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

Raghuraj Kumar  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 code, making programs difficult to write, reason about, and maintain. We investigate the connection between linearity and bidirectional transformations and develop a translation from a linear pattern syntax in formlenses 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,

```
data Speaker = Speaker { name :: String, date :: Date }
data Date = Date { year :: Int, month :: Int, day :: Int }
```

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

```
renderer :: Speaker -> Html
renderer (Speaker n (Date y m d)) =
    input! [name "name", value n] +++ br +++
    lineToHtml "Year: " +++
    input! [name "year", value (show y)]
    lineToHtml "Month: " +++
    input! [name "month", value (show m)] +++ br +++
    lineToHtml "Day: " +++
    input! [name "day", value (show d)] +++ br +++
```

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:

```
data Env = [(String, String)]
```

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

```
collector :: Env -> Speaker
```

```
let n = read (fromJust (lookup "name" e)) in
let y = read (fromJust (lookup "year" e)) in
let m = read (fromJust (lookup "month" e)) in
let d = read (fromJust (lookup "day" e)) in
Speaker n (Date y m d)
```

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 `Speakers` 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 namespace.

```
type Formlet a = Int -> (Html, Env -> a, Int)
```

The names of any form elements in the HTML are drawn from the namespace, 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.

¹This code uses combinators from the Text.Html module.
We can also define a formlet combinator that accepts an integer:

```haskell
inputInt :: Formlet Int
inputInt i = let n = show i in
    (input ! (name n),
     λe → read (fromJust (lookup n e)), i + 1)
```

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].

```haskell
class Functor f where
    fmap f :: (a → b) → f a → f b

class (Functor f) ⇒ Applicative f where
    pure :: a → f a
    (⊗) :: f (a → b) → f a → f b
```

Intuitively, the `pure` function injects a value of type `a` into the type `f a`, while the `⊗` 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 `⊗`:

- `pure id ⊗ u = u`
- `pure (f ⊗ v) ⊗ w = u ⊗ (v ⊗ w)`
- `pure f ⊗ pure x = pure (f x)`
- `u ⊗ pure x = pure (λx → f x) ⊗ u`

The applicative instance for formlets is defined as follows:

```haskell
instance Applicative Formlet where
    pure a i = (noHtml, const a, i)
    (f ⊗ g) i = let (x, p, i') = f i in
        let (y, q, i'') = g i' in
        (x ++ y, λe → p e (q e), i'')
```

The `pure` formlet generates empty HTML and has the constant function as its `collector`. The `⊗` operator threads the namespase through its arguments from left to right and accumulates the generated HTML. Its `collect` function applies the function (of type `a → 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 `Speakers` as follows:

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

speakerForm :: Formlet Speaker
    speakerForm = pure Speaker ⊗ inputString ⊗ dateForm
```

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 → 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 **Formlenses**. 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 → 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 non-trivial. 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`:

```haskell
class GApplicative c f where
gpure :: a → f a
 gap :: (f (c a b)) → f a → f b
```

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

