Nowadays, Javascript is used everywhere on the web, both client-side and server-side. However, Javascript’s quirky semantics introduce many problems. Debugging Javascript applications can be really cumbersome, and Javascript applications are easily subject to security issues. It is therefore necessary to be able to precisely analyze Javascript programs in order to reason about their behaviour and their potential defects.

Having a tool able to reason precisely about the behaviour of Javascript programs would prove useful in many situations. In the case of software evolution, being able to compute the data flow of a program could for example be used to analyze whether changes between two different versions of a program consist of refactoring, feature addition, or modification of the behaviour of the program.

However, Javascript’s surprising semantics complicate the job of designing static analysis tools. In order to detect complex defects or to compute the data flow in a program, one should go beyond the syntactic aspect of the language, and would need to simulate Javascript semantics, which is not an easy task. Existing tools either don’t support the full language, or tools such as TAJS\[5\] and JSAI\[3\] rely on older versions of Javascript, and are therefore not adapted to analyzing programs that make use of features introduced in ECMAScript 5.

There has been a recent effort in formalizing Javascript semantics\[1, 4\], and in reducing the complexity of those semantics by introducing simpler languages that encode those semantics. Javascript programs can be automatically translated into those simpler language via a desugaring process. This is the approach taken by $\lambda_{JS}$\[2\] and its successor $\lambda_{SS}$, which can desugar ECMAScript 5 code to a simpler scheme-like language called $S5$.

We investigate the combination of this desugaring process with abstract interpretation, a static analysis method that consists of approximately reproducing the semantics of the language in order to have a decidable approximation of every path a program can take. The formalism we use is based on Might and Van Horn’s CESK machine\[6\] and has the advantage of having tunable and arbitrarily high precision (at the cost of speed), depending on the precision needed for the analysis.

We developed a CESK machine for the $S5$ language and tried it out on desugared Javascript programs. Many challenges arose from this situation. To
keep the S5 language simple, there exists a big environment definition in S5 itself, that encodes the behaviour of Javascript standard library. This allows us to keep the CESK machine simple, but introduces an interesting trade-off: when analyzing a function inside this environment, we would like to keep a high precision; however, outside this environment, precision can be lower. Another problem coming from this environment is its size. CESK machines are typically tested on small or medium programs, whereas the S5 environment file comprises around 8000 LOC, and the desugaring process introduces a large increase between the size of the program, due to the semantic gap between Javascript and S5.

In this presentation, we will explain the advantages of using S5 as an intermediate step to analyze Javascript programs, therefore allowing analysis of the full ECMAScript 5 language. We will illustrate the problems this approach introduces, explain how we solved some of those problems and what are the remaining challenges.

References


