparable to $(2, 0)$. When using the Sorted product, we have, for example, $(1, 0) \preceq (1, 2) \preceq (2, 0)$. □

3.2 Weighted Tree Grammars

Since SyGuS defines search spaces using context-free grammars, we propose to extend this formalism with weights to assign costs to terms in the grammar. We focus our attention on a restricted class of context-free grammars called regular tree grammars—i.e., grammars generating regular tree languages—because, to our knowledge, the benchmarks appearing in the SyGuS competition [3] and in practical applications of SyGuS operate over tree grammars. Moreover, it was recently shown that SyGuS problems that are undecidable for context-free grammars become decidable with weighted tree grammars [8].

Trees A *ranked alphabet* is a tuple (Σ, rk_Σ) where Σ is a finite set of symbol and $rk_\Sigma : \Sigma \to \mathbb{N}$ associates a rank to each symbol. For every $m \geq 0$, the set

of all symbols in Σ with rank m is denoted by $\Sigma^{(m)}$. In our examples, a ranked alphabet is specified by showing the set Σ and attaching the respective rank to every symbol as superscript—e.g., $\Sigma = \{+^{(2)}, c^{(0)}\}$. We use T_Σ to denote the set of all (ranked) trees over Σ—i.e., T_Σ is the smallest set such that (i) $\Sigma^{(0)} \subseteq T_\Sigma$, (ii) if $\sigma \in \Sigma^{(k)}$ and $t_1, \ldots, t_k \in T_\Sigma$, then $\sigma(t_1, \cdots, t_k) \in T_\Sigma$. In the following we assume a fixed ranked alphabet (Σ, rk_Σ).

Weighted Tree Grammars. Tree grammars are similar to word grammars but they generate ranked trees instead of words. Weighted tree grammars augment tree grammars by assigning weights from a semiring to trees. They do so by associating weights to productions in the grammar. Weighted grammars can, for example, compute the height of a tree, the number of occurrences of some node in the tree, or the probability of a tree with respect to some distribution In the following, we assume a fixed semiring $(\mathbf{S}, \preceq)$ where $\mathbf{S} = (S, \oplus, \otimes, 0, 1)$.

Definition 3 (Weighted Tree Grammar). *A* weighted tree grammar *(WTG) is a tuple $G = (N, Z, P, \mu)$, where N is a set of non-terminal symbols with arity 0, Z is an axiom with $Z \in N$, P is a set of production rules of the form $A \to \beta$ where $A \in N$ is a non-terminal and β is a tree of $T(\Sigma \cup N)$, and $\mu : P \to S$ is a function assigning to each production a weight from the semiring.*

We can now define the semantics of a WTG as a function $\mathrm{W}_G : T_\Sigma \mapsto S$, which assigns weights to trees. Intuitively, the weight of a tree is $\oplus$-sum of the weight of every possible derivation of that tree in a grammar and the weight of a derivation is the $\otimes$-product of the weights of the productions appearing in the derivation. We use $MS(\beta) = \langle X_1, \ldots, X_k \rangle$ to denote the multi-set of all nonterminals appearing in β and $\beta[t_1/X_1, \ldots, t_k/X_k]$ to denote the result of simultaneously substituting each X_i with t_i in β. Given a derivation $p = A \to \beta$ such that $MS(\beta) = \langle X_1, \ldots, X_k \rangle$, we assume that p is a symbol of arity k. A derivation d starting at non-terminal X is a tree of productions $d \in T(P)$ representing one possible way to derive a tree starting from X. The derivation has to be such that: (i) the root of d is a production of the form $X \to \beta$, (ii) for every node $p = A \to \beta$ in d, if $MS(\beta) = \langle X_1, \ldots, X_k \rangle$, then, for every $1 \leq i \leq k$, the i-th child of p is a production $X_i \to \beta_i$. Given a derivation d with root $p = X \to \beta$, such that $MS(\beta) = \langle X_1, \ldots, X_k \rangle$ and p has children subtrees $d_1, \ldots, d_k$, the tree generated by d is recursively defined as $tree(d) = \beta[tree(d_1)/X_1, \ldots, tree(d_k)/X_k]$. We use $\mathrm{DER}(X, t)$ to denote the set of all derivations d starting at X, such that $tree(d) = t$. The weight $\mathrm{DW}(d)$ of a derivation d is the $\otimes$-product of the weights of the productions appearing in the derivation. Finally, the weight of a tree t is the $\oplus$-sum of the weights of all the derivations of t from the initial nonterminal $\mathrm{W}_G(t) = \bigoplus_{d \in \mathrm{DER}(Z,t)} \mathrm{DW}(d)$. A weighted tree grammar is *unambiguous* iff, for every $t \in T_\Sigma$, there exists at most one derivation—i.e., $|\mathrm{DER}(Z, t)| \leq 1$.

