

Generating Runtime Type Validations for JavaScript from the Static Type Information of its Superset TypeScript

Fabian Pirklbauer



MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2017

© Copyright 2017 Fabian Pirklbauer

This work is published under the conditions of the Creative Commons License *Attribution-NonCommercial-NoDerivatives 4.0 International* (CC BY-NC-ND 4.0)—see <https://creativecommons.org/licenses/by-nc-nd/4.0/>.

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

Declaration	iii
Abstract	vii
Kurzfassung	viii
1 Introduction	1
1.1 Problem Definition	1
1.2 Solution Approach	1
1.3 Thesis Structure	2
2 Technical Foundation	3
2.1 Type Systems	3
2.1.1 Explicitly and Implicitly Typed	4
2.1.2 Execution Errors	4
2.1.3 Safety and Good Behavior	5
2.1.4 Type Checking	5
2.2 JavaScript	5
2.2.1 Loose Typing	6
2.2.2 Value Types	6
2.2.3 Type Conversion	7
2.2.4 Value Comparison	8
2.2.5 Objects and Prototypal Inheritance	9
2.2.6 Latest Improvements	10
2.2.7 Further Reading	15
2.3 Abstract Syntax Tree	16
3 State of the Art	18
3.1 JavaScript Supersets	18
3.1.1 TypeScript	18
3.1.2 Flow	24
3.1.3 Others	24
3.2 Runtime Type Checks	25
3.3 Generated Runtime Type Checks	28

4	Theoretical Approach	29
4.1	Undetectable Errors	29
4.1.1	Compiler Analysis Circumvention	29
4.1.2	Polymorphism	30
4.1.3	Untyped JavaScript Libraries	31
4.1.4	Type Declaration Mistakes	31
4.1.5	Erroneous API Responses	32
4.2	Desired Result	32
4.3	Definition of Cases	33
4.3.1	Interfaces and Type Aliases	33
4.3.2	Variable Declarations and Assignments	33
4.3.3	Type Assertions	33
4.3.4	Functions	33
4.3.5	Enums	34
4.3.6	Classes	34
4.3.7	Type Queries	34
4.3.8	Externals	34
4.3.9	Ambient Declarations	35
4.4	Required Steps	35
4.4.1	Configuration	35
4.4.2	Read Source Files	35
4.4.3	Syntax Analyzation	36
4.4.4	Abstraction	36
4.4.5	Scan Abstraction	36
4.4.6	Static Type Checks	36
4.4.7	Transformations	36
4.4.8	Target Code Emit	37
4.5	Summary	37
5	Implementation	38
5.1	Technology	38
5.1.1	TypeScript Compiler	38
5.1.2	Runtime Type System	40
5.2	Architecture	41
5.2.1	Central Element	42
5.2.2	Components	42
5.2.3	Outline	43
5.3	Application Structure	44
5.4	Components	44
5.4.1	Transformer	44
5.4.2	Mutators	45
5.4.3	Factory	53
5.4.4	Context	53
5.4.5	Utility	54
5.4.6	Event Bus	54
5.4.7	Scanner	54
5.4.8	Options	55

5.5	Transformation Procedure	56
5.6	Usage	57
5.6.1	Application Programming Interface	58
5.6.2	Command Line Interface	59
5.6.3	Playground	60
6	Evaluation	62
6.1	Automated Unit Tests	62
6.2	Continuous Integration	63
6.3	Operational Test	63
6.4	Performance Analyzation	65
6.5	Summary	65
7	Summary and Outlook	68
A	Evaluation Results	70
A.1	Build Time	70
A.2	Unit Tests Execution Time	71
A.3	Benchmark Tests	72
B	CD-ROM Contents	79
	References	80
	Literature	80
	Online sources	81

Abstract

Even though JavaScript's dynamic type system can be of advantage in many scenarios, it adds the risk of introducing errors. Therefore TypeScript came up with a superset of JavaScript, with the ability to optionally add type annotations, resulting in increased code readability, scalability and maintainability. In addition, TypeScript's static compile time type checks can detect a multitude of conditions that may cause issues in the target code at runtime. Although type compatibility is checked during compile time, type information is not available in the compiled JavaScript code. The removal of types is intended and is defined in the design goals of the language. Therefore extensive type checks have to be performed manually, which results in increased development effort and greater susceptibility to errors. The fact that suitable type information is available for most situations suggests that generating runtime type checks based on the existing data at compile time is technically possible. The information which is usually removed by the TypeScript compiler is reflected in the target code to obtain it for type compatibility checks during program execution in this thesis. These checks are generated and inserted in the resulting JavaScript code automatically, which should help to identify possible issues during the development of a project the TypeScript compiler cannot detect.

Kurzfassung

Obwohl das dynamische Typ-System von JavaScript in vielen Szenarien von Vorteil sein kann, erhöht es das Risiko, Fehler einzuführen. Um dem entgegenzuwirken, wurde das JavaScript-Superset TypeScript entwickelt, welches die Möglichkeit bietet, optional Typ-Annotationen einzufügen, wodurch die Leserlichkeit, Skalierbarkeit und Wartbarkeit des Codes erhöht wird. Zusätzlich können die statischen Typ-Überprüfungen, die vom TypeScript-Kompilierer durchgeführt werden, eine Vielzahl an Zuständen erkennen, welche zur Laufzeit des Zielcodes möglicherweise zu Problemen führen können. Obwohl die Typ-Kompatibilität zum Zeitpunkt der Kompilierung überprüft wird, sind die Typ-Informationen zur Laufzeit nicht verfügbar. Das Entfernen dieser Informationen ist beabsichtigt und in den Design-Zielen der Programmiersprache festgelegt. Aus diesem Grund müssen umfangreiche Typ-Prüfungen manuell durchgeführt werden, was in erhöhtem Entwicklungsaufwand und höherer Fehleranfälligkeit resultiert. Die Tatsache, dass geeignete Typ-Informationen für die meisten Situationen verfügbar sind, suggeriert, dass die Generierung von Laufzeit-Typ-Überprüfungen, basierend auf den vorhandenen Daten während der Kompilierung, technisch möglich ist. Jene Informationen, die vom TypeScript-Kompilierer entfernt werden, werden im Zielcode reflektiert, um sie für Typ-Kompatibilitätsprüfungen während der Programmausführung zu verwenden. Diese Überprüfungen werden im Zuge dieser Arbeit automatisch generiert und in dem resultierenden JavaScript-Code eingefügt, was während der Entwicklung eines Projekts helfen soll, mögliche Probleme zu identifizieren, die der TypeScript-Kompilierer nicht feststellen kann.

Chapter 1

Introduction

JavaScript is a popular programming language on the client and server side. It has evolved considerably in recent years and the latest specification called ECMAScript 2015—also known as ES6—which, among other things, introduced classes to native JavaScript, was a major step for developers. Even though JavaScript’s dynamic type system can be of advantage in a lot of scenarios, it adds the risk of introducing errors. The language tries to perform type conversions in situations where values are not compatible, which can lead to unexpected behavior. Therefore TypeScript came up with a superset of JavaScript, giving developers the ability to optionally add type annotations to their projects, resulting in increased code readability, scalability and maintainability. In addition, TypeScript’s static compile time type checks can detect a multitude of conditions, that may cause issues in the target code at runtime. The compiler also adds support for the latest JavaScript features and proposals [38, 97], which enables the use of future language characteristics that are not yet supported.

1.1 Problem Definition

Although type compatibility is checked during compile time, type information is not available in the compiled JavaScript code. The removal of types is intended by Microsoft and is defined in the design goals of the language [37]. A number of issues have been filed on Microsoft’s GitHub repository, requesting the ability to automatically generate runtime type checks [61, 72, 83], which were rejected due to being out of scope of the language’s goals [36, 63]. Therefore extensive type checks have to be performed manually for situations in which the compiler cannot detect errors, such as HTTP requests or untyped third party code. This results in increased development effort and greater susceptibility to errors.

1.2 Solution Approach

Given the fact that suitable type information is available for most situations—either through type annotations or type inference—suggests that generating runtime type checks based on the existing data at compile time is technically possible. The information which is usually removed by the TypeScript compiler should be reflected in the

target code to obtain it for type compatibility checks during program execution. These checks should be generated and inserted in the resulting JavaScript code automatically, which should help to identify possible issues during the development of a project the TypeScript compiler cannot detect. In order to achieve a desirable result, situations where verifications need to take place have to be identified carefully. Also the footprint of code added to a project, as well as the performance impact on the program being executed, should be as small as possible. While the main purpose of this project is its use in the phase of development, employing its technique in a production environment should be considered as well.

1.3 Thesis Structure

While this chapter's intention was to give an overview of the contents of this thesis, in Chapter 2 the technical foundation, including definition of terminology and the introduction of the programming language JavaScript as well as various other terms and concepts forming the basis for the remaining chapters, will be introduced. In Chapter 3 the superset TypeScript will be explored and compared to a similar project, and an overview of the current state on automated runtime type checks for JavaScript will be provided. After the rudiments of the topic have been handled and the state of the art in the field of runtime type checks for JavaScript has been examined, the theoretical approach for the thesis project will be elaborated in Chapter 4, followed by its implementation in Chapter 5. Finally, the result will be evaluated in Chapter 6, before summarizing the outcome of the thesis as well as giving an outlook into the future in Chapter 7.

Chapter 2

Technical Foundation

This chapter provides an overview of the technical knowledge required for this thesis. It gives an exposure to the different type systems, the programming language JavaScript and JavaScript supersets. Also terminology used throughout this paper is specified, since standards differ across sources [1, p. 97-1].

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 a 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, which can hold any value [1, p. 97-2]. 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. 97-2]. According to Cardelli from *Microsoft Research*¹ a type system is “a collection of type rules for a typed programming language” [1, p. 97-38] with the purpose to “[...] prevent the occurrence of *execution errors* during the running of a program” [1, p. 97-1]. He further equates *type system* with *static type system* and also Pierce defines type systems as being static [11, p. 2], which categorizes languages as untyped that may distinguish between types at runtime but do not have knowledge about types during compilation or interpretation, such as JavaScript (see Sec. 2.2). This notion is further supported by Loudon 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. [8, p. 331]

A widely adopted consensus in terminology is to use both, *untyped* (e.g., in [19, p. 117]) and *dynamically typed* (e.g., in [5, p. 32] and [9, p. 203]) for languages without a static type system. Anyway, following the terminology of Cardelli, expressions like *statically*

¹ <https://www.microsoft.com/research/>

typed or *dynamically typed* are avoided in favor for *statically checked* and *dynamically checked*, respectively [1, p. 97-1]. 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., in Java) it is explicitly typed, whereas in implicitly typed languages, type annotations are assigned automatically by the type system [1, pp. 97-2–97-3]. 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 [11, 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 faults 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. 97-3], as outlined in Sec. 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 *trapped errors*, *untrapped errors*, and *forbidden errors* [1, p. 97-3].

Trapped Errors

A trapped error causes a program to stop immediately [1, p. 97-3] or to raise an exception, which may be handled in the program [11, p. 7]. An example for such an error is a division by zero [1, p. 97-3].

Untrapped Errors

Errors where a program does not crash or raise an exception immediately are called untrapped errors [1, p. 97-37]. They may remain unnoticed—at least for a while—and can lead to unexpected behavior [1, p. 97-3]. For example accessing data from an array that is out of bounds is legal in the programming language C [11, p. 7], but can lead to errors or arbitrary behavior later in the program [1, p. 97-3].

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. 97-3]”. 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. 97-4].

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. 97-3], consequently good behavior implies safety. Not all major languages are safe, and therefore not well behaved, such as C or C++ [11, 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. 97-5].

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. 97-37]), as described in Sec. 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. 97-3]. Dynamically checked programming languages, on the other hand, may also ensure good behavior by applying sufficient checks at runtime. Anyway, statically checked languages may also perform verifications during the execution of a program to guarantee safety, if not all untrapped errors can be discovered statically during compilation [1, p. 97-4].

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 [10, p. 28], an “industry association founded in 1961, dedicated to the standardization of information and communication systems” [46], 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 Sec. 3.1.1) is following up, which takes the 15th place with an increase of Pull Requests by 250% [62].

While JavaScript is known for programming inside browsers and for adding visual effects to websites [12, p. 4], its first use in a product was on the server-side in 1994 [10, 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*⁷, which is also used in the popular *Chrome*⁸ web browser [10, p. 369].

² *Netscape Communications*—founded as *Mosaic* in 1994—released its *Netscape Communicator* browser in 1995 which became the leading internet browser at that time [41].

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 [23].

⁶ <https://nodejs.org>

⁷ <https://developers.google.com/v8/>

⁸ <https://www.google.com/chrome/>

The following sections give an overview of the language JavaScript, outlining its most important and interesting concepts. As it would go beyond the scope of this thesis not all cases—especially the numerous exceptions—are described.

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 can 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, which has semantic errors, but evaluation is performed during execution, and errors are determined and reported during runtime [51, p. 291]. Therefore JavaScript can be deemed a *dynamically checked* language (see Sec. 2.1.4).

2.2.2 Value Types

JavaScript is an untyped and dynamically checked—but safe—scripting language, as defined in Sec. 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. 97-4]. Even though considered as untyped, the ECMAScript language specification defines seven value types [51, 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 runtime, the assigned value's type is determined and returned as a string [18, p. 30]:

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

The string that is returned by the `typeof` operator does not reflect the previously mentioned value types completely. As shown in Tab. 2.1, objects are differentiated by whether they are callable or not. For objects with a call signature, `"function"`⁹ is returned, and

⁹ Code sequences that are set in quotation marks denote a string, whereas, e.g., identifiers, keywords, and operators—such as `typeof`—are not quoted. If it is explicitly pointed out that a given code is a string, the quotation marks may be omitted.

"object" otherwise. For a value of type *Null* the result is "object" as well. A proposal to change the specification and to correct this issue—erroneously indicating that `null` is an object—was rejected, as existing code may break [85, 95].

Table 2.1: Result of the `typeof` operator by a value's type. [51, p. 164]

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

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 performs a conversion [4, p. 46] (see Sec. 2.2.4). Tab. 2.2 gives an overview of which values are evaluated as *true* or *false* when being converted to a boolean value.

Table 2.2: Values evaluated as *false* or *true* when converted to a boolean value. [4, p. 40]

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

There are various situations 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 tries to come up with a sufficient result. Anyway, it has a major influence on the outcome how such an expression is written. In the example above the string is seen first by JavaScript, therefore the subsequent numbers are 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"
```

Table 2.3: Type conversions in JavaScript. [4, p. 46, 51, pp. 36–44]

<i>Initial Value</i>	<i>String</i>	<i>Number</i>	<i>Boolean</i>	<i>Object</i>
undefined	"undefined"	NaN	false	<i>TypeError</i>
null	"null"	0	false	<i>TypeError</i>
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)	<i>TypeError</i>	<i>TypeError</i>	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 [51, 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* is preferred over *toString*.

(iv) The *toString* method of the *Array* object joins the array separated by a comma, which results in ["a", "b", "c"] being converted to "a,b,c" [25].

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

2.2.4 Value Comparison

In Sec. 2.2.3 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 value if it cannot

perform a comparison at first. Comparing a string that holds a numerical value to an actual number gives the same result as comparing two values of type *Number*, since the interpreter implicitly casts the string to a number:

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

When comparing with the equality operator (i.e., `==`) 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 reference¹⁰, whereas if the value to compare an object to is a number or a string, JavaScript tries to convert the object to a primitive value, either by using the object's *toString* or *valueOf* method.
- All other comparisons are not equal.

If a more detailed comparison is required and an automatic conversion of values is not desired, the *strict* equality operator (i.e., `===`) can be used. Only if type *and* value match, the expression evaluates to true:

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

Following the rules defined 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, which returns `"[object Object]"`, resulting in the compared values being equal. When making use of the *strict* equality operator no conversion is performed and the expression evaluates to false.

2.2.5 Objects and Prototypal Inheritance

Every value, which is not a primitive value (i.e., *Undefined*, *Null*, *Boolean*, *Number*, *Symbol*, or *String* [51, 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 a reference to another object. Anyway, the prototype is not accessible for all types of objects, but for functions, or more precisely, constructors [51, p. 3] (see Fig. 2.1). A constructor function is an ordinary JavaScript function, which—by convention—begins with a capital letter [12, p. 8]. Initial values can be defined, which may be different for every instance, created from the constructor:

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

¹⁰ In JavaScript an object's reference "[...] points to [its] location in memory" [24].

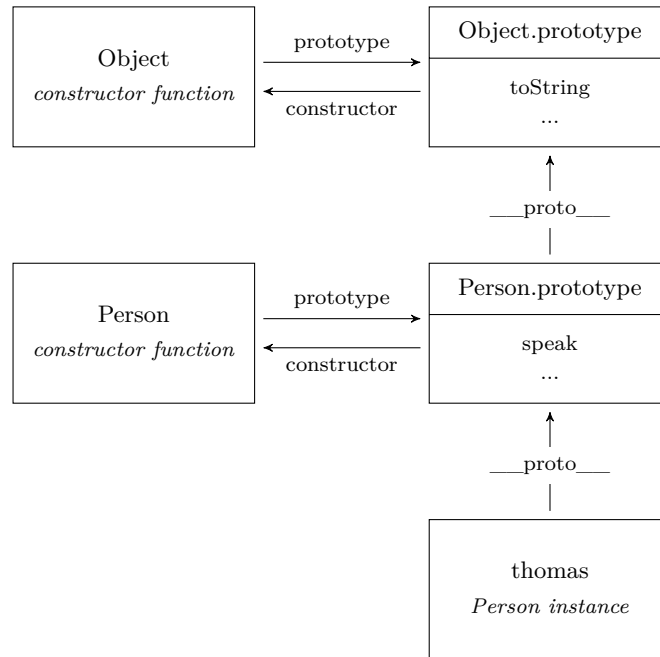


Figure 2.1: Prototypal inheritance in JavaScript.

On the other hand, if properties are defined on the function’s prototype, they 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 this name is non-existent on the object itself, the interpreter looks at the object’s prototype and successfully calls the method [16, pp. 85–86].

```

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

```

If no property *speak* would exist on the prototype neither, JavaScript would look at the prototype’s prototype recursively, until reaching *Object*. This is called the *prototype chain* [16, p. 86].

2.2.6 Latest Improvements

JavaScript is improving rapidly with a number of major and minor changes and improvements to the language’s standard. Some additions improve readability and reduce the amount of lines of code needed to accomplish the same outcome in previous versions. Others add completely new functionality and concepts to JavaScript. The following sections give an overview of additions to the sixth edition of ECMAScript—called *ECMAScript 2015*—that make it most distinctive to the standard’s previous version.

Declaration Keywords

Up to the fifth edition of ECMAScript (i.e., ES5), the only declaration keyword available was `var` [48, p. 87]. As of the 6th edition of ECMAScript (i.e., ES6), the keywords `let`—as seen in previous code snippets—and `const` are also available [51, p. 194]. In order to understand the impact of using one keyword over another, a fundamental understanding of scopes in JavaScript is indispensable. 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. [15, p. 13]

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 [15, p. 13]. This means that

no 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. [15, p. 16]

Prog. 2.1 gives an 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 [15, p. 14] and come with performance issues [15, p. 21], hence won't be covered here.

Program 2.1: Variable `i` is declared on line 7 as counter for a for loop. When function `bar` is called from within 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 it will never reach its condition to stop of `i` being equal to or greater than 10 [15, 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();
```

Before `let` and `const` were introduced, the easiest way to create a scope was a function [17, p. 7]. Other programming languages, like Java, support block scope [15, p. 7], which means that variables are scoped by any block that is created, including loops. JavaScript, however, makes use of a function scope, as shown previously in Prog. 2.1. As of ES6 the declaration keyword `let` can be used to block-scope variables, whereas `var` leads 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 combination with a `var` declaration does not scope the identifier to that block [17, 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 is scoped to its enclosing block:

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

While behavior may vary when using different declaration keywords, exchanging `var` with `let` on line 1 of the previous code example would not have any impact, as the variable lives in the global scope either way [67, 99]. However, 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, which is outlined when comparing Prog. 2.1 to Prog. 2.2.

Program 2.2: In this program `var` has been replaced in favor for `let` on line 7, compared to Prog. 2.1. This causes variable `i` being scoped to the for loop, and *not* to its enclosing function `foo`. Therefore the assignment on line 4 does not change the value of `i`, 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 `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 results in an error [15, 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 object is immutable or its properties are marked as not writeable [39].

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

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. [87, p. 18]

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

Arrow Functions

For the concept of arrow functions, introduced in ECMAScript 2015, a basic knowledge of the `this` keyword is required. In contrast to the function scope, `this` is bound during runtime and is not associated to where a function is placed in the code [16, p. 9]. Simpson puts it to the point that

when 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 is used for the duration of that function's execution. [16, p. 1]

In other words, 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` [17, p. 58]. Additionally, they inherit the `arguments` array from its parent, and also `super` and `new.target` are lexically bound [17, p. 59]. Prog. 2.3 shows the behavior when using a regular function alongside `this`.

To highlight the syntactical and behavioral differences of functions compared to arrow functions, the code below shows a function assigned to a constant, taking one parameter and returning its 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 Prog. 2.4. Using an arrow function over a function, or vice versa, without being aware of the differences may result in unexpected behavior.

Program 2.4: Unlike in Prog. 2.3, where line 5 and 9 logged different objects to the console, in this example, both 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();
```

Classes

The introduction of classes was a major step for JavaScript's standard, although the concept is not new to the programming language and has been used before. Prog. 2.5 shows a class in ES6, whereas Prog. 2.6 demonstrates how the same result was 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 Prog. 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 the concept of prototypes [17, p. 135].

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 }
```

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

```
1 function Foo(a, b) {  
2   this.a = a;  
3   this.b = b;  
4 }  
5  
6 Foo.prototype.bar = function() {  
7   return this.a + this.b;  
8 }
```

String Concatenation

In JavaScript the addition operator can be used for string concatenation, which is still possible in the 7th edition of the standard, also denoted as ECMAScript 2016 [50]. In the sixth edition of ECMAScript template literals were introduced [51, p. 148, 17, 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 needs to be added as a string between the two variables. The same result can 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 back-ticks rather than quotes—where expressions can be inserted [17, p. 48]:

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

The result of all the previously shown concatenation techniques is identical.

Beyond ECMAScript 2015

The development of JavaScript is dependent on its specification, defined by ECMAScript, and new editions were not released regularly [86]. Version 5.1 was published in 2011 [52], from where it took four years until the sixth edition was published in June 2015 [49]. Starting with ECMAScript 2015, a new specification will be released yearly [53].

2.2.7 Further Reading

This section outlined the most important concepts of JavaScript with a focus on the characteristics that encourage the value of runtime type checks in JavaScript, discussed later in Ch. 4. Various exceptions or details were not handled, as they would go beyond the scope of this thesis. If a more sophisticated knowledge of the programming 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 [7, p. 99-19], containing only the indispensable portions of the code [20, p. 12] for the most parts. A syntax tree—or abstract syntax tree—is normally created by a compiler at an early stage. More specifically it is usually the second out of five compilation phases [7, pp. 99-2–99-3]:

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 TypeScript, a variable declaration was inserted in the online editor *AST explorer*¹¹, which can visualize the syntax tree generated by numerous parsers, including JavaScript, various JavaScript supersets such as TypeScript and Flow, as well as CSS (i.e., Cascading Style Sheets) and HTML (i.e., Hypertext Markup Language) [70]. The resulting syntax tree is illustrated in Fig. 2.2.

¹¹ <https://astexplorer.net>

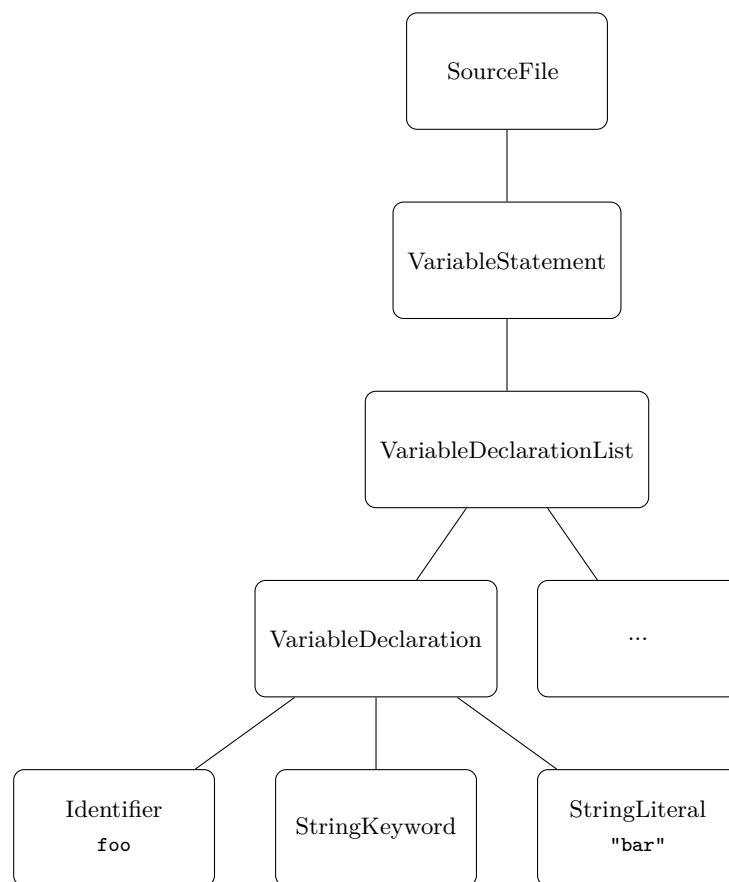


Figure 2.2: Abstract syntax tree of the TypeScript code `let foo: string = "bar".`

Chapter 3

State of the Art

This chapter’s main focus is on TypeScript, while also other supersets are explored. Furthermore, the current state of runtime type validations in JavaScript is highlighted, and the status of automatically generating type checks based on TypeScript code is discussed.

3.1 JavaScript Supersets

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 [...]. [101]

This means that every program that is valid in JavaScript is also legal in a JavaScript superset, where the purpose of such a superset can be 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 as well.

3.1.1 TypeScript

TypeScript was created by Anders Hejlsberg—the designer of C#—at Microsoft [14, p. 10] and was released in 2012 under the *Apache*¹ open source license version 2.0² [3, p. xix]. The most important aspect of TypeScript is that it includes a compilation step where static type checking is performed [14, p. 11]. Type annotations are optional and the compiler will infer type information where possible [87, 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 [14, p. 13]. The most significant particularities and features are explored in this section.

¹ <https://www.apache.org>

² <https://www.apache.org/licenses/LICENSE-2.0>

Basic Types

TypeScript defines a set of basic types, which overlap with JavaScript's types, listed in Sec. 2.2.2, while also introducing several new ones, as listed below [28]:

- *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 values.
- *Any* results in type checking not being performed by the compiler. This can be of advantage when using TypeScript alongside third party libraries where no type definitions are available.
- *Void* is the counterpart to *Any*. Again, it is used in other languages, e.g., to annotate functions that do not return a value. In TypeScript also variables may be typed as `void`, meaning that only `undefined` or `null` are 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.

While other types, such as *Function*, or more advanced structures are also available, the ones listed above in combination with those already defined in JavaScript (i.e., *Undefined*, *Null*, *Boolean*, *String*, *Symbol*, *Number*, and *Object*) are frequently seen in TypeScript projects.

Type Inference

As already mentioned, TypeScript tries to infer the type if no type annotation is provided. The example below shows a 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* is used over *error* here, since the compiler won't stop in such cases and will try to emit the final JavaScript code by default [87, p. 12]:

```
let num = 1;  
num = "foo";
```

The code above shows the compiled result, even though the compiler detected a type error.

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` is 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.

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. [28]

It is the developer who needs 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 type information of the external package may occur. Type assertions 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 example above the default export from the library `random-name` is 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 provide the information which type to assert. `RandomName` is casted to `any`, allowing property access independently of their existence. Again, because of type assertions (or type casts) not performing any special checking, this solution may lead to errors if the author of the package decides to change its *application programming interface* (i.e., API). Therefore manually checking for a callable `getName` property on `RandomName`, as well as verifying the returned value, is recommended. As an alternative to the type casting syntax with the `as` keyword, the following may be used:

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

However, the *angle-bracket* syntax shown above is not supported when using TypeScript with *JSX*³ [28], making the `as` syntax preferable.

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, which is not written in TypeScript, and no type definitions are available for the library. 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 an enum with the `declare` keyword, results in the declaration being ambient:

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

³ “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” [68].

Alternatively the imported package may be described more detailed:

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

From now on the TypeScript compiler can obtain the information of the import having a callable property `getName`, which returns a string. However, the declaration will not be part of the compiled JavaScript program.

Structural Types

Types in TypeScript are structural [87, p. 11], meaning that the type checker looks at the members of an object, or more specifically its type signature, to ensure type compatibility, while other major languages, such as C# or Java, use nominal type systems [94]. Prog. 3.1 gives an example, which would fail in a nominally typed language, but is possible in the structurally typed language TypeScript.

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. [94]

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

Classes

TypeScript not only enables static type checking for JavaScript applications, but it also adds language features. While EcmaScript 2015 introduced classes, TypeScript provides the enhancement to also define them as abstract, and to add visibility modifiers and interfaces 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 to provide default values outside of the constructor, as well as to mark properties as `readonly`, which prohibits reassignments at compile time:

```
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 missing and cannot be used in the running program:

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

Consequently, it is technically possible to assign an arbitrary value to `id` property of a `Person` instance at runtime.

Enums

Enumerations are beneficial for defining a set of values. The TypeScript compiler takes enum declarations and transforms them into runnable JavaScript code. Given is the following enum in TypeScript syntax:

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

This results in the JavaScript code below, which shows a self-executing function, initializing the identifier `HairColor` with the data of the enumeration:

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

The enum keys can now be accessed at runtime to obtain their corresponding values. Also it is possible to reveal a key by its value:

```
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 [54], unless the compiler option *preserveConstEnums* is used [38].

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 [75]. Code within a namespace only exposes its 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 is declared and an empty object is 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.

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 = console) {
  logger.log(message);
}
```

The example shows a 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 aligns with the `Console` interface. The compiled JavaScript code is shown below:

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

If the parameter `logger` equals `undefined`, which is the value that is returned by `void 0`⁴, the global variable `console` will be assigned to it. Otherwise the parameter passed to the function `log` will be used as is.

Future JavaScript

While the TypeScript compiler can target different JavaScript versions—such as ES3, ES5, and ES2015—it does also support future ECMAScript proposals, like decorators and asynchronous functions [38, 97], allowing the use of features which are possibly not yet implemented in various JavaScript engines. This is achieved by changing parts of the source, or by including additional code that mimics the behavior of a certain feature and delivers the same result. The code below uses a pattern, referred to as *destructuring assignment* [57], to assign the values 1 and 2 to the identifiers `foo` and `bar`, respectively:

```
let [ foo, bar ] = [1, 2];
```

⁴ The void operator can be used to retrieve the value `undefined` by calling `void(0)`, which is equivalent to `void 0` [100].

While this line would remain unchanged when targeting the ES2015 standard or later, where the *array binding pattern* is already specified [51, p. 198], the outcome is different for ES5 and below:

```
var _a = [1, 2], foo = _a[0], bar = _a[1];
```

As the pattern is not part of the fifth edition of ECMAScript [48], the compiler substitutes it with an alternative implementation.

3.1.2 Flow

*Flow*⁵ is an open source static type checker for JavaScript, developed by *Facebook*⁶ [58]. The most noticeable difference to TypeScript is the lack of an extensive compiler provided by the project itself. Instead, Flow relies on *Babel*⁷, a compiler for JavaScript [26], which “[...] will take [...] Flow code and strip out any type annotations” [65]. Alternatively the library *flow-remove-types*⁸ can be used [65].

Another difference between the two JavaScript supersets are their design goals. While TypeScript’s goal is not to “apply a sound or “provably correct” type system [but to] strike a balance between correctness and productivity” [37], Flow’s type system “[...] tries to be as sound and complete as possible” [96]. The syntax itself is mostly identical to the one of TypeScript [93]. Brzóška sums up the differences between the two languages, as shown in Tab. 3.1.

Table 3.1: Differences between TypeScript and Flow. [98]

	<i>TypeScript</i>	<i>Flow</i>
<i>Design Goal</i>	correctness and productivity	soundness and safety
<i>IDE Integrations</i>	top-notch	sketchy
<i>Autocompletion</i>	yes	unreliable
<i>Speed</i>	stable	degrades
<i>Generic Definitions</i>	yes	yes
<i>Generic Calls</i>	yes	no
<i>Library Typings</i>	many	few

3.1.3 Others

Apart from TypeScript and Flow, there are a variety of other languages that compile to JavaScript for different purposes. In [71] an extensive list of JavaScript supersets, parsers, and compilers can be found, containing the following maintained languages they refer to as superset:

- **JavaScript++:** This superset supports classes, type checking, and other features.

⁵ <https://flow.org>

⁶ <https://code.facebook.com>

⁷ <https://babeljs.io>

⁸ <https://github.com/flowtype/flow-remove-types>

- **Objective-J:** This language has the same relationship to JavaScript, as *Objective-C* to *C*.
- **JSX:** JSX got popular with the *React*⁹ framework [68] and adds XML-like syntax to represent HTML elements in JavaScript.
- **oj:** This is an *Objective-C* inspired superset with an experimental type checker.

The collection also contains languages like *Scala.js*¹⁰, which compiles *Scala*¹¹ code to JavaScript, or *Opal*¹², a *Ruby*¹³ to JavaScript compiler.

3.2 Runtime Type Checks

Type annotations are removed for the compiled JavaScript program in TypeScript and no additional code is introduced to add checks at runtime. The removal of types is intended and is defined in the design goals¹⁴ of the language:

[Do not] add or rely on runtime type information in programs, or emit different code based on the results of the type system. Instead, encourage programming patterns that do not require runtime metadata. [37]

Anyway, runtime type information and validation can be useful in several situations. For example they can give more accurate error messages during development and can draw attention to issues, which are not observable during compile time. There are proposals to expose type information to the runtime and to add runtime type checks, in the TypeScript community [61, 72, 83]. Regardless of the demand, these features won't be added, as they are out of scope for TypeScript [36, 63]. Currently, manually added type checks are required to identify and to easily trace errors during development. Prog. 3.2 shows a JavaScript function, which only takes up three lines of code, while Prog. 3.3 outlines a function—also in native JavaScript—with the same outcome but with added type checks, which now requires 13 lines of code.

Program 3.2: A JavaScript function without type checks.

```
1 function sum(arr) {
2   return arr.reduce((a, b) => a + b);
3 }
```

While these examples outline the verification of primitive types, like a number or an array, inspecting an object is more complex. Instances may be checked with the `instanceof` operator, which “[...] tests whether an object in its prototype chain has the prototype property of a constructor” [66], however, interfaces and type alias are removed by the TypeScript compiler, therefore this kind of verification method is not possible for such cases.

⁹ <https://facebook.github.io/react/>

¹⁰ <http://www.scala-js.org>

¹¹ <http://scala-lang.org>

¹² <http://opalrb.org>

¹³ <https://www.ruby-lang.org>

¹⁴ <https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals>

Program 3.3: The JavaScript function from Prog. 3.2 with type checks.

```
1 function sum(arr) {  
2   if (!Array.isArray(arr)) {  
3     throw new TypeError("array expected");  
4   }  
5  
6   return arr.reduce((a, b) => {  
7     if (typeof b !== "number") {  
8       throw new TypeError("number expected");  
9     }  
10  
11     return a + b;  
12   });  
13 }
```

To get around this issue, Rozentals describes three different techniques to employ type checks for the runtime environment:

- **Reflection:** The prototype of a JavaScript object holds some information about the object, which can be accessed. It might, for instance, contain the name of the constructor function, used to create the object. Limitations apply, since various information is only available from ECMAScript 5.1, or may not be available at all [13, pp. 98–100]. Also the name of a constructor is not always suitable to categorize an object as a type, as the same name may also be used for a different constructor function, while anonymous functions do not have a name at all. Simply obtaining the name is also not sufficient to check for implemented interfaces, or type aliases, as they are compiled away by TypeScript.
- **Checking an object for a property:** An object could be considered as being of a type, if specified properties exist on it [13, pp. 101–102, 6, pp. 18–20]. If, for example, a constructor function *Person* is given, which defines a *getName* property on its prototype, an arbitrary object could be considered as *Person*, if it also provides a *getName* property. This gets already closer to TypeScript’s structural types (see Sec. 3.1.1).
- **Interface checking with generics:** This concept requires the definition of a class for every interface, which holds the property names to identify an object as having a specific type [13, pp. 102–105, 6, pp. 17–19]. This solution is similar to the previous approach, but it introduces a pattern, which is more readable and maintainable.

Another mechanism is to use *decorators*¹⁵, a JavaScript language feature proposal, which is currently at stage two [69], meaning that it is still a draft and not yet in the specification [91]. They can, however, already be used with TypeScript or tools like Babel [44, 45]. The solution used in [84], which makes use of decorators, again requires to add them to the source code manually. Furthermore, only primitive types and instances can be checked automatically. Structural type checks—e.g., for custom objects or interfaces—have to be provided by the developer.

¹⁵ <https://tc39.github.io/proposal-decorators>

Program 3.4: The following code overwrites the default `instanceof` behavior for the given class. [81, 82]

```
1 class PrimitiveNumber {
2   static [Symbol.hasInstance](x) {
3     return typeof x === "number";
4   }
5 }
6
7 123 instanceof PrimitiveNumber; // true
```

Program 3.5: The ECMAScript proposal for pattern matching¹⁶ would add a sophisticated validation pattern in JavaScript. [81, 88]

```
1 match (obj) {
2   { x }: /* match an object with x */,
3   { x, ... y }: /* match an object with x, stuff any remaining properties in y */,
4   { x: [] }: /* match an object with an x property that is an empty array */,
5   { x: 0, y: 0 }: /* match an object with x and y properties of 0 */
6 }
```

Program 3.6: The code below shows an ECMAScript proposal for `Builtin.is` and `Builtin.typeOf`¹⁷, where the former command determines if two values point to the same built-in constructor, and the latter can obtain the type of primitive and built-in values, in contrast to the existing `typeof` operator which can get the primitive type only. [81, 89]

```
1 Builtin.is(Date, Date); // true
2
3 class MyArray extends Array { }
4 Builtin.typeOf(new MyArray()); // "Array"
```

As runtime type checks are of importance for an application to be robust, the operators `typeof` and `instanceof` are often used to verify a value's type, which according to Rauschmayer is "[...] less than ideal, because [it requires] to keep the difference between primitive values and objects in mind" [81]. Prog. 3.4 shows a technique to enable `instanceof` checks also for primitive values, such as strings. He further refers to using a library for checking types at runtime, and outlines two ECMAScript proposals that are related to runtime validations [81], which are shown in Prog. 3.5 and Prog. 3.6.

¹⁶ <https://github.com/tc39/proposal-pattern-matching>

¹⁷ <https://github.com/jasnell/proposal-istypes>

3.3 Generated Runtime Type Checks

During research, no libraries could be found which automatically generate runtime type checks from TypeScript code, and validations have to be implemented manually, as described in the previous section. However, there are libraries which aim to provide a runtime type system, which are explored in Sec. 5.1.2. While those packages are supportive in describing and validating data structures in JavaScript, few projects concentrate on automatically generating them. As it may not be feasible to create checks without the data provided by a static type system or some kind of supportive information—such as type annotations—also no libraries could be discovered which can provide runtime validations from native JavaScript code. However, the Babel plugins *babel-plugin-tcomb*¹⁸ and *babel-plugin-flow-runtime*¹⁹ can generate runtime validations for Flow syntax [33, 76]. Furthermore, a future release of Babel will support TypeScript syntax [27], which could make it possible to adapt the plugins to also transform TypeScript code.

¹⁸ <https://github.com/gcanti/babel-plugin-tcomb>

¹⁹ <https://github.com/codemix/flow-runtime/tree/master/packages/babel-plugin-flow-runtime>

Chapter 4

Theoretical Approach

After defining the terminology for this thesis, as well as giving an overview of available technologies and projects, the theoretical approach for generating runtime type checks from TypeScript type annotations for runnable JavaScript code is described in this chapter. The project of this thesis will further be referred to as *ts-runtime* (i.e., *typescript-runtime*) or *tsr*.

4.1 Undetectable Errors

There are various situations where the static type analysis of TypeScript cannot detect conditions that may lead to errors at runtime. Either a project is written in TypeScript, and the compiler can infer the type information needed, or type definition files are 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 already being trapped 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 Sec. 3.1.1. 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, e.g., a variable is annotated or asserted with the *Any* type, type checking and type inference will be disabled for this part of the code. The TypeScript documentation describes this as

[...] a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type-checking during compilation. [28]

It seems legitimate to use `any` alongside 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 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 a *TypeError* exception, stating that *foo.getNumber is not a function*. The examples above highlight the potentiality of creating conditions where a detectable mistake remains undiscovered by the compiler, which can cause a running program to be interrupted.

4.1.2 Polymorphism

Polymorphism can lead to errors at runtime in combination with type assertions. While the TypeScript compiler does check type compatibility in general, it allows to assert identifiers as types that could be assigned to it. To give an example of such a situation the following code is given:

```
class Animal { }

class Cat extends Animal {
  miow() {
    return "Miow";
  }
}

class Dog extends Animal {
  woof() {
    return "Woof";
  }
}
```

As a next step an instance of `Dog` is created and assigned to a variable, which is typed as `Animal`, as shown below:

```
let dog: Animal = new Dog();
```

In order to call the method `woof` on the `Dog` instance, it needs to be asserted as follows:

```
(dog as Dog).woof();
```

The TypeScript compiler does not raise any concern, as type `Dog` is assignable to `Animal` and therefore is allowing the cast. Subsequently, also the following type assertion passes without any compiler errors:

```
(dog as Cat).miow();
```

While static type checks are successful, the compiled JavaScript code will fail at runtime. As no method `miow` exists on the `dog` object, a *TypeError* exception with the message *dog.miow is not a function* will be thrown at runtime.

4.1.3 Untyped JavaScript Libraries

If a JavaScript project is written in native syntax, TypeScript cannot infer the type information needed to perform sufficient static type analysis. In this cases type declarations may be provided manually, as discussed in the following section. *DefinitelyTyped*¹ provides a collection of such type definitions for JavaScript libraries [90]. Anyway, not all packages have definitions available, especially small projects. A practice often used in such a situation is to declare the package name, to stop complaints from the TypeScript compiler about not finding the import:

```
declare module "MyModule"
```

After declaring the module it effectively has the *Any* type applied to it. Therefore, as already mentioned in the previous section, it is possible to access any property on the imported module, regardless of whether it exists or not. Also changes to the package's API won't be noticed, and a project depending on the module may break after updating its dependencies.

4.1.4 Type Declaration Mistakes

Libraries written in TypeScript are usually published as compiled JavaScript code alongside type declaration files with the extension *d.ts*. These files include all type information, which was removed for the runnable JavaScript code. The compiler can parse the definitions and can statically check the correct usage of the library, when imported in another project. If the definitions are generated during compilation, they can be considered relatively safe to use, unless the *Any* type is misused. If, however, declaration files are created manually for a JavaScript library which is not written in TypeScript, there is a chance that they contain mistakes or that they are not up-to-date with the implementation. A JavaScript file *foo.js* may contain the following code:

```
class Foo {  
  getName(): string {  
    return "Foo";  
  }  
}
```

Its corresponding declaration file *foo.d.ts* may provide the declaration as shown below:

```
declare class Foo {  
  getNumber(): number;  
}
```

The file containing the type declaration for class `Foo` does not reflect the actual implementation. The method `getNumber`, as suggested by the type definition, does not exist on the class. Code completion in an IDE (i.e., integrated development environment) may suggest to use this method, which can lead to a runtime exception. Also the static type checker would complain if attempting to use the implemented method `getName`, as it has no knowledge of its existence on the object.

¹ <http://definitelytyped.org>

4.1.5 Erroneous API Responses

When making use of an external API, the response should be of a given structure, which should be known to the consuming program. An interface may be created to describe the receiving object and to statically check its usage. However, if the format of the response changes unexpectedly, it is not possible to reveal this change at compile time, since the type checker relies on the interface provided. Runtime checks have to be added manually to ensure that the response conforms to the expected structure. Otherwise an error may be raised during program execution, or its behavior may be different than expected.

4.2 Desired Result

The situations discussed in the previous section of this chapter raised the concern of negligently or unknowingly opting out of static type checks for specific parts of the code, or providing insufficient or wrong type declaration, which may cause the compiler to miss incompatible types. These situations may be discoverable at runtime, if runtime type checks would be employed alongside compile time checks. Information about types are already available in a TypeScript development environment. Either it is provided through explicit type annotations, or the compiler tries to infer the type, which leads to the assumption that this metadata can be used to reflect types and generate representations for the runnable JavaScript code. This representations may then be used to verify if an object conforms to a type based on its structure. Below is a type alias declaration of its simplest form in TypeScript:

```
type MyString = string;
let foo: MyString = 10 as any;
```

This code compiles without any errors, since the number is assigned as *Any* type to a variable, which should only accept strings. In the resulting JavaScript program the type alias, as well as the type assertion, is removed. A concept to keep this declaration also for the compiled code is shown in the following code snippet:

```
const MyString = reflect("string");
let foo = 10;
```

In this case, the name of the type alias is used as identifier for a variable declaration. Furthermore, the name of the type is used to pass it as a string to a method, which should return a reflection for it. This still results in a number being assigned to a variable, which should only accept strings. To catch this type incompatibility, the final JavaScript code could check the value that should be assigned to `foo`:

```
const MyString = reflect("string");
let foo = MyString.accepts(10);
```

Before the number is assigned, it should be passed to the type representation of `MyString`, which should check if the received value is compatible. In case of a violation, the program should report an error.

4.3 Definition of Cases

The current situation for TypeScript projects, as discussed in Sec. 3.3, observed that runtime type checks cannot be generated automatically at this time. Additional effort is required to integrate code safety features for the compiled program. This means that situations may be missed where checks would be of advantage. In order to achieve a development environment where as many undetectable errors (see Sec. 4.1) as possible are reported during execution time, it may be beneficial to automate the inclusion of runtime validations. In order to provide generated runtime type checks for a TypeScript project, cases have to be collected where such verifications have to be performed.

4.3.1 Interfaces and Type Aliases

Interface and type alias declarations are removed by the TypeScript compiler and therefore need to be described for the runtime environment. The name of the given type definition should be used to declare a variable, holding all required information to check any value for conformance.

4.3.2 Variable Declarations and Assignments

If a variable was declared it also has a type bound to it during compile time. This type should be used to declare another variable alongside the original declaration, containing the type description or reference. When assigning a value to a variable, type compatibility should be checked by using the type description declaration.

4.3.3 Type Assertions

Type assertions are comparable to type casts in other languages, with the difference that no special checks or conversions are performed (see Sec. 3.1.1). To inspect if an assertion is valid, the same checks should be performed as for variable assignments. Values asserted as *any*, as discussed in Sec. 4.1, can be ignored, as they would always pass.

4.3.4 Functions

There are different types of functions, which need to be distinguished: *function declarations*, *function expressions*, and *arrow functions* (see Sec. 2.2.6). For any of these types the function has to be reflected to enable type comparison. The runtime description has to include its parameters—which can also be optional—and its return type. If the function is called, the parameters passed, as well as the returned value, have to be checked. Additionally, a function can make use of generics to define parameter or return types [60], as shown below:

```
function foo<T>(bar: T): T {  
    return bar;  
}
```

Whatever type the parameter `bar`—passed to function `foo`—has, the returned value must be of the same type, as both are annotated with the generic type parameter `T`.

Program 4.1: The enum `MyEnum { A }` compiled to JavaScript. [54]

```
1 var MyEnum;  
2 (function (MyEnum) {  
3     MyEnum[MyEnum["A"] = 0] = "A";  
4 })(MyEnum || (MyEnum = {}));
```

4.3.5 Enums

Enumerations (see Sec. 3.1.1) are compiled to self executing functions, which initialize a corresponding object [54] (see Prog. 4.1). To enable type checks for the runtime, the enum has to be described with its members.

4.3.6 Classes

For classes a multitude of cases requiring runtime reflection and checks have to be considered. Most importantly the entire class—including type parameters (i.e., generics), its members, extending classes, and implemented interfaces—has to be reflected to use it as type reference at other places of the program. Methods can be checked the same way as functions, with the difference that they may also use class type parameters as type annotations. Furthermore, when instantiating a class sufficient checks should be performed to ensure that it correctly implements its interfaces.

4.3.7 Type Queries

It is possible to use a value's type as type annotation in TypeScript, which looks like the following:

```
let foo: string = "Bar";  
type MyType = typeof foo;
```

In this case `MyType` is of type *String*, since TypeScript's `typeof` operator is not to be confused with JavaScript's built in operator of the same name. In TypeScript it is possible to query the type of any identifier, if not attempting to reuse it as a value. In JavaScript, on the other hand, a type query result may be used as value, while it can distinguish between six value types at runtime (see Tab. 2.1). If a variable is annotated with a type query, the type of this variable should also be obtainable at runtime.

4.3.8 Externals

JavaScript programs usually make use of other libraries, which are imported alongside other project code. If those packages are written in TypeScript, or provide type declaration files, the compiler can use the type information to perform compile time checks. However, as types are also removed from external projects, their interfaces, type aliases, class reflections, etc., have to be collected and have to be made available to the runtime code.

4.3.9 Ambient Declarations

If globals are not available in the development environment of a project, but it is known that they will be present in the environment of execution, modules, classes, functions, and variables can be declared for the compiler without an implementation:

```
declare function foo(bar: number): string;
```

After the function `foo` has been declared as shown above, it can be used according to its signature throughout the project, but it will be removed for the compiled code. Such declarations should be collected and should be made available to the runtime the same way as externals.

4.4 Required Steps

After situations of undetectable errors (see Sec. 4.1) have been clarified, the desired result of the project (see Sec. 4.2) has been outlined, and conditions where transformations should take place (see Sec. 4.3) have been pointed out, the steps required to accomplish automated runtime type checks are specified:

1. Set the configurations for the transformation process.
2. Read the source files of a TypeScript project.
3. Analyze the source code provided.
4. Represent the input as an abstract data structure.
5. Scan the abstraction to obtain type information and relationships.
6. Perform static type analysis and checks.
7. Insert runtime type reflections and assertions.
8. Emit target code for the JavaScript runtime engine.

These steps are described in more detail in the following sections. While giving a theoretical understanding of the concept of the thesis project, no technical details are provided at this point.

4.4.1 Configuration

As different projects have different requirements regarding the result of the JavaScript target code, configurations for the transformation process have to be set in advance. This includes the settings for the TypeScript compiler² itself, as well as adjustments for *ts-runtime*. While the project of this thesis is not intended to be a replacement of the TypeScript compiler, it should still honor the options of the development environment. These settings include, among others, the ECMAScript version of the resulting program, the module system to use, as well as the write location of the output [38].

4.4.2 Read Source Files

The starting point of the transformation process should be an existing project. As for a usual TypeScript compilation process, the entry files should be passed to *ts-runtime*, alongside a set of configurations. All files that are referenced or imported throughout the

² <https://www.typescriptlang.org/docs/handbook/compiler-options.html>

project should be loaded recursively, resulting in a reflection of the project's file system structure. This should enable further steps to interact with the input in memory, leaving the original files untouched.

4.4.3 Syntax Analyzation

After all contents of the project are available it should be determined, if the provided code is syntactically correct. This should prevent *ts-runtime* to fail, due to syntax errors in the source. If syntactic errors are detected the process should be stopped immediately to prevent the occurrence of unexpected behavior or results.

4.4.4 Abstraction

In order to perform special checks and transformations to the original code, abstracting the source is beneficial. A suitable data structure may be an abstract syntax tree (i.e., AST), as described in Sec. 2.3. Performing modifications on the input directly via string modifications is much more error prone, and semantic connections between parts of the code cannot be extracted easily.

4.4.5 Scan Abstraction

Once the provided source files can be considered syntactically correct and are represented in an abstract data structure, the type information has to be extracted for future modifications to the code. It should not only be possible to obtain the explicitly set type of an AST node, but to also receive the implicitly inferred type. In addition it should be practicable to follow a type reference's type, for further processing. To ensure that important data—e.g., type information, AST node relations, and declared identifiers—is not becoming inaccessible during the transformations, the abstract syntax tree may be scanned ahead of changing its nodes.

4.4.6 Static Type Checks

Another important aspect is to already perform static type checks, and to reject the input from further processing if type incompatibilities can be detected, which has the advantage of flagging issues to developers early. Also, if the static type analysis can already find possible violations, the target code may not behave correctly. Anyway, as it is possible to provide incorrect type declarations for accurate implementations, hence there should be the possibility to force the process to proceed and to solely rely on type compatibility checks at runtime. Warnings should be generated at compile time in this case, to clearly indicate that unexpected results may be a consequence.

4.4.7 Transformations

Situations where modifications have to take place to reflect all required type information, as well as to introduce runtime type checks based on these reflections, have already been identified in Sec. 4.3. All required data to perform extensive transformations on the AST should have already been prepared by the previous steps of the process. This should make it possible for *ts-runtime* to proceed with substituting and altering abstract

syntax tree nodes. The tree should be scanned from the bottom to the top, which should guarantee that transformations of high level tree nodes already include low level mutations. Furthermore, a node that is replaced or changed should be mapped to the original syntax tree node to assure that its initial state can be retrieved at any time.

4.4.8 Target Code Emit

After the transformations have been applied to the syntax tree, it should be converted to TypeScript compatible syntax code. In a next step this code can then be used to emit runnable JavaScript, according to the options that were passed when initially triggering the generation of runtime type checks (see Sec. 4.4.1). In case of inconsistencies or faults, appropriate warnings and errors should be triggered.

4.5 Summary

To achieve automatically generated runtime type reflections and checks, a series of steps have to be carried out. To give a better understanding of the conceptual procedure, Fig. 4.1 illustrates the idea of the transformation process. In the following chapter the theoretical approach is evaluated, and the project's technology and architecture is defined. Also technical peculiarities and limitations are identified to provide a solid base for the implementation.

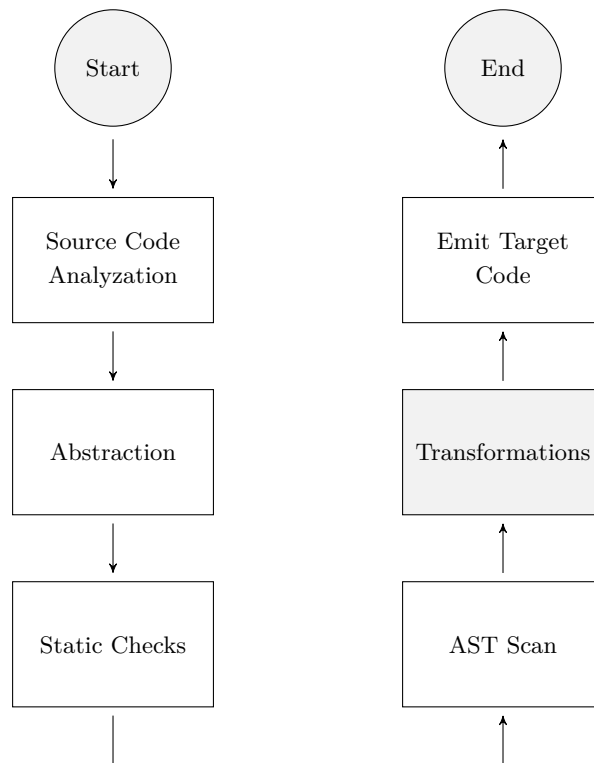


Figure 4.1: Conceptual procedure of applying transformations to a TypeScript project.

Chapter 5

Implementation

After elaborating situations where runtime errors—even with preceding static type checks by the TypeScript compiler—may occur (see Sec. 4.1), a program is implemented which should catch those situations during the execution of the compiled JavaScript program. All previously defined cases (see Sec. 4.3) should be honored and suitable technology should be selected to perform the required steps (see Sec. 4.4) to achieve the desired result (see Sec. 4.2).

5.1 Technology

The project itself is implemented in TypeScript, while the compiled program is executed in a JavaScript—usually Node.js—environment. It is published on the *npm* (i.e., node package manager) registry¹, a “[...] public collection of packages of open-source code for Node.js [...]” [22], which should make it easy for developers to install an executable version of *ts-runtime* on their system. Also other packages should be able to integrate with this project as fast as possible. This implies that both, an API (i.e., application programming interface) and a CLI (i.e., command line interface) is provided. Furthermore, to create an application that efficiently achieves its goals, it is important to choose appropriate tools and libraries. This includes the process of generating runtime type checks itself, as well as reflecting and checking type compatibility in the final JavaScript code. If trusted and established technology is available, which provides functionality that is needed for the implementation, it is utilized to decrease development and maintenance effort and to increase the quality of the resulting project.

5.1.1 TypeScript Compiler

The TypeScript compiler exposes an API to use its functionality programmatically. This makes it possible to read in an existing TypeScript project, perform static type checks on it, and to emit a compiled JavaScript program, while having control over various aspects of this process. Several steps that are required to generate type checks for the JavaScript runtime are provided by the TypeScript compiler. With version 2.3 an API was exposed to enable abstract syntax tree transformations [29] and an issue preventing

¹ <https://www.npmjs.com>

traversing the AST [78] was resolved with version 2.4 [30]. Not only the ability to modify the syntax tree is useful for the project of this thesis, also other features are beneficial. As *ts-runtime* makes use of the compiler API later in this chapter, some parts of it are described in the following sections.

Compiler Components

To receive a runnable JavaScript program from a TypeScript project, a number of components contribute to the TypeScript compiler [87, p. 251]:

- **Scanner:** The scanner is responsible for the tokenization of the source code and is controlled by the parser [87, p. 260].
- **Parser:** After a source file is tokenized, the parser creates an abstract syntax tree out of it [87, p. 263].
- **Binder:** In this part of the compiler, connections between nodes of the AST are created through *symbols* [87, p. 267].
- **Checker:** The checker is the largest part of the TypeScript compiler and performs static type checks on the source files [87, p. 282].
- **Emitter:** The emitter translates the TypeScript syntax tree of the source files to plain JavaScript [87, p. 286], based on the compiler options.

These components do not have to be triggered individually when using the compiler API, since a wrapper is provided, called *Program*. It holds the options and source files of the current compilation [87, p. 254], and provides access to the *Checker* [87, p. 282] and *Emitter* [87, p. 286].

Compiler Options

When starting a compilation through the TypeScript compiler API, a multitude of options [38] may be passed. They include, but are not limited to, settings for the type checking behavior, files that should be emitted, and the ECMAScript standard the resulting JavaScript code should comply to.

Program

A TypeScript project compilation can be triggered by providing the path to one or more entry files, alongside customized compiler options. All files that are referenced from the input files are loaded recursively, by making use of a *compiler host*. Also it exposes the functionality to emit the compiled JavaScript code.

Compiler Host

The compiler host abstracts, among other things, the reading and writing of input files by the *Program*. By default, files will be accessed on the file system, however, a custom compiler host may be provided.

Node

The abstract syntax tree, which is created during the compilation of a TypeScript project, consists of nodes, while every node has a specific kind. A file, for example, is of kind *SourceFile*, whereas a class declaration is represented by a node with the kind *ClassDeclaration*.

Syntax Kind

The TypeScript API exposes an enumeration, named *SyntaxKind*, which maps a numeric value to an AST node type (e.g., *InterfaceDeclaration*). As every syntax tree node defines a *kind* property, containing a number from the *SyntaxKind* enum, it is possible to always determine the type of a node.

Symbol

The syntax tree abstracts a source file to interact with it in various ways, but it lacks relations between nodes that are not directly connected to each other. Symbols are created to provide the relationships between such nodes. While it is possible to identify a type reference through the AST, there is no link to the declaration of the referenced type. However, by extracting the symbol of the type reference, the node of the type declaration can be obtained.

Printer

The compiler API exposes a printer, which can create text out of an AST node recursively. Consequently, it is possible to pass a *SourceFile* node to the printer and to get back a string containing TypeScript code.

5.1.2 Runtime Type System

A multitude of libraries are available, which aim to provide a runtime type system for JavaScript, while several of them are evaluated to use with *ts-runtime* in the following sections. Unmaintained libraries are not considered, since issues may not be fixed when discovered.

ObjectModel

*ObjectModel*² is an extensive type system, which “[...] intends to bring strong dynamic type checking to [JavaScript] web applications” [80]. While being actively maintained and a detailed documentation is available, this library makes use of a technique that requires the replacement of parts of the JavaScript code—e.g., object literals—to perform validations on them [80], which makes it not entirely suitable for the use with the project of this thesis.

² <https://github.com/sylvainpolletvillard/ObjectModel>

tcomb

The project *tcomb*³ argues to “[...] check the types of JavaScript values at runtime” [35]. While probably being one of the most famous runtime type checking libraries for JavaScript—with more than 1300 stars on GitHub [35]—it is again intended to be used for JavaScript code. Special considerations for TypeScript are not part of this package.

io-ts

Created by Giulio Canti, the author of *tcomb*, this project claims to be a “TypeScript compatible runtime type system [...]” [34]. While it did look promising to be used, some aspects did not meet the expectations. For example being able to define the type reflection of a class alongside the class declaration itself is not provided by the library, as well as being able to retrieve a type reference with type parameters (i.e., generics) is not possible. However, as *io-ts*⁴ may evolve over time, a transition of *ts-runtime* to use it at a later point is possible.

runtypes

The *runtypes*⁵ library is a fairly young project, which intends to provide “runtime validation for static types” [42]. Anyway, only basic validations can be performed, compared to more comprehensive systems such as *tcomb* or *io-ts*. For example when asserting a value for being a function, the built in JavaScript `typeof` operator (see Sec. 2.2.2) is used, which cannot compare the function’s signature, including parameters and the return type.

flow-runtime

The *flow-runtime*⁶ project states to be “a runtime type system for JavaScript with full Flow compatibility” [77]. As Flow and TypeScript have a lot of similarities in syntax and features, this library seems to be most suitable to reflect the static type system of TypeScript as close as possible. Additionally, *flow-runtime* provides a package which generates type checks for Flow projects [76], indicating that a multitude of cases for Flow syntax, and therefore also for TypeScript, are implemented in this library.

5.2 Architecture

In this section the architecture for the application is designed, which already outlines how the program operates, and also defines some of the components that are required. This should give an overview of the different parts of the program, before describing them more detailed later in this chapter.

³ <https://github.com/gcanti/tcomb>

⁴ <https://github.com/gcanti/io-ts>

⁵ <https://github.com/pelotom/runtypes>

⁶ <https://github.com/codemix/flow-runtime>

5.2.1 Central Element

The transformation process is a sequential process, as defined in Sec. 4.4, which already suggests that a central element is needed, coordinating all the different steps that need to be executed. It is responsible for interpreting and triggering specific application logic in the appropriate situations. Before being able to initiate the actual program flow, this crucial part of the project has to interpret different settings, including options for the TypeScript compiler. Also it has to react to possible errors and has to handle them adequately.

5.2.2 Components

Specific tasks are handed over to dedicated components, which contain the logic for a selected purpose to keep the project extensible and maintainable.

Options

It is beneficial to control the behavior of *ts-runtime* when initiating a transformation process. While the program provides sensible defaults, it is possible to optionally overwrite these default settings by passing the desired options to the application.

Event Bus

Some of the components of the application have access to other components and their API, whereas other parts of the program do not know the state of the transformation process. However, it is necessary to observe, or to get notified, if a condition changes, where an event bus is of advantage. Consequently, the event bus (i.e., bus) is accessible globally.

Scanner

Not to confuse with the scanner of the TypeScript compiler (see Sec. 5.1.1), this component of the thesis project scans the abstract tree of the source files. Ambient and external declarations are identified, which would not be included in the compiled program, but need to be reflected in order to guarantee that type checks can take place during runtime. Also identifier names across all source files are stored to avoid duplicate identifiers when introducing new variables during the insertion of runtime type checks.

Mutators

For every situation where runtime type checks are generated (see Sec. 4.3), a mutator exists, which performs the modification or substitution of the AST node.

Factory

To avoid code duplication, the factory provides a collection of common transformations, which are performed on syntax tree nodes. It is utilized by the mutators to keep their footprint as small as possible and to reduce code complexity.

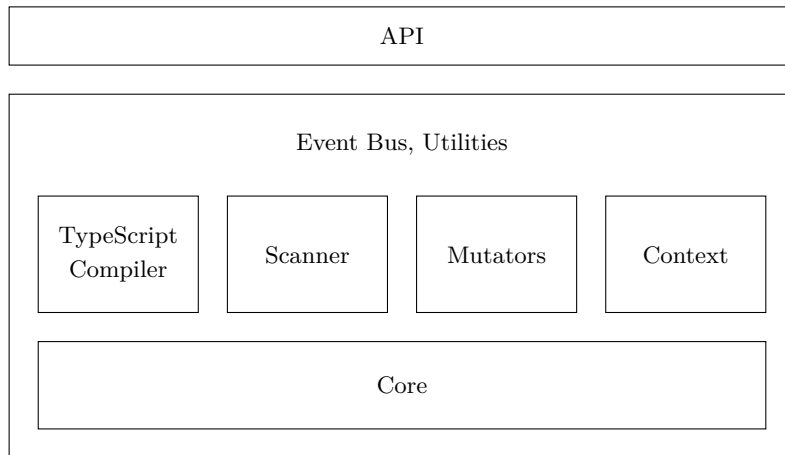


Figure 5.1: Component architecture of the thesis project.

Context

As not all components of the application are connected to each other, the context provides a centralized gateway to information, which is required by the mutators or the factory. It has, among other things, knowledge of the current source file being processed, the options of the application, and the TypeScript compiler program.

Utilities

Miscellaneous functionality that does not require any link to the state of the program is collected in the utilities of *ts-runtime*, which is available from any location of the project.

5.2.3 Outline

After the main parts of the program have been defined, it is possible to draw connections between them (see Fig. 5.1). As already stated, the central element (i.e., core) of the application controls the program flow, therefore having knowledge and access to all components of *ts-runtime*. It evaluates the options, creates a TypeScript compiler program (see Sec. 5.1.1), and triggers the scanning and transforming of the syntax tree, before emitting a compiled JavaScript project with inserted runtime type checks.

5.3 Application Structure

The following directory structure is used for *ts-runtime*, which at the same time shows the most important files and folders of the project:

```

/src
├── bin ..... Command Line Interface
├── lib ..... Runtime Type Checking Library
├── mutators ..... AST Node Transformers
├── bus.ts ..... Event Bus
├── context.ts ..... Mutation Context
├── index.ts ..... API Exposure
├── factory.ts ..... Common AST Node Transformations
├── options.ts ..... Default Options
├── scanner.ts ..... AST Scanner
├── transform.ts ..... Application Core
└── util.ts ..... Miscellaneous Utilities

```

5.4 Components

In this section the implementation of the core of *ts-runtime*, as well as of the application's components, is described, and connections between different parts of the program are already drawn.

5.4.1 Transformer

The transformer—located in `src/transformer.ts`—is the core of the thesis project and exposes three methods, which may be used via the project's API:

- **getOptions:** This function accepts an object as parameter which aligns with the *Options* interface, described later in this section. It then merges the passed object with the default settings, and returns the result. This ensures that all required options are contained in the resulting object.
- **transform:** By calling this method a transformation process is initiated. It is required to pass at least a list of entry file names. Optionally, an *Options* object may be passed. The compiled JavaScript files are written to disk, according to the TypeScript compiler options, if no errors occurred.
- **transformReflection:** The `transform` function loads the list of entry files from disk, which requires a file system to be present. On the contrary, this method accepts a list of file reflections, which must include the entry files, as well as all modules referenced, recursively. This enables the application to act without a file system. Also the target code is not persisted, but a list of file reflections, containing the compilation result, is returned.

Prog. 5.1 shows a simplified version of the `transform` function. It is not fully functional, but should give an idea of the program flow. On line 6, a variable `transformer` is passed to a function from the TypeScript compiler API, which references a function that visits every node of the AST of all source files from the TypeScript program.

Program 5.1: The `transform` function of the project's core, reduced to its essentials. The `ts` namespace from line 3 and 6 point to the TypeScript compiler API.

```
1 function transform(entryFiles: string[], options?: Options) {  
2   const opts = getOptions(options);  
3   const program = ts.createProgram(entryFiles, opts.compilerOptions);  
4   const scanner = new Scanner(program, opts);  
5   const files = program.getSourceFiles();  
6   const result = ts.transform(files, [transformer], opts.compilerOptions);  
7   emit(result);  
8 }
```

Program 5.2: An exemplary version of the transformer, that visits all nodes of a TypeScript program and triggers the transformations.

```
1 function transformer() {  
2   let context: MutationContext;  
3  
4   const visitor = node => {  
5     node = mutate(node, context);  
6     return ts.visitEachChild(node);  
7   }  
8  
9   return sourceFile => {  
10    context = createContext(sourceFile);  
11    return ts.visitNode(sourceFile, visitor);  
12  };  
13 }
```

The function code, again reduced to its essentials, is shown in Prog. 5.2, while line 5 indicates the AST node being passed to the mutators of *ts-runtime*, possibly returning a substitution. Technically, the abstract syntax tree node's children are followed to the very bottom before applying mutations on them. This should assure, that every transformation already includes modifications from its child nodes.

5.4.2 Mutators

Every mutator of the project extends a base mutator, which provides a simplified API that is used by the core (i.e., transformer) of the project. Therefore, each mutator must cohere with its base, which in its simplest form may look like the code below:

```
class InterfaceMutator extends Mutator {  
  
  protected kind = ts.SyntaxKind.InterfaceDeclaration;  
  
  protected mutate(node: ts.InterfaceDeclaration): ts.Node {  
    return node;  
  }  
  
}
```

A valid mutator must define a `kind` property containing a syntax kind—or an array of syntax kinds—to define which node types the mutator is able to process. Additionally, the method `mutate` must exist on a mutator. This function accepts a single parameter, which is the node to be processed. The mutator may then perform modifications on it, or can replace the node entirely. Each mutator is meant to be used through the method `mutateNode`, defined in the base class. This ensures that the following checks are performed to discover if the node should be processed:

1. Is the kind of the node supported by the mutator?
2. Is the node flagged to be skipped?
3. Is the node declared ambient, using the `declare` keyword?

If all of these checks pass the actual `mutate` method is called. If, however, any of the conditions above cannot be met, the original node is returned untouched. Based on the defined cases from Sec. 4.3, the following mutators are implemented, located in `src/mutators`, also showing their file names, while omitting the extension `.ts`:

- `ArrowFunctionMutator`,
- `AsExpressionMutator`,
- `BinaryExpressionMutator`,
- `BlockLikeMutator`,
- `ClassDeclarationMutator`,
- `FunctionDeclarationMutator`,
- `FunctionExpressionMutator`,
- `InterfaceDeclarationMutator`,
- `SourceFileMutator`,
- `TypeAliasDeclarationMutator`,
- `VariableDeclarationListMutator`.

Some of the implementations are more complex than others and special cases had to be taken into account. While every mutator is outlined, some of them are handled in more detail. All transformation results from this section mostly align with the API of *flow-runtime*, the runtime type system that is used in the compiled JavaScript code.

Arrow Function Mutator

The arrow function mutator modifies the body of the passed node, while some peculiarities have to be considered. As for every other function type (i.e., function expression and function declaration) the parameters are asserted, while they can be optional or may have a default value. Also every location where the function may return a value is observed and checked. A major difference to function expressions is that they can omit the function body, previously described in Sec. 2.2.6. In such a case it has to be created in order to be able to insert runtime type checks. Alongside changing the arrow function's body, it is also annotated. This means that a reflection of the function, including its parameter types and its return type, will be added to the function object to retrieve it in other places of the running program.

Program 5.3: The arrow function mutator of *ts-runtime*.

