Directing JavaScript with Arrows

Khoo Yit Phang Michael Hicks Jeffrey S. Foster Vibha Sazawal
University of Maryland, College Park
{khooyp,mwh,jfoster,vibha}@cs.umd.edu

Abstract

Event-driven programming in JavaScript often leads to code that is messy and hard to maintain. We have found *arrows*, a generalization of *monads*, to be an elegant solution to this problem. Our arrow-based *Arrowlets* library makes it easy to compose event-driven programs in modular units of code. In particular, we show how to implement *drag-and-drop* modularly using arrows.

1. Event-driven JavaScript is Messy

JavaScript is the *lingua franca* of Web 2.0, and is the basis of highly interactive web applications such as Google Maps and Flickr. Because JavaScript code runs in a client-side web browser, applications can present a rich, responsive interface without the latency associated with client-server communication.

JavaScript, however, is a single-threaded language that has to cooperate with the web browser's user interface. The JavaScript API is designed in event-driven style, and it is crucial that event callbacks execute quickly so that new events are handled in a timely fashion. Long-running loops, animations, and state machines are typically implemented by chaining callbacks, each of which ends by registering one or more additional callbacks. Unfortunately, writing this style of code is typically tedious, error-prone, and non-modular because the callback chaining code (the "plumbing") is often hard-coded and strewn throughout the program.

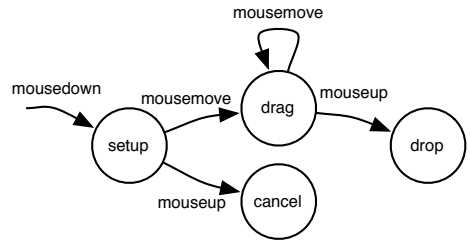
Drag-and-drop is a prototypical example that illustrates these issues. Figure 1 shows a common way to implement drag-and-drop in JavaScript. The four states—setup, drag, drop and cancel—are implemented as event handlers, and each handler is responsible for installing handlers for the next states. For example, when setup executes, it has to disable itself (line 4) and install drag and cancel (lines 5–6). Since the states are hard-coded, we cannot re-use setup in another application, and if we want to insert a new state in the state machine, we may need to edit several different handlers.

2. Arrows Point the Way

Inspired by libraries such as Fudgets (Carlsson and Hallgren 1993) and Yampa (Hudak et al. 2003) in Haskell, we discovered that *arrows* (Hughes 2000), a generalization of *monads*, is an elegant way to compose event-driven programs in JavaScript.

Arrows support at least two operations: *arr f* lifts a function *f* into an arrow, and $f \ggg g$ composes a new arrow where *g* is applied to the output of *f*. Figure 2 shows two (very) simplified definitions of function arrows. In Haskell, we define function arrows as the *Arrow* ($\rightarrow$) type with *arr* as the identity function and $\ggg$ as function composition. In JavaScript, we extend every function object with the arrow interface by adding the methods A and next, equivalent to *arr* and $\ggg$, to the built-in Function.prototype object.

We can apply arrows to the observation that event listener functions such as addEventListener are *continuation passing style* (CPS) functions where the callbacks are the continuations. By abstracting




```
1  function setup(event) { /* likewise for drag, drop, cancel */
2      var target = event.currentTarget;
3      /* setup drag-and-drop */
4      target.removeEventListener("mousedown", setup, false);
5      target.addEventListener("mousemove", drag, false);
6      target.addEventListener("mouseup", cancel, false);
7  }
8
9  document.getElementById("dragtarget")
10         .addEventListener("mousedown", setup, false);
```

Figure 1: Drag-and-drop state diagram and JavaScript code

```
1  instance Arrow (→) where
2      arr f       = f
3      (f ⋙ g) x = g (f x)
4
5  add1 x = x + 1
6  add2 = add1 ⋙ add1
7  result = add2 1      {- returns 3 -}
```

```
1  Function.prototype.A = function() { /* arr */
2      return this;
3  }
4  Function.prototype.next = function(g) { /* ⋙ */
5      var f = this; g = g.A(); /* ensure g is a function */
6      return function(x) { return g(f(x)); }
7  }
8
9  function add1(x) { return x + 1; }
10 var add2 = add1.next(add1);
11 var result = add2(1);      /* returns 3 */
```

Figure 2: Function arrows in Haskell (top) and JavaScript (bottom)

event listeners into a library of CPS arrows, we can still write event handlers as regular functions, and use arrow operations to lift and compose handlers with listeners. This nicely encapsulates the event handling code in arrows, and the plumbing in arrow combinators, thereby separating them and making re-use practical.