```haskell
gpure :: a → Formlens a
gap :: Formlens (c a b) → Formlens a → Formlens b
```
For the latter, we would need to (somehow) combine a form that consumes and produces a \( \text{Bij} \ a \ b \) or \( \text{Lens} \ a \ b \) 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 \( \text{gapp} \), which has a fundamentally asymmetric type.

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

\textbf{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 combinator. 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 \textit{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.

\section{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 skinned on a first reading and referred back to as needed.

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

\begin{verbatim}
  data Bij a b = Bij { fwd :: a -> b, bwd :: b -> a }
\end{verbatim}

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

\begin{equation}
\text{bwd} \circ \text{fwd} \circ \text{bij} = \text{id} = \text{fwd} \circ \text{bwd} \circ \text{bij}
\end{equation}

Bijections admit many useful combinators such as the following:

\begin{verbatim}
  idB :: Bij a a
  idB x = x
\end{verbatim}

\textbf{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:

\begin{verbatim}
  data Lens a b = Lens { get :: a -> b, put :: a -> b -> a }
\end{verbatim}

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

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

\begin{verbatim}
  put a (get a) = a -- GetPut
  get (put a b) = b -- PutGet
\end{verbatim}

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

Lenses admit the following combinators (among many others):

\begin{verbatim}
  idL :: Lens a a
  idL = lens id (const id)
  constL :: b -> Lens a b
  constL b = Lens (const b) (\lambda a -> a)
  fwdL :: Lens (a, b) a
  fwdL = Lens (\lambda (a,b) -> (\lambda (a -> (a,b)))
  bwdL x = x
  bwdL (\lambda (a,b) -> (\lambda (a -> (a,b)))
\end{verbatim}

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

\begin{verbatim}
  bij2lens :: Bij a b -> Lens a b
  bij2lens = Lens (\lambda a -> a) (\lambda a -> a)
\end{verbatim}

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

\begin{verbatim}
  class Category c where
    id :: c x x
    (o) :: c y z -> c x y -> c x z
\end{verbatim}

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

\begin{verbatim}
  instance Category (->) where
    id = \x -> x
    o o g = \x -> o (g x)

  instance Category Bij where
    id = Bij id id
    o o g = Bij (fwd o fwd g) (bwd g o bwd f)

  instance Category Lens where
    id = Lens id (const id)
    o o m = Lens (get o get m)
\end{verbatim}

An isomorphism in a category \( c \) is a morphism \( f :: c a b \) with an “inverse” \( g :: c b a \) satisfying \( f o g = id = g o f \). In a computational setting, it is convenient to represent isomorphisms explicitly.

\begin{verbatim}
  data Iso c a b = Iso { fwdI :: c a b, bwdI :: c b a }
\end{verbatim}

Isomorphisms are expected to satisfy the following condition:

\begin{verbatim}
  fwdI is o bwdI is = id = bwdI is o fwdI is
\end{verbatim}
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 \((\cdot)\) and pairing types \(\cdot\) and also includes a pairing operation on c-arrows \((\cdot \times \cdot)\) as well as c-isomorphisms relating unit and pairing:

\[
\begin{align*}
\text{class Category } c &\Rightarrow \text{MonoidalCategory } c \text{ where } \\
(\cdot \times \cdot) &:: c a_1 b_1 \rightarrow c a_2 b_2 \rightarrow c (a_1, a_2) (b_1, b_2) \\
munitl &:: \text{Iso } (c (a, ())) a \\
munitr &:: \text{Iso } c (a, (()) a \\
massoc &:: \text{Iso } c ((a, b), d) ((a, b), d)
\end{align*}
\]

Monoidal categories are expected to satisfy a number of additional laws, which essentially state that \((\cdot)\) is a unit with respect to \((\cdot \times \cdot)\) 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:

\[
\begin{align*}
\text{instance MonoidalCategory } (\rightarrow) \text{ where } \\
f \times g &:: \lambda (a, b) \rightarrow (f a, g b) \\
munitl &:: \text{Iso } (\lambda (a, ()) \rightarrow a) (\lambda a \rightarrow (a, ())) \\
munitr &:: \text{Iso } (\lambda ((), a) \rightarrow a) (\lambda a \rightarrow ()), a) \\
massoc &:: \text{Iso } (\lambda (a, (b, d)) \rightarrow ((a, b), d)) (\lambda ((a, b), d) \rightarrow (a, b), d))
\end{align*}
\]

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 \text{fst} and \text{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 \text{Bij} and \text{Lens}. For \text{Bij}, we first show how to lift isomorphisms on \text{Hask} to \text{Bij} (there is some redundancy here, which we tolerate for the sake of uniformity):

\[
\begin{align*}
\text{iso2bij} &:: \text{Iso } (\rightarrow) a b \rightarrow \text{Iso } \text{Bij } a b \\
\text{iso2bij } (\text{Iso } \text{to } \text{fro}) &:: \text{Iso } (\text{Bij } \text{to } \text{fro}) (\text{Bij } \text{to } \text{fro})
\end{align*}
\]

\[
\begin{align*}
\text{instance MonoidalCategory Bij } \text{where } \\
f \times g &:: \text{Bij } (\lambda (a, b) \rightarrow (\text{fwd } f a, \text{fwd } g b)) (\lambda (fa, gb) \rightarrow (\text{bwd } f fa, \text{bwd } g gb)) \\
munitl &:: \text{iso2bij } \text{munitl} \\
munitr &:: \text{iso2bij } \text{munitr} \\
massoc &:: \text{iso2bij } \text{massoc}
\end{align*}
\]

For \text{Lens}, we first lift the coercion \text{bij2lens} from bijections to lenses to act on isomorphisms:

\[
\begin{align*}
\text{iso2lens} &:: \text{Iso Bij } a b \rightarrow \text{Iso } \text{Lens a b} \\
\text{iso2lens } (\text{Iso } \text{to } \text{fro}) &:: \text{Iso } (\text{bij2lens } \text{to}) (\text{bij2lens } \text{fro})
\end{align*}
\]

\[
\begin{align*}
\text{instance MonoidalCategory Lens } \text{where } \\
l_1 \times l_2 &:: l_1 \times l_2 \\
munitl &:: \text{iso2lens } \text{munitl} \\
munitr &:: \text{iso2lens } \text{munitr} \\
massoc &:: \text{iso2lens } \text{massoc}
\end{align*}
\]

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

\[
\begin{align*}
\text{newtype } c^\text{op} a b &:: \text{Co } (\text{unCo } :: c b a) \\
\text{instance Category } c \Rightarrow \text{Category } c^\text{op } \text{where }
\end{align*}
\]

\[
\begin{align*}
\text{id} &:: \text{Co } \text{id} \\
(f \circ g) \circ (g \circ f) &:: \text{Co } (f \circ g) \circ (g \circ f)
\end{align*}
\]

The dual of any monoidal category is also monoidal:

\[
\begin{align*}
\text{iso2dual } (\text{Iso } \text{to } \text{fro}) &:: \text{Iso } (\text{Co } \text{fro}) (\text{Co } \text{to}) \\
\text{instance MonoidalCategory } c^\text{op } \text{where } \\
(f_1) \times (f_2) &:: (f_1 \times f_2) (\text{Co } f_1, \text{Co } f_2) \\
munitl &:: \text{iso2dual } \text{munitl} \\
munitr &:: \text{iso2dual } \text{munitr} \\
massoc &:: \text{iso2dual } \text{massoc}
\end{align*}
\]

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

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

\[
\begin{align*}
\text{class Category } c \Rightarrow \text{GFunctor } f \text{ where } \\
gmap &:: c a b \rightarrow f a \rightarrow f b
\end{align*}
\]

Again, since we are interested only in \text{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 \text{Hask}.

**Monoidal functors.** Next, we consider monoidal functors:\(^3\)

\[
\begin{align*}
\text{class Monoidal } f \text{ where } \\
\text{unit } &:: f () \\
(\ast ) &:: f a \rightarrow f b \rightarrow f (a, b)
\end{align*}
\]

Note that we do not explicitly identify the domain of \(f\) in the type class \text{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 \text{Hask}). However, in stating the laws for monoidal functors, we will implicitly assume that the domain is a monoidal category—in particular, that \(\cdot \times \cdot\), \text{munitl}, \text{munitr}, and \text{massoc} are defined.

McBride and Paterson use the following laws for monoidal functors,

\[
\begin{align*}
\text{fmap } (f \times g) (u \ast v) &:: \text{fmap } f u \ast \text{fmap } g u \\
\text{fmap } \text{fwd } f \ast \text{unit } &:: u \\
\text{fmap } \text{snd } (\text{unit } \ast u) &:: u \\
\text{fmap } \text{assoc } (u \ast (v \ast w)) &:: (u \ast v) \ast w
\end{align*}
\]

which implicitly assume we are working with endofunctors on \text{Hask}. In addition, they use the operations \text{fwd } :: (a, b) \rightarrow a and \text{snd } :: (a, b) \rightarrow b which are not available in all monoidal categories (for example, there is no bijection \text{Bij } (a, b) a, and while there is a lens \text{Lens } (a, b) a, it is not an isomorphism). Since we are only interested in functors whose range is \text{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 \textit{ 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.
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:

\[
\begin{align*}
data & ~ \text{Monoidal} Hask \text{if there exist operations } gmap \cdot (u \ast v) = fmap f \ast fmap g \\
(M1) & \quad gmap \, \text{munit} \, (u \ast \text{unit}) = u \\
(M2) & \quad gmap \, \text{munit} \, (u \ast \text{unit}) = u \\
(M3) & \quad gmap \, \text{massoc} \, (u \ast (v \ast w)) = (u \ast v) \ast w
\end{align*}
\]

This means that existing Applicative functors can be viewed as Monoidal functors. Another is that it makes it possible to adjust the behavior of formlets in a modular fashion, by composing other functors and vice versa, and the translations back and forth are inverses.

