FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Michal Pácal

Object Usage Analyser for TypeScript

Department of Software Engineering

Supervisor of the bachelor thesis:	Mgr. Ing. Robert Husák
Study programme:	Computer Science
Study branch:	Programming and software development

Prague 2023

I want to thank my supervisor for the time spent helping and supporting me during the work on this thesis.

Title: Object Usage Analyser for TypeScript

Author: Michal Pácal

Department: Department of Software Engineering

Supervisor: Mgr. Ing. Robert Husák, Department of Software Engineering

Abstract: During a programming change task, developers often need to understand where a certain variable or object is referenced and how it is used in order to implement the change.

For languages TypeScript and JavaScript the only available tools to find these points of interest are tools to find variable and property references and to display a call tree. These tools, however, start losing viability in larger codebases, where they can return hundreds or even thousands of results sorted only by the source file names.

In this thesis, we have developed an extension for Visual Studio Code that finds these references and categorise them into groups based on usage. It can also perform similar analysis on call arguments based on the position of the argument. We evaluated our solution and showcased it in several real-world use cases.

Keywords: Static analysis, TypeScript, Visual Studio Code

Contents

Chapter 1

Introduction

During programming, developers often need to understand and navigate existing code. When modifying existing code they also often need to find places that use or modify a variable or how is its value passed throughout the program. This leads to questions such as "Where is this method called or type referenced?", "Where are instances of this class created?", or "What data is being modified in this code?". [1]

Modern development environments provide many tools and overviews to find occurrences of a variable or a property, [1] but those tools are usually don't provide any context or categorisation of the occurrences. Example of this are tools like 'Find All References', or a simple text search. This can reduce the usefulness of these tools in larger projects, where they can return hundreds or even thousands of results that the programmer needs to go through manually.

In this thesis, we will focus on this problem in the context of the programming language TypeScript. [2] The most popular editor for this language is Visual Studio Code. As both are created by Microsoft, the team creating it and the team behind the TypeScript compiler closely cooperate, giving it an edge over other editors.

1.1 Visual Studio Code

Visual Studio Code[1] is a modular IDE developed by Microsoft, designed to be easily extensible to work with many programming languages. [3] It is also the most used integrated development environment according to respondents to the Stack Overflow developer survey. Over 74% of respondents answered that it is one of their preferred IDEs. [4]

[1]Visual Studio Code is commonly abbreviated VSCode; it is not to be confused with the older independent project Visual Studio

While the Visual Studio Code contains some proprietary code, the majority of it is open source in the *Visual Studio Code – Open Source* ("Code – OSS") project. [5] The editor itself is mostly written in TypeScript using the Electron framework[2].

One of the defining features of VSCode is that it isn't tailored to be used with a specific language; instead language support is completely provided through extensions, some of which are built-in. [6] VSCode implements this using Language Server Protocol (LSP for short), which defines a protocol for editors to communicate with 'Language Servers', to provide language support for all editors supporting it at the same time, instead of requiring implementation for each pair individually. [7] This, however, limits the capabilities of the language servers to those defined by the Language Server Protocol. While the LSP continuously adds more features, the standardisation is slow and without a consistent release cycle. This evident from the LSP changelog. [7]

Another important feature of VSCode is the ability to extend its functionality using its Extension API. [6] This API provides high flexibility, allowing for the implementation of a wide variety of features in the editor, including completely custom views. Extension API is the approach our tool will use for interacting with the editor.

1.2 Existing solutions

The Language Server Protocol provides API to: [7]

- Navigate to Declaration/Definition/Type Definition/Implementation; these, however, doesn't help with understanding how a specific variable or object is used elsewhere.

- Find References, which is useful, but the result of this operation is only defined as a list of locations (a pair of a document path and a text range inside the document) – it doesn't offer any semantic information, which can lead to hundreds or thousands of results that a programmer would have to go through manually, making it badly suited for larger projects. [1]

- Show call hierarchy (incoming and outgoing calls), which only works with functions and doesn't provide any information on the implementation of the mentioned functions. In VSCode's implementation, the only shown information about incoming/outgoing calls is the name of the function and in case of a method the name of the containing class, which makes it hard to pinpoint the origin of said function without navigating to the code itself.

[2]`https://github.com/electron/electron`

It also doesn't work for class constructors, where calls from child classes are ignored.

- Show type hierarchy, which doesn't work for TypeScript classes.

There are attempts to extend these features, but there is little activity on those proposals or suggestions[3].

Visual Studio Code has TypeScript support through an in-built extension, but it doesn't add many features above the scope of the Language Server Protocol, with the only notable feature for our purposes being 'Find File References', which lists files that import the currently opened file. [8]

There do exist tools that provide more detailed information or more powerful search tools (one such example is the 'FEAT' Eclipse plugin), but those tools are often language-specific or for other IDEs. [1] I wasn't able to find any tool that focuses on or at least supports TypeScript and can be used in VSCode.

Besides tools aimed at developers, there do exist tools for static analysis, one such example being 'CodeQL', [9] a tool designed to discover vulnerabilities through static analysis of data flow. Such tools usually focus more on identifying specific problematic patterns instead of working interactively with the developer, making them a bad fit, as they favour accuracy over analysis speed, or don't work well with codebases as they change, needing a long time to re-generate the model before being able to answer more questions about the code after each change.

1.3 The goal of this thesis

The goal of this thesis is to create a VSCode extension that provides features similar to existing 'Find All References' and 'Show Call Hierarchy' tools, but with the added ability to categorise the results. It should also show more information about individual results, while still focusing on only being an overview and navigation tool for developers that is intended to be used interactively. It should focus on being responsive, avoiding heavy computations even on large projects, at the cost of accuracy, and preferring to show false positives where reference or usage cannot be easily confirmed.

The tool will focus on a single language: TypeScript, allowing us to design a tool based on facts and features specific to this language. JavaScript, being a subset of the TypeScript language and being supported by the TypeScript compiler, will be supported by this project too, but some features might be limited due to the lack of types, reducing the ability to cross-reference properties on objects throughout the analysed project.

[3] `https://github.com/microsoft/language-server-protocol/issues?q=label%3Areferences`

1.4 The structure of this thesis

In Chapter 2 we will explain what JavaScript is and the ecosystem that has been created around it; without explaining the details of the language.

The language itself will be described in Chapter 3. This chapter will describe how JavaScript evolved over time and it will explain the basic principles of the language and how is the language processed by engines. These descriptions will present sometimes lesser-known details of the language that are important to this project.

In Chapter 4 we will describe what TypeScript is and how it differs from JavaScript. We will also describe some of its core concepts and how TypeScript code can be transformed into JavaScript code to be used in applications. Finally, we will describe the architecture of the TypeScript compiler, which we will be using for parts of our analysis.

The following chapters will focus on design and implementation of our solution. Chapter 5 will analyse the complexities of analysing JavaScript and will propose an architecture of the solution. It will also analyse the feasibility of reusing parts of existing projects to reduce the effort needed to both create and maintain this tool. Chapter 6 will describe individual parts of our solution in greater detail, including approaches we experimented with before the current solution. Chapter 7 will focus on interaction with the programmer and on the presentation of answers to the programmer's queries.

In the last chapter (Chapter 8) we will show examples of how our solution can be used to answer programmer's questions compared to the existing tools. We will also measure how much code can our tool process on popular public code repositories.

In the Conclusion we will describe how we met our goals and possible future work to be done on this project.

Chapter 2

JavaScript Ecosystem

In this chapter we describe the current state of the JavaScript ecosystem, how JavaScript is defined, and how it is commonly used and integrated into larger projects. The focus of this chapter is to create an image of the wider context of how is JavaScript being used today. This chapter does not describe the language itself – that is done in the following chapter.

2.1 Brief introduction

JavaScript is an interpreted or just-in-time compiled programming language, depending on the engine used to run it. It is most known for its usage on the World Wide Web as a language, using which web developers can create scripts that run on the client. There are also non-browser environments that use it, the most notable example being Node.js. [10]

For ten years now, JavaScript has been selected as the most commonly used programming language by the respondents to the Stack Overflow developer survey, with TypeScript occupying the fifth spot in the 2022 survey. [4]

2.2 Specification

The language itself is standardized in the ECMAScript Language Specification. [11] JavaScript is a continuously evolving language, with new editions of the specification being released yearly. At the time of writing, the latest edition was released in June 2022 with a size of 846 pages.

The ECMAScript Language Specification defines only the core of the language: The syntax, the rules for execution of source code, the primitives of the language (number, string, boolean, object, etc.), and the behaviour of several built-in objects and functions. It, however, doesn't define any interface for interacting with the environment the code is running in. There is also the 'Official ECMAScript Conformance Test Suite', [12] which contains close to a hundred thousand test cases to test the conformance of language interpreters.

This means that for JavaScript to be usable in real-world scenarios, there is a need for additional specifications, to define the interface for interacting with the environment. The most notable example are the WHATWG[1] standards, which include specifications for the debugging console, HTML and browser interaction, DOM operations, and more. [13] Other notable examples are the ECMAScript Internationalization API Specification [14] and Node.js API documentation. [15]

Most of these specifications are living/evolving standards, making it hard to create a JavaScript analysis tool that can keep up with the changes.

2.3 Implementations

There are many JavaScript implementations/engines available[2], however overall, there are three "main" ones:

- V8, [16] developed by Google and used in Chromium[3], Node.js, Deno, and more...

- SpiderMonkey JavaScript/WebAssembly Engine, [17] developed by Mozilla and used in Mozilla Firefox.

- JavaScriptCore[4], [18] developed by Apple and used in WebKit browsers, such as Apple Safari.

Both feature support and conformance to the specifications can vary across the engines, even for the ECMAScript Language Specification. This can be seen by running the official conformance test suite on various implementations. One such public project is the 'Test262 Report' (last updated on 22/09/2022) which reports, that the above three engines pass on average 85% of the tests. Many of the failing tests are features that have not yet been implemented in the engines (such as 'Temporal' and 'ShadowRealm' built-ins), but in some cases, it shows that engines can diverge from the specification. [19]

[1]WHATWG stands for: 'Web Hypertext Application Technology Working Group'

[2]A non-comprehensive list is available at: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/JavaScript_technologies_overview#javascript_implementations`

[3]Which includes Chromium-based browsers, such as Google Chrome and Microsoft Edge

[4]JavaScriptCore has also been marketed under the names 'SquirrelFish', 'SquirrelFish Extreme', 'Nitro', and 'Nitro Extreme'

This problem is even more apparent when comparing conformance to additional specifications (such as WHATWG specifications), where support differs not only based on the engine and its version but also the environment the engine is used in. Many vendors also add proprietary extensions or APIs. An example of these are the various browser extension APIs.

2.4 Libraries

Libraries are an integral part of the JavaScript ecosystem. The only notable repository of public packages is the npm Registry. [20] Npm stands for 'Node Package Manager', which is a tool for managing and installing dependencies. As the name suggests, it is closely linked to Node.js, which bundles it in its installer and recommends it. Npm also has a specification for the 'package.json' file, [21] which became the de facto standard for JavaScript projects to define their dependencies and development or build-time scripts.

For example, the TypeScript compiler has (at the time of writing) 42 direct dependencies and 367 dependencies in total (accounting for transitive dependencies).

2.5 Transpilers and Packagers

While JavaScript is an interpreted language (meaning there doesn't need to be a compilation step between writing code and using it as an application), there are many cases where requirements or best approaches to code differ between the development environment and the code that is to be shipped in the final application. Examples of those are:

- During development, we want to comment the code to make understanding it easier; in the the shipped application, the comments are not visible to the end user and only increase the application size and parsing time.

- During development, it is beneficial to name classes, variables, functions, and other identifiers in a meaningful way, while in the shipped application, identifiers don't serve any other purpose than to uniquely identify objects.

- New language features are usually created with the intent of making development easier; however, for the shipped application we want to avoid using bleeding-edge features to increase compatibility with older browsers that users are likely to be using.

Transpilers[5] and packagers solve this issue by transpiling the source code into a different source code by, for example, removing comments and unnecessary whitespace, shortening identifier names, or rewriting modern syntax using older, widely supported syntax. An example of such a transpiler is Babel. [22] Packagers might also optimize code further by combining multiple source files and used libraries into a single output file or removing unused code[6]. [23, 24]

2.6 Flavours

JavaScript flavours are programming languages that are compiled into JavaScript. Flavours can vary from languages with syntax similar to JavaScript, such as TypeScript and Flow, which have a grammar that is almost a superset[7] of the JavaScript grammar, [2] to languages with a completely different grammar that are simply compiled into JavaScript, for example, ClojureScript based on the Clojure language, which is a dialect of Lisp. [25]

The most popular flavour is currently TypeScript, with over 75% of respondents of the annual developer survey of the JavaScript ecosystem 'State of JavaScript' in 2020 having answered they have used it. [26]

[5]Transpilers are also sometimes referred to as 'source-to-source compilers'

[6]This operation is often referred to as 'Tree shaking'. The name have been popularized by bundler Rollup. [23, 24]

[7]Superset in this context means, that any valid JavaScript program is also a syntactically valid program of said flavour

Chapter 3

ECMAScript

This chapter describes the JavaScript language. First, it introduces brief history to set expectations about how often the language changes over time. Following that, it describes several core aspects of the language. Finally, it describes a few aspects of how interpreters should work based on the specification. This information forms the foundation for our design in the following chapters.

3.1 Brief history

"ECMAScript is based on several originating technologies, the most well-known being JavaScript (Netscape) and JScript (Microsoft). The language was invented by Brendan Eich at Netscape and first appeared in that company's Navigator 2.0 browser. It has appeared in all subsequent browsers from Netscape and in all browsers from Microsoft starting with Internet Explorer 3.0." [11]

Arguably the most significant release was the sixth edition (dubbed ES6 or ES2015) after almost 15 years of effort, which added many major language constructs used in modern JavaScript codebases. These include, for example, classes, lexical block scoping, iterators and generators, and promises for asynchronous programming and destructing patterns. The base library was also expanded with additional abstractions like sets, maps, and more. [11, 22]

The sixth release was followed by yearly iterations of the specification, each adding new features. [11] Here are a few highlights of each of the following releases:

- 2016: Thousands of fixes and clarifications in the specification with only a few features. New features were, for example, the array method 'includes' and an exponentiation operator.

- 2017: Introduction of async functions (with 'await'), Shared Memory and

Atomics for multi-agent programs, and many helper functions on ‘Object’.

- 2018: Asynchronous iterators, spread and rest properties for destructing assignments, and fair few Regexp improvements.
- 2019: Many array and string helper functions, requiring the array sort function to be a stable sort, and allowing newlines in string literals.
- 2020: Standardised ‘globalThis’ object, asynchronous dynamic imports, nullish coalescing and optional chaining operators.
- 2021: ‘replaceAll’ method for Strings, logical assignment operators (‘??=’, ‘&&=’, ‘||=’), weak references, and ‘FinalizationRegistry’.
- 2022: Top-level ‘await’ expressions, public and private fields for classes, static blocks inside classes, and the ‘at’ method for strings and arrays.
- 2023: Many useful array methods, and syntax to allow shebang comments (‘#!’) at the beginning of files.

3.2 Brief language overview

First, we will focus on a few aspects of the ECMAScript language that are important for this project from the view of the programmer.

