Parametric lenses: change notification for bidirectional lenses

László Domoszlai[1,2] Bas Lijnse[1] Rinus Plasmeijer[1]

[1]Radboud University Nijmegen, Netherlands, ICIS, MBSD
[2]Eötvös Loránd University, Budapest, Hungary, Software Technology Department
l.domoszlai@cs.ru.nl, b.lijnse@cs.ru.nl, rinus@cs.ru.nl

Abstract

Most complex applications inevitably need to maintain dependencies between subsystems based on some shared data. The dependent parts must be informed that the shared information is changed. As every actual notification has some communication cost, and every triggered task has associated computation cost, it is crucial for the overall performance of the application to reduce the number of notifications as much as possible. To achieve this, one must be able to define, with arbitrary precision, which party is depending on which data. In this paper we offer a general solution to this general problem. The solution is based on an extension to bidirectional lenses, called *parametric lenses*. With the help of parametric lenses one can define compositional *parametric views* in a declarative way to access some shared data. Parametric views, besides providing read/write access to the shared data, also enable to *observe* changes of some parts, given by an explicit parameter, the *focus domain*. The focus domain can be specified as a *type-based query language* defined over one or more resources using predefined combinators of parametric views.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Specialized application languages

Keywords Bi-directional programming, lenses, parametric lenses, parametric views, notification systems

1. Introduction

Complex applications commonly have to deal with shared data. It is often confined to the use of a relational database coupled with a simple concurrency control method, e.g., optimistic concurrency control [21]. In other cases, when a more proactive behavior is required, polling or some ad hoc notification mechanism can be invoked. At the farther end of the range there are some very involved applications (multi-user applications, workflow management systems, etc.), which are based on interdependent tasks connected by shared data. In the most general case, one has to deal with complex task dependencies defined by shared data coming from diverse sources, e.g. different databases, shared memory, shared files, sensors, etc.

IFL '14, October 1–3, 2014, Boston, MA, US.

http://dx.doi.org/10.1145/2746325.2746333

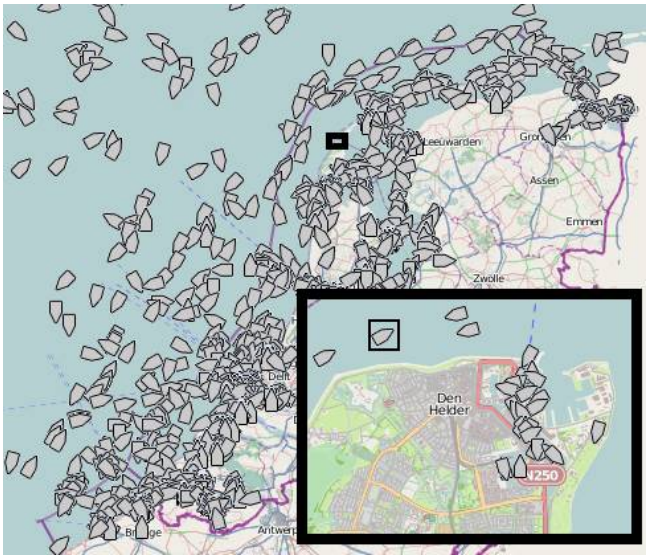

Figure 1. Ships around the Dutch coast

As an example, consider the following case which is based on a prototype we have developed for the Dutch Coastguard [22]; it is used throughout the paper to introduce the problem, and the concepts of the proposed solution. We have a small database which acts as a source of data of ships: name, cargo capacity, last known position, etc. The positions of the ships are updated repeatedly as the ships move; ships have a transponder on board which sends their latest position on a regular basis. As a basic task, we simply want to show the positions of the ships on a map, of which users are allowed to select a region to view, the *focus* of their interest. In this setting we can think of map instances and update processes as interdependent tasks that are connected by the data of ships they share. When the position of a ship is updated in the database, the map instances, of which focus covers the old or the new coordinates, must be refreshed.

From a theoretical perspective, it would be correct behavior to notify every map instance on every ship movement. However, this leads to huge efficiency issues in practice. There are many thousands of ships in the North Sea constantly moving around. Only those map instances need to be refreshed in which region the position of a ship is changed. As every actual notification has some communication cost, and every triggered task has associated computation cost, it is crucial for the overall performance of the application to reduce the number of notifications as much as possible.

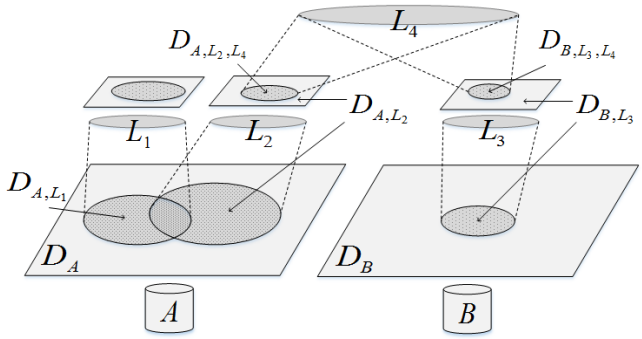

Figure 2. The notification problem

Thus, we need a notification system which, for efficiency reasons, can be as accurate as needed for optimal efficiency.

As the problem described above is a very common computational pattern, we would like to offer a general, reusable solution.

From the computational perspective, focusing on a specific domain of the underlying data can be achieved by creating and working with one of its abstract views. Lenses [7, 8, 12–14, 17, 18, 35] are commonly used for creating abstract views. They can be used to support partial reading and writing, for access restriction or to provide a specific view of the data. Lenses enable to define bidirectional transformations. In a nutshell, a lens describes two functions to map the input to an output and backwards.

In our example two kind of abstract views are needed for serving different processes: one to show the ships located in a given region of the map, and another one for the update process, which periodically updates the coordinates of a ship in the database.

The general notification problem is depicted in Figure 2. Given is a set of shared data sources of any type (*A* and *B* in the picture) holding a set of data (D_A, D_B). There are also given some lenses defined on top of the data sources and on each other. These are L_1, L_2, L_3 and L_4 in the picture. The additional subscripts of the original data sets, D_A and D_B, denotes the sets of data we gain after applying a series of lenses to the original data sets (e.g. D_{A,L_2} denotes the set of data that can be seen from A through L_2). One typical question can be, e.g., whether a given update through L_4 affects the D_{A,L_1} or not? What about the other way around?

Unfortunately, classical lens theory does not provide any tool to discover whether a given update through some lens affects the data that can be seen through another lens. In this paper we present a general extension to lenses as a solution for this general problem. In this extension, called *parametric lenses*, lenses are partially defunctionalized to extract a first-order parameter, *the focus domain*, that groups a set of similar lenses into a single parametric lens in which the parameter essentially encodes which part of the input domain is mapped to the output domain by the lens. This additional focus information will enable to read, update, and observe specific parts of the underlying data.

