Refinement Types in Haskell

Lecture 3: Measures, proofs, and termination

Brandon Hewer

Today's Lecture

In today's lecture, we will begin by introducing the notion of a measure in Liquid Haskell and consider a few examples.

We will then introduce termination metrics and demonstrate how they can be used through the example of parallel substitutions.

We will consider how to write manual proofs in Liquid Haskell using reflection and the proof combinators library.

Finally, we will introduce proof by logical evaluation, and then present a correctness proof for Hutton's razor.

Lecture Outline

1. Today's Lecture
2. **Lecture Outline**
3. Motivation for measures
4. What is a measure?
5. Example: Balanced BSTs
6. Termination metrics
7. Example: Parallel substitutions
8. Reflection and proof combinators
9. Reflection and proof by logical evaluation
10. Correctness of Hutton's razor

Motivation for measures

Thus far, we have considered simple examples of refinement types whose refinement predicates range over simple relations and operations on numbers and booleans.

Indeed, the only additional structure that we have been subtly making use of is the len function on lists. For example:

```
type Vector a N = { xs:[a] | len xs = N }
```

Motivation for measures

What if we wished to define predicates that made use of additional structure on our types? For example, lists that contain even numbers, lists that do not contain duplicates, or even balanced binary search trees.

The refinement logic of Liquid Haskell that we have considered thus far is insufficient to define predicates for these types.

This is what precisely what measures are intended to address, and the `len` function was in fact our first use of a measure!

What is a measure?

A *measure* is simply a function on an algebraic datatype that explicitly matches on each constructor of that type.

The len function is a measure on lists that is defined as follows:

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

However, the following is not a measure on lists:

```
isNotEmpty :: Eq a => [a] -> Bool
isNotEmpty xs = xs == []
```

What is a measure?

To define our own measures on datatypes, we need only define a Haskell function that matches on each constructor and annotate it with {−@ measure name_of_function @−}.

Example 1: Lists of even numbers

```
{−@ measure isAllEven @−}
isAllEven  ::  [Int] → Bool
isAllEven  []          = True
isAllEven  (n : ns) = n `mod` 2 == 0 && isAllEven ns

{−@ type Evens = { xs:[Int]  |  isAllEven  xs } @−}
```

What is a measure?

To define our own measures on datatypes, we need only define a Haskell function that matches on each constructor and annotate it with {−@ measure name_of_function @−}.

Example 2: Lists of even numbers (again)

```
{−@ measure odds @−}
odds :: [Int] → [Int]
odds [] = []
odds (n : ns)
  | n `mod` 2 /= 0 = n : odds ns
  | otherwise      = odds ns

{−@ type Evens = { xs:[Int] | odds xs = [] } @−}
```

Example: Balanced BSTs

Let us now consider using a measure to define balanced binary search trees in Liquid Haskell.

We begin with a simple definition of binary trees:

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

Example: Balanced BSTs

Let us now consider using a measure to define balanced binary search trees in Liquid Haskell.

Next, we implement a measure for the depth of a tree:

```
{-@ measure depth @-}
depth :: Tree a → Int
depth Leaf = 0
depth (Node _ l r)
    | dl >= dr = 1 + dl
    | otherwise = 1 + dr
    where
    dl = depth l
    dr = depth r
```

Example: Balanced BSTs

Let us now consider using a measure to define balanced binary search trees in Liquid Haskell.