Everything in this section is based on the ECMAScript® 2023 Language Specification *[11].*

ECMAScript is an object-oriented programming language with garbage collection. All code is executed inside a single thread where all code acts as coroutines – only operations that can happen in other threads are outside interactions (such as asynchronous file read/writes or network requests) and operations not observable by the code (such as garbage collection). True parallelism can only be achieved by starting multiple interpreter instances that communicate using events, shared memory, and/or atomics. The main event loop for long-running programs happens in the interpreter, making it primarily an event-driven programming language.

3.2.1 Strict mode

In the fifth edition of ECMAScript, a strict variant of the language was added. The strict mode adds several restrictions that forbid actions that can be seen as likely errors, or at least error-prone; for example, assigning to an identifier that doesn’t exist creates a new variable in non-strict mode, while it is an error in strict mode.

Strict mode is enabled by specifying a ‘`"use strict";`’ directive in the prologue[1] of a file or function body. Some syntaxes (e.g. classes) also enable it implicitly for their body. If strict mode is enabled, then all nested code also uses strict mode. It is valid to load multiple files where some are in strict mode and some are not. As those files can freely interact, strict mode only affects code originating from strict mode files/functions and only imposes restrictions that have local effects.

TypeScript allows enabling this mode by default using the configuration option ‘alwaysStrict’ (which is implied by option ‘strict’).

3.2.2 Value types

ECMAScript values have specific types, which can be categorised into two categories: primitive values and objects.

Primitive values are represented directly by the implementation and can be considered immutable value types. There are 7 types of primitive values:

- Undefined, which can have exactly one value: ‘undefined’.
- Null, which can have exactly one value: ‘null’.
- Boolean, which is either ‘true’ or ‘false’.
- String, which is a sequence of 16-bit unsigned integers. Strings are interpreted in UTF-16, but individual elements of the sequence can be arbitrary 16-bit values, meaning strings can contain ill-formed sequences.
- Number, which is a 64-bit double-precision float, as defined by ‘IEEE 754-2019’. ECMAScript makes use of positive and negative Infinity and differentiates between positive and negative zero. All Not-A-Number values are treated identically by the specification.
- BigInt, which is an arbitrary integer value, not limited to a particular bit-width. For binary operations BigInt acts as two’s complement binary string, with negative numbers treated as having bits set infinitely to the left.
- Symbol, which represents a unique and immutable value that can be used as a key of an Object property. There are also several "well-known" symbols that the specification uses.

[1]Prologue is the longest sequence of statements that only contain string literals at the start of a file or function

The second category of values are objects. Objects are always passed by reference and are mutable. Any object can be seen as a collection of properties, where each property is uniquely identified by a 'property key' (either a String[2] or a Symbol). Each property can be either a 'data property' – an ECMAScript value, or an 'accessor property'. Accessor properties are a set of 'get' and 'set' functions that are invoked when the property is read or written. All objects also have exactly one 'prototype', which is either an Object or 'null'. Prototypes will be described more in detail later.

Objects can be further categorised into ordinary objects and exotic objects. Ordinary objects are usually created by the object literal expression (''). Any object that isn't ordinary is called exotic. Exotic objects can define special behaviour for when code interacts with them, for example, when getting or setting a property or when calling the object. Examples of exotic objects defined by the ECMAScript specification include an Array object, a Function object, or a Proxy object which allows JavaScript code to define the behaviour of the object.

All primitive values except 'undefined' and 'null' can also be "wrapped" in a respective exotic object, which allows calling methods on primitives. Similar to the primitives, these objects are also immutable. This wrapping can either be done explicitly or, in some cases, happens implicitly, which allows for syntax such as '(42).toString()'. Attempting to access any property of 'undefined' or 'null' results in a runtime error.

3.2.3 Variables

Variables can be declared using three keywords: 'var', 'let', and 'const'. 'var' declarations are scoped to the function, meaning that even if they are declared in any nested block in the function, they are accessible anywhere inside the function, including before the declaration itself.

ES6 added 'let' and 'const' declarations, which are scoped "lexically" – this usually means to the closest containing block. The difference between 'let' and 'const' is, that 'const' declarations must have an initialiser and cannot be reassigned. 'let' and 'const' declarations also forbid declaring multiple variables with the same name in a single lexical context (including 'var' variables and function/class declarations), resulting in a syntax error.

Code can also access any variable from the outer context. For example, a variable declared in the root of the file can be accessed by any function in that file.

[2]Integers in the range from $+0$ to $(2^{53} - 1)$ can also be used as property keys. Such numbers are interpreted the same way as their decimal string equivalents (so '`obj[42]`' is the same as '`obj["42"]`')

Listing 1 A variable shadowing example.

```
function variableShadowingExample() {
    let myVariable = 1;
    myVariable; // -> 1

    if (true) {
        // The following line would result in a runtime error
        // myVariable;

        let myVariable = 2;
        myVariable; // -> 2

        myVariable = 3;
        myVariable; // -> 3
    }

    myVariable; // -> 1
}
```

It is legal to create a variable in a nested block or function when the outer context already has a variable of the same name. In such cases, reads or writes happen to the variable in the "closest" lexical context. Such an occurrence is usually called variable shadowing[3]. [27] You can see an example in Listing 1.

3.2.4 Functions

ECMAScript has first-class functions. To be exact, all functions are exotic objects, and as such, they can be freely assigned to any variable, property or passed as a call argument.

Function declarations behave similarly to 'var' variable declarations, including conflicting with lexical variable declarations and the possibility to reassign the variable (even with a non-function value). One important difference between function declarations and variable declarations is that function declarations are 'hoisted' – the function can be used even before the actual function declaration.

ECMAScript has a concept of generator functions and async functions (or combined async generator functions), which are functions that allow suspending the execution at certain points (using 'yield' and 'await' keywords), so other code can run, and resuming it at a later point in time.

In ECMAScript arguments are passed as a list. During standard calls, arguments are added to the list in order as they appear in the call expression. Standard

[3]The term 'shadowing' is not used in the ECMAScript specification

Listing 2 An IIFE pattern example.

```
// This creates a function using the function expression
   syntax
(function() {
    // Any code here has its own lexical environment
})(); // And immediately calls (invokes) it
```

arguments can be freely interleaved with ‘spread arguments’, which are iterated to produce an arbitrary count of arguments that are added to the list. This makes it difficult to track which position will the argument be in if it occurs after a spread argument. In a function declaration there exists a similar structure called the ‘rest parameter’. This can only occur in the last position, allowing us to easily track preceding parameters.

Functions can also be created using a ‘function expression’ instead of a declaration. This is a syntax that doesn’t implicitly create a variable but instead allows using the function as a value in an arbitrary expression. One common pattern that uses this is ‘IIFE’ – Immediately Invoked Function Expression. It is a pattern where a function is created and immediately called, with the benefit that the code inside the function has its own variable and lexical scope that is not visible from the outside. This pattern was commonly used before the standardisation of CommonJS and ES Modules [24] and is still used by many packagers to simulate module scopes when bundling multiple source files into one. Example of this pattern can be seen in Listing 2.

3.2.5 The ‘this’ value

During a function call, the function is passed a ‘this’ value that is used whenever the function implementation accesses it.

The value of ‘this’ usually depends on the caller – when a function is called normally, the ‘this’ is set to the global object (‘globalThis’). When a function is called using the property access syntax (i.e. as a method), the value of ‘this’ will be the value of the object. A function can also be called using the ‘Function.call’ and ‘Function.apply’ built-in methods, which allow specifying a custom ‘this’ value.

Two exceptions to this are ‘bound function’ exotic objects (which are created by calling ‘Function.bind’ on any function) and arrow functions. These ignore the passed this value and use the value they have stored during their creation instead – for bound functions, this is the value that was passed as an argument to the bind call; for arrow functions, it the is value of ‘this’ at the point they were created.

3.2.6 Prototypes

"Even though ECMAScript includes syntax for class definitions, ECMAScript objects are not fundamentally class-based such as those in C++, Smalltalk, or Java." [11] Instead, all ordinary objects have a 'prototype' – either an Object or 'null'. As a prototype can also be an object, and all objects have a prototype, this creates a 'prototype chain'. ECMAScript guarantees that for ordinary objects, this chain is acyclic – ending with 'null' instead of an object at some point.

If we try to get a property that is missing on the object, we try to recursively get it from the prototype instead, making all prototype properties shared between all objects that share that prototype, as long as those objects don't have their own property of that name. When we try to set a property that doesn't exist on the object, the prototype chain is checked for the presence of the property, similar to get. If one is found and it is an accessor property, then the setter of the accessor is invoked; otherwise a new property is created on the *original* object (not on the prototypes).

This creates a system that allows inheritance, however, unlike most object-oriented languages, the class hierarchy in ECMAScript can be modified at runtime, which is a feature likely to create confusion if used.

ECMAScript also has multiple flags that limit modifying objects, for example, '[[Extensible]]' on objects, and '[[Configurable]]' and '[[Writable]]' on properties. These can be used to prevent adding new properties, replacing data properties with accessor properties and vice versa, and changing the value of data properties, respectively. When these flags are set to 'false', they cannot be subsequently set by any means. Attempts to perform actions that are forbidden by these flags will cause a runtime error.

3.2.7 Classes

While ECMAScript has classes, until ES2022 they were mostly syntactic sugar on top of the prototype model, with only a few differences in behaviour from functions with the 'prototype' property. One notable difference is that while functions are hoisted and behave similarly to 'var' declarations, classes cannot be used until after their declaration, behaving similarly to 'let' declarations (meaning a class can still have its value reassigned). Another difference is that classes are always parsed in strict mode.

The feature that makes them stand out since ES2022 is the possibility for classes to have 'private fields', which are properties that are not accessible by any code that was not written as part of the class declaration.

3.2.8 Environment

Before any script is executed, the ‘global object’ is created. All properties of this object are available directly as if accessing a variable. The object itself is also available as a value of ‘globalThis’. The name comes from the fact that if a function is not called with a specified ‘this’ value (not called as a method, and not called using ‘Function.call’/‘Function.apply’), the global object is used as the ‘this’ value.

While the specification defines some properties and their values that must be on the object, the host (browser or any other runtime) is free to define any additional properties and the object’s prototype. For browsers, it contains all the browser APIs, and it is also common to set properties ‘window’ and ‘self’ to this object.

One more detail is that ‘var’ and function declarations that are done in the global context (in the root of a non-module file – not in any function) are added as properties of the global object, replacing existing ones instead of shadowing them.

3.2.9 Modules

ECMAScript programs can be divided into modules, which don’t execute in the global environment, but each module has its own, private ‘module environment’. Each module can then explicitly export variables that should be visible to the outside. The module can update the exported values, and the changes propagate to modules that imported them; these are read-only from outside of the module. Modules can also import other modules either as a ‘namespace’, which creates an exotic object whose properties correspond to the exports of the module, or they can import individual exports from other modules.

3.3 Interpreter

While the ECMAScript specification leaves many implementation details to the interprets, several key concepts must be followed by all implementations. We can use these while performing the analysis.

3.3.1 Static semantics

The specification generally divides operations over JavaScript code into two categories: Static semantics and Runtime semantics. Static actions are performed on a parsed code tree and are not dependent on any runtime state, making them usable for our analysis.

Listing 3 Comparison of direct and indirect eval.

```javascript
// An example of direct eval:
eval("1 + 1;");

// An example of indirect eval:
const myEval = eval;
myEval("1 + 1;");

// An example of indirect eval:
// The comma operator performs `GetValue` on the operand,
   which turns the reference into a value
(0,eval)("1 + 1;");
```

One important aspect is that static semantics dictate several 'early errors' that prevent any code in a problematic file from executing. This means we can safely ignore any files that are not syntactically valid, as well as several possible errors, without sacrificing any accuracy, as those files won't be able to run at all.

The second static semantic we will make heavy use of is name binding. While static semantics is not enough to resolve all identifiers statically because the resolution defaults to the properties of the global object, it is enough to reliably detect, for each identifier, whether it resolves to a variable local to the file or whether it falls back to the global object. This can be also be used to detect all uses of a local variable (where local means it is not a top-level variable in a non-module file or it is a non-exported variable in a module file). There are however two exceptions to this.

One exception is a 'with' statement, which makes properties of object available in the current scope, similar to variables. This feature is however considered legacy and its use is discouraged due to the unpredictable nature of any binding inside it. The other exception to this is the direct use of the 'eval' function, which can dynamically access the LexicalEnvironment and VariableEnvironment of the execution context it is executed by.

ECMAScript differentiates whether the 'eval' function was called directly or indirectly. If the call is literally 'eval(...)', and the 'eval' is a reference resolved to an in-built eval function, it is a direct eval call, which runs in the context of the caller. If the function isn't a reference but a value, or is a reference not named 'eval', then even if the called function is the built-in eval function, the evaluation happens in the global scope – unable to access local variables not normally visible from other files. Examples of these can be seen in Listing 3.

3.3.2 Execution

When executing code, the compiler has a current 'execution context'. This context contains data necessary during execution, for example, which function is currently being executed, which script or module this code is part of, and the current lexical, variable, and private environments (environment records), which are used for resolving bindings. While the current execution behaves mostly like a stack, it is not true in every case – sometimes execution can get suspended at a particular point to be resumed later. Examples of this are the 'await' expression in async functions and the 'yield' expression in generator functions.

Environment records behave similarly to objects, including prototype chains – each environment record has a 'base' record (except for the global environment record, which has no base). When looking up a variable, the lookup is performed on the current environment record and only if there is no matching variable, the resolution continues on the base record, recursively. If there is no resolution and no base record, the resolution fails.

Evaluation operation

The most important operation during runtime is evaluation. While interpreters are free to implement this differently than how the specification describes it, especially to optimize things (e.g. during just-in-time compilation), the observable effects must match the operation as described in the specification.

Evaluation is an operation that happens over a parsed syntax, takes no arguments and returns a 'Completion Record'. Almost all syntaxes define their own implementation of this operation, which defines actions the syntax resolves to at runtime. Some syntactic tokens don't have evaluation defined, but those can only be children of syntax that never performs generic evaluation on them, instead treating them specially.

Completion Records

Completion Records are what carries information about the result of the evaluation of any node. A Completion record contains the type of completion, optionally a value, and rarely also a 'target'[4]. The most common completion type is 'normal', which refers to completion that doesn't result in any control flow changes. The value of this completion is the value the expression evaluated to – it could be 'empty', ECMAScript value (primitive or object), or a reference record pointing to a variable or property.

[4]Completion target is used by 'continue' and 'break' statements with label.

Any completion that isn't 'normal' is called an 'abrupt completion'. Possible abrupt completions are: 'break', 'continue', 'return' and 'throw'. Most evaluation variants perform a check called 'ReturnIfAbrupt' on the completion of any child statement – if the child statement resulted in an abrupt completion, then evaluation of this statement is interrupted, and the abrupt completion is returned as-is; otherwise, the value of the normal completion is unpacked and used instead. This operation is used so often, that it has a shorthand in the specification pseudocode – '? operation()' is equivalent to 'ReturnIfAbrupt(operation())'. Some statements, however, react to abrupt completions – for example, loops handle 'break' and 'continue' completions and 'try' statements handle 'throw' completions specially.

Chapter 4

TypeScript

This chapter introduces TypeScript as a flavour of JavaScript. It describes how TypeScript is related to JavaScript both in the ecosystem, as well as more technically – what does it provide that JavaScript doesn't, and how it can be used in the same way as JavaScript. This is followed by a brief overview of TypeScript tooling, primarily focused on the TypeScript compiler from Microsoft.

4.1 Introduction

As said by the title page of TypeScript: "TypeScript is JavaScript with syntax for types". [2]

TypeScript is (almost) a superset of JavaScript, meaning (most) JavaScript code is syntactically valid TypeScript. TypeScript aims to address the problem of JavaScript being dynamically typed by providing ways for programmers to define and assign types and providing tooling to check the flow of types for errors. It also integrates well with IDEs to provide features like code completion.

It is a project developed by Microsoft that has a stable core team working on it. The TypeScript language is primarily defined by the TypeScript Compiler, [28] TypeScript compiler notes, [29] and the TypeScript website. [30, 2]

TypeScript currently lacks any official, up-to-date specification. A team member cited a lack of resources to update the specification. [31] The old specification has instead been removed, and TypeScript doesn't have a single source of truth, instead being described by the TypeScript handbook, [32] release notes, and implementation of the official compiler. [31]

The last specification was released in August 2013 and described TypeScript's syntax as of version 0.9.1. The version this project aims to analyse is 5.0.4, making it unusable for our purposes. The last officially released specification can be found archived in the Wayback Machine[1].

4.2 TypeScript in the context of the JavaScript ecosystem

TypeScript is widely adopted in the JavaScript ecosystem. That is evident from, for example, the fact that many packages provide TypeScript type definition files describing their API, even if the library itself is not written in TypeScript. There is also a project called 'Definitely Typed'[2], which at the time of writing contains community-contributed TypeScript definitions for 8639 packages, each available under the name '@types/<name of package>'.

These two cases are also recognised by the npm package registry, which shows a badge next to the package name for packages that either provide their own typings or have typings available as a Definitely Typed package.

4.3 TypeScript language

TypeScript as a language doesn't have an official grammar but is defined by what the TypeScript compiler can parse and process. The main difference versus JavaScript is that TypeScript, as the name suggests, allows specifying types for variables, parameters, and more.

TypeScript can be considered a best-effort checker, which is indicated, by one of TypeScript's goals being *not to* "apply a sound or "provably correct" type system. Instead, strike a balance between correctness and productivity." [33] TypeScript only provides static type-checking at compile time – the emitted JavaScript code contains no extra runtime checks and imposes no run-time overhead on the emitted programs. [33]

An important exception to this rule is that TypeScript does add several new syntaxes that either don't exist in ECMAScript or are at a level of not yet fully standardised proposals. Examples of these are namespaces[3], enums, and decorators.

There are types provided for all ECMAScript primitives (string, number, boolean, null, undefined, bigint, and symbol). Types of objects are defined using shapes – where the shape of an object describes what properties and/or methods

[1]`https://web.archive.org/web/20131117065339/http://www.typescriptlang.org/Content/TypeScript%20Language%20Specification.pdf`

[2]`https://github.com/DefinitelyTyped/DefinitelyTyped`

[3]These used to be called 'modules' or 'internal modules' in older TypeScript versions, as they predate ECMAScript modules

it has and their types. TypeScript also allows combining types in various ways, the most common being 'union type', which is formed from two or more other types and represents values that may be any of those types. [32]

While ECMAScript treats arrays as exotic objects, in TypeScript, arrays have a special type and are treatmented specially by the compiler. There is also a 'tuple' type, which defines array elements based on their index. An important caveat with array types is, that they are treated as covariant, while in reality they are not. This is a common and problematic pattern in programming languages, including C# and Java. [34]

There are also four special types:

- 'void' is a type similar to 'undefined' that is used to signify the absence of a value – for example, when a function returns nothing.
- 'any', which can be used in any way without causing type-checking errors.
- 'unknown', which defines a type that is checked for errors and can be an arbitrary type. Generally, there is a need to use type-guards before using values of this type.
- 'never', which signifies that code that would use this type never executes. Two common use cases for this are never-returning functions (that contain an infinite loop or always throw) and exhaustive switches/branches that handle all possible values.

TypeScript's type system is structural and shape-based. This means any two types that have the same shape (for example, objects that define the same properties) are considered the same type. This makes it problematic to define what is a usage of a type, as only referring to locations that mention the type explicitly may miss some uses of said type, while considering all types that match the shape of a specific type would lead to many false positives, where there are unrelated types with the same shape.

4.4 TypeScript compilers

There are multiple existing compilers capable of transpiling TypeScript code into JavaScript – e.g. Babel [22] or Speedy Web Compiler. [35] These compilers, however, either don't provide type-checking at all or only a limited subset of the checks that the TypeScript compiler does. They also generally lag behind the TypeScript compiler in implementing new features or don't have an API to use data from the compilation.

The TypeScript compiler from Microsoft [28] provides a good API for parsing and processing the language and is de facto the project that defines the TypeScript language. Using it in our solution would provide the benefit of easier and more up-to-date support of new features that might be added to the language, which usually happens every three months. [33]

Visual Studio Code also uses this compiler internally for providing TypeScript and JavaScript language features inside the editor, such as IntelliSense, code navigation, formatting, and refactoring tooling. [8]

4.5 TypeScript in VSCode

In VSCode, all extensions run in an extension host process and are isolated from each other. They can also define an API to be used by other extensions, but only this API is accessible from outside.

TypeScript support in VSCode is implemented using an in-built extension. [8] This extension doesn't have a TypeScript compiler bundled; instead, it finds a compiler in the opened project and spawns it as a child process 'tsserver'. It then communicates with the compiler using the TypeScript Server Protocol. This allows using the same version of TypeScript as the opened project does, but also limits available features to those requests defined in the TypeScript server API. [28, 33]

As the TypeScript language support extension doesn't provide any API to communicate with the TypeScript server, it cannot be reused in any way for our solution.

4.6 TypeScript compiler API

Besides the 'tsserver' API, TypeScript has two more official public APIs.

One of them is the Language Service API[4], which provides an API aimed towards language servers, such as detecting errors/diagnostics in processed files, finding rename locations, navigating to definitions or implementations, and performing quick code actions. [28, 33]

The second exposed API is the Compiler API. This API allows starting the compiler with any normally available settings from the commandline (and some extra ones), running specific compiler stages, and using most of the internal compiler data available between the stages. [33] Some cross-stage data, as well

[4]The TypeScript Language Service API does *not* implement the Language Server Protocol, but it does contain all that is necessary to implement it

as data internal to each stage, is not accessible through this API. An example of such non-accessible data is TypeScript's flow graph. [28]

4.7 Inner workings of the TypeScript compiler

At the core of the TypeScript compiler is a 'Program'. It can be seen as a coordinator for individual stages and for handling interaction with the outside – for example, all APIs and the file system watching during watch compilation.

A typical TypeScript compilation has the following stages: [29]

- Read config: The 'tsconfig.json' file (or equivalent) is read and parsed, the Program is created, and entry files specified in the config are found.
- Pre-process Files: The included files are recursively pre-processed, following imports to discover all files to be used during compilation. This does not parse the whole files, only the imports.
- Tokenize and Parse: All files relevant to the compilation are first scanned for syntax tokens, and those tokens are then converted into a syntax tree.
- Binder: Resolves all identifiers in syntax trees to symbols. It also creates base data that will be later used for flow analysis, but this data is not exposed through the API.
- Type Checker: Uses syntax tree and binder data to look for issues in the code, mostly by resolving types possible and required at all expressions and checking their compatibility. This is arguably the most complex stage of compilation and does most of the "smart" work, such as deriving types and tracking how they propagate through expressions. This data is, however, not observable from the outside, as it is optimised for speed, not external usability.
- Transform: Walks through the syntax tree and modifies it based on the compilation options. This mainly means removing or transforming TypeScript-specific syntaxes to produce valid JavaScript, and transpiling new JavaScript features using older ones, if requested by the config.
- Emit: Uses the resulting syntax tree from the transform stage to produce the final output files – either JavaScript ('.js') or type declaration ('.d.ts') files.

Listing 4 An example of a transient symbol occurrence.

```
const randomBoolean = Math.random() < 0.5;
(randomBoolean ? 42 : "fish").toString();
```

Each stage is designed to do a single pass over each source file (some can also skip parts of a file). Each operation of the stage is instead performed recursively on the source files (or the syntax trees from previous stages). [29, 28]

Symbols are the base building block of TypeScript's semantic system – they are used to uniquely identify any named declaration based on its role. For example, a unique symbol is assigned to each variable, class, type, and property. Symbols are mostly created in the binding phase, but some symbols can also be created in the type-check phase – those symbols usually represent a union of possibilities. Symbols created during type-checking are called 'Transient Symbols'. [29, 28]

In Listing 4 we can see example of transient symbol occurrence. This happens because 'randomBoolean' can be either true or false in this case, the ternary expression resolves to type 'number | string'[5]. Each of those types is then checked for presence of the 'toString' property, which is defined on both types, so it isn't an error. Each 'toString' declaration, however, has a unique symbol, so here a Transient Symbol is created that represents the possibility of either case. This symbol is then used when validating that the expression is callable and when validating the call signature.

[5]Modern TypeScript would actually resolve it to type '42 | "fish"' thanks to the support for number and string literals

Chapter 5

Design

In previous chapters, we have analysed the current state of the JavaScript ecosystem, some important aspects of the JavaScript language, and existing tools for working with the language. We also talked about what TypeScript is, how it fits into the wider ecosystem, and important details of the TypeScript compiler implementation.

In this chapter, we define the scope our goals more closely, in order to match a scope of a bachelor thesis. We also start designing our solution and choose technologies to use so that we can meet these goals.

5.1 Scope

5.1.1 Syntax support

Due to the complexity of JavaScript and the lack of specification for TypeScript, as well as their evolving nature, it is not feasible to support all language constructs they provide. Therefore we need a way to select a subset of the languages we will be targeting, which can be solved in two ways. Either we can limit the solution to older versions with fewer features, or we can support features we deem 'important', providing fallback behaviour that can handle arbitrary unsupported features with at least some degree of accuracy and usefulness.

Because the nature of the project is to aid developers working with modern code that is still being developed, the approach to limit the language version would also heavily limit the usefulness of the tool. Therefore we have chosen the second approach of supporting a subset of the language based on how important specific grammar is for our analysis and how often it is used.

The analyser should, however, still provide limited results even if it encounters syntax it doesn't recognise. The analyser should provide some results even if parts of the codebase fail to be analysed altogether for any reason and warn about such problems.

5.1.2 Environment support

Another important decision is which execution environments we want to support – such as web browsers in general, a specific web browser, Node.js, or others. Here the decision we made is to create a generic tool that can be used independently of the execution environment; therefore, the analysis should only use facts defined by the ECMAScript specification, or by the TypeScript behaviour.

5.1.3 Accuracy vs Performance

As our tool is meant to provide an interactive interface during development even as the code changes, our solution needs to focus on performance and interactivity at the cost of accuracy. Where there is doubt about including a specific reference in the result, the reference should be included with the available information, so the developer can inspect it manually.

The analyser should also cache computationally heavy calculations to allow faster consecutive queries on the same project and detect changes made to the code by the developer and invalidate cached data, redoing the necessary analysis on the next query.

5.1.4 Alias resolution

The nature of JavaScript makes it difficult to reliably resolve many operations. One such example is calls that can happen to methods or global functions – these can be replaced during runtime. As we cannot know the full state of an application at runtime we need to limit our analysis to the cases where we are certain of the outcome.

There are also many code patterns where hiding the point in code where a variable is aliased might omit critical information, leading to confusion. An example of this can be seen in Listing 5. There the ‘savedState’ variable definitely has the value of the ‘globalState’ variable. Treating the variables as a single one would, however, omit the fact, that ‘savedState’ saves the value while ‘globalState’ changes.

This means that we will focus more on analysing the direct usage of variables, leaving it up to the programmer to understand how they are related. In some cases we can, however, reliably detect that variable or a function has only a single source of value. In that case, we will provide a way to continue inspecting the chain following such action. The analyser will also provide support for analysing

Listing 5 An example of a problematic pattern for alias resolution.

```
let globalState = ...;

function DoActionWithDifferentState(differentState) {
    const savedState = globalState;
    globalState = differentState;

    // Do a complex action potentially spanning hundreds of
       lines

    globalState = savedState;
}
```

arguments passed to functions in a similar way, based on the position of the argument. This may in some cases allow following a value through many layers of functions.

5.2 Problem analysis

Both queries we are implementing (finding references and showing information about a function) solve the same problem but present it from different directions. In both cases, we need to search the codebase for references to variables and detect what action is being performed using them. The difference between the two queries comes from how these actions are filtered. In the case of finding references, we want to find all actions that target a specific symbol, while in the case of showing information about a function, we want to find those actions that happen in a specific area of code – the implementation of the function in question.

For both queries, we need to scan the program for actions that can be relevant to the query. This need can be split into the following parts:

- Finding the scope for the analysis – which source files might be relevant to the query.
- Parsing the source files into a form we can perform analysis on – for example, a syntax tree.
- Resolving identifiers semantically – we need to resolve which variables individual identifiers refer to in order to be able to detect references.
- Detecting operations that are performed on those symbols – such as reads, writes, and calls.

We will analyse each of these parts in more detail.

5.3 Finding the scope for the analysis

In order to find references, we need to detect what code can access this symbol. That is not a problem in the case of the symbol being a variable local to a function or a file, but due to the nature of the JavaScript environment allowing import of arbitrary files, even to the global environment in some cases that were discussed in sections 3.2.8 and 3.2.9, we have no reliable way to detect where the file might be included from. This is complicated even more by features such as adding scripts as HTML tags, the ability to import files dynamically in JavaScript, and the fact, that the current project might be intended as a library to be used by other projects. For these reasons, we need to use information from outside the code itself to detect the scope.

One source of information we can use is the editor itself, but as Visual Studio Code is primarily a text editor that works with folders instead of projects (unlike, for example, Visual Studio), we only have limited information from the editor itself – the currently opened file that the user invokes the query from and a folder that is open as the root folder in the editor. Instead of a single folder there might also be multiple open folders, in case of a multi-root workspace.

5.3.1 Specifying the files manually

A simple solution is to let the developer select the scope of the analysis, but while this is a simple and working solution, it makes the tool much harder and annoying to use, as it requires specifying all files or at least folders to scan. This, to a large degree, reduces the benefits the tool brings by requiring much more effort to use it.

Therefore we wanted a way to detect the relevant files automatically.

5.3.2 Scanning for files

One of the solutions we tried was detecting all JavaScript and TypeScript files recursively from the root of the open workspace. However, even detecting what is a JavaScript or a TypeScript file effectively proved challenging, as they can have a variety of extensions – the base extensions are '.js' and '.ts'. There is also the optional suffix 'x' (so '.jsx' and '.tsx') for files that also contain React syntax. The filenames might also be prefixed with either 'c' or 'm' to explicitly annotate that it is a CommonJS module or an ES6 module, respectively.

After solving this issue, the second issue came from the fact that such a scan resulted in a large number of files to analyse. This was caused by multiple factors. One such reason is dependencies, which are usually installed in the 'node_modules' folder. We tried solving this by ignoring this specific folder during crawling. This has, however, met similar problems as extensions – some package managers (namely modern Yarn) might install or store dependencies in other folders, such as '.pnp.*' and '.yarn', making it hard to predict the project structure users of our tool might have.

