Type Inference and Type Checking for JavaScript Strict Mode

Micha Reiser

University of Applied Sciences Rapperswil
Supervised by Prof. Dr. Luc Bläser
ST 2016

Abstract

In recent years, the popularity of JavaScript drastically increased and became a general-purpose language. However, the tool support for program verification is scarce due to the dynamic nature of JavaScript, that is hard to be covered using static analysis. 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 introduces an algorithm for type inference and type checking of JavaScript code written in strict mode. This algorithm combines the Hindley-Milner Algorithm W with abstract interpretation. The type system used is unsound, as the precision of type inference diminishes for reflection-like code that is mainly found in frameworks or libraries. The defined algorithm is implemented in *ESChecker* and is compared to competing type checkers. The evaluation results show that the presented type inference algorithm is precise for a majority of programs. It provides a valuable feedback to programmers if combined with type checking.

1 Introduction

The role of JavaScript drastically changed in recent 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 to prove 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 programmer has a very limited tooling for testing if changes have been applied correctly. Static type checking allows to detect common errors like accessing non-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 a dynamically typed language and therefore requires type inference for type checking. Type inference for JavaScript is a non-trivial task because of its very dynamic nature [11]. JavaScript has several features that makes static program analysis difficult. An explanation of the language features adding complexity to static analysis follows.

Dynamic Object-Structure A JavaScript object is a mapping from a string key to a value. Adding new properties or removing existing ones can be performed dynamically. A property name can either be a static or computed value. The latter 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 caller's 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 value referenced by *this* inside of a function declaration or expression 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 bound to the object to which the method belongs. If the function is not a member of an object, then *this* is bound to the global object.

Host-Environment The preliminary host-environment of JavaScript applications is still the browser. However, it is also used for standalone applications or to add scripting functionality to other applications. In this case, NodeJS [7], Rhino [6] or another JavaScript engine is used as host-environment. Each environment exposes different native objects and methods at runtime, e.g. the browser exposes the DOM-API. A type checker needs to specifically model these objects and methods 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 it does not detect all errors, but allows type checking of a far more complete set of JavaScript programs and therefore has a better chance to be applied in actual practice.

This work defines a sound algorithm that is capable of inferring the types and perform type checking for a majority of the written code. The precision diminishes for reflection-like code, e.g. code that uses dynamic object creation or manipulation. The algorithm is unsound for these cases as the inferred types might be imprecise. It is expected that these features are mainly used in frameworks or libraries. This work suggests to substitute type inference for these edge cases by using type annotations. The presented analysis is limited to JavaScript code written in strict mode. Features prohibited in strict mode are not supported by the analysis.

The first section compares this work with existing tools used to verify or analyze JavaScript programs. 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

This work is related to linters, transpiled languages and other type checkers for JavaScript.

Linters — like ESLint [5] — 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 analyses mostly are intra-procedural. 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 transcompiling a source language to JavaScript. The source language either allows type inference [2, 14] or uses type annotations. Well-known examples are TypeScript [15] and Flow [4]. 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 amount of potential programmers or requires additional training. This work differs from transpilers as the focus is on type checking JavaScript and not transpiled languages.

TAJS [11] is a sound type analyzer and type checker for JavaScript. The used algorithm is context- and path-sensitive. It uses abstract interpretation for type inference. The goal of TAJS is a precise and sound type checker that supports the full JavaScript language. If the precision of type inference diminishes, the severity of errors is reduced to warnings. The current version only supports ECMAScript 3 and a limited set of ECMAScript 5. Compared to TAJS, this work focuses on JavaScript programs using ECMAScript 6 and strict mode. Furthermore, this work is unsound to prevent false positives implied by imprecise type inference instead of emitting warnings for them.

Infernu [13] implements type inference and type checking for JavaScript. It uses the Hindley-Milner algorithm. The used type system only models a limited set of JavaScript. The subset is defined by the properties required by the Hindley-Milner algorithm. For instance, the unmodified Hindley-Milner algorithm requires that a variable has exactly one type in a program, the principal type. Therefore, Infernu disallows assigning values of different types to the same variable, contrary to the JavaScript specification. This work differs from Infernu as it extends the Hindley-Milner algorithm to support a wider set of JavaScript programs and to reduce the number of false positives.

