Reasonably-Most-General Clients for JavaScript Library Analysis

Erik Krogh Kristensen
Aarhus University
erik@cs.au.dk

Anders Møller
Aarhus University
amoeller@cs.au.dk

***Abstract*—A well-known approach to statically analyze libraries without having access to their client code is to model all possible clients abstractly using a *most-general client*. In dynamic languages, however, a most-general client would be too general: it may interact with the library in ways that are not intended by the library developer and are not realistic in actual clients, resulting in useless analysis results. In this work, we explore the concept of a *reasonably-most-general client*, in the context of a new static analysis tool REAGENT that aims to detect errors in TypeScript declaration files for JavaScript libraries.**

By incorporating different variations of reasonably-most-general clients into an existing static analyzer for JavaScript, we use REAGENT to study how different assumptions of client behavior affect the analysis results. We also show how REAGENT is able to find type errors in real-world TypeScript declaration files, and, once the errors have been corrected, to guarantee that no remaining errors exist relative to the selected assumptions.

I. INTRODUCTION

TypeScript has become a popular alternative to JavaScript for web application development. TypeScript provides static type checking, but libraries are still implemented mostly in JavaScript. These libraries use separate type declaration files to describe the typed APIs towards the TypeScript application developers. The DefinitelyTyped repository contains 5 677 such type declaration files as of February 2019 [1]. Previous work has shown that there are numerous mismatches between the type declarations in these files and the library implementations, causing spurious type errors and misleading IDE suggestions when used by application developers [12], [16], [28].

Existing approaches in the literature for detecting such mismatches include TSCHECK [12], TSTEST [17], and TPD [28]. TSCHECK applies light-weight static analysis of the library functions, but the analysis is unsound, and errors can therefore be missed. Also, TSCHECK only reports an error if an inferred function result type is disjoint from the declared one, which makes TSCHECK miss even more errors. TSTEST is based on automated testing and as such inherently underapproximates the possible behaviors of libraries, resulting in no more than 50% statement coverage of the library code on average [17]. TPD similarly uses dynamic analysis, although with existing test suites to drive execution instead of automated testing, and therefore also misses many errors.

Another line of work involves static analysis for JavaScript. By conservatively over-approximating the possible behavior of the program being analyzed, static analysis tools can in principle detect all type errors exhaustively. FLOW uses fast type inference, but it is incapable of reasoning about unannotated library code [9]. Similarly to TypeScript, FLOW relies on type declaration files for interacting with untyped library code, and FLOW blindly trusts these files to be correct. Several static analyzers have been specifically designed to detect type-related errors in JavaScript programs, without requiring type annotations. State-of-the-art tools are TAJS [4], [13], SAFE [22], and JSAI [14]. However, these analyzers have not been designed for analyzing libraries without client code, and they do not exploit or check TypeScript types.

Although previous approaches have been proven useful for finding mismatches between the TypeScript declaration file and the JavaScript implementation of a given library, *none of them can guarantee that they find all possible type mismatches* that a realistic client may encounter when using the declaration file and the library. In this work we present a novel framework that aims to complement existing techniques by having the ability to *exhaustively* find all possible type mismatches, or prove that there are none, relative to certain reasonable assumptions.

The approach we take is to build on an existing static type analysis tool for JavaScript, specifically the TAJS analyzer. The first challenge is that such tools have been designed with a closed-world assumption, i.e., where the entire program is available for the analysis, whereas we need to analyze library code without having access to client code. In the past, the problem of analyzing an open program using an analysis designed with a closed-world-assumption has been addressed through the notion of a *most-general client* for the library [3], [25]. A most-general client is an artificial program that interacts with the library in all possible ways, thereby soundly modeling all possible actual clients. However, we find that the concept of a most-general client does not work well for a dynamic language like JavaScript. Due to the poor encapsulation mechanisms in JavaScript, clients can in principle interfere with the library in ways that are not intended by the library developer and are not realistic in actual clients. As a simple example, a most-general client may overwrite parts of the library itself or the standard library that the library relies on, thereby breaking its functionality and rendering the static analysis results useless. For this reason, we introduce the concept of a *reasonably-most-general client* that restricts the capabilities of the artificial client. Our framework provides a methodology for library developers to exhaustively detect possible type mismatches under different assumptions of the client behavior.

Existing static type analysis tools for JavaScript, including TAJS, have not been designed with support for TypeScript type declarations, however, it turns out that TypeScript's notion of types fits quite closely with the abstract domains used by TAJS. A bigger challenge is that the TypeScript type declarations for libraries are written in separate files, with no clear link between, for example, the type declaration for a function and the code that implements that function. JavaScript libraries initialize themselves dynamically, often in complicated ways that are difficult to discover statically. To this end, we adapt the feedback-directed approach by TSTEST, which incrementally discovers the relation between the type declarations and the library implementation, to a static analysis setting.

By building on an existing static analysis tool for JavaScript, we naturally inherit some of its limitations (as well as any improvements made in the future). Although much progress has been made to such tools within the last decade, JavaScript libraries are notoriously difficult to analyze statically, even when considering simple clients [4], [20], [22], [24]. The goal of this paper is not to improve the underlying static analysis tool, but to explore how such a tool can be leveraged to exhaustively find errors in TypeScript declaration files. Usually, when JavaScript analyzers encounter difficulties regarding scalability and precision, they do not degrade gracefully but fail with an error message about a catastrophic loss of precision, inadequate memory, or a timeout. To partly remedy this problem, our framework selective stops the analysis of problematic functions.

In summary, the contributions of our work are the following.

- We introduce the concept of a *reasonably-most-general client (*RMGC*)* that restricts the traditional notion of most-general clients to enable static analysis of JavaScript libraries (Section III). Some of the restricting assumptions that we consider are necessary for the analysis to have meaningful results; others provide a trade-off between generality of the RMGC, i.e. what errors can possibly be found, and false positives in the analysis results.
- We discuss how to incorporate abstract models of the different variations of RMGCs on top of an existing static analysis tool (TAJS) that has originally been designed for whole-program JavaScript analysis, thereby enabling open-world analysis of JavaScript libraries (Section V). By adding support for creating abstract values from TypeScript types and, conversely, type-checking abstract values according to TypeScript types, the resulting analysis tool can exhaustively detect errors in TypeScript declaration files for JavaScript libraries. We adapt the feedback-directed technique from TSTEST to incrementally discover the relation between the type declarations in the TypeScript declaration files and the program code in the JavaScript implementation.
- We experimentally evaluate our tool, REAGENT, on 10 real-world libraries (Section VI). REAGENT uses TAJS largely unmodified, and we believe it could easily be ported to similar analyzers, such as SAFE or JSAI. With REAGENT, we detected and fixed type errors in these 10 libraries (totaling 27 lines changed across 7 libraries), with the guarantee that the fixed declaration files do not contain any remaining type errors, under the assumptions of the RMGC. Moreover, we investigate the impact of each optional assumption of the RMGC by evaluating the trade-off between generality of the RMGC and accuracy of the analysis.

II. MOTIVATING EXAMPLE

To motivate our approach, we begin by describing an example from the `semver` library,[1] which is a small library for handling version numbers according to the semantic versioning scheme. A simplified portion of the library implementation is shown in Figure 1a. The constructor in line 1 returns a `SemVer` object if the argument string matches the semantic versioning scheme. The DefinitelyTyped repository hosts a declaration file for `semver`, a small part of which is shown in Figure 1b. Our goal is to detect mismatches between the declaration file and the implementation of the library. For this reason, we need to consider how clients may interact with the library.

Because of the dynamic nature of JavaScript, a client of the library could in principle interact with the library in ways that are not possible with statically typed languages. A most-general client interacting with the `SemVer` library would perform every possible action, including replacing the `format` function (declared in line 15) with a function that just returns a constant string. The `format` function in `SemVer` is responsible for creating and setting the `version` property of the `SemVer` class (line 16), and thus replacing `format` will cause all objects created by the `SemVer` constructor to lack the `version` property. A most-general client will thus find that the declaration file, which states that the `version` property is present, may be erroneous. If the library developer did not intend for clients to overwrite the `format` function, and no client developer would ever consider doing so, then the missing `version` property is a false positive.

Our *reasonably-most-general client* (RMGC) works under a set of assumptions, described in Section III, that restrict the actions performed compared to a truly "most general" client. One of these assumptions includes that the RMGC does not overwrite library functions, and under this assumption the false positive related to the `version` property would not occur.

It is nontrivial for any automated technique that relies on concrete execution to provide sufficient coverage of the possible behavior of `semver`. For instance, the simple random string generator in TSTEST will generate a string matching the semantic versioning scheme with a probability of around $1/10^{13}$, and without such a string, no `SemVer` object will ever be created, so most of the library will remain untested. By the use of abstract interpretation, we overcome the shortcomings of the techniques that rely on concrete executions. This approach allows us to evaluate the `SemVer` constructor abstractly with an indeterminate string value passed as parameter. When the `SemVer` constructor is evaluated abstractly, the condition in line 5 is considered as possibly succeeding by the abstract interpreter, so that an abstract `SemVer` object is constructed and returned, which is necessary to test the rest of the library.

[1] https://github.com/npm/node-semver

```
1  function SemVer(version) {
2    ...
3    if (version.length > MAX_LENGTH) throw new TypeError()
4    var m = version.trim().match(REGEXP);
5    if (!m) throw new TypeError();
6    ...
7    // numberify any prerelease numeric ids
8    this.prerelease = m[4].split('.').map((id) => {
9      if (/^[0-9]+$/.test(id) && +id >= 0 && +id < MAX_INT)
10       return +id;
11     return id;
12   });
13   this.format();
14 }
15 SemVer.prototype.format = function() {
16   this.version =
17     this.major + '.' + this.minor + '.' + this.patch;
18   if (this.prerelease.length)
19     this.version += '-' + this.prerelease.join('.');
20   return this.version;
21 };
```

(a) A simplified version of the `SemVer` constructor.

```
22 export class SemVer {
23   constructor(version: string | SemVer);
24   major: number;
25   minor: number;
26   patch: number;
27   version: string;
28   prerelease: string[];
29   format(): string;
30   compare(other: string | SemVer): 1 | 0 | -1;
31 }
```

(b) The declaration for `SemVer` from the TypeScript declaration file.

```
32 for path: SemVer.new().prerelease.[numberIndexer]
33   Expected string or undefined but found number
```

(c) Error reported by REAGENT for the `prerelease` property.

Figure 1: Implementation and declaration file of the `semver` library.

An example of a type mismatch that is not easily detectable by techniques that rely on concrete executions, but is found by our abstract RMGC, is a mismatch related to the `prerelease` property declared in line 28. The correct type for the `prerelease` property is `(string | number)[]`, since a conversion to `number` is attempted for every element of the array during the initialization of the `SemVer` object (lines 8–12). The type violation reported by REAGENT for this error is shown in Figure 1c.

The downside of using abstract interpretation is that REAGENT is sometimes overly conservative and may report type violations in situations where no concrete execution could lead to a type mismatch. However, in return, it finds all possible mismatches, relative to the RMGC assumptions.

III. REASONABLY-MOST-GENERAL CLIENTS

A most-general client (MGC) uses a library by reading and writing its object properties and invoking its functions and constructors. We refer to such possible uses as *actions*.

A library can be stateful, so the behavior of its functions may depend on actions that have been performed previously, so an MGC must perform any possible sequence of actions, not just single actions. Some of the function invocations performed by the MGC involve callback functions that originate from the MGC and may similarly perform arbitrary actions. An MGC may invoke the functions of a library with arguments of any type, even if the declaration file declares that the function should be called with arguments of a specific type. However, the declaration file describes a contract between the library and the client. If we know that the client does not break this contract, then the library can be blamed for any type errors that are encountered.

A *reasonably-most-general client* (RMGC) does the same as an MGC, but with certain restrictions that ensure both that the RMGC does not break the contract in the declaration file, and that the RMGC does not interact with the library in ways that are unrealistic and unintended by the library developer. The first two assumptions we describe next are necessary for the RMGC to work meaningfully, whereas three other assumptions are optional and ultimately depend on what guarantees the user of our analysis wants.

A. Respecting declared types

A necessary assumption is that an RMGC respects declared types: If a function in a library is declared as receiving, e.g., numbers as its arguments, then the RMGC only calls the function with numbers. Otherwise we would be unable to blame the library and its type declaration file for any type mismatch that occurs when the client uses the library.

Assumption 1. [**RESPECT-TYPES**] An RMGC respects the types declared in the type declaration file, when passing values to the library.

Note that because TypeScript's type system is inherently unsound [6], this assumption is not the same as requiring that the client passes the TypeScript type check without warnings.

A consequence of this assumption is that the set of possible actions is bounded by the types that appear in the declaration file, which is useful when we in Section V define the notions of abstract RMGCs and action coverage.

B. Preserving the library

As motivated in Section II, TypeScript clients can in principle overwrite library functions (like the `format` function in the example), but it is clearly unreasonable to blame the library for type errors that result from that. A possible approach is to expect that properties that are intended to be read-only are declared as such in the declaration file. TypeScript properties are writable by default but can be declared with the modifier `readonly` or `const`. However, authors of declaration files rarely use these modifiers. Additionally, some features, such as class methods, cannot easily be declared as read-only.[2]

[2]It is possible to create a read-only method by declaring it as a property with a function type, but this feature is rarely used.

Overwriting properties of standard libraries, specifically the ECMAScript standard library, the browser DOM API, and the Node.js API, may similarly cause the library under test to malfunction. For instance, the `SemVer` constructor from the motivating example depends on the functionality of the `String.prototype.trim` function from the ECMAScript standard library. Only a few of the properties of the standard libraries are marked as read-only. Sometimes some of these non-read-only properties are overwriten on purpose, for example to improve support of certain features in outdated browsers by loading polyfills.[3] Still, both regarding the library under test and the standard libraries, it is reasonable to assume that a library does not depend on the client to overwrite functions in the library, which justifies the following assumption.

Assumption 2. [**PRESERVE-LIBRARIES**] An RMGC considers all properties of the standard libraries and all properties declared with a non-primitive type[4] from the library under test as read-only and thus never writes to those properties.

This assumption does not prevent the RMGC from writing to library properties declared with primitive types (such properties are occasionally used for library configuration purposes). In contrast, writing to an undeclared property is considered a type error in TypeScript, so the RESPECT-TYPES assumption ensures that the RMGC never does so.

C. Obtaining values for property writes and function arguments

Whenever an RMGC passes an argument to a library function or writes to a property of a library object, a value of the declared type is needed. There are two ways the RMGC can obtain such a value: either the value originates from the library (and the client receives the value via a function call, for example), or the value is constructed by the RMGC itself. We refer to these as *library-constructed values* and *client-constructed values*, respectively. Even an MGC cannot construct all possible values itself—for example, a library-constructed value may be a function that has access to the internal state of the library via its free variables—so we need to take both kinds of value constructions into account.

TypeScript is *structurally typed*, meaning that when a function is declared as taking an argument of some object type, then the type system allows the function to be called as long as the argument is an object of the right structure. According to the RESPECT-TYPES assumption, an RMGC should pass any structurally correct client-constructed or library-constructed value of the desired type. However, that is not always the intent of the library developers, as the structural types may not fully describe what is expected from the arguments. For instance, the structural types in TypeScript cannot describe prototype inheritance, and sometimes a library assumes other invariants about values constructed by the library itself.

[3]https://www.w3.org/2001/tag/doc/polyfills

[4]The primitive types in TypeScript are `boolean`, `string`, `number`, `undefined`, `symbol`, and `null`.

Example 1. In the code below, taken from the `Leaflet` library,[5] the value of `this.options.tileSize` is supplied by the client, and the `tileSize` property is declared to have type `L.Point | number`. If `tileSize` is set to a value that has the same structure as `L.Point` but is not constructed by the `L.Point` constructor, then the `instanceof` check in line 3 in the program will fail, resulting in an invalid `L.Point` being constructed.

```
1 var getTileSize = function () {
2   var s = this.options.tileSize;
3   return s instanceof L.Point ? s : new L.Point(s, s);
4 }
```

The following assumption may better align with the intended use of such a library.

Assumption 3. [**PREFER-LIBRARY-VALUES** – ***optional***] When passing values of non-primitive types to the library, an RMGC uses library-constructed values if possible; client-constructed values are only used if the RMGC is unable to obtain library-constructed values of the desired types according to the type declaration file.

A recent study of JavaScript object creation [29] has found that it rarely happens that the same property read in a program uses objects that were created at different program locations, suggesting that the PREFER-LIBRARY-VALUES assumption is satisfied by most clients in practice.

String values are optionally handled in a special way. Since a string provided by the RMGC might be used in a property lookup on an object in the library, a string that is the name of a property defined on, for example, `Object.prototype` may result in a property of `Object.prototype` being accessed. The (perhaps implicit) assumption of the library developer in this case might be that clients do not use strings that are property names of prototype objects from the standard library, as such accesses could have unintended consequences.

Example 2. In the simplified code below taken from the `loglevel` library,[6] the `getLogger` function (line 3) makes sure that only one `Logger` of a given name is constructed, by checking if a property of that name is defined on the `_loggers` object (lines 4–5). If a `Logger` has already been constructed it is returned (line 8), and otherwise a new one is created (line 6). However, if the name is, for example, `toString` then the property lookup in line 4 will return the `toString` method defined on `Object.prototype`, and that method will then be returned by `getLogger` resulting in a type mismatch.

```
1 declare function getLogger(name: string) : Logger

2 var _loggers = {};
3 function getLogger(name) {
4   var logger = _loggers[name];
5   if (!logger) {
6     logger = _loggers[name] = new Logger(...);
7   }
8   return logger;
9 };
```

This observation motivates the following assumption.

[5]https://github.com/Leaflet/Leaflet

[6]https://github.com/pimterry/loglevel

Assumption 4. [NO-PROTOTYPE-STRINGS – ***optional***] An RMGC does not construct strings that coincide with the names of properties of the prototype objects in the standard libraries.

Another issue is that TypeScript's type system supports *width subtyping*, which means that for an object to match a type, the object should have all the properties declared by the type, and any undeclared property in the type can be present in the object and have any value. Therefore it seems natural that when an RMGC constructs an object of some type, the constructed object may also have undeclared properties.

However, since these undeclared properties can have arbitrary values, false positives might appear if the library reads one of these undeclared properties.

Example 3. In the simplified example below from the `uuid` library,[7] the `v4` function obtains random numbers from the `opts` object (line 4) and puts them into the `buf` array (line 6). The `opts` object can have two different types (declared in line 1). The `v4` function attempts to detect which of the two types the concrete `opts` object has, and uses this to create an array of random numbers (line 4). A client can choose to use the second variant of the `opts` object that only has declared a `rng` property, but because of width subtyping the client is technically allowed to add a property `random` of any type to that object. If the client chooses to call `v4` with such an `opts` object, then the property read `opts.random` can read any value, which in turn can cause a false positive when non-number values are written to the `buf` array (line 6).

```
1 type Opts = {random: number[]} | {rng(): number[]};
2 declare function v4(opts: Opts, buf: number[]) : number[]
3 function v4(opts, buf) {
4   var rnds = opts.random || (opts.rng || _rng)();
5   for (var i = 0; i < 16; i++) {
6     buf[i] = rnds[i];
7   }
8   return buf;
9 }
```

We therefore leave it as an optional assumption whether client-constructed objects should have undeclared properties.

Assumption 5. [NO-WIDTH-SUBTYPING – ***optional***] An object constructed by the RMGC does not have properties that are not declared in the type.

If this assumption is disabled, for client-constructed objects, all properties that are not explicitly declared may have arbitrary values of arbitrary types.

In the following sections, we demonstrate that these five assumptions are sufficient to enable useful static analysis results for JavaScript libraries.

IV. Abstract Domains in Static Type Analysis

To be able to explain how to incorporate RMGCs into the TAJS static analyzer, we briefly describe the structure of the abstract domains used by TAJS [4], [13] (and related tools like SAFE [22], and JSAI [14]).

[7]https://github.com/kelektiv/node-uuid

Algorithm 1: The iterative algorithm performed by the abstract RMGC.

Input: library source code and TypeScript declaration
1 invoke TAJS to analyze the library initialization code
2 `allState` ← abstract state after library initialization
3 `vmap` ← [library type ↦ library abstract value]
4 **do**
5 **for** all functions f in `vmap` **do**
6 *args* ← use OBTAINVALUE to get arguments for f
7 propagate `allState` and *args* to function entry of f
8 invoke TAJS to analyze new dataflow
9 **for** all functions f in `vmap` **do**
10 propagate state at function exit of f to `allState`
11 ADDLIBVAL(abstract return value of f, declared return type of f)
12 **for** all properties p in all objects o in `vmap` **do**
13 ADDLIBVAL(abstract value of p in `allState`, declared type of p)
14 **while** `allState` or `vmap` changed

TAJS is a whole-program abstract interpreter that over-approximates the flow of primitive values, objects, and functions in JavaScript programs. (We here ignore many details of the abstract domains, including the use of context-sensitivity, that are not relevant for the topic of RMGCs.) Abstract objects are partitioned by the source locations, called *allocation-sites*, where the objects are created [8]. Basically, at each program point, TAJS maintains an *abstract state*, which is a map from allocation-sites to abstract objects, and an abstract object is a map from abstract property names to abstract values. Abstract values are described by a product lattice of sub-lattices for primitive values of the different types (strings, numbers, booleans, etc., as in traditional constant propagation analysis [7]) and a sub-lattice for object values (modeled by allocation-sites, like in traditional points-to analysis [8]) and abstract function values (like in traditional control-flow analysis [26]). As in other dataflow analyses, TAJS uses a worklist algorithm to propagate abstract states through the program until a fixed-point is reached. We refer to the literature on TAJS for more details.

V. Using RMGCs in Static Type Analysis

Our static analysis is made of two components: the TAJS abstract interpreter and an *abstract* RMGC. The abstract RMGC interacts with the library by using the abstract interpreter to model the actions described in Section III. It maintains an abstract state, `allState`, that models all program states that are possible with the actions analyzed so far.

The basic steps of the abstract RMGC are shown in Algorithm 1. The abstract RMGC cannot immediately invoke all functions in the library, because the connection between the implementation of a function and the declared type of the function is only known after a reference to the function has been returned by the library. In the motivating example (Section II), if the `compare` method had been defined in the `SemVer` constructor

Algorithm 2: Handling library-constructed abstract values.

Input: an abstract value and a TypeScript type
ADDLIBVAL(*val*, *type*)
15 **if** not TYPECHECK(*val*, *type*) **then**
16 report type violation
17 *fval* ← FILTER(*val*, *type*)
18 `vmap` ← `vmap` + [*type* ↦ *fval*]

instead of being present on the `SemVer.prototype` object, a client would only be able to invoke the method after having constructed an instance of `SemVer`. Therefore, a crucial component of the abstract RMGC is a map, called `vmap`, from types in the declaration file to abstract values as used by TAJS (see Section IV). A type is modeled as an access path in the declaration file; for example, the access path `SemVer.new().minor` is the TypeScript type `number` in the `semver` declaration file. This map allows the abstract RMGC to keep track of which parts of the library have been explored so far during the analysis, and for obtaining abstract library-constructed values for further exploration. Abstract values in `vmap` that contain allocation-sites (modeling references to objects, cf. Section IV) are interpreted relative to `allState`.

The abstract RMGC first loads the library by abstractly interpreting the library initialization code (line 1 in Algorithm 1), and then setting `allState` to be the resulting abstract state (line 2). The initialization of libraries intended for use in web browsers consists of dynamically building an object that is eventually written to a property of the JavaScript global object (which is treated as a special allocation-site in TAJS). Clients then use the property of the global object as an entry point for the library API. Our abstract RMGC uses this property by inserting it into `vmap` associated with the declared library type (line 3). (Initialization of Node.js libraries works slightly differently and is ignored here to simplify the presentation.) For the `semver` example from Section II, `vmap` then maps the type `SemVer` to the abstract value that models the object produced by the library initialization code. All other entries in `vmap` initially map to the bottom abstract value, denoted $\bot$.

After the initialization phase, the abstract RMGC works iteratively (lines 4–14). In each iteration, it abstractly invokes each library function that exists in `vmap`. The auxiliary function OBTAINVALUE provides abstract values for the arguments as explained in Section V-A. By propagating[8] `allState` and the arguments to the function entry (line 7) and then invoking TAJS (line 8), the function bodies are analyzed. Next, for each of the functions, the resulting abstract state at the function exit is propagated into `allState` (line 10) and the abstract return value is collected (line 11). Similarly, for every object whose type is contained in `vmap`, all properties declared by the object type are collected (lines 12–13). (For simplicity we here ignore properties with getters and setters.)

