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 static 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. 38]

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 [10, 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

> [l]anguages 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 [7, p. 331].

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

Obviously, a widely adopted consensus in terminology is to use both, *untyped* (e.g. in [17, p. 117]) and *dynamically typed* (e.g. in [5, p. 32] and [8, p. 203]) for languages without a static type system. Anyway, 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]. This should help to avoid confusion over languages having types, but are referred to as untyped.

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 [10, 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 [10, 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 [10, 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++ [10, 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].

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] [9, p. 28], an "industry association founded in 1961, dedicated to the standardization of information and communication systems [24]" 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 3.1) is following up, which takes the 15th place with an increase of Pull Request by 250% [32].

While JavaScript is known for programming inside browsers and adding visual effects to websites [11, p. 4], its first use in a product was on the server-side in 1994 [9, 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 [9, p. 369].

The following sections try to give an ample exposure to the language JavaScript,

[2] *Netscape Communications*—founded as *Mosaic* in 1994—released its *Netscape Communicator* browser in 1995 which became the leading internet browser at that time [23].

[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 [19].

[6] https://nodejs.org

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

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.

2.2.1 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 [28, 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 [28, 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 [16, 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 [37, 42].

Table 2.1: Result of the `typeof` operator by a value's type [28, 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 [4, p. 40].

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

2.2.3 Type Conversion

In JavaScript "any [...] value can be converted to a boolean value [4, p. 40]". If the interpreter expects a boolean value, it simply performs a conversion [4, p. 46] (see 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 [4, p. 46, 28, 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 [28, 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 [4, 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"` [20].

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 [4, 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 from 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 [28, p. 5])—including functions and arrays—is an object, hence making JavaScript a highly flexible language. A key concept of the language is the prototypal inheritance of objects. Every object has a prototype, which is just a reference to another object. Anyway, the prototype is not accessible for all types of objects, but for functions, or more precisely, constructors [28, p. 3] (see figure 2.1). A constructor function is just an ordinary JavaScript function, that—by convention—begins with a capital letter [11, p. 8]. Initial values can be defined, that may be different for every instance created from the constructor.

```
function Person(name) {
  this.name = name;
}
```

On the other hand, properties on the function's prototype may be defined. This properties are shared across all instances of *Person*.

```
Person.prototype.speak = function speak() {
  return "Hi, my name is " + this.name;
}
```

By calling `thomas.speak()` in the code below, JavaScript looks for a *speak* property on the object *thomas*. As there doesn't exist any property with that name, the interpreter looks at the object's prototype and successfully calls the method [14, pp. 85–86].

[8] Todo: Explain by reference and by value shortly

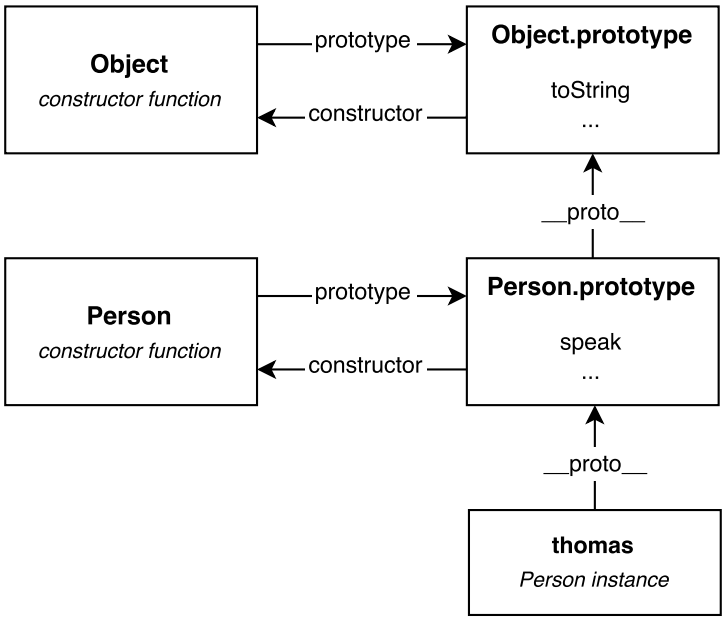


Figure 2.1: Prototypal inheritance in JavaScript.

```
let thomas = new Person("Thomas");
thomas.speak(); // "Hi, my name is Thomas"
```

If again no property *speak* would exist on the prototype, JavaScript would look at the prototype's prototype, and so on. This is called the prototype chain [14, p. 86].

2.2.6 Latest Improvements

JavaScript is recently improving rapidly, with a number of minor and major additions and improvements to the language's standard. Some additions improve readability and reduce the amount of lines of code needed to accomplish the same things in previous versions. Others add completely new functionality and concepts to JavaScript. The following few sections will give a shallow overview of a few additions to the 6th edition of ECMAScript—called *ECMAScript 2015*—as they make it most distinctive to the standard's previous edition.

Declaration Keywords

Until and including the fifth edition of ECMAScript (i.e., ES5), the only declaration keyword available was `var` [25, p. 87]. As of the sixth edition of ECMAScript (i.e., ES6), the keywords `let`—as seen in previous code snippets—and `const` were added [28, p. 194]. In order to understand the impact of using one keyword over another, it is important to know the basics of how variables are scoped in JavaScript. Simpson defines a scope as

> [...] the set of rules that govern how the engine can look up a variable by its identifier name and find it, either in the current scope, or in any of the nested scopes it's contained within [13, p. 13].

Program 2.1: Variable `i` was declared on line 7 as counter for a for loop. When function `bar` is called from the loop, the identifier `i` exists in the scope of `bar`, or rather in its enclosing scope `foo`, and `i` is assigned the value `2`. This results in an infinite loop, as the loop will never reach its condition to stop of `i` being equal to or greater than `10`. [13, p. 26]

```
1 function foo() {
2
3   function bar() {
4     i = 2;
5   }
6
7   for(var i = 0; i < 10; i++) {
8     bar();
9   }
10
11 }
12
13 foo();
```

JavaScript makes use of a *lexical scope* model, which is based on where variables and scope blocks (e.g. functions) are written in the code [13, p. 13]. This means that

> [n]o matter *where* a function is invoked from, or even *how* it is invoked, its lexical scope is *only* defined by where the function was declared [13, p. 16].

Program 2.1 gives a simple example of how scopes behave in JavaScript and also shows the importance to be aware of it. While there are ways to get around lexical scoping in JavaScript, those mechanisms are considered bad practice [13, p. 14] and come with performance issues [13, p. 21], hence won't be covered here.

Before `let` and `const` were introduced, the easiest way to create a scope was a function [15, p. 7]. Other programming languages, like Java, support block scope [13, p. 7], which means that variables will be scoped by any block that is created, including loops. JavaScript, however, has a function scope, as shown previously in program 2.1. As of ES6, with `let` it is possible to make use of block scoped variables, whereas `var` would lead to the variable being scoped to its parent function or the global scope, if no enclosing function exists. The code below demonstrates, that creating a simple block in JavaScript in combination with a `var` declaration, does not scope the identifier to that block [15, p. 8].

```
1 var a = 1;
2
3 {
4   var a = 2
5 }
6
7 console.log(a); // 2
```

On the other hand, when declaring `a` on line 4 with the `let` keyword, the variable will be scoped to its enclosing block. In this case it doesn't make any difference, whether

Program 2.2: The function scope was outlined in program 2.1 when using a `var` declaration for a loop, resulting in an infinite loop. In the example below, the only difference to program 2.1 is that `var` has been replaced in favor for `let` on line 7. This causes the variable i being block scoped to the for loop, and *not* to its enclosing function `foo`. Therefore the assignment in `bar` won't change the value of `i` from line 7 and the loop is called exactly ten times.

```
1 function foo() {
2
3   function bar() {
4     i = 2;
5   }
6
7   for(let i = 0; i < 10; i++) {
8     bar();
9   }
10
11 }
12
13 foo();
```

the declaration on the first line uses `var` or `let`, as it is top level and will live in the global scope anyway.

```
1 var a = 1;
2
3 {
4   let a = 2
5 }
6
7 console.log(a); // 1
```

Anyway, it may be a good practice to use the block scope behavior for variables with `let` or `const` over `var` at any time, if not explicitly needed otherwise. This may prevent errors and unexpected behavior, as shown in program 2.2, when comparing to program 2.1.

The `const` keyword behaves exactly the same as `let`, with the only difference that it is a constant, meaning that its value is fixed and cannot be changed. An attempt to reassign a constant identifier would result in an error [13, p. 39].

```
const a = 1;
a = 2; // TypeError
```

However, this does not affect e.g. properties of an object assigned to a constant variable, unless the property itself is marked as not writeable.

```
const b = { name: "Foo" };
b.name = "Bar";
```

Arrow Functions

In order to understand the differences of the declaration keywords in the previous section, a fundamental understanding of scopes in JavaScript is indispensable. For the concept of arrow functions, also introduced in ECMAScript 2015, a basic knowledge of

Program 2.3: Line 5 of the program logs the global `window` object in browsers, whereas on line 9 the object `bar` is logged to the console [40, p. 18].

```
1 const foo = function() {
2   console.log(this);
3 };
4
5 foo();
6
7 const bar = { foo };
8
9 bar.foo();
```

the `this` keyword is required. In contrast to the function scope, `this` is bound in runtime and is not associated to where a function is placed in the code [14, p. 9]. Simpson puts it succinctly, that

> [w]hen a function is invoked, [...] an execution context is created. This [context] contains information about where the function was called from (the call-stack), *how* the function was invoked, what parameters were passed, etc. One of the properties of this [context] is the `this` reference, which will be used for the duration of that function's execution [14, p. 1].

