Remote Lensing

Lyndon P. Meadow

B.S. Mathematics, University of Kansas, 2018

Submitted to the graduate degree program in Department of Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Computer Science.

Committee members

Matthew Moore, Chair

Perry Alexander, Member

Prasad Kulkarni, Member

Date defended: July ??, 2021

The Thesis Committee for Lyndon P. Meadow certifies
that this is the approved version of the following thesis :

Remote Lensing

Matthew Moore, Chair

Date approved: July ??, 2021

Abstract

Hopefully, you are reading “ku-thesis.pdf”. If you are reading a file with some other name, then there’s been a goof up.

I want you to use LaTeX. I think doing anything else is finger painting. I’ve watched 20 MA and PhD theses written with this template and I know a bright student will succeed and be happy with the result. If you have a laptop computer and you want to show me some errors, I can give you 5 or 10 minutes. But not much more...

CRMDA offers some workshops and I teach graduate courses in which we show how to use LaTeX. If you find something wrong with kuthesis.cls, I will be glad to try to fix it.

I don’t have time/ability to answer a million email questions about LaTeX or LyX. If you are stuck on a tough problem, I may be able to help, but if I have to exert myself, you should know I expect you to give me a t-shirt and/or add me to the acknowledgments in your dissertation. (I’m not joking on that.) I spent about 20 hours helping a KUMC student fix his LaTeX and he assured me I’ll get a big discount on treatment for any liver illnesses I may endure in the future. (I’m joking about that.) I did get a medical center license plate holder, though.

Paul Johnson <pauljohn@ku.edu>

Prof. Political Science & Director, Center for Research Methods and Data Analysis

2019-02-01

And that’s the end of the abstract.

Acknowledgements

I would like to thank all of the little people who made this thesis possible. Sleepy, Dopey, Grumpy, you know who you are.

Contents

List of Figures

List of Tables

Chapter 1

The Only Chapter

Abstract

Initial Attempt at Using University Package to Write my Initial Drafted Documentation from Microsoft Word.

1.1 Draft

This story begins with looking for overall abstractions between remote object systems and being curious as to seeing if adding an ordered lens-like structure was possible between multiple systems. Even if the abstraction is nothing more than a wrapper or boilerplate for existing code, it sparked my curiosity if these remote systems could stimulate or in fact provide the simplicity of lenses to an end user. And that these abstractions were representable of total systems in existence, due to their popularity with classical lenses. As well as the fact that several online server applications attempt to bring this boilerplate into their own code with different levels of effectiveness across their own systems.

This led to some initial questions and concerns, mainly what a lens is, what a lens is in a programing language – specifically Haskell, and what are the existing standards for what constitutes the different categories of lenses that exist in the broad community.

Keep in mind that Lenses are about solving a bijective dictionary problem, as so far as to provide convenience to the end user. Most commonly in the functional community lenses are abstractions that are popular to provide an object-oriented accessor notation to an object's elements. In the sense that the functions that wrap data into record structures are able to be acted on by functions that unwrap or rewrap data in a convenient way. There are extensional ideas that relate to this such as Traversable structures and those should be representable as well. But that is not the primary focus of this research, although in the case of Remote Monad, the Traversable is very much implementable abet with some extra efforts.

Lenses are in briefest summary a mathematical tool to manipulate dictionaries and nested dictionaries. They came as an expansion to the mostly solved bijective dictionary space and an attempt to further expand out the work with new levels of abstractions. Every lens has a set and a get function that it bundles together, and are molded after object-class accessors. And essentially accomplish the goal of retrieving or mutating values inside of those kinds of objects. As to what the community considers a well-behaved lens, this in itself can be varied and ends up becoming unclear based on who's implementation and rationalization you read on. Much like much of computer science terms end up thrown around interchangeably, meanings are lost and altered for established words, and so therefore it becomes important to establish what each body of work considers to be a lens and a well-defined lens.

For starters what separates well-defined from not? A well-defined function follows what are generally expected rules and results that the average user might expect. And a function that is not well-defined will possibly posses unexpected behaviors that the user doesn't intend to occur. A clever way to imagine this is to imagine the standards in place for the C++ compiler. There are well-defined and expected results if one were to define a simple addition function. However, there are no defined results as to what should happen if your main function does not have a return statement embedded at the end. So in this way a well-defined lens has some expected behaviors that will always hold true if it follows a communities prescribed expectations. But unfortunately, the manual itself is ill-defined based on the source that you prescribe, therefore it is important that in this body of work the manual for a well-defined lens is included. So that it becomes possible to judge in this body of work if the lens fits what we wished from its behaviors. And ultimately anything that isn't defined should be disregarded in the same way that one might disregard the C++ compilers handling of no return statement in main. We are ultimately concerned with defined behaviors as they relate to defined expectations.