The auxiliary function ADDLIBVAL (Algorithm 2) first type-checks the abstract value according to the declared type (lines 15–16) using the function TYPECHECK explained in Section V-B. The abstract value is then passed through a function, FILTER, that performs type refinement [15] to remove parts of the abstract value that do not match the type, and the resulting abstract value is added[9] to `vmap` (line 18).

The entire process is repeated until no more dataflow appears in `allState` and `vmap`.[10] When the fixed-point is reached, `allState` models an over-approximation of all possible states at the entries and exits of the reachable library functions. Thereby the state in the beginning of all library functions includes the side-effects from all other library functions, and the abstract RMGC therefore models all states that can result from calling the functions in any possible sequence.

[8] *Propagating* an abstract state X into Y means setting Y to the least-upper-bound of X and Y.

Algorithm 3: Algorithm for obtaining an abstract value for a given type.

Input: a TypeScript type
Result: abstract value modeling the type
OBTAINVALUE(*type*)
19 **if** *type* is primitive **then**
20 **return** create primitive value from *type*
21 **else if** *type* in stdlib **then**
22 **return** CREATENATIVE(*type*)
23 *val* ← `vmap`(*type*)
24 *obj* ← new abstract object
25 **for** all properties *p* in *type* **do**
26 *obj*[*p*] ← OBTAINVALUE(*type*[*p*])
27 **return** *val* ⊔ *obj*

A. Obtaining abstract values for library function arguments

The pseudo-code in Algorithm 3 shows how OBTAINVALUE provides abstract values for the abstract RMGC, either by producing new abstract values (to model the client-constructed values) or using abstract values from `vmap` (for the library-constructed values). Line 6 in Algorithm 1 calls OBTAINVALUE for every available function parameter type.

For primitive types, such as `number` or `string`, we let the abstract RMGC construct the value (line 20). Creating an abstract value that describes all possible values of a primitive type is trivial due to the abstract domains already supported by TAJS, as discussed in Section IV. If NO-PROTOTYPE-STRINGS is enabled, abstract string values are created accordingly.

Types declared in a standard library need special treatment. For example, the type declaration of `Function` contains no information that the value is in fact a callable function. The function CREATENATIVE (line 22) takes care of creating the right abstract values for such types; we omit the details here.

If the type is neither primitive nor from a standard library, an abstract value is created that models the relevant

[9] The '+' operator used in Algorithm 2 denotes updating using least-upper-bound; specifically, line 18 updates the `vmap` entry for *type* to become the least-upper-bound of the existing abstract value and *fval*.

[10] The lattices in TAJS have finite height, which ensures termination.

Algorithm 4: Type-checking abstract values.

Input: an abstract value and a TypeScript type
Result: true if the abstract value matches the type
TYPECHECK(*value*, *type*)

```
28  if type is primitive then
29      return true if value matches type
30  else if type in stdlib then
31      return CHECKNATIVE(value, type)
32  else if type is function then
33      return true if value is a function
34  else if value is not an object then
35      return false
36  else
37      for all properties p in type do
38          if not TYPECHECK(value[p], type[p]) then
39              return false
40      return true
```

library-constructed and client-constructed objects.[11] Library-constructed objects are taken from `vmap` (line 23), and client-constructed objects are made using OBTAINVALUE recursively by following the structure of the type in the TypeScript declaration (lines 25–26). (In case of recursive types, the abstract objects are reused to ensure termination.) When creating an object from a type, the type is used as an artificial allocation-site (line 24), which ensures that TAJS correctly models possible aliasing.[12]

The pseudo-code in lines 25–26 shows how abstract client-constructed objects are obtained when NO-WIDTH-SUBTYPING is enabled. If that assumption is disabled, the $\top$ ("top") abstract value is additionally assigned to all undeclared properties of the new abstract object.

Functions are just objects in JavaScript, and client-constructed functions are thus obtained in essentially the same way as ordinary objects. The only difference is that the artificial allocation-site is marked as being a function (not shown in the pseudo-code), which informs TAJS that the new object can be called as a function. When TAJS finds that the library invokes such a client-constructed function, each argument is processed using ADDLIBVAL and a return value is created using OBTAINVALUE.

If PREFER-LIBRARY-VALUES is enabled then we omit the client-constructed abstract value *obj* in line 27 and simply return *val* if values of the desired type can be obtained from the library according to the type declaration file, which can be implemented with a simple reachability check.

B. Type-checking abstract values

When the abstract RMGC receives a value from the library, either by invoking a function or by reading a property from an object, Algorithm 2 uses TYPECHECK to check whether the value has the right type according to the corresponding type declaration. This process is straightforward as outlined in Algorithm 4. Checking primitive types (line 29) is trivial for the same reason as creating abstract values of primitive types is trivial (see Section V-A), and types declared in a standard library need special treatment (line 31) for the same reason as creating them requires special treatment.

If an abstract value is checked against a function type, we only need to check whether the value is a function (line 33); the parameter types and the return type are not used until the function has been invoked by Algorithm 1.

Checking object types requires checking that the abstract value is an object and recursively checking the declared properties (lines 34–40). Recursive types are handled co-inductively (ignored in the pseudo-code for simplicity).

C. Coping with analysis precision and scalability issues

Our technique can in principle find all possible type mismatches that could be encountered by a client satisfying the RMGC assumptions. This is justified by TAJS being a "soundy" [19] static analysis, and by our abstract RMGC over-approximating the actions of an RMGC, meaning that there may be false positives but we should not expect false negatives. There are two possible causes of false positives: (1) imprecision of the underlying static analysis, and (2) the RMGC being "too general", lacking reasonable assumptions about how real clients may behave.

It is well-known that many real-world JavaScript libraries contain code that is extremely difficult to analyze statically [5], [21]. Inadequate precision of the static analysis may cause an avalanche of spurious dataflow, rendering the analysis results useless. We use some simple heuristics to detect if the analysis of a library function is encountering a catastrophic loss of precision or is taking too long.[13] In these cases we unsoundly stop the analysis of that function, but allow the analysis to proceed with other functions.

