Research philosophy
In my research, which falls in the area of programming languages design and implementation, I am mostly interested in the design of scalable, robust, and intellectually manageable methodologies for program analysis and verification. I increase understanding of principles of software construction with a concrete goal in mind: development of improved tools for computer-aided programming and verification.

My recent research projects targeted the construction of static analyses for higher-order programming languages and verification methodologies for concurrent code. My approach is to first rigorously develop a clean and simple formal model. This challenging and deceivingly time-consuming task pays off with a clear understanding of core principles underlying an analysis or a verification tool, enabling optimization of existing techniques and development of new approaches. In addition, the formal model led the way to abstractions that resulted in scalable and reusable implementations.

In the rest of this document, I will briefly elaborate on the key advances made in my recent research projects on static program analysis and verification of concurrent programs, and conclude with my vision of the future work in these two directions as well as in related areas. To keep the presentation focused, I will not discuss my previous work on semantics inter-derivation [4, 7, 8], type systems for object encapsulation [1, 9], design of domain-specific languages [2], and program construction [10].

Design and implementation of static analyses for higher-order languages
Static program analysis automatically infers properties about a program without executing it. Static analysis is a critical component of modern compilers, integrated development environments (IDEs) and program verification tools. A sufficiently expressive static analysis can ensure the absence of particular classes of bugs, such as memory inefficiencies, data races, and information flow leaks. Moreover, static analyses enable compile-time program optimizations, e.g., constant propagation, function inlining and efficient register allocation.

Challenges in static analysis design
Every static analysis is evaluated basing on the following three important characteristics:
- **Precision**: the analysis accurately infers properties of non-trivial programs,
- **Soundness**: it always delivers correct results, and
- **Scalability**: the analysis procedure can be run efficiently on large codebases.

It is impossible to design a static analysis that excels in all the three directions and satisfies the needs of any possible client application. The design of a reasonable analysis is further complicated by the fact that most modern programming languages, such as Java, C#, JavaScript, Haskell, etc., are inherently higher-order: code might depend on the data it operates with, and data itself might be an instance of code that can be executed (e.g., Haskell’s first-class functions or Java objects). This circularity makes it tricky to disentangle the analysis into the traditional independent data- and control-flow components.

My contributions
In a series of research projects, I have explored approaches that trade off precision/soundness/scalability for static analysis of higher-order languages.

**Efficient pushdown analysis for higher-order programs.** Formulating a whole-program static analysis for a higher-order language as a pushdown system makes it possible to significantly reduce its state space and implement the analysis as an efficient and sound summarization procedure. Another way to optimize the analysis for greater scalability is to supply it with an abstract garbage collector, which performs on-the-fly pruning of the irrelevant data. However, until recently it has been an open problem, whether one can combine the pushdown approach and abstract garbage collection to improve the analysis’ precision and speed. This problem was addressed by my collaborators and me in a series of works on pushdown flow analysis [3,5]. Starting from a principled model of a whole-program static analysis in a form of an abstract interpreter, we derived a novel decision procedure for program state reachability, and then specialized it to work with abstract garbage collection. We empirically validated claims of improved precision and speed on a suite of benchmarks and found synergies between pushdown analysis and abstract garbage collection.

**A unified framework for static analyzers.** While implementing a proof-of-concept prototype for the above-described work, I noticed some striking similarities in the structure of program interpreters and
abstract interpretation-based static analyzers. This insight provided a way of extracting the common patterns in the design and implementation of static analyzers and led to the idea of considering abstract interpretation as a particular case of computational effects, which can be abstracted over using a concept of monads from functional programming. This idea was presented in my work on monadic abstract interpreters [14] that described a unifying implementation methodology for static analyzers, which allows one, by changing a representation monad, to recover a spectrum of analyses.

Modular, higher-order cardinality analysis. When a static analyzer is a part of a production compiler, its precision takes a back seat to scalability: large programs should be analyzed and compiled within a reasonable time. The scalability has been a guiding principle in my work on designing a novel cardinality analysis for Haskell programming language [13], which provided a practically satisfactory solution to a 25-years old problem of soundly and efficiently identifying and exploiting program parts that are used only once in lazy functional programming languages. The resulting analysis procedure was simple to implement (it added 250 lines of code to a 140,000 line compiler) and gave real improvements, making uncharacteristic Haskell programs, already aggressively optimized by dedicated Haskell hackers, run 2% faster at average, and GHC (i.e., a state-of-the-art Haskell compiler, written in Haskell) run up to 4% faster. The developed analyzer has been included into GHC since version 7.8.

Structured mechanized verification of concurrent programs

In the past decade, significant progress has been made towards design and development of efficient fine-grained (i.e., lock-free) concurrent data structures and algorithms, which take full advantage of parallel computations. Due to sophisticated interference scenarios and a large number of possible interactions between concurrent threads, manipulating with the same shared data structures and employing fine-grained synchronization primitives (e.g., compare-and-swap command), reasoning about fine-grained concurrent programs is challenging and error-prone.

Logic-based approach to concurrent program verification

Program logics are an appealing way to specify the behaviour of concurrent programs as well as to verify that a program is correct (i.e., obeys its specification). A well-designed logic formalism is:

- **Expressive**: Program specifications and their proofs can accommodate modern programming paradigms, such as higher-order code, locally-created threads, etc.
- **Structured**: A proof of a program’s correctness can be carried out mechanically, basing on the program’s syntax, via a number of syntax-oriented inference rules, provided by the logical framework.
- **Compositional**: Once a library is given a suitable specification and verified against it, its code is not required to be re-examined ever again: all reasoning about the client code that makes use of that library can be done solely using the specification.
- **Foundational**: A program logic can be efficiently embedded into a general-purpose mechanized mathematical framework. With such embedding, the fact of absence of bugs in the logic itself is supported by a mechanized formal proof. Furthermore, the framework hosting the logic can be employed to write programs and verify them, therefore conflating programming and proving.

My contributions

In an ongoing research project, I am creating a uniform logical framework for implementing and verifying concurrent programs, that features all of the above characteristics.

A uniform model for shared resources. The framework of Fine-grained Concurrent Separation Logic (FCSL), developed by my collaborators and me, incorporates a novel model, defining the whole meaning of sharing a data structure between several concurrent computations [6]. The model is built via two fundamental mathematical structures. The first one are state-transition systems (STSs), which describe the invariant preserved by the structure as well as the data structure’s evolution protocol, as it’s being concurrently modified by different threads. The second structure are partial commutative monoids (PCMs) that abstractly capture the notion of a “contribution” made by a particular single thread with respect to a data structure’s evolution. The representation via STSs and PCMs gives the framework its simplicity and compositionality: proofs about programs operating with several disjoint data structures and running several threads in parallel can be built out of simpler proofs about programs operating with a single resource and executing just one thread.
Verification of real-world concurrent programs. My representation of threads’ abstract contributions via PCMs was powerful enough to instantiate FCSL for specifying and verifying realistic data structures, therefore, making a case for the framework’s expressivity. The proposed model and logic were sufficient to give useful specifications and verify a large class of state-of-the-art concurrent libraries, such as locks, atomic snapshots, concurrent collections, graph algorithms and non-blocking universal constructions, as well as to make use of these specifications in the client code [12]. In particular, the observation about PCM-structured contributions made it possible to provide the first mechanized proof of correctness for the higher-order flat combining data structure. The approach is scalable: even though the proofs for libraries might be large, they are done just once.

A tool for compositional mechanized verification of concurrency. To turn FCSL into a full-fledged verification tool for a realistic language, we made use of \textit{dependent types as a syntactic theory of compositionality in higher-order languages}. FCSL was implemented as an embedding into \textsc{coq}, a general-purpose mechanized framework for formal proofs [11]. Besides being a proof assistant, \textsc{coq} is also an expressive dependently-typed programming language with a type system powerful enough to accommodate FCSL’s language and logic. That is, concurrent programs written in FCSL’s language are also \textsc{coq} programs, so they can make use of all \textsc{coq}’s features as a programming language (e.g., higher-order functions and inductive datatypes). Moreover, proofs about concurrent programs in FCSL can be done directly in \textsc{coq} and can be structured via \textsc{coq}’s interactive proof construction machinery. This ability is crucial to build confidence in a program logic’s applicability and efficiently use it in practice for specification and verification of real-world programs.

The insights drawn from the mechanized verification of concurrent algorithms and data structures in FCSL indicate that STSs and PCMs can be a robust basis for understanding, formalizing and verifying existing fine-grained programs. I believe that the same foundational insights will play a role in future designs and proofs of correctness of novel concurrent algorithms.

Looking ahead

My previous work has established a foundation for building better techniques and tools for program analysis and verification, as well as for tackling problems in program design and implementation that lie beyond these specific areas. I will now outline my plans for future work.

Towards families of domain-specific static analyses. Different clients of a static analysis have different needs. Standalone analysis tools can afford greater precision at the expense of speed, whereas IDEs are located at the opposite pole of the spectrum and might even tolerate the loss of soundness for the sake of faster responses to the queries. Finding an appropriate solution for each concrete case is a feasible task, but a diversity of clients hints at the opportunity for \textit{static analysis product lines}. In my work on monadic abstract interpreters, I’ve made first steps towards the design of a parametrized family of general-purpose analyzers. In my future work, I plan to take these ideas further and, in particular, enhance them with the use of efficient data structures for queries in order to bridge the gap between the abstract interpretation-inspired general constructions and domain-specific analyzers for program properties relevant for security, optimizations, and program refactorings.

Towards computer-aided design of concurrent programs. Following the idea of concurrent protocols, described in my work on concurrency verification in FCSL, I plan to explore the possibility of \textit{correct-by-construction concurrent code synthesis and optimizations} based on the rules imposed by a user-defined state-transition system, which constrains a data structure. Given that programs in FCSL are augmented by proofs of their correctness, I am going to employ proof search procedures to explore the space of program optimizations within the boundaries allowed by the specification, so that the resulting implementations will remain correct according to the initial assumptions.

Dependent type theory as a general-purpose verification tool. The dependent type theory used as a framework for FCSL implementation by means of embedding into \textsc{coq}, is a powerful tool for compositional encoding of program properties, and its applicability spreads far beyond verification of concurrency. In particular, it gives a way to test novel systems for reasoning about program properties on a full-fledged programming language (i.e., \textsc{coq} itself) by encoding these properties as types. My near-term future work is to encode the abstract interpretation obligations via dependent types in a \textit{certified parametrized framework for program analyzers}. In the longer term, I plan to investigate ways to verify relevant security and energy-consumption properties of systems code via dependent types.
References