Odgaard describes in his master thesis a dynamic analysis for type inference [18]. The presented analysis uses code instrumentation to obtain the runtime values and derives the variable types from these. The inferred types are used to add JSDoc [12] type annotations. Compared to the approach presented by the work of Odgaard, this work uses static over dynamic analysis for type inference.

Tern [8] is an editor-independent JavaScript analyzer with type inference. It provides an API for editors but does not perform type checking. 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 actual invocation. Non-invoked functions are not analyzed. This work differs from Tern, because 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. It 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. It also prohibits some error-prone and hard-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 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]. Inside the *with* statement object properties can be accessed without the need to use member expressions. In the following example, the identifier *x* on line three either references 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 *obj.x*, otherwise it references the variable *x*. This behavior makes lexical scoping a non-trivial task [11]. The prohibition of the *with* statement allows static scoping.

Assignment to non-declared Variables An assignment to a non-declared variable introduces a global variable in non-strict mode. Strict mode prohibits assignments to non-declared variables and throws an error instead. Variables can not be implicitly declared in strict mode. Therefore, accessing a not yet known variable is always an error.

4 Algorithm

The classical approach for type inference is the Hindley-Milner algorithm [16]. The Hindley-Milner algorithm infers the principal type for every variable in a program. This is sufficient for languages restricting that the type of a variable can not change over its lifespan. In contrary, JavaScript has no such restriction, allowing values of different types to be assigned to the same variable. For instance, a common JavaScript pattern is to declare the variables and defer their initialization. The type of the variable after its declaration is *void*. The initialization changes the type of the variable from *void* to the type of the assigned value. Therefore, a single type for a variable in the whole program is insufficient for JavaScript. This requires that the variable types are kept distinct between different positions in the program. This is achieved by using data-flow analysis. The described algorithm combines the Hindley-Milner Algorithm W with abstract interpretation.

4.1 Data-Flow Analysis

The control flow graph used for the data-flow analysis is statement-based. A control flow graph node is created for each statement in the program. Each edge represents a potential control-flow between two statements. A node in the control flow graph only represents a basic block if none of the statement's expressions introduce new control flows. For instance, a conditional expression creates two possible control flows — one if the condition is true and another if the condition is not — that are not represented in the control flow graph. The control flow graph is statement-based to reduce the number of states and therefore, the number of states required for the data-flow analysis.

Property	Value
Traversal Order	Forward
Node Order	Statement Order
Transfer Function	Hindley-Milner Algorithm W
In- / Out-State	Type Environment Γ
Join Operation	$\Gamma_1 \cup \Gamma_2$
Sensitivity	Flow and Context

Table 1. Properties of the Worklist Algorithm

The analysis uses the work list algorithm [17] 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 unified. The union of two type environments contains the mappings of both environments. Conflicting mappings are merged using the $unify$ function of the Hindley-Milner algorithm. The unification can be defined as follows.

$$
\Gamma_1 \cup \Gamma_2 = \{(x, \tau) | x \in \Gamma_1 \vee x \in \Gamma_2\}
$$

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

The properties of the worklist algorithm are summarized in the table 1.

4.2 Function Invocation

The algorithm uses inlining for function calls. If a function is called, then the function body is evaluated in the caller's 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 missing mappings from the function declaration type environment Γ_{decl} are added to the caller's type environment Γ_{caller}. This is denoted as $\Gamma_{caller}\left[\Gamma_{decl}\right]$. The resulting type environment contains the mappings from both type environments. Mappings already present in the caller's type environment are not overridden. This is important, because the type of a variable might have changed since the function declaration. This can be defined as follows.

$$\Gamma_{caller}\left[\Gamma_{decl}\right] = \{(x,\tau) | x \in \Gamma_{caller} \lor x \in \Gamma_{decl}\}$$

$$\tau = \begin{cases} \Gamma_{decl}(x) & x \notin \Gamma_{caller} \\ \Gamma_{caller}(x) & \text{otherwise} \end{cases}$$

4.3 Type System

The type system is designed to infer the types for arbitrary JavaScript code without the need for type annotations. The precision of the inferred type diminishes for reflection-like code fragments. The current type system infers the types of all terms and has no support for type annotations. The type system is designed to catch the following errors:

1. Reading of or assigning to an undeclared variable
2. Accessing a property of *null* or *undefined*
3. Invoking a non-function value
4. Invoking a function with missing or incompatible arguments
5. Applying an operator with illegal operands

The defined type system does not distinguish errors by their severity.

