Directing JavaScript with Arrows

Khoo Yit Phang  Michael Hicks  Jeffrey S. Foster  Vibha Sazawal
University of Maryland, College Park
{khooyp,mwh,foster,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 >>> 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 (→) type with arr as the identity function and >>> 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 >>>, 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

![Figure 1: Drag-and-drop state diagram and JavaScript code](image1)

![Figure 2: Function arrows in Haskell (top) and JavaScript (bottom)](image2)

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
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.

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, arrow 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

