CS450
Structure of Higher Level Languages
Lecture 38: SimpleJS; translating LambdaJS to 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
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).

Functions are constructors

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

Constructor's prototype

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

What is the name of the paper we are studying?
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).

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

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

```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**

```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

\[
e ::= v \mid x \mid \lambda x.e \mid e(e) \mid \{s: e\} \mid e[e] \mid e[e] \leftarrow e \mid \text{alloc } e \mid e ::= e
\]
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))
```
Before

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

After

```javascript
// 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  \hspace{1cm} \lambda\text{-JS}

\texttt{this.x}  \hspace{1cm} (\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.

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

**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

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

**SimpleJS**

(set! data.x 10)

**λ-JS**

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

What happened here?
Free variables and bound variables

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

SimpleJS

(set! data.x 10)

\( \lambda \)-JS

(let [(data 10)]
  (begin
    (set! data
      (update-field (deref data) "x" 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 $\text{code}$ does not actually exist in JavaScript, so we prefix it with a dollar sign ($) to visually distinguish artifacts of the translation.

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

SimpleJS

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

\lambda-JS

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

The new keyword
The new keyword

\[
\text{SimpleJS} \\
(new \text{ Shape } 0 1)
\]

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

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

SimpleJS

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

\(\lambda\)-JS

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

;; In Racket pseudo code

\((\text{define} \ p1:\text{obj} (\text{deref} p1)) ;; 1. get obj from ref
\quad (\text{define} \ \text{translated}:\text{m} (\text{get-field} p1:\text{obj} "\text{translate}")) ;; 2. get field
\quad (\text{define} \ \text{translated}:\text{o} (\text{deref} \ \text{translated}:\text{m})) ;; 3. get object from ref
\quad (\text{define} \ \text{translated}:\text{f} (\text{get-field} \ \text{translated}:\text{o} "$\text{code}$")) ;; 4. get function
\quad (\text{translated}:\text{f} p1 10 20) ;; 5. call function pass this (p1)\)
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

```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"];
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.

\[
\begin{align*}
\text{C[} & \text{class extends } X \{ \text{body}\}\text{]} = \\
& \text{let parent } = \text{C[}X\text{]} \text{ in} \\
& \text{let parent'} = \text{function }()\{\} \text{ in} \\
& \text{parent'.prototype } := \text{parent.prototype} \\
& \text{let proto } = \text{new parent'} \text{ in} \\
& \text{let cls } = \text{function } (x \cdots)\{e_c\} \text{ in} \\
& \text{cls.prototype } := \text{proto;} \\
& \text{proto.m } := \text{function } (y \cdots)\{e_m\}; \cdots \\
& \text{cls}
\end{align*}
\]

where \( body = \text{constructor} (x \cdots)\{e_c\} \ m(y \cdots)\{e_m\} \cdots \)