Reactive Lenses

Ingo Maier
EPFL
ingo.maier@epfl.ch

Abstract

Functional Reactive Programming (FRP) is a powerful paradigm for writing reactive applications declaratively. Existing FRP implementations are restricted to one-way data flow, forcing programmers to use workarounds in order to implement multi-way data flow, which is common in some domains, such as user interface programming. We show how lenses, originally an abstraction to address the view-update problem in databases, are a natural fit for FRP and can be integrated with time-varying values (aka *behaviors* or *signals*). We present a very simple and efficient two-way propagation algorithm for asymmetric lenses, which does not affect the propagation mechanism for one-way data dependencies.

1. Introduction

Functional Reactive Programming (FRP) [2, 3, 5, 11] is a powerful paradigm for writing reactive applications declaratively. A central abstraction in FRP is that of a time-varying value, or *signal*[1]. In our FRP implementation for the Scala programming language, Scala.React, there are two different kinds of signals: *variable signals* and *expression signals*. A signal holding values of type `A` has base trait `Signal[A]`. Given two integer signals `a` and `b` of type `Signal[Int]`, we can create an expression signal of the sum of `a`'s and `b`'s values that automatically changes whenever `a` or `b` change values as follows:

```
val sum = Signal { a() + b() }
```

Expression signals are restricted to one-way data flow: the signal's value is a function of the value of its dependencies. Consider a simple unit converter as shown in Figure 1. The

Figure 1: The user interface of a unit converter.

user can edit either of the two text fields and the other will change accordingly. This two-way data dependency cannot be expressed in a functional reactive way.

[1] Signals are sometimes called *behaviors* [5, 11].

Variable signals are signals that do not have dependencies, but can be edited explicitly. They have base type `Var[A]` which is a subtype of `Signal[A]` and adds an `update` method to mutate the variable explicitly. One use case for variable signals is at the boundary of an FRP wrapper around a callback-driven framework. Another use case is as a workaround for problems that cannot be addressed in a functional reactive way as, for example, in the unit converter. Instead of using expression signals, a programmer can implement custom event logic in callbacks that take care of updating variable signals explicitly, for example, to implement two-way data flow common in user interface programming. There are two problems with implementing two-way data flow in callbacks, however: it is less declarative and implementing change propagation is cumbersome because of cyclic dependencies (it is even more difficult if glitches are to be avoided). A more declarative approach would be to connect two variable signals so one is the function of another and vice versa. There are two challenges in that. First, we have to make sure that the two functions match, and there is an existing abstraction for that: lenses [1, 7–9]. Second, we have to integrate them into our FRP system. The second problem is two-fold: the programming interface should integrate with that of FRP and we have to extend the efficient one-way propagation mechanism of our FRP implementation with a two-way mechanism. This work addresses these two challenges.

1.1 Contributions

We introduce the concept of *asymmetric reactive lenses* to integrate two-way data-flow into Scala.React. Reactive lenses extend the functional reactive programming style of creating dependency nodes together with their dependencies to multi-way data flow. Our contributions are as follows:

- We show how to integrate lenses into a one-directional data flow evaluation model such as Scala.React. We present a mechanism to separate multi-way from one-way dependencies based on the types of signals. This type-based separation allows us to avoid runtime detection of data-flow cycles and its associated performance penalty in previous work [13].

- Asymmetric reactive lenses are not only asymmetric because they distinguish between a model and a view [8], but also because they let one of the two variables they connect depend on other time-varying values. In other words, they seamlessly integrate with the one-way dataflow model of the host FRP system Scala.React.
- We show how our approach leads to a simple and efficient propagation mechanism.

2. Background

2.1 Scala.React

The expression in the curly braces of the sum signal above is a common Scala closure without parameters. Function call syntax `x()` for any `x` is rewritten by the Scala compiler to method call `x.apply()`. The `apply` method defined in trait `Signal` does all dependency management. Since Scala.React is a library, it does not depend on domain specific compile-time magic. Dependencies are instead tracked during the evaluation of signals. When the `sum` signal from above is evaluated, it puts a reference to itself onto a hidden dependency stack, which is read by method `Signal.apply` in order to register `sum` as a dependent of `a` and `b`. As in FrTime, forward references from dependencies to dependents are stored in weak references. Backward references in expression signals are captured implicitly in the closure argument of a signal constructor and are therefore strong. Consequently, dependencies cannot vanish spuriously and do not prevent dependents from being garbage collected.

