Dr. Formlens, Or: How I Learned to Stop Worrying and Love Monoidal Functors

Raghu Rajkumar Nate Foster
Cornell University

Sam Lindley James Cheney
The University of Edinburgh

Abstract

To make data available on the Web, one must typically implement two components: one to convert the data into HTML, and another to parse updates out of client responses. In current systems, these components are usually implemented using separate functions—an approach that replicates functionality across multiple pieces of code, making programs difficult to write, reason about, and maintain. This paper presents *formlenses*, a new abstraction based on *formlets* that makes it easy to bridge the gap between data stored on a server and values embedded into an HTML form. We present a new foundation for formlets based on monoidal functors (replacing the classic definition based on applicative functors), and show how to endow the resulting structures with a bidirectional semantics. We investigate the connection between linearity and bidirectional transformations and develop a translation from a linear pattern syntax into formlens combinators. Finally, we develop infrastructure for building formlenses over arbitrary algebraic datatypes.

1. Introduction

Putting data on the Web typically involves implementing two components: one to convert the data into HTML, and another to parse updated data from client responses. Unfortunately, in current systems, these components are usually implemented using separate functions—an approach that replicates functionality across multiple pieces of code, and makes programs difficult to write, reason about, and maintain.

To illustrate, consider a datatype representing visiting speakers,

$$\begin{array}{l}\mathbf{data}\ \mathit{Speaker} = \mathit{Speaker}\ \{\mathit{name} :: \mathit{String}, \mathit{date} :: \mathit{Date}\}\\ \mathbf{data}\ \mathit{Date} = \mathit{Date}\ \{\mathit{year} :: \mathit{Int}, \mathit{month} :: \mathit{Int}, \mathit{day} :: \mathit{Int}\}\end{array}$$

and suppose we want to build an application that supports viewing and editing a database of such speakers through a Web browser. The first step, would be to write a function *renderer* that converts a single *Speaker* value into HTML:[1]

$$\begin{array}{l}\mathit{renderer} :: \mathit{Speaker} \to \mathit{Html}\\ \mathit{renderer}\ (\mathit{Speaker}\ n\ (\mathit{Date}\ y\ m\ d)) =\\ \quad \mathit{input}\,!\,[\mathit{name}\ \texttt{"name"}, \mathit{value}\ n] +\!\!\!+\!\!\!+ \mathit{br} +\!\!\!+\!\!\!+\\ \quad \mathit{lineToHtml}\ \texttt{"Year: "} +\!\!\!+\!\!\!+\\ \quad \mathit{input}\,!\,[\mathit{name}\ \texttt{"year"}, \mathit{value}\ (\mathit{show}\ y)]\\ \quad \mathit{lineToHtml}\ \texttt{"Month: "} +\!\!\!+\!\!\!+\\ \quad \mathit{input}\,!\,[\mathit{name}\ \texttt{"month"}, \mathit{value}\ (\mathit{show}\ m)] +\!\!\!+\!\!\!+ \mathit{br} +\!\!\!+\!\!\!+\\ \quad \mathit{lineToHtml}\ \texttt{"Day: "} +\!\!\!+\!\!\!+\\ \quad \mathit{input}\,!\,[\mathit{name}\ \texttt{"day"}, \mathit{value}\ (\mathit{show}\ d)] +\!\!\!+\!\!\!+ \mathit{br} +\!\!\!+\!\!\!+\end{array}$$

The output produced by *renderer* contains an embedded form, which allows users to edit the details for the speaker and submit modifications back to the server. When the form is submitted, a response will be sent back to the server as an association list:

$$\mathbf{data}\ \mathit{Env} = [(\mathit{String}, \mathit{String})]$$

To handle these responses, we also need a function *collector* that extracts an updated *Speaker* value from an *Env*.

$$\begin{array}{l}\mathit{collector} :: \mathit{Env} \to \mathit{Speaker}\\ \mathit{collector}\ e =\\ \quad \mathbf{let}\ n\ = \mathit{read}\ (\mathit{fromJust}\ (\mathit{lookup}\ \texttt{"name"}\ e))\ \mathbf{in}\\ \quad \mathbf{let}\ y\ = \mathit{read}\ (\mathit{fromJust}\ (\mathit{lookup}\ \texttt{"year"}\ e))\ \mathbf{in}\\ \quad \mathbf{let}\ m = \mathit{read}\ (\mathit{fromJust}\ (\mathit{lookup}\ \texttt{"month"}\ e))\ \mathbf{in}\\ \quad \mathbf{let}\ d\ = \mathit{read}\ (\mathit{fromJust}\ (\mathit{lookup}\ \texttt{"day"}\ e))\ \mathbf{in}\\ \quad \mathit{Speaker}\ n\ (\mathit{Date}\ y\ m\ d)\end{array}$$

This function retrieves the appropriate items from the *Env*, parses them back into their original formats, and builds a *Speaker* value out of the results.

Together, *renderer* and *collector* effectively present a single *Speaker* value on the Web. But in general, developing Web applications manually leads to several challenges. First, it requires the programmer to write explicitly coercions to convert the data into and out of HTML. And often the correctness of the application requires that these coercions compose to the identity—something that is easy to get wrong, especially in large applications. Second, the programmer must construct the Web form manually, including choosing the names of each element. Although these names are semantically immaterial—the program would behave the same if different names were used—they must be globally unique to avoid clashes. In addition, the names introduced in the *renderer* function must be synchronized with the names used in the *collector* function to ensure that data is preserved on round-trips. These constraints make it difficult to build forms in a compositional manner. For instance, we cannot iterate the *renderer* and *collector* functions to obtain a program for a list of *Speaker*s because the names are "baked in" to the two functions.

Formlets. In previous work, Cooper et al. [10] introduced a high-level abstraction for building Web forms called *formlets*. formlets encapsulate a number of low-level details including selecting names for elements and parsing responses.

In Haskell, formlets are represented as functions that take a name source (concretely, an *Int*) as an argument and produce a triple comprising HTML, a collector function, and a modified namesource.

$$\mathbf{type}\ \mathit{Formlet}\ a = \mathit{Int} \to (\mathit{Html}, \mathit{Env} \to a, \mathit{Int})$$

The names of any form elements in the HTML are drawn from the namesource, and the collector looks up precisely these names.

As a simple example of a formlet, consider the *html* combinator, which generates static HTML without any embedded form elements,

[1] This code uses combinators from the $\mathit{Text.Html}$ module.

$$html \;\; :: Html \rightarrow Formlet\ ()$$
$$html\ h\ i = (h, const\ (), i)$$

and a wrapper for generating plain text:

$$text :: String \rightarrow Formlet\ ()$$
$$text = html \circ stringToHtml$$

We can also define a formlet combinator that accepts an integer:

$$\begin{aligned} &inputInt \;\; :: Formlet\ Int \\ &inputInt\ i = \mathbf{let}\ n = show\ i\ \mathbf{in} \\ &\qquad (input\,!\,[name\ n], \\ &\qquad\ \lambda e \rightarrow read\ (fromJust\ (lookup\ n\ e)), i + 1) \end{aligned}$$

Combinators for other primitive types like $String$ and $Bool$ can be defined in similar fashion.

Formlets can be combined into larger formlets using the interface of *applicative functors*, a useful and generic mathematical structure for representing computations with effects [26].

$$\begin{aligned} &\mathbf{class}\ Functor\ f\ \mathbf{where} \\ &\quad fmap\ f :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b \end{aligned}$$

$$\begin{aligned} &\mathbf{class}\ (Functor\ f) \Rightarrow Applicative\ f\ \mathbf{where} \\ &\quad pure :: a \rightarrow f\ a \\ &\quad (\otimes)\ \ :: f\ (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b \end{aligned}$$

Intuitively, the $pure$ function injects a value of type a into the type $f\ a$, while the $\otimes$ operator applies the underlying function to the underlying argument, accumulating effects from left to right. Applicative functor instances are expected to obey the following natural conditions which relate the behavior of $pure$ and $\otimes$:

$$\begin{aligned} pure\ id \otimes u &= u \\ pure\ (\circ) \otimes u \otimes v \otimes w &= u \otimes (v \otimes w) \\ pure\ f \otimes pure\ x &= pure\ (f\ x) \\ u \otimes pure\ x &= pure\ (\lambda f \rightarrow f\ x) \otimes u \end{aligned}$$

The applicative instance for formlets is defined as follows:

$$\begin{aligned} &\mathbf{instance}\ Applicative\ Formlet\ \mathbf{where} \\ &\quad pure\ a\ i\ = (noHtml, const\ a, i) \\ &\quad (f \otimes g)\ i = \mathbf{let}\ (x, p, i') = f\ i\ \mathbf{in} \\ &\qquad\qquad\quad \mathbf{let}\ (y, q, i'') = g\ i'\ \mathbf{in} \\ &\qquad\qquad\quad (x +\!\!\!+\!\!\!+ y, \lambda e \rightarrow p\ e\ (q\ e), i'') \end{aligned}$$

The $pure$ formlet generates empty HTML and has the constant function as its $collector$. The $\otimes$ operator threads the namesource through its arguments from left to right and accumulates the generated HTML. Its collector function applies the function (of type $a \rightarrow b$) produced by the collector for f to the value (of type a) produced by the collector for g.

Using the applicative functor interface and the simple combinators just defined, we can define a formlet for $Speaker$s as follows:

$$\begin{aligned} &dateForm :: Formlet\ Date \\ &dateForm = pure\ (\lambda\_\ y\ \_\ \_\ m\ \_\ \_\ d\ \_ \rightarrow Date\ y\ m\ d) \\ &\quad \otimes text\ \texttt{"Year: "} \otimes inputInt \otimes html\ br \\ &\quad \otimes text\ \texttt{"Month: "} \otimes inputInt \otimes html\ br \\ &\quad \otimes text\ \texttt{"Day: "} \otimes inputInt \otimes html\ br \end{aligned}$$

$$\begin{aligned} &speakerForm :: Formlet\ Speaker \\ &speakerForm = pure\ Speaker \otimes inputString \otimes dateForm \end{aligned}$$

The $\_$ arguments to the $pure$ function ignore the $()$ values produced by the collector for the $text$ and $html$ combinators.

Formlenses. Formlets are a useful abstraction, but they only address half of the problem! They make it easy to describe a collector function that *produces* a value of type a, but they do not provide a way to describe a renderer function that *consumes* a value of type a and embeds it in a form. Of course, programmers could write functions of type $a \rightarrow Formlet\ a$, but such functions do not have an applicative functor interface, so large formlets would have to be built by hand. Moreover, formlets do not help programmers establish that these transformations will have reasonable behavior—*e.g.*, that the a value will be preserved on round-trips.

Another way to think about these problems starts from the observation that forms (and formlets) are often used in the context of an *updatable view* of an underlying data source. The fundamental difficulty in putting data on the Web stems from the fact that programmers are maintaining these views manually, writing explicit forward transformations that put the data into forms, and separate backward transformations that propagate collected values back to the underlying sources.

As the terminology used in the preceding paragraph should suggest, we have a solution to this problem in mind: use ideas from *bidirectional transformations* [11] to define formlets that behave like updatable views. The primary goal of this paper is to show how formlets and bidirectional transformations such as lenses [15] can be combined, yielding an abstraction we call $Formlens$es. Essentially, a formlens is a gadget that takes a value of type a, renders it as a form that can be dispatched to the Web client, translates the response back to a new value of type a, and merges the result with the old value. Intuitively, one can think of a $Formlens\ a$ as a bidirectional mapping between an a value and a form that contains an a. But unlike the manual approach described above using explicit functions of type $a \rightarrow Formlet\ a$, we can design the formlens abstraction to support composition in a natural way and provide semantic guarantees.

Challenges. Combining formlets and lenses turns out to be nontrivial. If values of type $Formlens\ a$ consume a values, then a must appear in a negative position in their types, so $Formlens$ cannot be a covariant $Functor$ over the category of Haskell types and functions. Conversely, if values of type $Formlens\ a$ produce a values, then a must appear positively in their types, so $Formlens$ cannot be a contravariant functor over this category either. To avoid this difficulty, we shift perspective and consider functors from the following categories to the category $Hask$ of Haskell types and functions:

- Bij, the category of Haskell types and *bijective* embedding-projection pairs and
- $Lens$, the category of Haskell types and *bidirectional* transformations [15].

Although other researchers have considered using functors over other categories in Haskell, such as partial isomorphisms for invertible syntax descriptions [33], to the best of our knowledge, we are the first to use functors over lenses for programming.

Another complication is that, even restricting attention to functors on bijections or lenses, the applicative functor interface is too strong. Consider the following interface, which generalizes McBride and Paterson's definition to allow applicative functors from an arbitrary source category c to $Hask$:

$$\begin{aligned} &\mathbf{class}\ GApplicative\ c\ f\ \mathbf{where} \\ &\quad gpure :: a \rightarrow f\ a \\ &\quad gapp\ \ :: f\ (c\ a\ b) \rightarrow f\ a \rightarrow f\ b \end{aligned}$$

To instantiate this interface with $GApplicative\ \ c\ \ Formlens$, where c is Bij or $Lens$, we would need to define functions

$$\begin{aligned} &gpure :: a \rightarrow Formlens\ a \\ &gapp\ \ :: Formlens\ (c\ a\ b) \rightarrow Formlens\ a \rightarrow Formlens\ b \end{aligned}$$

For the latter, we would need to (somehow) combine a form that consumes and produces a $Bij\ a\ b$ or $Lens\ a\ b$ with another form that produces and consumes an a to obtain a form that produces and consumes a b. This does not seem to be possible to achieve in a natural way. Unlike formlets, which are only required to produce values, formlenses must produce and consume values, which leads to problems instantiating $gapp$, which has a fundamentally asymmetric type.

Fortunately we can avoid this problem by exploiting the connection between $Applicative$ functors and $Monoidal$ functors noted by McBride and Paterson. They showed that for Haskell $Functor$s, $Applicative$ functors are inter-definable with $Monoidal$ functors. It turns out that this extra structure is not required for $Formlet$s or $Formlens$es. By adapting the basic ideas of $Formlet$s to a different (weaker) mathematical structure, we are able to employ $Formlet$s in a broader range of settings—specifically we can compose formlets with bijections and lenses while retaining the ability to compose formlets formerly offered by the $Applicative$ interface and now provided by the $Monoidal$ interface instead.

Contributions. This paper makes the following contributions:

- We present a new foundation for formlets based on monoidal functors. This foundation makes it possible to endow formlets with a bidirectional semantics and provides a convenient abstraction for presenting data on the Web.
- We explore the connection between linear syntax and bidirectional transformations by developing a translation from a linear pattern language into our formlens combinators. This extension makes it possible to define formlenses using a convenient syntax (subject to a linearity constraint). We have implemented this translation using GHC's quasi-quoting mechanism and Template Haskell.
- We develop infrastructure for building formlenses over arbitrary algebraic datatypes, using ideas from datatype-generic programming.

The rest of the paper is structured as follows: Section 2 reviews (standard) definitions of monoidal categories and functors, phrased in terms of Haskell type classes. Section 3 shows how we can reconstruct classic formlets in terms of monoidal functors, and how the resulting formlets can be *lifted* to functors over bijections or lenses, resulting in formlenses. Section 4 adapts the syntactic sugar of classic formlets to formlenses, allowing for either bijective or bidirectional templates. Section 5 presents infrastructure for defining formlenses over algebraic datatypes. Section 6 discusses related work, and Section 7 concludes.

2. Monoidal categories and functors

The key ingredient in our approach is to use monoidal rather than applicative functors to define formlenses. This section reviews the (mostly standard) definitions for bijections, lenses, categories, and monoidal functors. It can be skimmed on a first reading and referred back to as needed.

