Electronic Communications of the EASST

BX 2013

# Lenses for Web Data

Raghu Rajkumar, Sam Lindley, Nate Foster, James Cheney

21 pages

Perdita Stevens and James Terwilliger
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

# Lenses for Web Data

**Raghu Rajkumar**[1], **Sam Lindley**[2], **Nate Foster**[1], **James Cheney**[3]

[1] Cornell University
[2] University of Strathclyde
[3] University of Edinburgh

**Abstract:** Putting data on the web usually involves implementing two transformations: one to convert the data into HTML, and another to parse modifications out of interactions with clients. Unfortunately, in current systems, these transformations are typically implemented using two separate functions—an approach that replicates functionality across multiple pieces of code, and makes programs difficult to write, reason about, and maintain. This paper presents a different approach: an abstraction based on *formlets* that makes it easy to bridge the gap between data stored on a server and values embedded in HTML forms. We introduce *formlenses*, which combine the advantages of formlets with those of lenses to provide compositional, bidirectional form-based views of Web data. We show that formlenses can be viewed as monoidal functors over lenses, analogously to formlets, which are applicative functors. Finally, we investigate the connection between linearity and bidirectional transformations and describe a translation from a linear pattern syntax into formlens combinators.

**Keywords:** Formlets, lenses, applicative functors, monoidal functors.

## 1 Introduction

Putting data on the web usually involves implementing two transformations: one to convert the data into HTML, and another to parse modifications out of interactions with clients. Unfortunately, in current systems, these transformations are typically implemented using two separate functions—an approach that replicates functionality across multiple pieces of code, and makes programs difficult to write, reason about, and maintain.

To illustrate, suppose that we have a database of visiting speakers and we want to build an application that supports viewing and editing this database through a Web browser. Figure 1 gives a possible implementation of this application in Haskell. The *render* function converts a single *Speaker* value into HTML with an embedded form, which allows a user to edit the details associated with the speaker and submit the modifications back to the server. The function *collect* handles these responses by extracting an updated *Speaker* value out of the association list *Env*, which is returned to the server by the browser.

Readers unfamiliar with Haskell language can find documentation for the functions used in this paper at http://www.haskell.org/hoogle. This example uses several functions based on the *Text.Html* module and the standard prelude: *inputTag* constructs an input element, *brTag* constructs a linebreak, *lineToHtml* converts a string to HTML, $(+\!\!+\!\!+)$ concatenates HTML, *lookup* retrieves a value from an association list and returns a *Maybe* value, *fromJust* extracts the encap-

```
data Date    = Date {month :: Int, day :: Int}
data Speaker = Speaker {name :: String, date :: Date}
render :: Speaker → Html
render (Speaker n (Date m d)) =
  inputTag {name = "name", value = n} ++ brTag ++
  lineToHtml "Month:  " ++ inputTag {name = "month", value = show m} ++ brTag ++
  lineToHtml "Day:  "   ++ inputTag {name = "day", value = show d}
data Env = [(String, String)]
collect   :: Env → Speaker
collect e = let n  = fromJust (lookup "name" e) in
            let m = read (fromJust (lookup "month" e)) in
            let d  = read (fromJust (lookup "day" e)) in
            Speaker n (Date m d)
```

Figure 1: Haskell code for Speaker example.

sulated value from a *Maybe* value, *show* converts an *Int* to a *String*, and *read* parses an *Int* from a *String*.

Together, the *render* and *collect* functions effectively present a single *Speaker* value on the Web. But in general, developing applications this way quickly leads to complications: First, it requires the programmer to explicitly coerce the data to and from HTML—something that is easy to get wrong, especially in larger examples. Second, it requires them to construct the form manually, including choosing names for each form field. Although these names are semantically immaterial they must be globally unique to avoid clashes. Moreover, the field names used by *render* must be synchronized with the names used by *collect*. These constraints make it difficult to construct forms in a compositional manner. For example, we cannot iterate *render* and *collect* to obtain a program for a list of *Speaker*s because specific field names are "baked into" the code.

**Formlets.** In prior work, Cooper et al. [CLWY08] introduced a high-level abstraction for building Web forms called *formlets*. Formlets encapsulate several low-level details including selecting field names for elements and parsing data from client responses. Formlets can be represented in Haskell as functions that take a source of names—concretely, an *Int*—as an argument and produce a triple consisting of an HTML document, a *collect* function, and a modified namesource.

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

The names of any generated form fields are drawn from the namesource, and the *collect* function 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,

```
html     :: Html → Formlet ()
html h i = (h, λ_ → (), i)
```

the *text* combinator, which wraps plain text as HTML,

$$\begin{aligned}&\mathit{text} :: \ \mathit{String} \to \mathit{Formlet}\ ()\\ &\mathit{text}\ s = \mathit{html}\ (\mathit{stringToHtml}\ s)\end{aligned}$$

and the *inputInt* combinator, which builds a form that accepts an integer:

$$\begin{aligned}&\mathit{inputInt} \ :: \mathit{Formlet}\ \mathit{Int}\\ &\mathit{inputInt}\ i = \mathbf{let}\ n = \mathit{show}\ i\ \mathbf{in}\\ &\qquad\qquad (\mathit{inputTag}\ \{\mathit{name} = n, \mathit{value} = \texttt{""}\}, \lambda e \to \mathit{read}\ (\mathit{fromJust}\ (\mathit{lookup}\ n\ e)), i+1)\end{aligned}$$

Combinators for other primitive types such as *String* and *Bool* can be defined similarly.

Formlets can be combined into larger formlets using the interface of *applicative functors*, a generic mathematical structure for representing computations with effects [MP08]. We give the definition of applicative functors here using a *type class*—a description of the functions that each applicative functor instances must provide.

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

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

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

The applicative functor instance for formlets is defined as follows:

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

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

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

$$
\begin{array}{l}
\mathit{dateForm} :: \mathit{Formlet}\ \mathit{Date} \\
\mathit{dateForm} = \mathit{pure}\ (\lambda\_\ \_\ m\ \_\ \_\ d\ \_ \rightarrow \mathit{Date}\ m\ d) \\
\quad \otimes \mathit{text}\ \texttt{"Month: "} \otimes \mathit{inputInt} \otimes \mathit{html}\ \mathit{br} \\
\quad \otimes \mathit{text}\ \texttt{"Day: "} \quad\ \otimes \mathit{inputInt} \otimes \mathit{html}\ \mathit{br} \\
\mathit{speakerForm} :: \mathit{Formlet}\ \mathit{Speaker} \\
\mathit{speakerForm} = \mathit{pure}\ \mathit{Speaker} \otimes \mathit{inputString} \otimes \mathit{dateForm}
\end{array}
$$

The _ arguments in the anonymous function supplied to *pure* discard the () values produced by the *text* and *html* combinators.