Propagation in Scala.React is generally push-based [4] using topologically sorted dependencies, similar to FrTime, Flapjax or Amulet [2, 11, 13]. Signals[2] can be evaluated strictly, i.e., whenever a dependency has changed, or lazily, i.e., not before their value is needed. For this, we use a *tick-tock* propagation model. A *tick* notifies a node in the dependency graph to be invalidated, a *tock* to be evaluated. Strict signals will request to receive a tock when ticked, lazy signals will never receive a tock but will be evaluated when queried, for example, via method `Signal.apply`. A node in the dependency graph is represented by an instance of a trait `Node`, which exposes the `tick` method, and an instance of trait `Master`, which exposes the `tock` method. Expression signals as the `sum` signal above, are both a node and a master (using Scala mixin composition). A master, however, may contain multiple nodes and maintain dependencies between them, independently from the dependency graph maintained by the main framework. This master-node separation is used in our work on incremental collections [10]. See Section 5 for how we can also use it to embed two-way propagation for subgraphs into Scala.React.

[2] Or more generally nodes in the dependency graph. Scala.React supports further abstractions besides signals.

2.2 Lenses

Lenses are an abstraction for well-behaved bidirectional transformations and have been studied as a solution to the view-update problem of databases and related problems. They were first introduced by [6] and have been studied in many variations since [1, 7**?** –9].

A lens l is a pair of functions, each mapping into one of two directions in order to bidirectionally relate two values:

$$l.\mathit{view}\colon M \to V$$
$$l.\mathit{model}\colon M \times V \to M$$

We refer to M as the set of *models* and V as the set of *views* and write $l \in M \leftrightarrow V$. The *view* function simply obtains a view from a model. The *model* function integrates changes from the view back into the model. Since a view may abstract from the model and not persist all necessary information to obtain a model from the view alone, the *model* function also takes the model as an argument. One example is an address book manager that displays the name of a person from a larger record that also includes address, birthday, etc. The view would be just the name. Once the name changes, we also need the original model, i.e., person record in order to obtain a correctly updated person record.

The functions of a lens must fulfill certain properties, called *lens laws*, such that a lens is sensical and such that the composition of lenses can preserve these properties:

$$l.\mathit{model}(l.\mathit{view}(m), m) = m \qquad \text{(VIEWMODEL)}$$
$$l.\mathit{view}(l.\mathit{model}(v, m)) = v \qquad \text{(MODELVIEW)}$$

They state that roundtrips trough the lens do not lead to surprising results. Property VIEWMODEL states that getting the model from an unchanged view should give an unchanged model. Property MODELVIEW states that integrating changes from the view into the model should not lose any information and we should therefore be able to obtain the same view from the model.

Any two lenses $l_1 \in A \leftrightarrow B$ and $l_2 \in B \leftrightarrow C$ can be concatenated to a lens $l = l_1 \circ l_2 \in A \leftrightarrow C$ that applies its argument lenses one after each other in both directions:

$$l.\mathit{view}(m) = l_2.\mathit{view}(l_1.\mathit{view}(m))$$
$$l.\mathit{model}(v, m) = l_1.\mathit{model}(l_2.\mathit{model}(v, l_1.\mathit{view}(m)), m)$$

It can be shown that l obeys the lens laws from above. This makes lenses compositional and *correct by construction*. It enables us to reason about the effects of a lens application locally. They also nicely fit into the mostly functional reactive programming model of Scala.React, as we will demonstrate now.

3. Simple lenses in Scala

In Scala, we can naturally represent a lens by an instance of a trait `Lens[M,V]` with two abstract methods:

```
trait Lens[M, V] {
  def toView(m: M): V
  def toModel(v: V, m: M): M

  def compose[W](lens: Lens[V,W]): Lens[M,W]
}
```

Method `compose` concatenates the enclosing lens and argument lens in the way defined above. The `toModel` function of some lenses does not depend on the original model. In that case, the lens is called *bijective* because it degenerates to a bijective function due to the lens laws above. A bijective lens is an instance of the following trait:

```
trait BijectiveLens[M, V] extends Lens[M, V] {
  def toView(m: M): V
  def toModel(v: V): M
  def toModel(v: V, m: M): M = toModel(v)

  def inverse: BijectiveLens[V, M]
  def compose[W]
      (that: BijectiveLens[V, W]): BijectiveLens[M, W]
}
```

Bijective lenses can be trivially inverted by means of `inverse`, swapping the `toView` and `toModel` functions. The concatenation of two bijective lenses is bijective again. Therefore, trait `BijectiveLens` overloads method `compose`. An example of a bijective lens is the addition lens defined as follows:

```
class AddLens[A](k: A)(implicit num: Numeric[A])
  extends BijectiveLens[A, A] {
  def toView(m: A): A = num.plus(m, k)
  def toModel(v: A): A = num.minus(v, k)
}
```

It adds a constant `k` towards the view and subtracts it towards the model. It expects an implicit argument `num` of type `Numeric[A]`, which models a type class [12] and provides arithmetic operators for operands of type `A`. Type `Numeric[A]` does not explicitly require that `x.plus(k).minus(k) == x` for all `x` and `k`. Even though this is a reasonable assumption, whether an `AddLens` instance obeys the lens laws really depends on the implementation of the given `Numeric` argument. In this specific case, we are therefore trading provably correct lenses for a practical integration with existing abstractions from the Scala standard library. Also note that class `AddLens` defines a set of lenses, one for each argument pair `k` and `num`.

Many mathematical properties of arithmetic operations do not hold for floating point representations. Sometimes, it makes sense to try and "fix" some of these properties. A multiplication lens, for example, can be defined as follows:

```
class MulLens[A](k: A)(implicit frac: Fractional[A])
  extends Lens[A, A] {
  if(k == 0) throw new IllegalArgumentException(
      "Illegal␣lens:␣mul/div␣by␣zero")

  def toView(m: A): A = frac.times(m, k)
  def toModel(v: A, m: A): A = {
    val res = frac.div(v, k)
    if (frac.equiv(res, m)) m
    else res
  }
}
```

It expects an implicit parameter of type `Fractional[A]`, which defines multiplication and division in methods `times` and `div`. The constructor throws an exception if the coefficient is zero, since in that case there is no way to fulfill the lens laws. The `toModel` method is a little peculiar. The `equiv` method from trait `Fractional` checks for equality. If it is implemented to check for equality plus/minus an epsilon, the lens will remove rounding errors during division by taking the model if the new view and model are close enough.

We can similarly define many other lenses for arithmetic operations. Other examples are lenses for projecting out individual components from tuples or case classes or more structured data. In practice, we can also define composition operations other than concatenation. Foster et. al [1], for example, define lens operators that lead to a bidirectional language for un-/pickling string data that looks not unlike regular expressions. Concatenation, however, is the fundamental composition operation that allows us to create composable multi-way dependencies as we will show now.

4. Reactive Lenses

We can create a variable which serves as the model for a lens with the following function:

```
def LVar[A](init: A): LVar[A]
```

Class `LVar` is a subclass of `Var` and therefore also `Signal` and defines method `applyLens` to create a new lens variable that is connected to the original one through the given lens:

```
def applyLens[B](lens: Lens[A, B]): LVar[B]
```

The resulting lens variable will be a view of the enclosing model variable. We define appropriate implicit conversions to make lens application syntactically convenient. For example, for the multiplication lens above, we define the following implicit conversion and wrapper class:

```
implicit def toFracLVar[A : Fractional]
                       (lvar: LVar[A]) =
  new FractionalLVar(lvar)

class FractionalLVar[A: Fractional](lvar: LVar[A]) {
  def *(k: A) = lvar.applyLens(new MulLens(k))
}
```

With this in scope, a lens variable of type `LVar[A]` can be implicitly converted to an instance of `FractionalLVar[A]`, if there is a `Fractional` instance for type `A` in scope as well. We can define a similar conversion for a `Numeric` instance and operators `+`, `-` and unary `-` (negation). The Scala standard library provides appropriate instances for all numeric primitive types such as `Int`, `Double`, etc.

With the above implicit conversion in scope, the bidirectional event logic of a unit converter can now be written as follows:

```
val yards = LVar(1.0)
val metres = yards * 0.9144
```

Since class `LVar` does not define a method with name `*`, the compiler looks for an implicit conversion from an `LVar` instance to one that defines an appropriate `*` method. It finds `toFractionalVar` and inserts an application in the last line as follows:

```
val metres = toFractionalLVar(yards) * 0.9144
```

Using a subtraction lens `SubtractLens` – defined similarly to `AddLens` above – with the following additional implicit conversion

```
implicit def toNumLVar[A : Numeric](lvar: LVar[A]) =
  new NumericLVar(lvar)

class NumericLVar[A: Numeric](lvar: LVar[A]) {
  def -(k: A) = lvar.applyLens(new SubtractLens(k))
  ...
}
```

a two-way conversion between Celsius and Fahrenheit is straightforward as well:

```
val celsius = LVar(20.0)
val fahrenheit = celsius * 1.8 - 32
```

The compiler inserts implicit conversions in the last line as follows:

```
val fahrenheit =
  toNumLVar(toFractionalLVar(celsius) * 1.8) - 32
```

4.1 Asymmetric signal dependencies

In a more refined unit converter as the one shown in Figure 2, the user can not only change the magnitudes but also the associated units. In this case, the data flow becomes more complex. When the user changes the value in the left text field, the value in the right text field is changed according to the selected units in the combo boxes at the bottom. When the user changes the value in the right text box, the value in the left text box is changed according to the selected units. This is what we have modeled so far. Additionally, there is the following data flow.

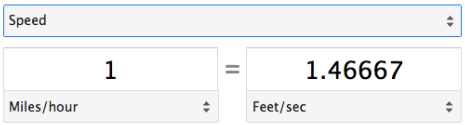


Figure 2: The user interface of an extended unit converter.

When the user changes the unit in either of the combo boxes at the bottom, the value of the *right* text box changes according to the selected units. Moreover, when the user chooses a different set of units to select from in the top combo box, the bottom unit boxes change, again causing the right text box to synchronize with the left field, using the new units. Therefore, whenever a change occurs that may affect the values in both text boxes, the current value of the left text box remains unchanged and only the value in the right text box is changed. We have examined several unit converters and all expose precisely this fixed, asymmetric behavior, even though there are viable alternatives. For example, the converter could always convert from the last edited text box and change the other box. We believe the behavior we found is popular for two reasons. It is relatively simple to implement since it does not involve state management for which box has last recently changed. It also seems to be the least surprising and therefore most intuitive solution for the user.

We can extend the built-in asymmetry in our lenses stemming from the model-view distinction, and integrate them with the one-way propagation model of the host framework. For each arithmetic lens we have defined above and which takes a constant parameter, we define a corresponding *signal lens*, that takes a signal instead of a constant parameter. The addition lens, for example, becomes:

```
class AddSigLens[A](k: Signal[A])
                   (implicit num: Numeric[A])
                   extends Lens[A, A] {
  def toView(m: A): A = num.plus(m, k())
  def toModel(v: A, m: A): A = num.minus(v, k.now)
}
```

Note that we apply the signal towards the view, and take the current value without applying the signal towards the model. This means the signal is applied only when the lens's view variable is evaluated and not for its model. Therefore, only the view variable establishes a dependency with parameter signal `k`. On a change in `k`, only the view is changed and subsequently all variables that are views of that view and so on. In order to keep this unidirectional signal dependency, it is very important that all signal lenses apply signals only in the view direction, since it allows for an efficient implementation as we will show below.

Conceptually, a signal lens defines a set of lenses, one lens for each value combination in its parameter signals. Each lens in this set must obey the lens laws. A value change in any parameter signal conceptually replaces the lens for the previous current values of the lens's parameter signals. This gives rise to a flattening operation, which flattens a signal of a lens to a lens:

```
implicit def flattenSignalOfLens[M, V]
             (sig: Signal[Lens[M, V]]) =
  new Lens[M, V] {
    def toView(m: M): V = sig().toView(m)
    def toModel(v: V, m: M) = sig.now.toModel(v, m)
  }
```

This implicit conversion creates a lens that always has the behavior of the current lens in the given signal. If all lenses in the given signal of lenses obey the lens laws, the resulting lens therefore also obeys the lens laws. We also overload the corresponding convenience methods in wrapper classes such as `FractionalLVar` with versions that take a signal as a parameter.

We can now implement the data flow in the extended unit converter as follows.

```
class UnitOfMeasure(name: String, factor: Double)
val unit1, unit2: Signal[UnitOfMeasure]

val factor = Signal {
  unit1().factor / unit2().factor
}
val magnitude1: LVar[Double] = LVar(1.0)
val magnitude2: LVar[Double] = magnitude1 * factor
```

We first define a class that represents units of measures in terms of their name and factors which define the unit conversion rate to the corresponding SI base unit. We obtain the selected units from the unit combo boxes via two signals `unit1` and `unit2`. The conversion rate between the two units is a quotient from each unit's factor. Since the units are signals, the resulting factor is a signal as well, which we use as the conversion coefficient to create the view variable in the last line.

In practice, some conversions such as the Fahrenheit/Celsius conversion above are more complex than simple products and quotients. To reflect this, we can replace the `factor` parameter for the unit of measure class by two functions mapping to and from the corresponding SI base unit – which is simply another lens. The example becomes:

```
class UnitOfMeasure(name: String,
      toSIBase: BijectiveLens[Double, Double])
val unit1, unit2: Signal[UnitOfMeasure]

val lensSig: Signal[Lens[Double, Double]] = Signal {
  unit1().toSIBase compose unit2().toSIBase.inverse
}
val magnitude1 = LVar(1.0)
val magnitude2 = mag1 applyLens lensSig
```

Class `UnitOfMeasure` now stores a bijective lens to map to and from the SI base unit. The interesting bit is the signal expression, which composes the conversion lenses, with the second one reversed, since the left lens maps towards the SI base unit and the right away from the SI base unit. All other lines remain unchanged. The compiler, however, inserts the above flattening conversion in the last line to convert the `lens` signal of a lens to a lens.

We can take this example even further. The `toSIBase` lens can in fact be a signal lens, i.e., the conversion rate can change. This can be useful if the unit converter supplies editing facilities to the user to add and modify conversion rates. Another example would be a currency converter continuously pulling time-varying conversion rates from a server. However, we have to pay attention to the `inversed` lens in the above example. The inverse operation of a bijective signal lens needs to swap the `Signal.apply` and `Signal.now` calls in methods `toModel` and `toView`. However, this cannot be done automatically in general, so the definition of a bijective signal lens needs to supply four methods. The `AddSigLens` from above is actually defined as follows:

```
class AddSigLens[A](k: Signal[A])
                   (implicit num: Numeric[A])
      extends BijectiveSigLens[A, A] {
  def toView(m: A): A = num.plus(m, k())
  def toViewNow(m: A): A = num.plus(m, k.now)
  def toModel(v: A): A = num.minus(v, k.now)
  def toModelSig(v: A): A = num.minus(v, k())
}
```

where `BijectiveSigLens` is defined as

```
abstract class BijectiveSigLens[M, V]
         extends BijectiveLens[M, V] {
  def toViewNow(m: M): V
  def toModelSig(v: V): M

  override def inverse: BijectiveLens[V, M] =
    new InvertedSigLens(this)
}

class InvertedSigLens[M, V]
        (inv: BijectiveSigLens[V, M])
        extends BijectiveSigLens[M, V] {
  def toView(m: M): V = inv.toModelSig(m)
  def toViewNow(m: M): V = inv.toModel(m)
  def toModel(v: V): M = inv.toViewNow(v)
  def toModelSig(v: V): M = inv.toViewNow(v)
}
```

Note how class `InvertedSigLens` makes sure that the signal is applied only towards the view. Non-bijective signal lenses are not reversible and are therefore not affected by the above consideration.

5. Lens clusters

The programming interface of lens variables leads to a separation of *lens clusters* each with tree-shaped lens dependencies. We call a lens variable created with constructor `LVar(init)` the *root* of a lens cluster. A variable created like this implicitly creates a new cluster. Any other lens in the same cluster is created from a single other lens variable through a lens application with `LVar.applyLens`. Lens dependencies therefore have a tree shape as depicted in Figure 3. Every lens variable, except the root, has precisely one model and can be the model for multiple views. A lens cluster is represented by a master in the implementation. Lens variables can depend on other lenses from the same cluster as well as signals from other masters if the lens variables were created with a signal lens. Other signals can depend on lens variables, since lens variables are also signals. The root of a

lens cluster cannot depend on other signals, since it is created with the `LVar` constructor. Lens dependencies are maintained by their master, dependencies to signals in other masters are one-way and maintained by the reactive framework.

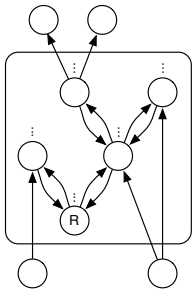


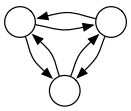
Figure 3: The tree shape of a lens cluster with root R. Lenses are represented as parallel edges pointing into opposite directions. At the bottom outside of the cluster are signals that are applied by signal lenses. At the top are signals that depend on lens variables, e.g., via common signal expressions.

5.1 Lens cycles

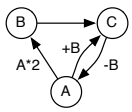
There is no way to connect two existing lens variables through a lens – neither from the same nor from different lens clusters. Therefore, we cannot create a lens cycle as in Figure 4a. This results in a programming style very similar to FRP, but with multi-way data flow. Lens variables in the same cluster are all on the same topological level (maintained by the master) in the one-way dependency graph of Scala.React. This constraint makes the system reject certain dependency structures at runtime. Consider the following example:

```
val a = LVar(1)
val b = Signal { 2 * a() }
val c = a + b
```

This creates a signal `b` from lens variable `a` and uses it as a lens parameter in the last line. This leads to the dependency graph in Figure 4b and will throw an exception at runtime since it cannot be topologically sorted. Variable `b` is one level above `a` in the one-way dependency graph. Since `c` is in the same master as a and therefore restricted to be on the same level as a, it will throw an exception.



(a) Impossible to create.



(b) A cyclic lens $C = A + B$ where $B = 2 * A$. Throws exception at runtime.

Figure 4: Example of the two different kinds of lens cycles.

This behavior allows for an efficient implementation of propagation in a lens cluster and keeps the system free from propagation cycles. Consider the hypothetical example in Figure 5, which shows the evaluation steps that would be necessary to update a cyclic lens graph. Initially, the graph is in a consistent state. Once lens variable C gets updated to 4, the lens master needs to propagate this change through the graph, which updates nodes and eventually the result of the lens functions, since they not only depend on the lens model at the bottom but also on the node on the upper left. In order to keep everything consistent, the node that has been edited to 4 eventually needs to change its value again, this time to 6, closing a propagation cycle.

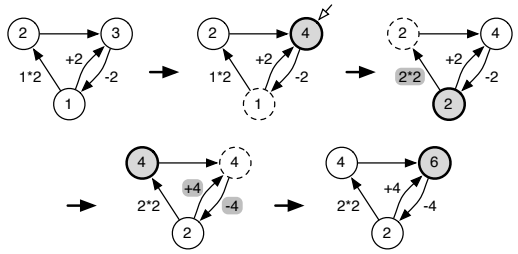


Figure 5: Sample evaluation of a lens cycle, where the lens view needs to be evaluated twice. Last recently updated nodes are highlighted, invalidated nodes are dashed.

5.2 Lens order

The asymmetry of reactive lenses and the tree-shaped dependency structure in a lens cluster impose an order on lenses and lens variables. We use this order for three different purposes: propagation order, disambiguating simultaneous edits, and memory management.

5.2.1 Propagation order

We have seen above that a lens variable that is not the root of its cluster can depend not only on another lens but also on external signals from other masters. The lens propagation mechanism has to distinguish between invalidations coming from another lens variable and coming from an external signal. If a lens variable is ticked by an external signal, it propagates the change away from the root. This reflects the asymmetry of data flow as in the unit converter from above. If a lens variable is notified of a change in another lens variable in the same cluster, we have to distinguish between two cases. If the other lens variable is closer to the root, the change is propagated away from the root. If the other lens variable is further away from the root, the change is propagated towards the root. Once such a change reaches the root, it is propagated away from the root to reach all other variables, ignoring the path from which the change originally came. Note that for simplicity, lens propagation is always strict.

Figure 6 shows a propagation example for a lens cluster. The root is initialized with a value of 0. Every other lens variable is created from another lens variable using the `AddLens` from above with a constant parameter of 1. Once the upper right variable receives an explicit update, it changes the variable's value and invalidates all lens variables it is connected to. In this case it has a single model variable. Once this model variable updates itself, it also invalidates all variables it is connected to except the one it just has been notified by. This process proceeds along the path towards the root. In the example, this leaves two nodes invalidated once the root has been updated. The propagation process continues from these invalidated nodes and away from the root.

We somehow have to keep track of which node has been invalidated and which has been updated already. Since we don't have propagation cycles as in Figure 5, a lens variable is updated only once per propagation turn and can simply maintain a flag of whether it has been validated in the current turn or not. Since lens dependencies have a tree shape, we can keep track of invalid nodes and the direction of propagation through lens levels and a priority queue similar to the approach in the one-way propagation of expression signals. Note that the lens levels are different from the master levels and are equal to the distance of a variable to the root in its lens cluster.

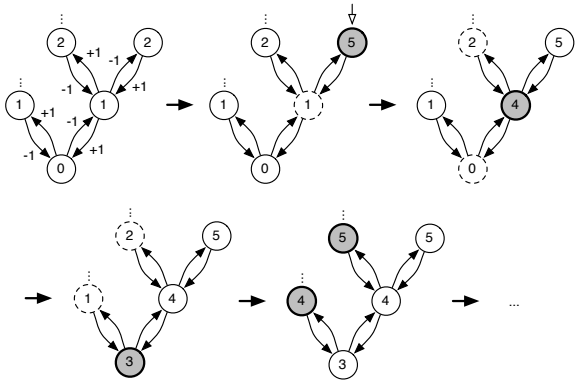


Figure 6: Sample evaluation of a lens cluster from an update arriving at a node different from the root. Each lens adds 1 for each variable one step further away from the root. Most recently updated nodes are highlighted, invalidated nodes are dashed.

5.2.2 Simultaneous updates

Since `LVars` are also `Vars` and can be edited explicitly, we have to prepare for the fact that there might be conflicting updates to different lenses in the same cluster and propagation turn. We disambiguate between such edits by taking the one that affects the node closest to the root and dropping all other edits. If two edits affect two different nodes that have the same distance to the root, we call a method `conflictingLensEdit` that can be overridden by clients to resolve the conflict gracefully. By default it throws an exception.

5.2.3 Memory management

Due to the asynchronous nature and the way we create lens variables, the issue of memory management for lenses is not much different than for the one-way dependency graph of expression signals as described in the introduction. Node dependencies away from the root are all weak references, dependencies towards the root are strong. This is similar to an expression signal which refers to all its dependencies strongly, which in turn refer to the expression signal weakly. Nothing in the lens framework persists lens variables permanently. Once an application releases all strong references to a lens variable v and all of $v's$ dependencies away from the lens root, the lens subtree with variable v at the root can be garbage collected.

References

[1] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In *POPL*, 2008.

[2] Gregory H. Cooper. *Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language*. PhD thesis, Brown University, 2008.

[3] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa arcade. In *Haskell*, 2003.

[4] Conal Elliott. Push-pull functional reactive programming. In *Haskell*, 2009.

[5] Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP*, 1997.

[6] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17, 2007.

[7] J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. In *ICFP*, 2008.

[8] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. Symmetric lenses. In *POPL*, 2011.

[9] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. Edit lenses. In *POPL*, 2012.

[10] Ingo Maier and Martin Odersky. Higher-order reactive programming with incremental lists. In *ECOOP*. 2013.

[11] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for ajax applications. *OOPSLA*, 2009.

[12] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *OOPSLA*, 2010.

[13] Brad Vander Zanden, Brad A. Myers, Dario A. Giuse, and Pedro Szekely. Integrating pointer variables into one-way constraint models. *ACM Trans. Comput.-Hum. Interact.*, 1994.