Arrows in Commercial Web Applications

Eric Fritz *, Jose Antony †, and Tian Zhao ‡
Department of Computer Science, University of Wisconsin – Milwaukee
Milwaukee, WI 53211
Email: * fritz@uwm.edu, † jantony@uwm.edu, ‡ tzhao@uwm.edu

***Abstract*—Developing a scalable and robust web application is difficult. One obstacle is JavaScript's event model, which makes asynchronous programs hard to maintain and extend. This paper describes the case study of an *Arrow*-based JavaScript library for asynchronous computation. Our library enables more modular and understandable applications and it improves productivity with static type inference.**

I. Introduction

JavaScript remains the dominant language of the Web. Accounting for scalability and reducing code complexity is critical for the success of any software application. Web clients are often tasked with receiving and processing large sets of data, which often requires clever tricks on the part of the developer (e.g. caching, paging, and prefetching). As part of making the client smarter, it is important to choose the right tools that provide developers the ability to create more efficient applications without increasing maintenance costs.

In this paper, we explore the applicability of an *Arrows*-based JavaScript library [1] which abstracts asynchronous computation on the client in the context of commercial web applications. To evaluate the library, we re-engineered a real-world application using Arrows as the primary framework. We found the Arrows code to have greater maintainability over the vanilla JavaScript version. It was trivial to add additional optimizations to make data handling more efficient.

The remainder of this paper is organized as follows. Section II provides a description of the Arrows library. Section III describes the structure and implementation of the application under evaluation. We contrast an Arrow-based implementation with a 'bare-metal' JavaScript implementation. Section IV discusses an optional type system on top of Arrows which aids the developer during design and implementation. Section V lists related work and and Section VI concludes. The source of the library and the application is available at the project homepage[1].

II. Event Programming, Promises, and Arrows

In this section, we give a brief introduction to event programming in JavaScript. We limit our focus to the following three choices: registering callback functions to fire when an event completes; using Promise objects to flatten nested callback functions; and using Arrows to *describe* an asynchronous computation.

[1]http://arrows.eric-fritz.com

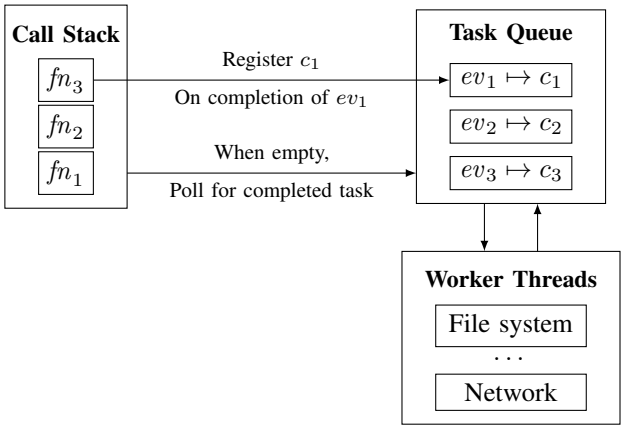

Fig. 1: The JavaScript Event Loop. The call-stack is cooperative. Code can register a callback with a task. Once the current call-stack is empty, a callback for a completed task is invoked.

A. Event Programming & The Event Loop

In many programming languages, concurrency is achieved through preemption. When a thread of control reaches a point where it blocks (e.g. accessing disk or reading data from a socket), it yields control and allows another thread to run. Contrarily, JavaScript executes all code synchronously in a single thread. While code executes, it can *register* a chunk of code to be invoked on the completion of some external event (e.g. a timer triggering, requesting data from a remote server, a user interacting with a field). Once the current thread of control runs *to completion*, the queue of registered tasks is polled for a completed event. If one is available, the chunk of code associated with the event is invoked. If one is not available, the interpreter blocks. Once the task queue is empty, the program halts. This sequence of events is illustrated by Figure 1.

No call in JavaScript that requires access to an external resource *blocks* – instead, it accepts a *callback function* which is registered to the completion of the event. Callbacks are seen as a form of continuation-passing-style (CPS) [2]. On the client-side, this allows responsive handling of events; on the server-side, this allows concurrent request processing.

An empirical study by Gallaba et al. [3] characterizing the use of callback functions in JavaScript has concluded that, on average, every 10^{th} function definition takes a callback argument, every 5^{th} function callsite takes a callback argument, and 72% of all callsites in client-side code are asynchronous

(eventually calling an asynchronous function).

Unfortunately, the heavy use of callbacks creates a non-linear flow of control. Often, callback functions are declared anonymously and cannot be invoked except from their point of definition, which hinders function reuse. Callback functions can be nested to arbitrary levels. All of these features make callbacks difficult to understand and maintain.

Furthermore, the built-in `try/catch` exception handling mechanism works only for synchronous code. There is no natural way to catch an exception which is thrown asynchronously. Over time, an *error-first protocol* idiom has emerged within the JavaScript community. In this idiom, the first argument to a callback is reserved for an error value. If the value is non-null, the function invoking the callback has, in some manner, failed. Unfortunately, adherence to this convention is non-enforceable.

```
http.get('/articles', (req, res) => {
 db.openConnection(host, creds, (err, conn) => {
  if (err != null)   res.send('Error: ' + err)
  else {
   conn.query('select * from articles', (err, results) => {
    if (err != null) res.send('Error: ' + err);
    else             res.send(results);
   });
  }
 });
});
```

Fig. 2: An example use of asynchronous callbacks and error-first protocol on the server-side.

Figure 2 shows an example use of callbacks. The first callback defines the function that handles a request to the `/articles` endpoint. The second callback is invoked after a connection to the database has been opened - if a database connection could not be created, then `err` is expected to be a non-null value. The third callback is invoked after the select query is executed. Again, the `err` value is non-null if the query does not succeed. Anonymous callback definition and callback nesting, found to be widely used in practice [3], taxes code comprehensibility.

B. Promises

The JavaScript ecosystem has seen the need for improving event programming idioms and has responded accordingly. jQuery 1.5 introduced *deferred objects* and other popular JavaScript framework and libraries have popularized *promises*. With ES6, Promises have arrived natively in JavaScript.

A Promise is an object which represents the result of a (possibly asynchronous) computation. The promise object may be in one of three states: *pending* (the required event has not yet fired), *resolved* (the required event fired successfully), or *failed* (the required event has fired unsuccessfully). A callback can be registered to the promise to be invoked when the promise transitions to either the *resolved* or *failed* state.

Promises provide a way to *compose* callbacks, and also provide a means of error handling. Figure 3 shows a Promise version of the callback-structured code in Figure 2. Notice the difference in nesting levels as well as the de-duplicated error handling code. In this example, both `openConnection` and `query` return a promise object instead of accepting a callback parameter. The user registers a function to be invoked on *success* with the `then` method, and a function to be invoked on *failure* with the `catch` method.

```
http.get('/articles', (req, res) => {
 db.openConnection('host', creds)
   .then(conn => conn.query('select * from articles'))
   .then(res.send)
   .catch(err => res.send('Error: ' + err);
});
```

Fig. 3: Using promises instead of callbacks.

Promises also provide two functions for convenience. The method `Promise.all` takes a collection of promises and resolves once *all* of the promises resolve, and the result of the promise is a symmetric collection containing each result. The method `Promise.race` takes a collection of promises and resolves once *any* of the promises resolves, and the result of the promise is the value of the promise which resolved *first*.

C. Arrows

Arrows [1] are a more expressive solution to the problem of callbacks inspired by the work of Khoo et al. [4]. An arrow is a function that takes some input and produces some output, possibly asynchronously. The value produced by an arrow can only be consumed by another arrow. A JavaScript function can be *lifted* into an arrow.

Arrows subsume the sequencing and error-handling features of Promises as described above. However, there are some major differences. Instead of representing the *value* of a computation, an arrow represents the *computation* itself. Composing arrows together builds a state machine which describes some sequence of user interaction events, access to external resources, and code execution. Arrows, in addition to sequencing and error handling, provide a means for *canceling* computation, *repeated* computation, and a more expressive version of the `Promise.race` method which allows the user to configure the definition of a *winning* computation (e.g. the first arrow to resume after blocking).

```
const value   = new LiftedArrow((el, ev) => $(el).val());
const fetch   = new AjaxArrow(q => {'url': buildUrl(q)});
const display = new LiftedArrow(list =>
  $('#results')
    .empty()
    .append(list.map(x => $('<li />').text(x))));

const handler = Arrow.seq([value, fetch, display])
  .on('keyup');

Arrow.seq([new ElemArrow('#input'), handler])
  .forever()
  .run();
```

Fig. 4: An example application using Arrows which populates a list in the DOM as the user types.

Figure 4 illustrates a minimal autocomplete example using Arrows. In this small example, two functions are *lifted* into two respective arrows (lines 1 and 3-6). The arrow `value` takes a text input element and returns its value. The arrow `fetch` takes a query string q as an argument and fetches results from a remote server. The arrow `display` takes a list of results from a remote server and appends them to a list element.

The composition on lines 8-12 create an abstract computation which is finally `run` on line 13. Arrows divorce the definition of computation from execution, allowing an arrow to be created without being executed. This allows complex arrow composition to be used in multiple arrows. To contrast, Promises have no such delineation which tends to hinder reuse.

The `ElemArrow` object is an arrow which simply returns a DOM element. The `seq` method is equivalent to the `Promise.then` method. The `on` method takes an element and waits for an event to fire on that element (in this case, the *keyup* event of the *#input* element). The handler of the `on` combinator is invoked once with the element and event object as arguments (and is then immediate de-registered). The `forever` method simply re-invokes the arrow after completion, indefinitely.

III. Application & Implementation

In this section we describe a non-trivial application which we use to evaluate use of Arrows in practice. The application is a reproduction of a real-world inventory management system. We focused on the display of the inventory in the form of a paginated grid which can be filtered by a search term.

Server-side pagination was a strong requirement, as the amount of data in a live application would be too large to transmit to the client all at once – and most of the data would be wasted at client-side in a large number of cases where the user only cares about a handful of results.

Filtering by a search term is implemented as another query to the remote server. To increase responsiveness on the client-side, we opted to implement filtering as a *typeahead*. Unfortunately, this increases the chatter between the client and the server when a new request is fired on each keystroke.

To decrease chatter between the client and the server, we implemented a cache for AJAX requests which are live for a configurable period of time. Applications with fewer data modifications and applications that can stand to serve users with partially stale data can increase the expiry of cached records.

To decrease the user-perceived latency in *paging-forward*, we chose to *prefetch* the next page of results into the AJAX cache after displaying the current page of the results. Often, the next page it will already be in the cache when the user requests the next page. As long as the user is on the current page of results for the amount of time that it takes to prefetch a page, loading will always appear instantaneous. However, the prefetch should not interfere with the ability of the user to control the interface, so the prefetch should be canceled when the user interacts with a component which changes the set of results being displayed.

To stop superfluous requests to the remote server when filtering, we cancel ongoing AJAX requests when the user begins to modify the search term. To further reduce the number of requests which will be subsequently canceled, we request the initial page for a search term 400 milliseconds after the last keystroke occurs.

A. Callback Client Implementation

An example implementation of result set filtering and paging using only callbacks is given in Figure 5. The optimizations described above disproportionally increase the complexity of the code. The control flow is convoluted and does not follow a linear sequence of events.

```
1  // Filtering
2  var t = null;
3  $('#filter').on('keyup', function() {
4    // Cancel previous timer, if one exists
5    if (t != null) { clearTimeout(t); t = null; }
6
7    // Send AJAX request after 400ms delay
8    t = window.setTimeout(function() {
9      sendRequest($('#filter').val(), 1, handle)
10   }, 400);
11 });
12
13 // Paging
14 function handle(resp) {
15   display(resp.results);
16
17   // Load next page after clicking next
18   $('#next').one('click', function() {
19     sendRequest(resp.query, resp.next, handle);
20   });
21 }
22
23 function sendRequest(query, page, callback) {
24   try {
25     // Lookup either returns results or throws the key
26     callback(lookup([query, page]));
27   } catch(key) {
28     $.ajax({
29       url: buildUrl(query, page),
30       success: function(response) {
31          var data = JSON.parse(response);
32          store(key, data); // Puts (key -> data) in cache
33          callback(data);   // Continue computation
34       }
35     });
36   }
37 }
```

Fig. 5: An example for filtering with delay and paging with cache (*but no prefetch*). Lines 1-21 implement result set filtering by sending a query after 400ms of delay. *sendRequest* checks for a value in a cache and sends an AJAX request on cache miss. The function also displays the result, stores it in the cache, and sets a callback for the *next* button of the result grid.

In the example, filtering is implemented as a callback that reacts to the *keyup* event on a textbox. To delay the invocation of the callback, we use a timer that goes off after 400ms. However, the timer is canceled if another *keyup* event occurs before time is up. The timer is stored in a global variable since there is no way for the callback to the textbox to get a reference to the timer otherwise. The filtering function calls the function *sendRequest*, which checks local cache using the

query text and a page number as the key sends an AJAX request to the server if the key is not found.