3. *Arrowlets*

Following this inspiration, we developed *Arrowlets*, a JavaScript library for event-driven programming. The core building block of the Arrowlets library is the AsyncA arrow prototype, from which all arrows are built. A simplified version of AsyncA is shown in Figure 3. The AsyncA constructor (lines 2–4) creates an arrow

```
1  /* AsyncA is the prototype for asynchronous arrows */
2  function AsyncA(cps) {                    /* constructor */
3      this.cps = cps; /* cps :: (x, k) → () */
4  }
5  AsyncA.prototype.AsyncA = function() {   /* identity */
6      return this;
7  }
8  AsyncA.prototype.next = function(g) {    /* sequencing */
9      var f = this; g = g.AsyncA();
10     /* CPS function composition */
11     return new AsyncA(function(x, k) {
12         f.cps(x, function(y) { g.cps(y, k); });
13     });
14 }
15 AsyncA.prototype.run = function(x) {     /* running */
16     this.cps(x, function(y) {});
17 }
18 Function.prototype.AsyncA = function() { /* lifting */
19     var f = this; /* wrap f in CPS function */
20     return new AsyncA(function(x, k) { k(f(x)); });
21 }
22
23 /* An EventA arrow waits for an event to fire on a target */
24 function EventA(eventname) {             /* constructor */
25     if (!(this instanceof EventA)) return new EventA(eventname);
26     this.eventname = eventname;
27 }
28 EventA.prototype = new AsyncA(function(target, k) {
29     var f = this;
30     function handler(event) {
31         target.removeEventListener(f.eventname, handler, false);
32         k(event); }
33     target.addEventListener(f.eventname, handler, false);
34 });
```

Figure 3: Simplified AsyncA arrow prototype and EventA arrow

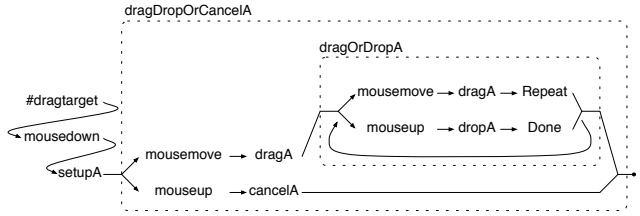
around a cps function, and arrow combinators such as next (lines 8–14) are implemented in CPS. We can execute an AsyncA arrow by invoking the run method (lines 15–17), which calls cps with an input x and an empty terminal continuation. Finally, we extend Function.prototype with an AsyncA method to lift a regular function into an AsyncA arrow (lines 18–21).

With AsyncA, we can define the EventA event listener arrow. The EventA constructor (lines 24–27) creates an arrow that listens to an event named eventname. The constructor contains a convenient JavaScript idiom (line 25) that allows us to create EventA arrows without using the **new** operator. EventA inherits from AsyncA (lines 28–34), and is built around a CPS function that listens for an event. When EventA executes, it first installs a stub event handler on the input target element. After the event fires, it uninstalls the stub event handler, and invokes the continuation k—the next arrow and actual event handler—with the received event. We chose to uninstall the event handler as this corresponds to transitions in a state machine, which is one of our motivating use cases.

Our library also provides other arrows, e.g., the ElementA arrow that ignores its input and returns a specified element from the host HTML document. We also provide additional combinators such as repeat, which puts an arrow in a loop; the arrow may return Repeat(x) to run another iteration, or Done(x) to end the loop. Another useful combinator is or, which composes two event arrows and allows only one, whichever is triggered first, to execute.

4. Drag-and-Drop with Arrowlets

Figure 4 shows how we can use Arrowlets to implement drag-and-drop in an intuitive and modular way. As before, we write four event handlers—setupA, dragA, dropA and cancelA—corresponding to the four states in drag-and-drop. Like setup in Figure 1, setupA (lines 1–4) is written as a regular function, but in contrast, it does not contain any callback plumbing code. Since it is not tied to the other handlers, it can be re-used in other applications.




```
1  function setupA(event) { /* likewise for dragA, dropA, cancelA */
2      /* setup drag−and−drop */
3      return event.currentTarget;
4  }
5
6  var dragOrDropA =
7      (    (EventA("mousemove").next(dragA).next(Repeat))
8       .or(EventA("mouseup").next(dropA).next(Done))
9      ).repeat();
10
11 var dragDropOrCancelA =
12         (EventA("mousemove").next(dragA).next(dragOrDropA))
13     .or(EventA("mouseup").next(cancelA));
14
15 var dragAndDropA = /* drag−and−drop */
16     EventA("mousedown").next(setupA).next(dragDropOrCancelA);
17 ElementA("dragtarget").next(dragAndDropA).run();
18
19 var jigsawA =          /* alternative use of dragOrDropA */
20     (nextPieceA
21         .next( (dragOrDropA.next(repeatIfWrongPlaceA)).repeat() )
22     ).repeat();
```

Figure 4: Drag-and-drop arrow diagram and code

The plumbing that composes the handlers has been extracted into the remainder of the code in Figure 4. We use various arrow combinators to compose the handlers and appropriate event listeners into the drag-and-drop state machine. We can also organize the composition modularly in three parts. For example, the first part, dragOrDropA (lines 6–9), is a repeat loop that handles the dragging animation during mousemove events, and the dropping action after a mouseup. In addition to drag-and-drop (lines 15–17), we can even re-use dragOrDropA in a jigsaw puzzle game (lines 19–22).

Finally, this drag-and-drop composition, shown graphically above Figure 4, mirrors the state diagram in Figure 1. We find it quite intuitive to convert a state diagram into an arrow composition.

In conclusion, arrows makes it easy to write event-driven programs in an intuitive and modular way. The Arrowlets library is available at our website (`http://www.cs.umd.edu/projects/PL/arrowlets`), along with a technical report, API documentation and several live examples.

Acknowledgments

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

References

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.

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`.