Thesis Proposal,
PhD Fellowship Program 2014

Niki Vazou

Haskell is a cutting edge functional programming language: it enforces clean and concise code, its expressive type system leads to documented and correct programs, its purity allows native concurrency and it provides higher performance than many mainstream languages (like Java or `C++`).

Though, formal verification of Haskell source code has been cumbersome. Haskell supports verification via dependent types, which requires major code modifications (ie., lifting expressions to types) and explicit proofs to be provided by the users.

The goal of my research project is to combine Haskell's type system with the power of SMT solvers to create a next-generation verifier for Haskell. A key feature of this verifier is *usability*: the specification language should be simple and verification should require minimum code modifications and annotations. Hopefully, since Haskell has been a prototype for mainstream programming languages, this type of verification could be integrated to standard application development chain.

Current Work

We have implemented liquidHaskell, a tool that given Haskell source code and some specifications verifies that the code satisfies the specifications.

Specifications are expressed as *refinement types*, ie., types annotated with logical predicates. As an example, the type $\{v : \text{Int} \mid v > 0\}$ describes a value v which is an integer and the refinement specifies that this value is greater than zero. The language of the predicates contains logical formulas, linear arithmetic and uninterpreted functions. Using uninterpreted functions the user defines *measures* on Haskell data types, for instance, they can define a measure *len* that captures the length of a list and use it to specify interesting list properties.

There are two major advantages on this formulation. Firstly, specification language is simple as most programmers are familiar with both its ingredients, ie., Haskell types and logical formulas. Secondly, verification is equivalent to refinement type checking, which in turn, through constraint generation, goes down to decidable implication checking that can be verified by an STM solver.

The tool: LiquidHaskell, takes as input a target Haskell source file, with the desired refinement types specified as a special form of comment annotation. After analysing the program, liquidHaskell returns as output:

- Either SAFE, indicating that all the specifications indeed verify, or UNSAFE, indicating there are refinement type errors, together with the positions in the source code where type checking fails.
- An HTML file containing the program source code annotated with inferred refinement types for all sub-expressions in the program. When a type error is reported, it is highlighted with red color and the programmer can use the inferred types to determine why their program does not typecheck: they can examine what properties liquidHaskell can deduce about various program expressions and provide stronger specification or alter the program as necessary so that it typechecks.

Benchmarks: We used liquidHaskell to verify safety and functional correctness properties of the following Haskell libraries:

- `GHC.List`, which implements many standard Prelude list operations; we verify various size related properties,
- `Data.Set.Splay`, which implements a set data type; we verify that all interface functions return well ordered trees,
- `Data.Map.Base`, which implements a functional map data type; we verify that all interface functions return binary-search trees,
- `Bytestring`, a library for manipulating byte arrays, we verify size-based low-level memory safety and high-level functional correctness properties, and
- `Text`, a library for high-performance unicode text processing; we verify various pointer safety and functional correctness properties, during which we find a subtle bug.

As a quantitative summary, these libraries sum up to 8706 lines of code and require 1750 lines of annotations. Moreover, verification time scales with respect to both the number of lines and the complexity of the specifications averaging about 1 sec per 8 lines of code.

Proposed Work

The goal of this project is to create a sound and usable verifier for real-world Haskell applications that can be embedded to an existing Haskell compiler. Many concrete subgoals exist towards this direction:

Soundness

- LiquidHaskell's soundness depends on a termination-checker and soundly describing infinite structures is not possible. Since infinite structures appear often in idiomatic Haskell programming we would like to cut this dependency without sacrificing any of the current features (ie., precision, expressiveness) of the tool.
- Defining the formal semantics of liquidHaskell is a long term goal, as it will provide to both us and the users a better understanding of the tool.

Usability

- Currently, typechecking is performed in granularity of module (Haskell file) as the type of each function depends on its call sites. We want to cut this dependency and get both modularity and speedup. Moreover, typechecking will become parallelizable and incremental, ie., local modifications to the source code will require rechecking only a small amount of code.

- Often verification requires some standard preprocessing of the source code. Two common examples are defining the size of a new data type and inlining higher order functions, both of which could be automatically performed. Thus, towards usability LiquidHaskell should perform code transformations to ease verification while keeping the semantics and minimize the annotation burden by inferring information whenever possible.

- Error reporting and diagnosis are crucial for the tool's usability. Currently, when liquidHaskell fails to prove some specification, it reports the location of the code where typechecking fails. Then the user needs to understand the source of the failure, ie., to simulate the reasoning that lead the tool to decide unsafety. Since this reasoning was already performed by liquidHaskell, it would be ideal to expose some of its steps either by providing informing error messages or by interactively helping the user identify the source of the error.

Real-world Applications

- Liquidhaskell has been used to verify code from the wild. As a next step we can use it to verify hot applications like security properties in web applications or data-race freedom in concurrent ones.

- We would like to investigate how liquidHaskell performs when verifying Haskell idiomatic code that depends on features like type classes, monads or infinite structures. For instance, we would like to verify and use invariants on the state of monadic functions.

Integration to a real compiler

- LiquidHaskell can be automatically used when compiling plain (without specifications) code to verify natively imposed specifications. For instance, it can prove the absence of partially defined functions that can lead to runtime exceptions or library imposed specifications, as that the used never asks for the head of an empty list.

- LiquidHaskell can interact with Haskell's dependent types. While being very expressive, dependent types require the user to explicitly provide proofs. An interesting research direction is to investigate whether liquidHaskell can be used to automatically infer some of the required proofs.

In a nutshell, the ultimate goal is to create a user-friendly, efficient, and sound tool so as to integrate formal verification, via liquidHaskell, into the development chain of Haskell applications.