Flapjax: Functional Reactive Web Programming

Leo Meyerovich
Department of Computer Science
Brown University
lmeyerov@cs.brown.edu

People whose names should be before mine.

Thank you to Shriram Krishnamurthi and Gregory Cooper for ensuring this project's success. I'm not sure what would have happened if not for the late nights with Michael Greenberg, Aleks Bromfield, and Arjun Guha. Peter Hopkins, Jay McCarthy, Josh Gan, Andrey Skylar, Jacob Baskin, Kimberly Burchett, Noel Welsh, and Lee Butterman provided invaluable input. While not directly involved with this particular project, Kathi Fisler and Michael Tschantz have pushed me along.

Contents

1. Introduction 3
- 1.1 Document Approach 3

2 Background 4
- 2.1. Web Programming 4
- 2.2. Functional Reactive Programming 5
  - 2.2.1 Events 6
  - 2.2.2 Behaviours 7
  - 2.2.3 Automatic Lifting: Transparent Reactivity 8
  - 2.2.4 Records and Transparent Reactivity 10
  - 2.2.5 Behaviours and Events: Conversions and Event-Based Derivation 10

3 Implementation 11
- 3.1. Topological Evaluation and Glitches 13
  - 3.1.1 Time Steps 15
  - 3.1.2 Paths 15
- 3.2 Dynamic Data Flow 15
- 3.3. Chain Compaction, Constant Independence, and Lowering 17
- 3.4 Behaviours and Events: Distinctions, Similarities, and Alternatives 20
  - 3.4.1 Distinctions 20
  - 3.4.2 Similarities 20
  - 3.4.3 Alternatives 20
  - 3.4.4 FranTk, Arrows: Haskell 20
  - 3.4.5 Frappe: Java beans 21
  - 3.4.6 Flex and Laszlo 21
  - 3.4.7 JavaFX (Previously F3) 21
  - 3.4.8 FrTime: Scheme 21

4 Functional Reactive Web Programming 21
- 4.1. Document Object Model 22
  - 4.1.1 Extraction 22
  - 4.1.2 Tag Creation 23
  - 4.1.3 Tag Insertion 24
- 4.2. Advice, Progressive Enhancement, Templating 25
  - 4.2.1 Cycles: tagRec 26
- 4.3. Time Sensitive Evaluation Techniques 27
  - 4.3.1 Scheduling 27
  - 4.3.2 Framerate 27
- 4.4. Client Server Relationships 28
  - 4.4.1 Transparent Web Services 29
  - 4.4.2 Transparent Persistent Store 29
  - 4.4.3 Consistency 31
- 4.5. Security 31
  - 4.5.1 Transparent Security 32
  - 4.5.2 Reflective Access Control Policies 32
- 4.6. Objects and Collections 32
  - 4.6.1 Delta Propagation 32
  - 4.6.2 Shallow vs Deep Binding 33
  - 4.6.3 DOM Binding 33
  - 4.6.4 Server Binding 33
  - 4.6.5 Disconnected Nodes and Garbage Collection 33
- 4.7. Evaluations 34
  - 4.7.1 Demos 34
  - 4.7.2 Applications 34
- 4.8 Errors 34
- 4.9 Library vs Language 34
  - 4.9.1 Dynamic Typing 34
  - 4.9.2 Optimization 35
  - 4.9.3 Components 35
  - 4.9.4 Compilation 35

5 Related Work 35

6 Future Work 35

7 Conclusion 37

8 Appendix 37

List of Figures

1 HTML Form 4
2 Functional dependencies 4
3 Time graph 6
4 Merge 7
5 Data flow graph 12
6 Glitch: depth first evaluation 13
7 No glitch: topological evaluation 16
8 Switch: usage 17
9 Switch: close up 18
10 Chain compaction 18
11 Graph prior to lowering 19
12 Lowering transformation 19
13 DOM node insertion positions 24
14 Framerate sensitive animations 27
15 Service Model 28

Abstract

The following presents a gentle introduction to a JavaScript library and language extension that support more declarative descriptions of rich web applications through the use of functional reactive[14] constructs. We isolate common tasks performed by rich web applications, including web service manipulation, persistent store access, and GUI interactions, and provide a core library for simplifying their interfaces. Every one of these items is traditionally achieved by the explicit specification of callbacks, inverting control flow. We see that by providing a library for functional reactivity, we can simplify interfaces used to achieve all of these actions, demonstrated in our support for persistent data binding, reflective discretionary access control, and functional specification of time varying visual interfaces, as well as our interface for treating web services as stream operators. Finally, we discuss our approaches to embedding reactivity into a typical language, including two instances of providing interfaces to persistent and externally controlled data structures as well as optimizations useful in the browser environment.

1. Introduction

Web applications are replacing many desktop applications, providing increased portability by only requiring users to have partially standards compliant web browsers, but seemingly at an expressive cost. While HTML and CSS support declarative definitions of static layouts, most complex computations are typically handled on the server-side. Processing on the client is generally imperatively handled with callbacks. Such clientside code can quickly grow unnecessarily complicated, so given the general desire to push interactive computation on to the client, we view this as a timely linguistic challenge. There are many attempts to introduce constructs from general programming languages [7, 28, 24] into web languages, while we are more interested in constructs aimed at particular difficulties of web programming [30, 20, 4, 27, 9]. The use of web programs to achieve many personal and professional computing tasks, such as processing emails, documents, and spreadsheets, and the wide penetration of JavaScript-enabled web browsers makes the browser a pragmatic target for linguistic improvement.

The basic technique we employ, functional reactive programming, has a rich history from the past decade. Originally introduced with Haskell implementations, the technique has been demonstrated to simplify the definition of simple games [13] and resource constrained mobile robots [17]. FRP is a variant of data flow programming typically supporting external data sources, sinks, dynamic changes to the underlying graph structure, and our implementation assumes a topological evaluation strategy so most cycles in the data flow graph are outside of the system as suggested by recent work for a transparent embedding in PLT Scheme [11]. While there are critical semantic and implementation challenges encountered in Flapjax, we show that our current form addresses many of the key challenges in the domain of rich Internet applications. Additionally, while some optimizations for FRP like parallelization [25] do not apply well to current browser environments, other recent work [6, 3] may, and we suggest some additional avenues of approach beyond the ones we do employ based on our experiences.

1.1 Document Approach

Inspired by reception of our work by the academic community as well as questions asked by web developers on our mailing lists, we choose to describe the basic intuitions behind various instantiations of FRP, with an emphasis on our own, as opposed to presenting the current formal semantics of Flapjax. Given recent work in formalizing JavaScript[16], a formal presentation may be more convenient to pursue soon. The following sections first provide a basic background on some challenges of web programming and then introduce the two basic datatypes introduced in functional reactive programming: discrete event streams and continuous behaviours. Those familiar with functional reactive programming may want to skip both, but those that are not may benefit from them. The next section discusses the basic event oriented library implementation used in Flapjax as well as transparently computing over time-varying values [11] instead of always explicitly using certain library methods. With an intuition built, alternate approaches to designing FRP languages are briefly discussed for the benefits of those interested in the general field.

After discussing the basic background on the inspiration for Flapjax as well as our basic implementation of a FRP engine and our general purpose extensions, we discuss aspects of JavaScript, the Document Object Model, and the web programming model we also address in our domain specific approach. First, we describe our interface to the client presentation layer and optimizations important in dealing with subtleties of using the Document Object Model (DOM) for manipulating parts of the HTML tree representing a page. Then, we describe our basic construct for sending and

receiving information through web services in a way that allows subsequent use of our core combinators. Modern AJAX applications typically have privileged web service to facilitate communication between clients and provide access to a persistent store, so we create one instance that exhibits several common, if not always cohabitant, features, and show how to use the static interface to support reactive interaction. Several interesting observations arise. First, this level of abstraction removes the distinction between polling and pushing models from the library user. Second, persistence can be handled transparently. Third, security specification benefits from a reactive and transparent presentation (assuming it is reflective). Finally, moving data binding into the language or a central library allows controlled consistency handling between multiple clients of persistent data, helping avoid problems such as the Orbitz bug common to shopping carts. [21]

2. Background

2.1. Web Programming

Typically, web programs perform a minimal amount of computation on the client, only employing small scripts like form validators, with most computation occurring on the server with an implicit or explicit use of continuation passing style[20]. Better tool support, standards compliance of browsers, resource availability, and user experience requirements are changing this trend, with more computation now occuring on the client side of the client server model of web applications. In practice, it is unclear whether preprocessors [1] that treat client code as syntactic objects, more integrated language approaches [10, 26, 30], or those completely separated by explicitly utilizing web services are preferable. For now, we sidestep the issue, pushing everything we can in to the client [4, 27, 7] for our proof of concept, and thus relegating the server to the status of a specialized web service, leaving room for latter integration. We consider this to be an important question, with a suspicion that context-sensitive environments influencing the evaluation of a single specification [15] may better achieve progressive enhancement[8]. No matter which model is chosen, if the intended applications are rich, many of the interactions we describe will probably exist.

As a concrete example of managing small scale interactions, consider the typical case of a form specified initially with a component layout using HTML and then with callbacks to JavaScript methods in order to compute validity to drive future layout changes. An event, such as a form value change, will propagate to an event handler, which may invoke another handler, such as the aggregate form validity checker. If there are multiple form fields, in one simple implementation, all of their change events will eventually flow into the same validity checking function, and then a variety of events, like visual indications to signify form validity such as alerts or color changes, will flow out to respond to the single value change. Consider the following realization:

Figure 1. HTML Form

```
<form id="myform">
    <input type="text" id="name"
        onchange="validateName()"/>
    <input type="text" id="creditcard"
        onchange="validateCC()"/>
</form>
```

with a validator (Figure 2, page 4).

```
1    <script type="javascript">
2    function gtz (s) { return s.length > 0; }
3    function validateName () { validate(); }
4    function validateCC () { validate(); }
5    function validate () {
6       var name = $('name').value;
7       var cc = $('creditcard').value;
8       var valid_name =  gtz(name);
9       var valid_cc = gtz(cc);
10      var valid_form = valid_name && valid_cc;
11      $('myform').style.borderColor =
            valid_form ? '#0F0' : '#F00';
12      $('name').style.borderColor =
            valid_name ? '#0F0' : '#F00';
13      $('creditcard').style.borderColor =
            valid_cc ? '#0F0' : '#F00';
14   }
15   </script>
```

Figure 2. Functional dependencies (read after write): incrementally compute to avoid redundant computations between repeated calls to *validate*

where the \$ has function type String → DOMNode.

This code has several undesirable properties, all of which become exaggerated when code reaches a more realistic size. An alternative formulation, with simplified common validation functions but expanded specific validation functions, will also be discussed. In the above example, first, unnecessary computation occurs between consecutive calls to validate: mutation of border colors only should occur if validity changes between invocations. The code can be expanded to support incremental computation, splitting apart assignment statements and recording values from previous invocations and only continuing evaluation if newly computed values differ from previous ones. Possible points are at the data dependencies from lines 6 to 8, 7 to 9, 8 and 9 to 10, 10 to 11, 8 to 12, and 9 to 13, so if during subsequent invocations any values during the beginning of a data dependency do not change, latter ones will not either and can be skipped. These optimizations may be significant in the presence of long dependency chains, but manually inserting them would make the program convoluted[3]. This insight is used subsequently, shifting the optimization burden to the compiler: dynamizing static algorithms can be automated[2], ideally without expressive loss. The essence of the above problem is that, while the imperative specification may provide the desired effects, there is a trade-off between the specification and the implementation. Consider converting the 2D convex hull algorithm such that if any of the points move, only the minimal recomputation for the new surrounding surface will be used given the incremental results from the previous computation. Now do so again, except use the 3D convex hull algorithm, etc. [2] The static algorithm is clearer than the dynamic algorithm and thus may be better to use in production environments due to readability, even though the dynamic algorithm may have the desired specification implementation. This decision is concisely made when using callbacks.

Secondly, as callbacks are relied upon for interaction, there may be multiple assignment positions for the same variable, so a cursory analysis of code will not give a concise specification of a border color, especially near where the border color is initially defined. This is a problem inherent to callbacks and a core argument for the use of functional reactive abstractions. As the type of a callback is $\alpha \rightarrow Void$, the chief use of one is mutation. If multiple callbacks are used to define one particular value, such as overall form validity, either there will be multiple functions mutating the same value, making it difficult to discern the specification of the mutated variable, or all of the involved callbacks must invoke one large callback at some point to do the mutation, causing the previous problem and hinting at an abstraction barrier. Additionally, in the above example, the specification seems clear, *assuming* no relevant variables are mutated by other event handlers. We can naturally describe the border color as a function of the validity of the form, and the validity of the form as functions of the *current* values of form fields, so the verbose use of mutation and callbacks seems distracting, especially for the more desirable memoized version.

Finally, considering the first 'a' in AJAX stands for asynchronous, if there is a dependency on the result of a webservice, an additional level of indirection will always be necessary. For example, if the above validate function must also execute a remote predicate on the credit card number, the function would be split into two parts between lines 10 and 11, with the second part registered as a handler for the result of the webservice invocation. If the code manually handled memoization beforehand, the amount of effort required for such a toy example seems excessive. The remembrance of previous values, or treatment of arguments as causal streams, seems almost intuitive at this point: the credit card checking webservice shouldn't be invoked repeatedly for the same number, and if the number is changing rapidly due to it being typed, the web service request should probably be delayed until the stream is less active. These are natural abstractions for stream based languages that are useful for web programming, but do not explicitly exist in JavaScript. Concepts like binding persistent values are too coarsely manipulated in typical JavaScript without them.

JavaScript is a dynamically typed language and fairly concise, yet manipulating GUI and webservice events is still verbose. These two tasks are primary uses of JavaScript and are our target for linguistic and library support. We extend a functional core of JavaScript to operate over time-varying values, including both discrete event streams and continuous behaviours. For example, instead of specifying a callback on a form value change to compute validity, validity would be defined in terms of a form value, with the callback change being set up implicitly. Instead of explicitly mutating a variable repeatedly to achieve an implicit time invariant definition, variables can be implicitly mutated to achieve a clearer, explicit time invariant definition.

2.2. Functional Reactive Programming

Functional reactive programming (FRP) [14] typically facilitates the description and manipulation of two types of time-varying values: always valued behaviours and occasionally valued event streams. Additionally, it

often provides the ability to go back and forth between the two as they are similar concepts.

A variety of approaches are possible, including those with arrows[17] in a point-free style or comonadic[29] styles, and more similarly, those with first class signals and embedded in a call-by-value language [11]. All of these systems have definitions of their semantics with varying levels of rigor, so given the similarity between Flapjax and FrTime and the more basic questions typically asked about Flapjax, this section will focus on a more accessible description of the motivations for and distinctions between events and behaviours.

2.2.1 Events

The core of Flapjax is in its handling of first class discrete streams of values, referred to as event streams or just events. For example, the mouse over events associated with a button form an event stream in which every value in the stream has information about the mouse position at the discrete moment that the mouse moves over a given object.

```
<textarea id="myElt"/>
...
var over2E = $E($('myElt'), "over");
```

or equivalently

```
var myDomElement = TEXTAREA();
var overE = $E(myDomElement, "over");
```

where \$E has type DOMNode * String → Event α

Operations typical for lists and streams are available. For example, we can create a new stream based on another, where every element in the new stream is a function of the corresponding element in the original, thus performing a simple mapping. The simplest such function is constant valued, like the following which creates a new stream with events at the same time as the original stream except every event has the value 1:

```
function toOne (elt) { return 1; }
var onesOnOverE =  overE.map_e(toOne);
```

The above usage is similar to stream manipulation common in shell scripting. In the Bourne Again Shell, we can use *ls* to get a stream of files in the current working directory and *sed* to replace every individual instance with the character '1':

```
ls -1 | sed s/.*/1/g
```

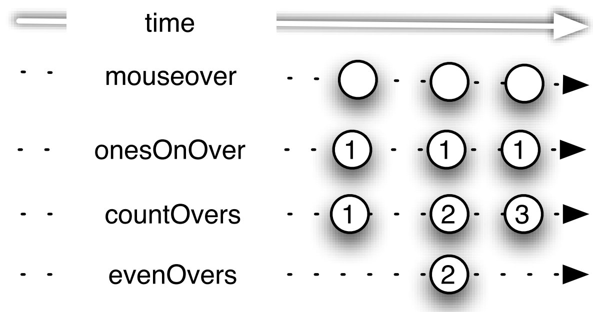

Figure 3. Time graph: note temporal (causal) dependencie going across x axis and functional dependencies going down y axis

Thus, instead of using the '|' operator, Flapjax uses *map_e*. They are not exactly the same, but they are similar enough so that the basic idea of data flow programming should be accessible as many Linux users do it. Instead of the '|' connecting a value stream from one program into the input of another that in turn produces an output stream, with no guarantee that the program will output a value for every input, *map* will provide such a guarantee.

Pipe notation is convenient syntax in shell scripting to chain sequences of transformations together, so in the absence of the ability to create new syntax in JavaScript, we add transformation functions to the prototypes of time varying objects. This allows dot notation. Thus, the following are equivalent:

```
overE.map_e(toOne);
map_e(overE, toOne);
```

When functions take multiple time varying values, we use our best judgement to specify the most intuitive value, or provide multiple aliases with different signals threaded in. Furthermore, this equivalences reinforces the idea that time varying value transformations are not destructive.

State can be maintained between steps in time by inductively defining one timestep in terms of the previous ones by collecting over an event stream, which is similar to folding over a list, just over a stream:

```
var countOversE =
    onesOnOverE.collect_e(
        0, //initial accumulator value
        function (elt, accumulator) {
            return accumulator + 1;
        });
```

The above, which will count the number of events in a stream corresponds to a quick implementation of *wc -l*, except instead of just one event of a final value, the events will be 1, 2, 3, 4, ..., *line count* as if *wc -l* kept reporting the results as they came instead of just on termination. Using techniques presented later, assuming an end of stream message, these initial values can be removed so only one event occurs: the final line count.

Just as a list can be filtered, so can a new event stream be formed such that the only elements it has are those from another that pass a predicate using $filter_e$:

```
var evenOversE =
    countOvers.filter_e(
        function (elt) {
            return elt % 2 == 0;
        });
```

A combination of *map_e*, *collect_e* and *filter_e* can be quite powerful, such as in extracting every other other mouse event:

```
function getEveryOther (eventStream) {
    return eventStream.
        collect_e(
            {c:0}, //initial acc value
            function(v,acc){
                return {c: acc.c + 1, v: v};
            }).
        filter_e(function(o){
                return o.c % 2 == 0;
            }).
        map_e(function (o) {  return o.v; });
}
```

The above example wraps every value into a tuple, labels the number of each event, ignores odd labeled events, and then unwraps the value from every remaining event. We see events are first class, can be named, and can be sources for multiple new events. Events can be combined to create new events based on temporal properties. For example, we can specify an event stream that has a new value when an object is either clicked or moved over. We merge two different event streams by defining a new event that occurs whenever one occurs in either of the other streams:

```
var clickOrOverE =
    merge_e(
        overE,
        $E(myDomElement, "click"));
```

More complicated combinators, like moderating how events are filtered based on their occurance rate, which

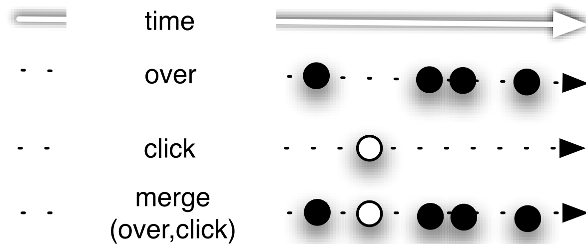

Figure 4. Merge: event values are preserved

is useful for animations, are possible without going beyond the provided combinators into the internals of the library, though they often take advantage of the other time-varying primitive, behaviours, that will be subsequently introduced. While combinators can be defined in terms of a basic set of core combinators, in practice, it is often simplest to just define a native JavaScript function in the style that the original combinators are defined, as presented in the implementation section. This is close to the Scheme mindset in which mutation is common locally, but not globally.

Event streams can be used in a message passing style, which might be useful when defining combinators to compose animations sequenced in a complex manner. Demos for other common uses, such as limiting the rate of elements by dropping some, which is useful in cases such as skipping intraword keypresses when piping a textfield through a spellchecker service, are available online. Overall, the manipulation of events is useful in moderating control and interfacing with the outside world.

2.2.2 Behaviours

A behaviour is like an event stream, except instead of having values at discrete moments in time, it always has a value. A typical behaviour found in a webpage is the current value of a form field, so Flapjax provides a primitive, *$B*, that will extract the value, irrespective of whether it is a dropdown menu, textfield, radio button, or checkbox:

```
<p><input type='checkbox' id='txtfld'/>
Checked: <span id="checked"/></p>
...
var checkedB = $B('txtfld');
insertDomB('checked', checkedB);
```

The above will display a checkbox. *checkedB* will automatically change whenever the checkbox status changes. *insertDomB*, which will be described in detail later, will insert advice to automatically modify the

DOM tree at the node 'checked' whenever checkedB changes. Because a behaviour always has a value, checkedB always has a value, and the DOM tree will be modified immediately to reflect the status of the checkbox, including its changes. We'll focus on interfacing with web specific notions like the DOM in subsequent sections, now describing what we can do with values like *checkedB* in typical FRP literature.

Just as event streams can be mapped, so can a behaviour. For example, we might want to determine whether a form field is filled in using the previously defined *gtz* function:

```
<p><input type='text' id='txtfld'/>
Validity: <span id="status"/></p>
...
var validFieldB = $B('txtfld').map_b(gtz);
insertDomB('status', validFieldB);
```

This small example is more direct than the equivalent using callbacks, but is still not as elegant as can be using Flapjax. The function *map_b* is very similar to function application, and insertDomB has a heavy syntax, which will be both addressed in detail subsequently. As form validity may be the function of the current validity of the form components, behaviours may be the function of several other *behaviour*s. *map_b* is a function over only one *behaviour*, so we introduce a multi-argument version, *lift_b*:

```
var validField2B =
    $B(INPUT({type:'text'}).map_b(isValid);
var formValidB =
    lift_b(
        function (valid1, valid2) {
            return valid1 && valid2;
        },
        validField1B,
        validField2B);
insertDomB(...
```

TODO INSERT DIAGRAM

The above call to *lift_b* is the application of a time-varying (or in this case, constant) function to time-varying (or possibly constant) values, returning a time-varying value (that might be constant valued if all of the arguments are also constant valued). There are subtleties to it that will be described in the next section, but in simple cases like above, it can be used to write many programs that constraint-based systems excel in. Propagation is automatic; if any behaviour changes, all dependent computations will recompute and validity will reflect the current state. An interpretation of *lift_b* is that it takes a function defined on the domain of values, lifts it to operate on the domain of values over time, and applies it to values that change over time, returning the time varying result.

As behaviours always have values, a behaviour such that its current value is that of another from a fixed amount of time beforehand is well-defined, assuming there is an initial value specified. While primarily useful in animations, it can be used to create combinators for web services such as the event stream rate modifier alluded to earlier. For a simple animation without callbacks, the following will draw a box that follows and circles the mouse, with a 100ms delay:

```
var xB = mouseLeft_b().delay_b(100);
var yB = mouseTop_b().delay_b(100);
var timeB = timer_b(100); //increment per .1s
insertDomB('somewhere',
    DIVB({style: {
        position: 'absolute',
        left: lift_b(function(mx,t) { return
            mx + 10*Math.cos(t);},
            xB
            timeB),
        top: lift_b(function(my, t) { return
            my + 10*Math.sin(t);},
            yB,
            timeB}}));
```

In the above example, callbacks are not used so control is clear. This demonstrates the basic usage of behaviours when using Flapjax as a library, except it causes the code to be filled with anonymous functions and invocations of lift that we will eliminate in the compiled language mode. An equivalent script using callbacks and explicit mutation would still have the extraneous functions in the sequence of callbacks, especially if incremental computation is used, so the library mode has similar code length but a clearer control flow.

2.2.3 Automatic Lifting: Transparent Reactivity

In the previous sections, calls to *map_e/b*, or the multiple argument version, *lift_e/b*, are explicitly written in order to take functions that are defined on values and apply them to *Events* and *Behaviours* of those same values, where a value can be any of the JavaScript types (string, number, object) or those created by users. However, as previously alluded to, there is an imbalance in the code. Typically, function invocations are of normal functions, not callbacks. Addition, subtraction, comparison, and general value manipulation are much more common than specialized functions like delay or collect that take advantage of the time varying

traits of *Events* and *Behaviours*. Invocations of lift accompany nearly every function application, assuming no attention is paid towards performance. This transformation is formulaic: every function application, unless the function is one of the special Flapjax ones, like *delay_e*, should be made using lift. We call this lifting, and as the transformation is formulaic, we let the compiler do it, leading to what we call transparent functional reactive programming: reactive values can be used like normal values, with lifting done automatically by the compiler.

Our current automatically inserted lifting function will actually discern whether lift_e, lift_b, or normal function application is most appropriate. Given a call to *someFunction(myArg)*, the compiler will transform it into the following:

```
someFunction(myArg)

=>  liftDispatcher(someFunction, myArg);
```

with liftDispatcher defined, in the simplest manner, as

```
 1 function liftDispatcher (fun, arg1) {
 2     if (fun.alreadyLifted) {
 3            return fun(arg1);
 4     } else (fun instanceof Behaviour ||
 5           arg1 instanceof Behaviour) {
 6            return lift_b(fun, arg1);
 7     } else (fun instanceof Event ||
 8         arg1 instanceof Event) {
 9            return lift_e(fun, arg1);
10     } else {
11           return fun(arg1);
12     }
13  }
```

The first case, on line 2, takes advantage of functions being objects in JavaScript. Whenever there is a function that already expects time varying arguments because it is meant to operate over temporal values, such as delay or collect, we annotate it with an *alreadyLifted* flag, and can directly apply the function instead of using the special capabilities of lift. As seen in the previous examples, we did not call *lift_b(delay_b, someB, 100)* but the more natural *delay_b(someB, 100)*.

The next two cases on lines 4 and 7 are similar. If a function is invoked on a time varying value, or vice versa, the result should also be time varying. In the first, if either the function or argument is a *Behaviour*, we delegate the call to *lift_b*, while if either is an *Event*, to *lift_e*. These were the overbearing calls to lift in the above examples that motivated the overall program transformation.

The final case, on line 10, is a simple optimization. We could call *lift_b*, returning a time-varying value that happens to have the same value at all times, but that is not necessary. If the values are neither *Events* nor *Behaviours*, we can directly apply the function and return the non-timevarying value. Statically typed or macro-based approaches can implement this at compile time.

This simple transformation is at the core of the transparent reactivity found in Flapjax. For FrTime [11], a similar transformation is achieved through Scheme macros. In both languages, functions are generally first class, meaning they can be passed as arguments to functions, which we take advantage of with our calls to lift. This is not universally true in JavaScript: the compiler must special case calls like *alert*, *+*, etc. This is done by wrapping them in functions with a simple eta-expansion:

var x = y + z;

⇒ var x = (function (a, b) { return a + b; })(y, z);

⇒ var x = lift(function(a, b) { return a + b;}, y, z);

We can now revisit an earlier example and treat time-varying values as typical ('flat') ones, trusting the compiler to find any Flapjax scripts and convert them into JavaScript programs:

```
<script type="text/flapjax">
var xB = mouseLeft.delay_b(100);
var yB = mouseTop.delay_b(100);
var timeB = timer_b(100);
insertDomB('somewhere',
    DIVB({style: {
        position: 'absolute',
        left: xB + 10*Math.cos(timeB),
        top: yB + 10*Math.sin(timeB)}}));
...
```

Whenever the compiler sees a script specified as a Flapjax script, it will convert it to the corresponding JavaScript program. Currently, we provide a standalone compiler and a compiler web service, though it is simple enough that several deployments are plausible. First, compilation can be integrated with typical page-serving preprocessing done by servers. Second, the compilation can be done on the client, whether through a webservice or a loaded library. This has the benefit that viewing the source of a page will display the original Flapjax source code instead of JavaScript, which can now be viewed as bytecode. Finally, if a

single language was used, such as one utilizing continuation based constructs, a more robust server side compilation framework might be more practical.

Various optimizations may be implemented in the compiler, inserting calls similar to $liftDispatcher$ and in more careful ways, and will be discussed once the actual implementation of $lift$ and the other library combinators and primitives are clear.

2.2.4 Records and Transparent Reactivity

While a compiler will automate the insertion of calls to lift, in practice, we have found it to be an undesirable in the case of object creation, and thus depart from some previous practices.

To be clear, consider

```
function pair (a, b) {
    return {first: a, second: b};
};
var p1 = {first: 1, second: 2};
var p2 = pair(1, 2);
var p3 =
    { first:  timer_e(1),
      second: timer_e(1) };
var p4 = pair(timer_e(1), timer_e(1));
```

Traditionally, the right hand side of the assignment to *p1* and *p3* would have effectively been first expanded to an application of the *pair* function and then lifted. Thus, the types of each of the variables would have been:

```
p1 :: {first: Int, second: Int}
p2 :: {first: Int, second: Int}
p3 :: Event {first: Int, second: Int}
p4 :: Event {first: Int, second: Int}
```

However, we do not perform this conversion. The above thus yields the following types:

```
p1 :: {first: Int, second: Int}
p2 :: {first: Int, second: Int}
p3 :: {first: Event Int, second: Event Int}
p4 :: Event {first: Int, second: Int}
```

In previous systems, these would be considered the same as there would be a conversion rule akin to the ones for arrays: x :: array event a = x :: event array a. With our change, general usage would not be impacted because function application, including indexing into a record, would still be lifted. However, changes to nested fields are masked, which can be observed. For example, the program to count the number of changes to *p3*,

```
p3.collect_e(0,function(_,cnt){return cnt++;})
```

Our decision is largely related to performance concerns and easing the process of integrating Flapjax code with other libraries. Time-varying values are more explicit, so special cases for integration are clearer. This is a drastic departure from previous work.

2.2.5 Behaviours and Events: Conversions and Event-Based Derivation

Events and *Behaviours* are very similar, and due to the implementation of transparency, can often be used in the same syntactic positions, though with different semantics. Either can be derived from the other, which will be discussed further, so they are presented as common abstractions, not parts of some minimal set of constructs.

Flapjax has an event oriented design, as opposed to a behaviour oriented design, which is an arbitrary choice for our current purposes. Assuming an *Event* datatype, we can create the essential *Behaviour* datatype:

```
function Behaviour (evtStream, init) {
    this.currentValue = init;
    this.changes = evtStream;
}
```

The decision between having a currentValue that will be mutated, as opposed to only storing the first initial value is mostly a matter of convience: external code can access this field, instead of only creating a sequence of callbacks to capture it on change. We can now introduce our first conversion operator that will take an event stream and turn it into a behaviour with some starting value. We call this function hold:

```
// hold :: Event a * a -> Behaviour a
function hold (evtStream, init) {
    return new Behaviour(evtStream, init);
}
```

TODO: rephrase An implicit modeling assumption was made in our notion of a *Behaviour*: it only changes at discrete points in time because we define its changes to be a discrete stream. This means that if we write *changes(sin(timer_b(100)))*, we will not get a smooth behaviour but a discrete event stream. Typically, when we describe a time-varying value as being continuous, we mean it is defined at all points in time, and given the digital nature of our underlying system, sample this value at discrete points in time. As such, a *Behaviour* can be described in terms of its samples and starting

value. Furthermore, in typical FRP systems, the samples may be filtered such that only changes in value trigger dependent computation, but as we will see, this may not always be desirable when we take advantage of mutation. For our definition of *Behaviour*, a variation of collect, filter, and map can be used to inefficiently define a *filterRepeats* combinator that would take a stream of possibly repeating samples and turn it into a stream of unique changes, making the decision a minor consideration.

We define a *Behaviour* as an initial value and a stream, so we may want to convert it back to an event stream:

```
// changes :: Behaviour a -> Event a
function changes (behave) {
    return behave.eventStream;
}
```

The distinction between representing a *Behaviour* as containing a stream of samples or a stream of changes becomes observable at this point. Given a stream of ones, we may *hold* it to make it continuous, and then call *changes* to discretize it again. If we then collected over this stream, adding the values as they arrived, depending on the choice between samples versus changes for *Behaviours*, we may get different results. With changes, we never increase the count, while with samples, they would steadily increase. In Flapjax, we take the samples approach which makes *changes* and *hold* inverses, reasoning that we can always use the *filterRepeats* function to achieve the alternative interpretation. Unnecessary computation can be optimized away later. Using seperate *hold* and *Behaviour* functions provides a stylistic choice to show intent in case the language is later extended.

