Typescript: Gradually typing JavaScript

Daniel Schneider

Ludwig-Maximilians-University, Geschwister-Scholl-Platz 1, 80539 Munich, Germany
poststelle@verwaltung.uni-muenchen.de

Abstract. "TypeScript is an open-source language which builds on JavaScript, one of the world's most used tools, by adding static type definitions."[2] This paper will discuss the benefits TypeScript can give developers in terms of software quality and security. It will highlight examples of the main features that are part of the TypeScript standard and how those can be leveraged to convert a previous JavaScript program into a safer TypeScript program.

Keywords: TypeScript · JavaScript · Type safety · Gradual typing

1 Introduction

Programing is the act or process of planning or writing a program[3]. The general goal of programing languages is to make it easier for humans to interact with computers in a comprehensible way. This results in programing languages that increasinly abstract away complex mechanisms to speed up or simplify program development. A relatively recent phenomena is the complete removal of type annotations within programing languages, whereby the compiler completely infers variable types. This gives the developer a simpler workflow by leaving out the mental step of thinking about typings. Although this may first seem to be a good thing, it has become increasingly apparent, that bigger programs degrade in software quality if software quality is not a priority. This can be mitigated by using typing systems with static type checking, but would require a complete rewrite in a type annotated and type checked language. One of the most popular scripting languages out there – JavaScript – does away with type annotations, which to the demise of big software projects, written in JavaScript, results in degrading software quality quite quickliy if not taken care of.
To remedy this fundamental issue of JavaScript, TypeScript was born. It allows for code type annotations and static type checking of the program at compile time, while still being able to infer untyped code areas to the point of compiling plain JavaScript code without any type annotations.

To get a better feeling for what benefits TypeScript can give us, we will have a look at a possibly problematic JavaScript snippet:

```
let anotherNumber = "13"
let numberList = [1,2,3,5,8]

function insertNumberAndReduce(list, newNumber){
  list.push(newNumber)
  return list.reduce((acc, val) => acc + val)
}

let result = insertNumberAndReduce(numberList, anotherNumber)
```

Listing 1.0.1. "Problematic JavaScript code snippet"

When executing this code snippet, we get the result of "1913" which obiously is false. To better understand what is happening here, we will follow this code in a more abstract manner:

1. String "13" gets inserted into the number list, consisting of integers
2. "insertNumberAndReduce" reduces all numbers by adding them
3. "13" is added on to the sum of all preceeding numbers in the numberList
4. Adding strings and numbers is not possible, therefore the sum of the numbers is cast into a string and both strings are concatenated
5. We recieve the result "1913"

Since there is no system that can check at compile time if what we are doing may not be intended, JavaScript tries its best to compute our code. Of course we could manually check if the parameters given to the function correspond to native JavaScript datatypes by using conditionals, but this would degrade code legibility, may impact runtime performance negatively and might be omitted by developers therefore also requiring thorough testing.

Claim. This issue could be easily prevented by keeping the developer from inserting a string into a list of numbers, that is otherwise only made up of integers

Let us now rewrite the example into TypeScript with the corresponding type annotations and see if we are able to compile this snippet successfully:

```
let anotherNumber = "13"
let numberList = [1,2,3,5,8]

function insertNumberAndReduce
(list: number[], newNumber: number): number {
  list.push(newNumber)
  return list.reduce((acc, val) => acc + val)
}

```

```
10 let result = insertNumberAndReduce(numberList, anotherNumber)
```

Listing 1.0.2. "Fixed TypeScript code snippet"

Note 1. We have added type annotations to the function definition such that the function will only accept a list of numbers, a additional number and will only return a value of type number

Problem 1. TypeScript will throw an error upon compilation, since the function call does not comply with the function definition and its type annotations.

2 Typescript in a nutshell

In this section we will have a look at the core principles of TypeScript.

2.1 Architecture

TypeScript itself is primarily a compiler that runs on preexisting TypeScript code. This compiler takes in TypeScript files and emits valid JavaScript which can then be executed by JavaScript runtimes such as Node.js or Deno.

The four building blocks of the TypeScript compiler:

- **Parser:** The parser parses the TypeScript code syntax into a Abstract Syntax Tree short AST.
- **Binder**: The binder links code modules to each other. It resolves dependencies and enables the additional importing of type definitions from other files.
- **Type Resolver**: The type resolver checks the code for type soundness and throws an exception for any type collisions, resulting in a failed compilation. This is the central part of the typescript compiler enabling static type checking.
- **Emitter**: The emitter then generates native JavaScript code from the previously generated AST, which can be executed by any capable JavaScript runtime.

2.2 History

To get to know TypeScript a bit better, we will have a quick look at the history

Typescript v0.8 was initially created by Microsoft in 2012 an published to the public.

In the following year they released v0.9 which brought with it a generics system for type annotations.

In 2014 Microsoft then released the first production-ready version v1.0 integrating the TypeScript compiler into their already existing Visual Sudio IDE platform, bringing TypeScript to a broader audience of developers that might want to try TypeScript.

Later that year, in July, a new compiler version was released with five times the performance of the old compiler, improving production implementaion.

In 2016 v2.0 was released, that tried to remedy a general programing language issue commonly called the "Billion-Dollar mistake"[1].

Later v3.0 was released in 2018 brining with it several smaller features like tuples.

The most recent version v4.0 was released 2020 and brings improvements to the existing Tuple system by adding Variadic Tuples.

2.3 Static type checking

Typescript if of it self is a compiler. It takes in TypeScript code and compiles it down to JavaScript while also doing static type checkint. Static type checking is the process of parsing the code into an AST (Abstract Syntax Tree) including type annotations. The types in the AST are then infered, if not existant, and are checked for soundness by the Type Resolver.

2.4 Gradual typing

TypeScript leverages gradual typing. Gradual typing is the principle of gradually adapting type annotations in your program, with the ability to complete leave out type annotations. This enables developers to gradually migrate over JavaScript projects to TypeScript without having to worry about code compatability. TypeScript will try to infer as many types as possible, but cannot guarantee complete type safety as long as there are sections in a program that are not type-annotated.

3 Features

We will have a look at the four most important features TypeScript brings with it. Knowing these features is enough to provide type safety to your code.

[1] The term "Billion-Dollar mistake" was first coined by Tony Hoare, who also created the ALGOL W language. Having implemented null referencing in that language he may have caused around a billion dollars of damage since 1965, due to errors, vulnerabilities and system crashes. In the JavaScript world the implementation of the primitive "null" datatype may have also caused high sums of damages due to a 25 year old JavaScript bug that causes the primitive "null" datatype to resolve to the type of "object"[1]

3.1 Interfaces and Types

Interfaces and Types are the most basic building blocks within TypeScript and allow us to define custom object and type schemas. First we will have a look at how Interfaces and Types are defined and referenced in code:

```
interface User {
  name: string
  email: string
  age?: number
}
// An equivalent to the interface can also be defined as a type
type User = {
  name: string
  email: string
  age?: number
}
```

Listing 3.1.1. "Interface and Type declarations"

Question 1. What is the difference between an Interface and a Type?

Solution 1. There are hardly any differences between Interfaces and Types. Types have the ability to define primitive types such as numbers and strings, while Interfaces can only defined object structures and types within. Types can not be merged with each other, while Interfaces have the capability to be merged, should the developer define several Interfaces with the same name. In general, Interfaces should be preferred to Types if they can be used interchangeably.

These Interfaces and Types can then be used in the following manner:

```
const user: User = {
  name: "John Doe",
  email: "john@doe.com"
}
```

Listing 3.1.2. "Interface and Type usage"

Note 2. The "age" key can be omitted, since we've made the key optional in the type definition by suffixing the key with a question-mark

3.2 Tuples

Tules give you the ability to type the structure of an array. This means that it is possible to define a specific type for each index of that array and enabling us to build tuples out of several other tuple building blocks, called variadic tuples. In the following example we can see how simple Tuples can be implemented:

```
// Type structure: <order>, <key>, <value>
type Detail = [number, string, number | string]

let userDetails: Detail[] = [
  [1, 'age', 22],
  [2, 'semester', 6],
  [10, 'name', 'John Doe']
]
```

Listing 3.2.1. "Tuple usage"

Note 3. Using a pipe symbol (|) between types resembles an OR operator. Therefore the type of that element can be either a number OR a string

By using Tuples, TypeScript can prevent wrong usage of arrays:

```
let userDetails: Detail[] = [
  [1, 'age', 22, 'a fourth element'],
  [2, 5, 6],
  ['some order', 'name', 'John Doe']
]
```

Listing 3.2.2. "Problematic Tuple usage"

Problem 2. TypeScript will throw an error upon compilation, since the Arrays within the 'userDetails' Array do not correspond to the 'Detail' type

Finally we also have the ability to create Tuples out of other Tuples. This is handy if one e.g. knows the first elements of an array but wants to dynamically define all following element types of that Array. These Variadic Tuples can reduce code complexity. Rewriting our previous working example into Variadic Tuples results in the following code:

```
// Type structure: <order>, <key>, <custom element types>
type Detail<T extends unknown[]> = [number, string, ...T]

let userDetails: Detail<[number | string]>[] = [
  [1, 'age', 22],
  [2, 'semester', 6],
  [10, 'name', 'John Doe']
]
```

Listing 3.2.3. "Variadic Tuples"

Remark 1. This Variadic Tuple system now lets us define the <custom element types> at a variable basis, giving us more freedom in reusing Types, while still forcing us to abide to general typing structures like the 'Detail' type.

3.3 Namespaces

Namespaces allow us to split Types into separate logical units. This also allows for equally named types without them interfering with each other.

```
namespace User {
  export interface Profile {
    email: string
  }
}

namespace Admin {
  export interface Profile {
    email: string
    roles: Role[]
  }
  interface Role {
    name: string
  }
}
```

Listing 3.3.1. "Variadic Tuples"

Note 4. The User and Admin namespace both have a Profile type. Although there are now two interfaces with the same name, they do not collide with each other since they exist in a different scope.

Note 5. Namespaces allow for private types that are only accessible within the namespace itself. Here, the Role type within the Admin namespaces is not exported, thus only types within the Admin namespace can reference it. This allows for better control over the typing API of your program by only making types visible that are intended to be used

To use a type within a namespace we access it through dot notation:

```
const userProfile: User.Profile = {
  name: "John Doe"
}
```

Listing 3.3.2. "Variadic Tuples"

3.4 Generics

Generics introduce an abstration layer to your typings. It allows for dynamic types depending on the use case. Let's have a look at how Generics are used:

```
function listify<T>(x: T): T[] {
  return [x]
}
```

Listing 3.4.1. "Variadic Tuples"

This listify function can take any type of parameter x and returns a list of the type of parameter x. This allows for fully dynamic typing without having to reimplement the same function for each type of parameter. It is also possible to manually control the type we pass into the function. Let's have a look at the following use-case:

```
listify<bool>(true)
```

Listing 3.4.2. "Variadic Tuples"

Here we are telling the listify function, that it should create a list of booleans, passing in one element of value "true"

4 Integration

4.1 Language server

5 Conclusion

Besides some configuration overhead and learning how to use TypeScript effectively, TypeScript can majorly improve software quality if used in the right way. Due to its gradual typing approach, preexisting JavaScript code need not be touched for you to get started with TypeScript. It can find bugs in your code and give you direct feedback while writing it, leveraging the power of the compiler and language-server. These bugs would normally have to be tested against in elaborate testing setups that may even take up more development time than simply implementing TypeScript in error prone program areas. All in all it can be said that TypeScript is a great tool to keep up and prevent degradation of the quality of large software projects.

References

1. Medium: Billion-dollar mistake(s)!, https://jagadeeshrampam.medium.com/billion-dollar-mistake-s-37620be56a12. Last accessed August 11, 2021
2. Typescript Homepage, https://www.typescriptlang.org/. Last accessed August 8, 2021
3. Thesaurus programing definition, https://www.dictionary.com/browse/programing. Last accessed August 9, 2021