

Directing JavaScript with Arrows (Functional Pearl)

Khoo Yit Phang Michael Hicks Jeffrey S. Foster Vibha Sazawal

University of Maryland, College Park
{khooyp,mwh,jfoster,vibha}@cs.umd.edu

Abstract

JavaScript, being a single-threaded language, makes extensive use of event-driven programming to enable responsive web applications. However, standard approaches to sequencing events are messy, and often lead to code that is difficult to understand and maintain. We have found that *arrows*, a generalization of *monads*, are an elegant solution to this problem. Arrows allow us to easily write asynchronous programs in small, modular units of code, and flexibly compose them in many different ways, while nicely abstracting the details of asynchronous program composition. In particular, we show how to use arrows to construct a variety of state machines, such as autoscrollers and drag-and-drop handlers.

1. Introduction

JavaScript is the *lingua franca* of Web 2.0. Web applications such as Google Maps and Flickr rely on it, and JavaScript libraries power the features of popular websites such as web portals (yahoo.com), retail sites (target.com), social networking applications (linkedin.com), and public forums (slashdot.org). Because JavaScript code runs in a client-side browser, applications can present a rich, responsive interface without unnecessary delays due to server communication.

Most JavaScript programs are written in an event-driven style, in which programs register callback functions that are triggered on events such as timeouts or mouse clicks. A single-threaded event loop dispatches the appropriate callback when an event occurs, and control returns back to the loop when the callback completes.

To keep web applications responsive, it is crucial that callbacks execute quickly so that new events are handled soon after they occur. Thus to implement non-trivial features like long-running loops (e.g., for animations) or state machines (e.g., to implement drag-and-drop), programmers must chain callbacks together—each callback ends by registering one or more additional callbacks. For example, each iteration of a loop would end by registering the current callback with a (short) timeout. Unfortunately, this style of event-driven programming is tedious, error-prone, and hampers reuse. The callback sequencing code is strewn throughout the program, and very often each callback must hard-code the names of the next events and callbacks in the chain.

This paper presents what we believe to be a very flexible and elegant solution to composing callbacks: a new JavaScript library which we call *Arrowlets*, which is based on *arrows*. Arrows

(Hughes 2000) are a programming pattern closely related to monads, used extensively in Haskell. An arrow abstraction resembles a normal function, with the key feature that an arrow can be composed in various ways to create new arrows. The core composition operator is sequencing: if f and g are arrows then $f \ggg g$ is an arrow that first runs f and then runs g .

In Arrowlets, arrows are based on functions written in *continuation passing style* (CPS). A CPS function takes a normal argument x and a continuation k , and completes by calling k with the result produced by the function. We provide *event arrows*, built around a CPS function that registers its continuation argument with a particular event. Users write *handler functions* that process particular events and lift them to arrows. An event arrow can then be composed with a handler arrow to create a single arrow that registers an event and handles it when the event occurs. We also provide looping and either-or composition operators, and include support for cancellation.

With our library, the code for handling events is clearly separated from the “plumbing” code required to chain event handlers together. This makes code easier to understand, change, and reuse: the flow of control is clearly evident in the composition of handlers, and the same handlers can be chained in different ways and set to respond to different events.

We begin by illustrating the standard approach for event-based programming in JavaScript, along with its difficulties. Then we introduce our basic approach to allaying these difficulties using JavaScript arrows. We next scale up to include richer combinators and present several examples. We conclude by comparing our work to related approaches, including other JavaScript frameworks that also improve event handling. We believe that our library provides JavaScript programmers a flexible, modular, and elegant way to structure their programs. The Arrowlets library, as well as several live examples, is freely available at <http://www.cs.umd.edu/projects/PL/arrowlets>.

2. Event Programming in JavaScript

In modern web browsers, JavaScript is implemented as a single-threaded programming language.¹ This is a real problem for web developers, because the browser’s JavaScript interpreter typically runs in the main UI thread. Thus a long-running script could stall a web page (and browser), making it appear unresponsive.

To solve this problem, JavaScript programs make heavy use of *event-driven programming*, in which programs register asynchronous callbacks to handle UI interactions and break up long-running tasks. A common idiom is to register a callback that handle events on a particular document element indefinitely. For example, after running the following program, `clickTarget` will be

Technical Report CS-TR-4923, Department of Computer Science, University of Maryland, College Park, August 2008.

¹ For brevity, the code described in this paper do not run in Internet Explorer due to minor API differences. The examples have been verified to work in Safari and Firefox.

called each time the HTML element named `target` is clicked with the mouse²:

```
function clickTarget (evt) {
    evt.currentTarget.textContent = "You clicked me!";
}
document.getElementById("target")
    .addEventListener("click", clickTarget, false);
```

Events are also used to slice long-running loops into small tasks that quickly return control to the UI. For example, the following program scrolls a document one pixel at a time until the document can be scrolled no further. The call to `setTimeout` schedules a call to `scrollDown(el)` to occur *0ms* in the future:

```
function scrollDown(el) {
    var last = el.scrollTop++;
    if (last !== el.scrollTop)
        setTimeout(scrollDown, 0, el);
}
scrollDown(document.body);
```

We can think of `scrollDown` as a state machine. In the initial state, it tries to scroll one pixel, and then either transitions to the same state (if the scroll succeeded) or to an accepting state (if scrolling is complete).

2.1 Chaining Callbacks is Ugly

The `scrollDown` function implements a very simple state machine with only one handler, and as such is easy to write. Chaining handlers together to implement more complicated state machines can be more difficult. For example, suppose we want to scroll a web page down and then up. We can accomplish this as follows:

```
1 function scrollUp (el) {
2     var last = el.scrollTop--;
3     if (last !== el.scrollTop)
4         setTimeout(scrollUp, 0, el);
5 }
6 function scrollDown (el, callback) {
7     var last = el.scrollTop++;
8     if (last !== el.scrollTop)
9         setTimeout(scrollDown, 0, el, callback);
10    else if (callback)
11        callback (el);
12 }
13 scrollDown(document.body, scrollUp);
```

Line 13 calls `scrollDown` to begin scrolling the web page downward. We pass in `scrollUp` as a callback, which is invoked after the downward scroll is finished (line 11). Then `scrollUp` runs until completion, and exits.

That was not too hard, but notice we had to specialize `scrollDown` to take a callback that itself takes `el` as an argument. This is rather inflexible—if we wanted to chain a third callback into the sequence, we would need to change `scrollDown` and `scrollUp`, and extend their parameter lists with more callbacks.

The challenging issue with functions using callbacks is that they “return” by invoking the callback, rather than returning to their caller. Functional programmers will recognize this style of programming as *continuation-passing style* (CPS) (Appel 1992). We can indeed overcome the limitations of the approach above by carefully converting the entire program into CPS:

```
1 function scrollUp (callback) {
2     return function scrollUpClosure (el) {
3         var last = el.scrollTop--;
```

```
4         if (last !== el.scrollTop)
5             setTimeout(scrollUpClosure, 0, el);
6         else if (callback)
7             callback (el);
8     }
9 }
10 function scrollDown (callback) {
11     return function scrollDownClosure (el) {
12         var last = el.scrollTop++;
13         if (last !== el.scrollTop)
14             setTimeout(scrollDownClosure, 0, el);
15         else if (callback)
16             callback (el);
17     }
18 }
19 scrollDown (scrollUp (scrollDown (scrollUp ()))) (document.body);
```

Now `scrollUp (callback)` returns a closure that either calls itself or `callback`, and similarly for `scrollDown (callback)`. This is very promising—we can now drive our web visitors crazy by forming long compositions of these functions, like the down/up/down/up scrolling example on line 19.

However, while CPS is a good approach, manually CPS-converting JavaScript code is not fun, especially considering that callback “chains” can be more complex than simple linear arrangements. A prototypical example is support for *drag-and-drop*.

2.2 Drag-and-Drop: A Pointed Problem

Consider the problem of supporting drag-and-drop in a web browser. Figure 1(a) gives a state machine showing the sequencing of event handlers we need for this feature. We begin with a `mousedown` event on an item of interest and transition to the `setup` state. From there, we cancel drag-and-drop if the user releases the mouse (`mouseup`), or start the drag for real on a `mousemove`. The user can keep dragging as much as they like, and we drop when the mouse button is released. In each of these states, we need to do various things, e.g., when we (re-)enter the `drag` state, we animate the motion of the selected object.

The standard approach to implementing this state machine is shown in Figure 1(b). Each function corresponds to one state, and mixes together “plumbing” code to install and uninstall the appropriate event handlers and “action” code to implement the state’s behavior. For example, we install the `setup` function to handle the `mousedown` event on line 27. When called, `setup` uninstalls itself and adds handlers to transition to the `drag` and `cancel` states (lines 3–5), and then carries out appropriate actions (line 6).

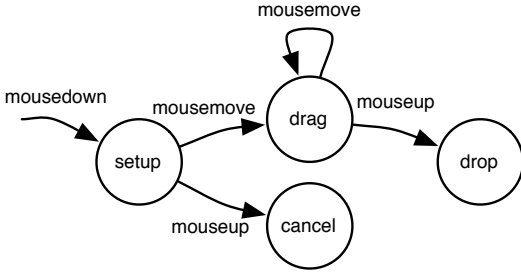
This code is not that easy to understand—the flow of control is particularly convoluted, because each event handler ends by returning, and control “continues” with the next event handler indirectly. Code reuse is also hard, because each event handler needs to know exactly where it is in the state machine, to adjust the set of registered events appropriately. For example, if we wanted to initiate drag-and-drop with a `mouseover` event, we would need to make a new copy of `setup`.

There are existing JavaScript libraries that make standard widgets like drag-and-drop quite easy to use. However, they are quite inflexible in our experience, and even small customizations would require modifying complex library internals. In contrast, arrows give us a much more flexible approach to building these kinds of interface elements. More discussion appears in Section 5.

3. Arrows Point the Way

While event-based compositions in JavaScript seem messy and awkward, there is no need to despair—we have found that by bringing *arrows* (Hughes 2000) into JavaScript, we can start writing event-based code that is clean, understandable, and reusable.

²The last parameter to `addEventListener`, required by Firefox, selects the order of event handling, and can be ignored for this and all other examples in the paper.



(a) State machine

```

1 function setup(event) {
2   var target = event.currentTarget;
3   target.removeEventListener("mousedown", setup, false);
4   target.addEventListener("mousemove", drag, false);
5   target.addEventListener("mouseup", cancel, false);
6   /* setup drag-and-drop */
7 }
8 function drag(event) {
9   var target = event.currentTarget;
10  target.removeEventListener("mouseup", cancel, false);
11  target.addEventListener("mouseup", drop, false);
12  /* perform dragging */
13 }
14 function drop(event) {
15  var target = event.currentTarget;
16  target.removeEventListener("mousemove", drag, false);
17  target.removeEventListener("mouseup", drop, false);
18  /* perform dropping */
19 }
20 function cancel(event) {
21  var target = event.currentTarget;
22  target.removeEventListener("mousemove", drag, false);
23  target.removeEventListener("mouseup", cancel, false);
24  /* cancel drag-and-drop */
25 }
26 document.getElementById("dragtarget")
27   .addEventListener("mousedown", setup, false);

```

(b) Standard JavaScript implementation

Figure 1: Drag-and-drop in JavaScript

Arrows are generalizations of monads, and like them, they help improve *modularity*, by separating composition strategies from the actual computations; they are *flexible*, because operations can be composed in many different ways; and they help *isolate* different program concerns from one another (Wadler 1992).

3.1 Function Arrows

Figure 2(a) gives a (very) simplified definition of the *Arrow* type class in Haskell. A type a is an instance of *Arrow* (written *Arrow a*) if it supports at least two operations³: $arr\ f$, which lifts function f into a , and $f \ggg g$, which produces a new arrow in which g is applied to the output of f . The simplest arrow instance, *Arrow* (\rightarrow), represent standard functions where arr is the identity function, and \ggg is function composition. With these operations, we can compose functions using arrow operations:

```

add1 x = x + 1
add2 = add1 >>> add1

result = add2 1    {- returns 3 -}

```

³ Arrows in Haskell also support a third primitive operation, *first*; we will defer the JavaScript implementation of *first* to future work.

```

class Arrow a where
  arr :: (b -> c) -> a b c
  (>>>) :: a b c -> a c d -> a b d

instance Arrow (->) where
  arr f = f
  (f >>> g) x = g (f x)

```

(a) Arrows in Haskell

```

Function.prototype.A = function() { /* arr */
  return this;
}
Function.prototype.next = function(g) { /* >>> */
  var f = this; g = g.A(); /* ensure g is a function */
  return function(x) { return g(f(x)); }
}

```

(b) Function arrows in JavaScript

Figure 2: Two definitions of arrows

Figure 2(b) shows function arrows in JavaScript. In JavaScript, every object has a *prototype* object (analogous to a class). Properties (e.g., methods) of an object are first looked up in the object itself, and if not found, are looked up in the object's prototype. JavaScript functions are themselves objects, and hence they have a prototype, named `Function.prototype`.

