Demo Proposal: Liquid Types for Haskell

Niki Vazou Eric L. Seidel Ranjit Jhala
UC San Diego

Abstract

We present LiquidHaskell, a verifier for Haskell programs which uses Liquid Types to reduce the verification of higher-order, polymorphic, recursive programs over complex data types, into first-order Horn Clauses over integers, booleans and uninterpretated functions, which are then solved using classical predicate abstraction. In this demo proposal, we present an overview of this approach, and describe how we handle Haskell specific features such as type classes, algebraic data structures and laziness.

1. Introduction

Refinement types offer an automatic means of verifying semantic properties of programs, by decorating types with predicates from logics efficiently decidable by modern SMT solvers. For example, the refinement type `{v: Int | v > 0}` denotes the basic type `Int` refined with a logical predicate over the "value variable" `v`. This type corresponds to the set of `Int` values `v` which additionally satisfy the logical predicate, *i.e.,* the set of positive integers. The (dependent) function type `x:{v:Int| v > 0} -> {v:Int| v < x}` describes functions that take a positive argument `x` and return an integer less than `x`.

Refinement type checking reduces to sub-typing queries of the form $\Gamma \vdash \{v : \tau \mid p\} \preceq \{v : \tau \mid q\}$, where p and q are refinement predicates. These sub-typing queries reduce to logical validity queries of the form $[|\Gamma|] \wedge p \Rightarrow q$, which can be automatically discharged using SMT solvers [1].

Liquid Types [2] is a technique that reduces refinement type inference, and therefore verification of higher-order programs, to solving a system of Horn-clauses which are essentially the above implications with logical variables representing unknown (*i.e.,* to be inferred) refinements.

In this demo proposal we present LiquidHaskell, an implementation of Liquid Types that supports Haskell-specific features such as type class constraints, data types and laziness.

2. Overview

We present a high-level overview of the modifications we made to Liquid Types to address Haskell-specific features.

2.1 Type Classes

Parametric Invariants via Type Polymorphism. Suppose we had a generic comparison `(<=):: a -> a -> Bool`. We could use it to write:

```
max      :: a -> a -> a
max x y = if x <= y then y else x
```

In essence, the type given for `max` states that *for any* `a`, two `a` values are passed into `max`, then the returned result is also an `a` value. For example, if two *prime* numbers are passed in the result is prime, and if two *odd* numbers are passed in the result is odd. Thus, we can use refinement types [2] to verify

```
type Odd = {v:Int | v % 2 = 1}

maxOdd :: Odd
maxOdd = max 3 5
```

As `3, 5 :: Odd`, the system has to verify that `max 3 5 :: Odd`. To this end, the type parameter of `max` is instantiated with the *refined* type `Odd`, yielding the instance:

```
max :: Odd -> Odd -> Odd
```

The refinement type instantiations can be inferred, using the abstract interpretation framework of Liquid Types [2]. Thus, parametric polymorphism offers an easy means of encoding second-order invariants, *i.e.,* of quantifying over or parametrizing the invariants of inputs and outputs of functions.

Parametric Invariants and Type Classes. In Haskell the functions above are typed

```
(<=)     :: (Ord a) => a -> a -> Bool
max      :: (Ord a) => a -> a -> a
```

We might be tempted to ignore the typeclass constraint, and treat `max` as `a -> a -> a`. This would be quite unsound, as typeclass predicates preclude universal quantification over refinement types. Consider the plus function `(+):: (Num a)=> a -> a -> a`. The `Num` class constraint implies that numeric operations may occur in the function, so if we pass `(+)` two odd numbers, we will *not* get back an odd number.

To soundly verify class constrained types we use *abstract refinements* [5], which let us quantify or parameterize a type over its constituent refinements. With abstract refinements, we can type `max` as

```
max:: forall <p::a->Prop>. (Ord a)=>
            a<p> -> a<p> -> a<p>
```

where `a<p>` is an abbreviation for the refinement type `{v:a | p(v)}`. Intuitively, an abstract refinement `p` is encoded in the refinement logic as an *uninterpreted function symbol*. Thus, it is trivial to verify, with an SMT solver, that `max` enjoys the above type: the input types ensure that both `p(x)` and `p(y)` hold and hence the returned value in either branch satisfies the refinement `{v:a | p(v)}`, thereby ensuring the output type.

Consequently, we can recover the verification of `maxOdd`. Now, instead of instantiating a *type* parameter, we first instantiate the type parameter with an unrefined `Int` type and then instantiate the *refinement* parameter of `max` with the concrete refinement `{\v -> v % 2 = 1}`, after which type checking proceeds as usual [2]. Thus, abstract refinements allow us to quantify over invariants without relying on parametric polymorphism, even in the presence of type classes.

2.2 Expressive Data Types

Next, we illustrate how abstract refinements allow us to specify and verify expressive data types. As an example we encode vectors as maps from `Int` to a generic range `a`. We specify vectors as

```
    data Vec a < dom :: Int -> Prop
               , rng :: Int -> a -> Prop>
      = V (i:Int<dom> -> a <rng i>)
```

Here we are parameterizing the definition of the type `Vec` with two abstract refinements, `dom` and `rng`, which respectively describe the *domain* and *range* of the vector. That is, `dom` describes the set of valid indices and `rng` specifies an invariant relating each `Int` index with the value stored at that index.

Describing Vectors. With this encoding we can describe various vectors. To start with, we can have vectors of `Int` defined on positive integers with values equal to their index:

```
  Vec <{\v -> v > 0}, {\_ v -> v = x}> Int
```

As a more interesting example, we can define a *Null Terminating String* with length `n`, as a vector of `Char` defined on a range `[0, n)` with its last element equal to the terminating character:

```
  Vec <{\v -> 0 <= v < n}
      ,{\i v -> i = n-1 => v = '\0'}> Char
```

Finally, we can encode a *Fibonacci memoization vector*, which can be used to efficiently compute a Fibonacci number, that is defined on positive integers and its value on index `i` is either zero or the `i`th Fibonacci number:

```
  Vec <{\v -> 0 <= v}
      ,{\i v -> v != 0 => v = fib(i)}> Int
```

Using Vectors. A first step towards using vectors is to supply the appropriate types for vector operations, (*e.g.,* `set`, `get` and `empty`). This usually means qualifying over the domain and the range of the vectors. Then the programmer has to specify interesting vector properties, as we did for the Fibonacci memoization and null terminating string vectors. Finally, the system can verify that user functions that transform vectors preserve these properties [5].

In LIQUIDHASKELL the pair data construct is parametrized over an abstract refinement that describes the second component with respect to the first, thereby encoding dependent pairs. Similarly, the list data type is parametrized over an abstract refinement that relates the head of the list with all the elements of the tail. This way the user can reason about recursive list properties like sortedness.

2.3 Laziness

Finally, we present how we modify Liquid Types to support verification under a lazy setting. Consider the following code:

```
        let incr n = n : incr (n+1)  in
        let xs     = incr 0          in
        map (\x -> assert x < 0) xs
```

In an *eager* setting this code is *safe*. To execute the code, all the let bindings should be fully evaluated. Thus, `incr 0` should be fully evaluated, but since it describes an infinite list execution diverges. The code does not terminate, so it is *partially* safe.

In a *lazy* setting however, the let bindings will not be evaluated until their values are needed. The assertion will therefore fail on the head of the list, making the example unsafe.

This example illustrates that on lazy evaluation, treatment of *non-termination* expressions should be changed. In eager settings diverging values are uninhabited, thus their type should be refined with `false` or `xs :: {v:[a] | false}`. Under lazy evaluation though, diverging values might not affect computation. In other words we cannot prove anything for these values, so their type should be trivially refined, *i.e.,* their refinement should be `true` or `xs :: {v:[a] | true}`.

But how do we determine termination? Trivially, a diverging value is the result of a recursive function. LIQUIDHASKELL runs a termination analysis, based on *Size-change termination* rule[3] for recursive functions. If termination can be proved for every recursive function then no diverging sub-term can be produced and verification proceeds as in an eager setting.

If LIQUIDHASKELL cannot prove termination for some function then an error is created. In this case, the user should disable the termination check for the function and provide a valid type for it, *i.e.,* a type with a trivially refined result.

In the code above, termination obviously cannot be proved for `incr` function. Thus the user has to disable the termination check (using a **`strict`** token) and provide a valid type:

```
      strict incr :: (Num a) => a -> [a]
```

Note that the type `a` inside the list can be refined, thus `[{v:a | v >=0 }]` is also a valid result type. In any case, there is no type that satisfies both `incr`'s behaviour, *i.e.,* that the elements of the list are non-negative, and the assertion, *i.e.,* that the elements are negative, thus the program will be unsafe.

2.4 Real-world Usage

We have used LIQUIDHASKELL to verify properties of commonly used Haskell libraries, including `text`, `bytestring`, `containers` and `XMonad`. For `text` and `bytestring` we focused on the functional correctness of the exposed API and the safety of memory accesses, whereas for `XMonad` we verified some provided `QuickCheck` properties, as in [4]. While verifying `text` we discovered a bug that results in an out-of-bounds write, which we have reported upstream. The bug is a subtle one due to the mixed 2- and 4-byte encoding of UTF-16 characters used by `text`.

2.5 Plan

- *Basic Refinement and Liquid Types* A rapid overview of refinement types and liquid type inference.
- *Abstract Refinement Types* Demonstration of Abstract Refinement Types and their usage on type classes and data types.
- *Laziness* Presentation of how the tool is used to perform termination analysis on recursive functions and give sound output on programs with diverging sub-expressions.
- *Analysing Libraries* Experience report on using LIQUIDHASKELL to verify properties of popular Haskell libraries.

Finally, the reader can find the source, an online demo, and a series of blog articles describing LIQUIDHASKELL at our website[1].

References

[1] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[2] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

[3] Damien Sereni and Neil D. Jones. Termination analysis of higher-order functional programs. APLAS, 2005.

[4] Wouter Swierstra. xmonad in coq (experience report): programming a window manager in a proof assistant. In *Haskell Symposium*, 2012.

[5] N. Vazou, P. Rondon, and R. Jhala. Abstract refinements. In *ESOP*, 2013.

[1] `http://goto.ucsd.edu/~rjhala/liquid/haskell/demo`