Need a liftA?

asynchronous application programming with **liftA**

bill enright

Why I Started Developing liftA

- Why functional programming matters by John Hughes
    - Glue is good: “higher-order functions […] can contribute significantly to modularity”
    - https://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf
- Turing award paper by John Backus
    - Basically a cry for help for mixing functional techniques with stateful techniques to create robust programs. Written in 1970!
    - https://www.cs.ucf.edu/~dcm/Teaching/COT4810-Fall%202012/Literature/Backus.pdf
- The important question is: how do we glue together things at the application level?

I <3 Arrows

- Arrows are structured computations. State flows *through* them.
    - Monads are a *subset* of arrows. You can build monads out of arrows, but you cannot build all arrows out of monads.
- We transform functions to create arrows. We use an *arrow constructor to create arrows from functions.*
- We compose arrows to create new arrows. We use *arrow combinators to do this*.
- Most importantly: when we compose arrows, we are taking into account both the inputs and the outputs of the arrows.
- So arrows are “about” how state (data) flows through functions and is altered by the computations.
- Now *that* sounds like something Backus was describing.

We focus on *asynchronous* function composition

- Because it's hard.
- Because this is what apps are made of now.
- Because arrows are very well suited to combining asynchronous operations.
- Because I <3 continuations.

Continuations

- Continuations are work to be performed when a function has completed. Usually we *return* from a function and continue working. With asynchronous completions we can't do that. We need the data from the completion and the context to do the next thing, along with the next function to perform.

- Any callback is a continuation. In JavaScript, we rely on closures for the context of the callback. And people wind up in "callback hell."

- Async/Await is a continuation mechanism. We have access to 'local' variables and parameters when we complete. Behind our backs, a closure carries the context forward.

- With arrows, we *carry the data and context along through the continuations.* We take *responsibility* for the heavy lifting.

Combinators and PTC

- Functions of x, returning x
- liftA to turn a function of x into an arrow (receives an input, continues with the output). thenA to combine arrows.
- We use the keyword “return”, in the following code, but it means “continue”
- We use it because ES 6 says to enable Proper Tail Calls, you must “use strict”; and the “return” keyword must precede your tail calls.
- We want to look at synchronous combined functions to get our bearings.
- PTC is supported in Node 6 (not later versions) and latest versions of Safari. **Chrome V8 is not ES 6 compliant.** Will they *ever* be?
- https://kangax.github.io/compat-table/es6/

Create and Combine Arrows liftA and thenA

```
// simple lift - lift f(x)

let liftA = (f) => (x, cont) => {
  return cont(f(x));
};
```

Create and Combine Arrows liftA and thenA

```
// first f, then g, f and g are arrows

let thenA = (f, g) => (x, cont) => {
  return f(x, (x) => {
    return g(x, cont);
  });
};
```

Simple liftA

- we are transforming a function
- take a function f that takes x and returns a value
- return an arrow that will take a value x and a continuation function
- when x and cont are delivered to the arrow, we continue with the result f(x)

```
let liftA = (f) => (x, cont) => {
  return cont(f(x));
};
```

```
function add1(x) {
  return x + 1;
}

let add1A = liftA(add1);

// is equivalent to
function add1A(x, cont) {
  return cont(add1(x));
}
```

Combine arrow with thenA: first f, then g

- take two arrows
- return an arrow that will take a value x and a continuation function
- when x and **cont** are delivered to the new arrow it will
    - run f, providing x and a continuation that will
        - run g, providing f(x) and the original **cont** parameter
- so we continue with the result g(f(x))

```
// first f, then g, f and g are arrows
let thenA = (f, g) => (x, cont) => {
  return f(x, (x) => {
    return g(x, cont);
  });
};
```

```
function mult2(x) {
  return x * 2;
}

let mult2A = liftA(mult2);

let addAndMultA = thenA(add1A, mult2A);

// then addAndMultA is effectively this
function addAndMultA(x, cont) => {
  return add1A(x, (x) => {
    return mult2A(x, cont);
  });
};
```

So What?

- We've just shown that we can transform 'functions of x' into arrows.

- We've shown that we can apply arrows in sequence.

- But why do we care? I could have just written mult2(add1(5)); //!!!!!!!

- Because what we have just seen uses continuations, not returned results, to move results through the combined arrow…and it produces the same result whether you do the work synchronously or *asynchronously.*

Look at Code
async-then-test.js
node async-then-test

arrow-repeat-test.js
synchronous demo lift, then, repeating and left or right
don’t forget about introducing *tuples*

node arrow-repeat-test
node —harmony arrow-repeat-test

We want asynchronous arrows

- We introduce “p” (extra context) which allows us to cancel outstanding asynchronous operations.
  - p.add() adds a canceller function for the outstanding operation. An arrow adds the canceller before waiting for a completion.
  - p.advance() removes a canceller. An arrow calls it when an outstanding operation completes.
  - p. cancelAll() can be used to cancel all arrows ‘in flight’.
- And we introduce an asynchronous lift

```
let liftAsyncA = (f) => (x, cont, p) => {
  let cancelId,
    clear = setTimeout(() => {
      let result = f(x);
      p.advance(cancelId);
      return cont(result, p);
    }, 0);
  cancelId = p.add(() => clearTimeout(clear));
};
```

Look at Code
arrow-asynch-repeat-test.js
demo lift, then, repeating and left or right

Did I Remember to Talk About Tuples?

- We use a convention that the “x” that is passed through arrows is a tuple. You saw it in the previous examples.

- I have implemented this as an Array. Generally, x[0] is data. x[1] is context.

- Sometimes operations are only on the data and we might use firstA(anArrow) so that only x[0] is passed through anArrow. If we do that, x[1] recombines on the other side of the arrow when the result of anArrow is delivered. This lets us pass context “around” an arrow when it is convenient. There is also secondA for when we want to operate on context, not data.

- I use firstA a lot when using ‘arrowized’ callback APIs

A More Fluent Syntax

- Property and function definitions on Function.prototype
- Automatic lifting of arity 1 functions; presume arity 3 functions are arrows
- Boolean: and, or, not, falseError
- Error handling: leftError, promoteError, barrier
- lifta-syntax.js

A More Fluent Syntax

Look at Code
lifta-syntax.js

Complex Examples

- **A web app needing to manage user login and update a user db.**

- app.js - line 299 - passport serialization storing user login data

- app.js - line 316 - putting user onto request with middleware

- userOnReq.js

Complex Examples

- **A node microservice that uses dynamo, fs, phantomjs and s3**
- Converting APIs to arrows
    - dynamo arrows
    - s3 arrows
    - fs arrows
- phantomjs arrow
- errorBarrier - send errors around an arrow
- complex context

Look at Code

lifta-thumbnail

node run thumbnail dev.env.json

Complex Examples

- **A web app needing to run multiple queries in order to render a page.**
- group page
- group admin page

Look at Code
careerninja

app.get(‘/group/:id’) - 1213 - false and true
app.get(‘/activity/:id’) - 1321

node run app.js dev.env.json

Public Repos

- lifta
- lifta-node
- lifta-dynamodb
- lifta-s3
- lifta-fs
- lifta-super

to be continued...