Tearing Down the Multicore Barrier for Web Applications

ACM Student Research Competition at MODULARITY: AOSD 2012

Jens Nicolay

Software Languages Lab
Vrije Universiteit Brussel
jnicolay@vub.ac.be

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis – Operational semantics

General Terms Algorithms, Experimentation

Keywords concurrency, parallelism, modularity, JavaScript, refactoring, static analysis

1. Background and Motivation

We are being confronted with two phenomena that will greatly influence the way people experience the internet. On one hand the browser, with JavaScript as embedded programming language, is getting more important as application delivery platform. On the other hand we are confronted with the multicore revolution. Both desktop machines and mobile devices are already equipped with processors that contain multiple cores. Applications that take advantage of the underlying parallel hardware are able to execute tasks in parallel, which benefits the performance and responsiveness of those applications. Despite the fact that many desktop applications already make use of multiple cores during their execution, the multicore revolution has mostly ignored web applications, leaving the potentential processing power on the client-side virtually unused. To keep up with the demand for ever increasing performance coupled to acceptable response times, browser applications will have to make use of multiple cores during execution. This is possible by designing them with concurrency in mind.

Web Workers[1] is a standard JavaScript API that makes it possible to add actor-like concurrency to an application. However, JavaScript applications have to be structured in a certain way in order for them to make efficient use of multiple cores at runtime. To avoid the situation where this task falls entirely on the shoulders of developers, decent tool support for concurrent web programming is absolutely necessary.

[1] `http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html`

AOSD'12, March 25-30, 2012, Potsdam, Germany.
ACM 978-1-4503-1222-6/12/03.

2. Goal and Research Problems

The goal of my research is to support the development of web applications that effectively make use of multicore systems on the client-side. This makes it possible to develop web applications that require a great deal of processing power without sacrificing performance and responsiveness. More specifically, I target web developers that build concurrent JavaScript applications using Web Workers. Supporting these developers includes performing on-the-fly detection of possible concurrency errors, and offering a collection of refactorings that deal with concurrency.

Refactoring relies on static analysis to determine non-trivial program properties without actually executing the program first. Support for the development of JavaScript programs that use Web Workers is nonexistent, because there currently exists no static analysis that handles concurrent JavaScript programs. Static analysis for JavaScript is a very active research area, but the language is considered to be "harsh terrain" for static analysis because of its dynamic features and extreme permissiveness [1, 7].

Another research problem I want to address is the identification of useful source code patterns and refactorings for JavaScript that are specifically geared towards concurrency. At present we do not have a catalog of such concurrency patterns and refactorings, let alone for the more general case.

Finally, I want to clearly and precisely specify the preconditions and mechanisms of those source code patterns and refactorings.

3. Approach and uniqueness

To address the issues raised above, I will start by compiling a catalog of useful refactorings related to concurrency and modularity. These refactorings analyze interactions between

entities in order to modularize them, and try to identify candidate-workers for example. I also want to detect certain patterns in the source code to be able to provide feedback to the programmer during development.

I will precisely specify preconditions and source code patterns by designing a set of general and reusable queries that enables one to reason about properties of concurrent JavaScript programs. The outcome of those queries will be based on the results of a sufficiently precise, powerful and fast static analysis of JavaScript programs that use Web Workers.[2] Dependence analysis plays an important role when reasoning about concurrency. Dependent expressions must be executed in a fixed order to preserve the sequential semantics of a program in which they appear. Parallel execution of these expressions, however, would destroy any such ordering guarantee. While some types of dependence are lexically apparent, procedure invocations in a higher-order, object-oriented setting may give rise to interprocedural dependencies that are not always evident to track.

The most innovative aspect of my research consists of joining two separate research paths, being (i) static analysis of JavaScript programs, and (ii) static analysis of actor languages. I will validate my research by maintaining a large corpus of examples to demonstrate that my results are sound. For additional validation and dissemination, I intend to implement and experiment with an Eclipse plugin specifically aimed at developing concurrent JavaScript applications.

4. Related work

Research on static analysis of JavaScript emerged shortly after the advent of Web 2.0 and acceptance of the language as a "serious" programming language. For example, TAJS [2] is a capable JavaScript analysis that has recently been added as part of the Eclipse JSDT plugin [1] to enable refactorings like RENAME and EXTRACT MODULE. DoctorJS[3] is a collection of tools for JavaScript, with an underlying CFA2 pushdown analysis [8] that focuses on precision rather than speed. None of these – and other tools and plugins – are capable of handling JavaScript programs that contain Web Workers.

My earlier work on automatic parallelization of Scheme programs required the implementation of a dependence analysis in a higher-order, functional setting [6]. I can reuse my results for the analysis of the functional part of JavaScript.

Wrangler [4] is an interactive refactoring tool for actor language Erlang that can be integrated in Emacs and Eclipse. It offers refactorings that are interesting in the context of my research. MOVE FUNCTION TO ANOTHER MODULE for example attempts to move a function definition from its current module to another module specified by the user. If the necessary preconditions are met, it will not only update the source and target modules but also all references to this function in the program [3].

5. Results and contributions

From a high-level perspective my research aims to advance the design, the implementation and the application of programming languages and their environments to support the software engineering life-cycle. More specifically there are three technological domains that my research focuses upon: parallel programming, distributed programming, and program analysis and transformation. I want to make meaningful contributions in each of these domains, more specifically:

- the design and implementation of an interprocedural dependence analysis for JavaScript,
- the development of a set of reusable queries to reason over concurrent JavaScript programs, and
- the compilation of a catalog of useful JavaScript refactorings that specifically deal with concurrency and help with modularization.

References

[1] A. Feldthaus, T. Millstein, A. Møller, M. Schäfer, and F. Tip. Tool-supported refactoring for JavaScript. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2011.

[2] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS)*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[3] H. Li and S. Thompson. Clone detection and removal for erlang/otp within a refactoring environment. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 169–178. ACM, 2009.

[4] H. Li, S. Thompson, G. Orosz, and M. Tóth. Refactoring with Wrangler, updated: data and process refactorings, and integration with eclipse. In *Proceedings of the Seventh ACM SIGPLAN Erlang Workshop*, page 12pp. ACM Press, 2008.

[5] M. Might and O. Shivers. Exploiting reachability and cardinality in higher-order flow analysis. *Journal of Functional Programming*, 18(5-6):821–864, 2008.

[6] J. Nicolay, C. D. Roover, W. D. Meuter, and V. Jonckers. Automatic parallelization of side-effecting higher-order scheme programs. In *Proceedings of the Eleventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2011)*, Williamsburg, VA, USA, September 2011.

[7] G. Richards, C. Hammer, B. Burg, and J. Vitek. The eval that men do. *ECOOP 2011–Object-Oriented Programming*, pages 52–78, 2011.

[8] D. Vardoulakis and O. Shivers. Pushdown flow analysis of first-class control. In *ACM SIGPLAN Notices*, volume 46, pages 69–80. ACM, 2011.

[2] Precision refers to how tight an analysis constrains a set of answers to the set of actually possible answers, while power refers to the class of questions an analysis can answer. [5]

[3] `https://github.com/mozilla/doctorjs`