Verified Binomial and Skew Heaps Using LiquidHaskell

Josh Cohen
University of Pennsylvania
joscoh@sas.upenn.edu

Palmer Paul
University of Pennsylvania
palmerpa@seas.upenn.edu

Abstract

We used LiquidHaskell to verify the structural invariants of binomial heaps and skew heaps and prove totality, termination, and functional correctness of all common heap operations on these structures. In this report, we describe the technical aspects of our project, challenges we faced, and our overall experience using LiquidHaskell. The code for our project is publicly available in the pzp1997/liquid-heaps GitHub repo.

1 Introduction and Overview of LiquidHaskell

LiquidHaskell is a project that extends Haskell's type system with refinement types. Refinements in LiquidHaskell are restricted to the quantifier-free theory of equality, uninterpreted functions, and linear arithmetic (QF-EUFLIA), a logic that can be efficiently decided by an SMT solver, such as Z3. The purported advantage of restricting the logic is that it makes proofs "extremely automatic." In theory, the programmer only needs to specify the properties of interest via refinement types and LiquidHaskell can deduce whether they hold for the given Haskell implementation. In comparison, verification tools that use richer logics, for instance dependelty-typed programming languages, often require explicit proofs.[3, Chapter 2]

In order to maintain full compatibility with Haskell, all features that exist only in LiquidHaskell are written using special Haskell comments that are interpreted by LiquidHaskell and ignored by conventional Haskell compilers. The syntax for these comments is {`-@` *some* LiquidHaskell *code...* `@-`}.

In the following subsections, we describe some of the fundamentals of LiquidHaskell that are relevant to our project. For a full explanation of these concepts and LiquidHaskell in general, see [3].

1.1 Refinement Types

A refinement type of the form $\{x : t \mid f(x)\}$ denotes a value of type t called x such that the predicate $f(x)$ holds. Refinements on the types of function arguments are analogous to the pre-conditions of the function and refinements on the return type of a function are analogous to the post-conditions of the function in Floyd-Hoare logic.[2]

1.2 Type Level Functions

In LiquidHaskell, there are three distinct approaches to lift functions to the type level so that they can be used in refinements: `measure`, `reflect`, and `inline`. We used all three for our project. Each of these approaches has its own set of restrictions governing its use and how it is interpreted by the logic. When attempting to lift a particular function to the type level, it is oftentimes unclear which of these approaches to use and even whether any of them will work until one tries each one. Through reading documentation and our own experience, we came up with the following working understanding of each approach.

Measure

Measures are total, unary functions that must be guaranteed to terminate, typically via structural recursion. Furthermore, measures must have exactly one pattern match case per data constructor and cannot use nested pattern matches, although in certain cases it is possible to get around this restriction (see § 5.3 for an example). Additionally, measures can only use primitive operations that belong to the underlying SMT theory and can only call type level functions.[3, 4]

Inline

Inline functions must be non-recursive and, like measures, can only use primitive operations and call type level functions. During compilation, LiquidHaskell replaces uses of an inline function in predicates with its body.[1]

Refinement Reflection

Reflection was added to LiquidHaskell in 2016. It allows the programmer to, "reflect the code implementing a user-defined function into the function's (output) refinement type." This allows for deeper specifications that depend on the implementation details of the function and not just the type of the function. When a reflected function is used in a predicate, LiquidHaskell will automatically unfold the definition of that function once. In our experience, functions marked `reflect` can call functions marked `reflect` or `measure` but not those marked `inline`.[7, 5]

1.3 Verification Errors

If LiquidHaskell cannot verify that a particular property holds, it declares the code unsafe and produces an accompanying error message. This error message includes the code where the property violation occurs and the properties LiquidHaskell was able to infer and those it was trying to prove.

