Formal, Executable Semantics of Web Languages: JavaScript and PHP

Sergio Maffeis
Imperial College London

In collaboration with:
J. Mitchell (Stanford), A. Taly (Google),
K. Bhargavan, M. Bodin, A. Charugeraud, A. Delignat-Lavaud, A. Schmitt (INRIA),
D. Filaretti, P. Gardner, D. Naudziuniene, G. Smith, S. Yuwen (Imperial)

PiP’14, San Diego
A Personal Perspective

• Goal: “language based web security”
  – 1st step: build formal models (this talk)
  – Next, analyze security properties

• Based on:
  – JSSec: small-step operational semantics of ES3
  – JSCert: Coq semantics and interpreter of ES5
  – KPHP: formal executable semantics of PHP in K

• (Not a literature survey, see my papers for references)
Given a language $L$ and an interpreter $X$, define a semantics $S$ such that for all $p$ in $L$, $S(p) \approx \approx X(p)$.

Real world: here’s an interpreter $X$. Good luck!
- Define a semantics $S$ such that $S(p) == X(p)$ for as many $p$ as possible.

Approach
- “Observe” a piece of syntax (experiments & documentation).
- Model behaviour using building blocks of meta-language.
- Formulate predictions to validate model (testing).
Handling Pre-Existing Systems Complexity
JavaScript and PHP

• Born as small languages
  – JavaScript: sanitize input of HTML forms
  – PHP: Personal Home Page Tools for tracking home page visits

• Now achieved world domination
  – All web pages, most servers
  – Top of Github/StackOverflow popularity
    • Chart from [http://langpop.corger.nl](http://langpop.corger.nl)

• Picked up lots of complexity along the way
JavaScript and PHP

- Critical points of failure for web security
  - Attacks come from obscure, difficult corner cases
  - Do not leave out tricky or inelegant constructs

```html
<a href="#" onclick="b()"> Test B (Safari, Opera and Chrome)</a>
<script>
function b(){
    try {throw (function(){return this});}
    catch (get_scope){get_scope().ref=function(x){return x};
    this.alert("Hacked!")}
</script>
```

- OK to look at conservative subsets
  - But beware of unsound simplifications

```php
$arr = array("one", "two", "three");
foreach ($arr as $value) {
    echo "Value: $value<br />\n";
}
```

```javascript
$arr = array("one", "two", "three");
reset($arr);
while (list($value) = each($arr)) {
    echo "Value: $value<br />\n";
}
```
JavaScript and PHP

- Critical points of failure for web security
  - Attacks come from obscure, difficult corner cases
  - Do not leave out tricky or inelegant constructs

```javascript
function b()
{
  try {throw (function(){return this});}
  catch (get_scope){get_scope().ref=function(x){return x};
  this.alert("Hacked!")}
}
```

- OK to look at conservative subsets
  - But beware of unsound simplifications

```php
$arr = array("one", "two", "three");
foreach ($arr as $value) {
  echo "Value: $value<br />
} reset($arr);
while (list(, $value) = each($arr)) {
  echo "Value: $value<br />
}
Libraries

- JavaScript, PHP = Master
- Browser, server = Blaster
- We need operational semantics of the core language
  - Plus a mechanism to invoke library functions
- Formalization of libraries is an independent task
  - Different goals, techniques
  - One language, many libraries
Formalization: The Pain

```javascript
$a = array('a', 'b', 'c');
foreach ($a as &$v) {...}  // aliasing
foreach ($a as $v) {...}
```

// Error cases: invalid argument
rule [foreach-scalar-1]:
  <@ (Ech('L'Loc, ...)) =>
  WARNING: invalid argument supplied for foreach() in X on line 23

rule [foreach-scalar-2]:
  <@ (Ech('V'ScaleValue, ...)) =>
  WARNING: invalid argument supplied for foreach() in X on line 23

rule [foreach-null]:
  <@ (Ech('ArgK', ...)) =>
  WARNING: invalid argument supplied for foreach() in X on line 23

// Invalid pattern
rule [foreach-invalid-pattern]:
  <@ (Ech('L'Loc, ...)) =>
  ERROR: invalid argument supplied for foreach() in X on line 23
```

// Error cases: invalid argument
rule [foreach-scalar-1]:
  <@ (Ech('L'Loc, ...)) =>
  WARNING: invalid argument supplied for foreach() in X on line 23

rule [foreach-scalar-2]:
  <@ (Ech('V'ScaleValue, ...)) =>
  WARNING: invalid argument supplied for foreach() in X on line 23

rule [foreach-null]:
  <@ (Ech('ArgK', ...)) =>
  WARNING: invalid argument supplied for foreach() in X on line 23

// Invalid pattern
rule [foreach-invalid-pattern]:
  <@ (Ech('L'Loc, ...)) =>
  ERROR: invalid argument supplied for foreach() in X on line 23
```
Formalization: The Pain

```plaintext
$a = array('a', 'b', 'c');
foreach ($$v as &$$v) { }; // aliasing
foreach ($$v as $$v) { ];
array(3) { [0]=> string(1) "a"
          [1]=> string(1) "b"
          [2]=> string(1) "c" ]
```

// Error cases: invalid argument

```plaintext
// foreach-scalar-1:
$ ForEach(3, [\$k], [\$k]) =>
    arrayWarning: Invalid argument supplied for foreach() in $x on line 3'\n
// foreach-scalar-2:
$ ForEach($x, $y) =>
    arrayWarning: Invalid argument supplied for foreach() in $x on line 3'\n```
Mechanization: The Gain
Parsing

- Manual or lightweight parsing
  - Ok for small projects, not scalable
- A “user-friendly” parser
  - Will get you started quickly but sometimes may be wrong
  - JSCert: based on Closure/Rhino
  - KPHP: based on PHP-front
- A “production” parser
  - Tried with Chromium AST: optimizations get in the way
- Parsing should be verified
  - Also source of security problems (XSS, SQLI, ...)
Execution and Testing

• JSSec: manual execution (not scalable)
  – Experiments with various browsers
  – Driven by corner cases of specification

• JSCert: Coq to OCAML extraction
  – JSRef + proof: significant overhead, but trusted
  – Systematic validation of JSRef using test262

• KPHP: semantics is directly executable
  – PHP has no analogous to ES3/5 specification
  – (Zend) test-driven semantics development
Testing, Proofs and Analyses
Coverage

- Lots of possible criteria (Daniel’s talk)
- JSCert: LOC
  - Mapping interpreter code/semantics rules
  - Bisect: general-purpose tool for LOC coverage
  - test262: ~95% LOC
- KPHP: ROS
  - Interpreter as black box
  - Instrumentation of semantics with rule traces
  - Zend tests (56% ROS) + our own tests: 100% ROS
- Open problem: automatically derive conformance test suite from formal semantics
Meta-Proofs

• JSSec: paper proof, labor intensive, error-prone

Theorem 1 (Progress and Preservation). For all states \( S = (H, l, t) \) and \( S' = (H', l', t') \):

- \( (\text{Wf}(S) \land S \rightarrow S') \Rightarrow \text{Wf}(S') \) (Preservation)
- \( \text{Wf}(S) \land t \notin v(t) \Rightarrow \exists S' (S \rightarrow S') \) (Progress)

where \( v(t) = \text{ve} \) if \( t \in \text{Expr} \) and \( v(t) = \text{co} \) if \( t \in \text{Stmtnt} \) or \( \text{Prog} \).

• JSCert: Coq proof, even more labor, but trusted

Theorem run\_javascript\_correct : \( \forall n : \text{nat} \) \( p : \text{prog} \) \( o : \text{out} \),

\( \text{run\_javascript} \text{ (runs n) p = result\_some (spec\_secret\_out o) \rightarrow red\_javascript} \text{ p o} \).

• Useful for debugging the semantics
• Basis for further proofs
  - Coq proof: 6 months to find the right way, 3 days to do
Analyses

• Secure subsets, Defensive JavaScript, Program logics
  – Proofs of reduction-closed invariants need only semantic rules used by subset

• Temporal verification of PHP programs
  – Based on built-in symbolic execution and LTL model checking
  – Verification tools based on meta-language cover whole semantics

• PHP taint analysis based on abstract interpretation
  – Easy to turn executable semantics into static analyzer
Engaging With the Industrial Communities
LANGUAGE EVOLUTION

- JSsec: formalizes ES3
- Horwat: Lisp interpreter for JavaScript 2.0/ES4
- Herman & Flanagan: ES4 specification in ML
- Lambda-JS: ES3 and now ES5S
- JSCert: starts with ES5, open ended
- Language evolution is indeed a challenge
  - Not a good excuse to avoid formalizations
  - You can design a semantics with evolution in mind
12.6.2 The while Statement

The production IterationStatement : while ( Expression ) Statement is evaluated as follows:

1. Let V = empty.
2. Repeat
   a. Let exprRef be the result of evaluating Expression.
   b. If ToBoolean(GetValue(exprRef)) is false, return (normal, V, empty).
   c. Let stmt be the result of evaluating Statement.
   d. If stmt.value is not empty, let V = stmt.value.
   e. If stmt.type is not continue && stmt.target is not in the current label set, then
      i. If stmt.type is break and stmt.target is in the current label set, then
         1. Return (normal, V, empty).
      ii. If stmt is an abrupt completion, return stmt.

Reporting Bugs

• JSSec:
  – Implementation inconsistencies in browsers
  – (Security) bugs in FBJS, ADSafe, etc.

• JSCert:
  – Bugs in SpiderMonkey, V8, WebKit
  – Problems with ES6, test262

• KPHP:
  – Several horror stories (= bugs)
  – No PHP spec: “It’s not a bug! It’s a feature!!”
PHP: What is a Bug?

- Evaluation order of expressions: LR or RL?

```php
$a = array("one");
$c = $a[0].($a[0] = "two");
echo $c; // prints "onetwo"
```

- PHP bug 61188

[2012-02-26 19:04 UTC] rasmus@php.net

I do see your argument, but you are making assumptions about how PHP handles sequence points in expressions which is not documented and thus not strictly defined.

[2012-09-01 19:01 UTC] avp200681 at gmail dot com

[...]
I've found in PHP documentation:
"Operators on the same line have equal precedence, in which case associativity decides the order of evaluation."
PHP: What is a Bug?

- Formal semantics explains what happens

```php
$a = array("one");
$c = $a[0].($a[0] = "two");
echo $c; // prints "onetwo"
```

- Evaluation order is LR
- Array accesses are evaluated to values
- Variables are evaluated to references
- References are resolved lazily

- Easy fix to expose LR evaluation consistently

```
$a = "one";
$c = $a.($a = "two");
echo $c; // prints "twotwo"
```

Conclusions

- Toy models of programming languages
  - Ok for new language features, analysis ideas.
  - Inadequate to provide security guarantees
- Full-blown formal semantics
  - Basis for trustworthy verification, certification.
  - Tools and techniques are now mature enough.
REFERENCES

• JSSec:
  – Secure subsets: CSF’09, ESORICS’09, OAKLAND’10
  – Program logics: POPL’12

• JSCert:

• KPHP:
  – Submitted. TR available 12/2/14 on http://www.doc.ic.ac.uk/~mafeis/