Refinement Types in Haskell

Lecture 2: Refinement logic, datatypes and subtyping

Brandon Hewer

Today's Lecture

In today's lecture, we will begin with a brief account of the syntax and semantics of Liquid Haskell's refinement logic.

We will then discuss SMT decidability and give a high-level overview of Liquid Haskell's verification procedure.

Finally, we will switch to our main focus which is to introduce how a variety of constructs in Haskell such as record types and inductive algebraic datatypes can be refined and used in Liquid Haskell.

Lecture Outline

1. Today's Lecture
2. **Lecture Outline**
3. Syntax of the refinement logic
4. Semantics of the refinement logic
5. Verification and SMT decidability
6. Subtyping
7. Refinement types in Liquid Haskell
8. Guards
9. Higher-order functions

Syntax of the refinement logic

The refinement logic of Liquid Haskell is based upon a first-order logic with *uninterpreted function symbols*.

Constants	**Examples**
Booleans	True, False
Integers	−27, 0, 4, 107
Doubles	2.46, −9.1, 3.289
Strings	"", "as", "astring"
Lists	[] , [1,3] , [["a"], []]

Syntax of the refinement logic

The refinement logic of Liquid Haskell is based upon a first-order logic with *uninterpreted function symbols*.

Operation	**Syntax**
Addition	x + y
Subtraction	x − y
Multiplication	x * y
Division	x / y
Modulus	x mod y

Syntax of the refinement logic

The refinement logic of Liquid Haskell is based upon a first-order logic with *uninterpreted function symbols*.

Relation	**Syntax**
Equality	x == y **or** x = y
Not equal	x /= y
Orderings	x < y, x > y, x <= y, x >= y

Syntax of the refinement logic

The refinement logic of Liquid Haskell is based upon a first-order logic with *uninterpreted function symbols*.

Logical operator	**Syntax**
And	p && q
Or	p || q
Not	not p
Implication	p => q **or** p ==> q,
If and only if	p <=> q

Syntax of the refinement logic

The refinement logic of Liquid Haskell is based upon a first-order logic with *uninterpreted function symbols*.

Expression	**Syntax**
Variables	`x`
Constants	`c`
Relations/Logical	`e1 r e2`
Application	`f e1 e2 ... en`
If-then-else	**`if`** `p` **`then`** `e1` **`else`** `e2`

Syntax of the refinement logic

Predicates in Liquid Haskell are merely boolean-valued expressions.

Examples of predicates include:

$$m * n <= m + n$$

$$xs == [] <=> x \ \%2 = 0$$

$$\textbf{if } \text{len } ys > 0 \textbf{ then } ys = xs \textbf{ else } ys = zs$$

$$False => not \ p \ \&\& \ p$$

Semantics of the refinement logic

The syntax of Liquid Haskell's refinement logic supports the definition of *decidable predicates* that depend on a context of typed bindings and *guards*.

A *guard* is a predicate that is assumed to be true when it appears in a given environment.

Semantics of the refinement logic

A Liquid Haskell predicate in an environment can simply be interpreted as a boolean valued Haskell function.

Typed bindings in an environment correspond to arguments of this function, while guards are translated to implications.

For example, the predicate `P = (p && q => r)` in the environment `p :: Bool, q :: Bool, r :: Bool` can be interpreted as follows:

```
P :: Bool → Bool → Bool → Bool
P True True False = False
P _ _ _           = True
```

Semantics of the refinement logic

Semantics of guarded predicates

A context Γ consists of typed bindings of the form $x : \tau$ and guards P where P is a predicate in the refinement logic.

If we write $\Gamma; P$ to denote the extension of the context Γ by the predicate P, then we have the following interpretation:

$$[\![\Gamma; P \models Q]\!] := [\![\Gamma \models P \Rightarrow Q]\!]$$

Semantics of the refinement logic

Satisfiability. A predicate P is *satisfiable* if there exists arguments for which its interpretation will evaluate to True.

Validity. A predicate P is *valid* if its interpretation evaluates to True for all arguments.