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.

\[
\begin{align*}
\text{instance} & \quad (\text{Monoidal } f, \text{Applicative } f) \Rightarrow \text{Applicative } f \\
\text{pure} & \quad a = \text{fmap } (\lambda x \rightarrow a) \text{ unit} \\
\text{mf} & \quad \text{unit} = \text{Comp } (\text{fmap } (\lambda \text{ unit} \rightarrow \text{unit})) \\
\text{strength} & \quad \text{Comp } (\text{fmap } (\lambda x \rightarrow f x)) \text{ (mf } \circ \text{ mx})
\end{align*}
\]

Lemma 1 ([26]). For any functor \( f : \text{Hask} \rightarrow \text{Hask} \), there exist operations \( (\text{pure}, \circ) \) making \( f \) an applicative functor if and only if there exists operations \( (\text{unit}, \ast) \) 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 \( \text{Hask} \) has rich structure, including so-called structure:

\[
\begin{align*}
\text{strength} \quad & : \quad \text{Functor } f \Rightarrow (a, f b) \Rightarrow f (a, b) \\
\text{strength} \quad & = \quad \text{fmap } (\lambda x \rightarrow f x) (mf \ast mx)
\end{align*}
\]

which is derived from the fact that \( \text{fmap} \) can be used on arbitrary higher-order functions. This is not possible for functors \( f \) from an arbitrary category \( c \rightarrow \text{Hask} \).

The traditional Formlet type based on applicative functors can easily 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

\[
\text{type Formlet } a = \text{Html} \rightarrow (\text{Html, Env} \rightarrow a, \text{Html})
\]

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 \( \text{Env} \rightarrow \text{Maybe} \, a \). 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:

\[
\quad \text{type Formlens } a = a \rightarrow \text{Int} \rightarrow (\text{Html, Env} \rightarrow a, \text{Int})
\]

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 \( \text{Lift} \) 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 \( \text{Bij} \) to \( \text{Hask} \) and \( \text{Lens} \) to \( \text{Hask} \). 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 formlenses as the composition of three simpler functors.

Namers. The functor \( \text{Namer } t \) generates names, which are used to identify elements of the form.

\[
\text{newtype Namer } t a = \text{Namer } (\text{runNamer } :: t \rightarrow (a, t))
\]

Note that \( \text{Namer } t \) holds the type \( t \) abstract. We will typically use \( \text{Ints} \). It is straightforward to show that \( \text{Namer } t \) is a functor

\[
\text{instance Functor } (\text{Namer } t) \text{ where } \text{fmap} \, f \, (\text{Namer } n) = \text{Namer } (\text{first } f \circ n)
\]

and also has monoidal structure:

\[
\text{instance Monoidal } (\text{Namer } t) \text{ where } \text{unit} = \text{Namer } (\lambda x \rightarrow ((x, t)) \\
\text{let } (n_1, t') = \text{Namer } (\lambda x \rightarrow \text{let } (a, t') = n_1, t \text{ in } \text{let } (b, t'') = n_2, t' \text{ in } ((a, b), t''))
\]

The following lemma records the fact that \( \text{Namer} \) is a valid instance of \( \text{Monoidal} \).

Lemma 2. \( \text{Namer } t \) 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 \( \text{Monoidal} \). However, we still expect these properties to hold.

Accumulators. The \( \text{Acc } m \) functor accumulates the HTML generated by formlets.

\[
\text{data Acc } m a = \text{Acc } m a
\]

\( \text{Acc} \) is parameterized on a monoid \( m \). Monoids are defined in Haskell using the following (standard) type class.

\[
\text{class Monoid } m \text{ where } \text{mempty} :: m, \text{mappend} :: m \times m \rightarrow m
\]

\( \text{mempty} \) and \( \text{mappend} \) are the identity and operation of the monoid, respectively.

\[\text{Note that the monoid type class and monoidal functors are different!}\]
class Monoid a where
  mempty :: a
  mappend :: a → a → a

where the \texttt{mempty} element is an identity for the \texttt{mappend} operation, which is associative. We will typically take \(m\) to be the monoid of \texttt{Html} documents:

\[
\text{instance Monoid } \texttt{Html} \text{ where}
\]
\[
\text{mempty = noHtml} \quad \text{mappend = (+ + +)}
\]

The functor and monoidal functor structures on \(Acc\ m\) can be defined as follows:

\[
\text{instance } \text{Functor } (\text{Acc } m) \text{ where}
\]
\[
fmap f \ (\text{Acc } m \ a) = \text{Acc } m \ (f \ a)
\]

\[
\text{instance } \text{Monoidal } (\text{Acc } m) \text{ where}
\]
\[
\text{unit = Acc mempty} \\
(\text{Acc } m_1 \ a) \ast (\text{Acc } m_2 \ b) = \text{Acc } (m_1 \ 'mappend' \ m_2) \ (a, b)
\]

Assuming \(m\) is a \texttt{Monoid}, it is straightforward to verify that \(Acc \ m\) is monoidal.

\textbf{Collectors.} The \texttt{Collect e} functor reconstructs the value encoded in a form response.

\[
\text{data } \texttt{Collect e} \ a = \text{Collect } (\epsilon \ a)
\]

Values of type \texttt{Collect e} \(a\) wrap a function \(\epsilon \ a\) that builds an \(a\) from an environment \(\epsilon\). The following definitions give the functor and monoidal functor structure on collectors:

\[
\text{instance } \text{Functor } (\text{Collect } \epsilon) \text{ where}
\]
\[
fmap f \ (\text{Collect } \epsilon \ g) = \text{Collect } (f \circ g)
\]

\[
\text{instance } \text{Monoidal } (\text{Collect } \epsilon) \text{ where}
\]
\[
\text{unit} = \text{Collect } (\lambda x \rightarrow ()) \\
(\text{Collect } c_1) \ast (\text{Collect } c_2) = \text{Collect } (\lambda x \rightarrow (c_1 \ e, c_2 \ e))
\]

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

\[
\text{newtype } \texttt{Formlet } a = \text{Formlet } \{ \text{unFormlet} :: (\text{Namer } \texttt{Int}) \circ ((\text{Acc Html}) \circ (\text{Collect Env})) \ a \}
\]

with functor and monoidal structure lifted to \texttt{Formlet} in the obvious way:

\[
\text{instance } \text{Functor } \texttt{Formlet} \text{ where}
\]
\[
fmap f \ f\text{Formlet} = f\text{Formlet} \circ fmap \ f \circ \text{unFormlet}
\]

\[
\text{instance } \text{Monoidal } \texttt{Formlet} \text{ where}
\]
\[
\text{unit} = \text{Formlet } \text{unit} \\
f_1 \ast f_2 = \text{Formlet } (\text{unFormlet } f_1 \ast \text{unFormlet } f_2)
\]

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

\[
\text{type } \texttt{Formlet } a = \text{Int} \rightarrow (\text{Html}, \text{Env} \rightarrow a, \text{Int})
\]

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

\textbf{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 \texttt{Maybe} type.

First, we check that \texttt{Maybe} admits functor and monoidal functor structure:

\[
\text{instance } \text{Functor } \texttt{Maybe} \text{ where}
\]
\[
fmap _\epsilon \texttt{Nothing} = \texttt{Nothing} \\
fmap f \ (\texttt{Just } a) = \texttt{Just } (f \ a)
\]

\[
\text{instance } \text{Monoidal } \texttt{Maybe} \text{ where}
\]
\[
\text{unit} = \texttt{Just } () \\
(\texttt{Just } x) \ast (\texttt{Just } y) = \texttt{Just } (x, y) \\
\texttt{Nothing} \ast \_ = \texttt{Nothing}
\]

Then we build “validating” formlets by composing the \texttt{Maybe} functor with the collector component,

\[
\text{newtype } \texttt{ValFormlet} a = \texttt{ValFormlet } \{ \text{unValFormlet} :: (\text{Namer } \texttt{Int}) \circ ((\text{Acc Html}) \circ (\text{Collect Env}) \circ \texttt{Maybe}) \ a \}
\]

which is isomorphic to the following type:

\[
\text{type } \texttt{ValFormlet } a = \text{Int} \rightarrow (\text{Html}, \text{Env} \rightarrow \text{Maybe } a, \text{Int})
\]

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 \texttt{Nothing} if the input is not a string encoding a positive integer.

\[
\text{inputNat } :: \texttt{ValFormlet } \texttt{Int}
\]
\[
\text{inputNat } i = \texttt{let } n = \texttt{show } i \ \text{in} \\
\text{input ! [\text{name } n],} \\
(\lambda x \rightarrow \text{case reads (fromJust (lookup n e)) of} \\
\quad [(v, **)] \mid v > 0 \rightarrow \text{Just v} \\
\quad _ \rightarrow \texttt{Nothing}), \\
\quad i + 1)
\]

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

\textbf{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 \(a \rightarrow \text{Formlet} a\). 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 \(a \rightarrow \text{Formlet} a\) 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 \texttt{Lens} \rightarrow \texttt{Hask} functors, and hence also \texttt{Bj} \rightarrow \texttt{Hask}, as a special case.

\textbf{Lifting.} Given a type operator \texttt{f}, the type operator \texttt{Lift } \texttt{f} maps each type \(a\) to the type \(a \rightarrow f \ a\).

\[
\text{data } \texttt{Lift } f \ a = \texttt{Lift } (a \rightarrow f \ a)
\]

In addition, \texttt{Lift } \texttt{f} preserves monoidal structure.

\[
\text{instance } \text{Monoidal } f \Rightarrow \text{Monoidal } (\text{Lift } f) \text{ where}
\]
\[
\text{unit} = \text{Lift } (\lambda x \rightarrow \text{unit}) \\
(\text{Lift } f) \ast (\text{Lift } g) = \text{Lift } (\lambda (a, b) \rightarrow f \ a \ast g \ b)
\]

Note that due to the contravariance of \texttt{Lift } \texttt{f}, in general \texttt{Lift } \texttt{f} may not be an instance of \texttt{Functor}. Nevertheless, provided that \(f\) is a
We lift the monoidal functor and generalized functor structures to
instance Functor f ⇒ GFunctor Bij (Lift f) where
gmap bij (Lift g) = Lift (fmap (fwd bij) ∘ g ∘ bwd bij)
instance Functor f ⇒ GFunctor Lens°° (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 functors from
Bij to Haskell and from Lens°° to Haskell. 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
Formlenses 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 f) = Formlens (gmap b f)
instance GFunctor Lens°° Formlens where
  gmap l (Formlens f) = Formlens (gmap l f)

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 (λm 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 formlenses versions of the html, text,
and inputInt combinators.

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

text :: 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 formlenses combinators takes the initial value v
for the form as a parameter.

Next, it will be useful to define several helper operators, ((*)
and (*)). These operators behave mostly like (*) 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)
  )

The definition of the (∗) is symmetric, but eliminates the () value
using munitr instead of munitl.

Third, let us define a bijection on Dates.

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 Dates
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 (∘) with
(∗), (∗), and (∗) 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 (λa → unit)
  (MLift f) ★ (MLift g) =
    MLift (λa →
    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°° (MLift f) where
  gmap (Co l) (MLift (g)) =
    MLift (λa →
    case ma of
      Just a → fmap (put l a) (g (Just (get l a)))
      Nothing → fmap (put l (Nothing)) (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,
\[
\text{data MLens } a \ b = \text{MLens } \{ \text{mget : } a \rightarrow b, \\
\text{mput : } \text{Maybe } a \rightarrow b \rightarrow a \}
\]
then we can make MLens into a monoidal category, analogous to Lenses (the details are in Appendix 7?). 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:
\[
\text{instance Functor } g \Rightarrow \text{GFUnctor } \text{MLens}^{op} (\text{MLift } g) \text{ where}
\]
\[
gmap \ (\text{Co } l) \ (\text{MLift } (g)) \Rightarrow \\
\text{MLift } (\text{Lma} \rightarrow \\
\text{case } \text{ma} \text{ of}
\]
\[
\text{Just } a \rightarrow \text{fnmap } (\text{mput } l \text{ ma}) \ (g \ (\text{Just } (\text{mget } l \text{ a})))
\]
\[
\text{Nothing} \rightarrow \text{fnmap } (\text{mput } l \text{ Nothing}) \ (g \text{ Nothing})
\]
Moreover, these functors also have monoidal structure.

**Theorem 5.** If \(f\) is a (monoidal) functor then \(\text{MLift } f\) is also a (monoidal) \(\text{GFUnctor } \text{MLens}\), and hence \(\text{MLift } f\) is a (monoidal) \(\text{GFUnctor } \text{Bij}\).

Finally we can define \(M\text{Formlens}\) as follows:
\[
\text{newtype } M\text{Formlens } a = M\text{Formlens } \{ \text{MLift Formlet } a \}
\]
We will use \(M\text{Formlens}\) to define a type-theoretic sum operator in Section 5.

### 3.3 Semantic Properties

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

Let \(\text{extract}\) be a function from \(\text{Html}\) to \(\text{Env}\) 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 \(\text{GETPUT}\) for all values \(x\) and namesources \(t\):

\[
(h, e, n') = fl \ x \ n \ \text{extract } h \subseteq e
\]
\[
\text{collect } e \equiv x
\]
\[
\text{COLLECTEXTRACT}
\]

We believe that our formlens combinators satisfy \(\text{COLLECTEXTRACT}\) (or preserve it assuming that their arguments satisfy it), and that if \(fl\) satisfies \(\text{COLLECTEXTRACT}\) then so do \((\text{gmap } bj fl)\) and \((\text{gmap } l fl)\), where \(bj\) 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 \(\text{Int}\) and \(g\) is a Formlens () then \(f \times g\) is a Formlens \((\text{Int},())\). If the programmer wants to obtain a Formlens \(\text{Int}\) instead, they must \(\text{gmap}\) the bijection \(\text{mget\ lift}\) on the result to eliminate the spurious (). Of course, the derived \((*)\) operator does this, but having to remember when to use \((*)\) versus \((*)\) 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-quotiation [24] to represent this syntax and Template Haskell [35] to implement the translation.

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 \([\text{formc} | \text{body } \text{yields } e]\). The body consists of a sequence of nodes where a node is either an element \(< t \text{ ats}\ n_1 \ldots n_k \text{ />},\) a text node \(s\), a spliced HTML expression \(e\); or a nested formlet binding \(f \rightarrow p\). 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 \((\rightarrow)\) first goes by structural recursion, producing formlet combinators as a result. Sequences of nodes are then combined using the \((*)\) operator and \(\text{unit}\) handles the empty sequence. On the other side, we extract patterns from the form using \((\rightarrow)\), again following a straightforward structural recursion. The functor map operation \(\text{fmap}\) 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 \(\text{dateForm}\) as:
\[
\text{dateFormC } :: \text{ Formlet Date}
\]
\[
\text{dateFormC } =
\]
\[
[\text{formc} | \text{Year } : \{ \text{inputInt } \rightarrow y\}<br/> \text{Month} : \{ \text{inputInt } \rightarrow m\}<br/> \text{Day } : \{ \text{inputInt } \rightarrow d\}<br/> \text{yields } \text{Date } y m d ]
\]

The translation of this program is the code for \(\text{dateForm}\) given in the introduction.

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

---

\footnote{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 \(<\neq/>\) for simulating sequences of nodes.}
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 allows **yields** clauses to be arbitrary, but handles the common case of bijective mappings between source values and forms. Instead of \[\textbf{form} \mid n_1 \ldots n_k \textbf{yields } p\] we write \[\textbf{form} \mid p \leftrightarrow n_1 \ldots n_k\], 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. This is required as the domain of **GFunctor Bij 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:

\[
\begin{align*}
\text{dateFormlensB} &:: \text{Formlens Date} \\
\text{dateFormlensB} &= \textbf{form} \mid \text{Date } y m d \\
&\quad \leftrightarrow \text{Year: } \{\text{inputIntL } \rightarrow y\} <br/> \\
&\quad \text{Month: } \{\text{inputIntL } \rightarrow m\} <br/> \\
&\quad \text{Day: } \{\text{inputIntL } \rightarrow d\} <br/> \end{align*}
\]

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

\[
\begin{align*}
\text{Bij } (\lambda (\text{Date } y m d) \rightarrow ()), (y, ()), (m, ()), (d, ())) \\
&\quad (\lambda ()), (y, ()), (m, ()), (d, ()) \rightarrow \text{Date } y m d
\end{align*}
\]

It would be straightforward to adapt the desugaring transformation to output \(*\) and \((\#)\) 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 \[[\textbf{forml} \mid p \leftrightarrow n_1 \ldots n_k]\]. 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 \(\text{putold}\) binds the parts of the original input value that are ignored by the formlens. The pattern \(\text{pnew}\) (which is also an expression as it contains no wildcards) combines the ignored part of the input bound by \(\text{putold}\) 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:

\[
\begin{align*}
\text{speakerFormL} &:: \text{Formlens Speaker} \\
\text{speakerFormL} &= \{\textbf{forml} \mid \text{Name: } \{\text{inputStringL } \rightarrow \text{name}\} \}
\end{align*}
\]

which is desugared into:

\[
\begin{align*}
\text{speakerFormL} &= \textbf{gmap} (\text{Lens } (\lambda (\text{Speaker name } \_ ) \rightarrow ())), (\lambda ()), (\lambda ()) \rightarrow \text{Speaker name } \_ ) \\
&\quad (\text{textL } "\text{Name: \_ } \ast \text{inputString})
\end{align*}
\]

where \(\_\) 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))

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 (⋆) operator, to combine formlenses that operate on the components of a product to obtain a formlens out of the entire product; and by using unfold 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 over arbitrary algebraic datatypes. These features make formlenses, algebraic datatypes, and boxed values.

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.

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.

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 (⋆) operator. Figure 4 defines the operators (⊕) and unfold, which handle sums and recursive types respectively.

At a high level, the sum formlens (f ⊕ g) works as follows. It adds a hidden field to the rendered HTML, which 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 indentation 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 Speakers, 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 Speakers 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 → 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.
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 Haskell-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 ⇒ Haskell functor. Functors on categories other than Haskell have appeared in other contexts; functors over isomorphisms are used in the fclabels library [39], whereas functors over partial isomorphisms Iso ⇒ Haskell are essential in Rendel and Ostermann’s invertible syntax descriptions [33]. They also employ a variant of Monoidal functors (which they call ProductFunctors). 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


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, MLenses, and MFormlenses.

```haskell
{-# 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 \rightarrow b, bwd :: b \rightarrow a }
inv :: Bij a b \rightarrow Bij b a
inv (Bij g p) = Bij p g
invl :: Iso c a b \rightarrow Iso c b a
invl iso = Iso (bwdI iso) (fwdI iso)

-- maybe lenses

data MLens a b = MLens { mget :: a \rightarrow b, mput :: Maybe a \rightarrow b \rightarrow a }

constML :: a \rightarrow MLens a b
constML b = MLens (\_ \rightarrow b) (\lambda _ \rightarrow b)

swapML :: MLens (a, b) (b, a)
swapML = MLens (\lambda (a, b) \rightarrow (b, a)) (\lambda (b, a) \rightarrow (a, b))

pairML :: MLens a a' \rightarrow MLens b b' \rightarrow MLens (a, b) (a', b')
pairML l1 l2 = MLens (\lambda (a, b) \rightarrow (mget l1 a, mget l2 b))

(b2ml :: Bij a b \rightarrow MLens a b)
b2ml bij = MLens (fwd bij) (const $ bwd bij)

-- Class declarations and compositional instances

class Category c \Rightarrow MonoidalCategory c where
- \otimes :: c a a' \rightarrow c b b' \rightarrow c (a, b) (a', b')
munitI :: Iso c (a()) a
munitr :: Iso c (((),) a) a
massoc :: Iso c (x, (y, z)) ((x, y), z)

class Category c \Rightarrow GFunctor c f where
  gmap :: c a b \rightarrow f a \rightarrow f b

instance Functor f \Rightarrow GFunctor (\rightarrow) f where
  gmap = fmap

-- Duality

newtype f'op a b = Co { unCo :: f b a }

instance Category c \Rightarrow Category c'op where
  id = Co id
  (Co f) \circ (Co g) = Co (g \circ f)

instance GFunctor Bij f \Rightarrow GFunctor Bij'op f where
  gmap (Co f) = gmap (inv f)

iso2dual :: Iso c a b \rightarrow Iso c'op a b
iso2dual iso = Iso (Co (bwdI iso)) (Co (fwdI iso))

instance MonoidalCategory c \Rightarrow MonoidalCategory c'op where
```

example, including the extension to lists of speakers using our generic infrastructure.

example, including the extension to lists of speakers using our generic infrastructure.
\[( l_1 ) \times ( l_2 ) = \text{Co} ( l_1 \times l_2 )\]
\[
munitl = \text{iso2dual munitl}
\]
\[
munitr = \text{iso2dual munitr}
\]
\[
\text{massoc} = \text{iso2dual massoc}
\]

-- Monoidal functors

class Monoidal f where

unit :: f ()

(*) :: f a -> f b -> f (a, b)

infixr 6 *

-- haskell functions

instance MonoidalCategory (\to\) where

f x g = \l(a,b) -> (f a, g b)

munitl = Iso (\l(a,()) -> a) (\a -> (a, ()))

munitr = Iso (\l(() , a) -> a) (\a -> (() , a))

massoc = Iso (\l(a, (b, c)) -> ((a, b), c)) (\l((a, b), c) -> (a, (b, c)))

-- bijections

iso2bij :: Iso (\to\) 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 o g = Bij (\a b -> fwd f o fwd g) (\a b -> bwd g o bwd f)

instance MonoidalCategory Bij where

f x g = Bij (\l(a,b) -> (fwd f a, fwd g b))
(\l(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 (\x -> x) (\x -> x)

l o m = MLens (\ma c -> \n
\case ma of

Just a ->

\nput m (Just a) (\n (mput m (\ma a) c)

Nothing ->

\nput m Nothing (\n (mput m Nothing c))

iso2mlens :: Iso Bij a b -> Iso MLens a b

iso2mlens iso = Iso (b2ml (fwdI iso)) (b2ml (bwdI iso))

instance MonoidalCategory MLens where

l_1 \times l_2 = \text{pairML l_1 l_2}

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 (\f o n)

instance Monoidal (Namer t) where

unit = Namer (\x -> ((), x))

(Namer n_1) \star (Namer n_2) =

Namer (\l t -> \n (let (b, t') = n_2 \ t' \n in

((a, b), t'))) (\n
-- 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 (λ e → (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, Monoidal f, Monoidal g) ⇒ Monoidal (f ∘ g) where
  unit = Comp (fmap (const () 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 ⧿

($\langle\rangle$) :: (GFunctor c f) ⇒ c a b → f a → f b
($\langle\rangle$) = gmap

($\langle\rangle$t) :: (Monoidal f, GFunctor Bij f)
  ⇒ f a → f () → f a
  f ($\langle\rangle$t g = gmap (fwdI (munitl :: Iso Bij (a,)) a))
  (f $\langle\rangle$t)

($\langle\rangle$t) :: (Monoidal f, GFunctor Bij f)
  ⇒ f () → f a → f a
  f ($\langle\rangle$t g = gmap (fwdI (munitr :: Iso Bij ((), a)) a))
  (f $\langle\rangle$t)
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 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 del list []
  where _split :: (Eq a) ⇒ [a] → [a] → [a] → [[a]]
      _split del [] acc = ifscons acc []
      _split del ls acc = case List.stripPrefix del ls of
                           Just l → _split del (tail ls) (acc ++ [head ls])
                           Nothing → _split del (tail ls) (acc ++ [head ls])

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

makeName, readName, nextName, branch, unbranch :: [Int] → [Int]

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]) o ((Acc Html) o (Collect Env)))
mkFormlens :: (Maybe a → [Int] → (Html, Env → a, [Int])) → HNFormlens a
mkFormlens f = MLift (λma → Comp (Namer (λl → 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) = (λma 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 (λl → 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)
                       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,
```
\begin{align*}
\lambda e \to \text{case lookup n e of} \\
\text{Just "left" } &\to \text{ Left $\& a e$} \\
\text{Just "right" } &\to \text{ Right $\& b e$, max $l_1 (l_2)$}
\end{align*}

\text{unfold :: HNFormlens } a \to \text{ HNFormlens } a

\text{unfold } f = \text{ mkFormlens } (\lambda ma l \to

\text{let } (n, l') = \text{nameName } l \text{ in}

\text{let collector } = \lambda e \to \text{let } k = \text{readName } \$ \text{ fromJust } \$ \text{ lookup } n \ e \text{ in}

\text{let } (\_, c, \_) = \text{runFormlens } f \text{ Nothing } k \text{ in}

c \ e \text{ in}

\text{let } (r, r, \_) = \text{runFormlens } f \text{ ma } (\text{branch } l) \text{ in}

(\text{hidden } n \text{ (makeName } \text{ (branch } l) ) \oplus r, \text{ collector, } l'))

\text{boxPair :: HNFormlens } \text{Value } \to \text{ HNFormlens } \text{Value } \to \text{ HNFormlens } \text{Value}

\text{boxPair } f \ g = \text{let } h = f \downarrow g \text{ in}

\text{mkFormlens } (\lambda mv l \to \text{let } mv' = \text{liftM} \ (\lambda (\text{Pair } a b) \to (a, b)) \text{ mv } \text{ in}

\text{let } (r, c, l') = \text{runFormlens } h \text{ mv' } l \text{ in}

(r, \text{uncurry } \text{Pair } c, l'))

\text{boxSum :: HNFormlens } \text{Value } \to \text{ HNFormlens } \text{Value } \to \text{ HNFormlens } \text{Value}

\text{boxSum } f \ g = \text{let } h = f \downarrow g \text{ in}

\text{mkFormlens } (\lambda mv l \to \text{let } mv' = \text{liftM} \ (\lambda v \to \text{ case } v \text{ of}

\text{Inl } a \to \text{ Left } a

\text{Inr } b \to \text{ Right } b) \text{ mv } \text{ in}

\text{let } (r, c, l') = \text{runFormlens } h \text{ mv' } l \text{ in}

(r, \text{either } \text{Inl } \text{Inr } c, l'))

\text{boxUnfold :: HNFormlens } \text{Value } \to \text{ HNFormlens } \text{Value}

\text{boxUnfold } f = \text{let } h = \text{unfold } f \text{ in}

\text{mkFormlens } (\lambda mv l \to \text{let } mv' = \text{liftM} \ (\lambda v \to \text{ case } v \text{ of}

\text{Inl } a \to \text{ Left } a

\text{Inr } b \to \text{ Right } b) \text{ mv } \text{ in}

\text{let } (r, c, l') = \text{runFormlens } h \text{ mv' } l \text{ in}

(r, \text{box } c, l'))

\text{data } \text{Value } = \text{ Base } \text{ String}

| Inl Value |
| Inr Value |
| Pair Value Value |
| Con Value |

deriving \text{Show}

\text{class } \text{BaseValue } a \text{ where}

\text{box :: } a \to \text{ Value}

\text{unbox :: Value } \to a

\text{instance } \text{BaseValue } \text{Value } \text{where}

\text{box } = \text{box } \circ \text{show}

\text{unbox } = \text{read } \circ \text{unbox}

\text{instance } \text{BaseValue } \text{String } \text{where}

\text{box } s = \text{Base } s

\text{unbox } (\text{Base } s) = s

\text{instance } \text{BaseValue } \text{Int } \text{where}

\text{box } = \text{box } \circ \text{show}

\text{unbox } = \text{read } \circ \text{unbox}

\text{instance } (\text{BaseValue } a, \text{BaseValue } b) \Rightarrow \text{ BaseValue } (a, b) \text{ where}

\text{box } (a, b) = \text{Pair } (\text{box } a) (\text{box } b)

\text{unbox } (\text{Pair } va \ vb) = (\text{unbox } va, \text{unbox } vb)

\text{instance } (\text{BaseValue } a, \text{BaseValue } b) \Rightarrow \text{ BaseValue } (\text{Either } a b) \text{ where}

\text{box } \text{(Left } a) = \text{Inl } (\text{box } a)

\text{box } \text{(Right } b) = \text{Inr } (\text{box } b)

\text{unbox } (\text{Inl } va) = \text{Left } (\text{unbox } va)

\text{unbox } (\text{Inr } vb) = \text{Right } (\text{unbox } vb)
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 v))) = (unbox v) : (unbox tl)

instance BaseValue Date where

    box (Date y m d) = box ((y, m), d)

unbox v = (uncurry o 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 o (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')

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 (λ , l → (h, const (λ ), 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 (λ ma l → let (r, c, l') = runFormlens fa ma l in
            (tag tagname r!attrs, c, l'))
preamble

:: [inputStringAttrL = inputStringL
inputStringL :: [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))

dateFormlenses :: HNFormlens Date
dateFormlenses =
    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"]))

speakerFormlenses :: HNFormlens Speaker
speakerFormlenses = Bij to fro

|$ textL "Name:"*
* inputStringL

* brL
  *(tagL "table" $ tagL "tr" $
     (tagL "td" $ textL "Date:")
     * (tagL "td" $ dateFormlenses))

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

```javascript
<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());

  </script>
```
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')
    ];
    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')
    ];
    addAll(iTableElements, td_1[0]);

    tr0Elements = [
        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
  \[ \text{gmap} \ (\text{Co id}) = (\text{Co id}) \]
  Accordingly, consider
  \[ \text{gmap} \ (\text{Co id}) = \lambda x \to \text{fmap} \ (\text{put id} x) \ (g \ (\text{get id} x)) \]
  \[ = \lambda x \to \text{fmap} \ (g \ id \ x) \]
  \[ = (\text{Co id}) \]

- Next we must show that
  \[ \text{gmap} \ (\text{Co} \ (l \circ m)) = \text{gmap} \ (\text{Co} \ l) \circ \text{gmap} \ (\text{Co} \ m) \]
  Accordingly, consider:
  \[ \text{gmap} \ (\text{Co} \ (l \circ m)) \]
  = \lambda c \to \text{fmap} \ (\text{put} \ (m \circ l) \ c) \ (g \ (\text{get} \ (m \circ l) \ c))
  = \lambda c \to \text{fmap} \ (\text{put} \ l \ c \circ \text{put} \ m \ (\text{get} \ l \ c)) \ (g \ (\text{get} \ l \ \text{circ} \ m))
  = \lambda c \to \text{fmap} \ (\text{put} \ l \ c) \ (\lambda b \to \text{fmap} \ (\text{put} \ m \ b) \ (g \ (\text{get} \ m) \ b))
  = \lambda c \to \text{gmap} \ (\text{Co} \ l) \ (\lambda b \to \text{fmap} \ (\text{put} \ m \ b) \ (g \ (\text{get} \ m) \ b))
  = \lambda c \to \text{gmap} \ (\text{Co} \ l) \ (\text{gmap} \ (\text{Co} \ m) \ g)
  = \text{gmap} \ (\text{Co} \ l) \circ \text{gmap} \ (\text{Co} \ m) \]

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 endo-functor on Set, we must show that \( \text{Lift} \ f \) is monoidal as a lens functor.

- For (M1), consider:
  \[ \text{gmap} \ (f \times g) \ (u \times v) \]
  = \lambda (b, b') \to \text{fmap} \ (\text{put} \ (f \times g) \ (b, b'))
  = \lambda (b, b') \to \text{fmap} \ (\text{put} \ (f \times g) \ (b, b')) \ ((u \times v) \ (\text{get} \ (f \times g) \ (b, b')))
  = \lambda (b, b') \to \text{fmap} \ (\text{put} \ (f \times g) \ (b, b')) \ ((u \times v) \ (g \ (f \ (b) \ (\text{get} \ g \ b'))))
  = \lambda (b, b') \to \text{fmap} \ (\text{put} \ (f \times g) \ (b, b')) \ ((u \ (\text{get} \ f \ b) \ (\text{get} \ g \ b')))
  = \lambda (b, b') \to \text{fmap} \ (\text{put} \ (f \times g) \ (b, b')) \ ((u \ (\text{get} \ f \ b) \ (\text{get} \ g \ b')))
  = \text{gmap} \ f \ u \times \text{gmap} \ g \ v

- For (M2), let \( m = \text{bij2lens} \ (\text{inv munitl}) \) in the following:
  \[ \text{gmap} \ \text{munitl} \ (u \times \text{unit}) \]
  = \lambda a \to \text{fmap} \ (\text{put} \ m \ a) \ ((u \times \text{unit}) \ (\text{get} \ m \ a))
  = \lambda a \to \text{fmap} \ (\text{put} \ m \ a) \ ((u \times \text{unit}) \ ((\lambda x \to (x, ())) \ a))
  = \lambda a \to \text{fmap} \ (\text{put} \ m \ a) \ ((u \times \text{unit}) \ a \times \text{unit})
  = \lambda a \to \text{fmap} \ (\lambda \_ (x, ())) \ x \ ((u \times \text{unit}) \ a \times \text{unit})
  = \lambda a \to u \ a \times \text{unit}
  = u

- For (M3), we must show
  \[ \text{lmap} \ \text{munitr} \ (\text{unit} * u) = u \]
  This is symmetric to the previous argument.

- For (M4), let \( m = \text{bij2lens} \ (\text{inv massoc}) \) in the following:
  \[ \text{gmap} \ \text{massoc} \ (u \times (v \times w)) \]
  = \lambda (x, y, z) \to \text{fmap} \ (\text{put} \ m \ ((x, y), z))
  = \lambda (x, y, z) \to \text{fmap} \ (\text{put} \ m \ ((x, y), z)) \ ((u \times (v \times w)) \ (\text{get} \ m \ ((x, y), z)))
  = \lambda (x, y, z) \to \text{fmap} \ (\text{put} \ m \ ((x, y), z)) \ ((u \times (v \times w)) \ w)
  = \lambda (x, y, z) \to \text{fmap} \ (\text{put} \ m \ ((x, y), z)) \ ((u \times v) \ y \ w)
  = (u \times v) \ w

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

- We must show that
  \[ \text{gmap} \ (\text{Co id}) = (\text{Co id}) \]
  Accordingly, consider:
  \[ \text{gmap} \ (\text{Co id}) \]
  = \lambda x \to \text{case} \ m \ x \ \text{of} \ Just x \to \text{fmap} \ (\text{mput} \ m \ (\text{Just} \ x)) \ (g \ (\text{Just} \ (\text{mget} \ m \ (\text{Just} \ x))))
  \quad \text{Nothing} \to \text{fmap} \ (\text{mput} \ m \ \text{Nothing}) \ (g \ \text{Nothing})
  = (\text{Co id})

- Next we must show that
  \[ \text{gmap} \ (\text{Co} \ (l \circ m)) = \text{gmap} \ (\text{Co} \ l) \circ \text{gmap} \ (\text{Co} \ m) \]
  Accordingly, consider:
  \[ \text{gmap} \ (\text{Co} \ (l \circ m)) \]
  = \lambda m \to \text{case} \ m \ \text{of} \ Just c \to \text{fmap} \ (\text{mput} \ (m \circ l) \ (\text{Just} \ c)) \ (g \ (\text{Just} \ (\text{mget} \ (m \circ l) \ c)))
  \quad \text{Nothing} \to \text{fmap} \ (\text{mput} \ (m \circ l) \ \text{Nothing}) \ (g \ \text{Nothing})
  = (\text{Co id})

For the first case we proceed as follows:

\[ \text{fmap} \ (\text{mput} \ (m \circ l) \ (\text{Just} \ c)) \ (g \ (\text{Just} \ (\text{mget} \ (m \circ l) \ c))) \]
\[ = \text{fmap} \ (\text{mput} \ l \ (\text{Just} \ c) \circ \text{put} \ m \ (\text{Just} \ (\text{mget} \ l \ c))) \ (g \ (\text{mget} \ l \ \text{circ} \ m \ l)) \]
\[ = \text{fmap} \ (\text{mput} \ l \ (\text{Just} \ c)) \ (\text{fmap} \ (\text{mput} \ m \ (\text{Just} \ (\text{mget} \ l \ c))) \ (g \ (\text{mget} \ l \ \text{circ} \ m \ l)) \]
\[ (\lambda m \to \text{case} \ m \ \text{of} \ Just b \to \text{fmap} \ (\text{put} \ m \ (\text{Just} \ b)) \ (g \ (\text{mget} \ l \ m))) \]
\[ (\text{Nothing} \to \text{e} \ (\text{Just} \ (\text{mget} \ l \ c))) \]
\[ = \text{fmap} \ (\text{mput} \ l \ (\text{Just} \ c)) \ ((\text{gmap} \ (\text{Co} \ m \ g) \ (\text{Just} \ (\text{mget} \ l \ c))) \]
and for the second case, we proceed as follows:

\[ \text{fmap} \ (\text{mput} \ (m \circ l) \ \text{Nothing}) \ (g \ \text{Nothing}) \]
\[ = \text{fmap} \ (\text{mput} \ l \ \text{Nothing} \circ \text{mput} \ m \ \text{Nothing}) \]
(g Nothing)
= fmap (mput l Nothing)
= fmap (mput (mput m Nothing) (g Nothing))
= fmap (mput l Nothing)

\((\lambda b \mapsto \text{case } mb \text{ of})\)
\hspace{1cm}
\begin{align*}
\text{Just } b & \mapsto e' \\
\text{Nothing} & \mapsto \text{mput } m \text{ Nothing (g Nothing)}
\end{align*}
= fmap (mput l Nothing) ((gmap (Co m) g) Nothing)

To conclude, we have
\[
\lambda g \text{ mc } \rightarrow \text{ case } mc \text{ of} \begin{align*}
\text{Just } c & \mapsto \text{fmap } (\text{mput } (m \circ l) (\text{Just } c)) \\
\text{Nothing} & \mapsto \text{fmap } (\text{mput } (m \circ l) \text{ Nothing})
\end{align*}
\]
= \lambda g \text{ mc } \rightarrow \text{ case } mc \text{ of} \begin{align*}
\text{Just } c & \mapsto \text{fmap } (\text{mput } l \text{ (Just } c)) \\
\text{Nothing} & \mapsto \text{fmap } (\text{mput } l \text{ Nothing})
\end{align*}
= \lambda g & \rightarrow \text{gmap } (Co l) \text{ (gmap } (Co m) g)\\
= \text{gmap } (Co l) \circ \text{gmap } (Co m)

This concludes the proof. 

\textit{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 \text{MLift } f\ is monoidal as a maybe-lens functor.

