Programmable Program Synthesis

Loris D'Antoni, Qinheping Hu, Jinwoo Kim, Thomas Reps

University of Wisconsin-Madison, Madison, USA

Abstract. Program synthesis is now a reality, and we are approaching the point where domain-specific synthesizers can now handle problems of practical sizes. Moreover, some of these tools are finding adoption in industry. However, for synthesis to become a mainstream technique adopted at large by programmers as well as by end-users, we need to design programmable synthesis frameworks that (*i*) are not tailored to specific domains or languages, (*ii*) enable one to specify synthesis problems with a variety of qualitative and quantitative objectives in mind, and (*iii*) come equipped with theoretical as well as practical guarantees. We report on our work on designing such frameworks and on building synthesis engines that can handle program-synthesis problems describable in such frameworks, and describe open challenges and opportunities.

1 Introduction

1.1 A Synthesis Tale

Monica, a software engineer, is trying to write a program for transforming data she has stored in an array of integer numbers. Monica needs to zero-out all the negative entries from the array (they represent irrelevant data points) and add 10 to all the positive entries (this is a normalization step needed in Monica's API). Of course, Monica is a great engineer and she could write this program herself, but since Monica knows that similar problems arise often in her company (i.e., reformatting arrays to match certain APIs), Monica decides to try out this new thing everyone is talking about: *program synthesis*.

Monica wants a tool that takes as input some examples of the desired transformation and a set of operators the program can use, and magically outputs the intended program. In fact, Monica already has an input, a unit test, that she wants to process using her newly synthesized program: $[-1, 2, 3, 10, 31, -14, -11]$, for which the output should be $[0, 12, 13, 20, 41, 0, 0]$.

Monica also knows that the final program will look like a loop that iterates over the input array arr, which leads her to develop the grammar in Figure 1. Monica thinks this grammar is general enough that it will cover a reasonable range of programs for similar tasks but limited enough that it will not result in spurious programs that overfit too much to the examples.

Quickly, Monica discovers that using program synthesis is not so straightforward. There are so many different tools! And they all take different kinds of inputs. After a bit more research, Monica decides to go for one of the many

$$
\begin{aligned}
\mathit{Start} &\to x \texttt{ = len(arr) - 1; while x>=0 do } S \\
S &\to \texttt{arr[}E\texttt{] = arr[}E\texttt{]} + E \mid \texttt{arr[}E\texttt{] = } E \mid \\
&\quad\; x \texttt{ = } E \mid S\texttt{; } S \mid \texttt{if arr[x]>0 then } S \texttt{ else } S \\
E &\to \texttt{0} \mid \texttt{1} \mid \texttt{x} \mid E \texttt{ + } E \mid E \texttt{ - } E
\end{aligned}
$$

Fig. 1: The grammar G_{ex} Monica has in mind for synthesizing programs that iterate over an input array (*Start* is the starting nonterminal). G_{ex} is general enough to cover most programs that iteratively normalize entries in an array.

tools, UltraSynth™, and encodes her problem. UltraSynth is written in a C-like language and Monica has mostly programmed in Python for her job. However, Monica decides to give UltraSynth a try and after a few days of learning the ins and outs of UltraSynth, she finally manages to encode her transformation-synthesis problem in UltraSynth. To achieve her goal, Monica had to tweak a bit what the grammar looks like to provide it to UltraSynth, which only accepted grammars without unbounded recursion (i.e., without infinitely many terms) and had to encode the examples in a way that was accepted by the tool.

The time has come and Monica manages to run UltraSynth on an instance of the synthesis problem. UltraSynth outputs the program in Figure 2b, which is correct on the example. However, this program is needlessly large and contains many unneeded operations.

```
x = len(arr)-1
while x>=0 do
  if arr[x]>0 then
    arr[x] = arr[x]+10
  else
    arr[x] = 0
  x = x-1
```

(a) A possible ideal solution for Monica's problem.

```
x = len(arr)-1
while x>=0 do
  if arr[x]>0 then
    arr[x] = arr[x]+1-1...+1
  else
    arr[x] = 0
  x = x-1
```

(b) A solution for Monica's problem synthesized by UltraSynth. The sequence of ± 1s on line 3 adds up to $+10$.

Fig. 2: Two possible solutions for Monica's synthesis problem.

Monica has already invested a lot of time in learning UltraSynth, so she tries to figure out a way to avoid such problematic programs. Monica astutely realizes that the needless computations in Line 3 of Figure 2b are due to repeated applications of the minus operator. Monica would like to ask UltraSynth to synthesize the program that contains as few minus operators as possible, but UltraSynth does not support a way to "prefer" one possible program over another. To bypass this limitation, Monica decides to remove the production $E \to E$ - E in order to suppress these programs.

Monica reruns UltraSynth after removing $E \to E$ - E from the grammar, and to her surprise, UltraSynth continues to run for hours and eventually times out without providing a solution. After investigating the matter, Monica finds out that she has made a mistake and disallowed too many programs—there is no longer a valid solution to the synthesis problem because without subtraction, the variable x cannot be decremented in line 5. UltraSynth was unable to report, or even detect this simple mistake—Why is it so difficult to program a synthesizer and why can't synthesis tools detect the simplest of mistakes?

Monica has finally had enough of synthesis. She goes back to her daily routine and just writes the 6-line piece of code that applies the transformation she intended (Figure 2a).

1.2 Programmable Synthesis Frameworks

The story of Monica is a common one in program synthesis, where most of the recent focus has been on solving problems rather than building general algorithms, tools, and methodologies. Existing synthesis frameworks are not **programmable** as they lack at least one of the following properties:

Domain-Agnostic. Existing synthesis ideas and algorithms have been introduced with specific domains in mind and are hard to apply to arbitrary synthesis problems. The languages used to specify synthesis problems are therefore domain-specific, and often fail to abstract the logical requirements of the synthesis problem. In our example, Monica had to look for a specific tool that accepted programs of the kind she was interested in. Moreover, she was not permitted to refine the specification to add a quantitative objective she had in mind (minimizing the number of minus operators).

Solver-Agnostic. Different synthesis tools are typically not interchangeable because their underlying solvers solve different types of problems. Even when two solvers can in principle solve the same types of problems, they typically cannot be interchanged or combined because they typically use drastically different formats written in different languages (e.g., Racket [28] vs C [27]). For example, when Monica found out that UltraSynth was not working as expected, she could not easily try another tool to see if that tool was better.

This state of affairs is unfortunate because synthesis is very general; if synthesis were easier to use, it would benefit many domains. The potential generality, which is currently held back by the need for better support for usability, underscores the need to answer the following question:

Can we make synthesis more programmable?

In this paper, we present the steps we have undertaken in the direction of making synthesis more programmable, including some of the challenges that we faced, and some of the opportunities that the work has opened.

2 An Overview of Programmable Program Synthesis

The goal of enlarging the scope of synthesis has focused our attention on the need to have a framework in which synthesis problems can be addressed. By a *framework*, we mean the conceptual underpinnings that allow one to build tools to automate the creation of solutions for problems in some domain, in this case, program-synthesis problems. The canonical example is how the theory of parsing underlies the `yacc` tool [13], which automates the construction of parsers. For instance, consider the problem that `yacc` addresses:

- An instance of a parsing problem, Parse(L,s), has two parameters: L, a context-free language; and s, a string to be parsed. String s changes more frequently than language L.
- Context-free grammars are a formalism for specifying context-free languages.
- Create a tool that implements the following specification:
  - Input: a context-free grammar that describes language L.
  - Output: a parser, `yyparse()`, such that invoking `yyparse()` on s computes Parse(L,s).

One consideration for building a framework is the existence of a well-defined "engine" (or collection of engines) for performing the desired task—in this case, parsing s with respect to L, once both L and s are at hand. `Yacc` supports just a single engine, which parses a string with respect to a grammar that is LALR(1). In principle, `yacc` could have been a more general tool by having it perform various tests on L to determine what grammar family L belongs to (e.g., LALR(1), LR(1), LL(1), LL(*)), and then emitting a parser that makes use of an appropriate parsing algorithm for that family, falling back on Generalized LR parsing [19] in case L is not in one of the specialized families supported.

Another aspect illustrated by `yacc` is that the parameters to the problem have different "binding times". In this case, string s changes more frequently than language L—i.e., L is bound early, and s is bound late. The framework implementation can exploit the known value of the early-bound parameter to create a more efficient implementation. In the case of `yacc`, it compiles L to tables used by a table-driven LALR(1) parsing algorithm.

2.1 Why isn't Existing Work in Synthesis Programmable?

There do exist synthesis tools (mostly, solver-aided languages [27,28]) that allow one to control some aspects of a synthesis problem in a programmable fashion. However, the nature of existing synthesis tools also forces an association between how a synthesis problem is written and how it is solved. For instance, in §1, the fact that solvers are tightly coupled to some specification language prevented Monica from trying out a different tool after UltraSynth produced an inadequate answer. The current state of program-synthesis tools is depicted in Fig. 3.

This situation is in direct conflict with the principles articulated at the end of §1, namely, that a user should be able to program the various aspects of a

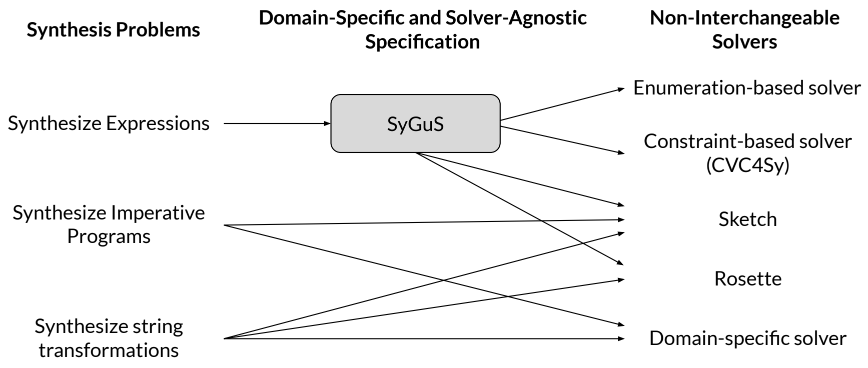
Synthesis Problems
Domain-Specific and Solver-Agnostic Specification
Non-Interchangeable Solvers
Synthesize Expressions
SyGuS
Synthesize Imperative Programs
Synthesize string transformations
Enumeration-based solver
Constraint-based solver (CVC4Sy)
Sketch
Rosette
Domain-specific solver

Fig. 3: Program synthesis today, where the lack of separation between specification and solver causes a user to have to encode a problem multiple times to use different tools.

synthesis problem using a formalism that is both (i) *domain-agnostic* and (ii) *solver-agnostic*. The first property addresses generality: the formalism should be powerful enough to capture a wide variety of synthesis problems (e.g., SQL, regular expressions, and imperative programs). The second property opens the door for synthesis-problem specifications to be fed—possibly after a compilation/translation step—to different specialized solvers, or to multiple solvers with different capabilities.

Another example that one may consider a programmable framework is Syntax-Guided Synthesis (SyGuS) [1], which is a successful synthesis framework targeted at expressions. The defining characteristic of SyGuS, compared to other synthesis approaches such as solver-aided languages, is that it allows one to write synthesis problems in a completely logical format.

Example 1. Consider the simple problem of synthesizing the maximum max of two input variables, x and y. There are two parts to a SyGuS problem: a syntactic part, written as a context free grammar such as the example G_S below:

$$G_S ::= \mathit{Start} \rightarrow \texttt{x} \mid \texttt{y} \mid \mathit{Start} \texttt{ + } \mathit{Start} \mid \texttt{if x<y then } \mathit{Start} \texttt{ else } \mathit{Start}$$

and the specification part, which is written as a Boolean formula ψ_S:

$$\psi_S \equiv \forall x, y. max(x,y) \geq x \wedge max(x,y) \geq y \wedge (max(x,y) = x \vee max(x,y) = y)$$

A SyGuS problem is simply the pair $sy = (G_S, \psi_S)$, where a solution to the SyGuS problem sy is a term $t \in L(G)$ such that ψ_S holds. For example, the following term is a solution for the function max in the problem we just described:

`if x<y then y else x`.

The advantage of such a logic-based formalism is that it achieves a separation from solver and specification, which allows SyGuS to be *solver-agnostic*. Several different SyGuS solvers have been developed (e.g., [26,7,4,21]), many of which use drastically different internal algorithms that have different strengths for solving different kinds of problems. Moreover, a user of SyGuS need not consider the differing input languages or characteristics of these solvers, and instead can encode their problem just once in the SyGuS format to have access to all the different solvers.

While SyGuS achieves—and shows the benefits of—solver-agnosticity, it *fails to achieve domain-agnosticity* because the framework is targeted specifically at expressions. For example, consider the scenario from §1: Monica would be unable to encode her problem in SyGuS, because the grammar G_{ex} in Figure 1 contains a production with a while loop, and loops, which require a custom semantics, cannot be expressed in any decidable theory—a key restriction of SyGuS. SyGuS also does not allow one to express intent outside of the behavioral specification ψ, which would have prevented Monica from trying to optimize the program obtained from UltraSynth in Figure 2b.

All in all, the current state of program synthesis is an unsatisfactory mess, as depicted in Figure 3. There are multiple non-interoperable solvers with different input languages, targeting different synthesis domains with varying degrees of overlap. SyGuS, by virtue of solver-agnosticity, provides a unified approach to synthesizing expressions, which forms the basis of multiple solvers. However, while SyGuS is a bright spot, it fails to be general: it does not cope with (1) the *variety of domains* used in synthesis, required to deal with arbitrary languages (e.g., SQL, regular expressions, and imperative programs), and (2) the *variety of collateral considerations* that arise for different domains (e.g., types, quantitative objectives, and probabilities).

2.2 What does a Programmable Synthesis Framework look like?

Our vision of programmable synthesis can be summed up as follows:

$$\begin{gathered}\text{programmable synthesis}\\ == \\ \text{easily instantiable, domain-agnostic, solver-agnostic synthesis framework.}\end{gathered}$$

In contrast with Figure 3, what we would like to have is depicted in Figure 4, where both user and solver work with a unified general format, regardless of domain or solving technique. Such an approach would allow one to specify a synthesis problem once and for all, without having to worry about the underlying solving strategy. To achieve this goal, it is necessary to distill out the essence of many program-synthesis problems into a specification formalism that is ground in formal methods (e.g., automata and logic) and is agnostic to any specific domain of application. This degree of abstraction also opens the opportunity to lift certain synthesis algorithms and ideas to a higher level that makes these algorithms reusable across different tools. Our framework can then interface to different solving tools (backend solvers) in a way that allows one to easily swap one solver for another, or to use multiple solvers in tandem. If our vision is achieved, the capabilities that would be available to tool designers—discussed in greater detail in §5—would allow synthesis tools to be created that have the kind of flexibility that Monica expected and needed in §1.1.

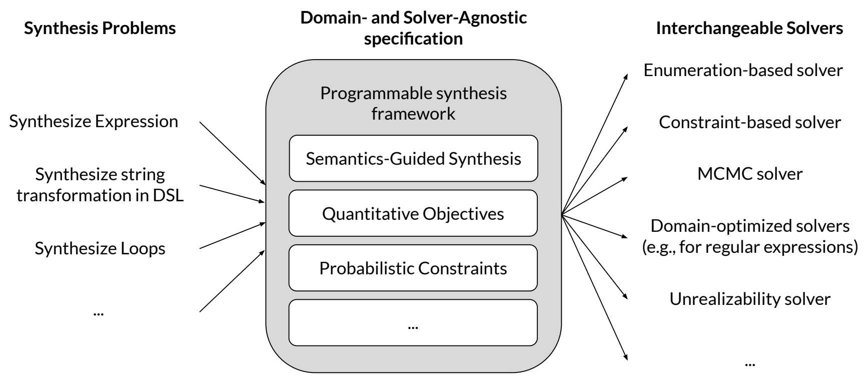
Synthesis Problems
Domain- and Solver-Agnostic specification
Interchangeable Solvers
Synthesize Expression
Synthesize string transformation in DSL
Synthesize Loops
...
Programmable synthesis framework
Semantics-Guided Synthesis
Quantitative Objectives
Probabilistic Constraints
...
Enumeration-based solver
Constraint-based solver
MCMC solver
Domain-optimized solvers (e.g., for regular expressions)
Unrealizability solver
...

Fig. 4: Programmable program synthesis, where a synthesis problem with arbitrary constraints can be written once and for all in a general format, which can then be dispatched to compatible solvers.

Let us now be more concrete about the requirements for such a framework for synthesis. Following the pattern for `yacc` given above, a framework for synthesis could follow a similar scheme:

– An instance of a synthesis problem $\text{Synthesize}(\mathcal{L}, [\![\cdot]\!]_{\mathcal{L}}, \varphi)$ has three parameters: $\mathcal{L}$, a formal language; $[\![\cdot]\!]_{\mathcal{L}}$, a semantics to ascribe to $\mathcal{L}$; and φ, a behavioral specification for some desired member of $\mathcal{L}$. The behavioral specification φ changes more frequently than $\mathcal{L}$ and $[\![\cdot]\!]_{\mathcal{L}}$.
– Let F_{syntax} and $F_{\text{semantics}}$ be appropriate formalisms for specifying $\mathcal{L}$ and $[\![\cdot]\!]_{\mathcal{L}}$, respectively.
– Create a tool that implements the following specification:
  - Input: an F_{syntax} specification of a language's syntax, and an $F_{\text{semantics}}$ specification of the language's semantics.
  - Output: a function $\text{Synth}_{\mathcal{L},[\![\cdot]\!]_{\mathcal{L}}}(\cdot)$ that takes φ as input and computes $\text{Synthesize}(\mathcal{L}, [\![\cdot]\!]_{\mathcal{L}}, \varphi)$.

To be even more concrete, F_{syntax} could be a regular-tree grammar [5],[1] and $F_{\text{semantics}}$ would be defined over the grammar in a compositional manner, production by production. What we have called collateral considerations (types, quantitative objectives, probabilities, etc.) would be handled as part of the F_{syntax} or $F_{\text{semantics}}$ specifications, depending on the issue at hand. For instance, constraints on program behavior, such as refinement types [24], minimizing/bounding evaluation resources usage [15,11], and probabilistic behavior [22], are semantic concerns that would be part of $F_{\text{semantics}}$. Other considerations would be part of F_{syntax}, such as bounds on the use of syntactic constructs [12],

[1] The grammar would also be equipped with production-by-production pretty-printing rules to specify how to convert a tree to its textual representation.

or the use of probabilistic generative models of syntactic structures [3,17]. For instance, for these two issues, one could weight the productions of the grammar with values from a semiring, and place a (possibly learned) distribution on the productions, respectively.

The scheme in the box above would allow us to meet the goals of being both domain-agnostic and solver-agnostic,[2] as long as (*i*) the formalisms for F_{syntax} and $F_{\text{semantics}}$ are sufficiently powerful to qualify as "domain-agnostic," and (*ii*) specifications in these formalisms can be analyzed and broken down into components that can be farmed out to existing solvers (or perhaps to new implementations of the kinds of algorithms used in existing solvers).

Who benefits from such a framework? The existence of a domain- and solver-agnostic framework benefits two parties: (*i*) users of synthesis tools such as Monica, and (*ii*) designers of synthesis tools, such as the team behind UltraSynth. Both scenarios can be illustrated by making an analogy with LLVM [20]—which provides an intermediate representation for compilation that is similarly both domain- and solver-agnostic. Users of LLVM, which are front-end language designers, benefit from two facts: (*i*) that the LLVM IR is rich enough to support the range of features their language might have, and (*ii*) that once their language is compiled down into LLVM IR, the entire library of LLVM IR optimizations is accessible to them. Similarly, a programmable synthesis framework benefits users in two ways: (*i*) by supporting the full range of features that may be required for a synthesis problem, and (*ii*) by putting multiple solvers within reach for problems written in the framework. Additionally, a well-defined framework also facilitates *reuse* of problem components: for example, Monica can reuse G_{ex} for synthesizing other array transformations.

On the other hand, backend optimization designers of LLVM benefit from the fact that once their optimization is written in LLVM, all LLVM users may easily access those optimizations if need be. Similarly, tool designers for a programmable synthesis framework rest easy knowing that once their tool supports the framework, those who need it will find it accessible and easy to use—regardless of what internal techniques they decide to use. Note that while the framework intends to be general, tools that interface with the framework can choose to be selective in the problems they support—it is up to the users, or perhaps the framework designers, to match a problem with an appropriate solver (similar to how language designers mix and match backend optimizations for their language in LLVM). In addition, advances at the framework-level—such as

[2] We also acknowledge that even the scheme given above, which was modeled on the one for `yacc`, is open to revision. In particular, the additional degree of parameterization for synthesis ($\mathcal{L}$, $[\![\cdot]\!]_{\mathcal{L}}$, and φ) opens the door for a variety of alternatives, based on different "binding times" for $\mathcal{L}$, $[\![\cdot]\!]_{\mathcal{L}}$, and φ. For instance, a solver that uses different abstract domains as part of a refinement-based search strategy [29] would have $\mathcal{L}$ and φ fixed, but vary $[\![\cdot]\!]_{\mathcal{L}}$. Similarly, when one has quantitative syntactic objectives [12], the solver would carry out its search with φ fixed, $\mathcal{L}$ varying, and $[\![\cdot]\!]_{\mathcal{L}}$ induced as $\mathcal{L}$ changes.

the development of meta-algorithms, as illustrated in §4.2—instantly benefit all tools that support the framework.

This paper. New technical challenges, as well as new opportunities, come along with our broader goals. In this paper, we present some of the work that we have done toward building the kind of framework sketched out above.

Specifying Programmable Synthesis Problems (§3) Semantics-guided synthesis (SemGuS) is our proposed framework that allows a user to provide both the syntax and semantics for the constructs in the language over which programs are to be synthesized. We show how SemGuS can easily be extended with quantitative objectives for specifying when a synthesized program is "good" according to a certain metric—e.g., the program should be of minimal size or should maximize a certain outcome.

Solving Programmable Synthesis Problems (§4) We present solvers that can tackle problems specified in the SemGuS framework. We also present a meta-solver that can be combined with other SemGuS solvers to support quantitative objectives. Because our framework does not impose solver-specific restrictions on how synthesis problems are programmed, our solvers can prove unrealizability—i.e., whether a synthesis problem has no solution—of many complex synthesis problems with infinite search spaces.

These steps are just the beginning of what we expect to be a multi-year journey into designing a framework that achieves our goals, and solvers for such a framework. We discuss some of the open challenges and opportunities in §5.

3 Programmable-Synthesis Specifications

Designing synthesis frameworks that are *programmable* requires one to formally abstract the essence of how one specifies different program-synthesis problems. While we do not claim to have developed a completely unified framework that can capture all synthesis problems yet, in this section we present two ideas for programming many practical synthesis problems: (*i*) SemGuS, a framework that uses logic and formal methods to make the search space and specifications of all synthesis problems easy to program in arbitrary domains (§3.1), and (*ii*) an extension of SemGuS that allows one to specify *quantitative objectives* over the syntactic structure of a synthesized program (§3.2).

3.1 Semantics-Guided Synthesis

Existing work on program synthesis [1] typically identifies two main components to a synthesis problem: (*i*) a *search space* of candidate programs, which is in essence a small programming language, and (*ii*) a *behavioral specification*, which describes what the synthesized program should do. A programmable synthesis framework must represent (at the very least) these two components in a domain- and solver-agnostic way. Take the syntax-guided synthesis (SyGuS) framework, for example: SyGuS achieves solver-agnosticity by representing the search space as a regular tree grammar, and the specification as a Boolean formula in a decidable background theory.

Then why is SyGuS, and this particular combination of representations, unable to achieve domain-agnosticity? The syntactic component of SyGuS—the grammar—actually does achieve some degree of domain-agnosticity, in the sense that one is free to define a language of one's own. However, SyGuS requires that the specified grammar be contained within a fixed background theory, which are terms with a pre-defined fixed and standardized *semantics*. While this design choice makes the solutions to SyGuS problems easy to verify (using an SMT solver), it limits the *programmability of the search space.*

For example, let us reconsider the example in §1. If Monica attempted to write her example as a SyGuS problem, she would have been unable to use loops because loops are not part of the supported background theory. What if Monica wanted a solution that operates over a DSL, or had some pre-defined components that she wanted to use (like $len(arr)$)? What if Monica wanted to synthesize regular expressions, or some other programs with relatively non-standard semantics?

One can intuitively understand these scenarios as synthesis problems over different programming languages (search spaces)—a DSL, library functions, regular expressions. To support different programming languages, a synthesis framework needs more than the ability to accept a syntax, it needs the ability to accept a semantics for a language as well. Therefore, developing a programmable synthesis framework capable of supporting all these scenarios requires designing a solver-agnostic way of specifying the semantics of such arbitrary programming languages. SyGuS has shown that regular tree grammars are an effective formalism for programming the syntax of a search space; we extend this with a formalism to program the semantics of the search space as well, which, to achieve true domain-agnosticity, need not be constrained to a fixed background theory.

Semantics as Constrained Horn Clauses. Our solution to this challenge is the Semantics-Guided Synthesis (SemGuS) framework [14], which allows users to customize the syntax and semantics of the search space. To see how SemGuS supports programmable semantics, let us consider the production $Start \rightarrow$ `while x>=0 do` S from Figure 1 as an example. This production is a while loop, and part of the semantics for a term produced by this production can be expressed using the inference rule below[3] (where Γ represents a state that maps variables to integer values):

$$\frac{[\![\texttt{x>=0}]\!](\Gamma) = \texttt{True} \quad [\![s]\!](\Gamma) = \Gamma_1 \quad [\![\texttt{while x>=0 do } s]\!](\Gamma_1) = \Gamma_2}{[\![\texttt{while x>=0 do } s]\!](\Gamma) = \Gamma_2} \tag{1}$$

Such semantics are supported in the SemGuS framework by expressing the inference rule in Eqn. (1) as a Constrained Horn Clause (CHC). CHCs are logical formulas, and more precisely, they are implications where one is only allowed to

[3] A similar rule must be added for the case in which the guard evaluates to false.

have a single relation in the conclusion, and a conjunction of relations along with one constraint in the premise:

Definition 1 (Constrained Horn Clauses.). *A* Constrained Horn Clause *is a first-order formula of the form*

$$\forall \overrightarrow{x}, \overrightarrow{x_1}, \ldots, \overrightarrow{x_n}.(\phi \wedge R_1(\overrightarrow{x_1}) \wedge \cdots \wedge R_n(\overrightarrow{x_n}) \implies H(\overrightarrow{x})),$$

where ϕ is a constraint over some background theory that may contain variables from $\overrightarrow{x}, \overrightarrow{x_1}, \ldots, \overrightarrow{x_n}$, and $R_1, \ldots, R_n$ and H are uninterpreted relations.

In SemGuS, search spaces are represented as regular tree grammars, where productions have associated semantics. In Eqn. (1), the semantics of a term `x>=0` is represented using the semantic function $[\![\cdot]\!]$. SemGuS, assumes that each nonterminal N appearing in the grammar has a corresponding logical relation $\mathtt{sem}_N$, which we refer to as the *semantic relation*, that represents the behavior of the semantic function $[\![\cdot]\!]$ in Eqn. (1). For example, the expression $[\![s]\!](\Gamma) = \Gamma_1$ from Eqn. (1) can be translated into the relation $\mathtt{sem}_S(\langle s, \Gamma\rangle, \Gamma_1)$.

Example 2 (Semantic Rules as CHCs). The following CHC captures how one would express in SemGuS the semantics of the production *Start* $\rightarrow$ `while x>=0 do` S shown in Eqn. (1):

$$\frac{\Gamma[\mathtt{x}] \geq 0 \quad \mathtt{sem}_S(\langle s, \Gamma\rangle, \Gamma_1) \quad \mathtt{sem}_{Start}(\langle \mathtt{while\ x>=0\ do}\ s, \Gamma_1\rangle, \Gamma_2)}{\mathtt{sem}_{Start}(\langle \mathtt{while\ x>=0\ do}\ s, \Gamma\rangle, \Gamma_2)} \tag{2}$$

One can read Eqn. (2) as the following implication:

$$\begin{gathered}\mathtt{sem}_S(\langle s, \Gamma\rangle, \Gamma_1) \wedge \mathtt{sem}_{Start}(\langle \mathtt{while\ x>=0\ do}\ s, \Gamma_1\rangle, \Gamma_2) \wedge \Gamma[\mathtt{x}] \geq 0 \implies \\ \mathtt{sem}_{Start}(\langle \mathtt{while\ x>=0\ do}\ s, \Gamma\rangle, \Gamma_2)\end{gathered} \tag{3}$$

Eqn. (3) is a CHC where $\mathtt{sem}_{Start}$ and $\mathtt{sem}_S$ are relations, and $\Gamma[\mathtt{x}] \geq 0$ corresponds to the first-order constraint ϕ.

SemGuS allows one to specify multiple such CHCs[4] for each *production* in the grammar. CHCs are the logical formalism of choice for expressing these semantics in a language-agnostic way, which are an intuitive and expressive format.

The SemGuS *Framework.* Once a user has understood how to define a semantics for their grammar, a SemGuS problem then can be specified simply as a synthesis problem over a grammar equipped with such a semantics.

Definition 2 (SemGuS). *A* SemGuS *problem over a theory $\mathcal{T}$ is a tuple $sem = (G_{[\![\cdot]\!]}, \psi(x, f(x)))$, where:*

- *G is a regular tree grammar equipped with the semantics $[\![\cdot]\!]$,*
- *$\psi(x, f(x))$ is a Boolean formula over the theory $\mathcal{T}$, that serves as the behavioral specification,*

[4] The ability to define multiple semantic rules for a production is useful for productions such as while loops, which are commonly equipped with two rules that describe looping and loop termination.

– *f is a free second-order variable that serves as the function to be synthesized.*

*A **solution** to the* SemGuS *problem sem is a term $s \in L(G_{[\![\cdot]\!]})$ such that $\psi(x, [\![s]\!](x))$ holds.*

Example 3 (Monica's Synthesis Problem in SemGuS*).* Consider the synthesis problem Monica had in §1. Let $G_{ex[\![\cdot]\!]}$ be the grammar G_{ex} from Figure 1, equipped with semantic rules such as the one defined in Eqn. (2). Let $\mathsf{E} = \{[-1, 2, 3, 10, 31, -14, -11]\}$, the input array Monica considered for her task. Let $\psi(arr, f(arr))$ be a formula over the theory of arrays and CLIA describing what it means for the program f to be correct on an input arr:

$$\psi(arr, f(arr)) \equiv \bigwedge_{0 \le i < len(arr)} f(arr)[i] = ITE(arr[i] > 0, arr[i] + 10, 0).$$

Then $sem_{ex} = (G_{ex[\![\cdot]\!]}, \bigwedge_{arr \in E} \psi(arr, f(arr)))$ is a SemGuS problem defined over a background theory of arrays and CLIA—the behavioral specification requires that the final program satisfies all the examples in E.[5] Moreover, sem_{ex} is written in a completely logical format, and is thus not tied to a specific tool like UltraSynth and can be dispatched to multiple backend solvers (assuming tooling) as Monica pleases.

The ability to customize the semantics for a language in a framework allows that framework to support a plethora of different synthesis problems. One can define synthesis problems over regular expressions, domain-specific languages, imperative languages, or any other language that has a semantics definable as CHCs within the framework, all of which can be tested using different solvers utilizing different strategies.

Example 4 (Regular Expressions Synthesis in SemGuS*).* Synthesis problems over regular expressions can be expressed succinctly in SemGuS. The grammar of regular expressions can be captured with the following grammar, where c is a character and ϕ the empty set:

$$R \to c \mid \epsilon \mid \phi \mid R + R \mid R \cdot R \mid R^*$$

Using CHCs, one can also naturally express the semantics of terms $r \in L(R)$. For example, the semantics of Kleene star can be given as the following two CHCs:

$$\frac{}{\mathtt{sem}_R(r^*, \epsilon)} \qquad \frac{\mathtt{sem}_R(r, s_1) \quad \mathtt{sem}_R(r^*, s_2) \quad s = s_1 s_2}{\mathtt{sem}_R(\langle r^* \rangle, s)}$$

The rules are based on the expansion $r^* \to \epsilon + r \cdot r^*$: the first rule lets r^* accept ϵ, and the second rule accepts a string s by finding two substrings s_1, s_2, such that s_1 is accepted by r, s_2 is accepted by r^*, and the concatenation $s_1 \cdot s_2$ is equal to s. The specification of the problem can then use expressions of the

[5] In this example, one could have used a formula simply describing the input/output examples instead of a more complex logical formula. We chose the latter option to illustrate how the behavioral specification can involve terms in interesting theories—e.g., CLIA and arrays.

form $\texttt{sem}_R(r, s)$ and $\neg\texttt{sem}_R(r, s)$ to denote whether an example s is positive or negative, respectively.

