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.

Matthew Moore, Chair

Committee members

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

List of Tables

Chapter 1

Introduction

The origin of this manuscript began 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 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 the popularity of classical lenses. It was discovered that 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 programming language — specifically Haskell, and what are the existing standards for what constitutes the different categories of lenses that exist in the broad community.

Keeping in mind that Lenses are about solving a bijective interaction with a data structure, as so little as to provide convenience to the end user and as far to provide many layers of type safety. 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 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 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 solve the bidirectional transformations problem. 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 modeled after object-class accessors. To 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 one reads 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-behaved lens.

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

Leading to the main body of this work which will be an exploration and attempt to quantitatively define Remote Lenses, and unique differences that one can describe when holding them up to the light in comparison to classical Local Lenses.

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 on it's own, with much stronger abstractions constructed to hold onto many different sub-fields within 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 discovering 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 then 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*.

The structure is so general, polymorphic that we need to establish laws that form what lenses are, and what rules that they should follow. 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, which makes them not well-behaved. In fact, this manuscript will only deal with the idea of well-behaved lenses and their applications. These typical mathematical lens have rules that shall be obeyed between strictly two sources of information, if one is to call them well-behaved.

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

Important note, most definitions have the direction of the lens operators established. i.e. where the Set or View will traverse along a given object to reach the designated destination. So the only arguments are the structure itself in *View*, with just structure and item in *Set*. The definitions implicate 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'd like to mutate a structure. It is perfectly possible that the lens laws themselves are defined with that extra argument as shown below, which exposes some addition structure underneath that the more instantiated definitions previously provided obfuscated.

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

There is a basic idea of what a simplistic lens accomplishes in Figure 2.1, 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 hints that it is possible to navigate into the focus, provided that the focus is a similarly formatted structure. This is true, if both the primary structure and all associated substructures have been made in a way that allows for pattern-matching, then pattern-matching over the entire structure is possible. The stacking of structures, creates a composability of the lenses so that it

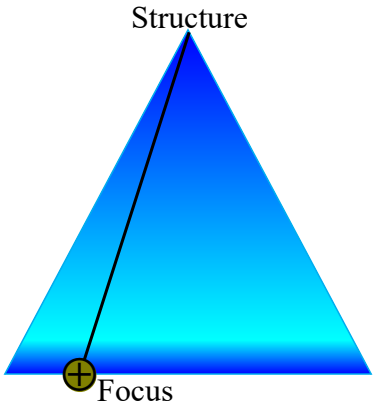

Figure 2.1: Lens Diagram

becomes possible to find a focus even if it is nested two or more structures deep. The embedding a lens friendly structure into a larger one 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 for all cases shown. 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.

To explain further, lenses have been so generalized and grown to such a wide sub-field of *bx*, that is becomes cumbersome to handle explanations for all the work that has been done, Czarnecki et al. (2009). These and many more topics will be restricted to monomorphic occurrences instead, whilst the rest will fall outside the scope of this thesis.

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.

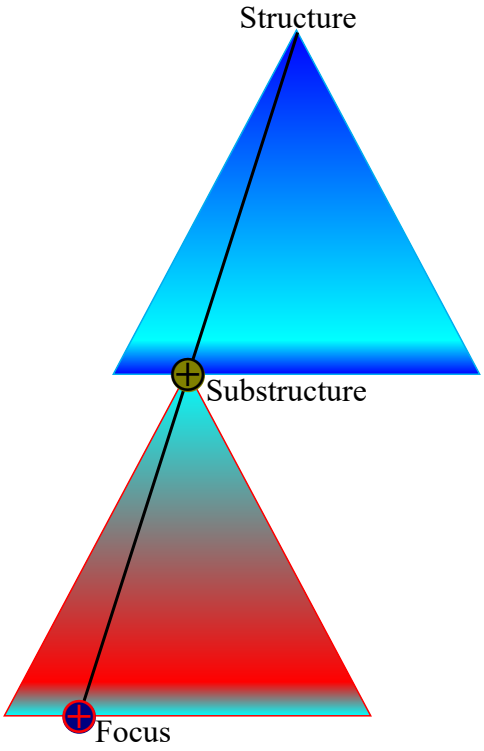

Figure 2.2: Diagram of Lens Composition

```
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)
```

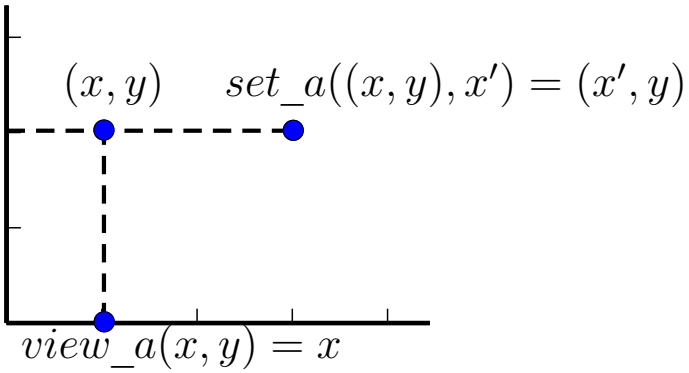

Figure 2.3: Elementary Haskell Lens

```
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)

getPatronTitle :: MyStructure -> String
getPatronTitle = view item_name

setPatronTitle :: MyStructure -> String -> MyStructure
setPatronTitle = 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 PatronTitle 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

Remote Lenses And The Remote Monad

In this chapter, we detail a novel extension of lenses to remote structures, by making use of the Remote Monad. 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. One now modifying that retrieved element with some function, before applying *Set* to that very same structure, shall result in a 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*.

These definitions 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*. This topic be fleshed out starting now and until the end of this document.

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)
```

It is now possible to extend the abstract syntax these modifications to the original well-formed lens rules, provided in Figure 3.2.

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

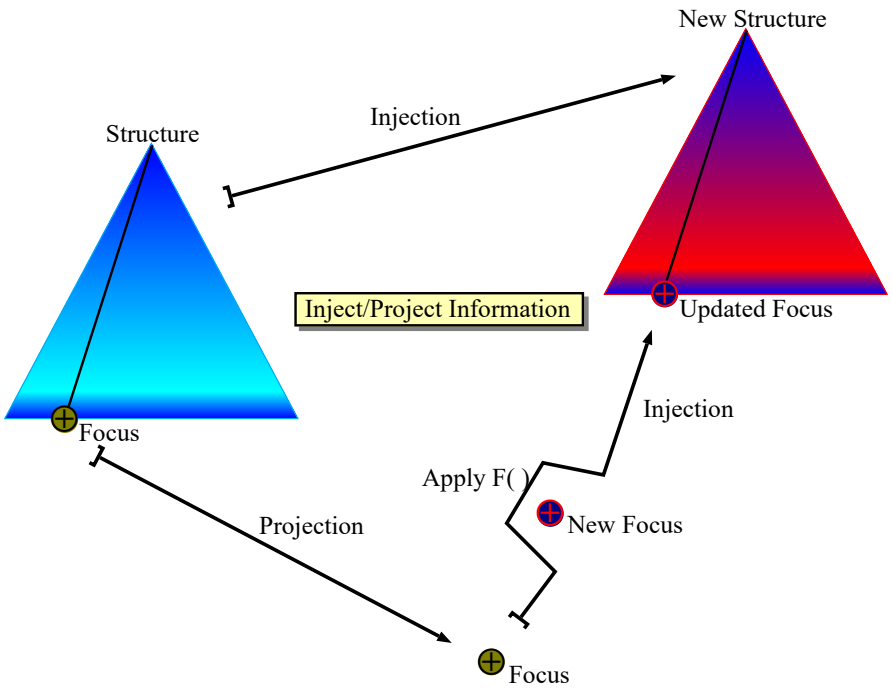

Figure 3.1: Expanded Lens

$$
\begin{aligned}
RemoteSet(acc\ RemoteView(acc\ obj)\ obj) &\equiv obj \\
RemoteView(acc\ RemoteSet(acc\ item\ obj)) &\equiv item \\
RemoteSet(acc\ item\ obj)\ ;\ RemoteSet(acc\ item2\ obj) &\equiv RemoteSet(acc\ item2\ obj)
\end{aligned}
$$

Figure 3.2: Remote Lens Laws
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.
Where these laws show the View-Set Eq(2.2), Set-View Eq(2.2), Set-Set Eq(2.2).

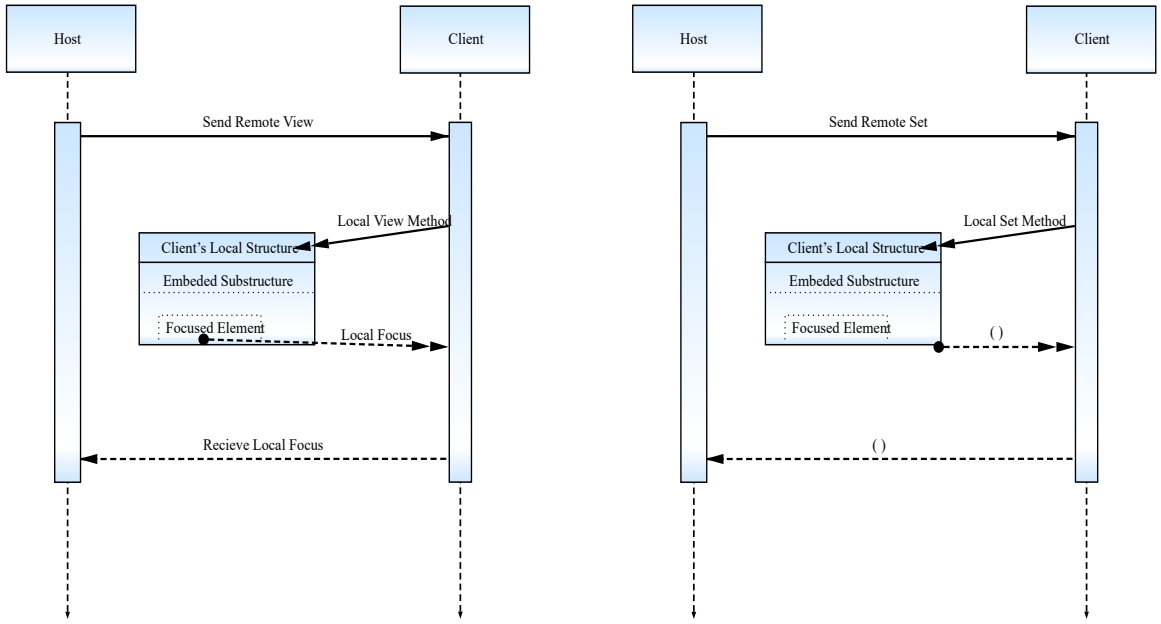

Figure 3.3: Remote Diagrams

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 is further illustrated in Figure 3.3 where it might be distinguished from the much similar local implementation in Figure 2.2.

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 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

Czarnecki, K., Foster, N., Hu, Z., Lämmel, R., Schürr, A., & Terwilliger, J. (2009). Bidirectional transformations: A cross-discipline perspective. volume 5563 (pp. 260–283).

Appendix A

My Appendix, Next to my Spleen

There could be lots of stuff here