Program Synthesis as Dependency Quantified Formula Modulo Theory[⋆]

Priyanka Golia[1,2], Subhajit Roy[1], and Kuldeep S. Meel[2]

[1] Computer Science and Engineering, Indian Institute of Technology Kanpur, India
[2] School of Computing, National University of Singapore, Singapore

Abstract. Given a specification $\varphi(X, Y)$ over inputs X and output Y, defined over a background theory $\mathbb{T}$, the problem of program synthesis is to design a program f such that $Y = f(X)$ satisfies the specification φ. Over the past decade, syntax-guided synthesis (SyGuS) has emerged as a dominant approach for program synthesis where in addition to the specification φ, the end-user also specifies a grammar L to aid the underlying synthesis engine. This paper investigates the feasibility of synthesis techniques without grammar, a sub-class defined as $\mathbb{T}$-constrained synthesis.
We show that $\mathbb{T}$-constrained synthesis can be reduced to DQF($\mathbb{T}$), i.e., to the problem of finding a witness of a Dependency Quantified Formula Modulo Theory. When the underlying theory is the theory of bitvectors, the corresponding DQF(BV) problem can be further reduced to Dependency Quantified Boolean Formulas (DQBF). We rely on the progress in DQBF solving to design DQBF-based synthesizers that outperform the domain-specific synthesis techniques, thereby positioning DQBF as a core representation language for program synthesis.

1 Introduction

In this work, we focus on a key problem in programming languages, program synthesis, and investigate its relationship to DQBF and its generalization, Dependency Quantified Formulas modulo Theory, henceforth referred to as DQF($\mathbb{T}$). Given a specification $\varphi(X, Y)$ over the set of inputs X and the set of outputs Y, the problem of program synthesis is to synthesize a program f such that $Y = f(X)$ would satisfy the specification φ. A significant breakthrough was achieved with the introduction of Syntax-Guided Synthesis (SyGuS) formulation wherein in addition to φ, the input also contains a grammar of allowed implementations of f. The grammar helps to constrain the space of allowed implementation of f, and therefore, it also allows development of techniques that can efficiently enumerate over the grammar. Often, the end user is primarily concerned with any function that can be expressed using a particular theory $\mathbb{T}$. For the sake of clarity, we use the term $\mathbb{T}$-*constrained* synthesis to characterize such class of synthesis problems. $\mathbb{T}$-constrained synthesis is a subclass of SyGuS. Of particular interest is the recent work in the development of specialized algorithms focused on $\mathbb{T}$-constrained synthesis, e.g., counterexample-guided quantifier instantiation algorithm in [7].

[⋆] The open source tool is available at https://github.com/meelgroup/DeQuS. To be appeared at IJCAI 2021. Full paper available at https://arxiv.org/abs/2105.09221

The primary contribution of our work is establishing a connection between Theory-constrained synthesis and DQF($\mathbb{T}$). In particular, our work makes the following contributions:

From $\mathbb{T}$-constrained synthesis to DQF($\mathbb{T}$) We present an reduction of $\mathbb{T}$-constrained synthesis to DQF($\mathbb{T}$). DQF($\mathbb{T}$) lifts the notion of DQBF from the Boolean domain to general Theory $\mathbb{T}$. We view the simplicity of the reduction from $\mathbb{T}$-constrained synthesis to DQF($\mathbb{T}$) as a core strength of the proposed approach.

Efficient $\mathbb{T}$-constrained synthesizers for $\mathbb{T}$=bitvectors The reduction to DQF($\mathbb{T}$) opens up new directions for further work. As a first step, we focus on the case when the $\mathbb{T}$ is restricted to bitvector theory, denoted by BV. We observe that the resulting DQF(BV) instances can be equivalently specified as a DQBF instance. We demonstrate that our reduction to DQBF allows us to simply plug-in the state of the art DQBF solvers [9,8].

2 Synthesis via DQF($\mathbb{T}$)

We propose the reduction of $\mathbb{T}$-constrained synthesis to DQF($\mathbb{T}$), i.e, to the problem of finding a witness of a dependency quantified formula modulo theory.

We refer to the function and its (ordered) list of arguments at an invocation (within φ) as its *call signature*. The set of all call signatures of a function symbol f in φ is referred by $\mathsf{CallSigns}(f)$. Note that the number of invocations of a function may not match $|\mathsf{CallSigns}(f)|$. For example, the following formula $\varphi : \forall a, b, c\ \exists f\ f(a, b) \wedge f(b, c) \wedge f(b, a) \wedge f(a, b)$, has 4 invocations of f while CallSigns(f) = $\{\langle a, b\rangle, \langle b, c\rangle, \langle b, a\rangle\}$. Note that $\langle a, b\rangle$ and $\langle b, a\rangle$ are considered as two different CallSigns of f.

