Automated Techniques for Creation and Maintenance of TypeScript Declaration Files

Erik Krogh Kristensen

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Automated Techniques for Creation and Maintenance of TypeScript Declaration Files

A Dissertation
Presented to the Faculty of Science and Technology
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Erik Krogh Kristensen
August 8, 2019

Abstract

JavaScript was initially a scripting language for creating small interactive web pages. However, massive applications, both web pages and server applications, are increasingly being developed using JavaScript. The dynamic nature of the JavaScript language makes it hard to create proper tooling with features such as auto-completion and code-navigation. TypeScript is a JavaScript superset that adds, on top of JavaScript, an optional static type system, which facilitates features such as auto-completion, code navigation, and detection of type errors. However, many TypeScript applications still use untyped libraries written in JavaScript. Developers or users of these JavaScript libraries can choose to write TypeScript declaration files, which provide API models of the libraries and are used to type-check TypeScript applications. These declaration files are, however, written manually and often not by the original authors of the library, which leads to mistakes that can misguide TypeScript application developers and ultimately cause bugs.

The goal of this thesis is to design automated techniques for assisting in the development of TypeScript declaration files. This thesis identifies several challenges faced by developers of TypeScript declaration files and tackles these challenges using techniques from programming language research. Type inference is used to create new, and update existing, declaration files. Automated testing is used to detect errors in declaration files. Finally, data-flow analysis and a novel concept of reasonably most-general-clients are used to verify the absence of errors in declaration files. Each of the techniques is used to improve the quality of real-world declaration files.

Resumé

JavaScript startede som et scriptingsprog til at lave små interaktive hjemmesider. Dog udvikles større applikationer, både hjemmesider og andre applikationer, i stigende grad ved brug af JavaScript. JavaScript-sprogets dynamiske karakter gør det vanskeligt at skabe fornuftige udviklingsmiljøer med funktioner som automatisk kode-fuldendelse og kode-navigation. TypeScript er et superset af JavaScript, der udover JavaScript tilføjer et valgfrit system af statiske typer, der muliggør funktioner som automatisk kode-fuldendelse, kode-navigation og detektion af typefejl. Mange TypeScript-applikationer bruger dog stadig ikke-typede biblioteker skrevet i JavaScript. Udviklere eller brugere af disse JavaScript-biblioteker kan vælge at skrive TypeScript-deklarationsfiler, der leverer API-modeller af bibliotekerne og bruges til at kontrollere typerne i TypeScript-applikationer. Disse deklarationsfiler er dog skrevet manuelt og ofte ikke af de originale forfattere af biblioteket, hvilket kan medføre fejltagelser, der kan vildlede TypeScript-applikationsudviklere og i sidste ende forårsage fejl.

Målet med denne afhandling er at designe automatiserede teknikker til at assistere i udviklingen af TypeScript-deklarationsfiler. Denne afhandling identificerer flere udfordringer, som udviklere af TypeScript-deklarationsfiler står overfor, og håndterer disse udfordringer ved hjælp af teknikker fra programmeringssprogforskning. Type inferens bruges til at oprette nye og opdatere eksisterende deklarationsfiler. Automatisk testning bruges til at opdage fejl i deklarationsfiler. Endeligt bruges datastrømningsanalyse og et nyt koncept af rimelige-mest-generelle klienter til at verificere fraværet af fejl i deklarationsfiler. Hver af teknikkerne bruges til at forbedre kvaliteten af faktisk benyttede deklarationsfiler.

Acknowledgments

I would like to thank the following people for their support and guidance during my Ph.D. studies.

Anders Møller for being an attentive and hard-working supervisor, for helping me navigate the world of academia, for letting me run with my ideas, and for significantly improving the quality of my writing. Esben Andreasen for all the excellent office discussions we had. Gianluca Mezzetti for being a coding guru without whom this project would have taken longer and been worse. Søren Eller Thomsen for dragging me to the coffee machine for a refill and otherwise interrupting my work. The rest of the Programming Languages research group and various people from Aarhus University for providing an excellent work and study environment with room for learning, discussions, and beer. Eric Bodden and his research group for the great time I had while visiting the Heinz Nixdorf Institut. Finally, my family for supporting me despite not being sure what I really do.

Erik Krogh Kristensen,
Aarhus, August 8, 2019.

Contents

Part I

Overview

Chapter 1

Introduction

Optionally typed languages, which mix static and dynamic typing, are increasingly being used by developers, with languages such as TypeScript, Python, and Groovy seeing an increase in contributors within a year on GitHub of 90%, 50%, and 40% respectively [4]. Many developers prefer to use TypeScript or Python with both being in the top 3 most loved languages according to Stack Overflow [7].

These optionally typed languages allow programmers to mix dynamically typed code, where type checking is deferred until runtime, with statically typed code, which can be type-checked before the program is executed. Some of the patterns used in dynamic programming languages can be very hard to express in statically typed languages [30], however, the tool support for these dynamic languages often lacks compared to statically typed languages. By mixing dynamic and static types, an optionally typed language gives programmers the option of using the programming style that fits the best when solving a problem.

Developers writing typed applications in optionally typed languages often want to use existing untyped libraries, and in languages such as TypeScript or Python typed API models describing the behavior of the untyped libraries are often available. These API models facilitate type checking of typed applications that use the untyped libraries, and the API models enable precise auto-completion and code-navigation in IDEs. However, the API models are ignored at runtime, where the untyped library is executed without any regard for the types in the API models. For TypeScript and Python, these API models are called declaration files and stub files, respectively. Both languages have official repositories containing API models for a large collection of libraries.[1]

[1]For Python this collection is called typeshed [15], and for TypeScript the collection is called DefinitelyTyped [14].

Some optionally typed languages are also gradually typed. A gradually typed language adds, on top of being optionally typed, type checks at the boundaries between statically and dynamically typed code [102, 103]. These type checks detect at runtime whether the value of a variable matches the type annotation for the variable. Programmers can, due to the type checks, trust type annotations on variables to be correct at runtime, and that no type errors will happen inside fully annotated code where dynamic types are not used. Most optionally typed languages, including all previously mentioned, are not gradually typed, as these languages do not insert type checks on *all* boundaries between statically and dynamically typed code. What sets a gradually typed language apart from languages such as Groovy and Dart, which has type checks at variable assignments, is that a gradually typed language can track blame on higher-order functions, such that for example assigning an untyped function `F` to a typed variable `X` will result in the arguments and return value of the function `F` being type-checked against the argument and return types of `X` at runtime when `X` is invoked.

As neither TypeScript nor Python is gradually typed, mistakes in untyped code can lead to runtime type error happening inside fully type annotated code. For example, mistakes in the API models for untyped libraries can lead to runtime type errors far from where the library was used in an otherwise fully typed application. A recent study found that 15% of errors in JavaScript projects sampled on GitHub could have been fixed by using a type system for JavaScript [51], which suggests that type errors are common and can cause real bugs in programs. Additionally, errors in the API models can misguide developers using IDE features such as auto-completion, thereby causing a developer to for example put a typo in their program by following the suggestions given by the IDE. It is therefore important for the types in the API models be to correct. However, there are no automated checks that an API model accurately describes the behavior of the untyped library implementation, and the API models are often not written by the original authors of the implementations. These factors result in the API models often containing mistakes.

Programming language research has throughout the years developed many techniques, such as type systems and dynamic analysis, for automatically detecting various kinds of errors in many different programming languages.

Type systems have been used with success to guarantee the absence of some type errors [41, 48]. Type systems generally overapproximate the possible behaviors of a program and reject a program if this overapproximation could exhibit a type error. However, any decidable type system which rejects all programs containing some type error will necessarily reject pro-

grams that never cause such a type error to happen when executed [104]. Type systems have been used to detect many kinds of type errors. One of the most straightforward type errors is when numerical operations are performed on non-numerical values. However, type systems can also reason about more complex errors. For example, a type system that includes information about the nullability of values can be used to prevent null pointer dereferences [45, 78], or a type system might guarantee that no secret information leaks from a program [113].

Type systems often require the programmers to add annotations to their programs. However, work on type inference tries to reduce the burden on the programmer to write annotations [50, 55, 80]. In some cases type inference allows a programmer to omit all type annotations and still write fully typed applications.

Static analysis is used to create an approximation of the potential behaviors of a program by analyzing the program without executing it [37, 61, 66]. Such an approximation can be used for many purposes, some of which include improving the performance of compiled programs [24, 66], detecting security issues or malicious programs [23, 34], and detecting potential type errors in programs without any type annotations [57].

The big challenge when creating a static analysis is achieving a good trade-off between performance and precision of the analysis. For some languages it is possible to create a useable, fast, and very imprecise analysis to decide some properties of interest. An example of such a fast imprecise analysis is unification based points-to analysis for C [106] or class hierarchy analysis on object-oriented programs [40]. Similar static analyses would in practice yield unusable results for a dynamic language like JavaScript, where the frequent use of reflection and other dynamic features requires a highly precise analysis in order to achieve nontrivial results [20, 63].

Data-flow analysis is often used to achieve the level of precision required for analyzing programs written in these highly dynamic languages [57, 63]. Data-flow analysis overapproximates a program by modeling the program state as a lattice [27] and using monotone functions operating on the state to model each statement in the program. These monotone functions can then be iteratively applied until until a fixpoint is reached. Using a data-flow analysis can result in a highly precise analysis. However, the performance of a data-flow analysis can vary widely depending on the complexity of the analyzed program, and sometimes the performance of a data-flow analysis degrades to the point where the analysis times out or runs out of memory.

Static analysis is most often sound; that is, the static analysis will always find a superset of all the possible errors in a program. However, sometimes soundness can be sacrificed in a static analysis to achieve better per-

formance, and still achieve a useable result [46, 67]. Even the static analyzers that claim to be sound are often only mostly sound [75].

Dynamic analysis is, as opposed to static analysis, an analysis method where the program being analyzed is executed concretely. A dynamic analysis is often made by modifying the program being analyzed, either by instrumenting the program [36, 100] or by modifying the runtime that executes the program [107]. A dynamic analysis can be used to find many of the same problems and errors as a static analysis, however, since a dynamic analysis is based on one or more concrete executions, which can inherently only underapproximate the possible behaviors of a program, any error found by a dynamic analysis is unlikely to be a false positive. In contrast, a static analysis is often plagued by false positives. However, sound static analysis finds all the possible errors in a program, which a dynamic analysis generally cannot guarantee to do.

Random testing is a dynamic analysis technique which is often used to find errors in programs. In random testing, the analysis creates some random input for the program being tested, after which the program is executed using this randomly generated input [35]. Fuzzing is a technique where dynamic analysis takes some existing known input and tries to modify the known input in an attempt to expose bad behavior from the program being tested [52]. The input modified by a fuzzer can even be other programs for testing compilers [56, 69]. Another variant of random testing is feedback directed random testing, where values obtained by the analysis from calling methods on the program under test is used when calling other methods, and thus the values produced by the program under test are fed back to the program itself [68, 82].

Static and dynamic analysis can be combined into a single hybrid analysis [110, 116]. These hybrid analyses try to combine the benefits of both static and dynamic analysis by for example using the information found by a dynamic analysis as the starting point for a static analysis [47, 53, 67].

Combining a static and dynamic analysis often results in an analysis, which creates an overapproximation of an underapproximation. Thereby an analysis is created which both reports false positives and misses real problems in a program. However, as research into developers usage of static analysis tools has found, having too many false positives may cause developers to ignore the output of a tool [96]. Therefore a middle-ground between dynamic and static analysis can sometimes have the right balance of finding most problems in some program without being plagued by false positives.

It is sometimes possible to get the benefits of both static and dynamic analysis without the drawbacks. For example, work has been done combining static and dynamic analysis in such a way that the static analysis

can be used to prove that the dynamic analysis has explored all possible executions [16].

These analysis methods in programing language research can be applied to the problem of API models of untyped libraries being erroneous. This thesis focuses on how automated techniques in programing language research can be applied to improve the quality of TypeScript declaration files. This thesis explores the use of type inference to infer new declaration files and evolve existing declaration files, dynamic analysis to automatically detect type errors in declaration files, and data-flow analysis to verify the absence of type errors in declaration files.

As the use of optionally typed languages is increasing,[2] this thesis will hopefully help not only with improving the quality of TypeScript declaration files today, but also inspire new tools for the optionally typed programming languages of tomorrow.

1.1 Challenges

We see the following challenges that developers face when developing TypeScript declaration files:

- **Creation of new declaration files**: As new declaration files for existing JavaScript libraries are often created, and given that new libraries written in untyped JavaScript are still emerging, the need to create new TypeScript declaration files will continue to exist.

- **Updating existing declaration files**: TypeScript declaration files need to follow the development of the corresponding JavaScript implementations and manually going through code and documentation changes can be difficult, especially if the implementation has been refactored.

- **Detecting errors in declaration files**: The TypeScript compiler does nothing to check if the types in the TypeScript declaration files match the untyped JavaScript implementation, and errors are therefore common in TypeScript declaration files.

- **Verifying the absence of errors**: Verifying that no further errors can be detected in a TypeScript declaration can give strict guarantees to the users and authors of a TypeScript declaration file.

[2] Both the Stack Overflow Developer Survey [7] and GitHub's The State of the Octoverse [4] found that TypeScript and Python have increased adoption and are highly loved among developers.

1.2 Thesis statement

The main thesis statement is:

> Automated program analysis techniques can be created for assisting in the development and maintenance of TypeScript declaration files.

In other words, the goal of this thesis is to explore automated techniques in programming languages research, such as type inference, automated testing, and static analysis, and find ways in which these techniques can be used to improve the development experience of developing TypeScript declaration files. Even though much work has already been done within each of these programming language techniques, applying them to the challenges described above is far from straight forward, as TypeScript's type system is complex, continuously evolving, and containing deliberately unsound features, and JavaScript has been a notoriously difficult analysis target.

1.3 Contributions

This thesis makes the following contributions:

- We develop an analysis which can infer types in a JavaScript library using a combination of a dynamically obtained heap snapshot and a static type inference based on subsets, and we use this analysis to create TypeScript declaration files automatically. We show that these automatically inferred TypeScript declaration files can be used as a starting point when authoring a declaration file for a library.
- We utilize our TypeScript type inference in a novel way to create the first approach for assisting in updating TypeScript declaration files when the corresponding JavaScript implementation has been updated.
- We develop an approach for using feedback-directed random testing on JavaScript libraries using the specification from a TypeScript declaration file, and we use this feedback-directed testing for automatically detecting mismatches between the TypeScript declaration file and the corresponding JavaScript implementation.
- We develop the concept of a reasonably-most-general client (RMGC) based on the idea of a most-general client (MGC). An RMGC and an MGC can be used for modeling all possible clients for a library. However, the RMGC is limited by a set of assumptions. We use the RMGC

idea to fix declaration files, such that we can guarantee the absence of errors in the fixed declaration files under certain assumptions.

1.4 Outline

This thesis first contains an overview of the research area (Part I). Secondly, this thesis contains the associated published research papers (Part II).

Part I starts with this introduction. Next in Part I is an introduction to the JavaScript and TypeScript programming languages. This introduction is not meant as a tutorial to the languages; rather, the introduction is meant to give the reader an understanding of the challenges in developing and analyzing JavaScript and TypeScript programs. Last in Part I is an introduction to the automated program analysis techniques used in the associated research papers. The introduction to automated program analysis techniques starts with two static analysis techniques: type analysis and data-flow analysis, and ends with an introduction to two dynamic analysis techniques: automated testing and snapshot assisted analysis.

1.4.1 Published Papers

A part of the author's work was co-authoring the following papers, which are included in Part II. The papers are unmodified except for layout changes to accommodate the format of this thesis, and two of the papers contain a section new to this thesis. These new sections add contents that were never part of the original paper due to a page limit.

- *Inference and Evolution of TypeScript Declaration Files*.
  Erik Krogh Kristensen and Anders Møller. Published in Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering, FASE 2017. Included as Chapter 5.
  Section 5.6 is new to this thesis and contains a detailed explanation of the implementation.

- *Type Test Scripts for TypeScript Testing*.
  Erik Krogh Kristensen and Anders Møller. Published in Proceedings of the ACM on Programming Languages Volume 1 Issue OOPSLA, October 2017. Included as Chapter 6.

- *Reasonably-Most-General Clients for JavaScript Library Analysis*
  Erik Krogh Kristensen and Anders Møller. Published in Proceedings of the 41th International Conference on Software Engineering, ICSE 2019. Included as Chapter 7.

Section 7.8 is new to this thesis and contains an evaluation on a larger set of benchmarks.

The author of this thesis has been a major contributor in all phases of the research projects that culminated in the three research papers listed above. The author of this thesis is the primary developer of all the implementations that were part of the research projects.

Chapter 2

JavaScript and TypeScript

JavaScript was initially created as a scripting language for the Netscape browser [5]. JavaScript was long the only language used for web development, which ensured an increasing usage of JavaScript. When Google released the JavaScript JIT compiler V8 in 2008 the performance of JavaScript increased dramatically, which further increased the adoption of JavaScript and eventually lead to developers using JavaScript in other environments than web browsers.

Today JavaScript is defined by the ECMAScript standard [1]. Throughout this thesis, the names JavaScript and ECMAScript will be used interchangeably to refer to the language described in the ECMAScript standard.

Many languages have attempted to replace JavaScript by using JavaScript as their compilation target. The most successful of the attempted replacements is TypeScript, which was introduced by Microsoft in 2012. TypeScript is designed as a superset of JavaScript, meaning that any valid JavaScript program is a valid TypeScript program. Therefore in order to understand TypeScript, one must first understand JavaScript.

2.1 JavaScript

JavaScript is a dynamically typed language with C inspired syntax, first-class functions, and prototype-based inheritance.

JavaScript is a language under constant development, and the language is continuously growing in complexity. The number of pages in the ECMAScript standard demonstrates this growth: The third edition from 1999 contained 188 pages [2], whereas the printable version of the latest 9th edition from 2018 contains 805 pages [3]. The ECMAScript standard has in later editions both added numerous methods to the standard library and new syntactic constructs to the language syntax. Some of the newer features

of JavaScript include modules, lambdas, classes, and generators. Developers who want to use the syntax from the later editions of JavaScript often compile their code to an earlier edition of JavaScript using a JavaScript-to-JavaScript compiler such as Babel [12]. Similarly, developers who want to use standard library methods from later versions of the ECMAScript standard can add polyfills [13] that mimic the functionality from later editions of the language.

JavaScript is an extensive language containing both good and bad parts [39]. Using the good parts of the language can help a developer create easy to understand programs. However, not all programmers do so, and it is, therefore, necessary to understand the bad parts of JavaScript in order to understand why JavaScript can be challenging to understand for programmers. Some of these difficult to understand features also make JavaScript a problematic analysis target.

This chapter is not a tutorial that will allow the reader to create JavaScript programs. The purpose of this chapter is to establish a foundation for a reader to understand the challenges that developers face when writing JavaScript programs and analyzers for JavaScript programs. This foundation is relevant for understanding the TypeScript language (Section 2.2) and the challenges tackled in the papers in Part II of this thesis.

2.1.1 Types

JavaScript is dynamically typed, and the language, therefore, allows all values to be assigned to any variable even if the variable has previously been assigned an existing value of a different type. The dynamic type system results in many valid JavaScript expressions that would be rejected by most type checkers for statically typed languages. An example of such an expression is `4()`, which attempts to invoke the number 4 as a function. However, the expression `4()` will always throw an exception when executed.

The dynamic types make static analysis of JavaScript trickier than static analysis of statically typed languages. In a statically typed language, a static analysis can assume the type of the expression at any program point that has a type annotation. These assumptions can both be used to filter values during static analysis, and to create conservative overapproximations by only observing the annotated types of variables [40]. A JavaScript static analysis can, if it loses precision, come in a situation where it overapproximates the value of an expression as potentially being of many different types of objects, functions, and primitives. When such a precision loss happens, some reads and writes on objects are overapproximated as happening on many different kinds of objects, which can cause imprecision to spread throughout the analysis quickly creating a massive precision loss.

JavaScript heavily employs type coercion, where values are automatically converted from one type to another. These type coercions allow great flexibility, as developers can often trust the language to do what the developers meant to do. Consider as an example the following program.

```
1 var input = prompt("Feed me number");
2 if (input > 42) {
3     console.log(input + " is greater than 42.");
4 }
```

The program first asks the user to input a number by using the `prompt` function. Afterwards the number is stored in the variable `input` (line 1). The program next outputs to the user whether the variable `input` is greater than `42` (line 2–4). In the above program the `input` variable will always have the type `string`, however, type coercion will automatically convert this `string` to a `number` on line 2, and the comparison on line 2 will therefore compare the number represented by the string in `input` with 42.

Type coercions can, however, make the behavior of a program hard to understand. As an example, both the expressions `"50"<"6"` and `"50">6` evaluate to `true`. The JavaScript equality operator == uses type coercions, and this leads to the == operator not having the usual transitivity property of an equality operator as both `""==0` and `"0"==0` evaluate to `true`, but `""==0` evaluates to `false`. An equality check without type coercion can be performed by using the === operator. For example the expression `"0"===0` evaluates to `false`.

These type coercions and the resulting behavior of the equality operator are some of the features of JavaScript that the creator of JavaScript wanted to fix [8]. However, changing these features would break a lot of existing code, and JavaScript is hence stuck with type coercions.

2.1.2 Objects

Objects in JavaScript are essentially maps from strings[1] to values. Reading an absent property from an object returns the special value `undefined`, and writing to an absent property results in the object changing shape by adding the new property to the object. In JavaScript, it is also possible to read or write a property based on a computed string as exemplified by the following program:

[1]Symbols can also be used as keys in objects in the newer editions of JavaScript. However, for this presentation these symbols are ignored.

```
5 var x = {}; // initializes a new empty object
6 x.foo = false; // assigns to the foo property
7 x["fo" + "o"] = 2 // assigns to the foo property again
8 console.log(x.foo); // prints 2
```

Any computation can be used to calculate which property should be read or written to, and such computations are sometimes used to create dynamic code where, for example, all the properties of one object are copied to another object.

These dynamic property writes are a central reason why static analysis of JavaScript can be complicated, as properties on objects are defined and afterward read based on arbitrarily complex computations. If for example a static analysis is unsure which property of an object is being written to, then the analysis might conclude that all the properties of the object could be overwritten, which will cause future property reads on the object to be modeled imprecisely.

The same complexity is also the reason for IDE's rarely having useful autocompletion for JavaScript, as it can sometimes be impossible to determine which properties could exist on an object.

2.1.3 Inheritance

As mentioned in the previous section, reads and writes on objects in JavaScript can be complicated. However, JavaScript objects can be part of an inheritance hierarchy through prototype chains, which further complicates property reads.

Objects in JavaScript can have a link to a prototype, which is another object. If a property read happens on an object and the requested property is absent from the object, then the property read is recursively attempted on the prototype of the object, and prototype links are thereby used to create an inheritance hierarchy between objects. The prototype of an object can change during the lifetime of the object. The below code shows how the prototype of an object can be changed and how this change affects a property read on the object.

```
9 var foo = {a: 2};
10 var bar = {b: 3};
11 console.log(foo.b); // prints undefined
12 // set prototype of foo to bar
13 Object.setPrototypeOf(foo, bar);
14 console.log(foo.b); // prints 3
```

In real JavaScript programs, several code patterns are used to instantiate the prototypes between objects, and it is therefore difficult for IDE's and static analyzers to reason about the inheritance of objects.

Later editions of JavaScript introduce a syntactical `class` construct, which can be used to create inheritance between classes as in Java. However, the class construct is just syntactic sugar; the same prototype links are still used to create the inheritance. Therefore it is still necessary to understand the prototype mechanism to understand how classes work.

2.1.4 Functions

Functions are first-class citizens in JavaScript, and they can therefore be assigned to variables, passed as arguments to functions, created inside other functions, and generally be treated as any other value. JavaScript is a multi-paradigm language, as both mostly object-oriented and mostly functional programs can be written using JavaScript.

JavaScript has no support for function overloading. Instead, programmers can write functions that detect the types of their arguments at runtime. The following code shows a simple example of such a function from the PathJS[2] library.

```
15 function enter(fns) {
16   if (fns instanceof Array) {
17     this.do_enter = this.do_enter.concat(fns);
18   } else {
19     this.do_enter.push(fns);
20   }
21   return this;
22 }
```

The `enter` function first detects whether or not the argument `fns` is an array (line 16). If `fns` is an array it will be concatenated onto the `this.do_enter` array (line 17), and otherwise the `fns` value will be pushed as the last element onto the `this.do_enter` array (line 19).

These functions detect at runtime what types they have been called with, and that makes it difficult for analyzers to reason about these functions, as the types of function arguments and the behavior of the functions can depend on how the functions are called.

[2] `https://github.com/mtrpcic/pathjs`

2.1.5 New Features of JavaScript

The newer editions of JavaScript have introduced various new syntactic constructs. Programmers can use these new constructs to create smaller and more readable programs. Among the new syntactic constructs are classes, lambdas, and block scoping. None of these new constructs allow for more expressiveness than what was possible without them; they merely provide an alternative way of expressing the same functionality. The following program shows an example of a JavaScript program that uses the class syntax, a lambda, and block scoping.

```
23 class Foo {
24   foo () {
25     setTimeout(() => this.bar());
26   }
27   bar() {
28     let i = 2;
29     {
30       let i = 3;
31     }
32     console.log(i); // prints 2
33   }
34 }
```

The class in the above code (line 23) can alternatively be written as a constructor function with added properties on its prototype, the lambda (line 25) can alternatively be written as a function, and the two block-scoped `i` variables can be written as two variables with different names.

A JavaScript-to-JavaScript compiler[3] has transformed the above code into the following code, which does not use any of the newer JavaScript features. The below code is 299 characters compared to the 152 characters in the code above. Although the two programs do the same thing, most programmers will find that the below code is harder to read.

```
35 var Foo = (function () {
36   function Foo() {}
37   Foo.prototype.foo = function () {
38     var _this = this;
39     setTimeout(function () { return _this.bar(); });
40   };
```

[3]The TypeScript compiler was used to transform the code: `https://www.typescriptlang.org/play/`

```
41   Foo.prototype.bar = function () {
42     var i = 2;
43     {
44       var i_1 = 3;
45     }
46     console.log(i); // prints 2
47   };
48   return Foo;
49 }());
```

2.1.6 JavaScript: The Bad Parts

JavaScript became a shipping product too soon according to its creator [8]. As a consequence of the fast release, JavaScript contains features that can make programs hard to read even for expert developers. Some of the features have since been fixed or are so rarely used that they can mostly be ignored. However, some of these features are used often, as they can provide the best way of implementing specific programming patterns. JavaScript programmers and developers of analysis tools therefore have to understand these challenging features.

Eval: The `eval` function in JavaScript takes a single string as an argument and executes this string as code. Many patterns that use `eval` can be rewritten into a more readable pattern where `eval` is not used [59, 93].

Misuse of the `eval` function can easily create security holes in applications, and using the `eval` function is therefore generally discouraged.[4]

For static analyzers, the `eval` function presents a significant challenge. If a static analysis is unsure exactly what string the `eval` function is called with, then a call to `eval` could potentially call all methods and have all possible side-effects in the program, which, if conservatively modeled by a static analysis, would destroy the precision of the analysis.

arguments: Consider the following function definition and corresponding function call:

```
50 function foo() {
51     ...
52 }
53 foo(4,2);
```

The function `foo` is called with two arguments (line 53), however, the function definition (line 50) takes no arguments. The above code will execute

[4] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval#Do_not_ever_use_eval!`

without throwing any exception, as the two arguments given to `foo` on line 53 are silently ignored. The implementation of `foo` can still access the given arguments using the `arguments` object even though the `foo` function is not declared as receiving any arguments. The `arguments` object always contain all the arguments used for calling a function.

This language feature was not well thought through by the designer of JavaScript: *"JavaScript has this hideous arguments object that I did in a tearing hurry" - Brendan Eich* [8].

As an example of how the `arguments` object can be used, the following code uses the `arguments` object to implement a sum function by iterating through all the arguments and adding them to `0`.

```
54 function sum() {
55   var total = 0;
56   for (var i = 0; i < arguments.length; i++) {
57     total += arguments[i];
58   }
59   return total;
60 }
61 console.log(sum(1,2,3,4,5)); // prints 15
```

The same sum function can also be implemented more elegantly using rest parameters.[5] Another common use of the `arguments` object is to implement default parameters manually.[6] Syntactic support for default parameters and rest parameters was added in ECMAScript 6. Using rest parameters or default parameters result in code that is generally more readable than using the `arguments` object. However, not all JavaScript runtimes support these newer features, and developers who want to support these older JavaScript runtimes can therefore not use these newer features.

Using the `arguments` object can be a source of bugs. As part of the research papers presented later in this thesis, several mismatches between TypeScript declaration files and JavaScript implementations for libraries were detected and fixed. One of those mismatches was a bug in the implementation of the massively used async library where a faulty comparison on the `length` property of the `arguments` object caused a mismatch between the documentation and the implementation of the library.[7]

[5] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters`

[6] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Default_parameters`

[7] `https://github.com/caolan/async/pull/1381`

2.2 TypeScript

TypeScript is a superset of JavaScript, and all syntactically valid JavaScript programs are therefore also valid TypeScript programs. TypeScript notably adds, on top of JavaScript, optional type annotations. Previously, TypeScript had multiple new syntactic constructs on top of JavaScript, such as: lambdas, classes, and block scoping. The developers of TypeScript have always attempted to develop these additional syntactic constructs such that they are compatible with future editions of JavaScript. JavaScript has, since the inception of TypeScript, caught up with most of these syntactic constructs such that the only distinguishing feature of TypeScript is its type system. TypeScript does, however, still allow for compiling these constructs to earlier editions of JavaScript, thereby letting developers use the latest JavaScript features on runtimes that do not yet support them.

Unlike JavaScript, TypeScript does not have a rigorous specification. There is a specification [79], however, the language specified in the specification is version 1.8 of the TypeScript language, whereas the latest version of the implementation is 3.5. The TypeScript language is therefore in reality specified by a series of blog posts, the open-source TypeScript compiler, and a documentation page [10]. Attempts have been made to formalize the TypeScript type system [26]. However, as TypeScript has been rapidly evolving, with a new release roughly every two months, such attempts are quickly outdated.

The purpose of this section is to allow the reader to understand the complexity of the TypeScript type system, both to understand the complexity of creating an analysis for TypeScript, and to have a foundation for understanding the kinds of errors authors of TypeScript declaration files make, as these are the kinds of errors two of the papers in Part II are attempting to detect.

2.2.1 Types in TypeScript

The type system in TypeScript has been designed such that JavaScript developers can continue to use common patterns from JavaScript. As a consequence of this design choice, the type system in TypeScript is unsound by design [26], and the type system has become quite complicated as it has been challenging to design types that can describe some of the common patterns in JavaScript. The type system has grown considerably since its introduction in 2012. The type system is exclusively used to provide developer feedback, and types will therefore not affect the code produced by the compiler or block the compiler from producing an output program.

The following sections will present an incomplete overview of the Type-

Script type system. The overview should give the reader an idea of the complexity of the TypeScript type system, as well as present a few corner cases.

2.2.2 Annotations and Inference

The following program illustrates an error caught by the TypeScript type system using only primitive types.

```
62 var foo : string = "horse";
63 var bar : number = 2 + foo;
```

Line 62 declares the variable `foo` as a **`string`** with the value `"horse"`, and in the next line the variable `foo` is added to the constant `2` to calculate the value assigned to the variable `bar`. However, `bar` is declared as being a **`number`** and the result of the expression is the string `"2horse"`. The TypeScript compiler correctly identifies and reports this type mismatch when compiling the above program.

TypeScript utilizes type inference which allow programmers to omit type annotations. For example in the following program the `foo` variable is inferred as having the type **`string`** without an explicit annotation, and the TypeScript compiler will therefore produce the same error when compiling the following program as it does when compiling the above program.

```
64 var foo = "horse";
65 var bar : number = 2 + foo;
```

The type inference in TypeScript is quite simple, and the TypeScript compiler will implicitly annotate a variable with the **`any`** type when there is no explicit type annotation and no other type can be inferred. These variables with an implicit **`any`** can create holes in the type checking, as no type error will be reported on those variables. To prevent these holes from occurring, the TypeScript compiler has an optional flag, which when enabled will cause a type error at all locations where an implicit **`any`** is inserted. Flow [33], another JavaScript alternative similar to TypeScript, has more advanced type inference compared to TypeScript. However, TypeScript is more widely used than Flow, which is why this thesis will focus entirely on TypeScript.

2.2.3 Primitives

The primitive types that will be covered during this introduction to TypeScript are **`number`**, **`boolean`**, **`string`**, **`void`**, and **`any`**.[8] The first 3 should be

[8]The following have been omitted for simplicity: `undefined`, `null`, `object`, `symbol`, `never`, and `unknown`.

self explanatory. The `void` type is used to declare when function returns the special `undefined` value or `null`. The `any` type is a special type that can represent any value. All types are subtypes of the `any` type, and `any` is assignable to and from all other types. If a variable is declared as having the type `any`, then type checking will effectively be disabled for that variable. Consider as an example the following program for which the TypeScript compiler will report no type warnings.

```
66 var foo : any = "horse";
67 var bar : number = 2 + foo;
```

Additionally, all JavaScript constants are TypeScript types. Constant types in TypeScript represent an exact value, as demonstrated by the following program.

```
68 var myNum : 2 = 3; // error
69 var myStr : "foo" = "foo";
70 var str2 : string = myStr;
```

The TypeScript compiler will produce a type error on line 68 as the number `3` does not satisfy the type `2`. However, no type error will be given for the other lines in the above program, as the type `"foo"` is a subtype of `string`.

2.2.4 Objects

Object types in TypeScript use structural typing, where the structure of objects types define whether a type is a subtype of another type, instead of nominal typing, where an explicitly defined type hierarchy defines the subtyping relations. Therefore the following program, which would result in type errors for an equivalent nominally typed program, compiles without warnings.

```
71 interface Foo {
72     name: string;
73 }
74 interface Bar {
75     name: string;
76     age: number;
77 }
78 var p: Bar = { name: "V", age: 5 }
79 var other: Foo = p;
```

The above program first declares a object type `Foo` containing a single property `name` (line 71–73), next a type `Bar` is declared containing a similar `name` property and additionally an `age` property. The program then assigns a new object to the variable `p` with the declared type `Bar` (line 78). This

assignment produces no type error as the object contains the `name` and `age` property as declared by the type `Bar`. Finally the program assigns the variable `p` to the variable `other` declared as having the type `Foo` (line 79). This last assignment produces no type error as TypeScript uses width subtyping [79], which is when properties can be added to an object to create a subtype, and the `Bar` type is, therefore, a subtype of `Foo`.

