Practical Verification of QuadTrees

Jonathan Brouwer <j.t.brouwer@student.tudelft.nl>

Abstract

Agda2hs is project which compiles a subset of Agda to Haskell. This paper aims to implement and verify the Haskell library QuadTree in this subset of Agda, so Agda2hs can then produce a verified Haskell implementation. Techniques are developed for proving invariants, preconditions, and postconditions, and are applied in order to implement and verify the QuadTree library. Additionally, recommendations are made to reduce the time needed for verification.

1 Introduction

Haskell is a strongly typed purely functional programming language [1]. A big advantage of this is that it makes reasoning about the correctness of algorithms and data structures relatively simple. However, these proofs are all done on paper, and making a mistake in these proofs is notoriously easy. There is also always the risk that the proof is no longer valid after the code changes. Agda is a dependently typed programming language and interactive theorem prover [2]. Using Agda and the Curry-Howard correspondence [3], one can write a formal proof about the code in the language itself, and use the compiler to verify the correctness of the proof [4, 5]. The compiler also verifies that the proof is still valid each time the code changes.

The Agda2hs [6] is a project that identifies a common subset of Agda and Haskell, and provides a tool that automatically translates code from this subset of Agda to Haskell. This makes it possible to write the program in this subset, using full Agda to prove properties about it, and then translate it to nice looking readable Haskell code. However, agda2hs is not finished yet, as it still lacks some Agda features that it cannot compile to Haskell. It is also not yet known how much effort extra it takes to write code in this subset of Agda.

In this paper, the QuadTree library is implemented and verified in this subset of Agda to determine whether agda2hs can be used to produce a verified implementation of a Haskell library (section 3.1). If this turns out to be difficult, changes to agda2hs or the library to make this possible will be determined (section 3.2-3.3). Then invariants, preconditions, and postconditions of the library will be stated and the techniques used to prove them will be shown shown (section 4). Section 5 discusses responsible research. Finally, the results are discussed (section 6) and the paper is concluded (section 7).

2 Preliminaries

2.1 QuadTrees

The QuadTree is a data structure that is used for storing two-dimensional information in a functional way. [7]. It is defined as:

```haskell
data Quadrant t = Leaf t
        | Node (Quadrant t) (Quadrant t) (Quadrant t) (Quadrant t)

data QuadTree t = Wrapper (Nat, Nat) (Quadrant t)
```

A QuadTree consists of the size of the QuadTree, and the root quadrant. A quadrant is either a leaf (in which case all the values inside the region of the quadrant is the same), or four other nodes. The four subquadrants are then called A (top left), B (top right), C (bottom left), and D (bottom right). Notice that in Figure 1, space is consistently split into four quadrants.

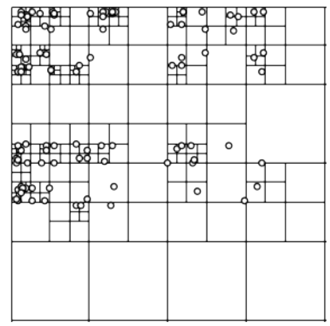

Figure 1: An example QuadTree

There are five functions that can be used to interact with QuadTrees:

```haskell
-- Create a new QuadTree with the specified size
makeTree :: (Nat, Nat) -> t -> QuadTree t
-- Get a lens to the specified location
atLocation :: (Nat, Nat) -> Lens (QuadTree t) t
-- Get the value at the specified location
getLocation :: (Nat, Nat) -> QuadTree t -> t
-- Set the value at the specified location
setLocation :: (Nat, Nat) -> t -> QuadTree t -> QuadTree t
-- Map the value at the specified location
mapLocation :: (Nat, Nat) -> (t -> t) -> QuadTree t -> QuadTree t
```

2.2 Lenses

The QuadTree library makes extensive use of Lenses. Lenses are composable functional references [8]. They allow one to access and modify data in a data structure. This paper chooses to use the Van Laarhoven representation [9], since this is what the original library used. It is defined as:

```haskell
type Lens s a = forall f. Functor f => (a -> f a) -> s -> f s
```

