Practical Verification of Lenses: Implementing Formally Verified Lenses using AGDA2HS

Marnix Massar
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 which translates a subset of Agda to readable Haskell. Using AGDA2HS, programmers can implement libraries in this subset of Agda, formally verify them, and then convert them to Haskell. In this paper we present a new, verified implementation of the lens data type, which is used to access data structures in a readable yet functionally pure way. We show successfully verified lenses for record types and tuples, and also present a lens operating on lists that could not be translated properly. We discuss the obstacles encountered during development, and offer thoughts on possible improvements to AGDA2HS.

1 Introduction

When developing software, it is invaluable to know beforehand that the code written will do exactly what was intended by its author, and nothing more or less. In functional programming languages such as Haskell [1], functions are pure, and therefore programs can be proven correct by reasoning about them. However, these proofs need to be done outside of Haskell, resulting in proofs becoming invalid as the code is updated. In languages like Agda [2], proofs can be written in the language itself because of its dependent type system, but Agda's ecosystem is relatively lacking compared to that of a more mature language such as Haskell. To this end, the tool AGDA2HS [3] was created, which translates Agda code to readable Haskell code within a certain subset of the two languages.

Some Haskell data structures and libraries have already been reimplemented using AGDA2HS by previous authors, like Sequences [4], QuadTrees [5], Maps [6], Ranged-sets [7], and Inductive Graphs [8], but many commonly used abstractions still remain unverified through this method.

This paper explores the reimplementation of an additional abstraction: *lenses*. A lens is a reference to a specific subpart of a data structure, and can be created for any nested data structure in order to ease access to its members in a functionally pure way. This results in code that is easier to write, read and maintain [9], as illustrated in Listing 1.

```
data Status = Status { _health :: Int, _level :: Int } deriving Show
data Player = Player { _name :: String, _status :: Status } deriving Show
data Game = Game { _player :: Player, _isStarted :: Bool } deriving Show

healPlayer :: Int -> Game -> Game
healPlayer points game = game { _player =
    (_player game) { _status =
        (_status (_player game)) { _health =
            _health (_status (_player game)) + points }}}

healPlayer' :: Int -> Game -> Game
healPlayer' points game = over (player ⊙ status ⊙ health) game (+ points)
```

Listing 1: An example use case for lenses. The function `healPlayer` increments a player's health without lenses. The much smaller and more readable function `healPlayer'` does the same by composing the lenses `player`, `status`, and `health`. The lens implementations have been left out for brevity.

Lenses are divided into different classes: *well-behaved lenses*, *very well-behaved lenses*,

and *bijective lenses*. These classes each satisfy different *lens laws* and provide different guarantees to their user [10][11][12]. In this research, we set out to implement very well-behaved lenses using AGDA2HS, and to verify their behavior by proving the different lens laws. In doing so, we answer the following questions:

1. Do the language features used by the Haskell implementation of lenses fall into the common subset of Agda and Haskell provided by AGDA2HS? If not, can AGDA2HS be extended to make this possible, or is an alternative lens implementation thinkable that does fall in this subset?

2. What laws need to be satisfied for each lens class (bijective, well-behaved or very well-behaved), and can these laws be formally verified in Agda?

3. Does AGDA2HS actually translate the Agda implementation of lenses resulting from question 1 and 2 to valid and idiomatic Haskell?

4. If question 3 is true, is the generated implementation similar to existing Haskell implementations, and what are the differences, if any? If question 3 is false, why is AGDA2HS not capable of translating the Agda lenses to valid Haskell, and what changes or additional features would AGDA2HS need for this to be possible?

The rest of this paper is structured as follows: first, in Section 2, some preliminaries necessary to answer these questions are explained. We then discuss the implementation of the lens library in Section 3, and its formal verification in Section 4. Section 5 (Responsible Research) discusses reproducibility, and Section 6 (Related work) places this research in a broader context. Finally, we draw our conclusions in Section 7.

2 Preliminaries

In order to understand the rest of the paper, some preliminary knowledge of Agda, proving in Agda, AGDA2HS and lenses is necessary, on which we elaborate in this section.

2.1 Agda, the Curry-Howard correspondence and agda2hs

Agda is a total, dependently typed functional programming language. Agda being *total* means that a function promising to return some type a always returns exactly that type a, and never another type or an error. Languages that are not total, like Haskell, are called *partial*. Because Agda is *dependently typed*, the types in a definition's signature can depend upon one another.

The Curry-Howard correspondence (Table 1) defines an isomorphic mapping between propositional logic and type systems [13]. We do this by representing truth as non-empty types, and falsity as empty types: if we then take a value of a type that represents a certain proposition as an argument to a function, this function can only be called when that type has any values we can provide, that is, if we can provide a proof for this property.

Moreover, Agda is smart enough to understand that function definitions can be used to imply equivalence: if one expression can be translated to another by applying function calls, Agda can confirm for us that these expressions are indeed equivalent. If they are not equivalent or additional information is needed, we will of course get a relevant error message telling us so.