Bijections. A bijection is a one-to-one and onto mapping between two sets. We represent bijections using the following datatype:

$$\mathbf{data}\ Bij\ a\ b = Bij\ \{fwd :: a \to b, bwd :: b \to a\}$$

Every bijection is expected to satisfy the following conditions, which state that fwd and bwd are inverses:

$$bwd\ bij \circ fwd\ bij = id = fwd\ bij \circ bwd\ bij$$

Bijections admit many useful combinators such as the following:

$$\begin{aligned}
&idB && :: Bij\ a\ a \\
&idB\ x && = x \\
&invB && :: Bij\ a\ b \to Bij\ b\ a \\
&invB\ bij && = Bij\ (bwd\ bij)\ (fwd\ bij) \\
&swapB && :: Bij\ (a,b)\ (b,a) \\
&swapB && = Bij\ (\lambda(a,b) \to (b,a))\ (\lambda(b,a) \to (a,b))
\end{aligned}$$

Lenses. Lenses generalize bijections by allowing the transformation in the forward direction to be non-injective. We represent lenses in Haskell using the following type:

$$\mathbf{data}\ Lens\ a\ b = Lens\ \{get :: a \to b, put :: a \to b \to a\}$$

Note that the type of put, the backward transformation, is asymmetric. It takes the original a and a new b as arguments and produces an updated a.

Every lenses is expected to satisfy the following laws [15]:

$$\begin{aligned}
&put\ a\ (get\ a) = a && \text{-- GetPut} \\
&get\ (put\ a\ b) = b && \text{-- PutGet}
\end{aligned}$$

Intuitively, these laws say that data is preserved on round-trips.

Lenses admit the following combinators (among many others):

$$\begin{aligned}
&idL && :: Lens\ a\ a \\
&idL && = Lens\ id\ (const\ id) \\
&constL && :: b \to Lens\ a\ b \\
&constL\ b && = Lens\ (const\ b)\ (\lambda a\ _ \to a) \\
&fstL && :: Lens\ (a,b)\ a \\
&fstL && = Lens\ (\lambda(a,b) \to a)\ (\lambda(_,b)\ a \to (a,b)) \\
&(\cdot \times_L \cdot) && :: Lens\ a_1\ b_1 \to Lens\ a_2\ b_2 \to Lens\ (a_1,b_1)\ (a_2,b_2) \\
&l_1 \times_L l_2 && = Lens\ (\lambda(a_1,a_2) \to (get\ l_1\ a_1, get\ l_2\ a_2)) \\
& && \qquad (\lambda(b_1,b_2)\ (a_1,a_2) \to \\
& && \qquad\quad (put\ l_1\ b_1\ a_1, put\ l_2\ b_2\ a_2))
\end{aligned}$$

Every bijection can be converted into a lens using a put function that ignores its second argument:

$$\begin{aligned}
&bij2lens && :: Bij\ a\ b \to Lens\ a\ b \\
&bij2lens\ b && = Lens\ (fwd\ b)\ (const\ (bwd\ b))
\end{aligned}$$

Categories. A category is a collection of objects and arrows such that every object has an identity arrow and arrows compose associatively.

$$\begin{aligned}
&\mathbf{class}\ Category\ c\ \mathbf{where} \\
&\quad id :: c\ x\ x \\
&\quad (\circ) :: c\ y\ z \to c\ x\ y \to c\ x\ z
\end{aligned}$$

Haskell functions, bijections, and lenses each form a category.

$$\begin{aligned}
&\mathbf{instance}\ Category\ (\to)\ \mathbf{where} \\
&\quad id \quad\ = \lambda x \to x \\
&\quad f \circ g = \lambda x \to f\ (g\ x) \\
&\mathbf{instance}\ Category\ Bij\ \mathbf{where} \\
&\quad id \quad\ = Bij\ id\ id \\
&\quad f \circ g = Bij\ (fwd\ f \circ fwd\ g)\ (bwd\ g \circ bwd\ f) \\
&\mathbf{instance}\ Category\ Lens\ \mathbf{where} \\
&\quad id \quad\ = Lens\ id\ (const\ \$\ id) \\
&\quad l \circ m = Lens\ (get\ l \circ get\ m) \\
&\qquad\qquad\qquad (\lambda a\ c \to put\ m\ a\ (put\ l\ (get\ m\ a)\ c))
\end{aligned}$$

An isomorphism in a category c is a morphism $f :: c\ a\ b$ with an "inverse" $g :: c\ b\ a$ satisfying $f \circ g = id = g \circ f$. In a computational setting, it is convenient to represent isomorphisms explicitly.

$$\mathbf{data}\ Iso\ c\ a\ b = Iso\ \{fwdI :: c\ a\ b, bwdI :: c\ b\ a\}$$

Isomorphisms are expected to satisfy the following condition:

$$fwdI\ iso \circ bwdI\ iso = id = bwdI\ iso \circ fwdI\ iso$$

Note that *Bij* is just *Iso* $(\rightarrow)$. However, we will keep the notation separate to avoid confusion.

Monoidal categories. A *monoidal category* is a category with additional structure, namely a unit and product on objects in the category. For simplicity, we will use the following definition of monoidal categories, which is specialized to Haskell's unit $()$ and pairing types $(,)$ and also includes a pairing operation on c-arrows $(\cdot \times \cdot)$ as well as c-isomorphisms relating unit and pairing:

```
class Category c ⇒ MonoidalCategory c where
  (· × ·) :: c a1 b1 → c a2 b2 → c (a1, a2) (b1, b2)
  munitl  :: Iso c (a, ()) a
  munitr  :: Iso c ((), a) a
  massoc  :: Iso c (a, (b, d)) ((a, b), d)
```

Monoidal categories are expected to satisfy a number of additional laws, which essentially state that $()$ is a unit with respect to $(,)$ and "all diagrams involving the above operations commute" [23]. We will also omit discussion of the relevant laws of monoidal categories; the standard laws hold for all of the monoidal categories we will consider in this paper.

Haskell types and functions form a monoidal category $(\rightarrow)$, with the following operations:

```
instance MonoidalCategory (→) where
  f × g  = λ(a, b) → (f a, g b)
  munitl = Iso (λ(a, ()) → a) (λa → (a, ()))
  munitr = Iso (λ((), a) → a) (λa → ((), a))
  massoc = Iso (λ(a, (b, d)) → ((a, b), d))
               (λ((a, b), d) → (a, (b, d)))
```

In fact, any category with finite products is monoidal, but there are many monoidal categories whose monoidal product does not form a full Cartesian product. This is the case, for example, for bijections, since the *fst* and *snd* mappings are not bijections; models of linear type theory [2] provide more examples.

Two examples of monoidal categories that are relevant for our purposes are *Bij* and *Lens*. For *Bij*, we first show how to lift isomorphisms on *Hask* to *Bij* (there is some redundancy here, which we tolerate for the sake of uniformity):

```
iso2bij              :: Iso (→) a b → Iso Bij a b
iso2bij (Iso to fro) = Iso (Bij to fro) (Bij fro to)
instance MonoidalCategory Bij where
  f × g  = Bij (λ(a, b) → (fwd f a, fwd g b))
               (λ(fa, gb) → (bwd f fa, bwd g gb))
  munitl = iso2bij munitl
  munitr = iso2bij munitr
  massoc = iso2bij massoc
```

For *Lens*, we first lift the coercion *bij2lens* from bijections to lenses to act on isomorphisms:

```
iso2lens              :: Iso Bij a b → Iso Lens a b
iso2lens (Iso to fro) = Iso (bij2lens to) (bij2lens fro)

instance MonoidalCategory Lens where
  l1 × l2 = l1 ×_L l2
  munitl = iso2lens munitl
  munitr = iso2lens munitr
  massoc = iso2lens massoc
```

Dual categories. Every category has a dual, obtained by reversing arrows:

```
newtype c^op a b = Co {unCo :: c b a}
instance Category c ⇒ Category c^op where
  id              = Co id
  (Co f) ∘ (Co g) = Co (g ∘ f)
```

The dual of any monoidal category is also monoidal:

```
iso2dual              :: Iso c a b → Iso c^op a b
iso2dual (Iso to fro) = Iso (Co fro) (Co to)
instance MonoidalCategory c ⇒
         MonoidalCategory c^op where
  (Co l1) × (Co l2) = Co (l1 × l2)
  munitl            = iso2dual munitl
  munitr            = iso2dual munitr
  massoc            = iso2dual massoc
```

In particular, $Lens^{\mathrm{op}}$ is monoidal. This fact will be useful later since *Formlens a* is a contravariant functor from *Lens* to *Hask*.

Functors. The built-in Haskell *Functor* type class models functors from *Hask* to *Hask*. For our purposes, we will need to generalize its definition slightly[2] to consider functors from other categories (such as *Bij* and *Lens*) to *Hask*. Accordingly, we introduce the following type classes:

```
class Category c ⇒ GFunctor c f where
  gmap :: c a b → f a → f b
```

Again, since we are interested only in *Hask*-valued functors rather than defining a type class for functors between arbitrary categories, we define a specific type class for functors from an arbitrary category c to *Hask*.

Monoidal functors. Next, we consider monoidal functors:[3]

```
class Monoidal f where
  unit :: f ()
  (⋆)  :: f a → f b → f (a, b)
```

Note that we do not explicitly identify the domain of f in the type class *Monoidal f*; this is not necessary (and leads to typechecking complications due to the unconstrained type variable) since the signature of the operations of a monoidal functor depends only on the codomain category (which for us is always *Hask*). However, in stating the laws for monoidal functors, we will implicitly assume that the domain is a monoidal category—in particular, that $\cdot \times \cdot$, *munitl*, *munitr*, and *massoc* are defined.

McBride and Paterson use the following laws for monoidal functors,

$$
\begin{array}{rcl}
\mathit{fmap}\ (f \times g)\ (u \star v) & = & \mathit{fmap}\ f\ u \star \mathit{fmap}\ g\ u \\
\mathit{fmap}\ \mathit{fst}\ (u \star \mathit{unit}) & = & u \\
\mathit{fmap}\ \mathit{snd}\ (\mathit{unit} \star u) & = & u \\
\mathit{fmap}\ \mathit{assoc}\ (u \star (v \star w)) & = & (u \star v) \star w
\end{array}
$$

which implicitly assume we are working with endofunctors on *Hask*. In addition, they use the operations $\mathit{fst} :: (a, b) \rightarrow a$ and $\mathit{snd} :: (a, b) \rightarrow b$ which are not available in all monoidal categories (for example, there is no bijection $\mathit{Bij}\ (a, b)\ a$, and while there is a lens $\mathit{Lens}\ (a, b)\ a$, it is not an isomorphism). Since we are only interested in functors whose range is *Hask*, a minor variant of these laws (adapted from MacLane [23]) suffices:

---

[2] Others have proposed much more general libraries for categorical concepts in Haskell [21, 38]; we believe our approach could be framed using such a library, but prefer to keep the focus on the needed concepts to retain accessibility to readers not already familiar with these libraries. This is also an appropriate place to mention that correct use of categorical concepts in Haskell requires some additional side-conditions such as avoidance of nontermination; we treat this issue informally.

[3] Some authors, such as McBride and Paterson, call these *lax monoidal functors* to distinguish them from other kinds of monoidal functors, but these distinctions are unimportant in this paper so we just say *monoidal*.

(M1) $\quad gmap\ (f \times g)\ (u \star v) = fmap\ f\ \ u \star fmap\ g\ u$
(M2) $\quad gmap\ munitl\ (u \star unit) = u$
(M3) $\quad gmap\ munitr\ (unit \star u) = u$
(M4) $\quad gmap\ massoc\ (u \star (v \star w)) = (u \star v) \star w$

Composition. Monoidal functors, like applicative functors, can be composed (unlike some other well-known abstractions such as monads). To realize this in Haskell, we introduce a type for functor compositions and type class instances for composing ordinary and general functors:

$$
\begin{array}{l}
\mathbf{data}\ (f \circ g)\ a = Comp\ \{ deComp :: f\ (g\ a) \} \\
\mathbf{instance}\ (Functor\ f, Functor\ g) \Rightarrow Functor\ (f \circ g)\ \mathbf{where} \\
\quad fmap\ h\ (Comp\ a) = Comp\ (fmap\ (fmap\ h)\ a) \\
\mathbf{instance}\ (Functor\ f, Monoidal\ f, Monoidal\ g) \Rightarrow \\
\qquad\qquad Monoidal\ (f \circ g)\ \mathbf{where} \\
\quad unit \qquad\qquad\qquad\ \ = Comp\ (fmap\ (const\ unit)\ unit) \\
\quad (Comp\ u) \star (Comp\ v) = Comp\ (fmap\ (uncurry\ (\star))\ (u \star v))
\end{array}
$$

Applicative functors. As shown by McBride and Paterson, in the context of $Hask$ functors, every $Applicative$ functor induces a $Monoidal$ functor and vice versa, and the translations back and forth are inverses.

$$
\begin{array}{l}
\mathbf{instance}\ (Monoidal\ f, Functor\ f) \Rightarrow Applicative\ f\ \mathbf{where} \\
\quad pure\ a \quad\ = fmap\ (\backslash\_ \rightarrow a)\ unit \\
\quad mf \otimes mx = fmap\ (\lambda(f, x) \rightarrow f\ x)\ (mf \star mx)
\end{array}
$$

Lemma 1 ([26])**.** *For any functor $f : Hask \rightarrow Hask$, there exist operations $(pure, \otimes)$ making f an applicative functor if and only if there exist operations $(unit, \star)$ making f a monoidal functor.*

This means that existing $Applicative$ functors can be viewed as $Monoidal$ and vice versa. However, this argument implicitly uses the fact that $Hask$ has rich structure, including so-called *strength*,

$$
\begin{array}{l}
strength \qquad\quad\ :: Functor\ f \Rightarrow (a, f\ b) \Rightarrow f\ (a, b) \\
strength\ (a, fb) = fmap\ (\lambda b \rightarrow (a, b))\ fb
\end{array}
$$

which is derived from the fact that *fmap* can be used on arbitrary higher-order functions. This is not possible for functors f from an arbitrary category c to $Hask$.

The traditional $Formlet$ type based on applicative functors can equally well be given using monoidal functors and deriving applicative structure, or vice versa. In fact, the monoidal functor laws are often significantly simpler to verify (as also noted by Paterson [30]). In the next section, we show how to construct formlets based on monoidal functors directly, without an unnecessary detour through applicative functors.

3. Formlenses

This section defines formlenses, the main abstraction presented in this paper. Our development proceeds in two phases. First, we show how to construct the classic formlet abstraction

$$\mathbf{type}\ Formlet\ a = Int \rightarrow (Html, Env \rightarrow a, Int)$$

as the composition of three smaller abstractions—one for generating names, one for accumulating HTML, and one for collecting values out of form responses—following the approach pioneered by Cooper et al. [10]. We also show that each of these smaller abstractions can be endowed with monoidal structure. Defining formlets in a modular fashion, as opposed to simply defining them to be the type stated above has several advantages. One is pedagogical—it explains the role of each smaller abstraction, and it demonstrates that we have not lost essential structure by shifting from applicative to monoidal functors. Another is that it makes it possible to adjust the behavior of formlets in a modular fashion, by composing other types corresponding to additional processing phases. We illustrate this latter point by composing a validation phase onto the bare definition, effectively changing the type of the collector component to $Env \rightarrow Maybe\ a$. We then show how to define new validating formlet combinators that check that the data submitted by the user satisfies certain well-formedness conditions.

Second, we define the formlens abstraction. The type of formlenses,

$$\mathbf{type}\ Formlens\ a = a \rightarrow Int \rightarrow (Html, Env \rightarrow a, Int)$$

closely resembles the type of formlets, except that it is parameterized on an additional argument of type a which intuitively represents the initial value stored on the server. Adding this extra parameter appears simple but creates numerous complications. In particular, a appears both covariantly and contravariantly in the type, and this is the main reason we are forced to sacrifice the richer applicative interface in favor of monoidal functors.

We obtain the formlens type using a higher-order type operator $Lift$ that builds a function type with an extra argument, as in the type of $Formlens$. We show that this type operator preserves monoidal structure and extends to (generalized) functors from Bij to $Hask$ and $Lens$ to $Hask$. Taken together, these results provide sufficient structure to support using bijections and lenses to define forms. We illustrate the use of formlenses for implementing bidirectional transformations through examples, and we discuss the expected semantic properties—*i.e.*, that the a value being transformed by the formlens is preserved on round-trips.

3.1 Classic Formlets

We begin by showing how to define classic formlets as the composition of three simpler functors.

Namers. The functor $Namer\ t$ generates names, which are used to identify elements of the form.

$$\mathbf{newtype}\ Namer\ t\ a = Namer\ \{ runNamer :: t \rightarrow (a, t) \}$$

Note that $Namer\ t$ holds the type t abstract. We will typically use Ints. It is straightforward to show that $Namer\ t$ is a functor

$$
\begin{array}{l}
\mathbf{instance}\ Functor\ (Namer\ t)\ \mathbf{where} \\
\quad fmap\ f\ (Namer\ n) = Namer\ (first\ f \circ n)
\end{array}
$$

and also has monoidal structure:

$$
\begin{array}{l}
\mathbf{instance}\ Monoidal\ (Namer\ t)\ \mathbf{where} \\
\quad unit = Namer\ (\lambda t \rightarrow ((), t)) \\
\quad (Namer\ n_1) \star (Namer\ n_2) = \\
\qquad Namer\ (\lambda t \rightarrow \mathbf{let}\ (a, t') = n_1\ t\ \mathbf{in} \\
\qquad\qquad\qquad\quad \mathbf{let}\ (b, t'') = n_2\ t'\ \mathbf{in} \\
\qquad\qquad\qquad\quad ((a, b), t''))
\end{array}
$$

The following lemma records the fact that $Namer$ is a valid instance of $Monoidal$.

Lemma 2. *$Namer\ t$ is a monoidal functor for any type t.*

To prove this lemma, we can simply check that properties M1 to M4 all hold. To save space, in the rest of this paper, we will usually not state explicit lemmas for each instance of $Monoidal$. However, we still expect these properties to hold.

Accumulators. The $Acc\ m$ functor accumulates the HTML generated by formlets.

$$\mathbf{data}\ Acc\ m\ a = Acc\ m\ a$$

Acc is parameterized on a monoid m. Monoids are defined in Haskell using the following (standard) type class,[4]

[4] Note that the *monoid* type class and *monoidal functors* are different!

```
class Monoid a where
  mempty :: a
  mappend :: a → a → a
```

where the *mempty* element is an identity for the *mappend* operation, which is associative. We will typically take m to be the monoid of *Html* documents:

```
instance Monoid Html where
  mempty = noHtml
  mappend = (+++)
```

The functor and monoidal functor structures on *Acc* m can be defined as follows:

```
instance Functor (Acc m) where
  fmap f (Acc m a) = Acc m (f a)
instance Monoid m ⇒ Monoidal (Acc m) where
  unit = Acc mempty ()
  (Acc m₁ a) ⋆ (Acc m₂ b) = Acc (m₁ `mappend` m₂) (a, b)
```

Assuming m is a *Monoid*, it is straightforward to verify that *Acc* m is monoidal.

Collectors. The *Collect* e functor reconstructs the value encoded in a form response.

```
data Collect e a = Collect (e → a)
```

Values of type *Collect* e a wrap a function $e \to a$ that builds an a from an environment e. The following definitions give the functor and monoidal functor structure on collectors:

```
instance Functor (Collect e) where
  fmap f (Collect g) = Collect (f ∘ g)
instance Monoidal (Collect e) where
  unit                         = Collect (λx → ())
  (Collect c₁) ⋆ (Collect c₂) = Collect (λe → (c₁ e, c₂ e))
```

Classic Formlets. Using the functors just defined, the type of classic formlets can be obtained using composition,

```
newtype Formlet a =
  Formlet { unFormlet ::
    (Namer Int) ∘ ((Acc Html) ∘ (Collect Env)) a }
```

with functor and monoidal structure lifted to *Formlet* in the obvious way:

```
instance Functor Formlet where
  fmap f = Formlet ∘ fmap f ∘ unFormlet

instance Monoidal Formlet where
  unit      = Formlet unit
  fl₁ ⋆ fl₂ = Formlet (unFormlet fl₁ ⋆ unFormlet fl₂)
```

Note that this type is isomorphic to the type for classic formlets stated at the start of this section:

```
type Formlet a = Int → (Html, Env → a, Int)
```

In general, we can either define formlets using this "direct" type, or using the type defined as the composition of the *Name*, *Acc*, and *Collect* functors. We will often use such "direct" types for simplicity, as it allows us to avoid explicitly introducing and eliminating the *Formlet* and *Comp* constructors.

Validation One of the benefits of defining classic formlets compositionally is that it allows us to modularly adjust their behavior by adding additional processing phases. To illustrate, suppose that we want to define a variant of formlets in which collectors may fail if the strings contained in the form response are ill-formed. A simple way to achieve this is to have the collector inject its result into the *Maybe* type.

First, we check that *Maybe* admits functor and monoidal functor structure:

```
instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just a) = Just (f a)
instance Monoidal Maybe where
  unit                 = Just ()
  (Just x) ⋆ (Just y) = Just (x, y)
  _ ⋆ _                = Nothing
```

Then we build "validating" formlets by composing the *Maybe* functor with the collector component,

```
newtype ValFormlet a =
  ValFormlet { unValFormlet ::
    (Namer Int) ∘ ((Acc Html) ∘ ((Collect Env) ∘ Maybe)) a }
```

which is isomorphic to the following type:

```
type ValFormlet a = Int → (Html, Env → Maybe a, Int)
```

Using this type, we can define combinators that fail when the strings in the form data are not well formed. For example, the following combinator returns *Nothing* if the input is not a string encoding a positive integer.

```
inputNat :: ValFormlet Int
inputNat i = let n = show i in
  (input ! [name n],
    (λe → case reads (fromJust (lookup n e)) of
      [(v, "")] | v > 0 → Just v
      _                 → Nothing),
    i + 1)
```

Note that we use the "direct" type for validating formlets for simplicity. Other validating combinators can be defined similarly.

3.2 Formlenses

Now we present the main result in this section: the definition of formlenses themselves. Recall that our goal is to bundle the code that builds a form together with the code that processes responses. That is, we would like a type that behaves like $a \to$ *Formlet* a. Such a function would be applied to the initial values of the form, so that if the user immediately clicks "submit" then the original a value is reconstructed, and if they modify the values embedded in the form, then the a value changes accordingly. As described in the introduction, there is a major problem with this idea: the type $a \to$ *Formlet* a cannot be made into an ordinary functor because a appears both positively and negatively in the type. Nevertheless, we can define a natural lifting construction that maps monoidal functors to monoidal *Lens* $\to$ *Hask* functors, and hence also *Bij* $\to$ *Hask*, as a special case.

Lifting. Given a type operator f, the type operator *Lift* f maps each type a to the type $a \to f\ a$.

```
data Lift f a = Lift (a → f a)
```

In addition, *Lift* f preserves monoidal structure.

```
instance Monoidal f ⇒ Monoidal (Lift f) where
  unit                 = Lift (λ() → unit)
  (Lift f) ⋆ (Lift g) = Lift (λ(a, b) → f a ⋆ g b)
```

Note that due to the contravariance of *Lift* f, in general *Lift* f may not be an instance of *Functor*. Nevertheless, provided that f is a

Functor, we do have that *Lift f* is a covariant *GFunctor* on *Bij* and a contravariant *GFunctor* on *Lens*.

```
instance Functor f ⇒ GFunctor Bij (Lift f) where
  gmap bij (Lift g) = Lift (fmap (fwd bij) ∘ g ∘ bwd bij)
instance Functor f ⇒ GFunctor Lens^op (Lift f) where
  gmap (Co l) (Lift g) =
    Lift (λa → fmap (put l a) (g (get l a)))
```

In categorical terms, we say that *Lift f* extends to functors from *Bij* to *Hask* and from $Lens^{op}$ to *Hask*. In plain terms, these *GFunctor* instances define functions that can be used to map a bijection or lens over the *Formlens* type:

```
gmap :: Bij a b → Formlens a → Formlens b
gmap :: Lens b a → Formlens a → Formlens b
```

The fact that these instances are valid is be captured in the following theorems.

Theorem 3. *If f is a (monoidal) functor, then Lift f is a (monoidal) GFunctor Lens, and hence Lift f is also a (monoidal) GFunctor Bij.*

The proof of this result is given in Appendix B.

Formlenses. To build the *Formlens* type operator, we lift *Formlet*, yielding a type that is isomorphic to the one stated previously:

```
newtype Formlens a = Formlens (Lift Formlet a)
```

We lift the monoidal functor and generalized functor structures to *Formlens*es in the obvious way:

```
instance Monoidal Formlens where
  unit                            = Formlens (unit)
  (Formlens f) ⋆ (Formlens g) = Formlens (f ⋆ g)
instance GFunctor Bij Formlens where
  gmap b (Formlens fl) = Formlens (gmap b fl)
instance GFunctor Lens^op Formlens where
  gmap l (Formlens fl) = Formlens (gmap l fl)
```

Theorem 4. *Formlens is a monoidal GFunctor Lens, and hence Formlens is also a monoidal GFunctor Bij.*

Example. To get a taste for how formlenses work, let us build a bidirectional version of the *dateForm* Formlet from the introduction. Recall the definition of *dateForm* as a classic formlet:

```
dateForm :: Formlet Date
dateForm = pure (λ_ y _ _ m _ _ d _ → Date y m d)
  ⊗ text "Month: " ⊗ inputInt ⊗ html br
  ⊗ text "Year: "  ⊗ inputInt ⊗ html br
  ⊗ text "Day: "   ⊗ inputInt ⊗ html br
```

As a first step, let us define formlens versions of the *html*, *text*, and *inputInt* combinators.

```
htmlL :: Html → Formlens ()
htmlL h v i = (h, const (), i)

textL :: String → Formlens ()
textL = htmlL ∘ stringToHtml

inputIntL :: Formlens Int
inputIntL v i = let n = show i in
                (input ! [name n, value (show v)],
                 λe → read (fromJust (lookup n e)),
                 i + 1)
```

Note that each of the formlens combinators takes the initial value v for the form as a parameter.

Next, it will be useful to define several helper operators, $(\langle\star)$ and $(\star\rangle)$. These operators behave mostly like $(\star)$ but they combine a *Formlens a* and *Formlens* () into a *Formlens a*, rather than a *Formlens* $(a, ())$ and *Formlens* $((), a)$ respectively.

```
(⟨⋆) :: (Monoidal f, GFunctor Bij f)
       ⇒ f a → f () → f a
f ⟨⋆ g = gmap (munitl :: Iso Bij (a, ()) a)
              (f ⋆ g)
```

The definition of the $(\star\rangle)$ is symmetric, but eliminates the () value using *munitr* instead of *munitl*.

Third, let us define a bijection on *Date*s.

```
dateB :: Bij ((Int, Int), Int) Date
dateB = Bij (λ((y, m), d) → Date y m d)
            (λ(Date y m d) → ((y, m), d))
```

Putting all these pieces together, we can build a formlens for *Date*s as follows:

```
dateFormlens :: Formlens Date
dateFormlens =
  gmap dateB
    (textL "Year: "  ⋆⟩ inputIntL ⟨⋆ htmlL br ⟨⋆
     textL "Month: " ⋆ inputIntL ⟨⋆ htmlL br ⟨⋆
     textL "Day: "   ⋆ inputIntL ⟨⋆ htmlL br)
```

Compared to the formlet *dateForm*, we have replaced $(\otimes)$ with $(\star)$, $(\langle\star)$, and $(\star\rangle)$ as appropriate, and applied *gmap dateB* at the top-level. Overall, *dateFormlens* maps bidirectionally between a *Date* value and a form that encodes a date, as desired.

Optional formlenses. The type *Formlens a* can sometimes be awkward to use, because it requires us to *always* construct an a before building a form for it. We can define a variant of formlenses that support both *creating* and *editing* data using the same code. The idea is to build a type similar to *Maybe a* → *Formlet a*, so that we can either build a form that creates an a from *Nothing*, or if we already have an a, build a form that allows editing that a from *Just a*.

A simple variant of *Lift*, called *MLift*, does precisely this.

```
data MLift f a = MLift (Maybe a → f a)
```

Like *Lift*, the *MLift* operator lifts functors over *Bij*:

```
instance Functor f ⇒ GFunctor Bij (LiftM f) where
  gmap f (MLift g) =
    MLift (fmap (fwd f) ∘ g ∘ fmap (bwd f))
```

with monoidal structure given by:

```
instance (Monoidal f) ⇒ Monoidal (MLift f) where
  unit = MLift (\_ → unit)
  (MLift f) ⋆ (MLift g) =
    MLift (λma →
      case ma
        of Just (a, b) → f (Just a) ⋆ g (Just b)
           Nothing → f Nothing ⋆ g Nothing)
```

But unlike *Lift*, even if *f* is a functor *MLift g* is not a contravariant functor on lenses, since in cases where the input is *Nothing* we do not have an a-value to provide to *put*:

```
instance Functor f ⇒ GFunctor Lens^op (MLift f) where
  gmap (Co l) (MLift (g)) =
    MLift (λma →
      case ma of
        Just a → fmap (put l a) (g (Just (get l a)))
        Nothing → fmap (put l ( ??? )) (g Nothing))
```

However, if we adjust the definition of a lens slightly so that the put direction can proceed if we just have a b value and no a,

```
data MLens a b = MLens { mget :: a → b,
                         mput :: Maybe a → b → a }
```

then we can make $MLens$ into a monoidal category, analogous to $Lens$es (the details are in Appendix **??**). The $MLens$ type is similar to classic lenses, which have a "create" function in addition to "get" and "put" functions [15].

With this generalization, we can then lift functors f to contravariant $MLens$ functors in the obvious way:

```
instance Functor g ⇒ GFunctor MLens^op (MLift g) where
  gmap (Co l) (MLift (g)) =
    MLift (λma →
      case ma of
        Just a → fmap (mput l ma) (g (Just (mget l a)))
          Nothing → fmap (mput l Nothing) (g Nothing))
```

Moreover, these functors also have monoidal structure.

Theorem 5. *If f is a (monoidal) functor then $MLift\ \ f$ is a (monoidal) $GFunctor\ \ MLens$, and hence $MLift\ \ f$ is also a (monoidal) $GFunctor\ Bij$.*

Finally we can define $MFormlens\ a$ as follows:

```
newtype MFormlens a = MFormlens (MLift Formlet a)
```

We will use $MFormlens$es to define a type-theoretic sum operator in Section 5.

3.3 Semantic Properties

Classic lenses satisfy natural well-behavedness conditions such as the GETPUT and PUTGET laws. A natural question to ask is: are there analogous laws for formlenses, and are they preserved by operations such as $(\star)$ and $gmap$?

Let $extract$ be a function from $Html$ to Env that crawls over an HTML document and extracts the association list containing the names and values of all form fields. We say that a $Formlens$ value fl is well behaved if it satisfies the following law (which is analogous to GETPUT) for all values x and namesources t:

$$\frac{(h, c, n') = fl\ x\ n \qquad extract\ h \subseteq e}{collect\ e = x}\ (\textsc{CollectExtract})$$

We believe that our formlens combinators satisfy COLLECTEXTRACT (or preserve it assuming that their arguments satisfy it), and that if fl satisfies COLLECTEXTRACT then so do $(gmap\ \ bij\ \ fl)$ and $(gmap\ \ l\ \ fl)$, where bij is a bijection and l a lens of appropriate type. We plan to prove these properties formally in the near future.

