Towards Modular Instrumentation of Interpreters in JavaScript

Florent Marchand de Kerchove Jacques Noyé Mario Südholt

ASCOLA team (Mines Nantes, Inria, LINA)
École des Mines de Nantes, Nantes, France

Abstract

With an initial motivation based on the security of web applications written in JavaScript, we consider the instrumentation of an interpreter for a dynamic analysis as a crosscutting concern. We define the *instrumentation problem* – an extension to the expression problem with a focus on modifying interpreters. We then illustrate how we can instrument an interpreter for a simple language using only the bare language features provided by JavaScript.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords JavaScript, Instrumentation, Interpreter, Expression Problem, Modularity

1. Introduction

Today, JavaScript, with its special connection to HTML5 [6], is considered the language of choice for building web applications. It has made it possible to turn the static pages of the early web into dynamic pages, providing effective interactions with remote sites. This context has created new security challenges, in particular, with respect to information flow security and its dual aspects of confidentiality (local private information should not flow to untrusted remote sites) and integrity (remote sites should not corrupt local information). Although these properties are traditionally enforced through static analysis [12], this new context has created the need for considering dynamic analyses as well as hybrid analyses with various theoretical and practical results [3]. In practice, the corresponding prototypes, when they exist and have been made accessible, turn out to be built each in a very specific way, making it hard to compare and reuse approaches. In particular, the integration of the analysis is often invasive, which means that it is even difficult to relate the principles of the analysis and the implementation or evolve either the analysis or the base JavaScript runtime to which it applies.

This can be seen as a problem of modular instrumentation: what are the basic principles making it possible to modularly compose a JavaScript runtime and a dynamic or hybrid analysis dealing with information flow? In this paper, we study this problem starting from a full-blown case study: the extension of Narcissus, a JavaScript interpreter, written by Mozilla, to *faceted evaluation* [2], a novel and expressive information flow analysis technique able to dynamically handle *implicit* flows (flows depending on conditionals).

On the basis of this case study, we define the four criteria that would enable the modular instrumentation of interpreters. We frame these criteria as the instrumentation problem – an extension to the expression problem (section 2). We then illustrate how to build an interpreter incrementally, and how to instrument it in a way that satisfies the requirements of the instrumentation problem using only simple features of the JavaScript language (section 3). Finally, we discuss how our proposal relates to other approaches to modular instrumentation (section 4).

MODULARITY Companion'15, March 16–19, 2015, Fort Collins, CO, USA
Copyright 2015 ACM 978-1-4503-3283-5/15/03...$15.00
http://dx.doi.org/10.1145/2735386.2736753

2. Instrumenting Interpreters For Dynamic Analyses

Instrumenting an interpreter for a dynamic analysis is straightforwardly achieved by modifying the source code of the interpreter. The dynamic analysis logic may require to extend or disable parts of the existing interpreter code, in several, separate places. When the code of the instrumentation is split across the interpreter code, the concerns of the dynamic analysis and the concerns of the language interpretation become tangled. As a result, extensibility, maintainability and reusability are all decreased; and modular reasoning is lost.

2.1 Case Study: Narcissus Instrumentation for Faceted Evaluation

Narcissus is a meta-circular interpreter for JavaScript, written by Mozilla, that is used for prototyping improvements to the language. Tom Austin and Cormac Flanagan used Narcissus as a base to implement their dynamic information flow analysis, called faceted evaluation [2], in JavaScript. This instrumentation was done on top of Narcissus in a straightforward way, that is, favoring ease of implementation. As a result, getting all the modifications made by the instrumentation requires extracting the difference between two well-chosen commits. If we extract such a diff[1], we can see that the instrumentation touches three files out of seven, and nearly all the changes are in the file concerned with executing the parsed code. Looking more closely at these changes, two patterns are apparent:

1. The addition of a *pc* parameter (Program Counter) at nine distinct locations. In the faceted evaluation strategy, sensible values are wrapped into pairs: one public value and one visible only to a set of principals. The program counter records the branches taken by the program, allowing implicit flows to be tracked. As the program counter is part of the state used by the faceted evaluation strategy, it has to be present in the execution context of Narcissus. Here are two occurrences of this pattern ('<' are lines removed, '>' are lines added):

[1] Extracted from the HEADs of https://github.com/taustin/narcissus and https://github.com/taustin/ZaphodFacets.

```
< function ExecutionContext(type, version) {
> function ExecutionContext(type, pc, version) {

< var x = new ExecutionContext(GLOBAL_CODE,
      Narcissus.options.version);
> var x = new ExecutionContext(GLOBAL_CODE, new
      ProgramCounter(), Narcissus.options.version);
```

2. The addition of tests for *FacetedValue* objects. New cases are created to handle faceted values, and tests are needed to distinguish these values from other runtime values, such as references.

```
< function getValue(v) {
> function getValue(v, pc) {
>   if (v instanceof FacetedValue) {
>     return derefFacetedValue(v, pc);
>   }
    if (v instanceof Reference) { ... }
    return v;
  }

< function putValue(v, w, vn) {
> function putValue(v, w, vn, pc) {
>   if (v instanceof FacetedValue) { ... }
    else if (v instanceof Reference) { ... }
  }
```

The fact that these patterns have multiple occurrences hint at a possible factorization.

Another observation is that all the changes made for the instrumentation are *scattered* throughout the code. Without the diff or prior knowledge of the base interpreter, it is impossible to tell the instrumentation parts from the base interpreter. The dynamic analysis and the interpreter are effectively tangled together.

2.2 The Instrumentation Problem

Based on the observations made in the case study, we define the requirements for a modular instrumentation of interpreters for dynamic analyses. They are:

Modularity The instrumentation should be defined as a module; all its code must be part of this module. In particular, no instrumentation code should appear in the base interpreter source.

Intercession The instrumentation may extend, partially alter or entirely replace the behavior of code present in the base interpreter.

Local state The instrumentation may refer to state that concerns only the dynamic analysis it implements.

Pluggability The instrumentation should be easily activated dynamically. Multiple, non-interfering instrumentations can be activated together without extra effort.

These requirements are not specific to the faceted evaluation analysis, to the Narcissus interpreter or to the JavaScript language. We think they apply broadly to other dynamic analyses, other interpreters, and any language where modules can be defined.

2.3 Relation to the Expression Problem

Extending an interpreter can be seen as an instance of the expression problem coined by Wadler [14]. In fact, the instrumentation problem can be thought of as the expression problem with additional requirements. Wadler defines the expression problem as:

> The Expression Problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts).

The lack of recompilation of existing code can be roughly translated as the *modularity* requirement of the instrumentation problem. Both aim for a clean separation between base code and extension code. The addition of cases and functions can be absorbed into the *intercession* requirement. Static type safety is not applicable in our case, since JavaScript is dynamically typed. More importantly, the expression problem in the context of interpreters is concerned only with *extending* interpreters via additional operations (like a pretty printer), rather than *modifying* existing operations (the evaluation function in particular).

2.4 A Minimal Example

While the instrumentation of faceted evaluation on Narcissus is far from simple, the changes it expresses are all covered by the requirements of the instrumentation problem. To focus on the underlying language mechanisms needed for instrumentation, we exhibit a minimal example. To that effect, we follow Wadler and Odersky [10] and define a language of arithmetic expressions. We show that we can add new data variants and new operations straightforwardly in JavaScript, thereby solving the expression problem. We then illustrate how we can modify existing operations and pass state, thus satisfying the requirements of the instrumentation problem. Along the way, we mention how the examples relate back to faceted evaluation.

3. Building an Interpreter with Modules

In all the following code examples[2], we use a subset of the EcmaScript 6 standard of JavaScript as implemented in Mozilla Firefox 37a, for the shorthand notations reduce syntax noise. All the examples can be translated back to standard EcmaScript 5 supported by all major JavaScript implementation. In the examples, the result of the preceding expression is indicated by the comment syntax '//:'.

3.1 Solving the Expression Problem

The simplest language has terms only for numbers, and an operation to evaluate a number to a numeric value.

