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

JavaScript programmers make extensive use of event-driven programming to help build 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 this paper, we present Arrowlets, a new JavaScript library that offers arrows to the everyday JavaScript programmer. We show how to use Arrowlets to construct a variety of state machines, including state machines that branch and loop. We also demonstrate how Arrowlets separate computation from composition with examples such as a drag-and-drop handler and a bubblesort animation.

1. INTRODUCTION

JavaScript is the *lingua franca* of Web 2.0. With JavaScript code running 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 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.

To combat this problem, many researchers and practitioners have developed libraries to ease the construction of rich and highly interactive web applications. Examples 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. Unfortunately, while these libraries do support a number of common scenarios, even slight customizations may be impossible to achieve without modifying the library internals.

In this paper, we describe *Arrowlets*, a new JavaScript library for *composable* event handling. Arrowlets is designed to be lightweight, and supports much easier reuse and customization than the approaches mentioned above. Arrowlets is based on *arrows*, a programming pattern closely related to monads [6], which are used extensively in Haskell. An arrow abstraction resembles a normal function, with the key feature that arrows can be composed in various ways to create new arrows. With Arrowlets, 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. Other approaches to enabling more modular and reusable JavaScript code have also been explored in the literature, e.g., Flapjax [9], which is based around functional reactive programming [12]. Arrowlets offer a complementary approach that may be more similar to existing JavaScript programming style while still offering the benefits of separating computation from composition.

In the remainder of the paper, 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 Arrowlets. We next scale up to include richer combinators and present several examples. We conclude by comparing our work to related approaches. We believe that Arrowlets 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. 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.

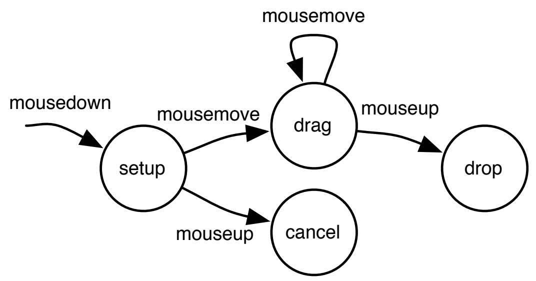
To solve this problem, JavaScript programs make heavy use of *event-driven programming*, in which programs register asynchronous callbacks to handle UI interactions and break up long-running tasks. For example, the following program[1] registers the clickTarget callback on the HTML element named target, and will be called each time target is clicked:[2]

```
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). 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, as we show next using drag and drop.



(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

2.1 Standard Drag-and-Drop

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).

Even though this code has been distilled down to only the control flow, it is not that easy to understand. The flow of control is particularly convoluted: each event handler ends by returning, but the actual control "continues" indirectly to one of several subsequent event handlers. Code reuse is also hard, because each event handler hard-codes its subsequent event handlers, and the side-effects of drag-and-drop are interspersed with the state transition logic. For example, if we wanted to initiate drag-and-drop with a mouseover event, we would need to make a new copy of setup.

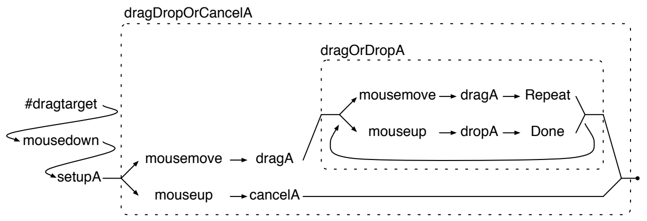
2.2 Drag-and-Drop with Arrowlets

Event-driven programming in JavaScript does not have to be convoluted and monolithic: with Arrowlets, we can easily write event-driven code that is clean, understandable and reusable.

Figure 2(a) gives an *arrow composition diagram* outlining

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

[2]The 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) Arrow diagram

