CS450

Structure of Higher Level Languages

Lecture 39: The Essence of JavaScript; Homework FAQ

Tiago Cogumbreiro

Today we will learn...

- A deeper look into "The Essence of JavaScript" paper
- Address some frequently asked questions about HW7 and HW8

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

UMASS BOSTON

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

UMASS BOSTON

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

UMASS BOSTON

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

UMASS BOSTON

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

UMASS BOSTON

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.

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

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

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

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

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

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

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

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

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

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)
```

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

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)
```

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

Source

LambdaJS: Formal specification

LambdaJS: Object semantics

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

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

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

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

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

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

Definition

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

Homework assignment questions

HW7 question

- What is the major difference between an eff and an eff-op?

HW7 question

> What is the major difference between an eff and an eff-op?

Answer

Let us look at `hw7-util.rkt`:

```
(struct eff (state result) #:transparent)
(struct eff-op (func))
```

- `eff` is the return of effectful operations
- `eff-op` a structure that holds an effectful operation, takes a ***state*** (eg, a heap) and produces an `eff`

Examples of effectful operations `eff-op`: `eff-bind`, `eff-pure`, `env-put`, `env-get`, `env-push`

HW7 question

- How do I test for if? How do I know if the term is curried?

$$\frac{e_c \Downarrow_E \texttt{\#f} \quad \blacktriangleright \quad e_f \Downarrow v_f}{(((\texttt{if}\ e_c)\ e_t)\ e_f) \Downarrow_E v_f}\ \texttt{(E-if-f)}$$

$$\frac{e_c \Downarrow_E v \qquad v \neq \texttt{\#f} \quad \blacktriangleright \quad e_t \Downarrow v_t}{(((\texttt{if}\ e_c)\ e_t)\ e_f) \Downarrow_E v_t}\ \texttt{(E-if-t)}$$

HW7 question

> How do I test for if? How do I know if the term is curried?

$$\frac{e_c \Downarrow_E \texttt{\#f} \quad \blacktriangleright \quad e_f \Downarrow v_f}{(((\texttt{if}\ e_c)\ e_t)\ e_f) \Downarrow_E v_f}\ \texttt{(E-if-f)} \qquad \frac{e_c \Downarrow_E v \qquad v \neq \texttt{\#f} \quad \blacktriangleright \quad e_t \Downarrow v_t}{(((\texttt{if}\ e_c)\ e_t)\ e_f) \Downarrow_E v_t}\ \texttt{(E-if-t)}$$

Answer

1. Use pattern matching with nested a pattern before the branch for apply.
2. Terms being evaluated are **always** curried.

Match examples

```
(match exp
  [(? s:value? x) x]
  [x #:when (s:value? x) x]
  ; 'x ← would match
  [(s:variable 'x)
   'pattern1] ; declares nothing
  [(s:lambda (list (s:variable _)) _)
   'pattern2] ; declares nothing
  ; (closure E0 (lambda (y) z)) ← would match
  [(s:closure _ (s:lambda (list x) eb))
   ; declares x (which we know is an s:variable) and eb (which we know is an s:expression)
   'pattern3 ]
   ; '((x 1) y) ← would match
  [(s:apply (s:apply (s:variable 'x) (list (s:number 1))) (list (s:variable foo)))
   'pattern4])
```

HW8 question

What does $(\texttt{lambda}\ (\underline{this}\ \underline{x} \cdots)\ J[\![e]\!])$ mean?

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

HW8 question

> What does $(\texttt{lambda}\ (\underline{this}\ \underline{x} \cdots)\ J[\![e]\!])$ mean?

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

Answer

Generate a lambda, whose

1. ***parameters*** are **this**, x $\cdots$, so translate the original parameters x, $\cdots$ and add a variable `this`
2. ***body*** is the translation of e

HW8 question

- What is `js-set!`?

```
(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))])))
```

HW8 question

UMASS BOSTON

What is `js-set!`?

```
(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))])))
```

Answer

- The generated code did not fit the slide, think of it as the translation of `(set! o.f a)`. I have highlighted in yellow the code being generated.

HW8 question

- What is the difference between `$proto` and `prototype`?

HW8 question

> What is the difference between `$proto` and `prototype`?

Answer

1. `$proto` is a field used for looking up the super object (the parent); works on any object. In JavaScript this is `__proto__`, in LambdaJS this is `$proto`.
2. `prototype` is the field of every `function`, used by `new` to initialize the `$proto` field of created objects

```
function A () {this.a = 1;}
A.prototype = {"__proto__": {"b": 10, "c": 10, "a": 10}, "b": 20}
a = new A; // {a: 1, *b: 20, *c: 10}
```

Functional parallelism

Parallelism with asynchronous evaluation

UMASS BOSTON

The idea is similar to `delay/force`

1. `(future t)` evaluates a thunk `t` in another task, possibly by another processor
2. Calling `(future t)` returns a ***future value*** `f`, a place holder to a parallel computation
3. One can await the termination of the parallel task with `(touch f)`, which blocks the current task until the task evaluating the future thunk terminates. Consecutive `(touch f)` are nonblocking.

```
(define f (future (thunk (sleep 2) 99))) ;; Spawns a task T1
(assert-equals? (touch f) 99)    ;; Blocks until T1 terminates and returns 99
(touch f)                        ;; We know that T1 has terminated
```

A parallel fold

```
(define (par-reduce f init v lo hi)
  (if (< (- lo hi) threshold)
    ;; Base case, call sequential version
    (foldl f init (vector-view v lo hi))
    ;; Otherwise, divide array into two and spawn another task
    (let* ([mid (floor (+ (/ lo 2) (/ hi 2)))]
           [l (future (thunk (par-reduce f init v lo mid)))]
           [r (par-reduce f init v mid hi)])
      (f (touch l) r))))
```

Map-reduce example

```
(f
  (f
    (foldl f 0 [ 0 ...  64])  ; Task 1
    (foldl f 0 [64 ... 128])) ; Task 2
  (foldl f 0 [ 128 ... 192])) ; Task 3
```

Example of parallel reduce

```
(define (f x y)
  (/ (- (+ (- (* x 2) y y 25) x y 56) x 36) 2))
(define (do-par l)
  (par-reduce f 0 (list->vector l)))
(define (do-seq l)
  (foldl f 0 l))
```

Example of parallel reduce

```
(define (f x y)
  (/ (- (+ (- (* x 2) y y 25) x y 56) x 36) 2))
(define (do-par l)
  (par-reduce f 0 (list->vector l)))
(define (do-seq l)
  (foldl f 0 l))
```

Output

```
Processing a list of size: 10000

 * Serial version *
Throughoutput: 25 elems/ms
Mean: 402.03±9.89ms

 * Parallel version *
Throughoutput: 25 elems/ms
Mean: 392.76±13.2ms
```

Parallelism in Racket

Let us try Clojure!

Parallel reduce

```
(defn do-reduce [f l treshold]
  (proxy [RecursiveTask] []
    (compute []
      (if (≤ (count l) treshold)
        ;; if the vector is small enough,
        ;; we just reduce over them
        (reduce f 0 l)
        ;; otherwise, we split the vector roughly in two
        ;; and recursively run two more tasks
        (let [half (quot (count l) 2)
              f1 (do-reduce f (subvec l 0 half) treshold)
              f2 (do-reduce f (subvec l half) treshold)]
          ;; do half the work in a new thread
          (.fork f2)
          ;; do the other half in this thread and combine
          (f (.compute f1) (.join f2)))))))
```

Demo

- Clojure 1.10
- OpenJDK 1.8.0_191
- Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
- 4 cores
- list with 1,000,000 elements

Demo

- Clojure 1.10
- OpenJDK 1.8.0_191
- Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
- 4 cores
- list with 1,000,000 elements

```
Serial version
"Elapsed time: 2769.94558 msecs"
Parallel version
"Elapsed time: 755.341055 msecs"
```

3.7× Increase!

Demo 2

- Let us vary the parameter being used...

Demo 2

UMASS BOSTON

Let us vary the parameter being used...

```
Serial version
"Elapsed time: 101.96357 msecs"
Parallel version
"Elapsed time: 219.819163 msecs"
```

2.0× slower!

Parallel overhead is significant!

Demo 3

- Let us vary the size of the data being used: **100,000 elements** rather than 1,000,000

Demo 3

UMASS BOSTON

- Let us vary the size of the data being used: **100,000 elements** rather than 1,000,000

```
Serial version
"Elapsed time: 179.724932 msecs"
Parallel version
"Elapsed time: 182.837934 msecs"
```

Data size is also significant!

Thank you!