Chapter 1

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, **?**. 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.

$$Set\ obj\ (View\ obj) \;=\; id \tag{1.1}$$

$$View\ (Set\ obj\ item) \;=\; obj \tag{1.2}$$

$$Set(Set(obj\ item)\ item' \;=\; Set(obj\ item') \tag{1.3}$$

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.

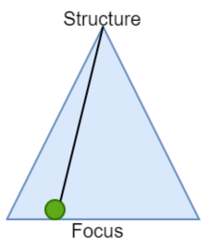

Figure 1.1: Lens Diagram

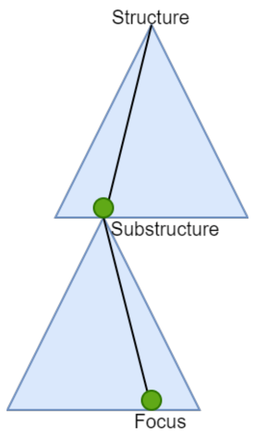

Figure 1.2: Diagram of Lens Composition

$$Set\ acc\ obj\ (View\ acc\ obj) = id \tag{1.4}$$

$$View\ acc\ (Set\ acc\ obj\ item) = obj \tag{1.5}$$

$$Set((acc\ Set(acc\ obj\ item)\ item' = Set((acc\ obj\ item') \tag{1.6}$$

There is a basic idea of what a simplistic lens accomplishes in Figure 1.1, 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 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 1.2.

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 1.3.

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 han-

```
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)
```

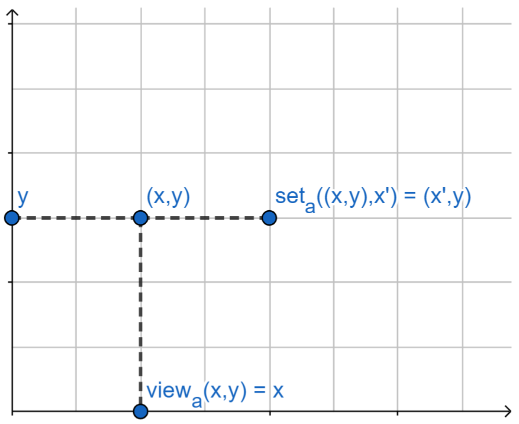

Figure 1.3: Elementary Haskell Lens

dled 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, **?**. 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 1.3, 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)

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 Equations1.1,1.2,1.3. 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.