Speaking in layman's terms: Get-Put Law: Putting back what you Get results in the same object. Put(Get (s) s) = s Put-Put Law: How one Put will Override a previous Put. Put(Put(a v) s) = Put(a s) Put-Get Law: Getting back what you Put results in the new object. Get(Put(a s)) = a

Are usually what are discussed when referencing classical lens implementations in computing languages for well-defined behaviors. Removing laws or altering them varies based on sources of work read.

~~*Mandatory Citation So Bibliotex Doesn't Complain*~~ Gelman et al. (2003); Pinheiro & Bates (2000)

Compare and Contrast.

Describe Remote-Monad:

view2 :: (FromJSON b) => (String -> String) -> (RemoteValue a) -> (RemoteMonad b)

set2 :: (String -> String) -> String -> (RemoteValue a) -> RemoteMonad (RemoteValue a)

over2 :: (FromJSON t, Show a) => (String -> String) -> (t -> a) -> (RemoteValue a) -> RemoteMonad (RemoteValue a)

These are the types for the set of functions that seek to emulate the Get and Put laws that lenses experience. And show that although they do not share the same internal structure at their locally implemented counterparts, that they will share a similar external behavior.

Such that as in the abstract syntax that:

Set2 (View2 (s) s) = s

Set2(Set2(a v) s) = Put(a s)

View2(Set2(a s)) = a

Is a similar result, however it is worth noting, that due to the fact that we are dealing with a remote connection that the connection must be passed around – in addition to the object you want to manipulate, where inside the object you wish to lens to, and a value if using the Put function.

Now, one more important thing to note, is while in Haskell you normally are manipulating and creating new objects every time you run a function or assign it to a value. When manipulating Javascript, you are never creating a new object on the other side. So, any code you'd want to write with these methods on the Haskell-side, would need to keep in mind that you never are dealing with a fresh object. The connection will always make sure you are pointing to the one and only original Javascript object you are trying to manipulate.

A key distinction on the improvements is that this removes any sort of backend coding, where the user is directly manipulating the delicate remote monad commands. And instead has a predefined way of accessing, editing, and setting values with minimal contact to the ideas and command structure beneath.

So much like lenses were used to re-solve the local dictionary problem.

Remote lenses here are used to re-solve the remote dictionary problem.

And this is a clear method that shows how it can be solved.

You get abstraction and type safety on every component of the command, and modularity that clearly expresses your intent to the reader of your code.

Example:

With basic syntactical sugar, with view2 as ^. And composition of remote lens fields as >>> . . . you get a clear expression that shows to anyone else on your coding team what you were doing.

```
f :: Double <- person ^. nest >>> nest2 >>> extra3
```

f :: Double <- person ^. nest >>> nest2 >>> extra3

```
-- This is visible in the paper and to the Haskell implementation.
f :: Double <- person ^. nest >>> nest2 >>> extra3
```

```
f :: Double <- person ^. nest >>> nest2 >>> extra3
```

Where for the remote-object of person, I am accessing 2 layers into it's fields to retrieve a double value from extra3.

The normal syntax for this would be represented as:

$$Set_2\,(View_2\,(s)\ s) = s$$

view2 (extra3(nest2(nest))) person

Which expands out to in the weeds as: —

g <- constructor $ JavaScript $ pack $ (extra3(nest2(nest))) (var_text person) procedure $ var g

Repeat this for Set, Over, and for the examples of Lens Laws? (Yes)

Chapter 2

The Haskell Lens

Lenses are restudy of the old bidirectional transformations (bx) problem that was already solved through existing dictionary and record systems, but their variation to this approach has been the subject of much study in the attempts to simply for the end user the ways that they can focus in on the individual parts of structures.

Important note, most definitions have the direction established. i.e. where one is going. So only arguments are the structure itself in view case, with just structure and item in set case. Which implicates that the overall pattern-matching system which is the individually implemented lens for each focus in a structure, has been embedded before the configuration of *Set* and *View* respectively in its descriptive rules. In practicality the lens is an active and changing argument that the conventional programmer sets themselves when they are establishing where they'd like to mutate a structure. Depending on the source, it is perfectly possible that the lens laws themselves are defined with that extra argument since including it in the hard coding is so common place.

$$
\begin{aligned}
Set\ w\ (View\ w) &= id \\
View\ (Set\ w\ p) &= p \\
Set(Set(w\ p)\ p' &= Set(w\ p')
\end{aligned}
$$

Figure 2.1: Lens Laws
Set-View, View-Set are well-behaved.
Set-Set is very well-behaved.

