Directing JavaScript with Arrows

Khoo Yit Phang, Michael Hicks, Jeffrey S. Foster, Vibha Sazawal

University of Maryland http://www.cs.umd.edu/projects/PL/arrowlets/

Cobwebs of Callbacks in JavaScript

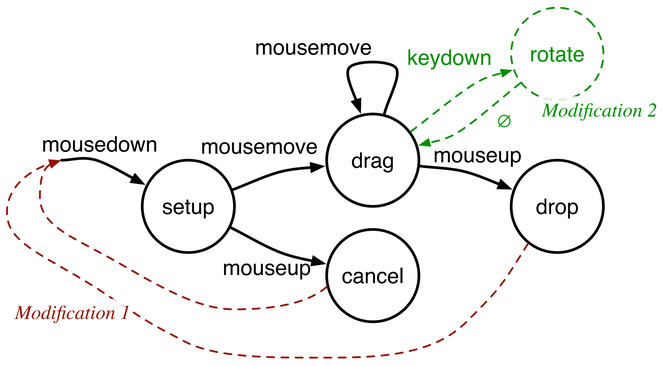
JavaScript is single-threaded, so responsive interactive applications are written in *event-driven style* using callbacks.

However, standard callback composition is **convoluted**, **tedious**, and **not modular**.

Example: Drag-and-Drop

Plumbing is convoluted: each handler disables itself and sets up the next callbacks:

```
function setup(event) {
    var target = event.currentTarget;
    /* setup drag–and–drop */
    target.removeEventListener("mousedown", setup, false);
    target.addEventListener ("mousemove", drag, false);
    target.addEventListener ("mouseup", cancel, false);
}
```



Modification is **tedious and error-prone**:

1. repeating drag-and-drop requires modifying *drag* and *cancel*;
2. adding a new state *rotate* requires modifying *drag* and *drop*.

Reusing parts of drag-and-drop is **practically impossible** due to hard-coded next states.

Arrows Point the Way…

…from Haskell…

```
instance Arrow (→) where
  arr f       = f
  (f >>> g) x = g (f x)

add1 x  = x + 1
add2    = add1 >>> add1
result  = add2 1 {– returns 3 –}
```

…to JavaScript (quite nicely too!)

```
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)); }
}

function add1(x) { return x + 1; }
var add2 = add1.next(add1);
var result = add2(1); /* returns 3 */
```

Arrowlets: Arrows for JavaScript

AsyncA prototype arrow wraps CPS functions

```
function AsyncA(cps) {
    this.cps = cps; /* cps :: (x, k) → () */
}
AsyncA.prototype.AsyncA = function() {
    return this;
}
AsyncA.prototype.next = function(g) { /* >>> */
    var f = this; g = g.AsyncA();
    return new AsyncA(function(x, k) {
        f.cps(x, function(y) { g.cps(y, k); });
    });
}
AsyncA.prototype.run = function(x) {
    this.cps(x, function(y) {});
    return p;
}
Function.prototype.AsyncA = function() { /* arr */
    var f = this;
    return new AsyncA(function(x, k) { k(f(x)); });
}
```

Prototype constructor

Type identity

CPS composition

"typecheck" and automatically lifts functions to arrows.

CPS invocation

CPS arrow lifting

EventA arrow waits for events using addEventListener…

```
function EventA(eventname) {
    if (!(this instanceof EventA)) return new EventA(eventname);
    this.eventname = eventname;
}
EventA.prototype = new AsyncA(function(target, k) {
    var f = this;
    function handler(event) {
        target.removeEventListener(f.eventname, handler, false);
        k(event);
    }
    target.addEventListener(f.eventname, handler, false);
});
```

…an asynchronous CPS function with a continuation.

Drag-and-Drop with Arrowlets

Handlers written as normal functions like before, but **contain no plumbing**.

```
function setupA(event) {
    /* setup drag-and-drop */
    return event.currentTarget;
}
```

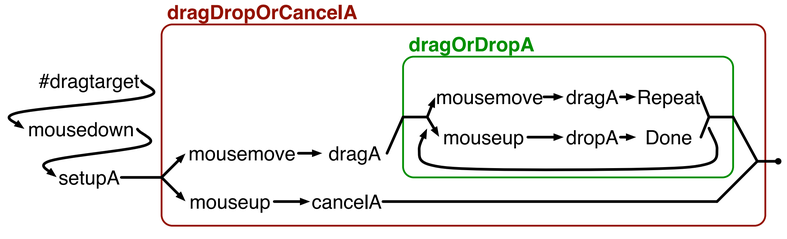
Drag-and-drop can be **composed modularly** using combinators such as next, or, and repeat:

```
var dragOrDropA =
    (   (EventA("mousemove").next(dragA).next(Repeat))
    .or(EventA("mouseup").next(dropA).next(Done))
    ).repeat();

var dragDropOrCancelA =
        (EventA("mousemove").next(dragA).next(dragOrDropA))
    .or(EventA("mouseup").next(cancelA));

var dragAndDropA =
    EventA("mousedown").next(setupA).next(dragDropOrCancelA);

ElementA("dragtarget").next(dragAndDropA).run();
```

or: allows only the first triggered arrow to complete

repeat: loop while Repeat(*x*); stop when Done(*x*)

Arrow composition **looks just like a state diagram**, making it easy to understand.



Parts of drag-and-drop can be **easily reused**:

```
var jigsawA =
    (nextPieceA
        .next( (dragOrDropA.next(repeatIfWrongPlaceA)).repeat() )
    ).repeat();
```