Lenses in Functional Programming

Albert Steckermeier

July 1, 2015

In photography a lens is used to focus on an important part of an image. Similarly the concept of lenses in functional programming offers a way of focusing on a particular part of a record. Furthermore lenses allow the simple and correct manipulation of records by fulfilling a number of lens laws. This work introduces the concept of lenses together with the related lens laws. It will then delve further into the class of *well-behaved* lenses. Lastly the actual implementation of the more specific class of *very well-behaved* lenses is discussed in the context of the popular Haskell `lens` library by Edward Kmett.

1 Introduction

The concept of lenses in functional programming offers a way of focusing on a particular part of a, possibly nested, data structure or container. This focused part is called the view. The container is called the source. Lenses were originally known as functional references. Twan van Laarhoven, for example, did a lot of work [13, 11, 14, 12] on functional references. Later Pierce et al. [3] came up with the name lens during their work on bidirectional programming languages. In the following chapters Haskell code will be used for all examples.
Using a lens we can access the view part and read, write or modify it. For that purpose, there are two functions which define the lens. The first function is a getter, often called `get`, which extracts the view from a source. The second function is a setter, often called `put`, which sets a view value in a source. Since the getter often projects away the part of the source not contained in the view, it usually takes the unmodified source as an additional argument as mentioned in [2, p. 12]. A lens is a first-class value, i.e. a value that can be passed around in a program, of the type `Lens s v`. Here `s` represents the source type and `v` the view type. An example would be a lens which focuses on the `name` field of type `String` of a `Person` data type. The lens type for this example would be `Lens Person String`. Since data structures can be nested, lenses can also be composed. An example for this would be a lens which extracts a `zipcode` of type `Int` from an `Address` data type which is nested in the previously mentioned `Person` data type. The result would be a lens of type `Lens Person Int`.

```haskell
data Address = Address { zipcode :: Int , street :: String,
                         city :: String }
data Person = Person { name :: String, age :: Int,
                       address :: Address }

setZipcode p zip = p
    { address = (address p) { zipcode = zip } }
```

Figure 1: Record updates in Haskell.

In the context of Haskell, the usage of lenses is motivated by the record update syntax shown in Figure 1. A setter for the nested `zipcode` field in a `Person` record is quite lengthy considering that there is only one level of nesting. One can imagine what a setter for a deeper nested structure would look like. Lenses will provide a more compact solution to this problem.
The structure of this paper is as follows. We start off with a basic data definition of lenses together with some basic operations. Then a more detailed look at the lens classes *well-behaved* and *very well-behaved* [1, p. 2, 2, p. 11-15] follows. These classes fulfill several so called lens laws which will be presented and discussed. Then a look at the `lens` library by Edward Kmett [4] follows. The simple usage of the library as well as the implementation of so called van Laarhoven lenses [11, 17] will be presented. We summarize the results and give an outlook on more advanced features in the last chapter.

2 Lenses

To start things off a simple definition of lenses in Haskell will be provided. Then a more detailed look at lens classes which define lens laws that all the lenses of that class have to adhere to follows.

2.1 Lens Basics

Before delving deeper into the laws and properties of lenses a simple definition of the basic construct of a lens will be presented as a data type definition in Haskell. The definition is inspired by the notion of a *Basic Lens* as defined by John Nathan Foster in his work on bidirectional programming languages [2, ch. 2], but omits a creation function for initial sources as in [1, p. 1].
Figure 2 shows the definition in Haskell. We can see that the lens consists of two functions. The first is a getter function `get` which returns the view of type `v` when applied to a source of type `s`. Function number two is a setter function `put` which sets the view part of the source in the second argument to the new view in its first argument. Furthermore there are two functions `view` and `set` which extract the respective getter and setter.

```haskell
data Lens s v = Lens (s -> v) (v -> s -> s)

view :: Lens s v -> (s -> v)
view (Lens get put) = get

set :: Lens s v -> (v -> s -> s)
set (Lens get put) = put
```

Figure 2: Simple definition of a lens in Haskell.

Simply defining a data type and some functions is not enough. Because lenses are a powerful tool, they must satisfy some intuitive laws to eliminate unwanted side effects of retrieving and setting views. There are many different laws that a lens can fulfill. Using the laws different lens classes can be defined. The following sections will discuss the *well-behaved* and *very well-behaved* lens classes as presented in [1, p. 2, 2, p. 11-15]. There are many more classes like isomorphic lenses [12] or oblivious lenses [2, p. 15]. They are however not relevant for this work.

2.2 Well-Behaved Lenses

Before getting into the laws of *well-behaved* lenses, we first require `put` and `get` to be total functions. In Haskell this means that there can not be undefined cases for the definition of `put` and `get`. Well-behaved lenses have to fulfill the following two laws. The first law simply states that if a view value `b` is set in an source `a` using the `put` function of a lens, the same value can be retrieved again using `get`.

```
get (put b a) = b
```
(PutGet)

Law number two states that setting the view retrieved from a source `a`, namely `get a`, does not change the container.

```
put (get a) a = a
```
(GetPut)

These laws already have an important implication. According to [1, p. 6] the definition of the `put` function in a *well-behaved* lens uniquely defines the corresponding `get` function. [1, p. 6 ff] provides a proof for this fact that is however non constructive, meaning that there is currently no deterministic way of deriving the corresponding `get` function from an already defined `put` function.

2.3 Very Well-Behaved Lenses

Very well-behaved lenses add a third law to the laws of *well-behaved* lenses. It states that setting the view value twice always results in the same source which is obtained when setting the value `c` set last. Meaning that the first application of `put` does not affect the latter one.

```
get x = (x,0)
put (x,_) _ = x

lens = Lens get put

-- Example:
-- get (put (1,1) 2) = (1,0) != (1,1)
```

Figure 3: Example of a lens that does not satisfy the PutGet law, but satisfies GetPut.

$$\texttt{put c (put b a) = put c a} \qquad \text{(PutPut)}$$

2.4 Lens Laws

Defining the lens classes is nice, understanding why the laws behind them are necessary is better. [2, p. 12 ff] provides an extensive overview with a list of three main properties which lens laws presented above guarantee. All examples are inspired by [2], but adapted in Haskell code. The following sections will summarize two of the three main goals relevant for the Haskell library presented later: *Exact translation* and *source integrity*. Note that the totality of `put` and `get` already provides some stability of programs which use lenses because the two functions always return some value for every possible input.

2.4.1 Exact Translation

When working with lenses, it is important to consider how updates are handled. In most cases an update of a view is expected to be translated exactly to the source container. No information should be lost or modified when putting the new view data in a source. The law PutGet assures that the lens satisfies this condition.
An example for a lens that does not satisfy PutGet can be defined as depicted in Figure 3. A comment with an example that does not fulfill the law is also provided and it is intuitive that the view is not translated exactly, since the setter `put` drops one argument.

2.4.2 Source Integrity

After looking at the exact translation of the view part, we can also look at what happens to the part of the source that is not covered by the view. The goal here is to minimize the changes to the source. One restriction to achieve this goal is provided by GetPut which states that the source must remain unchanged if the same view is set which was extracted. In other words: Setting the view that was previously set in the source does not change it at all.
An example for a lens which does not satisfy GetPut is given in Figure 4. It also includes a counterexample. It is easily visible why the condition above is not met, because the setter omits the second tuple argument and replaces it with 0, i.e. the setter

```haskell
get (x,_) = x
put x (_,_) = (x,0)

lens = Lens get put

-- Example:
-- put (get (2,2)) (2,2) = (2,0) != (2,2)
```

Figure 4: Example of a lens that does not satisfy the GetPut law, but satisfies PutGet.

modifies the source in almost any case. One thing to keep in mind is that the GetPut law only considers unchanged views as input for `put`. It does not say anything about updates with views which are actually different from the one currently in the source. Such updates might change the part of the source which is not represented in the view when calling `put`.
PutPut is used to restrict the modification of the source further. It ensures that a `put` does not change anything outside the scope of the view. In practice this law is very restrictive and rules out many lenses. [2, p. 14-15] shows an example where PutPut is violated. Since the example is very arbitrary and not really relevant for the lenses used for record updates, this paper will skip the example. Instead we will define a *very well-behaved* lens for our `Person` data type from Figure 1. The lens will focus the `name` field of the record. The definitions of `get` and `put` for this lens are provided in Figure 5. It also includes proofs for each of the laws of very well-behaved lenses which means that the `nameLens` is very well-behaved. In each step of the proof simply the function definitions are applied and we can verify the laws directly. Note that there are no assumptions made about the respective function arguments and fields of the `Person` data type.
Besides a formal correctness of lenses, the laws above can also be used as a heuristic to reject , possibly auto-generated, candidate lenses as mentioned in [2, p. 14]. Most of the lenses in the Haskell library are predefined or auto generated and fulfill these laws without further proofs. However the programmer is still responsible for ensuring the laws above for custom lenses, as Edward Kmett mentions in an overview on lenses. A type system alone is not enough to provide correctness of lenses w.r.t. the laws according to the overview [10] provided by Kmett.