The structure is so general, polymorphic that we need to establish laws that form what lenses are, and what rules that they should follow in Figure 2.1. This is important since there are lenses that do not follow any strict rules and arbitrarily apply bidirectional transformations in ways that aren't consistent. A prime example would be the implementation of partial functions such that the View-Set rule holds but the Set-View doesn't and vise-a-versa for Set-View and View-Set respectively.

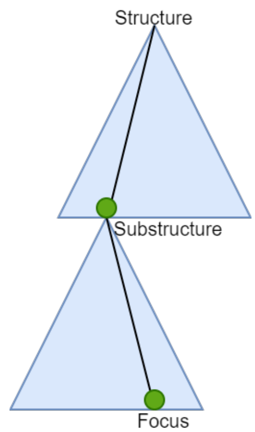

Figure 2.2: Lens Diagram

Here is a basic idea of what a simplistic lens accomplishes in Figure 2.2, by providing a means to pattern match internally the ability to navigate withing a structure and even its substructures are possible. As long as the substructure also has been formed in away for pattern matching to also work on it. This creates a composability of the lenses so that it becomes possible to find a focus even if it is nested two or more structures deep. Allowing for viewing or setting that focus, depending on the wish of a given programmer. Essentially embedding a lens friendly structure into a larger one, doesn't get rid of the ability to lens into it. Provided that the larger structure also has maintained a means to focus onto that substructure with it's own lens.

There are some basic Haskell types as well as examples that show that the idea of lenses applies to many sorts of structures in the programming world, including modifications or inspections of two-tuple structures — which is shown below.

```
view :: w -> p

set :: w -> p -> w
```

```
view_a :: (a,b) -> a
view_a (x,y) = x

set_a :: (a,b) -> a -> (a,b)
set_a (x,y) x' = (x',y)
```

In these sorts of cases, the general idea of pattern matching to obtain results is explicit and summarized inside of the code itself. However, the lens in which to focus on a specific element of the two-tuple is implied from the direct call to the pattern matching system provided by Haskell's compiler. This holds true for both view_a and set_a, where the inspection of the two-tuple is enough to view or set the contents of the structures respectively.

Now to explain further, the ideas should be given some space in what a typical lens is when locally implemented, since it is fine to talk about the wide variety that might be associated with injection and projection rules in homotopy type theory. It also is noted here that the wider categories or attempts of abstraction to lens will also be avoided since they aren't relevant to the contents of the papers which includes generalized discussions on asymmetric lenses, bimorphic lenses, and likely whatever else has been proposed in the community at this point and time. And rather than make any bold claims about polymorphic lenses, topics will be restricted to monomorphic occurrences instead.

Chapter 3

The Remote Monad

The purpose of this chapter is to expand past covering the mathematical background and the historical Haskell programming context. In this primary article will show both an introduction to remote lenses as well as their implementation with the *Remote Monad*. As described the two primary components of both programmatic and mathematical lenses are the *Set* and *View* which produce as a result of applying them both at the same time a command referenced as *Over*. This now allows us to describe a variation that copies on the structure and intent of these operators over a remote connection between a client and a host device, the host running Haskell and the client running some other language. This variation which is the main subject of study of this paper will be referred to as the *Remote Lens* and will be fleshed out starting now and until the end of this manuscript. (Clean up this sentence structure)

To define the individual components of *Remote Lenses*, we begin by providing the relevant abstract type definitions below.

```
RemoteView :: (String -> String) -> (Wrapped a) -> (ReturnType b)

RemoteSet :: (String -> String) -> String -> (Wrapped a) -> ReturnType (Wrapped a)

RemoteOver :: (String -> String) -> (t -> a) -> (Wrapped a) -> ReturnType (Wrapped a)
```

We extend the abstract syntax these modifications to the original well-formed lens rules.

$$RemoteSet(acc\ RemoteView(acc\ obj)\ obj)\ \equiv\ obj$$

$$RemoteSet(acc\ item\ obj)\ ;\ RemoteSet(acc\ item2\ obj)\ \equiv\ RemoteSet(acc\ item2\ obj)$$

$$RemoteView(acc\ RemoteSet(acc\ item\ obj))\ \equiv\ item$$

Where ";" denotes the sequence of operation and "≡" denotes congruence but not equality.

