---------- REVIEW 1 ----------
PAPER: 36
TITLE: Open scope: a pattern for modular instrumentation
AUTHORS: Florent Marchand de Kerchove, Jacques Noyé and Mario Südholt

OVERALL EVALUATION: 0 (borderline paper)
REVIEWER'S CONFIDENCE: 4 (high)

---------- REVIEW ----------
This paper proposes a pattern for instrumenting JS code. The pattern applies
to Narcissus, a JS interpreter written in JS itself. It could perhaps be applied to other meta-circular implementations of JS, under the condition that
those alternative implementations are also based on the module pattern.

The whole purpose of this work is to be able to modify the original interpreter in the less possible intrusive way, and this appears to be the
case, so I find this work interesting, and since it is self-contained and properly grounded into existing work, it is certainly worthy of publication.

Unfortunately, I have two objections against publication in its current form;
one on the form, one technical.

1. Section 3 is trivial and should be reduced to something like 10% of its
original size. It's essentially a lecture on lexical scoping and closures,
which I think nobody needs in this conference. The only part that needs to
remain is the workarounds you suggest.

Also, I don't see the need for both the Narcissus and the simplified examples. They are redundant. And finally, you contradict yourself in this
section (although I understand what you mean), claiming that direct access to
the definitions in the global scope is a "downside", but later on introducing
the "scope" property which essentially does that on your non-global scopes.
This should be rephrased.

2. The second problem I have is more technical. You propose the use of "with"
to open scopes. It works nicely on your simple examples, but I hope you're
aware that there's a lot of controversy in the usage of with [OK, now I see
that you are on page 11]. It's broken by design (Cf. your own explanation of
listing 6), ambiguous, not forward-compatible etc., to the point that it's

even considered deprecated and even forbidden in script mode. A typical forward-compatibility problem with it is JS adding new built-in properties on
objects that would entail name clashes in client code.

Because of the fragility of this construct, I fear that upcoming versions of
Narcissus (or JS itself, since Narcissus in written in it) could break your
instrumentation facility in unexpected ways. So IMO, a proper study of those
risks is mandatory for your work to be properly justified. At the very least,
those risks need to be clarified, well understood and properly circled.

Less important remarks:

I think you exaggerate a bit the code duplication argument, especially the
"twice the amount of code" part. VCSes tools with diff, branch and merge commands make this much less painful than you make it sound.

The simplicity argument is also somewhat bizarre: "The mechanisms used to achieve the modular instrumentation [..] should be at most as complex as the
analyses themselves". Why exactly? And define complex in this context. This
kind of sentences doesn't really mean anything.

Minor remarks:

- Section 2, introductory paragraph: "sketch AN THE ideal solution".

- Listing 5 line 8: pushScope S2 (not S).

- Section 4 bottom of page 7: "a manipulation scopeS".

- Very end of section 4: "to the THE four requirements".

---------- REVIEW 2 ----------
PAPER: 36
TITLE: Open scope: a pattern for modular instrumentation
AUTHORS: Florent Marchand de Kerchove, Jacques Noyé and Mario Südholt

OVERALL EVALUATION: -2 (reject)
REVIEWER'S CONFIDENCE: 4 (high)

---------- REVIEW ----------
Summary:

The paper studies how to allow the implementation and deployment of programming program analyses for JavaScript (focusing on information flow analyses). It first identifies four requirements to achieve a modular

instrumentation of an interpreter and then proposes an open scope pattern as a technique to instrument a JavaScript interpreter for information flow analyses. The technique is applied to the Narcissus interpreter and two analyses are implemented: faceted evaluation and taint analysis.

General Comments:

The topic of the paper is interesting, but the paper does not seem to make a significant new contribution. The paper reads more like a good tutorial on how to exploit scopes to change the semantics of function calls in the interpreter, and how to apply this to JavaScript. Readers not familiar with evaluation contexts and scoping will find that the paper provides a didactic introduction to the concepts, but familiarised readers will find the paper rather dull.

I can see that compared to directly modifying the JavaScript interpreter, the propose technique provides a modular instrumentation for dynamic program analyses. However, I am not convinced that the proposed solution provides a more effective foundation than the JavaScript reflection API [1] [2]. To be convincing, the paper should compare the open scope pattern to the proxy pattern already present in JavaScript and show that it can implement a wide range of analyses.

The paper needs to appropriately discuss relevant recent related work to allow dynamic program analyses in Javascript as well (see related work pointers in the detailed comments).

Detailed Comments:

The paper is mainly based on the assumption that the current practice for implementing dynamic programming analysis for JavaScript is to alter the interpreter, and it does not consider at all the use of reflection to this end. However, JavaScript reflection API adheres to the modular instrumentation requirements identified in section 2.2, and many recent work employs JavaScript proxies as underlying technique on which to build dynamic programming analysis like taint analysis [3], access permission contracts [4], etc . Alternatively, one could employ programming instrumentation platforms for JavaScript like Jalangi [5] [6] which would also adhere to the four requirements identified by the paper.

The paper argues that having access to the bindings of the module is sufficient for instrumenting the interpreter. That may be the case, but the technique seems too limited for information flow analyses. It would be nice that the evaluation section would have shown the use of the technique beyond information flow analyses to prove that it is able to ease the development of program analysis like access control. To show the strengths of the technique, the paper could also compare the code employing the open scope pattern, with the code for the same analysis employing the proxy pattern (offered by the Javascript reflection API) and show that their technique eases the writing of analysis(by comparing code quality/quantity), and/or incurs in less performance overhead.

Section 2.1. is far too long and at this point in the paper the reader only needs to know that modifying the source code of the interpreter is not a scalable technique because it leads to code duplication, and the instrumentation code is scattered and entangled with the interpreter itself. All the code details on the changes required to Narcissus for

facet instrumentation without the open scope pattern would be better described later in the evaluation section.

Section 2.2. indicates that the solution for a modular instrumentation "do not necessary know the points of extension required by the analysis in the interpreter" and that "any part of the interpreter can be changed by instrumentation". However, as far as I understand, in order to use the open scope pattern, one has to go to the interpreter and inject code at the concrete points (requiring prior knowledge of the interpreter), so it is difficult to see that the proposed solution upholds the hypotheses.

Section 3.1. is far too detailed even for unfamiliar readers. Figure 3 does not seem to correspond to the code of listing 2 which does not employ a g function whose body is f(x), and the activation of mkG does not bind x to 0, since it basically creates a new closure which is bound to g, rather than applying it.

In section 4, the text that explains figure 7 is difficult to follow. I do not see any binding object for the case that with is empty as said in the text.

It would have made the paper stronger if section 5 would have provided details on the case study for taint analysis.

References:
[1] T. V. Cutsem and M. S. Miller. Proxies: design principles for robust object-oriented intercession APIs. In W. D. Clinger, editor, DLS, pages 59–72. ACM, 2010. ISBN 978-1-4503-0405-4.
[2] T. V. Cutsem and M. S. Miller. Trustworthy Proxies Virtualizing Objects with Invariants, ECOOP 2013.
[3] T. H. Austin, T. Disney, and C. Flanagan. Virtual values for language extension. In C. V. Lopes and K. Fisher, editors, OOPSLA, pages 921– 938, Portland, OR, USA, 2011. ACM. ISBN 978-1-4503-0940-0.
[4] M. Keil and P. Thiemann. Efficient Dynamic Access Analysis Using JavaScript Proxies, DLS 2013.
[5] Koushik Sen and Swaroop Kalasapur and Tasneem Brutch and Simon Gibbs, "Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript," in 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13), ACM, 2013 .
[6] jalangi 2: http://www.cs.berkeley.edu/~ksen/

---------- REVIEW 3 ----------
PAPER: 36
TITLE: Open scope: a pattern for modular instrumentation
AUTHORS: Florent Marchand de Kerchove, Jacques Noyé and Mario Südholt

OVERALL EVALUATION: 1 (weak accept)
REVIEWER'S CONFIDENCE: 4 (high)

---------- REVIEW ----------
This paper is kind of interesting. It spends a huge amount of time discussing the basics of lexical scoping. It introduces environment diagrams that are very similar to the ones in John Mitchell's Concepts in Programming Languages (http://theory.stanford.edu/people/jcm/books.html) and nearly identical to the environment diagrams used in Cook's new

book Anatomy of Programming Languages (
http://www.cs.utexas.edu/~wcook/anatomy).

The pattern is either elegant JavaScript hacking or severe abuse of modularity. I can't tell which! I guess I like it. It illustrates how using language constructs to implement your own module system, rather than having a built-in one, enables it to be modified to allow extensibility.

The question I have is what happens when the module being instrumented isn't modularized in the right way, so that changes need to be made inside the methods of the module, not at their interface boundaries? You should mention this assumption as a severe potential problem with this approach. It is a well-known issue with classes.

"an the" -> "the"