Rewriting JavaScript Module System

[Extended Abstract]

Junhee Cho
KAIST
ssaljalu@kaist.ac.kr

ABSTRACT

Although JavaScript is one of the major languages used for web and other general applications, it does not have a language-level module system. Lack of module system causes name conflicts when programmer uses libraries. We introduce a JavaScript module system with formal semantics. As an implementation, we also introduce a source-to-source transformation from JavaScript with module to JavaScript itself for current JavaScript engines to use them.

Categories and Subject Descriptors

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

Keywords

JavaScript, module system, source-to-source transformation

1. INTRODUCTION

JavaScript [1] is the most prevalent client-side scripting language for the web. It enriches web documents with dynamic reaction to user's action. The primary method for a web document to react to a user's action is to reload another web document on the web browser triggered by a click on a link. For instance, in a small messanger widget nested in a web document, however, it is not necessary to reload the whole web document frequently for each message. With JavaScript, one can reload only the small widget by communicating with the server through HTTP and updating the web document with Brower API for Document Object Model (DOM) [2]. The fact that 98 out of the 100 most visited websites use JavaScript for client-side programming according to Alexa [3] presents its share on the market that cannot be ignored. Evenmore, JavaScript has been used outside client-side programming for the web; for example, node.js [4] for a general scalable network application, and Samsung SmartTV SDK [5] for a SmartTV app.

Nevertheless most of JavaScript program use libraries such as jQuery [6] or Prototype [7], JavaScript does not provide any language-level module system, which can cause a name conflict if distinct libraries using a common symbol are inserted in the same context. Since both jQuery and Prototype use the symbol `$`, if both libraries are inserted in a HTML document by `<script>` tags, the one inserted later overrides the other for the symbol `$`. In substitution for module system, module pattern is recommended. However, since it is not a system on language level, it is hard to guarantee properties on module system.

Because of the needs for language-level module system, ECMAScript Harmony [8], a proposal for the next generation of JavaScript, introduces a language-level module system. Since ECMAScript Harmony is still a work in progress, it does not provide complete module semantics, but part of module semantics in high-level description in prose. Kang *et al.* [9] introduced the formal specification and the implementation of a module system based on the module system in ECMAScript Harmony by the means of desugaring rule from JavaScript with module to λ_{JS} [10], a core-calculus of JavaScript. Unfortunately, due to the innate limitations of λ_{JS}, the module system introduced is impractical. First of all, desugared λ_{JS} program is very large, and interpretation of program involves a large amount of memory usage. Because the main objective for λ_{JS} is to prove properties on JavaScript easily, interpreting a general JavaScript program with λ_{JS} is nearly impossible. λ_{JS} is a purely functional language. Thus, it consumes a lot of memory for immutable objects while interpreting a program. Secondly, λ_{JS} does not support `eval` function which generates code dynamically in run-time while use of dynamic features is evident in websites [11, 12, 13]. In the web environment, most of data is encoded in either XML or JSON. While we need XML parser to parse XML string, we do not need JSON parser to parse JSON string because JSON string is JavaScript expression and evaluating JSON string with `eval` function gives JSON object. Likewise, though we can avoid use of `eval` function, it is widely used in practice due to its convenience.

2. JAVASCRIPT MODULE SYSTEM

We introduce the formal specification and the implementation of a module system based on the aforementioned previous works. The module system is based on Scalable Analysis Framework for ECMAScript (SAFE) [14]: the formal specification is based on the formal specification of SAFE Intermediate Representation (IR), and the implementation is done by adding module rewriter between SAFE Parser and SAFE Interpreter in the interpretation pipeline. The formal specification provides formal module semantics by itself and formal rewriting rule from JavaScript with module to JavaScript itself. The implementation provides the module rewriter implementing the rewriting rule. These are available at SAFE repository.

A module can be declared only in the global context and in

AOSD'13 Companion, March 24-29, 2013, Fukuoka, Japan.
ACM 978-1-4503-1873-0/13/03.

module bodies. Before evaluate program, module environment is constructed statically. It holds all the names in the global object and the modules, and all the export-import relations. In the global context, function, variable, module, and import declarations, and statements are evaluated in the order. A module declaration introduces new scope called module scope, and the module body is evaluated in the module scope. Also, it results module instance object with getters for the exported names in the module. Module declarations are evaluated by instantiating all the modules, making all the module instance objects read-only, and initializing all the modules. Instantiating a module is to construct the module scope and the module instance object. For mutually recursive imports, function, variable, and nested module declarations are evaluated in the module scope, and the getters for the exported names are set in the module instance object. The next step is to seal the module instance objects, i.e., to make them read-only. Finally, initializing a module is to evaluate the statements in the module body in the module scope. Also, each imported name is substituted by the canonical name of which the imported name is alias.

To rewrite the module system, module pattern is used. In JavaScript, any function can be used as a constructor in the new statement. A module declaration is rewritten to a new statement with a function as a constructor. The function scope is used as the module scope, and the newly created object bound to `this` is used as the module instance object. First of all, the function instantiates the module, i.e., the function body consists of the function, the variable, and the nested module declarations, and statements to set getters for the exported names in the module body. To come back to the function scope from the outside after instantiation, the function also set a temporary closure for initialization in the global context with a random fresh name. Then, the module instance objects are sealed by `Object.seal`. Now, calling the closure initializes the module, i.e., the statements of the module body is evaluated in the function scope. Finally, the closures are deleted from the global context.

The significant difference from Kang *et al.* is that JavaScript with module is translated to JavaScript itself instead of λ_{JS}. Additionally, the translation is source-to-source translation preserving the name space with limited support for `eval` function. Since the translation is source-to-source translation, it is able to utilize any current JavaScript engine with the module rewriter to interpret a program in JavaScript with module. For instance, programmers may develop their JavaScript program with module, and translate it to the current JavaScript program without module using the module rewriter so that any current JavaScript engine can run the program. JavaScript community tends to accept an advance of the specification slowly. There is a time gap of couple of years for them to accept new JavaScript specification. In the mean while, the module rewriter would be a substitution until JavaScript engines accept the module system. Besides, the module rewriter preserves the name space except during compiling modules. For the `eval` function calls after compiling modules, the semantics are preserved. However, the temporary helper functions have a chance of name conflict with new names that the `eval` function calls introduce during compiling modules. Though, by choosing long random strings of complicated characters as fresh names for the helper functions, for example, with non-Latin alphabets, we can reduce the possibility of name conflict.

3. CONCLUSION

In summary, we introduced a JavaScript module system based on ECMAScript Harmony and Kang *et al.* providing the formal specification of module semantics and module rewriter, and the implementation of module rewriter. The module rewriter is source-to-source translator with limited support of `eval` function. As a future work, to guarantee the validity of module environment and translation by formal proof takes the highest priority. Proving the validity of module environment might be analogous to Kang *et al.* because the module environment is analogous to that of them. Proving the validity of translation is more challenging because nothing has been proven for SAFE IR while some properties of λ_{JS} have been already proven like the safety.

Acknowledgments

This work is supported in part by the National Research Foundation of Korea (NRF) (Grants 2012-0000469 and 2012-0005256), Microsoft Research Asia, Samsung Electronics, and S-Core.

References

[1] ECMA. *ECMA-262: ECMAScript language specification*. 5th edition, 2009.

[2] W3C. DOM. `http://www.w3.org/DOM/`.

[3] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, Ryan Berg. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 20th ISSTA*, 2011.

[4] Joynet. Node.js. `http://nodejs.org/`.

[5] Samsung Electronics. Samsung Smart TV. `http://developer.samsung.com/smarttv`.

[6] jQuery Foundation. jQuery. `http://jquery.com/`.

[7] Prototype Core Team. Prototype. `http://prototypejs.org/`.

[8] Dave Herman and Sam Tobin-Hochstadt. Harmony Proposals – Modules. `http://wiki.ecmascript.org/doku.php?id=harmony:modules`.

[9] Seonghoon Kang and Sukyoung Ryu. Formal specification of a JavaScript module system. In *Proceedings of OOPSLA*, 2012.

[10] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of Javascript. In *Proceedings of the 24th ECOOP*, 2010.

[11] Gregor Richards, Christian Hammer, Brian Burg, Jan Vitek. The eval that men do: a large-scale study of the use of eval in JavaScript applications. In *Proceedings of the 25th ECOOP*, 2011.

[12] Gregor Richards, Sylvain Lebresne, Brian Burg, Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of PLDI*, 2010.

[13] Paruj Ratanaworabhan, Benjamin Livshits, David Simmons, and Benjamin Zorn, JSMeter. Comparing the behavior of JavaScript benchmarks with real web applications. In *Proceedings of the USENIX 2010 Conference on Web Application Development*, 2010.

[14] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript. In *Proceedings of FOOL*, 2012.