Modular Instrumentation of Interpreters in JavaScript

Florent Marchand de Kerchove, Jacques Noyé, Mario Südholt
ASCOLA, Mines Nantes

FOAL’15 @ Fort Collins, March 16, 2015
The Instrumentation Problem
The web

Web browsers

JavaScript engines

SpiderMonkey (C++)
V8 (C++)

Other JavaScript interpreters:

- Rhino (Java)
- Narcissus (JavaScript)
Narcissus

Metacircular JavaScript interpreter by Mozilla

Breeding ground for testing new language features

Used by Austin and Flanagan to implement the faceted evaluation analysis
Faceted evaluation is a dynamic information flow analysis.

- Each value has two facets.
- The private value is visible only to a set of principals.
- A “program counter” keeps track of the current set of principals in branches.
Faceted evaluation in vivo
Faceted evaluation in vivo

- Standalone concern, scattered code
- Any part of the interpreter liable to change
- Program counter is threaded through calls
- Difficult to compose analyses
The Instrumentation Problem

Four requirements for modular instrumentation:

- **Modularity**: interpreter and analysis as modules
- **Intercession**: can add or alter any part of the interpreter
- **Local state**: can thread state local to an analysis
- **Pluggability**: can toggle the analysis dynamically
Building an interpreter with modules
A language of arithmetic expressions

```
Num
Plus
```

```
eval
show
```

6
Ingredients

- Dictionary objects as modules
- Delegation via prototypes
- Name shadowing
- Closures
- Late binding

Same client code, different results
The num data variant

var num = {
    new(n) { return {__proto__: this, n} },
    eval() { return this.n }
}

var e1 = num.new(3)
e1.eval() //: 3
The num data variant

```javascript
var num = {
  new(n) { return {__proto__: this, n} },
  eval() { return this.n }
}

var e1 = num.new(3)
e1.eval() //: 3
```
Adding a data variant

```javascript
var plus = {
  new(l, r) { return {__proto__: this, l, r,} },
  eval() { return this.l.eval() + this.r.eval() }
}

var e2 = plus.new(num.new(1), num.new(2))
```
num.show = function() { return this.n.toString() }
plus.show = function() { ... }
e1.show() //: “3”
Adding an operation as a module

```javascript
var show = base => {
  var num = {__proto__: base.num,
              show() { return this.n.toString() }}
  var plus = {...
             return {num, plus} }

var s = show({num, plus})
```

Unsafe mixing of data variants

```
s.plus.new(num.new(1), s.num.new(2)).show()
```

```
//: TypeError: this.l.show is not a function
```
A use-case for `with`

```javascript
with(show({num, plus})) {
  plus.new(num.new(1), num.new(2)).show()
}
```

Inside `with`:
- `num`
- `show`
- `eval`

Outside `with`:
- `num`
- `eval`

14
Instrumented language

eval

Num

Plus

double

state
Modifying operations

```javascript
var double = num_orig => {
  var num = {__proto__: num_orig,
    eval() { return super.eval() * 2 }}
  return {num} }

with(double(num)) {
  plus.new(num.new(1), num.new(2)).eval() }
//: 6
```
with(double(num)) {
  with(double(num)) {
    plus.new(num.new(1), num.new(2)).eval() 
  }
}

//: 12

num

- eval: super.eval() * 2

- new
  - eval: this.n

- eval: super.eval() * 2
var state = (base, pc = 0) => {
  var num = {__proto__: base.num,
              eval() { pc++; return super.eval() }}
  var plus = {...}
  var getPC = () => pc
  return {num, plus, getPC} }

num

eval: pc++

new eval
with (state({num, plus})) {
  getPC() //: 0
  plus.new(num.new(1), num.new(2)).eval() //: 3
  getPC() //: 3
}
All combined

```javascript
with (state({num, plus})){
  with (double(num)) {
    with (show({num, plus})){
      getPC() ///: 0
      let n = plus.new(num.new(1), num.new(2))
      n.eval() ///: 6
      getPC() ///: 3
      n.show() ///: “1 + 2”
    }
  }
}
```

Diagram:
```
 n.l ....... n  -->  show  -->  eval  -->  eval  -->  new eval
```
Wrap-up
A simple modular interpreter

The instrumentation problem:

- Modularity
- Intercession
- Local state
- Pluggability

Simple language ingredients:

- Delegation
- Late binding
- Dictionaries as modules
- Closures