And the functions to interact with Lenses are:

```haskell
-- Get the value at this lens
view :: Lens a b -> a -> b
-- Set the value at this lens
set :: Lens a b -> b -> a -> a
-- Map the value at this lens
over :: Lens a b -> (b -> b) -> a -> a
-- Compose two lenses (Note: This is actually just regular function composition!)
compose :: Lens a b -> Lens b c -> Lens a c
```

3 Implementation

This section describes how the QuadTree library was implemented in Agda, and what challenges had to be overcome to do so. All the code for this project is available in the public domain. Each directory has a README.md which explains the purpose of all files and folders inside of the directory. It is available at: github.com/JonathanBrouwer/research-project.

3.1 Implementing QuadTree

The QuadTree library is implemented by composing lenses, to finally build the atLocation lens, which takes a location and a QuadTree, and returns a lens to the location in the QuadTree.

```
atLocation : (Nat x Nat) -> (depth : Nat) -> Lens (QuadTree t) t
```

atLocation is implemented by composing wrappedTree (which lenses from the QuadTree to its root quadrant) and go. go is the function that does most of the work. The function takes a coordinate and a maximum depth, and returns a lens from a quadrant to the location. Behind the scenes, if the maximum depth is zero, it calls lensLeaf. Otherwise, it composes lensA/B/C/D with a recursive call to itself, that does the rest of the lookup. For example, go (0 , 0) 5 = lensA ∘ go (0 , 0) 4.

```
lensWrappedTree : Lens (QuadTree t) (Quadrant t)
go : (Nat x Nat) -> (depth : Nat) -> Lens (Quadrant t) t
```

lensLeaf is a lens from a leaf quadrant to the value stored there. This function has as a precondition that the quadrant has a depth of 0 (a leaf). LensA/B/C/D is a lens from a quadrant to the A/B/C/D sub-quadrant. This function returns a lens from a quadrant with a certain maximum depth to a quadrant with a maximum depth that is one lower.

```
lensLeaf : Lens (Quadrant t) t
lensA : Lens (Quadrant t) (Quadrant t)
```

get/set/mapLocation can then be defined using the atLocation lens, by composing them with the lens functions. They are shortcut functions so users of the library don't have to interact with lenses directly.

```
getLocation : (Nat x Nat) -> QuadTree t -> t
getLocation = view ∘ atLocation
setLocation : (Nat x Nat) -> t -> QuadTree t -> QuadTree t
setLocation = set ∘ atLocation
mapLocation : (Nat x Nat) -> (t -> t) -> QuadTree t -> t
mapLocation = over ∘ atLocation
```

Finally, makeTree makes a new QuadTree with the same value everywhere, simply by calling the QuadTree constructor

```
makeTree : (size : Nat × Nat) -> (v : t) -> QuadTree t
```

Additionally, QuadTree implements Functor and Foldable. However, the implementation of functor in the original library breaks the compression invariant, so the implementation was changed slightly in this implementation to not break this invariant.

3.2 Challenges when converting Haskell to Agda

When converting Haskell to Agda, the challenge that Agda is a total language is encountered, so functions must terminate. Firstly, this is encountered when a function in the library is actually non-terminating. This would have to be solved by changing the function, or adding preconditions, such that the function does always terminate. Luckily, all the functions in the QuadTree library do always terminate, so this problem was not encountered.

Additionally, the totality of Agda can also be encountered when Agda is not convinced that a function is terminating, even though it is. This was actually encountered, and it was initially solved by adding the {-# TERMINATING #-} pragma in front of the function, together with an explanation for why the function is definitely terminating. Later on a different way of expressing the function was found that did not have this problem.

Finally, while Haskell has some escape latches such as 'error', Agda does not. For example, the get/set/mapLocation functions throw an error when the location provided is outside of the QuadTree. This can be solved by adding a precondition to the function, which states that the location must be inside the QuadTree. However, having to already worry about this when implementing the library is bothersome, so an alternative that was used is to temporarily postulate an 'error' function, and later on replacing it with preconditions.

