Javanni: A Verifier for JavaScript

Martin Nordio[1], Cristiano Calcagno[2], and Carlo A. Furia[1]

[1] Chair of Software Engineering, ETH Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch
[2] ETH Zurich, Imperial College London and Monoidics Ltd ccris@doc.ic.ac.uk

Abstract. JavaScript ranks among the most popular programming languages for the web, yet its highly dynamic type system and occasionally unintuitive semantics make programming particularly error-prone. This paper presents Javanni, a verifier for JavaScript programs that can statically detect many common programming errors. Javanni checks the absence of standard type-related errors (such as accessing undefined fields) without requiring user-written annotations, and it can also verify full functional-correctness specifications. Several experiments with JavaScript applications reported in the paper demonstrate that Javanni is flexibly usable on programs with non-trivial specifications. Javanni is available online within the CloudStudio web integrated environment.

1 Introduction

Originally developed by Netscape as a scripting language for lightweight web programming, JavaScript has rapidly become one of the most widely used programming languages[3] for the web. Its popularity has greatly exceeded its primary target—client-side web programming for nonprofessionals—and the language is now routinely used to develop large applications, such as Google Docs and Google Maps, and even some critical software such as on-line banking. Unfortunately, the original language design includes a number of quirks[4] which, combined with a highly dynamic and weakly-typed type system that mixes heterogeneous programming paradigms, make JavaScript programming particularly error-prone. Simple errors such as accessing undefined fields, invoking undefined functions, or calling functions with the wrong number of actual parameters are workaday in JavaScript programming. Even if programming frameworks exist (such as the Google Web Toolkit) that automatically translate from a higher-level language, a large part of JavaScript applications are still written by hand, with consequent risks in terms of reliability and security.

This paper describes Javanni, a static verifier for JavaScript programs that can detect many of these frustrating errors. Javanni translates JavaScript programs into Boogie [4], and then uses the Boogie verifier to check correctness properties on the translated program. Javanni is completely automatic, does not require user-written annotations, and can detect common type-related errors including: (1) invocation of undefined functions; (2) writing of undeclared variables; (3) reading of undefined values (e.g. undeclared or

[3] http://www.tiobe.com
[4] https://www.destroyallsoftware.com/talks/wat

uninitialized variables or fields); (4) incorrect number of actual parameters in function calls.

The focus of most related approaches to JavaScript verification [1,3] is restricted to standard type analysis. In contrast, Javanni supports full functional correctness properties; to our knowledge, this kind of support is available only in another quite recent work based on separation logic [2].

In the translation to Boogie, Javanni automatically introduces a specification of correct typing behavior in the form of pre- and postconditions of functions; on top of these, users may add custom assertions to the input programs and verify arbitrary functional properties of their JavaScript applications. The specification generated automatically by Javanni frames good programming practices, such as some discipline in field declaration and initialization, and hence Javanni may report false positive; in most cases, however, the reported errors are at least indicative of poor programming practices. The translation also applies *method inlining* and *loop unrolling* to improve verification accuracy without requiring extra annotations.

Javanni is a component of the CloudStudio [5] web-based multi-language integrated development environment, available online at `http://cloudstudio.ethz.ch/`. Section 3 presents a set of JavaScript programs that have been verified using Javanni, against both the automatically generated specification and more complex functional properties. Javanni is implemented in Java using the Rhino JavaScript framework. For a demo video, see `http://www.youtube.com/watch?v=K8yboTQZ9p0`.

2 Verifying JavaScript

For each JavaScript input program J, Javanni creates a Boogie file B_J containing an encoding of the source J annotated with assertions that formalize correctness properties. The translation also introduces some specification functions and axioms, used to enforce the semantics of the original JavaScript source in the Boogie language. This section succinctly illustrates the main features of the translation.

Type boxing and unboxing. To accommodate JavaScript's dynamically-typed variables within Boogie's static type system, Javanni includes *boxing* and *unboxing* functions in the translation. JavaScript variables (and fields) get a generic reference type **ref** in Boogie. Whenever we need to use some variable x according to its actual dynamic type (e.g., **int** or **bool**), we unbox it: *unbox*(x) in Boogie returns the value attached to x. Boxing works conversely; e.g., *box*(42) returns a reference attached to the integer 42. Additional axioms declare the behavior of arithmetic and Boolean operations with respect to boxing and unboxing primitive (e.g., *unbox*(*box*(i)+*box*(j))=i+j).

Object creation. JavaScript supports field initialization with the **prototype** keyword: C.**prototype**.a = v sets field a of class C to value v whenever an instance of C is created. Javanni introduces *initializer* predicates to encode the semantics of field initialization in Boogie. Javanni defines a predicate $init.C(this: \textbf{ref}, h: Heap)$ for each class C that holds for references *this* attached to objects whose fields satisfy the initializations (in a given model h of the heap). For example, a class *Student* initialized with *Student*.**prototype**.*age* = 18 determines a predicate initializer with body $init.Student(this, h) \{ h[this, age] = 18 \}$. Initializers also keep track of which

member functions are defined. For each function f member of C, $init.C(this, h)$ specifies that $h[this, func\$f] \neq undef$, where $func\$f$ is a fictitious field of C added to represent f in Boogie. Creations of an object o of class C become two assumptions in Boogie: **assume** $allocated[o, h]$ (reference o is not undefined or **null**) and **assume** $init.C(o, h)$ (the initializations hold); Boogie's **assume** statements are used as postulated facts in the reasoning.

