A Light-Weight Effect System for JavaScript

Christopher Schuster   Cormac Flanagan

University of California, Santa Cruz

{cschuste,cormac}@ucsc.edu

Abstract

While types describe what values an expression computes, the *effects* of an expression describe how it is computed, e.g. whether its evaluation manipulates global state, accesses the file system or may throw certain exceptions. Having to specify types and effects throughout the program might not be feasible in a scripting language but selective, sparse effect annotations may still help to prevent many programming errors. This paper described the design and implementation of a system that statically checks effects in otherwise dynamically-typed JavaScript programs.

1. The Need for Effect Checking

In most languages, the effects of an expression are often only implicitly defined, if at all. One approach used in pure functional programming languages like Haskell is to express effects in terms of monads at the cost of additional complexity.

However, it is possible to get some of the benefits of effect checking without fully specifying all types and effects by allowing the programmer to tag functions with the different kinds of effects involved in evaluating the function and annotate function arguments with the set of expected effects.

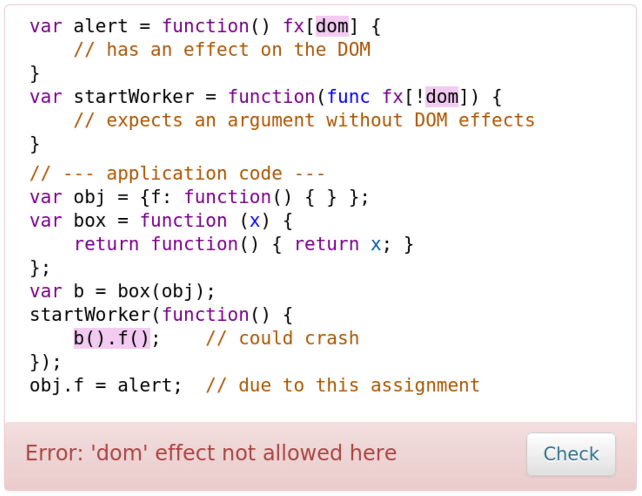
As a motivating example, the following snippet of JavaScript code calls `alert` from within a web worker[1] which causes a crash in all current browsers.

```
startWorker(function() {
  alert(); // crashes at runtime
});
```

It is easy to see that this example will crash but more complex examples could call `alert` indirectly in the worker code depending on conditions that are difficult to cover with tests. It is possible to dynamically enforce that `alert` is not called in a worker by tracking the allowed effects in the current dynamic scope. However, this would not be a significant improvement over the eventual crash without the check. By statically checking effect annotations, it is possible to prevent this kind of runtime crash for all possible executions.

[1] An HTML5 web worker runs JavaScript in the background without access to the page DOM. Here, a non-standard `startWorker` function is used.

STOP'15,   July 6, 2015, Prague.
Copyright © 2015 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn



```
var alert = function() fx[dom] {
    // has an effect on the DOM
}
var startWorker = function(func fx[!dom]) {
    // expects an argument without DOM effects
}
// --- application code ---
var obj = {f: function() { } };
var box = function (x) {
    return function() { return x; }
};
var b = box(obj);
startWorker(function() {
    b().f();    // could crash
});
obj.f = alert;  // due to this assignment
```

Error: 'dom' effect not allowed here

Figure 1. The editor automatically highlights effect checking errors in JavaScript programs. In this example, the static analysis has to take objects, aliasing, and closures into account to correctly detect the error (and potential crash) caused by the seemingly unrelated assignment in the last line.

Statically checking effect annotations requires reasoning about closures, object properties and aliasing, therefore a complete and sound solution is undecidable. Instead, our analysis overapproximates effects but never misses a potential effect error, so the crash in Figure 1 is correctly detected but in the following example our analysis erroneously reports an error:

```
startWorker(function() {
  if (false) alert(); // would never crash
});
```

2. Static Analysis

As first step of the analysis, JavaScript code gets rewritten using JS_WALA [1] to the following language ("JS Normal Form"):[2]

x, y, z : Identifiers $p \in P$: Property names

$f \in \Lambda$: Function definitions $o \in \Omega$: Object literals

$$\begin{aligned} s ::= {} & x = \lambda_f\ \overline{y}.\ \{\ \overline{s}\ \text{return}\ z\ \} \ \mid\ x = \{\overline{p : y}\}_o \\ \mid {} & \ x = y \ \mid\ x = y(\overline{z}) \ \mid\ \text{if}\ (x)\ \{\overline{s}\}\ \text{else}\ \{\overline{s}\} \\ \mid {} & \ x = y[z] \ \mid\ x = y.p \ \mid\ x[y] = z \ \mid\ x.p = z \end{aligned}$$

[2] Some syntactic forms like literals, arrays, method and constructor calls were omitted from the formalism as they do not add new insights.