Propositional logic		Type system
proposition	P	type
proof of proposition	$p : P$	program of type
implication	$P \to Q$	function type
truth	$\top$	unit type
falsity	$\bot$	bottom/empty type
negation	$P \to \bot$	function to $\bot$
equivalence	$(P \to Q) \times (Q \to P)$	pair of two functions
universal quantification	$(x : A) \to Px$	dependent function type
existential quantification	$\Sigma A(\lambda x \to Px)$	dependent pair type

Table 1: The Curry-Howard correspondence is an isomorphism between propositional logic and type systems [13].

AGDA2HS is a tool that translates a subset of Agda to readable Haskell. The advantage of taking this extra step is that libraries can be implemented in Agda, formally proven to behave as expected using the Curry-Howard correspondence, and then translated to readable Haskell that can be used in production code. AGDA2HS will not indiscriminately translate all code in its input to Haskell: instead, we have to mark code we wish to translate explicitly with the `COMPILE AGDA2HS x` compiler directive (or *pragma*), where x is the name of the thing we want to translate. We also have the `FOREIGN AGDA2HS x` pragma at our disposal, which inserts the Haskell code given at x verbatim into our resulting Haskell. Listing 2 gives a usage example of these pragmas.

```
-- `double` will  be checked by Agda,
-- and translated using agda2hs.
double : Int → Int
double x = 2 * x
{-# COMPILE AGDA2HS double #-}

-- `triple` will be checked by Agda,
-- but ignored by agda2hs.
triple : Int → Int
triple x = 3 * x

-- `quadruple` will be inserted
-- verbatim into our result.
{-# FOREIGN AGDA2HS
quadruple :: Int -> Int
quadruple x = 4 * x
#-}
```

```
double :: Int -> Int
double x = 2 * x

quadruple :: Int -> Int
quadruple x = 4 * x
```

Listing 2: `double` will be translated using AGDA2HS, `triple` will not, and `quadruple`'s Haskell definition will be inserted in the output exactly as given in the pragma. Agda on the left, resulting AGDA2HS Haskell output on the right.

Because Agda is a total language and Haskell is not, AGDA2HS facilitates a way to write partial functions. We write a function that requires a proof of some specific preconditions. We can then use a special error function for cases where the proofs that are supposed to be

supplied are missing, which AGDA2HS translates to a partial function.

2.2 Lenses

To implement lenses in Agda and formally verify them, it is important to understand how a lens works beneath the hood. Lenses are, however, not defined by their implementation, but by their behavior. To formalize this, we consider as lens any type isomorphic to a pair of getters and setters [14], as illustrated in Equation 1. For example, the `player` lens in Listing 1 is of type `Lens Game Player`.

$$\text{Lens } s\ a \simeq (s \to\ a) \times (s \to\ a \to\ s) \tag{1}$$

Due to this permissive definition, many possible lens implementations exist [15]. One could simply implement a lens as a literal pair of a getter and setter functions, or, more efficiently, as a pair of getter and modifier functions, as shown in Equation 2 and 3 respectively. We will refer to these implementations as a *record lens*. More involved alternatives using for example a Store comonad (Equation 4) or functors (Equation 5) are also possible.

This last mentioned version using functors, known as a *Van Laarhoven lens*, is especially interesting, since it can be combined using regular function composition [15][16].

$$(s \to\ a) \times (s \to\ a \to\ s) \tag{2}$$

$$(s \to\ a) \times (s \to\ (a \to\ a) \to\ s) \tag{3}$$

$$s \to \text{ Store } a\ s \tag{4}$$

$$\forall f.\text{Functor} f \Rightarrow (a \to\ fa) \to\ s \to\ fs \tag{5}$$

In this paper we focus on record lenses, because of their simplicity, and Van Laarhoven lenses, because of their inherent composability and the fact that they are used by Haskell's most downloaded lens library[1].

2.3 Lens laws

There exist several lens laws [12]. The most important of these, the *core laws*, are illustrated in the equations below. Here, $p(s, v) = s'$ is the set (or "put") function which maps a value v and a structure s to an updated structure s', and $g(s) = v$ is the get function which maps a value s to a value v.

The GETPUT law requires that updating a structure with its own field does nothing, since the new value is equal to the old. The STRONGGETPUT law is a stronger version of this, requiring that the structure is solely identified by this field. The PUTGET law requires that setting a value and then immediately retrieving it yields the value just set. Finally, the PUTPUT law requires that updating a field twice in a row is equivalent to only performing the second update, since the first update is overridden by the second.

Note that there also exist a number of weaker lens laws that imply the core laws, and that could thus be used to prove the core laws indirectly [11].

[1] hackage.haskell.org/package/lens

$$p(s, g(s')) \simeq s' \quad \forall s, s' \in S \qquad \text{(StrongGetPut)}$$
$$p(s, g(s)) \simeq s \quad \forall s \in S \qquad \text{(GetPut)}$$
$$g(p(s, v)) \simeq v \quad \forall s \in S, \forall v \in V \qquad \text{(PutGet)}$$
$$p(p(s, v), v') \simeq p(s, v') \quad \forall s \in S, \forall v, v' \in V \qquad \text{(PutPut)}$$

We divide lenses into three classes depending on which of these laws they obey [10]:

1. *Well-behaved* lenses satisfy the GetPut and PutGet laws.
2. *Very well-behaved* lenses satisfy the GetPut, PutGet and PutPut laws.
3. *Bijective* lenses satisfy the StrongGetPut and PutGet laws.

In this paper we mostly focus on very well-behaved lenses, since this class is the most practical when writing application code [17].

2.4 Lens methods

There exist a number of common methods that can be applied on lenses to use them. We consider four of them in this paper:

- `get : s → a` – given a structure of type s, returns the value of its field. This method is occasionally also called *view* in some libraries.
- `over : s → (a → a) → s` – given a structure of type s and a function from a to a, returns an updated structure with the given function applied to its field. This method is occasionally called *modify* in the literature.
- `put : s → a → s` – given a structure of type s and a value of type a, returns an updated structure with the given value replacing its field. This method is occasionally called *set* in the literature. It can be seen as a specialized case of `over`, where the method applied returns a constant value.
- $\odot$ `: Lens t a → Lens s t → Lens s a` – the lens composition operator, that, given two lenses, produces a new lens. With Van Laarhoven lenses, this is equal to normal function composition; other lens implementations need to have a distinct definition of this operator.

3 Implementation

In this section we explain why implementing Van Laarhoven lenses using agda2hs is currently not an option, and then present our implementation of record lenses.

3.1 Van Laarhoven lenses

Ideally, we would like to implement Van Laarhoven lenses via agda2hs, so we can use this powerful abstraction in a formally verified way in Haskell. In order to do this, we need to translate the type signature in Equation 5 to Agda. However, it turns out that while both Haskell and Agda support explicit *forall* types, agda2hs does not. While implementing this feature is an interesting exercise, it is out of the scope of this research, and we shall therefore continue to look into record lenses and their verification.

3.2 Record lenses

To implement the record lens definition from Equation 2, we simply create a record containing two values as in Listing 3: a getter and a setter function. We also show the modification function `over` for this implementation. It needs to iterate the structure s twice: once to retrieve the current value, and once to set the new value. This is a downside, as this operation is potentially very costly for large or complicated types.

```
record Lens (s a : Set) : Set where
    field
        get : s → a
        put : s → a → s

over : {s a : Set} → Lens s a → s → (a → a) → s
over l o f = (put l) o (f ((get l) o))
```

Listing 3: A get/put record lens.

To avoid this, we can instead implement Equation 3 and ask lens implementors to provide a get and an over function, after which we implement put in terms of applying the `const` function to this over method. This would result in the definitions of `Lens` and `put` shown in Listing 4.

```
record Lens (s a : Set) : Set where
    field
        get  : s → a
        over : s → (a → a) → s
open Lens public
{-# COMPILE AGDA2HS Lens #-}

put : {s a : Set} → Lens s a → s → a → s
put l o v = (over l) o (const v)
{-# COMPILE AGDA2HS put #-}
```

```
data Lens s a = Lens { get :: s -> a
                     , over :: s -> (a -> a) -> s }

put :: Lens s a -> s -> a -> s
put l o v = over l o (const v)
```

Listing 4: A get/modify lens and a matching implementation of `put` defined in terms of get and modify. Agda on the left, AGDA2HS Haskell output (with manually adjusted whitespace) on the right.

This is the Agda definition of lenses we use in the rest of this paper. We furthermore define the lens composition operator $\odot$ in Listing 5 in order to be able to arbitrarily combine lenses matching the necessary type constraints.

```
_⊙_ : {s t a : Set } → Lens s t → Lens t a → Lens s a
(l ⊙ m) = record { get  = (get m) ∘ (get l)
                 ; over = λ o f →
                    over l o (const (over m (get l o) f)) }
{-# COMPILE AGDA2HS _⊙_ #-}
```

```
(⊙) :: Lens s t -> Lens t a -> Lens s a
l ⊙ m
  = Lens (get m . get l)
      (\ o f -> over l o (const (over m (get l o) f)))
```

Listing 5: The lens composition operator $\odot$ combines some Lens s t and Lens t a into a new Lens s a. Agda on the left, AGDA2HS Haskell output on the right.

3.3 Concrete examples of lenses

Now that we have defined our lenses as in Equation 3, we define a few concrete example lenses to translate and verify. We present lenses operating on the members of tuples, records, and lists.

3.3.1 Lenses for tuples

Tuples, finite ordered collections, can be indexed using lenses. Listing 6 shows implementations of lenses `one` and `two`, which operate on the first and second element of any 2-tuple, respectively.

To define the `get` function of these lenses, we simply take the `fst` and `snd` functions, which return the relevant member of the tuple. To define the `over` function, we create a function of two arguments: a structure `o : (q × r)`, and a function `f : q → q` for `one` or `r → r` for `two`. We then apply `f` to the relevant member of the tuple, and construct a new tuple consisting of this updated member and the other, unchanged member.