Parametric lenses are *pure*, thus cannot be applied to some shared data directly, they must be lifted into an impure context. Therefore, they are attached to the shared data through a *non-pure* abstract interface called *parametric view*. The parametric views are allowed to be composed using predefined combinators. Using these combinators, one is able to specify the focus domain as a *type-based query language* defined over one or more resources. With the query language, one can focus on a specific part of the underlying shared data during reading, writing, or it can be used for notification purposes.

We use two examples throughout the paper to present our solution. The first example is based on the simplest form of parametric views and it is compact enough to give a nice insights in the main idea; it shows how to find a node, by some property, in an arbitrary tree structure. The selected node can be used then not only for reading or updating, but also for observing its changes.

The second example, our motivating one, is slightly more complex, and requires the introduction of additional combinators. For its development we parametrize some relational lenses developed in [8] for solving the so called *view-update problem*.

We offer the following contributions in the paper:

1. We introduce parametric lenses as a general extension to bidirectional lenses. Parametric lenses enable the development of efficient notification systems based on them. Parametric lenses are embedded into compositional parametric views which are defined over shared data;
2. We implement the executable semantics, using Haskell [30], of the combinators and an underlying notification engine. The complete Haskell implementation, along with the examples developed in the paper, can be found at `https://wiki.clean.cs.ru.nl/File:PViewIFL.zip`;
3. We develop two examples in the paper to demonstrate the usage of parametric lenses. An introductory example based on a simple recursive data structure, and a simplified real world example based on the iTasks coastguard prototype described above.

The remainder of this paper is structured as follows: in Section 2, after a brief overview of classical lenses, the parametric extension is introduced in Section 3. In Section 4, we introduce parametric views. In Section 5, the realization of the parametric and the classical, non-parametric variants of lenses, in the setting of parametric views, are contrasted. The first, introductory example is developed in Section 6. Then, before we proceed with more advanced cases, a new combinator is introduced in Section 7 to be able to join views of different data sources together. Using this combinator, our second, motivating example is developed in Section 8. In Section 9, an alternative implementation is provided of the second example to increase the accuracy of the notifications. It is followed by a discussion of related work in Section 10 and concluding remarks in Section 11.

The executable semantics and the examples are written in Haskell [30], and they are also dependent on some extensions of the Glasgow Haskell Compiler (GHC) [26] and its libraries. The given implementation and example code uses the following language extensions and libraries: generalized algebraic datatypes (GADT) [1], the Data.Typeable package [3], monads [34] (and in particular the *State* [4, 34] and *Writer* [5, 19] ones), monad transformers [2] and applicative functors [27]. Their basic knowledge is necessary for the comprehension of the paper.

2. Introduction to Lenses

The starting point for this work is the class of bidirectional transformations known as lenses. Thus, in this section a brief overview of lenses is given to explain what they are, and how they work.

Lenses enable the definition of bidirectional transformations. In a nutshell, a lens describes two functions to map the input (or source: X) to an output (or view: Y) and backwards. The *get* function maps the input to some output, while the *put* function maps the modified output, together with the original input, to a modified input:

$$get \in X \to Y$$
$$put \in Y \times X \to X$$

Lenses are expected to obey the following "round-tripping" laws for every $x \in X$ and $y \in Y$:

$$put\ (get\ x)\ x\ =\ x \qquad \text{(GETPUT)}$$
$$get\ (put\ y\ x)\ =\ y \qquad \text{(PUTGET)}$$

These laws express fundamental expectations about how the components of a lens should work together. The GETPUT law (also known as *consistency* [35]) ensures that all updates on a view are captured by the updated source, while the PUTGET law (also known as *acceptability*) prohibits changes to the source if no update has been made on the view. Lenses obeying these laws are called *well-behaved* [12].

Sometimes a third law, called PUTPUT, is also considered. For every $x \in X$ and $y, y' \in Y$:

$$put\ y\ (put\ y'\ x)\ =\ put\ y\ x \qquad \text{(PUTPUT)}$$

This law states that the effect of a sequence of two *puts* is just the effect of the second. Well-behaved lenses which also satisfy the PUTPUT law, are called *very well-behaved*.

In the next section we go beyond the classical theory and parametrize lenses. With a parametrized lens, we can focus on a specific part of the underlying data for reading, writing and observing.

3. Introduction to Parametric Lenses

In the parametric lens extension classical lenses are partially defunctionalized to extract a first-order parameter (the *focus domain*: Φ, Ψ) that groups a set of similar lenses into a single parametric lens in which the parameter essentially encodes which part of the input domain is mapped to the output domain by the lens. Parametric lenses additionally return a predicate in the *put* direction. This predicate, called the *invalidation function*, encodes the semantic information associated with the focus domain.

$$get_F \in \Phi \times X \to Y$$
$$put_F \in \Phi \times Y \times X \to X \times (\Phi \to Bool)$$

The invalidation function tells whether the particular update with some focus affects a given other focus from the same domain or not. To illustrate the role of this function, consider the following sequence of operations ($\phi, \psi \in \Phi$, $x, x' \in X$, $y, y', z \in Y$ and $inv \in \Phi \to Bool$):

$$\begin{aligned} y &= get_F\ \phi\ x \\ (x',\ inval) &= put_F\ \psi\ z\ x \\ y' &= get_F\ \phi\ x' \end{aligned}$$

We say that the invalidation function $inval$ is *consistent* if $y \neq y' \Rightarrow inval\ \phi = True$. If $y \neq y' \Leftrightarrow inval\ \phi = True$, we say that $inval$ is *accurate*. Consistency is a fundamental property, all invalidation functions must satisfy it. It expresses the fundamental requirement that all the actual changes can be observed, but also allows false notifications. Accuracy however is not obligatory and, especially when different focuses can overlap, may not be implemented effectively in practice. Nevertheless, during the development of parametric lenses, one should aim for accuracy to effectively reduce the number of triggered notifications (as an example, the constant True function satisfies consistency).

With the extended functions, the round-tripping laws take the following form for every $\phi \in \Phi$, $x \in X$ and $y \in Y$:

$$fst\ (put_F\ \phi\ (get_F\ \phi\ x)\ x)\ =\ x \qquad \text{(GETPUT)}$$
$$get_F\ \phi\ (fst\ (put_F\ \phi\ y\ x))\ =\ y \qquad \text{(PUTGET)}$$

```
data PView φ m a where
  Source  :: (Monad m, Typeable φ)
          ⇒ Source φ m a
          → PView φ m a
  Project :: (Monad m, Typeable φ)
          ⇒ PView φ m a → Lens a b
          → PView φ m b
  Focus   :: (Monad m, Typeable φ, Typeable ψ)
          ⇒ PView φ m a → ParametricLens ψ a b
          → PView (φ, ψ) m b
  Product :: (Monad m, Typeable φ, Typeable ψ)
          ⇒ PView φ m a → PView ψ m b
          → PView (φ, ψ) m (a, b)
```

Figure 3. The *PView* type

The PUTPUT law is the following for every $\phi \in \Phi$, $x \in X$ and $y, y' \in Y$:

$$put_F\ \phi\ y\ (fst\ (put_F\ \phi\ y'\ x))\ =\ put_F\ \phi\ y\ x \qquad \text{(PUTPUT)}$$

The elements of a given focus domain, e.g. $\phi \in \Phi$, are called the *focuses*. The semantics of the focuses are encoded by the invalidation function, and there is no general restriction on them; they can denote an arbitrary part of the underlying data, and, for instance, they can also overlap in an arbitrary way.