2.2.5 Functions

Most function definitions in TypeScript are simple. The below function definition is an example of a function that checks if a string is longer than a given length.

```
80 function isLong(str: string, len: number) : boolean {
81     return str.length > len;
82 }
```

The above function contains type annotations both for the argument types (`string` and `number`) and for the return type (`boolean`). However, as with variable declarations, all of those type annotations can be omitted. If the annotation for the return type is omitted the TypeScript compiler can usually infer the return type. However, if the type annotations for the argument types are omitted the TypeScript compiler will assign the type `any`.

2.2.6 Functions: Covariance and Contravariance

Readers familiar with subtyping for functions might be familiar with the terms covariance and contravariance. In TypeScript return types are covariant, therefore for a function `A` to be a subtype of another function `B`, the return type of `A` needs to be a subtype of the return type of `B`. Parameter types in TypeScript are by default bivariant, therefore for a function `A` to be a subtype of a function `B`, the parameters of one of the functions have to be subtypes of the other. However, the direction of the subtyping on the parameters does not matter. The choice of bivariant parameters types makes the type system unsound, and a fully annotated programs where the TypeScript compiler detects no type errors can therefore result in type errors at runtime caused by the use of bivariant parameters types. Bivariant parameter types have, however, been chosen in TypeScript because some patterns used by JavaScript developers require bivariant parameter types.[9]

Parameter types can be type-checked contravariantly using a flag on the TypeScript compiler. Type-checking parameter types contravariantly is, un-

[9] https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-6.html#strict-function-types

like the default bivariant type-checking, sound.[10] With contravariant parameter types a function `A` is a subtype of a function `B` if the parameters of `B` are subtypes of the parameters of `A`. Notice that covariant and contravariant are duals of each other, with the direction of the subtyping being flipped between the two.

2.2.7 Function Overloading

In TypeScript, as in JavaScript, functions can behave differently depending on how they are called. Consider the below `add` function written in TypeScript.

```
83 function add(a: number, b: number): number;
84 function add(a: string, b: string): string;
85 function add(a: any, b: any): any {
86     return a + b;
87 }
```

The first two lines (line 83–84) declare the function signatures for the function `add`, and the last lines (line 85–87) implement a function satisfying the declared function signatures. Importantly, the signature on line 85 is only used to implement the two declared method overloads, and this last signature cannot be used when calling the `add` function.

When type-checking a call site of an overloaded function, the TypeScript compiler needs to decide which overload is used. The compiler type checks if the arguments given in the function call match the arguments in the signature for each overload in textual order, and the compiler stops at the first matching overload. This first match policy can come as a surprise to developers. Consider the below function overloads (without implementation) taken from a real library.[11]

```
88 function make_color(options?: MakeColorOption): Array<string>;
89 function make_color(options?: MakeColorOption): Array<RGB>;
90 function make_color(options?: MakeColorOption): Array<HSV>;
```

If a program calls the `make_color` function the TypeScript compiler will use the first overload matching the given arguments at the function call, and

[10] This does not mean the entire type system becomes sound, there are other cases of unsoundness in TypeScript.

[11] `https://github.com/DefinitelyTyped/DefinitelyTyped/pull/32821/files#diff-8adcf7294333a71b62751ef7ce655d85L15`

as all three overloads have the same argument types, the TypeScript compiler will always use the first overload, and the second and third overloads are therefore useless.

2.2.8 Functions That Return `void`

The TypeScript type system has some non-obvious corner cases, which can cause confusion when used. Here we will detail one such example, which happens when the type `void` is used as a function return type. Consider the below declarations of the functions `foo` and `bar`. The TypeScript compiler reports a type error for the `foo` function but not the `bar` function.

```
91 var foo = function (): void {
92     return "foobar"; // error
93 }
94 var bar: () => void = function () : string {
95     return "foobar"; // no error
96 }
```

The two functions do exactly the same and have the same types. The only difference is that `foo` is directly assigned to a variable which is typed as a function with a `void` return. The function on line 94 is first declared as a function that returns a `string` and then afterwards assigned to the variable `bar` with return type `void`. The `foo` function gives rise to a type error as the string `"foobar"` (line 92) is not a subtype of `void` type. However, there is no type error related to the `bar` function, which is caused from how function subtyping works in TypeScript. The TypeScript subtyping rule specifies that for a function `A` to be a subtype of a function `B`, the return type of `B` either has to be subtype of the return type of `A`, or if the return type of `A` is `void` then the return type of `B` can be any type.[12]

2.2.9 This Types

In JavaScript and TypeScript it is common to use method chaining, where multiple method calls are chained together by having each method call return the object the method was called on. An example of such a chain can be seen in the following program on line 113–117. A method declared as returning `this` in TypeScript will always return the object the method was invoked on, and thereby chaining methods are supported even when the methods in the chain are declared across multiple classes. The following program serves as motivation for the `this` type.

[12] https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md#3113-subtypes-and-supertypes

```
97  class AddMachine {
98      public constructor(public value: number = 0) { }
99      public add(n: number): this {
100         this.value += n;
101         return this;
102     }
103 }
104
105 class MultMachine extends AddMachine {
106     public multiply(n: number): this {
107         this.value *= n;
108         return this;
109     }
110     // ... other operations go here ...
111 }
112
113 var v = new MultMachine(2)
114         .add(1)
115         .multiply(3)
116         .add(5)
117         .value;
```

The program declares a class `AddMachine` and a class `MultMachine`. Both of these classes contain a method for performing some calculation, and both of these methods are declared as returning `this` (line 99 and line 106). These two classes are then used on line 113 to line 117. If the method on line 99 had been declared as returning `AddMachine` instead of `this`, then the method call on line 115 would produce a type error, as the method call on line 114 would not return a `MultMachine` with a `multiply` method.

2.2.10 Union and Intersection Types

Union types in TypeScript are simple: if a variable `foo` is declared as having type `A|B`, then the variable `foo` must statisfy the type `A` or the type `B`. For example if a variable `foo` is declared as having the type `string|number`, then the value of `foo` must be either a `string` or a `number`.

Intersection types are the dual of union types. If a variable `foo` is declared as having type `A&B`, then the variable `foo` must statisfy both of the types `A` and `B`. Intersection types can easily be used to create impossible types, where no value can be of the type, such as the type `string&number`. Intersection types are mostly used to create object types where intersection

types can be used to add properties to an existing type. Below is a simplified example from the Underscore.js library[13] illustrating the use of intersection types.

```
118 interface Cancelable {
119     cancel(): void;
120 }
121
122 function throttle<T extends Function>(fn: T, wait: number):
123     T & Cancelable; {....}
```

The function declared on line 122 takes as argument a function `fn` of type `T`, and it returns a function that acts as a proxy to the input function. However, if the returned function is called more often than declared by the `wait` parameter (line 122), then the input function `fn` will only be called once every `wait` milliseconds.

The function returns a function of the same type as the input function (line 123), however, the returned function additionally has a `cancel` method as the return value is both of type `T` and the type `Cancelable`. This `cancel` method cancels an otherwise queued call to the input function `fn`.

2.2.11 Non-nullable Types

TypeScript has a non-nullable types feature that can be enabled using a compiler flag.[14] The following function gives an example where non-nullable types can be used to catch an error.

```
124 function foo(): number {
125     if (Math.random() > 0.5) {
126         return 2;
127     }
128     return null;
129 }
```

The function `foo` returns either the number `2` or `null` randomly. With TypeScript's default options, `null` and `undefined` are subtypes of all other types. A function declared as returning `number` can therefore return `null`. However, when the non-nullable types option is enabled, `null` and `undefined` are no longer subtypes of all other types. Therefore the above function `foo` only gives rise to a type error on line 128 when the non-nullable types

[13] https://github.com/DefinitelyTyped/DefinitelyTyped/blob/master/types/underscore/index.d.ts

[14] https://devblogs.microsoft.com/typescript/announcing-typescript-2-0/#non-nullable-types

feature is enabled. The correct return type for the above function would be `number|null` instead of `number`. The non-nullable types feature is not always used, as it burdens the programmers with extra annotations in all locations where `null` or `undefined` are possible, and enabling non-nullable types can result in many warnings inside a program that is working just fine. However, non-nullable types can be used to prevent real errors in programs [9], and it can therefore be a good idea to enable the feature.

2.2.12 Conditional Types

The following three subsections present a few interesting and rarely used features of the TypeScript type system. Readers that are only interested in understanding TypeScript to the level required to understand the remainder of this thesis can skip to Section 2.3.

Conditional types are used to create a type that depends on some condition on another type. The following code shows an example of using conditional types.

```
130 type Flatten<T> = T extends Array<infer E> ? E : T;
131
132 declare function flat<T>(t: T): Flatten<T>
```

On line 130 the conditional type `Flatten` is created. Using the <> brackets the `Flatten` type is declared as taking an unconstrained input type `T` as input. The `infer` keyword works like an exists operator, and the `extends` operator is used to create the query that determines which branch of the conditional type should be taken. The type of `Flatten<T>` can be described as: given a type `T`, if there exists a type `E` such that `T` is a subtype of `Array<E>` then the resulting type is `E`, otherwise the resulting type is `T`.

Line 132 declares a function using the `Flatten` type in the functions return type, thereby the declared function returns either an element from an array if the input is an array and otherwise the input itself is returned.

With conditional types, it is sometimes not possible to create an implementation satisfying the declared type without using type casts. The following is an attempt at implementing the declared `flat` function from the above code.

```
133 function flat<T>(t: T): Flatten<T> {
134     if (t instanceof Array) {
135         return t[0];
136     } else {
137         return t;
```

```
138     }
139 }
```

The implementation of `flat` implements the two cases from the type `Flatten`, and the TypeScript compiler is correctly able to infer that the first return on line 135 returns an element from the array, as required by the return type `Flatten<T>`. However, the TypeScript compiler reports a type error on line 137, as the TypeScript still thinks that the input `t` might be similar to an `Array`.

This type error reported on line 137 is not a bug or a missing feature of the TypeScript compiler. The error is caused due to the **`instanceof`** check (line 134) determining whether the input is an instance of the `Array` class and the conditional inside the `Flatten` type tests whether the input is structually like an `Array`. A value can be structually like an `Array` without being **`instanceof`** `Array`, and therefore the TypeScript compiler cannot determine which branch in the `Flatten` type should be taken for the return on line 137, and the TypeScript compiler therefore correctly reports an error.

2.2.13 Index Types

An index type is a type operator used to read a property from another type. Consider the following function, where both the type and the implementation accurately describe reading a specific property from an object.

```
140 function read<T, K extends keyof T>(obj: T, key: K): T[K] {
141     return obj[key]
142 }
```

The `read` functions is parameterized by two generic types (`T` and `K`), and the function takes two parameters as input (`obj` and `key`). The generic type `K` is constrained such that it must be a subtype of `keyof T`. The type `keyof T` is a union type containing all the string constants describing the property names from the type `T`. Therefore the type `K` can both be a single string describing a single property name from the `T` type, or a union type containing a subset of the property names from the `T` type.

The index type in the above is the `T[K]` type, which describes reading from the type `T` the property described by the property name in `K`.

By using the `read` function from above, the TypeScript compiler is able to correctly infer that the `quz` variable in the following code is a **`boolean`**.

```
143 var quz = read({foo: true, bar: 123}, "foo");
```

2.2.14 Index Types and Turing-completeness

The TypeScript type system disallows simple recursive type definitions, such as the following type, which describes a potentially arbitrarily deeply nested list.

```
144 type Deep<T> = T | Array<Deep<T>>
```

The above type is not allowed by the TypeScript compiler as the type `Deep` circularly references itself.

However, it is still possible to create recursive types using object types and index types. These recursive types allow, together with other features of the TypeScript language, for the creation of a Turing machine made entirely in the TypeScript type system [11]. As an example of the power of these constructs, the following code implements addition on a unary representation of numbers using TypeScript types. The code will not be described in detail. The recursion happens in the `Add` type where the condition "`IsZero<T1> extends true`" is used to determine whether another step in the recursion should happen, and the `forceEquality` method forces the two input types to represent the same number. Therefore the TypeScript compiler will produce a type error if the two additions on line 167 do not result in the same number.

```
145 interface AnyNumber { prev?: AnyNumber, isZero: true|false };
146 interface PositiveNumber<T extends AnyNumber>
147   { prev: T, isZero: false };
148
149 type IsZero<TNumber extends AnyNumber> = TNumber["isZero"];
150 type Next<TNumber extends AnyNumber> =
151   { prev: TNumber, isZero: false };
152
153 type Zero = {isZero: true}
154 type One = Next<Zero>;
155 type Two = Next<One>;
156 type Three = Next<Two>;
157
158 type Prev<TNumber extends AnyNumber> =
159   TNumber extends PositiveNumber<infer Prev> ? Prev : never;
160
161 type Add<T1 extends AnyNumber, T2 extends AnyNumber> = {
162   "recurse": Next<Add<Prev<T1>, T2>>
163   "stop": T2
164 }[IsZero<T1> extends true ? "stop" : "recurse"];
```

```
165
166 function forceEquality<T1, T2 extends T1>() { }
167 forceEquality<Add<Three, One>, Add<Two, Two>>();
```

2.3 TypeScript Declaration Files

As stated in the thesis statement (Section 1.2), the main focus of this thesis is TypeScript declaration files. Understanding TypeScript declaration files is only a small addition on top of the understanding of JavaScript applications and TypeScript types from the previous sections.

TypeScript declaration files are TypeScript files containing only types and no implementations. TypeScript declaration files use the keyword `declare` to declare that a variable exists without assigning a value to the variable. Consider as an example the below TypeScript declaration file:

```
168 interface IMyLib {
169     add(a: number, b: number): number;
170     mult(a: number, b: number): number;
171 }
172 declare var MathLib: IMyLib;
```

The declaration file declares the variable `MathLib` with the type `IMyLib` (line 172). The type `IMyLib` contains two methods: `add` and `mult` (line 169–170). The TypeScript compiler assumes the existence of the `MathLib` variable if a TypeScript application imports the above TypeScript declaration file.

When a TypeScript application imports a TypeScript declaration, the TypeScript compiler makes no checks that the variables in the declaration file exist when the application is executed. It is the programmers own responsibility to ensure that the types in a TypeScript declaration file correspond to values available at runtime. Checking that the types in a TypeScript declaration file match the values in a library implementation can be very difficult to do automatically. Consider the example implemetation in the following code where the `_.extend` method from the Underscore.js[15] is used to build the `MathLib` object.

```
173 var MathLib = {};
174 _.extend(MathLib, {
175     add: (a, b) => a + b
176 });
177 _.extend(MathLib, {
```

[15] https://underscorejs.org/docs/underscore.html#section-107

```
178     mult: (a, b) => a * b
179 });
```

For a human, it is easy to realize what the shape of the `MathLib` object will be after the library has finished initializing. However, the implementation of the `_.extend` method uses JavaScript patterns that are very difficult for any JavaScript program analysis to reason about. The TypeScript compiler would wrongly conclude that the above implementation does not satisfy the types in the TypeScript declaration file if tasked with checking whether the implementation matches the declaration.

Patterns similar to the above use of the `_.extend` method are used in real JavaScript libraries, and it can therefore be tough for an automated approach to conclude whether the types in a TypeScript declaration file match the values from an implementation. Developers could manually check whether a declaration file matches an implementation, however, the massive size and complexity of some libraries make this task overwhelming.

2.4 Errors in TypeScript Declaration Files

TypeScript declaration files are error-prone, and many different kinds of errors can occur.

This section will discuss various errors that have appeared in real-world TypeScript declaration files. Each error description contains a footnote with a link to a real example of the described error.

All the errors have been detected using the tools described in Part II. Pull requests were created by the author of the thesis for all the errors, and the declaration file authors merged all of these pull requests. The fact that the pull requests were merged supports the claim made in the thesis statement (Section 1.2) that automated program analysis techniques can assist in the development and maintenance of TypeScript declaration files.

Typos

Many errors are simple typos where the author of the declaration file has mistyped some name. An instance of this error happened in the PIXI.js library where `cacheUniformLocations` had been mistyped as `cachUniformLocations`.[16]

Dead Types

Libraries receive updates, and the declaration files of the libraries need to be updated to match the new library implementations. In that process the

[16] `https://github.com/DefinitelyTyped/DefinitelyTyped/pull/5569/files`

authors might forget to remove features that no longer exist in the library implementation.[17]

Missing Types

Sometimes developers forget to document some part of a library when writing a TypeScript declaration file.[18]

Static vs. Instance Fields

Similar to for example Java, TypeScript has classes with both static and instance fields. An author of a declaration file can accidentally forget to declare a field as static.[19]

Wrong Types

Sometimes the declared types in a declaration file are just wrong. A variable might be declared as having the type `boolean` while the variable contains a `string` when using the implementation.[20]

Wrong Location

The right type might be documented in the wrong place. Such a mistake can, for example, happen in TypeScript declaration files with multiple nested modules, where the authors got confused about where some feature of the library belonged.[21]

Wrong inheritance

Inheritance between classes can be complicated in larger libraries, especially if a few classes break a general pattern of most classes extending from a common super-class. In these situations, it can happen that classes are declared as extending the wrong super-class.[22]

Syntax Confusion

Classes and interfaces in TypeScript can both have constructors. A constructor is a function that is called using the `new` keyword from JavaScript. The

[17] https://github.com/DefinitelyTyped/DefinitelyTyped/pull/7856/files
[18] https://github.com/DefinitelyTyped/DefinitelyTyped/pull/7857/files
[19] https://github.com/DefinitelyTyped/DefinitelyTyped/pull/7858/files
[20] https://github.com/DefinitelyTyped/DefinitelyTyped/pull/14954/files
[21] https://github.com/pixijs/pixi-typescript/pull/32/files
[22] https://github.com/DefinitelyTyped/DefinitelyTyped/pull/7923/files

syntax to define a constructor is slightly different between classes and interfaces, and this difference can cause errors if an author of a declaration file gets it wrong.[23]

Similar confusion can happen inside interfaces, where function types are not declared the same way as function types outside interfaces.[24]

Another syntax confusion is related to classes and modules. A class in TypeScript has a constructor that can be called to construct an instance of the class, and a module does not have such a constructor. Sometimes developers wrongly declare a module as a class.[25]

Strict-nulls

The strict-nulls feature of TypeScript was introduced with TypeScript 2.0. Some TypeScript declaration files written before TypeScript 2.0 have not been updated with information about which variables can possibly be `null`.[26]

[23] https://github.com/pixijs/pixi-typescript/pull/50#issuecomment-209369473
[24] https://github.com/DefinitelyTyped/DefinitelyTyped/pull/7925/files
[25] https://github.com/pixijs/pixi-typescript/pull/60/files
[26] https://github.com/pixijs/pixi-typescript/pull/144/files

Chapter 3

Automated Program Analysis

This thesis is about automated program analysis of TypeScript declaration files. The previous chapter introduced the concepts necessary to understand TypeScript declaration files, and this chapter will introduce the automated program analysis techniques used in the remainder of the thesis. This section presents three different kinds of techniques for automated program analysis: 1) type analysis, which uses constraints to assign a type to all expressions in a program, 2) data-flow analysis, which uses lattices modeling an abstract state and a fixpoint computation to overapproximate all possible executions of a program, and 3) dynamic analysis, which observes executions of the program with concrete inputs.

Both type analysis and data-flow analysis are a kind of static analysis, as neither of these executes the program as part of their analysis.

The remainder of this chapter will present the automated analysis techniques used in the research articles presented in Part II. The first section introduces unification and subtyping-based type inference; these are used in Chapter 5. The next section introduces data-flow analysis, which is used in Chapter 7. The next section introduces two dynamic analysis techniques. The first dynamic analysis automatically tests applications, which is used in Chapter 6. The second dynamic analysis technique is a pragmatic approach for improving the precision of static analysis using snapshots, which is used in Chapter 5.

3.1 Type Analysis

Programmers often want some guarantees about their programs when the programs run. Such a guarantee could be that the program never attempts to call a value that is not a function. One tool to get these kinds of guarantees is type analysis. Type analysis uses constraints generated from the program

to assign a type to each of the expressions in the program.

As an example, consider the following JavaScript program.

```
180 foo = "a string";
181 foo();
```

The program first assigns a string value to the `foo` variable, and next the `foo` variable is called as a function. A type analysis will from line 180 add the constraint that `foo` is a **`string`**, and from line 181 a type analysis will add the constraint that `foo` must be a function. Solving the constraints that `foo` is a string while also being a function is impossible, and the type analysis therefore reports an error. Programs for which a solution to the constraints exists are said to be typeable. For these programs, the output of the type analysis is a type assigned to every expression in the program.

There are many ways for a type analysis to create and solve constraints. The following sections will describe two common techniques.

Type analysis techniques tend to execute very fast, and type analysis techniques also tend to be modular, meaning that one function can be analyzed while only using the inferred types of other functions.

Type-based analysis has been used for detecting type errors for realistic JavaScript applications [47]. Such type analysis for realistic JavaScript applications tends to be unsound, meaning that the analysis makes unrealistic assumptions about the behavior of the program in order to avoid situations where the output of the analysis would be unusable. However, by restricting the analysis to a subset of the JavaScript language, it is possible to create a sound type-based analysis that allows for ahead-of-time compilation of JavaScript [31].

3.2 Unification-Based Type Inference

The type inference used in Chapter 5 is a combination of unification-based type inference and subtyping-based type inference. This section will present unification-based type inference. Unification-based type inference is by far the fastest static analysis technique presented in this thesis. However, it is also the most imprecise.

Several real-world languages use a type inference system based on unification where type inference can infer types for all expressions without using any type annotations [60, 73, 81]. This type inference system was independently discovered by Roger Hindley [55] and Robin Milner [80] and is often called the Hindley-Milner type system. This section is not a presentation of the full Hindley-Milner type system; it a presentation for the reader to understand the analysis in Chapter 5.

This section will construct a simple type inference system for typing a subset of JavaScript without objects, functions, and type coercions. Unification-based type systems are mostly used for functional and not imperative languages. However, to better relate to JavaScript/TypeScript and the papers in Part II, the system presented here will be for an imperative language.

A unification based type inference system is based on constraints about which expressions in a program must have the same type. When the analysis constrains two expressions to have the same type, then the types of the two expressions are said to be unified.

First, some syntax will be defined that helps make the upcoming rules more compact. Two square brackets around an expression should be read as the "the type of the expression". For example, ⟦`x`⟧ should be read "the type of `x`". The symbol ≡ will be used to denote that two types are unified, e.g. ⟦`x`⟧ ≡ **`boolean`** means that the type of `x` must be **`boolean`**. The ≡ relation is reflective, communitative, and transitive, and it therefore defines a set of equivalence classes. We will assume that a variable has the same type in all program locations where the variable is used. As an example it is implicitly assumed that the `a` on line 182 has the same type as the `a` on line 183.

If we for now limit the language to just assignments, numbers and additions, we can introduce the following rules for typing programs:

- A number always has the **`number`** type. (For example, ⟦`2`⟧ ≡ **`number`**.)
- Both the left and the right side of an assignment has the same type. (For example for the expression `a=b` the following constraint can be added: ⟦`a`⟧ ≡ ⟦`b`⟧.)
- The + operator can either add two numbers or concatenate two strings. We therefore assume that both operands have the same type, and that the resulting type from the operation is the same type as the arguments. (For example for `a+b` the following constraint can be added: ⟦`a`⟧ ≡ ⟦`b`⟧ ≡ ⟦`a+b`⟧.)

Consider as an example that we want to find a unique type for all the variables in the following program.

```
182 a = 2;
183 b = a;
184 c = b + 3;
```

Using the previously defined syntax, we can add the following unification constraints.

- ⟦`2`⟧ ≡ **`number`**
- ⟦`2`⟧ ≡ ⟦`a`⟧
- ⟦`b`⟧ ≡ ⟦`a`⟧
- ⟦`b`⟧ ≡ ⟦`3`⟧

- ⟦3⟧ ≡ ⟦number⟧
- ⟦c⟧ ≡ ⟦b+3⟧

We have now defined a set of constraints about types that are supposed to be the same, and we are interested in finding the equivalence classes defined by these constraints. The equivalence classes are the unique solution obtained by merging unified types into the same equivalence class until all unified types are in the same equivalence class. For the above example, there is just one equivalence class containing all the types. Therefore all the expressions must have the **number** type, as this type is the only concrete type in the equivalence class. The union-find data structure [108] is often used for efficiently finding the equivalence classes.

Type errors happen in this type inference system when two different concrete types end up in the same equivalence class. Consider as an example the following two line program:

```
185 a = 2;
186 a = "foo"
```

Creating constraints for the above program similarly to before will end up with the type **number** and type **string** in the same equivalence class as ⟦a⟧. Therefore the type inference system is unable to assign a unique type for the a variable, and the type inference system therefore reports an error.

3.2.1 Unification-Based Inference Rules

The following rules describe how to type a subset of JavaScript with a unification-based type inference system. Each rule has the JavaScript syntax being typed on the left of the ⇒ symbol, and on the right of the ⇒ symbol is the constraint imposed by the syntactic construct. In each rule, the syntax shown on the left is an example. For example, even though the first rule is exemplified for the string "foo", the rule applies to all string literals.

- "foo" ⇒ ⟦"foo"⟧ ≡ **string**
- **true** ⇒ ⟦**true**⟧ ≡ **boolean**
- 42 ⇒ ⟦42⟧ ≡ **number**
- a + b ⇒ ⟦a⟧ ≡ ⟦b⟧ ≡ ⟦a+b⟧
- a - b ⇒ ⟦a⟧ ≡ ⟦b⟧ ≡ ⟦a-b⟧ ≡ **number**
- a / b ⇒ ⟦a⟧ ≡ ⟦b⟧ ≡ ⟦a/b⟧ ≡ **number**
- a % b ⇒ ⟦a⟧ ≡ ⟦b⟧ ≡ ⟦a%b⟧ ≡ **number**
- a == b ⇒ ⟦a⟧ ≡ ⟦b⟧ and ⟦a==b⟧ ≡ **boolean**
- a != b ⇒ ⟦a⟧ ≡ ⟦b⟧ and ⟦a!=b⟧ ≡ **boolean**
- a < b ⇒ ⟦a⟧ ≡ ⟦b⟧ and ⟦a<b⟧ ≡ **boolean**
- **if**(a) {..} **else** {..} ⇒ ⟦a⟧ ≡ **boolean**
- **while**(a) {..} ⇒ ⟦a⟧ ≡ **boolean**

- `a = b` $\Rightarrow [\![\texttt{a}]\!] \equiv [\![\texttt{b}]\!]$

With these rules, types can be assigned to expressions on all JavaScript programs that use only the above syntactic constructs.

3.2.2 Adding Objects

Object types can be added to unification-based inference rules for JavaScript [47]. When objects are added to the inference rules, two unified object types must have the same properties, and their properties are therefore recursively unified.

Unification-based inference for objects can work in some languages for toy examples in JavaScript. However, lots of realistic JavaScript applications use features such as width subtyping (described in section 2.2.4). Width subtyping cannot be described using unification-based inference constraints, and thus using unification-based constraints will result in a widely inaccurate analysis for realistic JavaScript applications.

3.2.3 Adding Functions

A crucial component of the Hindley-Milner type system is polymorphism. Consider as an example the following application:

```
187 function id(x) {
188     return x;
189 }
190 a = id(2);
191 b = id("foo");
```

The program would not be typeable with the approach described so far. The type of the variable `x` on line 188 would be unified with both a number and string type (from line 190 and 191).

The Hindley-Milner type system solves this issue by inferring polymorphic types for functions. In the above example, a polymorphic type would first be inferred for the `id` function, and this type would afterwards be used when typing the `a` and `b` variables. In the above program, the inferred type for the `id` function would be $\forall t : t \rightarrow t$, which is a type that describes that the `id` function can have any function type where the parameter and return type are the same.

The two variables `a` and `b` can be typed using this inferred type for the `id` function. For the `a` variable the type for the `id` function will be instantiated with the **`number`** type, and for the `b` variable the type for the `id` function will be instantiated with the **`string`** type.

Thereby the program becomes typeable. The type assigned to `a` is **`number`**, the type assigned to `b` is **`string`**, and the type assigned to `x` is a polymorphic type that depends on how the `id` function is called.

3.3 Subtyping-Based Type Inference

The analysis used in Chapter 5 is a combination of a unification-based and subtyping-based type inference. Subtyping-based type inference was formalized by Pottier [88]. The original work by Pottier was done on the lambda calculus, but it can be expanded to cover common JavaScript features such as objects. Subtyping-based type inference is used by the Flow type system for JavaScript [44]. Subtyping-based type inference is generally significantly slower than unification-based type inference, however, more programs can be typed using subtyping.

As with the section on unification-based type inference, this section will focus on presenting the parts of subtyping-based type inference that are relevant to understanding the analysis in Chapter 5. For the same reason, this section will create a subtyping-based type inference system for a subset of JavaScript.

This section will focus on typing programs with objects, as this is where subtyping is really useful compared to unification-based type inference. Subtyping is based on the idea that when a type `T` is expected in a program location, if `S` is a subtype of `T` (written $\texttt{S} \leq \texttt{T}$) then values of type `S` can safely be used instead of values of type `T`. For example, if we have a type system with a type `number` that represents all numbers and a type `float` representing all floating point numbers, then the subtyping relation would be **`float`** $\leq$ **`number`**, and importantly the reverse does not hold, so **`number`** $\not\leq$ **`float`**.

The subtyping relation $\leq$ forms a partial order on types as the $\leq$ relation is reflexive, transitive, and antisymmetric. We assume the existence of a bottom and a top type. The bottom type, represented by the symbol $\bot$, is a type that is a subtype of all other types. Similarly, the top type, represented by the symbol $\top$, has the property that all other types are subtypes of $\top$. Written concisely: $\forall t : \bot \leq t \leq \top$.

3.3.1 Objects, Lower Bounds and Upper Bounds

The object types presented here are structurally typed, meaning that only the shapes, and not the names of the types, determine whether two object types are subtypes of each other. Width-subtyping and depth-subtyping [87] are used to create a subtyping relation between object types. With width-

subtyping, an object type `A` is a subtype of another object type `B` if the properties of `A` are a subset of the properties of `B`. Depth subtyping between objects means that an object type `A` is a subtype of an object type `B` if both types have the same number of properties and all the properties of `A` are subtypes of the corresponding properties in `B`. The transitivity of the subtyping relation means that width and depth subtyping can be combined such that an object type `A` is a subtype of an object type `B` if all the properties of `A` are present in `B` and all those properties in `A` are subtypes of the corresponding properties in `B`.

The types of objects will be described similarly to object constants. For example, the type of the object declared in the following program on line 193 can be described as `{foo: null, bar: number}`, which is an object type where the property `foo` has type `null` and the property `bar` has type `number`. Because of the width-subtyping the following subtyping relation holds (assuming `float` $\leq$ `number`):

`{foo: float, bar: string}` $\leq$ `{foo: number}`.

The subtyping-based type inference created here will be able to assign types to a program like the following:

```
192 a = {foo: 42};
193 b = {foo: null, bar: 22};
194 if (input > 0) {
195    c = a;
196 } else {
197    c = b;
198 };
199 output = c.foo;
```

In order to create a type inference capable of typing the above program we will introduce the concepts of lower bound and upper bound types. In the same way that the unification-based type inference assigned a type to every expression, in the subtyping-based type inference both a lower bound and an upper bound type will be assigned to every expression.

Intuitively, the lower bound is the type an expression *must* have. For example on line 199 the `c` variable must be an object that has a `foo` property. Likewise, the upper bound is the type an expression *can* have. For example on line 192 the variable `a` *can* at most have the type `{foo: number}`.

Subtyping based type inference works by finding both the lower bound and the upper bound types of every expression, and then the inferred type for an expression must be between the found lower and the upper bound types. That is, if we for an expression `x` denote the upper bound type as $[\![\texttt{x}]\!]^{\uparrow}$ and the lower bound type as $[\![\texttt{x}]\!]^{\downarrow}$ then the inferred type for `x` is a type $[\![\texttt{x}]\!]$ such that $[\![\texttt{x}]\!]^{\uparrow} \leq [\![\texttt{x}]\!] \leq [\![\texttt{x}]\!]^{\downarrow}$. A type error happens when no such type

exists. Consider the following program as an example of a type error that can be detected using subtyping-based type inference.

```
200 x = {foo: 42};
201 x.bar
```

From line 200 it can be deduced that the upper bound type of `x` is an object type containing a `foo` property, and from line 201 it can be deduced that `x` must be an object with a `bar` property. Since there exist no object type that has at least a `bar` property and at most a `foo` property, the above program results in a type error.

3.3.2 Subtyping-Based Inference Rules

The subtyping-based inference rules presented above focus on objects and do not include rules for all the constructs that had rules in the unification-based type inference. As with the unification-based inference rules, the syntactic construct to the left of the $\Rightarrow$ symbol is an example of the syntax that can be typed using the rule, and the right side of the $\Rightarrow$ symbol is the constraint imposed.

In the following, the right side of the $\Rightarrow$ symbol describes type constraints where for example `{foo:bar}` is an object type that has a `foo` property with the `bar` type, and $[\![$`{foo:bar}`$]\!]$ is the type of the expression `{foo:bar}`.

- `"foo"` $\Rightarrow$ **`string`** $\leq [\![$`"foo"`$]\!]^{\uparrow}$
- `42` $\Rightarrow$ **`number`** $\leq [\![$`42`$]\!]^{\uparrow}$
- `{foo:bar...}` $\Rightarrow$ `{foo:`$[\![$`bar`$]\!]^{\uparrow}$`...}` $\leq [\![$`{foo:`$[\![$`bar`$]\!]^{\uparrow}$`...}`$]\!]^{\uparrow}$
- `a = b` $\Rightarrow [\![$`b`$]\!]^{\uparrow} \leq [\![$`a`$]\!]^{\uparrow}$
- `a = b` $\Rightarrow [\![$`b`$]\!]^{\downarrow} \leq [\![$`a`$]\!]^{\downarrow}$
- **`if`**`(a) {..}` **`else`** `{..}` $\Rightarrow [\![$`a`$]\!]^{\downarrow} \leq$ **`boolean`**
- `x.foo` $\Rightarrow [\![$`x`$]\!]^{\downarrow} \leq$ `{foo:`$[\![$`x.foo`$]\!]^{\downarrow}$`}`
- `a - b` $\Rightarrow [\![$`a`$]\!]^{\downarrow} \leq$ **`number`** and $[\![$`b`$]\!]^{\downarrow} \leq$ **`number`** and **`number`** $\leq [\![$`a-b`$]\!]^{\uparrow}$

The following program will be used to demonstrate how the inference rules works.

```
202 x = {foo: 42, bar: "str"};
203 z = x.foo;
204 z - 13;
```

In the program the subtyping rules intuitively force `z` to be at least a number (line 204), `x` must be at least an object that contains at least a `foo` property of type number (line 203), and `x` can at most be an object with `foo` and `bar` properties with **`number`** and `string` type, respectively (line 202). Therefore, the type of `z` must be **`number`**, and the type of `x` can either be `{foo:` **`number`**`}` or `{foo:` **`number`**`, bar:` **`string`**`}`.