4. High-level Syntax

Programming with raw formlens combinators can be difficult because the structure of the types produced (and consumed) by the Formlens closely matches the structure of the combinator program. For example, if f is a $Formlens\ Int$ and g is a $Formlens\ ()$ then $f \star g$ is a $Formlens\ (Int, ())$. If the programmer wants to obtain a $Formlens\ Int$ instead, they must $gmap$ the bijection $munitl$ on the result to eliminate the spurious $()$. Of course, the derived $\langle\star$ operator does this, but having to remember when to use $\star$ versus $\langle\star$ is inconvenient.

This section presents a better alternative. We define a syntax for describing formlenses based on pattern matching, and we give a translation from this high-level syntax into our low-level formlens combinators. In our implementation, we use quasi-quotation [24] to represent this syntax and Template Haskell [35] to implement the translation.

Meta variables

e expression	s string	t tag
f formlet expression	p pattern	ats attribute list

Quasiquotes

```
c ::= [formc | n1 ... nk yields e |]      classic formlet
b ::= [formb | p ↔ n1 ... nk |]           bijective formlens
l ::= [forml | p ↔ n1 ... nk |]           bidirectional formlens

n ::= <t ats>n1 ... nk</t> | s            node
   |  {e} | {f → p}
```

Figure 1. Syntax

The syntax is based on pairs of patterns over a shared set of variables. The pattern on the left is used to match values, while the pattern on the right is used to match snippets of HTML. Hence, when read from left to right, a program written in this syntax denotes a transformation from values into HTML; when read from right to left it denotes a transformation from form responses back into values. Overall, this syntax provides a much more convenient way of describing formlenses compared to manually constructing combinator programs by hand, and applying bijections (or lenses) to massage the data into the desired format.

Classic formlets. To define a classic formlet, we use the syntax [**formc** | $body$ **yields** e |]. The $body$ consists of a sequence of nodes[5] where a node is either an element <t ats>$n_1\ \ldots\ n_k$</t>, a text node s, a spliced HTML expression $\{e\}$; or a nested formlet binding $\{f \to p\}$. A full description of the syntax of our patterns is given in Figure 1.

Cooper et al. [10] defined a desugaring from such quasiquoted formlet programs into combinators using applicative functors. We will use an analogous translation, adapted to use monoidal functors instead. The translation of the body using $(-)^\circ$ first goes by structural recursion, producing formlet combinators as a result. Sequences of nodes are then combined using the $(\star)$ operator and $unit$ handles the empty sequence. On the other side, we extract patterns from the form using $(-)^\dagger$, again following a straightforward structural recursion. The functor map operation $fmap$ is then used to combine the results of the sub-formlets bound in the body through the extracted patterns. Figure 4 gives the formal definition of the translation.

Using this syntactic sugar we can rewrite $dateForm$ as:

```
dateFormC :: Formlet Date
dateFormC =
  [formc | Year :  {inputInt → y}<br/>
           Month : {inputInt → m}<br/>
           Day :   {inputInt → d}<br/>
   yields Date y m d |]
```

The translation of this program is the code for $dateForm$ given in the introduction.

Bijective formlenses. Bijective Formlenses are defined using the syntax [| **formb** | $p \leftrightarrow n_1\ \ldots\ n_k$ |]. For a bijective formlens, we have to additionally specify how to split up input values among sub-formlenses. Furthermore, to ensure that the overall formlens

[5] In practice, in order to make parsing easier, one could require additional syntax for delimiting the body. For instance, the Links language [9], requires the body to be a single element, and uses a special dummy element < # /> for simulating sequences of nodes.

$$
\begin{aligned}
[\![\mathbf{formc} \mid n_1 \ \ldots \ n_k \ \mathbf{yields} \ e]\!] &= \mathit{fmap}\, f\ (n_1 \ \ldots \ n_k)^\circ \\
&\quad \text{where} \\
&\qquad f = \lambda (n_1 \ \ldots \ n_k)^\dagger \ \rightarrow e
\end{aligned}
$$

$$
\begin{aligned}
s^\circ &= \mathit{text}\ s \\
\{e\}^\circ &= \mathit{html}\ e \\
\{f \rightarrow p\}^\circ &= f \\
\texttt{<}t\ \mathit{ats}\texttt{>}n_1 \ \ldots \ n_k\texttt{</}t\texttt{>}^\circ &= \mathit{tag}\ t\ \mathit{ats}\ (n_1 \ \ldots \ n_k)^\circ \\
(n_1 \ \ldots \ n_k)^\circ &= n_1 \star \ldots \star n_k
\end{aligned}
$$

$$
\begin{aligned}
s^\dagger &= () \\
\{e\}^\dagger &= () \\
\{f \rightarrow p\}^\dagger &= p \\
\texttt{<}t\ \mathit{ats}\texttt{>}n_1 \ \ldots \ n_k\texttt{</}t\texttt{>}^\dagger &= (n_1 \ \ldots \ n_k)^\dagger \\
(n_1 \ \ldots \ n_k)^\dagger &= (n_1^\dagger, \ldots, n_k^\dagger)
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathbf{formb} \mid p \leftrightarrow n_1 \ \ldots \ n_k]\!] = \\
&\quad \mathit{gmap}\ (\mathit{Bij}\ \mathit{get}\ \mathit{put})\ (n_1 \ \ldots \ n_k)^\circ \text{ where} \\
&\qquad \mathit{fwd} = \lambda (n_1 \ \ldots \ n_k)^\dagger \ \rightarrow p \\
&\qquad \mathit{bwd} = \lambda p \rightarrow \ (n_1 \ \ldots \ n_k)^\dagger
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathbf{forml} \mid p \leftrightarrow n_1 \ \ldots \ n_k]\!] = \\
&\quad \mathit{gmap}\ (\mathit{Co}\ (\mathit{Lens}\ \mathit{get}\ \mathit{put}))\ (n_1 \ \ldots \ n_k)^\circ \\
&\quad \text{where } (p_{old}, p_{new}) \ = \ p^\prec \\
&\qquad \mathit{get} \ = \ \lambda p \rightarrow \ (n_1 \ \ldots \ n_k)^\dagger \\
&\qquad \mathit{put} \ = \ \lambda p_{old} \rightarrow \lambda (n_1 \ \ldots \ n_k)^\dagger \ \rightarrow p_{new}
\end{aligned}
$$

$$
\begin{aligned}
\_^\prec &= (x, x), \quad x \text{ fresh} \\
x^\prec &= (\_, x) \\
(p_1, \ldots, p_k)^\prec &= ((p'_1, \ldots, p'_k), (e_1, \ldots, e_k)) \\
&\quad \text{where } (p'_i, e_i) = {p_i}^\prec, \ \ 1 \leqslant i \leqslant k \\
(K\ p)^\prec &= (K\ p', K\ e) \text{ where } (p', e) = p^\prec
\end{aligned}
$$

Figure 2. Translations.

is well behaved, the variables must be used *linearly* among those sub-formlenses.

We adopt a straightforward convention that guarantees linearity. The idea is to adapt the classic formlet syntax so that the **yields** clause is restricted to be a pattern and is used to specify both input and output values. This is more restrictive than classic formlet, which allow **yields** clauses to be arbitrary, but handles the common case of bijective mappings between source values and forms. Instead of $[\![\mathbf{formb} \mid n_1 \ \ldots \ n_k \ \mathbf{yields} \ p]\!]$ we write $[\![\mathbf{formb} \mid p \leftrightarrow n_1 \ \ldots \ n_k]\!]$, to highlight that the interface pattern p defines both the input and output interfaces for the formlens.

The key difference in the translation (see Figure 4) is that the *fmap* becomes *gmap* and is passed a bijection rather than a function[6]. This is required as the domain of $\mathit{GFunctor}\ \mathit{Bij}\ \mathit{Formlens}$ is the category of Haskell types and bijections. We construct the bijection using Template Haskell. Note, however, that the functions *fwd* and *bwd* are only well-typed if the linearity condition is satisfied. Observe also, that p cannot contain any wildcard patterns, as it is used as an expression. This means that the formlet must use all of the input—*i.e.*, we require strict linearity and merely affine patterns do not suffice. It would be tempting to extend interface patterns to support a richer language of bijections through a suitable linear typing discipline, and we intend to explore this in the future. However, patterns do cover an important and common case. Moreover, it is always possible to explicitly use *gmap* outside of the syntactic sugar in order to compose an arbitrary bijection with a formlens.

Using this syntactic sugar, we can rewrite the *dateFormlens* example as follows:

$$
\begin{aligned}
&\mathit{dateFormlensB} :: \mathit{Formlens}\ \mathit{Date} \\
&\mathit{dateFormlensB} = \\
&\quad [\![\mathbf{formb} \mid \mathit{Date}\ y\ m\ d \\
&\qquad\qquad \leftrightarrow \\
&\qquad\qquad \mathit{Year}: \quad \{\mathit{inputIntL} \rightarrow y\}\texttt{
} \\
&\qquad\qquad \mathit{Month}: \{\mathit{inputIntL} \rightarrow m\}\texttt{
} \\
&\qquad\qquad \mathit{Day}: \quad\ \{\mathit{inputIntL} \rightarrow d\}\texttt{
}\ ]\!]
\end{aligned}
$$

This is desugared into essentially the same code as *dateFormlens* in Section 3.2, but without using the $(\langle\star)$ and $(\star\rangle)$ operators. In particular, the outer bijection becomes:

$$
\begin{aligned}
&\mathit{Bij}\ (\lambda (\mathit{Date}\ y\ m\ d) \rightarrow ((), y, (), (), m, (), (), d, ())) \\
&\quad (\lambda ((), y, (), (), m, (), (), d, ()) \rightarrow \mathit{Date}\ y\ m\ d)
\end{aligned}
$$

It would be straightforward to adapt the desugaring transformation to output $\star\rangle$ and $\langle\star$ operators where appropriate. It is not clear whether this would be particularly desirable, but it would make the generated code more readable.

Bidirectional formlenses. Finally, bidirectional formlenses are defined using the syntax $[\![\ \mathbf{forml} \mid p \leftrightarrow n_1 \ \ldots \ n_k\ ]\!]$. Unlike bijective formlenses, patterns in bidirectional formlenses may ignore part of the input. Thus we generalize the syntax to allow wildcards in the pattern.

The key difference from the translation (see Figure 4) for bijective formlenses is the use of the *put* function. As we are now mapping a lens over the formlens, the *put* function must take an extra argument representing the original input value. The pattern p_{old} binds the parts of the original input value that are ignored by the formlens. The pattern p_{new} (which is also an expression as it contains no wildcards) combines the ignored part of the input bound by p_{old} with the output produced by the formlens.

As with the translation for bijective formlenses, if *get* and *put* are well-typed then linearity is guaranteed. Any wildcard patterns that appear in the interface pattern are filled in using the additional argument to *put*.

If no wildcards appear in the pattern, then **forml** desugaring produces essentially the same code as **formb** desugaring—the first argument to *put* is just ignored.

As a simple example, suppose we want a form, based on the *Speaker* example from the introduction, that only allows us to see and edit the name of a speaker, while maintaining the date, then we can write the following code:

$$
\begin{aligned}
&\mathit{speakerFormL} :: \mathit{Formlens}\ \mathit{Speaker} \\
&\mathit{speakerFormL} = [\![\ \mathbf{forml} \mid \mathit{Speaker}\ \mathit{name}\ \_ \\
&\qquad\qquad\qquad \leftrightarrow \mathit{Name}: \{\mathit{inputStringL} \rightarrow \mathit{name}\}\ ]\!]
\end{aligned}
$$

which is desugared into:

$$
\begin{aligned}
&\mathit{speakerFormL} = \\
&\quad \mathit{gmap}\ (\mathit{Lens}\ (\lambda (\mathit{Speaker}\ \mathit{name}\ \_) \rightarrow ((), \mathit{name})) \\
&\qquad\qquad (\lambda (\mathit{Speaker}\ \_\ \mathit{x0})\ ((), \mathit{name}) \rightarrow \mathit{Speaker}\ \mathit{name}\ \mathit{x0})) \\
&\qquad (\mathit{textL}\ \texttt{"Name: "} \star \mathit{inputString})
\end{aligned}
$$

where $\mathit{x0}$ is a fresh variable that tracks the old date value.

Aside. As Template Haskell provides no support for antiquotation in user-defined quasiquoters, we roll our own simple bracket-counting parser to determine the extent of embedded Haskell code and pass the output to Dominic Orchard's syntax-trees package[7].

[6] Of course, we must also adapt $(-)^\circ$ to output textL, htmlL, and tagL in place of text, html, and tag.

[7] `http://hackage.haskell.org/package/syntax-trees`

```
type HNFormlet = (Namer [Int]) ∘ ((Acc Html) ∘ (Collect Env))
type HNFormlens a = MLift HNFormlet a
data Value = Base String | Inl Value | Inr Value
           | Pair Value Value | Con Value
data GenFormlens =
    forall a . (BaseValue a) ⇒ GBase (HNFormlens a)
   | GPair GenFormlens GenFormlens
   | GSum GenFormlens GenFormlens
   | GVar String
   | GRec String GenFormlens
```

Figure 3. Basic structures for algebraic datatypes.

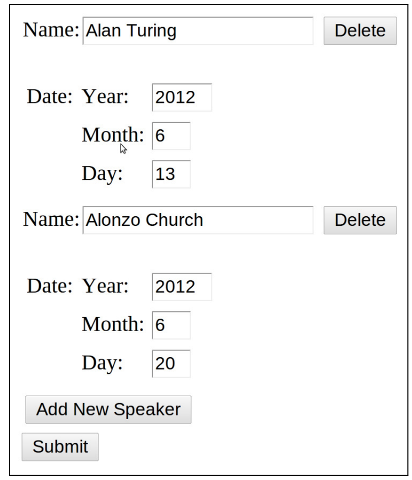

Figure 5. Screenshot of $Speaker$s formlens running in a browser.

5. Formlenses for Algebraic Datatypes

So far, we have seen several ways of constructing formlenses: by defining primitive formlenses; by exploiting the monoidal structure of formlenses, in particular the $(\star)$ operator, to combine formlenses that operate on the components of a product to obtain a formlens that operates on the entire product; and by using $gmap$ to modify the behavior of a formlens using a bijection or a lens. This section presents additional infrastructure that makes it possible to construct formlenses over arbitrary algebraic datatypes. These features make it possible to define formlenses for richer datatypes such as lists and trees. For example, we will be able to lift a $Formlens\ a$ to a $Formlens\ [a]$.

Our approach has two main ingredients. First, borrowing techniques from from datatype-generic programming [17], we represent datatypes as a recursive sum of products, and we define a formlens operator for each type operator. Second, we modify the $Formlens$ type, enriching the namesource to be an $[Int]$ instead of Int. Such namesources can be used to represent hierarchical names, which are needed to handle recursive formlenses.

Basic structures. Figure 5 defines types for hierarchically-named formlenses, algebraic datatypes, and boxed values. $HNFormlet$ is like $Formlet$ but has an $[Int]$ as a namesource. The type $HNFormlens$ is obtained by applying $MLift$ to $HNFormlet$. The type $Value$ represents explicitly boxed values belonging to an algebraic datatype. $Con\ v$ represents a (rolled) value of a recursive type. To inject Haskell base values in and out of $Value$s, we use the $BaseValue$ typeclass (included in the long version of this paper [32]); instances of $BaseValue$ are inter-expressible as $Value$s. The most interesting definition is of the $GenFormlens$ type, which represents the abstract syntax of an algebraic type, with $HNFormlens$es at the leaves. Note that we model recursive types using explicit type variables.