Note that in our `over` function we use the `fst` and `snd` functions again to reconstruct the given structure `o`. Normally, we would want to pattern match here, so that `o` would be split into its left and right parts. We are here however hindered by AGDA2HS's inability to translate pattern matching lambdas with more than one argument.

```
one : {q r : Set} → Lens (q × r) q
one = record { get  = fst
             ; over = (λ o f → f (fst o) , (snd o)) }
{-# COMPILE AGDA2HS one #-}

two : {q r : Set} → Lens (q × r) r
two = record { get  = snd
             ; over = (λ o f → (fst o) , f (snd o)) }
{-# COMPILE AGDA2HS two #-}
```

```
one :: Lens (q, r) q
one = Lens fst (\ o f -> (f (fst o), snd o))

two :: Lens (q, r) r
two = Lens snd (\ o f -> (fst o, f (snd o)))
```

Listing 6: Lenses `one` and `two`, which operate on the first and second fields of 2-tuples. Agda on the left, AGDA2HS Haskell output on the right.

3.3.2 Lenses for records

Because programs dealing with real life data often define new record types, we want to be able to create lenses for arbitrary records. In Listing 7 we provide an example record type `Foo`, and a matching lens `bar` to access one of its fields, `_bar`, with. We prefix the actual field with an underscore and give the lens the more easily writable name `bar`. This is common practice when using lenses, since the lens is the default way to access this field.

To define the `get` function of `bar`, we simply use the field accessor `_bar`. Our `over` function, given a structure `o : Foo`, and a function `f : (Int × Int) → (Int × Int)`, returns a record `o` with its `_bar` field replaced by its mapping from the given function `f`.

Although the translated version of `bar` is correct Haskell, the code generated is not entirely idiomatic:

- The `get` function is translated to a lambda expression, whereas a Haskell programmer would simply write `_bar`.

- The `over` function uses the `Foo` constructor instead of a record update expression like `\ o f -> o {_bar = f (_bar o)}`, even though we do use a record update expression in Agda. The advantage of such a record update expression is that only the field relevant to the lens is explicitly set in the code. In a record with this few fields this distinction does not matter much, but in records with many fields this poses a problem for readability, as every field would be mapped to itself explicitly.

```
record Foo : Set where
  field
    _word : String
    _bar : Char × Int
open Foo public
{-# COMPILE AGDA2HS Foo #-}

bar : Lens Foo (Char × Int)
bar = record { get  = _bar
             ; over = (λ o f → record o
                    { _bar = (f (_bar o)) } ) }
{-# COMPILE AGDA2HS bar #-}
```

```
data Foo = Foo{_word :: String, _bar :: (Char, Int)}

bar :: Lens Foo (Char, Int)
bar = Lens (\ r -> _bar r) (\ o f -> Foo (_word o) (f (_bar o)))
```

Listing 7: Lens `bar` can be used to operate on the `_bar` field of `Foo`. Agda on the left, AGDA2HS Haskell output on the right.

3.3.3 Lenses for lists

A commonly used family of lenses for lists is `ix i : Int → Lens [a] a`. This function creates a lens that operates on the ith element of a list. To understand why this lens proves to be more difficult to recreate than those operating on records or tuples, consider the Haskell code in Listing 8. In this example, we first instantiate a version of `ix` that accesses element five (the sixth element, since Haskell lists are 0-indexed) of a list. We then apply this to a list that contains only four elements. If we look at the types involved, nothing illegal is happening, yet this operation is obviously not valid, since we cannot access elements out of bounds of the list.

```
myList :: [Char]
myList = ['a', 'g', 'd', 'a']

main :: IO ()
main = print $ view (ix 5) myList
```

Listing 8: A problematic usage of the `ix` lens that is nonetheless correct to the type checker. First `ix 5`, the lens to access the sixth element, is created, then it is applied to a list of only 4 elements.

There are multiple solutions thinkable to deal with this discrepancy:

1. The common way to deal with this problem in Haskell implementations is to make the get and set functions of `ix` partial: that is, to return an error when an `ix` lens is trying to access an element out of bounds. Although AGDA2HS does support this in some cases as explained in Section 2.1, we need to provide a proof that a lens is in or out of

```
_!!!_ : (xs : List a) (n : Integer) → Maybe a
[] !!! _       = Nothing
(x :: xs) !!! n = if n == 0
                  then Just x
                  else xs !!! (n - 1)
{-# COMPILE AGDA2HS _!!!_ #-}

maybeSetHead : List a → Maybe a → List a
maybeSetHead [] _               = []
maybeSetHead (x :: xs) Nothing  = (x :: xs)
maybeSetHead (x :: xs) (Just v) = (v :: xs)
{-# COMPILE AGDA2HS maybeSetHead #-}

maybeOverList : (List a) → Integer →
    (Maybe a → Maybe a) → List a
maybeOverList [] _ _             = []
maybeOverList (x :: xs) n f =
    if n == 0
    then maybeSetHead (x :: xs) (f (Just x))
    else x :: (maybeOverList xs (n - 1) f)
{-# COMPILE AGDA2HS maybeOverList #-}

mix : Integer → Lens (List a) (Maybe a)
mix i = record { get  = λ o → o !!! i
               ; over = (λ o f → maybeOverList o i f) }
{-# COMPILE AGDA2HS mix #-}
```

```
(!!!) :: [a] -> Integer -> Maybe a
[] !!! _ = Nothing
(x : xs) !!! n = if n == 0 then Just x else xs !!! (n - 1)

maybeSetHead :: [a] -> Maybe a -> [a]
maybeSetHead [] _ = []
maybeSetHead (x : xs) Nothing = x : xs
maybeSetHead (x : xs) (Just v) = v : xs

maybeOverList :: [a] -> Integer -> (Maybe a -> Maybe a) -> [a]
maybeOverList [] _ _ = []
maybeOverList (x : xs) n f
  = if n == 0 then maybeSetHead (x : xs) (f (Just x)) else
      x : maybeOverList xs (n - 1) f

mix :: Integer -> Lens [a] (Maybe a)
mix i = Lens (\ o -> o !!! i) (\ o -> maybeOverList o i)
```

Listing 9: `mix i` is a family of lenses that operates on lists, maybe returns a value on `get`, and that may or may not update a structure on `over`. Agda on the left, AGDA2HS Haskell output on the right.

bounds of the list it is applied to, which we cannot do: after all, we do not know what list we are applying the lens to because lenses are always created for a type and not for the specific instance they are supposed to access. This could be worked around by adding a member of the empty type $\bot$ (which denotes falsity, see Table 1) to the Lens record type, and then checking locally if we are in bounds. If not, we use our lens's built in empty element to throw an error. This would however mean that we are proving things from nothing, and from falsehood one can prove anything. Therefore this option is not viable, since it *cannot be implemented without breaking trust in all lens proofs.*

2. A solution not involving preconditions or proofs is presented in Listing 9. We make the lens return a `Maybe` type wrapping the list's actual type: applying `get` would then return `Nothing` on an invalid index and `Just a` on a valid index, and applying `over` on an invalid index would simply not update the structure. However, in doing so, the PUTGET law would be broken: after all, if we first put a value v to too large an index and then try to retrieve that value from the overly large index, the result would be `Nothing`, and not v. Therefore, this option *can be implemented, but does not produce a well-behaved lens.*

3. In dependently typed languages like Agda, the above problem might be solved by using a data type with a set length, like `Vec` in the standard library. This type is parameterized at runtime with a number indicating its length. However, we cannot translate Agda code like this to Haskell directly, since parameterizing a type with a concrete number is not possible due to Haskell's lack of dependent types. Therefore, this option can be implemented using our Lens type in Agda like in Listing 10, but *cannot be translated to Haskell.*

```
getVec : ∀ {n} → Fin n → Vec a n → a
getVec zero    (x :: xs) = x
getVec (suc i) (x :: xs) = getVec i xs

overVec : ∀ {n} → Fin n → Vec a n → (a → a) → Vec a n
overVec zero    (x :: xs) f = f x :: xs
overVec (suc i) (x :: xs) f = x :: overVec i xs f

vix : ∀ {n} → Fin n → Lens (Vec a n) a
vix i = record { get  = getVec i
               ; over = overVec i }
```

Listing 10: `vix i` is a family of lenses that operates on the Vec type in Agda's standard library. Due to Haskell's lack of dependent types, this example cannot be easily translated and no AGDA2HS Haskell result can be shown.

4 Verification

Now that we have defined our implementation of the Lens type, its methods, and a number of concrete lenses, we start verifying our lenses. We want them all to be very well-behaved lenses, meaning we try to prove the GetPut, PutGet and PutPut laws. Since all three of these laws consist of equalities, we leverage Agda's equational reasoning for our proofs.

It is important to state here that our proofs do not end up in the AGDA2HS Haskell output, since we do not add a pragma to compile them. This has the nice effect that Agda checks the validity of our proofs for us and then AGDA2HS discards them, since the proofs would do nothing in our Haskell code if they would compile at all.

Proving the lens laws for very well-behaved lenses turns out to be trivial for tuples and records, both types with a fixed size. We use our proof of the PutGet law for the *two* lens as an example, as shown in Listing 11. Every step in this proof consists of nothing more than function application. Although writing proofs in this step by step manner has the benefit of being more readable for humans, the Agda compiler can actually do the whole proof for us, as demonstrated in Listing 12. Both of the tuple lenses and the record lens we defined in Section 3 satisfy the GetPut, PutGet and PutPut laws, and are thus very well-behaved lenses.