In Figure 2(b), we add two methods to every function object. The `A` method, corresponding to arr , lifts a function into an arrow. As before, the `A` method is just the identity. The `next` method, corresponding to \ggg , composes two arrows by returning a new (anonymous) function invoking the composition. In this code, binding `f` to `this` sets `f` to refer to the object (i.e., function) whose `next` method is invoked. We also lift argument `g` to an arrow with the call `g.A()`. This check helps ensure that `g` is a function (all of which have the `A` method).

With these simple definitions, we can now compose functions as arrows in JavaScript in the same way as Haskell:

```

function add1(x) { return x + 1; }
var add2 = add1.next(add1);

var result = add2(1);    /* returns 3 */

```

Notice that we did not need to apply the `A` method to `add1`, because we have added the `next` method to all functions, thus implicitly making them arrows.

JavaScript lacks Haskell's sophisticated type system, so we unfortunately cannot statically ensure arrows are used correctly. For example, the definition in Figure 2(b) only works for single argument functions. Moreover, we may have different kinds of arrows that cannot be simply mixed together. Thus in practice (and in the examples below), we will introduce new prototypes to distinguish different kinds of arrows from each other.

3.2 CPS Function Arrows

Regular function composition will not work for event handling arrows because event handlers are invoked asynchronously. Following the observation made at the end of Section 2.1, we can solve this problem using CPS functions. In particular, an event arrow can be viewed as a kind of CPS function that registers its continuation to handle some event. This arrow can then be composed with a handler arrow that actually processes the event (and then invokes its own continuation).

```

1  function CpsA(cps) {
2      this.cps = cps; /* cps :: (x, k) → () */
3  }
4  CpsA.prototype.CpsA = function() { /* identity */
5      return this;
6  }
7  CpsA.prototype.next = function(g) {
8      var f = this; g = g.CpsA();
9      /* CPS function composition */
10     return new CpsA(function(x, k) {
11         f.cps(x, function(y) {
12             g.cps(y, k);
13         });
14     });
15 }
16 CpsA.prototype.run = function(x) {
17     this.cps(x, function(y) { });
18 }
19 Function.prototype.CpsA = function() { /* lifting */
20     var f = this;
21     /* wrap f in CPS function */
22     return new CpsA(function(x, k) {
23         k(f(x));
24     });
25 }

```

Figure 3: CPS function arrows in JavaScript

The core building block of this approach is a definition of arrows for CPS functions, shown in Figure 3. A CPS function takes two arguments: the “normal” argument x and a continuation k , called with the function’s final result. In the figure, `CpsA` (lines 1–3) constructs a CPS arrow from a `cps` function. In JavaScript, constructors are simply regular functions, and when we invoke `new CpsA(cps)`, JavaScript creates a new object and then initializes it by calling `CpsA(cps)` with `this` bound to the new object.

On line 4 we introduce our convention of giving `CpsA` objects an identity `CpsA` method, which we use like the `A` method from Figure 2(b): if we successfully invoke $x = x.CpsA()$, we know x is a CPS arrow. The next method to compose CPS functions behaves as expected⁴, invoking `f`, and passing in a continuation that invokes `g`, which itself continues with `k` (lines 7–15). We call a CPS arrow by invoking its `run` method (lines 16–18), which simply calls the function in the `cps` field. We pass `cps` the actual argument and an empty continuation that does nothing. Note we did not need a `run` method in Figure 2(b) because regular function arrows can be executed simply by invoking the function itself. Finally, we extend `Function`’s prototype with a `CpsA` method to lift a normal one-argument function into a CPS arrow. With this method, programmers using the library can write regular functions, and the details of CPS are effectively hidden.

With these definitions, we can convert our `add1` and `add2` functions to CPS and compose them:

```

function add1(x) { return x + 1; }
var add2 = add1.CpsA().next(add1.CpsA());

var result = add2.run(1); /* returns 3 */

/* where: add1.CpsA().cps = function(x,k) { k(add1(x)); }
          add1.CpsA().next(add1.CpsA()).cps
          = function(x,k) { k(add1(add1(x))); } */

```

```

1  function SimpleEventA(eventname) {
2      if (!(this instanceof SimpleEventA))
3          return new SimpleEventA(eventname);
4      this.eventname = eventname;
5  }
6  SimpleEventA.prototype = new CpsA(function(target, k) {
7      var f = this;
8      function handler(event) {
9          target.removeEventListener(
10             f.eventname, handler, false);
11             k(event);
12         }
13         target.addEventListener(f.eventname, handler, false);
14     });

```

Figure 4: `SimpleEventA` for handling JavaScript listeners

3.3 Simple Asynchronous Event Arrows

Building on CPS arrows we can now define simple event arrows, which are CPS functions that register their continuations to handle particular events. Ultimately we will want several forms of composition, but for now, we define an event arrow `SimpleEventA` that supports linear sequencing, shown in Figure 4.

In this code, the function `SimpleEventA` acts as a constructor, where line 3 implements a convenient JavaScript idiom. If the constructor is called as a regular function (i.e., without `new`), it calls itself again as a constructor to create a new `SimpleEventA` object. This allows us to omit `new` when using `SimpleEventA`. Line 4 stores the name of the event this arrow handles.

Lines 6–14 define the `SimpleEventA` prototype object to be a `CpsA` arrow constructed from an anonymous function. By making the prototype a `CpsA` object, `SimpleEventA` inherits all the properties of `CpsA`. The anonymous function installs the local function handler to be triggered on `eventname` (line 13). When this event fires, handler deregisters itself from handling that event, and then invokes the continuation `k` with the received event. We chose to immediately deregister event handlers that have fired since this corresponds to transitions in a state machine, according to our motivating use case.

Although JavaScript lacks types, we can view `SimpleEventA` as having arrow type *CpsA target event* (analogous to function type *target → event*). To use such an arrow, we typically compose it with a handler arrow of type *CpsA event target* (analogous to type *event → target*), where the output is the target for the next event.