The following example demonstrates the format of these errors and the processes we used to debug them. In practice, the messages are significantly more complex since there may be many more variables, each with their own refinement types. In this case, the error occurs in a helper function called `insert'` that is used in the `insert` and `merge` functions. The implementation of the `insert'` function uses another helper function called `link`, which merges two binomial trees of equal rank.

```haskell
{-@ link :: t1:Tree a -> {t2:Tree a | rank t2 = rank t1} ->
  {v:Tree a | treeElts v = Bag.union (treeElts t1) (treeElts t2) && rank v = rank t1 + 1}
@-}
-- implementation of link omitted for brevity

{-@ insert' :: t:Tree a -> ts:[Tree a] ->
  {v:[Tree a] | treeListElts v = Bag.union (treeElts t) (treeListElts ts) }
@-}
insert' :: Ord a => Tree a -> [Tree a] -> [Tree a]
insert' t [] = [t]
insert' t ts@(t':ts')
  | rank t < rank t' = t : ts
  | otherwise        = insert' (link t t') ts'
```

According to the error message, LiquidHaskell cannot prove that the ranks of trees `t` and `t'` are equal and it is therefore unsafe to call `link`.

```
**** RESULT: UNSAFE ************************************************************

 ./liquid-heaps/BinomialHeap.hs:228:40-41: Error: Liquid Type Mismatch

 228 |    | otherwise          = insert' (link t t') ts'
```

```
                                                   ^^
  Inferred type
    VV : {v : (Data.Heap.Binominal.Tree a) | v == t'}

  not a subtype of Required type
    VV : {VV : (Data.Heap.Binominal.Tree a) | rank VV == rank t}

  In Context
    t : (Data.Heap.Binominal.Tree a)
    t' : (Data.Heap.Binominal.Tree a)
```

If we look back at the implementation of the `insert'` function we can see that we included an explicit case for when `rank t < rank t'` but we forgot to account for when `rank t > rank t'`. Adding this case allows LiquidHaskell to conclude that the `otherwise` branch is only taken (and therefore `link` is only called) when `rank t == rank t'`.

2 Heap Interface and Specification

The heap interface included here describes the heap operations we verified for both binomial heaps and skew heaps. The formal specification for each operation is provided in the form of a LiquidHaskell refinement type. Note that a Bag is a multiset, and is used to describe the elements in the heap.

```
measure size
size :: Heap a -> Nat

measure elts
elts :: Heap a -> Bag a

measure listElts
listElts :: [a] -> Bag a

empty :: {v:Heap a | elts v = Bag.empty}
singleton :: x:a -> {v:Heap a | elts v = Bag.put x Bag.empty}
insert :: x:a -> h:Heap a -> {v:Heap a | elts v = Bag.put x (elts h)}
merge :: h1:Heap a -> h2:Heap a -> {v:Heap a | elts v = Bag.union (elts h1) (elts h2)}

deleteMin :: {h:Heap a | 0 < size h} ->
    {v:(a, Heap {x:a | (fst v) <= x}) | Bag.put (fst v) (elts (snd v)) = elts h}

null :: h:Heap a -> {v:Bool | v <=> size h = 0}
(==) :: h1:Heap a -> h2:Heap a -> {v:Bool | v <=> elts h1 = elts h2}

fromList :: xs:[a] -> {v:Heap a | elts v = listElts xs}
toList :: h:Heap a -> {v:[a] | listElts v = elts h}
toSortedList :: h:Heap a -> {v:IncrList a | listElts v = elts h}
heapSort :: xs:[a] -> {v:IncrList a | listElts v = listElts xs}
```

In the final two operations, `toSortedList` and `heapSort`, we make use of the `IncrList a` type to specify that the returned list is sorted in increasing order. Typically, we would need universal quantification to specify the sorted list property, however this is not possible in LiquidHaskell since predicates are quantifier free. In order to get around this constraint, we make use a LiquidHaskell feature called abstract refinement types to write the `IncrList a` type.[6]

```
type IncrList a = [a]<{\xi xj -> xi <= xj}>
```

The way to interpret the `IncrList a` type is that it refines the built-in Haskell list type such that for every pair of elements x_i and x_j where x_i comes before x_j in the list, the predicate $x_i \leq x_j$ holds.

3 Heap Invariants

3.1 Structural Properties of Skew Heaps

A skew heap is an extremely simple heap implementation in the form of a binary tree. It can described recursively as follows:

1. A heap with a single element (no children) is a skew heap.
2. The result of merging two skew heaps (using a special merge algorithm) is a skew heap.[1]

Somewhat surprisingly, although there are no structural invariants, skew heaps have good amortized performance: all operations run in amortized time $\mathcal{O}(\log n)$.