We adopt an object-oriented decomposition where data variants are objects, and operations on the datatype are methods of these objects. The following code in JavaScript implements this datatype using objects and prototypical inheritance. The *num* object has two functions: *new* for returning an object that has *this* as prototype and holds the numeric value *n*, and *eval* for returning the numeric value held by such objects. On line 6, *e1* is an instance of the *num* term, and we can call *eval* on it to get its value.

```
1 var num = {
2   new(n) { return {__proto__: this, n} },
3   eval() { return this.n }
4 }
5
6 var e1 = num.new(3)
7 e1.eval() //: 3
```

[2] All the code examples from this paper can be found online at https://gist.github.com/fmdkdd/6f0faa7d105dbbd8c514

3.1.1 Adding a Variant

The 'plus' data variant takes two terms and evaluates to the sum of these terms.

Num	Plus
eval	eval

In our object-oriented implementation, a new data variant is easily added by creating a new object *plus* with an *eval* function. The evaluation of *plus* recursively calls *eval* on the left and right sub-expressions.

```
8  var plus = {
9    new(l, r) { return {__proto__: this, l, r} },
10   eval() { return this.l.eval() + this.r.eval() } }
11
12 var e2 = plus.new(num.new(1), num.new(2))
13 e2.eval() //: 3
```

In the instrumentation of faceted evaluation, *instanceof* tests were added to distinguish a *FacetedValue* from a *Reference*. Both objects are data variants of the datatype of runtime values manipulated by the interpreter. By dispatching on these data variants instead of using *instanceof* tests, we can write the *FacetedValue* like the *plus* object, and extend the interpreter incrementally.

3.1.2 Adding an Operation

We now wish to add another operation to print expressions instead of evaluating them.

Num	Plus
eval	eval
show	show

```
14 num.show = function() { return this.n.toString() }
15 plus.show = function() {
16   return this.l.show() + '+' + this.r.show() }
17
18 e1.show() //: "3"
19 e2.show() //: "1+2"
20 plus.new(num.new(1), num.new(2)).show() //: "1+2"
```

Lines 14 and 15 extend the prototypes of *num* and *plus* with the *show* operation. As we see on lines 18–20, extending the prototypes adds the *show* operation on new expressions (line 20) as well as already-created expressions (*e1*, *e2*). This flexibility is convenient when interactively adapting a running system, where objects cannot be recreated.

Num	Plus	show.Num	show.Plus
eval	eval	eval	eval
		show	show

In some cases however, we would like to preserve the original versions of *num* and *plus*, and create derivatives that can be used alongside the originals. To avoid name collisions, we create a plain object to serve as namespace for the derivatives. To prevent coupling of the derivatives to the originals, we parameterize the namespace on *num* and *plus*. Essentially, we simulate a simple module system: the *show* function takes imports as arguments and returns an object of exports. By constructing our extensions in this way, we satisfy the first requirement of the instrumentation problem: modularity.

```
14 var show = function(base) {
15   var num = {__proto__: base.num,
16     show() { return this.n.toString() }}
17
18   var plus = {__proto__: base.plus,
19     show() { return this.l.show() + '+' + this.r.show() }}
20
21   return {num, plus} }
22
23 e2.show //: undefined
24 var s = show({num, plus});
25 e2.show //: undefined
26 s.plus.new(s.num.new(1), s.num.new(2)).show() //: "1+2"
27 s.plus.new(num.new(1), s.num.new(2)).eval() //: 3
```

As lines 23 and 25 attest, existing expression objects are not affected by the definition or activation of *show*. But on line 26, a new expression is created using the derivatives of *plus* and *num*, and supports *show*.

On line 27, we see that it is possible to mix expression objects that support show, and those that do not: the second is created from *s.num*, while the first one is not. When calling *eval*, an operation that both have in common, this is not an issue. But since JavaScript is not statically typed, nothing would prevent us to call *show* on an expression of mixed languages, and raise an error only at runtime.

It is possible to prevent the writing of expressions of mixed languages by shadowing the bindings for the original *num* and *plus*. By using the *with* construct on an object, all the properties of this object are in scope of the code delimited by *with*. On line 29, *plus* and *num* refer to the derivatives created by the 'show' module. The original *num* and *plus* are shadowed by these bindings; it is impossible to refer to them inside the *with* construct. In addition, the extension of the scope does not persist when exiting *with*, so the *num* and *plus* bindings only exist inside, restricting the activation of the *show* module.

```
28 with (show({num, plus})) {
29   plus.new(num.new(1), num.new(2)).show() //: "1+2"
30 }
```

The *with* construct does not allow us to select which names to import from the *show* module; every property of the object returned by *show* on line 28 is put into scope. If we would rather selectively import a subset of names from the 'show' module, then we can use an immediately-invoked function expression (IIFE) and specify which names we want out as arguments to this function. For instance, here we only import *num*.

```
(function({num}) {
  num.new(1).show() //: "1"
}(show({num, plus})))
```

The last example shows that the *with* construct is not necessary to our scheme, since an IIFE is more expressive. Nevertheless, it provides an interesting use-case for this controversial construct, and since examples using *with* are easier to read, we will stick to it throughout the rest of the paper.

3.1.3 Summary

We have seen how we could:

- add variants to a datatype by defining new objects (3.1.1),
- add operations that act on all the existing variants by extending the prototype of objects (3.1.2),
- without modifying the existing code.

Thereby, we have solved the expression problem as it applies to a dynamic language such as JavaScript.

Furthermore, we have seen that we could add operations to the datatype in two ways: by directly modifying the prototype and affecting all the existing and future objects of the extended data variants, or by creating derivatives of the data variants. With the latter it is possible to mix data variants from separate languages in unsafe ways. We have seen how we can restrict the use of only one language at a time by shadowing and creating new scopes with function expressions and the *with* construct; this dynamic activation satisfies the pluggability requirement of the instrumentation problem.

Interestingly, in all the examples we have seen so far, we used only a fraction of the features provided by the language, without resorting to extra libraries or framework. We develop this point further in section 4.

3.2 Solving the Instrumentation Problem

We have seen how to extend the interpreter for the language of arithmetic expressions by adding variants and operations incrementally as modules – without modifying the extended code. Now we tackle the remaining requirements of the instrumentation problem: intercession and local state.

3.2.1 Modifying Operations

Let us imagine that we want to modify the 'eval' operation on the data variant 'num' to return twice the value it holds.

Num	Plus
eval*	eval

This is straightforwardly achieved in JavaScript, since functions held by objects are like other properties: replaceable by assignment. We used the same mechanism in the addition of *show*. Here we erase the previous definition of *eval* and replace it with the new one.

```
num.eval = function() { return this.n * 2 }

num.new(1).eval() //: 2
plus.new(num.new(1), num.new(2)).eval() //: 6
```

Now we change the specification of our modification to 'num': it should return twice the value of the result of the original 'eval' operation on 'num'. The solution requires the ability to call the previous version of *num.eval*, so we use a function expression to save this version in the closure of the new *eval*. Then we need to call this previous version with the current object as receiver, which is done by invoking the *call* function with the argument *this*.

```
(function(previous_eval){
  num.eval = function() { return previous_eval.call(this) * 2 }
}(num.eval))

num.new(1).eval() //: 2
plus.new(num.new(1), num.new(2)).eval() //: 6
```

Num	Plus	double.Num
eval	eval	eval

Modifying the original *num* object is destructive: the original *eval* is lost, only accessible by the closure created by the IIFE. In this instance, we can create a new data variant that refers to the original *eval* to avoid duplication. To that effect, we create a function parameterized by the original *num* object to serve as a module.

```
31 var double = function(base) {
32   var num = {__proto__: base.num,
33     eval() { return base.num.eval.call(this) * 2 }}
34
35   return {num} }
36
37 with (double({num})) {
38   plus.new(num.new(1), num.new(2)).eval() //: 6
39 }
```

On line 33 we define the new *eval* as a wrapper around the previous functionality. This enables us to compose extensions by passing a modified base as parameter to *double*. The following example illustrate how we can combine such extensions by cascading calls to *with*.

```
with (double({num})) {
  with (double({num})) {
    with (double({num})) {
      plus.new(num.new(1), num.new(2)).eval() //: 24
}}}
```

3.2.2 Passing State to Operations

In the instrumentation of Narcissus for faceted evaluation, the "program counter" is an object that must be passed down to recursive calls of the interpreter, representing state needed by the dynamic analysis. To mimic this behavior, here we add a program counter to the 'eval' operation that is incremented each time a data variant calls *eval*.

Num	Plus	state.Num	state.Plus
eval	eval	eval	eval

The *state* function creates derivatives of the *num* and *plus* data variants that increments the local program counter *pc* when calling *eval*. The value of the program counter is exposed via the exported *getPC* function.

```
40 var state = function(base, pc = 0) {
41   var num = {__proto__: base.num,
42     eval() { pc++; return base.num.eval.call(this) }}
43
44   var plus = {__proto__: base.plus,
45     eval() { pc++; return base.plus.eval.call(this) }}
46
47   var getPC = () => pc
48
49   return {num, plus, getPC} }
50
51 with (state({num, plus})) {
52   getPC() //: 0
53   plus.new(num.new(1), num.new(2)).eval() //: 3
54   getPC() //: 3
55 }
```

On line 52 we see that the program counter is zero, and is later incremented to 3 on line 54 after the evaluation of the expression on line 53.

On lines 42 and 45, the new *eval* functions are defined as wrappers around *base.num.eval* and *base.plus.eval*, which allows for composition of multiple extensions. For instance, we can add the *double* modification of the previous subsection.

```
56 with (double({num})) {
57   with (state({num, plus})) {
58     getPC() //: 0
59     plus.new(num.new(1), num.new(2)).eval() //: 6
60     getPC() //: 3
61 }}
62
```

```
63 with (state({num, plus})) {
64   with (double({num})) {
65     getPC() //: 0
66     plus.new(num.new(1), num.new(2)).eval() //: 6
67     getPC() //: 3
68 }}
```

On line 59 we see that the expression evaluates to 6, and lines 58 and 60 indicate that only three expressions were evaluated. Thus both the *double* modification and *state* modification are in effect. Lines 63–68 attest that the order of activation of these modifications is irrelevant. This is not always the case, as it depends on the semantics of each operation. In this instance, the program counter is a side effect that does not interfere with the doubling of the 'num' data variant, hence they commute.

Finally, we can also add the *show* module without modifying any of the existing definitions. Here again the order of activation of modules is irrelevant.

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

3.2.3 Summary

We have seen how to extend the solution to the expression problem of subsection 3.1 to tackle the additional requirements of *intercession* and *local state*. The solution we exposed, based on simple object modules, satisfy all the criteria of the instrumentation problem we defined in subsection 2.2:

Modularity The *show*, *double* and *state* extensions were all expressed as modules, without any need to alter or duplicate the code of the parts they augment. The *plus* data variant, expressed simply as an object, as well as the first definition of *num*, could have been built as a simple modules with no import.

Intercession The *plus* and *show* examples were two cases of extensions to the base language. The *double* and *state* examples illustrated replacement of the existing *eval* operation. In the *double* case, we first saw a complete replacement of the *eval* functionality. In all cases, the extension or alteration of the base language was done incrementally – without touching the extended code.

Local state The *state* example served to illustrate how we could add state that is local to an extension, without having to pass it as an argument to the modified *eval* operations, and without altering the base code.

Pluggability The *show*, *double* and *state* examples were all easily activated by the *with* construct, or an immediately-invoked function expression. Each time, the *exact same line of code* was used to add 1 and 2, and each time the effect varied depending on the active modules. We have shown how these extensions could all be activated at the same time, without modification needed to the base code or to the client code inside *with*.

In addition, we have illustrated how to solve two patterns we highlighted from the instrumentation of Narcissus for faceted evaluation in subsection 2.1: (1) the addition of *instanceof* tests can be transformed into dispatching on new data variants, and (2) the addition of local state (the program counter) can be encapsulated in module objects.

4. Discussion and Related Work

A combination of simple language features To solve the expression problem and instrumentation problem, we used only features provided by the JavaScript language; we did not extend JavaScript with a class system, a reflection framework or even a module system. Several key features of JavaScript made this possible.

The flexibility of the object system is an advantage here. Objects are essentially dictionaries with the ability to inherit keys from a prototype link. Since objects are open, it is easy to add or modify methods. Objects can also be created at any point at runtime, and this allowed us to define derivatives that delegate to base objects with a prototype link. To distinguish the derivatives from the original objects, we only needed a simple object as a dictionary to serve as namespace.

Our module objects are parameterized by the objects they extend just because they are defined as functions. Functions in JavaScript are first-class values, and can be used as expressions for defining object methods, or the module objects we presented. When activating modules, the shadowing of names in the outer scope was a desirable feature, provided by the *with* statement and functions expressions. In JavaScript, there are no methods strictly speaking: the value of the receiver, accessible inside a function via the *this* keyword, is determined only when a function is called, not when it is defined. We can even specify its value to be an arbitrary object using the *call* construct. We used this feature whenever we needed to invoke an overridden behavior on the current object.

Dynamic typing is also a benefit in this case. The absence of a static type system allows us to compose objects in a way that would be difficult to type in languages like Java, Haskell or Scala. Odersky and Zenger [10] show how they can use Scala to solve the expression problem and retain static type safety, but they do not consider the modification of existing operations. While it is one thing to safely extend interpreters, it is quite another to safely *change* a function signature at some point in runtime. In fact, the requirement of intercession in the instrumentation problem seems at odds with static type safety; but that remains to be shown.

The design philosophy behind our scheme is inspired by Findler and Flatt [5]. They exhibit a solution to the expression problem in MzScheme using mixins and a form of modules called units. Units and mixins follow a single design principle: "specify connections between modules or classes separately from their definitions". In our case, this is translated as adding indirection by names (arguments and namespaces). This loose coupling can be found in our module objects: they are parameterized by the objects they extend, rather than being bound to it prematurely. Late binding is why we can compose module objects in any order in the later examples.

Aspect-Oriented Programming (AOP) The instrumentation of Narcissus for faceted evaluation is a crosscutting concern. AOP is well-suited for expressing such concerns in a modular way, using predicates (pointcuts) to target specific parts of the code (joinpoints) where instrumentation should take over. For instance, FlowR [11] is a library that provides information flow control for the Ruby programming language. The authors used an AOP toolkit to write FlowR and successfully lowered the complexity of their implementation.

AspectScript [13] is an implementation of AOP for JavaScript with advanced scoping strategies, and the ability to capture joinpoints inside function bodies. As it wraps all the client code, not necessarily code that is targeted by pointcuts, AspectScript multiplies the execution time by a factor of 5 to 13. Achenbach and Ostermann define a meta-aspect protocol for developing dynamic program analyses [1], and provide a Ruby implementation.

While we have no doubts that the instrumentation problem we consider can be solved using AOP, the scheme we present here shows

that the flexibility of a few JavaScript constructs are sufficient. By leveraging the right parts of the language, we are able to define extensions as modules without having to refer to a sophisticated model.

Context-Oriented Programming (COP) Our way of activating modules via *with* is reminiscent of the activation of layers in COP systems. ContextJS [7] is a COP extension to JavaScript, enabling sideway composition of layers on a base class. In ContextJS, a layer refines a class with partial methods that override the behavior of methods of the base class; extra methods can be added as well. Layers can then be activated and deactivated dynamically, on all or specific instances of the class, for a delimited extent or globally. Layers can stack, and partial methods can refer to the behavior of upper layers indirectly by using the *proceed* special argument, allowing the gracefully composition of layers similar to our final examples in 3.2.2.

We can view our module objects as rudimentary layers. Our composition of modules could be improved syntactically by using a *proceed* keyword. The layer activation possibilities of ContextJS are more expressive, but they go beyond the needs of the instrumentation problem we defined. Our module objects are parameterized by the base object they extend, and as such are not tied to a particular class. The main difference is, again, that the simplicity of our scheme lies in combining the raw parts of the JavaScript language, and does not require external definitions.

Other approaches to modular instrumentation Marek et al. propose DiSL [8], a Java framework for writing dynamic program analyses using the AOP model of pointcuts and joinpoints, focusing on ease of instrumentation and efficiency of weaving. Polyglot [9] is an extensible compiler front-end for Java used notably for implementing the Jif language [4], which extends Java with information flow control functionality. These approaches share our goals of extending languages with dynamic analyses while preserving modular reasoning. However, they target the Java language, which apart from inspiring the name "JavaScript", bear little resemblance to the language and platform we address.

Modular instrumentation of Narcissus The remaining question is: how does the solution sketched here apply to the instrumentation of Narcissus for the faceted evaluation analysis? Our examples apply to the interpreter for arithmetic expressions that was ostensibly designed for extensibility. The implementation of Narcissus does not follow this clear-cut object-oriented decomposition of data variants and operations, instead favoring switch-cases and *instanceof* tests. Refactoring Narcissus is clearly needed as a first step to allow extensions to be written. The refactoring effort should be moderate, as the interpreter part of Narcissus is only 1500 lines, and the rest of the program is already split into modules. Once Narcissus is in a form more amenable to extension, we expect the scheme presented here to allow the modular expression of faceted evaluation and other dynamic information flow analyses.

Furthermore, since the instrumentation problem we defined is not particularly tied to Narcissus, we expect our solution to be applicable to other interpreters written in JavaScript.

Acknowledgments

This work has been partially funded by the SecCloud project of the French "Laboratoire d'Excellence" CominLabs and the Inria associated team REAL. We thank Nicolas Papredi for insightful discussions on and around the topic of modular instrumentation.

References

[1] M. Achenbach and K. Ostermann. "A Meta-Aspect Protocol for Developing Dynamic Analyses". In: *Runtime Verification*. Lecture Notes in Computer Science 6418, pp. 153–167. DOI: 10.1007/978-3-642-16612-9_13.

[2] T. H. Austin and C. Flanagan. "Multiple Facets for Dynamic Information Flow". In: *POPL'12. Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2012, pp. 165–178. DOI: 10.1145/2103656.2103677.

[3] N. Bielova. "Survey on JavaScript security policies and their enforcement mechanisms in a web browser". In: *The Journal of Logic and Algebraic Programming* 82.8 (2013), pp. 243–262. DOI: 10.1016/j.jlap.2013.05.001.

[4] S. Chong, A. C. Myers, K. Vikram, and L. Zheng. *Jif Reference Manual*. 2009. URL: www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html.

[5] R. B. Findler and M. Flatt. "Modular Object-Oriented Programming with Units and Mixins". In: *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. 1998, pp. 94–104. DOI: 10.1145/289423.289432.

[6] I. Hickson, R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O'Connor, and S. Pfeiffer. *HTML5 - A Vocabulary and associated APIs for HTML and XHTML*. W3C Recommendation. Oct. 2014. URL: http://www.w3.org/TR/2014/REC-html5-20141028/.

[7] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. "An open implementation for context-oriented layer composition in ContextJS". In: *Science of Computer Programming* 76.12 (2011), pp. 1194–1209. DOI: 10.1016/j.scico.2010.11.013.

[8] L. Marek, Y. Zheng, D. Ansaloni, L. Bulej, A. Sarimbekov, W. Binder, and P. Tuma. "Introduction to dynamic program analysis with DiSL". In: *Science of Computer Programming* 98 (2015), pp. 100–115. DOI: 10.1016/j.scico.2014.01.003.

[9] N. Nystrom, M. R. Clarkson, and A. C. Myers. "Polyglot: An Extensible Compiler Framework for Java". In: *Compiler Construction, 12th International Conference, CC 2003*. 2003, pp. 138–152. DOI: 10.1007/3-540-36579-6_11.

[10] M. Odersky and M. Zenger. "Independently Extensible Solutions to the Expression Problem". In: *Proceedings of the 12th International Workshop on Foundations of Object-Oriented Languages (FOOL'05)*. Jan. 2005. URL: homepages.inf.ed.ac.uk/wadler/fool/program/10.html.

[11] T. F. J.-M. Pasquier, J. Bacon, and B. Shand. "FlowR: Aspect Oriented Programming for Information Flow Control in Ruby". In: *MODULARITY'14. Proceedings of the 13th International Conference on Modularity*. 2014, pp. 37–48. DOI: 10.1145/2577080.2577090.

[12] A. Sabelfeld and A. C. Myers. "Language-based information-flow security". In: *IEEE Journal on Selected Areas in Communications* 21.1 (Jan. 2003), pp. 5–19. DOI: 10.1109/JSAC.2002.806121.

[13] R. Toledo, P. Leger, and É. Tanter. "AspectScript: expressive aspects for the web". In: *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD'10)*. 2010, pp. 13–24. DOI: 10.1145/1739230.1739233.

[14] P. Wadler. *The Expression Problem*. 1998. URL: homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt.