CS450

Structure of Higher Level Languages

Lecture 25: SimpleJS

Tiago Cogumbreiro
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
   - $\lambda$-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
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__`.

```javascript
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);
```
```javascript
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);
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
}  
var r1 = Rectangle({
    "__proto__": Rectangle.prototype
}, 0, 1);

Inheritance desugaring
Today we will...

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

Why are we learning all SimpleJS and LambdaJS?

- You already know $\lambda$-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).

```javascript
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.

```javascript
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.

```javascript
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 case, 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).

\[
e ::= x \mid \text{let } x = e \text{ in } e \mid x.y \mid x.y := e \mid x.y(e \cdots) \\
    \mid \text{function}(x \cdots)\{e\} \mid \text{new } e(e \cdots) \\
    \mid \text{class extends } e \{\text{constructor}(x \cdots)\{e\} \ m \cdots\}
\]

\[
m ::= x(x \cdots)\{e\}
\]
Writing Shape in SimpleJS

### JavaScript

```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

```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)))
```

---

---

CS450  SimpleJS  Lecture 25  Tiago Cogumbreiro
Writing Rectangle in SimpleJS

JavaScript

```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

```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

```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

```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

```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

```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**

```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**

```javascript
(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

\[ e ::= v \mid x \mid \lambda x.e \mid e(e) \mid \{s : e\} \mid e[e] \mid e[e] \leftarrow e \mid \text{alloc } e \mid e ::= e \]
Concrete LambdaJS S-expression syntax

<table>
<thead>
<tr>
<th>Formal syntax</th>
<th>S-expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda x. e )</td>
<td>(lambda (x) e)</td>
</tr>
<tr>
<td>( e_1(e_2) )</td>
<td>(e1 e2)</td>
</tr>
<tr>
<td>{ &quot;foo&quot; : 1 + 2, &quot;bar&quot; : x }</td>
<td>(object [&quot;foo&quot; (+ 1 2)] [&quot;bar&quot; x])</td>
</tr>
<tr>
<td>( o[&quot;foo&quot;] )</td>
<td>(get-field o &quot;foo&quot;)</td>
</tr>
<tr>
<td>alloc {}</td>
<td>(alloc (object))</td>
</tr>
<tr>
<td>( x := {} )</td>
<td>(set! x (object))</td>
</tr>
<tr>
<td>( x := 1; x )</td>
<td>(begin (set! x 1) x)</td>
</tr>
<tr>
<td>let ( x = 10 ) in ( x + 4 )</td>
<td>(let ([x 10]) (+ x 4))</td>
</tr>
</tbody>
</table>

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

Overview
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

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

Step 1: only objects and lambdas

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

### JavaScript

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

### Step 1: only objects and lambdas

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

### Step 2: explicit references

```javascript
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

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

Step 1: only objects and lambdas; no implicit this

```javascript
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

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

Step 1: only objects and lambdas; no implicit this

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

JavaScript

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

Step 1: only objects and lambdas; no implicit this

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

Step 2: explicit references

Formally

```javascript
m = (deref p1)."translate";
(deref m)."$code"(p1, 10, 20);
```

SimpleJS

```javascript
(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

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

SimpleJS

\[ \text{this.x} \]

\[ \lambda\text{-JS} \]

\[ (\text{get-field (deref this)"x")} \]
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.

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

SimpleJS

\[
(\text{set! } this.x x)
\]

\[\lambda\text{-JS}\]

\[
(\text{let } [(\text{data } x)]
(\text{begin}
(\text{set! } this
(\text{set-field (deref this) } "x" \text{ data}))
\text{data}))
\]
Free variables and bound variables

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

SimpleJS

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

\(\lambda\)-JS

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

What happened here?
Free variables and bound variables

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

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

**SimpleJS**

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

**$\lambda$-JS**

```javascript
(let [[js-set!
    (lambda (o f d)
        (begin (set! o (update-field (deref o) f d)) d))
    (alloc (object
        ["$\text{code}"
            (lambda (this x y)
                (begin (js-set! this "x" x)
                    (js-set! this "y" y))))
        ["\text{prototype}" (\text{alloc} (\text{object})))]))])
```
The new keyword
The `new` keyword

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

SimpleJS

```
(let ([new Shape 0 1])
    (new Shape 0 1))
```

\(\Lambda\)-JS

```
(let ([ctor (\text{deref } Shape)]) \\
    (o (\text{alloc } (\text{object } "$proto" (\text{get-field } ctor "prototype")))]) \\
(\text{begin } ((\text{get-field } ctor "$code") o 0 1) \\
    o))
```
Method invocation
Method invocation

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

SimpleJS

\((p1.\text{translate } 10 \ 20)\)

\( ((\text{get-field} \\
(\text{deref} (\text{get-field} (\text{deref } p1) "translate")) \\
"$\text{code}$) \\
\ p1 \ 10 \ 20) \)

\(;; \text{In Racket pseudo code} \)

\((\text{define } p1:\text{obj} (\text{deref } p1)) ; 1. \text{get obj from ref} \)

\((\text{define } translated:m (\text{get-field } p1:\text{obj} "\text{translate}")) ; 2. \text{get field} \)

\((\text{define } translated:o (\text{deref } translated:m)) ; 3. \text{get object from ref} \)

\((\text{define } translated:f (\text{get-field } translated:o "$\text{code}$")) ; 4. \text{get fun} \)

\((\text{translated:f } p1 \ 10 \ 20) ; 5. \text{call fun pass this (p1)} \)
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.

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

Example 1

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

Example 2

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

$$C[\text{class extends } X \{\text{body}\}] =$$

```javascript
let parent = C[X] in
let parent' = function (){} in
parent'.prototype := parent.prototype
let proto = new parent' in
let cls = function (x⋯){ec} in
cls.prototype := proto;
proto.m := function(y⋯){em};⋯
cls
where body = constructor(x⋯){ec} m(y⋯){em}⋯
```