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

The contents of this manuscript are intended to show a deeper look at the manipulation of information across a remote distance by providing a means to safely edit and manipulate data with the type-safety of lensing. In standard practice most programmers rely on complex methods to guarantee that they are editing without error, or in other cases they omit effort and use simple methods without any guarantees on errors. Taking the host side to be the strongly-typed language with lensing properties, and the client side to be a weakly-typed language with minimal lensing properties, this work will contribute to the existing body of research that has brought lenses from the realm of math to the space of computer science. And shall give a formal look on remote editing of data in type safety with remote monads and their variants.

Acknowledgements

A thanks to Dr. Matthew Moore for his inspiration and continued help throughout the entirety of my graduate career, from independent study on Homotopy Type Theory to this Master's Thesis. An appreciation for Dr. Andy Gill for letting me partake in this research, from my own interests in the topic. Dr. Man Kong for believing that I could turn out to be a decent GTA, resulting in the funding for this paper. And a thanks to Dr. Gary Minden and Dr. Perry Alexander for teaching the more formative classes in my decision to attend graduate school.

Contents

List of Figures

Chapter 1

Introduction

The origin of this work begins with looking for overall abstractions between remote object systems and investigating whether adding an ordered lens-like structure was possible between multiple systems. These remote systems could stimulate or in fact provide the simplicity of lenses to an end user, such that these abstractions were representative of systems already in existence. Due to the popularity of classical lenses, it was discovered that online applications attempt to bring this boilerplate into their own code with different levels of effectiveness across their own systems.

To begin with, Foster et al. (2007), this field has been strictly derived from the premise of database systems, with even this primary source having constrained itself to a strict representation in the fundamentals. The fact of this is not a surprise when considering that the original well-behaved lens laws were first formulated by Benjamin C. Pierce in the context of database manipulation, Pierce & Schmitt (2003); Foster et al. (2003). From this series of work, Lenses originated from studying bidirectional transformations, as providing convenience to the end user as well as layers of type safety. Most commonly in the functional programming community lenses are popular abstractions to provide an object-oriented accessor notation to an object's elements: the functions that wrap data into structures are able to be acted on by functions that unwrap or re-wrap data in a convenient way. There are extensional ideas that relate to this such as Traversable Structures, but that is not the primary focus of this research, although in the case of Remote Monad, the Traversable is very much implementable — albeit with some extra efforts.

There are several types of lenses that bridge the gap between remote and local interactions already in existence, with varying degrees of depth of implementation. From the broad categories of c-lenses to the category of d-lenses, Johnson & Rosebrugh (2013) use the concepts of opfibra-