```
1  function setupA(proxy, event) {
2      proxy.setupDrag(event);
3      return proxy;
4  }
5  function dragA(proxy, event) {
6      proxy.moveDrag(event);
7      return proxy;
8  }
9  function dropA(proxy, event) {
10     proxy.dropDrag(event);
11     return proxy;
12 }
13 function cancelA(proxy, event) {
14     proxy.cancelDrag(event);
15     return proxy;
16 }
17
18 var dragOrDropA =
19   (    ((EventA("mousemove").bind(dragA)).next(Repeat))
20     .or((EventA("mouseup").bind(dropA)).next(Done))
21   ).repeat();
22
23 var dragDropOrCancelA =
24       ((EventA("mousemove").bind(dragA)).next(dragOrDropA))
25   .or((EventA("mouseup").bind(cancelA)));
26
27 var dragAndDropA =  /* drag-and-drop */
28   (EventA("mousedown").bind(setupA))
29     .next(dragDropOrCancelA);
30
31 DragElementA("dragtarget").next(dragAndDropA).run();
```

(b) JavaScript implementation

```
32 DragElementA("dragtarget")
33   .next(EventA("mouseover").bind(setupA))
34   .next(dragDropOrCancelA)
35   .run()
```

(c) Alternative—setup on mouseover

```
36 (nextPieceA
37   .next(EventA("click").bind(setupA))
38   .next((dragOrDropA
39            .next(repeatIfWrongPlaceA)).repeat()
40        )
41 ).repeat()
42 .run()
```

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

Figure 2: Drag-and-drop with arrows

drag-and-drop with Arrowlets. It corresponds almost directly to the state machine in Figure 1(a), except we have indicated the target (#dragtarget) and written events as nodes. The two dashed boxes indicate portions of the composition we intend to reuse.

Figure 2(b) shows the complete implementation of drag-and-drop with Arrowlets. As before, we introduce four functions, setupA, dragA, dropA, and cancelA, to handle the respective drag-and-drop states. Each handler takes as input a *drag proxy* and the triggering event. The drag proxy wraps the actual target, and additionally implements four methods—setupDrag, moveDrag, dropDrag, cancelDrag. The bodies of these four methods correspond exactly to the code represented by lines 6, 12, 18, and 24 in Figure 1(b). Using proxy objects affords greater flexibility in customizing the side-effect of drag-and-drop, and is a standard trick found in many JavaScript libraries. For example, one drag-and-drop machine could use a proxy that moves the target when dragged, while another proxy could be used to draw a trail behind the dragged target.

Combinator	Use
f.AsyncA()	lift function f into an arrow (called automatically by combinators below)
h = f.next(g)	h(x) is f(g(x))
h = f.product(g)	h takes a pair (a,b) as input, passes a to f and b to g, and returns a pair (f(a), g(b)) as output
h = f.bind(g)	h calls f with its input x, and then calls g with a pair (x, f(x)).
h = f.repeat()	h calls f with its input; if the output of f is: Repeat(x), then f is called again with x; Done(x), then x is returned
h = f.or(g)	h executes whichever of f and g is triggered first, and cancels the other
h.run()	begins execution of h

Table 1: Arrow combinators used by drag-and-drop.

The remainder of Figure 2(b) assembles the drag-and-drop state machine from the handlers we have written. To do so, we use *combinators* to compose the handlers with the appropriate event triggers. These combinators compose *arrows*, which include regular functions or specialized event listeners that we provide, and describe serial and parallel combinations of arrows. Table 1 describes the arrow combinators we use to implement drag-and-drop; full details appear in Sections 3 and 4.

In Figure 2(b), we have split the composition into three pieces: dragOrDropA, dragDropOrCancelA, dragAndDropA. Each handler is connected to its triggering event using bind, and then connected to each other with other combinators. For example, the arrow dragOrDropA on lines 18–21 (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).

Finally, the entire drag-and-drop composition is installed on the DragElementA("dragtarget") drag proxy on line 31. The arrow DragElementA, which we have elided, locates and returns the named HTML element wrapped in the proxy object that implements the animation of drag-and-drop.

Notice that the composition on lines 18–31 is simply a transliteration of the arrow diagram. We think this is a much more direct way to construct this event handling code than the standard approach we saw earlier, while retaining its basic computational structure. In particular, with Arrowlets the code implementing the effect of each handler is essentially unchanged; the difference is that hard-coded, indirect state transitions have been traded for more flexible,

```
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 3: Two definitions of arrows

direct arrowlet compositions. Moving to a system like Flapjax, which uses an entirely different computational model, would require more pervasive changes [9]. (See Section 5 for more discussion.)

Once we have used Arrowlets to separate setupA, dragA, dropA, and cancelA from each other and their event triggers, we can reuse them in different compositions of arrows. For example, Figure 2(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 2(d) shows the basic control-flow of a jigsaw puzzle game. One piece of the jigsaw puzzle is first displayed (line 36), picked up with a click event (line 37), and then moved with the cursor until being dropped by the mouseup event inside dragOrDropA (line 38). However, the piece may be automatically picked up again if it was dropped in the wrong place (line 39). 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 non-trivial, while with asynchronous event arrows, building such compositions is straightforward.

3. IMPLEMENTATION OF ARROWLETS

The design of Arrowlets is based on the concept of arrows from Haskell [6]. Arrows are generalizations of monads [11], 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.

We will first explain the concept of arrows with the simplest possible arrows, namely regular functions, and then build on it to describe how various components of the Arrowlets library are implemented in JavaScript.

3.1 Function Arrows

Figure 3(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 operations: *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* (→), represent standard functions where *arr* is the identity function, and ⋙ 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 -}
```

Figure 3(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 3(b), we add two methods to every function object. The A method, corresponding to *arr*, lifts a function into an arrow. As in Haskell, the A method is just the identity. The next combinator, corresponding to ⋙, 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 combinator 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:[3]

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

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

JavaScript lacks Haskell's sophisticated type system, so we unfortunately cannot statically ensure arrows are used correctly. For example, the definition in Figure 3(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, e.g., they are registered as callbacks with functions such as addEventListener. Instead, in Arrowlets we use *continuation passing style* (CPS) [2], a well-known programming technique. We convert all functions to take a continuation parameter, which is akin to an event handler callback, and use CPS composition to compose regular functions and asynchronous functions.

Figure 4 shows how we use CPS in Arrowlets. 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.

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

```
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 4: CPS function arrows in JavaScript

On line 4 we introduce our convention of giving CpsA objects an identity CpsA method, which we use like the A method from Figure 3(b): if we successfully invoke $x = x$.CpsA(), we know x is a CPS arrow. The next combinator is implemented using the CPS equivalent of function composition,[4] 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 with the actual argument and a do-nothing function that acts as the final continuation. 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 Arrowlets 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)));} */
```

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 5.

In this code, the function SimpleEventA acts as a constructor, where line 3 implements a convenient JavaScript idiom.

[4]JavaScript lacks tail-call optimization, so this simple definition of next can cause the call stack to overflow. Our actual implementation uses trampolines to avoid this issue.

```
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 5: SimpleEventA for handling JavaScript listeners

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.

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 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 5 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. To do this we need to introduce a number of arrows and combinators. We introduce these by example next, and defer a detailed description of their implementation to Section 4.2.

4.1 Arrows and Combinators

Asynchronous Event Arrows with Progress. Previously, we introduced an arrow SimpleEventA for handling particular events. In practice, we want to support multiple arrows "in flight" at a time, e.g., in the setup state of drag-and-drop, we wait for multiple events at once. Thus we must be able to cancel an event handling arrow while it is active so that we can switch to another state.

Our solution is to create a new arrow AsyncA, which extends CpsA with support for tracking progress and cancellation. When an AsyncA is run, it returns a *progress arrow*, which can be used to observe or cancel execution of the arrow. We use AsyncA to build EventA, which similarly extends SimpleEventA with support for progress and cancellation. Using EventA, we can augment the two-click example from the prior section so that it returns a progress arrow p. We can then use p to affect the arrow in flight, e.g., to stop waiting for clicks after 10 seconds, as shown below.

```
var target = document.getElementById("target");
var p = EventA("click").next(clickTargetA)
    .next(EventA("click").next(clickTargetA))
  .run(target);
/* p can be used to abort the event arrow */

setTimeout(function() {
    p.cancel();  /* cancels event arrow */
    target.textContent = "Can't click this";
}, 10000);
```

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

```
var status = document.getElementById("status");
p.next(function() {
    /* called when event arrow finishes */
    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.

Looping with repeat(). To support loops in state machines, such as the drag state in our drag-and-drop example, we provide a repeat combinator. The expression f.repeat(t) creates an arrow that puts f in a loop. If f returns returns Repeat(x), then f is called again with x after t milliseconds have elapsed; otherwise if j returns Done(x), then the loop completes and returns x.

As an example, we can use repeat to implement an animation of bubble sort:

```
var bubblesortA = function(x) {
    var list = x.list, i = x.i, j = x.j;
    if (j + 1 < i) {
        if (list.get(j) > list.get(j + 1)) {
            list.swap(j, j + 1);
        }
        return Repeat({ list : list, i:i, j:j + 1 });
    } else if (i > 0) {
        return Repeat({ list : list, i:i − 1, j:0 });
    } else {
        return Done();
    }
}.AsyncA().repeat(100);

/* list is an object with methods get and swap */
bubblesortA.run({ list : list, i: list.length, j: 0 });
```

The arrow bubblesortA takes as input an object that contains three properties: the list to be sorted, and indices i and j representing the iteration state. Here, list is a proxy object with methods for looking up an element (get) and for swapping two elements (swap); these methods could be implemented to visualize the progress of sorting in the display.

The body of bubblesortA is a standard bubble sort, except instead of a **do-while** loop, we continue iteration by returning a Repeat object with updated values for list, i, and j, or we return Done when the loop is complete. We create the arrow by using the repeat combinator with an interval of $100ms$ so that the bubble sort can be visualized slowly. We could also use a shorter interval for improved performance.

Parallel arrows with product(). Arrows also provides combinators for parallel compositions. The product combinator (*** in Haskell) takes two arrows and produces a new arrow that takes as input a pair, applies the constituent arrows to each component of the pair, and outputs the result in a pair. For Arrowlets, we additionally define the product combinator to execute concurrently and synchronize at the completion of both arrows.

Using the product combinator, we can extend the example at the end of Section 3.3 to respond only after the user clicks on two targets in any order. First, we modify clickTargetA to update two targets:

```
function clickTargetsA(target1, target2) {
    target1.textContent = "You clicked me!";
    target2.textContent = "And me too!";
}
```

Note that Arrowlets automatically unpacks a pair of targets into the argument list of clickTargetsA. We can then register this handler on two targets with the following plumbing:

```
(EventA("click").product(EventA("click")))
```

```
    .next( clickTargetsA )
    .run( Pair(document.getElementById("target1"),
               document.getElementById("target2")));
```

Now only after both targets are clicked on, which may be in any order, will the clickTargetsA handler be called to display the message.

*Branching with Either-*or(). Our next 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.

As an example, we demonstrate a simple coin-toss game implemented with or below. We first define WriteA to create an arrow that writes into an event's target element. Then, we compose two arrows that respond to clicks in heads and tails. Finally, we combine the arrows with or, ensuring that the player can only click once, on either heads or tails. Unfortunately, in this game, you'd never win.

```
function WriteA(str) {
    return function(event) {
        var target = event.currentTarget;
        target.textContent = str;
        return target;
    };
}

var heads = ConstA(document.getElementById("heads"));
var tails = ConstA(document.getElementById("tails"));

   (heads.next(EventA("click")).next(WriteA("I win!")))
  .or( tails .next(EventA("click")).next(WriteA("You lose!")))
  .run();
```

Forwarding Arrow Inputs. Finally, most arrows take an input, perform some computation on it, and output the result of the computation to the next arrow in sequence. However, there are many cases where we want pass the input "around" the next arrow, i.e., to keep the original input in addition to the output. For example, in our drag-and-drop example from Section 2.2, an event handler receives not just the event it was triggered on, but also the proxy object on which the event listeners were installed on. The bind combinator provides this feature. In the short example below, we show how an image manipulator can display before-and-after shots of a brightened image.

```
var showBeforeAndAfterA = brightenA.bind(displayA);
```

4.2 Implementation Details

Progress and Asynchronous Arrows. The first step of our implementation is to extend CpsA so that continuations take both the normal function argument x and a progress arrow argument p. Progress arrows are created with the ProgressA constructor, described below. Our CpsA definition extended with progress arrows is called AsyncA, for *asynchronous arrow*, and is shown in lines 1–10 of Figure 6. The constructor (line 1) and lifting function (line 2) work analogously to CpsA; next (lines 3–10) simply passes the extra parameter through the CPS composition. The run method now

```
 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 6: Full asynchronous arrows (part 1)

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 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 7(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 3(b)), but since AsyncA is somewhat more flexible, we choose that as our starting place. Figure 7(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

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

```
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 7: Full asynchronous arrows (part 2)

```
1  function Pair(x, y) { return { fst:x, snd:y }; }
2  AsyncA.prototype.product = function(g) {
3      var f = this; g = g.AsyncA();
4      return new AsyncA(function(x, p, k) {
5          var out1, out2, c = 2;
6          function barrier() {
7              if (--c == 0) k(Pair(out1, out2), p);
8          }
9          f.next(function(y1) { out1 = y1; barrier(); })
10          .run(x.fst, p);
11         g.next(function(y2) { out2 = y2; barrier(); })
12          .run(x.snd, p);
13     });
14 }
```

Figure 8: Combining arrows in parallel

```
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 }
```

Figure 9: Looping with AsyncA

combinator 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 arrow 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 and invoked (line 20). This behavior is analogous to the removal of event listeners after an event triggers.

product(). Figure 8 shows the product combinator for the AsyncA arrow. We use the Pair function to create pairs as objects with two properties, fst and snd. To implement product, we first define the function barrier on lines 6–8, which serves to synchronize the arrow at the completion of its constituent arrows. Then, we run the arrows f on lines 9–10 and g on lines 11–12, giving as input the first and second components of x, respectively. Upon completion of each arrow, we store the result into out1 or out2 and call barrier. When c reaches 0, i.e., when both f and g have completed, the barrier function will in turn call the continuation k with the results. Although not shown above, Arrowlets also unpacks pairs automatically into the argument list of regular (non-CPS) functions for convenience.

```
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         function cancel() {
13             if (p1) p1.cancel();
14             if (p2) p2.cancel();
15         }
16         /* prepare callback */
17         function join(y, q) {
18             p.advance(cancel);
19             k(y, q);
20         }
21         /* and run both */
22         p.addCanceller(cancel);
23         f.cps(x, p1, join);
24         g.cps(x, p2, join);
25     });
26 }
```

Figure 10: Branching with AsyncA

repeat(). Figure 9(a) shows the implementation of repeat. 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 an object created with either Repeat or Done (lines 20–21). These methods store their argument x in a JavaScript approximation of a tagged union—an object 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).

***Either*-or().** Figure 10 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.

```
var idA = function(x) { return x; }.AsyncA();
var dupA = function(x) { return Pair(x, x); }.AsyncA();

CpsA.prototype.bind = function(g) {
    var f = this; g = g.AsyncA();
    return dupA.next(idA.product(f))
               .next(g);
}
```

Figure 11: Passing state around arrows

bind(). bind can be implemented with next and product along with two helper arrows, as shown in Figure 11. bind creates a new arrow that first duplicates its input into a pair with dupA, passes through the first component untouched via idA and applies f to the second component, and finally applies g to the resulting pair.

Many combinators in Arrowlets such as bind are implemented with just next and product. In general, we find it quite simple in practice to compose additional convenience combinators as needed.

4.3 Discussion

We believe our asynchronous arrow library is potentially useful for a variety of applications beyond those presented. In essence, Arrowlets can be used to construct arrows corresponding to arbitrary state machines, where transitions between states occur 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 repeat 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, Arrowlets 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 tool[6] 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.

5. RELATED WORK

An earlier version of this work appeared as a poster at ICFP 2008, which included a much shorter, 2-page writeup. There are several threads of related work.

JavaScript libraries. JavaScript libraries, such as jQuery, Prototype, YUI, MochiKit, and Dojo, provide high-level APIs for common GUI features, such as drag-and-drop and animation. For example, in jQuery, one can make a document widget box "draggable" within its surrounding text area with the syntax $(box).draggable(). These high-level APIs are easy to use, but they are often not modular and are less customizable, as we have found with drag-and-drop.

One form of customizability that is offered by library interfaces is an animation queue. The queue contains one or

[6]http://www.cs.umd.edu/projects/PL/PP/

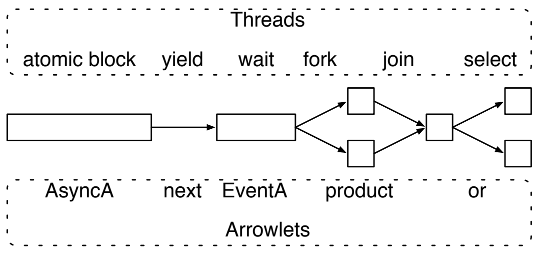


Figure 12: Analogies between threads and Arrowlets

more animation steps (called *effects*). Programmers can sequence animation steps dynamically by adding to the queue. Such queues and other more complex compositions are easily implemented with Arrowlets.

Some libraries also provide an idiom called *method chaining*. In this idiom, all methods of an object return the object itself, such that further methods can be invoked on it. Arrow combinators have a similar lightweight syntax, but are more flexible as they can be composed modularly.

Flapjax. Flapjax [9] is another approach to solving the problems we have mentioned, but takes a *data-flow oriented* approach, in the style of *functional reactive programming* [12]. In Flapjax, programmers use combinators to construct, compose, and filter first-class *event streams*, and these streams drive the processing that will ultimately update the display, write files, send messages, etc. Arrowlets is more *control-flow oriented*. In particular, handler arrows are naturally chained together to form state machines: each handler corresponds to a node in the machine, and how handlers are composed determines the transitions in the machine. We can customize the machine by composing handlers differently; in Flapjax we might instead customize the makeup of the event stream. Since Arrowlets do not manipulate first-class event streams, they are lower-level than Flapjax's abstractions, and arguably closer to Javascript programming practice. Indeed, in Section 2.2 we showed it was fairly straightforward to move from existing code to Arrowlets, and thus to gain Arrowlets' benefits of improved composability and reuse.

Threads and Event-driven programming. Perhaps surprisingly, programming with Arrowlets bears a striking resemblance to programming with threads, despite the fact that JavaScript is single-threaded. Indeed, it is well-known that threads and events are computationally equivalent [7]. Recently research has explored 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 [1, 10].

Li and Zdancewic have built a thread monad for Haskell that follows this observation [8]. We can translate their observation to arrows by viewing 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). By this view, an arrow composition using Arrowlets is analogous to a thread; Figure 12 suggests the relationship between threads and Arrowlets.

Haskell's Arrow libraries. Our inspiration to develop an arrow-based library comes from a number of related libraries in Haskell such as Fudgets [3] and Yampa [5]. 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 arrow-based library for functional reactive programming [5], as with Flapjax. In Yampa, arrow combinators compose *signals* with *signal functions*. Yampa has been used to implement a robotics simulator [5] as well a GUI library named Fruit [4].

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). Arrowlets is designed with this distinction in mind and focuses on composing event handlers in JavaScript.

Twisted's Deferred. Twisted (`twistedmatrix.com`) is an event-driven networking library for Python, which, like Arrowlets, provides an abstraction to manage event callbacks. In Twisted, asynchronous functions such as network file access do not take a callback parameter, but instead return `Deferred` objects that represent values that may have yet to arrive. The value in `Deferred` can be accessed by adding a callback function via its `addCallback` method. This approach resembles *monads*, a concept that is closely related to arrows.

Unlike `addCallback`, the `next` combinator in Arrowlets provides a consistent interface to compose asynchronous or synchronous functions with each other. To have a similarly consistent interface using `Deferred`, programmers have to be careful to wrap the return value of every function in a `Deferred` object, which is an error-prone endeavor in dynamically-typed languages such as Python or JavaScript. Also, `addCallback` composes `Deferred` objects with functions, whereas `next` composes arrows with arrows. This asymmetry makes it more verbose to write reusable compositions. Furthermore, Arrowlets provides a richer library of combinators for synchronization and asynchronous looping.

6. CONCLUSION

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

References

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. 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.

[2] A. Appel. *Compiling with continuations.* Cambridge University Press New York, NY, USA, 1992.

[3] M. Carlsson and T. 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.

[4] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *In Proceedings of the 2001 Haskell Workshop*, pages 41–69, 2001.

[5] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming, 4th International School, volume 2638 of LNCS*, pages 159–187. Springer-Verlag, 2003.

[6] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.

[7] H. C. Lauer and R. M. Needham. On the duality of operating systems structures. In *Proceedings Second International Symposium on Operating Systems*, Oct. 1978.

[8] P. Li and S. 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.

[9] L. Meyerovich, A. Guha, J. Baskin, G. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. Technical Report CS-09-04, Department of Computer Science, Brown University, 2009.

[10] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. 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.

[11] P. 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.

[12] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 242–252, New York, NY, USA, 2000. ACM.