Weighted tree grammars generalize weighted tree automata. In particular, a *weighted tree automaton* (WTA) is a WTG in which every production is of the form $A \to \sigma(T_1, \ldots, T_n)$, where $A \in N$, each $T_i \in N$, and $\sigma \in \Sigma^{(n)}$. Finally, a *tree automaton* (TA) is a WTA over the Boolean semiring—i.e., the TA accepts

all trees with some derivations yielding *true*. Similarly, a *tree grammar* (TG) is a WTG over the Boolean semiring. Given a TA (resp. TG) G, we use $L(G)$ to denote the set of trees accepted by G—i.e., $L(G) = \{t \mid \mathrm{w}_G(t) = true\}$.

Example 3. The weighted grammar in Fig. 1 operates over the semiring $\mathsf{Trop} \times \mathsf{Trop}$, $N = \{\text{Start}, \text{BExpr}\}$, $Z = \text{Start}$, P contains 9 productions, and μ assigns non-zero weights to two of them. □

Aside from being a natural formalism for assigning weights to trees, TGs and WTGs enjoy properties that make them a good choice for our model. First, WTGs (resp. TGs) are equi-expressive to WTAs (resp. TAs) and have logic characterizations [9–11]. Due to this reason, tree grammars are closed under Boolean operations and enjoy decidable equivalence [9]. Second, WTGs enjoy many closure and decidability properties—e.g., given two WTGs G_1 and G_2, we can compute the grammars $G_1 \oplus G_2$ and $G_1 \otimes G_2$ such that, for every f, $\mathrm{w}_{G_1 \oplus G_2}(f) = \mathrm{w}_{G_1}(f) \oplus \mathrm{w}_{G_2}(f)$ and $\mathrm{w}_{G_1 \otimes G_2}(f) = \mathrm{w}_{G_1}(f) \otimes \mathrm{w}_{G_2}(f)$. This operation is convenient for building grammars over product semirings.

3.3 QSyGuS

In this section, we formally define QSyGuS, which extends SyGuS with quantitative objectives. In SyGuS a problem is specified with respect to a background theory T—e.g., linear arithmetic—and the goal is to synthesize a function f that satisfies two constraints provided by the user. The first constraint describes a *functional semantic property* that f should satisfy and is given as a predicate $\psi(f) \stackrel{\text{def}}{=} \forall x.\phi(f, x)$. The second constraint limits the *search space* S of f and is given as a set of expressions specified by a context-free grammar G defining a subset of all the terms in T. A solution to the SyGuS problem is an expression e in S such that the formula $\psi(e)$ is valid.

We augment such a framework in two ways. First, we replace context free grammars with WTGs, which we use to assign weights (from a given semiring) to terms. Second, we augment the problem formulation with constraints over the weight of the synthesized program—i.e., only consider programs of weight greater than 2—and optimization objectives over the same weight—i.e., find the solution of minimal weight. Weight constraints range over the grammar

$$WC := WC \wedge WC \mid WC \vee WC \mid \neg WC \mid w \preceq s \mid s \preceq w \mid w \prec s \mid s \prec w,$$

where w is a special variable and s is an element of the semiring under consideration. Given a constraint $\omega \in WC$, we write $\omega(t)$ to denote the term obtained by replacing w with t in ω.

Definition 4 (QSyGuS). *A* QSyGuS *problem is a tuple* $(T, (\boldsymbol{S}, \preceq), \psi(f), G, \omega, \text{OPT})$ *where:*

- T *is a background theory.*
- $(\boldsymbol{S}, \preceq)$ *is an ordered semiring defining the set of weights and their operations.*

Algorithm 1. QSyGuS synthesis algorithm

```
1: procedure QSyGuS-solve(T, S, ψ, G, ω, opt)
2:     G' ← ReduceGrammar(G, ω)                    ▷ extract grammar satisfying ω
3:     f* ← SyGuS(T, ψ, G')                        ▷ solve corresponding SyGuS problem
4:     if opt = false then return f*
5:     while true do
6:         G' ← ReduceGrammar(G', w ≺ w_G(f*))
7:         f ← SyGuS(T, ψ, G')                     ▷ Try to find better solution
8:         if f = ⊥ then return f*                 ▷ Return the optimal solution
9:         f* ← f
```

- *G is a weighted tree grammar with weights over the semiring $\boldsymbol{S}$ and that only contains terms in T—i.e., $L(G) \subseteq T$.*
- *$\psi(f) \stackrel{\text{def}}{=} \forall x.\phi(f, x)$ is a Boolean formula constraining the semantic behavior of the synthesized program f.*
- *$\omega \in WC$ is a set of constraints over the weight w of the synthesized program.*
- *opt is a Boolean denoting whether the solution has to have minimal weight with respect to $\preceq$.*

A solution to the QSyGuS *problem is a term e such that $e \in L(G)$, $\psi(e)$ is true, and $\omega(\mathrm{w}_G(e))$ is true. If* opt *is true, we also require that there is no g that satisfies the previous conditions and such that $\omega(\mathrm{w}_G(g)) \prec \omega(\mathrm{w}_G(e))$.*

A SyGuS problem is a QSyGuS problem without weight constraints—i.e., $\omega \equiv true$ and $\text{opt} = false$. We denote such problems just as triples $(T, \psi(f), G)$.

Example 4. Consider the QSyGuS problem described in Sect. 2. We already described all the components but ω and opt in the rest of this section. In this example, $\omega = true$ and $\text{opt} = true$ because we want to synthesize the solution with minimal weight.

4 Solving QSyGuS Problems via Grammar Reduction

In this section, we present an algorithm for solving QSyGuS problems (Algorithm 1), which works as follows. First, given a QSyGuS problem, we construct (under certain assumptions) a SyGuS problem for which the solution is guaranteed to satisfy the weight constraints ω (line 2) and use existing SyGuS solvers to find a solution to such a problem (line 3). If the QSyGuS problem requires minimization, our algorithm produces a new SyGuS instance to search for a solution that is better than the previously found one and tries to solve it (lines 6-7). This procedure is repeated until an optimal solution is found (line 8).

4.1 From QSyGuS to SyGuS

The first step of our algorithm is to construct a SyGuS problem characterizing exactly all the solutions of the QSyGuS problem that satisfy the weight

constraints. Given a QSyGuS problem $P = (T, (\mathbf{S}, \preceq), \psi(f), G, \omega, \text{OPT})$, we construct a SyGuS problem $P' = (T, \psi(f), G')$ such that a function g is a solution to the SyGuS problem P' iff g is a solution of $P = (T, (\mathbf{S}, \preceq), \psi(f), G, \omega, false)$, where the optimization constraint has been dropped. We denote the grammar reduction operation as $G' \leftarrow \text{ReduceGrammar}(G, \omega)$.

Base case. First we show how to solve the problem when ω is an atomic formula—i.e. of the form $w \preceq s$, $s \preceq w$, $w \prec s$, or $s \prec w$. We start by showing how to solve the problem for $w \preceq s$ as the construction is identical for the other constraints.

Concretely, we are given a WTG $G = (N, Z, P, \mu)$ and we want to construct a TG $G_{\preceq s} = (N', Z', P')$ such that $t \in L(G_{\preceq s})$ iff $\text{w}_G(t) \preceq s$. In general, it is not possible to perform this construction for arbitrary semirings and grammars. We first present our algorithm and then describe sufficient conditions under which we can ensure termination and correctness.

The idea behind our construction is to introduce new nonterminals in the grammar $G_{\preceq s}$ to keep track of the weight of the trees that can be produced from those nonterminals. For example, a nonterminal pair (X, s') will derive all trees derivable from X using a single derivation of weight s'. Therefore, the set of nonterminals N' is a subset of $N \times S$ (plus an initial nonterminal Z'), where S is the universe of the WTG's semiring. We construct our set of nonterminals N' starting from the leaf productions of G and then recursively explore other productions. At the same time we generate the set of productions P'. Formally, N' and P' are the smallest sets such that the following conditions hold.

1. $Z' \in N'$ (the initial nonterminal).
2. For every production $p \in P$ such that $p = (A \to \beta)$ and $\beta \in T_\Sigma$—i.e., p is a leaf—and $\mu(p) \preceq s$, then $(A, \mu(p)) \in N'$ and $((A, \mu(p)) \to \beta) \in P'$. If $A = Z$, then $Z' \to (A, \mu(p)) \in P'$.
3. For every production $p \in P$ such that $p = (A \to \beta)$, $MS(\beta) = \langle X_1, \ldots, X_k \rangle$, $(X_1, s_1), \ldots, (X_k, s_k) \in N'$ (for some values $s_i \in S$), and $\mu(p) \otimes s_1 \otimes \ldots \otimes s_k = s'$, $s' \preceq s$, then $(A, s') \in N'$, and $((A, s') \to \beta[(X_1, s_1)/X_1, \ldots, (X_k, s_k)/X_k]) \in P'$. If $A = Z$, then $Z' \to (A, s') \in P'$.

Example 5. We illustrate our construction using the grammar in Fig. 1 . Assume the weight constraint is $w \preceq (1, 0)$ and the partial order is built using a Pareto product—i.e., we accept terms with 1 or less if-statements and no plus-statements. Our construction yields the following grammar.

$$
\begin{aligned}
\text{Z'} &::= (\text{Start},1,0) \mid (\text{Start},0,0) \\
(\text{Start},1,0) &::= \text{if}((\text{BExpr},0,0)) \text{ then } (\text{Start},0,0) \text{ else } (\text{Start},0,0) \mid x \mid y \mid 0 \mid 1 \\
(\text{Start},0,0) &::= x \mid y \mid 0 \mid 1 \\
(\text{BExpr},0,0) &::= (\text{Start},0,0) > (\text{Start},0,0) \mid \neg(\text{BExpr},0,0) \mid (\text{BExpr},0,0) \wedge (\text{BExpr},0,0)
\end{aligned}
$$

□

The construction of $G_{\preceq s}$ only terminates for certain semirings and grammars, and only guarantees that individual derivations yield the correct weight—i.e., it does not account for the $\oplus$-sum of multiple derivations.

Example 6. The following WTG over Prob is ambiguous and, if we apply the grammar reduction algorithm for $\omega := w \preceq 0.6$, the resulting grammar will be empty. However, the tree $1 + 1$ has weight $0.9 \preceq 0.6$ ($0.9 \geq 0.6$).

$$\begin{array}{ll} \text{Start} ::= \text{Start} + \text{Start}/0.5 & \text{Expr} ::= \text{Expr} + \text{Expr}/0.4 \\ \quad\quad |x \mid 0 \mid 1 \mid \text{Expr} & \quad\quad |x \mid 0 \mid 1 \end{array}$$ □

We now identify sufficient conditions under which the construction of $G_{\preceq s}$ terminates and is sound. In particular, we start by restricting our attention to unambiguous WTGs, which are the common ones in practice. We use $\text{WEIGHTS}(G) = \{s \mid p \in P \wedge \mu(p) = s\}$ to denote the set of weights used by G and $M_{\mathbf{S},G} = (S', \otimes, 1)$ to denote the submonoid of $\mathbf{S}$ generated by $\text{WEIGHTS}(G)$—i.e., the set of all weights we can generate using $\otimes$ and $\text{WEIGHTS}(G)$.

Theorem 1. *Given an unambiguous WTG G over a semiring $\boldsymbol{S}$ such that $M_{\boldsymbol{S},G} = (S', \otimes, 1)$, and a weight $s \in S$, the construction of $G_{\preceq s}$ terminates if the set $\{s' \mid s' \preceq s \wedge w \in S'\}$ is finite. Moreover, if the set of weights $\text{WEIGHTS}(G)$ is monotonically increasing with respect to $\preceq$—i.e. for every $s \in S$ and $s' \in \text{WEIGHTS}(G)$, $s \preceq s \otimes s'$—then $L(G_{\preceq s})$ contains exactly every tree t such that $\text{W}_G(t) \preceq s$.*

The theorem above also holds for other atomic constraints $w \prec s$, $s \preceq w$, or $s \prec w$ (for these last two, the direction of the monotonicity is reversed). Moreover, in certain cases, even if the construction may not terminate for, let's say $s \preceq w$, it might terminate for the negated constraint $w \prec s$. In such a case, we can use the closure properties of regular tree grammars/automata to construct the reduced grammar for $s \preceq w$ as $G_{\preceq w} = \text{INTERSECT}(G, \text{COMPLEMENT}(G_{\succ w}))$. The same idea can be applied to all atomic constraints.

In practice, the restriction of Theorem 1 holds for grammars that operate over the Boolean and probabilistic semirings, and the tropical semiring only with positive weights. Theorem 1 never holds when $\mathbf{S}$ is the tropical semiring and the grammar contains negative weights. In general, one cannot construct the constrained grammar in this case. However, it is easy to modify our algorithm to work with grammars that do not contain loops—i.e., derivations from a nonterminal to a tree containing the same nonterminal—with negative weights.

Intuitively, when the grammar contains no negative loops, we can find a constant SH such that any intermediate derivation with weight greater than $s+SH$ will never result in tree with weight smaller than s. We use this idea to modify the construction of $G^{\mathsf{Trop}}_{\leq s}$—i.e., $G_{\leq s}$ for Trop—as follows. First, this constant is bounded by ck^{n+1} where c is the absolute value of the smallest negative weight in the grammar, k is the largest number of nonterminals appearing in one grammar production, and $n = |N|$ is the number of nonterminals. Second, in steps 2 and 3 of the construction, a new nonterminal and the corresponding productions are produced if $\mu(p) \leq s + |SH|$ (previously $\mu(p) \leq s$). However, if $A = Z$ in steps 2 and 3, we add a new production $Z' \to (A, s')$ only if $s' \preceq s$.

We now show when this construction terminates and return correct values. Since the tropical semiring combines multiple runs using the min operator, we can *drop* the requirement that the grammar has to be unambiguous.

Theorem 2. *Given a WTG G over* Trop *and a weight $s \in \mathbb{Z}$, the construction of $G^{\mathsf{Trop}}_{\leq s}$ terminates if G contains no loop with cumulative negative weight. Moreover, $G^{\mathsf{Trop}}_{\leq s}$ contains exactly every tree t such that $\mathrm{W}_G(t) \leq s$.*

Composing semirings. We next discuss how Theorem 1 relates to product semirings. Given a grammar $G = (N, Z, P, \mu)$ over a semiring $\mathbf{S}_1 \times_{\mathbf{S}} \mathbf{S}_2$, we use $G^{\mathbf{S}_i}$ to denote the grammar (N, Z, P, μ_i) in which the weight function outputs the corresponding projected weight—i.e., if $\mu(p) = (s_1, s_2)$, then $\mu_i(p) = s_i$.

Let's first consider the case where the product semiring uses a Pareto partial order. In this case, if Theorem 1 holds for each grammar $G^{\mathbf{S}_i}$ and $w_i \preceq_i s_i$, then it holds for G and $(w_1, w_2) \preceq_p (s_1, s_2)$. However, the other direction is not true. Theorem 3 proves this intuition and states that, in some sense, solving Pareto partial orders is easier than solving the individual partial orders.

Theorem 3. *Given an unambiguous WTG G over the semiring $\boldsymbol{S} = \boldsymbol{S}_1 \times_{\boldsymbol{S}} \boldsymbol{S}_2$ with Pareto partial order $\preceq_p = \textsc{par}(\preceq_1, \preceq_2)$ and a weight $s = (s_1, s_2) \in S$, if the constructions $G^{\boldsymbol{S}_1}_{\preceq_1 s_1}$ and $G^{\boldsymbol{S}_2}_{\preceq_2 s_2}$ terminate, then the construction of $G_{\preceq s}$ terminates.*

When we move to Sorted partial order we cannot get an analogous theorem: if Theorem 1 holds for each grammar $G^{\mathbf{S}_i}$ and $w_i \preceq_i s_i$, then it does not necessary hold for G and $(w_1, w_2) \preceq_s (s_1, s_2)$. In particular, if the semiring $\mathbf{S}_2$ is infinite and there exists an $s_1' \prec s_1$, there will be infinitely many elements $(s_1', \_) \prec (s_1, s_2)$. Using this observation, we devise a modified algorithm for reducing grammars with sorted objectives. First, we compute the grammars $G^{\mathbf{S}_1}_{\prec_1 s_1}$, $G^{\mathbf{S}_1}_{=_{s_1}}$, and $G^{\mathbf{S}_2}_{\prec_2 s_2}$. Second, we use WTG closure properties to compute $G_{\preceq_s}(s_1, s_2)$ as the union of $G^{\mathbf{S}_1}_{\prec_1 s_1}$ and $\textsc{intersect}(G^{\mathbf{S}_1}_{=_{s_1}}, G^{\mathbf{S}_2}_{\prec_2 s_2})$.

General formulas. We can now inductively construct the grammar accepting only terms satisfying all constraints in ω. We again use the fact that tree grammars are closed under Boolean operations to compute intersections and unions and correctly characterize all conjunctions and unions appearing in the formulas.

4.2 Finding an Optimal Solution

If our QSyGuS problem does not require minimization—i.e., $\textsc{opt} = \mathit{false}$—the technique presented in Sect. 4.1 can be used to generate an equivalent SyGuS problem $P' = (T, \psi(f), G')$, which can be solved using off-the-shelf SyGuS solvers. In this section, we show how to extend this technique to handle minimization objectives. Our idea is to use SyGuS solvers to find a non-optimal solution for P' and then iteratively refine our grammar G' to search for a better solution. This loop is illustrated in Algorithm 1 (lines 5-9). Given the initial solution f^* to P' such that $\mathrm{W}_G(f^*) = s$, we can construct a new grammar $G_{\prec s}$ and look for a solution with lower weight. If the SyGuS solver we use is sound—it can find a solution if it exists—and complete—it can detect if a solution does not exist—Algorithm 1 terminates with an optimal solution.

In general, the above conditions are too strict and in practice this implies that the algorithm will often not terminate. However, if the SyGuS solver is

sound, the Algorithm 1 will eventually find the optimal solution, but it will not be able to prove that no smaller one exists. In our experiments, we will show that this approach can yield better solutions than those given by vanilla SyGuS solvers even when Algorithm 1 does not terminate.

5 Implementation and Evaluation

First, We extended the SyGuS format with new syntax for expressing QSyGuS problems. Our format supports all semirings presented in Sect. 3.1 as well as additional ones. The format also allows creating tuples of semirings using the product operation described in Sect. 3.1. We augment the original SyGuS syntax to support weights on grammar productions. Weight constraints are added using an SMT-like syntax.

Second, we implemented Algorithm 1 in a tool called QuaSi. QuaSi already interfaces with three SyGuS solvers: CVC4 [6], ESolver [4], and EUSolver [5]. QuaSi supports all the semirings allowed in our format and implements a library for tree automata/grammars and weighted tree automata/grammars operations, as well as several optimizations we did not discuss in the paper. In particular, QuaSi often uses simple grammar reduction techniques to simplify the generated grammars, remove unnecessary productions, and consolidate equivalent ones.

We evaluate QuaSi through the following questions (experiments performed on an Intel Core i7 4.00 GHz CPU with 32 GB/RAM).

Q1 Can QuaSi solve quantitative variants of real SyGuS benchmarks? (Sect. 5.1)
Q2 What is the overhead of synthesizing optimal solutions? (Sect. 5.2)
Q3 How do multiple iterations of Algorithm 1 affect the solution's weight? (Sect. 5.3)
Q4 Can QuaSi solve QSyGuS problems with multiple objectives? (Sect. 5.4)

Benchmarks. We perform our evaluation on 26 quantitative extensions of existing SyGuS competition benchmarks taken from 4 SyGuS benchmark tracks [4]: Hackers Delight, Integers, ICFP and Bitvector. 18 of our benchmarks only use a minimization objective over a single semiring (Table 1), while 8 use a minimization objective (Pareto or Sorted) over a product semiring (Table 2). We select SyGuS benchmarks using the following criteria: (*i*) the benchmark can be solved by either CVC4 [6] or ESolver [4], and (*ii*) the solution is not optimal according to some reasonable metric—e.g., size or number of if statements.