Examples Let us rewrite the very first example in Section 2 to use our simple event arrows. First we write a handler arrow for the event:

```

var count = 0;
function clickTargetA(event) {
    var target = event.currentTarget;
    target.textContent = "You clicked me! " + ++count;
    return target;
}

```

This function extracts the target of the event, updates its text, and then returns it (for the next event handler). To register this code to handle a single click, we write the following plumbing code:

```

SimpleEventA("click").next(clickTargetA)
    .run(document.getElementById("target"));

```

This code creates an event arrow for a click event (on the target element) and composes it with our handler arrow. When the event

⁴JavaScript is not tail-recursive, so this simple definition of `next` can cause the call stack to overflow. Our library implements CPS using trampolines to avoid this issue.

fires, `clickTargetA` is called with the event's target, and the event handler is removed. Also, in this code, the structure of event handling is quite apparent. And, because we have separated the plumbing from the actions, we can reuse the latter easily. For example, to count button clicks on another target, we just create another `SimpleEventA`, reusing the code for `clickTargetA` (the effect on count is shared by the two handlers):

```
SimpleEventA("click").next(clickTargetA)
  .run(document.getElementById("anotherTarget"));
```

If we want to track a sequence of events on the same target, we simply compose the handlers:

```
SimpleEventA("click").next(clickTargetA)
  .next( SimpleEventA("click").next(clickTargetA) )
  .run(document.getElementById("target"));
```

This code waits for one click, increments the count, and then waits again for a click, and increments the count once more. Sequential composition of asynchronous event arrows using `next` is associative, as expected, so we could equivalently write the above as

```
SimpleEventA("click").next(clickTargetA)
  .next(SimpleEventA("click"))
  .next(clickTargetA)
  .run(document.getElementById("target"));
```

Event arrows have another useful property in addition to easy composition: The details of different browser event handling libraries can be hidden inside of the arrow library, rather than being exposed to the programmer. For example, Internet Explorer uses `attachEvent` instead of `addEventListener`, and we could modify the code in Figure 4 to call the appropriate function depending on the browser.

4. Full Asynchronous Arrows

Now that we have developed simple event arrows, we can extend them with features for implementing more sophisticated examples, like drag-and-drop. First, we introduce *progress arrows* to track and/or abort the evaluation of an asynchronous event arrow. Second, we add a looping constructor, useful for a building a possibly-unbounded sequence of events. Third, we add “either-or” compositions of asynchronous arrows, in which a pair of handlers is installed, and once one fires, the other is cancelled. Put together, these features allow us to elegantly implement the drag-and-drop example from Section 2.2.

4.1 Tracking Progress with Asynchronous Arrows

The first step in adding progress arrows is to extend `CpsA` so that continuations take both the normal function argument `x` and a progress arrow argument `p`. A progress arrow (defined with the `ProgressA` constructor) can be used by other bits of code to observe or cancel the progress of asynchronous arrow, which is useful for arrows whose operation is long-running; we explain its operation in detail shortly. Our `CpsA` definition extended with progress arrows is called `AsyncA`, for *asynchronous arrow*, and is shown in lines 1–19 of Figure 5. The constructor (line 1) and lifting function (line 2) work analogously to `CpsA`, and `next` (lines 3–10) simply passes the extra parameter through the CPS composition. The `run` method now optionally takes a progress arrow argument `p`, or sets `p` to an empty progress arrow on line 12 if no argument is passed.⁵ Then `run` passes the arguments to `this.cps`, as before, and finally returns `p` back to the caller. This last step allows the caller of `run` to make use of the progress arrow later, in the case that it was created on

⁵ If the argument `p` is not given, then it is set to `undefined`, in which case `p || e` evaluates to `e`.

```
1 function AsyncA(cps) { this.cps = cps; }
2 AsyncA.prototype.AsyncA = function() { return this; }
3 AsyncA.prototype.next = function(g) {
4   var f = this; g = g.AsyncA();
5   return new AsyncA(function(x, p, k) {
6     f.cps(x, p, function(y, q) {
7       g.cps(y, q, k);
8     });
9   });
10 }
11 AsyncA.prototype.run = function(x, p) {
12   p = p || new ProgressA();
13   this.cps(x, p, function(y) {});
14   return p;
15 }
16 Function.prototype.AsyncA = function() {
17   var f = this;
18   return new AsyncA(function(x, p, k) { k(f(x), p); });
19 }
20
21 function ConstA(x) {
22   return (function() { return x; }).AsyncA();
23 }
```

Figure 5: Full asynchronous arrows

line 12 rather than passed in. Finally, the code to lift functions to `AsyncA` (lines 16–19) is the same as before. For convenience, we also introduce a function `ConstA(x)`, which produces an arrow that ignores its inputs and returns `x` (lines 21–23).

Next, we use `AsyncA` to implement an arrow constructor `EventA`, just as we used `CpsA` to implement `SimpleEventA`. The code is shown in Figure 6(a). The arrow constructor (lines 1–5) is as before. `EventA` inherits from `AsyncA` (line 6), also as before. When an event arrow is run, it registers (line 17) the cancel function (lines 8–11) with the progress arrow, and installs an event handler (line 18). This allows us to later abort the arrow (i.e., remove the event handler) if we wish. When an event is triggered, we inform the progress arrow by invoking its `advance` method (line 13). Upon receiving this method call, a progress arrow `p` will in turn alert any other objects that are listening for progress messages from `p`. For example, a progress bar object might ask to be informed each time an arrow composition advances, to update the image of the bar. The remainder of the code is as with `SimpleEventA`: we cancel the event handler (line 14) and call the continuation `k`, this time with both the event to process and the progress arrow (line 15).

To actually implement progress arrows, we could most likely extend regular function arrows (from Figure 2(b)), but since `AsyncA` is somewhat more flexible, we choose that as our starting place. Figure 6(b) defines `ProgressA`, our progress arrow type. Each progress arrow has two sets of listeners: cancellers (line 4), which are invoked when the arrow's `cancel` method is called, and observers (line 5), invoked via the arrow's `advance` method.

Users can add to the set of observers by invoking the `next` method inherited from `AsyncA`. On lines 7–9, we set the underlying CPS function of the arrow to push its argument onto the observer list. Thus, invoking `p.next(f).run()` for progress arrow `p` adds `f` to observers. Making `ProgressA` an asynchronous arrow gives it all the flexible compositional properties of arrows, e.g., it allows adding multiple, complex observers. For example, we could write `p.next(f).next(g)` for a progress arrow that invokes `g(f())` when progress occurs, or call `p.next(f).run()`; `p.next(g).run()` to add both observers `f` and `g`.

