Declarative, composable views

Michael Greenberg
Brown University
mgreenbe@cs.brown.edu

Abstract

Widespread use of HTML [W3Cb], JavaScript, and the DOM [W3Ca] has led to sub-optimal 'best practices'. We apply the bidirectional programming formalism of lenses [FGM+05] to define user interfaces in JavaScript that are declarative, composable, clear, and concise — in half the code. Additionally, we define two new bidirectional combinator over lists, order and list_map.

Categories and Subject Descriptors D [*3 Programming Languages*]: 3 Language Constructs and Features; H [*5 Information Interfaces and Presentation*]: 2 User Interfaces

1. Introduction

HTML [W3Cb], JavaScript, and the DOM [W3Ca] are the primary platform for world-wide web applications. Increasingly, entire application suites are developed for the so-called 'Web 2.0' with these technologies. Working with old HTML standards and parsers, multiple implementations of JavaScript, and an often difficult to use or non-standard DOM implemenation, many developers have resorted to tools like the Google Web Toolkit [GWT], which compile from Java into JavaScript. More often than not, though, programmers develop ad hoc solutions to the problems posed by these three tools.

In this paper, we address one of the DOM's problems. In particular, programmers use HTML and the DOM not only as the user interface, but as the application data model. To determine a piece of application-relevant data, programmers must query the DOM for the value. This querying is often done by hand, or through a thin wrapper that hides inconsistences in DOM implementations. This leads to a loss of composability, since queries tend to rely on unique identifiers and structural invariants. Since JavaScript programs can only rarely be composed from old programs, programmers are forced to write and rewrite subtle variations of the same DOM interaction code.

We apply *lenses*, the bidirectional tree combinators of **??**, to the DOM. During *get*, lenses produce a view given the data model; during *putback*, lenses merge the HTML view back into the data model. Programmers are able to use these combinators to declare DOM views, e.g.,

```
div_tag({'class': 'field'},
        ['Value: '],
        input_tag({'size': 2, 'type': 'text'}))
```

declares a view that puts the data model in an `input` tag nested inside a `div` tag, with the text 'Value: ' before it. This *lens*, defined in terms of the HTML it produces, can unpack the produced HTML back into the data model automatically. A system of *bindings* coordinates this effort, taking care of callbacks.

This compositional style liberates programmers from having to devise unique identifiers for nearly every relevant node in the DOM — something that is particularly problematic when there are lists in the data model. We believe that alleviating this problem is one of the most winning advantages of the lens system, for, as Andrew Appel says in [App92], "The beauty of FORTRAN ... is that it relieves the programmer of the obligation to make up names for intermediate results".

2. HTML, JavaScript and the DOM

2.1 HTML and the DOM

HTML [W3Cb] is the presentation language used throughout the world-wide web. In its first widespread incarnations — HTML 3, 4 — certain features were encouraged or, at the very least, common that pose many problems today. Two simple examples are non-XML markup (e.g., `<br>` without a close tag) and in-line JavaScript event handlers (e.g., `<img src="..." onclick="alert('1337 h4X0r')" />`). Browsers deal with this in an effective way by having two modes: standards mode and quirks mode. When non-standard XML is detected, the HTML parser switches to a mode able to handle the numerous oddities of real-world HTML.

Similar problems arise in JavaScript. Just like HTML, initial JavaScript implementations were poorly specified and unprincipled. Additionally, early versions of Internet Explorer and Netscape offered slightly different APIs; worse still, often the APIs were the same but the semantics were different!

The DOM API [W3Ca] emerged as a standard interface to the presentation layer, and was adopted with varying degrees of adherence by browser JavaScript implementations.

The API for the DOM is relatively low level and often verbose. For example, to create a node, one might run `var node = document.createElement('img')`. The programmer could call

```
node.setAttribute('src', 'http://...')
```

to set the attribute `src` of the `img` tag just created. Similarly, the method `node.appendChild(...)` can add children to the node. Frustratingly, there's no way to create a node with a set of attributes and a set of children in a single call, and thus there is no convenient way (within the DOM API) to make a series of JavaScript calls nest like the HTML they generate.

2.2 The state of the art

Many toolkits (e.g., [Moc], [jQu]) alleviate the DOM's problems by wrapping its API. This is often done with a function with a cryptically short name, such as `$`. In this paper, this function will be called `make_dom_node`, and it is called with a node name, an

Honors thesis Brown University, Providence, RI.

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
<title>Time Display</title>
<link rel="stylesheet" media="screen"
      href="testbed.css"></link>
<script type="text/javascript">
function loader() {
    // ...
}
</script>
</head>
<body onload="loader();">
<span id="time1"></span><br />
<span id="time2">
  <span id="start2"></span>
  <span id="stop2"></span>
</span>
</body>
</html>
```

Figure 1. The HTML base for a scripted page

optional JavaScript object mapping attribute names to values, and an optional list of child nodes. For example, the following code:

```
make_dom_node('div', { 'class': 'para' },
    [make_dom_node('p', {},
                   ['IN the beginning...']),
     make_dom_node('img', { 'src':
         'http://www.cistinechapel.com/...' })])
```

creates the HTML fragment:

```
<div class="para">
  <p>IN the beginning...</p>
  <img src="http://www.cistinechapel.com/..." />
</div>
```

Thus wrappers make the creation of HTML fragments effective, easy, and fairly similar to writing HTML itself.

Where wrappers provide less assistance is getting data in and out of the DOM. While a consistently named facade is placed in front of the DOM's `appendChild` function and its relatives, this is merely a whited sepulcher. Information is put into the DOM by reference to fixed IDs, with anywhere from one to hundreds of unique identifiers in the document. These identifiers are often generated mechanistically, but must still be referred to appropriately in order to retrieve the data that lies in the DOM.

While the issue of unique names is problematic, there are further issues with the DOM. With data coming from remote sources (e.g., the server, other clients editing a shared document) and local sources (the user), the most up-to-date model of the data is often the DOM itself. That is to say, present JavaScript practice uses the DOM as an *ad hoc* model. The application datais accessible only through the DOM, and so programmers are left struggling with a presentation API in order to manipulate application data.

Take, for example, an editable time display; it should show input boxes for hours, minutes, and seconds, for both a start time and a stop time. To show generality, we'll display (from a single model) two time displays working side-by-side. The base HTML structure which the JavaScript will hook into is in figure 1; the final rendering can be seen in 2.

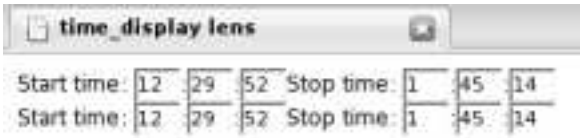

Figure 2. Two time displays, rendered side-by-side

```
// the global, top-level data model
var model = { 'start': { h: 12, m: 29, s: 52 },
              'stop': { h: 1, m: 45, s: 14 } };

function time_id(id, type) {
  return id + '_' + type;
}

function make_time_display(time, id, callback, text, sep) {
  text = text || ''; // no text, by default
  sep = sep || ':'; // default separator

  function time_v(outer_id, id, v) {
    var node = make_dom_node('input',
        { 'id': outer_id + '_' + id,
          'class': 'time_' + id, 'size': 2,
          'value': v }, []);
    // manage callbacks...
    add_event_handler(node, 'change', callback);
    add_event_handler(node, 'keypress', callback);

    return node;
  }

  // * node creation
  return make_dom_node('span', { 'id': id },
                       [text,
                        time_v(id, 'hours', time.h),
                        sep,
                        time_v(id, 'minutes', time.m),
                        sep,
                        time_v(id, 'seconds', time.s)]);
}

function extract_time_display(id) {
  return {
      'h': get_dom_object(time_id(id, 'hours')).value,
      'm': get_dom_object(time_id(id, 'minutes')).value,
      's': get_dom_object(time_id(id, 'seconds')).value
  };
}
```

Figure 3. A conventional JavaScript time display

