Verifying the Functional Correctness of Braun Trees with LiquidHaskell

Felipe de León[1], Alberto Pardo[1], and Marcos Viera[1]

Instituto de Computación
Universidad de la República Montevideo Uruguay

Abstract. LiquidHaskell is a refinement type system that allows integration of formal verification to software development in the form of type annotations over Haskell data structures and functions. In this paper, we explore the capabilities of the tool by focusing on balanced trees, and more specifically on Braun trees, a commonly used structure for representing functional arrays. We formalize some properties of balanced trees, concerning the relationship between their height and number of nodes. On top of this properties, we verify the invariants of functional and flexible arrays, resulting in an implementation that encodes most of the pre-conditions and post-conditions that preserve such invariants.

Keywords: LiquidHaskell, Verification, BraunTrees, Functional Arrays

1 Introduction

Formal verification has long promised increased software reliability, but its integration into mainstream programming practices remains limited due to complexity, verbosity, and steep learning curves. Since its introduction, LiquidHaskell (LH) [14] has been advertised as a lightweight approach to integrate formal verification into Haskell, offering a balance between expressiveness, low verbosity, and practical performance.

The aim of this work is to experiment with the main formalization and verification features of LH, applying them to a non-trivial example. In this sense, we focus on the formalization of functional arrays by means of their implementation in terms of Braun trees [3], a class of balanced binary trees with a well-defined structural property and efficient indexing operations. Due to their regular shape, Braun trees serve as an excellent benchmark to test the formal verification capabilities of LH. Braun trees are an interesting use case, since they are simple enough to admit elegant specifications, yet rich enough to expose the limits and power of current verification tools.

Our development is strongly guided by the work of Nipkow and Sewell [7], who present a comprehensive formalization of Braun trees in Isabelle/HOL. We replicate their verification effort, adapting the invariant definitions and their verification to the realm of LH.

In summary, we make the following contributions: (i) we formalize balanced binary trees in LH and prove some of their logarithmic properties; (ii) we formalize functional and flexible arrays implemented using Braun trees; and (iii) we use

Braun trees as a use case to explore recent LH features for theorem proving and refinement type reasoning, comparing them with those of other theorem provers.

The rest of the paper is organized as follows. In Section 2 we briefly describe the main features of LH. Then, in Section 3, we prove some logarithmic properties that balanced binary trees satisfy. On top of these properties, in Section 4, we prove the balance property of Braun trees and verify a series of invariants for operations on functional arrays. In Section 4.3 we verify flexible arrays, implemented in terms of Braun trees. Finally, we discuss related work in Section 5 and present some conclusions in Section 6.

The complete implementation, including all definitions, operations, and mechanized proofs, is available at `https://gitlab.fing.edu.uy/felipe.de.leon/brauntrees`.

2 LiquidHaskell

LH [4] is a refinement type system that allows the specification of invariants in the form of type annotations over Haskell type definitions and function types. Those invariants are given by logical predicates that can be regarded as Haskell Boolean expressions which can be verified by an SMT solver. For example, we can define a safe division operation by declaring a pre-condition that restricts the divisor to be non-zero:

```
{-@ safe_div :: Int -> d : { Int | d != 0 } -> Int @-}
safe_div :: Int -> Int -> Int
safe_div x y = div x y
```

By annotating the non-zero condition in the type of the second argument, we ensure that, when compiled, LH checks that in our code there is no case where `safe_div` is called with zero as divisor. If a function is invoked with an argument that is incompatible with its specification (e.g., a divisor that may be zero), LiquidHaskell reports the program as unsafe. In this case, the SMT solver identifies a counterexample to the specification, and compilation fails with an error message.

2.1 Promoting Functions

LH does not directly interpret Haskell functions within its refinement logic; functions are treated as black boxes. In order to reason about their behaviour, one can promote them to refinement logic using specific directives provided by LH.

Inline. The `inline` directive allows the promotion of non-recursive functions whose body is composed by other already-promoted functions. It is mostly used to promote simple predicates. An example is the following function that checks for non-zero values:

```
{-@ inline notZero @-}
notZero :: Int -> Bool
notZero x = x != 0
```

Once promoted, we can call these function inside refinements. For example, we can rewrite the signature of `safe_div` as:

```
{-@ safe_div :: Int -> d : { Int | notZero d } -> Int @-}
```

Measure. Functions promoted with the `measure` [13] directive must be structurally recursive on a decreasing argument and accept only a single parameter. In this case LH internally generates a refinement type where the promoted function definition is attached to the data constructor of the type of the parameter. If multiple measures are defined, they are merged into a single constructor.

For instance, we can define a function `allEven`, to check if all the elements of a list of integers are even, and promote it using `measure`:

```
{-@ measure allEven @-}
allEven :: [Int] -> Bool
allEven []     = True
allEven (x:xs) = even x && allEven xs
```

Having this function promoted, we can now use it to define e.g. the domain of lists of even numbers:

```
{-@ lstEvens :: xs : { [Int] | allEven xs } @-}
lstEvens = [2,4,6,8]
```

LH will verify that the list `lstEvens` indeed contains only even numbers.

The `type` directive can be used to define a named refinement type, allowing reusable and more readable specifications. Thus, if we define:

```
{-@ type EvenList = { xs : [Int] | allEven xs } @-}
```

the annotation for `lstEvens` can be simplified to:

```
{-@ lstEvens :: EvenList @-}
```

Reflection The `reflect` directive comes with slightly fewer limitations than the previous two. When a function is promoted with `reflect`, LH introduces an uninterpreted version of it in the refinement logic and embeds its body for use in logical reasoning. Once reflected, a function can be unfolded in logic formulas, allowing it to participate in more expressive proofs and theorem statements.

