

CS450

Structure of Higher Level Languages

Lecture 26: SimpleJS

Tiago Cogumbreiro

Press arrow keys   to change slides.

My goal with CS450 is to teach you ...

1. Fundamental concepts behind most programming languages

- functional programming, delayed evaluation, control flow and exceptions, object oriented systems, monads, macros, pattern matching, variable scoping, immutable data structures

2. A framework to describe language concepts

- λ -calculus and formal systems to specify programming languages
- functional programming and monads to implement specifications

3. A methodology to understand complex systems

- (formally) specify and implement each programming language feature separately
- understand a complex system as a combination of smaller simpler systems
- implement and test features independently



JavaScript `__proto__` deprecated!

- Direct access to attribute `__proto__` is discouraged and deprecated!
- However, getting/setting attribute `__proto__` is syntactic sugar for `GetPrototypeOf` and `SetPrototypeOf` in the JavaScript specification.
- We are using `__proto__` mainly because we are following the Essence of JavaScript.
- Prototypes can be updated dynamically due to mutation

JavaScript function objects

We can use field `prototype` to declare the prototype of a given class. We can also use field `prototype` to add methods to an object. Operation `new` assigns `Shape.prototype` to `p1.__proto__`.

```
function Shape(x, y) {  
  this.x = x;  
  this.y = y;  
}  
  
// This way we bind the method once  
Shape.prototype.translate = function (x, y) {  
  this.x += x;  
  this.y += y;  
}  
  
p1 = new Shape(0, 1);  
p1.translate(10, 20);  
console.assert(p1.x == 10);  
console.assert(p1.y == 21);
```



Desugaring object inheritance

```
var Shape = (obj, x, y) => { // Shape's constructor
  obj.x = x;
  obj.y = y;
  return obj
}
Shape.prototype = {} // Shape extends Object
Shape.prototype.translate = function (x, y) { // Also add method translate
  this.x += x;
  this.y += y;
}
p1 = Shape({ "__proto__": Shape.prototype }, 0, 1); // When creating, init prototype
p1.translate(10, 20);
console.assert(p1.x == 10);
console.assert(p1.y == 21);
```

Desugaring class creation

Version 3

```
class Shape {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
  translate(x, y) {  
    this.x += x;  
    this.y += y;  
  }  
}  
p1 = new Shape(0, 1);
```

Version 2

```
function Shape(x, y) {  
  this.x = x;  
  this.y = y;  
}  
Shape.prototype.translate =  
  function (x, y) {  
    this.x += x;  
    this.y += y;  
  }  
p1 = new Shape(0, 1);
```

Version 1

```
Shape = (obj, x, y) => {  
  obj.x = x;  
  obj.y = y;  
  return obj  
}  
Shape.prototype = {}  
Shape.prototype.translate =  
  function (x, y) {  
    this.x += x;  
    this.y += y;  
  }  
p1 = Shape(  
  {"__proto__": Shape.prototype},  
  0, 1);
```

Inheritance desugaring

```
class Rectangle extends Shape {  
  constructor(width, height) {  
    super(0, 0);  
    this.width = width;  
    this.height = height;  
  }  
}  
var r1 = new Rectangle(10, 20);
```

```
function Rectangle(width, height)  
  Shape.call(this, 0, 0);  
  this.width = width;  
  this.height = height;  
}  
Rectangle.prototype =  
  { "__proto__": Shape.prototype };  
var r1 = new Rectangle(10, 20);
```

```
Rectangle = (obj, w, h) => {  
  Shape(obj, 0, 0);  
  obj.width = w;  
  obj.height = h;  
  return obj;  
}  
Rectangle.prototype =  
  { "__proto__": Shape.prototype };  
r1 = Rectangle(  
  { "__proto__": Rectangle.prototype },  
  0, 1);
```

Today we will...

- Revise JavaScript's object system
- Introduce SimpleJS: S-Expression-based syntax and simpler JavaScript rules
- Introduce LambdaJS: λ -calculus + references + immutable objects
- Introduce translation from SimpleJS into LambdaJS

Why are we learning all SimpleJS and LambdaJS?

