Project Summary

CAREER: Program Synthesis with Quantitative Guarantees
Loris D’Antoni (University of Wisconsin–Madison)

Overview  Software has been changing our lives for many decades, but despite the advances in programming language design, how we write programs has not changed much: programmers keep repeating similar mistakes, writing similar programs, and fixing similar bugs. Program synthesis, the art of automatically generating programs that meet user intents, promises to increase the productivity of programmers and end-users of computing devices by automating tedious, error-prone, and time-consuming 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 size or with good runtime—and to understand when synthesis can result in such good programs.

We envision a world in which synthesis algorithms are more predictable and accurate. To realize this vision, we propose to attack the problem of performing program synthesis in the presence of quantitative objectives while providing quantitative guarantees on the results and on the performance of the synthesis algorithms. Concretely, we will re-design existing specification mechanisms and solving techniques for synthesis problems to accommodate optimization objectives on the synthesized program. To achieve this goal, we plan our investigation along two synergistic research thrusts:

T1 We will investigate the use of formal methods such as weighted tree grammars, weighted logics, and logics with probabilities to specify quantitative objectives on the syntax, semantics, and probabilistic outcomes of the synthesized programs. We will use such formalisms to design new algorithms for synthesizing programs that satisfy quantitative objectives on the syntax and semantics of programs.

T2 We will design new synthesis algorithms that can (i) generate programs that satisfy user-given probabilistic objectives, and (ii) run efficiently when operating on certain probabilistic inputs. We will draw inspiration from the theories of computational learning and random graphs to design synthesis algorithms with provable probabilistic guarantees.

Intellectual Merits  Whereas current synthesis algorithms do not provide explicit support for quantitative requirements over the synthesized programs, the proposed work will introduce a general framework for expressing synthesis problems with quantitative objectives and new algorithms for solving problems expressed in such a framework. The proposed work will lead to fundamental new approaches that use weighted tree grammars to synthesize programs of minimal size; new algorithms based on weighted logics for incrementally solving synthesis problems in the presence of noisy data; new algorithms that leverage techniques from computational learning theory to optimally and efficiently solve synthesis problems with probabilistic objectives; and new analysis tools based on the theory of random graphs for reasoning about the performance of synthesis algorithms that use version space algebras.

Broader Impacts  The proposed work will lead to more predictable, accurate, and robust synthesis algorithms, which will benefit widely used synthesis applications such as personalized education, network synthesis, programming by examples, and automated program repair. All our implementations and datasets will be released open-source. The PI will develop a new course and a new workshop on program synthesis and how to create visual art with it (to be held at UW-Madison). The course/workshop material will be made publicly available. The PI will also organize a series of lectures about the impact of program synthesis on the labor force and will lead several diversity and outreach efforts to undergraduates, women, under-represented minorities, and non-traditional students.

Keywords  Program synthesis; Computational learning theory; Logic; Graph theory; Automata theory.
1 Introduction

Software has been changing our lives for many decades, but how we write programs has not changed much: programmers keep repeating similar mistakes, writing similar programs, and fixing similar bugs. Program synthesis, the art of automatically generating programs that meet user intents, promises to increase the productivity of programmers and end-users of computing devices by automating tedious, error-prone, and time-consuming tasks. In most program synthesis problems, the user describes an intent using high-level specifications such as input-output examples or logical formulae, and the synthesizer magically produces a program that meets the user’s intent. For a long time, the high theoretical complexities and the lack of practical applications prevented program synthesis from being used in practice.

Recently, thanks to the advances in decision procedures and to the advent of the “everyone wants to be a programmer” era, the landscape has started to change and we have seen remarkable results in using synthesis to solve practical programs such as feedback generation in introductory programming assignments [85, 38], automating data wrangling and data extraction tasks [77, 79, 93, 53, 14, 51], generating network configurations that meet user intents [89, 47], optimizing low-level code [88, 83], and more [15, 52]. Despite these practical successes, we still do not have systematic frameworks for synthesizing programs that are “good” according to certain metrics—e.g., produce programs with a reasonable size or with good runtime—and to understand when synthesis can result in such good programs.

Goals of this proposal This proposal tackles this foundational problem and presents a multi-faceted research and education plan with the goals of (i) performing program synthesis in the presence of quantitative objectives to improve the quality of the synthesized programs, (ii) designing program synthesis algorithms that operate over probabilistic spaces and therefore have probabilistic guarantees, and (iii) disseminating the power of program synthesis among students through focused workshops, new coursework, and student involvement. We propose new algorithmic advances that marry concepts from formal methods, computational learning theory, and graph theory to enable new synthesis applications in domains such as automatic feedback generation, program repair, and programming by examples. Figure 1 provides an overview of the proposed research and education activities. As indicated, our formal framework will influence our algorithmic design and our education activities. A primary interface between all activities will be a synthesis toolbox centered on quantitative synthesis tools built by our research group [4, 58, 3, 25].

1.1 Overview of the Research and Education Plan

Before detailing our proposed activities, we summarize our two research thrusts and our education plan.

We envision a world in which synthesis algorithms produce better programs. To realize this vision, we propose to attack the foundational problem of performing program synthesis in the presence of quantitative objectives, a problem that is currently governed by ad hoc approaches.

Example 1.1. Suppose we want to write a program that forwards network packets to two switches $s_1$ and $s_2$, and that satisfies the following requirements: (i) Certain packets must be forwarded to certain switches. (ii) To allow load to be equally distributed, the probability of forwarding traffic to switch $s_1$ should be between 0.45 and 0.55 (we assume a probabilistic distribution over the input packets). (iii) Across all programs that satisfy requirements (i) and (ii), prefer the smaller ones because they occupy less space on a switch. Existing synthesis tools do not support these heterogeneous objectives and our goal is to create synthesis frameworks, algorithms, and tools that can handle all such requirements.

Specifically, we plan our investigation along two synergistic research thrusts:

Research Thrust 1: A Quantitative Framework for Syntax-Guided Synthesis In the first thrust, we shall investigate formal ways to impose quantitative objectives on synthesized programs.

• **Weighted tree grammars for quantitative syntactic specifications** In many applications, it can be useful to add quantitative objectives on the syntax of the synthesized programs—e.g., prefer smaller programs as imposed by requirement (iii) in Example 1.1, or programs with fewer constants, which are more likely to generalize to inputs outside of the specification [84]. For example, in our PLDI’17 paper on program inversion [58], we observed that for the same functional specification one synthesizer would
produce programs with thousands of terms while another one would produce more desirable programs with tens of terms. To express such requirements, we propose to investigate a formal framework for specifying quantitative objectives over the syntax of synthesized programs based on the formalisms of weighted tree grammars [44, 45, 65]. This model can assign weights to programs in the search space and it unifies existing techniques for assigning syntactic costs to programs [18]. Moreover, the properties of weighted tree grammars enable new ways to improve existing synthesis engines—e.g., given a constant \( c > 0 \), one can restrict a grammar to only produce programs of cost smaller than \( c \) and this technique can be directly used to adapt existing synthesizers to incorporate quantitative syntactic objectives.

