What are the foundations of safe, secure, and reliable software? How do we translate the rich logics and theories at the core of this foundation into practical programming languages and tools? I answer these questions by studying programming languages and systems, in particular type theory, functional programming, and compilers. In my research, I move back and forth between (1) developing foundational theories to understand and control how programs behave and (2) applying key insights from those theories toward solving practical programming problems.

My dissertation work focuses on developing foundations for type-directed program synthesis and exploring how this technology can support semi-automated programming in functional programming languages. I have also explored how this synthesis technology can be used to support programming and formal reasoning education at scale. Previously, I explored applying type theory and logic to other practical programming problems including low-level memory safety and language interoperability.

Type-directed Program Synthesis

When we formally verify a software artifact, we go through great lengths to write a precise specification that describes the artifact’s intended behavior. Could we use this specification to automatically generate the program in question? This process, program synthesis, has been studied since the 70s, but has seen a recent revival thanks to advances in automated theorem proving technology, general increases in computing power, and interests in semi-automated (rather than fully automated) and domain-specific (rather than general purpose) synthesis procedures [1].

With David Walker and Jonathan Frankle (Princeton University), I have focused on developing program synthesis techniques within the domain of richly-typed functional programming languages. Static types are one of the most ubiquitous form of lightweight specification, appearing in many programming languages. To study how we can leverage types within the synthesis process, I developed a core calculus, $\lambda_{syn}$, that utilizes techniques borrowed from intuitionistic proof theory to synthesize recursive functional programs with algebraic data types from input-output examples. I then implemented these ideas into a prototype synthesizer, Myth, that synthesizes programs compatible with OCaml or F#. This prototype out-performs the previous state-of-the-art systems in terms of size of recursive functional programs it can synthesize (upwards of 70 abstract syntax tree nodes). It also synthesizes programs at a comparable speed to previous work (sub-seconds in most cases) [2]. Myth demonstrates that type-directed synthesis is an effective synthesis for functional programs.

Tools for Programming Education at Scale

We can use program synthesis to support computer education at scale either in large classrooms or online courses. In these teaching scenarios, we would like to be able to provide students with as much individualized support and feedback as possible. Synthesis technology has previously been applied toward providing automated feedback for introductory programming assignments [3]. My current work provides similar kinds of support for teaching formal reasoning to undergraduates, in particular proofs by induction.
The focus of my work in this area is **INDUCTFUN**\(^1\), an online module for teaching mathematical induction by way of learning functional programming. **INDUCTFUN** is backed by the Coq proof assistant and provides a front-end where students can write small functional programs, properties about those programs, and proofs of those properties that are all checked by the system for correctness [4, 5]. Because Coq utilizes a dependently-typed programming language at its core, I can use type-directed synthesis techniques to analyze proofs and provide meaningful feedback or automatically generate problems depending on the sorts of proofs I would like my students to write.

I have also investigated alternative synthesis techniques in the context of education, in particular, integrating machine learning with compiler technology to analyze code. I supervised a group of senior undergraduates — Nathan Close, Amalia Hawkins, and Rupi Sureshkumar — who took these initial ideas and developed a tool, **JUDGMENT OF CODE STYLE**, that graded introductory programming assignments for style [6]. Our tool generated appropriate feedback by analyzing submissions with natural language processing and machine learning techniques.

**Previous Research**

In addition to program synthesis and programming education, I have also worked on importing ideas from type theory and logic to several other domains.

**Low-level Memory Safety**  Traditional inquiries into the safety of systems languages focus exclusively on C. However, C++, for all of its faults, is a more type-safe variant of C that is also widely used as a systems language. Unlike C, good C++ code is type-safe, affording us extra information during compilation that we can use to make the necessary pointer checks more efficient.

With Santosh Nagarakatte (Rutgers University), Milo Martin, Christian DeLozier, and Richard Eisenberg, we identified a safe subset of C++, **IRONCLAD C++**, that guarantees memory safety with good performance when used in conjunction with our safe libraries [7]. **IRONCLAD C++** builds upon previous efforts for efficient temporal and spatial memory safety [8, 9] by adapting those techniques to C++. In addition to collaborating on the design of the language and libraries, I led efforts to formalize this subset of C++ and prove that it is safe [10].

**Language Interoperability**  Most software systems are created not with a single language but instead with a collection of languages each carefully chosen to solve a particular set of problems within those systems. Typically the facilities provided for these languages to interoperate are constructed in an ad-hoc fashion and are thus tedious and error-prone to use [11]. Previous research has built safe interoperability facilities for mainstream languages such as C and Java. However, when dealing with languages that possess more elaborate type systems, we must ensure that the safety properties of the languages are preserved in reasonable, predictable ways.

With Vilhelm Sjöberg, we explored semantics-preserving interoperability for dependently-typed languages [12]. I developed a core calculus, SD, that models interop boundaries between simply-typed and dependently-typed lambda calculi with data types. With this calculus, we identified the dynamic checks necessary to preserve the guarantees made by dependently-typed code even while executing code in simply-typed contexts. Such checks could be lifted to languages with dependently-typed programming facilities such as Idris or Haskell to provide rich, safe interoperability facilities with other languages.

---

\(^{1}\)Available at [inductfun.org](http://inductfun.org)
Future Directions

I am currently continuing my efforts to develop the foundations of type-directed program synthesis. Looking into the future, I am eager to explore its various applications, particularly with respect to semi-automated programming and education. The scope of type-directed program synthesis and its applications is quite expansive, and I have begun to establish collaborations with other researchers at peer institutions whom are interested in my work to expand the program even further.

Program Synthesis Foundations  In my dissertation work, I have identified two primary dimensions to extend type-directed program synthesis further. One dimension consists of exploring and making concrete the connections between type-directed program synthesis and other synthesis techniques such as those imported from formal verification. The key distinction between these techniques is their treatment of the constraints that they put upon the synthesized program: either push constraints down into related synthesis sub-problems or bubble them outward to be solved later. By studying this behavior, we can better understand how to integrate the various synthesis techniques in the literature. Furthermore, insights about how proof theoretic and formal verification techniques can intermix in synthesis can be lifted to other domains where they have been traditionally separate, e.g., type inference and proof search.

The other dimension consists of studying how more elaborate type systems can be integrated into the synthesis process. These type systems have been used in the past to assist programmers in a variety of tasks such as specifying domain-specific languages, enforcing protocols, tracking information flow and privacy, and verifying arbitrary properties of programs. Type-directed synthesis becomes a force multiplier for applying type theory toward programming: any type system built to assist programmers with a particular problem becomes a synthesizer of programs that solve these problems.

Program Synthesis Applications  In addition to building up the foundations of type-directed program synthesis, I am also interested in exploring how this technology can be applied to create practical programming tools. In particular, this form of program synthesis suggests a semi-automated synthesis tool that the user interacts with and guides to generate a final program. For example, the user may write a partial program, get stuck, and use the synthesizer along with input-output examples or other constraints to fill in enough of the program to unblock themselves. This sort of tool-assisted programming is common when using proof assistants like Coq and Agda but not for general programs. Such a programming paradigm raises a whole set of intriguing questions about how such a tool should be designed and developers ought to work in tandem with these tools that I am eager to tackle in the future.

References


Peter-Michael Osera — Research Statement