Thus, the value bound to `this` differs and is influenced by *how* and from *where* a function is called. Arrow functions on the other hand use lexical instead of dynamic binding for `this` [15, p. 58]. Furthermore they inherit the `arguments` array from its parent, and also `super` and `new.target` are lexically bound [15, p. 59]. Program 2.3 shows the behavior when using a traditional function alongside `this`.

To highlight the syntactical and behavioral differences of a function compared to an arrow functions, the code below shows a normal function assigned to a constant, taking one parameter and returning a value.

```
const foo = function(a) {
  return a;
}
```

The same function can be written as an arrow function as shown in the following code snippet.

```
const foo = (a) => {
  return a;
}
```

It is possible to write the function even shorter. If only one parameter is given, the parenthesis around it can be omitted. Also when deciding not to wrap the function's body with curly brackets, the result of the statement is returned automatically, therefore typing `return` is not required, as shown in the code below.

```
const foo = a => a;
```

The main purpose of arrow functions, however, is not to reduce the number of characters needed for a function, but the lexical binding of `this` as shown in program 2.4. Using

Program 2.4: Unlike in program 2.3, where line 5 and 9 logged different objects to the console, in this example, both will log the global `window` object, due to the lexical binding of the arrow function, defined on line 1.

```
1 const foo = () => {
2   console.log(this);
3 };
4
5 foo();
6
7 const bar = { foo };
8
9 bar.foo();
```

Program 2.5: A class in JavaScript as of ECMAScript 2015.

```
1 class Foo {
2
3   constructor(a, b) {
4     this.a = a;
5     this.b = b;
6   }
7
8   bar() {
9     return this.a + this.b;
10  }
11
12 }
```

an arrow function over a function or vice versa, without being aware of the differences, may result in unexpected behavior.

Classes

The introduction of classes was a major step for the language's standard, although the concept is not new to JavaScript and has been used before, as seen in section 2.2.5. Program 2.5 shows a class in ES6, whereas program 2.6 demonstrates how the same result could be achieved in JavaScript prior to the sixth edition of ECMAScript. Both variants are valid in ES6 and can be used in the same way as follows:

```
const foo = new Foo(1, 2);
foo.bar(); // 3
```

When looking at program 2.6, which shows how to accomplish a class-like behavior in ES5 and below, it is made clear, that classes in JavaScript don't work like traditional classes in other languages, and actually rely on its concept of prototypes [15, p. 135].

Program 2.6: A class in JavaScript prior to ECMAScript 2015.

```
function Foo(a, b) {
  this.a = a;
  this.b = b;
}

Foo.prototype.bar = function() {
  return this.a + this.b;
}
```

String Concatenation

JavaScript has been using the addition operator for string concatenation, which is still possible in the latest standard [27]. In the sixth edition of ECMAScript, template literals were introduced [28, p. 148, 15, pp. 47–48], giving developers more flexibility when working with strings. To showcase the ordinary way to add one string to another, the following code is given:

```
let firstname = "Foo";
let lastname = "Bar";
```

To put the values of these variables together, the addition operator can be used:

```
firstname + " " + lastname; // "Foo Bar"
```

In order to insert a space between `firstname` and `lastname`, it has to be added as a string between the two variables. The same result could be achieved, by creating an array from these identifiers and to join the values by a space:

```
[firstname, lastname].join(" "); // "Foo Bar"
```

Starting with ES6, another possibility is to use template strings—delimited with backticks rather than quotes—where expressions can be inserted [15, p. 48]:

```
`${firstname} ${lastname}`; // "Foo Bar"
```

The result of all three concatenation techniques from above is identical.

Beyond ECMAScript 2015

The development of JavaScript is dependent on its specification defined by ECMAScript and it took long for new editions to be released [39]. Edition 5.1 was published in 2011 [29], from where it took four years until the sixth edition was released in June 2015 [26]. Starting with ECMAScript 2015, a new specification will be released yearly [30], with the seventh edition of ECMAScript from 2016 being the latest [38, 27].

