Research Statement

Towards Usable Program Verifiers

Niki Vazou
UC San Diego

Code deficiencies and bugs constitute an unavoidable part of software systems. In safety-critical systems, like aircrafts or medical equipment, even a single bug can lead to catastrophic impacts such as injuries or death. Formal verification can be used to statically track code deficiencies by proving or disproving correctness properties of a system. However, at its current state formal verification is a cumbersome process that is rarely used by mainstream developers.

The goal of my research is to build usable formal verifiers. A usable verifier *naturally integrates* the specification of correctness properties in the development process. Moreover, verification should be *automatic*, requiring no explicit proofs or complicated annotations. At the same time, the specification language should be *expressive*, allowing the user to write arbitrary correctness properties. *Error reporting and diagnosis* should provide insightful error messages and potential fixes any time verification fails. Finally, a usable verifier should be tested in *real-world programs*.

Current Work: Refinement Types for Haskell

LiquidHaskell is a verifier for Haskell programs that takes as input Haskell source code, annotated with correctness specifications in the form of refinement types and checks whether the code satisfies the specifications. LiquidHaskell is a successful project, with many users from the Haskell community, which in 2014 gave me the Microsoft Research Graduate Research Fellowship and lead to a subsequent NSF grant. We designed LiquidHaskell in such a way as to satisfy most of the aforementioned criteria of a usable verifier.

Natural Integration of correctness specifications comes by our choice of the functional programming language Haskell as a target language. Haskell's first class functions lead to modular specifications. The lack of mutations and side-effects allows a direct correspondence between source code and logic. Correctness specifications are naturally added to Haskell's expressive type system as *refinement types*, *i.e.,* types annotated with logical predicates. As an example, the type $\{v : \text{Int} \mid v \neq 0\}$ describes a value v which is an integer and the refinement specifies that this value is not zero. The specification language is simple as most programmers are familiar with both its ingredients, *i.e.,* Haskell types and logical formulas. Even though most of Haskell's features facilitate verification, lazy semantics rendered standard refinement typing unsound.

Refinement Types for Haskell [6] describes how we adjusted refinement typing to soundly verify Haskell's lazy programs. Refinement types were introduced in 1991 and since then have been successfully applied to many eager languages. When checking an expression, such type systems implicitly assume that all the free variables in the expression are bound to values. This property is trivially guaranteed by eager evaluation, but not in a lazy setting. Thus, to be sound and precise, a refinement type system for Haskell must take into account which subset of binders actually reduces to values. To track potentially diverging binders, we used refinement types to build a termination checker whose correctness is recursively checked by refinement types.

Automatic Verification comes by constraining refinements in specifications to decidable logics. Program verification checks that the source code satisfies a set of specifications. A trivial example is to *specify* that the second argument of a division operator is different than zero. In LiquidHaskell terms one would write this specification as: `div` :: `Int` $\rightarrow \{v : \text{Int} \mid v \neq 0\} \rightarrow$ `Int`. To *check* whether an expression with type $\{v : \text{Int} \mid v > 0\}$ is a safe argument to the division operator, the system checks whether $v > 0$ implies $v \neq 0$. By constraining all predicates to be drawn from decidable logics, such implications can be *automatically* checked via an SMT

solver. *Liquid Types* are a subset of refinement types that achieve automation and type inference by constraining the language of the logical predicates to quantifier-free *decidable* logics, including logical formulas, linear arithmetic and uninterpreted functions.

Expressiveness of the specifications is critically hindered by our choice to constrain the language of predicates to decidable logics. Liquid types specifications are naturally used to describe first order properties but prevent modular, higher order specifications. Consider a function that sorts lists of integers, with type `sort :: List Int → List Int`. Using LiquidHaskell we can specify that sorting positive numbers returns a list of positive numbers, but we cannot give a modular specification accounting for *all* different kinds of numbers `sort` will be invoked. We developed "Abstract" and "Bounded" refinement types to allow for modular specifications while preserving SMT decidability.

In **Abstract Refinement Types** [4] we parameterize a type over its refinements allowing modular specifications while preserving SMT-based decidable type checking. As an example, since `sort` preserves the elements of the input list, we can use abstract refinements to specify that *for every* refinement p on integers, `sort` takes a list of integers that satisfy p and returns a list of integers that satisfy the same refinement p. With this modular specification, we can prove that `sort` preserves the property that all the input numbers satisfy, for any property, ranging from being positive numbers to being numbers that are safe keys for a security protocol. We used abstract refinements to describe modular properties of recursive data structures. With such abstractions we simultaneously reasoned about challenging invariants such as *sortedness and uniqueness of lists* or preservation of *red-black invariants or heap properties* on trees. Without abstract refinements reasoning about each of these invariants would require a special purpose analysis. Crucially, abstractions over refinements preserve SMT-based decidability, simply by encoding refinement parameters as uninterpreted propositions within the ground refinement logic.

Bounded Refinement Types [2] constrain and relate abstract refinement and let us express even more interesting invariants while preserving SMT-decidability. As an example, we used bounds on refinement types to reason about *stateful computations*. We expressed the pre- and post-conditions of the computations with two abstract refinements, p and q respectively and used bounds to impose constraints upon them. For instance, when sequencing two computation we bound the first post-condition q_1 to imply the second pre-condition p_2. We implemented the above idea in a refined Haskell `IO` state monad that encodes *Floyd-Hoare logic state transformations* and used this encoding to track capabilities and resource usage. Moreover, we encoded *safe database access* using abstract refinements to encode key-value properties and bounds to express the constraints imposed by relational algebra operators, like disjointedness, union *etc.*. Bounds are internally translated to "ghost" functions, thus the increased expressiveness comes while preserving the automated and decidable SMT-based type checking that makes liquid typing effective in practice. Note that Abstract and Bounded refinement types are not Haskell specific. Even though we implemented these techniques in LiquidHaskell they can be used to enhance expressiveness in any refinement type setting.

Real World applications have been verified using LiquidHaskell. We proved critical safety and functional correctness of more that 10,000 lines of popular Haskell libraries [5] with minimal amount of annotations. We verified correctness of *array-based sorting algorithm* (`Vector-Algorithms`), preservation of *binary search tree* properties (`Data.Map`, `Data.Set`), preservation of *uniqueness invariants* (`XMonad`), *low-level memory safety* (`Bytestring`, `Text`), and even found and fixed a subtle correctness bug related to unicode handling in `Text`. In the above libraries we automatically proved *totality* and *termination* of all interface functions.

LiquidHaskell is an accessible verifier with many users from the Haskell community. One can learn LiquidHaskell from our interactive online tutorial and can use LiquidHaskell as an on-the-fly checker integrated to many editors (Emacs, Vim, Atom). We have been receiving feedback and feature-request from our users, via github, mailing list or personal emails. As a more direct way to communicate with our users, I took the opportunity to present LiquidHaskell in demos and tutorials (including HOPA'13, Haskell'13, CUFP'15).

Current Work: Types and Effects

Koka is a function oriented general-purpose programming language with javascript-like syntax. Using a sophisicated effect *inference* type checker it provides the guarantee that if a Koka program type checks, then no undesirable side effects (memory access, divergence, *etc.*) can occur. We recently [3] extended Koka to support user defined effects (randomness, ambiguity, asynchronicity, *etc.*) using an intermediate compiler transformation that makes use of monadic theory but is totally transparent to the user.

Future Directions

In the future I plan to further explore aspects of usable verifiers, by investigating the lines between expressiveness and decidability, improving error reporting and verifying real-world applications.

Expressiveness and Decidability define two main lines of research in the world of static verification. On the one hand, theorem provers (like Coq, Adga, etc.) allow the user to specify correctness properties using fully expressive (Turing Complete), but undecidable languages, requiring rigorous proofs or annotations. Research in this area aims to increase automation of verification. On the other hand, automated verification tools (like Liquid Types) constrain specifications to decidable languages sacrificing expressiveness. I plan to investigate how these two research approaches relate by examining whether a Turing Complete specification language can be mapped to decidable fragments and formally define the expressive bounds of decidable logics. My goal is to combine the advantages of both worlds, *i.e.,* build decidable and highly expressive verifiers.

Error Reporting and Diagnosis are crucial features for a usable verifier but very challenging to implement. Failure of verification should return with insightful error messages and fix suggestions to ease comprehension of the source of the error. In this way verification will optimize code development, instead of being an additional time-consuming step. The challenge in error reporting is to describe to the user the source of failure, without exposing any sophisticated technical details of the underlying verification theories and techniques. This challenge exists even in type checking of functional programming languages, *e.g.,* type errors in OCaml get obscure when they expose details of Hindler-Miller type inference. Type error and diagnosis is an emerging field of research that can be generalized to verification errors. We have already implemented TARGET [1] to explain verification errors via counterexamples when verification of LiquidHaskell fails. Another interesting approach would be to investigate how the user experience can be analyzed by learning techniques to track and explain the recurring patterns in verification errors and their respective fixes.

Applications. I envision that verification will soon be integrated in the development chain of software applications. Towards this end, the research community has to provide verified libraries that can be used to build real world applications. My plan is to apply software verification in hot applications of the 21st century, including web security and deep learning. *Web security* is critical now that online transactions, cloud storage and social networking dominate our lives. Formal verification can be used to statically guarantee that web transactions satisfy security policies. *Deep learning* is a relatively young and outstanding field that motivates the development of new programming techniques and the creation of new software libraries. Deep learning algorithms provide an interesting verification target as they rely on the (decidable) theory of real numbers and have well specified correctness properties (*e.g.,* the error function will eventually decrease). In both cases verification should be done via usable verifiers and lead to libraries attractive to professional developers.

I strongly believe that verification can and should be an integral part of software development. The increasing popularity of functional programming in industry shows that developers care about safe and secure code. At the same time, advances in verification techniques provide sophisticated tools to prove correctness properties. Yet, formal verification is still considered to be cumbersome. I aim to contribute in the development of usable verifiers and further bridge the gap between formal verification and mainstream software development.

References

[1] Eric L. Seidel, Niki Vazou, and Ranjit Jhala. Type targeted testing. In *ESOP*, 2015.

[2] Niki Vazou, Alexander Bakst, and Ranjit Jhala. Bounded refinement types. ICFP, 2015.

[3] Niki Vazou and Daan Leijen. From Monads to Effects, and Back. *PADL*, 2016.

[4] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. Abstract refinement types. ESOP, 2013.

[5] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. Liquidhaskell: Experience with Refinement Types in the Real World. *Haskell*, 2014.

[6] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for Haskell. In *ICFP*, 2014.