Type Inference and Type Checking for JavaScript Strict Mode

Micha Reiser

University of Applied Sciences Rapperswil
Supervised by Luc Bläser
FS2016 (May 17, 2016)

The popularity of JavaScript dramatically increased over the last years. It evolved from the programming language of the web to a general purpose language that is also used for mobile-, desktop- and server-applications. However, the tooling support for program verification is spare due to the dynamic nature of JavaScript that is hard to be covered using static analysis. The existing verification tools are either limited to simple bug patterns, are based on a super or subset of JavaScript, or only support outdated JavaScript versions. This work describes an algorithm for type inference and type checking of JavaScript code written in strict mode. The defined algorithm combines the Hindley Milner algorithm W with abstract interpretation. The algorithm is unsound as the precision of the type inference degrades for very dynamic code, code that is mainly located in frameworks or libraries. The defined algorithm is implemented in *ESChecker* and is compared to competitive type checkers. The results show that a precise type inference can be achieved for a majority of programs and provides a valuable feedback for programmers.

1 Introduction

The role of JavaScript dramatically changed over the last years. From an unpopular language used to add dynamic effects to web pages to a widely used language with a strong and growing community. It emerged from a browser only language to a general purpose language used to write web-, desktop-, mobile-, and server-applications. 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 only 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. Static type checking allows to detect common errors like accessing not declared variables, missing arguments, arguments in incorrect order or invoking a non function type, without the need to execute the program. It therefore is a valuable tool for providing a fast statement about the soundness of a 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]. JavaScript has several features that makes static program analysis difficult. An explanation of the language features adding complexity to static analysis 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. A property name can either be a static or computed value. The later is used for dynamic object creation or modification, similar to code using reflection in statically typed languages.

Closures Functions have access to variables from their enclosing scope. Invoking a function requires that the function is evaluated in its declaration scope. Therefore the analysis needs to be context sensitive.

Side effects Functions in JavaScript are not pure and therefore invoking a function can have side effects to arguments passed to the function or variables from the outer scope of the function declaration. These side effects can also affect the type of the involved variables, e.g. if the function assigns a value of a different type to an outer scope variable or adds a new property to a parameter. A precise analysis needs to reflect these side effects in the callers context.

This Binding The object referenced by `this` depends on the function kind and its usage. Arrow functions capture the `this` of the enclosing context. The `this` inside of a 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 execution environment of JavaScript applications is still the browser. Nevertheless, it is also used for standalone applications or to add scripting functionality to other applications. In this case NodeJS, Rhino or another JavaScript engine is used as execution environment. Each environment exposes different native objects and methods at runtime, e.g. the browser exposes the DOM-API. A type checker needs to model these object and methods specially as their JavaScript-Code is not existent.

Implementing a sound type checker for JavaScript risks to be over restrictive and only allows a very limited subset of JavaScript programs or reports a large amount of false positives. An unsound type system has the disadvantage that some errors are not detectable but allows a far more complete set of JavaScript programs for type checking and therefore has a better chance to be adopted in the wild.

This work defines a sound algorithm that is capable to infer types and perform type checking for a majority of the written code. The precision degrades for very dynamic code, e.g. code that uses dynamic object creation or manipulation. For these cases the algorithm is unsound as the inferred types might be imprecise. Its expected that these features are mainly used in frameworks are libraries, therefore can be well abstracted. The work suggest to use other techniques to work around these cases and substitute type inference for these edge cases by using external type annotations, making the type checking sound again. The analysis defined is based on JavaScript code written in strict mode. Features prohibited in strict mode are not supported by the analysis.

The first section compares this work with related work in the area of type checking JavaScript code or code executed in JavaScript environments. Section 3 explains the benefits of restricting the analysis to strict mode and why it is believed that the code written in strict mode will be growing in the near future. The basics of the algorithm are explained in section 4, the results from the evaluation are shown in the preceding section and is followed by the conclusion.

2 Related Work

The work relates with linters, transpiled languages and other type checkers for JavaScript.

Linters like ESLint [2] are used to enforce a specific coding style across a project or to find errors using common bug patterns. Linters use simple static analysis techniques for bug identification. These analysis are mostly intra procedural analysis. This work focuses on errors deducible by type checking, based on a sophisticated inter procedural analysis.

An alternative and quite popular approach for type checking programs executing in a JavaScript environment is by transpiling a source language to JavaScript. The source language either allows type inference [3], [4] or contains enough type annotations so that type checking is possible. Well known examples are TypeScript [5] and Flow [6]. The downside of a transpiled language is the need for an additional build step that slows down the development cycle and is a potential source for 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. This work differs from transpilers as the focus is on type checking JavaScript and a transpiled source language.

TAJS [1] is a sound type analyzer and type checker for JavaScript. The used algorithm is context and path sensitive. TAJS uses abstract interpretation for type inference. The goal of TAJS is a precise and sound type checker that supports the full JavaScript language. Compared to TAJS, this works is focused only on JavaScript programs using strict mode. Furthermore, the defined analysis is unsound to reduce the number of false positives.

Infernu [7] implements type inference and type checking for JavaScript. 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 it is not limited to a self defined subset of JavaScript.

Tern [8] is an editor independent JavaScript analyzer with type inference. Editors can use the API of Tern 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 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 ECMAScript 6 modules [9, p. 10.2.1]. Therefore it can be expected that newer code written 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 with an effect to the analysis 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    x; // references obj.x or variable x
4  }
```

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 An Assignment to a not declared variable introduce a global variable in non strict mode. Strict mode prohibits assignments to not declared variables and throws an error instead. In strict mode variables can not be implicitly declared. Therefore accessing a yet unknown variable is always an error.

4 Algorithm

The classical approach for type inference is the Hindley Milner algorithm [10]. The Hindley Milner algorithm infers the principal type for every variable in a program. This is sufficient for languages where the type of a variable does not change over its lifespan but is not for JavaScript. A common pattern in JavaScript is to first declare the variables and defer their initialization. A pattern that has its roots in variable hoisting. Besides deferred initialization, it is also completely legal JavaScript code if values of different types are assigned to the same variable. Therefore, the principal type is insufficient for JavaScript and requires that the types of variables are kept distinct between different positions in the program. This is achieved by using a data flow based analysis. The described algorithm combines the Hindley Milner algorithm W with abstract interpretation.

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 required in 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 in and out state of the data flow analysis is the type environment. If a node has multiple in branches, then the type environments of these branches are joined by unification. The union of the two type environments contains the mappings of both environments, conflicting mappings are merged using the $unify$ function of the Hindley Milner algorithm.

$$\Gamma_1 \cup \Gamma_2 = \{(x, \tau) | x \in \Gamma_1 \vee x \in \Gamma_2\}$$
$$\tau = \begin{cases} unify(\Gamma_1(x), \Gamma_2(x)) & x \in \Gamma_1 \wedge x \in \Gamma_2 \\ \Gamma_1(x) & x \in \Gamma_1 \wedge x \notin \Gamma_2 \\ \Gamma_2(x) & x \in \Gamma_2 \wedge x \notin \Gamma_1 \end{cases}$$

The algorithm uses inlining to handle function calls. If a function is called, then the function body is evaluated in the callers context making the algorithm context sensitive. Using the type environment of the caller is insufficient for the analysis of the function body as the called function might access variables from its declaration scope. Therefore, the mappings from the function declaration type environment are added to the callers type environment. The sum of the two type environment contains the mappings from both type environments. If a mapping exists in both type environments, then the type of the callers type environment is used. The mapping of the callers type environment needs to be used, as the type of a variable might have changed since the function has been declared. The equation defining how two type environments are added follows.

$$\Gamma_1 + \Gamma_2 = \{(x, \tau) | x \in \Gamma_1 \vee x \in \Gamma_2\}$$
$$\tau = \begin{cases} \Gamma_2(x) & x \notin \Gamma_1 \\ \Gamma_1(x) & \text{otherwise} \end{cases}$$

5 Evaluation

As part of this work, the tool ESChecker [12] has been implemented. It applies the described algorithm. The set of implemented JavaScript features is not sufficient to perform an evaluation using real life projects. The implementation covers the basic operations but has no support for classes, modules or prototyping. The support for object is limited to object expressions and subtyping of object uses structural typing (duck typing). The evaluation uses sample listenings to compare the implementation with Flow, TAJS, and infernu[1]. 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.

```
1  let x;
2  x = { name: "Micha" };
3
4  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.

```
1  let currentId = 0;
2
```

[1]The evaluation uses the following versions: Flow 0.24.1, TAJS 0.9-7, infernu 0.0.0.1.

```
function uniqueId() {
  return ++currentId;
}

uniqueId();
```

Closures are supported by all evaluated tools.

5.3 Side Effects

A function call can have side effects to the enclosing scope of the function declarations or the passed arguments. In the following example, a new property `address` is assigned to the passed in object. The analysis needs to reflect these side effects 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 the types of function call arguments are sealed. Flow rejects the program for the same reason. Flow does this intentionally to detect spelling errors. Therefore, only already defined properties can be accessed of function arguments.

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 to a function has not to exactly match the number of parameters in the function declaration. This allows optional arguments and therefore function overloading. The following example provides a `range` function that can be called with one or up to three arguments[2].

```
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;
}

const r = range(10);
const r2 = range(1, 10);
const r3 = range(10, 1, -1);
```

Infernu rejects the program, because it does not support optional arguments. A function must 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 as values. This requires that the flow of function values is tracked. The following example implements the `map` function and applies it to a an array of numbers with a function that doubles the values. The `map` function calls a callback for every array element and puts the result in the resulting array. The example also contains a second application of the `map` function with an array of strings to test if the type inference algorithm supports rank-1 polymorphism [13].

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

const array = [1, 2, 3, 4, 5, 6];
const doubled = map(array, x => x * 2);

const names = ["Anisa", "Ardelia", "Madlyn"];
const lengths = map(names, name => name.length);
```

Flow is the only tool that rejects this example. Flow does not to support type inference for rank-1 polymorphic functions. The example type checks if either the map function is attributed with type annotations or `map` is only applied once or with arrays of equal types.

5.6 Built in Types

JavaScript 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 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` functions[3]. The order of the arguments passed to the `reduce` function is intentionally incorrect to verify if the type checker validates callbacks passed to built in functions. The first argument of `reduce` is the callback, the second the initialization value of the accumulator.

```
const array = [1, 2, 3, 4, 5];

const doubled = array.map(x => x * 2);
const sum = array.reduce(0, (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.

[2] 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 this argument specifies the `end` and not the `start`.

[3] 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.

5.7 Dynamic Object Manipulation

Object creation or changing the object structure can be performed dynamically by using reflection like code. Such code uses computed property names, something that requires reflection in statically typed languages. The following example is an implementation of the common `defaults` function. The `defaults` function accepts two objects. The method adds the properties of the `source` object that are missing in the `target` object to the `target` object. This pattern is commonly used to initialize absent properties with default values for option-objects.

```
1  function defaults(target, source) {
2    target = target === undefined ? {} : target;
3    for (const key of Object.keys(source)) {
4      target[key] = target[key] ===
5        undefined ? source[key] : target[key];
6    }
7
8    return target;
9  }
10
11 let options = defaults({}, {
12   rounds: 1000,
13   precision: 1
14 });
15 for (let i = 1; i < options.rnds; ++i) {
16   // ...
17 }
```

A precise and sound analysis needs to unroll the for loop on line three to know which properties are copied from the `source` to the `target` object on line four. Unrolling requires that the analysis understands the semantics of `Object.keys`.

The example initializes a plain object with default values on line eleven. The initialized option object is then used on line 15, but the property name `rnds` for `rounds` is misspelled. TAJS is the only implementation that detects the misspelled property name. Flow and ESChecker are unsound for dynamic code and therefore do not report any errors. Infernu rejects the program as `for in` loops are not supported.

5.8 DOM Events

JavaScript is often used in web applications to add dynamic effects. Interactions can be added using the DOM-API. A type checker needs to model the DOM-API, as needed for other built in types like arrays. Besides the defined DOM types and methods, a type checker needs also to model the DOM events [14]. 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 listener 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 event name passed during the listener registration as shown in the following example on line 10 and 12.

```
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) {
10   input.addEventListener("keydown", onKeyDown, false);
11   input.addEventListener("blur", onKeyDown, false);
12 }
```

The registration on line 10 for the `keydown` is correct, as the `KeyboardEvent` defines the `getModifierState` used in the listener on line 2. The registration for the `blur` event is erroneous, as the `blur` event defines no method `getModifierState`.

TAJS is the only tool that detects the malicious invocation of the `getModifierState` method in case a `blur` event is triggered as potential error. But TAJS also reports a warning that the variable `input` can potentially be null online 10 and 11, regardless of the preceding null check on line 9.. The same is true for ESChecker, but it reports an error instead of a warning and therefore rejects the whole program. The problem of ESChecker lies in the fact that it is neither path sensitive not does it evaluate the test condition. Infernu fails to type check the given example as it does not model the DOM API. Flow does not reject the given program but neither detects the invocation of the not defined function `getModifierState` for the `blur` event.

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. Various frameworks like Backbone or Angular use events internally and as part of their API. This requires that their event model is integrated into the type checker too to achieve type safety. Because a type checker cannot Therefore, an external definition of the events is desired.

5.9 The Impact of Frameworks on Type Inference

Frameworks are a special challenge for type inference as they make heavy use of dynamic invocations. Frameworks are implemented according to the inversion of control principle, the framework invokes the application, not visa versa. This inverses the — for type inference important — control and data flow.

Inferring the correct type is often only possible if the type of the arguments are known. Without an actual invocation, these are unknown. The following example shows an Angular [15] controller implementation. The implementation loads the persons using the `$http` service. By mistake, the `get` method has been misspelled.

```
1  class PersonController {
2    persons;$http;
3
4    constructor($http) {
5      this.$http = $http;
6      this.persons = [];
7    }
8
9    loadPersons() {
10     this.$http.gt("/persons")
11       .then(response => this.persons = response.data);
12   }
13 }
```

The problem shown by this example is that type inference for code using a framework with dynamic invocation is impossible for a framework agnostic type checker as the control and

data flow cannot be inferred using static analysis. The `$http` service accepted in the constructor on line 3 is a service that is injected by angular at runtime. The implementation to inject is resolved by name, therefore no explicit connection between the service usage and the service implementation exists. But the type of `$http` must be known to perform type checking. As the type for `$http` is not known, type checking without type annotations is not possible for the given example.

6 Conclusion

The tool support for JavaScript is especially small compared to its popularity. Developers need to relay on manual testing or unit tests for revealing programming errors. The implemented, unsound type checker provides a tool that is capable to infer the types and catch a variety of errors through type checking. The evaluation showed that the analysis is precise for most of the scenarios and sometime provides better results then competitive tools. Therefore, the tool can provide valuable feedback. But the evaluation also shows that the precision abruptly decreases when objects are dynamically manipulated.

The use of dynamic object manipulation only affects type inference and not the type checking algorithm. A considerable alternative to adding support for these edge cases — that might add a high complexity to the type inference algorithm — is the used of type annotations to improve the overall result. Type annotations can either be extracted from JSDoc or be defined in external declaration files, similar to the TypeScript definition files. This increases the precision of the type inference algorithm and therefore the achievable results with type checking. A tool implementing this approach should emit a warning if the type inference algorithm cannot infer the types to motivate the programmer to annotate these terms. Allowing any form of type annotations also allows type checking of functions invoked by frameworks that otherwise cannot be analyzed.

The evaluation also showed that the used algorithm has its limitations. As the analysis is not path sensitive, possible null values cannot be accessed as their value is potentially always null in all branches. Another issue is that — without path sensitivity — type specific branches cannot be implemented, a technique often used to emulate function overloading.

Further the set of supported features is not sufficient to analyze real word projects. The implementation is still missing elementary features like classes, prototyping or modules, these are all essential features needed before the tool is useful. Supporting the features defined in ES6 and in the upcoming ES7 standard requires a tremendous amount of additional work that exceeds the scope of a project thesis by a multitude. But this project thesis showed that precise and good type inference results can be achieved for a majority of JavaScript that, combined with type checking, provides an immediate feedback to the developer. The cases where effective and precise type inference is not achievable can be substituted by adding type annotations. The analysis might not be sound, but the gained support is worth it.

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

[3] 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).

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

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

[6] 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).

[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] M. Reiser, *Eschecker*, online, 2016. [Online]. Available: `https://github.com/DatenMetzgerX/ESChecker`.

[13] B. C. Pierce, *Types and Programming Languages*, 1st. The MIT Press, 2002, ISBN: 0262162091, 9780262162098.

[14] 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).

[15] *Angularjs — superheroic javascript mvw framework*.
(visited on 05/17/2016).