Generating Runtime Type Checks for JavaScript from TypeScript

Fabian Pirklbauer

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2017

Declaration

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

Hagenberg, September 15, 2017

Fabian Pirklbauer

Contents

Abstract

Kurzfassung

Chapter 1

Introduction

Chapter 2

Technical Foundation

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

2.1   Type Systems

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

> [a] collection of type rules for a typed programming language [1, p. 38]

with the purpose to

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

He further equates *type system* with *static type system* and also Pierce defines type systems as being static [11, p. 2], which categorizes languages as untyped, that may distinguish between types in run time, but do not have knowledge about types during compile time or interpretation, such as JavaScript (see section **??**). This notion is further supported by Louden and Lambert, stating that

> [l]anguages without static type systems are usually called untyped languages (or dynamically typed languages). Such languages include [...] most scripting languages such as Perl, Python, and Ruby [8, p. 331].

---

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

Obviously, 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 typed* or *dynamically typed* are avoided in favor for *statically checked* and *dynamically checked*, respectively [1, p. 1]. This should help to avoid confusion over languages having types, but are referred to as untyped.

2.1.1 Explicitly and Implicitly Typed

If types are part of the syntax of a language (e.g. Java) it is explicitly typed, whereas in implicitly typed languages, type annotations are assigned automatically by the type system [1, pp. 2–3]. Some languages, however, make use of a mixture, allowing developers to omit type annotations in various scenarios, where the type can be inferred by the compiler [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 errors already during compile time, it is not required to guard against specific errors. There are mechanisms for untyped languages to make them safe [1, p. 3], as outlined in section 2.1.4.

2.1.2 Execution Errors

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

Trapped Errors

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

Untrapped Errors

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

Forbidden Errors

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

2.1.3   Safety and Good Behavior

A programming language can be considered as safe if no untrapped errors can appear, and is well behaved (i.e., good behaved), if no forbidden errors can occur [1, p. 3], consequently good behavior implies safety. Not all major languages are safe, and therefore not well behaved, such as C or C++ [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. 5].

2.1.4   Type Checking

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

2.2   JavaScript

JavaScript dates back to 1996, where its creator Brendan Eich from the company *Netscape*[2] submitted the language to Ecma International[3] [10, p. 28], an "industry association founded in 1961, dedicated to the standardization of information and communication systems [39]" and since became one of the most popular programming languages in the world [2, p. 2]. According to *GitHub*[4] it was the most popular language on its platform by opened *Pull Requests*[5] with a growth of 97% in 2016, followed by Java, which saw an increase of 63% compared to 2015. Also *TypeScript* (see 3.1) is following up, which takes the 15th place with an increase of Pull Request by 250% [49].

While JavaScript is known for programming inside browsers and 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*, also used in the popular *Chrome*[7] browser [10, p. 369].

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

---

[2]*Netscape Communications*—founded as *Mosaic* in 1994—released its *Netscape Communicator* browser in 1995 which became the leading internet browser at that time [35].

[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 [22].

[6]https://nodejs.org

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

outlining its most important and interesting concepts. Please note, that not all cases—especially the numerous exceptions—are described, as this would go beyond the scope of this thesis.

2.2.1 Loose Typing

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

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

The term *loose typing* may be misleading to infer that JavaScript has a type system. However, when following standard terminology and keeping in mind that type system is equal to *static* type system, it is made clear, that JavaScript is considered untyped. It does employ mechanisms to reject code from running, that has semantic errors, but evaluation is performed during execution, and errors are determined and reported during run time [43, p. 291]. Therefore JavaScript can be deemed a *dynamically checked* language (see section 2.1.4).

2.2.2 Value Types

JavaScript is an untyped and dynamically checked—but safe—scripting language, as defined in section 2.1. Most untyped programming languages are necessarily safe, as it would be exceedingly difficult to maintain the code if untrapped errors would remain unnoticed [1, p. 4]. Even though considered as untyped, the EcmaScript language specification defines seven value types [43, p. 16]:

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

A major difference to a (statically) typed language is, that in JavaScript only values are typed, variables are not. When requesting the type of a variable with the `typeof` operator during run time, the assigned value's type is determined and returned as a string [18, p. 30]:

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

The returned string by `typeof` does not reflect the specified value types entirely. As shown in table 2.1, objects are differentiated by wether they are callable or not. For objects with a call signature `"function"` will be returned and `"object"` otherwise. Strangely, for a value of type *Null*, the result will be `"object"`. A proposal to change the specification and to fix this bug—erroneously indicating that null is an object—was rejected, as existing code may break [63, 69].

Table 2.1: Result of the `typeof` operator by a value's type [43, p. 164].

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

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

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

2.2.3 Type Conversion

In JavaScript "any [...] value can be converted to a boolean value [4, p. 40]". If the interpreter expects a boolean value, it simply performs a conversion [4, p. 46] (see section 2.2.4). Because of that it is important to know, which values are considered true, and which conversion will result in being false, depending on their type, as shown in table 2.2. There are various situation where a conversion is desired, which happens implicitly in JavaScript. For example if a string should be added to a number, and vice versa, the number is converted to a string and the result is a concatenation of both values:

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

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

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

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

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

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

Table 2.3: Type conversions in JavaScript [4, p. 46, 43, pp. 36–44].

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

(i) For situations, where converting a value to an object does not throw a TypeError, a new object of the value's type is returned. E.g. for the value `"Hello world!"`, `new String("Hello world!")` is returned [43, p. 44].

(ii) When converting an object to a string, JavaScript tries to call the `toString` or `valueOf` method on the object and converts the returned value to a string. If no primitive value can be obtained from either of these methods, a TypeError is thrown [4, p. 50].

(iii) The same steps as in (ii) are performed with the difference, that `valueOf` will be preferred over `toString`.

(iv) The toString method of the Array object joins the array separated by a comma, which would result in `["a","b","c"]` being converted to `"a,b,c"` [23].

2.2.4 Value Comparison

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

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

When comparing with the *equality operator* `==` there are a few rules to keep in mind [4,

p. 72]:

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

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

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

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

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

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

2.2.5 Objects and Prototypal Inheritance

Every value that is not a primitive value (i.e., Undefined, Null, Boolean, Number, Symbol, or String [43, p. 5])—including functions and arrays—is an object, hence making JavaScript a highly flexible language. A key concept of the language is the prototypal inheritance of objects. Every object has a prototype, which is just a reference to another object. Anyway, the prototype is not accessible for all types of objects, but for functions, or more precisely, constructors [43, p. 3] (see figure 2.1). A constructor function is just an ordinary JavaScript function, that—by convention—begins with a capital letter [12, p. 8]. Initial values can be defined, that may be different for every instance created from the constructor.

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

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

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

---

[8]Todo: Explain by reference and by value shortly
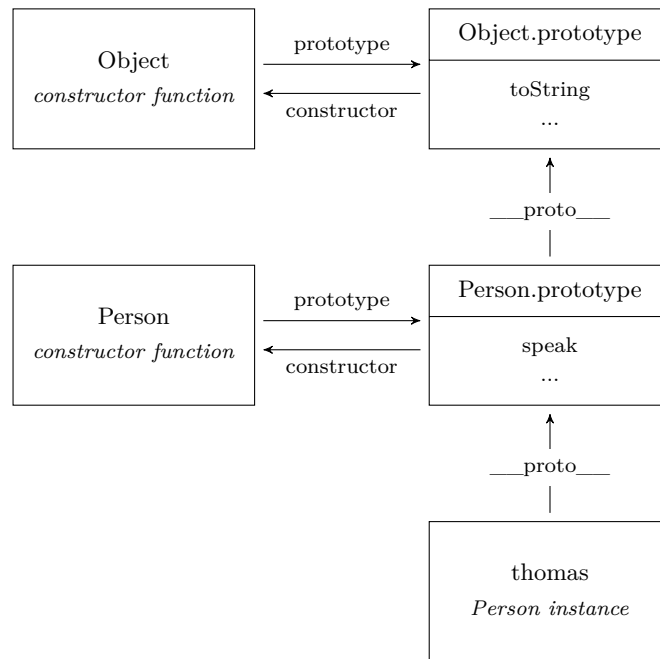
Figure 2.1: Prototypal inheritance in JavaScript.

By calling `thomas.speak()` in the code below, JavaScript looks for a *speak* property on the object *thomas*. As there doesn't exist any property with that name, the interpreter looks at the object's prototype and successfully calls the method [16, pp. 85–86].

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

If again no property *speak* would exist on the prototype, JavaScript would look at the prototype's prototype, and so on. This is called the prototype chain [16, p. 86].

2.2.6 Latest Improvements

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

Declaration Keywords

Until and including the fifth edition of ECMAScript (i.e., ES5), the only declaration keyword available was `var` [40, p. 87]. As of the sixth edition of ECMAScript (i.e., ES6), the keywords `let`—as seen in previous code snippets—and `const` were added [43, p. 194]. In order to understand the impact of using one keyword over another, it is important

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

to know the basics of how variables are scoped in JavaScript. Simpson defines a scope as

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

> [n]o matter *where* a function is invoked from, or even *how* it is invoked, its lexical scope is *only* defined by where the function was declared [15, p. 16].

Program 2.1 gives a simple example of how scopes behave in JavaScript and also shows the importance to be aware of it. While there are ways to get around lexical scoping in JavaScript, those mechanisms are considered bad practice [15, p. 14] and come with performance issues [15, p. 21], hence won't be covered here.

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 will be scoped by any block that is created, including loops. JavaScript, however, has a function scope, as shown previously in program 2.1. As of ES6, with `let` it is possible to make use of block scoped variables, whereas `var` would lead to the variable being scoped to its parent function or the global scope, if no enclosing function exists. The code below demonstrates, that creating a simple block in JavaScript in combination with a `var` declaration, does not scope the identifier to that block [17, p. 8].

```
 1 var a = 1;
 2
 3 {
```

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

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

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

On the other hand, when declaring `a` on line 4 with the `let` keyword, the variable will be scoped to its enclosing block. In this case it doesn't make any difference, whether the declaration on the first line uses `var` or `let`, as it is top level and will live in the global scope anyway.

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

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

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

```
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 [66, 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

In order to understand the differences of the declaration keywords in the previous section, a fundamental understanding of scopes in JavaScript is indispensable. For the concept of arrow functions, also introduced in ECMAScript 2015, a basic knowledge of the `this` keyword is required. In contrast to the function scope, `this` is bound in runtime and is not associated to where a function is placed in the code [16, p. 9]. Simpson puts it succinctly, that

> [w]hen a function is invoked, [...] an execution context is created. This [context] contains information about where the function was called from (the call-stack), *how* the function was invoked, what parameters were passed, etc. One of the properties of this [context] is the `this` reference, which will be used for the duration of that function's execution [16, p. 1].

Thus, the value bound to `this` differs and is influenced by *how* and from *where* a function is called. Arrow functions on the other hand use lexical instead of dynamic binding for `this` [17, p. 58]. Furthermore they inherit the `arguments` array from its parent, and also `super` and `new.target` are lexically bound [17, p. 59]. Program 2.3 shows the behavior when using a traditional function alongside `this`.

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

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

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

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

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

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

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

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

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

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

The main purpose of arrow functions, however, is not to reduce the number of characters needed for a function, but the lexical binding of `this` as shown in program 2.4. Using an arrow function over a function or vice versa, without being aware of the differences, may result in unexpected behavior.

Classes

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

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

When looking at program 2.6, which shows how to accomplish a class-like behavior in ES5 and below, it is made clear, that classes in JavaScript don't work like traditional classes in other languages, and actually rely on its concept of prototypes [17, p. 135].

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

JavaScript has been using the addition operator for string concatenation, which is still possible in the latest standard [42]. In the sixth edition of ECMAScript, template literals were introduced [43, 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 has to be added as a string between the two variables. The same result could be achieved, by creating an array from these identifiers and to join the values by a space:

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

Starting with ES6, another possibility is to use template strings—delimited with back-ticks rather than quotes—where expressions can be inserted [17, p. 48]:

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

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

Beyond ECMAScript 2015

The development of JavaScript is dependent on its specification defined by ECMAScript and it took long for new editions to be released [65]. Edition 5.1 was published in 2011 [44], from where it took four years until the sixth edition was released in June 2015 [41]. Starting with ECMAScript 2015, a new specification will be released yearly [45], with the seventh edition of ECMAScript from 2016 being the latest [64, 42].

2.2.7 Extensions

There are a multitude of extensions available for JavaScript. In [54] an extensive list of languages that compile to JavaScript, JavaScript supersets, parsers and compilers can be found. That list includes *CoffeeScript*[9]—a "[...] language that compiles into

---

[9]http://coffeescript.org

JavaScript" [33]—as well as the static type checkers *TypeScript*[10] and *Flow*[11]. Languages that have packages available for transforming its code to JavaScript include, but are not limited to, *Ruby, Python, Java, Scala* and *C#*.

2.2.8 Further Reading

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

2.3 Abstract Syntax Tree

An *abstract syntax tree* (i.e., *AST*) is the representation of a source program, created for analyzation purposes [7, p. 19], containing only the indispensable parts of the code [20, p. 12] for the most parts. A syntax tree—or abstract syntax tree—is usually created by the compiler at an early stage. More specifically it is usually the second out of five compilation phases [7, pp. 2–3]:

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

To demonstrate, how an abstract syntax tree may look like for TypeScript, a simple variable declaration was inserted in the online editor *AST explorer*[12], which can visualize the syntax tree generated by numerous parsers, including JavaScript, various JavaScript supersets such as TypeScript and Flow, CSS, and HTML [53]. The resulting syntax tree is illustrated in figure 2.2.
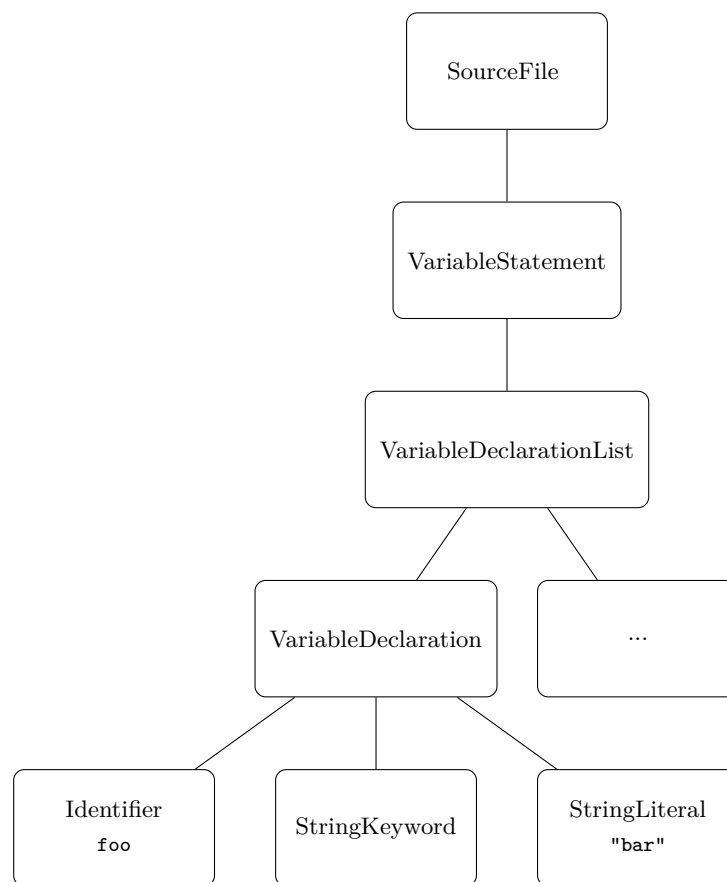
---

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

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

Chapter 3

State of the Art

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

> [a] set containing all elements of a smaller set. If B is a subset of A, then A is a superset of B [...] [70].

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

3.1 TypeScript

TypeScript was created by Anders Hejlsberg—the designer of C#—at Microsoft [14, p. 10] and was released in 2012 under the Apache 2.0 open source license [3, p. xix]. The most important aspect of TypeScript is, that it includes a compilation step, where static type checking is performed [14, p. 11]. Type annotations are optional and the compiler will infer type information where possible [66, 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 will be explored in this section.

3.1.1 Basic Types

Ï Like many major languages, TypeScript defines some basic types, that overlap with JavaScript's types, listed in section 2.2.2, while also introducing new ones [25]:

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

17

- *Any* results in type checking not being performed by the compiler. This is useful when using TypeScript alongside third-party-libraries where no type definitions are available. It also allows e.g. access to any property on an object, whether it does exist or not.
- *Void* is the counterpart to Any. Again, it is used in other languages, for example to annotate functions that do not return a value. In TypeScript also variables may be typed as `void`, meaning that only `undefined` or `null` will be accepted as value.
- *Never* for instance is useful for functions that always throw an error or result in an infinite loop, as no value will ever come back [25].

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

3.1.2   Type Inference

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