Cancellers are registered explicitly via the `addCanceller` method (lines 10–13). If `cancel` is invoked, the progress arrow calls all cancellers (lines 23–24). If `advance(c)` is invoked, the progress ar-

```

1 function EventA(eventname) {
2   if (!(this instanceof EventA))
3     return new EventA(eventname);
4   this.eventname = eventname;
5 }
6 EventA.prototype = new AsyncA(function(target, p, k) {
7   var f = this;
8   function cancel() {
9     target.removeEventListener(f.eventname,
10      handler, false);
11   }
12   function handler(event) {
13     p.advance(cancel);
14     cancel();
15     k(event, p);
16   }
17   p.addCanceller(cancel);
18   target.addEventListener(f.eventname, handler, false);
19 });

```

(a) Event arrows

```

1 function ProgressA() {
2   if (!(this instanceof ProgressA))
3     return new ProgressA();
4   this.cancellers = []; /* empty arrays */
5   this.observers = [];
6 }
7 ProgressA.prototype = new AsyncA(function(x, p, k) {
8   this.observers.push(function(y) { k(y, p); });
9 })
10 ProgressA.prototype.addCanceller = function(canceller) {
11   /* add canceller function */
12   this.cancellers.push(canceller);
13 }
14 ProgressA.prototype.advance = function(canceller) {
15   /* remove canceller function */
16   var index = this.cancellers.indexOf(canceller);
17   if (index >= 0) this.cancellers.splice(index, 1);
18   /* signal observers */
19   while (this.observers.length > 0)
20     this.observers.pop()();
21 }
22 ProgressA.prototype.cancel = function() {
23   while (this.cancellers.length > 0)
24     this.cancellers.pop()();
25 }

```

(b) Progress arrows

Figure 6: Full asynchronous arrows for simple events (cont'd)

row first removes *c* from *cancellers* (lines 16–17) and then calls any observers (lines 19–20). A call to *advance* implies that a unit of progress was made (e.g., an event triggered), and so the corresponding cancellation handler *c* for the unit of progress is removed as it is no longer needed. The corresponding observers are also removed as they are invoked. This behavior is analogous to the removal of event listeners after an event triggers.

Examples Even with the addition of progress arrows, we can still implement the two-click example from the prior section just as before:

```

var target = document.getElementById("target");
var p = EventA("click").next(clickTargetA)
    .next(EventA("click").next(clickTargetA))
    .run(target);

```

However, we can now use the returned progress arrow *p* to affect the arrow in flight, e.g., to stop waiting for clicks after 10 seconds:

```

setTimeout(function() {
  p.cancel();
  target.textContent = "Can't click this";
}, 10000);

```

We can also use it to track when an event begins:

```

var status = document.getElementById("status");
p.next(function() {
  status.textContent = "I've been clicked!";
}).run();

```

We can use progress arrows in combination with looping, described next, to build widgets like progress bars for long operations.

4.2 Looping with repeat()

To support examples like autoscrolling (and others that are more useful!) we need a composition operator for looping. Our first thought was to look to the Haskell implementation of arrows, which contains the *loop* fixpoint operator:

```

class Arrow arr => ArrowLoop arr where
  loop :: arr (a,c) (b,c) -> arr a b

```

```

instance ArrowLoop (→) where
  loop f a = b
  where (b,c) = f (a,c)

```

However, translating this declarative specification into JavaScript is a bit awkward, and the result could be hard to use, especially for programmers unfamiliar with fixpoint computation.

Instead, we introduce a more imperative style of looping with the *repeat(interval)* method in Figure 7(a). When run, the resulting arrow executes repeatedly, yielding to the UI thread via *setTimeout* recursion. Then we return a new asynchronous arrow containing the function *rep* (line 3). When invoked, *rep* calls *f* (the arrow from which the repeating arrow was created) and passes it a new, nested continuation with argument *y* and progress arrow *q*.

The argument *y* is a record created with either *Repeat* or *Done* (lines 20–21). These methods store their argument *x* in a JavaScript approximation of a tagged union—a record with the value field set to *x* and either the *Repeat* or *Done* field set to true.

Given argument *y*, there are two cases. If *y* is tagged with *Done* (lines 12–13), then we extract the value from *y* and pass it to the continuation *k* for the entire arrow. If *y* is tagged with *Repeat* (lines 5–11), we use the looping idiom from Section 2 to execute *rep* again after *interval* has elapsed. To allow the loop to be cancelled during this timeout period, we extend the list of cancellers to kill the timeout (lines 6 and 7), and since we progressed by one iteration, we advance the progress arrow (line 9).

We also provide *repeat()* on functions (lines 23–25).

Example *repeat*'s implementation is a bit tricky, but using it is easy. For example, we can now re-implement our auto-scroller example so that it scrolls up and down repeatedly until the document is clicked. Figure 7(b) shows the necessary JavaScript code. We bind *scrollUpA* and *scrollDownA* to arrows that scroll appropriately, using *repeat* so they scroll in one direction until done (lines 5 and 10). Notice that the bodies of *scrollUpA* and *scrollDownA* return *Repeat(e)* when the loop should continue (lines 3 and 8) and *Done(e)* when the loop should exit (lines 4 and 9).

We then build and run three asynchronous arrows. The first (lines 13–18) composes *scrollDownA* with *scrollUpA* (lines 15–16) and puts the composition into an infinite loop (line 17); notice that inserting *next(Repeat)* into the sequence wraps the output of *scrollUpA* in a *Repeat* record, ensuring we begin the down/up sequence again. We then initiate the down/up scrolling on the document (line 13), storing the returned progress arrow in *scrollingP*.

```

1 AsyncA.prototype.repeat = function() {
2   var f = this;
3   return new AsyncA(function rep(x, p, k) {
4     f.cps(x, p, function(y, q) {
5       if (y.Repeat) {
6         function cancel() { clearTimeout(tid); }
7         q.addCanceller(cancel);
8         var tid = setTimeout(function() {
9           q.advance(cancel);
10          rep(y.value, q, k);
11          }, 0);
12        } else if (y.Done)
13          k(y.value, q);
14        else
15          throw new TypeError("Repeat or Done?");
16      });
17    });
18  }
19
20 function Repeat(x) { return { Repeat:true, value:x }; }
21 function Done(x) { return { Done:true, value:x }; }
22
23 Function.prototype.repeat = function(interval) {
24   return this.AsyncA().repeat(interval);
25 }

```

(a) Loopy arrows

```

1 AsyncA.prototype.or = function(g) {
2   var f = this; g = g.AsyncA();
3   return new AsyncA(function(x, p, k) {
4     /* one progress for each branch */
5     var p1 = new ProgressA();
6     var p2 = new ProgressA();
7     /* if one advances, cancel the other */
8     p1.next(function() { p2.cancel();
9                          p2 = null; }).run();
10    p2.next(function() { p1.cancel();
11                        p1 = null; }).run();
12
13    function cancel() {
14      if (p1) p1.cancel();
15      if (p2) p2.cancel();
16    }
17    /* prepare callback */
18    function join(y, q) {
19      p.advance(cancel);
20      k(y, q);
21    }
22    /* and run both */
23    p.addCanceller(cancel);
24    f.cps(x, p1, join);
25    g.cps(x, p2, join);
26  });

```

(a) A tale of two arrows

```

1 var scrollUpA = function(el) {
2   var last = el.scrollTop - 1;
3   if (last !== el.scrollTop) return Repeat(el);
4   else return Done(el);
5 }.repeat();
6 var scrollDownA = function(el) {
7   var last = el.scrollTop + 1;
8   if (last !== el.scrollTop) return Repeat(el);
9   else return Done(el);
10 }.repeat();
11
12 /* Arrow 1: start scrolling down, up, down, up... */
13 var targetA = ConstA(document.body);
14 var scrollingP = (targetA
15   .next(scrollDownA)
16   .next(scrollUpA)
17   .next(Repeat)).repeat() /* infinitely */
18 .run();
19
20 /* Arrow 2: stop when clicked */
21 targetA
22   .next(EventA("click"))
23   .next(function() { scrollingP.cancel(); })
24   .run();
25
26 /* Arrow 3: indicate scroll % */
27 var status = document.getElementById("status");
28 (scrollingP
29   .next(function() {
30     status.textContent =
31       document.body.scrollTop
32       / document.body.scrollHeight + "%";
33   }).next(Repeat)).repeat()
34 .run()

```

(b) Scrolling is helpful, unless it is annoying

Figure 7: Looping with AsyncA

The second arrow (lines 21–24) waits for a click event on the document, and then cancels the scrolling operation using the

```

1 function WriteA(str) {
2   return function(event) {
3     var target = event.currentTarget;
4     target.textContent = str;
5     return target;
6   }
7 }
8
9 var heads = ConstA(document.getElementById("heads"));
10 var tails = ConstA(document.getElementById("tails"));
11
12 (heads.next(EventA("click")).next(WriteA("I win!")))
13 .or( tails .next(EventA("click")).next(WriteA("You lose!")))
14 .run();

```

(b) You'd never win

Figure 8: Branching with AsyncA

progress arrow. The third arrow (lines 27–34) updates a status box with the current scroll location as the document is being scrolled. We could have also implemented this scroll indicator as part of the first arrow; however, composing these as two arrows separates the concerns of performing and tracking the scrolling operation.

4.3 Either-or()

Our last addition to AsyncA is an “or” combinator that combines two asynchronous arrows and allows only one, whichever is triggered first, to execute. For example, this allows us to wait for a keystroke or a mouse movement, and respond to only one. An “or” combinator is necessary for supporting any branching state machine, as with drag-and-drop.

Looking again into Haskell’s quiver, *ArrowZero* and *ArrowPlus* seem relevant:

```

class Arrow a => ArrowZero a where
  zeroArrow :: a b c

```

```

class ArrowZero a => ArrowPlus a where
  (<|>) :: a b c -> a b c -> a b c

```

Semantically, *ArrowZero* arrows designate a special failing value that causes all remaining computation to be aborted, and *zeroArrow* is defined as an arrow that always fails. *ArrowPlus* complements *ArrowZero* by providing failure handling: $f \lt \! \! \! \Rightarrow g$ returns the output of f if it succeeds, or the output of g if f fails.

The combinator $\lt \! \! \! \Rightarrow$ is almost what we want, but there is one slight issue—events handlers in JavaScript never actually fail; they just wait indefinitely to be triggered. Thus, we introduce an *or* combinator that explicitly forces one of two asynchronous arrows to fail by canceling whichever one does not fire first.

Figure 8(a) gives the code for the *or* method, which combines the current arrow f with the argument g (line 2). Calling $f.or(g).run()$ executes whichever of f or g is triggered first, and cancels the other. To keep the presentation simpler, we assume both f and g are asynchronous event arrows. When invoked, the new arrow first creates progress arrows $p1$ and $p2$, which when advanced calls $cancel$ on the other arrow (lines 8–11). We also register (line 22) the $cancel$ function (lines 12–15), which will remove any handlers that are still installed. Then, we invoke the component arrows, f with $p1$ (line 23) and g with $p2$ (line 24). When either arrow completes, they call $join$ (lines 17–20), which first advances the progress arrow p for the composition itself (line 18) and then invokes the regular continuation.

Example In Figure 8(b), we demonstrate a simple coin-toss game implemented with *or*. We first define *WriteA* to create an arrow that writes into an event’s target element. Then, we compose two arrows (lines 12–13) that respond to clicks in *heads* and *tails*, respectively. Finally, we combine the arrows with *or*, ensuring that the player can only click once, on either *heads* or *tails*.

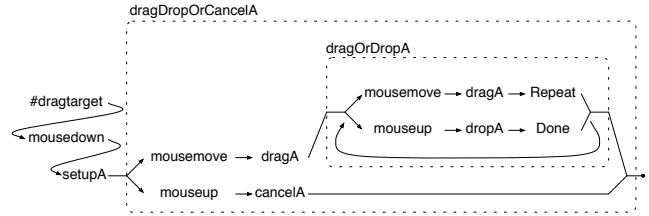
4.4 Arrow-based Drag-and-Drop

Now that we have done the hard work of building up a library of arrow combinators, we can implement the drag-and-drop example from Section 2.2 in a much cleaner, more modular way. Figure 9(a) graphically illustrates the composition of arrows for the code given in part (b) of the figure. We introduce four *regular* functions *setupA*, *dragA*, *dropA*, and *cancelA* to implement the actions in each possible drag-and-drop state. In contrast to Figure 1(b), these functions do not contain any of the plumbing of the state machine—all they do is carry out the appropriate actions based on the event, and then return the event target. Lines 18–32 compose these arrows with event handlers to build the state machine. For example, the arrow *dragOrDropA* on lines 18–20 (shown in a dashed box in the part (a)) connects the drag state to itself (upon a *mousemove*) or to the drop state (upon *mouseup*).