Another considered option was ignoring files ignored by git. This solution, however, had both false positives and false negatives. For example, some tools might generate source files that are later used, but the generated files are commonly not checked into source control. False negatives, on other hand, were caused by the project using zero-install dependencies[1] or not using Git versioning at all.

This approach wasn't perfect even in projects that were using only common tools such as 'npm' without any code-generating frameworks. The reason for this was that multiple projects might be part of a single repository. Such repositories are referred to as 'monorepos'. For those the analysis again attempted to scan too many files, resulting in a loss of performance.

5.3.3 Using compilation settings

One stage where all relevant files are definitely used is during compilation. Normal use of the TypeScript compiler is done by invoking the 'tsc' command supplied by the TypeScript package. The compiler is passed either a list of files and options or a link to a 'tsconfig.json'[2] file, which contains compilation settings, most importantly for us, the list of files to include as compilation entrypoints.

This approach again proved challenging due to the non-uniformity of the ecosystem, where many projects use TypeScript but don't use the official compiler, instead opting to use alternative transpilers such as Babel [22] or SWC. [35] Even for projects that use the TypeScript compiler, this is not feasible because there is no standardised way of invoking it. While the most common way is to run it from a script defined inside the 'package.json' file, the name of the script is not standardised, and it can also perform other things besides only running the compiler, making it hard to get compiler arguments. The compiler might also be invoked from other tools that orchestrate the build process of larger codebases. For example, this is the case in the TypeScript compiler project, which uses 'hereby'[3].

[1]Sometimes also called 'vendored' dependencies; it is a strategy where dependencies are included in the codebase of an application for greater reliability (for example, in cases where a dependency might get unpublished by the author) and control over the exact version of the dependency.

[2]The file can have an arbitrary name; it only needs to be a valid JSON file matching the expected schema

[3]`https://github.com/jakebailey/hereby`

Other common tools that share this pattern are Rollup.js[4] and Webpack[5].

5.3.4 Solutions of existing tools

Following these failures, we decided to investigate how existing solutions approach this problem, which required studying the code of the TypeScript language features extension in VSCode. [8] As a second solution to analyse, we chose typescript-eslint – a tool that allows using the popular linter ESLint with TypeScript. [36]

The TypeScript language extension offloads this task to the TypeScript compiler using the Language Service API. The compiler implements this by looking for 'tsconfig.json' and 'jsconfig.json' files by going up the directory tree from the currently open file.

The typescript-eslint tool solves this by requiring the user to specify a list of TS config files to try in the ESLint configuration. It then tries these files in order and checks whether the config file includes the file it is expected to lint, using the first one that matches. If none do, then an error is produced.

5.3.5 Solution

As the final solution, we chose a combination of these approaches: First, we try to resolve the config file using TypeScript, and if that fails, then we try the ESLint approach. If neither results in a valid config, we fail with an error message. The reasoning behind preferring the TypeScript solution is, that many projects have a more generic TS config for ESLint that doesn't reflect how the code is used when not being linted.

5.4 Parsing the source file

In order to do any analysis on a file, we need to first parse it into a syntax tree (either concrete or abstract).

5.4.1 Custom solution viability

We immediately discarded the possibility of creating a custom parser. Earlier in this chapter, we already established that it is out of the scope of this project to support all possible syntaxes, and parsing a file without supporting the full syntax would be error-prone at best. Having a custom parser also conflicts with

[4]`https://rollupjs.org/`

[5]`https://webpack.js.org/`

the goal of the project being easily maintainable, as any changes to TypeScript syntax would require the work to be replicated in our parser.

5.4.2 Existing parsers

Without the possibility of a custom parser, we need to look at existing TypeScript parsers and select one to use for our project. Because of our syntax support goals, we have certain requirements for the parser to be a good fit for our project:

- It must be possible to use the parser from a VSCode extension.
- The parser must support reasonably recent TypeScript and JavaScript syntax.
- The parser must be maintained (had a release within the last year).
- There must be a way to create a "default" analysis for any syntax node we don't support (e.g. the syntax tree must be walkable with the visitor pattern).

With those facts in mind, we considered the following popular projects that work with TypeScript and inspected them closer:

- The TypeScript compiler from Microsoft: [2, 28] The TypeScript compiler is already being used in similar use-cases as we need, it obviously supports its own syntax, and it is actively maintained by a stable team. The compiler provides a stable, well-defined API, making it easy to keep it up-to-date without breaking our application, and each parsed node has a 'forEachChild' method that walks through its semantic children, skipping syntactic tokens. This makes it a prime candidate for use in our project.
- SWC: [35] Speedy Web Compiler is written in Rust and requires being compiled into a binary. This makes it a bad fit for intergrating it with our extension, as we would need to ship a binary for all platforms we want to support. While it does provide a clear API for parsing a file into AST, there are no helper methods for navigating the tree without having custom code for each syntax type.
- typescript-eslint: [36] The implementation of typescript-eslint internally uses the TypeScript compiler and converts the TypeScript syntax tree into an ESTree-compatible AST that can be used by ESLint for linting, taking the position of an adapter. Using it would, therefore, only add a level of indirection over using the TypeScript compiler directly.

- Babel: [22] Babel is implemented in TypeScript, so it would be usable within an extension. It is also actively maintained and supports parsing modern TypeScript. Its primary use-case is transpiling source code by replacing new syntax with equivalent older, more widely supported syntax. While it does support TypeScript syntax, it doesn't support any type checking and is focused more on the use-case of transpiling, making it a worse fit for our purposes than TypeScript.

5.4.3 Solution

For parsing source files into a tree that we can analyse further, the TypeScript compiler from Microsoft is the best choice. This also has more benefits outside of only parsing – such as using the same compiler VSCode uses internally makes it almost assured that we will be able to process files in the same way as VSCode already does, increasing consistency for the end user.

5.5 Semantic identifier resolution

In order to detect references, we need to be able to resolve identifiers to variables or declarations (if they originate from an environment or a library). There are two ways in which we need to resolve identifiers. The most apparent one is when identifier syntax occurs in the code – for that, we need to resolve what the identifier binds to in its lexical context. The second request we need is for imports – when a module imports an exported variable from another module, it creates a binding in this module which identifiers resolve to. The resolution we need here is to resolve the import to the original, exported variable, so we can add the use as a reference to the original variable instead of the import.

Implementing this ourselves could be problematic with our goal of reliably supporting a subset of the language syntax because many syntaxes replace the current LexicalEnvironment, which is used for identifier resolution, as was described in sections 3.3.1 and 3.3.2.

As we have already decided to use the TypeScript compiler for parsing source files, it is only natural to also use it for binding resolution. In the TypeScript compiler, all variables and declarations have a unique symbol we can use to resolve what the identifier is accessing. Using the TypeScript compiler also gives us the benefit of being able to resolve symbols of properties. While tracking properties is outside of our goals, resolving their symbols on a 'best-effort' basis allows us to better replace the 'Find All References' tool, which does work for properties.

5.6 Detecting operations performed on symbols

Detecting a meaning of an operation performed on a symbol becomes harder as it crosses from simple syntax and binding analysis to semantics. In the TypeScript compiler, all semantic logic happens within the 'Type Checker' phase, and the amount of data visible from the outside is extremely limited compared to data the compiler uses internally. This means that the only two things we can reuse from the Type Check phase of the TypeScript compiler are type resolutions and Transient Symbols (see Section 4.7). The flow analysis performed during this phase is also not exposed by the compiler. This means that this part of the analysis we will have to implement fully ourselves.

First, we need to define what kinds of operations we want to track. Similar tools often categorise operations into Reads, Writes, and Calls. These tools are meant to answer developer questions like "Where is this method called or type referenced?", "Where are instances of this class created?", or "What data is being modified in this code?". [1]

As the aim of our tool is to categorise references, we will want to provide more detailed categories of how variables are used. For example, if we considered only the Reads, Writes, and Calls categories, then the object in the code in Listing 6 would only be considered to be read, negating the benefits of our tool versus the existing 'Find All References tool'.

Here is a more detailed explanation about why has the code in the Listing 6 been categorised the way it has.

- 1) The variable 'myVar' is assigned a new instance – this is clearly a write to the variable.
- 2) The object in 'myVar' is retrieved and then converted to a string. The string is then passed to a function as an argument.
- 3) The object in 'myVar' is retrieved, and then property 'valid' is read from it. The only important part here is that the value of the variable is read; the property read is not considered.
- 4) The object in 'myVar' is retrieved and then modified by setting a property on it. As we are only analysing the variable itself.
- 5) The object in 'myVar' is retrieved and passed to function with a potentially unknown implementation. As objects are always passed by reference, this might do arbitrary operations that will be observable on 'myVar' too.
- 6) The object in 'myVar' is retrieved, then the property 'destroy' is read, and the value of this property is called. As the called expression is a property

Listing 6 An example of a reference categorisation to Read/Write/Call.

```
// The query is finding all references of 'myVar' variable
const myVar = new ClassA(); // 1: Write

console.log(`${myVar}`) // 2: Read
if (myVar.valid) { // 3: Read
    myVar.config = 42; // 4: Read
    doSomething(myVar); // 5: Read
    myVar.destroy(); // 6: Read
}

const myVarReference = myVar; // 7: Read
myVarReference.config++; // 8: Not a 'myVar' reference
```

reference, the object in 'myVar' is also used as 'this' during the call (see Section 3.2.5).

- 7) The object in 'myVar' is retrieved and stored in the 'myVarReference' variable, which can be used by other code arbitrarily.
- 8) This isn't a 'myVar' reference, as we track the usage of symbols, not instances. Tracking instances might be possible in this simple case, but when variables are not constant, it would require reliable flow analysis.

As seen from this example, this level of granularity doesn't help much except for skipping any potential references in types, which don't have an impact on code execution (in TypeScript, one can write type containing 'typeof <variable name>', which resolves to the type of the variable).

Therefore we propose the following categorisation of actions performed on symbols. These are based on a combination of *Asking and Answering Questions During a Programming Change Task* thesis, [1] operations as they are defined by the ECMAScript specification, personal experience, and intuition.

- Set: This is equivalent to the 'Write' operation shown earlier. The object in the variable is not modified, but instead, the value of the variable itself is replaced by a different one.
- Modify: This marks situations where the object in a variable is modified. The most common way of modification is setting a property to a different value or deleting a property using the 'delete' operator.
- Use: For situations where the object in a variable is read, and then the value of the object is immediately used in some way. For example, it might be

converted to a primitive value, compared against another value, iterated in a loop, or a property of the object might be read. An important aspect of this use case is that after the action is performed, the object itself is not accessible by other code.

- Reference: This marks code where the object in the variable is passed by reference either to another variable or function argument or is returned/yielded from a function. This operation is important because it marks places where the track of the object is lost by the analyser, as potentially unknown code might perform operations on it.

- Call: The value in the variable is used as a function and called.

- Call method: This is a common combination of 'Use' and 'Reference' – Object in a variable is read and a property is read from it. The property is then called, and the object itself is used as 'this' during the call.

- Construct: The value in the variable is called using the 'new' operator. While this might appear similar to the Call use-case, and in many aspects it is, the ECMAScript specification differentiates between these actions in several important ways. One important difference is that if the callee is a property reference, then, during the call, the base object is used as 'this', while during construction it isn't. There are also several built-in objects that allow both to be called as a function and used as a constructor but behave differently in either case. An example of this is the 'Date' object, where construction returns a new Date object, while a call returns a string.

- Unknown: This is a fallback action category used when a reference to the variable is detected inside an unknown or unsupported syntax.

These are the categories shown as a summary in grouped views. When viewing the list of actual actions, some categories might show more details.

With these categories, we re-analysed the same code as in Listing 6. The result of this can be seen in Listing 7. The text in parenthesis is the detail shown for each action when viewing individual actions in the reference list.

A single expression can result in multiple actions, even of different categories. Examples of this are unary update expressions ('a++', '++a'), which read the value, convert it to a number (use), and then set a new value (set).

Listing 7 An example of a reference categorisation using more granular categories.

```
// The query is finding all references of 'myVar' variable
const myVar = new ClassA(); // 1: Set

console.log(`${myVar}`); // 2: Use (Use as string)
if (myVar.valid) { // 3: Use (Get property 'valid')
    myVar.config = 42; // 4: Modify (Set property 'config')
    doSomething(myVar); // 5: Reference (Used as argument)
    myVar.destroy(); // 6: Call method ('destroy')
}

const myVarReference = myVar; // 7: Reference
myVarReference.config++; // 8: Not a 'myVar' reference
```

5.7 Detecting operations performed on call arguments

Similar to variables, we can use this detection for detecting actions on call arguments. We will analyse the use of each argument independently, identifying arguments by position or by being a 'rest' argument. There are, however, several key differences in how we treat arguments. One such difference is performing a 'Set' on an argument – it doesn't affect the object that was passed as an argument but replaces the value with a different one, which complicates tracking which references of the variable used as an argument contain the original value and which don't. In cases when we cannot confirm if the argument contains the original value, we will display them as an assignment to the argument variable but won't continue analysing the variable itself as that might lead to confusing results in cases where the argument actually cannot reach the call.

We will also extend similar logic to when a tracked object is assigned to a constant variable – we will allow expanding such assignment to continue tracking the value through the indirection.

Chapter 6

Analyser implementation

In this chapter, we design an implementation architecture for our solution. This builds primarily on the aspects we discussed in the previous chapter. We also describe some parts of our solution in greater detail, including a few attempted approaches that didn't work and why.

6.1 Architecture

Because we are creating a Visual Studio Code extension, some of the architecture is dictated by the extension API. [6] For example, we shouldn't execute any code when our files are loaded; instead, we need to export an activation function, which is called when our extension is activated. All setup and even registration must happen as a result of this function being called.

The primary way for users to interact with our extension is by running commands (those can be executed directly, or buttons in the UI can be configured to execute them, this will be discussed in Chapter 7). Commands are statically defined in the extension manifest, but our activation function is responsible for registering handlers for those commands. The handler is simply a function, but we chose to instead create an interface and make all commands a class. This allows us to encapsulate some behaviour common to all commands, like the ability to indicate the progress of long-running commands. It also standardises the way of adding command handlers in our project. All commands also hold references to global instances they will need for execution.

At the core of analysing a TypeScript project is an 'AnalysisProject' class. This class is responsible for the TypeScript equivalent – 'Program' class, and for containing analysis results and updating them when necessary by clearing old analyses and re-running the analyser on the project.

There can be multiple projects loaded at the same time – all loaded projects are contained in a ‘ProjectResolver’ class. This class is responsible for managing active projects, disposing of them when necessary, and making sure there are no duplicate projects – only one project per config file. It also caches the results of resolving files to projects to increase responsiveness.

6.2 Project resolution

As was discussed in Section 5.3, before we can start analysing the program, we need to define the scope for the analysis. This means detecting source files that should be considered part of the current project and will later be checked for using identifiers exposed by the file user is asking information about.

This is implemented by having a generator of possible project configurations. At the core of this algorithm, we take the next possible config file, load it, and verify whether the requested file is included in the project. If it is, then we found our project and we can return it; otherwise, we dispose of the loaded project to free up resources and continue with the next possible project file. If none remain, then we failed to find a project for this file and an show error message to the user.

The generator of possible project configurations attempts to first use TypeScript’s in-built resolution, which works by walking up the directory tree and looking for a file with the specified name in the directory. At first, we attempt this with the standard ‘tsconfig.json’ config name. If that fails, we try with the ‘jsconfig.json’ filename, which is a name commonly used for projects written in pure JavaScript that use TypeScript for checking only, not for compilation.

If the above strategy fails, then we attempt the same strategy ESLint uses – first, we resolve the ESLint configuration for the current file. As this is non-trivial and involves configuration inheritance and possibly conditional configuration, we offload this task to the eslint library. Then we parse the resolved configuration for a list of TSConfig projects that are required for the ‘typescript-eslint’ library to function.

The use of a generator is important, as resolving the configuration and testing individual configuration files is slow. Thanks to using a generator and stopping as soon as we find a valid match, we usually only use the fast path of finding ‘tsconfig.json’, only rarely needing to use the slow resolution using ESLint.

6.3 Selecting files to analyse

When the TypeScript compiler loads a project configuration, it also parses all loaded files, performing the first three stages as described in Section 4.7. All other compilation stages are only performed when necessary. However, while the Type

Check stage is also lazy, only analysing code as necessary, running any query on the Type Checker requires creating it first, which performs Binding on all source files. [29, 28]

This means that for our analysis, there is little benefit in only selecting a subset of files to analyse, as doing so won't improve our asymptotic complexity in relation to the source file count. Therefore we perform a full walk of the project before answering any queries about references or providing information about functions. This part of the analysis is not dependent on the query and can be freely reused by any following queries, increasing their performance significantly and allowing for a smoother and more interactive experience at the cost of a slightly longer startup time.

We do, however, skip declaration files as these do not contain any executable code and therefore cannot contain runtime references to the analysed symbols. We also skip library files (where we can detect them by detecting the presence of the 'node_modules' folder in the path to those files), as any library files (if they are not declaration files, which are already skipped) are likely to be compiled code, while this tool is designed to work with source code containing type information. Libraries also cannot reference the application code directly, and analysing the exact implementation of library functions can be problematic, as it might change across library versions without notice.

Therefore we will treat any object passed to a library as simply being 'Referenced'.

6.4 Analysing a syntax tree

As one of our primary design goals is performance, we need to design the analysis such that it is performant on large and complex codebases. For this, we adopted a similar strategy as is used in the TypeScript compiler: The analysis that is performed on all source files doesn't contain any loops (with the exception of symbol resolution) – it performs a single walk over the the syntax tree of all analysed files, saving the rules in a way that they can be easily used later by more specific analysis without having to process any part of syntax tree again.

This is implemented using a visitor pattern, where there is a function defined for processing each supported syntax kind and one fallback function that performs analysis of unsupported syntax, which is done by analysing child nodes and marking their results as "used in an unknown way". As JavaScript doesn't have any function overloading capabilities, there is a need to implement jumping to the correct analysis code ourselves. This has been implemented by having a map of syntax kinds[1] to functions and a central function that finds a specific

[1]In the TypeScript compiler, syntax kinds are represented using a numeric enum. The version

Listing 8 An example of an update statement and the matching syntax tree.

```
variable.property.anotherProperty *= 42;

ExpressionStatement
\---BinaryExpression
    +---PropertyAccessExpression
    |   +---PropertyAccessExpression
    |   |   +---Identifier ("variable")
    |   |   \---Identifier ("property")
    |   \---Identifier ("anotherProperty")
    +---PlusEqualsToken
    \---NumericLiteral (42)
```

function based on the kind of the node and calls it, otherwise performing the default analysis.

During this analysis, we need some way of passing data between nodes because each action that happens in the code is a combination of the actual action and the symbol it is performed on.

6.4.1 Experiment: Passing the necessary data from parrents to descendants

In the first implementation of this, we attempted to consider an 'analysis context', which contained what action is being performed on the code. Any syntax could then create a modified variant of this context (for example, adding a mark that this node is being written to) and process its child nodes using this context.

Problem: Nodes that perform multiple operations on a single node

In Listing 8, there is an example of a binary expression with the '*=' operator. The behaviour of this expression is to read the value on the left side, convert it to a number, then read the value on the right side, convert it to a number, multiply the numbers, and write the result to the reference provided by the left side. When using the concept of passing the context top-down, this can be problematic, as it would either require visiting the left side twice (which would produce inaccurate results, as both 'variable' and 'variable.property' are only read once during runtime) or making the state that is passed down more complex. Making the state more complex became problematic, as then all statements had to consider all possibilities of how they are used and how it affects their descendants.

used in this project recognises 362 different syntax kinds

6.4.2 Experiment: Passing the necessary data from descendants to parents

The second approach we tried was that analysing a node produces a result which describes possible normal completions similar to it being interpreted (as described in Section 3.3.2). This allows us to only visit each node exactly once while maintaining a fairly simple state that is being passed – either a value or a symbol reference. The only complex case here is a "union" of completions, which contains multiple possible completions and performs any action on each of them individually.

Problem: Evaluation of some nodes is context-sensitive

With this approach we encountered the problem that some syntax kinds have different meanings based on their location. The most notable ones are object and array literals, which in the TypeScript tree represent both actual literals used to create objects and arrays, but also can be present on the left side of the assignment operator and in function parameter declarations, where they behave as a binding pattern (also called destructive assignment). This case is evaluated by getting the value that is assigned and matching its properties to the binding pattern properties, assigning the property values to the same-named variables. There is also a separate syntax with a similar meaning when the variables are being created during a destructing assignment.

An example of a more complex binding pattern situation is shown in Listing 9. There object 'globalThis.Math' is read and passed to the destructing pattern. The value of property 'PI' is assigned to the variable 'PI', and the value of property 'acos' is assigned to the variable 'invCos'.

6.4.3 The final solution

The final solution is a combination of the above methods. Most of the nodes pass evaluation data from descendants to parents, but some nodes also perform analysis on their children in all or some cases instead of processing them using the corresponding visitor functions. Examples of this are the above-mentioned assignment operator and parameters, object literals that also process their property assignments, and class declarations that aggregate and reorder their properties based on being a function, a field, a static field, or a static block before analysing them to match execution order during runtime.

Listing 9 An example of a binding pattern and the matching syntax tree.

```
let PI, invCos;
// Syntax tree of only the following statement is shown
({
    PI,
    acos: invCos
} = globalThis.Math);

ExpressionStatement
\---ParenthesizedExpression
    \---BinaryExpression
        +---ObjectLiteralExpression
        |   +---ShorthandPropertyAssignment
        |   |   \---Identifier ("PI")
        |   \---PropertyAssignment
        |       +---Identifier ("acos")
        |       \---Identifier ("invCos")
        +---EqualsToken
        \---PropertyAccessExpression
            +---Identifier ("globalThis")
            \---Identifier ("Math")
```

6.5 Function interactions analysis

In order to answer a query about how a function interacts with outside parts of the program, we need to detect which function each action belongs to and which symbols are "outside" of the function.

This is implemented by the analysis context containing the current "flow container", which represents an area of code which can be executed by any code that has a reference to it. Examples of these are the source file itself, any variant of a function declaration, class methods, and a class constructor. Any action that is performed as a result of a specific container being executed is added to the container.

While flow containers generally match a subtree of a syntax tree without subtrees of nested flow containers, there are exceptions to this. The most important exception is a class declaration. Classes can define both static members and instance fields – static members are initialised during declaration and so belong to the flow container that contains the class, but instance field initialisers are executed during the construction of the class instance, so they belong to the constructor flow container.

A flow container doesn't even have to be a specific node in the syntax tree. An example of this is (again) classes, which can have implicit constructors that only call the constructor of a base class (if any) and run field initialisers.

6.5.1 Arguments

We also need to resolve what happens to the individual arguments passed to the function. As we resolve arguments by index, we need to resolve each possible index independently and then what happens to the 'rest arguments'. This is only evaluated lazily when the query requires doing so and can have several results:

- If the function contains a reference to the 'arguments' object then we mention every usage of the 'arguments' object as a possible use, as we cannot track access to elements of an array.
- When the position of the argument being analysed matches a destructing pattern, it is processed the same as a destructive assignment.
- When the position of the argument being analysed matches an identifier, and the identifier is never 'Set' and only accessed from this functions flow container (not from a nested function), then the uses of the argument are the uses of the identifier.
- When the position of the argument being analysed matches an identifier, but the identifier is either set or referenced in another flow container, then we treat the usage of the argument as an assignment to this identifier.

When we are displaying the usage of an argument or a variable, its usage a as call argument can be resolved further if we can be sure about the function implementation. This is checked by the call resolving to a single symbol (ignoring aliased symbols) and checking that we know all the locations of possible writes to the symbol – it must be either a 'const' variable or a local variable (either defined in a function or in an ES module) for which we know about exactly one 'Set' use and no 'Unknown' use.

6.6 Syntax support

As TypeScript recognises 362 different syntax kinds, some of which have multiple meanings, it is not realistic to support all of them within the scope of the project, as that would mean having to catch up to the 9 years of TypeScript development. [28] Instead, we chose three approaches for selecting the syntax kinds we want to support the most.

6.6.1 Syntax important to our analysis

The first are syntaxes that are most important to our primary goal of finding references of identifiers and functions they belong to, that is, syntaxes that access or manipulate the current VariableEnvironment or LexicalEnvironment, or are a flow container. For ES2023, this means primarily the 'this' expression and syntaxes that call the internal method 'ResolveBinding'. For flow containers, this means any syntax that creates an object with either '[[Call]]' or '[[Construct]]' internal method, which consists primarily of all kinds of function and method declarations, and class declarations. This also includes any syntax that is necessary to properly support these, such as 'source file', 'statement list', and 'block' for detecting bodies and variable and parameter declarations.

6.6.2 Popularly used syntax

As the second factor for prioritisation, we chose to use empirical data based on public GitHub repositories. We scanned the 100 most popular (most starred) public repositories on GitHub, scanning each file and counting encountered syntax tokens using the 'forEachChild' visitor pattern and accumulating the total seen count. This data was used to choose more syntax kinds to support.

6.6.3 Other supported syntax

The third reason for implementing particular syntax was some relation to other supported syntax, such as completeness of possible children of some syntax or experimentation convenience. Examples of this is implementing all the possible class children and unary and binary operators, even if some are rarely used. Finally, some syntax support was implemented simply because it was part of some of the experiments we performed.

6.6.4 Default analysis behaviour

For syntax nodes which we don't support, there is a default behaviour. This is to visit all children and try to analyse those recursively. Completion values returned from the children are then marked as used in an 'Unknown' way, and a special 'Unsupported completion' is returned. This completion causes a warning in any node that uses it as a value (for example, property assignments and calls). Furthermore, the flow container in which this occurs is marked as containing unsupported syntax, which prevents doing a more detailed analysis of its interactions, instead keeping the unfiltered results in accordance with our goal to display actions when there is a doubt about whether they are used.

6.6.5 Intentionally unsupported syntax

Some syntax is intentionally unsupported due to its unpredictable behaviour or due to it being a legacy syntax that is no longer recommended to use. Examples of legacy, unsupported syntaxes are the 'caller' property, and the 'with' statement. Examples of syntax unsupported due to unpredictable behaviour are the 'arguments' object, direct 'eval' calls and JSX syntax that depends on the JSX interpreter.

6.7 Testing

Besides manual testing, we also use automated unit and integration testing. It is implemented using the Mocha framework. The testing downloads and start a full VSCode instance to also be able to test usage of VSCode API.

While the tests are not comprehensive, they do provide an automated way of testing for regressions when dependencies are updated. This helps with our goal to make the extension easily maintainable and allows us to easily update the version of the TypeScript compiler we are using.

Chapter 7

User interface

In this chapter, we discuss how our tool integrates into the Visual Studio Code from the user's point of view. We also design an interface for requesting data from our tool and for displaying results.

7.1 Integrating into Visual Studio Code

As our application is an extension, any user interface we create has to be integrated into Visual Studio Code using its API. The API provided by VSCode is powerful, allowing us to even create fully custom HTML-based views. It also provides many APIs for existing UI elements, such as creating buttons, trees, panels, notifications, and more. [6]

Running commands exposed by our extension can happen in three ways: As a result of interacting with a UI element we provide, by running a command from the command palette (which can always be accessed by pressing 'F1'), or by having a keybinding assigned to our command.

7.2 Example code for this section

Throughout this section, we will display various aspects of the interface on the code shown in Listing 10.

7.3 Interfaces of similar tools

Despite the ability of computers to generate visualisations containing condensed information, such as graphs, diagrams, and plots, the more complex solutions tend to fail to gain traction and developers continue showing a preference for

Listing 10 An example code for analysis.

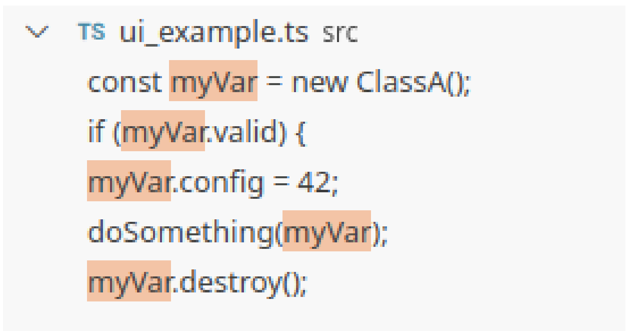
```
export class ClassA {
    public valid: boolean = true;
    public config: number = 0;

    public destroy(): void {
        this.valid = false;
    }
}

export function doSomething(aInstance: ClassA): void {
    aInstance.config--;
}

export function ReferenceResolutionExample(): void {
    const myVar = new ClassA();

    if (myVar.valid) {
        myVar.config = 42;
        doSomething(myVar);
        myVar.destroy();
    }
}
```



```
TS ui_example.ts src
    const myVar = new ClassA();
    if (myVar.valid) {
    myVar.config = 42;
    doSomething(myVar);
    myVar.destroy();
```

Figure 7.1 Result of a 'Find All References' query on the variable 'myVar' from Listing 10.

more simple visualisations, such as tree browsers. There are cases of successful graphical visualisation tools, such as UML, but those tools are relatively rare and specialised for their purposes. [37]

Because of that and because one of our goals is replacing the existing 'Find All References' and 'Show Call Hierarchy' tools, it might be a good idea to follow a similar design as these successful tools do. In Visual Studio Code, they are implemented using a tree view, which is one of the UI primitives provided by the API. [5, 6]

An example result of performing the 'Find All References' query can be seen in Figure 7.1.

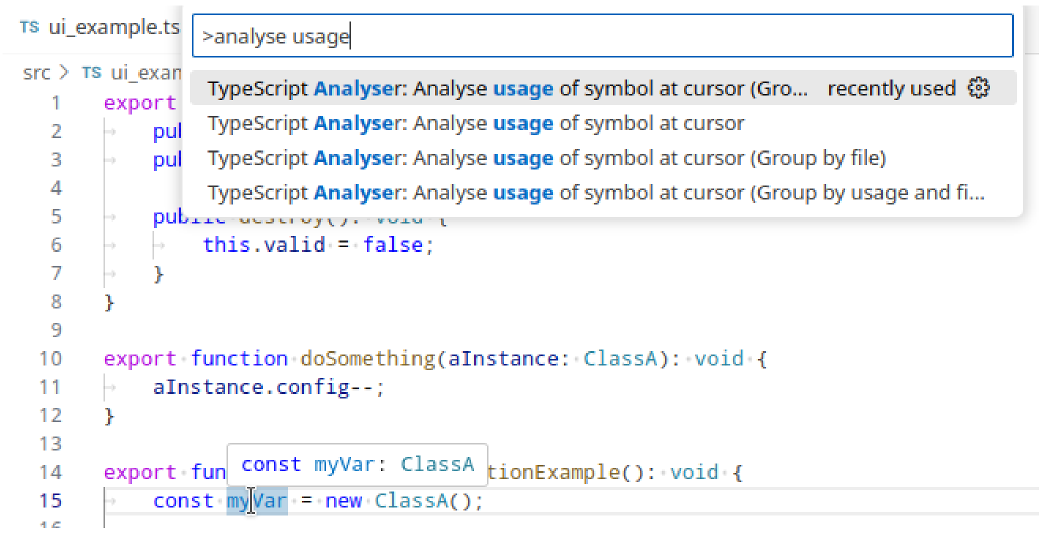
7.4 Request interface

In order to make a query to our analyser, the user must first select a symbol they are interested in and then run a command to show the requested visualisation. Here we again chose a similar way as the existing tools – for the user to position the cursor in the editor on a position containing the symbol they are interested in, followed by invoking a command from the command palette to find usages.

An example of running a command from our extension can be seen in Figure 7.2. The editor cursor is positioned on the 'myVar' text on line 4 as indicated by the mouse cursor (the cursor disappears when the command palette is opened).

Unlike built-in tools, we chose not to provide default keyboard shortcuts for our commands due to the high potential to conflict with existing tools or extensions. Instead, users can use the shortcut editor to assign a shortcut to our commands, potentially reassigning the shortcuts for finding references if they choose to do so.

A description of all the commands the tool provides can be found in Appendix A.



```
TS ui_example.ts
>analyse usage
src > TS ui_exam
TypeScript Analyser: Analyse usage of symbol at cursor (Gro...   recently used
TypeScript Analyser: Analyse usage of symbol at cursor
TypeScript Analyser: Analyse usage of symbol at cursor (Group by file)
TypeScript Analyser: Analyse usage of symbol at cursor (Group by usage and fi...
```

```
export
    pul
    pul

    public destroy(): void {
        this.valid = false;
    }
}

export function doSomething(aInstance: ClassA): void {
    aInstance.config--;
}

export fun const myVar: ClassA tionExample(): void {
    const myVar = new ClassA();
```

Figure 7.2 Flow of running a command from our extension.

7.4.1 Implementation of symbol selection

Deducing a symbol from the position of the cursor has several difficulties. One difficulty is that finding a corresponding syntax node is non-trivial, as multiple nodes can overlap with the cursor's position. Namely, for any nodes that overlap, any parent node also overlaps. We can solve this by selecting the lowest node, which has a high chance of corresponding to the expected symbol. Another problem is that cursor is always positioned between two characters, which might be the edge between two nodes. The third problem is that in the TypeScript syntax tree, there might be zero-length nodes inserted by the parser.

In TypeScript language tools, this is solved by having a rather complex adjustment mechanism that adjusts the selected position based on complex rules and types of nodes. This implementation is, however, not exposed in the API, and makes use of many other internal compiler APIs, making it hard to extract it.

Therefore we chose to use only a simple variant of the algorithm TypeScript language tool uses, which searches for the first lowest node in the syntax tree that touches the cursor, skipping specific kinds of nodes and finding a 'best match' for the position. While this is a worse solution than TypeScript offers and places greater requirements on cursor positioning when invoking commands of our extension, we have not yet encountered a case when this prevents selecting the desired target for the command.

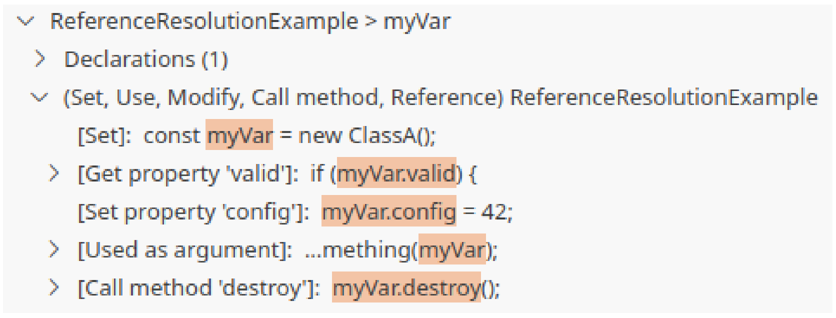

```
ReferenceResolutionExample > myVar
  Declarations (1)
  (Set, Use, Modify, Call method, Reference) ReferenceResolutionExample
    [Set]:  const myVar = new ClassA();
    [Get property 'valid']:  if (myVar.valid) {
    [Set property 'config']:  myVar.config = 42;
    [Used as argument]:  ...mething(myVar);
    [Call method 'destroy']:  myVar.destroy();
```

Figure 7.3 Result of a ‘Analyse usage of symbol at cursor’ query on the variable ‘myVar’ from Listing 10.

7.5 Result interface

Our extension provides several views, depending on the command used. Some commands have multiple variants that group the results differently to make it easier to find relevant results. All these views are implemented using the tree browsing API provided by VSCode. Clicking on a result focuses on the relevant code that created this entry.

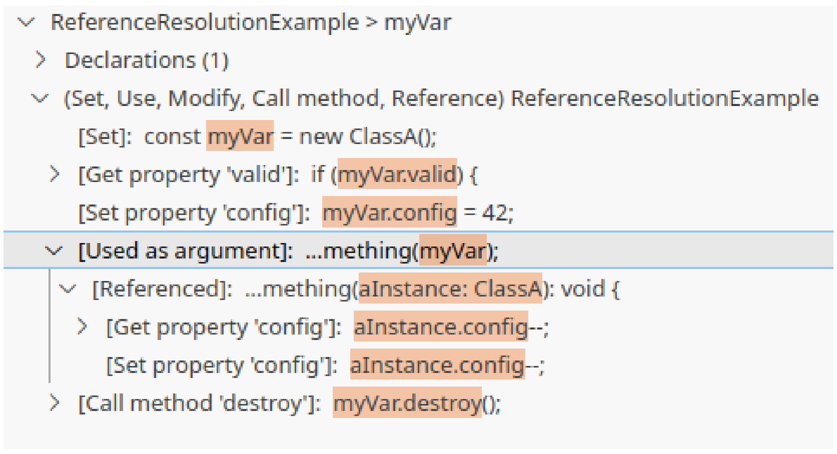
An example result of an ‘Analyse usage of symbol at cursor’ query can be seen in Figure 7.3. It can be seen that this result is similar to the result of ‘Find All References’, but also includes extra details, like the function that performs these cations and what actions are performed.

7.5.1 Lazy generation of nodes

One of the features the tree browsing API provides is that children of nodes are only requested when the node is displayed and expanded (nodes define if they are not expandable at all or if they are collapsed or expanded by default). This allows displaying more details that can be displayed by expanding specific nodes, including details that require extra calculations, without a performance penalty if these details are not requested.

An example of this can be seen in Figure 7.4. There the highlighted line represents passing a value to a function that has been successfully resolved. Users can expand such node to show how the function uses the argument – in this case, it is assigned to a variable named ‘aInstance’, followed by reading a property ‘config’ and modifying the passed object by updating the same property.

The benefit of lazy generation here is, that the usage of the argument is only analysed when the user expands the node, increasing the performance of the first query.



```
ReferenceResolutionExample > myVar
  Declarations (1)
  (Set, Use, Modify, Call method, Reference) ReferenceResolutionExample
    [Set]:  const myVar = new ClassA();
    [Get property 'valid']:  if (myVar.valid) {
    [Set property 'config']:  myVar.config = 42;
    [Used as argument]:  ...mething(myVar);
      [Referenced]:  ...mething(aInstance: ClassA): void {
        [Get property 'config']:  aInstance.config--;
        [Set property 'config']:  aInstance.config--;
    [Call method 'destroy']:  myVar.destroy();
```

Figure 7.4 Result of expanding an argument usage node from Figure 7.3.

7.5.2 Visible names for symbols

One of the drawbacks of the existing 'Show Call Hierarchy' tool is that functions are displayed only as a function name. In case of methods the name of the parent class is shown too. This can be problematic in cases where proper naming conventions are not used or when function names make sense only depending on their context, such as the file they are in.

We solve this by providing a mechanism that shows the full path to the referenced function, including their originating file.

7.5.3 Displaying actions

On the lowest level, we display the actions. We don't attempt to name individual actions; instead, we display a preview of the code responsible for the action, including a few characters in front of the relevant code and the rest of the line behind the code while highlighting the relevant part. The code is prefixed by a short description of the action type (such as 'Call', 'Use as string', or 'Get property'). In some cases, we also show extra information, such as the name of the property being accessed or modified. This is a behaviour similar to the search tool or the find references tool. We hope for this to allow quicker use of the results, using some results without having to focus on them in the editor.

Chapter 8

Evaluation

In this final chapter we analyse the tool we created, the use cases it supports and the benefits and drawbacks of using it over other tools. We also perform an experiment measuring the frequency of syntax usage in popular repositories and use it to measure the syntax support of our tool.

8.1 Requirements to use

Our solution makes certain assumptions about the project it is being used on. These assumptions require a project that can be compiled using a standard, unpatched TypeScript compiler from Microsoft and that the project is configured using either a ‘tsconfig.json’ file, ‘jsconfig.json’ file, or has a working ‘typescript-eslint’ setup with support for type-based rules.

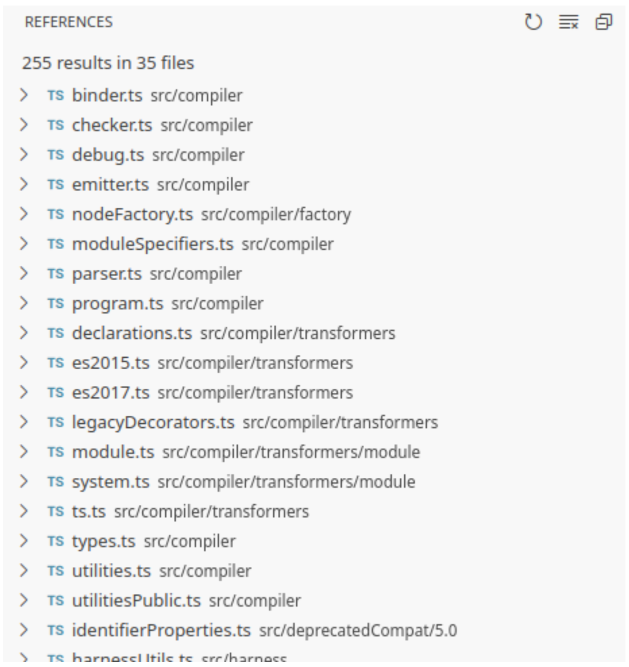
We also don’t support projects requiring a newer TypeScript compiler version than the version packed with our extension.

8.2 Example uses on real projects

In this section we will show a few examples of how our tool helps on existing projects.

8.2.1 Finding code that sets node flags in the TypeScript compiler

The TypeScript compiler defines a ‘Node’ interface. One of the properties this interface define is a ‘flags’ property containing information about the type of the node. During the development of this tool, we wanted to know which stages of compilation modify these flags in order to understand if we can reliably use them.


REFERENCES

255 results in 35 files

TS binder.ts src/compiler
TS checker.ts src/compiler
TS debug.ts src/compiler
TS emitter.ts src/compiler
TS nodeFactory.ts src/compiler/factory
TS moduleSpecifiers.ts src/compiler
TS parser.ts src/compiler
TS program.ts src/compiler
TS declarations.ts src/compiler/transformers
TS es2015.ts src/compiler/transformers
TS es2017.ts src/compiler/transformers
TS legacyDecorators.ts src/compiler/transformers
TS module.ts src/compiler/transformers/module
TS system.ts src/compiler/transformers/module
TS ts.ts src/compiler/transformers
TS types.ts src/compiler
TS utilities.ts src/compiler
TS utilitiesPublic.ts src/compiler
TS identifierProperties.ts src/deprecatedCompat/5.0
TS harnessUtils.ts src/harness

Figure 8.1 Results from the 'Find All References' tool when analysing 'flags' property of the 'Node' interface in the TypeScript compiler.

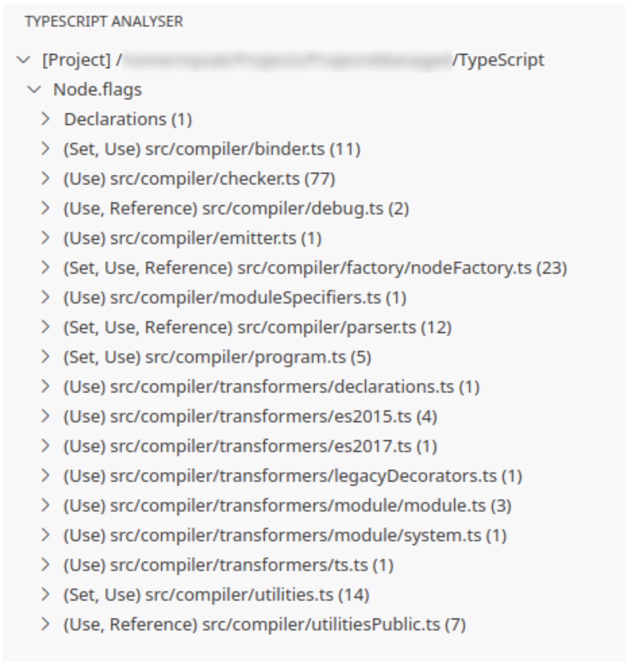
As the compilation stages of the TypeScript compiler are split into files we only needed to find files that set these flags.

Using the in-built 'Find All References' tool, however, returned 255 results across 35 files which we would have to go through manually. By using our tool (to be exact the 'Analyse usage of symbol at cursor (Group by file)' command) we received a concise summary of which files perform which operations. This can be seen in Figures 8.1 and 8.2.

Following that we wanted to find functions that modify these flags. To do so we used the 'Analyse usage of symbol at cursor (Group by usage)' command from our tool. The results of this can be seen in Figure 8.3. Expanding the node named '(Set)' displays 33 functions that manipulate the 'flags' property.

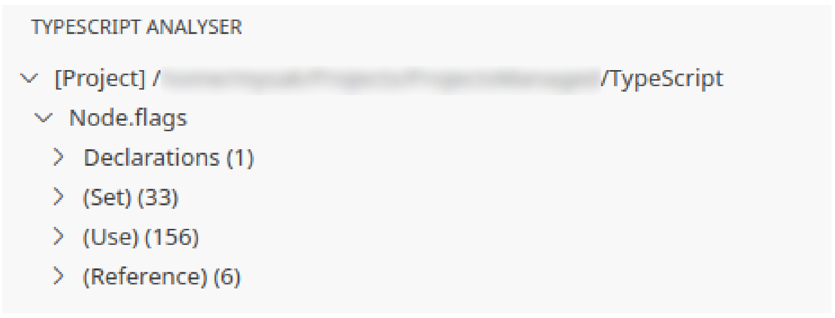
8.2.2 Overview of large function's interactions in Webpack code

Webpack code contains many large functions generating emitted code. One such function we were interested in is '`renderBootstrap`'. We were interested in what data or functions this function uses besides the passed arguments and 'this'. Running the 'Show information about function' query while focused on the



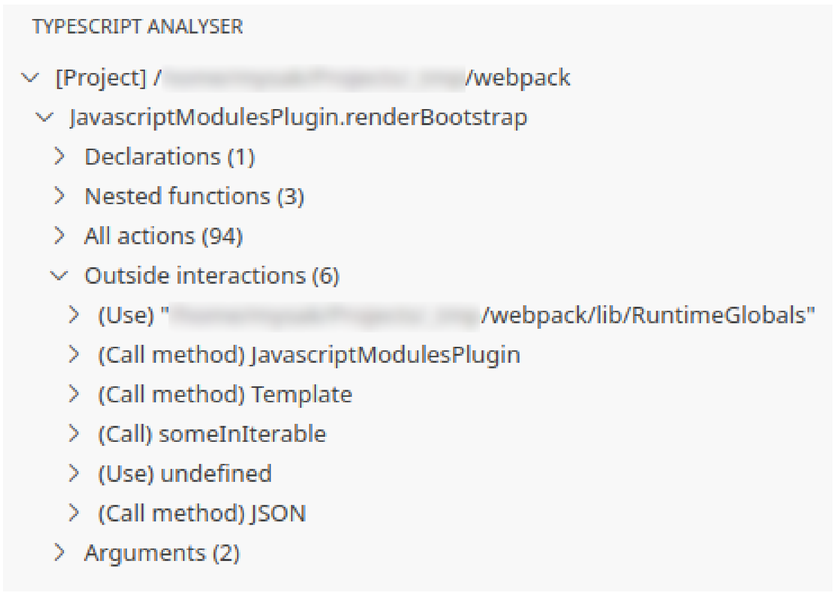
```
TYPESCRIPT ANALYSER
∨ [Project] /[illegible]/TypeScript
  ∨ Node.flags
    > Declarations (1)
    > (Set, Use) src/compiler/binder.ts (11)
    > (Use) src/compiler/checker.ts (77)
    > (Use, Reference) src/compiler/debug.ts (2)
    > (Use) src/compiler/emitter.ts (1)
    > (Set, Use, Reference) src/compiler/factory/nodeFactory.ts (23)
    > (Use) src/compiler/moduleSpecifiers.ts (1)
    > (Set, Use, Reference) src/compiler/parser.ts (12)
    > (Set, Use) src/compiler/program.ts (5)
    > (Use) src/compiler/transformers/declarations.ts (1)
    > (Use) src/compiler/transformers/es2015.ts (4)
    > (Use) src/compiler/transformers/es2017.ts (1)
    > (Use) src/compiler/transformers/legacyDecorators.ts (1)
    > (Use) src/compiler/transformers/module/module.ts (3)
    > (Use) src/compiler/transformers/module/system.ts (1)
    > (Use) src/compiler/transformers/ts.ts (1)
    > (Set, Use) src/compiler/utilities.ts (14)
    > (Use, Reference) src/compiler/utilitiesPublic.ts (7)
```

Figure 8.2 Results from our tool when analysing ‘flags’ property of the ‘Node’ interface in the TypeScript compiler and grouping by file.



```
TYPESCRIPT ANALYSER
∨ [Project] /[illegible]/TypeScript
  ∨ Node.flags
    > Declarations (1)
    > (Set) (33)
    > (Use) (156)
    > (Reference) (6)
```

Figure 8.3 Results from our tool when analysing ‘flags’ property of the ‘Node’ interface in the TypeScript compiler and grouping by use.



```
TYPESCRIPT ANALYSER
∨ [Project] /[redacted]/webpack
  ∨ JavascriptModulesPlugin.renderBootstrap
    > Declarations (1)
    > Nested functions (3)
    > All actions (94)
    ∨ Outside interactions (6)
      > (Use) "[redacted]/webpack/lib/RuntimeGlobals"
      > (Call method) JavascriptModulesPlugin
      > (Call method) Template
      > (Call) someInIterable
      > (Use) undefined
      > (Call method) JSON
    > Arguments (2)
```

Figure 8.4 Results from our tool when running the 'Show information about function' query on the '`JavascriptModulesPlugin.renderBootstrap`' method in the Webpack code.

function produced an overview shown in Figure 8.4. Opening the 'Arguments' folder in the results would show the usage of arguments.

Two notes about the output: The `(Use) "/path/..."` entry represents a module imported as a namespace. The '`undefined`' entry represents the *undefined* primitive value, but in ECMAScript, it is defined as a global variable, not a keyword.

The in-built tool to find references doesn't help with this task at all. The tool to 'Show Call Hierarchy' allows seeing outgoing calls, but those calls are not sorted at all and don't show where in the function the call happens – only that it does.

8.3 Experiment: Testing on popular repositories

In order to test our choice of syntax support prioritisation we have performed an experiment in which we scanned the 100 most starred public repositories on GitHub, which GitHub recognises as written in TypeScript and another 100 repositories identified as written in JavaScript. We have done this experiment multiple times during the development of our tool, using it to refine our choice of supported syntax further. The results described below represent the last run we performed on 16/07/2023.

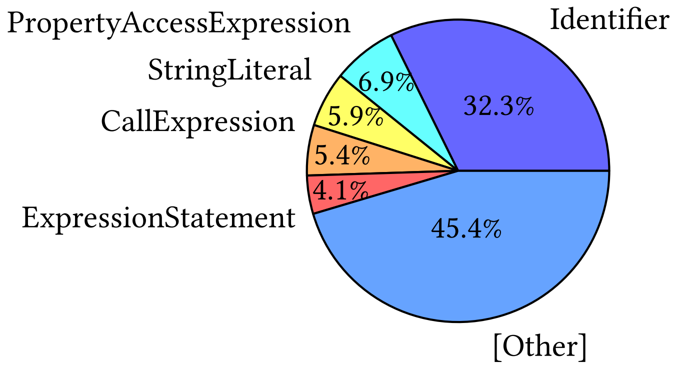


Figure 8.5 This chart shows the 5 most popular syntax tokens and the frequency of their occurrence in TypeScript repositories.

The experiment is not reproducible because the list of the 100 most popular repositories can change and because the repositories themselves are likely to change. More details about how this experiment has been performed can be found in the Appendix B, including instructions on how to run it again.

8.3.1 Results of the scan

The scan indicated how often is specific syntax used in TypeScript and JavaScript code. In the case of TypeScript repositories, a total of $82,124,196$ syntax nodes have been scanned. For JavaScript repositories a total of $100,649,046$ nodes.

In both cases, the most used syntax is 'Identifier', amounting to 32.3% and 31.1% of all syntax nodes in TypeScript and JavaScript repositories, respectively. The 5 most popular syntax nodes of TypeScript repositories scan are shown in Figure 8.5. For TypeScript repositories we also counted in how many repositories each syntax occurred in. The complete results are available in the attached data, as described by Appendix C. A summary is that 98 syntax kinds occurred in at least 95% of scanned repositories.

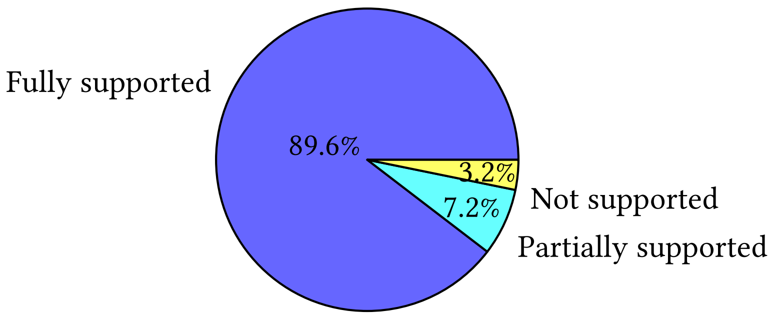


Figure 8.6 This chart shows syntax coverage of our tool on TypeScript repositories, proportional to total number of occurrences of a syntax kind.

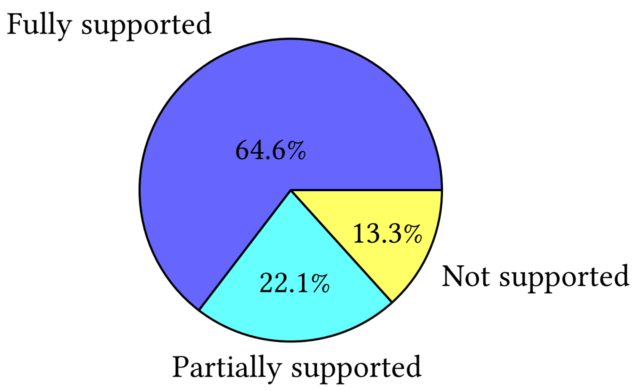


Figure 8.7 This chart shows syntax coverage of our tool on TypeScript repositories, proportional to the count of repositories the syntax occurs in.

8.3.2 Coverage

With this data, we calculated the coverage our tool has for the syntax used. We divided support for each syntax kind into three categories:

- Fully supported: Our tool supports analysing this syntax.
- Partially supported: Our tool supports analysing this syntax if the parent node is supported; this includes, for example, mathematical operators, which are only supported as part of unary and binary expressions.
- Not supported: Our tool doesn't recognise this syntax, analysing it using the fallback behaviour for unknown syntax.

Percentages of these categories have been calculated based on both total occurrences of each syntax kind, as well as the number of repositories the syntax kind occurs in. Results of this can be seen in figures 8.6 and 8.7.

Conclusion

Goals

We have met the goals of this thesis. The description of how we completed individual goals follows.

- We have analysed and implemented a Visual Studio Code extension for statically analysing TypeScript codebases.
- JavaScript is also supported by our solution to the same degree as it is supported by the TypeScript compiler.
- Our solution can analyse the uses of a value being passed to a function, even transitively, in cases where we can be sure which function is being called. The dynamics of the JavaScript language, allowing the replacement of declared functions or methods, heavily restrict which cases we can treat as reliable. There are also several cases where we chose to intentionally not provide transitive resolutions as they might lead to confusion.
- Our solution can be invoked using commands from within Visual Studio Code, and we display results using a tree-like view, which is also used by similar built-in tools. We have several different views that display results based on the query the user requested.
- We treat any usage potentially relevant to the query conservatively. In cases where we cannot be sure whether a certain action happens or about the specifics of the action, we treat is as potentially performing arbitrary action and show so to the user.
- Our solution is designed with performance in mind, caching project-wide analyses and deferring more specific ones until they are necessary for displaying results.

Future work

JavaScript and TypeScript are complex languages that are continuously evolving. Furthermore, code development is a complex subject, and there are many different approaches and project setups that we encountered during the development of this tool, which we don't fully support.

Increase syntax support

The syntax we currently support is not based on any standard or TypeScript compiler version. There is also no summary of what syntax is supported and which is not. It would be beneficial for the end user to have a summary of what syntax support to expect. This could be created by, for example, categorising all syntax kinds the TypeScript compiler recognises and documenting our support for these categories.

We also don't support JSX/TSX syntax at all because the implementation of the syntax is specific to the environment it is used in. Despite this, it is an often used syntax that is commonly used with a specific environment: React. It would be beneficial to support this specific use-case to increase our coverage.

Improve the accuracy of the analysis

At the moment, flow analysis when analysing an object passed as a function argument is quite limited and several cases prevent performing it altogether. This could be improved by using more sophisticated flow analysis to support cases like the argument variable being reassigned. There could also be improvements made for narrowing down possible paths based on known facts about the value passed to the function, based on the caller.

More testing

Our current testing setup only tests a subset of supported syntaxes and even fewer interactions between syntaxes. It would be beneficial to have more test cases by adopting existing tests from other projects, such as TypeScript or Test262, as well as more testing from the user base for various use cases we didn't consider while implementing this tool.

Bibliography

[1] Jonathan Sillito. “Asking and Answering Questions During a Programming Change Task.” PhD thesis. The University Of British Columbia, 2006.

[2] Microsoft. *TypeScript Homepage*. `https://www.typescriptlang.org/`. [Online; accessed 8-July-2023].

[3] Microsoft. *Visual Studio Code Homepage*. `https://code.visualstudio.com/`. [Online; accessed 8-July-2023].

[4] *Stack Overflow Developer Survey*. `https://survey.stackoverflow.co/2022`. [Online; accessed 8-July-2023]. 2022.

[5] Microsoft. *Visual Studio Code - Open Source Repository*. `https://github.com/microsoft/vscode`. [Online; accessed 8-July-2023].

[6] Microsoft. *Visual Studio Code API Documentation*. `https://code.visualstudio.com/api`. [Online; accessed 8-July-2023].

[7] Microsoft. *Language Server Protocol Specification - 3.17*. `https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/`. [Online; accessed 8-July-2023].

[8] *Code - OSS: Language Features for TypeScript and JavaScript files extension source*. `https://github.com/microsoft/vscode/tree/main/extensions/typescript-language-features`. [Online; accessed 8-July-2023].

[9] *CodeQL Homepage*. `https://codeql.github.com/`. [Online; accessed 8-July-2023].

[10] *Node.js Homepage*. `https://nodejs.org/en/about`. [Online; accessed 8-July-2023].

[11] *ECMAScript® 2023 Language Specification*. ECMA-262. 13th edition. Ecma International. June 2022.

[12] Ecma International. *Official ECMAScript Conformance Test Suite Repository*. `https://github.com/tc39/test262`. [Online; accessed 8-July-2023].

[13] WHATWG. *WHATWG Standards list*. `https://spec.whatwg.org/`. [Online; accessed 8-July-2023].

[14] *ECMAScript® 2021 Internationalization API Specification.* ECMA-402. 8th edition. Ecma International. June 2021.

[15] *Node.js API documentation.* `https://nodejs.org/api/`. [Online; accessed 8-July-2023].

[16] *V8 JavaScript engine Homepage.* `https://v8.dev/`. [Online; accessed 8-July-2023].

[17] *SpiderMonkey JavaScript/WebAssembly Engine Homepage.* `https://spidermonkey.dev/`. [Online; accessed 8-July-2023].

[18] *JavaScriptCore WebKit wiki page.* `https://trac.webkit.org/wiki/JavaScriptCore`. [Online; accessed 8-July-2023].

[19] *Test262 Report Project.* `https://test262.report/`. [Online; accessed 8-July-2023].

[20] *Node Package Manager Homepage.* `https://www.npmjs.com/`. [Online; accessed 8-July-2023].

[21] *NPM package.json documentation.* `https://docs.npmjs.com/cli/v9/configuring-npm/package-json`. [Online; accessed 8-July-2023].

[22] *Babel Documentation.* `https://babeljs.io/docs/`. [Online; accessed 8-July-2023].

[23] *Webpack documentation - Tree Shaking.* `https://webpack.js.org/guides/tree-shaking/`. [Online; accessed 8-July-2023].

[24] Tyler McGinnis. *JavaScript Modules: From IIFEs to CommonJS to ES6 Modules.* `https://ui.dev/javascript-modules-iifes-commonjs-esmodules`. [Online; accessed 8-July-2023]. 2019.

[25] *ClojureScript Homepage.* `https://clojurescript.org/`. [Online; accessed 8-July-2023].

[26] *State of JavaScript.* `https://2022.stateofjs.com/en-US/`. [Online; accessed 8-July-2023]. 2022.

[27] *ESLint documentation of a 'no-shadow' rule.* `https://eslint.org/docs/latest/rules/no-shadow`. [Online; accessed 8-July-2023].

[28] *TypeScript GitHub repository.* `https://github.com/microsoft/TypeScript`. [Online; accessed 8-July-2023].

[29] *TypeScript Compiler Notes GitHub repository.* `https://github.com/microsoft/TypeScript-Compiler-Notes`. [Online; accessed 8-July-2023].

[30] *TypeScript Website GitHub repository.* `https://github.com/microsoft/TypeScript-Website`. [Online; accessed 8-July-2023].

[31] *TypeScript GitHub Issue: Typescript Specifications version.* `https://github.com/microsoft/TypeScript/issues/15711`. [Online; accessed 8-July-2023].

[32] Microsoft. *TypeScript Handbook.* `https://www.typescriptlang.org/docs/handbook/intro.html`. [Online; accessed 8-July-2023].

[33] *TypeScript GitHub Wiki.* `https://github.com/Microsoft/TypeScript/wiki`. [Online; accessed 8-July-2023].

[34] Joshua Bloch. *Effective Java programming language guide.* Addison Wesley, 2001.

[35] *SWC Homepage.* `https://swc.rs/`. [Online; accessed 8-July-2023].

[36] *typescript-eslint project homepage.* `https://typescript-eslint.io/`. [Online; accessed 8-July-2023].

[37] Steven P Reiss. "The paradox of software visualization". In: *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis.* IEEE. 2005, pp. 1–5.

Appendix A

Using the extension

The extension has been tested on Manjaro Linux and a clean Windows 10 (22H2) installation. The tested hardware had a x86_64 architecture and 8GB of RAM. Visual Studio Code versions 1.79.2 and 1.80.1 were tested.

Installation

The included digital attachments include a prebuilt version of the extension – 'ts-usage-analyser.vsix'. More details about other included attachments can be found in Appendix C.

If one doesn't want to use the prebuilt version, the instructions to build the extension are included in the programmer documentation in the source folder. Another alternative is to use a version built by a GitLab CI script:

`https://gitlab.mff.cuni.cz/pacalm/ts-usage-analyser/builds/artifacts/master/browse/ts-usage-analyser?job=extension-build`

To install the extension an installation of Visual Studio Code is necessary first. Visual Studio Code can be downloaded from:

`https://code.visualstudio.com/download`

After installing and opening Visual Studio Code, the extension can be installed by going to the 'Extensions' tab, clicking on the three dots button in the top right of the extensions panel and selecting 'Install from VSIX...'. This will open a dialogue that allows selecting the 'ts-usage-analyser.vsix' file in order to install it. This button can be seen in Figure A.1.

Alternative to this menu is by dragging the extension file and dropping it over the list of extensions.

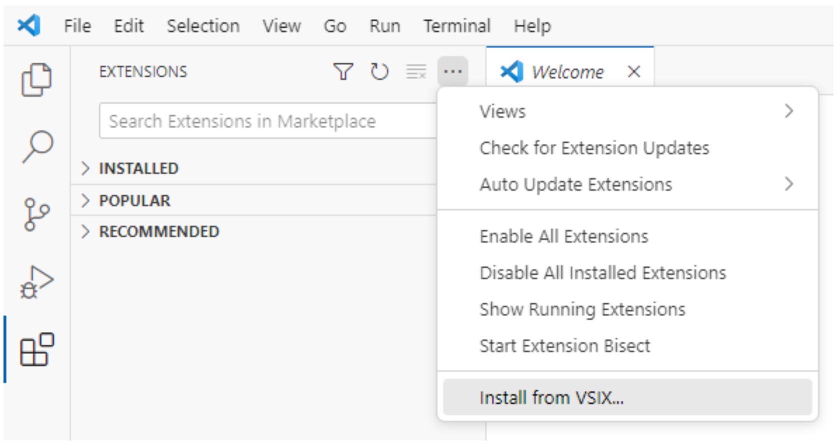


Figure A.1 Location of the button to manually install an extension.

Examples

The included digital attachments include example code that has been used to develop and test the extension. The code is present in the 'examples' folder. This code is also used for automated testing.

To open the example project open Visual Studio Code, select 'File'->'Open folder...' and select the 'examples' folder using the file picker. The extension will activate automatically when any JavaScript or TypeScript file is opened (the file must be detected as such by VSCode; this is indicated in the bottom right when any file is opened).

The example project contains several folders, each of these containing a README file describing what part of our analysis is showcased in the files in said folder.

Note, that while the extension includes all its dependencies in the vsix bundle, the example project cannot make use of these. Our tool should be able to analyse the project even without installing the project's dependencies, but in some cases there might some limitations – for example it might not be possible to open declarations of in-built methods. The example project's readme contains instructions about its dependencies.

Basic usage

The extension activates as soon as you open a TypeScript or JavaScript file. To run any usage query, position the cursor on top of the symbol you want to analyse and run one of the provided commands using command pallete.

Requirements

This extension depends on the TypeScript compiler for providing project-wide links. To perform any analysis at all, this extension needs a TypeScript configuration.

This means that this extension *requires* a presence of a 'tsconfig.json' or 'jsconfig.json' file in a directory of the source file being analysed or any of its parents. Alternative to this requirement is having an ESLint setup with 'typescript-eslint' and support for rules requiring type information. In this case it is required to open the same folder that contains ESLint config in VSCode.

Output

Most commands provided by the extension open a 'TypeScript Analyser' view on the left side when run. This view is used to display the result of each command and its contents always show the result of the last successful command that uses it.

Available commands

All commands provided by the extension are prefixed with a 'TypeScript Analyser:' prefix. This can be used to browse the available commands by pressing F1 followed by typing this prefix.

Here is a list of the provided commands and their functionality:

Analyse usage of symbol at cursor

This command displays usage of symbol the editor cursor is pointing at. The results are shown in a tree view, each entry representing usage of this symbol.

The entries are, by default, grouped by the function that performs the action. There are variants of this command to also group them by file, usage category, or both usage category and file. Each group shows a summary of categories of actions inside it in parenthesis before the name of the group.

Some entries might be expandable - this is the case when the analyser detects, that the target variable can only have value originating from this action. The entries under such expandable nodes represent actions performed on the other variable - not on the original target.

Show information about function at cursor

This command displays an overview of how the function a cursor is currently positioned in interacts with outside – how does it use arguments and variables declared outside of it.

The output is a tree containing the following folders:

- Analysis problems: This folder contains a list of problems encountered while analysing this function. If no problems were encountered this folder isn't shown.
- Nested functions: If the function contains any function declarations, expressions, or arrow functions inside, then this folder will show their list
- All actions: This folder contains a list of all symbols the function interacts with. The symbols are not sorted in any way.
- Outside interactions: This folder contains a subset of "All actions" that can be considered interacting with the environment outside of the function. This includes reading and writing variables and calling functions. This folder doesn't include properties and method calls, as can be seen by expanding the other interactions.
- Arguments: This folder contains arguments that are used by the function (identified by their order during the call)

Reload all projects

While the analyser tries to automatically detect changes in the analysed project, this detection might sometimes fail. This is a problem originating from the TypeScript compiler and usually happens when creating, deleting, or renaming files.

This command was created to handle this case - running it clears all cached data, forcing it to be reload when next query is requested.

Analyse current project

This command is more a debugging tool than intended to answer queries - it performs full scan of the current project and shows all files considered as part of the project. For each file or function it shows gathered information about it, including local and outside interactions and nested functions it contains.

Recognised actions on analysed symbols

This is a list of actions performed by code that are recognised by our analyser, grouped by the categories used to display summaries.

Set actions

This marks an action where the object in the variable is not modified, but instead, the value of the variable itself is replaced by a different one. Sources of this action include:

- Binary assignment operator (=); this includes both normal assignments to a variable and destructing assignments
- The ‘delete’ operator is treated the same as assigning ‘undefined’ to the variable
- Binary update operators (+=, -=, *=, &&=, ??=, ...)
- Unary update opertors (a++, ++a, ...)
- Variable declaration
- Function argument (we treat values passed to a function as being assigned to the implementation’s arguments)
- Catch clause varible (similar treatment to function argument, except we don’t track error propagation)
- Object literal is treated as assigning values to the object’s properties
- Class field initializers
- Function and class declarations

Use actions

This is arguably the most common category of actions. It is used in situations where the object in a variable is read, and then the value of the object is immediately used in some way. For example, it might be converted to a primitive value, compared against another value, iterated in a loop, or a property of the object might be read.

An important aspect of this use case is that after the action is performed, the object itself is not accessible by other code. Note, that we ignore special cases such as property getters and the ‘Proxy’ object - these might still cause side effects not detected by this tool.

Due to how common this action is we provide several descriptions of how exactly is the variable itself used. These are shown only as description of the line referring to the action itself - the category shown on a function performing this shows simply as ‘Use’.

- Get property: A property of the object is read
- Use as string or number: This is specific to binary operators + and += and comparision operators (<, >=, ...), as these convert the object to one of these primitives and perform different operation based on if it is a number or a string
- Use as string: The value is converted to string and used further. This happens for example for template expressions (`` `${myVar}` ``).
- Use as number: The value is converted to a number and used further. Examples of this are all arithmetic operators (except +), binary and shift operators, unary +, and unary udpate operators
- Use as property key: The value is converted to a property key (string, 'Symbol', or numeric index) and used to address a property. This primarily happens in property access expressions (`a[myVar]`) but can also happen in object literals and binding patterns (`{ [myVar]:  42 }`)
- Check truthiness: The value is converted to a boolean and used further. This includes the following cases:
  - Logical operators && and || for values on the left side. These operators use truthiness of the left side to decide which side they should use as a resulting value
  - Negation (!)
  - If, While, Do-While, and For statement conditions
  - Conditional expression condition (ternary operator)
- Check type: The value's type is used in some way. This includes the following cases:
  - Nullish coalescing operators (?? and ??=) – these check if the type is 'null' or 'undefined'; they don't check truthiness
  - The 'typeof' operator
  - The left side of the 'instanceof' operator
- Compare equality: The object is compared using one of the four comparison modes JavaScript has. This happens for:
  - Equality testing operators (==, !=, ===, !==)

  - Right side of the 'instanceof' operator (it compares equality to individual objects in the prototype chain)
  - Switch statement variable or case value

- Enumerate properties: The names of object's properties are enumerated or iterated without accessing the values themselves. This happens for the `for (... in ...)` statement and for the right side of the 'in' operator

- Iterate: The object is used as iterable and iterated in any way. This includes things ranging from `for (... of ...)` statement to spread syntax (`...myVar`). This action can be expanded to show how are individual elements used.

- Await: The object is used as a promise and awaited. This action can be expanded to show how is the awaited result used.

Modify actions

This marks actions that modify properties of an object. It happens when a 'Set' is performed on a property of the object and is shown as 'Set property'. The most common way for this to happen is assignment or update expression, but the 'delete' operator triggers this too.

This operation isn't transitive, so the example below contains two actions:

- 'Set' performed on 'anotherProperty'

- 'Modify' performed on 'property'

- The 'variable' use in here is only a 'Use' ('Get property' to be exact) – it is *not* 'Modify'

```
variable.property.anotherProperty = 42
```

Call action

The value in the variable is used as a function and called.

Call method action

This is a subcategory of 'Reference' actions, but one that both has a special meaning and happens commonly enough, that we added it as a separate category to make navigation easier.

If a call is performed on a property reference, then the object the property is read from is used as 'this' during the call. For our analysis this means loosing track of how the object used, as we cannot reliably track method implementations, so the method might perform arbitrary actions with the object passed this way.

Construct action

This action is similar to 'Call', but it is performed using a 'new' operator.

While this might appear similar to the Call use-case, and in many aspects it is, the ECMAScript specification differentiates between these actions in several important ways. One important difference is that if the callee is a property reference, then, during the call, the base object is used as 'this', while during construction it isn't.

There are also several built-in objects that allow both to be called as a function and used as a constructor but behave differently in either case. An example of this is the 'Date' object, where construction returns a new Date object, while a call returns a string.

Reference actions

This marks code where the object in the variable is passed by reference either to another variable or function argument or is returned/yielded from a function. This operation is important because it marks places where the track of the object is lost by the analyser, as potentially unknown code might perform operations on it.

Unknown actions

This is a fallback action category used when a reference to the variable is detected inside an unknown or unsupported syntax.

Appendix B

Repository scanning experiment

This appendix describes details of the repository scanning experiment from Section 8.3. The experiment is fully automated – sources for the experiment as well as our results can be found in the 'repo-scan' directory in the attached data.

The scan works by performing search on GitHub for public repositories with specified language ('TypeScript' or 'JavaScript'), ordered by number of stars in descending order. In then iterates the search until the requested count of repositories is obtained. The main branch of each repository is then shallowly cloned, or pulled using fast-forward if the repo has been cloned already.

Then all files with extensions described in Section 5.3.2 are parsed independently using TypeScript compiler and the syntax tree is walked. TypeScript offers two ways to get children of a node – 'forEachChild' and 'getChildren'. 'forEachChild' walks the nodes in semantic order and only returns nodes that have a semantic meaning to the parent. 'getChildren' on the other hand walks nodes in a way closest to the source file, including modifiers, keywords and tokens (such as commas, dots and semicolons).

As the 'forEachChild' approach is closer to our analysis, it is used for the results in Section 8.3. The attachment, however, includes result of both approaches. The 'forEachChild' approach has also been summarized based on number of repositories the specific syntax occurs in.

During the scan, the TypeScript compiler successfully parsed all files except one. This file has been excluded from the summary.

Running the experiment

To run the experiment a GitHub Personal Access Token is required, in order to be able to run search for popular repositories. After acquiring a token, the experiment can be run by entering the 'repo-scan' directory and running the following commands:

```
yarn install
GITHUB_PAT=<your token here> yarn run scan
```

Doing this requires installed Node.js with corepack enabled (tested on v18.15.0) and ‘git’ executable in the environment path.

This process can take several hours (in our case it took a bit over 3 hours on the first run), depending on the internet connection, processor, and disk used. The experiment requires a lot of harddisk space, even while using only shallow cloning (in our case the required disk space was roughly 23GB).

Appendix C

Digital attachments summary

This appendix describes the contents of the digital attachments. All mentioned data is also publically available in the following GitLab repository:
`https://gitlab.mff.cuni.cz/pacalm/ts-usage-analyser`

Extension sources

Extension sources are attached in the ‘`ts-usage-analyser`’ folder. This folder also includes:

- Readme file with user documentation similar to Appendix A
- Development documentation describing both the architecture of the extension and the build process
- Automated tests

Prebuilt extension

The file ‘`ts-usage-analyser.vsix`’ is a prebuilt version of the extension ready to be installed into any compatible Visual Studio Code instance.

More information about this is provided in Appendix A.

Examples

The folder ‘`examples`’ contains an example project with several source files to demonstrate various capabilities of our extension. This folder is also used by the automated tests the extension provides.

Repository scanning experiment

The ‘`repo-scan`’ and ‘`repo-scan-results`’ folders contain data relevant to the repository scanning experiment described in Appendix B.

The ‘`repo-scan`’ folder contains the source code necessary for performing the experiment. The ‘`repo-scan-results`’ folder contains the complete results of the last run we performed.