With the depth measure, we define balanced BSTs as follows:

```
{-@ predicate IsBalanced T U
      = depth T - depth U <= 1 && depth U - depth T <= 1
 @-}

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

Termination metrics

By default, Liquid Haskell will perform termination checking on all functions.

Without any additional information being provided, termination checking is restricted to *structural termination*.

Informally, this means that each recursive call must be applied to a structurally smaller input for at least one of the input arguments.

Unfortunately, this condition is particularly limiting, and we do not need to look far to find an example where it is insufficient.

Termination metrics

By default, Liquid Haskell will perform termination checking on all functions.

Without any additional information being provided, termination checking is restricted to *structural termination*.

Informally, this means that each recursive call must be applied to a structurally smaller input for at least one of the input arguments.

Unfortunately, this condition is particularly limiting, and we do not need to look far to find an example where it is insufficient.

Termination metrics

By default, Liquid Haskell will perform termination checking on all functions.

Without any additional information being provided, termination checking is restricted to *structural termination*.

Informally, this means that each recursive call must be applied to a structurally smaller input for at least one of the input arguments.

Unfortunately, this condition is particularly limiting, and we do not need to look far to find an example where it is insufficient.

Termination metrics

By default, Liquid Haskell will perform termination checking on all functions.

Without any additional information being provided, termination checking is restricted to *structural termination*.

Informally, this means that each recursive call must be applied to a structurally smaller input for at least one of the input arguments.

Unfortunately, this condition is particularly limiting, and we do not need to look far to find an example where it is insufficient.

Termination metrics

Consider the following definition of the merge function:

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs []     = xs
merge [] ys     = ys
merge (x:xs) (y:ys)
  | x < y       = x : merge xs (y:ys)
  | otherwise   = y : merge ys (x:xs)
```

This definition is not structurally terminating as neither of its inputs are strictly smaller at each recursive call.

We need to prove to Liquid Haskell that merge is terminating in another way: by using termination metrics.

Termination metrics

Consider the following definition of the merge function:

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs []     = xs
merge [] ys     = ys
merge (x:xs) (y:ys)
  | x < y       = x : merge xs (y:ys)
  | otherwise   = y : merge ys (x:xs)
```

This definition is not structurally terminating as neither of its inputs are strictly smaller at each recursive call.

We need to prove to Liquid Haskell that merge is terminating in another way: by using termination metrics.

Termination metrics

Consider the following definition of the merge function:

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs []     = xs
merge [] ys     = ys
merge (x:xs) (y:ys)
  | x < y       = x : merge xs (y:ys)
  | otherwise   = y : merge ys (x:xs)
```

This definition is not structurally terminating as neither of its inputs are strictly smaller at each recursive call.

We need to prove to Liquid Haskell that merge is terminating in another way: by using termination metrics.

Termination metrics

A *termination metric* in a given context is simply a set of well-formed natural number-valued expressions in that context.

Intuitively, each expression of a termination metric corresponds to a 'size' that we expect to reduce throughout the computation.

A recursive function provably terminates if for each recursive call, at least one expression in its termination metric becomes provably smaller after substituting for the new arguments.

Termination metrics

A *termination metric* in a given context is simply a set of well-formed natural number-valued expressions in that context.

Intuitively, each expression of a termination metric corresponds to a 'size' that we expect to reduce throughout the computation.

A recursive function provably terminates if for each recursive call, at least one expression in its termination metric becomes provably smaller after substituting for the new arguments.

Termination metrics

A *termination metric* in a given context is simply a set of well-formed natural number-valued expressions in that context.

Intuitively, each expression of a termination metric corresponds to a 'size' that we expect to reduce throughout the computation.

A recursive function provably terminates if for each recursive call, at least one expression in its termination metric becomes provably smaller after substituting for the new arguments.

Termination metrics

For example, in Lecture 2 we introduced the following fold function on sized vectors:

```
{−@ foldV :: (Nat → a → b → b) → b → v:Vector a → b @−}
foldV _ z (V _ []) = z
foldV f z (V sz (x : xs)) = f sz x $ foldV f z $ V (sz − 1) xs
```

A valid termination metric in the output of foldV can make use of the bound variable v, e.g. `size v`.

Termination metrics

We mentioned previously that this definition of foldV does not actually type-check in Liquid Haskell. The following correction addresses this:

```
{-@ foldV :: (Nat → a → b → b) → b → v:Vector a → b / [size v] @-}
foldV _ z (V _ []) = z
foldV f z (V sz (x : xs)) = f sz x $ foldV f z $ V (sz − 1) xs
```

In general, termination metrics are introduced by merely adding a / followed by a sequence of integer-valued expressions.

Termination metrics

Returning to our example of the merge function on lists, we can apply termination metrics to prove to Liquid Haskell that it terminates as follows:

```
{−@ merge :: xs:[a] → ys:[a] → [a] / [len xs + len ys]@−}
merge xs []     = xs
merge [] ys     = ys
merge (x:xs) (y:ys)
    | x < y       = x:(merge xs (y:ys))
    | otherwise = y:(merge ys (x:xs))
```

It is not difficult to see that the expression len xs + len ys does indeed reduce at each step.

Termination metrics

Returning to our example of the merge function on lists, we can apply termination metrics to prove to Liquid Haskell that it terminates as follows:

```
{−@ merge :: xs:[a] → ys:[a] → [a] / [len xs + len ys]@−}
merge xs []     = xs
merge [] ys     = ys
merge (x:xs) (y:ys)
    | x < y       = x:(merge xs (y:ys))
    | otherwise = y:(merge ys (x:xs))
```

It is not difficult to see that the expression len xs + len ys does indeed reduce at each step.

Example: parallel substitutions

We will now consider a more involved example that makes use of both termination metrics and measures: parallel substitution for de Bruijin indexed lambda terms (presented in 3.3 in Dependent Types and Multi Monadic Effects in F*).

Example: parallel substitutions

First, we will define the types of de Bruijin indexed lambda terms and substitutions in Haskell as follows:

```
data Expr = Var Int  |  Lam Expr | App Expr Expr

{-@ type Expr = Var Nat | Lam Expr | App Expr Expr @-}

{-@ type Subst = [(Nat, Expr)] @-}
```

Note: we will consider how to define the type of *closed* lambda terms in tomorrow's lab.

Example: parallel substitutions

Next, we define the following measure that corresponds to the 'size' of an expression:

```
{−@ measure elen @−}
{−@ elen :: Expr → Nat @−}
elen (Var v)     = 0
elen (Lam e)     = 1 + elen e
elen (App e1 e2) = 1 + elen e1 + elen e2
```

Example: parallel substitutions

Meanwhile, for substitutions we define a measure that checks whether a substitution is simply an α-renaming:

```
{-@ measure mysnd @-}
snd' :: (a, b) → b
snd' (_, y) = y

{-@ measure isVar @-}
isVar :: Expr → Bool
isVar (Var _) = True
isVar _       = False

{-@ measure isRenaming @-}
isRenaming :: Subst → Bool
isRenaming []         = True
isRenaming (vx:sus) = isVar (snd' vx) && isRenaming sus
```

Example: parallel substitutions

Next, we introduce a refined type of expressions that are indexed over both an expression and a substitution:

```
{-@ type SExpr E S
      = {v:Expr | (isVar E && isRenaming S) => isVar v }
 @-}
```

This will be the output type of our parallel substitution operation on expressions.

Example: parallel substitutions

For variables, we can define the following simple substitution function:

```
{−@ sub :: su:Subst → v:Nat →
                {v:Expr | isRenaming su => isVar v } @−}
sub []  v = Var v
sub ((vx,x):su) v
  | v == vx  = x
  | otherwise = sub su v
```

Example: parallel substitutions

To handle lambda expressions, we will make use of the following auxiliary definition for the type of substitutions that are simply renamings, together with the incrementing substitution:

```
{-@ type RenamingSubst = { s:Subst | isRenaming s } @-}

{-@ incrsubst :: RenamingSubst @-}
incrsubst :: Subst
incrsubst = [ (i, Var $ i + 1) | i <- [0..] ]
```

Example: parallel substitutions

Finally, we can bring all of this together to define our parallel substitution function. First, let us introduce its type:

```
{−@ subst :: e:Expr → s:Subst → SExpr e s
      / [ if (isVar e) then 0 else 1
        , if (isRenaming s) then 0 else 1
        , elen e
        ]
 @−}
```

To satisfy this type, our definition of subst must become smaller at each step by either being applied to a variable, being applied to a renaming substitution or becoming smaller in its ‘size’.

Example: parallel substitutions

We conclude by defining our substitution function as follows:

```
subst (Var v)       s = sub su v
subst (App e1 e2) s = App (subst e1 s) (subst e2 s)
subst (Lam e)   s | isRenaming s =
  Lam $ subst e
    $ (0, Var 0) : [ (i, subst (sub su (i-1)) incrsubst) | i <- [1..] ]

subst (Lam e)   su =
    Lam $ subst e
      $ (0, Var 0) : [ (i, subst (sub su (i-1)) incrsubst) | i <- [1..] ]
```

As we should hope, this does indeed type-check in Liquid Haskell.