All function definitions λ_f and object literals $\{\ \ldots\ \}_o$ are assigned unique names/indices f and o. Additionally, all property names p occurring in the program are known in advance.

Based on control flow-insensitive set constraint-based analysis [2, 8], we approximate the type τ with a subset of all possible function definitions Λ and object literals Ω with two functions or objects considered to be the same if they originate from the same function definition or object literal.

$$\Lambda = \lambda_1, \lambda_2, \ldots \qquad \Omega = \{\ldots\}_1, \{\ldots\}_2, \ldots$$
$$\tau \in \mathcal{P}(\Lambda \cup \Omega) \quad \text{(Type approximation)}$$

The goal of the static analysis is to assign every variable x a type τ_x, every function λ_f a return type $\tau_{f,R}$ and arguments types $\tau_{f,j}$ and every object literal $\{\}_o$ a type $\tau_{o,p}$ for every property p. To infer a suitable assignment, the analysis generates subset (type inclusion) constraints to a constraint system C for every statement in the program according the following rules:[3]

$$\tau_x : x \to \tau \ \text{(Variable)} \qquad \tau_{f,j} : \Lambda \times \mathcal{N} \to \tau \ \text{(Argument)}$$
$$\tau_{o,p} : \Omega \times P \to \tau \ \text{(Property)} \qquad \tau_{f,R} : \Lambda \to \tau \ \text{(Return)}$$

$$\begin{aligned}
C(\ x = y\ ) &= \{\tau_x \supseteq \tau_y\} \\
C(\ x = y.p\ ) &= \{\forall o \in \tau_y.\ \tau_x \supseteq \tau_{o,p}\} \\
C(\ x = y[z]\ ) &= \{\forall o \in \tau_y.\ \forall p \in P.\ \tau_x \supseteq \tau_{o,p}\} \\
C(\ x.p = z\ ) &= \{\forall o \in \tau_x.\ \tau_{o,p} \supseteq \tau_z\} \\
C(\ x[y] = z\ ) &= \{\forall o \in \tau_x\ \forall p \in P.\ \tau_{o,p} \supseteq \tau_z\} \\
C(\ x = \{\overline{p : y}\}_o\ ) &= \{\overline{\tau_{o,p} \supseteq \tau_y},\ \tau_x \supseteq \{o\}\} \\
C(\ x = y(\overline{z_j})\ ) &= \{\forall f \in \tau_y.\ \tau_x \supseteq \tau_{f,R},\ \overline{\tau_{f,j} \supseteq \tau_{z_j}}\} \\
C(\ x = \lambda_i\ \overline{y_j}.\ \{\ \overline{s}\ \text{return}\ z\ \}\ ) &= \\
&\{\tau_x \supseteq \{f\},\ \overline{\tau_{y_j} \supseteq \tau_{f,j}},\ \tau_{f,R} \supseteq \tau_z\} \cup \bigcup \overline{C(s)} \\
C(\ \text{if}\ (x)\ \{\overline{s_t}\}\ \text{else}\ \{\overline{s_e}\}\ ) &= \bigcup \overline{C(s_t)} \cup \bigcup \overline{C(s_e)}
\end{aligned}$$

This constraint system includes a complete call graph for all functions and forms the basis for the following analysis of the effect contracts.

3. Effect Checking

In order to specify effects, we extend function definitions in the language to allow a set of effect names $[\overline{e}]$ as well as expected effects $[!\overline{e}]$ to function arguments.

$$s ::= \ldots \mid x = \lambda_f[\overline{e}]\ \overline{y[!\overline{e}]}.\ \{\ \overline{s}\ \text{return}\ z\ \} \quad (e \in E : \text{Effects})$$

The effects are simple tags/strings which cannot be parametrized. In addition to types, the constraint system now also infers a mapping from functions λ_f to their effects ϕ_f based on two constraint generation rules. The first rule adds effect and effect restrictions; the second rule propagates effects from callee to the calling function which is the current function scope c during constraint generation C_f.

$$\phi_f : \Lambda \to \mathcal{P}(E) \qquad \text{(Function effects)}$$
$$\begin{aligned}
C_{\_}(\ x = \lambda_f[\overline{e_k}]\ \overline{y_j[!\overline{e_{j,l}}]}.\ \{\ \overline{s}\ \text{return}\ z\ \}\ ) &= \\
\{\overline{\phi_f \supseteq \{e_k\}},\ \overline{\forall f' \in \tau_{f,j}.\ \overline{\phi_{f'} \not\supseteq \{e_{j,l}\}}}\} &\cup \bigcup \overline{C_f(s)} \\
C_f(\ x = y(\overline{z})\ ) &= \{\forall f' \in \tau_y.\ \phi_f \supseteq \phi_{f'}\}
\end{aligned}$$

[3] These constraints are generated with complexity $O(n^3)$ [8].

The resulting constraint system can then be solved with an SMT solver like Z3 [5]. If the system is unsatisfiable, the unsatisfiable core of constraints is used to highlight code in the original program involved in the effect errors (see Figure 1).

4. Related Work

Prior work on effect systems encompasses decades of research and often goes beyond the restricted set of effect tags presented in this paper by better integrating the effect system with the operational semantics of the language [3, 9]. Effect systems with algebraic effects [10] allow an even richer effect language, e.g. Brady [4] showed an effect system for Idris in which effects can be parametrized by other types, so a `STATE Int` effect would allow type-safe state updates on integer values.

Additionally, set constraint-based program analysis [2, 8] for JavaScript has been explored before, e.g. the idea of typing objects and functions by their generating definition or literal has previously been applied by Hackett and Guo [7] to improve the performance of the SpiderMonkey JIT compiler.

5. Discussion

Effects are an integral part of the program behavior, so leaving these unspecified might increase the potential for accidental programming errors. Unfortunately, specifying all the types and effects in a program might hinder scripting and prototyping, so a light-weight effect system which automatically infers unspecified effects might serve as compromise solution.

Our prototype implementation[4] shows that effects can be checked statically in a language like JavaScript. It requires effect annotations which are automatically desugared by a set of sweet.js macros [6]. However, other ways of specifying effect contracts might be of interest for future work.

Apart from restricting certain kinds of programmer-specified effects like DOM or file system access, effect contracts could also be used to enforce checked exceptions as used in Java. Furthermore, effect annotations as presented in this paper can also be seen as statically-checked higher-order temporal contracts which is part of a broad area of research in contract systems.

References

[1] JS_WALA - WALA Analyses and tools that are implemented in JavaScript. https://github.com/wala/JS_WALA.

[2] A. Aiken and E. L. Wimmers. Type Inclusion Constraints and Type Inference. FPCA '93, New York, NY, USA, 1993.

[3] F. Bañados Schwerter, R. Garcia, and E. Tanter. A theory of gradual effect systems. ICFP '14, New York, NY, USA, 2014. ACM.

[4] E. Brady. Programming and Reasoning with Algebraic Effects and Dependent Types. ICFP '13, New York, NY, USA, 2013.

[5] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, LCNS. Springer Berlin Heidelberg, 2008.

[6] T. Disney, N. Faubion, D. Herman, and C. Flanagan. Sweeten Your JavaScript: Hygienic Macros for ES5. DLS '14, pages 35–44, 2014.

[7] B. Hackett and S.-y. Guo. Fast and Precise Hybrid Type Inference for JavaScript. PLDI '12, New York, NY, USA, 2012.

[8] N. Heintze. Set-based Analysis of ML Programs. *SIGPLAN Lisp Pointers*, VII(3), July 1994.

[9] D. Marino and T. Millstein. A generic type-and-effect system. TLDI '09, New York, NY, USA, 2009. ACM.

[10] G. Plotkin and M. Pretnar. Handlers of Algebraic Effects. In *Prog. Languages and Systems*, LNCS. Springer Berlin Heidelberg, 2009.

[4] Source and demo are available at https://github.com/levjj/jsfxs