3.2 Adding Quantitative Syntactic Objectives

In the example discussed in §1.1, the original synthesis problem Monica posed to the solver was under-constrained and caused the underlying tool to synthesize an undesirable solution that contained unnecessary operations. While the logical-specification mechanism 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, a programmable synthesis framework should provide a way to prefer one to the other—i.e., the user of the framework should be able to describe a quantitative objective. In this section, we show how the formal foundations of SemGuS (i.e., the use of grammars and logic) allow us to easily extend the framework to incorporate quantitative objectives over the syntax of the synthesized program. The ideas we present were originally described in the context of SyGuS [12]; here we show how they can also be applied to SemGuS.

Adding Quantitative Objectives using Weighted Grammars. Recall that a SemGuS problem is given along with a *regular tree grammar* specifying the search space. In our running example, Monica would like to synthesize a program that has few occurrences of the minus operator. A natural way to express this intent is allowing Monica to tag productions involving such an operator with a cost, let's say 1. Our quantitative extension of SemGuS builds on this intuition and allows users to add weights/costs to productions in the grammar. This extension leads to a well-studied formalism, weighted tree grammars, keeping the SemGuS framework general. Intuitively, a *weighted tree grammar* is a grammar in which each production p has an associated weight/cost $\mu(p)$.

Intuitively, the weight of a derivation tree is the *sum* of the weights of all productions.[6] For simplicity, in this paper, we assume that the domain of weights is the natural numbers, and that their sum is the usual application of the $+$-operator. We use $\mathrm{W}_G(t)$ to denote the weight of a term t with respect to the weighted grammar G.

With the weights specified by the weighted grammars, users can specify quantitative objectives as constraint objectives and optimization objectives. A constraint objective is a predicate $\omega(v)$ over a numerical variable v; we say that a term t satisfies the constraint objective if $\omega(\mathrm{W}_G(t))$ holds. An optimization objective is a flag $\textsc{opt} \in \{\texttt{True}, \texttt{False}\}$ indicating whether we want to minimize the weight of the solution.

[6] Weights have to come equipped with operators that tell us how to combine weights of individual productions to obtain the weights of terms. Formally, the weights must be from a semiring; we refer the reader to the original work on this topic [12] for details.

Example 5. Recall that in the example introduced in §1, Monica wants to avoid redundant occurrences of the minus (`-`) operator. To express this intent in SemGuS, Monica can utilize the following weighted grammar.

$$\begin{aligned}
\mathit{Start} &\rightarrow x\ \texttt{= len(arr) - 1; while x>=0 do}\ S \\
S &\rightarrow \texttt{arr[}E\texttt{] = arr[}E\texttt{]} + E \mid \texttt{arr[}E\texttt{] =}\ E \mid \\
&\quad\ x\ \texttt{=}\ E \mid S\texttt{;}\ S \mid \texttt{if arr[x]>0 then}\ S\ \texttt{else}\ S \\
E &\rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{x} \mid E\ \texttt{+}\ E \mid E\ \texttt{-}\ E/1
\end{aligned}$$

In the weighted grammar, only the rule $E \rightarrow E$ - E is assigned the weight 1. All other rules are assigned the weight 0 (omitted for readability). The weight of a term t with respect to this grammar is the number of occurrence of the minus operator in t. If Monica wants to restrict the number of occurrences of the minus operators to be less than 5, she can use the constraint objective $\omega(v) = v < 5$. Furthermore, if she want to minimize the occurrences of the minus operator, she can set the flag OPT to `True`.

To summarize, a SemGuS problem with quantitative syntactic objectives is a tuple $sem = (W_{[\![\cdot]\!]}, \psi(x, f(x)), \omega, \textsc{opt})$ where $W_{[\![\cdot]\!]}$ is a weighted grammar with a corresponding semantics, ψ is a Boolean formula like before, ω is the constraint objective, and the flag OPT is the optimization objective. The goal is to find a solution that not only satisfies the specification ψ, but also the quantitative objective ω, and is of minimal cost if OPT is set to `True`.

Quantitative syntactic objectives are useful in applications such as programming by examples [10] and program repair [6], where it is desirable to produce small programs with fewer constants, because such programs are more likely to generalize to examples and test cases outside of the set of examples given by the user. When allowing real-valued weights, syntactic objectives can be also used to find the most likely solution with respect to a given probability distribution. We can assign productions weights that represent their probabilities; the weight of a candidate solution is its likelihood.

4 Programmable-Synthesis Solvers

While a programmable synthesis framework as discussed in §3 is certainly desirable, it is of little practical use if one is unable to solve the problems that are written in such a framework. In this section, we show that SemGuS problems can be solved practically. We first describe two general solving techniques for SemGuS (§4.1) and then present new algorithmic solving techniques enabled by the SemGuS framework (§4.2).

4.1 General Solving Procedures for SemGuS Problems

We start off by presenting two solving procedures for general SemGuS problems we implemented as a tool, rooted in strategies commonly used in existing program synthesizers: enumeration (used in the tool MESSY-Enum) and constraint solving (used in the tool MESSY). Specifically, we will be considering

SemGuS-with-examples problems: SemGuS problems where the specification is given in terms of a finite set of examples E. An algorithm for solving SemGuS-with-examples problems can be combined with counterexample-guided inductive synthesis (CEGIS) [27], which generates counterexamples in case a synthesized answer does not meet the general specification, to iteratively increase the example set E and eventually obtain a correct program.

MESSY-Enum: A Basic Enumerator for SemGuS Problems. Because SemGuS also relies on a grammar to specify the syntax of valid terms, like SyGuS, one can employ a simple enumerator that generates terms of increasing size from the grammar and test the enumerated terms against the behavioral specification. With SemGuS, a term (representing a program) cannot be executed directly, because the semantics to ascribe to it has been specified in the semantic specification. However, because the semantics is specified with CHCs, the term can be executed with a level of interpretation supplied by an off-the-shelf CHC solver. Therefore, MESSY-Enum employs an off-the-shelf CHC solver such as [18] to check if the CHCs are consistent with the specification.[7]

Concretely, given a term t_e to test, one can use the following CHC to check whether t_e meets the specification:

$$\frac{\bigwedge_{e_i \in \mathsf{E}} \mathtt{sem}_{Start}(\langle e_i, t_e \rangle, o_i)}{\mathit{Realizable}}\ \mathsf{Query} \tag{4}$$

The Query rule in Eqn. (4) exactly encodes the specification as a CHC: it asks whether the semantics of t_e computed by $\mathtt{sem}_{Start}$ is consistent with the set of input-output examples E. If so, the conclusion *Realizable* is provable using the existing set of CHCs—i.e., t_e is a solution to the synthesis problem.

Because we cannot directly execute candidate terms and instead rely on CHC solvers (which may be treated as a blackbox), it is difficult to employ common enumeration optimizations, such as behavioral equivalence caching, or equality saturation. Developing an enumeration-based solver capable of utilizing these ideas would require generating an explicit and efficiently executable interpreter from the given semantics, which is an interesting research challenge and future direction that we discuss in §5.

MESSY: SemGuS Problem Solving as CHC-Solving. MESSY-Enum uses a CHC solver to check whether an enumerated term t_e is consistent with the specification or not—however, CHC solvers are also capable of automatically searching for terms that satisfy the specification, as well. Our next solver, MESSY, takes advantage of this fact by expressing *both* the syntax of the search space and the semantics using CHCs. Once the entire search space is modeled this way, one can then slightly modify the Query rule to accommodate this change and directly use a CHC solver to solve the entire SemGuS problem. In

[7] One can treat CHC solving as akin to a proof search, where the objective is to prove that a specific query holds (in this case, *Realizable* from Eqn. (4)) using the provided CHCs.

essence, Messy reduces solving the SemGuS problem into finding a configuration of variables for which the set of CHC rules (containing syntax, semantics, and specification) is valid—similar to how constraint-based methods in existing synthesizers reduce the synthesis problem to one of solving a set of constraints.