Maybe Type The type system distinguishes between absent values, values that may be present, and values that are present for certain. This is needed to detect access to properties on *undefined* or *null* values causing runtime exceptions. The values *null* and *void* are modeled as unit types and represent values that are absent for certain. Potentially absent values are represented by the type *Maybe<T>*. The type represents a value that is present in some paths but is not in others. It includes the values *void*, *null*, and all values defined by *T*. Accessing a property of a potentially absent value needs to be guarded by a null check.

Record Type Objects supporting members are represented as record types. A record type consists of a set of members. A member is defined by a unique label and the type. The object expression { *name: "Test", age: 18* } is represented as record type with two members. The labels of the members are *name* and *age*, the members have the type *string* and *number*.

The type system uses structural typing for record types. Therefore, a type τ_1 is a subtype of τ_2 if τ_1 has at least the same members as τ_2 and the type of each of these members is a subtype of the corresponding member in τ_2. The type system does not support nominal typing. Nominal typing is required to support classes and prototype-based type checking. The type system does not support classes nor prototypes, therefore nominal typing is not supported either.

Array Type The array type *T[]* describes an array containing elements of type *T*. The array type is a specialized record type. The elements contained in an array need to be homogenous. Heterogeneous arrays are not supported as the type system does not define a union type. A union type allows values of different types, for instance, the union "*string* or *number*" contains either *string* or *number* values.

The array type does not track the element type by the element's position in the array. Accessing element members of a heterogeneous array therefore requires a type check if the element is of the expected type. To increase the precision for small, static arrays, a tuple type can be defined. The tuple type tracks the type for every position in the array. Accessing a tuple element therefore does not require an explicit type check.

Function Type The function type describes a function or method. A function type is characterized by the type of its arguments and its return type, but also the type of the value referenced by *this*. The structure of *this* is defined by the accessed members on *this* inside the function body. The type for *this* is not implicitly defined by the object to which the method belongs.

The presented algorithm uses inlining for invocations of non-native functions. The exact type of a function is not inferred. The inferred type for a function declaration uses type variables for the type of *this*, the arguments, and the return value. Type checking of the function body is performed for every invocation by replacing the type variables with the actual types used in the invocation.

The function type is also used to describe native functions of the host-environment. Type checking an invocation of a native function requires testing if the actual *this* type and the types of the passed arguments are subtypes of the expected types. The *Maybe* type is used to define optional arguments.

The implementation does not yet support invoking a function using *call* nor *apply*. Binding the referenced value of *this* using *bind* or to the *this* of the outer scope by using arrow functions are not supported either.

Any Type The type system models the special type *any*. The type *any* is a super and subtype of all types. The type inference uses the *any* type as backdoor whenever the type can not be inferred. Accessing a property of an *any* value yields *any*. An *any* value can be invoked as function with arbitrary arguments, which returns *any*. Type checking is completely disabled for *any* values. Therefore, an inferred *any* value inherently leads to unsafe code.

Limitations The presented type system only supports a limited set of JavaScript features. It neither has support for classes nor prototype based objects. It does not support the module syntax introduced with ECMAScript 6 and therefore all analyzable programs are limited to a single file. Neither does the type system model all features precisely. The plus and compare operators are limited to numbers, even though *strings* can be concatenated using the plus operand and any object with a *valueOf* method can be compared.

5 Evaluation

As part of this work, the tool ESChecker [20] has been implemented, applying the described algorithm. The set of implemented JavaScript features is not sufficient to perform an evaluation using realistic projects in broader practice. The evaluation uses sample listings to compare the implementation with Flow, TAJS, and Infernu[1]. Infernu and TAJS do not support ECMAScript 6 and therefore, ECMAScript 5 equivalents of the listings are used.

The test cases and their results are summarized in table 2. Cells marked with a ✓ indicate that the test case (row) is handled correctly by the type checker (column). An empty cell indicates that the evaluation does not lead to the expected result. The test cases focus on features that are difficult to analyze statically — as mentioned in section 1 — or features commonly 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 initialization of the variable is deferred in the following listing.