Notice that the arrow composition diagram in Figure 9(a) corresponds almost directly to the state machine we drew in Figure 1(a). Furthermore, the composition on lines 18–32 is simply a transliteration of the arrow diagram. We think this is a much clearer way to structure this event handling code than the standard approach we saw earlier.

Perhaps more importantly, *setupA*, *dragA*, *dropA*, and *cancelA* are independent of each other and their event triggers. Thus, we can reuse them in different compositions of arrows. For example, Figure 9(c) shows an alternative drag-and-drop implementation that is initiated by a *mouseover* event, rather than a *mousedown* event.

We can even re-use the elementary arrows of drag-and-drop in a different application. Figure 9(d) shows the basic control-flow of a jigsaw puzzle game. One piece of the jigsaw puzzle is first displayed (line 1), picked up with a *click* event (line 2), and then moved with the cursor until being dropped by the *mouseup* event inside *dragOrDropA* (line 4). However, the piece may be automatically picked up again if it was dropped in the wrong place (line 5). And the whole composition repeats with the next jigsaw piece. Re-using the code in Figure 1(b) to build this new structure would be



(a) Arrow diagram

```

1  function setupA(event) {
2      /* setup drag-and-drop */
3      return event.currentTarget;
4  }
5  function dragA(event) {
6      /* perform dragging */
7      return event.currentTarget;
8  }
9  function dropA(event) {
10     /* perform dropping */
11     return event.currentTarget;
12 }
13 function cancelA(event) {
14     /* cancel drag-and-drop */
15     return event.currentTarget;
16 }
17
18 var dragOrDropA =
19     ( (EventA("mousemove").next(dragA).next(Repeat))
20     .or(EventA("mouseup").next(dropA).next(Done))
21     ).repeat();
22
23 var dragDropOrCancelA =
24     (EventA("mousemove").next(dragA).next(dragOrDropA))
25     .or(EventA("mouseup").next(cancelA));
26
27 var dragAndDropA = /* drag-and-drop */
28     EventA("mousedown")
29     .next(setupA).next(dragDropOrCancelA);
30
31 ConstA(document.getElementById("dragtarget"))
32     .next(dragAndDropA).run();

```

(b) JavaScript implementation

```

1  ConstA(document.getElementById("dragtarget"))
2      .next(EventA("mouseover"))
3      .next(setupA).next(dragDropCancelA)
4      .run()

```

(c) Alternative—trigger on mouseover

```

1  (nextPieceA
2      .next(EventA("click"))
3      .next(setupA)
4      .next((dragOrDropA
5          .next(repeatIfWrongPlaceA)).repeat()
6          )
7      ).repeat()
8      .run()

```

(d) Jigsaw game re-using drag-and-drop elementary arrows

Figure 9: Drag-and-drop with arrows

non-trivial, while with asynchronous event arrows, building such compositions is straightforward.

Figure 9(b) also illustrates how programmers gain the benefits of asynchronous event arrows without having to know much about our library’s implementation. `setupA`, `dragA`, `dropA`, and `cancelA` are all regular JavaScript functions, and `EventA`, `next`, and `run` all have intuitive semantics.

4.5 Discussion

We believe our asynchronous arrow library is potentially useful for a variety of applications beyond those presented. In essence, our library can be used to construct arrows corresponding to arbitrary state machines, where transitions between states are via synchronous or asynchronous calls. The combinators `next` and `or` can construct machines that are DAGs (where each machine state corresponds to a single handler), and with looping we can create arbitrary graphs. It is easy to imagine other applications that could be built from such state machines, such as games (where related actions in the game, e.g., matching two cards, could be composed into a state machine) or productivity applications (where various elements of the UI have state).

In addition to timeouts and UI events, our library can also support other kinds of events, e.g., completion of a network or file I/O call. Indeed, our original motivation for developing arrows was to make it easier to write a web-based code visualization tool⁶ that starts by loading XML files over the network. To do this we create a composite asynchronous arrow that first loads an index file, and then iteratively loads each file present in the index, where one load commences when the previous one completes.

There are also many possible extensions to our library. For example, right now `EventA` has arrow type *CpsA target event*, and these events are interleaved with arrows of type *CpsA event target*. It would be easy, and probably useful, to allow state to be threaded through an arrow composition, e.g., `EventA` could have type *CpsA (target ,a) (event ,b)*, and we could compose these with arrows of type *CpsA (event ,b) (target ,a)*, so that state can be carried through the computation. We may also want to add new variations on `EventA` combinators. For example, we may want a version of `next` that does not uninstall the previous handler, to improve efficiency in the case we have a handler that remains live over many events. We leave exploring these and other interesting directions to future work.

5. Related Work

There are three main areas to consider: libraries based on functional programming techniques; JavaScript libraries that provide high-level, event-driven widgets; and other work on event-driven programming.

Libraries based on functional programming Our inspiration to develop an arrow-based library comes from a number of related libraries in Haskell such as `Fudgets` (Carlsson and Hallgren 1993) and `Yampa` (Hudak et al. 2003). `Fudgets` is a library for building graphical user interfaces (GUI), and uses arrows to implement GUI elements such as buttons, menus, and scroll bars, as well as event handlers. A complete GUI application is composed of `Fudgets` using various combinators.

`Yampa` is another arrow-based library for Haskell, but is designed for *functional reactive programming* (FRP). FRP is a programming pattern that introduces the concept of *time-varying values* and automates the propagation of changes in such values. In `Yampa`, time-varying values are modeled as *signals*, and are processed by *signal functions*. Signal functions are in turn implemented as arrows, and can thus be composed using arrow combinators. While the authors demonstrated `Yampa` in a robotics simulator

(Hudak et al. 2003), a GUI library named `Fruit` has also been built with `Yampa` (Courtney and Elliott 2001), where events are modeled as signals that change on user input, and event handlers are analogous to signal functions.

Our library is intended for building interactive web applications. Unlike standard GUI applications such as those written in `Fudgets` or `Fruit`, web applications are typically developed in a combination of HTML and CSS to define the graphical layout of interface elements, and JavaScript for the interface behavior (i.e., event handling). Our library is designed with this distinction in mind and focuses on composing event handlers in JavaScript.