Intuitively, the information that `z` must be a **number** flows from the use of `z` in the subtraction and to the definition of `z`, and the constraint that `z` can at most be a **number** flows from the `42` on line 202 down to the use of `z` on line 204. Thereby the type constraints are bidirectional; they flow both forwards and backwards in the program.

On object types the constraints apply recursively on each property. So for example the constraint $[\![\texttt{x}]\!] \leq \{\texttt{foo:}[\![\texttt{bar}]\!]\}$ implies that $[\![\texttt{x}]\!].\texttt{foo} \leq [\![\texttt{bar}]\!]$.

Applying the subtyping-based inference rules to the above program gives the following constraints:

1. **string** $\leq [\![\texttt{"str"}]\!]^{\uparrow}$
2. **number** $\leq [\![\texttt{42}]\!]^{\uparrow}$
3. **number** $\leq [\![\texttt{13}]\!]^{\uparrow}$
4. $\{\texttt{foo:}[\![\texttt{42}]\!]^{\uparrow}\texttt{,bar:}[\![\texttt{"str"}]\!]^{\uparrow}\} \leq [\![\{\texttt{foo:}[\![\texttt{42}]\!]^{\uparrow}\texttt{,bar:}[\![\texttt{"str"}]\!]^{\uparrow}\}]\!]^{\uparrow}$
5. $[\![\{\texttt{foo:}[\![\texttt{42}]\!]^{\uparrow}\texttt{,bar:}[\![\texttt{"str"}]\!]^{\uparrow}\}]\!]^{\uparrow} \leq [\![\texttt{x}]\!]^{\uparrow}$
6. $[\![\texttt{x.foo}]\!]^{\uparrow} \leq [\![\texttt{z}]\!]^{\uparrow}$
7. $[\![\{\texttt{foo:}[\![\texttt{42}]\!]^{\downarrow}\texttt{,bar:}[\![\texttt{"str"}]\!]^{\downarrow}\}]\!]^{\downarrow} \leq [\![\texttt{x}]\!]^{\downarrow}$
8. $[\![\texttt{x.foo}]\!]^{\downarrow} \leq [\![\texttt{z}]\!]^{\downarrow}$
9. $[\![\texttt{z}]\!]^{\downarrow} \leq$ **number**
10. $[\![\texttt{13}]\!]^{\downarrow} \leq$ **number**
11. **number** $\leq [\![\texttt{z-13}]\!]^{\downarrow}$
12. $[\![\texttt{x}]\!]^{\downarrow} \leq \{\texttt{foo:}[\![\texttt{x.foo}]\!]^{\downarrow}\}$

Solving these constraints is done by joining all the constraints where the same type is mentioned and by applying all the implied constraints on object properties.

Focusing on the `z` variable, constraints 8 and 9 can be joined to form the new constraint $[\![\texttt{x.foo}]\!]^{\downarrow} \leq [\![\texttt{z}]\!]^{\downarrow} \leq$ **number**, which gives us what we need to know about the lower bound type of `z`.

Joining constraints 4 and 5, and applying the rule that object properties apply recursively, gives the new constraint $[\![\texttt{42}]\!]^{\uparrow} \leq [\![\texttt{x.foo}]\!]^{\uparrow}$. Joining this new constraint with constraints 2 and 6 gives the new constraint: **number** $\leq [\![\texttt{42}]\!]^{\uparrow} \leq [\![\texttt{x.foo}]\!]^{\uparrow} \leq [\![\texttt{z}]\!]^{\uparrow}$.

From these we now know that $[\![\texttt{z}]\!]^{\downarrow} \leq$ **number** and **number** $\leq [\![\texttt{z}]\!]^{\uparrow}$. The type $[\![\texttt{z}]\!]$ must satisfy the constraint $[\![\texttt{z}]\!]^{\uparrow} \leq [\![\texttt{z}]\!] \leq [\![\texttt{z}]\!]^{\downarrow}$. Joining these constraints gives the constraint:

$$\textbf{number} \leq [\![\texttt{z}]\!]^{\uparrow} \leq [\![\texttt{z}]\!] \leq [\![\texttt{z}]\!]^{\downarrow} \leq \textbf{number}.$$

From this final constraint it is clear that the type of `z` should both be a subtype of **number**, and **number** should be a subtype of `z`. The only possible solution to this is that the type of `z` is **number**, and thus we have found a type for `z`.

The same thing can be done for `x`, where we will end up with the constraint:

$\texttt{\{foo: number, bar: string\}} \leq [\![\texttt{x}]\!]^{\uparrow} \leq [\![\texttt{x}]\!] \leq [\![\texttt{x}]\!]^{\downarrow} \leq \texttt{\{foo: number\}}$

The type of `x` can then be chosen freely to be either the lower bound or the upper bound type in the above constraint.

3.4 Data-Flow Analysis

Chapter 7 uses data-flow analysis to verify the absence of errors in TypeScript declaration files. Data-flow analysis is based on approximating the behavior of programs in such a way that the approximation includes every possible execution of the program. The result from a data-flow analysis can therefore be used to reason about what cannot possibly happen during execution of a program. We use this property of a data-flow analysis in Chapter 7 to conclude whether it is possible for values violating the types described by a TypeScript declaration file to appear in the interactions between a library and a client.

Data-flow analysis is by far the slowest analysis technique described in this thesis. However, among the static analysis techniques, data-flow analysis can give the most precise and sound results for realistic JavaScript applications. In a data-flow analysis, a small change in one part of the program can potentially affect the data-flow in a completely different part of the program, and a data-flow analysis can therefore generally not be made modular.

Many data-flow analyzers have been made that can analyse realistic JavaScript programs [57, 63, 84, 105]. However, common to all of these is that they either have extreme difficulties analyzing commonly used JavaScript libraries, or the analysis makes many unrealistic assumptions on what can happen during program execution. Even the analyzers that claim to be sound make unsound assumptions about the behavior the programs in some corner cases [75].

Data-flow analysis can be made flow-sensitive, as opposed to the previously described unification based and subtyping based techniques, which were flow-insensitive. A flow-sensitive analysis can assign different abstract values to a variable depending on the program location, whereas a flow-insensitive analysis can only assign a single abstract value to a variable. For example, a flow-sensitive analysis might know that a variable has a particular constant value at a specific program statement, whereas a flow-insensitive analysis can only reason about what the value of a variable can be through the program as a whole.

As an example of how data-flow analysis works, and what data-flow analysis can be used for, consider the following example program [66]:

```
205 a = 1; c = 0;
```

```
206 for (i = 0; i <= 10; i++) {
207   b = 2;
208   d = a + b;
209   e = b + c;
210   c = 4;
211 }
```

This section will present a simple analysis for assisting a compiler in producing efficient machine code for the program. The analysis reasons about which variables have known constant values at some program statements. For example, looking at line 209, if the variables `b` and `c` are known constants at that program location, then the assignment to `e` can be simplified to an assignment of a known constant.

The analysis is a data-flow analysis that can reason about which variables have known constant values at which program locations. The analysis stores a set of known constants at every program location by using the constant propagation lattice [61]. Informally, the constant propagation lattice stores, at every program statement, which variables have a known constant value and the constant for these variables.

In this analysis, we will, for simplicity, ignore the variable `i`, which is only used for looping a limited amount of times. In order to create this simple data-flow analysis, the control flow of the program must be known. The control flow for the program above is shown in Figure 3.1, where each statement is named using an uppercase letter, and the arrows show the control flow of the program. Each arrow points to the successor(s) of a statement and the arrow points from the predecessor.

3.4.1 Implementing a Data-Flow Analysis

This section will not be a formal mathematical introduction to data-flow analysis. This section is more an informal presentation of the fundamentals of the analysis used in Chapter 7. Implementing a data-flow analysis for JavaScript is a massive undertaking. Even though this section is an informal introduction, there is much work, not touched in any way in this section, required to implement a data-flow analysis that works on realistic JavaScript applications.

At its core, data-flow analysis is about defining a monotone function that, given an abstract state and a program statement, computes a new abstract state [61]. We will call this function a `transfer` function. An initial abstract state is defined for the initial program statement. The data-flow analysis will use the initial abstract state and the transfer function to calculate new abstract states and propagate these abstract states to other program statements. The mathematical foundation for the abstract states is lattices [27].

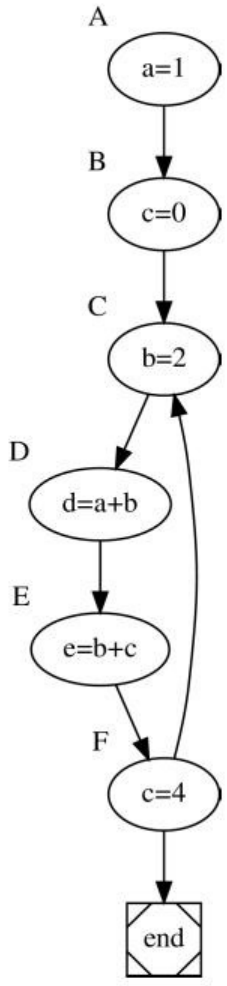

Figure 3.1: A control flow graph describing the control flow of the code example.

To construct an example data-flow analysis, consider the above program, which is shown as a control flow graph in Figure 3.1. For this program, we will calculate which variables are constants at some program point. We therefore define the abstract state in the data-flow analysis as the variables that have constant values along with the constant value of the variable. We will write such a state as `{a=1,b=2}` for the state where `a` is the constant `1` and `b` is the constant `2`.

The transfer function in the data-flow analysis is used to create a new state based on an incoming state at a program point, and the statement at

Algorithm 1: Worklist algorithm for computing fixpoint of dataflow constraints.

Input: A start node in the flow graph, the flow graph, and a transfer function
Output: *states*
DATAFLOWANALYSIS(*startNode*, *flowGraph*, *transfer*)

```
1  states ← map that maps all nodes to ⊥
2  worklist ← [startNode]
3  while worklist not empty do
4      node ← pop from worklist
5      oldState ← get mapping for node in states
6      nextState ← transfer(node, oldState)
7      for successors succ of node in flowGraph do
8          prevState ← get mapping for succ in states
9          succState ← nextState ⊔ prevState
10         if prevState = succState then
11             skip to next successor
12         states ← states + [succ → succState]
13         worklist ← worklist + succ
```

that program point. For example for the node A in Figure 3.1 the incoming state is the empty state `{}`, and the transfer function will create a new state reflecting the known constant variable values after the program point has executed, in this case that new state will be `{a=1}`.

Some program points have multiple predecessors. These program points will therefore receive a new state from all their precessors. For these program points, the state being calculated is the most precise overapproximation of all the incoming states. So for example for the node C in Figure 3.1, if the two incoming states are `{a=1,c=0}` and `{a=1,c=4}`, then the most precise overapproximation will be `{a=1}`, as the variable `a` was the only variable with the same constant value between the two incoming states.

The above description is specific to a data-flow analysis that finds constant values. Algorithm 1 shows one general implementation of data-flow analysis. The algorithm can be used to implement the analysis described above by setting the `transfer` function to be the transfer function informally described above.

Algorithm 1 begins by creating a map *states* mapping program points to their associated abstract state (line 1). The *states* map initially maps all program points to $\bot$. The abstract state $\bot$ represents that no abstract state has yet reached that program point in the data-flow analysis. In the previously described analysis, the $\bot$ abstract state would be different from the

abstract state `{}`, which represents that no variables have known constant values. Using this $\bot$ abstract state ensures that all program points will be visited by Algorithm 1 at least once.

Next, a worklist is created containing all the program points that have received a new abstract state (line 2). The algorithm then proceeds to process the elements in the *worklist* in a loop until no more elements are added to the *worklist* (line 3–13). First, the transfer function runs on the program statement and the new abstract state, and the result of this is the abstract state *nextState*, which represents the abstract state after the current program point (line 6). This *nextState* is then propagated to all the successors of the current program point (line 7–13).

Doing the propagation it is checked whether the new state changes the state at the successor, and if the new state is different, the successor is added to the *worklist* (line 13). The existing state associated with the successor is joined with the new state using the least upper bound operator $\sqcup$ (line 9). Intuitively, the least upper bound operator $\sqcup$ is the most precise overapproximation of two states, and it holds that $\forall X : \bot \sqcup X = X$. As an example of the use of the $\sqcup$ operator, in the example analysis we created before the following would hold:

$$\texttt{\{a=1,b=2,c=0\}} \sqcup \texttt{\{a=1,b=2,c=4\}} = \texttt{\{a=1,b=2\}}$$

Algorithm 1 will continue to execute until a fixpoint is reached. At this fixpoint, propagating the new state calculated from the *transfer* function will not change the state at any of the successors.

If the *transfer* function is monotone, and the lattice representing the abstract state has a finite height, then we know from the Kleene fixed-point theorem [6] that the algorithm will terminate due to Algorithm 1 always using the $\sqcup$ operator when replacing an existing state, and there only being finitely many program points containing an abstract state.

3.4.2 Real World Data-Flow Analysis of JavaScript

If one were to implement a data-flow analysis for realistic JavaScript applications based on just Algorithm 1, then it would first of all be very difficult to create an analysis capable of modeling realistic JavaScript applications, and secondly, if one successfully implements such an analysis, then the resulting data-flow analysis would likely be slow, imprecise, and generally useless.

The first problem encountered when trying to implement a data-flow analysis for realistic JavaScript applications is that the flowgraph is unknown. Specifically, the callgraph of the application is unknown. In a more statically typed language such as Java, the callgraph can be approximated using an analysis like class hierarchy analysis [40]. However, in JavaScript, such an approximation cannot be made, as the class hierarchy is unknown

at compile-time, and thus the only options are to assume a fully connected callgraph, where every call site can call every function, or to calculate the callgraph on the fly. All serious data-flow analyzers for JavaScript choose the latter option.

A second problem is the repeated application of the $\sqcup$ operator. In a data-flow analysis for realistic JavaScript applications, the abstract state is massive, as the abstract state contains all the objects in the heap. A naive implementation where the $\sqcup$ operator is applied naively for every iteration of the fixpoint computation will have terrible performance in part due to the massive size of the abstract state. Various tricks, such as lazy propagation [58], can be used to improve the performance.

A third problem is when the same program statement can behave in different ways during the execution of a program. Consider the following simple JavaScript function:

```
212 function plus(a, b) {
213   return a + b;
214 }
215 var foo = plus(1,2);
216 var bar = plus("a", "b");
```

In the algorithm outlined in Algorithm 1, each program statement has exactly one abstract state associated with the program statement, which will result in the function `plus` returning one type in the data-flow analysis. As the value returned by the `foo` function can be both a number and a string, the two variables `foo` and `bar` must both abstractly represent both a string and a number in this data-flow analysis, which is an imprecise overapproximation.

The usual trick used to avoid this imprecision is context-sensitivity. A context is an abstract representation of how the execution got to that program statement. For example, in the above program on line 213 a context could either represent that the `foo` function was called from line 215 or from line 216. Contexts are incorporated into Algorithm 1 by changing all the uses of a program statement to be a pair containing both a program statement and a context. Adding context-sensitivity to a data-flow analysis essentially duplicates the analysis effort for all the program locations that are analyzed with different contexts, and it is therefore essential to find a suitable context-sensitivity in order to create a data-flow analysis for JavaScript with the right tradeoff between precision and performance.

The problems mentioned here, and many more, need to be solved in order to create a data-flow analysis that works for realistic JavaScript applications. In Chapter 7 the state-of-the-art data-flow analysis TAJS [20, 21, 57–59] is used. TAJS addresses all the problems mentioned here, and more.

3.4.3 Most-general Clients

Libraries present a problem for a data-flow analysis. Consider as an example the following simple library.

```
217 function plus(a, b) {
218   return a + b;
219 }
```

A data-flow analysis analyzing the above program will never analyze the statement on line 218, as the function `plus` is never called from within the library. The data-flow analysis is therefore unable to find any potential issue inside the `plus` function.

The typical solution to this problem is to use a most-general client [18, 95]. A most-general client is a client designed to use all the available methods and properties of a specific library, and such a client is most-general as the client uses the library in all the ways any real-world client could use the library.

A most-general client for the above library would call the `plus` method with two random inputs. The type of these random inputs depends on whether there is some specification for how the `plus` function should be called.

Chapter 7 develops the concept of a reasonably-most-general client, which is very similar to a most-general client, except the reasonably-most-general client works under a set of assumptions. These assumptions are shown to be necessary for the analysis to analyze realistic JavaScript libraries.

3.5 Dynamic Analysis

A dynamic analysis is an analysis that executes the program being analyzed. A dynamic analysis records facts about the execution either after the program has finished executing or while the program is executed. These facts about the execution are then either presented to the user of the analysis or used for further analysis.

Dynamic analysis can only observe actual executions that happen. A dynamic analysis therefore underapproximates the possible behavior of a program, and the observations gained during the executions cannot, in general, include all possible executions. Dynamic analysis can therefore not be used to find all potential issues in a program. However, when a dynamic analysis detects an issue, the fact that an actual execution lead to the discovery of the problem means the problem is likely to be a true positive.

A benefit, compared to static analysis, is the performance of the analysis, which comes from basing the analysis on actual executions. A dynamic analysis does not execute as fast as the program would outside of the dynamic analysis. However, the slowdown of running the dynamic analysis compared to a normal execution is usually some constant factor less than 100, and performance is therefore rarely an issue when performing dynamic analysis.

3.5.1 Automated Testing

Automated testing is a dynamic analysis that automatically tests a program based on some input. That input might be a specification [35], the program itself [82], or manually written test cases [117]. The automated tester will, after inputs have been computed, execute the program as normal, with no performance penalty compared to executing the program outside of the analysis. These automated testing techniques are used to find failures where a program fails to satisfy some implicit or explicit specification.

Consider the following TypeScript function implementing division.

```
220 function safeDivision(a: number, b: number) {
221   return a / b;
222 }
```

A natural specification for the above function could be:

$$\forall \texttt{a, b: safeDivision(a, b) * b} \approx \texttt{a}$$

An automated tester can test that the function satisfies the specification by randomly creating number value that can be used to call the function `safeDivision`. For example, the automated tester might call `safeDivision(42, 7)` which results in the number `6`, and satisfies the specification as `6 * 7 = 42`. The automated tester will repeatedly test the `safeDivision` function with various random inputs for some amount of iterations. During these iterations, the automated tester might, or might not, find an input where a division by zero happens, which will break the specification. For example `safeDivision(42, 0)` will result in the JavaScript value `Infinity`, which does not satisfy that `Infinity * 0` $\approx$ `Infinity` as multiplying `Infinity` and `0` in JavaScript result in the special value `NaN`.

Feedback-Directed Automated Testing

Feedback-directed automated testing is an automated testing technique that uses values obtained through automated testing as inputs to the program being tested. As an example, consider the following example TypeScript program:

```
223 class Point {
224     constructor(public x: number, public y:number) {}
225
226     add(o: Point) : Point {
227         return new Point(this.x + o.x, this.y + o.y);
228     }
229
230     dot(o: Point): number {
231         return this.x + o.x * this.y + o.y;
232     }
233 }
```

The program defines a class `Point` with two fields `x` and `y`, and two methods `add` and `dot`. For an automated tester to test the `add` method, the automated tester needs to have a value of type `Point`. With feedback-directed automated testing, the automated tester will initially call a random method for which it has all the required inputs. In the above example program, only the constructor of the `Point` class can be called initially. The automated tester will therefore initially call the `Point` constructor, and it can then use the constructed `Point` object for further testing of the program.

In Chapter 6 feedback-directed automated testing is used to test JavaScript libraries automatically. The automated tester tests that the specification, defined by the types in a TypeScript declaration file, holds.

An approach that attempts to solve the same problem as Chapter 6, is TPD [117]. TPD uses the same specification as Chapter 6. However, TPD uses the existing test cases for the library being tested instead of automatically generating tests. TPD was developed in parallel with our technique.

3.5.2 Snapshot Assisted Analysis

Static analysis of realistic applications can be strenuous. Even after all the previously discussed problems have been addressed, some applications still present a significant challenge to even state-of-the-art data-flow analyzers.

One big issue, in particular, is how some JavaScript libraries bootstrap themselves. Consider as an example the following code, which is part of the Underscore.js initialization code.[1]

```
234 _.mixin = function(obj) {
235   _.each(_.functions(obj), function(name) {
```

[1] https://underscorejs.org/docs/underscore.html#section-187

```
236     var func = _[name] = obj[name];
237     _.prototype[name] = function() {
238       var args = [this._wrapped];
239       push.apply(args, arguments);
240       return chainResult(this, func.apply(_, args));
241     };
242   });
243   return _;
244 };
245 _.mixin(_);
```

The code first defines a `_.mixin` function, which takes all the methods from an object `obj` and assigns a modified version of the methods to the prototype of the _ object.

The important part is not to understand what the above code precisely does, but to understand how the code can destroy the precision of a static analysis. The tricky part that can destroy analysis precision is line 236, where the `name` variable can cause problems in the assignment. If a static analysis is unsure what the exact value of `name` is, then line 236 will potentially read all the properties from `obj`, and potentially overwrite all the properties of the _ object with the result of this imprecise property read.

If such imprecision happens, then the resulting abstract state in the static analysis contains an _ object where the properties are mangled together. Any use of the _ object after this initialization will thus result in the data-flow analysis being unsure which method from the library is being called, making the analysis virtually useless.

A simple and effective solution to this problem is not to let a data-flow analysis analyze the initialization of the library, but instead use dynamic analysis to analyze how the library initializes itself. This dynamic analysis would execute the initialization code of the library in some concrete environment, then record the resulting concrete state after the initialization has finished. Afterward, an abstract state can be constructed based on the concrete state from the dynamic analysis, and the data-flow analysis can start its analysis based on an abstract state constructed from the concrete state where the library is already initialized.

The approach described is used in Chapter 5, where a dynamic analysis initializes the library being analyzed, and the type inference afterward infers types based on the abstract state computed by the dynamic analysis.

Chapter 4

Conclusion

TypeScript declaration files are needed as long as libraries used by TypeScript application developers are written in JavaScript. TypeScript adoption is on the rise, however, new libraries written in JavaScript continue to appear, existing libraries receive updates, and existing libraries are rarely migrated to TypeScript. Thus the need for creating and maintaining TypeScript declaration files is not in decline.

There is little tool support for assisting with these tasks. Developers are, therefore, manually writing the declaration files, which is both tedious and error-prone.

The thesis explores and expands various techniques in programming language research and applies these techniques to the challenge of creating and maintaining TypeScript declaration files. A continuous challenge through this exploration is the complexity of both the JavaScript language and the TypeScript type system.

In conclusion, this thesis contributes four results on automated program analysis and its application to the creation and maintenance of TypeScript declaration files:

- A new type inference for automatically inferring TypeScript declaration files from JavaScript libraries. The type inference uses subset based type constraints, and dynamic analysis is used to create a snapshot that helps the analysis skip the complicated initialization of the libraries. This type inference is shown to be able to create TypeScript declaration files based on untyped JavaScript libraries automatically, and these inferred TypeScript declaration files are shown to be useful as a starting point for developers creating a new declaration file for a library.
- A novel technique for creating reports of what has changed between two versions of a library. The technique applies type inference on two

versions of the implementation and uses a TypeScript declaration file to filter out changes from the undocumented sections of the library. Evaluation of this technique showed it to be useful for assisting in the maintenance involved in keeping TypeScript declaration files up to date.

- A feedback-directed random testing approach for automatically testing JavaScript libraries, which is used for automatically detecting mismatches between the specification in a TypeScript declaration file and an implementation written in JavaScript. Developing this approach required tackling several challenges related to the complexity of the TypeScript type system such as structurally typed objects and first-class functions. The approach was used for finding real bugs in many real-world TypeScript declaration files with no false positives. Furthermore, the performance of the tool developed allows it to identify most of the bugs within a short time.

- The concept of a reasonably-most-general client (RMGC); an automated technique for data-flow analysis of JavaScript libraries. A data-flow analysis cannot achieve usable results on a JavaScript library without having a client that uses the library. An RMGC is a client, that models all possible behaviors that realistic clients could have under certain assumptions. The assumptions imposed by the RMGC were shown to be crucial for the performance and precision of the data-flow analysis when analyzing even small JavaScript libraries. The RMGC concept was used for fixing all issues in a number of TypeScript declaration files, and for, after the TypeScript declaration files had been fixed, verifying that no further errors could exist in the TypeScript declaration file under certain assumptions!

The thesis statement claimed that automated program analysis techniques could be created for assisting in the development and maintenance of TypeScript declaration files. The first contribution above has shown that an automated technique in the form of type inference assisted by dynamic analysis, can help with the development of TypeScript declaration files. The following three contributions have shown that automated techniques in the form of a new novel technique for finding changes between libraries, feedback-directed random testing, and data-flow analysis, can assist in the continuous maintenance of TypeScript declaration files.

With these four contributions, this thesis has created automated techniques that can assist in many aspects of creating and maintenance of TypeScript declaration files. All the resulting implementations are open source and available at `https://brics.dk/tstools/`, and the pull requests created

as part of the evaluations of the implementations can be found at `https://gist.github.com/webbiesdk/f82c135fc5f67b0c7f175e985dd0c889`.

Given the continued success of optionally typed languages, the challenges tackled in this thesis will continue to be relevant both for TypeScript and other optionally typed languages.

4.1 Future work

Common to the last three contributions listed above is that the tool implementing the approach outputs reports, where each report contains the affected type, and descriptions of the error found or the change detected. None of the tools can automatically make changes in the TypeScript declaration files based on the reports, the tools are currently not even able to report which line(s) in the TypeScript declaration file correspond to the report.

Future work could investigate the possibility of automatically applying patches to TypeScript declaration files based on the outputs of tools like the ones implemented in this thesis. For some errors, like "*wrong type*" from Section 2.4, it might be straight forward to implement automatic patching of the errors, as the change might be to change for example a `boolean` to a `string`. Other errors, like "*wrong location*", can present a more significant challenge to automatically patch, as multiple locations in the TypeScript declaration file needs to be changed in order to fix correctly fix a single error.

Part II

Publications

Chapter 5

Inference and Evolution of TypeScript Declaration Files

By Erik Krogh Kristensen and Anders Møller. Published in Proceedings of the 20th International Conference on Fundamental Approaches to Software Engineering.

5.1 Abstract

TypeScript is a typed extension of JavaScript that has become widely used. More than 2 000 JavaScript libraries now have publicly available TypeScript declaration files, which allows the libraries to be used when programming TypeScript applications. Such declaration files are written manually, however, and they are often lagging behind the continuous development of the libraries, thereby hindering their usability. The existing tool TSCHECK is capable of detecting mismatches between the libraries and their declaration files, but it is less suitable when creating and evolving declaration files.

In this work we present the tools TSINFER and TSEVOLVE that are designed to assist the construction of new TypeScript declaration files and support the co-evolution of the declaration files as the underlying JavaScript libraries evolve. Our experimental results involving major libraries demonstrate that TSINFER and TSEVOLVE are superior to TSCHECK regarding these tasks and that the tools are sufficiently fast and precise for practical use.

5.2 Introduction

The TypeScript [79] programming language has become a widely used alternative to JavaScript for developing web applications. TypeScript is a superset of JavaScript adding language features that are important when de-

veloping and maintaining larger applications. Most notably, TypeScript provides optional types, which not only allows many type errors to be detected statically, but also enables powerful IDE support for code navigation, auto-completion, and refactoring. To allow TypeScript applications to use existing JavaScript libraries, the typed APIs of such libraries can be described in separate *declaration files*. A public repository exists containing declaration files for more than 2 000 libraries, and they are a critical component of the TypeScript software ecosystem.[1]

Unfortunately, the declaration files are written and maintained manually, which is tedious and error prone. Mismatches between declaration files and the corresponding JavaScript implementations of libraries affect the TypeScript application programmers. The type checker produces incorrect type error messages, and code navigation and auto-completion are misguided, which may cause programming errors and increase development costs. The tool TSCHECK [47] has been designed to detect such mismatches, but three central challenges remain. First, the process of constructing the initial version of a declaration file is still manual. Although TypeScript has become popular, many new libraries are still being written in JavaScript, so the need for constructing new declaration files is not diminishing. We need tool support not only for checking correctness of declaration files, but also for assisting the programmers creating them from the JavaScript implementations. Second, JavaScript libraries evolve, as other software, and when their APIs change, the declaration files must be updated. We observe that the evolution of many declaration files lag considerably behind the libraries, which causes the same problems with unreliable type checking and IDE support as with erroneous declaration files, and it may make application programmers reluctant or unable to use the newest versions of the libraries. With the increasing adaptation of TypeScript and the profusion of libraries, this problem will likely grow in the future. For these reasons, we need tools to support the programmers in this co-evolution of libraries and declaration files. Third, TSCHECK is not sufficiently scalable to handle modern JavaScript libraries, which are often significantly larger than a couple of years ago.

The contributions of this paper are as follows.

- To further motivate our work, we demonstrate why the state-of-the-art tool TSCHECK is inadequate for inference and evolution of declaration files, and we describe a small study that uncovers to what extent the evolution of TypeScript declaration files typically lag behind the evolution of the underlying JavaScript libraries (Section 5.3).

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

- We present the tool TSinfer, which is based on TScheck but specifically designed to address the challenge of supporting programmers when writing new TypeScript declaration files for JavaScript libraries, and to scale to even the largest libraries (Section 5.4).
- Next, we present the tool TSevolve, which builds on top of TSinfer to support the task of co-evolving TypeScript declaration files as the underlying JavaScript libraries evolve (Section 5.5).
- We report on an experimental evaluation, which shows that TSinfer is better suited than TScheck for assisting the developer in creating the initial versions of declaration files, and that TSevolve is superior to both TScheck and TSinfer for supporting the co-evolution of declaration files (Section 5.7).

5.3 Motivating Examples

The PixiJS library PixiJS[2] is a powerful JavaScript library for 2D rendering that has been under development since 2013. A TypeScript declaration file[3] was written manually for version 2.2 (after some incomplete attempts), and the authors have since then made numerous changes to try to keep up-to-date with the rapid evolution of the library. At the time of writing, the current version of PixiJS is 4.0, and the co-evolution of the declaration file continues to require substantial manual effort as testified by the numerous commits and issues in the repository. Hundreds of library developers face similar challenges with building TypeScript declaration files and updating them as the libraries evolve.

From checking to inferring declaration files To our knowledge, only one tool exists that may alleviate the manual effort required: TScheck [47]. This tool detects mismatches between a JavaScript library and a TypeScript declaration file. It works in three phases: (1) it executes the library's initialization code and takes a snapshot of the resulting runtime state; (2) it then type checks the objects in the snapshot, which represent the structure of the library API, with respect to the TypeScript type declarations; (3) it finally performs a light-weight static analysis of each library function to type check the return value of each function signature. This works well for detecting errors, but not for inferring and evolving the declaration files. For example, running TScheck on PixiJS version 2.2 and a declaration file with an empty `PIXI` module (mimicking the situation where the module is known to

[2] `http://www.pixijs.com/`

[3] `https://github.com/pixijs/pixi-typescript`

```
246 export class Sprite extends PIXI.DisplayObjectContainer {
247     constructor (texture: PIXI.Texture);
248     static fromFrame: (frameId: string | number) => PIXI.Sprite;
249     static fromImage: (imageId: string, crossorigin: any,
250                        scaleMode: any) => PIXI.Sprite;
251     _height: number;
252     _width: number;
253     anchor: PIXI.Point;
254     blendMode: number;
255     onTextureUpdate: () => void;
256     setTexture: (texture: PIXI.Texture) => void;
257     shader: any;
258     texture: PIXI.Texture;
259     tint: number;
260 }
```

Figure 5.1: Example output from TSInfer, when run on PixiJS version 2.2.

exist but its API has not yet been declared) reports nothing but the missing properties of the `PIXI` module, which is practically useless. In comparison, our new tool TSInfer is able to infer a declaration file that is quite close to the manually written one. Figure 5.1 shows the automatically inferred declaration for one of the classes in PixiJS version 2.2. The declaration is not perfect (the types of `frameId`, `crossorigin`, `scaleMode`, and `shader` could be more precise), but evidently such output is a better starting point when creating the initial version of a declaration file than starting completely from scratch.

Evolving declaration files The PixiJS library has recently been updated from version 3 to version 4. Using TSCheck as a help to update the declaration file would not be particularly helpful. For example, running TSCheck on version 4 of the JavaScript file and the existing version 3 of the declaration file reports that 38 properties are missing on the `PIXI` object, without any information about their types. Moreover, 15 of these properties are also reported if running TSCheck on version 3 of the JavaScript file, since they are due to the developers intentionally leaving some properties undocumented. Our experiments presented in Section 5.7 show that many libraries have such intentionally undocumented features, and some also have properties that intentionally exist in the declaration file but not in the library[4]. While TSInfer does suggest a type for each of the new properties, it does not have any way to handle the intentional discrepancies. Our other tool

[4]This situation is rare, but can happen if, for example, documentation is needed for a class that is not exported `https://github.com/pixijs/pixi.js/issues/2312/#issuecomment-174608951`

```
Property PrimitiveShader removed from object on window.PIXI
Property FXAAFilter removed from object on window.PIXI
Property TransformManual added to object on window.PIXI
  Type: typeof PIXI.TransformBase
Property TransformBase added to object on window.PIXI
  Type:  class TransformBase { ... }
Property Transform added to object on window.PIXI
  Type: class Transform extends PIXI.TransformBase { ... }
```

(a) Some of the added or removed properties.

```
Type changed on
  window.PIXI.RenderTarget.[constructor].[return].stencilMaskStack
from StencilMaskStack to PIXI.Graphics[]
```

(b) A modified property.

Figure 5.2: Example output from TSevolve, when run on PixiJS versions 3 and 4.

TSevolve attempts to solve that problem by looking only at differences between two versions of the JavaScript implementation and is thereby better at only reporting actual changes. When running TSevolve on PixiJS version 3 and 4, it reports (see Figure 5.2(a)) that 8 properties have been removed and 24 properties have been added on the `PIXI` object. All of these correctly reflect an actual change in the library implementation, and the declaration file should therefore be updated accordingly. This update inevitably requires manual intervention, though; in this specific case, `PrimitiveShader` has been removed from the `PIXI` object but the developers want to keep it in the declarations as an internal class, and `TransformManual`, although it is new to version 4, is a deprecated alias for the also added `TransformBase`.