3 The `lens` Library

Edward Kmett, the creator of the `lens` Haskell library [4], originally started working on his first lens library `data-lens` [5] in 2011. Since then lenses have gained considerable traction in the Haskell community. At the time of this paper, the `data-lens` library is no longer maintained by Kmett and the new `lens` library has largely taken its place. There, the definition of lenses is based on van Laarhoven lenses [11, 17, ch. 4]. A term introduced by Russell O'Connor [17] in 2011. Van Laarhoven lenses as implemented in `lens` belong to the previously introduced class of *very well-behaved* lenses as mentioned

```haskell
get :: Person -> String
get (Person name age address) = name

put :: String -> Person -> Person
put newname (Person name age address) = Person newname age address

nameLens = Lens get put

-- PUTGET:
-- get (put nn (Person n age addr)) =
-- get (Person nn age addr) = nn
-- GETPUT:
-- put (get (Person n age addr)) (Person n age addr) =
-- put n (Person n age addr) = Person n age addr
-- PUTPUT:
-- put n1 (put n2 (Person n age addr)) =
-- put n1 (Person n2 age addr) = (Person n1 age addr) =
-- put n1 (Person n age addr)
```

Figure 5: Definition of a *very well-behaved* lens for the `name` field of the `Person` data type from Figure 1.

by Kmett in [8]. Compared to the simple definition of lenses presented at the beginning of this paper, the lens data type and functions are slightly more complicated. However this makes them more powerful than traditional lenses as we will see in the following. First, however, using the library is explained which will make clear that, although the definition is complicated, the usage is simple.

3.1 Using the `lens` Library

To work with the `lens` library, it first has to be installed on the target system. This can be done using the `cabal install lens` command. Also the `Control.Lens` module must be imported. Once the setup is done the first commands using lenses can be executed. Next the most important basic functions of lenses are listed and explained. The examples are largely taken from the examples page of the lens wiki [7] on GitHub. The general type of the lenses used in the examples is depicted in Figure 9 but not really important for the examples at this point. Its explanation follows in the implementation section.

- `get` (infix: `^.`) takes a lens and a source as an argument and returns the corresponding view. An example in GHCi is:

  ```
  ghci> view _2 ("hello","world") -- or ("hello","world")^._2
  "world"
  ```

  Here `_2` is the lens, the tuple `("hello","world")` is the source and `"world"` is the extracted view.

- `set` (infix: `.~`) takes a lens, the updated view and the source as an argument and returns the updated source. An example in GHCi is:

```
ghci> set _1 "hello" ("","world")
      -- or _1 .~ "hello" $ ("","world")
("hello","world")
```

  Here `_1` is the lens, `"hello"` is the updated view, `("","world")` is the original source and `("hello","world")` is the updated source.

- Lenses can be composed using the function composition `.` as follows.

```
ghci> set (_2._1) 42 ("hello",("world","!!!"))
("hello",(42,"!!!"))
```

- The `to` function allows the user to transform the output of a getter with another method. This can be done by appending `to` to the lens. An example in GHCi is:

```
ghci> (3,(5,7))^._2._2.to (*2)
14
```

- It is also possible to derive lenses, as data accessors, for any data type. The Template Haskell macro `makeLenses ''YourDataType` is simply added after the data type definition.

```
data Test a = Test
    { _foo :: Int, _bar :: Int, _sth :: a }

makeLenses ''Test

-- The following lenses are generated:
-- foo, bar :: Lens' (Foo a) Int
-- sth :: Lens (Foo a) (Foo b) a b
```

  The underscore before the field name indicates that a lens should be generated. To avoid confusion with the `Lens'` data type it should be mentioned that it is just a simpler form of a `Lens` that can be used and composed regularly with a `Lens`. The difference will be explained together with the implementation details.

- For our last example we will take the `Person` and `Address` data types from Figure 1 at the beginning of this paper. If we derive lenses similarly to the simple `Test` data type from above for both types we can define the setter function in one line.

```
setZipcode p zip = set (address.zipcode) zip p
```

  `address` and `zipcode` are the generated lenses and are composed with function composition `(.)` to write the setter. Obviously this is a big improvement over the cumbersome Haskell record update syntax.

