Certified Desugaring of Javascript Programs using Coq

Experience Report

Marek Materzok
INRIA, University of Wrocław
marek.materzok@inria.fr

Abstract

JavaScript is a programming language originally developed for client-side scripting in Web browsers; its use evolved from simple scripts to complex Web applications. It has also found use in mobile applications, server-side network programming, and databases.

A number of semantics were developed for the JavaScript language. We are specifically interested in two of them: JSCert and λ_{JS}. In order to increase our confidence that the two semantics correctly model JavaScript, we try to relate them formally using Coq. The size and complexity of the two semantics makes this a complex problem with many obstacles to be overcome.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

Keywords Coq, JavaScript, ECMAScript, mechanized semantics

1. Introduction

The JavaScript language is standardized by ECMA under the name ECMAScript. Several versions of the standard now exist, the latest two being ECMAScript 5.1 (ES5, released in 2011) and ECMAScript 6 (ES6, released in 2015). Its role is to make portability between different JavaScript implementations possible by specifying the intended behavior of the various language constructs and built-in functions of JavaScript. The specification also makes it possible to develop formal semantics for the language, which can then be used for, e.g., developing static analysis tools.

One such semantics is JSCert [1], which has the goal of formalizing the ECMAScript (version 5.1) specification in the form of a pretty-big-step semantics [2] expressed in the form of a Coq inductive predicate. It aims to be as close to the (informal English) specification as possible by maintaining close correspondence between the evaluation rules and steps in the ES5 pseudocode. The project also includes JSRef, a reference interpreter for JavaScript written in Coq and proven correct with respect to JSCert. The interpreter is also tested using the official ECMA test suite, test262. Thanks to the correctness proof, testing of JSRef increases the confidence we can have that JSCert correctly models ES5.

Another semantics we are interested in is λ_{JS}. It aims to capture the semantics of JavaScript by defining a simple core language (based on the untyped call-by-value lambda calculus), and then transforming (or "desugaring") the full JavaScript language into this simple core. The λ_{JS} semantics was originally developed to model the ECMAScript 3 specification [3], and it was later revised [5] to handle ECMAScript 5.1 features[1]. The interpreter for the core language and the JavaScript-to-core transformation were developed using OCaml[2]; the interpreter was tested using the official test262 test suite. The λ_{JS} semantics could be useful for tool developers, as it allows the tools to analyze JavaScript programs by working on the simple core language of λ_{JS}, having all the complexities of JavaScript handled in the desugared code. It was already successfully used to verify AdSafe.

We believe is worthwhile to formally relate the two semantics. One reason is, because the two semantics use a different approach, relating them will increase the confidence we can place in both of them, and is likely to find inconsistencies or other problems. Another reason is that having λ_{JS} formalized and certified (by proving correctness with respects to JSCert) enables development of certified analysis tools using λ_{JS} as the core.

This is easier said than done. The task is very complex, mostly due to the complexities of the JavaScript language. The ES5 specification, upon which JSCert and λ_{JS} are based, is 258 pages long and has 16 chapters. The JSCert pretty-big-step semantics, which models the specification, has over 900 rules; the entire Coq source code for the semantics has over 10000 lines. The λ_{JS} environment file, which contains the ECMAScript algorithms expressed in core λ_{JS} code, is almost 5000 lines long.

To reduce the complexity of the task, we are currently working only on proving λ_{JS} correct with respect to JSCert, i.e., that if the λ_{JS} program successfully terminates, then its result matches the result given by JSCert. We are also restricting our attention to ECMAScript strict mode, proving the correctness of non-strict features only when it is convenient.

We were able to prove correctness of a large part of the JavaScript semantics, including function calls, lexical environment handling, and most operations involving objects. The work is still in progress to prove the remaining parts, which include integer operations, the defineOwnProperty method, and elements of the standard library.

The development is publicly available and can be downloaded from GitHub:

https://github.com/tilk/LambdaCert

2. Formalization of λ_{JS}

For the purpose of relating the two semantics, we first developed a formalization of the λ_{JS} core language. This was necessary because

[1] The authors named the revised language S5; however, we continue to name it λ_{JS} for simplicity.

[2] https://github.com/brownplt/LambdaS5

COQPL '16, January 23, 2015, St. Petersburg, Florida, United States.
Copyright is held by the owner/author(s).
ACM 978-1-nnnn-nnnn-n/yy/mm.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

no such formalization existed previously; a formalization of the original, ES3-based λ_{JS} was available[3], but the two versions of λ_{JS} differ too much for this to be useful.

We decided to formalize the λ_{JS} core using a pretty-big-step semantics, as in JSCert, to make the job of relating the two semantics easier. We have also developed an interpreter for λ_{JS}, which we proved sound and complete with respect to the pretty-big-step semantics. The proof script was developed using domain-specific tactics, which made it simple and robust.

At this stage we stumbled upon an interesting performance issue involving the Coq proof assistant. The proof of completeness required inverting the pretty-big-step semantics judgment; this turned out to be very slow, and also generated large proof terms, which increased the memory footprint. The problem is well-known [4]; our solution involved precomputing the required inversion principles using `Derive Inversion`, and using the precomputed inversion principles to create a specialized inversion tactic.

We have also implemented in Coq the other two crucial components of the λ_{JS} semantics: the desugaring function and the environment. The desugaring was implemented as Coq functions, taking JavaScript statements and expressions (as defined in JSCert) to λ_{JS} expressions. The environment, which implements the complexities of the JavaScript semantics in λ_{JS} code, was translated from λ_{JS} syntax to Coq definitions with the help of an OCaml parser and our interpreter. The resulting Coq file is over 15000 lines long and takes almost a megabyte of disk space!

3. Correctness proof for λ_{JS} desugaring

In order to establish correctness, it was first needed to create a set of predicates specifying the relationship between JavaScript and λ_{JS} language features. As λ_{JS} was designed to implement JavaScript, the relationship is fairly simple and natural. One place where the two languages differ is the object heap, which is used in λ_{JS} to implement several JavaScript language features, including the JavaScript lexical environment. To manage this, we use a bisimulation relation, which relates JavaScript objects and environment records to λ_{JS} objects.

For the proof, we also need to keep track of certain invariants, which state that, e.g., the JavaScript objects and their λ_{JS} representations are correctly formed. The invariants are split into two relations, where one involves the evaluation contexts, and the other – the heaps. Both of these involve the bisimulation relation, but the context invariant uses it only positively. As the bisimulation relation only grows during the evaluation (as new objects are created), this makes it possible to keep the proofs simple.

The correctness proof is structured using well-founded induction on the depth of the λ_{JS} pretty-big-step semantics derivation. This grants a lot of freedom in structuring the proof, which is important because of certain structural differences between JSCert and λ_{JS}, which involve, e.g., handling of ES5 references. We use three kinds of induction hypotheses: for JavaScript statements, expressions and function calls. The separate induction hypothesis for function calls was required because of the complexity of calling functions in JavaScript, and also because function calls can happen in many possible places in JavaScript, including every object to primitive conversion.

We prove many lemmas about the different language constructs of JavaScript, including the internal "specification functions" (e.g., type conversions, operations on objects and environment records, etc.). Typically, the proof for a lemma consists of a series of forward reasoning steps and case analyses, and ends with constructing a JSCert semantics derivation and proving that the invariants were preserved. The forward reasoning is largely automated using domain-specific tactics; the JSCert derivation is constructed using the `eauto` tactic with a specialized hint database (`nocore` is used to avoid using the standard hints coming from Coq and Charguéraud's TLC library[4], which is used both by JSCert and our development).

[3] https://github.com/brownplt/LambdaJS

4. Conclusion

In the course of our work, we found numerous cases where λ_{JS} did not faithfully follow the ES5 specification[5]. Some of these issues were serious enough that far-reaching changes in the λ_{JS} environment were needed to solve them. Interestingly, no serious issues in JSCert were found, which strengthens our confidence that it accurately models the ES5 specification.

While proving the correctness of while loops, we found an interesting inconsistency between the ES5 specification and the behavior of the two most popular JavaScript implementations, Google's V5 and Mozilla's SpiderMonkey. The discovery of the inconsistency, which involves exceptions and statement result values, resulted in a correction being made in the official ES6 specification[6].

The proof scripts we developed are very taxing for the Coq proof assistant – the compilation of the whole development requires several gigabytes of RAM memory, and takes several hours to compile, even though efforts were made to optimize it. We are still looking for a solution for the performance issues.

The development is still not complete, but it already includes most of the important core features of the JavaScript language. We are looking forward to receiving feedback from the community.

References

[1] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith. A trusted mechanised JavaScript specification. In S. Jagannathan and P. Sewell, editors, *Proceedings of the Forty-First Annual ACM Symposium on Principles of Programming Languages*, pages 87–100, San Diego, CA, USA, Jan. 2014. ACM Press.

[2] A. Charguéraud. Pretty-big-step semantics. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems, 22th European Symposium on Programming, ESOP 2013*, number 7792 in Lecture Notes in Computer Science, pages 41–60, Rome, Italy, Mar. 2013. Springer-Verlag.

[3] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In T. D'Hondt, editor, *ECOOP 2010 – Object-Oriented Programming, 24th European Conference*, volume 6183 of *Lecture Notes in Computer Science*, pages 126–150, Maribor, Slovenia, June 2010. Springer-Verlag.

[4] J.-F. Monin. Proof trick: Small inversions. In *Second Coq Workshop*, Royaume-Uni Edinburgh, July 2010.

[5] J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In A. Warth, editor, *Proceedings of the 8th Symposium on Dynamic Languages*, pages 1–15, Tucson, Arizona, USA, Oct. 2012. ACM Press.

[4] http://www.chargueraud.org/softs/tlc/

[5] https://github.com/tilk/LambdaCert/wiki/Issues-with-LambdaS5

[6] https://esdiscuss.org/topic/loop-unrolling-and-completion-values-in-es6