The lens for lists, which have sizes that can change during runtime, presented much greater challenges, as Section 3 already detailed. Here we go over the same three options we mentioned there, and discuss them in the context of verification:

1. Creating an implementation that compiles to a partial function will break trust in all further proofs concerning lenses because of the empty type we need for it in our Lens

```
two-putget : {q r : Set} → (s : (q × r)) → (a : r) → get two (put two s a) ≡ a
two-putget s a =
    begin
        get two (put two s a)
    ≡⟨⟩
        snd (put two s a)
    ≡⟨⟩
        snd (over two s (const a))
    ≡⟨⟩
        snd ((fst s) , (const a (snd s)))
    ≡⟨⟩
        snd ((fst s) , a)
    ≡⟨⟩
        a
    □
```

Listing 11: Proof of the PutGet law for the *two* lens using human-readable steps.

```
two-putget : {q r : Set} → (s : (q × r)) → (a : r) → get two (put two s a) ≡ a
two-putget s a = refl
```

Listing 12: The same proof as Listing 11 of the PutGet law for the *two* lens using only Agda's reflexivity.

type. Although a disciplined programmer might indeed only use this empty type in places where it is necessary, we cannot know that any proofs written using the lens type are sound and do not use it without checking the proof manually, which we were trying to avoid by using Agda.

2. The implementation using `Maybe` as a return type is not a very well-behaved lens. The GetPut law does not hold, since putting `Just v` with v some valid value will do nothing if our index is too high, and getting it immediately after will in this case return `Nothing` instead of `Just v`. As an aside, the PutGet law does hold, since getting a value in range and then putting it will not modify the structure, and setting a value out of range will never modify the structure. The PutPut law does hold: doing this on a valid index indeed first puts the first value, and then puts the second value, which is equivalent to only performing the second put. Doing it on an invalid index is doing nothing twice, which is equivalent to doing nothing once.

3. Because the implementation using Agda's `Vec` type could not be converted to Haskell because of its dependent types, no proofs have been written for it. The problems with the first option should however not be an obstacle here, since this implementation uses an argument of a finite type to index the sized vector, so that using it on structures that are too small will not pass the type checker. The problems of the second option are also not present here, since this lens maps directly to the type of its field without wrapping it in another type.

5 Responsible research

It is of vital importance that research can be reproduced. In order to do so, this paper describes its verified version of lenses in detail, and mentions the decisions and trade-offs that were made in creating them. Furthermore, in this section we offer some guidance on how to reproduce our exact set up, so our implementation can be studied, expanded upon or used as-is in Haskell.

The code itself is made available to the public domain from its repository.[2] It contains complete instructions on how the code contained in it is structured and on how to compile and use it. The code presented depends on The Glasgow Haskell Compiler (GHC), Agda, Agda's standard library, and AGDA2HS. The versions used are as follows:

- GHC 8.10.7
- Agda 2.6.3
- agda-stdlib 1.7.1
- AGDA2HS on branch `master`, commit `160478a51bc78b0fdab07b968464420439f9fed6`

No other ethical concerns have been identified.

6 Related work

6.1 Prior Van Laarhoven lenses in agda2hs

In their work about porting QuadTrees to AGDA2HS, Brouwer implements the Van Laarhoven lens shown in Listing 13 [5], avoiding AGDA2HS's lack of support for explicit *forall* by telling AGDA2HS to insert a verbatim Haskell translation of their Lens definition. This is a clever workaround that allowed them to implement and verify QuadTrees, justified by the fact that lenses were not the focus of their research. Nevertheless, it is not a true AGDA2HS translation of lens types, which is why we have not re-used this definition in this paper.

```
Lens : Set -> Set -> Set₁
Lens s a = {f : Set -> Set} -> {{ff : Functor f}} -> (a -> f a) -> s -> f s
{-# FOREIGN AGDA2HS type Lens s a = forall f. Functor f => (a -> f a) -> s -> f s #-}
```

Listing 13: Brouwer's Van Laarhoven lens [5]. Note that the pragma below the lens is not `COMPILE AGDA2HS`, which would compile the definition using AGDA2HS, but `FOREIGN AGDA2HS`, which inserts the given Haskell code verbatim in AGDA2HS's Haskell output.

6.2 hs-to-coq

`hs-to-coq` is a project that translates total Haskell code to Coq [18], for the purpose of verifying it in Coq [19]. As a result, instead of writing verified code that is then ported to Haskell, one would first write Haskell and then verify it externally.

Although to our knowledge no translations of lenses have been performed yet, `hs-to-coq` supports explicit *forall* quantifications in type signatures, and could therefore theoretically be used to verify Van Laarhoven lenses.

[2] github.com/knarka/verified-lenses

`hs-to-coq` can only translate total Haskell programs to Coq, meaning that the problems we encountered with implementing the `ix` lens using AGDA2HS will here too cause issues: after all, the usual implementation of `ix` that throws an error on invalid indices is partial, not total.

6.3 Liquid Haskell

Liquid Haskell is a program verifier for Haskell that enables programmers to write proofs as annotations within their Haskell codebase, and checks functions for totality and termination [20]. Liquid Haskell might be easier to pick up for the average Haskell programmer than AGDA2HS, because learning a separate language for verification is not necessary.

We are not aware of any verification of lenses in Liquid Haskell, nor of any information about its support for explicitly using *forall* in function type signatures. It is therefore hard to say if Van Laarhoven lenses could be verified using Liquid Haskell.