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

instance Functor Maybe where
    fmap _ Nothing  = Nothing
    fmap f (Just x) = Just (f x)
```

Figure 6: Definition of the `Functor` type class and an example instance for `Maybe`.

There are many more features in the `lens` library but going into all of them would take many more pages. Several sources [7, 9] on the web provide additional material for interested readers. Most importantly one should keep in mind that using the generated lenses from the last example, we can use the simple syntax of `set` to update record values instead of the impractical record update syntax in Figure 1.

3.2 The `Functor` Type Class

Next a quick introduction to some data types required for the explanation of the implementation will be presented. First the `Functor` type class will be explained shortly according to [15]. The definition as well as an instance of a functor are provided in Figure 6.
A Functor is simply a data type that can be mapped over, hence the name `fmap` for the defined function. `fmap` lifts a function from the type `a -> b` to the type `f a -> f b`. Where `f` is a type constructor that takes one other type as an argument. Besides the provided example for `Maybe`, `List` would be another instance of `Functor`. For `List` the `fmap` function simply corresponds to the regular `List.map` function.

3.3 The `Identity` Data Type

The `Identity` data type, shown in Figure 7 is pretty straight forward and defined in `Data.Functor.Identity` [19]. Its constructor has only one field and acts as a wrapper for other data types. `runIdentity` simply peels off the `Identity` and will be useful later. Furthermore it is important that `Identity` is also a functor. The definition basically writes itself, because there are no possibilities to write it. The result type of `f` is unknown which means we can not supply a value to the `Identity` value constructor. This only leaves `f x` as the only possible argument.

3.4 The `Const` Data Type

The last relevant data type is called `Const` which is defined in `Control.Applicative` [16]. Figure 8 shows the definition together with a `getConst` function that works analogously to the `runIdentity` function of `Identity`. Interesting about the `Const` type is that the constructor ignores one type argument similar to the well known `const` function, only on the type level. We can also define an instance for `Functor` which does not

```haskell
newtype Identity a = Identity { runIdentity :: a }

-- runIdentity :: Identity a -> a

instance Functor Identity where
        fmap f (Identity x) = Identity (f x)
```

Figure 7: Definition of the `Identity` data type and an example instance for `Maybe`.

```haskell
newtype Const a b = Const { getConst :: a }

-- getConst :: Const a b -> a

instance Functor (Const a) where
        fmap f (Const x) = Const x
```

Figure 8: Definition of the `Const` data type.

require the usage of `f`. There is simply no argument, to which `f` could be applied to since `Const` has no field of the type `b`.

3.5 Implementation of the `lens` Library

With these things in mind we can now look at the definition of a van Laarhoven lens in the `lens` library together with the `view` and `set` function. The focus lies on how the implementation achieves the functionality of simple lenses (Figure 2), instead of explaining the theory [17] behind van Laarhoven lenses. Figure 9 shows the definition of a lens in the `lens` library. Compared to the simple definition in Figure 2 a lens is no longer defined by two functions but a function in itself. There are many data types like `Getter` and `Setter` defined in `lens` but we will only provide a simplified explanation of the implemented lenses. The library is simply too extensive to present everything.
Instead we will look at the much simpler explanation and implementation provided by Twan van Laarhoven [11]. However, the lens definition is still the same. Informally a lens now lifts a function of type `a -> f b` on the view of type `a` to a function `s -> f t` on the source of type `s`. Because lenses in this library also allow polymorphic updates of records [18], i.e. updates that change the type of a field with a polymorphic type in an instance, the view type changes from `a` to `b`. Since the view type changes, the source type consequently changes too, because the view is contained by the source. Hence the source type changes from `s` to `t`. `Lens'` is the definition of a lens for non polymorphic updates and was already mentioned in Section 3.1.
Furthermore there is now a type constructor `f` in the definition. It is useful when we try to create an update method for a lens by providing different behaviors [11]. The motivation behind the update method is that if a view is altered using a function, the simple lens definition only provides updates by first getting, then modifying and lastly setting the

```haskell
type Lens s t a b = Functor f => (a -> f b) -> (s -> f t)
type Lens' s a = Lens s s a a
     -- Functor f => (a -> f a) -> (s -> f s)
```

Figure 9: Definition of `Lens` and `Lens'` in the `lens` library.