**Formlenses.** Formlets are a useful abstraction, but they only address one half of the problem—they make it easy to construct a function that *produces* a value of type *a*, but they do not provide a way to describe a function that *consumes* a value of type *a* and embeds it in a form. Of course, programmers could write functions of type $a \rightarrow \mathit{Formlet}\ a$ (similar to the approach in Hanus' WUI library [Han06]), but such functions do not support the applicative functor interface (so large examples would need to be expressed using monolithic programs) and do not guarantee reasonable behavior (so programmers would have to prove that values are preserved on round-trips by hand).

A different way to think about this problem is to observe that forms are often used to present an *updatable view* of the 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 this terminology suggests, we have a solution to this problem in mind: use ideas from *bidirectional transformations* [CFH+09] to define formlets that behave like updatable views.

The main goal of this paper is to show how formlets and bidirectional transformations such as lenses [FGM+07] can be combined, yielding an abstraction called *Formlens*es. A formlens 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 web page that contains an editable *a*. But unlike manual approaches, where programmers write explicit *render* and *collect* functions, or the approach using functions of type $a \rightarrow \mathit{Formlet}\ a$ described above, we can design the formlens abstraction to support composition and strong semantic guarantees.

**Challenges.** Combining formlets and lenses turns out to be nontrivial. If a value of type *Formlens a* consumes *a* values, then *a* must appear in a negative position in its type. Conversely, if a value of type *Formlens a* produces *a* values, then *a* must appear positively in its type. These observations lead to serious problems defining *Formlens* as a type operator in Haskell: *Formlens* cannot be a covariant *Functor* over the category *Hask* of Haskell types and functions, and it cannot be a contravariant functor over this category either.

To avoid this difficulty, we shift perspective and consider functors from lenses to (the category of) Haskell types and functions. However, even after restricting attention to these functors, the applicative functor interface is still too strong. Consider the following interface, which generalizes McBride and Paterson's definition to allow applicative functors on lenses:

```
class LApplicative f where
  lpure :: a → f a
  lapp  :: f (Lens a b) → f a → f b
```

To instantiate *LApplicative Formlens*, we would need to define functions

```
lpure :: a → Formlens a
lapp  :: Formlens (Lens a b) → Formlens a → Formlens b
```

For the latter, we would need to (somehow) combine a form that consumes and produces a *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 possible to do in a natural way. Moreover, from a categorical perspective, *lapp* would not make sense because the category of lenses over Haskell types does not have exponential objects. This remains true if we consider contravariant functors.

Fortunately we can sidestep these problems by observing that the extra structure of *Applicative* functors is not required for *Formlet*s or *Formlens*es—monoidal structure suffices. By adapting the basic ideas of *Formlet*s to different (weaker) mathematical structures we are able to employ *Formlet*s in a broader range of settings. More specifically, we can compose formlets with lenses while retaining the ability to compose formlets using a monoidal interface.

**Contributions.** Overall, this paper makes the following contributions:

- We present a new foundation for formlets based on monoidal functors over lenses, which makes it possible to give them a bidirectional semantics;
- We describe an implementation of formlenses as a combinator library in Haskell;
- We explore the connection between linear syntax and bidirectional transformations and describe a translation from a language of linear patterns into formlens combinators.

The rest of the paper is structured as follows: Section 2 presents the design of formlenses by generalizing the classic definition based on formlets and showing that they are monoidal functors over lenses. Section 3 describes the formlens implementation, including syntactic sugar for describing formlenses using linear patterns. Section 4 briefly reviews related work. Section 5 concludes. The appendix presents background material on monoidal functors that may be helpful for some readers.

## 2 Formlens Design

This section defines formlenses, the main abstraction presented in this paper. We begin by reviewing the definition of monoidal functors as represented in Haskell. The built-in type class *Functor* models functors from *Hask* to *Hask*.

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

That is, instances of *Functor* are required to supply a function *fmap* that lifts functions from $a$ to $b$ to functions from $f\ a$ to $f\ b$. To allow formlenses to both produce and consume values, we will work with functors from *Lens* to *Hask*, where *Lens* is the category of (asymmetric) lenses,

$$\textbf{data}\ \mathit{Lens}\ a\ b = \mathit{Lens}\ \{\mathit{get} :: a \to b, \mathit{put} :: \mathit{Maybe}\ a \to b \to a\}$$

where *get* and *put* must satisfy:

$$\begin{array}{ll} \mathit{put}\ (\mathit{Just}\ a)\ (\mathit{get}\ a) = a & \text{-- GetPut} \\ \mathit{get}\ (\mathit{put}\ (\mathit{Just}\ a)\ b) = b & \text{-- PutGet} \\ \mathit{get}\ (\mathit{put}\ \mathit{Nothing}\ b) = b & \text{-- CreateGet} \end{array}$$

It is straightforward to show that *Lens* has an identity and is closed under composition, and therefore forms a category.

We model functors from *Lens* to *Hask* using the following type class:

$$\begin{array}{l} \textbf{class}\ \mathit{LFunctor}\ f\ \textbf{where} \\ \quad \mathit{lmap} :: \mathit{Lens}\ a\ b \to f\ b \to f\ a \end{array}$$

For technical reasons, we use the contravariant formulation: given a lens from $a$ to $b$, an *LFunctor f* maps an $f\ b$ to a $f\ a$. As we shall see below, this will allow us to use a *Lens a b* to transform a formlens on $b$ values into one on $a$ values.

The *Lens* category has monoidal structure, meaning that it supports the following operations:

$$\begin{array}{l} \textbf{class}\ \mathit{Category}\ c \Rightarrow \mathit{MonoidalCategory}\ c\ \textbf{where} \\ \quad (\cdot \times \cdot)\ \ :: c\ a_1\ b_1 \to c\ a_2\ b_2 \to c\ (a_1, a_2)\ (b_1, b_2) \\ \quad \mathit{munitl}\ \ :: \mathit{Iso}\ c\ ((), a)\ a \\ \quad \mathit{munitr}\ \ :: \mathit{Iso}\ c\ (a, ())\ a \\ \quad \mathit{massoc} :: \mathit{Iso}\ c\ (a, (b, d))\ ((a, b), d) \end{array}$$

Here, *Iso c a b* is simply a pair of maps $c\ a\ b$ and $c\ b\ a$ witnessing an isomorphism. Monoidal categories are also required to satisfy several additional laws [Mac98], which all hold for *Lens*.