3.3 Agda2hs modifications required

In order to make a working implementation, agda2hs needed some changes.

- Add support for type synonyms (Fixed in PR #56)
- Insert parentheses where required in infix applications (Fixed in PR #57)
- Support for constructors with implicit arguments (Fixed in PR #60)
- Instance arguments fail to compile (Fixed in PR #66)
- Pattern matching on natural numbers does not compile correctly. (Fixed in custom version)

The first 4 problems have been solved by the Agda2hs contributors in the official Agda2hs version. The last has not been fixed in the official version, but has been fixed in a custom version which is available at github.com/JonathanBrouwer/agda2hs. sectionProving Techniques

The properties that have been proven can be divided into three types of properties: Preconditions, Invariants and Postconditions [10]. Preconditions are properties that must be true when a function is called, postconditions must be true after a function is called, and invariants are properties that must be true of all values of a certain type. In this section it will be shown that these three types of properties each have their own way to be proven in Agda.

3.4 Properties to prove

First, all things that have been proven are listed, sorted into one of the types of properties:
Invariants of a QuadTree:

- Depth invariant: The depth of a QuadTree must be less than or equal to $\lceil log_2(max(width, height))\rceil$. This is to ensure that each coordinate maps to exactly one value.

- Compression invariant: No node can have four leaves that are identical. These need to be fused into a single leaf quadrant. This is needed to keep the QuadTree fast and space efficient.

Preconditions

- When calling atLocation, getLocation, setLocation or mapLocation, the location must be inside of the QuadTree.
- When calling lensLeaf, the quadrant needs to have a maximum depth of zero
- When calling lensA/B/C/D, the quadrant needs to have a maximum depth that is greater than zero

Postconditions

- The lenses returned by all the lens functions satisfy the lens laws: [8]
  - view l (set l v s) = v (Setting and then getting returns the value)
  - set l (view l s) s = s (Setting the value to what it already was doesn't change anything)
  - set l v2 (set l v1 s) = set l v2 s (Setting a value twice is the same as setting it once)
- The functor implementations for Quadrant and QuadTree satisfy the functor laws
  - fmap id = id (Identity law)
  - fmap (f . g) == fmap f . fmap g (Composition law)
- The foldable implementation returns an output of the correct length
  - length quadtreeFoldable vqt = width * height
- The foldable implementation satisfies the foldable-functor law
  - foldMap f = fold . fmap f

3.5 Techniques to prove invariants

Invariants are proven by creating a new datatype with one constructor, which takes the original datatype and a proof for all the invariants. As a simple example, this would be a natural number with the invariant that it is greater than 5.

```
data GreaterThanFive : Set where
  CGreaterThanFive : (n : Nat) -> { .( IsTrue (n > 5) ) } -> GreaterThanFive
```

The proof is marked as implicit {} so that it is removed when compiled to Haskell, and it is marked as as irrelevant .() so that will not interfere when proving postconditions later.

Using this technique, the datatype for a quadrant which is compressed and has a certain maximum depth is: (The datatype for a valid QuadTree is defined very similarly)

```
data VQuadrant (t : Set) {dep : Nat} : Set where
  CVQuadrant : (qd : Quadrant t)
             -> {.(IsTrue (depth qd <= dep && isCompressed qd))}
             -> VQuadrant t {dep}
```

Agda2hs flawlessly compiles this to the following, where the proof is erased:

```
data VQuadrant t = CVQuadrant (Quadrant t)
```

The advantage of making a new datatype over adding the proofs to the original datatype is that if the original datatype has multiple constructors, the datatype does not have to be case split when access to the proofs is needed. The disadvantage is however that this additional wrapper type is visible when compiled to Haskell. To avoid this, one can create a second function of all public functions that take the invariance proof as a precondition, and call the original functions with the wrapper type.

3.6 Techniques to prove preconditions

There are 2 techniques to prove preconditions.