```haskell
set :: Lens' s a -> a -> s -> s
set ln v s = runIdentity (ln (const (Identity v)) s)

over :: Lens' s a -> (a -> a) -> s -> s
over ln g s = runIdentity (ln (Identity . g) s)

view :: Lens' s a -> s -> a
view ln v s = getConst (ln Const s)
```

Figure 10: Definition of `set` and `view` for Lens' lenses.

value again. This is very inefficient because the data structure has to be traversed twice. Once for getting and once for setting. More suitable would be an update method which can apply a function to a view. However in practice such a function might want to fail, i.e. return a `Maybe` value as a result. The given definition provides this capability and also lifts the resulting source to the same `Maybe` data type which correctly reflects failed updates to the view in failed updates to the source. A lens, as defined in Figure 9, should be viewed as more of an access provider for applying functions to a view.

3.5.1 Defining `set` and `view`

With this in mind we can define the `set` and `view` function from the simple definition to provide the exact same functionality by utilizing `Identity` and `Const`. For the remaining explanation the simpler lens type `Lens'` will be used. Besides the two functions from above a simple update function will also be presented. For now, we assume that a lens correctly lifts the function from it is first argument to the source type.
`set`, `view` as well as the update function `over` are defined in Figure 10. In the `set` function, the update with a new value `v`, namely `const (Identity v)`, is lifted to the type of the source `s` using the lens `ln`. This results in a new source which is wrapped into an `Identity`. Then `runIdentitiy` is used to peel off the constructor and the `set` function returns the new source. The update function `over` is practically the same as the `set` function, but applies `g` to the view before wrapping it into an `Identity`.
While `set` is quite easy to explain, `view` is more difficult. First the lens `ln` lifts the `Const` constructor from the type `a -> Const a a` to the type `s -> Const a s`. `Const` will take the view as an argument in the lens and contain it. Even though the source type `s` is still part of the type of the `Const` container, only the view is stored (see Section 3.4). Using `getConst`, the view is extracted and returned as the result.
We have now seen that the definition of lenses as a function is just as powerful as the

```haskell
data Point = Point { _x :: Int, _y :: Int }

x :: Lens' Point Int
x mod (Point a b) = fmap (\x' -> Point x' b) (mod a)
```

Figure 11: Definition of lens `x` for the data type `Point`.

simple lenses with two functions. `set` and `view` can be defined to work analogously to the simple definition. The beauty of defining the lens as a function is, that lens composition is then simply function composition. To see how function composition interplays with lenses we refer to the derivation of lenses by Edward Kmett [6] as a detailed explanation is beyond the scope of this paper.

3.6 Defining a Lens

As we have seen in section 3.1, lenses are mostly predefined or auto generated in the library. It is still unclear, however, what the definition of a lens looks like. Up to this point lenses were always just used as some abstract concept which lifts a function on a view to a function on a source. We will remedy this now. Figure 11 shows the definition of a lens `x` for the `_x` value of a `Point` data type. The argument `mod` is the modifier function of type `Int -> f Int`, where `f` is a functor. This could be the function `const (Identity v)` from the definition of `set`, for example. `Identity` would be the functor `f` in this example. Next the right hand side will be discussed. The `fmap` function is used to lift the function `(\x' -> Point x' b)` from the type `Int -> Point` to `f Int -> f Point`. Lastly the new view value is passed which is the result of `mod a` with the type `f Int`. Also note that the `fmap` function is defined for `f`. If `f` is `Identity` then the `fmap` function from Figure 7 is called. To conclude we note that there are no longer the functions `put` and `get`. However the `set` and `view` functions still have the same result type as in the simple definition of a lens. Therefore we could define `put = set ln` and `get = view ln` for some lens `ln`, to achieve something similar to the `put` and `get` components of the simple definition.

4 Conclusion

During the course of this paper, a simple definition of lenses was presented. Next the concept of lens classes was introduced and the corresponding lens laws were first presented and then motivated. Section 3 concerned itself with the Haskell library `lens` by showing the simplicity and convenience of its usage, followed by a more detailed look at the implementation. When discussing the implementation, the slightly different definition of lenses by Twan van Laarhoven was presented and motivated. We have also shown the benefits of the van Laarhoven lenses, i.e. simple composition and polymorphic record updates. There are many more advanced concepts in the `lens` library which allow for very powerful features. One possibility is to require that the underlying data type is not

only a `Functor` but an instance of the `Traversable` type class. This allows for more than one focused field. In the provided simple explanation, only views focusing a single field in a data structure were discussed. With traversals this can be extended to any number fields as described in [6]. It should be mentioned that `lens` is by far not the only lens library [22, 20] and has been criticized [21] for being over-engineered. Although the functions like `view` and `set` are easy to use, their types are very hard to understand when compared to the simple lens definition in Figure 2 which makes the library less user friendly. Especially since most of the types of the functions provided by `lens` are often generalized further. To conclude, lenses in functional programming can be very useful as they solve problems like the impractical record update syntax of Haskell.

References

[1] Sebastian Fischer, Zhenjiang Hu, and Hugo Pacheco. "A clear picture of lens laws —Functional Pearl—". In: *Proceedings of the 12th Conference on Mathematics of Program Construction (MPC 2015)*. Springer Verlag, 2015. URL: `http://sebfisch.github.com/research/pub/Fischer+MPC15.pdf`.

[2] John Nathan Foster. *Bidirectional programming languages*. `http://www.cs.cornell.edu/~jnfoster/papers/jnfoster-dissertation.pdf`. 2010.

[3] John Nathan Foster et al. "Combinators for bi-directional tree transformations: a linguistic approach to the view update problem". In: *ACM SIGPLAN Notices* 40.1 (2005), pp. 233–246.

[4] Edward Kmett. *GitHub repository of the lens library*. `https://github.com/ekmett/lens`. June 2015.

[5] Edward Kmett, Russell O'Connor, and Morris Tony. *GitHub repository of the data-lens library*. `https://github.com/roconnor/data-lens`. June 2012.

[6] Edward Kmett et al. *Derivation page on the wiki of the lens GitHub repository*. `https://github.com/ekmett/lens/wiki/Derivation`. Mar. 2015.

[7] Edward Kmett et al. *Example page on the wiki of the lens GitHub repository*. `https://github.com/ekmett/lens/wiki/Examples`. 2015.

[8] Edward Kmett et al. *FAQ page on the wiki of the lens GitHub repository*. `https://github.com/ekmett/lens/wiki/FAQ`. Mar. 2015.

[9] Edward Kmett et al. *Operators page on the wiki of the lens GitHub repository*. `https://github.com/ekmett/lens/wiki/Operators`. Mar. 2015.

[10] Edward Kmett et al. *Overview page on the wiki of the lens GitHub repository*. `https://github.com/ekmett/lens/wiki/Overview`. Mar. 2015.

[11] Twan van Laarhoven. *CPS based functional references*. `http://twanvl.nl/blog/haskell/cps-functional-references`. July 2009.

[12] Twan van Laarhoven. *Isomorphism lenses @ONLINE*. `http://twanvl.nl/blog/haskell/isomorphism-lenses`. May 2011.

[13] Twan van Laarhoven. *Overloading functional references*. http://twanvl.nl/blog/haskell/overloading-functional-references. Sept. 2007.

[14] Twan van Laarhoven. *References, Arrows and Categories*. http://twanvl.nl/blog/haskell/References-Arrows-and-Categories. Nov. 2007.

[15] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. 1st. San Francisco, CA, USA: No Starch Press, 2011. ISBN: 1593272839, 9781593272838.

[16] Conor McBride and Ross Paterson. *Control.Applicative module documentation*. https://hackage.haskell.org/package/base-4.7.0.0/docs/Control-Applicative.html. 2005.

[17] Russell O'Connor. "Functor is to Lens as Applicative is to Biplate: Introducing Multiplate". In: *CoRR* abs/1103.2841 (2011). URL: http://arxiv.org/abs/1103.2841.

[18] Russell O'Connor. *Polymorphic Update with van Laarhoven Lenses*. http://r6.ca/blog/20120623T104901Z.html. June 2012.

[19] Ross Paterson. *Data.Functor.Identity module documentation*. https://downloads.haskell.org/~ghc/7.8.3/docs/html/libraries/transformers-0.3.0.0/Data-Functor-Identity.html. 2001.

[20] Henning Thielemann and Luke Palmer. *Hackage page for the data-accessor package*. http://hackage.haskell.org/package/data-accessor. May 2014.

[21] Sebastiaan Visser. *Why I don't like the lens library*. http://fvisser.nl/post/2013/okt/11/why-i-dont-like-the-lens-library.html. Oct. 2013.

[22] Sebastiaan Visser et al. *Hackage page for the fclabels package*. https://github.com/sebastiaanvisser/fclabels. May 2014.