Lecture 20

Implementing Arrowlets

lifting handlers with continuation-passing style

Ras Bodik
Shaon Barman
Thibaud Hottelier

Hack Your Language!
CS164: Introduction to Programming Languages and Compilers, Spring 2012
UC Berkeley

Summary of L19: Arrowlet Combinators

Combinator	**Use**
f .AsyncA()	lift function f into an arrow (called automatically by combinators)
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

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)	create an arrow from fun f
next(a1,a2)	compose arrows a1, a2

Our implementation steps

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

Function arrows in JS

```
Function.prototype.A = function() {
    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 */
```

Our implementation steps

1. Function Arrows

   compose functions in *direct style:* g(f(x))

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

CPS Function Arrows

We have seen regular function arrows.

- These won't work well for asynchronous composition because the second function is called only after an event, not sequentially after the first one

Example:

- imagine f wants to block and wait for a mousedown:

```
        function handler (e) {  g(f(x))  }
```

In JS, we can't suspend a handler and return to it later when an event happens

- handlers must return; new events invoke new handlers

CPS Function Arrows

We’ll use continuation-passing-style (CPS) functions

A CPS function f takes

a normal argument x and
a continuation k.

The continuation k is called with the result of f.

CPS example

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); }
```

CPS Function Arrows in JS

```
function CpsA(cps) {
    this.cps = cps; /∗ cps :: (x , k) → () ∗/
}
CpsA.prototype.CpsA = function() { /∗ identity ∗/
    return this;
}
CpsA.prototype.next = function(g) {
    var f = this; g = g.CpsA();
    /∗ CPS function composition ∗/
    return new CpsA(function(x, k) {
       f.cps(x, function(y) {
           g.cps(y, k);
       });
    });
}
```

CPS Function Arrows in JS

```
CpsA.prototype.run = function(x) {
    this.cps(x, function(y) {});
}


Function.prototype.CpsA = function() { /* lifting */
    var f = this;
    /* wrap f in CPS function */
    return new CpsA(function(x, k) {
        k(f(x));
    });
}
```

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 Asynchronous Event Arrows

Our sequence:

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 to handle a particular event

Simple Asynchronous Event Arrows

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

JS idiom:

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.

Simple Asynchronous Event Arrows

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

Example

```
var count = 0;

function clickTargetA (event) {
    var target = event.currentTarget ;
    target.textContent = "You clicked me! " + ++count;
    return target ;
}

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

Reuse of code is now possible

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

Another example (wait for two clicks)

```
SimpleEventA("click ")
   .next( clickTargetA )
   .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"));
```

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

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 will build EventA,

Which extends SimpleEventA to track progress and cancellation.

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 });
```