LiquidHaskell: Experience with Refinement Types in the Real World *

Niki Vazou Eric L. Seidel Ranjit Jhala

UC San Diego

Abstract

Haskell has many delightful features. Perhaps the one most beloved by its users is its type system that allows developers to specify and verify a variety of program properties at compile time. However, many properties, typically those that depend on relationships *between* program values are impossible, or at the very least, cumbersome to encode within the existing type system. Many such properties can be verified using a combination of Refinement Types and external SMT solvers. We describe the refinement type checker LIQUIDHASKELL, which we have used to specify and verify a variety of properties of over 10,000 lines of Haskell code from various popular libraries, including `containers`, `hscolour`, `bytestring`, `text`, `vector-algorithms` and `xmonad`. First, we present a high-level overview of LIQUIDHASKELL, through a tour of its features. Second, we present a qualitative discussion of the kinds of properties that can be checked – ranging from generic application independent criteria like totality and termination, to application specific concerns like memory safety and data structure correctness invariants. Finally, we present a quantitative evaluation of the approach, with a view towards measuring the efficiency and programmer effort required for verification, and discuss the limitations of the approach.

1. Introduction

Refinement types enable specification of complex invariants by extending the base type system with *refinement predicates* drawn from decidable logics. For example,

```
type Nat = {v:Int | 0 <= v}
type Pos = {v:Int | 0 <  v}
```

are refinements of the basic type `Int` with a logical predicate that states the *values* `v` being described must be *non-negative* and *postive* respectively. We can specify *contracts* of functions by refining function types. For example, the contract for `div`

```
div :: n:Nat -> d:Pos -> {v:Nat | v <= n}
```

* This work was supported by NSF grants CNS-0964702, CNS-1223850, CCF-1218344, CCF-1018672, and a generous gift from Microsoft Research.

Haskell '14, September 6, 2014, Gothenburg, Sweden.

http://dx.doi.org/10.1145/2633357.2633366

states that `div` *requires* a non-negative dividend `n` and a positive divisor `d`, and *ensures* that the result is less than the dividend. If a program (refinement) type checks, we can be sure that `div` will never throw a divide-by-zero exception.

What are refinement types good for? While there are several papers describing the *theory* behind refinement types [4, 13, 27, 29, 36, 42, 44], even for LIQUIDHASKELL [39], there is rather less literature on how the approach can be *applied* to large, real-world codes. In particular, we try to answer the following questions:

1. What properties can be specified with refinement types?
2. What inputs are provided and what feedback is received?
3. What is the process for modularly verifying a library?
4. What are the limitations of refinement types?

In this paper, we attempt to investigate these questions, by using the refinement type checker LIQUIDHASKELL, to specify and verify a variety of properties of over 10,000 lines of Haskell code from various popular libraries, including `containers`, `hscolor`, `bytestring`, `text`, `vector-algorithms` and `xmonad`. First (§ 2), we present a high-level overview of LIQUIDHASKELL, through a tour of its features. Second, we present a qualitative discussion of the kinds of properties that can be checked – ranging from generic application independent criteria like totality (§ 3), *i.e.* that a function is defined for all inputs (of a given type), and termination, (§ 4) *i.e.* that a recursive function cannot diverge, to application specific concerns like memory safety (§ 5) and functional correctness properties (§ 6). Finally (§ 7), we present a quantitative evaluation of the approach, with a view towards measuring the efficiency and programmer's effort required for verification, and we discuss various limitations of the approach which could provide avenues for further work.

2. LIQUIDHASKELL

We will start with a short description of the LIQUIDHASKELL workflow, summarized in Figure 1, and continue with an example driven overview of how properties are specified and verified using the tool.

Source LIQUIDHASKELL can be run from the command-line[1] or within a web-browser[2]. It takes as *input*: (1) a single Haskell *source* file with code and refinement type specifications including refined datatype definitions, measures (§ 2.3), predicate and type aliases, and function signatures; (2) a set of directories containing *imported modules* (including the `Prelude`) which may themselves contain specifications for exported types and functions; and (3) a set of predicate fragments called *qualifiers*, which are used to infer refine-

[1] `https://hackage.haskell.org/package/liquidhaskell`

[2] `http://goto.ucsd.edu/liquid/haskell/demo/`