Unlike measures, reflected functions can have multiple parameters and can call other reflected functions but their will be require to terminate. Also different from measures, reflected functions do not bind values to constructors, but add constraints to the result of the functions. For this reason, if we try to promote `allEven` using reflection, then LH is unable to check that a given list contains only even numbers unless that is explicitly verified elsewhere.

Reflected functions on their own are less powerful than measures. However, Vazou [11] developed a library of proof combinators, which allow the use of LH as a theorem prover. The library includes the following combinators: `Proof`, an alias for the unit type `()` indicating we are proving a theorem; `QED`, declares that a proof is ready; `(***)`, marks the end of a proof before `QED`; `(===)`, equality between expressions in an equational proof; `?`, the "because" operator, used to justify steps in proofs; and `&&&`, an AND operator that allows the combination of subproofs, e.g. when proofs are divided into cases.

For instance, using some of these combinators we can write a proof stating that a list contains only even elements:

```
{-@ lstEvens_prf :: {allEven [2,4,6,8] == True} @-}
lstEvens_prf :: Proof
lstEvens_prf = allEven [2,4,6,8]
             === (even 2 && allEven [4,6,8])
```

```
               === (even 4 && allEven [6,8])
               === (even 6 && allEven [8])
               === (even 8 && allEven []) *** QED
```

Every time it is applied to an even number, `even` returns `True`. Since we are using the Boolean `&&` operator, we can eliminate the intermediate `True` values in the proof, allowing us to simplify step by step until we reach the final result. The rest of the combinators will be used later, when we show more complex proofs.

PLE. Proof by Logical Evaluation (PLE) [11,12] is a feature of LH that adds the following steps during type checking: (i) transforms every function into its reflected form; (ii) unfolds the reflected functions; and (iii) repeats until a fixpoint is reached.

When PLE is enabled, the earlier `allEven` example can be automatically verified without having to write the proof for `lstEvens_prf`; i.e., we can either define `lstEvens` as in the measure example or define `lstEvens_prf` just as `()`.

As we will see later, while PLE is very effective for simple equational reasoning and inductive properties, it cannot always handle complex recursion, higher-order functions, or proofs that require more sophisticated case analysis or user guidance.

3 Balanced Binary Trees

Balanced trees are a very common structure to formalize, as their structural constraints make them a good case study for verification tools. Previous work in LH regarding balanced trees focuses on two subtypes of balanced trees AVL trees [10] and red-black trees [14]. In contrast, in this work we adopt a more general notion of balance and study the logarithmic properties that these structures satisfy.

The representation of balanced binary trees in LH starts with the definition of (parameterized) binary trees:

```
data Tree a  = Node a (Tree a) (Tree a) | Nil
```

Following [7], we say that a tree is balanced if the absolute difference between its maximum and minimum height is at most 1.This definition is more general than the balance criteria used in AVL or red-black trees.

```
{-@ measure balanced @-}
{-@ balanced :: Tree a -> Bool @-}
balanced Nil             = True
balanced t@(Node _ l r) = balanced l && balanced r &&
                          h t - mh t <= 1
```

Functions `h` and `mh` denote the maximum and minimum heights of the tree, respectively, i.e. the lengths of the longest and shortest paths from the root to a leaf. Since `mh` is always less than or equal to `h`, we encode this invariant in the function's return type. LH is able to prove it automatically. This allows us to directly write `h t - mh t <= 1`, knowing that the value will always be between 0 and 1 (and never negative). We only show the signatures and annotations of `h` and `mh`, their definitions are straightforward.

```
{-@ measure h @-}
{-@ h :: t: Tree a -> i : { Nat | i >= mh t }  @-}
```

```
h :: Tree a -> Int

{-@ measure mh @-}
{-@ mh :: t : Tree a -> i : { Nat | i <= h t } @-}
mh :: Tree a -> Int
```

We deliberately promote this function using `measure`, as we want the balance information to be embedded directly in the constructor of the tree. By promoting in such way, we strengthen the type system, allowing us to perform more granular pattern matching and, obtaining more precise information about the structure we are working with. If we had used `reflect` instead, the effectiveness would depend on how far the solver is able to unfold the definitions. In this case, we find that `measure` is more powerful than `reflect` when defining properties over data types.

Although the condition `h t - mh t <= 1` is sufficient to consider a tree balanced, we also explicitly require that the left and right subtrees are balanced. The addition of this extra condition on the subtrees is useful in future proofs to avoid repeatedly establishing that subtrees meet the balance condition.

Finally, we define the `BTree` structure, which consists of those trees that satisfy the definition of `balanced`.

```
{-@ type BTree a = { t: Tree a | balanced t } @-}
```

Every time we specify a tree as a `BTree`, LH automatically enforces the corresponding balance restriction on the structure. For example, if we try to construct a tree with three consecutive nodes on the left, then LH produces a type error, indicating that `t` does not satisfy the definition of a `BTree`, i.e., `Node 1 (Node 2 (Node 3 Nil Nil) Nil) Nil`.

3.1 Logarithmic Properties

Once we have established our structure to refer to balanced trees, we want to prove some standard properties about them using LH, in particular those concerning the relationship between the height of a tree and its number of nodes.

$$balanced(t) \Rightarrow h(t) = \lceil \log_2(nc(t) + 1) \rceil \tag{1}$$

$$balanced(t) \Rightarrow mh(t) = \lfloor \log_2(nc(t) + 1) \rfloor \tag{2}$$

Function `nc` counts the number of nodes of a given tree (we omit its definition because it is straightforward).

To prove these properties, we first define the floor and ceiling of the logarithm, since LH lacks built-in support for logarithmic operations. We compute the floor of the base-2 logarithm of a number `n`, `log2F n`, by counting how many times `n` can be divided by 2 until it reaches 0. The ceiling of the logarithm, `log2C n`, is then derived from `log2F n` by incrementing the result by 1 when `pow2 (log2F n) == n`.

We start with the proof of property (2). Its main ingredients are the following auxiliary properties:

$$balanced(t) \Rightarrow nc(t) + 1 < 2^{mh(t)+1} \tag{3}$$

$$balanced(t) \Rightarrow nc(t) \geq 2^{mh(t)} - 1 \tag{4}$$

$$\forall n \in \mathbb{N},\ \forall i \in \mathbb{N} \mid 2^i \leq n < 2^{i+1} \Rightarrow \lfloor \log_2(n) \rfloor = i \tag{5}$$

```
{-@ prop_2 :: t : BTree a -> { log2F (nc t + 1) == mh t } @-}
prop_2 t = log2F (nc t + 1) == mh t ?
           ((nc t + 1 >= pow2 (mh t) ? prop_4 t) *** QED)
         &&&
           (nc t + 1 < pow2 (mh t + 1) ? prop_3 t *** QED))
         ? prop_5 (mh t) (nc t + 1) *** QED
```

Fig. 1: Proof of property (2).

```
{-@ prop_1 :: t : BTree a -> { log2C (nc t + 1) == h t } @-}
prop_1 Nil = ()
prop_1 t | h t == mh t = log2C n == h t ? prop_6 t
         === log2F (nc t + 1) == h t ? prop_2 t
         === mh t == h t *** QED
         | h t == mh t + 1 = log2C n == h t ? prop_7 t
         === log2F (nc t + 1) + 1 == h t ? prop_2 t
         === mh t + 1 == h t *** QED
```

Fig. 2: Proof of property (1).

Given these properties, the proof of (2) is inmediate: By (3) and (4) it follows that $2^{mh(t)} \leq nc(t) + 1 < 2^{mh(t)+1}$, and by (5) we conclude that $\lfloor \log_2(nc(t) + 1) \rfloor = mh(t)$.

Figure 1 shows the proof written in LH. It is worth noticing that, due to the complexity of the proof, it is impossible to automate it without guiding the proof through the use of proof combinators up to a point where the SMT-solver can take over and determine whether the proof is correct.

The proof of property (1) requires more effort due to the definition of `log2C`, the ceiling of the logarithm. Since `log2C` distinguishes cases based on the result of `log2F`, one needs to associate the unfolding of the function with another property in order to work with a concrete result rather than a raw application. To achieve this, we state two other properties that relate a tree's height with its number of nodes.

$$balanced(t) \wedge mh(t) = h(t) \Rightarrow 2^{\lfloor \log_2(nc(t)+1) \rfloor} = nc(t) + 1 \tag{6}$$

$$balanced(t) \wedge mh(t) + 1 = h(t) \Rightarrow 2^{\lfloor \log_2(nc(t)+1) \rfloor} < nc(t) + 1 \tag{7}$$

Using these properties, we can perform case analysis in the proof of (1) replacing the function call to `log2C` by the corresponding case. Once this replacement is performed, we are able to reach the desired result. The complete proof in LH can be found in Figure 2.

4 Braun Trees

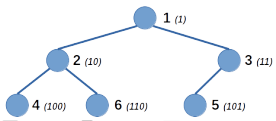

Fig. 3: Braun Tree.

Braun trees [3,9] are binary trees that support efficient element access by natural number index. Each node implicitly corresponds to a position determined by the path taken from the root. To access the element at position i, one considers the binary representation of i. Starting from the least significant bit, a bit value of 0 indicates that the left child should be followed, while a bit value of 1 indicates the right child. This process continues until the most significant bit is reached, at which point the desired element is found.

Due to their structure, Braun trees provide an efficient representation of functional arrays, supporting logarithmic-time access and updates. Additionally, their balanced nature allows them to grow or shrink with minimal cost. The key structural invariant of Braun trees states that, at every node, the left subtree is always either equal in size to the right subtree or larger by exactly one element. This invariant guarantees that the tree remains balanced, maintaining a depth of $O(\log_2 n)$.

We define Braun trees by extending the tree structure of the previous section with the structural invariant.

```
{-@ type Braun a = { t : Tree a | braun t } @-}
type Braun a = Tree a

{-@ reflect braun @-}
{-@ braun :: Tree a -> Bool @-}
braun (Node _ l r) = (nc l == nc r || nc l == nc r + 1) &&
                     braun l && braun r
braun Nil          = True
```

Element access is based on the binary representation of the index. To access a node whose binary index ends in a 1, the structure must already contain the node where that bit is 0. This ensures that the path exists in the tree. As a result, the left subtree always contains the same number of elements as the right subtree, or at most one more. Under this scheme, the index 0 is problematic since its binary representation consists entirely of zeros, and does not yield a valid path to the root. For this reason, our implementation adopts 1 as the minimum index.

An interesting property states that Braun trees are balanced trees.

$$braun(t) \Rightarrow balanced(t)$$

The proof is by induction on the structure of Braun trees. The `Nil` case is immediate. For the inductive case, we establish the following auxiliary property that relates the `balanced` predicate with the absolute value of the difference in the number of nodes between the immediate subtrees of a non-empty tree.

$$balanced(l) \wedge balanced(r) \wedge |\, nc(l) - nc(r) \,| \leq 1 \Rightarrow balanced(Node\, x\, l\, r) \tag{8}$$

The proof of (8) proceeds by case analysis over whether the subtree node counts are exact powers of two, and whether the maximum and minimum heights of the resulting tree `Node x l r` are derived from the left or right subtree. In each case, the goal is to show that the difference between the maximum and the minimum height of the tree is at most one. The proof uses properties (3) and (4) to convert the structural properties of the subtrees into logarithmic expressions. The final step in each branch applies the fact that the difference between the ceiling and floor of a logarithm (or between two such expressions from similar-sized subtrees) is at most one.