For (M1), consider:
\[
gmap (f \times g) (u \otimes v)
\]
= \lambda bb \rightarrow \text{ case } mb \text{ of} \begin{align*}
\text{Just } (b', b) & \mapsto \\
\text{fmap } (\text{mput } (\text{pairML } g f) \text{ (Just } (b, b'))) & \mapsto \\
\text{Nothing} & \mapsto \text{fmap } (\text{mput } (f \times L g) \text{ (g Nothing)})
\end{align*}
Consider the first case:
\[
\text{fmap } (\text{mput } (\text{pairML } f g) \text{ (Just } (b, b'))) \\
= \text{fmap } (\text{mput } (\text{pairML } g f) \text{ (Just } (b', b)))
\]
= \text{fmap } (\text{mput } (\text{pairML } f g) \text{ (g Nothing)})
\]
\[
\begin{align*}
(u & \text{ Just } (\text{mget } f b)) * v \rightarrow \text{Just } (\text{mget } g b'))
= \text{fmap } (\text{mput } (\text{Just } b)) \hspace{1cm}
\text{fmap } (\text{mput } (\text{Just } f)) \hspace{1cm}
\text{fmap } (\text{mput } (\text{Just } b')) \hspace{1cm}
\text{fmap } (\text{mput } (\text{Just } g))
\end{align*}
\]
\[
= (\text{gmap } f u \otimes \text{gmap } g v) \text{ (Just } (b, b'))
\]
Similarly, in the second case:
\[
\text{fmap } (\text{mput } (f \times L g) \text{ Nothing})
\]
= \text{fmap } (\text{mput } (\text{pairML } f g) \text{ Nothing})
\]
= \text{fmap } (\text{mput } (f \text{ Nothing}) \otimes u \text{ Nothing})
\]
\[
\begin{align*}
\text{fmap } (\text{mput } (f \text{ Nothing}) \otimes v \text{ Nothing}) & = \\
\text{gmap } f u \otimes \text{gmap } g v \text{ Nothing}
\end{align*}
\]
Hence, to conclude:
\[
\lambda bb \rightarrow \text{ case } mb \text{ of} \begin{align*}
\text{Just } (b, b') & \mapsto \\
\text{fmap } (\text{mput } (\text{pairML } f g) \text{ (Just } (b, b'))) & \mapsto \\
\text{Nothing} & \mapsto \text{fmap } (\text{mput } (f \times L g) \text{ Nothing})
\end{align*}
\]
\[
= \lambda bb \rightarrow \text{ case } mb \text{ of} \begin{align*}
\text{Just } (b, b') & \mapsto \\
\text{fmap } (\text{mput } (f \times L g) \text{ (g Nothing)}) & \mapsto \\
\text{Nothing} & \mapsto \text{fmap } (\text{mput } (f \times L g) \text{ Nothing})
\end{align*}
\]
\[
\lambda bb \rightarrow \text{ case } mb \text{ of} \begin{align*}
\text{Just } (b, b') & \mapsto \\
\text{fmap } (\text{mput } (f \times L g) \text{ (g Nothing)}) & \mapsto \\
\text{Nothing} & \mapsto \text{fmap } (\text{mput } (f \times L g) \text{ Nothing})
\end{align*}
\]
\[
\lambda bb \rightarrow \text{ case } mb \text{ of} \begin{align*}
\text{Just } (b, b') & \mapsto \\
\text{fmap } (\text{mput } (f \times L g) \text{ Nothing}) & \mapsto \\
\text{Nothing} & \mapsto \text{fmap } (\text{mput } (f \times L g) \text{ Nothing})
\end{align*}
\]
\[
\lambda bb \rightarrow \text{ case } mb \text{ of} \begin{align*}
\text{Just } (b, b') & \mapsto \\
\text{fmap } (\text{mput } (f \times L g) \text{ Nothing}) & \mapsto \\
\text{Nothing} & \mapsto \text{fmap } (\text{mput } (f \times L g) \text{ Nothing})
\end{align*}
\]
\[
\lambda bb \rightarrow \text{ case } mb \text{ of} \begin{align*}
\text{Just } (b, b') & \mapsto \\
\text{fmap } (\text{mput } (f \times L g) \text{ Nothing}) & \mapsto \\
\text{Nothing} & \mapsto \text{fmap } (\text{mput } (f \times L g) \text{ Nothing})
\end{align*}
\]
\[
\lambda bb \rightarrow \text{ case } mb \text{ of} \begin{align*}
\text{Just } (b, b') & \mapsto \\
\text{fmap } (\text{mput } (f \times L g) \text{ Nothing}) & \mapsto \\
\text{Nothing} & \mapsto \text{fmap } (\text{mput } (f \times L g) \text{ Nothing})
\end{align*}
\]
\[
\lambda bb \rightarrow \text{ case } mb \text{ of} \begin{align*}
\text{Just } (b, b') & \mapsto \\
\text{fmap } (\text{mput } (f \times L g) \text{ Nothing}) & \mapsto \\
\text{Nothing} & \mapsto \text{fmap } (\text{mput } (f \times L g) \text{ Nothing})
\end{align*}
\]
\[
\lambda bb \rightarrow \text{ case } mb \text{ of} \begin{align*}
\text{Just } (b, b') & \mapsto \\
\text{fmap } (\text{mput } (f \times L g) \text{ Nothing}) & \mapsto \\
\text{Nothing} & \mapsto \text{fmap } (\text{mput } (f \times L g) \text{ Nothing})
\end{align*}
\]
\[
\lambda bb \rightarrow \text{ case } mb \text{ of} \begin{align*}
\text{Just } (b, b') & \mapsto \\
\text{fmap } (\text{mput } (f \times L g) \text{ Nothing}) & \mapsto \\
\text{Nothing} & \mapsto \text{fmap } (\text{mput } (f \times L g) \text{ Nothing})
\end{align*}
\]
\[
\lambda bb \rightarrow \text{ case } mb \text{ of} \begin{align*}
\text{Just } (b, b') & \mapsto \\
\text{fmap } (\text{mput } (f \times L g) \text{ Nothing}) & \mapsto \\
\text{Nothing} & \mapsto \text{fmap } (\text{mput } (f \times L g) \text{ Nothing})
\end{align*}
\]
\[
\lambda bb \rightarrow \text{ case } mb \text{ of} \begin{align*}
\text{Just } (b, b') & \mapsto \\
\text{fmap } (\text{mput } (f \times L g) \text{ Nothing}) & \mapsto \\
\text{Nothing} & \mapsto \text{fmap } (\text{mput } (f \times L g) \text{ Nothing})
\end{align*}
\]
\[
\lambda bb \rightarrow \text{ case } mb \text{ of} \begin{align*}
\text{Just } (b, b') & \mapsto \\
\text{fmap } (\text{mput } (f \times L g) \text{ Nothing}) & \mapsto \\
\text{Nothing} & \mapsto \text{fmap } (\text{mput } (f \times L g) \text{ Nothing})
\end{align*}
\]
\[ \lambda mxyz \rightarrow \text{case } mxyz \text{ of} \]
\[ \quad \text{Just } ((x, y), z) \rightarrow \]
\[ \quad \quad \text{fmap } (\text{put } m (\text{Just } ((x, y), z))) \]
\[ \quad \quad ((u \star (v \star w)) \text{ Just } m ((x, y), z)) \]
\[ \quad \text{Nothing} \rightarrow \]
\[ \quad \quad \text{fmap } (\text{put } m \text{ Nothing}) \]
\[ \quad \quad ((u \star (v \star w)) \text{ Nothing}) \]
\[ = \lambda mxyz \rightarrow \text{case } mxyz \text{ of} \]
\[ \quad \text{Just } ((x, y), z) \rightarrow ((u \star v) \star w) \text{ Just } m ((x, y), z) \]
\[ \quad \text{Nothing} \rightarrow ((u \star v) \star w) \text{ Nothing} \]
\[ = \lambda mxyz \rightarrow ((u \star v) \star w) mxyz \]
\[ = (u \star v) \star w \]

To conclude,