Since there are no structural invariants, the LiquidHaskell type is quite simple:

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

3.2 Structural Properties of Binomial Trees and Heaps

Binomial heaps, on the other hand, are significantly more complex. They are built from binomial trees, which contain a value, an integer called the rank, and a list of subtrees, obeying the following invariants:

1. $rank = 0$ and the tree consists of a single node.
2. Let r be the rank. Then the subtrees, in order, are valid binomial trees and have ranks $r-1, r-2, ..., 0$. We will refer to this as the ranks of subtrees invariant.

A binomial heap is a collection of binominal trees of distinct ranks.[2]

The binomial tree invariants ensure that all operations take place in worst-case $\mathcal{O}(\log n)$ time. Notably, this includes the merge operation, making this asymptotically more efficient than a binary heap and much more efficient in the worst case than a skew heap.

Encoding the binomial tree invariants in LiquidHaskell was one of the most challenging parts of the project. We initially doubted whether it would even be feasible to encode the ranks of subtrees invariant in a way that LiquidHaskell can use to prove things, since only very simple and structurally recursive functions can be lifted to the type level (see § 1.2). After many failed attempts that can best be described as wrestling with the LiquidHaskell verifier, we were able to succeed by encoding the following properties about a tree's rank.

Given a binomial tree of rank r,

1. The length of the list of subtrees is r.
2. The rank of each subtree is strictly less than r.

[1]For a short explanation of the merge operation, see https://courses.cs.washington.edu/courses/cse326/08sp/lectures/markup/06-skew-heaps-markup.pdf

[2]Unlike skew heaps, binomial trees have a very rigid structure; in fact, it is possible to prove that each of the following hold for every binomial tree of rank r: (1) the height of the tree is r, (2) there are 2^r elements in the tree, (3) there are exactly $\binom{r}{i}$ nodes at depth i for $i \in \{0, \ldots, r\}$ (hence the name "binomial tree").

3. The rank of a tree is always a nonnegative integer.
4. The list of subtrees is strictly decreasing by rank (we again made use of abstract refinement types to encode this property).

A simple argument shows that the 4 above conditions imply the binomial heap invariants: from the first three properties we know we have a list of r integers in $\{0, \ldots, r-1\}$. The only way for such a list to be strictly decreasing is if the ranks of the subtrees in order are $r-1, \ldots, 0$. The other direction, that the binomial heap invariants imply these conditions, is trivial.

However, this sort of argument uses subtle logical deductions that are beyond the scope of LiquidHaskell's capabilities. Instead of proving the ranks of subtrees invariant, we proved that the above 4 conditions imply a slightly weaker property: given a node of rank r the first tree in its subtree list has rank $r-1$. The weaker property suffices for the purpose of verifying the rest of the code, since we wrote all the functions that recurse over the structure of a binomial tree in a special way such that we only inspect the first subtree of a node in each recursive step. Proving this weaker property was still extremely difficult, see § 5.2 for more details.

Now, given these conditions, we can encode the invariants as LiquidHaskell types. Proving that these invariants were maintained through the various heap operations was not always straightforward; see § 4 for some of the methods we used.

Note that we include a size field in order to be consistent with other Haskell heap representations and get the size of the heap in constant time; we also prove that it is actually the size of the heap:

```haskell
data Tree [rank] a =
    Node
        { root :: a
        , rank :: Nat
        , subtrees :: {ts:[{t:Tree a | rank > treeRank t}]<{\ti tj -> treeRank ti > treeRank tj}>
            | len ts = rank}
        , treeSize :: {v:Pos | v = 1 + treeListSize subtrees}
        }

{-@ measure treeListSize @-}
{-@ treeListSize :: xs:[Tree a] -> {v:Nat | (len xs <= v) && (v = 0 <=> len xs = 0)} @-}
treeListSize :: [Tree a] -> Int
treeListSize [] = 0
treeListSize (x:xs) = treeSize x + treeListSize xs

data Heap a = Heap { unheap :: [Tree a] }
```

In the first line of the code block above, we specify the rank as a default termination metric for binomial trees. This is important for proving the termination of heap operations that use structural recursion on binomial trees. More specifically, it guarantees that such functions will only be called on binomial trees of strictly smaller rank.