2.2.7 Extensions

There are a multitude of extensions available for JavaScript. In [35] an extensive list of languages that compile to JavaScript, JavaScript supersets, parsers and compilers can be found. That list includes *CoffeeScript*[9]—a "[...] language that compiles into

[9] http://coffeescript.org

JavaScript" [22]—as well as the static type checkers *TypeScript*[10] and *Flow*[11]. Languages that have packages available for transforming its code to JavaScript include, but are not limited to, *Ruby*, *Python*, *Java*, *Scala* and *C#*.

2.2.8 Further Reading

This section handled the most important concepts of JavaScript with a focus on the characteristics that will encourage the value of runtime type checks in JavaScript, discussed later in chapter **??**. Various exceptions or details were not handled, as they would go beyond the knowledge required for this thesis. If a more sophisticated knowledge of the language is desired, the *You Don't Know JS* series by Simpson, *JavaScript: The Good Parts* by Crockford and *JavaScript: The Definitive Guide* by Flanagan, among others, are recommended.

2.3 Abstract Syntax Tree

An *abstract syntax tree* (i.e., *AST*) is the representation of a source program, created for analyzation purposes [6, p. 19], containing only the indispensable parts of the code [18, p. 12] for the most parts. A syntax tree—or abstract syntax tree—is usually created by the compiler at an early stage. More specifically it is usually the second out of five compilation phases [6, pp. 2–3]:

1. The *scanner* or *lexical analyzer*, reads and tokenizes the source code, where a token is typically a keyword, an identifier or a literal.
2. In the next step the *parser*, also called *syntactic analyzer*, combines multiple tokens to e.g. an expression, a statement or a declaration. The result of the parser is the abstract syntax tree.
3. The *semantic analyzer* performs, among other things, type checks and range checking.
4. In the fourth step of a typical compiler, the *optimizer* creates intermediate code and applies code improvement algorithms.
5. The *code generator* is the last step, where the final target code of a program is generated.

To demonstrate, how an abstract syntax tree may look like for JavaScript, a simple variable declaration was inserted in the online editor *AST explorer*[12], which can visualize the syntax tree generated by numerous parsers, including JavaScript, various JavaScript supersets such as TypeScript and Flow, CSS, and HTML [34]. The resulting syntax tree is illustrated in figure **??**.

[10] https://www.typescriptlang.org
[11] https://flow.org
[12] https://astexplorer.net

Chapter 3

State of the Art

The following chapter will provide an overview of available technologies and techniques to introduce a static type system—as defined in section 2.1—to JavaScript. The main focus will be on TypeScript, but also other supersets will be explored. In conclusion, existing projects, to enable runtime type checks for these supersets, will be highlighted. Weisstein defines a *superset* as

> [a] set containing all elements of a smaller set. If B is a subset of A, then A is a superset of B [...] [43].

This means that every program that is valid in JavaScript is also legal in a JavaScript superset, where the purpose of such a superset is to add features to the original language. As the source written in the supersets syntax will be compiled to JavaScript, any additional functionality needs to be representable as JavaScript code.

3.1 TypeScript

TypeScript was created by Anders Hejlsberg—the designer of C#—at Microsoft [12, p. 10] and was released in 2012 under the Apache 2.0 open source license [3, p. xix]. The most important aspect of TypeScript is, that it includes a compilation step, where static type checking is performed [12, p. 11]. Type annotations are optional and the compiler will infer type information where possible [40, p. 10]. TypeScript also introduces concepts known from other programming languages, such as interfaces and enumerations (i.e., enums). Not only it is possible to develop a program in the TypeScript syntax, but also to add type annotations to existing JavaScript projects [12, p. 13]. The most significant particularities and features will be explored in this section.

3.1.1 Basic Types

Like many major languages, TypeScript defines some basic types, that overlap with JavaScript's types, listed in section 2.2.2, while also introducing new ones [21]:

- *Tuple* is a special kind of an array, only allowing a fixed number of elements.
- *Enum* may already be known from other programming languages, like Java, and is useful to define a set of numeric values.

- *Any* results in type checking not being performed by the compiler. This is useful when using TypeScript alongside third-party-libraries where no type definitions are available. It also allows e.g. access to any property on an object, whether it does exist or not.
- *Void* is the counterpart to Any. Again, it is used in other languages, for example to annotate functions that do not return a value. In TypeScript also variables may be typed as `void`, meaning that only `undefined` or `null` will be accepted as value.
- *Never* for instance is useful for functions that always throw an error or result in an infinite loop, as no value will ever come back [21].

While other types, such as *Function* or more advanced structures are also available, the ones listed in combination with those already defined in JavaScript (i.e., Undefined, Null, Boolean, String, Symbol, Number, and Object) are the most important ones to get started.