- You already know λ -calculus with references (heap)
- You already know how objects work (ie, a map with a lookup that work like frames and environments)
- **I want to teach you the fundamentals of JavaScript by building it on top of concepts that you already know!**
- I can introduce another kind of specifying the semantics of a system, by translating it into another system (denotational semantics)



Object prototypes

`A.__proto__ = B` links `A` object to `B`, if a field `f` is not available in `A`, then it is looked up in `B` (which works recursively until finding undefined).

```
a = {"x": 10, "y": 20}
b = {"x": 30, "z": 90, "__proto__": a}
b {x: 30, z: 90, *y: 20}
```

Functions are constructors

If we call a function `A` with `new`, then `A` is called as the constructor of a new object.

```
function C(x, y) { this.x = x; this.y = y }
c = new C(10, 20)
c {x: 10, y: 20}
```

Constructor's prototype

If `A` is a function, then `A.prototype` becomes the `__proto__` of every object created using `A` with `new`.

```
C.prototype = {"foo": true, "bar": 100}
d = new C(10, 20)
d {x: 10, y: 20, *foo: true, *bar: 100}
```



SimpleJS

Introducing SimpleJS

- SimpleJS is just a simplification of JavaScript with fewer corner cases, which is easier to learn.
- SimpleJS was created by your instructor for CS450 (yet close to what you have in The Essence of JavaScript)
- SimpleJS has a formal syntax (below) and also an S-expression syntax (`hw8-util.rkt`)
- Today we will **formally** describe SimpleJS in terms of how we can represent it in LambdaJS (defined in The Essence of JavaScript).

$$\begin{aligned} e ::= & x \mid \text{let } x = e \text{ in } e \mid x.y \mid x.y := e \mid x.y(e \cdots) \\ & \mid \text{function}(x \cdots)\{e\} \mid \text{new } e(e \cdots) \\ & \mid \text{class extends } e \{ \text{constructor}(x \cdots)\{e\} \ m \cdots \} \end{aligned}$$
$$m ::= x(x \cdots)\{e\}$$


Writing Shape in SimpleJS

JavaScript

```
function Shape(x, y) {  
    this.x = x;  
    this.y = y;  
}  
let p = new Shape(10, 20);  
Shape.prototype.translate =  
    function(x, y) {  
        this.x = this.x + x;  
        this.y = this.y + y;  
    };  
p.translate(1,2);  
return p;
```

SimpleJS

```
(let Shape  
  (function (x y)  
    (begin (set! this.x x)  
            (set! this.y y))))  
(let p (new Shape 10 20)  
  (let Shape-proto Shape.prototype  
    (begin  
      (set! Shape-proto.translate  
        (function (x y)  
          (begin  
            (set! this.x (! + this.x x))  
            (set! this.y (! + this.y y))))))  
      (p.translate 1 2)  
      p))))
```

Writing Rectangle in SimpleJS

JavaScript

```
function Rectangle(width, height) {  
  this.x = 0;  
  this.y = 0;  
  this.width = width;  
  this.height = height;  
}  
Rectangle.prototype =  
    Shape.prototype;  
let r1 = new Rectangle(10, 20);  
return r1;
```

SimpleJS

```
(let Rectangle  
  (function (width height)  
    (begin  
      (set! this.x 0)  
      (set! this.y 0)  
      (set! this.width width)  
      (set! this.height height)))  
    (set! Rectangle.prototype Shape.prototype)  
    (let r1 (new Rectangle 10 20)  
      r1))
```

Writing Rectangle in SimpleJS

JavaScript

```
function Rectangle(width, height) {  
  this.x = 0;  
  this.y = 0;  
  this.width = width;  
  this.height = height;  
}  
Rectangle.prototype =  
    Shape.prototype;  
let r1 = new Rectangle(10, 20);  
return r1;
```

SimpleJS

```
(let Rectangle  
  (function (width height)  
    (begin  
      (set! this.x 0)  
      (set! this.y 0)  
      (set! this.width width)  
      (set! this.height height)))  
    (set! Rectangle.prototype Shape.prototype)  
    (let r1 (new Rectangle 10 20)  
      r1)))
```

What are the possible problems of this form of inheritance?

Writing Rectangle in SimpleJS

JavaScript

```
function Rectangle(width, height) {  
  this.x = 0;  
  this.y = 0;  
  this.width = width;  
  this.height = height;  
}  
Rectangle.prototype =  
    Shape.prototype;  
let r1 = new Rectangle(10, 20);  
return r1;
```

SimpleJS

```
(let Rectangle  
  (function (width height)  
    (begin  
      (set! this.x 0)  
      (set! this.y 0)  
      (set! this.width width)  
      (set! this.height height)))  
    (set! Rectangle.prototype Shape.prototype)  
    (let r1 (new Rectangle 10 20)  
      r1)))
```

What are the possible problems of this form of inheritance?

How can we add a new method to Rectangle?



Writing Rectangle in SimpleJS

With the highlighted pattern we can safely mutate `Rectangle.prototype`. This is the same as `Rectangle.prototype = { '__proto__': Shape.prototype }`, but we have no syntax for such a pattern in SimpleJS.

JavaScript

```
function Rectangle(width, height) {  
  this.x = 0;  
  this.y = 0;  
  this.width = width;  
  this.height = height;  
}  
let p = function () {}  
p.prototype = Shape.prototype;  
Rectangle.prototype = new p();  
let r1 = new Rectangle(10, 20);  
return r1;
```

SimpleJS

```
(let Rectangle  
  (function (width height)  
    (begin (set! this.x 0) (set! this.y 0)  
      (set! this.width width)  
      (set! this.height height))))  
(let p (function () 0)  
  (begin  
    (set! p.prototype = Shape.prototype)  
    (set! Rectangle.prototype (new p))  
    (let r1 (new Rectangle 10 20)  
      r1))))
```


LambdaJS

LambdaJS

Think **Racket** without `define`, without macros, with **objects**, and **heap** operations.

Expressions

$$e ::= v \mid x \mid \lambda x. e \mid e(e) \mid \{s: e\} \mid e[e] \mid e[e] \leftarrow e \mid \mathbf{alloc} \ e \mid e := e$$

Concrete LambdaJS S-expression syntax

<i>Formal syntax</i>	<i>S-expression</i>
$\lambda x.e$	<code>(lambda (x) e)</code>
$e_1(e_2)$	<code>(e1 e2)</code>
$\{\text{"foo"} : 1 + 2, \text{"bar"} : x\}$	<code>(object ["foo" (+ 1 2)] ["bar" x])</code>
$o[\text{"foo"}]$	<code>(get-field o "foo")</code>
<code>alloc { }</code>	<code>(alloc (object))</code>
$x := \{ \}$	<code>(set! x (object))</code>
$x := 1; x$	<code>(begin (set! x 1) x)</code>
<code>let x = 10 in x + 4</code>	<code>(let ([x 10]) (+ x 4))</code>

In Racket you can actually allocate a reference with `(box e)`, which is equivalent to LambdaJS `(alloc e)`, and update the contents of that reference with `(set-box! b e)`, which is equivalent to LambdaJS `(set! e)`.

Translating SimpleJS into LambdaJS

1. A SimpleJS object is represented as a reference to an immutable LambdaJS object
2. A SimpleJS function is represented as an object with two fields: (a) a lambda-function that represents the code, a `prototype` field which points to an empty SimpleJS object
3. Create an object with `new` expects a SimpleJS function as argument and must create a new object, initialize its prototype, and call the constructor function (see point 2)
4. Method invocation corresponds to accessing a SimpleJS function and passing the implicit `this` object to it (see 2)

Objectives of the translation

- Explicit `this`
- Functions are not objects: convert `function` into an object+lambda
- Explicit memory manipulation
- No method calls: use function calls



Translating a function

JavaScript

```
function Shape(x, y) {  
  this.x = x;  
  this.y = y;  
};
```

Step 1: only objects and lambdas

```
Shape = {  
  '$code': (obj, x, y) => {  
    obj.x = x;  
    obj.y = y;  
  },  
  'prototype' = {}  
};
```

Translating a function

JavaScript

```
function Shape(x, y) {  
  this.x = x;  
  this.y = y;  
};
```

Step 1: only objects and lambdas

```
Shape = {  
  '$code': (obj, x, y) => {  
    obj.x = x;  
    obj.y = y;  
  },  
  'prototype' = {}  
};
```

