Lecture 23

Implementing Arrowlets

lifting handlers with continuation-passing style

Ras Bodik
Ali and Mangpo

Hack Your Language!
CS164: Introduction to Programming Languages and Compilers, Spring 2013
UC Berkeley

Arrowlets

Programming with Arrowlets

First, read the excellent lecture on Arrowlets

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

Their paper can be found here.

Summary of Arrowlets Combinators

Combinator	**Use**
h = f.next(g)	h(x) is f(g(x))
h = f.bind(g)	h calls f with its input x, and then calls g with a pair (x, f (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. 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
f .AsyncA()	lift function f into an arrow (called automatically by combinators)

Arrows

The design of Arrowlets is based on the concept of arrows introduced in the language Haskell.

Arrows help improve modularity, by separating composition strategies from actual computations. They help mutually isolate program concerns.

Arrows are flexible, because operations can be composed in many different ways.

Arrow

An arrow is a “lifting” class with two methods

A(f)	construct an arrow from fun f lift f to the “world” of arrows
next(a1,a2)	compose arrows a1, a2

It is sufficient for us to understand A and next.

Remaining operators are implemented ~ on top of these.

Our implementation plan

We'll start with a toy implementation, then grow it:

1. Function Arrows
2. CPS Function Arrows
3. Simple Async Event Arrows
4. Full Async Event Arrows

Function Arrows

function calls in direct style

Programs we want to write

This subset of Arrowlets can compose ordinary functions but not events.

We want to support programs like this:

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

Truly faithful to Arrowlets operators, we should write:

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

Function arrows in JS

```
// in JS, each function is an object of class Function.  It has callable methods.
// Here we add methods A and next to all functions in the JS program.
Function.prototype.A = function() {
    return this ;
}
Function.prototype.next = function(g) {
   var f = this; // in a call foo.next(bar), this binds to foo
   g = g.A();    // dyn type check: passes if g a fun, fails if it is, say, an int
   return function(x) { return g(f(x)); } // compose f and g
}
// Now we can run our Arrowlets program:
function add1(x) { return x + 1; }
var add2 = add1.next(add1);
var result = add2(1);   /*returns 3 */
```

call this function

Summary

We did nothing more than compose functions.

The only special trick: add A and next to Function

Exercise: implement arrowlet.run(x)

JavaScript Handler and CPS

In this section

Before we move on to CPS Arrows, we need to understand the rationale for CPS

- it's a "limitation" of our target language (JS events).

JavaScript Event Semantics

This is impossible to do with JS events

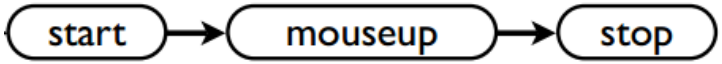
```
function my_handler(event) {
    foo();
    wait_for_event("click", some_target);  // no such
    bar();
}
```

It's impossible because JS event handlers are atomic:

- an event handler can't suspend it's execution
  - it can register new handlers, though
- it always finishes and returns to the "event loop"
  - it's single-threaded execution (one handler at a time)
- this single-threaded no-preemption is a good thing
  - because it prevents races among threads
- we'll see shortly how CPS overcomes this restriction

Overcoming JavaScript Event Semantics

Atomicity of handler execution explains why we cannot translate an Arrowlet program like this



into JS code shown on the previous slide. We need to transform it so:

```
function my_handler(event) {
    foo();
    some_target.addEventListener("click", continuation);
    bar();
}
function continuation(event) { bar(); }
```

Continued

This example also explains why Arrowlets are cool:

- their expressiveness allows us to write this forbidden code (suspending for an event wherever we want)

Continuation-passing-style (CPS) functions

A CPS function f takes

- a normal argument x and
- a continuation k

The continuation is a function

- it executes the "rest of the program"
- its single argument receives f's return value

So, after f evaluates its body, instead of returning to the caller, f calls k, passing to k the return value of f.

- instead of a *return*, f performs a *call*
  - but so what, both are control transfer statements
  - and in both cases, the execution continues in the same spot

CPS example and properties

Direct style:

```
print g(1)+2
```

```
def g(x) { return x/2; }
```

CPS style:

```
g(1, function(v){print v+2})
```

```
def g(x,k) { k(x/2); }
```

In CPS style, all calls are *tail calls*

- the caller need not be suspended at the call site because execution never needs to return to the call site

CPS Function Arrows

function calls in continuation-passing style

CPS Function Arrows in JS

Programs we want to write in this section.

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

Still no events, just functions. But with CPS Arrows in hand, adding support for events will be trivial.

CPS Function Arrows in JS

```
// create a class (prototype) CpsA with a field cps storing the continuation
function CpsA(cps) {    // cps is a fun in CPS style, returns void
    this.cps = cps;     // cps :: (x , k) → ()
}
// add methods A and next to the protptype
CpsA.prototype.CpsA = function() { return this; }
CpsA.prototype.next = function(g) {
    var f = this; g = g.CpsA();
    // compose two CPS functions into a CPS function
    return new CpsA(function(x, k) {     call f with x …
        f.cps(x, function(y) {     … passing it a continuation
            g.cps(y, k);           … which calls g with f's retval
        });                        finally g calls k
    });
}
```

cont for f

CPS Function Arrows in JS

```
// run calls the CPS function, passing it an empty continuation
// this continuation will stop the chain of tail calls, ending the evaluation
CpsA.prototype.run = function(x) {
    this.cps(x, function(y) {});
}

// lift a regular function into a continuation stored in a CpsA object
Function.prototype.CpsA = function() {
    var f = this;
    // wrap the regular function f in CPS function
    return new CpsA(function(x, k) {
        k(f(x));   // note f(x) in k(f(x)) is not tail call
    });
}
```

CPS Function Arrows in JS

```
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))); }
```

Simple Async Event Arrows

let’s add events

Where are we?

Function Arrows:

- compose functions in a wrapper function

CPS Function Arrows:

- compose functions with a continuation;
- CPS functions “never” return, always continue

next, Simple Async Event Arrows

- CPS functions that register their continuations as a handler for the particular event

Example of programs we want to write

```
var count = 0;

// this is a regular handler, nothing special here
function clickTargetA (event) {
    var target = event.currentTarget ;
    target.textContent = "You clicked me! " + ++count;
    return target ;
}

SimpleEventA("click")      //  wait for click event
  .next( clickTargetA )    //  call handler on target passed to next
  .run(document.getElementById("target"));  // select the target
```

Reuse of code is now possible

Same code composition run on a different target

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

Another example (wait for two clicks)

```
SimpleEventA("click")
    .next( clickTargetA )
     // as the next step of the pipeline, wait for click and call handler
    .next( SimpleEventA("click").next( clickTargetA ) )
    .run(document.getElementById("target"))
```

Same as this program (because .next is associative):

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

You may need to understand this JS idiom

```
function SimpleEventA(eventname) {
    if (!(this instanceof SimpleEventA))
        return new SimpleEventA(eventname);
    this.eventname = eventname;
}
```

Explanation:

If the constructor SimpleEventA 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, for example in `.next( SimpleEventA("click") )`

Simple Asynchronous Event Arrows

```
// SimpleEventA is set up as a "subclass" of CpsA

SimpleEventA.prototype = new CpsA(function(target, k) {
    var f = this;
    // essentially, the continuation becomes the handler for the event
    function handler(event) {
        target.removeEventListener(
            f.eventname, handler, false);
        k(event);
    }
    target.addEventListener(f.eventname, handler, false);
});
```

Full Async Event Arrows

a realistic system

Full Asynchronous Event Arrows

Function Arrows:

- compose functions in a wrapper function

CPS Function Arrows:

- compose functions with a continuation;
- CPS functions “never” return, always continue

Simple Async Event Arrows

- CPS functions that register their continuations to handle a particular event

next, Full Async Event Arrows

- We will add combinators needed by drag and drop
- see the paper for details if interested

Full Async Arrows

Want to support multiple arrows in flight

ie, wait for multiple events at once

Only one of the events will happen

So we must be able to cancel one of the two waiting events

Solution: Build AsyncA,

- AsyncA extends CpsA to support tracking progress and cancellation
- When AsyncA is run, it returns a *progress arrow*

Using AsyncA, we build EventA,

Which extends SimpleEventA to track progress and cancellation.

Example

The next example shows how to perform an animation of bubblesort. We want to sleep for 100ms between each iteration. How to do this nicely in JS?

The Arrowlets code on next slide looks almost like a vanilla bubblesort. The key is the repeat(100) operator that calls the body every 100 ms.

Example

```
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 });
```