Verification and SMT decidability

Liquid Haskell's refinement logic is *SMT-decidable*.

A decidable SMT (satisfiability modulo theories) theory is:

- an extension of propositional logic that can incorporate additional theories such as linear arithmetic and proof-by-logical-evaluation;
- such that there is an algorithm (SMT solver) that can correctly determine the truth value of any well-formed statement in the theory in finite time.

Verification and SMT decidability

Liquid Haskell's refinement logic is *SMT-decidable*.

A decidable SMT (satisfiability modulo theories) theory is:

- an extension of propositional logic that can incorporate additional theories such as linear arithmetic and proof-by-logical-evaluation;
- such that there is an algorithm (SMT solver) that can correctly determine the truth value of any well-formed statement in the theory in finite time.

Verification and SMT decidability

Liquid Haskell's refinement logic is *SMT-decidable*.

A decidable SMT (satisfiability modulo theories) theory is:

- an extension of propositional logic that can incorporate additional theories such as linear arithmetic and proof-by-logical-evaluation;
- such that there is an algorithm (SMT solver) that can correctly determine the truth value of any well-formed statement in the theory in finite time.

Verification and SMT decidability

To verify a program, Liquid Haskell will:

1. interpret the logical conditions given by user-specified refinements as verification conditions in an SMT logic;

2. query an external SMT solver (e.g. Z3) to determine the satisfiability of these conditions and thus decide whether your program is type safe.

Subtyping

In order to generate the verification conditions to be validated by an SMT solver, Liquid Haskell introduces a number of key *subtyping rules*.

Erasure

$$\frac{\Gamma \vdash \tau \qquad \Gamma, x : \tau \vdash P}{\Gamma \vdash \{x : \tau \mid P\} <: \tau}$$

Subtyping

In order to generate the verification conditions to be validated by an SMT solver, Liquid Haskell introduces a number of key *subtyping rules*.

Implies

$$\frac{\text{Valid } [\![\Gamma \models P \Rightarrow Q]\!]}{\Gamma \vdash \{x : \tau \mid P\} <: \{x : \tau \mid Q\}}$$

Subtyping

In order to generate the verification conditions to be validated by an SMT solver, Liquid Haskell introduces a number of key *subtyping rules*.

Functions

$$\frac{\Gamma \vdash \tau_2 <: \tau_1 \qquad \Gamma[x \mapsto \tau_2] \vdash \sigma_1 <: \sigma_2}{\Gamma \vdash (x : \tau_1 \to \sigma_1) <: (x : \tau_2 \to \sigma_2)}$$

Subtyping

In order to generate the verification conditions to be validated by an SMT solver, Liquid Haskell introduces a number of key *subtyping rules*.

Polymorphism

$$\frac{\Gamma \vdash \tau <: \sigma}{\Gamma \vdash (\textbf{forall}\ a.\tau) <: (\textbf{forall}\ a.\sigma)}$$

Refinement types in Liquid Haskell

In Liquid Haskell, a refinement type has the form $\{x : T \mid P\}$ and can be defined in a Liquid Haskell block by means of a type synonym:

```
{-@ type TRUE = { _:() | True } @-}

{-@ type Even = { n:Int | n mod 2 = 0 } @-}

{-@ type NonEmpty a = { xs:[a] | len xs /= 0 } @-}

{-@ type Monotonic = m:Int -> {n:Int | m <= n} @-}
```

Refinement types in Liquid Haskell

In Liquid Haskell, a refinement type has the form $\{x : T \mid P\}$ and can be defined in a Liquid Haskell block by means of a type synonym:

```
{-@ type TRUE = { _:() | True } @-}

{-@ type Even = { n:Int | n mod 2 = 0 } @-}

{-@ type NonEmpty a = { xs:[a] | len xs /= 0 } @-}

{-@ type Monotonic = m:Int -> {n:Int | m <= n} @-}
```

Refinement types in Liquid Haskell

In Liquid Haskell, a refinement type has the form $\{x : T \mid P\}$ and can be defined in a Liquid Haskell block by means of a type synonym:

```
{-@ type TRUE = { _:() | True } @-}

{-@ type Even = { n:Int | n mod 2 = 0 } @-}

{-@ type NonEmpty a = { xs:[a] | len xs /= 0 } @-}

{-@ type Monotonic = m:Int -> {n:Int | m <= n} @-}
```

Refinement types in Liquid Haskell

In Liquid Haskell, a refinement type has the form $\{x : T \mid P\}$ and can be defined in a Liquid Haskell block by means of a type synonym:

```
{-@ type TRUE = { _:() | True } @-}

{-@ type Even = { n:Int | n mod 2 = 0 } @-}

{-@ type NonEmpty a = { xs:[a] | len xs /= 0 } @-}

{-@ type Monotonic = m:Int -> {n:Int | m <= n} @-}
```

Refinement types in Liquid Haskell

In Liquid Haskell, a refinement type has the form $\{x : T \mid P\}$ and can be defined in a Liquid Haskell block by means of a type synonym:

```
{-@ type TRUE = { _:() | True } @-}
```

```
{-@ type Even = { n:Int | n mod 2 = 0 } @-}
```

```
{-@ type NonEmpty a = { xs:[a] | len xs /= 0 } @-}
```

```
{-@ type Monotonic = m:Int -> {n:Int | m <= n} @-}
```

Refinement types in Liquid Haskell: predicates

Predicate synonyms can also be defined in a Liquid Haskell:

```
{-@ predicate IsNotDivisibleBy M N = M mod N /= 0 @-}

{-@ predicate NotReducible M N = N = 1 || IsNotDivisibleBy M N @-}

{-@ predicate IsZero M N = M = 0 && N = 1 @-}

{-@ predicate IsRational M N
      = NotReducible M N && NotReducible N M || IsZero M N @-}

{-@ type Positive = { n:Int | n > 0 } @-}

{-@ type Rational
     = { m:(Int, Positive) | IsRational (fst m) (snd m) } @-}
```

Refinement types in Liquid Haskell: predicates

Predicate synonyms can also be defined in a Liquid Haskell:

```
{-@ predicate IsNotDivisibleBy M N = M mod N /= 0 @-}

{-@ predicate NotReducible M N = N = 1 || IsNotDivisibleBy M N @-}

{-@ predicate IsZero M N = M = 0 && N = 1 @-}

{-@ predicate IsRational M N
      = NotReducible M N && NotReducible N M || IsZero M N @-}

{-@ type Positive = { n:Int | n > 0 } @-}

{-@ type Rational
     = { m:(Int, Positive) | IsRational (fst m) (snd m) } @-}
```

Refinement types in Liquid Haskell: predicates

Predicate synonyms can also be defined in a Liquid Haskell:

```
{-@ predicate IsNotDivisibleBy M N = M mod N /= 0 @-}

{-@ predicate NotReducible M N = N = 1 || IsNotDivisibleBy M N @-}

{-@ predicate IsZero M N = M = 0 && N = 1 @-}

{-@ predicate IsRational M N
      = NotReducible M N && NotReducible N M || IsZero M N @-}

{-@ type Positive = { n:Int | n > 0 } @-}

{-@ type Rational
     = { m:(Int, Positive) | IsRational (fst m) (snd m) } @-}
```

Refinement types in Liquid Haskell: predicates

Predicate synonyms can also be defined in a Liquid Haskell:

```
{-@ predicate IsNotDivisibleBy M N = M mod N /= 0 @-}

{-@ predicate NotReducible M N = N = 1 || IsNotDivisibleBy M N @-}

{-@ predicate IsZero M N = M = 0 && N = 1 @-}

{-@ predicate IsRational M N
      = NotReducible M N && NotReducible N M || IsZero M N @-}

{-@ type Positive = { n:Int | n > 0 } @-}

{-@ type Rational
      = { m:(Int, Positive) | IsRational (fst m) (snd m) } @-}
```