5.1 Effectiveness of QSyGuS Solver

We evaluate the effectiveness of QuaSi on the 18 single-minimization-objective benchmarks. For each benchmark, we run QuaSi using either CVC4 or ESolver as the backend SyGuS solver (we also evaluated QuaSi using EUSolver [5], but, due to its poor performance, we do not report the results). The results are shown in Table 1. The timeout for each iteration of Algorithm 1 is 10 min.

With CVC4, QuaSi terminates with an optimal solution in 9/18 benchmarks, taking less than 5 s (avg: 0.7 s) to solve each sub-problem. In 3 of these cases, the initial solution is already optimal and the second iteration is used to prove optimality. With ESolver, QuaSi terminates with an optimal solution in 8/18 benchmarks, taking less than 7 s (avg: 0.9 s) to solve each sub-problem. In 1 cases, it can find a better solution than the original one, but it cannot prove that the solution is optimal. Overall, by combining solvers, QuaSi can find a better solution than the original SyGuS solution given by one of the two solvers in 9/18 benchmarks. QuaSi cannot improve the initial solution of the linear integer arithmetic benchmarks (`array_search` and `LinExpr_eq1ex`).

Both solvers timeout on large grammars. The grammars in Table 1 are 1 to 2 order of magnitude larger than those in existing SyGuS benchmarks (avg: 224 vs 13 rules) and existing solvers have not yet been optimized for this parameter. In some cases, the solver times out for intermediate grammars that do not contain a solution, but that generate infinitely many terms. In general, existing SyGuS solvers cannot prove unsatisfiability for these types of problems. To answer **Q1**, QuaSi can **solve quantitative variants of 10/18 real SyGuS benchmarks**.

Table 1. Performance of QuaSi. **Time** shows the sequence of times taken to solve individual iterations of Algorithm 1. **Largest** is the size of the largest SyGuS sub-problem. **Grammar Size** is the number of rules in the original grammar.

	Problem	CVC4		ESolver		Grammar
		Time [sec]	Largest	Time [sec]	Largest	Size
Trop	`max_ite(2,3)`	0.1+0.1	42	0.1	42	13
	`max_ite(2,15)`	0.1+0.1	239	0.3	239	13
	`max_ite(3,15)`	0.1+0.1+0.1	238	OOM	238	13
	`max_ite(10,15)`	0.5+0.5+0.9	226	OOM	226	13
	`parity_not`	0.1+TO	301	26.9+TO	43	6
	`max3_ite`	0.1+TO	31	OOM	–	14
	`array_search_3`	0.1+TO	135	TO	–	15
	`array_search_5`	0.1+TO	108	TO	–	16
	`hackers_5`	0.1+0.1	27	0.1+0.1+0.1	35	13
	`hackers_7`	0.1+0.3	35	0.1+0.1+0.2	41	13
	`hackers_17`	0.1+0.7	41	2.8+3.0+1.0	62	13
	`hackers_19`	0.2+TO	174	TO	–	13
	`icfp_7`	0.2+TO	146	TO	–	11
	`LinExpr_eq1ex`	0.7+TO	1717	TO	–	14
Prob	`hackers_2_prob`	0.6+4.1+0.1	95	0.8+0.1+0.2	154	13
	`hackers_5_prob`	0.1+0.9+0.1	96	0.1+0.2+0.1	154	13
	`hackers_7_prob`	0.1+TO	162	0.1+0.1+0.2	212	13
	`hackers_17_prob`	0.1+TO	187	3.4+6.5+OOM	291	13

5.2 Solving Time for Different Iterations

In this section, we evaluate the time required by each iteration of Algorithm 1. Figure 2 shows the ratio of time taken by each iteration with respect to the initial non-quantitative SyGuS solving time. Some of the iterations shown in Fig. 1 do not appear in Fig. 2 since they resulted in no solution—i.e., the initial solution was minimal. CVC4 is typically slower in subsequent iterations and can take up to 10 times the original solving time, while ESolver has comparable runtime to the initial run and is often faster. These numbers are largely due to how the two solvers work: CVC4 is optimized to solve problems where the grammar imposes no restrictions on the structure of the solution, while ESolver performs enumerative search and takes advantage of more restrictive grammars.

One interesting point is the `parity_not` benchmark. ESolver takes 26.9 s to find an initial solution. But, with a weight constraint $w < 11$, an solution can be found in 2.2 s. CVC4 can find the initial solution with weight 11 in 0.1 s but cannot solve the next iteration. We tried using different solvers in different iterations of our algorithm and, in fact, found that, if we use CVC4 to find an initial solution and then ESolver in subsequent iterations with restricted grammars we can fully solve this benchmark in a total of 2.3 s which is much better than the time taken by a single solver. To answer **Q2**, with appropriate choices of solvers **the overhead of synthesizing optimal solutions is minimal**.

