Flow: Abstract Interpretation of JavaScript for Type Checking and Beyond

(Invited Talk)

Avik Chaudhuri
Facebook

ABSTRACT

Flow (`https://flowtype.org`) is a powerful type checker for JavaScript that we built at Facebook, with significant contributions from the open-source community. It is heavily used for web and mobile development at Facebook.

Flow's overall goal is to maximize developer productivity without cramping the "flow" of normal JavaScript development. On the one hand, Flow uses advanced static analysis techniques to understand common JavaScript idioms precisely. This helps it find non-trivial bugs in code and provide code intelligence to editors without requiring significant rewriting or annotations from the developer. On the other hand, Flow uses aggressive parallelization and incrementalization to deliver near-instantaneous response times. This helps it avoid introducing any latency in the usual edit-refresh cycle of rapid JavaScript development.

At its core, Flow is an abstract interpreter for JavaScript. It uses *flow constraints* to statically model the dynamic semantics of JavaScript code, and computes a *closure* of the induced constraint graph to find inconsistencies. In particular, Flow can infer the type of a variable by observing the types of expressions flowing through it, while reporting errors whenever these types are incompatible with the ways the variable is used.

For example, an assignment of an expression to a variable is modeled by a simple flow from the type of the expression to the type of the variable. As a more interesting example, a function call is modeled by flowing the type of the function to a type modeling the arguments and result of the call. The flow constraint is then "executed" (as part of computing the closure) by flowing the arguments of the call to the parameters of the function, and the return of the function to the result of the call. This is analogous to how *subtyping* works, and indeed, a flow constraint can be thought of as a generalization of a subtype constraint. Carrying this idea further, path-sensitivity is also modeled using flow constraints. A dynamic test on a variable *refines* the type of the variable by flowing it to a type modeling the test. The result is a narrower type than the original, reflecting the test's success.

To be able to scale to millions of lines of code, Flow's analysis is *modular*. Each JavaScript module is parsed and analyzed in isolation, recording its dependencies. Then, the dependencies are "linked" together, propagating analysis results across modules.

Every part of the type checking pipeline is parallelized, by bucketing and distributing work among workers. As soon as a worker is done with its bucket, it gets another bucket. Workers write and read data (e.g., abstract syntax trees, constraint graphs, dependency information) in parallel via a shared heap, with a carefully designed API that enables a fast lock-free implementation.

After an initial pass over all files in the codebase, Flow monitors the file system for changes, rechecking any changed files and their dependencies in the background automatically. This architecture ensures that any analysis results—type errors, information to derive IDE tooltips like the type of a selected expression or the definition reaching a selected reference, etc.—are always up-to-date and ready whenever they're asked for.

As we continue to evolve Flow as a type checker, we have also begun exploring other areas of application, such as security analysis. For example, *untrusted* values can be modeled by extending the types of the underlying values with a "taint" type. If an untrusted value flows to a variable that doesn't expect taint, Flow can report an error. (This is analogous to how Flow already models, say, nullable types: if a nullable value flows to a variable that doesn't expect null, Flow reports an error.)

Ultimately, the ability of Flow to adapt to these new areas of application is largely due to its design and implementation as an abstract interpreter for JavaScript. Reusing Flow's existing model of data and control flows in JavaScript code, we can build new analyses simply by adding a few new types and a few new execution rules for flow constraints involving those types.

Keywords

type inference; static analysis; JavaScript

PLAS'16 October 24-24 2016, Vienna, Austria

© 2016 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-4574-3/16/10.

DOI: http://dx.doi.org/10.1145/2993600.2996280