We define *action coverage* as the percentage of actions that were successfully performed by the abstract RMGC. An action coverage of 100% means that the abstract RMGC was able to analyze the entire library exhaustively, without stopping analysis of any functions. If the action coverage is below 100%, it means that either our analysis fails to analyze one of the functions, or that an action is unreachable typically because of a mismatch between the TypeScript declaration file and the library implementation. In the first case, our abstract RMGC may miss type violations that could be encountered by a client that invokes those functions. Yet, the analysis remains exhaustive for those clients that do not perform any of the stopped actions. Hence, when the analysis terminates, action coverage measures the portion of the library API that has been analyzed exhaustively.

[11] '$\sqcup$' in line 27 denotes the least-upper-bound on abstract values.

[12] Subtyping is handled soundly by including the allocation-sites of super-types when creating the new abstract objects.

[13] We declare a function as timed out if TAJS has used more than 200 000 node transfers to analyze it (typically corresponding to a few minutes), and we characterize a catastrophic loss of precision as a property read on an abstract value that represents at least two different standard library objects.

VI. Evaluation

Two of the RMGC assumptions described in Section III are *necessary* to obtain any useful analysis results, and the remaining three assumptions are *optional* and can be selected by the analysis user. Imposing too many restrictions on the model of the clients may prevent detection of errors due to inadequate coverage, whereas imposing too few may cause false positives and also significantly degrade analysis performance because the abstract client becomes "too general".

To evaluate the usefulness of the concept of an RMGC and the effects of the optional assumptions, we have implemented the abstract RMGC described in Section V in a tool, ReaGenT (REAsonably-most-GENeral clienT). It uses the abstract interpreter TAJS unmodified, except for a small adjustment of its context-sensitivity strategy [11]: to increase analysis precision, every time the abstract RMGC invokes TAJS to analyze a function (lines 7–8 in Algorithm 1), the context is augmented by the access path of the function (using the same notion of access paths as in Section V). This adjustment can also easily be made to other JavaScript static analyzers [14], [22].

In our evaluation we aim to answer the following main research questions.

RQ1 Does the RMGC enable static type analysis of JavaScript libraries, using an off-the-shelf, state-of-the-art whole-program static analyzer?

RQ2 How does each of the optional RMGC assumptions affect the ability of ReaGenT to detect type errors in JavaScript libraries?

As benchmarks, we have randomly selected 10 JavaScript libraries that have TypeScript declarations in the DefinitelyTyped repository. For the reasons given in Sections I and V-C, we focus on small libraries only (up to 50 LOC in the declaration file). The libraries, which are available from the npm repository,[14] and the sizes of their declaration files (measured with CLOC) are shown in the first columns of Table I. We use the latest versions of all the libraries and declaration files.

Our implementation of ReaGenT and all experimental data are available at http://brics.dk/tstools/. The experiments are performed on computer with 16GB of RAM and an Intel i7-4712MQ CPU.

A. RQ1: Does the RMGC enable static type analysis of JavaScript libraries?

No existing static analysis is capable of helping programmers find all type errors in JavaScript libraries that have TypeScript declarations, even if restricting to small libraries like the selected benchmarks. To investigate whether our RMGC enables such analysis, we perform an experiment where we run ReaGenT on the 10 libraries, using the configuration where all optional assumptions are enabled. For each reported type violation, we manually classify it as a true positive (usually an error in the type declaration file) and then fix it, or mark it as a false positive (either caused by the RMGC being too general and therefore modeling unrealistic clients, or by the underlying static analysis being too imprecise). This process is repeated until ReaGenT no longer reports any violations.

Table I: Number of lines in the JavaScript implementation, along with total lines, changed lines resulting from our fixes, and lines with false positives after the fixes, for the corresponding type declaration file.

	Impl.	Type Declaration File		
Library	Lines	Total	Changed	False positives
classnames	37	9	0	0
component-emitter	72	13	3	6
js-cookie	127	23	4	0
loglevel	176	40	7	1
mime	915	9	1	0
pathjs	183	38	5	1
platform	741	22	2	4
pleasejs	630	46	5	2
pluralize	315	13	0	0
uuid	86	23	0	0

The results are summarized in Table I, which shows how many lines were changed in each declaration file to fix true positives, and for how many lines in the fixed declaration file ReaGenT falsely reports a violation. Even without expert knowledge of the libraries, classifying the type violation reports and fixing the true positives was straightforward based on the output of ReaGenT. For the seven declaration files being fixed, we created pull requests, which were all accepted by the maintainers. Table I shows that ReaGenT can find actual errors in many libraries, which is not surprising given that previous work has shown that many type declaration files are erroneous [12], [17], [28]. More importantly, Table I shows that ReaGenT does not overwhelm the user with false positives, as there are only 14 lines containing false positives across five of the libraries.

Example 4. The `getJSON` function in the `js-cookie` library parses the value of a browser cookie. According to the declaration file, the function always returns an object:

```
1  declare function getJSON(key: string) : object;
```

The implementation (shown below) iterates through all the cookies (lines 5–11) and parses each cookie using the `JSON.parse` function. If the `key` argument is the same as the name of the cookie then the value of the cookie is returned (line 10). However, the returned value is the result of the `JSON.parse` call, which can be any type, including primitives. The declared return type `object` is therefore wrong.[15]

```
2  function getJSON(key){
3    var result = {};
4    var cookies = document.cookie.split('; ');
5    for (var i = 0; i < cookies.length; i++) {
6      var parts = cookies[i].split('=');
7      var name = parts[0];
8      var cookie = JSON.parse(parts[1]);
9      if (key === name) {
10       return cookie;
11   } }
12   return result;
13 }
```

[14] https://www.npmjs.com/

[15] This error has since been fixed, see https://github.com/DefinitelyTyped/DefinitelyTyped/pull/28529.

Notice how difficult it would be to detect this error using other techniques: the client must set a cookie whose value results in a non-object value when passed through `JSON.parse`, and then call `getJSON` with the name of that cookie.

Example 5. One of the real errors found by REAGENT in the `component-emitter` library involves the `Emitter` function that is declared as returning an object of type `Emitter`:

```
1  declare function Emitter(obj: any) : Emitter;
```

In the implementation (shown below) if an object is passed as argument, the call to `mixin` will copy all the properties from `Emitter.prototype` to the object, resulting in the object satisfying the required type (lines 5–10). However, if, for example, the argument is the value `true`, then that value is returned, and its type is not `Emitter` but `boolean`.

```
2  function Emitter(obj) {
3      if (obj) return mixin(obj);
4  }
5  function mixin(obj) {
6      for (var key in Emitter.prototype) {
7          obj[key] = Emitter.prototype[key];
8      }
9      return obj;
10 }
```

We fix the error by changing the parameter type `any` to `object`, after which REAGENT no longer reports any error for the `Emitter` function.

Repeating the RQ1 experiment using the configuration where all three optional assumptions are disabled reveals no additional true positives, which indicates that the configuration used above is not overly restrictive. However, additional false positives appear in three of the libraries when all the optional assumptions are disabled. The impact of the individual optional assumptions is studied for RQ2 below.

In summary, our answer to RQ1 is affirmative. REAGENT is able to find real errors, and without an overwhelming amount of false positives. Unlike all other tools that have been developed to detect mismatches between JavaScript libraries and TypeScript declaration files, the use of the RMGC allows REAGENT to ensure that under the chosen set of assumptions, no additional type violations exist in these libraries.

B. RQ2: What are the effects of the optional assumptions?

We evaluate REAGENT on the 10 JavaScript libraries using 5 different configurations: one with all optional assumptions enabled, three with a single assumption disabled, and one with all the assumptions disabled. (Each optional assumption could in principle be enabled or disabled for individual functions, however, for simplicity we either enable or disable each assumption for all the library functions together.) For each library and analysis configuration, we measure the action coverage (Section V-C) and the number of type violations reported. The results are shown in Table II. We write *timeout* if the analysis has not terminated within one hour. Type violation reports may have the same root cause; the numbers shown here are without any attempt at deduplication.

When all assumptions are enabled we get 100% action coverage on all but two libraries. For `component-emitter` the lacking action coverage is caused by an error in the declaration file (demonstrated in Example 5). This error causes TAJS to have a catastrophic loss of precision, however, once the error is fixed TAJS runs successfully and REAGENT reaches 100% action coverage. The lacking action coverage in `mime` is caused by a type violation that causes most of the library to be unreachable for the abstract RMGC. After fixing the error, we obtain 100% action coverage also for this library.

Disabling assumptions causes REAGENT to report more violations for some libraries, however, manually inspecting the reports shows that they are all false positives. Note that disabling assumptions makes the RMGC become "more general", which may increase the ability to detect type violations, but it also increases the risk of timeouts and suboptimal action coverage and thereby fewer violations being reported.

Disabling WIDTH-SUBTYPING causes massive losses of precision in six of the libraries, resulting in either a timeout or a loss of action coverage. The precision loss typically comes from TAJS reading an undeclared property on a client-constructed value, causing an avalanche of spurious dataflow.

Disabling NO-PROTOTYPE-STRINGS only changes the results for two libraries. We see extra false positives for `loglevel` and `pathjs`.

Disabling PREFER-LIB-VALUES makes no significant difference for the 10 benchmarks, however, we know that disabling this assumption can cause false positives in other libraries as shown in Example 1.

Disabling all three assumptions causes a catastrophic loss of precision for most libraries, and REAGENT only terminates successfully on three of the libraries.

We can from these results conclude that the WIDTH-SUBTYPING assumption is critical for precision for most libraries, NO-PROTOTYPE-STRINGS improves precision in some cases, PREFER-LIB-VALUES makes no difference for these benchmarks, and no additional true positives are found when disabling the assumptions. This suggests that enabling all the assumptions seems to be a reasonable default configuration.

Threats to validity The following circumstances may affect our conclusions. Although we have selected the 10 libraries randomly, they may not be representative. TAJS is not fully sound, which may cause REAGENT to miss errors (see Section VII). We have not conducted a user study to evaluate whether the error reports generated by REAGENT are also actionable to others, and our fixes to the erroneous declaration files have not (yet) been confirmed by the library developers.

VII. RELATED WORK

Open-world analysis Many static analyses require whole programs to work, and developing useful modular analysis techniques has been a challenge for decades [10]. The idea of using most-general clients (also called most-general applications) when statically analyzing the possible behaviors of libraries appears often in the static analysis literature. One

Table II: Comparison of RMGC variants. Each column contains *action coverage* / *violations*.

Library	all assumptions enabled	NO-WIDTH-SUBTYPING disabled	NO-PROTOTYPE-STRINGS disabled	PREFER-LIB-VALUES disabled	all three disabled
classnames	100.0% / 0	*timeout* / 0	100.0% / 0	100.0% / 0	*timeout* / 0
component-emitter	96.0% / 72	52.0% / 21	96.0% / 72	96.0% / 72	*timeout* / 73
js-cookie	100.0% / 4	*timeout* / 0	100.0% / 4	100.0% / 4	*timeout* / 0
loglevel	100.0% / 3	100.0% / 3	100.0% / 15	100.0% / 3	100.0% / 15
mime	8.3% / 1	8.3% / 1	8.3% / 1	8.3% / 1	8.3% / 1
pathjs	100.0% / 7	*timeout* / 30	100.0% / 12	100.0% / 7	*timeout* / 36
platform	100.0% / 10	100.0% / 10	100.0% / 10	100.0% / 10	100.0% / 10
pleasejs	100.0% / 16	*timeout* / 0	100.0% / 16	100.0% / 16	*timeout* / 0
pluralize	100.0% / 0	100.0% / 0	100.0% / 0	100.0% / 0	100.0% / 0
uuid	100.0% / 0	61.9% / 0	100.0% / 0	100.0% / 0	61.9% / 0
Average	90.4% / 11.3	– / 6.5	90.4% / 13.0	90.4% / 11.3	– / 13.5

example is the modular static analysis by Rinetzky et al. [25] for reasoning about heap structures; another is the points-to analysis for Java libraries by Allen et al. [3]. To the best of our knowledge, none of the existing techniques work for dynamic languages like JavaScript.

The tool Averroes [2] is able to analyze an application without using the implementation of the library. This is done by creating a placeholder library that is similar to a most-general client but with the roles swapped.

FLOW [9] is similarly to TypeScript a typed extension of JavaScript. FLOW is based on static analysis, like REAGENT, but obtains modularity by relying on type annotations, not only at the library interface but also inside the library code. The static analysis in FLOW has been developed as a compromise between soundness and completeness. As an example, FLOW does not detect any type error in the following simple program, where the `foo` function possibly returns a string at run-time instead of a number as expected from the type declaration.

```
1 var obj = { f: "this is a string, not a number" }
2 function foo(obj: typeof obj, s: string) : number {
3   return obj[s];
4 }
```

FLOW programs can use JavaScript libraries via type declaration files, much like in TypeScript.

Sound whole-program analysis for JavaScript A lot of research has been done on how to perform sound static analysis for JavaScript, and significant progress has been made in recent years on making such analysis scale to real-world JavaScript programs. Among the recent work are TAJS [4], WALA [18], SAFE [22], and JSAI [14]. All these analyzers share that they are "soundy" [19], meaning that they are sound in most realistic cases but rely on assumptions in specific corner cases that cause the analysis to be unsound in general. For example, the TAJS tool we use for REAGENT does not fully model all standard library functions,[16] but we believe those limitations are insignificant for the experimental evaluation of REAGENT.

Detecting errors in TypeScript declaration files Multiple approaches have been developed for detecting errors in TypeScript declaration files. TPD [28] finds errors by adding type contracts to existing library unit tests. These type contracts then check that the values observed during execution match the declared types. The approach in TPD is thus a variant of gradual typing [27], which has also been applied to general TypeScript code [23].

Like TPD, TSTEST [17] uses concrete executions with run-time type checks to detect errors in TypeScript declaration files. However, TSTEST explores the library using automated testing instead of relying on existing unit tests. TSTEST is similar to REAGENT in that it attempts to find type violations in a TypeScript declaration file by simulating a client, but it lacks the exhaustiveness that characterizes REAGENT.[17]

TSCHECK [12] is the only previous work that uses static analysis to find errors in type declaration files. Being based on a fast unsound static analysis and only detecting errors that manifest as likely mismatches at function return types, it provides no guarantees that all errors are found, unlike REAGENT.

VIII. CONCLUSION

We have shown how the concept of a reasonably-most-general client (RMGC) enables static analysis of JavaScript libraries to detect mismatches between the library code and the TypeScript declaration files. An RMGC works under a set of assumptions that reflect how realistic clients may behave. Imposing too few or too many assumptions can result in false positives or false negatives, respectively. We have proposed five specific assumptions, some of which are necessary to obtain any meaningful results, and others can be configured by the analysis user.

Experiments with our proof-of-concept implementation REAGENT that builds on the existing static analyzer TAJS demonstrates that the approach works, at least for small libraries that are within reach of TAJS. REAGENT finds real type mismatches without an overwhelming amount of false positives. By design, it explores the library code exhaustively, relative to the RMGC assumptions, unlike all existing alternatives.

In addition to improving the quality of TypeScript declaration files, we believe this work may also guide further development of TAJS and related JavaScript static analyzers.

Acknowledgments We are grateful to Gianluca Mezzetti for his contributions to the early phases of this research. This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647544).

[16] See https://github.com/cs-au-dk/TAJS/issues/8.

[17] Running TSTEST on the same benchmarks confirms that it finds a strict subset of the errors detected by REAGENT.

References

[1] DefinitelyTyped, February 2019. http://definitelytyped.org.

[2] Karim Ali and Ondrej Lhoták. Averroes: Whole-program analysis without the whole program. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference*, volume 7920 of *Lecture Notes in Computer Science*, pages 378–400. Springer, 2013.

[3] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. Combining type-analysis with points-to analysis for analyzing Java library source-code. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2015*, pages 13–18. ACM, 2015.

[4] Esben Andreasen and Anders Møller. Determinacy in static analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA*, 2014.

[5] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017*, pages 31–36. ACM, 2017.

[6] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference*, pages 257–281, 2014.

[7] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pages 152–161. ACM, 1986.

[8] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI)*, pages 296–310. ACM, 1990.

[9] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and precise type checking for JavaScript. *PACMPL*, 1(OOPSLA):48:1–48:30, 2017.

[10] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Compiler Construction, 11th International Conference, CC 2002*, volume 2304 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2002.

[11] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 242–256. ACM, 1994.

[12] Asger Feldthaus and Anders Møller. Checking correctness of TypeScript interfaces for JavaScript libraries. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014*, pages 1–16. ACM, 2014.

[13] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Static Analysis, 16th International Symposium, SAS 2009*, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2009.

[14] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: a static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22)*, pages 121–132. ACM, 2014.

[15] Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. Type refinement for static analysis of JavaScript. In *DLS'13, Proceedings of the 9th Symposium on Dynamic Languages*, pages 17–26. ACM, 2013.

[16] Erik Krogh Kristensen and Anders Møller. Inference and evolution of TypeScript declaration files. In *Fundamental Approaches to Software Engineering - 20th International Conference, FASE*, volume 10202 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2017.

[17] Erik Krogh Kristensen and Anders Møller. Type test scripts for TypeScript testing. *PACMPL*, 1(OOPSLA):90:1–90:25, 2017.

[18] Sungho Lee, Julian Dolby, and Sukyoung Ryu. HybriDroid: static analysis framework for Android hybrid applications. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, pages 250–261. ACM, 2016.

[19] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, 2015.

[20] Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13*, pages 499–509. ACM, 2013.

[21] Changhee Park, Hongki Lee, and Sukyoung Ryu. Static analysis of JavaScript libraries in a scalable and precise way using loop sensitivity. *Softw., Pract. Exper.*, 48(4):911–944, 2018.

[22] Joonyoung Park, Inho Lim, and Sukyoung Ryu. Battles with false positives in static analysis of JavaScript web applications in the wild. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016 - Companion Volume*, pages 61–70. ACM, 2016.

[23] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 167–180. ACM, 2015.

[24] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010*, pages 1–12. ACM, 2010.

[25] Noam Rinetzky, Arnd Poetzsch-Heffter, Ganesan Ramalingam, Mooly Sagiv, and Eran Yahav. Modular shape analysis for dynamically encapsulated programs. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007*, volume 4421 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2007.

[26] Olin Shivers. Control-flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)*, pages 164–174. ACM, 1988.

[27] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[28] Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. Mixed messages: Measuring conformance and non-interference in TypeScript. In *31st European Conference on Object-Oriented Programming, ECOOP 2017*, volume 74 of *LIPIcs*, pages 28:1–28:29, 2017.

[29] Yu Yuning. A study of object creators in JavaScript. Master's thesis, University of Waterloo, 2017.