Using (8) it becomes straightforward to relate braun trees to balanced trees. The complete proof in LH can be found in Figure 4.

Directly from the structural property, it is possible to prove that in a Braun tree `nc t` satisfies the following inequalities:

$$\text{braun}(t) \Rightarrow 2^{h(t)-1} \leq \text{nc}(t) < 2^{h(t)} \tag{9}$$

```
{-@ braun_is_balanced :: t : Braun a -> { balanced t } @-}
braun_is_balanced Nil             = ()
braun_is_balanced t@(Node v l r) = balanced t
   === (balanced l && balanced r && h t - mh t <= 1)
       ? braun_is_balanced l
   === (balanced r &&  h t - mh t <= 1)
       ? braun_is_balanced r
   === h t - mh t <= 1 ? prop_8 v l r *** QED
```

Fig. 4: Proof of balance of Braun Trees.

A proof of this property in Why3 is presented by Filliâtre [1]. We were able to replicate it in LH by means of the following lemmas:

```
{-@ inv_size :: l : Braun a -> r : {Braun a | nc r <= nc l}
             -> {h l >= h r}

    type BraunGE a N = { t : Braun a | nc t >= N }

    inv_height1 :: t:BraunGE a 1 -> {pow2 (h t - 1) <= nc t}
    inv_height2 :: t:BraunGE a 1 -> {pow2 h t > nc t} @-}
```

Both `inv_height` lemmas establish the lower and upper bound on the `nc` function, respectively, while `inv_size` helps decide on the unfolding of the `h t` function. Compared to Why3, LH struggles more with reasoning about inequalities involving `pow2` and `nc`. We have to guide the proof manually until reaching a point where an inductive step can be applied, allowing LH to complete the verification; something that Why3 is often able to handle automatically.

4.1 Arrays

Having defined Braun trees, we now turn to using them as a foundation for implementing functional arrays. Our development on functional arrays is motivated by the one presented by Nipkow and Sewell [7].

```
{-@ type Array a = Braun a @-}
type Array a = Braun a
```

Associated with this structure, we define the following operations on arrays: `lookup`, `update`, `adds`, `len`, and `list`. As we develop these functions, it is desirable to establish their key correctness properties. LH facilitates this process by allowing us to specify and verify invariants relative to these functions.

For instance, using the refinement type `ArrayGE a n`, which represents an array with at least `n` elements, we can restrict the `lookup` function to ensure that the index we are looking for is always present in the array:

```
{-@ type ArrayGE a N = BraunGE a N @-}
{-@ lookup :: { n: Nat | n > 0 } -> ArrayGE a n -> a @-}
lookup 1 t@(Node v _ _) = v
lookup n t@(Node v l r) | even n     = lookup (div n 2) l
                        | otherwise = lookup (div n 2) r
```

However, when attempting to prove the correctness of this definition, a challenge arises: LH cannot automatically infer that if the index is not even, then `nc r >= div n 2`, even when the overall condition `nc (Node v l r) >= n` holds. This limitation stems from the definition of Braun trees, which enforces a left-heavy

balance. Although this invariant guarantees that the right subtree is populated when needed, LH is not able to deduce this implicitly. To assist the SMT-solver in reasoning about parity and branching, we explicitly promote the `even` function into the refinement logic.

```
{-@ reflect even @-}
even i = i `mod` 2 == 0
```

By combining this reflection with `PLE`, we allow LH to reason about parity, enabling it to correctly deduce the shape and size of subtrees during recursive calls. `PLE` is a key part of this proof since the solver needs the unfolding to correctly resolve the invariants.

This promotion will be the basis for all subsequent operations, as they rely on similar structural reasoning.

The `update` function either modifies an existing element or extends the array by one element, with the following refinement type:

```
{-@ type ArrayNON1 a N I
 = {arr : Array a | (I <= N => nc arr == N) &&
                    (I == N+1 => nc arr == N+1)} @-}

{-@ update :: {n:Nat | n>0} -> a -> arr:ArrayGE a {n-1}
           -> ArrayNON1 a {nc arr} {n} @-}
update :: Int -> a -> Array a -> Array a
```

Again we make sure that the element we want to update is in the structure by using `ArrayGE`. The return type `ArrayNON1` preserves the Braun invariant limiting the number of nodes to either the same number or increased by exactly one.

The `adds` operation extends an array by adding the elements of a list `xs` at the end. The size of the resulting array is then `n + length xs`.

```
{-@ type ArrayN a N = { arr : Array a | nc arr == N } @-}

{-@ reflect adds @-}
{-@ adds :: xs : [a] -> n : Nat -> ArrayN a n
         -> ArrayN a { n + length xs } @-}
adds :: [a] -> Int -> Array a -> Array a
```

The `len` operation is straightforward, as it simply returns the node count.

```
{-@ len :: arr : Array a -> {n : Nat | nc arr == n} @-}
len arr = nc arr
```

Lastly, we would like to be able to transform arrays to lists; for this we define the function `list`. This function is governed by the following invariant:

```
{-@ list :: arr:Array a -> {xs:[a] | nc arr == length xs} @-}
list Nil          = []
list (Node x l r) =  x : splice (list l) (list r)

{-@ reflect splice @-}
{-@ splice :: xs:[a] -> ys:[a]
      -> zs : {[a] | length zs == length xs + length ys} @-}
splice :: [a] -> [a] -> [a]
```

The type `{ xs : [a] | nc arr == length xs }` states that the output list `xs` contains exactly `nc arr` elements of type `a`. In this case the splice function alternates elements of the lists so the sorting is correct.

So far it has not been necessary to rely on the theorem-proving capabilities of LH. However, here we are faced with a difficulty: while we can verify that the sizes match, the next natural question is whether the order of the elements in the list corresponds to their traversal order in the tree. To verify this, two alternatives can be considered. The first one is to enrich the refinement type of the function by extending the invariant with an explicit notion of element order, allowing LH to check the alignment between the tree and list positions. The second alternative is to shift the problem towards the theorem-proving capabilities, where we can encode and prove a separate property that establishes the alignment. The first alternative corresponds to a correct-by-construction approach, whereas the second one a more traditional verification approach.

We finally chose the second alternative. We had previously explored the first one for other functions, but found that it significantly increased both complexity and verbosity of the code. In particular, maintaining the necessary invariants required adding new components to our structure, as well as implementing auxiliary types and helper functions to carry additional information throughout recursive transformations. While this was sometimes partially successful, it made the codebase more difficult to manage and less modular.

4.2 Functional Correctness of Array Operations

In this section, we validate the invariants proposed in [7] for the array operations introduced in the previous subsection. The following properties are automatically valid, essentially because they are encoded as part of the refinement types of the respective functions:

$(IA1)$ $length(list(t)) = nc(t)$

$(IA2)$ $braun(t) \wedge n \in [1,\ nc(t)] \Rightarrow nc(update(n, x, t)) = nc(t)$

$(IA3)$ $braun(t) \wedge n \in [1,\ nc(t)] \Rightarrow braun(update(n, x, t))$

$(IA4)$ $braun(t) \Rightarrow nc(update(nc(t) + 1, x, t)) = nc(t) + 1$

$(IA5)$ $braun(t) \Rightarrow braun(update(nc(t) + 1, x, t))$

$(IA6)$ $braun(t)$
$\Rightarrow nc(adds(xs,\ nc(t),\ t)) = nc(t) + length(xs) \wedge braun(adds(xs,\ nc(t),\ t))$

Invariant (IA1) is encoded in the refinement type of `list`. Invariants (IA2) through (IA5) are proved in the implementation of `update`, and (IA6) is checked by `adds`.

On the other hand, several other invariants require additional guidance for the SMT-solver to be proved. These are verified using equational reasoning with proof combinators:

$(IA7)$ $braun(t) \wedge i < nc(t) \Rightarrow list(t)\ !!\ i = lookup(i + 1\ t)$

$(IA8)$ $braun(t) \Rightarrow list(update(nc(t) + 1, x, t)) = list(t)\ ++\ [x]$

$(IA9)$ $braun(t) \Rightarrow list(adds(xs,\ nc(t),\ t)) = list(t)\ ++\ xs$

$(IA10)$ $braun(t) \wedge n \in [1,\ nc(t)] \Rightarrow list(update(n, x, t)) = \text{list}(t)[n - 1 := x]$

An expression `xs !! i` denotes list indexing, while `xs[i := x]` denotes the list `xs` with the `i`-th element replaced by `x` (positions in a list start from zero).

We start with (IA7), which states the alignment that exists between the tree positions and those at the list that is obtained with the `list` function.

```
{-@ ia7 :: arr: Array a -> n: { Nat | n < nc arr }
        -> {(list arr) !! n == lookup (n+1) arr} @-}
ia7 :: Eq a => Array a -> Int -> Proof
```

The idea of the proof is inspired by the one presented by Nipkow and Sewell [7]. To establish that the list and the tree share the same element order, we require a way to proceed recursively through the tree while traversing the list in parallel. Given that the result of the `list` function is constructed by concatenating the root element to the result of the `splice` operation applied to the recursive calls of `list` on the subtrees, we can associate the choice of a branch in the tree with a corresponding segment of the list based on the parity of the index `n`. In this way, the recursive descent into the tree is naturally aligned with the decomposition of the list.

We then establish the following property; assume $length(xs) > 0$.

$$n < length(xs) + length(ys) \wedge \mid length(xs) - length(ys) \mid \leq 1$$
$$\Rightarrow splice(xs, ys) \,!!\, n = (\text{if } even(n) \text{ then } xs \text{ else } ys) \,!!\, (div\, n\, 2)$$

Now we are able to rely on `PLE` to handle most of the proof. Once we structure the recursion properly and reach the inductive step, the SMT-solver is able to finish the proof without any assistance. With this result, we are able to successfully synchronize the recursive progress on both the list and tree, enabling LH to complete the proof inductively.

We now turn to the proof of Invariant (IA8):

```
{-@ ia8 :: arr:Array a -> x:a
     -> {list (update (nc arr + 1) x arr)==list arr++[x]} @-}
ia8 :: Eq a => Array a -> Int -> Proof
```

To prove this property, we must perform a case analysis based on the parity of `nc arr + 1`, as this determines the path taken by the `update` function. In case `nc arr + 1` is even, we can immediately apply the inductive hypothesis, reducing the recursive call to a list. This leads us to the following intermediate equality:

$$splice(list(l) +\!\!+ [x], list(r)) = splice(list(l), list(r)) +\!\!+ [x]$$

Similarly, in case `nc t + 1` is odd, we must prove the following:

$$splice(list(l), list(r) +\!\!+ [x]) = splice(list(l), list(r)) +\!\!+ [x]$$

In both cases, we establish the result by proving the respective equalities as separate lemmas. Each lemma calls the other after one step of unfolding, thereby completing the proof by mutual induction. Once these lemmas are verified, the overall property is established.

Invariant (IA9) ensures that `adds` behaves like standard list append:

```
{-@ ia9 :: arr:Array a -> xs:[a]
        -> {list (adds xs (nc arr) arr)==list arr ++ xs} @-}
ia9 :: Eq a => Array a -> Int -> Proof
```

The main challenge is the recursive nature of `adds`, which iterates over the input list until reaching the empty list. To proceed, we invoke the inductive hypothesis to replace each recursive call to `adds` with a call to `update`. We then

can leverage the property previously proven for `update`, thereby eliminating it in favor of list operations. At that point, the goal reduces to proving the following standard properties of list concatenation: $xs \mathbin{+\!\!+} [] = xs$ which is trivial and $list(t) \mathbin{+\!\!+} [x] \mathbin{+\!\!+} xs = list(t) \mathbin{+\!\!+} (x : xs)$. Once these auxiliary lemmas are established, LH is able to complete the proof automatically.

The last property we verify is (IA10), which reduces to proving the following:

```
{-@ ia10 :: v:a -> n:{Nat | n >= 1} -> arr:ArrayGE a n
    -> {list (update n v arr)==change (n-1) v (list arr)} @-}
ia10 :: Eq a => a -> Int -> Array a -> Proof
```

In the proof of this property, it is possible to quickly apply the inductive step and eliminate the `update` operation from the reasoning. The main difficulty stems from the fact that, regardless of the parity of the updated index, the recursive descent in the tree always yields an even index until the base case is reached at 1 or 2. To effectively complete the proof, one needs to establish additional properties about list updates, particularly how updates propagate through the `splice` operation.

$$\begin{aligned} & n \geq 2 \wedge n \leq len(xs) + len(ys) + 1 \wedge even(n) \\ & \quad \Rightarrow splice(xs[div\, n\, 2 - 1 := v], ys) = splice(xs, ys)[n - 2 := v] \\ & n \geq 2 \wedge n \leq len(xs) + len(ys) \wedge even(n) \\ & \quad \Rightarrow splice(xs, ys[div\, n\, 2 - 1 := v]) = splice(xs, ys)[n - 1 := v] \\ & n \geq 2 \wedge n \leq len(xs) + len(ys) \wedge \neg even(n) \\ & \quad \Rightarrow splice(xs, ys[div\, n\, 2 - 1 := v]) = splice(xs, ys)[n - 1 := v] \end{aligned}$$

Each of these properties use the other to prove that the induction is done correctly. With these auxiliary properties we are able to reason about the list structure sufficiently for the SMT-solver to verify the desired property.

4.3 Flexible Arrays

Flexible arrays are data structures that support dynamic growth and shrinkage at both ends, allowing elements to be efficiently added or removed from either the front or the back. We extend our array structure with four operations: `add_lo`, `add_hi`, `del_lo`, and `del_hi`. These functions provide the necessary functionality to manipulate the array from both ends while preserving its structural invariants.

Appending an element to the end of the array is already supported by `update`. Thus, implementing `add_hi` amounts to calling `update` with the current size of the array to append the new element.

The `add_lo` function, which inserts an element at the beginning, requires rebalancing the tree to preserve the Braun invariant. This is done by recursively inserting the existing root into the right subtree and swapping the branches:

```
{-@ add_lo :: x:a -> arr:Array a -> ArrayN a {nc arr + 1} @-}
add_lo x Nil           = Node x Nil Nil
add_lo x (Node v l r) = Node x (add_lo v r) l
```

Since the `Array` type guarantees the Braun property, the required invariants are enforced by construction.

To implement `del_lo`, which removes the first element (i.e., the root), we merge the left and right subtrees. The `merge` function must reverse the effect of `add_lo` to maintain structural integrity:

```
{-@ del_lo :: arr : Array a
     -> {res:Array a | nc arr > 0 => nc res == nc arr-1} @-}
del_lo Nil           = Nil
del_lo (Node _ l r) = merge l r

{-@ reflect merge @-}
{-@ merge :: arr1 : Array a
          -> { arr2 : Array a | nc arr2 == nc arr1 ||
                                nc arr1 == nc arr2 + 1 }
          -> ArrayN a { nc arr1 + nc arr2 } @-}
merge :: Array a -> Array a -> Array a
```

Finally, the `del_hi` operation removes the last element by navigating to it and replacing it with `Nil`. The type signature ensures that the resulting tree has exactly one fewer node than the input:

```
{-@ del_hi :: n : Nat -> ArrayN a n
           -> { a : Array a | n > 0 => nc a == n-1 } @-}
del_hi _ Nil      = Nil
del_hi n (Node v l r)
         | n == 1         = Nil
         | not (even n) = Node v l (del_hi (div n 2) r)
         | otherwise      = Node v (del_hi (div n 2) l) r
```

It is important to note that both `add_hi` and `del_hi` currently run in time $O(nc(t) + \log_2(nc(t)))$, rather than the desired $O(\log_2(nc(t)))$. Achieving logarithmic complexity would require extending the tree structure with an additional parameter that explicitly stores the nc value at each node.

4.4 Functional Correctness of Flexible Array Operations

Following [7], we validate the following invariants for flexible arrays.

(IF1) $nc(add\_lo(x,t)) = nc(t) + 1$
(IF2) $nc(t) > 0 \Rightarrow nc(del\_lo(t)) = nc(t) - 1$
(IF3) $nc(t) > 0 \Rightarrow nc(del\_hi(nc(t),t)) = nc(t) - 1$
(IF4) $nc(add\_hi(x,t)) = nc(t) + 1$
(IF5) $braun(Node\ x\ l\ r) \Rightarrow braun(merge(l,\ r))$
(IF6) $braun(t) \Rightarrow braun(del\_hi(nc(t),\ t))$
(IF7) $braun(t) \Rightarrow braun(add\_lo(x,\ t))$
(IF8) $braun(t) \Rightarrow braun(del\_lo(t))$

Invariants (IF1) through (IF3) are encoded in the output types of the functions. Similarly, invariants (IF5) through (IF8) are verified by setting the result type to `Array`, ensuring that the output satisfies, by construction, the structural requirements of a Braun tree. Invariant (IF4) is established by using the `ArrayNON1` type in the output of the `update` function.

