liquid-structures

This project contains Haskell implementations of data structures from Chris Okasaki's Book Purely Functional Data Structures with invariants statically checked using LiquidHaskell. The implementations are based on the those provided in the appendix of Okasaki's book but, in many cases they have been modified to facilitate the addition of refinement types. There are also some structures that were not taken from the book. These tend to be simpler structures used to experiment with and learn LiquidHaskell.

Verifying the Implementations

Verify the entire project

1. Install Haskell Stack either through get.haskellstack.org or through your package manager.
2. Execute `stack test`. This will obtain the required version of GHC, LiquidHaskell and all other dependencies before building project and checking it with LiquidHaskell. This will take a while.

Verifying single files

1. Install LiquidHaskell according to their instructions.
2. Navigate to the `src` directory and Run `liquid` on an individual file and all files that are imported by that file.

```
cd src
liquid Heap/Heap.hs Heap/LeftistHeap.hs
```

Queues

The first and perhaps simplest abstract data type presented in the book is the first-in-first-out queue. The interface implemented by each concrete queue provides functions for adding to the end, reading and removing from the front, obtaining an empty queue and testing if a given queue is empty. The refinement types for these functions are used to track the length of a queue and to ensure that `head` and `tail` are never called on empty queues.

```
class measure qlen :: forall a. a -> {v:Int | v >= 0}
class Queue q where
  empty   :: forall a.
    {q:(q a) | 0 == qlen q}
```

```
  isEmpty :: forall a.
    q:(q a) -> {v:Bool | v <=> (0 == qlen q)}
  snoc    :: forall a.
    q0:q a -> a -> {q1:q a | (qlen q1) == (qlen q0) + 1}
  head    :: forall a.
    {q:q a | qlen q /= 0} -> a
  tail    :: forall a.
    {q0:q a | qlen q0 /= 0} -> {q1:q a | (qlen q1) == (qlen q0) - 1}
```

Banker's Queue

The banker's queue is designed to enable using the banker's method for analysing amortized cost in a lazy data structure. Two lists are maintained in the data structure. The first is some prefix of the queue while the second is the remaining suffix of the queue. The primary invariant is that the prefix list cannot be shorter than the suffix list. This is checked by LiquidHaskell according to the following refinement type for the queue constructor.

This queue is used as a case study in the LiquidHaskell tutorial, but I have independently reproduced it here.

```
data BankersQueue a = BQ {
  lenf :: Nat,
  f    :: {v:[a] | len v == lenf},
  lenr :: {v:Nat | v <= lenf},
  r    :: {v:[a] | len v == lenr}
}
```

Physicist's Queue

The physicist's queue is similar in structure to the banker's queue but, it is instead designed for analysis using the physicist's method. The data structure maintains the same prefix and suffix lists as the banker's queue with the same constraint on the relative lengths of these lists. What distinguishes it form the banker's queue is that it maintains a second prefix list. The second prefix, as well as being a prefix of the entire queue, must be a prefix of the original prefix and it must be non-empty whenever the original prefix is non-empty.

The refinements for the first four record fields are identical to the banker's queue. The remaining two fields encode the more interesting second prefix list. The list itself is stored as `pre`. The refinement for `pre` forces the second prefix to be non-empty if the original prefix is not empty. The final field, `isPrefix` encodes the actual prefix relationship between `pre` and `f`. This is done using an inductive predicate similar to what can be used in Coq. To do this in Haskell,

LiquidHaskell can be used as a theorem prover which provides the means for writing such predicates and the ability to construct proofs using the predicates.

The inductive predicate for prefix has two cases. A list can be empty, in which case it is a prefix of every list. A list can be non-empty, in which case it is only the prefix of another list if they have same first element and the tail of the prefix is the prefix of the tail of the list.

While introducing an extra field to hold an inductive predicate is an interesting method for enforcing invariants, it has a significant draw back. Constructing the proof incurs an extra cost at runtime. Enforcing invariants purely with refinement types incurs no cost at runtime.

```
data PhysicistsQueue a = PQ {
  lenf :: Nat,
  f    :: {v:List a | llen v == lenf},
  lenr :: {v:Nat | v <= lenf},
  r    :: {v:List a | llen v == lenr},
  pre  :: {v:List a | (lenf /= 0 ==> llen v /= 0)},
  isPrefix :: Prop (Prefix pre f)
}
data Prefix a where
    PrefixNil  :: l:List a ->
                  Prop (Prefix Nil l)
  | PrefixCons :: h:a -> l0:List a -> l1:List a ->
                  Prop (Prefix l0 l1) ->
                  Prop (Prefix (Cons h l0) (Cons h l1))
data List a = Nil | Cons a (List a)
```

Sets

Okasaki's book presents a very simple interface for sets. Only insertion and a membership predicate are required. This could, of course, be extended to support other common functions over sets.