```
let num = 1;
```

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

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

The term *diagnostic* over *error* is used here, because by default the compiler won't stop in such cases and will do its best to emit the final JavaScript code [66, p. 12]:

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

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

3.1.3   Type Annotations

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

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

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

3.1.4   Type Assertions

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

> [...] like [...] type [casts] in other languages, but [perform] no special checking
> or restructuring of data [25].

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

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

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

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

However, this *angle-bracket* syntax is not supported when using TypeScript with *JSX*[2] [25], making the `as` syntax preferable.

3.1.5 Ambient Type Declarations

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

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

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

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

---

[1]TODO: Explain API

[2]"JSX is an embeddable XML-like syntax [...] meant to be transformed into valid JavaScript [which] came to popularity with the React framework, but has since seen other applications as well." [52]

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 [68].

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

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

3.1.6 Structural Types

Types in TypeScript are structural [66, p. 11], meaning that the type checker looks at members of an object, to ensure type compatibility, while other major languages, such as C# or Java, use nominal type systems [68]. Program 3.1 gives an example that would fail in a nominally typed language, but is possible in TypeScript.

3.1.7 Classes

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

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

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

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

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

```
class Person {
```

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

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

3.1.8   Enums

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

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

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

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

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

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

However, if the enumeration is declared as constant, the compiler will look up the numeric value and will insert it directly into the source code, before entirely removing its definition [46].

3.1.9   Namespaces

In TypeScript, namespaces provide a possibility to encapsulate code. They were previously referred to as *internal modules*, but have since been renamed to avoid confusion with native *modules* of the EcmaScript standard, previously denoted as *external modules* in TypeScript [56]. Code within a namespace only expose explicitly exported parts.

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

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

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

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

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

3.1.10 Parameter Default Values

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

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

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

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

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

3.1.11 Future JavaScript

3.2 Flow

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

---

[3]TODO: explain void 0

[4]https://flow.org

[5]https://code.facebook.com/projects/

3.3   Other Supersets

3.4   Comparison

3.5   Existing Runtime Projects

Chapter 4

Theoretical Approach

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

4.1 Initial Situation

This section is dedicated to point out, why runtime type checks are of importance. Also current approaches and solutions are discovered, to get an overview of the techniques available and steps required to make a TypeScript project more resistant to failure.

4.1.1 Undetectable Errors

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

Compiler Analysis Circumvention

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

> [...] a powerful way to work with existing JavaScript, allowing you to gradually opt-in and opt-out of type-checking during compilation. [25]

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

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

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

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

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

Polymorphism

Polymorphism in combination with type assertions can lead to errors at runtime. While the TypeScript compiler does check type compatibility in general, it allows casts to a type that would be assignable to the variable's type being asserted. 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 `Cat` is created and assigned to a variable, 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 the 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 will be thrown at runtime, saying that "dog.miow is not a function".

Untyped JavaScript Libraries

If a JavaScript project was written in native syntax, TypeScript can't infer the type information needed, to perform sufficient static type analysis. In this cases, type declarations may be provided manually, as discussed in the next section. *Definitely Typed*[1] provides a collection of such type definitions for JavaScript libraries [38]. 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 with `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. When creating more sophisticated type definitions, every change of the library has to be reviewed, to manually align its declaration files.

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 that 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. If, however, declaration files are created manually for a JavaScript library, not written in TypeScript, there is a chance, that they contain mistakes or that they are not be 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* provides 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 it existing on the object.

---

[1]http://definitelytyped.org

Program 4.1: A JavaScript function without type checks.

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

Erroneous API Responses

When making use of an external API, the response should be of a given structure, known to the consuming program. An interface may be created, to describe the receiving object and to statically check its usage. If the format of the response changes unexpectedly, it is impossible to reveal this change at compile time, as the type checker relies on the interface provided. Runtime checks have to be added manually, to make sure, the response conforms to the expected structure. Otherwise an error may be raised at the execution of the program, or behavior may be different than expected.

4.1.2   Runtime Type Checks

The situations discussed in the previous sections of this chapter raised the concern of negligently or unknowingly opting out of static type checks for specific parts of the code or of providing insufficient or wrong type declaration, which may cause the compiler to miss incompatible types. These situation may be discoverable at runtime, if runtime type checks would be employed alongside compile time checks. As already mentioned in chapter 3, type annotations are stripped out for the compiled JavaScript program and no additional code is introduced to add checks at runtime. The removal of types is intended and is defined in the design non-goals of the language [31]:

> [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.

Anyway, runtime type information and checks can be useful in several situations. For example they can give more accurate error messages during development and can flag issues, that 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 [48, 55, 61]. Regardless of the demand, these features won't be added, as they are out of scope for TypeScript [30, 50]. Currently manually added type checks are required to identify and easily trace errors during development. Program 4.1 shows a simple JavaScript function, which only has three lines of code, while program 4.2 shows a function with the same outcome, but added type checks—also in native JavaScript— that now requires 13 lines of code. While these examples outlined the verification of primitive types, like a number or an array, inspecting an object if it is of a given type 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 [51]", but interfaces or type alias are removed by the TypeScript compiler, therefore such a verification is not possible. To get around this issue, Rozentals describes three different techniques to employ type checks for the runtime engine:

Program 4.2: The JavaScript function from program 4.2 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 }
```

- **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, as 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 suitable to categorize an object as a type, as the same name may also be used for a different constructor function or an anonymous function has been used. 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]. For example every *Person* object must implement a *getName* method. By verifying if a function with this name is available on the object, it may be considered as the given type. This gets closer to TypeScript's structural types (see sec. 3.1.6).
- **Interface checking with generics:** For every interface there is a class, which holds its 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 one, but it introduces a pattern which is more readable and maintainable.

Another mechanism is to use *decorators*[2], a JavaScript language feature proposal, which is currently at stage two [32], meaning that it is still a draft and not yet in the specification [67]. They can, however, already be used with TypeScript or tools like *Babel*[3] [24, 37]. The solution used in [62], that 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.

---

[2]https://tc39.github.io/proposal-decorators
[3]https://babeljs.io

4.2 Desired Result

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, 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, as the number is assigned as *any* type to a variable, that 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 was 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, that 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 section 4.1 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. 4.1.1 as possible are reported during execution time, it may be beneficial to automate the inclusion of runtime type checks. In order to provide generated runtime type checks for a TypeScript project, cases have to be collected, where such checks 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.

Program 4.3: The enum `enum Enum { A }` compiled to JavaScript. [46]

```
1 var Enum;
2 (function (Enum) {
3     Enum[Enum["A"] = 0] = "A";
4 })(Enum || (Enum = {}));
5 var a = Enum.A;
6 var nameOfA = Enum[Enum.A]; // "A"
```

4.3.2   Variable Declarations and Assignments

If a variable is 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. 3.1.4). To inspect if an assertion is valid, the same checks are performed as for variable assignments. Values asserted as *any*, as discussed in section 4.1.1, can be ignored, as they would always pass.

4.3.4   Functions

There are different types of functions that need to be distinguished: classic function declarations, function expressions—which can be used for every case expressions are allowed—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 the return type and its parameters—including their types—which can also be optional. If the function is called, the values passed as parameters, as well as the returned value, have to be checked. Additionally a function may make use of generics, to define parameter or return types [47] 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`.

4.3.5   Enums

Enumerations (see sec. 3.1.8) are compiled to self executing functions, that initialize a corresponding object, by TypeScript [46] (see prog. 4.3). To enable type checks for the runtime, the enum has to be described with their members. In the case of program 4.3, it has exactly one member with the value of the number zero.

4.3.6 Classes

For classes a multitude of cases, that require 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, checks should be performed, to ensure that it correctly implemented 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` would be of type string. 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, a type query result may be used as value, but it can only distinguish between six value types at runtime (see tab. 2.1 in sec. 2.2.2). If a variable is annotated with a type query, the type of a 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 this packages are itself 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, among other, interfaces, type aliases and class reflections have to be collected and has to be made available to the runtime code.

4.3.9 Ambient Declarations

If globals for a project are not available in the development environment, 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 actual implementations:

```
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 declaration should be collected and made available to the runtime the same way as externals.

4.4 Required Steps

After having clarified the current situation (see 4.1), outlined the desired result of the project (see 4.2), and pointed out conditions where transformations should take place

(see 4.3), specifying the steps required to accomplish automated runtime type check generation, is a logical consequence:

1. Set 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 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 will be described in more detail in the following sections. While trying to give a theoretical understanding of the concept of the project, no technical details will be included at this point.

4.4.1 Configuration

As different projects have to meet 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 compiler itself[4], as well as adjustments for *ts-runtime*. While the project of this thesis is not intended to be a complete drop-in replacement for TypeScript, it should honor the options of the development environment. This settings especially include the ECMAScript version of the resulting program, if individual or a concatenated single file should be emitted, the module system to use, as well as the write location of the output [34].

4.4.2 Read Source Files

The starting point of the transformation process will 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, described in the previous section. All referenced files from the entry points should be followed and read in as well, 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 code provided is syntactically correct. This should prevent *ts-runtime* to fail, because of syntax errors in the source. If the code contains syntactic errors, it is impossible to perform any further analysis. Therefore the process should be stopped immediately, to prevent the occurrence of unexpected behavior or results.

---

[4]https://www.typescriptlang.org/docs/handbook/compiler-options.html

4.4.4 Abstraction

In order to perform special checks and transformations to the original code, abstracting the source will be beneficial. A suitable data structure may be an abstract syntax tree (i.e., AST), as described in section 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 Static Type Checks

Another important aspect would be, to already perform static type checks and to reject the input from further processing, if type incompatibilities could be detected. This has the advantage of flagging issues to developers early and to align with the usual behavior of a TypeScript compilation. If 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, there should be a 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.

4.4.6 Scan Abstraction

Once the provided source files can be considered syntactically correct, are represented in an abstract data structure, the type information has to be extracted for future modifications to the code. Not only should it be possible to obtain the explicitly set type of an AST node, also the implicitly inferred type should be receivable. In addition it should be should be practicable to follow a type reference's type, for further processing. To draw these connection between different parts of the program, the abstract syntax tree may be walked through entirely ahead of any modifications. This should ensure, that important data—like type information, AST node relations and declared identifiers—is not lost or becoming inaccessible while transformations are being applied.

4.4.7 Transformations

Cases where modification will have to take place, to reflect all required type information, as well as introducing runtime type checks based on these reflections, were already identified in section 4.3. All required data to perform extensive transformations to the AST should already have been prepared by the previous steps of the process. This enables *ts-runtime* to proceed with substituting and altering abstract syntax tree nodes. The tree should be walked from bottom to top, for changes applied to the very bottom of the AST does not interfere with modifications that have to take place on a higher level. This should guarantee, that transformations of high level tree nodes already include low level mutations. Furthermore, nodes that were replaced, or their changes would result in not being able to obtain required informations, should be mapped to the original parse tree node. This will make sure, that the initial state of the node can be retrieved at any time.
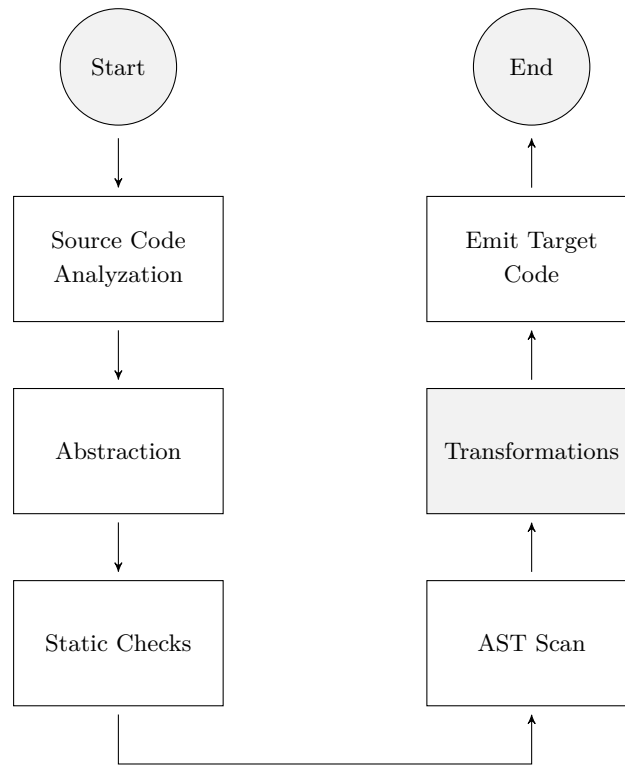
Figure 4.1: Conceptual procedure of applying transformations to a TypeScript project.

4.4.8 Target Code Emit

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

4.5 Summary

To achieve automatically generated runtime type reflections and checks, a series of steps—some more complex than others—have to be carried out. To give a better understanding of the conceptual procedure, figure 4.1 illustrates the idea of the transformation process. In a next step, the theoretical approach will be evaluated, and the program technology and architecture has to be defined. Also technical peculiarities and limitations have to be identified, to provide a solid base for the implementation.

Chapter 5

Implementation

After elaborating situations where runtime errors may occur in the previous chapter (see sec. 4.1.1), even with preceding static type checks by the TypeScript compiler, a program will be implemented, that should catch those cases while the compiled JavaScript program is running. 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 will be implemented in TypeScript, while the compiled program will be executed in a *Node.js*[1] environment. It will be published on the *npm* (i.e., node package manager) registry[2], a "[...] public collection of packages of open-source code for Node.js [...] [21]", 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 as fast as possible with this project. This implies that both, an API (i.e., application programming interface) and a CLI (i.e., command line interface), should be provided. Furthermore, to create an application that efficiently achieves its goal, it will be 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, that provides functionality needed for the implementation, it should be 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 utilize 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

---

[1]https://nodejs.org

[2]https://www.npmjs.com

an API was exposed to enable abstract syntax tree transformations [26] and an issue preventing traversing the AST [59] was resolved with version 2.4 [27]. Not only the ability to modify the syntax tree may be useful for the project of this thesis, also other features may be beneficial. As the project of this thesis will make use of the compiler API later in this chapter, some parts of it will be described below.

Compiler Components

To receive a runnable JavaScript program from a TypeScript project, a number of components contribute to the TypeScript compiler: [66, p. 251]:

- **Scanner:** The scanner is responsible for the tokenization of the source code and is controlled by the parser [66, p. 260].
- **Parser:** After a source file is tokenized, the parser creates an abstract syntax tree out of it [66, p. 263].
- **Binder:** In this part of the compiler, connections between nodes of the AST are created through *symbols* [66, p. 267].
- **Checker:** The checker is the largest part of the TypeScript compiler and performs static type checks on the source files [66, p. 282].
- **Emitter:** The emitter translates the TypeScript syntax tree of the source files to plain JavaScript [66, p. 286], based on the compiler options.

These components don't have to be triggered individually when using the compiler API and a wrapper is provided, named *Program*. It holds the options and source files of the current compilation [66, p. 254] and provides access to the *Checker* [66, p. 282] and *Emitter* [66, p. 286].

Compiler Options

When starting a a compilation through the TypeScript compiler API, a multitude of options [34] 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 a single or multiple 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 to a new *Program*.

Node

The abstract syntax tree, 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.*

Symbol

A 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 these 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 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. Therefore it is possible to pass a *SourceFile* node to the printer, and getting back a string containing TypeScript code.

5.1.2 Runtime Type System

A multitude of libraries are available that aim to provide a runtime type system for JavaScript and several of them were evaluated to use with *ts-runtime*. Libraries that did not seem to be maintained any more were not considered.

ObjectModel

ObjectModel[3] is an extensive type system, which "[...] intends to bring strong dynamic type checking to [JavaScript] web applications [60]". While being actively maintained and a detailed documentation is available, several features that are required to reflect TypeScript's static type system are not included in this library, making it not entirely suitable for the use with the project of this thesis.

tcomb

The project *tcomb*[4] argues to "[...] check the types of JavaScript values at runtime [29]". Probably one of the most famous runtime type checking libraries for JavaScript, with more than 1300 stars on GitHub [29], it is again intended to be used for JavaScript code. Considerations especially 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 [...] [28]". While it did look promising to be utilized,

---

[3]https://github.com/sylvainpolletvillard/ObjectModel
[4]https://github.com/gcanti/tcomb

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*[5] may evolve over time, a transition of *ts-runtime* to use it at a later point is possible.

runtypes

The *runtypes*[6] library is a fairly young project, which wants to provide "[r]untime validation for static types [36]". 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 does not include parameter and return type compatibility.

flow-runtime

The *flow-runtime*[7] project states to be "[a] runtime type system for JavaScript with full Flow compatibility [58]". As Flow and TypeScript have a lot of similarities in syntax and features (see sec. 6.3) this library seems to be most suitable for the thesis project and to reflect the static type system of TypeScript as close as possible. Additionally *flow-runtime* has a separate package, that generates type checks for Flow projects [57], which proves that a great number of cases for Flow syntax are implemented in this package.

5.2   Architecture

In this section, the architecture for the application will be designed, that should already outline how the program will operate and define some of the components that will be needed:

5.2.1   Central Element

The transformation process is a sequential process, as defined in section 4.4, which already suggests, that a central element will be needed, coordinating all different steps that should to be executed. It will be 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 will have to interpret different settings, including options for the TypeScript compiler. Also it will have to react to possible errors and will have to handle them adequately.

5.2.2   Components

Specific tasks will be dedicated to separate components, which contain the logic for a selected purpose, to keep the project extensible and maintainable.

---

[5]https://github.com/gcanti/io-ts
[6]https://github.com/pelotom/runtypes
[7]https://github.com/codemix/flow-runtime

Options

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

Event Bus

Some of the components of the application will have access to other components and their API, whereas other parts of the program won't know the state of the transformation process. It might be necessary to observe, or to get notified, if a condition changes, where an event bus will be of advantage. Consequently, the event bus (i.e., bus) should be accessible globally.

Scanner

Not to confuse with the scanner of the TypeScript compiler (see sec. 5.1.1), this component of the thesis project should scan the abstract tree of the source files. Ambient and external declarations should be identified, that won't 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 should be stored, to avoid duplicate identifiers when introducing new variables during the insertion of runtime type checks.

Mutators

For every situation where runtime type checks should be generated (see sec. 4.3), a mutator should exist, that performs the modification or substitution of the AST node.

Factory

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

Context

As not all components of the application are connected to each other, the context should provide a centralized gateway to information, that might be required in the mutators or the factory. It should, among other things, have 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, should be collected in the utilities of *ts-runtime*. It should be available from any location of the project.
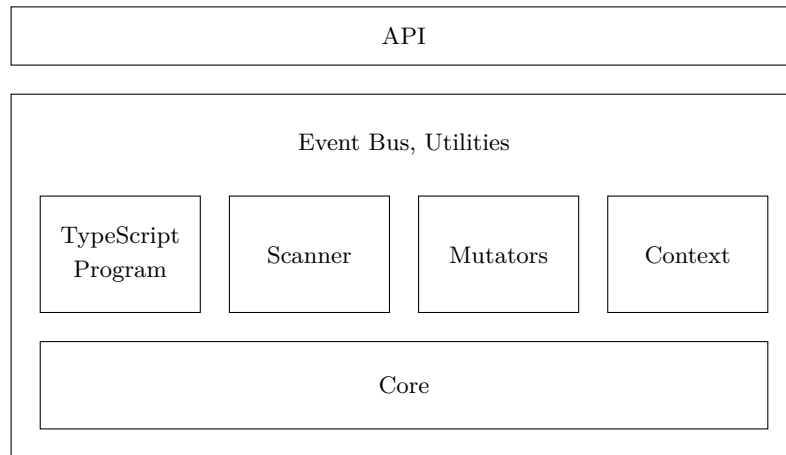
Figure 5.1: Component architecture of the thesis project.

5.2.3   Outline

After the main parts of the program have been defined, it is possible to draw connection 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 will be 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 the project, as well as the application's components, will be described. The main focus will be on challenges that had to be

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 API.

```
1 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 }
```

overcome, and on the interconnections of the different parts of the program.

5.4.1   Transformer

The *Transformer*, located in `src/transformer.ts`, is the core of the thesis project and exposes three methods, that may be utilized via the project's API:

- **getOptions:** This function accepts an object as parameter, that 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 as well. Nothing is returned, but 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 files from disk, which requires a file system to be present. On the contrary, this method accepts a list of file reflections, that must include the passed entry files, as well as all modules referenced, recursively. This enables the application, to act without a real file system. Also the target code is not persisted, but a list of file reflections, containing the compilation result, is returned.

Program 5.1 shows a, reduced to its essentials, version of the `transform` function. It is not fully functional, but should give an idea of the program flow, also depicted in figure REF!!!. On line 6, a variable `transformer` is passed to a function from the TypeScript compiler API. This identifier references a function, that visits every node of the AST from all source files of the TypeScript program, which are passed to the mutators of *ts-runtime*.

Chapter 6

Evaluation

Chapter 7

Summary and Outlook

References

Literature

[1]   Luca Cardelli. "Type Systems". In: *Computer Science Handbook*. Ed. by Allen B. Tucker. 2nd ed. CRC Press, 2004. Chap. 97, pp. 97-1–97-41 (cit. on pp. 2–5).

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

[3]   Stevens Fenton. *Pro TypeScript. Application-Scale JavaScript Development*. 2nd ed. Apress, Feb. 2017 (cit. on p. 17).

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

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

[6]   Ross Harmes and Dustin Diaz. *Pro JavaScript Design Patterns. Application-Scale JavaScript Development*. 1st ed. Apress, 2008 (cit. on p. 28).

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

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

[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*. 1st ed. Expert's voice in Web development. California, USA: Apress, 2014 (cit. on p. 4).

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

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

[13]   Nathan Rozentals. *Mastering TypeScript*. 1st ed. Packt Publishing, Apr. 2015 (cit. on pp. 27, 28).

[14]   Nathan Rozentals. *Mastering TypeScript*. 2nd ed. Packt Publishing, 2017 (cit. on p. 17).

[15]   Kyle Simpson. *Scopes & Closures*. 1st ed. You Don't Know JS. Sebastopol, California, USA: O'Reilly Media, Mar. 2014 (cit. on pp. 10, 11).

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

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

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

[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. Springer, 2014 (cit. on p. 15).

Online sources

[21]   *About npm*. URL: https://www.npmjs.com/about (visited on 07/15/2017) (cit. on p. 35).

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

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

[24]   *Babel decorators transform*. URL: https://babeljs.io/docs/plugins/transform-decorators/ (visited on 07/03/2017) (cit. on p. 28).

[25]   *Basic Types - TypeScript*. URL: https://www.typescriptlang.org/docs/handbook/basic-types.html (visited on 05/08/2017) (cit. on pp. 17–19, 24).

[26]   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. 36).

[27]   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. 36).

[28]   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. 37).

[29] Giulio Canti. *tcomb - Type checking and DDD for JavaScript*. URL: https://github.com/gcanti/tcomb (visited on 07/15/2017) (cit. on p. 37).

[30] 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 p. 27).

[31] 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 p. 27).

[32] *Class and Property Decorators*. URL: https://github.com/tc39/proposal-decorators (visited on 07/03/2017) (cit. on p. 28).

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

[34] *Compiler Options - TypeScript*. URL: https://www.typescriptlang.org/docs/handbook/compiler-options.html (visited on 07/04/2017) (cit. on pp. 32, 36).

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

[36] Tom Crockett. *runtypes - Runtime validation for static types*. URL: https://github.com/pelotom/runtypes (visited on 07/15/2017) (cit. on p. 38).

[37] *Decorators - TypeScript*. URL: https://www.typescriptlang.org/docs/handbook/decorators.html (visited on 07/03/2017) (cit. on p. 28).

[38] DefinitelyTyped. *The repository for high quality TypeScript type definitions*. 2017. URL: https://github.com/DefinitelyTyped/DefinitelyTyped (visited on 07/02/2017) (cit. on p. 26).

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

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

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

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

[43] *ECMAScript Language Specification. ECMA-262 6th Edition*. June 2015. URL: http://www.ecma-international.org/ecma-262/6.0/ECMA-262.pdf (visited on 04/23/2017) (cit. on pp. 5–9, 14).

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

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

[46]   *Enums - TypeScript.* URL: https://www.typescriptlang.org/docs/handbook/enums
.html (visited on 05/08/2017) (cit. on pp. 21, 30).

[47]   *Generics - TypeScript.* URL: https://www.typescriptlang.org/docs/handbook/gene
rics.html (visited on 07/03/2017) (cit. on p. 30).

[48]   *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 p. 27).

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

[50]   Mohamed Hegazy. *GitHub Issue #3015. also emit type arguments with –
emitDecoratorMetadata. Comment #128149650.* Aug. 2015. URL: https://githu
b.com/Microsoft/TypeScript/issues/3015#issuecomment-128149650 (visited on
07/02/2017) (cit. on p. 27).

[51]   *instanceof - JavaScript.* 2017. URL: https://developer.mozilla.org/en-US/docs/W
eb/JavaScript/Reference/Operators/instanceof (visited on 07/03/2017) (cit. on
p. 27).

[52]   *JSX - TypeScript.* URL: https://www.typescriptlang.org/docs/handbook/jsx.html
(visited on 05/08/2017) (cit. on p. 19).

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

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

[55]   Jed Mao. *GitHub Issue #1573. Runtime type checking.* Dec. 2014. URL: https://git
hub.com/Microsoft/TypeScript/issues/1573 (visited on 07/02/2017) (cit. on p. 27).

[56]   *Namespaces - TypeScript.* URL: https://www.typescriptlang.org/docs/handbook/n
amespaces.html (visited on 05/08/2017) (cit. on p. 21).

[57]   Charles Pick. *babel-plugin-flow-runtime.* URL: https://github.com/codemix/flow-r
untime/tree/master/packages/babel-plugin-flow-runtime (visited on 07/15/2017)
(cit. on p. 38).

[58]   Charles Pick. *flow-runtime - A runtime type system for JavaScript with full Flow
compatibility.* URL: https://github.com/codemix/flow-runtime/tree/master/packag
es/flow-runtime (visited on 07/15/2017) (cit. on p. 38).

[59]   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. 36).

[60]   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. 37).

[61]   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 p. 27).

[62]   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. 28).

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

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

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

[66]   Basarad Ali Syed. *TypeScript Deep Dive*. 2017. URL: https://www.gitbook.com/d ownload/pdf/book/basarat/typescript (visited on 05/08/2017) (cit. on pp. 12, 17, 18, 20, 36).

[67]   *The TC39 Process*. URL: https://tc39.github.io/process-document/ (visited on 07/03/2017) (cit. on p. 28).

[68]   *Type Compatibility - TypeScript*. URL: https://www.typescriptlang.org/docs/hand book/type-compatibility.html (visited on 05/08/2017) (cit. on p. 20).

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

[70]   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. 17).

Check Final Print Size

— Check final print size! —

width = 100mm
height = 50mm

— Remove this page after printing! —