Step 2: explicit references

```
Shape = alloc {'$code': (this, x, y) => {  
  this = (deref this)["x"] <- x; // In LambdaJS we have to replace the whole object  
  this = (deref this)["y"] <- y;},  
  'prototype': alloc {};
```

Translating new

JavaScript

```
p1 = new Shape(0, 1);
```

Step 1: only objects and lambdas; no implicit this

```
p1 = {"__proto__": Shape.prototype};  
Shape["$code"] (p1, 0, 1);
```


Translating new

JavaScript

```
p1 = new Shape(0, 1);
```

Step 1: only objects and lambdas; no implicit this

```
p1 = {"__proto__": Shape.prototype};  
Shape["$code"] (p1, 0, 1);
```

Step 2: explicit references

```
p1 = alloc {"__proto__": (deref Shape)["prototype"]}};  
(deref Shape)["$code"] (p1, 0, 1);
```

Translating method invocation

JavaScript

```
p1.translate(10, 20);
```

Step 1: only objects and lambdas; no implicit this

```
m = p1["translate"];    // get object method  
m["$code"] (p1, 10, 20); // get code for method
```

Translating method invocation

JavaScript

```
p1.translate(10, 20);
```

Step 1: only objects and lambdas; no implicit this

```
m = p1["translate"];    // get object method  
m["$code"] (p1, 10, 20); // get code for method
```

Step 2: explicit references

Formally

```
m = (deref p1)["translate"];  
(deref m)["$code"] (p1, 10, 20);
```

SimpleJS

```
(let ([m (get-field (deref p1) "translate")])  
  ((get-field (deref m) "$code") p1 10 20))
```



Translating SimpleJS into LambdaJS

Before

```
Shape.prototype.translate = function(x, y)
    this.x += x; this.y += y;
};
p1 = new Shape(0, 1);
p1.translate(10, 20);
```

After

```
// 1. Function declaration
Shape = alloc {
    "$code": (this, x, y) => { ... },
    "prototype" = alloc {}};
p = (deref Shape)["prototype"];
(deref p)["translate"] = alloc {
    "$code": (this, x, y) => { ... }
    "prototype": alloc {}};
// 2. new
p1 = alloc {"__proto__":
            (deref Shape)["prototype"]};
(deref Shape)["$code"] (p1, 0, 1);
// 3. method call
f = (deref p1)["translate"];
(deref f)["$code"] (p1, 10, 20);
```

Translation function

Translation function

- Field lookup
- Field update
- Function declaration
- The new keyword
- Method call
- Class declaration

Field lookup

Field lookup

$J[x.y] = (\text{get-field } (\text{deref } x) \text{ "y"})$

SimpleJS

```
this.x
```

λ -JS

```
(get-field (deref this) "x")
```


Field update

Field update

In JavaScript, assigning an expression e into a field, returns the evaluation of e . However, in LambdaJS assignment returns the reference being mutated.

$$J[x.y := e] = (\text{let } ([\text{data } J[e]]) (\text{begin} \\ (\text{set! } \underline{x} (\text{set-field } (\text{deref } \underline{x}) \text{"y"} \text{data})) \text{data})))$$

SimpleJS

```
(set! this.x x)
```

λ -JS

```
(let [(data x)]  
  (begin  
    (set! this  
      (update-field (deref this) "x" data))  
    data)))
```

Free variables and bound variables

$$J[x.y := e] = (\text{let } ([\text{data } J[e]]) (\text{begin} \\ (\text{set! } \underline{x} (\text{set-field } (\text{deref } \underline{x}) \text{"y"} \text{data})) \text{data})))$$

SimpleJS

```
(set! data.x 10)
```

λ -JS

```
(let [(data 10)]  
  (begin  
    (set! data  
      (update-field (deref data) "x" data))  
    data)))
```

What happened here?

Free variables and bound variables

$$J[x.y := e] = (\text{let } ([\text{data } J[e]]) (\text{begin} \\ (\text{set! } \underline{x} (\text{set-field } (\text{deref } \underline{x}) \text{"y"} \text{data})) \text{data})))$$

SimpleJS

```
(set! data.x 10)
```

λ -JS

```
(let [(data 10)]  
  (begin  
    (set! data  
      (update-field (deref data) "x" data))  
    data)))
```

What happened here?

1. Variable `data` is used in the generated code
2. We must ensure that `data` is not captured (free) in the generated code!

Quiz

What problem occurs when generating code?

(One sentence is enough.)

Function declaration

Function declaration

Field `prototype` can be accessed by the user, so we declare it as a reference. Field `$code` does not actually exist in JavaScript, so we prefix it with a dollar sign (\$) to visually distinguish artifacts of the translation.

$$J[\text{function}(x \dots) \{e\}] = (\text{alloc } (\text{object} \text{ } [\$code \text{ } (\text{lambda } (\underline{this}, \underline{x} \dots) J[e])][\text{prototype} \text{ } (\text{alloc } (\text{object} \text{ }))]))))$$

SimpleJS

```
(function (x y)
  (begin
    (set! this.x x)
    (set! this.y y)))
```

λ -JS

```
(let ([js-set!
      (lambda (o f d)
        (begin (set! o (update-field (deref o) f d)) d))
      (alloc (object
        ["$code"
         (lambda (this x y)
           (begin (js-set! this "x" x)
                  (js-set! this "y" y))))]
        ["prototype" (alloc (object))]))))
```

The new keyword

The new keyword

$$J[\text{new } e_f(e \dots)] =$$
$$(\text{let } ([\underline{c} \text{ (deref } J[e_f])])$$
$$(\text{let } ([\underline{o} \text{ (alloc (object ["\$proto" (get-field } \underline{c} \text{ "prototype")})])])$$
$$(\text{begin ((get-field } \underline{c} \text{ "\$code") } \underline{o} \text{ } J[e] \dots) \underline{o})))$$

SimpleJS

```
(new Shape 0 1)
```

λ -JS

```
(let [(ctor (deref Shape))  
      (o (alloc (object "$proto" (get-field ctor "prototype"))))]  
  (begin  
    ((get-field ctor "$code") o 0 1)  
    o))
```

Method invocation

Method invocation

$$J[x.y(e \dots)] = ((\text{get-field} (\text{deref} (\text{get-field} (\text{deref } x) \text{"y"})) \text{"$code"}) \underline{x} J[e \dots])$$

SimpleJS

```
(p1.translate 10 20)
```

λ -JS

```
((get-field
  (deref (get-field (deref p1) "translate"))
  "$code")
 p1 10 20)
;; In Racket pseudo code
(define p1:obj (deref p1)) ; 1. get obj from ref
(define translated:m (get-field p1:obj "translate")) ; 2. get field
(define translated:o (deref translated:m)) ; 3. get object from ref
(define translated:f (get-field translated:o "$code")) ; 4. get function
(translated:f p1 10 20) ; 5. call function pass this (p1)
```

Function call

What is the value of `this` when calling a function outside of `new`/method-call?

`this` is initialized to the global variable `window`.

■ We will not be implementing function calls in Homework Assignment 8.

$$J[e_f(e \cdots)] = ((\text{get-field} (\text{deref } J[e_f]) \text{ "$code"}) \text{ window } J[e \cdots])$$

Example 1

```
class Foo {  
  constructor() { this.x = 0; }  
  bar() { this.x++; }  
}  
var foo = new Foo();  
foo["bar"] (); // foo.bar();  
// Caveat: foo.bar() != (foo.bar)()
```

Example 2

```
class Foo {  
  constructor() { this.x = 0; }  
  bar() { this.x++; }  
}  
var foo = new Foo();  
var bar = foo["bar"];  
bar(); // TypeError: this is undefined
```

Class declaration

Class declaration

To allow dynamically dispatching to X 's methods, the first four lines instantiate X without calling its constructor. This way, we can safely mutate the `cls`'s prototype without affecting X and any changes to X are visible to `cls` via lookup.

```
C[[class extends  $X$  {body}]] =  
  let parent = C[[ $X$ ]] in  
  let parent' = function () {} in  
  parent'.prototype := parent.prototype  
  let proto = new parent' in  
  let cls = function ( $x \cdots$ ) { $e_c$ } in  
    cls.prototype := proto;  
  proto.m := function ( $y \cdots$ ) { $e_m$ }; ...  
  cls  
  
where body = constructor( $x \cdots$ ) { $e_c$ } m( $y \cdots$ ) { $e_m$ } ...
```