Reflection and proof combinators

Programs are proofs!

Reflection and proof combinators

This correspondence finds its most explicit application in proof assistants such as Agda, Coq, Lean and Isabelle.

An essential feature of proving theorems in proof assistants is the reflection of definitional equalities in the type system.

By default, the defining equations of functions are not reflected in the refinement logic of Liquid Haskell. However, this can be addressed by means of *reflection*.

Reflection and proof combinators

In Liquid Haskell, to reflect both inductive data constructors and functions, we add {−@ LIQUID "−−reflection" @−} to our file and annotate functions we wish to reflect with {−@ reflect name_of_function @−}.

Example: Hutton's Razor

```
data Expr = Val Int | Add Expr Expr

{−@ reflect eval @−}
eval :: Expr → Int
eval (Val n)      = n
eval (Add e1 e2) = eval e1 + eval e2
```

Reflection and proof combinators

Reflection transforms Liquid Haskell into a theorem prover.

To take advantage of this Liquid Haskell provides a proof combinator library that can be imported with:

```
import Language.Haskell.Liquid.ProofCombinators
```

Reflection and proof combinators

Combinators in Liquid Haskell's proof library include:

Composition of equality proofs

$(===) :: x{:}a \rightarrow y{:}\{a \mid y = x\} \rightarrow \{v{:}a \mid v = x \;\&\&\; v = y\}$

Reflection and proof combinators

Combinators in Liquid Haskell's proof library include:

Composition of ordering proofs

$(=<=)$ :: x:a $\rightarrow$ y:{a | x <= y} $\rightarrow$ {v:a | v = y}
$(=>=)$ :: x:a $\rightarrow$ y:{a | x >= y} $\rightarrow$ {v:a | v = y}

Reflection and proof combinators

Combinators in Liquid Haskell's proof library include:

Horizontal proof composition ('because')

```
(?) :: forall a b <pa :: a → Bool, pb :: b → Bool>.
       a<pa> → b<pb> → a<pa>
```

Reflection and proof combinators

Combinators in Liquid Haskell's proof library include:

Triviality and absurdity

```
type Proof = ()

trivial     :: {v : Proof | True}
unreachable :: {v : Proof | False}
```

Reflection and proof combinators

Combinators in Liquid Haskell's proof library include:

Proof holes and QED

```
data QED = Admit | QED

(***) :: a → p:QED → { if (isAdmit p) then false else true }
```

Reflection and proof combinators

With some of the proof combinators that we have just introduced, let us consider how to prove associativity of concatenation for the following custom list type:

```
data List a = Nil | Cons a (List a)

(++) :: List a → List a → List a
Nil         ++ ys = ys
Cons x xs   ++ ys = Cons x (xs ++ ys)
```

Reflection and proof combinators

```
data List a = Nil | Cons a (List a)


{-@ assoc :: xs:List a -> ys:List a -> zs:List a
          -> xs ++ (ys ++ zs) == (xs ++ ys)
@-}
assoc Nil ys zs
  =     Nil <> (y <> z)
  === y <> z
  === (Nil <> y) <> z
  *** QED
```

Reflection and proof combinators

```
data List a = Nil | Cons a (List a)


{-@ assoc :: xs:List a -> ys:List a -> zs:List a
          -> xs ++ (ys ++ zs) == (xs ++ ys)
@-}
assoc (Cons x xs) ys zs
  =     (Cons x xs) <> (y <> z)
  === Cons x (xs <> (y <> z))
  === Cons x ((xs <> y) <> z) ? assoc xs y z
  === (Cons x (xs <> y)) <> z
  === ((Cons x xs) <> y) <> z
  *** QED
```

Reflection and proof combinators

As a second example, we will consider how we can write a proof that the Add constructor of Hutton's Razor is associative under the eval function.