The refinement types for `empty` and `insert` encode these operations in terms of functions understood by LiquidHaskell's SMT solver. At the moment, the type of member is not refined.

```
class Set s a where
  empty  ::
    {v:s a | Set_emp (setElts v)}
  insert ::
    e:a -> v:s a -> {vv: s a | (setElts vv) == Set_cup (Set_sng e) (setElts v)}
  member ::
    e:a -> v:s a -> Bool
```

I want to ensure that member returns true if and only if the element is in the set. Writing a refinement type that encodes this property is simple but, I have not been able to write a set implementation that checks using such a refinement type.

```
member :: e:a -> v:s a -> {b:Bool | b <=> Set_mem e (setElts b)}
```

Unbalanced Set

Unbalanced set implements the set interface using an unbalanced binary search tree. Since it is a binary search tree, the invariant checked in the constructor is that every element of the left subtree is less than the root node and every element of the right subtree is greater than the root node. This is accomplished by refining the type of the left and right sub trees with a predicate that applies to every element in the tree.

```
data UnbalancedSet a =
  E | T {
    val   :: a,
    left  :: UnbalancedSet {vv:a | vv < val},
    right :: UnbalancedSet {vv:a | vv > val}
  }
```

Red-Black Set

The red-black set implements a set using a red-black tree. This data structure is very similar to the binary search tree. The difference is that each node in the tree has a color field that is used to enforce two invariants. As stated in *Purely Functional Data Structures*, these are

> **Invariant 1.** No red node has a red child.
> **Invariant 2.** Every path from the root to an empty node contains the same number of black nodes

The invariants are called `RedInvariant` and the `BlackInvariant` in the predicates below. `RedInvariant` compares the color of a child to its parent to ensure that the child is not red when the parent is red. `BlackInvariant` compares the number of black nodes on the path to a leaf for two child trees to ensure these values are the same.

```
data Color = Red | Black
data RedBlackSet a =
  Empty | Tree {
    color   :: Color,
    rbval   :: a,
    rbleft  :: {v:RedBlackSet {vv:a | vv < rbval} | RedInvariant color v},
```

```
    rbright :: {v:RedBlackSet {vv:a | vv > rbval} | RedInvariant color v &&
                                                    BlackInvariant v rbleft}
  }
predicate RedInvariant C S     = (C == Red) ==> (getColor S /= Red)
predicate BlackInvariant S0 S1 = (numBlack S0) == (numBlack S1)
```

In additions to the full red-black set, it is useful to have a version of the data structure with a weakened version of the red invariant. The weak invariant is that at least one of the root node and the two children must be black. The full red invariant holds for the children. This invariant is a necessary precondition for one of the inputs to the rebalancing procedure and it is a constant postcondition on the output of the rebalancing procedure.

```
data WeakRedInvariant a = WeakRedInvariant {
  weakColor :: Color,
  weakVal   :: a,
  weakLeft  :: RedBlackSet {vv:a | vv < weakVal},
  weakRight :: {v:RedBlackSet {vv:a | vv > weakVal} |
    (weakColor /= Red ||
    (getColor weakLeft) /= Red ||
    (getColor v) /= Red) &&
    (numBlack v) == (numBlack weakLeft)}
}
```

Given a tree with only the weak red invariant, it can be necessary to describe the conditions when the full red invariant holds. For this purpose, there is a predicate `HasStrongRedInvariant`. This checks enforces the same property as the `RedInvariant` property but, it is a function over the entire tree rather than only the root node and one child.

```
predicate HasStrongRedInvariant Wri = (weakColor Wri) == Red ==>
                                      (getColor (weakLeft Wri) /= Red &&
                                       getColor (weakRight Wri) /= Red)
```

These data types and functions are enough to write the refinement type for the auxiliary insertion function. This function does the majority of the work during insertion. The inputs to the function are not specially refined. It takes and element to insert and a `RedBlackSet` to insert it into. The refinement on the return type is more interesting. Rather than returning a full `RedBlackSet`, a `WeakRedInvariant` is returned. This means that red invariant does not necessarily hold for the returned tree; however, a condition is provided when the return value does maintain the red invariant. This is that when the color of the original tree was not red, the red invariant will hold for the output.

```
rb_insert_aux :: forall a. Ord a =>
  x:a ->
  s:RedBlackSet a ->
  {v:WeakRedInvariant a | ((getColor s) /= Red ==> HasStrongRedInvariant v) &&
```

```
                                (weakNumBlack v) == (numBlack s) &&
                                (setElts v) == (Set_cup (Set_sng x) (setElts s))}
```

Since the auxiliary function does not return a full red-black tree, the primary insert function must perform some transformation to convert it into a `RedBlackSet`. Since the strong red invariant still applies to the sub-trees of a `WeakRedInvariant` structure, the strong red invariant can be established for the full tree changing the color of the root node to black. This does not invalidate the black invariant because it will uniformly add either one or zero to the number of black nodes on every path from the root node to a leaf node.

Heaps

The refinement types for the `Heap` typeclass would ideally track the size of a heap and provide some guarantee that the element returned by `findMin` is, in fact, the smallest element in the heap. The current refinements only protect against calling `findMin` and `deleteMin` on empty heaps Individual heap implementations introduce their own invariants for that ensure that the minimum element is kept at the top.

