Constructing *correct software*—the kind that always does exactly what we expect it to do—has been traditionally associated with prohibitively high cost. My research challenges the accepted trade-off between correctness and cost, aiming to improve software quality and increase programmer productivity at the same time. **My agenda** is to exploit the growing power of automated logical reasoning to enable the next generation of *declarative* programming languages, in which the programmer directly states high-level system requirements (such as constraints on input-output behavior, resource bounds, or security policies) and the language infrastructure makes sure that these requirements are satisfied in every execution of the system.

My current research focuses on *program synthesis*, a technology that uses powerful search algorithms to find executable programs that satisfy a given declarative specification. The biggest obstacle to practical program synthesis is the size of the search space a synthesizer has to explore, which grows extremely fast with the complexity of the solution and the number of available building blocks. In addition, even checking whether a single program from the search space respects the specification on all executions—also known as *program verification*—is a hard, generally undecidable problem. As a result, state-of-the-art synthesis techniques avoid using verification as an oracle during search. Instead, they search for a program that behaves correctly on a small set of representative executions, resorting to verification only to validate the result, if at all.

Contrary to this trend, my work shows that using verification during search actually pays off. My **core insight** is that, despite the high cost, verification can be surprisingly effective at pruning the space of candidate programs, as long as the verification mechanism is designed to be *synthesis-friendly*. Based on this insight I developed Synquid [3], a synthesis-aided programming language for data structure manipulations. In Synquid, the programmer describes a data structure and the abstract effect of a manipulation (for example, "add an item to a set represented as a binary search tree"), and relies on the synthesis engine to discover a provably correct implementation. Synquid can generate programs that go beyond the state of the art in terms of complexity of data structures they manipulate: it is **the first synthesizer** to automatically discover verified implementations of textbook sorting algorithms, as well as operations on binary search trees, Red-Black Trees, and AVL Trees. For these programs, the required specifications are 1.5–7 times more concise than the implementations, and the synthesis times range from fractions of a second (for insertion sort) to under a minute (for Red-Black Tree rotations).

While data structure manipulations are a good test bed for synthesis technology, my ultimate goal is to automate everyday programming tasks, especially those that are both critical and error-prone. A prototypical example of such tasks is enforcing *information flow security*. Traditionally, it requires programmers to insert policy checks throughout the code, with any missing check potentially causing an information leak. To automate this tedious and error-prone task, I created a synthesis-aided language called Lifty [7], where the programmer specifies each information flow policy once, as a logical predicate associated with a piece of sensitive data. The Lifty compiler (which, like Synquid, is powered by a synthesis-friendly verification mechanism) automatically inserts the necessary checks throughout the program. Unlike existing policy enforcement techniques that rely on run-time analysis, Lifty does not incur any performance overhead relative to manually written checks. In a prototype implementation of a conference management system, Lifty was able to support complex, realistic policies (such as "a reviewer who is in conflict with a submission cannot learn who else is in conflict") and synthesize all necessary policy checks, including those required to prevent a set of reported real-world information leaks.

In what follows, I first describe my dissertation work on practical program verification and how it motivated my current research on synthesis-aided languages; then I outline three research directions I plan to pursue in the next five years.

Research Contributions

Tools for Program Verification. My interest in practical tools and techniques for constructing correct software began with *automated deductive program verification*. In this approach the programmer annotates executable code with a specification of desired behavior, and the verifier relies on an automatic reasoning engine (most commonly an SMT solver) to find a proof that the code satisfies the specification. When the verifier fails to discover a proof automatically, the programmer guides it by annotating the code with hints, such as loop invariants or lemmas.

Within this paradigm, I developed a verification technique for security-critical C code, which I used to verify secrecy and authentication guarantees of Microsoft's Trusted Platform Module [4]; the technique helps bridge the abstraction gap between high-level security properties and a low-level implementation, using an encoding of stepwise refinement in a general-purpose program verifier. For object-oriented programs, I introduced a methodology for reasoning about complex inter-object dependencies called *semantic collaboration* [6], which I implemented as part of the AutoProof verifier [8] for the Eiffel programming language. Using semantic collaboration, I completed **the first verification** of full functional correctness for a complete, general-purpose object-oriented container library; this work received a best paper award [5].

Having had first-hand experience with developing realistic verified code, I was dissatisfied with the amount of effort and expertise it required. To mitigate this problem, I contributed to several popular program verifiers with the goal of improving their interaction with the programmer. I extended the Dafny verifier with support for *calculational proofs* [1], which provide an elegant and efficient mechanism for guiding the tool through a series of intermediate steps in proving complex properties; the mechanism has since been widely used in large-scale Dafny verification efforts, such as Ironclad Apps and IronFleet. I also developed a plug-in for the Boogie verifier, which helps programmers understand and debug failed verification attempts by generating small, concrete test cases that showcase the error [2].

Even with the best notations and tools, verification places a significant burden on the programmer, who has to supply both the code and the specification (and often the additional hints). If the goal is to guarantee deep properties of the system, a detailed specification is indispensable; therefore, a promising approach to reducing the cost of correct software is to cut on the amount of executable code programmers have to write. As a result, I became interested in the design and implementation of synthesis-aided programming languages.

Synthesis-Aided Languages. The goal of a synthesis-aided language is to make it easier for programmers to express their intent. Perhaps the most popular synthesis-aided paradigm today is *programming by example*, where describing a program amounts to providing input-output examples that demonstrate its desired behavior. The downside of this paradigm is that correct behavior is only guaranteed on the given inputs, making it inapplicable for building high-assurance software. Moreover, the number and size of examples required to define a complex program can be quite large, and crafting outputs often requires a detailed knowledge of the algorithm: imagine using examples to specify Red-Black Tree insertion.

I believe that the best way to help programmers build correct software is to give them better means of decoupling different aspects of program behavior. For example, in Synquid [3] the programmer specifies *what* a data structure operation should achieve ("add an item to a set" or "sort a list") separately from *how* to achieve it efficiently ("represent the set as a binary search tree" or "use merge operations on sorted lists as a building block"). Similarly, Lifty [7] decouples the core functionality of an application from its security policies, allowing the programmer (or a domain expert) to specify the policies separately from the rest of the code.

Having designed a language that helps programmers express their intent, the remaining challenge is to develop a synthesis engine that can translate this intent into provably correct executable code. My work addresses this challenge through the design of synthesis-friendly verification techniques. For example, at the heart of SYNQUID's synthesis engine is a new verification technique called *local liquid type checking*, which is able to analyze an *incomplete* candidate program and discover that it cannot be completed to a valid solution, whereby discarding a large portion of the search space with a single check. For LIFTY, I designed a verification mechanism that features strong error localization: when it discovers an information leak, it is able to pinpoint the exact source of sensitive data that caused it, and infer the minimal policy change sufficient to make the program secure. This allows LIFTY to decompose the global problem of policy enforcement into simple, independent synthesis problems.

Future Outlook

Synthesis for Cross-Cutting Concerns. Information flow security is an example of a cross-cutting concern that can be factored out from the rest of the program behavior with the help of synthesis technology, as demonstrated in my work on LIFTY. I am interested in designing languages that factor out other cross-cutting concerns, such as more broad security and privacy considerations, exception handling, memory management, and resource consumption. In collaboration with Jan Hoffmann at CMU, I am currently developing a language for *resource-agnostic programming*, which aims to free the programmer from low-level reasoning about performance. Our approach combines synthesis with automated resource bounds analysis to rewrite programs in a way that preserves their input-output behavior and achieves optimal run time, memory, or energy consumption.

Different application domains have different cross-cutting concerns, and if it takes a programming languages expert to design a custom synthesis engine for each domain, this research will have a limited impact. Hence the long-term goal is to develop a general framework that allows programmers to define domain-specific cross-cutting concerns and specify what it means for a program to respect these concerns, leaving it to the language infrastructure to figure out the best way to integrate the new behavior into the rest of the program.

Verification Techniques for Synthesis. Designing synthesis-friendly verification techniques proved to be a fruitful strategy in two distinct problem domains: data structures and information flow security. Our experience suggests that this strategy can give rise to scalable synthesis algorithms in many other domains and help automate the development of some of the most intricate and critical software, such as device drivers, security protocol implementations, and cyber-physical systems. Consider a robot that makes decisions based on data derived from multiple sensors, each with its own trade-off between accuracy and cost. Building upon a suitable verification technique, capable of reasoning about uncertainty, we could design a synthesis-aided programming language that takes care of configuring the sensors for each safety-critical action in order to guarantee sufficient confidence in the data while minimizing cost.

Programmer's Assistant. While declarative programming is an attractive long-term goal, synthesis technology can also be used to improve programmer productivity in mainstream programming languages. To this end, I plan to develop a *programmer's assistant*, which enhances a traditional development environment with real-time *program completion* and *program repair*. The challenging aspect of building such an assistant is that synthesis engines normally rely on the programmer to provide a declarative specification that serves as a correctness criterion. In this

setting, however, it is unreasonable to expect the programmer to write a detailed specification *in addition* to the (partial or incorrect) implementation; instead, the correctness criteria must be inferred from existing code. A promising approach to this problem is to combine three complementary sources of information. First, we can derive user's likely intent from their own code; this requires developing novel specification inference methods, that are highly automated and use probabilistic reasoning to handle potentially faulty programs. Second, we can leverage testing code that often accompanies software systems. Tests are a convenient source of concrete examples of desired behavior; the challenge, however, is to develop a synthesizer that can efficiently make use of both logical specifications (inferred from the system) and concrete examples (obtained from tests). Finally, we can take advantage of large code bases, building upon existing machine learning techniques to bias the search towards "more natural" programs.

References

[1] K. R. M. Leino and N. Polikarpova. Verified calculations. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, 2013.

[2] N. Polikarpova, C. A. Furia, and S. West. To run what no one has run before: Executing an intermediate verification language. In *Runtime Verification (RV)*, 2013.

[3] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. In *Programming Language Design and Implementation (PLDI)*, 2016.

[4] N. Polikarpova and M. Moskal. Verifying implementations of security protocols by refinement. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, 2012.

[5] N. Polikarpova, J. Tschannen, and C. A. Furia. A fully verified container library. In *Formal Methods (FM)*, 2015. **Best Paper Award**.

[6] N. Polikarpova, J. Tschannen, C. A. Furia, and B. Meyer. Flexible invariants through semantic collaboration. In *Formal Methods (FM)*, 2014.

[7] N. Polikarpova, J. Yang, S. Itzhaky, and A. Solar-Lezama. Type-driven repair for information flow security. *CoRR*, abs/1607.03445, 2016.

[8] J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova. Autoproof: Auto-active functional verification of object-oriented programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2015.