Example 6 (MESSY Encoding). We show how the syntax and semantics used in the production $\mathit{Start} \rightarrow$ `while x>=0 do` S from Figure 1 can be captured using CHCs. This production states that one can obtain a syntactically valid term `while x>=0 do` $s \in L(\mathit{Start})$ for the nonterminal *Start*, given a valid term $s \in L(S)$. Eqn. (5) encodes this idea as a CHC using the *syntax relations* syn_S, and $\mathsf{syn}_{\mathit{Start}}$, which capture whether the supplied arguments are valid terms that may be derived from the corresponding nonterminals S, and *Start*.

$$\frac{\mathtt{syn}_S(s)}{\mathtt{syn}_{\mathit{Start}}(\mathtt{while\ x>=0\ do}\ s)} \tag{5}$$

Because the syntax relations provide a way to guarantee that a term t is a valid term in the syntax of a SemGuS problem, one can rewrite the Query rule from Eqn. (4) to use this relation instead of an explicitly enumerated term t_e.

$$\frac{\mathtt{syn}_{\mathit{Start}}(t) \quad \bigwedge_{e_i \in \mathsf{E}} \mathtt{sem}_{\mathit{Start}}(\langle e_i, t\rangle, o_i)}{\mathit{Realizable}}\ \mathsf{Query} \tag{6}$$

The new Query rule in Eqn. (6) has the term t as a free variable—i.e., proving *Realizable* amounts to finding a term $t \in L(\mathit{Start})$ that is consistent on the input-output examples. A CHC solver presented with this rule, in tandem with the syntax and semantic rules, will then attempt to find a configuration of t such that *Realizable* holds. If the solver can prove that the premises of Equation Eqn. (4) hold, then the term t is a solution to the SemGuS problem.

One of the advantages of using such a CHC-based method is when dealing with cases where there is no answer to the synthesis problem, i.e., when there exists no t such that *Realizable* holds. In this case, the SemGuS problem contains *no* answer satisfying the specification within its search space; we say that such a problem is *unrealizable.* Proving unrealizability is something that many existing solvers fail to consider, but is important: for example, Monica would not have had to wait for several hours after modifying the grammar in §1 if her solver had been able to show that the problem was unrealizable.

4.2 Meta Algorithms for Solving SemGuS Problems

Now that we have shown how to build solvers for general SemGuS problems (that do not involve quantitative objectives), we turn to 'meta'-algorithms for solving SemGuS problems, which are 'meta' in the sense that they (i) may be used atop any general SemGuS solver, (ii) generate modified SemGuS problems (rather than solutions) that can be easier to solve than the original SemGuS problem or can be used to solve SemGuS problems with quantitative objectives.

The key component behind these meta-algorithms is the customizability of the search-space description in SemGuS.

A Meta Solver for Quantitative Objectives. We first present an algorithm for solving SemGuS problems with quantitative objectives [12]—i.e., where productions in the grammar have weights. We assume, for simplicity, that the only quantitative objective is to find the program of *least cost* that satisfies the specification. The idea of the algorithm is to iteratively reduce the SemGuS problem with a quantitative objective to a sequence of SemGuS problems without quantitative objectives, which are used to iteratively find a solution that has least cost—i.e., at each step of the sequence the cost of the solution is improved.

The algorithm operates as follows. Initially, we are given a SemGuS problem *sem* with a weighted grammar W (we omit the semantic information for brevity) and with the minimization objective OPT set to *true*.[8] The first step of the algorithm is to construct an unweighted grammar G^W by merely erasing all the weights in W. We can now use any SemGuS solver to solve the resulting SemGuS problem and obtain a term t_0. This term will have a weight c according to the weighted grammar W, but it might not be the term of least cost that satisfies the specification. Our algorithm therefore tries to find out whether a solution with a lower weight exists, and accordingly constructs an (unweighted) grammar $G^W_{<c}$ such that a term t is accepted by the grammar $G^W_{<c}$ if and only if the weight of t according to W is less than c. When the weights are natural numbers, this construction is always possible [12]. We now have again an unweighted grammar, and we can use a SemGuS solver to solve the resulting problem. This procedure can be repeated until no better solution exists.

Example 7. Consider the weighted grammar W we presented in Ex. 5. In particular, let us focus our attention on the following subset of productions that involve non-zero weights:

$$E \to \texttt{0} \mid \texttt{1} \mid \texttt{x} \mid E \texttt{ + } E \mid E \texttt{ - } E/1$$

The grammar $G^W_{<3}$, which accepts all terms of weight less than 3 is as follows:

$$\begin{aligned} E &\to E_2 \mid E_1 \mid E_0 \\ E_2 &\to E_1 \texttt{ - } E_0 \;\mid\; E_0 \texttt{ - } E_1 \\ E_1 &\to E_0 \texttt{ - } E_0 \\ E_0 &\to \texttt{0} \mid \texttt{1} \mid \texttt{x} \mid E_0 \texttt{ + } E_0 \end{aligned}$$

Intuitively, each non-terminal E_i produces all and only terms with exactly i minus operators.

The meta solver for quantitative objectives shows how using a solver-agnostic specification formalism—i.e., grammars—enables algorithms that operate at the specification level and can be reused across multiple solvers.

[8] For simplicity, we assume no further quantitative objectives are present, but the general case can be handled using similar ideas [12].

Underapproximating Semantics with SemGuS. The previous section showed how the programmability of the search-space syntax (i.e., the grammar) allows us to design meta-algorithms to solve SemGuS problems involving quantitative objectives. In this section, we show how the programmability of the search-space *semantics* can be used to build meta-algorithms that can make synthesis faster. The key idea is to generate "simpler" variants of the original SemGuS problem that use an *underapproximating semantics*, where an underapproximating semantics is defined as a subset of the original semantics that must be precise on the subset on which it is defined.

Definition 3. *For a grammar G equipped with a semantics $[\![\cdot]\!]$, we say $[\![\cdot]\!]^\flat$* underapproximates $[\![\cdot]\!]$ *on G, or that $[\![\cdot]\!]^\flat$ is an* underapproximating semantics *for G with respect to $[\![\cdot]\!]$, if for every term $t \in L(G)$, every state Γ, and every value v on which $[\![\cdot]\!]^\flat$* is defined, *$[\![t]\!]^\flat(\Gamma, v) = [\![t]\!](\Gamma, v)$.*

One easy way to underapproximate a semantics is to simply "eliminate" certain operators from a grammar by not defining semantic rules for them. However, the concept of underapproximation need not be bounded to eliminating operators from a grammar—it may have a fully semantic meaning instead, for example, a bound on the number of possible loop iterations. The key intuition is that underapproximation is sound for use in synthesis—if a term t is the answer to a synthesis problem sy, sy actually does not need to contain any syntax or semantics outside of what is used to define and compute t. (In contrast, overapproximation is sound for proving unrealizability.)

Example 8. Recall, once again, the synthesis problem Monica has in §1. The grammar G_{ex} of Figure 1 contains a while loop, which has a complex semantics that can be expensive to compute and, most importantly, allows nonterminating behavior. Most existing synthesizers [27,28] explicitly prohibit nontermination by only considering finitely many unrollings for loops (because most answers to a synthesis problem will indeed terminate).

Fortunately, Monica knows that on her example $[-1, 2, 3, 10, 31, -14, -11]$, the loop should iterate no more than 7 times to process every element of the array. Monica may then choose to supply the synthesizer with an underapproximating semantics that limits the number of loop iterations to 7, which could greatly reduce the amount of computation the synthesizer must perform—for example, a naive enumerator might get stuck on a nonterminating loop when using the precise semantics, while terminating quickly when using the underapproximating semantics. Such a semantics can be expressed easily by adding a loop counter c to the semantics of loops given in Eqn. (2), yielding the following CHC:

$$\frac{c \geq 0 \quad \Gamma[\mathtt{x}] \geq 0 \quad \mathtt{sem}_S(\langle s, \Gamma, c-1\rangle, \Gamma_1) \quad \mathtt{sem}_{Start}(\langle \mathtt{while\ x>=0\ do}\ s, \Gamma_1, c-1\rangle, \Gamma_2)}{\mathtt{sem}_{Start}(\langle \mathtt{while\ x>=0\ do}\ s, \Gamma, c\rangle, \Gamma_2)} \tag{7}$$

Setting $c = 7$ in the Query rule now ensures that loops run at most 7 iterations.

Abstract Semantics with SemGuS. Similar to how we used underapproximating semantics to find solutions to a SemGuS problem, abstract (overappoximating) semantics can be used to prove that a SemGuS problem is unrealizable.

Definition 4. *For a grammar G equipped with a semantics $[\![\cdot]\!]$, we say $[\![\cdot]\!]^{\#}$ is an* abstract semantics *for G with respect to $[\![\cdot]\!]$ if there exists an abstraction function α and a concretization function γ, such that for all $t \in L(G)$, if $[\![t]\!](\Gamma, v)$ holds, then $[\![t]\!]^{\#}(\alpha(\Gamma), \alpha(v))$ holds, and $\Gamma \in \gamma(\alpha(\Gamma))$, $v \in \gamma(\alpha(v))$, i.e., α and γ form a Galois connection.*

In contrast to underapproximating semantics, abstract semantics are sound when used to prove *unrealizability*—i.e., that a synthesis problem has no solution that satisfies its specification within its search space. Consider the use of abstract interpretation in program analysis: abstract interpretation is most often used to prove that a program cannot reach a certain set of bad states, while often being unable to guarantee that a program will produce a specific value, due to lack of precision. Similarly, an abstract semantics will often be unable to guarantee that a synthesized program satisfies the specification, due to lack of precision—but it can guarantee that all programs in the search space will *never* be able to produce a certain set of values, which can be used to prove unrealizability.

Example 9. Consider the scenario from §1, in which Monica removed subtraction from her grammar in an attempt to simplify the synthesized program. The removal of subtraction made the problem unrealizable—and UltraSynth ran for hours on end because it could not prove that this was the case. While proving unrealizability can be very difficult in general, a solver capable of reasoning about abstract domains and semantics could have utilized an (abstract) semantic rule such as Eqn. (8) below:

$$\frac{\texttt{sem}_E(\langle e_1, \Gamma\rangle, \{pos\}) \quad \texttt{sem}_E(\langle e_2, \Gamma\rangle, \{pos\})}{\texttt{sem}_E(\langle e_1 + e_2, \Gamma\rangle, \{pos\})} \tag{8}$$

Eqn. (8) is defined on the abstract domain $\{pos, zero, neg\}$—corresponding to positive, zero, and negative values—and captures the fact that the sum of two positive numbers will always be positive. This rule will be able to prove that G_{ex} without subtraction will *never* be able to modify x to a negative value, and thus that no program in the search space will terminate (leading to unrealizability).

Unrealizability is a property that is ignored by many current synthesizers, but it is a very important property nonetheless. One practical way to think about unrealizability is as a sanity check, like a type system: the fact that a synthesis problem provided by an end user is unrealizable means that the synthesis problem is malformed in the sense that the user has got some of their specifications wrong. Similar scenarios happen daily with ordinary programming, and we expect them to happen with synthesis as well—thus, it is desirable that synthesizers be able to detect these problems, and report them early on if possible, without running indefinitely, as in §1. Unrealizability also has applications in computing optimal solutions, as in §4.2: unrealizability given a grammar with a lower weight bound ensures that the current solution is optimal.

5 The Future of Programmable Synthesis and SemGuS

We hope we have convinced the reader that synthesis could use more programmability, and that SemGuS addresses many of the programmability issues of existing synthesis work. But what lies ahead? How can we make programmable synthesis truly practical? In this section, we first outline some of the steps we are undertaking to answer this question (§5.1).

More importantly, we would also like to emphasize that *the vision of programmable program synthesis can only be realized through a community effort*. We will conclude this section with ideas to involve the synthesis community to help us realize our vision (§5.2).

5.1 What are we Working on Next?

In this section, we present some of the directions our group is pursuing in extending SemGuS to richer objectives and building better solvers for it. We also describe some open problems related to SemGuS.

Interfacing Existing Program Synthesizers with SemGuS. The bulk of our discussion in §3 was about achieving domain-agnosticity by building upon the ideas that SyGuS used in achieving solver-agnosticity. However, there also exist synthesis tools that are already domain-agnostic; most notably, solver-aided languages such as Sketch [27], Rosette [28], MiniKanren [8], and Prose [25]. While these tools are not solver-agnostic, they can in principle be used as SemGuS solvers by virtue of their domain-agnosticity.

To use such existing tools as SemGuS solvers, one must develop a compiler of sorts to translate a SemGuS problem (written in the logical format from §3) to the specific front-end language of the tool. This task is not trivial for a number of reasons. First, each of these tools implement restrictions on the types of synthesis problems they accept; these restrictions are what enables their fast algorithms. For example, Rosette, Sketch, and MiniKanren only support finite search spaces (i.e., finite grammars), and this fact is encoded in different ways for different tools (e.g., by imposing bounds on the search depth or by imposing syntactic bounds on the search space). Second, some of these solvers implicitly use limited semantics—e.g., Sketch limits how many times a loop can be executed. Third, some of these solvers require special inputs that are useful to guide the synthesis engine—e.g., Prose requires the user to provide a semantics for each operator in the input language as well as an *inverse semantics* that can be executed *backwards*; the inverse semantics is used to perform efficient enumeration.

Soundly compiling SemGuS problems to these tools requires one to modify the original problems to fit these restrictions. Thankfully, the flexibility of SemGuS comes to our aid! In §4.2, we have described ideas for transforming SemGuS problems using restricted grammars or underapproximating semantics. These transformations are sound for synthesis—i.e., a solution to the transformed synthesis problem, which satisfies the restrictions of a particular external tool, is still a solution to the original problem—and thus can be used to interface with external tools. We are currently working on automating such translations.

The case of Prose is particularly interesting in that it requires inverse semantics, which are not immediately available from a SemGuS problem. However, because SemGuS semantics are expressed logically as CHCs, one can automatically invert these semantics starting from the CHCs—we are currently developing a tool that performs this inversion automatically and uses the inverse semantics to interface with Prose.

Other more specialized solvers, such as those for synthesizing regular expressions [23], could also be interfaced with our framework, with the limitation that they will only be able to handle specific problems. The more general question here is: how can we determine whether a specific SemGuS problem is compatible with a specialized solver? We are working on designing "theories" that describe specific semantics for which specialized solvers exist. For example, if one were to use SemGuS to work with regular expressions, they could import the regular-expression theory, which by design would enable compatibility with certain solvers. Note that this approach is still solver-agnostic because any general SemGuS solver would still be able to use this problem definition.

Lifting Existing Synthesis Algorithms to Work with SemGuS. While interfacing existing synthesizers with SemGuS is one straightforward way of creating SemGuS solvers, we envision that higher efficiency can be achieved by designing solvers that take advantage of the structure of SemGuS problems. Is it possible to lift algorithms (not tools) that have previously been successful with SyGuS or other synthesizers up into SemGuS?

For example, consider the problem of building an efficient enumeration algorithm for SemGuS, an algorithmic technique that is now successfully employed in most SyGuS solvers [21,4,2]. The success of enumeration has been driven by a number of clever ideas for efficiently pruning the search space of relevant programs. An example was mentioned in §4.1, where we discussed the challenges with employing strategies such as behavioral equivalence caching or equality saturation on SemGuS due to the lack of an executable semantics—i.e., in SemGuS, evaluating a term on an input requires a costly call to a CHC solver. We are currently building an enumeration algorithm for SemGuS that addresses this limitation. Our algorithm first synthesizes an executable interpreter from the SemGuS problem semantics, and then uses this executable interpreter to guide the search. To scale, our approach must handle other challenges, which we are also working on—e.g., discovering which operators have a semantics that is associative or commutative can help us avoid enumerating equivalent terms.

While the generality of SemGuS is an obstacle to adapting some well-known algorithms, the same generality also helps SemGuS provide a natural interface to express other algorithms, such as program synthesis using abstraction refinement [29]. The approach taken here is to synthesize programs that work on an abstract domain, and repeatedly refine the abstract domain until a program is found that is correct under the concrete semantics. This approach, in a sense, uses a meta-algorithm that can be expressed naturally in SemGuS, as discussed in

§4.2. We believe that SemGuS will naturally be able to express many such meta-algorithms, and further accelerate the development of new meta-algorithms.

Supporting Richer Specifications. Beyond the basic specification mechanisms, SemGuS already supports syntactic quantitative objectives through weighted grammars (§3.2). To capture the breadth of specifications appearing in modern synthesis applications, the SemGuS framework will have to evolve over time. While we are investigating a number of complex objectives that will require extensions to the framework (e.g., probabilistic specifications), in the following paragraph we describe a specification mechanism the current SemGuS framework can already capture for free: types.

Consider the problem of synthesizing a program that meets a given time complexity (or asymptotic resource usage in general) [11,16]. In existing work, such bounds are specified (and proven correct) using a dependent type system. The solver uses the type system to guide the search, by enumerating only terms that satisfy a certain type. We observe that the SemGuS framework is already able to capture such type-based specifications! In particular, types are a form of *static* semantics that can be associated with terms and, in most cases, typing rules can be encoded as CHCs, similarly to how one encodes semantic rules. For example, the following dependent type rule can be captured using a CHC where each typing judgment $t : type$ is described using a relation $r(t, type)$.

$$\frac{a : \{\texttt{Int} \mid \varphi_a(v)\} \quad b : \{\texttt{Int} \mid \varphi_b(v)\} \quad + : x : \texttt{Int} \to y : \texttt{Int} \to \{\texttt{Int} \mid v = x + y\}}{a + b : \{\texttt{Int} \mid v = x + y \land \varphi_a(x) \land \varphi_b(y)\}} \tag{9}$$

5.2 What can the Synthesis Community Do?

As we mentioned at the beginning of this section:

The vision of programmable program synthesis can only be realized through a community effort.

We discuss problems the community can help with in this concluding section.

A Broader Scope for Synthesis. The scope and potential of synthesis is very broad, in fact even broader than what has been discussed in this paper. An invited paper by Gulwani began [9]

> Program Synthesis is the task of discovering an executable program from user intent expressed in the form of some constraints.

However, we feel that this viewpoint is actually somewhat narrow. We believe that insight on many problems can be obtained via the "lens" of synthesis: for many computing tasks, the goal is to produce some artifact to which some semantics is attached, and the process of producing that artifact can be thought of as a synthesis problem. For instance, in an AI planning problem, the artifact is a plan—i.e., Monica from §1 is a robot, and the sought-for program must navigate her from point A to point B (e.g., minimizing power consumption and time, while avoiding collisions and satisfying other safety guarantees). Closer to home for the CAV community, inside many tools for statically checking assertions in programs (such as SLAM or BLAST), the key component is one that creates an abstracted model of the program that is sufficiently precise to show that an assertion violation is not possible. Among the artifacts that may need to be synthesized are inductive invariants, abstract transformers, function summaries, and interface specifications. Thus, we conclude by offering the following wider definition of synthesis, which connects this broader outlook with the semantics-based perspective that we have presented in this paper:

> Synthesis is the task of discovering a syntactic object—selected from some formalism in which each syntactic object has a rigorously defined semantics—from an "intent" expressed in the form of some kind of constraint.

We believe that the issues discussed in this paper will be increasingly important if synthesis is to be applied successfully to the creation of artifacts that have semantics, but are not programs *per se.*

The generality of our framework can bridge the gap between the many applications of synthesis, and we hope that the community will engage in our work by modeling their synthesis problems in SemGuS, and by adapting their solvers to work with SemGuS. Such contributions will result in new benchmarks and solvers, contributing to the programmability and effectiveness of SemGuS.

Standardization and Competitions. We believe that the idea of a programmable synthesis framework, and SemGuS, the start of such a framework, represents a step forward in program synthesis. Similarly to what happened with SyGuS, SemGuS must be standardized, other researchers should build solvers for it, and these solvers should compete annually in SemGuS competitions.

We hope that this paper will encourage readers to experiment with and advance the ideas presented here, in three ways: First, we hope that the generality of the framework will make it easy for people to use it on various problems, which in turn will make it easy to collect large and diverse sets of benchmarks that will make the design of new solvers focused and effective. Second, we hope that researchers will build new algorithms and techniques that are general and can solve problems built in this framework. Third, we hope to soon create a yearly competition that will foster further interest in building general synthesizers for our framework. More than anything, this paper is a call-to-arms—an invitation to help broaden the scope and abilities of program synthesis, toward an era where Monica uses synthesizers just as much as Python during her daily work.

Acknowledgments. Supported, in part, by a gift from Rajiv and Ritu Batra; by multiple Facebook Research Awards; by a Microsoft Faculty Fellowship; by NSF under grants 1420866, 1763871, and 1750965; by ONR under grants N00014-17-1-2889 and N00014-19-1-2318; and a grant from the Korea Foundation for Advanced Studies. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities.

References

1. R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–8. IEEE, 2013.
2. R. Alur, A. Radhakrishna, and A. Udupa. Scaling enumerative program synthesis via divide and conquer. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 319–336. Springer, 2017.
3. M. Amodio, S. Chaudhuri, and T. W. Reps. Neural attribute machines for program generation. *CoRR*, abs/1705.09231, 2017.
4. S. Barke, H. Peleg, and N. Polikarpova. Just-in-time learning for bottom-up enumerative synthesis. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020.
5. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.
6. L. D'Antoni, R. Samanta, and R. Singh. Qlose: Program repair with quantitative objectives. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 383–401, 2016.
7. *ESolver*. `https://github.com/abhishekudupa/sygus-comp14`.
8. M. Q. Feldman, Y. Wang, W. E. Byrd, F. Guimbretière, and E. Andersen. Towards answering âĂIJam i on the right track?âĂİ automatically using program synthesis. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*, SPLASH-E 2019, page 13âĂŞ24, New York, NY, USA, 2019. Association for Computing Machinery.
9. S. Gulwani. Dimensions in program synthesis. In *PPDP*, 2010.
10. S. Gulwani. Programming by examples: Applications, algorithms, and ambiguity resolution. In *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, pages 9–14, 2016.
11. Q. Hu, J. Cyphert, L. D'Antoni, and T. Reps. Synthesis with asymptotic resource bounds. In *CAV*, 2021.
12. Q. Hu and L. D'Antoni. Syntax-guided synthesis with quantitative syntactic objectives. In *Computer Aided Verification - 30th International Conference, (CAV)*, pages 386–403, 2018.
13. S. Johnson. YACC: Yet another compiler-compiler. Technical Report Comp. Sci. Tech. Rep. 32, Bell Laboratories, 1975.
14. J. Kim, Q. Hu, L. D'Antoni, and T. Reps. Semantics-guided synthesis. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–32, 2021.
15. T. Knoth, W. Di, N. Polikarpova, and J. Hoffmann. Resource-guided program synthesis. In *PLDI*, pages 253–268, 2019.
16. T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann. Resource-guided program synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 253–268, 2019.
17. N. Kobayashi, T. Sekiyama, I. Sato, and H. Unno. Toward neural-network-guided program synthesis and verification. *CoRR*, abs/2103.09414, 2021.

18. A. Komuravelli, A. Gurfinkel, and S. Chaki. Smt-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, 2016.
19. B. Lang. Deterministic techniques for efficient non-deterministic parsers. In *ICALP*, 1974.
20. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.
21. W. Lee. Combining the top-down propagation and bottom-up enumeration for inductive program synthesis. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–28, 2021.
22. A. V. Nori, S. Ozair, S. K. Rajamani, and D. Vijaykeerthy. Efficient synthesis of probabilistic programs. pages 208–217, 2015.
23. R. Pan, Q. Hu, G. Xu, and L. D'Antoni. Automatic repair of regular expressions. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.
24. N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. pages 522–538, 2016.
25. O. Polozov and S. Gulwani. 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, Pittsburgh, PA, USA, October 25-30, 2015*, pages 107–126, 2015.
26. A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. Barrett. *Counterexample-Guided Quantifier Instantiation for Synthesis in SMT*, pages 198–216. Springer International Publishing, Cham, 2015.
27. A. Solar-Lezama. Program sketching. *International Journal on Software Tools for Technology Transfer*, 15(5):475–495, Oct 2013.
28. E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152, 2013.
29. X. Wang, I. Dillig, and R. Singh. Program synthesis using abstraction refinement. *PACMPL*, 2(POPL):63:1–63:30, 2018.