Changes in a library API from one version to the next often consist of extensions, but features are also sometimes removed, or types are changed. As an example of the latter, one of the changes from version 3 to 4 for PixiJS was changing the type of the field `stencilMaskStack` in the class `RenderTarget` from type `PIXI.StencilMaskStack` to type `PIXI.Graphics[]`. The developer updating the declaration file noticed that the field was now an array, but not that the elements were changed to type `PIXI.Graphics`, so the type was erroneously updated to `PIXI.StencilMaskStack[]`. In comparison, TSinfer reports the change correctly as shown in Figure 5.2(b).

A study of evolution of type declarations To further motivate the need for new tools to support the co-evolution of declaration files as the libraries evolve, we have measured to what extent existing declaration files lag be-

hind the libraries.[5] We collected every JavaScript library that satisfies the following conditions: it is being actively developed and has a declaration file in the the DefinitelyTyped repository, the declaration file contains a recognizable version number, and the library uses git tags for marking new versions, where we study the commits from January 2014 to August 2016. This resulted in 49 libraries. By then comparing the timestamps of the version changes for each library and its declaration file, respectively (where we ignore patch releases and only consider major.minor versioning), we find that for more than half of the libraries, the declaration file is lagging behind by at least a couple of months, and for some more than a year. This is notable, given that all the libraries are widely used according to the GitHub ratings, and it seriously affects the usefulness of the declaration files in TypeScript application development.

Interestingly, we also find many cases where the version number found in the declaration file has not been updated correctly along with the contents of the file.[6] Not being able to trust version numbers of course also affects the usability of the declaration files. For some high-profile libraries, such as jQuery and AngularJS, the declaration files are kept up-to-date, which demonstrates that the developers find it necessary to invest the effort required, despite the lack of tool support. We hope our new tools can help not only those developers but also ones who do not have the same level of manual resources available.

Scalability In addition to the limitations of TSCHECK described above, we find that its static analysis component, which we use as a foundation also for TSINFER and TSEVOLVE, is not sufficiently scalable to handle the sizes and complexity of contemporary JavaScript libraries. In Section 5.4 we explain how we replace the unification-based analysis technique used by TSCHECK with a more precise subset-based one, and in Section 5.7 we demonstrate that this modification, perhaps counterintuitively, leads to a significant improvement in scalability. As an example, the time required to analyze *Moment.js* is improved from 873 seconds to 12 seconds, while other libraries simply are not analyzable in reasonable time with the unification-based approach.

[5] Our data material from this study is available at `http://www.brics.dk/tstools/`.

[6] An example is Backbone.js, until our patch `https://github.com/DefinitelyTyped/DefinitelyTyped/pull/10462`.

5.4 TSinfer: Inference of Initial Type Declarations

Our inference tool TSinfer works in three phases: (1) it concretely initializes the library in a browser and records a snapshot of the resulting runtime state, much like the first phase of TScheck (see Section 5.3); (2) it performs a static analysis of all the functions in that snapshot, similarly to the third phase of TScheck; (3) lastly it emits a TypeScript declaration file. As two of the phases are quite similar to the approach used by TScheck, we here focus on what TSinfer does differently.

The Snapshot Phase

In JavaScript, library code needs to actively put entry points into the heap in order for it to be callable by application code. This initialization, however, often involves complex metaprogramming, and statically analyzing the initialization of a library like jQuery can therefore be extremely complicated [20]. We sidestep this challenge by concretely initializing the library in a real browser and recording a snapshot of the heap after the top-level code has finished executing. This is done in the same way as described by TScheck, and we work under the same assumptions, notably, that the library API has been established after the top-level code has executed. We have, however, changed a few things.

For all functions in the returned snapshot, we record two extra pieces of information compared to TScheck: (1) the result of calling the function with the `new` operator (if the call returned normally), which helps us determine the structure of a class if the function is found to be a constructor; (2) all calls to the function that occur during the initialization, which we use to seed the static analysis phase.

The last step is to create a class hierarchy. JavaScript libraries use many different and complicated ways of creating their internal class structures, but after the initialization is done, the vast majority of libraries end up with constructor functions and prototype chains. The class hierarchy is therefore created by making a straightforward inspection of the prototype chains.

The Static Analysis Phase

The static analysis phase takes the produced snapshot as input and performs a static analysis of each of the functions. It produces types for the parameters and the return value of each function.

The analysis is an unsound, flow-insensitive, context-insensitive analysis that has all the features described in previous work [47], including the treatment of properties and native functions. There are, however, some important changes.

TSCHECK analyzes each function separately, meaning that if a function `f` calls a function `g`, this information is ignored when analyzing function `g`. This works well for creating an analysis such as TSCHECK that only infers the return type of functions. When creating an analysis that also infers function parameter types, the information gained by observing calls to a function is important. Our analysis therefore does not analyze each function separately, but instead performs a single analysis that covers all the functions.

While TSCHECK opts for a unification-based analysis, we find that switching to a subset-based analysis is necessary to gain the scalability needed to infer types for the bigger JavaScript libraries, as discussed in Section 5.3. The subset-based analysis is similar to the one described by Pottier [88], as it keeps separate constraint variables for upper-bounds and lower-bounds. After the analysis, the types for the upper-bound and lower-bound constraint variables are merged to form a single resulting type for each expression.

Compared to TSCHECK, some constraints have been added to improve precision for parameter types, for example, so that the arguments to operators such as `-` and `*` are treated as numbers. (Due to the page limit, we omit the actual analysis constraints used by TSINFER.)

A subset-based analysis gives more precise dataflow information compared to a unification-based analysis, however, more precise dataflow information does not necessarily result in more precise type inference. For example, consider the expression `foo = bar || ""`, where `bar` is a parameter to a function that is never called within the library. A unification-based analysis, such as TSCHECK, will unify the types of `foo`, `bar` and `""`, and thereby conclude that the type of `bar` is possibly a string. A more precise subset-based analysis will only constrain the possible types of `foo` to be a superset of the types of `bar` and `""`, and thereby conclude that the type of `bar` is unconstrained. In a subset-based analysis with both upper-bound and lower-bound constraint variables, the example becomes more complicated, but the result remains the same. This shows that changing from unification-based to subset-based analysis does not necessarily improve the precision of the type inference. We investigate this experimentally in Section 5.7.

The Emitting Phase

The last phase of TSINFER uses the results of the preceding phases to emit a declaration for the library. A declaration can be seen as a tree structure that resembles the heap snapshot, so we create the declaration by traversing the heap snapshot and converting the JavaScript values to TypeScript types, using the results from the static analysis when a function is encountered.

Implementing this phase is conceptually straightforward, although it does involve some technical complications, for example, handling cycles in

the heap snapshot and how to combine a set of recursive types into a single type.

5.5 TSevolve: Evolution of Type Declarations

The goal of TSevolve is to create a list of changes between an old and a new version of a JavaScript library. To do this it has access to three input files: the JavaScript files for the old version `old.js` and the new version `new.js` and an existing TypeScript declaration file for the old version `old.d.ts`.

To find the needed changes for the declaration file, a naive first approach would be to compare `old.d.ts` with the output of running TSinfer on `new.js`. However, this will result in a lot of spurious warnings, both due to imprecisions in the analysis of `new.js`, but also because of intentional discrepancies in `old.d.ts`, as discussed in Section 5.3.

Instead we choose a less obvious approach, where TSevolve uses TSinfer to generate declarations for both `old.js` and `new.js`. These declarations are then traversed as trees, and any location where the two disagree is marked as a change. The output of this process will still contain spurious changes, but unchanged features in the implementation should rarely appear as changes, as imprecisions in unchanged features are likely the same in both versions. We then use `old.d.ts` to filter out the changes that concern features that are not declared in `old.d.ts`, which removes many of the remaining spurious changes. Relevant function sources code from `old.js` and `new.js` are also printed as part of the output, which allows for easy manual identification of many of the remaining spurious changes. As the analysis does not have perfect precision, it is necessary to manually inspect and potentially adjust the suggested changes before modifying the declaration file.

As an extra feature, in case a partially updated declaration file for the new version is available, TSevolve can use that file to filter out some of the changes that have already been made.

5.6 Implementation

This section is new to this thesis and was not part of the published paper.

The implementation consists of, as also explained in Section 5.4, three parts: 1) Generating a snapshot, 2) type inference, and 3) combining the snapshot and the type inference to create a declaration file. This section will explain in more detail how the first two of these works.

The last phase mostly consists of converting the sets of types found by the type inference into a single printable type. There is nothing conceptually

challenging about this, and the last phase will therefore not be explained further here, although it does take some engineering effort to implement.

5.6.1 Creating snapshots

Section 3.5.2 laid out the motivation for why dynamic analysis recording snapshots can improve the precision and performance of an analysis.

The snapshots in this work are used to allow the type inference to skip analyzing the initialization of the library. The snapshot contains all the objects and methods that the library put on the heap. The snapshot therefore contains the top-level properties and methods from the library. These top-level properties and methods are used, together with the types from the type inference, to create the TypeScript declaration file.

The snapshot collection implementation used in this paper works by instrumenting the library. The instrumented library is executed in order to initialize it, and the resulting heap from the initialization is dumped to a file. Dumping a heap to a file is straightforward: It is a recursive walk through the heap, where each object encountered in the heap is serialized and put into a big JSON array. During the serialization, pointers in the heap are converted into indices in the JSON array, and the serialized objects furthermore contain meta-information recorded by the instrumentation.

It is easy to dump the heap without doing any instrumentation beforehand. However, such a heap dump would not contain enough information about functions. For functions, it is challenging to record which function in the program text the function corresponds to, and which environment variables are in the environment of the function. Both of these are important for the type inference. Consider as an example the following library:

```
261 function f(x) {
262     return function(y) {
263         return x + y;
264     }
265 }
266 var g = f(5)
```

We wish to infer a type for the `g` function in the above program. It is impossible to infer a type unless we know which function in the program text the function corresponds to. Therefore the instrumentation will add, to every function allocated in the program, a descriptor of which function in the program text the function corresponds to.

The next problem when inferring a type for `g` is the `x` variable inside the function. The value of `x` is always the number `5` from line 266. However,

the captured value of x is not readable from a naively created snapshot. The program is therefore instrumented such that all environment variables are explicitly stored in the snapshot. The following program demonstrates how the environment is made visible. The program is the same as shown previously; however, the program has been modified such that the environments are visible when recording the heap snapshot.

```
267 function f(x) {
268     var env1 = {}
269     env1.x = x;
270     var inner = function(y) {
271         var env2 = { env: env1 }
272         env2.y = y;
273         return env1.x + env2.y;
274     }
275     inner.env = env1;
276     return inner;
277 }
278 var g = f(5)
```

The program above behaves the same way as the previous program. However, all variable accesses inside the functions are performed using explicitly created environment objects, which are stored on the associated functions.

When the g function is serialized, the env property will be included in the serialization in order to include the environment of g.

The above is still a simplified explanation of how the instrumentation works. The full implementation is more resilient and handles more corner-cases. However, these implementation details are not necessary for understanding what the analysis does, and thus we refer to the implementation of the instrumentation for the full details.[7]

5.6.2 The Type Inference

Subtyping based type inference was described in Section 3.1. This section will describe how the implementation of the type inference in this paper relates to subtyping based type inference.

The type inference in this paper attempts to infer types for libraries, which are open programs that are only meant to be executed along with some client. Because the library is an open program, there exist many methods in the library where the method is never called inside the library itself.

[7] https://github.com/webbiesdk/jsnap

Subtyping based type inference will, for a variable `x`, find a type $[\![\texttt{x}]\!]$ for the variable such that $[\![\texttt{x}]\!]^{\uparrow} \leq [\![\texttt{x}]\!] \leq [\![\texttt{x}]\!]^{\downarrow}$. This approach works well when for automatically inferring some machine-readable types for a whole statically typed program. However, the approach described in this paper attempts to find human-readable types for JavaScript libraries, for which we a traditional subtyping based approach will be unable to infer any type at all, due to the complex patterns used in JavaScript libraries.

A problem with using subtyping based type inference for inferring human-readable types for JavaScript libraries is that for some variables the upper bound type will be $\bot$, which can happen when a method in a library is not called within the library itself. For other variables, the lower bound type will be $\top$, which can happen with setter methods in the library.

Having the upper bound type as $\bot$ or the lower bound type as $\top$ produces terrible human-readable types. The type analysis in this paper therefore uses the inferred lower bound and upper bound types more as hints. This is achieved by replacing the subtype constraints from subtyping based type inference with subset constraints. Consider the following program for an example of how the subset based constraints work.

```
279 var x = 2;
280 var y = x;
281 var z = y - 2;
```

The above program gives rise to the following constraints. The constraints are exactly similar to the subtype based constraints, except for using subset operators instead of subtype operators. The $\equiv$ symbol is used as syntactic sugar for the two operands being subsets of each other, thereby being forced to be the same set. Curly brackets `{}` are used to denote sets.

- $[\![\texttt{2}]\!] \equiv \{\texttt{number}\}$
- $[\![\texttt{2}]\!] \subseteq [\![\texttt{x}]\!]^{\uparrow}$
- $[\![\texttt{x}]\!]^{\uparrow} \subseteq [\![\texttt{y}]\!]^{\uparrow}$
- $[\![\texttt{y-2}]\!] \equiv \{\texttt{number}\}$
- $[\![\texttt{y-2}]\!] \subseteq [\![\texttt{z}]\!]^{\uparrow}$
- $\{\texttt{number}\} \subseteq [\![\texttt{y}]\!]^{\downarrow}$
- $[\![\texttt{y}]\!]^{\downarrow} \subseteq [\![\texttt{x}]\!]^{\downarrow}$
- $[\![\texttt{x}]\!]^{\downarrow} \subseteq [\![\texttt{2}]\!]$

The upper bound and lower bound types are used more as hints to what the final type is. Therefore the type of a variable `x` is not found as a type that satisfies $[\![\texttt{x}]\!]^{\uparrow} \leq [\![\texttt{x}]\!] \leq [\![\texttt{x}]\!]^{\downarrow}$. With our subset based constraints, the final type instead satisfies both $[\![\texttt{x}]\!]^{\uparrow} \subseteq [\![\texttt{x}]\!]$ and $[\![\texttt{x}]\!]^{\downarrow} \subseteq [\![\texttt{x}]\!]$. The approach chosen would not work for generating types used by a compiler while optimizing

a program. However, the approach works well for finding human-readable types from JavaScript libraries.

The above constraints have the trivial solution that the type of all the variables is `number`.

Notice the two constraints related to the `y=x` assignment. With subtyping based inference rules the two constraints would be: $[\![\texttt{x}]\!]^{\uparrow} \leq [\![\texttt{y}]\!]^{\uparrow}$ and $[\![\texttt{x}]\!]^{\downarrow} \leq [\![\texttt{y}]\!]^{\downarrow}$. However, with subset based constraints the two constraints are: $[\![\texttt{x}]\!]^{\uparrow} \subseteq [\![\texttt{y}]\!]^{\uparrow}$ and $[\![\texttt{y}]\!]^{\downarrow} \subseteq [\![\texttt{x}]\!]^{\downarrow}$.

Both the subset based and the subtyping based constraints capture the intuition that the upper bound types flow from the definition of a variable to the use of a variable and that the lower bound types flow from the use of a variable to the definition of the variable. However, with subset based constraints, we have to flip the direction of the subset constraint between the constraints for the lower bound types and the constraints for the upper bound types. These flipped constraints are a side effect of how the final type is found with the subset based constraints compared to the subtyping based constraints.

The constraints are otherwise similar to the constraints for subtyping based type inference, and, just as the subtyping based type inference, the upper bound and lower bound types are computed separately.

Various small tricks are used throughout the implementation, which improves the quality of the generated declaration files. For example for an expression such as `a-b`, the only constraints added are that the expresssions `a`, `b`, and `a-b` are numbers. No subset constraint is added between any of the expressions, as such constraints can lead to imprecision spreading through the analysis.

5.7 Experimental Evaluation

Our implementations of TSInfer and TSEvolve, which together contain around 20 000 lines of Java code and 1 000 lines of JavaScript code, are available at `http://www.brics.dk/tstools/`.

We evaluate the tools using the following research questions.

- RQ1: Does the subset-based approach used by TSInfer improve analysis speed and precision compared to the unification-based alternative?
- RQ2: A tool such as TSCheck that only aims to check existing declarations may blindly assume that some parts of the declarations are correct, whereas a tool such as TSInfer must aim to infer complete declarations. For this reason, it is relevant to ask: How much infor-

mation in declarations is blindly ignored by TSCHECK but potentially inferred by TSINFER?

- RQ3: Can TSINFER infer useful declarations for libraries? That is, how accurate is the structure of the declarations and the quality of the types compared to handwritten declarations?

- RQ4: Is TSEVOLVE useful in the process of co-evolving declaration files as the underlying libraries evolve? In particular, does the tool make it possible to correctly update a declaration file in a short amount of time?

We answer these questions by running the tools on randomly selected JavaScript libraries, all of which have more than 5 000 stars on GitHub and a TypeScript declaration file of at least 100 LOC. Our tools do not yet support the `require` function from Node.js,[8] so we exclude Node.js libraries from this evaluation. All experiments have been executed on a Windows 10 laptop with 16GB of RAM and an Intel i7-4712MQ processor running at 1.5GHz.

RQ1 (subset-based vs. unification-based static analysis)

To compare the subset-based and unification-based approaches, we ran TSINFER on 20 libraries. The results can be found in the left half of Table 5.1. The *Funcs* column shows the number of functions analyzed for each library. The *Unification* and *Subset* columns show the analysis time for the unification-based and subset-based analysis, respectively, using a timeout of 30 minutes.

The results show that our subset-based analysis is significantly faster than the unification-based approach. This is perhaps counterintuitive for readers familiar with Andersen-style [19] (subset-based) and Steengaard-style [106] (unification-based) pointer analysis for e.g. C or Java. However, it has been observed before for JavaScript, where the call graph is usually inferred as part of the analysis, that increased precision often boosts performance [20, 105].

We compared the precision of the two approaches by their ability to infer function signatures on the libraries where the unification-based approach does not reach a timeout. Determining which of two machine generated function signatures is the most precise is difficult to do objectively, so we randomly sampled some of the function signatures and manually determined their precision. To minimize bias, each pair of generated function signatures was shown randomly.

[8] `https://nodejs.org/`

Table 5.1: Analysis speed and precision.

	Speed			*Precision*			
Library	*Funcs*	*Unification*	*Subset*	*Unification*	*Subset*	*Equal*	*Unclear*
Ace	1 249	`timeout`	13.8s	-	-	-	-
AngularJS	609	193.3s	7.8s	1	14	17	0
async	169	28.2s	4.9s	2	22	20	6
Backbone.js	176	28.7s	4.8s	1	9	44	0
D3.js	1 030	181.7s	15.8s	4	19	44	2
Ember.js	2 902	`timeout`	319.7s	-	-	-	-
Fabric.js	1 032	`timeout`	15.7s	-	-	-	-
Hammer.js	122	32.5s	3.2s	0	2	61	3
Handlebars.js	280	9.2s	6.9s	0	3	12	1
Jasmine	51	135.4s	4.6s	2	4	71	0
jQuery	500	`timeout`	41.2s	-	-	-	-
Knockout	325	168.8s	14.4s	2	7	41	8
Leaflet	758	`timeout`	11.6s	-	-	-	-
Moment.js	446	872.6s	12.4s	1	27	21	2
PixiJS	1 527	`timeout`	308.0s	-	-	-	-
Polymer.js	748	424.2s	8.5s	1	10	41	3
React	1 261	`timeout`	14.0s	-	-	-	-
three.js	1 243	`timeout`	208.8s	-	-	-	-
Underscore.js	298	81.2s	4.2s	0	4	47	0
vue.js	433	`timeout`	6.2s	-	-	-	-
Total	15 159	-	1 026.5s	14	121	419	25

The results from these tests are shown in the right half of Table 5.1 where the function signatures have been grouped into four categories: *Unification* (the unification-based analysis inferred the most precise signature), *Subset* (the subset-based analysis was the most precise), *Equal* (the two approaches were equally precise), and *Unclear* (no clear winner). The results show that the subset-based approach in general infers better types than the unification-based approach. The unification-based did in some cases infer the best type, which is due to the fact that a more precise analysis does not necessarily result in a more precise type inference, as explained in Section 5.4.

RQ2 (information ignored by TScheck but considered by TSinfer)

TSCHECK only checks the return types of the functions where the corresponding signature in the declaration file do not have a `void`/`any` return type, which may detect many errors, but the rest of the declaration file is blindly assumed to be correct. In contrast, TSINFER infers types for all functions, including their parameters, and it also infers classes and fields.

Table 5.2 gives an indication of the amount of extra information that TSINFER can reason about compared to TSCHECK. For each library, we show the number of functions that have return type `void` or `any` (and in parentheses the total number of functions), and the number of parameters, classes, and fields, respectively. The numbers are based on the existing handwritten declaration files.

We see that on the 20 benchmarks, TSCHECK ignores 1 714 of the 4 224

Table 5.2: Features in handwritten declaration files ignored by TSCHECK but taken into account by TSINFER.

Library	`void`/`any` *functions (all)*	*Parameters*	*Classes*	*Fields*
Ace	301 (460)	370	2	4
AngularJS	8 (26)	39	0	0
async	64 (80)	222	0	0
Backbone.js	67 (149)	210	7	31
D3.js	7 (219)	271	5	12
Ember.js	270 (629)	991	58	103
Fabric.js	93 (330)	382	25	17
Hammer.js	33 (53)	53	16	24
Handlebars.js	20 (20)	19	1	0
Jasmine	1 (1)	1	1	0
jQuery	19 (53)	88	1	0
Knockout	68 (125)	226	6	0
Leaflet	48 (325)	435	26	17
Moment.js	0 (70)	71	0	0
PixiJS	338 (522)	639	86	584
Polymer.js	3 (4)	3	0	0
React	3 (21)	30	1	4
three.js	328 (993)	1295	180	632
Underscore.js	36 (121)	241	0	0
vue.js	7 (23)	42	1	8
Total	1714 (4224)	5628	416	1436

functions, silently assumes 5 628 parameter types to be correct, and ignores 1 436 instance fields spread over 416 classes. In contrast TSINFER, and thereby also TSEVOLVE, does consider all these kinds of information.

RQ3 (usefulness of TSinfer)

As mentioned in Section 5.3, TSCHECK is effective for checking declarations, but not for inferring them. We are not aware of any other existing tool that could be considered as an alternative to TSINFER. To evaluate the usefulness of TSINFER, we therefore evaluate against existing handwritten declaration files, knowing that these contain imprecise information.

We first investigate the ability of TSINFER to identify classes, modules, instance fields, methods, and module functions (but without considering inheritance relationships between the classes and types of the fields, methods, and functions). These features form a hierarchy in a declaration file. For example, `PIXI.Matrix.invert` identifies the `invert` method in the `Matrix` class in the `PIXI` module of *PixiJS*. When comparing the inferred features with the ones in the handwritten declaration files, a true positive (*TP*) is one that appears in both, a false positive (*FP*) exists only in the inferred declaration, and a false negative (*FN*) exists only in the handwritten declaration. In case of *FP* or *FN* we exclude the sub-features from the counts. The quality of the types of the fields and methods is investigated later in this section; for now we only consider their existence.

Table 5.3: Precision of inferring various features of a declaration file.

	Classes			*Modules*			*Class fields*			*Class methods*			*Module functions*		
Library	*TP*	*FP*	*FN*	*TP*	*FP*	*FN*	*TP*	*FP*	*FN*	*TP*	*FP*	*FN*	*TP*	*FP*	*FN*
Ace	0	2	0	1	0	1	0	0	0	0	0	0	3	2	0
AngularJS	0	0	0	2	1	0	0	0	0	0	0	0	22	2	4
async	0	0	0	1	1	0	0	0	0	0	0	0	88	6	0
Backbone.js	5	0	2	1	1	0	18	3	12	183	8	3	12	10	2
D3.js	5	13	0	1	9	9	12	4	0	15	4	2	56	247	12
Ember.js	62	64	54	16	32	7	8	187	35	40	54	74	333	678	112
Fabric.js	25	21	0	7	3	1	16	193	1	248	402	8	165	24	3
Hammer.js	8	8	7	2	0	1	7	64	0	39	6	0	16	9	9
Handlebars.js	2	4	0	4	3	2	0	3	0	20	4	0	28	8	3
Jasmine	2	22	0	1	4	0	0	0	0	0	8	0	28	33	3
jQuery	2	6	0	4	29	2	0	6	0	0	6	0	90	59	6
Knockout	5	3	1	14	11	1	0	4	0	14	3	0	91	63	2
Leaflet	33	10	0	22	21	1	5	75	12	241	248	2	137	135	1
Moment.js	0	2	0	1	0	0	0	0	0	0	0	0	89	25	6
PixiJS	70	2	16	31	8	2	812	46	52	450	37	7	128	14	16
Polymer.js	0	2	0	1	19	0	0	0	0	0	0	0	2	9	0
React	1	0	0	4	3	0	3	7	1	2	0	1	26	6	130
three.js	169	12	11	12	18	0	2 348	71	33	907	105	24	241	26	8
Underscore.js	0	1	0	1	0	0	0	0	0	0	0	0	117	1	3
vue.js	1	1	0	2	4	0	8	22	0	23	21	0	12	1	1
Total	390	173	91	128	167	27	3 237	685	146	2 182	906	121	1684	1358	321
Precision/Recall	*Prec*: 69.3% *Rec*: 80.9%			*Prec*: 43.4% *Rec*: 82.6%			*Prec*: 82.5% *Rec*: 95.7%			*Prec*: 70.7% *Rec*: 94.8%			*Prec*: 55.36% *Rec*: 84.0%		

The counts are shown in Table 5.3, together with the resulting precision (*Prec*) and recall (*Rec*). We see that TSinfer successfully infers most of the structure of the declaration files, although some manual post-processing is evidently necessary. For example, 80.9% of the classes and 95.7% of the fields are found by TSinfer. Having false positives in an inferred declaration (i.e., low precision) is less problematic than false negatives (i.e., low recall): it is usually easier to manually filter away extra unneeded information than adding information that is missing in the automatically generated declarations.

The identification of classes, modules, methods, and module functions in TSinfer is based entirely on the snapshots (Section 5.4), so one might expect 100% precision for those counts. (Identification of fields is partly also based on the static analysis.) The main reason for the non-optimal precision is that many features are undocumented in the manually written declarations. By manually inspecting these cases, we find that most of these are likely intentional: although they are technically exposed to the applications, the features are meant for internal use in the libraries and not for use by applications. Non-optimal recall is often caused by intentional discrepancies as discussed in Section 5.3 or by libraries that violate our assumption explained in Section 5.4 about the API being fully established after the initialization code has finished. Other reasons for non-optimal precision or recall are simply that the handwritten declaration files contain errors or, in cases where the version number is not clearly stated in declaration file, we were unable to correctly determine which library version it is supposed to match.

To measure the quality of the inferred types of fields and methods, we again used the handwritten declaration files as gold standard and this time manually compared the types, in places where the inferred and handwritten declaration files agreed about the existence of a field or method. Such a comparison requires some manual work, so we settled for sampling: for each library, we compared 50 fields and 100 methods (thereof 50 that were classified as constructors), or fewer if not that many were found in the library.

The result of this comparison can be seen in Table 5.4 where *Perfect* means that the inferred and handwritten type are identical, *Good* means that the inferred type is better than having nothing, *Any* means that the main reason for the sample not being perfect is that either the inferred or the handwritten type is `any`, *Bad* means that the inferred type is far from correct, and *No params* means that the inferred type has no parameters while the handwritten does. Obviously, this categorization to some extent relies on human judgement, but we believe it nevertheless gives an indication of the quality

Table 5.4: Measuring the quality of inferred types of fields and methods.

	Class fields				*Class methods and module functions*				
Library	*Perfect*	*Good*	*Any*	*Bad*	*Perfect*	*Good*	*Any*	*Bad*	*No params*
Ace	0	0	0	0	0	3	0	0	0
AngularJS	0	0	0	0	10	10	2	0	0
async	0	0	0	0	0	26	18	0	6
Backbone.js	14	2	2	0	12	6	30	0	7
D3.js	3	0	9	0	11	36	5	2	1
Ember.js	3	3	2	0	42	37	11	5	5
Fabric.js	13	0	3	0	22	18	10	3	22
Hammer.js	0	0	1	0	7	17	9	0	8
Handlebars.js	0	0	0	0	6	22	9	2	7
Jasmine	0	0	0	0	1	12	6	0	9
jQuery	0	0	0	0	5	21	20	1	0
Knockout	0	0	0	0	5	25	24	0	1
Leaflet	3	2	0	0	14	36	7	0	19
Moment.js	0	0	0	0	8	15	21	0	6
PixiJS	32	5	13	0	38	40	21	1	0
Polymer	0	0	0	0	1	1	2	0	0
React	2	0	1	0	0	32	5	0	0
three.js	37	3	10	0	44	46	10	0	0
Underscore.js	0	0	0	0	0	11	35	3	1
vue.js	2	0	6	0	6	15	2	1	0
Total	109	15	47	0	232	429	247	18	92

of the inferred types. An example in the *Good* category is in *PixiJS* where TSInfer infers a perfect type for the `PIXI.Matrix().applyInverse` method, except for the first argument where it infers the type `{x: number, y: number}` instead of the correct `PIXI.Point`.

As can be seen in Table 5.4, the types inferred for fields are perfect in most cases, and none of them are categorized as *Bad*. The story is more mixed for method types. Here, there are relatively fewer perfect types, but function signatures are also much more complex, given that they often contain multiple parameters as well as a return type, and parameters can sometimes be extremely difficult to infer correctly. For many method types categorized as *Good*, the overall structure of the inferred type is correct but some spurious types appear in type unions for some of the parameters or the return type, or, as in the example with `applyInverse`, an object type is inferred whose properties is a subset of the properties in the handwritten type. The main reason that some method types are categorized as *No params* is that our analysis is unable to reason precisely about the built-in function `Function.prototype.apply` and the `arguments` object. We leave it as future work to explore more precise abstractions of these features.

RQ4 (usefulness of TSevolve)

To evaluate if TSevolve can assist in evolving declaration files, we performed a case study where TSevolve was used for updating declaration

Table 5.5: Classification of TSevolve output.

Library	*TP*	*FP*	*FP**	*Unclear*
async 1.4 → 2.0	38	0	52	2
Backbone.js 1.0 → 1.3	34	0	42	2
Ember.js 1.13 → 2.0	55	24	40	0
Ember.js 2.0 → 2.7	44	0	54	0
Handlebars.js 3 → 4	37	3	8	59
Moment.js 2.11 → 2.14	10	0	54	2
PixiJS 3 → 4	270	13	41	2
Total	488	40	291	67

files in 7 different evolution scenarios. In each case, we used the output from TSevolve to make a pull request to the relevant repository. All of these libraries have more than 10 000 stars on GitHub and had a need for the declaration file to be updated, but were otherwise randomly selected. We had no prior experience in using any of the libraries.

The output from TSevolve is a list of changes for each declaration file. We took the output lists from each of the 7 updates and classified each entry in each list based upon how useful it was in the process of evolving the specific library.

The result of this can be seen in Table 5.5 where each change listed by TSevolve is counted in one of the four columns. *TP* counts true positives, i.e. changes that reflect an actual change in the library that should be reflected in the declaration file. Both *FP* and *FP** count false positives, the difference being that changes counted in *FP** could easily be identified as spurious by looking at the output from TSevolve, as explained in Section 5.5. *Unclear* counts the listed changes that could not be easily categorized.

In the update from *Ember.js* version 1.13 to version 2.0, all of the 24 in the *Bad* category are due to *Ember.js* breaking our assumption about the API being fully established after the top-level code has executed. None of the other libraries violate that assumption.

In the update of *Handlebars.js* from version 3 to 4, all the 59 in the *Unclear* category are due to the structures of the handwritten and the inferred declaration files being substantially different. TSevolve is therefore not able to automatically filter out undocumented features, and all 59 entries are therefore filtered out manually.

From Table 5.5 we can see that the output from TSevolve mostly points out changes that should be reflected in the corresponding declaration file. Among the spuriously reported changes, most of them can easily be identified as being spurious and are therefore not a big problem.

Table 5.6: Pull requests sent based in TSevolve output.[9]

Library	*Lines added*	*Lines removed*	*Library author response*
async 1.4 → 2.0	46	13	"pretty thorough and seems to follow the 2.x API much better than what we currently have"
Backbone.js 1.0 → 1.3	27	3	
Ember.js 1.13→2.0	8	508	"LGTM[10]👍"
Ember.js 2.0→2.7	96	92	"👍"
Handlebars.js 3 → 4	49	2	
Moment.js 2.11 → 2.14	4	0	"thank you, looks good"
PixiJS 3 → 4 (pre-release)	158	261	"Awesome PR"
PixiJS 3 → 4	19	4	"I went through all of your changes and can confirm everything is perfect"

These outputs of TSevolve were used to create pull requests, which are described in Table 5.6. For each pull request, we show how many lines the pull request added and removed in the declaration file,[11] along with a response from a library developer, if one was given. For *Handlebars.js*, the pull request additionally contains a few corrections of errors in the declaration file that were spotted while reviewing the report from TSinfer. All 7 pull requests were accepted without any modifications to the changes derived from the TSevolve output.

The total working time spent going from TSevolve output to finished pull requests was approximately one day, despite having no prior experience using any of the libraries. Without tool support, creating such pull requests, involving a total of 407 lines added and 883 lines removed, for libraries that contain a total of 129 365 lines of JavaScript code across versions and declaration files containing 3 938 lines (after the updates), clearly could not have been done in the same amount of time.

[9] The pull requests: `https://gist.github.com/webbiesdk/f82c135fc5f67b0c7f175e985dd0c889`

[10] An acronym for "Looks Good To Me".

[11] The complete pull requests in some cases contain more lines changed, due to minor refactorings or copying and renaming of files to match the version numbers.

5.8 Related Work

The new tools TSinfer and TSevolve build on the previous work on TScheck [47], as explained in detail in the preceding sections. Other research on TypeScript includes formalization and variations of its type system [26, 91, 94, 111], and several alternative techniques for JavaScript type inference exist [31, 72, 90], however, none of that work addresses the challenges that arise when integrating JavaScript libraries into typed application code.

The need for co-evolving declaration files as the underlying libraries evolve can be viewed as a variant of collateral evolution [83]. By using our tools to increase confidence that the declaration files are consistent with the libraries, the TypeScript type checker becomes more helpful when developers upgrade applications to use new versions of libraries.

Our approach to analyze the JavaScript libraries differs from most existing dataflow and type analysis tools for JavaScript, such as, TAJS [20, 57] and SAFE [25], which are whole-program analyzers and not sufficiently scalable and precise for typical JavaScript library code. We circumvent those limitations by concretely executing the library initialization code and using a subset-based analysis that is inspired by Pottier [88], Rastogi et al. [91], and Chandra et al. [31].

Other languages, such as typed dialects of Python [71, 112], Scheme [109], Clojure [28], Ruby [77], and Flow for JavaScript[44], have similar challenges with types and cross-language library interoperability, though not (yet) at the same scale as TypeScript. Although TSinfer and TSevolve are designed specifically for TypeScript, we believe our solutions may be more broadly applicable.

5.9 Conclusion

We have presented the tools TSinfer and TSevolve and demonstrated how they can help programmers create and maintain TypeScript declaration files. By making the tools publicly available, we hope that the general quality of declaration files will improve, and that further use of the tools will provide opportunities for fine-tuning the analyses towards the intentional discrepancies found in real-world declarations.

Acknowledgments This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647544).

Chapter 6

Type Test Scripts for TypeScript Testing

By Erik Krogh Kristensen and Anders Møller. Published in Proceedings of the ACM on Programming Languages Volume 1 Issue OOPSLA, October 2017.

6.1 Abstract

TypeScript applications often use untyped JavaScript libraries. To support static type checking of such applications, the typed APIs of the libraries are expressed as separate declaration files. This raises the challenge of checking that the declaration files are correct with respect to the library implementations. Previous work has shown that mismatches are frequent and cause TypeScript's type checker to misguide the programmers by rejecting correct applications and accepting incorrect ones.

This paper shows how feedback-directed random testing, which is an automated testing technique that has mostly been used for testing Java libraries, can be adapted to effectively detect such type mismatches. Given a JavaScript library with a TypeScript declaration file, our tool TSTEST generates a *type test script*, which is an application that interacts with the library and tests that it behaves according to the type declarations. Compared to alternative solutions that involve static analysis, this approach finds significantly more mismatches in a large collection of real-world JavaScript libraries with TypeScript declaration files, and with fewer false positives. It also has the advantage that reported mismatches are easily reproducible with concrete executions, which aids diagnosis and debugging.

6.2 Introduction

The TypeScript programming language [79] is an extension of JavaScript with optional type annotations, which enables static type checking and other forms of type-directed IDE support. To facilitate use of untyped JavaScript libraries in TypeScript applications, the typed API of a JavaScript library can be described in a TypeScript declaration file. A public repository of more than 3000 such declaration files exists[1] and is an important part of the TypeScript ecosystem.

These declaration files are, however, written and maintained manually, which leads to many errors. The TypeScript type checker blindly trusts the declaration files, without any static or dynamic checking of the library code. Previous work has addressed this problem by automatically checking for mismatches between the declaration file and the implementation [47], and assisting in the creation of declaration files and in updating the declarations as the JavaScript implementations evolve [67]. However, those techniques rely on unsound static analysis and consequently often overlook errors and report spurious warnings.

Gradual typing [102] provides another approach to find this kind of type errors. Type annotations are ignored in ordinary TypeScript program execution, but with gradual typing, runtime type checks are performed at the boundaries between dynamically and statically typed code [91]. This can be adapted to check TypeScript declaration files [117] by wrapping the JavaScript implementation in a higher-order contract [64], which is then tested by executing application code against the wrapped JavaScript implementation. That approach has a significant performance overhead and is therefore unlikely to be used in production. For use in a development setting, a type error can only be found if a test case provokes it. The results by Williams et al. [117] also question whether it is feasible to implement a higher-order contract system that guarantees non-interference.

In this work we present a new method for detecting mismatches between JavaScript libraries and their TypeScript declaration files. Compared to the approaches by Feldthaus and Møller [47] and Kristensen and Møller [67] that rely on static analysis, our method finds more actual errors and also reports fewer false positives. It additionally has the advantage that each reported mismatch is witnessed by a concrete execution, which aids diagnosis and debugging. In contrast to the approach by Williams et al. [117], our method does not require existing test cases, and it avoids the performance overhead and interference problems of higher-order contract systems for JavaScript.

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

Our method is based on the idea of *feedback-directed random testing* as pioneered by the Randoop tool by Pacheco et al. [82]. With Randoop, a (Java) library is tested automatically by using the methods of the library itself to produce values, which are then fed back as parameters to other methods in the library. The properties being tested in Randoop are determined by user-provided contracts that are checked after each method invocation. In this way, method call sequences that violate the contracts are detected, whereas sequences that exhibit acceptable behavior are used for driving the further exploration. Adapting that technique to our setting is not trivial, however. Randoop heavily relies on Java's type system, which uses nominal typing, and does not support reflection, whereas TypeScript has structural typing and the libraries often use reflection. Moreover, higher-order functions in TypeScript, generic types, and the fact that the type system of TypeScript is unsound cause further complications.

Our tool TSTEST takes as input a JavaScript library and a corresponding TypeScript declaration file. It then builds a *type test script*, which is a JavaScript program that exercises the library, inspired by the Randoop approach, using the type declarations as contracts that are checked after each invocation of a library method. As in gradual typing, the type test scripts thus perform runtime type checking at the boundary between typed code (TypeScript applications) and untyped code (JavaScript libraries), and additionally, they automatically exercise the library code by mimicking the behavior of potential applications.

In summary, our contributions are the following.

- We demonstrate that type test scripts provide a viable approach to detect mismatches between JavaScript libraries and their TypeScript declaration files, using feedback-directed random testing.

- TypeScript has many features, including structural types, higher-order functions, and generics, that are challenging for automated testing. We describe the essential design choices and present our solutions. As part of this, we discuss theoretical properties of our approach, in particular the main reasons for unsoundness (that false positives may occur) and incompleteness (that some mismatches cannot be found by our approach).

- Based on an experimental evaluation of our implementation TSTEST involving 54 real-world libraries, we show that our approach is capable of automatically finding many type mismatches that are unnoticed by alternative approaches. Mismatches are found in 49 of the 54 libraries. A manual investigation of a representative subset of the mismatches

shows that 51% (or 89% if using the non-nullable types feature of TypeScript) indicate actual errors in the type declarations that programmers would want to fix. The experimental evaluation also investigates the pros and cons of the various design choices. In particular, it supports our unconventional choice of using potentially type-incorrect values as feedback.

The paper is structured as follows. Section 6.3 shows two examples that motivate TSTEST. Section 6.4 describes the basic structure of the type test scripts generated by TSTEST, and Section 6.5 explains how to handle the various challenging features of TypeScript. Section 6.6 describes the main theoretical properties, Section 7.7 presents our experimental results, Section 6.8 discusses related work, and Section 6.9 concludes.

6.3 Motivating Examples

We present two examples that illustrate typical mismatches and motivate our approach.

The *PathJS* Library

PathJS[2] is a small JavaScript library used for creating single-page web applications. The implementation consists of just 183 LOC. A TypeScript declaration file describing the library was created in 2015 and has since received a couple of bug fixes. As of now, the declaration file is 38 LOC.[3]

Even though the library is quite simple, the declaration file contains errors. Figures 6.1 and 6.2b contain parts of the declaration file and the implementation, respectively. The `Path.root` method (line 293) can be used by applications to set the variable `Path.routes.root`. According to the type declaration, the parameter `path` of the method should be a string (line 283), which does not match the type of the variable `Path.routes.root` (line 285). By inspecting how the variable is used elsewhere in the program, it is evident that the value should be a string. Thus, a possible consequence of the error is that the TypeScript type checker may misguide the application programmer to access the variable incorrectly.

This error is not found by the existing TypeScript declaration file checker TSCHECK, since it is not able to relate the side effects of a method with a variable. Type systems such as TypeScript or Flow [44] also cannot find the error,

[2] `https://github.com/mtrpcic/pathjs`

[3] `https://github.com/DefinitelyTyped/DefinitelyTyped/blob/354cec620daccfa0ad167ba046651fb5fef69e8a/types/pathjs/index.d.ts`

```
282 declare var Path: {
283     root(path: string): void;
284     routes: {
285         root: IPathRoute,
286     }
287 };
288
289 interface IPathRoute {
290     run():void;
291 }
```

Figure 6.1: A part of the TypeScript declaration.

```
292 var Path = {
293     root: function (path) {
294         Path.routes.root = path;
295     },
296     routes: {
297         root: null
298     }
299 };
```

(b) A part of the JavaScript implementation.

```
300 *** Type error
301   property access: Path.routes.root
302   expected: object
303   observed: string
```

(b) Output from the type test script generated by TSTEST.

Figure 6.2: Motivating example from the *PathJS* library.

because the types only appear in the declaration file, not as annotations in the library implementation.

Our approach instead uses dynamic analysis. The type test script generated by TSTEST automatically detects that invoking the `root` method with a string as argument, as prescribed by the declaration file, and then reading `Path.routes.root` yields a value whose type does not match the declaration file. Figure 6.2b shows the actual output of running the type test script. It describes where a mismatch was found, in this case that an object was expected but a string was observed at a property access type check.

```
function reflect(fn) {
  return initialParams(function (args, rflc) {
    args.push(rest(function (err, cbArgs) {
      if (err) {
        rflc(null, {
          error: err
        });
      } else {
        var value = null;
        if (cbArgs.length === 1) {
          value = cbArgs[0];
        } else if (cbArgs.length > 1) {
          value = cbArgs;
        }
        rflc(null, {
          value: value
        });
      }
    }));
    return fn.apply(this, args);
  });
}
```

(a) A part of the JavaScript implementation.

```
interface AsyncFunction<T, E> {
  (callback:
    (err?: E, result?: T) => void
  ): void;
}
reflect<T, E>(fn: AsyncFunction<T, E>):
  (callback:
    (err: void, result:
       {error?: Error, value?: T}
   ) => void) => void;
```

(b) Declaration of the `reflect` function.

```
*** Type error
  property access:
    async.reflect().[arg1].[arg2].error
  expected: undefined or Error
  observed: object {"_generic":true}
```

(c) Sample output from running the type test script.

Figure 6.3: Motivating example from the *Async* library.

The *Async* Library

The following example is more complex, involving higher-order functions and generics. The *Async*[4] library is a big collection of helper functions for

[4] https://github.com/caolan/async

working with asynchronous functions in JavaScript. It is extremely popular, with more than 20 000 stars on GitHub and over 1.5 million daily downloads through NPM.[5]

One of the functions provided by *Async* is `reflect`. It transforms a given asynchronous function, which returns either an error or a result value, into another asynchronous function, which returns a special value that represents the error or the result value. In the declared type of `reflect` (see Figure 6.3b), the error type for the input function is the generic type `E`, while the error type for the output function is the concrete type `Error`. The implementation (see Figure 6.3a) does not transform the error value in any way but merely passes it from the input function to the output function, so the two error types should be the same.

The type test script generated by TSTEST automatically finds this mismatch. The error report, which can be seen in Figure 6.3c, shows a type error involving the `error` property of an object that was the second argument (`arg2`) in a function that was the first argument (`arg1`) in a function returned by `reflect`. (The actual arguments that were used in the call to `reflect` have been elided.) The value is expected to be `undefined` or an `Error` object, but the observed value is an object where calling `JSON.stringify` results in the shown value. In this example, the observed value is a special marker object used by TSTEST to represent unbound generic types.

Because of the complexity of the library implementation (Figure 6.3a), it is unlikely that any existing static analysis is capable of finding this mismatch. In contrast, TSTEST finds in seconds. Whenever a type test script detects a mismatch, the error may be in the declaration file or in the library implementation. When inspecting the mismatch manually it is usually clear which of the two is at fault. Although the type test script uses randomization, detected type mismatches can usually be reproduced simply by running the script with a fixed random seed, which is useful for understanding and debugging the errors that cause the mismatches.

6.4 Basic Approach

The key idea in our approach is, given a JavaScript library and its TypeScript declaration, to generate a *type test script* that dynamically tests conformance between the library implementation and the type declarations by the use of feedback-directed random testing [82]. This section describes the basics of how this is done in TSTEST.

To test a library, feedback-directed random testing incrementally builds sequences of calls to the library, using values returned from one call as

[5] `https://www.npmjs.com/package/async`

parameters at subsequent calls. In each step, if a call to the library is unsuccessful (in our case, the resulting values do not have the expected types), an error is reported, and the sequence of calls is not extended further. Unlike the Randoop tool from the original work on feedback-directed random testing [82], our tool TSTEST does not directly perform this process but generates a script, called a *type test script*, that is specialized to the declaration file and performs the testing when executed. Generating the script only requires the declaration file, not the library implementation.

The basic structure of the generated type test script is as follows. When executed, it first loads the library implementation and then enters a loop where it repeatedly selects a random test to perform until a timeout is reached. Each test contains a call to a library function. The value being returned is checked to have the right type according to the type declaration, in which case the value is stored for later use, and otherwise an error is reported. The arguments to the library functions can be generated randomly or taken from those produced by the library in previous actions, of course only using values that match the function parameter type declarations. Applications may also interact with libraries by accessing library object properties (such as `Path.routes.root` in Figure 6.2). To simplify the discussion, we can view reading from and writing to object properties as invoking getters and setters, respectively, so such interactions can be treated as special kinds of function calls.

The strategy for choosing which tests to perform and which values to generate greatly affects the quality of the testing. For example, aggressively injecting random (but type correct) values may break internal library invariants and thereby cause false positives, while having too little variety in the random value construction may lead to poor testing coverage and false negatives. Other complications arise from the dynamic nature of the JavaScript language, compared to Java that has been the focus on previous work on feedback-directed random testing. We discuss such challenges and design choices in Section 6.5 and present results from an empirical evaluation in Section 6.7.

Figure 6.5 contains a small example of a type test script generated by TSTEST for a simplified version of the *Async* library (Figure 6.4) that we also discussed in Section 6.3. Figures 6.4a and 6.4b show the declaration file and the implementation, respectively, and Figure 6.5 shows the main code of the type test script (simplified for presentation). This library contains two functions, `memoize` and `unmemoize`, that both take functions as arguments and also return functions. The `memoize` function (lines 350–363) uses JavaScript's meta-programming capabilities to implement function memoization, as its name suggests. The `unmemoize` function (lines 364–366) is overloaded, such

```
341 declare module async {
342   function memoize(
343     fn: Function,
344     hasher?: Function
345   ): Function;
346   function unmemoize(fn: Function):
347     Function;
348 }
```

(a) A snippet of the declaration file for *Async*.

```
349 var async = {
350   memoize: function(fn, hasher) {
351     hasher = hasher ||
352       function (a) {
353         return JSON.stringify(a)};
354     var cache = {};
355     var result = function() {
356       var key = hasher(arguments);
357       return cache[key] ||
358        (cache[key] =
359          fn.apply(this, arguments));
360     };
361     result.unmemoized = fn;
362     return result
363   },
364   unmemoize: function(fn) {
365     return fn.unmemoized || fn;
366   }
367 };
```

(b) A simplified implementation.

Figure 6.4: The type test script generated by TSTEST for a subset of the *Async* library.

that it returns the original function if given a memoized function and otherwise behaves as the identity function.

This example contains no type errors, but it illustrates the use of feedback-directed testing for covering the interesting cases. In this example `unmemoize` is overloaded by the use of the property `unmemoized`, which can only be inferred by exercising the implementation. Note that the TypeScript

```
368 var vals = initializeVals();
369 function makeValue1() {
370   return selectVal(vals[1], vals[2],
371     mkFunction());
372 }
373 var lib = require("./async.js");
374 if (assertType(lib, "async"))
375   vals[0] = lib;
376 while (!timeout()) {
377   try {
378   switch (selectTest()) {
379     case 0: // testing async.unmemoize
380       var result =
381         vals[0].unmemoize(makeValue1());
382       if (assertType(result, "Function"))
383         vals[1] = result;
384       break;
385     case 1: // testing async.memoize
386       var result =
387         vals[0].memoize(makeValue1(),
388           makeValue1());
389       if (assertType(result, "Function"))
390         vals[2] = result;
391       break;
392   } } catch(e) {}
393 }
```

Figure 6.5: The type test script for the declaration in Figure 6.4a.

type system only specifies that `unmemoize` takes a function as argument (line 346). It is important to test both overloaded variants of `unmemoize`, one of which requires a function produced by `memoize`. The generated script first initializes an array named `vals` (line 368) for storing values returned later by the library. It then loads the library, checks that the resulting value matches the type declaration, and stores the object (lines 373–375). The main loop proceeds until a timeout is reached (line 376), which is determined either by the time spent or the number of iterations, according to a user provided configuration. The `selectTest` helper function (line 378) picks a test to run, based on which entries of the `vals` array have been filled in. The `assertType` helper function (lines 374, 382 and 389) checks if the first argument has the type specified by the second argument according to the declaration file. For example, line 374 checks that `lib` is an object and

that its `memoize` and `unmemoize` properties are functions. The declaration file in this simple example contains only one type, `Function`, that we need to generate values for. The function `makeValue1` makes such a value by randomly selecting between the relevant entries in `vals` and a simple dummy function (lines 369–372). Testing `unmemoize` now amounts to invoking it with a base object and an argument of the right type (which are obtained from `vals[0]` and `makeValue1`, respectively), checking the type of the returned value, and storing it for later use (lines 379–384). Testing `memoize` is done similarly. Raising an exception is never a type error in TypeScript, so we wrap all of the tests in a `try-catch`, to allow the execution to continue even if an exception is raised.

After a few iterations, both library functions are tested with different values of the right types as arguments. In particular, the feedback mechanism ensures that `unmemoize` is tested with a function that has been memoized by a preceding call to `memoize`.

6.5 Challenges and Design Choices

As mentioned in the preceding section, it is not obvious what strategy the type test script should use for generating and type checking values (specifically, how the `makeValue`, `selectVal` and `assertType` functions in Figure 6.5 work). The traditional approach to feedback-directed random testing, as in e.g. Randoop, heavily relies on Java's type system and common practice of structuring libraries and application code in Java. For example, in Java, if a class *C* is defined in the library, then the usual way for the application to obtain an object of type *C* is by invoking the class constructor or a library method that returns such an object (assuming that no subclasses of *C* are defined by the application). In contrast, TypeScript uses structural typing, so an object has type *C* if it has the right properties, independently of how the object has been constructed. As an example, the main entry point of the *Chart.js*[6] library takes as an argument a complex object structure (see line 398 in Figure 6.6), which is expected to be constructed entirely by the application. This complex object structure includes primitive values, arrays, and functions, all of which must be constructed by our type test script in order to thoroughly test the library. If the script only constructs primitive values and otherwise relies on the library itself to supply more complex values, then testing *Chart.js* would not even get beyond its main entry point.

JavaScript libraries with TypeScript declarations also often use generic types, callbacks, and reflection, and such features have mostly been ignored

[6] `https://github.com/DefinitelyTyped/DefinitelyTyped/blob/c16f53396ee3cf728364a64582627262eaa92bf0/chart.js/index.d.ts`

```
394 declare class Chart {
395   constructor(
396     context: string | JQuery | ..
397     options:
398       Chart.ChartConfiguration
399   );
400   ...
401 }
402
403 interface ChartConfiguration {
404   type?: ChartType | string;
405   data?: ChartData;
406   options?: ChartOptions;
407 }

408 interface ChartOptions {
409   events?: string[];
410   onClick?: (any?: any) => any;
411   title?: ChartTitleOptions;
412   legend?: ChartLegendOptions;
413   ...
414 }
415
416 interface ChartTitleOptions {
417   fontSize?: number;
418   fontFamily?: string;
419   fontColor?: ChartColor;
420   ...
421 }
```

Figure 6.6: An example of structural typing in *Chart.js*.

in previous work on feedback-directed random testing for Java. Furthermore, Java provides strong encapsulation properties, such as private fields and private classes, whereas JavaScript libraries often do not have a clear separation between public and private. Evidently, adapting the ideas of feedback-directed testing to TypeScript involves many interesting design choices and requires novel solutions, which we discuss in the remainder of this section.

Structural Types

As argued above, the type test script needs to generate values that match a given structural type, as supplement to the values obtained from the library itself. The `selectVal` helper function (Figure 6.5) picks randomly (50/50) between these two sources of values. In TypeScript, types can be declared with `interface` or `class` (see examples in Figure 6.6), one difference being that objects created as class instances use JavaScript's prototype mechanism to mimic inheritance at runtime. TypeScript's type system uses structural typing for both interface and class types, but some libraries rely on the prototype mechanism at runtime via `instanceof` checks. For this reason, for function arguments with a class type, we do not generate random values but only use previously returned values from the library. Likewise, base objects at method calls are only taken from values originating from the library (see lines 381 and 388), and are never generated randomly, since we need the actual methods of the library implementation for the testing.

The `makeValue` functions generate random values according to the types in the declaration file. For primitive types, e.g. `booleans`, `strings`, and `numbers`, it is trivial to generate values (the details are not important; for example, random strings are generated in increasing length with exponentially decreasing probability). We assume that the non-nullable types feature of TypeScript 2.0 is enabled, so a value of type e.g. `number` cannot be null, and `null` becomes a primitive type with a single value. For object types, we randomly either generate a new object with properties according to the type declaration or reuse a previously generated one. In this way, the resulting object structures may contain aliases, and, if the types are recursive, also loops. The generated object structures therefore resemble the memory graphs of CUTE [99]. Creation of values for function types is explained in Section 6.5.

Structural typing also affects how `assertType` performs the type checking. When checking that a value v returned from the library has the expected type, ideally all objects reachable from v should be checked. However, perhaps counterintuitively, it is sometimes better to perform a more shallow check. For an example, consider the following declaration, which is a simplified version of the declaration file for the *Handlebars*[7] library.

```
422 declare module Handlebars {
423     function K(): void;
424     function parse(input: string): hbs.AST.Program;
425     function compile(input: any, options?: CompileOptions):
426         HandlebarsTemplateDelegate;
```

[7] https://github.com/DefinitelyTyped/DefinitelyTyped/blob/354cec620daccfa0ad167ba046651fb5fef69e8a/types/handlebars/index.d.ts

```
427 }
```

The corresponding implementation does not have a `K` function on the `Handlebars` module object. This mismatch is immediately detected by the type test script. In traditional feedback-directed random testing, only values that pass the contract check are used for further testing. With that strategy, as the `Handlebars` module object fails the type check, the methods `parse` and `compile` will never be executed as part of the testing because no suitable base object is available. Thus, any additional errors arising from calling these method will remain undetected until the first mismatch is fixed. Some mismatches are introduced intentionally by the programmer, for example to circumvent limitations in the expressiveness of TypeScript's type system, as observed in previous work [47, 67], so it is important that we do not stop testing at the first mismatch we find and require the programmer to fix that mismatch before proceeding. For this reason, we let `assertType` perform the deep type check on all the reachable objects and report a mismatch if the check fails, but the decision whether the value shall be stored for feedback testing is based on a shallow check.[8] In this specific case, invoking `assertType` on the `Handlebars` module object and the type `Handlebars` will trigger a type error message that the `K` property is missing, but `assertType` nevertheless returns `true` (so the object is not discarded) because the value is, after all, an object, not a primitive type. The object is then available for subsequently testing the `parse` and `compile` functions.

Higher-Order Functions

One of the challenges in gradual typing is how to check the types of values at the boundary of typed and untyped code in presence of higher-order functions [102, 103]. In gradual typing, type annotated code (in our case, the TypeScript application code) is type checked statically, whereas untyped code (in our case, the library code) is type checked dynamically. TypeScript itself does not perform any dynamic type checking, which is why our type test scripts need to perform the type checking of the values received from the library code. The challenge with higher-order functions is that the types of functions cannot be checked immediately when they are passed between typed and untyped code, but the type checks must be postponed until the functions are invoked. The blame calculus [115] provides a powerful foundation for tracking function contracts (e.g. types) dynamically and deciding

[8] Since we thereby allow the use of a type incorrect value in subsequent tests, those tests may result in mismatches being reported later in the testing. Such mismatches may seem spurious since they can be reported far from the actual type error, but we find that this situation is rare in practice.

```
var foo = {
  twice: function (x, c) {
    return c(x) + c(x) + "";
  }
}
```

(a) The implementation.

```
export module foo {
  function twice(
    x: number | string,
    c: (s: string) => string
  ): string;
}
```

(b) The declaration.

```
foo.twice(mkStringOrNumber(),
  function(arg) {
    if (assertType(arg, "string"))
      vals[3] = arg;
    return selectVal(
      vals[2],
      vals[3],
      mkString()
    );
});
```

(c) Testing the `twice` function.

```
*** Type error
  argument: foo.twice.[arg2].[arg1]
  expected: string
  observed: number
```

(d) An error reported by the generated type test script.

Figure 6.7: Testing higher-order functions.

whether to blame the library or the application code if violations are detected.

TypeScript applications and JavaScript libraries frequently use higher-order functions, mostly in the form of library functions that take callback functions as arguments. However, we can exploit the fact that *well-typed applications can't be blamed*. In our case, the application code consists of the type test scripts, which are generated automatically from the declaration files and can therefore be assumed to be well typed. (In Section 6.7 we validate that the type test scripts generated by our implementation TSTEST are indeed well typed, in the sense that the construction of values is consistent with the dynamic type checking.) This means that runtime type checks are only needed when the library passes values to the application, not in the other direction. A simple case is when library functions are called from the application, for example when a value is returned from the `memoize` function in the *Async* library (Figure 6.4), but it also happens when the library calls a function that originates from the application, for example when the memoized function is invoked from within the *Async* library (line 359 in Figure 6.4).

When testing a library function whose parameter types contain function types (either directly as a parameter or indirectly as a property of an object

passed to the library function), the type test scripts produced by TSTEST generate dummy callback functions, which accept any arguments and have no side-effects except that they produce a return value of the specified type.

To demonstrate how TSTEST handles higher-order functions, consider the simple library and declaration file in Figures 6.7a and 6.7b, respectively. In this library, the function `twice` takes two arguments, a number or string `x` and a callback function `c`, and then invokes `c(x)` twice and converts the result to a string (line 430). The type declaration of the callback function, however, requires its argument to be a string (line 436). This mismatch is detected by the test shown in Figure 6.7c, which is a part of the type test script generated by TSTEST. A function is constructed (lines 440–448) according to the type declared on line 436. This function first checks if the argument matches the declared type, just like when receiving a value from the library. On line 443 a value satisfying the return type `string` is returned. Thus, the roles of arguments and return values are reversed for callback functions, as usual in contract checking [115]. When running the type test script, the report in Figure 6.7d is produced. Similar to the report from Section 6.4 it pinpoints the type mismatch at the first argument to the callback function, which is the second argument to the `twice` function in module `foo`.

TypeScript supports type-overloaded function constructors, so that the return type of a function can depend on the types of the arguments. Testing an overloaded library function is straightforward; we simply generate a separate test for each signature. The only minor complication is that TypeScript uses a first-match policy, so to avoid false positives, when generating arguments for one signature it is important to avoid values that match the earlier constructors.

Overloaded callback functions are a bit more involved. The callback generated by TSTEST first checks the argument types with respect to each of the constructors in turn. If exactly one of the constructors match, a value of the corresponding return type is produced, as for a non-overloaded callback. If none of the constructors match, our callback must have been invoked with incorrect arguments, so it reports a type error and aborts the ordinary control flow by throwing an exception (corresponding to returning the bottom type). However, if multiple constructors match, we cannot simply pick the first one like TypeScript's static type checker does. The reason is that our runtime type checks are necessarily incomplete in presence of higher-order functions, since function types are not checked until the corresponding functions are invoked, as explained above. If the overloaded callback itself takes a function as argument, we cannot tell simply by inspecting the argument at runtime which overloaded variant applies. For this reason, in case multiple

```
453 declare var c1: Cell<string>;
454 declare var c2: Cell<number>;
455 interface Cell<T> {
456   value: T;
457 }
```

(a) A simple declaration with a generic type.

```
458 interface _Chain<T> {
459    partition(...): _Chain<T[]>;
460    values(): T[];
461    ...
462 }
463 declare var foo: _Chain<boolean>;
```

(b) A recursively defined generic type.

Figure 6.8: Examples for explaining how generic types are handled by TSTEST.

constructors match, we let the constructed callback throw an exception (similar to the case where none of the constructors match, but without reporting a type error). Although the situation is not common in practice, this design choice may cause errors to be missed by the type test script, but it avoids false positives. To solve this without throwing an exception we would need higher-order intersections that delay testing [65].

Generic Types

Generics is an extensively used feature of the TypeScript type system, and TSTEST needs to support this feature to be able to find errors like the one discussed in Section 6.3. Figure 6.8a shows a simple example where the type of the `value` fields depends on the type arguments provided for the type parameter `T`. In this particular case, we add one test case to check that reading `c1.value` yields a string and one to check that `c2.value` yields a number.

Naively adding a test case for every instantiation of the type parameters is not always possible, because generics may be used recursively. As an example, consider the type `_Chain` in Figure 6.8b from the *Underscore.js*[9] library. We test that `foo` yields a value of type `_Chain<boolean>` and that

[9] https://github.com/DefinitelyTyped/DefinitelyTyped/blob/354cec620daccfa0ad167ba046651fb5fef69e8a/types/underscore/index.d.ts#L5046

`foo.values()` returns a value of type `boolean[]`. However, notice that invocations of `foo.partition(...).values()` should return values of type `boolean[][]`, and with additional successive invocations of `partition` we obtain arbitrarily deeply nested array types. For this reason, we choose to restrict the testing of such recursively defined generic types: at the recursive type instantiation, in this case `_Chain<T[]>` (line 459), we treat the type parameter `T` as TypeScript's built-in type `any` that matches any value (see also Section 6.5). Since we then test that the value has type `_Chain<any[]>` rather than `_Chain<boolean[]>` we may miss errors, but (due to TypeScript's covariant generics) we do not introduce false positives. The resulting value, which now has type `_Chain<any[]>`, can be used as base object for further testing the methods of `_Chain`, so the type test script can explore arbitrarily long successive invocations of `partition` with a bounded number of test cases.

Recursively defined generic types affect not only the type checks but also the construction of random values of a given type. To this end, we follow the same principle as above, treating type parameters that are involved in recursion as `any`. For example, if a library function takes a parameter of type `_Chain<boolean>`, we need to generate an object of that type, which ideally would require construction of `partition` functions that return arbitrarily deeply nested array types. By breaking the recursion using `any`, we ensure that it is only necessary to produce random values for a bounded number of different types. The drawback, however, is that this design choice may result in false positives, which we return to in Section 6.6.

TypeScript also supports generic functions. In recent work on TypeScript, Williams et al. [117] have chosen to enforce *parametricity* [114] of generic functions. With their interpretation, a generic function should act identically on its arguments irrespective of their types. As an example, Williams et al. insist that the only total function that matches the following TypeScript declaration is the identity function.

```
464 declare function weird<X>(x: X): X
```

Parametricity is useful in purely functional languages, but we argue it is a poor match with a highly dynamic imperative language like TypeScript where e.g. reflection is commonly used. A simple real-world example, which resembles the `weird` function by Williams et al., is the function `extend` in the *Lodash*[10] library:

```
465 declare function extend<A, B>(obj: A, src: B): A & B;
```

[10] https://github.com/DefinitelyTyped/DefinitelyTyped/blob/354cec620daccfa0ad167ba046651fb5fef69e8a/types/lodash/index.d.ts#L14903

According to its return type, `A & B`, the returned value has both types `A` and `B`. The implementation of `extend` copies all properties from `src` to `obj` and returns `obj`, which then indeed has the type `A & B`, but this implementation would be disallowed if parametricity were enforced. For this reason, we do not treat a generic function as erroneous just because it fails to satisfy parametricity.

Williams et al. [117] type check generic functions using dynamic sealing based on proxy objects. That approach is unsuitable as we do not want to enforce parametricity, and additionally Williams et al. report that it sometimes causes interference with the library implementation. Instead, we choose to test generic functions by the use of simple dummy objects. For example, as arguments to the `extend` function we could provide the two objects `{_generic1:true}` and `{_generic2:true}` (representing objects of type `A` and `B`, respectively), and then test that the returned value matches the object type[11] `{_generic1:true, _generic2:true}` (corresponding to type `A & B`). This approach obviously fits nicely with the `extend` example, but due to TypeScript's structural typing (specifically, its use of width-subtyping), it is sound also more generally to supply arbitrary objects for function parameters with generic types and then test that the returned value is an object with the right properties. If the type parameters have bounds (e.g. `A extends Foo`), we simply use the bound type (`Foo`) augmented with the special `_generic` property.

Using different dummy objects for different type parameters does not always work, though. Consider the following example.

```
466 class Foo<T> {...}
467 function foo<T1>(t: T1): Foo<T1>;
468 function bar<T2>(foo: Foo<T2>): T2;
```

If `T1` and `T2` were instantiated with different concrete types, then a result of calling `foo` could not be used as feedback for a call to `bar`, and thus any error that only happens when `bar` is called with an argument produced by `foo` is missed. In that situation it is better to use only a single object type, `{_generic:true}` (as in the error report in Figure 6.3c), for all the type parameters. In either case, we may miss errors. We choose the latter approach, and we experimentally evaluate whether one approach finds more mismatches than the other (see Section 6.7). Previous work on testing for Java [49] exploits casts and `instanceof` checks in the program under test as hints for generating values for generic types, however, the lack of an explicit cast operation in JavaScript and the common use of structural typing in JavaScript libraries makes that that approach less applicable in our setting.

[11] Object literals like these are also valid types since TypeScript 2.0.

Other Design Choices Involving Types

TypeScript has, as other languages with optional types, the special type `any` that effectively disables static type checking [26]. The type system not only allows any value to be written to a variable or property of type `any`, it also allows any method call or property access on such a variable or property. For a type test script, checking if a value returned from the library is of type `any` is trivial: the answer is always yes. However, when type test scripts need to generate values of type `any` as input to the library, instead of generating arbitrary values, we choose to construct a single special object: `{_any:true}`. This allows the users of TSTEST to easily recognize instances of the `any` type in the output when type mismatches are detected. A possible drawback of this design choice is discussed in Section 6.6.

In TypeScript, static fields in super-classes are inherited by sub-classes. When checking that a value matches a class, we choose to ignore inherited static fields, because some libraries are intentionally not implementing this feature. Libraries that do implement inheritance of static fields normally do so using a special "`createClass`" method, and such central methods are likely thoroughly tested already.

We described in Section 6.5 our motivation for performing only a shallow structural type check when deciding whether a value shall be used for feedback. However, union types are treated differently. If a value with declared type `A | B` is returned from the library, then we can use it as feedback in subsequent tests as a value of type `A` or as a value of type `B`, but only if we can determine which of the two types the value actually has. A shallow type check is sometimes insufficient to make the distinction, so in this situation we use a deep type check, similar to the choice described in Section 6.5 for overloaded function constructors.

6.6 Soundness and (Conditional) Completeness

Type test scripts perform purely dynamic analysis, so obviously they may be able to detect errors but they cannot show absence of errors.[12] Two interesting questions remain, however:

1. Whenever a mismatch between a TypeScript declaration file and its JavaScript implementation is reported by the type test script that is generated by TSTEST, is there necessarily a mismatch in practice? If

[12] Cf. the well-known quote by Dijkstra [42].

this is the case, we say that testing is *sound*.[13] If not, what are the possible reasons for false positive? Furthermore, how does the fact that TypeScript's type system is unsound affect the soundness of type testing?

2. Whether a specific mismatch is detected naturally depends on the random choices made by the type test scripts. But is it the case that for every mismatch, there exist random choices that will lead to the mismatch being revealed? If so, we say that testing is *conditionally complete*.[14] If not, what are the possible reasons for some mismatches being undetectable by the type test scripts generated by TSTEST?

The testing conducted by TSTEST is neither sound nor conditionally complete. There is one cause of unsoundness: as explained in Section 6.5, the way we break recursion in generic types using type `any` may cause type tests to fail even in situations where there is no actual mismatch. Specifically, if the input to a library function involves a type parameter that we treat as `any`, then we may generate invalid values that later trigger a spurious type mismatch. This is mostly a theoretical issue; we have never encountered false positives in practice. We do, however, encounter mismatches that are technically true positives but can be categorized as benign, in the sense that the programmers are likely not willing to fix them. We show a representative example in Section 6.7.

The discussion about soundness of the testing is complicated by the fact that TypeScript's type system is intentionally unsound, which is well documented [26]. It is possible to have a library implementation, an application (e.g. a type test script), and a declaration file where the implementation is correct with respect to the declaration file and the application is well-typed (according to TypeScript's type system) when using the declaration file, yet the application encounters type errors at runtime. We consider such runtime type errors as true positives, because the testing technique is not to blame. Nevertheless, we have not encountered this situation in our experiments.

Regarding the conditional completeness question, there are indeed mismatches that cannot be detected by even the luckiest series of random choices made by the type test scripts. The main reason is that our approach for generating random values for a given type (i.e., the `makeValue` functions) cannot produce all possible values. For example, when generating an object according to an interface type, we do not add properties beyond

[13] As customary in the software testing literature, we use the term *soundness* with respect to errors reported; from a static analysis or verification point of view, this property would be called completeness.

[14] In contrast to "full" completeness, this notion of *conditional completeness* does not require that all errors are found, only that they can be found with the right random choices.

those specified by the interface type. We could of course easily add extra properties to the objects, but doing so randomly without more sophisticated machinery, like dynamic symbolic execution [98], most likely would not make a difference in practice. We also use a single special value for the type `any`, for the reason described in Section 6.5, rather than all possible values. An example of an error that is missed by TSTEST because of that design choice is in the *Sortable*[15] library where a function is declared as taking an argument of type `any` but it crashes unless given a value of a more specific type. (Recall that a function is always allowed to throw exceptions, which is not considered a type error.) Finally, our treatment of static fields and unions, as described in Section 6.5, also cause incompleteness.

Some mismatches would remain undetectable by TSTEST even if the random value generator was capable of producing every possible value of a given type. One reason is that the type test scripts never generate random values for class types or for base objects at method calls, but only use values obtained via the feedback mechanism, as mentioned in Section 6.5. Another reason is that the library implementations may depend on global state, for instance the HTML DOM, which is currently ignored by TSTEST. As an example, for most of the code of *reveal.js*[16] to be executed, the HTML DOM must contain an element with class `reveal`, which TSTEST currently cannot satisfy. An interesting opportunity for future work is to extend TSTEST with, for example, symbolic execution capabilities to increase the testing coverage.

6.7 Experimental Evaluation

In this section we describe our implementation and experimental evaluation of TSTEST.

Implementation

Our implementation of TSTEST contains around 11 000 lines of Java code and 400 lines of JavaScript code, and is available at `http://www.brics.dk/tstools/`. It relies on the TypeScript 2.2 compiler for parsing TypeScript declarations, NodeJS and Selenium WebDriver for running type test scripts in browser and server environments, and Istanbul[17] for measuring coverage on executed code.

TypeScript models the ECMAScript native library and the browser DOM API using a special declaration file named `lib.d.ts`. As program analyz-

[15] `https://github.com/DefinitelyTyped/DefinitelyTyped/blob/354cec620daccfa0ad167ba046651fb5fef69e8a/types/sortablejs/index.d.ts#L154`

[16] `https://github.com/hakimel/reveal.js/`

[17] `https://istanbul.js.org/`

ers, TSTEST needs special models for parts of the standard library. Browsers do not use structural typing for built-in types but require, for example, instances of the interface `HTMLDivElement`, which represent HTML `div` elements, to be constructed by the DOM API—having the right properties is not enough. Instead of using the normal approach described in Section 6.5 when constructing values and performing type checks, TSTEST therefore uses the DOM API functionality for such types.

Errors reported by the type test scripts are easier to diagnose and debug if the execution is deterministic. Achieving completely deterministic behavior in JavaScript is difficult,[18] which is one of the reasons why we have not designed TSTEST to output individual tests that expose the detected mismatches. In TSTEST, most sources of nondeterminism are eliminated by "monkey patching" the standard library, specifically `Date` and `Math.random`, which suffices for our purposes.

The feedback-directed approach used by TSTEST is essential for obtaining suitable library input values. However, sometimes the use of feedback also has negative effects, for example causing the internal state of a library to grow such that the time spent running a single method of the library increases as more and more methods have been executed. Sometimes it even happens that a library method gets stuck in an infinite loop. For these reasons, periodically interrupting and resetting the library state often leads to more mismatches being uncovered within a given time budget. The experiments described below confirm that this pragmatic solution works well.

As part of validating that our implementation works as intended, TSTEST can be run in a special mode where it tests consistency between the construction of random values for a give type (i.e., the `makeValue` functions) and the converse type checks (i.e., `assertType`). When this validation mode is enabled, the generated type test script does not load the actual library implementation, but instead constructs a random value that has the type of the library. This value is then tested instead of the actual library implementation. If the resulting type test script reports any mismatches while running, either the constructed value is not well typed, or the type checking reports errors on a well typed value—both situations indicate errors in our implementation. The unsoundness of TypeScript's type system (discussed in Section 6.6) occasionally causes this approach to report spurious validation failures, but overall it has been helpful in finding bugs during the development of TSTEST and increasing confidence in our experimental results.

[18] The record/replay feature of the Jalangi tool [100] was abandoned for exactly this reason.

Research Questions

We evaluate three main aspects of the approach, each with some sub-questions:

1) Quantitative evaluation How many type mismatches does TSTEST find in real-world TypeScript declarations for JavaScript libraries, and how much time is needed to run the analysis? Furthermore, how do the different design choices discussed in Section 6.5 affect the ability to detect type mismatches? For potential future work it is also interesting to know what coverage is obtained by the automated testing of the library code and the type test scripts?

2) Qualitative evaluation Do the type mismatches detected by TSTEST indicate bugs that developers likely want to fix? In situations where mismatches are classified as benign, what are the typical reasons? Also, are there any false positives?

3) Comparison with alternatives Can TSTEST find errors that are missed by available alternative tools, specifically TSCHECK [47] and TSINFER [67]?

As benchmarks for experiments we use all of the libraries used by Feldthaus and Møller [47] and Kristensen and Møller [67], and 32 other randomly selected popular JavaScript libraries.[19] We exclude libraries that write to the local file system. (Concretely executing such libraries with TSTEST could harm our filesystem; this and similar issues could be circumvented with sandboxing or mocking, and it is therefore only a limitation of our current implementation and not of the general approach.) The resulting 54 JavaScript libraries are listed in Section 6.10.

Quantitative Evaluation

How many type mismatches does TStest find?

TSTEST may report multiple type mismatches that have the same root cause, for example if two methods return the same value. To avoid artificially inflating the number of mismatches found, we count two mismatches as the same if they involve the same property on the same type, even though the mismatches involve different property access paths. That is, mismatches in both `foo().baz` and `bar().baz` are counted as the same if `foo()` and `bar()`

[19] found via `https://www.javascripting.com/?sort=rating`.

Table 6.1: Total number of type mismatches found by TSTEST, for different time budgets and repeated runs.

	Mismatches found			
Timeout	1 run	5 runs	10 runs	20 runs
10 seconds	2 804	4 617	5 464	5 916
1 minute	3 534	5 265	6 180	-
5 minutes	3 478	5 898	-	-

return values of the same type. It is still possible that different mismatches have a common root cause, but it is inevitable that some errors will manifest in multiple mismatches.

To see how many mismatches are found and how long it takes to find them, we ran TSTEST with a timeout of 10 seconds, 1 minute, and 5 minutes. We also ran the type test script 5, 10, and 20 times for some of these timeouts (excluding the longest running ones), to measure the effect of periodically resetting the library state as discussed in Section 6.7. A summary of the results can be found in Table 6.1.

Mismatches were found in 49 of the 54 benchmarks, independently of the timeout and the number of repeated runs. This confirms the results from Feldthaus and Møller [47], Kristensen and Møller [67], and Williams et al. [117] that errors are common, even in declaration files for highly popular libraries. The numbers in Table 6.1 are quite large, and there are likely not around 6 000 unique errors among the 54 libraries tested. A lot of the detected mismatches are different manifestations of the same root cause. However, our manual study (see Section 6.7) shows that some declaration files do contain dozens of actual errors.

The randomness involved in running a type test script means that repeated executions often lead to different sets of type mismatches being reported. From Table 6.1 we see that a substantial number of the mismatches are found already after running each type test script for 10 seconds, and that increasing the duration does not help much. On the other hand, running the type test script multiple times leads to a significant improvement, which validates our claim from Section 6.7 that periodically resetting the library state is beneficial.

How do the various design choices affect the ability to detect type mismatches?

We evaluate the four most interesting design choices discussed in Section 6.5: (1) In Section 6.5 we discussed the use of a shallow type check for determining whether to use a value for feedback. What if a deep type

Table 6.2: Testing different configuration options for the type test scripts.

Configuration	Mismatches found (std. dev.) 1 run	5 runs
Reference configuration	3 280.3 (231.4)	5 181.9 (184.8)
Structural: Using only deeply checked values for feedback	172.8 (42.2)	313.1 (20.7)
Structural: Also generate random values for class types	2 939.9 (230.3)	4 813.7 (156.6)
Generics: Using multiple object types for generic methods	3 212.5 (284.1)	5 217.6 (234.0)
Writing properties: Write to properties of primitive type	3 020.9 (230.8)	5 125.1 (222.0)
Writing properties: Writing to properties of all types	2 299.9 (185.8)	5 217.1 (149.2)

check is used instead? (2) Another design choice in Section 6.5 involved the treatment of class types. What if we also generate random values for class types? (3) In Section 6.5 we discussed the use of a single dummy object type to instantiate unbound generic types. What if we instead use the approach with distinct dummy object types for different type parameters? (4) Type test scripts read object properties and invoke methods of the libraries. What happens if the type test scripts are allowed to also *write* properties of objects, either all properties or only properties of primitive types?

The results from running TSTEST with six different configurations (with 10 seconds timeout and both 1 and 5 runs) can be seen in Table 6.2. The number of mismatches are averages of 30 repetitions of the experiment (with the standard deviation in parentheses). The row 'Reference configuration' is the default setting, and the other five rows correspond to the alternative design choices.

Performing a deep type check for determining whether a value shall be used for feedback testing does result in significantly fewer type mismatches being reported. Looking closer at the mismatches found with that configuration reveals that for 40 of the benchmarks, only *one* mismatch is detected in each (executing the type test script repeatedly did not change this). That single mismatch originates from a core object of the library very early in the execution, which blocks the type test script from further testing. Many of these benchmarks do contain more than one error, so this result confirms that our choice of using the shallow type check is important for finding as many errors as possible.

Generating random values for class types results in slightly fewer mis-

matches compared to the reference configuration, because more time is required to find the same mismatches. We do however find that 25 of our 54 benchmarks use `instanceof` checks on classes defined in the library, and some internal invariants might break if the library is given a value that is structurally correct but fails the `instanceof` check. An example is in the *PixiJS* library where the constructor of the `Polygon` class can take an array of either `number` or `Point`, and the implementation of `Polygon` expects that `instanceof` checks can be used to distinguish between the two. This invariant breaks if the `Polygon` constructor is given an array of objects that are only structurally similar to `Point`, which leads to a benign type mismatch for the return value of the constructor.

We can also conclude that the choice regarding generic methods does not matter much, likely because only few mismatches involve generics. It is easy to construct examples where one approach can find a mismatch and the other cannot, so it seems reasonable to run with both configurations to find as many mismatches as possible. Using a single object type instead of multiple object types for generic methods increases the number of mismatches found, but the ones we inspected were all duplicates.

Allowing the type test scripts to also perform object property write operations does not seem to significantly increase the ability to detect type mismatches. Writing only to properties of primitive type merely increases the amount of time it takes to find the same mismatches. Writing to properties of all types also causes the type test script to take longer to reach the same number of detected mismatches, but it does perhaps find slightly more mismatches in the end. We found four libraries where writing to properties of all types significantly increased the amount of detected mismatches. Investigating some of the mismatches that were only reported when the library was allowed to write properties showed that these mismatches were all caused by the type test script overwriting a core method of the library, thereby introducing another source of false positives that is avoided in the reference configuration.

How much coverage does TStest obtain?

For any automated testing, it is relevant to ask how much of the program code is actually executed. In our case the program code is divided into the type test script and the library being tested.

First, we measured the statement coverage of the libraries. (The Istanbul coverage measurement system was unfortunately unable to instrument all our libraries and type test scripts, so we only have coverage data for 36 of the 54 libraries.) We also measured, across all 54 benchmarks, what percentage of the tests (i.e. cases in the `switch` block; see Figure 6.5) in the type

Table 6.3: Coverage of the library code and the type test scripts.

	initialization only	1 run	5 runs	10 runs	20 runs
Average statement coverage	20.6%	44.4%	48.1%	49.1%	49.8%
Average test coverage	-	57.1%	65.5%	69.1%	69.6%

test script were executed. (Recall from Section 6.4 that the `selectTest` function only chooses between the test cases where values are available for the relevant types.)

The results of these coverage measurements can be seen in Table 6.3. The first row with numbers shows the library statement coverage obtained if only initializing the library, and after 1, 5, 10, and 20 runs of the type test script. Running the type test scripts achieves much higher coverage than only initializing the library, and even after the type test scripts have run many times, there are still uncovered statements that could potentially be reached by running the scripts again. The statement coverage differs significantly between the libraries: from 6.2% to 94.0% (these numbers do not change by running the type test scripts multiple times). For many libraries, large parts of their code is never executed. The reasons for low statement coverage are highly individual, however the most common reason seems to be that the type test script is incomplete in modeling realistic application behavior. A good example is the library with 6.2% statement coverage, *Swiper*,[20] where the most of the code is only executed if the main entry point is given an `HTMLElement` object that contains child elements. It is also interesting to notice that large fractions of the type test script code are never executed (see the second row with numbers in Table 6.3). The dominant cause of uncovered test cases is the feedback mechanism being unable to provide the required base objects for testing method calls (see Section 6.5). As an example, for this reason only a few of the test cases for the *Sortable* library are reached.

Although TSTest succeeds in detecting numerous type mismatches with relatively simple means, these results indicate that it may be worthwhile in future work to extend TSTest with, for example, dynamic symbolic execution capabilities [98] to increase the coverage.

[20] `http://idangero.us/swiper/`

Do mismatches detected by TStest indicate bugs that developers want to fix?

To answer this question, we have randomly sampled 124 type mismatches reported by TSTEST and manually classified them into the following three categories. Those mismatches span 41 different benchmarks.

error (63/124): Mismatches that programmers would want to fix (excluding those that also match the following category).

strict nulls (47/124): Mismatches that programmers would want to fix, but are only valid when TypeScript's non-nullable types feature (introduced in TypeScript 2.0) is enabled (see Section 6.5).

benign (14/124): Mismatches that did not fit the above two categories.

We consider a type mismatch as something that programmers would want to fix if it is evident, by looking at the library implementation and documentation, that the actual behavior of the implementation is different from what was described in the declaration, and it is clear how the error can be fixed. An example is in the *P2.js*[21] library where the declaration states that `new p2.RevoluteConstraint(...).equeations` should result in an array, but the actual returned value is always `undefined` (because the property name `equations` was misspelled). The fix for this mismatch is clear: `equeations` should be corrected to `equations`. The classification is inevitably subjective, but we have striven to be conservative by classifying a mismatch as "benign" if there was any doubt about its category.[22] From this classification it is evident that most of the mismatches being detected are indeed errors that programmers would want to fix. None of the mismatches are false positive (in the sense defined in Section 6.6). Many of the errors are related to non-nullable types, which is unsurprising given that many declaration files were initially written before that feature was introduced in TypeScript. An example of such an error is from the library *lunr.js*,[23] where the method `get(id: string)` on the `Store` class is declared to always return an object of type `SortedSet<T>`. However, in the implementation such an object is only returned if a value has been previously set, and otherwise `undefined` is returned (which is valid when non-nullable types are disabled).

The 14 "benign" mismatches can be split into three sub-categories:

[21] `https://github.com/schteppe/p2.js`

[22] All details of the experiments are available at `http://www.brics.dk/tstools/`.

[23] `https://github.com/olivernn/lunr.js/`

Limitations of the TypeScript type system (4/14): With reflection being an often used feature of JavaScript, some constructs used by library developers are simply not expressible in the TypeScript language. Authors of the TypeScript declaration files therefore sometimes choose to write an incorrect type that is close to the actual intended type. The function declaration from the *Redux*[24] library is a typical example:

```
469 function bindActionCreators<A extends ActionCreator<any>>(
470   actionCreator: A,
471   dispatch: Dispatch<any>
472 ): A;
```

This declaration would lead one to believe that the return value of the function has the same type as the first argument `actionCreator`. However, the argument value and the return values do not have the same type. What happens instead is that the return value is an object containing only the function properties of the `actionCreator` argument (those properties are being transformed in a type preserving way). By using the same type parameter `A` as parameter type and return type, the TypeScript IDE is able to provide useful code completion and type checking for the function properties of the returned object in the application code, so in this case the author's choice is justifiable even though it is technically incorrect.

TStest constructing objects with private behavior (3/14): As explained in Section 6.5, type test scripts construct random values for function arguments with interface types. However, sometimes such values are only meant to be constructed by the library itself, since they contain private behavior that is intentionally not expressed in the declaration. This can lead to mismatches when the library tries to access the private behavior not present in the random values constructed by the type test scripts.

Intentional mismatches (7/14): For various reasons, declaration file authors sometimes intentionally write incorrect declarations even when correct alternatives are easily expressible, as also observed in previous work [47, 67]. A typical reason for such intentional mismatches is to document internal classes.[25]

In addition to the investigation of the 124 samples, for 6 of the 54 benchmarks (selected among the libraries that are being actively maintained and where TSTest obtained reasonable coverage) we created pull requests to fix

[24] `https://github.com/reactjs/redux/blob/f8ec3ef1c3027d6959c85c97459c109574d28b3c/index.d.ts#L343`

[25] An example of this: `https://github.com/pixijs/pixi.js/issues/2312/#issuecomment-174608951`

the errors reported by TSTEST.[26] The patches affect between 5 and 84 lines (totaling 331 lines) in the declaration files. All 6 pull requests were accepted by the maintainers of the respective declaration files. In almost all cases, the error was in the declaration file, however in one case the mismatch detected by TSTEST also revealed an error in the library implementation.[27]

Based on the output from TSTEST, it took only a couple of days to create all these patches, despite not having detailed knowledge of any of the libraries. This result demonstrates that TSTEST is capable of detecting errors that the developers likely want to fix, and that the output produced by TSTEST makes it easy to diagnose and fix the errors.

Can TStest find errors that are missed by other tools?

The only existing tool for automatically finding errors in TypeScript declaration files (without using existing unit tests) is TSCHECK [47]. Being based on static analysis, TSCHECK is in principle able to find mismatches that TSTEST cannot find due to the inherent incompleteness of dynamic analysis. However, TSCHECK is very cautious in reporting errors at all: it only reports an error if the static analysis concludes that there is no overlap between the inferred and the declared type, and TSCHECK is unable to reports errors involving function arguments. Although the more recent tool TSINFER [67] is designed for inferring rather than checking declaration files, the static analysis used in TSINFER has been demonstrated to be a significant improvement over TSCHECK, so we use TSINFER as a baseline representing the state of the art when measuring how many true positives are found by TSTEST but not by the existing techniques.

For each of the 63 type mismatches classified as errors in Section 6.7, we have investigated whether it could also be found by TSINFER. We chose not to test the mismatches classified as "strict nulls" because TSCHECK and TSINFER were developed before the introduction of that feature in TypeScript.

Some of the mismatches reported in Section 6.7 are quite easy to find, such as, properties missing on a globally defined object. We therefore created a simplified version of TSTEST, which does not call any functions except for constructors (constructors are invoked with no arguments). This simplified version of TSTEST mimics the dynamic analysis component of TSINFER. We classify as type mismatch as *trivial* if it can be found using this simplified version of TSTEST.

[26] List of the pull request: `https://gist.github.com/webbiesdk/eee08ce521f65536af1b87331e871421`

[27] The pull request fixing the implementation: `https://github.com/caolan/async/pull/1381`

As result, 33 of the 63 errors were classified as trivial mismatches and 30 as nontrivial mismatches. TSInfer was able to find all the trivial mismatches, however, it only found 10 of the 30 non-trivial mismatches. In other words, TSInfer finds only one third of the "nontrivial" mismatches that are found by TStest in this experiment.

Both TScheck and TSInfer suffer from false positives. In the evaluation of TScheck [47], 23% of the found mismatches were false positives (and other 16% were benign). While the evaluation of TSInfer [67] did not test for its efficiency in finding bugs, the quality of inferring method constructors was tested, and here TSInfer was able to infer the correct method signature for 23% of the constructors, and for 42% it was able to infer a signature that was close to the correct one. While those results are good when creating new declaration files from scratch, the false positive rates would be too big for it to have any practical use as an error finding tool. In comparison, as discussed in Section 6.7, we observe no false positives with TStest.

6.8 Related Work

Detecting type errors in dynamically typed programs The most closely related work is TScheck [47], which finds errors in TypeScript declaration files using a combination of static and dynamic analysis. The limitations of TScheck have been discussed in detail in Section 6.7. Recently, Kristensen and Møller [67] improved the analysis from TScheck and presented two new tools: TSInfer, which can automatically create TypeScript declaration files from JavaScript implementations, and TSevolve, which uses TSInfer to assist the evolution of declaration when the implementations are updated. The tool TPD [117] uses JavaScript's proxy mechanism to perform runtime checking of type contracts from TypeScript declaration files, based on the blame calculus by Wadler and Findler [115]. Unlike TStest, it does not perform automated exploration of the library code but relies on existing test suites. Also, as discussed in Section 6.5, that approach suffers from interference caused by the use of proxies. Safe TypeScript [91] extends TypeScript with more strict static type checks for annotated code and residual runtime type checks for the remaining code, but also without any automated exploration capabilities.

JSConTest [54] performs random testing of JavaScript programs with type-like contracts, but has to our knowledge not been applied to test TypeScript declaration files. Compared to TStest, its contract system does not support generics, and the automated testing is not feedback directed. TypeDevil [89] is a dynamic analysis that warns about inconsistent types in JavaScript programs, but it does not use TypeScript types nor automated

testing. TAJS [57] is a whole-program static analyzer for JavaScript that is designed to infer type information. It also does not use TypeScript types, and it is unable to analyze most of the JavaScript libraries mentioned in Section 6.7.

Flow [44] is a variant of TypeScript that performs more type inference and uses a similar notion of declaration files, called library definitions. We believe it is possible to adapt TSTEST to perform type testing of Flow's library definitions. Type systems have been developed also for other dynamically typed languages than JavaScript, including Scheme [109] and Python [71, 112]. These languages also provide typed interfaces to untyped libraries, so they have a similar need for tool support to detect errors, but are not yet used at the same scale as TypeScript.

Automated testing Automated testing is an extensively studied topic, and we can only discuss the most closely related work. As explained in Section 6.4, our approach builds on the idea of feedback-directed random testing pioneered by the Randoop tool [82]. Guided by feedback about the execution of previous inputs, Randoop aims to avoid redundant and illegal inputs and thereby increase testing effectiveness compared to purely random testing. Our main contribution is demonstrating that this approach can successfully be adapted to test TypeScript declarations.

Search-based testing is another approach to test automation, using for example genetic algorithms to maximize code coverage. A notable example is EvoSuite [49], which has support for testing generic classes in Java, similar to the challenge we address in Section 6.5. Property-based testing, or quickchecking [35], is another technique that can automatically generate inputs and check outputs for the system under test, often based on types. A fundamental difference in our work is the feedback mechanism.

In the area of JavaScript web application testing, the Artemis tool [22] also uses a feedback-directed approach, however using different forms of feedback, e.g. event handler registrations. Although the majority of the libraries considered in our experiments are intended for browser environments (see Appendix 6.10), TSTEST achieves good coverage and finds many errors even without taking the HTML UI event system into account. A possible avenue for future work is to investigate how the testing effectiveness of TSTEST could be improved by also triggering event handlers.

As mentioned in previous sections, TSTEST could in principle be extended with dynamic symbolic execution [98] to boost coverage. More specifically, the techniques used in CUTE [99] for systematically producing suitable object structures may be a useful supplement to randomly generated values for structural interface types. To this end, it may be possi-

ble to leverage previous work on symbolic execution for JavaScript from Kudzu [97], Jalangi [100], or SymJS [74].

6.9 Conclusion

We have demonstrated that feedback-directed random testing can successfully be adapted to detect errors in TypeScript declarations files for JavaScript libraries. Our approach works by automatically generating type test scripts that perform both runtime type checking and automated exploration of the library code. The prevalence of structural typing, higher-order functions, generics, and other challenging features of TypeScript have prompted many interesting design choices, most importantly how to use values obtained from the feedback process in combination with randomly generated values, and how to ensure that the feedback-directed process is not stopped each time a type error is encountered.

The experimental evaluation of our implementation, TSTEST, has shown that the technique is capable of fully automatically detecting numerous errors in TypeScript declarations files for a large range of popular JavaScript libraries. Despite the simplicity of the technique, errors were detected in 49 of 54 benchmarks. Among a sample of 124 reported type mismatches, 63 were classified as true errors, with additional 47 if using non-nullable types. Patches made for 6 erroneous declaration files have all been accepted by the declaration file authors, thereby confirming the usefulness of the technique. Moreover, TSTEST detects many errors that are missed by other tools, and without false positives.

Our coverage measurements show that substantial parts of the library code and the type test scripts are being covered, but also that there is a potential for improvement. In particular, we believe it may be interesting in future work to extend TSTEST with symbolic execution to increase coverage further. It may also be possible that our approach can be applied to other dynamically typed languages with optional types.

6.10 Appendix

Libraries used in the Experimental Evaluation

Name	environment	.js	.d.ts	Name	environment	.js	.d.ts
accounting.js	any	191	51	*MathJax*	browser	502	10
Ace	browser	7 958	629	*Medium Editor*	browser	5 211	140
AngularJS	browser	12 490	777	*Modernizr*	browser	1 193	349
async	any	1 733	202	*Moment.js*	any	3 244	501
axios	any	840	99	*P2.js*	browser	6 591	745
Backbone.js	browser	1 155	296	*pathjs*	browser	183	38
bluebird	any	4 939	195	*PDF.js*	browser	59 395	190
box2dweb	browser	10 718	1 139	*PeerJS*	browser	2 240	86
Chart.js	browser	11 870	385	*PhotoSwipe*	browser	2 602	146
CodeMirror	browser	7 302	402	*PixiJS*	browser	17 638	2 148
CreateJS	browser	8 955	1 325	*PleaseJS*	any	630	46
D3.js	browser	13 605	2 406	*Polymer*	browser	7 645	160
Ember.js	browser	31 357	1 299	*q*	any	1 137	100
Fabric.js	browser	14 633	1 099	*QUnit*	browser	3 038	109
Foundation	browser	5 646	285	*React*	browser	12 603	1 474
Hammer.js	browser	1 509	265	*Redux*	any	475	100
Handlebars	browser	3 444	241	*RequireJS*	browser	1 303	77
highlight.js	browser	128	10	*reveal.js*	browser	2 612	108
intro.js	browser	1 156	69	*RxJS*	any	9 281	1 002
Ionic	browser	8 660	310	*Sortable*	browser	879	76
Jasmine	any	2 891	442	*Sugar*	any	6 144	1 179
jQuery	browser	6 609	612	*Swiper*	browser	4 488	247
Knockout	browser	4 346	412	*three.js*	browser	23 299	4 292
Leaflet	browser	7 391	977	*Underscore.js*	any	2 896	1 171
Lodash	any	8 032	5 896	*Video.js*	browser	11 188	52
lunr.js	any	860	155	*Vue.js*	browser	6 733	581
Materialize	browser	5 253	88	*Zepto.js*	browser	1 298	3 336

The 'environment' column shows whether the library is primarily intended for browser-based applications. The '.js' and '.d.ts' columns show the sizes (line counts excluding dependencies) for the JavaScript implementation and the TypeScript declaration file, respectively.

Acknowledgments This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647544).

Chapter 7

Reasonably-Most-General Clients for JavaScript Library Analysis

By Erik Krogh Kristensen and Anders Møller. ICSE '19 Proceedings of the 41th International Conference on Software Engineering.

7.1 Abstract

A well-known approach to statically analyze libraries without having access to their client code is to model all possible clients abstractly using a *most-general client*. In dynamic languages, however, a most-general client would be too general: it may interact with the library in ways that are not intended by the library developer and are not realistic in actual clients, resulting in useless analysis results. In this work, we explore the concept of a *reasonably-most-general client*, in the context of a new static analysis tool ReaGenT that aims to detect errors in TypeScript declaration files for JavaScript libraries.

By incorporating different variations of reasonably-most-general clients into an existing static analyzer for JavaScript, we use ReaGenT to study how different assumptions of client behavior affect the analysis results. We also show how ReaGenT is able to find type errors in real-world TypeScript declaration files, and, once the errors have been corrected, to guarantee that no remaining errors exist relative to the selected assumptions.

7.2 Introduction

TypeScript has become a popular alternative to JavaScript for web application development. TypeScript provides static type checking, but libraries

are still implemented mostly in JavaScript. These libraries use separate type declaration files to describe the typed APIs towards the TypeScript application developers. The DefinitelyTyped repository contains 5 677 such type declaration files as of February 2019 [14]. Previous work has shown that there are numerous mismatches between the type declarations in these files and the library implementations, causing spurious type errors and misleading IDE suggestions when used by application developers [47, 67, 117].

Existing approaches in the literature for detecting such mismatches include TSCHECK [47], TSTEST [68], and TPD [117]. TSCHECK applies lightweight static analysis of the library functions, but the analysis is unsound, and errors can therefore be missed. Also, TSCHECK only reports an error if an inferred function result type is disjoint from the declared one, which makes TSCHECK miss even more errors. TSTEST is based on automated testing and as such inherently underapproximates the possible behaviors of libraries, resulting in no more than 50% statement coverage of the library code on average [68]. TPD similarly uses dynamic analysis, although with existing test suites to drive execution instead of automated testing, and therefore also misses many errors.

Another line of work involves static analysis for JavaScript. By conservatively over-approximating the possible behavior of the program being analyzed, static analysis tools can in principle detect all type errors exhaustively. FLOW uses fast type inference, but it is incapable of reasoning about unannotated library code [33]. Similarly to TypeScript, FLOW relies on type declaration files for interacting with untyped library code, and FLOW blindly trusts these files to be correct. Several static analyzers have been specifically designed to detect type-related errors in JavaScript programs, without requiring type annotations. State-of-the-art tools are TAJS [20, 57], SAFE [86], and JSAI [63]. However, these analyzers have not been designed for analyzing libraries without client code, and they do not exploit or check TypeScript types.

Although previous approaches have been proven useful for finding mismatches between the TypeScript declaration file and the JavaScript implementation of a given library, *none of them can guarantee that they find all possible type mismatches* that a realistic client may encounter when using the declaration file and the library. In this work we present a novel framework that aims to complement existing techniques by having the ability to *exhaustively* find all possible type mismatches, or prove that there are none, relative to certain reasonable assumptions.

The approach we take is to build on an existing static type analysis tool for JavaScript, specifically the TAJS analyzer. The first challenge is that such tools have been designed with a closed-world assumption, i.e., where the

entire program is available for the analysis, whereas we need to analyze library code without having access to client code. In the past, the problem of analyzing an open program using an analysis designed with a closed-world-assumption has been addressed through the notion of a *most-general client* for the library [18, 95]. A most-general client is an artificial program that interacts with the library in all possible ways, thereby soundly modeling all possible actual clients. However, we find that the concept of a most-general client does not work well for a dynamic language like JavaScript. Due to the poor encapsulation mechanisms in JavaScript, clients can in principle interfere with the library in ways that are not intended by the library developer and are not realistic in actual clients. As a simple example, a most-general client may overwrite parts of the library itself or the standard library that the library relies on, thereby breaking its functionality and rendering the static analysis results useless. For this reason, we introduce the concept of a *reasonably-most-general client* that restricts the capabilities of the artificial client. Our framework provides a methodology for library developers to exhaustively detect possible type mismatches under different assumptions of the client behavior.

Existing static type analysis tools for JavaScript, including TAJS, have not been designed with support for TypeScript type declarations, however, it turns out that TypeScript's notion of types fits quite closely with the abstract domains used by TAJS. A bigger challenge is that the TypeScript type declarations for libraries are written in separate files, with no clear link between, for example, the type declaration for a function and the code that implements that function. JavaScript libraries initialize themselves dynamically, often in complicated ways that are difficult to discover statically. To this end, we adapt the feedback-directed approach by TSTEST, which incrementally discovers the relation between the type declarations and the library implementation, to a static analysis setting.

By building on an existing static analysis tool for JavaScript, we naturally inherit some of its limitations (as well as any improvements made in the future). Although much progress has been made to such tools within the last decade, JavaScript libraries are notoriously difficult to analyze statically, even when considering simple clients [20, 76, 86, 92]. The goal of this paper is not to improve the underlying static analysis tool, but to explore how such a tool can be leveraged to exhaustively find errors in TypeScript declaration files. Usually, when JavaScript analyzers encounter difficulties regarding scalability and precision, they do not degrade gracefully but fail with an error message about a catastrophic loss of precision, inadequate memory, or a timeout. To partly remedy this problem, our framework selective stops the analysis of problematic functions.