Refinement types in Liquid Haskell: predicates

Predicate synonyms can also be defined in a Liquid Haskell:

```
{-@ predicate IsNotDivisibleBy M N = M mod N /= 0 @-}

{-@ predicate NotReducible M N = N = 1 || IsNotDivisibleBy M N @-}

{-@ predicate IsZero M N = M = 0 && N = 1 @-}

{-@ predicate IsRational M N
      = NotReducible M N && NotReducible N M || IsZero M N @-}

{-@ type Positive = { n:Int | n > 0 } @-}

{-@ type Rational
     = { m:(Int, Positive) | IsRational (fst m) (snd m) } @-}
```

Refinement types in Liquid Haskell: pairs and functions

Liquid Haskell supports a restricted form of both dependent pairs and functions in which bound variables may only appear within predicate refinements.

```
{-@ type Inverses = (x : Int, {y : Int | x + y = 0}) @-}
```

```
{-@ type Sized a = n:Nat -> {xs:[a] | len xs = n} @-}
```

```
{-@ type Factorise = o:Int -> (m:Int, { n: Int | o = m * n }) @-}
```

Refinement types in Liquid Haskell: pairs and functions

Liquid Haskell supports a restricted form of both dependent pairs and functions in which bound variables may only appear within predicate refinements.

```
{-@ type Inverses = (x : Int, {y : Int | x + y = 0}) @-}
```

```
{-@ type Sized a = n:Nat -> {xs:[a] | len xs = n} @-}
```

```
{-@ type Factorise = o:Int -> (m:Int, { n: Int | o = m * n }) @-}
```

Refinement types in Liquid Haskell: pairs and functions

Liquid Haskell supports a restricted form of both dependent pairs and functions in which bound variables may only appear within predicate refinements.

```
{-@ type Inverses = (x : Int, {y : Int | x + y = 0}) @-}
```

```
{-@ type Sized a = n:Nat -> {xs:[a] | len xs = n} @-}
```

```
{-@ type Factorise = o:Int -> (m:Int, { n: Int | o = m * n }) @-}
```

Refinement types in Liquid Haskell: sums and record types

Liquid Haskell also supports refinements of *inductive* sum and record types.

Sized Vectors

```
data Vector a = V { size :: Int , elems :: [a] }
data Vector a = V { size :: Nat, elems :: { xs :[ a] | len xs = size } }
```

Refinement types in Liquid Haskell: sums and record types

Liquid Haskell also supports refinements of *inductive* sum and record types.

Binary Search Trees

```
data Tree a
  = Leaf
  | Node { root  :: a
         , left  :: Tree a
         , right :: Tree a
         }
```

Refinement types in Liquid Haskell: sums and record types

Liquid Haskell also supports refinements of *inductive* sum and record types.

Binary Search Trees

```
{-@
data Tree a
  = Leaf
  | Node { root  :: a
         , left  :: Tree {v:a | v < a}
         , right :: Tree {v:a | v > a}
         }
@-}
```

Guards

Let us consider the following function and its refined type:

```
insert :: Ord a => a → Tree a → Tree a
insert v Leaf = Node v Leaf Leaf
insert v t@(Node x l r)
  | v < x       = Node k (insert v l) r
  | v > x       = Node k l (insert v r)
  | otherwise   = t
```

How is Liquid Haskell aware of the assumptions that introduced by guard expressions? This is precisely our reason for including guards in our type-checking environment.

Guards

Let us consider the following function and its refined type:

```
insert :: Ord a => a → Tree a → Tree a
insert v Leaf = Node v Leaf Leaf
insert v t@(Node x l r)
  | v < x       = Node k (insert v l) r
  | v > x       = Node k l (insert v r)
  | otherwise   = t
```

How is Liquid Haskell aware of the assumptions that introduced by guard expressions? This is precisely our reason for including guards in our type-checking environment.