Next, we consider monoidal functors[1] providing a unit and binary operations,

$$\begin{array}{l} \textbf{class}\ \mathit{Functor}\ f \Rightarrow \mathit{Monoidal}\ f\ \textbf{where} \\ \quad \mathit{unit} :: f\ () \\ \quad (\star)\ :: f\ a \to f\ b \to f\ (a, b) \end{array}$$

and satisfying the following laws:

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

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

Note that these definitions assume we are working in *Hask*, as well as operations $\mathit{fst} :: (a,b) \to a$ and $\mathit{snd} :: (a,b) \to b$, which are not available in all monoidal categories. Since we are interested in functors from *Lens* to *Hask*, a minor variant (adapted from MacLane [Mac98]) suffices:

$$\begin{array}{lr@{\;=\;}l}
\text{(M1)} & \mathit{lmap}\ (f \times g)\ (u \star v) & \mathit{lmap}\ f\ u \star \mathit{lmap}\ g\ u \\
\text{(M2)} & \mathit{lmap}\ \mathit{munitl}\ (\mathit{unit} \star u) & u \\
\text{(M3)} & \mathit{lmap}\ \mathit{munitr}\ (u \star \mathit{unit}) & u \\
\text{(M4)} & \mathit{lmap}\ \mathit{massoc}\ (u \star (v \star w)) & (u \star v) \star w
\end{array}$$

We are now in a position to define the *Formlens* type and show that it is a monoidal functor:

```
type Formlens a = Maybe a → [Int] → (Html, Env → Maybe a, [Int])
```

The differences between this definition and the standard definition for formlets are relatively minor: we have added an extra *Maybe a* parameter that provides an optional initial value of the form, changed the namesource to a list of integers, and allowed the *collect* function to return an optional value. However, the type variable $a$ appears both covariantly and contravariantly, which means that *Formlens* cannot be an ordinary Haskell functor, let alone an applicative functor. Hence, we are forced to consider formlenses as functors on *Lens*, whose arrows reflect their bidirectionality. But then since *Lens* is not closed, formlenses cannot be applicative functors. Fortunately, we will be able to show they are monoidal functors, which are weaker than applicative functors but still provide an interface that supports composition of formlenses.

As a first step, we show that *Formlens*es are lens functors:

```
instance LFunctor Formlens where
  lmap lns f = Formlens (λv i → let (html, c, i′) = f (fmap (get l) v) i
                                in (html, fmap (put l v) ∘ c, i′)
```

When we map a lens $\mathit{lns} :: \mathit{Lens}\ a\ b$ over a formlens $f :: \mathit{Formlens}\ b$, to obtain a *Formlens a*, we render $f$ using the lens's *get* function (to transform the $a$ value to a $b$), then we post-compose the function $c$ with the lens's *put* function applied to the original $a$. Note that this construction would not work with covariant *LFunctor*s, because given $\mathit{lns} :: \mathit{Lens}\ a\ b$ and $f :: \mathit{Formlens}\ a$, and a $b$ value, there would be no way to use *lns*'s *put* function to construct an appropriate $a$ value to use as argument to $f$.

Next we show that *Formlens* admits monoidal operations:

```
instance Monoidal Formlens where
  unit  = Formlens (λ_ i → (noHtml, λ_ → Just (), i))
  f ⋆ g = Formlens (λv i → let (a, b) = split v in
                           let (ha, ca, i′) = f a i
                               (hb, cb, i″) = g b i′
                           in (h1 ++ h2, λe → liftM2 (,) (ca e) (cb e), i″)
    where split v = case v of Just (a, b) → (Just a, Just b)
                              _           → (Nothing, Nothing)
```

That is, *unit* generates no HTML, allocates no names, and collects (), whereas $f \star g$ combines two formlenses that separately handle an $a$ and a $b$ to obtain one that handles a pair of $a$ and $b$

by concatenating the rendered HTML and using the two *collect* functions to form the result. The following theorem states that this definition satisfies the monoidal functor laws:

**Theorem 1** *Formlens is a monoidal LFunctor.*

**Example.** To get a taste for formlenses, let us build a bidirectional version of the *dateForm* formlet from the introduction starting with formlens versions of *html*, *text*, and *inputInt*:

$$
\begin{array}{l}
\mathit{htmlL} :: \mathit{Html} \to \mathit{Formlens}\ ()\\
\mathit{htmlL}\ h = \mathit{Formlens}\ (\lambda\_\ i \to (h, \lambda\_ \to \mathit{Just}\ (), i))\\
\mathit{textL} :: \mathit{String} \to \mathit{Formlens}\ ()\\
\mathit{textL}\ s = \mathit{htmlL}\ (\mathit{stringToHtml}\ s)\\
\mathit{inputIntL} :: \mathit{Formlens}\ \mathit{Int}\\
\mathit{inputIntL} = \mathit{Formlens}\ (\lambda v\ i@(h:t) \to\\
\quad \mathbf{let}\ n = \mathit{intercalate}\ \texttt{"\_"}\ (\mathit{map}\ \mathit{show}\ i)\ \mathbf{in}\\
\quad (\mathit{inputTag}\ \{\mathit{name} = n, \mathit{value} = (\mathit{maybe}\ \texttt{""}\ \mathit{show}\ v)\},\\
\quad\ \lambda e \to \mathit{fmap}\ \mathit{read}\ (\mathit{lookup}\ n\ e),\\
\quad\ (h+1):t)
\end{array}
$$

Note that the *collect* functions for these combinators ignore the initial value, if any—*i.e.*, the formlens is essentially bijective. The only situation where this does not happen is when we *lmap* a lens over a formlens. In this case, only a projection of the initial value is displayed on the form, and the part that was projected out is restored from the initial value when the form is collected.

