Practical Verification of the Reader Monad

Alex Haršáni
Supervisors: Jesper Cockx, Lucas Escot
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

AGDA2HS is a tool that allows developers to write verified programs using Agda and then translate these programs to Haskell while maintaining the verified properties. Previous research has shown that AGDA2HS can be used to produce a verified implementation of a wide range of programs. However, monads that model effectful computations were largely unexplored in the context of AGDA2HS. In this paper, we investigate the Reader monad, which gives us a way to model the global state. As monads are in practice commonly combined with other monads, we also investigate the transformer ReaderT. This paper provides the implementation of Reader and ReaderT in Agda, verifies its properties based on laws of type classes functor, applicative, and monad, as well as monad transformer laws for ReaderT, and assesses whether AGDA2HS produces their correct and useful translation.

1 Introduction

Agda [1] is a dependently typed programming language. Thanks to its type system, it can also be used as a proof assistant so that it can formally verify the properties of the programs. Formal verification gives us much higher confidence that the program is correct. However, Agda is not widely used in practice due to its high complexity and lack of useful libraries and tools found in other languages.

To make use of its proving capabilities in the more popular functional language Haskell [2], members of the Agda community are developing an open-source tool called AGDA2HS [3]. It makes it possible to write formally verified programs in Agda and then translate them to Haskell. In order to do this, AGDA2HS identifies the common subset between Agda and Haskell. All programs belonging to this subset can be translated accurately and correctly with all the properties preserved [4].

Currently, AGDA2HS is expressive enough to translate a wide range of programs [4]. However, monads that model effectful computation have been largely unexplored, even though they are commonly used by Haskell developers.

One of the monads that enable us to model effectful computation is the Reader [5] monad. It allows us to use a global environment of read-only values. This is a very useful capability that would otherwise be impossible because of Haskell's purity. It gives us some significant advantages. For example, this is useful if we have a large number of parameters that are not needed directly in the function but are only there to be passed to another function. With Reader, we can simplify the function definition by not using these unnecessary arguments but rather taking them from the global state.

To combine the advantages of Reader with other monads, we use its transformer, ReaderT. In general, monad transformers enable us to conveniently combine monads to improve both usability and readability. Additionally, since monad transformers are monads as well, we can combine them further with other monads.

Being an instance of functor, applicative, and monad, Reader and ReaderT have to adhere to the laws of their respective type classes. Additionally, ReaderT needs to adhere to the laws of monad transformers. These laws guarantee the correct behavior of their implementations; that is why it is essential to be able to verify them.

This research aims to re-implement the Reader and its transformer in Agda, verify previously mentioned laws, and investigate whether AGDA2HS can produce a verified implementation in Haskell that is both useful and correct. More specifically, this research answers the following question:

- Can we implement Reader and ReaderT in Agda using the language subset defined by AGDA2HS? (Section 3)
- What are the properties that need to be satisfied by Reader and ReaderT and how do we prove these properties? (Section 4)
- Does AGDA2HS provide correct and useful translation of Reader and ReaderT to Haskell? (Section 5.1)

This paper is structured in the following way. Section 2 describes all the concepts that are necessary to understand for later sections. Next, section 3 gives more detail about the implementation process. In section 4, we explore the properties and show how they are proved. Section 5 discusses the results and limitations of the translation process to Haskell on a concrete demo example and the limitations encountered. Section 6 talks about aspects of responsible research and how they relate to this paper. Furthermore, section 7 shows related work. Finally, section 8 concludes the paper.

2 Preliminaries

Before we investigate whether AGDA2HS can produce a verified implementation of Reader and ReaderT, we first need to look at some concepts to understand the following sections. Firstly, as Reader and its transformer are instances of type classes functor, applicative, and monad, we will give a brief description of these type classes and show some simple examples of how they can be used. Then, we will inspect the Reader monad itself with its corresponding functions. Finally, we will discuss the concept of the monad transformer and why it is useful.

2.1 Functor, Applicative and Monad

Type classes give us a way to use polymorphism in Haskell [6]. They declare functions that need to be implemented by its instances to define its behavior in the context of the type class. For example, instances of type class `Eq` need to define when they are equal to each other. In this subsection, we will take a look at type classes of functor, applicative, and monad that are subjects of this research.

Functor

Functor [7] is a type class that requires its instances to define a function called `fmap`. The function `fmap f x` applies function `f` to a wrapped value inside of functor `x`. As an example, we can take a look at a well-known functor `List`. When evaluating `fmap (even) [1, 2, 3]`, fmap will apply function `even` to every element of the list, resulting in `[False, True, False]`. The functor is defined as:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Applicative

Applicative [8] is a type class that is a subclass of functor and requires its instances to define two functions, called `pure` and sequential operator(`<*>`). Function `pure` wraps the value according to our applicative instance, while `<*>` applies wrapped function to a given wrapped value. As an example, let us again take a look at `List`. Evaluating `pure x` results in a list with a single element `[x]`. Perhaps more interestingly, evaluating `pure (+) <*> [1, 2] <*> [3, 4]` results in `[4, 5, 5, 6]`. In this case, `+` is wrapped in the list and applied to the first list to form partially applied functions and then applied to the second list to get final values. The applicative is defined as:

```
class Functor f => Applicative f where
    pure :: a -> f a
    <*> :: f ( a -> b) -> f a -> f b
```

Monad

Monad [9] is a type class that is a subclass of Applicative and requires its instances to define two functions, called `return` and bind operator(`>>=`). Function `return` works in the same way as `pure`, that is wrapping value with the given monad instance. On the other hand, function `>>=` unwraps the value from the monad and applies the value to the function that returns the new value wrapped in the monad. Wadler shows many useful applications of monads, such as error handling, exceptions, state, or output [10]. The monad is defined as:

```
class Applicative m => Monad m where
    return :: a -> m a
    >>= :: m a -> (a -> f b) -> f b
```

To demonstrate the usefulness of monads, we can look at the following example. Let us take a look at a program that checks whether exam and project grades are both passing and outputs the final combined grade. Without monads, we can write something like the following example

```
checkGrade :: Double -> Maybe Double
checkGrade grade = if grade >= 5.8 then Just (grade) else Nothing

evaluateGrade :: Double -> Double -> Maybe Double
evaluateGrade exam project = case checkGrade exam of
    Nothing -> Nothing
    Just exam_grade -> case checkGrade project of
        Nothing -> Nothing
        Just project_grade ->
            Just (0.2 * project_grade + 0.8 * exam_grade)
```

With monads, on the other hand, we can make the function `evaluateGrade` much more understandable and readable. In the next example we use `do` notation which is just syntactic sugar for previously mentioned monad functions [9].

```haskell
evaluateGrade :: Double -> Double -> Maybe Double
evaluateGrade exam project = do
    exam_grade <- checkGrade exam
    project_grade <- checkGrade project
    return (0.2 * project_grade + 0.8 * exam_grade)
```

2.2 Reader Monad

The Reader is a monad that makes it possible to use global, read-only variable [5]. More precisely, `Reader r a` represents a computation on a global variable of type `r` with the result of the computation of the type `a`. It is defined as the following:

```haskell
newtype Reader r a = Reader (r -> a)
```

In order to make use of this monad, we define following functions:

```haskell
-- Retrieves the global variable.
ask :: Reader r r
-- Applies the function r -> a to the global variable and retrieves it.
asks :: (r -> a) -> Reader r a
-- Runs the new Reader computation with modified the global variable.
-- It will not modify the existing global variable.
local :: (r -> r) -> Reader r a -> Reader r a
-- Runs the Reader computation and retrieves its result.
runReader :: Reader r a -> r -> a
```

As the previous definition may seem too abstract, we can take a look at the following example. We define a function `squareArea` of type Reader that obviously calculates the area of the square. However, instead of directly passing the parameter to this function, we use function `ask` to retrieve it from the global environment.

```haskell
squareArea :: Reader Int Int
squareArea = do
  x <- ask
  return (x * x)
```

2.3 Monad Transformer

Monad transformers give us a way to combine monads together [11]. They define a function called `lift`, that enables us to use operations of the inner monad from the context of the transformed monad. This paper looks specifically into the ReaderT transformer that can combine the `Reader` with other monads. Generally, monad transformers are defined as:

```haskell
class MonadTrans t where
    lift :: (Monad m) => m a -> t m a
```

To give an example of the use of ReaderT, as well as demonstrate its usefulness, we can take a look at the following example. This time we calculate the area of the circle. We use ReaderT combined with IO monad, with `pi` as a global variable and `r` as an input from the user.

```
circleArea :: ReaderT Double IO ()
circleArea = do
  r <- lift $ getLine
  pi <- ask
  lift $ putStrLn (show (pi * (read r) * (read r)))
```

3 Implementation

The first step in our investigation of Reader and ReaderT is implementing them in Agda. In this section, we will have a detailed look into how this was done, what problems were encountered, and how they were solved. The complete implementation can be found in the public repository on GitHub[1].

3.1 Implementing Reader

The Reader is implemented in Agda as a record type, as shown in Figure 1. It has its constructor `MkReader` and a field representing its computation. The implementation also includes the Reader functions previously mentioned in the section 2.2. Finally, it defines instances for type classes functor, applicative, monad, and their verified counterparts(verified type classes will be described in section 4.2).

```
record Reader (r a : Set) : Set where
    constructor MkReader
    field
        readerComputation : (r → a)
```

Figure 1: The definition of the Reader record type.

3.2 Implementing ReaderT

In order to implement the ReaderT, as well any other monad transformer, it was first necessary to implement a type class called MonadTrans [11]. This type class defines the `lift` function that wraps the given monad type in a transformer. In addition to this, we have added two erased properties based on monad transformer laws. These force every monad transformer instance to implement proofs for the respective laws. Since they are marked with erasure, they do not translate to Haskell. Implementation can be seen on Figure 2.

[1]https://github.com/AlexHarsani/monad-verification/releases/tag/paper

```
record MonadTrans (t : (Set → Set) → Set → Set) {{ @0 iT : ∀ {m}
    -> {{Monad m}} -> Monad (t m)}} : Set₁ where
    field
        lift : {{Monad m}} -> {@0 a : Set} →  m a → t m a
        @0 first-law : {@0 a : Set} {{iM : Monad m}} → (x : a)
            → lift (return {{iM}} x) ≡ return {{iT}} x
        @0 second-law : {a b : Set} → {{ iM : Monad m }}
            → (x : m a) → (f : a → (m a))
            → _>>=_ {{iT}} (lift x) ((lift ∘ f)) ≡ lift (x >>= f)
```

Figure 2: The definition of the type class MonadTrans.

Next, we define the ReaderT record type. Similarly to Reader, this record type defines constructor `MkReaderT`, as well as the field representing its computation. Additionally, we define functor, applicative, and monad instances, their verified versions, as well as the MonadTrans instance. The definition of Reader can be seen in Figure 3.

```
record ReaderT (r : Set) (m : Set -> Set) (a : Set) : Set where
    constructor MkReaderT
    field
        readerTComputation : (r → m a)
```

Figure 3: The definition of the ReaderT record type.

4 Verification

The next step in our research is verifying the Reader and ReaderT. In this section, we will explore the properties we need to prove, explain why it is important to prove them, discuss the techniques to do this and finally show two examples of the proofs.

4.1 Properties

Type class laws are properties that should hold for all instances of the respective type class to ensure their behavior is correct. While these laws are stated in the documentation of Haskell [9, 11], they cannot be enforced by the language itself. In Agda, on the other hand, we can enforce these laws thanks to Agda's type system. Reader and ReaderT have to adhere to laws of functor, applicative, and monad, being their respective instances. Additionally, monad transformers define their own set of laws, which should hold for ReaderT. Therefore, to make our implementation verified, we need to prove the following properties:

Functor Laws

1. Identity: `fmap id == id`
2. Composition: `fmap (f . g) == fmap f . fmap g`

Applicative Laws

1. Identity: `(pure id) <*> x = x`

2. Homomorphism: `(pure f) <*> (pure x) == pure (f x)`
3. Interchange: `x <*> (pure y) == pure (_$ y) <*> x`
4. Composition: (pure (.) <*> f <*> g) <*> x == x <*> (y <*> z)