Due to issues working with LiquidHaskell, none of the heap implementations are actually instances of this typeclass. The functions of the typeclass are implemented by each heap and, the same refinement types are applied to the implementations but, I have not been able to able to write them as a formal instance of the `Heap` typeclass. The interface below is what should be implemented when this problem is resolved.

```
class Heap h where
  empty     :: forall a. Ord a =>
    {h:h a | 0 == hsize h}
  isEmpty   :: forall a. Ord a =>
    h:h a -> {v:Bool | v <=> (0 == hsize h)}
  insert    :: forall a. Ord a =>
    a -> h0:h a -> {h1:h a | (hsize h1) == (hsize h0) + 1}
  merge     :: forall a. Ord a =>
    h0:h a -> h1:h a -> {h2:h a | (hsize h2) == (hsize h0) + (hsize h1)}
  findMin   :: forall a. Ord a =>
    {h:h a | hsize h /= 0} -> a
  deleteMin :: forall a. Ord a =>
    {h0:h a | hsize h0 /= 0} -> {h1: h a | (hsize h1) == (hsize h0) - 1}
```

Sorted List Heap

The Sorted List Heap is a trivial implementation of a heap not take from *Purely Functional Data Structures* that keeps track of the smallest element

by maintaining a sorted list. Of course, this is not a particularly efficient implementation (it has linear time insertion rather than logarithmic) but, writing refinement types to force an ordered list is slightly easier than doing the same for any of the tree based heap implementations.

The property ensured for this data structure is that for list larger than the singleton list the head of the list is at least as small as the head of the tail. This property is then recursively checked for the tail of the list.

```
data SortedListHeap a =
  Nil | Cons {
    t :: SLH a,
    h :: {v:a | IsMin v t}
  }
predicate IsMin N H = (hsize H == 0) || (N <= hmin H)
```

Leftist Heap

The Leftist Heap is a more legitimate heap implementation that *is* included in book. The data structure is a binary tree where the element at the root node is at least as small as the elements at both of the children. The Leftist Heap also requires that the rank (length of the shortest path to a leaf node) of the right sub-heap is smaller than the rank of the left sub-heap. Both of these properties are encoded in the refinement type for the heaps constructor.

```
data LeftistHeap a =
  E | T
    (r     :: {v:Nat | v > 0})
    (left  :: LH a)
    (right :: {v:LH a | (rank v == r - 1) && (rank left >= rank v)})
    (val   :: {v:a | IsMin v left && IsMin v right} )
```

The implementation of `merge` for this heap is particularly interesting. LiquidHaskell was not able to verify the Leftist Heap merge function given in *Purely Functional Data Structures*. I believe that this was because LiquidHaskell could not show that minimum after merging two heaps must be the minimum of one of the input heaps. I was able to encode a variant of this fact into the type of `merge` in a way that enabled LiquidHaskell to check the function.

The new type of the function says that any element that is at least as small as the minimum of both input heaps must be at least as small as the minimum of the output heap. The way this is written is unnecessarily heavy - to encode universal quantification, a new parameter is added to the function that is not used anywhere inside the body of the function. I would like to further modify the type to remove the extra parameter.

```
merge_aux :: forall a. Ord a =>
  e:a -> h0:LH a -> h1:LH a ->
```

```
  {h2:LH a | (hsize h2 == hsize h0 + hsize h1) &&
             (IsMin e h0 ==> IsMin e h1 ==> IsMin e h2)}
```

Random-Access Lists.

A random-access list as presented in *Purely Functional Data Structures* is an extension of the usual cons-list that supports lookup and update functions like those you would expect to see for a traditional array. While for a queue it sufficed to know that the structure was not empty before retrieving the next element, a random access list must verify that a requested index is with the bounds of the list before any lookup or update operation. This requirement is encoded in the refinement types for these functions.

```
class measure rlen :: forall a. a -> {v:Int | v >= 0}
class RandomAccessList r where
  empty   :: forall a.
    {r:(r a) | 0 == rlen r}
  isEmpty :: forall a.
    r:(r a) -> {v:Bool | v <=> (0 == rlen r)}
  cons    :: forall a.
    a -> r0:r a -> {r1:r a | (rlen r1) == (rlen r0) + 1}
  head    :: forall a.
    {r:r a | rlen r /= 0} -> a
  tail    :: forall a.
    {r0:r a | rlen r0 /= 0} -> {r1:r a | (rlen r1) == (rlen r0) - 1}
  lookup  :: forall a.
    r:r a -> {i:Nat | i < rlen r} -> a
  update  :: forall a.
    r0:r a -> {i:Nat | i < rlen r0} -> a -> {r1:r a | (rlen r1) == (rlen r1)}
```

Simple Random-Access List

This implementation is not one given in the book. It is a very simple implementation that I wrote to practice using LiquidHaskell and to quickly check that the refinement types I wrote for the `RandomAccessList` type class were reasonable.

The data type used for this implementation is a wrapper around Haskells List type and the functions are implemented as operations on the internal list. No additional refinements are given to the constructor as there are no invariants that must be maintained.

```
data SimpleRandomAccessList a = SRAL [a]
```