Dynamic Flow Analysis for JavaScript

Nico Naus[1](✉) and Peter Thiemann[2]

[1] Utrecht University, Utrecht, The Netherlands
n.naus@uu.nl
[2] Albert-Ludwigs-Universität Freiburg, Freiburg im Breisgau, Germany
thiemann@acm.org

Abstract. Static flow analyses compute a safe approximation of a program's dataflow without executing it. Dynamic flow analyses compute a similar safe approximation by running the program on test data such that it achieves sufficient coverage.

We design and implement a dynamic flow analysis for JavaScript. Our formalization and implementation observe a program's execution in a training run and generate flow constraints from the observations. We show that a solution of the constraints yields a safe approximation to the program's dataflow if each path in every function is executed at least once in the training run. As a by-product, we can reconstruct types for JavaScript functions from the results of the flow analysis.

Our implementation shows that dynamic flow analysis is feasible for JavaScript. While our formalization concentrates on a core language, the implementation covers full JavaScript. We evaluated the implementation using the SunSpider benchmark.

Keywords: Type inference · JavaScript · Flow analysis · Dynamic languages

1 Introduction

Flow analysis is an important tool that supports program understanding and maintenance. It tells us which values may appear during evaluation at a certain point in a program. Most flow analyses are static analyses, which means they are computed without executing the program. This approach has the advantage that information can be extracted directly from the program text. But it has the disadvantage that significant effort is required to hone the precision of the analysis and then to implement it, for example, in the form of an abstract interpreter.

Constructing the abstract interpreter is particularly troublesome if the language's semantics is complicated or when there are many nontrivial primitive operations. First, the implementer has to come up with suitable abstract domains to represent the analysis results. Then, a sound abstraction has to be constructed for each possible transition and primitive operation of the language. Finally, all these domains and abstract transition functions must be implemented. To obtain good precision, an abstract domain often includes a singleton abstraction, in which case the abstract interpreter necessarily contains a concrete interpreter for the language augmented with transitions for the more abstract points in the domain. Clearly, constructing such an abstraction presents a significant effort.

We follow the ideas of An and others [1] who propose dynamic type inference for Ruby, a class-based scripting language where classes have dedicated fields and methods. The benefit of their approach is that existing instrumentation tools can be used, which minimizes the implementation effort, and that high precision (i.e., context-sensitive flow information) is obtained for free.

This paper adapts their approach to dynamic type inference to JavaScript. As JavaScript is not class-based, the adaptation turns out to be nontrivial, although the principal approach—generating typing constraints during execution—is the same. Regarding the differences, in the Ruby work, class names are used as types. In (pre-ES6) JavaScript, there are no named classes, so we have to identify a different notion of type. Our solution is drawn from the literature on flow analysis: we use creation points [13], the program points of `new` expressions, as a substitute for class and function types. We argue that this notion is fairly close to using a class name: The typical JavaScript pattern to define a group of similarly behaving objects is to designate a constructor function, which may be identified by the program point of its definition, and then use this constructor in the `new` expression to create objects of that "class". Hence, the prototype of the constructor could substitute for a class, but it is hard to track. The program point of the `new` is much easier to track and it also approximates the class at a finer degree as the constructor. For simplicity, we use the latter. Choosing program points to approximate run-time entities means that we switch our point of view from type system to flow analysis.

Another difference between JavaScript and Ruby is the definition of what constitutes a type error. The Ruby work considers message-not-understood errors, the typical type error in a class-based object-oriented language. In JavaScript, no such concept exists. In fact, there are only two places in the standard semantics that trigger a run-time type error:

– trying to access a property of `undefined` or `null` and
– trying to invoke a non-function as a function.

We concentrate on the second error and set up our formal system to only avoid failing function calls. The first error may be tracked with similar means and is omitted in favor of a simpler system.

After looking at an example of our approach in Sect. 2, to build intuition, we construct a formal system for a JavaScript core language in Sect. 3. This core language simplifies some aspects of JavaScript to make proofs easier. We describe the analysis in detail, which consists of a training semantics and a monitoring semantics, and prove its soundness. Section 4 presents a practical implementation, which is evaluated in Sect. 5. Section 6 compares our work with previous work, and finally Sect. 7 concludes this paper.

```
1  function test(x){,
2    return{
3      if(x.val)                           e  <= [val : g]
4        then x.val = inc(x.val)
5        else x.val = 1                    e  <= [val : g], Num <= g
6    }
7  }
8  function inc(x){,
9    return x+1
10 }
11 function main(x){var foo bar result,
12   return {
13     foo = new null;
14     foo.val = 0;                        13 <= [val : c], Num <= c
15     bar = new foo;                      15 <= 13
16     bar.foo = new foo;                  15 <= [foo : d], 16 <= d, 16 <= 13
17     result = test(foo);                 1  <= e -> f, 13 <= e, Num <= f
18     inc(result);}}                      8  <= h -> i, f <= h, Num <= i
-------------------------------------------------------------------------------
e = main();                                11 <= a -> b, () <= a, i <= b
```

Fig. 1. Example program written in core JavaScript, with generated constraints

2 Example

Figure 1 shows an example program, written in the core JavaScript language that will be defined in the next section. On the right are the constraints generated by the flow analysis. Objects and functions are identified in the constraints by the line number they were created on. The symbols *a, b, c...* appearing in the constraints are type variables. Functions are defined as **`function`** `f(x){var y`***`,return`** `e}`, where `f` is the name of the function, `x` the name of the argument, `y`* a list of local variables, and `e` the function body.

Function calls, like the one on line 18, result in three constraints. First, we constrain the object we call to be a function (`8 <= h -> i`), second, we constrain the argument (`f <= h`) and third, we constrain the type of the return value (`Num <= i`). When a new object is created, as in line 15, we generate the constraint `15 <= 13`, since we want the type of the new object to be a subtype of the type of the old object.

Line 14 assigns a value to a property of an object. The assignment results in two constraints, `13 <= [val : c]` to constrain the type of the object to have the property `val` and `Num <= c` to constrain the type of the object's property to a supertype of `Num`.

The function `inc` does not generate any constraints, because it only accesses local variables. The function `test` only generates constraints for the else-branch, because we do not visit the then-branch.

After execution, the type of every object can be inferred using these constraints.