We are aware of another JavaScript library, `Flapjax` (Meyerovich 2007), that is inspired by research in functional programming. `Flapjax` is an implementation of FRP in JavaScript, in which data sources, e.g., user input or events, can be created and connected to data sinks, e.g., a text box. `Flapjax` maintains these connections in a data-flow graph, and changes in input data are automatically propagated through the data-flow graph to the data sinks. Although it is not based on arrows, `Flapjax` supports many of the same complex combinators such as loops and branches.

In contrast to `Flapjax`, our library was originally designed with the simpler goal of composing event handlers. Our lightweight CPS-based arrows are more suited to smaller sequences of events since we need not build a data-flow graph; whereas for complex applications, such as spreadsheets, `Flapjax` can optimize data propagation through the data-flow graph, e.g., to propagate events only when the data-sink is ready.

JavaScript libraries Many JavaScript libraries have been developed to ease the construction of rich and highly interactive web applications that are now associated with Web 2.0. Example libraries include `jQuery` (`jquery.com`), `Prototype` (`prototypejs.org`), `YUI` (`developer.yahoo.com/yui`), `MochiKit` (`mochikit.com`), and `Dojo` (`dojotoolkit.org`). These libraries generally provide high-level APIs for common features, e.g., drag-and-drop, animation, and network resource loading, as well as to handle API differences between browsers. For example, in `jQuery`, one can make a document widget box “draggable” within its surrounding text area with the syntax `$(box).draggable()`. The main drawback of these libraries is that high-level features can be used in relatively few (but hopefully common) scenarios; even slight variations may be impossible without modifying the library internals (which may be difficult to understand, for reasons discussed in Section 2.2). In contrast, with arrows as the foundational element, high-level features can be both understandable and more customizable. For example, we believe our arrow-based approach makes it much easier for programmers to reason about drag-and-drop, by separating plumbing code from action code.

One form of customizability often exposed in library interfaces is the notion of an *animation queue*. The queue contains one or more *effects*, each of which represents a particular type of animation, e.g., fading a HTML element. Effect playback is implemented using a loop where each iteration is triggered by a timer event (as the last example in Section 2). Programmers can thus sequence animations, even dynamically, by adding them to the queue. Such queues are easily implemented in our framework (by constructing loops as shown in Section 4.2), and indeed are more customizable. Based on the examples we have shown, it is easy to imagine customizing the queue timeout duration, creating a queue of queues, or cancelling a series of queued events.

To allow operations to be composed, some libraries provide an idiom called *method-chaining* (e.g., `jQuery` is built on this idiom). In this idiom, all methods of an object returns the object itself, such that further methods can be invoked on it. In the following `jQuery` snippet, the document body is first wrapped in a proxy object using

⁶<http://www.cs.umd.edu/projects/PL/PP/>

the `$` function. The first call to `animate` returns the proxy object, which we can then call `animate` on again:

```
var body = document.body;
$(body)
  .animate({scrollTop: body.scrollHeight }, 2000)
  .animate({scrollTop: 0}, 2000);
```

This code is quite readable, as it is easy to see that a sequence of operations is performed on the same object. Composed arrows have a similar lightweight syntax (particularly for sequencing, though less so for other combinators), but are more flexible as they also support asynchronous invocations.

Event-driven programming It is well-known that threads and events are computationally equivalent (Lauer and Needham 1978). Recently there has been a flurry of research that explores this relationship more closely. The basic observation is that single-threaded event-driven programming is analogous to multi-threading with cooperative scheduling, in which one thread runs at a time and yields the CPU either voluntarily or when it would block on an I/O operation (Adya et al. 2002; von Behren et al. 2003; Li and Zdancewic 2007). One can view a thread as a sequence of event handlers, where each handler’s final action is to register an event that triggers the next handler in the sequence (e.g., after a timeout or I/O event completion).

Li and Zdancewic (2007) have built a thread monad for Haskell that follows this observation. In particular, users write a thread as a monad that concludes with a potentially blocking operation, and monad composition weaves these atomic handlers into a thread with several potential blocking points. The `do` syntactic shorthand for monads in Haskell makes these blocking points more transparent to the user by making composition lightweight. Our simple event arrow provides a similar API: the user writes a “thread” as a sequence of handlers separated by various (blocking) events, composed together by `next`. Each simple event arrow on which we invoke the `run` method can be viewed as a separate thread. On the other hand, full asynchronous arrows provide more than sequential composition, which allow us to build more general state machines, and not just sequentially-composed threads.⁷ It would be interesting to explore whether our additional composition operators would be useful in their setting.

6. Conclusion

We presented the Arrowlets library for using arrows in JavaScript. The key feature of our library is support for asynchronous event arrows, which are triggered by events in a web browser such as mouse or key clicks. By providing sequencing, looping, and branching combinators, programmers can easily express how their event handlers are composed and also separate event handling from event composition. Our library supports sophisticated interface idioms, such as drag-and-drop. The library also includes progress arrows, which can be used to monitor the execution of an asynchronous arrow or abort it. In translating arrows into JavaScript, our library provides programmers the means to elegantly structure event-driven web components that are easy to understand, modify, and reuse.

Acknowledgments

This research was supported in part by National Science Foundation grants IIS-0613601 and CCF-0541036.

⁷Note that Li and Zdancewic support looping within the handler code, not at the level of code composition, using recursive function calls.

References

- Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-880446-00-6.
- A.W. Appel. *Compiling with continuations*. Cambridge University Press New York, NY, USA, 1992.
- Magnus Carlsson and Thomas Hallgren. Fudgets: a graphical user interface in a lazy functional language. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 321–330, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X. doi: <http://doi.acm.org/10.1145/165180.165228>.
- Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *In Proceedings of the 2001 Haskell Workshop*, pages 41–69, 2001.
- Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming, 4th International School, volume 2638 of LNCS*, pages 159–187. Springer-Verlag, 2003.
- John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000. URL <http://www.cs.chalmers.se/~rjmh/Papers/arrows.ps>.
- Hugh C. Lauer and Roger M. Needham. On the duality of operating systems structures. In *Proceedings Second International Symposium on Operating Systems*, October 1978.
- Peng Li and Steve Zdancewic. Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 189–199, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: <http://doi.acm.org/10.1145/1250734.1250756>.
- Leo Meyerovich. Flapjax: Functional Reactive Web Programming, 2007. URL <http://www.cs.brown.edu/research/pubs/theses/ugrad/2007/lmeyerov.pdf>.
- Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: scalable threads for internet services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 268–281, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: <http://doi.acm.org/10.1145/945445.945471>.
- Philip Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1992. ACM. ISBN 0-89791-453-8. doi: <http://doi.acm.org/10.1145/143165.143169>.