$\mathbb{T}$-Constrained Synthesis to DQF($\mathbb{T}$): As remarked in Section 1, a key strength of the reduction is its simplicity. Algorithm 1 formalizes the desired reduction of φ to DQF($\mathbb{T}$) formulation where φ is a specification over the vocabulary of background theory $\mathbb{T}$ with a set of typed function symbols $\{f_1, f_2, \dots f_m\}$ such that for all f_i, $|\mathsf{CallSigns}(f_i)| = 1$. The important point to note is that the Henkin quantifiers must be carefully constructed so that each f_i depends only on the set of variables that appear in its argument-list.

Now, let us turn our attention to the case when there exist a function f_i such that $|\mathsf{CallSigns}(f_i)| > 1$. In such cases, we pursue a Ackermannization-style technique that transforms φ into another specification $\hat{\varphi}$ such that every function f_i in $\hat{\varphi}$ has $|\mathsf{CallSigns}(f_i)| = 1$ (Algorithm 2). Note that this transformation allows the subsequent use of Algorithm 1 with $\hat{\varphi}$ to complete the reduction to DQF($\mathbb{T}$). The proposed transformations in Algorithm 2 are linear in the size of the formula like the transformation introduced in [5], however Algorithm 2 introduces lesser number of new variables.

The essence of Algorithm 2 is captured in the following two transformations:

(Line 5) We substitute instances of every call signature of f_i with fresh function symbols f_i^j (that corresponds to the j^{th} call signature of f_i). This reduces the formula from multiple-callsign to a single-callsign instance.

(Line 6) Introduction of an additional constraint for each f_i that forces all the functions f_i^j (introduced above) to mutually agree on every possible instantiation of arguments. Specifically, it introduces a fresh function symbol $f_i^{l_i}$ and a set of fresh variables

Algorithm 1: Reducing single-callsign instance φ to DQF($\mathbb{T}$)

```
Input: A background theory $\mathbb{T}$, a set of typed function symbols $\{f_1, f_2, \ldots f_m\}$, a
       specification $\varphi$ over the vocabulary of $\mathbb{T}$
1 Let $X = \bigcup_{f_i} \{h \mid h \in \mathsf{CallSigns}(f_i)\}$
2 Substitute every invocation of $f_i$ with a fresh variable $y_i$ in $\varphi$
3 Define $H_i = Set(h)$ as $\{h | h \in \mathsf{CallSigns}(f_i)\}$
Output: $\forall X \exists^{H_1} y_1. \exists^{H_2} y_2 \ \ldots \exists^{H_m} y_m \varphi(X, Y)$
```

Algorithm 2: Reducing multiple-callsign to single-callsign instance

```
Input: A background theory $\mathbb{T}$, a set of typed function symbols $\{f_1, f_2, \ldots f_m\}$, a
       specification $\varphi$ over the vocabulary of $\mathbb{T}$ such that $\ell_i = |\mathsf{CallSigns}(f_i)|$
1 for $i = 1$ to $m$ do
2     if $|\mathsf{CallSigns}(f_i)| > 1$ then
3         Add a fresh (ordered) set of variables $Z_i$ such that $|Z_i| = |\mathsf{CallSigns}(f_i)[0]|$
4         for $j \in [0 \ldots (\ell_i - 1)]$ do
5             Replace every $f_i$ whose $args(f_i) = \mathsf{CallSigns}(f_i)[j]$ with $f_i^j$
6             Add constraint $(args(f_i^j) = Z_i) \rightarrow f_i^j(args) = f_i^{\ell_i}(Z_i)$ to $\varphi$
7         $\mathsf{CallSigns}(f_i) \leftarrow \mathsf{CallSigns}(f_i) \cup \{Z_i\}$
Output: A set of typed function symbols $\{f_1^0, f_1^2, \ldots f_1^{\ell_1}, \ldots, f_m^0 \ldots f_m^{\ell_m}\}$, a
        specification $\hat{\varphi}$ over the vocabulary of $\mathbb{T}$ such that $\forall i, j$ we have
        $|\mathsf{CallSigns}(f_i^j)| = 1$
```

$z_1^i, \ldots, z_n^i \in Z_i$ such that, for all $args(f_i^j)$ argument lists, $(args(f_i^j) = Z_i) \implies f_i^j(args) = f_i^{\ell_i}(Z_i)$, where $j \in [0 \ldots l_{i-1}]$.