3.6.1 Using an implicit argument

Using the first technique, preconditions are proven by adding the proofs as implicit arguments to the function. As a simple example, this would be a function that takes a natural number greater than 5.

```
takesGtFive : (n : Nat) -> { .( IsTrue (n > 5) ) } -> ?
```

As with invariants, the proof is marked as implicit and irrelevant.

Using this technique, it can be proven that the location must be inside the tree for getLocation: (the proofs for setLocation and mapLocation are similar)

```
isInsideQuadTree : (Nat × Nat) -> QuadTree t -> Bool
isInsideQuadTree (x , y) (Wrapper (w , h) _) = x < w && y < h

getLocation : (loc : Nat × Nat) -> {dep : Nat}
    -> (qt : QuadTree t)
    -> {.( IsTrue (isInsideQuadTree loc qt) )} -> t
```

After being compiled with agda2hs, the implicit argument is removed from the function, just like with implicit constructor arguments.

```
getLocation :: (Nat, Nat) -> QuadTree t -> t
```

3.6.2 Using a datatype with invariants

Using the second technique, proofs are proven by passing in a datatype with an invariant, as was used in section 4.2. The simple example from 4.3.1 would then be written like this, using the type defined in section 4.2:

```
takesGtFive : (n : GreaterThanFive) -> ?
```

For the QuadTree verification, this was used to encode the maximum depth properties of the lens functions, using the same datatype that was defined for the invariants.

```
lensLeaf : Lens (VQuadrant t {0}) t
lensA : {dep : Nat}
    -> Lens (VQuadrant t {S dep}) (VQuadrant t {dep})
```

3.6.3 Comparison

The advantages of using implicit arguments is that one does not have to be define a separate datatype, and that the proof can be dependent on more than one argument. On the other hand, the advantages of the second method is that the defined function are cleaner and more compact. It is then also possible to use the type as an parameter to another type, like it is used in lensLeaf and lensA, and it allows for cleaner reuse of the property.

3.7 Techniques to prove postconditions

Postconditions are proven as separate functions. As a simple example, this is a proof that this function returns a number greater than 5.

```
gt5 : Bool -> Nat
gt5 _ = 42

gt5-is-gt5 : (b : Bool) -> IsTrue (gt5 b > 5)
gt5-is-gt5 b = IsTrue.itsTrue
```

For the QuadTree verification, this technique was used to verify the lens laws of all the lenses defined in the implementation. For example, this is the proof that the ViewSet law holds for lensLeaf.

```
ValidLens-Leaf-ViewSet :
    -> (v : t) (s : VQuadrant t {0})
    -> view (lensLeaf {t}) (set (lensLeaf {t}) v s) ≡ v
ValidLens-Leaf-ViewSet v (CVQuadrant (Leaf x)) = refl
```

When proving preconditions and invariants, these properties have to be marked as irrelevant. This is to ensure that when proving that two function calls are equal, one does not need to show that the proves of the preconditions and invariants are equal (Since the actual value of proofs is irrelevant).

3.8 Results

All of the properties mentioned in section 4.1 have been successfully proven. The implementation took approximately 900 lines of Agda code, and the verification took approximately 1500 lines of Agda code. The time it took to implement this library was only a few days, but the verification took several weeks, though this may be biased by the implementation just being a translation across languages. Whether this time is worth it, depends on the situation. For example, in a situation where even one small error could bring down an airplane, this is clearly worth it, however in most situations it is not.

During the verification phase, one bug that was introduced during the translation to Agda. (The bug was in the foldable implementation with quadtrees that are very wide in comparison with their height). This was not caught by the tests, though this may be because the tests on the foldable implementation are very limited (it was always testing square quadtrees).

4 Responsible Research

In this paper the QuadTree library is implemented and verified, and the techniques (method) used to do so are presented. These techniques are written with the goal that a reader who is trying to implement and verify their own library, can do so using these techniques, and reproduce the results. When there are doubts on how these techniques are actually applied, the code for this project is released on GitHub, so anyone who wants to verify that the techniques really work can see how they were applied in this project.