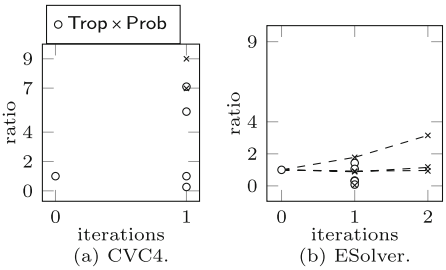

Fig. 2. Solving time across iterations

5.3 Solution Weight Across Iterations

In this section, we present how the weight of the synthesized solutions change across each iteration of Algorithm 1. Figure 3 shows the percentage of weight of solutions synthesized at each iteration with respect to the weight of the initial SyGuS solution. The result shows that we can improve the solutions of CVC4 by 15–25% in one iteration, and the solutions of ESolver by 20–50% when taking one iteration and 50–60% when taking two. The Prob benchmarks, which require two iterations, can be improved more when using ESolver because ESolver tends to synthesize small terms whose probability may also be small. To answer **Q3**, QuaSi can **improve the weights of SyGuS solutions by 20–60%**.

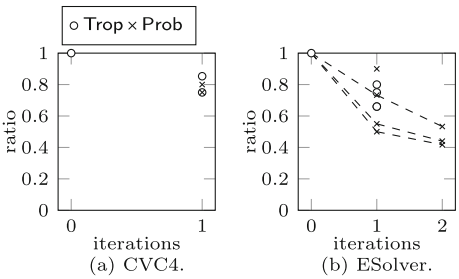

Fig. 3. Solution weight across iterations.

5.4 Multi-objective Optimization

In this section, we evaluate the effectiveness of QuaSi on the 8 benchmarks involving two minimization objectives. The benchmarks consists of two families, 4 for sorted optimization and 4 for Pareto optimization. The sorted optimization benchmarks ask to minimize first the number of occurrences of specified operator (`bvand` in `hacks` and `ite` in `array_search`) and then the size of the solution. The Pareto optimization benchmarks have the same objectives as sorted optimization but here we are synthesizing a Pareto optimal solution instead of sorted optimal one. The results are shown in Table 2. We do not present the results using CVC4 because it cannot solve any of the benchmarks.

The `array_search` times out since it is already hard on a single objective. For the `hackers_5` benchmarks, the initial solution is already optimized for the first objective, so the problem degenerates to the single-objective optimization problem. For the `hackers_7` and `hackers_17`, we present the weights of the intermediate solutions we can see that Pareto and Sorted optimizations yield different solutions. To answer **Q4**, QuaSi can **solve problems with multiple objectives** when the same problems are feasible with a single objective.

Table 2. Performance of QuaSi on multi-objective benchmarks. **Weight** denotes the sequence of weights explored during minimization.

	Problem	**Time** [sec]	**Weight**	**Largest**	**Size**
Trop × Trop	`array_search_sorted`	TO	-	-	15
	`hackers_5_sorted`	0.1+0.1+01	$(0,3) \rightarrow (0,2)$	31	13
	`hackers_7_sorted`	0.1+0.3+0.1	$(1,4) \rightarrow (0,5) \rightarrow (0,3)$	72	13
	`hackers_17_sorted`	0.1+156.1+TO	$(2,5) \rightarrow (1,4) \rightarrow (0,6)$	97	13
	`array_search_pareto`	TO	-	-	15
	`hackers_5_pareto`	0.1+0.1+01	$(0,3) \rightarrow (0,2)$	31	13
	`hackers_7_pareto`	0.1+0.3+0.1	$(1,4) \rightarrow (1,3) \rightarrow (0,3)$	74	13
	`hackers_17_pareto`	0.1+9.1+0.1	$(2,5) \rightarrow (2,4) \rightarrow (1,4)$	54	13

6 Related Work

Qualitative Synthesis. Existing program synthesizers fall in three categories: (*i*) enumeration solvers, which typically output the smallest program [1], (*ii*) symbolic solvers, which reduce the synthesis problem to a constraint solving problem and output whatever program is produced by the constraint solver [21], (*iii*) probabilistic synthesizers, which randomly search the space for a solution and are typically unpredictable [18]. Since the introduction of the SyGuS format [2], these techniques have been used to build several SyGuS solvers that have competed in SyGuS competitions [4]. The most effective ones, which are used in this paper are ESolver a2nd EUSolver [1] (enumeration), and CVC4 [6] (symbolic).

Quantitative synthesis. Domain-specific synthesizers typically employ hard-coded ranking functions that guide the search towards a "preferable" program [17], but these functions are typically hard to write and are decoupled from the functional specification. Unlike QSyGuS, these synthesizers allow arbitrary ranking functions to be expressed in general purpose languages, but typically only support limited grammars for synthesis. Moreover, in many practical applications the ranking functions are very simple. For example, the popular spreadsheet formula synthesizer FlashFill [12] uses a ranking function to prefer small programs with few constants. This type of objective is expressible in our framework.

The Sketch synthesizer supports optimization objectives over variables in sketched programs [20]. This work differs from ours in that sketches are a different specification mechanism from SyGuS. In Sketch the search space is encoded as a program with holes to facilitate synthesis by constraint solving. Translating SyGuS problems into sketches is non-trivial and results in poor performance.

The work closest to ours is Synapse [7], which combines sketching with an approach similar to ours. For the same reasons as for Sketch, Synapse differs from our work because it proposes a different search space mechanisms. However, there are a few analogies between our work and Synapse that are worth explaining in detail. Synapse supports syntactic cost functions that are defined using a decidable theory, and separately from the sketch search space. Synthesis is done using an iterative search where sketches—i.e., set of partial programs with holes—of increasing sizes are given to the synthesizer. At the high level, the intermediate sketches are related to our notion of reduced grammars—i.e., they accept solution of weight less than a given constant. However, while our algorithm generates reduced grammars automatically for a well-defined family of semirings, Synapse requires the user to provide a function for generating the intermediate sketches. Moreover, since Synapse requires cost functions that are defined using a decidable theory, it would not support certain families of costs QSyGuS supports—e.g., the probabilistic semiring.

Koukoutos et al. [15] have proposed the use of probabilistic tree grammars to guide the search of enumerative synthesizers on applications outside of SyGuS. Their algorithm enumerates all terms accepted by the grammar in decreasing probability using a variant of the search algorithm A* and requires the grammar to not contain transitions of weight 1 to avoid getting stuck. Probabilistic tree grammars are a special case of QSyGuS and our algorithm does not impose limitations of what weights can appear in the grammar. Moreover, our algorithm does not require implementing a new solver when changing the cost semiring.

7 Conclusion

We presented QSyGuS, a general framework for defining and solving SyGuS problems in the presence of quantitative objectives over the syntax of the programs. QSyGuS is (*i*) *natural*: requires minimal modification to the SyGuS format, (*ii*) *general*: it supports complex but practical types of weights, (*iii*)

formal: it is grounded in the theory of weighted tree grammars, (*iv*) *effective*: our tool QuaSi can solve quantitative variations of existing SyGuS benchmarks with little overhead. In the future, we plan to extend our framework to handle probabilistic objectives and quantitative objectives over the semantics of the program—e.g., synthesize programs that satisfy most of the specification.

Acknowledgements. The authors were supported by National Science Foundation Grants CCF-1637516, CCF-1704117 and a Google Research Award.

References

1. ESolver. https://github.com/abhishekudupa/sygus-comp14
2. Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: 2013 Formal Methods in Computer-Aided Design (FMCAD), pp. 1–8. IEEE (2013)
3. Alur, R., Fisman, D., Singh, R., Solar-Lezama, A.: Results and analysis of SyGuS-comp 2015. arXiv preprint arXiv:1602.01170 (2016)
4. Alur, R., Fisman, D., Singh, R., Solar-Lezama, A.: Sygus-comp 2016: results and analysis. arXiv preprint arXiv:1611.07627 (2016)
5. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 319–336. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_18
6. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
7. Bornholt, J., Torlak, E., Grossman, D., Ceze, L.: Optimizing synthesis with metasketches. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pp. 775–788. ACM, New York (2016)
8. Caulfield, B., Rabe, M.N., Seshia, S.A., Tripakis, S.: What's decidable about syntax-guided synthesis? CoRR abs/1510.08393 (2015)
9. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2007). http://www.grappa.univ-lille3.fr/tata. Accessed 12 Oct 2007
10. Droste, M., Kuich, W., Vogler, H.: Handbook of Weighted Automata, 1st edn. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01492-5
11. Droste, M., Vogler, H.: Weighted tree automata and weighted logics. Theor. Comput. Sci. **366**(3), 228–247 (2006). Automata and Formal Languages
12. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, 26–28 January 2011, Austin, TX, USA, pp. 317–330 (2011)
13. Gulwani, S.: Programming by examples: applications, algorithms, and ambiguity resolution. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 9–14. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_2
14. Hu, Q., D'Antoni, L.: Automatic program inversion using symbolic transducers. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, 18–23 June 2017, Barcelona, Spain, pp. 376–389 (2017)

15. Koukoutos, M., Raghothaman, M., Kneuss, E., Kuncak, V.: On repair with probabilistic attribute grammars. CoRR abs/1707.04148 (2017)
16. Ngo, V.C., Dehesa-Azuara, M., Fredrikson, M., Hoffmann, J.: Verifying and synthesizing constant-resource implementations with types. In: 2017 IEEE Symposium on Security and Privacy (SP), pp. 710–728, May 2017
17. Polozov, O., Gulwani, S.: Flashmeta: a framework for inductive program synthesis. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, 25–30 October 2015, Pittsburgh, PA, USA, pp. 107–126 (2015)
18. Schkufza, E., Sharma, R., Aiken, A.: Stochastic program optimization. Commun. ACM **59**(2), 114–122 (2016)
19. Singh, R., Gulwani, S.: Predicting a correct program in programming by example. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 398–414. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_23
20. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. In: Proceedings of PLDI 2013, pp. 15–26. ACM, New York (2013)
21. Solar-Lezama, A.: Program sketching. Int. J. Softw. Tools Technol. Transf. **15**(5), 475–495 (2013)