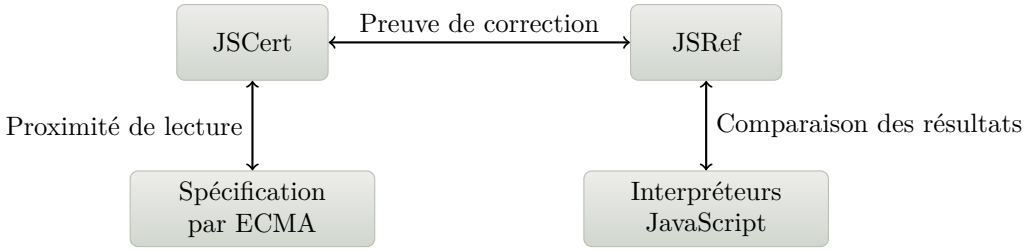
Ce document est un résumé en français de l'article “A Trusted Mechanised JavaScript Specification” [Bod+14], soit « Une spécification de confiance de JavaScript dans un assistant de preuve ». Cet article a été publié à POPL (“Principes of Programming Languages”, soit le symposium international sur les principes des languages de programmation), une conférence internationale $A*$, référence dans ce domaine. L'article est accompagné d'un site web disponible à l'adresse `http://jscert.org` donnant les liens vers la formalisation, ainsi que des informations supplémentaires sur le développement avant et après cette publication.

Contexte JavaScript est le langage de programmation le plus utilisé pour les applications clients sur Internet. C'est d'ailleurs le seul langage de programmation supporté nativement par tous les principaux navigateurs. Cela a d'ailleurs conduit à des compilateurs vers JavaScript [Zak11 ; VB14] : certains programmes présents sur Internet ont été écrits dans d'autres langages de programmation, puis compilés vers JavaScript. JavaScript était à l'origine (en 1996) essentiellement définit par ses implémentations. Cependant sa standardisation par ECMA a rapidement pris le pas, d'abord en 1997, puis dans les versions suivantes en 1999 et en 2009 qui sont maintenant suivies par tous les navigateurs. Cette spécification est cependant très complexe, et parfois ambigüe voire inconsistante.

Contributions Cet article présente JSCert, une formalisation de la cinquième version de JavaScript [ECM11] (en usage lors de la publication de l'article) écrite entièrement dans l'assistant de preuve Coq. De plus, cet article présente JSRef, un interpréteur de JavaScript écrit en Coq. Cet interpréteur a été extrait vers OCaml et est donc exécutable. Enfin, JSRef a été montré correct vis-à-vis de JSCert en Coq. Ceci permet d'assurer avec un haut niveau de confiance que JSRef suit effectivement ce que spécifie JSCert.

Le but de JSCert est de servir de base à des projets d'analyses de programmes JavaScript certifiés en Coq. Il est donc important que JSCert soit aussi correct que possible : un soin tout particulier à la confiance en la spécification a été mis en œuvre pour pouvoir faire confiance à JSCert. En particulier, devant la taille de cette formalisation — environ 900 règles de dérivations — il aurait été très probable de trouver des erreurs si nous n'avions pas mis en place ce protocole. Le protocole de confiance se résume en ce schéma :



Le premier lien est la proximité de lecture : un grand soin à été apporté à ce que la sémantique de JSCert soit aussi proche que possible de la spécification par ECMA. Pour cela, chaque règle de réduction de JSCert peut être mise en relation avec une ligne de la spécification par ECMA. Ceci permet à chacun de

vérifier si une règle donnée correspond effectivement à sa spécification. À chaque fois qu'une ambigüité a été vue dans la spécification par ECMA, nous avons vérifié le comportement des implémentations principales de JavaScript, ainsi que demandé à la liste de diffusion `es-discuss` du groupe de spécification ECMA. Ceci nous a permis d'obtenir un grand degré de confiance dans la spécification JSCert.

JSRef ayant été montré correct vis-à-vis de JSCert, cet interpréteur a le même comportement que la spécification JSCert. Nous avons exécuté cet interpréteur sur les suites de tests de JavaScript. Ceci nous as permis de tester de manière indirecte la spécification JSCert sur ces suites de tests.

