Practical Verification of QuadTrees

Jonathan Brouwer <j.t.brouwer@student.tudelft.nl>
Jesper Cockx <j.g.h.cockx@tudelft.nl>
Delft University of Technology

June 14, 2021

Abstract

Agda2hs is a program which compiles a subset of Agda to Haskell. In this paper, an implementation of the Haskell library QuadTree is created and verified in this subset of Agda, such that Agda2hs can then produce a verified Haskell implementation. To aid with this verification, a number of techniques have been proposed which are used to prove invariants, preconditions and post-conditions of 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]. An advantage of this is that it simplifies reasoning about the correctness of algorithms and data structures. Even though this reasoning is simple, Haskell does not provide mechanisms to guarantee this correctness as the code changes, so there is still a risk of making mistakes. Using tests can also never guarantee that code is correct. In contrast, Agda is a dependently typed programming language and interactive theorem prover [2]. Using Agda and the Curry-Howard correspondence [3], it is possible to write a formal proof about the code in the language itself, and to 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.

Agda2hs [6] is a program which 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 a program in this subset, using the power of Agda to prove properties about it, and then translate the Agda code to readable Haskell code. However, Agda2hs is not completed yet, as it still lacks some Agda features that it cannot compile to Haskell. It is also not yet known how much extra effort it takes to write code in this subset of Agda.

In this paper, the QuadTree library is implemented and verified in the subset of Agda that can be compiled by Agda2hs, to determine:

(i) Can the QuadTree library be implemented in the subset of Agda that can be compiled by Agda2hs? (section 3)
(ii) What properties does the QuadTree library guarantee? (section 4.1-4.2)
(iii) How can the properties that the QuadTree library guarantees be proven? (section 4.3-4.6)
(iv) How can the time and effort required to verify the QuadTree library be reduced? (section 5)

2 Preliminaries

2.1 Proofs using the Curry-Howard Correspondence

The Curry-Howard correspondence is a relationship that can be used to interpret typed computer programs as mathematical proofs [3]. This is done by representing false statements as empty types, and true statements as non-empty types. For example, take `IsTrue b` where `b` is some boolean expression. The type is constructed in such a way that it is empty if `b` is false, and non-empty if b is true. So if there exists a value of type `IsTrue b`, this value is a proof that `b` must be true.

These proofs can be used as function arguments, constructor arguments or even as a function result. Since Agda is dependently typed, the proof can also refer to other arguments of a function. For example, this function may only be called when `n` is greater than 5:

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

2.2 Lenses

The QuadTree library makes extensive use of Lenses. Lenses are composable functional references [7]. Using lenses, data in a data-structure can be accessed and modified. This paper chooses to use the Van Laarhoven representation [8], since this is what the Haskell implementation of the QuadTree library uses. It is defined as:

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

Using this representation, `Lens a b` means that given an object of type `a`, we can view or modify an inner object of type `b`. The functions to interact with lenses are:

```
-- 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
```

2.3 QuadTrees

The QuadTree is a data structure that is used for storing two-dimensional information in a functional way [9]. It is defined as:

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

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

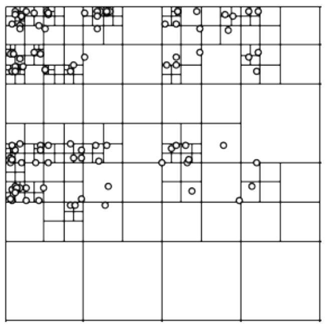
Figure 1: An example QuadTree

A QuadTree consists of the size (width × height) 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 are the same), or four subquadrants. 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.

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

```
-- Create a new QuadTree with the specified size
makeTree :: (Nat, Nat) -> t -> QuadTree t
-- Obtain 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
```

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: https://github.com/JonathanBrouwer/research-project.

3.1 Implementing QuadTree

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

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

`atLocation` is implemented by composing `wrappedTree` (which is a lens from a QuadTree to its root quadrant) and `go`. The `go` function takes a location and a maximum depth, and returns a lens from a quadrant to the specified location. In its implementation, 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 do not 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. The implementation of functor in the original library breaks one of the QuadTree invariants, so the implementation given in this paper was changed slightly in this implementation to not break this invariant.

3.2 Issues when converting Haskell to Agda

3.2.1 Totality

A number of issues arise when converting Haskell to Agda, because Agda is a total language and Haskell is not. Being a total language means that every function must terminate.

The first issue 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 was not a problem for this paper.

Additionally, the totality of Agda can also be encountered when Agda is not able to automatically prove that a function terminates, even though it does. This did actually occur during the implementation of the QuadTree library in Agda. It was initially solved by adding the `\{-\# TERMINATING \#-\}` pragma (which instructs the compiler that this function is terminating) in front of the function, together with an explanation for why the function is definitely terminating. Later on this was solved by expressing the function differently.

3.2.2 Error

Finally, while Haskell has some escape latches such as `error`, Agda does not. For example, the `get/set/mapLocation` functions call `error` when the provided location 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.

This approach is different from the technique presented in [10, p. 16]. The technique presented there is making sure that the `error` function may only be used with non-empty types, thus ensuring that the language stays sound. The technique was chosen because it allows for simpler automatic translation from Haskell. Since the translation in this paper is fully manual anyways, this does not add any value. The technique does come with the downside that the function can still be called with an incorrect input in Agda, resulting in an error at runtime rather than compile-time. Adding a precondition to the function prevents this, therefore this has been chosen.

3.3 Necessary modifications of Agda2hs

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 https://github.com/JonathanBrouwer/agda2hs. Since this makes some breaking changes to Agda2hs, this has not been submitted as a PR.

4 Proving Techniques

The properties that have been proven can be divided into three types: Preconditions, Invariants and Post-conditions [11]. Preconditions are properties that must be true before a function is called, post-conditions must be true after a function is called, and invariants are properties that must be true for 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.

4.1 Finding properties to prove

[10] presents multiple techniques to find properties to prove from Haskell code. A few of those techniques have been selected that were deemed useful for this paper:

1. Define an invariant property when there are types whose correctness depend on invariants [10, p. 7]
2. Define a post-condition property by deriving a definition directly from the test suite [10, p. 9]
3. Define a precondition when there is a risk of numeric overflow, or switch to using unbounded integers instead. [10, p. 9]
4. Define a post-condition when there are type classes which come with laws that all instances of the type class should satisfy [10, p. 10]

Finally, one technique that was not mentioned in [10] was used:

5. Define a precondition if it is required to make the function total, as was described in section 3.1

4.2 Properties to prove

Instead of defining a precondition when there is a risk of numeric overflow as described in technique 3, we switch to using unbounded integers in this papers' implementation of the QuadTree library. This is the same decision that [10] made. Furthermore, the test suite of the QuadTree library consists only of tests that test the type class laws, so no additional properties could be derived from technique 2. Using technique 1, 4, and 5, the following properties of the QuadTree library were derived:

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 there is exactly one value at each location. (Technique 1)
- 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. (Technique 1)

Preconditions of a QuadTree:

- When calling `atLocation`, `getLocation`, `setLocation` or `mapLocation`, the location must be inside of the QuadTree. (Technique 5)
- When calling `lensLeaf`, the quadrant needs to have a depth of zero (i.e. it must be a leaf) (Technique 5)
- When calling `lensA/B/C/D`, the quadrant needs to have a depth that is greater than zero (i.e. it must not be a leaf) (Technique 5)

Post-conditions of a QuadTree:

- The lenses returned by all the lens functions satisfy the lens laws: [7] (Technique 4)
  - `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 to the second value)
- The functor implementations for Quadrant and QuadTree satisfy the functor laws (Technique 4)
  - `fmap id = id` (Identity law)
  - `fmap (f . g) == fmap f . fmap g` (Composition law)
- The foldable implementation returns an output of the correct length (Technique 4)
  - `length quadtreeFoldable vqt = width * height`
- The foldable implementation satisfies the foldable-functor law (Technique 4)
  - `foldMap f = fold . fmap f`

4.3 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 datatype represents 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 post-conditions later. An irrelevant value means that the actual value of the proof does not matter, only its existence does.

Using this technique, the datatype for a compressed quadrant with a certain maximum depth is:

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

The datatype for a valid QuadTree is defined very similarly. Agda2hs flawlessly compiles this to the following Haskell code, where the proof is erased:

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

The advantage of making a new wrapper datatype over adding the proofs to the original datatype is that if the original datatype has multiple constructors, functions that use the proof do not need to be split into multiple cases (one for each constructor). The disadvantage is that this additional wrapper type is visible when compiled to Haskell. To avoid this, it is possible to create an additional function for all public functions. This function then takes the invariance proof as a precondition, and calls the original function with the wrapper type.

4.4 Techniques to prove preconditions

In this section, two techniques to prove preconditions are presented.

4.4.1 Using an implicit argument

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

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

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

Using this technique, a precondition can be used to ensure that the location given to the `getLocation` function must be inside the QuadTree:

```
-- Function that checks if a location is inside a given QuadTree
isInsideQuadTree : (Nat × Nat) -> QuadTree t -> Bool
isInsideQuadTree (x , y) (Wrapper (w , h) _) = x < w && y < h

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

After being compiled with Agda2hs, the precondition is removed from the function, just like with invariants.

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

4.4.2 Using a datatype with invariants

Another technique to prove preconditions is by passing in a datatype with an invariant, as was used in section 4.3. The simple example from 4.4.1 would then be written like this, using the type defined in section 4.3:

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

For the QuadTree verification, this technique 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})
```

4.4.3 Comparison

The advantages of using implicit arguments is that it is not necessary to define a separate datatype, and that the precondition can be dependent on more than one parameter of the function. On the other hand, the advantages of using a datatype with an invariant is that the defined function are cleaner and more compact. It is then also possible to use the type as a parameter for another type, like it is used in `lensLeaf` and `lensA`. It also allows for cleaner reuse of the property, as it does not need to be repeated each time it is used.

4.5 Techniques to prove post-conditions

Post-conditions 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 proofs of the preconditions and invariants are equal, since the actual value of the proofs is irrelevant.

4.6 Results

All of the properties mentioned in section 4.2 have been successfully proven. Most of these proofs have not been shown in the paper, since they are available in the source code. The amount of lines of code that this took is shown in figure 2. This counts all non-empty lines. The verification took about 3 times more lines of code than the implementation. While this comparison is an indication, this should not be taken to mean that the verification took 3 times as much effort, as the information density of the implementation and proofs is different.

In reality, the implementation took approximately one full-time week, while the verification took approximately five full-time weeks. This too should not be taken to mean the verification took 5 times as much effort, as this number may be biased by the fact that the implementation was just a translation from Haskell.

During the verification phase, one bug was found in this papers implementation of QuadTree that was accidentally introduced during the translation to Agda. This was not caught by the tests, though this may be because the tests on the foldable implementation are very limited.

Whether the verification is worth the time spent, 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 many common situations, this verification may not be worth it.

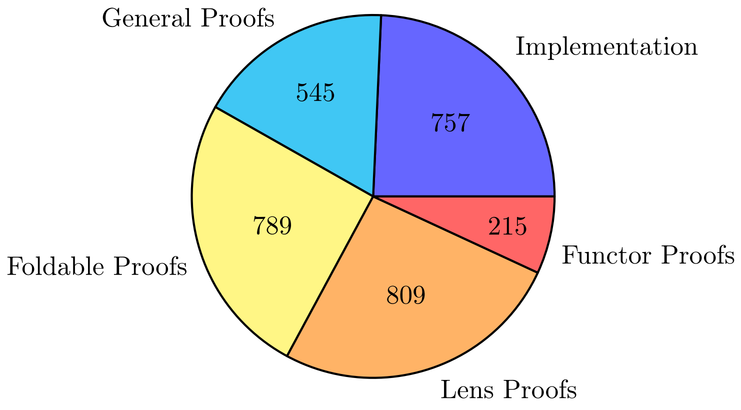

Figure 2: Division of lines of code

5 Reducing the time required for verification

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. Intuitively these are clearly true, but proving this in Agda takes a lot of time which depending on the situation may not be worth doing.

```
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 does not find a solution. But when it does it can save time. Searching is quick so it is worth trying the automatic search.
- *First prove invariants and preconditions, then prove post-conditions.* Invariants and preconditions change the signature of the function, so when any of them are changed, the proofs for post-conditions have to be updated. To prevent the extra work of doing this, one should prove invariants and preconditions first.

6 Future work

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.

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

- *A better interface to search for common proofs.* 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.* 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 ≡ a + (b + c)`. This is because the proof requires the `cong` function and `case splitting`, which the automatic proof search is not allowed to use by default. Giving it the options `-c cong` allows automatic proof search to find the proof quickly, but the options required may be different for other proofs.

7 Conclusions

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.

With some minor modifications to Agda2hs, the implementation of the library in Agda was successful. Some issues that were encountered in the process and solutions to those issues were presented. Next, the properties that the library guarantees were determined by using techniques from [10]. These properties were verified by using techniques developed for invariants, preconditions and post-conditions. Using these techniques, all the properties that were attempted to be verified, have been verified. Finally, it has been shown that the time required for verification can be reduced by postulating theorems about libraries, using automatic proof search and carefully considering the order in which properties are proven.

8 Reproducibility and Integrity

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 try to 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 to the public domain, so other researchers can use it and improve on it in their research.

All the conclusions made in this paper are specific to the implementation and verification of the QuadTree library. Since these techniques have not been applied to other libraries, no general conclusions can be made, and this has been made clear in the conclusions.

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

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

[9] 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.

[10] Joachim Breitnet, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, Joshua Cohen, and Stephanie Weirich. Ready, set, verify! applying hs-to-coq to real-world haskell code. *Journal of Functional Programming*, 31, 2021.

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