Goal-Directed Backwards Static Analysis for JavaScript

Extended Abstract

Benno Stein

University of Colorado Boulder

benno.stein@colorado.edu

Abstract

JavaScript is notoriously difficult to analyze due to its rampant use of standard dynamic features (e.g. duck typing, dynamic dispatch, first-class functions, and run-time string evaluation), as well as its idiosyncratic approach to scoping (scope object chains) and inheritance (prototyping). Therefore, despite its near-universal adoption as a client-side scripting language and its increasing use in server-side and mobile applications, JavaScript is rarely analyzed in practice and can be quite buggy, unreliable, and unsafe.

We present a novel technique to improve precision and efficiency of JavaScript analysis by combining a standard forwards abstract interpretation with a goal-directed backwards symbolic execution. The backwards analysis can operate either as a standalone tool to refute false alarms that arise from over-approximation in the forwards analysis, or on-line, refuting spurious data-flow on demand at critical points during the forwards analysis.

General Terms Languages, Algorithms, Verification

Keywords JavaScript, abstract interpretation, program analysis

1. Background and Related Work

Two largely distinct research areas make up the related work and relevant background for our research: JavaScript static analysis and backwards abstract interpretation. This work seeks to leverage the latter to improve the former.

1.1 JavaScript Analysis

Static analysis for dynamic languages, and JavaScript in particular, has been the focus of much recent research. Nonetheless, there are significant limitations to the currently known approaches.

Abstract interpreters such as TAJS[6] and JSAI[7] are the most comprehensive and ambitious published analyses for JavaScript. Though they are quite similar in many regards, they differ somewhat in their goals: TAJS specializes heavily in type analysis, trading generality and modularity for efficiency and precision, while JSAI presents a more general approach, sacrificing some degree of performance for a formal specification and far greater extensibility. Both systems, though, are unable to provide adequate precision to handle the rampant dynamic property accesses of some idiomatic JavaScript library code and can time out on popular open source libraries[1].

The IBM T.J. Watson Libraries for Analysis (WALA) provide a JavaScript front-end to compute points-to information and call graphs, as well as numerous analysis utilities and data structures. However, WALA faces serious scalability issues: the pointer analysis and call-graph construction are prohibitively expensive for programs that import any relatively recent version of `jQuery`, a hugely popular JavaScript library for front-end web development and DOM manipulation[8]. An earlier version of the backwards analysis tool described in this abstract was implemented over WALA's intermediate representation.

1.2 Backwards Analysis

The techniques we use for backwards analysis and query refutation are inspired by those developed by Blackshear et al. for analyzing heap reachability and Android memory leaks in the tools Thresher and Hopper[2–4]. Blackshear's tools, using a pointer analysis and callgraph generated upfront by WALA, attempt to refute false alarms by soundly exploring the state-space backwards from a given alarm site and bug condition. If all possible backwards paths lead to a contradiction, the alarm is refuted; if some path reaches an entry point without contradiction, the alarm is witnessed and confirmed to be legitimate.

2. Approach and Uniqueness

We present TAJS-Thresher: a tool that combines a forwards data-flow analysis(TAJS) with a backwards refutation analysis (à la Thresher). The tool operates in two modes: *batch*

mode, in which TAJS executes without input from Thresher, which then soundly refutes as many of the resulting alarms as possible, and *interactive* mode, in which TAJS makes queries to Thresher on-demand during the forwards analysis in order to improve precision at key points.

The challenges of designing a backwards query refutation engine for JavaScript are largely distinct from those encountered in a statically typed language like Java, which Blackshear targets with Thresher [3] and Hopper [4].

This is especially true with respect to the JavaScript heap. Handling property reads and writes precisely is extremely difficult in a backwards abstract interpretation, since the prototype semantics of a property read depend not only upon statically available class hierarchy information, as in Java, but also flow-sensitively upon the possible prototype relationships and object values at a particular program point. To address this, we design an assertion language that extends a standard field-split separation logic with novel constraints on the structure of the prototype chain, at a higher level of abstraction than the predicates proposed by Gardner et al.[5].

Furthermore, while it is a relatively straightforward task to compute a call graph for a Java program, to do so efficiently, soundly, and precisely for JavaScript remains a difficult problem because of the megamorphic call-sites that arise from imprecision with respect to dynamic dispatch. As a result, the backwards analysis of TAJS-Thresher relies on TAJS' approximated call graph, causing it to lose some degree of precision as a result of potentially spurious control-flow.

3. Results and Contributions

The primary contribution of our work will be a theoretical framework for extending a standard forwards data-flow analysis with a sound abstraction refinement mechanism. Our hope is that this system for reducing spurious data-flow by refining abstract state at pivotal locations will prove to be an effective technique for all sorts of abstract interpretations, not only for dynamic languages like JavaScript.

We will also present TAJS-Thresher, an instantiation of this framework for JavaScript type analysis, wherein TAJS will use a Thresher-style backwards query refutation mechanism to refine its abstract state at particularly problematic program points: megamorphic call sites and severely type-overloaded abstract values, for example. An implementation of TAJS-Thresher is well under way, but not yet complete.

In this abstract, I have described our system in its entirety, but my contributions are only to certain elements of it. I extended our Thresher formalism to handle JavaScript's dynamic features and implemented and tested said formalism in an earlier prototype that operated over WALA IR rather than TAJS flowgraphs. I am also responsible for building the backwards query refutation component of TAJS-Thresher, while the research group that created TAJS is handling the necessary modifications to their system. We are working collaboratively on designing a succinct and efficient API between the forwards and backwards components, and the TAJS group is taking the lead on formally specifying the algorithms needed to soundly incorporate a refinement system into the forwards data-flow analysis.

References

[1] E. Andreasen and A. Møller. Determinancy in Static Analysis for jQuery. In *OOPSLA*, 2014.

[2] S. Blackshear. *Flexible Goal-Directed Abstraction*. PhD thesis, University of Colorado Boulder, 2015.

[3] S. Blackshear, B.-Y. E. Chang, and M. Sridharan. Thresher: Precise refutations for heap reachability. In *PLDI*, 2013.

[4] S. Blackshear, B.-Y. E. Chang, and M. Sridharan. Selective Control-Flow Abstraction via Jumping. In *OOPSLA*, 2015.

[5] P. Gardner, S. Maffeis, and G. Smith. Towards a Program Logic for JavaScript. In *POPL*, 2012.

[6] S.H. Jensen, A. Møller, and P. Thiemann. Type Analysis for JavaScript. In *SAS*, 2009.

[7] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf. JSAI: A Static Analysis Platform for JavaScript. In *FSE*, 2014.

[8] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP*, 2012.