Generating runtime type checks for JavaScript from TypeScript

Fabian Pirklbauer

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2017

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, September 15, 2017

Fabian Pirklbauer

Contents

Abstract

Kurzfassung

Chapter 1

Introduction

1.1 Problem Definition

1.2 Solution Approach

1.3 Thesis Structure

Chapter 2

Technical Foundation

This chapter will give an overview of the technical knowledge required for this thesis. It will give an exposure to the different type systems, the programming language JavaScript and JavaScript supersets. Also the terminology used throughout this paper will be specified, since standard terminology differs across sources [1, p. 1].

2.1 Type Systems

There are different kinds of programming languages with different characteristics and specifications. An essential part of a language is its type system, which has a great impact on the behavior of a program and may influence the syntax the program is written in. In general a programming language can be categorized as typed or untyped, where untyped languages do not have a type system at all, or have a single type that can hold any value [1, p. 2]. More precisely a language is considered typed independently of types being part of the syntax, but simply by the existence of a (static) type system [1, p. 2]. According to Cardelli from *Microsoft Research*[1] a type system is

> a collection of type rules for a typed programming language [1, p. 1]

with the purpose to

> prevent the occurrence of execution errors during the running of a program [1, p. 1].

He further equates *type system* with *static type system* and also Pierce defines type systems as being static [6, p. 2], which categorizes languages as untyped, that may distinguish between types in run time, but do not have knowledge about types during compile time or interpretation, such as JavaScript (see section **??**). This notion is further supported by Louden and Lambert, stating that

> languages without static type systems are usually called untyped languages (or dynamically typed languages). Such languages include [...] most scripting languages such as Perl, Python, and Ruby [4, p. 331].

[1] https://www.microsoft.com/research/

2.1.1 Explicitly and Implicitly Typed

If types are part of the syntax of a language (e.g. Java) it is explicitly typed, whereas in implicitly typed languages, type annotations are assigned automatically by the type system [1, pp. 2, 3]. Some languages, however, make use of a mixture, allowing developers to omit type annotations in various scenarios, where the type can be inferred by the compiler [6, p. 10] as shown in the C# code below:

```
var implicitNum = 10; // implicitly typed as integer
int explicitNum = 10; // explicitly typed as integer
```

While a type system—explicit, implicit or a combination of both—may detect possible execution errors already during compile time, it is not required to guard against specific errors. There are mechanisms for untyped languages to make them safe [1, p. 3], as outlined in section 2.1.4.

2.1.2 Execution Errors

Errors can occur in various situations and in order to classify a language, it is important to understand the different types of errors. Cardelli distinguished between three error categories: Trapped errors, untrapped errors and forbidden errors [1, p. 3].

Trapped Errors

A trapped error causes a program to stop immediately [1, p. 3] or to raise an exception, which may be handled in the program [6, p. 7]. An example for such an error could be division by zero [1, p. 3].

Untrapped Errors

Errors where a program does not crash or raise an exception immediately are called untrapped errors [1, p. 37]. They may remain unnoticed—at least for a while—and can lead to unexpected behavior [1, p. 3]. For example accessing data from an array that is out of bounds is perfectly legal in the programming language C [6, p. 7], but can lead to errors or arbitrary behavior later in the program [1, p. 3].

Forbidden Errors

Following the definition of Cardelli, forbidden errors should include “all of the untrapped errors, plus a subset of the trapped errors [1, p. 3]”. They are not generally defined, but vary between programming languages and may even not include all untrapped errors, which leads to a language being considered as unsafe [1, p. 4].

2.1.3 Safety and Good Behavior

A programming language can be considered as safe if no untrapped errors can appear, and is well behaved (i.e., good behaved), if no forbidden errors can occur [1, p. 3], consequently good behavior implies safety. Not all major languages are safe, and therefore not well behaved, such as C or C++ [6, p. 6], as guaranteeing safety usually results in

increased execution time. An example for a safe language, with decreased development and maintenance time compared to an unsafe language, is Java [1, p. 5].

2.1.4 Type Checking

To ensure that a program follows the specified rules of its type system and to guarantee safety and good behavior (i.e., ensuring the absence of forbidden errors [1, p. 37]), as described in section 2.1.3, type checking may be performed. Again, Cardelli treats *type checking* and *static type checking* as equivalent and calls languages that employ such a technique *statically checked* [1, p. 3]. Dynamically checked programming languages, on the other hand, may also ensure good behavior by applying sufficient checks at run time. Anyway, statically checked languages may also perform checks during the execution of a program to guarantee safety, if not all untrapped errors can be discovered statically during compilation [1, p. 4]. Following the terminology of Cardelli, expressions like *statically typed* or *dynamically typed* are avoided in favor for *statically checked* and *dynamically checked*, respectively [1, p. 1].

2.2 JavaScript

JavaScript dates back to 1996, where its creator Brendan Eich from the company *Netscape*[2] submitted the language to Ecma International[3] [5, p. 28], an "industry association founded in 1961, dedicated to the standardization of information and communication systems [12]" and since became one of the most popular programming languages in the world [2, p. 2]. According to *GitHub*[4] it was the most popular language on its platform by opened *Pull Requests*[5] with a growth of 97% in 2016, followed by Java, which saw an increase of 63% compared to 2015. Also *TypeScript* (see 2.3.1) is following up, which takes the 15th place with an increase of Pull Request by 250% [14].

While JavaScript is known for programming inside browsers and adding visual effects to websites [7, p. 4], its first use in a product was on the server-side in 1994 [5, p. 369]. Since then, no application platform for JavaScript was available for 17 years until Ryan Dahl created and released *Node.js*[6] in 2011, which allowed developers to build cross-platform applications in JavaScript. It is built upon Google's *V8 JavaScript engine*, also used in the popular *Chrome*[7] browser [5, p. 369].

The following sections try to give an ample exposure to the language JavaScript, outlining its most important and interesting concepts. Please note, that not all cases—especially the numerous exceptions—are described, as this would go beyond the scope of this thesis.

[2] *Netscape Communications*—founded as *Mosaic* in 1994—released its *Netscape Communicator* browser in 1995 which became the leading internet browser at that time [11].

[3] https://www.ecma-international.org

[4] https://github.com

[5] Pull requests on GitHub are used to let other people know about changes made to a repository. From there on these modification can be reviewed and discussed with collaborators and can be rejected or merged into the repository [9].

[6] https://nodejs.org

[7] https://www.google.com/chrome/

2.2.1 Loose Typing

Like in other programming languages, variables can be declared and values may be assigned to them. An essential concept of JavaScript is its loose typing, meaning that any value may be assigned or reassigned at any time to any variable:

```
let foo = 10;
foo = "I've been a number, now I'm a string";
```

The term *loose typing* may be misleading to infer that JavaScript has a type system. However, when following standard terminology and keeping in mind that type system is equal to *static* type system, it is made clear, that JavaScript is considered untyped. It does employ mechanisms to reject code from running, that has semantic errors, but evaluation is performed during execution, and errors are determined and reported during run time [13, p. 291]. Therefore JavaScript can be deemed a *dynamically checked* language (see section 2.1.4).

2.2.2 Value Types

JavaScript is an untyped and dynamically checked—but safe—scripting language, as defined in section 2.1. Most untyped programming languages are necessarily safe, as it would be exceedingly difficult to maintain the code if untrapped errors would remain unnoticed [1, p. 4]. Even though considered as untyped, the EcmaScript language specification defines seven value types [13, p. 16]:

- Undefined
- Null
- Boolean
- String
- Symbol
- Number
- Object

A major difference to a (statically) typed language is, that in JavaScript only values are typed, variables are not. When requesting the type of a variable with the `typeof` operator during run time, the assigned value's type is determined and returned as a string [8, p. 30]:

```
let num = 10;
typeof num; // "number"
```

The returned string by `typeof` does not reflect the specified value types entirely. As shown in table 2.1, objects are differentiated by wether they are callable or not. For objects with a call signature `"function"` will be returned and `"object"` otherwise. Strangely, for a value of type *Null*, the result will be `"object"`. A proposal to change the specification and to fix this bug—erroneously indicating that null is an object—was rejected, as existing code may break [15, 16].

2.2.3 Type Conversion

In JavaScript "any [...] value can be converted to a boolean value [3, p. 40]". If the interpreter expects a boolean value, it simply performs a conversion [3, p. 46] (see

Table 2.1: Result of the `typeof` operator by a value's type [13, p. 164].

Type of Value	*Result*
Undefined	"undefined"
Null	"object"
Number	"number"
String	"string"
Symbol	"symbol"
Object (not callable)	"object"
Object (callable)	"function"

Table 2.2: Values evaluated as false or true when converted to a boolean value [3, p. 40].

