Dynamic Flow Analysis for JavaScript

Nico Naus[1] and Peter Thiemann[2]

[1] Utrecht University, The Netherlands
N.Naus@uu.nl
[2] Albert-Ludwigs-Universität Freiburg, Germany
thiemann@acm.org

Abstract. A static flow analysis computes a safe approximation of a program's dataflow without executing it. A dynamic flow analysis computes a similar safe approximation by running the program on test data such that it achieves sufficient coverage.
We design and implement 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 the full languages. 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 and/or there are many nontrivial primitive operations. First, the implementor 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.

Hence, we follow the ideas of Furr and others [4] who propose dynamic type inference for Ruby, a class-based scripting language where classes have dedicated fields and methods. The benefit of this 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.

In this paper, we adapt 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 [10] (i.e., 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. Alternatively, the program point of the *new* also approximates the class. 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 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.

We first construct a formal system for a JavaScript core language in Section 3. This core language simplifies some aspects of JavaScript to make our formal system to facilitate proofs. 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 using the Jalangi framework [8], which is evaluated in Section 5. Section 6 compares our work with previous work, and finally Section 7 concludes this paper.

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

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

Fig. 1. Example program written in our CoreJS language, with generated constraints

by the flow analysis system. Objects and functions are identified with the line number they were created on. These numbers are used as type variables in the constraints.

Function calls, like the one on line 18, result in three constraints. One, we constrain it to be a function (`8<=8_arg->8_ret`), two, we constrain the argument (`1_ret<=8_arg`) and lastly, we constrain the return value (`Num<=8_ret`). When a new object is created, line 15 for example, we generate the constraint `15<=13`, since we want the new object to be a subtype of the old one.

On line 14, we write a value to a property of an object. This results in the constraints `13<=[val:13_val]`, we constrain the type of the object to have the property "val", and `Num<=13_val`, we constrain the type of what we assigned to it, to be a subtype of the property.

The function `inc` does not generate any constraints, since it only accesses its local variables. The function `test` only generates constraints for the else branch, since we do not visit the Then branch.
After execution, you can infer the type of every object using these constraints.

3 Formal system

The formal system comprises several parts that are shown in Figure 2. The standard semantics evaluates a top-level expression e_1 in the context of a program *fundec*. The training semantics augments the standard semantics with type constraint collection and bookkeeping of the execution paths. Flow information in the form of a type equivalence map $\mathcal{P}$ and types $\mathcal{T}$ are inferred from the resulting constraints.

The monitoring semantics is an artifact to prove the soundness of the inferred types. It evaluates the same program, but considers a different top-level expression that represents a different input. It also takes the equivalence map and the

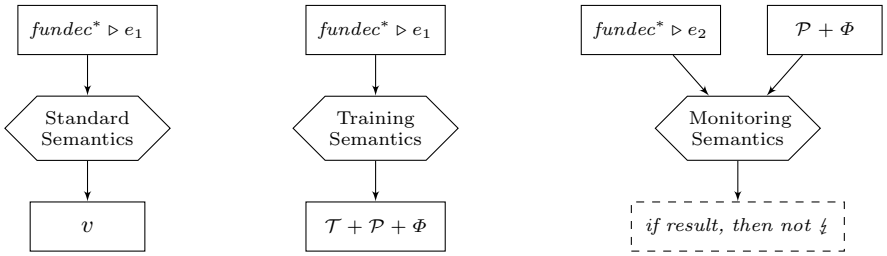

Fig. 2. Overview of formal system

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

Fig. 3. Syntax of JavaScript core language

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 trained (by the training semantics).

The formalization employs a JavaScript core language with the syntax defined in Figure 3. It features the usual JavaScript constants, variables, 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 the bracket notation to access properties. Thus, all property manipulation happens via the dot notation to an a-priori fixed set of properties. Other reflective features like eval are not considered, either.

Figure 4 declares the semantic objects for the core language. We keep state in heaps H and activation records S. A value is either a heap address l or a constant c. Objects obj contain their prototype in the first position, followed by a mapping from property names to values. There are type variables α and concrete types τ. Concrete types are composed of one of more type summands. To record executions paths, we define path sets Φ and single paths ϕ that are composed of a list of program labels of conditionals. A positive label ℓ indicates a true-branch taken, a negated label $\neg\ell$ indicates a false-branch. Lastly, a constraint set C collects constraints of the form $\bar{\tau} \leq \bar{\tau}'$. This constraint indicates that $\bar{\tau}$ is a subtype of $\bar{\tau}'$.

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

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

Fig. 4. Semantic objects and Types

3.1 Training semantics

We start with the definition of the training semantics that collects type constraints and keeps track of execution paths. We omit the standard semantics, which can be obtained by erasing the constraint and path manipulation from the training semantics.

Figure 5 defines a big-step reduction 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 augmented value ω with updated heap and activation record 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. Constraints are collected in four rules.

TNew When a new object is created, it should have at least the same type as its prototype, but it may have additional properties.

TProp A property lookup requires the property in the type.

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 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.

The TCall rule furthermore wraps a new type variable around the argument value passed to the function, it allocates 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.

The TRun-rule at the top level initializes the heap and the top-level bindings from the list of function declarations. It then evaluates the top level expression e.

$$\textsc{TVarLookup}\ \frac{S(x) = \omega}{H; S; x \longrightarrow H; S; \omega \mid \{\}; \{\}; \{\}} \qquad \textsc{TPCall}\ \frac{H; S; e \longrightarrow H'; S'; v : \_ \mid A \qquad [\![pr]\!]v = \omega}{H; S; pr(e) \longrightarrow H'; S'; \omega \mid A}$$

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

$$\textsc{TFun}\ \frac{\begin{array}{c} l = \text{fresh location} \\ \alpha = \ell \qquad H' = H\{l \mapsto (\texttt{null}, \$\text{fun} \mapsto fundec, \\ \$\text{vars} \mapsto S \downarrow_{fv(fundec)}) : \alpha\} \end{array}}{H; S; fundec \longrightarrow H'; S; l : \alpha \mid \{\}; \{\}; \{\}} \qquad \textsc{TProp}\ \frac{\begin{array}{c} H; S; e \longrightarrow H'; S'; l : \alpha \mid C; \phi; \Phi \\ C' = \alpha \leq [n : \alpha.n] \qquad H'; l.n \longrightarrow \omega \end{array}}{H; S; e.n \longrightarrow H'; S'; \omega \mid C, C'; \phi; \Phi}$$

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

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

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

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

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

Rules for prototype lookup

$$\textsc{TPropLookup}\ \frac{H(l)[n] = \omega}{H; l.n \longrightarrow \omega} \qquad \textsc{TProtoLookup}\ \frac{n \notin H(l) \qquad H; H(l)_{proto}.n \longrightarrow \omega}{H; l.n \longrightarrow \omega}$$

Fig. 5. Training semantics

Afterwards, it solves the constraints and returns a mapping from type variables to inferred types $\mathcal{T}$, an equivalence set mapping $\mathcal{P} : \alpha_1 \mapsto \{\alpha\}$, 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 [6].

3.2 Monitoring Semantics

The monitoring semantics is defined by the rule set in Figure 6. This 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 reduction judgment has the form $H; S; e \mid \phi \longrightarrow H'; S'; v \mid \phi'$, where ϕ is the path that the reduction has to follow, and ϕ' contains the remainder of the path after reduction. 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 Instead of just checking the condition, this rule also verifies that 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 initialisation, this rule executes the monitor rule on the top level expression.

There are three new rules compared with the standard semantics.

Monitor This rule applies a metafunction *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.

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.

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

$$\textsc{MVarLookup}\ \frac{S(x) = v}{H; S; x \mid \phi \longrightarrow H; S; v \mid \phi} \qquad \textsc{MPCall}\ \frac{H; S; e \mid \phi \longrightarrow H'; S'; v \mid \phi' \qquad [\![pr]\!]v = v'}{H; S; pr(e) \mid \phi \longrightarrow H'; S'; v' \mid \phi'}$$

$$\textsc{MNew}\ \frac{H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \qquad l' = \text{fresh label} \qquad \alpha = \ell \qquad obj = (v, \$\text{tyvar} \mapsto \alpha) \qquad H' = H\{l' \mapsto obj\}}{H; S; \texttt{new}^{\ell} e \mid \phi \longrightarrow H'; S; l \mid \phi'} \qquad \textsc{MVarAss}\ \frac{H; S; e \mid \phi \longrightarrow H'; S'; v \mid \phi' \qquad S'' = S'\{x \mapsto v\}}{H; S; x = e \mid \phi \longrightarrow H'; S''; v \mid \phi'}$$

$$\textsc{MFun}\ \frac{l = \text{fresh location} \qquad \alpha = \ell \qquad H' = H\{l \mapsto (\texttt{null}, \$\text{fun} \mapsto fundec, \$\text{vars} \mapsto S \downarrow_{fv(fundec)}, \$\text{tyvar} \mapsto \alpha)\}}{H; S; fundec^{\ell} \mid \phi \longrightarrow H; S; o \mid \phi} \qquad \textsc{MProp}\ \frac{H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \qquad H'; l.n \longrightarrow v}{H; S; e.n \mid \phi \longrightarrow H'; S'; v \mid \phi'}$$

$$\textsc{MPropSet}\ \frac{H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \qquad H'; S'; e' \mid \phi' \longrightarrow H''; S''; v \mid \phi'' \qquad H''' = H''\{l \mapsto H''(l)\{n \mapsto v\}\}}{H; S; e.n = e' \mid \phi \longrightarrow H'''; S''; v \mid \phi''} \qquad \textsc{MSeq}\ \frac{H; S; e \mid \phi \longrightarrow H'; S'; \_ \mid \phi' \qquad H'; S'; e' \mid \phi' \longrightarrow H''; S''; v \mid \phi''}{H; S; (e; e') \mid \phi \longrightarrow H''; S''; v \mid \phi''}$$

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

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

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

$$\textsc{MTLCall}\ \frac{H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \qquad H''(l) = obj \qquad \$\text{fun} \in obj \qquad H; S; e(e') \mid \phi \longrightarrow H''; S'; v' \mid \phi'}{H; S; \underline{e(e')} \mid \phi \longrightarrow H''; S'; v' \mid \phi'} \qquad \textsc{Error}\ \frac{H; S; e \mid \phi \longrightarrow H'; S'; l \mid \phi' \qquad H'; S'; e' \mid \phi \longrightarrow H''; S''; v \mid \phi'' \qquad H''(l) = obj \qquad \$\text{fun} \notin obj}{H; S; e(e') \mid \phi \longrightarrow \lightning}$$

$$\textsc{MRun}\ \frac{(H, S) = \text{initialize}(fundec^*) \qquad H; S; mon(e) \mid \{\} \longrightarrow \_ ; \_ ; v \mid \{\}}{\mathcal{T}; \mathcal{P}; \Phi \vdash fundec^* \triangleright e \uparrow v} \qquad \textsc{Monitor}\ \frac{\{\}; \{\}; mon(e) \mid \{\} \lightning}{\mathcal{T}; \mathcal{P}; \Phi \vdash fundec^* \triangleright e \lightning}$$

Rules for prototype lookup

$$\textsc{MPropLookup}\ \frac{H(l)[n] = v}{H; l.n \longrightarrow v} \qquad \textsc{MProtoLookup}\ \frac{n \notin H(l) \qquad H; H(l)_{proto}.n \longrightarrow v}{H; l.n \longrightarrow v}$$

Fig. 6. Monitoring semantics rules

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 *such that evaluation results in* $\lightning$ *under the inferred types and traversed paths, notated as* $\mathcal{T}; \mathcal{P}; \Phi \vdash fundec^* \triangleright e_2 \lightning$.

In words, 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.

We introduce a simulation to relate a training run to a monitoring run.

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.*

- $\forall x \in dom(S_t)$, $S_t(x) = l_t : \alpha_t \leftrightarrow S_m(x) = l_m$ *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 1is 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* $H_t'; S_t' \sim_{\mathcal{P}} H_m'; S_m'$.

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 get 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. Since 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. The only way that this can occur is if it was called at some point in the training run.

Now, our preconditions hold and we can apply the induction hypothesis once more on 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. Now with lemma 3 we have that $H'''_t; S'''_t \sim_{\mathcal{T}} H'''_m; S'''_m$.

Lemma 4 only holds within the execution. We need to do some extra work at top level. There we want to prove the following.

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 on 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.

4 Implementation

Our implementation is based on the same principles as the formal system. During execution, we observe what types occur by instrumenting the program to collect constraints. Based on these observations, we infer the types in the program.

We perform the instrumentation with the Jalangi2 framework [8]. Figure 7 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 have the form *(base, property, type)*, where *base* is the item that 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

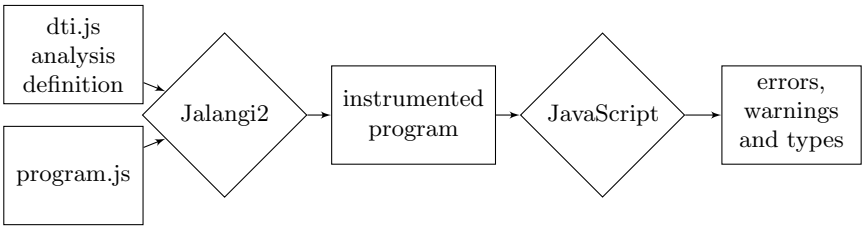

Fig. 7. Analysis pipeline diagram

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, object, function, null), *value* a primitive value, and *[location]* a list of source locations where the type is observed.