Cette double méthode de vérification a permis à JSCert d'être comparé non seulement à la spécification par ECMA, mais à ses suites de tests, lui offrant une grande confiance. De plus, ceci a été le tout premier pont entre la spécification par ECMA et les suites de tests JavaScript. Construire ce pont nous a permis de détecter des incohérences entre le comportement attendu des suites de tests, la spécification par ECMA, et les comportements de certains navigateurs. Ces incohérences [BT ; The13d ; BT12 ; The13g ; The13a ; The13b ; The13c ; The13e ; The13f] ont été reconnues et la plupart ont été corrigées par ECMA et les équipes de développement des navigateurs. Il est à noter que certaines de ces incohérences ont été résolues en changeant les suites de tests, mais que certaines ont été résolues en changeant la spécification par ECMA : ceci montre que JavaScript n'est pas définit par une source de confiance unique, et justifie notre approche consistant à se comparer à la fois à la spécification et aux suites de tests. Ce pont entre la spécification par ECMA et les suites de tests permet aussi d'évaluer les suites de tests vis-à-vis de la spécification. Nous avons ainsi pu estimer la couverture de ces tests vis-à-vis de la spécification et mettre en évidence les parties de la spécification par ECMA qui ne sont pas testées.

Dans le détail, la spécification de JavaScript est immense et il nous a fallut faire des choix : sur les 15 chapitres de la spécification, nous n'avons formalisé que le cœur — c'est-à-dire les chapitres 8 à 14. Les chapitres 1 à 7 ne servent qu'à décrire comment la spécification par ECMA doit être lue, et le chapitre 15 — le plus long — sert à décrire la bibliothèque standard. En particulier, nous n'avons pas formalisé les fonctions de manipulation de tableaux, de chaîne de caractères, de dates, ainsi que toutes les fonctions mathématiques disponibles en JavaScript. Nous considérons que la formalisation de ces fonctions est orthogonale au cœur de la spécification. Nous n'avons pas non plus spécifié le parseur de JavaScript et nous nous appuyons donc sur un parseur externe pour évaluer des commandes comme `eval`.

JSCert n'est pas la première sémantique formelle de JavaScript. C'est cependant la première formalisation qui cherche à être reliée à la fois à la spécification par ECMA et les suites de tests. C'est de plus la première sémantique de JavaScript exécutable dans un assistant de preuve. Les premières spécifications formelles de JavaScript remontent à 2005 [AGD05 ; Thi05]. Ces formalisations n'étaient cependant que des version idéalisées du langage, et ne considéraient pas tous les cas particuliers de sa sémantique. De nombreuses formalisations de sous-ensembles de JavaScript ont aussi été proposées [CHJ12 ; Chu+09 ; GL09 ; Gua+11 ; HS12 ; JC09 ; JMT09 ; PSC09 ; PLR11 ; PLR12]. Étudier un sous-ensemble de JavaScript permet d'étudier certains problèmes très spécifiques. Cependant, de tels sous-langages ne sont par construction pas généraux et ne permettent pas de considérer un contexte externe ne correspondant pas au sous-

langage considéré. Ces sous-ensembles de JavaScript n'ont de plus pas été reliés à la spécification par ECMA et leurs comportement peuvent donc différer. Il a de plus été plusieurs fois montré [MT09 ; MMT09 ; MMT10] que les cas particuliers de la sémantique de JavaScript sont d'une importance centrale pour la sécurité. Enfin, une analyse empirique [Ric+11] a montré que des constructions comme `eval`, presque systématiquement absentes de ces sous-langages, sont d'une grande importance pour les programmeurs. Il existe aussi de nombreuses sémantiques formelles de JavaScript testées [HF07 ; GSK10]. Ces sémantiques ne sont malheureusement que faiblement reliées à la spécification par ECMA, mais elles restent néanmoins une des sources principales d'inspiration de cet article. Enfin, la troisième version de la spécification par ECMA a été entièrement traduite — à l'exception de quelques parties très spécifiques comme les expressions régulières ou la gestion des dates — dans une spécification formelle [MMT08]. Cette spécification n'a cependant pas été testée. Tous ces travaux connexes ont eu de grandes retombées dans la recherche automatique de bogues ou de problèmes de sécurités dans des programmes JavaScript.

Cet article pose donc les bases d'une spécification en Coq de JavaScript *tel que spécifié* par ECMA et *tel qu'exécuté* par les principaux navigateurs. C'est ainsi la première sémantique formelle à la fois proche de la spécification par ECMA et des suites de tests de JavaScript. L'article apporte un soin tout particulier à ce que la spécification soit digne de confiance et puisse donc servir de base pour une analyse certifiée de JavaScript.

Références

[AGD05] Christopher Anderson, Paola Giannini et Sophia Drossopoulou. “Towards Type Inference for JavaScript”. In : *ECOOP*. 2005.

[Bod+14] Martin Bodin et al. “A Trusted Mechanised JavaScript Specification”. In : *POPL*. 2014.

[BT] André Bargull et The JSCert Team. *Mozilla Bug 819125*. url : `https://bugzilla.mozilla.org/show_bug.cgi?id=819125`.