Falsy	`undefined`, `null`, `0`, `-0`, `NaN` and `""` (empty string).
Truthy	Any other value, including `[]` (empty array) and `{}` (empty object).

section 2.2.4). Because of that it is important to know, which values are considered true, and which conversion will result in being false, depending on their type, as shown in table 2.2. There are various situation where a conversion is desired, which happens implicitly in JavaScript. For example if a string should be added to a number, and vice versa, the number is converted to a string and the result is a concatenation of both values:

```
"2" + 3 // "23"
"Hello" + 2 + 3 // "Hello23"
```

The outcome of such an operation will most likely complete without errors, as the interpreter does its best to come up with a sufficient result. Anyway, it has a major influence on the result, how such an expression is written. In the example above, the string was seen first by JavaScript, therefore the subsequent numbers were converted to a string. If, on the other hand, the numbers came first, the result would have been completely different:

```
2 + 3 + "Hello" // "5Hello"
```

Again, even a slight change to the code means an entirely different outcome:

```
"2" + 3 + "Hello" // "23Hello"
```

A more comprehensive overview of possible type conversions, summarized by Flanagan and extended with Symbol type conversions as of the latest ECMAScript 2015 specification, can be found in table 2.3, which also highlights situations, where type conversions are not possible or lead to an error.

Table 2.3: Type conversions in JavaScript [3, p. 46, 13, pp. 36-44].

Initial Value	*String*	*Number*	*Boolean*	*Object*
`undefined`	`"undefined"`	`NaN`	`false`	*TypeError*
`null`	`"null"`	`0`	`false`	*TypeError*
`true`	`"true"`	`1`		(i)
`false`	`"false"`	`0`		(i)
`""` (empty string)		`0`	`false`	(i)
`"1.2"` (non-empty, numeric)		`1.2`	`true`	(i)
`"one"` (non-empty, non-numeric)		`NaN`	`true`	(i)
`0`	`"0"`		`false`	(i)
`-0`	`"0"`		`false`	(i)
`NaN`	`"NaN"`		`false`	(i)
`Infinity`	`"Infinity"`		`true`	(i)
`-Infinity`	`"-Infinity"`		`true`	(i)
`1` (finite, non-zero)	`"1"`		`true`	(i)
`{}` (any object)	(ii)	(iii)	`true`	
`[]` (empty array)	`""`	`0`	`true`	
`[9]` (single numeric array)	`"9"`	`9`	`true`	
`["a"]` (any other array)	(iv)	`NaN`	`true`	
`() => {}` (any function)	(ii)	`NaN`	`true`	
`Symbol("sym")` (any symbol)	*TypeError*	*TypeError*	`true`	(i)

(i) For situations, where converting a value to an object does not throw a TypeError, a new object of the value's type is returned. E.g. for the value `"Hello world!"`, `new String("Hello world!")` is returned [13, p. 44].
(ii) When converting an object to a string, JavaScript tries to call the `toString` or `valueOf` method on the object and converts the returned value to a string. If no primitive value can be obtained from either of these methods, a TypeError is thrown [3, p. 50].
(iii) The same steps as in (ii) are performed with the difference, that `valueOf` will be preferred over `toString`.
(iv) The toString method of the Array object joins the array separated by a comma, which would result in `["a","b","c"]` being converted to `"a,b,c"` [10].

2.2.4 Value Comparison

In section 2.2.4, the flexibility of JavaScript has already been outlined. Types are converted (i.e., casted) to another type if required and possible. The same is true when comparing values. JavaScript tries to implicitly convert a value to another type if it cannot perform a comparison at first. Comparing a string—holding a numerical value—with an actual number, will give the same result as comparing two values of type Number, as the interpreter implicitly casts the string to a number:

```
"5" > 2 // true
"2" == 2 // true
```

When comparing with the *equality operator* `==` there are a few rules to keep in mind [3, p. 72]:

- The values `null` and `undefined` are considered equal.
- If a number and a string are compared, the string is converted to a number.
- the values `true` and `false` are converted to `1` and `0` respectively.
- Objects are compared by reference[8], whereas one value being a number and the other one being a string, JavaScript tries to convert it to a primitive value, either by using the object's toString or valueOf method.
- All other comparisons are not equal.

If a more strict comparison is required and an automatic conversion of values is not desired, the *strict equality operator* `===` may be used. Only if type and value matches, the expression evaluates to true:

```
"2" === 2 // false
2 === 2   // true
```

Following the rules above it is interesting to look at comparing an object to the string `[object Object]`:

```
{} == "[object Object]"  // true
{} === "[object Object]" // false
```

Using the equality operator, the object is converted using the default toString method, returning `[object Object]`, resulting in the compared values being equal. When making use of the *strict* equality operator, no conversion is performed and the expression is false.

2.2.5 Objects and Prototypal Inheritance

Every value that is not a primitive value (i.e., Undefined, Null, Boolean, Number, Symbol, or String [13, p. 5])—including functions and arrays—is an object, hence making JavaScript a highly flexible language.

2.2.6 Latest Improvements

Declaration Keywords

Arrow Functions

Classes

2.2.7 Field of Application

2.3 JavaScript Supersets

2.3.1 TypeScript

2.3.2 Flow

2.3.3 Other

[8] TODO: Explain by reference/by value

Chapter 3

State of the Art

3.1 Existing Projects

Chapter 4

Concept

4.1 Method Principle

4.2 Type Check Situations

4.2.1 Main Cases

4.2.2 Edge Cases

4.2.3 Exceptions

4.2.4 Procedure

Chapter 5

Implementation

5.1 Tools and Libraries

5.2 Structure

5.3 Components

5.3.1 Core

5.3.2 Mutators

5.3.3 Factory

5.3.4 Utilities

5.4 Usage

5.4.1 Application Programming Interface (API)

5.4.2 Command Line Interface (CLI)

Chapter 6

Evaluation

6.1 Automated Unit Tests

6.2 Performance Analyzation

6.3 Comparison

6.3.1 Javascript without Type Checks

6.3.2 Javascript with Manual Type Checks

6.3.3 Flow with Generated Type Checks

6.4 Interpretation and Conclusion

Chapter 7

Summary and Outlook

References

Literature

[1] Luca Cardelli. "Type Systems". In: *Computer Science Handbook*. Ed. by Allen B. Tucker. 2nd ed. CRC Press, 2004. Chap. 97, (cit. on pp. 2–5).

[2] Douglas Crockford. *JavaScript: The Good Parts. Unearthing the Excellence in JavaScript*. 1st ed. Sebastopol, California: O'Reilly Media, 2008 (cit. on p. 4).

[3] David Flanagan. *JavaScript: The Definitive Guide. Activate Your Web Pages*. 6th ed. O'Reilly Media, 2011 (cit. on pp. 5–7).

[4] Kenneth C. Louden and Kenneth A. Lambert. *Programming Languages: Principles and Practices*. 3rd ed. Boston, Massachusetts: Cengage Learning, 2011 (cit. on p. 2).

[5] Den Odell. *Pro JavaScript Development. Coding, Capabilities, and Tooling*. 1st ed. California: Apress, 2014 (cit. on p. 4).

[6] Benjamin C. Pierce. *Types and Programming Languages*. 1st ed. Cambridge, Massachusetts: The MIT Press, 2002 (cit. on pp. 2, 3).

[7] Martin Rinehart. *JavaScript Object Programming*. 1st ed. Apress, 2015 (cit. on p. 4).

[8] Kyle Simpson. *You Don't Know JS: Up & Going*. 1st ed. Sebastopol, California: O'Reilly Media, Apr. 2015 (cit. on p. 5).

Online sources

[9] *About pull requests*. URL: https://help.github.com/articles/about-pull-requests/ (visited on 04/21/2017) (cit. on p. 4).

[10] *Array.prototype.toString() - JavaScript*. 2017. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/toString (visited on 04/24/2017) (cit. on p. 7).

[11] Sean Cooper. *Whatever happened to Netscape?* Oct. 2014. URL: https://www.engadget.com/2014/05/10/history-of-netscape/ (visited on 04/21/2017) (cit. on p. 4).

[12] *Ecma International*. URL: https://www.ecma-international.org (visited on 04/21/2017) (cit. on p. 4).

[13] *ECMAScript 2015 Language Specification. ECMA-262 6th Edition.* June 2015. URL: http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf (visited on 04/23/2017) (cit. on pp. 5–8).

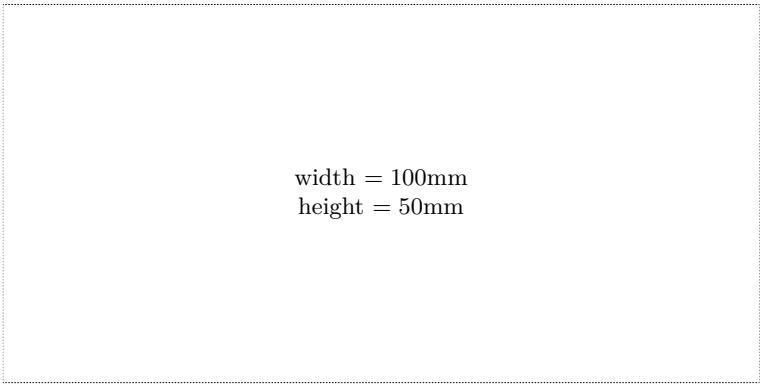
[14] *GitHub Language Stats.* 2016. URL: https://octoverse.github.com (visited on 04/21/2017) (cit. on p. 4).

[15] Chris Smith. *What to know before debating type systems.* Oct. 2013. URL: http://2ality.com/2013/10/typeof-null.html (visited on 04/24/2017) (cit. on p. 5).

[16] *typeof - JavaScript.* 2017. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof (visited on 04/24/2017) (cit. on p. 5).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —