Liquid-Structures

statically verifying data structure invariants with LiquidHaskell

John Kastner

LiquidHaskell

Banker's Queue

Red-Black Tree

Introduction

- Goal: statically verify data structure invariants
- Data structure implementations adapted from *Purely Functional Data Structures*
- LiquidHaskell's refinement types used to encode and statically check invariants

LiquidHaskell

LiquidHaskell

- An extension to the Haskell programming language
- Haskell already has a strong static type system but, it lacks dependant types such as those in Coq
- LiquidHaskell lets you annotate types with logical predicates (refinements)
- This is less powerful than Coq's dependant types because predicates must be solvable by an SMT solver

Refinement Types

- Consider a trivial Haskell expression: `1`
- Its type: `Int`
- This doesn't precisely characterize the expression. Refinement types can be used to improve the specification.

```
{-@ x :: {n:Int | n > 0} @-}
x :: Int
x = 1

{-@ y :: {n:Int | n == 1} @-}
y :: Int
y = 1
```

Refining Functions

Refinement types are much more interesting when applied to function argument types and return types.

Postconditions

```
{-@ abs :: n:Int -> {m:Int | m >= 0} @-}
abs n | n < 0      = - n
      | otherwise = n
```

Preconditions

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

Interesting Combinations

```
{-@ fib :: {n:Int | n >= 0} -> {v:Int | v >= 0} @-}
fib n | n <= 1     = n
      | otherwise = fib (n - 1) + fib (n - 2)
```

Refining Data Types

- Just like functions, data types can be refined.
- This defines the usual cons list but, the tail is recursively defined as a list where each element must be less than or equal the head.

```
{-@ data List a = Nil
                | Cons {
                  hd :: a,
                  tl :: List {v : a | v >= hd}
                }
  @-}
{-@ measure llen :: List a -> Nat
    llen Nil          = 0
    llen (Cons _ tl) = 1 + llen tl
  @-}
list_good = Cons 1 (Cons 2 Nil)
{- list_bad = Cons 2 (Cons 1 Nil) -}
```

Banker's Queue

Banker's Queue

- A Queue data structure designed for functional programming languages
- Provides efficient read access to head and append access to tail
- Maintains two lists: the first is some prefix of the queue while the second is the remaining suffix of the queue
- The invariant is that the prefix list cannot be shorter than the suffix list

Banker's Queue Datatype

- The interesting refinement type is on `lenr` which states that the length of the rear must be less than or equal to the length of the front
- The other refinements ensure the stored lengths are in fact the real lengths.

```
{-@ data BankersQueue a = BQ {
      lenf :: Nat,
      f    :: {v:[a] | len v == lenf},
      lenr :: {v:Nat | v <= lenf},
      r    :: {v:[a] | len v == lenr}
    }
  @-}
{-@ measure qlen :: BQ a -> Nat
    qlen (BQ f _ r _) = f + r
  @-}
type BQ a = BankersQueue a
```

Catching a Violated Invariant

- Using this definition, (some) errors will be automatically detected

```
snoc (BQ lenf f lenr r) x = BQ lenf f (lenr+1) (x:r)
```

- LiquidHaskell finds that `snoc` does not maintain the length invariant between the front and rear

```
165 | snoc (BQ lenf f lenr r) x = BQ lenf f (lenr+1) (x:r)
                                            ^^^^^^^^

  Inferred type
    VV : {v : GHC.Types.Int | v == lenr + 1}

  not a subtype of Required type
    VV : {VV : GHC.Types.Int | VV >= 0
                             && VV <= lenf}
```

Smart Constructor

- How can a queue be constructed if the invariant is not known?
- Write a function to massage data with weaker constraints until the invariant holds.

```
{-@ check ::
    vlenf : Nat                  ->
    {v:[_] | len v == vlenf} ->
    vlenr : Nat                  ->
    {v:[_] | len v == vlenr} ->
    {q:BQ _ | qlen q == (vlenf + vlenr)}
  @-}
check lenf f lenr r =
  if lenr <= lenf then
    BQ lenf f lenr r
  else
    BQ (lenf + lenr) (f ++ (reverse r)) 0 []
```

Banker's Queue Functions

Snoc

- An element can be added to a queue
- This maintains invariants and increments the length

```
{-@ snoc :: q0:BQ a -> a ->
            {q1:BQ a | (qlen q1) == (qlen q0) + 1} @-}
snoc (BQ lenf f lenr r) x = check lenf f (lenr+1) (x:r)
```

Head and tail

- After adding an element, it can be retrieved and removed
- Both functions require non-empty queues

```
{-@ head :: {q:BQ a | qlen q /= 0} -> a @-}
head (BQ lenf (x : f') lenr r) = x

{-@ tail :: {q0:BQ a | qlen q0 /= 0} ->
            {q1:BQ a | (qlen q1) == (qlen q0) - 1} @-}
tail (BQ lenf (x : f') lenr r) = check (lenf-1) f' lenr r
```

Red-Black Tree

Red-Black Tree

- A Red-Black Tree is a binary search tree with two key invariants.
  - **Red Invariant**: No red node has a red child.
  - **Black Invariant**: Every path from the root to an empty node contains the same number of black nodes
- The invariants keep the try approximately balanced
- When invariants are violated, the tree is rotated in such a way that they are restored

