Type inference and type checking for ECMAScript 6

Micha Reiser
University of Applied Sciences Rapperswil
Supervised by Luc Bläser
FS2016 (May 9, 2016)

Abstract

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 and also server components. These new applications have an effect on the size of code written in Javascript and the team members involved in JavaScript projects. Programing in larger teams and projects requires better tooling that supports 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. To verify the correctness of a program, every line of code needs to be executed. This can either be achieved by a coverage of 100% or by performing manual tests. Either of these solutions require an enormous effort, hence the test coverage in general is below 100% and manual tests are limited to some functionalities. Performing refactorings or adding new functionality is therefore a risky task, as the programer has a very limited assurance if the changes are correct. Specially refactorings are an error prone task in JavaScript, as IDE's cannot provide the same tooling support as for statically typed languages.

Many refactoring errors or wrong assumptions about a signature of function can be detected by using type checking. Type checking can provide an immediate feedback to type errors without an additional effort from the programmer. Very common errors are accessing not declared variables, missingt arguments, arguments in the wrong order or invoking non function types. These kind of errors can be detected by using type checking.

As JavaScript is an implicitly typed language, type inference is needed to perform type checking. The very dynamic nature of JavaScript makes type inference a non trivial task for multiple reasons [1].

Dynamic Objects A JavaScript object is a mapping from a string key to a value. New properties can be dynamically added or removed from an object. The property names can be dynamically computed, leading to reflection like code.

Dynamic this The object referenced by `this` depends on the invocation of the function (except for arrow functions and functions bound with `bind`). It can explicitly be defined if `call` or `apply` is used. It is implicitly defined by the object of which the function is a member. If a method of an object is aliased to a variable and invoked, then the `this` object points to `undefined`. The invocation of an aliased function is therefor not the same as the invocation of the original method.

Callee resolution Type inference requires that the called function is known. Invoking a function

has different affects on the passed arguments and to the variables in the enclosed closure. A invocation can add or remove properties of an argument, assign new values to parameter properties or variables from the enclosed closure. At the other hand, the function can access properties of its parameters and therefore requires parameters with a specific schema.
The difficulty is to resolve the callee of a function. The called function can be a member of an object, therefore the callee depends on the object, or the function is a higher order function bound to a variable.

Closure The context of a declared function needs to be associated with the function. TODO

Environment JavaScript can either be executed in a browser or on the Server using NodeJS, Rhino or another JavaScript engine. These environments expose different native methods at runtime. These native methods cannot be analyzed, as they are part of the runtime. Hence, this methods need to be defined externally.

Different projects solved the type inference problem by transpiling a static typed source language into JavaScript. 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.

The goal of this project thesis is to show that type inference for the majority of written JavaScript code is possible without the need of type annotations. Its assumed, that reflection like code is mostly found in frameworks and not very wide spread. Additionally, new language features of ES6 like classes reduce the need for self made inheritance utility functions and therefore of reflection like code.

2 Related Work

The work can be related with linters like ESLint [4]. Linters enforce a coding style and analyze the program for errors by using common bug patterns. The bug patterns are mostly limited to intra procedural errors. In comparison to linters, this work focuses on inter procedural errors, more specific, errors deducible by type checking.

This work relates to transpiled languages that support type checking. These languages are either a superset of JavaScript [2], [3] and use type annotations to solve the type inference problem or are based on a source language that supports type checking[5], [6]. In comparison to transpiled languages, the work focuses on type checking for JavaScript.

Tern [7] is an editor independent JavaScript analyzer. It performs type inference. An 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.

TAJS [1] is a type analyzer for JavaScript. It uses a context and path sensitive algorithm that is based on abstract interpretation. TAJS infers the types for every term and type checks the program. The goal of TAJS is a precise type analyzer that supports the full ECMASCript language. The work differs from TAJS that the focus is limited to JavaScript programs using strict mode and supports ES6.

Infernu [8] 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. The work differs from Infernu as Infernu infers the principal type for a term. This work describes an algorithm that infers the most specific type for every position in the program.

3 Benefits of an ES6 based analysis

The ECMAScript 6 (ES6) standard has been published on June 17, 2015 and therefore is also known as ES2015. Current browsers and Node 6 support in average 90% [9] of the new standard and older environments can be supported by using

ES6 to ES5 transpilers and polyfills. Hence applications written in pure ES6 are growing. The analysis focuses on code written in ES6 and therefore can benefit from the following simplifications.

3.1 let and const

The declarations of variables declared with `var` are moved to the top of the enclosing function or file. This is known as hoisting. Their scope is not the enclosing block scope, instead the enclosing function or the global scope). A `var` declared variable can be accessed before its declaration.

ES6 introduces `let` and `const` declaration. `Let` and `const` variables are bound to there enclosing block scope and hoisting does not apply. This simplifies the analysis as no pre processing is needed to find all declared variables in a function or the global scope.

Furthermore, `const` declared variables are readonly (but not immutable) and the value is always assigned in the declaration. Therefore the type of a `const` defined variable never changes.

The analysis treads `var` and `let` declarations equally. Meaning that `var` declarations are also block scoped. Access to a `var` declared variable before its declaration is therefore an error.

3.2 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. If it is the first statement in a file, then strict mode is enabled for the whole file. If it is the first statement in a function, then strict mode is only enabled for this function and all called functions. Strict mode is enforced for scripts using ES6 modules [10, p. 10.2.1]. Therefore it can be expected that ES6 code is using strict mode. For this reason, the analysis only supports ES6 code. An explanation of the changed semantics in strict mode that has a beneficial effect on static analysis follows.

Prohibited with statement The with statement is prohibited in strict mode [10, Annex C]. The with statement allows to access properties of a specific object in a block without the need to use member expressions.

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

E.g. the identifier `x` in the with block can reference the property `obj.x`, if the object has a property `x` or the variable `x` from the outer scope. The with statement makes lexical scoping a non trivial task [1]. The removal of the with statement from the language 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. Therefore variables can only be defined with an explicit declaration.

4 Algorithm

The classical approach for type inference is the Hindley Milner algorithm [11]. 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 [12] algorithm 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 using the Hindley Milner algorithm W. Therefore all types for the statement and its expressions are inferred. The output of the transfer function is the modified type environment that contains 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

The set of implemented JavaScript features is not sufficient to perform an evaluation using real live projects. Instead, sample listenings are used for comparing the implementation with the the Flow and Infernu project. Type checking for infernu requires that the examples are translated to ES5, as support for ES6 is missing.

accessing a possible null value

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

x.name;
```

Closures

```
let currentId = 0;

function uniqueId(prefix="") {
  const id = ++currentId;
  return `${prefix}${id}`;
}

uniqueId();
```

Side effects in functions

```
function setName(p, name) {
  p.name = name;
}

const person = { lastName: "Reiser" };
setName(person, "Micha");
person.name;
```

Optional arguments, built in functions, callbacks

```
function map(array, mapper) {
  const result = [];
  for (let i = 0; i < array.length; ++i) {
    result.push(mapper(array[i], i, array));
  }
  return result;
}

const array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const doubled = array.mapper(array, function (x) {
  return x * 2;
});
```

Built Ins

```
const array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const doubled = array.map(function (x) {
  return x * 2;
});
```

Method overloading

```
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);
  }
  return result;
}

range(10);
range(1, 10);
range(10, 1, -1);
```

Event Listeners

```
const name = "Something Good";
const element = document.getElementById("name");

function onClick(event) {
  console.log(name);
};

element.addEventListener("click", onClick, false);
```

6 Conclusion

1. Type inference is possible
2. Focus on es 6 was not possible
3. Separate refinement from unification
4. Hard to analyze code that depends on frameworks or events (DOM)
5. Branch / Path sensitive analysis
6. assignment expression sensitive?

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

[8] 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).

[9] Kangax, *Ecmascript 6 compatibility table*, May 2016. [Online]. Available: https://kangax.github.io/compat-table/es6/ (visited on 05/02/2016).

[10] 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).

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

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