As this example demonstrates, one of the main problems with web applications is that the control flow logic is often obscured by callbacks and side effects. In this example, caching logic is invoked by the AJAX callback, filtering delay adds a callback layer, while the timer cancellation depends on the side effect of a global variable. Together, the sequencing code becomes tightly coupled with the code that provides basic functionality, reducing modularity and decreasing readability and maintainability.

Furthermore, this implementation hides a subtle bug. Once the user types in some text and a page of results is displayed, a click event is registered on line 42. If the user types some more, another page of results is displayed and an additional click event is registered on the same line. At this point, two active handlers will race to load the second page of two different result sets when only one page is expected to be fetched. This can be fixed with the addition of another global definition which cancels a previously registered callback when a new one is registered.

Alternatively, we can use third-party libraries such as *select2* or *paramQuery*, which provide optimized solutions for common operations such as paging. While it is easy to learn and integrate these libraries, it is difficult to customize or extend their features. For example, these libraries have no built-in support for data caching, delayed typeahead, or AJAX cancellation. Other than editing the library directly, which makes it unsafe for upgrade, it is nearly impossible to implement extension points which invoke a method upon a particular action or event.

Another method of implementation is to use Promises, which can chain together asynchronous computation without explicit callbacks. However, Promise library only supports acyclic control flow so that recursive operations such as the delayed filtering cannot be directly encoded. Moreover, dataflow between chained functions is indirect and it is difficult to detect source of errors if the inputs received from previous function in the chain are used incorrectly.

B. Client Implementation using Arrows

Our Arrows library provides superior solution. Users can construct asynchronous and recursive computation using arrow constructors and combinators, which is more understandable and more modular. Since the composition of an arrow is separated from its execution, we can apply static analysis to detect errors early.

Figure 6a and Figure 6b list the **complete** set of problem-specific arrows defined for this implementation. We have discarded type annotations, discussed further in Section IV, attached to the functions for brevity.

Nine arrows were declared, but only `ajax` and `handle` are problem-specific. The `lookup` and `store` arrows are very general and work without modification in many applications. The extraction arrows in Figure 6b exist only to make routing of data more convenient and do not reduce any complexity inherent in the problem.

Figure 6c presents the remaining implementation, which builds the main arrows, `filtering` and `paging`, and executes the former. We describe each arrow, in turn, in the following.

The `filtering` arrow is a recursive arrow constructed from `fix`. First, the arrow blocks until a *keyup* event occurs on an element matching the *#filter* selector. Then, the arrow creates a tuple containing the value of *#filter* and the constant value 1, which is used later as the 'initial' page. This value is fed into both the `paging` arrow (beginning after a 400ms delay) and the `filtering` arrow (referenced recursively as a) which are run in parallel. The first arrow does not finish, as the execution of `paging` is recursive. However, if a makes progress (resumes after blocking), then the first arrow is canceled and the execution of this arrow effectively restarts from the beginning.

The `paging` arrow receives the target query term and page number as input. First, the current page is fetched from a remote server or a cache by `cachedAjax`. Then, a tuple of the form $((q,\ p),\ (q,\ n),\ \_)$ is constructed by the `fanout` arrow, where q is the query term, p is the previous page number, n is the next page number, and _ is the result of display, which we discard. The construction of the last value also executes the arrow `handle` as a side-effect, which modifies the DOM to display the current page of results.

Then, the execution of the arrow blocks until a *click* event occurs on either the *#prev* element or the *#next* element. Clicking the *#prev* element will extract the $(q,\ p)$ term from the tuple above and feed that back into a recursive call of the arrow. Clicking the *#next* element will pass $(q,\ n)$ instead. We do not use the `on` combinator here, as seen in other parts of the composition, as we care only that the event *did* fire, but not the event itself. We use the `EventArrow` directly to pause computation until the user directs the arrow to continue execution.

The `runFirstIfPossible` construction will attempt to *prefetch* the next page of results from the remote server (if it does not exist in the cache), but will abandon the request if the user selects a pagination control before it can complete.

As our solution demonstrated, the control-flow of arrows is declared **directly** by the arrow composition. With callbacks, the program-specific logic is mixed with sequencing and event registration so that its control flow is difficult to determine syntactically. Arrows are also reusable. The functions `makeCached` and `runFirstIfPossible` can be applied to other arrows for similar purpose. The arrow implementation is also more modular. In our example, the *paging* arrow could be used without *filtering*, or with a different outer layer.

Note that the code in Figure 5 is shorter than the code in Figure 6 because the latter also supports paging in the reverse direction, implements optimistic prefetching, and disables buttons when pending results are being loaded. Moreover, Figure 5 omits code related to cache lookup and expiration, which is explicit in Figure 6a.

```
var cache = {};
let lookup = new LiftedArrow(key => {
  if (key in cache && !isExpired(cache[key].time)
    return cache[key].value;
  throw key;
});

let store = new LiftedArrow((key, value) => {
  cache[key] = {'value': value, 'time': Date.now()};
});

let ajax = new AjaxArrow((query, page) => {
  'url': '/search?q=${query}&page=${page}'
});
```

(a) Caching and and Ajax definitions.

```
let extractPrev = new LiftedArrow(x => x.prev);
let extractNext = new LiftedArrow(x => x.next);
let extractQuery = new LiftedArrow(x => x.query);
let extractResults = new LiftedArrow(x => x.results);
let getVal = new LiftedArrow((elem, ev) => $(elem).val());

let handle = new LiftedArrow(x => {
  $('#results')
    .empty()
    .append(x.map(t => $('<li />').text(t)));
});
```

(b) Data-specific function definitions.

```
// Fetch a value (by key) from cache, or populate cache with value from arr
let makeCached = arr => lookup.catch(arr.carry().seq(store.remember()).nth(2));

// Replace arr in composition above with call to remote server
let cachedAjax = makeCached(ajax);

// Build an arrow that runs `main`, but also attempts to execute `task` first if main
// is too 'slow' to run. The `task` will be canceled if `main` makes progress.
let runFirstIfPossible = (task, main) => main.any(task.noemit().remember().seq(main));

function disableWhileLoading(a) {
    let enable  = new LiftedArrow(() => $('.paging-control').removeClass('disabled'));
    let disable = new LiftedArrow(() => $('.paging-control').addClass('disabled'));

    // Disable the paging buttons while the arrow `a` is running
    return Arrow.seq([disable.remember(), a, enable.remember()]);
}

let paging = Arrow.fix(a => Arrow.seq([
    // Fetch current page of data
    disableWhileLoading(cachedAjax),

    // Handle current page's data and extract prev/next pagination cursors
    Arrow.fanout([
        Arrow.fanout([extractQuery, extractPrev]),
        Arrow.fanout([extractQuery, extractNext]),
        extractResults.seq(handle),
    ]),

    runFirstIfPossible(
        // Attempt to prefetch next page of results, but cancel if the next arrow
        // makes progress before we get a result.
        Arrow.seq([new NthArrow(2), cachedAjax]),

        // Wait until the user clicks a pagination button. Each click will end up
        // passing a different pagination cursor recursively back into the arrow.
        Arrow.any([
            new ElemArrow('#prev').seq(new EventArrow('click')).remember().nth(1),
            new ElemArrow('#next').seq(new EventArrow('click')).remember().nth(2),
        ])
    ),

    a,
]));

let filtering = Arrow.fix(a => Arrow.seq([
    new ElemArrow('#filter'),
    Arrow.seq([
        // Get the current filter value and the first page (value 1)
        Arrow.fanout([getVal, new NumericArrow(1)]),

        // Page the current results until the user changes the filter
        Arrow.any([new Delay(400).seq(paging).noemit(), a])
    ]).on('keyup')
]));

filtering.run();
```

(c) Complete composition of arrows.

Fig. 6: The relevant source of the application - omitted code is trivial.

C. Performance Comparison

We measured non-idle performance of both the implementation using Arrows and using only callbacks. Averaged over 100 clicks of pagination controls (30 of which triggered a cache hit, 70 of which triggered a cache miss), the number of requests and time spent in-code was nearly identical. We find that using Arrows in this application added no measurable runtime overhead.

IV. Type System

The Arrows library comes with a *pluggable type system* which infers the type of an arrow during its composition. If an illegal composition is detected, it throws a type error. This, in many cases, can detect illegal compositions, which *seem* correct at first glance but have subtle problems at runtime. Often times, an illegal composition fails at an unexpected source location, making it difficult to track the true source of the bug.

In this type system, each function lifted into an arrow annotates the type it expects as input, the type it produces as output, a set of subtyping constraints, and a set of types of the exception values that can be thrown. For example, the type of the `lookup` arrow defined in Figure 6a is annotated as $\forall\alpha, \beta.\ \alpha \rightsquigarrow \beta \setminus (\emptyset,\ \{\alpha\})$ where the input of the arrow is a type variable α and its output is a type variable β. The pair $(\emptyset,\ \{\alpha\})$ represents an empty constraint set and the singleton exception type α.

Only the function definition of a lifted arrow requires a type annotation. The type of a composite arrow can be inferred from its components. For example, the inferred type of `cachedAjax`, defined in Figure 6c, is

$$(\mathit{String},\ \mathit{Number}) \rightsquigarrow \{\mathit{prev} : \mathit{Number},\ \mathit{next} : \mathit{Number},\ \mathit{query} : \mathit{String},\ \mathit{results} : [\tau]\} \setminus (\emptyset,\ \{\mathit{AjaxError}\})$$

where τ is the type of the value returned by the remote server, which should be annotated in the `ajax` arrow. Notice that all type variables have been unified with concrete values and that a value of type $\mathit{AjaxError}$ can now be thrown.

Inferring the type of combinators does come with a cost. The time to *bootstrap* between the two client implementations discussed above differs slightly. The callback-only version was ready to accept the first user event within 172ms, where the Arrows version required 207ms. However, we find this app to be of non-trivial size and a one-time cost of 35ms at startup, even in a production environment, is a negligible.

Very large applications or applications which require a very responsive startup may find this additional cost unacceptable. In these cases, type inference can be enabled only during development to detect type errors and be disabled in the deployed version.

V. Related Work

Arrows [5] are a generalization of monads [6], which enable functional composition of programs to support side effects such as exception handling, concurrency, and continuation without sacrificing referential transparency.

Our work is inspired by Arrowlets [4], a JavaScript library that combines arrows and continuation-passing style to support event-driven web development. However, it is difficult to locate the source of errors in Arrowlet programs due to the indirection of arrow constructors and combinators. Our arrow library overcame this weakness by adding a type inference component to check typing errors at arrow composition time. We also support a more general set of arrow constructors and combinators and include error-handling capability that subsumes the semantics of ES6 Promises.

Other than *control-flow* oriented solutions like Arrows, there are also *data-flow* oriented solutions such as Functional Reactive Programming (FRP) [7]. Notable JavaScript applications of FRP include Elm [8] and Flapjax [9]. Other related systems include Promises, which we explored earlier, and Factors [10], which defines *factor* abstraction, a state of a program, that can be *queried* either synchronously or asynchronously.

VI. Conclusion

In this paper, we demonstrated how Arrows could be easily integrated into a real-world web application. The flexibility it provides in terms of code maintenance and readability could be a big factor for companies that are interested in creating large scale web applications. Additionally, the optional type checking functionality protects against error-prone compositions. The ease of adding additional features make Arrows an ideal candidate for enterprise applications.

For future work, we plan on expanding the set of applications which uses Arrows. This will allow a richer set of data to use for benchmarks (overhead of arrows and their type inference on startup and runtime) and usability studies (readability of solutions, discoverability of the API, benefit of discovering possible runtime errors instead as typing errors).

References

[1] E. Fritz, T. Zhao, Type inference of asynchronous arrows in JavaScript, Reactive and Event-based Languages & Systems (2015).

[2] G. J. Sussman, G. L. Steele Jr, Scheme: A interpreter for extended lambda calculus, Higher-Order and Symbolic Computation 11 (1998) 405–439.

[3] K. Gallaba, A. Mesbah, I. Beschastnikh, Don't call us, we'll call you: Characterizing callbacks in javascript, in: Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on, IEEE, 2015, pp. 1–10.

[4] Y. P. Khoo, M. Hicks, J. S. Foster, V. Sazawal, Directing JavaScript with arrows, in: Proceedings of the 5th Symposium on Dynamic Languages, DLS '09, ACM, New York, NY, USA, 2009, pp. 49–58.

[5] J. Hughes, Generalising monads to arrows, Science of Computer Programming 37 (1998) 67–111.

[6] P. Wadler, The essence of functional programming, in: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1992, pp. 1–14.

[7] Z. Wan, P. Hudak, Functional reactive programming from first principles, in: ACM SIGPLAN Notices, Vol. 35, ACM, 2000, pp. 242–252.

[8] E. Czaplicki, S. Chong, Asynchronous functional reactive programming for guis, in: ACM SIGPLAN Notices, Vol. 48, ACM, 2013, pp. 411–422.

[9] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, S. Krishnamurthi, Flapjax: a programming language for ajax applications, in: ACM SIGPLAN Notices, Vol. 44, ACM, 2009, pp. 1–20.

[10] S. K. Muller, W. A. Duff, U. A. Acar, Practical abstractions for concurrent interactive programming, Tech. rep., Carnegie Mellon University (2015).