Where "*acc*" is the accessor to the structure, "*obj*" is the structure, and "*item*" is some substructure.

Where these laws show the View-Set (Modifying a subpart such that it is the same as it was before), Set-Set (Modifying a subpart with "a", then with "b" will be the same as if you just modified with "b"), Set-View (Modifying a subpart, then viewing will be the same as just viewing the modification itself). (These are the analogs, get rid of parenthesis. Represent these extended to a remote setting. More detailed explanation in previous chapter with equations, and reference those here TO DO)

The essentials of what is required is aptly similar to the definitions seen in the elementary Prelude construction of Haskell lenses. With that in mind, the first argument typically taken by any lens operator is a function that will access a deep rooted parameter inside of a structure. In the same vein, the *Remote Lens* variant accomplishes this with an abstraction of the *Remote Object's* nested fields that are to be queried into once the command is sent from host to client. The primary difference in how this is handled by Haskell, is that rather than simply pattern matching into the already existing structure, a completely different approach is needed instead.

This approach is best summarized as a conflict between the fundamental methods used in typical lens methods that rely strongly on internal pattern matching for the composability of lens, and the existence of a remote connection that prevents that internal pattern matching to take place. This can be essentially thought of any scenario where accessing a structure isn't possible inside of the host system, and the necessity to query a client system takes place.

This will involve internally concatenating a conjointment of the parameter. Each of the parameters that normally would be accessed with a composition of pattern matching functions, must instead be accessed with a composition of the *Remote Object's* internal parameters. It is fair and equally valid to imagine representing the target object as a remote record with the means to access the contents completely dependent on how that record was implemented inside of the client's language.

Another shared argument is the structure that will be accessed by the lens operator. Similarly, what is passed around in the *Remote Lens* is a reference to the location of what is to be accessed upon sending the constructed command. For these reasons directly accessing the information is not possible, since one is manipulating only a reference to the actual object on the client's side. That the careful composition method mentioned in the former paragraph was needed. Given that bundling the reference to the *Remote Object* along with the preformed command to operate on the *Remote Object* are absolutely necessary, otherwise none of the methods to view or alter the contents of that structure are at all possible to create. (Try to rewrite these sentences to flow better)

The final shared parameter is the return type from the result of applying the lens operator to the structure that one is inspecting. This is also represented by the *Remote Lens's* return type, where for a REMOTEVIEW the return type will be the desired queried item wrapped in the remote operator that allowed for the operation to take place and for REMOTESET/REMOTEOVER the return type will be the reference to the altered structure wrapped up in the same remote operator. The only major consideration when manipulating the returned structure, is that one must assign the type of the expected viewed parameter or trust that there is sufficient information that Haskell can self-interpret the type for itself.

Now it becomes possible to introduce the primary example that this manuscript will cover, specifically the *Remote Lens* as it relates to the *Remote Monad* the construction of it's key functions represented by the following types.

```
RemoteView :: (FromJSON b) => (String -> String) -> (RemoteValue a) -> (RemoteMonad b)
RemoteSet :: (String -> String) -> String -> (RemoteValue a) -> RemoteMonad (RemoteValue a)
RemoteOver :: (FromJSON t, Show a) => (String -> String) -> (t -> a) -> (RemoteValue a)
                                          -> RemoteMonad (RemoteValue a)
```

These are the types for the set of functions that seek to emulate the methods of *View*, *Set*, and *Over* that have been instantiated specifically for a *Remote Lens* based on using the *Remote Monad*. Despite that they do not share the same internal structure of their locally implemented counterparts, that in their behavior they will accomplish the same goal, which is to resolve standard dictionary manipulation. This wrapper style may not have been used by others, but it is specifically implemented to show that it can be viewed as a universal design pattern. Rather an attempt to show that for any client host system in which there is a an interaction between a host language, client language, and a remote structure — that a fully formed method of communicating between the two is presentable in this way.

Which can be viewed as the following implementations.

```
RemoteView accessor object = do{
        g <- constructor $ JavaScript $ pack $ accessor (var_text object)
        procedure $ var g
}

RemoteSet accessor new_item object = constructor $ JavaScript $ pack $
                (accessor (var\_text objectName)) ++ " = " ++ new_item

RemoteOver accessor my_function object = do{
        item <- RemoteView accessor object
        let new_item = show $ my_function item
        RemoteSet accessor new_item object
}
```

As a reminder, it is worth noting that due to the fact that we are dealing with a remote connection that the connection must be passed around — in addition to the structure you want to manipulate, where inside the structure one wishes to lens to, and a value if using the Set function.

