Type inference and type checking for JavaScript strict mode

Micha Reiser
University of Applied Sciences Rapperswil
Supervised by Luc Bläser
FS2016 (May 10, 2016)

ECMAScript is the programing language of the web, nevertheless an upcoming trend is to use ECMAScript for native mobile and desktop applications or server backends, business critical applications. The tool support for catching errors early in the development lifecycle however is spare, due static analysis is non trivial for the dynamic ECMAScript language. The existing tools are either limited to find simple bug patterns, are based on a super or subset of ECMAScript or do not support the current ECMAScript version. This work describes an algorithm for type inference and type checking ECMAScript 6 code. The defined type system is unsound but achieves good results for a majority of programs. The described algorithm is implemented in ESChecker and is to the competitive tools Flow, TAJS and infernu. The results are summarized and further techniques for improving type inference are explained.

1 Introduction

The role of JavaScript dramatically changed over the last years. From an unpopular language used to add dynamic effects to webpages to a widely used language with a strong and growing community. It emerged from a browser only language to a general programing language used to write web-, desktop- and mobile- applications but also server components. This shift is reflected in an increasing complexity and number of JavaScript projects. To tackle the higher complexity, a better tooling support is needed for effective development and refactoring.

JavaScript does not provide any static analysis for proving the soundness of a program. Type and nullability checking is performed at runtime. Performing refactorings or adding new functionality is therefore a risky task, as the programer has a very limited tooling for testing if changes have been applied correctly. By applying the type checking statically instead of at runtime, common errors like accessing not declared variables, missing arguments, arguments in incorrect order or invoking a non function type can be detected and reported to the developer, without the need to execute the program.

JavaScript is an implicitly typed language and therefore requires type inference for type checking. Type inference for JavaScript is a non trivial task because of the very dynamic nature of JavaScript [1]. An explanation of the most important aspects follows.

Dynamic Objects A JavaScript object is a mapping from a string key to a value. Adding new properties or removing existing once can be performed dynamically. The names of properties can be computed values. Such operations often require reflection in statically typed languages.

Side effects JavaScript has support for higher order functions, functions accepting other functions as parameter. In comparison to functional languages, functions do not need to be pure and therefore can have side effects to their declaration scope or passed parameters.

Closure Functions have access to variables from their enclosing scope. Invoking a function requires that the function is evaluated in its declaration scope.

This binding The object referenced by `this` depends on the kind of function and its usage. Arrow functions capture the `this` of the enclosing context. The `this` of function declarations or expressions depends on the usage. The `this` can explicitly be specified if the function is invoked using `call` or `apply`. Otherwise the binding of `this` is implicitly defined. If the function is called as a method of an object, then `this` is equal to the object, to which the method belongs. If the function is not a member of an object, then `this` is `undefined`.

Environment The preliminary target of JavaScript applications is still the Browser. But it can also be executed standalone using NodeJS, Rhino or another JavaScript engine. These engines expose different native objects and methods at runtime, e.g. the browser exposes the DOM-Model. The type checker needs to model these object and methods specially as their JavaScript-Code is not existent.

An alternative and quite popular approach for type checking programs executing in a JavaScript environment is by transpiling a source language into JavaScript. The source language either allows type inference or contains type annotations that make type checking possible. Well known examples are TypeScript [2] and Flow [3]. The downside of a transpiled language is an additional required build step that slows down the development cycle and is a potential source for additional errors. A developer using a transpiled language needs to have a good understanding of the source language and JavaScript. This limits the number of potential recruiting candidates or requires additional training.

Its assumed that most of the code rarely uses reflection like code and therefore precise type inference is possible. Reflective code is most commonly used in libraries or frameworks. Type inference for frameworks and libraries can be substituted by using external declaration files to make type checking possible. The goal of this project thesis is to show a type inference and type checking algorithm for JavaScript that is precise and sound for the most common JavaScript code, but might be unsound or imprecise for non trivial code segments. The analysis is optimized for ECMAScript 6 code written in strict mode. Features prohibited in strict mode are not supported by the analysis.

2 Related Work

The work relates with linters, transpiled languages and other projects with the goal for a type checker for JavaScript.

Linters like ESLint [4] are used to enforce a specific coding style across a project or to find errors using common bug patterns. Linters use only trivial analysis to find bugs, mostly scoped to a single function. This work focuses on errors deducible by type checking, based on a sophisticated inter procedural analysis.

This work relates to transpiled languages that support type checking. The source language is either a language with complete type inference[5], [6] or requires type annotations. Flow [3] and TypeScript [2] are an example where a superset of JavaScript with support for type annotations is used as source language. This work differs from transpilers as the focus is on type checking JavaScript and not a super set.

TAJS [1] is a sound type analyzer and type checker for JavaScript. TAJS uses, context and path sensitive, abstract interpretation to infer the types for every term. The goal of TAJS is a precise and sound type checker that supports the full ECMAScript language. Compared to TAJS, this works is focused only on JavaScript programs using strict mode. Additionally the analysis does not need to be sound.

Infernu [7] implements type inference and type checking. Infernu defines a strict type system that only allows a subset of JavaScript. It uses the Damas-Hindley-Milner Algorithm for type inference and type checking. This works differs from infernu as its not limited to a strict subset of JavaScript.

Tern [8] is an editor independent JavaScript analyzer with type inference. Editors can use the Tern api to query type information, provide auto completion and jump to definition functionality. Tern uses

abstract values and abstract interpretation for type inference. Tern can only infer types for functions with an invocation. Not invoked functions are not analyzed. The work differs from Tern as the project focuses on type checking and not on providing an api for editors.

3 Benefits of analyzing strict mode

Strict mode has been introduced in ECMAScript 5. Strict mode allows to opt in to a restricted variant of JavaScript. Strict mode is not only a subset of JavaScript, it intentionally changes the semantics from normal code. It eliminates silent errors by throwing exceptions instead and prohibits some error prone or difficult to optimize syntaxes and semantics from earlier ECMAScript versions.

Strict mode can be explicitly enabled by adding the `"use strict"` directive before any other statement in a file or function. Using the directive in a file enables strict mode for the whole file, using it in a function enables strict mode for a specific function. Strict mode is enforced for scripts using ES6 modules [9, p. 10.2.1]. Therefore it can be expected that ES6 code is using strict mode. The analysis only supports code written in strict mode to take advantages of the changed semantics. A description of the changed semantics follows

Prohibited with statement The with statement is prohibited in strict mode [9, Annex C]. The with statement allows to access properties of an object in a block without the need to use member expressions. In the following example, the identifier `x` on line four can either reference the property `obj.x` or the variable `x` defined on line one.

```
1  var x = 17;
2  with (obj)
3  {
4    x;
5  }
```

If the object `obj` has a property `x`, then the identifier references the property, otherwise it references the variable `x`. This behavior makes lexical scoping a non trivial task [1]. The removal of the with statement allows static scoping.

Assignment to not declared variables Assignment to not declared variables introduce a global variable in non strict mode. Strict mode prohibits assignment to not declared variables and throws an error instead. Variables can only be defined with an explicit declaration. Access to yet unknown variables is always an error.

4 Algorithm

The classical approach for type inference is the Hindley Milner algorithm [10]. This work combines the Hindley Milner algorithm W with abstract interpretation. This allows inferring the most specific type for every variable at every position in the program. The classical Hindley Milner algorithm identifies the principal type for every variable in the whole program. The principal type is not sufficient for JavaScript. Compared to ML-Like languages, the type of a variable can change in Javascript. Either the variable is initialized after its declaration or a value with a different type is assigned to a variable. Hence, every position in the program requires its own type environment. This is achieved by using a data flow based analysis.

The control graph used for the data flow analysis is statement based. For each statement in the program one control flow graph node is created. Each edge represents a potential control flow between two statements. The control flow graph is not created on expression level to reduce the number of nodes and therefore the number of states required during the data flow analysis.

The analysis uses the work list algorithm [11] to traverse the control flow nodes in forward order. The analysis is not path sensitive. The order of the nodes is the same as the order of the statements in the program. The transfer function infers the type for the statement and its expressions using the Hindley Milner algorithm W. The output of the transfer function is the modified type environment containing the types for the inferred variables. Two branches are joined by merging the type environments. The merged type environment is an union of the both type environments. Conflicting mappings are merged by using the unification function of the Hindley Milner algorithm that returns the most general type.

A function declared in JavaScript has access to its declaration context. These functions are also

known as closures. Therefore the data flow analysis needs to be context sensitive. Context sensitivity is achieved by inlining the invoked function. Calling a function requires that the type environment of the caller is merged with the type environment from the callee declaration. The merged type environment contains the mappings from both type environments. The type of the callers context overrides the mapping of the declaration type environment for conflicting mappings.

5 Evaluation

As part of this work, the tool ESChecker has been implemented. It applies the described algorithm. The set of implemented JavaScript features is not sufficient to perform an evaluation using real live projects. Instead, sample listenings are used to compare the implementation with Flow, TAJS and infernu. Infernu and TAJS do not support ES5 and therefore ES5 equivalents of the listings are used.

5.1 Variable redefinement

In JavaScript, values of different types can be assigned to the same variable. Therefore the same variable can have different types at different positions in the program. The following listing defers the initialization of the variable.

```
let x;
x = { name: "Micha" };

x.name;
```

Infernu is the only tool that rejects the program. Infernu defines a strict type system that does not allow assignments to variables with incompatible types. In this example, an object value is assigned to a variable of type **`undefined`**.

5.2 Closures

JavaScript supports closures, allowing functions to read and modify variables from the outer scope. The analysis needs to be context sensitive to support closures.

```
let currentId = 0;

function uniqueId() {
  return ++currentId;
}

uniqueId();
```

Closures are supported by all evaluated tools.

5.3 Side effects

Functions can have side effects to the enclosing scope or parameters. In the following example, a new property is assigned to the passed in object. These side effects needs to be reflected in the callers context.

```
function setAddress(p, street, zip) {
  p.address = { street, zip } ;
}

const person = {
  lastName: "Reiser"
};
setAddress(person, "Bahnhofstrasse 12", 8001);
const street = person.address.street;
```

Infernu rejects the program as types are sealed after first passing them as argument to a function. Flow rejects the program with for the same reason. The intention of flow is to detect spelling errors and therefore only allows access to already defined properties.

The program is accepted by TAJS and ESChecker.

5.4 Function overloading

JavaScript does not provide built in support for function overloading. A function can only have one definition. But the number of arguments passed for a function call do not need to exactly match the number of parameters in the function declaration. This allows optional arguments. The following example provides a `range` function that can be called with one or up to three arguments[1].

```
function range(start, end, step) {
  if (end === undefined) {
    end = start;
    start = 0;
  }

  if (step === undefined) {
    step = 1;
  }

  const result = [];

  for (let i= start; i < end; i = i + step) {
    result.push(i);
```

[1]The implementation cannot use the default parameters of ES6 as the semantic of the passed arguments depends on the number of arguments. If the function is called with a single argument, then the `start` and `end` value need to be switched.

```
15      }
16      return result;
17    }
18
19    const r = range(10);
20    const r2 = range(1, 10);
21    const r3 = range(10, 1, -1);
```

Infernu rejects the program, as it cannot handle optional arguments. A function needs always to be invoked with exactly the same number of arguments as parameters defined in the function declaration. The other tools type check the program correctly.

5.5 Callbacks

Callbacks are commonly used in JavaScript for asynchronous and functional programing. Functions can be passed like values in function calls. This requires tracking of the passed function and evaluation with the passed arguments when its called. The following example implements the map function and applies it to a number array to double its values. The map function calls a callback for each array element and puts the result in the resulting array. The example also contains a second application of the map function with a string array to test if rank-1 polymorphism is supported.

```
1   function map(array, mapper) {
2     const result = [];
3     for (let i = 0; i < array.length; ++i) {
4       result.push(mapper(array[i]));
5     }
6     return result;
7   }
8
9   const array = [1, 2, 3, 4, 5, 6];
10  const doubled = map(array, x => x * 2);
11
12  const names = ["Anisa", "Ardelia", "Madlyn"];
13  const lengths = map(names, name => name.length);
```

This example is supported by all tools except Flow. Flow seems not to support type inference for rank-1 polymorph functions. The example type checks if either type annotations are added to the map function or if `map` is only applied once or with arrays of equal types.

5.6 Built in types

ECMAScript defines built in objects and functions. These functions are natively implemented and not available as JavaScript source code, hence the source code cannot be analyzed by the type checker or type inference algorithm. This requires that native object and functions are defined in the type checker or that it provides an external declaration of these. The following example uses the native `map` and `reduce` function. The arguments passed to `reduce` are intentionally in the wrong order to test if the type checker validates callbacks passed to built in functions. The first argument of `reduce` is the callback, the second the accumulator initialization value[2].

```
1   const array = [1, 2, 3, 4, 5];
2
3   const doubled = array.map(x => x * 2);
4   const sum = array.reduce(0,
5     (accum, x) => accum + x);
```

Infernu rejects the example as it does not support the built in `reduce` method. TAJS rejects the example as it does not support the built in `map` method. Flow and ESChecker correctly type the application of the `map` function and reject the application of the `reduce` function.

5.7 DOM Events

JavaScript is often used in the browser to add dynamics to a web page. This requires interaction with the DOM. A type checker needs to model the DOM and its type, as needed for other built in types like arrays. Modeling the DOM requires to define the built in types and methods, but also to model the browser events [12]. Reacting to user input is done by registering a listener for a particular event on a specific DOM-Element, like an input field. The api for registering listeners is generic, but the event passed to the listeners depends on the event type. A `keydown` event passes a `KeyboardEvent` object to the listener. The `KeyboardEvent` has additional properties allowing the identification of the pressed key. The type of the event is defined by the name of the event passed in by the listener registration as shown in the following example.

```
1   function onKeyDown(event) {
2     if (event.getModifierState("Shift")) {
3       console.log("Shift...");
4     }
5   }
6
7   const input = document.getElementById("pwd");
8
9   if (input) {
```

[2] As infernu does not support optional arguments, all callback arguments have been added to the callbacks of map and reduce before analyzing the example with infernu.

```
10    input.addEventListener("keydown", onKeyDown,
11      false);
12    input.addEventListener("click", onKeyDown,
13      false);
14  }
```

The given example is correct for the keydown event, as the `KeyboardEvent` defines a method `getModifierState`. The second registration should fail, as the click event defines no method `getModifierState`.

None of the inspected tools correctly identifies this error. Infernu has no support for the DOM at all and therefore rejects the program because `document` is undefined. ESChecker reports an error and TAJS a warning that the variable `search` potentially can be **`null`** and therefore the event handler cannot be registered on line 10. ESChecker is neither path sensitive nor does it evaluate the tests in if expressions. Therefore it assumes that `search` might be **`null`**. The same seems to be true for TAJS, but its unknown if it does not evaluate the if condition or lacks path sensitivity.

Flow is the only tool that does not reject the given example. The external definition of the DOM api implies that flow has a special overload for keyboard and non keyboard events and therefore should be capable to correctly type the given example[3]. But flow does not reject the second registration, even if the event misses the `getModifierState` method. Its currently unknown why the example is not rejected. Flow achieves the overloading by defining an enum with the keyboard event names.

A special handling for DOM events is beneficial, as it adds additional type safety. But events are not only used when interacting with the DOM, but also if frameworks are being used. E.g. Backbone makes strong use of events, for the internal implementation but also in its api. Frameworks like Angular are completely abstracting DOM events and binds the event handlers dynamically. The binding of the callback methods on the controller are defined in the HTML view belonging to this controller. This case is very hard to handle... TODO redefine paragraph.

[3]Flow supports external definition files only containing declarations. One of this declaration files describes the DOM `https://github.com/facebook/flow/blob/master/lib/dom.js#L99`

5.8 Summarized

6 Conclusion

1. Type inference is possible
2. Can be improved by analysing JSDoc or using external definitions
3. Focus on es 6 was not possible
4. Separate refinement from unification
5. Hard to analyze code that depends on frameworks or events (DOM)
6. Branch / Path sensitive analysis

References

[1] S. H. Jensen, A. Møller, and P. Thiemann, "Type analysis for JavaScript," in *Proc. 16th International Static Analysis Symposium (SAS)*, ser. LNCS, vol. 5673, Springer-Verlag, Aug. 2009. [Online]. Available: `http://cs.au.dk/~amoeller/papers/tajs/paper.pdf` (visited on 04/26/2016).

[2] Microsoft, *Typescript*, 2012. [Online]. Available: `https://www.typescriptlang.org/` (visited on 04/26/2016).

[3] Facebook, *Flow, a new static type checker for javascript*, Nov. 2014. [Online]. Available: `https://code.facebook.com/posts/1505962329687926/flow-a-new-static-type-checker-for-javascript/` (visited on 04/26/2016).

[4] jQuery Foundation, *Eslint*, 2016. [Online]. Available: `http://eslint.org/` (visited on 04/26/2016).

[5] A. Ekblad, "Towards a declarative web," Master's thesis, University of Gothenburg, 2012. [Online]. Available: `http://haste-lang.org/pubs/hastereport.pdf` (visited on 04/26/2016).

[6] B. McKenna, *Roy*. [Online]. Available: `http://roy.brianmckenna.org/` (visited on 04/26/2016).

[7] N. Lewis, *Inernu*. [Online]. Available: `https://noamlewis.wordpress.com/2015/01/20/introducing-sjs-a-type-inferer-and-`

checker-for-javascript/ (visited on 04/26/2016).

[8] M. Haverbeke, *Ternjs*. [Online]. Available: http://ternjs.net/ (visited on 04/26/2016).

[9] E. International, *Ecmascript® 2015 language specification*, Ecma International, Jun. 2015. [Online]. Available: http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf (visited on 04/26/2016).

[10] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, pp. 348–375, 1978.

[11] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999, ISBN: 3540654100.

[12] S. H. Jensen, M. Madsen, and A. Møller, "Modeling the HTML DOM and browser API in static analysis of JavaScript web applications," in *Proc. 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Sep. 2011. [Online]. Available: http://cs.au.dk/~amoeller/papers/dom/paper.pdf (visited on 04/26/2016).