Liquid Haskell:
Refined, reflective, and classy

Ryan Scott

PL Wonks

March 23, 2018


LiquidHaskell

Refinements

```
divide :: Int
       -> {v:Int | v /= 0}
       -> Int
divide n d = n `div` d
```

Refinement reflection

```
{-@ reflect fib @-}
fib :: Int -> Int
fib i | i == 0     = 0
      | i == 1     = 1
      | otherwise = fib (i-1) + fib (i-2)

fibOne :: {fib 1 == 1}
fibOne = trivial *** QED
```

Refinement reflection + type classes?

Refinement reflection + type classes?

```haskell
class Semigroup a where
  (<>) :: a -> a -> a
```

Refinement reflection + type classes?

```
class Semigroup a where
  (<>) :: a -> a -> a

class Semigroup a => VerifiedSemigroup a where
  appendAssoc
    :: x:a -> y:a -> z:a
    -> { x <> (y <> z) == (x <> y) <> z }
```

Ref ment reflectio ype classes?

```
class Semigroup
  (<>) :: a -> a

class Semigroup a =>          Semigroup a where
  appendAssoc
    :: x:a -> y:
    -> { x <>           ) ==  (          <> z }
```

Ref ment reflectio ype classes?

```
class Semigroup
  (<>) :: a -> a

class Semigroup a =>          Semigroup a where
  appendAssoc
    :: x:a -> y:
    -> { x <>                 ) ==          <> z }
```

Why not?

Why not? Desugaring

```haskell
class Semigroup a where
  (<>) :: a -> a -> a
```

Why not? Desugaring

```
class Semigroup a where          -- Surface syntax
  (<>) :: a -> a -> a
```

Why not? Desugaring

```haskell
class Semigroup a where          -- Surface syntax
  (<>) :: a -> a -> a
```

```haskell
data Semigroup a {               -- GHC core syntax
  (<>) :: a -> a -> a
}
```

Desugaring instances

```
instance Semigroup Unit where
  Unit <> Unit = Unit
```

Desugaring instances

```
instance Semigroup Unit where
  Unit <> Unit = Unit
```

```
semigroupUnit :: Semigroup Unit
semigroupUnit = Semigroup {
  (<>) = appendUnit
}

appendUnit :: Unit -> Unit -> Unit
appendUnit Unit Unit = Unit
```

Desugaring functions

```haskell
smashList :: Semigroup a => a -> [a] -> a
smashList x []     = x
smashList x (y:ys) = smashList (x <> y) ys
```

Desugaring functions

```haskell
smashList :: Semigroup a => a -> [a] -> a
smashList x []     = x
smashList x (y:ys) = smashList (x <> y) ys
```

```haskell
smashList :: Semigroup a -> a -> [a] -> a
smashList _ x [] = x
smashList dSemigroup x (y:ys)
  = smashList dSemigroup
              ((<>) dSemigroup x y) ys
```

Key insight

Any refined type involving type classes must be able to survive the translation to GHC core.

First (naïve) attempt

```
class Semigroup a => VerifiedSemigroup a where
  appendAssoc
    :: x:a -> y:a -> z:a
    -> { x <> (y <> z) == (x <> y) <> z }
```

First (naïve) attempt

```
class Semigroup a => VerifiedSemigroup a where
  appendAssoc
    :: x:a -> y:a -> z:a
    -> { x <> (y <> z) == (x <> y) <> z }
```

```
data VerifiedSemigroup a {
  semigroupSuperClass :: Semigroup a
, appendAssoc ::
      x:a -> y:a -> z:a
   -> {     (<>) d x ((<>) d y z)
         == (<>) d ((<>) d x y) z
      }
}
```

First (naïve) attempt

```
class Semigroup a => VerifiedSemigroup a where
  appendAssoc
    :: x:a -> y:a -> z:a
    -> { x <> (y <> z) == (x <> y) <> z }
```

```
data VerifiedSemigroup a {
  semigroupSuperClass :: Semigroup a
, appendAssoc ::
      x:a -> y:a -> z:a
   -> {     (<>) d x ((<>) d y z)
         == (<>) d ((<>) d x y) z
      }
}
```

First (naïve) attempt

```
class Semigroup a => VerifiedSemigroup a where
  appendAssoc
    :: x:a -> y:a -> z:a
    -> { x <> (y <> z) == (x <> y) <> z }
```

```
data VerifiedSemigroup a {
  semigroupSuperClass :: Semigroup a
, appendAssoc ::
      x:a -> y:a -> z:a ->
   -> { forall d:VerifiedSemigroup a.
           (<>) d x ((<>) d y z)
         == (<>) d ((<>) d x y) z
      }
}
```

First (naïve) attempt

```
class Semigroup a => VerifiedSemigroup a where
  appendAs
    :: x:a                -> z:a
    -> { x                > z) == (x                z }
```

```
data VerifiedSemi
  semigroupSuperCla                 group a
, appendAssoc ::
      x:a -> y:a -
   -> { forall d                    up a.
          (<>                 (<>) d
        == (                  (<>) d x y
      }
}
```

We can't shove `foralls` within predicates willy-nilly.

Liquid Haskell is based on the quantifier-free logic of linear arithmetic and uninterpreted functions (QF-ULIA).

```
{ forall d:VerifiedSemigroup a. ... }
```

Can't be expressed in this system.

Observation

We can dictate the behavior of type classes in Liquid Haskell by their *instances*.

Better attempt

```
class Semigroup a => VerifiedSemigroup a where
  appendAssoc
    :: x:a -> y:a -> z:a
    -> { x <> (y <> z) == (x <> y) <> z }
```

Better attempt

```
class Semigroup a => VerifiedSemigroup a where
  appendAssoc
    :: x:a -> y:a -> z:a
    -> { x <> (y <> z) == (x <> y) <> z }
```

```
instance VerifiedSemigroup Unit where ...
```

Better attempt

```
class Semigroup a => VerifiedSemigroup a where
  appendAssoc
    :: x:a -> y:a -> z:a
    -> { x <> (y <> z) == (x <> y) <> z }
```

```
$dVSUnit :: VerifiedSemigroup Unit
$dVSUnit = VerifiedSemigroup { ... }

(appendAssoc $dVSUnit) ::
    x:a -> y:a -> z:a
 -> {     (<>) $dVSUnit x ((<>) $dVSUnit y z)
      == (<>) $dVSUnit ((<>) $dVSUnit x y) z
      && ...
    }
```

Better attempt

```
class Semigroup a => VerifiedSemigroup a where
  appendAssoc
    :: x:a -> y:a -> z:a
    -> { x <> (y <> z) == (x <> y) <> z }
```

```
instance VerifiedSemigroup Int where ...
```

Better attempt

```
class Semigroup a => VerifiedSemigroup a where
  appendAssoc
    :: x:a -> y:a -> z:a
    -> { x <> (y <> z) == (x <> y) <> z }
```

```
$dVSInt :: VerifiedSemigroup Int
$dVSInt = VerifiedSemigroup { ... }

(appendAssoc $dVSInt) ::
    x:a -> y:a -> z:a
 -> {     (<>) $dVSInt x ((<>) $dVSInt y z)
      == (<>) $dVSInt ((<>) $dVSInt x y) z
      && ...
    }
```

Too long; didn't watch

We begin to extend Liquid Haskell towards supporting refinement reflection + type classes:

- Accommodate typing rules to be instance-aware (not as simple as it looks!)
- Desugar refinements involving type classes into refinements involving dictionaries