Now, one more important thing to note, is while in Haskell you normally are manipulating and creating new objects every time you run a function or assign it to a value. When manipulating Javascript, you are never creating a new object on the other side. So, any code you'd want to write with these methods on the Haskell-side, would need to keep in mind that you never are dealing with a fresh object. The connection will always make sure you are pointing to the one and only original Javascript object you are trying to manipulate.

A key distinction on the improvements is that this removes any sort of backend coding, where the user is directly manipulating the delicate remote monad commands. And instead has a predefined way of accessing, editing, and setting values with minimal contact to the ideas and command structure beneath. One gets an abstraction and type safety on every component of the command, and modularity that clearly expresses your intent to the reader of your code.

Examples:

With basic syntactical sugar, with *RemoteView* as ^. And composition of remote lens fields as >>> ... one gets a clear expression that shows to anyone else on ones coding team what one was doing.

```
f :: Double <- person ^. nest >>> nest2 >> extra2
```

Where for the remote-object of person, I am accessing 2 layers into it's fields to retrieve a double value from extra3.

The normal syntax for this would be represented as.

```
RemoteView (extra3(nest2(nest))) person
```

Which expands out to in the weeds as.

```
g <- constructor $ JavaScript $ pack $ (extra3(nest2(nest))) (var_text person)
procedure $ var g
```

Which shows the virtualization of machinery from each layer to the next in this system, and that whilst the syntactical sugar for *RemoteSet* and *RemoteOver* has not been shown, these share the same levels of abstractions. For completeness, examples of their basic calling is provided now.

```
RemoteSet (extra3(nest2(nest))) "4" person


RemoteOver (extra3(nest2(nest))) (\ x -> (x + 1::Double)) person
```

FP Complete Lens Exercise Adapated To *RemoteMonad* And *Remote Lens*.

– Avoid, previous chapter mentions. The purpose of this chapter is ++ BLANK, In this chapter we will cover the historical programming context and their impelementaiton in the remote monad.

– Not as described,

– Operations, /imp but more defined \texttt{blah}

– For Code Words like RemoteView \texttt{blah}

– View and Set sequenced.

– Minted Package try again, and force it into Verbatism if I can't.

– Not applying them both at the same time.

– Over is a derived operation.

– Colon to introduce text is bad. End the previous sentence with a period.

– Talk about why these are congruent but not equal, in original math they are equal.

– Write more information why a different approach is needed.

– Where is needed less often, try deleting them.

```
address objectName = objectName ++ ".address"
street objectName = objectName ++ ".street"
city objectName = objectName ++ ".city"
name objectName = objectName ++ ".name"
age objectName = objectName ++ ".age"

wilshire :: String
wilshire = "\"Wilshire Blvd\""

aliceWilshire :: RemoteValue a -> RemoteMonad (RemoteValue a)
aliceWilshire newperson = RemoteSet (address >>> street) wilshire newperson

getStreet :: RemoteValue a -> (RemoteMonad String)
getStreet newperson = RemoteView (address >>> street) newperson
```

```
birthday :: RemoteValue Int -> RemoteMonad (RemoteValue Int)
birthday newperson = RemoteOver age (\ x -> (x + 1::Int)) newperson

getAge :: RemoteValue a -> (RemoteMonad Int)
getAge newperson = RemoteView age newperson

exercise1 :: Engine -> IO ()
exercise1 eng = do{
        send eng $ do{
                command $ call "console.log" [string "starting..."]
                render $ "Exercise1"
                newperson <- initializeObjectAbstraction2 "alice" alice
                aliceWilshire newperson
                mystreet_person <- getStreet newperson
                birthday newperson
                herNewAge <- getAge newperson
                render $ "mystreet_person: " ++ mystreet_person
                render $ "herNewAge: " ++ show herNewAge
}
}
```

References

Gelman, A., Carlin, J. B., Stern, H. S., & Rubin, D. B. (2003). *Bayesian Data Analysis, Second Edition*. Chapman & Hall, 2 edition.

Pinheiro, J. C. & Bates, D. M. (2000). *Mixed-effects models in S and S-PLUS*. Springer.

Appendix A

My Appendix, Next to my Spleen

There could be lots of stuff here