3.3 Min-Heap Property

The structural invariants above only guarantee that the data we are looking at is the correct shape. We have yet to specify any properties about the elements themselves, in particular the min-heap property: that the value at a `Node` is less than or equal to all the values in every subtree of that node. This property holds of both skew heaps and binomial trees.

We can do this in LiquidHaskell by editing the `Skew a` (resp., `Tree a`) data definition to replace every recursive reference to the type with the refined type `Skew {v:a | v >= root}` (resp., `Tree {v:a | v >= root}`) where `root` refers to the root field of the `Node`.

4 Approaches and Techniques

Since LiquidHaskell uses an SMT solver to solve the obligations resulting from the refinement types (we used Z3), the verification process is designed to be highly automatic. This has benefits and drawbacks—sometimes complex goals will succeed with no work required while other times seemingly trivial goals will fail with little information about what went wrong. In general, when we needed to help the solver, we used the following methods:

4.1 Assertions

LiquidHaskell provides an easy way to insert assertions anywhere within a function. This takes the form of the following function[3]:

```
{-@ assert :: TT -> a -> a @-}
assert :: Bool -> a -> a
assert _ a = a
```

Essentially, this forces LiquidHaskell to check during compilation that the type of the boolean we pass in to assert is true (`TT`). In addition, due to Haskell's laziness, there is no runtime cost for this. Using assertions, we can see what intermediate goals LiquidHaskell is able to infer and what it cannot. In this way, we could write a pseudo-proof of the property we wanted by asserting some steps, seeing what LiquidHaskell was missing, and then figuring out why it could not infer the desired goal.

Assertions can also be used to get additional information into the context in order to help LiquidHaskell solve its goals. An example of this is the `treeElts` function we include in § 5.2.

While we found assertions to be for the most part helpful, we found a few instances where assertions could make error messages less informative by changing the context information. Additionally, we noticed that having many assertions in a single function makes verification considerably slower.

4.2 Lemmas

Often, LiquidHaskell could not solve a goal since the type information available was not specific enough. To overcome this, we wrote what we called "lemmas," which were almost always identity functions that have a more specific return type. Then, we can use these lemmas to transform our input into a more specific type which has the information we need.

For example, when verifying the `link` function, which combines two binomial trees of equal rank into a new binomial tree, we needed to know that every element in a tree is bounded by the root. The invariants state that only for the subtrees, so we needed the following lemma:

```
{-@ type AtLeast a X = {n:a | X <= n} @-}
{-@ type AtLeastTree a X = Tree (AtLeast a X) @-}

{-@ treeAtLeastRoot :: t:Tree a -> {v:AtLeastTree a (root t) |  v = t} @-}
treeAtLeastRoot :: Tree a -> Tree a
treeAtLeastRoot (Node x r ts sz) = Node x r ts sz
```

We believe LiquidHaskell has a more elegant mechanism for expressing properties of this sort called "self-invariants," but we struggled to get this feature to work for our use cases and did not pursue the matter further since our `treeAtLeastRoot` lemma got the job done.

LiquidHaskell cannot infer this more specific type within the context of another function, but by unfolding the structure of the data in a separate lemma, there is enough information in the context for LiquidHaskell to infer the more specific type.

[3]The LiquidHaskell prelude includes a similar function called `liquidAssert`. The reason we created our own version was so that we could lift it to the type level and use it in type level functions as seen in § 5.2.

4.3 Induction

In contrast to proof assistants like Coq, we cannot manually do induction to introduce new proof obligations. However, whenever we are verifying a recursive function, once we show that the input to the recursive call satisfies all preconditions (which is often non-trivial), we get an effective induction hypothesis, since the result of the recursive call has the refined type. For an example, see our discussion of `treeElts` and `rankOfFirstTree` in § 5.2.

