JavaScript

"CS 242 in a nutshell"

Edward Z. Yang

Why JavaScript?

If we want to be concrete, we have to single out a language. CS242 chooses JavaScript as our exemplar. It is certainly not the theoretically most pure language. But its core (the good part) is built off of some of the most important fundamental ideas we want to cover in this course.

- Lingua franca of the Internet
- Illustrates many core concepts of CS242
- Interesting trade-offs and consequences

Old iterations of this course used to use Scheme to fill the same role as JavaScript. However, we've found students are far more familiar with JS than Scheme (for obvious reasons), and the two languages have a lot more in common than you might think...

JavaScript will be the setting in which we talk about these highlighted concepts.

JavaScript

Say more with less!

First-class functions

Pattern matching

Type inference

Type classes

Monads

Continuations

Reliability and Reuse!

Objects & Inheritance

Modules

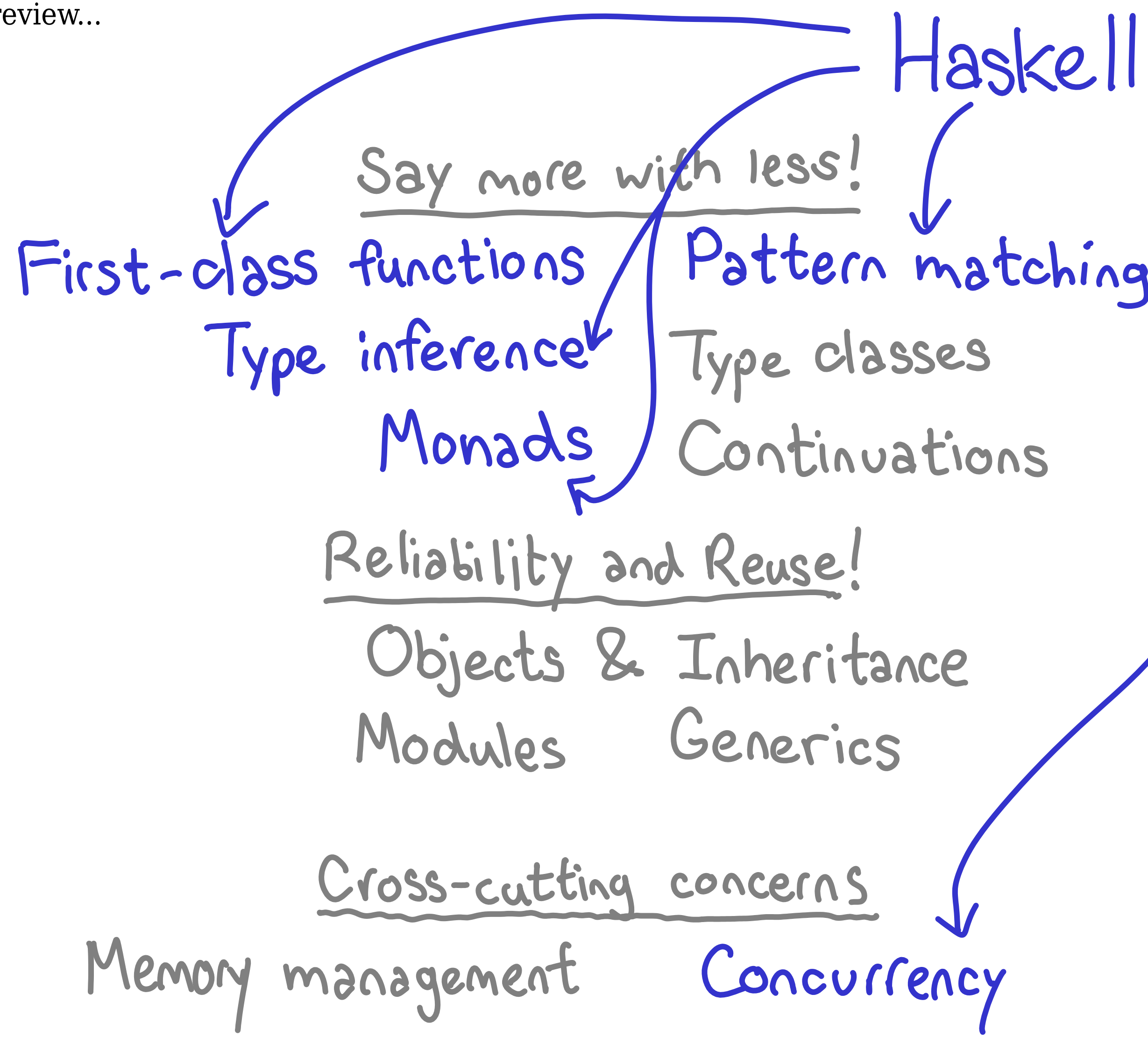
Generics

Cross-cutting concerns

Memory management

Concurrency (ish)

As a preview...


Haskell
Say more with less!
First-class functions
Pattern matching
Type inference
Type classes
Monads
Continuations
Reliability and Reuse!
Objects & Inheritance
Modules
Generics
Cross-cutting concerns
Memory management
Concurrency

Say more with less!

First-class functions

Pattern matching

Type inference

Type classes

Monads

Continuations

Reliability and Reuse!

Objects & Inheritance

Modules

Generics

C++

Java

Cross-cutting concerns

Memory management

Concurrency

May 1995

In the spring of 1995, Netscape had captured more than 90% of the browser market share, rebuffing Microsoft's buyout attempt. Java was the hot new language, and Netscape wanted a new scripting language in their browser to compete. They asked Brendan Eich to do an "HTML scripting language"... but

it had to look like Java!

We need an Internet scripting language for our browser!

Can I use Scheme?

No! It has to look like Java.

marketing deal

Brendan Eich

One week later...

Brendan hacked up a prototype in a week, and they spent the rest of the year embedding it in the browser. Mistakes in the language design were frozen early...

Here's a hacked up JS prototype!

Great! Ship it!

(It took another year to actually embed it in the browser)

Brenden Eich (ICFP 2005)

Design Goals

- Make it easy to copy/paste snippets of code
- Tolerate "minor" errors (missing semicolons)
- Simplified onclick, onmousedown, etc., event handling, inspired by HyperCard
- Pick a few hard-working, powerful primitives
  - First class functions for procedural abstraction
  - Objects everywhere, prototype-based
- Leave all else out!

Functions (based off Lisp/Scheme)

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

JavaScript as a language ecosystem has a lot of moving parts (the web APIs, etc), but JavaScript at its core is only two ideas. First, functions are first class values which can be specified anoymously and passed around as values; second, that objects are simply maps of strings to values (which could be functions.)

Objects (based off Smalltalk/Self)

```
var pt = {x:10,
          move: function(dx) {
                   this.x += dx}}
```

Functions = Full Lexical Closures

First class functions are implemented as full lexical closures: a function definition "captures" variables in its context, which can be used if the function is called later. In this example, the argument x is captured by the inner function (with its value equal to 2), which we can see when we call g later.

```
function curriedAdd(x) {
    return function(y) {return x+y;}
}

g = curriedAdd(2);

console.log(g(3)); // 5
console.log(g(5)); // 7
```

Functions = Full Lexical Closures [Eich]

With lexical closure in an untyped language, you can even do goofy things like define the Y combinator. The Y combinator comes from the lambda calculus, and is a way of implementing recursive functions "purely with functions".

```
function Y(g) {
  return function (f) {return f(f);}(
    function (f) {return g(function(x) {
      return f(f)(x);   }); }); }

var fact = Y(function (fact) {
  return function (n) {
    return (n <= 2) ? n : n * fact(n-1); }});

console.log(fact(5)); // 120
```

First-class functions and closures matter

First-class functions were something that set apart JavaScript from the other competing languages at the time (Tcl, Perl, Python, Java, VBScript). You REALLY want first-class functions if you want to easily script event handlers.

Event handlers in HTML DOM:

easy to use ⇒ first class functions

```
setTimeout(function() { alert("f"); }, 200})
```

Lack of closures hard to work around

(e.g. Java anonymous inner classes)

Closures a mechanism for information hiding

There is also a growing realization in the JavaScript community that closures can be used for information hiding. I remember personally being puzzled when I first programmed JS (after some Java programming) why there was no 'private' keyword. Turns out: you don't need it.

[Eich]

Detour: Lambda calculus

Prelude to Lambda Calculus lecture

Eventually in this course, we are going to talk about the lambda calculus, but here is a taster. The lambda calculus is a notation for representing function definition, using the lambda.

Expressions

$x+y$ $\quad\quad$ $x+2*y+z$

Teaching note: the parentheses here are important! Lambda calculus is unfamiliar syntax and explicitly parenthesizing everything makes it easier for students to see how things group together.

Functions

$\lambda x.(x+y)$ $\quad\quad$ $\lambda z.(x+2*y+z)$

Application

$(\lambda x.(x+y))(3) \Rightarrow 3+y$

$(\lambda z.(x+2*y+z))(5) \Rightarrow x+2*y+5$

Higher-order functions

that's function composition!

Given a function f, return f∘f

$$\lambda f.\,\lambda x.\,f\,(f\,x)$$

So in the lambda calculus, you can easily define higher order functions: that is, functions which return functions.

How does this work?

$$(\lambda f.\,\lambda x.\,f\,(f\,x))\,(\lambda y.\,y+1)$$

Higher-order functions

that's function composition!

Given a function f, return $f \circ f$

$$\lambda f.\ \lambda x.\ f\ (f\ x)$$

How does this work?

$$(\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda y.\ y+1)$$

Higher-order functions

that's function composition!

Given a function f, return f∘f

$$\lambda f.\ \lambda x.\ f\ (f\ x)$$

How does this work?

$$\lambda x.\ (\lambda y.\ y+1)\ ((\lambda y.\ y+1)\ x)$$

Higher-order functions

that's function composition!

Given a function f, return f∘f

$\lambda f.\, \lambda x.\, f\,(f\,x)$

How does this work?

$\lambda x.\, (\lambda y.\, y+1)\,((\lambda y.\, y+1)\,x)$

Higher-order functions

that's function composition!

Given a function f, return f∘f

$\lambda f.\, \lambda x.\, f\,(f\; x)$

How does this work?

$\lambda x.\, (\lambda y.\, y+1)\; (x+1)$

Higher-order functions

that's function composition!

Given a function f, return f∘f

$\lambda f.\, \lambda x.\, f\,(f\; x)$

How does this work?

$\lambda x.\, (x+1)+1$

Higher-order functions in Javascript

Given a function f, return fof

```
function (f) {return function (x) {
    return f(f(x)); }}
```

Objects

[Eich]

Objects are maps of strings to values...

```
var obj = new Object;
obj["prop"] = 42;
obj["0"] = "boo";
```

(obj.prop)

(obj[0])

...so methods are function-valued properties

```
obj.frob = function(n) { this.prop += n; }
obj.frob(6);
```

⇒ obj.prop == 48

Objects (Self influence)

[Eich]

Smalltalk dialect

Function to construct an object

```
function Car(make, model) {
    this.make=make;
    this.model=model; }
myCar = new Car("Porsche", "Boxter");
```

All functions have prototype property

```
Car.prototype.color = "black"
old = new Car("Ford", "T")
myCar.color = "silver"
```

⇒ default

⇒ black

⇒ override color

More about "this"

Prelude to lecture on scoping

Intuitively, this points to the object which has the function as a method

```
var o = {x:10,
         f:function() {return this.x}}

o.f();  ⇒ 10
```

Resolved dynamically when method is called

More goofiness

```
var o = {x:10,
         f:function() {return this.x}}
g = o.f
g()        ⇒ undefined
x = 20       (set global property)
g()        ⇒ 20
```

More and more goofiness

```
var o = {x:10,
         f:function() {
             function g() {
                 return this.x;
             }
             return g();
         }}

x=20;

o.f();  => 20
```

nested function

CS242 GOAL

Establish conceptual framework to say why this is strange!

CS242 language features in JS

- Stack memory management
- Closures
- Exceptions
- Continuations
- Objects
- Garbage collection
- Concurrency (though not parallelism*)

Stack memory management

```
function f(x) {
    var y = 3;
    function g(z) { return y+z; }
    return g(x);
}
var x = 1; var y = 2;
f(x) + y
```

$\Rightarrow$ 6

(You take this for granted...
but it wasn't always this way!)

Closures

(Return fn from fn call)

```
function f(x) {
    var y = x;
    return function(z) { y += z; return y; }
}
var h = f(5);
h(3);
```

⇒ 8

Exceptions

```
try {
    throw "Error2";
} catch (e if e == "Error1") {
    // do something
} catch (e if e == "Error2") {
    // do something else
} catch (e) {
    // catch all
}
```

Continuations "Callback hell"

```
button.onMouseDown = function(event) {
    if (event.button == 1) {
        setTimeout(function() {...}, 200);
    } else {
        setTimeout(function() {...}, 300);
    }
}
```

Objects

- → Dynamic lookup
- → Encapsulation
- → Subtyping
- → Inheritance

clearer in a typed language!

Garbage collection

2 No GC! Memory reclaimed when page changed.

3 Reference counted

4 Mark-and-sweep collector

Concurrency

JavaScript is single threaded

but cooperatively concurrent

The Big Ideas

Say more with less!

First-class functions — Pattern matching
Type inference — Type classes
Monads — Continuations

Reliability and Reuse!

Objects & Inheritance
Modules — Generics

Cross-cutting concerns

Memory management — Concurrency