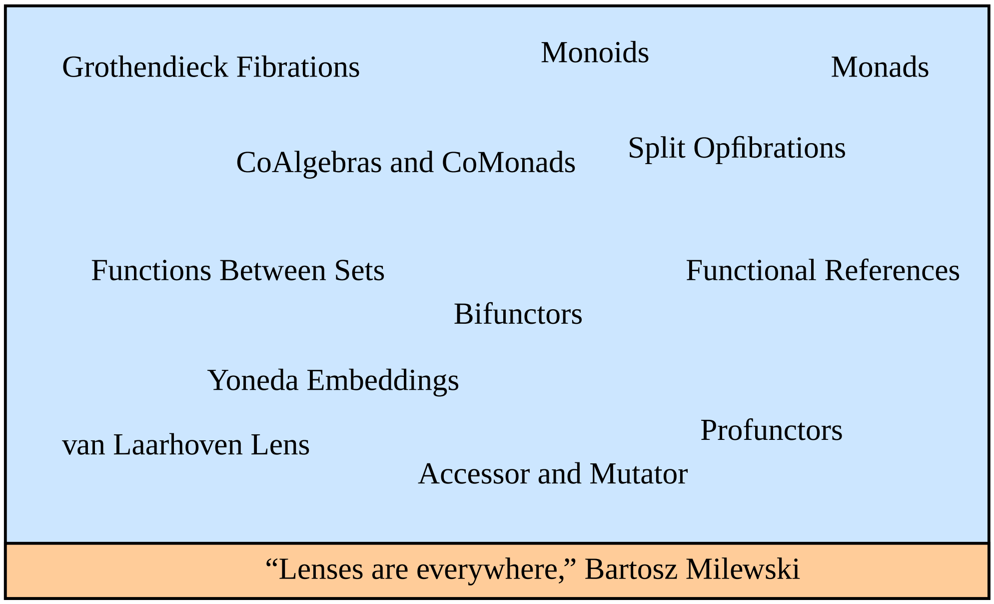

Figure 1.1: Variety of Lenses

tions to encapsulate the nature of transition information between a system and a database. The previous work derives its motivation from database systems such as SQL and the manipulation of table or record information, i.e. the original derivation of the c-lens from the categorical notion of Grothendieck opfibration, Grothendieck (1971), for a solution to the view update problem for functorial update processes, Johnson et al. (2011). From the idea that a host can only see a certain amount of information from the client side's database system, but both must maintain a consistently updated view parameter, Ahman & Uustalu (2014).

In fact, the advancements in category theory are heavily rooted in earlier works that bridged the gaps between algebraic topology and algebraic geometry, Grothendieck (1960). Which nearly paralleled the formalization of Beck's work in the formalization of the monadicity theorem which established when a functor is monadic, Beck (1967). These and multiple many discoveries gave rise to the derivation of several important concepts, including bifibrations and the monad itself, the details of which are not in the scope of this manuscript. Which gives rise to the formality of lenses despite their very humble beginnings, Foster et al. (2003). The notions of what can be considered entirely a lens or a portion of the children a lens structure, is illustrated in Figure 1.1.

Much of the general theory has assumed a typical database relationship such as SQL or other table based services, this is based from the hard coded implementations that preceded these constructed ideas, which are hand tailored in works such as Bohannon et al. (2006). Several of these works are representative of these direct methods and their specific heritage in database problems which would be too numerous to cite here, but an impactful reference on source-to-target and target-to-source transformations is Fuxman et al. (2006). Including a wide range of work finding uses for lenses outside of their typical domain, most notably Boomerang a bidirectional programming language for textual strings, Barbosa et al. (2010), with further development in Optician, Miltner et al. (2017).

Which is why there is still open space for pseudo-implementations of lenses that haven't received much coverage if any in literature, perhaps because they break from all of the much higher level concepts and exert effort in the more rudimentary space of custom constructed Domain Specific Languages (DSLs). A DSL that is built over existing systems, not from the ground up with an intention of being able to represent much deeper concepts. It is a focus on practical designing patterns that are concerned with construction in the execution space. Since one doesn't need more than an understanding of the well-behaved lens laws to be able to put them into application for remote and local systems.

Additionally, DSLs already have their own unique sub-field in the lensing community with respect to the bidirectional transformation problem Czarnecki et al. (2009). This topic focuses on the API advancements provided by the Remote Monad design pattern for remote calls to external systems for the purposes of Remote Procedure Calls that have their commands executed remotely before returned locally, Gill et al. (2015). The Remote Lensing is in the same vein as database lensing, but it distinguishes itself by being primarily focused on Remote Procedure Calls rather than the typical database view update problem — of the latter this is not an attempt to expand on. This manuscript is focused on bringing in the concepts of Remote Lenses which are neglected in documented literature, to an explicit focus on their usability and ease of implementation from the fundamental well-behaved rules, and the capacity to scaffold them onto existing API.

Chapter 2

The Haskell Lens

Lenses are a restudy of the old problem of bidirectional transformations (bx) that were historically solved using dictionary and record systems. It is still an active field of study, with many connections to different sub-fields of computer science. Explicitly described, "*bx* are a mechanism for maintaining the consistency of two (or more) related sources of information", Czarnecki et al. (2009). The nature of bidirectional transformations allows them to be applied to domain specific languages and the host language itself, as a means to translate between run-time values.

From the implementation of lenses in the *bx* programming community there has been in recent years a new passion for their joint implementation in a larger variety of scenarios than they were initially imagined for. The study has broadened to the attempt to abstract and simplify over many sorts of structures, including their individual parts. The attempt at generality provides for opportunities in anything that might be presentable as a meaningful *bx*.
Lenses are characterized by the satisfaction of abstract laws and rules, established fully in Pierce & Schmitt (2003). Since there exist so called "not well-behaved" lenses that only follow some proper subset of the well-behaved lens rules and arbitrarily apply bidirectional transformations in ways that aren't consistent. Such "not well-behaved" lenses are beyond the scope of this manuscript, we will only deal with well-behaved lenses and their applications.

$$\begin{aligned} Set\big(obj,\, View(obj)\big) &= obj \\ View\big(Set(obj,\, item)\big) &= item \\ Set\big(Set(obj,\, item),\, item'\big) &= Set(obj,\, item') \end{aligned} \tag{2.1}$$

It is an important note, that most definitions fix the direction of the lens operators *a priori*, i.e. where the *Set* or *View* will traverse along a given object to reach the designated destination.

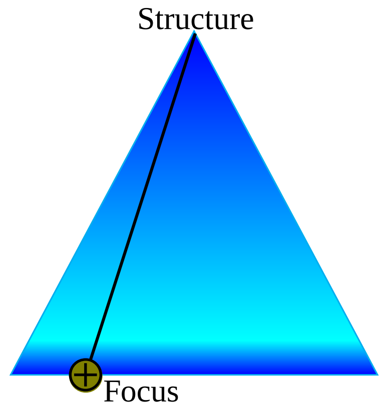

Figure 2.1: Lens Diagram

The only arguments to *View* are the structure itself, and with the only arguments to *Set* being the structure and item. The definitions assume 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 use, the lens is an active and changing argument that the programmer sets when they are establishing where they would like to mutate a structure. It is also possible to define the lens laws with an extra argument to determine the direction:

$$\begin{aligned} Set\big(acc,\, obj,\, View(acc,\, obj)\big) &= obj\,, \\ View\big(acc,\, Set(acc,\, obj,\, item)\big) &= item\,, \\ Set\big(acc,\, Set(acc,\, obj,\, item),\, item'\big) &= Set(acc,\, obj,\, item')\,. \end{aligned} \tag{2.2}$$

This exposes some additional structure which the more instantiated definition in Equation (2.1) had previously obfuscated.

Figure 2.1 illustrates a simplistic lens, showing a means to pattern match internally and a navigation of that structure to the desired focus with the help of a lens. The diagram implies that it is possible to navigate into the focus, provided that the focus is a similarly formatted structure. This is true, provided that both the primary structure and all associated substructures have been constructed in a way that allows for pattern-matching. If this is the case, then pattern-matching over the entire structure is possible. This "stacking" of structures, enables lenses to be composed so that it becomes possible to find a focus even if it is nested two or more structures deep. The

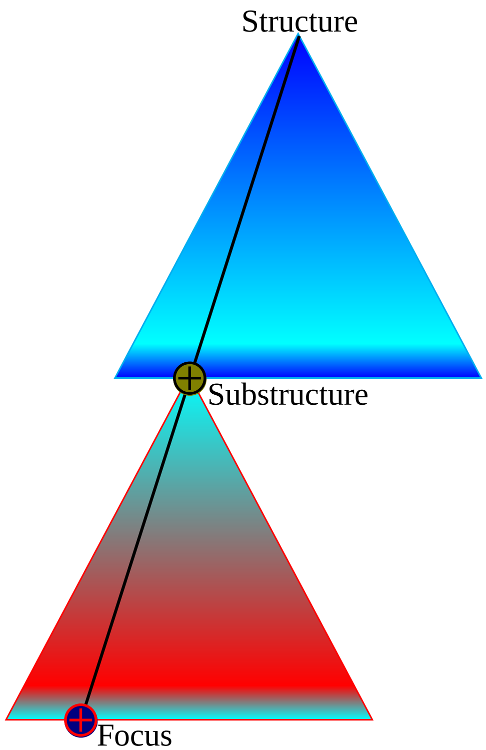

Figure 2.2: Diagram of Lens Composition

embedding of a lens equipped structure into a larger lens equipped and the resulting composition of lenses is illustrated in Figure 2.2.