4.4 Re-implementing Library Functions

Haskell has a large set of libraries; however, other than the libraries LiquidHaskell provides, none of it uses refinement types. Because of this, we needed to implement some of the library functions ourselves in order to give enough information in the return type. For example, we needed a more specific list append function, which states that the elements in the resulting list are the elements in both of the inputs (as well as the natural property about length):

```
{-@ appendPreservingListElts :: xs:[a] -> ys:[a] ->
    {v:[a] | listElts v = Bag.union (listElts xs) (listElts ys) && len v = len xs + len ys} @-}
appendPreservingListElts :: [a] -> [a] -> [a]
appendPreservingListElts [] ys = ys
appendPreservingListElts (x:xs) ys = x : appendPreservingListElts xs ys
```

5 Challenges

The skew heap, due to its simple structure and lack of invariants, presented very few challenges. Most of the proofs of desired properties went straight through. For the binomial heap, on the other hand, the situation was very different. We often needed to come up with clever implementations of functions, create additional lemmas, and use other tricks mentioned in § 4 to get LiquidHaskell to verify our code.

The primary structural difference between binomial heaps and skew heaps, which explains the vastly different levels of difficulty verifying the structures, is that a skew heap node has exactly two subtrees while a binomial tree node has a finite but unbounded number of subtrees. In other words, binomial trees resemble rose trees while skew heaps resemble binary trees. Functions that recurse over the structure of rose trees pose a significant challenge for LiquidHaskell's termination checker, since they must utilize recursion in two "directions": across the list of subtrees, and down into each subtree itself.

We felt this issue most acutely when implementing the `treeElts` measure, which collects the elements of a binomial tree into a multiset. This measure was crucial for being able to encode predicates about the elements of a binomial heap (see § 2 for examples). Implementing `treeElts` simultaneously dealt with checking termination, the encoding of the binomial tree invariants, and lifting non-trivial functions to the type level.

5.1 Proving Termination of `treeElts`

We tried implementing the `treeElts` function several different ways (with `map`, `foldr`/`foldl`, a separate recursive function, mutual recursion with the `treeListElts` measure, etc.) none of which satisfied LiquidHaskell's termination checker. After reading about how LiquidHaskell checks for termination, we were able to come up with the following successful implementation. The key observation that inspired this implementation is that if we remove the first subtree of a node of rank r, the "residual node" consisting of the original root and the remaining subtrees is a valid binomial tree of rank $r - 1$.

```
{-@ measure treeElts @-}
{-@ treeElts :: Tree a -> Bag a} @-}
{-@ treeElts :: t:Tree a -> {v:Bag a | v = Bag.put (root t) (treeListElts (subtrees t))} @-}
treeElts :: Ord a => Tree a -> Bag a
treeElts (Node x _ [] _) = Bag.put x Bag.empty
treeElts (Node x r tts@(_:ts) sz) =
    let residual = Node x (r - 1) ts (sz - (size t)) in
    Bag.union (treeElts t) (treeElts residual)

{-@ measure treeListElts @-}
treeListElts :: Ord a => [Tree a] -> Bag a
treeListElts [] = Bag.empty
treeListElts (t:ts) = Bag.union (treeElts t) (treeListElts ts)
```

5.2 Ensuring `residual` Satisfies the Binomial Tree Invariants

Now that our termination problems were behind us, we had to show that `residual` is a valid binomial tree (since `treeElts` requires an invariant-respecting tree as input). Most importantly, we needed to know that when we remove the first subtree, the resulting tree has rank $r - 1$. Proving this turned out to be very challenging, and it took many attempts at encoding the invariants before we could prove this property. It is not enough to require that, for instance, the first subtree has rank $r-1$, since then we have to show that the first subtree of ts has rank $r-2$, and so on. Using more complex functions that enable us to get elements from the list (to encode the property that the ith subtree has rank $r-i$) is not helpful because LiquidHaskell cannot unfold these functions effectively at the type level. However, using the invariants described in § 3.2, we were able to prove the property we wanted using induction, in the following two lemmas:

```
{-@ measure head @-}
{-@ head :: {xs:[a] | len xs > 0} -> a @-}
head (x:_) = x

{-@ measure tail @-}
{-@ tail :: {xs:[a] | len xs > 0} -> [a] @-}
tail (_:xs) = xs

{-@ reflect rankOfTailDecreases @-}
{-@ rankOfTailDecreases :: {ts:[{t:Tree a | len ts > treeRank t}]<{\ti tj -> treeRank ti > treeRank tj}
| len ts > 0} ->
{v:[{t:Tree a | len ts - 1 > treeRank t}]<{\ti tj -> treeRank ti > treeRank tj}> | v = tail ts} @-}
rankOfTailDecreases :: [Tree a] -> [Tree a]
rankOfTailDecreases (_:ts) = ts

{-@ reflect rankOfFirstTree @-}
{-@ rankOfFirstTree :: {ts:[{t:Tree a | len ts > treeRank t}]<{\ti tj -> treeRank ti > treeRank tj}>
| len ts >= 1} ->
{v:Tree a | treeRank v = len ts - 1 && v = head ts} @-}
rankOfFirstTree ::  [Tree a] -> Tree a
rankOfFirstTree [t] = t
rankOfFirstTree ts@(t:_:_) = rankOfFirstTree (lemma ts) `seq` t
```