Liquid Haskell's ability to perform proofs on partial functions does look promising: one can prove partial functions to be conditionally total in Liquid Haskell, that is, total given that some precondition holds [21]. Writing such a proof would mean that every usage of this lens would need to be annotated with a proof that that specific call is valid, but unlike AGDA2HS, this proof would not require its own type parameter in the abstract Lens type. Nevertheless, having to prove validity of every single call using a lens could be seen as either good practice or as simply cumbersome, depending on the programmer.

7 Conclusions

As shown in Section 3 and 4, creating a lens type usable in Haskell using AGDA2HS is possible in principle, as is verifying specific lenses created this way. We have shown that for fixed size types like 2-tuples and records these proofs can be trivial, but that for dynamically sized types like lists there are still many limitations to what is possible.

It appears this generalizes to most fixed sized and all dynamically sized types: after all, for types with a set number of fields we can usually write total functions for our Agda lenses that AGDA2HS can therefore translate, and we can write proofs that consist of simply applying function calls. For types which can change size during runtime on the other hand, we necessarily have cases where the lens we created is too large, small, or otherwise out of bounds for the structure it is applied to, and we need to write partial functions. This is still a large limitation for which a solution needs to be found before a fully usable lens library could be created with AGDA2HS.

7.1 Suggested changes to agda2hs

- The implementation of Van Laarhoven lenses, which is the normal way of implementing lenses in Haskell, is currently impossible because AGDA2HS is not able to translate explicit *forall* types in Agda to their counterparts in Haskell. Adding support to AGDA2HS for this feature, which both languages already support, would open up the possibility of properly implementing Van Laarhoven lenses using AGDA2HS.

- The AGDA2HS translation of pattern matching on numbers is flawed, which proves to be a minor problem when implementing indexed lenses such as those for lists. In Agda, natural numbers are defined as `zero` and `(suc n)`, so that 2 for example

would be written as `(suc (suc zero))`. An exhaustive pattern matching function would then match on `zero` and `(suc n)`. In Haskell, we would pattern match on `0` and `n`. However, AGDA2HS explicitly converts the successor function `suc` to the Haskell output, where its definition is absent. This can be worked around by translating Agda's number definition to Haskell, but the resulting code would not be idiomatic Haskell, and this solution would thus contradict AGDA2HS's goal of producing readable Haskell. Another workaround is to forgo the pattern match and use an if-statement, as we did in Listing 9, but this makes proving things about the Agda definition more convoluted. Therefore, the possibility to compile these Agda pattern matches to idiomatic Haskell pattern matches would be a good addition to AGDA2HS.

- Another minor issue with pattern matching is the fact that AGDA2HS does not translate Agda's record update syntax to Haskell's yet, as we saw in Listing 7. Although this does not appear to cause any functional problems, it will cause trouble when updating record types with many fields.

- Lambda expressions that use pattern matching on multiple variables are supported by Agda and Haskell, but not by AGDA2HS as we saw in Listing 6. This shortcoming can be worked around by nesting multiple pattern matching lambdas, but this workaround reduces readability of both the input and the output code, and this would therefore be a nice feature to add.

7.2 Other future research

Future research could be done into finding an implementation of lenses that is better able to support types of dynamic sizes. Another logical follow up to this research would be making AGDA2HS able to translate explicit *forall* usages in types, so that Van Laarhoven lenses can be translated properly.

Research could thereafter be done into a more complete translation of popular Haskell lens libraries, such as `lens`[3] and `microlens`[4]. These libraries include other, related constructs, such as Prisms and Isos [22], which may prove to pose their own interesting challenges during translation with AGDA2HS.

Moreover, it is worth looking into verifying lens laws on lenses for dynamically sized structures in Liquid Haskell. As we described in Section 6.3, verifying lenses operating on dynamically sized structures like lists may be more feasible in Liquid Haskell than in AGDA2HS, but this needs to be looked into further before any definite conclusions can be drawn.

Lastly, an interesting direction of research would be proving properties of Van Laarhoven lenses using `hs-to-coq`, since the required features for this do appear to be supported. If one would succeed at this, it might become possible to prove properties of the commonly used lens libraries mentioned before.

References

[1] *Haskell Language*. URL: https://www.haskell.org/ (visited on 04/19/2022).

[3] hackage.haskell.org/package/lens
[4] hackage.haskell.org/package/microlens

[2] *The Agda Wiki*. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php (visited on 04/19/2022).

[3] *agda/agda2hs: Compiling Agda code to readable Haskell*. Mar. 26, 2022. URL: https://github.com/agda/agda2hs (visited on 04/19/2022).

[4] Shashank Anand. "Producing a verified implementation of sequences using agda2hs". In: (2021). URL: https://repository.tudelft.nl/islandora/object/uuid%3A065b9e87-43e5-4e3c-8e37-04f1b50c8cf7 (visited on 04/19/2022).

[5] Jonathan Brouwer. "Practical Verification of QuadTrees". In: (2021). URL: https://repository.tudelft.nl/islandora/object/uuid%3A550c654e-0443-4f00-bab3-d24ed3afc879 (visited on 04/19/2022).

[6] Dixit Sabharwal. "Verifying correctness of Haskell programs using the programming language Agda and framework agda2hs". In: (2021). URL: https://repository.tudelft.nl/islandora/object/uuid%3A989e34ff-c81f-43ba-a851-59dca559ab90 (visited on 04/19/2022).

[7] Ioana Savu. "Practical Verification of the Haskell Ranged-sets Library". In: (2021). URL: https://repository.tudelft.nl/islandora/object/uuid%3A2c32c354-42e4-466e-9869-b7c68b4388ec (visited on 04/19/2022).

[8] Rico van Buren. "Practical Verification of the Inductive Graph Library". In: (2021). URL: https://repository.tudelft.nl/islandora/object/uuid%3Ac521ed18-b309-4a49-ab87-a21da72aec34 (visited on 04/19/2022).

[9] Albert Steckermeier. "Lenses in Functional Programming". In: (July 1, 2015), p. 13.

[10] Keisuke Nakano. "A Tangled Web of 12 Lens Laws". In: *Reversible Computation*. Ed. by Shigeru Yamashita and Tetsuo Yokoyama. Vol. 12805. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2021, pp. 185–203. ISBN: 978-3-030-79837-6. DOI: 10.1007/978-3-030-79837-6_11. URL: https://link.springer.com/10.1007/978-3-030-79837-6_11 (visited on 04/22/2022).

[11] Sebastian Fischer, Zhenjiang Hu, and Hugo Pacheco. "A Clear Picture of Lens Laws". In: *Mathematics of Program Construction*. Ed. by Ralf Hinze and Janis Voigtländer. Vol. 9129. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 215–223. ISBN: 978-3-319-19797-5. DOI: 10.1007/978-3-319-19797-5_10. URL: http://link.springer.com/10.1007/978-3-319-19797-5_10 (visited on 04/22/2022).

[12] Keisuke Nakano. "Towards a Complete Picture of Lens Laws". In: *arXiv:1910.10421 [cs]* (Oct. 23, 2019). arXiv: 1910.10421. URL: http://arxiv.org/abs/1910.10421 (visited on 04/22/2022).

[13] Jesper Cockx. "Programming and Proving in Agda". In: (), p. 18, Table 1. URL: https://github.com/jespercockx/agda-lecture-notes/blob/master/agda.pdf (visited on 06/07/2022).

[14] Russell O'Connor. *Functor is to Lens as Applicative is to Biplate: Introducing Multiplate*. arXiv:1103.2841. type: article. arXiv, July 11, 2011. DOI: 10.48550/arXiv.1103.2841. arXiv: 1103.2841[cs]. URL: http://arxiv.org/abs/1103.2841 (visited on 05/19/2022).

[15] Paolo Capriotti, Nils Anders Danielsson, and Andrea Vezzosi. "Higher Lenses". In: *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). June 2021, pp. 1–13. DOI: 10.1109/LICS52264.2021.9470613.

[16] *CPS based functional references*. URL: https://twanvl.nl/blog/haskell/cps-functional-references (visited on 05/19/2022).

[17] J. Nathan Foster et al. "Combinators for bi-directional tree transformations: a linguistic approach to the view update problem". In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '05. New York, NY, USA: Association for Computing Machinery, Jan. 12, 2005, pp. 233–246. ISBN: 978-1-58113-830-6. DOI: 10.1145/1040305.1040325. URL: https://doi.org/10.1145/1040305.1040325 (visited on 05/19/2022).

[18] *Welcome! | The Coq Proof Assistant*. URL: https://coq.inria.fr/ (visited on 04/19/2022).

[19] Antal Spector-Zabusky et al. "Total Haskell is reasonable Coq". In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. New York, NY, USA: Association for Computing Machinery, Jan. 8, 2018, pp. 14–27. ISBN: 978-1-4503-5586-5. DOI: 10.1145/3167092. URL: https://doi.org/10.1145/3167092 (visited on 04/19/2022).

[20] Niki Vazou. "Liquid Haskell: Haskell as a Theorem Prover". PhD thesis. UC San Diego, 2016. URL: https://escholarship.org/uc/item/8dm057ws (visited on 06/16/2022).

[21] Niki Vazou et al. "Theorem proving for all: equational reasoning in liquid Haskell (functional pearl)". In: *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*. Haskell 2018. New York, NY, USA: Association for Computing Machinery, Sept. 17, 2018, pp. 132–144. ISBN: 978-1-4503-5835-4. DOI: 10.1145/3242744.3242756. URL: https://doi.org/10.1145/3242744.3242756 (visited on 06/14/2022).

[22] Mitchell Riley. *Categories of Optics*. Number: arXiv:1809.00738. Sept. 7, 2018. DOI: 10.48550/arXiv.1809.00738. arXiv: 1809.00738[math]. URL: http://arxiv.org/abs/1809.00738 (visited on 06/17/2022).