There are some basic Haskell types as well as examples that show that the idea of lenses applies to many sorts of structures, including modifications or inspections of two-tuple structures as displayed in Figure 2.3. In simple cases, the general idea of pattern matching to obtain results is explicit and summarized inside the code itself. However, Haskell's compiler allows for implicit pattern matching to take place as well. The implicit calls to focus on specific elements of the two-tuple is handled in this manner, display in Figure 2.2. 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.

Lenses represent such a large sub-field of bx, and have been so generalized that a full view is well beyond the scope of this manuscript. The interested reader may refer to Czarnecki et al. (2009) for a good survey. These and many more topics will be restricted to monomorphic occurrences instead, whilst the rest will fall outside the scope of this thesis.

```
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)
```

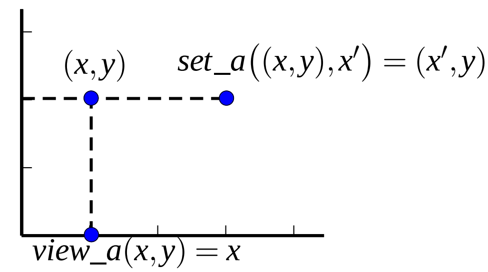

Figure 2.3: Elementary Haskell Lens

Expanding on how a lens may be implemented in Haskell, an important example to consider is how building lenses and operating on a structure work. Not an implicit lens like in Figure 2.3, but an explicit lens that allows access to a richer structure. One begins by initializing such a structure as follows below.

```
data MyStructure = MyStructure {
    MyName :: String
    , MyAge :: Int}
    deriving (Show, Eq)
```

The structure contains two important points of focus a String and an Int value, and one can make a lens with the ability to access either of the two elements. In similarity the two-tuple example, constructing an accessor to each element to view or set must be done individually. But in this case, there isn't a trick to get the Haskell compiler to do all the work, so a little bit of code is needed to break inside.

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

item_name :: MyLens MyStructure String
item_name some_function (MyStructure l r) =
                        (\l'->  MyStructure l' r) <$> (some_function l)
```

From this, there is a general structure that represents how a lens may be formed by the type abstraction of *`MyLens`*. Specifically instantiated the expansion for *`item_name`* would appear something along the lines of `String -> f String) -> MyStructure -> f MyStructure`. Which allows a set or view implementation to provide some functor that renders the inspection or modification of the structure in compliance with the well-behaved features of lenses. Another way to see this, is that upon inspecting the instantiation of $item\_name$, it is possible to spot out the initial pattern matching of the structure. The leftmost argument having $some\_function$ applied to it, before being injected back into the structure.

```
set :: forall s a. MyLens s a -> (a -> s -> s)
view :: MyLens s a -> (s -> a)

getName :: MyStructure -> String
getName = view item_name

setName :: MyStructure -> String -> MyStructure
setName = set item_name
```

The specifics of how *Set* and *View* are implementing their individual accesses to the structure aren't terribly important, they are only giving instantiation to the much more familiar usage and format of the well-behaved lens laws in Equation (2.1). This is made more clear in the $Name$ functions and type definitions, where after applying the lens to the set or view, then only the classic arguments are needed, since the lens has already focused on a specific area of the structure.

Chapter 3

Extending Lenses

As described, the two primary components of lenses are the *Set* and *View* functions which produce as a result of applying them both at the same time a command referenced as *Over*. When one applies *View* to the structure in Figure 3.1, an element of that structure is retrieved. We may then modify that retrieved element with some function before applying *Set* to that very same structure. This results in a new structure which accomplishes the same as having applied *Over* from the very beginning.

The application of *View* and *Set* in this sequence is equivalent to *Over*. For the a command sequence to be equal to *Over*, it must directly perform a given function onto the focus whilst in in the same singular injective process. This process of modeling the *Over* function can be best described as the following composition, $Set\Big(obj,\ F\big(View(obj)\big)\Big) \models Over(obj,\ F)$. For all valid objects, applying the desired function to the viewed object and then setting that modified object is equivalent to overing the object with that same function.

There exist many degrees of abstracting over elementary lenses like those provided in Figure 2.3. The documentation is not focused on the fuller expanses of the system of abstractions for lensing mechanisms inside of mathematics and programming, but instead finds relevance in the most simple formation of lenses which is now provided.

```
data BasicLens a b = BasicLens {
    viewer :: (a,b) -> a
    , setter :: (a,b) -> a -> (a,b)}
```

This shows the capacity to build the lens datatype from a structure that contains the view and set explicitly for a given focus. Additionally, the ability to retrieve both the viewer and the setter as independent but composable functions is maintained.

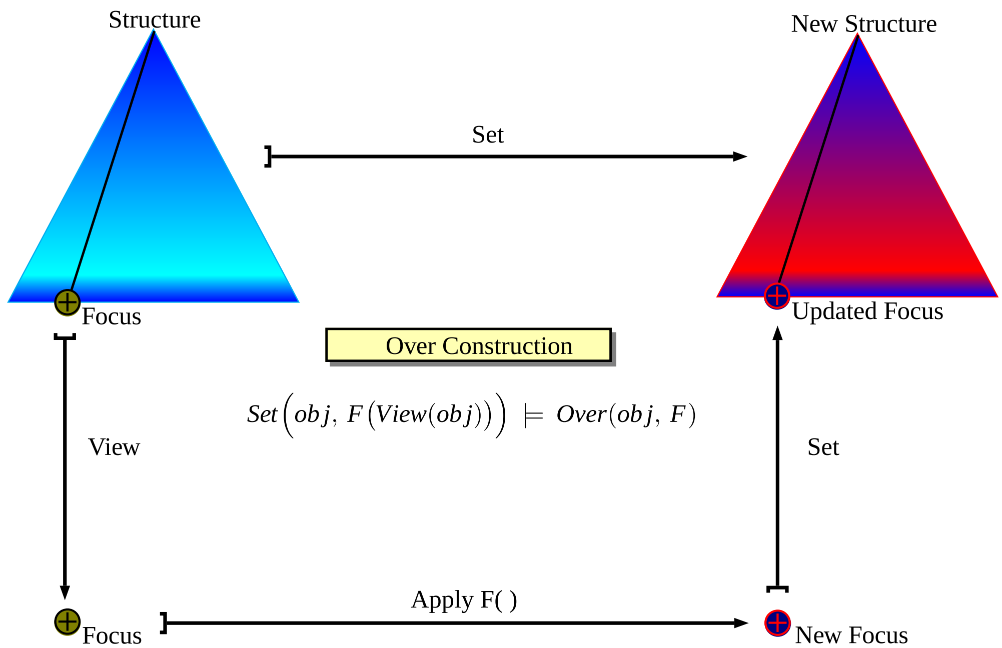

Figure 3.1: Expanded Lens

```
applyview ::  BasicLens a b -> (a, b) -> a
applyview (BasicLens viewer setter) s = viewer s

applyset ::  BasicLens a b -> (a,b) -> a -> (a,b)
applyset (BasicLens viewer setter) s = setter s
```

As shown above, `applyview` and `applyset` both use pattern matching to expose the internals of the elementary Haskell lens.

