CS450

Structure of Higher Level Languages

Lecture 26: Deconstructing JavaScript (part 2)

Tiago Cogumbreiro

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);
```

Field lookup

$$\mathbf{J}[\![x.y]\!] = (\textsf{deref } x)[\texttt{"y"}]$$

SimpleJS

```
this.x
```

λ-JS

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

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.

$$
\begin{aligned}
\mathbf{J}[\![x.y := e]\!] = \; & \mathsf{let\ data} = \mathbf{J}[\![e]\!] \; \mathsf{in} \\
& x := (\mathsf{deref\ obj})[''y''] \leftarrow \mathsf{data}; \\
& \mathsf{data}
\end{aligned}
$$

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

$$\mathbf{J}[\![x.y := e]\!] = \textsf{let data} = \mathbf{J}[\![e]\!]\ \textsf{in x} := (\textsf{deref x})[\text{"}y\text{"}] \leftarrow \textsf{data; data}$$

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

$$\mathbf{J}[\![x.y := e]\!] = \textsf{let data} = \mathbf{J}[\![e]\!] \textsf{ in x} := (\textsf{deref x})[\text{"}y\text{"}] \leftarrow \textsf{data; data}$$

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!
   Recall Lecture 11 where we introduced how to compute free variables.

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 `$code` does not actually exist in JavaScript, so we prefix it with a dollar sign (`$`) to visually distinguish artifacts of the translation.

$$
\mathrm{J}[\![\texttt{function}(x\cdots)\ \{e\}]\!] = \\
\mathsf{alloc}\ \{\texttt{"\$code"} : \lambda(\mathsf{this}, x\cdots).\mathrm{J}[\![e]\!], \texttt{"prototype"} : \mathsf{alloc}\ \{\}\}
$$

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

$$\mathbf{J}[\![\texttt{new}\ e_f(e\cdots)]\!] = $$
$$\text{let ctor} = \text{deref}\ \mathbf{J}[\![e_f]\!]\ \text{in}$$
$$\text{let obj} = \text{alloc}\ \{\texttt{"\$proto"} : \text{ctor}[\texttt{"prototype"}]\}\ \text{in}$$
$$\text{ctor}[\texttt{"\$code"}](\text{obj}, \mathbf{J}[\![e]\!]\cdots);$$
$$\text{obj}$$

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

$$\mathbf{J}[\![x.y(e\cdots)]\!] = (\textbf{deref}\ (\textbf{deref}\ x)[\texttt{"y"}])[\texttt{"\$code"}](x, \mathbf{J}[\![e\cdots]\!])$$

SimpleJS

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

λ-JS

```
((get-field
  (deref (get-field (deref p1) "translate"))
  "$code")
  p1 10 20)
```

Function call

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

$$\mathbf{J}[\![e_f(e\cdots)]\!] = $$
$$\text{let obj} = \mathbf{J}[\![e_o]\!] \text{ in}$$
$$(\text{deref obj})[\texttt{"\$code"}](\text{window}, \mathbf{J}[\![e\cdots]\!])$$

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

> 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.

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

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

The Essence of JavaScript

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi

Brown University

Abstract. We reduce JavaScript to a core calculus structured as a small-step operational semantics. We present several peculiarities of the language and show that our calculus models them. We explicate the desugaring process that turns JavaScript programs into ones in the core. We demonstrate faithfulness to JavaScript using real-world test suites. Finally, we illustrate utility by defining a security property, implementing it as a type system on the core, and extending it to the full language.

cs.brown.edu/research/plt/dl/jssem/v1/gsk-essence-javascript-r6.pdf

The Essence of JavaScript

> **Abstract.** We reduce JavaScript to a core calculus structured as a small-step operational semantics. We present several peculiarities of the language and show that our calculus models them. We explicate the desugaring process that turns JavaScript programs into ones in the core. We demonstrate faithfulness to JavaScript using real-world test suites. Finally, we illustrate utility by defining a security property, implementing it as a type system on the core, and extending it to the full language.

The Essence of JavaScript

> **Abstract.** We reduce JavaScript to a core calculus structured as a small-step operational semantics. We present several peculiarities of the language and show that our calculus models them. We explicate the desugaring process that turns JavaScript programs into ones in the core. We demonstrate faithfulness to JavaScript using real-world test suites. Finally, we illustrate utility by defining a security property, implementing it as a type system on the core, and extending it to the full language.

1. Introduce LambdaJS

The Essence of JavaScript

> **Abstract.** We reduce JavaScript to a core calculus structured as a small-step operational semantics. We present several peculiarities of the language and show that our calculus models them. We explicate the desugaring process that turns JavaScript programs into ones in the core. We demonstrate faithfulness to JavaScript using real-world test suites. Finally, we illustrate utility by defining a security property, implementing it as a type system on the core, and extending it to the full language.

1. Introduce LambdaJS
2. Present translation from JavaScript to LambdaJS

The Essence of JavaScript

> **Abstract.** We reduce JavaScript to a core calculus structured as a small-step operational semantics. We present several peculiarities of the language and show that our calculus models them. We explicate the desugaring process that turns JavaScript programs into ones in the core. We demonstrate faithfulness to JavaScript using real-world test suites. Finally, we illustrate utility by defining a security property, implementing it as a type system on the core, and extending it to the full language.

1. Introduce LambdaJS
2. Present translation from JavaScript to LambdaJS
3. Demonstrate faithfulness with test suites

The Essence of JavaScript

> **Abstract.** We reduce JavaScript to a core calculus structured as a small-step operational semantics. We present several peculiarities of the language and show that our calculus models them. We explicate the desugaring process that turns JavaScript programs into ones in the core. We demonstrate faithfulness to JavaScript using real-world test suites. Finally, we illustrate utility by defining a security property, implementing it as a type system on the core, and extending it to the full language.

1. Introduce LambdaJS
2. Present translation from JavaScript to LambdaJS
3. Demonstrate faithfulness with test suites
4. Illustrate utility of LambdaJS with a language extension.

$$
\begin{aligned}
c &= num \mid str \mid bool \mid \textsf{\textbf{undefined}} \mid \textsf{\textbf{null}} \\
v &= c \mid \textsf{\textbf{func}}\texttt{(}x\cdots\texttt{)}\ \texttt{\{}\ \textsf{\textbf{return}}\ e\ \texttt{\}} \mid \texttt{\{}\ str\texttt{:}v\cdots\ \texttt{\}} \\
e &= x \mid v \mid \textsf{\textbf{let}}\ \texttt{(}x\ \texttt{=}\ e\texttt{)}\ e \mid e\texttt{(}e\cdots\texttt{)} \mid e\texttt{[}e\texttt{]} \mid e\texttt{[}e\texttt{]}\ \texttt{=}\ e \mid \textsf{\textbf{delete}}\ e\texttt{[}e\texttt{]} \\
E &= \bullet \mid \textsf{\textbf{let}}\ \texttt{(}x\ \texttt{=}\ E\texttt{)}\ e \mid E\texttt{(}e\cdots\texttt{)} \mid v\texttt{(}v\cdots\ E\texttt{,}\ e\cdots\texttt{)} \\
&\quad \mid \texttt{\{}str\texttt{:}\ v\cdots\ str\texttt{:}E\texttt{,}\ str\texttt{:}e\cdots\ \texttt{\}} \mid E\texttt{[}e\texttt{]} \mid v\texttt{[}E\texttt{]} \mid E\texttt{[}e\texttt{]}\ \texttt{=}\ e \mid v\texttt{[}E\texttt{]}\ \texttt{=}\ e \\
&\quad \mid v\texttt{[}v\texttt{]}\ \texttt{=}\ E \mid \textsf{\textbf{delete}}\ E\texttt{[}e\texttt{]} \mid \textsf{\textbf{delete}}\ v\texttt{[}E\texttt{]}
\end{aligned}
$$

$$
\textsf{\textbf{let}}\ \texttt{(}x\ \texttt{=}\ v\texttt{)}\ e \hookrightarrow e[x/v] \qquad \text{(E-Let)}
$$

$$
\texttt{(}\textsf{\textbf{func}}\texttt{(}x_1\cdots x_n\texttt{)}\ \texttt{\{}\ \textsf{\textbf{return}}\ e\ \texttt{\})(}v_1\cdots v_n\texttt{)} \hookrightarrow e[x_1/v_1\cdots x_n/v_n] \qquad \text{(E-App)}
$$

$$
\texttt{\{}\ \cdots str\texttt{:}\ v\cdots\ \texttt{\}[}str\texttt{]} \hookrightarrow \text{v} \qquad \text{(E-GetField)}
$$

$$
\frac{str_x \notin (str_1\cdots str_n)}{\texttt{\{}\ str_1\texttt{:}\ v_1\ \cdots\ str_n\texttt{:}\ v_n\ \texttt{\}}\ \texttt{[}str_x\texttt{]} \hookrightarrow \textsf{\textbf{undefined}}} \qquad \text{(E-GetField-NotFound)}
$$

$$
\begin{array}{c}
\texttt{\{}\ str_1\texttt{:}\ v_1\cdots\ str_i\texttt{:}\ v_i\ \cdots str_n\texttt{:}\ v_n\ \texttt{\}}\ \texttt{[}str_i\texttt{]}\ \texttt{=}\ \texttt{v} \\
\hookrightarrow \texttt{\{}\ str_1\texttt{:}\ v_1\cdots\ str_i\texttt{:}\ v\ \cdots str_n\texttt{:}\ v_n\ \texttt{\}}
\end{array} \qquad \text{(E-UpdateField)}
$$

$$
\frac{str_x \notin (str_1\cdots)}{\texttt{\{}\ str_1\texttt{:}\ v_1\cdots\ \texttt{\}}\ \texttt{[}str_x\texttt{]}\ \texttt{=}\ v_x \hookrightarrow \texttt{\{}\ str_x\texttt{:}\ v_x\texttt{,}\ str_1\texttt{:}\ v_1\cdots\ \texttt{\}}} \qquad \text{(E-CreateField)}
$$

$$
\begin{array}{c}
\textsf{\textbf{delete}}\ \texttt{\{}\ str_1\texttt{:}\ v_1\cdots\ str_x\texttt{:}\ v_x\ \cdots str_n\texttt{:}\ v_n\ \texttt{\}}\ \texttt{[}str_x\texttt{]} \\
\hookrightarrow \texttt{\{}\ str_1\texttt{:}\ v_1\cdots str_n\texttt{:}\ v_n\ \texttt{\}}
\end{array} \qquad \text{(E-DeleteField)}
$$

$$
\frac{str_x \notin (str_1\cdots)}{\textsf{\textbf{delete}}\ \texttt{\{}\ str_1\texttt{:}\ v_1\cdots\ \texttt{\}}\ \texttt{[}str_x\texttt{]} \hookrightarrow \texttt{\{}\ str_1\texttt{:}\ v_1\cdots\ \texttt{\}}} \qquad \text{(E-DeleteField-NotFound)}
$$

The Essence of JavaScript

- LambdaJS (implemented in Racket)
- Translator from JS to λ-JS (Haskell)
- Coq formal semantics
- OCaml interpreter and translator (ECMAScript 5)
- Code: `github.com/brownplt/LambdaJS`
- Code: `github.com/brownplt/LambdaS5`

Desugar code review: field lookup

```haskell
expr :: Env → Expression SourcePos → ExprPos
expr env e = case e of
  -- ...
  ThisRef a → EId a "this"
  VarRef _ (Id _ s) → eVarRef env s
  DotRef a1 e (Id a2 s) → EGetField a1 (EDeref nopos $ toObject $ expr env e)
                          (EString a2 s)
  BracketRef a e1 e2 →
      EGetField a (EDeref nopos $ toObject $ expr env e1) (toString $ expr env e2)
NewExpr _ eConstr es → eNew (expr env eConstr) (map (expr env) es)
```

$$\mathbf{J}[\![x.y]\!] = (\textsf{deref}\ x)[\texttt{"y"}]$$

Source

Desugar code review: calls/invocations

```haskell
--desugar applying an object
applyObj :: ExprPos → ExprPos → [ExprPos] → ExprPos
applyObj efuncobj ethis es = ELet1 nopos efuncobj $ \x →
    EApp
      (label efuncobj)
      (EGetField
        (label ethis)
        (EDeref nopos $ EId nopos x)
        (EString nopos "$code"))
      [ethis, args x]
  where args x = ERef nopos $ ERef nopos $ eArgumentsObj es (EId nopos x)
```

$$\mathrm{J}[\![x.y(e\cdots)]\!] = (\textbf{deref}\ (\textbf{deref}\ x)[\texttt{"y"}])[\texttt{"\$code"}](x, \mathrm{J}[\![e\cdots]\!])$$

Source

AST Example 1

JavaScript

```
(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))))
```

Demo...

AST Example 2

```
(let [(ctor (deref Proto))
      (o (alloc (object "$proto" (get-field ctor "prototype"))))
      (y1 0) (y2 1)]
  (begin
    ((get-field ctor "$code") o y1 y2)
    o))
```

Demo...

LambdaJS: Formal specification

LambdaJS: Object semantics

$$\frac{\forall s. O(s) = \texttt{undef}}{\{\} \Downarrow_E O} \; \texttt{E-empty}$$

$$\frac{e_o \Downarrow_E O \qquad e_f \Downarrow_E s}{e_o[e_f] \Downarrow_E \mathrm{lookup}(O, s)} \; (\texttt{E-get})$$

$$\frac{e_o \Downarrow_E O \qquad e_f \Downarrow_E s \qquad e_v \Downarrow_E v}{e_o[e_f] = e_v \Downarrow_E O[s \mapsto v]} \; (\texttt{E-set})$$

LambdaJS: Heap operations

$$\frac{e \Downarrow v \qquad l \leftarrow \texttt{alloc}\ v}{\texttt{alloc}\ e \Downarrow l}$$

$$\frac{e \Downarrow l}{\texttt{deref}\ e \Downarrow \texttt{get}\ l}$$

$$\frac{e_1 \Downarrow_E l \qquad e_2 \Downarrow_E v \qquad \texttt{put}\ l\ v}{e_1 := e_2 \Downarrow l}$$

Lookup with references

$$\frac{O = \mathtt{get}\ l \qquad s \in O}{\mathrm{lookup}(l, s) = O(s)} \qquad \frac{O = \mathtt{get}\ l \qquad s \notin O \qquad \mathtt{'\$proto'} \notin O}{\mathrm{lookup}(l, s) = \mathtt{undef}}$$

$$\frac{O = \mathtt{get}\ l \qquad s \notin O \qquad O(\mathtt{'\$proto'}) = l'}{\mathrm{lookup}(l, s) = \mathrm{lookup}(l', s)}$$

Definition

Field membership: Let $s \notin O$ if, and only, $O(s) = \mathtt{undef}$, otherwise we say that $s \in O$.