# Chapter 1

# Remote Monad

## Abstract

Initial draft to describe and write about the remote monad and the remote lens.

As has been established in the earlier chapter, covering the mathematical background and the historical Haskell programming context, 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*. Which 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.

To start defining the individual components of *Remote Lenses*, we must begin with an abstract type definition which will prove useful in guiding through more concrete implementations that will be brought forward later on:

$$\begin{aligned} &\mathit{RemoteView} :: (\mathit{String} \to \mathit{String}) \to (\mathit{Wrapped}\ a) \to (\mathit{ReturnType}\ b) \\ &\mathit{RemoteSet} \quad :: (\mathit{String} \to \mathit{String}) \to \mathit{String} \to (\mathit{Wrapped}\ a) \to \mathit{ReturnType}\ (\mathit{Wrapped}\ a) \\ &\mathit{RemoteOver} :: (\mathit{String} \to \mathit{String}) \to (t \to a) \to (\mathit{Wrapped}\ a) \to \mathit{ReturnType}\ (\mathit{Wrapped}\ a) \end{aligned}$$

Such that as in the abstract syntax these modifications to the original well-formed lens rules:

$$\begin{gathered} \mathit{RemoteSet}(\mathit{acc}\ \mathit{RemoteView}(\mathit{acc}\ \mathit{obj})\ \mathit{obj}) \equiv \mathit{obj} \\ \mathit{RemoteSet}(\mathit{acc}\ \mathit{item}\ \mathit{obj})\ ;\ \mathit{RemoteSet}(\mathit{acc}\ \mathit{item}2\ \mathit{obj}) \equiv \mathit{RemoteSet}(\mathit{acc}\ \mathit{item}2\ \mathit{obj}) \\ \mathit{RemoteView}(\mathit{acc}\ \mathit{RemoteSet}(\mathit{acc}\ \mathit{item}\ \mathit{obj})) \equiv \mathit{item} \end{gathered}$$

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

Where "*acc*" denotes the accessor to the structure, "*obj*" denotes the structure, "*item*" denotes 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).

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. Along the same vein, the *Remote Lens* variant accomplishes the same 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 will involve internally concatenating a conjointment of the parameter. Where 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 a 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 be accessed upon sending the constructed command. Which is why directly accessing the information is not possible, since one is manipulating only a reference to the actual object in the client's side. And that the careful composition method mentioned in the former paragraph was needed. As 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.

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:

$$
\begin{aligned}
&\mathit{RemoteView} :: (\mathit{FromJSON}\ b) \Rightarrow (\mathit{String} \rightarrow \mathit{String}) \rightarrow (\mathit{RemoteValue}\ a) \rightarrow (\mathit{RemoteMonad}\ b) \\
&\mathit{RemoteSet} \quad :: (\mathit{String} \rightarrow \mathit{String}) \rightarrow \mathit{String} \rightarrow (\mathit{RemoteValue}\ a) \rightarrow \mathit{RemoteMonad}\ (\mathit{RemoteValue}\ a) \\
&\mathit{RemoteOver} :: (\mathit{FromJSON}\ t, \mathit{Show}\ a) \Rightarrow (\mathit{String} \rightarrow \mathit{String}) \rightarrow (t \rightarrow a) \rightarrow (\mathit{RemoteValue}\ a) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \mathit{RemoteMonad}\ (\mathit{RemoteValue}\ a)
\end{aligned}
$$

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 followed implementations:

$$\begin{aligned} &\mathit{RemoteView\ accessor\ object\ =\ }\mathbf{do}\\ &\qquad g \leftarrow \mathit{constructor}\ \$\ \mathit{JavaScript}\ \$\ \mathit{pack}\ \$\ \mathit{accessor}\ (\mathit{var\_text\ object})\\ &\qquad \mathit{procedure}\ \$\ \mathit{var}\ g \end{aligned}$$

$$\begin{aligned} &\mathit{RemoteSet\ accessor\ new\_item\ object\ =\ constructor}\ \$\ \mathit{JavaScript}\ \$\ \mathit{pack}\ \$\\ &\qquad (\mathit{accessor}\ (\mathit{var\_text\ objectName}))\ +\!\!+\ "\ =\ "\ +\!\!+\ \mathit{new\_item} \end{aligned}$$

$$\begin{aligned} &\mathit{RemoteOver\ accessor\ my\_function\ object\ =\ }\mathbf{do}\\ &\qquad \mathit{item} \leftarrow \mathit{RemoteView\ accessor\ object}\\ &\qquad \mathit{let\ new\_item\ =\ show}\ \$\ \mathit{my\_function\ item}\\ &\qquad \mathit{RemoteSet\ accessor\ new\_item\ object} \end{aligned}$$

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.

$$f :: \mathit{Double} \leftarrow \mathit{person} \cdot \mathit{nest} >\!\!>\!\!> \mathit{nest2} >\!\!> \mathit{extra2}$$

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:

$$g \leftarrow constructor\ \$\ JavaScript\ \$\ pack\ \$\ (extra3(nest2(nest)))\ (var\_text\ person)$$
$$procedure\ \$\ var\ g$$

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.

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

```
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
```