```
a_Lens :: BasicLens a b a_Lens = BasicLens {
    viewer = view_a
    , setter = set_a}
```

The initialized version of this simplistic lens structure, preforms it's role for the *Set* and *View* defined in Figure 2.3.

This leads to the assertion that although defined in heavy detail and with great investment in the proofing of algebraic geometry, category theory, and homotopy type theory — the hard coded versions of lenses do not need to subject themselves to these abstractions. In fact, the levels of proofing and rigor abstract away what lenses can be defined to do, such as when shoving a lens into a monadic container. Which is why only the well-behaved lens laws in Equation (2.2), are considered for the details of the manuscripts later chapters. If one has a curiosity if the pseudo-implementations of lenses discussed may be represented as standard lenses, then the continued reading is deeply recommended.

Chapter 4

Remote Lenses and The Remote Monad (Separate?)

In this chapter, we detail a novel extension of lenses to remote structures making use of the Remote Monad. The definitions provided in the previous chapters allow for one to describe a variation of lenses that copies the syntactical sugar and programmatic 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. The variation from standard local lenses is the main subject of study in this manuscript and shall be referred to as the *Remote Lens*.

We now extend the abstract identities in Equation (2.2) to the Remote Lens setting, obtaining the *Remote Lens Identities*,

$$\begin{aligned} RemoteSet\big(acc,\, RemoteView(acc,\, obj),\, obj\big) \;&\equiv\; obj\ , \\ RemoteView\big(acc,\, RemoteSet(acc,\, item,\, obj)\big) \;&\equiv\; item\ , \\ RemoteSet(acc,\, item,\, obj)\ ;\ RemoteSet(acc,\, item2,\, obj) \;&\equiv\; RemoteSet(acc,\, item2,\, obj)\ . \end{aligned} \tag{4.1}$$

When a lens interacts with a structure, a Remote Lens interacts with a Remote Structure. 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 a substructure. And where these laws show the View-Set Eq(2.2), Set-View Eq(2.2), Set-Set Eq(2.2).~~ (Talk about why equivalence "≡" here, but equal "="elsewhere) (Reference Equation 2.2).

The essentials of what is required to construct the *Remote Lens Identities* inside of Haskell is aptly similar to the construction of definitions seen in the elementary Prelude constructions of Haskell lenses. 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 Structure's* nested fields that are to be queried

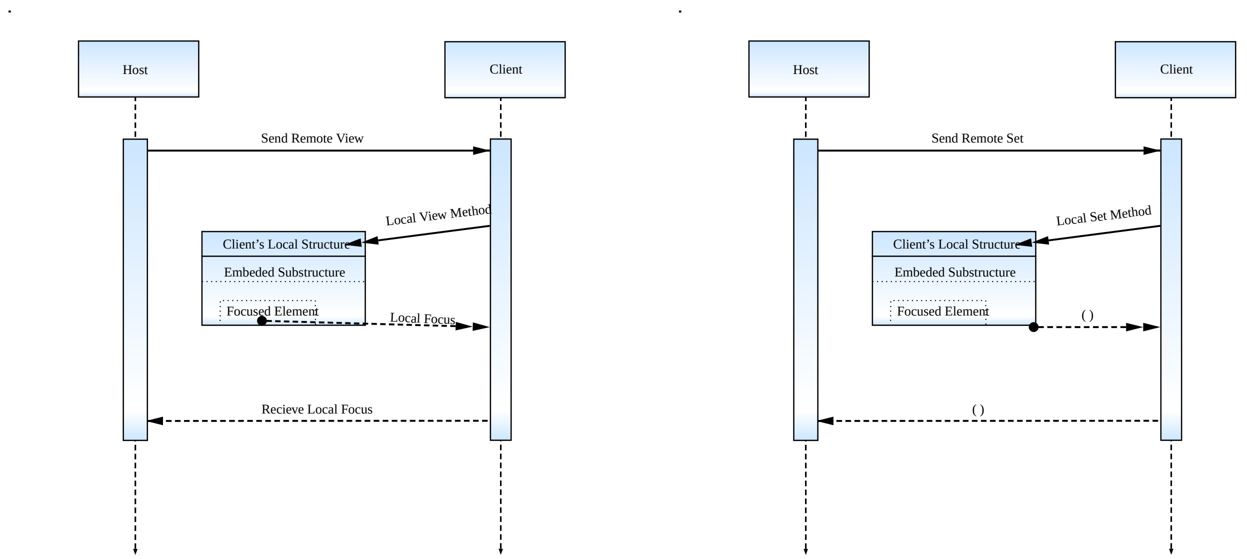

Figure 4.1: Remote Diagrams

into once the command is sent from host to client. The primary difference in how this is handled by Haskell. Rather than simply pattern matching into the already existing structure, a completely different approach is needed instead due to to the structure being remote. To define the individual components of *Remote Lenses* for an arbitrary type system 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)
```

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 is further illustrated in Figure 4.1 where it might be distinguished from the much similar local implementation in Figure 2.2.

This method will involve internally concatenating a conjointment of the parameters such that 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 equivalent 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 is the careful composition method mentioned in the former paragraph is needed to focus in on the remote structure. 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 can be accomplished.

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 one wants 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 one normally is manipulating and creating new objects every time one runs a function or assign it to a value. When manipulating Javascript, one is never creating a new object on the other side. So, any code one would want to write with these methods on the Haskell-side, would need to keep in mind that one is never dealing with a fresh object. The connection will always make sure one is pointing to the one and only original Javascript object one is 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
```

Continuing for the remote-object of person, one is 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
```

There is a particularly complete example from the first exercise of Haskell lensing from FP Complete, that was subsequently adapted to *RemoteMonad* and *Remote Lens* so that it could be clearly illustrated in long form what some of the major similarities and differences might be between remote and local implementations.

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

Some Details.

```
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
```

Some Details.

```
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
```

Some Details.

References

Ahman, D. & Uustalu, T. (2014). Coalgebraic update lenses. *Electronic Notes in Theoretical Computer Science*, 308, 25–48. Proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXX).

Barbosa, D. M., Cretin, J., Foster, N., Greenberg, M., & Pierce, B. C. (2010). Matching lenses: Alignment and view update. *SIGPLAN Not.*, 45(9), 193–204.

Beck, J. M. (1967). *Triples, algebras and cohomology*. Columbia University.

Bohannon, A., Pierce, B. C., & Vaughan, J. A. (2006). Relational lenses: A language for updatable views. In *Proceedings of the Twenty-Fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06 (pp. 338–347). New York, NY, USA: Association for Computing Machinery.

Czarnecki, K., Foster, N., Hu, Z., Lämmel, R., Schürr, A., & Terwilliger, J. (2009). Bidirectional transformations: A cross-discipline perspective. In *Theory and Practice of Model Transformations*, volume 5563 (pp. 260–283).: Springer Berlin Heidelberg.

Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., & Schmitt, A. (2003). A language for bi-directional tree transformations. *Pat*, 333, 4444.

Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., & Schmitt, A. (2007). Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), 17–es.

Fuxman, A., Kolaitis, P. G., Miller, R. J., & Tan, W.-C. (2006). Peer data exchange. *ACM Trans. Database Syst.*, 31(4), 1454–1498.

Gill, A., Sculthorpe, N., Dawson, J., Eskilson, A., Farmer, A., Grebe, M., Rosenbluth, J., Scott, R., & Stanton, J. (2015). The remote monad design pattern. *SIGPLAN Not.*, 50(12), 59–70.

Grothendieck, A. (1958-1960). Technique de descente et théorèmes d'existence en géométrie algébrique. i. généralités. descente par morphismes fidèlement plats. *Séminaire Bourbaki*, 5, 299–327.

Grothendieck, A. (1971). Categories fibrees et descente. In *Revêtements Etales et Groupe Fondamental* (pp. 145–194). Berlin, Heidelberg: Springer Berlin Heidelberg.

Johnson, M. & Rosebrugh, R. (2013). Delta lenses and opfibrations. *Electronic Communications of the EASST*, 57, Volume 57: Bidirectional Transformations 2013.

Johnson, M., Rosebrugh, R., & Wood, R. J. (2011). Lenses, fibrations and universal translations. *Mathematical Structures in Computer Science*, 22(1), 25.

Miltner, A., Fisher, K., Pierce, B. C., Walker, D., & Zdancewic, S. (2017). Synthesizing bijective lenses. *Proc. ACM Program. Lang.*, 2(POPL), 1–30.

Pierce, B. C. & Schmitt, A. (2003). Lenses and view update translation. *Manuscript*.

Appendix A

My Appendix, Next to my Spleen

There could be lots of stuff here