LiquidHaskell: Refinement Types for Haskell

Jakub Zárybnický <xzaryb00@stud.fit.vutbr.cz>

1 Introduction

LiquidHaskell [1] is an actively-developed SMT-based type-checker that adds *Logically Qualified Data Types* (abbreviated to *Liquid Types*), a variant of refinement types, to the Haskell type system. It allows the programmer to add *refinements* to values and data types, while retaining the type inference part of the Hindley-Milner type system and thus not requiring annotations across the entire program. Its main limitation is in the kind of properties that it can prove—only decidable formulas solvable by SMT solvers can be used.

A part of its initial development has been a successful application to many essential Haskell libraries, including `base`, `containers`, `vector`, and `bytestring`, proving data type invariants, function totality and termination, or safe access to indexed structures.

An example of what its syntax looks like can be seen below, where the prototypical *partial function* `head`, which is the scarecrow of introductory Haskell classes—calling it with an empty list will crash the program with a rather unhelpful error message, a direct contradiction to the type-safety guarantees brought up by proponents of Haskell.

The Liquid type signature below replaces the plain Haskell type signature, and asks LiquidHaskell to ensure that the input list is not empty—without changing the type signature for e.g. external users of a library who may not be using LiquidHaskell at all.

```
{-@ head :: {v:[a] | 0 < length v} -> a @-}
head (x:_)  = x

{-@ take :: i:Nat -> {xs:[a] | length xs >= i}
                  -> {ys:[a] | length ys = length xs - i} @-}
take 0 [] = []
```

[1] `https://ucsd-progsys.github.io/liquidhaskell-blog/`

```
take i (x:xs) = if i == 0 then N else (x:take (i-1) xs)
take i [] = impossible "Out of bounds indexing!"
```

1.1 Installation

Until recently, LiquidHaskell has been a separate command-line tool apart from the Haskell compiler, complicating the workflow especially for larger-scale projects despite having several editor-specific plugins, and did not really achieve wide-spread use.

In the summer of 2020, however, there has been initiative to better integrate the tool with the GHC Haskell compiler via a rather recent addition to GHC which makes it extensible via *compiler plugins*. This makes adding LiquidHaskell to a project as easy as adding two lines to the Cabal package specification file, and also removing the need for tool-specific editor support, which may encourage its future use. Given that this capability has been just made available in the latest release, it is also the way in which I tried out the tool.

To use LiquidHaskell, you will need the Haskell compiler GHC, and Z3 (or any other SMTLIB-compatible SMT solver) on your system. Stack [2] is currently perhaps the easiest way of obtaining GHC, and Z3 should be available via the usual software installation route of your system. After that, using LiquidHaskell should be as easy as adding `liquidhaskell` as a dependency, and specifying `{-# OPTIONS -fplugin=LiquidHaskell #-}` in any file to be checked by LiquidHaskell.

2 Theory

LiquidHaskell brings refinement types to Haskell. The combination of the Hindley-Milner system and refinement types has previously been successfully studied in OCaml [3] where it has greatly reduced the amount of manual program annotation required for program verification. While there are some complications that arise from translating the technique from a call-by-value language like OCaml to a call-by-need language like Haskell—that LiquidHaskell has solved by tying type verification to termination checking—the general technique still applies.

LiquidHaskell works by letting GHC translate the program into Core [4],

[2] `https://docs.haskellstack.org/en/stable/README/`

[3] `http://goto.ucsd.edu/~rjhala/liquid/liquid_types.pdf`

[4] `https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/core-syn-type`

a small, explicitly-typed version of System F. It then takes this output and the result of parsing the type refinement annotations, and translates them into Horn clause constraints, *verification conditions*, and lets an SMT solver solve them.

Refinement type systems are designed so that the refinement logic is efficiently decidable, and unlike full dependent types do not allow arbitrary functions in the refinement logic. LiquidHaskell constrains the annotation language to the *decidable logic of equality, uninterpreted functions and linear arithmetic*, QF-EUFLIA. This constrains the language of annotations, especially in the use of functions.

There are several types of annotations available: the most common one are function signature annotations which have already been mentioned above, that introduce pre- and post-conditions of the function. The next one is `data` that creates refined data types, adding refinements to fields and constructors of Haskell data types.

```
{-@
data Map k a <l :: k -> k -> Prop, r :: k -> k -> Prop>
  = Tip
  | Bin (sz    :: Size)
        (key   :: k)
        (value :: a)
        (left  :: Map <l, r> (k <l key>) a)
        (right :: Map <l, r> (k <r key>) a)
@-}
data Map k a = Tip
             | Bin Size k a (Map k a) (Map k a)
```

The next one is `measure`, and the related annotations `inline` and `reflect`, that copy the function definition to the refinement logic. While this may seem to allow arbitrary functions, they are quite restricted—`inline` cannot be recursive. `measure`, named as the *measure* of a data type, may only recurse on the constructors of the data type.

```
data Heap a = Empty | Node { left :: Heap a, el :: a, right :: Heap a }

{-@ measure heapSize @-}
heapSize :: Heap a -> Int
heapSize Empty = 0
heapSize (Node l _ r) = 1 + heapSize l + heapSize r
```

The last two (non-deprecated) annotations are `type` and `assume`, where `type` introduces a type alias that may contain pre- or post-conditions, and `assume` adds an unchecked fact into the logic. An example of a type alias can be seen below.

```
{-@ type HeapN a N = {h:Heap a | heapSize h = N} @-}

{-@ exampleHeap :: HeapN Int 2 @-}
exampleHeap = Node (Node Empty 1 Empty) 0 Empty
```

If, for example, the explicitly named size of the heap was specified badly, we would get an error message like below. This is the common format of all LiquidHaskell error messages—as is sometimes said about Haskell, reading error messages is a skill by itself, as the cause of the error is not always immediately obvious.

```
Liquid Type Mismatch
  .
  The inferred type
    VV : (Lib.Heap [GHC.Types.Int])
  .
  is not a subtype of the required type
    VV : {VV : (Lib.Heap [GHC.Types.Int]) | Lib.heapSize VV == 2}
  .
```

3 Showcase

LiquidHaskell comes with a large array of showcases, tutorials, and exercises—in this work I will walk through one of these, an implementation of the functional data structure *Lazy Queues* [5], that is available as a set of exercises without a publicly available solution [6]. A lazy queue is a structure based on two linked lists whose amortized performance of $O(1)$ of the *insert* and *remove* operations depends on an invariant being enforced between the two lists, namely that the first list need to be longer or equal to the second one—an ideal use-case a program verification tool. We will start with two data structures, a sized linked list, and the queue:

[5] http://www.westpoint.edu/eecs/SiteAssets/SitePages/Faculty%20Publication%20Documents/Okasaki/jfp95queue.pdf

[6] http://ucsd-progsys.github.io/liquidhaskell-tutorial/Tutorial_09_Case_Study_Lazy_Queues.html

```
data SList a = SL { size :: Int, elems :: [a] }

data Queue a = Q { front :: SList a, back  :: SList a }
```

To bind the value of the `size` field to the length of the list `elems`, we will need a few more annotations, one to declare the dependency, and one to declare the type of the field accessor function. We will also need the *measure* function, to give the SMT solver a way to get from the list to its size. We can also declare two type aliases that will make our life easier, one for a list of a specific size, one for a non-empty list:

```
{-@ measure listSize @-}
listSize :: [a] -> Int
listSize []     = 0
listSize (_:xs) = 1 + listSize xs

{-@ data SList a = SL
  { size  :: Nat
  , elems :: {v:[a] | listSize v = size}
  } @-}
{-@ size :: q:SList a -> {v:Nat | v = size q} @-}

{-@ type SListN a N = {v:SList a | size v = N} @-}
{-@ type NESList a  = {v:SList a | size v > 0} @-}
```

Doing the same for the queue will yield us the foundation, only this time we don't have a size field, but we need to declare the invariant—tell the solver that `back` needs to always be smaller or equal to `front`.

```
{-@ measure queueSize @-}
queueSize :: Queue a -> Int
queueSize (Q f b) = size f + size b

{-@ type SListLE a N = {v:SList a | size v <= N} @-}
{-@ data Queue a = Q {
  { front :: SList a
  , back  :: SListLE a (size front)
  } @-}

{-@ type QueueN a N = {v:Queue a | queueSize v = N} @-}
{-@ type NEQueue a = {v:Queue a | queueSize v > 0} @-}
```

If we now declare a badly-sized list like `list = SL 1 []` we will get a rather interesting message. We see that LiquidHaskell has inferred several properties of the empty list passed to the constructor—properties of its length and its identity—and that those fail to match with the invariants on the constructor.

```
Liquid Type Mismatch
  .
  The inferred type
    VV : {v : [a] | Lib.realSize v == 0
                  && len v == 0
                  && len v >= 0
                  && v == ?c}
  .
  is not a subtype of the required type
    VV : {VV : [a] | Lib.realSize VV == 1}
  .
  in the context
    ?c : {?c : [a] | Lib.realSize ?c == 0
                   && len ?c == 0
                   && len ?c >= 0}
```

Before starting to work on the queue itself, we need a few functions to manipulate the sized list, equivalents to the library functions `head` and `tail`, perhaps only there as an exercise. We can see quite well how the size of the list varies in each operation, their pre- and post-conditions.

```
{-@ nil :: SListN a 0 @-}
nil = SL 0 []

{-@ cons :: a -> xs:SList a -> SListN a {size xs + 1} @-}
cons x (SL n xs) = SL (n+1) (x:xs)

{-@ tl :: xs:NESList a -> SListN a {size xs - 1}  @-}
tl (SL n (_:xs)) = SL (n-1) xs
tl _ = error "empty SList"

{-@ hd :: xs:NESList a -> a @-}
hd (SL _ (x:_)) = x
hd _ = error "empty SList"
```

While the exercise is spread over several sections, I will condense the last definitions into a single block. We need to define the operations over the queue, `insert` and `remove`. In the simple case, we simply need to *cons* or *uncons* an element on the two lists, but if the invariant would be broken, we need a way to restore it—and that would be `makeq` and `rot`, where `rot` rotates the `back` list and appends it to the `front` one. Using these two, implementing `insert` and `remove` is trivial:

```
{-@ makeq :: f:SList a -> b:SListLE a {size f + 1}
          -> QueueN a {size f + size b} @-}
makeq f b
  | size b <= size f = Q f b
  | otherwise        = Q (rot f b nil) nil

{-@ rot :: l:SList a -> r:SListN a {size l + 1} -> a:SList a
         -> SListN a {size l + size r + size a} @-}
rot f b acc
  | size f == 0 = hd b `cons` acc
  | otherwise   = hd f `cons` rot (tl f) (tl b) (hd b `cons` acc)

{-@ insert :: a -> q: Queue a -> QueueN a {queueSize q + 1} @-}
insert e (Q f b) = makeq f (e `cons` b)
{-@ remove :: q:NEQueue a -> (a, QueueN a {queueSize q - 1})  @-}
remove (Q f b) = (hd f, makeq (tl f) b)
```

The exercise ends with a final test, to implement `replicate` and verify that it works:

```
{-@ replicate :: n:Nat -> a -> QueueN a n @-}
replicate 0 _ = emp
replicate n x = insert x (replicate (n-1) x)

{-@ okReplicate :: QueueN _ 3 @-}
okReplicate = replicate 3 "Yeah!"  -- accept

{-@ badReplicate :: QueueN _ 3 @-}
badReplicate = replicate 1 "No!"   -- reject
```

The last line is rejected with an error very similar to the above one, informing us of the type mismatch between the value and the type we claim it has, which also means that the refinement types of the above expressions are well-typed and work well.

4 Comparison with Dependent Haskell

Bringing fully dependent types to Haskell is a long-term aspiration of several interest groups [7], which may replace tools like LiquidHaskell and move Haskell closer in power to languages like Agda or Idris.

However, it is not unrealistic to get somewhat close to the power of refinement or dependent types by using several GHC extensions to Haskell, using `DataKinds` to promote values to the type-level, and `TypeFamilies` to achieve type-level functions. Below you can see a snipped of code which has been taken from a previous Haskell proof-of-concept of mine, an implementation of a type-safe and verified *Braun heap* data structure. You can see that it is necessary to manually prove certain properties, adding the data type `Offset` and prove type equalities using `Data.Type.Equality`, due to the lack of support for arithmetic—the last two lines simply prove that if $y + z = w$ and $x = 1 + y + z$, then $x = 1 + w$ to the compiler.

```
data Heap (n :: Nat) a where
  Empty :: Heap 0 a
  Node :: Offset m n -> Heap m a -> a -> Heap n a -> Heap (1 + m + n) a
data Offset m n where
  Even :: Offset n n
  Leaning :: Offset (1 + n) n

merge :: Ord a => Offset m n -> Heap m a -> Heap n a -> Heap (m + n) a
merge Even = mergeEven
merge Leaning = mergeLeaning

mergeEven :: Ord a => Heap n a -> Heap n a -> Heap (n + n) a
mergeEven l@(Node lo ll lx lr) r@(Node _ _ ly _)
  | lx <= ly = Node Leaning r lx (merge lo ll lr)
  | otherwise = let (x, l') = extract l in Node Leaning (replaceMin x r) ly l'
mergeEven _ _ = Empty

mergeLeaning :: Ord a => Heap (1 + n) a -> Heap n a -> Heap (1 + n + n) a
mergeLeaning h Empty = h
mergeLeaning Empty h = h
mergeLeaning l@(Node lo ll lx lr) r@(Node _ rl ly rr)
  | lx <= ly = Node Even r lx (merge lo ll lr)
  | otherwise = case proof r rl rr Refl of
      Refl -> let (x, l') = extract l in Node Even (replaceMin x r) ly l'
  where
    proof :: ((y + z) ~ w) => p x a -> p y a -> p z a -> x :~: (1 + y + z) -> x :~: (1 + w)
    proof _ _ _ Refl = Refl
```

[7] https://github.com/goldfirere/thesis/

Dependent types in Haskell, however, are most closely approximated by the `singletons` library, which allows using both values as types, and types as values - the following is taken from the `GPLVMHaskell` library, a function that takes an opaque matrix data type and promotes its dimensions to the type-level as `Nat`, and makes them available to a receiver function, perhaps allowing it to prove it is working with a square matrix.

```
withMat
  :: Matrix D Double
  -> (forall (x :: Nat) (y :: Nat). (SingI y, SingI x) => DimMatrix D x y Double -> k)
  -> k
withMat m f =
  let (Z  :.  y  :.  x) = extent m
  in
  case toSing (intToNat y) of
    SomeSing (sy :: Sing m) -> withSingI sy $
      case toSing (intToNat x) of
        SomeSing (sx :: Sing n) -> withSingI sx $ f (DimMatrix @D @m @n m)
```

5 Conclusion

While I personally would not reach for LiquidHaskell in my future projects due to several remaining ergonomy issues—lack of support for the documentation generation tool Haddock, or not being able to use GHCJS, the Haskell-to-JavaScript compiler—it seems to be an advanced tool that helps push the Haskell guarantees of type-safety and correctness-by-construction even further, that has already proved itself in a number of security-critical applications, unlike many other program verification tools that primarily serve as a research vehicle.