Flapjax: A Programming Language for Ajax Applications

Leo A. Meyerovich
University of California, Berkeley
lmeyerov@eecs.berkeley.edu

Arjun Guha
Brown University
arjun@cs.brown.edu

Jacob Baskin
Google
jacob.baskin@gmail.com

Gregory H. Cooper
Google
ghcooper@gmail.com

Michael Greenberg
University of Pennsylvania
mgree@seas.upenn.edu

Aleks Bromfield
Microsoft
albromfi@microsoft.com

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Abstract

This paper presents Flapjax, a language designed for contemporary Web applications. These applications communicate with servers and have rich, interactive interfaces. Flapjax provides two key features that simplify writing these applications. First, it provides *event streams*, a uniform abstraction for communication within a program as well as with external Web services. Second, the language itself is *reactive*: it automatically tracks data dependencies and propagates updates along those dataflows. This allows developers to write reactive interfaces in a declarative and compositional style.

Flapjax is built on top of JavaScript. It runs on unmodified browsers and readily interoperates with existing JavaScript code. It is usable as either a programming language (that is compiled to JavaScript) or as a JavaScript library, and is designed for both uses. This paper presents the language, its design decisions, and illustrative examples drawn from several working Flapjax applications.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Data-flow languages

General Terms Languages, Design

Keywords JavaScript, Web Programming, Functional Reactive Programming

1. Introduction

The advent of broadband has changed the structure of application software. Increasingly, desktop applications are migrating to the Web. Programs that once made brief forays to the network now "live" there. The network servers they communicate with not only provide data but also store data, enabling networked persistence. Some applications, known as *mashups*, combine data from multiple sources. Often, applications process not only static data but also continuous streams of information, such as RSS news feeds.

This paper presents Flapjax, a programming language built with such applications in mind. We make three key arguments, which this paper will substantiate:

- *Event-driven reactivity* is a natural programming model for Web applications.
- *Consistency* should be a linguistic primitive.
- *Uniformity* is possible when treating both external events (those from remote machines) and internal ones (those from local devices such as the mouse). Uniformity enables better abstractions and also reduces the number of concepts needed for reasoning and validation.

Rather than invent a language from fresh cloth, we chose to engineer Flapjax atop HTML and JavaScript (despite their warts). JavaScript offers three important benefits. First, it is found in all modern browsers, and has hence become a lingua franca. Second, it reifies the entire content of the current Web page into a single data structure called the Document Object Model (DOM), so that developers can naturally address and modify all aspects of the current page (including its visual style). Third, it provides a primitive, XMLHttpRequest, that permits asynchronous communication (in the style called Ajax [17]) without reloading the current page. This enables background communication with servers so Web applications can provide much of the reactivity of desktop applications. As a result of building atop JavaScript and HTML, Flapjax applications do not require plugins or other browser modifications; they can reuse existing JavaScript libraries; and the language can build on the existing knowledge of Web developers.

A formal presentation would hide the many pragmatic benefits and decisions in the design of Flapjax. We therefore present it through a series of increasingly sophisticated examples, including uses and mashups of popular Web services. These demonstrate how event streams, and automatic reaction to their changes, encourage several important software design principles including model-view clarification and policy-mechanism separation. We outline the implementation technique and its pragmatic choices. Finally, we discuss the use of the language in actual applications.

Language or Library? We have repeatedly referred to Flapjax as a *language*, but with a little extra effort, Flapjax can be used as a JavaScript *library*. That means the developer who does not want to add the Flapjax-to-JavaScript compiler (section 3.3) to their toolchain can include the Flapjax library and program purely in JavaScript itself; in fact, most Flapjax applications are actually written this way (section 4). This involves some overhead, as we discuss in section 3.3, but we leave this decision in the hands of developers rather than making it for them.

2. Flapjax by Example

We present Flapjax as a programming language through examples. They are necessarily short, but Flapjax is a living, breathing language! We invite the reader to view and run the demos on the language site,[1] read the documentation of the many primitives, and try out their own examples.

2.1 The Structure of JavaScript Programs

Before we study Flapjax, let us consider a very simple JavaScript program. It displays the time elapsed since starting or clicking on a button (figure 1, which elides some of the HTML scaffolding). The point of this program is to fill in a value for the `curTime` element on the HTML page. Consider the reasoning a developer must employ:

1. The value is ostensibly displayed by the second line of the function `doEverySecond`.
2. The value displayed is that of `elapsedTime`.

[1] www.flapjax-lang.org

```
var timerID = null;
var elapsedTime = 0;

function doEverySecond() {
  elapsedTime += 1;
  document.getElementById("curTime")
          .innerHTML = elapsedTime; }
function startTimer() {
  timerId = setInterval("doEverySecond()", 1000); }
function resetElapsed() {
  elapsedTime = 0; }

<body onload="startTimer()">
<input id="reset" type="button" value="Reset"
       onclick="resetElapsed()"/>
<div id="curTime">  </div>
</body>
```

Figure 1. Elapsed Time in JavaScript

3. `elapsedTime` is set in the previous line.
4. But this depends on the invocation of `doEverySecond`.
5. `doEverySecond` is passed inside a *string* parameter to `setInterval` inside `startTimer`.
6. `startTimer` is called by the `onload` handler...so it appears that's where the value comes from.
7. Is that it? No, there's also the initialization of the variable `elapsedTime` at the top.
8. Oh wait: `elapsedTime` is also set within `resetElapsed`.
9. Does `resetElapsed` ever execute? Yes, it is invoked in the `onclick`.

Just to understand this tiny program, the developer needs to reason about timers, initialization, overlap, interference, and the structure of callbacks. (We trust the reader spotted the semantic bug? See section 2.2 for the answer.)

The culprit here is not JavaScript, but the use of callbacks and their effect on program structure. Callbacks are invoked by a generic event loop (e.g., in the JavaScript runtime) which has no knowledge of the application's logic, so it would be meaningless for a callback to compute and return a non-trivial value. The return type of a callback is therefore the equivalent of `void`. That immediately means developers can no longer use types to guide their reasoning. Furthermore, a `void`-typed function or method must have side-effects to be useful, so even local reasoning about a program's data depends on global reasoning about the program's control flow, destroying encapsulation and abstraction. Indeed, Myers has forcefully made similar critiques of callbacks [25].

The problem of comprehension affects not only humans but also tools. For example, static analysis and verification engines must decipher a program's intent from a highly fragmented description, and must reconstruct its dataflow from a rat's nest of fragments of control. As previous work on model checking Web applications has shown [23], a more direct program structure is a great help in this regard.

The asynchronous nature of Ajax applications further compounds these problems. Because *the primary composition operator is a side-effect*, interleavings and interactions become the developer's responsibility. The network does not guarantee message ordering; deployment can further affect ordering (e.g., higher server loads, or users at a greater geographic distance from servers than developers). All the usual problems of concurrency manifest, without even the small comfort of locking.

Despite this, callbacks appear necessary to receive notification of events, and are charged with propagating them through the system to keep the data model up-to-date. In an Ajax application, there are numerous sources of updates a program must process, such as:

1. initial data from each host
2. user actions (e.g., button clicks, mouse movements)
3. updates from a data stream
4. changes to data made in another concurrent session
5. acknowledgments from servers (e.g., in response to a store request)
6. changes to the access-control policy

As a result, Ajax applications are agglomerations of callbacks with largely implicit control flows. Some operations exacerbate this: for instance, XMLHttpRequest requires a callback (`onreadystatechange`) that it invokes up to *four* times, providing the status in a field (not as a parameter to the callback).

2.2 The Flapjax Alternative

Flapjax endows JavaScript with a *reactive* semantics. In effect, if a developer defines $y = f(x)$ and the value of x changes, the value of y is recomputed *automatically*. Concretely, Flapjax is JavaScript augmented with two new kinds of data. A *behavior* is like a variable—it always has a value—except that changes to its value propagate automatically; an *event stream* is a potentially infinite stream of discrete events whose new events trigger additional computation [14, 28]. The propagation of updated behavior values and new events is the responsibility of the language.

Figure 2 uses the same timer example to illustrate these concepts. The elapsed time (always has a value, but the value keeps changing) is best represented as a behavior, while clicks on the reset button (may be clicked an arbitrary number of times, but the developer cannot anticipate when it will be clicked next) is best represented as an event stream. `timerB(1000)` creates a behavior that updates every second (i.e., 1000 milliseconds). The `valueNow` method extracts a snapshot of the behavior's value at the time it is invoked (i.e., it does *not* update automatically). `$E` defines an event stream,

```
var nowB = timerB(1000);
var startTm = nowB.valueNow();
var clickTmsB = $E("reset", "click").snapshotE(nowB)
                .startsWith(startTm);
var elapsedB = nowB - clickTmsB;
insertValueB(elapsedB, "curTime", "innerHTML");

<body onload="loader()">
<input id="reset" type="button" value="Reset"/>
<div id="curTime">  </div>
</body>
```

Figure 2. Elapsed Time in Flapjax

in this case one per `click` of the button named `reset`. The result is a stream of DOM event objects. The `snapshotE` method transforms this into a stream of the value—at the time of clicking—of the timer. `startsWith` converts the event stream into a behavior, initialized with `startTm`. Finally, `insertValueB` inserts the value of its behavior into the DOM. (*Appendix A recapitulates all Flapjax operations used in this paper.*)

Obviously, the Flapjax code may not appear any "easier" to a first-time reader. What is salient is both what is and isn't present. What is present is the composition of expressions, even when they involve I/O. A button is no longer just an imperative object; rather, `$E("reset", "click")` lets us treat it as a *value* that can be composed with and transformed by surrounding expressions. What is absent is the callbacks: the developer simply expresses the dependencies between expressions, and leaves it to the language to schedule updates.[2] Thus, `nowB` updates every second, and therefore so does `elapsedB` (which depends on `nowB`) and so does the value on the page (because `elapsedB` is inserted into it).