3.1.2 Type Inference

As already mentioned, TypeScript infers the type if possible. The snippet below shows a simple variable declaration in JavaScript (or TypeScript), where the TypeScript compiler can automatically infer the type Number from the declaration.

```
let num = 1;
```

Therefore it won't allow any subsequent assignment to `num` not being a number. For example the reassignment `num = "foo"` would result in the following compiler diagnostic:

```
Type '"foo"' is not assignable to type 'number'.
```

The term *diagnostic* over *error* is used here, because by default the compiler won't stop in such cases and will do its best to emit the final JavaScript code [40, p. 12]:

```
let num = 1;
num = 'foo';
```

The code above shows the resulting JavaScript program, even though the compiler detected a type error.

3.1.3 Type Annotations

While type inference can be useful in some situations, others require types to be set explicitly, as shown below:

```
let num: number;
num = 1;
```

The variable `num` was declared, but not initialized, requiring a type annotation, in order to be treated as a number by TypeScript. Omitting the explicit type information, the compiler would infer the Any type, allowing arbitrary assignments to the variable.

3.1.4 Type Assertions

Type assertions are a way to provide TypeScript with type information, which is not available to the compiler. They are

> [...] like [...] type [casts] in other languages, but [perform] no special checking or restructuring of data [21].

It is the developer that has to take care of performing sufficient checks when using a type assertion. Because of the possibility to use any existing JavaScript library with TypeScript, situations where the compiler does not have any type information of the external package available may occur. Type assertion can be a solution to prevent compile time type errors in such cases:

```
import RandomName from 'random-name';
let name: string = (RandomName as any).getName();
```

In the sample from above, the default export from the library `random-name` was imported as `RandomName`. This package is neither written in TypeScript nor does it have type definitions available. However, the library has a callable `getName` property, returning a string. As the compiler is not aware of the package's properties and their return types, it is necessary to tell it which type to assert. `RandomName` was casted to `any`, allowing property access independently of their existence. Again, because of type assertions (or type casts) not performing any special checking, the solution from above may lead to errors if the author of the package decides to change its API[1]. Therefore manually checking for if `RandomName` has a callable `getName` property and is actually returning a string may be recommended here. As an alternative to the type casting syntax with the `as` keyword, the following may be used:

```
let name: string = (<any>RandomName).getName();
```

However, this *angle-bracket* syntax is not supported when using TypeScript with *JSX*[2] [21], making the `as` syntax preferable.

3.1.5 Ambient Type Declarations

In TypeScript either existing structures—such as classes and basic types—can be used as type annotation, or they can be defined via interfaces or type aliases. The latter are not part of the code after compilation, while e.g. classes or enums remain in the JavaScript code. Anyway, it is also possible to declare, among others, a class or variable as ambient in TypeScript. This may be useful when consuming a third party package, that was not written in TypeScript and there are no type definitions for it available. In the previous section type casting was used to circumvent this issue. While this is a possibility, it may not be suitable, if the library is used frequently in a project. Prepending e.g. a variable, class, namespace or enum with the `declare` keyword, makes them ambient and will be treated by the compiler as if they were part of the code:

```
declare const RandomName: any;
```

This results in TypeScript always treating the variable `RandomName` as having type `any`. Alternatively the imported package may be described in more detail:

```
declare const RandomName: {
  getName: () => string;
};
```

[1] TODO: Explain API

[2] "JSX is an embeddable XML-like syntax [...] meant to be transformed into valid JavaScript [which] came to popularity with the React framework, but has since seen other applications as well." [33]

Program 3.1: An instance of `Person` can be assigned to a variable with type `Named` on line 10, because of TypeScript's structural type system. In languages with a nominal type system the class `Person` would need to implement the interface `Named` in their corresponding syntax, for this example to be valid [41].

```
1 interface Named {
2     name: string;
3 }
4
5 class Person {
6     name: string;
7 }
8
9 let p: Named;
10 p = new Person();
```

From now on, TypeScript will know, that the import has a callable property `getName` that returns a string.

3.1.6 Structural Types

Types in TypeScript are structural [40, p. 11], meaning that the type checker looks at members of an object, to ensure type compatibility, while other major languages, such as C# or Java, use nominal type systems [41]. Program 3.1 gives an example that would fail in a nominally typed language, but is possible in TypeScript.

3.1.7 Classes

TypeScript not only enables static type checking for JavaScript applications, but also includes additional language features. While EcmaScript 2015 introduced classes, with TypeScript it is possible to also add visibility modifiers and interface implementation to them as shown below.

```
class Person implements Human {
  public name: string;
  private age: number;
}
```

The keywords `public`, `protected` and `private` may be used for class members and methods. Also it is possible to define members and provide default values outside of the constructor, as well as to mark properties as `readonly`, preventing any reassignment.

```
class Person implements Human {
  public readonly id = uid();
}
```

Anyway, it is important to note, that the modifiers described, as well as implemented interfaces, are only relevant during compile time. After the final JavaScript code has been emitted, this information is stripped out and cannot be used in the running program, as shown in the compiled class, using plain JavaScript syntax, below.

```
class Person {
```

```
  constructor() {
    this.id = uid();
  }
}
```

This means that it is technically possible, to assign any arbitrary value to `id` of a `Person` instance during runtime.

3.1.8 Enums

As already mentioned earlier, an enumeration is beneficial for defining a set of numeric values. The TypeScript compiler will take any enum declaration and transform it into runnable JavaScript code. The following enum in TypeScript syntax

```
enum HairColor {
  Black, Blond, Brown, Red, Other
}
```

results in the JavaScript code below, which creates an object containing the values defined in the enumeration.

```
var HairColor;
(function (HairColor) {
    HairColor[HairColor["Black"] = 0] = "Black";
    HairColor[HairColor["Blond"] = 1] = "Blond";
    // ...
})(HairColor || (HairColor = {}));
```

The enum can now be accessed during runtime to obtain the corresponding numeric value. Also it is possible, to reveal the name of a numeric value from the enumeration:

```
HairColor.Black // 0
HairColor[0] // "Black"
```

However, if the enumeration is declared as constant, the compiler will look up the numeric value and will insert it directly into the source code, before entirely removing its definition [31].

3.1.9 Namespaces

In TypeScript, namespaces provide a possibility to encapsulate code. They were previously referred to as *internal modules*, but have since been renamed to avoid confusion with native *modules* of the EcmaScript standard, previously denoted as *external modules* in TypeScript [36]. Code within a namespace only expose explicitly exported parts.

```
namespace Capsule {
  let foo = "Hello from Capsule!";

  export function bar() {
    return foo;
  }
}
```

Accessing `foo` of the namespace `Capsule` would result in `undefined`, whereas calling `bar` would return the value of `foo`. If taking a look at the JavaScript code, generated from the namespace above, this behavior is made clear.

```
var Capsule;
(function (Capsule) {
    var foo = "Hello from Capsule!";
    function bar() {
        return foo;
    }
    Capsule.bar = bar;
})(Capsule || (Capsule = {}));
```

A variable with the name of the namespace will be declared and an empty object will be assigned to it. Only the namespace's exported parts will be added to this object to be exposed, while all other values remain exclusively accessible from within the self executing function itself.

3.1.10 Parameter Default Values

Another useful feature is the possibility to define default values for parameters in TypeScript. This gives developers the ability to avoid parameters being `undefined` if not passed, and can be useful in various other scenarios.

```
function log(message: string, logger = console) {
  logger.log(message);
}
```

The example above shows a simple log function which writes a string to the console, when omitting the second parameter. If another log mechanism is desired, it is possible to pass a different logger to this method, which also implements the Console interface, or at least shares its properties. The same result can be achieved with plain JavaScript as well, as shown in the compiled code below:

```
function log(message, logger) {
    if (logger === void 0) { logger = console; }
    logger.log(message);
}
```

If the parameter `logger` equals `void 0`, which is identical to `undefined`[3], the global variable `console` will be assigned to it. Otherwise the parameter passed to the function `log` will be used as is.

3.1.11 Future JavaScript

3.2 Flow

Flow[4] is another open source static type checker for JavaScript, developed by *Facebook*[5].

[3] TODO: explain void 0
[4] https://flow.org
[5] https://code.facebook.com/projects/

3.3 Other Supersets

3.4 Comparison

3.5 Existing Runtime Projects

Chapter 4

Theoretical Approach

After defining the terminology for this thesis, as well as giving an exposure to available technologies and projects, the theoretical approach for generating runtime type checks from TypeScript type annotations for runnable JavaScript code will be described in this chapter.

4.1 Undetectable Errors

There are various situations the static type analysis of TypeScript cannot detect errors that may occur during runtime. Either a project is written in TypeScript, so the compiler can infer type information needed, or type definition files may be provided for untyped JavaScript libraries. In both cases it is possible to introduce errors, which may cause the type checker to make wrong assumptions about type compatibility. Also particular programming techniques can result in errors not being trapped already during compilation.

4.1.1 Compiler Analysis Circumvention

In TypeScript it is possible to perform a special kind of type cast, called *type assertion*, as described in section 3.1.4. While the compiler will trigger an error, when trying to assert an incompatible type—e.g. asserting a string as a number—there exists a special case to bypass static type checks for a given variable or value entirely. If a variable is annotated or asserted with the *any* type, type checking and type inference will be disabled. The TypeScript documentation describes this type as

> [...] a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type-checking during compilation. [21]

It seems legitimate to use *any* alongside the utilization of third party libraries or in situations where the flexibility of JavaScript's loose typing (see sec. 2.2.1) is required. However, opening the possibility to opt-out of type checks can have a negative impact for projects depending on libraries where this technique is misused. The following code snippet outlines a situation, where compilation passes, but an error is thrown at runtime:

```
let foo: any = 'bar';
foo.getNumber();
```

Because of type checks being disabled for variable `foo`, access to the not existing property `getNumber` won't be detected by the compiler. Even if the identifier was annotated correctly, or its type could be inferred by omitting a type annotation, it is possible to get around type checks:

```
let foo: string = 'bar';
(foo as any).getNumber();
```

In both cases, the JavaScript runtime engine will throw *TypeError: foo.getNumber is not a function*. The examples from above highlight the potentiality of creating conditions where a detectable mistake remains undiscovered by the compiler, which could cause a program to crash.

4.1.2 Definition File Mistakes

4.1.3 Erroneous API Responses

4.2 Method Principle

4.3 Type Check Situations

4.3.1 Main Cases

4.3.2 Edge Cases

4.3.3 Exceptions

4.3.4 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, pp. 97-1–97-41 (cit. on pp. 2–5).

[2] Douglas Crockford. *JavaScript: The Good Parts. Unearthing the Excellence in JavaScript*. 1st ed. Sebastopol, California, USA: O’Reilly Media, 2008 (cit. on pp. 4, 15).

[3] Stevens Fenton. *Pro TypeScript. Application-Scale JavaScript Development*. 2nd ed. Apress, Feb. 2017 (cit. on p. 16).

[4] David Flanagan. *JavaScript: The Definitive Guide. Activate Your Web Pages*. 6th ed. O’Reilly Media, 2011 (cit. on pp. 6, 7, 15).

[5] Philippa Anne Gardner, Sergio Maffeis, and Gareth David Smith. “Towards a Program Logic for JavaScript”. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’12. Philadelphia, PA, USA: ACM, 2012, pp. 31–44 (cit. on p. 3).

[6] Kenneth C. Louden. “Compilers and Interpreters”. In: *Computer Science Handbook*. Ed. by Allen B. Tucker. 2nd ed. CRC Press, 2004. Chap. 99, pp. 99-1–99-30 (cit. on p. 15).

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

[8] Jaime Niño. “Type Systems Directed Programming Language Evolution: Overview and Research Trends”. In: *Proceedings of the 50th Annual Southeast Regional Conference*. ACM-SE ’12. Tuscaloosa, Alabama: ACM, 2012, pp. 203–208 (cit. on p. 3).

[9] Den Odell. *Pro JavaScript Development. Coding, Capabilities, and Tooling*. 1st ed. Expert’s voice in Web development. California, USA: Apress, 2014 (cit. on p. 4).

[10] Benjamin C. Pierce. *Types and Programming Languages*. 1st ed. Cambridge, Massachusetts, USA: The MIT Press, 2002 (cit. on pp. 2–4).

[11] Martin Rinehart. *JavaScript Object Programming*. 1st ed. Apress, 2015 (cit. on pp. 4, 8).

[12] Nathan Rozentals. *Mastering TypeScript*. 1st ed. Packt Publishing, 2014 (cit. on p. 16).

[13] Kyle Simpson. *Scopes & Closures*. 1st ed. You Don't Know JS. Sebastopol, California, USA: O'Reilly Media, Mar. 2014 (cit. on pp. 9–11).

[14] Kyle Simpson. *this & Object Prototypes*. 1st ed. You Don't Know JS. Sebastopol, California, USA: O'Reilly Media, Apr. 2015 (cit. on pp. 8, 9, 12).

[15] Kyle Simpson. *Types & Grammar*. 1st ed. You Don't Know JS. Sebastopol, California, USA: O'Reilly Media, Jan. 2016 (cit. on pp. 10, 12–14).

[16] Kyle Simpson. *Up & Going*. 1st ed. You Don't Know JS. Sebastopol, California, USA: O'Reilly Media, Apr. 2015 (cit. on pp. 5, 15).

[17] Sam Tobin-Hochstadt and Matthias Felleisen. "Logical Types for Untyped Languages". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. Baltimore, Maryland, USA: ACM, 2010, pp. 117–128 (cit. on p. 3).

[18] Christian Wagenknecht and Michael Hielscher. *Formale Sprachen, abstrakte Automaten und Compiler. Lehr- und Arbeitsbuch für Grundstudium und Fortbildung*. 2nd ed. Springer, 2014 (cit. on p. 15).

Online sources

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

[20] *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).

[21] *Basic Types - TypeScript*. URL: https://www.typescriptlang.org/docs/handbook/basic-types.html (visited on 05/08/2017) (cit. on pp. 16–18, 23).

[22] *CoffeeScript*. 2017. URL: http://coffeescript.org (visited on 05/08/2017) (cit. on p. 15).

[23] 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).

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

[25] *ECMAScript 2015 Language Specification. ECMA-262 5.1 Edition*. June 2011. URL: http://www.ecma-international.org/ecma-262/5.1/Ecma-262.pdf (visited on 04/23/2017) (cit. on p. 9).

[26] *ECMAScript 2015 Language Specification - ECMA-262 6th Edition*. June 2015. URL: https://www.ecma-international.org/ecma-262/6.0/ (visited on 04/23/2017) (cit. on p. 14).

[27] *ECMAScript 2016 Language Specification - ECMA-262 7th Edition*. June 2016. URL: https://www.ecma-international.org/ecma-262/7.0/ (visited on 05/04/2017) (cit. on p. 14).

[28] *ECMAScript 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. 5–9, 14).

[29] *ECMAScript Language Specification - ECMA-262 5.1 Edition*. June 2011. URL: https://www.ecma-international.org/ecma-262/5.1/ (visited on 05/04/2017) (cit. on p. 14).

[30] *ECMAScript Next support in Mozilla*. Jan. 2017. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_Next_support_in_Mozilla (visited on 05/04/2017) (cit. on p. 14).

[31] *Enums - TypeScript*. URL: https://www.typescriptlang.org/docs/handbook/enums.html (visited on 05/08/2017) (cit. on p. 20).

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

[33] *JSX - TypeScript*. URL: https://www.typescriptlang.org/docs/handbook/jsx.html (visited on 05/08/2017) (cit. on p. 18).

[34] Felix Kling. *A web tool to explore the ASTs generated by various parsers*. 2017. URL: https://github.com/fkling/astexplorer (visited on 05/08/2017) (cit. on p. 15).

[35] *List of languages that compile to JS*. 2017. URL: https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js (visited on 05/08/2017) (cit. on p. 14).

[36] *Namespaces - TypeScript*. URL: https://www.typescriptlang.org/docs/handbook/namespaces.html (visited on 05/08/2017) (cit. on p. 20).

[37] 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).

[38] *Standard ECMA-262*. URL: https://www.ecma-international.org/publications/standards/Ecma-262.htm (visited on 05/04/2017) (cit. on p. 14).

[39] *Standard ECMA-262 Archive*. URL: https://www.ecma-international.org/publications/standards/Ecma-262-arch.htm (visited on 05/04/2017) (cit. on p. 14).

[40] Basarad Ali Syed. *TypeScript Deep Dive*. 2017. URL: https://www.gitbook.com/download/pdf/book/basarat/typescript (visited on 05/08/2017) (cit. on pp. 12, 16, 17, 19).

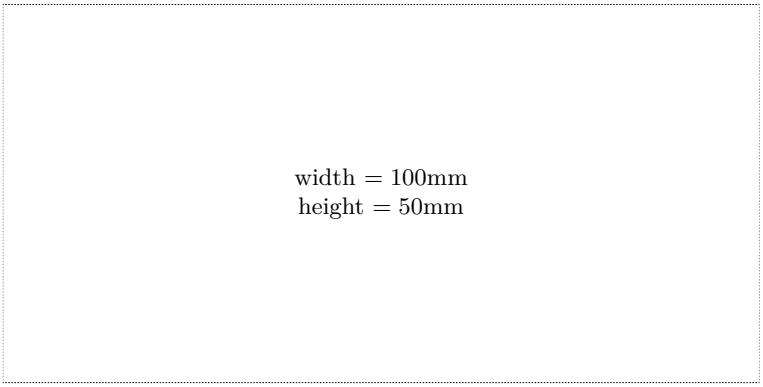
[41] *Type Compatibility - TypeScript*. URL: https://www.typescriptlang.org/docs/handbook/type-compatibility.html (visited on 05/08/2017) (cit. on p. 19).

[42] *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).

[43] Eric W. Weisstein. *"Superset." From MathWorld–A Wolfram Web Resource*. URL: http://mathworld.wolfram.com/Superset.html (visited on 05/08/2017) (cit. on p. 16).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —