Thesis Proposal,
PhD Fellowship Program 2014

Niki Vazou

Motivation

Haskell is a cutting edge functional programming language: it produces clean and concise code; its expressive type system force 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 modification (ie., lifting expressions to types) and explicit proofs.

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 project is to make verification *usable*: the specification language should be simple and verification should require minimum code modifications and annotations. Finally, since haskell has been a prototype for mainstream programming languages, by targeting haskell we aim to build a verification system that can hopefully be integrated to other languages.

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 refinements specifies that this value is greater than zero. The language of the predicates, contains logical formulas, linear arithmetic and uninterpreted functions. Using uninterpreted function, the user can define *measures* on haskell data types, for example, he can define a measure *len* that captures the length of a list and use it to specify interesting list properties.

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 analyzing the program, liquidHaskell returns as output:

- Either SAFE, indicating that all the specifications indeed verify, or UN-SAFE, 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 ver- ify size-based low-level memory safety and high-level func- tional correctness properties, and

- `Text`, a library for high-performance unicode text process- ing; we verify various pointer safety and functional correctness properties, during which we find a subtle bug.

These libraries sum up to 8706 lines of code and require 1750 lines of annotation. Moreover, verification time scales with respect to both the number of lines and the complexity of the specifications averaging about 1 sec for 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 soundeness depends on a termination-checker and 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 both to 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.

- Some standard requirements appear often when verifying code. For example, when defining a new data type we need to describe its size, or many times, to enable verification we need to inline higher order functions. LiquidHaskell can automatically perform these standard procedures.

- Error reporting and diagnosis are crucial in Currently, when liquidHaskell fails to prove some specification, it reports the location of the code where typechecking fails . It turns out that then the user needs to perform some reasoning to understand the source of this failure. As noted in Abductive Inference [PLDI 12], most of this reasoning is already performed by the tool. It would be ideal to expose some of these information 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 of the state in monadic functions.

Integration to a real compiler.

- LiquidHaskell can run at compile time to track common errors. For instance, it can be used to identify partially defined function, that can lead to run-time exceptions or that user code satisfies library function invariants, ie,. it never takes a 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. A nice 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.