Combinators. To convert a $GenFormlens$ into a formlens, we need to interpret each type operator as an operator on formlenses. We already have a definition of a product operator on formlenses—namely the $(\star)$ operator. Figure 4 defines the operators $(\oplus)$ and $unfold$, which handle sums and recursive types respectively.

At a high level, the sum formlens $(f \oplus g)$ works as follows. It adds a hidden field to the rendered HTML that indicates whether the value encoded in the form is a left or right injection. Otherwise, we only render the sub-formlens that produces a value (defaulting to the left one if neither does). The collector of the sum formlens checks the hidden tag and invokes the collector of the appropriate component. The modified name source is computed by taking the max of the two namesources, since the two components may not use the same number of names.

For recursive formlenses, we add a level of indirection at each unfolding, which can be thought of as simulating a pointer. At each level of recursive unfolding, we add a hidden element to the HTML that points to the first element of the underlying value is added. This enables editing the structures generated by a recursive formlens by modifying pointers. Our hierarchical naming scheme makes it easy to efficiently manipulate such pointers. For example, the parent of a block of HTML elements containing an element `"input_0_3"` is the element `"input_0"`. Such manipulations are necessary to implement operations such as insertions and deletions into a list.

Example. As an example, the following defines a formlens that handle a list of $Speaker$s, as discussed in the introduction:

```
speakerListFormlens :: HNFormlens Value
speakerListFormlens =
  tagL "table" $ tagL "tr" unit ⋆⟩ (getFormlens $
    GRec "x" $
      GSum (GBase $ tagL "tr"
        $ tagL "td" insertButton)
        (GPair (GBase $ tagL "tr"
          $ tagL "td" speakerFormlens
          ⟨⋆ tagL "td" deleteButton)
         (GVar "x")))
```

This code creates a formlens that essentially follows the recursive structure of a list: it uses $GRec$ to handle the recursive type, $GSum$ to handle the sum, $GBase$ for the nil case, and $GPair$ for the cons case, and $speakerFormlens$ for each element. The result of running this code in a web browser can be seen in Figure 5. Note that the form supports modifying $Speaker$s in place, as well as adding and deleting list elements. The complete code needed to run this example can be found in Appendix A.

6. Related work

Formlets. Cooper et al. [10] first proposed expressing formlets as the composition of several primitive applicative functors. There is a variety of earlier work on declarative abstractions for web forms (e.g. [1, 6, 8, 18, 19, 31, 37]). Of particular note is Hanus's WUI (Web User Interface) library [18]. The abstraction used in WUI is essentially the type $a \rightarrow Formlet\ a$ we adopt in this paper. In particular, the WUI library includes a combinator for lifting a bijection to WUIs, similar to our $GFunctor\ Bij\ Formlens$ instance; however, Hanus did not consider $Applicative$ or $Monoidal$ equational laws on WUIs, nor constructing them as functors over bijections or lenses out of simpler components.

```
nextName  l@(hd : tl) = (makeName l, (hd + 1) : tl)
makeName l            = "input_" ++ intercalate "_" (map show (reverse l))
readName  s           = reverse (map read (split "_" (drop 6 s)))

(⊕) :: HNFormlens a → HNFormlens b → HNFormlens (Either a b)
f ⊕ g = λmv l → let (n, l′) = nextName l in
                let (ma, mb, tag, keepA, keepB) = case v of
                  Just (Left a)  → (Just a,  Nothing, "left",  id, const noHtml)
                  Just (Right b) → (Nothing, Just b,  "right", const noHtml, id)
                  Nothing        → (Nothing, Nothing, "left",  id, const noHtml) in
                let (ra, ca, la) = f ma l′ in
                let (rb, cb, lb) = g mb l′ in
                (hidden n tag +++ keepA ra +++ keepB rb,
                λe → case lookup n e of
                       Just "left"  → Left  (ca e)
                       Just "right" → Right (cb e),
                max la lb)

unfold :: HNFormlens a → HNFormlens a
unfold f = λmv l →  let (n, l′) = nextName l in
                    let collector = (λe → let k = readName (fromJust (lookup n e)) in
                                          let (_, c, _) = f a k in c e) in
                    let (r, _, _) = f mv (0 : l) in
                    (hidden n (makeName (0 : l)) +++ r, collector, l′)
```

Figure 4. Sum and recursion combinators.

Formlets were originally implemented as part of the Links web programming language [9]. Subsequently formlet libraries have been implemented for Haskell, F#, Scala, OCaml, Racket, and Javascript. Eidhof's original Haskell library [13] evolved first into digestive functors [12], and most recently the reform package [34]. The latter integrates with various other web programming libraries, and extends formlets with better support for validation, and separating layout from formlet structure. The commercial WebSharper library for F# [3] introduces *flowlets*, which combine formlets with functional reactive programming [14] allowing forms to change dynamically at run-time. However, while flowlets involve both applicative and monadic combinators, this work does not develop the formal semantics or equational laws for flowlets.

Bidirectional transformations. Languages for describing bidirectional transformations have been extensively studied in recent years [4, 5, 15, 16, 25, 27–29, 40]. The original paper on lenses [15] describes work on databases and programming languages; another more recent survey also discusses work from the software engineering literature [11]. The XSugar [7] language defines bidirectional transformations between XML documents and strings. Similarly, the biXid [20] language specifies essentially bijective conversions between pairs of XML documents. Transformations in both XSugar and biXid are specified using pairs of intertwined grammars, which resemble our high-level pattern syntax. The most closely related work we are aware of is by Rendel et al. [33]. They propose using functors over partial isomorphisms to describe invertible syntax descriptions. Our design for formlenses is similar, but also supports using non-bijective bidirectional transformations, a high-level syntax, and full support for algebraic datatypes.

Applicative and monoidal functors. Applicative functors have been used extensively as an alternative to monads for structuring effectful computation. They were used (implicitly) by Swierstra and Duponcheel [36] for parser combinators, and named and recognized as a lighter-weight alternative to monads by McBride and Paterson [26]. The relationships among monads, arrows and applicative functors were further elucidated by Lindley et al. [22]. The connection to monoidal functors was discussed by McBride and Paterson and has been explored further in Paterson's upcoming paper [30], which also observes that it is often much easier to work with monoidal functors. However, Paterson considers only functors on cartesian closed categories and neither Bij nor $Lens$ is cartesian closed.

Paterson also observes that it is easier to work with functors that are $Hask$-valued. We initially tried to work directly with endofunctors on Bij or even $Lens$, but reconsidered when the extra generality was not buying us much, compared to the effort needed to verify the monoidal functor laws. Nevertheless, endofunctors on bijections or lenses (when they exist) are also of interest: any such endofunctor can be pre-composed with a $Lens \to Hask$ functor. Functors on categories other than $Hask$ have appeared in other contexts; functors over isomorphisms are used in the fclabels library [39], whereas functors over partial isomorphisms $Iso \to Hask$ are essential in Rendel and Ostermann's invertible syntax descriptions [33]. They also employ a variant of $Monoidal$ functors (which they call $ProductFunctor$s). A natural question for further work is whether $Monoidal$ functors over partial isomorphisms suffice for invertible syntax descriptions, so that one can easily compose parser or pretty-printer combinators with formlenses.

7. Conclusion

Formlenses combine the features of formlets and lenses in a powerful abstraction that makes it easy to make data available on the web. Our work is ongoing. In the future, we plan to develop the semantic properties of formlenses, including proving formal round-tripping properties. We also plan to investigate ways of interacting with browsers that are not based on forms—*e.g.*, using JavaScript. Finally, we plan to explore ways of leveraging the semantic properties of formlenses to obtain efficient mechanisms for maintaining the HTML even as the underlying data changes.

References

[1] David L. Atkins, Thomas Ball, Glenn Bruns, and Kenneth C. Cox. Mawl: A domain-specific language for form-based services. *IEEE Trans. Software Eng.*, 25(3):334–346, 1999.

[2] Gavin M. Bierman. What is a categorical model of intuitionistic linear logic? In *TLCA*, pages 78–93, 1995.

[3] Joel Bjornson, Anton Tayanovskyy, and Adam Granicz. Composing reactive GUIs in F# using WebSharper. In Jurriaan Hage and Marco Morazn, editors, *Implementation and Application of Functional Languages*, volume 6647 of *Lecture Notes in Computer Science*, pages 203–216. Springer Berlin / Heidelberg, 2011.

[4] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, CA*, pages 407–419, January 2008.

[5] Aaron Bohannon, Jeffrey A. Vaughan, and Benjamin C. Pierce. Relational lenses: A language for updateable views. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems (PODS), Chicago, IL*, pages 338–347, June 2006.

[6] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The <bigwig> project. *ACM Trans. Internet Techn.*, 2(2):79–114, 2002.

[7] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 33(4–5):385–406, 2008. Short version in DBPL '05.

[8] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level web service construction. *TOPLAS*, 25(6):814–875, 2003.

[9] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *FMCO 2006*, pages 266–296, 2007.

[10] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. The essence of form abstraction. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 205–220, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. GRACE meeting notes, state of the art, and outlook. In *International Conference on Model Transformations (ICMT), Zurich, Switzerland*, pages 260–283, June 2009.

[12] Jasper Van der Jeugt. The digestive-functors package, 2012. `http://hackage.haskell.org/package/digestive-functors`.

[13] Chris Eidhof. Formlets in Haskell, 2008. `http://blog.tupil.com/formlets-in-haskell/`.

[14] Conal Elliott and Paul Hudak. Functional reactive animation. In *ICFP*, pages 263–273, 1997.

[15] 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), May 2007.

[16] J. Nathan Foster, Alexandre Pilkiewcz, and Benjamin C. Pierce. Quotient lenses. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Victoria, BC*, pages 383–395, September 2008.

[17] Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[18] Michael Hanus. Type-oriented construction of web user interfaces. In *PPDP*, pages 27–38, 2006.

[19] Michael Hanus. Putting declarative programming into the web: translating Curry to JavaScript. In *PPDP*, pages 155–166, 2007.

[20] Shinya Kawanaka and Haruo Hosoya. bixid: a bidirectional transformation language for XML. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Portland, OR*, pages 201–214, September 2006.

[21] Edward Kmett and Daniel Wagner. category-extras metapackage version 1.0.2, 2012. `http://hackage.haskell.org/package/category-extras`.

[22] Sam Lindley, Philip Wadler, and Jeremy Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electron. Notes Theor. Comput. Sci.*, 229(5):97–117, March 2011.

[23] Saunders MacLane. *Categories for the working mathematician (second edition)*. Springer-Verlag, 1998.

[24] Geoffrey Mainland. Why it's nice to be quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell (Haskell 2007)*, pages 73–82, 2007.

[25] Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Freiburg, Germany*, pages 47–58, 2007.

[26] Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.

[27] Lambert Meertens. Designing constraint maintainers for user interaction, 1998. Manuscript, available from `ftp://ftp.kestrel.edu/pub/papers/meertens/dcm.ps`.

[28] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An algebraic approach to bi-directional updating. In *ASIAN Symposium on Programming Languages and Systems (APLAS)*, pages 2–20, November 2004.

[29] Hugo Pacheco and Alcino Cunha. Generic point-free lenses. In *International Conference on Mathematics of Program Construction (MPC), Québec City, QC*, pages 331–352, June 2010.

[30] Ross Paterson. Constructing applicative functors. In *MPC*, 2012. To appear.

[31] Rinus Plasmeijer and Peter Achten. iData for the world wide web: Programming interconnected web forms. In *FLOPS '06*, pages 242–258, 2006.

[32] Raghu Rajkumar, Nate Foster, Sam Lindley, and James Cheney. Dr. Formlens, or: How I learned to stop worrying and love monoidal functors, June 2012. Available at `http://www.cs.cornell.edu/~rajkumar/papers/drformlens-full.pdf`.

[33] Tillmann Rendel and Klaus Ostermann. Invertible syntax descriptions: unifying parsing and pretty printing. In *Proceedings of the third ACM Haskell symposium (Haskell 2010)*, pages 1–12, New York, NY, USA, 2010. ACM.

[34] Jeremy Shaw and Jasper Van der Jeugt. The reform package, 2012. `http://hackage.haskell.org/package/reform-0.1.1`.

[35] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. *SIGPLAN Not.*, 37(12):60–75, December 2002.

[36] S. Doaitse Swierstra and Luc Duponcheel. Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming, Second International School-Tutorial Text*, pages 184–207, London, UK, 1996. Springer-Verlag.

[37] Peter Thiemann. An embedded domain-specific language for type-safe server-side web scripting. *ACM Trans. Inter. Tech.*, 5(1):1–46, 2005.

[38] Sjoerd Visscher. data-category package version 0.4.1, 2011. `http://hackage.haskell.org/package/data-category`.

[39] Sebastiaan Visser, Erik Hesselink, Chris Eidhof, and Sjoerd Visscher. The fclabels package, version 1.1.3, May 2012. http://hackage.haskell.org/package/fclabels.

[40] Janis Voigtländer. Bidirectionalization for free! In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Savannah, GA*, pages 165–176, January 2009.

A. Speaker Formlens

This Appendix contains the complete source code for the *Speaker* example, including the extension to lists of speakers using our generic infrastructure.

A.1 `Decomposition.hs`

The first module contains the main definitions for bijections, *MLens*es, and *MFormlens*es.

```
{-# LANGUAGE FlexibleInstances, MultiParamTypeClasses, FlexibleContexts #-}
module Decomposition where

import Control.Arrow (first)
import Data.Monoid
import Control.Category
import Prelude hiding (id, (.))
import Text.Html

  -- Datatype Declarations
data Iso c a b = Iso {fwdI :: c a b, bwdI :: c b a}
data Bij a b = Bij {fwd :: a -> b, bwd :: b -> a}

inv :: Bij a b -> Bij b a
inv (Bij g p) = Bij p g

invI :: Iso c a b -> Iso c b a
invI iso = Iso (bwdI iso) (fwdI iso)

  -- maybe lenses
data MLens a b = MLens {mget :: a -> b, mput :: Maybe a -> b -> a}

constML :: b -> MLens a b
constML b = MLens (\_ -> b) (\b _ -> b)

swapML :: MLens (a, b) (b, a)
swapML = MLens (\(a, b) -> (b, a)) (\_(b, a) -> (a, b))

pairML :: MLens a a' -> MLens b b' -> MLens (a, b) (a', b')
pairML l1 l2 = MLens (\(a, b) -> (mget l1 a, mget l2 b))
                     (\mab (a', b') ->
                       case mab of
                         Nothing ->
                           (mput l1 Nothing a', mput l2 Nothing b')
                         Just (a, b) ->
                           (mput l1 (Just a) a', mput l2 (Just b) b'))

b2ml :: Bij a b -> MLens a b
b2ml bij = MLens (fwd bij) (const $ bwd bij)

  -- Class declarations and compositional instances
class Category c => MonoidalCategory c where
    (***) :: c a a' -> c b b' -> c (a, b) (a', b')
    munitl :: Iso c (a, ()) a
    munitr :: Iso c ((), a) a
    massoc :: Iso c (x, (y, z)) ((x, y), z)

class Category c => GFunctor c f where
  gmap :: c a b -> f a -> f b

instance Functor f => GFunctor (->) f where
    gmap = fmap

  -- Duality
newtype Op f a b = Co {unCo :: f b a}

instance Category c => Category (Op c) where
    id = Co id
    (Co f) . (Co g) = Co (g . f)

instance GFunctor Bij f => GFunctor (Op Bij) f where
    gmap (Co f) = gmap (inv f)

iso2dual :: Iso c a b -> Iso (Op c) a b
iso2dual iso = Iso (Co (bwdI iso)) (Co (fwdI iso))

instance MonoidalCategory c => MonoidalCategory (Op c) where
```

```
    (Co l1) × (Co l2)    = Co (l1 × l2)
    munitl               = iso2dual munitl
    munitr               = iso2dual munitr
    massoc               = iso2dual massoc
  -- Monoidal functors
class Monoidal f where
    unit :: f ()
    (⋆) :: f a → f b → f (a, b)
infixr 6 ⋆
  -- haskell functions
instance MonoidalCategory (→) where
    f × g = λ(a, b) →     (f a, g b)
    munitl = Iso (λ(a, ()) → a) (λa → (a, ()))
    munitr = Iso (λ((), a) → a) (λa → ((), a))
    massoc = Iso (λ(a, (b, c)) → ((a, b), c)) (λ((a, b), c) → (a, (b, c)))
  -- bijections
iso2bij :: Iso (→) a b → Iso Bij a b
iso2bij (Iso to fro) = Iso (Bij to fro) (Bij fro to)
instance Category Bij where
  id = Bij id id
  f ∘ g = Bij (fwd f ∘ fwd g) (bwd g ∘ bwd f)
instance MonoidalCategory Bij where
    f × g    = Bij (λ(a, b) → (fwd f a, fwd g b))
                   (λ(fa, gb) → (bwd f fa, bwd g gb))
    munitl  = iso2bij munitl
    munitr  = iso2bij munitr
    massoc  = iso2bij massoc
  -- maybe-lenses
instance Category MLens where
    id = MLens (λx → x) (\_x → x)
    l ∘ m = MLens (mget l ∘ mget m)
              (λma c → case ma of
                         Just a →
                           mput m (Just a) (mput l (Just (mget m a)) c)
                         Nothing →
                           mput m Nothing (mput l Nothing c))
iso2mlens :: Iso Bij a b → Iso MLens a b
iso2mlens iso = Iso (b2ml (fwdI iso)) (b2ml (bwdI iso))
instance MonoidalCategory MLens where
    l1 × l2              = pairML l1 l2
    munitl               = iso2mlens munitl
    munitr               = iso2mlens munitr
    massoc               = iso2mlens massoc
  -- Namer
newtype Namer t a = Namer {runNamer :: t → (a, t)}
instance Functor (Namer t) where
  fmap f (Namer n) = Namer (first f ∘ n)
instance Monoidal (Namer t) where
  unit = Namer (λi → ((), i))
  (Namer n1) ⋆ (Namer n2) =
    Namer (λt → let (a, t′)  = n1 t in
      let (b, t″) = n2 t′ in
      ((a, b), t″))
  -- Accumulator
data Acc m a = Acc m a
instance Functor (Acc m) where
  fmap f (Acc m a) = Acc m (f a)
```

```
instance Monoid m ⇒ Monoidal (Acc m) where
    unit = Acc mempty ()
    (Acc m₁ a) ⋆ (Acc m₂ b) = Acc (m₁ ‘mappend‘ m₂) (a, b)
instance Monoid Html where
  mempty = noHtml
  mappend = (+++)
  -- Collector
data Collect e a = Collect (e → a)
instance Functor (Collect e) where
    fmap f (Collect g) = Collect (f ∘ g)
instance Monoidal (Collect e) where
    unit = Collect (const ())
    (Collect c₁) ⋆ (Collect c₂) = Collect (λe → (c₁ e, c₂ e))
  -- Maybe
instance Monoidal (Maybe) where
    unit = Just ()
    x ⋆ y = do x′ ← x
      y′ ← y
      return (x′, y′)
  -- Composition
data f ∘ g a = Comp { deComp :: f (g a) }
instance (Functor f, Functor g) ⇒ Functor (f ∘ g) where
    fmap f (Comp a) = Comp $ fmap (fmap f) a
instance (Functor f, Monoidal f, Monoidal g)
     ⇒ Monoidal (f ∘ g) where
    unit                  = Comp (fmap (const unit) unit)
    (Comp u) ⋆ (Comp v) = Comp (fmap (uncurry (⋆)) (u ⋆ v))
  -- Maybe-Formlenses
data MLift f a = MLift (Maybe a → f a)
instance Functor f ⇒ GFunctor Bij (MLift f) where
  gmap f (MLift g) = MLift (fmap (fwd f) ∘ g ∘ fmap (bwd f))
instance Functor f ⇒ GFunctor MLens^op (MLift f) where
  gmap (Co l) (MLift (g)) =
    MLift (λma → case ma of
                     Just a → fmap (mput l (Just a)) (g (Just (mget l a)))
                     Nothing → fmap (mput l Nothing) (g Nothing))
instance (Monoidal g) ⇒ Monoidal (MLift g) where
    unit                  =  MLift (\_ → unit)
    (MLift f) ⋆ (MLift g) =
      MLift (λma → case ma of
                       Just (a, b) → f (Just a) ⋆ g (Just b)
                       Nothing → f Nothing ⋆ g Nothing)
  --
infixr 0 |$|
(|$|) :: (GFunctor c f) ⇒ c a b → f a → f b
(|$|) = gmap
(⟨⋆) :: (Monoidal f, GFunctor Bij f)
       ⇒ f a → f () → f a
f ⟨⋆ g = gmap (fwdI (munitl :: Iso Bij (a, ()) a))
  (f ⋆ g)
(⋆⟩) :: (Monoidal f, GFunctor Bij f)
       ⇒ f () → f a → f a
f ⋆⟩ g = gmap (fwdI (munitr :: Iso Bij ((), a) a))
  (f ⋆ g)
```

A.2 Speaker.hs

The next module contains the definitions specific to the *Speakers* example. It also contains code needed to run the example using the Happstack server.

```haskell
{-# LANGUAGE ExistentialQuantification, TypeSynonymInstances, FlexibleInstances, OverlappingInstances #-}
module Speaker where
import Text.Html
import Decomposition
import System.IO.Unsafe
import qualified Data.List as List (union, delete, intercalate, stripPrefix, isPrefixOf)
import Data.Maybe (fromJust, fromMaybe)
import Control.Monad (liftM, msum)
import qualified Data.ByteString.Lazy.Char8 as L (unpack)
import qualified Data.ByteString.Char8 as B (pack)
import qualified Data.ByteString.Lazy.UTF8 as LU (fromString)
import Happstack.Server hiding (method)
import Happstack.Server.RqData
import qualified Happstack.Server.Internal.Types as HST
split :: (Eq a, Show a) => [a] -> [a] -> [[a]]
split del list = _split del list []
  where _split :: (Eq a) => [a] -> [a] -> [a] -> [[a]]
        _split del [] acc = ifcons acc []
        _split del ls acc = case List.stripPrefix del ls of
                              Just l -> ifcons acc (_split del l [])
                              Nothing -> _split del (tail ls) (acc ++ [head ls])
        ifcons [] l = l
        ifcons hd l = hd : l
makeName :: [Int] -> String
makeName is = "input_" ++ List.intercalate "_" (map show (reverse is))
readName :: String -> [Int]
readName n = case List.stripPrefix "input_" n of
  Just s -> reverse (map read (split "_" s))
  Nothing -> [0]
nextName :: [Int] -> (String, [Int])
nextName l@(hd : tl) = (makeName l, (hd + 1) : tl)
branch :: [Int] -> [Int]
branch l = 0 : l
unbranch (hd : tl) = tl
type Env = [(String, String)]
data Date = Date {year :: Int, month :: Int, day :: Int} deriving Show
data Speaker = Speaker {spName :: String, date :: Date} deriving Show
type HNFormlens = MLift ((Namer [Int]) ∘ ((Acc Html) ∘ (Collect Env)))
mkFormlens :: (Maybe a -> [Int] -> (Html, Env -> a, [Int])) -> HNFormlens a
mkFormlens f = MLift (\ma -> Comp (Namer (\l -> let (r, c, l') = f ma l in
  (Comp $ Acc r (Collect c), l'))))
runFormlens :: HNFormlens a -> (Maybe a -> [Int] -> (Html, Env -> a, [Int]))
runFormlens (MLift f) = (\ma l -> let (Comp (Namer g)) = f ma in
                                  let (Comp (Acc r (Collect c)), l') = g l in
                                  (r, c, l'))
(⊕) :: HNFormlens a -> HNFormlens b -> HNFormlens (Either a b)
f ⊕ g = mkFormlens (\v l -> let (n, l') = nextName l in
  let (ma, mb, tag, dispA, dispB) = case v of
                                      Just (Left a) -> (Just a, Nothing, "left", id, const noHtml)
                                      Just (Right b) -> (Nothing, Just b, "right", const noHtml, id)
                                      Nothing        -> (Nothing, Nothing, "", id, const noHtml) in
                        let (ra, ca, l1) = runFormlens f ma l' in
                        let (rb, cb, l2) = runFormlens g mb l' in
                        (hidden n tag +++ dispA ra +++ dispB rb,
```

```
                                   λe → case lookup n e of
                                     Just "left"  → Left $ ca e
                                     Just "right" → Right $ cb e,
                                   max l1 l2))
unfold :: HNFormlens a → HNFormlens a
unfold f = mkFormlens (λma l →
  let (n, l′) = nextName l in
  let collector = λe → let k = readName $ fromJust $ lookup n e in
    let (_, c, _) = runFormlens f Nothing k in
    c e in
  let (r, _, _) = runFormlens f ma (branch l) in
  (hidden n (makeName (branch l)) +++ r, collector, l′))
boxPair :: HNFormlens Value → HNFormlens Value → HNFormlens Value
boxPair f g = let h = f ⋆ g in
        mkFormlens (λmv l → let mv′ = liftM (λ(Pair a b) → (a, b)) mv in
          let (r, c, l′) = runFormlens h mv′ l in
          (r, uncurry Pair ∘ c, l′))
boxSum :: HNFormlens Value → HNFormlens Value → HNFormlens Value
boxSum f g = let h = f ⊕ g in
         mkFormlens (λmv l → let mv′ = liftM (λv → case v of
                                                      Inl a → Left a
                                                      Inr b → Right b) mv in
                                   let (r, c, l′) = runFormlens h mv′ l in
                                   (r, either Inl Inr ∘ c, l′))
boxUnfold :: HNFormlens Value → HNFormlens Value
boxUnfold f = let h = unfold f in
        mkFormlens (λmv l → let mv′ = liftM (λ(Con v) → v) mv in
          let (r, c, l′) = runFormlens h mv′ l in
          (r, Con ∘ c, l′))
boxBase :: (BaseValue a) ⇒ HNFormlens a → HNFormlens Value
boxBase fa = mkFormlens (λmv l → let (r, c, l′) = runFormlens fa (liftM unbox mv) l in
     (r, box ∘ c, l′))
data Value = Base String
           | Inl  Value
           | Inr  Value
           | Pair Value Value
           | Con  Value
  deriving Show
class BaseValue a where
  box :: a → Value
  unbox :: Value → a
instance BaseValue () where
  box = box ∘ show
  unbox = read ∘ unbox
instance BaseValue String where
  box s          = Base s
  unbox (Base s) = s
instance BaseValue Int where
  box    = box ∘ show
  unbox  = read ∘ unbox
instance (BaseValue a, BaseValue b) ⇒ BaseValue (a, b) where
  box (a, b)          = Pair (box a) (box b)
  unbox (Pair va vb) = (unbox va, unbox vb)
instance (BaseValue a, BaseValue b) ⇒ BaseValue (Either a b) where
  box (Left a) = Inl (box a)
  box (Right b) = Inr (box b)
  unbox (Inl va) = Left (unbox va)
  unbox (Inr vb) = Right (unbox vb)
```

```haskell
instance (BaseValue a) ⇒ BaseValue [a] where
  box [] = Con $ Inl $ box ()
  box (hd : tl) = Con $ Inr $ Pair (box hd) (box tl)
  unbox (Con (Inl _))    = []
  unbox (Con (Inr (Pair v tl))) = (unbox v) : (unbox tl)
instance BaseValue Date where
  box (Date y m d) = box ((y, m), d)
  unbox v = (uncurry ∘ uncurry) Date (unbox v)
instance BaseValue Speaker where
  box (Speaker s d) = box (s, d)
  unbox v = uncurry Speaker (unbox v)
data GenFormlens = forall a ∘ (BaseValue a) ⇒ GBase (HNFormlens a)
   | GPair GenFormlens GenFormlens
   | GSum              GenFormlens GenFormlens
   | GRec              String GenFormlens
   | GVar              String
fv :: GenFormlens → [String]
fv (GVar s) = [s]
fv (GPair l1 l2) = fv l1 `List.union` fv l2
fv (GSum l1 l2) = fv l1 `List.union` fv l2
fv (GRec s l) = List.delete s (fv l)
fv _ = []
fresh :: (String, GenFormlens) → String
fresh (s, l) = _fresh 0 (fv l)
  where _fresh n vars
      | s ⧺ (show n) ∈ vars = _fresh (n + 1) vars
      | otherwise = s ⧺ (show n)
subst :: (String, GenFormlens) → GenFormlens → GenFormlens
subst (s, l) (GPair l1 l2) = GPair (subst (s, l) l1) (subst (s, l) l2)
subst (s, l) (GSum l1 l2) = GSum (subst (s, l) l1) (subst (s, l) l2)
subst (s, l) l'@(GVar s')
   | s ≡ s'             = l
   | otherwise          = l'
subst (s, l) (GRec s' l')
   | s ≡ s'             = GRec s' l'
   | s' ∈ fv (l)        = let t = fresh (s', l) in
                            GRec t (subst (s, l) $ subst (s', GVar t) l')
   | otherwise          = GRec s' (subst (s, l) l')
subst _ l'              = l'
getFormlens :: GenFormlens → HNFormlens Value
getFormlens (GBase fa)    = boxBase fa
getFormlens (GPair fa fb) = boxPair (getFormlens fa) (getFormlens fb)
getFormlens (GSum fa fb) = boxSum (getFormlens fa) (getFormlens fb)
getFormlens (GRec x fx)   = boxUnfold $ getFormlens $ subst (x, GRec x fx) fx
htmlL :: Html → HNFormlens ()
htmlL h = mkFormlens (\_l → (h, const (), l))
rawTextL :: String → HNFormlens ()
rawTextL s = htmlL $ primHtml s
textL :: String → HNFormlens ()
textL s = htmlL $ lineToHtml s
tagL :: String → HNFormlens a → HNFormlens a
tagL tagname fa = tagAttrL tagname [] fa
brL :: HNFormlens ()
brL = tagL "br" unit
tagAttrL :: String → [HtmlAttr] → HNFormlens a → HNFormlens a
tagAttrL tagname attrs fa =
  mkFormlens (λma l → let (r, c, l') = runFormlens fa ma l in
                        (tag tagname r ! attrs, c, l'))
```

```
inputStringL :: HNFormlens String
inputStringL = inputStringAttrL []

inputStringAttrL :: [HtmlAttr] → HNFormlens String
inputStringAttrL attrs = mkFormlens (λma l → let (n, l′) = nextName l in
                                   (input ! [name n, value (fromMaybe "" ma)] ! attrs,
                                   fromJust ∘ lookup n, l′))

inputL :: (Show a, Read a) ⇒ HNFormlens a
inputL = inputAttrL []

inputAttrL :: (Show a, Read a) ⇒ [HtmlAttr] → HNFormlens a
inputAttrL attrs = mkFormlens (λma l → let (n, l′) = nextName l in
                             (input ! [name n, value (maybe "" show ma)] ! attrs,
                               read ∘ fromJust ∘ lookup n, l′))

formL :: String → HNFormlens a → HNFormlens a
formL actn fa =
  mkFormlens (λma l → let (r, c, l′) = runFormlens fa ma l in
                            (tag "form" r ! [method "POST", action actn], c, l′))

submitL :: Maybe String → HNFormlens ()
submitL caption =
  mkFormlens (\_l → (submit "" (fromMaybe "Submit" caption),
                            const (), l))

dateFormlens :: HNFormlens Date
dateFormlens =
        Bij (λ(y, (m, d)) → Date y m d) (λ(Date y m d) → (y, (m, d)))
        |$| tagL "table"
          (      tagL "tr"
                 (tagL "td" (textL "Year:")
                  ⋆⟩ tagL "td" (inputAttrL [size "4"]))
           ⋆ tagL "tr"
                 (tagL "td" (textL "Month:")
                  ⋆⟩ tagL "td" (inputAttrL [size "2"]))
           ⋆ tagL "tr"
                 (tagL "td" (textL "Day:")
                  ⋆⟩ tagL "td" (inputAttrL [size "2"])))

speakerFormlens :: HNFormlens Speaker
speakerFormlens = Bij to fro
   |$| textL "Name:"
   ⋆⟩ inputStringL
   ⟨⋆ brL
   ⋆ (tagL "table" $ tagL "tr" $
                           ((tagL "td" $ textL "Date:")
                             ⋆⟩ (tagL "td" $ dateFormlens)))
  where to = uncurry Speaker
          fro (Speaker s d) = (s, d)

insertButton :: HNFormlens ()
insertButton = mkFormlens (\_l → let l′ = unbranch l in
              ((tag "script" $ primHtml $ backptrcode)
                 +++ input ! [value "Add New Speaker", thetype "button",
                               strAttr "onclick" $ "insert('" ++ makeName (branch l′) ++ "')"],
                const (), l))
  where backptrcode = "prev['" ++ makeName (branch l′) ++ "'] = '" ++ makeName l′ ++ "';"

deleteButton :: HNFormlens ()
deleteButton = mkFormlens (\_l → let l′ = unbranch l in
              ((tag "script" $ primHtml $ backptrcode)
                 +++ input ! [value "Delete", thetype "button",
                               strAttr "onclick" $ "remove('" ++ makeName (branch l′) ++ "')"],
                                              const (), l))
  where backptrcode = "prev['" ++ makeName (branch l′) ++ "'] = '" ++ makeName l′ ++ "';"

preamble :: HNFormlens ()
preamble = rawTextL $ unsafePerformIO (readFile "preamble.js")
```

```
speakerListFormlens :: HNFormlens Value
speakerListFormlens =
  tagL "table" $ tagL "tr" unit ⋆⟩ (getFormlens $
    GRec "x" $
      GSum (GBase $ tagL "tr"
        $ tagL "td"      insertButton)
        (GPair (GBase $ tagL "tr"
          $               tagL "td" speakerFormlens
          ⟨⋆ tagL "td" deleteButton)
              (GVar "x")))
speakerForm = htmlL (header $ style $ primHtml "td {vertical-align: baseline;}") ⋆⟩
  preamble ⋆⟩ tagL "body" (formL "submitForm" (speakerListFormlens ⟨⋆ (submitL $ Just "Submit")))
testSpList :: IO ()
testSpList = do
    let (r, _, _) = runFormlens speakerForm (Just $ box ([Speaker "Alan Turing" (Date 23 6 2012), Speaker "Alonzo Church" (Date 30
    putStrLn $ show r
    putStrLn ""
simplifyEnv :: RqEnv → Env
simplifyEnv (qs, b, c) =
    map (λ(s, i) → (s, L.unpack $ forceRight $ HST.inputValue i)) (qs ++ (fromMaybe [] b))
  where forceRight (Right b) = b
        forceRight _ = error "Unexpected File value in environment"
myPolicy :: BodyPolicy
myPolicy = (defaultBodyPolicy "" 0 1000 1000)
serveOnly :: Html → ServerPart Response
serveOnly html = ok $ toResponse html
processData :: (Show a) ⇒ (Env → a) → ServerPart Response
processData collect =
  do methodM POST
    x ← askRqEnv
    let e = simplifyEnv x
    ok $ toResponse $ show $ collect e
dispatchSpeakerList :: [Speaker] → IO ()
dispatchSpeakerList speakers = let (rendering, collect, _) = runFormlens speakerForm (Just $ box speakers) [0] in
                               let handlers = do decodeBody myPolicy;
                                   msum [dir "submitForm" $ processData (unbox ∘ collect :: Env → [Speaker]),
                                     serveOnly $ rendering]
                               in simpleHTTP nullConf handlers
main :: IO ()
main = dispatchSpeakerList [Speaker "Alan Turing" (Date 2012 6 13), Speaker "Alonzo Church" (Date 2012 6 20)]
```

A.3 `preamble.js`

The final module is a JavaScript preamble that enables inserting and deleting form elements.

```
<script type='text/javascript'>
// globals
var prev = {}; var lastName = [100];
function nextName() {
  name = makeName(lastName);
  lastName = nextn(lastName,1);
  return(name);
}
function makeName(numlist) {
  return('input_' + numlist.reverse().join('_'));
}
function  nextn(numlist, inc) {
  return([(numlist[0]+inc)].concat(numlist.slice(1)));
}
function readName(name) {
  return(name.slice(6).split('_').map(function(x) {
    return(parseInt(x));}).reverse());
```

```
}
function insert (tag) {
  var parent = document.getElementsByName(prev[tag])[0];
  var nextField = document.getElementsByName(parent.value)[0];

  var elements = createElements(tag, parent);

  for (var i = 0; i < elements.length; i++) {
    nextField.parentNode.insertBefore(elements[i], nextField);
  }
}

function createElements (tag, parent) {
  var elements = [ document.createElement('input'),
                   document.createElement('tr'),
                   document.createElement('input')];
  elements[0].name = nextName();
  elements[0].type = 'hidden';
  elements[0].value = 'right';

  tdElements = [document.createElement('td'), document.createElement('td')];
  addAll(tdElements, elements[1]);

  td0 = [ document.createTextNode('Name:'),
          document.createElement('input'),
          document.createElement('br'),
          document.createElement('table'),
          document.createElement('br')];
  td1 = [document.createElement('input')];
  addAll(td0, tdElements[0]);
  addAll(td1, tdElements[1]);

  td0[1].name = nextName();
  td0[1].type='text';

  td1[0].type = 'button';
  td1[0].value = 'Delete';
  td1[0].onclick = function() {remove(elements[0].name)};

  tableElements = [document.createElement('tr')];
  addAll(tableElements, td0[3]);

  trElements = [document.createElement('td'), document.createElement('td')];
  addAll(trElements, tableElements[0]);

  td_0 = [document.createTextNode('Date:')];
  td_1 = [document.createElement('table')];
  addAll(td_0, trElements[0]);
  addAll(td_1, trElements[1]);

  iTableElements = [ document.createElement('tr'),
                     document.createElement('tr'),
                     document.createElement('tr')];
  addAll(iTableElements, td_1[0]);

  tr0Elements = [document.createElement('td'), document.createElement('td')];
  tr1Elements = [document.createElement('td'), document.createElement('td')];
  tr2Elements = [document.createElement('td'), document.createElement('td')];
  addAll(tr0Elements, iTableElements[0]);
  addAll(tr1Elements, iTableElements[1]);
  addAll(tr2Elements, iTableElements[2]);

  tr0Elements[0].appendChild(document.createTextNode('Day:'));
  el = tr0Elements[1].appendChild(document.createElement('input'));
  el.type = 'text';
```

```
  el.name = nextName();
  el.size='2';

  tr1Elements[0].appendChild(document.createTextNode('Month:'));
  el = tr1Elements[1].appendChild(document.createElement('input'));
  el.type = 'text';
  el.name = nextName();
  el.size='2';

  tr2Elements[0].appendChild(document.createTextNode('Year:'));
  el = tr2Elements[1].appendChild(document.createElement('input'));
  el.type = 'text';
  el.name = nextName();
  el.size='4';

  elements[2].type = 'hidden';
  elements[2].name = nextName();
  elements[2].value = parent.value;

  parent.value = elements[0].name;
  prev[tag] = elements[2].name;
  prev[elements[0].name] = parent.name;

  return(elements);
}

function addAll(elements, node) {
  for (var i = 0; i < elements.length; i++)
    node.appendChild(elements[i]);
}

var topleveltags = ['input', 'tr', 'input'];

function remove (tag) {
  var parent = document.getElementsByName(prev[tag])[0];
  var node;
  var next = document.getElementsByName(parent.value)[0];
  for (var i = 0; i < topleveltags.length;) {
    node = next;
    next = node.nextSibling;
    if (node.nodeName.toLowerCase() == topleveltags[i]) {
      node.parentNode.removeChild(node);
      i++;
    }
  }

  prev[node.value] = prev[parent.value];
  delete prev[parent.value];
  parent.value = node.value;

}
</script>
```

B. Formlens Proofs

Proof of Theorem 3 for Bij. The definitions of the operations are given above. To simplify notation we omit the bijective coercions $Lift$ and just consider the underlying functions in the proofs.

- We must show that

  $$gmap\ (Co\ id) = (Co\ id)$$

  Accordingly, consider

  $$\begin{aligned} gmap\ (Co\ id) &= \lambda g\ x \to fmap\ (put\ id\ x)\ (g\ (get\ id\ x)) \\ &= \lambda g\ x \to fmap\ id\ (g\ id\ x) \\ &= \lambda g\ x \to g\ x \\ &= (Co\ id) \end{aligned}$$

- Next we must show that

  $$gmap\ (Co\ (l \circ m)) = gmap\ (Co\ l) \circ gmap\ (Co\ m)$$

  Accordingly, consider:

  $$\begin{aligned} & gmap\ (Co\ (l \circ m)) \\ &= \lambda g\ c \to fmap\ (put\ (m \circ l)\ c)\ (g\ (get\ (m \circ l)\ c)) \\ &= \lambda g\ c \to fmap\ (put\ l\ c \circ put\ m\ (get\ l\ c)) \\ &\qquad (g\ (get\ l \circ get\ m)) \\ &= \lambda g\ c \to fmap\ (put\ l\ c)\ (fmap\ (put\ m\ (get\ l\ c)) \\ &\qquad (g\ (get\ m\ (get\ l\ c)))) \\ &= \lambda g\ c \to fmap\ (put\ l\ c)\ (\lambda b \to fmap\ (put\ m\ b) \\ &\qquad (g\ (get\ m\ b)))\ (get\ l\ c) \\ &= \lambda g \to gmap\ (Co\ l)\ (\lambda b \to fmap\ (put\ m\ b) \\ &\qquad (g\ (get\ m\ b))) \\ &= \lambda g \to gmap\ (Co\ l)\ (gmap\ (Co\ m)\ g) \\ &= gmap\ (Co\ l) \circ gmap\ (Co\ m) \end{aligned}$$

$\square$

Proof of Theorem 3 for $Lens$. The first part is as in the previous proof. For the second part, assuming f is Monoidal as an endofunctor on Set, we must show that $Lift\ f$ is monoidal as a lens functor.

- For (M1), consider:

  $$\begin{aligned} & gmap\ (f \times g)\ (u \star v) \\ &= \lambda(b, b') \to fmap\ (put\ (f \times_L g)\ (b, b')) \\ &\qquad ((u \star v)\ (get\ (f \times_L g)\ (b, b'))) \\ &= \lambda(b, b') \to fmap\ (put\ (f \times_L g)\ (b, b')) \\ &\qquad ((u \star v)\ (get\ f\ b, get\ g\ b')) \\ &= \lambda(b, b') \to fmap\ (put\ (f \times_L g)\ (b, b')) \\ &\qquad (u\ (get\ f\ b) \star v\ (get\ g\ b')) \\ &= \lambda(b, b') \to fmap\ (put\ b)\ (u\ (get\ f\ b)) \\ &\qquad \star fmap\ (put\ b')\ (v\ (get\ g\ b')) \\ &= gmap\ f\ u \star gmap\ g\ v \end{aligned}$$

- For (M2), let $m = bij2lens\ (inv\ munitl)$ in the following:

  $$\begin{aligned} & gmap\ munitl\ (u \star unit) \\ &= \lambda a \to fmap\ (put\ m\ a)\ ((u \star unit)\ (get\ m\ a)) \\ &= \lambda a \to fmap\ (put\ m\ a)\ ((u \star unit)\ ((\lambda x \to (x, ()))\ a)) \\ &= \lambda a \to fmap\ (put\ m\ a)\ (u\ a \star unit) \\ &= \lambda a \to fmap\ (\lambda\_\ (x, ()) \to x)\ (u\ a \star unit) \\ &= \lambda a \to u\ a \\ &= u \end{aligned}$$

- For (M3), we must show

  $$lmap\ munitr\ (unit \star u) = u$$

  This is symmetric to the previous argument.

- For (M4), let $m = bij2lens\ (inv\ massoc)$ in the following:

  $$\begin{aligned} & gmap\ massoc\ (u \star (v \star w)) \\ &= \lambda((x, y), z) \to fmap\ (put\ m\ ((x, y), z)) \\ &\qquad ((u \star (v \star w))\ (get\ m\ ((x, y), z))) \\ &= \lambda((x, y), z) \to fmap\ (put\ m\ ((x, y), z)) \\ &\qquad (u \star (v \star w)) \\ &= \lambda((x, y), z) \to fmap\ (put\ m\ ((x, y), z)) \\ &\qquad (u\ x \star (v\ y \star w\ z)) \\ &= \lambda((x, y), z) \to ((u\ x \star v\ y) \star w\ z) \\ &= (u \star v) \star w \end{aligned}$$

$\square$

Proof of Theorem 5 for Bij. The definitions of the operations are given above. To simplify notation we omit the bijective coercions $MLift$ and just consider the underlying functions in the proofs.

- We must show that

  $$gmap\ (Co\ id) = (Co\ id)$$

  Accordingly, consider

  $$\begin{aligned} & gmap\ (Co\ id) \\ &= \lambda g\ mx \to \textbf{case}\ mx \\ &\quad \textbf{of}\ Just\ x \to fmap\ (mput\ id\ (Just\ x)) \\ &\qquad\qquad (g\ (Just\ (mget\ id\ x))) \\ &\quad\ \ Nothing \to fmap\ (mput\ id\ Nothing) \\ &\qquad\qquad (g\ Nothing) \\ &= \lambda g\ mx \to \textbf{case}\ mx \\ &\quad \textbf{of}\ Just\ x \to g\ (Just\ x) \\ &\quad\ \ Nothing \to g\ Nothing \\ &= \lambda g\ mx \to g\ mx \\ &= (Co\ id) \end{aligned}$$

- Next we must show that

  $$gmap\ (Co\ (l \circ m)) = gmap\ (Co\ l) \circ gmap\ (Co\ m)$$

  Accordingly, consider:

  $$\begin{aligned} & gmap\ (Co\ (l \circ m)) \\ &= \lambda g\ mc \to \textbf{case}\ mc\ \textbf{of} \\ &\quad Just\ c \to fmap\ \ (mput\ (m \circ l)\ (Just\ c)) \\ &\qquad (g\ (Just\ (mget\ (m \circ l)\ c))) \\ &\quad Nothing \to fmap\ (mput\ (m \circ l)\ Nothing) \\ &\qquad (g\ Nothing) \end{aligned}$$

  For the first case we proceed as follows:

  $$\begin{aligned} & fmap\ (mput\ (m \circ l)\ (Just\ c)) \\ &\quad (g\ (Just\ (mget\ (m \circ l)\ c))) \\ &= fmap\ (mput\ l\ (Just\ c) \circ put\ m\ (Just\ (mget\ l\ c))) \\ &\quad (g\ (mget\ l \circ mget\ m)) \\ &= fmap\ (mput\ l\ (Just\ c)) \\ &\quad (fmap\ (mput\ m\ (Just\ (mget\ l\ c))) \\ &\quad\ (g\ (mget\ m\ (mget\ l\ c)))) \\ &= fmap\ (mput\ l\ (Just\ c)) \\ &\quad (\lambda mb \to \textbf{case}\ mb\ \textbf{of} \\ &\qquad Just\ b \to fmap\ (put\ m\ (Just\ b)) \\ &\qquad\qquad (g\ (mget\ m\ b)) \\ &\qquad Nothing \to e)\ (Just\ (mget\ l\ c)) \\ &= fmap\ (mput\ l\ (Just\ c)) \\ &\quad ((gmap\ (Co\ m)\ g)\ (Just\ (mget\ l\ c))) \end{aligned}$$

  and for the second case, we proceed as follows:

  $$\begin{aligned} & fmap\ (mput\ (m \circ l)\ Nothing)\ (g\ Nothing) \\ &= fmap\ (mput\ l\ Nothing \circ mput\ m\ Nothing) \end{aligned}$$

$$
\begin{array}{l}
\quad (g\ \mathit{Nothing}) \\
= \mathit{fmap}\ (\mathit{mput}\ l\ \mathit{Nothing}) \\
\quad (\mathit{fmap}\ (\mathit{mput}\ m\ \mathit{Nothing})\ (g\ \mathit{Nothing})) \\
= \mathit{fmap}\ (\mathit{mput}\ l\ \mathit{Nothing}) \\
\quad ((\lambda mb \rightarrow \\
\quad\quad \mathbf{case}\ mb\ \mathbf{of} \\
\quad\quad\quad \mathit{Just}\ b \rightarrow e' \\
\quad\quad\quad \mathit{Nothing} \rightarrow \mathit{mput}\ m\ \mathit{Nothing}\ (g\ \mathit{Nothing})) \\
\quad\quad \mathit{Nothing}) \\
= \mathit{fmap}\ (\mathit{mput}\ l\ \mathit{Nothing})\ ((\mathit{gmap}\ (\mathit{Co}\ m)\ g)\ \mathit{Nothing})
\end{array}
$$

To conclude, we have

$$
\begin{array}{l}
\quad \lambda g\ mc \rightarrow \mathbf{case}\ mc\ \mathbf{of} \\
\quad\quad \mathit{Just}\ c \rightarrow \mathit{fmap}\ (\mathit{mput}\ (m \circ l)\ (\mathit{Just}\ c)) \\
\quad\quad\quad\quad (g\ (\mathit{Just}\ (\mathit{mget}\ (m \circ l)\ c))) \\
\quad\quad \mathit{Nothing} \rightarrow \mathit{fmap}\ (\mathit{mput}\ (m \circ l)\ \mathit{Nothing}) \\
\quad\quad\quad\quad (g\ \mathit{Nothing}) \\
= \lambda g\ mc \rightarrow \mathbf{case}\ mc\ \mathbf{of} \\
\quad\quad \mathit{Just}\ c \rightarrow \mathit{fmap}\ (\mathit{mput}\ l\ (\mathit{Just}\ c)) \\
\quad\quad\quad\quad ((\mathit{gmap}\ (\mathit{Co}\ m)\ g)\ (\mathit{Just}\ (\mathit{mget}\ l\ c))) \\
\quad\quad \mathit{Nothing} \rightarrow \mathit{fmap}\ (\mathit{mput}\ l\ \mathit{Nothing}) \\
\quad\quad\quad\quad ((\mathit{gmap}\ (\mathit{Co}\ m)\ g)\ \mathit{Nothing}) \\
= \lambda g \rightarrow \mathit{gmap}\ (\mathit{Co}\ l)\ (\mathit{gmap}\ (\mathit{Co}\ m)\ g) \\
= \mathit{gmap}\ (\mathit{Co}\ l) \circ \mathit{gmap}\ (\mathit{Co}\ m)
\end{array}
$$

This concludes the proof. □

Proof of Theorem 5 for MLens. The first part is given by the previous theorem. For the second part, assuming f is Monoidal as an endofunctor on Set, we must show that $\mathit{MLift}\ f$ is monoidal as a maybe-lens functor.

- For (M1), consider:

$$
\begin{array}{l}
\quad \mathit{gmap}\ (f \times g)\ (u \star v) \\
= \lambda mbb \rightarrow \mathbf{case}\ mbb\ \mathbf{of} \\
\quad\quad \mathit{Just}\ (b, b') \rightarrow \\
\quad\quad\quad \mathit{fmap}\ (\mathit{mput}\ (\mathit{pairML}\ f\ g)\ (\mathit{Just}\ (b, b'))) \\
\quad\quad\quad\quad ((u \star v)\ (\mathit{Just}\ (\mathit{mget}\ (f \times_L g)\ (b, b')))) \\
\quad\quad \mathit{Nothing} \rightarrow \\
\quad\quad\quad \mathit{fmap}\ (\mathit{mput}\ (f \times_L g)\ \mathit{Nothing}) \\
\quad\quad\quad\quad ((u \star v)\ \mathit{Nothing})
\end{array}
$$

Consider the first case:

$$
\begin{array}{l}
\quad \mathit{fmap}\ (\mathit{mput}\ (\mathit{pairML}\ f\ g)\ (\mathit{Just}\ (b, b'))) \\
\quad\quad ((u \star v)\ (\mathit{Just}\ (\mathit{mget}\ (f \times_L g)\ (b, b')))) \\
= \mathit{fmap}\ (\mathit{put}\ (\mathit{pairML}\ f\ g)\ (\mathit{Just}\ (b, b')) \\
\quad\quad ((u \star v)\ (\mathit{Just}\ (\mathit{mget}\ f\ b, \mathit{mget}\ g\ b'))) \\
= \mathit{fmap}\ (\mathit{mput}\ (\mathit{pairML}\ f\ g)\ (\mathit{Just}\ (b, b'))) \\
\quad\quad (u\ (\mathit{Just}\ (\mathit{mget}\ f\ b)) \star v\ (\mathit{Just}\ (\mathit{mget}\ g\ b'))) \\
= \mathit{fmap}\ (\mathit{mput}\ (\mathit{Just}\ b))\ (u\ (\mathit{Just}\ (\mathit{mget}\ f\ b))) \\
\quad \star\ \mathit{fmap}\ (\mathit{mput}\ (\mathit{Just}\ b'))\ (v\ (\mathit{Just}\ (\mathit{mget}\ g\ b'))) \\
= (\mathit{gmap}\ f\ u \star \mathit{gmap}\ g\ v)\ (\mathit{Just}\ (b, b'))
\end{array}
$$

Similarly, in the second case:

$$
\begin{array}{l}
\quad \mathit{fmap}\ (\mathit{mput}\ (f \times_L g)\ \mathit{Nothing}) \\
\quad\quad ((u \star v)\ \mathit{Nothing} \\
= \mathit{fmap}\ (\mathit{mput}\ (\mathit{pairML}\ f\ g)\ \mathit{Nothing}) \\
\quad\quad (u\ \mathit{Nothing} \star v\ \mathit{Nothing}) \\
= \mathit{fmap}\ (\mathit{mput}\ f\ \mathit{Nothing})\ (u\ \mathit{Nothing}) \\
\quad \star\ \mathit{fmap}\ (\mathit{mput}\ f\ \mathit{Nothing})\ (v\ \mathit{Nothing}) \\
\quad\quad (u\ \mathit{Nothing} \star v\ \mathit{Nothing}) \\
= (\mathit{gmap}\ f\ u \star \mathit{gmap}\ g\ v)\ \mathit{Nothing}
\end{array}
$$

Hence, to conclude:

$$
\begin{array}{l}
\quad \lambda mbb \rightarrow \mathbf{case}\ mbb\ \mathbf{of} \\
\quad\quad \mathit{Just}\ (b, b') \rightarrow \\
\quad\quad\quad \mathit{fmap}\ (\mathit{mput}\ (\mathit{pairML}\ f\ g)\ (\mathit{Just}\ (b, b'))) \\
\quad\quad\quad\quad ((u \star v)\ (\mathit{Just}\ (\mathit{mget}\ (f \times_L g)\ (b, b')))) \\
\quad\quad \mathit{Nothing} \rightarrow \\
\quad\quad\quad \mathit{fmap}\ (\mathit{mput}\ (f \times_L g)\ \mathit{Nothing}) \\
\quad\quad\quad\quad ((u \star v)\ \mathit{Nothing}) \\
= \lambda mbb \rightarrow \mathbf{case}\ mbb\ \mathbf{of} \\
\quad\quad \mathit{Just}\ (b, b') \rightarrow (\mathit{gmap}\ f\ u \star \mathit{gmap}\ g\ v)\ (\mathit{Just}\ (b, b')) \\
\quad\quad \mathit{Nothing} \rightarrow (\mathit{gmap}\ f\ u \star \mathit{gmap}\ g\ v)\ \mathit{Nothing} \\
= \lambda mbb \rightarrow (\mathit{gmap}\ f\ u \star \mathit{gmap}\ g\ v)\ mbb \\
= \mathit{gmap}\ f\ u \star \mathit{gmap}\ g\ v
\end{array}
$$

- For (M2), let $m = \mathit{bij2lens}\ (\mathit{inv}\ \mathit{munitl})$ in the following:

$$
\begin{array}{l}
\quad \mathit{gmap}\ \mathit{munitl}\ (u \star \mathit{unit}) \\
= \lambda ma \rightarrow \mathbf{case}\ ma\ \mathbf{of} \\
\quad\quad \mathit{Just}\ a \rightarrow \mathit{fmap}\ (\mathit{put}\ m\ (\mathit{Just}\ a)) \\
\quad\quad\quad\quad ((u \star \mathit{unit})\ (\mathit{Just}\ (\mathit{get}\ m\ a))) \\
\quad\quad \mathit{Nothing} \rightarrow \mathit{fmap}\ (\mathit{put}\ m\ \mathit{Nothing}) \\
\quad\quad\quad\quad ((u \star \mathit{unit})\ \mathit{Nothing})
\end{array}
$$

For the first case, we reason as follows:

$$
\begin{array}{l}
\quad \mathit{fmap}\ (\mathit{put}\ m\ a)\ ((u \star \mathit{unit})\ ((\lambda x \rightarrow (x, ()))\ a)) \\
= \mathit{fmap}\ (\mathit{put}\ m\ a)\ ((u\ (\mathit{Just}\ a) \star \mathit{unit})) \\
= \mathit{fmap}\ (\lambda\_\ (x, ()) \rightarrow x)\ (u\ (\mathit{Just}\ a) \star \mathit{unit}) \\
= u\ (\mathit{Just}\ a)
\end{array}
$$

and for the second case:

$$
\begin{array}{l}
\quad \mathit{fmap}\ (\mathit{put}\ m\ \mathit{Nothing})\ ((u \star \mathit{unit})\ \mathit{Nothing}) \\
= \mathit{fmap}\ (\mathit{put}\ m\ \mathit{Nothing})\ (u\ \mathit{Nothing} \star \mathit{unit}) \\
= \mathit{fmap}\ ((\lambda\_\ (x, ()) \rightarrow x)\ (u\ \mathit{Nothing} \star \mathit{unit}) \\
= u\ \mathit{Nothing}
\end{array}
$$

hence, we can conclude:

$$
\begin{array}{l}
\quad \lambda ma \rightarrow \mathbf{case}\ ma\ \mathbf{of} \\
\quad\quad \mathit{Just}\ a \rightarrow \mathit{fmap}\ (\mathit{put}\ m\ (\mathit{Just}\ a)) \\
\quad\quad\quad\quad ((u \star \mathit{unit})\ (\mathit{Just}\ (\mathit{get}\ m\ a))) \\
\quad\quad \mathit{Nothing} \rightarrow \mathit{fmap}\ (\mathit{put}\ m\ \mathit{Nothing}) \\
\quad\quad\quad\quad ((u \star \mathit{unit})\ \mathit{Nothing}) \\
= \lambda ma \rightarrow \mathbf{case}\ ma\ \mathbf{of} \\
\quad\quad \mathit{Just}\ a \rightarrow u\ (\mathit{Just}\ a) \\
\quad\quad \mathit{Nothing} \rightarrow u\ \mathit{Nothing} \\
= \lambda ma \rightarrow u\ ma \\
= u
\end{array}
$$

- For (M3), we must show

$$\mathit{lmap}\ \mathit{munitr}\ (\mathit{unit} \star u) = u$$

This is symmetric to the previous argument.

- For (M4), let $m = \mathit{bij2lens}\ (\mathit{inv}\ \mathit{massoc})$ in the following:

$$
\begin{array}{l}
\quad \mathit{gmap}\ \mathit{massoc}\ (u \star (v \star w)) \\
= \lambda mxyz \rightarrow \mathbf{case}\ mxyz\ \mathbf{of} \\
\quad\quad \mathit{Just}\ ((x, y), z) \rightarrow \\
\quad\quad\quad \mathit{fmap}\ (\mathit{put}\ m\ (\mathit{Just}\ ((x, y), z))) \\
\quad\quad\quad\quad ((u \star (v \star w))\ (\mathit{Just}\ (\mathit{get}\ m\ ((x, y), z)))) \\
\quad\quad \mathit{Nothing} \rightarrow \\
\quad\quad\quad \mathit{fmap}\ (\mathit{put}\ m\ \mathit{Nothing}) \\
\quad\quad\quad\quad ((u \star (v \star w))\ \mathit{Nothing})
\end{array}
$$

For the first case, we reason as follows:

$$
\begin{array}{l}
\quad \mathit{fmap}\ (\mathit{mput}\ m\ (\mathit{Just}\ ((x, y), z))) \\
\quad\quad ((u \star (v \star w))\ (\mathit{Just}\ (\mathit{get}\ m\ ((x, y), z))))
\end{array}
$$

$$
\begin{aligned}
&= \mathit{fmap}\ (\mathit{mput}\ m\ (\mathit{Just}\ ((x, y), z))) \\
&\qquad (u\ (\mathit{Just}\ x) \star (v\ (\mathit{Just}\ y) \star w\ (\mathit{Just}\ z))) \\
&= ((u\ (\mathit{Just}\ x) \star v\ (\mathit{Just}\ y)) \star w\ (\mathit{Just}\ z)) \\
&= ((u \star v) \star w)\ (\mathit{Just}\ ((x, y), z))
\end{aligned}
$$

and for the second case, we reason as follows:

$$
\begin{aligned}
&\mathit{fmap}\ (\mathit{mput}\ m\ \mathit{Nothing}) \\
&\qquad ((u \star (v \star w))\ \mathit{Nothing}) \\
&= \mathit{fmap}\ (\mathit{mput}\ m\ \mathit{Nothing}) \\
&\qquad (u\ \mathit{Nothing} \star (v\ \mathit{Nothing} \star w\ \mathit{Nothing})) \\
&= (u\ \mathit{Nothing} \star v\ \mathit{Nothing}) \star w\ \mathit{Nothing} \\
&= ((u \star v) \star w)\ \mathit{Nothing}
\end{aligned}
$$

To conclude,

$$
\begin{aligned}
&\lambda mxyz \to \mathbf{case}\ mxyz\ \mathbf{of} \\
&\qquad \mathit{Just}\ ((x, y), z) \to \\
&\qquad\quad \mathit{fmap}\ (\mathit{put}\ m\ (\mathit{Just}\ ((x, y), z))) \\
&\qquad\qquad ((u \star (v \star w))\ (\mathit{Just}\ (\mathit{get}\ m\ ((x, y), z)))) \\
&\qquad \mathit{Nothing} \to \\
&\qquad\quad \mathit{fmap}\ (\mathit{put}\ m\ \mathit{Nothing}) \\
&\qquad\qquad ((u \star (v \star w))\ \mathit{Nothing}) \\
&= \lambda mxyz \to \mathbf{case}\ mxyz\ \mathbf{of} \\
&\qquad \mathit{Just}\ ((x, y), z) \to ((u \star v) \star w)\ (\mathit{Just}\ ((x, y), z)) \\
&\qquad \mathit{Nothing} \to ((u \star v) \star w)\ \mathit{Nothing} \\
&= \lambda mxyz \to ((u \star v) \star w)\ mxyz \\
&= (u \star v) \star w
\end{aligned}
$$

$\square$