In our example, the focus domain can be the set of possible contiguous regions of the map. The elements of this domain, the focuses, are just individual regions of the map, which can overlap. It is the task of the invalidation function to predicate whether two values of the focus domain, two regions, actually overlap or not.

In the next section we introduce parametric views to implement this idea. In this implementation focus domains take the form of *types*, and the actual focuses of the domain are values of this type.

4. Introduction to Parametric Views

A parametric view is a compositional abstract interface to provide access to some *shared* data in a general way. The interface enables the underlying data to be read, written and observed according to a domain of focus. In this section we present the basic structure of parametric views and extend it later in the paper.

A parametric view is represented by the ($PView\ \ \phi\ \ m\ \ a$) generalized algebraic data type (GADT), see Figure 3. The type parameter ϕ is the focus domain, the type of the focus parameter, which must be $Typeable$ (this requirement is explained later on in this section). The type m is any monad which provides access to the underlying shared data. Finally, a is the type of the values that can be read from and written into the view.

The $PView$ data constructors play the role of combinators. With the combinators, one can create a view based on some shared data (the $Source$ combinator, introduced later on in this section), change the focus domain or the associated read/write type of the view ($Focus$ and $Project$ combinators, see Section 5), or one can combine views to synthesize a compound one ($Product$ combinator, see Section 7).

The interface of a parametric view consists of three functions: the *read* and *update* functions for reading and writing the view, and the *observe* function to ask for change notifications:

```
type ObserveId = String
type ObserveHnd m = m ()
read    :: (Monad m, Typeable φ)
 ⇒ PView φ m a → φ → PViewT m a
update  :: (Monad m, Typeable φ)
 ⇒ PView φ m a → φ → a → PViewT m ()
observe :: (Monad m, Typeable φ)
 ⇒ PView φ m a → φ → ObserveId → ObserveHnd m
 → PViewT m ()
```

The first two arguments of these functions are the parametric view of type ($PView\ \phi\ m\ a$) and the actual focus value of type ϕ. The *observe* function further takes a unique identifier of the subscription ($ObserveId$, a string value for the sake of simplicity), and an event handler function (an action in the monad m). The $ObserveId$ argument is used e.g. to remove the subscription later.

The interface functions are monadic, they are based on the $PViewT$ monad transformer, which basically introduces a state. The definition of this $PViewT$ type, along with the explanation of its purpose is given later on in this section.

Type m is constrained to be a $Monad$, but also used as an applicative functor to provide a more applicative style implementation. It is assumed that $Applicative \Rightarrow Monad$.

The $Source$ type describes how to interface with the actual data source in the associated monad. A parametric view is created from a $Source$ description by the $Source$ data constructor of the $PView$ GADT (see Figure 3). A *Source* is already parametric, thus the provided access functions require a focus value:

```
data Source φ m a where
  MkSource :: (Monad m, Typeable φ)
           ⇒ (φ → m a)                      -- sread
           → (φ → a → m (Invalidate φ))     -- supdate
           → Source φ m a
```

The *sread* function takes a focus and reads data according to that specific focus. The *supdate* function updates the part of the underlying data which is denoted by the focus value in the first argument. The new data is given in the second argument. After the update, it returns the *invalidation function*, discussed in the previous section.

```
type Invalidate φ = φ → Bool
```

Reading a $Source$ is straightforward, as the request is just forwarded to the underlying data source (the other alternatives of the $read$ function are given later when the related data constructors of the $PView$ GADT are introduced in detail):

```
read (Source (MkSource sread _)) p = lift (sread p)
```

The $update$ and $observe$ interface functions are more elaborate as those are involved in the notification process. The idea is to save the notification requests in a state monad (the associated functions of the state monad are qualified with ST in the paper), then lookup and trigger the event handlers of the matching ones during updates.

The state of the notification engine is introduced by the $PViewT$ monad transformer, and it keeps the list of notification requests. The $observe$ interface function maintains the list of requests in the state monad, which is used by the $update$ function to trigger notifications.

```
type PViewT m = StateT [NRequest m] m
  -- a notification request consists of an id, an event handler,
  -- and the observed focus which is encoded in a Dynamic
data NRequest m
  = NRequest ObserveId (ObserveHnd m) Dynamic
```

To save a request we need a *comparable* reference to the observed view; as the view contains functions only, saving itself would be pointless. Obviously, a machine address based identification is not feasible in a pure functional language, and generating identifiers is not viable either since we want to define the views in a pure, declarative way; for generating unique identifiers, a state would be required. We could require the developer to provide an identifier along with the views. However, we would like two views, defined at two different parts of the application using the same combinators with same arguments, to be equal. That would be error prone with explicit view ids.

However, the type of the focus domain is a perfect candidate for identification as it assigns a unique semantic information to the view. It is realized using the standard GHC extension, $Data.Typeable$. This extension enables us to associate type representations to (monomorphic) types, then, e.g., compare these representations. This is the reason why the types of the focus domains have to be $Typeable$. Note, however, that using the focus domains for identification, imposes a uniqueness requirement on them: focus domains must be *unique* between the semantically different parametric views.

In accordance, the reference is encoded as a value of type $Dynamic$; the associated type is the actual reference to the view, while the associated value is the focus value to be observed.

With the exception of the $Source$ data constructor, a parametric view is recursively defined by the help of other parametric views, creating in this way a directed acyclic graph. As this graph is created in a pure way, it does not have back edges. When a given node is updated, its parents cannot be informed to evaluate the notification requests which are in their focus domain.

```
observe :: (Monad m, Typeable φ)
   ⇒ PView φ m a → φ → ObserveId → ObserveHnd m
   → PViewT m ()
observe v p oid ohnd = do
  rs ← genreqs v p   -- generate implicit notification requests
  ST.modify (map (NRequest oid ohnd) rs++)
```

To overcome this limitation, the $observe$ function puts *implicit notification requests* on all of the views of the subgraph denoted by the target view. During an update, the most adequate of these, the one which has the least distance to the updated view, is triggered. These implicit notification requests are generated recursively by the $genreqs$ function.

The $genreqs$ function, when applied to the $Source$ data constructor, straightforwardly creates a one-element list storing the focus value provided for the $Source$:

```
genreqs (Source _) p = return [toDyn p]
```

Similarly to the $observe$ function, the $update$ interface function creates *notification events* of type $NEvent\ m$ for all the views of the subgraph denoted by the updated parametric view (by $update'$); then, in a subsequent step, it matches up these events with the stored notification requests (by $trigger$, defined further on). Please notice that for the sake of providing concise code, the $Writer$ monad transformer is exploited in $update$ and $update'$; its associated functions are qualified with W in the paper.

```
update v p a
 = W.execWriterT (update' v p a) >>= trigger ∘ reverse
```

The $update'$ function traverses the graph of views in a recursive manner during an update. It returns the invalidation function of the given view, which is used to create a computed one in the recursion step. It also generates a notification event for every view on the paths to the roots and stores them in a $Writer$ monad. The actual update happens at the $Source$ views in the roots, but all the invalidation functions computed for the non-leaf views have further information on the exact range of affected data.

```
  -- a notification event consists of a reference to the source
  -- view and the associated invalidation function
data NEvent m
 = ∃φ. Typeable φ ⇒ NEvent TypeRep (Invalidate m φ)
update' :: (Monad m, Typeable φ)
 ⇒ PView φ m a → φ → a
 → WriterT [NEvent m] (PViewT m) (Invalidate m φ)
```

The $update'$ function, when applied to a $Source$, just forwards the request to the underlying data source:

```
update' (Source (MkSource _ supdate)) p a
 = (lift ∘ lift) (supdate p a) >>= λinval → returnE p inval
returnE p inval
 = W.tell [NEvent (typeOf p) inval] >> return inval
```

Finally, in the $trigger$ function, the notification events generated by $update'$ are matched with the stored notification requests. It matches every event with every request, but for a given observation id, it takes the most adequate (which happens to be the first match), and the remaining are skipped:

```
trigger :: (Monad m) ⇒ [NEvent m] → PViewT m ()
trigger es = do
  rs ← ST.get   -- read notification request
  foldM_ (λskips event
          → foldM (match event) skips rs) [] es
  -- match a request to with an event
  -- event handlers are executed only once (using the skips list)
match (NEvent vid inval) skips (NRequest oid ohnd dyn)
  | notElem oid skips ∧ dynTypeRep dyn ≡ vid
    = case fromDynamic dyn of
      Just p   → when (inval p)
                      (lift ohnd) >> return (oid : skips)
      Nothing → return skips
match _ skips _ = return skips
```

The basic structure of the notification engine is now sketched out. Based on this foundation, the semantics of the remaining combinators are introduced in the consequent sections step by step as they are needed for the development of the examples.

5. Lens combinators

In this section parametric lenses are introduced in the context of parametric views in parallel with the introduction of classical lenses. In this way the two concepts, parametric and non-parametric lenses, can be easily contrasted. With parametric lenses, we are able to develop our first example in the next section.

There are many ways to represent lenses in a functional language like Haskell. For example, the lenses in the popular Control.Lens GHC package [20] are Laarhoven style lenses [33] implemented as functional references based on *applicative functors*. We could use the same representation for classical lenses, however it is not directly clear how the parametric variant would work with this technique. This is the main reason that we decided for a straightforward, record based representation. Another reason is that in our experience, the classical representation as a pair of functions is easier to comprehend.

Classical lenses

A classical lens is represented by the $Lens$ record type and applied to a view by the $Project$ combinator of the $PView$ GADT. When a lens of type ($Lens\ a\ b$) is applied to a parametric view of type ($PView\ \phi\ m\ a$), the resulting view (type of $PView\ \phi\ m\ b$) has the same focus domain inherited from the underlying view. This is expected as the classical theory has nothing to do with focus domains. The types of get and put components of the $Lens$ record type also straightforwardly reflect the classical theory:

```
data Lens a b = MkLens {
  get :: a → b,
  put :: a → b → a
}
```

When a $Project$ node of a parametric view is read, the value read from the underlying view is mapped by the get function of the lens. Similarly, in the write direction, the underlying view is read first, then the put function is used to incorporate the write value into the underlying data, which is updated in a final recursive step.

```
read (Project v l) p = get l <$> read v p
update' (Project v l) p a
   = lift (read v p) >>= λs → update' v p (put l s a)
```

The $Project$ nodes are simply ignored by the $genreqs$ function as they do not contain any additional focus information:

```
genreqs (Project v _) p = genreqs v
```

Parametric lenses

The parametric variant of the classical lens is represented by the $ParametricLens$ record type and applied to a view by the $Focus$ combinator. The combination of a view of type ($PView\ \phi\ m\ a$) and a parametric lens of type ($ParametricLens\ \psi\ a\ b$) results in a parametric view of type ($PView\ (\phi, \psi)\ m\ b$). That is, using the focus domain of the parametric lens, the focus domain of the initial view is refined.

The components of the $ParametricLens$ record type, the get_F and put_F functions, in accordance with the formal introduction, extend the classical functions with focus information and with the invalidation function:

```
data ParametricLens ψ a b = MkPLens {
  get_F :: ψ → a → b,
  put_F :: ψ → a → b → (a, Invalidate ψ)
}
```

When a $Focus$ node of a parametric view is read, first the underlying view is read recursively using the corresponding element of the focus value, then get_F is applied to the read value along with the second element of the tuple of the focus value (this is the only difference compared to the classical case):

```
read (Focus v l) (p, q) = get_F l q <$> read v p
```

The update logic is more different, compared to the classical case, when a $Focus$ node of a parametric view is updated. First the

original input value is read recursively from the underlying view using the corresponding element of the focus value. After that, put_F, the put part of the parametric lens, is applied to this original input value and the provided new output value along with the other part of the focus value to compute a new input value; it also returns a partial invalidation function, covering the ψ type. Then, the underlying view is updated recursively using the newly computed value; the update process also returns another partial invalidation function, covering the ϕ type. Finally, the two partial invalidation functions are combined together to cover the whole focus domain: it states that the validation can be safely decided solely by the invalidation function coming with the parametric lens, if the focus related to the underlying view is the same in the update and in the notification request.

```
update' (Focus v l) (p, q) a = do
  s ← lift $ read v p
  let (s', invall) = put_F l q s a
  invalw ← update' v p s'
  returnE (p, q) (comp invall invalw)
  where
    comp invall invalw (p', q')
      | p ≡ p' = invall $ q'
      | otherwise = invalw p'
```

In *genreqs*, an implicit notification request for the *Focus* view is generated prior to the recursion step:

```
genreqs (Focus v _) p = (toDyn p:) <$> genreqs v (fst p)
```

Above we gave the formal semantics of the most important combinators. In the following sections we give a more intuitive insight into their behavior through a series of illustrative examples.

6. First example: Self service storage

Let us consider that we own a company which provides self-service storage for its customers. The customers hire storage space, or just let us store some objects for them. The storage space is structured in a hierarchical manner: there are multiple buildings which contain rooms, the rooms have shelves or lockers which may contain boxes with some objects or the objects directly. The actual hierarchy is very flexible, and varies between buildings and rooms.

We would like to develop a piece of software to maintain the locations and properties of the objects we handle. The software will be based on the rose tree data structure as it is simple, but still flexible enough to describe any necessary hierarchies. The element type of the tree describes a container which has a type, properties and also can contain object items. Items have properties like name, quantity, etc.

```
  -- The store is a hierarchy of containers
data RoseTree a = RoseTree a [RoseTree a]
type Store      = RoseTree Container
  -- Containers are a collection of properties and object items
data Container  = Container CType [Property] [Item]
data CType      = Building String | Room String | ...
data Property   = Owner String | ...
data Item       = Item {name :: String, ...
```

There are two important requirements for the system: (1) we want to be able to find a container by an arbitrary property, and (2) we also want to be able to observe the changes of an arbitrary container or subtree. An example for the latter is that we may want to be informed when a specific object is removed from a container. In the following, we show how to implement these requirements based on parametric views.

```
type StoreT   -- monad transformer for accessing storage data
type StoreView q = PView q StoreT Store
  -- unique focus domain for storage data
data StoreSource = StoreSource deriving Typeable
store :: StoreView StoreSource
store = Source (MkSource sread supdate) where ...
```

First a base view must be implemented to provide access to the underlying data in a monad. The view has a simple, flat focus domain, *StoreSource*, to identify the source of the data which is available in the monad $StoreT$. The base view, the *store* function is created using the *Source* combinator (the definition of this function is simple, but depends on the actual monad, which is not important for the main idea, and thus not included here).

The implementation is based on a single parametric lens which enables to go one level down in the hierarchy by focusing on one of the children of a tree. The lens has the *Selector* focus domain with one data constructor to provide the index of the child:

```
data Selector = S {unS :: Int} deriving Typeable
selectLens :: ParametricLens Selector Store Store
selectLens = MkPLens {get_F = get, put_F = put} where
  get (S i) (RoseTree _ cs)   = cs !! i
  put (S i) (RoseTree r cs) c =
    (RoseTree r (take i cs ++ [c] ++ drop (i + 1) cs),
    λ(S j) → j ≡ i)   -- invalidation function
```

In the *get* direction, one of the children is selected straightforwardly based on the current focus value. The *put* direction is also straightforward, the indicated child is replaced with the given subtree; still, the invalidation function requires some consideration.

As we want to go deeper and deeper in the hierarchy, *selectLens* must be applied again and again, thus creating a wider focus domain. For example, applying the lens twice to the base view, we get the following:

```
store `Focus` selectLens `Focus` selectLens
  :: StoreView ((StoreSource, Selector), Selector)
```

This view enables us to select any child from the second level by providing a specific focus value e.g. $((StoreSource, S\ 1), S\ 0)$ selects the first child of the second child of the root element. This example also gives us the insight that two different values from the same focus domain always select different parts of the underlying data, in other words there is no overlapping. In this case, deciding whether a given update affects a focus or not, simply reduces to comparing the focus values. This is encoded in the invalidation function returned by *put*. In the ship example developed later on, this property does not hold, which makes the invalidation functions non-trivial.

Finally, the function which finds a container based on a predicate function is as follows:

```
findFirst :: Typeable q ⇒ StoreView q → q
                        → (Container → Bool)
                        → PViewT StoreT (Result)
findFirst v q pred = do
  (RoseTree container children) ← read v q
  if pred container
    then return $ Just $ StoreResult v q
    else case children of
      [] → return Nothing
      _ → searchChildren (length children)
```

```
  where
    searchChildren 0 = return Nothing
    searchChildren i = do
      res ← findFirst focusView (q, S (i − 1)) pred
      case res of
        Nothing  → searchChildren (i − 1)
        _        → return res
    focusView = v `Focus` selectLens
data Result where
  Result :: Typeable q ⇒ StoreView q → q → Result
```

First, the container of the root element is checked, then its children one by one if the container did not satisfy the predicate. For checking the children, the *selectLens* is applied to the view to be able to focus on one more level deeper, then the *findFirst* function is called recursively.

The only part to be considered is the return type. To be able to point out a specific part of the shared data, we need to return a view which has the proper focus domain, and a focus value of the same type. The main problem here is that the type of the focus domain is not known in advance as it depends on the actual structure of the data, and on the location of the sought container inside it.

Thus, the return value of the function cannot be typed directly. However, it is not even necessary: the API functions depend on the *Typeable* property of the focus domain only. This idea is encoded in the *Result* type. After the values from the *Result* data constructor are unwrapped, they can be used to read or update the returned view, use some combinators on it, or to observe the view.

7. Joining multiple sources

Before we continue with our main example, one more combinator must be introduced. A very common situation is where the data we are working on, is coming from multiple sources. In the aforementioned example of ships, there are two sources of data, one for the actual ship data and one for the current positions of the ships. It is also common that the data coming from the different sources is dependent; e.g. the position data provides additional properties of the ships. In this case, one may want to create a joint view based on the individual views e.g. a relational join of the ships and positions.

This can be achieved using the *Product* combinator. This combinator takes two views and simply tuples their focus domains and view types together. It only requires that the views share the same monad type.

When a *Product* view is read, the two underlying views are read one by one, then the results are simply put together in a tuple:

```
read (Product vl vr) (p, q)
  = (,) <$> read vl p <*> read vr q
```

During updating, first the underlying views are updated, then a new invalidation function is created which fires when at least one of the invalidation functions, resulting from the recursive updates, fires:

```
update′ (Product vl vr) (p, q) (a, b) = do
  invall ← update′ vl p a
  invalr ← update′ vr q b
  return $ λ(p, q) → invall p ∨ invalr q
```

In *genreqs*, no notification event is generated for the combined view, only for the underlying ones. It is not necessary as the underlying views generate their own events which cover the whole focus domain of the combined one.

```
genreqs (Product vl vr) (p, q)
  = (++) <$> genreqs vl p <*> genreqs vr q
```

In the following section, our main example is developed, which gives an intuition on how to use this combinator in practice.

8. Second example: Filtering ships

At this point we have all the tools to pick up the main exercise where we left it in the introduction. As a recap, we have two databases holding ship data and location data. We develop the following views (to develop our motivating example, we could use the second one for updating the positions of ships, and the last one to acquire the data of ships located in a specific region):

- Which ships have a given cargo capacity?
- Where is a given ship?
- Which ships are in a given region of the world?

The purpose of these views is not only to read and update the shared data, but we also would like to *observe* the data behind them (to be *notified* when the data of a view has changed).

The structure of the implementation is the following. All the views are built on two base views which access the data sources. First, filtering parametric lenses are applied to the base views. In the next step, a joint view is created from the filterable views using the *Product* combinator introduced in the previous section. Finally, the joint view is turned into a view of natural join of the ship and position records using a classical, non-parametric lens.

We work with lists of records, which makes the problem very similar to the classical *view update problem* well-known in the database literature [6, 16]. The view update problem arises from the fact that when the view update is translated to a database update, there exist more than one database update that may correspond to the same view update.

It is out of the scope of this paper to deal with this problem, but fortunately, the view update problem is already investigated in the context of bidirectional lenses [8, 12]. Thus, the lenses in this section are based on the relational lenses developed in [8].

The ship and position data is defined by the following record types:

```
data Ship = Ship
  { s_name     :: String
  , s_capacity :: Int
  }
data Position = Position
  { p_ship_name :: String
  , p_position  :: Coord
  }
type Coord = (Double, Double)
```

The base views, the *ships* and *positions* functions, use these types and flat focus domains as in the previous example. The implementation of these functions is not included, because it is irrelevant from the perspective of parametric lenses:

```
ships     :: PView Ships ShipsT [Ship]
positions :: PView Positions ShipsT [Position]

  -- monad transformer for accessing ship data
type ShipsT

  -- focus domains
data Ships     = Ships     deriving Typeable
data Positions = Positions deriving Typeable
```

To create filterable views, we use the following generic parametric lens:

```
selectLens :: (f → a → Bool) → ParametricLens f [a] [a]
selectLens pred = MkPLens {get_F = get, put_F = put}
```

```
  where
    get = filter ∘ pred
    put f ss vs = (m' ++ vs, inval)
      where
        (m, m') = partition (pred f) ss
        inval f' = any (pred f') (m ++ vs)
```

It is a polymorphic function that creates a parametric lens based on a predicate function. The predicate function decides whether some value of type a satisfies some properties, defined by a value of type f, or not. Type f becomes the focus domain of the parametric lens, while its value type becomes a list of type a. With this parametric lens, we can effectively filter a list of data, where the parameter of the filtering is represented by the f type.

The implementation of this parametric lens is based on the select lens definition in [8]. In the *get* direction, it simply filters the list using the predicate. In the *put* direction, the original data is split into two: m contains those elements of the original list (the source list, ss) which are affected by the view, that is, replaced by the view data, vs. The unaffected elements are in m'. The new source list is the concatenation of the unaffected data m' and the view data, vs.

The invalidation function encodes the idea that a focus is affected by an update, if any of the replaced or the new records (the list of the unaffected data m and the view data, vs) satisfy the predicate indicated by the focus value.

The filterable views are easy to define now using the *selectLens* function:

```
  -- additional focus domain for Ships
data ShipFilter = SNameEQ String
                | CapacityGT Int deriving Typeable

filteredShips :: PView (Ships, ShipFilter) ShipsT [Ship]
filteredShips = ships `Focus` (selectLens shipPred)

shipPred (CapacityGT c) s = s_capacity s >= c
shipPred (SNameEQ n) s = s_name s == n

  -- additional focus domain for Positions
data PositionFilter = WholeWorld
                    | PNameEQ String
                    | AreaIN (Coord, Coord)
                      deriving Typeable

filteredPositions
  :: PView (Positions, PositionFilter) ShipsT [Position]
filteredPositions = positions `Focus` (selectLens posPred)

posPred (WholeWorld) = True
posPred (AreaIN ((x, y), (x', y'))) p
  = inside (p_position p)
  where
    inside (a, b) = a >= x && a <= x' && b >= y && b <= y'
posPred (PNameEQ n) p = p_ship_name p == n
```

The *ShipFilter* and *PositionFilter* focus domains are rather ad-hoc here. The filtering predicates can be arbitrarily complex which makes it difficult to find the proper data type to encode them. Thus one may want to use a complex, generic type to describe filters. In this case the actual filter values can be given by a small DSL like in Groundhog [23]. However, providing any particular tool here would beyond the scope of this paper, and in this simple case, these types are satisfactory for presentation purpose.

The next step is to join the filterable views together using the *Product* combinator:

```
jointView :: PView
  ((Ships, ShipFilter), (Positions, PositionFilter))
  ShipsT
  ([Ship], [Position])
jointView = filteredShips `Product` filteredPositions
```

Unfortunately, we are not ready yet. The value type of the joint view, $([Ship], [Position])$, is not exactly what we want. When the two lists of records are properly joined together in a relational sense, we expect a value type like $[(Ship, Position)]$.

Depending on the update policy, there are many ways to implement such a relational lens properly [8]. To keep the example illustrative, an oversimplified version of such a lens is given here. Its update policy is that it may add and delete records from both lists, but it is safe only when the relation between the elements of the lists is one to one.

```
joinLens :: (a -> b -> Bool) -> Lens ([a], [b]) [(a, b)]
joinLens by = MkLens {get = get, put = put}
  where
    get (lss, rss) = mapMaybe f lss
      where f ls = (ls, ) <$> find (by ls) rss
    put (lss, rss) vs = (lms' ++ lvs, rms' ++ rvs)
      where
        (lvs, rvs) = unzip vs
        pairLeft = map (λls -> (ls, find (by ls) rss)) lss
        pairRight
          = map (λrs -> (rs, find (flip by rs) lss)) rss
        lms' = dropPaired pairLeft
        rms' = dropPaired pairRight
    dropPaired as = map fst (filter (isNothing ∘ snd) as)
```

Just like *selectLens*, this function is also polymorphic. It creates a relational join by a predicate, the element types of the lists depend on the argument types of the predicate. This time we create a traditional lens as there is nothing to be parametrized here.

The implementation, in the *get* direction, takes the elements of the left list and tries to find a connected record (the first such one) from the right list using the predicate function. If there is no such one, the record is skipped.

The *put* direction is non-obvious even in this simple case. Briefly, we determine which elements from the original lists are skipped by the join (lms' and rms'). These give one part of the modified input. The other part is coming from the modified output, vs. It is unzipped to separate the elements of the left (lvs) and right lists (rvs). For more details we refer to [8].

```
shipPositions :: PView
  ((Ships, ShipFilter), (Positions, PositionFilter))
  ShipsT
  [(Ship, Position)]
shipPositions = jointView `Project` (joinLens by)
  where
    by s = (s_name s ==) ∘ p_ship_name
```

Any of these views can be used to read or update the shared data, depending on the focus one needs. As they are interconnected, updating through *filteredShips* for example, triggers notifications for the other affected views, *ships* and *shipPositions* as well.

However, one must be aware that the accuracy of the notification system depends on the focus value provided explicitly for the interface functions. As an example, consider the *f1* focus value:

```
f1 = ((Ships, SNameEQ "Queen"),
      (Positions, WholeWorld))
```

Requesting notification for this focus of the *shipPositions* view may result in many false notifications. According to the semantics

of the *Product* node, a notification is triggered if the data of the ship "Queen" has been changed in the ship database *or* any data has been changed in the position database (the *WholeWorld* focus covers all the records). In this case one should use the *f2* focus value for improved accuracy:

```
f2 = ((Ships, SNameEQ "Queen"),
      (Positions, PNameEQ "Queen"))
```

The accuracy of the notification system is also affected by the actual implementation of the used lenses. For example, in this section we took a naive approach developing the lenses which is a hidden source of inaccuracy. Let us consider again that we use the *f1* focus value, this time to update the data (capacity and position) of the ship "Queen". Providing only records related to the ship "Queen", we expect that we trigger notifications for domains only which contains that ship. The join lens works on the list of records read from the underlying views using the explicit focus value. It means one record from the ship database (the (*Ships*, *SNameEQ* `"Queen"`) focus value selects exactly one record) and *all* the records from the position database (indicated by the (*Positions*, *WholeWorld*) focus value). The lens does a good job, it silently replaces the position of the ship "Queen" in the list of position records, then delegates the whole list to the underlying view, *filteredPositions*. The problem is that from the point of view of *filteredPositions*, the whole database is changed, thus it triggers notifications for all the notification requests indiscriminately. In the following section we develop a new variant of our generic lenses to overcome this issue.

9. Revised second example

The source of the problem described in the previous section is that the filtered views cannot take into account the *implicit* filtering imposed by the join lens. The list of affected records can be easily calculated in *joinLens*, but how make *selectLens* aware of this information?

Previously, we calculated the list of affected records in the *put* function of *selectLens* and generated the invalidation function based on that. The idea is to calculate this *change list* in the higher level views and pass it to *selectLens* as an argument. Thus the new implementation works on a pair of lists instead of a single list:

```
selectLens :: (f → a → Bool)
           → ParametricLens f [a] ([a],[a])
selectLens pred = MkPLens {get_F = get, put_F = put}
  where
    get f ss = (filter (pred f) ss, [])
    put f ss (vs, cs) = (m' ++ vs, inval)
      where
        m' = filter (¬ ∘ pred f) ss
        inval f' = any (pred f') cs
```

The difference compared to the previous implementation is in the *inval* function. Instead of calculating, we just use the given change list. Applying it to the *ships* view we get the following:

```
filteredShips'
  :: ShipsView (Ships, [ShipFilter]) ([Ship],[Ship])
filteredShips' = ships `Focus` (selectLens shipPred)
```

This is far from being ideal as it still depends on the change list. However, it can be easily calculated in one more step using an additional generic lens:

```
calcChangeList :: Lens ([a],[a]) [a]
calcChangeList = MkLens {get = get, put = put}
  where
    get (ss, cs) = ss
    put (ss, _) vs = (vs, ss ++ vs)
```

It creates a change list just as *selectedLens* did previously: the list of new records in addition to the list of replaced ones. The *filteredShips* view has the same semantics now as previously:

```
filteredShips :: PView (Ships, ShipFilter) ShipsT [Ship]
filteredShips = filteredShips' `Project` calcChangeList
```

If we create *filteredPositions'* the same way and put it together with *filteredShips'* by means of the *Product* combinator we get the following view:

```
jointView :: PView
  ((Ships, [ShipFilter]), (Positions, [PositionFilter]))
  ShipsT
  (([Ship],[Ship]), ([Position],[Position]))
jointView = filteredShips' `Product` filteredPositions'
```

Its type is rather lengthy, but it is an intermediate view only which is not supposed to be exposed. We still have to modify the *joinLens* function to calculate and push down a change list. Its type is as follows:

```
joinLens :: (a → b → Bool)
         → Lens (([a],[a]),([b],[b])) [(a,b)]
```

The actual implementation of this function is not included here, mostly because it is rather long. The modifications are straightforward to develop by *calcChangeList* and the new *selectLens* implementation, and it also provided in the online version.

In a final step, the new *joinLens* must be applied to the new *jointView* view exactly the same way as previously. This gives back the exact same read and update semantics, but radically improves the accuracy of the notifications in certain cases.

10. Related Work

There are three main groups of works closely related to parametric views. These are bidirectional lenses, publish/subscribe systems (including their light-weight counterpart in the object oriented world, the observer design pattern) and functional reactive programming (FRP).

Bidirectional lenses

The first group of related work is bidirectional programming, the so called lenses [12]. A lens is a bidirectional transformation that maps a "concrete" data structure into a simplified "abstract view" and, if the view is updated, maps the modified abstract view, together with the original concrete data, to a correspondingly modified concrete data. In recent years, many extensions (e.g. quotient lenses [13]) and improvements are suggested to the original idea. Most of the improvements would replace the original, *state-based* approach, with a more efficient, *edit-based* one, which work with descriptions of changes to structures, rather than with the structures themselves [7, 8, 18, 35]. As for parametric views, supporting observation of specific changes in the "concrete" data by extended lens definitions, is a novel idea.

Publish/subscribe systems

With systems based on the publish/subscribe interaction scheme [11], subscribers register their interest in an event, or a pattern of events, and are subsequently notified of events generated by the publishers. There are three basic variants of publish/subscribe schemes.

The earliest scheme was based on the notion of *topics*. Observers can subscribe to individual topics, which are identified by

keywords. A later extension enabled topics to be created in a hierarchal manner, giving more expressive power to the event system. Most systems also allow topic names to contain wildcards, thus enabling to publish and subscribe to several topics in the same time. The most notable of these is TIBCO [32].

The *content-based* variant, e.g. Java Message Service [29], extends the expressiveness of the topic-based approach by introducing a subscription scheme based on the actual content of the events. These systems usually offer a subscription language or enables the subscribers to provide a *predicate function* to filter events at runtime.

Finally, the latest approach, the *type-based* variant [10] replaces the name based topic classification model by a scheme that filters events according to their type. Subtyping can be used to achieve hierarchical topic descriptions.

Parametric views use an approach that is a mixture of the type- and content-based variants with additional modifications. Events are typed and carry some type specific properties. The semantics of the properties is described by the invalidation function during the definition of the views. An instance of the invalidation function (based on the actual data) is generated by the update process (the publisher) to filter events. In contrast, in content-based publish/subscribe systems, the predicate function is provided by the subscriber. Another difference is that the whole event system is connected, that is two events can be related in our system, even if their type is different. The conversion of events of different types is also described by the views using the predefined combinators.

Publish/subscribe systems are mostly used in inter-process communication. In object oriented programming the observer design pattern [15] is also used for notification purposes when one wants to stay between the boundaries of an application. In the observer pattern, an object, called the subject or observable, maintains a list of observers, and notifies them automatically if its state changes, usually by calling one of their methods.

In contrast to parametric lenses, the observer pattern does not allow the observers to subscribe for a specific focus. Most implementations, however, enable the observable to pass an arbitrary parameter to the observers along with the change notifications. If this parameter describes the nature of the change (the invalidation function in parametric lens terminology), the observer can use it to decide whether it is interested in the actual notification or not. If the observer classes are parametrized over the type of the value passed to the observer (e.g. in the .NET implementation), this also can be achieved in a type safe manner. This behavior is close to parametric lenses, but in this case, the actual notification logic would be decentralized, it should be scattered over all the observable and observer classes, which is error prone to implement.

Functional reactive programming

The last group of related work is functional reactive programming [9]. FRP is a programming paradigm oriented around time-varying values, called *behaviors*, or *signals*, in a functional programming setting. In FRP, the underlying execution model automatically propagates changes, which makes it similar to our approach. In the original theory a time-varying value is pure, represented as a function of time, thus lenses are not applicable (the data flow is one-way).

There is, however, a state-based approach, when using lenses has a rationale. An example of these frameworks is Scala.React [25], which also can be considered as a superior replacement of the observer pattern. It can help with migrating the observer-based event handling logic with a more declarative implementation.

In Scala.React there are two different kind of signals: *variable signals* and expression signals. Expressions signals are restricted to one-way data flow, but variable signals can be edited. This makes it possible in Scala.React to apply lenses to variable signals and to each other, creating a so called *lens cluster* [24]. This approach is comparable to parametric views, even though the technical details are very different as the Scala language imposes less restrictions on the developer (e.g. lenses are classes, thus have addresses as comparable identifiers).

Another functional reactive framework, Flapjax [28], is built on JavaScript. Flapjax uses lenses, developed for bidirectional tree transformations in [12], to define user interfaces to JavaScript. A piece of the tree based HTML Document Object Model (DOM) can be attached to some structured data model using these lenses. A signal can be created by connecting an actual part of the DOM (denoted by its DOM id) with an actual model object. The system takes care of propagating the changes of the DOM to the models. This can be considered as a special case of our solution (e.g. the source of the shared data is restricted to the DOM) in an FRP setting.

11. Conclusion and Future Work

We have developed *parametric views* to provide read/write access to some shared data according to a parameter, the *focus domain*, which describes which part of the shared data one is interested in, wants to focus on. Parametric views are also *observable*, one can be notified if some part of the underlying shared data is changed, and *compositional*. They are built up using some *combinators* either from *parametric sources*, which describe how to interface with the actual data source in use, or from other parametric views with associated information. To enable observation, we have developed a general extension to bidirectional lenses, called *parametric lenses*; they are attached to parametric views by one of the predefined combinators. We have presented the executable semantics, a naive implementation of our idea, and some basic examples in Haskell. We also have developed a library in Clean that has been integrated into the iTask system [31]; it is used to develop advanced prototypes for the Dutch Coastguard.

Naturally, the progress we have made on parametric views/lenses raises many further challenges. The current implementation is naive. For a production environment, efficiency needs to be considered more carefully. Furthermore, as it is shown in Section 9, the used lens also affect the performance of the system. We need further research to identify the patterns in the development of lenses which increases the accuracy of the notifications.

Currently, the most traditional state-based lens variant is used in our system, but we also would like to experiment with edit-based lenses [18]. The idea of using the *edits* easing the creation of invalidation functions is promising, as edits also describe which part of the data is changed.

Another area of further investigation is the design of additional combinators. One direction to be explored is combining two views by connecting the read value of one to the focus value of the other. Although, we are having a reference implementation of such a combinator, it is not clear yet what most practical type parameters for this combinator are.

Finally, we need to investigate how the usage of parametric lenses affects the performance of certain applications. It is obvious, that using parametric lenses, we can reduce the number of status updates in an application. However, the accuracy of the notification system depends on the actual implementation of the used lenses. Furthermore, the more accurate the system, the more computation must be done on some data during the updates. We need a methodology to be able to find a balance between accuracy and computational complexity.

Acknowledgments

The authors would like to thank Peter Achten for his helpful suggestions and assistance with this manuscript. We also thank the anonymous reviewers for their careful reading of our manuscript and their many insightful comments and suggestions.

References

[1] Generalized algebraic data types, Dec 2014. URL `https://www.haskell.org/ghc/docs/6.6/html/users_guide/gadt.html`.

[2] Monad transformers, Dec 2014. URL `http://book.realworldhaskell.org/read/monad-transformers.html`.

[3] The data.typeable package, Dec 2014. URL `http://hackage.haskell.org/package/base-4.7.0.1/docs/Data-Typeable.html`.

[4] The state monad, Dec 2014. URL `https://www.haskell.org/haskellwiki/State_Monad`.

[5] The writer monad, Dec 2014. URL `https://hackage.haskell.org/package/mtl-2.0.1.0/docs/Control-Monad-Writer-Lazy.html`.

[6] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, Dec. 1981. ISSN 0362-5915.

[7] D. M. Barbosa, J. Cretin, N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: Alignment and view update. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 193–204, 2010. ISBN 978-1-60558-794-3.

[8] A. Bohannon, B. C. Pierce, and J. A. Vaughan. 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, pages 338–347, 2006. ISBN 1-59593-318-2.

[9] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997. URL `http://conal.net/papers/icfp97/`.

[10] P. T. Eugster, R. Guerraoui, and C. H. Damm. On objects and events. *SIGPLAN Not.*, 36(11):254–269, Oct. 2001. ISSN 0362-1340.

[11] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2): 114–131, June 2003. ISSN 0360-0300.

[12] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM TPL*, 29(3), May 2007. ISSN 0164-0925.

[13] J. N. Foster, A. Pilkiewicz, and B. C. Pierce. Quotient lenses. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 383–396, 2008. ISBN 978-1-59593-919-7.

[14] J. N. Foster, B. C. Pierce, and S. Zdancewic. Updatable security views. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*, pages 60–74, 2009. . URL `http://doi.ieeecomputersociety.org/10.1109/CSF.2009.25`.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[16] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Trans. Database Syst.*, 13(4):486–524, Oct. 1988. ISSN 0362-5915.

[17] M. Hofmann, B. Pierce, and D. Wagner. Symmetric lenses. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 371–384, 2011. ISBN 978-1-4503-0490-0.

[18] M. Hofmann, B. Pierce, and D. Wagner. Edit lenses. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 495–508, 2012. ISBN 978-1-4503-1083-3.

[19] M. P. Jones. Functional programming with overloading and higher-order polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 97–136, London, UK, UK, 1995. Springer-Verlag. ISBN 3-540-59451-5. URL `http://dl.acm.org/citation.cfm?id=647698.734150`.

[20] E. Kmett. The lens package, Dec 2014. URL `https://hackage.haskell.org/package/lens`.

[21] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981. ISSN 0362-5915.

[22] B. Lijnse, J. Jansen, R. Nanne, and R. Plasmeijer. Capturing the netherlands coast guard's sar workflow with itasks. In D. Mendonca and J. Dugdale, editors, *Proceedings of the 8th'11*, Lisbon, Portugal, May 2011.

[23] B. Lykah. Groundhog, Dec 2014. URL `http://hackage.haskell.org/package/groundhog`.

[24] I. Maier. Reactive lenses. Oct 2013. URL `http://soft.vub.ac.be/REM13/papers/rem20130_submission_10.pdf`.

[25] I. Maier and M. Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, 2012.

[26] S. Marlow and S. Peyton Jones. The Glasgow Haskell Compiler. In *The Architecture of Open Source Applications, Volume 2*. Lulu, 2012. URL `http://www.aosabook.org/en/ghc.html`.

[27] C. Mcbride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, Jan. 2008. ISSN 0956-7968.

[28] L. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A programming language for ajax applications. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2009*, 2009.

[29] I. S. Microsystems. Java message service, version 1.0.2 (jms specification). Technical report, Sun Microsystems, Inc., 1998. URL `http://java.sun.com/products/jms`.

[30] S. L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987. ISBN 013453333X.

[31] R. Plasmeijer, B. Lijnse, S. Michels, P. Achten, and P. Koopman. Task-oriented programming in a pure functional language. In *Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming*, PPDP '12, pages 195–206, 2012. ISBN 978-1-4503-1522-7.

[32] TIBCO. TIB/Rendezvous. White paper, TIBCO, Palo Alto, CA, 1999.

[33] T. van Laarhoven. Cps based functional references, July 19 2009. URL `http://www.twanvl.nl/blog/haskell/cps-functional-references`.

[34] P. Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, 1995. ISBN 3-540-59451-5.

[35] M. Wang, J. Gibbons, and N. Wu. Incremental updates for efficient bidirectional transformations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 392–403, 2011. ISBN 978-1-4503-0865-6.