```

1 export class ArrowFunctionMutator extends Mutator {
2
3   protected kind = ts.SyntaxKind.ArrowFunction;
4
5   protected mutate(node: ts.ArrowFunction): ts.CallExpression {
6     return this.factory.annotate(
7       this.factory.mutateFunctionBody(node),
8       this.factory.functionReflection(node)
9     );
10  }
11
12 }

```

To better describe how the result of a transformation may look like, the following arrow function is given:

```
() : string => "bar";
```

Furthermore, the arrow function mutator itself is shown in Prog. 5.3, as it is relatively small compared to other mutators, while the call to `mutateFunctionBody` on line 7 returns a node, that transforms the function to the following:

```

() => {
  const _returnType = t.return(t.string());
  return _returnType.assert("bar");
}

```

Also a description of the function signature is retrieved via `functionReflection` on line 8, which is represented with:

```
t.function(t.return(t.string()));
```

Subsequently, the arrow function is annotated with the reflection. The code below depicts a fully transformed example, when assuming that the function was assigned to a variable:

```

const foo = t.annotate(() => {
  const _returnType = t.return(t.string());
  return _returnType.assert("bar");
}, t.function(t.return(t.string())));

```

The variable `foo` holds the arrow function, with added information about its signature. The type of the identifier may then be used to declare another variable, like in the following code snippet:

```
const bar: typeof foo = (): string => "hi";
```

This results in the following code in the compiled JavaScript program:

```
const bar = t.typeOf(foo).assert(/* transformed arrow function */);
```

A function `t.typeOf`—part of the *flow-runtime* library—is called with `foo`, which extracts the previously annotated information. Therefore it can be checked if the value that should be assigned to `bar` matches the type of `foo`.

As Expression Mutator

Also TypeScript type assertions are checked at runtime. This means that if a value is casted to another type, it is verified if the value is compatible:

```
"foo" as number;
```

The code above is therefore substituted with the statement below:

```
t.number().assert("foo");
```

While the assertion used in this example already raises an error when being statically checked by the TypeScript compiler, there are situations where a cast can be performed successfully, even though the types do not match (see Sec. 4.1).

Binary Expression Mutator

Binary expressions in JavaScript (and TypeScript) include, but are not limited to, assignments, comparisons and bitwise operations [56]. The mutator which is handling such nodes is only considering assignment operations. It is worth to note that an expression with an assignment operator is a different AST node than a variable declaration with an initializer. However, the outcome of the transformation is very similar and is therefore pictured later in this section.

Block Like Mutator

The transformation API of the TypeScript compiler allows substituting AST nodes by returning another node from a visitor, i.e., a mutator in case of *ts-runtime*. While this functionality is heavily used in the project of this thesis, it does only allow to replace a node with exactly one other node. There are cases where mutators need to substitute a node with a list of nodes. These situations include declarations for functions, classes, and enums:

- As shown in the mutator for arrow functions, they are annotated with their signature reflection. Also function declarations have to be annotated in the same way, however, it is not possible to wrap them into another function call, as this would change the scope of the declaration (see Sec. 2.2.6). Therefore, the annotation is added beneath the function declaration itself.
- The same applies to enumerations, as they are initialized by a self executing function in the target code (see Sec. 4.3). Also, there is no information available about the variable that will hold the enum object, after the TypeScript compiler has emitted the final JavaScript code, during transformation. Consequently, the annotation takes place after the initialization of the runtime representation of the enumeration.
- The situation is different for classes. Decorators can be used to annotate the class, but its type parameters need to be available before the first instantiation, to make use of them in the class signature reflection.

To better illustrate, how the transformation for classes differ from function and enum declarations in the block-like mutator, the code below is given:

```
class A<T> { }
```


The class shown above will result in the following transformation by the block-like mutator:

```
const _ATypeParametersSymbol = Symbol("ATypeParameters");
class A<T> { }
```

The main focus of this example is on the first line of the snippet—while the transformed class itself is omitted—where a symbol for the class’s type parameters is declared, to expose it to the same scope as of the class.

Class Declaration Mutator

The class declaration mutator is one of the most complex mutators. It has to consider a multitude of situations, including that a class may extend another class, may implement interfaces, may have method and non-method properties, may include function overloads [59], and can merge with interface declarations [43]. Also class members may have modifiers such as *static*, *readonly*, *public*, *private* and *protected*. To make sure that all particularities are taken into account, the following steps are performed successively:

1. **Reflection:** Foremost, the class is annotated with its signature to expose its type to the runtime. The reflection includes all properties of the class and a reference to its base class, if available. As a class’s type merges with interfaces of the same name, it is necessary to retrieve the properties from all declarations of the class identifier name. Subsequently, the obtained properties can be merged, while also method overloads are combined.
2. **Methods:** Class method properties are mutated similar to regular functions, with the difference, that they may also make use of class type parameters, alongside type parameters defined on the method itself.
3. **Variables:** Also non-method properties (i.e., member variables) have to be checked. Therefore a getter and a setter is defined for it to assert the member’s type each time a new value should be assigned. If the property is marked as *readonly*, the setter will be omitted. Additionally, the initializer is checked for type compatibility.
4. **Type Parameters:** All type parameters are initialized in the constructor, making them accessible to the entire class.
5. **Interfaces:** If the class should align with one or more interfaces, type compatibility is checked when the class is instantiated.

Prog. 5.4 shows a class in TypeScript, where several of these cases are met. The result after being processed by *ts-runtime* is shown in Prog. 5.5.

Function Declaration Mutator

The function declaration mutator consists of a single line of code in the `mutate` method, which is a call to `mutateFunctionBody` from the factory. A specialty about functions of all types, which has not been handled before in this section, is their support for type parameters, also referred to as generics, which may look like the following in TypeScript:

```
function foo<T>(bar: T): T[] {
  return [bar];
}
```

Program 5.4: A class in TypeScript, which extends a base class and implements a single interface. Furthermore, a *readonly* property is defined, and the method `convert` is overloaded. The result, after being processed by *ts-runtime*, is shown in Prog. 5.5.

```

1 class NumberConverter extends Singleton implements Converter {
2
3   readonly converter: string = "NumberConverter";
4
5   convert(val: number): number;
6   convert(val: string): number;
7   convert(val: string | number): number {
8     if (typeof val === "number") {
9       return val;
10    }
11
12    return parseFloat(val);
13  }
14
15 }

```

A parameter of a generic type `T`, which is not known at compile time, is accepted and the function should return a value, that is an array of this type. For example, if a number is being passed to `foo`, the returned value should be an array of numbers. To support generics with functions, the factory can detect if a type reference is a type parameter and adjusts the transformation accordingly:

```

function foo(bar) {
  const T = t.typeParameter("T");
  let _barType = t.flowInto(T);
  const _returnType = t.return(t.array(T));
  t.param("bar", _barType).assert(bar);
  return _returnType.assert([bar]);
}

```

In the code above the annotation, which has already been outlined with the arrow function mutator, is omitted. A variable is created for the type parameter and the type of `bar` is used on every function call. This type may be extracted from an annotation, or may be inferred from the actual runtime value as accurately as possible, if no type reflection is available for it. Furthermore, it is possible to provide a default type for the generic parameter, or to extend another type, which may look like the following:

```

function elementToString<T extends HTMLElement = HTMLDivElement>(el: T): string {
  return el.innerText;
}

```

This type parameter definition results in the reflection shown below:

```

const T = t.typeParameter("T", t.ref(HTMLElement), t.ref(HTMLDivElement));

```

In this case, the value passed to the function must be a `HTMLElement`, or a subset of it, while by default `T` refers to the `HTMLDivElement` type.

Program 5.5: The resulting JavaScript code after the transformation of the class from Prog. 5.4.

```

1 @t.annotate(t.class("NumberConverter", t.extends(t.ref(Singleton)),
2   t.property("converter", t.string()),
3   t.property("convert",
4     t.function(
5       t.param("val", t.union(t.number(), t.string())),
6       t.return(t.number())
7     )
8   )
9 )
10 class NumberConverter extends Singleton {
11
12   constructor(...args) {
13     super(...args);
14     this._converter = t.string().assert("NumberConverter");
15     t.ref(Converter).assert(this);
16   }
17
18   get converter() {
19     return this._converter;
20   }
21
22   convert(val) {
23     let _valType = t.union(t.string(), t.number());
24     const _returnType = t.return(t.number());
25     t.param("val", _valType).assert(val);
26
27     if (typeof val === "number") {
28       return _returnType.assert(val);
29     }
30
31     return _returnType.assert(parseFloat(val));
32   }
33
34 }

```

Interface Declaration Mutator

The interface declaration mutator requests a reflection of the node's type from the factory. If a class with the same name exists in its scope and therefore will be or has already been merged with the interface declaration, the interface is removed and no mutation takes place. Also generics and self references are considered during the transformation.

Source File Mutator

This mutator assures the existence of the import of the runtime type checking library, if required. Also, if ambient or external declarations are collected by the scanner, the file that holds these declarations is included in every entry file as well:

```

import "./tsr-declararions";
import t from "ts-runtime/lib";

```

The first statement will only be added if the file `tsr-declarations.js` is created by the transformer, whereas the second statement is always be included, unless the library has not been used throughout the source file at all. If the identifier `t` would have already been used in the project, it would be prefixed with an underscore, until it is guaranteed that no naming conflicts can occur.

Type Alias Declaration Mutator

Type alias substitutions are very similar to interface replacements, but an important aspect of them has not been handled yet. Interfaces, type aliases, and classes can reference themselves in TypeScript. When declaring a reflection of a type at runtime, the variable that holds the type description won't be available yet in such cases:

```
type Foo = { circular: Foo; }
```

This type alias has a single property `circular`, which points to its own type. To support such a behavior in JavaScript, a function is assigned to the identifier substituting the type, which will be called with a reference to itself when being used for the first time at execution time:

```
const Foo = Foo => t.object(t.property("circular", Foo));
```

This function contains the actual type description, which is not initialized until it is required by other parts of the program.

Variable Declaration List Mutator

Variable declarations are wrapped within a node of kind *VariableDeclarationList*, which includes at least one declaration. The mutator performs a transformation on every declaration that is annotated with a type—regardless of the existence of an initializer—unless they are part of a for-of statement, for-in statement, catch clause, or import clause:

```
let foo: string = "bar";
```

This variable declaration is transformed to the following:

```
let _fooType = t.string(), foo = _fooType.assert("bar");
```

Another identifier `_fooType` is introduced to retrieve the type of `foo` whenever another value is assigned to it. For constant variables there is no need to declare the variable's type alongside the actual declaration:

```
const foo: string = "bar";
```

Therefore, by using the `const` keyword instead of `let` or `var`, the assertion is performed in place:

```
const foo = t.string().assert("bar");
```

As the native JavaScript runtime engine should throw an error, if a constant variable is reassigned, a separate type declaration can be omitted.

5.4.3 Factory

The implementations of the mutators are making use of the factory, which is created by the mutation context (see Sec. 5.4.4). It can recursively create reflections for a type node of an abstract syntax tree, while keeping track of its state. For every type node kind there exists a method that can come up with a runtime description, e.g., `literalTypeReflection`, `arrayTypeReflection`, or `typeReferenceReflection`. If the kind of a node is not determined in advance, the method `typeReflection` can be called, which invokes the suitable reflection function. The following syntax kinds are supported:

- **Keywords:** Any, Boolean, Never, Null, Number, Object, String, Symbol, Undefined, Void
- **Types:** Array, Constructor, Function, Intersection, Literal, Parenthesized, This, Tuple, Union
- **Others:** TypeLiteral, TypePredicate, TypeQuery, TypeReference, Expression-WithTypeArguments

However, three types are not yet checked by *ts-runtime*. As they are reflected with the *Any* type by the factory, the transformation process can still finish without errors, but a warning will be issued if a node with one of the syntax kinds below occurs in the project:

- IndexedAccessType
- MappedType
- TypeOperator

In addition to type node reflections, also common transformations are collected in the factory. It can, for example, reflect classes, interfaces, and type aliases, while also providing methods for the substitution of types. Furthermore the merging of declarations or method overloads is carried out by this component in certain situations.

5.4.4 Context

The context—or mutation context—is created for every source file during the traversal of the AST in the transformer. It holds a reference to the TypeScript program and the TypeScript type checker. Also the compiler options and settings for *ts-runtime* can be retrieved from the mutation context, which allows this component to provide much more sophisticated functionality than, e.g., the utility component. Methods of the context include, but are not limited to:

- Is a node the implementation of an overload?
- Is a given name declared in the current context?
- Is an identifier used before its declaration?
- Does a given type reference point to itself?
- Does a node include a type reference, that points to itself?
- Retrieve a merged list of members of a class or an interface declaration.

All of these queries require a link to the TypeScript API or the scanner component in order to come up with a response successfully.

5.4.5 Utility

The utility component is globally available to all parts of the project. It can be imported and used without any dependencies. It provides a collection of functions, which are used throughout *ts-runtime*, to not introduce duplicated code. It includes methods for determining whether a node is a type parameter of a given type node (e.g., a class, an interface, or a function declaration), or to extract the *extends* clause from a class declaration node, although the entire API is not presented at this point.

5.4.6 Event Bus

The event bus (i.e., bus) is a lightweight wrapper around the *EventEmitter*, which is part of Node.js [55]. It includes certain predefined events which can be obtained from anywhere in the project. It is used to indicate changes of the state of the program (e.g., the start of the transformation), or to notify subscribers about other important events, such as errors and warnings.

5.4.7 Scanner

This component is a key part of *ts-runtime*. It is instantiated by the core before the actual transformations take place. It visits every node of the AST from each source file, whereas a given node is only processed if it may be required for a type reflection. This includes identifiers, type references, and function declarations, besides a multitude of other syntax kinds. Furthermore, the scanner saves the name of every identifier of the project, to prevent naming conflicts when variables are introduced by the mutators. Most importantly, for every node that is inspected by the scanner an object is created, which holds a variety of information, including the node's symbol, the source files where the node and its type are declared in (if applicable), as well as a list of declarations with the same name, if the node is, e.g., a class or an interface declaration. The extraction of this details enables the scanner to determine, whether the scanned node is an ambient or external declaration, which may look like the following in TypeScript:

```
declare class Person {  
  name: string;  
}
```

The class declaration above is declared ambient, meaning that it will only be used for static type check purposes, before it is being removed by the TypeScript compiler. The thesis project does not reflect this declaration in place, but adds a runtime representation to a separate file, as shown below:

```
t.declare("Person.3174411535",  
  t.class("Person", t.property("name", t.string()))  
);
```

A type reference may use the ambient class declaration, as follows:

```
let person: Person;
```

This TypeScript code will be transformed to the code below by the mutators:

```
let _personType = t.ref("Person.3174411535"), person;
```

The number in the runtime reflection is the hashed file name to avoid the overwriting of global declarations with the same name, and to uniquely identify the type at runtime.

Program 5.6: The interface for the options of *ts-runtime*.

```
1 interface Options {  
2   compilerOptions?: ts.CompilerOptions;  
3   force?: boolean;  
4   log?: boolean;  
5   noAnnotate?: boolean;  
6   libDeclarations?: boolean;  
7   declarationFileName?: string;  
8   excludeDeclarationFile?: boolean;  
9   excludeLib?: boolean;  
10  libIdentifier?: string;  
11  libNamespace?: string;  
12  declarationPrefix?: string;  
13 }
```

5.4.8 Options

In order to provide developers with the ability to adjust the behavior of the thesis project, the options component exposes an interface, describing the supported settings (see Prog. 5.6):

- **compilerOptions:** The options for the TypeScript compiler are included in the settings for *ts-runtime*. Especially the *rootDir* and *outDir* are important. The root directory option specifies a base folder, which contains the TypeScript project to be processed. If no such option is given the common directory of the entry files will be determined. The *outDir* option sets the location of the compiled project, whereas *outFile* would tell the compiler to concatenate the target code and to only emit a single file [38]. The option *preserveConstEnums* will always be enabled, since constant enumerations need to be available for runtime type checks. By default, the TypeScript compiler would replace the enum references with their constant value [38, 40].
- **force:** The processing is aborted if the TypeScript compiler detects errors for both, syntactics and semantics. By setting this flag to **true**, semantic errors do not cause the transformations to be stopped.
- **log:** By default, errors, warnings, and other messages will be printed to the console. To disable the output, this option can be set to **false**.
- **noAnnotate:** Functions and classes are annotated with their type reflection, which can be disabled. The type checking library will try to infer the type from the value available at runtime.
- **libDeclarations:** Specific functionality is available globally in a running JavaScript program, based on its execution context (e.g., Node.js, or web browser). Those globals won't be reflected by default.
- **declarationFileName:** The scanner collects all ambient and external declarations, which are then written to a separate file. The name for this file can be set via this option.

- **excludeDeclarationFile:** The file that holds the collection of global declarations is imported in every entry file of the target code, which may be changed.
- **excludeLib:** The runtime type checking library is not only loaded in every entry file, but in every single module that includes some kind of type reflection or assertion. To disable these automatic imports, this option can be set to **true**.
- **libIdentifier:** Even though naming conflicts should not occur, as the scanner stores identifiers in use, the name for the library variable may be changed, since the execution environment of the compiled JavaScript project may already define a set of global names.
- **libNamespace:** If a prefix for the library identifier is desired it can be set with this option.
- **declarationPrefix:** In some situations new variable declarations are introduced by the mutators. To easily distinguish generated identifiers from others, a prefix can be set.

While it is possible to provide settings to the transformer, it is not required to pass anything but the entry files. The project includes default settings (see Prog. 5.7) that will be used for every option that is not specified.

Program 5.7: The default options for *ts-runtime*.

```
1 {  
2   compilerOptions: {},  
3   force: false,  
4   log: true,  
5   noAnnotate: false,  
6   libDeclarations: false,  
7   declarationFileName: "tsr-declarations",  
8   excludeDeclarationFile: false,  
9   excludeLib: false,  
10  libIdentifier: "t",  
11  libNamespace: "",  
12  declarationPrefix: "_"  
13 }
```

5.5 Transformation Procedure

A variety of elements work together to achieve the desired result of the thesis project, which have been pictured in this chapter. While they have been described with a certain level of detail, not all characteristics of every component could be highlighted. To better illustrate the procedure of the program, and the interconnections of the different components, the steps performed by the transformer are described:

1. The transformer obtains a complete set of options by requesting the default settings for *ts-runtime* and the TypeScript compiler, which can then be merged with the settings passed. If the options are not valid the transformation is stopped.

2. The root directory for the TypeScript project to be processed needs to be discovered next. Either it is provided through the TypeScript compiler option *rootDir*, or it is computed based on the entry files.
3. At this point the transformer distinguishes between a compilation of files from the file system, or a reflection which is represented by a list of objects containing the file name and its contents as a string. In order to transform a project without a file system a custom compiler host (see Sec. 5.1.1) is required, which can provide the TypeScript program with the appropriate data from the reflection list. For a regular compilation the standard compiler host, provided by TypeScript, will be used.
4. A TypeScript program is instantiated with the compiler host, the compiler options, and the entry files. The TypeScript compiler processes the project and provides access to the type checker, alongside other useful functionality, like compiler diagnostics (i.e., errors).
5. If errors are detected by TypeScript, the processing is being stopped. In case of the force option being set, the transformation will only be aborted if diagnostics occur that are not related to semantics.
6. The abstract syntax tree for each source file is scanned, identifiers are stored, and ambient and external declarations are extracted.
7. The state of the application now allows for the actual transformations to take place. Every node is passed to the mutators, which perform modifications if required.
8. As the current TypeScript program is no longer synchronized with the AST of the source files, it has to be replaced with a new instance. The TypeScript printer (see Sec. 5.1.1) is used to create a reflection of the project, which—alongside a compiler host which supports the reflections—is used to create a new program.
9. The target code can now be emitted by the TypeScript program, and the result is written to disk, or a reflection of the emitted files is created.
10. All external and ambient declarations are requested from the scanner. The transformer then creates runtime representations through the factory and includes them in the emitted result.
11. The transformation process is finished and a file reflection list, containing the target files, is returned.

A diagram depicting the most significant parts of this process can also be found in Fig. 5.2.

5.6 Usage

In order to make use of the implemented project, Node.js version 6.0 or above needs to be installed on the system. Also npm—which comes with Node.js—or *yarn*⁷ is required to install *ts-runtime* from the *npm* registry. The following command can be used with yarn, to add the package as a dependency to a project via the command line:

```
yarn add ts-runtime
```

⁷ <https://yarnpkg.com>

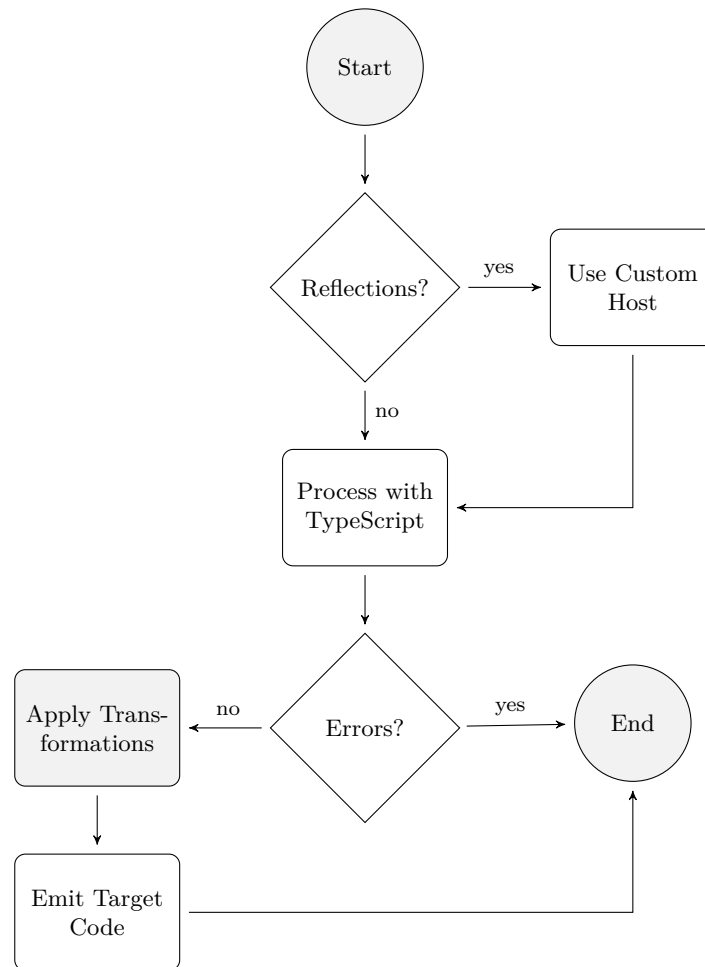


Figure 5.2: Simplified diagram of the transformation process.

With npm, the corresponding command is as follows:

```
npm install ts-runtime --save
```

After the package is available on a system, different approaches are provided to interact with *ts-runtime*, which are described in the following sections.

5.6.1 Application Programming Interface

The library exposes various parts of its internals via an application programming interface (i.e., API). To provide high flexibility when making use of this project, technically almost all parts are accessible from outside. However, for a typical setup only the options component, the transformer, as well as the bus may be used, which are exported from the main file of the published package. In JavaScript the library can be loaded as shown below:

```
import * as tsr from "ts-runtime";
```

Program 5.8: This code makes use of the API of the thesis project and utilizes the bus component to append TypeScript compiler diagnostics to a file.

```

1 import * as fs from "fs";
2 import * as ts from "typescript";
3 import { transform, bus } from "ts-runtime";
4
5 const stream = fs.createWriteStream("diags.log", { flags: "a" });
6
7 bus.on(bus.events.DIAGNOSTICS, diagnostics => {
8   logStream.write(
9     ts.formatDiagnostics(diagnostics, {
10       getCurrentDirectory: () => "",
11       getNewLine: () => "\n",
12       getCanonicalFileName: fileName => fileName
13     })
14   );
15 });
16
17 bus.on(bus.events.STOP, () => {
18   stream.end();
19 });
20
21 transform("./entry");

```

In this case, every exported member of the main file of *ts-runtime* is imported and is made available through the identifier `tsr`. To load only specific parts of the application the following syntax can be used:

```
import { transform } from "ts-runtime";
```

While the examples above make use of a code style from the EcmaScript 2015 language specification [51, p. 302], not all features from this specification are available in Node.js at this time [47], and the syntax shown below can be used [73]:

```
const tsr = require("ts-runtime");
```

However, when using a compiler that supports EcmaScript 2015 modules, which can produce Node.js compatible JavaScript—such as TypeScript [74] or Babel [79]—the syntax of the former two examples may be used. After successfully loading the thesis project, its functionality can be used programmatically, which is outlined in Prog. 5.8.

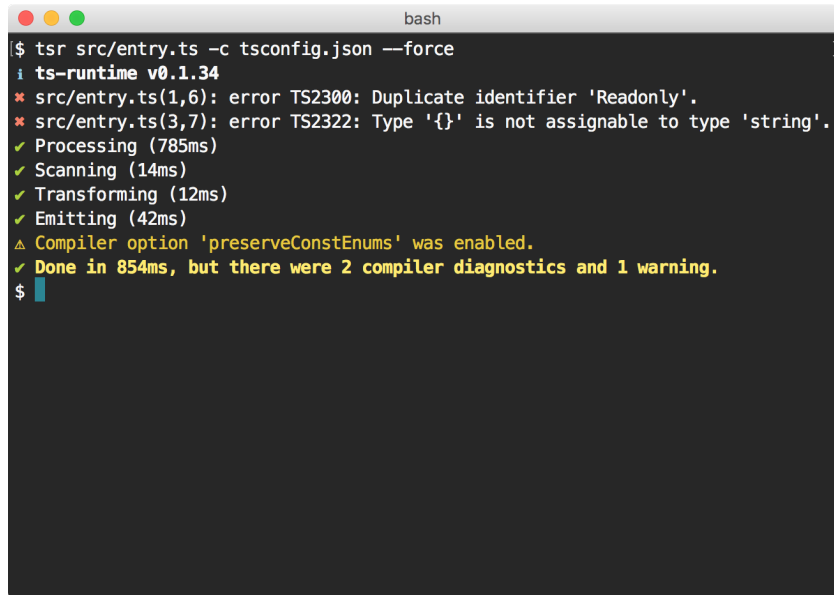
5.6.2 Command Line Interface

The project of this thesis does also include a *command line interface* (i.e., CLI), which requires the package to be installed globally via the command line:

```
yarn global add ts-runtime
```

Again, also npm may be used to install *ts-runtime*:

```
npm install -g ts-runtime
```



```

bash
$ tsr src/entry.ts -c tsconfig.json --force
i ts-runtime v0.1.34
* src/entry.ts(1,6): error TS2300: Duplicate identifier 'Readonly'.
* src/entry.ts(3,7): error TS2322: Type '{}' is not assignable to type 'string'.
✓ Processing (785ms)
✓ Scanning (14ms)
✓ Transforming (12ms)
✓ Emitting (42ms)
△ Compiler option 'preserveConstEnums' was enabled.
✓ Done in 854ms, but there were 2 compiler diagnostics and 1 warning.
$

```

Figure 5.3: Output of the command line interface, with compiler errors and warnings. The TypeScript file `entry.ts` within the directory `src` was compiled, while the TypeScript compiler options were loaded from `tsconfig.json`, and the transformation process was not aborted on the occurrence of semantic errors, as the `force` flag was set.

The CLI of the application should now be exposed to the environment variables and it may be executed from any location of the operating system. To display a help message, including available options and usage examples, the following command can be run:

```
tsr --help
```

The only argument that is required to be passed to the command line interface is a TypeScript entry file name, while the extension may be omitted. Fig. 5.3 depicts the CLI output, where the TypeScript compiler options were loaded from a file, and the transformation process was not aborted when semantic errors were raised by the compiler.

5.6.3 Playground

To test and try *ts-runtime* in a web browser, a playground was created. It takes advantage of the reflection transformation feature, which does not require an underlying file system. While the playground source is not included in the published package on the npm registry, it can be obtained from the repository on GitHub⁸, where the source code of the entire thesis project is available. The playground is served directly from this repository⁹, as shown in Fig. 5.4, where the transformed and compiled JavaScript code can be executed in the browser, to not only show the result of the target code, but to also inspect its behavior at runtime.

⁸ <https://github.com/fabiandev/ts-runtime>

⁹ <https://fabiandev.github.io/ts-runtime/>

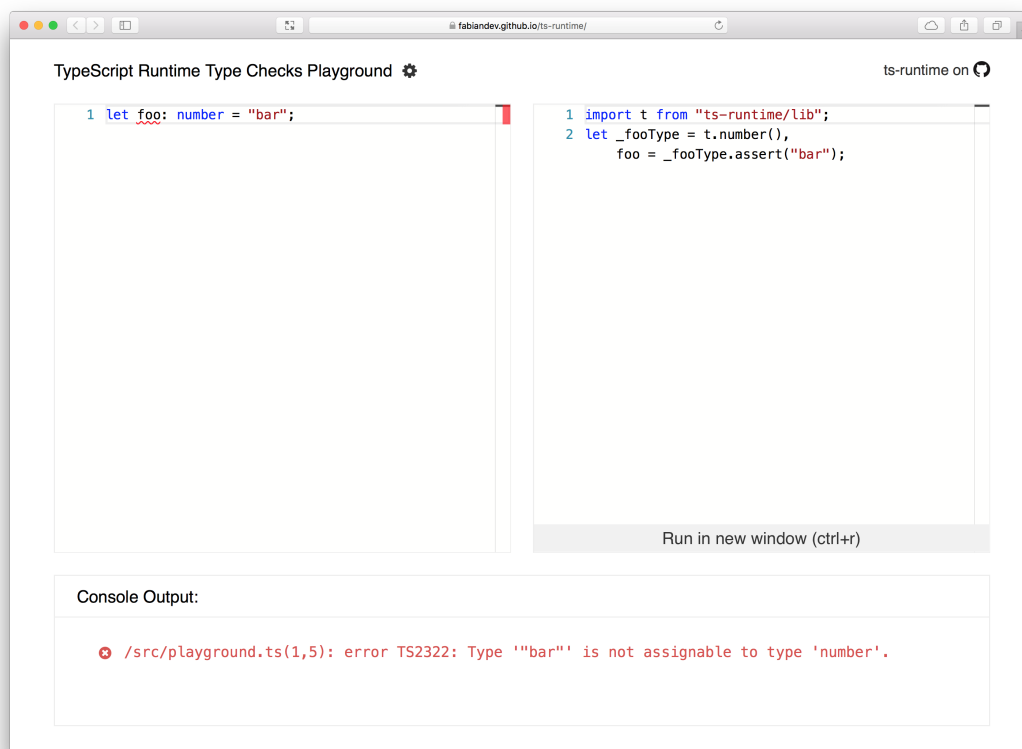


Figure 5.4: Screenshot of the playground of the thesis project, providing a user interface with support for several options to test transformations and its behavior at runtime.

Chapter 6

Evaluation

In this chapter the implemented thesis project is tested and evaluated. Different methods are used to measure and verify the quality and functionality of *ts-runtime*. This should assure that all components work as intended and the resulting transformations align with the expected outcome. Furthermore, it should highlight that the implementation was carried out carefully, while also pointing out potential compromises when making use of the library.

6.1 Automated Unit Tests

To ensure that the result of the source code transformations, applied to a TypeScript project, aligns with the expected result, a series of unit tests¹ are provided, which should be executed after modifications have been made to the project’s source code. These tests raise errors if a mutation changed unexpectedly, possibly resulting in wrong behavior when used at runtime. If such a change is intended, the corresponding tests have to be updated as well. The tests do not only cover the mutators itself, but also the project’s components. In total 456 tests have been written with a code coverage² of almost 91% (i.e., 90.59%), while Tab. 6.1 shows a detailed summary of the project’s coverage.

Table 6.1: Code coverage summary, with a total of 90.59% when considering statements, branches, and lines.

	<i>Statements</i>	<i>Branches</i>	<i>Functions</i>	<i>Lines</i>
<i>Components</i>	92.98%	82.63%	94.90%	93.34%
<i>Mutators</i>	97.14%	91.30%	98.18%	96.76%
<i>Total</i>	93.72%	84.14%	95.34%	93.92%
	2016/2151	891/1059	389/408	1870/1991

¹ “A unit test generally exercises the functionality of the smallest possible unit of code (which could be a method, class, or component) in a repeatable way [...] to verify that the logic of individual units is correct” [31].

² “Code coverage is the percentage of code which is covered by automated tests” [21].

It gives an overview of the total statements executed, the amount of branches—such as *else* statements or *case* clauses—visited, the percentage of functions executed, as well as the number of relevant lines covered in the tests. For the unit tests itself the framework *Mocha*³ is utilized, while the library *Istanbul*⁴ extracts the code coverage report.

6.2 Continuous Integration

The *ts-runtime* project takes advantage of the continuous integration (i.e., CI) practice, which can be defined as

[...] a development practice where developers integrate code into a shared repository frequently, [which] can then be verified by an automated build and automated tests. [102]

This reduces the risk of introducing errors to the library accidentally. When pushing changes to the remote repository⁵, the source code is built automatically by the continuous integration platform *Travis CI*⁶, where the state of each build is publicly visible⁷. The following steps are performed when invoking a CI build:

1. Build the project with the native TypeScript compiler.
2. Build the project again with *ts-runtime*, with the result of step one.
3. Run the unit tests for all components and mutators.
4. Execute the command line interface from the build of step two.

If one of the steps from above is not successful, the build is considered as failed, which is transparently indicated on the repository website of the project. In this case all issues should be addressed before releasing a new version of the library to the node package manager registry. If a build was successful, the code coverage statistics are transmitted to *Coveralls*⁸, a web service that keeps track of changes over time, providing an interface to explore coverage for individual files, alongside determining if code coverage increased or decreased in comparison to the previous builds. Again, these insights are available to the public⁹ and are clearly shown on the repository website of *ts-runtime*.

6.3 Operational Test

After the project of this thesis has been tested using automated unit tests, and its code coverage has been revealed, it should also be verified by applying transformations to a random library. For this purpose the TypeScript project *pretty-algorithms*¹⁰—containing “pretty, common and useful algorithms with modern [JavaScript] and beautiful tests” [64]—was obtained from a list of trending libraries on GitHub¹¹, which was

³ <https://mochajs.org>

⁴ <https://istanbul.js.org>

⁵ <https://github.com/fabiandev/ts-runtime>

⁶ <https://travis-ci.org>

⁷ <https://travis-ci.org/fabiandev/ts-runtime>

⁸ <https://coveralls.io>

⁹ <https://coveralls.io/github/fabiandev/ts-runtime>

¹⁰ <https://github.com/jiayihu/pretty-algorithms>

¹¹ <https://github.com/trending/typescript>

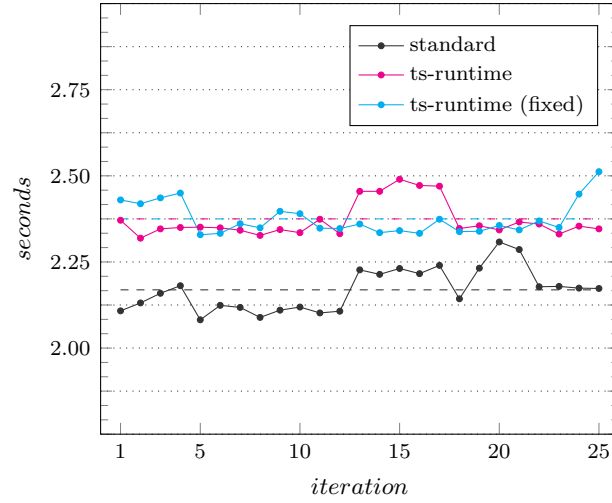


Figure 6.1: Comparison of unit test execution times of the library *pretty-algorithms*. The label *standard* refers to a build with the native TypeScript compiler, while *ts-runtime* refers to a build with generated runtime type checks with the project of this thesis, and *ts-runtime (fixed)* denotes a build where the type incompatibilities of the previous build were resolved. The dashed lines indicate the average of all iterations of a given build.

the most popular TypeScript repository on the day of testing [92]. The package was built locally and its tests were run against the original code base. Subsequently, it was built using the *ts-runtime* CLI and the unit tests were executed again. No errors were reported with the build of the native TypeScript compiler, and the average time required to execute a total of 52 unit tests after 25 runs was 2.17 seconds. With the build of the thesis project, three tests failed due to type incompatibility, with an average execution time of 2.38 seconds. After resolving the failing tests, the median time to complete the test suites was again 2.38 seconds, meaning that there was a difference of 0.21 seconds between running the unit tests against the two different builds. Fig. 6.1 depicts the time required to run the unit tests with all three builds. As the increase in execution time is given by the runtime type system provided by *flow-runtime*—which is utilized to enable runtime type reflections and assertions—the time required to build a project with *ts-runtime* is more meaningful to emphasize the performance of the thesis project, which is outlined in Tab. 6.2. On average, a *ts-runtime* build was 0.8 seconds slower, reaching around 71% of the performance of a native TypeScript compiler build, despite its supplementary functionality. The detailed results, which could be examined in this section, can further be found in Sec. A.1 and A.2. All data was gathered on a system running macOS 10.12.6 with a 2.5 GHz Intel Core i7 processor, 16 GB of 1600 MHz DDR3 memory and Node.js version 6.11.2.

Table 6.2: The average time required in seconds to build *pretty-algorithms* with the native TypeScript API, as well as the *ts-runtime* API and CLI. While a standard TypeScript build creates a JavaScript application out of TypeScript code, the *ts-runtime* builds also take care of generating and including runtime type checks.

	Average Build Time
<i>TypeScript CLI</i>	1.94s
<i>ts-runtime CLI</i>	2.72s
<i>ts-runtime API</i>	2.71s

6.4 Performance Analyzation

In addition to running the unit tests with a *ts-runtime* build of *pretty-algorithms* in a Node.js environment, the library was also compared in a browser with *Benchmark.js*¹², “a robust benchmarking library that supports high-resolution timers [and] returns statistically significant results” [32]. For graphically representing the results, *Astrobench*¹³ was used. Since including runtime type checks to a project also means that a representation of a type system is required, it was to be expected that the original build will outperform *ts-runtime*. Anyway, it may be of advantage to gain knowledge of the performance impact. When running the benchmarks, the original library could reach 56,564,588 operations per second on average, while the *ts-runtime* build scored 14,234 operations per second, meaning that it reached 0.025% of its performance. The closest result of the two builds was 21,462 operations per second with generated runtime type checks, compared to 1,648,386 operations per second with the original package, which is about 77 times faster. When comparing results of a primitive type check, a type compatibility verification of a class, as well as an interface, *ts-runtime* could accomplish its best result of 79,935 versus 338,363 operations per second (see Fig. 6.2), which is almost 24% of the performance of the JavaScript code with handwritten type checks. The entire dataset of the benchmarks regarding *pretty-algorithms* can be found in Sec. A.3, whereas Tab. 6.3 provides an overview of test results not related to that library. The benchmarks were captured on the same system as in Sec. 6.3, with the web browser *Chrome*¹⁴ version 60.0.3112.113.

6.5 Summary

The evaluation results could reveal that a project built with *ts-runtime* does not get close to the performance of a regular TypeScript build. This is due to the inclusion of a runtime type system, which is capable of reflecting and verifying complex data structures. While this overhead is added by the library *flow-runtime*, which is used for the runtime type checks itself, the performance of building a package with the thesis project performs close to building with the native TypeScript compiler, despite the

¹² <https://github.com/bestiejs/benchmark.js>

¹³ <https://github.com/kupriyanenko/astrobench>

¹⁴ <https://www.google.com/chrome/>

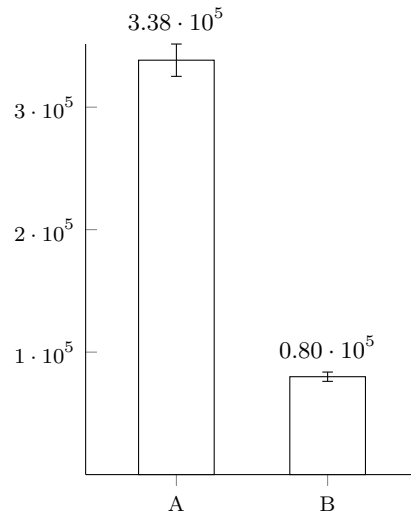


Figure 6.2: This diagram shows the benchmark results of checking a class for type compatibility in a web browser, in operations per second. *A* displays the outcome of verifying the type manually, while *B* shows the score when making use of generated runtime type checks.

additional tasks that have to be performed. Furthermore, different testing environments may influence the results. Although results may vary throughout operating systems and testing environments, the package *ts-runtime-test*¹⁵ is available on GitHub to reproduce the tests of this chapter locally, including the unit tests mentioned in Sec. 6.3, as well as the benchmark tests described in Sec. 6.4. Furthermore a detailed overview of the evaluation results can be found in Sec. A.

¹⁵ <https://github.com/fabiandev/ts-runtime-test>

Table 6.3: The results of the *generated checks* denote the runtime benchmarks for a build with the thesis project, for all of the tables below. For the *manual checks*, the `typeof` operator was used for the results of Tab. 6.3a, the `instanceof` operator for the benchmarks of Tab. 6.3b, and `Object.hasOwnProperty`, as well as the strict equality operator (i.e., `===`), for Tab. 6.3c.

(a) Results of checking a value for being a string.

	<i>Manual Checks</i>	<i>Generated Checks</i>
<i>Iterations/Cycle</i>	43,234,495	69,510
<i>Samples (Cycles)</i>	92	89
<i>Operations/Second</i>	791,979,841	1,275,061
<i>Margin of Error</i>	$\pm 0.75\%$	$\pm 1.03\%$

(b) Results of checking a value for being an instance of a class.

	<i>Manual Checks</i>	<i>Generated Checks</i>
<i>Iterations/Cycle</i>	27,266	5,139
<i>Samples (Cycles)</i>	65	81
<i>Operations/Second</i>	338,363	79,935
<i>Margin of Error</i>	$\pm 3.90\%$	$\pm 4.74\%$

(c) Results of checking a value for compatibility to an interface.

	<i>Manual Checks</i>	<i>Generated Checks</i>
<i>Iterations/Cycle</i>	1,800,557	3,140
<i>Samples (Cycles)</i>	86	90
<i>Operations/Second</i>	31,744,772	58,404
<i>Margin of Error</i>	$\pm 1.04\%$	$\pm 1.09\%$

Chapter 7

Summary and Outlook

This thesis explored the field of runtime type checks for JavaScript, with a detailed overview of its type system, which was also compared to those of other programming languages. Subsequently, the JavaScript superset TypeScript was examined in detail to provide a sophisticated overview of its characteristics and features, while also pointing out differences to and similarities with the superset Flow. It could be determined that the static compile time type analysis of TypeScript can detect a multitude of potential errors, while there are also situations where the compiler cannot detect possible issues for the target code. As no additional type checking techniques are included and type information is not available in the compiled JavaScript code, other techniques have to be employed to ensure that unexpected conditions can be observed and reacted to during program execution, resulting in increased development and maintenance effort. Therefore a method was elaborated to automatically generate runtime type checks based on the type annotations of a TypeScript project. A theoretical concept was constructed, before a project was implemented that can extract the required information and emit a JavaScript program with integrated runtime validations. For the runtime type system itself a third party library was used, which was developed with the JavaScript superset Flow in mind, but provides a multitude of features which are applicable for runtime type checks that align with the behavior of TypeScript's static type system.

Subsequently, the resulting project was evaluated—including its API, CLI and the runnable JavaScript code—to prove its quality and functionality and to also provide insights into performance analyzations and benchmarks. The findings verified the operability of the project of this thesis, and the functioning of the target code. Build times are in an acceptable range, compared to those of the native TypeScript compiler, and type incompatibilities are reported correctly during runtime. While the generation of type checks is efficient, the performance of the executable program with added runtime type checks cannot compete with the unmodified version. This is attributable to the comprehensive type system that is included, as well as its internal verification processes. While a decrease in execution time was to be expected, the results were not satisfactory to be used in a production system. However, making use of the thesis project in the phase of development may be beneficial to detect unexpected behavior and conditions at execution time. This suggests the use of a different library, which carries out type checks more efficiently to improve the performance at runtime. An interface could be provided which supports exchanging the runtime type system, without the requirement

to make changes to the underlying framework. This would also allow other developers to employ a custom implementation or a preferred library for the runtime type reflections and assertions with as little effort as possible.

To ensure that future development does not break the project’s operability, an extensive collection of automated unit tests—including code coverage statistics—is part of the project to also enable continuous integration, which is triggered automatically when changes to the remote code repository are detected, helping to preserve code quality over time and to report unexpected behavior that may be introduced with changes to the code base. The provided test suite is also useful to verify contributions to the original project from other developers.

The latest version of the source code of the project of this thesis—named *ts-runtime*—can be obtained from GitHub¹, while an online playground² is also available to transform TypeScript syntax in a browser and to execute the resulting JavaScript code.

¹ <https://github.com/fabiandev/ts-runtime>

² <https://fabiandev.github.io/ts-runtime/>

Appendix A

Evaluation Results

A.1 Build Time

<i>Iteration</i>	<i>TypeScript CLI</i>	<i>ts-runtime CLI</i>	<i>ts-runtime API</i>
1	1.942s	2.740s	2.677s
2	1.928s	2.716s	2.723s
3	1.946s	2.685s	2.671s
4	1.898s	2.697s	2.735s
5	1.903s	2.689s	2.734s
6	1.921s	2.686s	2.743s
7	1.946s	2.735s	2.633s
8	1.936s	2.685s	2.680s
9	1.913s	2.704s	2.686s
10	1.922s	2.740s	2.750s
11	1.926s	2.761s	2.655s
12	1.931s	2.706s	2.651s
13	1.947s	2.718s	2.717s
14	1.926s	2.702s	2.736s
15	1.916s	2.710s	2.736s
16	1.908s	2.716s	2.736s
17	1.949s	2.718s	2.763s
18	1.931s	2.733s	2.629s
19	1.924s	2.732s	2.718s
20	1.923s	2.716s	2.682s
21	1.944s	2.706s	2.784s
22	1.952s	2.717s	2.744s
23	1.955s	2.708s	2.686s
24	2.010s	2.793s	2.711s

<i>Iteration</i>	<i>TypeScript CLI</i>	<i>ts-runtime CLI</i>	<i>ts-runtime API</i>
25	1.969s	2.711s	2.682s
<i>Average</i>	1.935s	2.717s	2.708s

A.2 Unit Tests Execution Time

<i>Iteration</i>	<i>TypeScript Build</i>	<i>ts-runtime Build</i>	<i>ts-runtime Build (fixed)</i>
1	2.108s	2.371s	2.430s
2	2.131s	2.319s	2.419s
3	2.159s	2.346s	2.436s
4	2.181s	2.350s	2.450s
5	2.082s	2.351s	2.329s
6	2.124s	2.349s	2.333s
7	2.118s	2.342s	2.361s
8	2.089s	2.327s	2.349s
9	2.110s	2.344s	2.397s
10	2.119s	2.335s	2.390s
11	2.102s	2.374s	2.348s
12	2.107s	2.332s	2.347s
13	2.227s	2.455s	2.360s
14	2.214s	2.455s	2.335s
15	2.231s	2.490s	2.341s
16	2.216s	2.472s	2.333s
17	2.240s	2.470s	2.374s
18	2.143s	2.347s	2.338s
19	2.232s	2.355s	2.339s
20	2.308s	2.343s	2.356s
21	2.286s	2.366s	2.343s
22	2.178s	2.360s	2.369s
23	2.179s	2.331s	2.350s
24	2.174s	2.354s	2.447s
25	2.173s	2.460s	2.512s
<i>Average</i>	2.169s	2.376s	2.375s

A.3 Benchmark Tests

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	131,426	151
<i>Samples (Cycles)</i>	89	86
<i>Operations/Second</i>	2,455,478	2,693
<i>Margin of Error</i>	$\pm 0.87\%$	$\pm 2.36\%$

misc/activity-selection#activitySelector

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	50,024	195
<i>Samples (Cycles)</i>	93	93
<i>Operations/Second</i>	919,213	3,629
<i>Margin of Error</i>	$\pm 0.21\%$	$\pm 0.78\%$

misc/huffman#huffman

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	23,703	146
<i>Samples (Cycles)</i>	90	91
<i>Operations/Second</i>	441,323	2,698
<i>Margin of Error</i>	$\pm 0.65\%$	$\pm 0.49\%$

misc/inversions-count#countInversions

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	15,024	32
<i>Samples (Cycles)</i>	93	88
<i>Operations/Second</i>	277,705	579
<i>Margin of Error</i>	$\pm 0.87\%$	$\pm 0.69\%$

misc/longest-common-subsequence#findLCS

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	15,524	102
<i>Samples (Cycles)</i>	94	93
<i>Operations/Second</i>	291,322	1,912
<i>Margin of Error</i>	$\pm 0.87\%$	$\pm 0.68\%$

misc/longest-common-subsequence#lcsLength

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	90,632	455
<i>Samples (Cycles)</i>	92	92
<i>Operations/Second</i>	1,684,257	8,528
<i>Margin of Error</i>	$\pm 0.56\%$	$\pm 0.61\%$

misc/maximum-subarray#maxCrossSubarray

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	11,553	17
<i>Samples (Cycles)</i>	86	87
<i>Operations/Second</i>	204,981	315
<i>Margin of Error</i>	$\pm 0.62\%$	$\pm 1.65\%$

misc/maximum-subarray#maxSubarray

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	565,446	236
<i>Samples (Cycles)</i>	93	88
<i>Operations/Second</i>	10,313,122	4,281
<i>Margin of Error</i>	$\pm 1.07\%$	$\pm 0.67\%$

misc/priority-queue#extractMax

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	445,302	391
<i>Samples (Cycles)</i>	92	91
<i>Operations/Second</i>	8,108,396	7,159
<i>Margin of Error</i>	$\pm 0.88\%$	$\pm 0.71\%$

misc/priority-queue#increasePriority

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	286,848	206
<i>Samples (Cycles)</i>	92	90
<i>Operations/Second</i>	5,296,213	3,791
<i>Margin of Error</i>	$\pm 0.89\%$	$\pm 0.66\%$

misc/priority-queue#insert

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	51,412	322
<i>Samples (Cycles)</i>	92	92
<i>Operations/Second</i>	947,689	6,004
<i>Margin of Error</i>	$\pm 1.28\%$	$\pm 0.80\%$

misc/rod-cutting#bottomUpCutRod

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	27,315	180
<i>Samples (Cycles)</i>	92	90
<i>Operations/Second</i>	511,249	3,333
<i>Margin of Error</i>	$\pm 0.55\%$	$\pm 0.73\%$

misc/rod-cutting#cutRod

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	58,578	49
<i>Samples (Cycles)</i>	92	91
<i>Operations/Second</i>	1,079,036	912
<i>Margin of Error</i>	$\pm 0.88\%$	$\pm 0.65\%$

misc/rod-cutting#topDownCutRod

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	3,512,823	1,013
<i>Samples (Cycles)</i>	92	93
<i>Operations/Second</i>	64,198,881	18,877
<i>Margin of Error</i>	$\pm 0.92\%$	$\pm 0.56\%$

search/binary-search#binarySearch

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	4,338,946	331
<i>Samples (Cycles)</i>	89	90
<i>Operations/Second</i>	76,632,545	6,072
<i>Margin of Error</i>	$\pm 1.11\%$	$\pm 2.06\%$

search/binary-search-tree#insert

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	463,402	20
<i>Samples (Cycles)</i>	94	87
<i>Operations/Second</i>	8,496,621	368
<i>Margin of Error</i>	$\pm 0.94\%$	$\pm 0.68\%$

search/binary-search-tree#maximum

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	473,362	18
<i>Samples (Cycles)</i>	93	87
<i>Operations/Second</i>	8,672,193	334
<i>Margin of Error</i>	$\pm 0.93\%$	$\pm 0.53\%$

search/binary-search-tree#minimum

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	471,958	27
<i>Samples (Cycles)</i>	92	89
<i>Operations/Second</i>	8,655,508	499
<i>Margin of Error</i>	$\pm 0.84\%$	$\pm 1.33\%$

search/binary-search-tree#predecessor

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	423,236	21
<i>Samples (Cycles)</i>	91	89
<i>Operations/Second</i>	7,657,685	392
<i>Margin of Error</i>	$\pm 1.05\%$	$\pm 0.70\%$

search/binary-search-tree#remove

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	443,428	18
<i>Samples (Cycles)</i>	91	87
<i>Operations/Second</i>	8,162,362	331
<i>Margin of Error</i>	$\pm 0.91\%$	$\pm 0.59\%$

search/binary-search-tree#search

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	480,070	18
<i>Samples (Cycles)</i>	88	85
<i>Operations/Second</i>	8,592,765	329
<i>Margin of Error</i>	$\pm 1.04\%$	$\pm 0.67\%$

search/binary-search-tree#successor

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	411,043	26
<i>Samples (Cycles)</i>	90	89
<i>Operations/Second</i>	7,411,455	478
<i>Margin of Error</i>	$\pm 0.92\%$	$\pm 0.54\%$

search/binary-search-tree#transplant

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	68,002	309
<i>Samples (Cycles)</i>	94	89
<i>Operations/Second</i>	1,271,033	5,643
<i>Margin of Error</i>	$\pm 0.52\%$	$\pm 1.20\%$

sort/counting-sort#countingSort

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	46,278,689	8,884
<i>Samples (Cycles)</i>	92	93
<i>Operations/Second</i>	857,754,288	163,276
<i>Margin of Error</i>	$\pm 0.87\%$	$\pm 0.77\%$

sort/heap-sort#left

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	46,204,099	8,941
<i>Samples (Cycles)</i>	91	94
<i>Operations/Second</i>	850,272,444	163,986
<i>Margin of Error</i>	$\pm 0.93\%$	$\pm 1.21\%$

sort/heap-sort#right

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	588,664	202
<i>Samples (Cycles)</i>	88	91
<i>Operations/Second</i>	10,738,762	3,758
<i>Margin of Error</i>	$\pm 1.47\%$	$\pm 0.69\%$

sort/heap-sort#maxHeapify

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	196,323	122
<i>Samples (Cycles)</i>	91	90
<i>Operations/Second</i>	3,634,607	2,236
<i>Margin of Error</i>	$\pm 0.70\%$	$\pm 0.67\%$

sort/heap-sort#buildMaxHeap

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	96,408	36
<i>Samples (Cycles)</i>	90	87
<i>Operations/Second</i>	1,774,855	643
<i>Margin of Error</i>	$\pm 0.83\%$	$\pm 0.72\%$

sort/heap-sort#heapSort

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	327,267	1,765
<i>Samples (Cycles)</i>	93	89
<i>Operations/Second</i>	6,090,107	32,420
<i>Margin of Error</i>	$\pm 0.59\%$	$\pm 0.68\%$

sort/insertion-sort#insertionSort

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	330,844	887
<i>Samples (Cycles)</i>	92	91
<i>Operations/Second</i>	6,066,608	16,813
<i>Margin of Error</i>	$\pm 0.53\%$	$\pm 1.18\%$

sort/merge-and-insertion-sort#mergeAndInsertionSort

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	86,821	1,124
<i>Samples (Cycles)</i>	95	92
<i>Operations/Second</i>	1,648,386	21,426
<i>Margin of Error</i>	$\pm 0.59\%$	$\pm 0.43\%$

sort/merge-sort#merge

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	19,730	115
<i>Samples (Cycles)</i>	92	95
<i>Operations/Second</i>	356,236	2,158
<i>Margin of Error</i>	$\pm 0.69\%$	$\pm 1.60\%$

sort/merge-sort#mergeSort

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	188,938	345
<i>Samples (Cycles)</i>	94	94
<i>Operations/Second</i>	3,520,557	6,471
<i>Margin of Error</i>	$\pm 0.79\%$	$\pm 0.44\%$

sort/quick-sort#partition

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	59,085	80
<i>Samples (Cycles)</i>	92	94
<i>Operations/Second</i>	1,119,694	1,495
<i>Margin of Error</i>	$\pm 0.59\%$	$\pm 0.52\%$

sort/quick-sort#quickSort

	<i>TypeScript Build</i>	<i>ts-runtime Build</i>
<i>Iterations/Cycle</i>	186,947	238
<i>Samples (Cycles)</i>	93	87
<i>Operations/Second</i>	3,502,988	4,346
<i>Margin of Error</i>	$\pm 0.68\%$	$\pm 1.09\%$

sort/selection-sort#selectionSort

Appendix B

CD-ROM Contents

Format: CD-ROM, Single Layer, ISO9660-Format

Contents:

```
/
├── thesis.pdf ..... Digital Copy of the Thesis
├── assets/ ..... Vector and Raster Graphics
│   ├── diagrams/ ..... Diagrams and Charts
│   └── images/ ..... Images and Screenshots
├── project/ ..... Thesis Project Source
└── references/ ..... Online References
```

References

Literature

- [1] Luca Cardelli. “Type Systems”. In: *Computer Science Handbook*. Ed. by Allen B. Tucker. 2nd ed. Boca Raton, FL, USA: Chapman & Hall/CRC, 2004. Chap. 97 (cit. on pp. 3–6).
- [2] Douglas Crockford. *JavaScript: The Good Parts. Unearthing the Excellence in JavaScript*. Sebastopol, CA, USA: O’Reilly Media, 2008 (cit. on pp. 5, 16).
- [3] Stevens Fenton. *Pro TypeScript. Application-Scale JavaScript Development*. 2nd ed. Apress, Feb. 2014 (cit. on p. 18).
- [4] David Flanagan. *JavaScript: The Definitive Guide*. 6th ed. Sebastopol, CA, USA: O’Reilly Media, 2011 (cit. on pp. 7–9, 16).
- [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] Ross Harmes and Dustin Diaz. *Pro JavaScript Design Patterns. Application-Scale JavaScript Development*. Apress, 2008 (cit. on p. 26).
- [7] Kenneth C. Loudon. “Compilers and Interpreters”. In: *Computer Science Handbook*. Ed. by Allen B. Tucker. 2nd ed. Boca Raton, FL, USA: Chapman & Hall/CRC, 2004. Chap. 99 (cit. on p. 16).
- [8] Kenneth C. Loudon and Kenneth A. Lambert. *Programming Languages: Principles and Practices*. 3rd ed. Boston, MA, USA: Course Technology, 2011 (cit. on p. 3).
- [9] 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).
- [10] Den Odell. *Pro JavaScript Development. Coding, Capabilities, and Tooling*. Expert’s voice in Web development. Apress, 2014 (cit. on p. 5).
- [11] Benjamin C. Pierce. *Types and Programming Languages*. Cambridge, MA, USA: MIT Press, 2002 (cit. on pp. 3–5).
- [12] Martin Rinehart. *JavaScript Object Programming*. Apress, 2015 (cit. on pp. 5, 9).

- [13] Nathan Rozentals. *Mastering TypeScript*. Birmingham, UK: Packt Publishing, Apr. 2015 (cit. on p. 26).
- [14] Nathan Rozentals. *Mastering TypeScript*. 2nd ed. Birmingham, UK: Packt Publishing, 2017 (cit. on p. 18).
- [15] Kyle Simpson. *Scopes & Closures*. You Don't Know JS. Sebastopol, CA, USA: O'Reilly Media, Mar. 2014 (cit. on pp. 11, 12).
- [16] Kyle Simpson. *this & Object Prototypes*. You Don't Know JS. Sebastopol, CA, USA: O'Reilly Media, Apr. 2015 (cit. on pp. 10, 13).
- [17] Kyle Simpson. *Types & Grammar*. You Don't Know JS. Sebastopol, CA, USA: O'Reilly Media, Jan. 2016 (cit. on pp. 11, 13–15).
- [18] Kyle Simpson. *Up & Going*. You Don't Know JS. Sebastopol, CA, USA: O'Reilly Media, Apr. 2015 (cit. on pp. 6, 16).
- [19] 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).
- [20] Christian Wagenknecht and Michael Hielscher. *Formale Sprachen, abstrakte Automaten und Compiler. Lehr- und Arbeitsbuch für Grundstudium und Fortbildung*. 2nd ed. Wiesbaden, DE: Springer Vieweg, 2014 (cit. on p. 16).

Online sources

- [21] *About Code Coverage*. June 2017. URL: <https://confluence.atlassian.com/clover/about-code-coverage-71599496.html> (visited on 08/22/2017) (cit. on p. 62).
- [22] *About npm*. URL: <https://www.npmjs.com/about> (visited on 07/15/2017) (cit. on p. 38).
- [23] *About pull requests*. URL: <https://help.github.com/articles/about-pull-requests/> (visited on 04/21/2017) (cit. on p. 5).
- [24] Arnav Aggarwal. *Explaining Value vs. Reference in Javascript*. July 2017. URL: <https://codeburst.io/explaining-value-vs-reference-in-javascript-647a975e12a0> (visited on 09/01/2017) (cit. on p. 9).
- [25] *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. 8).
- [26] *Babel - The compiler for writing next generation JavaScript*. URL: <http://babeljs.io> (visited on 08/23/2017) (cit. on p. 24).
- [27] *babel-plugin-syntax-typescript*. URL: <https://github.com/babel/babel/tree/7.0/packages/babel-plugin-syntax-typescript> (visited on 08/30/2017) (cit. on p. 28).
- [28] *Basic Types - TypeScript*. URL: <https://www.typescriptlang.org/docs/handbook/basic-types.html> (visited on 05/08/2017) (cit. on pp. 19, 20, 29).

- [29] Ron Buckton. *GitHub Pull Request #13940. Expose public API for transformation*. Feb. 2017. URL: <https://github.com/Microsoft/TypeScript/pull/13940> (visited on 07/15/2017) (cit. on p. 38).
- [30] Ron Buckton. *GitHub Pull Request #15377. Fix visitEachChild for signatures*. Apr. 2017. URL: <https://github.com/Microsoft/TypeScript/pull/15377> (visited on 07/15/2017) (cit. on p. 39).
- [31] *Building Effective Unit Tests*. URL: <https://developer.android.com/training/testing/unit-testing/index.html> (visited on 08/08/2017) (cit. on p. 62).
- [32] Mathias Bynens. *A benchmarking library*. URL: <https://github.com/jiayihu/pretty-algorithms> (visited on 08/28/2017) (cit. on p. 65).
- [33] Giulio Canti. *Babel plugin for static and runtime type checking using Flow and tcomb*. URL: <https://github.com/gcanti/babel-plugin-tcomb> (visited on 08/30/2017) (cit. on p. 28).
- [34] Giulio Canti. *io-ts - TypeScript compatible runtime type system for IO validation*. URL: <https://github.com/gcanti/io-ts> (visited on 07/15/2017) (cit. on p. 41).
- [35] Giulio Canti. *tcomb - Type checking and DDD for JavaScript*. URL: <https://github.com/gcanti/tcomb> (visited on 07/15/2017) (cit. on p. 41).
- [36] Ryan Cavanaugh. *GitHub Issue #1573. Runtime type checking. Comment #68374376*. Dec. 2014. URL: <https://github.com/Microsoft/TypeScript/issues/1573#issuecomment-68374376> (visited on 07/02/2017) (cit. on pp. 1, 25).
- [37] Ryan Cavanaugh and Jonathan D. Turner. *TypeScript Design Goals*. Sept. 2014. URL: <https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals> (visited on 07/02/2017) (cit. on pp. 1, 24, 25).
- [38] *Compiler Options - TypeScript*. URL: <https://www.typescriptlang.org/docs/handbook/compiler-options.html> (visited on 07/04/2017) (cit. on pp. 1, 22, 23, 35, 39, 55).
- [39] *const - JavaScript*. 2017. URL: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const> (visited on 09/07/2017) (cit. on p. 12).
- [40] *Constant Enum Declarations - TypeScript Language Specification*. URL: <https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md#94-constant-enum-declarations> (visited on 07/19/2017) (cit. on p. 55).
- [41] 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. 5).
- [42] Tom Crockett. *runtypes - Runtime validation for static types*. URL: <https://github.com/pelotom/runtypes> (visited on 07/15/2017) (cit. on p. 41).
- [43] *Declaration Merging - TypeScript*. URL: <https://www.typescriptlang.org/docs/handbook/declaration-merging.html> (visited on 07/18/2017) (cit. on p. 49).
- [44] *decorators transform - Babel*. URL: <https://babeljs.io/docs/plugins/transform-decorators/> (visited on 07/03/2017) (cit. on p. 26).

- [45] *Decorators - TypeScript*. URL: <https://www.typescriptlang.org/docs/handbook/decorators.html> (visited on 07/03/2017) (cit. on p. 26).
- [46] *Ecma International*. URL: <https://www.ecma-international.org> (visited on 04/21/2017) (cit. on p. 5).
- [47] *ECMAScript 2015 (ES6) and beyond*. URL: <https://nodejs.org/en/docs/es6/> (visited on 08/02/2017) (cit. on p. 59).
- [48] *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 pp. 11, 24).
- [49] *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. 15).
- [50] *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. 15).
- [51] *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. 6–9, 11, 15, 24, 59).
- [52] *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. 15).
- [53] *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. 15).
- [54] *Enums - TypeScript*. URL: <https://www.typescriptlang.org/docs/handbook/enums.html> (visited on 05/08/2017) (cit. on pp. 22, 34).
- [55] *Events - Node.js Documentation*. URL: <https://nodejs.org/api/events.html> (visited on 07/18/2017) (cit. on p. 54).
- [56] *Expressions and operators*. 2017. URL: https://developer.mozilla.org/en/docs/Web/JavaScript/Guide/Expressions_and_Operators (visited on 07/17/2017) (cit. on p. 48).
- [57] Nick Fitzgerald and Jason Orendorff. *ES6 In Depth: Destructuring*. May 2015. URL: <https://hacks.mozilla.org/2015/05/es6-in-depth-destructuring/> (visited on 08/23/2017) (cit. on p. 23).
- [58] *flow - Open Source at Facebook - Facebook Code*. URL: <https://code.facebook.com/projects/1524880081090726/flow/> (visited on 08/23/2017) (cit. on p. 24).
- [59] *Functions - TypeScript*. URL: <https://www.typescriptlang.org/docs/handbook/functions.html> (visited on 07/18/2017) (cit. on p. 49).
- [60] *Generics - TypeScript*. URL: <https://www.typescriptlang.org/docs/handbook/generics.html> (visited on 07/03/2017) (cit. on p. 33).

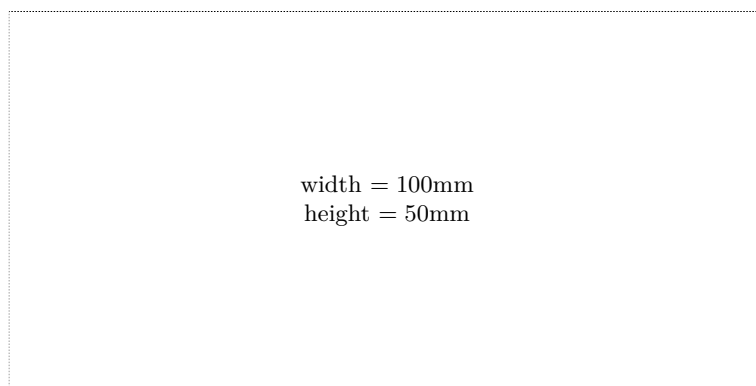
- [61] *GitHub Issue #3015. also emit type arguments with `-emitDecoratorMetadata`*. Aug. 2015. URL: <https://github.com/Microsoft/TypeScript/issues/3015> (visited on 07/02/2017) (cit. on pp. 1, 25).
- [62] *GitHub Language Stats*. 2016. URL: <https://octoverse.github.com> (visited on 04/21/2017) (cit. on p. 5).
- [63] Mohamed Hegazy. *GitHub Issue #3015. also emit type arguments with `-emitDecoratorMetadata`. Comment #128149650*. Aug. 2015. URL: <https://github.com/Microsoft/TypeScript/issues/3015#issuecomment-128149650> (visited on 07/02/2017) (cit. on pp. 1, 25).
- [64] Jiayi Hu. *Pretty, common and useful algorithms with modern JS and beautiful tests*. URL: <https://github.com/bestiejs/benchmark.js> (visited on 08/28/2017) (cit. on p. 63).
- [65] *Installation - Flow*. URL: <https://flow.org/en/docs/install/> (visited on 08/23/2017) (cit. on p. 24).
- [66] *instanceof - JavaScript*. 2017. URL: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof> (visited on 07/03/2017) (cit. on p. 25).
- [67] Mike Jones and Saisang Cai. *Variable Scope (JavaScript)*. Jan. 2017. URL: <https://docs.microsoft.com/en-us/scripting/javascript/advanced/variable-scope-javascript> (visited on 09/07/2017) (cit. on p. 12).
- [68] *JSX - TypeScript*. URL: <https://www.typescriptlang.org/docs/handbook/jsx.html> (visited on 05/08/2017) (cit. on pp. 20, 25).
- [69] Yehuda Katz and Brian Terlson. *Class and Property Decorators*. URL: <https://github.com/tc39/proposal-decorators> (visited on 07/03/2017) (cit. on p. 26).
- [70] 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. 16).
- [71] *List of languages that compile to JS*. URL: <https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js> (visited on 05/08/2017) (cit. on p. 24).
- [72] Jed Mao. *GitHub Issue #1573. Runtime type checking*. Dec. 2014. URL: <https://github.com/Microsoft/TypeScript/issues/1573> (visited on 07/02/2017) (cit. on pp. 1, 25).
- [73] *Modules - Node.js Documentation*. URL: <https://nodejs.org/api/modules.html> (visited on 08/02/2017) (cit. on p. 59).
- [74] *Modules - TypeScript*. URL: <https://www.typescriptlang.org/docs/handbook/modules.html> (visited on 08/02/2017) (cit. on p. 59).
- [75] *Namespaces - TypeScript*. URL: <https://www.typescriptlang.org/docs/handbook/namespaces.html> (visited on 05/08/2017) (cit. on p. 22).
- [76] Charles Pick. *babel-plugin-flow-runtime*. URL: <https://github.com/codemix/flow-runtime/tree/master/packages/babel-plugin-flow-runtime> (visited on 07/15/2017) (cit. on pp. 28, 41).

- [77] Charles Pick. *flow-runtime - A runtime type system for JavaScript with full Flow compatibility*. URL: <https://github.com/codemix/flow-runtime/tree/master/packages/flow-runtime> (visited on 07/15/2017) (cit. on p. 41).
- [78] Fabian Pirklbauer. *GitHub Issue #15192. "Lexical environment is suspended" when using visitEachChild from nightly transformer API (2.3.0-dev)*. Apr. 2017. URL: <https://github.com/Microsoft/TypeScript/issues/15192> (visited on 07/15/2017) (cit. on p. 39).
- [79] *Plugins - Babel*. URL: <http://babeljs.io/docs/plugins/> (visited on 07/03/2017) (cit. on p. 59).
- [80] Sylvain Pollet-Villard. *ObjectModel - Strong Dynamically Typed Object Modeling for JavaScript*. URL: <https://github.com/sylvainpolletvillard/ObjectModel> (visited on 07/15/2017) (cit. on p. 40).
- [81] Axel Rauschmayer. *Beyond typeof and instanceof: simplifying dynamic type checks*. Aug. 2017. URL: <http://2ality.com/2017/08/type-right.html> (visited on 08/30/2017) (cit. on p. 27).
- [82] Axel Rauschmayer. *Customizing basic language operations via well-known symbols*. 2017. URL: http://exploringjs.com/es6/ch_oop-besides-classes.html#_property-key-symbolhasinstance-method (visited on 08/30/2017) (cit. on p. 27).
- [83] Sam Rijs. *GitHub Issue #7607. Proposal: Run-time Type Checks*. Mar. 2016. URL: <https://github.com/Microsoft/TypeScript/issues/7607> (visited on 07/02/2017) (cit. on pp. 1, 25).
- [84] Victor Savkin. *Runtime type checks for JavaScript and TypeScript*. Nov. 2015. URL: <https://github.com/vsavkin/RuntimeTypeChecks> (visited on 07/03/2017) (cit. on p. 26).
- [85] 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. 7).
- [86] *Standard ECMA-262 Archive*. URL: <https://www.ecma-international.org/publications/standards/Ecma-262-arch.htm> (visited on 05/04/2017) (cit. on p. 15).
- [87] 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. 13, 18, 19, 21, 39).
- [88] Brian Terlson and Sebastian Markbåge. *Pattern matching syntax for ECMAScript*. URL: <https://github.com/tc39/proposal-pattern-matching> (visited on 08/30/2017) (cit. on p. 27).
- [89] Brian Terlson and Sebastian Markbåge. *TC-39 Proposal for additional is{Type} APIs*. URL: <https://github.com/jasnell/proposal-istypes> (visited on 08/30/2017) (cit. on p. 27).
- [90] *The repository for high quality TypeScript type definitions*. 2017. URL: <https://github.com/DefinitelyTyped/DefinitelyTyped> (visited on 07/02/2017) (cit. on p. 31).
- [91] *The TC39 Process*. URL: <https://tc39.github.io/process-document/> (visited on 07/03/2017) (cit. on p. 26).

- [92] *Trending TypeScript repositories on GitHub on Aug. 24, 2017*. URL: <https://web.archive.org/web/20170824162508/https://github.com/trending/typescript?since=daily> (visited on 08/24/2017) (cit. on p. 64).
- [93] *Type Annotations - Flow*. URL: <https://flow.org/en/docs/types/> (visited on 08/23/2017) (cit. on p. 24).
- [94] *Type Compatibility - TypeScript*. URL: <https://www.typescriptlang.org/docs/handbook/type-compatibility.html> (visited on 05/08/2017) (cit. on p. 21).
- [95] *typeof - JavaScript*. 2017. URL: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof> (visited on 04/24/2017) (cit. on p. 7).
- [96] *Types & Expressions - Flow*. URL: <https://flow.org/en/docs/lang/types-and-expressions/> (visited on 08/23/2017) (cit. on p. 24).
- [97] *TypeScript - JavaScript that scales*. URL: <https://www.typescriptlang.org> (visited on 08/23/2017) (cit. on pp. 1, 23).
- [98] *TypeScript vs Flow*. URL: <https://github.com/niieani/typescript-vs-flowtype> (visited on 08/23/2017) (cit. on p. 24).
- [99] *var - JavaScript*. 2017. URL: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var> (visited on 09/07/2017) (cit. on p. 12).
- [100] *void operator - JavaScript*. 2017. URL: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/void> (visited on 08/23/2017) (cit. on p. 23).
- [101] 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. 18).
- [102] *What is Continuous Integration?* URL: <https://codeship.com/continuous-integration-essentials> (visited on 08/23/2017) (cit. on p. 63).

Check Final Print Size

— Check final print size! —



— Remove this page after printing! —