Next we define two helper operators, $(\langle\star)$ and $(\star\rangle)$, which streamline definitions where we combine formlenses on $()$ values. These operators behave like $(\star)$ but combine a *Formlens a* and *Formlens* $()$ into a *Formlens a*, rather than a *Formlens* $(a,())$ and *Formlens* $((),a)$ respectively.

$$
\begin{array}{l}
(\langle\star) :: (\mathit{Monoidal}\ f, \mathit{LFunctor}\ f) \Rightarrow f\ a \to f\ () \to f\ a\\
f\ \langle\star\ g = \mathit{lmap}\ (\mathit{munitr} :: \mathit{Iso}\ \mathit{Lens}\ (a,())\ a)\ (f \star g)
\end{array}
$$

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

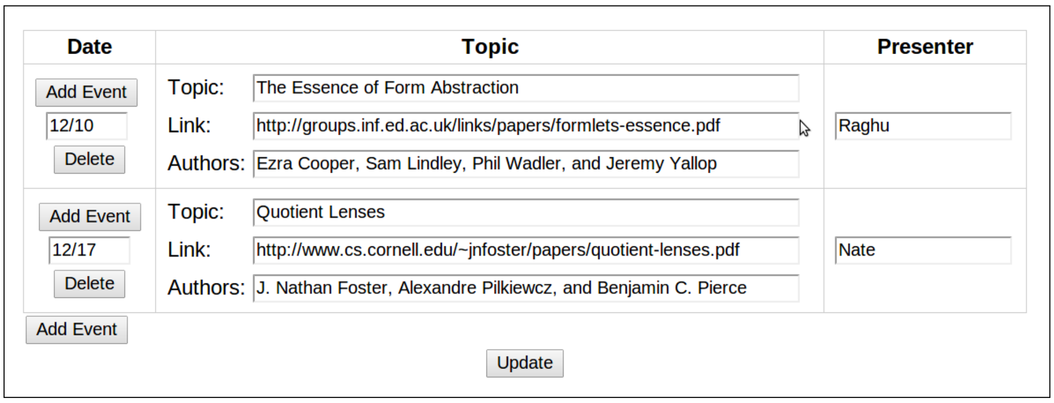
Next, we define a lens on *Date* values:

$$
\begin{array}{l}
\mathit{dateL}\ ::\ \mathit{Lens}\ (\mathit{Int},\mathit{Int})\ \mathit{Date}\\
\mathit{dateL} = \mathit{Lens}\ (\lambda(m,d) \to \mathit{Date}\ m\ d)\ (\lambda\_\ (\mathit{Date}\ m\ d) \to (m,d))
\end{array}
$$

Finally, putting all these pieces together, we define a formlens for *Date*s as follows:

$$
\begin{array}{l}
\mathit{dateFormlens} :: \mathit{Formlens}\ \mathit{Date}\\
\mathit{dateFormlens} = \mathit{lmap}\ \mathit{dateL}\ (\mathit{textL}\ \texttt{"Month: "} \star \mathit{inputIntL}\ \langle\star\ \mathit{htmlL}\ \mathit{brTag}\ \langle\star\\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{textL}\ \texttt{"Day: "} \quad \star \mathit{inputIntL}\ \langle\star\ \mathit{htmlL}\ \mathit{brTag})
\end{array}
$$

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



| Date | Topic | | Presenter |
|---|---|---|---|
| Add Event | Topic: | The Essence of Form Abstraction | |
| 12/10 | Link: | http://groups.inf.ed.ac.uk/links/papers/formlets-essence.pdf | Raghu |
| Delete | Authors: | Ezra Cooper, Sam Lindley, Phil Wadler, and Jeremy Yallop | |
| Add Event | Topic: | Quotient Lenses | |
| 12/17 | Link: | http://www.cs.cornell.edu/~jnfoster/papers/quotient-lenses.pdf | Nate |
| Delete | Authors: | J. Nathan Foster, Alexandre Pilkiewcz, and Benjamin C. Pierce | |
| Add Event | | | |

Figure 2: Screenshot of *Speakers* formlens running in a browser.

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

To answer this question positively, we must restrict our attention to formlenses that satisfy certain well-formedness properties: a formlens should only draw names from the namesource provided as an argument, and the *collect* function should only look up corresponding names.

Let *extract* be a function from *Html* to *Env* that extracts an environment containing the names and values of all form fields from an HTML document. Also, for every formlens $f$ and namesources $n, n'$, define an equivalence relation on *Env* values as follows:

$$e \sim_f^{n,n'} e' \overset{\Delta}{\iff} \forall v :: Maybe\ a.\ (\_, c, \_) = f\ v\ n \text{ and } (\_, c', \_) = f\ v\ n' \text{ implies } c\ e = c'\ e'$$

We say that a formlens $f$ is well behaved if it satisfies the following properties, which are analogous to GETPUT and PUTGET (the property for CREATEGET is similar):

**Definition 1** (Acceptability) A formlens $f :: Formlens\ a$ is *acceptable* if for all $v :: a$, $n :: Int$, and $e :: Env$, if $(h, c, \_) = f\ (Just\ v)\ n$ and $extract\ h \subseteq e$, then $c\ e = Just\ v$.

**Definition 2** (Consistency) A formlens $f :: Formlens\ a$ is *consistent* if for all $v :: a$, $v' :: Maybe\ a$, $n, n' :: Int$, and $e :: Env$, if $(\_, c, \_) = f\ v'\ n'$ and $c\ e = (Just\ v)$ and $(h, \_, \_) = f\ (Just\ v)\ n$, then $extract\ h \sim_f^{n,n'} e$.

Readers familiar with lenses may note that these properties are essentially the quotient lens laws [FPP08]. Each of the primitive formlenses described in the next section satisfies these laws, and each of the formlens combinators preserves them.

## 3 Implementation

We have developed an implementation of formlenses in Haskell. This implementation provides a collection of formlens primitives and a translation from a higher-level syntax based on linear

Meta variables

| | | | | | |
|---|---|---|---|---|---|
| $e$ | Haskell expression | $s$ | string literal | $t$ | HTML tag |
| $f$ | Formlens expression | $p$ | Haskell pattern | $atts$ | HTML attribute list |

Syntax

$$\begin{array}{rcll} l & ::= & [\textbf{formlens} \mid p \leftrightarrow ns \mid] & \text{(Formlens)} \\ n & ::= & \text{<}t\ atts\text{>}ns\text{</}t\text{>} \mid s \mid \{e\} \mid \{f \to p\} & \text{(HTML node)} \\ ns & ::= & n_1\ \ldots\ n_k & \text{(HTML node sequence)} \end{array}$$

Figure 3: Linear pattern syntax.

patterns into these primitives, and it runs on top of the Happstack Web server [Hap12].

**Combinator library.** Our formlens combinator library provides a rich collection of formlens primitives and formlens combinators that allow a wide variety of data to be presented on the Web. The library includes primitives for building forms containing integers, strings, booleans, checkboxes, radio groups, buttons, and other values as HTML. It also includes combinators for combining smaller formlenses into larger ones. Most important are the following:

$$\begin{array}{l} \star :: \mathit{Formlens}\ a \to \mathit{Formlens}\ b \to \mathit{Formlens}\ (a,b) \\ \oplus :: \mathit{Formlens}\ a \to \mathit{Formlens}\ b \to \mathit{Formlens}\ (\mathit{Either}\ a\ b) \\ \mathit{iter} :: \mathit{Formlens}\ a \to \mathit{Formlens}\ [a] \end{array}$$

The $\star$ operator composes formlenses "horizontally" using the monoidal functor interface discussed in the previous section. The $\oplus$ operator behaves like a conditional operator on formlenses that branches on the *Left* and *Right* tags attached to *Either* values. Finally, the *iter* operator iterates a lens over a list of values. This operator turns out to be especially interesting due to the way it uses namesources—to allow in-place modifications to the list on the HTML side, it is convenient to make the namesource a list of integers rather than a single integer. The definitions of the $\oplus$ and *iter* combinators are presented in appendix B.

To showcase the use of these combinators, we have built a website for viewing and editing a database of *Speaker*s as shown in Figure 2. The main functionality of this formlens is given by applying *iter* to the formlens for *Speaker* values. The formlens also introduces appropriate HTML rendering and uses a JavaScript library to allow the *collect* function to be invoked through an AJAX interface instead of (slower) POST requests. In addition, the list allows elements to be added or removed from any location, and also edited in place.

**Linear patterns.** Writing programs using combinators is extremely tedious because the structure of the values produced (and consumed) by the formlens closely matches the structure of the combinator program itself. For example, if $f$ is a *Formlens Int* and $g$ is a *Formlens* $()$ then $f \star g$ is a *Formlens* $(\mathit{Int},())$. If the programmer wishes to obtain a *Formlens Int* instead, they must *lmap*

$$\begin{array}{rcl@{\qquad}rcl}
s^{\circ} & = & \mathit{textL}\ s & s^{\dagger} & = & () \\
\{e\}^{\circ} & = & \mathit{htmlL}\ e & \{e\}^{\dagger} & = & () \\
\{f \to p\}^{\circ} & = & f & \{f \to p\}^{\dagger} & = & p \\
\texttt{<}t\ \mathit{atts}\texttt{>}\mathit{ns}\texttt{</}t\texttt{>}^{\circ} & = & \mathit{tagL}\ t\ \mathit{atts}\ \mathit{ns}^{\circ} & \texttt{<}t\ \mathit{atts}\texttt{>}\mathit{ns}\texttt{</}t\texttt{>}^{\dagger} & = & (\mathit{ns})^{\dagger} \\
(n_1\ \ldots\ n_k)^{\circ} & = & {n_1}^{\circ} \star \ldots \star {n_k}^{\circ} & (n_1\ \ldots\ n_k)^{\dagger} & = & (n_1^{\dagger}, \ldots, n_k^{\dagger})
\end{array}$$

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

$$\begin{array}{rcl}
[\mathbf{formlens} \mid p \leftrightarrow \mathit{ns} \mid] & = & \mathit{lmap}\ (\mathit{Lens}\ \mathit{get}\ \mathit{put})\ (\mathit{ns})^{\circ} \\
 & & \quad \text{where } (p_{old}, p_{new}, p_{create}) = p^{\prec} \\
 & & \qquad \mathit{get} = \lambda p \to (\mathit{ns})^{\dagger} \\
 & & \qquad \mathit{put} = \lambda e\ (\mathit{ns})^{\dagger} \to \\
 & & \qquad\ \ \mathbf{case}\ e\ \mathbf{of} \\
 & & \qquad\quad \mathit{Just}\ (p_{old}) \to p_{new} \\
 & & \qquad\quad \mathit{Nothing}\ \ \to p_{create}
\end{array}$$

Figure 4: Linear pattern translation.

the bijection *munitr* on the result to eliminate the spurious $()$. Of course, the derived $\langle\star$ operator does this, but having to remember when to use $\star$ versus $\langle\star$ and $\star\rangle$ is inconvenient.

This section describes 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. Our translation extends previous work by Cooper et al. [CLWY08] on translating formlet syntax into combinators using applicative functors. We have implemented this syntax using Haskell's quasi-quotation features [Mai07] and Template Haskell [SJ02].

To describe a formlens using the high-level syntax, the programmer defines a pair of patterns over a shared set of variables. The pattern on the left matches Haskell values, while the pattern on the right matches HTML. When read from left to right, programs written in this way denote transformations from values to HTML forms; when read from right to left, they denote transformations from form responses back to values.

Formally, we define a formlens using the syntax $[\mid \mathbf{formlens} \mid p \leftrightarrow n_1\ \ldots\ n_k \mid]$, where $p$ is a Haskell pattern and *ns* is a sequence of elements $\texttt{<}t\ \mathit{atts}\texttt{>}n_1\ \ldots\ n_k\texttt{</}t\texttt{>}$, text nodes $s$, spliced HTML expressions $\{e\}$, or nested formlens bindings $\{f \to p\}$. The complete syntax is given in Figure 3. The pattern $p$ may bind some of the variables referenced in *ns* and ignore others (by using the wildcard symbol _). However, to ensure that the overall formlens is well-defined, the pattern must be *linear*—variables mentioned in the pattern and any sub-formlenses must be used exactly once on the other side.

Figure 4 defines the translation from the high-level syntax into combinators. The translation of the body using $(-)^{\circ}$ goes by structural recursion, producing formlens combinators such as *textL*, *htmlL*, and *tagL* as results. Sequences of nodes are combined using the $(\star)$ operator and *unit*, which handles the empty sequence. On the other side, we extract patterns from the form using $(-)^{\dagger}$, again following a straightforward structural recursion. Next, the lens functor map operation *lmap* is used to combine the results of the sub-lens bound in the body through extracted patterns. Finally, we construct a lens to pre-compose with the generated form, which means we need to construct an appropriate *put* function. As we are now mapping a lens over the formlens, we must construct a *put* function taking an extra argument representing the original input value. The pattern $p_{old}$ binds the parts of the original input value that are ignored by the formlens. The pattern $p_{new}$ (which is also an expression as it contains no wildcards) combines the ignored part of the input bound by $p_{old}$ with the output produced by the formlens. 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 the lens we construct is essentially a (partial) bijection—the linearity constraint guarantees that the form data is in bijective correspondence with the pattern data.

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 date of a speaker, while maintaining the name, then we can write the following code,

$$
\begin{aligned}
&\mathit{speakerFormL} :: \mathit{Formlens\ Speaker}\\
&\mathit{speakerFormL} = [\textbf{formlens} \mid \mathit{Speaker}\ \_\ d \leftrightarrow\\
&\qquad\qquad\qquad\qquad \mathit{Name} : \{\mathit{stringToHtml\ name}\} < br / >\\
&\qquad\qquad\qquad\qquad \{\mathit{dateFormL} \rightarrow d\}\ |]
\end{aligned}
$$

which translates into,

$$
\begin{aligned}
\mathit{speakerFormL} = \mathit{lmap}\ (&\mathit{Lens}\ (\lambda(\mathit{Speaker}\ \_\ d) \rightarrow ((),((),((),d))))\\
&\qquad (\lambda p0\ ((),((),((),d))) \rightarrow \textbf{case}\ p0\ \textbf{of}\\
&\qquad\quad \mathit{Just}\ (\mathit{Speaker}\ x_0\ \_) \rightarrow \mathit{Speaker}\ x_0\ d\\
&\qquad\quad \mathit{Nothing} \qquad\qquad\quad \rightarrow \mathit{Speaker}\ \bot\ d))\\
&(\mathit{textL}\ \texttt{"Name: "} \star \mathit{htmlL}\ (\mathit{stringToHtml\ name})\\
&\quad \star \mathit{tag}\ \texttt{"br"}\ [\,]\ [\,] \star \mathit{dateFormL})
\end{aligned}
$$

where $x_0$ is a fresh variable that tracks the old name value. The translation handles the tedious tasks of generating the boilerplate lens code to map from the underlying type $((),((),((),\mathit{Date})))$ of the formlens body and generating *textL* and *htmlL* combinators representing the HTML form components themselves. Overall, programs written using the high-level syntax are both more concise and easier to maintain than the generated code.

## 4 Related work

**Formlets.** There is a variety of earlier work on declarative abstractions for web forms. Cooper et al. [CLWY08] first proposed expressing formlets as the composition of several primitive applicative functors, and gives a detailed comparison to prior work. The abstraction in Hanus's

WUI (Web User Interface) library [Han06] has several similarities to the formlenses. In particular, WUI includes a combinator for lifting a bijection to WUIs, similar to our *LFunctor Formlens* instance (restricted to bijections); however, Hanus did not consider *Applicative* or *Monoidal* equational laws on WUIs, nor constructing them as functors over lenses.

Formlets were originally developed as part of Links [CLWY07]. Formlet libraries have since been implemented for Haskell, F#, Scala, OCaml, Racket, and Javascript. Eidhof's Haskell library [Eid08] evolved first into digestive functors [dJ12], and most recently the reform package [SJ12]. The latter integrates with various other libraries, and extends formlets with better support for validation and for separating layout from formlet structure. The F# WebSharper library [BTG11] introduces *flowlets*, which combine formlets with functional reactive programming [EH97] allowing forms to change dynamically at run-time.

**Bidirectional transformations.** Languages for describing bidirectional transformations have been extensively studied in recent years; a recent Dagstuhl seminar report [HSST11] presents a comprehensive survey. The original paper on lenses [FGM$^+$07] describes work on databases and programming languages; another more recent survey also discusses work from the software engineering literature [CFH$^+$09]. The XSugar [BMS08] language defines bidirectional transformations between XML documents and strings. Similarly, the biXid [KH06] 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. [RO10]. 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, while allowing for a convenient high-level syntax similar to that for formlets.

**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 [SD96] for parser combinators, and named and recognized as a lighter-weight alternative to monads by McBride and Paterson [MP08]. The relationships among monads, arrows, and applicative functors were further elucidated by Lindley et al. [LWY11]. The connection to monoidal functors was explored further by Paterson [Pat12], who also observes that it is often much easier to work with monoidal functors. However, Paterson focuses attention on functors on cartesian closed categories and *Lens* is not closed.

Functors on categories other than *Hask* have appeared in other contexts. Functors over isomorphisms are used in the fclabels library [VHEV12]. However, fclabels also provides an applicative functor interface, which can be used to produce intuitively incorrect bidirectional transformations. Similarly, functors over partial isomorphisms from *Iso* to *Hask* are essential in Rendel and Ostermann's invertible syntax descriptions [RO10]. They also employ a variant of *Monoidal* functors (which they call *ProductFunctor*s). A natural question for further work is whether *Monoidal* functors over partial isomorphisms suffice for invertible syntax descriptions, so that one can easily compose parser or pretty-printer combinators with formlenses. To the best of our knowledge, we are the first to use (monoidal) functors over lenses for programming.

## 5 Conclusion

Formlenses combine the features of formlets and lenses in a powerful abstraction that makes it easy for programmers to present data on the web. Our work on formlenses is ongoing. In the future, we plan to further develop the semantic properties of formlenses, investigate additional combinators and generic programming techniques for formlenses, and examine other formalisms for Web interaction, such as flowlets [BTG11]. We also plan to investigate ways of interacting with browsers that are not based on forms such as 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.

**Acknowledgements:** The authors wish to thank Arjun Guha, James McKinna, Ross Tate, Philip Wadler, Daniel Wagner, the Cornell Database Group, and the BX '13 reviewers for helpful comments. This work is supported in part by a Google Research Award (Lindley), EPSRC grant EP/J014591/1 (Lindley), National Science Foundation grant CCF-1253165 (Rajkumar and Foster), a Royal Society University Research Fellowship (Cheney).

## Bibliography

[Bie95] G. M. Bierman. What is a Categorical Model of Intuitionistic Linear Logic? In *TLCA*. Pp. 78–93. 1995.

[BMS08] C. Brabrand, A. Møller, M. I. Schwartzbach. Dual Syntax for XML Languages. *Information Systems* 33(4–5):385–406, 2008. Short version in DBPL '05.

[BTG11] J. Bjornson, A. Tayanovskyy, A. Granicz. Composing Reactive GUIs in F# Using WebSharper. In *IFL*. LNCS 6647, pp. 203–216. Springer, 2011.

[CFH$^+$09] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, J. F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. GRACE Meeting notes, State of the Art, and Outlook. In *ICMT*. Pp. 260–283. June 2009.

[CLWY07] E. Cooper, S. Lindley, P. Wadler, J. Yallop. Links: Web Programming Without Tiers. In *FMCO*. Pp. 266–296. 2007.

[CLWY08] E. Cooper, S. Lindley, P. Wadler, J. Yallop. The Essence of Form Abstraction. In *APLAS*. Pp. 205–220. Springer, 2008.

[EH97] C. Elliott, P. Hudak. Functional Reactive Animation. In *ICFP*. Pp. 263–273. 1997.

[Eid08] C. Eidhof. Formlets in Haskell. 2008. Available at http://blog.tupil.com/formlets-in-haskell.

[FGM$^+$07] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* 29(3), May 2007.

[FPP08] J. N. Foster, A. Pilkiewcz, B. C. Pierce. Quotient Lenses. In *ICFP*. Pp. 383–395. Sept. 2008.

[Han06] M. Hanus. Type-oriented construction of web user interfaces. In *PPDP*. Pp. 27–38. 2006.

[Hap12] The Happstack web server. 2012. Available from http://happstack.com.

[HSST11] Z. Hu, A. Schürr, P. Stevens, J. Terwilliger. Bidirectional Transformation “bx”. *Dagstuhl Reports* 1(1):42–67, 2011. Available at http://drops.dagstuhl.de/opus/volltexte/2011/3144.

[dJ12] J. V. der Jeugt. The digestive-functors package. 2012. http://hackage.haskell.org/package/digestive-functors.

[KH06] S. Kawanaka, H. Hosoya. biXid: a bidirectional transformation language for XML. In *ICFP*. Pp. 201–214. Sept. 2006.

[KW12] E. Kmett, D. Wagner. category-extras metapackage version 1.0.2. 2012. Available at http://hackage.haskell.org/package/category-extras.

[LWY11] S. Lindley, P. Wadler, J. Yallop. Idioms are Oblivious, Arrows are Meticulous, Monads are Promiscuous. *Electron. Notes Theor. Comput. Sci.* 229(5):97–117, Mar. 2011.

[Mac98] S. MacLane. *Categories for the working mathematician (second edition)*. Springer-Verlag, 1998.

[Mai07] G. Mainland. Why it’s nice to be quoted: Quasiquoting for Haskell. In *Haskell*. Pp. 73–82. 2007.

[MP08] C. McBride, R. Paterson. Applicative Programming with Effects. *Journal of Functional Programming* 18(1):1–13, Jan. 2008.

[Pat12] R. Paterson. Constructing Applicative Functors. In *MPC*. LNCS 7342, pp. 300–323. Springer, 2012.

[RO10] T. Rendel, K. Ostermann. Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing. In *Haskell*. Pp. 1–12. 2010.

[SD96] S. D. Swierstra, L. Duponcheel. Deterministic, Error-Correcting Combinator Parsers. In *Second International School on Advanced Functional Programming*. LNCS 1129, pp. 184–207. Springer, 1996.

[SJ02] T. Sheard, S. P. Jones. Template metaprogramming for Haskell. In *Haskell*. Pp. 1–16. 2002.

[SJ12] J. Shaw, J. V. der Jeugt. The reform package. 2012. Available at http://hackage.haskell.org/package/reform-0.1.1.

[VHEV12] S. Visser, E. Hesselink, C. Eidhof, S. Visscher. The fclabels package, version 1.1.3. May 2012. Available at http://hackage.haskell.org/package/fclabels.

[Vis11] S. Visscher. data-category package version 0.4.1. 2011. Available at http://hackage.haskell.org/package/data-category.

## A Monoidal categories and functors

The key to our approach is to use monoidal rather than applicative functors to define formlenses. This section reviews the (mostly standard) definitions for categories and monoidal functors.

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

```
class Category c where
  id  :: c x x
  (∘) :: c y z → c x y → c x z
```

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

```
instance Category (→) where
  id    = λx → x
  f ∘ g = λx → f (g x)
instance Category Bij where
  id    = Bij id id
  f ∘ g = Bij (fwd f ∘ fwd g) (bwd g ∘ bwd f)
instance Category Lens where
  id    = Lens id (const $ id)
  l ∘ m = Lens (get l ∘ get m)
               (λa c → put m a (put l (get m a) c))
```

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

```
data Iso c a b = Iso {fwdI :: c a b, bwdI :: c b a}
```

Isomorphisms are expected to satisfy the following condition:

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

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

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

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

$$
\begin{aligned}
&\mathit{munitr} :: \mathit{Iso}\ c\ (a,())\ a\\
&\mathit{massoc} :: \mathit{Iso}\ c\ (a,(b,d))\ ((a,b),d)
\end{aligned}
$$

Monoidal categories are expected to satisfy a number of additional laws, which essentially state that $()$ is a unit with respect to $(,)$ and "all diagrams involving the above operations commute" [Mac98]. 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 $(\to)$, with the following operations:

$$
\begin{aligned}
&\textbf{instance}\ \mathit{MonoidalCategory}\ (\to)\ \textbf{where}\\
&\quad f \times g \quad = \lambda(a,b) \to (f\ a, g\ b)\\
&\quad \mathit{munitl}\ = \mathit{Iso}\ (\lambda((),a) \to a)\ (\lambda a \to ((),a))\\
&\quad \mathit{munitr}\ = \mathit{Iso}\ (\lambda(a,()) \to a)\ (\lambda a \to (a,()))\\
&\quad \mathit{massoc} = \mathit{Iso}\ (\lambda(a,(b,d)) \to ((a,b),d))\\
&\qquad\qquad\qquad (\lambda((a,b),d) \to (a,(b,d)))
\end{aligned}
$$

In fact, any category with finite products is monoidal, but there are many monoidal categories whose monoidal product does not form a full Cartesian product. This is the case, for example, for bijections, since the *fst* and *snd* mappings are not bijections; models of linear type theory [Bie95] provide more examples.

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

$$
\begin{aligned}
&\mathit{iso2bij} \qquad\qquad\quad :: \mathit{Iso}\ (\to)\ a\ b \to \mathit{Iso}\ \mathit{Bij}\ a\ b\\
&\mathit{iso2bij}\ (\mathit{Iso}\ \mathit{to}\ \mathit{fro}) = \mathit{Iso}\ (\mathit{Bij}\ \mathit{to}\ \mathit{fro})\ (\mathit{Bij}\ \mathit{fro}\ \mathit{to})\\
&\textbf{instance}\ \mathit{MonoidalCategory}\ \mathit{Bij}\ \textbf{where}\\
&\quad f \times g \quad = \mathit{Bij}\ (\lambda(a,b) \to (\mathit{fwd}\ f\ a, \mathit{fwd}\ g\ b))\\
&\qquad\qquad\qquad (\lambda(\mathit{fa},\mathit{gb}) \to (\mathit{bwd}\ f\ \mathit{fa}, \mathit{bwd}\ g\ \mathit{gb}))\\
&\quad \mathit{munitl}\ = \mathit{iso2bij}\ \mathit{munitl}\\
&\quad \mathit{munitr}\ = \mathit{iso2bij}\ \mathit{munitr}\\
&\quad \mathit{massoc} = \mathit{iso2bij}\ \mathit{massoc}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{iso2lens} \qquad\qquad\quad :: \mathit{Iso}\ \mathit{Bij}\ a\ b \to \mathit{Iso}\ \mathit{Lens}\ a\ b\\
&\mathit{iso2lens}\ (\mathit{Iso}\ \mathit{to}\ \mathit{fro}) = \mathit{Iso}\ (\mathit{bij2lens}\ \mathit{to})\ (\mathit{bij2lens}\ \mathit{fro})
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ \mathit{MonoidalCategory}\ \mathit{Lens}\ \textbf{where}\\
&\quad l_1 \times l_2 \quad = l_1 \times_L l_2\\
&\quad \mathit{munitl}\ = \mathit{iso2lens}\ \mathit{munitl}\\
&\quad \mathit{munitr}\ = \mathit{iso2lens}\ \mathit{munitr}\\
&\quad \mathit{massoc} = \mathit{iso2lens}\ \mathit{massoc}
\end{aligned}
$$

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

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

The dual of any monoidal category is also monoidal:

```
$iso2dual$              :: $Iso$ $c$ $a$ $b$ → $Iso$ $c^{op}$ $a$ $b$
$iso2dual$ ($Iso$ $to$ $fro$) = $Iso$ ($Co$ $fro$) ($Co$ $to$)
instance $MonoidalCategory$ $c$ ⇒ $MonoidalCategory$ $c^{op}$ where
  ($Co$ $l_1$) × ($Co$ $l_2$) = $Co$ ($l_1 \times l_2$)
  $munitl$          = $iso2dual$ $munitl$
  $munitr$          = $iso2dual$ $munitr$
  $massoc$          = $iso2dual$ $massoc$
```

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

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

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

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

**Monoidal functors.** Next, we consider monoidal functors:

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

Note that we do not explicitly identify the domain of *f* in the type class *Monoidal f*; this is not necessary (and leads to typechecking complications due to the unconstrained type variable) since the signature of the operations of a monoidal functor depends only on the codomain category

[2] Others have proposed much more general libraries for categorical concepts in Haskell [KW12, Vis11]; 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.

(which for us is always *Hask*). However, in stating the laws for monoidal functors, we will implicitly assume that the domain is a monoidal category—in particular, that $\cdot \times \cdot$, *munitl*, *munitr*, and *massoc* are defined.

$$\begin{array}{ll}
\text{(M1)} & gmap\ (f \times g)\ (u \star v) = fmap\ f\ u \star fmap\ g\ u \\
\text{(M2)} & gmap\ munitl\ (unit \star u) = u \\
\text{(M3)} & gmap\ munitr\ (u \star unit) = u \\
\text{(M4)} & gmap\ massoc\ (u \star (v \star w)) = (u \star v) \star w
\end{array}$$

## B Formlens Combinators : $\oplus$ and *iter*

This section presents definitions of the $\oplus$ and *iter* formlens combinators in Haskell. We first define several helper functions:

```
    makeName :: [Int] → String
    makeName l = intercalate "_" (map show l)

    readName :: String → [Int]
    readName s = map read (split "_" s)

    next :: [Int] → [Int]
    next (hd : tl) = (hd + 1) : tl

    branch :: [Int] → [Int]
    branch l = 0 : l
```

Next we define the $\oplus$ combinator:

```
    (⊕) :: Formlens a → Formlens b → Formlens (Either a b)
    (f ⊕ g) v t =
      let s = makeName t
          t′ = next t in
      let collect ca cb e = case lookup s e of
            Just "left"   → fmap Left (ca e)
            Just "right"  → fmap Right (cb e)
            _             → Nothing in
      case v of
        Just (Left a)  → let (ha, ca, t1) = f a t′
                             (_, cb, t2)  = g Nothing t′ in
                         (hidden s "left" +++ ha, collect ca cb, max t1 t2)
        Just (Right b) → let (_, ca, t1)  = f Nothing t′ in
                             (hb, cb, t2) = g b t′ in
                         (hidden s "right" +++ hb, collect ca cb, max t1 t2)
        Nothing        → let (_, ca, t1)  = f Nothing t′ in
                             (_, cb, t2)  = g Nothing t′ in
                         (hidden s "nothing", collect ca cb, max t1 t2))
```

Finally we define the *iter* combinator:

```
$iter :: Formlens\ a \rightarrow Formlens\ [a]$
$iter\ f\ v\ t = (recrender\ v\ t, reccollect\ f\ v\ t, next\ t))$
  where $recrender\ (Just\ (hd : tl))\ t$ = let $t' = branch\ t$ in
                                     let $(h, \_, t'') = f\ (Just\ hd)\ t'$ in
                                     $hidden\ (makeName\ t)\ (makeName\ t')$
                                     $+\!\!+\!\!+\ h\ +\!\!+\!\!+\ recrender\ (Just\ tl)\ t''$
        $recrender\ t\ \_$           $= hidden\ (makeName\ t)$ "none"
        $reccollect\ f\ v\ t\ e$ = case $lookup\ (makeName\ t)\ e$ of
          $Nothing$      $\rightarrow Nothing$
          $Just$ "none" $\rightarrow Just\ [\,]$
          $Just\ s$       $\rightarrow$ let $t' = readName\ s$ in
                         case $v$ of
                           $Just\ (hd : tl) \rightarrow$ let $(\_, c, t'') = f\ (Just\ hd)\ t'$ in
                                          do $hd' \leftarrow c\ e$
                                             $tl' \leftarrow reccollect\ f\ (Just\ tl)\ t''\ e$
                                             $return\ (hd' : tl')$
                           $\_$             $\rightarrow$ let $(\_, c, t'') = f\ Nothing\ t'$ in
                                          do $hd' \leftarrow c\ e$
                                             $tl' \leftarrow reccollect\ f\ v\ t''\ e$
                                             $return\ (hd' : tl')$
```