Informally, the proof proceeds as follows:

We want to prove `rankOfFirstTree`, which says that, given a nonempty list of binomial trees whose ranks are strictly bounded above by the length of the list and whose ranks are strictly decreasing, we can return a binomial tree which is the head of the input list and which has rank $len\ ts - 1$. (Note that, by replacing $len\ ts$ with r as in the invariants, this is exactly what we need to show.) We would like to proceed by induction, via the following proof:

1. In the base case, a singleton list, $ts = [t]$, $len\ ts = 1$, and $treeRank\ t < 1$, so $treeRank\ t = 0 = 1 - 1$, which is what we want to show.
2. Now suppose that we have $ts = t :: t'$, where t' is nonempty. By induction, the head of t' has rank $(len\ ts) - 2$. Since ts is strictly decreasing by rank and bounded by $len\ ts$, $(len\ ts) - 2 < rank\ t < len\ ts$,

hence, $rank\ t = len\ ts - 1$.

Case 1 proceeds as expected. However, in case 2, in order to use induction on t', we need to know that the rank of every tree in t' is bounded by the length of t', or, for all $tr \in t', rank\ tr < len\ ts - 1$. All we know from the preconditions is that $rank\ tr < len\ ts$.

In order to resolve this, we use the second lemma, called `rankOfTailDecreases`. It states the following: suppose we have a nonempty list of trees in decreasing order of rank whose ranks are all strictly bounded above by $len\ ts$. Then, we can return the tail of the list, which has the property that all trees are bounded above by $len\ ts - 1$. Though this seems to be very similar to what we actually want to prove in `rankOfFirstTree`, it can be stated and proved much more generally. Informally, this lemma says that, if we have a set of integers bounded by n and we remove the largest, the set is bounded by $n - 1$ (we initially proved it in this more general form, but that causes issues when lifting the lemma to the type level).

With this lemma, we can now satisfy the preconditions for the recursive call and use induction, proving the claim.

This example demonstrates how even seemingly simple assertions can be quite complex to state and prove in LiquidHaskell, and showcases how we could use lemmas and induction in order to prove theorems. With these lemmas, we could finally prove in `treeElts` that `residual` was a valid binomial tree, leading to the following definition (which requires the assertion about `treeRank t` in order to verify):

```haskell
{-@ measure treeElts @-}
{-@ treeElts :: t:Tree a -> {v:Bag a | v = Bag.put (root t) (treeListElts (subtrees t))} @-}
treeElts :: Ord a => Tree a -> Bag a
treeElts (Node x _ [] _) = Bag.put x Bag.empty
treeElts (Node x r tts@(_:ts) sz) =
  let t = rankOfFirstTree tts in
  assert (treeRank t == r - 1) $
  let residual = Node x (r - 1) ts (sz - (size t)) in
  Bag.union (treeElts t) (treeElts residual)
```

5.3 Making `treeElts` a Measure

In order to be able to use `treeElts` to specify predicates about the elements of a binomial tree, we need to lift it to the type level. Unfortunately, the implementation of `treeElts` above does not quite satisfy the criteria for being a measure, as described in § 1.2, since `treeElts` has two pattern match cases for the `Node` data constructor both of which use a nested pattern match on the subtrees field. We described our conundrum on the LiquidHaskell Slack and received a prompt response from Ranjit Jhala with a solution. We could make `treeElts` a measure just by including the following short snippet of code at the top of the binomial heap file.

```haskell
-- Automatically generate singleton types for data constructors
{-@ LIQUID "--exactdc" @-}
-- Disable ADTs (only used with exactDC)
{-@ LIQUID "--no-adt" @-}
```

This snippet enables a pair of LiquidHaskell options that, as we understand it, relax the measure requirements. We could not find any documentation about either of these options. Furthermore, it is not apparent to us why they are disabled by default, since we noticed no downsides to using them.

5.4 Other Applications

Once we established this general structure for writing functions and lemmas over binomial trees, we could apply it to other properties. For instance, we could verify the property that a valid binomial tree of rank r has 2^r elements (this follows from the binomial tree invariants). We were able to prove this fact in LiquidHaskell via the following lemma (which mirrors closely the structure of `treeElts` above):

```
{-@ reflect pow2 @-}
{-@ pow2 :: Nat -> Pos @-}
pow2 :: Int -> Int
pow2 0 = 1
pow2 n = let acc = pow2 (n - 1) in acc + acc


{-@ pow2Lemma :: t:Tree a ->
    {v:Tree a | size v = pow2 (rank v) && rank v = rank t && treeElts v = treeElts t} @-}
pow2Lemma :: Ord a => Tree a -> Tree a
pow2Lemma t@(Node _ _ [] _) = assert (pow2 0 == 1) $ t
pow2Lemma (Node x r tts@(_:ts) sz) =
  let t = rankOfFirstTree tts in
  let residual = Node x (r - 1) ts (sz - (size t)) in
  assert (treeRank t == r - 1) $
  assert (pow2 r == pow2 (r - 1) + pow2 (r - 1)) $
  link (pow2Lemma t) (pow2Lemma residual)
```

Informally, the lemma proves this property by induction. We construct a `residual` tree in the same way as in `treeElts`, and for the same reason, we know it has rank $r - 1$. Then, when we combine the two trees, both of rank $r - 1$, by induction they each have size 2^{r-1}, giving us a total size of 2^r. Thus, this general method of writing recursive functions over binomial trees is quite powerful, and we used it in several places throughout the code.

6 Bugs Reported

In the course of our project, we ran into one bug that we believe is caused by the interplay of abstract refinement types with certain kinds of measures. The bug breaks some internal invariant of LiquidHaskell causing it to error even on trivial assertions. We reported the bug in the LiquidHaskell Slack and opened a GitHub issue. Luckily, we were able to think of a workaround that involved changing the order of fields in the binomial tree type.

7 Related Work

To the best of our knowledge, our project represents the most complete verification of binomial heaps in LiquidHaskell, as no existing project encodes the full invariants of a binomial tree or specifies full functional correctness of the heap API. In this section, we describe other efforts to verify binomial heaps in Haskell and more generally, other data structure verification projects in LiquidHaskell.

The fpsyd-liquid-haskell project uses LiquidHaskell to verify that the invariant that a binomial tree node's rank is equal to the length of its subtrees list is preserved for the `singleton` heap operation and the `link` helper function. Thus, it does not reason about the rest of the invariants, the other functions in the heap API, and functional correctness.

The liquid-structures project verifies various data structures from Okasaki's "Purely Functional Data Structures" using LiquidHaskell, including leftist heaps and "sorted list heaps." Leftist heaps use a binary tree structure, so verifying them is of a similar level of difficulty to verifying skew heaps. While the heap specification used by this project encodes how the heap operations affect a heap's size, it does not reason about the heap's elements and is missing some invariants for the included data structures.

A blog post by Donnacha Oisín Kidney uses Haskell with the DataKinds, TypeFamilies, GADTs, and TypeInType extensions to verify the structural correctness of several heaps, including binomial heaps and skew heaps. The blog post does not attempt to prove totality, termination and functional correctness of the heap operations on those structures. Most importantly, this verification uses dependent types rather than LiquidHaskell.

While unrelated to the topic of heaps, [3, Chapter 11] uses LiquidHaskell to verify that AVL trees satisfy the binary search tree invariants, AVL balancing invariants, and specifies how the BST operations affect the elements of the tree.

8 Conclusions

Through our experience verifying both skew heaps and binomial heaps, we saw firsthand how LiquidHaskell excels at proving certain kinds of properties and how trying to prove properties outside of LiquidHaskell's wheelhouse causes the effort required for verification to skyrocket. In particular, LiquidHaskell is very good at dealing with simple functions (for example, list head/tail), structurally recursive functions over simple recursive data structures (such as lists and binary trees), and arithmetic and set/multiset properties. However, when dealing with more complicated data structures, invariants, and functions, the amount of work and expertise needed grows very rapidly.

Although LiquidHaskell has impressive proof automation capabilities, we found that for verifying nontrivial properties, it is still necessary for the programmer to have at least an outline of the proof in their head or on paper. For this reason, programming in LiquidHaskell requires some amount of mathematical maturity.

Beyond the knowledge barrier, proving things in LiquidHaskell is quite time consuming. It took us approximately 70 hours of pair programming to verify the code for this project, including time spent to familiarize ourselves with LiquidHaskell and to read documentation. We anticipate that for the vast majority of Haskell code, the cost (in terms of hours of labor) of verifying code with LiquidHaskell is too high in relation to the benefit.

In addition, one of the supposed advantages of using LiquidHaskell over a theorem prover such as Coq or Agda is that one can add refinement types to an existing codebase and can write verified code without having to rewrite everything in a different language. However, we found that to verify nontrivial properties of more complex code, we needed to rewrite some functions and add additional lemmas and assertions. While it is possible that some of these modifications could have been avoided with a different approach to verification, it is clear that the exact implementation of a function determines whether or how verification will succeed. For instance, we found that adding a refinement type to the usual recursive list `reverse` function did not succeed, while the same property held for a tail-recursive `reverse` function without modifications. This means that, for best results, programs have to be written with LiquidHaskell in mind, somewhat negating the benefit of using refinement types.

We enjoyed our experience with LiquidHaskell for this project and could see ourselves using it in the future. We learned a huge amount about refinement types, automated verification, and type-level programming. LiquidHaskell is an impressive tool that can handle a wide array of use cases, but it is definitely not for everyone.

Acknowledgements

We would like to thank Ranjit Jhala, Niki Vazou, and Rose Kunkel for answering our questions in the LiquidHaskell Slack channel about lifting `treeElts` to the type level and using multisets in refinement predicates. We would also like to think Rajeev Alur for vetting our project proposal and teaching us about computer-aided verification this semester.

We used the kazu-yamamoto/llrbtree library as a reference for the Haskell implementations of skew heaps and binomial heaps in our project.

References

[1] URL: https://github.com/ucsd-progsys/liquidhaskell. (accessed: 04.13.2020).

[2] Ranjit Jhala. *Liquid Types vs. Floyd-Hoare Logic*. URL: https://ucsd-progsys.github.io/liquidhaskell-blog/2019/10/20/why-types.lhs/. (accessed: 04.13.2020).

[3] Ranjit Jhala, Eric Seidel, and Niki Vazou. *Programming With Refinement Types. An Introduction to LiquidHaskell*. 2017. URL: https://ucsd-progsys.github.io/liquidhaskell-tutorial/book.pdf.

[4] Ricardo Peña. "An Introduction to Liquid Haskell". In: *Proceedings XVI Jornadas sobre Programación y Lenguajes, PROLE 2016, Salamanca, Spain, 14-16th September 2016*. Ed. by Alicia Villanueva. Vol. 237. EPTCS. 2016, pp. 68–80. DOI: 10.4204/EPTCS.237.5. URL: https://doi.org/10.4204/EPTCS.237.5.

[5] Niki Vazou. *Haskell as a Theorem Prover*. URL: https://ucsd-progsys.github.io/liquidhaskell-blog/2016/09/18/refinement-reflection.lhs/. (accessed: 04.13.2020).

[6] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. "Abstract Refinement Types". In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 209–228. ISBN: 978-3-642-37036-6.

[7] Niki Vazou et al. "Refinement reflection: complete verification with SMT". In: *Proceedings of the ACM on Programming Languages* 2.POPL (Jan. 2018), pp. 1–31. ISSN: 2475-1421. DOI: 10.1145/3158141. URL: http://dx.doi.org/10.1145/3158141.