Even though some properties could be directly verified through function definitions and refinement types, other invariants require more powerful reasoning. This is the case of the following invariants, which require the application of theorem-proving features:

$$(IF9)\ braun(t) \Rightarrow list(add\_lo(a,\ t)) = a\ :\ list(t)$$
$$(IF10)\ braun(Node\,x\,l\,r) \Rightarrow list(merge(l,\ r)) = splice(list(l),\ list(r))$$
$$(IF11)\ braun(t) \wedge nc(t) > 0 \Rightarrow list(del\_lo(t)) = tail(list(t))$$
$$(IF12)\ braun(t) \wedge nc(t) > 0 \Rightarrow list(del\_hi(nc(t),\ t)) = init(list(t))$$

where `tail` and `init` are the Haskell functions that return all the elements of a non-empty list except for the first one and last one, respectively.

For (IF9), we prove that inserting an element at the beginning of an array corresponds to prepending that element to the list representation.

```
{-@ if9 :: x: a -> arr: Array a
        -> {  x : list (add_lo x arr) == list arr } @-}
if9 :: Eq a => a -> Array a -> Proof
```

In this case, unfolding `add_lo` and reaching its inductive step is sufficient for LH to automatically complete the proof without additional guidance.

Property (IF10) associates the merging of trees with the `splice` operation:

```
{-@ if10 :: arr1 : Array a
         -> arr2:{Array a | nc arr2 == nc arr1 ||
                            nc arr1 == nc arr2 + 1 }
         -> { splice (list arr1)(list arr2)
                  == list (merge arr1 arr2) } @-}
if10 :: Eq a => Array a -> Array a -> Proof
```

In this case, LH is able to solve the proof automatically.

Property (IF11) states that converting the result of `del_lo` to a list yields the same result as taking the tail of the list obtained from the original array:

```
{-@ if11 :: arr : Array a
         -> {tail (list arr) == list (del_lo arr)} @-}
if11 :: Eq a => Array a -> Proof
```

Here we encounter a challenge: on one side we are doing recursion on a list, and on the other side, on a tree that is being merged. The proof relies on (IF10).

Finally, (IF12) states that removing the last element from a Braun tree corresponds to dropping the last element from its list representation.

```
{-@ if12 :: arr:  { Array a | nc arr > 0 }
      -> {init (list arr) == list (del_hi (nc arr) arr)} @-}
if12 :: Eq a => Array a -> Proof
```

The proof again relies on reasoning about the `splice` operation. In the inductive step, after unfolding `del_hi`, the proof splits into two cases based on the parity of the node count of the tree. Although we can successfully reduce the operation, this is not enough for LH to complete the proof. It is necessary to establish the following two properties, which, based on the parity of the sum of the list lengths, state that the application of `init` to a `splice` can be reduced to its application to one of its argument lists:

$$even(len(xs) + len(ys)) \Rightarrow init(splice(xs, ys)) = splice(xs, init(ys))$$
$$\neg\, even(len(xs) + len(ys)) \Rightarrow init(splice(xs, ys)) = splice(init(xs), ys)$$

5 Related work

Previous work has explored the formal verification of Braun trees and other balanced data structures in various proof assistants. However, to our knowledge, this is the first formal verification of Braun trees using LH. Nipkow and Sewell [7] provide a comprehensive Isabelle/HOL development, including correctness and complexity proofs for Braun tree operations. Their work inspired our LH formalization, although we restrict our focus to the structural invariants and functional

correctness of the array operations. From an implementation point of view, the main difference with the development in Isabelle is LH's ability to embed proofs directly within function definitions, rather than requiring all properties to be proved separately as theorems. In fact, a relevant feature of LH is the possibility to write proofs in the same language as the final programs. This provides a significant safety benefit: any modification to the code can be rechecked automatically, without needing to first prove the logic in a separate proof language. Furthermore, this integration ensures that all proof-related information remains within the same codebase and incorporated it to the development process.

Filliâtre [1] implemented a variant of Braun trees to model heaps in Why3, proving their correctness with respect to a priority queue specification. Although Why3 and LH differ in underlying logic and methodology, both aim to express structural invariants through annotations. Compared to Filliâtre's implementation, the `inv_height` property required only 14 lines of Why3 code and almost no proof guidance. In contrast, our LH proof of the same property spanned 34 lines, most of which were dedicated to guiding the system towards the correct reasoning. On the other hand, cases like `fast_size`, an operation introduced by Okasaki [8] and formally verified by Filliâtre. (not included in this paper due to lack of space), required us only the introduction of a simple invariant at the logic level (which is automatically verified by LH). This allowed us to obtain results very similar to those of Why3, with minimal additional effort. In addition to the prior implementation Filliâtre [2] also presented a Flexible Array implementation in this case very similar to ours.

Unlike other approaches that only check properties externally, our implementation encodes many function invariants, and the pre- and post-conditions that preserve them, directly at the refinement type level. As a result, functions like `lookup` and `update` cannot be called with an invalid index, nor can any operation be applied to a tree that does not satisfy the Braun tree invariant.

Okasaki [8] and Nipkow and Sewell [7] have also explored alternative definitions of some of the array operations in order to improve their performance. Our formalization includes the implementation of these efficient operations, but we do not include them here due to lack of space. They can be found in our online repository (`https://gitlab.fing.edu.uy/felipe.de.leon/brauntrees`).