In particular, we want to construct a function of the following type:

```
{-@ evalAssoc
      :: e1:Expr → e2:Expr → e3:Expr
      → { eval (Add e1 (Add e2 e3)) = eval (Add (Add e1 e2) e3) }
 @-}
```

Reflection and proof combinators

We can construct a proof of this theorem as follows:

```
evalAssoc e1 e2 e3 =
      eval  (Add e1 (Add e2 e3))
=== eval e1 + eval (Add e2 e3)
=== eval e1 + eval e2 + eval e3
=== eval (Add e1 e2) + eval e3
=== eval (Add (Add e1 e2) e3)
***   QED
```

Reflection and proof combinators

We can construct a proof of this theorem as follows:

```
evalAssoc e1 e2 e3 =
        eval  (Add e1 (Add e2 e3))
=== eval e1 + eval (Add e2 e3)
=== eval e1 + eval e2 + eval e3
=== eval (Add e1 e2) + eval e3
=== eval (Add (Add e1 e2) e3)
***   QED
```

Reflection and proof combinators

We can construct a proof of this theorem as follows:

```
evalAssoc e1 e2 e3 =
        eval  (Add e1 (Add e2 e3))
=== eval e1 + eval (Add e2 e3)
=== eval e1 + eval e2 + eval e3
=== eval (Add e1 e2) + eval e3
=== eval (Add (Add e1 e2) e3)
***   QED
```

Reflection and proof combinators

We can construct a proof of this theorem as follows:

```
evalAssoc e1 e2 e3 =
        eval  (Add e1 (Add e2 e3))
=== eval e1 + eval (Add e2 e3)
=== eval e1 + eval e2 + eval e3
=== eval (Add e1 e2) + eval e3
=== eval (Add (Add e1 e2) e3)
***   QED
```

Reflection and proof combinators

We can construct a proof of this theorem as follows:

```
evalAssoc e1 e2 e3 =
        eval  (Add e1 (Add e2 e3))
=== eval e1 + eval (Add e2 e3)
=== eval e1 + eval e2 + eval e3
=== eval (Add e1 e2) + eval e3
=== eval (Add (Add e1 e2) e3)
***   QED
```

Reflection and proof combinators

We can construct a proof of this theorem as follows:

```
evalAssoc e1 e2 e3 =
        eval  (Add e1 (Add e2 e3))
=== eval e1 + eval (Add e2 e3)
=== eval e1 + eval e2 + eval e3
=== eval (Add e1 e2) + eval e3
=== eval (Add (Add e1 e2) e3)
***   QED
```

Proof by logical evaluation

Notably, the only manual work we performed to construct the proof evalAssoc was to layout each of the unfolding steps along the definitional equalities of evalAssoc. Liquid Haskell took care of associativity of integer addition for us.

But, in the case that we need only unfold along a (small) finite number of definitional equalities to construct a proof, then this is also a mechanical process that can be automated!

Indeed, Liquid Haskell provides a feature for handling this part of the proof for us too: proof by logical evaluation.

Proof by logical evaluation

Proof by logical evaluation (PLE) is a guard-sensitive normalisation procedure that incorporates various additional features such as rewriting (REST 2012, Z.Grannan et. al.).

PLE can be enabled for a specific proof by annotating it with {−@ ple name_of_function @−} or for an entire file with {−@ LIQUID "−−reflection" @−}.

Proof by logical evaluation

Let us again consider constructing a proof of evalAssoc, but this time we will make use of PLE:

```
{-@ ple evalAssoc @-}
{-@ evalAssoc
        :: e1:Expr → e2:Expr → e3:Expr
        → { eval (Add e1 (Add e2 e3)) = eval (Add (Add e1 e2) e3) }
 @-}
evalAssoc _ _ _ = trivial
```

Proof by logical evaluation

Let us again consider constructing a proof of evalAssoc, but this time we will make use of PLE:

```
{-@ ple evalAssoc @-}
{-@ evalAssoc
        :: e1:Expr → e2:Expr → e3:Expr
        → { eval (Add e1 (Add e2 e3)) = eval (Add (Add e1 e2) e3) }
 @-}
evalAssoc _ _ _ = trivial
```

Next Lecture

In the next lecture we will introduce quotient types, consider how they can be integrated into a refinement type system.

We will consider examples that demonstrate the expressive power of quotients, and consider alternative definitions for types we've already introduced.

Please join me in the lab tomorrow where we will prove the correctness of a compiler for Hutton's Razor using the techniques introduced in today's lecture.