3 Formal System

The formal system employs a JavaScript core language with the syntax defined in Fig. 2. It features the usual JavaScript constructs like constants, variables,

Expressions	$e ::= c \mid x \mid \mathit{fundec}^\ell \mid e(e) \mid e; e \mid \mathit{prim}(e) \mid x = e$
	$\mid \texttt{if}^\ell\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{new}^\ell\ e \mid e.n \mid\ e.n = e$
Constants	$c ::= \mathit{num} \mid \mathit{str} \mid \mathit{bool} \mid \texttt{null} \mid \texttt{udf}$
Function	$\mathit{fundec}^\ell ::= \texttt{fun}^\ell\ f(x)\{\texttt{var}\ y^*, \texttt{return}\ e\}$
Variables	$x, f, n \in$ set of names
Primitives	$\mathit{prim} \in$ set of primative operation
Labels	$\ell \in$ set of Labels
Program	$\mathit{prog} ::= \mathit{fundec}^* \rhd e$

Fig. 2. Syntax of the JavaScript core language

heaps $H ::= (l \mapsto \mathit{obj})^*$	path $\phi ::= p^*$
activation record $S ::= (x \mapsto v)^*$	literal $p ::= \ell \mid \neg\ell$
values $v ::= l \mid c$	constraints $C ::= (\tau \leq \tau')^*$
object $\mathit{obj} ::= (v, (n \mapsto v)^*)$	$\mathit{Falsey} ::= \texttt{udf} \mid \texttt{null} \mid 0 \mid$ "" $\mid \texttt{false}$
wrapped values $\omega ::= v : \bar{\tau}$	$l \in$ Heap addresses
abstract types $\bar{\tau} ::= \tau \mid \alpha$	$\alpha \in$ Type variables
paths $\Phi ::= \phi^*$	$n \in$ Property names

Fig. 3. Semantic objects

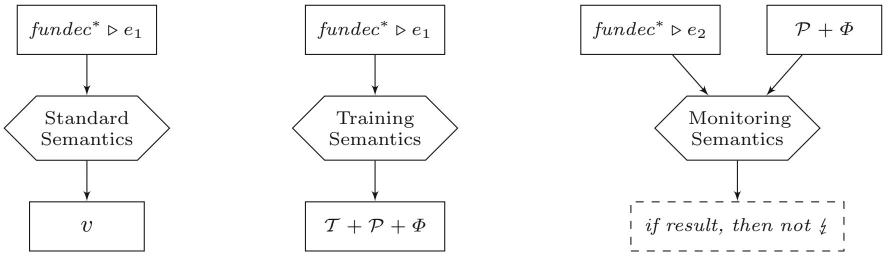
functions, function application, primitive operations, conditional, assignment to local variables, new object creation (where the argument is the prototype), property get and property set. Function definition, `new`, and conditional are marked with program labels ℓ to address them in the inference phase. The most notable difference to full JavaScript is the omission of all reflective features: there is no `eval` and no bracket notation to access properties. Thus, there is an a-priori fixed set of properties and all property manipulation happens via the dot notation. This restriction simplifies our semantics considerably compared to existing semantics for JavaScript.

Figure 3 declares the semantic objects for the core language. We keep state in heaps H and activation records S. The heap contains a mapping from locations to objects. An activation record, or stack entry, is a mapping from variables to values. A value is either a heap address l or a constant c. Objects *obj* contain their prototype and a mapping from property names to values. The property names "$fun", "$vars" and "$tyvar" are reserved and cannot be used by the programmer. Their use will become clear in the next section.

There are type variables α and concrete types τ. Concrete types, as defined in Fig. 4, are composed of one or more type summands. To record execution paths, we define path sets Φ and single paths ϕ, which are lists of potentially negated program labels of conditionals. A positive label ℓ indicates a then-branch taken, a negated label $\neg\ell$ indicates an else-branch. Lastly, a constraint set C collects constraints of the abstract form $\bar{\tau} \leq \bar{\tau}'$. Such a constraint indicates that $\bar{\tau}$ is a subtype of $\bar{\tau}'$.

$$
\begin{aligned}
\text{types } \tau &::= \sum_{i \in T, T \subseteq \{\bot, u, b, s, n, f, o\}} \varphi_i \\
\text{row } \varrho &::= str : \bar{\tau}, \varrho \mid \alpha
\end{aligned}
\qquad
\begin{aligned}
\text{undefined } \varphi_\bot &::= \texttt{Udf} \\
\text{null } \varphi_u &::= \texttt{Null} \\
\text{boolean } \varphi_b &::= \texttt{Bool} \\
\text{string } \varphi_s &::= \texttt{String} \\
\text{number } \varphi_n &::= \texttt{Number} \\
\text{function } \varphi_f &::= \texttt{Function}(\bar{\tau} \to \bar{\tau}) \\
\text{object } \varphi_o &::= \texttt{Obj}(\varrho)
\end{aligned}
$$

Fig. 4. Types



fundec* $\triangleright$ e_1
Standard Semantics
v
fundec* $\triangleright$ e_1
Training Semantics
$\mathcal{T} + \mathcal{P} + \Phi$
fundec* $\triangleright$ e_2
$\mathcal{P} + \Phi$
Monitoring Semantics
if result, then not $\lightning$

Fig. 5. Overview of formal system

The formal system comprises three parts as shown in Fig. 5. The standard semantics evaluates a top-level expression e_1 in the context of a program *fundec**. The training semantics augments the standard semantics by collecting type constraints and execution paths. Flow information in the form of a type equivalence map $\mathcal{P}$ and types $\mathcal{T}$ is inferred from the resulting constraints.

The monitoring semantics is an artifact to prove the soundness of the inferred types. It evaluates the same program for a different top-level expression. It also takes the equivalence map and the set of paths collected by the training semantics. The monitoring semantics is constructed such that no type errors occur during execution if the equivalence information is respected and if all execution paths in every function have been exercised by the training semantics.

3.1 Training Semantics

The training semantics collects type constraints and keeps track of execution paths. The main challenge in the construction of this semantics was to identify the correct abstraction of functions and objects by *program locations*. As any function can serve as a constructor in JavaScript, it is not straightforward to recognize a constructor application and give it a proper type. We omit the standard semantics, which can be obtained by erasing the collection of constraints and paths from the training semantics.

Figure 6 defines a big-step evaluation judgment of the form $H; S; e \longrightarrow H'; S'; \omega \mid C; \phi; \Phi$. Given heap H and activation record S, the expression e evaluates to an augmented value ω with updated heap and activation record

TVarLookup

$$\frac{S(x) = \omega}{H; S; x \longrightarrow H; S; \omega \mid \{\}; \{\}; \{\}}$$

TPCall

$$\frac{H; S; e \longrightarrow H'; S'; v : \_ \mid C; \phi; \Phi \qquad [\![prim]\!]v = \omega}{H; S; prim(e) \longrightarrow H'; S'; \omega \mid C; \phi; \Phi}$$

TNew

$$\frac{\begin{array}{c} H; S; e \longrightarrow H'; S'; v : \bar{\tau} \mid C; \phi; \Phi \qquad l = \text{fresh location} \\ \alpha' = \ell \qquad \text{if } (\bar{\tau} = \alpha) \text{ then } (C' = \alpha' \leq \alpha) \text{ else } (C' = \{\}) \\ obj = (v : \bar{\tau}, \{\}) \qquad H'' = H'\{l \mapsto obj : \alpha'\} \end{array}}{H; S; \texttt{new}^{\ell} e \longrightarrow H''; S'; l : \alpha' \mid C, C'; \phi; \Phi}$$

TVarAss

$$\frac{\begin{array}{c} H; S; e \longrightarrow H'; S'; \omega \mid C; \phi; \Phi \\ S'' = S'\{x \mapsto \omega\} \end{array}}{H; S; x = e \longrightarrow H'; S''; \omega \mid C; \phi; \Phi}$$

TFun

$$\frac{l = \text{fresh location} \qquad \alpha = \ell \qquad H' = H\{l \mapsto (\texttt{null}, \$\text{fun} \mapsto fundec, \$\text{vars} \mapsto S\downarrow_{fv(fundec)}) : \alpha\}}{H; S; fundec \longrightarrow H'; S; l : \alpha \mid \{\}; \{\}; \{\}}$$

TProp

$$\frac{H; S; e \longrightarrow H'; S'; l : \alpha \mid C; \phi; \Phi \qquad C' = \alpha \leq [n : \alpha.n] \qquad H'; l.n \longrightarrow \omega}{H; S; e.n \longrightarrow H'; S'; \omega \mid C, C'; \phi; \Phi}$$

TPropSet

$$\frac{\begin{array}{c} H; S; e \longrightarrow H'; S'; l : \alpha \mid C; \phi; \Phi \qquad H'; S'; e' \longrightarrow H''; S''; v : \bar{\tau} \mid C'; \phi'; \Phi' \\ C'' = \alpha \leq [n : \alpha.n], \bar{\tau} \leq \alpha_n \qquad H''' = H''\{l \mapsto H''(l)\{n \mapsto v : \alpha.n\}\} \end{array}}{H; S; e.n = e' \longrightarrow H'''; S''; v : \alpha.n \mid C, C', C''; \phi, \phi'; \Phi, \Phi'}$$

TSeq

$$\frac{H; S; e \longrightarrow H'; S'; \_ \mid C; \phi; \Phi \qquad H'; S'; e' \longrightarrow H''; S''; \omega \mid C'; \phi'; \Phi'}{H; S; (e; e') \longrightarrow H''; S''; \omega \mid C, C'; \phi, \phi'; \Phi, \Phi'}$$

TConditional

$$\frac{\begin{array}{c} H; S; e \longrightarrow H'; S'; c : \tau \mid C; \phi; \Phi \\ \text{if } (c \notin \mathit{Falsey}) \text{ then } (p = \ell, e_p = e') \text{ else } (p = \neg\ell, e_p = e'') \qquad H'; S'; e_p \longrightarrow H''; S''; \omega \mid C'; \phi'; \Phi \end{array}}{H; S; \texttt{if}^{\ell}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' \longrightarrow H''; S''; \omega \mid C, C'; \phi, p, \phi'; \Phi, \Phi'}$$

TCall

$$\frac{\begin{array}{c} H; S; e \longrightarrow H'; S'; l : \alpha \mid C; \phi; \Phi \qquad H'; S'; e' \longrightarrow H''; S''; v : \bar{\tau} \mid C'; \phi'; \Phi' \\ H''(l) = (\_, \$\text{fun} \mapsto \texttt{fun}\ f(x^f)\{(\texttt{var}\ y)^*, \texttt{return}\ e^f\}, \$\text{vars} \mapsto S^f, \ldots) : \alpha \\ S^{f'} = S^f\{f \mapsto l : \alpha, x^f \mapsto v : \alpha_{\texttt{arg}}, (y \mapsto \texttt{udf} : \texttt{Udf})^*\} \qquad C^{\texttt{call}} = \alpha \leq \alpha_{\texttt{arg}} \longrightarrow \alpha_{\texttt{ret}}, \bar{\tau} \leq \alpha_{\texttt{arg}} \\ H''; S^{f'}; e^f \longrightarrow H'''; \_; v' : \bar{\tau}' \mid C''; \phi''; \Phi'' \qquad C^{\texttt{ret}} = \bar{\tau}' \leq \alpha_{\texttt{ret}} \end{array}}{H; S; e(e') \longrightarrow H'''; S''; v' : \alpha_{\texttt{ret}} \mid C, C^{\texttt{call}}, C', C^{\texttt{ret}}; \phi, \phi'; \Phi, \Phi', \phi'', \Phi''}$$

Prototype lookup

TPropLookup

$$\frac{H(l)[n] = \omega}{H; l.n \longrightarrow \omega}$$

TProtoLookup

$$\frac{n \notin H(l) \qquad H; H(l)_{\texttt{proto}}.n \longrightarrow \omega}{H; l.n \longrightarrow \omega}$$

Top-level initialization rule

TRun

$$\frac{(H, S) = \text{initialize}(fundec^*) \qquad H; S; e \longrightarrow \_; \_; \_ \mid C; \_; \Phi}{fundec^* \triangleright e \uparrow (\mathcal{T}, \mathcal{P}) = \text{Solve(C)}; \Phi}$$

Fig. 6. Training semantics

H' and S'. The C-component contains the constraints collected during evaluation, ϕ contains the evaluation path inside the currently executed function, and Φ contains paths collected during evaluation. An evaluation path records the outcomes of the conditionals that were executed.

Most rules are standard, apart from the constraint collection, path recording, and passing of observed paths and constraints. Variables are looked up directly in the activation record. TPcall performs primitive operations and is assumed to only return non-object values. Variable assignment is performed by updating the activation record. Rule TSeq evaluates the first expression, then the second, and returns the value of the second. TConditional checks if the condition evaluates to true or false and acts accordingly. Function literals are converted to objects by the TFun rule. Function objects do not have a prototype, the actual function is stored in the "\$fun" property, and the free variables of e are stored in "\$vars". This treatment of functions is analogous to the actual semantics of JavaScript. Prototypes are set when a new object is created using the TNew rule. We explicitly allow creating a new object from either an object or just a regular value. If a regular value like `null` is used, then the object has no prototype. Prototype lookup is performed by TPropLookup and TProtoLookup, when a property of an object is requested in the TProp rule. Lookup relies on a different judgment $H; l.n \longrightarrow \omega$, with heap H, heap location l, and property name n that returns a wrapped value ω.

The rule TCall also deserves some extra explanation. From the heap, we retrieve the desired function, which must be an object as mentioned above. We construct a new activation record by taking the bound variables in "\$vars" and adding references to the function (for recursive calls), to the argument, and to local variables. We execute the actual function with this new activation record.

Four of the evaluation rules collect constraints.

TNew. When a new object is created, it should have at least the same type as its prototype, but it may have additional properties. Therefore, we constrain the type of the new object to be a subtype of the type of its prototype.

TProp. A property lookup requires the property to be present in the type of the object.

TPropSet. Setting a property requires existence of the property and the type of the new value is a subtype of the property's type.

TCall. The type of the object that is called as a function is constrained to be a function. The type of the argument must be a subtype of the function argument. The return type of the function should be a subtype of the outcome of the function call.

Two rules record the paths taken by the execution

TConditional. Either a positive or negative label is added to the current path, depending on the value of the condition.

TCall. The path taken by the dispatch is added to the set of observed paths Φ.

The TCall rule furthermore wraps a new type variable around the argument value passed to the function, it assigns types to the freshly initialized local variables, and wraps a new type variable around the value returned from the function. The type variables are connected to the prior type wrappings through the above-mentioned constraints.

At top level, we have a different judgment, $\mathit{fundec}^* \triangleright e \uparrow \mathcal{T}; \mathcal{P}; \Phi$, that is evaluated by the TRun rule. This rule initializes the heap and the top-level bindings from the list of function declarations. It then evaluates the top-level expression e. Afterwards, it solves the constraints and returns a mapping from type variables to inferred types $\mathcal{T}$, an equivalence set mapping $\mathcal{P}$, and the observed paths Φ. Whenever the constraint solver determines that two type variables must be equal, it records this fact in mapping $\mathcal{P}$, which is implemented using a union-find algorithm, as in Henglein's binding-time analysis [9].

3.2 Monitoring Semantics

The rule set in Fig. 7 defines the monitoring semantics. This big-step semantics is defined in terms of the outcome $\mathcal{P}, \Phi$ of a preceding training run and it is restricted to execute only paths that have been trained according to Φ. Hence, the evaluation judgment has the form $H; S; e \mid \phi \longrightarrow H'; S'; v \mid \phi'$, where ϕ is the path that the evaluation has to follow, and ϕ' contains the remainder of the path after evaluation. To avoid clutter, we leave the parameter Φ implicit. It is only used in the MCall rule.

Some rules deviate from the standard semantics to take paths into account:

MConditional. This rule only applies if the outcome of the condition coincides with the head of the path the execution has to take.

MCall. To execute the method dispatch, the rule nondeterministically selects a path for the function body from the set of trained paths Φ.

MNew. The type variable of the newly created object is also stored in the reserved property "$tyvar".

MFun. The type variable for the function is stored in the function object.

MRun. Besides the usual initialization, this rule executes the monitor rule on the top-level expression.

There are three new rules compared to the standard semantics.

Monitor. This rule applies a meta-function *mon* to the top-level expression which replaces all property assignments and function calls with their underlined version to enforce their evaluation with MTLPropSet and MTLCall.

Error. This rule defines what we consider a type error: when a non-function object is used as a function. The rules for error propagation are standard and omitted for space reasons.

MTLPropSet. This rule verifies if the property assignments from the top-level expression e meet the precondition required in the soundness proof.

MTLCall. This rule verifies that the object in function position is indeed a function. If so, it proceeds with the standard function call rule MCall.

MVarLookup

$$\frac{S(x) = v}{H; S; x \mid \phi \longrightarrow H; S; v \mid \phi}$$

MPCall

$$\frac{H; S; e \mid \phi \longrightarrow H'; S'; v \mid \phi' \qquad [\![\mathit{prim}]\!] v = v'}{H; S; \mathit{prim}(e) \mid \phi \longrightarrow H'; S'; v' \mid \phi'}$$

MNew

$$\frac{\begin{array}{c} H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \qquad l' = \text{fresh label} \\ \alpha = \ell \qquad \mathit{obj} = (v, \$\text{tyvar} \mapsto \alpha) \qquad H' = H\{l' \mapsto \mathit{obj}\} \end{array}}{H; S; \texttt{new}^{\ell}\, e \mid \phi \longrightarrow H'; S; l \mid \phi'}$$

MVarAss

$$\frac{\begin{array}{c} H; S; e \mid \phi \longrightarrow H'; S'; v \mid \phi' \\ S'' = S'\{x \mapsto v\} \end{array}}{H; S; x = e \mid \phi \longrightarrow H'; S''; v \mid \phi'}$$

MFun

$$\frac{\begin{array}{c} l = \text{fresh location} \\ \alpha = \ell \qquad H' = H\{l \mapsto (\texttt{null}, \$\text{fun} \mapsto \mathit{fundec}, \\ \$\text{vars} \mapsto S\downarrow_{fv(\mathit{fundec})}, \$\text{tyvar} \mapsto \alpha)\} \end{array}}{H; S; \mathit{fundec}^{\ell} \mid \phi \longrightarrow H; S; o \mid \phi}$$

MProp

$$\frac{H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \qquad H'; l.n \longrightarrow v}{H; S; e.n \mid \phi \longrightarrow H'; S'; v \mid \phi'}$$

MPropSet

$$\frac{\begin{array}{c} H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \\ H'; S'; e' \mid \phi' \longrightarrow H''; S''; v \mid \phi'' \\ H''' = H''\{l \mapsto H''(l)\{n \mapsto v\}\} \end{array}}{H; S; e.n = e' \mid \phi \longrightarrow H'''; S''; v \mid \phi''}$$

MSeq

$$\frac{\begin{array}{c} H; S; e \mid \phi \longrightarrow H'; S'; \_ \mid \phi' \\ H'; S'; e' \mid \phi' \longrightarrow H''; S''; v \mid \phi'' \end{array}}{H; S; (e; e') \mid \phi \longrightarrow H''; S''; v \mid \phi''}$$

MConditional

$$\frac{\begin{array}{c} H; S; e \mid \phi \longrightarrow H'; S'; c \mid p, \phi' \\ \text{if } (c \notin \mathit{Falsey}) \text{ then } (p = \ell, e_p = e') \text{ else } (p = \neg\ell, e_p = e'') \qquad H'; S'; e_p \mid \phi' \longrightarrow H''; S''; v \mid \phi'' \end{array}}{H; S; \texttt{if}^{\ell}\ e\ \texttt{then}\ e'\ \texttt{else}\ e'' \mid \phi \longrightarrow H''; S''; v \mid \phi''}$$

MCall

$$\frac{\begin{array}{c} H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \qquad H'; S'; e' \mid \phi \longrightarrow H''; S''; v \mid \phi'' \\ H''(l) = (\_, \$\text{fun} \mapsto \texttt{fun}\ f(x^f)\{(\texttt{var}\ y)^*, \texttt{return}\ e^f\}, \$\text{vars} \mapsto S^f, \$\text{tyvar} \mapsto \alpha, \ldots) \\ S^{f'} = S^f\{f \mapsto l, x^f \mapsto v, (y \mapsto \texttt{udf})^*\} \qquad \bar{\phi} \in \Phi \qquad H'; S^{f'}; e^f \mid \bar{\phi} \longrightarrow H''; \_; v' \mid \_ \end{array}}{H; S; e(e') \mid \phi \longrightarrow H''; S'; v' \mid \phi'}$$

MTLPropSet

$$\frac{\begin{array}{c} H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \qquad H'; S'; e' \mid \phi' \longrightarrow H''; S''; v \mid \phi'' \\ \mathit{runtype}_{H''}(v) \in \mathcal{P}(\mathit{runtype}_{H''}(l).n) \qquad H''' = H''\{l \mapsto H''(l)\{n \mapsto v\}\} \end{array}}{H; S; \underline{e.n = e'} \mid \phi \longrightarrow H'''; S''; v \mid \phi''}$$

MTLCall

$$\frac{\begin{array}{c} H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \qquad H''(l) = \mathit{obj} \\ \$\text{fun} \in \mathit{obj} \qquad H; S; e(e') \mid \phi \longrightarrow H''; S'; v' \mid \phi' \end{array}}{H; S; \underline{e(e')} \mid \phi \longrightarrow H''; S'; v' \mid \phi'}$$

Error

$$\frac{\begin{array}{c} H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \\ H'; S'; e' \mid \phi \longrightarrow H''; S''; v \mid \phi'' \\ H''(l) = \mathit{obj} \qquad \$\text{fun} \notin \mathit{obj} \end{array}}{H; S; e(e') \mid \phi \longrightarrow \lightning}$$

Top-level initalization rules

MRun

$$\frac{\begin{array}{c} (H, S) = \text{initialize}(\mathit{fundec}^*) \\ H; S; \mathit{mon}(e) \mid \{\} \longrightarrow \_; \_; v \mid \{\} \end{array}}{\mathcal{T}; \mathcal{P}; \Phi \vdash \mathit{fundec}^* \triangleright e \uparrow v}$$

Monitor

$$\frac{\{\}; \{\}; \mathit{mon}(e) \mid \{\} \lightning}{\mathcal{T}; \mathcal{P}; \Phi \vdash \mathit{fundec}^* \triangleright e \lightning}$$

Rules for prototype lookup

MPropLookup

$$\frac{H(l)[n] = v}{H; l.n \longrightarrow v}$$

MProtoLookup

$$\frac{n \notin H(l) \qquad H; H(l)_{\texttt{proto}}.n \longrightarrow v}{H; l.n \longrightarrow v}$$

Fig. 7. Monitoring semantics rules

The function $runtype_H$ converts values to types. For location values l, a type variable is retrieved from the heap. A constant value results in a concrete type.

$$\begin{aligned} runtype_H = \{ & l \mapsto H(l)[\$\text{tyvar}], num \mapsto \texttt{Number}, str \mapsto \texttt{String}, \\ & bool \mapsto \texttt{Bool}, \texttt{null} \mapsto \texttt{Null}, \texttt{udf} \mapsto \texttt{Udf}\} \end{aligned}$$

3.3 Soundness

In this section, we show that the types and flows inferred by this system are sound. Formally, we prove the following soundness theorem.

Theorem 1 (Soundness). *Suppose there is a training run $fundec^* \triangleright e_1 \uparrow \mathcal{T}; \mathcal{P}; \Phi$, with $\mathcal{T}$ the types and $\mathcal{P}$ the equivalence mapping resulting from constraint solving and Φ the set of traversed paths.*

Then there cannot be an expression e_2 that evaluates to $\lightning$ with the monitoring semantics using the previously inferred types and traversed paths, notated as $\mathcal{T}; \mathcal{P}; \Phi \vdash fundec^* \triangleright e_2 \lightning$.

That is, if the training run has inferred a set of types for a certain program, then there can be no expression that triggers a not-a-function error inside of the program *fundec*, given the types, equivalence information, and coverage of the training run. Note that applications inside the top-level expression e_2 **are** checked at run time because of the application of *mon* in the MRun rule.

Before we begin with the proof sketch, we introduce a simulation to relate a training run to a monitoring run and we define stability.

Definition 1 (Simulation). The simulation relation on $H_t; S_t$ and $H_m; S_m$ under equivalence mapping $\mathcal{P}$, denoted by $H_t; S_t \sim_{\mathcal{P}} H_m; S_m$, holds iff the following holds.

- $\forall x \in dom(S_t)$, $S_t(x) = l_t : \alpha_t$ iff $S_m(x) = l_m$ and $runtype_{H_m}(l_m) \in \mathcal{P}(\alpha_t)$.
- $\forall l_t \in dom(H_t)$, whenever $H_t(l_t) = obj_t : \alpha$ such that $obj_t.p = v_t : \alpha'$, we have $\mathcal{P}(\alpha') = \mathcal{P}(\alpha.p)$.
- $\forall l_m \in dom(H_m)$, whenever $H_m(l_m) = obj_m$ such that $obj_m.p = v_m$, we have $runtype_{H_m}(v_m) \in \mathcal{P}(runtype_{H_m}(l_m).p)$.

Definition 2 (Training heap stability). H_t is training-stable under equivalence mapping $\mathcal{P}$ iff, for all $l_t \in dom(H_t)$, whenever $H_t(l_t) = obj : \alpha$ such that $obj.p = v_t : \alpha'$, we have $\mathcal{P}(\alpha') = \mathcal{P}(\alpha.p)$.

Definition 3 (Monitoring heap stability). H_m is monitoring-stable under equivalence mapping $\mathcal{P}$ iff, for all $l_m \in dom(H_m)$, whenever $H_m(l_m) = obj$ such that $obj.p = v_m$, we have $runtype_{H_m}(v_m) \in \mathcal{P}(runtype_{H_m}(obj).p)$.

Lemma 1 (Simulation Splitting). *Suppose that $H_t; S_t \sim_{\mathcal{P}} H_m; S_m$. Then H_t is training stable and H_m is monitoring stable.* □

Lemma 2 (Every training heap is stable). *For all heaps in the training run it holds that the heap is training-stable.* □

Lemma 3 (Simulation from stability). *Suppose that H_t is training stable, H_m is monitoring stable, and $S_t \sim_{\mathcal{P}} S_m$ under H_m (i.e., Item 1 in Definition 1 is the mean). Then $H_t; S_t \sim_{\mathcal{P}} H_m; S_m$.* □

Lemma 4 (Preservation). *Suppose there is a training derivation $fundec^* \triangleright e_0 \uparrow \mathcal{T}, \mathcal{P}, \Phi$ with a subderivation for the judgment $H_t; S_t; e_1 \longrightarrow H'_t; S'_t; \_ : \bar{\tau}_t \mid \_ ; \phi_t$. Let H_m, S_m, and ϕ_m be such that $H_t; S_t \sim_{\mathcal{P}} H_m; S_m$ and $\phi_m = \phi_t$.*

If $H_m; S_m; e_1 | \phi_m \longrightarrow R$, then $R = H'_m; S'_m; v_m | \phi'_m$, $runtype_{H_m}(v_m) \in \mathcal{P}(\bar{\tau}_t)$ and $H'_t; S'_t \sim_{\mathcal{P}} H'_m; S'_m$. □

With the definitions and lemmas listed above we can prove Theorem 1.

Proof. (Sketch) The proof is by induction on the derivation of $H_m; S_m; e_1 | \phi_m \longrightarrow R$ in the monitoring semantics. The only difficult case is dealing with MCall.

On the callee and argument e and e' we can just apply the induction hypothesis. At this point, we also obtain that the simulation relation must hold.

To proceed, we need to find a subderivation in the training semantics that is suitable for executing the function body, that is, its entry state must simulate the current monitoring state. As we have simulation after executing e and e', we know that the (type variable of the) function e^f_m from the monitoring run is in the equivalence set of the (type variable of the) function e^f_t. This situation can only arise if the function was called at some point in the training run.

Now, our preconditions hold and we can apply the induction hypothesis once more to e^f_m. The only thing left to show is that we end up with simulation again. From Lemma 1 we have that H'''_m is monitoring stable, from Lemma 2 we have that H'''_t is training stable. With Lemma 3 we obtain $H'''_t; S'''_t \sim_{\mathcal{T}} H'''_m; S'''_m$. □

The full proof is available elsewhere [15]. Lemma 4 only holds within the execution. We need to do some extra work at top level, which is explicated in the following lemma.

Lemma 5 (Top-Level Preservation). *Let $fundec^* \triangleright e_0 \uparrow \mathcal{T}, \mathcal{P}, \Phi$ be a training execution. Let H_m be such that monitoring heap simulation holds.*

If $H_m; S_m; mon(e_2) | \phi \longrightarrow R$, then $R = H'_m; S'_m; v_m | \phi'$ and H'_m is monitoring stable.

Proof. We perform induction on the monitoring semantics. In all cases except $mon(e_2) \equiv e.n = e'$ and $mon(e_2) \equiv e(e')$ can we directly apply the induction hypothesis.

In the case where $mon(e_2) \equiv \underline{e.n = e'}$, we apply the induction hypothesis to both e and e'. We now need to show that H'''_m is monitoring stable. This heap has one updated field. For this field, it must hold that $runtype_{H''_m}(v_m) \in \mathcal{P}(runtype_{H''_m}(l_m).n)$. But this precondition is enforced by rule MTLPropSet.

If $mon(e_2) \equiv \underline{e(e')}$, then the derivation rule enforces that e must result in a function. □

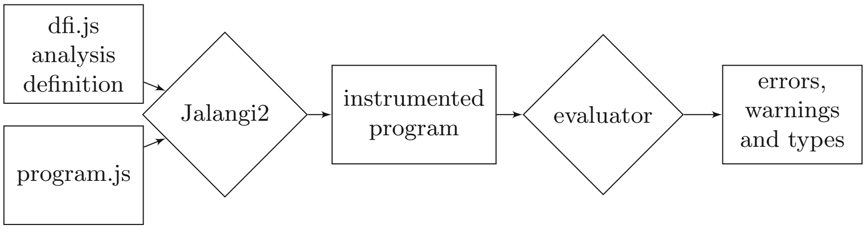


Fig. 8. Analysis pipeline diagram

4 Implementation

Our implementation is based on the formal system. During execution, the analysis observes what types occur by instrumenting the program to collect constraints. Based on these observations, it infers the types of the program.

4.1 Overview

The instrumentation is performed with Jalangi2 [11], a dynamic analysis framework for JavaScript. Figure 8 gives an overview of the instrumentation. The framework takes the original JavaScript program and instruments it according to the analysis definition. Running the instrumented program yields errors, warnings, and inferred types.

The constraints that are collected have the form (*base*, *property*, *type*), where *base* is the item which has the *property*, and *type* is the type of that property. There are three kinds of base items: objects, functions, and frames. For an object, a property represents a field, for a function, a property is either an argument or **return**, and for a frame, a property is a variable.

Types are defined as (*type*, *value*, [*location*]), where *type* is a primitive JavaScript type (number, boolean, string, undefined, null, object, function), *value* a primitive value, and [*location*] a list of source locations where the type has been observed. Values are included in the type if only one value is observed for a particular property. Otherwise, top ($\top$) is reported.

Programmers may annotate their programs with trusted type signatures for both functions and frames. These signatures are extracted during evaluation and verified against the observed types.

Programs are instrumented with constraint collection in the following places.

Function invocation. For each argument passed to the function, a constraint of the form (*fname*, *arg_n*, *type*) is generated, where *fname* is the name of the function invoked, n the index of the argument, and *type* the type of the argument. We also generate a constraint for the return value. Additionally, we check if the function is used as a constructor. If so, we also constrain the new object.

Field read. On a field read, we traverse the prototype chain to find the object providing the field. Then we constrain the provider to contain that property with the type of the value that was read.

Field write. When a field is written, we constrain the object to have that field with the type of the value we assign to it.

Variable read. When a local variable is read, we constrain the current frame that the variable belongs to.

Variable write. Same as with variable read, we constrain the current frame.

Literal string. Type annotations are provided as literal strings in the source code. From these annotations, we generate trusted constraints.

After the program has been executed, the constraints are processed. First, the constraints are solved. Then, the analysis check for type errors. Type errors are defined as conflicts between the annotated type and the inferred type. For each annotation, we check if it matches the inferred types. If not, an error is issued.

Besides errors, which are clashes between type annotations and inferred types, the algorithm also reports warnings. Warnings are issued when type inconsistencies are found. Roughly, an inconsistency is defined as one property having more than one type. In practice, it turns out that there are several cases where it is fine to have more than one type for a property. Pradel et al. [16], who have implemented a dynamic type inconsistency analysis for JavaScript, suggest methods for pruning inconsistencies that are probably not problematic. We implemented some of their methods.

Null-related warning. The value `null`, unlike `undefined`, does not occur in JavaScipt, unless the programmer explicitly assigns it. Hence, the type `Null` only occurs intentionally, so that `null`-related warnings can be pruned.

Degree of inconsistency. Polymorphic code generates many inconsistency warnings, which are most likely false positives. We therefore define a maximum number of types (i.e., 2) that we consider to be inconsistent.

Max difference. Besides pruning on base types, we also measure the difference between object types. The programmer can set the maximum difference between object types that should cause a warning.

These pruning metrics are only applied to warnings, and can be configured or turned off by the programmer.

4.2 Complete Example

This subsection explores a complete example to demonstrate how a programmer can use the implementation. Listing 1 shows the source code for the program "access-nsieve" from the SunSpider benchmark [22]. The program calculates three large prime numbers.

The program has been augmented with three type annotations, on lines 4, 13 and 29. These annotations are straightforward and result from inspecting the source code.

```
// The Great Computer Language Shootout
// http://shootout.alioth.debian.org/
//
// modified by Isaac Gouy
"function pad:{number->number->string}"
function pad(number,width){
   var s = number.toString();
   var prefixWidth = width - s.length;
   if (prefixWidth>0){
      for (var i=1; i<=prefixWidth; i++) s = " " + s;
   }
   return s;
}
"function nsieve:{number->Array->number}"
function nsieve(m, isPrime){
  var i, k, count;

   for (i=2; i<=m; i++) { isPrime[i] = true; }
   count = 0;

   for (i=2; i<=m; i++){
      if (isPrime[i]) {
         for (k=i+i; k<=m; k+=i) isPrime[k] = false;
         count++;
      }
   }
    console.log(count);
   return count;
}
"function sieve:{undefined}"
function sieve() {
    for (var i = 1; i <= 3; i++ ) {
        var m = (1<<i)*10000;
        var flags = Array(m+1);
        nsieve(m, flags);
    }
}

sieve();
```

Listing 1. access-nsieve.js from SunSpider benchmark

The type annotations are not required for the implementation to work. Alternatively, the programmer can inspect the inferred types by hand. The benefit of supplying type annotations is that the algorithm will verify them for the programmer and issue errors where a clash occurs.

The annotations will be extracted during the execution of the instrumented version of the program. Figure 9 shows the type constraints collected during execution of this program. The constraints are first condensed and then checked against the trusted type annotations.

The output of this process is shown in Fig. 10. The program returns two type errors for this program, namely "pad not observed in frame global" and "function pad not observed". These errors arise because the function "pad" is never called and therefore the function was never observed and no constraints were generated. We observed no warnings for this program.

5 Evaluation

To evaluate our implementation, we applied it to the SunSpider benchmark [22] where we hand-annotated every program with types. The results of our

```
'frame global':
   sieve:    'function sieve'
   Array:    'function Array'
   nsieve:   'function nsieve'
   result:   number
   expected: number

'frame sieve':
   sum:      number
   i:        number
   m:        number
   flags:    Array
```

```
'frame nsieve':
   i:        number
   m:        number
   isPrime:  Array
   count:    number
   k:        number

'function nsieve':
   arg0:     number
   arg1:     Array
   return:   number

'function sieve':
   return:   number
```

Fig. 9. Constraints generated for Listing 1, omitting values and locations

```
We detected 2 type error(s)

pad not observed in frame global
function pad not observed

We inferred the following types:

frame global has the following properties:
    sieve with type: function sieve
    Array with type: function Array
    nsieve with type: function nsieve
    result with type: number(14302)
    expected with type: number(14302)
frame sieve has the following properties:
    sum with type: number(T)
    i with type: number(T)
    m with type: number(T)
    flags with type: Array
frame nsieve has the following properties:
    i with type: number(T)
    m with type: number(T)
    isPrime with type: Array
    count with type: number(T)
    k with type: number(T)
function nsieve has the following type:
    arg0 number(T) -> arg1 Array -> return number(T)
function sieve has the following type:
    return number(14302)
```

Fig. 10. Output for access-nsieve.js

evaluation are listed in Fig. 11. All three aforementioned pruning methods were turned on. Programs that did not result in an error or warning are not listed.

Most errors (114) are caused by unused code. This code is annotated, but not executed so that no constraints are generated. No constraints means no types can be inferred, so the annotations cannot coincide with the inferred types.

Fifteen errors are artifacts of our type annotation system, that turned out to be too limited for two programs in the benchmark. The annotation language does not allow recursive types.

In three cases, errors were caused by native functions. Here the problem is that we are unable to generate constraints for native code.

Seven of the 139 errors were actual programming errors. The programs "crypto-md5" and "crypto-sha1" both contained problematic code. These problems were also discovered by Pradel et al. [16].

When looking at the inconsistency warnings, we observe 17 true warnings. In "3d-cube", some function returns either undefined or an Array, depending on the arguments. This could lead to problems when accessing the Array. Both "crypto-md5" and "crypto-sha1" resulted in warnings, identifying the same problems. "date-format-xparb" contains the function "leftPad", which has an inconsistent return type. This problem is also found by Pradel et al. [16].

As shown by the results and discussion above, our analysis yields useful errors and warnings that can be used by programmers to increase the quality of their programs.

		Total	3d-cube	3d-raytrace	access-nbody	access-nsieve	bitops-nsieve-bits	crypto-aes	crypto-md5	crypto-sha1	date-format-tofte	date-format-xparb	math-cordic	math-partial-sums	regexp-dna	string-validate-input
Errors	Unused Code	114	3	2	0	2	2	8	18	18	47	0	2	11	1	0
	Type Limits	15	0	9	6	0	0	0	0	0	0	0	0	0	0	0
	Native Fncts	3	0	0	0	0	0	0	0	0	0	0	0	0	0	3
	True Errors	7	0	0	0	0	0	0	6	1	0	0	0	0	0	0
Warnings	False	13	0	6	0	0	0	1	0	0	5	0	0	0	0	1
	True	17	1	0	0	0	0	0	12	2	0	2	0	0	0	0

Fig. 11. Breakdown of errors and warnings

6 Related Work

Quite some work has been done on bringing type checking and type inference to object oriented dynamic languages such as Ruby, Python and JavaScript.

Anderson and Giannini describe a formal static type inference system for a core JavaScript language with limited syntax [2]. This work builds upon previous work by the same authors together with Drossopoulou [3]. They show that this type system is sound.

Thiemann lays the groundwork for a static type system for JavaScript [24]. In this work, he also presents a JavaScript core language. For this language, typing is defined and type soundness is proved. This core language and the type structure are used in later work with Jensen and Møller to construct a static analyzer for JavaScript [12]. Their analyzer is based on the standard monotone framework with significant extensions to improve precision. Flow graphs are constructed and analysis lattices and transfer functions are presented. The downside of their method is that it is quite intricate and therefore hard to implement.

Facebook has also developed a static type inference system for JavaScript called Flow [6]. There are no formal publications about this system, but according to Facebook, it is based on control and data-flow analysis and aims at inferring types and finding type errors.

Instead of trying to implement a static type system for JavaScript directly, many alternative strategies have been explored to tackle this problem and to overcome the shortcomings of static type inference for dynamic languages. Lerner et al. present TeJaS, an extensible type systems for JavaScript [14]. Chugh et al. have developed Dependent JavaScript (DJS), wich is a typed dialect of JavaScript [5]. Ren and Foster have worked on doing just-in-time static type checking [17].

Furr et al. introduce a static type inference algorithm for Ruby [7]. Their implementation, called DRuby, is similar in complexity to the aforementioned systems for JavaScript. The authors reduced the burden for their implementation by compiling Ruby to an intermediate language, which has an explicit flow.

For Python, Michael Salib developed Starkiller, a comprehensive static type inference system [19]. Starkiller aims to remove the burden of constantly checking types at run time before every operation.

All these approaches are static analyses. JavaScript is a dynamic language and many properties of programs including types are only known at run time. As noted by Jakob et al. [10], static analyses either yield many false positives or restricts the expressiveness of the language. Cartwright and Fagan introduced the concept of Soft Typing to overcome these limitations [4]. They argue that both static and dynamic typing have their drawbacks and that soft typing could potentially provide the best of both worlds. The idea is to do some static type inference first and insert dynamic checks in cases where static inference falls short. Cartwright and Wright implemented such a system for Scheme [25]. More recent work on gradual typing further investigates these ideas [21].

Soft typing has been applied to JavaScript by Hackett and Guo in Spider-Monkey [8]. Their hybrid inference algorithm first performs a static "may have type" analysis on the program. This analysis generates constraints and identifies at what points in the program the constraints may be incomplete. Using this information, type barriers are inserted in the program. During execution, a "must have type" analysis is performed, using the previously inserted information. The type information is used to reduce the run time of the program by omitting some run-time type checks. The only information reported back to the programmer is how many times a dynamic check was needed. More recently, Swamy et al. introduce TS*, a sound gradual type system for JavaScript [23]. An obvious downside to hybrid approaches like soft typing is that a complex static type inference system has to be developed.

Pradel et al. [16] present a dynamic type inconsistency analysis for JavaScript, called TypeDevil. Their system is implemented with the Dynamic Analysis Framework Jalangi2 [20]. It checks JavaScript programs for inconsistent properties, which have more than one type. However, they only develop a practical implementation and do not present a complete formal type inference system. An et al. [1] present a complete dynamic inference algorithm for Ruby. They note that doing a dynamic analysis has several benefits. Implementing such an analysis is much easier and less error prone than a static or hybrid one, since one does not have to capture the whole language and every possible flow. Furthermore, the results respect flow sensitivity. Similar results are achieved by Saftoiu, who has developed JSTrace, a dynamic type discovery system for JavaScript, based on program traces [18].

7 Conclusion

We show that dynamic flow analysis for JavaScript is feasible. To demonstrate that the general idea is useful, we develop a formal system for a JavaScript core language and prove its soundness.

To demonstrate that the concept of dynamic flow analysis for JavaScript is also useful in practice, we develop an implementation based on the same principles as the formal system. We implemented a prototype dynamic flow analysis system for JavaScript. We evaluated our system on benchmark programs. From this evaluation we obtained useful errors and warnings that allow developers to improve the quality of their JavaScript code.

References

1. An, J.D., Chaudhuri, A., Foster, J.S., Hicks, M.: Dynamic inference of static types for Ruby. In: Ball, T., Sagiv, M. (eds.) Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, 26–28 January 2011, pp. 459–472. ACM (2011)
2. Anderson, C., Giannini, P.: Type checking for JavaScript. Electr. Notes Theor. Comput. Sci. **138**(2), 37–58 (2005)
3. Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for JavaScript. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 428–452. Springer, Heidelberg (2005). https://doi.org/10.1007/11531142_19
4. Cartwright, R., Fagan, M.: Soft typing. In: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, 26–28 June 1991, pp. 278–292 (1991)
5. Chugh, R., Herman, D., Jhala, R.: Dependent types for JavaScript. In: Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, Part of SPLASH 2012, Tucson, AZ, USA, 21–25 October 2012, pp. 587–606 (2012)
6. Facebook Flow (2016). https://flowtype.org/. Accessed 21 June 2016
7. Furr, M., An, J.D., Foster, J.S., Hicks, M.W.: Static type inference for Ruby. In: Shin, S.Y., Ossowski, S. (eds.) Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, 9–12 March 2009, pp. 1859–1866. ACM (2009)
8. Hackett, B., Guo, S.: Fast and precise hybrid type inference for JavaScript. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, Beijing, China, 11–16 June 2012, pp. 239–250. ACM (2012)
9. Henglein, F.: Efficient type inference for higher-order binding-time analysis. In: Hughes, J. (ed.) FPCA 1991. LNCS, vol. 523, pp. 448–472. Springer, Heidelberg (1991). https://doi.org/10.1007/3540543961_22
10. Jakob, R., Thiemann, P.: A falsification view of success typing. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NFM 2015. LNCS, vol. 9058, pp. 234–247. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17524-9_17
11. Jalangi2 GitHub (2015). https://github.com/Samsung/jalangi2. Accessed 9 July 2015
12. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 238–255. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03237-0_17
13. Jones, N.D., Muchnick, S.S.: Flow analysis and optimization of Lisp-like structures. In: Aho, A.V., Zilles, S.N., Rosen, B.K. (eds.) Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, 1979 January, pp. 244–256. ACM Press (1979)

14. Lerner, B.S., Politz, J.G., Guha, A., Krishnamurthi, S.: TeJaS: retrofitting type systems for JavaScript. In: DLS 2013, Proceedings of the 9th Symposium on Dynamic Languages, Part of SPLASH 2013, Indianapolis, IN, USA, 26–31 October 2013, pp. 1–16 (2013)
15. Naus, N.: Dynamic type inference for JavaScript. Master Thesis (2015)
16. Pradel, M., Schuh, P., Sen, K.: TypeDevil: dynamic type inconsistency analysis for JavaScript. In: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, 16–24 May 2015, vol. 1, pp. 314–324. IEEE (2015)
17. Ren, B.M., Foster, J.S.: Just-in-time static type checking for dynamic languages. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, 13–17 June 2016, pp. 462–476 (2016)
18. Saftoiu, C.: JSTrace: run-time type discovery for JavaScript. Technical report, Brown University (2010)
19. Salib, M.: Faster than C: static type inference with Starkiller. In: PyCon Proceedings, Washington DC, vol. 3 (2004)
20. Sen, K., Kalasapur, S., Brutch, T.G., Gibbs, S.: Jalangi: a tool framework for concolic testing, selective record-replay, and dynamic analysis of JavaScript. In: Meyer, B., Baresi, L., Mezini, M. (eds.) Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013, Saint Petersburg, Russian Federation, 18–26 August 2013, pp. 615–618. ACM (2013)
21. Siek, J., Taha, W.: Gradual typing for objects. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 2–27. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73589-2_2
22. SunSpider 1.0.2 JavaScript Benchmark (2016). https://webkit.org/perf/sunspider/sunspider.html. Accessed 5 July 2016
23. Swamy, N., et al.: Gradual typing embedded securely in JavaScript. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, 20–21 January 2014, pp. 425–438 (2014)
24. Thiemann, P.: Towards a Type system for analyzing JavaScript programs. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 408–422. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_28
25. Wright, A.K., Cartwright, R.: A practical soft type system for Scheme. ACM Trans. Program. Lang. Syst. **19**(1), 87–152 (1997)