Research Statement

Michael H. Borkowski

My research goal is to develop software verification techniques to make correct and performant software systems easier to write and to understand, as well as having stronger theoretical soundness guarantees. Today, correctness and performance are orthogonal goals for software: the most verified software is not the fastest. For instance, a verified list algorithm may use linked lists for simplicity. Verified compilers like Compcert (a C compiler) do not perform all the optimizations of other compilers. The functional programming language Haskell is not known for fast code, but high-performance coding is possible by sidestepping the safety guarantees of the type system and directly using primitives types exposed by the compiler. Haskell supports parallel programming, but its lazy semantics make it difficult to reason about the runtime behavior of a program.

As I discuss below, in the future, I envision that programmers will not have to make such a sharp tradeoff between performance and correctness. Additionally, I envision that software verification techniques can find a place in undergraduate instruction to interactively help students understand the code they write and understand mathematics proofs, as well as the connections between key ideas like induction and recursion.

Metatheory of Refinement Types Refinements constrain types with logical predicates to specify new concepts. For example, the refinement type $\mathsf{Pos} := \mathsf{Int}\{v : 0 < v\}$ describes positive integers and $\mathsf{Nat} := \mathsf{Int}\{v : 0 \leq v\}$ specifies natural numbers. Refinements on types have been successfully used in the past to define many sophisticated concepts, such as security policies and resource constraints, that can then be verified in programs developed in various programming languages like Haskell[1], Racket[2], and Rust[3].

In my PhD work, I presented[4] [1] λ_{RF}, a lambda calculus with a refinement type system that combines semantic subtyping (logical implication of refinements, as opposed to a strict syntactic definition) with refined polymorphic type variables. This was the first system to be formalized with proofs of soundness that combined both these features with refinement types. I mechanized this proof both in Coq, which has stronger soundness guarantees, and also in LiquidHaskell, which showed the feasibility of a large metatheoretic formalization as a refined Haskell program without a specialized proof assistant.

Future Work

Verification of Parallel Code There is often a large gap between work on software correctness and software performance, and thus far these have generally been treated as orthogonal goals. My

[1] `https://ucsd-progsys.github.io/liquidhaskell/`

[2] `https://blog.racket-lang.org/2017/11/adding-refinement-types.html`

[3] `https://flux-rs.github.io/flux/`

[4] `https://arxiv.org/abs/2207.05617`

collaborators and I envision that to achieve a combination of high-performance and high-assurance through formal verification, the path of least resistance is a strict, functional, programming style with linear types. A linear type system allows us to enforce that a function argument must be used exactly once, which can provide assurances of memory safety by preventing situations such as more than one thread having access to the same shared area of memory.

We believe in the combination of strict functional programming with linear types for two reasons: First, performant software necessarily implies parallel software on any modern architecture. Linear types, available in Haskell (as a language extension in GHC 9), are uniquely positioned to provide guarantees about resource usage that rule out data races, while placing only a light burden on the programmer. Second, automation in verification is most advanced for functional representations, as in systems like LiquidHaskell and Agda.

Our work on this project is ongoing. The need for case studies in verified algorithms from different domains presents the opportunity to carve out several research projects for undergraduates.

Verification to Teach Functional Programming The work above has inspired me to investigate in the future how proof assistants can help undergraduate students of mathematics and computer science learn both formal proof and difficult concepts like induction and recursion. Unfortunately, most proof assistants are difficult to use and have a learning curve. Using Coq to prove a simple fact like "if $x < y$ then $x + 1 < y + 1$" requires looking up and piecing together multiple lemmas from the standard library, the names of which are still shifting over time. LiquidHaskell includes annotations on Haskell code only, and no tactics, which lessens the learning curve but it does not provide the ergonomics of immediate, interactive feedback.

My future work in this direction will center around what kind of tool or environment within which LiquidHaskell could be situated to provide immediate, interactive feedback with minimal cognitive overhead. Also, I intend to investigate what teaching methods could be designed and what tools could be built to relate key concepts of programming and discrete mathematics. For instance, recursive programming and inductive proofs are taught to first and second-year students in entirely different courses but are closely related conceptually. But the correctness of a recursively defined algorithm is argued inductively. And a (constructive) inductive proof of proposition P can be seen as a recursively defined function for transforming a proof of $P(n)$ into a proof of $P(n+1)$.

These lines of investigation present many opportunities to involve undergraduates directly in research. A project along these lines would involve both empirical components such as user studies and engineering components such as building tools in the form of IDE extensions.

References

1. Michael H. Borkowski, Niki Vazou, and Ranjit Jhala. 2024. Mechanizing Refinement Types. To appear in *51st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2024)*, January 17-19, 2024, London, United Kingdom.