Higher-order functions

Recall the following refinement types for sized vectors:

```
{-@ data Vector a
      = V { size :: Nat, elems :: { xs :[a] | len xs = size } }
@-}
```

We can define a size-sensitive fold over vectors as follows:

```
foldV :: Vector a → (Int → a → b → b) → b → b
foldV (V _ [])       _ z = z
foldV (V sz (x : xs)) f z = f sz x $ foldV (V (sz - 1) xs) f z
```

Higher-order functions

Recall the following refinement types for sized vectors:

```
{-@ data Vector a
      = V { size :: Nat, elems :: { xs :[a] | len xs = size } }
@-}
```

We can define a size-sensitive fold over vectors as follows:

```
foldV :: Vector a → (Int → a → b → b) → b → b
foldV (V _ [])       _ z = z
foldV (V sz (x : xs)) f z = f sz x $ foldV (V (sz - 1) xs) f z
```

Higher-order functions

We can define a size-sensitive fold over vectors as follows:

```
foldV :: (Int -> a -> b -> b) -> b -> Vector a -> b
foldV _ z (V _ []) = z
foldV f z (V sz (x : xs)) = f sz x $ foldV f z $ V (sz - 1) xs
```

Let us consider applying this fold to take the sum of each element multiplied by the size of the vector for which they are the head:

```
sumSize :: Vector Int -> Int
sumSize v = foldV v (\s m n -> n + s * m) 0
```

Can we refine the type of sumSize to Vector Nat → Nat? Yes!

Higher-order functions

We can define a size-sensitive fold over vectors as follows:

```
foldV :: (Int -> a -> b -> b) -> b -> Vector a -> b
foldV _ z (V _ []) = z
foldV f z (V sz (x : xs)) = f sz x $ foldV f z $ V (sz - 1) xs
```

Let us consider applying this fold to take the sum of each element multiplied by the size of the vector for which they are the head:

```
sumSize :: Vector Int -> Int
sumSize v = foldV v (\s m n -> n + s * m) 0
```

Can we refine the type of sumSize to Vector Nat → Nat? Yes!

Higher-order functions

Liquid Haskell incorporates inference for higher-order functions to determine logical conditions that are certain to hold.

For example, it can be inferred that we only apply the folding function of foldV to the sizes of vectors, which we know must be natural numbers.

As such, Liquid Haskell can infer that our foldV function has the refined type: $(\text{Nat} \rightarrow \text{a} \rightarrow \text{b} \rightarrow \text{b}) \rightarrow \text{b} \rightarrow \text{Vector a} \rightarrow \text{b}$.

Higher-order functions

Liquid Haskell incorporates inference for higher-order functions to determine logical conditions that are certain to hold.

For example, it can be inferred that we only apply the folding function of foldV to the sizes of vectors, which we know must be natural numbers.

As such, Liquid Haskell can infer that our foldV function has the refined type: $(\text{Nat} \to a \to b \to b) \to b \to \text{Vector } a \to b$.

Higher-order functions

Liquid Haskell incorporates inference for higher-order functions to determine logical conditions that are certain to hold.

For example, it can be inferred that we only apply the folding function of foldV to the sizes of vectors, which we know must be natural numbers.

As such, Liquid Haskell can infer that our foldV function has the refined type: $(\text{Nat} \rightarrow \text{a} \rightarrow \text{b} \rightarrow \text{b}) \rightarrow \text{b} \rightarrow \text{Vector a} \rightarrow \text{b}$.

Higher-order functions: a caveat

When termination checking is turned on for Liquid Haskell, as it is by default, you'll find that our definition of foldV yields an error.

Without any additional information, Liquid Haskell cannot prove that foldV terminates. However, there is an easy fix: termination metrics.

We need only provide the following refinement for the type of foldV:

foldV :: (Nat → a → b → b) → b → v:Vector a → b / [size v]

Next Lecture

In the next lecture we will look more closely at termination metrics, measures, and the manual proof combinators available in Liquid Haskell.

Please join me in the labs tomorrow to practice applying refinements to trees, vectors and more with a few exercises.