Red-Black Tree Datatype

- BST ordering is enforced by recursive refinements on the sub-trees.
- Red and black invariants are enforced by respective predicates

```
data Color = Red | Black deriving Eq
{-@ data RedBlackTree a = Empty |
      Tree { color :: Color,
             val   :: a,
             left  :: {v:RedBlackTree {vv:a | vv < val} |
                         RedInvariant color v},
             right :: {v:RedBlackTree {vv:a | vv > val} |
                         RedInvariant color v &&
                         BlackInvariant v left}}@-}
{-@ predicate RedInvariant C S =
      (C == Red) ==> (getColor S /= Red) @-}
{-@ predicate BlackInvariant S0 S1 =
      (blackHeight S0) == (blackHeight S1) @-}
```

Red-Black Tree Insertion

- We can try to write an insertion function for red-black trees
- This is nontrivial and we might do it wrong

```
insert x Empty = Tree Red x Empty Empty
insert x t@(Tree c y a b) | x<y = Tree c y (insert x a) b
                          | x>y = Tree c y a (insert x b)
                          | otherwise = t
```

- LiquidHaskell will generate a warning if this error causes the data structures invariants to no longer hold

```
284 |   | x < y     = Tree c y (insert x a) b
                               ^^^^^^^^^^^^

   Inferred type
     VV:{v:(Main.RedBlackTree a##xo) | blackHeight v >= 0
                                       && v == ?a}
   not a subtype of Required type
     VV:{VV:(Main.RedBlackTree {VV:a##xo | VV < y}) |
        c == Red => getColor VV /= Red}
```

Red-Black Tree Balancing

- There is a function to fix an incomplete Red-Black Tree

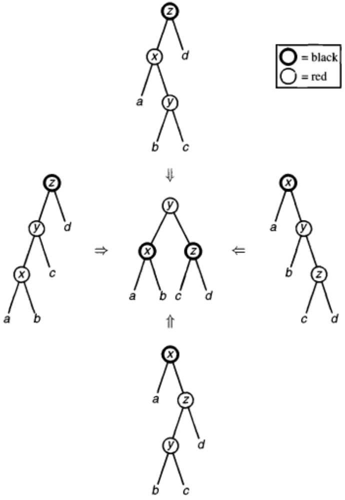

Diagram taken from *Purely Function Data Structures*

Red-Black Tree (Real) Insertion

```
{-@ insert :: e:a -> v:RedBlackTree a -> RedBlackTree a @-}
insert x s = forceRedInvarient (rb_insert_aux x s)
  where forceRedInvarient (WeakRedInvariant _ e a b) =
          Tree Black e a b

{-@ rb_insert_aux :: forall a. Ord a =>
      x:a ->
      s:RedBlackTree a ->
      {v:WeakRedInvariant a |
        (getColor s /= Red ==> HasStrongRedInvariant v)&&
        (weakBlackHeight v) == (blackHeight s)}
  @-}
rb_insert_aux x Empty = WeakRedInvariant Red x Empty Empty
rb_insert_aux x (Tree c y a b)
  | x < y     = balanceLeft c y (rb_insert_aux x a) b
  | x > y     = balanceRight c y a (rb_insert_aux x b)
  | otherwise = (WeakRedInvariant c y a b)
```

An Extra Data Type

- During insertions and balancing, there are values that are almost red-black trees but are missing part of the red invariant
- This type gives an easy way to represent these values and a way to describe when the invariant does hold

```
{-@ data WeakRedInvariant a = WeakRedInvariant {
        weakColor :: Color,
        weakVal   :: a,
        weakLeft  :: RedBlackTree {vv:a | vv<weakVal},
        weakRight :: {v:RedBlackTree {vv:a | vv>weakVal}|
          (weakColor /= Red ||
          (getColor weakLeft) /= Red ||
          (getColor v) /= Red) &&
          (blackHeight v) == (blackHeight weakLeft)}} @-}
{-@ predicate HasStrongRedInvariant Wri =
      (weakColor Wri) == Red ==>
      (getColor (weakLeft Wri) /= Red &&
       getColor (weakRight Wri) /= Red) @-}
```

Red-Black Tree Balancing Functions

- Smart constructor for red-black trees
- Only partially guarantees the red invariant
- Full invariant obtained in other calls to balance during recursion of after all recursion finishes

```
{-@ balanceLeft :: forall a. Ord a =>
      c:Color ->
      t:a ->
      l:{v:WeakRedInvariant {vv:a | vv < t} |
           c == Red ==> HasStrongRedInvariant v} ->
      r:{v:RedBlackTree {vv:a | vv > t} |
           RedInvariant c v &&
           (blackHeight v) == (weakBlackHeight l)} ->
      {v:WeakRedInvariant a |
           (c /= Red ==> HasStrongRedInvariant v) &&
           (weakBlackHeight v) ==
           (if c==Black then 1 else 0)+weakBlackHeight l}
  @-}
```

Red-Black Tree Balancing Functions

```
{-@ balanceRight :: forall a. Ord a =>
        c:Color ->
        t:a ->
        l:{v:RedBlackTree {vv:a | vv < t} |
             RedInvariant c v} ->
        r:{v:WeakRedInvariant {vv:a | vv > t} |
             (c == Red ==> HasStrongRedInvariant v) &&
             (weakBlackHeight v) == (blackHeight l)} ->
        {v:WeakRedInvariant a |
             (c /= Red ==> HasStrongRedInvariant v) &&
             (weakBlackHeight v) ==
             (if c==Black then 1 else 0)+blackHeight l}
  @-}
```