It is instructive to return to the pure JavaScript version. The bug is that the time displayed on-screen—which we might *think* of as having to always represent the value of `elapsedTime` (i.e., a model-view relationship)—is undefined between when the user clicks on the Reset button and when the timer next fires. This is because the program resets `elapsedTime` but not does propagate it to the screen. The error is subtle to detect during testing because it is a function of when in this interval the user clicks on the button. Put differently, the developer has incurred the burden of synchronizing the on-screen value with the model of time, and even in such a simple program, it is possible to slip up. In contrast, the Flapjax developer has left maintenance of consistency to the language.

In a sense, behaviors should not surprise JavaScript developers. When a program makes a change to the JavaScript DOM, the update is automatically propagated to the screen without the need to notify the browser's renderer. In other

[2] The callbacks for the timer and the button are set up and managed by `timerB` and `$E`. Callback management is discussed in section 3.5.

words, the DOM already acts rather like a behavior! Flapjax thus asks why only the bottommost layer should provide this feature, and instead exposes to developers the same functionality previously reserved for a special case.

2.3 From Buttons to the Network (and Back)

The example above shows that button clicks are event streams and clocks are behaviors. Flapjax makes it possible for developers to treat all the other components of Ajax programs in these terms. The mouse's location is a behavior. A text box can be treated either as a behavior (its current content) or as an event stream (a stream of changes), depending on which is more convenient. A server is simply a function that consumes an event stream of requests and produces an event stream of responses. This view is slowly also being adopted by industry.[3]

As an illustration, consider a component that appears in many Web applications: a text buffer with auto-saving. (Many readers will recognize this from the mail composition buffer of Google Mail.) This is a component we reuse in several of our own applications (section 4). It is therefore instructive to consider its design and implementation. We will build the auto-save buffer incrementally to show how a developer might approach such a problem.

At its simplest, the developer wants to create a text box that saves every `t` seconds:

```
function mkSaveBox(t) {
  var draftBox = // make a <textarea>
  setInterval("...XMLHttpRequest...", t*1000);
  return draftBox; }
```

In fact, however, the buffer should also save whenever the user clicks the Save button. That means the function needs two parameters:

```
function mkSaveBox(t, btn) {
  var draftBox = // make a <textarea>
  // save every t seconds or when btn clicks
  return draftBox; }
```

Now it isn't quite as clear what to do: using `setInterval`, the save callback will run every `t` seconds, but is that what we want? Should it auto-save every `t` seconds regardless of Save clicks, or only `t` seconds after the last Save? Irrespective, should it auto-save even if there are no changes? And then, if some user needs it to auto-save after every keystroke, this abstraction is useless.

The problem is that the abstraction confuses *mechanism*—setting up timers, dispatching messages, and so on—with *policy*. Obtaining this separation is not straightforward in JavaScript and most libraries fail to demonstrate it.

[3] In September 2006, Opera 9 added support for their new standard on *Server-Sent Events*. Though we are critical of some details of their specification, this push from a commercial vendor indicates that these ideas have the potential for broad support.

In Flapjax, the event stream is an excellent representation of policy. Thus, we would write,

```
function mkSaveBox(whenE) {
  var draftBox = // make a <textarea>
  // save every time there is an event on whenE
  return draftBox; }
```

where `whenE` represents the policy. Armed with this abstraction, the concrete policy is entirely up to the developer's imagination; here are three simple examples:

```
mkSaveBox(timerE(60000))
mkSaveBox($E(btn, "click"))
mkSaveBox(mergeE(timerE(60000),
                 $E(btn, "click")))
```

Respectively, these save every minute, every time the user clicks a button, or when either of these occurs.

Now we complete the definition of `mkSaveBox`. First, we create the `<textarea>`:

```
var draftBox = TEXTAREA();
```

`TEXTAREA` creates a new `<textarea>` and exposes it as a behavior. Flapjax defines similar constructors for all the HTML elements.

We now define a simple function that identifies the Web service and marshals the text buffer's contents:

```
function makeRequest(v) {
  return {url: "/saveValue", fields: {value: v},
          request: "post"};}
```

We use Flapjax's `$B` function to obtain a behavior carrying the current value of `draftBox`. (`$B` may be applied to any input element.) When a save event fires on `whenE`, we `snapshot` the current value of `draftBox` and use `makeRequest` to wrap the draft into a request:

```
var requestsE = whenE.snapshotE($B(draftBox))
                    .mapE(makeRequest);
```

Given this event stream of requests we invoke the function `getWebServiceObjectE`, which consumes a stream of server requests and returns a stream of server responses:

```
var savedE = getWebServiceObjectE(requestsE);
```

`getWebServiceObjectE` encapsulates both the call to `XMLHttpRequest` and its callback. When the callback is invoked to indicate that the response is ready, Flapjax fires an event carrying the response.

Presentation The above provides a complete implementation of the auto-save functionality, using event streams to represent the policy. We can go further: in our applications, we have found it valuable for the abstraction to indicate when the buffer is out-of-sync with the server (i.e., between edits and responses). We use a Cascading Style Sheet (CSS) annotation to alter the presentation by changing the editor's border:

```
  var changedE = $B(draftBox).changes();
  styleE = mergeE(changedE.constantE("unsaved"),
                  savedE.constantE("saved"));
  styleB = styleE.startsWith("saved");
  insertValueB(styleB, draftBox, "className");}
```

changes creates an event stream that fires each time the value of $B(draftBox) changes (i.e., on each keystroke). constantE transforms the keyboard events (which indicate the buffer has changed) and the network responses (which indicate that the server has saved) to the strings "unsaved" and "saved" respectively. The merged event stream, styleE, propagates events from both its arguments. We use startsWith (seen earlier in figure 2) to transform the discrete event stream into a continuous behavior, styleB. The value carried by styleB, is the value of the last event, which is the current state of the auto-save buffer. We need to specify an initial value for styleB to hold before the first event fires. Initially, the (empty) buffer is effectively "saved". This behavior is inserted into the DOM as the CSS className; CSS class entries for saved and unsaved will correspondingly alter the box's appearance.

2.4 Higher-Order Event Streams: Drag-and-Drop

So far, we have seen event streams of keystrokes and network requests. Flapjax events may, however, represent arbitrary actions. In particular, they can represent events much more complex than those exposed by the DOM. To illustrate this, we build a drag-and-drop event abstraction.

For simplicity, consider dragging and dropping a box:

```
<div id="target"
     style="position: absolute;
            border: 1px solid black">
Drag this box
</div>
```

The DOM provides mouseup, mousedown and mousemove events for each element. A drag-and-drop operation begins with a mousedown, followed by a sequence of mousemoves, and ends with mouseup.

We wish to define a function that, given an element, produces an event stream of drags and a drop:[4]

```
dragE :: element -> EventStream (drag or drop)
```

Both drag and drop events are a record of three fields. The fields left and top are the mouse coordinates. The third field, drag or drop, carries the element being manipulated.

We begin with mousemove, which naturally leads to the creation of drag events:

```
function dragE(elt) {
  return $E(elt,"mousemove").mapE(
    function(mm) {
      return { drag: elt,
               left: mm.clientX,
               top: mm.clientY };});}
```

[4] Throughout this paper, we specify each function's interface using a Haskell-like type annotation. Because JavaScript is a latently typed language, these should be regarded as comments, though they could be enforced by a static type checker; in Flapjax, they are enforced using contracts (section 5).

The function above is permanently stuck dragging. We should start responding to mousemove events only after we register a mousedown event:

```
return $E(elt,"mousedown").mapE(
  function(md) {
    return $E(elt,"mousemove").mapE(
      function(mm) {
        return { drag: elt,
                 left: mm.clientX,
                 top: mm.clientY }})});
```

Above, the mousemove event stream is created only after an enclosing mousedown event fires. In fact, each mousedown produces a new stream of mousemove events.

This code appears to have a runtime type error: it produces an event stream of event streams. Such higher-order event streams are in fact perfectly legal and semantically sound. The problem is that dragE ultimately needs the coordinates of the *latest* inner event stream (the latest drag sequence). To flatten higher-order streams, Flapjax offers the primitive:

```
switchE :: EventStream (EventStream a)
        -> EventStream a
```

switchE fires events from the *latest* inner event stream. With switchE, we can easily fix our type error:

```
var moveEE = $E(elt,"mousedown")
  .mapE(function(md) { ... as before ...
return moveEE.switchE();
```

We have not accounted for drop events, which should "turn off" the stream of drag events. We thus map over the stream of mouseup events and return a singleton drop event:

```
var moveEE = ... as before ...
var dropEE = $E(elt,"mouseup")
  .mapE(function(mu) {
    return oneE({ drop: elt,
                  left: mu.clientX,
                  top: mu.clientY })});
```

We can combine these two event streams with mergeE, which fires events from either of its arguments:

```
return mergeE(moveEE,dropEE).switchE();
```

Because switchE fires events from the latest inner event stream, when the mouse button is pressed, moveEE produces an event stream of drags. This becomes the latest inner event stream, and switchE thus produces a stream of drags.

When the mouse button is released, dropEE produces a new event stream (oneE({ drop ... })). When this new event stream arrives, `switchE` stops forwarding `drag` events from the previous event stream. It fires the single `drop` event and waits for more (in this case, we know there is just one drop event).

When the mouse button is pressed again, `moveEE` produces a new stream of `drag`s that supersede the earlier stream from `dropEE`. This abstraction therefore lets us drag the box repeatedly.

Using Drag-and-Drop The `dragE` function merely reports the position where the target is dragged and dropped. It does not, as one might expect, actually move the target. This omission is intentional. We can easily move the target when it is dragged:

```
var posE = dragE("target");
insertValueE(posE.mapE(function(p)
                {return p.left}),
              "target","style","left");
insertValueE(posE.mapE(function(p)
                {return p.top}),
              "target","style","top");
```

However, by separating the drag-and-drop event stream from the action of moving the element, we've enabled a variety of alternate actions. For example, the following action ignores the `drag` events and immediately moves the target when it is dropped:

```
insertValueE(
  posE.filterE(function(p) {return p.drop;})
      .mapE(function(p) { return p.left;}),
  "target","style","left");
```

In the next example, the target moves continuously but lags behind the mouse by 1 second:

```
insertValueE(
  posE.delayE(1000)
      .mapE(function(p) {return p.left;}),
  "target","style","left");
```

Further possibilities include confining the drag area, aborting drag operations, etc. Our example has omitted a start-drag event, which opens up a range of new uses. The reader might wish to also compare our approach to that of Arrowlets [22], which we discuss in section 6.

2.5 Compositional Interfaces: Building Filters

We built drag-and-drop by combining event streams that were extracted from a single DOM element. Flapjax also allows behaviors to be built from multiple, independently-updating sources. We will illustrate this with an example taken from Resume, an application we discuss in section 4.

Resume presents reviewers with a list of a job candidates. A large list is unusable without support for filtering and sorting. Figure 3 defines two possible filters for selecting candidates: by sex and by score.

```
function pickSex() {
  var ui = SELECT(
    OPTION({ value: "Female" }, "Female"),
    OPTION({ value: "Male" }, "Male"));

  return {
    dom: ui,
    pred: function(person) {
     return person.sex == $B(ui);
}};}

function pickScore() {
  var ui = INPUT({ type: "text", size: 5 });

  return {
    dom: ui,
    pred: function(person) {
     return person.score == $B(ui);
}};}
```

Figure 3. Filters for Single Criteria

```
function pickFilter(filters) {
  var options = mapKeys(function(k, _) {
      return OPTION({ value: k }, k);},
    filters);
  var sel = SELECT(options);
  var subFilter = filters[$B(sel)]();

  return {
    dom: SPAN(sel, " is ", subFilter.dom),
    pred: subFilter.pred };};
```

Figure 4. Selecting Filters

Each function returns both the interface for the filter and the predicate that defines the filter. The interface elements are built using Flapjax's constructors, such as `SELECT` and `INPUT`, which construct behaviors (just like `TEXTAREA` in section 2.3). We can display these DOM behaviors with `insertDomB`:

```
var filterObj = pickScore();
insertDomB(filterObj.dom,"filterDiv");
```

We can use the predicate to filter an array of candidates:

```
var filteredCandidates =
  filter(filterObj.pred,candidates);
```

Observe in `pickScore` that `$B(ui)` is a behavior dependent on the current score. The value of `filteredCandidates` thus updates automatically as the user changes their desired score. We can then `map` over the list of candidates, transforming each candidate object to a string. The resulting strings can be inserted into the DOM. Flapjax tracks these data dependencies and automatically keeps them consistent.

```
function modalFilter(subFilter) {
  var button = A({ href: "" }, "Update");
  var subPred = subFilter.pred;

  return {
    dom: DIV(subFilter.dom, button),
    pred: $E(button, "click")
          .snapshotE(subPred)
          .startsWith(subPred.valueNow()) };};
```

Figure 5. Filters with an Update Button

Real systems may have many available filters. If the user wants only one at any given time, displaying them all would clutter the screen. We might instead indicate the available filters in a drop-down box, and show only the controls for the selected filter.

Figure 4 implements a filter selector. It chooses between filters of the form in figure 3, where each filter is an object with `dom` and `pred` fields (of the appropriate type). The result of `pickFilter` is also an object of the same type.

The function is parameterized over a dictionary of available filters; for example:

```
basicFilters = {
  "Sex": pickSex,
  "Score": pickScore };
var filterObj = pickFilter(basicFilters);
```

We build the drop-down box (`sel`) by mapping over the names of the filters (`options`); we elide `mapKeys`—it maps a function over the key-value pairs of an object. The `pickFilter` interface displays `sel` and the interface for the selected filter (`subFilter.dom`). The predicate for `pickFilter` is that of the selected filter.

When the user chooses a different filter, `$B(sel)` updates, and so does `subFilter.dom`. Since the DOM depends on `subFilter.dom`, Flapjax automatically removes the interface of the old filter and replaces it with that of the new one. The developer does not need to engineer the DOM update. The filtering predicate (`subFilter.pred`) updates similarly, changing any displayed results that depend on it.

These filters update the list immediately when the user makes a selection in the filtering GUI. An alternate, modal interface would not affect the list until an Update button is clicked. We can reuse our existing filtering abstractions for modal filters. To do so, we build the interface by composing filters, exactly as we did in figure 4. When we're done, we apply `modalFilter` (figure 5) to add an Update button:

```
var filterObj =
  modalFilter(pickFilter(basicFilters));
```

As the user makes selections in the filtering interface, `subPred` continuously updates in `modalFilter`. However, `modalFilter`'s predicate is not the current value of `subPred`, but a snapshot of its value when Update was last clicked. As a result, we get a modal filtering interface.

```
// flickrSearchRequest :: String -> Request
// Packages the search text into a Flickr API call.
function flickrSearchRequest(req) { ... }

// flickrSearchResponse :: Response -> Listof(Url)
// Extracts URLs from a Flickr API response.
function flickrSearchResponse(resp) { ... }

// makeImg :: Url -> Element
function makeImg(url) {
  return IMG({ src: url }); }

var queryE = $B("search").changes().calmE(1000);
var requestE = queryE.mapE(flickrSearchRequest);
var responseE = getForeignWebServiceObjectE(requestE)
 .mapE(flickrSearchResponse);
var imgs =
  DIV(map(makeImg, responseE.startsWith([])));

insertDomB(imgs, "thumbs");
```

Figure 6. Flickr Thumbnail Viewer

By preserving the interface to a filter at each level, we obtain a variety of substitutable components. For example, `filterObj` may be any one of:

```
filterObj = pickScore();
filterObj = pickSex();
filterObj = pickFilter(
  { "Sex": pickSex, "Score": pickScore });
filterObj =
  modalFilter(pickFilter(
    { "Sex": pickSex, "Score": pickScore }));
```

Regardless of the definition of `filterObj`, the code to apply and display filters does not need to change:

```
insertDomB(filterObj.dom, "filterDiv");
var filteredCandidates =
  filter(filterObj.pred, candidates);
```

In Resume, we have even more general combinators such as the conjunction and disjunction of multiple filtering options. Though it is unrelated to the notion of filtering itself, this idea of bundling interface with behavior is strongly reminiscent of Formlets [9], which we discuss in section 6.

2.6 Pipes (and Tubes) for Web Services

As Unix showed many decades ago, pipelines are good component connectors. The `getForeignWebServiceObjectE` primitive extends this to the Web. Suppose, for instance, the developer wants to erect a pipeline from a text box to a Web service to the screen. Assuming the HTML document contains an input box with id `search` and a presentation element with id `thumbs`, the program in figure 6 extracts each

```
//    EventStream {data: a, loc: String }
// -> EventStream {data: a, point: Point or false}
function makeGoogleGeocoderE(requestE) {
  var geocoder = new google.maps.ClientGeocoder();
  var resultE = receiverE(); // primitive stream

  var callback = function(d) {
    return function(p) { resultE.sendEvent(
      { data: d, point: p })}};

  requestE.mapE(function(req) {
    geocoder.getLatLng(req.loc,
      callback(req.data));});

  return resultE;};
```

Figure 7. Geocoder Service as Event Stream Transformer

query typed into the search box, sends the query to the photo sharing site `flickr.com`, obtains a list of thumbnails, and displays them in the DOM.

In the definition of `queryE`, `calmE` builds a "muted" event stream for a given time period. By calming an event stream associated with a buffer's keystrokes for a second, the developer can keep the system from responding to every keystroke, waiting for a pause when the user is not typing. We use this method frequently to provide smoother user interfaces.

2.7 Mashups: Composing Web Services

If Web services expose themselves as consumers and producers of event streams, they naturally fit the Flapjax mold. However, many external Web services are not designed this way. Some such as Twitter (`www.twitter.com`, which lets users broadcast short messages called *tweets*) return responses containing JavaScript code that must be `eval`'d to obtain a local callback. Others, such as Google Maps (`maps.google.com`), supply extensive (callback-based) libraries. However, with just a little effort, these callback-based APIs can be turned into event stream transformers that fit naturally into Flapjax applications. We first show this adaptation, then use it to build a mashup of Twitter and Google Maps.

Google Geocoder The Google Geocoder is a part of the Google Maps API. It accepts the name of a location (e.g., "Providence, RI") and, if it successfully interprets the name, returns its latitude and longitude. The lookup occurs asynchronously on Google's servers, so it is unsurprising that the Geocoder function uses a callback:

```
getLatLng :: String * (Point -> void) -> void
```

whose use tends to follow this template:

```
var data = ...
var callback = function(p) { ... };
getLatLng(data.location,callback);
```

An application that uses `getLatLng` continuously needs to associate points returned in the callback with data about the corresponding request; we can encapsulate in a closure:

```
var callback = function(d) {
  return function(p) { ... }};
getLatLng(data.location,callback(data));
```

We will package this pattern into an event stream transformer from strings to points, along with an arbitrary datum that is passed from each request to its associated result:

```
   EventStream { data: a, loc: String }
-> EventStream { data: a, point: Point }
```

We can easily map over a stream of requests:

```
function makeGoogleGeocoderE(requestE) {
  var callback = ...;
  requestE.mapE(function(req) {
    getLatLng(req.loc,callback(req.data)); });;}
```

However, the result is not available within the body of `function(req) { ... }`, so the event stream above does not produce meaningful events. Since results arrive asynchronously, they are conceptually a new event stream. The operation `receiverE` creates a primitive event stream with no sources, so it does not fire any events:

```
var resultE = receiverE();
var callback = ...;
requestsE.mapE(...);
return resultE;
```

However, we can imperatively push an event to it using `sendEvent`. Since points are sent to the callback, the body of our callback becomes:

```
resultE.sendEvent({ data: d, point: p });
```

The complete function is shown in figure 7. It exposes itself as a pure Flapjax event transformer, completely hiding the internal callback. Using this pattern, we can erect a similar reactive interface—which can then be treated compositionally—to any Web service with a callback-based API [20].

Twitter/Google Maps Mashup Now that we've seen how callback-based Web services can be turned into event stream transformers, we can combine Web services into a mashup. Consider a simple mashup that takes Twitter's most recent public tweets and plots them on a Google Map. This mashup operates in three steps: (1) Fetch the live feed of public tweets from Twitter. Tweets are accompanied by the location (e.g., "Providence, RI") of their sender. (2) Using the Google Maps Geocoder API, transform these locations into latitudes and longitudes. (3) If the Geocoder recognizes the location, plot the corresponding tweet on an embedded Google Map. Since we have a live feed, repeat forever.

Figure 7 shows the code for the Geocoder event stream. We can similarly build a public tweet event stream:

```
var googleMap = new google.maps
  .Map2(document.getElementById("map"));
googleMap.setCenter(
  new google.maps.LatLng(0, 0), 2);

// Fetch the live feed of public tweets
var tweetE = getTwitterPublicTweetsE();

// Transform locations into coordinates
var pointsE = makeGoogleGeocoderE(
  tweetE.mapE(function(tweet) {
    return { data: tweet.text,
             location: tweet.user.location };}));

// Elide points the Geocoder did not recognize
makeMapOverlayE(googleMap,
  pointsE.filterE(function(x) {
    return x.point != false; }));
```

Figure 8. Mashup of Twitter and Google Maps

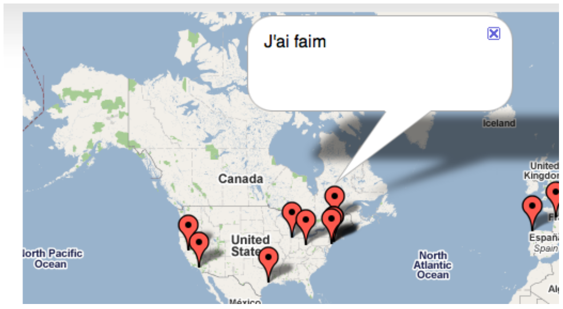

Figure 9. Clicking on a Pin Displays Tweet

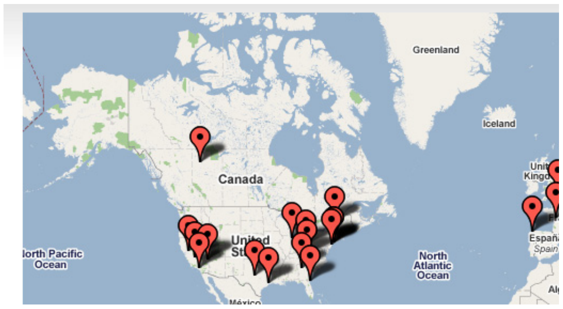

Figure 10. After a While, More Pins Appear

```
getTwitterPublicTweetsE ::
  -> EventStream Tweet
```

We are using Google Maps to plot points returned by the Geocoder. It thus suffices to build an event steam consumer:

```
makeMapOverlayE ::
     GoogleMap * EventStream { data: String,
                               point: Point }
  -> void
```

In the interest of space, we elide the definitions of these functions. However, they are similar in length and in spirit to the Geocoder function.

Figure 8 shows the code for our mashup (figure 9 and figure 10 show two instances of its execution). Aside from the initialization of the embedded map, the mashup is almost directly a transcription of strategy outlined in prose above. Furthermore, it employs reusable abstractions that other mashups can also share.

2.8 From Web Services to Persistent Objects

Web services are good abstractions for procedural and message-passing interfaces, such as for finding a list of movies playing on a particular evening. They do not, however, directly model shared, mutable, persistent objects, such as a meeting in a group calendar. The Flapjax system provides a custom persistent object store to save such objects, and a client-side library to interface with it.[5]

The server maintains notions of identity for users and applications (with standard authentication details). It presents each application with a filesystem-like tree, which developers can also browse through a trusted Web interface. The operation `writePersistentObject` associates an event stream with a location—which is a path through the tree—while `readPersistentObject` reflects the values stored at that location into a stream. Thus, referring to the draft-saver from section 2.3:

```
writePersistentObject(draftBox.changes(),
                      ["draft"]);
```

saves drafts at the top-level "file" `"draft"`. In practice, an application will wrap `writePersistentObject` in an abstraction. This can be used to erect another policy-mechanism separation: `writePersistentObject` is a mechanism, but various filters can be applied to the event stream it consumes to perform, e.g., rate limiting. `readPersistentObject` reverses the direction of binding to reflect persistent objects in the application, with extra parameters for the initial value and polling rate.

Access Control In keeping with the filesystem analogy, every persistent object is subject to an access-control policy. Naturally, this set of permissions can also change at any time. User interface elements that depend on the permissions should also update their appearance or behavior.

The Flapjax primitive `readPermissionsB` produces a behavior representing the current permissions of a location in the persistent store. The application can use its value to drive the user interface. For instance, if `permsB` is bound to the permission of a store location,

```
INPUT({type: "text",
       disabled: !(permsB.has("WRITE", true))});
```

[5] Flapjax applications do not have to use the object store. This is a proof-of-concept server.

```
function EventStream(sources,update) {
  this.sources = sources;
  this.sinks = [ ]; // filled in by sources
  for (var i = 0; i < sources.length; i++) {
    this.sources[i].sinks.push(this); }
  this.update = update; }
```

Figure 11. The Event Stream Constructor

declaratively ties the disabling of the input box with lack of write permission. Thus, the user interface will automatically update in step with changing permissions.

3. Implementation

The theory of Flapjax is rooted in signal processing: events and behaviors are essentially signal-processing abstractions. The theoretical underpinnings of Flapjax can be found in Cooper's dissertation [10]. Here, we focus on the implementation strategy that enables the above programs to run.

3.1 The Evaluation Model

The central idea behind Flapjax is push-driven dataflow evaluation. Flapjax converts JavaScript programs into dataflow graphs. Dataflow graphs are mostly-acyclic directed graphs from sources (clocks, user inputs, the network, etc.) to sinks (screen, network, etc.). When an event occurs at a source, Flapjax "pushes" its value through the graph. A graph node represents a computation; when it receives an event, a, it applies a function f (representing the computation) to a and may further propagate $f(a)$ to its children. This push-based, demand-driven evaluation strategy is a good match for systems with many kinds of external stimuli that do not obey a single central clocking strategy. We discuss other strategies in section 6.

3.2 Dataflow Graph Construction

Though we have presented event streams and behaviors as distinct entities, the astute reader will have guessed that they are almost duals of each other. Given a behavior, issuing an event whenever its value changes yields a corresponding event stream. Given an event stream and an initial value, continuously yielding the most recent value on the stream (and the initial value before the first event appears) gives a corresponding behavior. In Flapjax, nodes in the dataflow graphs are event streams while behaviors are derived objects. We describe the construction of a dataflow graph of event streams below.

To developers, an event stream is an abstract data type that may only be manipulated with event stream combinators (e.g., `mapE`, `calmE`, etc). Internally, an event stream node is implemented as an object with three fields:

```
sources :: listof(EventStream)
sinks :: listof(EventStream)
update :: a -> (b or StopValue)
```

```
// EventStream a * EventStream a -> EventStream a
function mergeE(src1,src2) {
  var update = function(a) {
    return a; }
  return new EventStream([src1,src2],update); }

// (a -> b) * EventStream a -> EventStream b
function mapE(f,src) {
  var update = f;
  return new EventStream([src],update); }

// (a -> Bool) * EventStream a -> EventStream a
function filterE(pred,src) {
  var update = function(a) {
    if (pred(a)) {
      return a; }
    else {
      return StopValue; }};
  return new EventStream([src],update); }

// EventStream (EventStream a) -> EventStream a
function switchE(srcE) {
  var outE = new EventStream([], function(a) {
    return a; });

  var prevE = null;

  var inE = new EventStream([srcE], function(aE) {
    if (prevE) {
      outE.sources.remove(prevE);
      prevE.sinks.remove(outE); }
    prevE = aE;
    outE.sources.push(aE);
    aE.sinks.push(outE);
    return StopValue; });

  return outE; }
```

Figure 12. Implementation of Event Stream Combinators

`sources` and `sinks` specify a node's position in the graph. When a value (of type `a`) is pushed to a node, Flapjax applies the node's `update` function to the value. If `update` returns the `StopValue` sentinel, the value does not propagate further from the node. Otherwise the result, `b`, is propagated further by Flapjax's evaluator, as described in section 3.4.