In summary, the contributions of our work are the following.

- We introduce the concept of a *reasonably-most-general client* (RMGC) that restricts the traditional notion of most-general clients to enable static analysis of JavaScript libraries (Section 7.4). Some of the restricting assumptions that we consider are necessary for the analysis to have meaningful results; others provide a trade-off between generality of the RMGC, i.e. what errors can possibly be found, and false positives in the analysis results.
- We discuss how to incorporate abstract models of the different variations of RMGCs on top of an existing static analysis tool (TAJS) that has originally been designed for whole-program JavaScript analysis, thereby enabling open-world analysis of JavaScript libraries (Section 7.6). By adding support for creating abstract values from TypeScript types and, conversely, type-checking abstract values according to TypeScript types, the resulting analysis tool can exhaustively detect errors in TypeScript declaration files for JavaScript libraries. We adapt the feedback-directed technique from TSTEST to incrementally discover the relation between the type declarations in the TypeScript declaration files and the program code in the JavaScript implementation.
- We experimentally evaluate our tool, REAGENT, on 10 real-world libraries (Section 7.7). REAGENT uses TAJS largely unmodified, and we believe it could easily be ported to similar analyzers, such as SAFE or JSAI. With REAGENT, we detected and fixed type errors in these 10 libraries (totaling 27 lines changed across 7 libraries), with the guarantee that the fixed declaration files do not contain any remaining type errors, under the assumptions of the RMGC. Moreover, we investigate the impact of each optional assumption of the RMGC by evaluating the trade-off between generality of the RMGC and accuracy of the analysis.

7.3 Motivating example

To motivate our approach, we begin by describing an example from the `semver` library,[1] which is a small library for handling version numbers according to the semantic versioning scheme. A simplified portion of the library implementation is shown in Figure 7.1a. The constructor in line 473 returns a `SemVer` object if the argument string matches the semantic versioning scheme. The DefinitelyTyped repository hosts a declaration file for `semver`, a small part of which is shown in Figure 7.1b. Our goal is to detect mismatches between the declaration file and the implementation of the

[1] `https://github.com/npm/node-semver`

```
function SemVer(version) {
  ...
  if (version.length > MAX_LENGTH) throw new TypeError()
  var m = version.trim().match(REGEXP);
  if (!m) throw new TypeError();
  ...
  // numberify any prerelease numeric ids
  this.prerelease = m[4].split('.').map((id) => {
    if (/^[0-9]+$/.test(id) && +id >= 0 && +id < MAX_INT)
      return +id;
    return id;
  });
  this.format();
}
SemVer.prototype.format = function() {
  this.version =
    this.major + '.' + this.minor + '.' + this.patch;
  if (this.prerelease.length)
     this.version += '-' + this.prerelease.join('.');
  return this.version;
};
```

(a) A simplified version of the `SemVer` constructor.

```
export class SemVer {
  constructor(version: string | SemVer);
  major: number;
  minor: number;
  patch: number;
  version: string;
  prerelease: string[];
  format(): string;
  compare(other: string | SemVer): 1 | 0 | -1;
}
```

(b) The declaration for `SemVer` from the TypeScript declaration file.

```
for path: SemVer.new().prerelease.[numberIndexer]
  Expected string or undefined but found number
```

(c) Error reported by ReaGenT for the `prerelease` property.

Figure 7.1: Implementation and declaration file of the `semver` library.

library. For this reason, we need to consider how clients may interact with the library.

Because of the dynamic nature of JavaScript, a client of the library could

in principle interact with the library in ways that are not possible with statically typed languages. A most-general client interacting with the `SemVer` library would perform every possible action, including replacing the `format` function (declared in line 487) with a function that just returns a constant string. The `format` function in `SemVer` is responsible for creating and setting the `version` property of the `SemVer` class (line 488), and thus replacing `format` will cause all objects created by the `SemVer` constructor to lack the `version` property. A most-general client will thus find that the declaration file, which states that the `version` property is present, may be erroneous. If the library developer did not intend for clients to overwrite the `format` function, and no client developer would ever consider doing so, then the missing `version` property is a false positive.

Our *reasonably-most-general client* (RMGC) works under a set of assumptions, described in Section 7.4, that restrict the actions performed compared to a truly "most general" client. One of these assumptions includes that the RMGC does not overwrite library functions, and under this assumption the false positive related to the `version` property would not occur.

It is nontrivial for any automated technique that relies on concrete execution to provide sufficient coverage of the possible behavior of `semver`. For instance, the simple random string generator in TSTEST will generate a string matching the semantic versioning scheme with a probability of around $1/10^{13}$, and without such a string, no `SemVer` object will ever be created, so most of the library will remain untested. By the use of abstract interpretation, we overcome the shortcomings of the techniques that rely on concrete executions. This approach allows us to evaluate the `SemVer` constructor abstractly with an indeterminate string value passed as parameter. When the `SemVer` constructor is evaluated abstractly, the condition in line 477 is considered as possibly succeeding by the abstract interpreter, so that an abstract `SemVer` object is constructed and returned, which is necessary to test the rest of the library.

An example of a type mismatch that is not easily detectable by techniques that rely on concrete executions, but is found by our abstract RMGC, is a mismatch related to the `prerelease` property declared in line 500. The correct type for the `prerelease` property is `(string | number)[]`, since a conversion to `number` is attempted for every element of the array during the initialization of the `SemVer` object (lines 480–484). The type violation reported by REAGENT for this error is shown in Figure 7.1c.

The downside of using abstract interpretation is that REAGENT is sometimes overly conservative and may report type violations in situations where no concrete execution could lead to a type mismatch. However, in return, it finds all possible mismatches, relative to the RMGC assumptions.

7.4 Reasonably-Most-General Clients

A most-general client (MGC) uses a library by reading and writing its object properties and invoking its functions and constructors. We refer to such possible uses as *actions*.

A library can be stateful, so the behavior of its functions may depend on actions that have been performed previously, so an MGC must perform any possible sequence of actions, not just single actions. Some of the function invocations performed by the MGC involve callback functions that originate from the MGC and may similarly perform arbitrary actions. An MGC may invoke the functions of a library with arguments of any type, even if the declaration file declares that the function should be called with arguments of a specific type. However, the declaration file describes a contract between the library and the client. If we know that the client does not break this contract, then the library can be blamed for any type errors that are encountered.

A *reasonably-most-general client* (RMGC) does the same as an MGC, but with certain restrictions that ensure both that the RMGC does not break the contract in the declaration file, and that the RMGC does not interact with the library in ways that are unrealistic and unintended by the library developer. The first two assumptions we describe next are necessary for the RMGC to work meaningfully, whereas three other assumptions are optional and ultimately depend on what guarantees the user of our analysis wants.

Respecting declared types

A necessary assumption is that an RMGC respects declared types: If a function in a library is declared as receiving, e.g., numbers as its arguments, then the RMGC only calls the function with numbers. Otherwise we would be unable to blame the library and its type declaration file for any type mismatch that occurs when the client uses the library.

Assumption 1. [**respect-types**] An RMGC respects the types declared in the type declaration file, when passing values to the library.

Note that because TypeScript's type system is inherently unsound [26], this assumption is not the same as requiring that the client passes the TypeScript type check without warnings.

A consequence of this assumption is that the set of possible actions is bounded by the types that appear in the declaration file, which is useful when we in Section 7.6 define the notions of abstract RMGCs and action coverage.

Preserving the library

As motivated in Section 7.3, TypeScript clients can in principle overwrite library functions (like the `format` function in the example), but it is clearly unreasonable to blame the library for type errors that result from that. A possible approach is to expect that properties that are intended to be read-only are declared as such in the declaration file. TypeScript properties are writable by default but can be declared with the modifier `readonly` or `const`. However, authors of declaration files rarely use these modifiers. Additionally, some features, such as class methods, cannot easily be declared as read-only.[2]

Overwriting properties of standard libraries, specifically the ECMAScript standard library, the browser DOM API, and the Node.js API, may similarly cause the library under test to malfunction. For instance, the `SemVer` constructor from the motivating example depends on the functionality of the `String.prototype.trim` function from the ECMAScript standard library. Only a few of the properties of the standard libraries are marked as read-only. Sometimes some of these non-read-only properties are overwriten on purpose, for example to improve support of certain features in outdated browsers by loading polyfills.[3] Still, both regarding the library under test and the standard libraries, it is reasonable to assume that a library does not depend on the client to overwrite functions in the library, which justifies the following assumption.

Assumption 2. [**preserve-libraries**] An RMGC considers all properties of the standard libraries and all properties declared with a non-primitive type[4] from the library under test as read-only and thus never writes to those properties.

This assumption does not prevent the RMGC from writing to library properties declared with primitive types (such properties are occasionally used for library configuration purposes). In contrast, writing to an undeclared property is considered a type error in TypeScript, so the RESPECT-TYPES assumption ensures that the RMGC never does so.

Obtaining values for property writes and function arguments

Whenever an RMGC passes an argument to a library function or writes to a property of a library object, a value of the declared type is needed. There

[2] It is possible to create a read-only method by declaring it as a property with a function type, but this feature is rarely used.

[3] `https://www.w3.org/2001/tag/doc/polyfills`

[4] The primitive types in TypeScript are `boolean`, `string`, `number`, `undefined`, `symbol`, and `null`.

are two ways the RMGC can obtain such a value: either the value originates from the library (and the client receives the value via a function call, for example), or the value is constructed by the RMGC itself. We refer to these as *library-constructed values* and *client-constructed values*, respectively. Even an MGC cannot construct all possible values itself—for example, a library-constructed value may be a function that has access to the internal state of the library via its free variables—so we need to take both kinds of value constructions into account.

TypeScript is *structurally typed*, meaning that when a function is declared as taking an argument of some object type, then the type system allows the function to be called as long as the argument is an object of the right structure. According to the RESPECT-TYPES assumption, an RMGC should pass any structurally correct client-constructed or library-constructed value of the desired type. However, that is not always the intent of the library developers, as the structural types may not fully describe what is expected from the arguments. For instance, the structural types in TypeScript cannot describe prototype inheritance, and sometimes a library assumes other invariants about values constructed by the library itself.

Example 1. In the code below, taken from the `Leaflet` library,[5] the value of `this.options.tileSize` is supplied by the client, and the `tileSize` property is declared to have type `L.Point | number`. If `tileSize` is set to a value that has the same structure as `L.Point` but is not constructed by the `L.Point` constructor, then the `instanceof` check in line 508 in the program will fail, resulting in an invalid `L.Point` being constructed.

```
506 var getTileSize = function () {
507   var s = this.options.tileSize;
508   return s instanceof L.Point ? s : new L.Point(s, s);
509 }
```

The following assumption may better align with the intended use of such a library.

Assumption 3. [**prefer-library-values** – ***optional***] When passing values of non-primitive types to the library, an RMGC uses library-constructed values if possible; client-constructed values are only used if the RMGC is unable to obtain library-constructed values of the desired types according to the type declaration file.

A recent study of JavaScript object creation [118] has found that it rarely happens that the same property read in a program uses objects that were

[5] `https://github.com/Leaflet/Leaflet`

created at different program locations, suggesting that the PREFER-LIBRARY-VALUES assumption is satisfied by most clients in practice.

String values are optionally handled in a special way. Since a string provided by the RMGC might be used in a property lookup on an object in the library, a string that is the name of a property defined on, for example, `Object.prototype` may result in a property of `Object.prototype` being accessed. The (perhaps implicit) assumption of the library developer in this case might be that clients do not use strings that are property names of prototype objects from the standard library, as such accesses could have unintended consequences.

Example 2. In the simplified code below taken from the `loglevel` library,[6] the `getLogger` function (line 512) makes sure that only one `Logger` of a given name is constructed, by checking if a property of that name is defined on the `_loggers` object (lines 513–514). If a `Logger` has already been constructed it is returned (line 517), and otherwise a new one is created (line 515). However, if the name is, for example, `toString` then the property lookup in line 513 will return the `toString` method defined on `Object.prototype`, and that method will then be returned by `getLogger` resulting in a type mismatch.

```
510 declare function getLogger(name: string) : Logger

511 var _loggers = {};
512 function getLogger(name) {
513   var logger = _loggers[name];
514   if (!logger) {
515     logger = _loggers[name] = new Logger(...);
516   }
517   return logger;
518 };
```

This observation motivates the following assumption.

Assumption 4. [**no-prototype-strings** – ***optional***] An RMGC does not construct strings that coincide with the names of properties of the prototype objects in the standard libraries.

Another issue is that TypeScript's type system supports *width subtyping*, which means that for an object to match a type, the object should have all the properties declared by the type, and any undeclared property in the type can be present in the object and have any value. Therefore it seems natural that

[6] `https://github.com/pimterry/loglevel`

when an RMGC constructs an object of some type, the constructed object may also have undeclared properties.

However, since these undeclared properties can have arbitrary values, false positives might appear if the library reads one of these undeclared properties.

Example 3. In the simplified example below from the `uuid` library,[7] the `v4` function obtains random numbers from the `opts` object (line 522) and puts them into the `buf` array (line 524). The `opts` object can have two different types (declared in line 519). The `v4` function attempts to detect which of the two types the concrete `opts` object has, and uses this to create an array of random numbers (line 522). A client can choose to use the second variant of the `opts` object that only has declared a `rng` property, but because of width subtyping the client is technically allowed to add a property `random` of any type to that object. If the client chooses to call `v4` with such an `opts` object, then the property read `opts.random` can read any value, which in turn can cause a false positive when non-number values are written to the `buf` array (line 524).

```
519 type Opts = {random: number[]} | {rng(): number[]};
520 declare function v4(opts: Opts, buf: number[]) : number[]

521 function v4(opts, buf) {
522   var rnds = opts.random || (opts.rng || _rng)();
523   for (var i = 0; i < 16; i++) {
524     buf[i] = rnds[i];
525   }
526   return buf;
527 }
```

We therefore leave it as an optional assumption whether client-constructed objects should have undeclared properties.

Assumption 5. [**no-width-subtyping** – ***optional***] An object constructed by the RMGC does not have properties that are not declared in the type.

If this assumption is disabled, for client-constructed objects, all properties that are not explicitly declared may have arbitrary values of arbitrary types.

In the following sections, we demonstrate that these five assumptions are sufficient to enable useful static analysis results for JavaScript libraries.

[7] `https://github.com/kelektiv/node-uuid`

7.5 Abstract Domains in Static Type Analysis

To be able to explain how to incorporate RMGCs into the TAJS static analyzer, we briefly describe the structure of the abstract domains used by TAJS [20, 57] (and related tools like SAFE [86], and JSAI [63]).

TAJS is a whole-program abstract interpreter that over-approximates the flow of primitive values, objects, and functions in JavaScript programs. (We here ignore many details of the abstract domains, including the use of context-sensitivity, that are not relevant for the topic of RMGCs.) Abstract objects are partitioned by the source locations, called *allocation-sites*, where the objects are created [32]. Basically, at each program point, TAJS maintains an *abstract state*, which is a map from allocation-sites to abstract objects, and an abstract object is a map from abstract property names to abstract values. Abstract values are described by a product lattice of sub-lattices for primitive values of the different types (strings, numbers, booleans, etc., as in traditional constant propagation analysis [29]) and a sub-lattice for object values (modeled by allocation-sites, like in traditional points-to analysis [32]) and abstract function values (like in traditional control-flow analysis [101]). As in other dataflow analyses, TAJS uses a worklist algorithm to propagate abstract states through the program until a fixed-point is reached. We refer to the literature on TAJS for more details.

7.6 Using RMGCs in Static Type Analysis

Our static analysis is made of two components: the TAJS abstract interpreter and an *abstract* RMGC. The abstract RMGC interacts with the library by using the abstract interpreter to model the actions described in Section 7.4. It maintains an abstract state, `allState`, that models all program states that are possible with the actions analyzed so far.

The basic steps of the abstract RMGC are shown in Algorithm 2. The abstract RMGC cannot immediately invoke all functions in the library, because the connection between the implementation of a function and the declared type of the function is only known after a reference to the function has been returned by the library. In the motivating example (Section 7.3), if the `compare` method had been defined in the `SemVer` constructor instead of being present on the `SemVer.prototype` object, a client would only be able to invoke the method after having constructed an instance of `SemVer`. Therefore, a crucial component of the abstract RMGC is a map, called `vmap`, from types in the declaration file to abstract values as used by TAJS (see Section 7.5). A type is modeled as an access path in the declaration file; for example, the access path `SemVer.new().minor` is the TypeScript type `number` in the `semver` declaration file. This map allows the abstract RMGC to keep track of which

Algorithm 2: The iterative algorithm performed by the abstract RMGC.

```
   Input: library source code and TypeScript declaration
14 invoke TAJS to analyze the library initialization code
15 allState ← abstract state after library initialization
16 vmap ← [library type ↦ library abstract value]
17 do
18     for all functions f in vmap do
19         args ← use obtainValue to get arguments for f
20         propagate allState and args to function entry of f
21     invoke TAJS to analyze new dataflow
22     for all functions f in vmap do
23         propagate state at function exit of f to allState
24         addLibVal(abstract return value of f,
                     declared return type of f)
25     for all properties p in all objects o in vmap do
26         addLibVal(abstract value of p in allState,
                     declared type of p)
27 while allState or vmap changed
```

parts of the library have been explored so far during the analysis, and for obtaining abstract library-constructed values for further exploration. Abstract values in `vmap` that contain allocation-sites (modeling references to objects, cf. Section 7.5) are interpreted relative to `allState`.

The abstract RMGC first loads the library by abstractly interpreting the library initialization code (line 14 in Algorithm 2), and then setting `allState` to be the resulting abstract state (line 15). The initialization of libraries intended for use in web browsers consists of dynamically building an object that is eventually written to a property of the JavaScript global object (which is treated as a special allocation-site in TAJS). Clients then use the property of the global object as an entry point for the library API. Our abstract RMGC uses this property by inserting it into `vmap` associated with the declared library type (line 16). (Initialization of Node.js libraries works slightly differently and is ignored here to simplify the presentation.) For the `semver` example from Section 7.3, `vmap` then maps the type `SemVer` to the abstract value that models the object produced by the library initialization code. All other entries in `vmap` initially map to the bottom abstract value, denoted $\bot$.

After the initialization phase, the abstract RMGC works iteratively (lines 17–27). In each iteration, it abstractly invokes each library function that exists in `vmap`. The auxiliary function obtainValue provides abstract

Algorithm 3: Handling library-constructed abstract values.

Input: an abstract value and a TypeScript type
ADDLIBVAL(*val*, *type*)
28 **if** not TYPECHECK(*val*, *type*) **then**
29 report type violation
30 *fval* ← FILTER(*val*, *type*)
31 `vmap` ← `vmap` + [*type* ↦ *fval*]

values for the arguments as explained in Section 31. By propagating[8] `allState` and the arguments to the function entry (line 20) and then invoking TAJS (line 21), the function bodies are analyzed. Next, for each of the functions, the resulting abstract state at the function exit is propagated into `allState` (line 23) and the abstract return value is collected (line 24). Similarly, for every object whose type is contained in `vmap`, all properties declared by the object type are collected (lines 25–26). (For simplicity we here ignore properties with getters and setters.)

The auxiliary function ADDLIBVAL (Algorithm 3) first type-checks the abstract value according to the declared type (lines 28–29) using the function TYPECHECK explained in Section 40. The abstract value is then passed through a function, FILTER, that performs type refinement [62] to remove parts of the abstract value that do not match the type, and the resulting abstract value is added[9] to `vmap` (line 31).

The entire process is repeated until no more dataflow appears in `allState` and `vmap`.[10] When the fixed-point is reached, `allState` models an over-approximation of all possible states at the entries and exits of the reachable library functions. Thereby the state in the beginning of all library functions includes the side-effects from all other library functions, and the abstract RMGC therefore models all states that can result from calling the functions in any possible sequence.

Obtaining abstract values for library function arguments

The pseudo-code in Algorithm 4 shows how OBTAINVALUE provides abstract values for the abstract RMGC, either by producing new abstract values (to model the client-constructed values) or using abstract values from `vmap` (for

[8] *Propagating* an abstract state X into Y means setting Y to the least-upper-bound of X and Y.

[9] The '+' operator used in Algorithm 3 denotes updating using least-upper-bound; specifically, line 31 updates the `vmap` entry for *type* to become the least-upper-bound of the existing abstract value and *fval*.

[10] The lattices in TAJS have finite height, which ensures termination.

Algorithm 4: Algorithm for obtaining an abstract value for a given type.

Input: a TypeScript type
Result: abstract value modeling the type
OBTAINVALUE(*type*)
32 **if** *type* is primitive **then**
33 **return** create primitive value from *type*
34 **else if** *type* in stdlib **then**
35 **return** CREATENATIVE(*type*)
36 *val* ← `vmap`(*type*)
37 *obj* ← new abstract object
38 **for** all properties *p* in *type* **do**
39 *obj*[*p*] ← OBTAINVALUE(*type*[*p*])
40 **return** *val* ⊔ *obj*

the library-constructed values). Line 19 in Algorithm 2 calls OBTAINVALUE for every available function parameter type.

For primitive types, such as `number` or `string`, we let the abstract RMGC construct the value (line 33). Creating an abstract value that describes all possible values of a primitive type is trivial due to the abstract domains already supported by TAJS, as discussed in Section 7.5. If NO-PROTOTYPE-STRINGS is enabled, abstract string values are created accordingly.

Types declared in a standard library need special treatment. For example, the type declaration of `Function` contains no information that the value is in fact a callable function. The function CREATENATIVE (line 35) takes care of creating the right abstract values for such types; we omit the details here.

If the type is neither primitive nor from a standard library, an abstract value is created that models the relevant library-constructed and client-constructed objects.[11] Library-constructed objects are taken from `vmap` (line 36), and client-constructed objects are made using OBTAINVALUE recursively by following the structure of the type in the TypeScript declaration (lines 38–39). (In case of recursive types, the abstract objects are reused to ensure termination.) When creating an object from a type, the type is used as an artificial allocation-site (line 37), which ensures that TAJS correctly models possible aliasing.[12]

The pseudo-code in lines 38–39 shows how abstract client-constructed objects are obtained when NO-WIDTH-SUBTYPING is enabled. If that assumption is disabled, the ⊤ ("top") abstract value is additionally assigned to all

[11] '⊔' in line 40 denotes the least-upper-bound on abstract values.

[12] Subtyping is handled soundly by including the allocation-sites of supertypes when creating the new abstract objects.

Algorithm 5: Type-checking abstract values.

Input: an abstract value and a TypeScript type
Result: true if the abstract value matches the type
TYPECHECK(*value*, *type*)

```
41  if type is primitive then
42      return true if value matches type
43  else if type in stdlib then
44      return CHECKNATIVE(value, type)
45  else if type is function then
46      return true if value is a function
47  else if value is not an object then
48      return false
49  else
50      for all properties p in type do
51          if not TYPECHECK(value[p], type[p]) then
52              return false
53      return true
```

undeclared properties of the new abstract object.

Functions are just objects in JavaScript, and client-constructed functions are thus obtained in essentially the same way as ordinary objects. The only difference is that the artificial allocation-site is marked as being a function (not shown in the pseudo-code), which informs TAJS that the new object can be called as a function. When TAJS finds that the library invokes such a client-constructed function, each argument is processed using ADDLIBVAL and a return value is created using OBTAINVALUE.

If PREFER-LIBRARY-VALUES is enabled then we omit the client-constructed abstract value *obj* in line 40 and simply return *val* if values of the desired type can be obtained from the library according to the type declaration file, which can be implemented with a simple reachability check.

Type-checking abstract values

When the abstract RMGC receives a value from the library, either by invoking a function or by reading a property from an object, Algorithm 3 uses TYPECHECK to check whether the value has the right type according to the corresponding type declaration. This process is straightforward as outlined in Algorithm 5. Checking primitive types (line 42) is trivial for the same reason as creating abstract values of primitive types is trivial (see Section 31), and types declared in a standard library need special treatment (line 44) for the same reason as creating them requires special treatment.

If an abstract value is checked against a function type, we only need to check whether the value is a function (line 46); the parameter types and the return type are not used until the function has been invoked by Algorithm 2.

Checking object types requires checking that the abstract value is an object and recursively checking the declared properties (lines 47–53). Recursive types are handled co-inductively (ignored in the pseudo-code for simplicity).

Coping with analysis precision and scalability issues

Our technique can in principle find all possible type mismatches that could be encountered by a client satisfying the RMGC assumptions. This is justified by TAJS being a "soundy" [75] static analysis, and by our abstract RMGC over-approximating the actions of an RMGC, meaning that there may be false positives but we should not expect false negatives. There are two possible causes of false positives: (1) imprecision of the underlying static analysis, and (2) the RMGC being "too general", lacking reasonable assumptions about how real clients may behave.

It is well-known that many real-world JavaScript libraries contain code that is extremely difficult to analyze statically [21, 85]. Inadequate precision of the static analysis may cause an avalanche of spurious dataflow, rendering the analysis results useless. We use some simple heuristics to detect if the analysis of a library function is encountering a catastrophic loss of precision or is taking too long.[13] In these cases we unsoundly stop the analysis of that function, but allow the analysis to proceed with other functions.

We define *action coverage* as the percentage of actions that were successfully performed by the abstract RMGC. An action coverage of 100% means that the abstract RMGC was able to analyze the entire library exhaustively, without stopping analysis of any functions. If the action coverage is below 100%, it means that either our analysis fails to analyze one of the functions, or that an action is unreachable typically because of a mismatch between the TypeScript declaration file and the library implementation. In the first case, our abstract RMGC may miss type violations that could be encountered by a client that invokes those functions. Yet, the analysis remains exhaustive for those clients that do not perform any of the stopped actions. Hence, when the analysis terminates, action coverage measures the portion of the library API that has been analyzed exhaustively.

[13] We declare a function as timed out if TAJS has used more than 200 000 node transfers to analyze it (typically corresponding to a few minutes), and we characterize a catastrophic loss of precision as a property read on an abstract value that represents at least two different standard library objects.

7.7 Evaluation

Two of the RMGC assumptions described in Section 7.4 are *necessary* to obtain any useful analysis results, and the remaining three assumptions are *optional* and can be selected by the analysis user. Imposing too many restrictions on the model of the clients may prevent detection of errors due to inadequate coverage, whereas imposing too few may cause false positives and also significantly degrade analysis performance because the abstract client becomes "too general".

To evaluate the usefulness of the concept of an RMGC and the effects of the optional assumptions, we have implemented the abstract RMGC described in Section 7.6 in a tool, ReaGenT (REAsonably-most-GENeral clienT). It uses the abstract interpreter TAJS unmodified, except for a small adjustment of its context-sensitivity strategy [43]: to increase analysis precision, every time the abstract RMGC invokes TAJS to analyze a function (lines 20–21 in Algorithm 2), the context is augmented by the access path of the function (using the same notion of access paths as in Section 7.6). This adjustment can also easily be made to other JavaScript static analyzers [63, 86].

In our evaluation we aim to answer the following main research questions.

RQ1 Does the RMGC enable static type analysis of JavaScript libraries, using an off-the-shelf, state-of-the-art whole-program static analyzer?

RQ2 How does each of the optional RMGC assumptions affect the ability of ReaGenT to detect type errors in JavaScript libraries?

As benchmarks, we have randomly selected 10 JavaScript libraries that have TypeScript declarations in the DefinitelyTyped repository. For the reasons given in Sections 7.2 and 53, we focus on small libraries only (up to 50 LOC in the declaration file). The libraries, which are available from the npm repository,[14] and the sizes of their declaration files (measured with CLOC) are shown in the first columns of Table 7.1. We use the latest versions of all the libraries and declaration files.

Our implementation of ReaGenT and all experimental data are available at `http://brics.dk/tstools/`. The experiments are performed on computer with 16GB of RAM and an Intel i7-4712MQ CPU.

[14] `https://www.npmjs.com/`

Table 7.1: Number of lines in the JavaScript implementation, along with total lines, changed lines resulting from our fixes, and lines with false positives after the fixes, for the corresponding type declaration file.

	Impl.	Type Declaration File		
Library	Lines	Total	Changed	False positives
`classnames`	37	9	0	0
`component-emitter`	72	13	3	6
`js-cookie`	127	23	4	0
`loglevel`	176	40	7	1
`mime`	915	9	1	0
`pathjs`	183	38	5	1
`platform`	741	22	2	4
`pleasejs`	630	46	5	2
`pluralize`	315	13	0	0
`uuid`	86	23	0	0

RQ1: Does the RMGC enable static type analysis of JavaScript libraries?

No existing static analysis is capable of helping programmers find all type errors in JavaScript libraries that have TypeScript declarations, even if restricting to small libraries like the selected benchmarks. To investigate whether our RMGC enables such analysis, we perform an experiment where we run ReaGenT on the 10 libraries, using the configuration where all optional assumptions are enabled. For each reported type violation, we manually classify it as a true positive (usually an error in the type declaration file) and then fix it, or mark it as a false positive (either caused by the RMGC being too general and therefore modeling unrealistic clients, or by the underlying static analysis being too imprecise). This process is repeated until ReaGenT no longer reports any violations.

The results are summarized in Table 7.1, which shows how many lines were changed in each declaration file to fix true positives, and for how many lines in the fixed declaration file ReaGenT falsely reports a violation. Even without expert knowledge of the libraries, classifying the type violation reports and fixing the true positives was straightforward based on the output of ReaGenT. For the seven declaration files being fixed, we created pull requests, which were all accepted by the maintainers. Table 7.1 shows that ReaGenT can find actual errors in many libraries, which is not surprising given that previous work has shown that many type declaration files are erroneous [47, 68, 117]. More importantly, Table 7.1 shows that ReaGenT does not overwhelm the user with false positives, as there are only 14 lines containing false positives across five of the libraries.

Example 4. The `getJSON` function in the `js-cookie` library parses the value of a browser cookie. According to the declaration file, the function always returns an object:

```
528 declare function getJSON(key: string) : object;
```

The implementation (shown below) iterates through all the cookies (lines 532–538) and parses each cookie using the `JSON.parse` function. If the `key` argument is the same as the name of the cookie then the value of the cookie is returned (line 537). However, the returned value is the result of the `JSON.parse` call, which can be any type, including primitives. The declared return type `object` is therefore wrong.[15]

```
529 function getJSON(key){
530   var result = {};
531   var cookies = document.cookie.split('; ');
532   for (var i = 0; i < cookies.length; i++) {
533     var parts = cookies[i].split('=');
534     var name = parts[0];
535     var cookie = JSON.parse(parts[1]);
536     if (key === name) {
537       return cookie;
538   } }
539   return result;
540 }
```

Notice how difficult it would be to detect this error using other techniques: the client must set a cookie whose value results in a non-object value when passed through `JSON.parse`, and then call `getJSON` with the name of that cookie.

[15] This error has since been fixed, see `https://github.com/DefinitelyTyped/DefinitelyTyped/pull/28529`.

Table 7.2: Comparison of RMGC variants. Each column contains *action coverage* / *violations*.

Library	all assumptions enabled	no-width-subtyping disabled	no-prototype-strings disabled	prefer-lib-values disabled	all three disabled
`classnames`	100.0% / 0	*timeout* / 0	100.0% / 0	100.0% / 0	*timeout* / 0
`component-emitter`	96.0% / 72	52.0% / 21	96.0% / 72	96.0% / 72	*timeout* / 73
`js-cookie`	100.0% / 4	*timeout* / 0	100.0% / 4	100.0% / 4	*timeout* / 0
`loglevel`	100.0% / 3	100.0% / 3	100.0% / 15	100.0% / 3	100.0% / 15
`mime`	8.3% / 1	8.3% / 1	8.3% / 1	8.3% / 1	8.3% / 1
`pathjs`	100.0% / 7	*timeout* / 30	100.0% / 12	100.0% / 7	*timeout* / 36
`platform`	100.0% / 10	100.0% / 10	100.0% / 10	100.0% / 10	100.0% / 10
`pleasejs`	100.0% / 16	*timeout* / 0	100.0% / 16	100.0% / 16	*timeout* / 0
`pluralize`	100.0% / 0	100.0% / 0	100.0% / 0	100.0% / 0	100.0% / 0
`uuid`	100.0% / 0	61.9% / 0	100.0% / 0	100.0% / 0	61.9% / 0
Average	90.4% / 11.3	– / 6.5	90.4% / 13.0	90.4% / 11.3	– / 13.5

Example 5. One of the real errors found by ReaGenT in the `component-emitter` library involves the `Emitter` function that is declared as returning an object of type `Emitter`:

```
541 declare function Emitter(obj: any) : Emitter;
```

In the implementation (shown below) if an object is passed as argument, the call to `mixin` will copy all the properties from `Emitter.prototype` to the object, resulting in the object satisfying the required type (lines 545–550). However, if, for example, the argument is the value `true`, then that value is returned, and its type is not `Emitter` but `boolean`.

```
542 function Emitter(obj) {
543     if (obj) return mixin(obj);
544 }
545 function mixin(obj) {
546     for (var key in Emitter.prototype) {
547         obj[key] = Emitter.prototype[key];
548     }
549     return obj;
550 }
```

We fix the error by changing the parameter type `any` to `object`, after which ReaGenT no longer reports any error for the `Emitter` function.

Repeating the RQ1 experiment using the configuration where all three optional assumptions are disabled reveals no additional true positives, which indicates that the configuration used above is not overly restrictive. However, additional false positives appear in three of the libraries when all the optional assumptions are disabled. The impact of the individual optional assumptions is studied for RQ2 below.

In summary, our answer to RQ1 is affirmative. ReaGenT is able to find real errors, and without an overwhelming amount of false positives. Unlike all other tools that have been developed to detect mismatches between JavaScript libraries and TypeScript declaration files, the use of the RMGC allows ReaGenT to ensure that under the chosen set of assumptions, no additional type violations exist in these libraries.

RQ2: What are the effects of the optional assumptions?

We evaluate ReaGenT on the 10 JavaScript libraries using 5 different configurations: one with all optional assumptions enabled, three with a single assumption disabled, and one with all the assumptions disabled. (Each optional assumption could in principle be enabled or disabled for individual

functions, however, for simplicity we either enable or disable each assumption for all the library functions together.) For each library and analysis configuration, we measure the action coverage (Section 53) and the number of type violations reported. The results are shown in Table 7.2. We write *timeout* if the analysis has not terminated within one hour. Type violation reports may have the same root cause; the numbers shown here are without any attempt at deduplication.

When all assumptions are enabled we get 100% action coverage on all but two libraries. For `component-emitter` the lacking action coverage is caused by an error in the declaration file (demonstrated in Example 5). This error causes TAJS to have a catastrophic loss of precision, however, once the error is fixed TAJS runs successfully and ReaGenT reaches 100% action coverage. The lacking action coverage in `mime` is caused by a type violation that causes most of the library to be unreachable for the abstract RMGC. After fixing the error, we obtain 100% action coverage also for this library.

Disabling assumptions causes ReaGenT to report more violations for some libraries, however, manually inspecting the reports shows that they are all false positives. Note that disabling assumptions makes the RMGC become "more general", which may increase the ability to detect type violations, but it also increases the risk of timeouts and suboptimal action coverage and thereby fewer violations being reported.

Disabling WIDTH-SUBTYPING causes massive losses of precision in six of the libraries, resulting in either a timeout or a loss of action coverage. The precision loss typically comes from TAJS reading an undeclared property on a client-constructed value, causing an avalanche of spurious dataflow.

Disabling NO-PROTOTYPE-STRINGS only changes the results for two libraries. We see extra false positives for `loglevel` and `pathjs`.

Disabling PREFER-LIB-VALUES makes no significant difference for the 10 benchmarks, however, we know that disabling this assumption can cause false positives in other libraries as shown in Example 1.

Disabling all three assumptions causes a catastrophic loss of precision for most libraries, and ReaGenT only terminates successfully on three of the libraries.

We can from these results conclude that the WIDTH-SUBTYPING assumption is critical for precision for most libraries, NO-PROTOTYPE-STRINGS improves precision in some cases, PREFER-LIB-VALUES makes no difference for these benchmarks, and no additional true positives are found when disabling the assumptions. This suggests that enabling all the assumptions seems to be a reasonable default configuration.

7.8 Evaluation on larger benchmarks

This section is new to this thesis and contains an evaluation on a set of larger benchmarks.

Table 7.3 is a table similar to Table 7.2. An extra column showing how many lines of code are in the JavaScript implementation and TypeScript declaration file has been added. The benchmarks used in Table 7.3 are a mix of smaller and larger realistic JavaScript libraries.

The main conclusion is that our approach does not scale well to larger libraries. Even with all assumptions enabled, far most of the libraries either get terrible action coverage or reaches a timeout. For the benchmarks with an extremely low action coverage, the dominating reason for the action coverage being low is that the analysis encounters some mismatch between the JavaScript implementation and the TypeScript declaration file that prevents the analysis from continuing.

We also tried to fix some of the TypeScript declaration files in order to overcome the extremely low action coverage, however, doing so mostly resulted in the analysis timing out once a non-trivial action coverage was achieved.

As opposed to the original set of benchmarks in Table 7.2, the PREFER-LIB-VALUES assumption makes a difference on some of the benchmarks. For two of the benchmarks disabling PREFER-LIB-VALUES allows the analysis the get a higher action coverage, and for one benchmark disabling PREFER-LIB-VALUES results in the analysis timing out. This result confirms our results from the previous section that enabling assumptions can cause the number of reported violations to decrease while disabling them can cause the performance to degrade.

Table 7.3: Comparison of RMGC variants across a larger set of benchmarks.

Library	LOC JS/TS	all assumptions enabled	no-width-subtyping disabled	no-prototype-strings disabled	prefer-lib-values disabled	all three disabled
`accounting.js`	191/51	100.0% / 6	*timeout* / 0	100.0% / 6	100.0% / 6	*timeout* / 0
`async`	1 733/202	*timeout* / 89	*timeout* / 301	*timeout* / 89	*timeout* / 89	*timeout* / 301
`axios`	840/99	0,1% / 13	0,1% / 13	0,1% / 13	0,1% / 13	0,1% / 13
`bluebird`	4 939/195	0,0% / 1	0,0% / 1	0,0% / 1	0,0% / 1	0,0% / 1
`box2dweb`	10 718/1139	0,0% / 4	0,0% / 4	0,0% / 4	0,0% / 4	0,0% / 4
`CodeMirror`	7 302/402	0,1% / 2	0,1% / 2	0,1% / 2	0,1% / 2	0,1% / 2
`component-emitter`	72/13	96.0% / 72	*timeout* / 21	96.0% / 72	96.0% / 72	*timeout* / 73
`CreateJS`	8 955/1 386	*timeout* / 104	*timeout* / 104	*timeout* / 104	*timeout* / 104	*timeout* / 104
`Hammer.js`	1 509/265	1,0% / 128	1,0% / 128	1,0% / 128	1,0% / 128	1,0% / 128
`Handlebars`	3 444/241	0,5% / 9	0,5% / 9	0,5% / 9	0,5% / 9	0,5% / 9
`highlight.js`	783/128	0,6% / 21	0,6% / 21	0,6% / 21	0,6% / 21	0,6% / 21
`Intro.js`	1 156/69	*timeout* / 5	*timeout* / 7	*timeout* / 5	*timeout* / 5	*timeout* / 7
`Knockout`	4 346/412	0,1% / 7	0,1% / 7	0,1% / 7	0,1% / 7	0,1% / 7
`Medium Editor`	5 211/140	1,0% / 32	1,0% / 32	1,0% / 32	1,0% / 32	1,0% / 32
`Moment.js`	3 244/501	0,1% / 1	0,1% / 1	0,1% / 1	0,1% / 1	0,1% / 1
`PeerJS`	2 240/86	0,8% / 1	0,8% / 1	0,8% / 1	0,8% / 1	0,8% / 1
`PhotoSwipe`	2 602/146	1,4% / 1	1,4% / 1	1,4% / 1	2,1% / 29	*timeout* / 116
`QUnit`	3 038/109	0,5% / 19	0,5% / 19	0,5% / 19	0,5% / 19	0,5% / 19
`reveal.js`	2 612/108	86,0% / 36	86,0% / 36	86,0% / 36	*timeout* / 46	*timeout* / 56
`RxJS`	9 281/1 005	0,0% / 47	0,0% / 47	0,0% / 47	0,0% / 47	0,0% / 47
`semver`	979/74	23,2% / 7	23,2% / 7	23,2% / 7	36,0% / 9	*timeout* / 0
`Sortable`	879/76	1,6% / 3	1,6% / 3	1,6% / 3	1,6% / 3	1,6% / 3
`Zepto.js`	1 298/336	0,5% / 9	0,5% / 9	0,5% / 9	0,5% / 9	0,5% / 9

Threats to validity The following circumstances may affect our conclusions. Although we have selected the 10 libraries randomly, they may not be representative. TAJS is not fully sound, which may cause ReaGenT to miss errors (see Section 7.9). We have not conducted a user study to evaluate whether the error reports generated by ReaGenT are also actionable to others, and our fixes to the erroneous declaration files have not (yet) been confirmed by the library developers.

7.9 Related Work

Open-world analysis Many static analyses require whole programs to work, and developing useful modular analysis techniques has been a challenge for decades [38]. The idea of using most-general clients (also called most-general applications) when statically analyzing the possible behaviors of libraries appears often in the static analysis literature. One example is the modular static analysis by Rinetzky et al. [95] for reasoning about heap structures; another is the points-to analysis for Java libraries by Allen et al. [18]. To the best of our knowledge, none of the existing techniques work for dynamic languages like JavaScript.

The tool Averroes [17] is able to analyze an application without using the implementation of the library. This is done by creating a placeholder library that is similar to a most-general client but with the roles swapped.

Flow [33] is similarly to TypeScript a typed extension of JavaScript. Flow is based on static analysis, like ReaGenT, but obtains modularity by relying on type annotations, not only at the library interface but also inside the library code. The static analysis in Flow has been developed as a compromise between soundness and completeness. As an example, Flow does not detect any type error in the following simple program, where the `foo` function possibly returns a string at run-time instead of a number as expected from the type declaration.

```
551 var obj = { f: "this is a string, not a number" }
552 function foo(obj: typeof obj, s: string) : number {
553   return obj[s];
554 }
```

Flow programs can use JavaScript libraries via type declaration files, much like in TypeScript.

Sound whole-program analysis for JavaScript A lot of research has been done on how to perform sound static analysis for JavaScript, and significant progress has been made in recent years on making such analysis scale to real-world JavaScript programs. Among the recent work are TAJS [20],

WALA [70], SAFE [86], and JSAI [63]. All these analyzers share that they are "soundy" [75], meaning that they are sound in most realistic cases but rely on assumptions in specific corner cases that cause the analysis to be unsound in general. For example, the TAJS tool we use for ReaGenT does not fully model all standard library functions,[16] but we believe those limitations are insignificant for the experimental evaluation of ReaGenT.

Detecting errors in TypeScript declaration files Multiple approaches have been developed for detecting errors in TypeScript declaration files. TPD [117] finds errors by adding type contracts to existing library unit tests. These type contracts then check that the values observed during execution match the declared types. The approach in TPD is thus a variant of gradual typing [102], which has also been applied to general TypeScript code [91].

Like TPD, TStest [68] uses concrete executions with run-time type checks to detect errors in TypeScript declaration files. However, TStest explores the library using automated testing instead of relying on existing unit tests. TStest is similar to ReaGenT in that it attempts to find type violations in a TypeScript declaration file by simulating a client, but it lacks the exhaustiveness that characterizes ReaGenT.[17]

TScheck [47] is the only previous work that uses static analysis to find errors in type declaration files. Being based on a fast unsound static analysis and only detecting errors that manifest as likely mismatches at function return types, it provides no guarantees that all errors are found, unlike ReaGenT.

7.10 Conclusion

We have shown how the concept of a reasonably-most-general client (RMGC) enables static analysis of JavaScript libraries to detect mismatches between the library code and the TypeScript declaration files. An RMGC works under a set of assumptions that reflect how realistic clients may behave. Imposing too few or too many assumptions can result in false positives or false negatives, respectively. We have proposed five specific assumptions, some of which are necessary to obtain any meaningful results, and others can be configured by the analysis user.

Experiments with our proof-of-concept implementation ReaGenT that builds on the existing static analyzer TAJS demonstrates that the approach works, at least for small libraries that are within reach of TAJS. ReaGenT finds real type mismatches without an overwhelming amount of false pos-

[16] See `https://github.com/cs-au-dk/TAJS/issues/8`.

[17] Running TStest on the same benchmarks confirms that it finds a strict subset of the errors detected by ReaGenT.

itives. By design, it explores the library code exhaustively, relative to the RMGC assumptions, unlike all existing alternatives.

In addition to improving the quality of TypeScript declaration files, we believe this work may also guide further development of TAJS and related JavaScript static analyzers.

Acknowledgments We are grateful to Gianluca Mezzetti for his contributions to the early phases of this research. This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647544).

Bibliography

[1] Standard ecma-262. URL https://www.ecma-international.org/publications/standards/Ecma-262.htm. 11

[2] Ecmascript language specification, 3rd edition. URL https://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf. 11

[3] Ecmascript 2018 language specification. URL https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf. 11

[4] Projects | the state of the octoverse. URL https://octoverse.github.com/projects#languages. 3, 7

[5] A brief history of javascript. URL https://www.youtube.com/watch?v=GxouWy-ZE80. 11

[6] Kleene fixed-point theorem. URL https://en.wikipedia.org/wiki/Kleene_fixed-point_theorem. 48

[7] Stack overflow developer survey 2019, . URL https://insights.stackoverflow.com/survey/2019#most-loved-dreaded-and-wanted. 3, 7

[8] The state of javascript, . URL https://www.infoq.com/presentations/State-JavaScript/. 13, 17, 18

[9] Strict null checking the visual studio code codebase. URL https://code.visualstudio.com/blogs/2019/05/23/strict-null. 27

[10] Documentation · typescript. URL https://www.typescriptlang.org/docs/home.html. 19

[11] Typescripts type system is turing complete. URL https://github.com/Microsoft/TypeScript/issues/14833. 29

[12] Babel · the compiler for next generation javascript. URL https://babeljs.io/. 12

[13] Polyfill - mdn web docs glossary: Definitions of web-related terms | mdn. URL https://developer.mozilla.org/en-US/docs/Glossary/Polyfill. 12

[14] DefinitelyTyped, May 2019. http://definitelytyped.org. 3, 120

[15] typeshed, May 2019. https://github.com/python/typeshed. 3

[16] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Analyzing test completeness for dynamic languages. In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 142–153. ACM, 2016. doi: 10.1145/2931037.2931059. URL https://doi.org/10.1145/2931037.2931059. 7

[17] Karim Ali and Ondrej Lhoták. Averroes: Whole-program analysis without the whole program. In Giuseppe Castagna, editor, *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, volume 7920 of *Lecture Notes in Computer Science*, pages 378–400. Springer, 2013. doi: 10.1007/978-3-642-39038-8_16. URL https://doi.org/10.1007/978-3-642-39038-8_16. 144

[18] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. Combining type-analysis with points-to analysis for analyzing java library source-code. In Anders Møller and Mayur Naik, editors, *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2015, Portland, OR, USA, June 15 - 17, 2015*, pages 13–18. ACM, 2015. doi: 10.1145/2771284.2771287. URL https://doi.org/10.1145/2771284.2771287. 50, 121, 144

[19] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Copenhagen, 1994. 75

[20] Esben Andreasen and Anders Møller. Determinacy in static analysis for jquery. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 17–31. ACM, 2014. doi: 10.1145/2660193.2660214. URL https://doi.org/10.1145/2660193.2660214. 5, 49, 67, 75, 82, 120, 121, 130, 144

[21] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. Systematic approaches for increasing soundness and precision of static analyzers. In Karim Ali and Cristina Cifuentes, editors, *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2017, Barcelona, Spain, June 18, 2017*, pages 31–36. ACM, 2017. doi: 10.1145/3088515.3088521. URL `https://doi.org/10.1145/3088515.3088521`. 49, 135

[22] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of javascript web applications. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 571–580. ACM, 2011. doi: 10.1145/1985793.1985871. URL `https://doi.org/10.1145/1985793.1985871`. 115

[23] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269. ACM, 2014. doi: 10.1145/2594291.2594299. URL `https://doi.org/10.1145/2594291.2594299`. 5

[24] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In Lougie Anderson and James Coplien, editors, *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96), San Jose, California, USA, October 6-10, 1996.*, pages 324–341. ACM, 1996. doi: 10.1145/236337.236371. URL `https://doi.org/10.1145/236337.236371`. 5

[25] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. SAFEWAPI: web API misuse detector for web applications. In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 507–517. ACM, 2014. doi: 10.1145/2635868.2635916. URL `https://doi.org/10.1145/2635868.2635916`. 82

[26] Gavin M. Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In Richard E. Jones, editor, *ECOOP 2014 -*

Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings, volume 8586 of *Lecture Notes in Computer Science*, pages 257–281. Springer, 2014. doi: 10.1007/978-3-662-44202-9_11. URL https://doi.org/10.1007/978-3-662-44202-9_11. 19, 82, 102, 103, 125

[27] Garrett Birkhoff. *Lattice theory*, volume 25. American Mathematical Soc., 1940. 5, 45

[28] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical optional types for clojure. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 68–94. Springer, 2016. doi: 10.1007/978-3-662-49498-1_4. URL https://doi.org/10.1007/978-3-662-49498-1_4. 82

[29] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In Richard L. Wexelblat, editor, *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, Palo Alto, California, USA, June 25-27, 1986*, pages 152–161. ACM, 1986. doi: 10.1145/12276.13327. URL https://doi.org/10.1145/12276.13327. 130

[30] Oscar Callaú, Romain Robbes, Éric Tanter, and David Röthlisberger. How (and why) developers use the dynamic features of programming languages: the case of smalltalk. *Empirical Software Engineering*, 18(6): 1156–1194, 2013. doi: 10.1007/s10664-012-9203-2. URL https://doi.org/10.1007/s10664-012-9203-2. 3

[31] Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Young-Il Choi. Type inference for static compilation of javascript. In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 410–429. ACM, 2016. doi: 10.1145/2983990.2984017. URL https://doi.org/10.1145/2983990.2984017. 36, 82

[32] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In Bernard N. Fischer, editor, *Proceedings*

of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990, pages 296–310. ACM, 1990. doi: 10.1145/93542.93585. URL https://doi.org/10.1145/93542.93585. 130

[33] Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levi. Fast and precise type checking for javascript. *PACMPL*, 1(OOPSLA):48:1–48:30, 2017. doi: 10.1145/3133872. URL https://doi.org/10.1145/3133872. 20, 120, 144

[34] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*. USENIX Association, 2003. URL https://www.usenix.org/conference/12th-usenix-security-symposium/static-analysis-executables-detect-malicious-patterns. 5

[35] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 268–279. ACM, 2000. doi: 10.1145/351240.351266. URL https://doi.org/10.1145/351240.351266. 6, 51, 115

[36] James A. Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In David S. Rosenblum and Sebastian G. Elbaum, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, pages 196–206. ACM, 2007. doi: 10.1145/1273463.1273490. URL https://doi.org/10.1145/1273463.1273490. 6

[37] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi: 10.1145/512950.512973. URL https://doi.org/10.1145/512950.512973. 5

[38] Patrick Cousot and Radhia Cousot. Modular static program analysis. In R. Nigel Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on*

Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings, volume 2304 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2002. doi: 10.1007/3-540-45937-5_13. URL https://doi.org/10.1007/3-540-45937-5_13. 144

[39] Douglas Crockford. *JavaScript: The Good Parts: The Good Parts*. " O'Reilly Media, Inc.", 2008. 12

[40] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Walter G. Olthoff, editor, *ECOOP'95 - Object-Oriented Programming, 9th European Conference, Århus, Denmark, August 7-11, 1995, Proceedings*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101. Springer, 1995. doi: 10.1007/3-540-49538-X_5. URL https://doi.org/10.1007/3-540-49538-X_5. 5, 12, 48

[41] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In Michael Burke and Mary Lou Soffa, editors, *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 59–69. ACM, 2001. doi: 10.1145/378795.378811. URL https://doi.org/10.1145/378795.378811. 4

[42] Edsger W. Dijkstra. Notes on structured programming. Technical Report EWD249, Technological University Eindhoven, 1970. 102

[43] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa, editors, *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, pages 242–256. ACM, 1994. doi: 10.1145/178243.178264. URL https://doi.org/10.1145/178243.178264. 136

[44] Facebook. Flow, 2016. http://flowtype.org/. 40, 82, 86, 115

[45] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In Ron Crocker and Guy L. Steele Jr., editors, *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, pages 302–312. ACM, 2003. doi: 10.1145/949305.949332. URL https://doi.org/10.1145/949305.949332. 5

[46] Asger Feldthaus and Anders Møller. Semi-automatic rename refactoring for javascript. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 323–338. ACM, 2013. doi: 10.1145/2509136.2509520. URL `https://doi.org/10.1145/2509136.2509520`. 6

[47] Asger Feldthaus and Anders Møller. Checking correctness of typescript interfaces for javascript libraries. In Andrew P. Black and Todd D. Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 1–16. ACM, 2014. doi: 10.1145/2660193.2660215. URL `https://doi.org/10.1145/2660193.2660215`. 6, 36, 39, 62, 63, 67, 82, 84, 96, 106, 107, 112, 113, 114, 120, 137, 145

[48] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In Monica S. Lam, editor, *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*, pages 219–232. ACM, 2000. doi: 10.1145/349299.349328. URL `https://doi.org/10.1145/349299.349328`. 4

[49] Gordon Fraser and Andrea Arcuri. Automated test generation for java generics. In Dietmar Winkler, Stefan Biffl, and Johannes Bergsmann, editors, *Software Quality. Model-Based Approaches for Advanced Software and Systems Engineering - 6th International Conference, SWQD 2014, Vienna, Austria, January 14-16, 2014. Proceedings*, volume 166 of *Lecture Notes in Business Information Processing*, pages 185–198. Springer, 2014. doi: 10.1007/978-3-319-03602-1_12. URL `https://doi.org/10.1007/978-3-319-03602-1_12`. 101, 115

[50] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael W. Hicks. Static type inference for ruby. In Sung Y. Shin and Sascha Ossowski, editors, *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*, pages 1859–1866. ACM, 2009. doi: 10.1145/1529282.1529700. URL `https://doi.org/10.1145/1529282.1529700`. 5

[51] Zheng Gao, Christian Bird, and Earl T. Barr. To type or not to type: quantifying detectable bugs in javascript. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings*

of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, pages 758–769. IEEE / ACM, 2017. doi: 10.1109/ICSE.2017.75. URL https://doi.org/10.1109/ICSE.2017.75. 4

[52] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *ACM Queue*, 10(1):20, 2012. doi: 10.1145/2090147.2094081. URL https://doi.org/10.1145/2090147.2094081. 6

[53] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Shooting from the heap: ultra-scalable static analysis with heap snapshots. In Frank Tip and Eric Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 198–208. ACM, 2018. doi: 10.1145/3213846.3213860. URL https://doi.org/10.1145/3213846.3213860. 6

[54] Phillip Heidegger and Peter Thiemann. Contract-driven testing of javascript code. In Jan Vitek, editor, *Objects, Models, Components, Patterns, 48th International Conference, TOOLS 2010, Málaga, Spain, June 28 - July 2, 2010. Proceedings*, volume 6141 of *Lecture Notes in Computer Science*, pages 154–172. Springer, 2010. doi: 10.1007/978-3-642-13953-6_9. URL https://doi.org/10.1007/978-3-642-13953-6_9. 114

[55] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969. 5, 36

[56] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In Tadayoshi Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 445–458. USENIX Association, 2012. URL https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler. 6

[57] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In Jens Palsberg and Zhendong Su, editors, *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2009. doi: 10.1007/978-3-642-03237-0_17. URL https://doi.org/10.1007/978-3-642-03237-0_17. 5, 44, 49, 82, 114, 120, 130

[58] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In Radhia Cousot and Matthieu Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 320–339. Springer, 2010. doi: 10.1007/978-3-642-15769-1_20. URL `https://doi.org/10.1007/978-3-642-15769-1_20`. 49

[59] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In Mats Per Erik Heimdahl and Zhendong Su, editors, *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 34–44. ACM, 2012. doi: 10.1145/2338965.2336758. URL `https://doi.org/10.1145/2338965.2336758`. 17, 49

[60] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003. 36

[61] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317, 1977. doi: 10.1007/BF00290339. URL `https://doi.org/10.1007/BF00290339`. 5, 45

[62] Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. Type refinement for static analysis of javascript. In Antony L. Hosking, Patrick Th. Eugster, and Carl Friedrich Bolz, editors, *DLS'13, Proceedings of the 9th Symposium on Dynamic Languages, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 17–26. ACM, 2013. doi: 10.1145/2508168.2508175. URL `https://doi.org/10.1145/2508168.2508175`. 132

[63] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: a static analysis platform for javascript. In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 121–132. ACM, 2014. doi: 10.1145/2635868.2635904. URL `https://doi.org/10.1145/2635868.2635904`. 5, 44, 120, 130, 136, 145

[64] Matthias Keil and Peter Thiemann. Treatjs: Higher-order contracts for javascripts. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 28–51. Schloss Dagstuhl -

Leibniz-Zentrum fuer Informatik, 2015. doi: 10.4230/LIPIcs.ECOOP.2015.28. URL https://doi.org/10.4230/LIPIcs.ECOOP.2015.28. 84

[65] Matthias Keil and Peter Thiemann. Blame assignment for higher-order contracts with intersection and union. In Kathleen Fisher and John H. Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 375–386. ACM, 2015. doi: 10.1145/2784731.2784737. URL https://doi.org/10.1145/2784731.2784737. 99

[66] Gary A. Kildall. A unified approach to global program optimization. In Patrick C. Fischer and Jeffrey D. Ullman, editors, *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, pages 194–206. ACM Press, 1973. doi: 10.1145/512927.512945. URL https://doi.org/10.1145/512927.512945. 5, 44

[67] Erik Krogh Kristensen and Anders Møller. Inference and evolution of typescript declaration files. In Marieke Huisman and Julia Rubin, editors, *Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10202 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2017. doi: 10.1007/978-3-662-54494-5_6. URL https://doi.org/10.1007/978-3-662-54494-5_6. 6, 84, 96, 106, 107, 112, 113, 114, 120

[68] Erik Krogh Kristensen and Anders Møller. Type test scripts for typescript testing. *PACMPL*, 1(OOPSLA):90:1–90:25, 2017. doi: 10.1145/3133914. URL https://doi.org/10.1145/3133914. 6, 120, 137, 145

[69] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 216–226. ACM, 2014. doi: 10.1145/2594291.2594334. URL https://doi.org/10.1145/2594291.2594334. 6

[70] Sungho Lee, Julian Dolby, and Sukyoung Ryu. Hybridroid: static analysis framework for android hybrid applications. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 250–261. ACM, 2016. doi:

10.1145/2970276.2970368. URL https://doi.org/10.1145/2970276.2970368. 145

[71] Jukka Lehtosalo et al. Mypy, 2016. http://www.mypy-lang.org/. 82, 115

[72] Benjamin S. Lerner, Joe Gibbs Politz, Arjun Guha, and Shriram Krishnamurthi. Tejas: retrofitting type systems for javascript. In Antony L. Hosking, Patrick Th. Eugster, and Carl Friedrich Bolz, editors, *DLS'13, Proceedings of the 9th Symposium on Dynamic Languages, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 1–16. ACM, 2013. doi: 10.1145/2508168.2508170. URL https://doi.org/10.1145/2508168.2508170. 82

[73] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The objective caml system–documentation and user's manual, 2002. 36

[74] Guodong Li, Esben Andreasen, and Indradeep Ghosh. Symjs: automatic symbolic testing of javascript web applications. In Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey, editors, *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 449–459. ACM, 2014. doi: 10.1145/2635868.2635913. URL https://doi.org/10.1145/2635868.2635913. 115

[75] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 58(2):44–46, 2015. doi: 10.1145/2644805. URL https://doi.org/10.1145/2644805. 6, 44, 135, 145

[76] Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of javascript applications in the presence of frameworks and libraries. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 499–509. ACM, 2013. doi: 10.1145/2491411.2491417. URL https://doi.org/10.1145/2491411.2491417. 121

[77] Yukihiro 'Matz' Matsumoto. RubyConf 2014 – opening keynote, 2014. http://confreaks.tv/videos/rubyconf2014-opening-keynote. 82

[78] Microsoft. Typescript 2.0, 2016. URL https://www.typescriptlang.org/docs/handbook/release-notes/typescript-2-0.html. 5

[79] Microsoft. TypeScript language specification, February 2016. https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md. 19, 22, 61, 84

[80] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978. doi: 10.1016/0022-0000(78)90014-4. URL https://doi.org/10.1016/0022-0000(78)90014-4. 5, 36

[81] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of standard ML: revised*. MIT press, 1997. 36

[82] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 75–84. IEEE Computer Society, 2007. doi: 10.1109/ICSE.2007.37. URL https://doi.org/10.1109/ICSE.2007.37. 6, 51, 85, 89, 90, 115

[83] Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In Joseph S. Sventek and Steven Hand, editors, *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, pages 247–260. ACM, 2008. doi: 10.1145/1352592.1352618. URL https://doi.org/10.1145/1352592.1352618. 82

[84] Changhee Park and Sukyoung Ryu. Scalable and precise static analysis of javascript applications via loop-sensitivity. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 735–756. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi: 10.4230/LIPIcs.ECOOP.2015.735. URL https://doi.org/10.4230/LIPIcs.ECOOP.2015.735. 44

[85] Changhee Park, Hongki Lee, and Sukyoung Ryu. Static analysis of javascript libraries in a scalable and precise way using loop sensitivity. *Softw., Pract. Exper.*, 48(4):911–944, 2018. doi: 10.1002/spe.2552. URL https://doi.org/10.1002/spe.2552. 135

[86] Joonyoung Park, Inho Lim, and Sukyoung Ryu. Battles with false positives in static analysis of javascript web applications in the wild. In Laura K. Dillon, Willem Visser, and Laurie Williams, editors,

Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume, pages 61–70. ACM, 2016. doi: 10.1145/2889160.2889227. URL https://doi.org/10.1145/2889160.2889227. 120, 121, 130, 136, 145

[87] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8. 40

[88] François Pottier. A framework for type inference with subtyping. In Matthias Felleisen, Paul Hudak, and Christian Queinnec, editors, *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998.*, pages 228–238. ACM, 1998. doi: 10.1145/289423.289448. URL https://doi.org/10.1145/289423.289448. 40, 68, 82

[89] Michael Pradel, Parker Schuh, and Koushik Sen. Typedevil: Dynamic type inconsistency analysis for javascript. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 314–324. IEEE Computer Society, 2015. doi: 10.1109/ICSE.2015.51. URL https://doi.org/10.1109/ICSE.2015.51. 114

[90] Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. The ins and outs of gradual type inference. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 481–494. ACM, 2012. doi: 10.1145/2103656.2103714. URL https://doi.org/10.1145/2103656.2103714. 82

[91] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 167–180. ACM, 2015. doi: 10.1145/2676726.2676971. URL https://doi.org/10.1145/2676726.2676971. 82, 84, 114, 145

[92] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*,

pages 1–12. ACM, 2010. doi: 10.1145/1806596.1806598. URL https://doi.org/10.1145/1806596.1806598. 121

[93] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do - A large-scale study of the use of eval in javascript applications. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, volume 6813 of *Lecture Notes in Computer Science*, pages 52–78. Springer, 2011. doi: 10.1007/978-3-642-22655-7_4. URL https://doi.org/10.1007/978-3-642-22655-7_4. 17

[94] Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete types for typescript. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 76–100. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi: 10.4230/LIPIcs.ECOOP.2015.76. URL https://doi.org/10.4230/LIPIcs.ECOOP.2015.76. 82

[95] Noam Rinetzky, Arnd Poetzsch-Heffter, Ganesan Ramalingam, Mooly Sagiv, and Eran Yahav. Modular shape analysis for dynamically encapsulated programs. In Rocco De Nicola, editor, *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2007. doi: 10.1007/978-3-540-71316-6_16. URL https://doi.org/10.1007/978-3-540-71316-6_16. 50, 121, 144

[96] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):58–66, 2018. doi: 10.1145/3188720. URL https://doi.org/10.1145/3188720. 6

[97] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 513–528. IEEE Computer Society, 2010. doi: 10.1109/SP.2010.38. URL https://doi.org/10.1109/SP.2010.38. 115

[98] Koushik Sen. DART: directed automated random testing. In Kedar S. Namjoshi, Andreas Zeller, and Avi Ziv, editors, *Hardware and Software: Verification and Testing - 5th International Haifa Verification Con-*

ference, HVC 2009, Haifa, Israel, October 19-22, 2009, Revised Selected Papers, volume 6405 of *Lecture Notes in Computer Science*, page 4. Springer, 2009. doi: 10.1007/978-3-642-19237-1_4. URL https://doi.org/10.1007/978-3-642-19237-1_4. 104, 110, 115

[99] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272. ACM, 2005. doi: 10.1145/1081706.1081750. URL https://doi.org/10.1145/1081706.1081750. 95, 115

[100] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 488–498. ACM, 2013. doi: 10.1145/2491411.2491447. URL https://doi.org/10.1145/2491411.2491447. 6, 105, 115

[101] Olin Shivers. Control-flow analysis in scheme. In Richard L. Wexelblat, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, pages 164–174. ACM, 1988. doi: 10.1145/53990.54007. URL https://doi.org/10.1145/53990.54007. 130

[102] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006. 4, 84, 96, 145

[103] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, volume 32 of *LIPIcs*, pages 274–293. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi: 10.4230/LIPIcs.SNAPL.2015.274. URL https://doi.org/10.4230/LIPIcs.SNAPL.2015.274. 4, 96

[104] Robert I. Soare. *Recursively enumerable sets and degrees - a study of computable functions and computability generated sets*. Perspectives in mathematical logic. Springer, 1987. ISBN 978-3-540-15299-6. 5

[105] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. Correlation tracking for points-to analysis of javascript. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 435–458. Springer, 2012. doi: 10.1007/978-3-642-31057-7_20. URL https://doi.org/10.1007/978-3-642-31057-7_20. 44, 75

[106] Bjarne Steensgaard. Points-to analysis in almost linear time. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 32–41. ACM Press, 1996. doi: 10.1145/237721.237727. URL https://doi.org/10.1145/237721.237727. 5, 75

[107] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient dynamic analysis for node.js. In Christophe Dubach and Jingling Xue, editors, *Proceedings of the 27th International Conference on Compiler Construction, CC 2018, February 24-25, 2018, Vienna, Austria*, pages 196–206. ACM, 2018. doi: 10.1145/3178372.3179527. URL https://doi.org/10.1145/3178372.3179527. 6

[108] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975. doi: 10.1145/321879.321884. URL https://doi.org/10.1145/321879.321884. 38

[109] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 395–406. ACM, 2008. doi: 10.1145/1328438.1328486. URL https://doi.org/10.1145/1328438.1328486. 82, 115

[110] Omer Tripp, Pietro Ferrara, and Marco Pistoia. Hybrid security analysis of web javascript code via dynamic partial evaluation. In Corina S. Pasareanu and Darko Marinov, editors, *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 49–59. ACM, 2014. doi: 10.1145/2610384.2610385. URL https://doi.org/10.1145/2610384.2610385. 6

[111] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Refinement types for typescript. In Chandra Krintz and Emery Berger, editors,

Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016, pages 310–325. ACM, 2016. doi: 10.1145/2908080.2908110. URL https://doi.org/10.1145/2908080.2908110. 82

[112] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for python. In Andrew P. Black and Laurence Tratt, editors, *DLS'14, Proceedings of the 10th ACM Symposium on Dynamic Languages, part of SLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 45–56. ACM, 2014. doi: 10.1145/2661088.2661101. URL https://doi.org/10.1145/2661088.2661101. 82, 115

[113] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4 (2/3):167–188, 1996. doi: 10.3233/JCS-1996-42-304. URL https://doi.org/10.3233/JCS-1996-42-304. 5

[114] Philip Wadler. Theorems for free! In Joseph E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989. doi: 10.1145/99370.99404. URL https://doi.org/10.1145/99370.99404. 100

[115] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2009. doi: 10.1007/978-3-642-00590-9_1. URL https://doi.org/10.1007/978-3-642-00590-9_1. 96, 98, 114

[116] Shiyi Wei and Barbara G. Ryder. Practical blended taint analysis for javascript. In Mauro Pezzè and Mark Harman, editors, *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 336–346. ACM, 2013. doi: 10.1145/2483760.2483788. URL https://doi.org/10.1145/2483760.2483788. 6

[117] Jack Williams, J. Garrett Morris, Philip Wadler, and Jakub Zalewski. Mixed messages: Measuring conformance and non-interference in typescript. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*,

volume 74 of *LIPIcs*, pages 28:1–28:29. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017. doi: 10.4230/LIPIcs.ECOOP.2017.28. URL `https://doi.org/10.4230/LIPIcs.ECOOP.2017.28`. 51, 52, 84, 100, 101, 107, 114, 120, 137, 145

[118] Yu Yuning. A study of object creators in JavaScript. Master's thesis, University of Waterloo, 2017. 127