An Operational Semantics for JavaScript

Sergio Maffeis[1], John C. Mitchell[2], Ankur Taly[2],

[1] Department of Computing, Imperial College London
[2] Department of Computer Science, Stanford University

Abstract. We define a small-step operational semantics for the ECMAScript standard language corresponding to JavaScript, as a basis for analyzing security properties of web applications and mashups. The semantics is based on the language standard and a number of experiments with different implementations and browsers. Some basic properties of the semantics are proved, including a soundness theorem and a characterization of the reachable portion of the heap.

1 Introduction

JavaScript [8,14,10] is widely used in Web programming and it is implemented in every major browser. As a programming language, JavaScript supports functional programming with anonymous functions, which are widely used to handle browser events such as mouse clicks. JavaScript also has objects that may be constructed as the result of function calls, without classes. The *properties* of an object, which may represent methods or fields, can be inherited from a prototype, or redefined or even removed after the object has been created. For these and other reasons, formalizing JavaScript and proving correctness or security properties of JavaScript mechanisms poses substantial challenges.

Although there have been scientific studies of limited subsets of the language [7,21,24], there appears to be no previous formal investigation of the full core language, on the scale defined by the informal ECMA specifications [14]. In order to later analyze the correctness of language-based isolation mechanisms for JavaScript, such as those that have arisen recently in connection with online advertising and social networking [1,2,6,20], we develop a small-step operational semantics for JavaScript that covers the language addressed in the ECMA-262 Standard, 3rd Edition [14]. This standard is intended to define the common core language implemented in all browsers and is roughly a subset of JavaScript 1.5. We provide a basis for further analysis by proving some properties of the semantics, such as a progress theorem and properties of heap reachability.

As part of our effort to make conformance to the informal standard evident, we define our semantics in a way that is faithful to the common explanations of JavaScript and the intuitions of JavaScript programmers. For example, JavaScript scope is normally discussed in relation to an object-based representation. We therefore define execution of a program with respect to a heap that contains a linked structure of objects instead of a separate stack. Thus entering a JavaScript scope creates an object on the heap, serving as an activation record for that scope but also subject to additional operations on JavaScript objects. Another unusual aspect of our semantics, reflecting the unusual nature of JavaScript, is that declarations within the body of a function are handled by a two-pass method. The body of a function is analyzed for declarations, which are then added to the scope before the function body is executed. This allows a declaration that appears after the first expression in the function body to be referenced in that expression.

While the ECMAScript language specification guided the development of our operational semantics, we performed many experiments to check our understanding of the specification and to determine differences between various implementations of JavaScript. The implementations that we considered include SpiderMonkey [17] (used by Firefox), the Rhino [4] implementation for Java, JScript [3] (used by Internet Explorer), and the implementations provided in Safari and Opera. In the process, we developed a set of programs that test implementations against the standard and reveal details of these implementations. Many of these program examples, a few of which appear further below, may be surprising to those familiar with more traditional programming languages. Because of the complexity of JavaScript and the number of language variations, our operational semantics (reported in full in [15]) is approximately 70 pages of rules and definitions, in ascii format. We therefore describe only a few of the features and implications of the semantics here. By design, our operational semantics is modular in a way that allows individual clauses to be varied to capture differences between implementations.

Since JavaScript is an unusual language, there is value and challenge in proving properties that might be more straightforward to verify for some other languages (or for simpler idealized subsets of JavaScript). We start by proving a form of soundness theorem, stating that evaluation progresses to an exception or a value of an expected form. Our second main theorem shows, in effect, that the behavior of a program depends only on a portion of the heap. A corollary is that certain forms of garbage collection, respecting the precise characterization of heap reachability used in the theorem, are sound for JavaScript. This is non-trivial because JavaScript provides a number of ways for an expression to access, for example, the parent of a parent of an object, or even its own scope object, increasing the set of potentially reachable objects. The precise statement of the theorem is that the operational semantics preserve a similarity relation on states (which include the heap).

There are several reasons why the reachability theorem is important for various forms of JavaScript analysis. For example, a web server may send untrusted code (such as an advertisement) as part of a trusted page (the page that contains third-party advertisement). We would therefore like to prove that untrusted code cannot access certain browser data structures associated with the trusted enclosing page, under specific conditions that could be enforced by web security mechanisms. This problem can be reduced to proving that a given well-formed JavaScript program cannot access certain portions of the heap, according to the operational semantics of the language. Another future application of heap bisimilarity (as shown in this paper) to security properties of JavaScript applications is that in the analysis of automated phishing defenses, we can reduce the question of whether JavaScript can distinguish between the original page and a phishing page to whether there exists a bisimulation between a certain good heap (corresponding to the original page) and a certain bad heap (corresponding to the phishing page). Thus the framework that we develop in this paper for proving basic progress and heap reachability theorems provides a useful starting point for JavaScript security mechanisms and their correctness proofs.

1.1 JavaScript Overview and Challenges

JavaScript was originally designed to be a simple HTML scripting language [8]. The main primitives are first-class and potentially higher-order functions, and a form of object that can be defined by an object expression, without the need for class declarations. Commonly, related objects are constructed by calling a function that creates objects and returns them as a result of the function call. Functions and objects have *properties,* which are accessed via the "dot" notation, as in x.p for property p of object x. Properties can be added to an object or reset by assignment. This makes it conceptually possible to represent activation records by objects, with assignable variables considered properties of the object corresponding to the current scope. Because it is possible to change the value of a property arbitrarily, or remove it from the object, static typing for full JavaScript is difficult. JavaScript also has eval, which can be used to parse and evaluate a string as an expression, and the ability to iterate over properties of an object or access them using string expressions instead of literals (as in x["p"]). Many online tutorials and books [10] are available.

One example feature of JavaScript that is different from other languages that have been formally analyzed is the way that declarations are processed in an initial pass before bytecode for a function or other construct is executed. Some details of this phenomenon are illustrated by the following code:

```
var f = function(){if (true) {function g() {return 1}}
                      else {function g() {return 2}};
                    function g() {return 3};
                    return g();
                    function g() {return 4}}
```

This code defines a function f whose behavior is given by one of the declarations of g inside the body of the anonymous function that returns g. However, different implementations disagree on which declaration determines the behavior of f. Specifically, a call f() should return 4 according to the ECMA specification. Spidermonkey (hence Firefox) returns 4, while Rhino and Safari return 1, and JScript and the ECMA4 reference implementation return 2. Intuitively, the function body is parsed to find and process all declarations before it is executed, so that reachability of second declarations is ignored. Given that, it is plausible that most implementations would pick either the first declaration or the last. However, this code is likely to be unintuitive to most programmers.

A number of features of JavaScript are particularly challenging for development of a formal semantics and proving properties of the language. Below, we just cite some. Global values such as undefined or Object can be redefined, so the semantics cannot depend on fixed meanings for these predefined parts of the language. Some JavaScript objects, such as Array.prototype, are implicitly reachable even without naming any variables in the global scope. The mutability of these objects allows apparently unrelated code to interact. Some properties of native JavaScript objects are constrained to be for example ReadOnly, but there is no mechanism to express these constraints in the (client) language. JavaScript's rules for binding this depend on whether a function is invoked as a constructor, as a method, as a normal function, etc.. If a function written to be called in one way is instead called in another way, its this property might be bound to an unexected object or even to the global environment.

1.2 Beyond this Paper

Our framework for studying the formal properties of JavaScript closely follows the specification document and models all the features of the language that we have considered necessary to represent faithfully its semantics. The semantics can be modularly extended to *user-defined getters and setters*, which are part of JavaScript 1.5 but not in the ECMA-262 standard. We believe it is similarly possible to extend the semantics to *DOM objects*, which are part of an independent specification, and are available only when JavaScript runs in a Web-browser. However, we leave development of these extensions to future work.

For simplicity, we do not model some features which are laborious but do not add new insight to the semantics, such as the switch and for construct (we do model the for−in), parsing (which is used at run time for example by the eval command), the native Date and Math objects, minor type conversions like ToUInt32, etc. and the details of standard procedures such as converting a string into the numerical value that it actually represents. For the same reason, we also do not model *regular expression matching*, which is used in string operations.

In Section 5 we summarize some directions for future work.

2 Operational Semantics

Our operational semantics consists of a set of rules written in a conventional meta-notation. The notation is not directly executable in any specific automated framework, but is designed to be humanly readable, insofar as is possible for a programming language whose syntax requires 16 pages of specification, and a suitable basis for rigorous but un-automated proofs. Given the space constraints of a conference paper, we describe only the main semantic functions and some representative axioms and rules; the full semantics is currently available online [15].

In order to keep the semantic rules concise, we assume that source programs are legal JavaScript programs, and that each expression is disambiguated (e.g.

5+(3∗4)). We also follow systematic conventions about the syntactic categories of metavariables, to give as much information as possible about the intended type of each operation. In addition to the source expressions and commands used by JavaScript programmers, our semantics uses auxiliary syntactic forms that conveniently represent intermediate steps in our small-step semantics.

In principle, for languages whose semantics are well understood, it may be possible to give a direct operational semantics for a core language subset, and then define the semantics of additional language constructs by showing how these additional constructs are expressible in the core language. Instead of assuming that we know how to correctly define some parts of JavaScript from others, we decided to follow the ECMA specification as closely as possible, defining the semantics of each construct directly as given in the ECMA specification. While giving us the greatest likelihood that the semantics is correct, this approach also did not allow us to factor the language into independent sublanguages. While our presentation is divided into sections for *Types*, *Expressions*, *Objects*, and so on, the execution of a program containing one kind of construct may rely on the semantics of other constructs. We consider it an important future task to streamline the operational semantics and prove that the result is equivalent to the form derived from the standard.

Syntactic Conventions. In the rest of the paper we abbreviate t1,..., tn with t~ and t1 ... tn with t∗ (t+ in the nonempty case). In a grammar, [t] means that t is optional, t|s means either t or s, and in case of ambiguity we escape with apices, as in escaping [by *"["*. Internal values are prefixed with &, as in &NaN. For conciseness, we use short sequences of letters to denote metavariables of a specific type. For example, m ranges over strings, pv over primitive values, etc.. These conventions are summarized in Figure 1. In the examples, unless specified otherwise, JavaScript code prefixed by *js>* is verbatim code from the SpiderMonkey shell (release 1.7.0 2007-10-03).

2.1 Heap

Heaps and Values. Heaps map locations to objects, which are records of pure values va or functions fun(x,...){P}, indexed by strings m or internal identifiers @x (the symbol @ distinguishes internal from user identifiers). Values are standard. As a convention, we append w to a syntactic category to denote that the corresponding term may belong to that category or be an exception. For example, lw denotes an address or an exception.

Heap Functions. We assume a standard set of functions to manipulate heaps. alloc(H,o) = H1,l allocates o in H returning a fresh address l for o in H1. H(l) = o retrieves o from l in H. o.i = va gets the value of property i of o. o−i = fun([x~]){P} gets the function stored in property i of o. o:i = {[a~]} gets the possibly empty set of attributes of property i of o. H(l.i=ov)=H1 sets the property i of l in H to the object value ov. del(H,l,i) = H1 deletes i from l in H. i !< o holds if o does not have property i. i < o holds if o has property i.

```
H ::= (l:o)~ % heap
l ::= #x % object addresses
x ::= foo | bar | ... % identifiers (do not include reserved words)
o ::= "{"[(i:ov)~]"}" % objects
i ::= m | @x % indexes
ov ::= va["{"a~"}"] % object values
       | fun"("[x~]")"{"P"}" % function
a ::= ReadOnly| DontEnum | DontDelete % attributes

pv ::= m | n | b | null | &undefined % primitive values
m ::= "foo" | "bar" | ... % strings
n ::= −n | &NaN | &Infinity | 0 | 1 | ... % numbers
b ::= true | false % booleans
va ::= pv | l % pure values
r ::= ln"∗"m % references
ln ::= l | null % nullable addresses
v ::= va | r % values
w ::= "<"va">" % exception
```

Fig. 1. Metavariables and Syntax for Values

2.2 Semantics Functions

We have three small-step semantic relations for expressions, statements and programs denoted respectively by $\xrightarrow{e}$, $\xrightarrow{s}$, $\xrightarrow{P}$. Each semantic function transforms a heap, a pointer in the heap to the current scope, and the current term being evaluated into a new heap-scope-term triple. The evaluation of expressions returns either a value or an exception, the evaluation of statements and programs terminates with a completion (explained below).

