---------- REVIEW 1 ----------
PAPER: 4
TITLE: Towards Modular Instrumentation of Interpreters in JavaScript
AUTHORS: Florent Marchand de Kerchove, Jacques Noyé and Mario Südholt

---------- REVIEW ----------
This paper describes a pattern for implementing interpreters in JavaScript, such that instrumentation can be performed on them in a modular way, without having to change the interpreter code.

Overall, the paper is well written and is easy to read. It provides a good introduction to the world of interpreter instrumentation.

Reading through this paper I got the feeling it should end with a description of a refactored version of the Narcissus interpreter and a modular implementation of faceted evaluation. Instead, the paper ended with a discussion on what it should take to implement such artifacts. I guess this is OK for a workshop paper, but I would definitely encourage the authors to complete this.

One thing that bothered me with the pattern described by the authors is in the central role the "with" keyword takes. The "with" keyword is powerful, as it allows users to modify the lexical scoping of a block. However, it is also deprecated. ES5 strict mode does not allow the use of "with", and this means code that uses "with" in the real world, lives on borrowed time. The main reason for that (according to Brendan Eich) is security (quoting his tweet on the subject):

@angusTweets no, optimization is not the issue. with violates lexical scope, making program analysis (e.g. for security) hard to infeasible.

Interestingly, security (and in particular, dynamic analysis) is the motivation behind this paper, so I wonder how well an interpreter designed using the pattern described in the paper would be successful analyzing its own code (it is, after all, metacircular).

In Section 4, the authors claim that the dynamic typing in Javascript allowed the authors do things that would otherwise be "difficuls or even impossible to do". I tend to disagree. I've seen some heroic compositions done in languages such as Scala and Haskell. Unlike the original instrumentation, that modified the signature of methods, the pattern described in this paper only decorates functions and objects with additional functionality. When an extra parameter needs to flow down (such as the PC), it is done through a closure. The eval() method's signature was never changed. I would suggest that the authors lower the tone of this paragraph, only claiming it was easier this way.

Some point of personal taste maybe, but I did not like the use of Mozilla-specific dialect in the code examples. They are unfamiliar to most readers (as developers are discouraged from using them due to compatibility issues), and may give the (wrong) impression that this pattern can only work in this dialect of JS. Although less elegant, I would suggest using standard EcmaScript for the code examples in the paper.

---------- REVIEW 2 ----------

PAPER: 4
TITLE: Towards Modular Instrumentation of Interpreters in JavaScript
AUTHORS: Florent Marchand de Kerchove, Jacques Noyé and Mario Südholt

---------- REVIEW ----------
This paper describes the instrumentation problem, which is an extension of the expression problem defined by Wadler. They define four requirements for modular instrumentation of interpreters to implement dynamic analyses (modularity, intercession, local state, and pluggability). The then look at a case study in the Narcissus interpreter by extending it support faceted evaluation.

The authors explicitly state they don't believe their approach works in a static typed language (like Java), due to the requirement of intercession. This seems reasonable to argue. They also state at the end they believe their approach would generalize to other JS interpreters, which again seems reasonable. My question is do the authors think their approach generalizes to any other dynamically typed languages (say, Python)? What is the exact set of features a language must contain in order for this approach to be applicable? The authors heavily hint at such features in Sec. 4 - I'd like to see it made more explicit.

I do not believe the authors are quite correct in assuming that an AOP solution to the problem would suffer "performance penalties of dynamic weaving." Previous works have shown that simple caching mechanisms can avoid most overhead for dynamic weaving.

Minor typos:
- abstract: 'of an an interpreter'
- 3.1.2 pg 3 - 'to print expression*S* instead of evaluating'
- 4 pg 5 - last word - 'we present here show*S*'