Further manipulations of *Behaviours* follow a similar pattern as the above: define the current value using typical logic based on the values of the arguments at the time of invocation, and define the underlying sample stream using Event combinators. For example, we can define *map_b* in terms of *map_e*:

```
function map_b (fun, arg) {
   return new Behaviour(
       map_e(fun, arg.changes),
       fun(cur_a));
}
```

As more manipulations of *Events* and *Behaviours* are introduced, the convenience of having both types will become clearer.

For another example, we can define *delay_b* in terms of *delay_e*:

```
function delay_b(behave) {
    return new Behaviour(
        delay_e(behave.changes),
        behave.currentValue);
}
```

The conversion is rather formulaic, but occasionally the handling of the initial value may be different from subsequent changes, as seen with *delay_b*. In it, instead of starting out undefined, we decide to have the values diverge only after a change. Alternatively, we could have started the value as being undefined, and initialized it after the delay time. Initialization is often a special case in programs, which this view of *Behaviours* makes explicit prior to the typically reactionary definition of event combinators.

3. Implementation

Flapjax maintains a directed acyclic data flow graph where every node corresponds to an event stream. This is what we mean by an event-oriented implementation. Thus, if an event occurs, it is represented by a node firing, which will propagate the event value to whatever nodes are directly dependent upon it for further processing. Forward references are used to propagate events through the system, calling a receiver function on a listening node to generate new events to propagate further. Back references are maintained for potential optimizations such as crawling a branch of the graph and temporarily disconnecting the receiver functions on its entire fringe nodes if none of its sink nodes are externally accessible and no processing function associated with any of its node contain impurities that may effect global state. As previously described, *Behaviours* are represented as a current value and an underlying event stream of changes, and thus can also be represented by the same data flow graph. Calls to *Event* and *Behaviour* combinators, such as lift, merge, and delay, correspond to additions to the data flow graph connected to existing nodes.

```
function Event (sources, updaterFn) {
    this.sinks = []; //forward
    this.souces = sources; //back
    this.updaterFn = updaterFn;
}
```

Nodes, upon receiving a value, may process it to create a new one, and then send that value to their dependents. However, this is not always done. Furthermore, as we will see later, there may be global properties we choose to enforce in how values propagate between

nodes, and these are most easily maintained by providing the same general propagate function to each node. Thus, we define the *updaterFn* from the above *Event* definition to follow a rather permissive signature. Every event specifies a function that can be invoked by any of its sources. This function will take a value to process, and a callback it can call to give any eventual values that should be propagated. A little more precisely:

updaterFn :: ($\beta \rightarrow$ Void) * $\alpha \rightarrow$ Void

We define a simple function to propagate values between nodes, which are sent through updater functions along with a way to continue propagation:

```
function propagate (val, node) {
    for (var sink in node.sinks) {
        sink.updaterFn(
            function (sinkResult) {
                propagate(
                    sinkResult,
                    sink);
            },
            val);
    }
}
```

We can now define three useful functions: *map_e*, *merge_e*, and *delay_e* that take advantage of this structure.

```
//map_e :: Event a * (a -> b) -> Event b
function map_e (evtStream, fn) {
    return new Event(
        [evtStream],
        function (prop, val){
            prop(fn(val));
        });
}

//merge_e :: Event a * Event b -> Event a u b
function merge_e (evtStreamA, evtStreamB) {
    return new Event(
        [evtStreamA, evtStreamB],
        function (prop, val) {
            prop(val);
        });
}

//delay_e :: Event a * Integer -> Event a
function delay_e (evtStream, wait) {
    return new Event(
        [evtStream],
        function (prop, val){
            setTimeout(prop(val), wait);
        });
}

//sample usage
var a = map_e(function(v){return v + 1;}, x);
var b = merge_e(a, x);
var c = delay_e(a, 300);
```

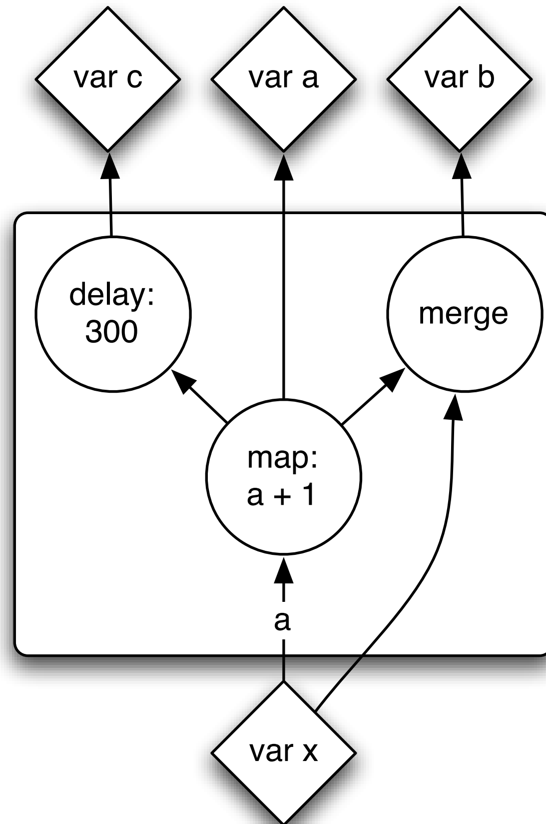

Figure 5. Data flow graph

These three constructs have already been demonstrated to be useful and were very simply implemented. While, as mentioned above, a combinator like *filterRepeats* that filters out any event value that is the same as the one before can be implemented with *collect_e*, *filter_e*, and *map_e*, such an approach is not necessary. The goal is better readability; just as mutation is often used locally in Scheme and OCaml, it is not necessary to write small combinators in terms of other ones. However, doing so may aid program reasoning and the ability of a compiler or dynamic profiling to optimize the dependency graph structure.

3.1. Topological Evaluation and Glitches

The above definition of propagate will yield difficulties with methods that are dependent upon multiple time varying values. For example, consider a naive implementation of *lift_b* which will recompute a function application whenever either of its arguments change. We can implement a multiargument *lift_b* using a single argument lift (*map_e*) and *merge_e*:

```
//naiveLift_b ::
// (a * b -> c) * Behaviour a * Behaviour b
//    -> Behaviour c
function naive2Lift_b (fn, a, b) {
    return new Behaviour(
        fn(a.currentValue, b.currentValue),
        map_e(
            merge_e(a.changes, b.changes),
            function(_){
                return fn(
                    a.currentValue,
                    b.currentValue);
            }));
}
```

but an undesirable result arises on the following program under the reactivity transformation. Consider the following example, assuming y is some integer valued *Behaviour*:

```
var a = y + 0;
var b = y + a;
var c = b + 1;
var d = c % 2;
var e = 5 / d;
```

Namely, if this block was always executed atomically, b would always be even, c, odd, d, 1, and e would thus should always yield 5. However, consider the expansion using *lift_b* and the evaluation order under *propagate* as defined:

```
a = lift_b(function(l,r){return y+0;}, y, 1);
b = lift_b(function(l,r){return l+r;}, y, a);
c = lift_b(function(l,r){return l+r;}, b, 1);
d = lift_b(function(l,r){return l%r;}, c, 2);
e = lift_b(function(l,r){return l/r;}, 5, d);
```

Under the transformation, when y changes, an exception may be raised if we used the proposed implementation of *propagate* and a multiargument *lift_e/b*. This is illustrated in (Figure 6, page 13), showing how the evaluation would yield a division by zero exception by the top node if y changes from 3 to 2.

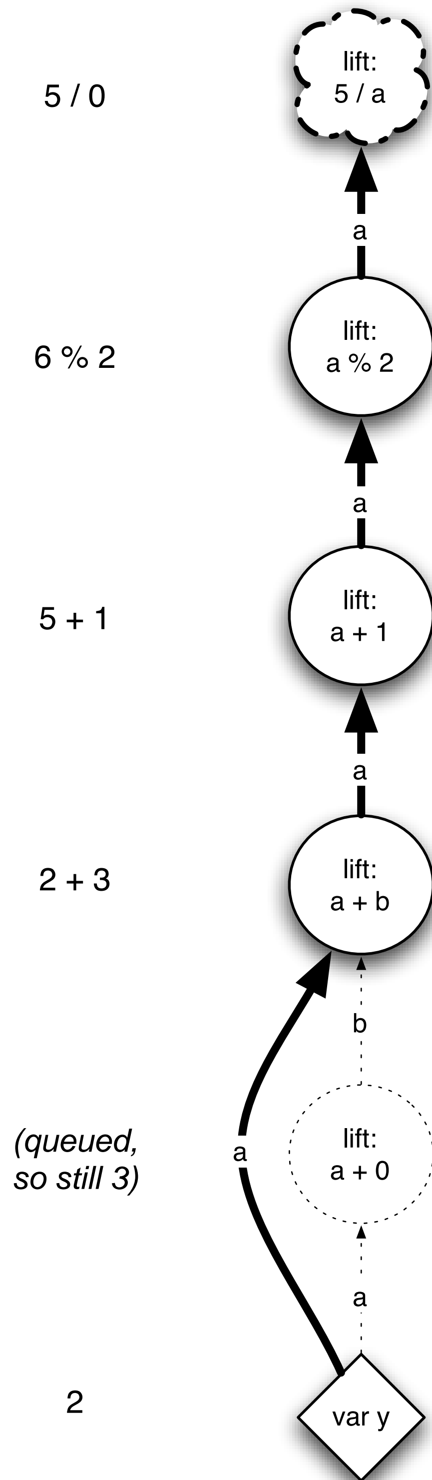

Figure 6. Glitch: depth first evaluation order incorrectly raises an exception when y changes from 3 to 2. The solid line connections are executed first, reaching an exception before the dotted path is evaluated.

Assuming no error was thrown, the change to the right branch of a would then be propagated through the system, and the originally intended result would succeed. A trivial change to this example would be to have a change of either argument sources of a lift node cause the node to look at the current value fields of both branches and recompute based on them. However, this would not work if we replaced one of the ys in the assignment to a with *(function(v)return v;)(y)*. In this case, an additional node would be between a and y. More generally, determining the next value of a source before it has changed is undecidable, so to prevent this consistency problem, a multiargument *lift* should wait until both of its children have updated. However, as is, this is also an undecidable property - one of the sources may not even update, and probably, won't. An option is to optimistically evaluate and propagate on notification of a change in one branch, ignore exceptional values, and upon receiving an update from the other branch in the same timestep, just rollback. However, this may lead to excessive computation, and as Flapjax is embedded in an impure environment and should support external interactions with legacy applications, rolling back may not be an option.

Instead, we enforce a variant of a breadth-first evaluation order. As we require that before a node evaluates that any of its children needing evaluation have already done so, we evaluate the children first. After a node has evaluated, if it needs to propagate values further, it will notify the propagation function as usual. We modify the propagation function to, instead of directly propagating the value to the nodes waiting for it, to defer it. The lowest level node scheduled to receive a value is processed instead, and the desired invariant is thus established. To achieve this, we also modify node creation such that a node is defined to have a rank that is the maximal rank of its children plus one, and add a priority queue to propagate:

```
function Event (sources, updaterFn) {
    this.sinks = []; //forward
    this.souces = sources; //back
    this.updaterFn = updaterFn;
    this.rank = 1 + Math.max.apply({}
            sources.append[0]);
}

var pq = new PriorityQueue();
function propagate (val, node) {
    for (var sink in node.sinks) {
        pq.push(sink.rank, {s: sink, v: val});
    }
    while(!pq.isEmpty) {
        var o = pq.pop();
        o.s.updaterFn(
            function (sinkResult){
                propagate(
                    sinkResult,
                    o.s);
            },
            o.v);
    }
}
```

An implicit assumption of such an ordering is that the dependency is a directed acyclic graph. We sometimes break this rule, adding edges for propagation that are not considered in rank calculation. This is because we do allow callbacks in our system: it is sometimes inconvenient to use our cyclic dependency constructs (to be discussed subsequently), so we allow callbacks to inject into event streams.

A slight problem still exists: while we guarantee order, lift will still be called twice. However, this can be easily fixed. We start with the more intuitive version of lift:

```
function lift2_b (fn, a, b) {
    var cur = fn(
            a.currentValue,
            b.currentValue);
    var changes =
        merge_e(a.changes, b.changes).
        map_e(function(_){
                return fn(
                    a.currentValue,
                    b.currentValue);
            });
    return new Behaviour(cur, changes);
}
```

We know both source behaviours, a and b, will have already updated in this time step if they will at all. Therefore, the first notification of a change will yield the correct output. However, a second notification of a change, from the other branch, will cause evaluation again. For some nodes, such as a merge node, this may be a good thing, while in this case, it should be filtered out. To promote local reasoning about such multiple invocations within the same timestep, we change the signature of the node update function. Instead of reasoning about only a value, it is wrapped inside an object that contains some information about the context it arose in. In our implementation, we currently use two annotations: when the event occurred, and which nodes have strong causal links to it. Instead of receiving a value and a callback that expects a new value, we define the updater function to receive a *Pulse* and

a callback that expects the next *Pulse*, where one field of the *Pulse* is the original value of interest.

$snapshot_e$ :: Behaviour a * Event b → Event a, which will take snapshot of a *Behaviour* during specific discrete points in time, provides a safe, glitch-preventing approach to sampling a value, as a opposed to using *lift_e* and potentially prematurely sampling a *Behaviour*.

3.1.1 Time Steps

We informally define a timestep to occur between when an event begins to propagate through the dependency graph to when the last event caused by it, under transitive closure, has occurred, discounting delayed events. When a new event occurs from within the system, such as one caused by *map_b*, it is considered to occur within the same timestep. When an event occurs from outside of the system with no known causal link, such as from $\$E$, or due to a known timestep, as with *timer_b*, it cannot be processed until all of the events associated with the current time steps are handled, or we may introduce glitches. An additional queue is therefore used for these latter types of events. When the originally presented priority queue is empty, also known as the dependency graph having stabilized, the external event queue is checked for events needing processing and the timestep increments.

When a node receives a change event, it also receives the timestep the event occurred in as part of the pulse. Thus, *lift* can filter out subsequent invocations on the same timestep:

```
function lift2_b (fn, a, b) {
    var res = {val: null};
    var lastStep;
    var cur = fn(
            a.currentValue,
            b.currentValue);
    var changes = new Event(
        [a.changes, b.changes],
        function (prop, pulse) {
            if (lastStep != pulse.timestep) {
                lastStep = pulse.timestep;
                prop(new Pulse(
                        lastStep,
                        link(
                            res.val,
                            pulse.path)));
            }
        });
    res.val = new Behaviour(cur, changes);
    return res.val;
}
```

We have now achieved our desired evaluation order (Figure 7, page 16).

3.1.2 Paths

Every event has a causal path. Some of our earlier implementations took advantage of this data, as did some derivative work. Occasionally, we may create constructs that allow cycles in our dependency graph, and instead of having an endless feedback loop, may use path information to break them. An updater function can store the time stamps of previously received pulses for one to detect a repeated invocation in a time step, which would be detected by the same stamp appearing repeatedly, but path information may be useful for a finer notion of causality.
For example, cyclic evaluation could be detected and stopped given full path information.

In summary: to prevent inconsistencies, any change propagation leading to the application of a multiargument function will delay evaluation by using a priority queue indexed by topological level, which is the maximal rank of dependent nodes plus one. Nodes in the queue only evaluate and propagate when popped off the queue. During evaluation, events and value changes may enter the system, indexed by time, so a queue of external events is also kept. When the internal queue has been exhausted and the data flow graph has stabilized, an event from the external queue is popped off and propagates through the data flow graph.

We find this to be a critical property in practice. As an experiment, we originally did not include it, and our new users were confused about the cause of certain bugs, even when aware that this property was not maintained. The problem applies to many multiargument function, not just *lift*. For example, this enforces the intuitive observational behaviour of *if_b*.

3.2 Dynamic Data Flow

Functional reactive programming, in addition to lifting, supports *dynamic* data flow, or data flow graphs whose structure changes over time. How the structure can change is specified beforehand, but the actual connections between nodes, and even existence of nodes, changes over time. Web services may return data not available during initialization, so this is a natural occurrence in our domain. The key operator to facilitate removing and adding connections is *switch*.

Consider a webservice that returns a stream of links and a program that visualize the links as they come in and also shows the history of links that the mouse has run over: s

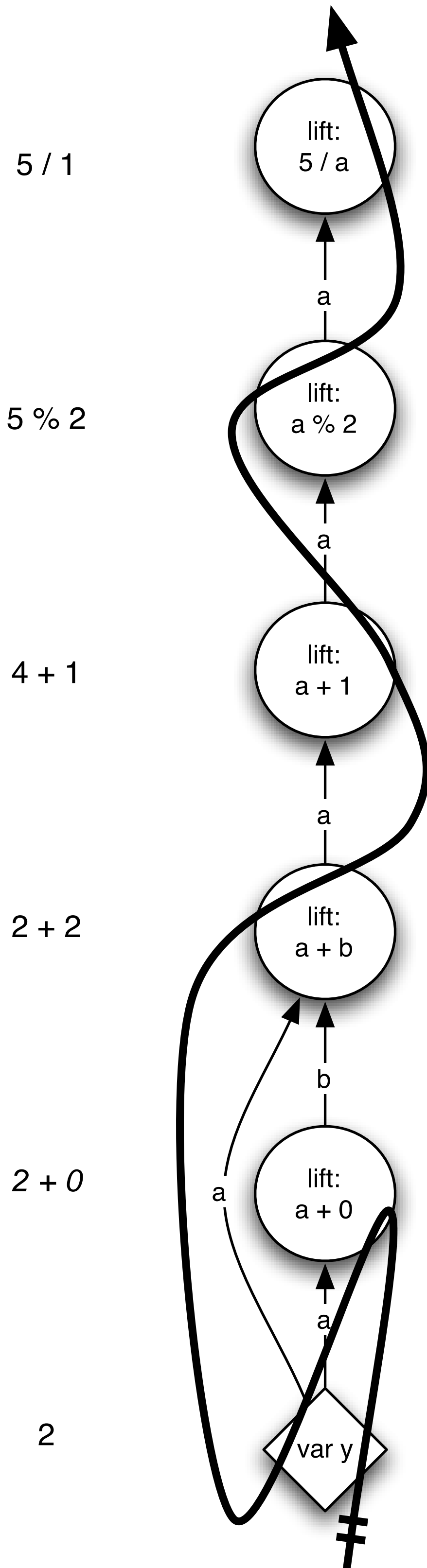

Figure 7. Glitch: topological evaluation order prevents the introduction of inconsistent states

```
<script type="text/flapjax">
var result =  // result :: Event DOMNode
    service().map_e(
        function(txt){
            return A({href: txt}, txt); });
var overs = // overs :: Event Array Event
    result.collect_e([ ],
        function (link, arr) {
            arr.push($E(link, 'over'));
            return arr; });
var anyOver = //anyOver :: Event Event
    overs.map_e(function(arr) {
            return merge_e.apply({},arr);
        });
```

The script takes a stream of strings and turns each into a link. Then, it takes the stream of links and collects their corresponding mouseover streams into an array. Finally, it compacts each array of streams into one merged stream, so on every addition, a new updated stream is provided. If we wrote *pulse*, a stream of true/false values, in the naive manner,

```
var badPulse = anyOver.collect_e(
    false, function (_, v) { return !v})
```

we would create a stream of booleans that flip every time the service returns a new link, not when the mouse moves over a link. In the above usage of *collect_e*, the accumulation function is not impacted by the value of the event passed in. However, we really want to define pulsing in terms of this value, for it is the merged stream of mouseover events from all current link objects. We need a function that will take a stream of event streams, and whenever a new stream occurs, start returning events from that stream instead of the previous stream. As an analogy, at a railroad junction, we may switch which tracks will be used to send trains further along, shutting off the other pair. In other FRP systems that are concerned about providing space guarantees in terms of how many streams are active, this is a close analogy: only two streams are possible to choose from. In our case, we allow an infinite set of streams that appear over time in a stream and assume unused streams can be garbage collected, which will be discussed later. Our stream of pulses can now be written using *switch_e*, which has type *switch_e* :: Event Event $\alpha \rightarrow$ Event α:

```
var goodPulse = anyOver.switch_e().collect_e(
        false, function (_, v) { return !v})
```

All that was required was a call to *switch_e* to decrease the order of reactivity of events: segments of data flow graph were traveling through the data flow

graph, and at a switching point, we connected them (and removed the previously connected node). We can see the data flow graph corresponding to the above example (Figure 8, page 17) and,

more generally, to the usage of *switch_e* (Figure 9, page 18).

Within Flapjax, the most common way to introduce events is the *$E* function, and as it is already defined to work over time varying nodes, *switch_e* is often not necessary as it is implicitly already used within $E. However, when collections are also used, it becomes useful.

3.3. Chain Compaction, Constant Independence, and Lowering

Key static optimizations may be implemented by the compiler [6]. Hinted by the presence of an underlying data flow graph to guide computation, every node in the graph incurs a cost during event propagation due to the actual message passing as well as guaranteeing a topological evaluation order.

A simple optimization is in the handling of chains of unary functions. For example, consider the fragment:

```
var z = foo(bar(baz(x)));
```

A simple lifting compiler would produce the following:

```
var z = lift(foo, lift(bar, lift(baz, x)));
```

However, we can condense the chains of *lift*:

```
var z = lift(
        function (a) {
            return foo(bar(baz(a)));
        },
        x);
```

The optimization will fail, however, if *foo*, *bar*, or *baz* is already lifted to operate over time varying values, such as if it is the delay function already set for a specific amount of time. However, in such places, functions can be annotated for the compiler or in languages with finer type systems, typed. Alternatively, we can use the original lifting transformation, but, during graph construction, we can check for chains based on dynamic function annotations and connect nodes in a way that skips the priority queue. Instead of delaying node processing by deferring to a priority queue, a message in a chain can be instantly handled, circumventing the performance costs related to using a priority queue.

In a typical program, there are many multi-argument functions. In our original *validate* function,

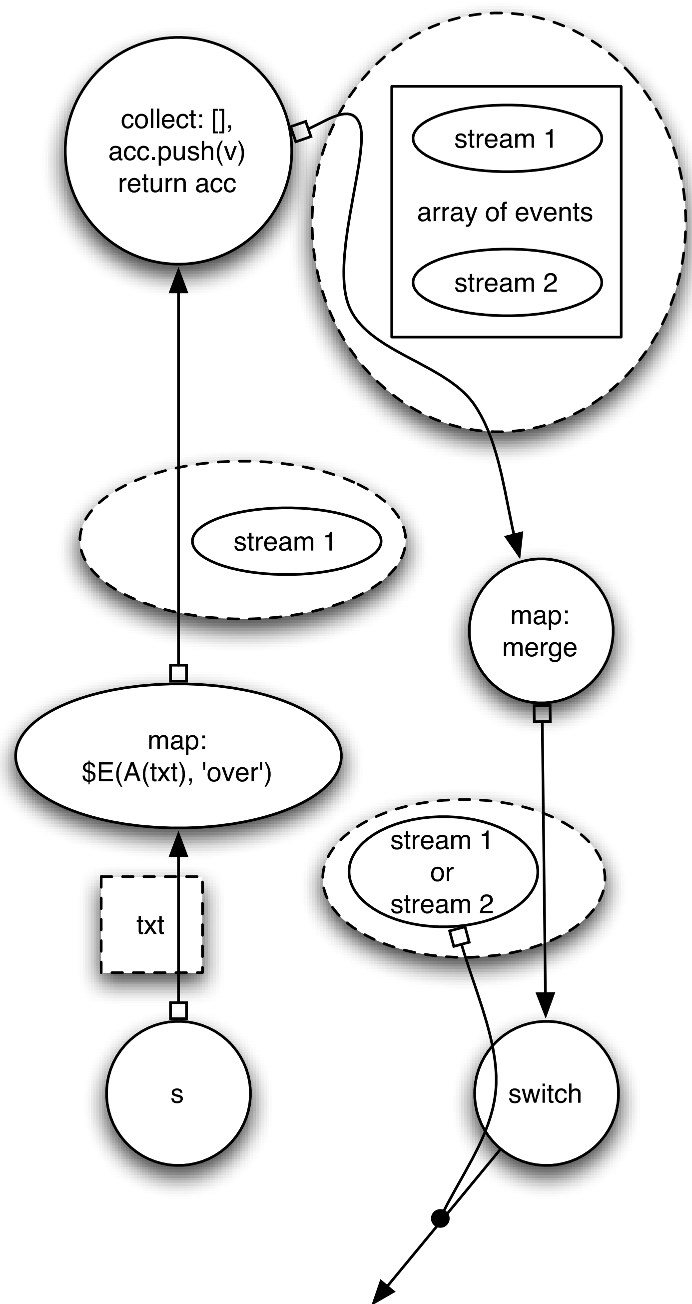

Figure 8. Switch: data flow graph representing reactions to mouse clicks of objects that are dynamically generated. Dashed circles are individual events and solid circles are data flow nodes through which events propagate, so solid shapes within dashed circles signify values of events.

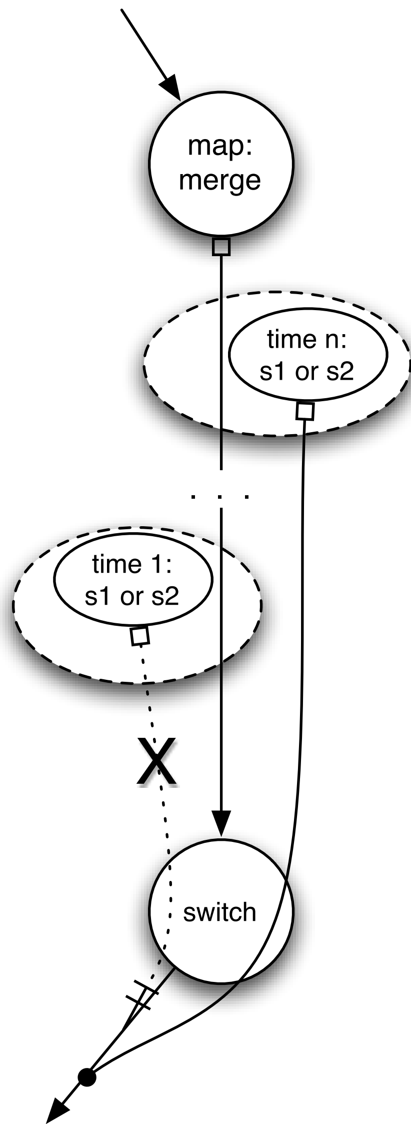

Figure 9. Switch: a switch node, on notification, will connect a newly received event stream and disconnect the previously connected stream if any. Dashed circles are individual events and solid circles are data flow nodes through which events propagate, so solid shapes within dashed circles signify values of events.

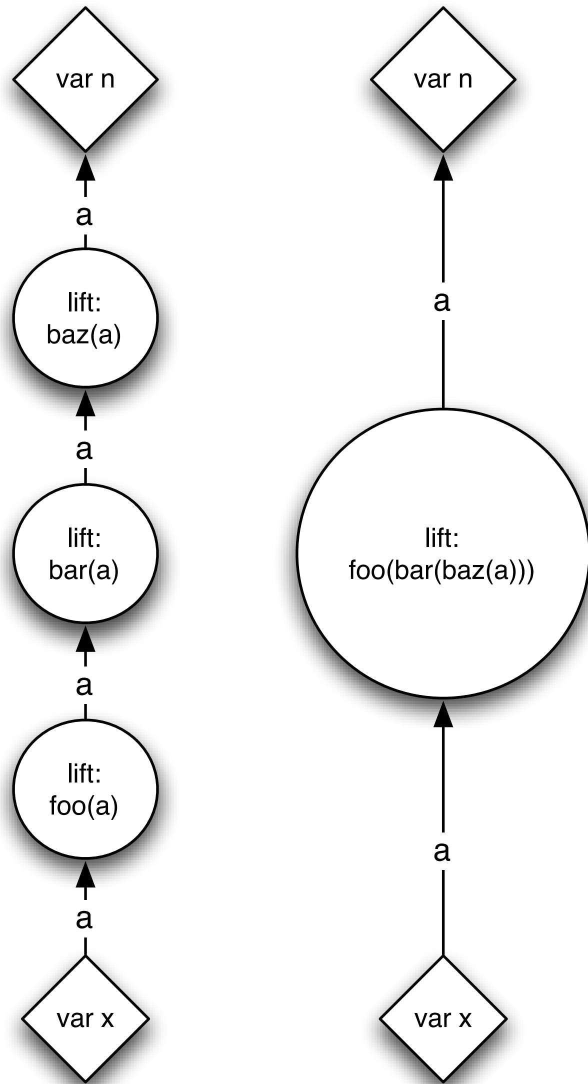

Figure 10. Chain transformation: original (left), compacted (right)

there are two invocations of unary functions but four invocations of binary ones (*or(a,b)* and *?(testcondition, truebranch, falsebranch)*. However, at least for the shown invocations of the ternary conditional operator '?', we can actually describe its use with a unary function. For example, consider

```
form_border = valid_form ? '#0F0' : '#F00';
```

with a naive translation to

```
var form_border = lift(
    function(a,b,c){ return a ? b : c; },
```

```
    valid_form,
    '#0F0',
    '#F00');
```

Two of the parameters are constant over time and thus are not variables that need to be propagated through the dependency graph. Instead, they can be placed within a node:

```
var form_border = lift(
    function(a){return a? '#0F0' : '#F00'},
    valid_form);
```

Thus, by moving constants within lifted functions, we can often take multi-argument functions and represent them as unary ones, and then can gain further benefit from the chain compaction optimization previously mentioned. This step can be done statically without relying on any annotations. Again, in more finely typed languages, we may be able to do more. For example, the optimization would be able to support named constants.

These optimizations avoid unnecessary use of the priority queue. A generalization, with the same caveat around already lifted functions (and thus implicitly stateful nodes), is called lowering [6], that we do not implement. A section of the dependency graph can be condensed into as many nodes as there are outgoing edges, assuming no temporal functions that cannot be compacted. For example, we can convert the following,

```
var n = (a + b) + c * d;
```

to

```
var n = lift_b(
         function (w,x,y,z) {
             return (w + x) + y * z;
         },
         a, b, c, d);
```

but at the expense of not reusing previously computed values through incremental computation. Chain compaction, however, will incrementally compute. The basic insight is that for small functions, the cost of message passing may be higher than that of recomputing the value. There may be a continuum between lowering and the optimization we describe: we may be able to take advantage of node cost and event rate profiling or known complexity to dynamically or statically structure the graph and paths through it. More local optimizations, such as analyzing event rates when deciding whether branches of conditionals should be connected and filtered or disconnected based on the

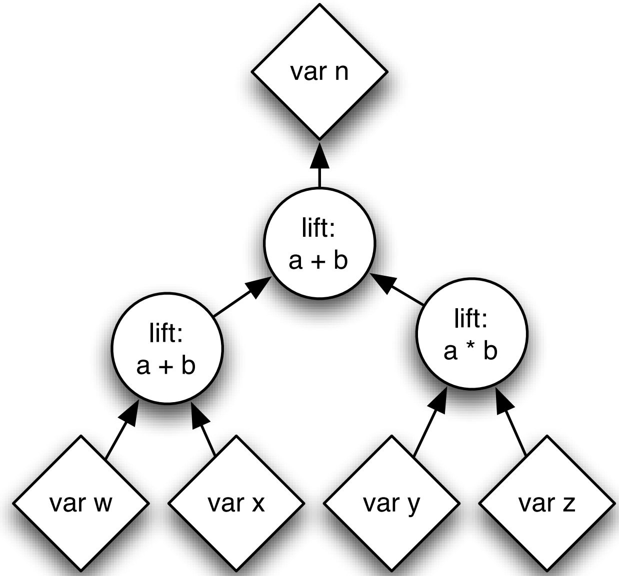

Figure 11. Graph prior to lowering

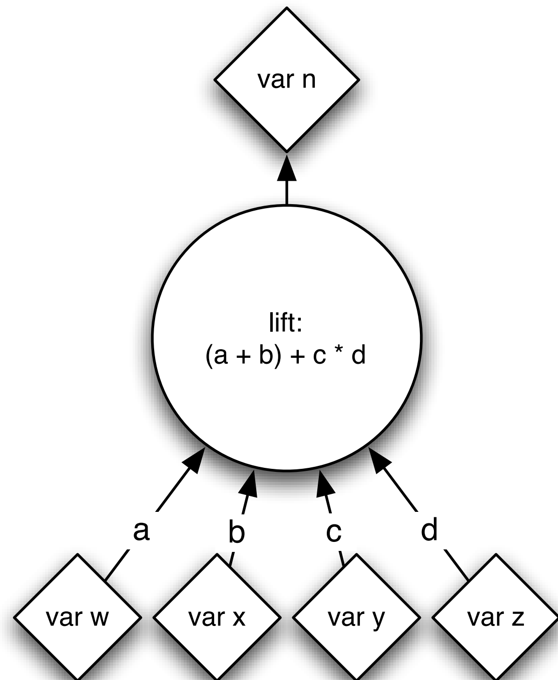

Figure 12. Lowering transformation

test branch face similar issues. These sort of optimizations are difficult to do by hand, especially when programs are updated in the standard development process where usage changes through iterations, because incrementalized and memoized code is difficult to read, so automation may be very beneficial.

3.4 Behaviours and Events: Distinctions, Similarities, and Alternatives

3.4.1 Distinctions

Initialization of a program, in practice, is a significant special case. If a program is written entirely as event streams, upon initialization, there will be no values flowing out of the system. To have any visible effect, an external event must occur or an internal generator like *timer_e* must be used. This creates an artificial spatial gap in the syntactic specificaiton. Furthermore, while describing programs as function of events rather than callbacks reverts potentially inverted control flow specifications, relying on descriptions of event streams may make code hard to understand in practice. For example, given the program

```
...
var a = x + y;
var r = a + z;
```

and assuming only events are used and the semantic that *lift_e* will only have an event when all of its arguments have an event in the same time stamp, it is unclear as to when r will occur: at the beginning of the program, will an event occur that causes x and y to occur, and thus also cause r to occur?

Conversely, consider the case that only behaviours are used. If r is fed in to some logging function or is bound to a value on the server, a write would be triggered on initialization. In the former case of only events, an event would have to be manually triggered and thus initialization behaviour would be explicit, while in the latter case of only behaviours, network resources would be wasted during an undesirable initial write. If there are many webservices on a page, there can be a significant delay. However, due to initialization costs, we are finding some users to take the event based approach whenever values can be initialized statically with HTML. This approach may work for small applications, but suffers when application size grows.

3.4.2 Similarities

Events and behaviours are similar: we have already shown how to implement behaviours in terms of events. Assuming an implementation of behaviours, we can also easily implement events.

They just become behaviours of type *Maybe*. For example, type *Event Integer* is equivalent to type *Behaviour Maybe Integer*. At any point in time, the behaviour has a value, or it does not. Again, our decision of assuming behaviours are discretely changing or potentially continuously with a sampling rate drives an implementation of *changes*. We show how to define one sample combinator, merge, assuming it is correctly lifted with respect to the implementation of behaviours:

```
function merge_b(a, b) {
    return a instanceof MaybeFull ? a
        : b instanceof MaybeFull ? b
        : new MaybeEmpty();
}
```

Note how such a decision implies one event stream cannot have multiple events in the same time cycle. Cycles in our implementation of the dependency graph can lead to such an occurrence, which we call a tight-cycle, as opposed to a cycle that occurrence between causal dependencies between multiple time steps.

3.4.3 Alternatives

The basic notion of data flow is not new, and to a significantly lesser extent, nor is functional reactive programming. Conceived within the Haskell community [14], it was also demonstrated to work over Java beans [12]. Both of these realizations were different in implementation, but similar in spirit. Flapjax is most similar to FrTime in implementation.

3.4.4 FranTk, Arrows: Haskell

Seminal work in reactive programming can be traced to FranTk [14]. Arrows, a generalization of arrows, are used. These are more similar to forementioned Unix pipes than our style of embedding. *Events* and *Behaviours* are both time varying, and thus are simply described as *Signals*. Values are actually a function of time, so a constant value, such as 1 (*Signal Int*) actually has type $Time \rightarrow Int$. The *add1* :: $Int \rightarrow Int$ function can be lifted by *arr* to work on the Int domain extended over time, *arr add1* :: *Signal Int Int*, and then applied to the base signal to achieve the new one. Such an approach is possible in JavaScript, but the nature of the strict, stateful environment makes a FrTime [11] style embedding more natural.

3.4.5 Frappe: Java beans

In Java beans, object fields are accessed by getter/setters. Thus, all communication between beans is explicitly made through method calls instead of more implicitly through mutation of public fields. Frappe [12] takes advantage of this basic insight, allowing us chain getters rather than setters, reversing the direction and ambiguities of control. Such an approach would be possible in coming versions of JavaScript, but is rather heavy handed.

3.4.6 Flex and Laszlo

Some commercial rich web application frameworks[4, 27] feature data binding. Typically, this means an explicit data model object is made, at which point it can be inserted into a page in a manner that looks similar to our curly bang syntax. However, there is little transparency, first class behaviour, etc. Thus, while {*!name+1*}*!* may work, {*!name+time*}*!* will probably not, and if it did, without our glitch handling semantics. Additionally, we support transparency in scripts, not just templated code, so functions can be written that rely upon transparency. In these systems, to declare a time varying value in code, a special data object must be made that is explicitly set, typically with callbacks. While templated code is simplified, the bulk of a program still suffers from inverted, convoluted control flow.

3.4.7 JavaFX (Previously F3)

A recent framework implements static data flow with related transparency support. A variable can be bound to be the result of a function of another, with an intuitive compiler transformation to insert advice into relevant setter functions. However, such an approach loses two important properties of our style of embedding: time-varying variables are no longer first class and data flow cannot be dynamic. The approach of requiring the use of an explicit binding construct to indicate a time-invariant relationship between variables provides an interesting approach of mixing time-varying and flat code, especially in the face of possible mutation. asdf

3.4.8 FrTime: Scheme

Our approach is in the style of FrTime, mostly different in minutea and particular optimizations. *Signals* are first-class values as in FrTime, with an underlying data flow graph, and unlike FrTime, to support integration with other libraries and traditional code, we do not maintain the isomorphism between *Collection Signal* and *Signal Collection* types, as motivated elsewhere in this document. Additionally, we find other optimizations necessary, and find new guidelines for integrating with persistent data structures.

4. Functional Reactive Web Programming

So far, the majority of topics discussed are either reviews of existing work in the general field of functional reactive programming or our basic adaptations of general concepts, but not specific to the environment provided by modern web browsers nor the broader domain of rich Internet applications. In the following sections, we focus on how Flapjax simplifies interfaces for key components of rich web applications, particularly those for GUI manipulation and web services.

Flapjx supports GUI specification, including properties relevant to interactions. We discuss the impact of our constructs in terms of program presentation through compiler support and current best practices such as progressive enhancement. The discussion of GUI data structures motivates an brief examination of cyclic dependencies and recursive constructs to support their expression. A subtlety of the implementation browser renderers, as well as renderers in other domains, is discussed, and we discuss how modifications of our evaluation model or frame-rate sensitive (reactive) constructs can overcome it.

After the client environment is discussed, we focus on the client interacting with services, including those of the web host. General constructs for interacting web services with web services are discussed. Services of the host are of particular interest, especially those providing persistence and security. We show how a typical set/get webservice for persistence can be turned into a binding construct that abstracts away the difference between pushing and pulling. These binding points become convenient locations to place consistency handling constructs. Furthermore, our use of propagation provides an explicit manner of tracking security dependencies, and thus transparently propagating security information and providing a way to react to potentially changing client capabilities.

Finally, we overview some simple demos written to examine our constructs and basic user feedback in response to some of our design decisions. From there, we briefly mention some of the related work again, where we can go next, and review what we have shown.

4.1. Document Object Model

Currently, the most polished portion of Flapjax is in its interactions with the user interface, which is surprisingly difficult given the notoriously inconsistent implementations of standardized interfaces for GUI manipulations. We discuss three categories of constructs relevant to cleanly describing rich interactions in a style closer to HTML than traditional JavaScript. These are GUI data extraction, GUI object construction, and GUI object insertion.

A web page layout is initially described in HTML by a static tree where every child node is nested inside of its parent container. For example, a link may be put inside a paragraph:

```
<p>
    this is my paragraph with
    <a href="http://www.flapjax-lang.org">
        a link
    </a>
    and
    <span id="mytext">
        some more
    </span>
    text.
</p>
```

Text fields, such as '*this is my paragraph with*', '*a link*', '*and*', '*some more*' and '*text.*' are also implicitly nodes in this tree. We refer to this as the Document Object Model tree, or just DOM tree. However, properties of this tree may change over time, especially in response to user interactions. No matter how a component is created, in traditional javascript, its changes over time will have to be externally specified through mutation. Thus, we will see how to extract time varying values, define new ones in terms of them, and define these new values to be actual GUI objects. The realization is that the DOM tree is an external time-varying structure and the challenge is in how to interface with it.

4.1.1 Extraction

The first constructs we introduce are to extract event streams and behaviours from the DOM.

In traditional JavaScript programs, we specify a callback to be invoked when an event of interest occurs:

```
<input id="myfield" onclick="validate()"/>
```

In Flapjax we expose the stream of clicks produce by any DOM object:

```
var clicksE =
    extractEvent_e($("myfield"), "click");
```

or, as it is so common, more simply:

```
var clicksE = $E($("myfield"), "click");
```

All *extractEvent_e* must do is create a new source node, and register a callback that will insert an event into that node:

```
function extractEvent_e(domObj, evt) {
    var node = new Event(
            [],
            function (send, p) { send(p);});
    domObj.addListener(
        evt,
        function (e) {
            propagate(new Pulse(e), node);
        });
    return node;
}
```

The actual implementation is a little more advanced. If there are multiple invocations of *extractEvent_e* on the same DOM object for the same event, all of the events must propagate with the same timestamp so we actually hash extracted event streams. Additionally, there are browser compatibility bugs with the simple code above which we handle, such as not all browsers actually providing the event of interest as a part of the callback, and we are interested in providing a consistent interface. Finally, instead of manually initiating the propagation of a pulse through the graph, we should enqueue the event in the store of external events for subsequent time steps. A more subtle issue regarding reactive objects will be discussed later.

Significantly more prone to compatibility issues is the extraction of values from the DOM and registering callbacks in response to their updates - to the extent that we still have a couple of known issues with environments that are difficult to duplicate. Which field is selected in a drop down menu, the status of a check box, and the contents of a text field or text area are all similar concepts with very inconsistent interfaces, so we provide a simple one: extractValue_b :: DOM $\rightarrow \alpha$, also know as *$B*. It is implemented similarly to *extractEvent_e* except with merging of relevant events and specialized field access depending on the type of the DOM object. We can now easily create a variable that reflects that validity of a form at all times with the following:

```
<form>
    Exactly 3 characters:
```

```
    <input type="text" id="txtField1">
    Exactly 4 characters:
    <input type="text" id="txtField2">
    <input type="submit" id="submitter"/>
</form>
<script type="text/flapjax">
    var valid1 = $B("txtField1").length == 3;
    var valid2 = $B("txtField2").length == 4;
    var formValid = valid1 && valid2;
</script>
```

where we expect the "dot" operation to access a field value, in this case the length, to be lifted. We will discuss optimizations and semantics related to objects and collections later.

4.1.2 Tag Creation

Once we have time varying values, we want to create time varying DOM objects in terms of them, as opposed to encoding such a definition explicitly with callbacks. For example, let us make the form submitable only if it is valid. If we replace the angle braces in the above example with parentheses, we can include validity as part of the definition of the submit button:

```
var fld1 = INPUT({type: 'text'});
var fld2 = INPUT({type: 'text'});
var frm =
    FORM(
      'Exactly 3 characters: ', fld1,
      'Exactly 4 characters: ', fld2,
      INPUT({
            type: 'submit',
            disabled: $B(fld1).length == 3
                && $B(fld2).length == 4}));
```

There are still some undesirable properties of this code. First, instead of writing *$B(fld1).length*, it should be possible to write *fld1.value.length*. However, this is more of an interface issue: we wanted to provide a lightweight wrapper around the traditional, and more verbose, DOM creation methods. Instead of special casing every single field that may be updated due to interactions external to our library (such as by user interaction), we picked a popular subset and guarantee that changes that come from within the system will be detected. Alternatively, we could have isolated all possible events that may cause a member field to change, and then on any such event, check if any field changed. However, DOM fields are different enough between browsers that this might lead to different behaviours between browsers. Additionally, instead of having to name the two text fields through an external variable so that we can both insert and extract them, it would more convenient to just use IDs, especially when there are no cyclic dependencies as in this case.

Using JavaScript to create and modify DOM objects can be slow, so we focus on optimizing our DOM methods. The browser provided DOM library assumes that any given object can exist in only one location on the DOM tree, which we were able to leverage in optimizing lifted DOM creation methods. Consider the following:

```
var oneChild = A({href: 'http://'}, nameB);
var res =
    FORM( styleProperties,
          oneChild,
          arrayOfChildren);
```

With typically lifted code, upon a change in *nameB*, the link anchor constructor *A* will be invoked again to create a new link object, also triggering a recomputation of *FORM* creating another object there as well. If we consider the entire HTML tree and a change at a leaf node, we see that the entire path from the leaf to the root will be recomputed. Given the above property that a DOM object can only be inserted into a tree in one location, we see that if a change occurs, it is safe to mutate the current node and potentially the parent node without recomputing all the way up the path to the root. Currently, we have four special cases:

1. **Value change** If a value (leaf on the DOM tree), such as a time varying string for a DOM text node or attribute value, changes, the parent DOM node will be mutated and the corresponding data flow node notified of a change.

   ```
   DIVB('the time is: ', timer_b(100));
   ```

2. **DOM node change** When the data flow node corresponding to a DOM node signifies a change, the reference to the DOM node is either the same as in the previous step or is different. If the change was due to a mutation, the reference would be the same, and the current node will not have to change in any way. The corresponding data flow node will be notified of a change. If the reference changed, the current (parent) DOM node will be mutated to point to the new reference instead of the old. The corresponding data flow node parent will be notified of a change - in both cases, of the same object reference as the originally created parent DOM object.

   ```
   DIVB( testb ?
       DIV('branch1')
       : DIV('branch2'))
   ```

3. **Dynamic array change** Instead of making a DOM node a function of a fixed number of other DOM nodes, Flapjax supports the definition of one DOM nodes in terms of a potentially time varying array of other DOM nodes. Upon an update, the minimal splice to convert the previous array to the changed array is computed. If there is no change, that means an identity transformation was performed, or an element mutated, with either case requiring no mutations on the parent node but still an upward notification in the data dependency graph. If a change did occur, the exact nodes that changed will be mutated and the notification propagates.

```
var elts =  collect_e([],
  function (v, acc) {
      return acc.concat[LIB(v)]; })
  .startsWith([]);
ULB(elts);
```

4. **Object change** When an object changes, it is either because a nested value changed or a nested object change, with the latter including the possibility that a field was added or removed. Discuss problem of default / lingering values - FX reconstructs, library mode will take an object whose fields may be behaviours, and if those fields refer to values, a change does not cause reconstruction, but if they refer to objects, DOM node will be reconstructed for same stated reason. This case is similar to the DOM node one, except def val issues, so more complicated.

```
DIVB({style: {left: timer_b(100)}});
DIVB({style: testb?
    {left: 100 }
    : {right: 100 }});
```

For example, in the best case, we could just modify the DOM node represented by the return from A. This is not always the case and we may choose to reconstruct the DOM node, so the reference contained within the parent node must be updated to point to the new DOM node. In both cases, we should still propagate the notification of a change and to what value, to data flow nodes corresponding to the parent DOM nodes and ultimately to any nodes dependent upon these, except we can cut back on computation. Specifically, if a time varying array of child DOM nodes changes, the operation can typically be represented as a splice and the corresponding mutation can be carried out. Additionally, if a single DOM node changes its reference (because it was reconstructed), the parent data flow node will be alerted and it will only have to change one the child reference in the corresponding DOM node. As long as no major array manipulation occurs, propagation should be lightweight.

4.1.3 Tag Insertion

Just as we created a construct, *$B*, to extract a time varying value from a fixed position in a data structure, we introduce two constructs to insert time varying values in to a static data structure. Given the statically defined HTML tree that is exposed through the JavaScript DOM interface, we can advise certain nodes in it as being time varying.

Given a static tree structure, there are several points of insertion we can choose around a node. Thus, as shown (Figure 13, page 24), for some fixed target DOM hook node position, we may either want to replace it with a time-varying node or one of its neighbors with one:

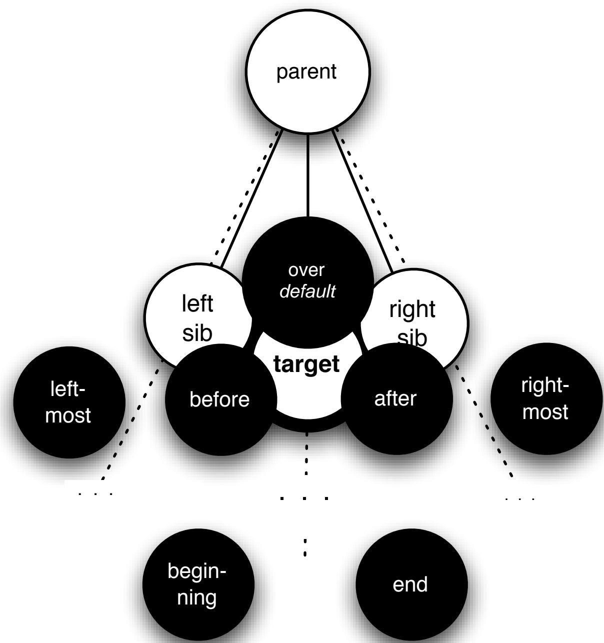

Figure 13. DOM node insertion positions

We provide a function,

```
insertDomB :: DOMNode U String id
* Behaviour DOMNode [* String position] -> Void
```

that will modify an existing DOM tree relative to the first argument to use a time varying node specified by the second argument, with the actual insertion position being specified by the last argument. The most

usual case is to directly replace a statically defined node, so we make the last parameter a default of the 'over' position. The implementation is fairly straightforward given the rules outlined in the tagCreation section with some subtleties mentioned in the time varying data structures sections and below. A similar function, *insertValue_b*, is also provided to make an attribute of a DOM node time varying.

One subtlety arises. Consider the following:

```
<input id="myfld"  type="text"
  value="{! timer_e(10000).startsWith(0) !}"/>
{! (extractValue_e('myfld') + '').
        reverse().
        startsWith('') !}
```

Several interpretations are possible:

1. **Behaviour:** Only the reversed, and updating, time will be printed, ignoring user input This is because the form value was defined as a *Behaviour*, which is defined for all times. Additionally, to be consistent, this would suggest any user change should be instantly overwritten for the view. Such an approach is not as convenient for typical usage as the ones below.

2. **User:** Only values entered by a user will be entered. Programmatic changes can be merged by the programmer, so *extractValue_e* is only a convenience method to capture strictly user input. This approach is tempting because it simplifies the implementation of *extractValue_e* and corresponds to our original approach: only DOM events such as clicks and keypresses must be checked. However, the values do not correspond to what the user sees and is thus counterintuitive.

3. **Unified:** Exact user input can be discerned by filtering out programmatic input, and would suggest a different method name. This could be useful, but in practice, what is desired is exactly whatever is the current value of the form field. Thus, whenever a portion of the DOM is made reactive, any value extraction should respond to such changes.

Essentially, when dealing with shared or persistent data structures, there should be an interface for injecting update events such that changes can be reacted to, and when Flapjax code inserts changes, it should react to its own changes as needed. This can be complicated when an object with several levels is switched in and there is a dependency on some lower level, but, judging by unanimous responses on our newsgroup, the unified approach is expected by the programmer.

4.2. Advice, Progressive Enhancement, Templating

While the majority of users have JavaScript enabled, typically cited as between 90% and 98%, it is still common practice to not assume that JavaScript is accessible: pages are enhanced with JavaScript, not dependent upon it. This technique is referred to as progressive enhancement, [8] as opposed to graceful degradation. Describing a page using HTML and then inserting advice using *insertDomB* supports this style.

If we assume the presence of JavaScript, for small applications, having to separately define the basic page structure with HTML and then elsewhere injecting the advice makes it tedious to correlate interactions with the page structure. Thus, we introduce a syntax to ease this burden. It is similar to data binding support found in other systems [4, 27] except we allow, and promote, time varying objects like DOM nodes, instead of just time varying values like strings. The following two programs are equivalent, with the first being externally advised and the other utilizing a templating syntax that expands to a form similar to the first.

```
//advised
<script type="text/flapjax">
insertDomB('replaceme',
    $B('mybox') ?
        P(
          H1('checked'),
          INPUT({type:'submit'}))
        : H1('not checked'));
</script>
<form>
    <input type="checkbox" id="mybox"/>
    <p>The box is <span id="replaceme"/></p>
 </form>

//templated
<form>
    <input type="checkbox" id="mybox"/>
    <p>{! $B('mybox') ?
                P(H1('checked'),
                  INPUT({type:'submit'}))
                : H1('not checked') !}
     </p>
 </form>
```

The {*!, !*} tags, pronounced curly bang, should surround a string, boolean, integer, DOM object, or a behaviour of one of the same. The latter example expands in to the former, inserting a <*script*> tag wherever the templating syntax is used. For instances in which we

want to use the statically defined HTML structure as a default, we allow events instead of behaviours.

The choice in syntax is largely arbitrary. We could have chosen an xml route:

```
<form>
    <input type="checkbox" id="mybox"/>
    <p>The box is
        <if test="$B('mybox')">
            <true>
                <p>
                    <h1>checked</h1>
                    <input type="submit"/>
                </p>
            </true>
            <false>
                <h1>not checked</h1>
            </false>
        </if>
    </p>
 </form>
```

except we would have to also include namespace information in addition to the above, which we find counterproductive, syntactically.

We also provide a way to define both static and dynamic structure using templating syntax. If JavaScript is enabled, the dynamic structure will be enabled, and if not, it will be ignored and the static structure will be followed. The following two programs take advantage of ||| within our templating syntax, pronounced triple-stick:

```
<form>
    <input type="text" id="myfield"/>
    {!
        $B('myfield').length > 0 ?
            INPUT({type: 'submit'})
            : 'Invalid: type something'
        |||
            Enable JavaScript for hints.
            <input type=submit/>
    !}
</submit>
```

One additional argument for curly bang, as opposed to xml syntax, is that it is easier to discern dynamic elements with triple sticks, easing towards progressive enhancement. Ultimately, a more context sensitive approach may be more appropriate that responds to more than just the presence of JavaScript. For example, handicap related accessibility constraints may be relevant [15] - triple stick syntax requires a separate specification of static behaviour in a manner that is not modular. However, it is lightweight and a start.

4.2.1 Cycles: tagRec

Often, we have cyclic data dependencies, so we provide an abstraction for a common pattern. One example of its occurrence is with a box that highlights during a mouse interaction with it. Consider a box whose border color is defined in terms of the position of the mouse relative to it: when the mouse moves over the box, the border should be green, out, blue, and to initialize, black. The border is a function of the events from the box and an initial value, the events are a function of the box, and the box is a function of its initial border style and subsequent changes. We have two challenges in this case: how can we declaratively describe the actual relationship, and how do we translate this description into valid, representative HTML and JavaScript? The distinction between behaviours and events is useful at this point: one has an initial value, and the other doesn't. As long as we properly initialize our structure, we can describe any behaviours in terms of held events rather than functions of behaviours and thus not worry about cyclic dependencies with no basis. Ideally, we would be using a program representation that could visually show cyclic dependencies without resorting to naming as visually done in MaxMSP, which is difficult with textual representations. However, we can still use absolute IDs corresponding to nodes in the DOM tree (*$E, $B*), or named variables:

```
<div id="bx" style="{! {borderColor:
  merge_e(
    $E('bx', 'mouseover').snapshot_e('#0F0'),
    $E('bx'', 'mouseout').snapshot_e('#00F')).
  .startsWith('#000'))} !}">
    A fabulous box
</div>
```

or alternatively, manually closing the cycle with *receiver_e* and *sendEvent* at a particular node without resorting to global naming to achieve it:

```
var overE = receiver_e();
var outE = receiver_e();
var mydiv = DIV({style: {borderColor:
        merge_e(
            overE.snapshot_e('#0F0'),
            outE.snapshot_e('#00F')).
        startsWith('#000') }});
$E(mydiv, 'over').map_e(
    function(v){overE.sendEvent(v);});
$E(mydiv, 'out').map_e(
    function(v){outE.sendEvent(v);});
```

This latter manual tying of the loop amounts to a callback in message passing style, though the communication channel is explicit through a local proxy node

(*receiver_e()*) instead of a global name (*$E('foo',..)*). However, we can create a function that does the tying for the programmer, providing the proxy streams for the definition of an object and properly extracting the actual changes once they occur, which should not occur until after initialization:

```
var mydiv = tagRec(['over', 'out'],
    function (overE, outE) {
        return DIV({style: {borderColor:
            merge_e(
                overE.snapshot_e('#0F0'),
                outE.snapshot_e('#00F')).
            startsWith('#000')}}); });
```

We create a function that will take the names of streams to extract and a function that defines a potentially time varying DOM object in terms of them. The function creates proxy event streams, invokes the passed in function with them to construct a DOM node, and then extracts the actual streams from the node and connects them to the original proxies. Additionally, as the DOM node may be reconstructed, we take advantage of the *switch* combinators to swap out the old DOM node event streams to be replaced by the new ones whenever the DOM node is reconstructed.

Our provided function is specific to the DOM; if the DOM node changes, we know to call *$E* to extract the new event streams. We can simply generalize it by allowing the programmer to specify their own extraction functions that can be called and manipulated using a *switch_e* whenever the base object changes. Additionally, we could provide compiler support for a cleaner syntax, but do not at this point either.

4.3. Time Sensitive Evaluation Techniques

Web browsers typically interpret scripts in individuals pages using one thread, alternating between script invocations and screen rendering updates. Often, if many updates, especially to the DOM, occur in one time-step, browsers will visibly freeze between updates. Simple examples like the ones we have described above and even complex animations can be made more responsive using the general concept of reactivity, and especially with our particular implementation that contains a data flow graph through which messages are passed.

4.3.1 Scheduling

The problem with a browser stalling due to many updates is common when large array operations occur on DOM objects. By modifying the *propagate* function, we can transparently control when nodes evaluate over time while still maintaining topological evaluation ordering invariants. Specifically, the propagate function can record the time when a pulse starts percolating through the dependency graph, and, if at moment when a node finishes evaluating and returns control to the propagation function, the time stamp is taking a long time to evaluate, can be paused. The *propagate* function will prematurely exit, implicitly running the renderer, but not before setting a timer to resume propagation at the next activated time slice. If browsers ever support threading, programs will already be written in a way such that exploiting inherent parallelism, namely, simultaneous message propagation through time-invariangly disjoint sections of the data flow graph, could be done by patching the library, not the program. Determining whether sections of the graph are disjoint before propagating a message may be complicated by functions like *switch_e* because they facilitate the restructuring of a graph mid-message, but if the semantics are changed such that the restructuring is not active until the next time step, the process would be fairly easy. Furthermore, additional optimizations like choosing when to discern graph disjointness or dynamically profiling to determine which paths should be followed are simple, classic problems.

4.3.2 Framerate

Modifying the scheduler for single threaded applications may distribute the computation load over time, but in some cases, responsiveness is more important than propagating all of the messages in the graph. Particularly, if we create a stream of pulses that occur steadily through time, and computations like animations that are triggered by these pulses take longer than the interval between time, some of these pulses should be ignored. Just as a back-off algorithm may help in the case of congestion in TCP/IP, individual animations may learn to filter out requests. This can be simply done in two ways:

Figure 14. Framerate sensitive animations

1. **Generic Filtering** Instead of modifying the

propagate function to split evaluation over multiple timesteps based on how long the general computation is occurring, we can modify the connection point between the timer guiding the animation and the animation itself to filter out excess pulses. Consider a rudimentary implementation:

```
var now = (new Date()).getTime();
var t = timer_e(40); //40ms pulses
var h = t.collect_e(
    {next: now},
    function (time, acc) {
        acc.old = acc. next;
        acc. next = time;
        return acc;
    });
var f = h.filter_e(function(o){
    return o.next - o.old < 60; });
var time = hold(f.next, now);
```

A more complicated scheme can be built like exponential back-off, as mentioned, but the basic potential is clear.

2. **Specialized Filtering** While the above scheme was useful in one simple demo involving a large array of glowing boxes responding to mouse events, it has the drawback that there may be contention between animations adding noise to their individual optimizing filters and perhaps different animations should be running with a different priority. Currently, no distinction is made between separate disjoint segments of data flow graph implicitly corresponding to separate animations in terms of propagation beyond topological evaluation. However, we can follow the connections from the source or sink nodes of an animation and label the intermediate nodes as belonging to an animation and then start to compute run time for individual paths, and as part of the pulse or less invasively, separate path re-traversals, calculate the cumulative run time and optimize based on that.

The use of functional reactive programming to describe animations is rather convenient as we can use time itself as a behaviour. Additionally, to sequence different stages of animations, we can use either conditionals or more traditional message passing techniques with event streams. This overall usage was one of the original leading motivations for the general technique [14]. It may be possible to find mappings between an executable temporal logic and functional reactive languages that would foster further useful constructs.

4.4. Client Server Relationships

Up to this point, we have focused on peculiarities of specifying clientside computation in a browser, but have not described interactions with non-local systems. Internet Explorer introduced the asynchronous request function that facilitates communication with web services without requiring a page reload, but it is rather simple and does not provide support for the protocols typical of modern web applications. Furthermore, just as shell scripting facilitates composing programs by communication through input, output, and error streams, functional reactive programming simplifies the use of web services within a client program. We currently explore the client server relationship with general services in terms of support for transparency and event manipulation, and a specific but representative service supporting data binding with a persistent store as well as the related discretionary access control policy.

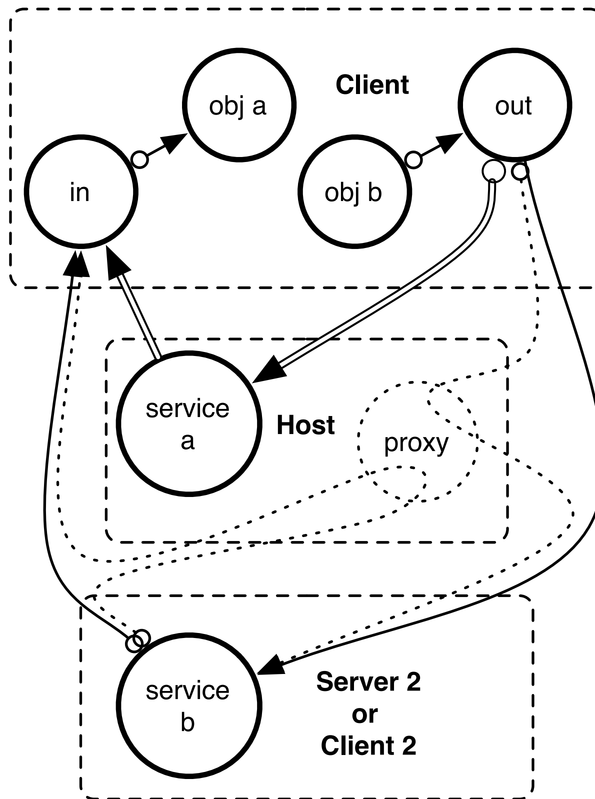

Figure 15. Service Model: Lightweight, corresponding to current practices. Transparent proxy as needed, specialized services for cooperative servers like the host, and basic wrappers for traditional services.

Our current approach is lightweight (Figure 15, page 28) and corresponds to current practices. In the case of a malleable host, we can specialize web services such as those for persistence. When manipulating data from other servers, we provide convenient methods for manipulating them noninvasively, transparently, and with minimal assumptions. Additionally, current browser security models often require significant effort to communicate with services on different domains, so we first try to use specialized client scripting techniques such as using a backdoor in Flash before (transparently) resorting to using a proxy server to make requests. Traditional, fixed services and more malleable ones that can be instrumented are both described.

4.4.1 Transparent Web Services

Our first goal is to simplify the invocation of existing web services and manipulation of their results. First, there is little reason to manually serialize JavaScript objects into JSON, XML, etc. objects for transmission, and de-serialize them on the return. These actions are common enough that we include their support in our library, also supporting various return protocols such as the JSON callback style. Thus, all a user should do is specify the service url, input type to convert to, and output type to convert from. Nominally, assuming a WSDL style web service to specify types, we can make even these latter two steps optional.

Next, we then see that services can be dealt with naturally, described as an input request stream and being transformed into an output response stream. Once services are treated as stream functions, we can use functional reactive combinators. For example, consider a web service that performs spellchecking, or as follows, queries the social bookmarking site Del.icio.us for a user's links as the user types into a text field. Instead of forcing the user to use a submit button, as the webservice is computationally inexpensive and has no persistent side effects, repeatedly calling it and displaying the results as they come in while the user types is acceptable. However, if we synchronize service requests to generate with every character addition, there will be gratuitous results. Instead, we may want to only make the request when the stream of character additions has momentarily paused. We call such a function that holds on to an event for a fixed amount of time and only releases it if no other event occurs before that time *calm_e*, taking the window size as a parameter.

```
<input type="text" id="name"/>
<script type="text/flapjax">
var names = $B('name').changes().calm_e(300);
var requests =
  names.map_e(function(n) { return
      {url: 'http://del.icio.us/rss/' + n,
       serviceType: 'xml' }; });
var news = getWebServiceObjects(requests);
</script>
Current title: {! news.title ||| 'loading' !}
```

The above script is simple yet powerful, just like a shell script. Recently, Yahoo released a service [31] that allows the composition of services through a GUI, but, unlike Yahoo Pipes, we support viewing services as functions over streams and provide programmatic control, temporal operators, and transparent dependency support. IDE support can be built upon such a foundation and will be discussed in the future work section. Finally, while not explored in the current system, message passing can be viewed as a generalization of reactive programming, so nominally, message passing constructs can also be used for services following complicated communication protocols.

4.4.2 Transparent Persistent Store

One of the most common types of web services used in web applications is one that provides access to a persistent store. Often, this is simply a wrapper around a stored procedure to update or extract information in a data base, but the server side implementation can be abstracted away through the web service interface. Whether the interface is manually created or automatically generated, it must exist as the client somehow communicates with the server. Currently, we assume a web service and focus on the question of how to interface with it. Future work may benefit from considering the service layer, such as in automatically inferring desirable web service schema statically based on object-relational mapping or dynamically through an analysis of usage. An examination may be useful as, once a service is exposed publicly, backwards compatibility becomes relevant, and the common expectations of data available through services are becoming more clear. For example, Google web applications never actually delete any data, and given the ad driven infrastructure of websites, per user navigation and data centric trends form context for content selection.

We created a simple web service, providing an interface to a persistent store through which JavaScript values can be manipulated. We support an extension of JSON, including nested and potentially cyclic nested objects but still excluding closures as the semantics are ambiguous and the necessity seems minimal. Our web service permits reading and writing values, specifying access control policies, and simple reflection upon information relating to past manipulations and policies.

With such a simple but rather typical conglomeration of web service features, we can take advantage of reactivity in several ways. We will discuss the benefits related to security in a following section and currently focus on basic data binding.

We structure the store like a file system. Every user has a root object to which programs can add data. Collections, in the forms of arrays and objects, are possible, allowing nesting. While we do not currently provide root objects for individual applications, that can be encoded by the application owner using his or her own root object with that application. For an application to read an object called 'mike' in the array called 'directory', and then locally access the phone field, we write:

```
mike's phone #: {!
    readPersistentObject(
        {path: 'directory/mike'}).phone
!}
```

The lack of a leading '/' in the above path signifies it is relative. Every user has an 'applications' array in their home object, with an entry corresponding to every application used - by default, paths are relative to the entry in the current user's applications array corresponding to the current program. As JavaScript arrays are autoassociative, only applications that have been used will have an entry. Every object, including home objects of users, have a unique id, so by specifying an additional field, relative paths can be rooted by these unique IDs. For convenience, if an optional name field is specified, a path will be relative to the home object of the user with the corresponding name.

Consider the following program to write to a persistent object:

```
<input type="text" id="phone"/>
...
writePersistentObject(
    $B('phone').changes().map_e(function(p) {
        return {phone: p}; }),
    { path: 'directory/mike' });
```

While it is possible to use the unique object IDs, it is generally better to use relative, named paths. If the phone number in the above example changes, a new container object will be created corresponding to 'mike', so any absolute references to the old 'mike' object would become outdated. Allowing the access of object IDs is similar to exposing inode information in a filesystem and can be viewed as being beyond the abstraction barrier. However, if object IDs are entirely abstracted over, changing or obscuring ownership should still be supported in some other way.

As repeatedly changing the phone number in the above example will correspond to creating old, garbage phone numbers with respect to the notion of the current phone number, we have a method to archive old objects that have been recreated. There is merit to storing this data history, even if we do not yet take advantage of it: persistent data in a web application is often subject to changes over time and by multiple users, so basic version control primitives are often desired. As mentioned earlier, Google applications may remove references to data, but the data itself is typically not removed. For example, Blogger blogs are often inadvertently deleted by their owners and thus must be restorable without rolling back other blogs. Our decision to archive but not expose removed objects leaves room for future work. Applications will have different persistence needs, even for different data within the same application, so generalizing these needs to a transparent framework level will probably not involve a one-size-fits-all solution.

Our general insight into supporting binding to persistent stores is that there are two components: the desired programmer interface, and short and long term requirements for stored objects.

1. **Programmer Interface:** First, programmer interfaces to bound data are generally very similar between different services. Additionally, similar manipulations are often made to receive and handle subsequent updates of results to the same query. These motivate creating an interface for reactive values, simplifying the initial manipulation of data as well as simplifying subsequent ones in terms of both service requests and using data elsewhere in the program. First, serialization is simplified, with the programmer only specifying desired input and output types. Next, connections to services are be improved, with our current approach being to prioritize direct connections from the client to a server when the client has the Flash plugin, and otherwise resorting to a proxy. Another strong inspiration to be discussed in the next section is cleanly reflecting upon security capabilities. While developing the library, we also found that the decision between pushing and polling can be just a parameter for binding, assuming the underlying services support the choice. Additionally, data sent to and from the server can be manipulated using stream combinators, eliminating the need for explicit callbacks as noted before and allowing the use of temporal constructs such as our stream limiter *calm_e*. Many of these insights have been previously made as data binding exists in several new interactive web application frame-

works [4, 27], except they do not support first class signals and thus the ability to bind any value in a program, manipulation through temporal constructs, nor other abstractions we build upon our time varying values such as consistency handling.

2. **Services:** Second, interactive and page-based applications can be abstracted to use similar backends through web services. Legacy services from a page-based application can be used in an interactive application, and, conversely, data generated through a web service that an interactive application would rely upon should be accessible to page-based applications. Currently, we constructively show a service may be used through our binding interface by doing so, and we believe object relation mapping techniques can be used to automate this step for more finely structured services that already exist. As a minimum, our general web service interface should be sufficient for using typical services in Flapjax applications, demonstrated by several demos interacting with popular services, even though this alternative does not provide infrastructure for consistency guarantees. In the converse, by using a persistent store service with Flapjax, page-based applications can automatically also access the persistent data. The logical next step would be to create finer object-relation mappings for such services rather than directly exposing our store which coarsely represents data in our extended variant of JSON. Explicit schema specification, or even dynamic inference, are plausible future directions to generate finer web services. Additionally, as many existing persistent stores do not expose information necessary for consistent binding constructs, an interesting problem is how to non-invasively extend these services to provide consistency information.

4.4.3 Consistency

Flapjax facilitates the simple enforcement of consistency constraints on bound persistent data. Just as it may be useful to specify whether a bound datum should be pushed or polled, the data binding point is a good place to specify what consistency constraints should govern persistent data readers and writers on a page in case there are races with users on other pages modifying the data.

We currently provide one simple default race handling mode for a client that both reads and writes to the persistent store. In all cases, messages with old local (client) time stamps or global (server) time stamps are ignored. Our protocol is as follows:

1. **Write:** A write to the server is accepted.
2. **Read of write from same client on same page session:** Ignored. To receive these writes, a client can use *merge_e* to incorporate the stream of successful writes as the stream is returned by the change stream binding function.
3. **Read, otherwise:** Accepted.

Consider a textfield bound to a persistent datum. The above default protocol supports binding the write stream emanating from the field and merging in the read stream from the server with the intuitive results. We slightly extended this to optionally group change streams into sets that are not necessarily disjoint. Persistent data readers may specify which change streams to potentially filter out, and if none are specified, all persistent data write streams are considered, giving the expected default behaviour.

Note that the second rule prevents the cloning, or Orbitz, bug [21] if directly recreated: if a bound item is viewed in one window, and then modified in another, the view of the first will also be modified. Furthermore, the use of interactive applications with local state would further decrease the likelihood of this style of bug. These rules also prevent tight cycles in bound form fields in which a write by the client would trigger a later read from the server, which would output to the form field and cause another write, perpetuating the cycle. While it is conceivable to apply such an approach to similar cycles that occur entirely within the client, lenses used on top of Flapjax may be more appropriate [5] as races are not possible.

Other consistency handlers may be useful, especially if integrated with GUI widgets. For example, when multiple users are editing a datum, displaying their names may help, as would explicit options for merging in conflicts between users. This latter case may occur if one user edits data, then the second user does as well but based off of the original version. Additionally, if the data is large as in a text document, the changes may not necessarily conflict in this situation. Our main point is the data binding points for input and output are useful locations for specifying how to handle these constraints, with reactivity being the natural way to propagate changes.

4.5. Security

A common trait of web applications is the ability of users to share data, so in our persistence web service, we provide minimal support for discretionary access control. Our approach has two key features: it is built

in transparently, and reflective. As always, reactivity simplifies these interfaces.

4.5.1 Transparent Security

Just as in h our storage of data, security is user oriented, as opposed to application or code oriented. Requests by users to perform actions on persistent resources are monitored. Currently, we do not guarantee end-to-end security by securing the client but assume whatever program uses the web service and our corresponding client-side library is secure for some vague notion of the term. Instead of focusing on providing security guarantees, we begin to question how a web application writer would like to integrate security concerns with an application. Thus, every persistent object has an associated access control list stating what user can perform what action on an object. Currently, users can create new objects, and read and write values and references within objects. The existence of a tuple $(user_a, action_a)$ in the access control list associated with obj_b signifies that user $user_a$ can perform $action_a$ on object obj_b. Unless otherwise specified, a new object will inherit the access control list of its initial parent object, with users secured against each other by default by only have themselves listed in their home objects. While we do not support role based access control, we do support two groups: all users, and logged in users. If a mandatory access control policy was also used to add global constraints, requiring a user to be logged in may help with quality of service concerns in terms of resource abuse and tracking.

The current server providing our persistence web service offers free hosting of Flapjax using applications. Multiple applications owned by different users may be present, so we currently require changes to access control policies to be done through a miniature application we host as a means to guarantee requests to share objects were intentional. This constraint can be relaxed in more typical hosting environments, in which case the methods our miniature application uses could be exposed as web services. Currently, a web developer may create a form in which the user enters the email address of a friend they would like to share an object for reading with, which would generate a link to a prefilled page in our sharing console for confirmation. As part of our intent to work with existing services, supporting OpenID may be a good choice. Compared with the rest of Flapjax, this portion of our system is in early development.

The essential result is that an application writer can write a basic application for one user, and when done, support sharing by simply adding links to share information based off of an identifier such as a user name or email address.

4.5.2 Reflective Access Control Policies

If capabilities defined within access control policies can change over time, an application designer should indicate what a user can and cannot do to some extent [19]. For example, if whether a field is read-only cannot be determined when at development time, such as when the object is sharable, the field should be rendered as plain text or as text input field depending on the current capabilities of the viewer. Thus, the program should be able to reflect upon the *current* security policy. Data binding the access control list of a persistent object to a clientside behaviour is a natural approach with Flapjax. There may be an inadvertent information leak if a user may learn about the entire access control leak, so we currently only allow a user to reflect upon actions he or she may take, and view this as a challenge to web programming and security policy communities orthogonal to the use of reactivity. We may introduce more fine-grained concepts such as metapolicies, but currently have little motivation to do so.

4.6. Objects and Collections

One of the general difficulties with functional reactive programming is to efficiently propagate changes of collections and other data structures. For example, if an array is defined to be the same value as another array except every element is incremented, and only one entry in the dependent array changes, only one entry in the generated needs to be changed. Binding to nested collections on a server further complicates the implementation.

4.6.1 Delta Propagation

As mentioned earlier, when a change notification from an array of children reaches a DOM node, the minimal splice is computed to represent the change, after which only the calculated elements must be recomputed. However, consider the following example:

```
var x = map(
    function(v){return v + 1; },
    arr);
var y = map(
    function(v){return v + 1; },
    x);
```

We take an array, add one to every element to create an intermediate array, and then create a final array

that adds one to every element again. In this case, if no computation is dependent upon x, the entire code segment should be compacted to adding 2 to every element of arr. Otherwise, in a recomputation, if the elements of *arr* that changed are known, we can propagate that labeling to the dependent recomputations, eliminating the need to calculate the minimal splice we found in the case of DOM node changes. We have not yet implemented this optimization in the library, but believe it to be beneficial.

4.6.2 Shallow vs Deep Binding

It is not even always desired that the check for a change to occur. Consider the case of an object located on a server containing a photo gallery. It can be bound to in order to extract the name of the gallery, but if this is the extent of the information used, no knowledge of the actual photos is necessary. Binding to the object would transfer it, and its children, to the client. If another user changes a photo, represented as a child object of the gallery, the client would be able to discern that the change does not impact the gallery name. This is unnecessary, wasting processing time on the client and server as well as bandwidth. If we represent objects as edge labeled graphs where each node correponds to an object, edge label to a field, and fringe nodes as values, we see that in this scenario, the binding is only needed from the gallery object to whatever node contains the name. Ideally, this would be done automatically by the library, but for now, we allow the user to perform a *shallow* binding, as opposed to the default of *deep* binding, to only receive the desired layer of an object. If we monitored field access to bound objects, we could perform this optimization transparently, extracting only necessary fields from the server and binding shallowly to them. Further optimizations typical of pre-fetching would be natural.

4.6.3 DOM Binding

The DOM tree is an interesting data structure because, as it changes, it also performs computations. For example, we have been assuming throughout this paper that, if a field of the DOM changes, the change propagates to the display. Additionally, changes may occur that do not originate from the Flapjax framework, performed instead by some user library. Finally, as we allow changes to the DOM to occur as advice to specific locations in the DOM tree, instead of forcing the developer to define the entire page dynamically, changes may be injected into both a parent and a grand child node. All of these scenarios imply that there may be dependencies in the data structure not modeled by our dependency graph as presented. This is true of externally maintained data structures in general and is thus an important consideration for any functional reactive system that is meant to integrate with other systems.

Our approach to this issue is simple: at an insertion point, made by an invocation to *insertDomB*, if there is a change, we also propagate the change to the event stream *topE*. Thus, when extract values, we may also check *topE* on top of the other events that may signify a change. For a library to insert its own changes, it must either insert notifications into dependent reactive values, or more simply and generally but at the cost of efficiency, into *topE*. This is distinct from the previous section on *$B* which implicitly merges in changes from *insertValue* as the relationships would be known.

4.6.4 Server Binding

In the case of the server, we propagate changes based on labels showing which user last modified a value and when. Currently, we achieve this by the client polling the server for bound values. Our current reasoning is that, first, this is a simple interface, and second, the server does not have to use resources to use resources to track what users are bound to what data in case a change occur. Our decision is sufficient for scenarios with very little users on a single server or many users on many servers, but a different approach may be desirable for scenarios that are mix of these two. Dependency tracking may move to the server, in which case the server would *push* changes to the client, as opposed to clients repeatedly *polling* the server in case a change occurred since the last data request. This is an implementation concern separate from the use of the bound data; with a data binding abstraction, the program developer potentially would only need to specify which choice to make:

```
readPersistentObject({mode: 'push', ...});
```

Thus, we see that there are subtleties and performance issues surrounding collections, and exasperated by web programming, but it should be possible to automate optimizations we would normally make by hand.

4.6.5 Disconnected Nodes and Garbage Collection

Our current implementation has avoidable space and time leaks. Some are related to bugs with garbage collectors of popular browser implementations, which we will not discuss. The other is due to unused data flow nodes. If the value of a node does not eventually flow into a persistent data binding nor is inserted into the

page, it has no observable effect beyond wasting computation time. The one exception is if the user adds imperative code into a node, which we will consider to be the same as inserting the value into the page. Thus, if a node is not used in such a manner, it can be disconnected from the data flow graph, as can anything dependent upon it. A subtlety is that, even if the node is not directly connected to an output node at a given instant, it may eventually be due to dynamic data flow constructs such as *switch_e*. It is possible to temporarily disconnect these nodes and reconnect them later, but this would impact intermediate nodes such as *collect_e* that can be used to gather state over time. We do not perform such an optimization at this time but may in the future after more work on the desired semantics of Flapjax.

4.7. Evaluations

While we have not run explicit user studies, Flapjax has been used by our group as well as other parties.

4.7.1 Demos

Throughout the course of the project, we have written demonstrations of how particular features of our system work. These include a sharable tasklist application as well as using a slider widget provided by the Scriptaculous and both controlling and extracting its values. These uses show Flapjax can be used to create new components and interface with existing ones. More generally, they demonstrate Flapjax is useful for rich computations and web computations.

4.7.2 Applications

We have received short evaluations from two groups on their use of Flapjax. The first, a London-based commercial web design firm, is using the client side capabilities of Flapjax to create a data grid widget and also taking advantage of reactivity to create a simplified interface to their own web service. The second evaluation is from a group developing an interactive Wiki.

1. **Data Grid** The commercial design firm found room for improvement in terms of speed, but the summary is that "Flapjax performed well" in terms of "shorter code and a faster development cycle", also with the fringe benefit of insulating the developers "from most of the cross-browser issues."

2. **Wiki** The evaluation of the use of Flapjax in the development of a Wiki said much of the same as the above. In addition, the developers found that *Behaviours*, 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. Thus, much computation in the Wiki system is done in terms of events, with the cost of initializing the graph, but not creating initial output. Thus, the developers found they preferred to use Flapjax as a library, coding in a style known as unobtrusive javascript, advising changes to the page.

4.8 Errors

A common question is how to handle errors in Flapjax, in terms of actual application code as well as the debugging process. We do not have an explicit method for the other as we are still developing best practices. For example, we separate the return of a web service into a failure stream and a result stream. Additionally, we have a global error stream that can be tracked.

Debugging JavaScript is a problem for rich web application developers, with no system that matches the tools available for most other languages with even remotely similar use. However, just as our system simplifies the specification of systems with convoluted control flows due to the use of callbacks, we believe it also simplifies debugging them. Individual time varying values may be monitored and dynamically checked against time-invariant assertions. Furthermore, as we simplify interactive code by making it composable, isolating an error simply requires isolating which variable is defined incorrectly. Finally, previous work has shown how reactive concepts can be used to simplify the debugging process when a program is treated as a generator of time varying values[22], which is true of Flapjax programs.

4.9 Library vs Language

As found within the user evaluations, the library and language modes have varying receptions. We have found these to be due to different reasons, often depending on the scale of application being developed.

4.9.1 Dynamic Typing

The lifting transformation automates lifting functions to operate upon time-varying values. However, the choice of which lift function to apply, *lift_e* or *lift_b*, for events or behaviours, respectively, is one way to document the types of the arguments within the code. JavaScript is a dynamically typed language and in common usage types of data are not specified, so while it

is clear from context that a value may be, for example, a time-varying integer, it may not be as clear if the value is a *Behaviour* or an *Event*. This is an important distinction, as suggested by the existence of the *change* and *hold* operators and also discussed earlier, so even something as simple as optional type annotations may greatly improve usability. One recent approach has been the development of a contract system. Additionally, as users become more comfortable with the library, it appears the developing small portions of code may be done in Flapjax. The decision to develop larger portions of code in the library will be discussed in subsequent items. Additionally, the order of reactivity of an object may be unclear, especially as we remove the idempotency rule *Behavior Behaviour a = Behaviour a*. Recent discussions have arisen on specifying function argument reactivity order and lifting based on it.

4.9.2 Optimization

Flapjax is new, with much of the development focus currently on designing appropriate interfaces and realizing them with various systems. Thus, while there is work on optimization as noted in previous sections, there is a long list of other known techniques that can and should be implemented. Thus, system users may find they want to make particular optimizations in some of their more rich and interactive code and thus want to use the library mode in which the data flow graph is exposed in terms of *Event* and *Behaviour* nodes.

4.9.3 Components

While not directly a reason to choose one approach over another, a clear missing feature from the current system is the inability to define new components. For example, there is a popular proposal to extend HTML to include a calendar component, yet such a component can be generated from existing HTML elements: instead, it should be possible for developers to define their own components. Similar new markup languages allow the specification of new components [4, 27, 9] and it would be an obvious choice for Flapjax as well. Additionally, given the functional style of Flapjax programs, parens can be replaced by angle braces and thus allow users to never leave the component language. This latter approach would not work in the other systems as there is no way to declaratively, or at least functionally, describe interactions and animations. Finally, we do not have a module or unit system, so abstractions relevant to larger applications are not in place, just as they are not in current specifications of JavaScript. The lack of components and modules makes the language mode less effective for large applications.

4.9.4 Compilation

We provide an open source compiler that users may install on their system, as well as a web service and an online interface to always be up to do and facilitate simple test scripts, respectively. However, we see merit for adding two extensions: first, incorporating Flapjax into the publishing phase of a server, and second, including source code in a compiled page. The former is mostly about convenience and optimization, while the latter has pedagogic and adaptation value: being able to view the source of an active website and understanding it will aid those newer to the system. The latter can trivially be achieved by just outputting source in comments, but compilation on page load would be cleaner and better facilitate robustness as the compiler evolves.

5 Related Work

Flapjax extends existing work with functional reactive and data flow programming [14, 29] in an adaptation of an existing embedding style [11] for dynamic data flow, focusing on the browser environment and interoperability with other libraries. This includes integration with object oriented code bases [18, 23, 12], but with more of an orientation towards persistent data structures with shared control. Additionally, on top of distilling what it means to be a rich web application, we further discern what exactly is meant as a GUI value by using reactivity [14] and focus on allowing traditional coding practices such as mutation when transparently incrementalizing code [2].

6 Future Work

There are many directions we would like to take Flapjax, with the main constraint being time. These largely fall into four categories: functional reactive programming expressiveness improvement, compiler optimization, web programming support, and general language additions.

1. **FRP Expressiveness**

   While there have been previous attempts in interfacing functional reactive programs with more traditional languages such as object oriented ones [18, 23], our experiences with reactivity constructs allowing the simplification of interfaces suggests this is a useful area for further examination. Furthermore, (external) persistent data structures are

used twice with Flapjax: the DOM tree and the persistent data store. Arguably, we did not run into the object oriented code interfacing issue because there is currently no dominant object oriented system in JavaScript beyond the simple prototype support, but this related problem is still relevant and possibly more general.

Additionally, we find users want to mix imperative code with reactive code. Largely, this is to write traditional iterators or intermediate named values for a long computation, which can be supported. Effectively, users often want to write small function bodies with impure JavaScript, so an option is to only have reactivity extend to the function border. Current work in the vein of [2] suggests ways to support mutation when dynamizing a static algorithm, which might provide some insight into a possible semantic for mutation.

While functional reactive programming was originally introduced to simplify animation, it has received little attention from the intended audiences, probably due to the choice of embedding languages. Our current embedding has fared slightly better, and given the recent efforts by the animation community with Processing to modernize animation specification techniques and make them more accessible to artists, we are interested in investigating better animation support. For example, object oriented models and temporal operators might be explored in how to achieve actions like sequencing, or even rewinding, animations with nonlinear timelines and complex interactions. This seems like a very rich area, and given Flash, a popular animation environment, relies upon a language similar to JavaScript, we are in a good position to leverage our existing system.

Furthermore, it is currently difficult to write programs with cyclic dependencies. We provide *tagRec*, and the more general *genRec*, but believe this is not entirely sufficient. One approach that we are investigating is the use of bidirectional programming but have several other ideas as well. For example, if a cycle is tied by using a callback (instead of requiring the knot to be explicitly defined as in a letrec), the decision of which edge to make a callback is arbitrary, and it might be possible to automatically restructure code at will to switch which dependency is the callback.

2. **Compiler Optimization** Many useful compiler optimizations were mentioned earlier. The question of finding the appropriate in between point between lifting and just chain compaction is largely unexplored, as is efficient manipulation of change propagation relating to collections. Much existing work is oriented towards finding appropriate semantics, with less of an emphasis on performance.

3. **Web Programming** We have discussed Flapjax as an extension of the style of embedding of reactivity demonstrated with FrTime [11], and how it can be used to simplify rich web application programs. However, Flapjax is intended to be a useful system beyond this initial insight: there are future directions we should take to simplify web programming, with or without reactivity. For example, we are interested in the seeming duality between page-based applications in which computation occurs on the server and rich web applications. One goal is to integrate Flapjax with one such system to simplify porting of legacy code, or even code reuse in the case of concurrent development. Program evolution is important in the life cycle of a web program, especially given the low entry barrier for small sites, and the unpredictable power law activity and usage patterns: first to market and scalability are desirable features, with scriptable systems like Flapjax aiding the former, but, counter-intuivitively, page-based versions for the latter.

   A difficulty we've found was with consistently inserting and extracting values from the DOM, and potentially, the server, so a separate, nascent bidirectional programming library has been built that can interface with Flapjax. While this is a more general solution, it may be unnecessary in the domain of web applications. Bidirectional program is useful in building these initial widgets, but the developer's role of connecting them together may be more conveniently done reactively. A widget library, and common consistent binding mechanisms integrated with them, would greatly aid the usability of Flapjax.

   Finally, we should focus on an approach to providing strong security guarantees. Popular browsers provide a simple, ineffective, and even occasionally counterproductive security model. Messages can be passed between the client and the host server, but to receive data from a foreign server, foreign code must be evaluated. Even if we address this issue, as hinted by our support of access control list manipulation in conjunction with our persistent data web service, we our interested in securing particular data, and thus may want to enforce information flow guarantees like noninterference.

Applications may be hosted together, or can otherwise be mutually verified or trusted, so the more abstract security model appropriate from rich web applications should be discerned.

4. **General Purpose Language** Our target embedding language, JavaScript, was chosen due to unparalleled (and possibly, unparalellable) penetration, developer familiarity, and more technically, because of the support for first class functions. Otherwise, popular JavaScript implementations are rather sparse in features. For example, developers have often found its prototype based object system to be insufficient for direct use and create their own object systems. While, as mentioned earlier, we are interested in how to extend an object system to take advantage of reactivity, developers using our system are still left with the question as to how to modularize their code. Reactivity allows one to compose programs that traditionally could not because they were represented by callbacks, but beyond function composition, further support of composition is lackluster. Additionally, tool support would be useful for Flapjax (and JavaScript). Debugging is simplified with reactive programming as control flow is simplified and reactive scripts can be written to monitor applications traces [22] as previously mentioned, but testing harness and IDE support does not currently exist for Flapjax. Data flow programming can be usefully visualized, as previously mentioned in conjunction with MaxMSP, so richer models than abstract syntax trees may lead to useful tool support (hinted by the previous point on cycles).

7 Conclusion

We have introduced a new web programming language that can also be used as a library for the dominant rich web application language. We see rich interactions, web services, persistence data, and discretionary access control as being common capabilities of rich web applications and build support for all of them into our libraries. Importantly, we see including functional reactivity as part of language simplifies interfaces to all of these capabilities. Finally, we have further developed techniques for library and compiler assisted embedding of functional reactivity into traditional programming languages. While still at an exploratory stage, the current implementation and feedback from real-world deployments are promising.

8 Appendix

See http://www.flapjax-lang.org for demonstrations, documentation, client-side libraries, the compiler, and server functionality.

References

[1] A. Abualsamid. A flexible scripting language for building dynamic web pages. In *Dr. Dobbs*, 2001.

[2] U. Acar, G. Blelloch, R. Harper, J. Vittes, and M. Woo. Dynamizing static algorithms with applications to dynamic trees and history independence, 2004.

[3] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, 2006.

[4] Adobe. Flex 2 technical overview. http://www.adobe.com/products/flex/whitepapers/.

[5] A. Bohannon, J. A. Vaughan, and B. C. Pierce. Relational lenses: A language for updateable views. In *Principles of Database Systems (PODS)*, 2006. Extended version available as University of Pennsylvania technical report MS-CIS-05-27.

[6] K. Burchett, G. H. Cooper, and S. Krishnamurthi. Lowering: A static optimization technique for transparent functional reactivity. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 2007.

[7] N. Cannasse. haxe: A cross-platfom web language. *Open Source Convention*, July 2005.

[8] S. Champeon and N. Finck. Inclusive web design for the future with progressive enhancement. In *South-by-Southwest Interactive*, Austin, Texas, Mar. 2003.

[9] B. Cohen. Silverlight architecture overview. http://msdn2.microsoft.com/en-us/library/bb428859.aspx.

[10] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. http://groups.inf.ed.ac.uk/links/papers/links-fmco06.pdf.

[11] G. H. Cooper and S. Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming, 2006*, 2006.

[12] A. Courtney. Frapp: Functional reactive programming in java. In *Proceedings of Practical Aspects of Declarative Languages*. Springer-Verlag, Mar. 2001.

[13] A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18, Uppsala, Sweden, Aug. 2003. ACM Press.

[14] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[15] K. Gajos and D. S. Weld. Supple: automatically generating user interfaces. In *IUI '04: Proceedings of the*

9th international conference on Intelligent user interfaces, pages 93–100, New York, NY, USA, 2004. ACM Press.

[16] D. Herman. Formalizing javascript. http://calculist.blogspot.com/2006/06/formalizing-javascript.html.

[17] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming, 2002.

[18] D. Ignatoff, G. H. Cooper, and S. Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In *FLOPS*, pages 259–276, 2006.

[19] A. Kapadia, G. Sampemane, and R. H. Campbell. Know why your access was denied: regulating feedback for usable security. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 52–61, New York, NY, USA, 2004. ACM Press.

[20] S. Krishnamurthi, P. W. Hopkins, J. McCarthy, and M. F. Paul T. Graunke, Greg Pettyjohn. Implementation and use of the plt scheme web server. *Higher-Order and Symbolic Computation*, 2007.

[21] D. R. Licata and S. Krishnamurthi. Verifying interactive web progams. In *IEEE International Symposium on Automated Software Engineering*. IEEE Press, 2004.

[22] G. Marceau, G. H. Cooper, S. Krishnamurthi, and S. P. Reiss. A dataflow language for scriptable debugging. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 218–227, Washington, DC, USA, 2004. IEEE Computer Society.

[23] S. McDirmid and W. C. Hsieh. Superglue: Component programming with object-oriented signals. In *ECOOP*, pages 206–229, 2006.

[24] N. Mix. Narrative javascript. http://neilmix.com/narrativejs/doc/index.html.

[25] J. Peterson, V. Trifonov, and A. Serjantov. Parallel functional reactive programming. In *Proc. 2nd Int'l Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 16–31, Mass., USA, Jan. 2000. Springer.

[26] M. Serrano, E. Gallesio, and F. Loitsch. Hop, a language for programming the web 2.0. In *Proceedings of the First Dynamic Languages Symposium*, Portland, Oregon, Oct. 2006. ACM Press.

[27] L. Systems. An open architecture framework for advanced ajax applications. www.openlaszlo.org/whitepaper/download.

[28] B. Taylor and B. Johnson. Using google web toolkit. *Open Source Convention*, July 2006.

[29] T. Uustalu and V. Vene. The essence of dataflow programming. In *APLAS*, pages 2–18, 2005.

[30] T. M. VIII, K. Crary, R. Harper, and F. Pfenning. A symmetric modal lambda calculus for distributed computing. *LICS*, 2004.

[31] Yahoo. Yahoo pipes. http://pipes.yahoo.com/pipes.