- **Weighted logics for quantitative semantic specifications** Some applications require quantitative objectives on the behavior of the program. For example, in the context of programming by examples, when no program in the search space is consistent with all the given examples (perhaps due to noise in the data [78]), one might want to synthesize a program that agrees with as many examples as possible. We propose to use weighted logics, such as MaxSAT and MaxSMT [16], as a natural mechanism for describing quantitative properties over the program semantics. In the example, the specification would be expressed as a MaxSAT formula that assigns weight 1 whenever the program is consistent with one of the input-output examples and the synthesis problem is to find a program that maximizes the weight of the formula. Using this formal approach, we can leverage recent advances in divide and conquer for synthesis [11] to design new synthesizers that can efficiently find optimal solutions.

- **Probabilistic logics for probabilistic specifications** Recently, there has been interest in programs that operate over probabilistic input distributions [5, 4, 72]. In this setting, it is useful to synthesize programs that satisfy some probabilistic constraints—e.g., requirement (ii) in Example 1.1—or maximize/minimize some variable expectation—e.g., minimize the probability of failure in a network [89]. We propose to use probabilistic variants of logics as a natural mechanism to specify such problems. While even verification is hard in this setting, we show that there is a new opportunity to create algorithms that provide approximate guarantees using concepts such as Probably Approximately Correct (PAC) learning [60].

**Research Thrust 2: Synthesis with Probabilistic Objectives and Guarantees** The last point of Thrust 1 dwells with probabilistic specifications. Our second thrust is entirely dedicated to solving problems in this realm where the goal is to design new synthesis algorithms that can (i) generate programs that satisfy probabilistic objectives, and (ii) run efficiently when operating on certain probabilistic inputs.

- **Synthesis for probabilistic programs with finitely many outputs** In many applications of probabilistic programming, programs produce a finite number of possible outputs—e.g., what router to forward traffic to next based on some finite packet history—and the probabilistic objectives can require to minimize the probability of a certain bad event—e.g., router congestion. We formally investigate the problem of synthesizing programs in this setting. We draw inspiration from the field of computational learning theory [60] and show how we can reduce the complex problem of synthesizing programs that are correct on potentially infinitely many inputs to the problem of synthesizing programs that are correct on a small finite number of input-output examples. In our preliminary work, we show that under certain assumptions, it is enough to synthesize programs that are consistent with polynomial many examples to get probabilistic guarantees over the whole input distribution the program operates on [4].

- **Synthesis for probabilistic programs with infinitely many outputs** When probabilistic programs require an infinite output space—e.g., a program that decides the size of the TCP protocol congestion window—many of the existing tools from computational learning theory stop working, making the problem
challenging. We therefore propose to design new theoretical tools that are tailored for the types of specification appearing in synthesis with probabilistic guarantees. For example, we propose to leverage the predicates appearing in the probabilistic specification of the synthesis problem to extract finitely many classes of outputs that can be used to reduce the synthesis problem to the one with a finite number of outputs.

- Probabilistic foundations for programming by example Most synthesis algorithms incur in large theoretical complexities due to worst-case behavior, but perform well in practice [25]. We propose to re-analyze existing algorithms for programming by example under the theory of random graphs and show that when the algorithms operate on inputs drawn from certain probability distributions the algorithms will behave well with high probability. We then propose to design new algorithms that are aware of the input distribution and can therefore perform well in practical applications.

Education Plan We will integrate our research and our education plan through the following activities.

- Curriculum development Since his arrival at the UW-Madison’s Computer Sciences department, the PI has been teaching both graduate and undergraduate programming languages courses. We plan to modernize programming paradigms education by focusing on program synthesis; specifically, we will carefully design an undergraduate class that teaches synthesizer development, while retaining foundational traditional concepts. One of our primary goals is to inspire students to transfer synthesis ideas to their future careers in industry. (See Sec. 6 for a detailed rationale.)

- “Visual art with synthesis” workshops for undergraduates To further publicize program synthesis, we will establish annual “Visual art with synthesis” workshops for undergraduate students. The primary goal of the workshops will be to expose students to program synthesis technologies and let them use such technologies in new creative ways. As detailed in Sec. 6, the PI will take concrete action in creating an inclusive environment that is welcoming to women and students from under-represented minorities.

- Awareness, outreach, and undergraduate research Looking forward to the potential of automated program synthesis, the PI is concerned about its use to replace tasks in the labor force—e.g., TAs with automatic graders. To introduce students to this issue, we plan to organize a series of lectures that look at this problem from various angles [71]; we will ensure that the lectures are accessible to everyone and benefit the campus community at large. Since the PI strongly believes that getting hands-on experience in an exciting research environment is crucial to motivate students to pursue graduate studies, we have budgeted funds for undergraduate student support. In fact, the PI has already advised 5 undergraduate students on program synthesis projects during his first 2 years at UW-Madison.

1.2 PI Preparation and Preliminary Work

The proposal outlines an ambitious research and education program and the PI is confident he will complete the proposed tasks during the allotted time frame (see Sec. 7 for a timeline). To better substantiate the above claim, the PI enjoys breadth and depth of expertise in formal methods, program synthesis, and program verification, he has already published preliminary work related to this proposal, and he has a good record of involving undergraduate students in his research.

Expertise The PI’s recent works on program synthesis have appeared at premier verification, programming languages, and software engineering venues—CAV [38, 4], TACAS [43], POPL [23, 89], PLDI [58], ICSE [82], and FSE [25]—showing evidence of the community’s interest in the PI’s line or research. In addition, the PI is an expert in formal methods [6, 27, 7, 26, 34, 29, 36, 30, 50, 32, 33, 31, 39], program verification [74, 9, 67, 28, 5], and applying synthesis to personalized education [8, 35, 37, 56, 90]. His dissertation won the Morris and Dorothy Rubinoff award at the University of Pennsylvania.

Preliminary work The PI and his graduate students have recently published papers on synthesis for automatic program inversion [58] and synthesis for probabilistic programs [4]. The first paper inspired Research Thrust 1 and motivates the need for a framework for formally reasoning about quantities in program synthesis. The second paper inspired Research Thrust 2 and is our first milestone on the problem of synthesizing program with probabilistic objectives. In addition, the PI has performed preliminary work on several of the proposed research tasks, which will enable rapid progress and dissemination of results. Further, as a product of his recent work, the PI’s group has built flexible synthesis tools—e.g., Genic [58],
Genesis [89], and DIGITS [4]—which will be used as a platform for evaluation and experimentation, as well as enablers of teaching and student involvement.

**Student involvement** The PI is committed to enriching undergraduate and graduate student experience through education, research, and outreach. For example, he has recently organized the Programming Languages Mentoring Workshop (PLMW) co-located with the Conference on Principles of Programming Languages POPL ’17 [76]. In the process, he raised more than $60,000 to allow 46 graduate and undergraduate students to attend POPL 2017. Of these students, 13 were women and 5 were members of other under-represent minorities. The PI has also been an advocate of undergraduate student research—serving as a judge for the undergraduate and graduate student research competition at POPL 2016 [87] and working with 5 undergraduate students in the last two years on developing synthesis tools.

### 1.3 Broader Impacts of the Proposed Work

**Impacts on society** Our proposed research will produce reliable synthesis mechanisms that will, among other things, enable computer users to perform error-prone and time-consuming tasks in an automated fashion. Thus, our work can democratize computing and make simple forms of programming more accessible in the era of everyone wants to be a programmer. All our tools will be made open-source.

**Impacts on programmers** Our algorithmic investigations will deepen our understanding of program synthesis and lead to tools that scale to larger domains, therefore increasing programmer productivity and leading to software that is correct by design. As these tools get incorporated in modern software-design environments, their broader impact will continue to manifest. Moreover, as students who are involved in our outreach activities enter the workforce, we expect that they will spread the acquired knowledge to new and exciting domains, thus affecting software development as well as society at large.

**Impacts on education** We lay out a comprehensive education and outreach plan in Sec. 6. In our prior work, we used program synthesis to build education tools that have helped thousands of students and many teachers around the world, particularly in developing countries with limited access to higher education [35, 37]. We also intend to incorporate our technologies in tools for education. At the teaching level, we will amplify our impact on education through modernized curriculum, inclusive synthesis-themed workshops, and undergraduate involvement in research. Finally, we will educate students about the potential of synthesis to replace jobs and the corresponding countermeasures through a series of lectures.

### 2 Background on Program Synthesis with Quantitative Objectives

In this section, we present an overview of synthesis techniques that handle quantitative objectives.

**Program synthesis** The goal of program synthesis is to find a program in some search space that meets a given specification, which can be given as a set of examples, a logical formula, or in other ways. Recently, many problems in this paradigm have been unified into a framework called syntax-guided synthesis (SyGuS). A SyGuS problem is specified by a context-free grammar, which describes the search space of the synthesizer, and a logical formula, which describes the specification. Many synthesizers now support SyGuS inputs and can be used for any problem that can be encoded in SyGuS, making this framework of great practical value. Existing synthesizers can be grouped in three categories: (i) enumerative synthesizers, which search for a program using smart brute-force algorithms [11]; (ii) symbolic synthesizers, which solve the search problem using constraint solving [86]; (iii) stochastic synthesizers, which explore the search space using stochastic methods such as Montecarlo Markov Chain search [83].

**Adding quantitative objectives** The Boolean specification mechanism provided by SyGuS is powerful and can capture the functional requirements of many synthesis problems. However, problems that also have non-functional quantitative requirements are currently behind the ability of SyGuS—e.g., one cannot ask a SyGuS solver to return the smallest program that satisfies a specification. Several synthesis approaches have attempted to include some form of quantitative specification to express this kind of preferences.

One domain where quantitative specifications arise is automatic grading of programming assignments, where the problem is to synthesize a modification to a program written by a student to make it pass a set of test cases given by an instructor [85, 38]. Here, the quantitative objective is to find the program that is syntactically closest to the one written by the student. Existing techniques solve this problem by simply augment the synthesizer with some counter that keeps track of the difference between the synthesized
program and the student’s one. The synthesizer then uses a linear search to find a program of smallest cost [85]. While the problem of repairing a program can be encoded in SyGuS, it is currently not possible to specify the requirement of producing the smallest repair. A more formal attempt at incorporating syntactic costs in program synthesis has been proposed by Bornholt et al. [18]. In this approach, the SyGuS search space is structured into an ordered set of individual search spaces of increasing size, which allows modelling syntactic costs over, for example, the domain of natural numbers.

In general, existing approaches are domain specific, somewhat cumbersome, and cannot handle quantities pertaining to semantic properties of the synthesized program. In particular, current mechanisms cannot synthesize programs that agree with as many examples as possible or programs that satisfy probabilistic objectives. The goal of this proposal is to design a unified framework for specifying synthesis problems with quantitative objectives and to provide new techniques that leverage such a framework to efficiently solve new synthesis problems. To achieve our goals, in Research Thrust 1 (Sec. 3), we propose a SyGuS framework with new formal specification mechanisms that can express complex quantitative objectives and enable new synthesis algorithms. In Research Thrust 2 (Sec. 4), we propose to address the challenge of synthesizing programs in the presence of probabilistic objectives.

3 Thrust 1: A Quantitative Framework for Syntax-Guided Synthesis

In this thrust, we propose to address three core formal problems with the goals of defining a unifying framework for describing synthesis problems with quantitative objectives and designing new algorithms for solving synthesis problems expressed in this framework.

Background Existing program synthesis tools can efficiently synthesize programs that adhere to a given functional specification—e.g., the program performs correctly on a given set of input/output examples. When multiple programs adhere to the specification these tools do not provide guarantees about which of these programs is returned by the synthesizer. Solvers based on program enumeration typically output the syntactically smallest program [2], symbolic solvers output whatever program is produced during the constraint solving [86], and probabilistic synthesizers are unpredictable [83]. Certain solvers employ complex hard-coded ranking functions that guide the search towards a “preferable” program [77], but these functions are hard to write and are decoupled from the functional specification. In summary, existing specification mechanisms only deal with functional requirements and do not provide explicit ways to rank programs to express which one should be preferred if multiple solutions are possible. The lack of a formal treatment of quantitative requirements stands in the way of designing synthesizers that are aware of quantitative objectives and can use such objectives to perform smarter forms of synthesis. In this thrust, we attack this problem and introduce a unifying framework for synthesis problems with quantitative objectives together with new synthesis algorithms that take advantage of our framework.

The ideas presented in this thrust build on the SyGuS framework. A SyGuS 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 users. The first constraint describes a functional semantic property that f should satisfy and is given as a predicate $\psi(f) \equiv \forall x.\phi(f, x)$. The second constraint limits the search space of f and is given as a set S of expressions specified by a context-free grammar 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.

Example 3.1. The following SyGuS problem asks to synthesize a function that computes the max of two numbers. The syntactic constraint is given by the following context-free grammar.

```
Start ::= Start + Start | if(BExpr) then Start else Start | x | y | 0 | 1
BExpr ::= BExpr > BExpr | ~BExpr | BExpr ∧ BExpr
```

The semantic constraint is given by the following formula.

$\psi(f) \equiv \forall x, y. f(x, y) \geq x \land f(x, y) \geq y \land (f(x, y) = x \lor f(x, y) = y)$

Two possible solutions are the following semantically equivalent, but syntactically different programs.

- $\max (x, y) = \text{if}(x > y) \text{then } x \text{ else } y$
- $\max'(x, y) = \text{if}(y > x) \text{then } y \text{ else } x$

3.1 Weighted Tree Grammars for Quantitative Syntactic Specifications

The most common quantitative requirements are those that somehow restrict the size of the synthesized program. For example, in synthesis from input/output examples, it is desirable to produce smaller programs that are more likely to generalize to examples outside of the specification [54] and, in automated
feedback generation for programming assignments, the “repaired” program should be syntactically similar to the incorrect one the student wrote so that the student can understand what mistake was made [38]. SyGuS does not provide ways to specify requirements on the size of the program and existing synthesizers cannot directly support these kinds of objectives. Therefore we propose to tackle the following question.

**Research question 1** Can we design a SyGuS framework that supports quantitative specifications on the syntax of the synthesized programs?

This question poses a number of fundamental technical challenges: (i) What is a good formalism for describing quantitative syntactic requirements? (ii) How can we solve the synthesis problem in the presence of such requirements? (iii) Can we reuse existing SyGuS solvers to solve such problems?

**Action plan** Since SyGuS defines the synthesis search space using context-free grammars, we propose to extend this formalism with weights to assign costs to programs in the grammar. Before starting, we decide to focus our attention on a restricted class of context-free grammars called regular-tree grammars because, to our knowledge, all the benchmarks appearing in the SyGuS competition [10] and in practical applications of SyGuS operate over regular tree grammars. Moreover, it was recently shown that many synthesis problems that are undecidable for SyGuS become decidable when using regular tree grammars [20]. A regular-tree grammar is a context-free grammar in which each terminal symbol is a term with an associated arity—e.g., $x + y$ is a term $+$($x, y$) where the $+$ symbol has arity 2—i.e., it has two children.

To add quantities to the picture, we adopt weighted tree grammars [45], a well-studied formalism with many desirable properties. A weighted tree grammar (WTG) is a tree grammar in which each production is assigned a weight from a semiring $(S, \oplus, \otimes, 0, 1)$. Common examples are the tropical semiring $(\mathbb{N}, \min, +, \infty, 0)$ and the probabilistic semiring $([0,1], +, *, 0, 1)$. The cost of a derivation is the $\otimes$-product of all the productions involved in the derivation and the cost of a term is the $\oplus$-sum of the costs of all its possible derivations. Given a weighted tree grammar $G$ and a term $f$, we use $\text{cost}(G, f)$ to denote the cost of $f$ in $G$. A weighted tree grammar can, for example, compute the height of a term or the number of occurrences of some element in a term—e.g., the number of constants or the number of if statements. A SyGuS problem with *quantitative syntactic objectives* is given by a functional constraint $\psi(f)$, a weighted tree grammar $G$, and a constraint on the weight of the program—e.g., $\min(\text{cost}(G, f))$ or $\text{cost}(G, f) < 10$.

**Example 3.2.** Consider again the grammar in Example 3.1. The following weighted grammar over the $(\mathbb{N}, \min, +, \infty, 0)$ semiring assigns to each function a cost corresponding to the number of if statements in the function. We write $RHS/k$ to assign cost $k$ to a production $RHS$ and omit the cost for productions with cost 0.

\[
\begin{align*}
\text{Start}::= \text{Start + Start} \\
| \text{if(BExpr) then Start else Start/1} \\
| x | y | 0 | 1 \\
\text{BExpr} ::= \text{BExpr > BExpr} \\
| \text{~BExpr} \\
| \text{BExpr \& BExpr}
\end{align*}
\]

The functions $\text{max}$ and $\text{max'}$ from Example 3.1 have costs 1 and 2, respectively, and $\text{max}$ is a function that minimizes the syntactic cost $\text{cost}(G, f)$. In this case, there doesn’t exist a function of cost 0.

Aside from being a natural formalism, WTGs enjoy properties that make them a good choice for our model. First, WTGs are equi-expressive to weighted tree automata and have logic characterizations [24, 44, 45]. Second, WTGs enjoy many closure and decidability properties—e.g., given two WTGs $G_1$ and $G_2$, we can compute the grammars $G_1 \otimes G_2$ and $G_1 \oplus G_2$ such that, for every $f$, $\text{cost}(G_1 \oplus G_2, f) = \text{cost}(G_1, f) \oplus \text{cost}(G_2, f)$ and $\text{cost}(G_1 \otimes G_2, f) = \text{cost}(G_1, f) \otimes \text{cost}(G_2, f)$. WTGs are also closed under tree substitution and Kleene star. These properties open many research opportunities, some of which we discuss here.

**Search using closure properties** Consider a WTG $G$ that operates over the $(\mathbb{N}, \min, +, \infty, 0)$ semiring.

Given a constant $k$, one can effectively construct a regular-tree grammar $G^{(k)}$ that only produces terms in $G$ with weight smaller or equal than $k$ [45]. Using this property, we can use existing SyGuS solvers to find the program of smallest cost. First, we can use a solver to generate any solution to the SyGuS instance and ignore the weight. Second, we can restrict the grammar to only produce programs with smaller cost than the synthesized one. We can then repeat until the smallest program is found.

**Combining objectives** Using properties of semirings, we can combine multiple WTGs into a single grammar over a product semiring and generate more complex objectives—e.g., minimize one quantity and then the other, or find a Pareto optimal program. This property can be used to define quantitative metrics in a modular fashion and perhaps solve the synthesis problems modularly.
**Minimization** WTGs and weighted tree automata can be minimized [64]. This property can be used to simplify the grammars and speed up symbolic solvers, which encode the structure of the grammar as a constraint. Notice that this is not the case for context-free grammars, the current model of SyGuS.

**Search as learning** WTGs over the probabilistic semiring, also called probabilistic tree grammars (PTGs), assign probabilities to programs in the grammar. PTGs can be learned from data [19, 66] and we can use this property to design probabilistic synthesizers for domains for which we have many examples of good and bad programs (according to some definition). In particular, we can use the examples to generate a PTG and use it as the search space to our SyGuS program. We can then, for example, design an enumeration based solver that instead of searching for programs in a breadth-first manner, explores the search space preferring programs with higher probabilities first. Using the closure under $\oplus$, $\otimes$, and intersection we can also define variants of the search problem in which we restrict the grammar to certain templates where the solution is likely to reside, a common technique in program synthesis [86]. This foundational framework is general, enjoys desirable properties, and it opens opportunities for developing more formal and well-reasoned solvers. Our formulation also opens questions about the decidability and complexity of SyGuS with quantitative objectives. Moreover, the formalism could be extended to more complex models such as attributed tree grammars [13] and capture more complex objectives.

**Preliminary work** The PI and his student, Qinheping Hu, have compiled more than 200 benchmarks that can be encoded in this framework from their PLDI’17 paper [58], that same paper that provided the inspiration for the proposed framework. In the paper, SyGuS was used to invert complex bit-vector functions and we observed that some solvers were producing very large outputs—e.g., CVC4 [81] often returned functions with thousands of terms when a program with 7 terms existed. Qinheping has a designed a format for enriching SyGuS with weights and has encoded the problems from our paper in this format, giving us an initial suite of benchmarks of different natures: in some we minimize the size of the functions and in others we minimize the number of constant terms. In our preliminary results, we manually restricted the SyGuS grammars to produce terms of size at most $k$ and we showed that some of the existing SyGuS solvers [2] can find optimal solutions in the same amount of time taken without quantitative objectives. Finally, the PI and his student curate two automata libraries [3, 1] and already have many of the starting components needed to implement tree grammar operations. We also plan to build on Tiburon [66], a tool that supports training algorithms for probabilistic WTG.

**Related work** FlashMeta [51, 77] uses ranking functions to prefer small programs. The functions are hard-coded in the synthesizer and specialized for certain problems. Synapse [18] supports syntactic cost functions that are defined separately from the SyGuS grammar using a decidable theory. Synthesis is done using an iterative search where sketches of increasing sizes are given to the synthesizer. The examples provided in [18] can be naturally encoded in the framework we presented using appropriate choices of semirings. Similarly, the synthesis technique discussed in [18] can be naturally adapted to our framework and falls in the special case in which the semiring is over a positive countable domain. In summary, our framework works in richer domains—e.g., probabilities—and it couples the syntactic cost directly with the search space, therefore enabling properties like minimization and closure under semiring operations.

### 3.2 Weighted Logics for Quantitative Semantic Specifications

In some synthesis domains, it can be useful to impose quantitative objectives over semantic properties of the program. For example, when synthesizing spreadsheet transformations [42], the data available to the synthesizer may be noisy and some of the examples might contain typos or have an incorrect format. In this case, there might not be a program in the search space that is consistent with all the examples and one might want to instead find a program that is consistent with most of the them, a very common trade-off in machine learning. Similar issues arise in quantitative program repair [38]. Existing specification frameworks do not provide ways to specify such a requirement and existing synthesizers do not provide good support for these kinds of objectives. Therefore we propose to tackle the following question.

**Research question 2** Can we design a SyGuS framework that supports quantitative specifications over semantic properties of the synthesized programs?

This question leads to several challenges, which involve picking a formalism that is adequate to specify commonly occurring objectives and that enables new efficient forms of synthesis.
**Action plan** Since SyGuS specifies semantic requirements using logic, we propose to enrich this formalism and assign costs to “subformulas” in the specification using logics with quantitative objectives [46]. In the following, we only focus on the variants of such logics that are pertinent to our proposal.

One of the simplest quantitative extension of first order logic is the MaxSMT problem, which is defined as follows: given a set of first order formulas \( \varphi_0, \ldots, \varphi_n \) over some theory and associated weights \( w_0, \ldots, w_n \), find a subset \( M \subseteq \{ 1, \ldots, n \} \) such that (i) \( \varphi_0 \land \bigwedge_{i \in M} \varphi_i \) is satisfiable and (ii) the award \( \sum_{i \in M} w_i \) is maximized (or minimized). The constraints \( \varphi_1, \ldots, \varphi_n \) are soft constraints while \( \varphi_0 \) is a hard constraint that must be satisfied by the solution. This quantitative extension preserves the decidability of the underlying logic and it has many applications in programming languages [16]. When the functional requirement is given as a MaxSMT problem, the goal is to find a program in the SyGuS search space that, when replaced in the formula, maximizes the sum of the weights. More powerful logics we plan to investigate allow linear objectives over numbers appearing in the formula while retaining decidability [63, 16].

**Example 3.3.** Consider the grammar from Example 3.1. If we assume programs can use at most one if statement—we can set this constraint using WTGs—no program can realize the following specification.

\[ \psi(f) \overset{\text{def}}{=} f(2, 3) = 3 \land f(4, 1) = 4 \land f(6, 3) = 6 \land f(5, 5) = 6 \]

If we modify the specification to a weighted one in which each conjunct has weight 1, the program max is a solution that assigns weight 3 to the formula—i.e., the maximum possible weight.

Modeling semantic objectives using weighted logics opens a variety of exciting research opportunities.

**Parallel incremental solvers for quantitative objectives** The solutions to a max MaxSMT specification are monotonic—i.e., a solution that satisfies a set of formulas \( C \) also satisfies all the subsets of \( C \).

Using this property, we can re-design divide-and-conquer synthesis techniques [11], which search for programs that satisfy subformulas in a functional specification, to incrementally (and in parallel) search for programs that satisfy subsets of clauses with increasingly large weights until an optima is found.

**Combining objectives** Problems might combine syntactic and semantic costs—e.g., synthesize the smallest program that satisfies as many examples as possible. We plan to investigate variants of quantitative logics that could help unify these different types of costs. A particular appealing logic for such a task is the Semiring-Induced Propositional Logic [61], which assigns weights from a semiring to assignments of a formula, and could be used to unify the weights assigned by the formula and by WTG.

**Preliminary work** We have identified several problems that can be modeled in this framework and have compiled an initial set of benchmarks from synthesis from noisy data [78] and our previous work on personalized education [38] where the goal is to find programs that pass a set of test cases and have similar semantic behavior to another program. We are also collecting problems from quantitative network repair [89], another domain where the PI is active. We will use these benchmarks to drive our research.

**Related work** Classic synthesis algorithms have been adapted to work over noisy data [78, 42]. These approaches are tailored to this domain and do not handle general quantitative semantic objectives.

### 3.3 Probabilistic Logics for Probabilistic Specifications

In certain applications—e.g., automated decision making [72, 4]—programs operate over inputs drawn from probability distributions and the synthesis specification can impose constraints on the expected outcomes of the program. Almost no tool support exists for specifying and solving synthesis problems with probabilistic objectives. Therefore, we propose to tackle the following question.

**Research question 3** Can we design a SyGuS framework that supports quantitative specifications on probabilistic outcomes of the synthesized programs?

**Action plan** We discussed in Sec. 3.1 how WTAs can associate probabilities to terms in the SyGuS grammar. In this section, we propose to also allow the functional specification to use probabilistic constructs and have the following form.

\[ \psi(f) \overset{\text{def}}{=} \Phi_D(x). \phi(f, x) \]

Here, \( \Phi_D(x) \) (notation from [75]) denotes that inputs are randomly drawn from a distribution \( D \) and \( \phi \) can contain expressions of the form \( P(p) \), which denote the probability of an event \( p \).
Example 3.4. Consider the grammar in Example 3.1 and assume again that we can use at most one if stringers or network switches deciding which switch to forward traffic to. In the following, we write

\[ P \]

We start by considering programs that output finitely many possible values—e.g., machine learned classifiers. For example, programs that compute the max or the min of a Boolean value as this assumption can be easily extended to finitely many values.

### 4 Thrust 2: Synthesis with Probabilistic Objectives and Guarantees

To follow the last topic of Thrust 1, in this thrust, we investigate the problem of designing algorithms for synthesizing programs that operate over probabilistic input distributions.

#### 4.1 Synthesis for Probabilistic Programs with Finitely Many Outputs

We start by considering programs that output finitely many possible values—e.g., machine learned classifiers or network switches deciding which switch to forward traffic to. In the following, we write \( P(x) \) to denote the output of \( P \) on input \( x \) and assume that the program inputs are distributed according to some probability distribution \( D \) (recall Sec. 3.3). Without loss of generality, we assume the program returns a Boolean value as this assumption can be easily extended to finitely many values.

**Research question 4** Can we solve the SyGuS problem for functions with finitely many outputs in the presence of probabilistic objectives?

**Preliminary work** In our CAV’17 paper [4], we presented a new algorithm called distribution-guided inductive synthesis (\textsc{digits}), for repairing probabilistic programs, a problem that falls in the framework presented in Section 3.3. The overall flow of \textsc{digits} is illustrated in Figure 2. Suppose we have a program \( P \) such that \( I_{D}(x), \phi(P, x) \) does not hold. The goal of \textsc{digits} is to construct a new program \( P' \) that satisfies \( I_{D}(x), \phi(P', x) \) and is semantically close to \( P \)—i.e., \( P_{D}(P \neq P') \equiv P(D(x) \neq P'(x) | x \sim D) \) is small. \textsc{digits} learns the correct repair from a finite set of samples by tightly integrating the following three phases:

1. **Sampling** \textsc{digits} begins by sampling a finite set \( I \) of program inputs from \( D \)—we call \( I \) the set of samples.
   - The set \( I \) is used to sidestep having to deal with arbitrary distributions directly in the synthesis process.
2. **Synthesis** The second step is a synthesis phase, where \textsc{digits} searches for a set of candidate programs \( \{P'_1, \ldots, P'_l\} \) in the search space \( S \) where each \( P'_i \) classifies the set of samples \( I \) differently.
3. **Quantitative verification** Every generated candidate program \( P' \) is checked for correctness using an automated probabilistic inference technique and \textsc{digits} outputs the repair semantically closest to \( P \).
Example 4.1. Recall Example 1.1 and the problem of synthesizing a network switch that divides traffic between two next-hop switches. Packets arrive following a distribution $D$ and the switch has to decide whether to forward each packet to switch $s_1$ or switch $s_2$. Assume we have a forwarding function $P$ that violates the following property, which states that traffic is split almost equally among $s_1$ and $s_2$.
\[ J_{D,x}: P(f(x) = s_1) > 0.45 \land P(f(x) = s_2) > 0.45 \]
We want a program $P'$ that satisfies the property and, since reconfiguring switches incurs a cost, we require $P_D(P' \neq P)$ to be as small as possible. digits solves this problem by sampling inputs from the packet distribution $D$ and synthesizing a program for each possible way to output $s_1$ and $s_2$ on the inputs.

To implement digits, one needs to provide two components: a) the procedure synth that produces programs consistent with labeled examples and b) a (sound) probabilistic inference algorithm to check whether the synthesized program satisfies the target property and to compute $P_D(P \neq P')$. We assume such components are given as other tools for such problems are available [5, 49]. Remarkably, nothing prevents the procedure synth from using the techniques for syntactic and semantic quantitative synthesis presented in Sec. 3, tying together all the quantitative objectives supported by our framework.

The digits algorithm is simple, but enjoys interesting convergence properties, which we now describe in detail. First, we need to ensure that there are enough programs close to the optimal solution $P^*$ that satisfy $\psi(f)$; for a program $\hat{P}$ and $\alpha > 0$, we define the set $B_{\alpha}(\hat{P}) = \{P' \in R \mid P_D(\hat{P} \neq P') \leq \alpha\}$ of programs close to $\hat{P}$. We say that the pair $(\hat{P}, \psi)$ is $\alpha$-robust iff $\forall P' \in B_{\alpha}(\hat{P}), \psi(P')$ holds. Second, we recall the concept of Vapnik–Chervonenkis (VC) dimension from computational learning theory [60, 17], which we use to capture the expressiveness of the search space $S$. We say that the search space $S$ has VC dimension $k$ if $k + 1$ is the smallest number for which, for any set of samples $I$ of size $k + 1$, there exists two sets $I^+$ and $I^-$ that partition $I$ for which no program $P' \in S$ is consistent with $I^+$ and $I^-$. 

**Theorem 4.1** (Convergence of digits [4]). Assume $(P^*, \psi)$ is $\alpha$-robust for some $\alpha > 0$ and is $k$ the VC dimension of the search space $S$. For all bounds $0 \leq \epsilon \leq \alpha$ and $\delta > 0$, function synth, and $n \geq \text{poly}(\epsilon, \delta, k)$, with probability $\geq 1 - \delta$ digits enumerates a program $P'$ with $P_D(P^* \neq P') \leq \epsilon$ and $J_{D,x}(\phi(P', x))$.

In words, digits converges to the optimal solution using polynomially many samples and, for search spaces for which we manually proved upper bounds on the VC dimension—e.g., bounded loop-free programs—digits could compute good repairs for programs with <50 lines of code in <10 minutes.

**Action plan** Our preliminary work opens new doors to exciting problems.

**From repair to synthesis** Our convergence result relies on the structure of the repair problem—i.e., digits converges depending on whether the property $\psi$ is robust with respect to the optimal repair $P^*$. Our first goal is to adapt digits to solve synthesis problems and to identify a different condition for convergence. We conjecture that digits converges whenever the set of programs in $S$ that satisfy $\psi$ has non-zero measure, meaning that the probability of finding some program satisfying $\psi$ is greater than 0. We believe this condition to be sufficient and necessary. We then plan to extend the result to synthesizing programs that maximize or minimize certain probability expectations—e.g., minimize the probability of network failure. A unique challenge is that, unlike for functional synthesis, a probabilistic specification typically does not specify what the output of the program should be on a certain input.

**Scaling digits** digits currently scales to small numbers of samples because it tries all possible output combinations for the sampled inputs. We propose to modify digits to only output combinations that are “close” to those needed to prove the probabilistic property. For example, if the goal is to synthesize a program $P$ such that $P(P(x) = \text{true}) \geq 0.8$, it will be unlikely that, given $n$ samples from $D$, a program that returns false on all such samples (or even on 50% of them) produces a useful solution. We plan to formally derive provable $\epsilon/\delta$-guarantees for this kind of sampling strategies.

**Measure VC dimension** Our convergence bounds require knowing the VC-dimension of the search space and computing this quantity is a $\Sigma_2^p$-hard problem [70]. Although exact solutions are hard to compute, we plan to investigate other quantities, such as the Rademacher complexity [69], which can be efficiently estimated and could lead to similar testable convergence results.

**Related work** In computational learning theory [60], the goal is to provide provable guarantees for machine learning algorithms. In classic machine learning there are no functional requirements imposed on...
the learned program and the search space is typically well-behaved, unlike what happens in synthesis. Other techniques for synthesizing programs with probabilistic objectives have no provable guarantees [72]. The marriage between synthesis and computational learning theory we propose, can explain why synthesis works in this domain and opens new algorithmic opportunities.

4.2 Synthesis for Probabilistic Programs with Infinitely Many Outputs

We next consider programs that output infinitely many values—e.g., a controller that sets the temperature of an air conditioner or a program that decides the size of the TCP protocol congestion window.

Research question 5 Can we solve the SyGuS problem for functions with infinitely many outputs in the presence of probabilistic objectives?

Action plan The techniques we presented in Section 4.1 require the output of the program to be finite-valued so that one can try all the possible different output combinations when performing synthesis. We propose new fundamental ideas for using such techniques even for synthesizing programs with infinitely many possible output values.

Output refinement As we explained earlier, a probabilistic specification might not tell us what the output of the program should be on a certain input. While in the finite output case one can simply try all possible outputs, this is not possible in the infinite setting. We propose to use a notion of output refinement in which we reduce the problem of synthesizing a program that can output infinitely many values to the problem of synthesizing one that outputs finitely many. Concretely, we propose to proceed in two phases. First, we define a partitioning of the output space into finitely many classes—e.g., positive and negative numbers. Second, we synthesize a program that symbolically outputs in those classes. If the process fails, we further refine the classes into smaller ones and repeat. This approach requires to carefully choose the classes based on the property we want our synthesized program to satisfy. For example, if we are trying to satisfy $P(P(x) \geq 2) \geq 0.5$, it will be natural to, at first, partition the program outputs into those greater or equal than 2 and those that are not.

Convergence While the partitioning technique is sound, it is unclear whether and when it will terminate or converge to optimal solutions. To provide convergence guarantees, we plan to study new notions of VC dimension and Rademacher complexity that are tailored to our symbolic approach. In particular, our output refinement technique induces finite sets of output values and we want to understand how to define and compute symbolic notions of the VC dimension that operate over symbolic outputs.

Preliminary work Our investigation will build on our open source implementation of DIGITS [4].

Related work We are not aware of a similar problem in classic computational learning theory. The closest is perhaps the work on neural networks with continuous output values [12]. However, the challenges we face are unique to having a combination of probabilistic and functional specifications.

4.3 Probabilistic Foundations for Programming by Examples

The next problem is on a somewhat different space than the two we just described and is a foundational approach to trying to understand certain classes of synthesis problems. In particular, we are interested in explaining why certain synthesis algorithms based on the concept of version space algebras (VSA) [62] work in practice despite their high theoretical complexities.

Background VSA synthesis techniques are used to succinctly represent the set of all expressions consistent with a given set of examples [55]. For certain search spaces (usually called domain specific languages), one can efficiently compute the VSA representation at least in practice [55, 82, 25]. However, it is often the case that adding more features to the language makes VSA impractical for larger numbers of examples.

Consider a language for computing string transformations that only contains the operator $\text{substring}(l, r)$ where $l$ and $r$ are integers—e.g., $\text{substring}(3, 5)$ extracts the substring of the input between the third and fifth character, while $\text{substring}(1, -4)$ extracts the substring between the first and fourth to last character. Given a set of examples one can easily find all the substring expressions consistent with the examples; there are in fact at most four. The problem becomes more complicated if we add an if-then-else operator to the language that allows to check whether a string starts with a certain substring. Consider the input/output examples $abc \rightarrow bc$, $abce \rightarrow bce$, and $ccc \rightarrow ccc$. In this case, a single substring expression cannot describe all
Research question 6  For what input distributions do VSA algorithms perform well?

Action plan  Given a set of examples \( I \) we propose to build an undirected graph \( G = (V, E) \) with nodes \( V = I \) and edges between any two examples \( i \) and \( i' \) for which there exists a program in \( L \) (the language without if-then-else) consistent with both \( i \) and \( i' \). We can then characterize the number of subsets of \( I \) considered by the VSA algorithm using this graph. For any edge \( (i, i') \) the VSA clearly generates a non-empty set of programs consistent with the set \( \{i, i'\} \). Similarly, for any two nodes \( i \) and \( i' \) that are not connected by an edge, there won’t be any set of examples \( I' \) containing both \( i \) and \( i' \) in the VSA. This means that the largest set of examples in the VSA is smaller than the largest clique in the graph \( G \). Therefore, if cliques are not “too large”, the VSA algorithm will run efficiently in practice.

Probabilistic graph models  We propose to use probabilistic graph models—e.g., random graphs—to explain why VSA algorithms perform well in practice [40]. In particular, if the input distribution of a problem can be modeled using a certain type of probabilistic graph model we can analyze the expected size of cliques, their distributions, and use them to better understand why on certain inputs the algorithms perform well. We will show some positive results in our preliminary work.

Distribution-aware VSA  If the input distribution is known a priori, we can use our probabilistic models to design new synthesis algorithms that are aware of such a distribution. For example, if the input is distributed according to a power law graph [40], most of the nodes will be part of a potentially very large clique, causing existing algorithms to run poorly. However, an algorithm that is aware of the input distribution will detect which cliques are the most represented ahead of time and it will avoid considering all its subsets. Such an algorithm will build a correct VSA with high probability.

Testing input distribution type  To designing algorithms that behave well for certain input distributions, we need to devise mechanisms to detect what distribution the input examples resemble most. We will adopt and extend existing graph detection theory techniques [68] to perform this task.

Ours is a new fundamental approach to a challenging problem and no similar algorithmic analyses have been developed to explain why synthesis algorithms perform well in practice.

Preliminary work  We collected around 500 examples from the web implementation of our tool NoFAQ [25], which learns how to fix command line errors by allowing people to provide examples of fixes. NoFAQ uses VSA to synthesise programs in a language that follows the grammar we presented earlier and, based on our preliminary investigations, the collected data behaves similarly to a random graph where each edge appears with probability 0.05; in a random graph nodes are given and each edge is added at random with a certain probability \( p \). In a random graph with \( n \) nodes and edges appearing with probability smaller than 0.5, the largest clique has size \( O(\log n) \) with high probability. Therefore, even if we consider all subsets of each clique, we will consider at most \( O(n) \) subsets of examples. Therefore, VSA synthesis scales linearly in the number of examples, a trend we also observed in our implementation.

Related work  Our problem is related to rule learning, where the goal is to learn a set of logical rules—e.g., horn clauses—that describe a certain relation [48]. Our setup differs as we consider complex synthesis languages and probabilistic input distributions, which have not been considered in rule learning.
5 Development, Benchmark Collection, and Evaluation

Tool development We will implement all of our contributions on top of our existing synthesis tools. First we will develop our quantitative SyGuS framework as an open source library with APIs and parsers that can be accessed by other tools. We will build on top of our automata libraries [3, 1] to design the algorithms for SyGuS with syntactic costs (§ 3.1). For the semantic costs (§ 3.2), we will implement on top of our synthesizer for syntactic costs and we will leverage existing SMT solvers [41]. We will implement the probabilistic version of SyGuS (§ 3.3) as an extension of our framework and integrate it with our implementation of digits. The final result will be a comprehensive and extensible tool for syntax guided synthesis with quantitative objectives.

Benchmark collection and evaluation As part of our research on synthesis, we have collected a comprehensive set of benchmarks for various quantitative synthesis problems. We have benchmarks that require syntactic constraints on functions used to invert string encoders [58], benchmarks with syntactic and semantic constraints in the domain of automatic grading for introductory programming [38, 82], simple imperative programs with probabilistic objectives [4], and examples used in VSA-style learning [25, 82]. We also collected benchmarks from the related work we presented in the proposal [18, 78] and we plan to collect benchmarks from the SyGuS competitions and modify them to add quantitative specifications [10].

For evaluation, we will be primarily concerned with assessing the effectiveness of the individual algorithmic contributions using our benchmark collection. Our key metrics of interest include: (i) synthesis performance and overhead of quantitative objectives; (ii) expressiveness and succinctness of the proposed specification framework; (iii) scalability w.r.t. search space size, target program size, and specification size.

For synthesis problems with syntactic and semantic costs, we will compare our new approaches against the Synapse system [18] and tools for synthesis from noisy data [78]. Our first objective is to replicate the running times of these tools while supporting a rich specification framework like ours. Next, we will aim at using the properties of our formalisms to further improve these results. As no tool support exists for synthesis with probabilistic objectives, we will use the current status of our preliminary work—i.e., the tool digits—as a baseline for our experiments. On top of our evaluation, we will use the proposed algorithms to improve QLoSe [38] and Refazer [82], two of our tools for personalized education that can benefit from quantitative objectives, and that we will use to assess the effectiveness of our algorithms in practice. Our evaluation will proceed simultaneously with the research activities outlined in Research Thrusts 1 and 2 and both the graduate and undergraduate students we budgeted support for will collaborate in designing a comprehensive evaluation plan.

6 Education Plan

6.1 Curriculum Development

Rationale Since beginning his academic career at UW-Madison in 2015, the PI has taught the undergraduate compilers course (CS536) and the graduate course on program verification and synthesis (CS703). Since program synthesis is making its way into industry [51, 52], we would like to also introduce undergraduate students to this emerging topic during their senior year. Learning these advanced concepts will make senior students better prepared for graduate school and the complex programming projects we will add to the class will also qualify them to work in industry. Since the undergraduate compilers course is not a good platform for teaching these topics, the PI has already received approval to reintroduce another undergraduate course on programming paradigms (CS538). Traditionally, CS538 introduces students to functional and logic programming and we propose to redesign the course by rephrasing several of the taught topics as synthesis questions. For example, we can have the students build a small synthesizer using logic programming. The idea is that students taking the class will learn the fundamental aspects of program synthesis as well as how to use it to automate simple tasks and boost their productivity.

Detailed course structure The course will comprise of many small projects that expose the features of different programming paradigms. Among others, we will cover functional, logic, and probabilistic programming and also introduce students to SAT and SMT solvers [41].

• **Enumerative synthesis in a lazy functional language** In the first part of the course, the students will learn the basics of a functional programming language. One of the important topics in functional programming
is laziness—i.e., the ability to evaluate a program on a “by need” fashion. To let the students better appreciate the power of laziness we will have them build a synthesizer that implements, with and without laziness, enumerative search for a term that agrees with a set of examples.

- **Symbolic search using logic programming** The second part of the course covers logic and declarative programming. For this domain the students will have to complete a project in which they use the declarative aspects of logic programming to build a small symbolic synthesizer. For example, the students will build a simple tool for generating Boolean circuits that meet a certain functional specification [91].

- **Advanced paradigms: probabilistic programming** In the last part of the course, the students will be exposed to emerging programming paradigms, including probabilistic programming. By the time we will offer the course for the first time in year 3, we expect our inference and synthesis tools for probabilistic programs to be mature enough to have the students use them to complete a small project for synthesizing a simple probabilistic network switch like the one presented in Example 1.1.

**Deployment plan** As discussed in the project management plan (§ 7), we will offer this course for the first time in the third year. Over the course of this proposal’s time frame, we will teach the course three times, allowing us to evaluate effectiveness of material, receive feedback from students, and incrementally refine and improve course content.

### 6.2 Visual Art with Synthesis

**Goals and plan** We will organize student workshops where the attendees will use program synthesis to create forms of visual art. Our goal is to capitalize on the ability of visual arts to attract a diverse set of students (potentially from different majors) and to expose them to the idea of program synthesis. Specifically, we plan to hold three annual synthesis-themed workshops over the time-frame of this proposal. (We budgeted funds for organization.) Student projects will be judged for their merit by a panel of faculty from departments across campus—e.g., CS and Art. Students will learn how synthesizers can automate repetitive tasks and use this idea to generate artistic patterns from examples of interactions. We plan to have prototypes of our synthesizers usable by the time of the first workshop, so that students can rapidly play with their ideas over the short course of a workshop. Before the first official workshop, we will run a small beta workshop to ensure robustness of our synthesis infrastructure. We will also use other synthesizers such as Prose [77] and Sketch-n-Sketch [57]. Prior to each workshop, the PI and his graduate students will give an introductory tutorial about program synthesis, the expectations from participants, and how to use the given tools. We will also seek organizational help from the UW Art Institute [59].

**Diversity plan** The PI is very aware of the inclusiveness and diversity concerns surrounding computer science events. It has been repeatedly observed in the tech community that workshops like the ones we propose tend to isolate under-represented minorities in computer science, a phenomenon that has even gained coverage by mainstream corporations and media [95]. The PI will take concrete measures to ensure a welcoming and inclusive environment: We will (i) draw inspiration from other tech conferences and workshops that have been successful in being more inclusive [22]; (ii) organize all workshops in close collaboration with WACM [92], Wisconsin’s chapter of ACM Women in Computing, and the Wisconsin Emerging Scholars-Computer Sciences [94], a program to recruit a broader cross-section of UW students to the field of computer science, particularly women and under-represented minorities; (iii) consult with the Office for Equity and Diversity at UW-Madison [73], a vital campus resource that provides guidance on implementing diversity strategies.

### 6.3 Outreach and Undergraduate Research

**Undergraduate student research** The PI strongly believes that getting hands-on experience in an exciting research environment is crucial to motivate students to pursue graduate studies and we have budgeted funds for undergraduate students. The PI has advised 5 undergraduate students during his first 2 years at UW-Madison. Students were recruited during the PI’s undergraduate compilers course. His first advisee, Fang Wang, helped designing the experimental evaluation of new algorithm that was published at MFPS XXXIII [39] and is now a graduate student at CMU. The other students have read papers from formal methods conferences and implemented or extended the techniques presented in the papers. Two of them are writing a paper with their findings. Leveraging the UW-Madison diversity resources mentioned above, the PI will ensure hiring of women and minorities for these positions.
Lectures on the impact of synthesis on the workforce  Looking forward at the potential of automated program synthesis, the PI is concerned about its use to replace tasks in the labor force—e.g., TAs with automatic graders. To make students aware of this issue and of the possible countermeasures, we will organize a series of lectures that look at this problem from various angles [71], including examining existing jobs where machines have replaced humans. The lectures will mostly target CS seniors and graduate students, who will soon enter the workforce. However, we will ensure that lectures are accessible to everyone and benefit the campus community at large. Our goal is to ensure that CS students are aware of the possible future landscape and how similar phenomena were handled in the past [80]. For the lectures, we will seek experts from across campus, for example, Dr. Pilar Ossorio, who studies ethics in the workplace, and with whom the PI is collaborating on another project on algorithmic fairness.

7 Project Management Plan

Figure 3 shows our proposed timeline for the five-year time frame of the proposal. We have budgeted support for one graduate student who will work with the PI on the research proposed in this document. In addition, we have budgeted annual summer support for an undergraduate researcher, who will be thoroughly involved in the project, both in terms of implementation and research activities.

Research Thrust 1  We will begin our research activities with the first research thrust. Specifically, we have allocated overlapping two-year investigation periods for the first two algorithmic problems outlined in Thrust 1. The third problem—i.e., the design of the probabilistic fragment of our framework—will start in parallel in the second year. Our goal is to lay the grounds for research Thrust 2 earlier in the timeline.

Research Thrust 2  The investigation of our probabilistic algorithms will start in year 2, hand in hand with making progress in the design of the probabilistic specification mechanism from Thrust 1. We will continuously pursue evaluations and benchmark collection efforts throughout the project timeline.

Education activities  We believe that deploying a successful course will require careful planning and tool development; we thus allocate the first two years for course development. In the second year, we will also begin planning the synthesis-themed visual art workshop, which will be held in years 3-5. The proposed course and workshops are carefully planned in alternating terms, thus allowing a senior batch of students who have been exposed to synthesis to guide junior students. The proposed lecture series will be scheduled throughout the project and is not shown in the timeline.

Prior NSF support  Loris D’Antoni is a PI on ongoing NSF Award CCF-1637516: AitF: Collaborative Research: Foundations of Intent-based Networking, 09/01/2016 to 08/31/2018, $339,985. This project aims to develop techniques to automatically produce network configurations that satisfy a set of high-level policies and objectives. This project will help realizing the vision of intent-based networking by allowing network operators to design data and control planes declaratively and without having to provide device-specific configurations. The developed techniques will also be used to create automatic techniques to provide feedback to students learning networking concepts. The work has already resulted in one publication [89]. The PI received $20,000 through CCF-1650816 for organizing the Programming Language Mentoring Workshop at POPL’17. This grant allowed 12 students from under-represented minorities to attend POPL’17 and hear about career opportunities in programming languages. The PI is a co-PI in the proposal SHF: Medium: Formal Methods for Program Fairness, which will start in September 2017.
References