It is also important that other research which aims to improve on the ideas presented in this paper, can do so. This is why the code is released in the public domain, so other researchers can use it and improve on it in their research.

5 Discussion

5.1 Reducing the time required for verification

In this section, some techniques used to speed up the verification process are presented. First of all, these are some techniques to reduce the time required for verification:

- Postulate theorems about libraries. For example, proving that the following 3 statements about lenses are true, turned out to be difficult enough that it was not worth doing. Intuitively these are clearly true, but proving this in Agda takes a lot of time which depending on the situation may not be worth it.

  ```
  view (l1 ∘ l2) ≡ view l2 ∘ view l1
  set (l1 ∘ l2) ≡ over l1 (set l2 t) v
  over l ≡ set l (view l v) v
  ```

- Use Agda automatic proof search. Automatic proof search often doesn't find a solution, but sometimes it does, and trying it does not cost anything.

- First prove invariants and preconditions, then prove postconditions. Invariants and preconditions change the signature of the function, so when any of them are changed, the proofs for postconditions have to be updated.

Additionally, there are some long-term recommendations I would like to make to improve the process of verifying code in Agda:

- It would be useful to have a better interface to search common proofs which already exist. It is difficult for a novice Agda programmer to find and use the proofs that are already in the standard library. For example, associativity and commutativity of addition and multiplication do not have "associativity" or "commutativity" in their name, though even if they did, there is no easy way to search the names of proofs.

- Improvements to automatic proof search would be useful. The automatic proof search often doesn't find a solution, even if the proof is relatively simple. For example, it cannot find a relatively simple proof such as $(a + b) + c \equiv a + (b + c)$. This is because the proof requires cong and case splitting, which the automatic proof search is not allowed to do by default. Giving it the options '-c cong' makes it find the proof quickly, but this is bothersome.

6 Conclusions and Future Work

In this paper, the QuadTree library is implemented and verified in the subset of Agda that agda2hs supports to determine whether agda2hs can be used to produce a verified implementation of a Haskell library. After some minor modifications to agda2hs, the library has successfully been implemented. Then, the library was verified by verifying invariants by creating a datatype which takes the proof, verifying preconditions by adding the proofs as implicit arguments to the function or by passing in a datatype with an invariant, and postconditions are proven as separate functions. Then, it is shown that verification can be sped up by postulating theorems about libraries that are used. Using these techniques, all the properties that were attempted to be verified, have been verified. MORE POSITIVE + INCLUDE BIAS OF ALREADY HAVING IMPLEMENTATION

The techniques presented have only been used to verify this library, it is possible that other libraries cannot be verified using these techniques, so more research should be done to obtain general conclusions by trying to use these techniques with other libraries.

References

[1] Haskell language. `https://www.haskell.org/`, 2021.

[2] Agda. `https://github.com/agda/agda`, 2021.

[3] Curry-howard correspondence. `https://www.cs.cornell.edu/courses/cs3110/2021sp/textbook/adv/curry-howard.html`.

[4] Christopher Schwaab and Jeremy G. Siek. Modular type-safety proofs in agda. *Proceedings of the 7th workshop on Programming languages meets program verification - PLPV '13*, 2013.

[5] Paul van der Walt and Wouter Swierstra. Engineering proof by reflection in agda. *Implementation and Application of Functional Languages*, pages 157–173, 2013.

[6] Agda2hs. `https://github.com/agda/agda2hs`, 2021.

[7] R. A. Finkel and J. L. Bentley. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, Mar 1974.

[8] Edward Kmett. Lens wiki. `https://github.com/ekmett/lens/wiki/Overview`, 2015.

[9] Twan van Laarhoven. Cps based functional references. `https://www.twanvl.nl/blog/haskell/cps-functional-references`, Jul 2009.

[10] B. Meyer. Applying design by contract. *Computer*, 25(10):40–51, 1992.