Monad Laws

1. Left Identity: `(return x) >>= f == (f x)`
2. Right Identity: `(x >>= return) == x`
3. Associative: `(x >>= f) >>= g == x >>= (\y -> ((f y) >>= g))`

Monad Transformer Laws

1. `lift . return == return`
2. `lift (x >>= f) == (lift x) >>= (lift . f)`

4.2 Verified Type Classes

While type classes functor, applicative, and monad have already been implemented in AGDA2HS before the start of this research project, their implementations did not verify whether they adhere to the laws of their respective type class. For this reason, we have implemented verified subclasses with functions representing the necessary proofs.

Using these type classes not only forces the developer to implement the proofs but also makes it easier to implement them as they can see the type of proof they need to write. As an example for future proofs, each of the verified type classes already implements proofs for two commonly used instances `Maybe` and `Either`.

The verified type classes are implemented in Agda as record types, with the proof functions as fields. Since the verified type class only makes sense in the context of Agda, there is no need for them to be translated to Haskell. Figure 4 shows the VerifiedFunctor as an example.

```
record VerifiedFunctor (f : Set → Set) {{@0 iF : Functor f}} : Set₁ where
    field
        @0 f-id-law : {a : Set} (x : f a) → fmap id x ≡ x
        @0 f-composition-law : {A B C : Set} (g : B → C) → (h : A → B)
            → (x : f A) → fmap (g ∘ h) x ≡ (fmap g ∘ fmap h) x
```

Figure 4: The definition of the type class VerifiedFunctor, that formalizes functor laws.

4.3 Proofs

Proofs are done using a technique called equational reasoning. This technique works in a similar way to solving math equations. We start with the given expression, apply functions and simplify until we get the necessary answer. Hutton [12] shows how to use this technique to reason about Haskell functions in Chapter 16 of the book Programming in Haskell. However, unlike in Haskell, we can actually write these proofs within the Agda language. To do this, we use the following functions. These help us to reduce the expressions until we get the needed result.

```
_=⟨_⟩_ : {A : Set} → (x : A) → {y z : A} → x ≡ y → y ≡ z → x ≡ z
x =⟨ p ⟩ q = trans p q

_=⟨⟩_ : {A : Set} → (x : A) → {y : A} → x ≡ y → x ≡ y
x =⟨⟩ q = x =⟨ refl ⟩ q
```

Let us first take a look at the first law of monad transformers. This law states that lifting a monad results in a transformed monad. The proof for this law can be seen in Figure 5.

```
iMonadTrans .first-law y =
    begin
        lift (return y)
    =⟨⟩ -- applying lift
        MkReaderT (λ _ -> (return y))
    =⟨⟩ -- applying return
        MkReaderT (λ _ -> (pure y))
    =⟨⟩ -- unapplying outer pure
        pure {{iApplicativeT}} y
    =⟨⟩ -- unapplying return
        return {{iMonadT}} y
    end
```

Figure 5: Proof for the first law of monad transformers.

As the next proof, we demonstrate that the id law holds for the ReaderT functor. This law states that mapping functor with `id` function does not change it. However, in order for this law to hold for ReaderT, it needs to hold for the inner monad as well. To prove this, we need to postulate the axiom called functional extensionality. This axiom states that if two functions are equal for every possible input value, they must be equal. We define this as the following [13]:

```
postulate
  functional_extensionality : ∀ {A B : Set} {f g : A → B} → (∀ (x : A) → f x ≡ g x)
    → f ≡ g
```

Using this axiom, we can construct the proof as seen in Figure 6. The rest of the proofs can be found in the public repository.

```
iVerifiedFunctor .f-id-law (MkReaderT f) =
    begin
        fmap id (MkReaderT f)
    =⟨⟩ -- applying fmap
        (MkReaderT $ ((fmap id) ∘ f))
    =⟨⟩ -- applying ∘
        MkReaderT (λ x → ((fmap id (f x))))
    =⟨ cong (MkReaderT $_ ) (functional_extensionality (λ x → f-id-law (f x)))  ⟩
        (MkReaderT (λ x → f x))
    end
```

Figure 6: Proof for the id law of the functor that uses functional extensionality.

5 Results and Discussion

Finally, after implementing and verifying Reader and ReaderT, we inspect the results of our research. In this section, we assess the usage of translated Reader and ReaderT and discuss verification results and their limitations.

5.1 Implementation Results

Both Reader and ReaderT were successfully implemented in Agda using the subset defined by AGDA2HS. In Figure 7, we have written a simple example in Haskell that imports and uses the AGDA2HS generated verified ReaderT. In this demo, ReaderT is combined with the IO monad. When the user runs the demo, they are prompted to type a password for the vault. If the password is correct, they get the diamonds. Otherwise, the contents of the vault will disappear, and they will get nothing. The complete runnable demo, along with the translated Haskell code, can be found in the public repository.

However, while the Haskell translation works as expected, there are slight differences caused by limitations of AGDA2HS and time constraints of this research:

- *Newtype*: In the original Haskell library, ReaderT is defined as `newtype ReaderT r m a`. In AGDA2HS we cannot produce definitions that are `newtype`. Instead, we use record types on Agda's side that are then translated to `data ReaderT r m a`.

- *MonadTrans Constraint*: Newer versions of Haskell use a following quantified constraint in definition of MonadTrans class:

  ```
  class (forall m. Monad m => Monad (t m)) => MonadTrans t where
  ```

  While we have implemented this constraint in Adga, as can be seen in Figure 2, it was not translated to Haskell.

- *Reader as ReaderT*: In the original library, Reader is implemented by using ReaderT with Identity monad:

  ```
  type Reader r = ReaderT r Identity
  ```

  Using an Identity monad with ReaderT means that it will essentially become a simple Reader. During this research, Reader was implemented earlier than ReaderT, so we present it as a standalone version as was seen in Section 3.1.

```
import Control.MonadReader.ReaderT
import Control.MonadReader.MonadTrans

data Vault = Vault
  { password :: String
  , content :: String
  }

hide_vault_content :: Vault -> Vault
hide_vault_content (Vault p c) = (Vault p "nothing")

openVault :: ReaderT Vault IO ()
openVault = do
    given_password <- lift $ getLine
    actual_password <- asksT (password)
    if (given_password /= actual_password)
      then do
        vault_content <- localT (hide_vault_content) (asksT (content))
        lift $ putStrLn ("You get: " ++ vault_content)
      else do
        vault_content <- asksT (content)
        lift $ putStrLn ("You get: " ++ vault_content)

main :: IO ()
main = runReaderT openVault (Vault "secretPassword" "diamonds")
```

Figure 7: Demo written in Haskell, that uses the generated verified ReaderT and MonadTrans.

- MonadReader class: The original library also includes a class called MonadReader, which contains functions associated with Reader monad. In our implementation, these functions are included under the definitions of Reader and ReaderT.

5.2 Verification Results

All the properties shown in Section 4.1 were successfully verified for Reader and ReaderT. Proofs for type class laws for Reader were rather trivial. While they could have been implemented using Agda's function `refl`, we provide step-by-step proofs that are much more understandable and can be useful as a template for implementing future, possibly more complicated proofs. On the other hand, proofs for ReaderT required more work. As ReaderT is only verified if its inner transformed monad is also verified, we had to postulate functional extensionality, as was described in Section 4.3. Complete implementation of proofs can be found in the public repository.

6 Responsible Research

Research integrity is essential for the trustworthiness of the science community. That is why its values were the absolute priority during this research. This section will explore how we follow these values in different aspects. All of the code produced during this research, including the code presented in the figures, is available in the public repository on GitHub. This repository also includes `README` file that describes the setup process so that it can be run in the same conditions as during this research. The code is structured in an organized way with the necessary documentation for easy understanding for everyone with knowledge of Agda and Haskell. This is done in order to make the research more transparent and reproducible. As an additional means to ensure integrity, all the implemented proofs include step-by-step comments that indicate what actions were taken to get to the next step. Finally, no further ethical issues were identified.

7 Related Work

7.1 hs-to-coq

hs-to-coq [14] is a tool for verification of Haskell programs similar to AGDA2HS. It translates Haskell code into Coq, where it is then verified. In section 2.1 of "Total Haskell is Reasonable Coq," the authors demonstrated the use of hs-to-coq in the verification of monad laws. To verify the translated library, users of hs-to-coq have access to a type class `MonadLaws` that formally defines the monad laws [15]. This is very similar to the approach we take in this paper. In AGDA2HS, we formally define monad laws in a record type called `VerifiedMonad`, as was described in Section 4.2.

7.2 Liquid Haskell

Unlike AGDA2HS and hs-to-coq, verification with Liquid Haskell is done directly in the Haskell language. This is done by using refinement types. These are base Haskell types, annotated with properties that should hold for them. In Liquid Haskell, verification of type class laws of monad uses the concept of refinement reflection. With this approach, we use the `reflect` keyword to strengthen the type of function we want to verify, then use it to define the property, which we can then be applied in the proof [16].

8 Conclusions and Future Work

In this paper, we showed that we could produce verified implementation of monads Reader and ReaderT using AGDA2HS. We implemented previously mentioned monads, along with the type class MonadTrans that defines lift operation used by monad transformers. For the purpose of verifying the laws of type classes functor, applicative, and monad, we have implemented records with erased fields that declare the proofs for respective laws. We then created Reader and ReaderT instances of these records, in which the proofs were implemented. Similarly, the MonadTrans type class also includes erased fields for the proofs of the laws of monad transformers. Finally, we have demonstrated the correctness of verified Haskell code generated by AGDA2HS by using it in a demo.

However, there are still possible improvements and questions for future research. The most notable improvement would be implementing MonadReader class that would give an interface with all Reader functions, making the Reader more easily combinable with other monad transformers. Another question that arose during this research is whether it is possible to produce Haskell code with quantified constraints, such as the one in the MonadTrans type class.

References

[1] Agda Development Team, *Agda 2.6.2.1 documentation*, 2021.

[2] "Haskell language." `https://www.haskell.org`.

[3] "agda2hs." `https://github.com/agda/agda2hs`.

[4] S. Anand, D. Sabharwal, J. Chapman, O. Melkonian, U. Norell, L. Escot, and J. Cockx, "Reasonable agda is correct haskell: Writing verified haskell using agda2hs.".

[5] "Control.monad.reader." `https://hackage.haskell.org/package/mtl-2.3/docs/Control-Monad-Reader.html`. Accessed: 2022-06-19.

[6] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 60–76, 1989.

[7] "Data.functor." `https://hackage.haskell.org/package/base-4.16.1.0/docs/Data-Functor.html#t:Functor`. Accessed: 2022-06-19.

[8] "Control.applicative." `https://hackage.haskell.org/package/base-4.16.1.0/docs/Control-Applicative.html#t:Applicative`. Accessed: 2022-06-19.

[9] "Control.monad." `https://hackage.haskell.org/package/base-4.16.1.0/docs/Control-Monad.html`. Accessed: 2022-06-19.

[10] P. Wadler, "Monads for functional programming," in *International School on Advanced Functional Programming*, pp. 24–52, Springer, 1995.

[11] "Control.monad.trans.class." `https://hackage.haskell.org/package/transformers-0.6.0.4/docs/Control-Monad-Trans-Class.html`. Accessed: 2022-06-19.

[12] G. Hutton, *Programming in Haskell*. Cambridge University Press, 2016.

[13] "Axiom.extensionality.propositional." `https://agda.github.io/agda-stdlib/Axiom.Extensionality.Propositional.html`. Accessed: 2022-06-19.

[14] "hs-to-coq." `https://github.com/plclub/hs-to-coq`.

[15] A. Spector-Zabusky, J. Breitner, C. Rizkallah, and S. Weirich, "Total haskell is reasonable coq," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 14–27, 2018.

[16] N. Vazou, *Liquid Haskell: Haskell as a theorem prover*. University of California, San Diego, 2016.