The semantic functions are recursive, and mutually dependent. The semantics of programs depends on the semantics of statements which in turn depends on the semantics of expressions which in turn, for example by evaluating a function, depends circularly on the semantics of programs. These dependencies are made explicit by contextual rules, that specify how a transition derived for a term can be used to derive a transition for a bigger term including the former as a sub-term. In general, the premises of each semantic rule are predicates that must hold in order for the rule to be applied, usually built of very simple mathematical conditions such as $t < S$ or $t \mathrel{!}= t'$ or $f(a) = b$ for set membership, inequality and function application.

Transitions axioms (rules that do not have transitions in the premises) specify the individual transitions for basic terms (the redexes). For example, the axiom H,l,(v) $\longrightarrow$ H,l,v describes that brackets can be removed when they surround a value (as opposed to an expression, where brackets are still meaningful).

Contextual rules propagate such atomic transitions. For example, if program H,l,P evaluates to H1,l1,P1 then H,l,@FunExe(l',P) (an internal expression used to evaluate the body of a function) reduces in one step to H1,l1,@FunExe(l',P1).

The rule below show exactly that: @FunExe(l,−) is one of the contexts eCp for evaluating programs.

$$\frac{\mathsf{H,l,P} \xrightarrow{P} \mathsf{H1,l1,P1}}{\mathsf{H,l,eCp[P]} \xrightarrow{e} \mathsf{H1,l1,eCp[P1]}}$$

As another example, sub-expressions are evaluated inside outer expressions (rule on the left) using contexts eC ::= typeof eC | eCgv | ..., and exceptions propagated to the top level (axiom on the right).

$$\frac{\mathsf{H,l,e} \xrightarrow{e} \mathsf{H1,l1,e1}}{\mathsf{H,l,eC[e]} \xrightarrow{e} \mathsf{H1,l1,eC[e1]}} \qquad \mathsf{H,l,eC[w]} \xrightarrow{e} \mathsf{H,l,w}$$

Hence, if an expression throws an exception ($\mathsf{H,l,e} \xrightarrow{e} \mathsf{H1,l,w}$) then so does say the typeof operator: $\mathsf{H,l,typeof\ e} \xrightarrow{e} \mathsf{H1,l,typeof\ w} \xrightarrow{e} \mathsf{H1,l,w}$.

It is very convenient to nest contexts inside each other. For example, contexts for GetValue (the internal expression that returns the value of a reference), generated by the grammar for eCgv ::= −[e] | va[−] | eCto | eCts | ..., are expression contexts. Similarly, contexts for converting values to objects eCto ::= −[va] | ... or to strings eCts ::= l[−] | ... are get-value contexts.

$$\mathsf{H,l,eCgv[ln{*}m]} \xrightarrow{e} \mathsf{H1,l,eCgv[@GetValue(ln{*}m)]}$$

$$\frac{\mathsf{Type(va)\ !{=}\ Object} \quad \mathsf{ToObject(H,va) = H1,lw}}{\mathsf{H,l,eCto[va]} \xrightarrow{e} \mathsf{H1,l,eCto[lw]}}$$

$$\frac{\mathsf{Type(v)\ !{=}\ String} \quad \mathsf{ToString(v) = e}}{\mathsf{H,l,eCts[v]} \xrightarrow{e} \mathsf{H,l,eCts[e]}}$$

As a way to familiarize with these structures of nested contexts, we look in detail at the concrete example of member selection. The ECMA-262 specification states that in order to evaluate `MemberExpression[Expression]` one needs to:

```
MemberExpression : MemberExpression [ Expression ]
1. Evaluate MemberExpression.
2. Call GetValue(Result(1)).
3. Evaluate Expression.
4. Call GetValue(Result(3)).
5. Call ToObject(Result(2)).
6. Call ToString(Result(4)).
7. Return a value of type Reference whose base object is Result(5) and
whose property name is Result(6).
```

In our formalization, the rule for member selection is just $\mathsf{H,l,l1[m]} \xrightarrow{e} \mathsf{H,l,l1{*}m}$. As opposed to the textual specification, the formal rule is trivial, and makes it obvious that the operator takes an object and a string and returns the corresponding reference. All we had to do in order to model the intermediate steps was to insert the appropriate contexts in the evaluation contexts for expressions, values, objects and strings. In particular, $-[\mathsf{e}]$ is value context, and value contexts are expression contexts, so we get for free steps 1 and 2, obtaining $\mathsf{va}[\mathsf{e}]$. Since $\mathsf{va}[-]$ is also a value context, steps 3 and 4 also come for free, obtaining $\mathsf{va}[\mathsf{va}]$. Since $-[\mathsf{va}]$ is an object context, and $\mathsf{l}[-]$ a string context, steps 6 and 7 are also executed transparently. The return type of each of those contexts guarantees that the operations are executed in the correct order. If that was not the case, then the original specification would have been ambiguous. The rule for propagating exceptions takes care of any exception raised during these steps.

The full formal semantics [15] contains several other contextual rules to account for other mutual dependencies and for all the implicit type conversions. This substantial use of contextual rules greatly simplifies the semantics and will be very useful in Section 3 to prove its formal properties.

Scope and Prototype Lookup. The scope and prototype chains are two distinctive features of JavaScript. The stack is represented implicitly, by maintaining a chain of objects whose properties represent the binding of local variables in the scope. Since we are not concerned with performance, our semantics needs to know only a pointer to the head of the chain (the current scope object). Each scope object stores a pointer to its enclosing scope object in an internal @Scope property. This helps in dealing with constructs that modify the scope chain, such as function calls and the with statement.

JavaScript follows a prototype-based approach to inheritance. Each object stores in an internal property @Prototype a pointer to its prototype object, and inherits its properties. At the root of the prototype tree there is @Object.prototype, that has a null prototype. The rules below illustrate prototype chain lookup.

$$\mathsf{Prototype}(\mathsf{H},\mathsf{null},\mathsf{m})=\mathsf{null}$$

$$\frac{\mathsf{m} < \mathsf{H}(\mathsf{l})}{\mathsf{Prototype}(\mathsf{H},\mathsf{l},\mathsf{m})=\mathsf{l}} \qquad \frac{\mathsf{m}\,!\!< \mathsf{H}(\mathsf{l}) \quad \mathsf{H}(\mathsf{l}).\mathsf{@Prototype}=\mathsf{ln}}{\mathsf{Prototype}(\mathsf{H},\mathsf{l},\mathsf{m})=\mathsf{Prototype}(\mathsf{H},\mathsf{ln},\mathsf{m})}$$

Function $\mathsf{Scope}(\mathsf{H},\mathsf{l},\mathsf{m})$ returns the address of the scope object in H that first defines property m, starting from the current scope l. It is used to look up identifiers in the semantics of expressions. Its definition is similar to the one for prototype, except that the condition $(\mathsf{H},\mathsf{l}.\mathsf{@HasProperty}(\mathsf{m}))$ (which navigates the prototype chain to check if l has property m) is used instead of the direct check $\mathsf{m} < \mathsf{H}(\mathsf{l})$.

Types. JavaScript values are dynamically typed. The internal types are:

$$\begin{aligned}\mathsf{T} ::= {} & \mathsf{Undefined} \mid \mathsf{Null} \mid \mathsf{Boolean} \mid \mathsf{String} \mid \mathsf{Number} \ \textit{\% primitive types}\\ & \mid \mathsf{Object} \mid \mathsf{Reference} \ \textit{\% other types}\end{aligned}$$

Types are used to determine conditions under which certain semantic rules can be evaluated. The semantics defines straightforward predicates and functions which perform useful checks on the type of values. For example, $\mathsf{IsPrim}(\mathsf{v})$ holds when v is a value of a primitive type, and $\mathsf{GetType}(\mathsf{H},\mathsf{v})$ returns a string corresponding to a more intuitive type for v in H. The user expression typeof e, which returns the type of its operand, uses internally GetType. Below, we show the case for function objects (i.e. objects which implement the internal @Call property).

$$\frac{\mathsf{Type(v)=Object} \quad \mathsf{@Call} < \mathsf{H(v)}}{\mathsf{GetType(H,v)} = \textit{"function"}}$$

An important use of types is to convert the operands of typed operations and throw exceptions when the conversion fails. There are implicit conversions into strings, booleans, number, objects and primitive types, and some of them can lead to the execution of arbitrary code. For example, the member selection expression e1[e2] implicitly converts e2 to a string. If e2 denotes an object, its re-definable toString propery is invoked as a function.

```
js> var o={a:0}; o[{toString:function(){o.a++; return "a"}}] % res: 1
```

The case for ToPrimitive (invoked by ToNumber and ToString) is responsible for the side effects: the result of converting an object value into a primitive value is an expression l.@DefaultValue([T]) which may involve executing arbitrary code that a programmer can store in the valueOf or toString methods of said object.

$$\frac{\mathsf{Type(l)=Object}}{\mathsf{ToPrimitive(l[,T])} = \mathsf{l.@DefaultValue([T])}}$$

2.3 Expressions

We distinguish two classes of expressions: internal expressions, which correspond to specification artifacts needed to model the intended behavior of user expressions, and user expressions, which are part of the user syntax of JavaScript. Internal expressions include addresses, references, exceptions and functions such as @GetValue,@PutValue used to get or set object properties, and @Call,@Construct used to call functions or to construct new objects using constructor functions. For example, we give two rules of the specification of @Put, which is the internal interface (used also by @PutValue) to set properties of objects. The predicate H,l1.@CanPut(m) holds if m does not have a ReadOnly attribute.

$$\frac{\begin{array}{l}\mathsf{H,l1.@CanPut(m)}\\ \mathsf{m} \mathrel{!<} \mathsf{H(l1)} \quad \mathsf{H(l1.m=va\{\})=H1}\end{array}}{\mathsf{H,l,l1.@Put(m,va)} \xrightarrow{e} \mathsf{H1,l,va}}$$

$$\frac{\begin{array}{l}\mathsf{H,l1.@CanPut(m)}\\ \mathsf{H(l1):m=\{[a\tilde{\ }]\}} \quad \mathsf{H(l1.m=va\{[a\tilde{\ }]\})} = \mathsf{H1}\end{array}}{\mathsf{H,l,l1.@Put(m,va)} \xrightarrow{e} \mathsf{H1,l,va}}$$

These rules show that fresh properties are added with an empty set of attributes, whereas existing properties are replaced maintaining the same set of attributes.

Object Literal. As an example of expressions semantics we present in detail the case of object literals. The semantics of the object literal expression {pn:e,...,pn':e'} uses an auxiliary internal construct AddProps to add the result of evaluating each e as a property with name pn to a newly created empty object.

Rule (1) (with help from the contextual rules) creates a new empty object, and passes control to AddProps. Rule (2) converts identifiers to strings, and rule (3) adds a property to the object being initialized. It uses a sequential expression to perform the update and then return the pointer to the updated object l1, which rule (4) releases at the top level.

$$\mathsf{H,l,\{[(pn{:}e)\tilde{}]\}} \xrightarrow{e} \mathsf{H,l,@AddProps(new\ Object()[,(pn{:}e)\tilde{}])} \quad (1)$$

$$\mathsf{H,l,@AddProps(l1,x{:}e[,(pn{:}e)\tilde{}])} \xrightarrow{e} \mathsf{H,l,@AddProps(l1,"x"{:}e[,\ (pn{:}e)\tilde{}])} \quad (2)$$

$$\mathsf{H,l,@AddProps(l1,m{:}va[,(pn{:}e)\tilde{}])} \xrightarrow{e} \mathsf{H,l,@AddProps((l1.@Put(m,va),l1)[,(pn{:}e)\tilde{}])} \ (3)$$

$$\mathsf{H,l,@AddProps(l1)} \xrightarrow{e} \mathsf{H,l,l1} \quad (4)$$

Rule (1) is emblematic of a few other cases in which the specification requires to create a new object by evaluating a specific constructor expression whose definition can be changed during execution. For example,

```
js> var a = {}; a % res: [object Object]
ljs> Object = function(){return new Number()}; var b = {}; b % res: 0
```

where the second object literal returns a number object. That feature can be useful, but can also lead to undesired confusion.

2.4 Statements

Similarly to the case for expressions, the semantics of statements contains a certain number of internal statements, used to represent unobservable execution steps, and user statements that are part of the user syntax of JavaScript. A completion is the final result of evaluating a statement.

```
co ::= "("ct,vae,xe")" vae ::= &empty | va xe ::= &empty | x
ct ::= Normal | Break | Continue | Return | Throw
```

The completion type indicates whether the execution flow should continue normally, or be disrupted. The value of a completion is relevant when the completion type is Return (denoting the value to be returned), Throw (denoting the exception thrown), or Normal (propagating the value to be return during the execution of a function body). The identifier of a completion is relevant when the completion type is either Break or Continue, denoting the program point where the execution flow should be diverted to.

Expression and Throw. Evaluation contexts transform the expression operand of these constructs into a pure value. All the semantic rules specify is how such value is packaged into a completion. The rules for return,continue,break are similar.

$$\mathsf{H,l,va} \xrightarrow{s} \mathsf{H,l,(Normal,va,\&empty)} \qquad \mathsf{H,l,throw\ va;} \xrightarrow{s} \mathsf{H,l,(Throw,va,\&empty)}$$

2.5 Programs

Programs are sequences of statements and function declarations.

$$\mathsf{P} ::= \mathsf{fd}\ [\mathsf{P}] \mid \mathsf{s}\ [\mathsf{P}] \qquad\qquad \mathsf{fd} ::= \mathsf{function}\ \mathsf{x}\ \text{"("}[\mathsf{x}\tilde{}]\text{")\{"}[\mathsf{P}]\text{"\}"}$$

As usual, the execution of statements is taken care of by a contextual rule. If a statement evaluates to a break or continue outside of a control construct, an SyntaxError exception is thrown (rule (9)). The run-time semantics of a function declaration instead is equivalent to a no-op (rule (10)). Function (and variable) declarations should in fact be parsed once and for all, before starting to execute the program text. In the case of the main body of a JavaScript program, the parsing is triggered by rule (11) which adds to the initial heap NativeEnv first the variable and then the function declarations (functions VD,FD).

$$\frac{\begin{array}{l}\mathsf{ct} < \{\mathsf{Break},\mathsf{Continue}\}\\ \mathsf{o} = \mathsf{new\_SyntaxError}() \quad \mathsf{H1},\mathsf{l1} = \mathsf{alloc}(\mathsf{H},\mathsf{o})\end{array}}{\mathsf{H},\mathsf{l},(\mathsf{ct},\mathsf{vae},\mathsf{xe})\ [\mathsf{P}] \xrightarrow{P} \mathsf{H1},\mathsf{l},(\mathsf{Throw},\mathsf{l1},\&\mathsf{empty})} \quad (9)$$

$$\mathsf{H},\mathsf{l},\mathsf{function}\ \mathsf{x}\ ([\mathsf{x}\tilde{}])\{[\mathsf{P}]\}\ [\mathsf{P1}] \xrightarrow{P} \mathsf{H},\mathsf{l},(\mathsf{Normal},\&\mathsf{empty},\&\mathsf{empty})\ [\mathsf{P1}] \quad (10)$$

$$\frac{\begin{array}{l}\mathsf{VD}(\mathsf{NativeEnv},\#\mathsf{Global},\{\mathsf{DontDelete}\},\mathsf{P}) = \mathsf{H1}\\ \mathsf{FD}(\mathsf{H1},\#\mathsf{Global},\{\mathsf{DontDelete}\},\mathsf{P}) = \mathsf{H2}\end{array}}{\mathsf{P} \xrightarrow{P} \mathsf{H2},\#\mathsf{Global},\mathsf{P}} \quad (11)$$

2.6 Native Objects

The initial heap NativeEnv of core JavaScript contains native objects for representing predefined functions, constructors and prototypes, and the global object @Global that constitutes the initial scope, and is always the root of the scope chain. As an example, we describe the global object. The global object defines properties to store special values such as &NaN, &undefined etc., functions such as eval, toString etc. and constructors that can be used to build generic objects, functions, numbers, booleans and arrays. Since it is the root of the scope chain, its @Scope property points to null. Its @this property points to itself: @Global = {@Scope:null, @this:#Global, *"eval"*:#GEval{DontEnum},...}. None of the non-internal properties are read-only or enumerable, and most of them can be deleted. By contrast, when a user variable or function is defined in the top level scope (i.e. the global object) it has only the DontDelete attribute. The lack of a ReadOnly attribute on *"NaN"*,*"Number"* for example forces programmers to use the expression 0/0 to denote the real &NaN value, even though @Number.NaN stores &NaN and is a read only property.

Eval. The eval function takes a string and tries to parse it as a legal program text. If it fails, it throws a SyntaxError exception (rule (12)). If it succeeds, it parses the code for variable and function declarations (respectively VD,FD) and spawns the internal statement @cEval (rule (13)). In turn, @cEval is an execution context for programs, that returns the value computed by the last statement in P, or $\&\mathsf{undefined}$ if it is empty.

$$\frac{\begin{array}{l}\mathsf{ParseProg(m)} = \&\mathsf{undefined}\\ \mathsf{H2,l2} = \mathsf{alloc(H,o)} \quad \mathsf{o} = \mathsf{new\_SyntaxError()}\end{array}}{\mathsf{H,l,\#GEval.@Exe(l1,m)} \xrightarrow{e} \mathsf{H2,l,<l2>}} \qquad (12)$$

$$\frac{\begin{array}{l}\mathsf{l} \mathrel{!}= \mathsf{\#Global} \quad \mathsf{ParseProg(m)} = \mathsf{P}\\ \mathsf{VD(H,l,\{\},P)} = \mathsf{H1} \quad \mathsf{FD(H1,l,\{\},P)} = \mathsf{H2}\end{array}}{\mathsf{H,l,\#GEval.@Exe(l1,m)} \xrightarrow{e} \mathsf{H2,l,@cEval(P)}} \qquad (13)$$

$$\frac{\mathsf{va} = (\mathit{IF}\ \mathsf{vae}{=}\&\mathsf{empty}\ \mathit{THEN}\ \&\mathsf{undefined}\ \mathit{ELSE}\ \mathsf{vae})}{\mathsf{H,l,@cEval((ct,vae,xe))} \xrightarrow{e} \mathsf{H,l,va}} \qquad (14)$$

As we are not interested in modeling the parsing phase, we just assume a parsing function $\mathsf{ParseProg(m)}$ which given string m returns a valid program P or else $\&\mathsf{undefined}$. Note how in rule (13) the program P is executed in the caller's scope l, effectively giving dynamic scope to P.

Object. The @Object constructor is used for creating new user objects and internally by constructs such as object literals. Its prototype @ObjectProt becomes the prototype of any object constructed in this way, so its properties are inherited by most JavaScript objects. Invoked as a function or as a constructor, @Object returns its argument if it is an object, a new empty object if its argument is undefined or not supplied, or converts its argument to an object if it is a string, a number or a boolean. If the argument is a host object (such as a DOM object) the behavior is implementation dependent.

$$\frac{\begin{array}{l}\mathsf{o} = \mathsf{new\_object}(\mathit{"Object"}\mathsf{,\#ObjectProt})\\ \mathsf{H1,lo} = \mathsf{Alloc(H,o)} \quad \mathsf{Type(pv)} < \{\mathsf{Null,Undefined}\}\end{array}}{\mathsf{H,l,\#Object.@Construct(pv)} \xrightarrow{e} \mathsf{H1,l,lo}}$$

$$\frac{\begin{array}{l}\mathsf{Type(pv)} < \{\mathsf{String,\ Boolean,\ Number}\}\\ \mathsf{H1,le} = \mathsf{ToObject(H,pv)}\end{array}}{\mathsf{H,l,\#Object.@Construct(pv)} \xrightarrow{e} \mathsf{H,l,le}} \qquad \frac{\mathsf{Type(l1)} = \mathsf{Object} \quad \mathsf{!IsHost(H,l1)}}{\mathsf{H,l,\#Object.@Construct(l1)} \xrightarrow{e} \mathsf{H,l,l1}}$$

The object @ObjectProt is the root of the scope prototype chain. For that reason, its internal prototype is null. Apart from *"constructor"*, which stores a pointer to @Object, the other public properties are native meta-functions such as toString or valueOf (which, like user function, always receive a value for @this as the first parameter).

2.7 Relation to Implementations

JavaScript is implemented within the major web browsers, or as standalone shells. In order to clarify several ambiguities present in the specification, we have run experiments inspired by our semantics rules on different implementations.

We have found that, besides resolving ambiguities in different and often incompatible ways, implementations sometimes openly diverge from the specification. In [15] we report several cases.

For example, in SpiderMonkey (the Mozilla implementation) the semantics of functions depends on the position, *within unreachable code* of statements which should have no semantic significance! The function call below returns 0, but if we move the `var g;` after the last definition of `g`, it returns 1.

```
(function(){if (true) function g(){return 0};return g();var g;function g(){return 1}})()
```

Apart from pathological examples such as this, Mozilla's JavaScript extends ECMA-262 in several ways, for example with *getters* and *setters*. A getter is a function that gets called when the corresponding property is accessed and a setter is a function that gets called when the property is assigned. While we can extend our semantics to deal with these further constructs, we leave for future work the full integration of this approach.

3 Formal Properties

In this section we give some preliminary definitions and set up a basic framework for formal analysis of well-formed JavaScript programs. We prove a progress theorem which shows that the semantics is sound and the execution of a well-formed term always progresses to an exception or an expected value. Next we prove a Heap reachability theorem which essentially justifies mark and sweep type garbage collection for JavaScript. Although the properties we prove are fairly standard for idealized languages used in formal studies, proving them for real (and unruly!) JavaScript is a much harder task.

Throughout this section, a program state S denotes a triple (H, l, t) where H is a heap, l is a current scope address and t is a term. Recall that a heap is a map from heap addresses to objects, and an object is a collection of properties that can contain heap addresses or primitive values.

3.1 Notation and Definitions

Expr, *Stmnt* and *Prog* denote respectively the sets of all possible expressions, statements and programs that can be written using the corresponding internal and user grammars. $\mathbb{L}$ denotes the set of all possible heap addresses. *Wf(S)* is a predicate denoting that a state S is well-formed. *prop(o)* is the set of property names present in object *o*. *dom(H)* gives the set of allocated addresses for the heap *H*.

For a heap address l and a term t, we say $l \in t$ iff the heap address l occurs in t. For a state $S = (H, l, t)$, we define $\Delta(S)$ as the set of heap addresses $\{l\} \cup \{l | l \in t\}$. This is also called the set of *roots* for the state S.

We define the well-formedness predicate of a state $S = (H, l, t)$ as the conjunction of the predicates $Wf_{Heap}(H)$, $Wf_{scope}(l)$ and $Wf_{term}(t)$. A term t is well-formed iff it can be derived using the grammar rules consisting of both the language constructs and the internal constructs, and all heap addresses contained in t are allocated ie $l \in t \Rightarrow l \in dom(H)$. A scope address $l \in dom(H)$ is well-formed iff the scope chain starting from l does not contain cycles, and $(@Scope \in prop(H(l))) \wedge (H(l).@Scope \neq null \Rightarrow \mathrm{Wf}(H(l).@Scope))$. A heap H is well-formed iff it conforms to all the conditions on heap objects mentioned in the specification (see the long version [15] for a detailed list).

Definition 1 (Heap Reachability Graph). *Given a heap H, we define a labeled directed graph G_H with heap addresses $l \in dom(H)$ as the nodes, and an edge from address l_i to l_j with label p iff $(p \in prop(H(l_i)) \wedge H(l_i).p = l_j)$.*

Given a heap reachability graph G_H, we can define the view from a heap address l as the subgraph $G_{H,l}$ consisting only of nodes that are reachable from l in graph G_H. We use $\mathrm{view}_H(l)$ to denote the set of heap addresses reachable from l: $\mathrm{view}_H(l) = Nodes(G_{H,l})$. view_H can be naturally extended to apply to a set of heap addresses. Observe that the graph G_H only captures those object properties that point to other heap objects and does not say anything about properties containing primitive values.

Definition 2 (l-Congruence of Heaps $\cong_l$). *We say that two heaps H_1 and H_2 are l-congruent (or congruent with respect to heap address l) iff they have the same* views *from heap address l and the corresponding objects at the heap addresses present in the views are also equal. Formally,*

$$H_1 \cong_l H_2 \Leftrightarrow (G_{H_1,l} = G_{H_2,l} \ \wedge \ \forall\ l' \in \ view_{H_1}(l)\ H_1(l') = H_2(l')).$$

Note that if $H_1 \cong_l H_2$ then $\mathrm{view}_{H_1}(l) = \mathrm{view}_{H_2}(l)$. It is easy to see that if two heaps H_1 and H_2 are congruent with respect to l then they are congruent with respect to all heap addresses $l' \in \mathrm{view}_{H_1}(l)$.

Definition 3 (State congruence $\cong$). *We say that two states $S_1 = (H_1, l, t)$ and $S_2 = (H_2, l, t)$ are congruent iff the heaps are congruent with respect to all addresses in the roots set. Formally, $S_1 \cong S_2 \Leftrightarrow \forall\ l' \in \Delta(S_1)\ (H_1 \cong_{l'} H_2)$).*

Note that $\Delta(S_1) = \Delta(S_2)$ because the definition of Δ depends only on l and t. In the next section we will show that for a state $S = (H, l, t)$, $\mathrm{view}_H(\Delta(S))$ forms the set of *live* heap addresses for the S because these are the only possible heap addresses that can be accessed during any transition from S.

Definition 4 (Heap Address Renaming). *For a given heap H, a heap address renaming function f is any one to one map from* dom*(H) to $\mathbb{L}$.*

We denote the set of all possible heap renaming functions for a heap H by $\mathbb{F}_H$. We overload f so that $f(H)$ is the new heap obtained by renaming all heap addresses $l \in \mathrm{dom}(H)$ by $f(l)$ and for a term t, $f(t)$ is the new term obtained by renaming all $l \in t$ by $f(l)$. Finally, for a state $S = (H, l, t)$ we define $f(S) = (f(H), f(l), f(t))$ as the new state obtained under the renaming.

Definition 5 (State similarity $\sim$). *Two states $S_1 = (H_1, l_1, t_1)$ and $S_2 = (H_2, l_2, t_2)$ are similar iff there exists a renaming function f for H_1 such that the new state $f(S_1)$ obtained under the renaming is congruent to S_2. Formally,*

$$S_1 \sim S_2 \Leftrightarrow \exists\ f \in \mathbb{F}_{H_1}\ f(S_1) \cong S_2.$$

Property 1. Both $\cong$ and $\sim$ are equivalence relations. Moreover, $\cong \subsetneq \sim$.

3.2 Theorems and Formal Properties

We now present the main technical results. Our first result is a progress and preservation theorem, showing that evaluation of a well-formed term progresses to a value or an exception.

Lemma 1. *Let C denote the set of all valid contexts for expressions, statements and programs. For all terms t appropriate for the context C we have $Wf_{term}(C(t)) \Rightarrow Wf_{term}(t)$.*

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

- *$(Wf(S) \ \wedge \ S \rightarrow S') \Rightarrow Wf(S')$ (Preservation)*
- *$Wf(S) \ \wedge \ t \notin v(t) \Rightarrow \exists \ S' \ (S \rightarrow S'))$ (Progress)*

where $v(t) = ve$ if $t \in Expr$ and $v(t) = co$ if $t \in Stmnt$ or $Prog$.

Our second result shows that similarity is preserved under reduction, which directly gives a construction for a simple mark-and-sweep-like garbage collector for JavaScript. The proofs for the theorems are given in the long version [15].

Lemma 2. *For all well-formed program states $S = (H, l, t)$, if $H, l, t \rightarrow H', l', t'$ then $H(l'') = H'(l'')$, for all $l'' \notin view_H(\Delta(H, l, t)) \ \cup \ view_{H'}(\Delta(H', l', t'))$.*

The above lemma formalizes the fact that the only heap addresses accessed during a reduction step are the ones present in the initial and final live address sets. We can formally prove this lemma by an induction over the rules.

Theorem 2 (Similarity preserved under reduction). *For all well-formed program states S_1, S_2 and S_1', $S_1 \sim S_2 \wedge S_1 \rightarrow S_1' \Rightarrow \exists S_2'. S_2 \rightarrow S_2' \wedge S_1' \sim S_2'$.*

We can intuitively understand this theorem by observing that if the reduction of a term does not involve allocation of any new heap addresses then the only addresses that can potentially be accessed during the reduction would be the ones present in the live heap address set. When the program states are similar, then under a certain renaming the two states would have the same live heap address sets . As a result the states obtained after reduction would also be congruent(under the same renaming function). On the other hand, if the reduction involves allocation of new heap addresses then we can simply extend the heap address renaming function by creating a map from the newly allocated addresses in the first heap (H_1') to the newly allocated addresses in the second heap (H_2'). Thus state similarity would be preserved in both cases.

A consequence of Theorem 2 is that we can build a simple mark and sweep type garbage collector for JavaScript. For any program state $S = (H, l, t)$, we mark all the heap addresses that are reachable from $\Delta(S)$. We modify the heap H to H' by freeing up all unmarked addresses and obtain the new program state $S' = (H', l, t)$. It is easy to show that $S' \sim S$. Hence by Theorem 2, a reduction trace starting from t, in a system with garbage collection, would be similar to the one obtained without garbage collection. In other words, garbage collection does not affect the semantics of programs.

4 Related Work

The JavaScript approach to objects is based on Self [23] and departs from the foundational object calculi proposed in the 1990s, e.g., [5,9,16]. Previous foundational studies include operational semantics for a subset of JavaScript [11] and formal properties of subsets of JavaScript [7,19,22,21]. Our aim is different from these previous efforts because we address the full ECMA Standard language (with provisions for variants introduced in different browsers). We believe a comprehensive treatment is important for analyzing existing code and code transformation methods [1,2]. In addition, when analyzing JavaScript security, it is important to consider attacks that could be created using arbitrary JavaScript, as opposed to some subset used to develop the trusted application. Some work on containing the effects of malicious JavaScript include [18,24]. Future versions of JavaScript and ECMAScript are documented in [13,12].

In the remainder of this section, we compare our work to the formalizations proposed by Thiemann [22] and Giannini [7] and comment on the extent to which formal properties they establish for subsets of JavaScript can be generalized to the full language.

Giannini et al. [7] formalize a small subset of JavaScript and give a static type system that prevents run-time typing errors. The subset is non-trivial, as it includes dynamic addition of properties to objects, and constructor functions to create objects. However the subset also lacks important features such as object prototyping, functions as objects, statements such as `with`, `try−catch`, `for−in`, and native functions and objects. This leads to substantial simplifications in their semantics, relative to ours. For example, function definitions are stored in a separate data structure rather than in the appropriate scope object, so there is no scope-chain-based resolution of global variables appearing inside a function body. Their simplification also makes it possible to define a sound type system that does not appear to extend to full JavaScript, as further discussed below.

Thiemann [22] proposes a type system for a larger subset of JavaScript than [7], as it also includes function expressions, function objects, and object literals. The type system associates type signatures with objects and functions and identifies suspicious type conversions. However, Thiemann's subset still does not allow object prototyping, the `with` and the `try−catch` statements, or subtle features of the language such as property attributes or arbitrary variable declarations in the body of a function. As we showed in section 2, these non-trivial (and non-intuitive) aspects of JavaScript make static analysis of arbitrary JavaScript code very difficult.

The substantial semantic difference between the subsets covered in [7,22] and full JavaScript is illustrated by the fact that: (i) Programs that are well-typed in the proposed subsets may lead to type errors when executed using the complete semantics, and (ii) Programs that do not lead to a type error when executed using the complete semantics may be considered ill-typed unnecessarily by the proposed type systems. The first point is demonstrated by `var x = "a"; x.length = function(){ }; x.length()`, which is allowed and well-typed by [22]. However it leads to a type error because although the type system considers implicit type conversion of the string x to a wrapped string object, it does not consider the prototyping mechanism and attributes for properties. Since property length of String.prototype has the ReadOnly attribute, the assignment in the second statement fails silently and thus the method call in the third statement leads to a type error. An example demonstrating the second point above is function f(){return o.g();}; result = f(), which is allowed by [7,22]. If method g is not present in the object o then both type systems consider the expression result = f() ill-typed. However g could be present as a method in the one of the ancestoral prototypes of o, in which case the expression will not lead to a type error. Because object prototyping is the main inheritance mechanism in JavaScript and it is pervasive in almost all real world JavaScript, we believe that a type system that does not consider the effects of prototypes will not be useful without further extension.

5 Conclusions

In this paper, we describe a structured operational semantics for the ECMA-262 standard [14] language. The semantics has two main parts: one-step evaluation relations for the three main syntactic categories of the language, and definitions for all of the native objects that are provided by an implementation. In the process of developing the semantics, we examined a number of perplexing (to us) JavaScript program situations and experimented with a number of implementations. To ensure accuracy of our semantics, we structured many clauses after the ECMA standard [14]. In a revision of our semantics, it would be possible to depart from the structure of the informal ECMA standard and make the semantics more concise, using many possible optimization to reduce its apparent complexity. As a validation of the semantics we proved a soundness theorem, a characterization of the reachable portion of the heap, and some equivalences between JavaScript programs.

In ongoing work, we are using this JavaScript semantics to analyze methods for determining isolation between embedded third-party JavaScript, such as embedded advertisements provided to web publishers through advertising networks, and the hosting content. In particular, we are studying at YAHOO!'s ADsafe proposal [1] for safe online advertisements, the BeamAuth [6] authentication bookmarklet that relies on isolation between JavaScript on a page and JavaScript contained in a browser bookmark, Google's Caja effort [2] to provide code isloation by JavaScript rewriting, and FBJS [20], the subset of JavaScript used for writing FaceBook applications. While trying to formally prove properties of these mechanisms, we have already found violations of the intended security policies of two of them [15].

Acknowledgments. Sergio Maffeis is supported by EPSRC grant EP/E044956 /1. This work was done while the first author was visiting Stanford University, whose hospitality is gratefully acknowledged. Mitchell and Taly acknowledge the support of the National Science Foundation.

References

1. AdSafe: Making JavaScript safe for advertising.
   `http://www.adsafe.org/`.
2. Google-Caja: A source-to-source translator for securing JavaScript-based Web.
   `http://code.google.com/p/google-caja/`.
3. Jscript (Windows Script Technologies).
   `http://msdn2.microsoft.com/en-us/library/hbxc2t98.aspx`.
4. Rhino: Javascript for Java.
   `http://www.mozilla.org/rhino/`.
5. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.
6. B. Adida. BeamAuth: two-factor Web authentication with a bookmark. In *ACM Computer and Communications Security*, pages 48–57, 2007.
7. C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *ECOOP'05*, page 429452, 2005.
8. B. Eich. Javascript at ten years.
   `www.mozilla.org/js/language/ICFP-Keynote.ppt`.
9. K. Fisher, F. Honsell, and J.C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. Computing (*formerly *BIT)*, 1:3–37, 1994.
10. D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly, 2006.
    `http://proquest.safaribooksonline.com/0596101996`.
11. D. Herman. Classic JavaScript.
    `http://www.ccs.neu.edu/home/dherman/javascript/`.
12. D. Herman and C. Flanagan. Status report: specifying JavaScript with ML. In *ML '07: Proc. Workshop on ML*, pages 47–52, 2007.
13. ECMA International. ECMAScript 4.
    `http://www.ecmascript.org`.
14. ECMA International. ECMAScript language specification. stardard ECMA-262, 3rd Edition.
    `http://www.ecma-international.org/publications/ECMA-ST/Ecma-262.pdf`, 1999.
15. S. Maffeis, J. Mitchell, and A. Taly. Complete ECMA 262-3 operational semantics and long version of present paper. Semantics: `http://jssec.net/semantics/` Paper: `http://jssec.net/semantics/sjs.pdf`.
16. J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *POPL '90*, pages 109–124, 1990.
17. Mozilla. Spidermonkey (javascript-c) engine.
    `http://www.mozilla.org/js/spidermonkey/`.
18. C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic HTML. *ACM Transactions on the Web*, 1(3), 2007.
19. J. Siek and W. Taha. Gradual typing for objects. In *ECOOP*, 2007.
20. The FaceBook Team. FBJS.
    `http://wiki.developers.facebook.com/index.php/FBJS`.
21. P. Thiemann. Towards a type system for analyzing JavaScript programs. In *European Symp. on Programming (ESOP), LNCS 3444*, page 408422, 2005.
22. P. Thiemann. A type safe DOM api. In *DBPL*, pages 169–183, 2005.
23. D. Ungar and R.B. Smith. Self: The power of simplicity. In *Proc. OOPSLA)*, volume 22, pages 227–242, 1987.
24. D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *ACM POPL*, pages 237–249, 2007.