[BT12] André Bargull et The JSCert Team. *V8 Issue 2446*. 2012. url : `http://code.google.com/p/v8/issues/detail?id=2446`.

[CHJ12] Ravi Chugh, David Herman et Ranjit Jhala. “Dependent Types for JavaScript”. In : *OOPSLA*. 2012.

[Chu+09] Ravi Chugh et al. “Staged Information Flow for JavaScript”. In : *PLDI*. 2009.

[ECM11] ECMA International, éd. *ECMAScript Language Specification. Standard ECMA-262, Edition 5.1*. 2011. url : `http://www.ecma-international.org/publications/standards/Ecma-262-arch.htm`.

[GL09] Salvatore Guarnieri et Benjamin Livshits. “GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code”. In : *USENIX Security Symposium*. 2009.

[GSK10] Arjun Guha, Claudiu Saftoiu et Shriram Krishnamurthi. “The Essence of JavaScript”. In : *ECOOP*. 2010.

[Gua+11] Salvatore Guarnieri et al. “Saving the World Wide Web from Vulnerable JavaScript”. In : *ISSTA*. 2011.

[HF07] David Herman et Cormac Flanagan. “Status Report : Specifying JavaScript with ML”. In : *ML*. 2007.

[HS12] Daniel Hedin et Andrei Sabelfeld. “Information-Flow Security for a Core of JavaScript”. In : *CSF*. IEEE, 2012.

[JC09] Dongseok Jang et Kwang-Moo Choe. “Points-to Analysis for JavaScript”. In : *SAC* (2009).

[JMT09] Simon Holm Jensen, Anders Møller et Peter Thiemann. “Type Analysis for JavaScript”. In : *SAS*. 2009.

[MMT08] Sergio Maffeis, John C. Mitchell et Ankur Taly. “An Operational Semantics for JavaScript”. In : *APLAS*. 2008.

[MMT09] Sergio Maffeis, John C. Mitchell et Ankur Taly. “Isolating JavaScript with Filters, Rewriting, and Wrappers”. In : *ESORICS*. 2009.

[MMT10] Sergio Maffeis, John C. Mitchell et Ankur Taly. “Object Capabilities and Isolation of Untrusted Web Applications”. In : *SP*. IEEE, 2010.

[MT09] Sergio Maffeis et Ankur Taly. “Language-Based Isolation of Untrusted JavaScript”. In : *CSF*. IEEE. 2009.

[PLR11] Changhee Park, Hongki Lee et Sukyoung Ryu. “An Empirical Study on the Rewritability of `with` Statement in JavaScript”. In : *FOOL*. 2011.

[PLR12] Changhee Park, Hongki Lee et Sukyoung Ryu. “SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript”. In : *FOOL*. 2012.

[PSC09] Phu H. Phung, David Sands et Andrey Chudnov. “Lightweight Self Protecting JavaScript”. In : *ASIACCS*. 2009.

[Ric+11] Gregor Richards et al. “The `eval` that Men Do. A Large-Scale Study of the Use of `eval` in JavaScript Applications”. In : *ECOOP*. 2011.

[The13a] The JSCert Team. *ES6 Bug 1442*. 2013. url : `https://bugs.ecmascript.org/show_bug.cgi?id=1442`.

[The13b] The JSCert Team. *ES6 Bug 1443*. 2013. url : `https://bugs.ecmascript.org/show_bug.cgi?id=1443`.

[The13c] The JSCert Team. *ES6 Bug 1444*. 2013. url : `https://bugs.ecmascript.org/show_bug.cgi?id=1444`.

[The13d] The JSCert Team. *Mozilla Bug 862771*. 2013. url : `https://bugzilla.mozilla.org/show_bug.cgi?id=862771`.

[The13e] The JSCert Team. *Test262 Bug 1445*. 2013. url : `https://bugs.ecmascript.org/show_bug.cgi?id=1445`.

[The13f] The JSCert Team. *Test262 Bug 1450*. 2013. url : `https://bugs.ecmascript.org/show_bug.cgi?id=1450`.

[The13g] The JSCert Team. *V8 Issue 2529*. 2013. URL : `http://code.google.com/p/v8/issues/detail?id=2529`.

[Thi05] Peter Thiemann. “Towards a Type System for Analyzing JavaScript Programs”. In : *ESOP*. 2005.

[VB14] Jérôme Vouillon et Vincent Balat. “From Bytecode to JavaScript : the Js_of_ocaml Compiler”. In : *Software : Practice and Experience* (2014).

[Zak11] Alon Zakai. “Emscripten : An LLVM-To-JavaScript Compiler”. In : *OOPSLA*. 2011.