TypeScript: An Analysis

Dave E. VanDerWeele

Lakeland University

Author Note

Computer Science program, Kellett School, Lakeland University. For Programming Languages class taught by instructor Ahilan Avisamy.

Author email (for errors, questions, criticism, or commentary): **weele.me@gmail.com** or **vanderweeled@lakeland.edu**.

Abstract

In this essay, I explore the phenomenon of TypeScript, which is a force within the world of JavaScript that must be reckoned with. It has major backing and adoption from industry titans, and it has ample feature sets all revolving around typing concepts to enhance developer experience and ease the way people work with JavaScript, which is perhaps simultaneously the world's most hated and most loved programming language.

Keywords: TypeScript, EcmaScript, JavaScript, compilation, transpilation, transcompilation, interpreted, source-to-source, programming languages

TypeScript: An Analysis

TypeScript is a popular language that is often touted as a superset of the juggernaut known as JavaScript or EcmaScript. Metaphors can often limit our understanding as much as expand them — yet it can make sense to understand TypeScript as a kind of dialect of EcmaScript if we are to take on a broad enough definition of the term dialect such that putatively separate natural languages like Swedish and Norwegian and Danish can be considered dialects of one another or within a single larger politically disunited language. As EcmaScript itself has many of its own versions and specifications, I prefer the latter version of the metaphor. Due to its growing status in industry, it is a worthwhile exercise to differentiate TypeScript from EcmaScript so that we may understand how it happened, where it is today, and what the future may hold for the language.

A Background and History

TypeScript is a compiled language and invention of Microsoft (Microsoft, 2016). It is in the eighth year of its lifespan (TypeScript Documentation), but that is not to say that it is adolescent. The oft-used word “superset” in descriptions of the relationship of TypeScript to EcmaScript belies its dualistic nature. Before we discuss the particulars of TypeScript, we must first discuss some of the specifics of its syntactic base and compilation target: EcmaScript.

The syntax of the TypeScript language is a sugary supplement to EcmaScript (Microsoft, 2016), or alternatively a set of supplements if you consider the multi-faceted nature of EcmaScript. EcmaScript has its own syntax that has evolved over time through many different specifications. New language features for EcmaScript necessarily take years to manifest in the

real world after implementation in official specifications as different EcmaScript engines, many with massive consumer and industrial rates of adoption, adopt new features at different paces and consumers engage with and update software that contains EcmaScript engines at different rates. Commonly, there is a syntactic lag between the latest EcmaScript features and what will actually end up working out in the real world. TypeScript uses a compilation process to allow for the developer experience of the latest and even non-existent EcmaScript features while simultaneously ensuring deployability of the final code product to the real world; indeed, these features are by design optional so that even valid, pure EcmaScript programs will compile successfully (TypeScript Documentation).

TypeScript does not implement compilation in the traditional source-to-binary sense of compilation (e.g. as with C++), but source-to-source compilation or transcompilation. Traditional source-to binary compilation works in two stages: 1) the source code is translated to a binary program, and 2) that binary program is run and receives input during its execution (Aho et al, 2007). Debate over whether EcmaScript itself is a compiled or interpreted language often suffers from an insufficient understanding of the distinction between compilation and interpretation (Simpson, 2020). The source code of an interpreted language indeed does initially go through a translation step in the execution environment whereby it is transformed into a lower level form of code — commonly referred to as bytecode — which is subsequently passed to a virtual machine which both executes the bytecode and receives input on its behalf (Aho et al, 2007). The conclusion then that we must draw about the relationship of TypeScript and EcmaScript is that code is destined to go through two translation processes before it is ever executed: once during the development compilation process and once in the execution

"interpretation" environment. This translation heavy approach to software engineering clearly exists in service of developer experience.

There are limits to the kind of superset that TypeScript can ever be for EcmaScript, and this aspect bears mentioning even if such cases might commonly be relegated to the fringe. Other source-to-EcmaScript transcompilers exist, such as the famous Babel tool, which interestingly can be used as an alternative compiler to the native TypeScript compiler (TypeScript Documentation) among other things. There are novel features introduced in newer versions of EcmaScript, such as the Proxy API, that logically cannot be polyfilled or transpiled backwards to ES5 or earlier versions of EcmaScript (Babel). This logical and semantic limitation necessarily applies to and limits the potential of the TypeScript compiler as well if, for instance, you are utilizing EcmaScript's Proxy API in a TypeScript program and trying to compile to ES5 or earlier. Therefore, it must be noted that despite TypeScript's attempts to improve on what is offered out of the box with EcmaScript, TypeScript development teams will always be required to stay in touch with the nature and versioning schemes of native EcmaScript.

Key Language Features

TypeScript, like EcmaScript, is a multi-paradigm language. Containing all of the features of EcmaScript, such as the prototypal inheritance flavor of object-oriented programming as well as support for powerful functional programming features like closure, higher-order functions, and recursion, TypeScript also adds a number of features that are absent in EcmaScript.

Unironically, TypeScript adds *typing* features that are absent in EcmaScript. That is not to say that vanilla EcmaScript has no types. EcmaScript is dynamically typed, where variables may at any time contain any type, and subject to a heavy shroud of type coercion that is often confusing to developers new to EcmaScript (MDN Web Docs); so it is necessarily also with TypeScript.

```
let foo = 42;    // foo is now a number
foo     = 'bar'; // foo is now a string
foo     = true;  // foo is now a boolean
```

Above: Dynamic Typing in vanilla EcmaScript (MDN Web Docs).

TypeScript adds optional typing features such as explicit, static declarations of primitive types for variables (TypeScript Documentation).

```
let isDone: boolean = false;
```

Above: explicit primitive types in TypeScript (TypeScript Documentation).

There is also support for providing types for arrays (which in EcmaScript are actually more like contiguous arrays or objects with gaps that behave kind of like vectors) as well as an implementation of tuples with statically typed slots; while I cannot comment on how they are in the end transpiled into EcmaScript, I can say that this typing looks very different (in perhaps a more flexible way) from the typing offered by the TypedArrays in native EcmaScript, which are powered by the raw memory of ArrayBuffers and DataViews (MDN Web Docs).

```
let fruits = ['Apple', 'Banana']

console.log(fruits.length)
// 2
```

Above: EcmaScript array, each slot dynamically typed (MDN Web Docs).

```
let list: Array<number> = [1, 2, 3];
```

Above: TypeScript statically typed array (TypeScript Documentation).

```
// Declare a tuple type
let x: [string, number];
// Initialize it
x = ["hello", 10]; // OK
// Initialize it incorrectly
x = [10, "hello"]; // Error
```

```
Type 'number' is not assignable to type 'string'.
Type 'string' is not assignable to type 'number'.
```

Above: TypeScript statically typed tuples (TypeScript Documentation).

In a sense, these explicit typing feature sets of TypeScript can enable a developer experience in which there is a reduction in the amount of time spent debugging issues rooted in type coercion confusion.

TypeScript additionally offers “duck typing” for objects through its Interface feature (TypeScript Documentation).

```
interface LabeledValue {
  label: string;
}

function printLabel(labeledObj: LabeledValue) {
  console.log(labeledObj.label);
}

let myObj = { size: 10, label: "Size 10 Object" };
printLabel(myObj);
```

Above: TypeScript interface utilizes duck typing at compile time to check that objects passed to *printLabel* at the very least have a member called *label* of type *string*.

TypeScript also implements traditional features of class-based object-oriented programming, such as truly private class fields or protected class fields (TypeScript Documentation), which EcmaScript did not roll out even with its implementation of the syntactically sugary *class* keyword; actually native EcmaScript cannot have objects with truly private fields (even though you can restrict them in a few limited ways; that's changed with recent EcmaScript specifications, but it will be a while before a majority of web browsers in the real world support it) due to its prototypal inheritance scheme, and instead such a feature must be "simulated" or attained by way of closure and functions (it is a matter of speculation for me but I suspect the TypeScript compiler may indeed utilize functions and closure in EcmaScript to implement its private class fields, depending on which version of EcmaScript is targeted by the TypeScript compiler).

```
class Rectangle {
  height = 0;
  width;
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

Above: EcmaScript class is syntactic sugar built on EcmaScript functions and prototypes, with all class members necessarily public (MDN Web Docs).

```
class Animal {
  private name: string;

  constructor(theName: string) {
    this.name = theName;
  }
}

new Animal("Cat").name;
```

```
Property 'name' is private and only accessible within class 'Animal'.
```

Above: a TypeScript class with an explicitly defined *private* member (TypeScript Documentation).

Another traditional object-oriented programming feature implemented by TypeScript is enums (TypeScript Documentation), which developers with a background in C++ might welcome.

```
enum Direction {
  Up = 1,
  Down,
  Left,
  Right
}
```

Above: TypeScript enum defined to start at 1 instead of default 0 (TypeScript Documentation).

Features absent from TypeScript are certain advanced Object-Oriented Programming concepts, like multiple inheritance, although it is questionable whether such features are really necessary (considering how far abstracted even vanilla EcmaScript is from bare metal) and whether native EcmaScript is even capable of emulating them. As far as concurrency, TypeScript does not add anything to EcmaScript's native features for asynchronous and multi-threaded programming.

Industry

Naturally, the compiled nature of TypeScript makes it a bit better than EcmaScript for tooling such as IDEs as well as Test Driven Development. This, along with its backing from Microsoft and the ability to target via compilation EcmaScript versions that are present on virtually every consumer electronic device in the world, is likely the reason for its widespread adoption in industry. Industry titans like Slack, AirBNB, and Google use it (TypeScript Documentation).

Key Resources

- https://www.typescriptlang.org/
    - https://www.typescriptlang.org/play
    - https://www.typescriptlang.org/docs
- https://stackoverflow.com/questions/tagged/typescript
- https://github.com/microsoft/TypeScript

Summary

It is clear that TypeScript has caught on and will continue to catch on in industry. EcmaScript is not without its warts, but it is a beautiful monster with its own unique learning curve. As one of the largest languages in the world, there is always a new influx of and demand for developers that can work in EcmaScript environments. TypeScript's massive contributions to developer experience greatly ease the transition into the world of EcmaScript, even if it is at times disappointing to see the embrace of TypeScript come from a place of misunderstanding of EcmaScript's quirky features like prototypal inheritance, closure, and type coercion.

References

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Introduction. In *Compilers: Principles, Techniques, & Tools* (2nd ed., pp. 1-3). Boston, MA: Pearson Addison-Wesley.

Babel ES6 Feature Overview. (n.d.). Retrieved November 09, 2020, from http://hzoo.github.io/babel.github.io/docs/learn-es2015/

MDN Web Docs. (2020, October 29). Retrieved November 09, 2020, from https://developer.mozilla.org/en-US/

Microsoft. TypeScript Documentation. (n.d.). Retrieved November 06, 2020, from https://www.typescriptlang.org/docs/

Microsoft. (2016, January 19). TypeScript Language Specification. Retrieved November 06, 2020, from https://github.com/Microsoft/TypeScript/blob/730f18955dc17068be33691f0fb0e0285ebbf9f5/doc/spec.md

Simpson, K. (2020, March 27). Getify/You-Dont-Know-JS. Retrieved November 06, 2020, from https://github.com/getify/You-Dont-Know-JS/blob/2nd-ed/get-started/ch1.md

Course Evaluation Screenshot: