Due to the ubiquity of computing, programming has started to become an essential skill for an increasing number of people, including data scientists, financial analysts, and spreadsheet users. While it is well known that building any complex and reliable software is difficult, writing even simple scripts is challenging for novices with no formal programming background. Therefore, there is an increasing need for technology that can provide basic programming support to non-expert computer end-users.

The goal of my research is to democratize programming and make it possible for millions of people around the globe to automate otherwise tedious tasks using programming. To achieve this goal, I develop novel and foundational program synthesis techniques that automatically generate programs from high-level specifications (such as input-output examples) for a wide spectrum of application domains (such as data wrangling and data migration). I believe this is an important and necessary first step towards making programming accessible to a broader community. More specifically, my research focuses on devising efficient general-purpose program synthesis algorithms by leveraging and advancing the state-of-the-art in program analysis and automated logical reasoning [3, 4, 5, 6, 7, 8]. Rather than basing program synthesis on brute-force enumeration of programs with ad-hoc heuristics (as in prior work), my research pioneers the use of program abstractions in program synthesis and leads to a principled synthesis framework. My vision is that program synthesis technology will greatly assist humans in tasks that involve programming effort in the future. Broadly speaking, my research falls at the intersection of programming languages and software engineering and aims to develop artificial agents with programming capabilities.

Challenges in program synthesis. Program synthesis enables creating programs from specifications. In order for this technology to have broad impact, it must support specifications that can be easily provided by non-expert computer end-users. In particular, inductive specifications (such as input-output examples) have proven to be quite useful for non-expert users. As a result, inductive program synthesis has been successfully used to automate various tasks such as writing simple text transformations and SQL queries. However, for this technology to have truly transformative impact on society, I believe that three fundamental challenges must be addressed:

• Broad applicability. Most existing program synthesis algorithms are extremely domain-specific and automate a very specific class of tasks. While the domain-specific nature of synthesis algorithms has partly been responsible for their success, it requires deep expertise in program synthesis in the development process and prevents the wide-spread adoption of synthesis in a broader range of application domains. Therefore, there is a pressing need for developing foundational program synthesis algorithms that can be re-used across many domains.

• Efficiency. Because any practical synthesis system is expected to interact with human users, it must synthesize programs in real time. Therefore, a key challenge in program synthesis is to achieve this real-time interactivity while being able to synthesize sufficiently interesting programs.

• Generalization power. Since inductive specifications (e.g., input-output examples) are inherently incomplete, a program that satisfies the specification may not actually meet the user’s intent. Therefore, for program synthesis to be useful, the synthesizer must have sufficient “inductive bias” to “guess” the intended programs from very incomplete specifications. Hence, a third major challenge in this area is to develop synthesis algorithms that have good generalization power (i.e., synthesize programs that are likely to generalize beyond the given specification).

My research has addressed these three challenges by developing efficient and foundational synthesis algorithms that can be re-used across different application domains. I have proposed novel program synthesis algorithms that leverage tree automata and program abstractions and developed a unified synthesis framework that is both efficient and has good generalization power. In particular, I have introduced a generic synthesis algorithm that compactly represents the underlying search space using tree automata [5]. This technique is independent of any application domain and allows effective inductive bias to be injected into the synthesizer in order to improve generalization power. Another key contribution of my research is to use program abstractions for program synthesis [4].
Rather than basing synthesis on brute-force enumeration of programs with ad-hoc heuristics, my research allows the synthesizer to use smart program abstractions to prune the search space in a principled way. Based on these insights, I have developed a general-purpose program synthesis framework and successfully applied it to several challenging application domains (such as data wrangling, matrix transformations, data imputation, and data migration). Using these ideas, I have obtained orders of magnitude improvement in terms of synthesis time across different application domains compared to state-of-the-art synthesis techniques. In what follows, I first describe the key contributions of my doctoral research; then I outline three research directions I plan to pursue in the future.

**Research Contributions**

*Synthesis with tree automata.* A key challenge in inductive program synthesis is to efficiently explore an enormous search space and find a program that is likely to generalize. To cope with this challenge, inductive program synthesizers typically construct a so-called “version space” which represents all programs that satisfy the specification and then pick a “best program” in this version space. This notion of “best program” introduces inductive bias that allows the synthesizer to obtain a program which is likely to generalize. In this context, the key challenge is to represent the version space *compactly* because, the larger the version space, the less efficient the synthesis algorithm becomes.

During my doctoral research, I have proposed a novel version space learning technique based on tree automata that represents the version space more compactly than prior techniques [5]. The key insight that allows me to obtain such a compact representation is by clustering programs into equivalence classes based on their behaviors. Specifically, given a programming language in a context-free grammar and a specification in the form of input-output examples, this technique constructs a finite tree automaton where states correspond to *constants* (i.e., concrete output values of programs) and transitions are constructed using the *concrete* semantics of the given programming language. This version space learning approach has broad applicability and can be applied to synthesizing programs over any context-free grammar. It is also efficient in practice because all programs that produce the same output on the given input are compactly represented using one state.

I have implemented a general-purpose synthesis framework, called BLAZE, based on these ideas, and empirically demonstrated its broad applicability and efficiency by applying the framework to automate a wide range of computer end-user programming tasks. For instance, my OOPSLA 2017 work [5] shows how to instantiate this framework for data imputation, an important task that involves programmatically filling missing values in tabular data. As another example, my VLDB 2018 paper [8] demonstrates how to apply this idea to automatically migrate hierarchical data into relational tables. Besides being quite general, this technique also significantly outperforms prior approaches in terms of synthesis time. For instance, as demonstrated experimentally in my OOPSLA 2017 paper, this technique results in an average of $22\times$ speed-up compared to prior techniques when automating data imputation tasks.

*Leveraging program abstractions.* In addition to having a compact representation of the underlying search space, another key prerequisite for efficient synthesis is to effectively reduce the size of the search space. In my doctoral research, I have also explored effective search space reduction techniques by leveraging program abstractions. In particular, my POPL 2017 work [4] introduces a novel synthesis paradigm, called SYNGAR, which is the first program synthesis algorithm that is based on *counterexample guided abstraction refinement*. The key insight underlying this paradigm is to cluster programs into equivalence classes based on their *abstract* behaviors (rather than concrete behaviors as in my prior OOPSLA 2017 paper): Because many more programs share the same abstract behavior, this approach is able to reduce the search space more dramatically. More specifically, the SYNGAR paradigm consists of an *Abstractor* and a *Refiner*: Given an abstraction (i.e., a set of predicates), the goal of the Abstractor is to synthesize a program that satisfies the specification according to the *abstract* semantics of the programming language. This abstract synthesizer is still based on tree automata but states now represent *predicates* (rather than constants as in my OOPSLA 2017 work). Since a predicate corresponds to a *set* of constants, the version space becomes much smaller compared to the alternative and therefore the Abstractor is much more efficient.

One implication of such an abstraction-based approach is that the synthesized programs may now be *spurious*. That is, a program synthesized by the Abstractor may not actually satisfy the specification according to the concrete semantics. To deal with this issue, our approach iteratively eliminates the spurious programs by automatically refining the abstraction (i.e., adding new predicates). Specifically, my POPL 2017 work proposes a novel counterexample...
guided abstraction refinement algorithm that refines the abstraction by constructing an *incorrectness proof* for any spurious program, and the new abstraction includes all predicates that are used in the proof. Using this new abstraction, the Abstractor is able to not only refute the spurious program, but also prune out many other programs that are incorrect for similar reasons.

I have incorporated this SYNGAR paradigm into the BLAZE synthesis framework from my earlier work [5] and obtained orders of magnitudes improvement (up to $450 \times$) in terms of synthesis time. Furthermore, the instantiation of the BLAZE framework using this idea significantly outperforms state-of-the-art tools across different application domains (e.g., string transformations and matrix manipulations) both in terms of what it can synthesize as well as in terms of synthesis time.

**Learning abstractions for program synthesis.** While the two aforementioned ideas allowed me to build a practically efficient program synthesis framework that can be instantiated in different application domains, a developer still has to manually provide a suitable abstract domain (i.e., predicate templates) and the corresponding abstract semantics when applying the framework to the chosen application, which can be a non-trivial task. My CAV 2018 work [3] has addressed this usability issue by automatically learning a suitable abstract domain as well as abstract semantics for an abstraction-based synthesis framework. In order to learn an abstract domain that is both expressive and concise, our key insight is to incrementally enrich an initially coarse abstract domain whenever necessary. In particular, our technique learns an abstract domain by employing the synthesis framework to solve a training set of problems. More specifically, our approach first mines all predicates that are necessary to solve all training problems; then, it generalizes the mined predicates into *predicate templates*, which form an abstract domain that is likely to generalize to synthesis problems beyond those in the training set. Empirically, in addition to significantly reducing the manual labor involved in applying the BLAZE framework to a new application domain, our abstraction learning technique also speeds up synthesis. For instance, in the two application domains that I have looked at (i.e., string and matrix transformations), the automatically learned abstractions outperform human-crafted ones by $8.3 \times$ and $9.2 \times$ (on average) in terms of synthesis time.

**Other research in program analysis.** In addition to the three pillars of my dissertation work mentioned above, my research has also addressed some of the key problems in program analysis, which is the dual of program synthesis. Instead of synthesizing a program that satisfies the given specification, program analysis aims to prove the correctness of a program with respect to the given specification. For example, our OOPSLA 2015 work [2] proposed a general framework for statically answering a broad class of inter-procedural control-flow queries that frequently arise in many static program analyzers. As another example, I have also worked on pointer analysis, which is a fundamental problem that underlies any static program analysis and developed a novel compositional pointer analysis [1] that is both scalable and precise. Both of these two techniques have important applications in security, bug detection, and program understanding, and have been used for identifying implementation issues (performance bugs, security vulnerabilities, etc) in Java programs.

**Future Directions**

In my past doctoral research, I have advanced the state-of-the-art in program synthesis by introducing novel and fundamental algorithms that broaden the applicability of synthesis, improve search efficiency as well as allow good generalization power. However, in order to realize my vision of making synthesis-aided software development part of our everyday toolkit, there is still much work to be done. In particular, I plan to (a) expand the ways that humans interact with synthesizers by leveraging multiple modalities of information, (b) identify emerging and important application domains where program synthesis can help, and (c) combine logical and statistical techniques to create program synthesizers that can “learn to learn”. In what follows, I briefly outline three research directions that I plan to pursue in the future.

**Multi-modal program synthesis.** While there has been great progress in program synthesis in the past few years, almost all existing synthesis techniques use a single mode of specification, such as input-output examples or natural language alone. However, in reality, users typically convey their intent in a *mix* of such modalities. For instance, consider descriptions in a programming assignment or a programming-related question on StackOverflow: These descriptions typically involve a natural language explanation of the problem to be solved, together with some examples
demonstrating how the solution should behave on “interesting” inputs. Different kinds of specifications describe the task at different levels. I believe that using a combination of such forms will make synthesis-aided software easier to use as well as help the synthesizer to learn the intended programs more efficiently. Therefore, I am interested in exploring multi-modal program synthesis techniques that synergistically combine a multitude of different forms of specifications (such as input-output examples, natural language, demonstrations, types, and logics). I have recently started exploring this idea in the context of synthesizing regular expressions from a combination of natural language and examples, and I believe that there are many interesting challenges to be solved. Because this kind of research is inherently cross-disciplinary, I plan to pursue collaborations with colleagues in relevant disciplines, particularly in natural language processing and software engineering.

Synthesizing program analyzers. Program analysis techniques are becoming increasingly useful for software engineering, maintenance, and security, especially given the emergence of modern web applications, mobile devices, and blockchain technology. However, developing effective program analyzers is non-trivial, as it requires deep expertise in program analysis, insights from application domains, as well as strong software engineering skills. I believe that this is a domain where synthesis can help: in particular, I am intrigued by the idea of synthesizing useful program analyzers from data. As a starting point, I plan to focus on synthesis of “linters” which are static analyzers that are capable of flagging certain errors (such as syntactic mistakes and structural problems) in source code. Specifically, given an API and examples of bug fixes in clients of the API (which can be collected from commit histories in online repositories), I believe it is feasible to automatically synthesize a simple analyzer that detects incorrect usage patterns of that API. As another example, I am also interested in applying synthesis techniques in software testing. In particular, I plan to explore techniques that automatically synthesize input patterns that lead to security vulnerabilities (such as denial of service and side-channels) from concrete vulnerable inputs. An input pattern defines a family of inputs, which can help developers analyze software behavior, improve test coverage, as well as fix bugs. Since it is possible to write a program to generate a family of vulnerable inputs, I believe it is feasible to use program synthesis to automatically construct such an input-generation program from a few vulnerable inputs. More broadly, I believe inductive program synthesis as a technique for generalizing concrete program behaviors into a program would be useful in analyzing and understanding real-world codebases. In addition to bringing together my expertise in program synthesis and program analysis, this research direction would also help improve the productivity of software developers in finding bugs and security vulnerabilities without requiring deep program analysis expertise.

Data-driven program synthesis. Program synthesis has been tackled by the programming languages community using formal and symbolic approaches such as automated logical reasoning and program abstractions. While these techniques have achieved promising results, they rely heavily on human-crafted knowledge and lack the ability to learn from data and prior experience (in the form of past synthesis attempts). I believe that future program synthesizers should be able to learn from existing data and improve themselves using past experience. For instance, all existing symbolic synthesis techniques (including my own work) employ domain-specific inductive bias (in the form of a search strategy or so-called “ranking function”) as a critical component for effectively prioritizing the search and selecting the “best” program. These design decisions are typically made by human experts after manually solving dozens of synthesis problems. Apart from requiring deep domain knowledge, this design process is also tedious and requires several iterations in order to converge on good heuristics. I believe that such inductive bias can be effectively learned from existing data and past experience using machine learning. For example, the ranking function in my synthesis framework can be replaced by a statistical model: The model is trained on a set of solved synthesis tasks and learns how to select the intended programs from a version space. Besides completely automating this otherwise tedious design process, the machine-learned heuristics also have the potential to better guide the search and lead to more efficient synthesis compared to human-crafted heuristics. More broadly, I envision a future where synthesizers consist of both symbolic components (e.g., the overall search architecture) as well as learned statistical components (e.g., the search strategy). Therefore, I plan to explore possibilities of synergistically combining symbolic techniques and learning-based approaches and develop frameworks that not only “learn” but also “learn to learn”.

4
References