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, argn, 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 thevalue 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 condensed. Then, we 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 added.

After checking for errors, we look for inconsistencies. 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. [11] suggest methods for pruning inconsistencies that are probably not problematic. We implemented some of their methods.

	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
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 Functions	3	0	0	0	0	0	0	0	0	0	0	0	0	0	3
Errors	7	0	0	0	0	0	6	1	0	0	0	0	0	0	0
False Warnings	13	0	6	0	0	0	1	0	0	5	0	0	0	0	1
True Warnings	17	1	0	0	0	0	0	12	2	0	2	0	0	0	0

Fig. 8. Breakdown of errors and warnings

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 Another symptom of polymorphic code is that widely different objects may be passed to a generic function. Therefore, we allow the programmer to set the maximum difference between the types. If there are more differences than this number, the warning is pruned and considered to be a false positive due to intentionally polymorphic code.

5 Evaluation

To evaluate our implementation, we applied it to the SunSpider benchmark where we hand-annotated every program with types. The results of our evaluation are listed in Figure 8. All three aforementioned pruning methods were turned on. Programs that did not result in an error or warning are not listed.

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

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. [11].

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 as noticed above. "date-format-xparb" contains the function "leftPad", which has an inconsistent return type. This problem is also found by Pradel et al. [11].

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.

6 Related Work

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

Thiemann lays the groundwork for a static type system for JavaScript [4]. 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 [9]. This analyzer is based on the standard monotone framework with some extensions to boost 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.

Furr et al. introduce a static type inference algorithm for Ruby [4]. 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 [12]. Starkiller aims to remove the burden of constantly checking types at run time before any operation is done.

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 Jakobs et al. [7], static analysis either yields many false positives or restricts the expressiveness of the language. Cartwright and Fagan introduced the concept of Soft Typing to overcome these limitations [15]. 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 [16]. More recent work on gradual typing further investigates these ideas [14].

Soft typing has been applied to JavaScript by Hackett and Guo in Spider-Monkey [5]. 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 the 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. An obvious downside to hybrid approaches like soft typing is that a complex static type inference system has to be developed.

Pradel et al. [11] present a dynamic type inconsistency analysis for JavaScript, called TypeDevil. Their system is implemented with the Dynamic Analysis Framework Jalangi [13]. 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 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.

7 Conclusion

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

To demonstrate that the concept of dynamic flow analysis for JavaScript is also useful in practice, we developed 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, January 26-28, 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 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3586, pp. 428–452. Springer (2005)
4. 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, March 9-12, 2009. pp. 1859–1866. ACM (2009)

5. 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 '12, Beijing, China - June 11 - 16, 2012. pp. 239–250. ACM (2012)
6. Henglein, F.: Efficient type inference for higher-order binding-time analysis. In: Hughes, J. (ed.) Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA, August 26-30, 1991, Proceedings. Lecture Notes in Computer Science, vol. 523, pp. 448–472. Springer (1991)
7. Jakob, R., Thiemann, P.: A falsification view of success typing. In: Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9058, pp. 234–247. Springer (2015)
8. Samsung/jalangi2 GitHub. https://github.com/Samsung/jalangi2 (2015), [Online; accessed 9-July-2015]
9. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for javascript. In: Palsberg, J., Su, Z. (eds.) Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5673, pp. 238–255. Springer (2009)
10. 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, January 1979. pp. 244–256. ACM Press (1979)
11. 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, May 16-24, 2015, Volume 1. pp. 314–324. IEEE (2015)
12. Salib, M.: Faster than C: Static type inference with starkiller. PyCon Proceedings, Washington DC 3 (2004)
13. 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'13, Saint Petersburg, Russian Federation, August 18-26, 2013. pp. 615–618. ACM (2013)
14. Siek, J.G., Taha, W.: Gradual typing for objects. In: Ernst, E. (ed.) ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4609, pp. 2–27. Springer (2007)
15. Wright, A.K., Cartwright, R.: A practical soft type system for scheme. In: LISP and Functional Programming. pp. 250–262 (1994)
16. Wright, A.K., Cartwright, R.: A practical soft type system for scheme. ACM Trans. Program. Lang. Syst. 19(1), 87–152 (1997)