In the LH ecosystem, Vazou et al. [14] demonstrated the use of LH to verify properties of red-black trees. Their work validates that refinement types can express and enforce non-trivial invariants such as balancedness. Although they demonstrate such properties, the version of LH they used lacked both the `reflect` directive and the proof combinators that our solution takes advantage of to a large extent.

Other efforts have been made to formalize related data structures in LH. Rondo [10], for example, presented an implementation of AVL trees purely using `measures`. We experimented with this approach during our development, and while it yielded good results in some cases, it often required substantial auxiliary code and structures. In comparison, a hybrid approach, leveraging both measures

and reflected functions, has provided significantly better results by combining the strengths of both techniques.

Finally, while most work involving LH focuses on theoretical developments or isolated examples, our contribution takes a more practical approach by using the tool to replicate and validate existing proofs in a use case that is simple enough to be accessible, yet rich enough to be non-trivial.

6 Conclusions and Future Work

During the course of this work, we experimented in a non-trivial use case with the different capabilities offered by LH to formalize and verify program invariants. LH directives proved to be particularly useful for automatically verifying structural properties, such as those related to balanced trees and the Braun condition. In some cases, the proofs were remarkably straightforward, with the solver handling constraint enforcement automatically. A striking case is, for example, the one where the simply promotion of a single function like `even` enabled the solver to reason effectively about parity.

However, in other cases, particularly those involving non-trivial mathematical properties such as logarithmic bounds, considerable development effort was required to assist the solver in reasoning through the proof. This also marks a significant shift in the programming paradigm, as developers must adopt the mindset of a theorem prover, along with the associated learning curve that comes with understanding and effectively using these formal verification tools.

With respect to LH's reasoning capabilities, we initially aimed to heavily rely on its PLE feature to automate much of the verification effort. While this was effective for simple local properties, we found it insufficient for more complex proofs that lacked straightforward recursion patterns suitable for unfolding. Nonetheless, PLE remained a valuable tool, even if its applicability was more limited than we had originally expected.

In summary, LH proved to be a flexible and powerful tool for formal program verification. While it is less expressive than full proof assistants, its seamless integration with Haskell enables verification to become a natural part of the development workflow.

We envision two main directions for future work. The first involves to further continue with the formalization of operations on Braun trees. In particular, we have already begun the formalization of some of the efficient operations on Braun trees presented by Okasaki [8]. Another interesting structure to analyze is the Braun-tree-based implementation of priority queues, following the works of Nipkow and Sewell [7] and Filliâtre [1].

The second direction focuses on reworking the definition of functions and proofs using an intrinsically-typed definition of Braun trees, where the structural invariant is encoded as part of the data type definition (and not as an external predicate). We have already experimented with this technique in our LH implementation of AVL trees [6,5].

References

1. Filliâtre, J.C.: Purely applicative heaps implemented with braun trees. https://toccata.gitlabpages.inria.fr/toccata/gallery/braun_trees.en.html (2015), formal proof development in Why3
2. Filliâtre, J.C.: Flexible arrays implemented with braun trees. Toccata (Why3 project gallery) (nd), https://toccata.gitlabpages.inria.fr/toccata/gallery/flexible_arrays.en.html
3. Hoogerwoord, R.R.: A logarithmic implementation of flexible arrays. In: Bird, R.S., Morgan, C., Woodcock, J. (eds.) Mathematics of Program Construction, Second International Conference, Oxford, U.K., June 29 - July 3, 1992, Proceedings. Lecture Notes in Computer Science, vol. 669, pp. 191–207. Springer (1992)
4. Jhala, R.: Writing Specifications liquidhaskell docs. https://ucsd-progsys.github.io/liquidhaskell/specifications/ (2020), accessed: 2024-05-15
5. de León Arias, F.: Estudio del lenguaje LiquidHaskell. Final degree project, Facultad de Ingeniería, Universidad de la República (Uruguay) (2024), available at https://www.colibri.udelar.edu.uy/jspui/handle/20.500.12008/47485
6. de León Arias, F.: Implementation of AVL Trees in LiquidHaskell (2024), https://gitlab.fing.edu.uy/felipe.de.leon/liquid-structures/-/blob/main/src/AVLTrees.hs
7. Nipkow, T., Sewell, T.: Proof pearl: Braun trees. In: Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 18–31. CPP 2020, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372885.3373834
8. Okasaki, C.: Three algorithms on braun trees. J. Funct. Program. **7**(6), 661–666 (Nov 1997). https://doi.org/10.1017/S0956796897002876
9. Paulson, L.C.: ML for the Working Programmer. Cambridge University Press, Cambridge, 2 edn. (1996)
10. Rondon, P., Vazou, N.: Programming with refinement types: Liquidhaskell tutorial. https://ucsd-progsys.github.io/liquidhaskell-tutorial/book.pdf, accessed: May 24, 2025
11. Vazou, N., Breitner, J., Kunkel, R., Van Horn, D., Hutton, G.: Theorem proving for all: equational reasoning in liquid haskell (functional pearl). In: Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell. p. 132–144. Haskell 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3242744.3242756
12. Vazou, N., Breitner, J., Kunkel, W., Horn, D.V., Hutton, G.: Reasoning about programs. https://goto.ucsd.edu/~nvazou/theorem-proving-for-all/02-Reasoning-About-Programs.html (2013), accessed: 2024-05-15
13. Vazou, N., Rondon, P.M., Jhala, R.: Abstract refinement types. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. pp. 209–228. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
14. Vazou, N., Seidel, E., Jhala, R.: Liquidhaskell: Experience with refinement types in the real world. In: Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell. pp. 39–51. Haskell '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2633357.2633366