Type correctness assertions. Javanni automatically generates Boogie assertion statements that encode the type correctness of invocations and readings of functions, variables, and fields. If Boogie can discharge all the **assert** statements, there are no type errors of these kinds. For example, every field access of the form $x.a$ determines two assertions in Boogie: the target is defined (**assert** $allocated[x, h]$); and the field is defined (**assert** $Heap[x,a] \neq undef$). More complex assertions encode type conformance and correctness of function invocations.

Contracts. To verify full functional correctness properties in JavaScript programs, Javanni supports preconditions, postconditions, and assume and assert instructions a la Boogie. While JavaScript does not natively support assertions, Javanni recognizes method calls with the special names *requires* (preconditions), *ensures* (postconditions), and *assume* and *assert*. These methods directly translate to the corresponding **requires**, **ensures**, **assume** and **assert** instructions in Boogie.

Inlining and loop unrolling. Boogie is a *modular* verifier: it reasons about function invocations using only the pre- and postconditions of the callees. This means that if function *foo* calls *bar* but the latter has no postcondition, Boogie will be oblivious of the effects of *bar* when reasoning about *foo*. To reduce the amount of user-written annotations required to reason modularly in Boogie, Javanni features *inlining*: replace a call to *bar* within *foo* with a copy of *bar*'s code, so that its effects within *foo* can be evaluated directly. A similar feature is *loop unrolling*, useful to reason inductively about loops without loop invariants. Unrolling replaces a loop by a sequence (of finite length) of conditional executions of its body; the depth of the unrolling can be set by users to find the best trade-off between scalability and annotation burden.

3 Case Study

Table 1 lists a set of examples verified using Javanni. For each example, it shows the length (in LOC) of the JavaScript source, the length of the specification added manually, the length of the Boogie source generated by Javanni without and with inlining and loop unrolling, and the time taken to check the Boogie program (on a Windows 7 machine with a 3.1 GHz dual core Intel Pentium processor and 4GB of RAM). The examples are available at `http://se.inf.ethz.ch/people/nordio/javanni/`.

Programs 1–2 only include standard type correctness properties generated automatically by Javanni. Program 1 is a collection of small JavaScript applets from `http://www.jsworkshop.com`; program 2 is one single larger application, a poker game, from the same source. Programs 3–6 are JavaScript implementations of object-oriented standard examples also used in previous work of ours [7,6], each equipped with functional specifications (pre- and postcondition) for each method. In this case, verification also required intermediate assertions, but these were much fewer than in [7,6] thanks to

NAME	LOC JS	LOC SPEC	LOC BOOGIE	LOC INLINED	TIME [S]	FEATURE
1. JS workshop	237	0	2448	3400	3.3	Type Correctness
2. Poker game	320	0	1139	12240	11.0	Type Correctness
3. Cell / Recell	130	21	580	1372	0.6	Functional
4. Counter	42	6	325	431	0.5	Functional
5. Expression	79	2	381	595	0.5	Functional
6. Sequence	102	3	440	1216	0.6	Functional
Total	**910**	**32**	**5313**	**19254**	**16.5**	

Table 1. JavaScript programs automatically verified with Javanni.

Javanni's inlining and unrolling. Programs 3–4 use object-oriented features to model: cells that store integer values (3); and a counter (4). Program 5 features nonnegative integer expression objects which can be evaluated. Program 6 models integer sequences including monotone, strict, arithmetic, and Fibonacci sequences.

4 Conclusions

This paper presented the essential features of Javanni, a verifier for JavaScript programs. Javanni works by automatically transforming JavaScript programs into the Boogie verification language; it then uses the Boogie verifier to determine if the original JavaScript program is correct. Verifying standard type correctness properties does not require special annotations; functional properties, on the other hand, are also supported and specified by means of standard pre- and postconditions, which define the expected behavior of methods. To improve verification accuracy without requiring extra annotations, Javanni's translation applies *method inlining* and *loop unrolling*. The case study suggests that these techniques can be instrumental in reducing the annotation burden required to automatically verify programs in practice.

Acknowledgments. This work was partially supported by the Swiss SNF (proj. ASII 200021-134976); and by the Gebert-Ruf Stiftung through CloudStudio's funding.

References

1. C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *ECOOP*, pages 429–452. Springer, 2005.
2. P. Gardner, S. Maffeis, and G. D. Smith. Towards a program logic for javascript. In *POPL*, pages 31–44, 2012.
3. S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *SAS*, volume 5673 of *LNCS*. Springer-Verlag, 2009.
4. K. R. M. Leino. This is Boogie 2. Technical report, Microsoft Research, 2008.
5. M. Nordio et al. Collaborative software development on the web, 2011. arXiv:1105.0768v3.
6. J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. In *SEFM*, LNCS. Springer, 2011.
7. J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Verifying Eiffel programs with Boogie. In *BOOGIE workshop*, 2011. `http://arxiv.org/abs/1106.4700`.