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

x.name;
```

Infernu is the only tool that rejects the program — that is well defined at runtime. The type system used by Infernu is over-restrictive. It prohibits value assignments to variables if the value type is not a subtype of the variable type. In this example, an object value is assigned to a variable of the incompatible type *undefined*.

5.2 Closures

JavaScript supports closures, allowing functions to read and modify variables from the outer scope. The following listening defines the *createUniqueIdGenerator* function that returns a generator function. The generator uses the *currentId* variable as memory to remember the last returned value. This variable is only defined in the declaration scope of the generator but not in the caller's scope when the generator-function is invoked (line 8). Therefore, the analysis needs to be context-sensitive to support closures.

```
1 function createUniqueIdGenerator() {
2   let currentId = 0;
3 
4   return () => ++currentId;
5 }
6 
7 const generator = createUniqueIdGenerator();
8 generator();
```

Closures are supported by all evaluated tools.

5.3 Side Effects

Functions in JavaScript do not have to be pure. A function call can have side effects to passed arguments or variables from the enclosing scope of the called function. For type checking, only side effects affecting the type of a variable are relevant. The function called in the following example adds the new property *address* to the object passed as argument. This change of the arguments structure needs to be reflected in the caller's context.

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

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

Neither Flow nor Infernu allow side effects changing the structure of an argument and therefore reject the program. Adding new properties is intentionally prohibited by Flow to detect spelling errors.

This 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. However, the number of arguments passed to a function does not have to exactly match the number of declared parameters. This allows optional arguments and therefore function overloading. The following listing defines and applies the *range* function. The function 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 = ...;
  return result;
}

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

Infernu does not support optional arguments and therefore rejects the program. The other tools support optional arguments.

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. The *map* function invokes a callback for every array element and puts the result in a new array that is returned. The *map* function is a rank-1 polymorphic function, it can be applied with arrays of arbitrary types. Therefore, the type inference

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

[2] The implementation can not 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 argument defines the *end* of the range and not the *start*. At the other hand, if *range* is invoked with two arguments, then the first argument specifies the *start* and the second the *end*. Therefore, the semantic of an argument is not only defined by its position but also by the number of arguments.

Test Case	Infernu	TAJS	Flow	ESChecker
Variable Redefinement		✓	✓	✓
Closures	✓	✓	✓	✓
Side Effects		✓		✓
Function Overloading		✓	✓	✓
Callbacks	✓	✓		✓
Built-in Types			✓	✓
Dynamic Object Manipulation		✓		
DOM Events		✓		
Frameworks				

Table 2. Evaluation Results

algorithm needs to infer the principle type and not a specialization [19]. To verify that the principal type is inferred, the *map* function is applied once with an array of numbers and once with an array of strings.

```
function map(array, mapper) {
  const result = [];
  // ...
  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 does not support type inference for rank-1 polymorphic functions. The example is accepted by Flow if either type annotations are added to *map*, the function is only applied once, or the passed arrays are of equal types. All other tools support type checking of callbacks and type inference for rank-1 polymorphic functions.

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 can not be analyzed by the type inference algorithm. This requires that native object and functions are defined in the type checker or externally. The following example uses the native array functions *map* and *reduce*[3].

The result of the *reduce* function is an accumulated value over all array elements. The function uses a callback — passed as first argument — to accumulate the values in the array. The first argument of the callback is the accumulated value over all preceding array elements. The second argument of the callback is the current array element. The callback adds the current element to the accumulated value of the preceding elements and returns the accumulated value. The second argument of *reduce* is the initial accumulator value. The *reduce* function is invoked with the arguments in incorrect order to verify if the type checker validates callbacks passed to built-in functions.

```
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 program as it does not support the built-in *reduce* method. TAJS rejects the program as it does not support the built-in *map* method. Flow and ESChecker correctly type the application of the *map* function and state the incorrect application of the *reduce* function.

5.7 Dynamic Object Manipulation

The structure of an object can be dynamically defined or manipulated by using reflection-like code. Such code uses computed property names. The following listing shows an implementation of the widely used *defaults* function. The *defaults* function accepts two objects and initializes the properties with value *undefined* of the *target* object with the values defined in *source*. 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] === undefined ?
   ↪   source[key] : target[key];
5    }
6
7    return target;
8  }
9
10 let options = defaults({}, {rounds: 1000, step: 1});
11 const end = options.rnds * options.step;
```

A precise and sound analysis needs to unroll the *for* loop (line 3) to know which properties are copied from the *source* to the *target* object (line 4). Unrolling requires that the analysis understands the semantics of *Object.keys*.

This example initializes an empty object with default values (line 10). The initialized option-object is used on line 11, but the property name *rnds* — for *rounds* — is misspelled. TAJS is the only implementation that detects the misspelled property name, but only if a *for in* loop is used instead of the unsupported *Object.keys* method. But it also emits a warning that the property *step* is potentially absent, and thus a false

[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.

positive. Flow and ESChecker are unsound for such dynamic code and therefore do not detect the misspelled property name. Infernu does not support *for in* loops and therefore fails to type check the program.

5.8 DOM Events

JavaScript is often used in web applications to add dynamic effects by using the DOM-API. A type checker needs to model the DOM-API, as needed for other built-in types like arrays. Modeling the types and methods defined by the DOM-API is insufficient to achieve type safety, the type checker also needs to model the DOM events [10].

A JavaScript application reacts to user input 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 type of the handled event. 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 specified when the listener is registered as shown in the following listing on line 10 and 11.

```
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 }
```

Registering the *onKeyDown* listener for the *keydown* event is legitimate as the *KeyboardEvent* defines the *getModifierState* method used by the listener. On the other hand, registering the same listener for the *blur* event is erroneous as the *blur* event does not define the *getModifierState* method.

TAJS is the only tool that detects the malicious invocation of the *getModifierState* method in case of a *blur* event. But TAJS also reports a warning that the variable *input* can potentially be null on line 10 and 11, regardless of the preceding null check (line 9). ESChecker also reports an error that *input* is potentially absent. The reason for this is that ESChecker is neither path-sensitive nor does it evaluate the test condition. Therefore, *input* is potentially absent in all branches and accessing a property is wrongly detected as error for all branches.

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 it is noteworthy that events are not only used when interacting with the DOM. Events are also used by frameworks like Angular [1] or Ember [3] internally or as part of their API. A type checker therefore does not only need to model the DOM events but also the framework events to achieve type safety.

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 vice versa. This inverses the control- and data-flow, that are decisive for type inference.

Inferring the correct type is often only possible if the types of the arguments are known. Without an actual invocation, these are unknown. The Angular [1] controller implementation shown in this example loads a remote resource using the *$http* service (line 8) and stores the result in the *persons* array (line 9). The name of the *get* method is misspelled.

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

The problem shown by this example is that type inference for framework-based code — if the framework uses dynamic invocation — is hardly possible for a framework-agnostic type checker, because the control and data-flow can not be inferred using static analysis. The *$http* service accepted in the constructor (line 2) is a service that is injected by angular at runtime. The implementation to inject is resolved by name, therefore no explicit connection between the service usage and the service implementation exists. But the type of *$http* must be known to perform type checking. As the type for *$http* is not known, its usages can not be type checked. Explicit type annotations are needed for the *$http* parameter in the constructor.

5.10 Interpretation of Evaluation Results

The evaluation shows that none of the evaluated tools supports all test cases. ESChecker achieves preciser results than Flow and Infernu, but not as precise as TAJS. TAJS has the most precise type inference and therefore achieves the best type checking results. TAJS preliminary limitation is the lack of support for many ECMAScript 5 features. This might be the major impediment for its adaption for real world projects. It also tends to emit false positives if the precision of the type inference diminishes. Flow at contrary has the best support for ECMAScript 6 features but type inference is less precise. The precision can be increased by adding type annotations. The type system used by Infernu is over-restrictive and rejects most of the test cases, all of them are well typed at runtime. Compared to Infernu, ESChecker has a lower false positive rate and therefore combining the Hindley Milner Algorithm W with data-flow analysis has proven to be beneficial to increase the precision.

But the precision is not sufficient for all examples. Path-sensitivity is needed to access members of a potentially *null* object, otherwise the value is potentially null in all branches, independent of preceding null checks. Path-sensitivity is also needed for type-specific branches, commonly used for function overloading.

The evaluation also indicates that the precision for the type inference diminishes for reflection-like code, for instance, if the structure of an object is dynamically manipulated. A considerable alternative to type inference for these edge cases — supporting this edge cases might add a high complexity to the type inference algorithm — is the use 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 type inference and therefore leads to better type checking results. A tool implementing this approach should emit a warning if the type inference algorithm can not infer the types to encourage the programmer to add type annotations. Allowing any form of type annotations also has the benefit that functions invoked by frameworks can be annotated and therefore can be type checked.

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 type checker provides a tool that is capable of inferring the types and catch a variety of errors through type checking. The evaluation shows that the analysis is precise and sound for most of the scenarios and sometime provides better results than competitive tools. Therefore, the tool can provide valuable feedback.

But the evaluation also indicates that the presented algorithm has its limitations. First, the precision diminishes for reflection-like code. The precision for these cases can be improved if type annotations are used instead of type inference to improve the overall result. Another limitation is the inability to access properties of potential absent values as the analysis is not path-sensitive and therefore, the value is potentially absent in all branches. Path-sensitivity is also required to support type-specific branches, 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, such as classes, prototyping or modules. The tool has to support all these essential features to be useful. Supporting a majority of the features defined in the ECMAScript 6 and the upcoming ECMAScript 7 standard require a tremendous amount of additional work that exceeds the scope of a project thesis by a multitude. But this project thesis shows that precise inference results can be achieved for a majority of JavaScript that — combined with type checking — provides a valuable and immediate feedback to programmers.

References

[1] *AngularJS — Superheroic JavaScript MVW Framework*. URL: `https://angularjs.org/` (visited on 05/17/2016).

[2] Anton Ekblad. "Towards a Declarative Web". MA thesis. University of Gothenburg, 2012. URL: `http://haste-lang.org/pubs/hastereport.pdf` (visited on 04/26/2016).

[3] *Ember.js*. URL: `http://emberjs.com/` (visited on 05/17/2016).

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

[5] jQuery Foundation. *ESLint*. 2016. URL: `http://eslint.org/` (visited on 04/26/2016).

[6] Mozilla Foundation. *Rhino*. URL: `https://www.mozilla.org/rhino/` (visited on 05/24/2016).

[7] Node.js Foundation. *Node.js*. URL: `https://nodejs.org` (visited on 05/24/2016).

[8] Marijn Haverbeke. *Ternjs*. URL: `http://ternjs.net/` (visited on 04/26/2016).

[9] Ecma International. *ECMAScript® 2015 Language Specification*. Ecma International, June 2015. URL: `http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf` (visited on 04/26/2016).

[10] Simon Holm Jensen, Magnus Madsen, and Anders 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)*. Sept. 2011. URL: `http://cs.au.dk/~amoeller/papers/dom/paper.pdf` (visited on 04/26/2016).

[11] Simon Holm Jensen, Anders Møller, and Peter Thiemann. "Type Analysis for JavaScript". In: *Proc. 16th International Static Analysis Symposium (SAS)*. Vol. 5673. LNCS. Springer-Verlag, Aug. 2009. URL: `http://cs.au.dk/~amoeller/papers/tajs/paper.pdf` (visited on 04/26/2016).

[12] *JSDoc*. URL: `http://usejsdoc.org/` (visited on 05/22/2016).

[13] Noam Lewis. *Inernu*. URL: `https://noamlewis.wordpress.com/2015/01/20/introducing-sjs-a-type-inferer-and-checker-for-javascript/` (visited on 04/26/2016).

[14] Brian McKenna. *Roy*. URL: `http://roy.brianmckenna.org/` (visited on 04/26/2016).

[15] Microsoft. *TypeScript*. 2012. URL: `https://www.typescriptlang.org/` (visited on 04/26/2016).

[16] Robin Milner. "A theory of type polymorphism in programming". In: *Journal of Computer and System Sciences* 17 (1978), pp. 348–375.

[17] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999. ISBN: 3540654100.

[18] Morten Passow Odgaard. "JavaScript Type Inference Using Dynamic Analysis". MA thesis. Aarhus University, 2014. URL: `http://cs.au.dk/fileadmin/site_files/cs/Masters_and_diplomas/MortenPassowOdgaard.pdf` (visited on 04/26/2016).

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

[20] Micha Reiser. *ESChecker*. online. 2016. URL: `https://github.com/DatenMetzgerX/ESChecker`.