Consider `mergeE`, which builds a node that propagates all values from both its sources without transforming them:

```
merged = mergeE(src1,src2)
```

`mergeE` must build a new node by specifying the sources, sinks, and updater. The sources are the event streams `src1` and `src2`. The update function propagates all values:

```
function update(a) { return a; }
```

Since Flapjax is push-driven, the node bound to `merged` must know the sinks to which it pushes values. However, we can defer specifying the sinks until `merged` is used as a source. To consistently follow this pattern, `mergeE` must set the node bound to `merged` as a sink for `src1` and `src2`. We abstract this pattern into the `EventStream` constructor (figure 11), which only requires `sources` and `update` as arguments. `mergeE` and other event stream combinators (figure 12) are therefore pure JavaScript functions, and so are the `update` functions. This ensures that individual expressions do not change their meaning relative to JavaScript, a problem that might ensue if we wrote a specialized interpreter to implement the dataflow evaluation strategy.

Derived Behaviors A behavior in Flapjax is an extension of event streams. A behavior node maintains its current value, in addition to sources, sinks, and an update function. The initial current value is an additional parameter of the behavior constructor. The update function computes a new value n; if n is different from the current value it sets n as the current value and propagates to all sinks, otherwise it returns `StopValue` to prevent further propagation.

3.3 The Compiler

We mentioned in section 1 that Flapjax can be viewed as a language or, with a little extra work, a library. Now we can explain precisely what this extra work is.

The compiler consumes files containing HTML, JavaScript and Flapjax. Flapjax code is identified by the

```
<script type="text/flapjax">
```

directive. The compiler transforms Flapjax code into JavaScript and produces standard Web pages containing just HTML and JavaScript. It includes the Flapjax library and elaborates Flapjax source code to call library functions as necessary. Flapjax source code has JavaScript's syntax, but the compiler's elaboration gives it a reactive semantics. Furthermore, the compiler allows Flapjax and JavaScript code to seamlessly interoperate. This strategy of *transparent reactivity* [11] has great advantage for beginning users and in teaching contexts. We outline its main components below.

Implicit Lifting The principal task of the compiler is to automatically *lift* functions and operators to work over behaviors. The compiler does so by transforming function applications to invocations of `liftB`. This allows us to write expressions such as

```
timerB(1000) + 1
```

where JavaScript's + operator is applied to the `timerB(1000)` behavior. The compiler transforms the expression above to code equivalent to:

```
liftB(function(t) { return t + 1; },
      timerB(1000))
```

The function `liftB` creates a node in the dataflow graph with `timerB(1000)` as the source and

```
function(t) { return t+1 }
```

as the update function.

Without the compiler, such transformations must be performed manually. In practice, this appears to be less onerous than it sounds, as our experience suggests (section 4). Without the compiler, the development cycle involves just editing code and refreshing the page in the browser. The compiler introduces another step in the development cycle which may sometimes be inconvenient.

JavaScript Interoperability The compiler enables Flapjax code to interoperate with raw JavaScript. Flapjax already shares JavaScript's namespace, so either language can readily use identifiers defined in the other. Consider Flapjax calling JavaScript functions. In the simplest case, if a JavaScript function is applied to behaviors, the compiler can lift the application. The JavaScript function is thus applied to the values carried by the behaviors, rather than the behaviors themselves (which it presumably would not comprehend). Whenever a behavior changes, the function is reapplied.

For example, suppose `filter` is defined in JavaScript:

```
filter :: (a -> Bool) * listof(a) -> listof(a)
```

and consider the following Flapjax code:

```
var tock = filter(function(x) {
  return (x % 2) == timerB(1000) % 2;
  }, [0,1]);
```

`filter` expects a predicate and a list. The result of `tock`, however, is not a boolean, but a behavior carrying a boolean. The compiler thus wraps the predicate so that the current value of its result is extracted on application. Furthermore, when the result is invalidated (as it is every second), `filter` is reapplied. As a result, `tock` is a behavior that alternates between `[0]` and `[1]`.

In the other direction—JavaScript calling Flapjax code—the compiler performs no transformations. There is no need to do so, since the natural way to invoke Flapjax from JavaScript is to treat it as a library with explicit calls to the event stream and behavior combinators.

Inline Flapjax The compiler provides one more convenience that we have not yet discussed in this paper, called *inline Flapjax*. It recognizes the special matching tokens `{!` and `!}` [6] in any HTML context and treats the text contained within as Flapjax code. This code is expected to evaluate to a behavior. The compiler inserts the behavior into the document at that point without the need for additional code.

For instance, given some function `validCC` that validates a credit card number and these JavaScript declarations—

```
function validColor(valid) {
  return valid ? "aqua" : "cyan"; }
var ccNumField = $B("ccNum");
var ccNumValid = validCC(ccNumField);
```

—this inline Flapjax term

[6] Pronounced "curly-bang".

```
<input id="name"
  style={! { borderColor:
             validColor(ccNumValid)} !}
  disabled={! !ccNumValid !}/>
```

creates an input element that is enabled or disabled depending on the validity of the credit card number, and whose border color is correspondingly aqua or cyan. In particular, the JavaScript object `{ borderColor: ... }` is automatically converted into a CSS style specification string.

While inline expressions can become unwieldy in general, we find them especially useful for writing program expressions such as the validator above. In particular, when the validity of an element depends on several data, it is easier and clearer to express this as a localized functional dependency rather than diffuse it into callbacks, which inverts the dependency structure.

3.4 Propagation

The recursive propagation model (which is implemented with trampolining [31], to prevent stack overflow) requires additional explanation, particularly to prevent some undesirable behavior. Consider the following expressions, where `y` is some numeric behavior:

```
var a = y + 0;
var b = y + a;
var c = b + 1;
var d = c % 2;
```

We would expect `b` to always be twice `y`, `c` to be odd, `d` to be `1`, and so on. Unfortunately, nothing in our description above guarantees this. The update from `y` may recompute `b` before it recomputes `a`, which might trigger the subsequent recomputations, resulting in all the invariants above being invalidated. Of course, once the value of `a` updates, the invariants are restored. This temporary disruption is called a *glitch* in signal-processing lingo. (There is another, subtle problem above: some nodes may be computed more than once. Not only is this wasteful, this computation may be noticed in case the updater functions have side-effects.)

Fortunately, there is a simple solution: to use topological order. This prevents nodes from being evaluated before they should, and avoids repeated computation. To perform this, the node data structure tracks its partial-order rank in the dataflow graph. The Flapjax evaluator calls nodes' update functions (figure 12) in topological order. Instead of propagating values immediately, the evaluator inserts them into a priority queue in topological order.

The only obstacle to topological ordering is the presence of cycles in the graph. Cyclic dependencies without delays would, however, be ill-defined. Flapjax therefore expects that every cycle is broken by at least one delay (either the primitive `delayE` or a higher-level procedure, such as integration, that employs delays). This restores the delay-free sub-graph to a partial order.

3.5 Primitive Event Streams and Callbacks

Flapjax programs do not directly use callbacks, so the Flapjax library encapsulates callback management code. The simplest function that encapsulates a callback is `$E`, which encapsulates a DOM callback and exposes it as an event stream of DOM events. For example:

```
var moveE = $E(document.body, "mousemove");
```

As we showed for the Geocoder (figure 7), we can use `sendEvent` to encapsulate the callback-based DOM API:

```
function $E(elt, evt) {
  var stream = receiverE();
  var callback = function(e) {
    stream.sendEvent(e); };
  elt.addEventListener(evt, callback);
  return stream; }
```

However, `$E` above never removes the callback. (It does not call the DOM function `removeEventListener`.) If the event stream `moveE` becomes unreachable from the rest of the program, we may expect it to be garbage collected. However, `addEventListener` creates an internal reference from `elt` to `callback`. Therefore, as long as `elt` is reachable, the event stream fires in perpetuity.

This scenario occurs when we have higher-order event streams, such as the drag-and-drop example (section 2.4), where a new stream of `mousemove` events is created for each `mousedown` event. `switchE` makes the previous `mousemove` unreachable by the program, though the element being dragged keeps a reference to the `mousemove` callback. Since the element is never removed from the DOM, the previous `mousemove` callback is continually invoked. After a number of drag operations, the browser becomes noticeably slower.

We solve this issue by adding an `isDetached` field to all event streams. If `isDetached` is set, the callback in `$E` removes itself, instead of propagating the event:

```
function $E(elt, evt) {
  var stream = receiverE();
  var callback = function(e) {
    if (stream.isDetached) {
      elt.removeEventListener(evt, callback); }
    else {
      stream.sendEvent(e); }};
  elt.addEventListener(evt, callback);
  return stream; }
```

`isDetached` is initialized to `false`. In the definition of `switchE` (figure 12), we set `prevE.isDetached = true` when removing `prevE` from the dataflow graph.

However, `prevE` may not itself be a `$E` expression, so the `isDetached` flag of its sources must be updated as well. For any event stream `isDetached` is set if all its sinks are detached. We compute this expression while propagating values (section 3.4).

3.6 Library Design

The Flapjax library is unusual in that it serves both the compiler (as its runtime system) and developers using it directly. In both capacities it needs to be efficient; the latter is unusual as most languages' runtime systems are not directly used by developers. Here we discuss some design decisions that have proven important over several years of use.

Functions versus Objects Flapjax encourages making Web applications more functional in style. This can, however, lead to deeply nested function applications, which are syntactically alien to many JavaScript developers. We have therefore found it convenient to make all the standard functions available as methods in the `Behavior` and `EventStream` prototypes. This means that instead of

```
var name = calmE(changes($B("name")), 300);
```

developers can write

```
var name = $B("name").changes().calmE(300);
```

which is arguably more readable than standard functional notation, since the left-to-right order of operations corresponds to the direction of dataflow. We do offer all these operations as standard functions also, so developers can use whichever style they favor.

Lifting Constants The compiler inserts behavior combinators automatically. To aid developers who do not use the compiler, Flapjax's behavior combinators lift constant arguments to constant behaviors; this does not require compiler support. For example, the type of `timerB` is

```
timerB :: Behavior Int -> Behavior Int
```

so that the interval may itself vary over time. Library users may, however, simply write `timerB(1000)`. The function will treat `1000` as a constant behavior.

Element Addressing The library includes many DOM manipulation functions that consume HTML elements. There are many ways for JavaScript functions to acquire DOM elements; one of the more common techniques is to give them a name (an `id`). All functions that consume elements also accept strings that name elements.

Reactive* DOM *Elements Developers may be concerned about the cost of Flapjax's reactive element constructors. Perhaps hand-coding imperative DOM updates would be significantly faster?

Indeed, a naive implementation of a constructor would rebuild the entire element on any update. For example,

```
DIV(timerB(1000))
```

might construct a new `<div>` every second. Our implementation updates changes in-place, so only one `<div>` is constructed, but its text updates every second. This strategy significantly ameliorates such efficiency concerns.

4. Evaluation

All the Flapjax code above is real. Developers can run the Flapjax compiler on Flapjax code to generate pure JavaScript applications which can then be deployed. As a result, all these programs execute on stock browsers.

Flapjax has been public since October 2006, and has been used by several third-parties (i.e., non-authors) to build working applications:

Data Grid The Data Grid application[7] was developed by a commercial consulting firm based in London. They were able to successfully construct their application in the language, and reported that it resulted in "shorter code and a faster development cycle". They did identify places where the implementation could be faster, an issue we discuss in greater detail below. They found that one fringe benefit of using Flapjax was that it insulated developers "from most of the cross-browser issues", which are considerable in JavaScript, due to notoriously poor standardization of Web technologies.

Interactive Wiki Another group used Flapjax to build a Wiki system that updates on-the-fly. Their evaluation largely concurred with that of the Data Grid developers. They added that behaviors, while a convenient and intuitive abstraction, could cause a performance hit while initializing an application: the initial values are better defined statically with HTML when possible. (We conjecture that this is at least partially due to the Flapjax implementation's event-orientation, with behaviors treated as a derived type.) Thus, much of the computation in their Wiki system is done in terms of events.

Network Monitor Another developer has used Flapjax to construct a network monitoring client. A network of workstations each expose status information as a Web service; the monitor obtains this information as event streams and collates and combines them to present individual and collective status information.

In addition there are several applications that we have built ourselves. One example is TestFest, which enables students to separately upload their homework programs and test cases; each student's test is run against every other student's homework. This application has been used for two years at Brown and at another university.

More significantly, we have written and deployed Resume (`resume.cs.brown.edu`) and Continue 2.0 (`continue2.cs.brown.edu`). Resume is a program for managing the application and review process for academic jobs; Continue is a conference paper manager. Though these sound similar, the two workflows—and hence applications—are quite different. Both applications are in daily use. Resume has been used for job searches for three years in multiple academic

[7] `http://www.untyped.com/untyping/2007/01/19/flapjax-in-action/`

departments, and has been solicited by others who noticed it while submitting letters. Continue has been used by over twenty-five workshops and conferences.

Both applications use Flapjax as a library; they feature fewer than 10 `lifts` per KLOC, suggesting that eschewing the compiler to use the library directly is not a major impediment. With event-streams and reactivity, it was easy to reproduce and cleanly encapsulate the kinds of features users are accustomed to seeing in commercial applications. For instance, the auto-save buffer example of section 2.3 is inspired by that of Google Mail; in our applications, it is used to save reviews and comments. Searching also reacts to keystrokes without the need for a Search button, just as in applications like iTunes. We also use reactivity to (judiciously) affect styling to notify users of unsaved data.

Performance It is impossible to measure the "performance" of a language; we can only measure the performance of individual programs. The problem is exacerbated when programs are highly interactive, because performance is equally a function of the nature of inputs used to drive applications. It is, nevertheless, worth asking what impact the Flapjax abstractions have on program performance. We answer this question at several levels.

At the highest level, we argue that Flapjax simply automates much of the work that a JavaScript developer would have done by hand: to propagate values through computations and keep them consistent. A developer can certainly use the full force of human knowledge to short-circuit some evaluation; however, these same attempts often produce inconsistent or erroneous Web applications in practice.

The most significant cost in Flapjax is from scheduling in the dataflow graph. This breaks what might have been one large call-by-value evaluation into several small call-by-value fragments interspersed by dataflow graph manipulation and traversal. We have used several (sound) heuristics to eliminate constants and to compact chains of nodes into single nodes, inspired by the lowering work of Burchett, et al. [4]. Because these have yielded reasonable performance we have not investigated this topic further, but there is considerable opportunity for performance improvement. Qualitatively, we have used Flapjax to develop several animations and games. Flapjax smoothly renders these programs. In addition, Resume and Continue have full-featured GUIs built entirely in Flapjax. These systems have been tested and used successfully with hundreds of records displayed on screen. Recent advances in the performance of commercial JavaScript evaluators have made the runtime cost of dataflow evaluation negligible.

While dataflow evaluation might slow down programs, it also has the potential to speed them up through parallel execution. One of the major obstacles to parallelism is the use of unfettered, dependency-creating side-effects; these are precisely what good Flapjax programming style eliminates. Indeed, David Patterson's plenary speech [29] at the International Symposium on Low Power Electronics and Design (2007) outlined a research project at Berkeley that exploits Flapjax for parallelism [21].

Finally, as mentioned above, Flapjax is a working language. In particular, each of Resume and Continue 2.0 is accompanied by a *demo mode*, which automatically creates an instance of a job search or conference, respectively, and lets the user experiment with the program, without need for an account. Readers are invited to try these applications for themselves to get a feel for how much perceived overhead dataflow evaluation might cause.[8]

5. Perspective

Our design choices in Flapjax raise a variety of interesting issues. We discuss these below.

Consistency as a Linguistic Primitive One of the central goals of Flapjax is to explore the idea of consistency as a linguistic primitive. The motivation for the dataflow evaluation model is to enable propagation of updates, and the complexities of propagation (section 3.4) are to make this notion semantically sensible. Our propagation algorithm ensures that the developer never sees a value that is inconsistent with the text of the program. This means developers can reason algebraically about their programs (a task simplified because programs tend to become much less imperative)—for instance, they can refactor their program using algebraic reasoning, while knowing that so long as they preserved the algebraic meaning, the program's behavior will not change upon execution.

We view consistency as analogous to garbage collection: a sensible requirement that is so pervasive that languages should try to support and optimize it. As with garbage collection, developers do sometimes need to manually inject behavior. For instance, `valueNow` samples a behavior at a particular instant; similarly, `snapshotE` samples a behavior at the instant an event fires (section 2.2). We find that the number of uses of these primitives is small: 14 and 19, respectively, in Continue, and 19 and 10, respectively, in Resume. This is from about 4.3 KLOC of Continue and 2.3 KLOC of Resume, including about 1 KLOC of shared code. This suggests that consistency is indeed the right default. It would be interesting to study traditional JavaScript codebases to determine how much programming effort is expended in the other direction: to obtain what Flapjax provides intrinsically.

Security We have not discussed security, which is a pervasive concern in Web applications, beyond access control (section 2.8). This is partly intentional: security means so many different things in this context (avoiding cross-

[8] In demo mode, the application runs in one frame while the demo program—running in the other frame—points to elements of the first and suggests what the user might click on. The demo advances by the interactions of the user with the application under demonstration. In other words, the demo framework is itself a reactive application. Naturally, it too is written in Flapjax.

site attacks, preventing server attacks, detecting malicious dataflows, and so on) that it is impossible for a language to cover it all. Rather, we believe that by reducing the number of callbacks, Flapjax enables better program analysis, which is a prerequisite to many security analysis techniques. Separately, we have already applied control flow analysis to JavaScript and Flapjax for intrusion detection [18].

Debugging and Contracts The state of debugging support for JavaScript is still quite primitive, and is even more so for Flapjax. Unfortunately, debugging Flapjax can be somewhat challenging because debuggers expose the underlying evaluation mechanism. This exposes the convoluted control flow of event-driven Web applications that Flapjax abstracts away.

Halting on program errors in the innards of Flapjax's implementation is not useful. To avoid this, we have created a higher-order contract system for Flapjax that accurately tracks blame [19]. For example, the contract of `switchE` is:

```
EventStream (EventStream a) -> EventStream a
```

Consider, for example, this illegal use of `switchE`:

```
switchE(timerE(60000))
```

The run-time error, "expected event stream of event streams, received event stream of integers", occurs in the innards of `switchE`, a whole minute after the line executes. Stepping through this code achieves nothing. The contract system, however, identifies this call-site as the source of the error.

The contracts have an ancillary benefit: they precisely document the Flapjax API, which has a straightforward type structure. This is especially valuable in a language without a formal static type system.

How Many DOMs? (or, Beyond Functions to Relations) Because the DOM is a predefined, readily available data structure that often reflects the shape of program data, JavaScript developers routinely conflate the display model—the DOM—with the data model. Not only is this an inappropriate conflation that hinders later maintenance, it can also result in bugs: for instance, browsers can behave in undesirable ways if the same DOM node is inserted as a child of two different parents (which can happen when the actual datum is a DAG or graph, not a tree). Separating the true data model from the display model is known in Web parlance as a *Dual*-DOM approach [2].

Unfortunately, maintaining two models (or three, counting the persistent store) greatly complicates the developer's job. Because changes on one side may trigger updates to the other, the OpenAjax alliance observes [2], "It is usually necessary to establish bidirectional event listeners between the Ajax DOM and the Browser DOM in order to maintain synchronization" (and this ignores the third model). Not only does this mean many more callbacks (with interference caused by updates), developers must take care to not cause cascading cyclic updates.

Recognizing that these situations create *relational*, rather than *functional* (or directed) dependencies, we have experimented with a limited form of principled relational support in Flapjax. Specifically, we have implemented variants of both *lenses* [15] and *constraint maintainers* [24] adapted to the JavaScript object system. Our experiments show that lenses transparently and consistently maintain the model/view relationship in a way that requires less focus on *when* the models change; using Flapjax along with lenses allows developers to use modular reasoning about the occurrence of these changes. Conceptually, developers view the various models—interface, client, and server—as distinct, but conventional practice confounds this distinction with consistency maintenance and race prevention. Flapjax with lenses reifies this conceptual distinction in code.

6. Related Work

A key feature that distinguishes Flapjax from other Web programming libraries is its adoption of functional reactive programming (FRP) [14, 28]. In FRP, instead of using callbacks to respond imperatively to events, a program defines *signals* that vary implicitly as other values in their defining equations change. While this essential idea originates from dataflow programming [6, 33], FRP applies the idea in a dynamic and higher-order setting. The specific approach taken in Flapjax is mainly informed by FrTime [10, 11], a call-by-value instantiation of the FRP model. Unlike FrTime, Flapjax is explicitly designed for use as a library (a design choice that has proven very valuable in hindsight); it models interactions with a Web page's DOM, which requires the ability to define signals that can model richly structured mutable data in a meaningful way; it uses events to interface with Web services; and it handles the inconsistencies and complexities of JavaScript. FrTime and Flapjax differ from the other FRP systems by employing a purely push-based, event-driven update strategy; the Haskell-based systems are pull-based (and driven by polling). Frappé [12], a Java FRP library, uses a hybrid push/pull evaluation strategy. Like Flapjax in library mode, Frappé is closer to a library than a language, not supporting the transparent reuse of host-language programs in a reactive context.

A number of other languages with dataflow-like features have been developed in recent years. For example, Yahoo! Pipes (`pipes.yahoo.com`) is an interactive, graphical domain-specific language for assembling pipelines that filter and aggregate dynamic Web content. Nodes subscribe to and produce feeds, which are updated automatically whenever their source changes. Beyond the Web, StreamIt [32] is a language for constructing networks of stream processors, targeted in particular for high-performance systems with relatively stable graph structures. The individual StreamIt processors operate imperatively on their input and output streams, but are assembled into a declarative graph. Aurora [5] and Borealis [7] offer similar capabilities to StreamIt but, instead of having developers write imperative processing nodes, they support query evaluation for a declarative, high-level SQL-like language over streaming data. They also provide built-in operators for computing various aggregates (e.g., average, maximum) over sliding time windows. All of these languages deal exclusively in discrete data, in contrast with FRP systems like Flapjax, which also provide distinct notions of continuous behaviors and support general purpose programming such as building GUIs, I/O operations, etc.

Systems such as Open Laszlo (`www.openlaszlo.org`), Flex (`www.adobe.com/products/flex`), and JavaFX (`www.sun.com/software/javafx`) have also applied dataflow programming ideas to the Web. They permit user interface elements to be *bound* to expressions: whenever the value of the element changes, the whole expression is reevaluated and the result assigned to the bound variable. This means, however, that behaviors are no longer first-class values. These systems have very limited or no support for higher-order reactivity—the ability to dynamically rebind a variable in response to an event. Furthermore, these systems do not describe any guarantees comparable to our glitch-freedom.

Various languages and systems have been designed around the idea of constraint programming, which is a generalization of dataflow evaluation. For example, ThingLab [3] is an object-oriented constraint-programming language designed for expressing and running simulations. Like Flapjax, it maintains dependencies between objects and automatically propagates updates when values change. The language supports bidirectional constraints and employs a sophisticated constraint-solving engine, which allows it to express programs that Flapjax cannot support directly. However, the Flapjax language is richer in other ways, including support for higher-order functions and reactivity, as well as exposing separate notions of discrete events and continuous behaviors.

Kaleidoscope [16] allows for mixed imperative and constraint programming with multidirectional constraints. It maintains consistency using a constraint solver with support for a hierarchy of constraint strengths, as well as temporal control over constraints, like our events and behaviors. Kaleidoscope conflates intra-model constraints and model/view constraints, while ours are orthogonal libraries built on top of Flapjax. Our model/view constraint system, lenses [15], guarantees well-behavedness of composed constraints—it is unclear what Kaleidoscope's constraint solver guarantees of composed and user-defined constraints.

Other related systems that support constraint programming include the Garnet [26] and Amulet [27] user-interface toolkits. Although these employ a unidirectional constraint-propagation algorithm, they do support cyclic constraint networks, without the need for explicit time delays as Flapjax requires. Instead, they resolve cycles with a simple depth-first "once-around" algorithm, which stops propagating when it returns to a node that has already been updated.

Arrowlets [22] allow developers to specify the control flow of JavaScript programs, across multiple event handlers, using the functional programming concept of arrows. Like Flapjax, Arrowlets abstracts away underlying callbacks. This makes it possible to reason algebraically about control flow across multiple callbacks. However, for an Arrowlet to have a visible effect on the DOM, an arrow must cause a side-effect.

This is apparent in their central drag-and-drop example. Arrows specify a state machine, but the actual effects of dragging are scattered throughout individual arrows, despite the fact that they are abstracted into a "proxy". The proxy methods are essentially callbacks—their return values are discarded. Arrows do allow the DOM mouse events to be composed into a new drag-and-drop event stream. However, the arrow-bassed drag-and-drop abstraction has the same callback-based interface as the DOM. Interesting behavior that uses the results from the various drag-and-drop callbacks will require shared state.

jQuery (`jquery.com`) allows DOM transformations and event handlers to be sequenced and applied to destructively update collections of elements. Flapjax focuses the flow of values through the program, sourced from arbitrary, heterogeneous data sources, including the DOM. Data-flow evaluation is orthogonal to the specification of sequences of effects.

Web application frameworks such as Ruby on Rails (`rubyonrails.org`) and Django (`djangoproject.com`) enable easily creating CRUD (create-read-update-delete) interfaces. They focus on simplifying the object-relational mapping at the database level, and impose good practices such as model-view-controller separation. They are, however, primarily designed for the server, whereas Flapjax connects to any Web service; they also do not offer linguistic support comparable to Flapjax's abstractions.

Formlets [9] address the problem of building Web forms and extracting their input compositionally. Our compositional filters example (section 2.5) is in the same spirit. While Flapjax does not provide the syntactic sugar of formlets, we arguably also do not miss it: in our filter example, we simply use the language's existing binding mechanism to *name* the sub-filter (`subFilter`) and use it in subsequent expressions. We are not constrained by the type-structure of applicative functors and synchronous form submission; as a result we believe that our composed filter example is an instance of their "impossible" formlet, one that renders before producing a result.

Links [8] addresses the problem that Web programs inherently span several *tiers*—the browser, the application server, and the persistent store—each of which must typically be programmed in a different language. The key contribution of Links is to allow the entire application to be written monolithically in one (typed functional) language, and to translate fragments into lower-level code appropriate for their respective execution platforms: JavaScript for the browser, SQL for the database, etc. The compiler can help to ensure, among other things, that the tiers agree on the types and representations of the data they pass to each other, saving the developer the burden of writing such logic by hand. Although Links is nominally "tierless", it provides a relatively server-centric programming model, with the user interface and database acting as second-class citizens. Its user interface support focuses on forms, which lack the rich interactivity of modern applications.

Hop [30] is similar in spirit to Links, but it places more emphasis on the server and user interface tiers, making an explicit distinction between them, and supporting interesting control flow both within and between the layers. Hop allows both layers to be written in a (lexically) single program expressed in essentially the same dialect of Scheme. Its sophisticated compiler can compile any fragment of Scheme code into JavaScript for execution in the browser. The generated code is efficient enough to perform smooth animations or play multimedia in a modern browser. In Hop, both tiers can call into each other, or send events to each other, using asynchronous HTTP request and response messages. However, event-handling uses a conventional callback-based mechanism, which forces the reactive aspects of the program to be written in an imperative style. This is one significant difference between Hop and Flapjax.

Flapjax differs from Links and Hop by taking an exclusively client-centric view. All of the application-specific logic is driven from the user interface, which is written in (an extension of) JavaScript and run in the browser. The Flapjax server is a general-purpose, non-programmable object store with innate notions of users, applications, and access control. At its core, Flapjax is just a library, so no compilation is necessary, although some syntactic sugar for reactive programming is provided through a lightweight compiler. The client library provides an implementation of the server's communication protocol, which simply exchanges objects in JSON notation over HTTP. Its support for modern social Web applications is arguably superior to that of any non-commercial system of which we are aware, given the server's explicit support for users and controlled data-sharing. It also supports *mash-ups* that coordinate data and services from several Web sites, a feature that Links does not seem to provide.

A key source of complexity in modern Web applications is the need to handle asynchronous communication between the user interface and application server. The standard Ajax model provides a callback-based event model; this imposes an imperative style on the developer, as well as demanding explicit continuation management and resulting in the phenomenon of *stack-ripping* [1]. MapJAX [13] addresses this problem as it pertains to the manipulation of shared, persistent data. It abstracts away the asynchronous HTTP communication and callbacks typically needed for Ajax and instead presents a familiar, high-level interface in terms of shared memory and locking. MapJAX frees the developer from the details of the communication protocol between the client and server, and it implements general techniques for making such communication more efficient and effective. A side benefit of using MapJAX is that applications may perform significantly better, in addition to enjoying considerably simpler implementations. Unlike Flapjax, MapJAX only addresses interactions between the client and server, not those between the user and client, which still require callbacks. However, it does provide a notion of concurrency control, which we have not yet explored for Flapjax.

7. Conclusion

We have presented the Flapjax programming language. Flapjax is designed with the needs of Ajax developers in mind. It provides a unified framework for programming with events, both within the program and when communicating with Web services. Flapjax is a dataflow-based reactive language wherein values are automatically updated to be consistent, relieving developers of this burden. Through examples and discussion, we have shown that the mechanisms of Flapjax collaborate to make programs more declarative, and to achieve valuable separations of concerns.

Flapjax is currently only a client-side programming language. This is because of the relative uniformity of Web clients: estimates are that over 90% of browsers have JavaScript enabled, making it the Web's other lingua franca besides HTML. In contrast, there is a much greater range of technologies in use on servers, making it harder to target one platform. Nevertheless, it would be valuable to build support for reactivity and events for server applications as well, and to make these consistent with Flapjax clients to enable the establishment of properties such as glitch-freedom across an entire distributed system.

Acknowledgments

We thank our users, most of all Sean Barker, Morgan McGuire, Noel Welsh, Artyom Shalkhakov, and therac25 [sic]. We are grateful to Pete Hopkins, David Reiss, and Kimberley Burchett for feedback, Cormac Flanagan for constructive provocation, Trevor Jim and Jack Templin for encouragement, and Andrey Skylar, Eli Barzilay, and Jay McCarthy for technical support. Thanks to the authors of Links, Hop, and Arrowlets for useful conversations. This work was partially supported by the NSF. All authors were affiliated with Brown during their primary work on Flapjax.

References

[1] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the 2002 Usenix Annual Technical Conference*, 2002.

[2] OpenAjax Alliance. Successful deployment of Ajax and OpenAjax. `http://www.openajax.org/whitepapers/SuccessfulDeploymentofAjaxandOpenAjax.php`.

[3] Alan Hamilton Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, 1981.

[4] Kimberley Burchett, Gregory H. Cooper, and Shriram Krishnamurthi. Lowering: A static optimization technique for transparent functional reactivity. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 71–80, 2007.

[5] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams — a new class of data management applications. In *International Conference on Very Large Databases*, pages 215–226, 2002.

[6] Paul Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–188, 1987.

[7] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *First Biennial Conference on Innovative Data Systems Research*, 2003.

[8] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects*, 2006.

[9] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. The essence of form abstraction. In *Asian Symposium on Programming Languages and Systems*, 2008.

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

[11] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, pages 294–308, March 2006.

[12] Antony Courtney. Frappé: Functional reactive programming in Java. In *Practical Aspects of Declarative Languages*. Springer-Verlag, March 2001.

[13] Daniel Myers and Jennifer Carlisle and James Cowling and Barbara Liskov. MapJAX: Data Structure Abstractions for Asynchronous Web Applications. In *Proceedings of the 2007 USENIX Annual Technical Conference*, June 2007.

[14] Conal Elliott and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming*, pages 263–277, 1997.

[15] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems*, 29(3), May 2007.

[16] Bjorn N. Freeman-Benson and Alan Borning. Integrating constraints with an object-oriented language. In *European Conference on Object-Oriented Programming*, pages 268–286. Springer-Verlag, 1992.

[17] Jesse James Garrett. Ajax: A new approach to web applications. `www.adaptivepath.com/ideas/essays/archives/000385.php`.

[18] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Using static analysis for Ajax intrusion detection. In *International World Wide Web Conference*, April 2009.

[19] Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *Dynamic Languages Symposium*, October 2007.

[20] Daniel Ignatoff, Gregory H. Cooper, and Shriram Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In *International Symposium on Functional and Logic Programming*, pages 259–276, 2006.

[21] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanovic', and Rastislav Bodik. Parallelizing the web browser. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism*, April 2009.

[22] Yit Phang Khoo, Michael Hicks, Jeffrey S. Foster, and Vibha Sazawal. Directing JavaScript with Arrows. In *ACM SIGPLAN Dynamic Languages Symposium*, 2009.

[23] Daniel R. Licata and Shriram Krishnamurthi. Verifying interactive Web programs. In *IEEE International Symposium on Automated Software Engineering*, pages 164–173, September 2004.

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

[25] Brad A. Myers. Separating application code from toolkits: Eliminating the spaghetti of call-backs. In *ACM Symposium on User Interface Software and Technology*, pages 211–220, November 1991.

[26] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, David S. Kosbie, Edward Pervin, Andrew Mickish, Brad Vander Zanden, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *IEEE Computer*, 23(11):71–85, 1990.

[27] Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, 1997.

[28] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *ACM SIGPLAN Workshop on Haskell*, pages 51–64, 2002.

[29] David A. Patterson. The parallel computing landscape: a Berkeley view. In *International Symposium on Low Power Electronics and Design*, 2007.

[30] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: A language for programming the Web 2.0. In *ACM SIGPLAN Dynamic Languages Symposium*, October 2006.

[31] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: compiling Standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, 1992.

[32] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*, pages 179–196, 2002.

[33] W. W. Wadge and E. A. Ashcroft. *Lucid, the dataflow programming language*. Academic Press U.K., 1985.

A. Flapjax API

In this appendix, we document the portion of the Flapjax API covered by the paper. The Flapjax API defines many more functions, and some of the functions listed here have richer interfaces. We encourage the curious reader to explore the full Flapjax reference at `www.flapjax-lang.org`.

```
$B :: InputElement -> Behavior InputValue
$B(element)
```

Creates a behavior carrying the value of `element`. The element must be a form control (e.g., a text box, a checkbox, etc.)

```
$E :: Element * String -> EventStream DOMEvent
$E(element,eventName)
```

Creates an event stream of DOM events on `element`. `eventName` may be any DOM event (e.g., "`click`", "`load`", etc.)

```
calmE :: Int * EventStream a -> EventStream a
calmE(t,evt)
```

Repeatedly "mutes" `evt` for `t` milliseconds. The result will therefore never fire more than one event in a `t` millisecond interval.

```
changes :: Behavior a -> EventStream a
changes(src)
```

Creates an event stream that fires an event carrying the value of `src` whenever that value changes.

```
constantE :: EventStream a * b -> EventStream b
constantE(src,val)
```

Creates an event stream that fires a constant value, `val`, whenever `src` fires an event.

```
delayE :: Int * EventStream a -> EventStream a
delayE(t,src)
```

Fires all events from `src`, but delays each of them by `t` milliseconds.

```
DIV :: Behavior Element * ... -> Behavior <div>
div(child,...)
```

Creates a behavior carrying a `<div>` element. Similar constructors exist for other HTML tags (e.g., `P`, `TABLE`, etc.).

```
filterE :: (a -> Bool) * EventStream a
        -> EventStream a
filterE(pred,src)
```

Fires only those events of `src` that satisfy `pred`.

```
getForeignWebServiceObjectE :: ...
```

Same interface as getWebServiceObjectE. However, this function can communicate with allowed remote servers via a Flash proxy.[9]

```
getWebServiceObjectE :: EventStream request
                     -> EventStream response
request = { url :: String,
            fields :: Object,
            request :: "get" or "post",
            response :: "json" or "xml" };
response = JSON or XML
getWebServiceObjectE(request)
```

Sends and receives messages from the server. Due to Web browsers' security policies, url must be on the domain serving the Flapjax application.

```
insertDomB :: Behavior Element * Element
           -> void
insertDomB(src,dest)
```

Inserts the element carried by src into the DOM, replacing the static element placeholder dest.

```
insertValueB :: Behavior a * Element * String
             -> void
insertValueB(val,elt,attr)
```

Assigns the value val to the attr attribute of elt. Updates as val changes.

```
insertValueE :: EventStream a * Element * String
             -> void
insertValueE(val,elt,attr)
```

When an event fires on val, sets the attr attribute of elt to the value of the event.

```
liftB :: (a * ... -> r) * Behavior a * ...
      -> Behavior r
liftB(f,src ...)
```

Creates a behavior whose value is the result of f applied to the values of src .... f is applied in topological order (section 3.4) to preserve the algebraic semantics of the program.

```
mapE :: (a -> b) * EventStream a
     -> EventStream b
mapE(f,src)
```

Applies f to all events of src.

```
mergeE :: EventStream a * EventStream a
       -> EventStream a
mergeE(src1,src2)
```

Fires events from both src1 and src2.

[9] For more information on cross-domain security policies, see livedocs.adobe.com/flash/8/main/00001621.html.

```
oneE :: a -> EventStream a
oneE(val)
```

Creates an event stream that fires val just once. The event is fired immediately after the current event has finished propagating.

```
receiverE :: -> EventStream a
receiverE()
```

Creates an event stream that does not fire any events itself. See sendEvent.

```
startsWith :: EventStream a * a -> Behavior a
startsWith(src,init)
```

Returns a behavior that initially holds the value init. When a new event fires on src, the behavior holds the value of the event.

```
snapshotE :: EventStream a * Behavior b
          -> EventStream b
snapshotE(src,sample)
```

Fires an event carrying the current value of sample whenever an event fires on src.

```
sendEvent :: a * EventStream a -> void
sendEvent(val,dest)
```

Imperatively pushes val todest, where dest is created with receiverE.

```
switchE :: EventStream (EventStream a)
        -> EventStream a
switchE(src)
```

Given an event stream of event streams, fires events from the latest inner event stream. When a new event stream arrives, switchE stops firing events from the previous stream and starts firing events from the new stream.

```
timerB :: Int
       -> Behavior Int
timerB(interval)
```

Creates a behavior carrying the current time. The behavior updates every interval milliseconds.

```
timerE :: Int -> EventStream Int
timerE(interval)
```

Creates an event stream that fires an event carrying the current time every interval milliseconds.

```
valueNow :: Behavior a -> a
valueNow(src)
```

Returns the value of src at the point in time when valueNow is applied.