```
function make_time1(model) {
  var time1 = get_dom_object('time1');

  // manually deal with DOM creation/deletion
  while (time1.hasChildNodes()) {
    time1.removeChild(time1.firstChild);
  }

  // * multiple calls to dom_update
  time1.appendChild(make_time_display(
      model.start, 'start', dom_update(0, 'start'),
      'Start time: '));
  time1.appendChild(make_time_display(
      model.stop, 'stop', dom_update(0, 'stop'),
      'Stop time: '));
}

function make_time2(model) {
  // slightly more direct
  var start2 = get_dom_object('start2');
  start2.parentNode.replaceChild(make_time_display(
      model.start, 'start2',
      dom_update(1, 'start2', 'start'),
      'Start time: '),
    start2);
  var stop2 = get_dom_object('stop2');
  stop2.parentNode.replaceChild(make_time_display(
      model.stop, 'stop2',
      dom_update(1, 'stop2', 'stop'),
      'Stop time: '),
    stop2);
}
```

Figure 4. Time displays with start and stop times

```
var model_updaters = [];

var dom_update = function (idx, id, which) {
  which = which || id; // update start or stop?

  var callback = function () {
    var v = clone(model);
    v[which] = extract_time_display(id); // *

    if (!equal(model, v)) {
      model = v;

      for (var i = 0;i < model_updaters.length;i++) {
        if (idx != i) { model_updaters[i](model); }
      }
    }
  };

  return function () {
    window.setTimeout(callback, 5);
    return true;
  };
}

make_time1(model);
model_updaters.push(make_time1);

make_time2(model);
model_updaters.push(make_time2);
```

Figure 5. DOM update callbacks in conventional JavaScript

The JavaScript object `model` is the global data model — a JavaScript object with properties `start` and `stop`, each of which refers to a 'time' — a JavaScript object with properties `h`, `m`, and `s`.

The function `make_time_display` in figure 3 takes in a time (represented as the aforementioned JavaScript object, a sub-object of `model`), an id for future reference to the time display, a callback to call when the user edits the time, a label `text` (the empty string by default), and a separator `sep` (a colon by default). The functions `make_dom_node` and `add_event_handler` are common wrappers around the DOM API to simplify object creation. As an example, the line marked `*` in figure 3 creates a `span` element with an `id` attribute equal to the value of the `id` variable; its children are a a `PCDATA` node with the value of `text` and the `sep`-separated input boxes formed by the inner function `time_v`. `add_event_handler` is used to hide inconsistences in the callback registration and event models between browsers.

The function `extract_time_display` looks up the time display rendered with a given id and extracts the value as a JavaScript object with properties `h`, `m`, and `s`.

2.3 Problems

There are several problems with the above. First and foremost, the structure of the view is duplicated in code — first in `make_time_display`, where the JavaScript object provided in `time` is deconstructed into the three `input` tag children of the `span` tag; and again in `extract_time_display`, where the `input` tags are looked up by generated id and the values extracted.

A second problem lies in the callbacks. Either the same callback for every `input` tag must be used — the option taken in `make_time_display` in figure 3 — or a callback for each must be written. In either case, logic for determining where the event occurred and which part of the model must be updated.

The role of `idx` is subtle. When the first view, created by `make_time1`, is updated, the model will be updated by `dom_update`. This new model should be redisplayed in `make_time2`, so that it stays up to date. It's important that `make_time1` is *not* updated, however, since it would do so by recreating the DOM nodes — possibly losing user edit events and certainly losing text-caret focus.[1] Thus each `dom_update` function has an `idx`, which represents the index of that time display in `model_updaters`, an array of callbacks that redisplay the given time display. In order to reduce code duplication, a single `dom_update` function is written that is parameterized over a single sub-time display, passed in the paramter `which`, the value of which can be either `'start'` or `'stop'`. Note that in the definition of `make_time1` in 4, there are two calls to `dom_update`, one for each time display.

Within the callbacks, a third problem is the high level of redundancy not only the macro level — `make_time_display` and `extract_time_display` share logic — but on the micro level, as well: the unique identifiers `start2` and `stop2` are repeated four times! When JavaScript's lax static error checking is taken into account, a single typo can cause aggravating, hard-to-find bugs.[2]

[1] While it is possible to implement an update mechanism that doesn't create or destroy DOM nodes, this requires yet another function beyond `make_time_display` and `extract_time_display`: `update_time_display`. Since selective updates assuage the need for non-destructive updates and are easy to implement, that route was taken.

[2] Using technology like [Fir] makes this process easier, but not painless.

Note also that almost every data point has unique identifiers; the alternative to this is bug-prone by-hand iteration of the DOM.[3]

Another downside of unique identifiers is that they must be maintained in order to compose constructors (`make_time_display`) and accessors (`extract_time_display`). When used on a large scale, machine generated identifiers become an attractive alternative, however hard to maintain.

3. Lenses

Lenses are an abstraction for bidirectional mappings. When a programmer defines a lens, a mapping is given from JavaScript values to DOM nodes and from those DOM nodes back into JavaScript values. Thus a lens effectively defines both `make_time_display` and `extract_time_display` at once.

The advantages of lenses solve all of the problems with traditional JavaScript programming except for callback maintenance: they eliminate the duplication inherent in constructors and accessors (e.g., `make_time_display` and `extract_time_display`), remove the need for extraction code that accesses unique identifiers, and in fact make unique identifiers unnecessary except for as a way of binding to specific locations in the document.

In addition to solving all but one of the problems in the conventional JavaScript/DOM model, lenses are extremely easy to compose. Their ease of composition makes lenses useable as a widget set without the architecture of systems like [jQu] or [Ope].

3.1 Formalism

The presentation of the theory of lenses follows that of [FGM+05], which presents the theory in more detail, as well as more lenses.

The expression $a \sqsubseteq b$ is true iff $a = b$ or $a = \Omega$, where Ω is a special value marking 'undefined'. The meta-variables A and C represent arbitrary sets; a and c are elements in those sets, respectively.

A *lens* l is the tuple $\langle l\nearrow, l\searrow\rangle$, which consists of two operators, read *get* and *putback*. The operator $l\nearrow$ (*get*) is a partial function from a set C to a set A; the operator $l\searrow$ (*putback*) is a partial function from $A \times C$ to C. While there need not be such an assocation, the set C is the 'concrete' or 'source' set, while A is the 'abstract' or 'target' set. A lens is *well-behaved* if it upholds the following laws:

$$l\searrow((l\nearrow c),\ c) \sqsubseteq c \qquad \forall\, c \in C \qquad \text{(GETPUT)}$$
$$l\nearrow(l\searrow(a,\ c)) \sqsubseteq a \quad \forall\,(a,c) \in A \times C \quad \text{(PUTGET)}$$

The GETPUT law requires that a *putback* immediately after a *get* produces the original (concrete) value. This is a weak, unidirectional guarantee of invertability.

The PUTGET law requires that a *get* immediately after a putback produces the original (abstract) value. This is the other unidirectional guarantee of invertability.

In general, lenses work over a set of *trees* $\mathcal{T}$, which is taken to mean unordered trees with named edges, where names come from some set $\mathcal{N}$. Written in horizontal notation,

$$t = \{w \mapsto \{\}, b \mapsto \{c \mapsto \{\}\}\}$$

is a double-edged tree with the edge labelled w pointing to the empty tree (the *value* w), and the edge labelled b pointing to the value c. The set of labels of t is $\mathrm{dom}(t) = \{w, b\}$. The expression $t(w)$ is equivalent to $\{\}$, while $t(b) = \{c \mapsto \{\}\}$. For an $n \notin \mathrm{dom}(t)$, let $t(n) = \Omega$.

[3] Toolkits like [jQu] abstract over this a little, but it must still be written by hand and is often fragile — a node intervening between a parent node and children being searched for can disrupt the results.

$$\begin{aligned}
l &= \mathrm{hoist}\ a; \mathrm{hoist}\ b \\
t &= \{a \mapsto \{b \mapsto \{c \mapsto \{\}\}\}\} \\
l\nearrow t &= \{c \mapsto \{\}\} \\
l\searrow(\{d \mapsto \{\}\},\ t) &= \mathrm{hoist}\ a \searrow (\mathrm{hoist}\ b \searrow (\{d \mapsto \{\}\},\ t(a)), t) \\
&= \mathrm{hoist}\ a \searrow (\{b \mapsto \{d \mapsto \{\}\}\},\ t) \\
&= \{a \mapsto \{b \mapsto \{d \mapsto \{\}\}\}\}
\end{aligned}$$

Figure 6. $\mathrm{hoist}\ a; \mathrm{hoist}\ b$

3.2 Lenses

A few relevant lenses are presented. The proofs of well-behavedness and other properties are available in [FGM+05].

3.2.1 Basic lenses

First, the id lens takes trees in C to trees in A.

$$\begin{aligned}
\mathrm{id}\nearrow c &= c \\
\mathrm{id}\searrow(a,\ c) &= a
\end{aligned}$$

Note that GETPUT and PUTGET hold trivially. Note additionally that the second part of the tuple is ignored during *putback*; such lenses are called *oblivious*.

A more useful lens is const:

$$\begin{aligned}
(\mathrm{const}\ v\ d)\nearrow c &= v \\
(\mathrm{const}\ v\ d)\searrow(a,\ c) &= \begin{cases} c & c \neq \Omega \\ d & c = \Omega \end{cases}
\end{aligned}$$

The const lens is not oblivious, since it relies on the value of c during *putback*.

The last of the basic lenses is a simple tree lens, hoist:

$$\begin{aligned}
(\mathrm{hoist}\ n)\nearrow c &= c(n) \quad \text{if } \mathrm{dom}(c) = \{n\} \\
(\mathrm{hoist}\ n)\searrow(a,\ c) &= \{n \mapsto a\}
\end{aligned}$$

During *get*, hoist "hoists" its argument out of a single-valued tree with a given label; during *putback*, the given abstract argument is "plunged" back under the given label. It is important to note that hoist is defined only for certain trees, e.g. both $\{\}$ and $\{n \mapsto \{q \mapsto \{\}\}, v \mapsto \{\}\}$ are invalid arguments to $\mathrm{hoist}\ n$. In the formalism, this simply means that those trees aren't in the domain of $\mathrm{hoist}\ n$ (in fact, they're not in the domain of any hoist).

The hoist lens has a dual, plunge:

$$\begin{aligned}
(\mathrm{plunge}\ n)\nearrow c &= \{n \mapsto c\} \\
(\mathrm{plunge}\ n)\searrow(a,\ c) &= t \quad \text{if } a = \{n \mapsto t\}
\end{aligned}$$

3.2.2 Combining forms

Given this basic set of lenses, some combining forms can be introduced. First, there is a general sequencing operator, called ; in [FGM+05]. It is pronounced 'seq'. Here we use metavariables l and k to represent arbitrary lenses.

$$\begin{aligned}
(l; k)\nearrow c &= k\nearrow(l\nearrow c) \\
(l; k)\searrow(a,\ c) &= l\searrow(k\searrow(a, l\nearrow c),\ c)
\end{aligned}$$

In the *get* direction, ; is intuitive, and behaves like $\circ$, the standard function composition operator. During *putback*, however, ; behaves differently. In order to get the domains right, k must be given the result of $l\nearrow c$ as its concrete argument — during *get*, $k\nearrow$ received $l\nearrow c$, and so it should receive that on *putback* as well. l, however, received just c during *get*, and so should use that as its concrete argument.

Consider figure 6 as an example. Note how an edit to the abstract view, changing the value c to the value d was propagated back through the lenses to be a deep edit to the original tree t.

The simplest conditional lens, fork, uses predicates over edge names to split objects between two different lenses. First we define the operators $|_p$ and $\backslash_p$ on objects: $o|_p \equiv \{n \mapsto o(n) \mid p(n)\}$. That is, $o|_p$ is the object o restricted to edges satisfying the predicate p. We then define its dual, $o\backslash_p = o|_{\neg p}$. We say that $o_1 \cdot o_2$ is the tree such that $\forall\, t_1 \in \mathrm{dom}(o_1).(o_1 \cdot o_2)(t_1) = o_1(t_1)$ and $\forall\, t_2 \in \mathrm{dom}(o_2).(o_1 \cdot o_2)(t_2) = o_2(t_2)$ and $\mathrm{dom}(o_1) \cap \mathrm{dom}(o_2) =$; that is, it $o_1 \cdot o_2$ the disjoint tree-intersection of o_1 and o_2.

$$\begin{aligned}(\text{fork } p\, l_p\, l_f) \nearrow c &= (l_p \nearrow c|_p) \cdot (l_f \nearrow c\backslash_p) \\ (\text{fork } p\, l_p\, l_f) \searrow (a,\, c) &= (l_p \searrow (a|_p, c|_p)) \cdot \\ &\quad (l_f \searrow (a\backslash_p, c\backslash_p))\end{aligned}$$

The first lens, l_p, is the 'pass' lens, and l_f is the 'fail' lens. They are so-named because l_p is given the object with properties that 'pass' the predicate p, and l_f is given the object with properties that 'fail' p.

Using fork, we can define filter and focus (see section 3.2.3). The filter lens filters out edges that don't match a predicate:

$$\text{filter } p\, d \equiv \text{fork } p \text{ id } (\text{const } \{\}\ d)$$

The passing properties go through the $l_p = \text{id}$ lens, so the object of passing properties is left alone. Failing properties are sent through $l_f = \text{const } \{\}\ d$. When the results of each lens' *get* are merged, the object of passing properties is merged with the empty object produced by l_f. On *putback*, constwill either restore the missing properties or put in the given default value.

There is a more general conditional lens, xfork, that uses two predicates: p_c and p_a. Its *get* direction is identical to fork with $p = p_c$, but during *putback* the abstract argument $a \in A$ is split using p_a:

$$(\text{xfork } p_c\, p_a\, l_p\, l_f) \searrow (a,\, c) = (l_p \searrow (a|_{p_a}, c|_{p_c})) \cdot (l_f \searrow (a\backslash_{p_a}, c\backslash_{p_c}))$$

The predicate p_a allows for more refined handling during *putback*; for an example of its use, see add in section 3.2.3.

In addition to the basic sequencing combinator ; and the conditional combinators fork and xfork, there are tree-structural combining forms. The most general tree-structural combining form is wmap:

$$\begin{aligned}(\text{wmap } m) \nearrow c &= \{n \mapsto m(n) \nearrow c(n) | n \in \mathrm{dom}(c)\} \\ (\text{wmap } m) \searrow (a,\, c) &= \left\{ \begin{array}{c} n \mapsto m(n) \searrow (a(n),\, c(n)) \mid \\ n \in \mathrm{dom}(a) \end{array} \right\}\end{aligned}$$

Here m is a partial function from $\mathcal{N}$ to lenses; it is total on $\mathrm{dom}(c) \cup \mathrm{dom}(a)$. Remember that $c(n) = \Omega$, for $n \notin \mathrm{dom}(c)$. So wmap applies a (possibly) different lens to each child of a tree on both *get* and *putback*; during the latter, it will new children in the abstract argument will be *putback* with Ω as the abstract argument.

3.2.3 Focus and add

The reason *putback* takes a second argument, $c \in C$, is so that *get* can project away information. One way to do this is with the lens focus. Using filter and letting $\lambda e.e = n$ define the predicate that matches only n, we define focus:

$$\text{focus } n\, d \equiv \text{filter } (\lambda e.e = n)\, d; \text{hoist } n$$

All edges but n are filter-ed out, and then n is hoist-ed up.

Just as focus restricts the information presented by *get*, add adds constant, immutable information that is projected away on *putback*. It is defined using the more complicated conditional operator xfork(where $\lambda e.\ \bot$ is the predicate that is always false):

$$\text{add } n\, t \equiv \text{xfork } (\lambda e.\ \bot)\, (\lambda e.e = n)\, (\text{const } t\ \{\}\,; \text{plunge } n)\ \text{id}$$

The add lens is complicated enough that it should be stepped through carefully.

$$\begin{aligned}t &= \{baz \mapsto \{quux \mapsto \{\}\}\} \\ l &= \text{add } foo\ \{bar \mapsto \{\}\} \\ &= \text{xfork } (\lambda e.\ \bot)\ (\lambda e.e = foo) \\ &\qquad (\text{const } \{bar \mapsto \{\}\}\ \{\}\,; \text{plunge } foo)\ \text{id} \\ l \nearrow t &= ((\text{const } \{bar \mapsto \{\}\}\ \{\}\,; \text{plunge } foo) \nearrow t) \cdot (\text{id} \nearrow t) \\ &= \{foo \mapsto \{bar \mapsto \{\}\}\} \cdot t \\ t' &= \{foo \mapsto \{bar \mapsto \{\}\}, baz \mapsto \{quux \mapsto \{\}\}\}\end{aligned}$$

On putback, $p_a = \lambda e.e = n$ does its work.

$$\begin{aligned}l \searrow (t', t) &= (\text{id} \searrow (t'\backslash_{p_a}, t\backslash_{p_c})) \cdot \\ &\quad ((\text{const } \{bar \mapsto \{\}\}\ \{\}\,; \text{plunge } foo) \searrow \\ &\quad\ (t'|_{p_a},\, t|_{p_c})) \\ &= (\text{id} \searrow (t, t)) \cdot \\ &\quad ((\text{const } \{bar \mapsto \{\}\}\ \{\}\,; \text{plunge } foo) \searrow \\ &\quad\ (\{foo \mapsto \{bar \mapsto \{\}\}\}, \{\})) \\ &= t \cdot \{\} \\ &= t\end{aligned}$$

Note that $l_p = \text{const } \{bar \mapsto \{\}\}\ \{\}\,; \text{plunge } foo$ takes what was added and puts it back to the empty tree. It may help to recall the definition of ; from section 3.2.1.

3.2.4 Lists and ordered data

Unordered edge-labelled trees in $\mathcal{T}$ can distinguish between children, but there is no way to establish an order between children. In [FGM+05], ordered data are modelled with linked lists, trees with children hd and tl, corresponding to the value of an entry in the list and a pointer to the next node in the list. They are able to to get quite far with this model, defining filters and iterators over lists. Their iterator over lists, list_map, is fairly weak: adding and removing items to and from the list is not reliable, and can result in mismatches between entries in the abstract list and the concrete list.

In the sequel, we will present a formalism for array-style lists, addressed by index, and a list_map that supports edits. For discussion of how this work differs from [BFPS07], see 5.

Let a list be defined as the language $\mathcal{L} = \mathcal{T}^*$, strings of zero or more trees. We write $\mathcal{L}_A$ for lists of trees in the set A. Thus we have ϵ as the empty list and $\cdot$ as the list concatenation operator. We inductively define the function $\text{length} : \mathcal{L} \to \mathbb{Z}$.

$$\begin{aligned}\text{length}(\epsilon) &= 0 \\ \text{length}((t \in \mathcal{T}) \cdot (l \in \mathcal{L})) &= 1 + \text{length}(l)\end{aligned}$$

We use sum notation to represent concatenation over a series; that is, $\sum_{i=0}^{n} t_i = t_0 \cdot t_1 \cdot \ldots \cdot t_n$. We define equality-modulo Ω pointwise and write it, as usual, with the $\sqsubseteq$ operator.

Before addressing list_map, we'll define two lenses that find much use in the DOM: layout, and its workhorse, order. Each HTML element has an unordered set of attributes and an ordered set of children. Suppose we want to create the following HTML fragment:

```
<span class="time">
  Hours: <input class="time hours"
                type="text" size="2" />
</span>
```

The `span` has two children: a text node, with the value `Hours:` , and an `input`tag, with a set of attributes. If we define this as a lens, we'll want to send some input value (say, the `hours` attribute of

some object passed in as the concrete tree during *get*) to the `value` attribute of the `input` tag. Leaving explanation of the DOM lenses themselves to section 4, the list-to-object transformation works like so:

```
l = layout('text', 'Hours: ',
       'hours',
           input_tag({'class': 'time hours',
                      'type': 'text',
                      'size': 2 })).
    seq(span_tag({ 'class': 'time' }))
```

During *get*, this lens will take an object `o` with a single property `hours` and map it to a list

```
['Hours: ', input_tag(...).get(o)]
```

which is in turn passed to the span_tag lens, which will use the provided list as its children. For example, calling `l.get('hours': 12)` will create the HTML fragment above with the `value` attribute of the `input` tag set to `12`.

The layout lens works as a combination of wmap, add, and order. The arguments to layout come in name/mapping pairs, where the mapping can be either a lens or a constant (a number or some text to be transformed into a DOM node, a pre-made DOM node). layout is defined in three stages: first wmap with the name/lens mappings, and then a series adds with the name/constant mappings. Finally, layout delegates to order. The orderlens takes a list of edge names, and then maps objects with those names to a list, with the values in order according to the list of names.

$$(\text{order } \sum_{i=0}^{k} n_i) \nearrow c \quad = \quad \sum_{i=0}^{k} o(n_i)$$

$$(\text{order } \sum_{i=0}^{k} n_i) \searrow (\sum_{i=0}^{k} a_i, c) \quad = \quad \{n_i \mapsto a_i\}$$

Theorem 1. *For all* $c \in C$ *such that* $\text{dom}(c) = \{n_i | 0 \leq i < k\}$ *and* $a \in A$ *such that* $\text{length}(a) = k$, order $\sum_{i=0}^{k} n_i$ *upholds* GETPUT *and* PUTGET.

Proof. By case analysis. □

3.2.5 The list_map lens

Let $\Sigma = \{a_i(v), d_i\}$ be the set of edit operations. $a_i(v)$ 'adds' the value v to a list at index i; d_i 'deletes' the value from index i. This is expressed formally in the partial function $\text{apply} : \Sigma \times \mathcal{L} \rightharpoonup \mathcal{L}$.

$$\begin{aligned} \text{apply}(a_i(v), \sum_{i=0}^{n} t_i) &= t_0 \cdot t_1 \cdot ... \cdot t_i \cdot v \cdot t_{i+1} \cdot ... \cdot t_n \\ &\quad \text{for } i \leq n+1 \\ \text{apply}(d_i, \sum_{i=0}^{n} t_i) &= t_0 \cdot t_1 \cdot ... \cdot t_{i-1} \cdot t_{i+1} \cdot ... \cdot t_n \\ &\quad \text{for } i \leq n \end{aligned}$$

The *partial* function apply applies the edit to the list. To see why must apply be partial, consider the value of $\text{apply}(d_0, \epsilon)$ — there's nothing to delete! An edit σ is *valid* for a given list l iff $(\sigma, l) \in \text{dom}(\text{apply})$. Out of convenience, the validity predicate valid will be defined not over individual edits, but over the language $\mathbb{E} = \Sigma^*$, strings of zero or more edits; it maps onto $\mathbb{B}$, the boolean set containing $\top$ and $\bot$. Thus we have $\text{valid} : \mathbb{E} \times \mathcal{L} \rightarrow \mathbb{B}$:

$$\begin{aligned} \text{valid}(\epsilon, l) &= \top \\ \text{valid}((\sigma \in \Sigma) \cdot (e \in \mathbb{E}), l) &= (\sigma, l) \in \text{dom}(\text{apply}) \wedge \\ &\quad \text{valid}(e, \text{apply}(\sigma, l)) \end{aligned}$$

We define the replay of a string of edits as the partial function $\text{replay} : \mathbb{E} \times \mathcal{L} \rightharpoonup \mathcal{L}$.

$$\begin{aligned} \text{replay}(\epsilon, l) &= l \\ \text{replay}((\sigma \in \Sigma) \cdot (e \in \mathbb{E}), l) &= \text{replay}(e, \text{apply}(\sigma, l)) \end{aligned}$$

Lemma 1. $\text{dom}(\text{replay}) = \{(e, l) \in \mathbb{E} \times \mathcal{L} | \text{ valid}(e, l)\}$

Proof. By induction on e. □

Lemma 2. *Given lists* l_1 *and* $l_2 \in \mathcal{L}$ *such that* $\text{length}(l_1) = \text{length}(l_2)$*, for all edits* $e \in \mathbb{E}$ *such that* $\text{valid}(e, l_1)$*, we have:*

1. $\text{valid}(e, l_2)$
2. $\text{length}(\text{replay}(e, l_1)) = \text{length}(\text{replay}(e, l_2))$

Proof. By induction on the number of edits in e. □

We write $\text{replay}_k(e, l)$ to mean $\text{replay}(e, l)$ where every edit of the form $a_i(v)$is replaced with $a_i(k \nearrow v)$.

Lemma 3. *If, for* $l_1, l_2 \in \mathcal{L}$ *and* $e \in \mathbb{E}$, $\text{valid}(e, l_1)$*, then for all lenses* k*,*

1. $\text{length}(\text{replay}(e, l)) = \text{length}(\text{replay}_k(e, l))$
2. *Letting* $\sum_{i=0}^{n} a_i = \text{replay}_k(e, l_2)$ *and* $\sum_{i=0}^{n} c_i = \text{replay}(e, l_1)$*, then for each* i *either* $a_i = kgetv$ *and* $c_i = v$ *for some* $a_{i'}(v)$ *or* $a_i = l_{2i'}$ *and* $c_i = l_{1i'}$*.*

Proof. By induction on the number of edits in e, with help from lemma 2. □

We are now prepared to introduce the E-lens list_map, which follows a variation of GETPUT and PUTGET. An *E-lens* l is the tuple $\langle l \nearrow : \mathcal{L}_C \rightarrow \mathcal{L}_A, l \searrow_e : \mathcal{L}_A \times \mathcal{L}_C \rightarrow \mathcal{L}_C \rangle$, read 'get' and 'putback with edits'. The *putback* operator is parameterized over a string of edits $e \in \mathbb{E}$. E-lenses could just as easily be formulated, then, as the triple $\langle l \nearrow : \mathcal{L}_C \rightarrow LS_A, l \searrow : \mathbb{E} \times \mathcal{L}_A \times \mathcal{L}_C \rightarrow \mathcal{L}_C, e \in \mathbb{E} \rangle$.

The only example we have of an E-lens is the list-iteration lens list_map. In the *get* direction, it applies ('maps') a lens k over each tree in the list:

$$(\text{list\_map}_e\ k) \nearrow \sum_{i=0}^{n} c_i = \sum_{i=0}^{n} k \nearrow c_i$$

The *putback* of list_map is more complicated — it uses the string of edits e to alter both the abstract and concrete lists.

$$(\text{list\_map}_e\ k) \searrow (a = \textstyle\sum_{i=0}^{n} a_i, c = \sum_{i=0}^{n} c_i) = \\ \textbf{let } \begin{array}{rcl} \sum_{i=0}^{n'} a'_i & = & \text{replay}_k(e, a) \\ \sum_{i=0}^{n'} c'_i & = & \text{replay}(e, c) \end{array} \textbf{ in} \\ \textstyle\sum_{i=0}^{n'} k \searrow (a'_i, c'_i)$$

We then prove variants of GETPUT and PUTGET for list_map.

Theorem 2. *For the lens* $l = list_map_e k$*, all* $c \in C$*,* $a \in A$*, and* $e \in \mathbb{E}$ *such that* $\text{length}(a) = \text{length}(c)$ *and* $\text{valid}(e, a)$*:*

$$l \searrow_e ((l \nearrow c),\ c) \sqsubseteq \text{replay}(e, c) \quad \textit{(GETPUT}_e\textit{)}$$
$$l \nearrow (l \searrow_e (a,\ c)) \sqsubseteq \text{replay}_k(e, a) \quad \textit{(PUTGET}_e\textit{)}$$

Proof. (GETPUT$_e$) In the simplest case, we note that $l \nearrow c = \epsilon$, that the only list with length equal to ϵ is ϵ, and that $l \searrow (\epsilon, \epsilon) = \epsilon$.

First, we have $l \nearrow \sum_{i=0}^{n} c_i = \sum_{i=0}^{n} k \nearrow c_i$. We wish to show that $l \searrow_e (\sum_{i=0}^{n} k \nearrow c_i, c) \sqsubseteq \text{replay}(e, c)$. Running the *putback*, we end up with

$$(\text{list\_map}_e\ k) \searrow (a = \textstyle\sum_{i=0}^{n} k \nearrow c_i, c = \sum_{i=0}^{n} c_i) = \\ \textbf{let } \begin{array}{rcl} \sum_{i=0}^{n'} a'_i & = & \text{replay}_k(e, k \nearrow c_i) \\ \sum_{i=0}^{n'} c'_i & = & \text{replay}(e, c) \end{array} \textbf{ in} \\ \textstyle\sum_{i=0}^{n'} k \searrow (a'_i, c'_i)$$

First, we have that the two lists with replay-ed edits are of the same length by lemma 3. To restate our earlier goal, we must show that $\sum_{i=0}^{n'} k \searrow (a'_i, c'_i) \sqsubseteq \text{replay}(e, c)$, or, more concisely, that $\sum_{i=0}^{n'} k \searrow (a'_i, c'_i) \sqsubseteq \sum_{i=0}^{n'} c'_i$.

Also by lemma 3, for each i, a'_i is either the result of an edit $a_{i'}(v)$ (for some i', since a'_i may have been 'pushed up' by edits earlier in the list) or corresponds to $k \nearrow c_{i''}$. In the first case, we have $c'_i = v$ for that edit, and so $a'_i = k \nearrow c'_i$; in the latter, we have that $c'_i = c_{i''}$ and so $a'_i = k \nearrow c'_i$.

Thus, in both cases, we wish to prove that $k \searrow (k \nearrow c'_i, c'_i) \sqsubseteq c'_i$ for each i. Since k upholds GETPUTand $\sqsubseteq$ is point-wise on $\mathcal{L}$, l upholds GETPUT_e. □

(PUTGET_e) The ϵcase is as in GETPUT_e: $l \searrow_e (\epsilon, \epsilon) = \epsilon$, $l \nearrow \epsilon = \epsilon$, and $\epsilon \sqsubseteq \epsilon$.

Given $l \searrow_e (a, c) = \sum_{i=0}^{n'} k \searrow (a'_i, c'_i)$, we wish to prove that:

$$l \nearrow \sum_{i=0}^{n'} k \searrow (a'_i, c'_i) \quad \sqsubseteq \quad \text{replay}_k(e, a) = a'_i$$

$$\sum_{i=0}^{n'} k \nearrow (k \searrow (a'_i, c'_i) \quad \sqsubseteq \quad a'_i$$

Given that k upholds PUTGET and that $\sqsubseteq$ is point-wise on $\mathcal{L}$, l upholds PUTGET_e. □

QED. □

There are two important limitations of this formalization and proof. First, where do the edits e of $\searrow_e$ come from? Second, how is nesting of list_map lenses supported? Given a lens working on a list of lists of numbers, the lens list_map (list_map (focus foo)) won't be able to track edits in the inner list_map.

The solution to the second problem is a technical one, of which only a sketch is given, since it doesn't affect the proof. All lenses must be converted into E-lenses. For some lenses, like hoist and focus, this requires no changes — edits make no sense in their non-list context. Combinators like ; and wmap, however, must keep track of edits to be passed down to their sublenses. The sequencing lens, ;, keeps track of edits for each of its child lenses; wmap must keep track of edits for each property it maps.[4]

The solution to the first problem is the JavaScript/DOM event handling system of binding system — see section 4.1.

As an example of the weakness of this system, note that the edits $a_i(v)$ and d_i are insufficient to implement a swap edit — during putback, the value from c would be lost, replaced with the v of $a_i(v)$. Extending the system with a swap event is not complicated — it only requires a slight modification to lemma [**?**] — but it shows the awkwardness of the system.

4. DOM Lenses

The system of lenses described in section 3 can be applied to JavaScript-enabled web pages to alleviate this problem. We take JavaScript values as C and DOM nodes as A. The *get* operator 'projects' or 'displays' a value in the DOM; the *putback* operators 'merges' changes in the DOM back into a JavaScript model.

The most general DOM lens is the tag lens. Given a tag name, a placement for values (e.g. as a child node, as an attribute), and a set of default attributes and child nodes, it maps values to and from DOM objects. This is best shown by example:

$$\begin{aligned} l_1 &\equiv \text{tag } \texttt{'input' 'value'} \\ &\quad \{\ \texttt{'id': 'hours'}\ \}\ [] \\ \text{html}\, x &\equiv \texttt{<input id="hours" value="}x\texttt{" />} \\ l_1 \nearrow 12 &= \text{html}\, 12 \\ l_1 \searrow (\text{html}\, 15, 12) &= 15 \end{aligned}$$

The first argument to tag in l_1 says that an `input` tag should be created; the second says that the concrete-tree value should be put in the `value` attribute during get and that the salient data will reside there on putback. Next, a JavaScript object is given indicating that the attribute `id` should be set to `hours`. The final argument is a list of children for the node; children of `input` tags do nothing, so the empty list is given. As an alternative example:

$$\begin{aligned} l_2 &\equiv l_1; \text{tag } \texttt{'span' 'child'} \\ &\quad \{\}\ \texttt{['Hours: ']} \\ \text{html}\, x &\equiv \texttt{Value:} \\ &\quad \texttt{<input id="hours" value="}x\texttt{" />} \\ &\quad \texttt{} \\ l_2 \nearrow 12 &= \text{html}\, 12 \\ l_2 \searrow (\text{html}\, 15, 12) &= 15 \end{aligned}$$

The second lens of l_2 is a `span` tag with no attributes and a single child, a text node reading "Hours: ". The second argument, `'child'`, instructs tag to send the value on get to the first child after the default children; on putback, the last child of the node will be extracted. The lens l_1 is sequenced before the `span` lens of l_2, so the `input` tag created by l_1 is put as the last (second) child of the `span` lens. To wit:

$$\begin{aligned} l_2 \nearrow 12 &= (\text{tag } \texttt{'input' 'value'}\ \{\ \texttt{'id': 'hours'}\ \}\ \texttt{[]}; \\ &\quad \text{tag } \texttt{'span' 'child'}\ \{\}\ \texttt{['Hours: ')} \nearrow 12 \\ g_1 &= (\text{tag } \texttt{'input' 'value'}\ \{\ \texttt{'id': 'hours'}\ \}\ \texttt{[]}) \nearrow 12 \\ &= \texttt{<input id="hours" value="12" />} \\ l_2 \nearrow 12 &= (\text{tag } \texttt{'span' 'child'}\ \{\}\ \texttt{['Hours: ']}) \nearrow g_1 \end{aligned}$$

The formulation of l_2 is somewhat counterintuitive: the child of the `span` tag must be given before its parent! To alleviate this problem, a special form of 'tree sequencing' is introduced, along with per-tag lenses which use intelligent default placements. We can then rewrite l_2 as:[5]

$$\text{span\_tag}\ \{\}\ \texttt{['Hours: ']}\ (\text{input\_tag}\ \ \{\texttt{'id': 'hours'}\})$$

The span_tag automatically gets and puts its values to and from its last child; the input_tag gets and puts its values to and from the `value` attribute.

To show how lenses express more complex interfaces, the direct HTML-generating code of figure 3 will be re-written with lenses.

4.1 list_map in action

The implementation of list_map makes it easy to apply edits. list_map takes a lens contructor function, which it calls with three functions: `add_before` , `add_after` , and `delete` . During get, for each item in the list, the lens constructor is called, parameterized with functions specialized to its index. These can be embedded in, say, the `onclick` callback of a link to cause a given item to be deleted, or to add list entries before or after a given item. This is perhaps best explained in an example.

[4] As an optimization, combinators which have no list_map children don't keep track of edits at all.

[5] To simplify the argument-list parser of the `_tag` lenses, the real code would have to be plunge `v` ; span_tag `({}, 'text', 'Hours: ', 'v',` input_tag `{'id': 'hours'})`

```
// the global, top-level data model
var model = { 'start': { h: 12, m: 29, s: 52 },
              'stop': { h: 1, m: 45, s: 14 } };

function time_display(id, text, sep) {
  text = text || ''; // no text, by default
  sep = sep || ':'; // default separator

  // an abstraction for an input field
  function time_v(id) {
    return input_tag({ 'class': 'time ' + id,
                       'size': 2 });
  }

  // wrap the list as the child of a span element with
  // id="$id" while laying out the arguments in this
  // order, mapping h, m, and s to time_v inputs with
  // appropriate ids
  return span_tag({ 'id': id },
                  'text', text,
                  'h', time_v('hours'),
                  'sep1', sep,
                  'm', time_v('minutes'),
                  'sep2', sep,
                  's', time_v('seconds'));
}
```

Figure 7. A JavaScript time display with lenses

We take our model to be a list of objects with properties `header` and `txt` — a paragraph heading and some text. For example,

```
[{ header: "Intro",
   txt: "Hello, this is some text!" },
 { header: "Conclusion",
   txt: "My, that was enlightening." }]
```

The code in figure 8 uses prune, the dual of focus, to strip away the paragraph text in `txt` and then put the header in an `input` tag. It then adds references to links that, when clicked, either add or delete the entry. The `def` parameter is passed as the value to put in C — that is, the v of $a_i(v)$.

The code for `full_edit` in figure 9 displays the paragraph to edit in a `textarea` and the header above it in an `input` tag.

Since edits are managed by the three functions passed in to the lens constructor, nesting of list_maps is possible.

4.2 Binding and bind_lens

Building an HTML representation of a model is only half of the work; the function bind_lens coordinates with the DOM so that the HTML representation presented (the view) and the JavaScript model are 'in sync'.

In the JavaScript example of figure 3, the `change` and `keypress` events are singled out as events that require the model and the view to be resynchronized. Additionally, the `make_time1`and `make_time2`functions take a model and update the DOM to reflect it.

These two synchronization actions — updating the DOM from a new model and updating a model with information from the DOM when an event indicates that an edit has occured — mirror the two operations of the DOM lenses: *get* and *putback*. Ad-hoc, hand-optimized code is often used instead of the relatively clear system of figures 4 and 5 in order to mutate rather than replace the model. In general, however, following [FGM+05] and [FGK+06],

```
function edit_header(def) {
  return function (add_bef, add_aft, del) {
    var add_bef_btn =
      make_dom_node('a', { href: '#' }, ['^']);
    var add_aft_btn =
      make_dom_node('a', { href: '#' }, ['v']);
    var del_btn =
      make_dom_node('a', { href: '#'}, ['x']);

    add_event_handler(add_bef_btn, 'click',
      function (e) {
        add_bef(def);
        // false -> don't change browser URL
        return false;
    });

    add_event_handler(add_aft_btn, 'click',
      function (e) {
        add_aft(def);
        return false;
    });

    add_event_handler(del_btn, 'click',
      function (e) {
        del();
        return false;
    });

    return prune('txt').li_tag({},
      'header', input_tag({}),
      'bef', add_bef_btn,
      'sp1', ' ',
      'aft', add_aft_btn,
      'sp2', ' ',
      'del', del_btn);
  };
}

toc_lens =
  ol_tag({ id: 'toc' },
         list_map(edit_header(
           { header: 'New Section',
             txt: ['You forgot to enter text!'] })));
```

Figure 8. A list_map lens — table of contents

synchronization engines work in terms of the lens operations: get from the model, putback from the DOM.

The basic bind_lens accepts a DOM-update callback and returns a model-update callback. The latter is called with a new model when the DOM has changed, and the former is called with a new model when an external agent has modified the model, e.g. the remote value on the server has been updated.

Thus we write the actual binding as in figure 11, with moderate similarity to figure 5. There are three notable differences. First, there is nothing specific to the time display program in the lens-based definition of `model_update`; the line marked with `*` in the JavaScript-based definition highlights that requirement. Second, the `dom_update` function is passed in to the binding mechanism, rather than being part of the DOM creation itself, as in figure 4. Third, and perhaps most importantly, unique identifiers are only given to bind_lens as a hook for DOM node insertion. The JavaScript definitions of `dom_update` and `model_update` required

```
function full_edit(def) {
  return function (add_bef, add_aft, del) {
    var add_bef_btn =
      make_dom_node('a', { href: '#' }, ['^']);
    var add_aft_btn =
      make_dom_node('a', { href: '#' }, ['v']);
    var del_btn =
      make_dom_node('a', { href: '#'}, ['x']);

    add_event_handler(add_bef_btn, 'click',
      function (e) {
        add_bef(def);
        return false;
    });

    add_event_handler(add_aft_btn, 'click',
      function (e) {
        add_aft(def);
        return false;
    });

    add_event_handler(del_btn, 'click',
      function (e) {
        del();
        return false;
    });

    return div_tag({},
      'header', input_tag({}),
      'bef', add_bef_btn,
      'sp1', ' ',
      'aft', add_aft_btn,
      'sp2', ' ',
      'del', del_btn,
      'txt', div_tag({'class':'para'},
                     textarea_tag({rows: 5,
                                   cols: 80})),
      'br', make_dom_node('br', {}, []));
  };
}

content_lens =
  div_tag({ id: 'content' },
          list_map(full_edit(
            { header: 'New Section',
              txt: [''] })));
```

Figure 9. A list_map lens — document editor

```
var lens1 = span_tag({ 'id': 'time1' },
                     'start',
                     time_display('start',
                                  'Start time: '),
                     'stop',
                     time_display('stop',
                                  'Stop time: '));

var lens2_start = focus('start').
                  seq(time_display('start2',
                                   'Start time: '));
var lens2_stop = focus('stop').
                 seq(time_display('stop2',
                                  'Stop time: '));
```

Figure 10. Composing lenses

```
var model_updaters = [];

var dom_update = function (idx) {
  return function (v) {
    if (!equal(model, v)) {
      model = v;

      for (var i = 0;i < model_updaters.length;i++) {
          if (idx != i) { model_updaters[i](model); }
      }
    }
  };
};

set_error_handler(debugger_on_error);

var model_update = lens1.bind_to('time1',
                                 dom_update(0));
model_update(model);
model_updaters.push(model_update);

model_update = lens2_start.bind_to('start2',
                                   dom_update(1));
model_update(model);
model_updaters.push(model_update);

model_update = lens2_stop.bind_to('stop2',
                                  dom_update(2));
model_update(model);
model_updaters.push(model_update);
```

Figure 11. Binding lenses

two unique identifiers each — one for the hook into the DOM and another for a hook into the model.

Thus the event handling mechanism for bind_lens is general – if only a single lens works off of the model, then there is no need for a complicated `dom_update`and `model_update` as defined here; the former can be reduced to just swapping out a new model for the old one, while the latter can be kept around by reference, rather than in a list like `model_updaters`.

4.3 Lenses and Flapjax

The Flapjax programming language [Fla] is a functional-reactive programming language in the spirit of FrTime [CK06], built on top of JavaScript. It offers support for complex event-based systems both as a library and as a compiled language. Flapjax makes JavaScript/DOM programming easier by means of events — streams of values — and behaviors — special objects the value of which can change over time. Expressions which depend on behaviors (events) become behaviors (events) themselves, either explicitly by means of *lifting* or implicitly by means of the compiler. For example, in compiled Flapjax, one can declare

```
var timeB = timer_b(100); // ms since the epoch
var secsB = Math.floor(timeB / 1000)
```

The value of the variable `timeB` updates (approximately) every 100 milliseconds to a new value; whenever those updates occur, the value of the variable `secsB` is recomputed as `Math.floor(timeB / 1000)`. In turn, anything depending on `secsB` will be updated whenever it changes, viz. once a second.

Lenses take advantage of this system by allowing binding to time-varying models. As those models update, the DOM updates;

```
var model = model_b({'start': {h: 12, m: 29, s: 52},
                     'stop': {h: 1, m: 45, s: 14}});

lens1.bind_to_b(model, 'time1');
lens2_start.bind_to_b(model, 'start2');
lens2_stop.bind_to_b(model, 'stop2');
```

Figure 12. Binding lenses with Flapjax

as the DOM updates, those models are updated. We can then rewrite the binding of the time display example as simply as in figure 12.

The `_b` indicates that the values are behaviors, values which change over time. Since is based on a framework of managed callbacks, the lens binding framework can bind directly into Flapjax, with no need for user callback definitions.

5. Related Work

The two most similar systems for bidirectional computation are the basis for this work, Foster, et al. [FGM+05], and Hu, et al. [HMT04].

The seminal work of Foster, et al. on 'lens' combinators serves as the foundation for this work, which is a nontrivial re-application of their work for user interface components. Additionally, this work reifies a multi-lens synchronization system that is a hybrid of the original, single-lens synchronization originally described and the SYNC-maintenance rules of [HMT04].

In Bohannon, et al. [BFPS07], *resourceful* lenses are defined over regular expressions over strings. In particular, a Kleene-star iteration operator match is given that will adequately 'match up' any edits made to items in the sequence by means of a 'key' (a semi-unique part of the data). Our formulation of $\mathrm{list\_map}$ differs from match in several ways. First, the domains are different — regular expressions over strings and combinators over edge-labeled trees — though this can most likely be reconciled without excessive difficulty. Second, match is purely functional, working by means of a dictionary by key of sequence items, while $\mathrm{list\_map}$ must have its edits managed by a third party and is effectively stateful. Third, and perhaps most importantly, $\mathrm{list\_map}$ is partially operation based, and so matches the event model of the DOM more accurately. (For more on this, see future work in section 6.) The purely functional nature of match makes its formalization much more elegant than that of $\mathrm{list\_map}$; moreover, they prove more interesting properties of their entire R/S/K-lens hierarchy than we do for our single lens. Integrating the two is definitely a component of future work (again, see section 6).

The bidirectional formulation of Hu, et al. relates more closely to this work — both deal with the bidirectional transfer of structured model data to structured view data. Their theoretical basis differs significantly this work's, since the GETPUTGET and PUTGETPUT laws offer much weaker guarantees than their correspondents GETPUT and PUTGET. Additionally, their index-based tree combinators are quite unnatural in an HTML environment. Their system of structured edits is also a poor match for the JavaScript DOM, which doesn't provide the level of event granularity necessary to implement their system.

In contrast to the systems of both Foster, et al. and Hu, et al. linear sequencing operator ';', this work's tree-like sequencing better fits the domain.

While Meertens manuscript [Mee98] is ostensibly about "user interaction", his system, as Foster, et al. point out, is much more generally defined than this work's, though no practical application is given. His event system, like Hu, et al.'s, is "operation-based" [FGM+05], and so a poor match for the JavaScript DOM.

Despite the similarity in name, Wadler's work on 'views' [Wad87] isn't directly related to the views of this work. Not only are his *get* and *putback* user-defined, there is no structural support for user interfaces.

6. Future Work

This work grew out of frustrations with the need to use the DOM as an ad hoc data model when programming even the simplest Flapjax programs with remote updates. Fully integrating the lens system described into Flapjax is a natural first step of the future work; expanding the widget set available beyond basic HTML widgets (e.g., edit-in-place text, toolbars) would serve as a good test of both the lens system itself and its integration into Flapjax.

When integrated, the lens system will allow us to focus our attention on principled client/server synchronization for shared data. At present, clients are self-deprecating: any new model from the server (and, indirectly, from another client) trumps the current client. Other models of interest would be clients with time-based update models (e.g., don't take a value from the server unless local data hasn't changed in x milliseconds) and user-specified merging algorithms (perhaps in a variant of the lens language, where the A and C are remote and local clients, respectively).

As mentioned in section [**?**], the resourceful lenses of Bohannon, et al. [BFPS07] could serve as a launching pad for an investigation of different $\mathrm{list\_map}$ primitives and laws more general than GETPUT_e and PUTGET_e. As browser support for DOM Level 2 `DOMSubtreeModified` events [W3Ca] becomes widely available, operation-based solutions will become more viable.

At present, lens combinators are defined in JavaScript and runtime. Since lenses are amenable to an HTML-like tree-sequencing syntax, there is no reason that lenses could be written statically in the HTML itself, by means of tags in a special namespace. A compiler could then compile those special nodes into either lenses or, even better, callbacks installed directly. This would wed the efficiency of the direct callback style with the correctness, easy to use, succinct lens syntax. The compiler could be implemented either as a program run on HTML-with-lenses to generate plain old HTML, or as a dynamic JavaScript library which compiled the HTML-with-lenses to HTML on the fly, by means of the DOM API. There are advantages to both approaches: a dynamic compiler must process the page at each load, and could be expensive, but it would simplify the debugging and development process – until, perhaps, a static copy could be compiled and distributed.

References

[App92] Andrew W. Appel. *Compiling with continuations*. Cambridge University Press, New York, NY, USA, 1992.

[BFPS07] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, and Alan Schmitt. Resourceful lenses for ordered data, April 2007. Submitted for publication.

[CK06] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *Programming Languages and Systems. 15th European Symposium on Programming, ESOP 2006*, volume 3924/2006 of *Lecture Notes in Computer Science*, pages 294–308, Vienna, Austria, 2006. Springer-Verlag.

[FGK+06] J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, and Alan Schmitt. Exploiting schemas in data synchronization. *Journal of Computer and System Sciences*, 2006. To appear. Extended abstract in *Database Programming Languages (DBPL)* 2005.

[FGM+05] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to

the view update problem. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 233–246, New York, NY, USA, 2005. ACM Press.

[Fir] FireBug JavaScript debugger, http://getfirebug.com/.

[Fla] Flapjax, http://www.flapjax-lang.org/.

[GWT] Google Web Toolkit, http://code.google.com/webtoolkit/.

[HMT04] Zhenjiang Hu, Shin-Cheng Mu, and Masato Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 178–189, New York, NY, USA, 2004. ACM Press.

[jQu] jQuery, http://jquery.com/.

[Mee98] L. Meertens. Designing constraint maintainers for user interaction, 1998.

[Moc] MochiKit, http://mochikit.com/.

[Ope] OpenLaszlo, http://www.openlaszlo.org/.

[W3Ca] W3C DOM Specification, http://www.w3.org/DOM/.

[W3Cb] W3C HTML Specification, http://www.w3.org/html/.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In Steve Munchnik, editor, *Proceedings, 14th Symposium on Principles of Programming Languages*, pages 307–312. Association for Computing Machinery, 1987.