When $\mathbb{T}$ is bitvector (BV): When the specification $\varphi(X, Y)$ is in BV. If $\varphi(X, Y)$ is a multiple-callsign instance, we use Algorithm 2 to covert it to a single-callsign instance $\hat{\varphi}(\hat{X}, \hat{Y})$. We then use Algorithm 1 to generate the DQF(BV) instance of $\hat{\varphi}(\hat{X}, \hat{Y})$ as $\forall \hat{X} \exists^{H_1} \hat{y}_1. \ldots \exists^{H_m} \hat{y}_m \hat{\varphi}(\hat{X}, \hat{Y})$. Finally, we solve the DQF(BV) instance by compiling it down to a DQBF instance, thereby allowing the use of off-the-shelf DQBF solvers.

As the first step to DQBF compilation, we perform *bit-blasting* over $\hat{\varphi}$ to obtain $\hat{\varphi}'$.

$$\forall \hat{X} \exists^{H_1} \hat{y}_1 \ \ldots \exists^{H_m} \hat{y}_m \hat{\varphi}(\hat{X}, \hat{Y}) \ \equiv \ \forall X' \exists^{X'} V. \ \exists^{H_1'} Y_1' \ldots \exists^{H_m'} Y_m' \varphi'(X', Y') \quad (1)$$

where, X', Y_i', H_i' are the (bit-blasted) *sets* of propositional variables mapping to the bitvector variables $\hat{X}, \hat{y}_i, H_i$ respectively. Furthermore, V is the set of auxiliary variables introduced during bit-blasting. The auxiliary variables can be allowed to depend on all the input variables X'. Our current framework simply employs off-the-shelf SMT solvers for the bit-blasting. As the formula on the right-hand side in Eq. 1 is an instance of DQBF, we can simply invoke an off-the-shelf *certifying DQBF* solvers to generate the Henkin functions for Y'.

3 Experimental Evaluation

The objective of our experimental evaluation was to study the feasibility of solving BV-constrained synthesis via the state-of-the-art DQBF solvers. To this end, we perform an evaluation over an extensive suite of 645 general-track bitvector (BV) theory benchmarks from SyGuS competition 2018, 2019. We used CVC4 [7], EUSolver [1], ESolver [10] as SyGuS-tools. CVC4 was also used in its BV-constrained version. We used state-of-the-art DQBF (QBF) solvers CADET [6], Manthan [2], DepQBF [4], DCAQE [9] and DQBDD [8].

Table 1 represents the instances solved by the virtual best solver for SyGuS, BV constrained, and DQBF tools.

Table 1: Number of SyGuS solved using different techniques. Timeout 900s.

	Total	SyGuS-tools	BV-constrained	DQBF-based
SyGuS	645	513	606	610

As shown in Table 1, with syntax guided synthesis, we could synthesize the functions for 513 out of 645 SyGuS instances only, whereas, with BV-constrained synthesis, we could solve 606 such instances. Surprisingly, BV-constrained synthesis performs better than the syntax-guided synthesis. Table 1 also shows that the DQBF based synthesis tools perform similar to BV-constrained synthesis tools for SyGuS instances; this provides strong evidence that the general purpose DQBF solvers can match the efficiency of the domain specific synthesis tools.

We defer to technical report [3] for detailed experiment results.

4 Conclusion

Syntax-guided synthesis has emerged as a dominant paradigm for program synthesis. Motivated by the impressive progress in automated reasoning, we investigate the usage of syntax as a tool to aid the underlying synthesis engine. To this end, we formalize the notion of $\mathbb{T}$-constrained synthesis, which can be reduced to DQF($\mathbb{T}$). We then focus on the special case when $\mathbb{T} = BV$. The corresponding BV-constrained synthesis can be reduced to DQBF, highlighting the importance of the scalability of DQBF solvers.

References

1. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: Proc. of TACAS (2017)
2. Golia, P., Roy, S., Meel, K.S.: Manthan: A data-driven approach for Boolean function synthesis. In: Proc. of CAV (2020)
3. Golia, P., Roy, S., Meel, K.S.: Program synthesis as dependency quantified formula modulo theory (2021), https://arxiv.org/abs/2105.09221
4. Lonsing, F., Biere, A.: DepQBF: A dependency-aware QBF solver. Proc. of JSAT (2010)
5. Rabe, M.N.: A resolution-style proof system for DQBF. In: Proc. of SAT (2017)
6. Rabe, M.N.: Incremental determinization for quantifier elimination and functional synthesis. In: Proc. of CAV (2019)
7. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: Proc. of CAV (2015)
8. Sìč, J.: Satisfiability of DQBF Using Binary Decision Diagrams. Master's thesis (2020), https://is.muni.cz/th/prexv/
9. Tentrup, L., Rabe, M.N.: Clausal abstraction for DQBF. In: Proc. of SAT (2019)
10. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M., Alur, R.: TRANSIT: specifying protocols with concolic snippets. ACM SIGPLAN Notices (2013)